Appunti di Ricerca Operativa

2014/2015

Prefazione

La Ricerca Operativa è un campo in continua evoluzione, il cui impatto sulle realtà aziendali ed organizzative è in costante crescita. L'insegnamento di questa disciplina, ed in particolare delle sue basi metodologiche ed algoritmiche, è quindi indispensabile nei corsi che mirano a formare molte figure con elevate capacità tecnologiche e manageriali, ad esempio—ma non solamente—nei corsi di Laurea in Informatica, Matematica, Ingegneria e materie affini.

Queste dispense sono state sviluppate dal Gruppo di Ricerca Operativa del Dipartimento di Informatica dell'Università di Pisa per il supporto a diversi corsi afferenti all'area della Ricerca Operativa, sia di base che avanzati, in molti diversi curricula della stessa Università, quali i Corsi di Laurea in Informatica, Informatica Applicata, Matematica, Ingegneria Gestionale, Ingegneria Elettronica, ed Ingegneria delle Telecomunicazioni, ed i Corsi di Laurea Magistrale in Informatica, Informatica e Networking, Informatica per l'Economia e per l'Azienda, ed Ingegneria Gestionale. Inoltre, le dispense sono state adottate anche in Corsi di Laurea di altre Università italiane. Le dispense coprono sia gli aspetti di base del curriculum in Ricerca Operativa (problemi e modelli di ottimizzazione, Programmazione Lineare, algoritmi su grafi) che aspetti più avanzati, in particolare legati alla soluzione di problemi di Ottimizzazione Combinatoria $\mathcal{NP}$-hard. Ciò non esaurisce certamente lo spettro delle metodologie della Ricerca Operativa, né delle sue possibili applicazioni; in particolare non vengono affrontate le tematiche relative alla presenza di elementi nonlineari nei modelli di ottimizzazione, e non vengono discusse in dettaglio le molteplici possibili applicazioni delle metodologie descritte in campi quali i trasporti e la logistica, le telecomunicazioni, l'energia, l'economia e la finanza, la scienze fisiche e biologiche, l'informatica, e molti altri.

Queste dispense sono il frutto di un lavoro collettivo, svolto nel corso di molti anni da diverse persone, in forme e ruoli diversi. In particolare, hanno collaborato alla stesura di questo documento Giorgio Gallo, Stefano Pallottino, Maria Grazia Scutellà, Antonio Frangioni e Giancarlo Bigi. Un aiuto particolare alla stesura e al miglioramento delle dispense è stato dato da Paola Cappanera e Maria Paola Scaparra. Molte altre persone, tra cui molti studenti dei corsi di Ricerca Operativa all'interno dei corsi di Laurea e di Laurea Magistrale dell'Università di Pisa, hanno contribuito a queste dispense segnalando errori e suggerendo miglioramenti; a tutti loro va il ringraziamento degli estensori. Ogni errore ed imprecisione rimasta nel testo è esclusivamente responsabilità degli autori; segnalazioni a tal proposito sono caldamente benvenute.

L'utilizzo di questo materiale in corsi di studio diversi da quelli tenuti dagli estensori del documento è permesso ed incoraggiato, a condizione che sia opportunamente citata la fonte, che non venga tratto profitto dal fornire il materiale agli studenti, e che tale utilizzo venga segnalato agli autori. La modalità di distribuzione consigliata è quella di fare riferimento alla pagina web dei Corsi di Ricerca Operativa presso il Dipartimento di Informatica

`http://www.di.unipi.it/optimize/Courses`

in cui si trovano le versioni più aggiornate del testo, insieme ad altro materiale che può risultare utile per gli studenti. Esperti di Ricerca Operativa intenzionati a contribuire alle dispense, ferme restando le modalità di distribuzione, possono contattare gli estensori per accordarsi a tal proposito.

Pisa, 21/09/2014

Indice

1 Problemi e Modelli **1**
1.1 Modelli e Problemi . . . 2
1.2 Tecniche di Modellazione . . . 7
1.2.1 Programmazione Lineare . . . 8
1.2.2 Variabili logiche . . . 11
1.2.3 Relazioni binarie . . . 15
1.2.4 Vincoli di assegnamento e semiassegnamento . . . 16
1.2.5 Selezione di sottoinsiemi . . . 20
1.2.6 Variabili a valori discreti . . . 22
1.2.7 Minima quantità positiva prefissata . . . 25
1.2.8 Funzione con carico fisso . . . 25
1.2.9 Vincoli di soglia . . . 26
1.2.10 Come rappresentare il valore assoluto . . . 27
1.2.11 Funzioni lineari a tratti . . . 28
1.2.12 Vincoli disgiuntivi . . . 30
1.2.13 Un esempio di formulazione e alcuni esercizi . . . 32

2 Programmazione Lineare **37**
2.1 Problemi di Programmazione Lineare . . . 37
2.1.1 Geometria della Programmazione Lineare . . . 40
2.2 Teoria della Dualità . . . 51
2.2.1 Coppie di problemi duali . . . 51
2.2.2 Il teorema debole della dualità . . . 54
2.2.3 Il teorema forte della dualità e sue conseguenze . . . 56
2.2.4 Il teorema degli scarti complementari . . . 59
2.2.5 Soluzioni complementari e basi . . . 62
2.3 Algoritmi del Simplesso . . . 64
2.3.1 L'algoritmo del Simplesso Primale . . . 65
2.3.2 L'algoritmo del Simplesso Duale . . . 75
2.3.3 Analisi post-ottimale . . . 81

3 Grafi e reti di flusso **87**
3.1 Flussi su reti . . . 87
3.1.1 Alcuni modelli di flusso . . . 89
3.1.2 Trasformazioni equivalenti . . . 90
3.1.3 Algoritmi del Simplesso per (MCF) . . . 91
3.2 Cammini di costo minimo . . . 95
3.2.1 Il problema . . . 96
3.2.2 Alberi, etichette e condizioni di ottimo . . . 98
3.2.3 L'algoritmo *SPT* . . . 99
3.2.4 Algoritmi a coda di priorità . . . 101
3.2.5 Algoritmi a selezione su lista . . . 104

3.2.6 Cammini minimi su grafi aciclici . . . 107
3.2.7 Cammini minimi con radici multiple . . . 107
3.3 Il problema di flusso massimo . . . 108
3.3.1 Tagli, cammini aumentanti e condizioni di ottimo . . . 109
3.3.2 Algoritmo per cammini aumentanti . . . 111
3.3.3 Flusso massimo con più sorgenti/pozzi . . . 114
3.3.4 Algoritmo basato su preflussi . . . 115
3.4 Il problema del Flusso di Costo Minimo . . . 118
3.4.1 Cammini, cicli aumentanti e condizioni di ottimo . . . 118
3.4.2 Algoritmo basato su cancellazione di cicli . . . 120
3.4.3 Algoritmo basato su cammini minimi successivi . . . 122
3.5 Problemi di accoppiamento . . . 126
3.5.1 Accoppiamento di massima cardinalità . . . 127
3.5.2 Assegnamento di costo minimo . . . 129
3.5.3 Accoppiamento di massima cardinalità bottleneck . . . 131

4 Ottimizzazione Combinatoria **135**
4.1 Introduzione . . . 135
4.2 Programmazione Lineare Intera (Mista) . . . 136
4.2.1 Il rilassamento continuo . . . 137
4.2.2 Formulazioni di PL equivalenti per la PLI . . . 138
4.2.3 Diseguaglianze valide . . . 139
4.3 Dimostrazioni di ottimalità . . . 141

5 Algoritmi euristici **143**
5.1 Algoritmi greedy . . . 143
5.1.1 Esempi di algoritmi greedy . . . 144
5.1.2 Algoritmi greedy con garanzia sulle prestazioni . . . 151
5.1.3 Matroidi . . . 158
5.2 Algoritmi di ricerca locale . . . 161
5.2.1 Esempi di algoritmi di ricerca locale . . . 163
5.2.2 Intorni di grande dimensione . . . 169
5.2.3 Metaeuristiche . . . 171

6 Tecniche di rilassamento **181**
6.1 Rilassamento continuo . . . 182
6.1.1 Efficacia del rilassamento continuo . . . 183
6.1.2 Informazione generata dal rilassamento continuo . . . 185
6.2 Eliminazione di vincoli . . . 188
6.2.1 Esempi di rilassamenti per eliminazione di vincoli . . . 188
6.3 Rilassamento Lagrangiano . . . 192
6.3.1 Teoria del rilassamento Lagrangiano . . . 194
6.3.2 Algoritmi per il rilassamento Lagrangiano . . . 198
6.3.3 Informazione generata dal rilassamento Lagrangiano . . . 203
6.4 Rilassamento surrogato . . . 205

7 Algoritmi enumerativi **207**
7.1 Algoritmi di enumerazione implicita . . . 207
7.2 Implementare un algoritmo enumerativo . . . 214
7.2.1 Rilassamento ed euristica . . . 214
7.2.2 La strategia di visita . . . 216
7.2.3 Regole di dominanza . . . 217
7.2.4 Regole di branching . . . 218

7.2.5 Preprocessing . . . 220
7.3 Esempi di algoritmi enumerativi . . . 222
7.3.1 Il problema dello zaino . . . 222
7.3.2 Il problema del commesso viaggiatore . . . 225
7.3.3 Il problema del cammino minimo vincolato . . . 226
7.4 Programmazione dinamica . . . 228
7.5 Tecniche poliedrali . . . 228

A Algoritmi e complessità 229
A.1 Modelli computazionali . . . 229
A.2 Misure di complessità . . . 229
A.3 Problemi trattabili e problemi intrattabili . . . 230
A.3.1 Le classi $\mathcal{P}$ e $\mathcal{NP}$ . . . 230
A.3.2 Problemi $\mathcal{NP}$-completi e problemi $\mathcal{NP}$-ardui . . . 231
A.3.3 Complessità ed approssimazione . . . 231

B Grafi e Reti 233
B.1 I grafi: notazione e nomenclatura . . . 233
B.1.1 Grafi, nodi, archi . . . 233
B.1.2 Cammini, cicli . . . 234
B.1.3 Tagli e connettività . . . 235
B.1.4 Alberi . . . 236
B.2 Rappresentazione di grafi ed alberi . . . 236
B.2.1 Matrici di incidenza e liste di adiacenza . . . 236
B.2.2 Rappresentazione di alberi: la funzione predecessore . . . 238
B.2.3 Visite di un albero . . . 238
B.2.4 Livello dei nodi di un albero . . . 239
B.3 Visita di un grafo . . . 239
B.3.1 Implementazioni della procedura di visita . . . 239
B.3.2 Usi della procedura di visita . . . 241
B.4 Albero di copertura di costo minimo . . . 242
B.4.1 Algoritmo di Kruskal . . . 243
B.4.2 Algoritmo di Prim . . . 243
B.4.3 Albero di copertura bottleneck . . . 245

Capitolo 1

Problemi e Modelli

La *Ricerca Operativa* ha come oggetto lo studio e la messa a punto di metodologie per la soluzione di *problemi decisionali*. I problemi affrontati nell'ambito della Ricerca Operativa sono tipicamente quelli in cui bisogna prendere *decisioni* sull'uso di risorse disponibili in quantità limitata in modo da rispettare un insieme assegnato di condizioni (vincoli) e massimizzando il "beneficio" ottenibile dall'uso delle risorse stesse. La Ricerca Operativa considera quindi, in generale, tutte le metodologie utili a migliorare l'efficacia delle decisioni; ciò significa, in linea di principio, considerare tutte le fasi del *processo decisionale* che porta a prenderle. In modo molto schematico, queste fasi sono:

1) individuazione del problema;

2) analisi della realtà e raccolta dei dati;

3) costruzione del modello;

4) determinazione di una o più soluzioni;

5) analisi dei risultati ottenuti.

Questi punti non devono essere visti come strettamente sequenziali, in quanto ciascuno dei punti è fortemente correlato con quelli che lo precedono e seguono. La stessa raccolta dei dati presuppone un'idea (magari embrionale) sul tipo di modello che sarà costruito, e la scelta del modello è spesso funzionale alle esigenze della fase successiva, in quanto il modello deve ammettere approcci risolutivi in grado di determinare soluzioni in tempo compatibili con le esigenze del processo decisionale (se la decisione va presa entro domani, non posso aspettare una settimana per avere la risposta). Viceversa, in un processo decisionale reale è frequente il caso in cui una delle fasi richieda modifiche dei risultati ottenuti in una fase precedente. Ad esempio, nella costruzione del modello può emergere l'esigenza di nuovi dati in aggiunta a quelli già raccolti. Oppure, la determinazione delle soluzioni costringe a rivedere il modello in quanto il costo computazionale della sua soluzione si rivela essere troppo elevato. O ancora, l'analisi dei risultati ottenuti mostra che il modello non cattura in modo adeguato la realtà che dovrebbe rappresentare, e quindi porta alla sua modifica.

La Ricerca Operativa ha quindi sia un vastissimo ambito di applicazione, che, in linea di principio, la necessità di utilizzare o sviluppare metodologie molto diversificate che vanno dall'analisi dei sistemi produttivi e sociali, alla capacità di raccogliere dati statisticamente significativi (possibilmente estraendoli da una grande mole di dati scarsamente strutturati), fino allo sviluppo di metodologie per costruire modelli efficaci di porzioni della realtà che possano essere risolti efficientemente. Questi ultimi due aspetti, corrispondenti ai punti (3) e (4), sono quelli sui quali si concentrano queste dispense. La scelta è dettata dal fatto che la costruzione di modelli e la loro soluzione algoritmica sono le fasi che più si prestano ad una rigorosa trattazione matematica, e quindi quelle più adatte a studenti in materie tecniche e scientifiche. In ogni caso, a questi argomenti sono dirette gran parte delle metodologie messe a punto nell'ambito della Ricerca Operativa; il materiale qui presentato si situa in particolare nell'area di studio che prende il nome di *Programmazione Matematica*.

È opportuno sottolineare come queste dispense non presentino di certo un panorama esaustivo della Ricerca Operativa. In primo luogo, infatti, la Programmazione Matematica è solo una delle componenti, per quanto spesso necessaria ed importante, di quella che è più in generale la *Scienza delle Decisioni*. In secondo luogo, ovviamente, i contenuti di queste dispense riguardano solo alcuni tra i più basilari risultati del settore. L'enorme mole di ricerche svolte in questo campo ha determinato un corpus di risultati, peraltro in continua espansione, di cui queste note possono fornire solo un primissimo assaggio, cercando ove possibile di far intravedere *in nuce* tematiche che possono essere sviluppate in profondità ed ampiezza grandemente superiori. Infatti, queste dispense sono state pensate per corsi di base di Ricerca Operativa; corsi più avanzati sono disponibili nei quali vengono affrontati altri aspetti, quali:

- algoritmi e metodologie per modelli di Programmazione Matematica diversi, quali quelli con prevalenza di elementi non lineari (*programmazione non lineare*) oppure caratterizzati da forte incertezza sui dati (*ottimizzazione robusta e/o stocastica*);
- applicazioni a particolari ambiti operativi quali i *sistemi logistici* o le *reti di telecomunicazione*;
- metodologie per tipi di modelli diversi da quelli considerati nella Programmazione Matematica, quali le *tecniche di simulazione* oppure la *teoria dei giochi*.

È utile segnalare infine come la Ricerca Operativa da una parte sia utilizzata in, e dall'altra tragga risultati e metodologie da, moltissimi altri ambiti tecnologici e culturali, tra i quali ad esempio la statistica, l'analisi numerica, l'algoritmica, le metodologie di programmazione, le basi di dati e sistemi informativi, le tecniche di programmazione parallela, l'apprendimento automatico, la fisica, la chimica, moltissimi ambiti dell'ingegneria, e molti altri ancora.

1.1 Modelli e Problemi

L'elemento centrale nel processo decisionale è il *modello*, una descrizione, in generale per mezzo di strumenti di tipo logico-matematico, della porzione di realtà di interesse. Si distinguono almeno tre classi principali di modelli:

- Nei *giochi*, la difficoltà di modellare in modo matematico il comportamento degli individui o dei gruppi di individui presenti nella realtà sotto esame viene superata introducendo direttamente l'uomo nel modello attraverso i giocatori, a ciascuno dei quali viene affidato un prefissato ruolo.
- Nei *modelli di simulazione* si cerca di descrivere nel modo più accurato possibile il comportamento del sistema che si vuole studiare per mezzo di relazioni matematiche; quindi si studia su calcolatore la sua risposta a sollecitazioni che vengono realizzate, in genere per mezzo di generatori di numeri pseudo casuali, in modo che siano il più possibile simili a quelle reali.
- Nei *modelli analitici* invece tutto il sistema sotto esame è descritto per mezzo di relazioni matematiche (o logiche) tra variabili che rappresentano gli elementi del sistema; quindi si cercano valori per tali variabili che soddisfino i vincoli e che massimizzino o minimizzino una *funzione obiettivo* opportunamente definita.

Nell'analizzare la realtà per mezzo di modelli non va mai dimenticato lo scarto esistente tra la realtà stessa ed il modello: le soluzioni di un modello sono in realtà sempre soluzioni della *rappresentazione* che abbiamo costruito del problema reale, ma "la mappa non è il mondo". È quindi sempre necessario prestare grande attenzione alla fondatezza del modello costruito, in quanto esso sarà sempre una descrizione molto limitata della realtà, ma dovrà rappresentare con ragionevole accuratezza almeno gli aspetti che interessano ai fini della soluzione del problema decisionale che si sta affrontando.

Una caratteristica fondamentale di ogni modello è quindi la sua capacità di fornire previsioni corrette sul comportamento della realtà modellata in determinate circostanze di interesse. Ma altrettanto importante è la sua *utilizzabilità operativa*: deve essere possibile raccogliere i dati che caratterizzano il modello e determinare le soluzioni in un tempo compatibile con le esigenze del processo decisionale corrispondente. Ciò può voler dire cose molto diverse, dai problemi di controllo ottimo in cui la risposta deve essere disponibile in millisecondi fino ai problemi di pianificazione di lungo termine in cui è possibile attendere settimane prima di avere una risposta. In ogni caso, però, esistono dei limiti (tempo e risorse) entro i quali la risposta deve essere ottenuta. Quest'ultimo è un punto assolutamente non banale, in quanto molti modelli sono, allo stato dell'arte, computazionalmente intrattabili (si veda ad esempio l'Appendice A). La "qualità" di un modello è quindi il risultato del (difficile) bilanciamento tra due necessità contrastanti: da una parte quella di tenere in conto di tutti gli elementi necessari ad una corretta descrizione dei fenomeni, e dall'altra quella di avere un modello "sufficientemente semplice" affinché sia possibile ottenere le risposte (entro i limiti imposti dal processo decisionale). Questo è probabilmente lo snodo fondamentale di tutte le attività di creazione di modelli, ed è profondamente correlato con alcune delle domande più profonde riguardo alla capacità che gli uomini hanno di conoscere, prevedere e controllare il mondo che li circonda.

In queste dispense verranno trattati solamente modelli analitici, ed in particolare solo specifiche classi di questi. In questo contesto esiste una sostanziale equivalenza tra il concetto di modello e quello di *problema di ottimizzazione*; forniamo pertanto adesso alcune prime definizioni generali al riguardo. Per *problema* si intende una domanda espressa in termini generali, la cui risposta dipende da un certo numero di *parametri* e *variabili*. Un problema viene usualmente definito per mezzo di:

- una descrizione dei suoi parametri, in generale lasciati indeterminati;
- una descrizione delle proprietà che devono caratterizzare la risposta o *soluzione* desiderata.

Un'*istanza* di un dato problema (P) è quella particolare domanda che si ottiene specificando valori per tutti i parametri di (P).

Molto spesso un problema viene definito fornendo l'insieme F delle possibili risposte o soluzioni. Di tale insieme, detto *insieme ammissibile*, viene in generale data la struttura mediante i parametri da cui essa dipende; i suoi elementi vengono detti *soluzioni ammissibili*. Frequentemente F viene specificato indicando un insieme di "supporto" F' tale che $F \subseteq F'$, ed ulteriori condizioni (vincoli) che gli elementi di F devono soddisfare. In questo caso, si parla spesso degli elementi di $F' \setminus F$ come di *soluzioni non ammissibili*.

In un *problema di ottimizzazione*, sull'insieme ammissibile F viene definita una *funzione obiettivo*

$$c \;:\; F \;\rightarrow\; \mathbb{R}$$

che fornisce il costo o il beneficio associato ad ogni soluzione; la soluzione del problema è un elemento di F che rende minima, oppure massima, la funzione obiettivo. Un generico *problema di minimo* può essere scritto come

$$(P) \qquad \min \{\, c(x) : \; x \in F \,\}. \tag{1.1}$$

Sostituendo "min" con "max" in (1.1) si ottiene un *problema di massimo*. Chiamiamo

$$z(P) \;=\; \min \{\, c(x) \;:\; x \in F \,\}$$

il *valore ottimo* del problema. Una soluzione ammissibile $x^* \in F$ tale che $c(x^*) = z(P)$ è detta *soluzione ottima* per (P). Un problema di ottimizzazione può essere indifferentemente codificato come problema di massimo o di minimo: infatti, è immediato verificare che il problema

$$(P') \qquad -\max \{\, -c(x) \;: x \in F \,\},$$

è *equivalente* a (P): $z(P) = z(P')$, e i due problemi hanno lo stesso insieme di soluzioni ottime.

Esempio 1.1: Un problema di equipartizione

Il problema di equipartizione corrisponde alla seguente domanda: dato un insieme di n numeri naturali, $N = \{a_1, a_2, \ldots, a_n\}$, qual'è il sottoinsieme S di N tale che la differenza in modulo tra la somma dei numeri in S e quella dei numeri in $N \setminus S$ è la più piccola possibile? Una formulazione matematica del problema è

$$(EQ) \quad \min \left\{ \, c(S) = \left| \, \sum_{i \in S} a_i - \sum_{i \notin S} a_i \, \right| \; : \; S \subseteq N \right\} \; . \tag{1.2}$$

In questo caso, F è l'insieme di tutti i sottoinsiemi di N; infatti, l'unica condizione (vincolo) che una risposta (soluzione) deve soddisfare è di essere un sottoinsieme degli n numeri dati. Per questo problema, i *parametri* sono il numero "n" e gli n numeri "$a_1, a_2, \ldots, a_n$"; scegliendo ad esempio $n = 4$ e $N = \{7, 3, 4, 6\}$ si ottiene una particolare istanza del problema, in cui tutti i parametri sono specificati. Invece, S è la *variabile* del problema: $S = \{3, 7\}$ è una soluzione ottima per l'istanza considerata, con valore ottimo pari a zero.

Esempio 1.2: Un problema di impaccamento

Il problema del *Circle Packing* corrisponde alla seguente domanda: dato un quadrato di lato unitario ed un numero naturale n, qual è il massimo raggio che possono avere n cerchi identici inscritti nel quadrato che non si intersecano tra di loro. La domanda può anche essere posta in modi diversi ma equivalenti, ad esempio qual è la minima dimensione di un quadrato che può contenere n cerchi di raggio unitario, oppure qual è il modo di localizzare n punti nel quadrato in modo che la minima distanza tra di essi sia massima. Di questo problema è dimostrabile algebricamente quale sia il numero delle soluzioni ottime per n piccolo (ad esempio $\sqrt{2}$ per $n = 2$, $\sqrt{6} - \sqrt{2}$ per $n = 3$, $1/4$ per $n = 4$, $\ldots$), ma il problema è "difficile" per n arbitrario. In questo caso la descrizione dell'istanza è un semplice numero naturale, n. L'insieme ammissibile F è infinito (non enumerabile); una soluzione ammissibile può essere rappresentata mediante n coppie $(x_i, y_i) \subseteq [0, 1]^2$, $i = 1, \ldots, n$, indicanti i centri dei cerchi, più il singolo numero $r \in \mathbb{R}$ indicante il raggio. Una formulazione analitica del problema è quindi

$$
(CP) \quad \begin{array}{lrr}
\max & r & \\
 & \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \geq 2r & i = 1, \ldots, n \quad , \quad j = i+1, \ldots, n \\
 & r \leq x_i \leq 1 - r & i = 1, \ldots, n \\
 & r \leq y_i \leq 1 - r & i = 1, \ldots, n
\end{array} . \tag{1.3}
$$

Il problema (CP) si classifica come *continuo*, in quanto il suo insieme ammissibile ha (in principio) un numero infinito *non enumerabile* di elementi diversi, a differenza del problema (EQ) che si classifica come *combinatorio* perchè ha un insieme *enumerabile* di soluzioni ammissibili diverse; l'insieme ammissibile di (EQ) è in effetti *finito*, ma il numero delle soluzioni cresce esponenzialmente con la dimensione del problema (n), e quindi diviene rapidamente enorme. Nella pratica queste due classi problemi, nonostante alcune differenze di rilievo, sono usualmente "ugualmente difficili".

Dato un problema di ottimizzazione (P), sono possibili quattro casi:

- Il problema è *vuoto*, ossia $F = \emptyset$; in questo caso si assume per convenzione $z(P) = +\infty$ ($-\infty$ per un problema di massimo). L'esistenza di problemi vuoti potrebbe a tutta prima parere paradossale, ma in generale, come vedremo, non è facile stabilire se un insieme F specificato attraverso una lista di condizioni (vincoli) contenga oppure no elementi.

- Il problema è *inferiormente illimitato* (*superiormente illimitato* per un problema di massimo), ossia comunque scelto un numero reale M esiste una soluzione ammissibile $x \in F$ tale che $c(x) \leq M$ ($\geq M$); in questo caso il valore ottimo è $z(P) = -\infty$ ($+\infty$).

- Il problema ha *valore ottimo finito* $-\infty < z(P) < \infty$ ma *non ha soluzione ottima*, ossia non esiste nessun $x \in F$ tale che $c(x) = z(P)$. Un semplice esempio è dato dai problemi

$$\inf \{\, 1/x \ : \ x \geq 0 \,\} \qquad \text{e} \qquad \inf \{\, x \ : \ x > 0 \,\}$$

  che ammettono valore ottimo 0 ma nessuna soluzione ottima. Nella pratica tale situazione è indesiderabile, e viene evitata avendo cura di scegliere in modo opportuno c ed F; ciò sarà sempre fatto per i problemi trattati in questo corso.

- Il problema ha *valore ottimo finito* ed *ammette soluzione ottima*.

In certi casi ciò che il problema richiede è semplicemente la determinazione di una qualsiasi soluzione ammissibile, ovvero di fornire un elemento $x \in F$, se ne esiste uno, oppure di dichiarare che F è vuoto; in questo caso si parla di *problema decisionale* oppure di *problema di esistenza*. Dato un problema decisionale definito su $F \subseteq F'$, ad esso è naturalmente associato il *problema di certificato*: dato $x \in F'$, verificare se $x \in F$. Il problema di certificato è un problema decisionale che richiede semplicemente una risposta "sì" oppure "no".

In teoria, qualsiasi problema decisionale può essere formulato come problema di ottimizzazione: basta scegliere un opportuno insieme $F' \supseteq F$ e definire $c(x) = 0$ per ogni $x \in F$, $c(x) = 1$ per ogni $x \in$

$F' \setminus F$. Analogamente, un problema di ottimizzazione può essere espresso come problema decisionale: basta usare come insieme in cui si cerca una soluzione ammissibile l'insieme delle sue soluzioni ottime. Quest'ultima equivalenza è però solamente teorica; in pratica è difficile definire esplicitamente l'insieme delle soluzioni ottime di un problema non essendo noto il suo valore ottimo. In alternativa, dato il problema di ottimizzazione (1.1) possiamo definire il suo *problema decisionale associato*, o sua *versione decisionale*, come il problema di verificare l'esistenza di un soluzione ammissibile nell'insieme

$$F_k = \{ x \in F \, : \, c(x) \leq k \},$$

dove k è un prefissato valore. Si cerca cioè se esiste una soluzione ammissibile del problema di ottimizzazione che fornisca un valore della funzione obiettivo non superiore a k. In un certo senso, il problema decisionale associato ad un problema di ottimizzazione ne è una versione parametrica: facendo variare il parametro k e risolvendo ogni volta un problema di esistenza, è possibile determinare il valore ottimo della funzione obiettivo, o almeno una sua approssimazione con precisione arbitraria (risolvere il problema di ottimizzazione equivale a risolverne la variante decisionale per ogni possibile valore di k).

Per il problema di equipartizione (1.2) il problema decisionale associato richiede di stabilire se esiste un sottoinsieme S tale che la differenza in valore assoluto tra la somma dei numeri in S e quella dei numeri in $N \setminus S$ sia minore od uguale di un dato numero (intero) k. Per $k = 0$ si ottiene il problema di decidere se esiste un sottoinsieme S tale che la somma dei numeri in S e quella dei numeri in $N \setminus S$ sia uguale. Per il problema del Circle Packing (1.3) il problema decisionale associato richiede, dato un numero reale $\bar{r}$, di stabilire se esiste un modo di "impaccare" n cerchi di raggio $\bar{r}$ nel quadrato unitario.

Il senso di definire un problema di ottimizzazione è, almeno per le applicazioni pratiche, strettamente collegato alla possibilità di sviluppare procedure di calcolo (*algoritmi*) in grado di risolverne efficientemente le istanze. Da questo punto di vista, i problemi di ottimizzazione possono essere divisi in (almeno) due classi: quella dei problemi polinomiali ($\mathcal{P}$), e quella dei problemi $\mathcal{NP}$-ardui ($\mathcal{NP}$-completi quando si parli di problemi decisionali); per un rapido richiamo di questi concetti si veda l'Appendice A. Data l'esistenza di problemi per i quali non si conoscono algoritmi risolutivi di complessità polinomiale, è opportuno discutere più in dettaglio cosa significhi *risolvere un problema di ottimizzazione*.

In generale, un algoritmo che risolva il problema (P) è una procedura che prende in input una qualsiasi istanza p di (P) e fornisce in output una soluzione ottima x^* per quell'istanza[1]. Per molti problemi ciò è sostanzialmente—a meno di un fattore moltiplicativo polinomiale—equivalente a fornire in output il solo *valore ottimo* dell'istanza (per un approfondimento di questi concetti si veda il §4.3); possiamo quindi definire formalmente un algoritmo come una funzione $\mathcal{A} \, : \, P \to \mathbb{R}$. Un algoritmo per (P) che determini una soluzione ottima per *qualsiasi* istanza del problema viene detto *algoritmo esatto*. Poiché gli algoritmi esatti possono avere complessità troppo elevata, ad esempio esponenziale nelle dimensioni dell'istanza, ci si trova spesso nella necessità di ricorrere ad *algoritmi euristici*, detti anche semplicemente *euristiche* (si veda il Capitolo 5), ossia algoritmi che determinano solamente *una qualsiasi* soluzione ammissibile. In generale si è interessati ad ottenere "buone" valutazioni superiori (per problemi di minimo); per questo è opportuno introdurre misure che indichino "quanto buona" è una data soluzione. Data un'istanza p del problema di ottimizzazione (P), con valore ottimo $z(p)$, ed una sua soluzione ammissibile $\bar{x} \in F$, l'*errore assoluto* di $\bar{x}$

$$E_{\bar{x}} = c(\bar{x}) - z(p)$$

è una misura della "bontà" di $\bar{x}$ come soluzione di p; si noti che $c(\bar{x}) \geq z(p)$, ossia l'euristica produce un'*approssimazione superiore* (inferiore per un problema di massimo) del valore ottimo dell'istanza, e pertanto $E_{\bar{x}} \geq 0$ (nel caso di problemi di massimo bisogna quindi invertire il segno nella definizione).

[1] In alcuni casi si può essere interessati a determinare *tutte* le soluzioni ottime alternative del problema, ma molto spesso tale insieme può risultare "troppo grande", ad esempio di cardinalità esponenziale nelle dimensioni dell'istanza, e quindi normalmente si accetta che l'algoritmo riporti *una qualsiasi* soluzione ottima.

Poiché tale misura non è invariante rispetto ai cambiamenti di scala, si preferisce utilizzare l'*errore relativo*

$$R_{\bar{x}} = \frac{c(\bar{x}) - z(p)}{z(p)}$$

(assumiamo $z(p) > 0$ per semplificare la discussione, altrimenti occorre adattare marginalmente la trattazione). Dato $\varepsilon > 0$, la soluzione $\bar{x}$ si dice *ε-ottima* se $R_{\bar{x}} \leq \varepsilon$: un algoritmo euristico può essere considerato "buono" se produce soluzioni con "piccolo" errore relativo per *tutte* le istanze di (P). Un algoritmo $\mathcal{A}$ si dice quindi *ε-approssimato* se produce una soluzione ε-ottima per ogni istanza. Algoritmi ε-approssimati con ε "piccolo" possono essere valide alternative agli algoritmi esatti; per ulteriori informazioni si consulti l'Appendice A.

Si noti comunque che, per molti problemi di ottimizzazione, il problema di determinare *una qualsiasi* soluzione ammissibile ha la stessa complessità del problema originario; quindi, in generale gli algoritmi euristici possono "fallire", ossia non riportare nessuna soluzione ammissibile anche per istanze in cui $F \neq \emptyset$. In questo caso si assume che la valutazione superiore (inferiore, per un problema di massimo) ottenuta dall'algoritmo sia $+\infty$ $(-\infty)$, ossia "arbitrariamente cattiva", e quindi $R_{\bar{x}} = +\infty$.

Un problema fondamentale delle misure appena introdotte è che, in generale, il valore $z(p)$ non è noto, ed anzi calcolarlo è "altrettanto difficile" che determinare una soluzione ottime (si veda il §4.3), per cui calcolare l'errore (assoluto o relativo) di una soluzione $\bar{x}$ è non banale. Un metodo per stimare $z(p)$ è quello di costruire una qualche approssimazione del problema dato, ad esempio considerando solamente alcune delle condizioni (vincoli) che le soluzioni ammissibili devono soddisfare. In particolare, si definisce *rilassamento* di (1.1) qualsiasi problema

$$(\bar{P}) \qquad \min \{\ \bar{c}(x)\ :\ x \in \bar{F}\ \} \tag{1.4}$$

tale che $F \subseteq \bar{F}$ e $\bar{c}(x) \leq c(x)$ per ogni $x \in F$. In altre parole, $(\bar{P})$ è un rilassamento di (P) se ha "più soluzioni" di (P) e/o se la sua funzione obiettivo è un'approssimazione inferiore della funzione obiettivo c di (P) sull'insieme F. È immediato verificare che il valore ottimo di $(\bar{P})$ fornisce una *valutazione inferiore* del valore ottimo di (P), ossia $z(\bar{P}) \leq z(P)$; nel caso di problemi di massimo, la seconda condizione diventa $\bar{c}(x) \geq c(x)$ per ogni $x \in F$, ed il rilassamento fornisce una *valutazione superiore* del valore ottimo. È spesso possibile definire i rilassamenti in modo che siano risolubili mediante algoritmi di complessità inferiore rispetto a quelli necessari per il problema originale; si veda il Capitolo 6.

L'utilità fondamentale di un rilassamento è quella di permettere la *stima della qualità* di una soluzione ammissibile $\bar{x}$ dell'istanza p, ad esempio prodotta da un'euristica; infatti, se $\bar{p}$ è un rilassamento di p (e assumendo ancora $z(\bar{p}) > 0$ per semplicità), si ha che

$$R_{\bar{x}} = \frac{c(\bar{x}) - z(p)}{z(p)} \leq \frac{c(\bar{x}) - z(\bar{p})}{z(\bar{p})} \quad .$$

A differenza di $R_{\bar{x}}$, l'espressione a destra può essere effettivamente calcolata una volta ottenuta $\bar{x}$ e risolto $\bar{p}$; se risulta $(c(\bar{x}) - z(\bar{p}))/z(\bar{p}) \leq \varepsilon$, allora si può senz'altro concludere che $\bar{x}$ è ε-ottima. In altri termini, un rilassamento "facile" può fornire un *certificato di ottimalità* (approssimata) per un problema di ottimizzazione "difficile".

In effetti, può persino accadere che risolvere il rilassamento permetta di risolvere direttamente il problema originale. Questo capita in particolare se la soluzione ottima x^* di $(\bar{p})$ soddisfa le condizioni $x^* \in F$ e $\bar{c}(x^*) = c(x^*)$, ossia è ammissibile per il problema originale e la funzione obiettivo $\bar{c}$ ha in x^* lo stesso valore della funzione obiettivo reale c. In questo caso, infatti, è immediato verificare che x^* è anche *soluzione ottima per* (p), in quanto

$$\bar{c}(x^*) = z(\bar{p}) \leq z(p) \leq c(x^*) = \bar{c}(x^*)$$

ossia x^* fornisce contemporaneamente sia una valutazione inferiore che superiore del valore ottimo, e le due coincidono.

Quando ciò non accade, la valutazione inferiore $z(\bar{p})$ e la soluzione x^* del rilassamento possono comunque essere sfruttate per risolvere il problema (P). Infatti, combinando opportunamente euristiche e rilassamenti è possibile realizzare algoritmi esatti, anche se di complessità esponenziale al caso pessimo, per molti problemi di ottimizzazione; si veda il Capitolo 7.

1.2 Tecniche di Modellazione

La costruzione di un modello analitico di un sistema reale è una delle attività più creative e certamente più utili nello studio di sistemi organizzativi e gestionali, nella progettazione industriale, nella descrizione di sistemi altamente complessi quali quelli informatici ed in molte altre discipline. In quanto attività creativa, la costruzione di modelli non può affidarsi solamente all'uso di tecniche standard; non esistono cioè metodologie formali in grado di costruire automaticamente un modello analitico, anche se esistono tecniche e strumenti software in grado di facilitare ed automatizzare alcune fasi del processo di modellazione. La costruzione di un modello è comunque lasciata fondamentalmente alla fantasia, alla creatività ed all'esperienza del singolo, il quale può certamente fare riferimento ai modelli che ha incontrato precedentemente cercando di adattarli ove possibile, ma può anche trovarsi nella necessità di crearne di interamente nuovi.

Come abbiamo discusso, nella costruzione di modelli (in generale, ed in particolare di ottimizzazione) è sempre presente la necessità di bilanciare accuratamente due esigenze contrastanti: da una parte il modello deve essere *significativo*, ossia rappresentare in maniera sufficientemente accurata la realtà, ma dall'altra deve essere *effettivo*, ossia deve permettere di ottenere risultati in tempi compatibili con quelli del processo decisionale.

In generale, una data situazione reale può essere modellata in molti modi diversi, a seconda di quali aspetti della realtà vengono inseriti nel modello e di quali assunzioni (spesso semplificate) si fanno sulle relazioni che li legano. Tali scelte possono influenzare in modo spesso drastico la difficoltà del problema di ottimizzazione che si ottiene; come vedremo in molteplici occasioni, accade sovente che modifiche apparentemente minori ad un problema di ottimizzazione lo rendano significativamente più facile o difficile da risolvere. Anche una volta specificate con precisione le proprietà che il modello deve avere, la decisione sulla *forma* del problema può essere non banale: esistono molti modi diversi per specificare lo stesso insieme di enti matematici, ma alcuni sono più utili di altri al fine di sviluppare approcci algoritmici. I motivi specifici per cui alcuni modelli sono "migliori" di altri sono non banali, e verranno discussi al momento opportuno.

In queste note viene effettuata una scelta precisa sulla forma dei modelli utilizzati: essi sono tutti problemi di *Programmazione Lineare Intera* (*PLI*). La giustificazione di tale scelta risiede sostanzialmente nella combinazione di due fatti:

1. come vedremo, tale classe di problemi possiede una notevole "espressività" che consente di modellare moltissimi problemi di interesse pratico;

2. d'altra parte, le assunzioni restrittive sulla forma del modello permettono di utilizzare alcune delle tecniche algoritmiche attualmente ritenute più efficienti per la soluzione di problemi $\mathcal{NP}$-ardui.

È opportuno sottolineare da subito che questa scelta è in qualche modo arbitraria, e fondamentalmente dettata dal fatto che queste note si intendono per un corso di base di Ricerca Operativa. Esistono molti casi in cui si utilizzano modelli di tipo diverso. Ad esempio, esistono molte situazioni pratiche in cui non è appropriato (o possibile) evitare di utilizzare relazioni nonlineari nel modello, ma d'altra parte esistono forme di problemi di ottimizzazione che consentono l'uso di relazioni nonlineari e che sono comunque "facili". Infine, esistono altri formalismi che hanno proprietà espressive analoghe a quella della *PLI* e per i quali esistono approcci algoritmici di analoga sofisticazione e potenza, ad esempio la *programmazione logica con vincoli*. Si può però affermare che la *PLI* fornisce un valido compromesso tra semplicità, potenza espressiva e disponibilità di solutori (per quanto possibile) efficienti per la sua risoluzione. Inoltre, molti dei concetti algoritmici utilizzati per risolvere problemi più generali sono presenti, almeno in forma semplificata, all'interno degli approcci algoritmici per la *PLI*, che fornisce quindi un adeguato "banco di prova" per lo studio degli approcci risolutivi per problemi $\mathcal{NP}$-ardui.

Una volta fissate le regole di base per la costruzione dei modelli, in questo caso l'uso della Programmazione Lineare Intera (*PLI*), la modellazione è fondamentalmente lasciata alla capacità umana di determinare relazioni tra enti molto diversi quali un frammento di realtà e specifici elementi di un ente matematico. Per quanto questo processo non sia quindi riconducibile ad un insieme di regole fisse meccanicamente applicabili, è certamente possibile fornire una lista di alcune "tecniche base" che consentono di affrontare con successo un gran numero di situazioni diverse. In questo paragrafo forniamo quindi una breve introduzione alla modellazione di problemi di ottimizzazione sotto forma di *PLI* presentando un insieme di semplici modelli di alcuni noti problemi di ottimizzazione che si incontrano frequentemente nelle applicazioni, o direttamente o—più spesso—come sottoproblemi di problemi più complessi. Questi modelli contengono diversi "blocchi di base" che "codificano" parti di problema che si incontrano in moltissime applicazioni; componendo in modo opportuno questi blocchi si riescono a scrivere modelli di sistemi molto complessi.

1.2.1 Programmazione Lineare

Un problema di *Programmazione Lineare* (*PL*) è un problema di ottimizzazione caratterizzato dalle seguenti proprietà:

1. un numero *finito* n di variabili, che possono assumere valori reali;

2. una funzione obiettivo lineare, cioè del tipo $f(x) = cx$ dove $c \in \mathbb{R}^n$ è il vettore dei costi (fissato) ed $x \in \mathbb{R}^n$ è il vettore delle variabili;

3. un insieme ammissibile definito da un insieme finito di m vincoli lineari del tipo $ax = b$, e/o $ax \leq b$, e/o $ax \geq b$, dove $a \in \mathbb{R}^n$ e $b \in \mathbb{R}$.

I problemi di *PL* formano una delle più importanti classi di modelli di ottimizzazione, poiché permettono di modellare con sufficiente accuratezza molte situazioni reali e possono essere risolti efficientemente in generale, come mostrato nel Capitolo 2). Introduciamo ora alcuni esempi di problemi di *PL*.

1.2.1.1 Pianificazione della produzione

La società *Pintel* deve pianificare la produzione della sua fabbrica di microprocessori. La Pintel possiede due diverse linee di prodotti: i processori *Pintium*, più potenti e destinati al mercato "server", ed i *Coloron*, meno potenti e destinati al mercato "consumer". L'impianto è in grado di realizzare 3000 "wafer" alla settimana: su ogni wafer trovano posto o 500 Coloron oppure 300 Pintium. La resa di un wafer dipende anch'essa dal tipo di processore: i Coloron, di minori dimensioni, hanno una resa media del 60%, mentre i Pintium, più grandi e quindi maggiormente sottoposti a difetti, solamente del 50%. I processori Pintium si vendono a 500$ al pezzo, mentre i Coloron si vendono a 200$ al pezzo. La divisione commerciale della Pintel ha anche stabilito che la massima quantità di processori che possono essere messi sul mercato ogni settimana senza causare un calo dei prezzi è di 400000 unità per i Pintium e di 700000 unità per i Coloron. Si vuole determinare le quantità di ciascun tipo di processore da produrre settimanalmente in modo da massimizzare il ricavo totale.

Formuliamo, matematicamente il problema. A questo scopo introduciamo le variabili x_P ed x_C, che indicano il numero di processori rispettivamente di Pintium e Coloron da produrre. Su queste variabili possiamo innanzitutto porre le seguenti limitazioni superiori ed inferiori sul valore delle variabili

$$0 \leq x_P \leq 400000$$

$$0 \leq x_C \leq 700000$$

che corrispondono al fatto che non è possibile produrre quantità negative di processori, ed alle limitazioni imposte dalla divisione commerciale per impedire un calo dei prezzi.

Restano da considerare le limitazioni legate al processo produttivo. Per esprimerle, indichiamo con w_P e w_C il numero di wafer utilizzati per produrre rispettivamente i Pintium e i Coloron; il seguente vincolo lega queste due variabili alla produzione settimanale

$$w_P + w_C \leq 3000 \ .$$

Conoscendo il numero di pezzi per wafer e la resa produttiva si ottengono queste relazioni che legano il numero di pezzi processori funzionanti con il numero di wafer prodotti:

$$\begin{array}{l} x_P = w_P \cdot 300 \cdot 0.5 = w_P \cdot 150 \\ x_C = w_C \cdot 500 \cdot 0.6 = w_C \cdot 300 \end{array} \ . \tag{1.5}$$

Eliminando le due variabili w_P e w_C dal vincolo sulla produzione settimanale si ottiene il vincolo

$$2x_P + x_C \leq 900000 \ .$$

Quindi, l'insieme ammissibile per il problema è

$$F = \{ \ (x_P, x_C) \ : \ 0 \leq x_P \leq 400000 \ , \ 0 \leq x_C \leq 700000 \ , \ 2x_P + x_C \leq 900000 \ \} \ .$$

Ad esempio, $(x_P, x_C) = (0, 0)$ è ovviamente una soluzione ammissibile per il problema. Un ovvio insieme di "supporto" per il modello è $F' = \mathbb{R}^2$, e la soluzione $(x_P, x_C) = (400000, 700000)$ non è ammissibile per il problema in quanto viola il vincolo sulla capacità di produzione dei wafers.

Il ricavo ottenuto dalla vendita della produzione settimanale è dato dalla seguente funzione (lineare) nelle variabili decisionali del problema:

$$c(x_P, x_C) = 500x_P + 200x_C \ .$$

Di conseguenza, un modello analitico per il problema della Pintel è il seguente:

$$\begin{array}{rrcrl} \max & 500x_P & + & 200x_C & \\ & x_P & & & \leq 400000 \\ & & & x_C & \leq 700000 \\ & 2x_P & + & x_C & \leq 900000 \\ & x_P & , & x_C & \geq 0 \end{array}$$

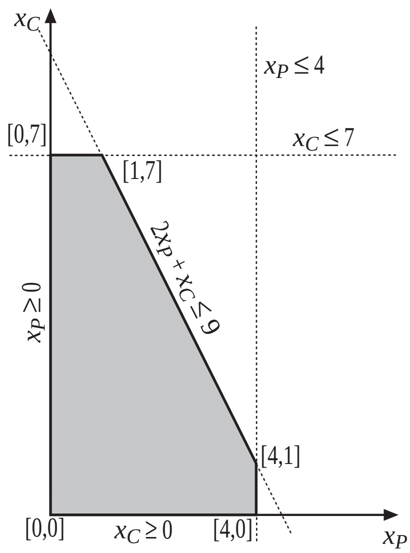


Figura 1.1: Insieme ammissibile per il problema della Pintel

Una volta determinata la soluzione (x_P, x_C), ossia in termini di processori, è possibile ricavare la soluzione (w_P, w_C), ossia in termini di wafers, semplicemente utilizzando le relazioni (1.5).

Esercizio 1.1 *Fornire tre diverse soluzioni ammissibili e valutare i ricavi che si ottengono.*

L'insieme ammissibile del problema della Pintel è mostrato in Figura 1.1 (entrambi gli assi sono stati scalati di un fattore 100000 per comodità di rappresentazione). L'insieme ammissibile è il *poliedro convesso* tratteggiato, delimitato dalle rette corrispondenti ai vincoli del problema. È possibile verificare che, per la linearità della funzione obiettivo, una soluzione ottima del problema (se esiste) si trova sempre in un *vertice* del poliedro, in questo caso quello corrispondente al punto $[\,4\,,\,1\,]$.

Un aspetto che resta ancora da discutere è il dominio ammissibile per la variabili w_P e w_C. Tali variabili rappresentano la produzione di unità discrete di bene (i singoli wafers), e quindi sarebbe ragionevole imporre l'ulteriore vincolo che, nella soluzione attesa del problema, esse possano assumere soltanto valori interi. Si noti che imporre l'interezza di w_P e w_C *non* è equivalente ad imporre l'interezza di x_P e x_C; la soluzione ottima del problema è $[\, x_P \,,\, x_C \,] = [\, 400000 \,,\, 100000 \,]$, quindi intera, ma tale soluzione corrisponde, in termini di wafers, a $[\, w_P \,,\, w_C \,] = [\, 2666.\bar{6} \,,\, 333.\bar{3} \,]$. D'altro canto, i dati del problema sono tali per cui si può ragionevolmente ipotizzare che una differenza di un wafer non cambi sostanzialmente la qualità della soluzione in pratica: le stime della divisione commerciale hanno presumibilmente un rilevante margine di incertezza, ed anche il vincolo sul numero di wafers è probabilmente soggetto a rilevanti incertezze (guasti di macchinari o scioperi, eventi imprevedibili, ecc.) per cui la produzione di wafers in ogni specifica settimana potrebbe essere leggermente diversa da quella ipotizzata. In queste circostanze, si può pensare che, ad esempio, la soluzione ammissibile intera $[\, w_P \,,\, w_C \,] = [\, 2666 \,,\, 334 \,]$ sia comunque una "buona" decisione in pratica; per questo, è ragionevole ammettere valori frazionari per le variabili w_P ed w_C.

1.2.1.2 Il problema della Fonderia

Una fonderia deve produrre 1000 pezzi del peso ciascuno di un chilogrammo. Il ferro con cui tali pezzi sono fatti dovrà contenere manganese e silicio nelle seguenti quantità:

$$
\begin{array}{rcccl}
0.45\% & \leq & \text{manganese} & & \\
3.25\% & \leq & \text{silicio} & \leq & 5.5\%
\end{array}
$$

Sono disponibili tre tipi di materiale ferroso con le seguenti caratteristiche:

Materiale ferroso	A	B	C
Silicio (%)	4.00	1.00	0.60
Manganese (%)	0.45	0.50	0.40
Costo (Euro / kg.)	0.025	0.030	0.018

Inoltre si può aggiungere direttamente manganese al costo di 10 Euro al kg. Il problema che si vuole modellare è quello di determinare il piano di produzione che minimizza il costo del materiale utilizzato. Si vuole cioè individuare le quantità di materiale per ciascuno dei tre tipi A, B, o C e di manganese puro da acquistare per produrre i 1000 pezzi richiesti, spendendo il meno possibile.

Per costruire un modello analitico per il problema introduciamo le variabili x_1, x_2, x_3, x_4, aventi il seguente significato:

$x_1 (\geq 0)$: la quantità in kg di materiale ferroso A da utilizzare;
$x_2 (\geq 0)$: la quantità in kg di materiale ferroso B da utilizzare;
$x_3 (\geq 0)$: la quantità in kg di materiale ferroso C da utilizzare;
$x_4 (\geq 0)$: la quantità in kg di manganese da utilizzare.

Abbiamo imposto che le quantità di prodotto acquistate siano dei valori non negativi (vincoli di non negatività). Esistono poi altri vincoli che dovranno essere rispettati e che descriviamo di seguito. Il numero totale di kg prodotti deve essere 1000:

$$x_1 + x_2 + x_3 + x_4 = 1000 \; .$$

La quantità di silicio, in kg, presente nel prodotto risultante è data da

$$0.04x_1 + 0.01x_2 + 0.006x_3 \; .$$

La percentuale di silicio nel prodotto finito sarà quindi data da

$$100 \frac{0.04x_1 + 0.01x_2 + 0.006x_3}{1000} \; .$$

Tale numero deve essere compreso tra 3.25 e 5.5. Possiamo quindi esprimere la condizione sulla percentuale di silicio mediante i due vincoli lineari

$$4x_1 + x_2 + 0.6x_3 \geq 3250 \quad , \quad 4x_1 + x_2 + 0.6x_3 \leq 5500 \; .$$

Analogamente, per la condizione sulla percentuale di manganese si ottiene

$$0.45x_1 + 0.5x_2 + 0.4x_3 + 100x_4 \geq 450 \ .$$

Infine il costo del prodotto risultante è

$$0.025x_1 + 0.030x_2 + 0.018x_3 + 10x_4 \ .$$

Il problema della determinazione di un piano di produzione che minimizza il costo può quindi essere formulato come segue:

$$\begin{array}{rrcrcrcrcl}
\min & 0.025x_1 & + & 0.030x_2 & + & 0.018x_3 & + & 10x_4 & & \\
& 4x_1 & + & x_2 & + & 0.6x_3 & & & \geq & 3250 \\
& 4x_1 & + & x_2 & + & 0.6x_3 & & & \leq & 5500 \\
& 0.45x_1 & + & 0.5x_2 & + & 0.4x_3 & + & 100x_4 & \geq & 450 \\
& x_1 & + & x_2 & + & x_3 & + & x_4 & = & 1000 \\
& x_1 & , & x_2 & , & x_3 & , & x_4 & \geq & 0
\end{array}$$

Le variabili x_1, x_2, x_3 e x_4 corrispondono alle scelte operative che il problema reale richiede di compiere, e ciascun vincolo del modello corrisponde ad una condizione imposta dal problema reale. Determinare i valori delle variabili in modo che i vincoli siano soddisfatti e la funzione obiettivo assuma il minimo valore fornisce il miglior piano di produzione.

1.2.2 Variabili logiche

Negli esempi precedenti abbiamo incontrato variabili che rappresentano il livello di attività produttive, come il numero di processori da produrre nel §1.2.1.1 oppure il numero di pezzi da produrre nella fonderia nel §1.2.1.2. Variabili di questo tipo vengono anche dette *variabili quantitative*; ciò che le caratterizza è il fatto di rappresentare quantità di beni, di oggetti da produrre, comprare o utilizzare, oppure valori assunti da grandezze fisiche o economiche.

In altri casi le variabili rappresentano delle scelte di tipo logico (vero, falso); si parla allora di *variabili logiche*. Per esprimere questo tipo si scelta si utilizzano usualmente *variabili booleane*, o *binarie*, che possono assumere uno solo dei due valori numerici 0 o 1, attribuendo al valore 0 il significato di *falso* ed a 1 quello di *vero*.

È importante sottolineare come l'uso di variabili binarie, o più in generale di variabili vincolate ad assumere solamente un insieme discreto di valori, se da un lato aumenta l'espressività dei modelli di ottimizzazione, dall'altro ne può rendere molto difficile la soluzione. Si definiscono problemi di *Programmazione Lineare Intera* (*PLI*), o *Mista*, i problemi in cui i vincoli e la funzione obiettivo sono lineari, come nel caso della *PL*, ma in cui tutte le variabili, o un loro sottoinsieme, possono assumere solamente valori interi. Mentre la *PL* è un problema "facile", la *PLI* è in generale un problema "difficile" (si veda l'Appendice A).

Vediamo adesso alcuni esempi di modelli che usano variabili logiche.

1.2.2.1 Il problema dello zaino

Sia dato un insieme $E = \{ 1, 2, \ldots, n \}$ di elementi, a ciascuno dei quali sia assegnato un peso a_i ed un costo c_i, $i = 1, \ldots, n$, interi e positivi: il problema dello zaino (KP, da *Knapsack Problem*) consiste nel determinare un sottoinsieme di elementi che abbia costo totale massimo ed il cui peso totale non superi un prefissato intero b. Il nome deriva dal fatto che viene usualmente descritto come il problema di scegliere quali oggetti di un dato insieme mettere in uno zaino in modo da non superare un dato peso (o capacità), massimizzando il valore complessivo degli oggetti selezionati. Si assume che sia $0 < b < \sum_i a_i$, altrimenti il problema sarebbe banale; si assume anche che sia $a_i \leq b$ per $i = 1, \ldots, n$, in quanto nessun elemento di peso superiore alla capacità b può far parte di una soluzione ammissibile, e quindi ogni elemento di peso superiore a b può essere eliminato da E.

Il problema può essere scritto come un problema di massimo, con

$$c(S) = \sum_{i \in S} c_i \qquad F = \left\{ \; S \subseteq E \; : \; \sum_{i \in S} a_i \leq b \; \right\} \quad .$$

In questa descrizione del problema, il sottoinsieme S è l'unica variabile, che prende valori nell'insieme delle parti di E (2^E) ed è soggetta al vincolo $\sum_{i \in S} a_i \leq b$.

Possiamo formulare il problema come *PLI* introducendo, per ogni oggetto $i = 1, 2, \ldots, n$, una variabile binaria $x_i \in \{0, 1\}$, con il significato che la variabile assume valore 1 se l'elemento i-esimo appartiene al sottoinsieme selezionato, e 0 altrimenti (si decide cioè se inserire o meno l'oggetto). La condizione relativa alla capacità dello zaino diviene

$$\sum_{i=1}^{n} a_i x_i \leq b \quad ;$$

infatti, dato che ciascuna x_i può assumere solo i valori 0 o 1, nella somma vengono considerati i pesi dei soli oggetti selezionati. Analogamente, la funzione obiettivo, da massimizzare, è

$$c(x) = \sum_{i=1}^{n} c_i x_i$$

in quanto nella funzione obiettivo si sommano i costi dei soli oggetti selezionati. La formulazione che si ottiene è quindi

$$\text{(KP)} \qquad \max \left\{ \; \sum_{i=1}^{n} c_i x_i \; : \; \sum_{i=1}^{n} a_i x_i \leq b \; , \; x_i \in \{0, 1\} \qquad i = 1, \ldots, n \; \right\} \quad .$$

Il problema dello zaino può essere trasformato in un problema di minimo con vincolo di $\geq$ (in letteratura si trovano entrambe le formulazioni).

Esercizio 1.2 *Costruire un'istanza del problema dello zaino con 6 oggetti, definendone costo e peso; formulare quindi l'istanza, fornire due soluzioni ammissibili e valutarne il costo.*

1.2.2.2 Albero di copertura di costo minimo

La banca Gatto & Volpe ha molte filiali sparse per l'Italia ed un Centro Elettronico Unificato (CEU) in cui vengono svolte tutte le transazioni. La banca ha bisogno di collegare tutte le filiali al CEU: per ragioni di sicurezza i dati non possono transitare sulla rete pubblica, e quindi occorre affittare linee dedicate. È possibile affittare una linea dedicata dal CEU ad ogni filiale, ma, se la capacità delle linee è sufficientemente grande, ciò non è necessario: può essere più conveniente collegare gruppi di filiali "vicine" tra loro, e solo una di esse al CEU. Il problema è quindi determinare quali linee affittare (nell'ipotesi che la capacità delle linee sia sufficiente) per collegare tutte le filiali al CEU minimizzando il costo di affitto.

Per modellare il problema, si può considerare un grafo non orientato $G = (N, A)$, dove N è l'insieme dei nodi con $|N| = n$, A è l'insieme degli archi con $|A| = m$ e ad ogni arco $(i, j) \in A$ è associato un costo c_{ij} reale positivo (per maggiori dettagli sui grafi, si rinvia all'Appendice B). Nel grafo, ogni nodo rappresenta una filiale, tranne un nodo "radice" che rappresenta il CEU, e gli archi rappresentano i potenziali collegamenti tra coppie di filiali o tra filiali e CEU, con il relativo costo. Il problema del progetto della rete dati della banca può quindi essere formulato matematicamente come il problema di determinare un grafo parziale $G' = (N, A')$, connesso e di costo minimo, dove il costo di G' è dato dalla somma dei costi degli archi in A'. È facile verificare che la struttura di collegamento ottimale (qualora la capacità delle linee sia sufficiente) è un albero di copertura di G, cioè un grafo parziale di G che sia connesso e privo di cicli. Il problema del progetto della rete dati della banca corrisponde quindi al problema di ottimizzazione con

$$\begin{array}{rcl} F & = & \{ \; T \subseteq A : (N, T) \text{ è un albero di copertura per } G \; \} \\ c(T) & = & \sum_{(i,j) \in T} c_{ij} . \end{array}$$

Possiamo formulare come *PLI* il problema di determinare un *albero di copertura di costo minimo* (MST, da *Minimal Spanning Tree*) introducendo, per ogni arco $(i, j) \in A$, una variabile binaria x_{ij}

che assume il valore 1 se l'arco (i,j) viene selezionato per formare l'albero di copertura e 0 se l'arco non viene selezionato. Affinché l'insieme degli archi selezionati formi un grafo parziale connesso, è necessario e sufficiente che, per ogni sottoinsieme S dei nodi (non vuoto e non coincidente con N) vi sia almeno un arco selezionato che ha un estremo in S e l'altro in $N \setminus S$ (ossia sia selezionato almeno uno degli archi del *taglio* $(S, N \setminus S)$; per maggiori dettagli si veda l'Appendice B). Pertanto, imponendo i seguenti *vincoli di connessione*

$$\textstyle\sum_{i \in S, j \notin S} x_{ij} \geq 1 \qquad \emptyset \subset S \subset N \tag{1.6}$$

si garantisce che i valori assunti dalle variabili decisionali definiscano un grafo parziale connesso. La funzione obiettivo, da minimizzare, è

$$\textstyle\sum_{(i,j) \in A} c_{ij} x_{ij} \ , \tag{1.7}$$

pertanto una formulazione *PLI* per il problema è la seguente:

$$\text{(MST)} \qquad \min \left\{ \textstyle\sum_{(i,j) \in A} c_{ij} x_{ij} \ : \ \sum_{i \in S, j \notin S} x_{ij} \geq 1 \quad \emptyset \subset S \subset N \ , \ x_{ij} \in \{0,1\} \quad (i,j) \in A \right\} .$$

Nell'ipotesi che i pesi siano strettamente positivi, è facile verificare che qualunque soluzione ottima x^* del problema definisce un albero di copertura di costo minimo. Infatti, qualunque soluzione ammissibile rappresenta un sottografo connesso di G, che quindi contiene sicuramente un albero di copertura. Se per assurdo la soluzione rappresentasse un sottografo con più di $n-1$ archi, ossia non fosse un albero, allora tale sottografo conterrebbe almeno un ciclo: eliminando un qualsiasi arco di questo ciclo si otterrebbe ancora un grafo connesso, il cui vettore di incidenza $\bar{x}$ sarebbe una soluzione ammissibile per il problema con costo strettamente minore di x^*, il che contraddice l'ottimalità di quest'ultima.

Questa formulazione "basata sui tagli" ha un numero esponenziale di vincoli, uno per ogni possibile sottoinsieme proprio di N. Esistono altre formulazioni con un numero polinomiale di vincoli. In generale, come vedremo nel seguito, formulazioni diverse dello stesso problema possono essere utili in quanto possono suggerire approcci algoritmici diversi.

Esercizio 1.3 *Costruire un'istanza del problema dell'albero di copertura di costo minimo per un grafo con 4 nodi e 6 archi, definendo i pesi degli archi; formulare quindi l'istanza utilizzando i vincoli sui tagli, fornire due soluzioni ammissibili e valutarne il costo.*

1.2.2.3 Il problema del commesso viaggiatore

Un commesso viaggiatore deve consegnare le sue mercanzie in n località, compresa quella in cui si trova. Egli conosce le distanze tra ogni coppia di località, e vuole organizzare il suo viaggio in modo da minimizzare la distanza percorsa. Il suo problema può essere rappresentato mediante un grafo non orientato e completo (cioè contenente tutti i possibili archi) $G = (N, A)$, dove N è l'insieme dei nodi ($|N| = n$) che rappresentano le località, A è l'insieme degli archi ($|A| = m = n(n-1)/2$) e ad ogni arco $(i,j) \in A$ è associato un costo c_{ij}, reale e positivo, che rappresenta la distanza tra le località associate ai nodi i e j.

Il piano di viaggio del commesso viaggiatore, ossia visitare in sequenza le n località rientrando alla località di partenza, corrisponde ad un *ciclo Hamiltoniano* sul grafo G, cioè a una sequenza di archi che inizia e termina nello stesso nodo e che include ogni altro nodo una ed una sola volta. La lunghezza del ciclo Hamiltoniano è la somma dei pesi (distanze) dei suoi archi. Quello che il commesso viaggiatore vuole determinare è il più corto tra tutti i cicli Hamiltoniani, per cui il *problema del commesso viaggiatore* (TSP, da *Travelling Salesman Problem*) è anche detto *problema del ciclo Hamiltoniano di lunghezza minima*. (TSP) ha molte applicazioni pratiche, tipicamente collegate a problemi di trasporto, ma non solo: ad esempio, problemi analoghi si ritrovano nel servire in modo ottimale n richieste di lettura/scrittura su uno stesso disco magnetico (in ambiente parallelo o concorrente) minimizzando il ritardo dovuto ai movimenti della testina. Si osservi che l'avere supposto il grafo completo non comporta alcuna limitazione; infatti, è sempre possibile rendere completo un grafo con l'aggiunta di archi a costo opportunamente elevato ($+\infty$).

(TSP) è un problema di ottimizzazione in cui l'insieme ammissibile F è l'insieme di tutti i cicli Hamiltoniani del grafo G, e la funzione obiettivo $c(P)$ è la lunghezza del ciclo Hamiltoniano P. La sua versione decisionale richiede di determinare se il grafo G ha un ciclo Hamiltoniano di lunghezza non superiore ad un prefissato valore k.

Per definire una formulazione *PLI* di (TSP) utilizziamo, analogamente a quanto fatto per (MST), una variabile logica x_{ij} per ogni arco $(i,j) \in A$, che vale 1 se (i,j) appartiene al ciclo scelto e 0 altrimenti; la funzione obiettivo, da minimizzare, è allora la stessa (1.7) di (MST). Il fatto che si voglia ottenere un ciclo comporta che in ciascun nodo incidano esattamente due archi, proprietà che può essere imposta per mezzo dei vincoli:

$$\textstyle\sum_{(i,j)\in A} x_{ij} = 2 \quad i \in N \qquad , \qquad x_{ij} \in \{0,1\} \quad (i,j) \in A \ . \tag{1.8}$$

Si noti come il vincolo di integralità sulle variabili sia critico per garantire che i vincoli 1.8 assicurino che nel sottografo rappresentato dalle variabili $x_{ij} > 0$ ciascun nodo abbia esattamente due archi incidenti; ove tale vincolo non fosse imposto il sottografo potrebbe contenere molti archi incidenti nello stesso nodo, "distribuendo" le due unità disponibili frazionalmente su di essi. In generale, si noti che in un modello matematico qualsiasi vincolo logicamente necessario debba essere indicato *esplicitamente*: anche se una condizione può apparire ovvia a chi modella, il modello non ha "alcuna percezione della realtà" e la corrispondenza tra le due entità è completamente a carico del modellatore.

I vincoli (1.8) non garantiscono che gli archi (i,j) le cui variabili associate x_{ij} assumono il valore 1 formino un ciclo Hamiltoniano; infatti esse garantiscono solo una *copertura per cicli* del grafo come illustrato nell'esempio in Figura 1.2, in cui gli archi selezionati formano una copertura di tutti i nodi del grafo mediante due cicli disgiunti, uno di 3 archi e l'altro di 4. Per imporre che gli archi selezionati formino un unico ciclo (Hamiltoniano) possiamo utilizzare i vincoli di connessione basati sui tagli (1.6), introdotti per (MST). L'aggiunta dei vincoli di connessione ai vincoli di copertura per cicli (1.8) impone che la copertura per cicli formi un grafo connesso: ciò è possibile solo se si ha un unico ciclo nella copertura, che è quindi un ciclo Hamiltoniano. La formulazione completa di (TSP) diventa quindi

$$(\text{TSP}) \quad \begin{array}{lll} \min & \sum_{(i,j)\in A} c_{ij}x_{ij} & \\ & \sum_{i\in S, j\notin S} x_{ij} \geq 1 & \emptyset \subset S \subset N \\ & \sum_{(i,j)\in A} x_{ij} = 2 & i \in N \\ & x_{ij} \in \{0,1\} & (i,j) \in A \end{array} \ .$$

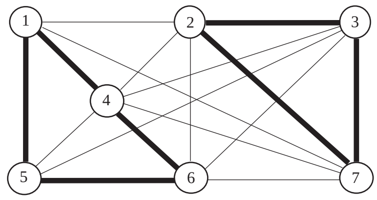


Figura 1.2: una copertura per cicli (archi in grassetto)

Come per (MST), esistono formulazioni per (TSP) aventi un numero polinomiale di vincoli; per esse si rinvia alla bibliografia suggerita.

Esercizio 1.4 *Assegnare dei pesi agli archi del grafo in Figura 1.2 supponendo che gli archi mancanti abbiano peso infinito. Costruire quattro cicli Hamiltoniani e valutarne la lunghezza.*

Esercizio 1.5 *Costruire un'istanza relativa ad un grafo completo con* 4 *nodi, e formulare (TSP) per tale istanza.*

Si può notare che la formulazione di (TSP) coincide con quella di (MST) a cui sono stati aggiunti i vincoli (1.8). Pertanto, l'insieme ammissibile di (MST) contiene quello di (TSP); siccome la funzione obiettivo è identica, (MST) è un *rilassamento* di (TSP). Questa osservazione ha una sua utilità pratica, in quanto mentre (TSP) è un problema $\mathcal{NP}$-arduo, esistono algoritmi polinomiali (e molto efficienti computazionalmente) per risolvere (MST).

È anche possibile osservare che rimuovendo dalla formulazione i vincoli (1.6), e lasciando quindi solamente i vincoli (1.8), si ottiene un altro rilassamento polinomiale, che può essere risolto attraverso algoritmi per problemi di assegnamento (si veda il Paragrafo 1.2.4.1).

1.2.3 Relazioni binarie

Spesso, le relazioni tra i valori di variabili booleane sono assimilabili alle ben note relazioni logiche tra *variabili proposizionali*, ossia variabili che possono assumere i valori *vero* o *falso*. In effetti, si possono costruire vincoli lineari tra variabili binarie equivalenti alle classiche relazioni logiche del calcolo proposizionale. Nel seguito, dato un letterale (proposizione elementare) a del calcolo proposizionale indicheremo con $x(a)$ la corrispondente variabile booleana, associando il valore 1 di $x(a)$ al valore *vero* di a ed il valore 0 di $x(a)$ al valore *falso* di a. Analizziamo adesso le più comuni relazioni tra variabili proposizionali.

Negazione. Data la variabile proposizionale a, la variabile complementare $b = \neg a$ viene rappresentata facilmente dalla *variabile complementare* $x(b) = 1 - x(a)$, con $x(b) \in \{0, 1\}$. Se si hanno due variabili proposizionali a e b e si vuole imporre che una sia il complemento dell'altra, è sufficiente imporre alle corrispondenti variabili booleane di rispettare il vincolo $x(a) + x(b) = 1$.

Implicazione. La relazione logica $a \Rightarrow b$ (a "implica" b) è esprimibile mediante la disuguaglianza $x(b) \geq x(a)$; infatti, $x(b)$ è forzata ad assumere il valore 1 se $x(a) = 1$.

Unione (Or). Date due variabili proposizionali a e b, la variabile $c = a \vee b$, che assume il valore *vero* quando almeno una delle due variabili è vera, può essere espressa mediante le seguenti relazioni:

$$x(c) \geq x(a) \quad , \quad x(c) \geq x(b) \quad , \quad x(c) \leq x(a) + x(b) \quad , \quad x(c) \in \{0, 1\} \ .$$

Infatti, le due prime diseguaglianze impongono alla variabile booleana $x(c)$ di assumere il valore 1 se una delle due altre variabili ha il valore 1. La terza impone il valore $x(c) = 0$ se $x(a) = x(b) = 0$.

Unione esclusiva (Or esclusivo). Date due variabili proposizionali a e b, la variabile $c = a \oplus b$, che assume il valore *vero* quando una sola delle due variabili è vera, può essere espressa mediante le seguenti relazioni:

$$\begin{array}{c} x(c) \geq x(a) - x(b) \quad , \quad x(c) \geq x(b) - x(a) \\ x(c) \leq x(a) + x(b) \quad , \quad x(c) \leq 2 - x(a) - x(b) \quad , \quad x(c) \in \{0, 1\} \end{array} .$$

Infatti, le due prime diseguaglianze impongono alla variabile booleana $x(c)$ di assumere il valore 1 quando una sola delle due altre variabili vale 1. La terza impone il valore $x(c) = 0$ se $x(a) = x(b) = 0$ e la quarta impone $x(c) = 0$ se $x(a) = x(b) = 1$.

Intersezione (And). Date due variabili binarie a e b, la variabile $c = a \wedge b$, che assume il valore *vero* solo quando ambedue le variabili sono vere, può essere espressa mediante le seguenti relazioni:

$$x(c) \leq x(a) \quad , \quad x(c) \leq x(b) \quad , \quad x(c) \geq x(a) + x(b) - 1 \quad , \quad x(c) \in \{0, 1\} \ .$$

Infatti, le prime due diseguaglianze impongono alla variabile booleana $x(c)$ di assumere il valore 0 quando almeno una delle due altre variabili vale 0. La terza impone $x(c) = 1$ se $x(a) = x(b) = 1$.

In generale, è possibile formulare molti problemi del calcolo proposizionale sotto forma di problemi di ottimizzazione. Questo tipo di formulazione permette di utilizzare tecniche di ottimizzazione in alternativa o in appoggio alle normali tecniche inferenziali usate nel calcolo logico.

1.2.3.1 Soddisfattibilità

Il problema della *Soddisfattibilità Proposizionale* (SAT, da *SATisfiability*) richiede di determinare se una data formula del calcolo proposizionale in *Forma Normale Congiuntiva*

$$A = C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

è soddisfattibile, dove $C_1, C_2, \ldots, C_m$ sono clausole del tipo

$$C_i = \pm P_1 \vee \pm P_2 \vee \ldots \vee \pm P_r$$

e con $\pm P_j$ si indica o il letterale P_j o la sua negazione $\neg P_j$, $j = 1, \ldots, n$. Si vuole cioè determinare se esiste un assegnamento di valore di verità *vero* o $false$ alle proposizioni elementari $P_1, P_2, \ldots, P_n$ che renda vera la formula A. Siccome qualsiasi formula del calcolo proposizionale può essere portata in FNC, questo problema ha rilevanti applicazioni pratiche, ad esempio per il progetto e la verifica di circuiti digitali VLSI.

Introduciamo n variabili binarie x_j associate ai letterali P_j, $j = 1, \ldots, n$, e definiamo

$$a_{ij} = \begin{cases} 1 & \text{se il letterale } P_j \text{ appare diretto nella clausola } C_i \\ -1 & \text{se il letterale } P_j \text{ appare negato nella clausola } C_i \\ 0 & \text{se il letterale } P_j \text{ non appare nella clausola } C_i \end{cases} .$$

Dato un qualunque vettore $x \in \{0,1\}^n$, che possiamo interpretare come un assegnamento di valori di verità agli n letterali $P_1, \ldots, P_n$, è facile verificare che la clausola C_i è soddisfatta dall'assegnamento di valori di verità corrispondente ad x se e solo se risulta

$$\textstyle\sum_{j=1}^n a_{ij} x_j \geq 1 - n(i) ,$$

dove $n(i)$ è il numero di letterali che appaiono negati in C_i. Una formulazione *PLI* di (SAT) è quindi

$$\min \left\{ 0 \ : \ \textstyle\sum_{j=1}^n a_{ij} x_j \geq 1 - n(i) \quad i = 1, \ldots, m \ , \ x_j \in \{0,1\} \quad j = 1, \ldots, n \right\}$$

Questo è un problema di ottimizzazione con una funzione obiettivo costante: il suo valore ottimo è quindi 0 se esiste almeno una soluzione ammissibile x, ossia un assegnamento di valori di verità alle proposizioni elementari che rende vere tutte le clausole $C_1, \ldots, C_m$, e $+\infty$ altrimenti. In effetti (SAT) è un *problema di decisione* piuttosto che non un problema di ottimizzazione: si è interessati solamente a determinare l'esistenza di una qualsiasi soluzione ammissibile, senza distinguere in alcun modo tra di esse. Una variante di (SAT) che è un problema di ottimizzazione è quella nella quale si cerca l'assegnamento di valori di verità ai letterali che massimizza il numero delle clausole C_i soddisfatte (Max-SAT).

Esercizio 1.6 *Si dia una formulazione analitica di (Max-SAT).*

Esiste una rilevante interfaccia tra problemi legati al calcolo logico e problemi di ottimizzazione: in effetti, (SAT) è stato il primo problema che è stato dimostrato essere $\mathcal{NP}$-completo (si veda l'Appendice A). Ciò permette di utilizzare tecniche di ottimizzazione per la soluzione di problemi relativi al calcolo logico e, viceversa, tecniche di inferenza per risolvere problemi di ottimizzazione. sistono persino alcuni interessanti risultati teorici che mostrano come le deduzioni logiche nel calcolo proposizionale possono essere viste come combinazioni lineari dei vincoli nella corrispondente formulazione di *PLI*, e quindi come le tecniche inferenziali siano un caso particolare di alcune tecniche per la risoluzione di problemi di *PLI*, dimostrando come la relazione tra ottimizzazione e calcolo logico sia profonda.

1.2.4 Vincoli di assegnamento e semiassegnamento

Descriviamo adesso due tipi di vincoli—su variabili binarie—che sono utilizzati molto di frequente in modelli di *PLI*. Sia $N = \{1, 2, \ldots, n\}$ un insieme di n oggetti e $V = \{v_1, v_2, \ldots, v_m\}$ un insieme di m elementi, che possono rappresentare altri tipi di oggetti, persone, sottoinsiemi, ecc. a seconda del contesto. Introduciamo la variabile logica x_{ij} col seguente significato: $x_{ij} = 1$ indica che all'oggetto i è stato assegnato l'elemento v_j, mentre $x_{ij} = 0$ indica che l'elemento v_j non è stato assegnato ad i. I *vincoli di semiassegnamento* impongono che a ciascun oggetto sia assegnato uno ed un solo elemento:

$$\textstyle\sum_{j=1}^m x_{ij} = 1 \quad , \quad i = 1, \ldots, n \ .$$

Si noti ancora una volta come l'ipotesi che le variabili siano (intere) binarie sia critica per garantire la correttezza di questi vincoli: rilassando il *vincolo di integralità*, esiste ad esempio, la soluzione frazionaria

$$x_{ij} = 1/m \quad , \quad i = 1, \ldots, n \ , \ j = 1, \ldots, m$$

che ovviamente non ha nessuna possibile interpretazione in termini del significato che si vuol dare alle variabili.

Capita spesso che un certo oggetto i possa essere assegnato solamente ad un dato insieme $B(i)$ di elementi "ammissibili" per i; in questo caso, è sufficiente definire le variabili x_{ij} solamente per le coppie (i,j) con $j \in B(i)$, e modificare i vincoli di semiassegnamento in

$$\sum_{j \in B(i)} x_{ij} = 1 \quad , \quad i = 1, \ldots, n \ .$$

Quando i due insiemi N e V hanno la stessa cardinalità, cioè $m = n$, è possibile che nel modello venga richiesto che anche a ciascun elemento sia assegnato uno e un solo oggetto; in tal caso si utilizzeranno i *vincoli di assegnamento*

$$\sum_{j=1}^{n} x_{ij} = 1 \quad i = 1, \ldots, n \qquad , \qquad \sum_{i=1}^{n} x_{ij} = 1 \quad j = 1, \ldots, n \ . \tag{1.9}$$

I vincoli (1.9) possono essere utilizzati per creare un *ordinamento* tra oggetti. Si supponga che, all'interno di un problema, si debba decidere con quale ordine effettuare n lavori $1, 2, \ldots, n$. In tal caso, i vincoli (1.9) impongono un ordinamento dei lavori se $x_{ij} = 1$ indica che il lavoro i viene effettuato come j-esimo; in questo modo, una soluzione ammissibile per i vincoli di assegnamento assegna ad ogni lavoro i una posizione all'interno dell'ordinamento.

Presentiamo adesso tre modelli in cui vengono utilizzati i vincoli di semiassegnamento.

1.2.4.1 Assegnamento di costo minimo

L'agenzia matrimoniale Cuori Solitari deve organizzare il gran ballo di fine anno. L'agenzia ha n clienti maschi e n clienti femmine, ed ha prenotato n tavoli da due posti al famoso ristorante Cupido. Dai profili psicologici raccolti dai clienti, l'agenzia è in grado di calcolare, per ogni maschio i, l'insieme $F(i)$ delle femmine con le quali potrebbe essere interessato ad intrecciare una relazione, e che potrebbero essere interessate ad intrecciare una relazione con lui; un analogo insieme $M(j)$ può essere ottenuto per ogni femmina j. Dai profili dei clienti, l'agenzia è anche in grado di calcolare, per ogni coppia (i,j) "compatibile", il costo c_{ij} della cena da offrire, che deve consistere di piatti graditi ad entrambi i commensali. L'agenzia vuole quindi decidere come formare le coppie per il gran ballo in modo da evitare coppie incompatibili e minimizzare il costo complessivo delle cene.

Una formulazione del problema dell'agenzia Cuori Solitari fa uso dei vincoli di assegnamento in cui C indica l'insieme delle coppie compatibili:

$$\begin{array}{lll} \min & \sum_{(i,j) \in C} c_{ij} x_{ij} & \\ & \sum_{j \in F(i)} x_{ij} = 1 & i = 1, \ldots, n \\ & \sum_{i \in M(j)} x_{ij} = 1 & j = 1, \ldots, n \\ & x_{ij} \in \{0, 1\} & (i,j) \in C \end{array}$$

Questo problema, noto come il problema dell'assegnamento di costo minimo, è polinomiale ed ha molte applicazioni in pratica; algoritmi per la sua soluzione sono descritti nella Sezione 3.5.

Dato un grafo non orientato G, il problema di determinare una copertura per cicli di costo minimo del grafo (si veda il §1.2.2.3) può essere formulato come un problema di assegnamento di costo minimo: sia gli oggetti i che gli elementi j corrispondono ai nodi del grafo, e le coppie (i,j) "compatibili" corrispondono agli archi del grafo, con il relativo costo. Di conseguenza, il problema dell'assegnamento di costo minimo è un rilassamento di (TSP).

Si noti quindi come uno stesso problema possa avere rilassamenti diversi: ad esempio, sia il problema dell'assegnamento di costo minimo che (MST) sono rilassamenti di (TSP). Diversi rilassamenti possono "catturare" parti diverse della struttura di un problema: ad esempio, (MST) incorpora i vincoli di connessione ma non quelli di formazione di cicli, mentre il problema dell'assegnamento incorpora i vincoli di formazione di cicli ma non quelli di connessione.

Infine, si noti che sia il problema dell'assegnamento che (MST) sono problemi "facili", mentre (TSP), che può essere considerato "l'intersezione" di questi due problemi, è "difficile". In generale, solo in casi molto particolari l'intersezione di due strutture combinatorie corrispondenti a problemi "facili" genera un problema a sua volta "facile".

1.2.4.2 Ordinamento di lavori su macchine: minimizzazione del numero delle macchine

Questo problema appare frequentemente in diversi ambiti, ad esempio in quello manifatturiero, nella gestione di sistemi informatici e nella gestione di progetti. Siano dati n *lavori* e m *macchine* uguali, su cui far eseguire i lavori. Il lavoro i-esimo, $i = 1, \ldots, n$, può essere svolto su ciascuna macchina e richiede un tempo di esecuzione d_i indipendente dalla macchina su cui viene eseguito. Il lavoro i-esimo viene consegnato al tempo t_i e deve essere immediatamente eseguito; essendo d_i la sua durata, l'esecuzione terminerà al tempo $t_i + d_i$. Il numero m di macchine è a priori illimitato (ovviamente, è sufficiente che sia $m = n$; se fosse $m > n$, alcune delle macchine sarebbero comunque inutilizzate). L'obiettivo è utilizzare il minimo numero possibile di macchine per eseguire tutti i lavori, assegnando alla stessa macchina lavori i cui tempi di esecuzione non si sovrappongano.

Si noti che una soluzione ammissibile corrisponde ad una partizione dell'insieme $N = \{1, 2, \ldots, n\}$ in m sottoinsiemi $N(1), N(2), \ldots, N(m)$, che possono anche essere vuoti, in cui il generico sottoinsieme $N(j), j = 1, \ldots, m$, rappresenta i lavori assegnati alla j-esima macchina.

Per descrivere il problema, introduciamo mn variabili binarie x_{ij}, intendendo che $x_{ij} = 1$ se il lavoro i viene eseguito sulla macchina j, e $x_{ij} = 0$ altrimenti. La variabile x_{ij} rappresenta l'appartenenza o meno di i a $N(j)$. Per rappresentare mediante le variabili x_{ij} l'insieme delle soluzioni ammissibili, dobbiamo imporre che ogni lavoro sia assegnato ad una ed una sola macchina, il che può essere espresso mediante i vincoli di semiassegnamento

$$\textstyle\sum_{j=1}^{m} x_{ij} = 1 \quad , \quad i = 1, \ldots, n \ . \tag{1.10}$$

Dobbiamo inoltre garantire che se due lavori i e h sono assegnati alla stessa macchina j, allora i loro tempi di elaborazione, cioè gli intervalli $[t_i, t_i + d_i]$ e $[t_h, t_h + d_h]$, siano *disgiunti*; in altri termini si deve avere che o $t_i + d_i \leq t_h$ (il lavoro i termina prima dell'inizio del lavoro h), oppure il viceversa, $t_h + d_h \leq t_i$. Per ogni lavoro, $i = 1, \ldots, n-1$, definiamo l'insieme $S(i)$ dei lavori h, con $h > i$, che sono *incompatibili* con esso:

$$S(i) = \left\{ \ h \in \{i+1, \ldots, n\} \ : \ [t_i, t_i + d_i] \cap [t_h, t_h + d_h] \neq \emptyset \ \right\} \quad , \quad i = 1, \ldots, n-1 \ .$$

Si noti che gli *insiemi di incompatibilità* $S(i)$, $i = 1, \ldots, n-1$, sono dei dati di "input" del problema. Mediante tali insiemi possiamo scrivere i seguenti *vincoli di compatibilità*:

$$x_{ij} + x_{hj} \leq 1 \qquad i = 1, \ldots, n-1 \quad , \quad h \in S(i) \quad , \quad j = 1, \ldots, m \ .$$

I vincoli di compatibilità impediscono che le variabili relative a due lavori incompatibili assumano entrambe valore 1. Il numero dei vincoli di incompatibilità dipende dai conflitti tra gli intervalli di tempo, e quindi dalla somma delle cardinalità degli insiemi di incompatibilità moltiplicato il numero m di macchine; tale numero è certamente non superiore a $mn(n-1)/2$.

Esercizio 1.7 *Dimostrare l'asserzione precedente.*

Per formulare la funzione obiettivo si deve esprimere il numero di macchine utilizzate; in questo caso è conveniente introdurre, per ogni macchina j, un'ulteriore variabile logica $y_j \in \{0, 1\}$ che assume il valore 1 se essa viene utilizzata e 0 altrimenti. In tal modo la funzione obiettivo, da minimizzare, è

$$\textstyle\sum_{j=1}^{m} y_j \quad .$$

Si deve quindi esprimere il legame tra le macchine utilizzate (cioè le variabili y_j) e i lavori assegnati ad esse (cioè le variabili x_{ij}); in altri termini si deve imporre che se alla macchina j è stato assegnato almeno un lavoro, allora "deve" essere $y_j = 1$. Ciò è esprimibile nei due modi diversi

$$y_j \geq x_{ij} \qquad i = 1, \ldots, n \quad , \quad j = 1, \ldots, m \tag{1.11}$$

$$ny_j \geq \textstyle\sum_{i=1}^{n} x_{ij} \qquad j = 1, \ldots, m \tag{1.12}$$

Il primo insieme di mn vincoli (1.11) forza $y_j = 1$ se almeno una delle variabili x_{ij} ha valore 1; tale insieme di vincoli è corrispondente alle implicazioni logiche "se il lavoro i viene eseguito sulla macchina j, allora la macchina j è utilizzata" (cf. 1.2.3). Il secondo insieme, di soli m vincoli, (1.12) mette in relazione y_j con il numero $\sum_{i=1}^{n} x_{ij}$ di lavori assegnati a j. Se esso è positivo, sicuramente non superiore ad n, il vincolo impone $y_j = 1$. Si noti ancora una volta come le variabili booleane abbiano una "doppia natura", logica e numerica.

La formulazione *PLI* del problema (MCMS, da *Minimal Cardinality Machine Scheduling*) che utilizza i vincoli (1.11) è:

$$
(\mathrm{MCMS}) \quad
\begin{array}{lll}
\min & \sum_{j=1}^{m} y_j & \\
& \sum_{j=1}^{m} x_{ij} = 1 & i = 1, \ldots, n \\
& x_{ij} + x_{hj} \leq 1 & i = 1, \ldots, n-1 \quad , \quad h \in S(i) \quad , \quad j = 1, \ldots, m \\
& y_j \geq x_{ij} & i = 1, \ldots, n \quad , \quad j = 1, \ldots, m \\
& x_{ij} \in \{0, 1\} & i = 1, \ldots, n \quad , \quad j = 1, \ldots, m \\
& y_j \in \{0, 1\} & j = 1, \ldots, m
\end{array}
$$

Si noti che, se alla macchina j non sono stati assegnati lavori, y_j può assumere entrambi i valori 0 e 1; siccome si minimizza la somma delle y_j, e nessun vincolo impone che y_j sia 1, all'ottimo tale variabile assumerà il corretto valore 0. Questo vale anche per il modello che usa i vincoli (1.12).

Si ricordi sempre che i vincoli di integralità sulle variabili devono sempre essere esplicitamente indicati, altrimenti le variabili potrebbero assumere valori non interi. È immediato verificare che nel *rilassamento continuo*, in cui le variabili x_{ij} possono assumere qualsiasi valore nell'intervallo $[0, 1]$, viene persa una parte sostanziale della struttura del problema originario.

Esercizio 1.8 *Dimostrare che la rimozione dei vincoli $y_j \in \{0, 1\}, j = 1, \ldots, m$ non può essere applicata se si usano i vincoli (1.12) al posto dei vincoli (1.11).*

Esercizio 1.9 *Costruire un'istanza del problema (MCMS) con 7 lavori, definendo le durate e i tempi di inizio di essi; formulare quindi l'istanza, fornire due soluzioni ammissibili e valutare il numero di macchine occorrenti.*

1.2.4.3 Assegnamento di frequenze

Un gestore di telefonia cellulare dispone di n stazioni radio base in grado di coprire tutta la città. Per poter attivare la rete, però, deve assegnare a ciascuna antenna una frequenza di trasmissione in modo tale che le antenne adiacenti—che servono celle sovrapposte—non producano interferenza. È disponibile un insieme $F = \{f_1, \ldots, f_m\}$, di frequenze, in cui ogni f_i è un valore numerico (espresso in Mhz o Ghz), e la relazione di adiacenza tra le antenne è rappresentata da un grafo (non orientato) $G = (N, A)$ in cui i nodi rappresentano le antenne ed esiste l'arco $(i, j) \in A$ se e solo se le due antenne i e j sono adiacenti, ossia servono celle che presentano sovrapposizioni, e quindi assegnando la stessa frequenza a i ed a j si creerebbe interferenza. Dato che acquisire il diritto di uso di ogni frequenza ha un costo, il gestore vuole determinare un'assegnamento di frequenze alle antenne che non produca interferenza e che utilizzi il minimo numero di frequenze: si noti che a due nodi non adiacenti nel grafo può in linea di principio essere assegnata la stessa frequenza. In altre parole, si vuole "colorare" il grafo G con i "colori" f_i in modo tale che i nodi adiacenti abbiano colori diversi e che il numero di colori utilizzati sia minimo.

Per descrivere il problema, introduciamo nm variabili binarie x_{if}, intendendo che $x_{if} = 1$ se la frequenza f viene assegnata all'antenna i. Siccome ad ogni antenna deve essere assegnata una ed una sola frequenza, dobbiamo imporre i vincoli di semiassegnamento

$$\sum_{f \in F} x_{if} = 1 \qquad i = 1, \ldots, n \ .$$

Dobbiamo inoltre garantire che l'assegnamento di frequenze non causi interferenza, ossia che ai nodi adiacenti siano assegnate frequenze diverse. Questo può essere fatto attraverso i vincoli logici

$$x_{if} + x_{jf} \leq 1 \qquad (i,j) \in A \quad , \quad f \in F \ .$$

Per formulare la funzione obiettivo si deve esprimere il numero di frequenze utilizzate; in questo caso è conveniente introdurre, per ogni frequenza f, un'ulteriore variabile logica y_f che assume il valore 1 se essa viene utilizzata, 0 altrimenti. La funzione obiettivo, da minimizzare, è $\sum_{f \in F} y_f$. Si deve quindi esprimere il legame tra le frequenze utilizzate (cioè le variabili y_f) e le antenne a cui sono assegnate (cioè le variabili x_{if}); questo può essere fatto ad esempio mediante i vincoli

$$y_f \geq x_{if} \qquad i = 1, \ldots, n \quad , \quad f \in F$$

che garantiscono che la variabile y_f ha valore 1 se almeno un'antenna utilizza la frequenza f. La formulazione *PLI* del problema (GC, da *Graph Coloring*) è:

$$(GC) \quad \begin{array}{lll} \min & \sum_{f \in F} y_f & \\ & \sum_{f \in F} x_{if} = 1 & i = 1, \ldots, n \\ & x_{if} + x_{jf} \leq 1 & (i,j) \in A \quad , \quad f \in F \\ & y_f \geq x_{if} & i = 1, \ldots, n \quad , \quad f \in F \\ & x_{if} \in \{0,1\} & i = 1, \ldots, n \quad , \quad f \in F \\ & y_f \in \{0,1\} & f \in F \end{array}$$

Il problema (GC) è quindi un molto simile caso al problema (MCMS) se interpretiamo le frequenze come macchine, le antenne come lavori e gli insiemi di incompatibilità tra lavori (antenne) come la relazione di adiacenza sul grafo G.

Esercizio 1.10 *Si formuli la generalizzazione del problema (GC) in cui ad ogni antenna i devono essere assegnate esattamente n_i frequenze diverse, e anche "frequenze vicine" fanno interferenza, nel senso che se a due nodi adiacenti i e j vengono assegnate due frequenze h e k, allora occorre che h e k siano "distanti", ossia che $|f_h - f_k| > \delta$ per una soglia δ fissata.*

1.2.5 Selezione di sottoinsiemi

Abbiamo già visto, considerando il problema di ordinamento di lavori su macchine 1.2.4.2, come formalizzare la partizione di un insieme in più sottoinsiemi. Riprendiamo ora questo problema generalizzandolo: sia E un insieme finito di elementi (ad esempio $E = \{1, 2, \ldots, n\}$), e

$$F = \{ F_1, F_2, \ldots, F_m \}$$

una famiglia di m suoi sottoinsiemi, ovvero $F_j \subseteq E$ per $j = 1, \ldots, m$. Ad ogni sottoinsieme F_j è associato un costo c_j, $j = 1, \ldots, m$; il problema che vogliamo considerare è quello di determinare una sottofamiglia $S \subseteq F$ di costo minimo che soddisfi particolari vincoli.

Introduciamo per questo m variabili binarie $x_1, x_2, \ldots, x_m$, col significato che $x_j = 1$ se $F_j \in S$, e $x_j = 0$ altrimenti, per $j = 1, \ldots, m$. Il costo di S è allora definito da:

$$\textstyle\sum_{j=1}^{m} c_j x_j \ .$$

Considereremo tre tipi di vincoli, detti rispettivamente di copertura, di partizione e di riempimento, che danno origine a tre diversi problemi di selezione di sottoinsiemi.

Problema di copertura

Supponiamo che si voglia che ogni elemento di E sia selezionato almeno una volta, cioè che appaia in almeno uno degli insiemi di S: ciò può essere espresso attraverso il vincolo

$$\sum_{j\,:\,i \in F_j} x_j \geq 1 \qquad i \in E \ , \tag{1.13}$$

che garantisce che per ogni $i \in E$ esista almeno un indice j per cui $F_j \in S$ e $i \in F_j$. Il problema di copertura è allora così definito:

$$\text{(PC)} \qquad \min \left\{ \sum_{j=1}^m c_j x_j \ : \ \sum_{j\,:\,i \in F_j} x_j \geq 1 \quad i \in E \quad , \quad x \in \{0,1\}^m \right\}$$

Si noti che è possibile rappresentare la famiglia F mediante una matrice $A \in \mathbb{R}^{n \times m}$ il cui generico elemento a_{ij} vale 1 se $i \in F_j$ e 0 altrimenti; così facendo, i vincoli di copertura divengono

$$\sum_{j=1}^m a_{ij}\, x_j \geq 1 \qquad i \in E \ .$$

Esempio 1.3:
Il direttore di un canale televisivo per le informazioni locali deve organizzare il lavoro delle squadre di giornalisti e operatori per coprire n diversi servizi fuori sede. Il capo-redattore ha predisposto m possibili attività che una singola squadra può svolgere, dove una attività è l'insieme dei servizi che possono essere svolti e comporta un determinato costo di retribuzione della squadra, comprensivo dei costi per la trasferta e eventuali ore di straordinario. Il direttore deve decidere quali delle attività far svolgere in modo da pagare il meno possibile con la garanzia che ciascuno dei servizi sia "coperto" da almeno una squadra.

Problema di partizione

Supponiamo che si voglia che ogni elemento di E sia selezionato esattamente una volta, cioè che appaia in uno ed uno solo degli insiemi di S. Formalizziamo questa richiesta attraverso i vincoli di partizione

$$\sum_{j\,:\,i \in F_j} x_j = 1 \qquad i \in E \ , \tag{1.14}$$

che garantiscono, che per ogni $i \in E$, esista un solo indice j per cui $i \in F_j$ e $F_j \in S$. Il problema di partizione si ottiene facilmente sostituendo i vincoli (1.14) a quelli di copertura (1.13):

$$\text{(PP)} \qquad \min \left\{ \sum_{j=1}^m c_j x_j \ : \ \sum_{j\,:\,i \in F_j} x_j = 1 \quad i \in E \quad , \quad x \in \{0,1\}^m \right\}$$

Esempio 1.4:
Il direttore di un cantiere nel Mugello per la nuova linea di alta velocità deve appaltare n diversi lavori. L'ufficio appalti ha predisposto m diversi possibili incarichi di appalto, ciascuno dei quali è formato da un sottoinsieme dei lavori che, per ragioni di efficienza esecutiva, è bene siano eseguiti dalla stessa ditta. Per ciascun incarico di appalto è definito anche il costo. Il problema è di decidere quali appalti assegnare affinché tutti i lavori siano svolti e il costo degli appalti sia minimo. Questo problema può essere facilmente formulato come problema di partizione.

Problema di riempimento

Supponiamo che si voglia che ogni elemento di E sia selezionato non più di una volta, cioè che appaia in al più uno degli insiemi di S. Formalizziamo questa richiesta attraverso i vincoli di riempimento

$$\sum_{j\,:\,i \in F_j} x_j \leq 1 \qquad i \in E \ ,$$

che garantiscono che per ogni $i \in E$ non possa esistere più di un indice j per cui $i \in F_j$ e $F_j \in S$. Il problema di riempimento è allora così definito:

$$\text{(PR)} \qquad \min \left\{ \sum_{j=1}^m c_j x_j \ : \ \sum_{j\,:\,i \in F_j} x_j \leq 1 \quad i \in E \quad , \quad x \in \{0,1\}^m \right\}$$

Si noti che, se tutti i costi sono non negativi, questo problema ha banalmente soluzione ottima nulla; per questo, lo si trova più spesso formulato come problema di massimizzazione.

Esempio 1.5: United colors of flowers
Il famoso allestitore di vetrine Ulivetto Laziali è stato chiamato a predisporre la vetrina del più importante fioraio di Pescia. Con gli n fiori, di forma e colore diversi, che il fioraio gli ha fornito, Ulivetto produce m diversi bozzetti di composizione floreale e per ciascuno di essi fornisce anche un punteggio di "bellezza compositiva". Decide quindi di allestire in vetrina un insieme di composizioni floreali che massimizzi il punteggio globale (definito come somma dei punteggi delle singole composizioni realizzate). Il problema è chiaramente un problema di riempimento in quanto non si possono realizzare due composizioni contenenti lo stesso fiore.

Esercizio 1.11 *Sia data la famiglia* $F = \{\{1,3,5\},\{2,3\},\{1,4\},\{3,4,5\},\ \{2\},\{5\},\{1,5\}\}$ *formata da 7 sottoinsiemi di* $E = \{1,2,3,4,5\}$. *Formulare i problemi di copertura, partizione e riempimento sapendo che il vettore dei costi è* $c = [3,5,1,9,2,4,1]$. *Determinare una soluzione ammissibile, se esiste, per ciascun problema.*

Esercizio 1.12 *Si formulino analiticamente i problemi di copertura, partizione e riempimento generalizzati in cui ogni oggetto i deve far parte di, rispettivamente, almeno, esattamente ed al più b_i sottoinsiemi.*

Esercizio 1.13 *Si formulino analiticamente le ulteriori generalizzazioni dei problemi di copertura, partizione e riempimento, generalizzate nell'esercizio precedente, in cui di ogni sottoinsieme F_j si possano prendere più copie, eventualmente limitate ad un numero massimo di u_j.*

1.2.6 Variabili a valori discreti

Come abbiamo visto in molti esempi precedenti, le variabili booleane hanno, nei modelli, una "doppia natura": da una parte vengono interpretate come variabili binarie, e quindi il loro valore viene associato a valori di verità, dall'altra sono normali variabili numeriche il cui dominio è limitato ad un insieme di valori discreti, per cui si può operare su di esse con le usuali operazioni aritmetiche. In questa e nelle successive sezioni mostreremo ulteriori esempi in cui questa "doppia natura" viene sfruttata.

Una variabile che può assumere solo alcuni, specifici, valori è detta *variabile a valori discreti*. Si pensi, ad esempio, a una variabile x che rappresenta la velocità di trasmissione fra due nodi collegati da modem e da un cavo telefonico: in questo caso essa può assumere solo uno dei valori corrispondenti alle velocità dei modem in commercio. Per esprimere la condizione $x \in \{v_1, v_2, \ldots, v_n\}$ possiamo introdurre n variabili booleane $y_1, y_2, \ldots, y_n$, in cui, per ogni i, $y_i = 1$ se $x = v_i$, e $y_i = 0$ altrimenti. I vincoli

$$\textstyle\sum_{i=1}^n y_i = 1 \qquad , \qquad y_i \in \{0,1\} \quad i = 1, \ldots, n$$

garantiscono che $x = \sum_{i=1}^n v_i y_i$ assuma uno solo degli n possibili valori. Vediamo adesso un esempio di uso di queste variabili.

Negli esempi precedenti le variabili utilizzate, siano esse binarie o quantitative, sono *variabili decisionali*, ossia che rappresentano decisioni effettive del problema, sia pure di "tipo" diverso. In generale, tutte le variabili di un modello sono variabili decisionali; può in alcuni casi far comodo, però, distinguere tra le *variabili strutturali* del modello, ovvero quelle che riflettono le decisioni fondamentali da assumere, e le *variabili ausiliarie*, ovvero quelle che sono introdotte nel modello col solo scopo di permettere la formulazione di qualche specifica condizione. Variabili ausiliarie sono, ad esempio, le variabili $y_1, y_2, \ldots, y_n$ appena introdotte.

1.2.6.1 Progetto di reti

Si vogliono collegare le reti fognarie di alcuni centri residenziali e industriali ad un impianto di depurazione e smaltimento. Ogni centro produce una quantità nota di rifiuti (liquidi), ed è necessario dimensionare opportunamente l'impianto di collegamento al depuratore in modo da consentirne il trasporto. Si possono scegliere condotte di diversa portata e il costo di messa in opera di una condotta sarà una funzione della sua portata.

Si consideri ad esempio il grafo $G = (N, A)$ di Figura 1.3(a) dove i nodi da 1 a 4 rappresentano i centri ed il nodo 5 rappresenta il depuratore. Accanto ad ogni nodo è indicata la quantità di liquido da depurare per unità di tempo prodotta dal centro ad esso corrispondente. Ogni arco $(i, j) \in A$ rappresenta un possibile collegamento; fra di essi andranno scelti quelli che verranno effettivamente costruiti, e per ciascuno dei collegamenti costruiti si dovrà determinare la portata. Osserviamo che i collegamenti hanno una direzione prefissata di percorrenza. Una soluzione ammissibile è data da un insieme di collegamenti che garantiscano il fluire dei liquidi da tutti i centri fino al depuratore. In Figura 1.3(b) è rappresentata una possibile soluzione.

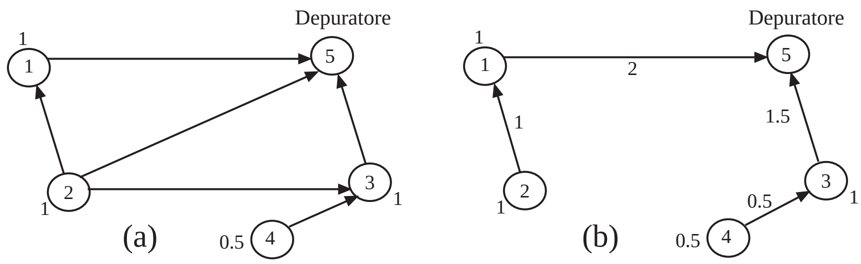


Figura 1.3: Progetto di una rete fognaria

Forniamo una descrizione analitica del problema mediante delle variabili quantitative x_{ij}, una per ogni arco (i, j) del grafo, che rappresentano la quantità di rifiuti liquidi inviati da i a j, nell'unità di tempo, lungo tale condotta. Esse sono chiaramente vincolate ad essere non negative; il valore $x_{ij} = 0$ indica che si è deciso di non utilizzare l'arco (i, j). Dobbiamo adesso introdurre dei vincoli che garantiscano che i rifiuti prodotti da ciascun centro giungano al depuratore. I rifiuti, per giungere al depuratore, dovranno passare per qualcuna delle condotte che vi arrivano direttamente, dovrà essere

$$x_{15} + x_{25} + x_{35} = 3.5 \quad .$$

Vincoli analoghi possono essere scritti per i nodi corrispondenti agli altri centri; in questo caso, però, è necessario tenere in conto anche del flusso in uscita dal nodo. In altri termini, per ogni centro occorre garantire che tutto il flusso che vi giunge, sia quello ivi prodotto che quello che lo attraversa proveniendo da altri centri, sia inviato, attraverso qualcuna delle condotte che escono dal nodo, o direttamente al depuratore o comunque ad un diverso centro, dal quale poi sarà ulteriormente instradato fino a raggiungere il depuratore. Ad esempio per il nodo 3 si ha

$$x_{23} + x_{43} + 1 = x_{35} \ ,$$

mentre per il nodo 1 si ha

$$x_{21} + 1 = x_{15} \ .$$

Per i nodi che hanno solamente archi uscenti, 2 e 4, i vincoli saranno

$$1 = x_{21} + x_{25} + x_{23} \qquad , \qquad 0.5 = x_{43} \quad .$$

Il costo di posa della condotta corrispondente all'arco (i, j) dipende fondamentalmente dalla portata della condotta e dalla lunghezza del tratto da compiere. Se supponiamo che siano disponibili condotte di qualsiasi portata, possiamo pensare di installare, su ogni arco, una condotta di portata esattamente pari a quella richiesta dal flusso dei rifiuti, in quanto non si avrebbe nessun vantaggio a fare altrimenti. Se supponiamo inoltre che il costo della condotta dipenda linearmente sia dalla lunghezza del tratto da compiere che dalla portata, abbiamo che il costo di installare una condotta sufficiente ad inviare la quantità di rifiuti x_{ij} dal centro i al centro j attraverso la condotta che li unisce direttamente è $c_{ij}x_{ij}$, dove $c_{ij} = \delta l_{ij}$, l_{ij} è la lunghezza del tratto da i a j e δ è il costo per unità di lunghezza di una condotta di capacità unitaria. In questo caso il problema del progetto di reti può essere formulato come

$$\begin{array}{rcrcrcrcrcrcrcr}
\min & c_{15}x_{15} & + & c_{21}x_{21} & + & c_{23}x_{23} & + & c_{25}x_{25} & + & c_{35}x_{35} & + & c_{43}x_{43} & & \\
& -x_{15} & + & x_{21} & & & & & & & & & = & -1 \\
& & - & x_{21} & - & x_{23} & - & x_{25} & & & & & = & -1 \\
& & & & + & x_{23} & & & - & x_{35} & + & x_{43} & = & -1 \\
& & & & & & & & & & - & x_{43} & = & -0.5 \\
& x_{15} & & & & & + & x_{25} & + & x_{35} & & & = & 3.5 \\
& x_{15} & , & x_{21} & , & x_{23} & , & x_{25} & , & x_{35} & , & x_{43} & \geq & 0
\end{array} .$$

In generale, però, a causa delle economie di scala il costo di posa di una condotta sarà una funzione concava del tipo di quella indicata in Figura 1.4. In ascissa si ha il valore x della sezione della condotta e in ordinata si ha il costo $f(x)$ della posa in opera della condotta; ad ogni $(i, j) \in A$ è associata una funzione $f_{ij}(x)$ che fornisce il costo globale dell'acquisto e posa in opera della condotta con portata x lungo tutto il tratto da i a j. Se supponiamo che siano disponibili condotte di qualsiasi portata, il problema diventa

$$\begin{array}{rcrcrcrcrcrcrcr}
\min & f_{15}(x_{15}) & + & f_{21}(x_{21}) & + & f_{23}(x_{23}) & + & f_{25}(x_{25}) & + & f_{35}(x_{35}) & + & f_{43}(x_{43}) & & \\
& -x_{15} & + & x_{21} & & & & & & & & & = & -1 \\
& & - & x_{21} & - & x_{23} & - & x_{25} & & & & & = & -1 \\
& & & & + & x_{23} & & & - & x_{35} & + & x_{43} & = & -1 \\
& & & & & & & & & & - & x_{43} & = & -0.5 \\
& x_{15} & & & & & + & x_{25} & + & x_{35} & & & = & 3.5 \\
& x_{15} & , & x_{21} & , & x_{23} & , & x_{25} & , & x_{35} & , & x_{43} & \geq & 0
\end{array} .$$

Si noti che questo problema ha una funzione obiettivo nonlineare, quindi non è un problema di *PLI*, ma piuttosto di *ottimizzazione nonlineare*. Nella realtà, però, le condotte sono solitamente disponibili soltanto in un numero finito di portate diverse; nel nostro esempio, supponiamo che le condotte sono disponibili in tre sole dimensioni, con portate rispettivamente 0.7, 1.4 e 3. In questo caso non possiamo più supporre che la portata della condotta installata sia uguale alla quantità di rifiuti effettivamente inviati: potrebbe infatti essere necessario installare condotte di portata maggiore. Per questo introduciamo un altro insieme di variabili quantitative y_{ij}, una per ogni arco (i, j) del grafo, che rappresentano la portata della condotta installata tra il nodo i ed il nodo j (0 se non viene installata nessuna condotta). Ovviamente, occorre inserire i vincoli

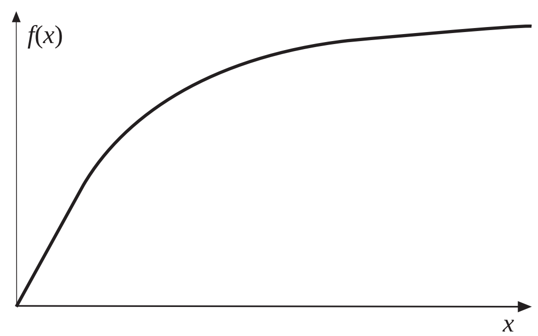


Figura 1.4: Funzione costo per il problema di progetto di rete

$$x_{ij} \leq y_{ij} \qquad (i, j) \in A$$

insieme a quelli

$$y_{ij} \in \{\ 0, 0.7, 1.4, 3\ \} \tag{1.15}$$

per garantire che la quantità di flusso inviata lungo ogni condotta non sia superiore alla portata della condotta effettivamente installata. Possiamo adesso trasformare questo problema in modo da sostituire le variabili quantitative y_{ij}, vincolate ad assumere un insieme discreto di valori, con variabili binarie. Per esprimere il vincolo (1.15) introduciamo, per ogni arco (i, j), tre nuove variabili binarie y^1_{ij}, y^2_{ij} e y^3_{ij} e sostituiamo il vincolo con

$$0.7y^1_{ij} + 1.4y^2_{ij} + 3y^3_{ij} \ = \ y_{ij} \tag{1.16}$$

$$y^1_{ij} + y^2_{ij} + y^3_{ij} \ \leq \ 1 \tag{1.17}$$

$$y^1_{ij}\ ,\ y^2_{ij}\ ,\ y^3_{ij} \ \in \ \{0, 1\}$$

Utilizzando (1.16) possiamo poi sostituire y_{ij} ovunque con

$$0.7y^1_{ij} + 1.4y^2_{ij} + 3y^3_{ij} \ ,$$

eliminando così tutte le variabili y_{ij} ed i vincoli (1.16) dal problema. Questo ci consente anche di ottenere nuovamente una funzione obiettivo lineare: infatti si ha che

$$f_{ij}(0.7y^1_{ij} + 1.4y^2_{ij} + 3y^3_{ij}) = f_{ij}(0.7)y^1_{ij} + f_{ij}(1.4)y^2_{ij} + f_{ij}(3)y^3_{ij}$$

e quindi il problema del progetto di reti può essere modellato come un problema di *PLI*.

Si noti che, di fatto, in questo caso non siamo interessati al valore delle funzioni f_{ij} in tutti i punti, ma ai loro valori nei punti di ascissa 0.7, 1.4 e 3. In effetti, è possibile formulare il problema eliminando completamente le variabili relative alle condotte, e considerando come costo per il flusso x_{ij} che passa per l'arco (i, j) il costo della più piccola sezione in commercio sufficiente a trasportare quella quantità di flusso. Di conseguenza, il costo del flusso ha la forma, detta *a gradini*, della funzione mostrata in figura 1.5. Nel seguito verrà mostrato come rappresentare questo tipo di funzioni mediante funzioni lineari e variabili binarie.

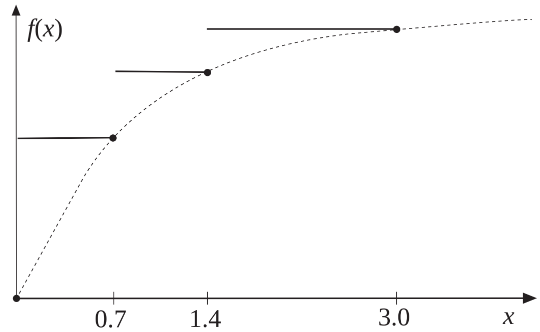


Figura 1.5: Una funzione costo a gradini

1.2.7 Minima quantità positiva prefissata

Spesso nella pianificazione della produzione (o trasporto) di beni, si ha un doppio livello di decisione: il primo livello sulla produzione o meno del bene, e il secondo livello sulla quantità di beni da produrre all'interno di un intervallo $[l, u]$, con $l > 0$. In pratica si ha che la variabile x, che rappresenta la produzione, può assumere valori solamente nell'insieme $\{0, [l, u]\}$, dove $x = 0$ rappresenta la decisione di "non produzione", mentre $x \in [l, u]$ indica la produzione in caso di decisione positiva. Per modellare i valori che x può assumere, introduciamo una variabile binaria y che assume il valore 0 se si decide di non produrre, e il valore 1 se si è deciso di produrre. Possiamo allora scrivere:

$$ly \leq x \leq uy \quad , \quad y \in \{0, 1\} \quad , \quad x \in \mathbb{R} \ ;$$

infatti, se $y = 0$, la variabile x è "forzata" ad assumere il valore 0, mentre se $y = 1$, essa può assumere un qualsiasi valore reale nell'intervallo $[l, u]$.

1.2.8 Funzione con carico fisso

Si consideri la seguente funzione "con carico fisso", che si vuole minimizzare,

$$f(x) = \begin{cases} 0 & \text{se } x = 0 \\ b + cx & \text{se } 0 < x \leq u \end{cases} \tag{1.18}$$

dove è $b > 0$ e u è un opportuno reale positivo (si veda anche la Figura 1.6). La funzione costo $f(x)$ compare spesso nei problemi di produzione: se non si produce ($x = 0$) non vi è costo ($f(x) = 0$), mentre se si decide di produrre (con il limite della capacità produttiva u) allora si ha un costo di investimento b, fisso ed indipendente dalla produzione, e un costo di produzione $c > 0$ per ogni unità di prodotto.

Per rappresentare analiticamente una tale funzione *concava* nell'intervallo $[0, u]$, analogamente a quanto fatto nel caso precedente, introduciamo la variabile decisionale y col seguente significato: $y = 0$ implica $x = 0$ e $y = 1$ implica $x \geq 0$. Le condizioni che legano la variabile x alla y sono:

$$0 \leq x \leq yu \quad , \quad y \in \{0, 1\} \quad , \quad x \in \mathbb{R} \ .$$

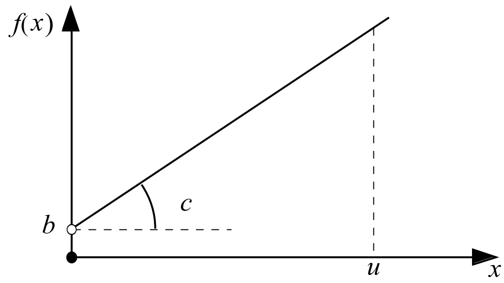


Figura 1.6: Una funzione con "carico fisso"

Si noti che il limite superiore u è essenziale nella formulazione. Nei casi pratici è sempre possibile introdurre un valore finito che sia un limite superiore ai valori che la variabile x può assumere. Possiamo ora sostituire la funzione $f(x)$ con la nuova funzione

$$g(x, y) = by + cx \ ;$$

infatti, quando $y = 0$ si ha $g(0, 0) = f(0) = 0$, mentre quando $y = 1$ si ha $g(x, 1) = b + cx$. Osserviamo che i vincoli introdotti non escludono la possibilità che sia contemporaneamente $y = 1$ e $x = 0$, in questo caso avremmo $g(0, 1) \neq f(0)$. Quindi la $g(x, y)$ non riesce a rappresentare in modo univoco $f(x)$; tuttavia, poiché il nostro obiettivo è minimizzare $f(x)$ e quindi $g(x, y)$, la soluzione $(x, y) = (0, 1)$ viene esclusa essendo

$$g(0, 1) = b > g(0, 0) = 0 = f(0) \ .$$

Naturalmente, la formulazione proposta non sarebbe adeguata se l'obiettivo fosse la massimizzazione della funzione $f(x)$.

Esercizio 1.14 *Disegnare e descrivere analiticamente la funzione $f(0) = 0$, e $f(x) = 180 + 2x$ per $0 < x \leq 250$.*

1.2.9 Vincoli di soglia

In molti problemi si ha bisogno di rappresentare dei "valori soglia" mediante variabili e vincoli lineari. Più formalmente, siano $x_1, x_2, \ldots, x_n$, variabili reali; si vogliono definire due variabili l ed u che siano una approssimazione rispettivamente per difetto e per eccesso di ciascuna delle variabili:

$$l \leq \min\{\, x_i \ : \ i = 1, \ldots, n \,\} \quad , \quad u \geq \max\{\, x_i \ : \ i = 1, \ldots, n \,\} \ .$$

Tali condizioni sono facilmente ottenute imponendo i vincoli

$$l \leq x_i \quad , \quad u \geq x_i \quad i = 1, \ldots, n \ ,$$

infatti così facendo l sarà non superiore ed u sarà non inferiore a ciascuna delle n variabili. Se si massimizza l o se si minimizza u, all'ottimo esse saranno uguali rispettivamente al massimo e al minimo dei valori assumibili dalle n variabili. Vediamo ora un'applicazione dei vincoli di soglia relativamente ad un altro problema di ordinamento di lavori su macchine.

1.2.9.1 Ordinamento di lavori su macchine: minimizzazione del tempo di completamento

Questo problema è una variante di (MCMS) (si veda 1.2.4.2) in cui il tempo di inizio di ciascun lavoro non è fissato, e può essere deciso liberamente. È invece fissato il numero di macchine da utilizzare, e si chiede di completare tutti i lavori nel minor tempo possibile.

Se $N(j)$ è l'insieme dei lavori assegnati alla macchina j, $j = 1, \ldots, m$, allora il tempo di lavoro della macchina è $D(j) = \sum_{i \in N(j)} d_i$; il *tempo di completamento* T, da minimizzare, è il massimo dei tempi di lavoro delle macchine

$$T = \max\{\, D(j) \ : \ j = 1, \ldots, m \,\} \ ,$$

ossia il tempo necessario alla macchina più carica per terminare. Utilizziamo le stesse variabili x_{ij} utilizzate in (MCMS); useremo i vincoli di semi-assegnamento (1.10), assieme ai vincoli di integralità, per imporre che ciascun lavoro sia eseguito da una ed una sola macchina.

Per formulare la funzione obiettivo, da minimizzare, dobbiamo esprimere il massimo tra m quantità. Per questo utilizziamo la tecnica appena introdotta ed introduciamo una nuova variabile t che rappresenta una approssimazione per eccesso del tempo di completamento:

$$\sum_{i=1}^{n} d_i x_{ij} \leq t, \quad j = 1, \ldots, m \ .$$

La formulazione risultante del problema (MMMS, da *Minimal Makespan Machine Scheduling*) è:

$$(\text{MMMS}) \qquad \begin{array}{lll} \min & t & \\ & \sum_{j=1}^{m} x_{ij} = 1 & i = 1, \ldots, n \\ & \sum_{i=1}^{n} d_i x_{ij} \leq t & j = 1, \ldots, m \\ & x_{ij} \in \{0, 1\} & i = 1, \ldots, n \quad , \quad j = 1, \ldots, m \end{array} .$$

Come abbiamo già notato, in qualsiasi soluzione ottima del problema la variabile ausiliaria t fornisce il tempo di completamento e non una sua approssimazione; infatti, se per assurdo si avesse $t > D(j)$ per ogni $j = 1, \ldots, m$, si otterrebbe un valore inferiore della funzione obiettivo ponendo $t = \max\{\ D(j)\ :\ j = 1, \ldots, m\ \}$.

Esercizio 1.15 *Costruire un'istanza del problema (MMMS) con 3 macchine e 7 lavori, definendo le durate di essi; formulare quindi l'istanza, fornire tre soluzioni ammissibili e valutarne il tempo di completamento.*

Esercizio 1.16 *Una ditta di costruzioni edilizie ha deciso di subappaltare n diverse opere a n diversi artigiani. Ad ogni artigiano $i = 1, \ldots, n$ chiede di fornire il costo preventivo c_{ij} che richiede per effettuare l'opera j, per ogni $j = 1, \ldots, n$. Si vuole assegnare un'opera a ciascun artigiano in modo che tutte le opere siano effettuate e il costo massimo dei subappalti assegnati sia minimizzato. Formulare il problema.*

Esercizio 1.17 *Si provi ora a massimizzare il costo minimo dei subappalti assegnati.*

Esercizio 1.18 *Si formuli il problema in cui si vuole che la differenza tra il massimo e il minimo costo dei subappalti assegnati sia la minore possibile.*

1.2.10 Come rappresentare il valore assoluto

Può capitare di dover trattare un vincolo del tipo:

$$|g(x)| \leq b \ , \tag{1.19}$$

dove $g(x)$ è una funzione dell'insieme di variabili x e b è un numero reale positivo. Se $g(x) \geq 0$ allora il vincolo diviene $g(x) \leq b$, mentre se $g(x) < 0$ il vincolo diviene $-g(x) \leq b$. Pertanto, per rappresentare il vincolo (1.19) è sufficiente imporre i seguenti due vincoli:

$$g(x) \leq b \quad , \quad -g(x) \leq b \ .$$

Vediamo ora come trattare una funzione obiettivo espressa mediante un valore assoluto. Si supponga di dover massimizzare $|f(x)|$, con $x \in X$. È sufficiente risolvere i due diversi problemi

$$\max\{\ f(x)\ :\ x \in X\ \} \quad , \quad \max\{\ -f(x)\ :\ x \in X\ \}$$

e prendere come soluzione quella che fornisce il valore più alto della funzione obiettivo.

Se $f(x)$ è una funzione lineare nella singola variabile x, cioè$f(x) = b + cx$, allora basta sostituire alla $f(x)$ la funzione lineare a tratti

$$g(x) = \begin{cases} -b - cx & x \leq -b/c \\ \ \ b + cx & x > -b/c \end{cases}$$

che può essere trattata con le tecniche che verranno spiegate nel paragrafo 1.2.11.

Esercizio 1.19 *Formulare il problema* $\min\{\ |3-4x|\ :\ |x| \leq 2\ \}$.

Esercizio 1.20 *Siano* $a_1, \ldots, a_n$ *numeri reali positivi. Partizionare tali numeri in due insiemi* I *e* J *in modo tale che le somme dei valori assegnati a ciascun sottoinsieme abbiano la minima differenza in valore assoluto.*

1.2.10.1 Minima distanza

Dato un insieme $X \subseteq \mathbb{R}^n$, si vuole determinare "il più piccolo" elemento di X, ossia l'elemento di X che minimizza una qualche norma. Questo problema ha numerose applicazioni, ad esempio in statistica ed ingegneria. La sua versione decisionale consiste nel determinare se l'insieme X contiene un elemento di norma minore o uguale a k, cioè se la sfera (nella norma prescelta) avente centro nell'origine e raggio k ha intersezione non vuota con X. Un caso particolare molto rilevante è quello in cui $k = 0$, cioè si vuole determinare se X contiene l'origine. Le norme più comunemente usate sono

$$L_1(x) = \sum_{i=1}^n |x_i| \quad , \quad L_2(x) = \sqrt{\sum_{i=1}^n x_i^2} \quad , \quad L_\infty(x) = \max\{\ |x_i|\ :\ i = 1, \ldots, n\ \} \ .$$

Nel caso in cui X sia rappresentabile attraverso vincoli lineari, che si può assumere abbiano in generale la forma $Ax \leq b$ (si veda il Paragrafo 2.1), il problema di minima distanza corrispondente alle norme L_1 o L_∞ è un problema di *PL*; infatti può essere formulato rispettivamente mediante

$$(PMD_1) \qquad \min\left\{\ \sum_{i=1}^n v_i\ :\ -v_i \leq x_i \leq v_i \ \ i = 1, \ldots, n\ ,\ Ax \leq b\ \right\}$$

$$(PMD_\infty) \qquad \min\left\{\ v\ :\ -v \leq x_i \leq v \ \ i = 1, \ldots, n\ ,\ Ax \leq b\ \right\}$$

1.2.11 Funzioni lineari a tratti

Consideriamo la seguente funzione $f(x)$:

$$f(x) = \begin{cases} b_1 + c_1 x & \text{se } x \in [a_1, a_2] \\ b_2 + c_2 x & \text{se } x \in (a_2, a_3] \end{cases}$$

dove assumiamo

$$b_2 + c_2 a_2 \geq b_1 + c_1 a_2. \tag{1.20}$$

La funzione $f(x)$ è definita nell'intervallo $[a_1, a_3]$ ed è la composizione di due funzioni lineari definite nei due sotto-intervalli $[a_1, a_2]$ e $(a_2, a_3]$ (si veda un esempio in Figura 1.7). Il caso precedente della funzione con carico fisso può essere visto come un caso particolare, in cui il primo intervallo si riduce ad un unico punto. Introduciamo due variabili booleane y_1 e y_2 con il seguente significato:

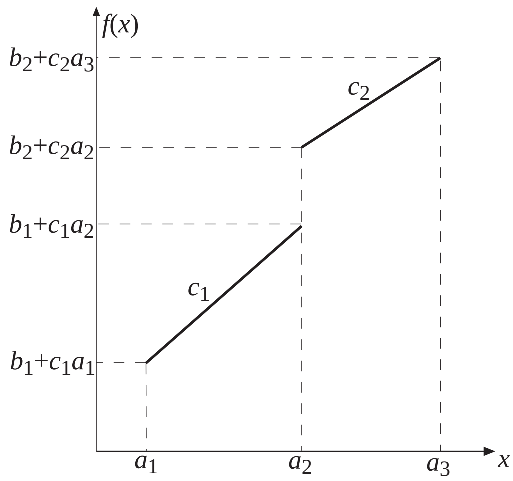


Figura 1.7: Una funzione lineare a due tratti

$$y_1 = \begin{cases} 1 & \text{se } x \in [a_1, a_2] \\ 0 & \text{altrimenti} \end{cases} \quad , \quad y_2 = \begin{cases} 1 & \text{se } x \in (a_2, a_3] \\ 0 & \text{altrimenti.} \end{cases} \ .$$

Dovendo x appartenere a uno ed uno solo dei due sottointervalli, occorre aggiungere il vincolo $y_1 + y_2 = 1$ ai vincoli $y_1, y_2 \in \{0, 1\}$; alternativamente, si potrebbe sostituire $1 - y_1$ a y_2 (le due variabili sono complementari). Si noti che se $x \in [a_1, a_2]$ possiamo porre $x = a_1 + z_1$, dove z_1 $(= x - a_1)$ non è altro che la porzione di valore di x che supera a_1, pertanto il suo valore è limitato dalle disuguaglianze $0 \leq z_1 \leq a_2 - a_1$. Analogamente, se $x \in (a_2, a_3]$, poniamo $x = a_2 + z_2$, dove z_2 ha un significato analogo a quello di z_1, con $0 < z_2 \leq a_3 - a_2$. Considerato il significato delle variabili booleane y_1 e y_2, possiamo scrivere:

$$x = a_1 y_1 + z_1 + a_2 y_2 + z_2,$$

purché siano rispettati i vincoli

$$0 \leq z_1 \leq (a_2 - a_1)y_1 \quad , \quad 0 \leq z_2 \leq (a_3 - a_2)y_2 \quad , \quad y_1 + y_2 = 1 \quad , \quad y_1, y_2 \in \{0, 1\} \quad . \tag{1.21}$$

che impongono che solo una delle due variabili z_1 e z_2 possa avere valore non nullo. Possiamo adesso esprimere la funzione non lineare $f(x)$ mediante la seguente funzione lineare

$$\begin{array}{ll} g(z_1, z_2, y_1, y_2) & = b_1 y_1 + c_1(a_1 y_1 + z_1) + b_2 y_2 + c_2(a_2 y_2 + z_2) \\ & = (b_1 + c_1 a_1)y_1 + c_1 z_1 + (b_2 + c_2 a_2)y_2 + c_2 z_2 \end{array} .$$

con le variabili soggette ai vincoli (1.21).

Discutiamo ora l'ambiguità della funzione $g(z_1, z_2, y_1, y_2)$ nel punto a_2: $x = a_2$ può essere espresso sia ponendo $z_1 = a_2 - a_1$, $y_1 = 1$, $z_2 = y_2 = 0$ che ponendo $z_1 = y_1 = z_2 = 0$, $y_2 = 1$; chiaramente solo il primo dei due casi è accettabile. Analogamente a quanto osservato per la funzione con carico fisso nel paragrafo precedente, se si vuole minimizzare $f(x)$, per l'assunzione (1.20) si ha

$$g(0, 0, 0, 1) = b_2 + c_2 a_2 \geq b_1 + c_1 a_2 = g(a_2 - a_1, 0, 1, 0)$$

e quindi, se $x = a_2$ fosse la soluzione ottima, la soluzione $z_1 = a_2 - a_1$, $y_1 = 1$, $z_2 = y_2 = 0$ risulterebbe ottima, nonostante l'ambiguità. Si noti che, se $f(x)$ fosse continua in $[a_1, a_3]$, cioè $b_2 + c_2 a_2 = b_1 + c_1 a_2$, allora non sarebbe scorretto considerare il valore a_2 in entrambi i casi (cioè il secondo intervallo sarebbe $[a_2, a_3]$); non essendovi ambiguità sul valore di $f(x)$ in a_2, in questo caso la funzione $g(z_1, z_2, y_1, y_2)$ può essere anche massimizzata.

La trasformazione presentata può essere generalizzata al caso di funzioni lineari a tratti definite su più di due intervalli:

$$f(x) = \begin{cases} b_1 + c_1 x & \text{se } x \in [a_1, a_2] \\ b_2 + c_2 x & \text{se } x \in (a_2, a_3] \\ \quad \vdots & \quad \vdots \\ b_n + c_n x & \text{se } x \in (a_n, a_{n+1}] \end{cases}$$

con condizioni analoghe a (1.20) nei punti di discontinuità. La funzione in sostituzione di $f(x)$ è

$$g(z_1, \ldots, z_n, y_1, \ldots, y_n) = \textstyle\sum_{i=1}^{n} (b_i + c_i a_i) y_i + \sum_{i=1}^{n} c_i z_i \quad ,$$

con le variabili soggette ai vincoli

$$\begin{array}{ll} 0 \leq z_i \leq (a_{i+1} - a_i) y_i & i = 1, \ldots, n \\ \sum_{i=1}^{n} y_i = 1 & \\ y_i \in \{0, 1\} & i = 1, \ldots, n \end{array} .$$

Il valore della variabile originale è dato da $x = \sum_{i=1}^{n} a_i y_i + \sum_{i=1}^{n} z_i$.

Esercizio 1.21 *Sia data la seguente funzione:*

$$f(x) = \begin{cases} 1 + 2x & \textit{se } x \in [0, 2] \\ 8 - x & \textit{se } x \in (2, 5] \\ 3 & \textit{se } x \in (5, 6] \\ 2 + x/2 & \textit{se } x \in (6, 10] \end{cases}$$

Disegnare $f(x)$ e fornire la formulazione PLI del problema che consiste nella minimizzazione di $f(x)$.

La necessità di utilizzare variabili binarie per rappresentare le funzioni lineari a tratti degli esempi precedenti deriva dalla loro *non convessità* (si veda il Capitolo 4); la formulazione "trasferisce" la non convessità del problema dalla funzione obiettivo all'insieme ammissibile per mezzo delle variabili a valori interi. Questo non è necessario qualora la funzione sia convessa, come quella mostrata in Figura 1.8.

Affinché ciò accada, devono essere verificate due condizioni:

- f deve essere continua, ossia $b_{i+1} + c_{i+1}a_{i+1} = b_i + c_i a_{i+1}$ per $i = 1, \ldots, n-1$;
- la derivata di f (nei tratti lineari) deve essere nondecrescente, ossia $c_{i+1} \geq c_i$ per $i = 1, \ldots, n-1$.

In questo caso, la minimizzazione di $f(x)$ può essere equivalentemente espressa mediante la minimizzazione di

$$g(z_1, \ldots, z_n) = b_1 + \sum_{i=1}^{n} c_i z_i$$

soggetta ai vincoli

$$0 \leq z_i \leq a_{i+1} - a_i \qquad i = 1, \ldots, n \ .$$

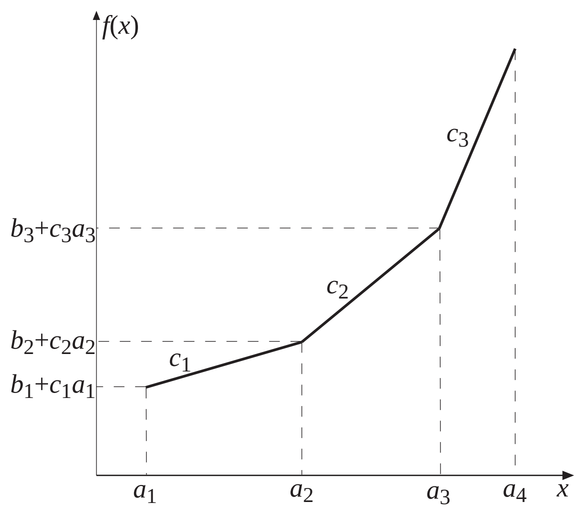


Figura 1.8: Una funzione lineare a tratti e convessa

In particolare, la variabile originale è definita da $x = a_1 + \sum_{i=1}^{n} z_i$. Infatti, se all'ottimo la variabile x assume il valore $\bar{x}$, tale valore dovrà essere "ottenuto" aumentando il valore di alcune delle variabili z_i finché la loro somma non dia $\bar{x} - a_1$. Ma siccome la derivata di f è non decrescente, è chiaramente conveniente "far crescere prima il valore della variabili z_i di indice più basso"; in altri termini, in una soluzione ottima del problema si avrà certamente che

$$z_i = \begin{cases} a_{i+1} - a_i & \text{se } i < h \\ \bar{x} - a_h & \text{se } i = h \\ 0 & \text{se } i > h \end{cases}$$

dove h è il più piccolo indice tale che $\bar{x} \geq a_h$.

Esercizio 1.22 *Dimostrare l'affermazione precedente.*

Questa proprietà non vale nel caso in cui f non sia convessa, o nel caso in cui venga massimizzata. Una formulazione senza variabili binarie, analoga a questa, è possibile per la massimizzazione—ma non per la minimizzazione—di funzioni lineari a tratti *concave*, ossia la cui derivata sia non crescente.

1.2.12 Vincoli disgiuntivi

Come vedremo più in dettaglio nel Paragrafo 2.1.1, nei modelli di *PL* (o *PLI*) si hanno un numero finito di vincoli lineari del tipo

$$A_i x \leq b_i \quad , \quad i = 1, \ldots, m \tag{1.22}$$

che individuano un *poliedro convesso*. Ad esempio, il sistema

$$\begin{array}{rcrcl} x_1 & + & x_2 & \leq & 2 \\ & & x_2 & \leq & 1 \\ x_1 & & & \leq & 2 \\ x_1 & & & \geq & 0 \\ & & x_2 & \geq & 0 \end{array}$$

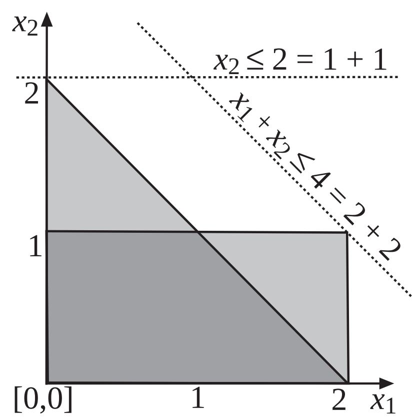


Figura 1.9: Rappresentazione di poliedri non convessi

definisce il poliedro di vertici $[0, 0]$, $[0, 1]$, $[1, 1]$ e $[2, 0]$ in Figura 1.9. Supponiamo ora che l'insieme ammissibile che si vuole rappresentare sia tutta l'area grigia: si tratta di una regione *non convessa*, che può essere rappresentata come l'*unione* (invece dell'intersezione) di due poliedri convessi, il triangolo di vertici $[0,0]$, $[2,0]$ e $[0,2]$ ed il rettangolo $[0,0]$, $[0,1]$, $[2,1]$ e $[2,0]$. Anche ora, come in casi precedenti, si presenta una scelta fra due alternative: le soluzioni ammissibili appartengono *o* al primo poliedro, *oppure* al secondo. In particolare, sono il primo ed il secondo vincolo ad essere critici, poiché tutti gli altri sono soddisfatti da tutti i punti della regione che vogliamo rappresentare; invece i punti ammissibili possono soddisfare *anche uno solo* di quei due vincoli. Si parla quindi in questo caso di *vincoli disgiuntivi.* Per rappresentare vincoli di questo tipo è naturale introdurre due variabili binarie y_1 ed y_2, con la convenzione che $y_1 = 0$ significa che x appartiene al primo insieme, e quindi il secondo vincolo può essere violato, mentre $y_2 = 0$ significa che x appartiene al secondo insieme, e quindi è il primo vincolo a poter essere violato. Possiamo quindi rappresentare l'insieme ammissibile per mezzo dei vincoli

$$\begin{array}{ccccccccc}
x_1 & + & x_2 & - & M_1y_1 & & & \leq & 2 \\
 & & x_2 & & & - & M_2y_2 & \leq & 1 \\
x_1 & & & & & & & \leq & 2 \\
x_1 & & & & & & & \geq & 0 \\
 & & x_2 & & & & & \geq & 0 \\
 & & & & y_1 & + & y_2 & \leq & 1 \\
 & & & & y_1 & , & y_2 & \in & \{0,1\}
\end{array}$$

purché M_1 ed M_2 siano numeri "sufficientemente grandi" da rendere *ridondanti* i vincoli quando la variabile binaria corrispondente assume il valore 1. Si può facilmente verificare dalla figura che in questo caso specifico è sufficiente porre $M_1 = 2$ e $M_2 = 1$. Si noti che il vincolo $y_1+y_2 \leq 1$ assicura che al più una delle variabili abbia valore 1, ossia che almeno uno dei due insiemi di vincoli sia soddisfatto; alternativamente si può usare una sola variabile binaria $y = y_1$ ponendo $y_2 = 1 - y$.

Più in generale, consideriamo il caso in cui si abbiano gli m vincoli (1.22), e che $S_1, S_2, \ldots, S_p$ siano p sottoinsiemi, non necessariamente disgiunti, dell'insieme $\{1, 2, \ldots, m\}$. Definiamo per ogni $h = 1, \ldots, p$ l'insieme $X_h = \{\ x\ :\ A_ix \leq b_i \quad i \in S_h\ \}$ di tutti i punti che soddisfano i vincoli i cui indici sono in S_h, e consideriamo l'insieme $X = X_1 \cup X_2 \cup \ldots \cup X_p$. Se tutti gli insiemi X_h sono limitati possiamo rappresentare X introducendo p variabili binarie e generalizzando l'idea vista in precedenza per il caso $p = 2$: X è l'insieme di tutti i vettori x che soddisfano i vincoli

$$\begin{array}{cccccccccl}
A_ix & - & M_1y_1 & & & & & & \leq b_i & i \in S_1 \\
A_ix & & & - & M_2y_2 & & & & \leq b_i & i \in S_2 \\
\vdots & & & & & \ddots & & & \vdots & \vdots \\
A_ix & & & & & & - & M_py_p & \leq b_i & i \in S_p \\
 & & y_1 & + & y_2 & \cdots & + & y_p & \leq p-1 & \\
 & & y_1 & , & y_2 & \cdots & , & y_p & \in \{0,1\} &
\end{array} .$$

Per ogni $h = 1, \ldots, p$, M_h è una costante tale che tutti i vincoli $A_ix \leq b_i + M_h$ per $i \in S_h$ sono ridondanti per tutti gli insiemi S_k, $k \neq h$; una tale costante esiste certamente perché tutti gli insiemi X_k sono limitati per ipotesi.

Esercizio 1.23 *Si proponga una procedura che permetta di determinare un valore opportuno per ciascuna costante M_h (suggerimento: si risolva un numero opportuno di problemi di PL).*

Una variante interessante è il caso in cui si vuole che almeno k degli insiemi di vincoli siano soddisfatti (cioè x appartenga all'intersezione di almeno k degli insiemi X_i, $i = 1, \ldots, p$). In questo caso è sufficiente sostituire al vincolo $y_1 + y_2 + \cdots + y_p \leq p - 1$ il nuovo vincolo $y_1 + y_2 + \cdots + y_p \leq p - k$.

Esercizio 1.24 *La casa di produzione di cibi in conserva "Stella" intende immettere sul mercato una nuova linea di preparati per insalate di riso, chiamati "GhiottoRiso", che possono contenere, oltre ad altri prodotti vegetali, anche funghetti rosa (nel seguito 'fr'), cipolline ovali ('co'), peperoncini piccanti ('pp') e crauti indiani ('ci'); di ciascuno di essi si deve decidere la presenza e quantità.*

La divisione marketing della Stella ha definito 6 *diversi standard qualitativi tali che il raggiungimento di ciascuno di essi farà aumentare le vendite del prodotto. Essi sono espressi come:*

$$\begin{array}{rcrcrcrcr}
2x_{fr} & + & 4x_{co} & + & x_{pp} & + & 3x_{ci} & \geq & 150 \\
x_{fr} & + & 2x_{co} & + & 5x_{pp} & + & 2x_{ci} & \geq & 95 \\
3x_{fr} & + & x_{co} & + & 2x_{pp} & + & x_{ci} & \geq & 80 \\
5x_{fr} & + & 3x_{co} & + & 3x_{pp} & + & 4x_{ci} & \geq & 200 \\
x_{fr} & + & 5x_{co} & + & 2x_{pp} & + & x_{ci} & \geq & 70 \\
4x_{fr} & + & x_{co} & + & x_{pp} & + & 4x_{ci} & \geq & 100
\end{array}$$

dove le variabili x rappresentano le quantità in grammi dei quattro prodotti che si intende mettere nel GhiottoRiso. Inoltre, dalle ultime indagini svolte sull'apprezzamento presso la clientela dei prodotti delle ditte concorrenti, si è riusciti a prevedere che, se si riescono a soddisfare almeno 3 *dei* 6 *standard di qualità, l'intera produzione sarà assorbita dal mercato.*

Il costo al grammo dei quattro prodotti in questione è $c_{fr} = 12, c_{co} = 6, c_{pp} = 15, c_{ci} = 5$. Infine, per ciascuno di essi, la massima quantità in grammi che si può immettere nel GhiottoRiso è 15 *grammi. Si vuole ottenere la composizione ottimale di GhiottoRiso, cioè quella che rispetti le indicazioni date e che abbia il minimo costo; formulare il problema e determinare una soluzione ammissibile.*

1.2.13 Un esempio di formulazione e alcuni esercizi

Concludiamo questa parte dedicata alle tecniche di modellazione con un esempio nel quale si utilizzano alcune delle tecniche precedentemente illustrate; l'uso delle tecniche potrà poi essere autonomamente sperimentato svolgendo gli esercizi proposti alla fine del paragrafo.

1.2.13.1 Dislocazione ottima di impianti

La società informatica MilanNet ha deciso di aprire nel territorio pisano sino a n possibili uffici di assistenza ai suoi m clienti. Per ogni sito $i = 1, \ldots, n$ si conosce il costo d_i di installazione e il numero massimo u_i di clienti che l'ufficio può assistere qualora sia attivato; inoltre, per ogni sito $i = 1, \ldots, n$ si conosce il costo c_{ij} derivante dalla gestione del cliente $j = 1, \ldots, m$ presso tale centro. Si vuole decidere in quali delle n località aprire gli uffici di assistenza e, per ciascuno di essi, l'insieme dei clienti assegnati, in modo tale che ogni cliente sia assegnato ad uno ed un solo ufficio di assistenza e che il costo complessivo (di installazione e gestione) sia minimo.

Per formulare tale problema occorre introdurre due insiemi di variabili binarie: le variabili y_i, $i = 1, \ldots, n$, per rappresentare la scelta relativa agli uffici da aprire, e le variabili x_{ij}, $i = 1, \ldots, n, j = 1, \ldots, m$, per assegnare i clienti agli uffici. La funzione obiettivo, da minimizzare, che include sia i costi di gestione che quelli di installazione è

$$\textstyle\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} + \sum_{i=1}^{n} d_i y_i \ \ .$$

I vincoli di semiassegnamento garantiscono che ogni cliente sia assegnato ad uno ed un solo ufficio:

$$\textstyle\sum_{i=1}^{n} x_{ij} = 1 \qquad j = 1, \ldots, m \ \ .$$

Dobbiamo poi aggiungere sia i vincoli sul numero massimo di clienti per ufficio

$$\textstyle\sum_{j=1}^{m} x_{ij} \leq u_i \qquad i = 1, \ldots, n \ \ , \tag{1.23}$$

sia quelli che garantiscono che i clienti siano assegnati ad uffici di cui sia stata decisa la costruzione:

$$x_{ij} \leq y_i \qquad j = 1, \ldots, m \quad , \quad i = 1, \ldots, n \ \ . \tag{1.24}$$

Questi ultimi esprimono l'implicazione $x_{ij} > 0 \Rightarrow y_j = 1$. Per evitare di usare mn vincoli, si può imporre che la somma delle x_{ij} per i fissato, cioè il numero di clienti assegnati al sito i, sia nulla quando $y_i = 0$ e possa assumere un valore non superiore a u_i quando $y_i = 1$, mediante i vincoli

$$\textstyle\sum_{j=1}^{m} x_{ij} \leq u_i y_i \qquad i = 1, \ldots, n \ \ .$$

Osserviamo che il vincolo relativo all'ufficio i, $i = 1, \ldots, n$, implica sia il corrispondente vincolo (1.23) che i corrispondenti vincoli (1.24). Il problema può quindi essere formulato come

$$\begin{array}{llll} \min & \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} + \sum_{i=1}^{n} d_i y_i & & \\ & \sum_{i=1}^{n} x_{ij} = 1 & j = 1, \ldots, m & i = 1, \ldots, n \\ & \sum_{j=1}^{m} x_{ij} \leq u_i y_i & & i = 1, \ldots, n \\ & y_i \in \{0, 1\} & & i = 1, \ldots, n \\ & x_{ij} \in \{0, 1\} & j = 1, \ldots, m & i = 1, \ldots, n \end{array}$$

Esercizio 1.25 *Formulare un problema di installazione ottima di al più 4 impianti con 11 clienti, dando anche i costi di installazione e di gestione e le capacità degli impianti.*

1.2.13.2 Esercizi di modellazione

Esercizio 1.26 *La Fintus produce tre tipi di patatine surgelate, denominati A, B e C. La compagnia acquista patate di due tipi diversi, denominati P_1 e P_2. I diversi tipi di prodotto usano parti diverse della patata originaria, per cui 1Kg di patate acquistato determina la produzione di una certa quantità di tutti e tre i prodotti. I rendimenti dei due tipi di patata sono diversi, come indicato nella seguente tabella:*

patata/tipo	*A*	*B*	*C*
P_1	*.2*	*.2*	*.3*
P_2	*.3*	*.1*	*.3*

Il profitto della Fintus è di .03 Euro al Kg per le patate P_1 e di .025 Euro al Kg per le patate P_1: la Fintus intende produrre non più di 6.000 Kg di A, 4.000 Kg di B e 8.000 kg di C, massimizzando il profitto. Formulare come PL *il problema di ottimizzazione corrispondente.*

Esercizio 1.27 *L'azienda Caramelli produce un olio speciale per cosmetici, ottenuto dalla raffinazione e miscelazione di oli. Gli oli si dividono in due categorie, oli vegetali ed oli non vegetali. Sono disponibili due diversi oli vegetali, che indichiamo con V_1 e V_2, e tre diversi oli non vegetali che indichiamo con N_1, N_2 e N_3. I costi (Euro/tonnellata) e la densit degli oli sono i seguenti:*

	V_1	V_2	N_1	N_2	N_3
Costo	*110*	*120*	*130*	*110*	*115*
Densità	*8.8*	*6.1*	*2.0*	*4.2*	*5.0*

Gli oli vegetali e quelli non vegetali richiedono differenti linee di produzione per la raffinazione. In ogni mese non è possibile raffinare più di 200 tonnellate di olio vegetale e 250 tonnellate di olio non vegetale. Non vi è perdita di peso nel processo di raffinamento, ed il costo di tale processo può essere ignorato. Vi è invece una restrizione tecnologica sulla densità del prodotto finale: nellunità di misura opportuna, questa deve essere compresa tra 3 e 6. Si assume che la densità degli oli si misceli nel prodotto finale in modo lineare. Il prodotto finale sarà venduto a 150 Euro/tonnellata. Formulare come PL *il problema di produrre il bene massimizzando il profitto.*

Esercizio 1.28 *Unindustria dolciaria produce 3 diversi tipi di dolci: A, B, C. Si deve stabilire il piano di produzione giornaliero dellindustria, avente una capacit produttiva massima di* 10000 *dolci al giorno, in modo che la produzione di A non ecceda il 50% della produzione globale giornaliera, e che la produzione di C sia uguale al pi al 25% della produzione di B. Sapendo che il guadagno garantito dalla produzione di un dolce di tipo A, B e C è rispettivamente di* 0.2 *Euro,* 0.1 *Euro e* 0.4 *Euro, si vuole individuare un piano di produzione che massimizzi il guadagno. Si formuli il problema come* PLI.

Esercizio 1.29 *Un'impresa ha a disposizione tre procedimenti differenti, chiamati P_1, P_2 e P_3, per la produzione di un certo bene. Per produrre una unità di tale bene sono necessarie lavorazioni tre macchine diverse, chiamate A, B e C. I tempo di lavorazione per ogni macchina necessario a produrre un'unità di bene dipende dal procedimento usato, come mostrato nella tabella seguente:*

procedimento/macchina	A	B	C
P_1	*2*	*4*	*3*
P_2	*1*	*2*	*4*
P_3	*3*	*3*	*2*

Ogni macchina è disponibile per 50 unità di tempo. Il profitto per la vendita di un'unità di bene dipende dal procedimento usato: è 15 *se si è usato il procedimento P_1,* 18 *se si è usato P_2 e* 10 *se si è usato P_3 (in Euro). Formulare come* PL *il problema di minimizzare il numero di unità di tempo di impiego della macchina B, con il vincolo che il profitto sia almeno pari a* 200.

Esercizio 1.30 *Il direttore amministrativo dell'ospedale Santa Cara deve stabilire i turni ospedalieri delle ostetriche, in modo da garantire un minimo numero di ostetriche presenti in ogni turno (indicato nella tabella). Il direttore vuole utilizzare il minor numero totale di ostetriche, tenendo conto che le ostetriche che si recano in reparto per uno dei primi cinque turni sono obbligate a lavorare per 8 ore consecutive (due turni consecutivi), mentre quelle impiegate nell'ultimo turno (turno 6) lavorano solo 4 ore. Si formuli il problema come* PLI.

Turno	*1*	*2*	*3*	*4*	*5*	*6*
Orario	*6 - 10*	*10 - 14*	*14 - 18*	*18 - 22*	*22 - 2*	*2 - 6*
N. ostetriche	*70*	*80*	*50*	*60*	*40*	*30*

Esercizio 1.31 *Sia data la matrice* 3×3 *di numeri reali in figura, in cui sono anche indicate le somme degli elementi di ogni riga e di ogni colonna. Si vuole arrotondare ogni elemento della matrice o alla sua parte intera inferiore, oppure alla sua parte intera superiore. Lo stesso procedimento di arrotondamento deve essere applicato alla somma degli elementi di ogni riga ed alla somma degli elementi di ogni colonna. Si vogliono eseguire tali operazioni di arrotondamento in modo che, nel problema trasformato, la somma degli elementi arrotondati in ogni riga (colonna) sia uguale alla rispettiva somma di riga (colonna) arrotondata. Si formuli il problema come* PLI.

3.1	*6.8*	*7.3*	*17.2*
9.6	*2.4*	*0.7*	*12.7*
3.6	*1.2*	*6.5*	*11.3*
16.3	*10.4*	*14.5*	*somme*

Esercizio 1.32 *Un villaggio ha 7 abitanti $\{a_1, \ldots, a_7\}$, 4 clubs politici $\{C_1, \ldots, C_4\}$, e 3 partiti politici $\{P_1, \ldots, P_3\}$. Ogni abitante è membro di almeno un club, ed è iscritto ad un solo partito politico. Più precisamente, i clubs hanno i seguenti membri: $C_1 = \{a_1, a_2\}$; $C_2 = \{a_2, a_3, a_4\}$; $C_3 = \{a_4, a_5\}$; $C_4 = \{a_4, a_5, a_6, a_7\}$, mentre i partiti hanno i seguenti iscritti: $P_1 = \{a_1, a_2\}$; $P_2 = \{a_3, a_4\}$; $P_3 = \{a_5, a_6, a_7\}$. Ogni club deve nominare uno dei suoi membri come rappresentante al consiglio del villaggio, costituito da 4 membri. Formulare il problema di decidere se sia possibile costituire un consiglio del villaggio con la propriet che al più un membro appartenga a P_2 e al più un membro appartenga a P_3.*

Esercizio 1.33 *Il C.T. Tramattoni, dopo la clamorosa esclusione dell'Itaglia dalla Coppa del Tondo, decide di rivolgersi ad un esperto di Ricerca Operativa per le prossime partite per i Campionati della Moneta Unica. Per le partite di qualificazione ha già deciso di avvalersi di una rosa di n giocatori; di ciascun giocatore i, $i = 1, \ldots, n$, si conosce la bravura b_i e il ruolo che può ricoprire in campo (uno e uno solo per giocatore). Gli n giocatori sono partizionati negli insiemi P dei portieri, D dei difensori, C dei centrocampisti e A degli attaccanti. Si dovrà far giocare almeno un giocatore per ciascun ruolo* *ma non più di 1 portiere, 6 difensori, 5 centrocampisti e 4 attaccanti. Tramattoni fornisce all'esperto anche la lista L, $|L| = m$, delle coppie di giocatori che non potranno giocare assieme per incompatibilità tattiche e/o caratteriali. Egli richiede all'esperto di aiutarlo a definire una formazione di 11 giocatori che rispetti sia le limitazioni sui ruoli sia le incompatibilità e che massimizzi la bravura complessiva, data dalla somma delle bravure dei singoli giocatori. Svolgere il ruolo dell'esperto di Ricerca Operativa formulando come* PLI *il problema per Tramattoni.*

Esercizio 1.34 *La commissione arbitri di calcio ha deciso di formare le terne (un arbitro più due guardialinee) in modo automatico per eliminare sospetti di "combine". Inoltre, per rispettare la legge sulle pari opportunità, la commissione ha deciso che ogni coppia di guardialinee sia formata da un uomo e una donna. Per le n partite in programma domenica prossima sono a disposizione a arbitri (maschi e femmine), m guardialinee maschi e f guardialinee femmine (con $a > n$, $m > n$, $f > n$). Il valore p_i indica la qualità (esperienza, capacità, ...) dellarbitro o del/della guardialinee i; il valore di una terna è la somma dei valori delle tre persone che formano la terna stessa. Per evitare che si formino terne troppo difformi tra loro, la commissione decide di formare n terne in modo che sia minima la differenza tra il massimo e il minimo valore delle terne. Formulare il problema della commissione come problema di* PLI.

Esercizio 1.35 *Il partito del Limone si sta attivando per le elezioni europee e, per formare la lista dei candidati, ha a disposizione n volontari. Dopo un rapido sondaggio tra i limoncini, si valuta che il volontario i può ricevere c_i preferenze, $i = 1, \dots, n$; inoltre il Capolimone conosce l'insieme $D(i)$ dei volontari in grado di collaborare col volontario i, $i = 1, \dots, n$. Dall'insieme dei volontari si vuole selezionare una lista L di candidati tale che ognuno sia in grado di collaborare con gli altri e che la somma delle preferenze ricevute sia massima. Formulare il problema in termini di* PLI.

Esercizio 1.36 *Giro e Tond, i due maggiori produttori di automobili europei, hanno deciso di effettuare una fusione. Ciò comporta una gestione comune degli impianti delle due aziende, che dovranno produrre gli stessi modelli. Indichiamo con I e J rispettivamente gli insiemi degli impianti della Giro e della Tond, con K l'insieme dei mercati in cui dovrà operare la nuova azienda, la GiroTond, con b_k, $k \in K$, la domanda del k-esimo mercato e con c_{ik}, $i \in I \cup J$, $k \in K$, il costo unitario di trasporto dall'impianto i al mercato k. Si vuole assegnare ogni mercato ad uno ed uno solo degli impianti, chiudendo gli impianti in eccesso. Formulare, come problema di* PLI, *il problema dell'assegnamento degli impianti ai mercati, con l'obiettivo di minimizzare i costi di trasporto ed il vincolo che almeno il 50% degli impianti di ciascuno dei due produttori rimanga aperto.*

Riferimenti Bibliografici

F.S. Hillier, G.J. Lieberman, "**Introduzione alla ricerca operativa**", Franco Angeli, Milano (1999).

F. Maffioli, "**Elementi di programmmazione matematica**", Casa Editrice Ambrosiana, Milano (2000).

M. Pappalardo, M. Passacantando, "**Ricerca Operativa**", Edizioni Plus, Pisa (2010).

A. Sassano, "**Modelli e algoritmi della ricerca operativa**", Franco Angeli, Milano (1999).

E. Specht "**The best known packings of equal circles in a square (up to N = 10000)**"
`http://hydra.nat.uni-magdeburg.de/packing/csq/csq.html`

C. Vercellis, "**Modelli e decisioni**", Progetto Leonardo, Bologna (1997).

Capitolo 2

Programmazione Lineare

In questo capitolo ci occuperemo di una classe particolarmente importante di problemi di ottimizzazione, ossia i problemi di *Programmazione Lineare* (*PL*). Questi problemi sono caratterizzati dal fatto che tutte le relazioni (vincoli) tra le quantità in gioco (variabili), come pure la funzione obiettivo, sono lineari; inoltre, le variabili non sono vincolate ad assumere insiemi discreti di valori (ad esempio solamente valori interi), ma possono assumere valori reali. Per quanto l'assunzione di linearità nei fenomeni rappresentati dal modello possa apparire piuttosto restrittiva, questa classe di problemi ha un forte interesse pratico. Infatti, in molte situazioni reali i componenti di un sistema reagiscono in modo almeno approssimativamente lineare alle decisioni prese, quindi molti problemi reali possono essere modellati con sufficiente precisione in termini di Programmazione Lineare. Inoltre, per questa classe di problemi sono disponibili algoritmi risolutivi efficienti che consentono di risolvere istanze di dimensione elevata (fino a centinaia di migliaia o milioni di vincoli e variabili) anche su computer di potenza limitata. Infine, molti approcci per problemi complessi (si vedano i Capitoli 4 e 7) sono basati sulla risoluzione di un numero, spesso elevato, di sottoproblemi di *PL* che "approssimano" il problema.

2.1 Problemi di Programmazione Lineare

Un problema di Programmazione Lineare è un problema di ottimizzazione (di massimo o di minimo) caratterizzato dalle seguenti proprietà:

1. la funzione obiettivo $c(x) : \mathbb{R}^n \to \mathbb{R}$ è lineare, ovvero tale che $c(\alpha x + \beta y) = \alpha c(x) + \beta c(y)$ per ogni $x, y \in \mathbb{R}^n$ e per ogni $\alpha, \beta \in \mathbb{R}$; ricordiamo che $c(x)$ è una funzione lineare se e solo se esiste $c \in \mathbb{R}^n$ tale che $c(x) = cx$.

2. l'insieme ammissibile è definito da un insieme finito di vincoli lineari del tipo $ax = b$ e/o $ax \leq b$ e/o $ax \geq b$, dove $a \in \mathbb{R}^n$ e $b \in \mathbb{R}$.

I vincoli di un problema di *PL* possono quindi essere sia di uguaglianza che di disuguaglianza, e questi ultimi possono essere sia di maggiore od uguale che di minore od uguale. Nell'analisi dei problemi di *PL* conviene però adottare una qualche forma standard in cui tutti i vincoli abbiano lo stesso formato; nel seguito, utilizzeremo principalmente la forma standard

$$\max \{ \, cx \; : \; Ax \leq b \, \} \tag{2.1}$$

in cui A è una matrice reale $m \times n$, $b \in \mathbb{R}^m$ e $x \in \mathbb{R}^n$.

Esempio 2.1:
Per il problema della *Pintel*, descritto nel paragrafo 1.2.1.1, si ha

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 1 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \quad , \quad b = \begin{bmatrix} 4 \\ 7 \\ 9 \\ 0 \\ 0 \end{bmatrix} \quad , \quad c = \begin{bmatrix} 500 \, , \, 200 \end{bmatrix} \quad .$$

Si noti che, data la forma che abbiamo scelto per il problema, i vincoli di non negatività sono incorporati nella matrice A; inoltre, per semplicità di notazione abbiamo scalato di un fattore 100000 i termini noti (questo corrisponde a considerare come unità un lotto di 100000 processori).

Assumere che un problema di PL sia dato nella forma (2.1) non comporta alcuna perdita di generalità, in quanto qualsiasi problema di PL può essere agevolmente ricondotto alla forma (2.1) introducendo oppure eliminando vincoli e/o variabili per mezzo delle seguenti equivalenze:

$$
\begin{array}{rlcl}
i) & \max \sum_{j=1}^{n} c_j x_j & \equiv & -\min \sum_{j=1}^{n} (-c_j) x_j \\
ii) & \sum_{j=1}^{n} a_{ij} x_j = b_i & \equiv & \left\{ \begin{array}{lcl} \sum_{j=1}^{n} a_{ij} x_j & \leq & b_i \\ \sum_{j=1}^{n} a_{ij} x_j & \geq & b_i \end{array} \right. \\
iii) & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \equiv & \sum_{j=1}^{n} (-a_{ij}) x_j \leq -b_i \\
iv) & \sum_{j=1}^{n} a_{ij} x_j \geq b_i & \equiv & \sum_{j=1}^{n} a_{ij} x_j - s_i = b_i, \quad s_i \geq 0 \\
v) & \sum_{j=1}^{n} a_{ij} x_j \leq b_i & \equiv & \sum_{j=1}^{n} a_{ij} x_j + s_i = b_i, \quad s_i \geq 0
\end{array}
\tag{2.2}
$$

Le variabili s_i in (2.2.iv) e (2.2.v) sono dette *variabili di scarto*, e sono variabili “nuove”, introdotte appositamente nel modello, che appaiono solamente nei vincoli mostrati e non appaiono (ossia hanno coefficiente zero) in nessun altro vincolo e nella funzione obiettivo; le variabili x_i sono invece dette *variabili strutturali*, per evidenziare il loro scaturire dalla struttura del problema e non da manipolazioni algebriche effettuate sul modello. Infine, è sempre possibile trasformare un problema in cui una variabile x_i non è vincolata in segno in uno equivalente in cui al posto di x_i appaiono due variabili vincolate in segno, che possiamo indicare con x_i^+ e x_i^-, tramite la trasformazione algebrica

$$
x_i = x_i^+ - x_i^- \quad , \quad x_i^+ \geq 0 \quad , \quad x_i^- \geq 0 \ . \tag{2.3}
$$

Esempio 2.2: Trasformazioni equivalenti
Si consideri il problema della Fonderia del §1.2.1.2:

$$
\begin{array}{rrcrcrcrcl}
\min & 0.025x_1 & + & 0.030x_2 & + & 0.018x_3 & + & 10x_4 & & \\
 & 4x_1 & + & x_2 & + & 0.6x_3 & & & \geq & 3250 \\
 & 4x_1 & + & x_2 & + & 0.6x_3 & & & \leq & 5500 \\
 & 0.45x_1 & + & 0.5x_2 & + & 0.4x_3 & + & 100x_4 & \geq & 450 \\
 & x_1 & + & x_2 & + & x_3 & + & x_4 & = & 1000 \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & \geq & 0
\end{array}
$$

Il problema può essere portato in forma (2.1) applicando la trasformazione (2.2.i) alla funzione obiettivo, la trasformazione (2.2.ii) al quarto vincolo, la trasformazione (2.2.iii) al primo ed al terzo vincolo, ed introducendo vincoli espliciti per la non negatività delle variabili. Ciò porta alla formulazione

$$
\begin{array}{rrcrcrcrcl}
\max & -0.025x_1 & - & 0.030x_2 & - & 0.018x_3 & - & 10x_4 & & \\
 & -4x_1 & - & x_2 & - & 0.6x_3 & & & \leq & -3250 \\
 & 4x_1 & + & x_2 & + & 0.6x_3 & & & \leq & 5500 \\
 & -0.45x_1 & - & 0.5x_2 & - & 0.4x_3 & - & 100x_4 & \leq & -450 \\
 & x_1 & + & x_2 & + & x_3 & + & x_4 & \leq & 1000 \\
 & -x_1 & - & x_2 & - & x_3 & - & x_4 & \leq & -1000 \\
 & -x_1 & & & & & & & \leq & 0 \\
 & & - & x_2 & & & & & \leq & 0 \\
 & & & & - & x_3 & & & \leq & 0 \\
 & & & & & & - & x_4 & \leq & 0.
\end{array}
$$

Si ottiene quindi

$$
c = \left[\begin{array}{cccc} -0.025 & -0.030 & -0.018 & -10 \end{array} \right]
$$

$$
A = \left[\begin{array}{rrrr}
-4 & -1 & -0.6 & 0 \\
4 & 1 & 0.6 & 0 \\
-0.45 & -0.5 & -0.4 & -100 \\
1 & 1 & 1 & 1 \\
-1 & -1 & -1 & -1 \\
-1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 \\
0 & 0 & 0 & -1
\end{array} \right] , \qquad
b = \left[\begin{array}{c}
-3250 \\ 5500 \\ -450 \\ 1000 \\ -1000 \\ 0 \\ 0 \\ 0 \\ 0
\end{array} \right] .
$$

Si noti che tramite le equivalenze (2.2) e (2.3) è anche possibile trasformare un problema avente forma (2.1) in un altro equivalente di forma diversa. Ad esempio, si potrebbe alternativamente utilizzare come forma standard la seguente:

$$\min\{\ cx\ :\ Ax = b\ ,\ x \geq 0\ \}\ . \tag{2.4}$$

Esempio 2.3:
Introducendo delle variabili di scarto ed esplicitando i vincoli di non negatività, il problema della Pintel può essere posto nella forma (2.4) con

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 & 1 \end{bmatrix} \quad , \quad b = \begin{bmatrix} 4 \\ 7 \\ 9 \end{bmatrix} \quad , \quad c = [\ -500\ ,\ -200\ ]\ .$$

In questa nuova formulazione le ultime tre componenti del vettore $x \in \mathbb{R}^5$ sono le variabili di scarto.

Se il numero di variabili è pari a due oppure tre, un problema di *PL* può essere descritto (e risolto) mediante una sua rappresentazione geometrica nel piano o nello spazio. In Figura 2.1 ciò è mostrato per il problema della *Pintel*: avendo indicato con x_1 il numero di Pintium e con x_2 il numero di Coloron prodotti, l'area evidenziata col tratteggio rappresenta *l'insieme ammissibile*, o regione ammissibile, del problema, cioè l'insieme di tutti i punti che soddisfano i vincoli. Si tratta dell'intersezione di un certo numero di semipiani, uno per ogni vincolo: un insieme di questo tipo viene detto *poliedro* (*politopo* nel caso particolare in cui sia limitato, come nell'esempio che stiamo considerando). Nel seguito vedremo che tale insieme è convesso; infatti, si usa spesso il termine *poliedro convesso* per caratterizzare l'insieme ammissibile di un problema di Programmazione Lineare. Nella figura sono evidenziati i vincoli per mezzo delle rette luogo dei punti che li soddisfano come uguaglianze: tali rette costituiscono la frontiera dei semipiani che essi definiscono. I vincoli di non negatività sono individuati dagli assi cartesiani. Le rette corrispondenti ai vincoli individuano nel poliedro delle *facce* e dei *vertici*: sono vertici ad esempio i punti $[\,4\,,\,1\,]$ e $[\,1\,,\,7\,]$, ed è una faccia il segmento che li unisce. I vertici e le facce giocano un ruolo particolarmente importante nella Programmazione Lineare: dimostreremo infatti che, sotto opportune ipotesi, se l'insieme delle soluzioni ottime di un problema di *PL* è non vuoto, allora almeno una soluzione ottima si trova in corrispondenza ad un vertice; inoltre, se un punto interno ad una faccia è soluzione ottima del problema, allora tutti i punti della faccia sono soluzioni ottime. La verità di queste proprietà per il caso particolare in esame può essere facilmente compresa esaminando la figura. A questo scopo consideriamo la retta $500x_1 + 200x_2 = z$: essa definisce l'insieme delle soluzioni (eventualmente non ammissibili) che hanno valore della funzione obiettivo uguale a z. In figura sono indicate tre di tali rette, corrispondenti ai valori 1000, 2200 e 3000: al crescere di z le rette vengono traslate muovendosi nella direzione definita dal vettore $c = [\,500\,,\,200\,]$, *gradiente* della funzione obiettivo. Chiaramente, per ogni valore prefissato di z è possibile realizzare un profitto pari a quel valore se e solo se la corrispondente retta ha intersezione non vuota con la regione ammissibile: nel nostro esempio, ciò accade per $z = 1000$ e $z = 2200$, ma non per $z = 3000$. Pertanto, per trovare una soluzione ottima del nostro problema, basta traslare la retta nella direzione del gradiente fintanto che l'intersezione con la regione ammissibile si mantiene non vuota. Nel nostro caso, il massimo valore attribuibile a z è 2200: per tale valore, l'intersezione tra la retta e l'insieme ammissibile si riduce ad un solo punto, il vertice $[\,4\,,\,1\,]$, che è pertanto l'unica soluzione ottima del problema.

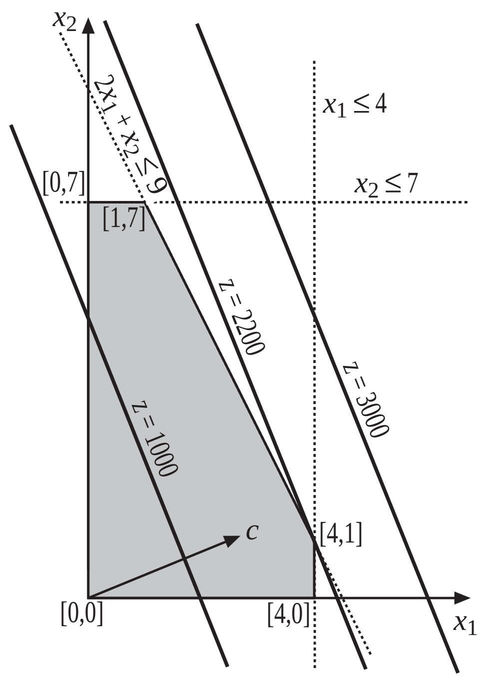
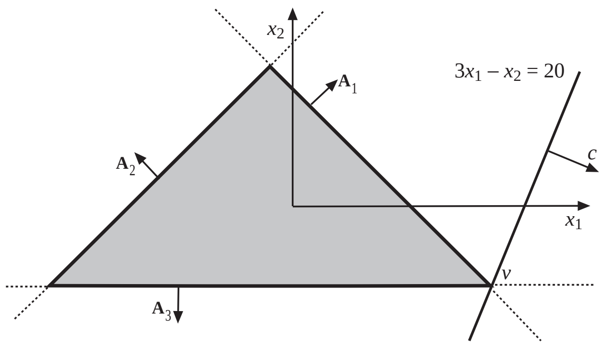


Figura 2.1: Il problema della Pintel

Nel nostro esempio la regione ammissibile del problema è limitata; in generale, però, la regione ammissibile di un problema di *PL* può essere non limitata lungo alcune direzioni. In questi casi, a seconda della direzione del gradiente della funzione obiettivo, possono esistere direzioni lungo le quali è possibile spostarsi mantenendo l'ammissibilità e facendo crescere il valore della funzione obiettivo senza mai raggiungere un valore massimo. Ad esempio, se nel problema della Pintel non ci fossero il secondo ed il terzo vincolo, potremmo fare crescere all'infinito il valore di z senza mai trovarne uno per cui la retta $500x_1 + 200x_2 = z$ abbia intersezione vuota con la regione ammissibile: questo è un caso di problema illimitato. Un caso in un certo senso opposto è quello in cui alcuni dei vincoli sono tra loro incompatibili, per cui la regione ammissibile risulta essere vuota.

Esempio 2.4: Rappresentazione geometrica della *PL*
Si consideri il problema di *PL*

$$\begin{array}{rrrcl} \max & 3x_1 & -x_2 & & \\ & x_1 & +x_2 & \leq & 4 \\ & -x_1 & +x_2 & \leq & 5 \\ & & -x_2 & \leq & 2 \end{array} \qquad (2.5)$$



rappresentato geometricamente qui accanto. Mediante la trasformazione 2.2(v), il problema può essere trasformato nel problema equivalente

$$\begin{array}{rrrrrrcl} \max & 3x_1 & -x_2 & & & & & \\ & x_1 & +x_2 & +s_1 & & & = & 4 \\ & -x_1 & +x_2 & & +s_2 & & = & 5 \\ & & -x_2 & & & +s_3 & = & 2 \\ & & & s_1, & s_2, & s_3 & \geq & 0 \end{array} .$$

Nella rappresentazione geometrica in figura, le soluzioni ammissibili del problema appartengono al poligono evidenziato (regione ammissibile). Le variabili di scarto sono associate ai vincoli e definiscono la retta di supporto del lato del poligono ($s_i = 0$) e il semipiano ammissibile corrispondente al vincolo ($s_i \geq 0$). Nella figura è indicato il gradiente della funzione obiettivo, c, e la sua curva di livello, cioè l'insieme dei punti tali che $3x_1 - x_2 = z$, per un dato valore z: la soluzione ottima del problema è data dal punto v in figura, cui corrisponde la soluzione $x_1 = 6$, $x_2 = -2$, $s_1 = 0$, $s_2 = 13$, $s_3 = 0$.

Esercizio 2.1 *Costruire ulteriori esempi di* PL *nel piano. In particolare, fornire problemi per cui risulti rispettivamente: regione ammissibile vuota, problema illimitato, almeno due soluzioni ottime.*

2.1.1 Geometria della Programmazione Lineare

In questo paragrafo introdurremo in maniera formale i concetti geometrici che permettono di caratterizzare l'insieme delle soluzioni ammissibili di un problema di Programmazione Lineare. In un primo momento ci serviremo, come nella trattazione precedente, di esempi geometrici in due o tre dimensioni per introdurre alcuni concetti in modo solamente intuitivo, riservandoci di darne definizioni e dimostrazioni formali nel seguito.

Poliedri e coni

Nello spazio vettoriale $\mathbb{R}^n$, un vincolo lineare del tipo $A_i x = b_i$ individua l'*iperpiano* $P_i = \{\, x : A_i x = b_i \,\}$, che costituisce la *frontiera* del semispazio $S_i = \{\, x : A_i x \leq b_i \,\}$[1] individuato dal vincolo $A_i x \leq b_i$. Un insieme P è un *poliedro* se è esprimibile come intersezione di un numero finito m di semispazi, cioè se esistono una matrice $A \in \mathbb{R}^{m \times n}$ ed un vettore $b \in \mathbb{R}^m$ per cui

$$P = \{\, x \,:\, Ax \leq b \,\} .$$

[1]Qui e nel seguito, con una piccola imprecisione di linguaggio useremo il termini iperpiano e semispazio per indicare o un iperpiano/semispazio propriamente detto ($b_i = 0$) oppure una sua traslazione, cioè un iperpiano/semispazio affine ($b_i \neq 0$).

Esempio 2.5: Vincoli e facce
In Figura 2.2 è mostrato ancora una volta il poliedro corrispondente all'insieme ammissibile del problema della Pintel. Ciascuna linea tratteggiata indica un iperpiano (retta) corrispondente ad uno dei vincoli $A_i x \leq b_i$, $i = 1, \ldots, 5$ del problema. Nella figura, a ciascun iperpiano è associato il vettore A_i (la corrispondente riga della matrice dei coefficienti), che è perpendicolare all'iperpiano e "punta" verso il semispazio (semipiano) in cui il vincolo è violato. Si noti come ciascun vincolo del problema sia in questo modo associato ad uno dei "lati" (facce) del poliedro.

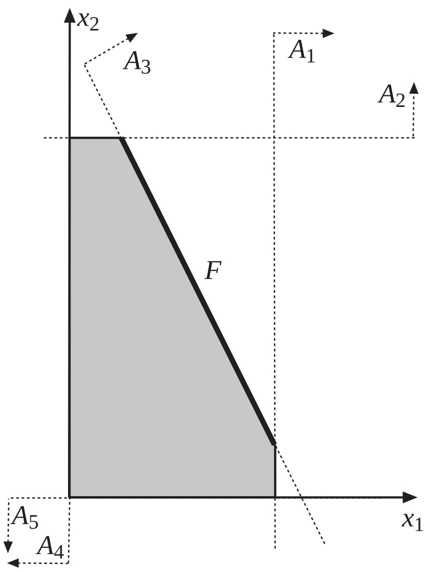


Figura 2.2: Vincoli e facce

Una delle principali proprietà dei poliedri è la loro *convessità*, dove ricordiamo che un insieme C è detto convesso se, comunque presi due punti x ed y appartenenti a C, il segmento avente x ed y per estremi è contenuto in C, cioè

$$\alpha x + (1-\alpha)y \in C \qquad \forall \alpha \in [0,1] \ .$$

Un semispazio è un insieme convesso, ed essendo l'intersezione di insiemi convessi a sua volta un insieme convesso, anche un poliedro è un insieme convesso.

Esercizio 2.2 *Si dimostrino le affermazioni precedenti.*

Una definizione alternativa utilizza l'*inviluppo convesso* di insieme finito $X = \{x^1, \ldots, x^s\} \subset \mathbb{R}^n$, ossia

$$conv(X) \ = \ \left\{ \, x = \sum_{i=1}^{s} \lambda_i x^i \ : \ \sum_{i=1}^{s} \lambda_i = 1 \ , \ \lambda_i \geq 0 \quad i = 1, \ldots, s \, \right\} \ ;$$

un insieme è convesso se contiene l'inviluppo convesso di ciascun sottoinsieme finito dei suoi punti.

Esempio 2.6:
Si consideri il poliedro $P \subset \mathbb{R}^3$ definito dal seguente insieme di vincoli:

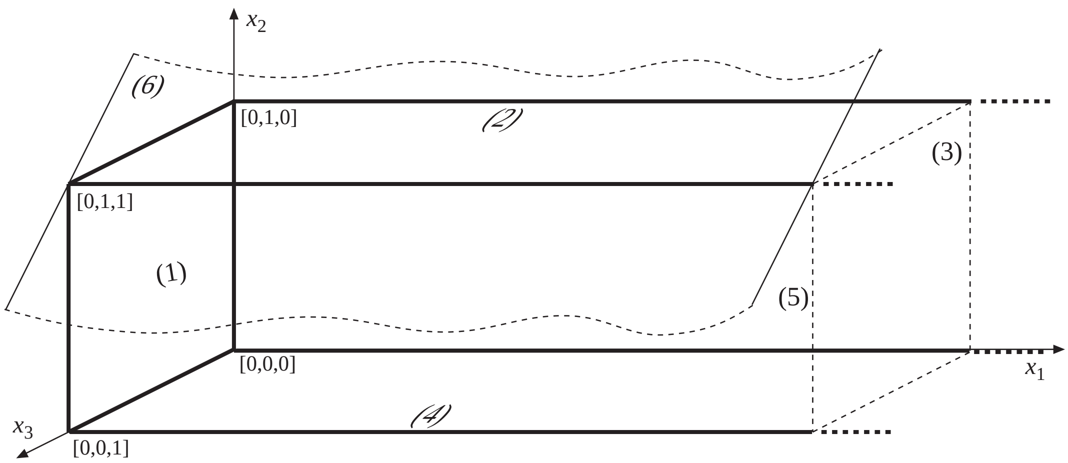


Figura 2.3: Un poliedro in $\mathbb{R}^3$

$$\begin{array}{rrrcll} -x_1 & & & \leq & 0 & (1) \\ & x_2 & & \leq & 1 & (2) \\ & & -x_3 & \leq & 0 & (3) \\ & -x_2 & & \leq & 0 & (4) \\ & & x_3 & \leq & 1 & (5) \\ & x_2 \ + & x_2 & \leq & 2 & (6) \end{array} \quad , \quad A = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \quad , \quad b = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 2 \end{bmatrix} .$$

Il poliedro è il "parallelepipedo illimitato" rappresentato in Figura 2.3. In figura sono indicati i sei iperpiani che delimitano la frontiera dei sei semispazi corrispondenti ai sei vincoli: ad esempio, il semispazio $x_1 = 0$ corrisponde al piano verticale $x_2 x_3$, che include il "lato sinistro" del parallelepipedo. Anche in questo caso, i vincoli hanno una certa corrispondenza con le "facce" del poliedro. Ad esempio, il vincolo (1) individua il "lato sinistro" del parallelepipedo; questo può essere facilmente verificato considerando che qualora si rimuovesse il vincolo (1) si otterrebbe un "parallelepipedo infinito" avente come facce i quattro "rettangoli infiniti" corrispondenti al "lato superiore", al "lato posteriore", al "lato inferiore" ed al "lato anteriore" del parallelepipedo, "prolungati all'infinito a sinistra". In altri termini, la rimozione del vincolo fa scomparire la faccia corrispondente dal poliedro. Considerazioni analoghe possono essere fatte per i vincoli (2) ("lato superiore"), (3) ("lato posteriore"), (4) ("lato inferiore") e (5) ("lato anteriore"). Un caso diverso è quello del vincolo (6): la sua rimozione non cambia il poliedro, ossia il vincolo è *ridondante*. In effetti il vincolo non è associato ad alcuna faccia, nel senso intuitivo usato fin qui, del poliedro; vedremo in seguito che è comunque associato ad una faccia di dimensione minore (in particolare, allo spigolo "di fronte in alto").

Oltre alle facce individuate dai vincoli (che in seguito definiremo come faccette o facce massimali), elementi geometricamente caratterizzanti un poliedro sono i suoi *vertici* (facce minimali). La definizione formale che corrisponde al concetto intuitivo è quella di *punto estremo*

$$x \in conv(\ \{\ x'\ ,\ x"\ \}\ ) \quad , \quad x' \in P \quad , \quad x" \in P \qquad \Longrightarrow \qquad x' = x" \quad ,$$

ossia x è punto estremo se non può essere espresso come combinazione convessa di due punti diversi di P. Ad esempio, è facile verificare che il poliedro P di Figura 2.3 ha quattro vertici: $[\ 0\ ,\ 0\ ,\ 0\ ]$, $[\ 0\ ,\ 1\ ,\ 0\ ]$, $[\ 0\ ,\ 0\ ,\ 1\ ]$ e $[\ 0\ ,\ 1\ ,\ 1\ ]$. È immediato però osservare dall'esempio che i vertici non consentono di descrivere completamente il poliedro; i quattro "spigoli illimitati" di P hanno sì origine nei vertici, ma non terminano in vertici (si può pensare che lo facciano in "vertici all'infinito"). Per rappresentare formalmente questa proprietà si introduce il concetto di *direzione* (di recessione) di un poliedro, dove una direzione di un poliedro P è un qualsiasi vettore $d \in \mathbb{R}^n$ tale che

$$\forall\, x \in P \ \ \forall \lambda \geq 0 \qquad (x + \lambda d) \in P \ \ .$$

In particolare, il poliedro di Figura 2.3 ha una sola direzione, $[\ 1\ ,\ 0\ ,\ 0\ ]$, che "identifica" i quattro spigoli illimitati.

Vogliamo adesso studiare formalmente le principali proprietà dei poliedri; per questo conviene introdurne la "forma particolarmente semplice"

$$C \ = \ \{\ x\ :\ Ax \leq 0\ \} \tag{2.6}$$

in cui il lato destro dei vincoli è tutto nullo. Ciò implica immediatamente

$$x \in C \quad , \quad \alpha \geq 0 \qquad \Longrightarrow \qquad \alpha x \in C \ \ ; \tag{2.7}$$

un insieme con questa proprietà è detto *cono*, e dato che C è anche un insieme convesso, si parlerà di un *cono convesso*. Si noti che questo non è implicito nella definizione di cono, come mostrato in in Figura 2.4: A non è convesso, mentre B lo è. Poiché C è un cono (convesso) ed anche un poliedro, viene detto *cono poliedrico*.

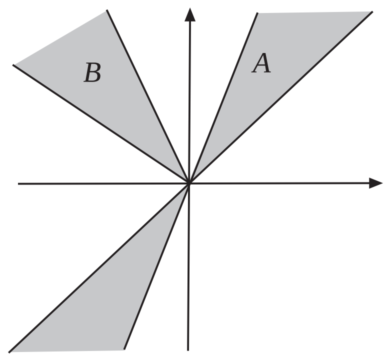


Figura 2.4: Coni convessi e non

È facile verificare che una definizione alternativa a (2.7), per il caso convesso, è

$$x, y \in C \quad , \quad \lambda, \mu \in \mathbb{R}_+ \qquad \Longrightarrow \qquad \lambda x + \mu y \in C \ \ . \tag{2.8}$$

La relazione (2.8) utilizza il concetto di *inviluppo conico* di due vettori; analogamente al caso dell'inviluppo convesso, questo può essere esteso ad un insieme finito $V = \{v^1, \ldots, v^t\} \subset \mathbb{R}^n$ come

$$cono(V) \ = \ \left\{\ v = \sum_{i=1}^{t} \nu_i v^i \ :\ \nu_i \geq 0 \quad i = 1, \ldots, t\ \right\} \ . \tag{2.9}$$

Possiamo adesso caratterizzare l'insieme $rec(P)$ di tutte le direzioni (di recessione) di P.

Teorema 2.1 $rec(P) \ = \ \{\ d\ :\ Ad \leq 0\ \}$.

Dimostrazione È immediato verificare, direttamente dalla definizione, che se $Ad \leq 0$ allora d è una direzione di recessione. Viceversa, assumiamo che $d \in rec(P)$, ossia

$$A(x + \lambda d) \leq b \quad \Longrightarrow Ad \leq (b - Ax)/\lambda \qquad \forall \lambda > 0 \ ;$$

poiché $x \in P \Longrightarrow b - Ax \geq 0$, si ha $Ad \leq 0$. $\diamond$

In altri termini, azzerando il lato destro dei vincoli si "cattura il comportamento all'infinito" di un poliedro. Si noti infine che $rec(P)$ non è mai vuoto: chiaramente, 0 vi appartiene sempre. D'altro canto, 0 non è una direzione; quindi, si ha che o $rec(P) = \{ 0 \}$, oppure esiste un $0 \neq d \in rec(P)$. È possibile verificare che questo caratterizza univocamente la *compattezza* di P: esiste una sfera di raggio finito che contiene interamente P se e solo se $rec(P) = \{ 0 \}$. Infatti, se $0 \neq d \in rec(P)$ allora P non è compatto: per ogni M grande a piacere esiste $x \in P$ tale che $\|x\| \geq 0$ (l'implicazione opposta è un ovvio corollario di risultati che saranno mostrati nel seguito). Un caso particolare, come vedremo, è quello in cui per una stessa d risulti

$$d \in rec(P) \quad , \quad -d \in rec(P) \ ;$$

un vettore siffatto si definisce *direzione di linealità* di P. Ad esempio, rimuovendo il vincolo (1) dal poliedro di Figura 2.3 si ottiene che $\left[\ 1\ ,\ 0\ ,\ 0\ \right]$ è una direzione di linealità per il poliedro risultante. È un immediato corollario del Teorema 2.1 che d è una direzione di linealità se e solo se $Ad = 0$. È anche immediato verificare che *se P ha direzioni di linealità allora non può avere vertici*; anche questo è confermato dall'esempio.

Le direzioni di P sono importanti nella nostra trattazione in quanto strettamente collegate al caso in cui il problema di *PL* corrispondente sia superiormente illimitato. Infatti, è immediato verificare che se esiste $x \in P$ e $d \in rec(P)$ tale che $cd > 0$, allora il valore della funzione obiettivo dei punti della semiretta $x + \lambda d$ per $\lambda \geq 0$ (interamente contenuta in P) è una funzione crescente di λ che non ammette estremo superiore finito. Quindi, determinare $d \in rec(P)$ tale che $cd > 0$ (e verificare che $P \neq \emptyset$) significa nei fatti aver risolto il problema di *PL*. Determinare se esiste oppure no una siffatta direzione sarebbe "facile" se il cono $rec(P)$, invece che nella forma (2.6), fosse dato nella forma (2.9), ossia se si conoscesse un insieme finito di vettori V tale che $rec(P) = cono(V)$.

Lemma 2.1 *Esiste $d \in C = cono(V)$ tale che $cd > 0$ se e solo se esiste $v_i \in V$ tale che $cv_i > 0$.*

Dimostrazione Una delle implicazioni è ovvia. Per dimostrare l'altra implicazione, si osservi che se $cv_i \leq 0$ per ogni $i = 1, \ldots, t$, allora

$$cd = c\big(\textstyle\sum_{i=1}^{t} \nu_i v^i \big) = \sum_{i=1}^{t} \nu_i (cv^i) \leq 0$$

per ogni $d \in cono(V)$ (dato che $\nu_i \geq 0$). $\diamond$

In altri termini, sarebbe potenzialmente interessante se fosse possibile rappresentare un qualsiasi cono poliedrico, oltre che nella forma "per facce" (2.6), anche nella forma "per direzioni" (2.9). Dimostriamo che questo è in effetti possibile, in entrambe le direzioni.

Teorema 2.2 *Dato $C = cono(V)$, esiste una matrice A per cui $C = \{ \ x \ : \ Ax \leq 0 \ \}$.*

Dimostrazione È chiaro che $cono(V)$ definito da (2.9) è un cono poliedrico nelle spazio esteso delle $(x, \nu) \in \mathbb{R}^{n+t}$; infatti i lati destri di tutti i vincoli sono nulli (alcuni dei vincoli sono di uguaglianza ma questo non è un problema, si veda (2.2.ii)). Occorre quindi dimostrare che la proiezione di un cono poliedrico su un sottospazio è ancora un cono poliedrico.

Questo risultato è vero in generale per i poliedri, e di dimostra mediante un procedimento algoritmico e costruttivo noto come *eliminazione di Fourier-Motzkin*. Il processo è iterativo e considera la proiezione di una variabile per volta; si consideri quindi un generico poliedro $\{ \ x \ : \ Ax \leq b \ \} \subseteq \mathbb{R}^n$ e se ne voglia calcolare la proiezione sul sottospazio $\mathbb{R}^{n-1}$ delle variabili $x_2, x_3, \ldots, x_n$. Per fare ciò si esamina il generico vincolo i-esimo, $i = 1, \ldots, m$,

$$\textstyle\sum_{j=1}^{n} a_{ij} x_j \leq b_i$$

e lo classifica in uno dei tre sottoinsiemi I^+, I^- ed I^0 a seconda che risulti $a_{i1} > 0$, $a_{i1} < 0$, ed $a_{i1} = 0$, rispettivamente. I vincoli in I^0 di fatto "non contengono la variabile x_1"; gli altri si possono riscrivere come

$$\begin{array}{lcl} i \in I^+ & \Longrightarrow & x_1 \leq \big(b_i - \sum_{j=2}^{n} a_{ij} x_j\big)/a_{i1} \\ i \in I^- & \Longrightarrow & x_1 \geq \big(b_i - \sum_{j=2}^{n} a_{ij} x_j\big)/a_{i1} \end{array}$$

A questo punto è possibile eliminare la variabile x_1 eliminando tutti i vincoli $i \in I^+ \cup I^-$ (e lasciando inalterati quelli in I^0 che non la contengono) e rimpiazzandoli con i vincoli

$$\big(b_h - \textstyle\sum_{j=2}^{n} a_{hj}x_j\big)/a_{h1} \leq \big(b_i - \sum_{j=2}^{n} a_{ij}x_j\big)/a_{i1} \tag{2.10}$$

per tutte le possibili coppie $(i,h) \in I^+ \times I^-$. Ripetendo tale procedimento un numero opportuno di volte si può calcolare la proiezione di un poliedro su un qualsiasi sottospazio delle sue variabili. Si noti che se il poliedro è un cono, ossia $b = 0$, allora $b_i = b_h = 0$ in (2.10) ed anche i nuovi vincoli introdotti hanno lato destro nullo, ossia la proiezione di un cono poliedrico è ancora un cono poliedrico. $\diamond$

Un cono definito tramite la (2.9) si dice *finitamente generato*. Il Teorema 2.2 dice quindi che qualsiasi cono finitamente generato è un cono poliedrico. Un'osservazione rilevante è che, sfortunatamente, il numero di righe nella matrice A può essere una funzione esponenziale del numero di elementi di V. Infatti, nell'eliminazione di Fourier-Motzkin il numero di diseguaglianze che definiscono il poliedro proiettato può aumentare in modo esponenziale; nel primo passo si eliminano fino a $O(m)$ vincoli ma se ne aggiungono fino a $O(m^2)$, nel secondo passo questi ultimi possono dar luogo a $O(m^4)$ vincoli, e così via. Questo, peraltro, non è solo un rischio teorico, ma accade in effetti, a meno che i poliedri (coni) non siano strutturati in modo molto particolare; ciò avrà rilevanza nel seguito.

Esercizio 2.3 *Si determini un poliedro in $\mathbb{R}^n$ la cui proiezione su un sottospazio con $n-k$ variabili mediante l'eliminazione di Fourier-Motzkin non produca un numero di vincoli esponenziale in k.*

Vogliamo adesso dimostrare la relazione inversa, ossia che qualsiasi cono poliedrico può essere espresso come cono finitamente generato. Per questo, una definizione cruciale è la seguente:

Definizione 2.1 *Dato il cono (poliedrico) $C = \{\, x \ :\ Ax \leq 0 \,\}$, il suo* cono duale *(finitamente generato) è $C^* = \{\, x = A^T\nu \ :\ \nu \in \mathbb{R}^m_+ \,\}$.*

Il cono duale C^* utilizza "gli stessi dati" di C, ma "in modo diverso". Ovviamente, C e C^* non sono uguali; in effetti, è facile dimostrare che

$$x \in C \quad , \quad d \in C^* \qquad \Longrightarrow \qquad d^T x \leq 0 \ ; \tag{2.11}$$

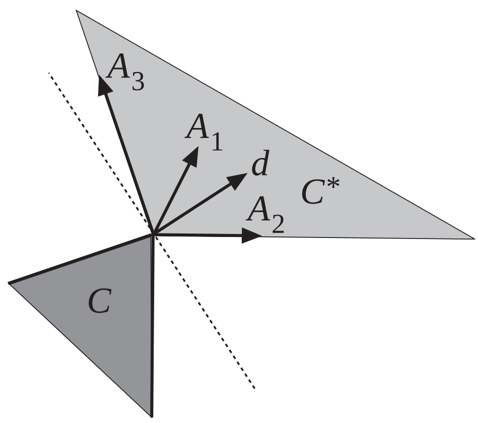


Figura 2.5: Un cono C ed il suo duale C^*

questo deriva immediatamente dal fatto che $x \in C \equiv Ax \leq 0$ e $d \in C^* \equiv d = A^T\nu$, quindi $d^T x = \nu^T Ax \leq 0$ perché $\nu \geq 0$. Geometricamente, questo significa che dato un qualsiasi vettore $d \in C^*$ l'intero cono C è compreso nel semispazio identificato da d, come mostrato ad esempio nella figura 2.5. Algebricamente, ciò corrisponde al fatto che il vincolo $dx \leq 0$ è soddisfatto da tutti i punti di C, il che è ovvio perché tale vincolo si ottiene da una combinazione lineare dei vincoli $Ax \leq 0$ che definiscono C usando coefficienti non negativi. Una proprietà del cono duale che ci sarà utile nel seguito è:

Lemma 2.2 $C^* = \{\, x \ :\ Qx \leq 0 \,\}$ *per un'opportuna matrice Q tale che tutte le righe di Q appartengono a C.*

Dimostrazione L'esistenza della rappresentazione di C^* come cono poliedrico è data dal Teorema 2.2. Per il resto, è immediato verificare che $A_i \in C^*$ per ogni $i = 1,\ldots,m$ (si prenda $\nu_i = 1$, $\nu_h = 0$ per $h \neq i$); quindi $QA_i \leq 0$ per ogni $i = 1,\ldots,m$. In altre parole, per ogni riga Q_j si ha $Q_jA_i \leq 0$ per ogni $i = 1,\ldots,m$, ossia $Q_j \in C$. $\diamond$

Possiamo adesso dimostrare il risultato principale.

Teorema 2.3 *Dato $C = \{\, x \ :\ Ax \leq 0 \,\}$, esiste un insieme V per cui $C = cono(V)$.*

Dimostrazione Si consideri $C^* = \{ x \; : \; Qx \leq 0 \}$ (cf. Lemma 2.2), ed il suo duale (il bi-duale di C)

$$C^{**} = \{ d = Q^T \nu \; : \; \nu \geq 0 \} \;\; ;$$

per il Teorema 2.2, $C^{**} = \{ x \; : \; Wx \leq 0 \}$ per una qualche matrice W. Vogliamo adesso dimostrare che $C^{**} = C$.

Per il Lemma 2.2, tutte le righe di Q sono elementi di C, quindi il loro inviluppo conico C^{**} è contenuto in C.

Viceversa, ogni $x \in C$ appartiene a C^{**}. Infatti, applicando ancora il Lemma 2.2 a C^{**} si ottiene che tutte le righe della matrice W appartengono a C^*; ciò per definizione di cono duale significa che data una qualsiasi W_j esistono moltiplicatori $\nu_i^j \geq 0$ tali che

$$W_j = \sum_{i=1}^m \nu_i A_i \;\; .$$

Quindi, dal fatto che $A_i x \leq 0$ segue che $\sum_{i=1}^m \nu_i A_i x = W_j x \leq 0$, ossia $x \in C^{**}$. ◇

La famiglia dei coni poliedrici coincide quindi con quella dei coni finitamente generati; questo risultato è alla base di un teorema fondamentale di caratterizzazione dei poliedri che presentiamo adesso. Si noti che, ancora una volta, questa corrispondenza "non mantiene le dimensioni": una rappresentazione come cono finitamente generato di C si ottiene dalla rappresentazione $Qx \leq 0$ del suo duale C^* come cono poliedrico. Sfortunatamente questa rappresentazione, come abbiamo visto, può essere esponenzialmente più grande della rappresentazione come cono poliedrico di C^*, che ha le stesse dimensioni della rappresentazione di C. Questo significa che in pratica non è possibile determinare se esiste un elemento $d \in rec(P)$ tale che $cd > 0$ computando una rappresentazione di $rec(P)$ come cono finitamente generato e verificando il segno di tutti i prodotti scalari cv_i, perché questo potrebbe richiedere un tempo esponenziale nelle dimensioni di P. Questi risultati forniscono comunque spunti fondamentali per la progettazione di algoritmi efficienti, quali quelli che descriveremo nel seguito.

Il teorema di decomposizione dei poliedri

Possiamo adesso generalizzare i risultati precedenti all'intera classe dei poliedri.

Teorema 2.4 *L'insieme $P \subseteq \mathbb{R}^n$ è un poliedro se e solo se esistono due insiemi finiti (eventualmente vuoti) $X = \{x^1, \ldots, x^s\} \subset \mathbb{R}^n$ e $V = \{v^1, \ldots, v^t\} \subset \mathbb{R}^n$ per cui $P = conv(X) + cono(V)$*[2]*. L'insieme $cono(V)$ coincide con $rec(P)$. In più, se P non contiene direzioni di linealità, allora una rappresentazione minimale di X contiene tutti e soli i punti estremi di P.*

Dimostrazione Le soluzioni (x, λ, ν) del sistema

$$x = \sum_{i=1}^s \lambda_i x^i + \sum_{j=1}^t \nu_j v^j \quad , \quad \sum_{i=1}^s \lambda_i = 1 \quad , \quad \lambda \geq 0 \quad , \quad \nu \geq 0 \tag{2.12}$$

che definisce $conv(X) + cono(V)$ formano un poliedro in $\mathbb{R}^{n+s+t}$; operando come nel Teorema 2.2, questo può essere proiettato sul sottospazio delle sole x. Viceversa, dato $P = \{ x \; : \; Ax \leq b \}$ si consideri il cono

$$C = \left\{ \; (x, \alpha) \; : \; Ax - b\alpha \leq 0 \; , \; \alpha \geq 0 \; \right\} \subseteq \mathbb{R}^{n+1} \;\; ;$$

è chiaro che $P = \{ \; x \; : \; (x, 1) \in C \; \}$. Possiamo adesso applicare il Teorema 2.3 a C, ottenendo che esiste un insieme $Z = \{z_1, \ldots, z_k\} \subset \mathbb{R}^{n+1}$ tale che $C = cono(Z)$. Esplicitiamo adesso l'ultima componente di ciascuno di tali vettori come $z_i = [z'_i, z"_i]$ con $z'_i \in \mathbb{R}^n$ e $z"_i \in \mathbb{R}$, ed assumiamo senza perdita di generalità che Z sia ordinato in modo tale che $z"_i \neq 0$ per $i = 1, \ldots, s$, mentre $z"_i = 0$ per $i = s+1, \ldots, k$. Si noti che, poiché $\alpha \geq 0$ per ogni $(x, \alpha) \in C$, deve sicuramente risultare $z"_i > 0$ per ogni $i = 1, \ldots, s$, in quanto $\alpha = \sum_{i=1}^s \lambda_i z"_i$ e $\lambda \geq 0$; definendo adesso $x_i = z'_i / z"_i$ per $i = 1, \ldots, s$ e $v_j = z'_{s+j}$ per $j = 1, \ldots, t = k - s$, si ottiene immediatamente (2.12) da $(x, 1) \in cono(Z)$, dimostrando così la prima parte del teorema.

Mostriamo adesso che $cono(V) = rec(P)$. Il fatto che qualsiasi $d = \sum_{j=1}^t \nu_j v^j$ (per $\nu \geq 0$) appartenga al cono di recessione di P è ovvio dal fatto, appena dimostrato, che qualsiasi $x \in P$ può essere scritto nelle forma (2.12). Per l'implicazione inversa, si consideri che $d \in rec(P) \equiv Ad \leq 0 \equiv (d, 0) \in C = cono(Z)$. Da ciò segue, con la partizione degli z_i e le definizioni date, che

$$d = \sum_{j=1}^t \nu_j v^j \quad , \quad 0 = \sum_{i=1}^s \lambda_i$$

e quindi $\lambda = 0$, da cui $d \in cono(V)$.

Infine, mostriamo che se P non contiene direzioni di linealità allora una rappresentazione minimale di X contiene tutti e soli i punti estremi di P. Intanto, i punti estremi di P *devono* essere in X, poiché per definizione non c'è alcun

[2]Dati due insiemi S e T in $\mathbb{R}^n$, entrambi non vuoti, definiamo l'insieme somma $Z = S + T$ come l'insieme di tutti quei punti z per cui esistano $x \in S$ e $y \in T$ tali che $z = x + y$; nel caso uno dei due insiemi sia vuoto, per definizione l'insieme somma coincide con l'altro insieme.

modo di esprimerli come combinazione convessa dei altri punti $x \in P$. Assumiamo quindi per assurdo che X sia una rappresentazione minimale ma un certo $x_h \in X$ non sia un punto estremo, ossia

$$x_h = \gamma x' + (1-\gamma)x" \quad , \quad x', x" \in P \quad , \quad x' \neq x" \quad , \quad \gamma \in (0,1) \ .$$

Per quanto dimostrato precedentemente si ha, per opportuni λ', ν', $\lambda"$ e $\nu"$ (non negativi)

$$x' = \textstyle\sum_{i=1}^{s} \lambda'_i x^i + \sum_{j=1}^{t} \nu'_j v^j \quad , \quad x" = \sum_{i=1}^{s} \lambda"_i x^i + \sum_{j=1}^{t} \nu"_j v^j \ ;$$

definendo adesso

$$\lambda = \gamma\lambda' + (1-\gamma)\lambda" \quad , \quad \nu = \gamma\nu' + (1-\gamma)\nu"$$

si ha chiaramente che $x_h = \sum_{i=1}^{s} \lambda_i x^i + \sum_{j=1}^{t} \nu_j v^j$, da cui

$$x_h(1-\lambda_h) = \textstyle\sum_{i\in\{1,\ldots,s\}\setminus\{h\}} \lambda_i x^i + \sum_{j=1}^{t} \nu_j v^j \ . \tag{2.13}$$

Dobbiamo dividere adesso l'analisi in due casi distinti:

- $\lambda_h < 1$: dividendo entrambi i membri di (2.13) per $1/(1-\lambda_h) \geq 0$ e definendo $\bar\lambda_i = \lambda_i/(1-\lambda_h) \geq \lambda_i \geq 0$, $\bar\nu_j = \nu_j/(1-\lambda_h) \geq \nu_j \geq 0$ si ottiene

  $$x_h = \textstyle\sum_{i\in\{1,\ldots,s\}\setminus\{h\}} \bar\lambda_i x^i + \sum_{j=1}^{t} \bar\nu_j v^j \quad , \quad \sum_{i\in\{1,\ldots,s\}\setminus\{h\}} \bar\lambda_i = 1$$

  come è facile verificare; ciò significa che $x_h \in conv(X' = X \setminus \{x_h\}) + cono(V)$, dal che segue facilmente che $P = conv(X') + cono(V)$, contraddicendo la minimalità di X.

- $\lambda_h = 1$: questo implica che $\lambda_i = 0$ per $i \neq h$, e quindi usando (2.13) e la definizione di λ e ν si ottiene

  $$0 = \gamma\left(\textstyle\sum_{j=1}^{t} \nu'_j v^j\right) + (1-\gamma)\left(\sum_{j=1}^{t} \nu"_j v^j\right) = \gamma d' + (1-\gamma)d"$$

  dove chiaramente $d', d" \in cono(V) = rec(P)$. Ma il fatto che $-\alpha d' = d" \in rec(P)$ con $\alpha = \gamma/(1-\gamma) > 0$ implica immediatamente che $-d' \in rec(P)$, e quindi che d' è una direzione di linealità, contraddicendo l'ipotesi. L'unico caso a cui fare attenzione è che non possa essere $d' = d" = 0$; ma questo, come è facile verificare, può accadere solo se $\nu' = \nu" = 0$, e considerato che si ha anche $\lambda'_i = \lambda"_i = 0$ per ogni $i \neq h$ (altrimenti un qualche λ_i sarebbe $\neq 0$, essendo che $\gamma \in (0,1)$) questo implicherebbe che $x_h = x' = x"$, contro l'ipotesi.

Questo conclude la dimostrazione del teorema. ◇

Il significato geometrico del Teorema 2.4 è illustrato nella Figura 2.6, dove il poliedro P è dato dalla somma del triangolo $Q = conv(\{x^1, x^2, x^3\})$ con il cono $C = cono(\{v^1, v^2\})$. Si potrebbe anche dimostrare formalmente che, come suggerito dalla figura, una rappresentazione minimale di V usa solamente i *raggi estremi* del cono di recessione, definiti in modo analogo ai punti estremi; questo è lasciato per esercizio.

Un altro esempio è fornito dal poliedro P di Figura 2.3, che può essere decomposto tramite i due insiemi finiti $X = \{ [\ 0, 0, 0\ ], [\ 0, 1, 0\ ], [\ 0, 0, 1\ ], [\ 0, 1, 1\ ] \}$ e $V = \{ [\ 1, 0, 0\ ] \}$; quindi $Q = conv(X)$ è il "lato sinistro" del poliedro, mentre $C = cono(V)$ è il semiasse x_1 positivo, che è parallelo a tutti e quattro gli spigoli illimitati di P.

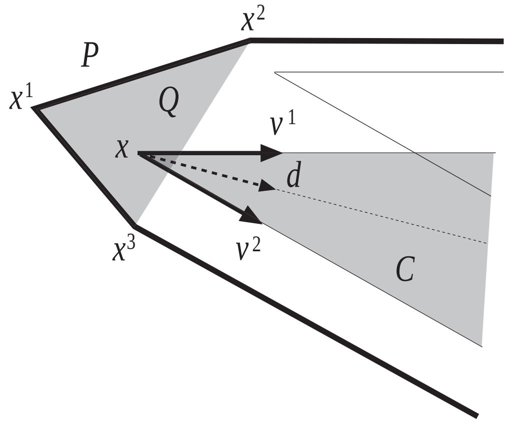


Figura 2.6: Decomposizione di un poliedro

Il Teorema 2.4 fornisce un'utile caratterizzazione teorica dei poliedri dalla quale possono essere fatte discendere molte proprietà, come ad esempio il fatto che P è compatto (e prende quindi il nome di *politopo*) se e solo se $rec(P) = \{0\}$, oppure il fatto che $conv(X)$ è il più piccolo poliedro che contiene tutti i punti di X. Dal punto di vista della PL, la proprietà più rilevante è la seguente:

Corollario 2.1 *Sia $P = \{\ x\ :\ Ax \leq b\ \} = conv(X) + cono(V) \neq \emptyset$: allora il problema (2.1) ha ottimo finito se e solo se $cv^j \leq 0$ per $j = 1, \ldots, t$, ed in questo caso esiste un $h \in \{1, \ldots, s\}$ tale che x^h è una soluzione ottima del problema.*

Dimostrazione Per il Teorema 2.4, il problema (2.1) è equivalente al seguente problema (di PL, ma con una forma particolarmente semplice dei vincoli) nelle variabili λ e ν:

$$\max\left\{ \textstyle\sum_{i=1}^{s} \lambda_i(cx^i) + \sum_{j=1}^{t} \nu_j(cv^j) \ : \ \sum_{i=1}^{s} \lambda_i = 1 \ , \ \lambda \geq 0 \ , \ \nu \geq 0 \right\} \ .$$

Questo problema è ovviamente non vuoto, ed ha ottimo finito se e solo se $cv^j \leq 0$ per ogni $j = 1, \ldots, t$. Infatti, se fosse $cv^j > 0$ per un qualche indice j, allora facendo crescere ν_j all'infinito e fissando tutte le altre variabili il valore della funzione obiettivo crescerebbe anch'esso all'infinito. Se invece $cv^j \leq 0$ per ogni $j = 1, \ldots, t$, considerando un qualsiasi $x \in P$ ed i corrispondenti λ e ν risulta

$$cx = \textstyle\sum_{i=1}^{s} \lambda_i(cx^i) + \sum_{j=1}^{t} \nu_j(cv^j) \leq \sum_{i=1}^{s} \lambda_i(cx^i) \leq \sum_{i=1}^{s} \lambda_i(cx^h) = cx^h$$

dove x^h è tale che $cx^h = \max\{ cx^i \ : \ i = 1, \ldots, s \}$. Quindi il problema ha ottimo finito, ed inoltre x^h è una soluzione ottima di (2.1). $\diamond$

Se $rango(A) = n$, questo teorema permette di affermare che tra le soluzioni ottime di un problema di PL, se ne esistono, c'è sempre almeno un punto estremo. Ciò renderebbe possibile, in teoria, studiare i problemi di PL tramite l'analisi dei soli punti estremi del poliedro e non di tutta la regione ammissibile.

Esempio 2.7: Risoluzione della PL tramite decomposizione
Consideriamo i tre vettori costo $c = [\ -1\ ,\ 1\ ,\ 1\ ]$, $c' = [\ 0\ ,\ 1\ ,\ 1\ ]$ e $c'' = [\ 1\ ,\ 1\ ,\ 1\ ]$ per il poliedro P di Figura 2.3. Siccome $V = \{ v_1 = [\ 1\ ,\ 0\ ,\ 0\ ] \}$, il Teorema 2.1 ci garantisce che i problemi di PL corrispondenti a c e c' hanno ottimo finito, mentre quello corrispondente a c'' è illimitato.

Infatti, è immediato verificare che l'unica soluzione ottima del problema corrispondente a c è il punto estremo $[\ 0\ ,\ 1\ ,\ 1\ ]$, mentre il problema corrispondente a c' ha infinite soluzioni ottime: tutti i punti dello spigolo "anteriore in alto" del parallelepipedo hanno lo stesso valore della funzione obiettivo, che risulta essere il valore ottimo. Questo corrisponde al fatto che $c'v_1 = 0$. Si noti che, comunque, tra i punti ottimi c'è almeno un punto estremo.

Per quanto riguarda il problema corrispondente al vettore c'', non esiste un valore ottimo: dato un qualsiasi punto ammissibile, ad esempio $[\ 0\ ,\ 0\ ,\ 0\ ]$, tutti i punti ottenuti muovendosi di un passo $\alpha \geq 0$ lungo il generatore del cono, ossia i punti della forma $[\ 0\ ,\ 0\ ,\ 0\ ] + \alpha [\ 1\ ,\ 0\ ,\ 0\ ] = [\ \alpha\ ,\ 0\ ,\ 0\ ]$ sono ammissibili ed hanno valore della funzione obiettivo α; facendo crescere α si ottengono quindi punti ammissibili con valore della funzione obiettivo grande a piacere.

Se però $rango(A) < n$, ossia P contiene direzioni di linealità, l'analisi precedente non si applica in quanto l'esistenza di direzioni di linealità è incompatibile con quella dei punti estremi. Questo è chiaramente visibile rimuovendo il vincolo (1) dalla definizione del poliedro di Figura 2.3: il corrispondente poliedro (la cui forma è già stata descritta) può essere decomposto tramite i lo stesso insieme X già indicato e $V = \{ [\ 1\ ,\ 0\ ,\ 0\ ]\ ,\ [\ -1\ ,\ 0\ ,\ 0\ ] \}$, ma X non contiene punti estremi (in effetti esistono infiniti altri modi minimali di scegliere X). Questa condizione è potenzialmente problematica per lo sviluppo degli algoritmi che discuteremo, in quanto essi sono pensati per esplorare l'insieme dei vertici del poliedro. È però possibile dimostrare che in tal caso, senza perdita di generalità, ci si può ricondurre alla risoluzione di un problema definito su un poliedro che non ammette direzioni di linealità.

Teorema 2.5 *Se P ammette direzioni di linealità (ossia $Ad = 0$ per qualche d, ossia $rango(A) < n$), allora il problema (2.1) può essere risolto studiando in sua vece un diverso problema di* PL *la cui matrice dei coefficienti si ottiene eliminando una colonna da A.*

Dimostrazione Se $rango(A) < n$, esiste almeno una colonna di A che può essere espressa come combinazione lineare delle altre; assumiamo senza perdita di generalità che si tratti dell'ultima. Abbiamo quindi che $A = [\ A'\ ,\ a^n\ ]$, $c = [\ c'\ ,\ c_n\ ]$, $x = [\ x'\ ,\ x_n\ ]$, ed esiste un vettore $\mu \in \mathbb{R}^{n-1}$ tale che $A'\mu = a^n$ ($d = [\ \mu\ ,\ -1\ ]$). Allora, il problema (2.1) può essere risolto studiando in sua vece il problema

$$\max \{ c'x' \ : \ A'x' \leq b \}. \tag{2.14}$$

ossia rimuovendo l'ultima colonna di A e la corrispondente variabile. Per verificare questo notiamo che a qualsiasi soluzione ammissibile x' di (2.14) corrisponde una soluzione ammissibile $x = [\ x'\ ,\ 0\ ]$ di (2.1) avente lo stesso costo, mentre da una qualsiasi soluzione ammissibile $x = [\ x'\ ,\ x_n\ ]$ di (2.1) si può costruire la soluzione $x' + \mu x_n$ ammissibile per (2.14). Da questo segue che:

- se (2.14) non ha soluzione ammissibile allora non può averla nemmeno (2.1);
- se (2.14) è superiormente illimitato allora tale è pure (2.1).

Rimane adesso da esaminare il caso in cui si determini una soluzione ottima $\bar{x}'$ di (2.14). In questo caso, se $cd \neq 0$, allora (2.1) è superiormente illimitato: $x(\alpha) = [\ \bar{x}' \,,\, 0\ ] + \alpha d$ è ammissibile per ogni α, è sempre possibile scegliere α in modo opportuno (> 0 se $cd > 0$, < 0 se $cd < 0$) affinché $cx(\alpha)$ sia superiore a qualsiasi soglia fissata. Se invece $cd = 0 \equiv c_n = c'\mu$, (2.1) e (2.14) hanno lo stesso valore ottimo: infatti, data qualsiasi soluzione ammissibile $x = [\ x' \,,\, x_n\ ]$ di (2.1), la soluzione ammissibile corrispondente $x' + \mu x_n$ di (2.14) ha lo stesso costo

$$c'(x' + \mu x_n) = c'x' + (c'\mu)x_n = c'x' + c_n x_n \ ,$$

e viceversa a qualsiasi soluzione x' di (2.14) corrisponde una soluzione $x = [\ x' \,,\, 0\ ]$ di (2.1) con lo stesso costo. Pertanto, $\bar{x} = [\ \bar{x}' \,,\, 0\ ]$ è ottima per (2.1). ◇

Esempio 2.8: Riduzione al caso della matrice di rango pieno
Consideriamo ancora il poliedro P' ottenuto da quello di Figura 2.3 rimuovendo il vincolo (1); per quanto riguarda l'ottimizzazione su P', la variabile x_1 del problema può essere eliminata e trattata implicitamente lavorando sul politopo $\bar{P}' \subset \mathbb{R}^2$ definito da

$$A' = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & -1 \\ -1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \ ,$$

ossia il quadrato (ipercubo) unitario nello spazio $[\ x_2 \,,\, x_3\ ]$ (si noti che $\bar{P}'$ ha punti estremi, e $rango(A') = 2$). Dato qualsiasi punto $[\ x_2 \,,\, x_3\ ]$ ammissibile per $\bar{P}'$, è immediato costruire un punto $[\ x_1 \,,\, x_2 \,,\, x_3\ ]$ ammissibile per P. Pertanto, se il coefficiente di x_1 in funzione obiettivo è diverso da zero, allora il problema è superiormente illimitato; altrimenti è possibile fissare x_1 in modo arbitrario (ad esempio al valore 0) e risolvere il problema ristretto alle rimanenti due variabili.

In ogni caso, la risoluzione di un problema di PL tramite l'enumerazione di tutti i punti ed i raggi estremi del poliedro corrispondente è chiaramente non praticabile per problemi che non siano di piccole dimensioni. Infatti, anche qualora si assuma che la matrice A che caratterizza il poliedro abbia un numero di righe "ragionevolmente contenuto" (per quanto questo non sia sempre vero, né strettamente necessario, come vedremo), il numero di punti estremi del poliedro può essere esponenzialmente grande.

Esempio 2.9: Diversa dimensionalità delle rappresentazioni
Un ovvio esempio è l'*ipercubo unitario*

$$\{\ x \in \mathbb{R}^n \ :\ 0 \leq x_i \leq 1 \quad i = 1, \ldots, n\ \}$$

che è caratterizzato da $2n$ vincoli, ma che ha, come è facile verificare, 2^n punti estremi, corrispondenti a tutti i punti $x \in \{0,1\}^n$. Un esempio opposto è quello dell'*n-co-cubo*, ossia l'insieme

$$\{\ x \in \mathbb{R}^n \ :\ \textstyle\sum_{i=1}^n |x_i| \leq 1\ \}$$

(la sfera unitaria nella norma L_1). Questo insieme, nonostante sia espresso in termini della funzione non lineare $|\cdot|$, è un poliedro: infatti, applicando le tecniche della §1.2.10 lo si può riscrivere come

$$\{\ (x,v) \in \mathbb{R}^{2n} \ :\ \textstyle\sum_{i=1}^n v_i \leq 1 \ ,\ -v_i \leq x_i \leq v_i \quad i = 1, \ldots, n\ \} \ .$$

Possiamo ora applicare l'eliminazione di Fourier-Motzkin per rappresentare il poliedro in $\mathbb{R}^n$, nelle sole variabili x. Poiché abbiamo $v_1 \geq -x_1$, $v_1 \geq x_1$, per eliminare la variabile v_1 dobbiamo inserire i due vincoli

$$-x_1 + \textstyle\sum_{i=2}^n v_i \leq 1 \quad , \quad x_1 + \sum_{i=2}^n v_i \leq 1 \ .$$

Analogamente, per eliminare la variabile v_2 dobbiamo introdurre i quattro vincoli

$$-x_1 - x_2 + \textstyle\sum_{i=3}^n v_i \leq 1 \quad , \quad -x_1 + x_2 + \sum_{i=3}^n v_i \leq 1 \quad , \quad x_1 - x_2 + \sum_{i=3}^n v_i \leq 1 \quad , \quad x_1 + x_2 + \sum_{i=3}^n v_i \leq 1 \ .$$

Proseguendo si arriva a determinare che per descrivere l'insieme come un poliedro sono necessarie tutte le 2^n disuguaglianze che possono essere formate scegliendo i coefficienti delle variabili in tutti i modi possibili dall'insieme $\{-1, 1\}$; è facile però verificare che il poliedro ha solamente $2n$ punti estremi, corrispondenti agli n vettori della base canonica di $\mathbb{R}^n$ ed ai loro opposti.

L'algebra dei vincoli e delle facce

Vogliamo adesso riesaminare i concetti geometrici sviluppati nei paragrafi precedenti con strumenti algebrici. Dato il poliedro $P = \{\ x\ :\ Ax \leq b\ \}$, sia $I \subseteq \{\ 1, \ldots, m\ \}$ un qualsiasi sottoinsieme dell'insieme degli indici di riga; indichiamo allora con A_I e b_I, rispettivamente, la sottomatrice di A ed il sottovettore di b ristretti alle righe i cui indici sono in I. Indicando con $\bar{I} = \{\ 1, \ldots, m\ \} \setminus I$ il complemento di I, l'insieme

$$P_I = \{\ x\ :\ A_I x = b_I\ \ ,\ \ A_{\bar{I}} x \leq b_{\bar{I}}\ \}$$

è chiaramente un sottoinsieme di P: se $P_I \neq \emptyset$, tale insieme viene detto *faccia* di P. Da (2.2.*ii*) segue immediatamente che ogni faccia di un poliedro è a sua volta un poliedro. Il numero di facce distinte di un poliedro è al più pari al numero di sottoinsiemi distinti di $\{\ 1, \ldots, m\ \}$, e quindi è finito, anche se potenzialmente esponenziale in m. Osserviamo che, se si sceglie $I = \emptyset$, si ottiene una faccia particolare che è il poliedro stesso.

Una faccia propria (cioè non coincidente con tutto il poliedro P) che non sia contenuta in nessun'altra faccia è detta *faccia massimale* o *faccetta*, mentre una faccia che non contenga nessuna faccia distinta da essa è detta *faccia minimale*. Si noti che $P_A \subseteq P_B$ se $B \subseteq A$: quindi le facce massimali corrispondono ad insiemi "piccoli" di indici I, mentre le facce minimali corrispondono ad insiemi "grandi" di indici I. La *dimensione* di una faccia è definita come la dimensione del più piccolo sottospazio (affine) che la contiene; è possibile verificare che una faccia determinata da una matrice A_I di rango k ha dimensione $n - k$ o inferiore.

Esempio 2.10: Facce e faccette in Figura 2.3

Per il poliedro P in Figura 2.3, le facce $P_{\{1\}}$, $P_{\{2\}}$, $P_{\{3\}}$, $P_{\{4\}}$ e $P_{\{5\}}$ sono tutte e sole le faccette del poliedro; esse corrispondono rispettivamente al "lato sinistro", al "lato superiore", al "lato posteriore", al "lato inferiore" ed al "lato anteriore" del parallelepipedo. Tutte queste facce hanno dimensione $n-k = 3-1 = 2$, infatti il più piccolo sottospazio che contiene ciascuna di esse è un piano. Invece la faccia $P_{\{6\}}$ non è una faccetta del poliedro, ed ha dimensione $1 < n-k = 2$.

Si noti che la faccia $P_{\{6\}}$ coincide con le facce $P_{\{2,5\}}$, $P_{\{2,6\}}$ e $P_{\{5,6\}}$ ed ognuna di esse individua lo spigolo "anteriore in alto" del poliedro; si osservi che se è vero che ad ogni insieme $I \subset \{1, \ldots, m\}$ può corrispondere una faccia, è anche vero che ad insiemi diversi può corrispondere la stessa faccia. Inoltre, non a tutti gli insiemi I corrisponde necessariamente una faccia: questo è il caso di $P_{\{3,6\}}$, che è vuoto, in quanto i vincoli (3) e (6) individuano la retta $x_3 = 0, x_2 = 2$ che ha intersezione vuota con il poliedro.

Le facce determinate da sottomatrici A_I di rango n, se ne esistono, hanno dimensione 0, cioè sono punti: infatti, in questo caso il sistema lineare $A_I x = b_I$ ammette una ed una sola soluzione. Tali facce, dette i *vertici* di P, sono ovviamente minimali, e si può verificare che questo concetto coincide con quello di punto estremo precedentemente analizzato. A tal fine introduciamo adesso alcuni concetti che ci saranno comunque utili nel seguito.

Dato un punto $\bar{x} \in P$, i vincoli che vengono soddisfatti da $\bar{x}$ come uguaglianze vengono detti vincoli *attivi* in $\bar{x}$; indichiamo con $I(\bar{x})$ l'*insieme degli indici dei vincoli attivi*:

$$I(\bar{x}) = \{\ i\ :\ A_i \bar{x} = b_i\ \}\ \ .$$

Osserviamo che $P_{I(\bar{x})}$ è una faccia del poliedro che contiene $\bar{x}$. In generale, qualsiasi $I \subseteq I(\bar{x})$ definisce una faccia P_I che contiene $\bar{x}$: chiaramente, $I(\bar{x})$ definisce la faccia minimale tra tutte queste. Un punto $\bar{x}$ è *interno* ad una faccia P_I se è contenuto in P_I ma non è contenuto in nessuna faccia propria di P_I; questo vuol dire che $A_i \bar{x} < b_i$ per ogni $i \notin I$. Ovviamente, $\bar{x}$ è interno alla faccia $P_{I(\bar{x})}$.

Dato $\bar{x} \in P$, un vettore $\xi \in \mathbb{R}^n$ è detto una *direzione ammissibile* per $\bar{x}$ se esiste un $\bar{\lambda} > 0$ per cui $x(\lambda) = \bar{x} + \lambda\xi$ è ammissibile per (P) per ogni $\lambda \in [0, \bar{\lambda}]$, cioè per ogni $i = 1, \ldots, m$ vale

$$A_i x(\lambda) = A_i \bar{x} + \lambda A_i \xi \leq b_i\ \ . \tag{2.15}$$

Chiaramente, qualsiasi direzione $\xi \in rec(P)$ è una direzione ammissibile per qualsiasi $\bar{x}$, ma in generale l'insieme è più grande. In particolare, è facile verificare che per ogni $i \in I(\bar{x})$, ovvero per cui $A_i \bar{x} = b_i$, (2.15) è verificata se e solo se $A_i \xi \leq 0$. Se invece $i \notin I(\bar{x})$, ovvero $A_i \bar{x} < b_i$, allora (2.15) è verificata da *ogni direzione* $\xi \in \mathbb{R}^n$ purché il passo λ sia sufficientemente piccolo. Possiamo pertanto caratterizzare algebricamente le direzioni ammissibili nel modo seguente:

Proprietà 2.1 *ξ è una direzione ammissibile per $\bar{x}$ se e solo se $\xi \in C(\bar{x}) = \{ \, d \in \mathbb{R}^n \, : \, A_{I(\bar{x})} d \leq 0 \, \}$.*

L'insieme $C(\bar{x})$ di tutte le direzioni ammissibili per $\bar{x}$ è perciò un cono poliedrico; si noti che se $\bar{x}$ è un punto interno al poliedro ($A\bar{x} < b \equiv I(\bar{x}) = \emptyset$) allora $C(\bar{x}) = \mathbb{R}^n$: qualunque vettore $\xi \in \mathbb{R}^n$ è una direzione ammissibile per $\bar{x}$. Possiamo adesso dimostrare il risultato annunciato:

Teorema 2.6 *$\bar{x}$ è un punto estremo di P se e solo se $A_I \bar{x} = b_I$ per una sottomatrice A_I di rango n.*

Dimostrazione La dimostrazione si basa sulla seguente caratterizzazione: $\bar{x}$ è un punto estremo se e solo se il cono delle direzioni ammissibili $C(\bar{x})$ per $\bar{x}$ non contiene direzioni di linealità, il che a sua volta equivale a $rango(A_{I(\bar{x})}) = n$. Quindi, per ogni punto estremo $\bar{x}$ esiste $I = I(\bar{x})$ con $rango(A_I) = n$ tale che $P_I = \{\bar{x}\}$, e, viceversa, qualsiasi faccia P_I con $rango(A_I) = n$ individua un punto estremo. ◇

Come ulteriore verifica, si noti che se $rango(A) < n$ allora $rango(A_I) < n$ per ogni insieme I, ossia le facce minimali hanno dimensione maggiore di zero; infatti sappiamo che questo coincide con il fatto che P possegga direzioni di linealità, e quindi non possegga vertici.

Un insieme di indici B, di cardinalità n, tale che la sottomatrice quadrata A_B sia invertibile viene detto una *base*; A_B viene detta *matrice di base* corrispondente a B, e $\bar{x} = A_B^{-1} b_B$ viene detta *soluzione di base*. Se $\bar{x} \in P$ allora B viene detta *ammissibile*; se invece $\bar{x} \notin P$, B viene detta *non ammissibile*. Dall'analisi precedente segue immediatamente:

Corollario 2.2 *$\bar{x}$ è un punto estremo di P se e solo se esiste una base B per cui $\bar{x} = A_B^{-1} b_B$.*

Se $I(\bar{x})$ è una base, e $|I(\bar{x})| = n$, allora il vertice (e con esso la base) si dice *non degenere*; altrimenti ($|I(\bar{x})| > n$) il vertice si dice *degenere*. Le facce individuate da sottomatrici A_I di rango $n - 1$ hanno dimensione (al più) 1, sono cioè segmenti se sono limitate, semirette o rette altrimenti; tali facce sono dette *spigoli* di P.

Esempio 2.11: Basi e vertici in Figura 2.3

Ancora con riferimento al poliedro P in Figura 2.3, i quattro vertici $[\ 0\,,0\,,0\ ]$, $[\ 0\,,1\,,0\ ]$, $[\ 0\,,0\,,1\ ]$ e $[\ 0\,,1\,,1\ ]$ corrispondono a opportune basi. Infatti, $[\ 0\,,0\,,0\ ]$ corrisponde alla base $\{1,3,4\}$, $[\ 0\,,0\,,1\ ]$ corrisponde alla base $\{1,4,5\}$, $[\ 0\,,1\,,0\ ]$ corrisponde alla base $\{1,2,3\}$, e $[\ 0\,,1\,,1\ ]$ corrisponde alle basi $\{1,2,5\}$, $\{1,2,6\}$ e $\{1,5,6\}$. Si noti quindi che mentre una base ammissibile identifica univocamente un vertice del poliedro, il viceversa non è vero: esistono vertici ai quali sono associate più basi. Esistono anche basi non ammissibili, ossia che non corrispondono a vertici del poliedro: nel nostro esempio questo è il caso delle basi $\{1,3,6\}$ e $\{1,4,6\}$, che individuano le soluzioni di base non ammissibili $[\ 0\,,2\,,0\ ]$ e $[\ 0\,,0\,,2\ ]$. I tre vertici $[\ 0\,,0\,,0\ ]$, $[\ 0\,,1\,,0\ ]$ e $[\ 0\,,0\,,1\ ]$ sono non degeneri, mentre $[\ 0\,,1\,,1\ ]$ è degenere.

Il poliedro ha quattro spigoli limitati, corrispondenti al "lato sinistro", e quattro spigoli illimitati, corrispondenti ai "lati infiniti" del parallelepipedo.

Scopo fondamentale del nostro studio è sviluppare algoritmi per risolvere i problemi di *PL*. Assumendo senza perdita di generalità che la matrice dei coefficienti A abbia rango massimo, come consentito dal Teorema 2.5, abbiamo come conseguenza del Teorema 2.1 che un tale algoritmo può limitarsi a visitare i vertici del poliedro corrispondente; infatti, se il problema non è superiormente illimitato (né vuoto) allora esiste sempre una soluzione ottima che coincide con un vertice. Il Corollario 2.2 fornisce un meccanismo algebrico per generare un vertice del poliedro in modo "efficiente", ossia al costo della soluzione di un sistema lineare $n \times n$ ($O(n^3)$ per matrici dense non strutturate usando metodi elementari, molto meno in pratica).

Anche i raggi estremi di $rec(P)$, che è pure (in linea di principio) necessario esaminare ai sensi del Teorema 2.1, possono essere enumerati attraverso basi. Ciò discende da una proprietà particolare dei coni poliedrici associati alle matrici di base, detti *coni simpliciali*, enunciata (e dimostrata) dalla seguente serie di semplici equivalenze:

$$\{ \, x \, : \, A_B x \leq 0 \, \} \; = \; \{ \, x \, : \, -A_B x = \nu \, , \, \nu \geq 0 \, \} \; = \; \{ \, x = -A_B^{-1} \nu \, : \, \nu \geq 0 \, \} \; . \tag{2.16}$$

Questo significa che mentre per la famiglia dei coni poliedrici la conversione in cono finitamente generato, sebbene sempre possibile (cf. Teorema 2.3), può avere un costo esponenziale in quanto il cono finitamente generato può avere un numero di generatori esponenziale, i coni simpliciali hanno esattamente n generatori, corrispondenti all'opposto delle colonne della matrice inversa A_B^{-1}, e quindi una loro descrizione si può calcolare con complessità $O(n^3)$ o inferiore. Si potrebbe anche dimostrare (cosa che eviteremo di fare) l'equivalente del Corollario 2.2:

Teorema 2.7 *Dato un cono poliedrico $C = \{\ x\ :\ Ax \leq 0\ \}$, v è un generatore di C se e solo se esiste una base B tale che v è un generatore del* cono di base $C_B = \{\ x\ :\ A_B x \leq 0\ \}$ *(che si ottiene quindi attraverso (2.16)) e $v \in C$.*

Esempio 2.12: Generatori di un cono simpliciale
Si consideri il P in Figura 2.3 e la base $B = \{1, 4, 5\}$. Si ha quindi

$$A_B = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad , \quad b_B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad , \quad A_B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \ .$$

Calcolando $A_B^{-1} b_B$ si ottiene infatti il vertice $[0, 0, 1]$ del poliedro; inoltre, il cono simpliciale associato alla base,

$$-x_1 \leq 0 \quad , \quad -x_2 \leq 0 \quad , \quad x_3 \leq 0$$

corrisponde al cono finitamente generato dai vettori

$$\begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \quad , \quad \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \quad , \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

ossia l'opposto delle colonne di A_B^{-1} (che in questo caso, ma solamente per caso, coincidono con le colonne di A_B).

In questo caso, poiché il vertice non è degenere, i generatori del cono individuano i tre spigoli del poliedro incidenti nel vertice; ciò, come vedremo, sarà fondamentale per gli algoritmi. È possibile vedere che non sempre i generatori dei coni simpliciali associati a basi ammissibili (che individuano vertici del poliedro) individuano spigoli del poliedro: si lascia per esercizio di verificare che, ad esempio, questo non capita per tutti i generatori del cono simpliciale corrispondente alla base $\{1, 3, 6\}$ che individua il vertice degenere $[\ 0\,,\,1\,,\,1\ ]$. Questa occorrenza verrà discussa nel dettaglio nel seguito.

Ovviamente, enumerare tutte le basi non porta ad un approccio efficiente, perché il numero di basi è esponenziale in n ed m. Infatti, gli algoritmi che proporremo non enumerano ciecamente le basi, ma cercano di "esplorare lo spazio delle basi" in modo "guidato", e quindi più efficiente.

2.2 Teoria della Dualità

I problemi di *PL* hanno complessità polinomiale: una motivazione risiede nel fatto che esistono metodi efficienti per verificare se una data soluzione sia o meno ottima. In generale, per fare questo servono tecniche che permettano di valutare il valore ottimo della funzione obiettivo del problema. Nel caso di un problema di *PL*, questo avviene con l'ausilio di un altro problema di *PL*, detto *duale* del problema dato.

2.2.1 Coppie di problemi duali

Prima di formalizzare il concetto di problema duale, analizziamo due esempi.

Esempio 2.13: Un problema di dieta
In un allevamento di polli si usano per il mangime due tipi di cereali, A e B. Il mangime deve soddisfare certi requisiti nutritivi: deve contenere almeno 8 unità di carboidrati, 15 unità di proteine e 3 unità di vitamine per unità di peso. Il contenuto unitario di carboidrati, proteine e vitamine ed il costo unitario di A e B sono riportati nella seguente tabella, insieme ai requisiti minimi giornalieri.

	A	B	min. giornaliero
carboidrati	5	7	8
proteine	4	2	15
vitamine	2	1	3
costo unitario	1200	750	

Siano x_1 ed x_2, rispettivamente, il numero di unità di cereale di tipo A e B impiegate nel mangime; il numero di unità di carboidrati presenti nel mangime è allora dato da $5x_1 + 7x_2$, e poiché il fabbisogno minimo di carboidrati è di 8 unità, deve risultare $5x_1 + 7x_2 \geq 8$; analogamente, per le unità di proteine deve risultare $4x_1 + 2x_2 \geq 15$ e per le unità di vitamine $2x_1 + x_2 \geq 3$. Ai tre vincoli precedenti si devono aggiungere le ovvie condizioni di non negatività delle variabili, $x_1 \geq 0$, $x_2 \geq 0$. Infine, la funzione obiettivo è $1200x_1 + 750x_2$. La dieta di costo minimo è data dunque da una soluzione del seguente problema di *PL*:

$$\begin{array}{rrcrcl}
\min & 1200x_1 & + & 750x_2 & & \\
 & 5x_1 & + & 7x_2 & \geq & 8 \\
 & 4x_1 & + & 2x_2 & \geq & 15 \\
 & 2x_1 & + & x_2 & \geq & 3 \\
 & x_1 & , & x_2 & \geq & 0
\end{array}$$

Al problema in esame è "naturalmente" associato un altro problema che chiameremo il *problema del venditore di pillole per polli*: si tratta di stabilire i prezzi di vendita di pillole rispettivamente di carboidrati, proteine e vitamine in modo che il ricavato della vendita sia massimo e che i prezzi siano competitivi, ossia che l'allevatore di polli ritenga non svantaggioso acquistare le pillole invece dei cereali A e B. Supponiamo che ciascuna pillola contenga un'unità del corrispondente elemento nutritivo, e siano y_1, y_2 e y_3 i prezzi, rispettivamente, di una pillola di carboidrati, proteine e vitamine: poiché l'allevatore deve percepire la dieta a base di pillole non più costosa della dieta a base di cereali, dovrà risultare $5y_1 + 4y_2 + 2y_3 \leq 1200$, cioè il costo dell'equivalente (da un punto di vista nutritivo) in pillole del cereale A deve essere non superiore a 1200. Analogamente, per il cereale B si ottiene $7y_1 + 2y_2 + y_3 \leq 750$. I prezzi di vendita devono essere non negativi ed il ricavato della vendita è dato $8y_1 + 15y_2 + 3y_3$, dato che 8, 15 e 3 sono il minimo numero di pillole di carboidrati, proteine e vitamine necessari alla corretta alimentazione del pollo. Il problema del venditore di pillole per polli è dunque

$$\begin{array}{rrcrcrcl}
\max & 8y_1 & + & 15y_2 & + & 3y_3 & & \\
 & 5y_1 & + & 4y_2 & + & 2y_3 & \leq & 1200 \\
 & 7y_1 & + & 2y_2 & + & y_3 & \leq & 750 \\
 & y_1 & , & y_2 & , & y_3 & \geq & 0
\end{array}$$

I due problemi sono riassunti nella seguente tabella

$$\begin{array}{cccccc}
 & x_1 & x_2 & & \max \\
y_1 & 5 & 7 & & 8 \\
y_2 & 4 & 2 & \geq & 15 \\
y_3 & 2 & 1 & & 3 \\
 & \multicolumn{2}{c}{\leq} & & \\
\min & 1200 & 750 & &
\end{array}$$

La coppia di problemi appena costruita gode di un'importante proprietà: comunque si scelgano i prezzi y_1, y_2 e y_3 delle pillole, il ricavo del venditore di pillole è sicuramente minore o uguale del costo di *qualsiasi* dieta ammissibile che l'allevatore di polli possa ottenere dai due mangimi. Infatti, i vincoli del venditore di pillole assicurano che le pillole siano più convenienti dei singoli mangimi, e quindi anche di ogni loro combinazione. La proprietà può essere verificata algebricamente nel seguente modo: moltiplicando per x_1 e x_2 i due vincoli del problema del venditore di pillole e sommando le due diseguaglianze così ottenute, si ha

$$x_1(5y_1 + 4y_2 + 2y_3) + x_2(7y_1 + 2y_2 + y_3) \leq 1200x_1 + 750x_2 \quad .$$

Riordinando i termini in modo da mattere in evidenza le variabili y_i, si ottiene

$$y_1(5x_1 + 7x_2) + y_2(4x_1 + 2x_2) + y_3(2x_1 + x_2) \leq 1200x_1 + 750x_2 \quad .$$

A questo punto possiamo utilizzare i vincoli del problema della dieta per minorare le quantità tra parentesi, ottenendo

$$8y_1 + 15y_2 + 3y_3 \leq 1200x_1 + 750x_2 \quad .$$

Il costo di una qualsiasi dieta ammissibile fornisce quindi una valutazione superiore del massimo ricavo del venditore di pillole e, analogamente, qualsiasi ricavo ammissibile per il venditore di pillole fornisce una valutazione inferiore del costo della miglior dieta possibile. Ciascuno dei due problemi fornisce quindi, attraverso il costo di qualsiasi soluzione ammissibile, una valutazione (inferiore o superiore) del valore ottimo dell'altro problema.

Esempio 2.14: Un problema di trasporto

Un produttore di birra ha n fabbriche e m depositi. Siano:

- a_i il numero di litri di birra prodotti dalla fabbrica i ($i = 1, \ldots, n$);
- b_j la capacità del deposito j ($j = 1, \ldots, m$), espressa in litri di birra;
- c_{ij} il costo di trasporto di un litro di birra dalla fabbrica i al deposito j, detta *costo unitario* di trasporto.

Assumiamo che sia $\sum_{i=1}^{n} a_i = \sum_{j=1}^{m} b_j$, cioè che la produzione totale eguagli la capacità totale dei depositi. Si vuole trasportare il prodotto dalle fabbriche ai depositi in modo da minimizzare il costo di trasporto.
Indichiamo con x_{ij} il numero di litri di birra trasportate da i a j. I vincoli, oltre alle ovvie condizioni di non negatività delle variabili ($x_{ij} \geq 0$), impongono che

- da ogni fabbrica i venga trasportata tutta la birra prodotta, e cioè $\sum_{j=1}^{m} x_{ij} = a_i$;
- la birra trasportata ad ogni deposito j saturi la sua capacità, e cioè $\sum_{i=1}^{n} x_{ij} = b_j$.

Il costo di trasporto è $\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij}$. Il problema può essere quindi formulato come il seguente problema di *PL*:

$$\begin{array}{lll} \min & \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} & \\ & \sum_{j=1}^{m} x_{ij} = a_i & i = 1, \ldots, n \\ & \sum_{i=1}^{n} x_{ij} = b_j & j = 1, \ldots, m \\ & x_{ij} \geq 0 & i = 1, \ldots, n \;\; , \;\; j = 1, \ldots, m \end{array} .$$

Anche in questo caso possiamo definire un altro problema che utilizza gli stessi dati del problema di trasporto e che è ad esso strettamente legato. Una ditta di trasporti, per ogni coppia (i, j), offre di comprare la birra alla fabbrica i al prezzo unitario λ_i, rivendendola al deposito j al prezzo unitario μ_j. Il problema di questa ditta è definire i valori di λ_i e μ_j, per ogni (i, j), in modo da massimizzare il guadagno ottenuto dall'operazione, che è dato da $-\sum_{i=1}^{n} a_i \lambda_i + \sum_{j=1}^{m} b_j \mu_j$, con il vincolo che l'offerta fatta risulti per il produttore di birra competitiva rispetto al costo di trasporto cui andrebbe incontro se decidesse di rifiutarla. In base all'offerta della ditta di trasporto, l'industria pagherebbe $-\lambda_i + \mu_j$ il trasporto di un litro di birra da i a j, pertanto dovrà risultare $-\lambda_i + \mu_j \leq c_{ij}$: il problema della ditta di trasporto può quindi essere formulato come il seguente problema di *PL*:

$$\begin{array}{lll} \max & -\sum_{i=1}^{n} a_i \lambda_i + \sum_{j=1}^{m} b_j \mu_j & \\ & -\lambda_i + \mu_j \leq c_{ij} & i = 1, \ldots, n \;\; , \;\; j = 1, \ldots, m \end{array}$$

I due problemi, nel caso di due industrie e tre depositi, sono rappresentati in modo compatto nella seguente tabella, in cui il generico vincolo $\sum_j x_{ij} = a_i$ è stato trasformato nel vincolo equivalente $-\sum_j x_{ij} = -a_i$.

$$\begin{array}{c|cccccc|c|c|} & x_{11} & x_{12} & x_{13} & x_{21} & x_{22} & x_{23} & & \max \\ \cline{2-7} \cline{9-9} \lambda_1 & -1 & -1 & -1 & & & & & -a_1 \\ \lambda_2 & & & & -1 & -1 & -1 & & -a_2 \\ \mu_1 & 1 & & & 1 & & & = & b_1 \\ \mu_2 & & 1 & & & 1 & & & b_2 \\ \mu_3 & & & 1 & & & 1 & & b_3 \\ \cline{2-7} \cline{9-9} \multicolumn{1}{c}{} & \multicolumn{6}{c}{\leq} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \\ \min & c_{11} & c_{12} & c_{13} & c_{21} & c_{22} & c_{23} & \multicolumn{1}{c}{} & \multicolumn{1}{c}{} \end{array}$$

Ancora una volta, è facile verificare algebricamente che il costo di una qualsiasi soluzione ammissibile di uno dei due problemi fornisce una valutazione del valore ottimo dell'altro: utilizzando i vincoli del problema della ditta di trasporto si ottiene

$$\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \geq \sum_{i=1}^{n} \sum_{j=1}^{m} (\mu_j - \lambda_i) x_{ij} \;\; ,$$

dopodiché, mettendo in evidenza μ_j e λ_i ed utilizzando i vincoli del problema della fabbrica di birra, si ottiene

$$\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \geq \sum_{j=1}^{m} b_j \mu_j - \sum_{i=1}^{n} a_i \lambda_i \;\; .$$

Nei due esempi precedenti si è visto come, in modo naturale, ad un problema di *PL* è associabile un altro problema di *PL* avente con esso una stretta relazione. Generalizzando, dato un problema di *PL* nella forma

$$(P) \qquad \max \{ \, cx \; : \; Ax \leq b \; , \; x \geq 0 \, \} \tag{2.17}$$

possiamo sempre associare ad esso un problema così definito:

$$(D) \qquad \min \{ \, yb \; : \; yA \geq c \; , \; y \geq 0 \, \} \;\; . \tag{2.18}$$

I problemi (2.17) e (2.18) costituiscono una *coppia di problemi duali*: spesso (2.17) viene chiamato il *primale* e (2.18) il *duale*. Vale il seguente teorema.

Teorema 2.8 *Il duale del duale è il primale.*

Dimostrazione Utilizzando (2.2.*i*) ed (2.2.*iii*) il problema (2.18) può essere trasformato in

$$-\max \{ \, -yb \; : \; -yA \leq -c \; , \; y \geq 0 \, \} \;\; ;$$

secondo la definizione (2.17)–(2.18), il duale di questo problema è

$$-\min \{ \, -cx \; : \; -Ax \geq -b \; , \; x \geq 0 \, \} \;\; ,$$

che, utilizzando di nuovo (2.2.*i*) ed (2.2.*iii*), può essere trasformato in (2.17). ◇

La coppia di problemi duali che abbiamo introdotto è detta *coppia simmetrica*. Una definizione equivalente di coppia di problemi duali si ha con la *coppia asimmetrica*

$$(P) \quad \max\{\, cx \;:\; Ax \le b \,\} \qquad\qquad (D) \quad \min\{\, yb \;:\; yA = c \,,\; y \ge 0 \,\} \;.$$

Infatti, (P) può essere scritto equivalentemente, utilizzando (2.3), come

$$\max\{\, cx^+ - cx^- \;:\; Ax^+ - Ax^- \le b \,,\; x^+, x^- \ge 0 \,\} \quad ;$$

applicando la definizione di coppia simmetrica si ottiene il duale

$$\min\{\, yb \;:\; yA \ge c \,,\; -yA \ge -c \,,\; y \ge 0 \,\}$$

che, via (2.2.ii), è equivalente a (D).

Esercizio 2.4 *Si dimostri il viceversa, cioè che partendo dalla definizione di coppia asimmetrica si ottiene la definizione di coppia simmetrica.*

Esercizio 2.5 *Si dimostri il Teorema 2.8 per la coppia asimmetrica.*

In generale, il duale di un qualunque problema di PL può essere scritto applicando le corrispondenze (P)–(D) indicate nella seguente tabella, dove A_i e A^j indicano, rispettivamente, la i-esima riga e la j-esima colonna della matrice A.

max	c	b	$A_i x \le b_i$	$A_i x \ge b_i$	$A_i x = b_i$	$x_j \ge 0$	$x_j \le 0$	$x_j \gtrless 0$
min	b	c	$y_i \ge 0$	$y_i \le 0$	$y_i \gtrless 0$	$yA^j \ge c_j$	$yA^j \le c_j$	$yA^j = c_j$

(2.19)

Esempio 2.15: Applicazione delle corrispondenze (P)–(D)
Il duale del problema di minimo

$$\begin{array}{rrcrcr} \min & 12x_1 & + & 7x_2 & & \\ & 5x_1 & + & 7x_2 & = & 8 \\ & 4x_1 & + & 2x_2 & \ge & 15 \\ & 2x_1 & + & x_2 & \le & 3 \\ & x_1 & & & \ge & 0 \end{array} \qquad \text{è} \qquad \begin{array}{rrcrcrcr} \max & 8y_1 & + & 15y_2 & + & 3y_3 & & \\ & 5y_1 & + & 4y_2 & + & 2y_3 & \le & 12 \\ & 7y_1 & + & 2y_2 & + & y_3 & = & 7 \\ & & & y_2 & & & \ge & 0 \\ & & & & & y_3 & \le & 0 \end{array} \;.$$

Esercizio 2.6 *Si dimostrino le relazioni (P)–(D) della tabella (2.19) a partire sia dalla definizione di coppia simmetrica che da quella di coppia asimmetrica.*

Esempio 2.16: Geometria della dualità
Si consideri la coppia di problemi duali:

$$(P) \quad \begin{array}{rrcrcr} \max & 3x_1 & - & x_2 & & \\ & x_1 & + & x_2 & \le & 4 \\ & -x_1 & + & x_2 & \le & 5 \\ & & - & x_2 & \le & 2 \end{array} \qquad\qquad (D) \quad \begin{array}{rrcrcrcr} \min & 4y_1 & + & 5y_2 & + & 2y_3 & & \\ & y_1 & - & y_2 & & & = & 3 \\ & y_1 & + & y_2 & - & y_3 & = & -1 \\ & y_1 & , & y_2 & , & y_3 & \ge & 0 \end{array}$$

In Figura 2.7 è data la rappresentazione della coppia sia sotto forma di tabella che sotto forma geometrica. Osserviamo come il duale sia il problema di esprimere il vettore c come combinazione lineare non negativa delle righe A_1, A_2 e A_3, scegliendo fra i vettori di coefficienti della combinazione lineare uno che abbia peso minimo, dove il peso del coefficiente della j-esima riga è il termine noto (la risorsa) del j-esimo vincolo del primale.

2.2.2 Il teorema debole della dualità

Come già evidenziato negli esempi 2.13 e 2.14, i problemi (P) e (D) non sono legati soltanto da relazioni di tipo sintattico: il seguente teorema fornisce una prima relazione tra i valori delle funzioni obiettivo dei due problemi. Qui, come nel seguito, salvo indicazione contraria useremo la forma asimmetrica della dualità, ma i risultati ottenuti sono indipendenti dalla particolare forma usata.

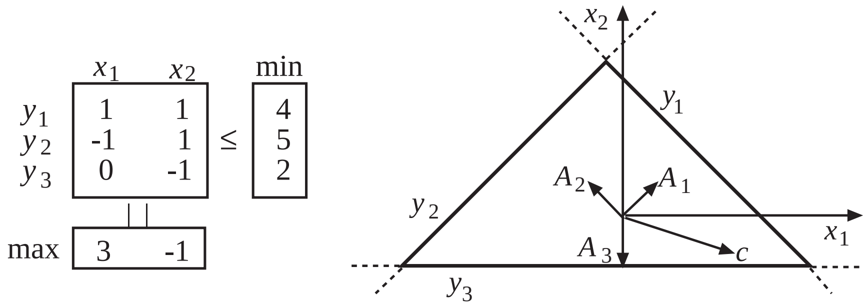


Figura 2.7: Rappresentazione algebrica e geometrica di una coppia di problemi duali

Teorema 2.9 *(Teorema debole della dualità) Se $\bar{x}$ e $\bar{y}$ sono soluzioni ammissibili per (P) e (D) rispettivamente, allora $c\bar{x} \leq \bar{y}b$.*

Dimostrazione

$$\left.\begin{array}{rcl} \bar{y}A = c & \Longrightarrow & \bar{y}A\bar{x} = c\bar{x} \\ A\bar{x} \leq b \;\;,\;\; \bar{y} \geq 0 & \Longrightarrow & \bar{y}A\bar{x} \leq \bar{y}b \end{array}\right\} \Longrightarrow c\bar{x} \leq \bar{y}b \;\; .$$

$\diamond$

La dimostrazione di questo teorema, per quanto elementare, merita di essere commentata, riprendendo i concetti già espressi negli esempi. L'idea fondamentale è che, poiché i vincoli $A_ix \leq b_i$ sono rispettati da tutti i punti della regione ammissibile, allora anche qualsiasi loro combinazione lineare a coefficienti non negativi

$$(\,yA\,)x = \left(\,\sum_{i=1}^{m} y_iA_i\,\right)x \leq \left(\,\sum_{i=1}^{m} y_ib_i\,\right) = yb$$

ha la stessa proprietà; si dice quindi che $(\,yA\,)x \leq yb$ è una *disuguaglianza valida* per il problema. Geometricamente, questo corrisponde ad un semispazio (affine) che contiene interamente il poliedro. Scegliendo le y in modo opportuno, ossia in modo tale che $yA = c$, l'iperpiano che definisce il vincolo ha lo stesso gradiente della funzione obiettivo; una disuguaglianza di tipo $cx \leq \gamma$ definisce una *curva di livello* della funzione obiettivo, ossia l'insieme di tutti i punti che hanno valore della funzione minore od uguale a γ. Determinare che una disuguaglianza di tipo $cx \leq \gamma$ è valida per il poliedro del problema (la sua regione ammissibile) dimostra chiaramente che nessun punto del poliedro può avere valore della funzione obiettivo maggiore di $\gamma(= yb)$, e quindi che questa è una corretta valutazione superiore del valore ottimo di (P).

Corollario 2.3 *Se (P) è illimitato, allora (D) è vuoto.*

In generale, se (P) e (D) sono non vuoti, si può quindi affermare che

$$\max\{\; cx \;:\; Ax \leq b \;\} \;\leq\; \min\{\; yb \;:\; yA = c \;,\; y \geq 0 \;\} \;\; .$$

Questa relazione vale in generale assumendo che il valore ottimo di un problema vuoto sia $-\infty$ se il problema è di massimo e $+\infty$ se il problema è di minimo. Una conseguenza apparentemente banale, ma in realtà cruciale, di questi risultati è:

Corollario 2.4 *Se $\bar{x}$ e $\bar{y}$ sono soluzioni ammissibili rispettivamente per (P) e (D) e $c\bar{x} = \bar{y}b$, allora $\bar{x}$ e $\bar{y}$ sono anche soluzioni ottime.*

Data una soluzione primale ammissibile $\bar{x}$, una soluzione duale ammissibile $\bar{y}$ tale che $c\bar{x} = \bar{y}b$ è detta un *certificato di ottimalità* di $\bar{x}$; si noti che, viceversa, $\bar{x}$ è un certificato di ottimalità di $\bar{y}$.

Esempio 2.17: Certificati di ottimalità
Ritorniamo al problema della *Pintel*. Abbiamo già evidenziato, con considerazioni puramente geometriche, come il vertice

$[\ 4\ ,\ 1\ ]$ sia la soluzione ottima: possiamo ora utilizzare il teorema debole della dualità per fornire una prova algebrica della sua ottimalità. Per questo, scriviamo il problema ed il suo duale in forma di coppia asimmetrica:

$$\begin{array}{rrrrl} \max & 500x_1 & + & 200x_2 & \\ & x_1 & & & \leq 4 \\ & & & x_2 & \leq 7 \\ & 2x_1 & + & x_2 & \leq 9 \\ & -x_1 & & & \leq 0 \\ & & - & x_2 & \leq 0 \end{array} \qquad \begin{array}{rrrrrrrrrrl} \min & 4y_1 & + & 7y_2 & + & 9y_3 & & & & & \\ & y_1 & & & + & 2y_3 & - & y_4 & & & = 500 \\ & & & y_2 & + & y_3 & & & - & y_5 & = 200 \\ & y_1 & , & y_2 & , & y_3 & , & y_4 & , & y_5 & \geq 0 \end{array}$$

È facile verificare che la soluzione duale $\bar{y} = [\ 100\ ,\ 0\ ,\ 200\ ,\ 0\ ,\ 0\ ]$ è ammissibile con $\bar{y}b = 2200$: poiché $\bar{x} = [\ 4\ ,\ 1\ ]$ è ammissibile per il problema primale ed ha anch'essa valore della funzione obiettivo $c\bar{x} = 2200$, il Corollario 2.4 garantisce l'ottimalità di entrambe le soluzioni (per il rispettivo problema). Si osservi che il caso in cui i valori delle funzioni obiettivo siano diversi tra loro non permette di certificare la *non* ottimalità delle soluzioni considerate. Ad esempio, avendo avuto $\bar{y}' = [\ 0\ ,\ 100\ ,\ 200\ ,\ 0\ ,\ 0\ ]$ con $\bar{y}' = 2500$ non avremmo potuto dichiarare che $\bar{x}$ non è ottima per il primale, così come $\bar{x}' = [\ 1\ ,\ 7\ ]$ con $c\bar{x}' = 1900$ non permette di dichiarare che $\bar{y}$ non è ottima per il duale.

2.2.3 Il teorema forte della dualità e sue conseguenze

Data la coppia asimmetrica di problemi duali (P) e (D), sia $\bar{x}$ una soluzione ammissibile per (P). Abbiamo già introdotto (cf. la Proprietà 2.1) il concetto di direzione ammissibile ξ per $\bar{x}$. Definiamo ξ una *direzione di crescita* per $\bar{x}$ se è possibile effettuare uno spostamento $\lambda > 0$ lungo ξ che migliori il valore della funzione obiettivo, cioè tale che

$$cx(\lambda) = c\bar{x} + \lambda c\xi > c\bar{x} \ .$$

È facile verificare che questo non dipende dal particolare punto $\bar{x}$, e che vale

Proprietà 2.2 *ξ è una direzione di crescita se e solo se $c\xi > 0$.*

Si noti che se $c = 0$ non esistono direzioni di crescita; infatti, la funzione obiettivo vale sempre zero e tutte le soluzioni ammissibili sono quindi ottime. Quando invece $c \neq 0$, se esiste una direzione ammissibile per $\bar{x}$ che sia anche di crescita, allora $\bar{x}$ non può essere soluzione ottima di (P): infatti in tal caso sarebbe possibile effettuare un passo di spostamento $\lambda > 0$ lungo ξ che migliori il valore della funzione obiettivo. Quindi, la non esistenza di *direzioni ammissibili di crescita* per $\bar{x}$ è condizione necessaria affinché $\bar{x}$ sia ottima; si può dimostrare che la condizione è anche sufficiente.

Lemma 2.3 *Sia $c \neq 0$: una soluzione ammissibile $\bar{x}$ è ottima per (P) se e solo se $\bar{x}$ non ammette direzioni ammissibili ξ che siano anche di crescita.*

Dimostrazione Una delle implicazioni è già stata discussa; supponiamo ora per assurdo che $\bar{x}$ non sia ottima ma non esistano direzioni ammissibili di crescita per il punto. Deve esistere una soluzione ammissibile x' tale che $cx' > c\bar{x}$, cioè tale che $c\xi = c(x' - \bar{x}) > 0$: ξ è quindi una direzione di crescita, ma è anche ammissibile perchè la regione ammissibile del problema è convessa, il che fornisce una contraddizione. $\diamond$

In particolare, nessun punto $\bar{x}$ interno al poliedro può essere una soluzione ottima per (P) (a meno che sia $c = 0$), in quanto $\xi = c$ costituisce una sua direzione ammissibile di crescita.

I risultati precedenti non usano, in realtà, il fatto che la regione ammissibile sia un poliedro, ma solo il fatto che è convessa; queste considerazioni possono quindi essere immediatamente estese ai problemi di ottimizzazione non lineare i cui vincoli definiscono una regione convessa.

Combinando le Proprietà 2.1 e 2.2 ed il Lemma 2.3 possiamo affermare che una soluzione ammissibile $\bar{x}$ è ottima per (P) se e solo se il sistema

$$\begin{cases} A_{I(\bar{x})}\xi & \leq 0 \\ c\xi & > 0 \end{cases}$$

non ammette soluzione. Questa caratterizzazione è di fondamentale importanza per stabilire un forte legame tra il problema primale ed il suo duale. Introduciamo a tal fine il seguente Lemma, anche noto come *Teorema Fondamentale delle Disuguaglianze Lineari*. Di tale risultato si possono dare diverse dimostrazioni (si consultino ad esempio, i testi di Murty e Padberg indicati nelle referenze di fine capitolo); noi eviteremo in questo contesto di fornire una dimostrazione formale, anche perché è possibile derivarne una da risultati relativi agli algoritmi che discuteremo nel seguito (si veda il Teorema 2.19).

Teorema 2.10 *Data $A \in \mathbb{R}^{m\times n}$ e $c \in \mathbb{R}^n$, i due sistemi*

$$(S_P) \quad \begin{cases} A\xi & \leq 0 \\ c\xi & > 0 \end{cases} \qquad\qquad (S_D) \quad \begin{cases} yA & = c \\ y & \geq 0 \end{cases}$$

sono mutuamente esclusivi, cioè o ha soluzione il sistema (S_P) oppure ha soluzione il sistema (S_D).

Interpretazione geometrica del Lemma di Farkas

Per interpretare geometricamente il Lemma di Farkas facciamo riferimento al cono C ed il suo duale C^* (si ricordi la Definizione 2.1): il Lemma afferma che o $c \in C^*$, oppure esiste $\xi \in C$ tale che $c\xi > 0$. Che le due proprietà non possano verificarsi contemporaneamente discende direttamente da (2.11): $d\xi \leq 0$ per ogni $d \in C^*$ e $\xi \in C$, per cui se (S_D) ha soluzione allora (S_P) non può averne, e viceversa (infatti il punto critico è dimostrare che i due sistemi non possono essere entrambi vuoti). Un vettore $\xi \in C$ tale che $c\xi > 0$ definisce un iperpiano che *separa* c da C^*: infatti c appartiene al semispazio (aperto) $\{\ d\ :\ d\xi > 0\ \}$, mentre per la proprietà appena enunciata $C^* \subseteq \{\ d\ :\ d\xi \leq 0\ \}$. Questa interpretazione geometrica è illustrata nella Figura 2.8, dove nel caso (a) si ha che $c \in C^*$, ossia c può essere espresso come combinazione lineare non negativa dei vettori A_i (in effetti utilizzando i soli due vettori A_2 ed A_3, per cui $c \in cono(\ \{\ A_2\ ,\ A_3\ \}\ )$) mentre nel caso (b) è indicato il piano (ortogonale a $\bar{\xi}$) che separa c dal cono C^*. Il Lemma di Farkas è in effetti solo un esempio di una rilevante classe di teoremi, detti *teoremi di separazione*, che riguardano il fatto che (sotto opportune ipotesi) dato un insieme convesso (in questo caso C^*) ed un punto che non gli appartiene (in questo caso c) esiste sempre un iperpiano che li separa.

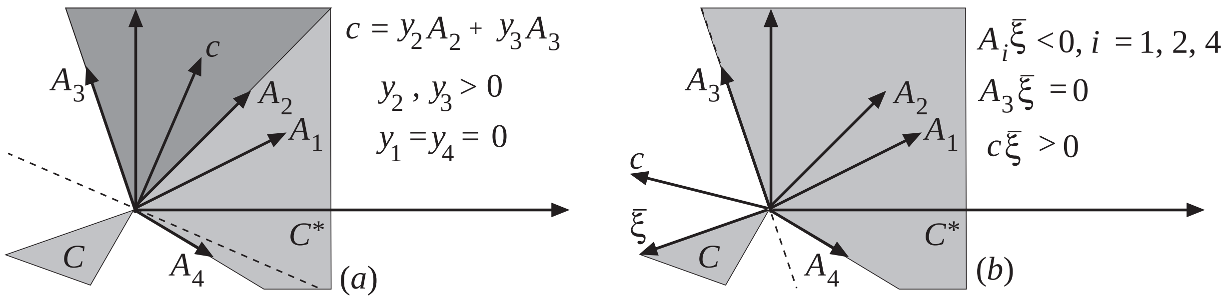


Figura 2.8: Interpretazione geometrica del Lemma di Farkas

Il teorema forte della dualità

Consideriamo il sistema che caratterizza le direzioni ammissibili di crescita per una soluzione ammissibile $\bar{x}$ e il sistema ad esso associato dal Lemma di Farkas. Per semplificare la notazione scriviamo $I = I(\bar{x})$: i due sistemi risultano essere

$$(P_R) \quad \begin{cases} A_I\xi & \leq 0 \\ c\xi & > 0 \end{cases} \qquad\qquad (D_R) \quad \begin{cases} y_IA_I & = c \\ y_I & \geq 0 \end{cases}$$

e verranno indicati, rispettivamente, come *Primale Ristretto* e *Duale Ristretto* (ai soli vincoli attivi). Si noti che per $c = 0$ il sistema (P_R) non può avere soluzioni; infatti, il sistema (D_R) ammette come ovvia soluzione $\bar{y}_I = 0$. Basandoci sul Lemma di Farkas, possiamo caratterizzare l'ottimalità di (P) in termini della risolubilità del sistema duale ristretto.

Corollario 2.5 *Sia $\bar{x}$ una soluzione ammissibile per (P). Allora, $\bar{x}$ è una soluzione ottima per (P) se e solo se esiste una soluzione $\bar{y}_I$ del sistema (D_R).*

Vale il seguente risultato:

Teorema 2.11 *(Teorema forte della dualità) Se (P) e (D) ammettono entrambi soluzioni ammissibili, allora*

$$z(P) = \max\{\ cx\ :\ Ax \leq b\ \}\ =\ \min\{\ yb\ :\ yA = c\ ,\ y \geq 0\ \} = z(D)\ .$$

Dimostrazione Per il Teorema debole della dualità, poiché (D) ammette soluzioni ammissibili (P) non può essere illimitato; essendo non vuoto, (P) ha allora ottimo finito. Se $c = 0$, allora $z(P) = 0$ e $y = 0$, ammissibile per (D), è quindi ottima. Assumiamo perciò $c \neq 0$, e sia $\bar{x}$ una soluzione ottima per (P) con il relativo insieme di indici dei vincoli attivi I; come osservato in precedenza, $I \neq \emptyset$. Per il Corollario 2.5, il sistema (D_R) ammette almeno una soluzione $\bar{y}_I$. La soluzione $\bar{y} = [\,\bar{y}_I\,,\,0\,]$ è ammissibile per (D), poiché $\bar{y}A = \bar{y}_I A_I = c$ e $\bar{y}_I \geq 0$ implica $\bar{y} \geq 0$; inoltre, risulta $\bar{y}b = \bar{y}_I b_I = \bar{y}_I A_I \bar{x} = c\bar{x}$. Per il Teorema debole della dualità $\bar{y}$ è ottima per (D) e la tesi segue. $\diamond$

Esempio 2.18: Applicazione del Teorema forte della dualità

Si consideri la seguente coppia di problemi duali:

$$(P)\quad \begin{array}{rrrrrl} \max & 2x_1 & + & x_2 & & \\ & x_1 & & & \leq & 5 \\ & & & x_2 & \leq & 5 \\ & x_1 & + & x_2 & \leq & 10 \\ & -x_1 & - & x_2 & \leq & -5 \end{array} \qquad (D)\quad \begin{array}{rrrrrrrrl} \min & 5y_1 & + & 5y_2 & + & 10y_3 & - & 5y_4 & \\ & y_1 & & & + & y_3 & - & y_4 & = 2 \\ & & & y_2 & + & y_3 & - & y_4 & = 1 \\ & y_1 & , & y_2 & , & y_3 & , & y_4 & \geq 0 \end{array}$$

Sia $\bar{x} = \left[\ 5\ ,\ 5\ \right]$ una soluzione ammissibile di (P), con $c\bar{x} = 15$. L'insieme degli indici dei vincoli attivi è $I = \{1, 2, 3\}$, a cui corrisponde la coppia di sistemi

$$(P_R)\quad \begin{array}{rrrl} \xi_1 & & & \leq 0 \\ & & \xi_2 & \leq 0 \\ \xi_1 & + & \xi_2 & \leq 0 \\ 2\xi_1 & + & \xi_2 & > 0 \end{array} \qquad (D_R)\quad \begin{array}{rrrrrl} y_1 & & & + & y_3 & = 2 \\ & & y_2 & + & y_3 & = 1 \\ y_1 & , & y_2 & , & y_3 & \geq 0 \end{array}$$

Ogni direzione ammissibile e di crescita per $\bar{x}$ deve essere soluzione di (P_R), ma è facile verificare che tale sistema non ha soluzione: infatti, dovendo essere ξ_1 e ξ_2 entrambe non positive, non è possibile che $2\xi_1 + \xi_2$ risulti positivo. Per il Lemma di Farkas deve quindi avere soluzione il sistema (D_R), che ha infatti ne ha infinite; ponendo $y_3 = \alpha$ e sostituendo nelle equazioni si ottiene che $y_I(\alpha) = \left[\ 2-\alpha\,,\,1-\alpha\,,\,\alpha\ \right]$ soddisfa le equazioni del sistema duale per ogni valore di α. Imponendo anche le condizioni di non-negatività si ottiene che $y_I(\alpha)$ è una soluzione del sistema duale per ogni $\alpha \in [0, 1]$. Fissando a zero tutte le y_i per $i \notin I$ si ottiene $y(\alpha) = \left[\ 2-\alpha\,,\,1-\alpha\,,\,\alpha\,,\,0\ \right]$, che è ammissibile per (D) per ogni $\alpha \in [0, 1]$. Quindi $\bar{x}$ è una soluzione ottima per (P) e $y(\alpha)$ è una soluzione ottima per (D) per ogni $\alpha \in [0, 1]$: infatti, $c\bar{x} = y(\alpha)b = 15$.

Teorema 2.12 *Se (P) ha ottimo finito, allora (D) ha ottimo finito.*

Dimostrazione Poiché (P) è non vuoto, dal Teorema debole della dualità segue che (D) è limitato inferiormente, e quindi ha ottimo finito se e solo se è non vuoto. Per assurdo, supponiamo che il duale non abbia soluzione: per il Lemma di Farkas esiste quindi un vettore ξ tale che $A\xi \leq 0 \equiv \xi \in rec(P)$ e $c\xi > 0$; come abbiamo visto questo implica che (P) è superiormente illimitato in contraddizione con l'ipotesi. $\diamond$

Esercizio 2.7 *Dimostrare il teorema opposto: se (D) ha ottimo finito, allora (P) ha ottimo finito.*

Teorema 2.13 *Data la coppia di problemi duali (P) e (D), si può verificare uno solo tra i casi indicati con $*$ nella seguente tabella:*

	(P)	*ottimo finito*	*illimitato*	*vuoto*
(D)	*ottimo finito*	*		
	illimitato			*
	vuoto		*	*

Dimostrazione Il seguente esempio mostra che (P) e (D) possono essere entrambi vuoti, come è immediato verificare:

$$(P)\ \max\{\ x_2\ :\ -x_1 - x_2 \leq -1\ ,\ x_1 + x_2 \leq -1\ \}\quad (D)\ \min\{\ -y_1 - y_2\ :\ -y_1 + y_2 = 0\ ,\ -y_1 + y_2 = 1\ ,\ \ y_1, y_2 \geq 0\ \}\ .$$

Il completamento della dimostrazione è lasciato per esercizio. $\diamond$

2.2.4 Il teorema degli scarti complementari

I teoremi debole e forte della dualità permettono di caratterizzare l'ottimalità di una coppia di soluzioni di (P) e (D). Più precisamente, il Corollario 2.4 e il Teorema 2.11 garantiscono che, date una soluzione $\bar{x}$ ammissibile per (P) e una soluzione $\bar{y}$ ammissibile per (D), queste sono ottime se e solo se i valori delle rispettive funzioni obiettivo concidono, ovvero $c\bar{x} = \bar{y}b$. Poiché la catena di equivalenze

$$c\bar{x} = \bar{y}b \quad \Longleftrightarrow \quad \bar{y}A\bar{x} = \bar{y}b \quad \Longleftrightarrow \quad \bar{y}(\, b - A\bar{x}\,) = 0$$

vale quando $\bar{y}A = c$ indipendentemente dall'eventuale ammissibilità delle soluzioni, la seguente definizione risulterà utile nel seguito.

Definizione 2.2 *Le soluzioni $\bar{x} \in \mathbb{R}^n$ e $\bar{y} \in \mathbb{R}^m$ formano una* coppia di soluzioni complementari *se $\bar{y}A = c$ e viene verificata la seguente proprietà, detta degli* scarti complementari*:*

$$\bar{y}(\, b - A\bar{x}\,) = 0 \quad . \tag{2.20}$$

Si noti che la definizione non richiede l'ammissibilità delle soluzioni; qualora siano entrambe ammissibili, ciò è sufficiente a garantirne l'ottimalità. Infatti, da quando detto sopra si deduce immediatamente il seguente teorema, noto come *Teorema degli scarti complementari*:

Teorema 2.14 *Date due soluzioni $\bar{x}$ e $\bar{y}$, ammissibili rispettivamente per (P) e (D), esse sono ottime se e solo se verificano le condizioni degli scarti complementari (2.20).*

Esplicitando il prodotto scalare, l'equazione (2.20) può essere riscritta

$$\bar{y}(\, b - A\bar{x}\,) = \textstyle\sum_{i=1}^{m} \bar{y}_i(\, b_i - A_i\bar{x}\,) = 0 \quad .$$

Quindi, una condizione *sufficiente* perché valga (2.20) è

$$\bar{y}_i(\, b_i - A_i\bar{x}\,) = 0 \qquad\qquad i = 1, \ldots, m \tag{2.21}$$

in quanto la somma di termini nulli da sicuramente zero; a sua volta, (2.21) è equivalente a

$$\begin{array}{rcccccl} \bar{y}_i & > & 0 & \Longrightarrow & A_i\bar{x} & = & b_i \\ A_i\bar{x} & < & b_i & \Longrightarrow & \bar{y}_i & = & 0 \end{array} \qquad\qquad i = 1, \ldots, m \quad . \tag{2.22}$$

Si noti che (2.21) e (2.22) sono *equivalenti* a (2.20) quando $\bar{x}$ e $\bar{y}$ sono ammissibili rispettivamente per (P) e (D): infatti, l'ammissibilità implica che ciascun termine $\bar{y}_i(\, b_i - A_i\bar{x}\,)$ è non negativo, e la somma di m termini non negativi è zero se e solo se ciascun addendo è zero.

Per poter utilizzare il teorema degli scarti complementari direttamente, è necessario avere sia una soluzione primale sia una soluzione duale. Comunque, quando sia nota soltanto una soluzione primale ammissibile, è possibile verificare se sia ottima o meno tramite la ricerca di una soluzione duale ammissibile che formi con essa una coppia di soluzioni complementari. Infatti, vale la seguente caratterizzazione dell'ottimalità primale:

Proposizione 2.1 *Sia $\bar{x}$ una soluzione ammissibile per (P). Allora, $\bar{x}$ è ottima se e solo se esiste $\bar{y}$ ammissibile per (D) complementare a $\bar{x}$.*

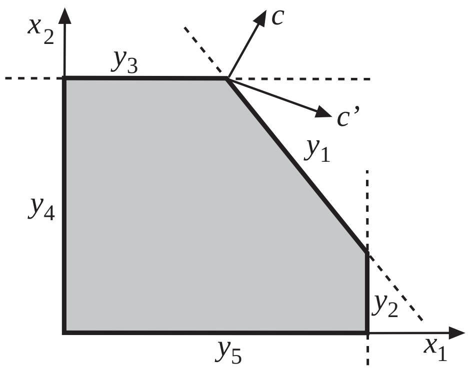
Dimostrazione La parte "se" del teorema è un'immediata conseguenza del teorema degli scarti complementari. Consideriamo allora il caso in cui $\bar{x}$ sia una soluzione ottima per (P): per il Teorema 2.12 esiste una soluzione $\bar{y}$ ottima per (D) e per il Teorema 2.11 si ha $c\bar{x} = \bar{y}b$, ovvero vale (2.20). $\diamond$

Una soluzione ammissibile $\bar{y}$ complementare a $\bar{x}$ soddisfa (2.22), e quindi deve valere $\bar{y}_i = 0$ per ogni $i \notin I$ (dove $I = I(\bar{x})$ indica l'insieme degli indici dei vincoli attivi). Poiché $\bar{y}A = c$, risulta anche $\bar{y}_I A_I = c$ e pertanto $\bar{y}_I$ è una soluzione del Duale Ristretto (D_R). Ritroviamo quindi che $\bar{x}$ è una soluzione ottima se e solo se esiste una soluzione $\bar{y}_I$ del Duale Ristretto (D_R), come già espresso dal Corollario 2.5.

Esempio 2.19: Applicazione del Teorema degli scarti complementari
Si consideri la seguente coppia di problemi duali:

$$
(P)\quad
\begin{array}{rrrrrl}
\max & x_1 & + & 2x_2 & & \\
 & x_1 & + & x_2 & \leq 5 & \\
 & x_1 & & & \leq 4 & \\
 & & & x_2 & \leq 3 & \\
 & -x_1 & & & \leq 0 & \\
 & & - & x_2 & \leq 0 &
\end{array}
\qquad
(D)\quad
\begin{array}{rrrrrrrrrrl}
\min & 5y_1 & + & 4y_2 & + & 3y_3 & & & & & \\
 & y_1 & + & y_2 & & & - & y_4 & & & = 1 \\
 & y_1 & & & + & y_3 & & & - & y_5 & = 2 \\
 & y_1 & , & y_2 & , & y_3 & , & y_4 & , & y_5 & \geq 0
\end{array}
$$

La soluzione ottima di (P) è $\bar{x} = [\ 2\,,\,3\ ]$, di valore $c\bar{x} = 8$, come si può verificare dalla figura qui a fianco. Per il Teorema 2.14 si ha quindi che per *qualsiasi* soluzione ottima $\bar{y}$ di (D) deve valere



$$
\begin{array}{r}
\bar{x}_1 < 4 \Longrightarrow \bar{y}_2 = 0 \\
-\bar{x}_1 < 0 \Longrightarrow \bar{y}_4 = 0 \\
-\bar{x}_2 < 0 \Longrightarrow \bar{y}_5 = 0
\end{array} \quad .
$$

Quindi, le uniche componenti possibilmente non nulle di qualsiasi soluzione ottima $\bar{y}$ di (D) devono soddisfare il sistema

$$y_1 = 1 \qquad , \qquad y_1 + y_3 = 2 \ .$$

Tale sistema ammette un unica soluzione $[\ y_1\,,\,y_3\ ] = [\ 1\,,\,1\ ]$, e pertanto esiste una sola candidata ad essere una soluzione duale ottima: $\bar{y} = [\ 1\,,\,0\,,\,1\,,\,0\,,\,0\ ]$. Poiché $\bar{y}$ è ammissibile ($\bar{y} \geq 0$), per il Teorema 2.14 essa è effettivamente ottima: infatti, anch'essa ha valore $\bar{y}b = 8$.

Se invece il vettore dei costi di (P) fosse stato $c' = [\ 3\,,\,-1\ ]$, ripetendo gli stessi ragionamenti si sarebbe giunti a

$$y_1 = 3 \qquad , \qquad y_1 + y_3 = -1$$

che ha come unica soluzione $[\ y_1\,,\,y_3\ ] = [\ 3\,,\,-4\ ]$; di conseguenza, l'unica soluzione che avrebbe rispettato gli scarti complementari con $\bar{x}$ sarebbe stata $\bar{y} = [\ 3\,,\,0\,,\,-4\,,\,0\,,\,0\ ] \not\geq 0$, quindi non ammissibile. Questo dimostra *algebricamente* che $[\ 2\,,\,3\ ]$ non è ottimo per il problema con vettore dei costi c', come è facile verificare *geometricamente*.

Analogamente, è possibile verificare l'ottimalità di una soluzione duale tramite la ricerca di una soluzione primale ammissibile ad essa complementare.

Proposizione 2.2 *Sia $\bar{y}$ una soluzione ammissibile per (D). Allora, $\bar{y}$ è ottima se e solo se esiste $\bar{x}$ ammissibile per (P) complementare a $\bar{y}$.*

Tutti i risultati precedenti sono validi per la coppia asimmetrica di problemi duali; tuttavia, si possono definire opportunamente condizioni degli scarti complementari per coppie duali in qualsiasi forma. Ad esempio, il seguente corollario fornisce quelle per la coppia simmetrica di problemi duali.

Corollario 2.6 *Siano $\bar{x}$ e $\bar{y}$ soluzioni ammissibili rispettivamente per (2.17) e (2.18), allora $\bar{x}$ e $\bar{y}$ sono soluzioni ottime se e solo se $\bar{y}(\ b - A\bar{x}\ ) = 0$ e $(\ \bar{y}A - c\ )\bar{x} = 0$.*

Esercizio 2.8 *Dimostrare il corollario 2.6.*

Interpretazione economica del teorema degli scarti complementari

Il problema (P) si interpreta naturalmente come il problema di utilizzare le risorse disponibili (vettore b), distribuendole fra un dato insieme di attività in modo da massimizzare il profitto: ogni colonna della matrice A corrisponde ad un'attività, ed il valore della corrispondente variabile x_j fornisce il livello dell'attività. In questa interpretazione, ciascun coefficiente a_{ij} della matrice A indica quante unità della risorsa i sono necessarie per effettuare un'unità dell'attività j; per questo A viene anche chiamata *matrice delle tecnologie*.

Sia $\bar{x}$ una soluzione ottima di (P) e $\bar{y}$ una soluzione ottima di (D). Consideriamo adesso una piccola variazione del vettore b, della forma $b(\varepsilon) = b + \varepsilon$, che lasci la soluzione ottima nell'intersezione degli iperpiani in cui era precedentemente (si veda la figura 2.9).

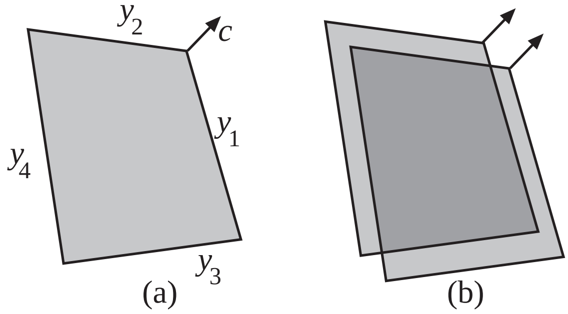


Figura 2.9: Costi ombra delle risorse

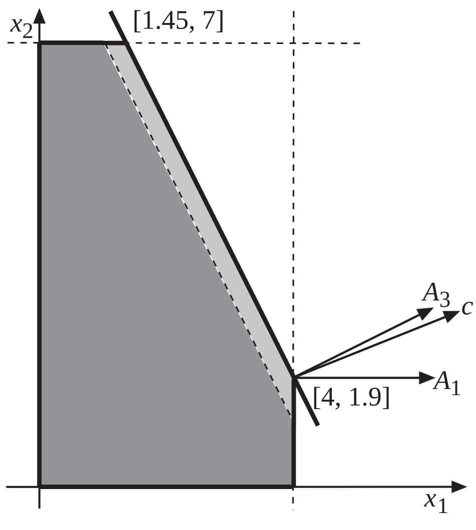
Chiaramente, l'aver modificato i termini noti non ha alcun effetto sulla ammissibilità di $\bar{y}$; inoltre, indicando con $\bar{x}(\varepsilon)$ il nuovo valore assunto da $\bar{x}$, le condizioni del Teorema degli scarti complementari valgono anche per la coppia di soluzioni $\bar{x}(\varepsilon)$ e $\bar{y}$. In altre parole, $\bar{x}(\varepsilon)$ e $\bar{y}$ sono soluzioni ottime, e la variazione del problema primale non ha avuto effetto sulla soluzione ottima del duale. Tale variazione ha però effetto sul valore della funzione obiettivo, che diventa $c\bar{x}(\varepsilon) = \bar{y}(b + \varepsilon) = \bar{y}b + \bar{y}\varepsilon$. Pertanto, nelle ipotesi fatte, il vettore $\bar{y}$ rappresenta il *gradiente* del valore ottimo della funzione obiettivo espresso in funzione della variazione ε di b, calcolato nell'origine ($\varepsilon = 0$). La singola componente $\bar{y}_i$ fornisce la variazione di valore ottimo della funzione obiettivo per una variazione unitaria[3] del valore della i-esima risorsa, pertanto essa indica il massimo valore che è ragionevole pagare per disporre di un'unità aggiuntiva di tale risorsa: in questo senso si dice che i valori ottimi delle variabili duali forniscono i *valori marginali* (detti anche *prezzi o costi ombra*) delle risorse. I valori ottimi delle variabili duali forniscono una valutazione del valore relativo che hanno le diverse risorse in quanto utilizzate nel processo produttivo definito dal problema (P). Risulta pertanto comprensibile come la variabile duale corrispondente ad un vincolo soddisfatto all'ottimo come disuguaglianza stretta (nell'esempio di figura 2.9 (a), y_3 e y_4, corrispondenti alle risorse b_3 e b_4), cioè ad una risorsa sovrabbondante, debba avere valore ottimo nullo.

Esempio 2.20: Costi ombra per il problema della Pintel

Riprendiamo, per esemplificare i concetti esposti, il problema della Pintel: $[4, 1]$ e $[100, 0, 200, 0, 0]$ sono rispettivamente soluzioni ottime del primale e del duale, con valore della funzione obiettivo pari a 2200. Chiediamoci adesso cosa accadrebbe se il reparto tecnico fosse in grado, noleggiando altri macchinari, di aumentare la produzione settimanale di wafers del 10%, portandola a 3300 unità: questo cambierebbe il terzo vincolo in

$$2x_1 + x_2 \leq 9.9$$

(la risorsa aumenta del 10%), lasciando tutti gli altri inalterati. Come si vede nella figura qui a fianco, questo corrisponde graficamente a traslare la faccia corrispondente al terzo vincolo nel verso di A_3, cosicché il vertice intersezione delle facce corrispondenti al primo ed al terzo vincolo si sposta nel punto $[4, 1.9]$. Poiché non sono cambiati i vincoli attivi, né i vincoli del duale, la soluzione $[100, 0, 200, 0, 0]$ continua ad essere ammissibile per il duale. Essa inoltre soddisfa le condizioni degli scarti complementari con la nuova soluzione primale $[4, 1.9]$: infatti, il vettore c è ancora esprimibile come combinazione non negativa dei gradienti del primo e del terzo vincolo, con gli stessi coefficienti. Pertanto, il vertice intersezione delle facce corrispondenti al primo ed al terzo vincolo continua ad essere ottimo: la posizione del vertice in $\mathbb{R}^2$ è però diversa, ed infatti cambia il valore della funzione obiettivo, che è ora 2380. Osserviamo che l'incremento è dato dal valore della variabile duale corrispondente al terzo vincolo (200) per l'incremento della terza risorsa (0.9);



quindi, è possibile stimare se il noleggio dei nuovi macchinari sia o no conveniente confrontando il costo di noleggio con il prezzo ombra del vincolo corrispondente ai wafer.

Si noti che lo stesso ragionamento potrebbe essere ripetuto per tutti gli altri vincoli del primale, anche quelli che non sembrano rappresentare risorse nel problema reale. Si consideri ad esempio il caso in cui il reparto marketing della Pintel fosse in grado, attraverso un'opportuna campagna pubblicitaria, di garantire la vendita di 100.000 Pintium in più del previsto, aumentando la "prima risorsa" di un'unità; il valore marginale del secondo vincolo potrebbe essere usato per stimare la convenienza dell'operazione, noti i costi della campagna pubblicitaria. È però necessario rilevare che il corretto uso dei prezzi ombra richiede cautela: bisogna sempre verificare che la soluzione primale resti ammissibile, e

[3]Naturalmente nell'ipotesi che una tale variazione non alteri la soluzione ottima del duale.

quindi ottima, in corrispondenza al previsto aumento di risorse. Ad esempio, l'aumento di un'unità della "prima risorsa", trasformando il primo vincolo in $x_1 \leq 5$, porterebbe l'intersezione del primo e terzo vincolo nel punto $[5, -1]$, che non è ammissibile: è facile vedere che, per via del vincolo 3, il massimo incremento della prima risorsa che produce un effettivo miglioramento del valore della funzione obiettivo è pari a 0.5. È comunque possibile, sfruttando alcune informazioni fornite dagli algoritmi che vedremo nel prossimo paragrafo, stimare l'intervallo in cui la variazione di una risorsa porta ad un miglioramento della funzione obiettivo esattamente pari a quello previsto dal prezzo ombra.

2.2.5 Soluzioni complementari e basi

Data la coppia asimmetrica di problemi duali, abbiamo già visto che ad una *base* B vengono associati la *matrice di base* A_B ed il punto $\bar{x} = A_B^{-1} b_B \in \mathbb{R}^n$, che è detto *soluzione primale di base*; nel caso sia ammissibile, $\bar{x}$ corrisponde ad un vertice del poliedro che individua la regione ammissibile di (P). Vogliamo costruire una soluzione duale che formi con $\bar{x}$ una coppia di soluzioni complementari: considerando anche l'insieme degli indici non in base $N = \{1, \ldots, m\} \setminus B$, possiamo associare alla base B la *soluzione duale di base*

$$\bar{y} = [\ \bar{y}_B\ ,\ \bar{y}_N\ ] = \left[\ c A_B^{-1}\ ,\ 0\ \right] .$$

Tale soluzione soddisfa per costruzione la proprietà $\bar{y}A = c$ richiesta dalla Definizione 2.2; inoltre, segue facilmente da (2.22) che $\bar{y}$ è complementare a $\bar{x}$. Infatti, per $i \in B$ si ha $A_i \bar{x} = b_i$ (e $\bar{y}_i$ può essere diverso da zero), mentre per $i \in N$ si ha $\bar{y}_i = 0$ (e $A_i \bar{x}$ può essere diverso da b_i). Da ciò segue:

Proposizione 2.3 *La coppia di soluzioni associate ad una base B soddisfa le condizioni degli scarti complementari (2.20).*

Le soluzioni $\bar{x}$ e $\bar{y}$ associate ad una base vengono pertanto dette *soluzioni di base complementari*; qualora siano entrambe ammissibili, il teorema degli scarti complementari garantisce la loro ottimalità. In particolare, $\bar{x}$ e $\bar{y}$ risultano essere soluzioni di base *ammissibili*, *non ammissibili*, *degeneri* o *non degeneri* se verificano le condizioni riportate nella seguente tabella:

	$\bar{x}$	$\bar{y}$
ammissibile	$A_N \bar{x} \leq b_N$	$\bar{y}_B \geq 0$
non ammissibile	$\exists i \in N : A_i \bar{x} > b_i$	$\exists i \in B : \bar{y}_i < 0$
degenere	$\exists i \in N : A_i \bar{x} = b_i$	$\exists i \in B : \bar{y}_i = 0$
non degenere	$\forall i \in N : A_i \bar{x} \neq b_i$	$\forall i \in B : \bar{y}_i \neq 0$

Una base B è detta primale [duale] ammissibile, non ammissibile, degenere o non degenere a seconda che lo sia la soluzione di base primale [duale] ad essa associata. Si può dimostrare che anche $\bar{y}$ è un vertice della regione ammissibile di (D) nel caso in cui sia ammissibile.

Esempio 2.21: Soluzioni di base complementari

Si consideri la coppia di problemi duali rappresentata nella tabella e nel grafico di Figura 2.10. Per $B = \{2, 5\}$ si ha

	x_1	x_2		min
y_1	-2	1		1
y_2	1	-2		-4
y_3	1	0	$\leq$	8
y_4	1	1		14
y_5	0	-1		-4
	||			
max	1	3		

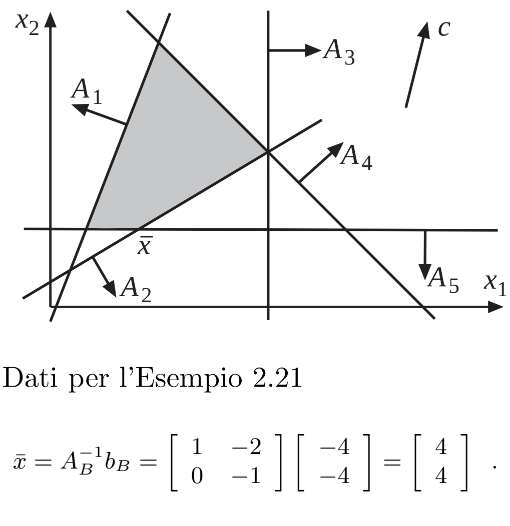


Figura 2.10: Dati per l'Esempio 2.21

$$A_B = A_B^{-1} = \begin{bmatrix} 1 & -2 \\ 0 & -1 \end{bmatrix} \quad \Longrightarrow \quad \bar{x} = A_B^{-1} b_B = \begin{bmatrix} 1 & -2 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} -4 \\ -4 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \end{bmatrix} .$$

Si ha quindi $A_1\bar{x} = -4 < 1$, $A_3\bar{x} = 4 < 8$ e $A_4\bar{x} = 8 < 14$, cioè $\bar{x}$ è una soluzione di base primale ammissibile; la corrispondente $\bar{y}$ è data da $\bar{y} = [\bar{y}_B \; 0]$ dove $\bar{y}_B = cA_B^{-1} = [1, -5]$ e quindi non è ammissibile ($\bar{y}_5 = -5 < 0$). Si noti che $\bar{x}$ e $\bar{y}$ sono soluzioni non degeneri.

Consideriamo adesso il problema di stabilire se una data soluzione primale $\bar{x}$ sia una soluzione di base: dato l'insieme I degli indici dei vincoli attivi, si ha che $\bar{x}$ è una soluzione di base se e solo se $rango(A_I) = rango(A) = n$. Infatti, $rango(A_I) = n$ implica che esista $B \subseteq I$ tale che $|B| = n$ e $det(A_B) \neq 0$, e quindi $\bar{x} = A_B^{-1}b_B$. Se $|I| = n$, allora $B = I$ e $\bar{x}$ è una soluzione di base *non degenere*. In questo caso, esiste un solo vettore $\bar{y}$ tale che $(\bar{x}, \bar{y})$ sia una coppia di soluzioni di base complementari; infatti la matrice di base A_B associata a $\bar{x}$ è univocamente determinata, e di conseguenza è univocamente determinato il vettore $\bar{y}$. Viceversa, se $|I| > n$, allora $\bar{x}$ è una soluzione *degenere*: in questo caso più matrici di base possono corrispondere a $\bar{x}$, e conseguentemente più soluzioni di base di (D) possono costituire con $\bar{x}$ una coppia di soluzioni complementari.

Esempio 2.22: Soluzioni primali degeneri

Nel problema dell'esempio precedente, si consideri $\bar{x} = (8, 6)$: si ha

$$\begin{array}{rcrcr} A_1\bar{x} & = & -10 & < & 1 \\ A_2\bar{x} & = & -4 & = & -4 \\ A_3\bar{x} & = & 8 & = & 8 \\ A_4\bar{x} & = & 14 & = & 14 \\ A_5\bar{x} & = & -6 & < & -4 \end{array}$$

e quindi $\bar{x}$ è ammissibile e soddisfa come equazione i vincoli 2, 3 e 4, cioè $I = \{2, 3, 4\}$. $\bar{x}$ è quindi una soluzione di base degenere, cui corrispondono le basi $B' = \{2, 3\}$, $B'' = \{2, 4\}$ e $B''' = \{3, 4\}$, come è facilmente possibile verificare. Le soluzioni duali complementari corrispondenti a $\bar{x}$ sono

$$\bar{y}' = [\,0,\ -3/2,\ 5/2,\ 0,\ 0\,] \quad , \quad \bar{y}'' = [\,0,\ -2/3,\ 0,\ 5/3,\ 0\,] \quad , \quad \bar{y}''' = [\,0,\ 0,\ -2,\ 3,\ 0\,]$$

nessuna delle quali è ammissibile per (D); si ha comunque $c\bar{x} = \bar{y}'b = \bar{y}''b = \bar{y}'''b = 26$.

Consideriamo adesso il problema di stabilire se una data soluzione duale $\bar{y}$, tale che $\bar{y}A = c$, sia una soluzione di base: dato $J = J(\bar{y}) = \{\ j \ : \ \bar{y}_j \neq 0\ \}$, si ha che $\bar{y}$ è una soluzione di base se e solo se tutte le righe di A_J sono linearmente indipendenti. Infatti, se $|J| = n$ allora la matrice di base corrispondente a $\bar{y}$ è $A_B = A_J$. In questo caso esiste un solo vettore $\bar{x}$ tale $(\bar{x}, \bar{y})$ sia una coppia di soluzioni di base complementari; infatti la matrice di base A_B associata a $\bar{y}$ è univocamente determinata e di conseguenza è univocamente determinato il vettore $\bar{x}$. Se $|J| < n$, allora $\bar{y}$ è una soluzione degenere: a tale soluzione corrispondono più matrici di base, ottenute aggiungendo $n - |J|$ righe di A alla matrice A_J in modo che la matrice A_B risultante abbia determinante non nullo (tali righe esistono per l'ipotesi che $rango(A) = n$). Conseguentemente, più soluzioni di base $\bar{x}$ di (P) possono costituire con $\bar{y}$ una coppia di soluzioni complementari.

Esempio 2.23: Soluzioni duali degeneri

Si consideri il problema degli esempi precedenti con una diversa funzione obiettivo, data da $c = [1, 1]$, e sia $\bar{y} = [0, 0, 0, 1, 0]$, cosicché $\bar{y}b = 14$. $\bar{y}$ è una soluzione ammissibile di base degenere per (D): infatti risulta $J = \{4\}$. Si può facilmente verificare che le basi per $\bar{y}$ sono $B' = \{1, 4\}$, $B'' = \{2, 4\}$, $B''' = \{3, 4\}$ e $B'''' = \{5, 4\}$, infatti risulta $cA_{B'}^{-1} = cA_{B''}^{-1} = cA_{B'''}^{-1} = cA_{B''''}^{-1} = [0, 1]$. Le soluzioni primali di base complementari a $\bar{y}$ sono riportate nella seguente tabella:

$\bar{x}' = [\ 13/3\ ,\ 29/3\ ]$	$\bar{x}'' = \bar{x}''' = [\ 8\ ,\ 6\ ]$	$\bar{x}'''' = [\ 10\ ,\ 4\ ]$
(ammissibile)	(ammissibile degenere)	(non ammissibile)

In assenza di degenerazione, ad una soluzione di base è pertanto associata una sola base ed una sola soluzione di base complementare. In questo caso è possibile verificare se la soluzione sia ottima o meno tramite l'ammissibilità della soluzione complementare.

Teorema 2.15 *Siano $\bar{x}$ una soluzione di base primale ammissibile non degenere e B la corrispondente base. Allora, $\bar{x}$ è ottima se e solo se la soluzione duale complementare $\bar{y} = [cA_B^{-1}, 0]$ è ammissibile.*

Dimostrazione La parte "se" del teorema segue immediatamente dalla Proposizione 2.3 e dal teorema degli scarti complementari. Consideriamo allora il caso in cui $\bar{x}$ sia una soluzione ottima. Per la Proposizione 2.1 esiste una soluzione duale ammissibile $\bar{y}$ complementare a $\bar{x}$ e quindi valgono le condizioni degli scarti complementari (2.20). Poiché $\bar{x}$ è non degenere e quindi $I = B$, risulta $\bar{y}_{\bar{I}} = \bar{y}_N = 0$. Dall'ammissibilità di $\bar{y}$ segue che $\bar{y}_B A_B = c$, ovvero $\bar{y}_B = cA_B^{-1}$. Quindi, $\bar{y}$ è la soluzione duale di base associata a B. $\diamond$

È possibile dimostrare l'analoga caratterizzazione dell'ottimalità di una soluzione di base duale non degenere.

Teorema 2.16 *Siano $\bar{y}$ una soluzione di base duale ammissibile non degenere e B la corrispondente base. Allora, $\bar{y}$ è ottima se e solo se la soluzione primale complementare $\bar{x} = A_B^{-1} b_B$ è ammissibile.*

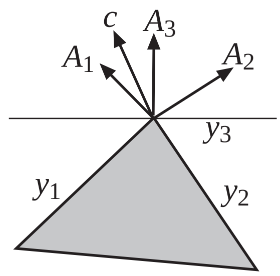
Nel caso di soluzioni di base degeneri, invece, l'ammissibilità della soluzione di base complementare fornisce soltanto una condizione sufficiente di ottimalità, come mostra il seguente esempio.

Esempio 2.24: Condizioni di ottimo in presenza di degenerazione

Si considerino, per il problema in figura qui accanto, la soluzione di base $\bar{x}$ tale che $I = \{1, 2, 3\}$ e le basi $B' = \{1, 2\}$, $B'' = \{1, 3\}$ e $B''' = \{2, 3\}$. Si ha che:

$$\begin{aligned} c \in cono(\, A_1 \,,\, A_2 \,) &\Longrightarrow \bar{y}_{B'} \geq 0 \\ c \in cono(\, A_1 \,,\, A_3 \,) &\Longrightarrow \bar{y}_{B''} \geq 0 \quad ; \\ c \notin cono(\, A_2 \,,\, A_3 \,) &\Longrightarrow \bar{y}_{B'''} \not\geq 0 \end{aligned}$$

quindi, $A_{B'''}$ non soddisfa la condizione di ottimo, pur essendo $\bar{x}$ una soluzione ottima.



In presenza di degenerazione, ad una soluzione di base corrispondono più soluzioni complementari, una per ciascuna base. Si possono però dimostrare gli analoghi dei Teoremi 2.15 e 2.16:

Teorema 2.17 *Sia $\bar{x}$ una soluzione di base primale ammissibile. Allora, $\bar{x}$ è ottima se e solo se esiste una base B associata a $\bar{x}$ tale che la soluzione duale complementare $\bar{y} = [\, cA_B^{-1} \,,\, 0 \,]$ sia ammissibile.*

Teorema 2.18 *Sia $\bar{y}$ una soluzione di base duale ammissibile. Allora, $\bar{y}$ è ottima se e solo se esiste una base B associata a $\bar{y}$ tale che la soluzione primale complementare $\bar{x} = A_B^{-1} b_B$ sia ammissibile.*

Evitiamo di discutere le dimostrazioni perché, come quella del Lemma di Farkas, possono essere ricavate da risultati relativi agli algoritmi che presenteremo nel seguito.

Esercizio 2.9 *Costruire esempi di problemi (P), (D) rappresentabili in $\mathbb{R}^2$ aventi soluzione ottima degenere, individuando una matrice di base che non soddisfi le condizioni di ottimo.*

Esercizio 2.10 *Costruire, se esiste, una coppia di problemi (P), (D) tale che le soluzioni ottime $\bar{x}$ e $\bar{y}$ siano entrambe degeneri.*

Esercizio 2.11 *Dimostrare il seguente teorema: Se (P) $[(D)]$ ammette più soluzioni ottime di base, allora la soluzione ottima di base di (D) $[(P)]$ è degenere.*

2.3 Algoritmi del Simplesso

La teoria della dualità sviluppata nel paragrafo precedente fornisce gli strumenti per la costruzione di un algoritmo per la soluzione dei problemi di *PL*, che chiameremo genericamente *algoritmo del simplesso.* Questo algorimto costituisce il primo approccio computazionalmente efficiente per la soluzione di problemi di *PL*; originalmente proposto da G.B. Dantzig [1951] a partire da un idea di J. Von Neumman, il metodo del simplesso è stato sviluppato in diverse versioni e sta alla base dei più diffusi codici di *PL*. Quelle che presenteremo sono solamente alcune delle diverse varianti del metodo sviluppate a partire dall'algoritmo originale di Dantzig.

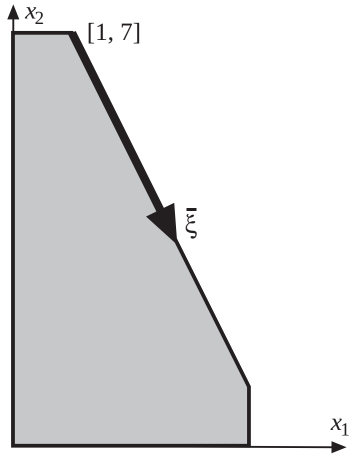
2.3.1 L'algoritmo del Simplesso Primale

L'algoritmo del simplesso considera l'insieme dei vertici del problema primale e, ad ogni passo, cerca un vertice che migliori il valore della funzione obiettivo. L'individuazione di un tale vertice richiede la determinazione di una direzione ammissibile di crescita, come evidenziato nel paragrafo 2.2.3.

Esempio 2.25: Esecuzione "informale" del Simplesso Primale
Riprendiamo il problema della Pintel, e consideriamo il vertice $\bar{x} = [\,1\,,\,7\,]$, che già sappiamo non essere una soluzione ottima; tale vertice è una soluzione primale di base non degenere in quanto $B = I(\bar{x}) = \{2,3\}$. Poiché $\bar{x}$ non è una soluzione ottima per il problema, dovrà necessariamente esistere una direzione ξ che sia di crescita, cioè tale per cui $c\xi > 0$, e che sia ammissibile, cioè tale che valga $A_B\xi \leq 0$. Cerchiamo allora di determinare tale direzione considerando la coppia di sistemi studiata dal Lemma di Farkas:

$$\begin{array}{rcrcl} & & \xi_2 & \leq & 0 \\ 2\xi_1 & + & \xi_2 & \leq & 0 \\ 500\xi_1 & + & 200\xi_2 & > & 0 \end{array} \quad , \quad \begin{array}{rcrcl} & & 2y_2 & = & 500 \\ y_1 & + & y_2 & = & 200 \\ y_1 & , & y_2 & \geq & 0 \end{array} \quad .$$

Per verificare quale tra i due sistemi ammetta soluzione, calcoliamo la sola soluzione candidata a risolvere il sistema duale, $\bar{y}_B = cA_B^{-1} = [\,-50\,,\,250\,]$. Essendo $\bar{y}_2 < 0$, tale soluzione non è ammissibile: sappiamo pertanto che esiste una soluzione per il sistema primale, vale a dire una direzione ammissibile di crescita per $\bar{x}$. Infatti $\bar{\xi} = [\,0.5\,,\,-1\,]$ è ammissibile per il sistema primale.
Come evidenziato nella figura qui accanto, $\bar{\xi}$ individua la direzione di spostamento che va dal vertice $[\,1\,,\,7\,]$ verso il vertice ad esso *adiacente* $[\,4\,,\,1\,]$; in particolare, la direzione è uno dei (due) generatori del cono delle direzioni ammissibili per $[\,1\,,\,7\,]$, espresso come cono finitamente generato. Lungo tale direzione ci si sposta rimanendo sulla faccia individuata dal vincolo $2x_1 + x_2 \leq 9$, mentre ci si allontana dalla faccia individuata dal vincolo $x_2 \leq 7$. Possiamo allora migliorare il valore della funzione obiettivo spostandoci lungo $\bar{\xi}$ il più possibile, purché si rimanga all'interno della regione ammissibile: un tale spostamento ci porta al vertice $[4,1]$, che come già sappiamo è il punto ottimo. Per verificare tale ottimalità iteriamo il ragionamento, e cerchiamo di determinare se esistono direzioni ammissibili di crescita rispetto a $[\,4\,,\,1\,]$: poiché il sistema duale



$$\begin{array}{rcrcl} y_1 & + & 2y_3 & = & 500 \\ & & y_3 & = & 200 \\ y_1 & , & y_3 & \geq & 0 \end{array}$$

ammette la soluzione non negativa $[\,100\,,\,200\,]$, si può concludere che $[\,4\,,\,1\,]$ è una soluzione ottima del problema.

Formalizziamo ora le idee dell'esempio precedente, specificando, in particolare, le modalità di individuazione di una direzione ammissibile di crescita, se essa esiste. Per questo ci serviremo dei risultati sviluppati nel paragrafo 2.1.1, ed in particolare di (2.16). Assumiamo di avere a disposizione una base ammissibile B e la corrispondente soluzione di base $\bar{x} = A_B^{-1}b_B$; supponiamo inoltre, per il momento, che la base sia non degenere, ossia $B = I(\bar{x})$. Vogliamo determinare se esiste una direzione ammissibile di crescita rispetto a $\bar{x}$, ossia $\xi \in C(\bar{x})$ tale che $c\xi > 0$. Sfruttiamo adesso la caratterizzazione (2.16) di $C(\bar{x})$: i generatori del cono sono gli opposti delle colonne dell'inversa della matrice di base. Come vedremo, ciascuna di queste direzioni è "naturalmente associata" ad uno specifico vincolo in base; per sottolineare questo indicizziamo le direzioni con gli indici $i \in B$ mediante

$$\xi_i = -A_B^{-1}u_{B(i)} \qquad\qquad i = 1, \ldots, n$$

dove $B(i)$ indica la posizione dell'indice i in B (la riga di A_B in cui si trova A_i), mentre $u_j \in \mathbb{R}^n$ indica il vettore che ha tutte le componenti nulle tranne la j-esima, che vale 1 (si tratta cioè del j-esimo vettore della base canonica di $\mathbb{R}^n$). Usando il Lemma 2.1 possiamo quindi affermare che *esiste* $\xi \in C(\bar{x})$ *tale che* $c\xi > 0$ *se e solo se* $c\xi_h > 0$ *per un qualche* $h \in B$.

Nel caso di una base non degenere disponiamo quindi di un meccanismo in grado di determinare una direzione ammissibile di crescita, se ce n'è una oppure dimostrare che $\bar{x}$ è una soluzione ottima. Infatti, $c\xi_i \leq 0$ per ogni $i \in B$ equivale a $\bar{y}_i = cA_B^{-1}u_{B(i)} = -c\xi_i \geq 0$ per ogni $i \in B$, ossia $\bar{y} = [\,\bar{y}_B\,,\,0\,] \geq 0$: pertanto, $\bar{x}$ e $\bar{y}$ sono entrambe ammissibili (rispettivamente per il primale ed il duale), e siccome la Proposizione 2.3 garantisce che sono anche complementari, il Teorema degli Scarti Complementari ci garantisce di aver risolto entrambi i problemi.

Nel caso in cui $\bar{x}$ sia una soluzione di base degenere, però, il risultato vale in una sola direzione. In particolare, se $\bar{y}_B \geq 0$ allora possiamo comunque affermare di aver determinato una soluzione ottima. Se invece $c\xi_h > 0$ per un qualche $h \in B$, non possiamo per questo affermare che $\bar{x}$ non sia ottima, perché in effetti non abbiamo alcuna garanzia che ξ_h sia una direzione ammissibile per $\bar{x}$. Infatti, ξ_h è un generatore del cono $C_B = \{ \xi \in \mathbb{R}^n \;:\; A_B\xi \leq 0 \}$, che peró non coincide col cono delle direzioni ammissibili $C(\bar{x})$ in quanto $I(\bar{x}) \supset B$; si ha quindi $C(\bar{x}) \subset C_B$, e può accadere che $\xi_h \notin C(\bar{x})$, ossia esiste un indice $i \in I \setminus B$ tale che $A_i\xi_h > 0$. Si può infatti dimostrare che $C(\bar{x})$ coincide con l'intersezione di tutti i coni C_B al variare di $B \subseteq I(\bar{x})$, e che i generatori di $C(\bar{x})$ sono tutti e soli i vettori che sono generatori di uno dei coni C_B e che appartengono a $C(\bar{x})$ (si ricordi il Teorema 2.7). Questo equivale al fatto che la direzione ξ_h è una soluzione non del "vero" Primale Ristretto, ma di un suo rilassamento in cui sono stati arbitrariamente ignorati i vincoli in $I \setminus B$: se il sistema così approssimato non ha soluzioni allora non ne ha neanche quello originario e deve avere soluzione il Duale Ristretto (infatti si può affermare che $\bar{x}$ è ottima), ma determinare una soluzione del sistema approssimato non garantisce di averne determinata una del Primale Ristretto originario.

Esempio 2.26: Direzioni ξ_h non ammissibili
Si consideri il seguente coppia di problemi duali

$$
(P) \quad \begin{array}{rrrrl} \max & x_1 & + & x_2 & \\ & & - & x_2 & \leq 1 \\ & -x_1 & - & x_2 & \leq 1 \\ & x_1 & & & \leq 0 \\ & x_1 & + & x_2 & \leq 2 \end{array} \quad , \quad (D) \quad \begin{array}{rrrrrrrl} \min & y_1 & + & y_2 & & & - & y_4 \\ & & - & y_2 & - & y_3 & - & y_4 \;\; = 0 \\ & -y_1 & - & y_2 & & & + & y_4 \;\; = 1 \\ & y_1 & , & y_2 & , & y_3 & , & y_4 \;\; \geq 0 \end{array}
$$

e la base $B = \{1, 2\}$. Le corrispondenti matrice e soluzioni di base sono

$$
A_B = \begin{bmatrix} 0 & -1 \\ -1 & -1 \end{bmatrix} \;,\; \bar{x} = A_B^{-1}b_B = \begin{bmatrix} 1 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \;,\; \bar{y}_B = cA_B^{-1} = [\; 1 \,,\, 1 \;] \begin{bmatrix} 1 & -1 \\ -1 & 0 \end{bmatrix} = [\; 0 \,,\, -1 \;]
$$

ed è immediato verificare che la soluzione $\bar{x}$ è ammissibile; inoltre, l'insieme degli indici dei vincoli attivi è $I = \{1, 2, 3\}$ e quindi $\bar{x}$ è degenere. Infatti, si ha

$$
C_B = \left\{ \begin{array}{rrrl} & - & \xi_2 & \leq 0 \\ -\xi_1 & - & \xi_2 & \leq 0 \end{array} \right. \quad \supset \quad C(\bar{x}) = \left\{ \begin{array}{rrrl} & - & \xi_2 & \leq 0 \\ -\xi_1 & - & \xi_2 & \leq 0 \\ \xi_1 & & & \leq 0 \end{array} \right. .
$$

I generatori di C_B sono gli opposti delle colonne di A_B^{-1}; il fatto che $\bar{y}_2 = -1$ corrisponde al fatto che

$$
c\xi_2 = [\; 1 \,,\, 1 \;] \begin{bmatrix} 1 \\ 0 \end{bmatrix} 1 = -\bar{y}_2 > 0
$$

e quindi ξ_2 è una direzione di crescita. È però facile verificare che $\xi_2 \notin C(\bar{x})$ ($A_3\xi_2 > 0$), per cui la direzione non è ammissibile per $\bar{x}$.

Il fatto che una direzione di crescita ξ si riveli non essere ammissibile non la rende per ció stesso inutilizzabile; come vedremo, uno spostamento "degenere" lungo una direzione di questo tipo causa comunque un *cambiamento di base*, il che permette di proseguire con l'algoritmo. Scegliamo quindi per il momento di ignorare il problema, e di utilizzare la direzione come se fosse ammissibile. Per questo calcoliamo il massimo passo di spostamento che è possibile effettuare lungo ξ mantenendo l'ammissibilità, ossia il massimo valore di λ per cui la soluzione $x(\lambda) = \bar{x} + \lambda\xi$ risulta essere ammissibile (sapendo che questo valore sarà non negativo, in quanto $x(0) = \bar{x}$ è ammissibile). Per $i \in B$, risulta

$$
A_i x(\lambda) = A_i\bar{x} + \lambda A_i\xi \leq A_i\bar{x} \leq b_i
$$

per ogni $\lambda \geq 0$, in quanto $A_i\xi \leq 0$. Analogamente, se $i \in N$ è tale che $A_i\xi \leq 0$ la soluzione $x(\lambda)$ soddisfa il vincolo i-esimo per ogni valore non negativo di λ. Se invece $A_i\xi > 0$, abbiamo

$$
A_i x(\lambda) = A_i\bar{x} + \lambda A_i\xi \leq b_i \quad \Longleftrightarrow \quad \lambda \leq (\, b_i - A_i\bar{x} \,)/(\, A_i\xi,) \;\; .
$$

Pertanto, il massimo passo che può essere effettuato lungo ξ a partire da $\bar{x}$ senza violare il vincolo i-esimo risulta essere

$$
\lambda_i = \left\{ \begin{array}{ll} \frac{b_i - A_i\bar{x}}{A_i\xi} & \text{se } A_i\xi > 0 \\ +\infty & \text{altrimenti} \end{array} \right. . \tag{2.23}
$$

Scegliendo il più piccolo di questi valori, ovvero

$$\bar{\lambda} = \min\{\, \lambda_i \ :\ i \in N \,\} \ , \tag{2.24}$$

si ottiene il massimo passo di spostamento consentito. In particolare, se $A_N\xi \leq 0$, abbiamo $\bar{\lambda} = +\infty$ e $x(\lambda)$ è ammissibile per ogni valore positivo di λ: poiché ξ è una direzione di crescita, il valore della funzione obiettivo cresce indefinitamente al crescere di λ, e siamo nel caso in cui il problema primale (P) è illimitato ed il problema duale (D) è vuoto. Se invece esiste un indice $i \in N$ tale che $A_i\xi > 0$, abbiamo $\bar{\lambda} < +\infty$: $x(\lambda)$ è ammissibile per ogni $\lambda \in [0, \bar{\lambda}]$ e non ammissibile per $\lambda > \bar{\lambda}$. Nel caso in cui $\bar{x}$ sia una soluzione di base non degenere, abbiamo $\lambda_i > 0$ per ogni $i \in N$ e quindi $\bar{\lambda} > 0$: come già verificato in precedenza, la direzione ξ è ammissibile. Possiamo quindi effettuare il massimo spostamento consentito $\bar{\lambda}$, spostandoci sul punto $x(\bar{\lambda})$ e migliorando strettamente il valore della funzione obiettivo ($cx(\bar{\lambda}) > c\bar{x}$). . Abbiamo definito in tal modo la tipica iterazione dell'Algoritmo del Simplesso. Per poter ripetere il procedimento, tuttavia, dobbiamo essere certi che $x(\bar{\lambda})$ sia un vertice del poliedro come lo era $\bar{x}$: è possibile dimostrare che questo è in effetti il caso. Questo risultato si basa su una proprietà peculiare delle direzioni ξ_h: $A_i\xi_h = 0$ per ogni $i \in B \setminus \{h\}$, mentre $A_h\xi_h = -1$, come è immediato verificare dalla definizione di ξ_h. Geometricamente, questo significa che ξ_h è perpendicolare a tutti gli A_i in base (tranne A_h), e che quindi "punta all'interno della frontiera" degli iperpiani corrispondenti; in altri termini, per ogni possibile valore di λ

$$A_i x(\lambda) = A_i(\bar{x} + \lambda\xi_h) = A_i\bar{x} = b_i \qquad\qquad i \in B \setminus \{h\} \ , \tag{2.25}$$

ossia i vincoli in base (tranne l'h-esimo), che sono per definizione attivi in $\bar{x}$, sono anche attivi in $x(\lambda)$. Invece il fatto che $A_h\xi_h = -1$ significa che A_h non è attivo in $x(\lambda)$ per $\lambda > 0$; infatti, al crescere di λ il punto "si allontana dalla frontiera dell'h-esimo iperpiano", verso l'interno del semispazio corrispondente. Da questo segue:

Lemma 2.4 *Data una base ammissibile B, la corrispondente soluzione di base $\bar{x}$, una direzione $\xi = \xi_h$ per $h \in B$, ed un indice $k \in N$ tale che $\bar{\lambda} = \bar{\lambda}_k$, $B' = B \cup \{k\} \setminus \{h\}$ è una base e $x(\bar{\lambda})$ è la corrispondente soluzione primale di base.*

Dimostrazione Essendo $A_i\xi = 0$, $\forall i \in B \setminus \{h\}$, si ha che ξ è ortogonale al sottospazio generato da $A_{B\setminus\{h\}}$. Inoltre, essendo $A_k\xi > 0$ (dato che $\bar{\lambda}_k < +\infty$, cf. (2.23)), il vettore A_k non può appartenere a tale sottospazio e, quindi, non può essere espresso come combinazione lineare dei vettori A_i, $i \in B \setminus \{h\}$. Da questo e dal fatto che, per ipotesi, le righe di $A_{B\setminus\{h\}}$ sono linearmente indipendenti, segue che B' è una base.

Per dimostrare che $x(\bar{\lambda})$ sia la soluzione di base associata a B', basta verificare che $B' \subseteq I(x(\bar{\lambda}))$. Questo per $i \in B \setminus \{h\}$ segue da (2.25), mentre per $i = k$ segue da (2.24): il passo scelto è precisamente quello che rende attivo il vincolo k-esimo, come è immediato verificare. $\diamond$

Nonostante l'analisi precedente sia stata sviluppata per il caso non degenere, è facile verificare che essa si estende senza alcun problema a quello degenere, in cui possiamo avere sia $\bar{\lambda} > 0$ che $\bar{\lambda} = 0$. Nel primo caso si effettua lo spostamento analogamente al caso non degenere. Se invece il massimo spostamento consentito è nullo, la direzione $\xi = \xi_h$ individuata non è ammissibile in quanto esiste un indice $k \in I \setminus B$ tale che $A_k\xi > 0$, e quindi $\xi \notin C(\bar{x})$. Poiché $\bar{\lambda} = \bar{\lambda}_k = 0$, $x(\bar{\lambda}) = \bar{x}$ e quindi non abbiamo cambiato vertice; abbiamo comunque individuato una *diversa base* $B \cup \{k\} \setminus \{h\}$ che individua lo stesso vertice $\bar{x}$. Parliamo in questo caso di un *cambiamento di base degenere.*

Un cambiamento di base non degenere, ovvero tale che $\bar{\lambda} > 0$, consente di spostarsi da un vertice della regione ammissibile ad un altro; i due vertici sono collegati da uno spigolo e vengono detti *adiacenti.* Inoltre, il nuovo vertice migliora il valore della funzione obiettivo rispetto al vertice precedente. Un cambiamento di base degenere, ovvero tale che $\bar{\lambda} = 0$, non consente di cambiare vertice ma fornisce una nuova base relativa allo stesso vertice. Utilizzando una regola che impedisca di individuare basi già analizzate precedentemente (si veda il Teorema 2.19) nelle successive iterazioni degeneri, nel caso pessimo si analizzano tutte le basi relative al vertice: quindi, in un numero finito di passi o si determina una direzione di crescita ammissibile, oppure si certifica l'ottimalità del vertice.

Esempio 2.27: Cambiamenti di base degeneri
Per esemplificare lo sviluppo precedente di consideri la seguente coppia di problemi duali

$$
(P)\quad
\begin{array}{rrrrrr}
\max & & & x_2 & & \\
 & & - & x_2 & \leq & 1 \\
 & -x_1 & - & x_2 & \leq & 1 \\
 & -x_1 & & & \leq & 0 \\
 & -x_1 & + & x_2 & \leq & -1 \\
 & x_1 & & & \leq & 2
\end{array}
\qquad\qquad
(D)\quad
\begin{array}{rrrrrrrrrrl}
\min & y_1 & + & y_2 & & & - & y_4 & + & 2y_5 & \\
 & & - & y_2 & - & y_3 & - & y_4 & + & y_5 & = 0 \\
 & -y_1 & - & y_2 & & & + & y_4 & & & = 1 \\
 & y_1 & , & y_2 & , & y_3 & , & y_4 & , & y_5 & \geq 0
\end{array}
$$

e la base $B^1 = \{1, 2\}$. Le corrispondenti matrice e soluzione primale di base sono

$$
A_{B^1} = \begin{bmatrix} 0 & -1 \\ -1 & -1 \end{bmatrix} \quad , \quad \bar{x} = A_{B^1}^{-1} b_{B^1} = \begin{bmatrix} 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \ .
$$

La soluzione $\bar{x}$ è ammissibile in quanto

$$
A_{N^1}\bar{x} = \begin{bmatrix} -1 & 0 \\ -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \leq b_{N^1} = \begin{bmatrix} 0 \\ -1 \\ 2 \end{bmatrix} \ .
$$

L'insieme degli indici dei vincoli attivi è $I = \{1, 2, 3, 4\}$, e quindi $\bar{x}$ è degenere. Verificare l'ottimalità di $\bar{x}$, o individuare una sua direzione ammissibile di crescita, corrisponde a verificare quale dei due sistemi ristretti

$$
(P_R)\quad \left\{
\begin{array}{rrrrr}
 & & \xi_2 & > & 0 \\
 & - & \xi_2 & \leq & 0 \\
-\xi_1 & - & \xi_2 & \leq & 0 \\
-\xi_1 & & & \leq & 0 \\
-\xi_1 & + & \xi_2 & \leq & 0
\end{array}
\right.
\qquad\qquad
(D_R)\quad \left\{
\begin{array}{rrrrrrrl}
 & - & y_2 & - & y_3 & - & y_4 & = 0 \\
-y_1 & - & y_2 & & & + & y_4 & = 1 \\
y_1 & , & y_2 & , & y_3 & , & y_4 & \geq 0
\end{array}
\right.
$$

ha soluzione. Poiché (D_R) è sottodeterminato (almeno per quanto riguarda le eguaglianze), una possibile strategia è quella di fissare a zero le variabili 3 e 4 (quelle fuori base) e risolvere il sistema ristretto a quelle in base; ciò corrisponde ad ignorare temporaneamente i vincoli attivi 3 e 4. In altri termini, la soluzione duale di base corrispondente a B^1 è

$$
y_{B^1}^1 = cA_{B^1}^{-1} = \begin{bmatrix} 0 \, , \, 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} -1 \, , \, 0 \end{bmatrix} \ .
$$

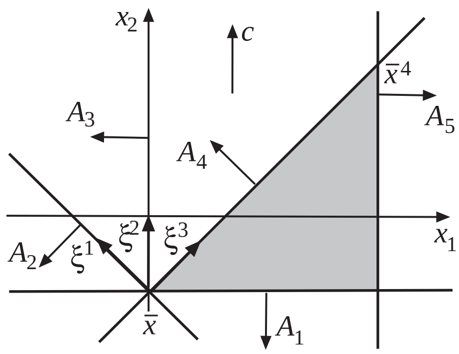
Poiché $y_1^1 < 0$, selezionando $h = 1$ ($B^1(h) = 1$) la direzione

$$
\xi^1 = -A_{B^1}^{-1} u_{B^1(h)} = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \ ,
$$

è di crescita per. Poiché però

$$
A_3\xi^1 = \begin{bmatrix} -1 \, , \, 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = 1 > 0
$$

$$
A_4\xi^1 = \begin{bmatrix} -1 \, , \, 1 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = 2 > 0
$$



questa direzione non è ammissibile per $\bar{x}$, come si può facilmente verificare dalla figura qui accanto. Quindi, la scelta (arbitraria) di fissare $y_3 = y_4 = 0$ in (D_R) non si è rivelata vincente: non si è ottenuta né una soluzione duale ammissibile, né una direzione ammissibile di crescita. Possiamo però effettuare un cambiamento di base degenere, scegliendo $k_1 = 3$. Per la nuova base $B^2 = B^1 \setminus \{h\} \cup \{k\} = \{3, 2\}$ risulta

$$
A_{B^2} = \begin{bmatrix} -1 & 0 \\ -1 & -1 \end{bmatrix} \quad , \quad A_{B^2}^{-1} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} \quad , \quad y_{B^2}^2 = \begin{bmatrix} 0 \, , \, 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 \, , \, -1 \end{bmatrix} \ ,
$$

da cui la scelta $h = 4$ ($B^2(h) = 2$) consente di individuare la direzione di crescita

$$
\xi^2 = -A_{B^2}^{-1} u_{B^2(h)} = \begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \ ,
$$

che però ancora non è ammissibile per $\bar{x}$ in quanto

$$
A_4\xi^2 = \begin{bmatrix} -1 \, , \, 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1 > 0 \ .
$$

Possiamo comunque effettuare un nuovo cambiamento di base degenere sostituendo $k = 4$ a h: otteniamo così la base $B^3 = \{3, 4\}$ per cui abbiamo

$$
A_{B^3} = \begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix} \quad , \quad A_{B^3}^{-1} = \begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix} \quad , \quad y_{B^3}^3 = \begin{bmatrix} 0 \, , \, 1 \end{bmatrix} \begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} -1 \, , \, 1 \end{bmatrix} \ .
$$

La scelta $h = 3$ ($B^3(h) = 1$) consente allora di individuare la direzione di crescita

$$\xi^3 = -A_{B^3}^{-1} u_{B^3(h)} = \begin{bmatrix} -1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \ ,$$

che è ammissibile per $\bar{x}$ in quanto

$$A_1\xi^3 = \begin{bmatrix} 0 \, , -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = -1 < 0 \quad , \quad A_2\xi^3 = \begin{bmatrix} -1 \, , -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = -2 < 0 \ .$$

Pertanto, ξ^3 risolve (P_R) e quindi $\bar{x}$ non è una soluzione ottima di (P). Infatti, il massimo spostamento consentito lungo ξ^3, calcolato mediante (2.24), questo direzione,

$$\bar{\lambda} = \lambda_5 = \frac{b_5 - A_5\bar{x}}{A_5\xi_3} = \frac{2-0}{1} = 2 \ ,$$

permette di individuare $k = 5$, la nuova base $B^4 = \{5, 4\}$ e la corrispondente soluzione di base

$$\bar{x}^4 = A_{B^4}^{-1} b_{B^4} = x(\bar{\lambda}) = \bar{x} + \bar{\lambda}\xi^3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix} + 2 \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \ ,$$

per cui la funzione obiettivo vale $c\bar{x}^4 = 1 > -1 = c\bar{x}$.

Possiamo ora fornire una descrizione formale di un algoritmo per la soluzione dei problemi di PL, che verrà chiamato *Simplesso Primale*.

```
procedure Simplesso_Primale( A , b , c , B , x̄ , ȳ , stato ) {
  for( stato = “” ;  ; ) {
    x̄ = A_B^{-1} b_B; ȳ = [ ȳ_B , ȳ_N ] = [ c A_B^{-1} , 0 ];
    if( ȳ_B ≥ 0 )   then { stato = “ottimo”; break; }
    h = min { i ∈ B : ȳ_i < 0 }; ξ = −A_B^{-1} u_{B(h)};
    if( A_N ξ ≤ 0 ) then { stato = “P.illimitato”; break; }
    λ̄ = min{ λ_i = ( b_i − A_i x̄ )/( A_i ξ ) : A_i ξ > 0 , i ∈ N };
    k = min{ i ∈ N : λ_i = λ̄ }; B = B ∪ {k} \ {h};
    }
}
```

Procedura 2.1: Simplesso Primale

L'algoritmo riceve in input una descrizione del problema ed un base B che si assume essere *primale ammissibile*, ed iterativamente esegue le seguenti operazioni:

1. Verifica l'ottimalità della soluzione primale di base $\bar{x}$, nel qual caso termina fornendo anche la corrispondente soluzione duale $\bar{y}$, oppure individua una direzione di crescita ξ.

2. Calcola il massimo passo di spostamento lungo la direzione ξ, che può essere nullo in caso di degenerazione primale: se il passo risulta essere $+\infty$, cioè se la direzione ξ consente una crescita illimitata della funzione obiettivo, allora l'algoritmo termina avendo provato che (P) è illimitato e quindi (D) è vuoto.

3. Viene aggiornata la base: $B' = B \cup \{k\} \setminus \{h\}$, con h e k selezionati come precedentemente definito; ciò corrisponde o ad un cambiamento di vertice, ed in questo caso il valore della funzione obiettivo cresce, oppure ad un cambiamento di base degenere.

L'algoritmo termina in un numero finito di passi fornendo una base ottima B e la corrispondente coppia di soluzioni ottima, se esiste, oppure fornendo la risposta che il problema è illimitato. Infatti, quando la direzione di crescita ξ è ammissibile, o si determina che il problema è illimitato (ed in tal caso l'algoritmo termina), oppure viene individuata una nuova soluzione di base avente un valore della funzione obiettivo maggiore di quello corrente. Essendo tale nuova soluzione un vertice, ed essendo il numero di vertici finito, la proprietà di generare una sequenza crescente di valori della funzione obiettivo di (P) garantisce che l'algoritmo esegua un numero finito di iterazioni in corrispondenza di vertici non degeneri. Quando invece l'algoritmo visita un vertice degenere, allora, come già evidenziato,

esso può generare direzioni non ammissibili e non spostarsi dal vertice corrente. L'algoritmo proposto garantisce però la terminazione finita anche in presenza di basi primali degeneri utilizzando uno specifico *criterio di selezione degli indici entrante ed uscente* noto come *regola anticiclo di Bland*: nel caso in cui esistano più indici h candidati ad uscire dalla base corrente B e/o più indici k candidati ad entrare in base, l'algoritmo seleziona sempre l'indice minimo, cioè $h = \min\{\ i \in B\ :\ \bar{y}_i < 0\ \}$ e $k = \min\{\ i \in N\ :\ \lambda_i = \bar{\lambda}\ \}$. Nel seguito (cf. Teorema 2.19) dimostreremo che questo criterio garantisce che una base non venga esaminata più di una volta e quindi, essendo il numero di basi finito, garantisce che venga eseguito un numero finito di iterazioni anche in corrispondenza dei vertici degeneri Osserviamo che l'ordine delle righe e delle colonne è arbitrario: basta fissare un qualsiasi ordinamento degli indici e applicare il criterio di Bland rispetto a tale ordinamento.
È utile discutere brevemente alcuni elementi dell'algoritmo:

- Per semplicità abbiamo assunto che una base primale ammissibile venga fornita in input, ma in generale la determinazione di una qualsiasi base primale ammissibile è non banale; per questo si può ricorrere ad una fase di inizializzazione che verrà descritta nel seguito.

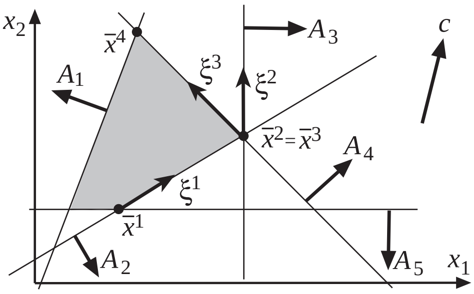
- Anche se non evidenziato nello pseudo-codice, nel caso in cui il problema è superiormente illimitato può essere utile fornire in output anche la direzione ξ determinata. Questa, insieme a $\bar{x}$ costituisce infatti un *certificato di illimitatezza* del problema: qualsiasi punto nella semiretta
$$\bar{x} + \lambda\xi \qquad\qquad \lambda \geq 0$$
è una soluzione ammissibile con un costo che cresce al crescere di λ. In effetti è possibile derivare da ξ anche un *certificato di inammissibilità* del problema duale. Questo si basa sul fatto che
$$c\xi > 0 \qquad \text{e} \qquad A\xi = \left[\begin{array}{c} A_B \\ A_N \end{array}\right]\xi = \left[\begin{array}{c} -u_{B(h)} \\ z_N \end{array}\right] \qquad \text{con} \qquad z_N = A_N\xi \leq 0 \ \ .$$
Qualsiasi soluzione ammissibile del duale dovrebbe avere $yA = c$ e $y \geq 0$: moltiplicando entrambe i lati della prima uguaglianza per ξ si ottiene un uguaglianza che deve essere verificata per ogni soluzione y, ossia
$$c\xi + y_h = z_N y_N \ \ ,$$
che però non può avere soluzioni in quanto il lato destro è ≤ 0 (tutte le y_i sono non negative, $z_N \leq 0$) mentre quello sinistro è positivo ($c\xi > 0$, $y_h > 0$). Quindi ξ fornisce un certificato "compatto" del fatto che i vincoli di non-negatività sulle y_N sono incompatibili con quello sulla y_h e con i vincoli strutturali $yA = c$.

- Ad ogni passo occorre calcolare l'inversa della matrice A_B, o equivalentemente fattorizzare A_B, e poi calcolare prodotti scalari o risolvere sistemi lineari per determinare $\bar{x}$, $\bar{y}_B$ e ξ: il primo passo ha costo computazionale $O(n^3)$ ed i successivi $O(n^2)$ per matrici dense non strutturate utilizzando metodi elementari. In pratica è però possibile determinare l'inversa di $A_{B'}$ ($B' = B \setminus \{h\} \cup \{k\}$) a partire dalla conoscenza di A_B^{-1}, o aggiornare le fattorizzazioni, con costo computazionale circa $O(n^2)$. Inoltre, la matrice A nelle applicazioni reali è di solito molto sparsa, e questo può essere sfruttato per ridurre ulteriormente il costo computazionale.

- Sperimentalmente si prova che la scelta di h suggerita dalla regola di Bland produce un algoritmo abbastanza inefficiente. Poiché nei problemi reali l'algoritmo, anche in assenza di regole anticiclo, cicla raramente, si preferisce la scelta di direzioni che garantiscano (sperimentalmente) una migliore efficienza della procedura, quali ad esempio selezionare l'indice h a cui corrisponde il valore $\bar{y}_i$ minore (negativo con valore assoluto più grande), il che determina la direzione che fornisce il massimo incremento unitario della funzione obiettivo (dato che $cx(\lambda) = c\bar{x} - \lambda\bar{y}_i$). Tecniche alternative al criterio di Bland per evitare cicli sono le cosiddette tecniche di perturbazione, in cui i dati vengono leggermente modificati in modo da rendere tutte le soluzioni di base non degeneri, senza perdere l'ottimalità delle basi ottime nel problema originario.

Esempio 2.28: Esecuzione dell'Algoritmo del Simplesso Primale

Consideriamo la coppia asimmetrica di problemi duali descritta nell'esempio 2.21 e applichiamo l'Algoritmo del Simplesso a partire dalla base primale ammissibile $B^1 = \{2,5\}$ già analizzata in quell'esempio, a cui corrisponde il vertice $\bar{x}^1 = [\,4\,,\,4\,]$. Come abbiamo già visto, la soluzione duale di base è $\bar{y}^1 = [\,0\,,\,1\,,\,0\,,\,0\,,\,-5\,]$: quindi l'indice uscente è $h = 5$, con $B^1(h) = 2$ e la direzione di crescita è



$$\xi^1 = -A_{B^1}^{-1} u_{B^1(h)} = \begin{bmatrix} -1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \; .$$

La ricerca dell'indice uscente è effettuata analizzando i vincoli non in base:

$$A_{N^1}\xi^1 = \begin{bmatrix} -2 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \end{bmatrix} = \begin{bmatrix} -3 \\ 2 \\ 3 \end{bmatrix} \; .$$

Essendo positive solo $A_3\xi^1$ ed $A_4\xi^1$, l'insieme $J^1 = \{\; i \in N^1 \;:\; A_i\xi^1 > 0 \;\}$ degli indici candidati ad entrare in base è $\{3,4\}$; infatti, dalla figura 2.10 si può osservare come, partendo da $\bar{x}^1$, e muovendosi nella direzione ξ^1 lungo l'iperpiano (retta) di supporto del secondo vincolo, si incontrano gli iperpiani (rette) di supporto de vincoli 3 e 4. Il valore negativo di $A_1\xi^1$ indica che ci si sta allontanando dall'iperpiano (retta) di supporto del vincolo 1.

Determiniamo ora il passo dello spostamento lungo ξ^1, dato da $\bar{\lambda} = \min\{\; \lambda_i \;:\; i \in J^1 \;\}$, dove λ_i è data da (2.23):

$$\lambda_3 = (14-8)/3 = 2 \qquad , \qquad \lambda_4 = (8-4)/2 = 2 \;\; ;$$

essendo $\bar{\lambda} = \lambda_3 = \lambda_4 = 2$, per la regola anticiclo di Bland si ha $k = \min\{3,4\} = 3$. La nuova base è perciò $B^2 = B^1 \setminus \{h\} \cup \{k\} = \{2,5\} \setminus \{5\} \cup \{3\} = \{2,3\}$.

Effettuiamo la seconda iterazione:

$$A_{B^2} = \begin{bmatrix} 1 & -2 \\ 1 & 0 \end{bmatrix} \quad , \quad A_{B^2}^{-1} = \begin{bmatrix} 0 & 1 \\ -1/2 & 1/2 \end{bmatrix} \quad , \quad b_{B^2} = \begin{bmatrix} -4 \\ 8 \end{bmatrix} \;\; ;$$

pertanto la nuova soluzione di base primale è

$$\bar{x}^2 = A_{B^2}^{-1} b_{B^2} = \begin{bmatrix} 0 & 1 \\ -1/2 & 1/2 \end{bmatrix} \begin{bmatrix} -4 \\ 8 \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \end{bmatrix}$$

mentre quella duale è

$$\bar{y}_{B^2} = cA_{B^2}^{-1} = [\; 1 \quad 3 \;] \begin{bmatrix} 0 & 1 \\ -1/2 & 1/2 \end{bmatrix} = [\; -3/2 \quad 5/2 \;] \quad , \quad \bar{y}_{N^2} = 0$$

e quindi $\bar{y}^2 = [\,0\,,\,-3/2\,,\,5/2\,,\,0\,,\,0\,]$. Essendoci solo $\bar{y}^2_2 < 0$, l'indice uscente è $h = 2$. Calcoliamo la direzione di crescita e il passo lungo essa:

$$\xi^2 = \begin{bmatrix} 0 & -1 \\ 1/2 & -1/2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1/2 \end{bmatrix} \quad , \quad A_{N^2}\xi^2 = \begin{bmatrix} -2 & 1 \\ 1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1/2 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/2 \\ -1/2 \end{bmatrix} \; ,$$

$$J^2 = \{1,4\} \quad , \quad \lambda_1 = 22 \quad , \quad \lambda_4 = 0 = \bar{\lambda} \quad , \quad k = 4 \; .$$

$\lambda_4 = 0$ è una conseguenza del fatto che $A_4\bar{x}^2 = 14 = b_4$, cioè che la soluzione di base è primale degenere, come avevamo già mostrato nell'esempio 2.22. Pertanto, $\bar{\lambda} = 0$ indica che la direzione di crescita ξ^2 non è una direzione ammissibile. La nuova base è $B^3 = B^2 \setminus \{2\} \cup \{4\} = \{3,4\}$ e la corrispondente soluzione primale di base è

$$A_{B^3} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad , \quad A_{B^3}^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \quad , \quad b_{B^3} = \begin{bmatrix} 8 \\ 14 \end{bmatrix} \quad , \quad \bar{x}^3 = \begin{bmatrix} 8 \\ 6 \end{bmatrix} \; .$$

Abbiamo effettuato un cambio di base degenere, in quanto la soluzione primale individuata coincide con la precedente; la nuova soluzione di base duale, come visto nell'esempio 2.22, è diversa dalla precedente, infatti:

$$\bar{y}_{B^3} = [\; 1 \quad 3 \;] \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} = [\; -2 \quad 3 \;] \quad , \quad \bar{y}_{N^3} = 0 \quad , \quad \bar{y}^3 = [\; 0 \quad 0 \quad -2 \quad 3 \quad 0 \;] \; .$$

L'indice uscente è $h = 3$, e

$$\xi^3 = \begin{bmatrix} -1 & 0 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad , \quad A_{N^3}\xi^3 = \begin{bmatrix} -2 & 1 \\ 1 & -2 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 \\ -3 \\ -1 \end{bmatrix} \;\; ;$$

si ha pertanto $J^3 = \{1\}$ e $\bar{\lambda} = \lambda_1 = 11/3$, l'indice entrante è $k = 1$ e la nuova base è $B^4 = B^3 \setminus \{3\} \cup \{1\} = \{1, 4\}$. Poiché

$$A_{B^4} = \begin{bmatrix} -2 & 1 \\ 1 & 1 \end{bmatrix} \quad , \quad A_{B^4}^{-1} = \begin{bmatrix} -1/3 & 1/3 \\ 1/3 & 2/3 \end{bmatrix} \quad , \quad b_{B^4} = \begin{bmatrix} 1 \\ 14 \end{bmatrix} \quad , \quad \bar{x}^4 = \begin{bmatrix} 13/3 \\ 29/3 \end{bmatrix}$$

$$\bar{y}_{B^4} = \begin{bmatrix} 2/3 & 7/3 \end{bmatrix} \quad , \quad \bar{y}_{N^4} = 0 \quad , \quad \bar{y}^4 = \begin{bmatrix} 2/3 & 0 & 0 & 7/3 & 0 \end{bmatrix} \geq 0$$

la soluzione duale è ammissibile e quindi l'algoritmo termina: $\bar{x}^4$, $\bar{y}^4$ costituiscono la coppia di soluzioni ottime associate alla base ottima $B = \{1, 4\}$. La sequenza di soluzioni primali e di direzioni di crescita è mostrata nella figura in alto.

Esempio 2.29: Interpretazione geometrica del Simplesso Primale
Consideriamo l'esempio di figura 2.11(a) e la base primale ammissibile $B = \{1, 2\}$. La soluzione duale corrispondente non è ammissibile poiché risulta $y_1 < 0$: esce quindi dalla base il vincolo $h = 1$. Entra poi in base il vincolo 3: infatti, le scelte possibili per k sono 3 e 4 (poiché $A_5\xi^1 = 0$), ma k è scelto in modo da determinare una base adiacente ammissibile e $k = 4$ non determina una base primale ammissibile. La nuova base è $B = \{3, 2\}$: la soluzione duale corrispondente non è ammissibile poiché risulta $y_2 < 0$ (si veda la figura 2.11(b)). Esce quindi dalla base il vincolo 2, ci si muove lungo la direzione ξ^2 ortogonale ad A_3, ed entra in base il vincolo 4 (le alternative per k essendo 4 e 5); la nuova base $\{3, 4\}$ è ottima in quanto primale e duale ammissibile.

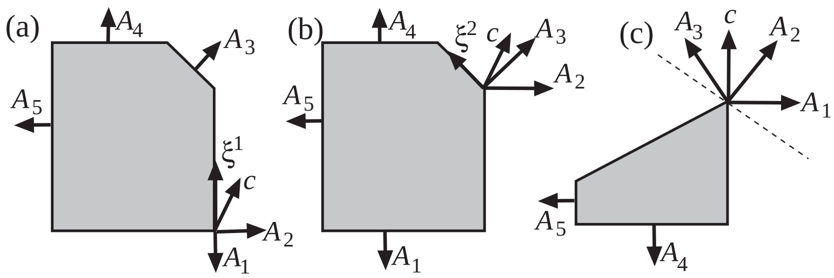


Figura 2.11: Interpretazione geometrica del Simplesso Primale

Il caso di una soluzione degenere è illustrato in figura 2.11(c), ove si consideri la base $B = \{1, 2\}$: tale base non è duale ammissibile ($y_1 < 0$), esce quindi di base il vincolo 1 ed entra in base il vincolo 3 (in questo caso, $\bar{\lambda} = 0$). La nuova base $\{3, 2\}$ è ottima: in questo caso l'iterazione dell'algoritmo modifica la matrice di base senza modificare la soluzione di base *primale* (il vertice non cambia).

L'algoritmo *Simplesso Primale* non può mai determinare che il problema (P) sia vuoto, in quanto abbiamo supposto di avere in input una soluzione di base ammissibile. In generale, mentre determinare una base è semplice, non è altrettanto semplice determinare una base che sia anche ammissibile. Se pertanto non si dispone di una base ammissibile di partenza, si ricorre ad un *problema ausiliario*, presentato nel paragrafo successivo.

Esercizio 2.12 *Determinare la complessità delle singole istruzioni dell'algoritmo Simplesso_Primale.*

Esercizio 2.13 *Determinare l'occupazione di memoria dell'algoritmo Simplesso_Primale.*

Individuazione di una base primale ammissibile.

La procedura *Simplesso_Primale* non può mai determinare che il problema (P) sia vuoto, in quanto abbiamo supposto di avere in input una soluzione di base primale ammissibile. In generale, mentre determinare una qualsiasi base è semplice, non è altrettanto semplice determinarne una che sia anche primale ammissibile. Per questo si può però utilizzare la stessa procedura *Simplesso_Primale*, applicandola ad un *problema ausiliario* opportunamente costruito.
Possiamo innanzitutto assumere di avere a disposizione una base B; questa esiste per l'assunzione che $rango(A) = n$, e può essere facilmente determinata.

Esercizio 2.14 *Proporre un algoritmo per determinare una base B.*

Siano quindi date B e la corrispondente soluzione di base $\bar{x} = A_B^{-1} b_B$. Siano poi $H = \{ \ i \ : \ A_i\bar{x} \leq b_i \ \}$ e $J = \{ \ i \ : \ A_i\bar{x} > b_i \ \}$ rispettivamente l'insieme degli indici dei vincoli rispettati e di quelli violati da $\bar{x}$: è ovvio che $B \subseteq H$ mentre $J \cap B = \emptyset$. Chiaramente, se $J = \emptyset$ allora $\bar{x}$ è una soluzione ammissibile di base per (P), altrimenti si può costruire il problema ausiliario

$$
(PA) \qquad \begin{array}{rrcrl}
\max & & - & u\nu & \\
 & A_H x & & & \leq b_H \\
 & A_J x & - & \nu & \leq b_J \\
 & & - & \nu & \leq 0
\end{array}
$$

dove u è il vettore che ha tutte le componenti uguali ad 1. Tale problema ha $x = \bar{x}$, $\bar{\nu} = A_j\bar{x} - b_j$ come soluzione ammissibile di base; infatti, il problema ha $n + |J|$ variabili ed il vettore $[\ \bar{x}\ ,\ \bar{\nu}\ ]$ è soluzione del sistema di $n + |J|$ equazioni in $n + |J|$ incognite

$$
\left[\begin{array}{cc} A_B & 0 \\ A_J & -I \end{array} \right] \left[\begin{array}{c} x \\ \nu \end{array} \right] = \left[\begin{array}{c} b_B \\ b_J \end{array} \right] \qquad \text{dove} \qquad \left[\begin{array}{cc} A_B & 0 \\ A_J & -I \end{array} \right]
$$

è la *matrice di base ampliata*, che è non singolare in quanto A_B è non singolare. Quindi, la procedura *Simplesso_Primale* può essere utilizzata per determinare una soluzione ottima di base $[\ x^*\ ,\ \nu^*\ ]$ di (PA); si osservi che il problema non può essere illimitato, poiché $-u\nu \leq 0$ per ogni $\nu \geq 0$. Si hanno due casi possibili:

- $\nu^* \neq 0$: in questo caso (P) è vuoto, infatti ad una qualsiasi soluzione ammissibile x di (P) corrisponderebbe la soluzione $[x\ ,\ 0\ ]$ di (PA) con valore nullo della funzione obiettivo, e di conseguenza migliore di $[\ x^*\ ,\ \nu^*\ ]$ che ha un valore negativo della funzione obiettivo;

- $\nu^* = 0$: in questo caso x^* è una soluzione di base ammissibile per (P).

Esempio 2.30: Individuazione di una base primale ammissibile
Si consideri il seguente problema di PL

$$
(P) \qquad \begin{array}{rrcrcl}
\max & x_1 & + & 3x_2 & & \\
 & x_1 & + & 2x_2 & \leq & 5 \\
 & x_1 & - & x_2 & \leq & 2 \\
 & x_1 & + & 3x_2 & \leq & 7 \\
 & & - & x_2 & \leq & 0
\end{array}
$$

e la base $B = \{1, 4\}$ non primale ammissibile, in quanto

$$
A_B = A_B^{-1} = \left[\begin{array}{rr} 1 & 2 \\ 0 & -1 \end{array} \right] \quad , \quad \bar{x} = A_B^{-1} b_B = \left[\begin{array}{rr} 1 & 2 \\ 0 & -1 \end{array} \right] \left[\begin{array}{c} 5 \\ 0 \end{array} \right] = \left[\begin{array}{c} 5 \\ 0 \end{array} \right]
$$

$$
A_N\bar{x} = \left[\begin{array}{rr} 1 & -1 \\ 1 & 3 \end{array} \right] \left[\begin{array}{c} 5 \\ 0 \end{array} \right] = \left[\begin{array}{c} 5 \\ 5 \end{array} \right] \not\leq \left[\begin{array}{c} 2 \\ 7 \end{array} \right] = b_N \ .
$$

A partire da B possiamo trovare, se esiste, una base primale ammissibile risolvendo il problema primale ausiliario (PA) ed il suo duale (DA); poiché $J = \{2\}$ e $H = \{3\}$, essi sono

$$
(PA) \quad \begin{array}{rrcrcrl}
\max & & & & - & \nu & \\
 & x_1 & + & 2x_2 & & & \leq 5 \\
 & x_1 & - & x_2 & - & \nu & \leq 2 \\
 & x_1 & + & 3x_2 & & & \leq 7 \\
 & & - & x_2 & & & \leq 0 \\
 & & & & - & \nu & \leq 0
\end{array}
\qquad
(DA) \quad \begin{array}{rrcrcrcrcrl}
\min & 5y_1 & + & 2y_2 & + & 7y_3 & & & & & \\
 & y_1 & + & y_2 & + & y_3 & & & & & = 0 \\
 & 2y_1 & - & y_2 & + & 3y_3 & - & y_4 & & & = 0 \\
 & & & y_2 & & & & & + & y_5 & = 1 \\
 & y_1 & , & y_2 & , & y_3 & , & y_4 & , & y_5 & \geq 0
\end{array}
$$

La base $\mathcal{B}^1 = B \cup J = \{1, 2, 4\}$ è primale ammissibile per costruzione, possiamo quindi applicare l'Algoritmo del Simplesso Primale a partire da essa.

it.1) $\mathcal{B}^1 = \{1, 2, 4\}, \mathcal{N}^1 = \{3, 5\}$:

$$
\mathcal{A}_{\mathcal{B}^1} = \left[\begin{array}{rrr} 1 & 2 & 0 \\ 1 & -1 & -1 \\ 0 & -1 & 0 \end{array} \right] \quad , \quad \mathcal{A}_{\mathcal{B}^1}^{-1} = \left[\begin{array}{rrr} 1 & 0 & 2 \\ 0 & 0 & -1 \\ 1 & -1 & 3 \end{array} \right] \quad , \quad \left[\begin{array}{c} \bar{x}^1 \\ \bar{\nu}^1 \end{array} \right] = \left[\begin{array}{rrr} 1 & 0 & 2 \\ 0 & 0 & -1 \\ 1 & -1 & 3 \end{array} \right] \left[\begin{array}{c} 5 \\ 2 \\ 0 \end{array} \right] = \left[\begin{array}{c} 5 \\ 0 \\ 3 \end{array} \right]
$$

$$\mathcal{A}_{\mathcal{N}^1}\begin{bmatrix}\bar{x}^1\\ \bar{\nu}^1\end{bmatrix}=\begin{bmatrix}1 & 3 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}5\\ 0\\ 3\end{bmatrix}=\begin{bmatrix}5\\ -3\end{bmatrix}\leq\begin{bmatrix}7\\ 0\end{bmatrix}=b_{\mathcal{N}^1}$$

$$\bar{y}_{\mathcal{B}^1}=\begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}1 & 0 & 2\\ 0 & 0 & -1\\ 1 & -1 & 3\end{bmatrix}=\begin{bmatrix}-1 & 1 & -3\end{bmatrix}\quad,\quad\begin{bmatrix}-1 & 1 & 0 & -3 & 0\end{bmatrix}\not\geq 0\ .$$

Pertanto, per la regola anticiclo di Bland $h=1, \mathcal{B}^1(h)=1$. Per individuare l'indice entrante calcoliamo

$$\xi^1=\begin{bmatrix}-1\\ 0\\ -1\end{bmatrix}\quad,\quad\mathcal{A}_{\mathcal{N}^1}\xi^1=\begin{bmatrix}1 & 3 & 0\\ 0 & 0 & -1\end{bmatrix}\begin{bmatrix}-1\\ 0\\ -1\end{bmatrix}=\begin{bmatrix}-1\\ 1\end{bmatrix}\quad,\quad\bar{\lambda}=\lambda_5=3\quad,\quad k=5\ .$$

it.2) $\mathcal{B}^2=\{5,2,4\}, \mathcal{N}^2=\{1,3\}$:

$$\mathcal{A}_{\mathcal{B}^2}=\begin{bmatrix}0 & 0 & -1\\ 1 & -1 & -1\\ 0 & -1 & 0\end{bmatrix}\quad,\quad\mathcal{A}_{\mathcal{B}^2}^{-1}=\begin{bmatrix}-1 & 1 & -1\\ 0 & 0 & -1\\ -1 & 0 & 0\end{bmatrix}\quad,\quad\begin{bmatrix}\bar{x}^2\\ \bar{\nu}^2\end{bmatrix}=\begin{bmatrix}-1 & 1 & -1\\ 0 & 0 & -1\\ -1 & 0 & 0\end{bmatrix}\begin{bmatrix}0\\ 2\\ 0\end{bmatrix}=\begin{bmatrix}2\\ 0\\ 0\end{bmatrix}$$

$$\mathcal{A}_{\mathcal{N}^2}\begin{bmatrix}\bar{x}^2\\ \bar{\nu}^2\end{bmatrix}=\begin{bmatrix}1 & 2 & 0\\ 1 & 3 & 0\end{bmatrix}\begin{bmatrix}2\\ 0\\ 0\end{bmatrix}=\begin{bmatrix}2\\ 2\end{bmatrix}\leq\begin{bmatrix}5\\ 7\end{bmatrix}=b_{\mathcal{N}^2}$$

$$\bar{y}_{\mathcal{B}^2}=\begin{bmatrix}0 & 0 & -1\end{bmatrix}\begin{bmatrix}-1 & 1 & -1\\ 0 & 0 & -1\\ -1 & 0 & 0\end{bmatrix}=\begin{bmatrix}1 & 0 & 0\end{bmatrix}\quad,\quad\begin{bmatrix}0 & 0 & 0 & 0 & 1\end{bmatrix}\geq 0\ .$$

Poiché $\bar{y}^2$ è ammissibile per (DA), l'algoritmo termina con le soluzioni ottime $[\bar{x}^2, \bar{\nu}^2]$ e $\bar{y}^2$. Si osservi che $\bar{\nu}^2=0$, e quindi $\bar{x}^2$ è una soluzione di base primale ammissibile per (P), avente $B=\mathcal{B}^2\setminus\{5\}=\{2,4\}$ come base; infatti:

$$A_B=A_B^{-1}=\begin{bmatrix}1 & -1\\ 0 & -1\end{bmatrix}\quad,\quad\bar{x}=A_B^{-1}b_B=\begin{bmatrix}1 & -1\\ 0 & -1\end{bmatrix}\begin{bmatrix}2\\ 0\end{bmatrix}=\begin{bmatrix}2\\ 0\end{bmatrix}=\bar{x}^2$$

da cui

$$A_N\bar{x}=\begin{bmatrix}1 & 2\\ 1 & 3\end{bmatrix}\begin{bmatrix}2\\ 0\end{bmatrix}=\begin{bmatrix}2\\ 2\end{bmatrix}\leq\begin{bmatrix}5\\ 7\end{bmatrix}=b_N.$$

Esercizio 2.15 *Risolvere il problema (P) dell'esempio precedente utilizzando l'Algoritmo del Simplesso Primale a partire dalla base $B=\{2,4\}$.*

La regola anticiclo di Bland.

Riportiamo adesso una dimostrazione formale del fatto che la regola anticiclo di Bland garantisce che una base non possa venir esaminata più di una volta, e quindi la terminazione finita dell'Algoritmo del Simplesso.

Teorema 2.19 *La procedura Simplesso_Primale, che implementa la regola anticiclo di Bland, non esamina la stessa base più di una volta.*

Dimostrazione Supponiamo per assurdo che una base B venga visitata due o più volte. Chiamando $B(i)$, $h(i)$ e $k(i)$ rispettivamente la base, l'indice entrante e l'indice uscente all'i-esima iterazione dell'algoritmo, questa ipotesi afferma che esistono due iterazioni v ed ℓ, con $v<\ell$, per cui $B(v)=B(\ell)=B$ e $B(i)\neq B$ per ogni i tale che $v<i<\ell$. Poiché ogni volta che l'algoritmo visita un nuovo vertice il valore della funzione obiettivo cresce, tutte le basi visitate nelle iterazioni del ciclo corrispondono allo stesso vertice $\bar{x}$, e quindi tutti gli indici che entrano ed escono di base appartengono a $I(\bar{x})$. Definiamo

$$r:=\max\{\ h(i)\ :\ v\leq i\leq\ell\ \}\ =\ \max\{\ k(i)\ :\ v\leq i\leq\ell\ \}\ ,$$

ossia r è il massimo degli indici entranti ed uscenti dalla base in tutte le iterazioni comprese tra v ed ℓ, estremi inclusi. Il fatto che il massimo degli indici entranti sia uguale al massimo degli indici uscenti segue immediatamente dal fatto che $B(v)=B(\ell)$: qualsiasi indice che entri in base dopo l'iterazione v deve uscirne prima dell'iterazione l, ed analogamente per gli indici entranti.

Sia adesso p una qualsiasi iterazione $v\leq p\leq\ell$ in cui r è l'indice uscente, ossia $r=h(p)$, e sia q una qualsiasi iterazione $v\leq q\leq\ell$ in cui r è l'indice entrante, ossia $r=k(p)$. Sia adesso $\bar{y}=[\bar{y}_{B(p)},0]$ la soluzione duale all'iterazione p, in cui r esce di base, e sia $\xi=-A_{B(q)}^{-1}u_{B(h(q))}$ la direzione all'iterazione q, in cui r entra. Per costruzione vale $c\xi>0$ e $\bar{y}_{B(p)}A_{B(p)}=c$, e quindi

$$c\xi=\bar{y}_{B(p)}A_{B(p)}\xi=\textstyle\sum_{i\in B(p)}\bar{y}_iA_i\xi>0\ \ .$$

Da questo possiamo ottenere una contraddizione mostrando che tutti i termini nella sommatoria della relazione precedente sono minori o uguali a zero. Dobbiamo quindi distinguere tre casi $i=r$, $i>r$ ed $i<r$.

$i = r$ Poiché r è l'indice uscente all'iterazione p si ha $\bar{y}_r < 0$; dal fatto che r è l'indice entrante all'iterazione q si ha invece che $A_r\xi > 0$, da cui $\bar{y}_r A_r \xi < 0$.

$i > r$ Per definizione di r, tutti gli indici $i > r$ che stanno in $B(v)$ (se esistono) appartengono a *tutte* le basi in tutte le iterazioni comprese tra v ed ℓ; quindi, in particolare appartengono sia a $B(p)$ che a $B(q)$. Per definizione, nessuno di questi indici può essere $h(q)$, l'indice uscente all'iterazione q: di conseguenza si ha per ogni indice $i > r$ in $B(p)$ (se ne esistono) che $i \in B(q)$ e $i \neq h(q)$. Per costruzione si ha allora che $A_i\xi = 0$, e quindi $\bar{y}_i A_i \xi = 0$ per ogni $i > r$.

$i < r$ Dalla definizione di r segue immediatamente che

$$r = h(p) = \min\{\, j \in B(p) \ : \ \bar{y}_j < 0 \,\} = k(q) = \min\{\, j \in I(\bar{x}) \setminus B(q) \ : \ A_j\xi > 0 \,\} \ .$$

Dalla prima relazione discende quindi che $\bar{y}_i \geq 0$ per ogni $i < r$. Siccome qualsiasi indice i in $B(p)$ è l'indice di un vincolo attivo in $\bar{x}$, allora si ha certamente $A_i\xi \leq 0$ per $i < r$: infatti, se $i \in B(q)$ si ha $A_i\xi \leq 0$ per costruzione, mentre per $i \notin B(q)$ si ha che r è il minimo indice j per cui $A_j\xi > 0$, e quindi $A_i\xi \leq 0$ per $i < r$. Da tutto questo segue che $\bar{y}_i A_i \xi \leq 0$ per $i < r$.

Abbiamo quindi ottenuto una contraddizione, il che dimostra che l'algoritmo visita ciascuna base al più una volta. $\diamond$

Oltre a mostrare formalmente la correttezza dell'Algoritmo del Simplesso Primale, il Teorema 2.19 può essere sfruttato per dimostrare un certo numero di risultati teorici.

Esercizio 2.16 *Dati $c \in \mathbb{R}^n$ e $A \in \mathbb{R}^{m \times n}$ con rango$(A) = n$, il che implica $m \geq n$, si dimostri il Lemma di Farkas utilizzando il Teorema 2.19 (suggerimento: si applichi l'algoritmo del Simplesso Primale alle coppia di problemi*

$$(P) \quad \max\{\, cx \ : \ Ax \leq 0 \,\} \qquad\qquad (D) \quad \min\{\, y0 \ : \ yA = c\ ,\ y \geq 0 \,\}$$

notando che (P) non è vuoto, e che se esiste una soluzione $\bar{x}$ tale che $c\bar{x} > 0$ allora (P) è illimitato).

Esercizio 2.17 *Dati $c \in \mathbb{R}^n$ e $A \in \mathbb{R}^{m \times n}$ con rango$(A) < n$, il che accade sicuramente ad esempio se $m < n$, si dimostri il Lemma di Farkas utilizzando il Teorema 2.19 (suggerimento: si sfrutti l'esercizio precedente ed il Teorema 2.5).*

Esercizio 2.18 *Si dimostrino i Teoremi 2.17 e 2.18.*

2.3.2 L'algoritmo del Simplesso Duale

L'algoritmo del Simplesso Primale, descritto nel precedente paragrafo, genera una sequenza di coppie di soluzioni di base primali ammissibili e duali non ammissibili (tranne al più l'ultima): se la soluzione di base duale risulta ammissibile allora l'algoritmo termina, avendo determinato un vertice ottimo. Risulta naturale pensare ad un algoritmo che generi coppie di soluzioni di base complementari, in cui la soluzione duale sia sempre ammissibile, il quale cerchi invece di raggiungere l'ammissibilità primale. Un algoritmo di questo tipo è l'algoritmo *Simplesso Duale*, presentato nel seguito.

Il Simplesso Duale altro non è che il Simplesso Primale applicato al problema duale della coppia asimmetrica, riscritto in forma primale; esaminando nel dettaglio le operazioni compiute dall'algoritmo se ne ottiene poi una versione semplificata in cui si fa riferimento alle stesse entità (basi e soluzioni di base) che appaiono nel Simplesso Primale. Pur senza entrare nei dettagli, mostriamo nel seguito i principali passaggi che portano alla definizione dell'algoritmo. Il duale della coppia asimmetrica, scritto in forma primale diviene

$$(D') \quad \max\{\, c'y \ : \ A'y \leq b' \,\} \qquad \text{dove} \qquad c' = -b^T \quad , \quad A' = \begin{bmatrix} A^T \\ -A^T \\ -I \end{bmatrix} \quad , \quad b' = \begin{bmatrix} c \\ -c \\ 0 \end{bmatrix} \ .$$

Poiché per ogni $\bar{y}$ ammissibile deve risultare $\bar{y}A^i = c_i$ per ogni $i = 1, \ldots, n$, almeno uno dei due vincoli $A^iy \leq c_i$ e $-A^iy \leq -c_i$ deve appartenere a qualsiasi base "primale" ammissibile B' per il problema; dato che i due vincoli sono linearmente dipendenti, al più uno di essi potrà appartenere ad una base. Quindi, degli m vincoli in una base "primale" ammissibile per (D'), n devono essere scelti in modo da contenere l'intera matrice A^T (eventualmente con un sottoinsieme delle righe cambiate di segno) mentre gli altri $m-n$ devono essere scelti tra gli m vincoli $-Iy \leq 0$. Detto $N \subset \{1,\ldots,m\}$ con $|N| = m-n$ l'insieme degli indici che caratterizzano quest'ultimo blocco di vincoli e $B = \{1,\ldots,m\} \setminus N$ il suo complementare, ne segue che (modulo un riordinamento di righe e colonne) si può sempre assumere che la matrice di base $A'_{B'}$ corrispondente ad una base B' "primale" ammissibile per (D') ed il corrispondente vettore $b'_{B'}$ abbiano la forma

$$A'_{B'} = \begin{bmatrix} \pm A_B^T & \pm A_N^T \\ 0 & -I_N \end{bmatrix} \qquad b'_{B'} = \begin{bmatrix} \pm c \\ 0 \end{bmatrix}$$

dove "$\pm$" indica che alcune delle righe potrebbero essere l'opposto della corrispondente colonna di A (e quindi anche il corrispondente elemento di $b'_{B'}$ è l'opposto del coefficiente dei costi). È immediato verificare che $A'_{B'}$ è invertibile se e solo se A_B è invertibile, e che la corrispondente soluzione "primale" di base $\bar{y}$ è esattamente $[\ \bar{y}_B\ ,\ \bar{y}_N\ ] = [\ cA_B^{-1}\ ,\ 0\ ]$.

Esercizio 2.19 *Dimostrare le affermazioni precedenti.*

Si noti che la conoscenza della "struttura" della matrice di base permette di calcolarla invertendo solamente A_B, una matrice $n \times n$, invece che l'intera $A'_{B'}$ che è $m \times m$; ciò permette un notevole incremento delle prestazioni nel caso in cui $m \gg n$. Analoghe considerazioni possono essere fatte per gli altri passi dell'algoritmo del Simplesso Primale, che si semplificano e trasformano quando l'algoritmo sia applicato a (D'). Ad esempio, gli unici indici che possono realmente uscire dalla base B' sono quelli relativi ai vincoli $-Iy \leq 0$, ossia quelli in N; se esce il vincolo $A^i y \leq c_i$ deve necessariamente entrare $-A^i y \leq -c_i$, e viceversa, il che corrisponde ad una particolare iterazione degenere che può sempre essere trattata a parte. Pertanto, l'indice uscente da B' esce da N, e quindi entra in B. Infatti, l'algoritmo che presenteremo seleziona prima l'indice entrante in B e poi quello uscente, (apparentemente) al contrario di ciò che fa il Simplesso Primale.

Esercizio 2.20 *Si continui l'analisi dell'applicazione del Simplesso Primale a (D') scrivendo il duale (P'), calcolando la soluzione "duale" di base $\bar{x}'$ corrispondente ad una base B', verificando cosa significhi che una variabile duale sia negativa, calcolando la corrispondente direzione di crescita ξ ed il massimo passo lungo di essa.*

Forniamo adesso una descrizione formale dell'algoritmo del *Simplesso Duale.*

```
procedure Simplesso_Duale( A , b , c , B , x̄ , ȳ , stato ) {
  for( stato = "" ; ; ) {
    x̄ = A_B^{-1} b_B; ȳ = [ ȳ_B , ȳ_N ] = [ c A_B^{-1} , 0 ];
    if( A_N x̄ ≤ b_N ) then { stato = "ottimo"; break; }
    k = min { i ∈ N : A_i x̄ > b_i }; η_B = A_k A_B^{-1};
    if( η_B ≤ 0 )     then { stato = "P.vuoto"; break; }
    θ̄ = min { θ_i = ȳ_i/η_i : η_i > 0 , i ∈ B };
    h = min{ i ∈ B : θ_i = θ̄ }; B = B ∪ {k} \ {h};
    }
}
```

Procedura 2.2: Simplesso Duale

L'algoritmo del Simplesso Duale riceve in input una base duale ammissibile B, e calcola la corrispondente coppia di soluzioni di base $\bar{x}$ e $\bar{y}$ (con $\bar{y} \geq 0$). Se anche $\bar{x}$ è ammissibile, cioè se $A_N\bar{x} \leq b_N$, l'algoritmo termina, avendo individuato una coppia di soluzioni ottime. Altrimenti, cioè se esiste un indice $k \in N$ tale che $A_k\bar{x} > b_k$, l'algoritmo determina una direzione d di decrescita per $\bar{y}$, in taluni casi ammissibile, definita nel seguente modo:

$$d_i = \begin{cases} -\eta_i & \text{se } i \in B \\ 1 & \text{se } i = k \\ 0 & \text{altrimenti} \end{cases}$$

dove $\eta_B = A_k A_B^{-1}$. Per verificare che d sia effettivamente una direzione di decrescita, consideriamo la soluzione duale parametrica $y(\theta) = \bar{y} + \theta d$ che si ottiene spostandosi da $\bar{y}$ lungo d di un passo $\theta \geq 0$. d è una direzione di decrescita, infatti per ogni $\theta > 0$ si ha

$$y(\theta)b = (\bar{y}_B - \theta\eta_B)b_B + \theta b_k = \bar{y}_B b_B + \theta(b_k - A_k A_B^{-1} b_B) = \bar{y}b + \theta(b_k - A_k\bar{x}) < \bar{y}b$$

essendo $A_k\bar{x} > b_k$. Per quanto riguarda l'ammissibilità, è facile verificare che $y(\theta)A = c$ per qualsiasi scelta di θ, in quanto

$$y(\theta)A = (\bar{y}_B - \theta\eta_B)A_B + \theta A_k = \bar{y}_B A_B + \theta(A_k - A_k) = \bar{y}_B A_B = c \ .$$

Per garantire che d sia ammissibile, va quindi verificato che sia $y(\theta) = \bar{y} + \theta d \geq 0$ per un opportuno passo di spostamento $\theta > 0$. Osserviamo che gli indici critici sono quelli in B, dovendo essere $(\bar{y}_B - \theta\eta_B) \geq 0$ (infatti, $\bar{y}_k + \theta \geq 0$ per ogni $\theta \geq 0$). In particolare, se $\eta_B \leq 0$, allora l'ammissibilità di $y(\theta)$ è assicurata per qualsiasi $\theta > 0$; essendo d una direzione di decrescita, segue che (D) è illimitato, e conseguentemente (P) è vuoto. Se invece esiste almeno un indice $i \in B$ per cui $\eta_i > 0$, il massimo passo $\bar{\theta}$ che può essere compiuto lungo d senza perdere l'ammissibilità duale è

$$\bar{\theta} = \min \{ \ \bar{y}_i/\eta_i \ : \ i \in B \ , \ \eta_i > 0 \ \}.$$

Se risulta $\bar{\theta} > 0$ allora d è effettivamente una direzione di decrescita, altrimenti siamo in un caso, del tutto analogo a quello visto nel primale, di cambiamento di base degenere: infatti ciò può accadere solamente se esiste un indice $i \in B$ tale che $\bar{y}_i = 0$, ossia se B è duale degenere. In ogni caso l'algoritmo fa uscire da B un indice h che determina il passo $\bar{\theta}$ lungo d, ossia l'indice di una componente di $y(\theta)$ che diverrebbe negativa se fosse effettuato un passo più lungo di $\bar{\theta}$; se $\bar{\theta} > 0$ la nuova base individua un diverso vertice del poliedro delle soluzioni duali, altrimenti la soluzione duale di base non cambia. Analogamente a quanto visto per l'algoritmo del Simplesso Primale, per evitare cicli si può applicare la regola anticiclo di Bland; nel caso dell'algoritmo del Simplesso Duale, tale regola si traduce nella selezione del minimo tra gli indici $k \in N$ tali che $A_k\bar{x} > b_k$ e del minimo tra gli indici h che determina il passo di spostamento $\bar{\theta}$ lungo d.
Si noti che, analogamente al caso primale, se il Simplesso Duale termina dichiarando che il duale è inferiormente illimitato allora ha fornito un *certificato di inammissibilità* del primale, sotto forma del vettore $\eta_B \leq 0$. Infatti, per la definizione di $\bar{x}$ ed η_B

$$\eta_B b_B = A_k A_B^{-1} b_B = A_k\bar{x} > b_k \ .$$

Qualsiasi soluzione ammissibile x del primale deve soddisfare in particolare $A_B x \leq b_B$; moltiplicando entrambi i lati della disequazione per $\eta_B \leq 0$ e ricordando la definizione di η_B si ottiene

$$(\,\eta_B A_B x = A_k x\,) \geq \eta_B b_B \qquad \text{che è incompatibile con} \qquad A_k x \leq b_k$$

in quanto $\eta_B b_B > b_k$. In altri termini, il sistema di disequazioni $A_B x \leq b_B$ è *incompatibile* con $A_k x \leq b_k$: non può esistere nessun x che soddisfa contemporaneamente tutti quei vincoli. L'insieme $B \cup \{k\}$ caratterizza quindi un *insieme inconsistente* (IS), dal quale si può poi cercare di determinare un insieme inconsistente *irriducibile* (IIS), ossia tale che la rimozione di qualsiasi vincolo lo rende consistente. Si noti che $B \cup \{k\}$ può non essere irriducibile; ciò ad esempio accade, come è facile verificare, se $\eta_i = 0$ per qualche $i \in B$. Gli IIS sono utili ad esempio per "debuggare" i modelli: se ci si aspetta che il modello abbia soluzione, perché la realtà modellata notoriamente ne ha, e questo non risulta vero, l'IIS indica "di quali vincoli è la colpa" e quindi può aiutare a determinare dove sia il problema. Gli (I)IS possono comunque avere anche altri usi.

Esempio 2.31: Applicazione del Simplesso Duale per via algebrica
Si consideri la coppia asimmetrica di problemi duali

$$(P)\quad \begin{array}{rrcrl} \max & x_1 & & & \\ & x_1 & + & 2x_2 & \leq 6 \\ & x_1 & - & 2x_2 & \leq 6 \\ & 2x_1 & + & x_2 & \leq 4 \\ & 2x_1 & - & x_2 & \leq 4 \\ & -x_1 & & & \leq 0 \end{array} \qquad (D)\quad \begin{array}{rrcrcrcrcrl} \min & 6y_1 & + & 6y_2 & + & 4y_3 & + & 4y_4 & & & \\ & y_1 & + & y_2 & + & 2y_3 & + & 2y_4 & - & y_5 & = 1 \\ & 2y_1 & - & 2y_2 & + & y_3 & - & y_4 & & & = 0 \\ & y_1 & , & y_2 & , & y_3 & , & y_4 & & y_5 & \geq 0 \end{array}$$

e la base $B^1 = \{1, 2\}$. Essendo

$$A_{B^1} = \begin{bmatrix} 1 & 2 \\ 1 & -2 \end{bmatrix} \quad , \quad A_{B^1}^{-1} = \begin{bmatrix} 1/2 & 1/2 \\ 1/4 & -1/4 \end{bmatrix} \quad , \quad b_{B^1} = \begin{bmatrix} 6 \\ 6 \end{bmatrix}$$

la corrispondente soluzione di base primale è

$$\bar{x}^1 = A_{B_1}^{-1} b_{B_1} = \begin{bmatrix} 1/2 & 1/2 \\ 1/4 & -1/4 \end{bmatrix} \begin{bmatrix} 6 \\ 6 \end{bmatrix} = \begin{bmatrix} 6 \\ 0 \end{bmatrix} \ ,$$

che risulta essere non ammissibile; infatti

$$A_{N^1} \bar{x}^1 = \begin{bmatrix} 2 & 1 \\ 2 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 6 \\ 0 \end{bmatrix} = \begin{bmatrix} 12 \\ 12 \\ -6 \end{bmatrix} \not\leq \begin{bmatrix} 4 \\ 4 \\ 0 \end{bmatrix} = b_{N^1} \ .$$

Viceversa, c può essere espresso come combinazione lineare a coefficienti non negativi di A_1 ed A_2, e quindi la base B^1 è duale ammissibile; infatti

$$\bar{y}_{B^1} = c A_{B^1}^{-1} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 1/4 & -1/4 \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \end{bmatrix} \quad , \quad \bar{y}^1 = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 & 0 \end{bmatrix} \geq 0 \ .$$

Possiamo quindi applicare l'Algoritmo del Simplesso Duale a partire dalla base B^1. L'indice entrante è

$$k = \min\{\ i \in N^1 \ : \ A_i \bar{x}^1 > b_i\ \} = \min\{3, 4\} = 3$$

e quindi

$$\eta_{B^1} = A_3 A_{B^1}^{-1} = \begin{bmatrix} 2 & 1 \end{bmatrix} \begin{bmatrix} 1/2 & 1/2 \\ 1/4 & -1/4 \end{bmatrix} = \begin{bmatrix} 5/4 & 3/4 \end{bmatrix}$$

$$\bar{\theta} = \min\left\{ \frac{\bar{y}_1^1}{\eta_1^1} \, , \, \frac{\bar{y}_2^1}{\eta_2^1} \right\} = \min\left\{ \frac{1/2}{5/4} \, , \, \frac{1/2}{3/4} \right\} = \min\left\{ \frac{2}{5} \, , \, \frac{2}{3} \right\} = \frac{2}{5} \ ;$$

pertanto l'indice uscente è $h = 1$. La nuova base è $B^2 = B^1 \setminus \{h\} \cup \{k\} = \{1, 2\} \setminus \{1\} \cup \{3\} = \{3, 2\}$. Effettuiamo quindi la seconda iterazione, dove

$$A_{B^2} = \begin{bmatrix} 2 & 1 \\ 1 & -2 \end{bmatrix} \quad , \quad A_{B^2}^{-1} = \begin{bmatrix} 2/5 & 1/5 \\ 1/5 & -2/5 \end{bmatrix} \quad , \quad b_{B^2} = \begin{bmatrix} 4 \\ 6 \end{bmatrix} \quad , \quad \bar{x}^2 = \begin{bmatrix} 2/5 & 1/5 \\ 1/5 & -2/5 \end{bmatrix} \begin{bmatrix} 4 \\ 6 \end{bmatrix} = \begin{bmatrix} 14/5 \\ -8/5 \end{bmatrix}$$

$$\bar{y}_{B^2} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 2/5 & 1/5 \\ 1/5 & -2/5 \end{bmatrix} = \begin{bmatrix} 2/5 & 1/5 \end{bmatrix} \quad , \quad \bar{y}^1 = \begin{bmatrix} 0 & 1/5 & 2/5 & 0 & 0 \end{bmatrix} \ .$$

La soluzione primale $\bar{x}^2$ non è ammissibile in quanto

$$A_{N^2} \bar{x}^2 = \begin{bmatrix} 1 & 2 \\ 2 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 14/5 \\ -8/5 \end{bmatrix} = \begin{bmatrix} -2/5 \\ 36/5 \\ -14/5 \end{bmatrix} \not\leq \begin{bmatrix} 6 \\ 4 \\ 0 \end{bmatrix} = b_{N^2}$$

e quindi l'indice entrante è $k = 4$ (l'indice dell'unico vincolo violato); pertanto

$$\eta_{B^2} = \begin{bmatrix} 2 & -1 \end{bmatrix} \begin{bmatrix} 2/5 & 1/5 \\ 1/5 & -2/5 \end{bmatrix} = \begin{bmatrix} 3/5 & 4/5 \end{bmatrix}$$

$$\bar{\theta} = \min\left\{ \frac{\bar{y}_3^2}{\eta_3^2} \, , \, \frac{\bar{y}_2^2}{\eta_2^2} \right\} = \min\left\{ \frac{2/5}{3/5} \, , \, \frac{1/5}{4/5} \right\} = \min\left\{ \frac{2}{3} \, , \, \frac{1}{4} \right\} = \frac{1}{4} \ ,$$

da cui l'indice uscente è $h = 2$. La nuova base è perciò $B_3 = \{3, 2\} \setminus \{2\} \cup \{4\} = \{3, 4\}$. Nella terza iterazione

$$A_{B^3} = \begin{bmatrix} 2 & 1 \\ 2 & -1 \end{bmatrix} \quad , \quad A_{B^3}^{-1} = \begin{bmatrix} 1/4 & 1/4 \\ 1/2 & -1/2 \end{bmatrix} \quad , \quad b_{B^3} = \begin{bmatrix} 4 \\ 4 \end{bmatrix} \quad , \quad \bar{x}^3 = \begin{bmatrix} 1/4 & 1/4 \\ 1/2 & -1/2 \end{bmatrix} \begin{bmatrix} 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

$$\bar{y}_{B^3} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1/4 & 1/4 \\ 1/2 & -1/2 \end{bmatrix} = \begin{bmatrix} 1/4 & 1/4 \end{bmatrix} \quad , \quad \bar{y}^3 = \begin{bmatrix} 0 & 0 & 1/4 & 1/4 & 0 \end{bmatrix} \ .$$

Stavolta la soluzione primale $\bar{x}^3$ è ammissibile, in quanto

$$A_{N^3} \bar{x}^3 = \begin{bmatrix} 1 & 2 \\ 1 & -2 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ -2 \end{bmatrix} \leq \begin{bmatrix} 6 \\ 6 \\ 0 \end{bmatrix} = b_{N^3}$$

e quindi l'algoritmo termina: $\bar{x}^3$, $\bar{y}^3$ costituiscono la coppia di soluzioni ottime associate alla base ottima $B = \{3, 4\}$.

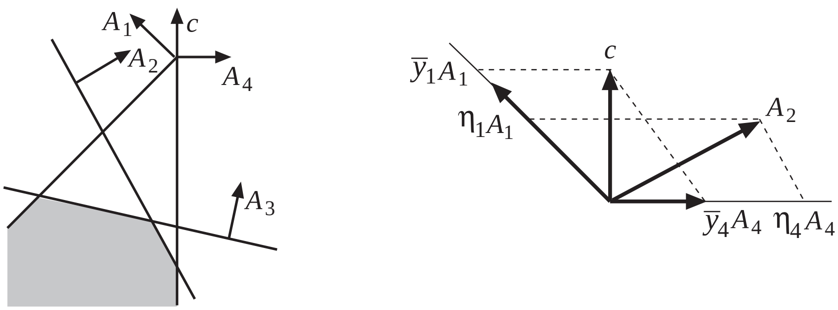


Figura 2.12: Interpretazione geometrica del Simplesso Duale (1)

Esempio 2.32: Interpretazione geometrica del Simplesso Duale
È possibile dare un'interpretazione geometrica delle operazioni effettuate dal Simplesso Duale *nello spazio del primale.* Per questo consideriamo l'esempio di Figura 2.12, dove accanto a ciascun vincolo è indicata la corrispondente riga di A. La base $B = \{1, 4\}$ è chiaramente duale ammissibile, in quanto c appartiene al cono finitamente generato da A_1 e A_4; non è però primale ammissibile, essendo violati i vincoli 2 e 3. In accordo alla regola anticiclo, l'algoritmo pone quindi $k = \min\{2, 3\} = 2$. La scelta del vincolo uscente dalla base è effettuata mediante il criterio del minimo rapporto; i coefficienti η_B sono le soluzioni del sistema

$$\eta_1 A_1 + \eta_4 A_4 = A_2,$$

che corrisponde ad esprimere A_2 come combinazione lineare dei vettori A_1 ed A_4. Poiché, come mostrato in figura, anche A_2 appartiene al cono finitamente generato da A_1 e A_4, si ha che $\eta_1 > 0$ e $\eta_4 > 0$. La figura mostra anche che $\bar{y}_1 A_1$ ha una norma maggiore di $\eta_1 A_1$, ossia si ha che $\bar{y}_1/\eta_1 > 1$, mentre $\bar{y}_4 A_4$ ha una norma minore di $\eta_4 A_4$, e quindi $\bar{y}_4/\eta_4 < 1$: il criterio del minimo rapporto seleziona quindi $h = 4$ come indice uscente da B, ottenendo la nuova base $\{1, 2\}$. Si noti che, operando in uno spazio bidimensionale, è possibile verificare le conseguenze che si possono avere eliminando uno o l'altro degli indici in base in seguito all'inserimento dell'indice $k = 2$. Le uniche basi adiacenti che si possono ottenere sono $\{2, 4\}$ e $\{1, 2\}$, corrispondenti rispettivamente all'uscita di 1 ed all'uscita di 4. È immediato verificare geometricamente che la seconda è duale ammissibile mentre la prima non lo è; siccome il Simplesso Duale individua sempre basi duali ammissibili, la nuova base non può essere che $\{1, 2\}$.
Un caso in cui (D) è illimitato, e quindi (P) è vuoto, è mostrato in Figura 2.13(a), ove $B = \{1, 2\}$ è una base duale ammissibile. I vincoli 3 e 4 sono violati dalla soluzione primale di base, quindi l'algoritmo selezione $k = 3$; il sistema $\eta_B A_B = A_3$ ha soluzioni negative, cioè $\eta_1 < 0$ e $\eta_2 < 0$, in quanto A_3 appartiene al cono finitamente generato dagli opposti dei vettori in base, $-A_1$ e $-A_2$.

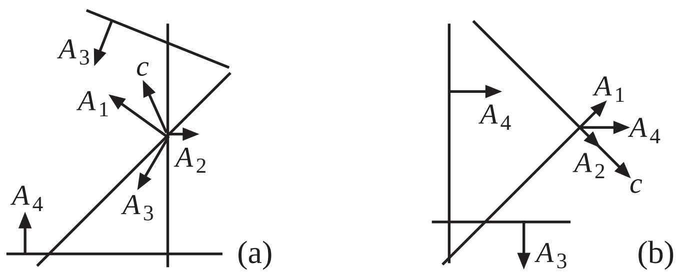


Figura 2.13: Interpretazione geometrica del Simplesso Duale (2)

Data la soluzione di base $[\ \bar{y}_1\ ,\ \bar{y}_2\ ,\ 0\ ,\ 0\ ]$ corrispondente a B, si ha quindi che

$$\bar{y}(\theta) = [\ \bar{y}_1 - \eta_1\theta\ ,\ \bar{y}_2 - \eta_2\theta\ ,\ \theta\ ,\ 0\ ]$$

è ammissibile per (D) per ogni $\theta \geq 0$; siccome si ha $[\ \eta_1\ ,\ \eta_2\ ] = A_3 A_B^{-1}$, risulta

$$-\eta_1 b_1 - \eta_2 b_2 + b_3 = -A_3 A_B^{-1} b_B + b_3 = -A_3 \bar{x} + b_3 < 0.$$

e pertanto la funzione obiettivo decresce indefinitamente al crescere di θ.
Il caso di una soluzione duale degenere è mostrato in Figura 2.13(b), ove la soluzione corrispondente alla base $B = \{1, 2\}$ è degenere essendo $\bar{y}_1 = 0$; infatti, c è collineare con A_2, ossia appartiene al cono finitamente generato dal solo A_2. È immediato verificare che $k = 4$ e $\eta_B = A_4 A_B^{-1} > 0$; per il criterio del minimo rapporto si ha $h = 1$ e la nuova base (non primale ammissibile) è $\{2, 4\}$, con uguale valore della funzione obiettivo.

Esercizio 2.21 *Si risolva geometricamente il problema dell'Esempio 2.31 mediante il Simplesso Duale, partendo dalla base* $B = \{1, 2\}$*; si confrontino i risultati ottenuti con quelli riportati nell'esempio.*

Individuazione di una base duale ammissibile.

Per completare lo studio del Simplesso Duale resta da considerare il caso in cui non è nota una soluzione di base duale ammissibile da cui partire. Nel problema duale (D) possiamo supporre senza perdita di generalità che sia $c \geq 0$: infatti, essendo i vincoli del problema in forma di uguaglianza, è possibile moltiplicare per -1 ogni vincolo per cui sia $c_i < 0$ (ciò corrisponde alla sostituzione di variabile $x_i = -x_i$ nel primale). Introduciamo quindi il *duale ausiliario* (ed il suo duale)

$$(DA) \quad \min \left\{ \, yb + wM \; : \; yA + w = c \, , \; y \geq 0 \, , \; w \geq 0 \, \right\} \qquad (PA) \quad \max \left\{ \, cx \; : \; Ax \leq b \, , \; x \leq M \, \right\}$$

dove M è un vettore con componenti positive e opportunamente grandi. Per (DA) è nota la base "artificiale" $B = \{ \, m+1 \, , \, \ldots \, , \, m+n \, \}$ contenente gli indici relativi alle variabili "artificiali" w, ossia le variabili duali dei vincoli "artificiali" $x \leq M$, con la corrispondente matrice di base $A_B = I$: è immediato verificare che la soluzione duale di base è $[\, \bar{y} \; , \; \bar{w} \,] = [\, 0 \, , \, c \,]$, che quindi è ammissibile $(c \geq 0)$. Da tale base si può quindi partire con l'algoritmo del Simplesso Duale. Si può dimostrare che, scegliendo le componenti di M sufficientemente grandi, (DA) è equivalente a D; più precisamente, se (D) è inferiormente illimitato allora anche (DA) lo è, indipendentemente dal valore scelto per le componenti di M, mentre se (D) non è inferiormente illimitato allora esiste un valore per tali componenti tale che, data una qualsiasi soluzione ottima di base $[\, y^* \, , \, w^* \,]$ per (DA), se $w^* = 0$ allora y^* è una soluzione ottima per (D), altrimenti (cioè se $w^* \neq 0$) allora (D) non possiede soluzioni ammissibili. È possibile però che (DA) risulti inferiormente non limitato anche nel caso in cui (D) sia vuoto; in tal caso almeno una variabile ausiliaria avrà valore positivo. Poiché determinare a priori un valore opportuno di M non è in generale possibile, si procede usualmente fissando un certo valore e risolvendo il problema per quel fissato valore; una volta determinata una base ottima, oppure che il problema è inferiormente illimitato, si usano le tecniche di analisi parametrica rispetto ai cambiamenti del vettore b (che in questo caso contiene anche M), descritte nel paragrefo 2.3.3, per determinare se il valore di M è corretto, oppure deve essere aumentato, oppure a nessun valore finito di M corrisponde una soluzione duale ammissibile (il che dimostra che (D) è vuoto). È in effetti possibile implementare l'algoritmo del Simplesso Duale in modo tale che risolva (DA) per un opportuno valore delle componenti di M, senza che esso venga scelto a priori; questi dettagli sono lasciati per esercizio al lettore.

Esempio 2.33: Individuazione di una base duale ammissibile
Si consideri la coppia di problemi duali:

$$(P) \quad \begin{array}{rrcrcr} \max & 4x_1 & + & 2x_2 & & \\ & -x_1 & + & 4x_2 & \leq & 2 \\ & x_1 & - & 2x_2 & \leq & -3 \\ & -x_1 & + & x_2 & \leq & -1 \end{array} \qquad (D) \quad \begin{array}{rrcrcrcl} \min & 2y_1 & - & 3y_2 & - & y_3 & & \\ & -y_1 & + & y_2 & - & y_3 & = & 4 \\ & 4y_1 & - & 2y_2 & + & y_3 & = & 2 \\ & y_1 & , & y_2 & , & y_3 & \geq 0 & \end{array}$$

Non avendo né una base primale né una base duale ammissibile di partenza, risolviamo il problema Duale Ausiliario (DA) ed il suo duale (PA), ponendo $M = [\, 20 \, , \, 20 \,]$:

$$(DA) \quad \begin{array}{rrcrcrcrcrl} \min & 2y_1 & - & 3y_2 & - & y_3 & + & 20w_1 & + & 20w_2 & \\ & -y_1 & + & y_2 & - & y_3 & + & w_1 & & & = 4 \\ & 4y_1 & - & 2y_2 & + & y_3 & & & + & w_2 & = 2 \\ & y_1 & , & y_2 & , & y_3 & , & w_1 & , & w_2 & \geq 0 \end{array} \qquad (PA) \quad \begin{array}{rrcrcr} \max & 4x_1 & + & 2x_2 & & \\ & -x_1 & + & 4x_2 & \leq & 2 \\ & x_1 & - & 2x_2 & \leq & -3 \\ & -x_1 & + & x_2 & \leq & -1 \\ & x_1 & & & \leq & 20 \\ & & & x_2 & \leq & 20 \end{array}$$

La base di partenza $B^1 = \{4, 5\}$ è quella relativa alle variabili ausiliarie w_1 e w_2.

it.1) $A_{B^1} = A_{B^1}^{-1} = I$, $\bar{x}^1 = A_{B^1}^{-1} b_{B^1} = b_{B^1} = [\, 20 \, , \, 20 \,]$, $\bar{y}^1 = [\, 0 \, , \, 0 \, , \, 0 \,]$, $\bar{w}^1 = [\, 4 \, , \, 2 \,]$ e il valore della funzione obiettivo è $[\, y^1 \, , \, y^1 \,][\, b \, , \, M \,] = 120$. Risulta quindi

$$A_{N^1} \bar{x}^1 = \begin{bmatrix} -1 & 4 \\ 1 & -2 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 20 \\ 20 \end{bmatrix} = \begin{bmatrix} 60 \\ -20 \\ 0 \end{bmatrix} \not\leq \begin{bmatrix} 2 \\ -3 \\ -1 \end{bmatrix} \quad ;$$

essendo violati due vincoli, per la regola anticiclo di Bland si ha $k = \min\{1, 3\} = 1$. Il corrispondente

$$\eta_{B^1} = A_k A_{B^1}^{-1} = A_1 = [\; -1 \, , \, 4 \;]$$

ha solo la seconda componente positiva, quindi l'indice uscente è $h = B^1(2) = 5$; pertanto la nuova base è $B^2 = B^1 \setminus \{5\} \cup \{1\} = \{4, 1\}$.

it.2)

$$A_{B^2} = \begin{bmatrix} 1 & 0 \\ -1 & 4 \end{bmatrix} \quad , \quad A_{B^2}^{-1} = \begin{bmatrix} 1 & 0 \\ 1/4 & 1/4 \end{bmatrix} \quad , \quad \bar{x}^2 = \begin{bmatrix} 20 \\ 11/2 \end{bmatrix}$$

$$[\,\bar{y}^2 \,|\, \bar{w}^2\,]_{B^2} = cA_{B^2}^{-1} = [\,4\,,\,2\,] \begin{bmatrix} 1 & 0 \\ 1/4 & 1/4 \end{bmatrix} = [\,9/2\,,\,1/2\,] \quad , \quad [\,\bar{y}^2 \,|\, \bar{w}^2\,] = [\,1/2\,,\,0\,,\,0 \,|\, 9/2\,,\,0\,]$$

e la funzione obiettivo vale quindi 91. La base non è primale ammissibile in quanto è violato il secondo vincolo:

$$A_{N^2}\bar{x}^2 = \begin{bmatrix} 9 \\ -29/2 \\ 11/2 \end{bmatrix} \not\leq \begin{bmatrix} -3 \\ -1 \\ 20 \end{bmatrix} \quad \Longrightarrow \quad k = 2 \quad \Longrightarrow \quad \eta_{B^2} = [\,1/2\,,\,-1/2\,]$$

e pertanto $h = 4$ (si ha di nuovo una sola componente positiva).

it.3)

$$B^3 = \{2, 1\} \quad , \quad A_{B^3} = \begin{bmatrix} 1 & -2 \\ -1 & 4 \end{bmatrix} \quad , \quad A_{B^3}^{-1} = \begin{bmatrix} 2 & 1 \\ 1/2 & 1/2 \end{bmatrix} \quad , \quad \bar{x}^3 = \begin{bmatrix} -4 \\ -1/2 \end{bmatrix}$$

$$[\,\bar{y}^3 \,|\, \bar{w}^3\,]_{B^3} = [\,9\,,\,5\,] \quad , \quad [\,\bar{y}^3 \,|\, \bar{w}^3\,] = [\,5\,,\,9\,,\,0 \,|\, 0\,,\,0\,]$$

e la funzione obiettivo vale -17. Si ha

$$A_{N^3}\bar{x}^3 = \begin{bmatrix} 7/2 \\ -4 \\ -1/2 \end{bmatrix} \not\leq \begin{bmatrix} -1 \\ 20 \\ 20 \end{bmatrix} \quad \Longrightarrow \quad k = 3 \quad \Longrightarrow \quad \eta_{B^3} = [\,-3/2\,,\,-1/2\,] \quad :$$

poiché $\eta_{B^3} \leq 0$, (DA) è inferiormente illimitato e di conseguenza anche (D) lo è, quindi (P) è vuoto.

Esercizio 2.22 *Si studi il problema (P) dell'esempio precedente, rappresentandolo geometricamente, e si verifichi che esso è vuoto. Inoltre, considerando il problema (PA), si interpretino geometricamente le iterazioni svolte nell'esempio.*

Esercizio 2.23 *Si usi l'informazione disponibile al termine dell'algoritmo per determinare un sistema inconsistente per il problema (P) dell'esempio precedente.*

2.3.3 Analisi post-ottimale

Data la coppia asimmetrica (P)–(D) di problemi duali, sia A_B una base ottima con $\bar{x}$ e $\bar{y} = [\,\bar{y}_B\,,\,0\,]$ la corrispondente coppia di soluzioni complementari ottime: vogliamo studiare come varia la coppia di soluzioni ottime al variare dei dati del problema.

Si tratta di un problema di notevole interesse pratico, per diversi motivi: innanzitutto, va ricordato che per costruire il modello di *PL* sono state fatte approssimazioni (spesso notevoli), ad esempio perché sono stati assunti come lineari fenomeni che non lo sono, oppure perché alcuni parametri non sono noti con precisione. È quindi utile conoscere quanto sia "stabile" la soluzione ottima ottenuta risolvendo il problema di *PL*, cioè quanto essa sia sensibile a piccole variazioni dei dati; in questo caso si parla di *analisi di sensitività.* Altre volte si considerano alcuni dei dati come funzione di uno o più parametri e ci si pone il problema di determinare il valore ottimo della funzione obiettivo come funzione dei parametri stessi; in questo caso si parla di *analisi parametrica.* Infine, un altro caso di notevole interesse pratico è quello in cui, a partire dalla soluzione trovata, si voglia risolvere un nuovo problema che differisca dal precedente solo in alcuni dei dati (ad esempio, un nuovo vincolo, una diversa funzione obiettivo o un diverso vettore di risorse); in questo caso si parla di *riottimizzazione.*

Tratteremo nel seguito, separatamente, il caso in cui le variazioni riguardano i vettori c e b ed i casi in cui viene aggiunto al problema, o da esso eliminato, un vincolo o una variabile.

Variazione del vettore c

Supponiamo innanzitutto che il vettore dei costi c venga sostituito da un nuovo vettore c': ovviamente, la matrice di base A_B continua ad essere primale ammissibile, e $\bar{x}$ continua ad essere la soluzione primale ad essa corrispondente. La soluzione duale diventa $\bar{y}' = [\bar{y}'_B, \bar{y}'_N] = [c'A_B^{-1}, 0]$: se $\bar{y}'_B \geq 0$, allora A_B rimane ottima, altrimenti si può applicare, a partire da A_B, il Simplesso Primale per determinare una nuova base ottima.

Consideriamo ora il caso in cui il vettore dei costi sia una funzione lineare di un parametro λ, ossia $c(\lambda) = c + \lambda\nu$ con ν un qualsiasi vettore reale, e si voglia determinare quale è l'intervallo in cui può variare λ senza che la base A_B perda l'ottimalità. Per questo, è sufficiente che risulti $y(\lambda)_B = (c + \lambda\nu)A_B^{-1} \geq 0$, cioè

$$(c+\lambda\nu)A_B^{-1}u_{B(i)} \geq 0 \qquad i \in B \ ,$$

dove, ricordiamo, $A_B^{-1}u_{B(i)}$ è la colonna dell'inversa della matrice di base relativa alla variabile di base $\bar{y}_i = cA_B^{-1}u_{B(i)}$. Esplicitando queste condizioni, risulta che, per ogni $i \in B$, deve essere

$$\lambda \geq \frac{-cA_B^{-1}u_{B(i)}}{\nu A_B^{-1}u_{B(i)}} = -\frac{\bar{y}_i}{\nu A_B^{-1}u_{B(i)}} \qquad \text{se } \nu A_B^{-1}u_{B(i)} > 0 \ ,$$

$$\lambda \leq \frac{-cA_B^{-1}u_{B(i)}}{\nu A_B^{-1}u_{B(i)}} = -\frac{\bar{y}_i}{\nu A_B^{-1}u_{B(i)}} \qquad \text{se } \nu A_B^{-1}u_{B(i)} < 0 \ ;$$

queste disequazioni definiscono l'intervallo desiderato (si noti che tale intervallo deve contenere lo 0, in quanto abbiamo supposto che B sia duale ammissibile). Può essere interessante studiare l'andamento del valore ottimo della funzione obiettivo in funzione del parametro λ: indicando con $z(\lambda)$ tale funzione, è facile vedere che si tratta di una funzione convessa lineare a tratti.

Esempio 2.34: Variazione del vettore c
Si consideri il problema dell'Esempio 2.31; al termine dell'applicazione del Simplesso Duale si è ottenuta la base ottima $B = \{3, 4\}$, a cui corrispondono le soluzioni ottime $\bar{x} = [\ 2\ ,\ 0\ ]$ e $\bar{y} = [\ 0\ ,\ 0\ ,\ 1/4\ ,\ 1/4\ ,\ 0\ ]$. Si consideri ora il gradiente del costo in forma parametrica: $c(\lambda) = c + \lambda\nu$, con $\nu = [\ 1\ ,\ 1\ ]$; si vuole conoscere per quali valori di $c(\lambda)$ la base B resta ottima. Dalle formule sopra esposte si ha che

$$\nu A_B^{-1} = [\ 1\ ,1\ ] \begin{bmatrix} 1/4 & 1/4 \\ 1/2 & -1/2 \end{bmatrix} = [\ 3/4\ , -1/4\ ] \ ,$$

pertanto

a) $\nu A_B^{-1}u_{B(1)} = 3/4 > 0 \quad \Longrightarrow \quad \lambda \geq -\frac{\bar{y}_3}{3/4} = -\frac{1/4}{3/4} = -1/3$

b) $\nu A_B^{-1}u_{B(2)} = -1/4 < 0 \quad \Longrightarrow \quad \lambda \leq \frac{\bar{y}_4}{-(-1/4)} = \frac{1/4}{1/4} = 1$

da cui si ottiene che $B = \{3, 4\}$ resta ottima per $-1/3 \leq \lambda \leq 1$. Per studiare la variazione del valore ottimo della funzione obiettivo al variare di λ si ricorda che $z(\lambda) = y(\lambda)_B b_B = (c + \lambda\nu)\bar{x}$, e quindi

$$z(\lambda) = [\ 1+\lambda\ , \lambda\ ] \begin{bmatrix} 2 \\ 0 \end{bmatrix} = 2 + 2\lambda \ :$$

quando λ varia nell'intervallo $[\,-1/3\ ,\ 1\,]$, il valore ottimo della funzione obiettivo varia linearmente nell'intervallo $[\,4/3\ ,\ 4\,]$.

Esercizio 2.24 *Si mostri geometricamente che, per il problema dell'esempio precedente, il vettore parametrico $c(\lambda)$ descrive il cono finitamente generato da A_3 e A_4, al variare di λ nell'intervallo $[\,-1/3\ ,\ 1\,]$.*

Esercizio 2.25 *Studiare le soluzioni ottime che si ottengono per $\lambda < -1/3$ e per $\lambda > 1$. Definire inoltre la funzione $z(\lambda)$ per qualsiasi valore di λ.*

Variazione del vettore b

Consideriamo il caso della variazione di una componente del vettore b (l'estensione al caso più generale è immediata), e sia b'_k il nuovo valore assunto da b_k; la soluzione duale non cambia, mentre per la soluzione primale si devono distinguere due casi:

$k \in N$: se risulta $A_k\bar{x} \leq b'_k$ allora A_B rimane ottima, altrimenti si può applicare il Simplesso Duale per determinare la nuova base ottima, conoscendo già l'indice k entrante in base;

$k \in B$: si calcola $\bar{x}' = A_B^{-1}b'_B$: se risulta $A_N\bar{x}' \leq b_N$ allora B rimane ottima, altrimenti si può applicare il Simplesso Duale per determinare la nuova base ottima.

Esempio 2.35: Variazione del vettore b

Si consideri la seguente coppia di problemi di PL

$$
(P)\quad \begin{array}{lrcrl}
\max & 3x_1 & + & x_2 & \\
 & -2x_1 & + & x_2 & \leq 1 \\
 & x_1 & - & 2x_2 & \leq -4 \\
 & x_1 & + & x_2 & \leq 14 \\
 & x_1 & & & \leq 8 \\
 & & - & x_2 & \leq -4
\end{array}
\qquad
(D)\quad \begin{array}{lrcrcrcrcrl}
\min & y_1 & - & 4y_2 & + & 14y_3 & + & 8y_4 & - & 4y_5 & \\
 & -2y_1 & + & y_2 & + & y_3 & + & y_4 & & & = 3 \\
 & y_1 & - & 2y_2 & + & y_3 & & & - & y_5 & = 1 \\
 & y_1 & , & y_2 & , & y_3 & , & y_4 & , & y_5 & \geq 0
\end{array}
$$

È facile verificare che la base ottima è $B^1 = \{2, 3\}$ e le soluzioni ottime sono $\bar{x}^1 = [\,8\ ,\ 6\,]$ e $\bar{y}^1 = [\,0\ ,\ 2/3\ ,\ 7/3\ ,\ 0\ ,\ 0\,]$. Sia ora $b'_4 = 7$ il nuovo valore del termine noto del quarto vincolo: siccome $4 \in N^1$, le soluzioni di base non cambiano. Si deve controllare se, dopo la perturbazione, il quarto vincolo è ancora soddisfatto:

$$A_4\bar{x}_1 = [\ 1\ ,\ 0\ ] \begin{bmatrix} 8 \\ 6 \end{bmatrix} = 8 > 7\ \ ;$$

poiché ciò non accade, la base B^1 non è primale ammissibile. Si applica allora il Simplesso Duale partendo da B^1:

$$A_{B^1} = \begin{bmatrix} 1 & -2 \\ 1 & 1 \end{bmatrix} \quad , \quad A_{B^1}^{-1} = \begin{bmatrix} 1/3 & 2/3 \\ -1/3 & 1/3 \end{bmatrix}$$

e l'indice entrante è ovviamente $k = 4$; si ha quindi

$$\eta_{B^1} = A_k A_{B^1}^{-1} = [\ 1\ ,\ 0\ ] \begin{bmatrix} 1/3 & 2/3 \\ -1/3 & 1/3 \end{bmatrix} = [\ 1/3\ ,\ 2/3\ ]\ \ .$$

Siccome entrambe le componenti di η_{B^1} sono positive, il passo di spostamento e l'indice uscente dalla base si ottengono mediante il criterio del minimo rapporto

$$\bar{\theta} = \min\left\{ \frac{\bar{y}^1_2}{\eta^1_2}\ ,\ \frac{\bar{y}^1_3}{\eta^1_3} \right\} = \min\left\{ \frac{2/3}{1/3}\ ,\ \frac{7/3}{2/3} \right\} = \min\left\{ 2\ ,\ \frac{7}{2} \right\} = 2\ \ ;$$

quindi $h = 2$ e si ottiene la nuova base $B^2 = \{4, 3\}$. Alla successiva iterazione si ha quindi

$$A_{B^2} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \quad , \quad A_{B^2}^{-1} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \quad , \quad \bar{x}^2 = \begin{bmatrix} 7 \\ 7 \end{bmatrix} \quad , \quad \bar{y}^2_{B^2} = [\ 2\ ,1\ ] \quad , \quad \bar{y}^2 = [\ 0, 0, 1, 2, 0\ ]$$

$$A_{N^2}\bar{x}^2 = \begin{bmatrix} -2 & 1 \\ 1 & -2 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} 7 \\ 7 \end{bmatrix} = \begin{bmatrix} -7 \\ -7 \\ -7 \end{bmatrix} < \begin{bmatrix} 1 \\ -4 \\ -4 \end{bmatrix} = b_{N^2}\ \ .$$

Poiché $\bar{x}^2$ è ammissibile, B^2 è ottima; $\bar{x}^2$ e $\bar{y}^2$ sono, rispettivamente, le soluzioni ottime dei problemi (P') e (D') ottenuti a seguito della variazione del vettore dei termini noti b.

Esercizio 2.26 *Si analizzino geometricamente i passi svolti nell'esempio precedente.*

In analogia a quanto fatto per il vettore dei costi, assumiamo che il vettore delle risorse sia una funzione lineare di un parametro λ, cioè del tipo $b(\lambda) = b + \lambda\nu$ con ν un qualsiasi vettore, e si voglia determinare quale è l'intervallo in cui può variare λ senza che la base A_B perda l'ottimalità: per questo, basta notare che la soluzione primale di base corrispondente a B per un dato valore di λ è

$$x(\lambda) = A_B^{-1}(b_B + \lambda\nu_B) = \bar{x} + \lambda A_B^{-1}\nu_B\ \ ,$$

per cui deve risultare

$$A_N x(\lambda) = A_N\bar{x} + \lambda A_N A_B^{-1}\nu_B \leq b_N + \lambda\nu_N\ \ ,$$

e, a partire da tali relazioni, è immediato determinare l'intervallo in cui può variare λ.

Esercizio 2.27 *Determinare, per la coppia di problemi dell'Esempio 2.35, l'intervallo in cui può variare λ senza che la base A_{B^1} perda l'ottimalità a seguito della variazione parametrica del vettore dei termini noti $b(\lambda) = b + \lambda\nu$ con $\nu = [\,0\ ,\ 1\ ,\ 1\ ,\ 0\ ,\ 0\,]$.*

Esercizio 2.28 *Si utilizzino le idee appena esposte per sviluppare i dettagli relativi alla determinazione di una base ammissibile per il Simplesso Duale. In particolare, sia B la base ottima di (DA) e si assuma che almeno una delle variabili "artificiali" w_i sia in base; si discuta come determinare se esiste oppure no un valore sufficientemente grande di M per cui la base non sia più ottima (w_i esca di base). Si estenda poi la discussione al caso in cui (DA) sia inferiormente illimitato ma almeno una variabile artificiale appartenga a $B \cup \{\,k\,\}$ al momento in cui l'algoritmo termina.*

Aggiunta o rimozione di un vincolo in (P) (di una variabile in (D))

Sia $A_{m+1}x \leq b_{m+1}$ il vincolo da aggiungere: se $A_{m+1}\bar{x} \leq b_{m+1}$, allora A_B rimane ottima, altrimenti si può applicare il simplesso duale per determinare la nuova base ottima, conoscendo già l'indice $(m+1)$ della variabile che dovrà entrare in base. Questo caso è equivalente a quello in cui si debba inserire una nuova variabile in (D).

La rimozione del vincolo i-esimo corrisponde a porre $b_i = +\infty$, e quindi può essere affrontata con i meccanismi descritti nel paragrafo precedente.

Aggiunta o rimozione di una variabile in (P) (di un vincolo in (D))

Sia x_{n+1} la nuova variabile da aggiungere, con corrispondente colonna A^{n+1} e coefficiente di costo c_{n+1}. Supponiamo che $x_{n+1} \geq 0$; in questo caso, oltre alla nuova colonna si aggiunge anche la riga corrispondente al vincolo di non negatività della nuova variabile. La nuova matrice è allora

$$A' = \begin{bmatrix} A & A^{n+1} \\ 0 & -1 \end{bmatrix} \qquad \text{e la sottomatrice} \qquad A'_{B'} = \begin{bmatrix} A_B & A_B^{n+1} \\ 0 & -1 \end{bmatrix}$$

è quadrata e non singolare, quindi è una matrice di base associata alla base $B' = B \cup \{m+1\}$, con $N' = N$. Alla nuova base corrispondono le seguenti soluzioni di base:

- $x' = [\,\bar{x}\,,\, x'_{n+1}\,] = [\,\bar{x}\,,\,0\,]$, che risulta essere ovviamente ammissibile;
- $y' = [\,y'_{B'}\,,\, y'_N\,]$, dove $y'_N = 0$ e $y'_{B'} = [\,\bar{y}_B\,,\, y'_{m+1}\,]$, con $y'_{m+1} = \bar{y}_B A_B^{n+1} - c_{n+1}$.

La base B' risulta duale ammissibile se e solo se $y'_{m+1} \geq 0$; nel caso B' non sia duale ammissibile e quindi non ottima, si ha comunque una base di partenza per la procedura *Simplesso Primale*, per cercare l'ottimo del problema trasformato partendo dalla base e dalle soluzioni ottime del problema originario.

Esercizio 2.29 *Si dimostri che le soluzioni x' e y', sopra indicate, sono le soluzioni di base associate alla base $B' = B \cup \{m+1\}$; si dimostrino le asserzioni fatte sull'ammissibilità primale e duale.*

Supponiamo ora che la nuova variabile x_{n+1} non sia vincolata in segno. In tal caso si possono applicare le trasformazioni (2.3), sostituendo la nuova variabile con la differenza di due variabili non negative, ed ottendo un problema equivalente la cui matrice dei vincoli è

$$A'' = \begin{bmatrix} A & A^{n+1} & -A^{n+1} \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{e la cui sottomatrice} \qquad A''_{B''} = \begin{bmatrix} A_B & A_B^{n+1} & -A_B^{n+1} \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

è la matrice di base associata alla base $B'' = B \cup \{m+1, m+2\}$, con $N'' = N$. A B'' sono associate le seguenti soluzioni di base:

- $x'' = [\ \bar{x}\ ,\ x^+_{n+1}\ ,\ x^-_{n+1}\ ] = [\ \bar{x}\ ,\ 0\ ,\ 0\ ]$, che risulta essere ammissibile;
- $y'' = [\ y''_{B''}\ ,\ y''_N\ ]$, dove $y''_N = 0$ e $y''_{B''} = [\ \bar{y}_B\ ,\ y''_{m+1}\ ,\ y''_{m+2}\ ]$, con $y''_{m+1} = y''_{m+2} = \bar{y}_B A_B^{n+1} - c_{n+1}$.

Anche in questo caso la base B'' risulta essere duale ammissibile, e quindi ottima, se e solo se $y''_{m+1} \geq 0$ (e quindi $y''_{m+2} \geq 0$). Se ciò non si verifica, B'' costituisce una base di partenza per il Simplesso Primale.

Esercizio 2.30 *Si dimostri che le soluzioni x'' e y'' sopra indicate sono le soluzioni di base associate alla base $B'' = B \cup \{m+1, m+2\}$; si dimostrino le asserzioni fatte sull'ammissibilità primale e duale.*

Esempio 2.36: Aggiunta di una variabile
Si consideri la coppia di problemi di PL dell'Esempio 2.35, a cui viene aggiunta una nuova variabile x_3, la relativa colonna A^3 e la componente c_3 riportati nella seguente coppia di problemi trasformati

$$
(P') \quad \begin{array}{rrrrrrrr}
\max & 3x_1 & + & x_2 & - & 2x_3 & & \\
& -2x_1 & + & x_2 & + & x_3 & \leq & 1 \\
& x_1 & - & 2x_2 & - & 3x_3 & \leq & -4 \\
& x_1 & + & x_2 & & & \leq & 14 \\
& x_1 & & & + & x_3 & \leq & 8 \\
& & - & x_2 & + & 2x_3 & \leq & -4 \\
& & & & - & x_3 & \leq & 0
\end{array}
\qquad
(D') \quad \begin{array}{rrrrrrrrrrrrr}
\min & y_1 & - & 4y_2 & + & 14y_3 & + & 8y_4 & - & 4y_5 & & & \\
& -2y_1 & + & y_2 & + & y_3 & + & y_4 & & & & = & 3 \\
& y_1 & - & 2y_2 & + & y_3 & & & - & y_5 & & = & 1 \\
& y_1 & - & 3y_2 & & & + & y_4 & + & 2y_5 & - \; y_6 & = & -2 \\
& y_1 & , & y_2 & , & y_3 & , & y_4 & , & y_5 & , \; y_6 & \geq & 0
\end{array}
$$

La base ottima per (P) è $B = \{2,3\}$, e le corrispondenti soluzioni ottime sono $\bar{x} = [\,8\,,\,6\,]$ e $\bar{y} = [\,0\,,\,2/3\,,\,7/3\,,\,0\,,\,0\,]$. Inoltre, la matrice di base e la sua inversa sono:

$$
A_B = \begin{bmatrix} 1 & -2 \\ 1 & 1 \end{bmatrix} \quad , \quad A_B^{-1} = \begin{bmatrix} 1/3 & 2/3 \\ -1/3 & 1/3 \end{bmatrix} \; .
$$

La base per il problema trasformato (P') è $B' = B \cup \{6\} = \{2,3,6\}$ ($N' = N = \{1,4,5\}$), in cui il vincolo $-x_3 \leq 0$ è attivo. La nuova matrice di base e la sua inversa sono

$$
A'_{B'} = \begin{bmatrix} A_B & A_B^{n+1} \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & -3 \\ 1 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \quad , \quad A'^{-1}_{B'} = \begin{bmatrix} 1/3 & 2/3 & -1 \\ -1/3 & 1/3 & 1 \\ 0 & 0 & -1 \end{bmatrix}
$$

e quindi è facile verificare che la corrispondente soluzione primale di base

$$
\bar{x}' = A'^{-1}_{B'} b_{B'} = \begin{bmatrix} 1/3 & 2/3 & -1 \\ -1/3 & 1/3 & 1 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} -4 \\ 14 \\ 0 \end{bmatrix} = \begin{bmatrix} 8 \\ 6 \\ 0 \end{bmatrix}
$$

è esattamente $\bar{x}' = [\,\bar{x}\,,\,0\,]$. La soluzione duale di base è data da

$$
\bar{y}'_{B'} = c' A'^{-1}_{B'} = \begin{bmatrix} 3 & 1 & -2 \end{bmatrix} \begin{bmatrix} 1/3 & 2/3 & -1 \\ -1/3 & 1/3 & 1 \\ 0 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 2/3 & 7/3 & 0 \end{bmatrix} \quad , \quad \bar{y}' = \begin{bmatrix} 0 & 2/3 & 7/3 & 0 & 0 & 0 \end{bmatrix} \; .
$$

Si noti che si è ottenuto $\bar{y}'_6 = 0$; infatti, per verifica, applicando la formula fornita si ottiene:

$$
\bar{y}'_6 = \bar{y}_B A_B^6 - c_6 = \begin{bmatrix} 2/3 & 7/3 \end{bmatrix} \begin{bmatrix} -3 \\ 0 \end{bmatrix} + 2 = -2 + 2 = 0 \; .
$$

Si è ottenuta una soluzione duale ammissibile e degenere; pertanto la base B' è ottima, e le soluzioni di base ottime, rispettivamente per (P') e (D'), sono $\bar{x}'$ e $\bar{y}'$.

La rimozione della variabile x_j corrisponde all'inserimento nel problema del vincolo $x_j = 0$, o equivalentemente dei due vincoli $x_j \geq 0$ e $x_j \leq 0$, e quindi può essere affrontata con i meccanismi descritti nel paragrafo precedente.

Riferimenti Bibliografici

F.S. Hillier, G.J. Lieberman, "**Introduzione alla ricerca operativa**", Franco Angeli, Milano (1999).

F. Maffioli, "**Elementi di programmmazione matematica**", Casa Editrice Ambrosiana, Milano (2000).

K.G. Murty, "**Linear and combinatorial programming**", Wiley, New York (1976).

M. Padberg, "**Linear optimization and extensions**", Springer-Verlag, Berlino (1995).

A. Sassano, "**Modelli e algoritmi della ricerca operativa**", Franco Angeli, Milano (1999).

M. Pappalardo, M. Passacantando, "**Ricerca Operativa**", Edizioni Plus, Pisa (2010).

P. Serafini, "**Ottimizzazione**", Zanichelli, Bologna (2000).

Capitolo 3

Grafi e reti di flusso

Molti problemi di ottimizzazione sono caratterizzati da una struttura di grafo: in molti casi questa struttura emerge in modo naturale, in altri nasce dal particolare modo in cui i problemi vengono modellati. Ad esempio, una rete stradale è naturalmente rappresentabile come un grafo in cui i nodi sono gli incroci e gli archi le strade. Non è pertanto strano che il settore dei trasporti sia uno di quelli in cui la teoria dei grafi trovi maggiore applicazione. In molti altri problemi, invece, la struttura di grafo è più nascosta.

In questo capitolo studieremo alcuni problemi di base definiti su grafi e reti. Di essi forniremo le proprietà più importanti e introdurremo alcuni algoritmi risolutivi, in generale i più semplici; per maggiori approfondimenti, si rinvia alla letteratura indicata e a corsi avanzati dell'area di Ricerca Operativa. Tutti i problemi che tratteremo sono "facili", ossia ammettono algoritmi risolutivi di complessità polinomiale (e molto efficienti in pratica). Inoltre, tutti i problemi che studieremo sono casi particolari del problema di Programmazione Lineare, e pertanto potrebbero essere risolti utilizzando direttamente gli strumenti presentati nel Capitolo 2. È però opportuno presentare approcci specifici per questa classe di problemi, per almeno due motivi:

1. algoritmi specializzati possono essere ordini di grandezza più efficienti in pratica, e questo è rilevante in quanto problemi su grafi si incontrano in moltissime branche della scienza quali la matematica, la fisica, la biologia, e l'informatica;

2. come vedremo, i problemi di cui discuteremo si pongono in una particolare posizione, ovvero "nell'interfaccia tra l'ottimizzazione continua e quella combinatoria": sono cioè problemi combinatori che ammettono algoritmi polinomiali perché possono essere affrontati con tecniche efficienti di ottimizzazione per problemi continui, cosa che in generale è difficile, come verrà discusso in dettaglio nel Capitolo 4.

3.1 Flussi su reti

In questo capitolo daremo per noti i concetti elementari di teoria dei grafi riassunti nell'Appendice B, facendo specifici rinvii a tale appendice solo quando necessario. Introdurremo ora alcuni concetti tipici dei problemi di ottimizzazione su grafo.

Con il termine "rete" indichiamo un grafo $G = (N, A)$ pesato, cioè ai cui nodi ed archi sono associati valori numerici; nel seguito indicheremo con $n = |N|$ il numero di nodi e con $m = |A|$ il numero di archi. In generale, in una rete gli archi sono interpretabili come canali attraverso cui fluiscono dei beni, che possono essere rappresentati per mezzo di grandezze discrete (ad esempio il numero di auto su una strada, o il numero di messaggi su una rete di comunicazione) o continue (quantità di petrolio che fluisce in un oleodotto), possono rappresentare dei valori assoluti oppure dei valori relativi (per unità di tempo). In questo contesto, i pesi degli archi rappresentano usualmente delle capacità e dei costi, mentre i pesi dei nodi rappresentano la quantità dei beni che entrano nella rete in quei nodi, o che ne escono. Più precisamente, nei problemi di cui tratteremo:

- ad ogni nodo $i \in N$ è associato un valore reale b_i, che può essere:
  - positivo, e in tal caso rappresenta la quantità del bene che esce dalla rete al nodo i; b_i è allora detto *domanda del nodo*, ed il nodo viene detto *destinazione*, *pozzo* o *nodo di output*;
  - negativo, e in tal caso rappresenta la quantità di bene che entra nella rete al nodo i; $-b_i$ è allora detto *offerta del nodo*, ed il nodo viene detto *origine*, *sorgente* o *nodo di input*;
  - nullo, ed in questo caso i viene detto *nodo di trasferimento*;
- ad ogni arco $a_k = (i, j)$ sono associati un *costo* c_k (o c_{ij}), che indica il costo che viene pagato per ogni unità del bene che attraversi l'arco, ed una *capacità inferiore* l_k (l_{ij}) e *superiore* u_k (u_{ij}), che indicano, rispettivamente, il minimo ed il massimo numero di unità di bene che possono attraversare l'arco. In molte applicazioni la capacità inferiore viene assunta uguale a 0, e quindi viene fornita tra i parametri della rete solamente la capacità superiore.

Nei problemi di flusso la domanda globale, cioè la somma di tutte le domande, è uguale all'offerta globale, cioè alla somma, cambiata di segno, di tutte le offerte; più formalmente, detti D e O rispettivamente gli insiemi dei nodi di domanda e di offerta

$$D = \{\, i \in N \,:\, b_i > 0 \,\} \quad \text{e} \quad O = \{\, i \in N \,:\, b_i < 0 \,\} \quad \text{si ha} \quad \textstyle\sum_{i \in D} b_i = -\sum_{i \in O} b_i \enspace .$$

In altre parole, il vettore b, detto vettore dei *bilanci* dei nodi, deve soddisfare la relazione $\sum_{i \in N} b_i = 0$. Questa, come vedremo, è una condizione necessaria (ma non sufficiente) affinché esista un vettore $x = [x_{ij}]_{(i,j) \in A} \in \mathbb{R}^m$ che soddisfa i *vincoli di conservazione del flusso*

$$\textstyle\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} = b_i \qquad i \in N \enspace , \tag{3.1}$$

dove $BS(i)$ e $FS(i)$ sono rispettivamente la stella entrante e la stella uscente di $i \in N$ (si veda l'Appendice B); un siffatto x si dice un *flusso* su G, ed il valore x_k (o x_{ij}) è detto *flusso dell'arco* $a_k = (i, j)$. Un flusso è poi detto *ammissibile* se sono verificati i *vincoli di capacità sugli archi*

$$l_{ij} \leq x_{ij} \leq u_{ij} \qquad (i, j) \in A \enspace . \tag{3.2}$$

Associando adesso ad ogni flusso x un costo dato dalla somma dei flussi degli archi per il loro costo

$$cx = \textstyle\sum_{(i,j) \in A} c_{ij} x_{ij}$$

possiamo definire il *problema del flusso di costo minimo* (MCF, da *Min Cost Flow* problem) come

$$(\text{MCF}) \qquad \min \{\, cx \,:\, Ex = b \,,\; l \leq x \leq u \,\} \tag{3.3}$$

dove $Ex = b$ rappresenta in forma vettoriale i vincoli di conservazione del flusso (3.1) utilizzando la *matrice di incidenza* del grafo G (si veda il §B.2.1) e $b = [b_i]_{i \in N}$, mentre $l \leq x \leq u$ rappresenta in forma vettoriale i vincoli di capacità (3.2) dove $l = [l_{ij}]_{(i,j) \in A}$ e $u = [u_{ij}]_{(i,j) \in A}$.

Esempio 3.1: Un'istanza di (MCF)
Sia dato il grafo orientato in Figura 3.1, in cui sono riportati domande, offerte, costi e capacità superiori (si suppone che le capacità inferiori siano nulle). I vincoli di conservazione del flusso dei 5 nodi sono i seguenti:

$$\begin{array}{cccccccccccc}
-x_{14} & -x_{15} & +x_{21} & & & +x_{31} & & & +x_{51} & & = & 0 \\
 & & -x_{21} & -x_{24} & -x_{25} & & +x_{42} & & & & = & -3 \\
 & & & & & -x_{31} & & +x_{43} & & +x_{53} & = & 2 \\
+x_{14} & & & +x_{24} & & & -x_{42} & -x_{43} & & & = & 6 \\
 & +x_{15} & & & +x_{25} & & & & -x_{51} & -x_{53} & = & -5
\end{array}$$

Si noti la forma della matrice di incidenza E. I vincoli di capacità degli archi sono

$$\begin{array}{ccccc}
0 \leq x_{14} \leq 5 & 0 \leq x_{15} \leq 1 & 0 \leq x_{21} \leq 1 & 0 \leq x_{24} \leq 3 & 0 \leq x_{25} \leq 4 \\
0 \leq x_{31} \leq 8 & 0 \leq x_{42} \leq 5 & 0 \leq x_{43} \leq 2 & 0 \leq x_{51} \leq 7 & 0 \leq x_{53} \leq 1
\end{array}$$

mentre il costo del flusso è

$$cx = -x_{14} + x_{15} - 4x_{21} + 4x_{24} + 2x_{25} + 4x_{31} + x_{42} + 3x_{51} + 6x_{53} \enspace .$$

Esercizio 3.1 *Determinare un flusso ammissibile per la rete in Figura 3.1 e valutarne il costo.*

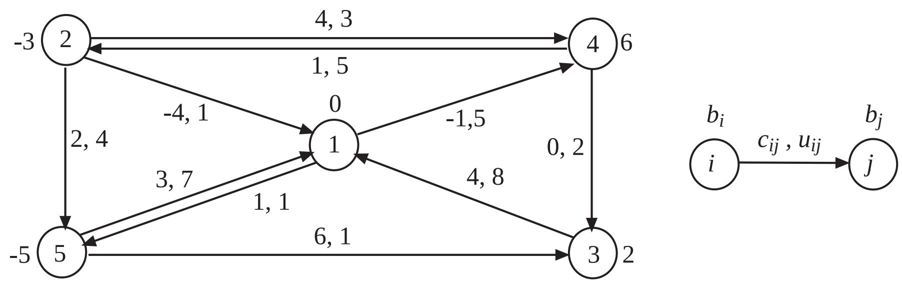


Figura 3.1: Un grafo con domande, offerte, costi e capacità

3.1.1 Alcuni modelli di flusso

Esiste un gran numero di problemi reali, in ambiti molto vari, che si modellano efficacemente come problemi di flusso: ne riportiamo di seguito alcuni esempi, caratterizzati dal fatto che il grafo sul quale il flusso è definito non è immediatamente evidente dalla descrizione del problema.

Schedulazione della produzione

L'industria dolciaria PereCani, nota produttrice di panettoni, deve decidere come utilizzare al meglio, nel corso dell'anno, la sua linea di produzione per tali dolci. La PereCani conosce una stima b_i, $i = 1, \ldots, 12$, del numero di panettoni che venderà in ogni mese dell'anno. Il costo unitario di produzione e la massima quantità di panettoni producibile in un mese variano anch'esse, a seconda di alcuni fattori quali il prezzo delle materie prime e la disponibilità di personale impegnato anche su altre linee di produzione, e anche di esse si hanno le stime, rispettivamente, c_i ed u_i, $i = 1, \ldots, 12$. I panettoni prodotti al mese i possono essere venduti immediatamente, oppure immagazzinati per essere poi venduti nei mesi successivi: il magazzino ha una capacità massima di U, ed un costo unitario di immagazzinamento pari a C. All'inzio il magazzino contiene b_0 panettoni, e si desidera che alla fine dell'anno ne contenga b_{13}.

Il problema della PereCani, noto in letteratura come problema di *Lot Sizing*, può essere formulato come un problema di flusso di costo minimo come mostrato in figura 3.2. Gli archi dal nodo fittizio 0 ai nodi 1, ..., 12 rappresentano la produzione, mentre gli archi di tipo $(i, i+1)$ rappresentano il magazzino. I bilanci ai nodi sono scelti in modo da rappresentare la vendita di panettoni in ogni mese, al netto dei panettoni già presenti in magazzino (per il nodo 1) o di quelli che dovranno esservi lasciati (per il nodo 12); il bilancio al nodo 0 è la produzione totale di panettoni durante l'anno.

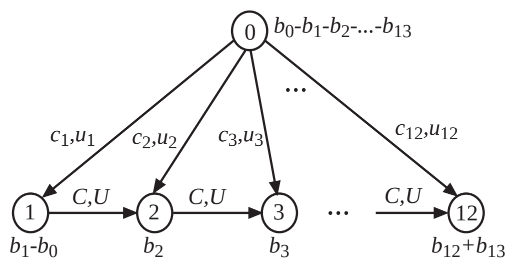


Figura 3.2: Il problema della PereCani

Schedulazione di viaggi di camion

La ditta di trasporti Andamiento Lento deve organizzare una giornata lavorativa per i suoi camion. La ditta deve effettuare n viaggi, ognuno caratterizzato da un tempo di inizio, una località di origine, un tempo di percorrenza ed una località di destinazione. I camion della ditta, tutti uguali, all'inizio della giornata sono tutti nello stesso deposito, e devono tutti trovarsi nel deposito alla fine della giornata. La località di origine del generico viaggio i, $i = 1, \ldots, n$, può essere raggiunta da un camion che parte dal deposito prima dell'istante di partenza del viaggio corrispondente, ed è noto il costo c_i^I (del carburante) che questo comporta; analogamente, è sempre possibile raggiungere il deposito dalla località di destinazione del viaggio i, $i = 1, \ldots, n$, prima della fine della giornata lavorativa, ed è noto il costo c_i^F che questo comporta. Una coppia di viaggi (i, j) è detta *compatibile* se essi possono essere effettuati, nell'ordine dato, dallo stesso camion; cioè se è possibile per il camion che ha effettuato il viaggio i, partire dalla località di destinazione di i e raggiungere la località di origine del viaggio j, prima del tempo di inizio di j. Per ogni coppia (i, j) di viaggi compatibile, è noto il costo c_{ij} del viaggio (a vuoto) tra la località di destinazione di i e la localit di origine di j. Inoltre, esiste un costo fisso C che deve essere pagato per ogni camion che viaggia durante la giornata, indipendentemente dal numero di viaggi effettuati e dai km percorsi. Supponendo che la Andamiento Lento possegga abbastanza camion per eseguire tutti i viaggi (ovviamente non ne serviranno più di n), si vuole formulare il problema di scegliere quali viaggi far effettuare dallo stesso camion in modo da minimizzare il costo complessivo, dato dalla somma dei costi dei movimenti a vuoto (dal deposito alla prima origine, dalle destinazioni alle origini dei viaggi successivi e dalla destinazione dell'ultimo viaggio nuovamente al deposito) e dei costi fissi.

Questo problema può essere formulato come un problema di flusso di costo minimo nel seguente modo. Il grafo G ha $2n + 2$ nodi: un nodo origine s, un nodo destinazione t, e n coppie di nodi (i', i''), uno per ogni viaggio i, $i = 1, \ldots, n$. I nodi i' hanno bilancio 1, i nodi i'' hanno bilancio -1 mentre s e t hanno bilancio 0. Esistono archi (s, i') per ogni $i = 1, \ldots, n$, con costo c_i^I e capacità 1; analogamente, esistono archi (i'', t) per ogni $i = 1, \ldots, n$, con costo c_i^F e capacità 1. Per ogni coppia di viaggi (i, j) compatibile esiste un arco (i'', j') con costo c_{ij} e capacità 1. Infine, esiste un arco (t, s) con costo C e capacità infinita (anche se il flusso non può superare n). Un esempio di grafo per un problema con 5 viaggi è mostrato in figura 3.3, in cui il viaggio 1 è compatibile col 2 e col 3, il viaggio 2 è compatibile col 3 e col 5, il viaggio 4 è compatibile col 5 mentre i viaggi 3 e 5 non sono compatibili con nessun viaggio. Per semplicità, nella figura sono mostrati i bilanci ai nodi ma non i costi e le capacità degli archi. I nodi i' ed i'' rappresentano rispettivamente l'inizio e la fine del viaggio i, ed il flusso rappresenta i camion: un'unità di flusso su un arco (s, i') indica che il viaggio i viene effettuato da un camion appena uscito dal deposito, un'unità di flusso su un arco (i'', j') indica che i viaggi i e j vengono effettuati (in sequenza) dallo stesso camion mentre un'unità di flusso su un arco (j'', t) indica che il camion che effettua il viaggio j torna immediatamente dopo al deposito. I vincoli di equilibrio ai nodi i' ed i'' garantiscono rispettivamente che il viaggio i sia compiuto da esattamente un camion (o proveniente dal deposito oppure a seguito di un altro viaggio) e che il camion che ha compiuto i torni al deposito oppure compia un altro viaggio. I vincoli di equilibrio ai nodi s e t garantiscono che tutti i camion che escono dal deposito vi rientrino; il numero di camion usati è pari al flusso sull'arco (t, s).

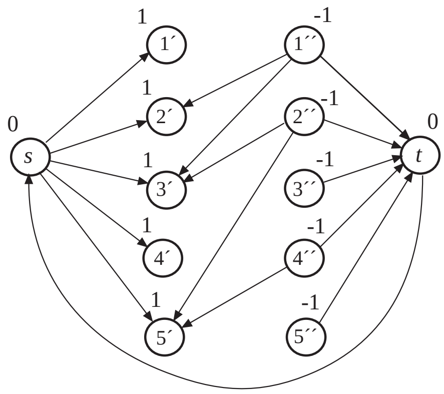


Figura 3.3: Schedulazione di viaggi

3.1.2 Trasformazioni equivalenti

Molti problemi reali possono essere formulati come problemi di flusso di costo minimo; questo è in parte dovuto alla grande flessibilità del modello stesso. Infatti, sui problemi di (MCF) si possono fare alcune assunzioni che ne semplificano la descrizione e l'analisi: se tali assunzioni non sono rispettate dall'istanza che effettivamente occorre risolvere, è sempre possibile costruire un'istanza di (MCF), equivalente a quella data, che le rispetti. Tali assunzioni sono:

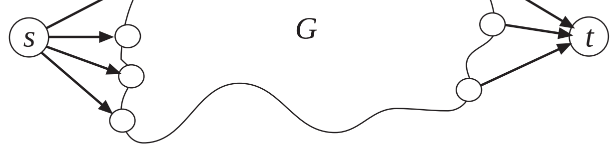


Figura 3.4: La rete ampliata G'

- *Singola sorgente – singolo pozzo.* Se si hanno più sorgenti e/o pozzi, è possibile introdurre una *rete ampliata* $G' = (N', A')$ in cui $N' = N \cup \{ s , t \}$ ed $A' = A \cup \{ (s,j) \, : \, j \in O \} \cup \{ (i,t) \, : \, i \in D \}$, dove i nodi fittizi s e t sono "la nuova sorgente e il nuovo pozzo". Ad ogni arco fittizio

$(s, j) \in FS(s)$ viene associata una capacità $u_{sj} = -b_j$, uguale cioè al flusso in ingresso a $j \in O$ nel problema originario; analogamente, ad ogni arco fittizio $(i, t) \in BS(t)$ viene associata una capacità $u_{it} = b_i$, uguale cioè al flusso in uscita da $i \in D$ nel problema originario; tutti gli archi fittizi hanno costo 0. L'offerta di s e la domanda di t sono date da

$$b_s = \sum_{j \in O} b_j \qquad\qquad b_t = \sum_{i \in D} b_i$$

mentre tutti gli altri nodi sono di trasferimento ($b_i = 0$ per $i \neq s, t$). Un esempio di trasformazione della rete G nella rete ampliata G' è mostrato in Figura 3.4. È facile dimostrare che ad ogni flusso ammissibile x' di G' corrisponde un flusso ammissibile x di G, e viceversa; infatti x' satura tutti gli archi (s, j) (e, di conseguenza, tutti gli archi (i, t)) e quindi le offerte e le domande ai nodi in O e in D sono rispettate.

- *Capacità inferiori nulle.* Se un arco (i, j) ha capacità inferiore $l_{ij} \neq 0$, questo vuol dire che il flusso x_{ij} dovrà essere almeno pari a l_{ij}. Quindi, è possibile costruire un'istanza equivalente sottraendo l_{ij} a b_j e u_{ij}, sommando l_{ij} a b_i, ed aggiungendo un termine costante $c_{ij}l_{ij}$ alla funzione obiettivo: questo equivale a sostituire la variabile di flusso x_{ij} con $x'_{ij} = x_{ij} - l_{ij}$, e considerare l_{ij} unità di flusso permanentemente presenti sull'arco (i, j). È facile verificare che il problema della Andamiento Lento, descritto nel pragarafo precedente, è equivalente al problema di flusso di costo minimo in cui tutti i bilanci ai nodi sono nulli ed esistono archi (i', i'') con capacità sia inferiore che superiore pari a 1.

- *Nessuna capacità associata ai nodi.* In alcuni casi esiste una limitazione superiore u_i e/o una limitazione inferiore l_i alla massima quantità di flusso che può attraversare un nodo $i \in N$. Per questo, è sufficiente costruire un nuovo grafo G' in cui il nodo i è sostituito da due nodi i' ed i''. Tutti gli archi $(j, i) \in BS(i)$ vengono sostituiti con archi del tipo (j, i'), mentre gli archi $(i, j) \in FS(i)$ vengono sostituiti con archi del tipo (i'', j), con costi e capacità uguali agli archi originali. Inoltre, viene aggiunto un arco (i', i'') con costo 0 e capacità superiore ed inferiore rispettivamente u_i e l_i. La domanda del nodo i viene attribuita ad i' se è positiva ed a i'' se negativa.

- *Eccesso di domanda o di offerta.* In alcuni problemi, il valore $-b_i$ per $i \in O$ non rappresenta l'offerta al nodo i, ma la *massima* offerta che il nodo i può fornire alla rete. Se i valori b_j per $j \in D$ rappresentano effettivamente la domanda di flusso dei nodi pozzo, il problema che si vuole risolvere è quello di determinare un flusso che soddisfi tutte le domande e per cui ogni nodo $i \in O$ fornisca al più $-b_i$ unità di flusso; chiaramente dovrà risultare $-\sum_{i \in O} b_i \geq \sum_{i \in D} b_i$. Una formulazione del problema si ottiene da (MCF) modificando i vincoli (3.1) per $i \in O$ in

$$\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} \geq b_i \qquad\qquad i \in O \ .$$

  È facile trasformare questo problema in un'istanza di (MCF) ricorrendo ancora una volta ad una rete ampliata $G' = (N', A')$, in cui $N' = N \cup \{\, s \,\}$ e $A' = A \cup \{\, (s, i) \,:\, i \in O \,\}$. In altri termini, si ha un nuovo nodo sorgente s la cui offerta coincide con la domanda globale dei nodi pozzo ($b_s = -\sum_{j \in D} b_j$), mentre si pone $b_i = 0$ per ogni $i \in O$ in modo che risulti $\sum_{i \in N'} b_i = 0$. Per ogni $i \in O$, l'arco fittizio (s, i) ha costo 0 e capacità $u_{si} = -b_i$. La trasformazione effettuata permette, una volta calcolati i flussi su G', di conoscere, per ciascuna sorgente $i \in O$, la capacità produttiva effettivamente utilizzata, x_{si}, e quella non utilizzata, $u_{si} - x_{si}$. In modo analogo si tratta il caso in cui per ciascun nodo $i \in O$ è data l'offerta $-b_i$, mentre per ciascun nodo $j \in D$ è data una limitazione superiore b_j del valore che la domanda può assumere.

3.1.3 Algoritmi del Simplesso per (MCF)

Poiché (MCF) è un problema di PL, è possibile risolverlo applicando gli algoritmi del Simplesso presentati nel Capitolo 2. La struttura molto particolare del problema consente tuttavia di derivare proprietà specifiche che permettono di velocizzare tali algoritmi in modo sostanziale. In questo paragrafo definiremo brevemente tali proprietà ed indicheremo come sia possibile farne uso, senza entrare nei dettagli che verranno lasciati per esercizio al lettore.

Innanzitutto, essendo un problema di *PL* (MCF) ha un duale, ossia

$$\text{(DMCF)} \qquad \max\left\{ \, yb - wu \; : \; yE - w \leq c \; , \; w \geq 0 \, \right\} \; .$$

Le variabili duali possono quindi essere distinte in: i *potenziali* $[y_i]_{i \in N}$ associati ai vincoli di conservazione del flusso (e quindi ai nodi), e le variabili $[w_{ij}]_{(i,j) \in A}$ associate ai vincoli di capacità (e quindi agli archi). Data la struttura della matrice di incidenza E, i vincoli duali (uno per ciascun arco) si possono riscrivere come

$$y_j - y_i - w_{ij} \leq c_{ij} \qquad (i,j) \in A \; . \tag{3.4}$$

Si noti come, a parte per i vincoli di capacità $x \leq u$, (MCF) abbia la forma del duale della coppia asimmetrica, mentre (DMCF) abbia la forma del suo primale. Per semplificare la discussione assumeremo inizialmente che tutte le capacità degli archi siano infinite ($u_{ij} = +\infty$) in modo tale che i due problemi abbiano esattamente la forma di (D) e (P), rispettivamente.

Vogliamo quindi esaminare l'applicazione degli algoritmi del Simplesso a (MCF)/(DMCF). Per questo è necessario esaminare la struttura delle *matrici di base* di (MCF). Occorre innanzitutto notare che la matrice di incidenza E non ha rango pieno: detto e il vettore (di dimensione appropriata) i cui elementi sono tutti pari ad 1, è immediato verificare infatti che $eE = 0$, e quindi E ha rango al più $n-1$. Questo giustifica anche il fatto che la condizione $eb = \sum_{i \in N} b_i = 0$ sia necessaria affinchè possa esistere un flusso. Più in generale, possiamo enunciare il seguente Lemma, riguardante la somma di un sottoinsieme dei vincoli (3.1).

Lemma 3.1 *Sia (N', N'') un taglio del grafo (si veda il §B.1.3): si ha*

$$\textstyle\sum_{i \in N''} \left(\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} \right) = \sum_{(i,j) \in A^+(N',N'')} x_{ij} - \sum_{(i,j) \in A^-(N',N'')} x_{ij} \; .$$

Dimostrazione Nella sommatoria a sinistra, i flussi degli archi $(j,i) \in A^+(N', N'')$ appaiono con coefficiente $+1$ in quanto entranti nel nodo $i \in N''$, quelli degli archi $(i,j) \in A^-(N', N'')$ appaiono con coefficiente -1 in quanto uscenti dal nodo $i \in N''$, mentre i flussi degli archi (i,j) i cui estremi appartengono entrambi a N'' appaiono una volta con coefficiente $+1$ ed una con coefficiente -1 e quindi non contribuiscono alla somma. $\diamond$

Il Lemma 3.1 dimostra che il rango di E è al più pari ad n meno il numero di *componenti connesse* di G; infatti, ad ogni componente connessa è associato un taglio (N', N'') tale che $A^+(N', N'') = A^-(N', N'') = \emptyset$. Tuttavia, si può sempre assumere che G sia connesso; infatti, se così non fosse, allora (MCF) potrebbe essere suddiviso in tanti problemi indipendenti quante sono le componenti connesse di G, in quanto non esistono vincoli che "legano" il flusso su un qualsiasi arco di una delle componenti con il flusso su un qualsiasi arco di una diversa componente. Per rendere E di rango pieno è quindi necessario eliminare almeno una delle righe, corrispondente ad uno qualsiasi dei nodi del grafo. Questo corrisponde al fatto che, in (DMCF), una delle variabili y_i può senza perdita di generalità essere fissata ad un valore qualsiasi, ad esempio 0. Infatti, se $[\, \bar{y} \, , \, \bar{w} \,]$ è una soluzione ammissibile di (DMCF), allora lo è anche $[\, \bar{y} + \alpha e \, , \, \bar{w} \,]$ per qualsiasi valore di α, come è immediato verificare dai vincoli (3.4) (alle variabili y_i viene infatti dato il nome di "potenziali" in quanto è solamente la differenza tra il valore associato a coppie di nodi che impatta ai fini dei vincoli (3.4)); inoltre le due soluzioni hanno lo stesso valore della funzione obiettivo in quanto $(\bar{y} + \alpha e)b = \bar{y}b$. Pertanto si può assumere che il vincolo (3.1) corrispondente ad un fissato $i \in N$ sia rimosso da (MCF), e la corrispondente variabile y_i sia rimossa da (DMCF) (o, equivalentemente, fissata a zero). Questo è sufficiente per rendere la matrice E di rango pieno, ovvero $(n-1)$. Per dimostrarlo enunciamo prima un ulteriore lemma, stavolta relativo alla somma di colonne di E.

Lemma 3.2 *Sia C un ciclo (non necessariamente orientato) nel grafo: allora le colonne di E corrispondenti agli archi di C sono linearmente dipendenti.*

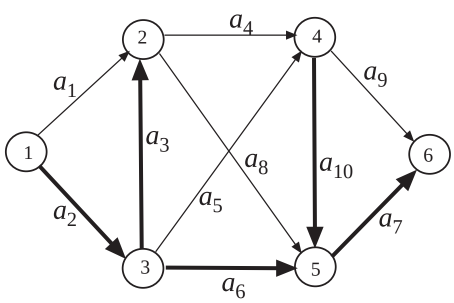
Dimostrazione Si stabilisca un verso arbitrario sul ciclo C, e si dividano gli archi di C nei due sottoinsiemi C^+ degli archi concordi (ossia percorsi nel verso corrispondente alla loro orientazione) e C^- degli archi discordi (ossia percorsi nel verso opposto alla loro orientazione). Sia adesso $\nu_{ij} = 1$ se $(i,j) \in C^+$ e $\nu_{ij} = -1$ se $(i,j) \in C^-$: è facile verificare che

$$\sum_{(i,j)\in C} \nu_{ij} E^{ij} = 0$$

dove E^{ij} indica la colonna di E corrispondente ad $(i,j) \in A$. Infatti, ogni nodo o non è toccato da nessuno degli archi del ciclo, e quindi nei corrispondenti elementi dei vettori E^{ij} si ha il valore zero, oppure è toccato un numero pari di volte. Grazie alla definizione di ν_{ij}, nella sommatoria si trova un $+1$ per metà di queste volte ed un -1 per l'altra metà, e quindi la somma fa zero. $\diamond$

Si consideri adesso un sottoinsieme $B \subseteq A$, e sia E_B la restrizione di E alle sole colonne corrispondenti agli archi in B. Con un piccolo abuso di notazione indicheremo con E_B anche la sottomatrice di E alla quale sia stata eliminata una qualsiasi riga, come precedentemente discusso. Vogliamo caratterizzare gli insiemi B per cui E_B sia di rango pieno: dai Lemmi 3.1 e 3.2 risulta che il sottografo corrispondente a B, che indicheremo con G_B, deve essere connesso (altrimenti le righe sarebbero linearmente dipendenti) ed aciclico (altrimenti le colonne sarebbero linearmente dipendenti), e quindi deve essere un albero di copertura. In effetti si può dimostrare che la condizione è non solo necessaria ma anche sufficiente: qualsiasi sottomatrice E_B corrispondente ad un albero di copertura è invertibile, e quindi ha rango $n-1$. Infatti è facile verificare che E_B può essere ricondotta, tramite scambi di righe e colonne, ad una forma triangolare (superiore) in cui tutti gli elementi diagonali sono non nulli. Per questo si selezioni un qualsiasi nodo, ad esempio $i = 1$, e si visiti l'albero G_B a partire da quel nodo con una visita in ampiezza (cf. §B.3). Si rinominino adesso i nodi (il che equivale a scambiare le righe di E_B) secondo l'ordine in cui sono raggiunti dalla procedura di visita, e si riordinino le colonne di E_B in modo tale che l'ultima colonna corrisponda all'arco entrante nel nodo n (quello tra il predecessore di n ed n), la penultima all'arco entrante nel nodo $n-1$, e così via. Poiché ogni nodo nell'albero (tranne la radice) ha un arco entrante, che proviene da un nodo (dopo la rinumerazione) con un "indice di nome" inferiore, la matrice corrispondente è triangolare superiore con elementi tutti diversi da zero sulla diagonale.

Esempio 3.2: Triangolarizzazione di una matrice E_B
Si consideri il grafo in figura 3.5 ed il suo sottografo G_B corrispondente agli archi (evidenziati nella figura) $a_2 = (1,3)$, $a_3 = (3,2)$, $a_6 = (3,5)$, $a_7 = (5,6)$ ed $a_{10} = (4,5)$. La corrispondente matrice E_B è mostrata al centro. Selezionando il nodo 1 come radice e visitando l'albero G_B si ottiene un ordine di visita 1, 3, 2, 5, 4 e 6. Riordinando le righe di E_B secondo quest'ordine e le colonne come indicato in precedenza, si ottiene la matrice a destra. Eliminando la prima riga (corrispondente al nodo 1) si ottiene infine una matrice 5×5 triangolare superiore.



	a_2	a_3	a_6	a_7	a_{10}
1	-1	0	0	0	0
2	0	1	0	0	0
3	1	-1	-1	0	0
4	0	0	0	0	-1
5	0	0	1	-1	1
6	0	0	0	1	0

	a_2	a_3	a_6	a_{10}	a_7
1	-1	0	0	0	0
3	1	-1	-1	0	0
2	0	1	0	0	0
5	0	0	1	1	-1
4	0	0	0	-1	0
6	0	0	0	0	1

Figura 3.5: Triangolarizzazione di una matrice E_B

Abbiamo quindi mostrato che le basi B di (MCF) (almeno nell'ipotesi in cui le capacità siano pari a infinito) corrispondono agli alberi di copertura non orientati di G. Possiamo inoltre assumere senza perdita di generalità che la corrispondente E_B sia triangolare superiore; siccome il numero di elementi diversi da zero di E_B è esattamente $2n-3$, questo significa che il calcolo delle soluzioni primali e duali di base corrispondenti a B, ossia

$$\bar{x}_B = E_B^{-1} b \qquad , \qquad \bar{y} = c E_B^{-1}$$

possa essere ottenuto in $O(n)$. In effetti è facile verificare che tale calcolo può essere effettuato *durante* la visita di G_B. In particolare, il calcolo di $\bar{y}$ può essere effettuato durante una visita "dalla radice verso le foglie" (pre-visita): fissato arbitrariamente $\bar{y}_1 = 0$, i vincoli duali (presi come uguaglianza) $y_j - y_i = c_{ij}$ permettono di determinare i valori di $\bar{y}_i$ per tutti i figli della radice, e così via fino alle foglie. Viceversa, il calcolo di $\bar{x}_B$ può essere effettuato mediante una visita "dalle foglie verso la radice" (post-visita): dato un nodo foglia i ed il suo predecessore j nella visita, il vincolo (3.1) corrispondente ad i può essere soddisfatto solamente ponendo $x_{ji} = b_i$ se $(j, i) \in B$, e $x_{ij} = -b_i$ se $(i, j) \in B$. Una volta che sia stato calcolato il flusso su tutti gli archi tra un generico nodo i ed i suoi figli si può analogamente determinare il flusso che deve essere presente sull'unico arco che collega il nodo i a suo padre, tenendo conto del verso di tale arco.

Esercizio 3.2 *Si sviluppi lo pseudo-codice di una procedura che determini le soluzioni primali e duali di base corrispondenti ad un dato albero di copertura non orientato G_B, discutendo quali strutture dati siano necessarie affinché tale calcolo possa essere effettuato in $O(n)$.*

Esercizio 3.3 *Si assegnino deficits ai nodi e costi agli archi del grafo in Figura 3.5 e si calcolino la soluzioni primali e duali di base corrispondenti all'albero G_B indicato sia mediante la risoluzione algebrica dei sistemi lineari corrispondenti che mediante la procedura sviluppata nell'esercizio precedente.*

Le considerazioni precedenti possono essere estese anche al calcolo dell'altro elemento critico degli algoritmi del Simplesso, ossia le direzioni di decrescita/crescita. In particolare è possibile verificare che:

- nel Simplesso Duale applicato alla soluzione di (MCF), si individua un arco non presente in B che sia destinato ad entrarvi: tale arco forma con quelli in G_B un ciclo; utilizzare la direzione d determinata dall'algoritmo corrisponde a modificare il flusso lungo tutti e soli gli archi di tale ciclo;

- nel Simplesso Primale applicato alla soluzione di (DMCF), si individua un arco presente in B che sia destinato ad uscire da B: la rimozione di tale arco da G_B determina la separazione dell'albero in due componenti connesse; utilizzare la direzione ξ determinata dall'algoritmo corrisponde a modificare i potenziali (uniformemente) su una delle due rive del taglio così determinato, mantenendo fissi i potenziali sull'altra riva.

Esercizio 3.4 *Si verifichino le affermazioni precedenti.*

Tutto ciò porta allo sviluppo di varianti specializzate degli algoritmi del Simplesso Primale e Duale, nelle quali il costo per calcolare le soluzioni di base ed aggiornarle è $O(n)$ per iterazione; è possibile osservare che la complessità della singola iterazione diviene così $O(m)$ in entrambi i casi, essendo questo il costo per individuare l'arco entrante (prima del calcolo della direzione nel Simplesso Duale, durante il calcolo del massimo passo nel Simplesso Primale). Tutto ciò si estende, in modo non particolarmente complesso ma piuttosto tedioso da sviluppare nel dettaglio, al caso in cui siano presenti vincoli di capacità superiore $x \leq u$ sugli archi. La modifica sostanziale è che, in tale caso, una base corrisponde non ad una bi-partizione dell'insieme degli archi tra quelli in B (che formano un albero di copertura) e quelli non in B (sui quali il flusso è nullo), ma in una *tri-partizione* (B, L, U): gli archi non in base sono infatti ulteriormente suddivisi in L, per i quali $x_{ij} = 0$, e in U per i quali $x_{ij} = u_{ij}$. L'algoritmo precedentemente delineato si generalizza facilmente a questo caso. In effetti, questo tipo di trattazione può essere riservata a qualsiasi problema di PL nel quale (come accade molto spesso in pratica) siano presenti vincoli "di scatola" del tipo $l \leq x \leq u$, dando luogo a quelli che in letteratura sono noti come *algoritmi del Simplesso per Variabili Limitate.*

Esercizio 3.5 *Si descrivano nel dettaglio gli pseudo-codici degli algoritmi del Simplesso Primale e Duale per (MCF) e (DMCF), nelle versioni originarie (senza vincoli di capacità) e rivista (con vincoli di capacità), discutendo quali strutture dati siano necessarie per fare in modo che il costo per iterazione sia $O(m)$.*

È quindi possibile proporre implementazioni efficienti degli algoritmi del Simplesso, sia primale che duale, per (MCF). Questi sono in sostanza algoritmi di ottimizzazione continua, anche se la loro applicazione a (MCF), come abbiamo visto, permette di sostituire "operazioni continue" (inversioni di matrice) con "operazioni combinatorie" (visite di alberi). Come abbiamo precedentemente accennato, nel resto del capitolo presenteremo invece algoritmi di tipo "puramente combinatorio" per (MCF), ossia progettati considerando direttamente le proprietà combinatorie del problema. È opportuno enfatizzare che (MCF) è uno tra i più generali problemi di ottimizzazione su reti che ammettano algoritmi polinomiali. Questa generalità implica però che gli algoritmi risolutivi per (MCF) sono tra i più complessi fra quelli per problemi di ottimizzazione su reti. Nel seguito introdurremo prima i problemi "più facili", per i quali si possono definire algoritmi risolutivi relativamente semplici. Tali algoritmi verranno poi utilizzati all'interno di approcci per (MCF).

3.2 Cammini di costo minimo

Il problema della determinazione di cammini di costo minimo, detti anche *cammini minimi*, è uno tra i più semplici, ma allo stesso tempo tra i più importanti problemi di ottimizzazione su reti. Ad esempio, il calcolo dei percorsi all'interno dei dispositivi GPS, negli smartphones e nei servizi quali Google Maps richiede la soluzione di problemi di cammino minimo su grafi di dimensione molto grande (milioni di nodi ed archi), rappresentanti la rete stradale e/o di trasporto pubblico, in tempi estremamente brevi. Questo però è solamente un esempio di possibile utilizzo, nel quale la formulazione in termini di problema di cammini minimi è una naturale e diretta conseguenza delle caratteristiche della realtà modellata. Come già osservato per (MCF), un problema di cammini minimi può servire anche a formulare e risolvere problemi che apparentemente non hanno alcun rapporto con i grafi e le reti.

Esempio 3.3: Ispezioni su una linea di produzione
Si abbia una linea di produzione con n celle di lavorazione. Ogni lotto è costituito da B pezzi che passano attraverso le n celle ed in ciascuna di esse subiscono una lavorazione. La probabilità di produrre un difetto in un pezzo nella cella i è p_i. Possono essere fatte ispezioni alla fine di ogni lavorazione: le ispezioni vengono fatte su tutti i pezzi del lotto e quelli trovati difettosi vengono scartati. Non essendo accettabile l'invio ai clienti di pezzi difettosi, viene comunque fatta una ispezione alla fine; tuttavia può essere conveniente effettuare ispezioni già dopo le prime lavorazioni in modo da evitare il costo di lavorazioni effettuate su pezzi difettosi e quindi da scartare. Sono dati il costo unitario q_i di lavorazione alla cella i, il costo fisso f_{ij} di ispezione di un lotto all'uscita della cella j nell'ipotesi che la precedente ispezione fosse stata effettuata all'uscita della cella $i(< j)$, ed il costo unitario h_{ij} di una ispezione effettuata all'uscita della cella j, nell'ipotesi che la precedente ispezione fosse stata effettuata all'uscita della cella $i(< j)$. Il numero atteso di pezzi non difettosi alla fine della lavorazione i è dato da

$$B_i = B \prod_{k=1}^{i}(1 - p_k);$$

B_i è il numero di pezzi su cui si effettueranno le lavorazioni nelle celle successive alla cella i, sino a quella in cui si effettuerà una nuova ispezione. Il costo di un'ispezione effettuata alla cella j nell'ipotesi che la precedente sia stata effettuata alla cella $i(< j)$ è dato da $f_{ij} + B_i h_{ij}$; sommando ad esso il costo di lavorazione dei pezzi in tutte le celle da $i+1$ a j comprese, si ottiene il costo globale (produzione e ispezione) nel segmento produttivo da i escluso a j compreso:

$$c_{ij} = f_{ij} + B_i h_{ij} + B_i \sum_{k=i+1}^{j} q_k \quad .$$

Nel seguito con 0 indicheremo una cella fittizia, che precede l'inizio del processo produttivo, affinché siano definiti i valori f_{0j}, h_{0j}, c_{0j} e $B_0 = B$ relativi al caso in cui la prima ispezione sia effettuata nella cella j.

Il problema di determinare il piano di ispezioni ottimo, cioè decidere quando effettuare le ispezioni in modo da minimizzare il costo globale (la somma dei costi di produzione e di quelli di ispezione), può essere formulato come il problema di cercare un cammino di costo minimo dal nodo 0 al nodo n, nel grafo $G = (N, A)$, con $N = \{ 0 , 1 , \dots , n \}$ e $A = \{ (i, j) \, : \, i \in N \setminus \{n\} \, , \, j > i \}$, in cui ad ogni arco (i, j) è associato il costo c_{ij} sopra definito. In figura 3.6 è mostrato il grafo nel caso di $n = 4$. È facile dimostrare che ogni cammino orientato del grafo da 0 a n corrisponde a un piano

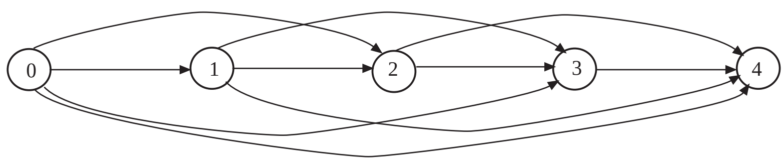


Figura 3.6: Grafo associato al problema delle ispezioni

di ispezione e, viceversa, qualunque piano di ispezione è rappresentato da un cammino orientato da 0 a n, il cui costo (come somma dei costi degli archi) è proprio il costo globale di produzione e ispezione.

3.2.1 Il problema

Sia $G = (N, A)$ un grafo orientato e pesato dove ad ogni arco $(i, j) \in A$ è associato un costo $c_{ij} \in \mathbb{R}$. Per ogni cammino P in G, il costo $C(P) = \sum_{(i,j)\in P} c_{ij}$ è dato dalla somma dei costi degli archi che lo costituiscono. Dati due nodi r e t, definiamo $\mathcal{P}_{rt}$ l'insieme dei cammini (orientati) che connettono r a t. Il *problema del cammino minimo da r a t* ((SP), da *Shortest Path* problem) è quindi

$$(SP) \qquad \min\left\{ C(P) \;:\; P \in \mathcal{P}_{rt} \right\} \; . \tag{3.5}$$

Il problema (3.5) può essere formulato come un particolare problema di flusso di costo minimo su G, in cui gli archi hanno capacità infinita, i costi sono quelli del problema del cammino minimo, r è l'unica sorgente, che produce 1 unità di flusso, t è l'unico pozzo, che richiede 1 unità di flusso, mentre ogni altro nodo è di trasferimento. In altri termini, (3.3) in cui

$$u_{ij} = +\infty \qquad (i,j) \in A \qquad , \qquad b_i = \begin{cases} -1 & \text{se } i = r \\ 1 & \text{se } i = t \\ 0 & \text{altrimenti} \end{cases} \qquad i \in N \; .$$

Infatti, il modo migliore per soddisfare la richiesta di un'unità di flusso da parte di t è inviarla lungo un cammino di costo minimo da r a t. Si noti che, nel caso in cui esistano più cammini aventi uguale lunghezza minima, si possono inviare frazioni dell'unità di flusso lungo cammini diversi; il flusso ottimo può cioè, in questo caso, corrispondere a più di un cammino minimo. Per evitare questo inconveniente si può imporre un *vincolo di integralità sul flusso*, ossia richiedere che $x \in \mathbb{Z}^m$ (il che in effetti significa $x \in \{0, 1\}^m$). Questo normalmente renderebbe il problema "difficile" (cf. il Capitolo 4), ma in questo caso, come vedremo, la condizione viene "naturalmente" soddisfatta dagli algoritmi che svilupperemo.

È possibile che (3.5) sia vuoto: questo accade se e solo se in G non esiste nessun cammino da r a t, il che può essere facilmente verificato mediante una visita del grafo (si veda il §B.3). È anche possibile che (3.5) sia inferiormente illimitato: questo accade se e solo se in G esiste un *ciclo negativo*, cioè un ciclo orientato il cui costo sia negativo, raggiungibile da r e dal quale t sia raggiungibile. Infatti, consideriamo un cammino da r a t che includa un tale ciclo: percorrendo il ciclo si ottiene un cammino *non semplice*, e più volte si percorre il ciclo più si riduce il costo del cammino, mostrando così che il problema non è inferiormente limitato. Vedremo nel seguito che è possibile verificare in tempo polinomiale se un grafo contiene oppure no un ciclo negativo.

Se il grafo non contiene cicli negativi, rimuovendo i cicli da un cammino non semplice si ottiene un cammini semplice non più costoso. Pertanto, esiste sempre una soluzione ottima che è un cammino semplice. In altri termini, *in assenza di cicli negativi (3.5) coincide col problema del cammino semplice di costo minimo* (o cammino semplice minimo). Invece, *in presenza di cicli negativi il problema del cammino semplice minimo è $\mathcal{NP}$-arduo.* È infatti possibile esprimere in questo modo il problema di determinare un *cammino Hamiltoniano minimo* su un grafo, problema che è notoriamente $\mathcal{NP}$-arduo.

Esercizio 3.6 *Si riconduca il problema del commesso viaggiatore (cf. §1.2.2.3) al problema del cammino Hamiltoniano minimo.*

Per questo si definiscano, per lo stesso grafo G, dei nuovi costi $c'_{ij} = c_{ij} - M$ dove M è un numero "molto grande", ad esempio $M = (n-1)c_{max} + 1$ dove c_{max} è il massimo dei valori assoluti dei costi (originali) degli archi. Se esiste un cammino Hamiltoniano da r a t, il cammino semplice di costo minimo da r a t con i nuovi costi c' sarà sicuramente Hamiltoniano: infatti i cammini Hamiltoniani sono i più lunghi possibili (in termini di numero di archi) tra i cammini semplici su un grafo, ed i costi c' sono tali per cui qualsiasi cammino con $k+1$ archi ha un costo inferiore di qualsiasi cammino con k archi. Pertanto, se il cammino semplice di costo minimo individuato su G con costi c' non è Hamiltoniano, allora il problema del cammino Hamiltoniano minimo è vuoto; altrimenti il cammino semplice di costo minimo su G con costi c' corrisponde ad un cammino Hamiltoniano minimo con i costi originari c. Questo è dovuto al fatto che tutti i cammini Hamiltoniani hanno $n-1$ archi, e quindi la differenza del costo di un cammino Hamiltoniano usando i costi c' ed i costi c è la costante $(n-1)M$. Pertanto la modifica dei costi non cambia l'ordinamento dei cammini Hamiltoniano rispetto ai costi originari, e quindi assicura che un cammino semplice minimo sia un cammmino Hamiltoniano minimo (qualora ne esista uno).

Esercizio 3.7 *Si noti che quando non esistono cicli negativi si possono aggiungere vincoli di capacità $x_{ij} \leq 1$ su tutti gli archi senza cambiare la soluzione. Si noti inoltre che aggiungere tali vincoli in presenza di cicli negativi rende il problema limitato. In aggiunta, è possibile utilizzare le trasformazioni viste nel paragrafo 3.1.2 per modificare il grafo in modo tale da imporre il vincolo che per ciascun nodo passi al più un'unità di flusso. Si discuta perché il problema di flusso così ottenuto (che può essere risolto in tempo polinomiale)* non *è una formulazione appropriata del problema del cammino semplice minimo.*

È possibile considerare un problema più generale rispetto a (3.5): data una radice r, determinare in G un cammino di costo minimo da r a i, per ogni $i \neq r$. È facile vedere che il problema può essere formulato come

$$\min \left\{ \textstyle\sum_{i \neq r} C(P_i) \ : \ P_i \in \mathcal{P}_{ri} \quad i \neq r \right\} \ . \tag{3.6}$$

Infatti, la scelta del cammino per un dato nodo i non influenza la scelta del cammino per tutti gli altri nodi; quindi il modo migliore per minimizzare la somma dei costi di tutti i cammini è quella di selezionare per ogni nodo il cammino di costo minimo. Il motivo per cui usualmente si considera (3.6) invece di (3.5) è che, nel caso peggiore, la determinazione di un cammino minimo per un solo nodo destinazione richiede di determinare anche gli altri cammini minimi. Inoltre, in molte applicazioni si debbono calcolare più cammini minimi aventi un'origine comune. Valgono per (3.6) le considerazioni fatte in precedenza per (3.5); in particolare, il problema può essere formulato come un problema di flusso di costo minimo su G in modo analogo, con l'unica differenza che la radice r è la sorgente di $n-1$ unità di flusso, mentre ogni altro nodo i è un nodo pozzo che richiede un'unità di flusso, ossia

$$b_i = \begin{cases} -(n-1) & \text{se } i = r \\ \quad 1 & \text{altrimenti} \end{cases} \qquad i \in N \ .$$

Si noti che la funzione obiettivo è la *somma* delle lunghezze di tutti i cammini P_i, $i \neq r$. Quindi, se nella soluzione ottima un certo arco (i,j) è contenuto in k distinti cammini di costo minimo da r a k diversi nodi, il costo di quell'arco viene conteggiato k volte, il che corrisponde al fatto che nella formulazione di flusso si ha $x_{ij} = k$. Se G è privo di cicli negativi, (3.6) ha una soluzione ottima finita $\{ \ P_i \ , \ i \neq r \ \}$ se e solo se esiste almeno un cammino da r a ciascun altro nodo i. In effetti si può assumere senza perdita di generalità che questo accada: infatti, è sempre possibile aggiungere un arco fittizio (r,i), per ogni $i \neq r$ tale che $(r,i) \notin A$, con costo "elevato" $c_{ri} = M$. Il cammino di costo minimo sarà costituito dal solo arco fittizio (r,i) solo se non esiste alcun cammino da r a i; infatti, se tale cammino esistesse il suo costo sarebbe certamente inferiore a M.

È facile verificare che, tra tutte le soluzioni ottime di (3.6), ne esiste almeno una in cui *l'unione dei cammini P_i forma un albero di copertura per G radicato in r e orientato*, ossia un albero di radice r i cui archi sono orientati da r verso le foglie. Infatti, se P_i è un cammino minimo da r a i e j è un nodo interno al cammino, allora il sottocammino di P_i che arriva sino a j è a sua volta un cammino minimo da r a j. Quindi, se esistono più cammini minimi da r ad un certo nodo i, è possibile selezionarne uno, P_i, ed imporre che i cammini minimi da r verso altri nodi, aventi i come nodo intermedio, abbiano P_i come sottocammino da r ad i. Ogni soluzione ottima di (3.6) che possa essere rappresentata mediante un albero di copertura orientato di radice r è detta un *albero di cammini minimi di radice r*. Nel seguito considereremo quindi la seguente forma equivalente di (3.6): determinare in G un albero di cammini minimi di radice r. Questo viene detto *problema dell'albero di cammini minimi* ((SPT), da Shortest Path Tree), o, più semplicemente, *problema dei cammini minimi*. Si noti che la proprietà che caratterizza le soluzioni ottime di (3.6) in termini di alberi di copertura del grafo può essere derivata direttamente dalle considerazioni espresse nel §3.1.3.

3.2.2 Alberi, etichette e condizioni di ottimo

Sia $T = (N, A_T)$ una soluzione ammissibile per (SPT), ossia un albero di copertura radicato in r e orientato. Verifichiamo se T sia una soluzione ottima. Dobbiamo cioè verificare se, per qualche nodo $i \neq r$, esiste un cammino orientato da r ad i di costo minore di $C(P_i^T)$, dove P_i^T è l'unico cammino da r ad i in T. Per fare questo calcoliamo il costo dei cammini in T: costruiamo quindi un vettore di *etichette dei nodi* $d \in \mathbb{R}^n$ tale che $d_i = C(P_i^T)$, per $i \neq r$, e $d_r = 0$. Il vettore d può essere facilmente determinato per mezzo di una procedura di visita dell'albero a partire dalla radice r. Si noti che, se l'albero contiene l'arco (i,j), allora $d_i + c_{ij} = d_j$; infatti, l'unico cammino da r a j è formato dal sottocammino da r ad i, di costo d_i, e dall'arco (i,j), che ha costo c_{ij}. Dato il vettore delle etichette d corrispondente a T, è possibile verificare se qualche arco $(i,j) \notin A_T$ può essere utilizzato per costruire un cammino da r a j migliore di P_j^T. Infatti, supponiamo che per un qualche arco (i,j) risulti $d_i + c_{ij} < d_j$, e sia h il predecessore di j in T (il nodo immediatamente precedente j nel cammino P_j^T): sostituendo nell'albero l'arco (h,j) con (i,j) si ottiene un nuovo albero T' in cui il nodo j è raggiunto con un cammino di costo inferiore, come mostrato in Figura 3.7. Se invece ciò non accade per alcun arco, allora T è una soluzione ottima per (SPT). Per dimostrarlo utilizziamo il seguente lemma:

Lemma 3.3 *Sia $d \in \mathbb{R}^n$ un vettore di etichette dei nodi che verifica le* condizioni di Bellman

$$d_i + c_{ij} \geq d_j \qquad\qquad (i,j) \in A \tag{3.7}$$

e $d_r = 0$; allora, per ogni $i \neq r$, d_i è una valutazione inferiore del costo del cammino minimo da r a i.

Dimostrazione Sia $P_i = \{ j_1 , j_2 , \ldots , j_k \}$ un qualsiasi cammino, non necessariamente semplice, da r a i (quindi $r = j_1$ e $j_k = i$). Per ipotesi si ha

$$\begin{array}{lcl} d_{j_k} & \leq & d_{j_{k-1}} + c_{j_{k-1}j_k} \\ d_{j_{k-1}} & \leq & d_{j_{k-2}} + c_{j_{k-2}j_{k-1}} \\ \vdots & & \vdots \\ d_{j_2} & \leq & d_{j_1} + c_{j_1 j_2} \end{array} .$$

Sommando membro a membro, e sapendo che $d_{j_1} = d_r = 0$, si ottiene

$$d_{j_k} = d_i \leq C(P_i) = \sum_{\ell=1}^{k-1} c_{j_\ell j_{\ell+1}} \ ,$$

il che, essendo vero per ogni cammino $P_i \in \mathcal{P}_{ri}$, è vero in particolare per il cammino di costo minimo. $\diamond$

Esercizio 3.8 *Le condizioni (3.7) corrispondono alle condizioni di ammissibilità del vettore d per (DMCF), cf. (3.4): si dimostri il Lemma 3.3 utilizzando gli strumenti noti di teoria della dualità.*

Quindi, se il vettore di etichette d corrispondente a T verifica (3.7), allora T è chiaramente ottimo: per il Lemma 3.3, il cammino minimo da r ad un qualsiasi nodo $i \neq r$ non può costare meno di d_i, ed il cammino P_i^T ha esattamente quel costo. Se invece il vettore di etichette d corrispondente a T non verifica (3.7) allora, come abbiamo visto (cf. figura 3.7), T non può essere ottimo. Si ottiene quindi il seguente risultato:

Teorema 3.1 *Sia $T = (N, A_T)$ un albero di copertura radicato in r e orientato, e sia d il corrispondente vettore di etichette: T è un albero dei cammini minimi di radice r se e solo se d verifica le condizioni di Bellman (3.7).*

Si noti che, per verificare l'ottimalità di T, abbiamo associato un'etichetta ad ogni nodo: anche se fossimo interessati solamente al cammino minimo da r ad un dato nodo t, per dimostrarne l'ottimalità attraverso le condizioni di Bellman dovremmo comunque associare un'opportuna etichetta anche a tutti gli altri nodi. Inoltre, un vettore di etichette che rispetta le condizioni di Bellman fornisce informazione sul costo del cammino minimo da r ad ogni altro nodo. Questo giustifica perché, usualmente, venga studiato il più generale problema dell'albero dei cammini minimi.

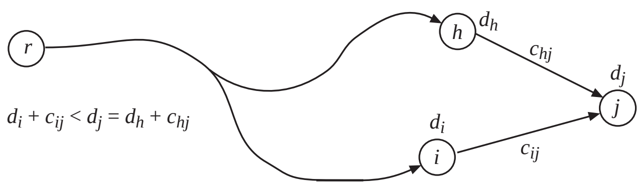


Figura 3.7: un cammino di costo inferiore per j

Se nel grafo esistesse un ciclo orientato di costo negativo, allora non esisterebbe (almeno per alcuni nodi del grafo) nessun limite inferiore al costo dei cammini minimi, e quindi non potrebbe esistere nessun vettore di etichette che rispetti le condizioni di Bellman. Infatti è facile mostrare, utilizzando il Lemma 3.2, che l'esistenza di un ciclo negativo C implica che (DMCF) è vuoto.

Esercizio 3.9 *Si dimostri l'affermazione precedente.*

3.2.3 L'algoritmo *SPT*

Le condizioni di ottimalità presentate nel precedente paragrafo suggeriscono in modo naturale il seguente algoritmo per la determinazione di un albero dei cammini minimi:

- mediante una visita del grafo si determina un albero di copertura radicato in r ed orientato (rappresentato dal vettore $p[\cdot]$) e le relative *etichette* d_i, che rappresentano il costo dell'unico cammino dell'albero da r a i;

- si controlla se esiste un arco $(i,j) \in A$ tale che $d_i + c_{ij} < d_j$; in caso affermativo si modifica l'albero togliendo l'arco $(p[j], j)$ e sostituendovi l'arco (i,j) (ponendo $p[j] = i$), si calcola il vettore delle etichette corrispondente al nuovo albero e si itera; altrimenti l'algoritmo termina avendo dimostrato che l'albero corrente è ottimo (sono verificate le condizioni di Bellman).

Tale algoritmo può essere considerato come una versione del Simplesso Duale applicato a (MCF) nel caso particolare di (SPT). Il costo di ogni iterazione è $O(m)$ per determinare se esiste un arco che viola le condizioni di Bellman, più $O(n)$ per aggiornare le etichette a seguito della sostituzione di $(p[j], j)$ con (i,j); è facile vedere che il ricalcolo va effettuato solo per i nodi nel sottoalbero di radice j, in cui tutte le etichette diminuiscono della stessa quantità $d_j - d_i - c_{ij} > 0$.

Per ottenere un algoritmo più efficiente è possibile integrare le due operazioni e differire l'aggiornamento delle etichette. L'algoritmo mantiene ad ogni iterazione una soluzione ammissibile, rappresentata da un vettore di predecessori $p[\cdot]$, una struttura, che indicheremo con Q, che contiene tutti i nodi i cui archi uscenti potrebbero violare le condizioni di Bellman (3.7), ed un vettore di etichette $d[\cdot]$ in cui $d[i]$, in questo caso, rappresenta in generale un'*approssimazione superiore* del costo dell'unico cammino dell'albero da r a i. All'inizio l'albero è formato da archi fittizi (r,i) aventi costo M molto elevato ($p[i] = r$ e $d[i] = M$ per $i \neq r$).

```
procedure SPT ( G , c , r , p , d )  {
  foreach( i ∈ N ) do { p[i] = r; d[i] = M; }
  d[r] = 0; Q = { r };
  do { select u from Q; Q = Q \ { u };
       foreach( (u, v) ∈ FS(u) ) do
         if( d[u] + c[u, v] < d[v] ) then { d[v] = d[u] + c[u, v]; p[v] = u;
                                           if( v ∉ Q ) then Q = Q ∪ { v };
                                          }
     } while( Q ≠ ∅ );
}
```
Procedura 3.1: Algoritmo *SPT*

L'algoritmo *SPT* controlla se le condizioni di Bellman sono verificate e, ogni volta che trova un arco (i,j) per cui esse sono violate, cioè per cui $d_i + c_{ij} < d_j$, modifica la coppia (p_j, d_j) ponendo $p_j = i$ e $d_j = d_i + c_{ij}$. A seguito della diminuzione di d_j, tutti gli archi uscenti da j possono violare le condizioni di Bellman; j viene detto per questo *nodo candidato*, e viene inserito in Q. Ad ogni iterazione si verifica se Q è vuoto. In questo caso l'algoritmo termina avendo determinato una soluzione ottima; infatti il vettore $d[\cdot]$ rispetta le condizioni di Bellman e contiene le etichette dell'albero T rappresentato dal vettore $p[\cdot]$. Altrimenti, si estrae un nodo u da Q e si controlla se le condizioni (3.7) valgono per ciascun arco della sua stella uscente. Per ogni arco (u, v) che non soddisfa le condizioni, si pone $p[v] = u$ e l'etichetta di v viene aggiornata. Non si effettua però l'aggiornamento delle etichette di tutti i nodi del sottoalbero di radice v, ma si inserisce v in Q: si effettua cioè un "aggiornamento differito" delle etichette.

Esempio 3.4: Esecuzione dell'algoritmo *SPT*
Si voglia determinare un albero dei cammini minimi di radice $r = 1$ sul grafo in figura 3.8(a), applicando l'algoritmo *SPT* in cui Q è implementato come una fila. Esaminando la stella uscente del nodo 1 si pone $p[2] = 1$, $d_2 = 3$, $p[3] = 1$

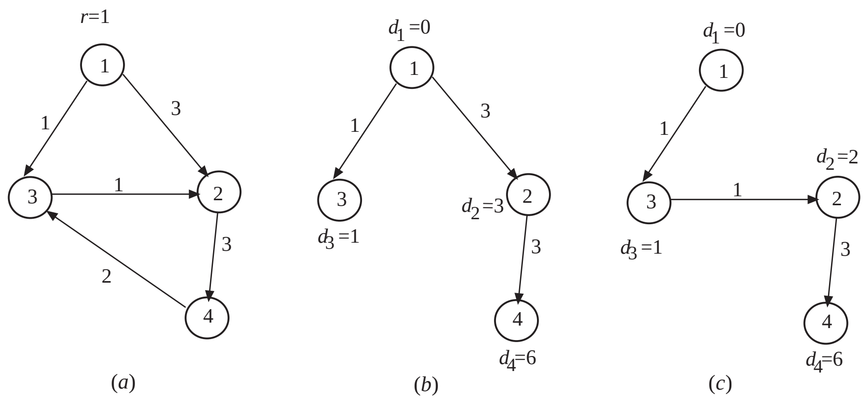


Figura 3.8: Un passo della procedura *SPT*

e $d_3 = 1$; al termine della prima iterazione si ha dunque $Q = \{2, 3\}$. Viene quindi estratto 2 da Q, ed esaminando la sua stella uscente si pone $p[4] = 2$ e $d_4 = 6$, come mostrato in figura 3.8(b), con $Q = \{3, 4\}$. A questo punto si seleziona 3 da Q e si controlla la condizione di Bellman per l'arco $(3, 2)$: $d_3 + c_{32} = 1 + 1 < 3 = d_2$. Si modificano pertanto l'etichetta ($d_2 = d_3 + c_{32} = 2$) e il predecessore del nodo 2 ($p[2] = 3$) e si inserisce nuovamente 2 in Q ottenendo l'albero di figura 3.8(c), con $Q = \{4, 2\}$. Osserviamo che l'etichetta del nodo 4 non viene modificata, e quindi non rappresenta il costo del cammino da 1 a 4 nell'albero rappresentato dal vettore $p[\cdot]$. Solo dopo aver estratto da Q il nodo 4 (senza causare nessun cambiamento), quando verrà selezionato il nodo 2 e controllato l'arco $(2, 4)$, allora d_4 verrà aggiornato e posto al valore 5, diventando così uguale al costo del cammino da 1 a 4 sull'albero; 4 verrà quindi inserita in Q. Si tratta di un esempio di aggiornamento differito delle etichette di un sottoalbero. L'algoritmo termina con la seconda estrazione di 4, senza alcun ulteriore aggiornamento di etichette.

Per dimostrare la terminazione dell'algoritmo *SPT* abbiamo bisogno del seguente risultato:

Teorema 3.2 *Ad ogni passo della procedura* SPT*, il valore d_v dell'etichetta del nodo v, per ogni $v \in N$, rappresenta il costo di un cammino da r a v nel grafo G (oppure è M).*

Dimostrazione La dimostrazione è per induzione sul numero delle iterazioni. La tesi è certamente vera alla prima iterazione, subito dopo la fase di inizializzazione. Assumendo che la tesi sia vera all'iterazione k, verifichiamo che lo sia anche all'iterazione $k + 1$. Sia j un nodo la cui etichetta è migliorata all'iterazione $k + 1$: il nuovo valore sarà $d_j = d_i + c_{ij}$ per qualche nodo i. Poiché l'etichetta di i è stata modificata in un'iterazione precedente, per ipotesi induttiva essa rappresenta il costo di un cammino da r ad i. Pertanto d_j è il costo di un cammino del grafo costituito da un sottocammino da r ad i, di costo d_i, e dall'arco (i, j). $\diamond$

A questo punto possiamo dimostrare che, se il grafo non contiene cicli di costo negativo, la procedura *SPT* termina dopo un numero finito di passi. Infatti, per il Teorema 3.2 il valore d_i dell'etichetta di qualsiasi nodo i è sempre uguale al costo di un cammino del grafo G da r ad i. Osserviamo che il nodo i è inserito in Q solo quando la sua etichetta diminuisce: poiché il numero di cammini semplici da r ad i è finito, d_i può diminuire solamente un numero finito di volte, e quindi il nodo i potrà essere inserito in Q solamente un numero finito di volte. Di conseguenza, dopo un numero finito di iterazioni si avrà $Q = \emptyset$ e la procedura terminerà. Il prossimo esempio mostra che, se invece il grafo contiene un ciclo negativo (e almeno uno dei nodi del ciclo è raggiungibile da r), allora la procedura *SPT non termina.*

Esempio 3.5: Effetto di un ciclo negativo
Si consideri nuovamente il problema di Figura 3.8(a), in cui però l'arco $(4,3)$ abbia costo -5. La prima estrazione di 4 da Q non causa nessun cambiamento in $p[\cdot]$ e $d[\cdot]$, poiché $d_4 + c_{43} = 6 - 5 = 1 = d_3$. Quando però 4 viene estratto da Q la seconda volta, avendo etichetta $d_4 = 5$, ciò causa il rietichettamento di 3 in quanto $d_4 + c_{43} = 5 - 5 = 0 < 1 = d_3$. Si noti che il vettore $p[\cdot]$ non descrive più un albero; infatti si ha $p[3] = 4, p[4] = 2$ e $p[2] = 3$, che individua il ciclo $\{3, 2, 4, 3\}$. Il controllo delle condizioni di Bellman per il nodo 3 causa la diminuzione dell'etichetta di 2 al valore 1, il che causa la diminuzione dell'etichetta di 4 al valore 4, che a sua volta causa la diminuzione dell'etichetta di 3 al valore -1 e così via. In altre parole, i nodi del ciclo vengono inseriti in Q un numero infinito di volte, mentre il valore delle loro etichette diminuisce indefinitamente.

L'algoritmo *SPT* è un algoritmo molto generale il cui effettivo comportamento dipende dal modo con cui viene implementato l'insieme Q dei nodi candidati. In effetti, ad implementazioni diverse corrispondono comportamenti molto diversi in termini di complessità computazionale. Ad alto livello possiamo pensare a due scelte alternative:

1. Q è una *coda di priorità*, cioè un insieme in cui ogni elemento ha associato un valore (chiave), e la scelta dell'elemento da estrarre avviene sulla base di questo valore;

2. Q viene implementato come una *lista* e la scelta dell'elemento da estrarre è determinata dalla posizione dell'elemento nella lista.

Tali scelte corrispondono a strategie implementative diverse, realizzabili in modi molto differenti fra loro: nel seguito discuteremo alcune di queste possibili implementazioni e le conseguenze che esse hanno sull'efficienza dell'algoritmo.

3.2.4 Algoritmi a coda di priorità

L'insieme Q viene implementato come coda di priorità; ad ogni elemento i è cioè associata una chiave di priorità, che nel nostro caso è l'etichetta d_i, e la priorità di i cresce al decrescere di d_i. Le operazioni elementari eseguibili su Q sono:

- inserimento di un elemento con l'etichetta associata,

- modifica (riduzione) dell'etichetta di un elemento di Q,

- selezione dell'elemento con etichetta minima e sua rimozione da Q.

Chiamiamo *SPT.S* (da *Shortest-first*) la versione di *SPT* in cui ad ogni iterazione si estrae da Q un elemento ad etichetta minima. L'operazione "select u from Q;" viene pertanto realizzata come

$$\text{select } u \text{ from } Q \text{ such that } d_u = \min\{ \ d_i \ : \ i \in Q \ \} \ .$$

Vale il seguente Teorema:

Teorema 3.3 [Dijkstra, 1959] *Nel funzionamento di* SPT.S *su grafi con costi non negativi, ogni nodo viene inserito in Q (e rimosso da esso) al più una volta.*

Dimostrazione Indichiamo con u_k e $d^k[\cdot]$ rispettivamente il nodo estratto da Q ed il vettore delle etichette all'iterazione k ($u_1 = r$). Vogliamo innanzitutto dimostrare che la successione dei valori delle etichette dei nodi estratti da Q è non decrescente, ossia che $d^{k+1}[u_{k+1}] \geq d^k[u_k]$, per ogni $k \geq 1$. Per questo basta considerare due casi:

- $d^{k+1}[u_{k+1}] = d^k[u_{k+1}]$, ossia l'etichetta di u_{k+1} non è cambiata durante la k-esima iterazione. In questo caso u_{k+1} apparteneva certamente a Q all'inizio della k-esima iterazione (un nodo può entrare in Q solo se il valore della sua etichetta diminuisce) e quindi $d^{k+1}[u_{k+1}] = d^k[u_{k+1}] \geq d^k[u_k]$ perché u_k è uno dei nodi di Q con etichetta di valore minimo al momento in cui viene estratto.

- $d^{k+1}[u_{k+1}] < d^k[u_{k+1}]$, ossia l'etichetta di u_{k+1} è cambiata durante la k-esima iterazione: questo significa che u_k è il predecessore di u_{k+1} all'inizio della $k+1$-esima iterazione e quindi $d^{k+1}[u_{k+1}] = d^k[u_k] + c_{u_k,u_{k+1}}$, da cui, dato che $c_{u_k,u_{k+1}} \geq 0$, si ottiene ancora una volta $d^{k+1}[u_{k+1}] \geq d^k[u_k]$.

Poiché i nodi vengono inseriti in Q quando il valore della loro etichetta decresce, se un nodo entrasse in Q una seconda volta, la sua etichetta avrebbe un valore inferiore a quello che aveva nel momento della sua prima estrazione. Ciò contraddice quanto appena dimostrato. ◇

Il fatto che un nodo non possa essere inserito in Q, e quindi estratto, più di una volta fa sì che il generico arco (i, j) possa essere esaminato al più una volta, cioè quando viene selezionato il nodo i, e pertanto *SPT.S* ha complessità polinomiale. Inoltre, questa proprietà si dimostra molto utile nel caso in cui si sia in effetti interessati a determinare solamente il cammino minimo da r ad uno specifico nodo t, o comunque verso un sottoinsieme proprio dei nodi. Infatti, è immediato verificare che in questo caso si può far terminare l'algoritmo non appena tutti i nodi che si desidera raggiungere sono stati estratti dalla coda perché, per la proprietà appena mostrata, non appena ciò si verifica tutti i cammini minimi relativi risultano essere stati determinati.

Esercizio 3.10 *Dimostrare, per* SPT.S*, che se i costi degli archi sono non negativi allora, ad ogni iterazione, il valore dell'etichetta di un nodo è il costo del cammino di T che va dall'origine a quel nodo, e non una sua approssimazione superiore.*

Si può dimostrare invece che, nel caso di *costi negativi*, esistono grafi per i quali l'algoritmo esegue un numero esponenziale di iterazioni; per i dettagli si rimanda alla letteratura citata.
Sono possibili diverse implementazioni dell'algoritmo *SPT.S*, che differiscono per il modo in cui è implementata la coda di priorità Q. La scelta dell'implementazione di Q non cambia il comportamento dell'algoritmo (si può pensare che la sequenza di estrazioni da Q sia indipendente da tale scelta), ma ne influenza la complessità e l'efficienza computazionale. Nel seguito discuteremo brevemente alcune possibili implementazioni.

Lista non ordinata

Q è implementata come una lista non ordinata, ad esempio mediante un vettore a puntatori, in cui i nodi vengono inseriti in testa o in coda e la selezione del nodo di etichetta minima viene effettuata per mezzo di una scansione completa della lista. Le operazioni elementari hanno quindi la seguente complessità:

inizializzazione delle etichette e della lista Q:	$O(n)$,
selezione del nodo di etichetta minima:	$O(n)$,
rimozione da Q del nodo di etichetta minima:	$O(1)$,
inserzione di un nodo o modifica della sua etichetta:	$O(1)$.

L'algoritmo risultante è noto come *algoritmo di Dijkstra*: è facile verificare che, su grafi con costi non negativi, questo algoritmo ha complessità $O(n^2)$. Infatti, dal Teorema (3.3) discende che non verranno estratti più di n nodi da Q: ad ogni estrazione del nodo di etichetta minima si scandisce l'intera lista in tempo $O(n)$ e quindi il costo totale delle operazioni di gestione della lista è $O(n^2)$. Siccome ogni nodo viene estratto da Q al più una volta, ogni arco (i, j) viene esaminato, una sola volta, se e quando i viene estratto da Q. Le operazioni sui singoli archi sono effettuate in tempo costante, e quindi costano complessivamente $O(m)$; dato che $m < n^2$, la complessità in tempo dell'algoritmo è $O(n^2)$.

Esercizio 3.11 *In generale un grafo può avere "archi paralleli", ossia più copie dello stesso arco (i, j) con costi diversi; si esamini la complessità dell'algoritmo di Dijkstra in questo caso.*

Esempio 3.6: Esecuzione dell'algoritmo di Dijkstra
Si vuole determinare l'albero dei cammini minimi di radice $r = 1$ sul grafo di figura 3.9(a) con la procedura *SPT.S*, usando per Q una lista non ordinata. L'albero fittizio iniziale è quello di figura 3.9(b). Gli alberi da (b) a (g) sono quelli che si ottengono nell'inizializzazione e come risultato delle iterazioni dell'algoritmo riportate nella seguente tabella. Gli archi disegnati in queste figure corrispondono a quelli indicati dal vettore dei predecessori; quelli tratteggiati corrispondono agli archi fittizi di costo $M = 26$. I valori delle etichette sono indicati accanto ai nodi. Nell'ultima iterazione si seleziona il nodo 6, ma non si hanno modifiche di etichette e quindi l'albero rimane inalterato.

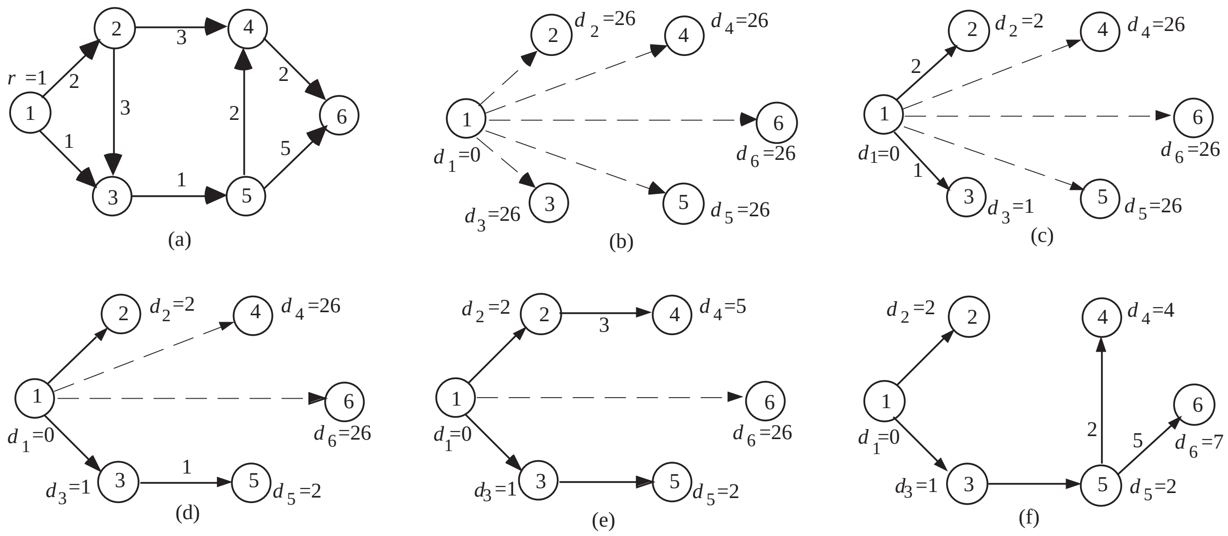
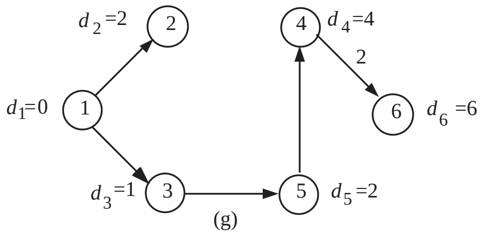


Figura 3.9: Alberi generati dall'Algoritmo *SPT.S*

Iter.	Q	u	Archi esaminati	Etichette modificate	Predecessori modificati	Albero
1	$\{1\}$	1	$(1,2),(1,3)$	d_2, d_3	p_2, p_3	(c)
2	$\{2,3\}$	3	$(3,5)$	d_5	p_5	(d)
3	$\{2,5\}$	2	$(2,3),(2,4)$	d_4	p_4	(e)
4	$\{4,5\}$	5	$(5,4),(5,6)$	d_4, d_6	p_4, p_6	(f)
5	$\{4,6\}$	4	$(4,6)$	d_6	p_6	(g)
6	$\{6\}$	6				(g)



Esercizio 3.12 *Applicare* SPT.S *con Q implementato come lista non ordinata al grafo di figura 3.10 con radice $r = 1$.*

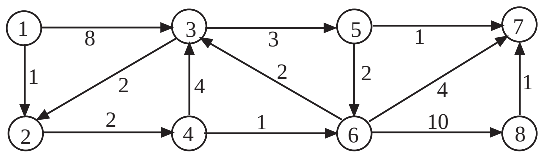


Figura 3.10: Un'istanza del problema (SPT)

Heap binario bilanciato

Come abbiamo visto, la complessità dell'algoritmo di Dijkstra è fondamentalmente dovuta alla gestione dell'insieme Q: per questo, sono state proposte diverse strutture di dati per implementare Q in modo efficiente mantenendo l'insieme Q parzialmente ordinato. Una delle alternative più utilizzate è quella di realizzare Q mediante uno *heap binario blianciato*, in modo che il costo delle operazioni su Q divenga

inizializzazione delle etichette e dello heap Q:	$O(n)$,
selezione del nodo di etichetta minima:	$O(1)$,
rimozione da Q del nodo di etichetta minima:	$O(\log n)$,
inserzione di un nodo o modifica della sua etichetta:	$O(\log n)$.

Se i costi degli archi sono non negativi, le operazioni di ordinamento dello heap a seguito di inserimenti o rimozioni di nodi da Q sono al più $m+n$: pertanto, la versione di *SPT.S* che utilizza un heap binario ha complessità $O(m \log n)$. Si noti che tale complessità è migliore di quella dell'algoritmo di Dijkstra nel caso di grafi *sparsi* ($m \approx n$), mentre è peggiore di quella dell'algoritmo di Dijkstra nel caso di grafi *densi* ($m \approx n^2$). Sono stati proposte molte implementazioni di Q basate su differenti implementazioni di code di priorità, quali ad esempio i *Buckets*, i *Radix Heaps* ed i *Fibonacci Heaps*; per ulteriori dettagli si rinvia alla letteratura citata.

3.2.5 Algoritmi a selezione su lista

In questi algoritmi l'insieme Q viene implementato come una *lista*, cioè una *sequenza* di elementi su cui possono essere effettuate operazioni di rimozione ed inserzione alle estremità della sequenza, chiamate rispettivamente *testa* e *coda* della lista. Si noti che l'aggiornamento dell'etichetta di un nodo che appartiene a Q non influisce sulla posizione dell'elemento nella lista, ossia non causa la rimozione del nodo da Q ed il suo reinserimento in una posizione diversa (formalmente, questo corrisponde al controllo **if**($v \notin Q$) ... nell'algoritmo SPT). Esistono diversi tipi di liste; nel nostro caso, hanno particolare rilevanza

***fila*:** l'inserzione viene effettuata in coda e la rimozione dalla testa (regola *FIFO*);

***pila*:** l'inserzione e la rimozione vengono effettuate in testa (regola *LIFO*);

***deque*:** (double-ended queue, o lista a doppio ingresso) l'inserzione viene effettuata sia in testa che in coda e la rimozione solo dalla testa.

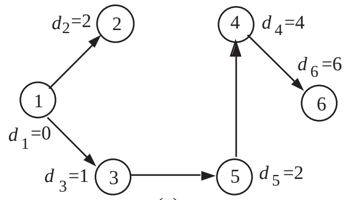
Indichiamo nel seguito con $SPT.L$ le versioni di SPT nelle quali l'insieme Q è implementato come lista. La lista può essere realizzata in diversi modi (lista a puntatori, vettore di puntatori, lineare o circolare, semplice o doppia, ecc.), ed è sempre possibile fare in modo che le operazioni elementari ed il controllo di appartenenza di un elemento alla lista abbiano complessità costante, $O(1)$. La complessità di $SPT.L$, anche nel caso di costi negativi, dipende quindi linearmente dal numero di controlli delle condizioni di Bellman sugli archi uscenti dai nodi estratti da Q.

Fila

Esaminiamo ora l'algoritmo che si ottiene realizzando la lista Q come *fila* (*queue*), conosciuto in letteratura come *algoritmo di Bellman*: l'inserzione dei nodi avviene in coda e la rimozione dalla testa (regola FIFO).

Esempio 3.7: Esecuzione di $SPT.L.Queue$
Si vuole determinare l'albero dei cammini minimi di radice $r = 1$ sul grafo in Figura 3.9(a) con $SPT.L$ e Q implementata come una *fila*. Gli alberi che vengono man mano costruiti sono indicati in Figura 3.11. L'albero fittizio iniziale è in (a). La simbologia nelle figure e nella tabella seguente coincide con quella utilizzata nell'esempio 3.6.

Iter.	Q	u	Archi esaminati	Etichette modificate	Predecessori modificati	Albero
1	$\{1\}$	1	$(1,2),(1,3)$	d_2, d_3	p_2, p_3	(b)
2	$\{2,3\}$	2	$(2,3),(2,4)$	d_4	p_4	(c)
3	$\{3,4\}$	3	$(3,5)$	d_5	p_5	(d)
4	$\{4,5\}$	4	$(4,6)$	d_6	p_6	(e)
5	$\{5,6\}$	5	$(5,4),(5,6)$	d_4	p_4	(f)
6	$\{6,4\}$	6				(f)
7	$\{4\}$	4	$(4,6)$	d_6	p_6	(g)
8	$\{6\}$	6				(g)



(g)

Esercizio 3.13 *Applicare* SPT.L.Queue *al grafo di figura 3.10 con radice* $r = 1$.

L'utilizzo di una strategia FIFO corrisponde ad una "visita a ventaglio" (bfs) del grafo. Dimostreremo ora un'utile conseguenza di questo tipo di visita, ossia che l'algoritmo determina in sequenza, per uno stesso nodo i, i cammini di costo minimo da r ad i con un numero crescente di archi. Per questo dobbiamo definire il concetto di *fasi* dell'algoritmo: la fase 0 corrisponde all'inizializzazione, in cui $Q = \{ r \}$, mentre la generica fase $k+1$ inizia quando viene estratto da Q il primo nodo che vi è stato inserito nella fase k, e termina quando viene estratto da Q l'ultimo nodo che vi è stato inserito nella fase k (quindi la fase 1 coincide con la prima iterazione, al termine della quale sono stati visitati e messi in Q tutti i nodi raggiungibili direttamente da r). Si noti che, per la specifica scelta di Q come fila, i nodi inseriti in Q durante la fase k vengono estratti dopo quelli che erano già presenti in Q all'inizio della fase. Definendo $\bar{d}_i^k$ come la lunghezza minima tra tutte quelle dei cammini da r ad i che contengono al più k archi ($\bar{d}_i^k = +\infty$, o equivalentemente $\bar{d}_i^k = M$, se non ne esiste nessuno), si può dimostrare il seguente risultato:

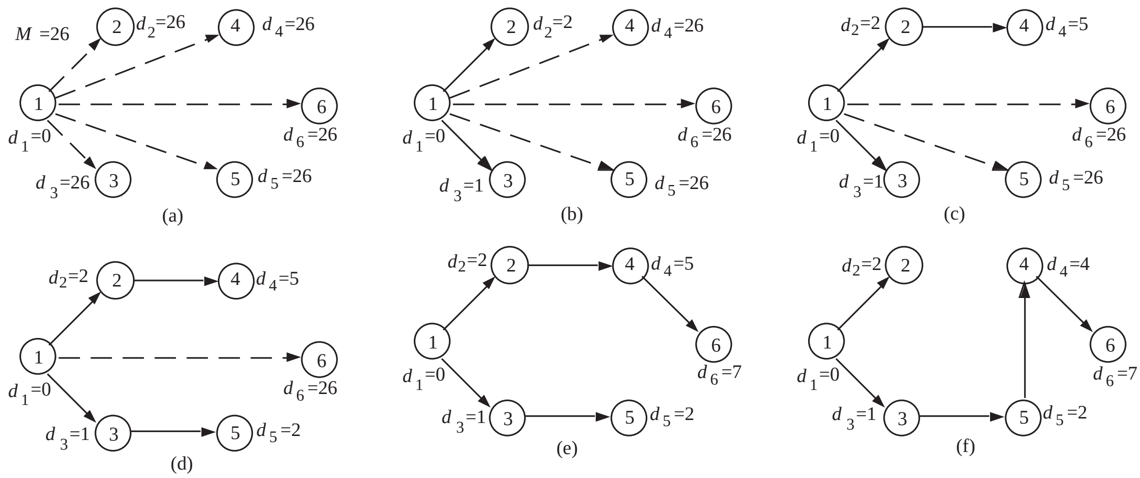


Figura 3.11: Alberi generati da $SPT.L.Queue$

Teorema 3.4 *Sia $d^k[\cdot]$ il vettore delle etichette al termine della fase k: allora $d^k \leq \bar{d}^k$.*

Dimostrazione Il teorema si basa sulla seguente caratterizzazione di $\bar{d}_i^k$, la cui correttezza è facile dimostrare:

$$\bar{d}_j^{k+1} = \min \left\{ \; \bar{d}_j^k \; , \; \{ \; \bar{d}_i^k + c_{ij} \; : \; (i,j) \in A \; \} \; \right\} \tag{3.8}$$

(infatti il cammino minimo con al più $k+1$ archi fino a j o ha in effetti al più k archi, oppure è formato da un arco (i,j) e poi dal cammino minimo con k archi da r ad i). Durante la dimostrazione ci sarà utile assumere che $M = +\infty$, cosa che è sicuramente possibile, per semplificare gli argomenti evitando di distinguere il caso in cui un nodo sia oppure no già stato visitato in precedenza.

Il teorema può adesso essere mostrato per induzione sulle fasi. La proprietà è certamente vera al termine della fase 1, nella quale, come abbiamo già notato, si visitano i nodi raggiungibili direttamente da r, costruendo così tutti e soli i cammini di lunghezza 1, che sono ovviamente minimi. Assumiamo adesso che la proprietà sia vera al termine della fase k, e dimostriamo che allora è vera anche al termine della fase $k+1$. Dobbiamo distinguere due casi: quello dei nodi i che appartengono a Q alla fine della fase k (che indicheremo con Q^k), e quelli che non ci appartengono. Per $i \notin Q^k$ possiamo affermare che

$$d^k[j] \leq d^k[i] + c_{ij} \qquad \forall (i,j) \in A \;\; .$$

Infatti la proprietà vale se i non è mai stato visitato, e quindi $d^k[i] = +\infty$, indipendentemente dal valore di $d^k[j]$; se invece i è in un qualche momento stato in Q e poi ne è uscito, per il funzionamento dell'algoritmo la proprietà valeva nell'ultimo momento in cui i è stato esaminato, ma dopo quel momento $d[i]$ non è più cambiata (altrimenti i sarebbe rientrato in Q) mentre le $d[j]$ possono solo essere diminuite, e quindi la proprietà è ancora valida. Poiché $d^{k+1}[j] \leq d^k[j]$, possiamo concludere che

$$d^{k+1}[j] \leq d^k[i] + c_{ij} \qquad \forall i \notin Q^k \;\; , \;\; (i,j) \in A \;\; .$$

Consideriamo invece $i \in Q^k$; per il funzionamento dell'algoritmo, il nodo verrà estratto da Q in un qualche momento della fase $k+1$, e quindi possiamo affermare che

$$d^{k+1}[j] \leq d^k[i] + c_{ij} \qquad \forall i \in Q^k \;\; , \;\; (i,j) \in A \;\; .$$

Infatti la relazione vale per il valore dell'etichetta di i al momento in cui viene estratto da Q, durante la fase $k+1$, e questo è sicuramente minore o uguale a $d^k[i]$; inoltre $d[j]$ può diminuire ulteriormente nelle iterazioni successive della fase $k+1$. Tutto questo porta a

$$d^{k+1}[j] \leq \min \{ \; d^k[i] + c_{ij} \; : \; (i,j) \in A \; \} \leq \min \{ \; \bar{d}_i^k + c_{ij} \; : \; (i,j) \in A \; \}$$

dove la seconda disuguaglianza deriva dall'ipotesi induttiva $d^k \leq \bar{d}^k$. Poiché si ha inoltre, utilizzando ancora l'ipotesi induttiva, che $d^{k+1}[j] \leq d^k[j] \leq \bar{d}^k[j]$, possiamo finalmente concludere che

$$d^{k+1}[j] \leq \min \left\{ \; \bar{d}_j^k \; , \; \{ \; \bar{d}_i^k + c_{ij} \; : \; (i,j) \in A \; \} \; \right\} = \bar{d}_j^{k+1} \;\; .$$

dove per l'ultima uguaglianza abbiamo usato (3.8). ◇

Il teorema precedente ha alcune interessanti conseguenze. Intanto, interrompendo opportunamente l'algoritmo si possono ottenere valutazioni inferiori sulla lunghezza dei *cammini minimi vincolati*, ossia i cammini minimi soggetti all'ulteriore condizione che il numero di archi del cammino non possa superare una certa soglia prefissata k.

Esercizio 3.14 *Si discuta sotto quali condizioni le valutazioni inferiori sono esatte (suggerimento: si faccia riferimento al Teorema 3.2 ed al Teorema B.1).*

Inoltre, il teorema mostra che, in assenza di cicli negativi, nessun nodo può essere inserito più di $n-1$ volte in Q, e quindi che il numero di volte in cui si esamina un nodo o un arco è limitato superiormente da n. Infatti, per definizione un nodo può uscire da (ed entrare in) Q al massimo una volta per fase, ed in assenza di cicli negativi non possono esserci più di $n-1$ fasi. Da ciò segue che:

- siccome tutte le operazioni sui nodi (estrazione e rimozione da Q) e sugli archi (loro scansione, controllo delle condizioni di Bellman, eventuale aggiornamento di predecessore ed etichetta) sono implementabili in modo da avere complessità costante, la complessità della procedura è dominata dal massimo numero di volte che si esamina lo stesso arco ($n-1$) per il numero di archi (m), e quindi è $O(mn)$;
- *SPT.L.Queue* può essere utilizzata per controllare se un grafo orientato possiede cicli negativi contando il numero di estrazioni da Q di ciascun nodo: appena un nodo viene estratto per l'n-esima volta si può affermare che quel nodo appartiene ad un ciclo negativo, che può poi essere percorso all'indietro, a partire dal nodo trovato, per mezzo del vettore di predecessori $p[\cdot]$.

Esercizio 3.15 *Scrivere la procedura* SPT.L *in cui Q è una fila ed è presente il controllo sui cicli negativi, assicurandosi che tutte le operazioni siano implementate in modo tale che la complessità sia $O(mn)$.*

Liste a doppio ingresso

Nella letteratura scientifica sono state proposte altre realizzazioni dell'insieme Q come lista. Molto utilizzata è la lista a doppio ingresso, o *deque*, in cui i nodi sono inseriti in coda a Q la prima volta, mentre tutte le altre volte vengono inseriti in testa a Q. Si ottiene pertanto una politica ibrida *LIFO–FIFO*; in effetti, la lista Q può essere interpretata come una coppia di liste Q' e Q'' connesse in serie, vedi Figura 3.12. Q' conterrà solo i nodi reinseriti in Q mentre Q'' conterrà solo i nodi inseriti per la prima volta in Q ed ancora non rimossi. Il nodo testa di Q'' viene rimosso solo se $Q' = \emptyset$, pertanto Q' è una *pila* (stack) e Q''è una *fila* (queue).

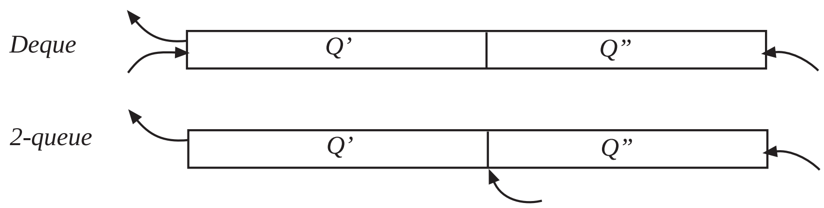


Figura 3.12: Liste a doppio ingresso

Esercizio 3.16 *Applicare SPT.L.deque al grafo di figura 3.10 con radice $r = 1$.*

La motivazione per l'uso di una deque risiede nel fatto che, se un nodo i viene inserito in Q dopo essere stato precedentemente rimosso, la sua etichetta d_i è stata utilizzata per aggiornare etichette di altri nodi, discendenti di i nell'albero corrente. Una nuova inserzione di i in Q avviene poiché d_i è stata diminuita; appare pertanto conveniente correggere quanto prima le etichette dei discendenti di i (sia pure senza ricalcolare immediatamente tutte le etichette del sottoalbero), in modo da evitare il più possibile che vengano compiute iterazioni con valori delle etichette che rappresentano una "cattiva" approssimazione del valore reale del costo del cammino.

La versione di *SPT.L* in cui Q è implementata come *deque* ha però complessità esponenziale $O(n2^n)$; esistono infatti grafi per i quali una tale versione effettua un numero esponenziale di inserimenti e rimozioni di nodi da Q (per ulteriori dettagli si rimanda alla letteratura citata). Comunque, l'analisi della complessità computazionale nel caso peggiore fornisce solo una misura di salvaguardia nella crescita del numero di operazioni: infatti, nei problemi reali in cui le reti sono abbastanza *sparse* ($m \approx n$) questa variante ha un comportamento molto buono, anzi spesso risulta il più efficiente algoritmo per i cammini minimi. Ciò è particolarmente vero per reti stradali, in cui si è osservato sperimentalmente che il numero medio di estrazioni di uno stesso nodo da Q è inferiore a 2. Un'alternativa all'uso della deque consiste nell'implementare anche Q' come fila; per questo basta mantenere un puntatore all'ultimo elemento della porzione Q' di Q ed effettuare gli inserimenti successivi al primo nella coda di Q' (vedi figura 3.12). I nodi verranno così inseriti, la prima volta, in coda a Q" (che coincide con la coda di Q), le altre volte in coda a Q'. La struttura è conosciuta come *doppia coda* (o 2-queue). L'algoritmo risultante risulta sperimentalmente molto efficiente e, dal punto di vista teorico, ha complessità polinomiale: si può infatti dimostrare che il massimo numero di inserimenti dello stesso nodo in Q' è $O(n^2)$, e pertanto la complessità dell'algoritmo è $O(mn^2)$. Sono stati proposte altre implementazioni di Q basate su idee analoghe, ad esempio introducendo una "soglia" (*threshold*) opportunamente calcolata per decidere, in base al valore dell'etichetta d_i, se il nodo i sia da inserire in Q' oppure in Q''; per ulteriori dettagli si rinvia alla letteratura citata.

3.2.6 Cammini minimi su grafi aciclici

Un grafo orientato è detto *aciclico* se non contiene cicli orientati. È immediato verificare che un grafo orientato è aciclico se e solo se è possibile *ben numerare* i suoi nodi, ovvero in modo tale che

$$(i, j) \in A \quad \Longrightarrow \quad i < j \ . \tag{3.9}$$

Il problema di verificare se un dato grafo orientato sia aciclico e, in caso affermativo, di numerare i nodi del grafo in modo da soddisfare la proprietà (3.9), può essere risolto per mezzo di una visita del grafo (si veda il paragrafo B.3) ed ha pertanto complessità $O(m)$.

Esercizio 3.17 *Scrivere una procedura che, in $O(m)$, controlli se un grafo orientato sia aciclico e, in caso positivo, ne numeri i nodi in modo che sia soddisfatta la proprietà (3.9) (suggerimento: se un grafo è aciclico, deve esistere almeno un nodo con stella entrante vuota; eliminando tale nodo e gli archi uscenti da esso, il sottografo indotto risulta a sua volta aciclico).*

Nel seguito è descritta una procedura per il problema della determinazione dell'albero dei cammini minimi, di radice 1, su un grafo aciclico i cui nodi sono stati numerati in accordo alla (3.9); si lascia per esercizio la dimostrazione della sua correttezza e del fatto che la sua complessità è $O(m)$.

```
procedure SPT.Acyclic ( p , d ) {
  foreach( i ∈ N ) do { p[i] = 1; d[i] = M; }; d[1] = 0;
  for( u = 1 ; u < n ; u++ )
    foreach( (u, v) ∈ FS(u) ) do
      if( d[u] + c[u, v] < d[v] ) then { d[v] = d[u] + c[u, v]; p[v] = u; }
}
```

Procedura 3.2: Algoritmo *SPT.Acyclic*

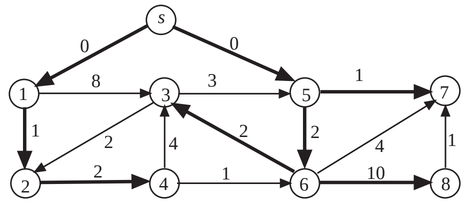
Esercizio 3.18 *Applicare* SPT.Acyclic *al grafo di Figura 3.10 da cui siano stati eliminati gli archi* $(3, 2)$ *e* $(6, 3)$*, con radice* $r = 1$*, dopo aver eventualmente rinumerato i suoi nodi.*

3.2.7 Cammini minimi con radici multiple

In alcuni casi è necessario risolvere il seguente problema: dato un grafo $G = (N, A)$ con costi sugli archi, e dato un insieme non vuoto R di nodi "radice", determinare per ogni nodo $i \notin R$ il cammino minimo da uno dei nodi $r \in R$ ad i. In altre parole si vuole determinare, per ogni nodo i, la "radice" dalla quale sia più conveniente raggiungerlo, ossia la radice alla quale corrisponda il cammino meno costoso fino ad i. È facile verificare che questo problema può essere risolto mediante un'applicazione della procedura *SPT* al grafo $G' = (N', A')$ in cui $N' = N \cup \{ s \}$, dove s è un nodo fittizio che svolge il ruolo di "super-radice", $A' = A \cup \{ (s,r) \,:\, r \in R \}$, e in cui i costi degli archi in $A' \setminus A$ (uscenti da s) sono nulli. L'albero dei cammini minimi su G' fornisce una soluzione ottima al problema dei cammini minimi con radici multiple.

Esempio 3.8: Cammini minimi con radici multiple

Si vogliono determinare i cammini minimi rispetto all'insieme di radici $R = \{1, 5\}$ sul grafo di figura 3.10. Qui accanto è mostrato il corrispondente grafo G' ed il relativo albero dei cammini minimi (archi in grassetto). Quindi, per i nodi 2 e 4 è conveniente selezionare il nodo 1 come radice, mentre per i nodi 3, 6, 7 e 8 è conveniente selezionare il nodo 5 come radice.



Esercizio 3.19 *Si discuta come implementare l'algoritmo suggerito lavorando direttamente sul grafo originale G, senza la necessità di costruire fisicamente G' (suggerimento: si modifichi l'inizializzazione in modo opportuno).*

Esercizio 3.20 *Dato che il grafo G' è sicuramente aciclico se il grafo originale G lo è (s può essere numerato con un indice minore di tutti quelli dei nodi originali, ad esempio 0), si mostri che è possibile risolvere il problema dei cammini minimi con radici multiple su un grafo aciclico in $O(m)$ modificando opportunamente la procedura* SPT.Acyclic.

3.3 Il problema di flusso massimo

Dato il grafo orientato $G = (N, A)$, il vettore $u = [u_{ij}]_{(i,j) \in A}$ di capacità superiori degli archi, e due nodi distinti s e t, detti rispettivamente *origine* (o *sorgente*) e *destinazione* (o *pozzo*), il *problema del flusso massimo* ((MF), da Max Flow problem) consiste nel determinare la massima quantità di flusso che è possibile inviare da s a t attraverso G. Più precisamente, si vuole determinare il massimo valore v per cui ponendo $b_s = -v$, $b_t = v$ e $b_i = 0$ per ogni $i \notin \{s, t\}$ esiste un flusso ammissibile x. Questo significa massimizzare il *valore v del flusso x* che soddisfa

$$\sum_{(j,i) \in BS(i)} x_{ji} - \sum_{(i,j) \in FS(i)} x_{ij} = \begin{cases} -v & i = s \\ v & i = t \\ 0 & \text{altrimenti} \end{cases} \qquad i \in N \;, \tag{3.10}$$

oltre, ovviamente, ai vincoli di capacità (3.2). Il problema di flusso massimo è in realtà un caso particolare del problema di flusso di costo minimo. Infatti, la formulazione (3.10) può essere vista come corrispondente ad un problema (MCF) su un grafo G' ottenuto da G aggiungendo un arco fittizio (t, s), detto *arco di ritorno*, il cui flusso x_{ts} è proprio il valore v: la colonna dei coefficienti relativa alla variabile v, interpretata come una colonna della matrice di incidenza di G', individua proprio l'arco (t, s), come mostrato anche in Figura 3.13.

In G' i nodi sono di trasferimento, compresi s e t, cioè $b = 0$; un tale problema di flusso è detto *di circolazione*. I costi degli archi sono nulli salvo quello dell'arco (t, s) che è posto uguale a -1: di conseguenza, minimizzare $-v$ equivale a massimizzare il valore v del flusso che transita lungo l'arco (t, s), ossia del flusso che in G è inviato da s a t.

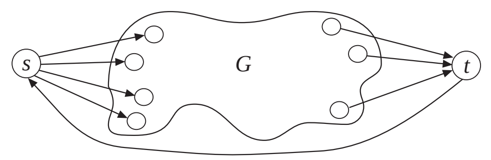


Figura 3.13: Un grafo con l'arco di ritorno (t, s)

Esempio 3.9: Un esempio di flusso ammissibile

Si consideri il grafo in Figura 3.14, in cui la sorgente è $s = 1$ ed il pozzo è $t = 6$. Il vettore x definito sugli archi e riportato in figura è un flusso ammissibile, in quanto sono verificati sia i vincoli di conservazione di flusso che i vincoli di capacità; in particolare, nei nodi 2 e 5 entrano ed escono 7 unità di flusso, mentre nei nodi 3 e 4 ne entrano ed escono 4. Il valore del flusso è $v = 8$, pari alle unità di flusso uscenti da 1 e, equivalentemente, da quelle entranti in 6; se si aggiungesse l'arco di ritorno $(6, 1)$ il suo flusso sarebbe proprio 8. Si noti che non si hanno archi *vuoti*, cioè archi con flusso nullo, mentre gli archi $(1, 2)$, $(3, 5)$ e $(4, 6)$ sono *saturi*, cioè sono archi il cui flusso è uguale alla capacità superiore.

Si osservi quindi che, rispetto al problema di Flusso di Costo Minimo generale, il problema di flusso massimo presenta due importanti caratteristiche: il vettore dei bilanci è nullo, ed il vettore dei osti è nullo tranne che in corrispondenza all'arco fittizio (t, s), in cui è negativo. Come vedremo, queste caratteristiche permettono di sviluppare algoritmi specifici molto efficienti per il problema.

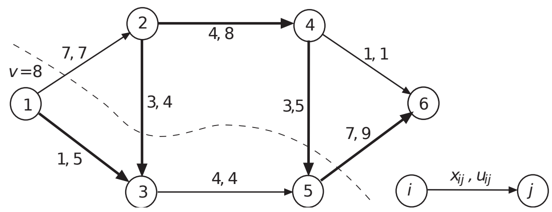


Figura 3.14: un flusso ammissibile x di valore $v = 8$

3.3.1 Tagli, cammini aumentanti e condizioni di ottimo

Come nel caso del problema dei cammini minimi, consideriamo un flusso ammissibile x di valore v, e poniamoci il problema di determinare se x sia oppure no ottimo. Se x non è ottimo, ciò significa che è possibile inviare altro flusso dall'origine alla destinazione; è intuitivo pensare che questo flusso possa essere "instradato" lungo un cammino da s a t. Sia quindi P un cammino, non necessariamente orientato, da s a t: gli archi di P possono essere partizionati nei due insiemi P^+ e P^-, detti *insieme degli archi concordi* ed *insieme degli archi discordi* di P, che contengono rispettivamente gli archi che, andando da s a t, vengono attraversati nel verso del loro orientamento e gli archi attraversati nel verso opposto a quello del loro orientamento. Il cammino P può essere utilizzato per inviare ulteriore flusso da s a t se è possibile modificare il valore del flusso su tutti i suoi archi senza perdere l'ammissibilità ed aumentando il valore v del flusso corrente. È immediato verificare che l'unico modo in cui si può modificare il valore del flusso sugli archi del cammino senza violare i vincoli di conservazione del flusso nei nodi intermedi è aumentare il flusso lungo tutti gli archi concordi, e diminuire il flusso lungo tutti gli archi discordi, di una stessa quantità θ. In altre parole, se x rispetta i vincoli di conservazione del flusso, allora anche $x(\theta) = x \oplus \theta P$ con

$$x_{ij}(\theta) = \begin{cases} x_{ij} + \theta & \text{se } (i,j) \in P^+ \\ x_{ij} - \theta & \text{se } (i,j) \in P^- \\ x_{ij} & \text{altrimenti} \end{cases} \tag{3.11}$$

li rispetta per qualsiasi valore di θ, ed il valore di $x(\theta)$ è $v+\theta$. Tale *operazione di composizione* $\oplus$ tra il flusso x ed il cammino P corrisponde quindi all'invio di θ unità di flusso dall'origine alla destinazione utilizzando il cammino P. Non per tutti i valori di θ, però, l'operazione di composizione produce un flusso ammissibile, in quanto i vincoli (3.2) potrebbero essere violati. La quantità

$$\theta(P, x) = \min\{\ \min\{\ u_{ij} - x_{ij} : (i,j) \in P^+\ \}\ ,\ \min\{\ x_{ij} : (i,j) \in P^-\ \}\ \}\ \ (\geq 0)\ \ , \tag{3.12}$$

detta *capacità del cammino P rispetto al flusso x*, rappresenta la massima quantità di flusso che, aggiunta agli archi concordi di P, non produce flussi maggiori delle rispettive capacità, e sottratta agli archi discordi di P, non produce flussi negativi. Si noti che può essere $\theta(P, x) = 0$: ciò accade se e solo se almeno uno degli archi concordi è *saturo* oppure almeno uno degli archi discordi è *vuoto*. Se invece $\theta(P, x) > 0$, P è detto un *cammino aumentante*, cioè è un cammino lungo il quale può essere inviata una quantità positiva di flusso da s verso t.

Esempio 3.10: Cammini aumentanti
Sia dato il grafo in Figura 3.14, e si consideri il cammino (non orientato) $P = \{1, 3, 2, 4, 5,\ 6\}$, anch'esso mostrato in igura (archi evidenziati). L'insieme degli archi concordi è $P^+ = \{(1,3), (2,4), (4,5), (5,6)\}$ mentre l'insieme degli archi discordi è $P^- = \{(2,3)\}$. La capacità di P rispetto a x è $\theta(P, x) = \min\{\min\{\,5-1\,,\,8-4\,,\,5-3\,,\,9-7\,\}\,,\,\min\{\,3\,\}\,\} = \min\{\,2\,,\,3\,\} = 2 > 0$, e pertanto P è un cammino aumentante rispetto a x: infatti, nessun arco concorde è saturo e nessun arco discorde è vuoto. Possiamo quindi utilizziare P inviare altre due unità di flusso da s a t, ossia costruire il nuovo flusso $x' = x \oplus 2P$; applicando la definizione (3.11) si ottiene $x'_{13} = 1 + 2 = 3$, $x'_{23} = 3 - 2 = 1$, $x'_{24} = 4 + 2 = 6$, $x'_{45} = 3 + 2 = 5$ e $x'_{56} = 7 + 2 = 9$, mentre il flusso su tutti gli altri archi è invariato. È immediato verificare che x' è un flusso ammissibile di valore $v' = 8 + 2 = 10$.

Si pone quindi il problema di definire un algoritmo che, dato un grafo G e un flusso ammissibile x, determini (se esiste) un cammino aumentante P rispetto ad x. A questo scopo si introduce il *grafo residuo* $G_x = (N, A_x)$ rispetto al flusso x, dove

$$A_x = A_x^+ \cup A_x^- = \{ \ (i,j) \ : \ (i,j) \in A \ , \ x_{ij} < u_{ij} \ \} \cup \{ \ (i,j) \ : \ (j,i) \in A \ , \ x_{ji} > 0 \ \} \ .$$

Il grafo residuo, cioè, contiene al più due “rappresentanti” di ciascun arco (i,j) del grafo originale: uno, orientato come (i,j), se (i,j) non è saturo e quindi può appartenere all'insieme degli archi *concordi* di un cammino aumentante, mentre l'altro, orientato in modo opposto ad (i,j), se (i,j) non è vuoto, e quindi può appartenere all'insieme degli archi *discordi* di un cammino aumentante. È immediato verificare che G_x permette di ricondurre il concetto di cicli e cammini aumentanti al più usuale concetto di cicli e cammini orientati:

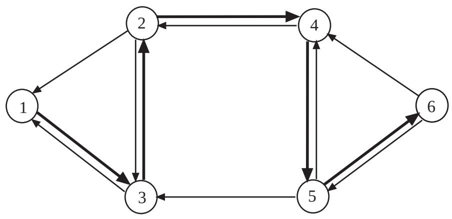


Figura 3.15: il grafo residuo per l'istanza in Figura 3.14

Lemma 3.4 *Per ogni cammino aumentante da s a t rispetto ad x in G esiste uno ed un solo cammino orientato da s a t in G_x.*

Esempio 3.11: Grafo residuo
In Figura 3.15 è mostrato il grafo residuo corrispondente all'istanza ed al flusso x di Figura 3.14. Il cammino orientato $P = \{1, 3, 2, 4, 5, 6\}$ di G_x corrisponde al cammino aumentante di G mostrato nell'esempio precedente.

Un cammino aumentante, se esiste, può quindi essere determinato mediante una visita del grafo residuo G_x a partire da s. Se la visita raggiunge t, allora si è determinato un cammino aumentante (si noti che la visita può essere interrotta non appena questo accada), ed il flusso x non è ottimo perché il cammino aumentante permette di ottenere un nuovo flusso x' di valore strettamente maggiore. Se invece al termine della visita non si è visitato t, allora x è un flusso massimo. Per dimostrarlo introduciamo alcuni ulteriori concetti.

Indichiamo con (N_s, N_t) un taglio di G che separa s da t, cioè un taglio per cui sia $s \in N_s$ e $t \in N_t$, ed indichiamo con $A^+(N_s, N_t)$ ed $A^-(N_s, N_t)$, rispettivamente, l'insieme degli archi diretti e quello degli archi inversi del taglio (si veda l'Appendice B). Dato un flusso x, per ogni taglio (N_s, N_t) definiamo il *flusso del taglio* $x(N_s, N_t)$ e la *capacità del taglio* $u(N_s, N_t)$ come segue:

$$x(N_s, N_t) \ = \ \textstyle\sum_{(i,j) \in A^+(N_s,N_t)} x_{ij} - \sum_{(i,j) \in A^-(N_s,N_t)} x_{ij} \ , \tag{3.13}$$
$$u(N_s, N_t) \ = \ \textstyle\sum_{(i,j) \in A^+(N_s,N_t)} u_{ij} \ . \tag{3.14}$$

Il flusso del taglio è la quantità di flusso che attraversa il taglio (N_s, N_t) da s verso t. Il seguente teorema fornisce la relazione esistente tra il valore del flusso x, ed i flussi e le capacità dei tagli di G.

Teorema 3.5 *Per ogni flusso ammissibile x di valore v e per ogni taglio (N_s, N_t) vale*

$$v = x(N_s, N_t) \leq u(N_s, N_t).$$

Dimostrazione La disuguaglianza deriva da (3.13), (3.14) e dal fatto che $0 \leq x_{ij} \leq u_{ij}$ per ogni $(i,j) \in A$; infatti

$$\textstyle\sum_{(i,j) \in A^+(N_s,N_t)} x_{ij} \leq \sum_{(i,j) \in A^+(N_s,N_t)} u_{ij} \qquad \text{e} \qquad - \sum_{(i,j) \in A^-(N_s,N_t)} x_{ij} \leq 0 \ .$$

L'uguaglianza deriva immediatamente dal Lemma 3.1. $\diamond$

Esempio 3.12: Flusso e capacità di un taglio
Consideriamo il taglio $(N_s, N_t) = (\{1, 3, 5\}, \{2, 4, 6\})$ mostrato in figura; l'insieme degli archi diretti del taglio è $A^+(N_s, N_t) = \{(1, 2), (5, 6)\}$, mentre quello degli archi inversi è $A^-(N_s, N_t) = \{(2, 3), (4, 5)\}$. Il flusso del taglio è $x(N_s, N_t) = x_{12} + x_{56} - x_{23} - x_{45} = 7 + 7 - 3 - 3 = 8 = v$, mentre la capacità del taglio è $u(N_s, N_t) = u_{12} + u_{56} = 7 + 9 = 16$.

Esercizio 3.21 *Cercare, se esiste, un taglio (N_s, N_t) nel grafo in Figura 3.14 avente una capacità inferiore a 16.*

Esercizio 3.22 *Si consideri il taglio* $(N_s, N_t) = (\{1,2,5\},\{3,4,6\})$ *per il grafo in Figura 3.14: si forniscano gli insiemi degli archi diretti e inversi del taglio, e si calcolino il flusso e la capacità del taglio.*

Esercizio 3.23 *Ripetere l'esercizio precedente per il taglio* $(N_s, N_t) = (\{1,\ 4,5\},\{2,3,6\})$.

Il Teorema 3.5 mostra che, comunque si prenda un taglio che separa t da s, il valore del flusso massimo non può eccedere la capacità di tale taglio. Di conseguenza, possiamo dimostrare che x è un flusso massimo se determiniamo un taglio (N_s, N_t) la cui capacità sia uguale a v. Un taglio di questo tipo è in effetti determinato dalla visita del grafo residuo G_x nel caso in cui t non viene raggiunto. Sia infatti N_s l'insieme dei nodi visitati a partire da s, e $N_t = N \setminus N_s$: tutti gli archi di $A^+(N_s, N_t)$ sono saturi, altrimenti l'algoritmo avrebbe potuto visitare un ulteriore nodo, e analogamente, tutti gli archi di $A^-(N_s, N_t)$ sono vuoti. Dal Teorema 3.5 si ha allora $v = x(N_s, N_t) = u(N_s, N_t)$, ossia il flusso e la capacità del taglio coincidono: di conseguenza, x è un flusso massimo. Un'ulteriore importante conseguenza di questa relazione è che (N_s, N_t) è un *taglio di capacità minima* tra tutti i tagli del grafo che separano s da t. Abbiamo quindi dimostrato il seguente teorema:

Teorema 3.6 *(Flusso Massimo-Taglio Minimo) Il massimo valore dei flussi ammissibili su G è uguale alla minima delle capacità dei tagli di G che separano s da t.*

Esercizio 3.24 *I Teoremi 3.5 e 3.6 sono casi speciali, rispettivamente, del Teorema debole della dualità e del Teorema forte della dualità. Si dimostrino tali teoremi in questo modo, derivando il duale di (MF) ed interpretandone le sue soluzioni in termini di tagli* (N_s, N_t).

3.3.2 Algoritmo per cammini aumentanti

Le proprietà enunciate nel precedente paragrafo permettono di progettare un algoritmo per la determinazione di un flusso massimo: partendo da un flusso nullo, si determina ad ogni passo un cammino aumentante, incrementando il valore del flusso, e ci si ferma quando non esiste più alcun cammino aumentante, restituendo quindi anche un taglio di capacità minima.

```
procedure Cammini-Aumentanti ( G , u , s , t , x , p )  {
  x = 0;
  while Trova-Cammino ( G , s , t , x , u , p , θ ) do
    Aumenta-Flusso( x , p , θ );
  }
```

Procedura 3.3: Algoritmo basato su cammini aumentanti

La procedura *Trova-Cammino* cerca di determinare un cammino aumentante da s a t, dato il flusso x. Se il cammino esiste *Trova-Cammino* restituisce *vero* e fornisce il cammino P, attraverso il vettore $p[\cdot]$ dei predecessori dei nodi nell'albero della visita, e la sua capacità $\theta = \theta(P, x)$; altrimenti restituisce $false$, ed in questo caso il vettore $p[\cdot]$ fornisce un taglio (N_s, N_t) di capacità minima (N_t sono tutti i nodi con predecessore 0, N_s gli altri). La procedura *Aumenta-Flusso* aggiorna il flusso x inviando lungo il cammino P la quantità di flusso θ, ossia implementa l'operazione di composizione $x = x \oplus \theta P$. La procedura *Trova-Cammino* è essenzialmente una visita del grafo residuo G_x a partire dall'origine s; è possibile evitare di costruire una rappresentazione di G_x modificando opportunamente la procedura *Visita* in modo che possa lavorare direttamente sulle strutture dati che descrivono il grafo originario G (si veda il §B.3.2 per altri esempi). Inoltre, è facile implementare *Trova-Cammino* in modo tale che, contemporaneamente al cammino, determini anche la sua capacità, memorizzando per ciascun nodo j, raggiunto nella visita, la capacità $d(j)$ dell'unico cammino da s a j nell'albero determinato fino a quel momento. Infatti, si supponga di visitare il nodo j provenendo dal nodo i: se si è usato l'arco (i, j) allora si ha che $d(j) = \min\{\, d(i)\, ,\, u_{ij} - x_{ij}\,\}$, mentre se si è usato l'arco (j, i) allora si ha che $d(j) = \min\{\, d(i)\, ,\, x_{ji}\,\}$; per inizializzare la procedura si pone $d(s) = +\infty$. La procedura *Aumenta-Flusso* può essere implementata percorrendo il cammino in senso inverso da t a s, per mezzo della funzione predecessore $p[\cdot]$, e modificando il flusso in accordo alla (3.11).

Esercizio 3.25 *La funzione predecessore della procedura di visita su G_x non distingue l'orientamento degli archi del grafo originale: in altre parole, $p[j] = i$ non distingue se $(i, j) \in A_x^+$ oppure $(i, j) \in A_x^-$, ossia se si è usato l'arco (i, j) oppure l'arco (j, i) del grafo originario per raggiungere j. Si proponga una soluzione a questo problema. Si discuta inoltre come trattare il caso in cui siano presenti nel grafo archi "paralleli", ossia più copie dell'arco (i, j), con capacità diversa.*

Esercizio 3.26 *Si reinterpreti l'algoritmo* Cammini-Aumentanti *come un caso particolare dell'algoritmo del Simplesso per (MCF).*

La terminazione dell'algoritmo è facile da mostrare sotto un'opportuna ipotesi relativa alle capacità:

Teorema 3.7 *Se le capacità degli archi sono numeri interi, allora esiste almeno un flusso massimo intero, e l'algoritmo* Cammini-Aumentanti *ne determina uno in un numero finito di iterazioni.*

Dimostrazione È facile verificare che, nel caso in cui le capacità siano intere, tutti i flussi x determinati dall'algoritmo sono interi: infatti lo è il flusso alla prima iterazione, e se all'inizio di un'iterazione x è intero, allora θ è un valore intero, e quindi anche $x(\theta)$ sarà un flusso a valori interi. Di conseguenza, ad ogni iterazione (a parte l'ultima) il valore del flusso viene aumentato di almeno un'unità; poiché il valore del flusso massimo è finito, necessariamente l'algoritmo deve terminare in un numero finito di iterazioni determinando la soluzione ottima del problema. $\diamond$

Il Teorema 3.7 fornisce immediatamente una valutazione di complessità $O(mnU)$ per l'algoritmo, con $U = \max\{\, u_{ij} \,:\, (i, j) \in A \,\}$. Infatti nU è maggiore della capacità del taglio $(\{\, s \,\}, N \setminus \{\, s \,\})$, e pertanto anche della capacità minima dei tagli, e di conseguenza anche del massimo valore del flusso. Pertanto, il numero di iterazioni è al più nU, e poiché ogni iterazione costa $O(m)$, segue che la complessità dell'algoritmo è $O(mnU)$. Osserviamo che questa è una complessità *pseudopolinomiale*, essendo U uno dei dati numerici presenti nell'input del problema; ciò significa che anche istanze su grafi "di piccole dimensioni" possono richiedere molte iterazioni se le capacità sono "grandi".

Esercizio 3.27 *Si costruisca un'istanza di (MF) con al più* 4 *nodi e* 5 *archi per la quale l'algoritmo* Cammini-Aumentanti *richieda effettivamente* $\Theta(U)$ *iterazioni.*

Non solo, qualora le capacità non siano numeri interi l'algoritmo *Cammini-Aumentanti* non è né corretto né completo. Infatti è possibile costruire istanze, in cui alcune capacità sono numeri irrazionali, per cui la successione dei flussi x costruiti dall'algoritmo è infinita, e ancor peggio il valore del flusso converge ad un valore strettamente inferiore a quello del valore del flusso massimo (per i dettagli si rimanda alla letteratura citata). Ciò non è di grande rilevanza pratica (i numeri rappresentati su un computer digitale sono tipicamente interi o razionali), comunque entrambi i problemi possono essere risolti scegliendo in maniera opportuna il modo in cui viene determinato il cammino aumentante, ossia come è implementato l'insieme Q nella procedura *Trova-Cammino*. Infatti, mentre i risultati precedenti non dipendono in alcun modo da questa scelta, si possono dimostrare proprietà specifiche qualora, ad esempio, Q sia una *fila*, ossia si realizzi una *visita a ventaglio* del grafo residuo. Ciò permette di visitare ogni nodo mediante il cammino (aumentante) *più corto*, formato cioè dal minimo numero di archi (cf. il Teorema B.1). Ovvero si privilegiano inizialmente cammini aumentanti "corti", aumentando nelle iterazioni successive la loro lunghezza, ed arrivando eventualmente solo nelle ultime iterazioni ad utilizzare cammini aumentanti che passano attraverso tutti (o quasi) i nodi. L'algoritmo risultante prende il nome di *algoritmo di Edmonds & Karp*.

Esempio 3.13: Esecuzione dell'algoritmo di Edmonds & Karp
Un esempio di esecuzione dell'algoritmo di Edmonds & Karp è mostrato in Figura 3.16. Per ogni iterazione sono mostrati lo stato iniziale del flusso ed il cammino aumentante selezionato (archi in grassetto); nell'ultima iterazione è mostrato il taglio di capacità minima determinato dall'algoritmo.

Per l'algoritmo di Edmonds & Karp è possibile dimostrare una valutazione della complessità migliore di quella generale. Per dimostrarlo si consideri il generico flusso x^k determinato alla k-esima iterazione, il corrispondente grafo residuo G^k ed il cammino orientato p^k su G^k utilizzato per ottenere

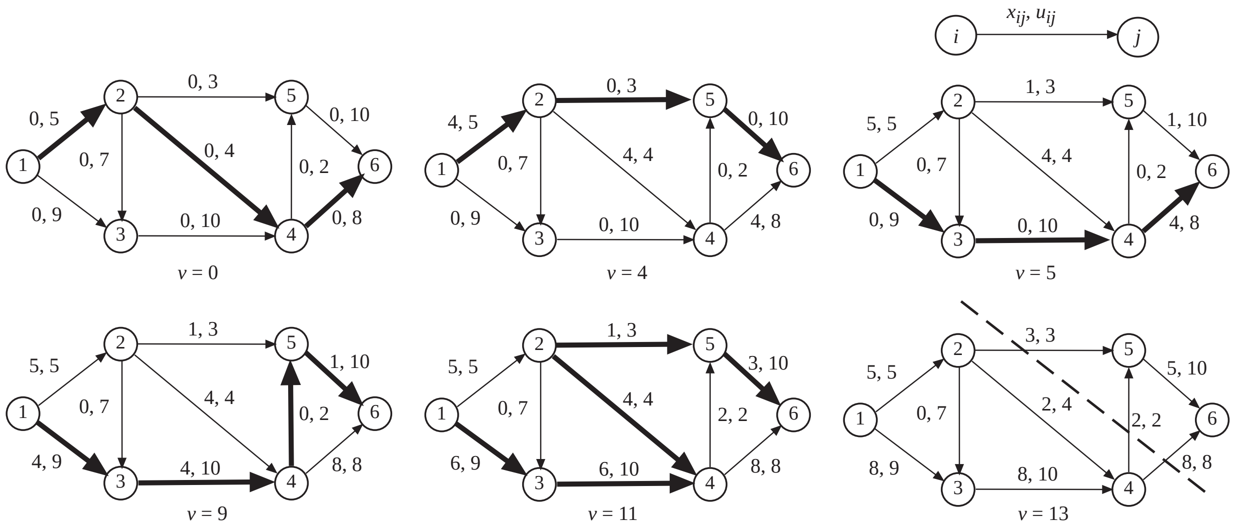


Figura 3.16: Esecuzione dell'algoritmo di Edmonds & Karp

x^{k+1}. Indichiamo con $|p|$ il numero di archi di un generico cammino p, e con $\delta^k(i,j)$ la lunghezza del cammino minimo, in termini di numero di archi, da i a j su G^k (ponendo $\delta^k(i,j) = \infty$ se j non è raggiungibile da i). Vogliamo dimostrare che i cammini aumentanti "corti", per via della visita a ventaglio di G^k, sono utilizzati prima di quelli "lunghi".

Lemma 3.5 *Per ogni i e k vale*

$$\delta^k(s,i) \leq \delta^{k+1}(s,i) \qquad e \qquad \delta^k(i,t) \leq \delta^{k+1}(i,t) \ , \tag{3.15}$$

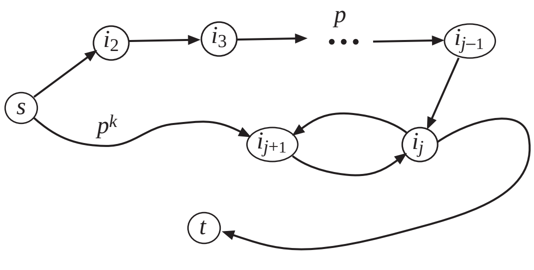
e quindi, in particolare, $\delta^k(s,t) = |p^k|$ *è non decrescente in* k.

Dimostrazione Dimostriamo la disuguaglianza a sinistra in (3.15), perché l'altra segue in modo del tutto analogo. Per questo si fissino arbitrariamente i e k: se $\delta^{k+1}(s,i) = \infty$ non c'è niente da dimostrare, per cui si assuma $\delta^{k+1}(i,j) < \infty$. Sia adesso $p = \{ i_1 , i_2 , \ldots , i_h , i_{h+1} \}$ uno qualsiasi dei cammini di lunghezza minima da s ad i in G^{k+1} (quindi $s = i_1$, $i = i_{h+1}$, $|p| = h = \delta^{k+1}(s,i)$): vogliamo mostrare che vale

$$\delta^k(s, i_{j+1}) \leq \delta^k(s, i_j) + 1 \qquad j = 1, \ldots, h \ . \tag{3.16}$$

Per questo si consideri un fissato j, ed il corrispondente arco $(i_j, i_{j+1}) \in p$: se $(i_j, i_{j+1}) \in G^k$, allora (3.16) è verificata perché l'arco permette di costruire un cammino da s ad i_{j+1} a partire da un qualsiasi cammino da s ad i_j.

Se invece $(i_j, i_{j+1}) \notin G^k$, poiché $(i_j, i_{j+1}) \in G^{k+1}$ allora possiamo sicuramente concludere che $(i_{j+1}, i_j) \in p^k$: l'arco non esisteva nel grafo residuo all'iterazione k ma esiste alla successiva, quindi o era saturo ed è stato parzialmente vuotato o viceversa, in ogni caso è stato usato "in direzione inversa" durante l'iterazione, come mostrato in figura qui accanto. Ma p^k è uno dei cammini di lunghezza minima su G^k, il che significa che $\delta^k(s, i_j) = \delta^k(s, i_{j+1}) + 1$, e quindi $\delta^k(s, i_{j+1}) = \delta^k(s, i_j) - 1 < \delta^k(s, i_j) + 1$, confermando anche in questo caso (3.16).



A questo punto, sommando (3.16) per tutti i valori di $j = 1, \ldots, h$, usando $s = i_1$, $i = i_{h+1}$ e $\delta^k(s,s) = 0$ si ottiene $\delta^k(s,i) \leq h = \delta^{k+1}(s,i)$, ossia (3.15). ◇

Da questa proprietà deriva il seguente teorema:

Teorema 3.8 *L'algoritmo di Edmonds & Karp ha complessità* $O(m^2 n)$.

Dimostrazione Definiamo *bottleneck* un arco che viene saturato o vuotato in una data iterazione dell'algoritmo. Si consideri un arco (i,j) che è bottleneck per due iterazioni $h < l$; ovviamente $(i,j) \in p^h$ e $(i,j) \in p^l$, ma è anche ovvio che deve risultare $(j,i) \in p^k$ per qualche $h < k < l$ (se l'arco è bottleneck all'iterazione h non appare più in G^k per $k > h$ finché non viene utilizzato "al contrario" in un cammino aumentante). Possiamo allora dimostrare che $|p^k| \geq |p^h| + 2$.

Infatti, poiché $(i,j) \in p^h$ abbiamo $|p^h| = \delta^h(s,i) + 1 + \delta^h(j,t)$; in più, $\delta^h(s,j) = \delta^h(s,i) + 1$ (il cammino minimo tra s e j passa per i), ed analogamente $\delta^h(i,t) = \delta^h(j,t) + 1$. Inoltre da $(j,i) \in p^k$ abbiamo

$$|p^k| = \delta^k(s,j) + 1 + \delta^k(i,t) \geq \delta^h(s,j) + 1 + \delta^h(i,t) = (\,\delta^h(s,i) + 1\,) + 1 + (\,\delta^h(j,t) + 1\,) = |p^h| + 2$$

dove per la prima disuguaglianza abbiamo usato (3.15). In altri termini, ogni arco non può essere bottleneck più di $n/2$ volte, e siccome ad ogni iterazione (ciascuna delle quali costa $O(m)$) almeno un arco è bottleneck non possono essere fatte più di $O(mn)$ iterazioni, ad un costo totale di $O(m^2n)$. ◇

Si noti che questo risultato vale anche nel caso in cui le capacità non siano intere.

Esercizio 3.28 *Partendo dal flusso $x = 0$, determinare il flusso massimo da 1 a 6 sul grafo in Figura 3.14, dove la capacità dell'arco $(4,6)$ è $u_{46} = 5$, utilizzando l'algoritmo di Edmonds & Karp. Fornire per ogni iterazione l'albero della visita, il cammino aumentante, la sua capacità, il flusso e il suo valore. Fornire al termine il taglio di capacità minima.*

3.3.3 Flusso massimo con più sorgenti/pozzi

Una generalizzazione del problema di flusso massimo è quella in cui si ha un insieme S di nodi sorgente ed un insieme T di nodi pozzo (entrambi non vuoti), e si vuole individuare il massimo flusso che può essere spedito dai nodi sorgente ai nodi pozzo. Questo problema può essere facilmente risolto applicando un qualunque algoritmo per il flusso massimo ad un grafo ampliato $G' = (N', A')$ con $N' = N \cup \{s,t\}$ e $A' = A \cup \{\,(s,j) \,:\, j \in S\,\} \cup \{\,(i,t) \,:\, i \in T\,\}$. I nodi s e t sono una "super-sorgente" ed un "super-pozzo", collegati rispettivamente a tutti i nodi in S e T con archi a capacità infinita. È possibile modificare l'algoritmo *Cammini-Aumentanti* in modo tale che risolva direttamente il problema più generale senza la necessità di costruire esplicitamente il grafo ampliato G'. Per questo è sufficiente che la procedura *Trova-Cammino* implementi una visita a partire dall'insieme di nodi S invece che dal singolo nodo s, ossia inizializzando $Q = S$, (si veda il paragrafo B.3.2); la visita è interrotta non appena si raggiunga un qualsiasi nodo in T, nel qual caso si è determinato un cammino aumentante da una delle sorgenti ad uno dei pozzi, oppure quando Q è vuoto, e quindi si è determinato un taglio saturo che separa *tutte* le sorgenti da *tutti* i pozzi.
Un'ulteriore generalizzazione del problema è quella in cui ciascuna sorgente $i \in S$ e ciascun pozzo $i \in T$ ha una *capacità finita* u_i, ossia una massima quantità di flusso che può immettere nella/prelevare dalla rete. Anche questo problema può essere risolto applicando un algoritmo per il flusso massimo alla rete ampliata G', con l'unica differenza che la capacità degli archi (s,i) e (i,t) viene posta a u_i e non a $+\infty$. Anche in questo caso è possibile modificare l'algoritmo per cammini aumentanti in modo tale che risolva la versione più generale del problema senza costruire esplicitamente la rete G'.

Esercizio 3.29 *Si fornisca una descrizione formale, in pseudo-codice, dell'algoritmo per risolvere il problema del flusso massimo da un insieme di sorgenti S ad un insieme di destinazioni T, nei casi con e senza capacità associate alle sorgenti/destinazioni. Nel secondo caso, si presti attenzione alla necessità di mantenere traccia del valore x_i (il flusso sugli archi (s,i) e (i,t)) del flusso già immesso dalla sorgente/prelevato dal pozzo i nel flusso corrente x, mantenere gli insiemi S_x e T_x delle sorgenti/pozzi che possono ancora immettere/prelevare flusso (tali che gli archi $(s,i)/(i,t)$ non sono saturi) dai quali iniziare/terminare la visita, ed utilizzare opportunamente le x_i durante il calcolo della capacità dei cammini.*

Il problema appena introdotto è particolarmente interessante anche in quanto esso coincide con il problema di determinare se esiste oppure no una soluzione ammissibile per (MCF).

Esercizio 3.30 *Si dimostri l'affermazione precedente (suggerimento: si ricordi la Figura 3.4); si determini poi un flusso ammissibile per la rete in Figura 3.1 utilizzando l'algoritmo dell'esercizio precedente.*

3.3.4 Algoritmo basato su preflussi

Un limite degli algoritmi basati su cammini aumentanti è il fatto che, ad ogni iterazione, nel caso peggiore può essere necessario esplorare tutto il grafo, senza alcuna garanzia di determinare un cammino aumentante lungo il quale sia possibile inviare una consistente quantità di flusso. Un approccio alternativo è basato sul concetto di *preflusso*, ossia di un vettore x che rispetta (3.2) e

$$e_i = \sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} \geq 0 \qquad i \in N \setminus \{s,t\} \ .$$

Quindi, in un preflusso la quantità di flusso che arriva ad un nodo può essere maggiore di quella che ne esce. Un nodo i viene detto *attivo* se il suo *eccesso* e_i è positivo, altrimenti ($e_i = 0$) viene detto *bilanciato*. Per semplificare la presentazione dell'algoritmo e delle sue proprietà faremo costantemente riferimento al *grafo aumentato* $G' = (N, A' = A^+ \cup A^-)$ che contiene la coppia di archi $(i,j) \in A^+$ e $(j,i) \in A^-$ (detti "gemelli") per ciascun arco $(i,j) \in A$. Su G' si definisce l'immagine x' di un qualsiasi flusso x ponendo $x'_{ij} = x_{ij}$ per $(i,j) \in A^+$, $x'_{ij} = u_{ji} - x_{ji}$ per $(i,j) \in A^-$; si pone inoltre $u'_{ij} = u_{ij}$ per $(i,j) \in A^+$, $u'_{ij} = u_{ji}$ per $(i,j) \in A^-$. È quindi ovvio che il grafo residuo G_x rispetto al preflusso x, definito come abbiamo già visto per il caso dei flussi, coincide col sottografo di G' che contiene i soli archi non saturi, ossia $x'_{ij} < u'_{ij}$. L'aggiornamento del flusso su un arco comporta quello sul suo "gemello": se $x'_{ij} = x'_{ij} + \theta$ per $(i,j) \in A^+$ ($x_{ij} = x_{ij} + \theta$ per $(i,j) \in A$) allora $x'_{ji} = x_{ji} - \theta$ per $(j,i) \in A^-$, e viceversa se $x'_{ij} = x'_{ij} + \theta$ per $(i,j) \in A^-$ ($x_{ji} = x_{ji} - \theta$ per $(j,i) \in A$) allora $x'_{ji} = x_{ji} - \theta$ per $(j,i) \in A^+$. In questo modo l'aggiornamento degli eccessi avviene in modo uniforme nei due casi: $x'_{ij} = x'_{ij} + \theta$ per $(i,j) \in A'$ implica $e_i = e_i - \theta$ ed $e_j = e_j + \theta$.

L'idea che sta alla base dell'algoritmo basato su preflussi è di cercare di spingere flusso verso la destinazione, usando ogni volta solo informazione locale, ossia relativa al nodo in esame e a quelli ad esso adiacenti. A questo scopo si definisce, per ogni nodo i, una etichetta d_i con la proprietà che $d_t = 0$ e $d_i - d_j \leq 1$ se $(i,j) \in A_x$. Un siffatto insieme di etichette viene detto *etichettatura valida*, ed è facile verificare che d_i è una *valutazione per difetto* della lunghezza (numero di archi) dei cammini aumentanti da i a t, ossia per qualsiasi etichettatura valida non esistono cammini dal nodo i al pozzo t su G_x formati da meno di d_i archi. Data un'etichettatura valida, un arco $(i,j) \in A_x$ è detto *ammissibile per i* se $d_i = d_j + 1$. Se i è un nodo in attivo per x ed esiste un arco ammissibile (i,j), allora è possibile inviare l'eccesso, o una parte di esso, da i al nodo j che risulta, per l'etichettatura valida, "più vicino" a t attraverso l'*operazione di push*

$$\textit{Push}(\ i\ ,\ j\ ,\ x\ ,\ d\ )\{\ \theta = \min\{\ e_i\ ,\ u'_{ij} - x'_{ij}\ \};\ x'_{ij} = x'_{ij} + \theta; e_i = e_i - \theta;\ e_j = e_j + \theta;\ \}\ .$$

Grazie all'uso del grafo aumentato possiamo evitare di distinguere tra le operazioni di *push in avanti*, in cui $(i,j) \in A_x^+$, e quelle di *push all'indietro*, in cui $(i,j) \in A_x^-$, evitando anche di specificare l'aggiornamento del flusso sull'arco "gemello". Se invece il nodo attivo i non ha archi incidenti che siano ammissibili, significa che per ogni arco $(i,j) \in A'$ si ha $x'_{ij} = u'_{ij}$ (l'arco è saturo, ossia $(i,j) \notin A_x$) oppure $d_i < d_j + 1$. In altri termini, possiamo affermare che non esistono cammini aumentanti da i a t formati da d_i archi; quindi l'etichetta di i può essere incrementata attraverso l'*operazione di relabel*

$$\textit{Relabel}(\ d\ ,\ i\ )\{\ d_i = 1 + \min\{\ d_j\ :\ (i,j) \in A_x\};\ \}\ .$$

L'operazione di relabel rende ammissibile almeno un arco incidente in i (quello per cui si è ottenuto il valore minimo). Si può facilmente verificare che, se da un certo nodo i non si possono effettuare operazioni di push, allora l'applicazione di un'operazione di relabel del nodo i a partire da un'etichettatura valida produce una nuova etichettatura anch'essa valida. Si noti che la massima lunghezza di un cammino aumentante è $n - 1$. Pertanto, per ogni nodo i con $d_i \geq n$ si può affermare che non esistono cammini aumentanti da esso a t. Presentiamo adesso l'algoritmo basato su preflussi, che utilizza le operazioni di push e relabel per risolvere il problema del flusso massimo. L'idea dell'algoritmo consiste nell'introdurre nel grafo una quantità di flusso maggiore o uguale al valore del flusso massimo saturando tutti gli archi uscenti da s, e poi utilizzare un'etichettatura valida d per "assistere" il flusso nel suo tragitto fino a t.

```
procedure Preflow-Push( G , u , s , t , x , d ) {
  x = 0; foreach( (s,j) ∈ FS(s) ) do x_sj = u_sj;
  Etichettatura-Valida( G , d ); d_s = n;
  while( ∃i ∈ N con e_i > 0 ) do
    if( ∃(i,j) ∈ A_x con d_i = d_j + 1 )
      then Push( i , j , x , d );
      else Relabel( d , i );
}
```

Procedura 3.4: Algoritmo basato su preflussi

L'inizializzazione introduce nel grafo una quantità di flusso pari alla capacità del taglio $(\{s\}, N \setminus \{s\})$, e quindi maggiore o uguale al valore del flusso massimo, ed un'etichettatura valida d, ad esempio ponendo d_i uguale alla lunghezza del cammino minimo, in termini di numero di archi, da i a t. Ciò può essere facilmente ottenuto mediante una visita a ventaglio (in cui Q è implementato come una fila) "all'indietro" di G_x a partire dal nodo t, ossia percorrendo gli archi all'inverso del loro orientamento (cf. §B.3.2 ed il Teorema B.1). Si noti che la visita può escludere tutti gli archi uscenti da s, e quindi visitare solamente archi vuoti ($x_{ij} = 0$) nel verso opposto a quello del loro orientamento; in termini di G' si possono quindi visitare esattamente gli archi di $A' \setminus A_x$. Inizia a questo punto il "main cycle" dell'algoritmo: se è possibile effettuare un'operazione di push, l'algoritmo sposta flusso da un nodo attivo i ad un nodo j "più vicino" a t rispetto a i (secondo l'etichettatura valida d), mentre se non sono possibili operazioni di push, allora l'etichetta di un nodo attivo viene incrementata mediante un'operazione di relabel. L'algoritmo termina quando non ci sono più nodi attivi: tutto il flusso possibile è arrivato a t, il rimanente (la differenza tra la capacità del taglio $(\{s\}, N \setminus \{s\})$ e quella del taglio di capacità minima) è stato rispedito indietro a s. Al termine l'algoritmo, oltre ad un flusso massimo, individua anche un taglio (N_s, N_t) di capacità minima, "codificato" attraverso il vettore di etichette d: in particolare, fanno parte di N_s tutti quei nodi che, al termine dell'algoritmo, hanno un'etichetta di valore maggiore od uguale a n.
Un modo figurato per visualizzare il significato delle etichette è quello di considerarle come l'altezza dei nodi rispetto ad un piano comune: il flusso va dai nodi "più in alto" a quelli "più in basso", ma scendendo di un solo livello. Se un nodo ha un eccesso di flusso che non riesce a smaltire, viene portato al livello superiore del più in basso dei suoi vicini (raggiungibili attraverso archi non saturi o non vuoti) in modo da consentirgli di diminuire il suo sbilanciamento. Per visualizzare il comportamento dell'algoritmo riportiamo adesso un esempio di una sua esecuzione.

Esempio 3.14: Esecuzione di *Preflow-Push*
Nella tabella 3.1 è riportata una descrizione di tutte le operazione compiute dall'algoritmo *Preflow-Push* per risolvere l'istanza di Figura 3.16, a partire dall'etichettatura valida $d = [6, 2, 2, 1, 1, 0]$. Una descrizione grafica delle operazioni è anche mostrata in Figura 3.17 (da sinistra a destra, dall'alto in basso); per brevità, nella figura non sono mostrate tutte le iterazioni, ma sono evidenziati gli archi coinvolti in operazioni di push.

L'esempio precedente illustra alcune peculiarità dell'algoritmo: si inizia con un preflusso che ha sicuramente un valore maggiore od uguale al valore del flusso massimo. Una volta che tutto il flusso possibile è giunto a t, quello eventualmente in eccesso viene riportato a s: per questo, le etichette dei nodi appartenenti a N_s nel taglio minimo devono crescere ad un valore maggiore o uguale a n. L'algoritmo esegue allora una sequenza di operazioni di push e relabel mediante le quali i nodi in N_s si inviano ripetutamente le unità di flusso in eccesso al solo scopo di far aumentare le loro etichette fino ad un valore che consenta di re-instradare il flusso in eccesso fino ad s. Nell'esempio questa fase dell'algoritmo inizia alla nona iterazione, e coinvolge i nodi 2, 3 e 5 e gli archi $(2,3)$, $(2,5)$ e $(3,5)$.

Si può dimostrare che, scegliendo opportunamente il preflusso iniziale, è sempre possibile effettuare o un'operazione di push oppure un'operazione di relabel. Da questo può essere dedotta la completezza dell'algoritmo: la sequenza di operazioni di push tra due operazioni di relabel consecutive sposta eccesso da nodi più lontani verso nodi più vicini a t (rispetto alla etichettatura valida d che non cambia nel corso di tali operazioni). Pertanto, non è possibile avere due volte lo stesso stato di etichettatura e di eccessi. Inoltre, nessun nodo avrà mai etichetta maggiore di $2n - 2$: infatti, quando l'etichetta di un nodo in eccesso i viene aggiornata ad un valore $d_i \geq n$, essa eccede di un'unità l'etichetta di un

nodo	oper.	arco	θ	correzioni			
2	push	(2,4)	3	$e_x(2) = 2$	$x_{24} = 3$	$e_x(4) = 3$	
2	push	(2,5)	2	$e_x(2) = 0$	$x_{25} = 2$	$e_x(5) = 2$	
3	push	(3,5)	9	$e_x(3) = 0$	$x_{35} = 9$	$e_x(5) = 11$	
4	push	(4,6)	3	$e_x(4) = 0$	$x_{46} = 3$	$v_t = 3$	
5	push	(5,6)	8	$e_x(5) = 3$	$x_{56} = 8$	$v_t = 11$	
5	relabel			$d_5 =$	$d_4 + 1$	$= 2$	
5	push	(5,4)	2	$e_x(5) = 1$	$x_{54} = 2$	$e_x(4) = 2$	
4	push	(4,6)	2	$e_x(4) = 0$	$x_{46} = 5$	$v_t = 13$	
5	relabel			$d_5 =$	$d_2 + 1$	$= 3$	
5	push	(2,5)	1	$e_x(5) = 0$	$x_{25} = 1$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_3 + 1$	$= 3$	
2	push	(2,3)	1	$e_x(2) = 0$	$x_{23} = 1$	$e_x(3) = 1$	
3	relabel			$d_3 =$	$d_2 + 1$	$= 4$	
3	push	(2,3)	1	$e_x(3) = 0$	$x_{23} = 0$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_5 + 1$	$= 4$	
2	push	(2,5)	1	$e_x(2) = 0$	$x_{25} = 2$	$e_x(5) = 1$	
5	relabel			$d_5 =$	$d_2 + 1$	$= 5$	
5	push	(2,5)	1	$e_x(5) = 0$	$x_{25} = 1$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_3 + 1$	$= 5$	
2	push	(2,3)	1	$e_x(2) = 0$	$x_{23} = 1$	$e_x(3) = 1$	
3	relabel			$d_3 =$	$d_2 + 1$	$= 6$	
3	push	(2,3)	1	$e_x(3) = 0$	$x_{23} = 0$	$e_x(2) = 1$	
2	relabel			$d_2 =$	$d_5 + 1$	$= 6$	
2	push	(2,5)	1	$e_x(2) = 0$	$x_{25} = 2$	$e_x(5) = 1$	
5	relabel			$d_5 =$	$d_2 + 1$	$= 7$	
5	push	(3,5)	1	$e_x(5) = 0$	$x_{35} = 8$	$e_x(3) = 1$	
3	relabel			$d_3 =$	$d_2 + 1$	$= 7$	
3	push	(3,1)	1	$e_x(3) = 0$	$x_{13} = 8$	$v_s = 13$	STOP

Tabella 3.1: Esecuzione dell'algoritmo *Preflow-Push*

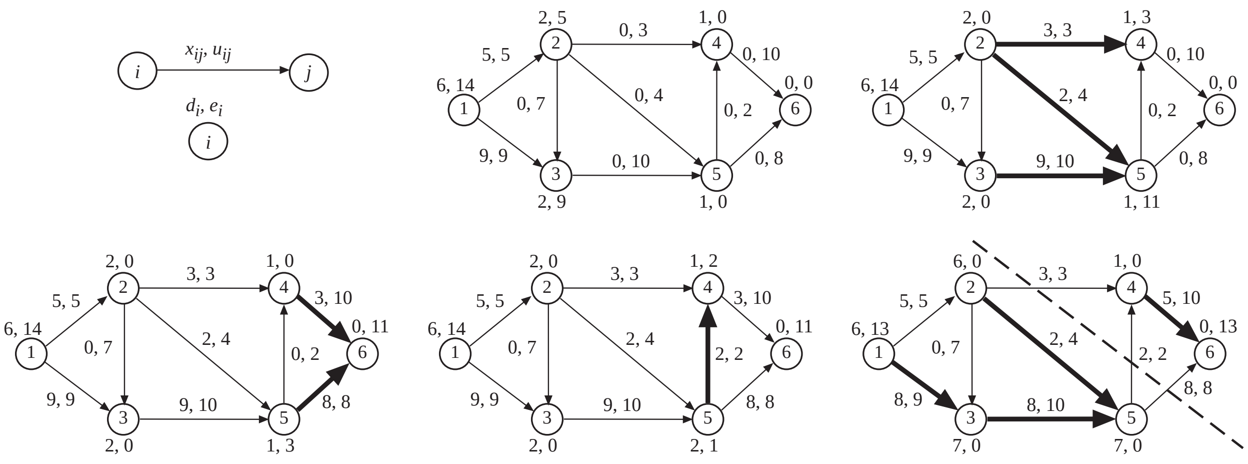


Figura 3.17: Esecuzione dell'algoritmo *Preflow-Push*

nodo j che si trova lungo un *cammino di ritorno* verso s, formato da al più $n-2$ archi. Pertanto non si eseguiranno mai più di $2n-3$ operazioni di relabel per uno stesso nodo, e quindi il numero globale di operazioni di relabel è $O(n^2)$. È possibile dimostrare che, se i nodi in eccesso vengono selezionati secondo criteri specifici, il numero globale di push che possono essere effettuate è $O(n^3)$, e che questo domina la complessità dell'algoritmo. Per i dettagli si rimanda alla letteratura citata.

Altrettanto importante è il fatto che l'algoritmo, se opportunamente implementato, risulta essere molto efficiente in pratica. Ciò richiede alcune accortezze, in particolare per evitare, almeno parzialmente, la fase finale di re-instradamento del flusso in eccesso. Alcune di tali accortezze si basano sull'evento che vi sia un *salto di etichettatura*, cioè che esista un valore intero $k < n$ per cui nessun nodo ha etichetta uguale a k. In tal caso, a tutti i nodi i con etichetta $k < d_i < n$ può essere assegnata un'etichetta $d_i = n + 1$.

Esempio 3.15: Salto di etichettatura
Questo capita nell'esempio precedente, subito dopo l'operazione di relabel del nodo 3 in cui $d_3 = 4$. Infatti, in quel momento, il vettore delle etichette è $d = [\,6\,,\,3\,,\,4\,,\,1\,,\,3\,,\,0\,]$. Verificandosi un salto di etichettatura per $k = 2$, si può porre $d_2 = d_3 = d_5 = 7$, risparmiando 14 operazioni e potendo re-instradare immediatamente l'unità di flusso in eccesso verso il nodo sorgente.

L'algoritmo si presta anche ad altre modifiche. Ad esempio, si può dimostrare che se si è interessati solamente al taglio di capacità minima (e non anche al flusso massimo), è possibile terminare l'algoritmo nel momento in cui tutti i nodi attivi hanno etichetta $d_i \geq n$: l'insieme di tutti i nodi in quel momento non raggiungibili da s attraverso cammini aumentanti fornisce infatti l'insieme N_t di un taglio di capacità minima. Si noti che, nell'esempio precedente, questo avviene nel momento del salto di etichettatura, che permetterebbe quindi di terminare l'algoritmo. Infine, è possibile proporre implementazioni parallele dell'algoritmo in ambiente asincrono, che richiedono solamente comunicazione locale tra processori che controllano nodi adiacenti; per questi dettagli si rimanda ancora una volta alla letteratura citata.

Esercizio 3.31 *Si discuta come modificare la procedura* Preflow-Push *per risolvere le due varianti del problema di flusso massimo con più sorgenti/pozzi, presentate nel paragrafo 3.3.3, senza costruire esplicitamente il grafo aumentato G' (si noti che, nella prima variante, gli archi uscenti dalla "super-radice" s hanno capacità infinita).*

3.4 Il problema del Flusso di Costo Minimo

Studiamo adesso algoritmi risolutivi (diversi da quelli accennati in §3.1.3) per il problema (MCF), introdotto nel paragrafo 3.1. Notiamo che (MCF) è più generale sia di (SPT) che di (MF): infatti, entrambi questi problemi possono essere formulati come particolari problemi di flusso di costo minimo. In particolare, (SPT) e (MF) presentano caratteristiche distinte, che sono contemporaneamente presenti in (MCF): in (SPT) gli archi hanno associati costi ma non capacità, mentre in (MF) gli archi hanno associate capacità ma non costi. In entrambi i casi, inoltre, la struttura delle domande/offerte dei nodi è molto particolare. In effetti, gli algoritmi per (MCF) spesso fanno uso, al loro interno, di algoritmi per (SPT) o (MF).

3.4.1 Cammini, cicli aumentanti e condizioni di ottimo

Il primo passo per lo sviluppo degli algoritmi consiste nello studio delle condizioni di ottimalità per il problema, ossia di proprietà che consentano di verificare se una data soluzione sia ottima. Per poter presentare una più vasta gamma di algoritmi, deriveremo le condizioni di ottimo non per i flussi, ma per il sovrainsieme degli *pseudoflussi*, dove uno pseudoflusso è un vettore $x \in \mathbb{R}^m$ che rispetta i soli vincoli di capacità sugli archi (3.2). Definiamo *sbilanciamento* di un nodo i rispetto ad x la quantità

$$e_x(i) = \textstyle\sum_{(j,i)\in BS(i)} x_{ji} - \sum_{(i,j)\in FS(i)} x_{ij} - b_i$$

e indichiamo con $O_x = \{\, i \in N \,:\, e_x(i) > 0 \,\}$ e $D_x = \{\, i \in N \,:\, e_x(i) < 0 \,\}$, rispettivamente, l'insieme dei nodi con *eccedenza di flusso* e con *difetto di flusso*: se $D_x = \emptyset \equiv O_x = \emptyset \equiv e_x = 0$, ossia tutti i nodi sono *bilanciati*, il vettore x rispetta anche i vincoli (3.1) ed è pertanto un flusso ammissibile. Un'altra condizione equivalente usa lo *sbilanciamento complessivo* di x

$$g(x) = \textstyle\sum_{i\in O_x} e_x(i) \;\; (= -\sum_{j\in D_x} e_x(j)\,) \quad :$$

x è un flusso ammissibile se e solo se $g(x) = 0$.

Dato un cammino P, non necessariamente orientato, tra una qualunque coppia di nodi s e t del grafo, definiamo come verso di P quello che va da s a t. Gli archi del cammino risultano quindi partizionati nei due insiemi P^+ e P^-, rispettivamente degli archi concordi e discordi con il verso del cammino. Un cammino si dirà *aumentante* se la sua capacità $\theta(P,x)$, definita in (3.12), è positiva. Dato uno pseudoflusso x, è possibile inviare una quantità di flusso $0 < \theta \leq \theta(P,x)$ lungo P mediante l'*operazione di composizione* definita in (3.11), ottenendo un nuovo pseudoflusso $x(\theta) = x \oplus \theta P$ tale che

$$e_{x(\theta)}(i) = \begin{cases} e_x(s) - \theta & \text{se } i = s \\ e_x(t) + \theta & \text{se } i = t \\ e_x(i) & \text{altrimenti} \end{cases} \quad ;$$

in altre parole, inviare flusso lungo un cammino aumentante modifica solamente lo sbilanciamento dei nodi estremi del cammino, mentre lo sbilanciamento di tutti gli altri nodi rimane invariato. Un caso particolare di cammino aumentante è quello in cui $s = t$, ossia il cammino è un *ciclo* su cui è arbitrariamente fissato un verso di percorrenza. Chiaramente, inviare flusso lungo un ciclo (aumentante) non modifica lo sbilanciamento di alcun nodo; di conseguenza, se in particolare x è un flusso *ammissibile*, allora ogni flusso $x(\theta) = x \oplus \theta C$ per $0 \leq \theta \leq \theta(C,x)$ è ancora ammissibile.
Il *costo* di un cammino (o ciclo) P, che indicheremo con $c(P)$, è il costo di invio di un'unità di flusso lungo P secondo il verso fissato, ossia

$$c(P) = \textstyle\sum_{(i,j)\in P^+} c_{ij} - \sum_{(i,j)\in P^-} c_{ij} \quad ; \tag{3.17}$$

ed è immediato verificare che

$$cx(\theta) = c(x \oplus \theta P) = cx + \theta c(P) \; . \tag{3.18}$$

Per determinare cicli e/o cammini aumentanti si può usare il *grafo residuo* $G_x = (N, A_x)$ rispetto allo pseudoflusso x, definito come per il problema di flusso massimo: per ogni arco $(i,j) \in A$ si pone (i,j) in A_x, con costo $c'_{ij} = c_{ij}$, se e solo se $x_{ij} < u_{ij}$, e si pone (j,i) in A_x, con costo $c'_{ji} = -c_{ij}$, se e solo se $x_{ij} > 0$. È immediato verificare che vale la seguente generalizzazione del Lemma 3.4:

Lemma 3.6 *Comunque si fissino s e t, per ogni cammino aumentante da s a t rispetto ad x in G esiste uno ed un solo cammino orientato da s a t in G_x, ed i due cammini hanno lo stesso costo.*

Possiamo ora dimostrare che cammini e cicli aumentanti sono gli "strumenti base per costruire pseudoflussi, e quindi flussi".

Teorema 3.9 *Siano dati due qualunque pseudoflussi x' ed x'': allora esistono $k \leq n + m$ cammini o cicli aumentanti (semplici) rispetto a x', $P_1, \ldots, P_k$, di cui al più m sono cicli, tali che $x^1 = x'$, $x^{i+1} = x^i \oplus \theta_i P_i$, per $i = 1, \ldots, k$, $x^{k+1} = x''$, dove $0 < \theta_i \leq \theta(P_i, x')$. In particolare, tutti i cammini aumentanti hanno come estremi nodi in cui lo sbilanciamento di x' è diverso dallo sbilanciamento di x'', per cui se $e_{x'} = e_{x''}$ allora tutti i P_i sono cicli.*

Dimostrazione La dimostrazione è costruttiva: manteniamo uno pseudoflusso x, inizialmente pari ad x', ed in un numero finito di passi lo rendiamo uguale ad x'' utilizzando cammini e cicli aumentanti rispetto a x'. Per questo definiamo il grafo $\bar{G}_x = (N, \bar{A}^+_x \cup \bar{A}^-_x)$, dove $\bar{A}^+_x = \{\, (i,j) \,:\, x''_{ij} > x_{ij} \,\}$ e $\bar{A}^-_x = \{\, (j,i) \,:\, x''_{ij} < x_{ij} \,\}$. $\bar{G}_x$ "descrive" la differenza tra x ed x''; è immediato verificare che $\bar{A}^+_x = \bar{A}^-_x = \emptyset$ se e solo se $x'' = x$. Ad ogni arco $(i,j) \in \bar{A}^+_x$ associamo la capacità $u^x_{ij} = x''_{ij} - x_{ij} > 0$, e, analogamente, ad ogni arco $(j,i) \in \bar{A}^-_x$ associamo la capacità $u^x_{ji} = x_{ij} - x''_{ij} > 0$; per ogni cammino (ciclo) orientato P in $\bar{G}_x$ definiamo come sua capacità $\theta^x(P) = \min\{\, u^x_{ij} \,:\, (i,j) \in P \,\}$. Definiamo inoltre gli insiemi $\bar{O}_x = \{\, i \in N \,:\, e_x(i) > e_{x''}(i) \,\}$ e $\bar{D}_x = \{\, i \in N \,:\, e_x(i) < e_{x''}(i) \,\}$, rispettivamente, dei nodi che hanno sbilanciamento rispetto a x maggiore dello sbilanciamento rispetto a x'' e di quelli in cui avviene l'opposto. È facile verificare che $\bar{O}_x = \bar{D}_x = \emptyset$ se e solo se x ed x'' hanno lo stesso vettore di sbilanciamento; inoltre tutti i nodi in $\bar{O}_x$ hanno almeno un arco uscente in $\bar{G}_x$, tutti i nodi in $\bar{D}_x$ hanno almeno un arco entrante in $\bar{G}_x$, mentre tutti i nodi in $N \setminus (\bar{O}_x \cup \bar{D}_x)$ o non hanno né archi entranti né archi uscenti oppure hanno sia almeno un arco entrante che almeno un arco uscente.
Utilizzando $\bar{G}_x$ è possibile costruire iterativamente i cicli e cammini richiesti. Se $\bar{A}^+_x = \bar{A}^-_x = \emptyset$, ossia $x = x''$, il procedimento termina, altrimenti consideriamo gli insiemi $\bar{O}_x$ e $\bar{D}_x$: se $\bar{O}_x \neq \emptyset$ si seleziona un nodo $s \in \bar{O}_x$, altrimenti, per costruire un ciclo, si seleziona un qualsiasi nodo s che abbia almeno un arco uscente (e quindi almeno uno entrante). Si visita quindi $\bar{G}_x$ a partire da s, che ha sicuramente almeno un arco uscente; siccome ogni nodo tranne al più quelli in $\bar{D}_x$ ha almeno un arco uscente, in un numero finito di passi la visita:

- o raggiunge un nodo $t \in \bar{D}_x$;
- oppure raggiunge un nodo precedentemente visitato.

Nel primo caso si determina un cammino (semplice) P in $\bar{G}_x$ da un nodo $s \in \bar{O}_x$ a un nodo $t \in \bar{D}_x$; su questo cammino viene inviata una quantità di flusso pari a $\theta = \min\{\, \theta^x(P)\,,\, e_x(s) - e_{x''}(s)\,,\, e_{x''}(t) - e_x(t)\,\}$. Altrimenti si determina un ciclo (semplice) C in $\bar{G}_x$ e su C viene inviata una quantità di flusso pari a $\theta^x(C)$. In questo modo si ottiene un nuovo pseudoflusso x "più simile" ad x'' del precedente, in quanto u^x_{ij} diminuisce della quantità $\theta > 0$ per ogni $(i,j) \in P$ (C). In particolare, se si determina un ciclo si avrà $u^x_{ij} = 0$ per almeno un $(i,j) \in C$, e quindi tale arco non comparirà più in $\bar{G}_x$; se invece si determina un cammino allora si avrà che o $u^x_{ij} = 0$ per almeno un $(i,j) \in P$, oppure lo sbilanciamento rispetto ad x di almeno uno tra s e t diventa pari allo sbilanciamento rispetto ad x'', e quindi almeno un nodo tra s e t non comparirà più in $\bar{O}_x$ o in $\bar{D}_x$. Si noti che sugli archi di $\bar{A}^+_{x'}$ il flusso può solo aumentare, mentre sugli archi di $\bar{A}^-_{x'}$ il flusso può solo diminuire; siccome il flusso su (i,j) non viene più modificato non appena $x_{ij} = x''_{ij}$, nessun "nuovo" arco può essere creato in $\bar{G}_x$. Pertanto, ad ogni passo $\bar{G}_x$ è un sottografo del grafo residuo iniziale $G_{x'}$, e quindi un qualunque cammino (o ciclo) che viene utilizzato è aumentante rispetto allo pseudoflusso iniziale x'; è anche facile verificare che la quantità θ inviata lungo ogni cammino (o ciclo) è minore od uguale della capacità di quel cammino rispetto ad x'. Siccome ad ogni passo o si cancella almeno un arco da $\bar{G}_x$ o si cancella almeno un nodo da $\bar{O}_x \cup \bar{D}_x$, in al più $n + m$ passi tutti gli archi di $\bar{G}_x$ vengono cancellati e l'algoritmo termina. $\diamond$

Il caso particolare in cui $x' = 0$ mostra che qualsiasi pseudoflusso x può essere costruito inviando opportune quantità di flusso lungo un "piccolo" numero di cammini e cicli. Il Teorema 3.9 prende infatti il nome di *teorema di decomposizione de(gl)i (pseudo)flussi*, in quanto consente di rappresentare qualsiasi (pseudo)flusso come una composizione di un piccolo numero di sue "costituenti elementari", ossia flussi lungo cammini/cicli. Inoltre, il teorema consente di caratterizzare gli pseudoflussi, e quindi i flussi, "ottimi". Definiamo infatti *minimale* uno pseudoflusso x che abbia costo minimo tra tutti gli pseudoflussi aventi lo stesso vettore di sbilanciamento e_x; si noti che ogni soluzione ottima di (MCF) è un flusso ammissibile minimale, avendo costo minimo tra tutti gli (pseudo)flussi con $e_x = 0$.

Corollario 3.1 *Uno pseudoflusso (flusso ammissibile) x è minimale (ottimo) se e solo se non esistono cicli aumentanti rispetto ad x il cui costo sia negativo.*

Dimostrazione Se esiste un ciclo aumentante C rispetto ad x il cui costo $c(C)$ è negativo, allora x non è minimale: per ogni $0 < \theta \leq \theta(C, x)$, lo pseudoflusso $x(\theta) = x \oplus \theta C$ ha lo stesso vettore di sbilanciamento di x, ma $cx(\theta) < cx$ (si veda (3.18)). Viceversa, sia x uno pseudoflusso tale che non esistono cicli aumentanti di costo negativo rispetto ad x, e supponiamo che x non sia minimale, ossia che esista uno pseudoflusso x' con lo stesso vettore di sbilanciamento tale che $cx' < cx$: per il Teorema 3.9 si ha $x' = x \oplus \theta_1 C_1 \oplus \ldots \oplus \theta_k C_k$, dove C_i sono cicli aumentanti rispetto a x, ma siccome tutti i θ_i sono numeri positivi, $cx' < cx$ e (3.18) implicano che $c(C_i) < 0$ per un qualche i, il che contraddice l'ipotesi. $\diamond$

Nei prossimi paragrafi vedremo come la teoria appena sviluppata può essere utilizzata per progettare algoritmi risolutivi per (MCF).

3.4.2 Algoritmo basato su cancellazione di cicli

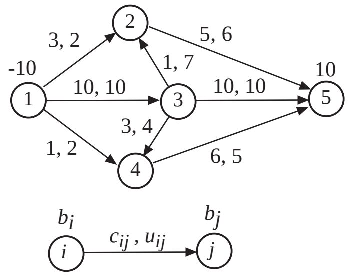
Il Corollario 3.1 suggerisce immediatamente un approccio risolutivo per (MCF): si determina un flusso ammissibile, e poi si utilizzano cicli aumentanti di costo negativo per ottenere flussi ammissibili di costo inferiore. L'algoritmo termina quando non esistono più cicli aumentanti di costo negativo: il flusso ammissibile così determinato è sicuramente di costo minimo. Quando si satura un ciclo aumentante C, inviando lungo i suoi archi un flusso pari a $\theta(C, x)$, si dice che si "cancella" il ciclo C in quanto esso non risulta più aumentante per il flusso $x(\theta)$; non si può però escludere che C possa tornare ad essere aumentante per flussi generati successivamente. Ciò porta alla definizione del seguente algoritmo.

```
procedure Cancella-Cicli( G , c , b , u , x , caso ) {
  if( Flusso-Ammissibile( G , b , u , x )
    then { while( Trova-Ciclo( G , c , u , x , C , θ ) do
                Cambia-Flusso( x , C , θ );
           caso = "ottimo";
           }
    else caso = "vuoto";
}
```

Procedura 3.5: *Algoritmo basato sulla cancellazione di cicli*

La procedura *Flusso-Ammissibile* determina, se esiste, un flusso ammissibile: in tal caso restituisce *vero* ed il flusso x, altrimenti restituisce $falso$. Una possibile implementazione di questa procedura è stata discussa nel paragrafo 3.3.3. La procedura *Trova-Ciclo* determina, dato x, se esiste un ciclo aumentante rispetto ad x di costo negativo: in questo caso restituisce *vero* ed il ciclo individuato C, con il suo verso e la sua capacità $\theta = \theta(C, x)$, altrimenti restituisce $falso$. Dal Lemma 3.6 segue che il problema di determinare un ciclo aumentante di costo negativo in G rispetto ad x è equivalente al problema di determinare un ciclo orientato di costo negativo in G_x; tale problema può essere risolto in diversi modi, ad esempio utilizzando la procedura $SPT.L.queue$. In particolare, si può utilizzare la procedura per il problema dell'albero dei cammini minimi con radici multiple (si veda il paragrafo 3.2.7) con insieme di radici $R = N$; ciò corrisponde ad aggiungere al grafo residuo una radice fittizia r e un arco (r, j) di costo $c_{rj} = 0$ per ogni nodo $j \in N$. Una volta determinato il ciclo C ed il valore θ, la procedura *Cambia-Flusso* costruisce il nuovo flusso $x(\theta) = x \oplus \theta C$.

Esempio 3.16: Esecuzione dell'algoritmo *Cancella-Cicli*



Sia data l'istanza di (MCF) raffigurata qui accanto; il funzionamento dell'algoritmo basato sulla cancellazione di cicli è mostrato in Figura 3.18. Una soluzione ammissibile può essere facilmente ottenuta inviando 10 unità di flusso da 1 a 5 lungo gli archi $(1, 3)$ e $(3, 5)$. I successivi cinque grafi (da sinistra a destra, dall'alto in basso) mostrano i passi svolti dall'algoritmo a partire da tale soluzione ammissibile: per ogni iterazione vengono mostrati il valore del flusso su tutti gli archi in cui non è nullo, il ciclo selezionato (in grassetto) ed il suo verso (freccia tratteggiata), il valore della funzione obiettivo, $c(C)$ e $\theta(C, x)$. Il grafo in basso a destra fornisce invece la *dimostrazione* che la soluzione determinata è effettivamente ottima: in figura viene mostrato il grafo residuo G_x corrispondente a tale soluzione, con i relativi costi degli archi, ed il corrispondente albero dei cammini minimi con insieme di radici $R = N$, con le relative etichette (è facile verificare che le condizioni di Bellman sono rispettate). Dato che G_x ammette un albero dei cammini minimi, non esistono cicli orientati di costo negativo in G_x, e quindi non esistono cicli aumentanti rispetto ad x di costo negativo, il che garantisce che x sia un flusso ottimo.

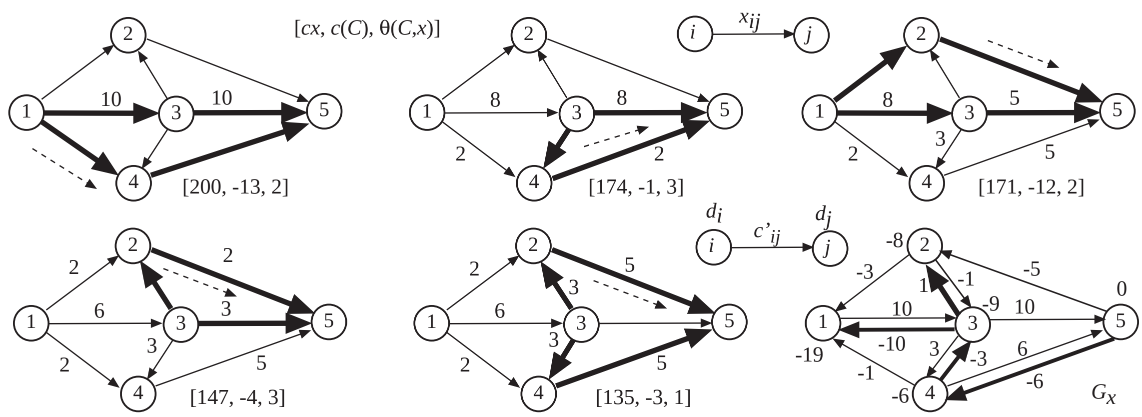


Figura 3.18: Esecuzione dell'algoritmo *Cancella-Cicli*

La correttezza dell'algoritmo *Cancella-Cicli* discende direttamente dal Corollario 3.1; analizziamo adesso la sua complessità. Innanzitutto notiamo che, se le capacità degli archi ed i bilanci dei nodi sono numeri interi, allora ad ogni passo dell'algoritmo il flusso x è intero. Infatti possiamo assumere che il flusso restituito da *Flusso-Ammissibile* sia intero (si veda il Teorema 3.7): se all'inizio di un'iterazione x è intero allora lo è anche $\theta = \theta(C, x)$, e quindi lo è anche il flusso al termine dell'iterazione. Sia adesso $\bar{u} = \max\{\, u_{ij} \,:\, (i,j) \in A \,\}$ la massima capacità degli archi, che assumiamo finita, e sia $\bar{c} = \max\{\, |c_{ij}| \,:\, (i,j) \in A \,\}$ il massimo valore assoluto dei costi degli archi: è facile verificare che il costo di qualunque soluzione ammissibile è compreso tra $m\bar{u}\bar{c}$ e $-m\bar{u}\bar{c}$. Se tutti i costi sono interi, il costo di qualsiasi ciclo aumentante utilizzato dall'algoritmo sarà sicuramente inferiore od uguale a -1. Siccome $\theta \geq 1$, ad ogni iterazione il valore della funzione obiettivo diminuisce di almeno un'unità, e quindi non possono essere eseguite più di $O(m\bar{u}\bar{c})$ iterazioni. Quindi, se i vettori b, c ed u hanno tutte componenti finite e intere, allora l'algoritmo termina. La complessità dipende allora dal modo in cui è realizzata la procedura *Trova-Ciclo*: poiché la procedura $SPT.L.queue$ permette di determinare un ciclo di costo negativo in $O(mn)$, l'algoritmo *Cancella-Cicli* può sicuramente essere implementato in modo da avere complessità pseudopolinomiale $O(nm^2\bar{u}\bar{c})$.

L'analisi precedente indica come ci siano molte strade possibili per migliorare la complessità asintotica (e quindi, sperabilmente, anche l'efficienza in pratica) dell'algoritmo. Innanzitutto, l'algoritmo dovrebbe "decrementare di molto" il valore della funzione obiettivo ad ogni iterazione, in modo da diminuire il numero di iterazioni che compie. Per questo si potrebbe pensare di determinare il ciclo che minimizza il valore $c(C) \cdot \theta(C, x) < 0$ (di quanto la funzione obiettivo diminuisce), ma questo è difficile: infatti, anche solo determinare un ciclo che minimizzi il valore $c(C)$ è un problema $\mathcal{NP}$-arduo (si veda il paragrafo 3.2.1). Esistono diversi modi possibili per affrontare il problema. Un'idea interessante sono le *tecniche di scalatura*, nelle quali i costi e le capacità vengono mappati su piccoli intervalli di numeri interi. Si immagini ad esempio di mappare tutte le capacità su $\{0, 1\}$; ciò significa che qualsiasi arco con capacità $< \bar{u}/2$ viene considerato avere capacità 0 (e quindi temporaneamente eliminato). Se sul grafo così modificato si determina un ciclo aumentante di costo negativo, quindi, la sua capacità reale è almeno pari a $\bar{u}/2$. Quando questo non è più possibile, si raffina l'insieme delle capacità (ad esempio ponendolo pari a $\{0, 1, 2, 3\}$); operando in modo opportuno, e similmente sui costi, si riescono ad ottenere valutazioni di complessità simili a quella precedentemente esposta, in cui i termini pseudopolinomiali $\bar{c}$ e $\bar{u}$ si riducono però a $\log \bar{c}$ e $\log \bar{u}$ (risultando quindi polinomiali).

Esercizio 3.32 *Si sviluppino in dettaglio versioni dell'algoritmo basato sulla cancellazione di cicli con tecniche di scalatura sulle capacità e/o sui costi che abbiano complessità polinomiale.*

Esistono anche idee diverse. Ad esempio si dimostra che il problema di determinare un ciclo di *lunghezza media* (lunghezza diviso per il numero di archi) minima è polinomiale, e che algoritmi di cancellazione di cicli che usano questi cicli possono essere implementati in modo da avere complessità *pienamente polinomiale* (ovvero in polinomiale in n ed m, e indipendente da $\bar{c}$ e $\bar{u}$). Per ulteriori dettagli si rimanda alla letteratura citata.
Un'altra possibilità di miglioramento consiste nel diminuire la complessità di determinazione di un ciclo di costo negativo. Ad esempio, è facile verificare che l'algoritmo del Simplesso Duale applicato alla soluzione di (MCF), descritto nel paragrafo 3.1.3, è un'implementazione dell'algoritmo *Cancella-Cicli* in cui il costo di determinare un ciclo è $O(m)$. Ciò si ottiene considerando solamente i cicli ottenibili aggiungendo all'albero di base un singolo arco. È possibile dimostrare che esiste un ciclo aumentante di costo negativo se e solo se uno di questi $m - n$ cicli è aumentante ed ha costo negativo.

Esercizio 3.33 *Si dimostri l'affermazione precedente.*

3.4.3 Algoritmo basato su cammini minimi successivi

Un approccio sostanzialmente diverso è quello dell'algoritmo dei *cammini minimi successivi*, che mantiene ad ogni passo uno pseudoflusso *minimale* x e determina un *cammino aumentante di costo minimo* da un nodo $s \in O_x$ a un nodo $t \in D_x$ per diminuire, al minor costo possibile, lo sbilanciamento di x. L'uso di cammini aumentanti di costo minimo permette di conservare la minimalità degli pseudoflussi:

Teorema 3.10 *Sia x uno pseudoflusso minimale, e sia P un cammino aumentante rispetto a x avente costo minimo tra tutti i cammini che uniscono un dato nodo $s \in O_x$ ad un dato nodo $t \in D_x$: allora, comunque si scelga $\theta \leq \theta(P, x)$, $x(\theta) = x \oplus \theta P$ è uno pseudoflusso minimale.*

Dimostrazione Fissato $\theta \leq \theta(P, x)$, sia x' un qualsiasi pseudoflusso con vettore di sbilanciamento $e_{x(\theta)}$. Il Teorema 3.9 mostra che esistono k cammini aumentanti $P_1, \ldots, P_k$ da s a t (in quanto s e t sono gli unici nodi in cui e_x ed $e_{x(\theta)}$ differiscono) e $h \leq m$ cicli aumentanti $C_1, \ldots, C_h$ rispetto ad x tali che

$$x' = x \oplus \theta_1 P_1 \oplus \ldots \oplus \theta_k P_k \oplus \theta_{k+1} C_1 \oplus \ldots \oplus \theta_{k+h} C_h \quad .$$

In più, deve sicuramente essere $\theta_1 + \ldots + \theta_k = \theta$. Siccome x è minimale, ciascuno degli h cicli aumentanti deve avere costo non negativo; inoltre, siccome P ha costo minimo tra tutti i cammini aumentanti da s a t, si ha $c(P) \leq c(P_i)$, $i = 1, \ldots, k$. Di conseguenza

$$cx' = cx + \theta_1 c(P_1) + \ldots + \theta_k c(P_k) + \theta_{k+1} c(C_1) + \ldots + \theta_{k+h} c(C_h) \geq cx + \theta c(P) = cx(\theta) \quad ,$$

e quindi $x(\theta)$ è minimale. ◇

Con un'opportuna scelta di θ, l'operazione di composizione tra lo pseudoflusso x ed il cammino P permette di diminuire lo sbilanciamento complessivo: infatti, è immediato verificare che per

$$\theta = \min\{\ \theta(P, x)\ ,\ e_x(s)\ ,\ -e_x(t)\ \} > 0 \tag{3.19}$$

(si ricordi che $e_x(s) > 0$ e $e_x(t) < 0$), $x(\theta)$ è uno pseudoflusso (minimale) con sbilanciamento complessivo $g(x(\theta)) = g(x) - \theta < g(x)$. Questa scelta di θ corrisponde alla maggior diminuzione possibile dello sbilanciamento complessivo corrispondente al cammino P ed allo pseudoflusso x. Ciò conduce direttamente alla definizione del seguente algoritmo:

```
procedure Cammini-Minimi-Successivi( G , c , b , u , x , caso ) {
  Inizializza( x , c , u ); caso = "ottimo";
  while( g(x) ≠ 0 ) do
    if( Trova-Cammino-Minimo( G_x , O_x , D_x , P , θ ) )
      then Aumenta-Flusso( x , P , θ );
      else { caso = "vuoto"; break; }
}
```

Procedura 3.6: *Algoritmo basato su cammini minimi successivi*

La procedura *Inizializza* costruisce uno pseudoflusso x minimale: un semplice modo per implementare tale procedura è quello di porre, per ogni $(i, j) \in A$, $x_{ij} = 0$ se $c_{ij} \geq 0$, e $x_{ij} = u_{ij}$ altrimenti. In tal modo i costi degli archi in G_x sono tutti non negativi, e quindi non esistono cicli orientati in G_x (cicli aumentanti rispetto ad x in G) di costo negativo, per cui x è minimale. In effetti è anche facile verificare che lo pseudoflusso così costruito ha costo minimo tra tutti i possibili pseudoflussi per il dato vettore di capacità u. Si noti che questa fase di inizializzazione richiede che non esistano archi con costo negativo e capacità infinita.

La procedura *Trova-Cammino-Minimo* determina un cammino aumentante di costo minimo P da un qualsiasi nodo $s \in O_x$ a un qualsiasi nodo $t \in D_x$. Un possibile modo per implementare questa procedura è risolvere un problema di albero dei cammini minimi con insieme di nodi radice O_x (si veda il paragrafo 3.2.7) su G_x; in altri termini, se $|O_x| > 1$ si aggiunge a G_x un nodo "radice" r collegato a tutti i nodi in O_x con archi a costo nullo, e poi si risolve un problema di albero dei cammini minimi di radice r sul grafo così ottenuto (altrimenti basta usare come radice l'unico nodo in O_x). La procedura determina sicuramente un albero dei cammini minimi: infatti x è minimale, e quindi non esistono cicli negativi in G_x. Una volta calcolato l'albero dei cammini minimi, si seleziona un qualsiasi nodo $t \in D_x$ (ad esempio, quello con etichetta d_t minima, corrispondente al più corto tra tutti i cammini minimi) e si esamina il valore della sua etichetta. Se $d_t = \infty$, allora non esiste alcun cammino aumentante da qualsiasi nodo $s \in O_x$ a t, *Trova-Cammino-Minimo* restituisce $false$ e di conseguenza *Cammini-Minimi-Successivi* restituisce *caso* = "vuoto"; infatti, in questo caso non esiste nessuna soluzione ammissibile per il problema di flusso di costo minimo.

Esercizio 3.34 *Si dimostri l'affermazione precedente (suggerimento: si consideri il paragrafo 3.3.3).*

Se invece $d_t < \infty$ allora *Trova-Cammino-Minimo* restituisce *vero* ed il cammino aumentante P che unisce un nodo $s \in O_x$ al nodo $t \in D_x$ selezionato, insieme alla quantità di flusso θ, definita in (3.19), che deve essere inviata lungo P. Questo viene fatto dalla procedura *Aumenta-Flusso*, che implementa l'operazione di composizione $\oplus$, in modo simile alla *Aumenta-Flusso* utilizzata per l'algoritmo *Cammini-Aumentanti*. Si noti che se $\theta = e_x(s)$ allora il nodo s risulterà bilanciato rispetto al nuovo flusso, ed analogamente per il nodo t se $\theta = -e_x(t)$; altrimenti, θ è determinato dalla capacità del cammino, il che significa che almeno un arco di P diviene saturo oppure vuoto.

Siccome l'algoritmo usa sempre cammini aumentanti di costo minimo, per il Teorema 3.10 ad ogni passo lo pseudoflusso x è minimale: quindi, se l'algoritmo termina con $g(x) = 0$, allora x è un flusso ottimo. La terminazione dell'algoritmo può essere facilmente provata nel caso in cui b e u siano interi. Infatti, in questo caso lo pseudoflusso iniziale è anch'esso intero, e quindi lo è la quantità θ a quell'iterazione, e quindi lo è anche lo pseudoflusso x ottenuto al termine dell'iterazione. Di conseguenza, ad ogni iterazione x è intero, $\theta \geq 1$ e $g(x)$ diminuisce di almeno un'unità, e quindi l'algoritmo termina in un numero finito di iterazioni. Da questa analisi segue:

Teorema 3.11 *Se le capacità degli archi ed i bilanci dei nodi sono interi, allora per qualsiasi vettore di costi associati agli archi esiste una soluzione ottima intera per il problema (MCF).*

Questa *proprietà di integralità* è molto importante per le applicazioni: si pensi ad esempio al caso in cui il flusso su un arco rappresenta il numero di camion, o carrozze ferroviarie, o containers, o ancora pacchetti in una rete di comunicazione. In effetti, una delle motivazioni principali per cui si studia (MCF) risiede nel fatto che essa costituisce la più grande ed utile classe di problemi che possiedono questa proprietà, come discusso nel Capitolo 4 ed in quelli successivi.

Esempio 3.17: Esecuzione dell'algortimo *Cammini-Minimi-Successivi*

Consideriamo di nuovo l'istanza del problema (MCF) dell'Esempio 3.16; il funzionamento dell'algoritmo basato su cammini minimi successivi è mostrato in figura 3.19. Siccome tutti i costi sono non negativi, la procedura *Inizializza* costruisce uno pseudoflusso iniziale identicamente nullo. Le iterazioni procedono da sinistra a destra: in alto viene mostrato il grafo residuo G_x, e sotto lo pseudoflusso ottenuto al termine dell'iterazione. In G_x non sono riportati (per chiarezza di visualizzazione) i costi degli archi, ma è evidenziato l'albero dei cammini minimi con i valori delle corrispondenti etichette; è inoltre mostrato il valore θ del flusso inviato lungo il relativo cammino aumentante da 1 a 5. I valori del flusso e degli sbilanciamenti sono mostrati solamente per quegli archi/nodi in cui sono diversi da zero. Nella quarta iterazione tutti i nodi hanno sbilanciamento nullo, e la soluzione è ottima.

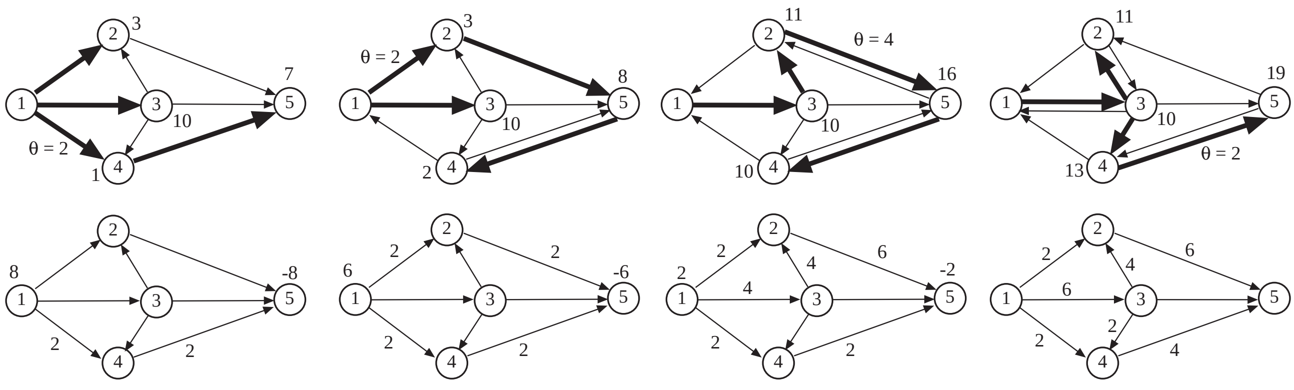


Figura 3.19: Esecuzione dell'algoritmo basato su cammini minimi successivi

Per analizzare la complessità dell'algoritmo, si consideri che lo sbilanciamento complessivo dello pseudoflusso x costruito da *Inizializza* è limitato superiormente da $\overline{g} = \sum_{c_{ij}<0} u_{ij} + \sum_{b_i>0} b_i$; siccome $g(x)$ diminuisce di almeno un'unità ad ogni iterazione, il numero di iterazioni non potrà eccedere $\overline{g}$. È facile vedere che tutte le operazioni effettuate durante una singola iterazione hanno complessità $O(n)$, esclusa l'invocazione della procedura *Trova-Cammino-Minimo*: se utilizziamo l'algoritmo *SPT.L.queue*, che ha complessità $O(mn)$, la procedura *Cammini-Minimi-Successivi* risulta avere complessità pseudopolinomiale $O(\overline{g}mn)$.

Analogamente al caso dell'algoritmo *Cancella-Cicli*, ci sono due modi per migliorare la complessità asintotica (e quindi, sperabilmente, anche l'efficienza in pratica) dell'algoritmo: utilizzare cammini che portano "molto flusso", in modo da diminuire il numero di iterazioni necessarie ad ottenere un flusso ammissibile, oppure utilizzare algoritmi più efficienti di $SPT.L.queue$ per calcolare il cammino minimo. Per il primo approccio si possono utilizzare, ad esempio, le tecniche di scalatura già accennate in precedenza. Per il secondo è necessario (almeno in teoria) utilizzare algoritmi tipo $SPT.S$ su grafi con costi non negativi. Questo è in effetti possibile modificando opportunamente i costi mediante un vettore $\pi \in \mathbb{R}^n$ di *potenziali* dei nodi, mediante i quali si può definire il *costo ridotto* $c^{\pi}_{ij} = c_{ij} + \pi_i - \pi_j$ di ogni arco (i, j) rispetto a π.

Teorema 3.12 *Uno pseudoflusso x è minimale se e solo se esiste un vettore di potenziali π tale che*

$$\begin{array}{lcl} x_{ij} > 0 & \Longrightarrow & c^{\pi}_{ij} \leq 0 \\ x_{ij} < u_{ij} & \Longrightarrow & c^{\pi}_{ij} \geq 0 \end{array} \quad . \tag{3.20}$$

Dimostrazione È facile verificare che, comunque si scelga un vettore di potenziali π ed un ciclo C (con un verso di percorrenza), il costo del ciclo $c(C)$ è uguale al suo *costo ridotto* $c^{\pi}(C)$ (definito come in (3.17) usando i costi ridotti al posto dei costi): infatti, per ciascun nodo i del ciclo il corrispondente potenziale π_i appare due volte nella sommatoria, una per ciascuno dei due archi incidenti, e sempre con coefficienti opposti. Si consideri adesso il grafo residuo G_x con i costi definiti da c^{π}. Se (3.20) è vera, allora i costi (ridotti) su tutti gli archi di G_x sono non negativi. Quindi il costo ridotto di qualsiasi ciclo in G_x è non negativo, ovvero lo è il costo di qualsiasi ciclo aumentante, e quindi x è minimale. Viceversa, se x è minimale allora non esistono cicli aumentanti di costo negativo rispetto a x; esiste quindi un albero dei cammini minimi su G_x (con i costi originali, e, ad esempio, insieme di radici $R = N$), con il corrispondente vettore di etichette d che rispetta le condizioni di Bellman (3.7): ciò significa che

$$c_{ij} + d_i - d_j \geq 0 \qquad (i, j) \in A_x$$

ed è facile verificare che questo implica (3.20) prendendo $\pi = d$. ◇

Il teorema precedente suggerisce una variante dell'algoritmo che usa un vettore di potenziali π, inizializzato come $\pi = 0$, e risolve il problema (SPT) in G_x utilizzando i costi ridotti c^{π} associati agli archi invece dei costi originari. Si noti che alla prima iterazione i due insiemi di costi sono uguali, ed abbiamo già notato come in quel caso il grafo residuo abbia solamente archi di costo non negativo, per cui è possibile utilizzare algoritmi $SPT.S$ per determinare un albero dei cammini minimo, con complessità inferiore a $O(mn)$. Oltre al cammino P utilizzato per inviare il flusso, l'algoritmo SPT restituisce anche un vettore di etichette d: è facile dimostrare che se x ed π rispettano (3.20) (all'inizio dell'iterazione), allora anche $x \oplus \theta P$ e $\pi + d$ le rispettano (alla fine dell'iterazione).

Esercizio 3.35 *Dimostrare l'affermazione precedente.*

Pertanto, anche per questa variante dell'algoritmo si ottiene, tramite il Teorema 3.12, che tutti gli pseudoflussi generati sono minimali, e quindi che l'algoritmo è corretto. Il vantaggio è che ad ogni passo si calcola un albero dei cammini minimi su un grafo con archi di costo non negativo, e questo può essere ottenuto in $O(n^2)$ oppure $O(m \log n)$.

Esercizio 3.36 *Un ulteriore vantaggio di $SPT.S$ è che sarebbe possibile, in linea di principio, interrompere la computazione non appena un nodo j con $e_j < 0$ viene estratto da Q, evitando così di esplorare inutilmente parti del grafo. In questo caso, però, le etichette dei nodi che non sono ancora stati estratti da Q al momento della terminazione anticipata possono non soddisfare le condizioni di Bellman, e quindi $\pi + d$ potrebbe non soddisfare le condizioni (3.20). Si discuta come si possa gestire adeguatamente questa occorrenza.*

Oltre a suggerire implementazioni potenzialmente più efficienti dell'algoritmo, che possono essere utili in casi specifici (si veda ad esempio il paragrafo (3.5.2), queste considerazioni sono interessanti perché introducono ad un'ampia classe di algoritmi per (MCF), detti *primali-duali*, che sono tra i più efficienti in pratica. Questi sono basati sull'osservazione che, in uno pseudoflusso minimale, il costo ridotto di tutti gli archi (i, j) che non sono né saturi né vuoti ($0 < x_{ij} < u_{ij}$) deve essere zero; viceversa, dato

un vettore di potenziali π, il valore di x_{ij} su qualsiasi arco (i, j) tale che $c^{\pi}_{ij} = 0$ può essere fissato arbitrariamente, ottenendo sempre uno pseudoflusso minimale. Infatti, si può facilmente osservare che, nella variante sopra discussa, i costi ridotti calcolati rispetto a $\pi + d$ sono nulli per tutti gli archi dell'albero dei cammini minimi individuato.
Gli algoritmi di questa classe procedono quindi mantenendo una coppia (x, π) che rispetta (3.20). Nella *fase primale* si mantiene π fisso, e si cerca di modificare x in modo tale da ottenere un flusso ammissibile; per questo si usano tecniche per il problema di flusso massimo, ad esempio analoghe a quelle studiate nel paragrafo 3.3.4. Se si ottiene un flusso ammissibile l'algoritmo termina; altrimenti la fase primale determina informazione (ad esempio un opportuno taglio) che permette di modificare π nella *fase duale*, creando nuovi archi a costo ridotto nullo ove sia possibile inviare liberamente il flusso. Per ulteriori dettagli relativi a questa importante classe di approcci risolutivi si rimanda alla letteratura citata.

Esercizio 3.37 *Le condizioni (3.20) sono dette* condizioni degli scarti complementari*: si discutano le relazioni tra le diverse condizioni di ottimalità introdotte per (MCF) e quelle studiate per la* PL *in generale, e si analizzino le relazioni esistenti tra i diversi algoritmi introdotti in questo paragrafo e quelli per la* PL.

3.5 Problemi di accoppiamento

Sia $G = (O \cup D, E)$ un grafo bipartito non orientato, dove $O = \{ 1, \dots, n \}$ è l'insieme dei *nodi origine*, $D = \{ n + 1, \dots, n + d \}$ è l'insieme dei nodi destinazione, e $E \subseteq O \times D$, con $|A| = m$, è l'insieme dei lati, ai quali possono essere associati costi c_{ij}. Non è restrittivo supporre $n \leq d$.
Un *accoppiamento* (*matching*) M è un sottoinsieme di lati che non hanno nodi in comune. I lati in M sono detti *interni*, mentre i lati in $A \setminus M$ sono detti *esterni*. Dato un accoppiamento M, un nodo i è *esposto* rispetto a M se nessun lato di M incide in i, altrimenti i è detto *accoppiato*; indicheremo con O_M e D_M gli insiemi dei nodi rispettivamente in O e D che sono esposti. Nel caso in cui $|O| = |D|$, cioè $d = n$, M è un *accoppiamento perfetto* (o *assegnamento*) se nessun nodo è esposto, ovvero se $|M| = n$. Un esempio è fornito in figura 3.20. La *cardinalità* di un accoppiamento M è $|M|$, mentre il costo $C(M)$ di un accoppiamento M è la somma dei costi dei lati di M (si assume $C(\emptyset) = 0$). Dato un accoppiamento $M \neq \emptyset$, il lato $\{i, j\} \in M$ di costo massimo è detto *lato bottleneck* (collo di bottiglia) di M; il valore $V(M) = \max\{ c_{ij} \, : \, (i, j) \in M \}$ è detto il *valore bottleneck* di M.

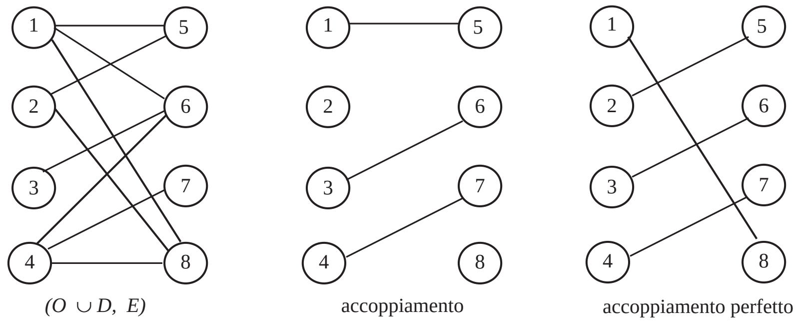


Figura 3.20: Esempi di accoppiamenti

Nel seguito studieremo i seguenti problemi:

1. *Accoppiamento di massima cardinalità* in G.

2. *Assegnamento di costo minimo*: si vuole determinare, tra tutti gli accoppiamenti perfetti in G, uno che abbia di costo minimo.

3. *Assegnamento di massima cardinalità bottleneck*: si vuole determinare, tra tutti gli accoppiamenti di massima cardinalità in G, uno che abbia valore bottleneck minimo, cioè tale che il massimo costo degli archi sia minimo.

3.5.1 Accoppiamento di massima cardinalità

Il problema di accoppiamento di massima cardinalità in un grafo $G = (O \cup D, E)$ può essere trasformato in un problema equivalente di flusso massimo con più sorgenti e pozzi sul grafo orientato $\bar{G} = (N, A)$ dove A contiene un arco per ogni lato di E, orientato dal nodo in O a quello in D. Ogni arco $(i, j) \in A$ ha capacità superiore $u_{ij} = 1$, O è l'insieme delle sorgenti, D è l'insieme dei pozzi, ed ogni sorgente/pozzo può immettere nella/prelevare dalla rete un'unità di flusso. Equivalentemente, si possono aggiungere a $\bar{G}$ una "super sorgente" s ed un "super pozzo" t, collegati rispettivamente a tutti i nodi di O e D da archi di capacità unitaria, e considerare il problema di flusso massimo da s a t. È facile verificare che l'insieme degli archi saturi in qualunque flusso ammissibile (intero) x in $\bar{G}$ forma un accoppiamento M in G la cui cardinalità è pari al valore v del flusso; viceversa, da un qualunque accoppiamento M si costruisce un flusso ammissibile. Nell'esempio in figura 3.21, relativo al grafo G di figura 3.20, è mostrato in (a) un accoppiamento M con $|M| = 3$, ed in (b) il corrispondente flusso ammissibile x su $\bar{G}$ con valore del flusso $v = 3$ (sono indicati solo i flussi diversi da zero).

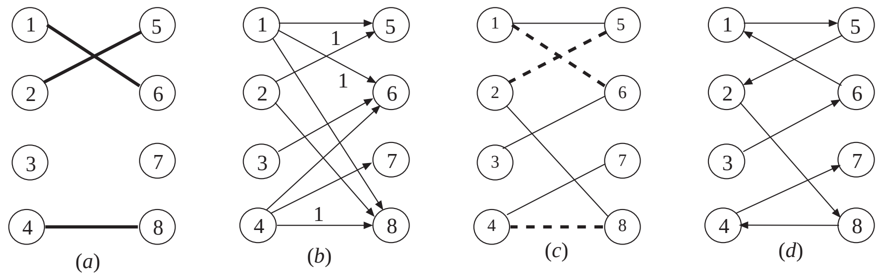


Figura 3.21: Flussi, accoppiamenti e cammini (alternanti) aumentanti

È quindi possibile risolvere il problema dell'accoppiamento di massima cardinalità in G applicando un qualsiasi algoritmo per il problema del flusso massimo (con più sorgenti e pozzi) in $\bar{G}$. Data la particolare struttura del problema, però, alcune operazioni degli algoritmi possono essere implementate in maniera più efficiente, o hanno un particolare significato che è possibile sfruttare ai fini algoritmici. Si consideri ad esempio il concetto di cammino aumentante sul grafo $\bar{G}$ rispetto ad un qualche flusso x che rappresenta un accoppiamento M, ossia tale che $x_{ij} = 1$ per $(i, j) \in A$ se e solo se $\{i, j\} \in M$. Un arco $(i, j) \in A$ è saturo se e solo se il corrispondente lato $\{i, j\} \in E$ è interno ad M. Siccome il grafo è bipartito, i nodi di qualsiasi cammino su $\bar{G}$ devono appartenere alternativamente ad O ed a D. Ma tutti i lati $\{i, j\} \notin M$ su G corrispondono ad archi vuoti su $\bar{G}$: quindi, tali archi possono essere utilizzati in un cammino aumentante solamente in modo concorde col loro verso, ossia da un nodo di O ad un nodo di D. Viceversa, tutti i lati $\{i, j\} \in M$ su G corrispondono ad archi saturi su $\bar{G}$: quindi, tali archi possono essere utilizzati in un cammino aumentante solamente in modo discorde al loro verso, ossia da un nodo di D ad un nodo di O. Da tutto questo segue che un cammino aumentante su $\bar{G}$ rispetto ad un flusso x corrisponde ad un *cammino alternante* su G rispetto all'accoppiamento M, ossia un cammino formato alternativamente da archi esterni ed archi interni rispetto a M.
Non tutti i cammini alternanti su G rappresentano però cammini aumentanti su $\bar{G}$; affinché questo accada, occorre anche che il cammino parta da un'origine esposta e termini in una destinazione esposta (in questo caso, il cammino alternante è detto *aumentante*). Infatti, le origini/destinazioni esposte sono quelle per cui non transita ancora nessun flusso: siccome ogni origine/destinazione ha "capacità" unitaria, le origini esposte sono i nodi nell'insieme S_x delle sorgenti "attive" e le destinazioni esposte sono i nodi nell'insieme T_x dei pozzi "attivi" (si veda il paragrafo 3.3.3). Per questo, qualsiasi cammino aumentante su $\bar{G}$ deve avere come primo nodo un'origine esposta e come ultimo nodo una destinazione esposta. Esiste quindi una corrispondenza biunivoca tra cammini aumentanti su $\bar{G}$ e *cammini alternanti aumentanti* su G. Un cammino alternante P su G è aumentante se, detti $P^+ = P \setminus M$ e $P^- = M \cap P$ l'insieme degli archi esterni e quello degli archi interni di P, si ha $|P^+| - |P^-| = 1$, ossia gli archi esterni sono esattamente uno in più di quelli interni.

Esempio 3.18: Cammini alternanti aumentanti

Le affermazioni precedenti possono essere facilmente verificate nell'esempio in figura 3.21; in (c) e (d) sono mostrati rispettivamente un cammino alternante aumentante su G rispetto all'accoppiamento M ed un cammino aumentante su $\bar{G}_x$ rispetto al flusso x.

Si noti che, dato un cammino (alternante) aumentante, la capacità del cammino è sempre 1. Questo corrisponde al fatto che, in questo caso, l'operazione di composizione $x' = x \oplus P'$, dove P' è un cammino aumentante su $\bar{G}$, corrisponde a

$$M' = M \oplus P = M \setminus P^- \cup P^+$$

dove P è un cammino alternante aumentante su G: in altre parole, l'operazione di composizione corrisponde a togliere da M i lati interni di P ed aggiungere quelli esterni di P. Siccome $|P^+| = |P^-| + 1$, si ha che $|M \oplus P| = |M| + 1$; infatti, il nuovo flusso x' ha valore $v' = v + 1$.

Esempio 3.19: Operazione di composizione
Proseguendo l'esempio di figura 3.21, ed applicando l'operazione di composizione aull'accoppiamento M mostrato in (a) ed al cammino P mostrato in (c), si ottiene il nuovo accoppiamento $M' = \{(1,5),(2,8),(3,6),(4,7)\}$. È immediato verificate che il nuovo accoppiamento corrisponde al flusso che si ottiene dal flusso mostrato in (b) inviando un'unità di flusso lungo il cammino aumentante mostrato in (d).

Con queste notazioni, possiamo costruire una versione specializzata dell'algoritmo 3.3.2 per risolvere il problema dell'accoppiamento di massima cardinalità.

```
procedure Accoppiamento-MaxCard( O , D , E , M ) {
  M = ∅;
  while( Cammino-Aumentante( O , D , E , M , P ) ) do
  Cambia-Accoppiamento( M , P );
  }
```

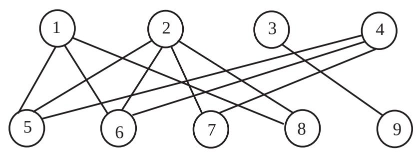
Procedura 3.7: Algoritmo *Accoppiamento-MaxCard*

L'inizializzazione, $M = \emptyset$, corrisponde a scegliere $x = 0$ come flusso iniziale. La procedura *Cammino-Aumentante* determina, se esiste, un cammino alternante aumentante: per questo è sufficiente visitare il grafo bipartito G partendo dai nodi di O_M e visitando alternativamente archi esterni e interni, il che corrisponde alla procedura *Visita* con semplici modifiche. Si noti che, rispetto al caso del flusso massimo, il controllo di ammissibilità di un arco è più semplice, e non è necessario determinare la capacità del cammino. Se alla fine della visita non si è raggiunto alcun nodo in D_M allora non esistono cammini aumentanti e l'accoppiamento è di massima cardinalità: ciò corrisponde al fatto che non esistono cammini aumentanti su $\bar{G}_x$, e quindi il flusso x ha valore massimo. In particolare, questo accade sicuramente qualora $O_M = \emptyset$, ossia se è già stato prodotto un accoppiamento di massima cardinalità. Se invece viene determinato un cammino alternante aumentante P, viene invocata la procedura *Cambia-Accoppiamento* che realizza l'operazione di composizione $M \oplus P$; tale operazione è analoga alla *Aumenta-Flusso* dell'algoritmo *Cammini-Aumentanti*, ma più semplice.

Esercizio 3.38 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Cammino-Aumentante *e* Cambia-Accoppiamento*.*

La complessità di *Accoppiamento-MaxCard* è $O(mn)$, in qualunque modo venga implementata la visita: infatti, la complessità della generica procedura *Cammini-Aumentanti* è $O(mnU)$, ma in questo caso $U = 1$. In effetti, è immediato verificare che la procedura termina dopo al più n iterazioni, ognuna delle quali richiede una visita del grafo e quindi costa $O(m)$.

Esercizio 3.39 *Si applichi* Accoppiamento-MaxCard *al grafo in figura qui accanto, fornendo ad ogni iterazione l'accoppiamento, l'albero della visita ed il cammino aumentante.*



3.5.2 Assegnamento di costo minimo

Analogamente al problema dell'accoppiamento di massima cardinalità, il problema dell'assegnamento costo minimo è equivalente al problema (MCF) sul grafo orientato $\bar{G}$ in cui le capacità degli archi sono unitarie, i costi degli archi sono quelli del problema di accoppiamento, ogni nodo in O produce un'unità di flusso ed ogni nodo in D consuma un'unità di flusso. La trasformazione è illustrata in figura 3.22 (b) per l'istanza in (a) (per chiarezza di visualizzazione non sono indicate le capacità degli archi, tutte pari a 1).

Analogamente al caso del problema dell'accoppiamento di massima cardinalità, è possibile specializzare gli algoritmi per il problema del flusso di costo minimo al caso particolare del problema dell'assegnamento di costo minimo. Nell'algoritmo basato sui cammini minimi successivi, ad esempio, si determina ad ogni passo un cammino aumentante di costo minimo che connette un nodo con eccesso di flusso ad uno con difetto di flusso: è immediato verificare che, nel caso del problema dell'assegnamento di costo minimo, ciò corrisponde a determinare il *cammino alternante aumentante di costo minimo* rispetto all'accoppiamento corrente M. Per fare questo si può utilizzare il grafo ausiliario $G_M = (N, A_M)$ con

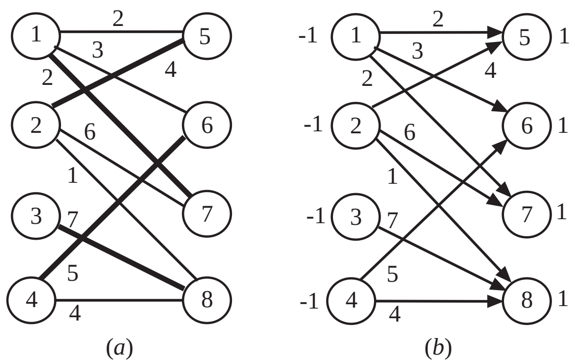


Figura 3.22: Trasformazione in un (MCF)

$$A_M = A_M^+ \cup A_M^- = \{\ (i,j)\ :\ \{i,j\} \in E \setminus M\ \} \cup \{\ (j,i)\ :\ \{i,j\} \in M\ \}\ ,$$

dove $(i,j) \in A_M^+$ ha costo c_{ij} mentre $(j,i) \in A_M^-$ ha costo $-c_{ij}$. È facile verificare che G_M è il grafo residuo $\bar{G}_x$ per lo pseudoflusso x corrispondente all'accoppiamento M. Un qualunque cammino orientato P_{st} da un nodo $s \in O_M$ ad un nodo $t \in D_M$ corrisponde ad un cammino alternante aumentante P, e viceversa; inoltre, per costruzione il costo di P_{st} in G_M è uguale al costo di P, definito come in (3.17). Inviare un'unità di flusso lungo un cammino aumentante corrisponde ad applicare l'operazione di composizione $M' = M \oplus P$ del paragrafo precedente; è facile verificare che risulta $C(M') = C(M) + C(P)$.

Esempio 3.20: Cammini alternanti aumentanti
Consideriamo il grafo G e l'accoppiamento M in figura 3.23(a) di costo $C(M) = 3$. Il grafo ausiliario G_M è descritto in figura 3.23(b). Al cammino orientato $P_{37} = \{ (3,8)\, , (8,2)\, , (2,7) \}$ nel grafo ausiliario in figura 3.23(b), avente costo 12, corrisponde nel grafo originario il cammino aumentante $P = \{ \{3,8\}\, , \{2,8\}\, , \{2,7\} \}$, avente anch'esso costo 12.

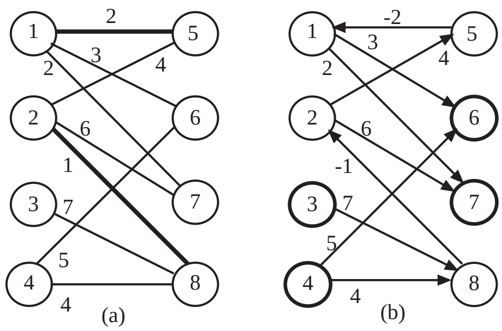


Figura 3.23: Cammini aumentanti su G e cammini su G_M

Quanto detto finora porta alla definizione del seguente algoritmo per la soluzione del problema.

```
procedure Assegnamento-MinCost( O , D , E , c , M ) {
  M = ∅;
  while( Cammino-AA-Minimo( G_M , P ) ) do
    Cambia-Accoppiamento( M , P , O_M , D_M , G_M );
}
```

Procedura 3.8: Algoritmo *Assegnamento-MinCost*

L'algoritmo parte dall'accoppiamento iniziale vuoto, corrispondente al flusso iniziale $x = 0$. Si noti che non è necessario eseguire la procedura *Inizializza* del paragrafo 3.4.3, anche in presenza di costi negativi, in quanto x è sicuramente minimale: il grafo residuo, essendo bipartito, non contiene nessun ciclo orientato. La procedura *Cammino-AA-Minimo* cerca di determinare un cammino alternante aumentante di costo minimo tra un nodo $s \in O_M$ ed un nodo $t \in D_M$, restituendo *vero* se ha successo e $false$ altrimenti, nel qual caso l'algoritmo termina. In particolare, restituisce $false$ se $O_M = \emptyset$, ossia se non esistono nodi esposti in O; altrimenti determina un albero dei cammini minimi con insieme di nodi radice O_M per G_M, col relativo vettore di etichette d e seleziona un nodo $t \in D_M$ (ad esempio, quello con etichetta d_t minore): se $d_t = \infty$ allora t non è connesso ad alcun nodo di O_M e la procedura restituisce $false$, altrimenti si è determinato il cammino desiderato. La procedura *Cambia-Accoppiamento* esegue l'operazione di composizione tra l'accoppiamento M corrente ed il cammino P, ed aggiorna gli insiemi dei nodi esposti, O_M e D_M, ed il grafo ausiliario G_M associato al nuovo accoppiamento.

Esercizio 3.40 *Si fornisca una descrizione formale, in pseudo-codice, delle procedure* Cammino-AA-Minimo *e* Cambia-Accoppiamento*.*

Quando l'algoritmo termina, se $|M| = |O| = |D|$ allora si è determinato un assegnamento di costo minimo, altrimenti non esistono accoppiamenti perfetti in G.

Esercizio 3.41 *Dimostrare l'affermazione precedente.*

Esempio 3.21: Esecuzione di *Assegnamento-MinCost*
In figura 3.24 sono mostrate due iterazioni dell'algoritmo.

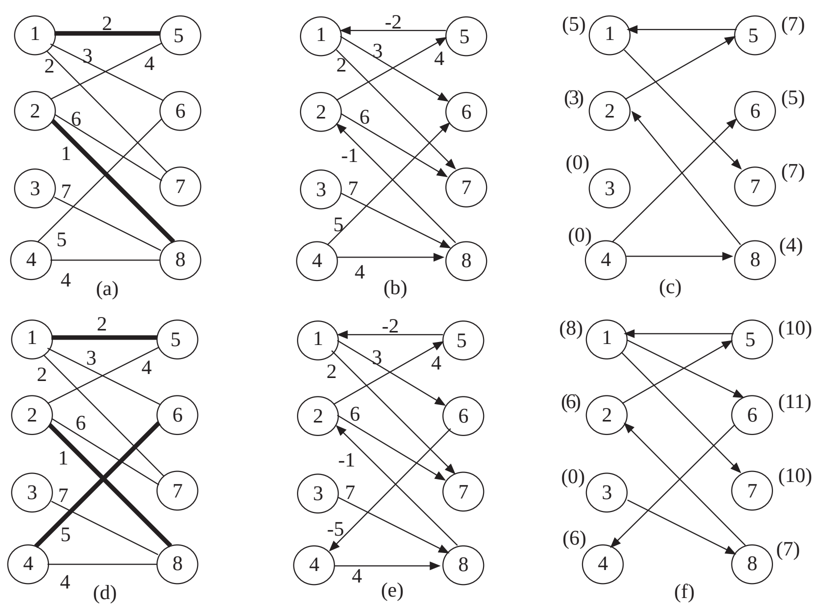


Figura 3.24: Due iterazioni di *Assegnamento-MinCost*

In (a) l'accoppiamento M corrente, in (b) il corrispondente grafo ausiliario G_M ed in (c) l'albero dei cammini minimi con insieme di radici $R = O_M = \{3, 4\}$, con le relative etichette ottime ai nodi. Essendo $D_M = \{6, 7\}$, si pone $t = 6$ poiché $d(6) = 5 < 7 = d(7)$, selezionando così il cammino alternante aumentante $P = \{\, \{4, 6\} \,\}$. In figura 3.24(d) è mostrato il nuovo accoppiamento $M' = M \oplus P = \{\, \{1, 5\}\, , \{2, 8\}\, , \{4, 6\} \,\}$, di cardinalità 3 e costo $C(M') = C(M)+C(P) = 3+5 = 8$, mentre in 3.24(e) ed (f) sono mostrati rispettivamente il corrispondente grafo ausiliario $G_{M'}$ e l'albero dei cammini minimi. Il cammino alternante aumentante di costo minimo è $P' = \{\, \{3, 8\}\, , \{2, 8\}\, , \{2, 5\}\, , \{1, 5\}\, , \{1, 7\} \,\}$, con $C(P') = d(7) = 10$. L'assegnamento ottimo, $M'' = M' \oplus P' = \{\, \{1, 5\}\, , \{2, 8\}\, , \{4, 6\} \,\} \setminus \{\, \{2, 8\}\, , \{1, 5\} \,\} \cup \{\, \{3, 8\}\, , \{2, 5\}\, , \{1, 7\} \,\}$ $= \{\, \{1, 7\}\, , \{2, 5\}\, , \{3, 8\}\, , \{4, 6\} \,\}$, di costo $C(M'') = C(M') + c(P') = 8 + 10 = 18$, è mostrato in Figura 3.22(a).

La correttezza dell'algoritmo deriva direttamente dalla correttezza della procedura *Cammini-Minimi-Successivi* per il problema del flusso di costo minimo; dall'analisi svolta per quella procedura risulta immediatamente che, se utilizziamo l'algoritmo *SPT.L.queue* per implementare *Cammino-AA-Minimo*, la complessità di *Assegnamento-MinCost* è $O(mn^2)$, essendo n il massimo numero di iterazioni. Con le tecniche descritte nel paragrafo 3.4.3 tale complessità può essere diminuita.
È anche possibile dimostrare che l'algoritmo *Assegnamento-MinCost* può essere usato anche per risolvere un problema più generale dell'assegnamento di costo minimo, ossia il problema dell'*accoppiamento di massima cardinalità e costo minimo*, nel quale si vuole determinare, tra tutti gli accoppiamenti di

massima cardinalità (anche se non necessariamente perfetti), quello di costo minimo. Per fare questo è solamente necessario garantire che, ad ogni iterazione, il nodo $t \in D_M$ selezionato come destinazione sia uno di quelli di etichetta minima, ossia $d_t = \min\{\, d_j \;:\; j \in D_M \,\}$, ovvero che il cammino alternante aumentante selezionato sia di costo minimo tra *tutti* i cammini che uniscono *un qualsiasi* nodo di O_M ad un nodo di D_M.

Esercizio 3.42 *Si dimostri l'affermazione precedente. Si discuta inoltre se esiste una versione di questo risultato che si applica al problema di (MCF) (o meglio ad una sua generalizzazione).*

3.5.3 Accoppiamento di massima cardinalità bottleneck

Il problema dell'accoppiamento di massima cardinalità bottleneck non è facilmente formulabile come problema di flusso di costo minimo, ma può essere facilmente risolto utilizzando come sottoprogrammi algoritmi visti nei paragrafi precedenti. Descriveremo le idee di base per il caso, più semplice, dell'assegnamento bottleneck, estendendole in seguito al caso più generale. Si supponga di conoscere il valore bottleneck ottimo

$$z = \min\{\, V(M) \;:\; M \text{ è un assegnamento in } G \,\} \;\;,$$

e si consideri il grafo parziale $G_v = (O, D, E_v)$, parametrico rispetto al valore reale v, dove

$$E_v = \{\; \{i, j\} \in E \;:\; c_{ij} \leq v \;\} \;\;.$$

Dalla definizione discende che se $v < z$, allora G_v non contiene accoppiamenti perfetti, altrimenti ($v \geq z$) G_v contiene almeno un assegnamento. Ciò suggerisce il seguente algoritmo:

1. si parte da un valore di v abbastanza piccolo, ossia tale che icuramente $v \leq z$ (ad esempio $v = \min\{\, c_{ij} \;:\; \{i, j\} \in E \,\}$);
2. si calcola un accoppiamento M di massima cardinalità in G_v: se $|M| = n$ ci si ferma, altrimenti si determina il più piccolo valore di v che permette di aggiungere archi ad E_v (cioè il minimo costo di un arco strettamente maggiore di v) e si itera.

Il processo di aggiunta di archi ad E_v viene iterato fino a che M non sia un accoppiamento perfetto, oppure sino a che $E_v = E$ e $|M| < n$: nel primo caso M è un assegnamento bottleneck, ossia ha valore $V(M) = v$ minimo tra tutti i possibili assegnamenti, mentre nel secondo caso il grafo G è privo di accoppiamenti perfetti. In questo caso, operando opportunamente, si può garantire che l'ultimo accoppiamento M prodotto sia bottleneck tra tutti gli accoppiamenti di massima cardinalità.

```
procedure Accoppiamento-Bottleneck( O , D , E , c , M , v )  {
  Inizializza( v , E_v ); M = ∅;
  do { Accoppiamento-MaxCard( O , D , E_v , M );
       if( |M| ≥ n ) then break;
       v = min{ c_ij : {i,j} ∉ E_v }; E_v = E_v ∪ { {i,j} : c_ij = v };
       } while( E_v ⊊ E );
}
```

Procedura 3.9: Algoritmo *Accoppiamento-Bottleneck*

La procedura *Inizializza* determina un opportuno valore v ed il corrispondente insieme di archi E_v. Se si vuole trovare un assegnamento è possibile scegliere

$$v = \max\{\, \min\{\, c_{ij} \;:\; \{i, j\} \in S(i)\} \;:\; i \in O \cup D \,\}$$

poiché per valori inferiori almeno un nodo risulterebbe isolato dagli altri, impedendo l'esistenza di un accoppiamento perfetto in G_v. Se invece si vuole risolvere il problema dell'accoppiamento bottleneck di massima cardinalità è possibile scegliere $v = \min\{\, c_{ij} \;:\; \{i, j\} \in E\}$, poiché per valori inferiori E_v è vuoto. La procedura *Accoppiamento-MaxCard* determina un accoppiamento di massima cardinalità in G_v. Si noti che, durante il processo iterativo, è possibile utilizzare come accoppiamento di partenza l'accoppiamento di massima cardinalità determinato all'iterazione precedente, che è sicuramente ancora un accoppiamento valido in quanto tutti gli archi che erano in E_v all'iterazione precedente ci sono anche in quella attuale (v è crescente). Poiché ad ogni iterazione si aggiunge ad E_v almeno un arco, non si effettueranno più di m iterazioni. Se si modifica la procedura *Accoppiamento-MaxCard* in modo da utilizzare come accoppiamento di partenza l'accoppiamento M_v in input, si determineranno al più n cammini aumentanti durante l'intera esecuzione dell'algoritmo; quindi che la complessità di tutte le chiamate ad *Accoppiamento-MaxCard* è $O(mn)$. Per determinare in modo efficiente il nuovo valore di v a ciascuna iterazione è sufficiente ordinare E all'inizio in modo tale che, globalmente, il costo di determinare il nuovo valore di v e i lati da aggiungere ad E_v sia $O(m)$. Siccome l'ordinamento costa $O(m \log n)$ e viene fatto una volta sola, la complessità di *Accoppiamento-Bottleneck* è $O(mn)$.

Esempio 3.22: Esecuzione di *Accoppiamento-Bottleneck*
Si vuole determinare un assegnamento bottleneck nel grafo G di Figura 3.25(a). Il valore di partenza è $v = 4$, corrispondente al minimo costo di lati uscenti dal nodo 4, poiché per valori inferiori il nodo 4 risulterebbe isolato. In Figura 3.25(b) vengono mostrati il grafo parziale G_4 e l'accoppiamento M con $|M| = 3$. Nella prima iterazione $v = 6$, ma l'aggiunta di $\{2, 7\}$ non modifica l'accoppiamento. All'iterazione successiva, $v = 7$; in Figura 3.25(c) viene mostrato il grafo G_7 ed il nuovo accoppiamento $M = \{\, \{1, 5\}\, , \{2, 8\}\, , \{3, 7\}\, , \{4, 6\}\, \}$ ottenuto dal precedente mediante il cammino aumentante $P = \{\, \{4, 6\}\, \}$. M_7 è perfetto: si tratta quindi di un assegnamento bottleneck, con valore $V(M) = 7$.

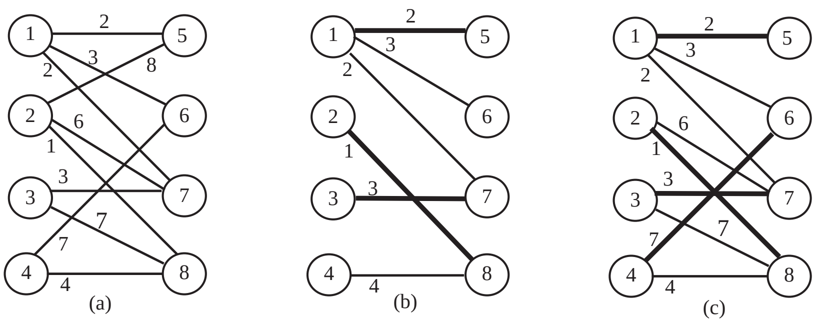


Figura 3.25: Alcune iterazioni di *Accoppiamento-Bottleneck*

Utilizzare la versione modificata di *Accoppiamento-MaxCard* che riparte dall'accoppiamento M in input non è soltanto un utile accorgimento che permette di velocizzare l'algoritmo, ma è necessario per la correttezza nel caso in cui non esistano accoppiamenti perfetti. Si consideri ad esempio il caso del grafo $G = (O \cup D, A)$ con $O = \{1, 2\}$, $D = \{3, 4\}$, $E = \{\, \{1, 3\}\, , \{2, 3\}\, \}$, $c_{13} = 1$ e $c_{23} = 10$. Al primo passo della procedura $v = 1$ e quindi $E_v = M = \{\, \{1, 3\}\, \}$. Al secondo passo $v = 10$ e quindi $E_v = E$. Se *Accoppiamento-MaxCard* partisse con l'accoppiamento iniziale $M = \emptyset$, potrebbe determinare come accoppiamento di massima cardinalità su G_v sia $M = \{\, \{1, 3\}\, \}$ che $M' = \{\, \{2, 3\}\, \}$, dato che entrambe sono accoppiamenti di cardinalità 1: chiaramente solo M è un accoppiamento bottleneck di massima cardinalità, per cui se venisse determinato M' l'algoritmo darebbe una risposta errata.

In generale, se non esistono assegnamenti ed il valore bottleneck è minore del massimo costo dei lati, l'algoritmo eseguirà una sequenza finale di iterazioni in cui cerca senza successo di costruire un accoppiamento di cardinalità maggiore di quello disponibile, finchè non esaurisce l'insieme dei lati e termina. Per la correttezza dell'algoritmo, è cruciale che durante queste iterazioni l'algoritmo non modifichi l'accoppiamento corrente costruendo un accoppiamento con la stessa cardinalità ma contenente lati a costo più alto: per garantire questo è sufficiente fare in modo che *Accoppiamento-MaxCard* riparta dal precedente accoppiamento M. Infatti, la procedura modificherà l'accoppiamento solo se può aumentarne la cardinalità: quindi, ad ogni iterazione di *Accoppiamento-Bottleneck* l'accoppiamento corrente M contiene solo lati il cui costo è minore od uguale del valore v' corrispondente all'ultima iterazione in cui la cardinalità di M è aumentata. Questo garantisce che, a terminazione, l'accoppiamento sia bottleneck anche nel caso in cui non sia perfetto.

Riferimenti Bibliografici

R.K. Ahuja, T.L. Magnanti, J.B. Orlin, "**Network flows. Theory, algorithms, and applications**", Prentice Hall, Englewood Cliffs, NJ (1993).

M.S. Bazaraa, J.J. Jarvis, H.D. Sherali, "**Linear programming and network flows**", Wiley, New York, NY (1990).

M. Pappalardo, M. Passacantando, "**Ricerca Operativa**", Edizioni Plus, Pisa (2010).

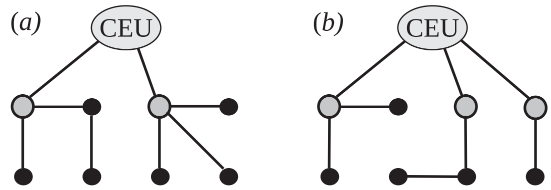
Capitolo 4

Ottimizzazione Combinatoria

4.1 Introduzione

L'*Ottimizzazione Combinatoria* (OC) studia i problemi di ottimizzazione in cui l'insieme ammissibile è definito in termini di strutture combinatorie, tra le quali svolgono sicuramente un ruolo di rilievo i grafi. La caratteristica fondamentale di tali problemi è quella di avere insiemi ammissibili *discreti*, a differenza ad esempio della PL, in cui l'insieme ammissibile è *continuo.* Le metodologie necessarie per risolvere problemi di OC sono pertanto spesso diverse da quelle utilizzate per risolvere problemi nel continuo.
Nei Capitoli 1 e 3 abbiamo già incontrato molti problemi di OC. In particolare, nel Capitolo 3 abbiamo descritto alcune importanti classi di problemi di OC che ammettono algoritmi risolutivi di complessità polinomiale. Moltissimi problemi di OC sono invece "difficili" ($\mathcal{NP}$-ardui, si veda l'Appendice A), ed è di questi problemi che ci occuperemo in questo capitolo e nei successivi. In effetti, i problemi di OC per i quali esistono algoritmi polinomiali hanno caratteristiche molto peculiari, e capita molto spesso che, non appena si introduca qualche variante, apparentemente trascurabile, ad un problema di OC "facile", il problema diventi subito difficile.

Esempio 4.1: Un problema di progetto di rete



Si consideri il problema della Banca Gatto & Volpe definito in 1.2.2.2. Si supponga adesso che la banca intenda aggiornare il sistema informativo, sostituendo la gestione attraverso terminali a caratteri che operano su un mainframe centralizzato con un più moderno sistema client-server in cui le filiali possono effettuare interrogazioni sul database centrale ed elaborare i risultati localmente. Chiaramente, questo tipo di cambiamento aumenta sensibilmente la quantità di informazione che viene inviata sulla rete. Data una soluzione, cioè un albero, tutto il traffico inviato al CEU da tutte le filiali appartenenti ad un certo sottoalbero deve passare per l'unico collegamento tra il nodo "radice", che rappresenta il CEU, e la filiale che funge da radice del sottoalbero. Quindi, sarebbero proprio questi collegamenti "critici" ad essere saturati per primi qualora la rete non fosse dimensionata opportunamente. Inoltre, in caso di un guasto ad una di queste linee tutte le filiali rappresentate da nodi nel sottoalbero corrispondente verrebbero disconnesse dal CEU. Ad ogni nodo i diverso dalla radice possiamo associare quindi un peso b_i corrispondente alla massima banda utilizzata dalla corrispondente filiale: per fare in modo che tutte le filiali abbiano sempre sufficiente banda per comunicare con il CEU, dobbiamo richiedere che nella soluzione del problema la somma dei pesi dei nodi in ciascun sottoalbero della radice sia al più Q, dove Q è la capacità dei collegamenti. Esempi di un albero di copertura non ammissibile ad ammissibile per $Q = 3$ sono presentati nella figura qui sopra, ove tutti i nodi hanno peso unitario ed i nodi evidenziati sono quelli collegati al CEU dai collegamenti "critici". Il corrispondente problema di OC, noto come *Constrained MST* (CMST) è una variante apparentemente trascurabile di (MST). Si tratta invece di un problema $\mathcal{NP}$-arduo (tranne per valori particolari di Q) mentre, come abbiamo visto, (MST) è polinomiale. Infatti, nella pratica istanze di (MST) su grafi con 10000 e più nodi non presentano alcuna difficoltà dal punto di vista della risolubilità, mentre istanze di (CMST) con poche centinaia di nodi sono a tutt'oggi per lo più insolubili.

Si può affermare che la grande maggioranza dei problemi di Ottimizzazione Combinatoria che si incontrano nella realtà sono difficili. Per questo, la conoscenza dei problemi di OC più rilevanti, e dei relativi algoritmi risolutivi, è una parte importante delle competenze specifiche di un esperto di

Ricerca Operativa. In effetti, il fatto che non esistano algoritmi generali in grado di risolvere problemi di *OC* di grande dimensione, mentre esistono metodologie generali che possono essere applicate caso per caso per la costruzione di algoritmi ad-hoc per un certo problema, giustifica in buona parte la necessità di formare specialisti con competenze di ottimizzazione. Inoltre, tipicamente gli algoritmi per problemi di *OC* "difficili" fanno ricorso ad algoritmi per problemi più "facili", quali la *PL* o i problemi di flusso su rete: questo giustifica l'interesse per la soluzione efficiente di problemi "facili", anche se la maggior parte dei modelli provenienti da applicazioni reali è relativa a problemi "difficili".

In questo capitolo introdurremo e discuteremo in modo generale alcune delle principali proprietà dei problemi di *OC* che hanno rilevanza per lo sviluppo di approcci algoritmici per tale classe di problemi. I Capitoli 5, 6 e 7 descriveranno invece alcune classi di algoritmi per problemi di *OC*.

4.2 Programmazione Lineare Intera (Mista)

I problemi di *Programmazione Lineare Intera* (*PLI*) si differenziano da quelli di *PL* unicamente per il fatto che le variabili possono assumere solamente valori interi. Tale *vincolo di integralità* ha però un enorme impatto. Innanzitutto, come abbiamo visto nel Capitolo 1, "l'espressività" del modello aumenta in maniera consistente: le variabili intere possono essere utilizzate per modellare condizioni logiche (decisioni "tutto o niente") e situazioni in cui le decisioni si prendono tra un numero finito di possibili alternative. Si può affermare che la grande maggioranza dei modelli utilizzati in pratica sono di *PLI*, in quanto nella maggior parte delle applicazioni reali esistono condizioni logiche ed è necessario compiere scelte discrete. Come abbiamo visto nel Capitolo 1, molti problemi di *OC* possono essere formulati come problemi di *PLI*. In effetti, nella pratica si tende a considerare sostanzialmente coincidenti le due classi dei problemi. Questo è dovuto a due ragioni concomitanti:

- da una parte, i problemi di *OC* vengono normalmente formulati come problemi di *PLI*, e buona parte degli approcci risolutivi per i problemi di *OC* si basa su tali formulazioni;

- d'altra parte, quasi tutte le tecniche efficienti per la soluzione di problemi di *PLI* si fondano sull'individuazione e sullo sfruttamento di strutture combinatorie specifiche nel modello *PLI*, ossia sull'individuazione di (sotto)problemi di *OC* corrispondenti al modello di *PLI* o a sue parti.

Si consideri ad esempio il problema (MST): per tale problema abbiamo fornito sia una formulazione in termini di *OC*, nella quale l'insieme ammissibile è definito come l'insieme di tutti gli alberi di copertura di un grafo dato, sia una formulazione in termini di *PLI*, in cui l'insieme ammissibile è definito come l'insieme di tutti i vettori in $\{0,1\}^m$ ($m = |A|$) che rispettano un certo insieme di vincoli lineari. Tali vincoli assicurano che tutte le soluzioni ammissibili del problema possano essere interpretate come vettori di incidenza di sottografi connessi del grafo originario, e la funzione obiettivo assicura che tra tutti questi venga selezionato il vettore di incidenza di un albero di copertura di costo minimo.

Esistono quindi una forte connessione tra i problemi di *OC* e quelli di *PLI*. Le due classi non sono però completamente coincidenti. Da una parte, la *PLI* fornisce un potente linguaggio per formulare in modo uniforme sia problemi di *OC* definiti su strutture molto diverse, ma anche problemi che può essere molto difficile ricondurre a problemi di *OC*: la *PLI* è in qualche senso "più espressiva" di *OC*. D'altra parte, possono esistere molte formulazioni *PLI* diverse di uno stesso problema di *OC*, e la formulazione come problema di *OC* è spesso "più informativa" delle formulazioni *PLI*, nel senso che può essere più facile derivare proprietà utili per la soluzione del problema lavorando direttamente sulla sua struttura combinatoria piuttosto che sulle sue formulazioni in termini di *PLI*.

Visto che ai problemi di *PLI* si possono applicare le stesse trasformazioni che abbiamo visto nel Capitolo 2 per i problemi di *PL*, possiamo assumere che i problemi di *PLI* siano esprimibili in forme standard analoghe a quelle già introdotte, ad esempio

$$(PLI) \qquad \max\{\, cx \;:\; Ax \le b \,,\; x \in \mathbb{Z}^n \,\} \;.$$

Si parla inoltre di problemi di *Programmazione Lineare Mista* (PLM) quando solamente alcune delle variabili sono vincolate ad essere intere: tali problemi hanno quindi la forma

$$(PLM) \qquad \max\{\ c'x' + c''x''\ :\ A'x' + A''x'' \leq b\ ,\ x' \in \mathbb{Z}^n\ \}\ .$$

Quasi tutti gli approcci per la *PLI* che descriveremo possono essere generalizzati alla *PLM*, spesso solamente al costo di complicazioni nella descrizione. Per semplicità ci riferiremo quindi sempre a problemi di *PLI*.

4.2.1 Il rilassamento continuo

Il motivo principale per cui è ampiamente diffusa la pratica di formulare e risolvere problemi di OC come problemi di PLI risiede nel fatto che, per questi ultimi, si possono utilizzare i potenti risultati teorici e le efficienti metodologie algoritmiche relative ai problemi di *PL*. Descriviamo adesso brevemente le principali relazioni tra la *PLI* e la *PL*; questo argomento sarà poi ripreso in maggiore dettaglio nel Capitolo 6. L'insieme ammissibile del problema (PLI),

$$\mathcal{F} = \{\ x \in \mathbb{Z}^n\ :\ Ax \leq b\ \}\ ,$$

ha fondamentalmente le caratteristiche di quello mostrato in Figura 4.1(a): i vincoli lineari $Ax \leq b$ definiscono un poliedro convesso

$$\bar{\mathcal{F}} =\ \{\ x \in \mathbb{R}^n\ :\ Ax \leq b\ \}\quad ,$$

e l'insieme ammissibile è formato dall'intersezione tra la "griglia" dei punti a coordinate intere e $\bar{\mathcal{F}}$, ossia da tutti i punti a coordinate intere che appartengono al poliedro (punti bianchi). Essendo formato da punti isolati, $\mathcal{F}$ non è convesso, il che spiega sostanzialmente la difficoltà dei problemi *PLI*.

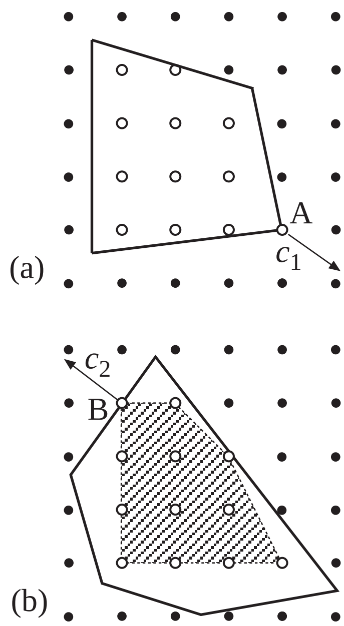


Figura 4.1: *PL* e *PLI*

In questa trattazione non spiegheremo in dettaglio perchè la non convessità renda il problema difficile, ma ci limiteremo ad illustrare i principi generali con un esempio. Seguiranno poi ulteriori commenti. Consideriamo a tal fine la minimizzazione di una funzione non convessa su $\mathbb{R}^n$: questo compito è in generale difficile quanto minimizzare (o massimizzare) una funzione lineare su un insieme non convesso. Infatti, è possibile riformulare (quasi) tutti i problemi di *PLI* come problemi di minimizzazione di funzioni non convesse; si noti che abbiamo visto esempi della trasformazione inversa, cioè da problemi di minimizzazione non convessa a problemi di *PLI*, nel Capitolo 1.

Esercizio 4.1 *Si dimostri formalmente l'equivalenza tra la* PLI *e la minimizzazione di una funzione non convessa (suggerimento: si determini un* singolo vincolo non lineare *equivalente alla richiesta* $x \in \{0,1\}$*, si estenda l'idea a* $x \in \{0,1\}^n$ *e poi a* $x \in \mathbb{Z}^n$*, ed infine si sostituisca il vincolo con un'adeguata modifica della funzione obiettivo, sotto opportune ipotesi se necessario).*

In Figura 4.2 sono mostrate un esempio di funzione convessa (a) e uno di funzione non convessa (b) in una sola variabile: con x^* sono indicati i *minimi globali* delle funzioni, ossia i punti cui corrisponde la minimizzazione di tali funzioni. Con x_1 ed x_2 sono indicati due *minimi locali* della funzione non convessa che non sono anche minimi globali; invece, tutti i minimi locali di qualsiasi funzione convessa sono anche minimi globali. In generale, determinare un minimo locale di una funzione con opportune proprietà di regolarità, ad esempio differenziabile con continuità, è "facile": l'informazione al primo ordine sulla funzione indica "da che parte andare". Nel caso non convesso, però, una volta determinato un minimo locale non si ha nessuna indicazione sull'esistenza di altri minimi locali (migliori) e sulla loro posizione. Questi concetti saranno comunque ripresi nel Capitolo 5.

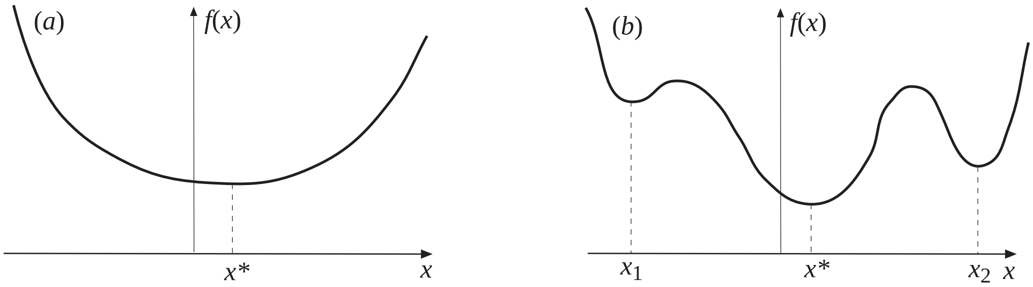


Figura 4.2: Funzioni convesse e non convesse

Dato che $\mathcal{F}$ è contenuto in $\bar{\mathcal{F}}$, quest'ultimo fornisce "un'approssimazione" di $\mathcal{F}$ che può essere sfruttata algoritmicamente. Si consideri infatti il *rilassamento continuo* di (PLI)

$$(RC) \qquad \max \{ \; cx \; : \; Ax \leq b \; \} \;\; ,$$

cioè il problema di PL corrispondente al rilassamento dei vincoli di integralità $x \in \mathbb{Z}^n$. Questo problema può essere efficientemente risolto, e permette di derivare informazione sul problema originario. Ad esempio, il valore ottimo della sua funzione obiettivo, $z(RC)$, fornisce una valutazione superiore del valore ottimo della funzione obiettivo di (PLI), ossia $z(RC) \geq z(PLI)$. L'utilità di questa relazione risiede nel fatto che $z(RC)$, al contrario di $z(PLI)$, è efficientemente calcolabile in quanto (RC) è facile. Inoltre, è immediato verificare il seguente risultato:

Lemma 4.1 *Sia x^* una soluzione ottima di (RC): se $x^* \in \mathbb{Z}^n$, ossia x^* è ammissibile per (PLI), allora x^* è ottima per (PLI).*

Un caso in cui si verificano le condizioni del Lemma 4.1 è mostrato in Figura 4.1(a), se la funzione obiettivo è c_1: è immediato verificare geometricamente che il punto "A", la soluzione ottima del rilassamento continuo, è intera ed è anche la soluzione ottima del problema di *PLI*. Il rilassamento continuo fornisce quindi un modo per tentare di calcolare una soluzione ottima per (PLI), ed in ogni caso fornisce una valutazione superiore del valore $z(PLI)$.

Esempio 4.2: Valutazioni superiori ed inferiori
Si consideri ad esempio il problema della Pintel: abbiamo visto che il suo rilassamento continuo ha soluzione ottima $(4, 1)$ (in unità di 100000 processori). Se consideriamo il vincolo di integralità sui wafers tale soluzione non è ammissibile, ma ci fornisce comunque una stima per eccesso del massimo ricavo disponibile, pari a 220 milioni di dollari. Si consideri adesso la soluzione ammissibile $(w_P, w_C) = (2666, 334)$ corrispondente a $(x_P, x_C) = (3.995, 1.002)$: tale soluzione permette un ricavo di 219.99 milioni di dollari. Possiamo quindi affermare che la soluzione ammissibile fornisce, alla peggio, un ricavo inferiore di 10000\$ al massimo possibile, ossia più del 99.995% del massimo ricavo possibile. Per la maggior parte degli scopi pratici, determinare una soluzione di questo tipo può essere considerato equivalente ad aver risolto il problema. Si noti che l'aver determinato la soluzione non è di per sé sufficiente: quello che permette di "esserne soddisfatti" è l'essere in grado di valutarne la "qualità", il che è reso possibile dalla valutazione superiore fornita dal rilassamento continuo.

4.2.2 Formulazioni di PL equivalenti per la PLI

Un'importante osservazione è che *lo stesso insieme* ammissibile per un problema di *PLI* può essere specificato attraverso poliedri "diversi". Ciò è mostrato in Figura 4.1(b), dove il poliedro mostrato (in tratto continuo) definisce lo stesso insieme di soluzioni ammissibili di quello in Figura 4.1(a) pur essendo "diverso". Esistono quindi *formulazioni diverse di uno stesso problema di PLI.* Queste formulazioni sono equivalenti per quanto riguarda il problema di *PLI*, ma non per quanto riguarda i rilassamenti continui. Risulta infatti intuitivamente chiaro come la valutazione superiore di $z(PLI)$ fornita da $z(RC)$ sia tanto migliore quanto più il poliedro $\bar{\mathcal{F}}$ risulti "aderente" all'insieme ammissibile $\mathcal{F}$ di (PLI). In effetti questa nozione dipende anche dalla funzione obiettivo: ad esempio, come abbiamo visto il poliedro di Figura 4.1(a) è "buono" se la funzione obiettivo è c_1, mentre se la funzione obiettivo fosse c_2 allora sarebbe il poliedro di Figura 4.1(b) ad essere "buono", in quanto la soluzione ottima del corrispondente rilassamento continuo è il punto "B", che è anche la soluzione ottima del problema intero. Viceversa, nessuno dei due poliedri è "buono" per l'altra funzione obiettivo: è immediato verificare che le soluzioni ottime del rilassamento continuo non sono intere, e che $z(RC) > z(PLI)$. Tra tutti i rilassamenti continui, ne esiste comunque uno "buono" per *qualsiasi* funzione obiettivo, in quanto "completamente aderente" a $\mathcal{F}$. Infatti, è possibile dimostrare che se tutti gli elementi della matrice A e del vettore b sono razionali (interi)[1], allora *l'inviluppo convesso* di $\mathcal{F}$

$$\tilde{\mathcal{F}} = Conv(\mathcal{F})$$

è un poliedro, cioè può essere rappresentato da un sistema finito di disequazioni $\tilde{A}x \leq \tilde{b}$. In Figura 4.1(b), il poliedro tratteggiato è l'inviluppo convesso dei punti ammissibili. Quanto appena detto sembrerebbe contraddire l'affermazione secondo cui i problemi di *PLI* sono difficili. Infatti, il problema

$$(\tilde{RC}) \qquad \max\{\ cx\ :\ \tilde{A}x \leq \tilde{b}\ \}$$

è chiaramente un rilassamento di (PLI), in quanto $\mathcal{F} \subset \tilde{\mathcal{F}}$, ma è possibile dimostrare che $\tilde{\mathcal{F}}$ gode della seguente *proprietà di integralità*:

Definizione 4.1 *Un poliedro $\mathcal{P}$ (non vuoto) gode della* proprietà di integralità *se vale una delle due seguenti definizioni equivalenti:*

- *tutti i vertici di $\mathcal{P}$ hanno coordinate intere;*
- *il problema* $\max\ \{\ cx\ :\ x \in \mathcal{P}\ \}$ *ammette una soluzione ottima intera per* qualsiasi *scelta del vettore $c \in \mathbb{R}^n$ per cui il problema non risulti superiormente illimitato.*

Esercizio 4.2 *Dimostrare l'equivalenza delle due condizioni nella definizione precedente (suggerimento: un'implicazione è ovvia, per l'altra si usi il fatto che ad ogni vertice del poliedro è associata almeno una base B di cui il vertice è la soluzione di base associata, e si usi la matrice di base A_B per costruire un vettore c tale per cui quel vertice è ottimo).*

Come sappiamo dalla teoria della *PL*, esiste una soluzione ottima x^* di $(\tilde{RC})$ che giace su un vertice di $\tilde{\mathcal{F}}$: siccome tutti i vertici di $\tilde{\mathcal{F}}$ hanno coordinate intere, x^* è anche una soluzione ottima di (PLI). Di conseguenza, è possibile risolvere (PLI) al costo della soluzione del problema di *PL* $(\tilde{RC})$.

Abbiamo visto (nel Teorema 3.11) che la formulazione "naturale" del problema di flusso di costo minimo, e quindi di tutti i problemi del Capitolo 3 che ad esso possono essere ricondotti, gode della proprietà di integralità (purchè le capacità degli archi e i deficit dei nodi siano interi). Come vedremo nel paragrafo 5.1.3, questo vale anche per il problema MST: si può quindi affermare che questa proprietà, ossia il fatto di trovarsi "al confine" tra l'ottimizzazione discreta e quella continua, sia ciò che rende "facili" i problemi del Capitolo 3. Per il generico problema di *PLI*, però, la rappresentazione di $\tilde{\mathcal{F}}$, ossia l'insieme dei vincoli $\tilde{A}x \leq \tilde{b}$ che lo definiscono, non è nota: tutto quello di cui si dispone è la sua rappresentazione "approssimata" data dai vincoli $Ax \leq b$. In generale è possibile dimostrare che, per un problema di (PLI) qualsiasi, è "difficile" rappresentare $\tilde{\mathcal{F}}$. Formalizzeremo meglio questo discorso introducendo alcuni concetti che risulteranno utili in seguito.

4.2.3 Diseguaglianze valide

Una diseguaglianza $dx \leq \delta$ si dice *valida* per $\tilde{\mathcal{F}}$ se è soddisfatta da ciascun punto $x \in \tilde{\mathcal{F}}$; equivalentemente, si può dire che $dx \leq \delta$ è valida per (PLI) se è soddisfatta da tutte le soluzioni (intere) del problema. Si noti che tutte le disequazioni $\tilde{A}_i x \leq \tilde{b}_i$ che definiscono $\tilde{\mathcal{F}}$ sono valide; $\tilde{A}$ e $\tilde{b}$ formano una rappresentazione minimale di $\tilde{\mathcal{F}}$ se la rimozione di una qualsiasi di queste disequazioni definisce un poliedro che contiene strettamente $\tilde{\mathcal{F}}$. In generale, non è necessario disporre di tutta la rappresentazione di $\tilde{\mathcal{F}}$ per risolvere $(\tilde{RC})$; è sufficiente essere in grado di risolvere il seguente *problema di separazione*

[1] Questi sono comunque i numeri che i calcolatori digitali trattano efficientemente; mediante sistemi di calcolo simbolico è possibile trattare anche numeri irrazionali ma con un'efficienza enormemente minore, il che ne limita fortemente l'uso in applicazioni come quelle descritte in queste note.

Dato $\bar{x} \in \mathbb{R}^n$, esiste una diseguaglianza valida $dx \leq \delta$ per $\tilde{\mathcal{F}}$
che non è soddisfatta da $\bar{x}$, ossia tale che $d\bar{x} > \delta$?

Il problema di separazione permette di determinare se un dato punto $\bar{x}$ appartiene o no a $\tilde{\mathcal{F}}$. Nel caso in cui $\bar{x} \notin \tilde{\mathcal{F}}$ viene richiesta una "dimostrazione" della non appartenenza sotto forma di una diseguaglianza che separa $\bar{x}$ da $\tilde{\mathcal{F}}$. Una tale diseguaglianza viene anche detta un *taglio* per $\bar{x}$. Per meglio comprendere l'utilità del problema di separazione, si assuma di avere una rappresentazione "approssimata" $A^{(0)}x \leq b^{(0)}$ di $\tilde{\mathcal{F}}$, ad esempio quella fornita dal sistema $Ax \leq b$, e di risolvere il corrispondente rilassamento continuo. Sia $x^{(0)}$ la soluzione ottima del rilassamento, e si risolva il corrispondente problema di separazione: se $x^{(0)} \in \tilde{\mathcal{F}}$ allora è una soluzione ottima per $(\tilde{RC})$, e quindi $cx^{(0)} = z(PLI)$, altrimenti viene determinato un taglio $dx \leq \delta$ per $x^{(0)}$. Si consideri quindi il sistema $A^{(1)}x \leq b^{(1)}$ in cui

$$A^{(1)} = \begin{bmatrix} A^{(0)} \\ d \end{bmatrix} \qquad \text{e} \qquad b^{(1)} = \begin{bmatrix} b^{(0)} \\ \delta \end{bmatrix} \;;$$

$A^{(1)}$ e $b^{(1)}$ definiscono una nuova rappresentazione di $\tilde{\mathcal{F}}$ "meno approssimata" della precedente; infatti, almeno il punto $x^{(0)}$ non ne fa più parte. Risolvendo il nuovo rilassamento di (PLI) corrispondente a questo nuovo sistema si otterrà una nuova soluzione $x^{(1)} \neq x^{(0)}$, e si potrà iterare il procedimento fino a determinare una soluzione ottima di $(\tilde{RC})$.

Disponendo di un algoritmo efficiente per risolvere il problema di separazione rispetto a $\tilde{\mathcal{F}}$, si disporrebbe quindi di un algoritmo per risolvere $(\tilde{RC})$, e quindi (PLI). In effetti, è possibile dimostrare che, con tecniche in parte analoghe a quella sopra accennata (ma che utilizzano un diverso modo per risolvere i problemi di *PL*, detto *metodo degli ellissoidi*), è possibile risolvere in tempo polinomiale qualunque problema di *PL* su un poliedro $\mathcal{P}$, anche se definito da un numero esponenziale di disequazioni, purché si disponga di un *separatore polinomiale* per $\mathcal{P}$, ossia si sia in grado di risolvere il corrispondente problema di separazione in tempo polinomiale. La conseguenza di questo risultato è che, dato un problema $\mathcal{NP}$-arduo, il problema di separazione associato all'inviluppo convesso delle soluzioni intere di una qualunque sua formulazione in termini di *PLI* è anch'esso $\mathcal{NP}$-arduo.
Può essere utile discutere un esempio in qualche modo inverso, in cui un problema la cui formulazione *PLI* "esatta" richiede un numero esponenziale di vincoli risulta ciò nonostante facilmente risolubile. Il problema è (MST): si può infatti dimostrare che la formulazione presentata nel paragrafo 1.2.2.2, che usa le disuguaglianze (1.6) (dette *cutset inequalities*), individua esattamente l'inviluppo convesso delle soluzioni intere del problema (si veda il paragrafo 5.1.3). Il fatto che il problema ammetta algoritmi risolutivi polinomiali corrisponde al fatto che esista un separatore polinomiale per le disuguaglianze (1.6). Si consideri infatti una soluzione x^* (possibilmente frazionaria): vogliamo verificare se esiste $S \subseteq V$ a cui corrisponde un vincolo è violato, ossia tale che risulti

$$\textstyle\sum_{i \in S, j \notin S} x^*_{ij} < 1 \;\; .$$

Per determinarlo, si può determinare il sottoinsieme S a cui corrisponde la *minima* di tali sommatorie. A tal fine, notiamo che la sommatoria che compare nel membro sinistro della relazione sopra riportata corrisponde alla *capacità del taglio* $(S, V \setminus S)$, in cui x^*_{ij} è interpretata come la capacità del lato $\{i, j\}$. Supponiamo quindi di determinare il taglio (V', V'') di capacità minima rispetto a quelle capacità: se la capacità del taglio è minore di 1, allora abbiamo individuato una specifica disuguaglianza violata, mentre se è maggiore od uguale ad 1 allora non esiste nessuna disuguaglianza violata. Il problema del taglio di capacità minima può essere risolto in tempo polinomiale (si veda il paragrafo 3.3), e quindi esiste un separatore polinomiale per le cutset inequalities. (MST) ammette infatti algoritmi polinomiali (si veda il paragrafo B.4). In pratica, gli algoritmi combinatori per (MST) risultano notevolmente più efficienti rispetto alla soluzione di una sequenza di problemi di *PL* e di separazione. L'esempio serve principalmente a sottolineare la fondamentale relazione tra i due concetti.

Esercizio 4.3 *I problemi di taglio di capacità minima per i quali abbiamo discusso algoritmi risolutivi efficienti sono definiti su grafi orientati e sono relativi ad una specifica coppia di nodi s e t. Si discuta come adattarli alla definizione di un separatore per (MST).*

Per riassumere, l'esistenza di una formulazione di *PL* equivalente a ciascun problema di *PLI* non rende la *PLI* facile: la formulazione esiste, ma non abbiamo nessun modo efficiente per generarla, e neanche per generarne una parte sufficiente a caratterizzare la soluzione ottima (si noti che basterebbe una base ottima). Questo però suggerisce alcuni interessanti approcci per la *PLI*, detti *metodi poliedrali*, che possono risultare molto utili per la soluzione di ampie classi di problemi. Questi metodi verranno descritti più in dettaglio nel seguito, ma la loro struttura di base può essere descritta nel modo seguente: anche se il problema di separazione per $\tilde{\mathcal{F}}$ è "difficile", accade sovente che molte delle disequazioni importanti che lo caratterizzano (faccette) possano essere determinate efficientemente. In altri termini, è possibile risolvere un rilassamento del problema originale "più debole" di $(\tilde{RC})$, ma che comunque comprende molta più "informazione" sul problema originario rispetto ad una formulazione *PLI* "naturale", quali quelle che abbiamo visto nel Capitolo 1. Questo rilassamento fornisce quindi valutazioni superiori di $z(PLI)$ più accurate di quelle fornite da formulazioni più semplici, e questo può rendere possibile risolvere (PLI) in modo più efficiente.

4.3 Dimostrazioni di ottimalità

In generale, il processo di soluzione di un qualsiasi problema di ottimizzazione, ad esempio della forma

$$(P) \qquad \max \{ \, c(x) \; : \; x \in X \, \} \quad ,$$

può essere considerato come composto di due parti distinte:

- individuazione di una soluzione ottima x^*;
- individuazione di una valutazione superiore di $z(P)$ che *dimostri* l'ottimalità di x^*, ossia un valore $\bar{z}$ per il quale sia garantito che $z(P) \leq \bar{z}$, ma per il quale risulti anche $c(x^*) = \bar{z}$, in modo tale che
$$\bar{z} = c(x^*) \leq z(P) \leq \bar{z} \quad .$$

In molti algoritmi visti nei capitoli precedenti le valutazioni superiori (o inferiori) erano esplicitamente descritte. In altri le dimostrazioni di ottimalità non facevano uso esplicito di valutazioni superiori (o inferiori, per problemi di minimo), ma tali valutazioni potrebbero essere costruite e mostrate.

Per valutazioni superiori disponibili in modo esplicito, si considerino i problemi dell'albero dei cammini minimi, del Flusso Massimo e di *PL*. Nel primo caso, è facile dimostrare, usando il Teorema 3.3 e la definizione della funzione obiettivo, che una valutazione inferiore del costo dell'albero ottimo è data dalla somma delle etichette associate ai nodi per qualsiasi vettore di etichette che rispetti le condizioni di Bellman: quando l'algoritmo termina, la valutazione inferiore è pari al costo dell'albero individuato. Nel secondo caso, una valutazione superiore del valore del massimo flusso è fornita dalla capacità di un qualsiasi taglio che separi la sorgente dal pozzo (Teorema 3.5): al termine, l'algoritmo ha costruito un taglio di capacità pari al valore del flusso determinato, che risulta quindi massimo. Nel caso della *PL*, una valutazione superiore del valor della funzione obiettivo di qualsiasi soluzione primale ammissibile è data dal valore della funzione obiettivo di qualsiasi soluzione dual ammissibile (Teorema 2.9): al termine, gli algoritmi del Simplesso hanno individuato una coppia di soluzioni ammissibili per il primale ed il duale con lo stesso valore di funzione obiettivo, e quindi hanno una dimostrazione esplicita di ottimalità per la soluzione primale (e per quella duale). Nel caso di (MCF), valutazioni sul valore ottimo della funzione obiettivo possono essere ricavate utilizzando argomenti duali, anche se non ne abbiamo fatto esplicitamente uso. Per (MST) questo è mostrato nel paragrafo 5.1.3. Ad una più attenta ispezione, le valutazioni superiori (o inferiori) descritte risultano derivare dalla teoria della dualità della Programmazione Lineare, che è infatti uno dei metodi più potenti e generali per derivare condizioni di ottimalità. Nel Capitolo 6 vedremo comunque una tecnica che la generalizza.

Per i problemi $\mathcal{NP}$-ardui di *OC* non si hanno, in generale, tecniche per determinare efficacemente valutazioni esatte del valore della funzione obiettivo del problema. Un diverso modo di vedere la cosa è il seguente: per quasi tutti i problemi di *OC*, la difficoltà del problema consiste fondamentalmente nella determinazione del *valore ottimo* della funzione obiettivo. Infatti, moltissimi problemi di *OC* godono di una proprietà chiamata *auto-riducibilità*, dalla quale segue che se esistesse un algoritmo efficiente per determinare il valore ottimo della funzione obiettivo, allora esisterebbe un algoritmo efficiente per determinare una soluzione ottima del problema.

Senza formalizzare completamente questi concetti, ne illustriamo l'uso con un esempio. Si consideri ancora (CMST), e si supponga di avere un algoritmo $\mathcal{A}$ in grado di calcolare, data una qualunque istanza I di (CMST), il valore ottimo della funzione obiettivo $z(I)$, senza però fornire una corrispondente soluzione ottima. Vediamo come sia possibile utilizzare $\mathcal{A}$ per costruire una soluzione ottima del prolema.

Per prima cosa, si utilizza $\mathcal{A}$ per calcolare $z^* = z(I)$. A questo punto, si seleziona un qualunque lato tra due nodi i e j (entrambe diversi dalla radice) e si verifica, utilizzando l'algoritmo $\mathcal{A}$, se $\{i, j\}$ fa oppure no parte di una soluzione ottima del problema. Per questo, è sufficiente construire una nuova istanza I' in cui il lato $\{i, j\}$ è cancellato, e calcolare $z(I')$ utilizzando $\mathcal{A}$. Se $z(I') = z^*$, allora esiste effettivamente una soluzione ottima di I in cui $\{i, j\}$ non è presente: è quindi possibile cancellare definitivamente $\{i, j\}$. Se invece $z(I') > z^*$, allora $\{i, j\}$ fa parte di qualsiasi soluzione ottima di I. È facile costruire una nuova istanza I'' di (CMST) corrispondente al fissare $\{i, j\}$ come parte della soluzione ottima: per questo, è sufficiente "accorpare" i e j in un nuovo nodo h, con $b_h = b_i + b_j$, fissando il costo dei lati $\{h, k\}$ per $k \neq i$ e $k \neq j$ al minimo tra c_{ik} e c_{jk}. È immediato verificare che ogni soluzione ottima di (CMST) per questa nuova istanza corrisponde ad una soluzione ottima per l'istanza originale una volta che il nodo h viene nuovamente "espanso" nella coppia di nodi i e j collegati dal lato $\{i, j\}$. In entrambi i casi abbiamo ricondotto il problema ad uno "più piccolo", con un lato oppure un nodo in meno: iterando il procedimento è possibile costruire una soluzione per l'istanza originale I con al più $O(n^2)$ chiamate all'algoritmo $\mathcal{A}$ (si noti che il problema in cui esistono solo lati uscenti dalla radice è banale). Di conseguenza, se $\mathcal{A}$ fosse polinomiale sarebbe possibile costruire una soluzione ottima di (CMST) in tempo polinomiale.

Questa osservazione giustifica, anche dal punto di vista teorico, l'interesse per tecniche in grado di determinare valutazioni superiori (inferiori nel caso di un problema di minimo) del valore ottimo della funzione obiettivo di un dato problema di *OC*. Usando la terminologia propria della teoria della complessità computazionale, possiamo considerare il valore ottimo della funzione obiettivo di un problema di *OC* come un *certificato* di ottimalità. I problemi di *OC* "facili" sono quelli per i quali sono disponibili tecniche efficienti per costruire un tale certificato, mentre quelli "difficili" sono quelli per cui non sono note tecniche in grado di svolgere efficientemente questo compito.

Le tecniche per determinare valutazioni superiori del valore ottimo della funzione obiettivo di un dato problema di *OC* sono comunque molto importanti anche nella pratica. In effetti, per molte classi di problemi di *OC* esiste *in pratica* una consistente differenza tra *produrre* una soluzione ε-ottima e *certificare* la ε-ottimalità di una soluzione data. Ad esempio, in molti algoritmi enumerativi per problemi "difficili" (si veda il Capitolo 7) capita sovente che l'algoritmo determini la soluzione ottima in tempo relativamente breve, ma sia poi ancora necessario un grandissimo sforzo computazionale per *dimostrare* che tale soluzione è effettivamente ottima. In altri termini, la difficoltà del problema risiede non tanto nel costruire una soluzione ottima, quando nel verificarne l'ottimalità, ossia nel determinare il valore ottimo della funzione obiettivo. Alle tecniche utili a determinare questo valore, o una sua approssimazione accurata, è dedicata una parte rilevante della ricerca attuale volta a sviluppare algoritmi "efficienti" per problemi di *OC*.

Riferimenti Bibliografici

F. Maffioli "**Elementi di Programmazione Matematica**", *Casa Editrice Ambrosiana*, 2000.

A. Sassano "**Modelli e Algoritmi della Ricerca Operativa**", *FrancoAngeli*, 1999.

L. Wolsey "**Integer Programming**", *Wiley-Interscience*, 1998.

Capitolo 5

Algoritmi euristici

Dato che molti problemi di *OC* sono “difficili”, è spesso necessario sviluppare algoritmi *euristici*, ossia algoritmi che non garantiscono di ottenere una soluzione ottima, ma in generale sono in grado di fornire una “buona” soluzione ammissibile per il problema. Normalmente gli algoritmi euristici hanno una ridotta complessità in tempo, ma in alcuni casi, per problemi di grandi dimensioni e struttura complessa, può essere necessario sviluppare algoritmi euristici sofisticati e di elevata complessità. Inoltre, è possibile, in generale, che un algoritmo euristico “fallisca” e non sia in grado di determinare nessuna soluzione ammissibile del problema, pur senza essere in grado di dimostrare che non ne esistano.

Il progetto di algoritmi euristici efficaci richiede un’attenta analisi del problema da risolvere volta ad individuarne la “struttura”, ossia le caratteristiche specifiche utili, ed una buona conoscenza delle principali tecniche algoritmiche disponibili. Infatti, anche se ogni problema ha le sue caratteristiche specifiche, esistono un certo numero di tecniche generali che possono essere applicate, in modi diversi, a moltissimi problemi, producendo *classi* di algoritmi di ottimizzazione ben definite. In questo capitolo ci soffermeremo su due tra le principali tecniche algoritmiche utili per la realizzazione di algoritmi euristici per problemi di *OC*: gli algoritmi *greedy* e quelli di *ricerca locale*.

Queste tecniche algoritmiche non esauriscono certamente lo spettro delle euristiche possibili, per quanto forniscano una buona base di partenza per l’analisi e la caratterizzazione di moltissimi approcci. In particolare, vale la pena sottolineare che l’enfasi sulla “struttura” del problema di ottimizzazione è comune anche alle tecniche utilizzate per la costruzione di valutazioni superiori del valore ottimo della funzione obiettivo, che saranno esaminate nel Capitolo 6. Questo fa sì che spesso una stessa struttura del problema venga utilizzata sia per realizzare euristiche che per determinare valutazioni superiori; si può così avere una “collaborazione” tra euristiche e rilassamenti, come nei casi delle *tecniche di arrotondamento* e delle *euristiche Lagrangiane*, che saranno discusse nel Capitolo 6.

Comunque, esempi di situazioni in cui la computazione di una valutazione superiore è parte integrante di—o comunque guida—un approccio euristico saranno presentate già in questo capitolo. Per contro, le sopracitate tecniche di arrotondamento ed euristiche Lagrangiane sono spesso classificabili come euristiche greedy o di ricerca locale che sfruttano informazione generata dalla computazione di una valutazione superiore. Pertanto risulta ragionevole concentrarsi inizialmente su queste due grandi classi di approcci.

5.1 Algoritmi greedy

Gli algoritmi *greedy* (voraci) determinano una soluzione attraverso una sequenza di decisioni “localmente ottime”, senza mai tornare, modificandole, sulle decisioni prese. Questi algoritmi sono di facile implementazione e notevole efficienza computazionale ma, sia pure con alcune eccezioni di rilievo, in generale non garantiscono l’ottimalità, e a volte neppure l’ammissibilità, della soluzione trovata.

La definizione che abbiamo dato di algoritmo greedy è molto generale, e quindi possono essere ricondotti a questa categoria algoritmi anche all’apparenza molto diversi tra loro. È comunque possibile, a titolo di esemplificazione, definire uno schema generale di algoritmo greedy, adatto a tutti quei casi in cui l'insieme ammissibile può essere rappresentato come una famiglia $F \subset 2^E$ di sottoinsiemi di un dato insieme "base" E.

```
procedure Greedy( E , F , S )  {
  S := ∅; Q = E;
  do { e = Best( Q ); Q = Q \ { e };
       if( S ∪ { e } ∈ F ) then S = S ∪ { e }
       } while( Q ≠ ∅ and not Maximal( S ) )
  }
```

Procedura 5.1: Algoritmo *Greedy*

Nella procedura, S è l'insieme degli elementi di E che sono stati inseriti nella soluzione (parziale) corrente, e Q è l'insieme degli elementi di E ancora da esaminare: gli elementi in $E \setminus (S \cup Q)$ sono quelli per cui si è deciso che *non* faranno parte della soluzione finale. La sottoprocedura $Maximal$ restituisce *vero* se non è più possibile aggiungere elementi alla soluzione S, per cui l'algoritmo può terminare senza esaminare gli elementi eventualmente ancora presenti in Q. In molti casi, le soluzioni ammissibili del problema sono solamente gli elementi "massimali" di F: quindi, se l'algoritmo greedy termina avendo esaurito gli elementi di Q senza che $Maximal$ restituisca *vero*, allora l'algoritmo "fallisce", ossia non è in grado di determinare una soluzione ammissibile del problema. La sottoprocedura $Best$ fornisce il miglior elemento di E tra quelli ancora in Q sulla base di un prefissato criterio, ad esempio l'elemento di costo minimo nel caso di problemi di minimo.

5.1.1 Esempi di algoritmi greedy

È immediato verificare che l'algoritmo di Kruskal (si veda il pragrafo B.4.1) per (MST) ricade nello schema generale appena proposto: E è l'insieme degli archi del grafo, e S è la soluzione parziale corrente, ossia un sottografo privo di cicli. La procedura $Best$ determina un arco di costo minimo tra quelli non ancora considerati: ciò viene eseguito scorrendo la lista degli archi in ordine di costo non decrescente. Il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che l'arco $e = (i, j)$ selezionato non formi un ciclo nel sottografo individuato da S. La procedura $Maximal$ restituisce *vero* quando S contiene esattamente $n - 1$ archi, ossia è un albero di copertura; se il grafo non è connesso, $Maximal$ non restituisce mai *vero* e l'algoritmo "fallisce", ossia non è in grado di determinare un albero di copertura (semplicemente perchè non ne esistono). Ricordiamo che l'algoritmo di Kruskal ha complessità $O(m \log n)$, essenzialmente dovuta all'ordinamento degli archi per costo non decrescente. Come abbiamo visto (e rivedremo, cf. §5.1.3), per il caso di (MST) si può dimostrare che la soluzione generata dall'algoritmo greedy è ottima (l'algoritmo è *esatto*). Vediamo adesso altri esempi di algoritmi greedy, per diversi problemi di *OC*, che risultano invece avere un comportamento euristico.

5.1.1.1 Il problema dello zaino

Si consideri il problema dello zaino (KP) definito al paragrafo 1.2.2.1. Un semplice algoritmo greedy per questo problema consiste nel costruire una soluzione inserendo per primi nello zaino gli oggetti "più promettenti", ossia quelli che con maggiore probabilità appartengono ad una soluzione ottima, secondo un qualche criterio euristico. L'algoritmo inizializza l'insieme S degli oggetti selezionati come l'insieme vuoto, e poi scorre la lista degli oggetti ordinati secondo il criterio euristico: l'oggetto a_h di volta in volta selezionato viene accettato se la capacità residua dello zaino è sufficiente, cioè se $b - \sum_{i \in S} a_i \geq a_h$; in questo caso l'oggetto a_h viene aggiunto ad S, altrimenti viene scartato e si passa al successivo nell'ordinamento. L'algoritmo termina quando tutti gli oggetti sono stati esaminati oppure la capacità residua dello zaino diviene 0. È immediato verificare che anche questo algoritmo ricade nello schema generale. E è l'insieme degli oggetti tra i quali scegliere, la procedura $Best$ determina l'oggetto migliore secondo il criterio euristico, il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che la capacità residua dello zaino sia sufficiente ad accogliere il nuovo oggetto, e la procedura $Maximal$ restituisce *vero* quando la capacità residua dello zaino è zero. In questo caso le soluzioni sono ammissibili anche se l'algoritmo termina senza che $Maximal$ restituisca *vero* (ciò non sarebbe più vero qualora il problema richiedesse di determinare un insieme di elementi di peso *esattamente uguale* a b).

Si noti che quello appena proposto non è *un* algoritmo, ma piuttosto una *famiglia* di algoritmi greedy per (KP) che si differenziano per il criterio con cui vengono selezionati gli oggetti. Consideriamo ad esempio i seguenti tre criteri:

- *pesi non decrescenti*: $a_1 \leq a_2 \leq \ldots \leq a_n$;
- *costi non crescenti*: $c_1 \geq c_2 \geq \ldots \geq c_n$;
- *costi unitari non crescenti*: $c_1/a_1 \geq c_2/a_2 \geq \ldots \geq c_n/a_n$.

Ciascuno dei tre criteri è "ragionevole": con il primo si cercano di inserire nello zaino "molti" oggetti, col secondo quelli di costo maggiore, con il terzo quelli che hanno il maggiore costo per unità di spazio occupato. Nessuno dei tre criteri di ordinamento degli elementi domina gli altri; tuttavia, è facile rendersi conto del fatto che l'ultimo (costi unitari non crescenti) è il più ragionevole, ed in generale quello che fornisce risultati migliori. Chiamiamo CUD l'algoritmo greedy per (KP) che utilizzi il criterio dei costi unitari non crescenti.

Esercizio 5.1 *Per ciascuno dei tre criteri, costruire un esempio in cui la soluzione fornita secondo quel criterio domini le soluzioni fornite in accordo agli altri.*

Esempio 5.1: Esecuzione di Greedy-CUD per (KP)

Consideriamo la seguente istanza del problema dello zaino:

$$\begin{array}{lccccccccccccc}
\max & 7x_1 & + & 2x_2 & + & 4x_3 & + & 5x_4 & + & 4x_5 & + & x_6 & & \\
 & 5x_1 & + & 3x_2 & + & 2x_3 & + & 3x_4 & + & x_5 & + & x_6 & \leq & 8 \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & , & x_6 & \in & \{0,1\}
\end{array}$$

In questo caso, l'algoritmo CUD esegue i seguenti passi:

1. la variabile con costo unitario maggiore è x_5, per cui risulta $c_5/a_5 = 4$: si pone allora $x_5 = 1$, e lo zaino rimane con una capacità residua di 7 unità;
2. la seconda variabile, nell'ordine scelto, è x_3, il cui costo unitario è 2: essa ha un peso minore della capacità residua, e si pone quindi $x_3 = 1$; la capacità residua dello zaino scende di conseguenza a 5;
3. la terza variabile esaminata è x_4, il cui costo unitario è $5/3$: anche essa ha un peso minore della capacità residua, e si pone quindi $x_4 = 1$ cosicché lo zaino rimane con una capacità residua di 2 unità;
4. la quarta variabile considerata è x_1, il cui costo unitario è $7/5$: essa ha però peso 5, superiore alla capacità residua 2 dello zaino, e pertanto si pone $x_1 = 0$.
5. la quinta variabile, nell'ordine, è x_6, che ha costo unitario 1: la variabile ha peso 1, inferiore alla capacità residua, pertanto si pone $x_6 = 1$ e lo zaino rimane con una capacità residua di 1 unità;
6. l'ultima variabile considerata è x_2: tale variabile ha un peso (5) superiore alla capacità residua (1), e quindi si pone $x_2 = 0$.

La soluzione ottenuta è allora $[\,0\,,\,0\,,\,1\,,\,1\,,\,1\,,\,1\,]$, con costo 14 e peso totale 7: è facile vedere che questa soluzione non è ottima, dato che la soluzione $[\,1\,,\,0\,,\,1\,,\,0\,,\,1\,,\,0\,]$, con peso totale 8, ha un costo di 15.

Esercizio 5.2 *Si esegua l'algoritmo greedy per (KP) sull'istanza dell'esempio precedente utilizzando gli altri due criteri euristici per la selezione dell'elemento.*

L'algoritmo greedy ha complessità $O(n \log n)$, con ciascuno dei tre criteri sopra descritti, essenzialmente dovuta all'ordinamento degli oggetti; se gli oggetti sono forniti in input già ordinati secondo il criterio selezionato, la complessità dell'algoritmo è lineare.

5.1.1.2 Il problema dell'assegnamento di costo minimo

Si consideri il problema accoppiamento di massima cardinalità e costo minimo discusso al paragrafo 3.5.2. Un algoritmo greedy per questo problema può essere facilmente costruito nel modo seguente. Si ordinano gli archi per costo non crescente, e si inizializza un vettore di booleani, con una posizione per ogni nodo, a $false$, per indicare che nessun nodo è assegnato. Si esaminano poi gli archi nell'ordine dato; se entrambe gli estremi dell'arco non sono assegnati l'arco viene aggiunto all'accoppiamento e si pongono a *vero* le posizioni corrispondenti nell'array, per segnalare che i nodi sono adesso accoppiati. L'algoritmo termina quando non ci sono più nodi da accoppiare, oppure quando sono stati esaminati tutti gli archi. Questo algoritmo ricade ovviamente nello schema generale. E è l'insieme A degli archi del grafo, la procedura *Best* determina l'arco di costo minimo tra quelli non ancora esaminati, il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che entrambe gli estremi dell'arco non siano già assegnati, la procedura $Maximal$ ritorna *vero* se tutti i nodi sono accoppiati. In particolare, se si desidera un assegnamento (accoppiamento perfetto) l'algoritmo "fallisce" se termina avendo esaminato tutti gli archi senza che $Maximal$ abbia ritornato *vero*. È facile dimostrare che questo algoritmo non costruisce necessariamente una soluzione ottima del problema; in particolare può non essere neanche in grado di determinare un assegnamento nel grafo anche se ne esiste uno.

Esercizio 5.3 *Si fornisca un esempio che dimostri l'affermazione precedente (suggerimento: si veda il paragrafo 5.1.3).*

In compenso l'algoritmo ha complessità $O(m \log n)$, essenzialmente dovuta all'ordinamento degli archi, sostanzialmente inferiore a quella $O(mn^2)$ degli algoritmi esatti discussi nel paragrafo 3.5.2; quindi questo algoritmo potrebbe ad esempio risultare utile per ottenere rapidamente "buoni" assegnamenti per problemi di grandissima dimensione. Si noti inoltre che questo algoritmo, a differenza di quelli del paragrafo 3.5.2, è adatto anche al caso in cui il grafo G non sia bipartito. Sono comunque stati proposti algoritmi esatti per il caso bipartito con complessità inferiore a $O(mn^2)$ (uno dei quali basato sull'idea di "trasformare" la soluzione ottenuta dall'algoritmo greedy), ed anche algoritmi esatti per il caso non bipartito; per ulteriori dettagli si rimanda alla letteratura citata.

5.1.1.3 Il problema del commesso viaggiatore

Si consideri il problema del commesso viaggiatore (TSP) definito al paragrafo 1.2.2.3. Una famiglia di algoritmi greedy per questo problema può essere costruita come segue. L'algoritmo inizializza l'insieme S degli archi appartenenti al ciclo come l'insieme vuoto, e definisce come nodo "corrente" il nodo iniziale (1). Ad ogni iterazione, poi, esamina il nodo corrente i e tutti gli archi che lo uniscono a nodi che non sono ancora toccati dal ciclo parziale S: tra di essi seleziona l'arco (i, j) "più promettente" secondo un certo criterio euristico, lo aggiunge a S e definisce j come nuovo nodo corrente. L'algoritmo termina quando tutti i nodi sono toccati da S, inserendo l'arco di ritorno dall'ultimo nodo al nodo 1. Anche questo algoritmo ricade nello schema generale: E è l'insieme degli archi del grafo, *Best* determina l'arco più promettente, tra tutti quelli che escono dal nodo corrente, secondo il criterio euristico, il controllo "$S \cup \{ e \} \in F$" corrisponde a verificare che il nodo terminale j dell'arco (i, j) non sia già stato visitato, e $Maximal$ restituisce *vero* quando tutti i nodi sono stati visitati.

Anche in questo caso abbiamo una famiglia di algoritmi che si differenziano per il criterio utilizzato per determinare l'arco "più promettente". Un criterio molto intuitivo è semplicemente quello di selezionare un arco di lunghezza minima: ciò corrisponde a scegliere ad ogni passo, come prossima tappa, la località più vicina a quella in cui il commesso si trova attualmente, anche noto come algoritmo "Nearest Neighbour".

Esempio 5.2: Esempio di esecuzione di "Nearest Neighbour"

Come nel caso del problema dello zaino, l'algoritmo greedy non è esatto (del resto, entrambi i problemi sono NP-ardui): questo può essere facilmente verificato mediante l'istanza rappresentata in figura 5.1(a). L'algoritmo "Nearest Neighbour", partendo dal nodo 1, produce il ciclo rappresentato in figura 5.1(b), con lunghezza 12, che è peggiore del ciclo ottimo rappresentato in figura 5.1(c), che ha costo 11.

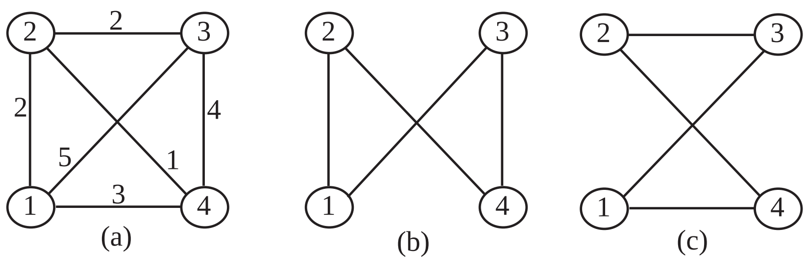


Figura 5.1: Un'istanza del problema del commesso viaggiatore

Si noti che l'algoritmo costruisce sicuramente un ciclo hamiltoniano se il grafo G è completo, mentre può "fallire" nel caso in cui G non sia completo: ciò accade se tutti gli archi uscenti dal nodo corrente i portano a nodi già visitati dal ciclo parziale S, oppure se i è l'ultimo dei nodi da visitare ma non esiste l'arco fino al nodo iniziale. Quindi, a differenza del problema dello zaino, l'algoritmo greedy non solo non garantisce di determinare una soluzione ottima, ma può non essere in grado di produrre neanche una qualsiasi soluzione ammissibile. Ciò non deve stupire: mentre per il problema dello zaino è immediato costruire una soluzione ammissibile (lo zaino vuoto), il problema di decidere se esiste un ciclo Hamiltoniano in un grafo non completo è $\mathcal{NP}$-arduo, e quindi non è pensabile che un algoritmo greedy sia in grado di risolverlo.

Sono comunque stati proposti molti altri criteri di selezione del nodo successivo che possono rivelarsi più efficienti in pratica. Ad esempio, quando il grafo G può essere rappresentato su un piano (ad esempio, quando i nodi corrispondono effettivamente a località geografiche) un criterio interessante è quello che seleziona j in modo tale che il segmento (arco) (i, j) formi il più piccolo angolo possibile con il segmento (arco) (h, i), dove h è il nodo visitato immediatamente prima di i nel ciclo parziale (ossia $(h, i) \in S$). Pur senza entrare nei dettagli, segnaliamo il fatto che questo criterio è motivato da alcune proprietà della frontiera dell'inviluppo convesso di un insieme di punti del piano e dalle relazioni che esistono tra l'inviluppo convesso dei punti che rappresentano i nodi ed il ciclo Hamiltoniano di costo minimo; quindi, per quanto il criterio sia semplice da capire e da implementare, la sua ideazione è stata resa possibile solamente da uno studio accurato delle proprietà (di alcuni casi rilevanti) del problema in oggetto. Si noti come, comunque, ancora una volta, il costo computazionale della procedura è molto basso, essendo lineare nel numero degli archi del grafo ($O(n^2)$).

5.1.1.4 Ordinamento di lavori su macchine con minimizzazione del tempo di completamento

Si consideri il problema di ordinamento di lavori su macchine con minimizzazione del tempo di completamento (MMMS) definito al paragrafo 1.2.9.1. Una famiglia di algoritmi greedy per questo problema può essere costruita come segue. All'inizio, nessun lavoro è assegnato e tutte le macchine sono scariche, ossia $N(j) = \emptyset$ per $j = 1, \ldots, m$. Ad ogni iterazione si seleziona uno dei lavori i ancora da assegnare, secondo un certo criterio euristico, e lo si assegna alla macchina "più scarica", ossia a quella con tempo di completamento $D(j) = \sum_{i \in N(j)} d_i$ (relativo alla soluzione parziale corrente) più basso; in caso di parità, si sceglie una qualunque delle macchine col tempo di completamento corrente minimo. L'algoritmo termina quando tutti i lavori sono stati assegnati. Anche questo algoritmo può essere fatto ricadere nello schema generale: E è l'insieme delle coppie (i, j) con $i = 1, \ldots, n$ e $j = 1, \ldots, m$, ossia dei possibili assegnamenti di lavori a macchine. *Best* seleziona prima un lavoro i non ancora assegnato, secondo il criterio euristico, e poi la macchina (più scarica) a cui assegnarlo. Il controllo "$S \cup \{ e \} \in F$" non esiste, in quanto dato un lavoro non assegnato è sempre possibile assegnarlo a qualsiasi macchina. Anche la procedura $Maximal$ non fa nulla, in quanto le soluzioni sono ammissibili se e solo se tutti i lavori sono stati assegnati, ossia Q è vuoto.

Anche in questo caso quella appena proposta è una famiglia di algoritmi greedy, detti *list scheduling*, che si differenziano solamente per il criterio utilizzato per determinare il prossimo lavoro i da assegnare. Tra tutti i possibili criteri, due sono quelli più significativi:

- *SPT (Shortest Processing Time)*: i lavori vengono assegnati in ordine non decrescente dei tempi di esecuzione (quelli più "corti" per primi);
- *LPT (Longest Processing Time)*: i lavori vengono assegnati in ordine non crescente dei tempi di esecuzione (quelli più "lunghi" per primi).

Esempio 5.3: Esecuzione di algoritmi list scheduling
Nelle figure 5.2 e 5.3 sono riportati i risultati ottenuti con i due criteri su due esempi; osserviamo che nel primo caso LPT fornisce una soluzione ottima, mentre nel secondo caso nessuno dei due algoritmi riesce a raggiungere l'ottimo.

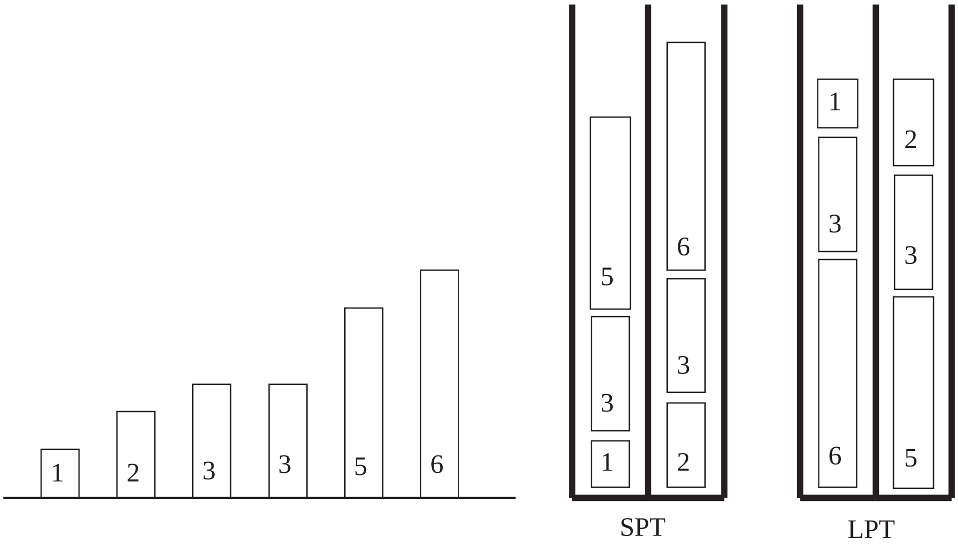


Figura 5.2: Un'istanza del problema (MMMS)

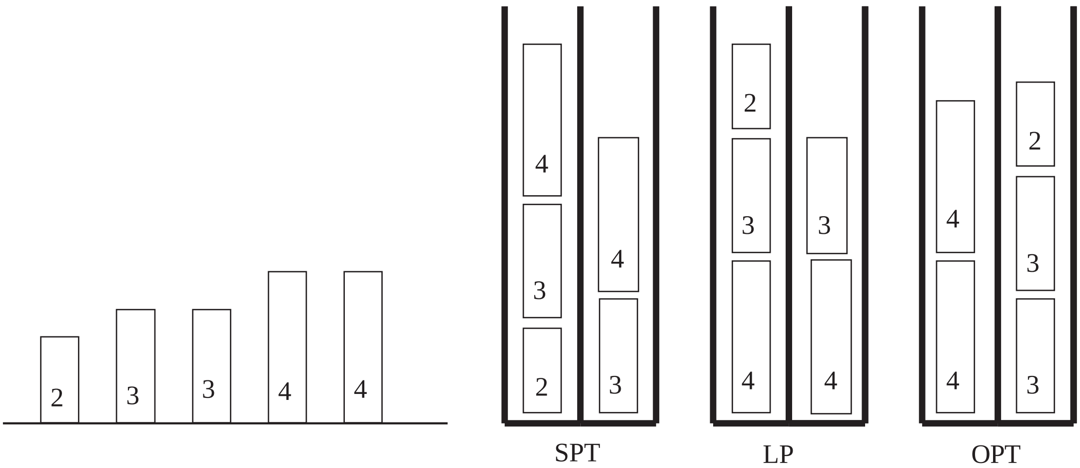


Figura 5.3: Un'istanza del problema (MMMS)

Come vedremo in seguito, LPT ha "migliori proprietà" di SPT, e di fatto in generale risulta più efficace, ossia produce soluzioni di migliore qualità. Per entrambe i criteri di ordinamento, comunque, l'algoritmo greedy è facile da implementare e risulta molto efficiente in pratica.

Esercizio 5.4 *Si discuta la complessità dell'algoritmo greedy per (MMMS); come cambia il risultato se i lavori sono forniti in input già ordinati secondo il criterio selezionato?*

5.1.1.5 Ordinamento di lavori su macchine con minimizzazione del numero delle macchine

Si consideri la variante di (MMMS) in cui sono dati i tempi di inizio e di fine di ciascun lavoro e si vuole minimizzare il numero di macchine utilizzate, ossia il problema (MCMS) definito al paragrafo 1.2.4.2. Una famiglia di algoritmi greedy per questo problema può essere costruita in modo analogo a quello visto nel paragrafo precedente. All'inizio, nessun lavoro è assegnato e nessuna macchina è utilizzata. Ad ogni iterazione si seleziona uno dei lavori i ancora da assegnare, secondo un certo criterio euristico, e si scorre la lista delle macchine già utilizzate, secondo un altro opportuno criterio euristico, assegnando il lavoro alla prima macchina sulla quale è possibile eseguirlo: se non è possibile eseguire i su nessuna delle macchine già utilizzate, lo si assegna ad una nuova macchina fino a quel momento scarica, che viene quindi aggiunta all'insieme di macchine utilizzate. Questo algoritmo può essere fatto ricadere nello schema generale analogamente a quanto visto nel paragrafo precedente; si noti che, ancora una volta, quella appena proposta è una famiglia di algoritmi greedy che si differenziano per i criteri utilizzati per determinare il prossimo lavoro i da assegnare e la macchina j già utilizzata a cui assegnarlo (se possibile).

Esercizio 5.5 *Si propongano almeno due criteri diversi per la selezione del prossimo lavoro i da assegnare e almeno due criteri diversi per la selezione della macchina j già utilizzata a cui assegnarlo, discutendo i possibili vantaggi e svantaggi di ciascuna combinazione.*

Tra tutti gli algoritmi greedy appartenenti allo schema appena introdotto ne esiste uno che costruisce certamente una soluzione ottima per il problema. Si consideri infatti l'algoritmo in cui il lavoro da assegnare viene selezionato in ordine di tempo di inizio t_i non crescente; in altre parole vengono assegnati per primi i lavori che iniziano prima. Supponiamo di ordinare le macchine in un qualsiasi ordine, e di esaminarle per l'inserzione del lavoro corrente sempre nell'ordinamento dato; se il lavoro non può essere inserito nelle macchine attualmente utilizzate, sarà attivata la successiva macchina nell'ordinamento. Questo algoritmo costruisce un assegnamento che utilizza sempre il minor numero possibile di macchine. Sia k l'indice dell'ultima macchina attivata nel corso dell'algoritmo, ossia il numero di macchine utilizzate nella soluzione costruita, e sia i il primo lavoro assegnato a quella macchina. Infatti, si consideri lo stato delle altre macchine "attive" al momento in cui viene esaminato i: a ciascuna di esse è stato assegnato un lavoro h incompatibile con i, ossia tale che $[t_i, t_i + d_i] \cap [t_h, t_h + d_h] \neq \emptyset$. Ma, per via della strategia di selezione dei lavori, ciascun lavoro h assegnato ad una macchina in quel momento ha $t_h \leq t_i$: non è quindi possibile che il lavoro i sia incompatibile con il lavoro h perchè $t_i < t_h \leq t_i + d_i \leq t_h + d_h$, ne consegue che deve risultare $t_i \in [t_h, t_h + d_h]$. Di consegeunza, nell'istante t_i devono necessariamente essere in esecuzione k lavori: i, che inizia in quel momento, e gli altri $k-1$ che occupano le altre macchine nell'assegnamento (parziale) costruito fino a quel momento. Di conseguenza sono necessarie almeno k macchine per eseguire tutti i lavori: poichè la soluzione costruita ne usa esattamente k, essa è ottima.

Esercizio 5.6 *Si discuta come implementare l'algoritmo greedy "ottimo" per (MCMS) in modo da ottenere una bassa complessità computazionale.*

Questo esempio mostra come la conoscenza di un algoritmo greedy per un certo problema di *OC* possa suggerire algoritmi greedy analoghi per problemi di *OC* "simili", ma anche come problemi apparentemente "simili" possano in effetti risultare molto diversi in termini di facilità di soluzione.

Esercizio 5.7 *Si proponga, fornendone una descrizione formale, un algoritmo greedy per il problema (GC) di colorare i nodi di un grafo $G = (N, A)$ con il minimo numero di colori con il vincolo che due nodi adiacenti non abbiano mai lo stesso colore (si veda il paragrafo 1.2.4.3). Si discuta sotto quali ipotesi sul grafo si può costruire un algoritmo greedy equivalente a quello "ottimo" per (MCMS) che riporti sicuramente una soluzione ottima per (GC). Si discuta poi come modificare l'algoritmo per il caso più generale in cui ad ogni nodo i devono essere assegnati esattamente n_i colori diversi e/o i colori assegnati a nodi adiacenti devono essere "distanti" di almeno una soglia δ fissata.*

5.1.1.6 Il problema di copertura

Si consideri il problema di copertura (PC) definito al paragrafo 1.2.5. Una famiglia di algoritmi greedy per questo problema può essere costruita come segue. L'algoritmo inizializza l'insieme S dei sottoinsiemi selezionati come l'insieme vuoto. Ad ogni iterazione, seleziona uno dei sottoinsiemi F_j ancora da esaminare, secondo un certo criterio euristico e lo aggiunge a S se il nuovo sottoinsieme "copre" almeno un elemento di N che non era "coperto" dalla soluzione parziale precedente, ossia se $F_j \not\subset F_S = \cup_{F_i \in S} F_i$. L'algoritmo termina quando Q è vuoto oppure quando $F_S = N$, ossia tutti gli elementi di N sono "coperti" da S.

Esercizio 5.8 *Si mostri che l'algoritmo appena descritto ricade nello schema generale di algoritmo greedy.*

Anche in questo caso, quella appena proposta è una famiglia di algoritmi greedy che si differenziano per il criterio utilizzato per determinare il sottoinsieme "più promettente". Consideriamo ad esempio i seguenti tre criteri:

- *costi non decrescenti*: vengono esaminati prima i sottoinsiemi F_j con costo c_j più basso;
- *costi unitari non decrescenti*: vengono esaminati prima i sottoinsiemi F_j con "costo unitatio" $c_j/|F_j|$ più basso, ossia si tiene conto del numero di oggetti che un dato sottoinsieme può coprire;
- *costi unitari attualizzati non decrescenti*: vengono esaminati prima i sottoinsiemi F_j con "costo unitatio attualizzato" $c_j/|F_j \setminus F_S|$ più basso, ossia si tiene conto del numero di oggetti che un dato sottoinsieme copre e che non sono già coperti dalla soluzione parziale S.

Non è sorprendente che il terzo criterio, quello dei costi unitari attualizzati, sia in pratica spesso migliore degli altri due, in quanto è l'unico dei tre che utilizza informazione sulla soluzione corrente S per decidere il prossimo sottoinsieme da esaminare. Si noti anche, però, che tale criterio è potenzialmente più costoso da implementare: infatti per i primi due criteri è possibile ordinare gli oggetti all'inizio e poi semplicemente scorrere la lista ordinata, il che ha complessità $(m \log m)$, mentre nel terzo l'ordinamento deve essere ricalcolato ogniqualvolta un oggetto viene inserito in S, e quindi F_S aumenta.

5.1.1.7 Il problema (CMST)

Si consideri il problema dell'albero di copertura di costo minimo capacitato (CMST) definito nell'esempio 4.1. Per il problema "più semplice" dell'albero di copertura di costo minimo (MST) conosciamo algoritmi greedy esatti, ossia in grado determinare una soluzione ottima. Chiaramente, tali algoritmi ottengono la soluzione ottima per (CMST) se la capacità Q degli archi uscenti dalla radice, ossia il massimo peso dei sottoalberi, è "grande"; per questo, è ragionevole cercare di costruire algoritmi greedy per (CMST) che si ispirino agli algoritmi per (MST). Nell'algoritmo di Kruskal, ad esempio, si pone $S = \emptyset$ e si esaminano gli archi in ordine di costo non decrescente: l'arco (i, j) esaminato viene aggiunto ad S se non crea cicli con gli archi già presenti in S, ossia se collega due diverse componenti connesse del grafo (N, S). Non è difficile modificare l'algoritmo in modo tale che tenga conto delle capacità: basta mantenere il peso (somma del peso dei nodi) di ogni componente connessa, e non accettare l'arco (i, j) se la sua inserzione in S causa l'unione di due componenti connesse la cui somma dei pesi è maggiore di Q. Questo ovviamente non si applica ad archi di tipo (r, i), ossia che collegano una componente connessa alla radice. Se esistono archi da r a tutti gli altri nodi allora l'algoritmo così modificato costruisce sicuramente una soluzione ammissibile per il problema, altrimenti può "fallire". È possibile implementare il controllo sul peso delle componenti connesse, mediante opportune strutture dati, in modo da non aumentare la complessità dell'algoritmo di Kruskal.

Questo esempio mostra come la conoscenza di algoritmi per un dato problema possa essere utilizzata per guidare la realizzazione di approcci per problemi simili ma più complessi. Naturalmente, non sempre è facile adattare gli algoritmi noti per risolvere problemi più complessi in modo naturale: ad esempio, adattare l'algoritmo di Prim al (CMST) è molto meno immediato.

5.1.2 Algoritmi greedy con garanzia sulle prestazioni

Nel paragrafo precedente abbiamo presentato un certo numero di algoritmi greedy per alcuni problemi di ottimizzazione rilevanti. Una volta che un algoritmo sia stato ideato ed implementato, si pone il problema di valutarne in qualche modo l'efficacia, ossia la capacità di fornire effettivamente "buone" soluzioni con errore relativo basso. Ci sono essenzialmente due modi per studiare l'efficacia di un algoritmo euristico:

- *sperimentale*: si seleziona un sottoinsieme "rilevante" di istanze del problema, ad esempio tali che siano rappresentative delle istanze da risolvere in una o più applicazioni pratiche, si esegue l'algoritmo su quelle istanze misurando l'errore relativo ottenuto e poi se ne esaminano le caratteristiche statistiche (media, massimo, minimo, varianza, ecc.); si noti che per fare questo è necessario essere in grado di calcolare il valore ottimo della funzione obiettivo, o una sua buona approssimazione, per le istanze test;

- *teorico*: si dimostrano matematicamente relazioni che forniscono valutazioni relative al massimo errore compiuto dall'algoritmo quando applicato ad istanze con caratteristiche date; più raramente è possibile valutare anche altre caratteristiche statistiche dell'errore compiuto (media ecc.)

Le due metodologie di valutazione non sono alternative ma complementari: lo studio teorico risulta in genere meno accurato nel valutare l'errore compiuto per le istanze reali, ma fornisce valutazioni generali valide per grandi classi di istanze e aiuta a comprendere meglio il comportamento dell'algoritmo, eventualmente suggerendo come modificarlo per renderlo più efficace; d'altro canto, lo studio sperimentale permette di valutare con maggiore accuratezza l'errore compiuto sulle istanze utilizzate e di estrapolare con ragionevole accuratezza l'errore che ci si può attendere su istanze simili, ma non fornisce alcuna garanzia, specialmente per istanze con caratteristiche diverse da quelle effettivamente testate. Ciò è del tutto analogo alla differenza tra lo studio della complessità nel caso pessimo di un algoritmo e la verifica sperimentale dell'efficienza pratica della sua implementazione.

In questo paragrafo mostreremo alcuni semplici risultati relativi alla valutazione dell'errore di alcuni algoritmi greedy, e illustreremo come sia possibile progettare algoritmi in modo da garantire che abbiano determinate prestazioni. Si noti che lo studio teorico delle prestazioni degli algoritmi è in principio possibile anche per altre classi di algoritmi euristici, come quelli di ricerca locale che presenteremo nel paragrafo 5.2; essendo però in generale piuttosto complesso, risulta spesso troppo difficile per algoritmi che non abbiano una struttura semplice, ed è per questo che è principalmente effettuato su algoritmi di tipo greedy.

Dato un algoritmo euristico $\mathcal{A}$, si distinguono due diversi modi per valutare l'errore compiuto da $\mathcal{A}$: *a priori* e *a posteriori*. La valutazione a posteriori dell'errore viene effettuata dopo che l'algoritmo ha determinato la soluzione corrispondente ad una singola istanza I, e permette di valutare l'errore compiuto per quella particolare istanza. La valutazione a priori invece fornisce una stima del massimo errore compiuto da $\mathcal{A}$ per *qualsiasi* istanza I, ed è quindi disponibile prima che l'istanza venga risolta. La valutazione a priori è quindi più generale, ma siccome è una valutazione nel caso pessimo è anche usualmente meno precisa di quella a posteriori.

5.1.2.1 L'algoritmo CUD per (KP)

Per valutare l'errore compiuto dall'algoritmo CUD, è necessario per prima cosa ottenere una valutazione superiore del valore ottimo della funzione obiettivo del problema. Per fare ciò consideriamo il rilassamento continuo del problema dello zaino, ossia il problema

$$(\underline{KP}) \qquad \max \left\{ \ \textstyle\sum_{i=1}^n c_i x_i \ : \ \sum_{i=1}^n a_i x_i \leq b \ , \ \ x \in [0,1]^n \ \right\} \ .$$

È possibile verificare che una soluzione x^* ottima per $(\underline{KP})$ può essere costruita nel modo seguente: si ordinano gli oggetti per costo unitario non crescente, si inizializza l'insieme S degli oggetti selezionati (ossia degli indici delle variabili i con $x_i^* = 1$) all'insieme vuoto e si iniziano ad inserire oggetti in S (porre variabili a 1) finchè è possibile, esattamente come nell'algoritmo CUD. Quando però si raggiunge il primo oggetto h (nell'ordine dato) per cui la capacità residua dello zaino non è più sufficiente, cioè $b - \sum_{i \in S} a_i < a_h$, si pone $x_h^* = (\, b - \sum_{i \in S} a_i \,)/a_h$ e $x_i^* = 0$ per $i \notin S \cup \{\, h \,\}$ (si noti che h è ben definito in quanto per ipotesi risulta $b < \sum_i a_i$). Infatti, si consideri il duale di ($\underline{KP}$):

$$(\underline{DKP}) \qquad \min \left\{ \, yb + \textstyle\sum_{i=1}^n w_i \; : \; ya_i + w_i \geq c_i \quad i = 1, \ldots, n \;\; , \;\; y \geq 0 \;\; , \;\; w_i \geq 0 \quad i = 1, \ldots, n \, \right\} \; .$$

Dalla teoria della PL sappiamo che una coppia di soluzioni $\bar{x}$ e $(\bar{y}, \bar{w})$ è ottima per ($\underline{KP}$) e ($\underline{DKP}$) se e solo è ammissibile e rispetta le condizioni degli scarti complementari, che in questo caso sono

$$\bar{y} \big(\, b - \textstyle\sum_{i=1}^n a_i \bar{x}_i \, \big) = 0 \; ,$$

$$\bar{w}_i (\, 1 - \bar{x}_i \,) = 0 \quad , \quad \bar{x}_i (\, \bar{y} a_i + \bar{w}_i - c_i \,) = 0 \qquad i = 1, \ldots, n \; .$$

È immediato verificare che la soluzione duale

$$y^* = c_h / a_h \qquad w_i^* = \left\{ \begin{array}{ll} c_i - y^* a_i & \text{se } i < h, \\ 0 & \text{altrimenti} \end{array} \right. \; :$$

è ammissibile, in quanto, essendo gli oggetti ordinati per costi unitari non crescenti, si ha $w_i^* = c_i/a_i - c_h/a_h \geq 0$. È anche facile controllare che (y^*, w^*) verifica le condizioni degli scarti complementari con x^*: per la prima condizione si ha $\sum_{i=1}^n a_i x_i^* = b$, per la seconda condizione si ha che $w_i^* > 0$ solo per per gli indici $i < h$ per cui $x_i^* = 1$, per la terza condizione si ha che $x_i^* > 0$ al più per gli indici $i \leq h$ (può essere $x_h^* = 0$) per cui $y^* a_i + w_i^* = c_i$.

Possiamo adesso procedere a valutare *a posteriori* l'errore relativo commesso da CUD. Infatti si ha $z(P) \leq \sum_{i=1}^n c_i x_i^* = \sum_{i<h} c_i + c_h (b - \sum_{i<h} a_i)/a_h$ e $\sum_{i<h} c_i \leq z_{CUD} \leq z(P)$, da cui

$$R_{CUD} = \frac{z(P) - z_{CUD}}{z(P)} \leq \frac{c_h \frac{b - \sum_{i<h} a_i}{a_h}}{\sum_{i<h} c_i} \leq \frac{c_h}{\sum_{i<h} c_i} \; .$$

Si noti che se $\sum_{i<h} a_i = b$, ossia l'ultimo oggetto (nell'ordine dei costi unitari non crescenti) che entra interamente nello zaino ne satura la capacità, allora $R_{CUD} = 0$, ossia CUD determina una soluzione ottima del problema. Infatti, in questo caso si ha che la soluzione prodotta da CUD è proprio x^*, in quanto x_h^* (l'unica componente di x^* che può assumere valore frazionario) vale 0. Quindi, la soluzione ottima del rilassamento continuo di (KP) ha valori interi, ossia è una soluzione ammissibile per (KP); il Lemma 4.1 garantisce quindi che x^* sia una soluzione ottima per (KP).

Esempio 5.4: Stime dell'errore per Greedy-CUD
Consideriamo la seguente istanza del problema dello zaino:

$$\begin{array}{rrcrcrcrcc} \max & 11x_1 & + & 8x_2 & + & 7x_3 & + & 6x_4 & & \\ & 5x_1 & + & 4x_2 & + & 4x_3 & + & 4x_4 & \leq & 12 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & \in & \{0,1\} \end{array} \; .$$

Gli oggetti sono già ordinati per costo unitario non crescente. L'algoritmo CUD riporta la soluzione $S = \{1, 2\}$ di costo 19, mentre la soluzione ottima è $\{2, 3, 4\}$ di costo 21: l'errore relativo commesso da CUD in questo caso è $(21-19)/21 \approx 0.095$, ossia del 9.5%. La soluzione ottima di $\underline{KP}$ è $x^* = [\, 1 \, , \, 1 \, , \, 3/4 \, , \, 0 \,]$ di costo $97/4 (= 24 + 1/4)$; ad essa corrisponde infatti la soluzione duale $y^* = 7/4$, $w^* = [\, 9/4 \, , \, 1 \, , \, 0 \, , \, 0 \,]$, anch'essa di costo 97/4. Utilizzando questa valutazione superiore su $z(P)$ per stimare l'errore si ottiene $R_{CUD} \leq (97/4 - 19)/19 \approx 0.276$, ossia il 27.6%. Utilizzando la formula che dipende solamente dai costi e da h si ottiene $R_{CUD} \leq 7/(11 + 8) \approx 0.368$.

Esercizio 5.9 *Per i valori*

$$U' = \textstyle\sum_{i<h} c_i + c_{h+1} \dfrac{b - \sum_{i<h} a_i}{a_{h+1}} \qquad e \qquad U'' = \textstyle\sum_{i \leq h} c_i + c_{h-1} \dfrac{b - \sum_{i \leq h} a_i}{a_{h-1}} \quad ,$$

dimostrare che vale la relazione

$$z(KP) \leq max\{\, U' \, , \, U'' \,\} \leq c_h \frac{b - \sum_{i<h} a_i}{a_h} \; ,$$

ossia che il massimo tra U' ed U'' fornisce una valutazione superiore di $z(P)$ non peggiore di quella utilizzata per valutare R_{CUD} (suggerimento: U' ed U'' sono i valori dele soluzioni ottime di due opportuni problemi di LP *ottenuti supponendo che x_h sia rispettivamente* 0 *e* 1 *in una soluzione ottima di KP).*

5.1.2.2 Gli algoritmi SPT e LPT per (MMMS)

Come per il caso del problema dello zaino, per poter valutare l'errore compiuto dagli algoritmi SPT e LPT per (MMMS) dobbiamo innanzitutto determinare una valutazione—in questo caso inferiore—-sul valore ottimo della funzione obiettivo. Per (MMMS) questo è facile: infatti, si ha

$$z(MMMS) \geq L = \sum_{i=1}^{n} d_i/m$$

(nel caso migliore, i lavori sono perfettamente ripartiti tra le macchine che terminano tutte allo stesso momento, ossia si ha "speedup lineare").

Iniziamo l'analisi dimostrando una valutazione *a priori* che vale per qualsiasi algoritmo di tipo list scheduling, ossia indipendentemente dall'ordine con cui sono assegnati i lavori: in ogni caso, l'errore relativo compiuto non supera $(m-1)/m$. Si noti che l'errore peggiora all'aumetare del numero di macchine, tendendo a 1 (cioè al 100%) quando m cresce. Per dimostrare il risultato occorre definire alcune quantità critiche: in particolare, indichiamo con h il lavoro che termina per ultimo (l'ultimo lavoro eseguito sulla macchina che termina per ultima) e con $H = (\sum_{i=1}^{n} d_i - d_h)/m$: H è il minimo tempo possibile di completamento per le macchine se h fosse rimosso dalla lista dei lavori. Chiamiamo z_{LS} il valore della funzione obiettivo della soluuzione determinata da un generico algoritmo di list scheduling: l'osservazione fondamentale è che risulta $z_{LS} \leq H + d_h = L + d_h(m-1)/m$. Infatti, quando il lavoro h viene assegnato alla macchina "più scarica" rispetto alla soluzione corrente al momento in cui h è esaminato: è quindi ovvio che la situazione peggiore, quella alla quale corrisponde il maggior valore di z_{LS}, è quella in cui h è anche l'ultimo lavoro ad essere assegnato, e le macchine erano "perfettamente bilanciate" al momento in cui h è stato assegnato (si veda la figura 5.4). Da questo e da $z(MMMS) \geq L$ otteniamo

$$R_{LS} = \frac{z_{LS} - z(MMMS)}{z(MMMS)} \leq \frac{L + d_h(m-1)/m - L}{z(MMMS)}$$

Ma siccome chiaramente $z(MMMS) \geq d_h$ (almeno il tempo necessario a completare d_h deve passare) si ha $R_{LS} \leq (m-1)/m$.

Osserviamo che questa valutazione non è "ottimistica", ossia non è possibile migliorarla. Infatti, esiste almeno una classe di istanze ed un ordinamento, in particolare SPT, per cui si ottiene una soluzione che ha esattamente errore relativo $(m-1)/m$. Si consideri infatti l'istanza che contiene $(m-1)m$ lavori di lunghezza unitaria ed un solo lavoro di lunghezza m: l'algoritmo SPT assegna i lavori di lunghezza unitaria per primi, ottenendo $m-1$ macchine con tempo di completamento $m-1$ ed una singola macchina con tempo di completamento $2m-1$. In figura 5.4 è mostrato un esempio per il caso $m = 4$. Ma l'assegnamento ottimo consiste invece nell'assegnare ad una macchina l'unico lavoro di lunghezza m e distribuire poi uniformemente sulle altre i restanti $(m-1)m$: in questo caso tutte le macchine terminano in tempo m.

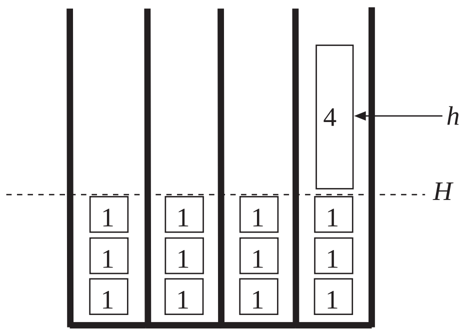


Figura 5.4: Valutazione dell'errore per (MMMS)

Si osservi che l'algoritmo LPT applicato all'istanza "critica" produce la soluzione ottima. In effetti, la valutazione precedente è valida per qualsiasi algoritmo di tipo list scheduling, ma se utilizziamo il particolare ordinamento LPT è possibile dimostrare una valutazione migliore, ossia che $R_{LPT} \leq (m-1)/3m$. Per semplicità di notazione ci limiteremo a considerare il caso $m = 2$ (per cui $R_{LPT} \leq 1/6$) e supporremo che i lavori siano forniti in input già ordinati per durata non crescente, ossia

$d_1 \geq d_2 \geq \ldots \geq d_n$. La dimostrazione è per assurdo: supponiamo che esista un'istanza per cui l'errore relativo sia maggiore di 1/6. Consideriamo in particolare l'istanza I *con* n *minimo* per cui LPT ottiene un errore relativo maggiore di 1/6: in questa istanza si ha $h = n$, ossia l'ultimo lavoro completato è anche quello di durata minore. Infatti, se fosse $h < n$ potremmo eliminare tutti gli oggetti di indice maggiore di h, che chiaramente non contribuiscono alla determinazione del tempo di completamento (essendo h quello che termina per ultimo): otterremmo così un'istanza I' con meno lavori e con errore relativo non minore di quello di I, contraddicendo l'ipotesi che I sia l'istanza con meno lavori che ha errore relativo maggiore di 1/6. Sia quindi z_{LPT} il tempo di completamento della soluzione ottenuta dall'algoritmo LPT sull'istanza I: per ipotesi abbiamo

$$R_{LPT} \geq \frac{z_{LPT} - z(MMMS)}{z(MMMS)} > \frac{1}{6}$$

da cui $z_{LPT} > 7z(MMMS)/6$. Siccome LPT è un particolare algoritmo di list scheduling, vale

$$z_{LPT} \leq H + d_h = L + d_h/2 \leq z(MMMS) + d_n/2$$

si ha quindi $7z(MMMS)/6 < z(MMMS) + d_n/2$, ossia $z(MMMS) < 3d_n$. Siccome d_n è il più corto dei lavori, dalla relazione precedente si ottiene che $n \leq 4$: qualsiasi istanza con almeno 5 lavori di lunghezza almeno d_n ha tempo di completamento non inferiore a $3d_n$. È però facile verificare che per istanze con non più di 4 lavori LPT fornisce la soluzione ottima, da cui l'assurdo.

Esercizio 5.10 *Si dimostri che qualsiasi istanza con al più 4 lavori viene sicuramente risolta all'ottimo da LPT (suggerimento: i casi con 1, 2 e 3 lavori sono banali; chiamando $a \geq b \geq c \geq d$ le lunghezze dei 4 lavori, i primi tre sono sempre assegnati da LPT nello stesso modo – a da solo sulla prima macchina, b e c insieme sulla seconda – mentre d viene assegnato in modo diverso a seconda che risulti $a \geq b + c$ oppure $a < b + c$).*

Esercizio 5.11 *Si estenda la dimostrazione della valutazione superiore dell'errore relativo di LPT al caso generale $m > 2$.*

5.1.2.3 Algoritmo "twice around MST" per il (TSP)

Consideriamo nuovamente il problema del commesso viaggiatore (TSP); è possibile definire un'euristica greedy con errore relativo nel caso pessimo pari ad 1 se il grafo G è completo ed i costi sugli archi sono non negativi e rispettano la *diseguaglianza triangolare*:

$$c_{ij} + c_{jh} \leq c_{ih} \qquad \forall\, i\,,\, j\,,\, h\ . \tag{5.1}$$

L'algoritmo si basa sulla seguente osservazione: il costo di un albero di copertura di costo minimo sul grafo G fornisce una valutazione inferiore del costo del ciclo hamiltoniano ottimo del grafo. Infatti, sia C^* il ciclo hamiltoniano di costo minimo del grafo, e consideriamo il cammino P^* ottenuto eliminando un qualsiasi arco da C^*: il costo di P^* è non superiore a quello di C^* e non inferiore a quello dell'albero di copertura di costo minimo T^*, essendo P^* un particolare albero di copertura per il grafo. L'albero di copertura di costo minimo T^* è efficientemente calcolabile con gli algoritmi visti nel paragrafo B.4. Tale albero può essere trasformato in un cammino hamiltoniano che, se i costi soddisfano (5.1), non può avere costo maggiore di due volte il costo di T^*. La trasformazione è illustrata in Figura 5.5: inizialmente si considera il grafo orientato ottenuto da T^* duplicando gli archi ed orientando ogni coppia degli archi così ottenuti nelle due direzioni possibili. Si ottiene così (Figura 5.5(b)) un cammino hamiltiano orientato (non semplice) per il grafo, il cui costo (considerando i costi degli archi orientati pari a quelli degli archi non orientati originali) è pari a due volte il costo di T^*. Tale cammino può essere trasformato in un cammino hamiltoniano attraverso l'operazione di "scorciatoia" mostrata in Figura 5.5(c). Partendo da un nodo qualsiasi, ad esempio il nodo 1, si inizia a percorrere il ciclo, marcando tutti i nodi visitati: se il successore j di un nodo i visitato per la prima volta è un nodo già visitato, si continua a percorrere il ciclo finché non si incontra un nodo h non ancora visitato. A quel punto, al posto del cammino da i ad h sul ciclo hamiltoniano originale viene scelta la "scorciatoia" (i,h) ed il procedimento viene iterato. Quando viene visitato l'ultimo nodo, il ciclo si chiude aggiungendo l'arco fino al nodo 1 (ossia operando una scorciatoia dall'ultimo nodo visitato a 1). Da (5.1) segue immediatamente che ogni operazione di "scorciatoia" non incrementa la lunghezza del ciclo hamiltoniano: di conseguenza, per il ciclo hamiltoniano H^* ottenuto al termine del procedimento si ha

$$C(T^*) \leq C(C^*) \leq C(H^*) \leq 2C(T^*)$$

e quindi l'errore relativo è non superiore a 1. Questo algoritmo prende, per ovvie ragioni, il nome di euristica "twice around MST".

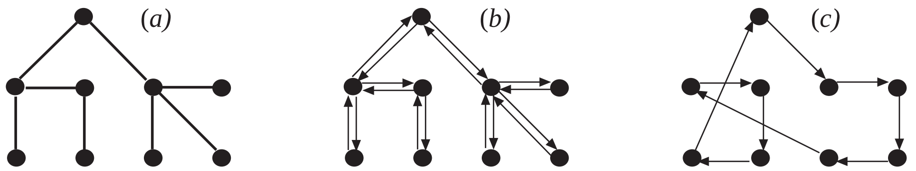


Figura 5.5: Funzionamento dell'euristica "twice around MST"

Esercizio 5.12 *Si proponga una descrizione formale, in pseudo-codice, dell'euristica "twice around MST".*

Esiste una versione di questa euristica, nota come *euristica di Christofides*, nella quale le operazioni di "scorciatoia" sono rimpiazzate dalla soluzione di un problema di assegnamento di costo minimo. Questo esempio mostra come la conoscenza di algoritmi per problemi "facili" possa essere utile per costruire approcci per problemi più "difficili".

5.1.2.4 Un algoritmo greedy per il Weighted Vertex Cover

Nei due esempi precedenti abbiamo prima introddoto gli algoritmi e poi, separatamente, valutato le loro prestazioni. È però possibile costruire algoritmi *facendosi guidare* dalle dimostrazioni di efficacia, ossia in modo tale che "naturalmente" se ne possano dimostrare buone proprietà di approssimazione. Esistono alcune tecniche generali per fare questo, una delle quali sarà illustrata per il problema del seguente esempio.

Esempio 5.5: Un problema di selezione di nodi
L'agenzia spionistica EMC, sempre un pò indietro rispetto alla concorrente CIA, ha deciso di dotarsi anch'essa di un sistema per monitorare tutte le comunicazioni di Internet. Per fare questo dispone di una mappa dei "backbones" di Internet attraverso un grafo (non orientato) $G = (V, E)$, in cui i nodi rappresentano i routers ed i lati rappresentano i collegamenti principali. L'agenzia dispone di apparecchiature che, se installate su un certo router, permettono di monitorare tutte le comunicazioni che transitano attraverso quel nodo. Installare l'apparecchiatura in un certo nodo i ha però un costo $c_i > 0$, dovuto in parte al costo dell'hardware ed in parte al costo di corrompere o intimidire i gestori del router per convincerli a permetterne l'installazione. L'agenzia dispone dei fondi necessari per installare le apparecchiature in tutti i nodi, ma ciò è evidentemente inutile: per poter monitorare tutte le comunicazioni, è sufficiente che per ogni lato $\{i,j\} \in E$ *almeno* uno dei nodi i e j sia monitorato. Per risparmiare soldi da poter inserire nei propri fondi neri, l'agenzia EMC deve quindi risolvere il seguente problema, detto di *Weighted Vertex Cover* (WVC): selezionare un sottoinsieme di nodi, $S \subseteq V$, di costo minimo (dove $C(S) = \sum_{i \in S} c_i$), che "copra" tutti i lati del grafo, ossia tale che per ogni $\{i,j\} \in E$ sia $i \in S$ oppure $j \in S$.

Un modello analitico per il problema è il seguente:

$$(WVC) \qquad \min \left\{ \; \sum_{i=1}^{n} c_i x_i \; : \; x_i + x_j \geq 1 \quad \{i,j\} \in E \;\; , \;\; x \in \mathbb{N}^n \; \right\} .$$

Chiaramente, qualsiasi soluzione ottima x^* del problema avrà solamente componenti 0 e 1, ossia $x^* \in \{0,1\}^n$, anche se non sono presenti vincoli espliciti $x_i \leq 1$: infatti, per qualsiasi soluzione ammissibile $\bar{x}$ con $\bar{x}_i > 1$ è possibile costruire un'altra soluzione ammissibile x' identica a $\bar{x}$ tranne per il fatto che $x'_i = 1$, e siccome i costi sono positivi x' ha un costo minore di $\bar{x}$.

Vogliamo costruire un algoritmo greedy per (WVC) che abbia buone proprietà di approssimazione: come abbiamo visto nei casi precedenti, per stimare l'errore compiuto è necessario per prima cosa ottenere una valutazione (in questo caso inferiore) del valore ottimo della funzione obiettivo. Come nel caso del problema dello zaino, otteniamo una tale valutazione considerando il *rilassamento continuo* di (WVC):

$$(\underline{WVC}) \qquad \min\Big\{\ \textstyle\sum_{i=1}^{n} c_i x_i \ :\ x_i + x_j \geq 1 \quad \{i,j\} \in E \ ,\ \ x \geq 0\ \Big\}$$

o, equivalentemente, il suo duale

$$(\underline{DWVC}) \qquad \max\Big\{\ \textstyle\sum_{\{i,j\}\in E} y_{ij} \ :\ \sum_{\{i,j\}\in S(i)} y_{ij} \leq c_i \quad i \in V \ ,\ \ y \geq 0\ \Big\}\ ,$$

dove $S(i)$ è l'insieme dei lati indicenti nel nodo i. Vogliamo costruire un algoritmo per (WVC) che costruisca contemporaneamente una soluzione ammissibile per ($\underline{\text{WVC}}$) *intera*, e quindi ammissibile per (WVC), ed una soluzione ammissibile per ($\underline{\text{DWVC}}$) che ci permetta di valutare la bontà della soluzione primale ottenuta. In effetti, ci serviremo della soluzione duale (parziale) per *guidare* la costruzione della soluzione primale: per questo, l'algoritmo viene detto *primale-duale*. Le condizioni degli scarti complementari per la coppia di problemi duali ($\underline{\text{WVC}}$) e ($\underline{\text{DWVC}}$) sono

$$x_i\big(\, c_i - \textstyle\sum_{\{i,j\}\in S(i)} y_{ij}\,\big) = 0 \qquad i \in V \tag{5.2}$$

$$y_{ij}\big(\, x_i + x_j - 1\,\big) = 0 \qquad \{i,j\} \in E\ . \tag{5.3}$$

Se le soluzioni primale e duale ottenute dall'algoritmo rispettassero sia (5.2) che (5.3), allora avremmo determinato una coppia di soluzioni ottime per ($\underline{\text{WVC}}$) e ($\underline{\text{DWVC}}$); dato che la soluzione primale sarà costruita in modo da essere ammissibile anche per (WVC), si sarebbe quindi ottenuta una soluzione ottima per (WVC) (si veda il Lemma 4.1). Naturalmente, essendo (WVC) un problema $\mathcal{NP}$-arduo non è pensabile che ciò sia sempre possibile: per questo l'algoritmo si limiterà ad assicurare che sia verificata (5.2), mentre permetterà violazioni in (5.3).

```
Procedure Greedy-WVC( G , c , S ) {
  S = ∅; Q = E;
  foreach( {i,j} ∈ E ) do y_ij = 0;
  do { {i,j} = Next(Q); Q = Q \ { {i,j} };
       y_ij = min{ c_i − Σ_{{i,h}∈S(i)} y_ih , c_j − Σ_{{j,h}∈S(j)} y_jh };
       if( c_i = Σ_{{i,h}∈S(i)} y_ih ) then {
          foreach( {i,h} ∈ E ) do Q = Q \ { {i,h} };
          S = S ∪ {i};
          }
       if( c_j = Σ_{{j,h}∈S(j)} y_jh ) then {
          foreach( {j,h} ∈ E ) do Q = Q \ { {j,h} };
          S = S ∪ {j};
          }
       } while( Q ≠ ∅ );
}
```

Procedura 5.2: Algoritmo *Greedy-WVC*

L'algoritmo mantiene in Q l'insieme dei lati non ancora "coperti" dalla soluzione corrente S, che inizialmente non contiene alcun nodo. Ad ogni passo seleziona un qualsiasi ;ato $\{i,j\}$ non ancora coperto ed aumenta il valore della corrispondente variabile y_{ij} al massimo valore possibile che non viola le condizioni (5.2) per i e j; questo valore è tale per cui, dopo l'aumento di y_{ij}, vale almeno una delle due condizioni

$$c_i = \textstyle\sum_{\{i,h\}\in S(i)} y_{ih} \qquad \text{e} \qquad c_j = \sum_{\{j,h\}\in S(j)} y_{jh}\ .$$

Se vale la prima condizione i viene aggiunto ad S e tutti i lati incidenti in i (che sono a questo punto coperti da S) sono eliminati da Q; analogamente, se vale la seconda condizione j viene aggiunto ad

S e tutti i lati incidenti in j sono eliminati da Q. È facile verificare che la soluzione y costruita dall'algoritmo è duale ammissibile ad ogni iterazione: lo è infatti sicuramente all'inizio (dato che i costi sono positivi), e non appena il vincolo duale corrispondente al nodo i diviene "attivo" tutti i lati incidenti in i vengono eliminati da Q, "congelando" i loro valori y_{ij} fino al termine dell'algoritmo e quindi assicurando che il vincolo resti verificato. A terminazione, la soluzione primale S "copre" tutti i nodi ($Q = \emptyset$), ed è quindi ammissibile.

Esempio 5.6: Algoritmo *Greedy-WVC*
Un esempio del funzionamento dell'algoritmo *Greedy-WVC* è mostrato in figura 5.6. L'istanza è mostrata in (a), con i costi indicati vicino a ciascun nodo.
La prima iterazione è mostrata in (b): viene selezionato il lato $\{1, 2\}$, e si pone $y_{12} = 1$, in modo tale che $y_{12} + y_{16} + y_{14} = 1 + 0 + 0 = 1 = c_1$. Quindi, il nodo 1 viene inserito in S ed i lati $\{1, 2\}$, $\{1, 6\}$ e $\{1, 4\}$ risultano quindi coperti.
La seconda iterazione è mostrata in (c): viene selezionato il lato $\{2, 6\}$, e si pone $y_{26} = 2$; in questo modo risulta sia $y_{12} + y_{26} + y_{23} = 1 + 2 + 0 = 3 = c_2$ che $y_{16} + y_{26} + y_{36} + y_{46} + y_{56} = 0 + 2 + 0 + 0 + 0 = 2 = c_6$, e quindi sia il nodo 2 che il nodo 6 vengono aggiunti a S, coprendo i corrispondenti lati adiacenti.
La terza ed ultima iterazione è mostrata in (d): viene selezionato il lato $\{3, 5\}$, e si pone $y_{35} = 4$, in modo che risulti $y_{35} + y_{45} + y_{56} = 4 + 0 + 0 = 4 = c_5$, e quindi che 5 sia aggiunto a S coprendo gli ultimi lati e determinando quindi la soluzione ammissibile $S = \{ 1 , 2 , 5 , 6 \}$ d costo 10.

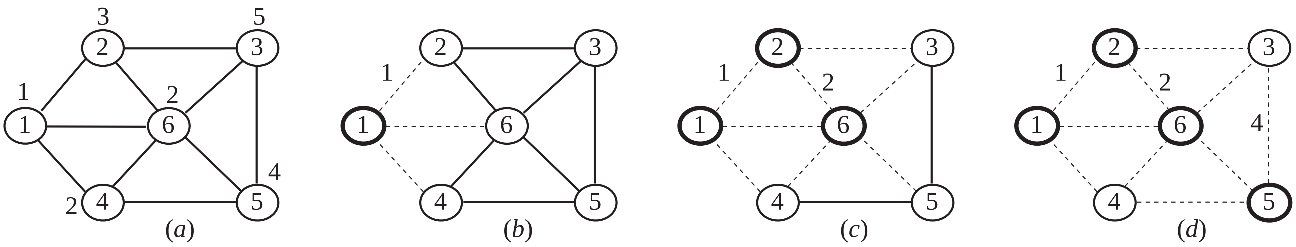


Figura 5.6: Esecuzione dell'algoritmo *Greedy-WVC*

Valutiamo adesso l'efficacia dell'algoritmo *Greedy-WVC*. L'osservazione fondamentale è che ad ogni passo dell'algoritmo si ha

$$c(S) = \sum_{i \in S} c_i \leq 2 \sum_{\{i,j\} \in E} y_{ij} \ ;$$

infatti, quando il generico nodo i è stato inserito in S si aveva $c_i = \sum_{\{i,h\} \in S(i)} y_{ih}$ (e gli y_{ih} non sono più cambiato da quell'iterazione), ma ciascun y_{ij} può contribuire ad al più due sommatorie, quella corrispondente ad i e quella corrispondente a j. Di conseguenza si ha

$$\sum_{\{i,j\} \in E} y_{ij} \leq z(\underline{DWVC}) = z(\underline{WVC}) \leq z(WVC) \leq c(S) \leq 2 \sum_{\{i,j\} \in E} y_{ij}$$

da cui $R_{Greedy-WVC} \leq 1$; la soluzione ottenuta dall'algoritmo *Greedy-WVC* può costare al più il doppio della soluzione ottima. Quindi, *a priori* possiamo affermare che l'algoritmo *Greedy-WVC* compie al massimo un errore del 100%; questa valutazione può poi essere raffinata *a posteriori* esaminando i risultati per l'istanza specifica. Nel caso di figura 5.6(a) si ha ad esempio

$$\sum_{\{i,j\} \in E} y_{ij} = 7 \leq z(WVC) \leq c(S) = 10$$

da cui $R_{Greedy-WVC} \leq (10 - 7)/7 \approx 0.428$, ossia la soluzione ottenuta da *Greedy-WVC* è al più il 42.8% più costosa della soluzione ottima. Si noti che anche questa valutazione non è esatta: è possibile verificare (enumerando tutte le possibili soluzioni) che la soluzione ottenuta da *Greedy-WVC* è in effetti *ottima* per l'istanza in questione. La soluzione duale ottenuta non è però in grado di dimostrare l'ottimalità della soluzione primale.

L'idea alla base dell'algoritmo *Greedy-WVC*, ossia quella di costruire contemporaneamente sia una soluzione primale intera che una duale ammissibile per il duale del rilassamento continuo, è generale e può essere applicata per produrre algoritmi con garanzia sul massimo errore compiuto per molti altri problemi combinatori. Questa è comunque soltanto una delle molte tecniche possibili, per ulteriori dettagli ed approfondimenti si rimanda alla letterature citata.

Esercizio 5.13 *Il problema (WVC) è un caso particolare del problema di copertura (PC) in cui tutti gli insiemi F_j hanno esattamente due elementi. Si estenda quindi l'algoritmo primale-duale per (WVC) a (PC): che valutazione può essere data sull'errore commesso di tale algoritmo?*

Esercizio 5.14 *Si mostri che l'algoritmo CUD per il problema dello zaino può essere interpretato come un algoritmo primale-duale (suggerimento: per qualsiasi valore di $\bar{y}$, la migliore soluzione $\bar{w}$ duale ammissibile compatibile con quel valore di $\bar{y}$ si ottiene ponendo $\bar{w}_i = c_i/a_i - \bar{y}$ se $c_i/a_i > \bar{y}$ e $\bar{w}_i = 0$ altrimenti).*

5.1.3 Matroidi

Nei paragrafi precedenti abbiamo visto che per alcuni algoritmi greedy è possibile ricavare valutazioni, a priori o a posteriori, sull'errore compiuto, e che un algoritmo greedy può anche essere esatto. Sorge quindi spontanea la domanda: è possibile caratterizzare i problemi per i quali gli algoritmi greedy hanno errore nullo, ossia sono esatti? In effetti questo è possibile, sia pure per una classe particolare di problemi e di algoritmi greedy, ossia facendo un alcune ulteriori assunzioni sul problema e sull'algoritmo. La prima assunzione è la seguente:

- (E, F) sono un *sistema di insiemi indipendenti*, ossia $A \in F$ e $B \subseteq A$ implica $B \in F$;

Da questo deriva immediatamente che $\emptyset \in F$; inoltre, possiamo assumere senza perdita di generalità che sia $\{e\} \in F$ $\forall e \in E$, in quanto se fosse $\{e\} \notin F$ per un qualche $e \in E$ allora nessuno dei sottoinsiemi in F potrebbe contenere e, e quindi potremmo rimuovere e da E (ciò è analogo a quanto già osservato per gli oggetti nel problema dello zaino). Dato un sistema di insiemi indipendenti (E, F), diciamo che S è *massimale* per E se $S \in F$ e $S \cup \{e\} \notin F$ per ogni $e \in E \setminus S$, ossia se non esiste nessun elemento di F che contiene strettamente S. Più in generale, S è massimale per $E' \subseteq E$ se $S \in F$, $S \subseteq E'$ e $S \cup \{e\} \notin F$ per ogni $e \in E' \setminus S$. La seconda assunzione è:

- a ciascun elemento $e_i \in E$ è associato un costo $c_e \geq 0$, ed il problema da risolvere è

$$\text{(MSII)} \qquad \max \left\{ \; c(S) = \textstyle\sum_{e \in S} c_e \; : \; S \text{ massimale per } E \; \right\} \; .$$

Si noti che, anche per via di questa assunzione, la teoria sviluppata in questo paragrafo non copre tutti i casi di algoritmi greedy che determinano l'ottimo di un problema di OC; infatti, il problema (MCMS), che è risolto all'ottimo da un algoritmo greedy (cf. §5.1.1.5), non ha una funzione obiettivo di questo tipo. Si consideri un grafo non orientato e connesso $G = (V, E)$: (E, F), dove F è la famiglia dei sottoinsiemi di lati che non inducono cicli su G, è chiaramente un sistema di insiemi indipendenti, e gli alberi di copertura per G sono chiaramente i suoi insiemi massimali. Quindi, (MST) è un problema di tipo (MSII). Si noti che la funzione obiettivo *non* è di questo tipo in alcuni dei problemi per i quali abbiamo presentato algoritmi greedy, come ad esempio (MMMS) e (MCMS). Per un problema nella forma (MSII) esiste un'implementazione "naturale" della procedura *Best*; la terza assunzione è infatti

- la sottoprocedura *Best* dell'algoritmo ritorna l'elemento $e \in Q$ di costo c_e massimo.

Si noti come molte delle regole *Best* che abbiamo discusso *non* rientrano in questa categoria, quali ad esempio quelle per (TSP) e due di quelle per (PC). L'algoritmo di Kruskal è un esempio di algoritmo greedy che rispetta le assunzioni (se invertiemo il segno dei costi o, alternativamente, vogliamo risolvere il problema dall'albero di copertura di costo massimo) e che determina la soluzione ottima per il problema. Vogliamo caratterizzare i problemi di tipo (MSII) per cui un algoritmo greedy di questo tipo è esatto.

Un sistema di insiemi indipendenti (E, F) è detto *matroide* se tutti gli insiemi massimali hanno la stessa cardinalità; più precisamente, si richiede che

$$\forall E' \subseteq E, \; \text{se } I \text{ e } J \text{ sono massimali per } E', \; \text{allora } |I| = |J|. \tag{5.4}$$

La proprietà (5.4) è *necessaria* affinchè l'algoritmo greedy possa risolvere (MSII) per *qualsiasi* scelta dei costi c_i. Infatti, supponiamo che la proprietà non valga, ossia siano I e J due insiemi massimali rispetto ad un certo $V \subseteq E$ tali che $|I| < |J| \leq n$. Poniamo allora $c_e = 1 + \epsilon$ per $e \in I$, $c_e = 1$ per $e \in J \setminus I$, e $c_e = 0$ per tutti gli altri elementi di E. L'algoritmo greedy pone in S inizialmente tutti gli elementi di I, poi esamina e scarta tutti gli elementi di $J \setminus I$ ($V \supseteq I \cup J$ e I è massimale per V), infine eventualmente aggiunge ad S altri elementi di costo nullo, ottenendo una soluzione di costo $|I|(1 + \epsilon)$; la soluzione J di costo $|J| \geq |I| + 1$ è migliore di I se scegliamo $\epsilon < 1/|I|$. Si noti che dalla proprietà (5.4) segue che l'assunzione $c_e \geq 0$ può essere fatta senza perdita di generalità: come abbiamo visto per il caso di (MST), dato che ogni soluzione ammissibile del problema ha la stessa cardinalità è possibile sommare al costo di ogni elemento un'opportuna costante C lasciando invariato l'insieme delle soluzioni ottime.
Dato un sistema di insiemi indipendenti (E, F), il *rango* di un qualsiasi insieme $E' \subseteq E$ è

$$rango(E') = \max \left\{ \, |\, S \,| \, : \, S \text{ massimale per } E' \, \right\} \; ;$$

se (E, F) è un matroide, allora la funzione *rango* può essere calcolata facilmente.

Esercizio 5.15 *Si dimostri che l'algoritmo greedy con costi $c_e = 1$ per $e \in E'$ e $c_e = 0$ altrimenti determina una soluzione S tale che $c(S) = rango(E')$ (suggerimento: se al termine dell'algoritmo $S \cup E'$ non fosse massimale per E', allora dovrebbe esistere un $S' \subseteq E'$ con $|S'| > |S|$; si consideri cosa accade al momento in cui l'algoritmo greedy esamina l'ultimo elemento di $S \cup S'$ e si usi (5.4) per ottenere una contradizione).*

Quindi, l'algoritmo greedy risolve all'ottimo almeno alcuni problemi di tipo (MSII). Vogliamo ora mostrare che l'algoritmo greedy risolve all'ottimo tutti i problemi di tipo (MSII) per qualsiasi scelta dei costi; un modo interessante per farlo è quello di considerare la seguente formulazione *PLI* di (MSII)

$$\text{(MSII-PLI)} \qquad \begin{array}{lll} \max & \sum_{e \in E} c_e x_e & \\ & \sum_{e \in S} x_e \leq rango(S) & \emptyset \subset S \subseteq E \\ & x_e \in \mathbb{N} & e \in E \end{array}$$

Esercizio 5.16 *Si verifichi che tutte le soluzioni ammissibili di (MSII-PLI) hanno $x_e \in \{0, 1\}$ per ogni $e \in E$ e che sono tutti e soli i vettori di incidenza degli elementi di F, ossia vettori nella forma*

$$x_e = \begin{cases} 1 & se\ e \in S \\ 0 & altrimenti \end{cases}$$

per un qualche $S \in F$ (suggerimento: per $S \in F$ si ha $rango(S) = |S|$ mentre per $S \notin F$ si ha $rango(S) < |S|$).

Come abbiamo visto nei paragrafi 5.1.2.1 e 5.1.2.4, consideriamo il rilassamento continuo di (MSII-PLI), $(\underline{\text{MSII-PLI}})$, ottenuto sostituendo il vincolo $x_e \in \mathbb{N}$ con $x_e \geq 0$, ed il suo duale

$$(\underline{\text{DMSII}}) \qquad \begin{array}{lll} \min & \sum_{S \subseteq E} rango(S) y_S & \\ & \sum_{S \,:\, e \in S} y_S \geq c_e & e \in E \\ & y_S \geq 0 & \emptyset \subset S \subseteq E \end{array} \quad .$$

Possiamo mostrare che l'algoritmo greedy costrusce una soluzione primale ammissibile *intera* per $(\underline{\text{MSII-PLI}})$ ed una soluzione duale ammissibile per $(\underline{\text{DMSII}})$ che rispettano le condizioni degli scarti complementari

$$x_e \big(\textstyle\sum_{S \,:\, e \in S} y_S - c_e \big) = 0 \qquad e \in E \; , \tag{5.5}$$

$$y_S \big(rango(S) - \textstyle\sum_{e \in S} x_e \big) = 0 \qquad \emptyset \subset S \subseteq E \; . \tag{5.6}$$

Per semplificare la notazione, supponiamo che sia $E = \{1, 2, \ldots, n\}$ e che gli oggetti siano ordinati per costo non crescente, ossia $c_1 \geq c_2 \geq \ldots \geq c_n \geq 0$; introduciamo inoltre gli insiemi $S(1) = \{1\}$, $S(2) = \{1, 2\}$, $\ldots$, $S(e) = \{1, 2, \ldots, e\}$. Possiamo allora riscrivere l'algoritmo greedy sotto forma di un algoritmo primale-duale per la coppia $(\underline{\text{MSII-PLI}})$, $(\underline{\text{DMSII}})$:

```
Procedure Greedy-PD( E , F , S , y ) {
  y_{1} = c_1; S = S(1) = {1};                    /* x_1 = 1 */
  for( e = 2,...,n ) do {
    y_S(e) = c_e; y_S(e-1) = y_S(e-1) - c_e;
    if( S ∪ {e} ∈ F ) then S = S ∪ { e };         /* x_e = 1 */
                /* else                              x_e = 0 */
  }
}
```

Procedura 5.3: Algoritmo *Greedy-PD*

È facile verificare che, data l'assunzione sulla procedura *Best* e l'ordinamento di E, la procedura *Greedy-PD* produce la stessa soluzione S della procedura *Greedy*. Inoltre, alla generica iterazione e la soluzione primale x (implicitamente) calcolata è ammissibile e rispetta le condizioni (5.5) e (5.6) con la soluzione duale y (prendendo ovviamente $y_S = 0$ per tutti gli S ai quali non è esplicitamente dato un valore diverso), in quanto si ha che:

- $S \subseteq S_e$;
- siccome (E, F) è un matroide, $rango(S) = rango(S(e))$;
- $\sum_{S\,:\,h\in S} y_S = c_h$ per $h = 1, \ldots, e$.

All'iterazione e viene soddisfatto il vincolo duale relativo all'elemento e, senza violare nessuno dei vincoli relativi agli elementi $h < e$; di conseguenza, a terminazione la soluzione duale y costruita dall'algoritmo è ammissibile, e quindi dimostra che la soluzione primale S ottenuta è ottima per (MSII-PLI).

Esempio 5.7: Algoritmo *Greedy-PD*
Esemplifichiamo i concetti della dimostrazione applicando l'algoritmo *Greedy-PD* al semplice problema di (MST) su un grafo completo con 3 nodi in cui $c_{12} = 6$, $c_{13} = 3$ e $c_{23} = 2$. In questo caso l'insieme E è l'insieme dei lati del grafo; per semplificare la notazione chiameremo $a = \{1, 2\}$, $b = \{1, 3\}$ e $c = \{2, 3\}$, per cui $E = \{a, b, c\}$. I problemi (MSII) e (DMSII) in questo caso sono

$$
\begin{array}{lrcrcrcl}
\max & 6x_a & + & 3x_b & + & 2x_c & & \\
& x_a & + & x_b & + & x_c & \leq & 2 \\
& x_a & + & x_b & & & \leq & 2 \\
& x_a & & & + & x_c & \leq & 2 \\
& & & x_b & + & x_c & \leq & 2 \\
& x_a & & & & & \leq & 1 \\
& & & x_b & & & \leq & 1 \\
& & & & & x_c & \leq & 1 \\
& x_a & , & x_b & , & x_c & \geq & 0
\end{array}
\qquad
\begin{array}{ll}
\min & 2y_{abc} + 2y_{ab} + 2y_{ac} + 2y_{bc} + y_a + y_b + y_c \\
& y_{abc} + y_{ab} + y_{ac} + y_a \geq 6 \\
& y_{abc} + y_{ab} + y_{bc} + y_b \geq 3 \\
& y_{abc} + y_{ac} + y_{bc} + y_c \geq 2 \\
& y_{abc}\,,\ y_{ab}\,,\ y_{ac}\,,\ y_{bc}\,,\ y_a\,,\ y_b\,,\ y_c \geq 0
\end{array}
$$

All'inizializzazione si pone $x_a = 1$ e $y_{\{a\}} = 6$. Alla prima iterazione si pone $y_{\{a,b\}} = 3$ e $y_{\{a\}} = 6 - 3 = 3$; siccome b non forma cicli, si pone $x_b = 1$. Alla terza iterazione si pone $y_{\{a,b,c\}} = 2$ e $y_{\{a,b\}} = 3 - 2 = 1$; siccome c forma un ciclo, si pone $x_c = 0$. È immediato verificare che y è duale ammissibile: tutti i vincoli del duale sono soddisfatti come uguaglianza. È anche facile verificare che le condizioni degli scarti complementari sono rispettate, ma più semplicemente si può notare che il costo della soluzione primale è $6 + 3 = 9$ ed il costo della soluzione duale è $2 \cdot 2 + 2 \cdot 1 + 1 \cdot 3 = 9$.

Un interessante corollario dell'analisi appena svolta è il seguente:

Corollario 5.1 *Se (E, F) è un matroide allora (MSII) gode della proprietà di integralità.*

Quindi, i problemi di ottimizzazione su matroidi godono di una proprietà analoga a quella dei problemi di flusso (si veda il Teorema 3.11): esiste una formulazione di *PLI* "naturale" per il problema il cui rilassamento continuo ha sempre soluzioni intere, ossia tutti i vertici del poliedro sono interi. A differenza dei problemi di flusso, la formulazione è di dimensione esponenziale, ma, come abbiamo visto nel paragrafo 4.2.3, ammette un separatore polinomiale.
I matroidi sono dunque strutture combinatorie i cui corrispondenti problemi di ottimizzazione sono risolti esattamente dall'algoritmo greedy. Esempi di matroidi sono:

- E è l'insieme degli archi di un grafo non orientato ed F è la famiglia dei sottoinsiemi di archi che non inducono cicli: questo tipo viene detto un *matroide grafico*;
- $E = \{ A_1 , A_2 , \dots , A_n \}$ con $A_i \in \mathbb{R}^m$, e F è la famiglia di tutti gli insiemi di vettori di E linearmente indipendenti: questo viene detto un *matroide matrico*;
- E è un insieme, $P = \{ E_1 , E_2 , \dots , E_m \}$ è una sua partizione ed $F = \{ S \subseteq E \ : \ |I \cap E_j| \leq 1 \quad j = 1, \dots , m \}$; questo viene detto un *matroide di partizione.*

Naturalmente molte strutture combinatorie *non* sono matroidi: si consideri ad esempio il caso in cui E è l'insieme degli archi di un grafo bipartito non orientato $G = (O \cup D, E)$ ed F è la famiglia dei suoi accoppiamenti (si veda il paragrafo 3.5). È facile verificare che questo non è un matroide; infatti, non tutti gli insiemi indipendenti massimali hanno la stessa cardinalità. Del resto, abbiamo visto che per risolvere i problemi di accoppiamento sono necessari algoritmi "più complessi" del semplice greedy. È interessante notare però che i nodi appartenenti a O ed a D inducono su E due diverse partizioni

$$P' = \{ \ S(i) \ , \ i \in O \ \} \qquad P'' = \{ \ S(i) \ , \ i \in D \ \}$$

corrispondenti alle stelle dei due insiemi di nodi. A partire da tali partizioni possiamo definire due matroidi di partizione (E, F') ed (E, F''): chiaramente $F = F' \cap F''$, ossia il sistema di insiemi indipendenti (E, F) che rappresenta gli accoppiamenti è l'intersezione dei due matroidi (E, F') e (E, F''), da cui si deduce che, in generale, l'intersezione di due matroidi *non* è un matroide. Per quanto gli algoritmi greedy non siano esatti per i problemi di accoppiamento, esistono algoritmi polinomiali per risolverli; in effetti si può dimostrare che qualsiasi problema di ottimizzazione che può essere espresso come l'*intersezione* di *due* matroidi ammette algoritmi polinomiali che ricalcano il funzionamento degli algoritmi per i problemi di accoppiamento visti al paragrafo 3.5.
Abbiamo anche già osservato al paragrafo 1.2.4.1 che il (TSP) può essere visto come un problema di ottimizzazione sull'intersezione di un problema di accoppiamento e di uno di albero minimo; in altri termini, (TSP) è problema di ottimizzazione sull'intersezione di *tre* matroidi. Ciò dimostra che i problema di ottimizzazione sull'intersezione di tre matroidi sono invece, in generale, $\mathcal{NP}$-ardui.

5.2 Algoritmi di ricerca locale

Gli algoritmi di ricerca locale sono basati su un'idea estremamente semplice ed intuitiva: data una soluzione ammissibile, si esaminano le soluzioni ad essa "vicine" in cerca di una soluzione "migliore" (tipicamente, con miglior valore della funzione obiettivo); se una tale soluzione viene trovata essa diventa la "soluzione corrente" ed il procedimento viene iterato, altrimenti—ossia quando nessuna delle soluzioni "vicine" è migliore di quella corrente—l'algoritmo termina avendo determinato un *ottimo locale* per il problema (si consideri la Figura 4.2). Elemento caratterizzante di un algoritmo di questo tipo è la definizione di "vicinanza" tra le soluzioni. In generale, se F è l'insieme ammissibile del problema in esame, possiamo definire una *funzione intorno* $I : F \rightarrow 2^F$: l'insieme $I(x)$ è detto *intorno* di x, e contiene le soluzioni considerate "vicine" ad x. Opportunamente definita la funzione intorno, un algoritmo di ricerca locale può essere schematizzato come segue:

```
procedure Ricerca_Locale ( F , c , x ) {
  x = Ammissibile(F);
  while( σ(x) ≠ x ) do
    x = σ(x)
  }
```

Procedura 5.4: Algoritmo di Ricerca Locale

L'algoritmo necessita di una soluzione ammissibile x^0 da cui partire: una tale soluzione può essere costruita, ad esempio, usando un algoritmo greedy. L'algoritmo genera una sequenza di soluzioni ammissibili $\{ x^0 , x^1 , \dots , x^k , \dots \}$ tale che $x^{i+1} = \sigma(x^i)$; la tipica implementazione di σ, basata sulla funzione intorno I, è

$$\sigma(x) = \text{argmin}\{ \ c(y) \ : \ y \in I(x) \ \} \ . \tag{5.7}$$

In questo modo, ad ogni passo dell'algoritmo viene risolto un problema di ottimizzazione ristretto all'intorno considerato. Più in generale, si può definire σ in modo tale che, per ogni x, fornisca un qualsiasi elemento $y \in I(x)$ con $c(y) < c(x)$, se un tale elemento esiste, oppure x stesso se un tale elemento non esiste; in questo modo, ad ogni passo dell'algoritmo viene risolto un problema decisionale. Si ha quindi $x = \sigma(x)$ quando nell'intorno di x non esiste nessuna soluzione migliore di x. In generale, la soluzione determinata dall'algoritmo di ricerca locale non è ottima per il problema, ma solamente un *ottimo locale* relativamente alla funzione intorno scelta. I è detta una funzione intorno *esatta* per un problema P se l'algoritmo di ricerca locale basato sulla corrispondente trasformazione σ è in grado di fornire una soluzione ottima per ogni istanza di P e comunque scelto il punto di partenza x_0. Questo modo di operare è estremamente generale, ed infatti moltissimi algoritmi per problemi di ottimizzazione—tra cui quasi tutti quelli che abbiamo descritto nei capitoli precedenti—possono essere classificati come algoritmi di ricerca locale.

Esercizio 5.17 *Per tutti gli algoritmi presentati nei capitoli precedenti si discuta se essi possono o no essere considerati algoritmi di ricerca locale, fornendo la definizione delle relative funzioni intorno.*

La definizione della funzione intorno è quindi la parte fondamentale della definizione di un algoritmo di ricerca locale. Usualmente, si richiede che I possieda le due seguenti proprietà:

1) $x \in F \Longrightarrow x \in I(x)$;

2) $x, y \in F \Longrightarrow$ esiste un insieme finito $\{z^0, z^1, \dots, z^p\} \subseteq F$ tale che
$z^0 = x$, $z^i \in I(z^{i-1})$ per $i = 1, 2, \dots, p$, $z^p = y$.

La proprietà 1) richiede che ogni soluzione appartenga all'intorno di se stessa, mentre la proprietà 2) richiede che sia teoricamente possibile per l'algoritmo di ricerca locale raggiungere in un numero finito di passi qualsiasi soluzione $y \in F$ (ad esempio quella ottima) a partire da qualsiasi altra soluzione $x \in F$ (ad esempio quella iniziale x^0). Si noti che la proprietà 2) non garantisce affatto che un algoritmo di ricerca locale, partendo da x, possa effettivamente arrivare a y: per questo sarebbe anche necessario che $c(z_i) < c(z_{i-1})$ per $i = 1, 2, \dots, p$, il che in generale non è vero. Inoltre, in pratica si utilizzano anche funzioni intorno che non soddisfano questa proprietà.

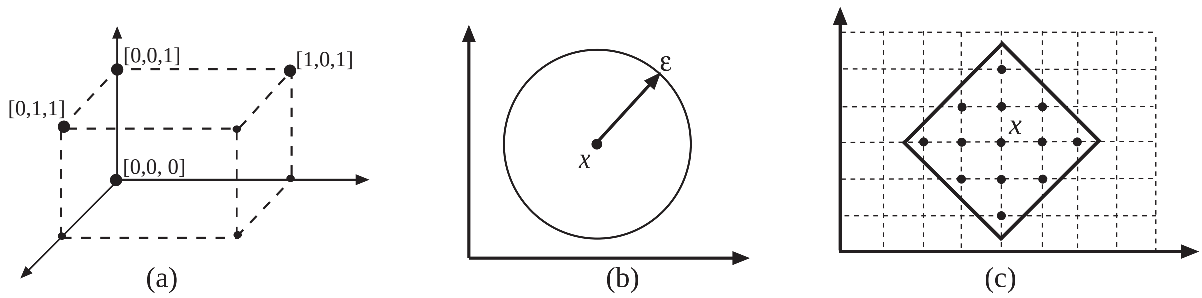


Figura 5.7: Alcuni esempi di funzioni intorno

Alcuni esempi di funzioni intorno sono mostrati in Figura 5.7, in particolare

- Figura 5.7(a): $F \subseteq \{0,1\}^n$, $I(x) = \{\ y \in F\ :\ \sum_i |x_i - y_i| \leq 1\ \}$ (è mostrato l'intorno di $[0,0,1]$);
- Figura 5.7(b): $F \subseteq \mathbb{R}^n$, $I_\varepsilon(x) = \{\ y \in F\ :\ \|\, x - y\, \| \leq \varepsilon\ \}$ (intorno Euclideo);
- Figura 5.7(c): $F \subseteq \mathbb{Z}^n$, $I(x) = \{\ y \in F\ :\ \sum_i |\, x_i - y_i\, | \leq 2\ \}$ (i punti evidenziati costituiscono l'intorno di x).

I tre intorni precedenti sono molto generali; usualmente, gli intorni utilizzati negli algoritmi di ricerca locale sono più specifici per il problema trattato. Inoltre, spesso la definizione di intorno è fornita in modo implicito, ovvero definendo una serie di operazioni ("mosse") che trasformano una soluzione ammissibile del problema in un'altra soluzione ammissibile.

5.2.1 Esempi di algoritmi di ricerca locale

Discutiamo adesso funzioni intorno per alcuni problemi di *OC*, in modo da fornire una panoramica di alcune delle principali metodologie utilizzate per costruire approcci di ricerca locale.

5.2.1.1 Il problema dello zaino

Si consideri il problema dello zaino (KP) definito al paragrafo 1.2.2.1. Un primo esempio di funzione intorno per questo problema potrebbe essere identificato dalle seguenti "mosse di inserzione e cancellazione": data una soluzione ammissibile dello zaino, si costruisce una diversa soluzione ammissibile inserendo nello zaino uno degli oggetti attualmente non selezionati – e per il quale esiste ancora sufficiente capacità residua – oppure togliendo dallo zaino uno degli oggetti attualmente selezionati (il che non può che determinare una nuova soluzione ammissibile). Data una soluzione ammissibile x per il problema dello zaino, che possiamo considerare un vettore di n variabili binarie x_i, $i = 1, \ldots, n$, questa funzione intorno, che chiameremo I_1, associa ad x tutte le soluzioni ammissibili che possono essere ottenute trasformando un singolo x_i da 0 a 1 o viceversa; si noti che questa è esattamente la funzione intorno rappresentata in figura 5.7(a). Tali soluzioni sono quindi al più n. È quindi facile verificare che la funzione σ data da (5.7) può essere calcolata in tempo lineare $O(n)$ nel numero di oggetti dello zaino.

Esercizio 5.18 *Si descriva formalmente, in pseudo-codice, una procedura che calcola la funzione σ, in $O(n)$, per la funzione intorno I_1.*

È però facile osservare che questo intorno non permette sicuramente di migliorare le soluzioni ottenute dall'algoritmo *greedy* descritto nel paragrafo 5.1.1.1. Infatti, al termine di quell'algoritmo lo zaino non ha capacità residua sufficiente per nessuno degli oggetti non selezionati: se l'avesse allora l'avrebbe avuta anche al momento in cui l'oggetto è stato esaminato (la capacità residua è non crescente nel corso dell'algoritmo), e quindi l'oggetto sarebbe stato inserito. Inoltre, per l'ipotesi che i costi siano non negativi togliere un oggetto dallo zaino non può migliorare (aumentare) il valore della fuunzione obiettivo. In altri termini, l'algoritmo *greedy* produce un *ottimo locale* rispetto all'intorno I_1. Se si volesse utilizzare un algoritmo di ricerca locale per tentare di migliorare la soluzione ottenuta dall'algoritmo *greedy* occorrerebbe quindi definire un intorno diverso. Ad esempio, si potrebbe utilizzare la funzione intorno I_2 che, oltre alle mosse di inserzione e cancellazione, utilizza anche "mosse di scambio" tra un oggetto selezionato ed uno non selezionato. In altri termini, la funzione associa ad x tutte le soluzioni (ammissibili) x' che differiscono da x in *al più* due posizioni; se x' differisce da x in i e j, con $i \neq j$, deve essere $x_i = 0$ e $x_j = 1$, e quindi $x'_i = 1$ e $x'_j = 0$. È facile verificare che la funzione σ data da (5.7) per I_2 può essere calcolata in tempo $O(n^2)$, in quanto il numero di soluzioni $x' \in I(x)$ è certamente minore o uguale al numero di coppie (i, j) con i e j in $1, \ldots, n$, e la verifica dell'ammissibilità di una mossa può essere fatta in $O(1)$.

Esercizio 5.19 *Si descriva formalmente, in pseudo-codice, una procedura che calcola la funzione σ, in $O(n^2)$, per la funzione intorno I_2.*

Le soluzioni prodotte dall'algoritmo *greedy* possono *non* essere ottimi locali rispetto a questo nuovo intorno.

Esempio 5.8: Mosse di scambio per il problema dello zaino
Si consideri la seguente istanza del problema dello zaino:

$$\begin{array}{lccccccccccc}
\max & 2x_1 & + & 8x_2 & + & 5x_3 & + & 6x_4 & + & x_5 & & \\
 & 3x_1 & + & 2x_2 & + & 4x_3 & + & 6x_4 & + & 3x_5 & \leq & 8 \\
 & x_1 & , & x_2 & , & x_3 & , & x_4 & , & x_5 & \in & \{0,1\}
\end{array}$$

L'ordine CUD è: $2, 3, 4, 1, 5$. Pertanto, l'euristica Greedy CUD determina la soluzione $x = [\,0\,,\,1\,,\,1\,,\,0\,,\,0\,]$ di valore $cx = 13$. Ma scambiando l'oggetto 3, nello zaino, con l'oggetto 4, fuori dallo zaino, si ottiene la soluzione $x' = [\,0\,,\,1\,,\,0\,,\,1\,,\,0\,]$, pure ammissibile, di valore $cx' = 14$. Pertanto, x non è un ottimo locale rispetto all'intorno I_2.

Esercizio 5.20 *Per ciascuno degli altri due ordinamenti (costi non crescenti e pesi non decrescenti), si costruisca (se esiste) un'istanza del problema dello zaino per cui la soluzione prodotta dall'algoritmo greedy con l'ordinamento dato non sia un ottimo locale rispetto all'intorno* I_2*.*

Analizzando ulteriormente le mosse si possono poi scoprire proprietà che risultano utili nell'implementazione dell'algoritmo. Ad esempio, si consideri il caso di due oggetti i e j che hanno lo stesso costo; se hanno anche lo stesso peso allora qualsiasi scambio è inutile. Se invece si ha $c_i = c_j$ e $a_i < a_j$, allora è facile vedere che se esiste una soluzione ottima che contiene j allora esiste anche una soluzione ottima che contiene i al posto di j (la si ottiene semplicemente scambiando i con j). Si può quindi operare in modo tale che in ciascuna soluzione generata siano presenti, tra tutti gli oggetti con un dato costo, solo quelli di peso minore, e quindi in modo tale che non vengano mai scambiati oggetti con lo stesso costo, evitando di valutare mosse "inutili".

La funzione intorno I_2 *domina* la funzione intorno I_1, ossia $I_2(x) \supseteq I_1(x)$ per ogni x. In questo caso si può affermare che I_2 è "migliore" di I_1 nel senso che tutti gli ottimi locali rispetto a I_2 sono anche ottimi locali rispetto a I_1, ma il viceversa può non essere vero. Infatti, x è un ottimo locale rispetto a I_2 se $c(x') \leq c(x)\ \forall x' \in I_2(x)$, e questo sicuramente implica che $c(x') \leq c(x)\ \forall x' \in I_1(x)$. Ciò non significa che un algoritmo di ricerca locale che usa I_1 determinerà necessariamente una soluzione peggiore di quella determinata da un algoritmo di ricerca locale che usa I_2, perchè le "traiettorie" seguite dai due algoritmi nello spazio delle soluzioni ammissibili saranno diverse. È però vero che, avendo I_1 "più" minimi locali, un algoritmo di ricerca locale che usa I_1 può arrestarsi una volta giunto ad una certa soluzione dalla quale un algoritmo di ricerca locale che usa I_2 proseguirebbe invece la ricerca, come l'esempio precedente mostra.
È interessante notare come ciascuna mossa di scambio sulla coppia (i, j) possa essere considerata la *concatenazione* di due mosse, rispettivamente una di cancellazione di j ed una di inserzione di i. Il motivo per cui l'intorno basato sullo scambio è "più potente" è che il risultato della concatenazione delle due mosse viene "visto" immediatamente. Infatti, nessuna mossa di cancellazione presa a sè potrebbe mai essere accettata da un algoritmo di ricerca locale, in quanto comporta (se i costi sono tutti positivi) un peggioramento della funzione obiettivo. Cancellare un oggetto può però permettere di inserirne uno di costo maggiore, ottenendo un miglioramento complessivo del valore della funzione obiettivo; quindi la concatenazione delle due mosse è conveniente, anche se la prima mossa presa singolarmente non lo è. In generale, intorni basati su "mosse complesse", ottenute concatenando un certo numero di "mosse semplici", permettono di "vedere le conseguenze" delle singole mosse semplici e quindi di effettuare mosse che non sarebbero accettabili da intorni basati direttamente sulle "mosse semplici". Per contro, spesso, come nell'esempio precedente, valutare la funzione σ per intorni basati su "mosse complesse" è più costoso. Inoltre, si noti che questo comportamento è dovuto al fatto che, nell'algoritmo di ricerca locale, si insiste sull'ottenere un miglioramento ad ogni passo. L'osservazione precedente suggerisce due strategie, in qualche modo alternative, per migliorare la qualità delle soluzioni determinate da un algoritmo di ricerca locale:

- aumentare la "complessità" delle mosse, ossia la "dimensione" dell'intorno, per permettere di "vedere" una frazione maggiore dello spazio delle soluzioni ad ogni passo,
- permettere peggioramenti della funzione obiettivo purchè ci sia un "miglioramento a lungo termine".

Nei prossimi paragrafi discuteremo entrambe queste strategie, non prima però di aver introdotto altri esempi di intorno.

5.2.1.2 Ordinamento di lavori su macchine con minimizzazione del tempo di completamento

Si consideri il problema di ordinamento di lavori su macchine con minimizzazione del tempo di completamento (MMMS) definito al paragrafo 1.2.9.1. È facile definire per questo problema una funzione intorno basata su "mosse" analoghe a quelle viste per il problema dello zaino:

- spostamento: selezionare un lavoro i attualmente assegnato ad una certa macchina h ed assegnarlo ad una macchina $k \neq h$;
- scambio: selezionare un lavoro i attualmente assegnato ad una certa macchina h ed un lavoro j attualmente assegnato ad una certa macchina $k \neq h$, assegnare i alla macchina k ed assegnare j alla macchina h.

Come nel caso del problema dello zaino, una mossa di scambio può essere vista come la concatenazione di due mosse di spostamento, e l'intorno che usa entrambe i tipi di mosse domina l'intorno che usa solamente quelle di spostamento. Si consideri ad esempio il caso di un problema con due macchine e cinque lavori di lunghezza 5, 4, 4, 3 e 2, e la soluzione (determinata da LPT) $\{5,3,2\}$, $\{4,4\}$ con makespan 10. Qualsiasi mossa di spostamento peggiora il makespan della soluzione e quindi non viene accettata; per contro, scambiare il lavoro di lunghezza 5 con uno di quelli di lunghezza 4 produce una soluzione migliore (in particolare ottima).

Esercizio 5.21 *Si determini la complessità di valutare la funzione σ data da (5.7) per l'intorno proposto; si indichi poi come cambia il risultato qualora si effettuino solamente mosse di spostamento.*

Questo esempio mostra come per costruiree funzioni intorno per un certo problema di *OC* si possa trarre ispirazione da funzioni intorno già viste per altri problemi di *OC*. Ogni problema ha però le sue caratteristiche, che devono essere prese in debita considerazione. Il caso di (MMMS) e quello di (KP) sono diversi in molti aspetti: ad esempio, mentre in (KP) le mosse devono tenere in conto del vincolo di capacità dello zaino, in (MMMS) non ci sono di fatto vincoli, e quindi tutte le mosse producono soluzioni ammissibili. Inoltre, in (KP) tutte le mosse producono una variazione della funzione obiettivo (a meno che non siano scambiati due oggetti dello stesso costo, il che come abbiamo visto può essere evitato), mentre in (MMMS) tutte le mosse che non coinvolgono almeno una delle macchine che determinano il makespan della soluzione non possono migliorare il valore della funzione obiettivo. In effetti, è facile vedere che sono ottimi locali per l'algoritmo di ricerca locale basato sull'intorno così definito tutte le soluzioni in cui ci siano almeno due macchine che determinano il makespan: nessuna mossa di scambio può ridurre il tempo di completamento di due macchine (tipicamente ridurrà il tempo di completamento di una delle macchine coinvolte ed aumenterà quello dell'altra), e quindi migliorare il makespan della soluzione. Per questo è conveniente, nell'algoritmo di ricerca locale, accettare di spostarsi anche su soluzioni con lo stesso makespan di quella corrente, perché siano migliori per qualche altra caratteristica. Ad esempio, è possibile accettare mosse che diminuiscano il *numero* di macchine che determinano il makespan; ciò corrisponde a minimizzare una funzione obiettivo del tipo $v + \varepsilon ns$, dove v è il valore del makespan, ns è il numero di macchine che hanno tempo di completamento esattamente pari a v e ε è un valore opportunamente piccolo ($\varepsilon < 1/m$). Questa funzione obiettivo discrimina tra soluzioni con lo stesso makespan, e quindi le soluzioni in cui ci siano almeno due macchine che determinano il makespan non sono più (necessariamente) ottimi locali per l'intorno.

Esercizio 5.22 *In generale, non è necessariamente vero che funzioni intorno per due problemi di* OC *"simili" possano essere simili; si proponga una funzione intorno per il problema dell'ordinamento di lavori su macchine con minimizzazione del numero delle macchine.*

5.2.1.3 Il problema del commesso viaggiatore

Si consideri il problema del commesso viaggiatore (TSP) definito al paragrafo 1.2.2.3. In questo caso, i costituenti elementari di una soluzione sono lati, e quindi si può pensare, in analogia con gli esempi precedenti, ad operazioni di tipo "scambio" che coinvolgano gli archi del ciclo. In questo caso non è chiaramente possibile operare su un solo lato del grafo, in quanto cancellando un

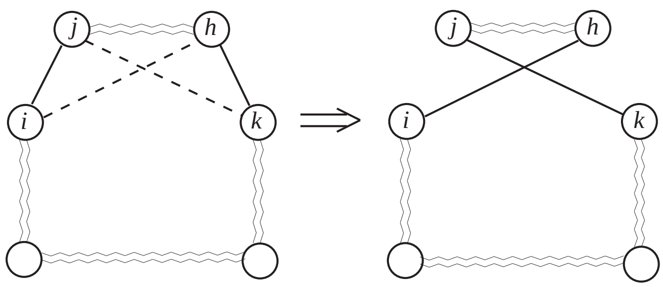


Figura 5.8: Un "2-scambio" per il (TSP)

qualsiasi lato dal ciclo si ottiene un cammino (Hamiltoniano), e l'unico modo possibile per trasformare il cammino in un ciclo Hamiltoniano è quello di aggiungere nuovamente il lato appena cancellato. È quindi necessario operare su almeno due lati contemporaneamente: ad esempio, data una soluzione ammissibile x, l'intorno $I(x)$ basato sui "2-scambi" contiene tutti i cicli Hamiltoniani che si possono ottenere da x selezionando due lati $\{i, j\}$ ed $\{h, k\}$ non consecutivi del ciclo e sostituendoli con gli archi $\{i, h\}$ e $\{j, k\}$, se esistono. Questa operazione è esemplificata in figura 5.8.

Esempio 5.9: Ricerca locale per il (TSP)
Si consideri l'istanza in figura 5.9(a) ed il suo ciclo Hamiltoniano $C = \{ 1 , 2 , 3 , 4 , 5 \}$ (lati in neretto), di costo $c = 14$. Per generare l'intorno basato sul 2-scambio di C è sufficiente enumerare tutte le coppie di lati $\{i, j\}$ ed $\{h, k\}$ non consecutive su C; in questo caso si ottengono i tre cicli Hamiltoniani mostrati in figura 5.9(b), (c) e (d), ciascuno col corrispondente costo e con indicati i lati di C sostituiti. Quindi, C non è un ottimo locale rispetto all'intorno basato sui 2-scambi (si noti che, per definizione, C appartiene all'intorno): il ciclo $C' = \{ 1 , 2 , 3 , 5 , 4 \}$ ha costo minore.

Il costo di valutare la funzione σ corrispondente a questo intorno è $O(n^2)$, poichè tante sono le coppie di lati non consecutivi del ciclo, e la valutazione del costo del ciclo ottenuto da un 2-scambio può essere fatta in $O(1)$ (conoscendo il costo del ciclo originario). Gli algoritmi di ricerca locale per il TSP attualmente ritenuti piè efficienti fanno uso di operazioni di 2-scambio o simili.

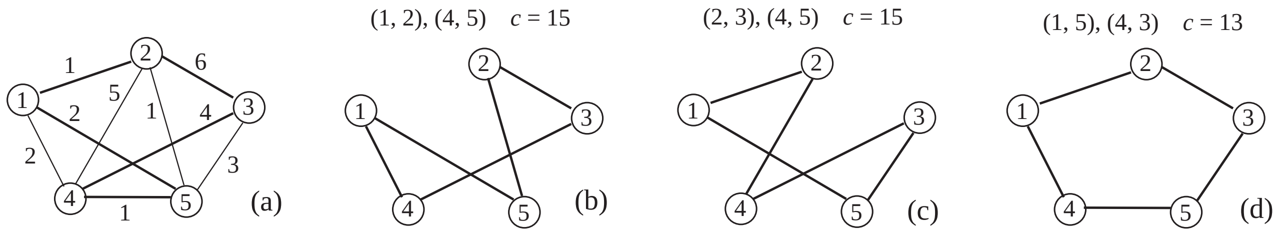


Figura 5.9: Ricerca locale basata sui "2-scambi" per il (TSP)

5.2.1.4 Il problema del Constrained MST

Si consideri il problema del Constrained MST definito nell'Esempio 4.1. In questo caso le soluzioni ammissibili del problema sono alberi, e si può pensare ad operazioni analoghe a quelle viste in precedenza che coinvolgano lati dell'albero. È però necessario porre attenzione al vincolo sul massimo peso dei sottoalberi della radice. Dato che ogni soluzione ammissibile x del problema è un albero di copertura radicato del grafo, possiamo rappresentarla mediante un vettore $p[\cdot]$ di predecessori. Indicheremo inoltre con $T(i)$ il sottoalbero di radice i della soluzione (se i è una foglia, $T(i)$ contiene il solo nodo i) e con $Q(i) = \sum_{h \in T(i)} q_h$ il suo peso: siccome la soluzione è ammissibile si avrà sicuramente $Q(i) \leq Q$ per ogni $i \neq r$, dove r è la radice dell'albero,

Un primo esempio di mossa per il (CMST) è la cosiddetta "Cut & Paste", che consiste nel selezionare un nodo $i \neq r$ ed un nodo $j \notin T(i)$ tale che $j \neq p[i]$ e porre $p[i] = j$. Ciò corrisponde ad eliminare dall'albero il lato $\{p[i], i\}$ e sostituirlo con il lato $\{j, i\}$, ossia a "potare" il sottoalbero $T(i)$ dalla sua posizione ed "innestarlo sotto il nodo j", come mostrato in figura 5.10(a). Naturalmente, la mossa può essere compiuta solamente se $Q(i) + Q(h) \leq Q$, dove h è il figlio della radice tale che $j \in T(h)$.

È possibile implementare il calcolo della funzione σ in modo tale che il controllo di tutte le possibili mosse abbia costo $O(n)$; è necessario mantenere per ogni nodo i il valore di $Q(i)$ ed esaminare i nodi $j \neq i$ in ordine opportuno (ad esempio visitando l'albero a partire dalla radice) per disporre in $O(1)$ di $Q(h)$ e dell'informazione che j non appartiene a $T(i)$. Poichè la variazione del valore della funzione obiettivo corrispondente ad una mossa di questo tipo è $c_{ji} - c_{p[i]i}$, e quindi può essere calcolato in $O(1)$, il costo di valutare la funzione σ data da (5.7) per l'intorno che usa mosse di "Cut & Paste" è $O(n^2)$.

Esercizio 5.23 *Si descriva formalmente, in pseudo-codice, una procedura che calcola la funzione σ data da (5.7) in $O(n^2)$ per la funzione intorno che usa mosse di "Cut & Paste".*

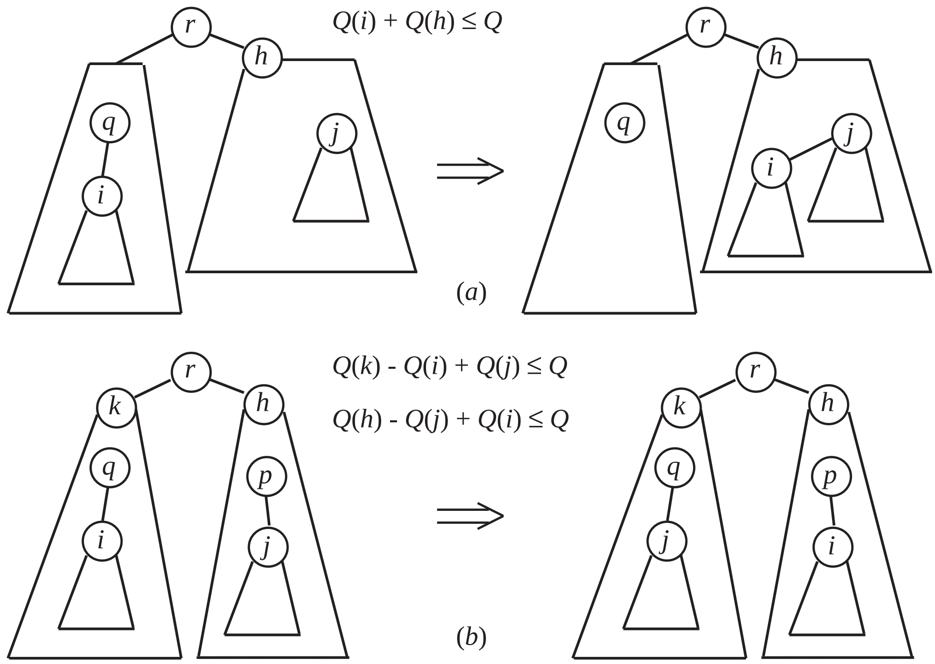


Figura 5.10: Un'operazione di "Cut & Paste" per il (CMST)

Analogamente ai casi visti in precedenza, si possono realizzare mosse "più complesse" combinando opportunamente mosse "semplici", in questo caso quelle di "Cut & Paste". Ad esempio, è possibile pensare a mosse di scambio in cui si selezionano due nodi i e j tali che $i \notin T(j)$ e $j \notin T(i)$ e si scambiano i predecessori di i e j, purché ovviamente ciò non violi il vincolo sul massimo peso dei sottoalberi della radice: l'operazione è esemplificata in figura 5.10(b). Siccome durante la mossa di scambio si tiene conto, nel valutare l'ammissibilità della soluzione ottenuta, del fatto che un insieme di nodi viene rimosso da ciascun sottoalbero della radice coinvolto mentre un altro sottoinsieme viene aggiunto, è chiaro che possono esistere ottimi locali per l'intorno basato sulle sole mosse di "Cut & Paste" che non sono ottimi locali per l'intorno che usa anche mosse di scambio, in quanto le due mosse di "Cut & Paste" corrispondenti ad una mossa di scambio potrebbero non essere eseguibili sequenzialmente per via del vincolo sul massimo peso dei sottoalberi della radice.

Esercizio 5.24 *Si discuta la complessità di calcolare la funzione σ data da (5.7) per la funzione intorno che usa sia mosse di "Cut & Paste" che mosse di scambio.*

È interessante notare che dopo ogni mossa di ricerca locale è possibile effettuare un'operazione di "ottimizzazione globale" sui sottoalberi coinvolti. Infatti, il problema (CMST) sarebbe risolubile efficientemente se si conoscessero i sottoinsiemi di nodi che formano ciascun sottoalbero della radice in una soluzione ottima: basterebbe applicare una procedura per l'(MST) a ciascun grafo parziale individuato da un sottoinsieme di nodi per determinare gli archi del sottoalbero ottimo. In generale, i sottoalberi ottenuti dopo una mossa di scambio possono non essere alberi di copertura di costo minimo per l'insieme di nodi che coprono, anche nel caso in cui lo fossero i sottoalberi di partenza, per cui applicando una procedura per determinare l'albero di copertura di costo minimo ai sottoalberi coinvolti dalla mossa si potrebbe ulteriormente migliorare la soluzione ottenuta. Questo è vero anche per il sottoalbero "destinazione" di una mossa di "Cut & Paste", ma chiaramente non per quello di "partenza".

Esercizio 5.25 *Si discuta come modificare le procedure note per l'(MST) in modo tale che risolvano in modo efficiente i problemi di (MST) generati da una mossa di "Cut & Paste" o di scambio.*

5.2.1.5 Dislocazione ottima di impianti

Si consideri il problema della dislocazione ottima di impianti definito al paragrafo 1.2.13.1: sono date n possibili località dove aprire impianti, ciascuno con un dato costo di installazione d_i ed un numero massimo u_i di clienti che può servire, ed m clienti, ciascuno dei quali può essere servito dall'impianto i al costo c_{ij}, $j = 1, \ldots, m$. Si vuole decidere in quali delle n località aprire gli impianti e, per ciascuno di essi, l'insieme dei clienti assegnati, in modo tale che ogni cliente sia assegnato ad uno ed un solo impianto e che il costo complessivo (di installazione e gestione) sia minimo. Il problema ha due tipi diversi di variabili binarie: le y_i, $i = 1, \ldots, n$, per rappresentare la scelta relativa agli impianti da aprire, e le x_{ij}, $i = 1, \ldots, n, j = 1, \ldots, m$, per assegnare i clienti agli impianti. Per analogia con gli esempi precedenti si possono definire mosse che riguardano poche variabili: ad esempio, sono mosse possibili l'apertura o chiusura di un singolo impianto (fissare ad 1 o a 0 una data variabile y_i), lo scambio tra un impianto chiuso ed uno aperto, l'assegnazione di un cliente ad un diverso impianto e così via. In questo caso è bene però tener presente che esiste una chiara "gerarchia" tra le variabili: le y rappresentano le decisioni "principali", mentre le x rappresentano decisioni "secondarie". Ciò deriva dal fatto che, una volta fissato il valore delle y, il valore ottimo delle x può essere facilmente ottenuto risolvendo un problema di flusso di costo minimo.

Esercizio 5.26 *Si descriva come istanza di un problema di (MCF) il problema di determinare, data una soluzione y, l'assegnamento ottimo dei clienti agli impianti aperti; in particolare, si discuta cosa accade nel caso in cui non siano aperti abbastanza impianti per servire tutti i clienti.*

Quindi, in linea di principio le mosse potrebbero essere effettuate solamente sulle variabiil y: una volta effettuata una mossa, applicando un algoritmo per (MCF) è possibile determinare il miglior valore possibile per le variabili x data la nuova scelta di y.
Un possibile svantaggio di operare in questo modo consiste nel dover risolvere un problema di (MCF) per valutare qualsiasi mossa: anche limitandosi a mosse semplici, quali l'apertura e chiusura di un singolo impianto, ciò richiede la soluzione di n problemi di (MCF) per valutare la funzione σ ad ogni passo dell'algoritmo di ricerca locale. Sono possibili diverse strategie per cercare di ridurre il costo computazionale del calcolo di σ. Ad esempio, si potrebbero specializzare gli algoritmi noti per (MCF) alla particolare forma delle istanze da risolvere in questo caso, analogamente a quanto fatto nel paragrafo 3.5 per i problemi di accoppiamento. Alternativamente, si potrebbero determinare inizialmente soluzioni approssimate del (MCF), ossia costruire (velocemente) un assegnamento non necessariamente ottimo dei clienti agli impianti aperti. Ad esempio, nel caso di chiusura di un impianto i si potrebbero distribuire gli utenti j precedentemente assegnati a quell'impianto con un semplice criterio greedy basato sui costi, esaminandoli in ordine arbitrario ed assegnando ciascuno all'impianto aperto h con costo c_{hj} minimo tra quelli che hanno ancora capacità residua; analogamente, nel caso di apertura di un impianto i si potrebbero assegnare ad i utenti j, attualmente assegnati ad un diverso impianto h, per cui risulti $c_{ij} < c_{hj}$, partendo da quelli in cui la $c_{hj} - c_{ij}$ è maggiore. Se la nuova soluzione (x, y) così ottenuta è migliore della soluzione corrente allora sicuramente lo sarà anche quella ottenuta risolvendo il (MCF); si può quindi selezionare la mossa da compiere sulla base di questa valutazione approssimata, risolvendo il (MCF) una sola volta per iterazione. Se invece così facendo non si determina nessuna souzione migliore di quella corrente si possono riesaminare le mosse risolvendo esattamente il (MCF), in quanto la soluzione approssimata può aver portato a non effettuare mosse che invece risultano convenienti. Infine, invece che valutazioni superiori del costo del(MCF) si potrebbero utilizzare valutazioni inferiori, sfruttando informazione duale analogamente a quanto mostrato nel paragrafo 2.3.3. È in generale difficile valutare a priori quale di queste strategie possa risultare più conveniente; sono necessari a tal proposito esperimenti su un adeguato insieme di istanze "campione" che confrontino la qualità delle soluzioni ottenute e lo sforzo computazionale richiesto dall'algoritmo di ricerca locale con le diverse varianti di funzione intorno.
I due esempi precedenti mostrano come lo studio della struttura dei problemi sia fondamentale per costruire funzioni intorno opportune. In particolare, in molti casi emergono "gerarchie" tra gruppi di variabili del problema, in quanto è possibile utilizzare algoritmi efficienti per determinare il valore ottimo di un gruppo di variabili una volta fissato il valore delle altre. Questo è un caso in cui si rivela

l'importanza della conoscenza degli algoritmi per i problemi "facili" per costruire algoritmi efficienti per problemi "difficili".

5.2.2 Intorni di grande dimensione

Il costo computazionale di un algoritmo di ricerca locale dipende fondamentalmente dalla complessità della trasformazione σ, che a sua volta dipende tipicamente dalla dimensione e dalla struttura della funzione intorno I. La dimensione dell'intorno $I(x)$ è anche collegata alla qualità degli ottimi locali che si determinano: intorni "più grandi" tipicamente forniscono ottimi locali di migliore qualità. È necessario quindi operare un attento bilanciamento tra la complessità della trasformazione σ e la qualità delle soluzioni determinate; come esempio estremo, la trasformazione σ data da (5.7) basata sull'intorno $I(x) = X$ fornisce sicuramente la soluzione ottima, ma richiede di risolvere all'ottimo il problema originario, e quindi è di fatto inutile. In molti casi risulta comunque conveniente utilizzare intorni "di grande dimensione", corrispondenti a "mosse complesse". Ciò è spesso legato alla possibilità di implementare la funzione σ in modo efficiente, ad esempio utilizzando algoritmi per problemi di ottimizzazione per calcolare (5.7) senza dover esaminare esplicitamente tutte le soluzioni nell'intorno $I(x)$. Discutiamo nel seguito alcuni esempi di intorni di questo tipo.

5.2.2.1 Il problema del commesso viaggiatore

Una famiglia di funzioni intorno di grande dimensione che generalizza quella discussa nel §5.2.1.3 è quella basata sui "k-scambi": in un k-scambio si selezionano e rimuovono k archi del ciclo e si costruiscono tutti i possibili cicli Hamiltoniani che è possibile ottenere combinando i sottocammini rimanenti. Non è difficile vedere che la complessità di calcolare σ quando I è definita mediante operazioni di "k-scambio" cresce grosso modo come n^k: quindi, per valori di k superiori a 2 o 3 determinare la migliore delle soluzioni in $I(x)$ può diventare molto oneroso dal punto di vista computazionale. D'altra parte, gli ottimi locali che si trovano usando operazioni di "3-scambio" sono usualmente migliori di quelli che si trovano usando operazioni di "2-scambio", e così via.
Non essendo noti modi per calcolare σ che non richiedano di esaminare sostanzialmente tutte le soluzioni nell'intorno, si possono utilizzare schemi di ricerca locale in cui la dimensione dell'intorno (il numero k di scambi) varia dinamicamente. Si può ad esempio utilizzare preferibilmente 2-scambi, passando ai 3-scambi solamente se il punto corrente si rivela essere un ottimo locale per l'intorno basato sui 2-scambi; analogamente, se il punto corrente si rivela essere un ottimo locale anche per l'intorno basato sui 3-scambi si può passare ai 4-scambi e così via, fino ad un qualche valore limite di k fissato a priori. In questo caso risulta spesso conveniente ritornare immediatamente ai 2-scambi non appena si sia effettuata una mossa di ricerca locale con un k-scambio per $k > 2$, in quanto il k-scambio potrebbe aver generato una soluzione che non è un ottimo locale per il 2-scambio. Inoltre, quando k è "grande" può essere conveniente evitare di calcolare esattamente (5.7), terminando la computazione non appena si determina una qualunque soluzione nell'intorno che migliori "abbastanza" il valore della funzione obiettivo.

5.2.2.2 Il problema del Constrained MST

Come abbiamo già rilevato, il problema del (CMST) consiste di fatto nel determinare la partizione ottima dell'insieme dei nodi in un numero opportuno (non noto a priori) di sottoinsiemi disgiunti, in ciascuno dei quali la somma dei pesi dei nodi non sia superiore alla soglia Q; nota la partizione ottima, il problema è facilmente risolubile. Quindi, il (CMST) è un esempio di problema in cui le soluzioni ammissibili sono identificabili con una partizione di un insieme base in un certo numero di sottoinsiemi disgiunti con opportune proprietà: altri problemi con questa struttura sono ad esempio i problemi di ordinamento di lavori su macchine (ogni macchina corrisponde ad un insieme).

Un modo abbastanza generale per costruire intorni di grande dimensione per problemi con questo tipo di struttura è quello dello "scambio ciclico". Per il caso del (CMST), ciò corrisponde a selezionare un insieme di nodi i_1, i_2, …, i_k, appartenenti ciascuno a un diverso sottoalbero della radice, ed

effettuare una mossa di scambio simultanea che coinvolge tutti i nodi: porre $p[i_1] = p[i_2]$, $p[i_2] = p[i_3]$, …, $p[i_{k-1}] = p[i_k]$ e $p[i_k] = p[i_1]$. Naturalmente, la mossa è possibile solo se la soluzione così ottenuta non viola il vincolo di capacità sui sottoalberi. È chiaro come queste mosse generalizzino quella di scambio vista al paragrafo 5.2.1.4, e quindi che permettano potenzialmente di non rimanere "intrappolati" in soluzioni che siano ottimi locali per queste ultime.
Implementare un'algoritmo di ricerca locale basato su mosse di scambio ciclico richiede però di determinare in modo efficiente, ad ogni iterazione, un insieme di nodi i_1, i_2, …, i_k a cui corrisponda una mossa di scambio ciclico ammissibile e che migliori il valore della funzione obiettivo. Può risultare utile esprimere tale problema come un problema di decisione su grafi (su un grafo diverso da quello del problema originale). Sia data infatti una soluzione ammissibile di (CMST), e si consideri il grafo orientato che ha per nodi quelli del grafo originale, tranne la radice, e nel quale esiste l'arco (i, j), di costo $c_{p[j]i} - c_{p[i]i}$, se e solo se i e j appartengono a sottoalberi della radice diversi ed è possibile "innestare" i sotto il predecessore di j senza violare il vincolo di capacità, purchè nel contempo j sia "potato" (ed innestato sotto un diverso sottoalbero, non importa quale). È facile verificare che un ciclo orientato di costo negativo su questo grafo, in cui tutti i nodi appartengano a sottoalberi diversi, rappresenta una mossa di scambio ciclico che migliora il valore della funzione obiettivo, e, viceversa, qualsiasi mossa di scambio ciclico corrisponde ad un ciclo di questo tipo. Il problema di determinare una mossa di scambio ciclico può quindi essere quindi formulato nel seguente modo: dato un grafo orientato $G = (N, A)$ con costi associati agli archi, in cui l'insieme dei nodi N sia partizionato in un certo numero di sottoinsiemi disgiunti $N_1, \ldots, N_k$, determinare se esiste nel grafo un ciclo orientato di costo negativo che contenga al più un nodo per ciascuno dei sottoinsiemi N_i. Risolvere questo problema di decisione corrisponde a determinare se esiste oppure no una mossa di scambio ciclico che determina una soluzione migliore di quella corrente; la funzione σ data da (5.7) richiederebbe invece di determinare il ciclo di costo minimo tra tutti quelli aventi le caratteristiche desiderate. Come abbiamo visto nel paragrafo 3.2.5, il problema di determinare se esiste un ciclo orientato di costo negativo in un grafo ha complessità polinomiale; invece, determinare il ciclo orientato di costo minimo è $\mathcal{NP}$-arduo. Il problema in esame richiede però non di determinare un qualsiasi ciclo di costo negativo, ma un ciclo di costo negativo che contenga al più un nodo per ciascuno dei sottoinsiemi N_i; come spesso accade, pur essendo una variante apparentemente minore di un problema polinomiale, questo problema è $\mathcal{NP}$-completo. È però possibile costruire euristiche per questo problema che cerchino di determinare un ciclo di costo negativo con le caratteristiche richieste, pur non garantendo di determinarlo anche nel caso in cui esista; se il ciclo viene trovato è possibile effettuare la mossa di scambio ciclico corrispondente, altrimenti l'algoritmo di ricerca locale si ferma dichiarando—forse erroneamente—che la soluzione corrente è un ottimo locale per l'intorno basato sullo scambio ciclico. Utilizzando euristiche opportune per la determinazione del ciclo di costo negativo si possono ottenere algoritmi di ricerca locale in grado di determinare efficientemente, in pratica, soluzioni di buona qualità per il problema.

Esercizio 5.27 *Si discuta come modificare l'algoritmo basato su* SPT.L *per la determinazione dei cicli orientati di costo negativo in un grafo in modo da ottenere un algoritmo euristico per determinare mosse di scambio ciclico.*

Questo esempio mostra come tecniche di ottimizzazione, esatte o euristiche, possano essere utilizzate per implementare in modo efficiente la funzione σ anche per intorni di grandi dimensioni. Si noti come la conoscenza di un algoritmo esatto per risolvere un problema di decisione "facile", in questo caso il problema di determinare l'esistenza di un ciclo orientato di costo negativo in un grafo, suggerisca algoritmi euristici per un problema simile, che possono a loro volta essere utilizzati per implementare algoritmi euristici per un problema molto diverso. Questo mostra il livello di complessità e sofisticazione che è a volte necessario affrontare per realizzare algoritmi efficienti, esatti o euristici, per problemi di *OC*, e quindi l'importanza della conoscenza degli algoritmi per problemi di ottimizzazione "facili".

Esercizio 5.28 *Si proponga una funzione intorno basata su mosse di scambio ciclico per il problema di ordinamento di lavori su macchine con minimizzazione del tempo di completamento (MMMS), discutendo l'implementazione della procedura σ.*

5.2.3 Metaeuristiche

Anche utilizzando intorni di grande dimensione, quali quelli visti nel paragrafo precedente, tutti gli algoritmi di ricerca locale per problemi di OC si arrestano per aver determinato un ottimo locale rispetto alla funzione intorno utilizzata, o, più in generale, perchè non sono in grado di trovare "mosse" che generino soluzioni migliori della soluzione corrente. Qualora ci sia motivo per credere che la soluzione corrente non sia un ottimo globale, si pone quindi il problema di cercare di determinare un diverso, e possibilmente migliore, ottimo locale. In questo paragrafo discuteremo brevemente alcune possibili strategie che permettono di fare questo, cercando di illustrare alcune delle loro principali caratteristiche e potenziali limitazioni. Queste strategie sono chiamate *metaeuristiche* perchè non sono algoritmi specifici per un dato problema, ma metodi generali che possono essere applicati per tentare di migliorarne le prestazioni di molti diversi algoritmi di ricerca locale.

5.2.3.1 Multistart

In generale, la qualità dell'ottimo locale determinato da un algoritmo di ricerca locale dipende da due fattori: l'intorno utilizzato e la soluzione ammissibile iniziale da cui la ricerca parte. Nei paragrafi precedenti ci siamo soffermati principalmente sul primo fattore, assumendo che la soluzione iniziale venisse determinata con qualche euristica, ad esempio di tipo greedy. Come abbiamo visto in molti degli esempi discussi nel paragrafo 5.1.1, spesso è possibile definire più di un algoritmo greedy per lo stesso problema, ed in molti casi gli algoritmi sono anche molto simili, differendo solamente per l'ordine in cui sono compiute alcune scelte, per cui non è irragionevole pensare di avere a disposizione più di un algoritmo in grado di produrre soluzioni iniziali. In questo caso, le soluzioni prodotte saranno normalmente diverse; inoltre, non è detto che l'ottimo locale determinato eseguendo l'algoritmo di ricerca locale a partire dalla migliore delle soluzioni così ottenute sia necessariamente il migliore degli ottimi locali ottenibili eseguendo l'algoritmo di ricerca locale a partire da ciascuna delle soluzioni separatamente. Tutto ciò suggerisce un'ovvia estensione dell'algoritmo di ricerca locale: generare più soluzioni iniziali, eseguire l'algoritmo di ricerca locale a partire da ciascuna di esse, quindi selezionare la migliore delle soluzioni così ottenute.
Questo procedimento è particolarmente attraente quando sia possibile, tipicamente attraverso l'uso di tecniche randomizzate, generare facilmente un insieme arbitrariamente numeroso di soluzioni iniziali potenzialmente diverse. Si pensi ad esempio agli algoritmi greedy di tipo *list scheduling* per (MMMS) presentati al paragrafo 5.1.1.4: è facile costruire un'*euristica randomizzata* per il problema semplicemente costruendo un ordinamento pseudo-casuale dei lavori e poi assegnando i lavori alle macchine (selezionando ogni volta la macchina "più scarica") in quell'ordine. In questo modo è chiaramente possibile produrre un gran numero di soluzioni iniziali diverse, a partire da ciascuna delle quali si può poi eseguire un algoritmo di ricerca locale. Mediante l'uso di numeri pseudo-casuali è di solito facile trasformare euristiche deterministiche per un problema di ottimizzazione in euristiche randomizzate.

Esercizio 5.29 *Si discuta se e come sia possibile trasformare ciascuno degli algoritmi euristici visti nel paragrafo 5.1.1 in euristiche randomizzate.*

La combinazione di un algoritmo di ricerca locale e di un'euristica randomizzata viene denominato *metodo multistart*; quando, come spesso accade, l'euristica randomizzata è di tipo greedy, si parla di *GRASP* (greedy randomized adaptive search procedure). Una volta sviluppata una qualsiasi euristica randomizzata per determinare la soluzione iniziale ed un qualsiasi algoritmo di ricerca locale per un dato problema, è praticamente immediato combinare le due per costruire un metodo multistart.
Sotto opportune ipotesi tecniche è possibile dimostrare che ripetendo un numero sufficientemente alto di volte la procedura, ossia generando un numero sufficientemente alto di volte soluzioni iniziali, si è praticamente certi di determinare una soluzione ottima del problema. Si noti che, al limite, non è neppure necessaria la fase di ricerca locale: un'euristica randomizzata ripetuta più volte è comunque un modo per generare un insieme di soluzioni ammissibili, e se l'insieme è abbastanza ampio è molto probabile che contenga anche la soluzione ottima. Questo risultato non deve illudere sulla reale possibilità di giungere ad una (quasi) certezza di ottimalità utilizzando un metodo di questo tipo: in

generale, il numero di ripetizioni necessarie può essere enorme, in modo tale che risulterebbe comunque più conveniente enumerare tutte le soluzioni del problema.
Le euristiche multistart forniscono quindi un modo semplice, ma non particolarmente efficiente, per cercare di migliorare la qualità delle soluzioni determinate da un algoritmo di ricerca locale. Anche se non esistono regole che valgano per qualsiasi problema, è possibile enunciare alcune linee guida che si sono dimostrate valide per molti problemi diversi. In generale, il meccanismo della ripartenza da un punto casuale non fornisce un sostituto efficiente di una ricerca locale ben congegnata: se si confrontano le prestazioni di un metodo multistart che esegue molte ricerche locali con intorni "piccoli" e di uno che esegue poche ricerche locali con intorni "grandi", è di solito il secondo a fornire soluzioni migliori a parità di tempo totale utilizzato. Questo comportamento è spiegato dal fatto che una ripartenza da un punto pseudo-casuale "cancella la storia" dell'algoritmo: l'evoluzione successiva è completamente indipendente da tutto quello che è accaduto prima della ripartenza. In altri termini, il metodo multistart non è in grado di sfruttare in alcun modo l'informazione generata durante la ricerche locali precedenti per "guidare" la ricerca locale corrente. Ad esempio, è noto che per molti problemi le soluzioni di buona qualità sono normalmente abbastanza "vicine" le une alle altre (in termini di numero di mosse degli intorni utilizzati); quindi, l'aver determinato una "buona" soluzione fornisce una qualche forma di informazione che il metodo multistart non tiene in alcun modo in considerazione. Al contrario, la ricerca locale ha un qualche tipo di informazione sulla "storia" dell'algoritmo, in particolare data dalla soluzione corrente. La capacità di sfruttare l'informazione contenuta nelle soluzioni precedentemente generate è in effetti uno degli elementi che contraddistinguono le metaeuristiche più efficienti, quali quelle discusse nel seguito.

5.2.3.2 Simulated annealing

L'idea di base del *simulated annealing* è quella di modificare l'algoritmo di ricerca locale sostituendo i criteri deterministici di selezione del nuovo punto nell'intorno corrente e di accettazione della mossa con un criteri randomizzati. Uno schema generale di algoritmo di tipo simulated annealing (per un problema di minimo) è il seguente:

```
Procedure Simulated-Annealing( F , x )  {
  x = Ammissibile(F); x* = x; c = InitTemp();
  while( c ≥ c̄ ) do {
    for i = 1 to k(c) do
      ⟨ seleziona x' ∈ I(x) in modo pseudocasuale ⟩;
      if( c(x') < c(x*) )then x* = x';
      if( c(x') < c(x) )  then x = x';                           /* downhill */
                         else { r = random(0,1);
                                if( r < e^(-(c(x')-c(x))/c) ) then x = x';   /* uphill */
                              }
    ⟨ decrementa c ⟩;
    }
  x = x*;
  }
```

Procedura 5.5: Algoritmo *Simulated annealing*

L'algoritmo inizia determinando una soluzione iniziale ed un opportuno valore del parametro "temperatura" c, che controlla l'accettazione di mosse che peggiorano il valore della funzione obiettivo. L'algoritmo esegue quindi un certo numero di "fasi", all'interno della quali la "temperatura" è costante. Ad ogni iterazione si seleziona in modo pseudo-casuale un punto nell'intorno del punto corrente: se questo punto ha un miglior valore della funzione obiettivo (si tratta cioè di una normale mossa di ricerca locale, o "downhill") viene certamente accettato come il punto corrente. In caso contrario il punto può essere ugualmente accettato come punto corrente (mossa "uphill"), basandosi sull'estrazione di un numero pseudo-casuale: la probabilità che il punto sia accettato dipende dal peggioramento della funzione obiettivo e dal valore di c. Una fase in cui la temperatura è costante termina quando il sistema raggiunge uno "stato stazionario"[1], ossia quando si suppone che il valore della funzione obiettivo della soluzione corrente x sia sufficentemente "vicino"—usando c come "unità di misura"—a quello della soluzione ottima; in pratica si fissa il numero di iterazioni di ogni fase secondo regole opportune. Alla fine di ogni fase la temperatura viene diminuita, rendendo meno probabile l'accettazione di mosse che peggiorino sensibilmente il valore della funzione obiettivo, e quindi concentrando la ricerca su soluzioni più "vicine" a quella corrente. L'algoritmo termina quando la temperatura è sufficientemente bassa (il sistema è "congelato"), riportando la migliore delle soluzioni trovate: nell'ultima fase di fatto l'algoritmo si comporta in modo simile ad un normale algoritmo di ricerca locale.

L'idea alla base dell'algoritmo è quella di permettere consistenti peggioramenti del valore della funzione obiettivo nelle fasi iniziali dell'esecuzione, in modo da evitare di rimanere intrappolati in ottimi locali molto "lontani" dall'ottimo globale. Dopo un numero sufficiente di iterazioni l'algoritmo dovrebbe avre raggiunto una parte dello spazio delle soluzioni "vicina" all'ottimo globale: a quel punto la temperatura viene diminuita per raffinare la ricerca. L'algoritmo prende spunto da un metodo usato in pratica per produrre cristalli con elevato grado di regolarità: si scalda il materiale in questione per rompere i legami chimici non desiderati (di più alta energia), si lascia ad una certa temperatura per un tempo sufficientemente lungo affinchè si creino con alta probabilità i legami chimici desiderati (di più bassa energia), quindi si raffredda il materiale per rendere più stabili i legami chimci di bassa energia, ripetendo il tutto finchè, sperabilmente, il materiale contiene solo legami di più bassa energia.

Come per il caso del multistart, anche il simulated annealing possiede, sotto opportune ipotesi tecniche, interessanti proprietà teoriche. In particolare, si può dimostrare che esiste una costante C tale che se la temperatura decresce non più rapidamente di $C/\log(k)$, dove k è il numero di iterazioni compiute, allora l'algoritmo determina una soluzione ottima con probabilità pari a uno. Come nel caso del multistart, però, queste proprietà teoriche non hanno grande utilità in pratica: la *regola di raffreddamento* sopra indicata corrisponde ad una diminuzione molto lenta della temperatura, e quindi ad un numero di iterazioni talmente grande da rendere più conveniente l'enumerazione di tutte le soluzioni ammissibili. In pratica si usano quindi *regole di raffreddamento esponenziali* in cui la temperatura viene moltiplicata per un fattore $\theta < 1$ dopo un numero fissato di iterazioni, e la temperatura iniziale c viene scelta come la maggior differenza di costo possibile tra due soluzioni x ed x' tali che $x' \in I(x)$ (in modo tale da rendere possibile, almeno inizialmente, qualsiasi mossa). In questo modo non si ha alcuna certezza, neanche in senso stocastico, di determinare una soluzione ottima, ma si possono ottenere euristiche di buona efficienza in pratica.
Gli algoritmi di tipo simulated annealing hanno avuto un buon successo in alcuni campi di applicazioni dell'Ottimizzazione Combinatoria, ad esempio nella progettazione di circuiti VLSI. La ragione di questo successo risiede nel fatto che sono relativamente semplici da implementare, poco dipendenti dalla struttura del problema e quindi facilmente adattabili a problemi diversi ed abbastanza "robusti", nel senso che, se lasciati in esecuzione sufficientemente a lungo, solitamente producono soluzioni di buona qualità in quasi tutti i problemi in cui sono applicati. Un vantaggio in questo senso è il fatto che contengano relativamente pochi parametri, ossia la temperatura iniziale, il valore di θ e la durata di ciascuna fase. Ogniqualvolta il comportamento di un algoritmo dipende da parametri che possono essere scelti in modo arbitrario (sia pur seguendo certe regole) si pone infatti il problema di determinare un valore dei parametri che produce soluzioni di buona qualità quanto più rapidamente possibile per le istanze che si intende risolvere. Per fare questo è normalmente necessario eseguire l'algoritmo molte volte su un nutrito insieme di istanze di test con combinazioni diverse dei parametri, in modo da ottenere informazione sull'impatto dei diversi parametri sull'efficacia ed efficienza dell'algoritmo. Questo processo può richiedere molto tempo, specialmente se l'algoritmo è poco "robusto", ossia il suo comportamento varia considerevolmente per piccole variazioni dei parametri o, con gli stessi parametri, per istanze simili. Il simulated annealing risulta piuttosto "robusto" da questo punto di vista, anche per la disponibilità di linee guida generali per l'impostazione dei parametri che si sono rivelate valide in molti diversi contesti. In linea generale, si può affermare che normalmente la tecnica del

[1] La successione dei passi compiuti dall'algoritmo quando c è costante è un processo stocastico noto come *Catena di Markov*, del quale sono note molte importanti proprietà statistiche; in particolare, dopo un numero opportuno di passi il sistema tende a raggiungere uno stato stazionario indipendentemente dal punto di partenza.

simulated annealing permette di migliorare la qualità delle soluzioni fornite da un algoritmo di ricerca locale con una ragionevole efficienza complessiva. Il metodo risulta molto spesso più efficiente dei metodi multistart basati sullo stesso intorno, poiché mantiene una maggiore quantità di informazione sulla storia dell'algoritmo – non solo la soluzione corrente, ma anche la temperatura – che in qualche modo "guida" la ricerca di una soluzione ottima. Questo tipo di tecnica risulta particolarmente adatta a problemi molto complessi, di cui sia difficile sfruttare la struttura combinatoria, e nel caso in cui l'efficacia del metodo (la qualità della soluzione determinata) sia più importante della sua efficienza (la rapidità con cui la soluzione è determinata). In casi diversi altri tipi di tecniche, come quelle discusse nel seguito, possono risultare preferibili; questo è giustificato dal fatto che l'informazione sulla storia dell'algoritmo mantenuta da un algoritmo di tipo simulated annealling è molto "aggregata", e quindi non particolarmente efficiente nel guidare la ricerca. Inoltre, il simulated annealing si basa fondamentalmente su decisioni di tipo pseudo-casuale, il che tipicamente produce un comportamento "medio" affidabile ma difficilmente è più efficiente di decisioni deterministiche guidate da criteri opportuni. Infine, la generalità del metodo, ossia il fatto che l'algoritmo sia largamente indipendente dal problema, implica che il metodo non sfrutta appieno la struttura del problema; la capacità di sfruttare quanto più possibile tale struttura è uno dei fattori fondamentali che caratterizzano gli algoritmi più efficienti.

5.2.3.3 Ricerca Taboo

Un problema fondamentale che deve essere superato quando si intende modificare un algoritmo di ricerca locale consiste nel fatto che accettare mosse che peggiorano il valore della funzione obiettivo ("uphill") pone ad immediato rischio di ritornare sulla precedente soluzione corrente, e quindi di entrare in un ciclo in cui si ripetono sempre le stesse soluzioni. Infatti, normalmente accade che $x' \in I(x)$ implica che $x \in I(x')$; di conseguenza, qualora si abbia $c(x') > c(x)$ (in un problema di minimo) e si accetti ugualmente di porre x' come punto corrente, all'iterazione successiva si avrà la soluzione x nell'intorno del punto corrente con un valore minore della funzione obiettivo, e quindi sarà possibile (e probabile) effettuare una mossa "downhill" ritornando su x. Le tecniche di tipo simulated annealling non sono immuni da questo problema, ma sono basate su decisioni stocastiche, per cui la ripetizione del punto corrente non causa necessariamente l'entrata in un ciclo. Infatti, nella situazione precedente il nuovo punto corrente viene scelto in modo casuale nell'intorno di x', per cui anche se $c(x)$ fosse l'unico punto in $I(x')$ con valore migliore della funzione obiettivo non sarebbe necessariamente scelto; anche qualora ciò accadesse, comunque, il nuovo punto x'' scelto nell'intorno di x sarebbe con alta probabilità diverso da x', e quindi la probabilità di ritornare un numero molto alto di volte sullo stesso punto corrente x è estremamente bassa.

Qualora si vogliano implementare algoritmi deterministici questo tipo di considerazione non è più valida, ed è quindi necessario porre in effetto tecniche che impediscano, o comunque rendano altamente improbabile, l'entrata in un ciclo. La più diffusa da queste tecniche è quella delle *mosse Taboo*, che caratterizza un'ampia famiglia di algoritmi detti di *ricerca Taboo* (TS, da Taboo Search). Questi algoritmi sono, in prima approssimazione, normali algoritmi di ricerca locale, in cui cioè si compiono mosse "downhill" fin quando ciò sia possibile; quando però il punto corrente è un ottimo locale per l'intorno utilizzato, e quindi un normale algoritmo di ricerca locale terminerebbe, un algoritmo TS seleziona comunque una soluzione $x' \in I(x)$ secondo criteri opportuni e compie una mossa "uphill". Per evitare i cicli, l'algoritmo mantiene una *lista Taboo* che contiene una descrizione delle mosse "uphill"; ad ogni iterazione, nel cercare una soluzione $x' \in I(x)$ con $c(x') < c(x)$, l'algoritmo controlla la lista Taboo, e scarta tutte le soluzioni che sono generate da una mossa Taboo. Siccome l'algoritmo permette mosse "uphill", non è garantito che la soluzione corrente al momento in cui l'algoritmo termina sia la migliore determinata; come nel caso del simulated annealling, si mantiene quindi, oltre alla soluzione corrente, la miglior soluzione x^* tra tutte quelle determinate. In prima approssimazione si potrebbe pensare che la lista Taboo contenga tutte le soluzioni precedentemente trovate con valore della funzione obiettivo migliore di quello della soluzione corrente; questo chiaramente eviterebbe i cicli, ma normalmente risulta eccessivamente costoso, sia in termini di memoria richiesta che in termini del costo di verificare che una data soluzione appartenga alla lista. In generale si preferisce quindi mantenere nella lista Taboo una descrizione, anche parziale delle mosse (che definiscono la funzione intorno) che hanno generato la soluzione corrispondente. Questa descrizione dipende quindi dalla particolare funzione intorno utilizzata, e deve essere "compatibile" con la funzione σ, ossia non deve rendere "eccessivamente più costosa" la determinazione della nuova soluzione.

Si consideri ad esempio il problema del (TSP) e la semplice funzione intorno basata su mosse di 2-scambio descritta nel §5.2.1.3, ossia sullo scambio della coppia di lati $\{\, \{i,j\}\, ,\ \{h,k\}\, \}$ con $\{\, \{i,k\}\, ,\ \{h,j\}\, \}$. Una possibile implementazione della lista Taboo per questo caso potrebbe contenere semplicemente la coppia $\{\, \{i,k\}\, ,\ \{h,j\}\, \}$ che caratterizza la mossa: una mossa di 2-scambio sarebbe dichiarata Taboo—e quindi scartata—se coinvolgesse esattamente quei due lati. Si noti che questa scelta può impedire di generare molte soluzioni oltre a quella che originariamente ha causato l'inserzione di una data coppia di lati nella lista Taboo. Infatti, se x ed x' sono rispettivamente il punto corrente ed il nuovo punto corrispondenti ad una mossa "uphill", ed $x'' \neq x$, $x'' \in I(x')$ è il punto accettato all'iterazione successiva, allora effettuare la mossa Taboo (qualora possibile) non genererebbe x, e quindi non causerebbe necessariamente l'entrata in un ciclo. Quindi, questa implementazione della lista Taboo è in qualche modo "eccessiva" rispetto al compito di evitare i cicli; ciononostante può risultare opportuna perchè può essere facilmente ed efficientemente integrata nel calcolo della funzione σ.

In effetti, per semplificare il controllo della lista Taboo non è infrequente che si implementi una lista Taboo che contiene una *descrizione parziale* delle mosse utilizzate. Si consideri ad esempio il problema (MMMS) e la funzione intorno basata su mosse di "spostamento" e "scambio" descritta nel §5.2.1.2. Una possibile implementazione della lista Taboo per questo caso potrebbe contenere semplicemente coppie (d, i) dove d è una durata di lavoro ed i è una macchina: è considerata una mossa Taboo assegnare alla macchina i un lavoro di durata d. Questa è ovviamente una descrizione parziale sia di una mossa di spostamento che di scambio: se si pone (d, i) nella lista Taboo ogniqualvolta si elimina un lavoro di durata d dalla lista di quelli assegnati alla macchina i, si evitano sicuramente i cicli. Analogamente a quello che abbiamo visto per il (TSP), questa implementazione impedisce di generare molte altre soluzioni oltre a quelle già visistate dall'algoritmo.

Inserire mosse nella lista Taboo diminuisce il numero di soluzioni considerate ad ogni passo come possibili nuove soluzioni correnti: questo può seriamente limitare la capacità dell'algoritmo di scoprire soluzioni migliori. Per questo sono necessari dei meccanismi che limitino l'effetto della lista Taboo. Il primo meccanismo, usato in tutti gli algoritmi di TS, consiste semplicemente nel limitare la massima lunghezza della lista Taboo ad una costante fissata: quando la lista ha raggiunto tale lunghezza ed una nuova mossa deve essere resa Taboo, la più "vecchia" delle mosse nella lista viene nuovamente resa ammissibile. In generale non è facile determinare valori della massima lunghezza della lista delle mosse Taboo che garantiscano che l'algoritmo non entri in un ciclo; in pratica, però, gli algoritmi di tipo TS non entrano in ciclo se la lista Taboo contiene anche solo poche decine di mosse. Ciò dipende dal fatto che il numero di possibili soluzioni che possono essere visitate con k mosse di ricerca locale aumenta esponenzialmente in k, e quindi è molto poco probabile che l'algoritmo visiti esattamente, tra tutte quelle possibili, una delle soluzioni già generate. Limitare la lunghezza della lista Taboo ha anche, chiaramente, un effetto positivo sulle prestazioni dell'algoritmo: normalmente, il costo di verificare che una mossa non sia Taboo cresce con la lunghezza della lista, e quindi ciò potrebbe risultare molto costoso qualora la liste divenisse molto lunga.

Un secondo metodo per limitare gli effetti negativi della lista Taboo è quello di inserire un cosiddetto *criterio di aspirazione*, ossia un insieme di condizioni logiche tale per cui se la soluzione $x' \in I(x)$ generata da una mossa soddisfa almeno una di queste condizioni, allora x' può essere considerata tra i possibili candidati a divenire la prossima soluzione corrente anche se la mossa appartiene alla lista Taboo. Ovviamente, il criterio di aspirazione deve essere scelto in modo tale da garantire—o rendere molto probabile—che l'algoritmo non entri in un ciclo: un possibile criterio di aspirazione è ad esempio $c(x') < c(x^*)$, dove x^* è la migliore delle soluzioni generate, dato che in questo caso x' non può sicuramente essere stata visitata in precedenza. Sono comunque possibili anche altri criteri di aspirazione: come il nome suggerisce, e l'esempio precedente mostra, i criteri di aspirazione corrispondono spesso a condizioni che richiedono che x' sia sotto qualche aspetto una "buona" soluzione. Ad esempio, nel problema (MMMS) si potrebbe permettere di assegnare un lavoro di durata d ad una data macchina i anche se (d, i) fa parte della lista Taboo purchè il tempo di completamento della macchina i dopo l'assegnamento sia minore del tempo di completamento che la macchina aveva al momento in cui (d, i) è stata inserita nella lista delle mosse Taboo (questo richiederebbe chiaramente di memorizzare anche tale durata nella lista). Chiaramente questo evita i cicli in quanto la soluzione ottenuta dalla mossa è "localmente", ossia limitatamente alla macchina i, migliore di quella che c'era al momento in cui la mossa Taboo è stata effettuata.

Oltre a queste componenti base, gli algoritmi TS spesso contengono altre strategie che risultano utili per rendere il processo di ricerca più efficiente. Ad esempio, per quei problemi in cui le "buone" soluzioni sono spesso "vicine" sono spesso utilizzate tecniche di *intensificazione*, che concentrano temporaneamente la ricerca "vicino" alla soluzione corrente. Questo può essere ottenuto ad esempio restringendo la dimensione dell'intorno, oppure modificando la funzione obiettivo con un termine che penalizza leggermente le soluzoni dell'intorno più "lontane" da quella corrente. L'intensificazione viene solitamente effettuata quando si determina una nuova miglior soluzione x^*, e mantenuta per un numero limitato di iterazioni, nella speranza che vicino alla migliore soluzione determinata fino a quel momento ci siano altre soluzioni ancora migliori. Una strategia in un certo senso opposta è quella della *diversificazione*, che cerca di indirizzare la ricerca verso aree dello spazio delle soluzioni diverse da quelle esplorate fino a quel momento. Modi possibili per implementare strategie di diversificazione sono ad esempio penalizzare leggermente le soluzioni dell'intorno "vicine" alla soluzione corrente x, allargare la dimensione dell'intorno, effettuare in una sola iterazione combinazioni di molte mosse, ad esempio selezionandole in modo randomizzate, per "saltare" in una zona dello spazio delle soluzioni "lontana" da x, o persino effettuare una ripartenza pseudo-casuale come nel metodo multistart. Normalmente, vengono effettuate mosse di diversificazione quando per "molte" iterazioni successive la migliore soluzione trovata x^* non cambia, suggerendo che l'area dello spazio delle soluzioni in cui si trova la soluzione corrente sia già stata sostanzialmente esplorata, e che sia quindi necessario visitarne una diversa. Nell'implementare tecniche di intensificazione e diversificazione risulta spesso utile sfruttare l'informazione contenuta nella sequenza di soluzioni generate, ad esempio associando a ciascun costituente elementare delle soluzioni un qualche valore numerico che ne rappresenti in qualche modo "l'importanza". Si pensi ad esempio di applicare un qualche algoritmo di ricerca locale per risolvere il problema del commesso viaggiatore, e di mantenere per ogni arco un contatore che indichi di quante soluzioni correnti l'arco ha fatto parte fino a quel momento: archi con un valore alto del contatore sono in qualche senso "più rilevanti" di archi con un valore basso del contatore. In una procedura di diversificazione potrebbe essere ragionevole concentrarsi su archi con valore alto del contatore, in modo da esplorare soluzioni "diverse" da quelle visitate fino a quel momento. Alternativamente si potrebbe contare per ogni arco di quante delle migliori k soluzioni generate ha fatto parte, dove k è un valore fissato; più in generale, si potrebbe mantenere per ogni arco un valore numerico calcolato a partire dal costo delle soluzioni di cui ha fatto parte, in modo tale che il valore sia "alto" per archi cha fanno "spesso" parte di soluzioni "buone". Con un'opportuna scelta della funzione e dei parametri, questo tipo di informazione sulle soluzioni generate dall'algoritmo può rivelarsi utile per guidare la ricerca.

Infine, una strategia che talvolta si rivela utile è quella di permettere che la soluzione corrente x diventi inammissibile per un numero limitato di iterazioni, se questo serve a migliorare sensibilmente il valore della soluzione obiettivo. Questo viene fatto per i problemi in cui sia particolarmente difficile costruire soluzioni ammissibili: le "poche" soluzioni ammissibili del problema possono quindi essere "lontane", ed è necessario permettere mosse in cui si perde l'ammissibilità, seguite da una successione di mosse in cui si cerca di riguadagnarla.

È chiaro come, rispetto ad un algoritmo di tipo simulated annealling, un algoritmo TS sia più complesso da realizzare. L'implementazione della lista Taboo, e spesso anche dei criteri di aspirazione, è fortemente dipendente dalla scelta della funzione intorno, e quindi l'algoritmo deve essere progettato tenendo in conto di tutti questi aspetti contemporaneamente. Normalmente, anche le strategie di intensificazione e diversificazione dipendono in qualche modo dall'intorno utilizzato, per cui anche questi aspetti devono essere considerati insieme agli altri. Per tutte le strategie sopra elencate è normalmente necessario determinare sperimentalmente diversi parametri, quali la lunghezza della lista Taboo, il numero di iterazioni per l'intensificazione, il numero di iterazioni per la diversificazione, e così via: il numero complessivo di parametri può essere alto, il che può rendere difficoltosa la determinazione di "buoni" valori per tutti i parametri. La grande flessibilità data dai molti parametri, ed il fatto che ogni aspetto di un algoritmo TS debba essere adattato allo specifico problema di OC risolto, sono però anche le caratteristiche che possono rendere questi algoritmi molto efficienti in pratica: un algoritmo TS ben congegnato ed in cui i parametri siano scelti in modo opportuno si rivela spesso più efficiente, ad esempio, di un algoritmo di tipo simulated annealling per lo stesso problema che usi la stessa funzione intorno, nel senso che produce soluzioni di migliore qualità a parità di tempo. Qualora l'efficienza sia almeno altrettanto importante della qualità delle soluzioni fornite, quindi, può essere ragionevole investire le risorse necessarie a realizzare un algoritmo TS.

5.2.3.4 Algoritmi genetici

Uno dei principali fattori che determina l'efficienza di un algoritmo di ricerca locale è la capacità di sfruttare l'informazione contenuta nella sequenza di soluzioni generate per guidare la ricerca di soluzioni migliori. Come abbiamo visto, le diverse strategie algoritmiche ottengono questo risultato in gradi diversi, e con diversi accorgimenti. Esiste una classe di algoritmi euristici, che in linea di principio *non* sono algoritmi di ricerca locale, che portano questo concetto alle estreme conseguenze: si tratta dei cosiddetti *algoritmi genetici*, che cercano di riprodurre alcuni dei meccanismi che si ritiene stiano alla base dell'evoluzione delle forme di vita.

La caratteristica peculiare di questo tipo di algoritmi è di mantenere non una singola soluzione corrente ma una *popolazione* di soluzioni. L'algoritmo procede per fasi, corrispondenti a "generazioni" nella popolazione. In ogni fase vengono ripetute un numero opportuno di volte le seguenti operazioni:

- all'interno della popolazione vengono selezionate in modo pseudo-casuale due (o più) soluzioni "genitrici";

- a partire da tali soluzioni vengono generate in modo pseudo-casuale un certo numero di soluzioni "discendenti", ottenute "mescolando" le caratteristiche di tutte le soluzioni genitrici;

- a ciascuna soluzione così generata vengono applicate alcune "mutazioni casuali" che cercano di introdurre nella popolazione caratteristiche altrimenti non presenti.

Alla fine di ogni fase si procede alla *selezione*: a ciascuna delle soluzioni, sia quelle presenti nella popolazione all'inizio della fase che quelle generate durante la fase, viene associato un *valore di adattamento* (fitness), ad esempio il valore della funzione obiettivo, e vengono mantenute nella popolazione (sopravvivono alla generazione successiva) solo le soluzioni con miglor fitness, mantenendo costante la dimensione della popolazione; alternativamente, le soluzioni sopravvissute sono selezionate in modo pseudo-casuale con probabilità dipendente dal fitness. Sono state poi proposte molte varianti a questo schema base, ad esempio suddividendo la popolazione in sotto-popolazioni più piccole con scambi limitati (in modo da simulare le barriere geografiche presenti in natura) o imponendo la "morte" delle soluzioni dopo un numero massimo di generazioni indipendentemente dal loro valore di fitness.
Tutte le operazioni di un algoritmo genetico devono essere specializzate per il problema da risolvere, anche se per alcune esistono implementazioni abbastanza standard. Ad esempio, la selezione delle soluzioni genitrici viene normalmente effettuata semplicemente estraendo in modo casuale dalla popolazione; la probabililità di essere estratta può essere uniforme oppure dipendere dal valore di fitness. Analogamente, la selezione alla fine di ogni fase è normalmente effettuata, in modo deterministico o pseudo-casuale, semplicemente privilegiando le soluzioni con miglior fitness. Per quanto riguarda la definizione del valore di fitness, spesso si usa semplicemente il valore della funzione obiettivo. In alcuni casi può risultare opportuno modificare la funzione obiettivo per premiare soluzioni che, a parità di funzione obiettivo, appaiano più desiderabili: ad esempio, per il problema (MMMS) si può assegnare un valore di fitness leggermente più alto, a parità di makespan, a quelle soluzioni in cui le macchine

che non contribuiscono a determinare il makespan sono “bilanciate”, ossia hanno tutte tempo di completamento simile. Per i problemi in cui sia particolarmente difficile costruire soluzioni ammissibili è possibile ammettere nella popolazione soluzioni non ammissibili, penalizzando opportunamente la non ammissibilità nel corrispondente valore di fitness. La generazione di soluzioni e le mutazioni casuali, invece, sono strettamente dipendenti dal problema. Questo non è stato inizialmente ben compreso: dato che qualsiasi soluzione ammissibile rappresentata in un computer può essere vista come una stringa di bits, si è sostenuto che le operazioni di generazione e mutazione potessero essere implementate in modo generico, ad esempio rimpiazzando nel “patrimonio genetico” di un genitore una o più sottostringhe di bits con quelle prelevate dalle corrispondenti locazioni nel “patrimonio genetico” dell'altro genitore, e cambiando il valore di un piccolo numero dei bits scelti in modo pseudo-casuale. Questo risulta in effetti possibile per problemi con “pochi vincoli”, in cui sia molto facile costruire una soluzione ammissibile. Si pensi ad esempio a (MMMS): una soluzione ammissibile è data da un qualisasi assegnamento dei lavori alle macchine, ossia da qualsiasi vettore $s[\cdot]$ di n componenti in cui ogni elemento $s[i]$ appartenga all'insieme $\{1, \ldots, m\}$, col significato che $s[i] = h$ se e solo se il lavoro i è assegnato alla macchina h. Dati due vettori $s_1[\cdot]$ ed $s_2[\cdot]$ di questo tipo, è molto facile produrre un nuovo vettore $s_3[\cdot]$ possibilmente diverso da entrambe: ad esempio si possono selezionare in modo pseudo-casuale due numeri $1 \leq i \leq j \leq n$ e porre $s_3[h] = s_1[h]$ per $i \leq h \leq j$ e $s_3[h] = s_2[h]$ per tutti gli altri indici h. Le mutazioni possono poi essere ottenute semplicemente cambiando in modo pseudo-casuale il valore di un piccolo numero di elementi di $s_3[\cdot]$, scelti in modo pseudo-casuale. Nella maggioranza dei casi questo però non risulta effettivamente possibile: per la maggior parte dei problemi, operando in modo analogo a quanto appena visto si ottengono quasi sempre soluzioni non ammissibili. È quindi necessario sviluppare euristiche specifiche che costruiscano soluzioni ammissibili cercando di replicare, per quanto possibile, le caratteristiche di entrambe i genitori.

Si consideri ad esempio il problema del (TSP): dati due cicli Hamiltoniani diversi, si devono produrre uno o più cicli che “combinino” le caratteristiche dei due genitori. Un possibile modo di procedere è quello di considerare fissati, nei cicli “figli”, tutti gli archi che sono comuni ad entrambe i genitori, e poi cercare di completare questa soluzione parziale fino a formare un ciclo Hamiltoniano mediante un'euristica costruttiva randomizzata analoga alla “Nearest Neighbour” descritta nel paragrafo 5.1.1.3. Definito un nodo “corrente” iniziale i, l'euristica prosegue sicuramente sul nodo j se il lato $\{i, j\}$ appartiene ad entrambe i genitori. Se invece il “successore” di i nei due cicli Hamiltoniani è diverso, l'euristica seleziona uno dei due successori che non sia ancora stato visitato secondo un qualche criterio opportuno; ad esempio, quello a cui corrisponde il lato di costo minore, oppure in modo pseudo casuale con probabilità uniforme, oppure con probabilità dipendente dal costo del lato, oppure ancora con probabilità dipendente dal valore della funzione obiettivo del “genitore” a cui il lato corrispondente appartiene (favorendo le scelte del “genitore migliore”). Qualora entrambe i successori siano stati già visitati l'euristica selezionerà un diverso nodo secondo altri criteri (ad esempio quello “più vicino); se tutti i nodi raggiungibili da i sono già visitati (e non si è formato un ciclo) l'euristica fallirà, “abortendo” la generazione della nuova soluzione.

Una procedura in qualche modo analoga si può utilizzare nel caso di (CMST). Le due soluzioni “genitrici” sono alberi di copertura diversi dello stesso grafo: esse avranno quindi un certo numero di sottoalberi comuni, che risulterebbero quindi sicuramente ammissibili se fossero utilizzate come sottoalberi della radice (al limite i sottoalberi potranno essere formati da un solo nodo). In ciascun sottoalbero, il predecessore di tutti i nodi tranne la radice è identico nelle due soluzioni, mentre la radice ha due predecessori diversi, uno per ciascuna soluzione. Si può quindi realizzare una procedura euristica che tenti di combinare questi sottoalberi per formare una soluzione ammissibile: selezionata la radice i di un sottoalbero, l'euristica seleziona uno dei due predecessori secondo un qualche criterio opportuno, ad esempio quello a cui corrisponde l'arco di costo minore, oppure in modo pseudo-casuale con probabilità uniforme, oppure con probabilità dipendente dal costo dell'arco, oppure ancora con probabilità dipendente dal valore della funzione obiettivo del “genitore” a cui l'arco corrispondente appartiene. Naturalmente un predecessore può essere selezionato solamente se ciò non crea un sottoalbero di peso superiore alla soglia massima fissata: se ciò accade per entrambe i predecessori corrispondenti alle soluzioni genitrici l'euristica tenterà di selezionare un diverso predecessore – al li-

mite, la radice dell'albero – mediante criteri opportuni. Se nessun predecessore può essere selezionato senza violare i vincoli l'euristica fallirà, "abortendo" la generazione della nuova soluzione.
Si noti che euristiche simili possono essere utilizzate anche per problemi in cui sia "facile" produrre soluzioni ammissibili. Nel caso di (MMMS), ad esempio, si potrebbe utilizzare un approccio simile alle euristiche list scheduling viste al paragrafo 5.1.1.4: ordinati i lavori secondo un qualche criterio, si potrebbe assegnare ciascun lavoro h ad una delle due macchine $s_1[h]$ ed $s_2[h]$, scegliendo quella che ha il tempo di completamento minimo oppure con un qualche criterio pseudo-casuale analogo a quelli visti precedentemente (i lavori che siano assegnati alla stessa macchina in entrambe le soluzioni genitrici sarebbero quindi sicuramente assegnati a quella macchina in tutte le soluzioni discendenti).

Le mutazioni casuali sono normalmente più facili da realizzare, soprattutto qualora si disponga già di un algoritmo di ricerca locale per il problema. Infatti, un modo tipico per realizzare mutazioni è quello di effettuare un piccolo numero di mosse scelte in modo pseudo-casuale, a partire dalla soluzione generata. Ad esempio, per il (TSP) si potrebbero effettuare un piccolo numero di 2-scambi tra coppie $\{i, j\}$ e $\{h, k\}$ di lati del ciclo selezionate in modo pseudo-casuale; analogamente, per (CMST) si potrebbero effettuare un piccolo numero di mosse di "Cut & Paste" tra coppie di nodi i e j scelte in modo pseudo-casuale. Si noti che nel caso del (CMST) le mosse di "Cut & Paste" potrebbero produrre una soluzione non ammissibile: in questo caso si ha la scelta tra non effettuare la mossa oppore "abortire" la soluzione così generata e passare a generarne un'altra.

Infine, qualsiasi algoritmo genetico richiede una prima fase in cui viene generata la popolazione iniziale. Ciò può essere ottenuto in vari modi: ad esempio, si può utilizzare un'euristica randomizzata analogamente a quanto visto per il metodo multistart. Alternativamente si può utilizzare un algoritmo di ricerca locale, eventualmente con multistart, tenendo traccia di un opportuno sottoinsieme delle soluzioni visitate, che costituiranno a terminazione la popolazione iniziale per l'algoritmo genetico.

Anche gli algoritmi genetici, come quelli di ricerca Taboo, dipendono da un numero di parametri il cui valore può non essere facile da fissare. Un parametro molto importante è ad esempio la cardinalità della popolazione: popolazioni troppo piccole non permettono di diversificare a sufficienza la ricerca, che può rimanere "intrappolata" in una zona dello spazio delle soluzioni, mentre una popolazione troppo numerosa può rendere poco efficiente l'algoritmo. Altri parametri importanti riguardano il numero di nuove soluzioni costruite ad ogni generazione, i parametri per la scelta dei genitori e la selezione e così via.

Gli algoritmi genetici "puri", ossia che seguono lo schema precedente, possono rivelarsi efficienti ed efficaci; spesso, però algoritmi basati sulla ricerca locale risultano maggiormente efficienti. In effetti, gli algoritmi genetici di maggior successo sono normalmente quelli che integrano entrambe le tecniche: ad ogni soluzione generata, dopo la fase di mutazione, viene applicata una procedura di ricerca locale per tentare di migliorarne il valore di fitness. Un algoritmo di questo tipo può anche essere considerato un metodo "multistart con memoria", in cui cioè le soluzioni di partenza del metodo multistart non sono selezionate in modo completamente pseudo-casuale, ma si cerca di utilizzare l'informazione precedentemente generata. La combinazione degli algoritmi genetici e degli algoritmi di ricerca locale può assumere molte forme diverse. I casi estremi corrispondono all'algoritmo genetico "puro", in cui cioè non si effettua nessuna ricerca locale, ed all'algoritmo di ricerca locale "puro", in cui cioè la popolazione è formata da un solo individuo. I metodi ibridi si differenziano fondamentalmente per la quantità di risorse che vengono spese nell'uno e nell'altro tipo di operazione: un algoritmo in cui la ricerca locale sia molto semplice, ed eventualmente eseguita comunque per al più un numero fissato di mosse, avrà un comportamento più simile a quello di un algoritmo genetico "puro", mentre un algoritmo in cui la ricerca locale sia effettuata in modo estensivo – ad esempio con tecniche Taboo e con intorni di grande dimensione – avrà un comportamento più simile a quello di un algoritmo di ricerca locale "puro" con multistart. Esiste inoltre la possibilità di eseguire sequenzialmente un algoritmo genetico ed uno di ricerca locale, eventualmente più volte: l'algoritmo di ricerca locale fornisce la popolazione iniziale per quello genetico e questi fornisce una nuova soluzione iniziale per la ricerca locale una volta che questa si sia fermata in un ottimo locale. Dato che esistono moltissimi modi per combinare i due approcci, è molto difficile fornire linee guida generali; ciò è particolarmente vero in quando in un approccio ibrido è necessario determinare il valore di molti parametri (per entrambe gli algoritmi e per la loro combinazione), rendendo la fase di messa a punto dell'algoritmo potenzialmente lunga e complessa. Solo l'esperienza può quindi fornire indicazioni sulla migliore strategia da utilizzare per un determinato problema di ottimizzazione. Si può però affermare che un'opportuna combinazione delle tecniche descritte in questo capitolo può permettere di determinare efficientemente soluzioni ammissibili di buona qualità per moltissimi problemi di ottimizzazione, per quanto al costo di un consistente investimento nello sviluppo e nella messa a punto di approcci potenzialmente assai complessi.

Riferimenti Bibliografici

E. Aarts and J.K. Lenstra (eds.) "**Local Search in Combinatorial Optimization**", *J. Wiley & Sons*, 1997.

F. Maffioli "**Elementi di Programmazione Matematica**", *Casa Editrice Ambrosiana*, 2000.

V. Varzirani "**Approximation Algorithms**", *Springer-Verlag*, 2001.

Capitolo 6

Tecniche di rilassamento

Come abbiamo osservato nei Capitoli 4 e 5, uno dei passi fondamentali nel processo di risoluzione di un problema di *OC* consiste nell'individuazione di valutazioni superiori (se il problema è di massimo, inferiori se è di minimo) del valore ottimo della funzione obiettivo. Ciò permette di certificare l'ottimalità, o almeno stimare la qualità, delle soluzioni ammissibili di cui si disponga, ad esempio avendole costruite attraverso euristiche come quelle descritte nel Capitolo 5. Se la qualità della soluzione disponibile non risulta soddisfacente, le valutazioni superiori sono uno degli elementi fondamentali per costruire algoritmi in grado di determinare, in linea di principio, soluzioni ottime o comunque con qualsiasi grado di accuratezza desiderato, come vedremo nel Capitolo 7. In effetti, come già discusso nel Paragrafo 4.3, si può ritenere che la difficoltà di risolvere un problema di *OC* consista proprio nel calcolare (o stimare in modo sufficientemente accurato) il valore ottimo della funzione obiettivo del problema.

La tecnica più comunemente utilizzata per produrre valutazioni superiori per un problema di *OC* è risolvere un *rilassamento* del problema. Il concetto di rilassamento è già stato introdotto nel Capitolo 1 e discusso nei Capitoli 4 e 5. In questo capitolo descriveremo alcune delle principali tecniche utilizzate per definire rilassamenti per problemi di *OC*, sfruttando spesso le loro formulazioni in termini di *PLI*, ed in particolare:

- il rilassamento continuo;
- il rilassamento per eliminazione di vincoli (o combinatorio);
- il rilassamento Lagrangiano;
- il rilassamento surrogato.

Esistono altri modi generali per costruire rilassamenti di problemi di *OC*, quali i rilassamenti semidefiniti positivi, e per molti problemi specifici possono essere sviluppati rilassamenti ad-hoc. Le tecniche che presenteremo sono comunque tra le più generali ed utilizzate, oltre ad essere relativamente facili da comprendere. Alla descrizione di ciascuna di queste tecniche, con le relative esemplificazioni per problemi specifici, premettiamo alcune semplici considerazioni generali sull'uso dei rilassamenti, nella pratica, per la soluzione di problemi di *OC*.

Nella costruzione di un rilassamento per un problema "difficile" di *OC* occorre sempre tenere conto del bilanciamento tra due obiettivi necessariamente contrastanti: l'*efficacia* del rilassamento, ossia la qualità della valutazione superiore da esso prodotta, e l'*efficienza* del rilassamento, ossia il tempo necessario a determinare tale valutazione per l'istanza in esame. L'efficacia di un rilassamento (P') di un dato problema (P) si misura solitamente attraverso il corrispondente *gap assoluto o relativo*, ossia (assumendo $z(P) > 0$)

$$G_A = z(P') - z(P) \qquad\qquad G_R = \frac{z(P') - z(P)}{z(P)} \ ;$$

un rilassamento sarà tanto più efficace quanto più il gap corrispondente è piccolo. È intuitivo come questi due obiettivi siano fondamentalmente in contrasto tra loro; un'esemplificazione estrema si può ottenere considerando i due rilassamenti "estremi" consistenti nel riportare sempre la valutazione "$+\infty$" e nel risolvere il problema all'ottimo, rispettivamente. È chiaro come il primo sia un rilassamento valido a costo nullo (costante), che tuttavia non produce di fatto nessuna informazione utile sull'istanza in oggetto; per contro, il secondo produce informazione sommamente utile, nel contesto della soluzione del problema, ma ad un costo che potenzialmente può non essere ragionevole pagare. In generale, per un dato problema di ottimizzazione si possono costruire molti rilassamenti diversi, che avranno quindi efficienza ed efficacia diverse; si pone quindi il problema di confrontarli tra loro. Dati due rilassamenti (P') e (P'') di uno stesso problema (P), non è in generale ragionevole esprimere preferenze tra (P') e (P'') tranne nel caso in cui—ad esempio—(P') *domini* (P''), ossia per ogni istanza del problema (o per un sottoinsieme di istanze rilevanti) (P') determini una valutazione non superiore (e quindi un gap non peggiore) di quella determinata da (P'') in un tempo non superiore. In tutti gli altri casi, ossia quando uno dei due rilassamenti fornisca gap migliori in un tempo superiore, almeno per insiemi rilevanti di istanze, la scelta tra (P') e (P'') dipenderà dalla specifica applicazione, oltreché possibilmente dall'istanza a cui il procedimento si applica. Ad esempio, se la valutazione viene computata una sola volta per stimare la qualità di una soluzione ottenuta con l'applicazione di un'euristica particolarmente costosa (ad esempio una metaeuristica), e se da tale stima dipendono decisioni potenzialmente critiche (ad esempio far proseguire ancora la metaeuristica, pagando un ulteriore sostanziale costo), allora presumibilmente l'efficacia sarà maggiormente rilevante dell'efficienza. Se invece la valutazione deve essere computata molte volte, come accade all'interno degli algoritmi enumerativi descritti nel Capitolo 7, l'efficienza può assumere un'importanza preponderante.
Nella pratica, comunque, si ritiene più spesso che l'efficacia di un rilassamento abbia importanza superiore all'efficienza; capita cioè sovente che rilassamenti più "costosi", ma in grado di fornire informazione molto accurata sul problema, risultino più utili di rilassamenti meno costosi ma meno accurati, anche nell'ambito di approcci enumerativi; ciò sarà discusso più in dettaglio nel Capitolo 7.

Per terminare queste brevi riflessioni generali aggiungiamo che il ruolo del rilassamento è quello di estrarre "informazione" dall'istanza del problema da risolvere, informazione che possa poi essere utilizzata all'interno del processo risolutivo. Questa informazione comprende sicuramente, e fondamentalmente, la valutazione superiore, ma può essere anche più ricca. Si consideri ad esempio il problema dello zaino e la valutazione superiore ottenuta risolvendone il rilassamento continuo come mostrato nel Paragrafo 5.1.2.1; oltre alla valutazione superiore tale rilassamento restituisce anche una soluzione primale frazionaria. Questa soluzione è un'ulteriore fonte di informazione: ad esempio, come abbiamo visto, mediante una semplice tecnica di arrotondamento da essa si deriva una soluzione ammissibile per il problema. L'utilità di tale informazione è particolarmente evidente nel caso in cui la soluzione del rilassamento continuo sia intera: il rilassamento ha di fatto risolto il problema. Ulteriori esempi dell'utilità di informazione prodotta da rilassamenti saranno discussi nel seguito.

6.1 Rilassamento continuo

Il rilassamento continuo è già stato introdotto e discusso nel paragrafo 4.2.1. Data una formulazione *PLI* di un problema di *OC*

$$(P) \qquad \max\{\ cx\ :\ Ax \leq b\ ,\ x \in \mathbb{Z}^n\ \}$$

il suo rilassamento continuo

$$(\bar{P}) \qquad \max\{\ cx\ :\ Ax \leq b\ \}$$

si ottiene semplicemente rimuovendo il *vincolo di integralità* sulle variabili. È quindi immediato costruire il rilassamento continuo di un problema di *OC* una volta che lo si sia formulato come *PLI*. In generale si avranno quindi molti possibili rilassamenti continui per uno stesso problema di *OC*, uno per ogni diversa formulazione *PLI* del problema (tra questi uno è "ottimo", come discusso nel paragrafo 4.2.1, ma la formulazione corrispondente non è in generale disponibile). L'ulteriore vantaggio del rilassamento continuo è che, essendo un problema di *PL*, abbiamo a disposizione algoritmi altamente efficienti per risolverlo.

6.1.1 Efficacia del rilassamento continuo

Il rilassamento continuo può quindi essere considerato efficiente; in alcuni casi è anche ragionevolmente efficace. Ad esempio, molte delle dimostrazioni di garanzia delle prestazioni di algoritmi greedy riportate nel paragrafo 5.1.2 possono essere "rilette" come *dimostrazioni di efficacia* di rilassamenti continui. Infatti, si consideri ad esempio l'algoritmo greedy per il Weighted Vertex Cover (cf. 5.1.2.4); la relazione

$$\textstyle\sum_{\{i,j\}\in E} y_{ij} \leq z(\underline{\text{WVC}}) \leq z(\text{WVC}) \leq 2\sum_{\{i,j\}\in E} y_{ij}$$

mostra come il gap relativo ottenuto dal rilassamento continuo della formulazione di *PLI* "naturale" del problema non sia mai superiore al 100%.

Esercizio 6.1 *Si ripeta lo stesso ragionamento per derivare valutazioni al caso pessimo per i gaps dei rilassamenti continui di (KP) e (MMMS); per quest'ultimo si mostri prima che il valore ottimo della funzione obiettivo del rilassamento continuo è esattamente pari a L.*

Nella pratica, una valutazione del valore ottimo della funzione obiettivo affetta da un gap del 100% difficilmente può essere considerata accurata; usualmente si considerano tali, a seconda del problema in esame, valutazioni con un gap di pochi punti percentuali. Essendo questa una stima dell'errore nel caso pessimo, naturalmente, è possibile che le valutazioni prodotte siano considerevolmente più precise; si ha cioè anche per i rilassamenti la distinzione tra comportamento nel caso pessimo e comportamento "medio" su una classe di istanze, come già discusso nel paragrafo 5.1.2 per le euristiche. Ad esempio, per il caso di (MMMS) il gap dal rilassamento continuo su istanze generate in modo pseudo-casuale è molto spesso una piccola frazione dell'1%, ossia molto migliore dal 33% che si potrebbe desumere dai risultati del paragrafo 5.1.2.2. Tuttavia, in molti casi la valutazione del valore ottimo prodotta dal rilassamento continuo è di qualità fortemente scadente, come mostrato dai due esempi seguenti.

6.1.1.1 Assegnamento di frequenze

Si consideri il problema dell'assegnamento di frequenze descritto nel paragrafo **??**, e si supponga che l'insieme di frequenze disponibili abbia cardinalità superiore ad uno. È facile dimostrare che, qualsiasi siano la dimensione e la densità del grafo, la valutazione inferiore sul valore ottimo della funzione obiettivo computata risolvendo il rilassamento continuo della formulazione *PLI* fornita del problema, ossia

$$(\underline{\text{GC}})\qquad \begin{array}{llll} \min & \sum_{f=1}^m y_f & & \\ & \sum_{f=1}^m x_{if} = 1 & i = 1,\ldots,n & \\ & x_{if} + x_{jf} \leq 1 & (i,j) \in A & f = 1,\ldots,m \\ & y_f \geq x_{if} \geq 0 & i = 1,\ldots,n & f = 1,\ldots,m \\ & 1 \geq y_f \geq 0 & & f = 1,\ldots,m \end{array}$$

è non superiore a 1. Per questo si consideri la soluzione frazionaria $y_f = 1/m$, $f = 1,\ldots,m$ e $x_{if} = 1/m$, $i = 1,\ldots,n$, $f = 1,\ldots,m$; in altre parole, in questa soluzione si assegna $1/m$ di ciascuna frequenza a ciascun nodo. Tale soluzione è ammissibile per $(\underline{\text{GC}})$ in quanto $x_{if} + x_{jf} = 2/m \leq 1$ e $\sum_{f=1}^m x_{if} = 1$; il corrispondente valore della funzione obiettivo è 1, e quindi $z(\underline{\text{GC}}) \leq 1$. Si noti che questo è valido anche per un grafo completo, che richiede un numero di frequenze esattamente pari al numero dei nodi, e di qualsiasi dimensione; quindi, la valutazione inferiore sul valore ottimo della funzione obiettivo fornita da $(\underline{\text{GC}})$ può essere affetta da un gap arbitrariamente grande. Il problema in questo caso è che, passando al rilassamento continuo, il vincolo numerico "$x_{if} + x_{jf} \leq 1$" perde completamente il suo originario significato logico "al massimo uno tra i due valori x_{if} e x_{jf} può essere diverso da zero".

6.1.1.2 Il problema (CMST)

Per poter presentare un rilassamento continuo di (CMST) dobbiamo in primo luogo introdurne una formulazione in termini di *PLI*. Per assicurare l'efficienza del rilassamento continuo non possiamo estendere la formulazione data nel Paragrafo 1.2.2.2, che ha un numero esponenziale di vincoli; considereremo quindi una diversa formulazione, che sfrutti variabili di flusso. Poichè il problema originario è definito su un grafo $G = (V, E)$ non orientato, definiamo un corrispondente grafo orientato $G' = (N, A)$ tale che $N = V$ ad A contenga i due archi (orentati) (i, j) e (j, i) per ciascun lato $\{i, j\} \in E$, assegnando ad entrambi costo pari a quello dell'arco originale; su G' definiamo un vettore di flusso x. Una formulazione *PLI* di (CMST) è

$$\min \textstyle\sum_{\{i,j\} \in E} c_{ij} y_{ij} \tag{6.1}$$

$$\textstyle\sum_{\{i,j\} \in E} y_{ij} \leq 1 \qquad j \in N \setminus \{r\} \tag{6.2}$$

$$\textstyle\sum_{(j,i) \in A} x_{ji} - \sum_{(i,j) \in A} x_{ij} = b_i \qquad i \in N \setminus \{r\} \tag{6.3}$$

$$x_{ij} + x_{ji} \leq Q y_{ij} \qquad \{i, j\} \in E \tag{6.4}$$

$$y_{ij} \in \{0, 1\} \;\;,\;\; x_{ij} \geq 0 \;\;,\;\; x_{ji} \geq 0 \qquad \{i, j\} \in E \tag{6.5}$$

In questa formulazione, ogni nodo $i \neq r$ richiede b_i unità di flusso; la radice è quindi l'unica sorgente del grafo, anche se il corrispondente vincolo di conservazione del flusso

$$\textstyle\sum_{(j,r) \in A} x_{jr} - \sum_{(r,j) \in A} x_{rj} = - \sum_{i \neq r} b_i \;\;,$$

essendo linearmente dipendente dagli altri, può essere omesso dalla formulazione. Il flusso che raggiunge ciascun nodo i deve quindi transitare dagli archi uscenti dalla radice; questi archi (come gli altri) hanno capacità massima pari a Q. Ciascuna variabile y_{ij} vale 1 se sull'arco (i, j) passa flusso, ossia $x_{ij} > 0$; i vincoli sulle y garantiscono che passi flusso solamente su uno (al più) degli archi entranti in ciascun nodo e quindi, dato il vincolo di connessione implicito nei vincoli di flusso, l'insieme degli archi corrispondenti a variabili y_{ij} di valore 1 in una soluzione ammissibile forma un albero di copertura del grafo (orientato e radicato in r). Tutto il flusso che raggiunge ciascun nodo i deve passare per l'unico arco che entra nella radice del sottoalbero a cui i appartiene (provenendo direttamente da r); quindi, la somma dei b_i relativi ai nodi in uno stesso sottoalbero non può essere superiore alla capacità di tale arco, ossia a Q.

Passare al rilassamento continuo di tale formulazione ha due effetti sulla struttura delle soluzioni ammissibili. Il primo è che, analogamente a quanto visto nell'esempio precedente, possono assumere valore diverso da zero più variabili y corrispondenti ad archi che entrano nello stesso nodo; quindi, l'insieme degli archi corrispondenti a variabili $y_{ij} > 0$ può contenere strettamente un albero di copertura, il che non permette più di definire i sottoalberi della radice (il flusso che soddisfa la richiesta di un qualche nodo i può passare su più di un arco uscente dalla radice). Inoltre, è facile vedere che se x^* è il flusso corrispondente ad una soluzione ottima del rilassamento continuo, allora il valore delle corrispondenti variabili y è

$$y^*_{ij} = (x^*_{ij} + x^*_{ji})/Q \quad \{i, j\} \in E \;\; .$$

In altre parole, il costo che viene "visto" dal rilassamento continuo per utilizzare un lato $\{i, j\}$ non è necessariamente il costo originale c_{ij}, ma una sua frazione che dipende da quanto flusso passa sull'arco. Quindi, anche se le variabili y individuassero un albero di copertura per G (che sarebbe quindi ammissibile per i vincoli relativi al massimo peso dei sottoalberi), il valore ottimo della funzione obiettivo sarebbe una valutazione inferiore, potenzialmente molto distante, del reale costo di quell'albero. Si consideri ad esempio l'istanza di (CMST) mostrata in Figura 6.1(a), in cui tutti i nodi hanno peso unitario e $Q = 2$. È facile verificare che la soluzione ottima del rilassamento continuo si ottiene ponendo $x_{r1} = x_{r2} = x_{r3} = 1$ e $y_{r1} = y_{r2} = y_{r3} = 1/2$ (e a zero tutte le altre variabili), che corrisponde all'albero mostrato in Figura 6.1(b): il valore della funzione obiettivo è 6, mentre il costo dell'albero è 12. Si noti che l'albero non è ottimo, un albero ottimo, di costo 11, essendo mostrato in Figura 6.1(c); la migliore soluzione del rilassamento continuo compatibile con tale albero è $x_{r1} = 2$, $x_{r2} = x_{13} = 1$, $y_{r1} = 2$, $y_{r2} = y_{13} = 1/2$, di costo 7.5.

In questo caso, quindi, il costo di utilizzare un certo lato all'interno di una soluzione può essere fortemente sottostimato rispetto al costo originale. Ragionamenti analoghi possono essere fatti per problemi quali il progetto di rete (cf. §1.2.6.1), la dislocazione ottima di impianti (cf. §1.2.13.1) e, più in generale, i problemi in cui siano presenti funzioni lineari a tratti (es. di carico fisso), minime quantità positive prefissate o vincoli disgiuntivi. In ciascuno di questi casi, passando al rilassamento continuo i vincoli numerici del tipo "$x \leq Uy$" perdono completamente il loro originario significato logico "y è pari a 1 se x è maggiore di zero"; la risorsa rappresentata da y può essere costruita—-e quindi pagata—solo limitatamente alla frazione strettamente necessaria a permettere un dato valore di x, il che può drasticamente sottostimare il reale costo di una soluzione.

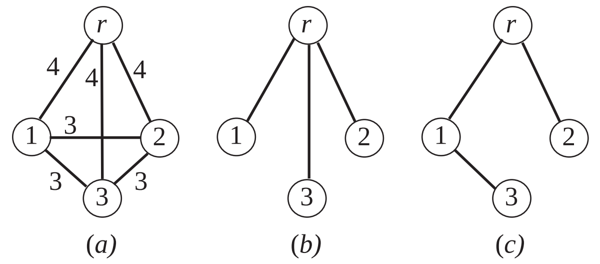


Figura 6.1: Un'istanza di CMST (a) e due soluzioni ammissibili

Gli esempi precedenti mostrano il principale limite del rilassamento continuo. In un modello *PLI* di un problema di *OC* le variabili intere vengono utilizzate principalmente per esprimere condizioni logiche; il rilassamento del vincolo di integralità può distruggere quindi la struttura fondamentale del modello, dando vita ad un problema di ottimizzazione le cui soluzioni ottime possono avere poco o nulla a che fare con quelle del problema originario. Ciò giustifica intuitivamente il fatto che, spesso, il rilassamento continuo risulti scarsamente efficace, anche se esistono tecniche—che verranno illustrate in seguito—che possono permettere di diminuire il gap corrispondente.

Esercizio 6.2 *Si discuta come sia possibile migliorare la valutazione inferiore ottenuta dal rilassamento continuo modificando la formulazione* PLI*, ed in particolare i vincoli (6.4) per gli archi* (i, j) *che non escono direttamente dalla radice (suggerimento:* Q *può essere sostituito con* $Q - b_j$*; si discuta una formula più generale che fornisca il più piccolo coefficiente possibile).*

6.1.2 Informazione generata dal rilassamento continuo

Una volta risolto il rilassamento continuo di un problema di *OC*, ad esempio utilizzando il metodo del simplesso, si ha a disposizione, oltre ad una valutazione superiore del valore ottimo della funzione obiettivo, molta informazione che può essere sfruttata all'interno del processo risolutivo del problema. In particolare si hanno a disposizione una soluzione primale ottima x^* ed una corrispondente soluzione duale ottima y^*; nel seguito illustreremo brevemente alcuni possibili usi di tale informazione.

6.1.2.1 Uso dell'informazione primale

Come enunciato attraverso il Lemma 4.1, e ricordato per il problema dello zaino, l'informazione primale prodotta dal rilassamento continuo è di grande utilità qualora risulti intera: infatti, in questo caso è certamente una soluzione ottima per il problema *OC* che si vuole risolvere. Qualora questo non avvenga, ossia x^* abbia componenti frazionarie, è spesso possibile utilizzare la soluzione frazionaria per guidare la costruzione di "buone" soluzioni intere attraverso opportune euristiche, le più semplici delle quali sono le *tecniche di arrotondamento.* Abbiamo già di fatto visto all'opera una tecnica di arrotondamento nel paragrafo 5.1.2.1 per il caso del problema dello zaino, in cui sappiamo che al più una variabile può essere frazionaria. Discutiamo adesso un caso più complesso ed interessante relativo al Problema di Copertura presentato nel paragrafo 1.2.5. Consideriamo quindi il rilassamento continuo

$$(\underline{\text{PC}}) \qquad \min \left\{ \sum_{j=1}^{m} c_j x_j \;:\; \sum_{j\,:\,i \in F_j} x_j \geq 1 \quad i \in E \;\;,\;\; x \in [0,1]^m \right\}$$

e sia x^* una sua soluzione ottima. Un modo ovvio di determinare una soluzione ammissibile per (PC) è quello di selezionare tutti gli insiemi F_j tali che $x^*_j > 0$; ciò però tipicamente produce soluzioni di scarsa qualità. Un modo migliore per costruire una soluzione ammissibile consiste nel selezionare solo gli insiemi che hanno x_j^* "sufficientemente grande". A tal fine, definiamo la *frequenza* f_i di ciascun elemento $i \in E$ come il numero di sottoinsiemi F_j di cui i fa parte, e sia $f = \max\{f_i : i \in E\}$ la massima delle frequenze. Costruiamo quindi una soluzione S per (PC) selezionando tutti gli insiemi F_j tali che $x_j^* \geq 1/f$. S è sicuramente una soluzione ammissibile per (PC): infatti, dato un qualsiasi elemento $i \in E$, esiste almeno un sottoinsieme F_j che lo contiene e che ha $x_j^* \geq 1/f$ (nel caso peggiore tutti gli insiemi j che contengono i hanno $x_j^* = 1/f$, analogamente a quanto accade in 6.1.1.1). È facile dimostrare che questa soluzione ha un errore relativo al più $f - 1$. Infatti, si consideri la soluzione intera $\bar{x}$ ad essa corrispondente: per ogni j tale che $\bar{x}_j = 1$ si ha $x_j^* \geq 1/f$, e quindi $\bar{x}_j \leq f x_j^*$. Quindi il costo di S, $c\bar{x}$, è minore o uguale di fcx^*, ma $cx^* \leq z(PC)$, da cui

$$R_{arr} = \frac{c\bar{x} - z(PC)}{z(PC)} \leq \frac{fcx^* - cx^*}{cx^*} = f - 1 \; .$$

Sono state proposte tecniche di arrotondamento per molti altri problemi di OC; spesso per queste tecniche è anche possibile dimostrare garanzie sulle prestazioni, come il caso di (PC) mostra. Per ulteriori dettagli su questi temi si rimanda alla letteratura citata.
L'informazione primale generata dal rilassamento continuo ha anche altri usi importanti, ad esempio all'interno delle *regole di separazione* degli algoritmi di enumerazione implicita che verranno discussi nel Capitolo 7.

6.1.2.2 Uso dell'informazione duale

La soluzione duale y^* del rilassamento continuo di (una formulazione PLI di) un problema di OC può essere interpretata come un'indicazione dell'"importanza" relativa dei diversi vincoli lineari della formulazione. Abbiamo infatti visto nel paragrafo 2.2.3 come le variabili duali ottime di un problema di PL forniscano i *prezzi o costi ombra* delle risorse associate a ciascun vincolo; in particolare, le condizioni degli scarti complementari assicurano che siano diverse da zero solo variabili duali corrispondenti a vincoli soddisfatti come uguaglianza, ossia a risorse completamente sfruttate. Quindi, il fatto che $y_i^* = 0$ indica che il corrispondente vincolo primale $A_i x \leq b_i$ è irrilevante ai fini della determinazione della soluzione ottima del rilassamento continuo. Si può quindi pensare che le variabili duali contengano una qualche indicazione su quali dei vincoli del problema siano più o meno rilevanti ai fini della determinazione della soluzione ottima del problema di OC originario. Questa indicazione è comunque da considerarsi euristica, in quanto è valida per la soluzione ottima del rilassamento continuo, che, come abbiamo visto, può essere fortemente scorrelata dalla soluzione ottima del problema di OC. Se però il rilassamento continuo fornisce una "buona approssimazione" del problema di OC, si può pensare che la soluzione del rilassamento continuo fornisca informazione rilevante per la soluzione del problema di OC; questa è in effetti la giustificazione delle tecniche di arrotondamento che abbiamo visto nel paragrafo precedente, che possono, per alcuni problemi, essere notevolmente efficaci. In questo caso si può pensare che la soluzione duale contenga anch'essa informazione rilevante per la soluzione ottima del problema. Anche qualora il rilassamento continuo sia molto accurato, questa informazione è da considerarsi comunque "approssimata". Pur senza entrare nel dettaglio, notiamo a questo proposito che il numero di variabili duali diverse da zero, ossia di vincoli "importanti", in un problema di PL nella forma di (P) da noi comunemente utilizzata, è al più pari al numero n delle variabili; viceversa, si può dimostrare che il numero di vincoli di una formulazione PLI la cui rimozione cambia la soluzione ottima del problema—e che quindi sono decisamente "importanti"—può essere anche $O(2^n)$. Quindi, l'informazione duale fornita dal rilassamento continuo non può essere corretta in tutti i casi; esistono però problemi per i quali tale informazione risulta comunque utile.

Descriviamo adesso un possibile modo per sfruttare l'informazione duale prodotta dal rilassamento continuo, che viene comunemente indicato come *fissaggio basato sui costi ridotti*. Questo approccio si applica a problemi di PLI in cui le variabili abbiano limitazioni superiori e/o inferiori; per semplicità ci limiteremo ad esporlo per il caso della Programmazione 0/1. Sia quindi dato il problema

$$(P) \qquad \max\{\, cx \;:\; Ax \leq b \;,\; x \in \{0,1\}^n \,\}$$

ed il suo rilassamento continuo

$$(\bar{P}) \qquad \max\{\ cx\ :\ Ax \leq b\ ,\ x \in [0,1]^n\ \}\ ;$$

il duale di $(\bar{P})$ è

$$(\bar{D}) \qquad \min\{\ yb + ze\ :\ yA + z - w = c\ ,\ y \geq 0\ ,\ z \geq 0\ ,\ w \geq 0\ \}$$

dove e è il vettore avente tutte le componenti pari ad 1. Siano adesso x^* ed (y^*, z^*, w^*) una coppia di soluzioni ottime ripsettivamente per $(\bar{P})$ e $(\bar{D})$; indichiamo con

$$c_i^* = c_i - y^* A^i = z_i^* - w_i^*$$

il *costo ridotto* della variabile x_i. Dalle condizioni degli scarti complementari (si vedano ad esempio le discussioni fatte nei paragrafi 5.1.2.1, 5.1.2.4 e 5.1.3) si ottengono facilmente le seguenti relazioni:

$$\begin{array}{rcl} z_i^* > 0 \Longrightarrow w_i^* = 0 & \text{e} & w_i^* > 0 \Longrightarrow z_i^* = 0 \\ x_i^* < 1 \Longrightarrow c_i^* \leq 0 & \text{e} & x_i^* > 0 \Longrightarrow c_i^* \geq 0 \\ c_i^* > 0 \Longrightarrow x_i^* = 1 & \text{e} & c_i^* < 0 \Longrightarrow x_i^* = 0 \end{array} \quad .$$

Il costo ridotto di una variabile è quindi dato dai prezzi ombra dei vincoli $x_i \geq 0$ e $x_i \leq 1$; in particolare, se il costo ridotto è positivo allora il vincolo $x_i \leq 1$ è soddisfatto come uguaglianza, e si potrebbe ottenere un incremento del valore della funzione obiettivo (del rilassamento continuo) aumentando il lato destro del vincolo (ossia trasformandolo in $x_i \leq u$ con $u > 1$), mentre se il costo ridotto è negativo allora il vincolo $x_i \geq 0$ è soddisfatto come uguaglianza, e si potrebbe ottenere un incremento del valore della funzione obiettivo aumentando il lato sinistro del vincolo (ossia trasformandolo in $x_i \geq l$ con $l > 0$). Se disponiamo di una valutazione inferiore $\underline{z}$ su $z(P)$, ad esempio fornita dal costo di una soluzione ammissibile, possiamo utilizzare i costi ridotti per cercare di determinare se alcune delle variabili che hanno valore intero (0 o 1) in x^* debbano avere tale valore anche nella soluzione ottima di (P). Supponiamo ad esempio che sia $x_i^* = 0$ e $c_i^* = -w_i^* < 0$, e consideriamo il problema di *PL* $(\bar{P}')$ identico a $(\bar{P})$ tranne per il fatto che il vincolo $x_i \geq 0$ è rimpiazzato col vincolo $x_i \geq 1$ (nella forma standard, il vincolo, $-x_i \leq 0$ diviene $-x_i \leq -1$, ossia il lato destro del vincolo è *diminuito* di un'unità); come abbiamo visto nel paragrafo 2.2.3, si ha che

$$z(\bar{P}') \leq z(\bar{P}) + (-1)w_i^* = z(\bar{P}) + c_i^* \ .$$

Se quindi accade che $z(\bar{P}) + c_i^* < \underline{z}$, allora è possibile fissare $x_i = 0$, perchè tale deve essere il valore della variabile in tutte le soluzioni ottime di (P). Infatti, $(\bar{P}')$ è il rilassamento continuo del problema di *PLI* (P') identico a (P) tranne per il fatto che x_i è fissata al valore 1 (sono presenti i vincoli $1 \leq x_i \leq 1$); si ha quindi

$$z(P') \leq z(\bar{P}') \leq z(\bar{P}) + c_i^* < \underline{z} \leq z(P) \ ,$$

ossia fissare ad 1 la variabile x_i produce certamente un peggioramento del valore ottimo della funzione obiettivo di (P), e quindi tale variabile può essere fissata ad 0. Si noti che, per essere in grado di fissare la variabile, deve risultare

$$z(\bar{P}) - z(P) \leq z(\bar{P}) - \underline{z} \leq -c_i^* \ ,$$

ossia il costo ridotto deve essere (in valore assoluto) più grande del gap assoluto corrispondente al rilassamento continuo. Quindi, sarà tanto più facile fissare variabili in base ai costi ridotti quanto più il gap del rilassamento continuo è piccolo e $\underline{z}$ è una valutazione accurata di $z(P)$.

Esercizio 6.3 *Si ripeta il ragionamento precedente per il caso di una variabile per cui risulti $c_i^* > 0$ e $x_i^* = 1$.*

Esercizio 6.4 *Si estenda il fissagio per costi ridotti al caso generale di un problema di* PLI *in cui le variabili hanno limiti superiori ed inferiori $l_i \leq x_i \leq u_i$; si può ancora parlare di "fissaggio" di variabili?*

Esercizio 6.5 *Si discutano le relazioni tra il procedimento appena visto e quello presentato nel Paragrafo 4.3 relativamente alla determinazione di una soluzione ottima di problemi di* OC *auto-riducibili.*

L'informazione duale prodotta dal rilassamento continuo può quindi essere usata per dedurre proprietà importanti della soluzione ottima. Comunque, vedremo un ulteriore possibile utilizzo di tale informazione duale al termine di questo capitolo.

6.2 Eliminazione di vincoli

Come abbiamo visto nel paragrafo precedente, rilassare il vincolo di integralità può distruggere quasi completamente la struttura di un problema, rendendo inefficace il rilassamento. In questo caso è quindi necessario sviluppare metodologie alternative per costruire rilassamenti efficaci. Moltissimi problemi di *PLI* posseggono un qualche tipo di "struttura" che può essere sfruttata per costruire rilassamenti. Un possibile modo generale per descrivere la struttura di un problema è quello di considerare formulazioni del tipo

$$(P) \qquad \max\{\, cx \ : Ax \leq b \ , \ Ex \leq d \ , \ x \in \mathbb{Z}^n \,\} \tag{6.6}$$

in cui i vincoli $Ex \leq d$ sono "facili", ossia la rimozione dei *vincoli complicanti* $Ax \leq b$ trasformerebbe (P) in un problema "facile". In altre parole, l'introduzione dei vincoli $Ax \leq b$ distrugge la "struttura" presente nei vincoli $Ex \leq d$, che invece permetterebbe di usare algoritmi specifici, più efficienti, per la soluzione del problema. Alternativamente, o in aggiunta a questo, i vincoli $Ex \leq d$ possono essere *separabili*, ossia il problema si decomporrebbe in sottoproblemi indipendenti di dimensione inferiore se non fosse per la presenza dei vincoli $Ax \leq b$, che "legano insieme" le variabili dei singoli sottoproblemi; in questo caso i vincoli $Ax \leq b$ sono anche detti *vincoli accoppianti*. Chiaramente, in tale situazione un possibile rilassamento di (P) si ottiene rimuovendo dal problema i vincoli complicanti, ossia considerando il problema

$$(P') \qquad \max\{\, cx \ : \ Ex \leq d \ , \ x \in \mathbb{Z}^n \,\} \ .$$

(P') viene detto *rilassamento per eliminazione di vincoli* di (P) rispetto ai vincoli $Ax \leq b$. Poichè (P') è un problema di *PLI*, e quindi di *OC*, un tale rilassamento viene anche detto *combinatorio* per distinguerlo dal rilassamento continuo. Chiedere che (P') sia un rilassamento efficiente corrisponde ad assumere che la rimozione dei vincoli $Ax \leq b$ permetta di ottenere un problema "facile"; è bene peršottolineare che, in questo contesto, "facile" non significa necessariamente polinomiale, ma solamente sostanzialmente più facile da risolvere di (P), in pratica e per le dimensioni delle istanze in esame, ossia tale che sia possibile calcolare $z(P')$ in modo notevolmente più efficiente di quanto non sia possibile calcolare $z(P)$. È anche utile far presente che, in questo contesto, "risolvere" (P') significa calcolare *esattamente* $z(P')$, o al limite una sua (buona) valutazione superiore: infatti, una soluzione euristica di (P') non necessariamente fornisce una valutazione superiore di $z(P)$.

6.2.1 Esempi di rilassamenti per eliminazione di vincoli

In generale, dato un problema di *OC*, ed una sua formulazione *PLI*, possono esistere più modi per costruire rilassamenti per eliminazione di vincoli, in quanto possono esistere più blocchi di vincoli la cui rimozione consenta di ottenere un problema "facile"; quindi, esiste spesso una molteplicità di rilassamenti per eliminazione di vincoli corrispondenti ad uno stesso problema di *OC*. Si noti che il rilassamento continuo potrebbe essere considerato una caso speciale del rilassamento per eliminazione di vincoli, in quanto corrisponde all'eliminazione dei vincoli di integralità. Inoltre, può capitare che formulazioni *PLI* diverse dello stesso problema suggeriscano rilassamenti diversi, in quanto evidenzino parti diverse della struttura del problema. Discutiamo adesso alcuni rilassamenti per eliminazione di vincoli per un certo numero di problemi di *OC*.

6.2.1.1 Il problema (CMST)

Come è stato osservato in precedenza, molti problemi di *OC* "difficili" sono varianti apparentemente minori di problemi "facili". (CMST) è un tipico esempio. In casi come questi è solitamente facile individuare rilassamenti efficienti per eliminazione di vincoli; è sufficiente eliminare i vincoli corrispondenti alle condizioni "in più" rispetto a quelle presenti nel problema "facile". Per (CMST), ad esempio, questo significa rimuovere il vincolo sul massimo peso dei sottoalberi: (MST) è chiaramente un rilassamento efficiente per eliminazione di vincoli di (CMST). Tipicamente, l'efficacia di un tale rilassamento dipenderà dal valore della costante Q, che è completamente ignorata in (MST): se la costante è "grande" si può sperare che solo "pochi" sottoalberi della radice nella soluzione ottima di (MST) violino il vincolo, e quindi che il gap relativo al rilassamento sia basso (esiste certamente un valore di Q sufficientemente grande per cui il gap è nullo), mentre al diminuire di Q è presumibile che il gap cresca. Può essere interessante notare come la formulazione *PLI* (6.1)–(6.5) suggerisca un diverso rilassamento. In particolare, in questo caso possono essere considerati "complicanti" i vincoli (6.4), che sono gli unici a "collegare"le variabili x e le variabili y (sono quindi "accoppianti"). Il problema ottenuto dalla rimozione di tali vincoli è quindi separabile in due sottoproblemi indipendenti: un problema di flusso sulle variabili x, ed il problema di selezionare, per ciascun nodo (tranne la radice), l'arco entrante di costo minimo, che è a sua volta separabile in $n-1$ sottoproblemi indipendenti. Poichè in effetti la soluzione del problema di flusso non ha alcuna influenza sul valore della funzione obiettivo, questo rilassamento può essere risolto in $O(m)$ semplicemente scandendo la lista degli archi del grafo. È facile verificare come questo rilassamento sia a sua volta un rilassamento di (MST): in un albero di copertura (orientato) ciascun nodo (tranne la radice) ha un arco entrante. Quindi, il rilassamento basato su (MST) ha sicuramente un gap inferiore, ma ha anche una complessità superiore. È interessante notare come questo rilassamento potrebbe essere stato ideato riflettendo sul problema di *OC* originale, ma venga facilmente suggerito dall'esame di una formulazione *PLI* del problema. Ciò mostra come formulazioni diverse delle stesso problema possano essere utili per suggerire rilassamenti diversi.

Esercizio 6.6 *Si dimostri con due esempi che non esiste nessuna relazione di dominanza tra il gap fornito dal rilassamento continuo (basato sulla formulazione (6.1)–(6.5)) e quello basato su (MST), ossia che esistono istanze in cui l'uno è migliore dell'altro ed altre istanze in cui accade il contrario.*

Esercizio 6.7 *Si esaminino tutti i possibili rilassamenti per eliminazione di vincoli ottenibili dalla formulazione (6.1)–(6.5), discutendo per ciascuno di essi se possa essere considerato "facile" e se si possano individuare relazioni di dominanza rispetto al gap relativo ai rilassamenti discussi finora.*

6.2.1.2 Il problema del cammino minimo vincolato

Un caso analogo al precedente si ha per il problema del *cammino minimo vincolato* (CSP, da Constrained Shortest Path). Sia $G = (N, A)$ un grafo orientato e pesato dove ad ogni arco $(i, j) \in A$ è associato un costo $c_{ij} \in \mathbb{R}_+$ ed una lunghezza $l_{ij} \in \mathbb{R}_+$, e siano dati i due nodi r e t: si vuole determinare un cammino di costo minimo tra tutti i cammini da r a t di lunghezza inferiore o uguale ad una data soglia L. Una formulazione *PLI* per (CSP) può essere ottenuta immediatamente da quella del problema del cammino minimo introdotta nel paragrafo 3.2.1

$$
\begin{array}{lll}
\min \sum_{(i,j)\in A} c_{ij}x_{ij} & & \\
\sum_{(j,i)\in A} x_{ji} - \sum_{(i,j)\in A} x_{ij} = b_i & i \in N & (6.7) \\
\sum_{(i,j)\in A} l_{ij}x_{ij} \leq L & & (6.8) \\
x_{ij} \in \{0, 1\} & (i,j) \in A &
\end{array}
$$

dove $b_i = -1$ se $i = r$, $b_i = 1$ se $i = t$, e $b_i = 0$ altrimenti. Per questo problema, il singolo vincolo (6.8) risulta "complicante"; l'introduzione del vincolo rende il problema $\mathcal{NP}$-arduo, mentre la sua eliminazione consente di ottenere un problema polinomiale. Quindi, un ovvio ed efficiente rilassamento per eliminazione di vincoli di (CSP) si ottiene eliminando il vincolo (6.8) relativo alla massima lunghezza dei cammini. Anche in questo caso si può pensare che l'efficacia del rilassamento sia legata al valore della costante L: per valori "grandi" il vincolo è scarsamente rilevante (o addirittura irrilevante per valori sufficientemente grandi) ed è ragionevole attendersi gap relativamente bassi, che però tenderanno a crescere con il diminuire di L. In effetti, si noti che per valori sufficientemente piccoli di L (CSP) potrebbe non avere alcuna soluzione ammissibile, ossia $z(CSP) = +\infty$, mentre il corrispondente problema del cammino minimo potrebbe avere soluzione: in questo caso il gap sarebbe infinito. In questo caso non sono facilmente costruibili altri rilassamenti per eliminazione di vincoli "ragionevoli"; rilassando i vincoli (6.7) si otterrebbe infatti un problema la cui soluzione ottima è nulla, in quanto nessun vincolo forza le variabili ad assumere valori diversi da zero.

Esercizio 6.8 *Si discuta la struttura delle soluzioni ammissibili del rilassamento continuo della formulazione data di (CSP) e le eventuali relazioni di dominanza, in termini di gap, tra i due rilassamenti. Cosa si può dire in termini di efficienza?*

6.2.1.3 Il problema del flusso "multicommodity" indivisibile

Un problema in parte analogo al precedente si presenta, spesso come sottoproblema, durante la progettazione di reti di comunicazione. È dato un grafo $G = (N, A)$, che rappresenta ad esempio una rete di comunicazione, con costi di routing $c_{ij} \in \mathbb{R}_+$ e capacità $u_{ij} \in \mathbb{R}_{++}$ associate agli archi. Su questo grafo è individuato un insieme K di coppie origine/destinazione (o_h, d_h), ciascuna con un'associata domanda di comunicazione δ_h. Per ciascuna coppia (o_h, d_h) si vuole selezionare un singolo cammino lungo il quale instradare tutte le δ_h unità di flusso che rappresentano la comunicazione tra l'origine e la destinazione, rispettando i vincoli di capacità associati agli archi, e minimizzando il costo di routing complessivo, dato dalla somma pesata dei costi dei cammini utilizzati. Introducendo variabili di flusso $x^h = [x^h_{ij}]_{(i,j)\in A}$ per ciascuna coppia $h \in K$ (detta *commodity*), il problema del *flusso Multicommodity "indivisibile" di costo minimo* (UMMCF, da Unsplittable Multicommodity Min Cost Flow problem) può essere formulato come

$$
\begin{array}{llr}
\min \sum_{h\in K} \delta_h \sum_{(i,j)\in A} c_{ij} x^h_{ij} & & \\
\sum_{(j,i)\in A} x^h_{ji} - \sum_{(i,j)\in A} x^h_{ij} = b^h_i & i \in N, h \in K & (6.9) \\
\sum_{h\in K} \delta_h x^h_{ij} \le u_{ij} & (i,j) \in A & (6.10) \\
x^h_{ij} \in \{0,1\} & (i,j) \in A \;\; , \;\; h \in K & (6.11)
\end{array}
$$

dove $b^h_i = -1$ se $i = o_h$, $b^h_i = 1$ se $i = d_h$, e $b^h_i = 0$ altrimenti. I vincoli di capacità (6.10) risultano "complicanti" per due motivi: il primo è che se fossero rimossi il problema si decomporrebbe in $|K|$ sottoproblemi indipendenti, uno per ciascuna coppia origine/destinazione, e quindi i vincoli risultano "accoppianti". Il secondo è che ciascuno di tali sottoproblemi è un problema di cammino minimo, e quindi può essere risolto molto efficientemente. La presenza dei vincoli (6.10) rende invece (UMMCF) "difficile", sia per le grandi dimensioni sia per il fatto che la totale unimodularità dei vincoli di flusso (6.9) viene perduta, rendendo il problema $\mathcal{NP}$-arduo. In altri termini, come nei due casi precedenti, mentre la soluzione ottima del rilassamento per eliminazione dei vincoli (6.10) è certamente intera, la soluzione ottima del rilassamento continuo può non esserlo.

Esercizio 6.9 *Si mostri attraverso un esempio che, anche se tutti i dati del problema sono interi, può esistere una soluzione ottima del rilassamento continuo di (UMMCF) non intera.*

Come negli esempi precedenti, la rimozione dei vincoli di capacità congiunta (6.10) può essere poco rilevante se le capacità sono "grandi", ma il gap tenderà a crescere qualora le capacità diminuiscano. Come per il caso di (CSP), rimuovere i vincoli di conservazione di flusso (6.9) porterebbe invece ad un problema che ha soluzione ottima nulla.

6.2.1.4 Il problema (TSP)

Come è già stato notato nei paragrafi 1.2.2.3, 1.2.4.1 e 5.1.3, (TSP) può essere visto come un problema di ottimizzazione il cui insieme ammissibile è l'intersezione tra l'insieme ammissibile di un problema di accoppiamento e quello di un problema di albero di copertura di costo minimo. In altri termini, nella formulazione *PLI*

$$
(\mathrm{TSP})\qquad
\begin{array}{lll}
\min & \sum_{(i,j)\in A} c_{ij}x_{ij} & \\
 & \sum_{i\in S, j\notin S} x_{ij} \geq 1 & \emptyset \subset S \subset N \\
 & \sum_{(i,j)\in A} x_{ij} = 2 & i \in N \\
 & x_{ij} \in \{0,1\} & (i,j) \in A
\end{array}
$$

sia il primo blocco di vincoli (vincoli di connessione) che il secondo blocco di vincoli (vincoli di copertura per cicli) possono essere considerati "complicanti". La rimozione di ciascun blocco di vincoli lascia un rilassamento efficiente, in quanto risolubile con un algoritmo polinomiale (sia pure di diversa complessità); l'efficacia relativa dei due approcci dipenderà tipicamente dalla singola istanza a cui vengono applicati. (TSP) è un tipico caso in cui all'interno dello stesso problema sono presenti più "strutture" che, prese singolarmente, sono "facili" da trattare, ma la cui intersezione caratterizza un problema difficile. Per questi problemi è quindi possibile definire una molteplicità di rilassamenti per eliminazione di vincoli "ragionevoli", in ciascuno dei quali vengono mantenuti solamente i vincoli che caratterizzano ciascuna delle strutture "facili".
Può essere utile notare che, in qualche caso, si riescono a gestire, all'interno di una delle strutture "facili" del problema, "forme più deboli" dell'altra struttura. Per il (TSP), ad esempio, è facile dimostrare che tutte le coperture per cicli del grafo hanno esattamente n lati, mentre gli alberi di copertura ne hanno $n-1$. In altri termini, alla formulazione di (MST) (ad esempio quella (1.7)–(1.6)) si può aggiungere il vincolo

$$\sum_{(i,j)\in A} x_{ij} = n \quad ;$$

questo viene detto il problema dell'*1-albero di copertura di costo minimo* (MS1-T), ed è un rilassamento valido di (TSP), non peggiore di (MST) (se i costi sono non negativi) in quanto (MST) è un rilassamento di (MS1-T).

Esercizio 6.10 *Si dimostri l'affermazione precedente mostrando che (MS1-T) è un rilassamento surrogato di (TSP) (si veda il paragrafo 6.4).*

Un 1-albero di copertura di un grafo G è un sottografo connesso di G con esattamente n lati. Si può dimostrare che l'1-albero di copertura di costo minimo di$G$ può essere calcolato efficientemente determinando un albero di copertura T di costo minimo di G, e poi aggiungendo a T il lato di costo minimo in G che non appartenga già a T.

Esercizio 6.11 *Si dimostri l'affermazione precedente (suggerimento: la procedura appena descritta è una modifica dell'algoritmo di Kruskal).*

Quindi, introducendo in (MST) una forma rilassata dei vincoli di copertura si ottiene un rilassamento non peggiore ed altrettanto facile da risolvere. Nel prossimo paragrafo presenteremo un modo diverso per fornire ai rilassamenti combinatori informazione sui vincoli rilassati.

Per terminare questa sezione sottolineiamo che non per tutti i problemi di *OC* sia facile costruire rilassamenti combinatori "ragionevoli". Si consideri ad esempio il problema dello zaino: questo problema ha un solo vincolo, a parte quelli di integralità. Rilassare questo vincolo porta ad un problema privo di vincoli, la cui soluzione ottima consiste nel prendere tutti gli oggetti. Per un caso più rilevante si consideri invece il problema (MMMS), ed in particolare la sua formulazione *PLI* presentata nel paragrafo 1.2.9.1. Rilassare ciascuno dei blocchi di vincoli del problema porta ad un problema la cui soluzione ottima non fornisce alcuna reale informazione sull'istanza in questione. Infatti, rilassare i vincoli di semiassegnamento elimina tutti i vincoli che costringono alcune delle variabili x ad essere

non nulle; di conseguenza, la soluzione ottima del rilassamento corrisponde a non assegnare nessun lavoro a nessuna macchina, con un conseguente makespan pari a zero. Rilassare i vincoli che computano il makespan, ossia quelli che garantiscono che la singola variabile t sia maggiore o uguale del tempo di completamento di ogni macchina, fa sì che nel rilassamento non ci siano vincoli che "legano" t alle variabili x; di conseguenza, qualunque semiassegnamento scelto avrebbe un valore della funzione obiettivo pari a zero. In entrambi i casi il rilassamento fornisce una valutazione inferiore del valore ottimo del problema pari a zero, per qualsiasi istanza del problema.
Gli esempi precedenti mostrano che eliminare completamente alcuni vincoli da un problema di *OC* può permettere di costruire un rilassamento efficiente; spesso però tali rilassamenti possono essere scarsamente efficaci, in quanto i vincoli eliminati possono avere un profondo impatto sulla caratterizzazione della soluzione ottima, caratterizzazione che viene completamente perduta nel rilassamento. Sarebbe quindi utile una tecnica in grado di eliminare i vincoli "complicanti" da un problema "tenendone traccia" in qualche modo; questo è lo scopo del rilassamento Lagrangiano, descritto nel prossimo paragrafo.

6.3 Rilassamento Lagrangiano

Un diverso modo per utilizzare la struttura presente in problemi con la forma (6.6) è quello di effettuare un *rilassamento Lagrangiano* di (P) rispetto agli m vincoli complicanti $Ax \le b$, ossia considerare il problema

$$(P_y) \qquad \max\{\, cx + y(\, b - Ax\,) \;:\; Ex \le d \,,\; x \in \mathbb{Z}^n \,\} \tag{6.12}$$

per un fissato vettore $y \in \mathbb{R}^m_+$ di *moltiplicatori Lagrangiani*. Si osservi che, per y fissato, (P_y) è equivalente al rilassamento per eliminazione di vincoli tranne per il diverso vettore dei costi $c_y = c - yA$ (detti *costi Lagrangiani*); considereremo il caso in cui (P_y) sia più facile da risolvere in pratica di (P). È immediato verificare che, comunque si scelga y, (P_y) è un rilassamento di (P), ossia risulta $z(P_y) \ge z(P)$. Infatti la regione ammissibile di (P_y) contiene quella di (P) e si ha

$$cx + y(\, b - Ax\,) \ge cx$$

per ogni x ammissibile per (P). Il termine aggiuntivo $y(b - Ax)$ nella funzione obiettivo ha il compito di "penalizzare" le soluzioni che non rispettano i vincoli rilassati $Ax \le b$: infatti, se $\bar{x}$ rispetta tutti i vincoli rilassati allora $y(b - A\bar{x}) \ge 0$, mentre se $\bar{x}$ viola un dato vincolo $A_i x \le b_i$, e $y_i > 0$, allora $y_i(b_i - A_i\bar{x}) < 0$. Quindi, nel rilassamento Lagrangiano si "tiene traccia" dei vincoli complicanti: nonostante se ne permetta la violazione, si aggiunge alla funzione obiettivo un termine che favorisce le soluzioni che non violano i vincoli rilassati. Si noti che il rilassamento per eliminazione di vincoli è un caso particolare del rilassamento Lagrangiano, in quanto si ottiene ponendo $y = 0$. Vedremo nel seguito che, in un certo senso, il rilassamento Lagrangiano generalizza anche il rilassamento continuo, ed i moltiplicatori Lagrangiani sono funzionalmente analoghi alle variabili duali, nella *PL*, dei vincoli rilassati.

In particolare, le regole che determinano il segno dei moltiplicatori Lagrangiani sono le stesse di quelle valide per la dualità lineare: il moltiplicatore Lagrangiano di un vincolo complicante nella forma $A_i x \ge b_i$ è vincolato ad essere non positivo ($y_i \le 0$), ma accadrebbe il contrario se (P) fosse un problema di minimo. Inoltre, il moltiplicatore Lagrangiano di un vincolo complicante nella forma $A_i x = b_i$ non è vincolato in segno, ed il duale Lagrangiano (definito nel seguito) è un problema di massimo se (P) è un problema di minimo.

Il rilassamento Lagrangiano di un problema di *OC* non è un singolo problema, ma piuttosto una famiglia infinita di problemi dipendenti dal vettore di parametri y. Ciascuno di questi problemi fornisce una, potenzialmente diversa, valutazione superiore di $z(P)$; ha quindi senso porsi il problema di determinare la migliore, ossia la minore, di tali valutazioni. Ciò corrisponde a risolvere il *duale Lagrangiano*

$$(D) \qquad \min\{\, z(P_y) \;:\; y \ge 0 \,\} \tag{6.13}$$

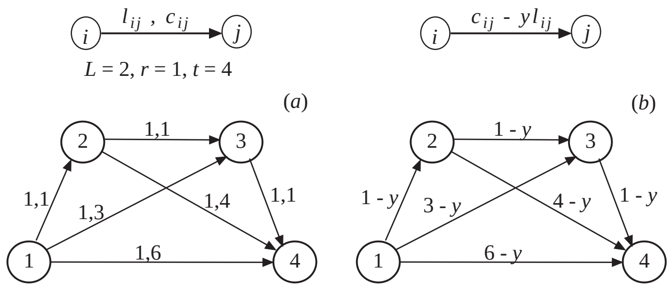
di (P) rispetto ai vincoli complicanti $Ax \leq b$. Siccome vale $z(P_y) \geq z(P)$ per ogni $y \in \mathbb{R}^m_+$, (D) è ancora un rilassamento di (P), ossia vale $z(D) \geq z(P)$.

Esempio 6.1: Duale Lagrangiano di (CSP)

Si consideri ad esempio l'istanza di (CSP) rappresentata nella figura (a) qui accanto; il suo rilassamento Lagrangiano rispetto al vincolo "complicante" (6.8) è il problema di cammino minimo, parametrico rispetto al singolo moltiplicatore Lagrangiano $y \leq 0$ in cui la funzione obiettivo è

$$Ly + \min \sum_{(i,j)\in A} (c_{ij} - y l_{ij}) x_{ij} \quad ,$$

rappresentato in figura (b).



Le soluzioni ammissibili del rilassamento sono i quattro cammini $c_1, \ldots, c_4$ mostrati in Figura 6.2($a1$)–($a4$) con il corrispondente costo Lagrangiano. Al variare di y il costo Lagrangiano delle singole soluzioni varia in modo differente, e quindi per ciascun valore di y una (o più) soluzioni sarà quella di costo minimo. In particolare, come mostrato in Figura 6.2(b), il cammino c_1 ha il minimo costo Lagrangiano per $y \in [-1, 0]$, il cammino c_2 ha il minimo costo Lagrangiano per $y \in [-2, -1]$, il cammino c_4 ha il minimo costo Lagrangiano per $y \leq -2$ mentre il cammino c_3 non ha il minimo costo Lagrangiano per nessun valore di y; si noti che per $y = -1$ e $y = -2$ ci sono due soluzioni che hanno il medesimo costo Lagrangiano minimo (c_1 e c_2 nel primo caso, c_2 e c_4 nel secondo). L'esempio illustra chiaramente le caratteristiche del

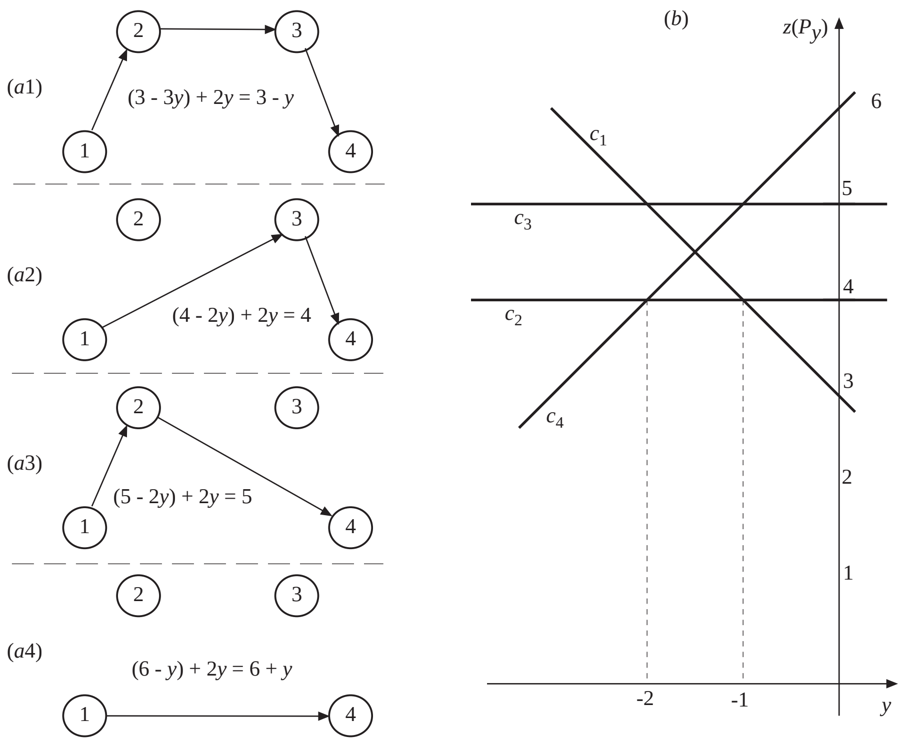


Figura 6.2: Soluzioni ammissibili e duale Lagrangiano

rilassamento Lagrangiano. Poichè tutti i coefficienti l_{ij} sono unitari, il costo Lagrangiano di un qualsiasi cammino p è pari al suo costo più il termine penalità $y(2 - l(p))$, dove $l(p)$ è la lunghezza del cammino (numero di archi che lo compongono). Per $y = 0$ la lunghezza del cammino non viene tenuta in nessun conto, e la soluzione ottima del rilassamento Lagrangiano (e di quello per eliminazione di vincoli) è il cammino c_1, di costo 3 e lunghezza 3, quindi non ammissibile. Mano mano che y diminuisce il costo del cammino c_1, che viola il vincolo, aumenta, il costo dei cammini c_2 e c_3, che rispettano il vincolo come uguaglianza, resta inalterato, mentre il costo del cammino c_4, che rispetta il vincolo come stretta diseguaglianza, diminuisce. Per $y = -1$ il cammino c_2 ha lo stesso costo Lagrangiano di c_1, e per $y < -1$ ha un costo migliore. Per $y = -2$ il cammino c_4 ha lo stesso costo Lagrangiano di c_2, e per $y < -2$ ha un costo migliore; mano mano che y diminuisce i cammini "più corti" sono sempre più convenienti, in termini di costo Lagrangiano, rispetto a quelli "più lunghi". Per $y \in (-2, -1)$ l'unica soluzione del rilassamento Lagrangiano, ovvero il cammino c_2, rispetta il vincolo "complicante" come uguaglianza. Dalla Figura 6.2(b) è evidente che qualsiasi $y \in [-2, -1]$ è una soluzione ottima del duale Lagrangiano, ossia che $z(D) = 4$; questo è anche il costo nel problema (CSP) del cammino c_2, che infatti è la soluzione ottima del problema. In questo particolare caso il duale Lagrangiano fornisce una valutazione esatta di $z(P)$ (il gap è nullo): il termine di penalità $y(b - Ax)$, con un'opportuna scelta del moltiplicatore Lagrangiano y, "guida" il rilassamento Lagrangiano permettendogli di individuare la soluzione ottima del problema originario. Questo non sempre accade; in molti casi, come vedremo, il duale Lagrangiano ha un gap non nullo.

L'esempio precedente mostra come il duale Lagrangiano fornisca una valutazione superiore non peggiore di quella fornita dal corrispondente rilassamento per eliminazione di vincoli; ovvero $z(D) \leq z(P_0)$. Inoltre può accadere che il duale Lagrangiano individui una soluzione ottima di (P). Per questo non è sufficiente, come nel caso del rilassamento continuo (cf. il Lemma 4.1), che la soluzione del rilassamento Lagrangiano sia ammissibile per i vincoli rilassati, ma è necessaria anche la condizione

$$\bar{y}(b - A\bar{x}) = 0 \tag{6.14}$$

Lemma 6.1 *Sia $\bar{x}$ una soluzione ottima di $P_{\bar{y}}$: se $\bar{x}$ è ammissibile per (P) $(A\bar{x} \leq b)$ e vale la condizione (6.14), allora $\bar{x}$ è una soluzione ottima per (P) e $\bar{y}$ è una soluzione ottima per (D).*

Dimostrazione Si ha $c\bar{x} \leq z(P) \leq z(D) \leq z(P_{\bar{y}}) = c\bar{x} + \bar{y}(b - A\bar{x}) = c\bar{x}$. ◇

È interessante notare che (6.14) corrisponde alle condizioni degli scarti complementari introdotte nel caso della *PL*. Nel prossimo paragrafo mostreremo che questa corrispondenza non è casuale, e presenteremo alcuni risultati che consentono, in molti casi, di confrontare la qualità della valutazione superiore fornita da (D) con quella delle valutazioni fornite da rilassamenti diversi.

6.3.1 Teoria del rilassamento Lagrangiano

Nel caso della *PL* la dualità Lagrangiana coincide con quella lineare. Si consideri infatti il rilassamento continuo di (P)

$$(\bar{P}) \qquad \max\{\, cx \;:\; Ax \leq b \,,\; Ex \leq d \,\} \,, \tag{6.15}$$

il rilassamento Lagrangiano di $(\bar{P})$ rispetto ad $Ax \leq b$

$$(\bar{P}_y) \qquad \max\{\, cx + y(b - Ax) \;:\; Ex \leq d \,\} \tag{6.16}$$

ed il corrispondente duale Lagrangiano

$$(\bar{D}) \qquad \min\{\, z(\bar{P}_y) \;:\; y \geq 0 \,\} \,. \tag{6.17}$$

Teorema 6.1 $(\bar{D})$ *è il duale—nel senso della* PL—*di* $(\bar{P})$.

Dimostrazione Poiché $(\bar{P}_y)$ è un problema di *PL* si ha

$$z(\bar{P}_y) = \min\{\, wd \;:\; wE = c - yA \,,\; w \geq 0 \,\} + yb \;\;,$$

e di conseguenza

$$\begin{aligned} z(\bar{D}) &= \min\{\, yb + \min\{\, wd : wE = c - yA \,,\; w \geq 0 \,\} \;:\; y \geq 0 \,\} \\ &= \min\{\, yb + wd \;:\; wE + yA = c \,,\; y \geq 0 \,,\; w \geq 0 \,\} \;\; ; \end{aligned}$$

è immediato riconoscere nella precedente espressione il duale lineare di $(\bar{P})$. ◇

Quindi, nel caso della *PL* il duale Lagrangiano è in effetti il duale lineare (si può dire che sia un *duale parziale*). Ciò giustifica le analogie tra i due duali precedentemente accennate. Nel caso della *PLI* vale comunque una generalizzazione del risultato precedente. A tal fine conviene esprimere il problema originale come

$$(P) \qquad \max\{\, cx \;:\; Ax \leq b \,,\; x \in X \,\} \tag{6.18}$$

dove $X = \{\, x \in \mathbb{Z}^n \;:\; Ex \leq d \,\}$. Come abbiamo visto nel Paragrafo 4.2.2, massimizzare (una funzione lineare) sull'insieme discreto X è equivalente a farlo sul suo *inviluppo convesso* $conv(X)$, ossia

$$z(P_y) = \max\{\, cx + y(b - Ax) \;:\; x \in conv(X) \,\}$$

per ogni $y \in \mathbb{R}^m_+$. Si può dimostrare che se la matrice E ed il vettore d hanno tutte componenti razionali, allora $conv(X)$ è un poliedro convesso, ossia esiste un insieme finito di vincoli lineari tali che

$$conv(X) = \{\, x \;:\; \tilde{E}x \leq \tilde{d} \,\} \;\; .$$

Definiamo quindi il *rilassamento convessificato* di (P) come

$$(\tilde{P}) \qquad \max\{\, cx \;:\; Ax \leq b \;,\; x \in conv(X) \,\} \; . \tag{6.19}$$

$(\tilde{P})$ è quindi un problema di *PL*, per quanto in generale la descrizione poliedrale di $conv(X)$ non sia nota. È però possibile dimostrare il seguente risultato:

Teorema 6.2 *$z(D) = z(\tilde{P})$.*

Dimostrazione Dalla definizione si ha

$$z(\tilde{P}) = \max\{\, cx \;:\; Ax \leq b \;,\; \tilde{E}x \leq \tilde{d} \,\} \; .$$

Dalla dualità lineare si ottiene quindi

$$z(\tilde{P}) = \min\{\, yb + w\tilde{d} \;:\; yA + w\tilde{E} = c \;,\; y \geq 0 \;,\; w \geq 0 \,\}$$

che può essere riscritto come

$$\min\{\, yb + \min\{\, w\tilde{d} \;:\; w\tilde{E} = c - yA \;,\; w \geq 0 \,\} \;:\; y \geq 0 \,\} \;\; .$$

Ancora per dualità lineare, applicata al problema interno, si ha

$$z(\tilde{P}) = \min\{\, yb + \max\{\, (c - yA)x \;:\; \tilde{E}x \leq \tilde{d} \,\} \,\}$$

e quindi il teorema è dimostrato. $\diamond$

Il Teorema 6.2 ha le seguenti importanti conseguenze.

Corollario 6.1 *Il duale Lagrangiano fornisce una valutazione superiore non peggiore (non maggiore) di quella fornita dal rilassamento continuo, ossia $z(D) \leq z(\bar{P})$; in più, se i vincoli $Ex \leq d$ hanno la* proprietà di integralità *allora si ha che $z(D) = z(\tilde{P}) = z(\bar{P})$.*

Dimostrazione La regione ammissibile di $(\tilde{P})$ è contenuta in quella di $(\bar{P})$, e le funzioni obiettivo dei due problemi coincidono. La proprietà di integralità (Definizione 4.1) è equivalente a $conv(X) = \{\, x \;:\; Ex \leq d \,\}$, ossia $\tilde{E} = E$ e $\tilde{d} = d$. $\diamond$

Il corollario precedente mostra che si ha un "principio di conservazione della difficoltà": se il rilassamento Lagrangiano è "facile", ossia il vincolo di integralità è soddisfatto automaticamente, come avviene ad esempio nei problemi di flusso, allora il duale Lagrangiano è equivalente al rilassamento continuo. Per ottenere una valutazione superiore strettamente migliore è necessario che il sottoproblema Lagrangiano sia "difficile", ossia che i vincoli $Ex \leq d$ non forniscano una descrizione "esatta" di $conv(X)$.

Le osservazioni precedenti consentono in alcuni casi di valutare le prestazioni relative di rilassamenti Lagrangiani diversi dello stesso problema. Ad esempio, si consideri il duale Lagrangiano di (P) rispetto al secondo blocco di vincoli

$$(D') \qquad \min\{\, \max\{\, cx + w(d - Ex) \;:\; x \in X' \,\} \;:\; w \geq 0 \,\} \; .$$

dove $X' = \{\, x \in \mathbb{Z}^n \;:\; Ax \leq b \,\}$. Se sia i vincoli $Ax \leq b$ che i vincoli $Ex \leq d$ posseggono la proprietà di integralità allora $z(D') = z(D) = z(\bar{P})$; se invece i vincoli $Ex \leq d$ posseggono la proprietà di integralità ed i vincoli $Ax \leq b$ non la posseggono allora $z(D') \leq z(D) = z(\bar{P})$, e la diseguaglianza può essere stretta.

Si consideri il problema (UMMCF). Il rilassamento Lagrangiano rispetto ai vincoli di capacità (6.10) si decompone in $|K|$ problemi di cammino minimo, uno per ciascuna coppia (o_h, d_h), rispetto ai costi Lagrangiani $c_{ij} - y_{ij}$ (c'è un moltiplicatore Lagrangiano y_{ij} per ciascun vincolo di capacità, e quindi per ciascun arco). Siccome i problemi di cammino minimo hanno la proprietà di integralità, il corrispondente duale Lagrangiano fornisce esattamente la stessa valutazione inferiore del rilassamento continuo. Invece, il rilassamento Lagrangiano rispetto ai vincoli di conservazione di flusso (6.9) ha la forma

$$\begin{array}{lll} \min & \sum_{h \in K} \sum_{(i,j) \in A} (\delta_h c_{ij} - w_i^h + w_j^h) x_{ij}^h & \\ & \sum_{h \in K} \delta_h x_{ij}^h \leq u_{ij} & (i,j) \in A \\ & x_{ij}^h \in \{0,1\} & (i,j) \in A \ , \ \ h \in K \end{array}$$

dato che esiste un moltiplicatore Lagrangiano w_i^h per la copia corrispondente alla commodity h del vincolo di conservazione di flusso relativo al nodo i. Il rilassamento si decompone in $|A|$ problemi indipendenti, uno per ogni arco; quando $w = 0$ questi problemi hanno soluzione ottima identicamente nulla, ma per valori diversi del vettore dei moltiplicatori Lagrangiani alcuni dei costi Lagrangiani $\delta_h c_{ij} - w_i^h + w_j^h$ possono diventare negativi. Quindi, in generale, il rilassamento Lagrangiano richiede la soluzione di $|A|$ problemi di zaino, ciascuno con al più $|K|$ variabili (solo gli oggetti con costo Lagrangiano negativo possono essere inseriti nello zaino in una soluzione ottima), e quindi non possiede la proprietà di integralità. Di conseguenza, il corrispondente duale Lagrangiano può fornire una valutazione inferiore di $z(P)$ strettamente migliore di quella del rilassamento continuo.
Nel caso di (TSP), invece, sia il rilassamento Lagrangiano rispetto ai vincoli di connessione che quello rispetto ai vincoli di copertura per cicli posseggono la proprietà di integralità; di conseguenza i corrispondenti duali Lagrangiani forniscono la stessa valutazione inferiore, che è anche la stessa fornita dal rilassamento continuo. Si noti che, in questo caso, risolvere direttamente il rilassamento continuo, ad esempio mediante un algoritmo del Simplesso, non sarebbe possibile dato il numero esponenziale di vincoli della formulazione. Comunque, anche risolvere un duale Lagrangiano non è in principio banale; alcune tecniche risolutive saranno discusse nel prossimo paragrafo.

Esercizio 6.12 *Si discutano le relazioni tra la valutazione fornita dal rilassamento continuo e quella fornita da tutti i possibili duali Lagrangiani per i problemi (KP), (MMMS), (CMST) e (CSP).*

Un diverso modo di sfruttare il rilassamento Lagrangiano è la cosiddetta *decomposizione Lagrangiana*, che corrisponde a riscrivere il problema nella forma equivalente

$$(P) \qquad \max \ \{ \ c(x+x')/2 \ : \ Ax' \leq b \ , \ x' \in \mathbb{Z}^n \ , \ Ex \leq d \ , \ x \in \mathbb{Z}^n \ , \ x = x' \ \}$$

e a risolvere il duale Lagrangiano rispetto ai vincoli $x = x'$, ossia

$$(D'') \qquad \min \ \Big\{ \ \max\{ \ (c/2 - w)x : x \in X \ \} \ + \ \max\{ \ (c/2 + w)x' : x' \in X' \ \} \ \Big\} \ .$$

Dal Teorema 6.2 segue che:

Corollario 6.2 *$z(D'') = \max\{ \, cx \, : \, x \in conv(X) \cap conv(X') \, \}$, e quindi $z(D'') \leq \min\{z(D'), z(D)\} \leq z(\bar{P})$.*

Esercizio 6.13 *Dimostrare il Corollario precedente.*

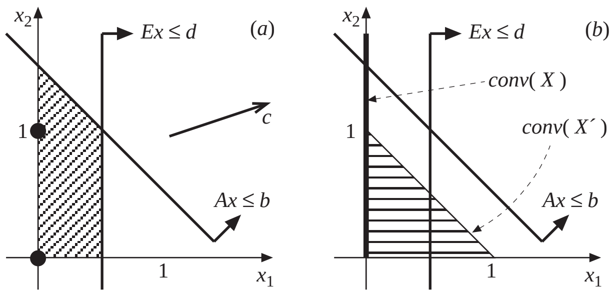
In altri termini, la decomposizione Lagrangiana fornisce una valutazione superiore di $z(P)$ non peggiore (non maggiore) di quella fornita da ciascuno dei due rilassamenti Lagrangiani, e quindi non peggiore di quella fornita dal rilassamento continuo. In particolare, è facile verificare che $z(D'')$ coincide con $z(\bar{P})$ se sia i vincoli $Ax \leq b$ che i vincoli $Ex \leq d$ posseggono la proprietà di integralità, che $z(D'')$ coincide con la valutazione fornita dal migliore dei due rilassamenti Lagrangiani se uno solo dei due insiemi di vincoli possiede la proprietà di integralità, e che $z(D'')$ può essere strettamente minore di $\min\{z(D'), z(D)\}$ se nessuno dei due insiemi di vincoli possiede la proprietà di integralità.

Esercizio 6.14 *Dimostrare le affermazioni precedenti (suggerimento: per l'ultima affermazione consultare l'esempio seguente).*

Esempio 6.2: Rilassamento e decomposizone Lagrangiana
Si consideri il seguente problema di *PLI*:

$$(P) \qquad \max\{\ 3x_1 + x_2\ :\ x_1 + x_2 \leq 3/2\ ,\ x_1 \leq 1/2\ ,\ (x_1, x_2) \in \mathbb{N}^2\ \}\ .$$

Indichiamo con $Ax \leq b$ il primo vincolo ($x_1 + x_2 \leq 3/2$) e con $Ex \leq d$ il secondo vincolo ($x_1 \leq 1/2$); si noti che sono presenti, nella formulazione del problema, anche i vincoli $x_1 \geq 0$ ed $x_2 \geq 0$, che non verranno mai rilassati, e quindi che verranno sempre implicitamente considerati come facenti parte dell'"altro" blocco di vincoli.



Il problema è illustrato geometricamente in figura (a); in particolare, nella figura sono evidenziati i vincoli lineari, l'insieme delle soluzioni ammissibili del problema (i punti $[0, 0]$ e $[0, 1]$) e l'insieme ammissibile del rilassamento continuo (zona tratteggiata). Dalla figura si deduce facilmente che la soluzione ottima del problema è $[0, 1]$, e quindi che $z(P) = 1$, mentre la soluzione ottima del rilassamento continuo ($\bar{P}$) è $[1/2, 1]$, e quindi $z(\bar{P}) = 5/2$. Consideriamo adesso il rilassamento Lagrangiano rispetto al primo vincolo ($Ax \leq b$)

$$\max\ \{\ (3 - y)x_1 + (1 - y)x_2\ :\ x_1 \leq 1/2\ ,\ (x_1, x_2) \in \mathbb{N}^2\ \} + (3/2)y\ \ ,$$

ed il corrispondente duale Lagrangiano. Dal Teorema 6.2 abbiamo che il duale Lagrangiano fornisce la stessa valutazione superiore del rilassamento convessificato, la cui regione ammissibile è evidenziata in Figura 6.3(a). Infatti, l'insieme ammissibile X del rilassamento Lagrangiano contiene tutti i punti $[0, x_2]$ con $x_2 \in \mathbb{N}$, e quindi $conv(X)$ è il semiasse positivo di x_2 (si veda la figura (b) sopra); la sua intersezione con $Ax \leq b$ restituisce il segmento di estremi $[0, 0]$ e $[0, 3/2]$. La soluzione ottima del rilassamento convessificato è quindi il punto $[0, 3/2]$; ci attendiamo pertanto che sia $z(D) = 3/2$. Possiamo verificare che ciò sia vero disegnando la *funzione Lagrangiana* $\varphi(y) = z(P_y)$ per tutti i valori di $y \geq 0$; a tale scopo, in Figura 6.3(b) vengono mostrati i vettori dei costi Lagrangiani corrispondenti ai valori $y = 0$, $y = 1$, $y = 2$, $y = 3$, ed il limite del vettore dei costi Lagrangiani per $y \to \infty$. Si ha che:

- il rilassamento Lagrangiano è superiormente illimitato ($z(P_y) = +\infty$) per $y < 1$;
- tutte le soluzioni ammissibili del rilassamento Lagrangiano sono ottime per $y = 1$, avendo tutte costo Lagrangiano $3/2$;
- $[0, 0]$ è l'unica soluzione ottima del rilassamento Lagrangiano per $y > 1$, con costo Lagrangiano $(3/2)y$.

La funzione Lagrangiana è quindi quella rappresentata in Figura 6.3(c) (nella zona tratteggiata la funzione ha valore $+\infty$), ed ha minimo per $y = 1$ con $\varphi(1) = 3/2 = z(D)$.

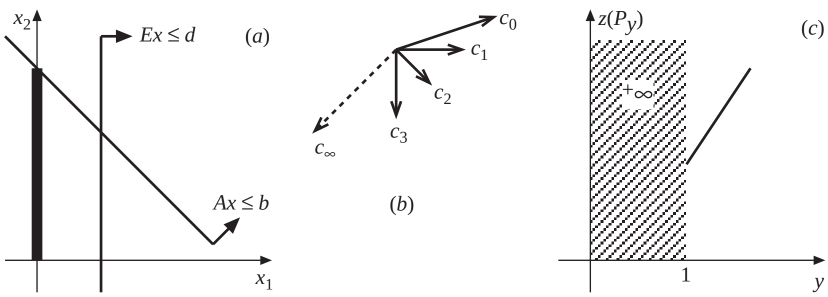


Figura 6.3: Un esempio di rilassamento Lagrangiano (2)

Consideriamo ora il rilassamento Lagrangiano rispetto al secondo vincolo ($Ex \leq d$)

$$\max\ \{\ (3 - y)x_1 + x_2\ :\ x_1 + x_2 \leq 3/2\ ,\ (x_1, x_2) \in \mathbb{N}^2\ \} + (1/2)y\ ,$$

ed il corrispondente duale Lagrangiano. L'insieme ammissibile X' del rilassamento Lagrangiano contiene i punti $[0, 0]$, $[0, 1]$ e $[1, 0]$, e quindi $conv(X')$ è il triangolo avente quei punti come vertici; la sua intersezione con $Ex \leq d$ restituisce la regione ammissibile del corrispondente rilassamento convessificato, mostrata in Figura 6.4(a) (zona in doppio tratteggio). La soluzione ottima del rilassamento convessificato è quindi il punto $[1/2, 1/2]$; ci attendiamo pertanto che sia $z(D) = 2$. Possiamo verificare che ciò sia vero disegnando la funzione Lagrangiana $\varphi(y)$ per tutti i valori di $y \geq 0$; a tale scopo, in Figura 6.4(b) vengono mostrati i vettori dei costi Lagrangiani corrispondenti ai valori $y = 0$, $y = 2$, $y = 3$, $y = 4$, ed il limite del vettore dei costi Lagrangiani per $y \to \infty$. Si ha che:

- per $0 \leq y \leq 2$, il punto $[1, 0]$ è soluzione ottima del rilassamento Lagrangiano (unica se $y < 2$), con costo Lagrangiano $3 - y/2$;
- per $y \geq 2$, il punto $[0, 1]$ è soluzione ottima del rilassamento Lagrangiano (unica se $y > 2$), con costo Lagrangiano $2 + y/2$.

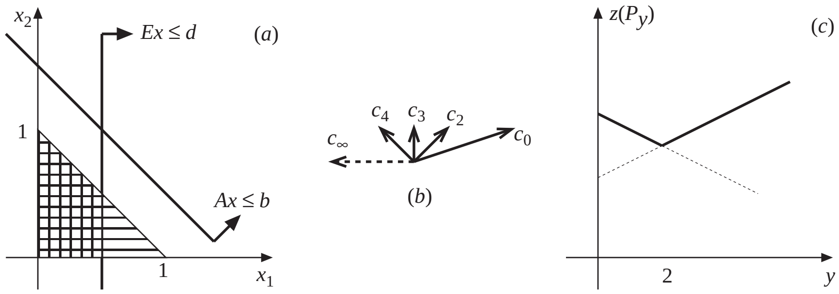


Figura 6.4: Un esempio di rilassamento Lagrangiano (3)

La funzione Lagrangiana è quindi quella rappresentata in Figura 6.4(c), ed ha minimo per $y = 2$ con $\varphi(2) = 2 = z(D')$. Consideriamo infine la decomposizione Lagrangiana di (P) corrispondente ad entrambi i blocchi $Ax \le b$ ed $Ex \le d$, ossia

$$\begin{array}{c} \max \{\ (3/2 - y_1)x_1 + (1/2 - y_2)x_2\ :\ x_1 \le 1/2\ ,\ (x_1, x_2) \in \mathbb{N}^2\ \} \\ + \\ \max \{\ (3/2 + y_1)x_1 + (1/2 + y_2)x_2\ :\ x_1 + x_2 \le 3/2\ ,\ (x_1, x_2) \in \mathbb{N}^2\ \} \end{array} .$$

L'insieme ammissibile del corrispondente rilassamento convessificato è l'intersezione tra $conv(X)$ e $conv(X')$, ossia il segmento di estremi $[0, 0]$ e $[0, 1]$; pertanto la sua soluzione ottima è $(0, 1)$ e si ha $z(D'') = z(P) = 1$. Per verificare che ciò sia vero si consideri il vettore di moltiplicatori Lagrangiani $[\bar{y}_1, \bar{y}_2] = [-1/2, 1/2]$. In corrispondenza a tali moltiplicatori, il primo problema della decomposizione Lagrangiana ha soluzione ottima $[0, 0]$ ed il secondo ha soluzione ottima $[1, 0]$, da cui $\varphi(\bar{y}) = 1$. Poichè $1 \le z(P) \le z(D'') \le \varphi(\bar{y}) = 1$, $\bar{y}$ è una soluzione ottima del corrispondente duale Lagrangiano, che ha effettivamente gap nullo.

Esercizio 6.15 *Si caratterizzi la funzione Lagrangiana della decomposizione Lagrangiana nell'ultimo caso dell'esempio precedente, disegnandone le curve di livello.*

Esercizio 6.16 *Costruire un esempio di problema* PLI *con due variabili e due vincoli in cui la decomposizione Lagrangiana fornisca una valutazione superiore non esatta (con gap non nullo) del valore ottimo della funzione obiettivo.*

6.3.2 Algoritmi per il rilassamento Lagrangiano

Per semplificare la trattazione assumeremo temporaneamente che l'insieme X sia *compatto*; nel seguito indicheremo poi come l'assunzione possa essere eliminata al costo di complicare leggermente la notazione e gli algoritmi. La conseguenza di questa assunzione è che la funzione Lagrangiana

$$\varphi(y) = z(P_y) = \max\{\ cx + y(b - Ax) : x \in X\ \}$$

è finita ovunque. Essendo il massimo di (un numero finito di) funzioni lineari, φ è convessa (sarebbe concava se (P) fosse un problema di minimo). Dato che X è un insieme discreto, come abbiamo visto negli esempi, quando $m = 1$ φ è una funzione lineare a tratti; in generale φ è *poliedrale*. Infatti, il suo *epigrafo*

$$Epi(\varphi) = \{\ (z, y)\ :\ z \ge \varphi(y)\ \} = \{\ (z, y)\ :\ z \ge cx + y(b - Ax)\ ,\ x \in X\ \}$$

è un poliedro; ad ogni elemento x di X che risolve (P_y) per un qualche y è associato un vincolo lineare che definisce una delle facce del poliedro. È quindi possibile riscrivere (D) come un problema di PL; questo non dovrebbe stupire, in quanto (D) è il duale lineare di $(\tilde{P})$. Infatti, un modo equivalente di formulare (D) è

$$(D) \qquad \min\{\ z\ :\ z \ge cx + y(b - Ax)\ ,\ x \in X\ ,\ y \ge 0\ \}\ . \tag{6.20}$$

In (6.20), ciascun elemento di X definisce un vincolo del problema, ossia possibilmente definisce una faccia della regione ammissibile, che altro non è che $Epi(\varphi)$. Il fatto che sia possibile scrivere (D) come un problema di PL non implica che (D) sia di "facile" risoluzione: il numero di vincoli del problema, corrispondente al numero di elementi di X, può essere enorme. È però vero che non tutti i vincoli di (D) sono necessari per la determinazione della soluzione ottima; al limite, sarebbero sufficienti gli $m+1$

vincoli corrispondenti ad una base ottima del problema. Ciò suggerisce un approccio di *generazione di vincoli*, in cui ad ogni iterazione si risolve il *Problema Master* (Duale)

$$(D_{\mathcal{B}}) \qquad \min\{\ z\ :\ z \geq cx + y(b - Ax)\ ,\ x \in \mathcal{B}\ ,\ y \geq 0\ \}\ . \tag{6.21}$$

ove $\mathcal{B} \subset X$ è un "piccolo" sottoinsieme delle soluzioni ammissibili di (P_y). Ciò corrisponde a risolvere

$$\min\{\ \varphi_{\mathcal{B}}(y) = \max\{\ cx + y(b - Ax) : x \in \mathcal{B}\ \}\ \}\ ,$$

ossia a minimizzare la funzione convessa poliedrale $\varphi_{\mathcal{B}}$, tale che $\varphi_{\mathcal{B}}(y) \leq \varphi(y)\ \forall y$, al posto di φ; $\varphi_{\mathcal{B}}$ è detta *modello* di φ. La soluzione ottima (z^*, y^*) di $(D_{\mathcal{B}})$, dove $z^* = \varphi_{\mathcal{B}}(y^*)$, può quindi essere usata per generare un ulteriore vincolo, se necessario, semplicemente risolvendo il rilassamento Lagrangiano (P_{y^*}), ossia calcolando $\varphi(y^*)$; in questo contesto (P_{y^*}) viene detto *problema di separazione*. Se infatti si ha $z^* < \varphi(y^*)$, allora una qualsiasi soluzione ottima $\bar{x}$ di (P_{y^*}) fornisce un vincolo di (D) violato dalla soluzione corrente (z^*, y^*), che può quindi essere aggiunto a $\mathcal{B}$. Altrimenti, ed è facile verificare che in questo caso risulta $z^* = \varphi(y^*)$, (z^*, y^*) rispetta tutti i vincoli in (D), anche quelli non esplicitamente rappresentati in $\mathcal{B}$, e quindi è ottima per (D).

```
⟨ inizializza B ⟩
do
   (z*, y*) = argmin{ z : z ≥ cx + y(b − Ax) , x ∈ B , y ≥ 0 };   /* (D_B) */
   x̄ = argmax{ (c − y*A)x : x ∈ X };                             /* (P_y*) */
   φ(y*) = cx̄ + y*(b − Ax̄); B = B ∪ { x̄ };
while(z* < φ(y*));
```

Figura 6.5: L'algoritmo dei piani di taglio

L'*algoritmo dei piani di taglio*, sintetizzato in Figura 6.5, determina quindi ad ogni passo una valutazione inferiore ed una superiore di $z(D)$, in quanto $z^* \leq z(D) \leq \varphi(y^*)$, e termina in un numero finito di passi (al limite $\mathcal{B} = X$) quando le due coincidono. Occorre solamente assicurarsi che l'insieme di vincoli $\mathcal{B}$ determinato dalla fase di inizializzazione sia sufficiente ad assicurare che $(D_{\mathcal{B}})$ abbia soluzione ottima finita; un modo in cui questo può essere ottenuto è ponendo $\mathcal{B} = \{\hat{x}\}$, dove $\hat{x}$ è una soluzione ammissibile per (P) tale che $A\hat{x} = b$, il che corrisponde ad inserire in $(D_{\mathcal{B}})$ il vincolo $z \geq c\hat{x}$.
L'algoritmo dei piani di taglio può essere "rivisitato" in notazione primale, nel qual caso prende il nome di *metodo di decomposizione di Dantzig-Wolfe*. Per questo occorre notare che $(D_{\mathcal{B}})$ ha un duale (lineare), il *Problema Master Primale*

$$(P_{\mathcal{B}}) \qquad \max\left\{\ c\left(\textstyle\sum_{x\in\mathcal{B}} x\theta_x\right)\ :\ A\left(\textstyle\sum_{x\in\mathcal{B}} x\theta_x\right) \leq b\ \ ,\ \ \theta \in \Theta\ \right\} \tag{6.22}$$

dove $\Theta = \{\ \theta \geq 0 : \sum_{x\in\mathcal{B}} \theta_x = 1\ \}$ è il simplesso unitario di dimensione opportuna. $(P_{\mathcal{B}})$ ha una variabile per ogni riga di $(D_{\mathcal{B}})$, ossia per ciascun elemento di $\mathcal{B}$. È interessante notare che questa "forma esplicita" del problema è equivalente alla "forma implicita"

$$(P_{\mathcal{B}}) \qquad \max\{\ cx\ :\ Ax \leq b\ \ ,\ \ x \in X_{\mathcal{B}} = conv(\mathcal{B})\ \}\ \ . \tag{6.23}$$

Questo chiarisce la relazione tra l'algoritmo dei piani di taglio e $(\tilde{P})$; infatti, (6.22) con $\mathcal{B} = X$ è una formulazione di $(\tilde{P})$, in cui sono esplicitamente rappresentati i moltiplicatori convessi θ. In particolare, (6.22) con $\mathcal{B} = X$ è il duale lineare di (6.20), come anticipato dal Teorema 6.2; solamente, in questo caso $conv(X)$ è espresso mediante una rappresentazione *per punti*, piuttosto che mediante la più usuale rappresentazione *per facce* utilizzata nel teorema.
L'algoritmo dei piani di taglio può quindi essere "rivisitato" dal punto di vista primale, notando che la "forma esplicita" di $(\tilde{P})$ è un problema di *PL* con "molte" colonne, una per ciascun elemento di X. Questo suggerisce un approccio di *generazione di colonne* nel quale si risolve la restrizione di $(\tilde{P})$ rispetto al sottoinsieme di colonne $\mathcal{B}$, ottenendo una soluzione ottima θ^* a cui corrisponde la soluzione $x^* = \sum_{x\in\mathcal{B}} x\theta^*_x$ ammissibile per $(\tilde{P})$ (la soluzione ottima della "forma implicita" di $(P_{\mathcal{B}})$).

Inoltre, dalla soluzione del Problema Master primale si ottiene il vettore di variabili duali ottime y^* dei vincoli $Ax \leq b$, le quali determinano il *costo ridotto* $(c - yA)x$ (si veda il paragrafo 6.1.2.2) della variabile θ_x, ossia della colonna corrispondente. Dai vincoli del Problema Master duale si ha che $z^* - y^*b \geq (c - y^*A)x$, e le condizioni degli scarti complementari garantiscono che si abbia uguaglainza per ogni $x \in \mathcal{B}$ tale che $\theta_x > 0$. Si vuole quindi determinare se esiste oppure no una colonna $x \in X \setminus \mathcal{B}$ il cui costo ridotto sia maggiore di $z^* - y^*b$; questo viene fatto risolvendo (P_{y^*}), che determina la colonna $\bar{x}$ di costo ridotto massimo, e che, in questo contesto, viene detto *problema di pricing*. Se il costo ridotto massimo tra le colonne di X è maggiore di $z^* - y^*b$, allora è stata generata una colonna "promettente", che può essere inserita in $\mathcal{B}$ per migliorare la soluzione x^*; altrimenti x^* è ottima per $(\tilde{P})$. Questi passi sono *esattamente* quelli effettuati dall'algoritmo in Figura 6.5, in cui però x^* non è stata esplicitata. Quindi l'algoritmo dei piani di taglio fornisce, al termine, una soluzione x^* ottima per $(\tilde{P})$.

Esercizio 6.17 *Nell'algoritmo dei piani di taglio (metodo di decomposizione di Dantzig-Wolfe) vengono risolti una sequenza di problemi di* PL *correlati tra loro. Facendo riferimento al paragrafo 2.3.3, si discuta quale algoritmo del Simplesso sia più conveniente utilizzare per la loro risoluzione.*

Esempio 6.3: L'algoritmo dei piani di taglio
Si consideri il problema di *PLI*

$$(P) \qquad \max\{\, x_1 + x_2 \;:\; x_2 \leq 1/2 \;,\; (x_1, x_2) \in \{0,1\}^2 \,\}$$

ed il suo rilassamento Lagrangiano rispetto al vincolo $x_2 \leq 1/2$

$$(P_y) \qquad \max\{\, x_1 + (1-y)x_2 \;:\; (x_1, x_2) \in \{0,1\}^2 \,\} + (1/2)y \;.$$

Applichiamo l'algoritmo dei piani di taglio a partire da $\mathcal{B} = \{\, [0,0] \,,\, [0,1] \,\}$. I Problemi Master duale e primale sono quindi

$$(D_{\mathcal{B}}) \quad \left\{ \begin{array}{llll} \min & z & & \\ & z & \geq & 0 + y(1/2) \\ & z & \geq & 1 - y(1/2) \\ & & & y \geq 0 \end{array} \right. \qquad (P_{\mathcal{B}}) \quad \left\{ \begin{array}{llccc} \max & & \theta_{[0,1]} & & \\ & & \theta_{[0,1]} & \leq & 1/2 \\ & \theta_{[0,0]} & +\theta_{[0,1]} & = & 1 \\ & \theta_{[0,0]} & & \geq & 0 \\ & & \theta_{[0,1]} & \geq & 0 \end{array} \right. .$$

La soluzione ottima di $(D_{\mathcal{B}})$ è $y^* = 1$, $z^* = \varphi_{\mathcal{B}}(y^*) = 1/2$, a cui corrisponde un costo ridotto $c - y^*A = [1,0]$. La soluzione ottima di $(P_{\mathcal{B}})$ è $\theta^*_{[0,0]} = \theta^*_{[0,1]} = 1/2$, a cui corrisponde $x^* = [\,0 \,,\, 1/2\,]$. I punti $[\,1 \,,\, 0\,]$ e $[\,1 \,,\, 1\,]$ sono entrambi soluzioni ottime di (P_{y^*}); supponiamo che tra i due venga restituito $\bar{x} = [1,0]$. Siccome si ha $\varphi(y^*) = 3/2 > z^*$, l'algoritmo prosegue. Questa situazione è illustrata in Figura 6.6(a) e (b).

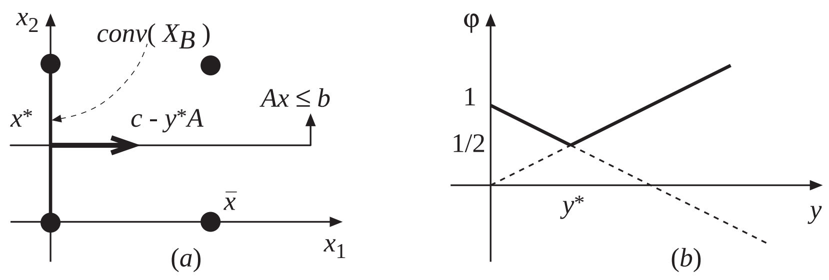


Figura 6.6: Applicazione dell'algoritmo dei piani di taglio (1)

Alla seconda iterazione si ha pertanto $\mathcal{B} = \{\, [0,0] \,,\, [0,1] \,,\, [1,0] \,\}$. I Problemi Master duale e primale sono quindi

$$(D_{\mathcal{B}}) \quad \left\{ \begin{array}{llll} \min & z & & \\ & z & \geq & 0 + y(1/2) \\ & z & \geq & 1 - y(1/2) \\ & z & \geq & 1 + y(1/2) \\ & & & y \geq 0 \end{array} \right. \qquad (P_{\mathcal{B}}) \quad \left\{ \begin{array}{lllccc} \max & & \theta_{[0,1]} & +\theta_{[1,0]} & & \\ & & \theta_{[0,1]} & & \leq & 1/2 \\ & \theta_{[0,0]} & +\theta_{[0,1]} & +\theta_{[1,0]} & = & 1 \\ & \theta_{[0,0]} & & & \geq & 0 \\ & & \theta_{[0,1]} & & \geq & 0 \\ & & & \theta_{[1,0]} & \geq & 0 \end{array} \right. .$$

La soluzione ottima di $(D_{\mathcal{B}})$ è $y^* = 0$, $z^* = \varphi_{\mathcal{B}}(y^*) = 1$, a cui corrisponde un costo ridotto $c - y^*A = (1,1)$. $(P_{\mathcal{B}})$ ha soluzioni ottime multiple, tra cui quelle estreme sono $\theta^*_{[0,0]} = 0$, $\theta^*_{[0,1]} = \theta^*_{[1,0]} = 1/2$ e $\theta^*_{[0,0]} = \theta^*_{[0,1]} = 0$, $\theta^*_{[1,0]} = 1$; queste soluzioni corrispondono a tutti i punti del segmento di estremi $[1/2, 1/2]$ e $[1,0]$. Il punto $\bar{x} = [1,1]$ è la soluzione ottima di (P_{y^*}). Siccome si ha $\varphi(y^*) = 2 > z^*$, l'algoritmo prosegue. Questa situazione è illustrata in Figura 6.7(a) e (b).

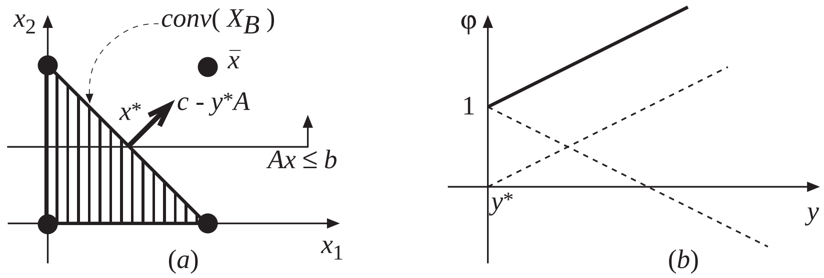


Figura 6.7: Applicazione dell'algoritmo dei piani di taglio (2)

Alla terza iterazione si ha dunque $\mathcal{B} = \{\, [0,0]\,,\, [0,1]\,,\, [1,0]\,,\, [1,1]\,\} = X$. I Problemi Master duale (equivalente a (D)) e primale sono quindi

$$
(D_\mathcal{B}) \quad \left\{ \begin{array}{lrcl}
\min & z & & \\
 & z & \geq & 0 + y(1/2) \\
 & z & \geq & 1 - y(1/2) \\
 & z & \geq & 1 + y(1/2) \\
 & z & \geq & 2 - y(1/2) \\
 & & & y \geq 0
\end{array} \right.
\qquad
(P_\mathcal{B}) \quad \left\{ \begin{array}{lccccccc}
\max & & \theta_{[0,1]} & +\theta_{[1,0]} & +2\theta_{[1,1]} & & \\
 & & \theta_{[0,1]} & & +\theta_{[1,1]} & \leq & 1/2 \\
 & \theta_{[0,0]} & +\theta_{[0,1]} & +\theta_{[1,0]} & +\theta_{[1,1]} & = & 1 \\
 & \theta_{[0,0]} & & & & \geq & 0 \\
 & & \theta_{[0,1]} & & & \geq & 0 \\
 & & & \theta_{[1,0]} & & \geq & 0 \\
 & & & & \theta_{[1,1]} & \geq & 0
\end{array} \right. .
$$

La soluzione ottima di $(D_\mathcal{B})$ è $y^* = 1$, $z^* = \varphi_\mathcal{B}(y^*) = 3/2$, a cui corrisponde un costo ridotto $c - y^*A = [\,1\,,\,0\,]$. La soluzione ottima di $(P_\mathcal{B})$ è $\theta^*_{[\,0\,,\,0\,]} = \theta^*_{[\,0\,,\,1\,]} = 0$, $\theta^*_{[\,1\,,\,0\,]} = \theta^*_{[\,1\,,\,1\,]} = 1/2$, a cui corrisponde $x^* = [\,1\,,\,1/2\,]$. I punti $[\,1\,,\,0\,]$ e $[\,1\,,\,1\,]$ sono entrambi soluzioni ottime di (P_{y^*}), e $\varphi(y^*) = 3/2 = z^*$: l'algoritmo quindi termina, avendo determinato la soluzione ottima $y^* = 1$ di (D) e la soluzione ottima $x^* = [\,1\,,\,1/2\,]$ di $(\tilde{P})$. Questa situazione è illustrata in Figura 6.8(a) e (b).

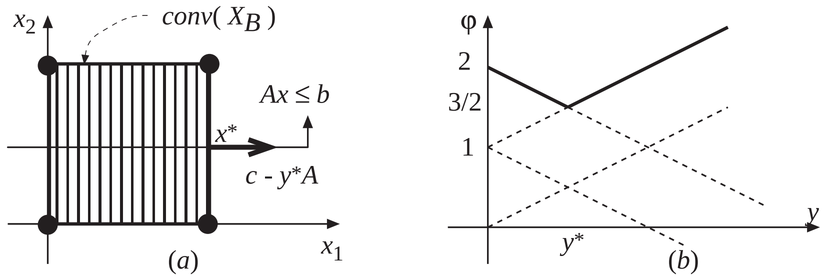


Figura 6.8: Applicazione dell'algoritmo dei piani di taglio (3)

Esercizio 6.18 *Si discuta come sia possibile modificare l'algoritmo dei piani di taglio in modo che non sia necessario partire con un insieme di vincoli $\mathcal{B}$ sufficientemente grande da garantire che $(D_\mathcal{B})$ abbia ottimo finito (suggerimento: prendendo spunto dal Primale Ausiliario per il Simplesso Primale introdotto nel paragrafo 2.3.1, si modifichi $(P_\mathcal{B})$ in modo tale che $(D_\mathcal{B})$ abbia sicuramente ottimo finito).*

L'analisi precedente mostra che l'algoritmo dei piani di taglio risolve *contemporaneamente* (D) e $(\tilde{P})$, e suggerisce le seguenti considerazioni:

- Dal punto di vista primale, il Problema Master utilizza l'*approssimzaione interna* $X_\mathcal{B}$ di $conv(X)$, che viene espansa aggiungendo i punti generati dal problema di pricing (P_{y^*}) finché essa non contiene l'ottimo di $(\tilde{P})$; dal punto di vista duale, il Problema Master utilizza l'*approssimazione esterna* $\varphi_\mathcal{B}$ di φ, e quindi della regione ammissibile di (D), che viene raffinata inserendo i vincoli generati dal problema di separazione finché non risulta "esatta" nell'ottimo di (D).

- La struttura del problema viene utilizzata per *generare efficientemente punti* (estremi) di $conv(X)$; in altri termini, questo procedimento è particolarmente attraente nel caso in cui ottenere una rappresentazione esplicita di $conv(X)$ sia significativamente più difficile che ottimizzare su X. In effetti, come mostrato nel paragrafo precedente, la soluzione del duale Lagrangiano di un problema di *PLI* è particolarmente attraente nel caso in cui i vincoli $Ex \leq d$ non rilassati *non* possiedano la proprietà di integralità, ossia non rappresentino esattamente $conv(X)$, ma si disponga comunque di un algoritmo "ragionevolmente efficiente" per risolvere il rilassamento Lagrangiano.

L'algoritmo dei piani di taglio può essere facilmente esteso al caso in cui X non sia compatto. Il poliedro $conv(X)$ può essere in generale espresso come la somma di un politopo P e di un cono finitamente generato C (si veda il paragrafo 2.1.1), detto *cono delle direzioni* di $conv(X)$; quando X è compatto si ha $C = \{ 0 \}$. Ogni vettore $\nu \in C$ costituisce una direzione ammissibile illimitata per $conv(X)$; ne consegue che se per un qualche y si ha $(c - yA)\nu > 0$, ossia la direzione è di crescita, allora il rilassamento Lagrangiano (P_y) è superiormente illimitato, ossia $\varphi(y) = +\infty$. In altri termini, a qualunque vettore $\nu \in C$ è associato un vincolo lineare $(c - yA)\nu \leq 0$ che è rispettato da tutti i punti y in cui $\varphi(y) < +\infty$. L'algoritmo dei piani di taglio si estende dunque al caso in cui X non sia compatto semplicemente mantenendo l'insieme $\mathcal{B} = \mathcal{B}^0 \cup \mathcal{B}^1$ in cui $\mathcal{B}^0 \subset C$ e $\mathcal{B}^1 \subset X$. Il Master Problem Primale e Duale divengono rispettivamente

$$(P_{\mathcal{B}}) \quad \max \left\{ \begin{array}{l} c\left(\sum_{x \in \mathcal{B}^1} x\theta_x + \sum_{\nu \in \mathcal{B}^0} \nu\theta_\nu \right) \\ A\left(\sum_{x \in \mathcal{B}^1} x\theta_x + \sum_{\nu \in \mathcal{B}^0} \nu\theta_\nu \right) \leq b \\ \sum_{x \in \mathcal{B}^1} \theta_x = 1 \quad , \quad \theta \geq 0 \end{array} \right. \qquad (D_{\mathcal{B}}) \quad \min \left\{ \begin{array}{ll} yb + z & \\ z \geq (c - yA)x & x \in \mathcal{B}^1 \\ 0 \geq (c - yA)\nu & \nu \in \mathcal{B}^0 \end{array} \right. .$$

Un modo equivalente di riscrivere i problemi è

$$(P_{\mathcal{B}}) \quad \max\left\{ \, cx \; : \; Ax \leq b \, , \; x \in conv(\mathcal{B}^1) + cono(\mathcal{B}^0) \, \right\} \qquad (D_{\mathcal{B}}) \quad \min\left\{ \, \varphi_{\mathcal{B}^1}(y) \; : \; y \in Y_{\mathcal{B}} \, \right\} \; ,$$

dove $Y_{\mathcal{B}} = \{ \; y : (c - yA)\nu \leq 0 \; , \; \nu \in \mathcal{B}^0 \; \}$ è un'approssimazione esterna dell'insieme dei punti y in cui $\varphi(y) < +\infty$. Ad ogni iterazione dell'algoritmo, la soluzione del rilassamento Lagrangiano riporta o una soluzione ottima $\bar{x} \in X$, che viene quindi aggiunta a $\mathcal{B}^1$, oppure una direzione di decrescita illimitata $\bar{\nu} \in C$, che viene quindi aggiunta a $\mathcal{B}^0$. Si noti che nel secondo caso si ha $\varphi(y^*) = +\infty$, e di conseguenza in questo tipo di iterazioni non si ha a disposizione una nuova valutazione superiore su $z(D)$.

Una diversa estensione dell'algoritmo dei piani di taglio si ha nel caso, molto frequente nelle applicazioni, in cui X è il prodotto cartesiano di k di insiemi (che assumiamo temporaneamente compatti) $X = X_1 \times X_2 \times \ldots \times X_k$, ossia il rilassamento Lagrangiano si decompone in k problemi indipendenti ed una soluzione ottima $\bar{x} = [\, \bar{x}^1 \, , \, \bar{x}^2 \, , \, \ldots \, , \, \bar{x}^k \,]$ del rilassamento Lagrangiano si ottiene semplicemente giustapponendo le soluzioni ottime dei k sottoproblemi; questo è ad esempio il caso di (UMMCF). In altri termini, la funzione Lagrangiana si decompone nella somma di k funzioni

$$\varphi(y) = yb + \sum_{h \in K} \left(\varphi^h(y) = \min\{ \, (c^h - yA^h)x^h \; : \; x^h \in X^h \, \} \right)$$

($K = \{ 1 \ldots , k \}$, e si può pensare che la funzione lineare yb sia la $(k+1)$-esima funzione). Si può allora risolvere ad ogni iterazione, al posto di (6.22), il *Problema Master primale e duale disaggregato*

$$\begin{array}{ll} (P_{\mathcal{B}}) & \max\left\{ \; \sum_{h \in K} c^h \sum_{x^h \in \mathcal{B}^h} x^h \theta^h_x \; : \; \sum_{h \in K} A^h \sum_{x^h \in \mathcal{B}^h} x^h \theta^h_x \leq b \; , \; \theta^h \in \Theta \quad h \in K \; \right\} \\ (D_{\mathcal{B}}) & \min\left\{ \; yb + \sum_{h \in K} z^h \; : \; z^h \leq (c^h - yA^h)x^h \quad x^h \in \mathcal{B}^h \quad , \quad h \in K \; \right\} \end{array} \qquad (6.24)$$

in cui tutte le componenti h-esime delle soluzioni generate sono immagazzinate nell'insieme $\mathcal{B}^h$ ed hanno un moltiplicatore θ^h_x indipendente dalle altre componenti della stessa soluzione. I due problemi possono essere riscritti come

$$(P_{\mathcal{B}}) \quad \max \left\{ \begin{array}{l} \sum_{h \in K} c^h x^h \\ \sum_{h \in K} A^h x^h \leq b \\ x^h \in conv(\mathcal{B}^h) \;\; h \in K \end{array} \right. \qquad (D_{\mathcal{B}}) \quad \min\left\{ \; yb + \sum_{h \in K} \varphi_{\mathcal{B}^h}(y) \; \right\} \qquad (6.25)$$

in cui $\varphi_{\mathcal{B}^h}$ è il modello dell'h-esima componente φ^h di φ. È facile verificare come, dato uno stesso insieme di soluzioni $\mathcal{B} \subset X$, l'insieme ammissibile di (6.25) contenga strettamente quello di (6.23); infatti, (6.22) è la restrizione di (6.24) in cui tutte le componenti x^h corrispondenti ad una stessa soluzione x sono forzate ad avere lo stesso moltiplicatore. In altri termini, $conv(\mathcal{B}^1) \times conv(\mathcal{B}^2) \times \ldots \times conv(\mathcal{B}^k)$ è una migliore approssimazione di $conv(X)$ rispetto a $conv(\mathcal{B})$; alternativamente, si può dire che la somma dei k modelli $\varphi_{\mathcal{B}^h}$ è una migliore approssimazione di φ rispetto al modello "aggregato" $\varphi_{\mathcal{B}}$. I Problemi Master disaggregati hanno dimensione maggiore di un fattore k rispetto a quelli aggregati (a parità di informazione raccolta), e sono quindi più costosi da risolvere; il miglior uso dell'informazione disponibile determina però spesso una convergenza sensibilmente più rapida dell'algoritmo dei piani di taglio (un minor numero di iterazioni), che può abbondantemente controbilanciare il maggior costo della soluzione del Problema Master. Nel caso poi in cui alcuni degli insiemi X^h non siano compatti l'algoritmo può essere esteso analogamente a quanto visto in precedenza.

Sono stati proposti molti altri algoritmi per la soluzione del duale Lagrangiano; alcuni sono basati sull'algoritmo dei piani di taglio e cercano di migliorarne le prestazioni evitando alcune delle sue limitazioni ("instabilità" e necessità di un opportuno insieme iniziale $\mathcal{B}$), mentre altri sono basati su idee diverse. Per una descrizione approfondita di tali algoritmi si rimanda alla letteratura citata.

6.3.3 Informazione generata dal rilassamento Lagrangiano

Risolvere un duale Lagrangiano, ad esempio utilizzando l'algoritmo dei piani di taglio presentato nel paragrafo precedente, fornisce, oltre alla valutazione superiore, una quantità di informazione sul problema equivalente a quella prodotta da un rilassamento continuo (cf. 6.1.2). Infatti, al termine dell'algoritmo si dispone sia di una soluzione ottima y^* di (D) che di una soluzione ottima x^* di $(\tilde{P})$; quest'ultima è una soluzione frazionaria del tutto analoga a quella prodotta dal rilassamento continuo—le due coincidono se il rilassamento Lagrangiano ha la proprietà di integralità. In effetti, il rilassamento Lagrangiano produce un'informazione "più ricca" di quella fornita dal rilassamento continuo: non solo una soluzione continua x^*, ma un insieme di soluzioni $\mathcal{B} \subset X$ ed i relativi moltiplicatori convessi θ_x che producono x^*. Discuteremo adesso brevemente come sia possibile sfruttare questa informazione per la soluzione di un problema di *OC*.

6.3.3.1 Uso dell'informazione primale

La soluzione continua x^* di $(\tilde{P})$ può chiaramente essere usata all'interno di tecniche di arrotondamento, quali quelle viste nel Paragrafo 6.1.2.1, esattamente allo stesso modo in cui viene utilizzata la soluzione ottima di un rilassamento continuo (lo stesso dicasi per le regole di separazione negli algoritmi enumerativi, che saranno discusse nel prossimo capitolo). Poichè, nel caso in cui il rilassamento Lagrangiano non abbia la proprietà di integralità, $(\tilde{P})$ è un rilassamento "più accurato" di quello continuo, ci si può aspettare che le soluzioni euristiche costruite a partire da x^* siano, in generale, di qualità almeno comparabile a quelle costruite a partire dalla soluzione ottima del rilassamento continuo.

In più, il processo di soluzione del duale Lagrangiano può essere sfruttato per produrre soluzioni ammissibili di (P). Si parla in tal caso di *euristiche Lagrangiane*. In molti casi si utilizza la soluzione $\bar{x}$ del rilassamento Lagrangiano all'iterazione corrente, intera ma che viola i vincoli $Ax \leq b$, e la si rende ammissibile mediante una procedura euristica, spesso di tipo greedy. Alla soluzione ammissibile così ottenuta possono poi essere applicate euristiche di raffinamento, tipicamente di ricerca locale, per migliorarne la qualità. Il processo iterativo per la soluzione di (D) funge quindi da "multistart" per normali euristiche, che in più possono utilizzare i costi Lagrangiani—che contengono informazione relativa ai vincoli rilassati—per guidare la costruzione della soluzione ammissibile. Le euristiche possono essere invocate ad ogni iterazione, oppure ad intervalli prestabiliti, oppure solamente quando viene prodotta una soluzione $\bar{x}$ con determinate caratteristiche. Le euristiche possono essere invocate in modo uniforme durante il processo iterativo oppure essere invocate più spesso in determinate fasi, tipicamente verso il termine dell'esecuzione dell'algoritmo, in quanto i moltiplicatori Lagrangiani y sono di "migliore qualità". Al limite è possibile invocare le euristiche solamente in corrispondenza del vettore di moltiplicatori Lagrangiani ottimo, per quanto ciò non garantisca di determinare la migliore

tra le soluzioni ottenibili. Le soluzioni ammissibili prodotte dall'euristica Lagrangiana possono poi essere usate, ad esempio, come popolazione di partenza per algoritmi di tipo genetico.
Si consideri ad esempio (TSP); per questo problema abbiamo a disposizione un rilassamento basato su (MST) e l'euristica "twice around MST" che costruisce soluzioni ammissibili a partire da un albero di copertura. È quindi facile trasformare l'euristica "twice around MST" in un'euristica Lagrangiana semplicemente applicandola all'albero di copertura prodotto dal rilassamento Lagrangiano. Il ciclo Hamiltoniano così ottenuto può poi essere utilizzato come punto di partenza per un'euristica di ricerca locale come quelle discusse nel capitolo precedente.
Si consideri ora (CMST); per costruire un rilassamento Lagrangiano del problema possiamo operare come nel paragrafo 6.1.1.2 e rilassare i vincoli (6.4) della formulazione (6.1)–(6.5), in modo tale da ottenere un rilassamento Lagrangiano che si decompone in due sottoproblemi indipendenti. Alternativamente, possiamo considerare una diversa formulazione in cui i vincoli (6.2) sono sostituiti dai vincoli di connessione (1.6) che definiscono il problema (MST) (sulle variabili y_{ij}). In questo modo, il rilassamento Lagrangiano dei vincoli di conservazione di flusso (6.3) diviene un problema (MST) in cui i costi delle variabili x vengono "proiettati" sulle variabili y.

Esercizio 6.19 *Si discuta come sia possibile risolvere questo rilassamento Lagrangiano utilizzando una sola computazione di un albero di copertura di costo minimo.*

Il rilassamento Lagrangiano potrebbe, specialmente in corrispondenza a "buoni" moltiplicatori Lagrangiani, produrre alberi ammissibili rispetto al vincolo di peso sui sottoalberi della radice, e quindi fornire a costo nullo anche soluzioni ammissibili per il problema. Qualora ciò non accadesse si potrebbero modificare gli algoritmi di ricerca locale visti nel capitolo precedente per cercare di rendere ammissibile l'albero prodotto, ad esempio attraverso mosse di "Cut & Paste" o scambio di sottoalberi. Una volta prodotta una soluzione ammissibile sarebbe poi naturale applicarvi le normali procedure di ricerca locale per cercare di migliorare la funzione obiettivo.

Esercizio 6.20 *Si discuta come modificare le euristiche di ricerca locale per il problema (CMST) in modo tale da produrre una soluzione ammissibile a partire da un albero di copertura che viola il vincolo di peso sui sottoalberi della radice.*

Nel contesto delle euristiche Lagrangiane può risultare utile non solo la soluzione $\bar{x}$, ma l'intero insieme delle soluzioni $\mathcal{B}$ generato nel corso della soluzione del duale Lagrangiano, ed i moltiplicatori θ_x ad esse associati. In particolare, è interessante rilevare che i moltiplicatori θ_x hanno la forma di una distribuzione di probabilità sugli elementi di $\mathcal{B}$. Ciò può suggerire approcci in cui le soluzioni $x \in \mathcal{B}$ vengono combinate per costruire una soluzione ammissibile sfruttando queste indicazioni. Nel caso poi in cui X sia decomponibile e venga utilizzato un algoritmo di tipo disaggregato (cf. §6.3.2), si hanno "probabilità" diverse per componenti x^h diverse provenienti dalla stessa soluzione del rilassamento Lagrangiano, e quindi si ottiene naturalmente un effetto di "ibridazione" in cui la soluzione complessiva viene costruita sfruttando componenti provenienti da soluzioni diverse del rilassamento Lagrangiano. Nel caso di (UMMCF), ad esempio, ad ogni passo ciascun insieme $\mathcal{B}^h$ contiene cammini da o_h a d_h, e i moltiplicatori θ possono essere (arbitrariamente) interpretati come "probabilità che il cammino faccia parte di una soluzione ottima di (P)".

Esercizio 6.21 *Si propongano euristiche Lagrangiane per i problemi di* OC *presentati in questo e nei precedenti capitoli.*

6.3.3.2 Uso dell'informazione duale

La soluzione ottima y^* di (D) fornisce un'informazione sui vincoli rilassati $Ax \leq b$ del tutto analoga a quella fornita dalla soluzione duale ottima del rilassamento continuo—le due coincidono se il rilassamento Lagrangiano ha la proprietà di integralità. Vedremo nel prossimo paragrafo un possibile uso di tale informazione. Poichè, nel caso in cui il rilassamento Lagrangiano non abbia la proprietà di integralità, $(\tilde{P})$ è un rilassamento "più accurato" di quello continuo, ci si può aspettare che l'informazione

sull'"importanza" dei vincoli contenuta in y^* sia, in generale, di qualità almeno comparabile a quella contenuta nella soluzione duale ottima del rilassamento continuo.
Per quanto riguarda i costi ridotti, se il rilassamento Lagrangiano (P_y) è un problema di PL—o un suo caso particolare, come ad esempio un problema di flusso su grafo o di cammino minimo—allora i costi ridotti delle variabili in (P_{y^*}) possono essere usati, ad esempio, per il fissaggio basato sui costi ridotti (cf. §6.1.2.2), esattamente allo stesso modo in cui vengono usati i costi ridotti del rilassamento continuo. Ad esempio, se $c_i^* < 0$ è il costo ridotto di una variabile binaria x_i cha ha valore 0 nella soluzione ottima di (P_{y^*}), e si ha $z(D) + c_i^* < \underline{z}$, allora x_i ha sicuramente valore pari a 0 in qualsiasi soluzione ottima di (P), ed analogamente per il caso in cui $x_i^* = 1$ e $c_i^* > 0$.

Esercizio 6.22 *Si dimostri l'affermazione precedente.*

In effetti, nel caso del rilassamento Lagrangiano queste relazioni possono essere verificate non solamente al termine dell'algoritmo, ossia quando si conosce y^*, ma ogniqualvolta si risolva un rilassamento Lagrangiano $(P_{\bar{y}})$ in corrispondenza ad un qualsiasi vettore $\bar{y}$ di moltiplicatori Lagrangiani.

6.4 Rilassamento surrogato

Un ulteriore modo per utilizzare la struttura presente in problemi con la forma (6.6) è quello di effettuare un *rilassamento surrogato* di (P) rispetto agli m vincoli complicanti $Ax \leq b$: fissato un vettore $y \in \mathbb{R}_+^m$ di *moltiplicatori surrogati*, questo è il problema

$$(RS_y) \qquad \max\{ \, cx \; : \; (yA)x \leq (yb) \;\; , \;\; Ex \leq d \;\; , \;\; x \in \mathbb{Z}^n \, \} \;\; .$$

In questo caso, però, non è detto che il problema sia "facile": gli m vincoli complicanti $Ax \leq b$ sono stati rimpiazzati dal singolo *vincolo surrogato* $(yA)x \leq (yb)$, che potrebbe però a sua volta essere "complicante". In effetti, il rilassamento surrogato viene utilizzato molto meno frequentemente di quello Lagrangiano proprio perchè, come abbiamo spesso accennato nei paragrafi e capitoli precedenti, la semplice aggiunta di un singolo vincolo lineare spesso trasforma un problema "facile" in uno "difficile": si veda ad esempio il caso di (CSP). Comunque, poichè (RS_y) ha "meno vincoli" di (P), si può sperare che sia in qualche senso più facile da risolvere. Si pensi ad esempio al caso di un problema di Programmazione 0/1 generico (i vincoli $Ex \leq d$ sono semplicemente $x \in [0,1]^n$): in questo caso (RS_y) è un problema dello zaino. Come vedremo in seguito, il problema dello zaino è in qualche modo "più facile" di un generico problema di Programmazione 0/1. Assumiamo dunque di avere a disposizione un algoritmo "ragionevolmente efficiente" per risolvere (RS_y).

È immediato verificare che, comunque scelto y, (RS_y) è un rilassamento di (P), ossia risulta $z(RS_y) \geq z(P)$. Infatti, le due funzioni obiettivo coincidono e la regione ammissibile di (RS_y) contiene quella di (P): data una qualsiasi soluzione $\bar{x}$ ammissibile per (P), ossia tale che $A\bar{x} \leq b$, si ha chiaramente $(yA)\bar{x} \leq yb$ per ogni $y \geq 0$. Si noti che il viceversa non è vero, ossia possono esistere soluzioni che rispettano il vincolo surrogato ma non il sistema di vincoli originario $Ax \leq b$.

Esercizio 6.23 *Fornire un esempio che dimostri l'affermazione precedente.*

L'interesse del rilassamento surrogato risiede nel fatto che, in generale, fornisce valutazioni superiori non peggiori di quelle prodotte dal rilassamento Lagrangiano con lo stesso vettore di moltiplicatori y.

Teorema 6.3 $z(RS_y) \leq z(P_y) \quad \forall \, y \geq 0$.

Dimostrazione Basta notare che (P_y) può essere visto come un rilassamento Lagrangiano di (RS_y) rispetto all'unico vincolo "complicante" $(yA)x \leq yb$, con moltiplicatore Lagrangiano pari ad 1. $\diamond$

Di conseguenza, il *duale surrogato* di (P) rispetto ai vincoli $Ax \leq b$

$$(DS) \qquad \min\{ \, z(RS_y) \; : \; y \geq 0 \, \} \;\; .$$

fornisce una valutazione superiore non peggiore (non maggiore) di quella fornita dal duale Lagrangiano (D). Purtroppo, anche qualora si disponga di un algoritmo efficiente per risolvere il (RS_y), risolvere (DS) è molto più "difficile" di risolvere il duale Lagrangiano. Infatti, mentre la funzione Lagrangiana φ è convessa, la *funzione surrogata* $\phi(y) = z(RS_y)$ non lo è; addirittura è una funzione non continua. Minimizzare una funzione di questo tipo è in generale un problema $\mathcal{NP}$-arduo. Sono stati proposti alcuni algoritmi che tentano di determinare minimi locali o globali di funzioni di questo tipo, ma la loro efficienza in pratica non è comparabile con quella degli algoritmi per l'ottimizzazione di funzioni convesse, quali quelli illustrati nel paragrafo precedente.
Per ovviare a questo inconveniente si ricorre spesso ad uno "stratagemma" interessante, che sfrutta la similitudine "sintattica" tra il rilassamento Lagrangiano e quello surrogato. Si risolve cioè il duale Lagrangiano (D), e si usano i moltiplicatori Lagrangiani ottimi y^* come moltiplicatori surrogati, risolvendo un singolo rilassamento surrogato. Questo fornisce una valutazione superiore non peggiore di $z(P_{y^*}) = z(D)$, e quindi può consentire di migliorare la valutazione fornita dal duale Lagrangiano; naturalmente non si ha nessuna garanzia che y^* sia una soluzione ottima del duale surrogato. Alternativamente si possono utilizzare come moltiplicatori surrogati le variabili duali ottime dei vincoli $Ax \leq b$ nel rilassamento continuo di (P), il che può consentire di ottenere una valutazione superiore migliore di quella fornita dal solo rilassamento continuo.

Esercizio 6.24 *Si discuta sotto quali condizioni la procedura appena accennata ottiene sicuramente una valutazione superiore non peggiore di quella determinata dal duale Lagrangiano.*

Riferimenti Bibliografici

C. Lemaréchal, *Lagrangian Relaxation*, in "**Computational Combinatorial Optimization**", M. Jünger and D. Naddef eds., *Springer-Verlag*, 2001.

V. Varzirani "**Approximation Algorithms**", *Springer-Verlag*, 2001.

L. Wolsey "**Integer Programming**", *Wiley-Interscience*, 1998.

Capitolo 7

Algoritmi enumerativi

Come indicato, né gli algoritmi greedy né quelli basati sulla ricerca locale sono in grado, in molte situazioni, di garantire l'ottimalità della soluzione trovata. Nel caso in cui sia importante determinare una soluzione ottima del problema è necessario quindi ricorrere ad algoritmi diversi. Esistono più approcci per la determinazione della soluzione esatta di un problema di *OC* $\mathcal{NP}$-arduo. Tra questi, quelli di *enumerazione implicita* sono certamente i più diffusi. Questi algoritmi esplorano in modo sistematico lo spazio delle soluzioni alla ricerca di una soluzione ottima. Le valutazioni inferiori (euristiche) e superiori (rilassamenti) del valore ottimo della funzione obiettivo discussi nei capitoli precedenti, insieme ad opportune regole di dominanza, vengono sfruttate per ottenere informazioni sul problema che permettano di escludere dalla ricerca aree dello spazio delle soluzioni in cui dimostrabilmente non si trovi una soluzione ottima; queste aree si dicono quindi *visitate implicitamente* dall'algoritmo. Le modalità della ricerca fanno sì che, al termine, si abbia la garanzia dell'ottimalità della soluzione determinata. Se opportunamente implementati, utilizzando euristiche e rilassamenti efficaci ed efficienti e molti altri importanti dettagli discussi nel seguito (regole di visita, separazione e dominanza, pre-trattamento, tecniche poliedrali ...), gli algoritmi di enumerazione implicita riescono spesso a risolvere in tempi accettabili istanze di dimensioni rilevanti di problemi $\mathcal{NP}$-ardui. Comunque, anche usando le migliori tecnologie disponibili, non si può mai escludere l'eventualità di dover esaminare una frazione consistente dello spazio delle soluzioni, per cui questi algoritmi hanno in generale una complessità esponenziale. Per questo non esistono implementazioni generiche in grado di risolvere istanze di dimensione arbitraria di qualsiasi problema di *OC*. Anche se sono a tutt'oggi disponibili strumenti software in grado di risolvere in tempi brevi qualsiasi problema di *PLI* di piccole dimensioni (qualche decina di variabili), il successo di queste tecniche per problemi di scala maggiore è estremamente variabile e dipende fortemente dal problema in esame e dalle istanze specifiche. Molto spesso il processo che porta alla determinazione di soluzioni ottime di problemi di *OC* di interesse pratico passa attraverso la realizzazione di approcci ad-hoc, o quantomeno attraverso l'implementazione, all'interno di software generici, di moduli specifici per il problema e le istanze in esame. Questo giustifica l'interesse nella formazione di esperti in grado di comprendere il funzionamento di questi algoritmi ad un elevato livello di dettaglio, per intervenire su di essi sfruttando al meglio tutte le informazioni disponibili sullo specifico problema da risolvere.

7.1 Algoritmi di enumerazione implicita

In questo paragrafo descriveremo le idee base degli algoritmi di enumerazione implicita, o *Branch and Bound* (B&B), fornendo uno schema molto generale di algoritmo e discutendone le principali proprietà. Ci soffermeremo poi su alcuni importanti aspetti dell'implementazione di algoritmi di questo tipo, discutendo alcuni esempi.

Gli algoritmi di enumerazione implicita possono essere visti come un caso particolare del ben noto schema algoritmico "divide et impera", che affronta la soluzione di un problema mediante i seguenti passi:

- suddividere il problema in un certo numero di sottoproblemi "più piccoli";
- risolvere separatamente i singoli sottoproblemi, tipicamente applicando ricorsivamente lo stesso procedimento finchè la soluzione non può essere ottenuta mediante un qualche procedimento alternativo (caso base);
- combinare le soluzioni dei singoli sottoproblemi per ottenere una soluzione del problema originale.

Consideriamo un generico problema di *OC*

$$(P) \qquad \max \{ \; c(x) \; : \; x \in X \; \} \;\; ,$$

per il quale abbiamo a disposizione due procedimenti (arbitrariamente complessi) che forniscano, rispettivamente, una valutazione superiore $\bar{z}(P)$ ed una inferiore $\underline{z}(P)$ di $z(P)$. Dalla discussione del Paragrafo 4.3 ricordiamo che "risolvere" (P) significa certificare $\underline{z}(P) = \bar{z}(P)$. Se, come spesso accade, alla valutazione inferiore è anche associata una soluzione $\bar{x} \in X$ (tale che $c(\bar{x}) = \underline{z}(P)$), allora si è anche ottenuta una soluzione dimostrabilmente ottima, ma aver determinato il valore $z(P)$ può solitamente essere considerato sufficiente. In generale, come già osservato, il rilassamento e l'euristica possono non essere sufficientemente efficaci per garantire la risoluzione del problema. Si deve pertanto applicare il meccanismo del "divide et impera", utilizzando la seguente osservazione:

Lemma 7.1 *Sia $X_1, X_2, \ldots, X_k$ una suddivisione di X ($X_i \subset X$, $i = 1, \ldots, k$, $\bigcup_{i=1}^k X_i = X$) e sia*

$$(P_i) \quad \max \{ \; c(x) \; : \; x \in X_i \; \} \quad ; \qquad\qquad \textit{allora}$$
$$\max \{ \; \underline{z}(P_i) \; : \; i = 1, \ldots, k \; \} \leq z(P) \leq \max \{ \; \bar{z}(P_i) \; : \; i = 1, \ldots, k \; \} \quad .$$

Esercizio 7.1 *Dimostrare il lemma precedente, assumendo per semplicità che X sia un insieme finito. Si discuta poi quali ipotesi aggiuntive sono necessarie qualora X abbia cardinalità infinita, enumerabile e non enumerabile.*

In altre parole, suddividendo un problema in un certo numero di sottoproblemi, l'unione delle cui regioni ammissibili sia la regione ammissibile del problema originario, ed ottenendo una valutazione superiore ed inferiore del valore ottimo della funzione obiettivo di ciascuno dei sottoproblemi individualmente, si ottengono (mediante una semplice operazione di massimo) una valutazione superiore ed inferiore del valore ottimo della funzione obiettivo del problema originario. Ciò mostra come implementare l'operazione di ricombinazione dei risultati dei sottoproblemi nello schema del "divide et impera". Si noti che, delle due operazioni di massimo, una è "favorevole" mentre l'altra è "sfavorevole". In particolare, per restringere il gap tra la valutazione inferiore e quella superiore è necessario far crescere la prima e decrescere la seconda. L'operazione di massimo sulle valutazioni inferiori è quindi "favorevole" in quanto aiuta a far crescere la valutazione inferiore; ciò corrisponde al fatto che la migliore delle soluzioni trovate dall'euristica per i singoli sottoproblemi è una soluzione ammissibile per il problema originario. Viceversa, l'operazione di massimo sulle valutazioni superiori è "sfavorevole": per poter dimostrare che $z(P) \leq \bar{z}$ occorre dimostrare che $z(P_i) \leq \bar{z}$ per *ogni* $i = 1, \ldots, k$. In altri termini, per migliorare la valutazione superiore disponibile occorre che le valutazioni superiori corrispondenti a *tutti* i sottoproblemi siano migliori di essa.
Occorre però notare che ciascun sottoproblema è definito su un insieme ammissibile "più piccolo" di quello di (P), e quindi ci si può aspettare che sia "più facile". In effetti, se l'insieme ammissibile del sottoproblema è "abbastanza piccolo" il problema diviene banale: ad esempio, per $X_i = \{ \, \bar{x} \, \}$ si ha ovviamente $\underline{z}(P_i) = c(\bar{x}) = \bar{z}(P_i)$, mentre per $X_i = \emptyset$ si ha $\underline{z}(P_i) = \bar{z}(P_i) = +\infty$. In altre parole, esiste una suddivisione "sufficientemente fine" di X in un opportuno numero (esponenziale) di sottoinsiemi tale che i corrispondenti sottoproblemi siano sicuramente risolubili; è sufficiente che i sottoproblemi abbiano al più una soluzione. Questi sono quindi sicuramente possibili "casi base" del procedimento "divide et impera" per un problema di *OC*.
Questo tipo di considerazioni può essere facilmente esteso a problemi con parziale struttura combinatoria, quali ad esempio i problemi di Programmazione Lineare Mista: esiste un numero finito

(per quanto esponenziale) di possibili sottoproblemi, corrispondenti ai possibili valori del vettore delle variabili intere, ciascuno dei quali è un problema di PL e quindi “facilmente” risolubile[1].

Conviene a questo punto introdurre il modo più utilizzato (per quanto non l'unico possibile) per implementare l'*operazione di separazione*, ossia la suddivisione di X in $X_1, X_2, \ldots, X_k$: infatti, ciò mostra un diverso modo di descrivere questi algoritmi. Per semplificare l'esposizione supponiamo inizialmente che sia $X \subseteq \{0,1\}^n$, ossia che le soluzioni del problema possano essere descritte attraverso n decisioni binarie, ciascuna rappresentata da una variabile x_i, $i = 1, \ldots, n$: un modo possibile per suddividere X è quello di *prendere decisioni* su alcune delle variabili. Ad esempio, fissato un qualsiasi indice i, possiamo partizionare X come $X_0 \cup X_1$, dove

$$X_0 = \{x \in X : x_i = 0\} \quad \text{e} \quad X_1 = \{x \in X : x_i = 1\} \ .$$

Esistono quindi molti modi diversi di partizionare lo stesso insieme X, a seconda ad esempio della scelta dell'indice i (della decisione da prendere). Inoltre, gli insiemi così ottenuti possono a loro volta essere partizionati seguendo lo stesso schema; ad esempio, fissato un altro indice $j \neq i$, possiamo partizionare X_0 e X_1 come

$$\begin{array}{l} X_0 = (\ X_{00} = \{x \in X_0 : x_j = 0\}\ ) \cup (\ X_{01} = \{x \in X_0 : x_j = 1\}\ ) \\ X_1 = (\ X_{10} = \{x \in X_1 : x_j = 0\}\ ) \cup (\ X_{11} = \{x \in X_1 : x_j = 1\}\ ) \end{array} \ .$$

In questo modo si possono ottenere partizioni di X di qualsiasi dimensione; infatti, per ottenere un sottoinsieme che contenga una sola soluzione è sufficiente ripetere il procedimento n volte, ossia prendere decisioni su tutte ed n le variabili. In effetti, è possibile rappresentare l'insieme ammissibile X attraverso un *albero delle decisioni*, che associa a ciascuna soluzione ammissibile una sequenza di decisioni che la generi.
Si consideri il caso in cui $X = \{\ (x_1, x_2, x_3) \in \{0,1\}^3\ \}$: un albero delle decisioni per X è mostrato in Figura 7.1. Ciascuna foglia dell'albero corrisponde ad un elemento di X; equivalentemente, ciascun cammino dalla radice ad una foglia rappresenta la sequenza di decisioni che genera quell'elemento. Ciascun nodo interno rappresenta un sottoinsieme di X, ed il cammino dalla radice a quel nodo rappresenta la sequenza di decisioni che caratterizzano tutti gli elementi di quel sottoinsieme. La radice dell'albero corrisponde all'intero insieme X, ossia alla sequenza vuota di decisioni.

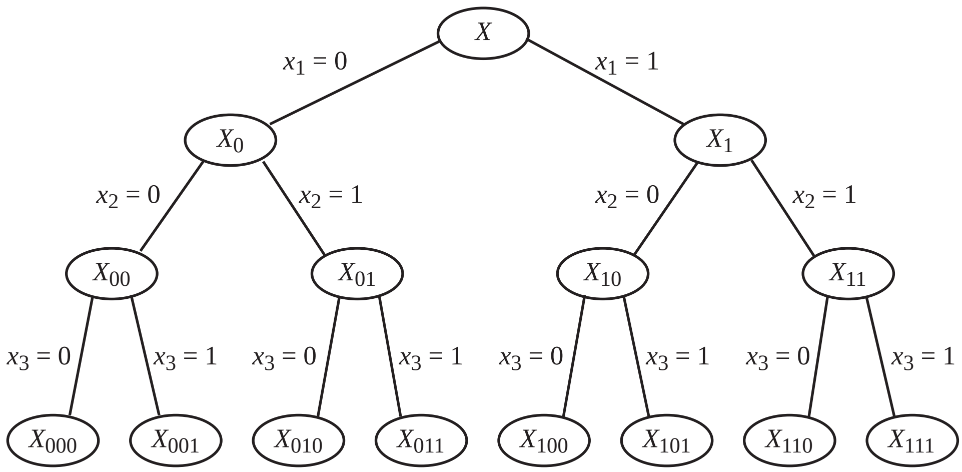


Figura 7.1: Un albero delle decisioni

Si noti che l'albero delle decisioni corrispondente ad uno stesso insieme X non è unico, non foss'altro che per l'ordine in cui vengono prese le decisioni (l'ordinamento delle variabili). In generale, per problemi di OC con particolari strutture può essere utile costruire l'albero delle decisioni in modo

[1]In realtà questo è vero, strettamente parlando, solamente se i vincoli del problema definiscono un insieme compatto rispettivamente alle variabili intere, come ad esempio $\{0,1\}^n$; si può però dimostrare che se i coefficienti della matrice dei vincoli e del vettore dei lati destri sono razionali questo può essere assunto senza perdita di generalità.

tale da "rispettare" la struttura del problema. Si consideri ad esempio l'insieme X di tutti i cicli Hamiltoniani di un dato grafo $G = (V, E)$: poichè X può essere identificato con un opportuno sottoinsieme di $\{0,1\}^{|E|}$, è possibile costruire un albero delle decisioni per X esattamente come illustrato in precedenza: si considerano i lati (le variabili) secondo un qualsiasi ordinamento prefissato e, ad ogni livello dell'albero delle decisioni, si decide se un determinato lato appartiene oppure no al ciclo. Un diverso albero delle decisioni per X può però essere ottenuto nel modo seguente: selezionato in G un nodo arbitrario (ad esempio il nodo 1), si suddivide X in tanti sottoinsiemi quanti sono i lati uscenti dal nodo, ove ciascun sottoinsieme contiene tutti i cicli Hamiltoniani che contengono quel particolare lato. In altre parole, in ciascuno dei sottoinsiemi si è presa la decisione che il lato corrispondente deve appartenere al ciclo Hamiltoniano. Per continuare la costruzione dell'albero delle decisioni si itera il procedimento: a ciascun nodo interno X' dell'albero delle decisioni è associato un cammino semplice P di G che inizia dal nodo 1 e termina in un certo nodo $i \in N$, e $X' = \{ C \in X \,:\, P \subseteq C \}$. X' avrà quindi tanti figli quanti sono i lati $\{i,j\} \in E$ tali che j non appartiene a P, ossia uno per ciascun arco che può essere aggiunto a P ottenendo ancora un cammino semplice. Il figlio di X' corrispondente al lato $\{i,j\}$ contiene tutti i cicli Hamiltoniani che contengono $P \cup \{ \{i,j\} \}$; si è cioè presa l'ulteriore decisione che anche $\{i,j\}$ deve appartenere al ciclo. A ciascuna foglia dell'albero delle decisioni corrisponde quindi un cammino semplice P di G che non può essere ulteriormente esteso; se P è Hamiltoniano ed esiste il lato tra il suo nodo terminale ed 1, allora al nodo è univocamente associato un ciclo Hamiltoniano di G. L'albero delle decisioni relativo al grafo in Figura 7.2(a) è mostrato in Figura 7.2(b). Per semplificare la rappresentazione, in ciascun nodo dell'albero delle decisioni è riportato il nodo terminale i del corrispondente cammino semplice P in G (nella radice è riportato il nodo iniziale, ovvero 1, di ogni cammino).

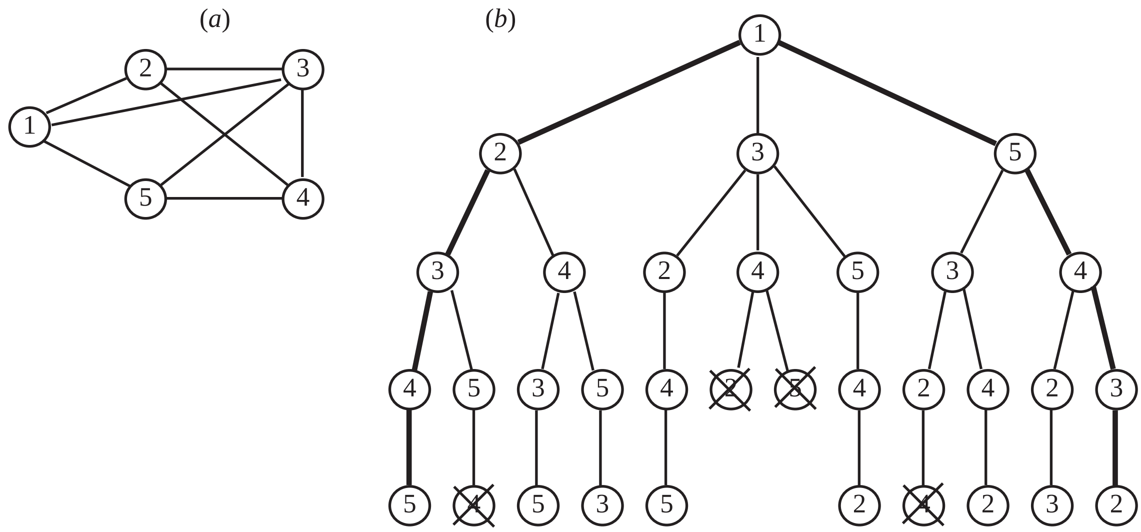


Figura 7.2: Un albero delle decisioni per (TSP)

Questo albero delle decisioni ha alcune caratteristiche rilevanti che è opportuno sottolineare, e che lo differenziano da quello in Figura 7.1:

- il numero di nodi per ogni figlio può essere diverso da due e non costante;
- alcuni cammini nell'albero delle decisioni terminano in nodi (evidenziati in figura con una croce) che corrispondono a sottoinsiemi di X vuoti, ossia a sequenze di decisioni che non generano nessun ciclo Hamiltoniano;
- vengono prese solamente decisioni di un certo tipo, ossia si decide quali lati appartengono al ciclo ma non si decide quali lati *non* appartengono al ciclo, ovvero si fissano variabili a 1 ma non si fissano mai variabili a 0: le decisioni rispetto i lati che non appartengono al ciclo sono prese *implicitamente* (per ogni nodo $h \in N$ interno al cammino P associato ad un nodo dell'albero delle decisioni sono già decisi i due lati incidenti che faranno parte del ciclo, quindi tutti gli altri lati incidenti in h sono di fatto esclusi dal ciclo);

- i sottoinsiemi X'_i, figli di un certo sottoinsieme X' corrispondente ad un nodo nell'albero delle decisioni, non necessariamente formano una partizione di X', ossia può risultare $X'_i \cap X'_j \neq \emptyset$; ad esempio, i due cammini nell'albero delle decisioni evidenziati nella figura corrispondono ovviamente allo stesso ciclo Hamiltoniano di G.

L'albero delle decisioni è uno strumento in grado di generare in modo sistematico tutte le soluzioni ammissibili di X; un algoritmo che intenda esplorare in modo esaustivo X può quindi procedere visitando un qualsiasi albero delle decisioni di X. Ovviamente, nel caso di un problema di OC l'albero delle decisioni avrà dimensione esponenziale, e quindi una sua visita completa è in generale troppo costosa. L'uso di valutazioni superiori ed inferiori può però consentire di evitare di visitare effettivamente alcune zone dell'albero delle decisioni. Possiamo formulare adesso uno schema generale di algoritmo di enumerazione implicita (B&B). L'algoritmo costruisce e visita un albero delle decisioni del problema: la radice dell'albero rappresenta il problema originale (P) (il suo intero insieme ammissibile X), mentre il generico nodo dell'albero rappresenta un sottoproblema

$$(P') \quad \max \{\, c(x) \;:\; x \in X' \,\} \ ,$$

con $X' \subseteq X$. La relazione di discendenza nell'albero corrisponde all'applicazione ricorsiva del procedimento "divide et impera": i sottoproblemi rappresentati dai figli di un nodo hanno come regioni ammissibili quelle ottenute dalla suddivisione della regione ammissibile del problema rappresentato dal nodo. Le valutazioni superiori ed inferiori vengono utilizzate per evitare di visitare interi sottoalberi dell'albero delle decisioni; ad esempio, in corrispondenza ad un nodo (P') per cui si abbia $\underline{z}(P') = \bar{z}(P')$, ossia il problema venga risolto dalla combinazione del rilassamento e dell'euristica disponibili, la visita del sottoalbero di radice (P') viene evitata, in quanto si conoscono già la migliore valutazione superiore ed inferiore di $z(P')$. In questo caso si dice che il sottoalbero viene *visitato implicitamente*. Uno schema generale di algoritmo B&B è rappresentato nel seguente pseudo-codice.

```
procedure B&B( P , z )  {
  Q = { (P) }; z = −∞;
  do { (P′) = Next(Q); Q = Q \ { (P′) };
       (z̄, z̲) = rilassamento(P′);
       if( z̲ > z ) then z = z̲;
       if( z̄ ≤ z ) then continue;
       z̲ = euristica(P′);
       if( z̲ > z ) then z = z̲;
       if( z̄ ≤ z ) then continue;
       Q = Q ∪ branch(P′);
       } while( Q ≠ ∅ );
}
```

Procedura 7.1: Algoritmo *B&B*

La procedura visita un sottoinsieme dell'albero delle decisioni. Siccome l'albero ha una dimensione in principio esponenziale, la parte dell'albero visitata viene costruita dinamicamente nel corso della visita. L'insieme Q contiene i *nodi attivi*, ossia i nodi che sono stati generati ma non ancora esplorati (esaminati), e viene inizializzato con la radice (P) dell'albero. L'algoritmo mantiene la miglior valutazione inferiore $z \leq z(P)$ determinata fino all'iterazione corrente. z viene inizializzato a $-\infty$, ed assume valore finito appena viene generata la prima soluzione ammissibile; se al termine dell'algoritmo si ha $z = -\infty$ allora $X = \emptyset$, altrimenti si ha $z = z(P)$. Normalmente a tale valore è associata la miglior soluzione $x \in X$ determinata fino a quel momento (detta *incumbent*), tale che $z = c(x)$; al termine x viene restituita come soluzione ottima del problema. Nello pseudocodice questo aspetto viene trascurato e si assume di essere interessati solamente al calcolo del *valore ottimo* del problema (il che in un certo senso può essere assunto senza perdita di generalità, si veda il §4.3).
Nella generica iterazione dell'algoritmo viene estratto un nodo (P') da Q; la regola di selezione del nodo in Q determina la *strategia di visita* dell'albero delle decisioni. Viene quindi risolto un *rilassamento*

$$(\bar{P}') \quad \max \{\, \bar{c}(x) \;:\; x \in \bar{X}' \,\}$$

di (P'), in cui si ha cioé $\bar{X}' \supseteq X'$ e $\bar{c}(x) \geq c(x)$ per ogni $x \in X'$; la soluzione ottima x' di $(\bar{P}')$ produce una valutazione superiore $\bar{c}(x') = \bar{z} \geq z(P')$. Tale valutazione viene utilizzata per l'operazione critica (detta *bounding*) dell'algoritmo: se si ha $\bar{z} \leq z$, allora nessuna soluzione in X' ha un valore migliore dell'incumbent, e si può evitare di esplorare ulteriormente la parte dello spazio delle soluzioni rappresentata da X'. In questo caso si dice che il nodo (P') è stato *potato* ("pruned") *dalla valutazione superiore*; infatti, il nodo viene scartato e si passa immediatamente a visitare un altro dei nodi attivi (se ve ne sono).

Un caso particolare in cui ciò sicuramente accade è quando $\bar{X}' = \emptyset \to X' = \emptyset$, e quindi si pone, per definizione $\bar{z} = -\infty$; in questo caso si dice che il nodo è stato *potato per inammissibilità*. Come abbiamo già notato, è necessario essere in grado di *risolvere all'ottimo* $(\bar{P}')$, in quanto una soluzione euristica non garantisce di determinare una corretta valutazione superiore di $z(P')$; ciò implica che si deve, in particolare, essere in grado di determinare se l'insieme ammissibile $\bar{X}'$ di (P') è oppure no vuoto (questo può non essere un compito banale, si pensi al caso della *PL* e, a maggior ragione, a molti problemi $\mathcal{NP}$-ardui). Se $\bar{X}' = \emptyset$ allora anche $X' = \emptyset$ (in quanto $X' \subseteq \bar{X}'$); è quindi inutile esplorare ulteriormente questa parte dello spazio delle soluzioni, in quanto non ne contiene alcuna. Si noti che, in generale, non vale l'inclusione opposta: può cioè capitare che $X' = \emptyset$ ma $\bar{X}' \neq \emptyset$. In questo caso è in effetti ancora "inutile" proseguire l'esplorazione di quel sottoalbero, ma ciò può risultare comunque necessario perché il rilassamento non è in grado di *dimostrare* che $X' = \emptyset$.

Come evidenziato nello pseudo-codice, il rilassamento può anche produrre una valutazione *inferiore*; questo in particolare capita se la sua soluzione ottima è ammissibile ($x' \in X'$), e si può quindi porre $\underline{z} = c(x')$. Se inoltre i valori delle due funzioni obiettivo coincidono, ossia $\bar{c}(x') = c(x')$, allora x' è in particolare ottima per (P'): in questo caso si dice che il nodo viene *potato per ottimalità*, in quanto sarebbe inutile proseguire ulteriormente nell'esplorazione di quel sottoalbero, avendo già dimostrabilmente ottenuto la migliore delle soluzioni in X'. Ciò è assicurato dal fatto che, come risulta naturale, qualora $z < \underline{z}$ il valore dell'incumbent viene aggiornato per tener conto della nuova e migliore valutazione inferiore ottenuta. Se il rilassamento non determina alcuna soluzione ammissibile, basta porre $\underline{z} = -\infty$.

Tutto ciò aiuta a capire come il rilassamento (P') sia una componente fondamentale dell'algoritmo. Infatti esso:

- determina una valutazione superiore che può permettere di evitare di esplorare parti dell'albero delle decisioni;
- può determinare se parti dell'albero delle decisioni non contengono nessuna soluzione;
- può determinare soluzioni ammissibili, e quindi anche le valutazioni inferiori che sono parimenti necessarie per "potare" i nodi.

In principio si potrebbe implementare l'algoritmo *B&B* anche qualora non si disponga di un rilassamento: basta porre $\bar{z} = +\infty$ a meno che $X' = \{\, \bar{x} \,\}$, nel qual caso si pone invece $\bar{z} = c(\bar{x})$. Naturalmente, in questo caso tipicamente si esplora tutto l'albero delle decisioni, e quindi l'algoritmo non è efficiente. Ciò in effetti può essere in parte evitato attraverso l'uso di opportune *regole di dominanza*; per semplicità rimanderemo però la discussione di questo concetto ad un paragrafo successivo.
È bene rimarcare che i nodi "potati" sono quelli in cui la visita si interrompe, ossia le foglie del sottoalbero che viene effettivamente generato (e visitato); non necessariamente sono, e sperabilmente *non* sono, foglie dell'intero albero delle decisioni. Eliminare un nodo permette di non visitare (esplicitamente) tutto il corrispondente sottoalbero dell'albero delle decisioni (per questo si parla di "potatura"); è quindi di fondamentale l'importanza utilizzare rilassamenti (ed euristiche) efficaci, che permettano di potare nodi quanto più precocemente possibile nella visita, evitando così di generare ampie porzioni dell'albero delle decisioni.
Il fatto che nodo non venga potato, ossia che $\bar{z} > z$, può dipendere da due diversi fattori: o la valutazione superiore non è sufficientemente piccola, oppure la valutazione inferiore non è sufficientemente grande. In generale non è dato di sapere quale delle due cause sia quella prevalente; è sempre possibile provare a potare il nodo applicando un'euristica a (P') per determinare una nuova valutazione inferiore

$\underline{z} \leq z(P')$. Se la valutazione inferiore è migliore del (valore dell')incumbent z, si aggiorna z: questo costituisce una nuova possibilità di potare il nodo, anche a parità di valutazione superiore, e quindi il corrispondente controllo viene ripetuto.

Come abbiamo già osservato per il rilassamento, in linea di principio è possibile implementare l'algoritmo *B&B* anche qualora non si disponga di un'euristica: basta porre $\underline{z} = -\infty$ a meno che $\bar{X} = \{\bar{x}\}$, nel qual caso si pone invece $\underline{z} = c(\bar{x})$. A differenza del caso del rilassamento, però, questo tipo di scelta non è necessariamente disastrosa in pratica; in certi casi è persino la scelta più efficiente. Ciò dipende dal fatto che l'algoritmo ha altri due modi per produrre valutazioni inferiori:

1. ripetendo l'operazione di suddivisione si generano insieme X' "piccoli", come ad esempio $X' = \{\, \bar{x} \,\}$, nei quali è "facile" determinare soluzioni ammissibili (in pratica, come vedremo, l'operazione spesso coincide col fissare valori per la variabili del problema: una volta fissate tutte le variabili si è determinata una soluzione, ammissibile o meno);

2. come abbiamo visto, il rilassamento può produrre autonomamente valutazioni inferiori (soluzioni ammissibili), ed in effetti questo tipicamente accade, almeno "ad una certa profondità nell'albero delle decisioni" (al limite nelle foglie, come nel caso precedente).

Se nonostante tutto questo il nodo non viene potato, ossia $\bar{z} > \underline{z}$, allora (P') non è ancora stato risolto: il rilassamento e l'euristica non sono "abbastanza efficaci" per (P'). Si applica quindi il procedimento "divide et impera": si suddivide la regione ammissibile X' in un certo numero finito di regioni ammissibili più piccole $X'_1, \ldots, X'_k$ (normalmente prendendo decisioni sul problema) e si aggiungono i corrispondenti sottoproblemi a Q. Ciò viene detto *branching* in quanto si aggiungono nuovi nodi ed archi al sottoalbero effettivamente generato dell'albero delle decisioni. A questo punto l'iterazione termina e si passa ad esaminare un altro dei nodi attivi. Si noti che, nello schema generale, non è detto che i sottoproblemi corrispondenti ad $X'_1, \ldots, X'_k$ siano visitati immediatamente, né che siano visitati in sequenza; ciò dipende dalla strategia di visita dell'albero, che sarà discussa nel seguito.

Sotto alcune ipotesi, è facile dimostrare che l'algoritmo *B&B* determina $z(P)$ in un tempo finito.

Esercizio 7.2 *Si dimostri l'affermazione precedente, discutendo quali siano le ipotesi necessarie.*

Se l'euristica produce soluzioni ammissibili, oltre alla valutazione inferiore, allora l'algoritmo produce anche una soluzione ottima del problema. In effetti non è difficile modificare lo schema in modo da ottenere possibilmente più soluzioni ottime alternative, qualora ne esistano.

Esercizio 7.3 *Si discuta come modificare l'algoritmo* B&B *perchè possa produrre più di una soluzione ottima di (P). Si discuta inoltre se sia possibile modificarlo in modo tale che produca* tutte *le soluzioni ottime del problema.*

Infine, è facile modificare l'algoritmo *B&B* per trasformarlo in un'euristica con una garanzia sulle prestazioni (assoluta) ε per qualsiasi $\varepsilon \geq 0$ fissato. Per questo è infatti sufficiente modificare i controlli "$\bar{z} \leq z$" in

$$\bar{z} \leq z + \varepsilon \ .$$

Per dimostrare che l'algoritmo così modificato produce una valutazione inferiore di $z(P)$ affetta da errore assoluto minore od uguale a ε, ossia $z \geq z(P) - \varepsilon$, mostreremo che l'algoritmo in effetti produce una valutazione *superiore* arbitrariamente accurata di $z(P)$. Ciò si basa sul seguente risultato:

Lemma 7.2 *Al termine di una qualsiasi iterazione dell'algoritmo* B&B*, sia Q' l'insieme dei predecessori (nell'albero delle decisioni) dei nodi in Q e Q'' l'insieme dei nodi "potati": allora*

$$z(P) \leq \max \{ \ \bar{z}(P') \ : \ P' \in Q' \cup Q'' \ \} \quad .$$

Dimostrazione È facile verificare che

$$\bigcup_{P' \in Q} X' \ \cup \bigcup_{P' \in Q''} X' = X \ ,$$

ossia che l'unione delle regioni ammissibili dei problemi corrispondenti a tutti i nodi potati e dei problemi corrispondenti a tutti i nodi attivi è equivalente all'insieme ammissibile originario X. Infatti, questo è vero alla prima iterazione (in cui $Q = \{(P)\}$ e $Q'' = \emptyset$) e resta vero ogniqualvolta un nodo viene potato oppure viene applicata l'operazione di separazione. Ovviamente, questo è a maggior ragione vero sostituendo Q con Q' (alcuni nodi potati potrebbero essere figli di nodi in Q'). Il risultato segue quindi dal Lemma 7.1. $\diamond$

Corollario 7.1 *Al termine di una qualsiasi iterazione dell'algoritmo* B&B *si ha*

$$z \leq z(P) \leq \max \{ \, z + \varepsilon \, , \, \max \{ \, \bar{z}(P') \, : \, P' \in Q' \, \} \, \} \quad ;$$

quindi, quando l'algoritmo termina ($Q = \emptyset \Rightarrow Q' = \emptyset$), allora z è ε-ottima.

Dimostrazione Sia (P') un nodo potato. Se (P') è stato potato per inammissibilità si ha $\bar{z} = -\infty$, se (P') è stato potato dalla valutazione superiore si ha $\bar{z}(P') \leq z+\varepsilon$ (ciò è vero per il valore di z nel momento in cui il nodo è stato potato, e z è non decrescente nel corso dell'algoritmo), mentre se (P') è stato potato per ottimalità si ha $\bar{z}(P') = \underline{z}(P') \leq z$. Quindi in ogni caso si ha $\bar{z}(P') \leq z + \varepsilon$, ed il risultato segue dal Lemma 7.2. $\diamond$

Si noti che la valutazione superiore indicata dal Corollario 7.1 può essere effettivamente calcolata ad ogni iterazione dell'algoritmo, in quanto tutti i valori di $\bar{z}(P')$ richiesti sono stati effettivamente calcolati. Di consequenza, l'algoritmo *B&B* produce ad ogni iterazione una valutazione superiore ed una inferiore di $z(P)$, e termina quando le due coincidono. Analogamente, al termine l'algoritmo *B&B* modificato restituisce una valutazione superiore con gap assoluto minore o uguale di ε.

7.2 Implementare un algoritmo enumerativo

Lo schema generale del paragrafo precedente include chiaramente algoritmi di efficienza pratica molto diversa. Infatti, come abbiamo già notato, lo schema include anche algoritmi completamente enumerativi, che esaminano cioè tutte le soluzioni ammissibili del problema. Si noti comunque che, per istanze di piccole dimensioni, questi algoritmi possono anche risultare efficienti, in quanto l'esplorazione dell'albero delle decisioni, senza lo sforzo ulteriore richiesto dalla computazione del rilassamento e dell'euristica ad ogni nodo, può essere organizzata in modo da generare e valutare ogni soluzione ammissibile a basso costo. L'efficienza pratica degli algoritmi di enumerazione implicita dipende da come sono implementati alcune componenti fondamentali dell'algoritmo, tra cui le principali sono:

- il *rilassamento* e l'*euristica*;
- la *strategia di visita* dell'albero delle decisioni;
- le eventuali *regole di dominanza* utilizzate;
- la *regola di branching* utilizzata;
- le eventuali operazioni di *pretrattamento* (*preprocessing*) dei dati utilizzate.

Nel seguito discuteremo ciascuna di queste componenti individualmente.

7.2.1 Rilassamento ed euristica

Le euristiche ed i rilassamenti sono già stati ampiamente discussi nei capitoli precedenti; in questo contesto ci limiteremo pertanto ad alcune considerazioni generali.
Come già rimarcato, il rilassamento e l'euristica devono essere quanto più possibile efficaci ed efficienti; ma, come discusso in precedenza, purtroppo molto spesso un aumento dell'efficacia va a discapito di una diminuzione dell'efficienza e viceversa. È quindi necessario operare un attento bilanciamento tra i due aspetti, sia per l'euristica che per il rilassamento. In molti casi pratici risulta essere di fondamentale importanza l'efficacia del rilassamento. Gli algoritmi di enumerazione implicita più efficienti sono

spesso quelli che, grazie ad un rilassamento molto efficace anche se computazionalmente costoso, riescono a mantenere molto basso il numero di nodi dell'albero delle decisioni effettivamente visitati; spesso questi algoritmi fanno ampio uso di *tecniche poliedrali*, che saranno discusse nel paragrafo 7.5, per ottenere valutazioni superiori molto accurate. Una regola pratica, valida in molti casi, indica che si hanno discrete probabilità di ottenere un algoritmo ragionevolmente efficiente qualora il *gap relativo al nodo radice*, ossia $(\bar{z}(P) - \underline{z}(P))/\underline{z}(P)$, sia al massimo dell'$1-2\%$; usualmente la valutazione superiore è il punto critico che permette, o non permette, di ottenere gap sufficientemente bassi.
Naturalmente ciò deve essere considerato solamente come indicativo. In particolare, le considerazioni precedenti sono adeguate per il caso di problemi in cui la difficoltà consiste non tanto nel trovare una soluzione ottima, quanto nel dimostrarne l'ottimalita. Non tutti i problemi di *OC* ricadono in questa classe. Ad esempio, per i problemi decisionali (che possono essere considerati problemi di ottimizzazione in cui tutte le soluzioni ammissibili hanno lo stesso valore della funzione obiettivo) il problema è evidentemente quello di determinare una soluzione ammissibile, oppure dimostrare che non ne esistono; per questi problemi, il ruolo del rilassamento può solamente essere quello di determinare precocemente che $\bar{X}' = \emptyset \rightarrow X' = \emptyset$ per evitere l'esplorazione di alcune aree dello spazio delle soluzioni. In effetti, si può ribaltare la regola precedentemente enunciata per dire che i problemi che si prestano ad essere risolti efficientemente da algoritmi di enumerazione implicita sono quelli in cui la difficoltà consiste fondamentalmente nel dimostrare l'ottimalita della soluzione. Per i problemi con "molti" vincoli e quindi "poche" soluzioni ammissibili, in cui la difficoltà è determinare una soluzione, possono infatti risultare più efficienti tecniche risolutive alternative (per quanto anch'esse utilizzino visite di strutture simili all'albero delle decisioni) basate su tecniche di inferenza logica, che rientrano sotto il nome generico di *programmazione logica con vincoli*. Per una descrizione delle idee base di queste tecniche, e circa la possibilità di utilizzarle in alternativa o in congiunzione con le tecniche di *programmazione matematica* qui descritte, si rimanda alla letteratura citata.

Terminiamo questa breve discussione con alcune considerazioni molto generali ed intuitive. Poiché euristica e rilassamento devono essere eseguite ad ogni nodo dell'albero delle decisioni, è chiaramente fondamentale, insieme all'efficacia, la loro efficienza. Si può affermare che l'importanza di sviluppare tecniche risolutive sempre più efficienti per problemi "facili" risieda in buona parte nell'impatto che ciò ha sull'efficienza di tecniche enumerative che le utilizzano. In particolare, risulta spesso importante scegliere un rilassamento ed un'euristica che "collaborino", ossia tale che il primo sia in grado di sfruttare parte del lavoro svolto dalla seconda, o viceversa. Esempi di euristiche e rilassamenti di questo tipo sono:

- rilassamenti ed euristiche che sfruttano la risoluzione di uno stesso problema, si pensi ad esempio al rilassamento basato su (MS1-T) ed all'euristica "twice around MST" per il problema del commesso viaggiatore, oppure al rilassamento continuo ed all'euristica CUD per il problema dello zaino;
- come estensione del caso precedente, rilassamenti Lagrangiani e corrispondenti euristiche Lagrangiane;
- il rilassamento continuo (o Lagrangiano) e le tecniche di arrotondamento.

Infine, è utile osservare come risulti fondamentale, ai fini dell'efficienza complessiva dell'approccio, non tanto l'efficienza della soluzione di un singolo rilassamento/euristica, quanto il tempo complessivo utilizzato per risolvere tutti quelli richiesti dalla visita dello spazio delle soluzioni. L'osservazione fondamentale è che durante la visita vengono risolti una serie di problemi di ottimizzazione simili, ossia che differiscono per "pochi" dati: in questi casi, la conoscenza di una soluzione ottima per uno di questi problemi può costituire un valido aiuto nel determinare la soluzione ottima degli altri. Si consideri ad esempio un algoritmo *B&B* in cui ad ogni nodo dell'albero delle decisioni venga risolto un problema di *PL*, ad esempio il rilassamento continuo di una formulazione *PLI* del problema, e la regola di separazione sia implementata semplicemente fissando una variabile (questo esempio sarà discusso più in dettaglio nel seguito). In questo caso si possono applicare le *tecniche di riottimizzazione* per la *PL* discusse nel paragrafo 2.3.3. Ciò tipicamente permette di rendere estremamente più efficiente l'approccio complessivo, in quanto il tempo necesario per risolvere i problemi di PL nei nodi interni dell'albero della visita è una frazione di quello necessario per risolvere il rilassamento continuo iniziale al nodo radice (questo può avere un impatto anche sulla scelta delle strategie di visita dell'albero delle decisioni, come discusso nel seguito). Può quindi accadere che algoritmi per la PL che siano efficienti nel risolvere "da zero" i problemi, ma non molto efficienti nel riottimizzare a seguito di piccoli cambiamenti nei dati del problema, risultino complessivamente meno efficaci, quando utilizzati all'interno di un approccio di enumerazione implicita, di algoritmi magari meno efficienti "da zero", ma molto efficienti in riottimizzazione (questo è ad esempio il caso dei *metodi del punto interno*, un'alternativa agli algoritmi del simplesso discussi in queste dispense). Ciò vale ovviamente anche per altri problemi di ottimizzazione. Possiamo concludere questo paragrafo ribadendo che, per l'efficienza complessiva di un approccio di enumerazione implicita, è necessario scegliere accuratamente rilassamento ed euristica, tenendo conto non solo dell'efficienza ed efficacia di ciascuno dei due separatamente, ma anche delle interazioni tra i due. Occorre inoltre selezionare accuratamente, tra gli algoritmi risolutivi disponibili, quelli più adatti all'uso nel contesto di un algoritmo enumerativo.

7.2.2 La strategia di visita

La strategia di visita dell'albero di enumerazione è fondamentalmente dettata dalla strategia di selezione del prossimo nodo da visitare dall'insieme Q (procedura $Next()$). Si distingue usualmente tra *visite topologiche* e *visite basate sull'informazione*.

Le visite topologiche scelgono il prossimo nodo da visitare unicamente sulla base della struttura topologica dell'albero delle decisioni; ciò corrisponde a strategie di selezione del nodo che dipendono unicamente dalla sua posizione in Q. Le due strategie di visita topologica più note sono quella a ventaglio, o *breadth-first*, corrispondente ad implementare Q come una fila (queue), e quella a scandaglio, o *depth-first*, corrispondente ad implementare Q come una pila (stack). Di queste, nel contesto degli algoritmi di enumerazione implicita può essere particolarmente utile la strategia depth-first nel caso in cui non si disponga di un'euristica (efficace). Poichè per potare i nodi attraverso la valutazione superiore è necessario disporre di una valutazione inferiore (tranne nel caso in cui il rilassamento sia vuoto), l'uso di una strategia depth-first può essere indicato in quanto porta la visita velocemente verso le foglie dell'albero delle decisioni, e quindi può consentire di generare velocemente soluzioni ammissibili. Le visite topologiche offrono alcuni vantaggi dal punto di vista dell'implementazione. In primo luogo sono semplici da realizzare ed il costo di gestione di Q è basso. In secondo luogo, la strategia depth-first si presta ad essere implementata in modo tale da mantenere molto basso il numero di nodi attivi nell'albero delle decisioni. Infatti, è possibile, durante l'operazione di separazione, evitare di generare tutti i figli per inserirli in Q; si può invece iniziare l'esame del figlio appena generato, rimandando la generazione dei suoi ulteriori fratelli (se ne ha) al momento in cui sia terminata la visita del sottoalbero corrispondente. In questo modo, ad ogni istante saranno attivi solo un numero di nodi non superiore all'altezza dell'albero (sono attivi anche i nodi parzialmente visitati, ossia per cui la visita non ha ancora terminato di esaminare tutti i figli). Poichè può essere necessario, nelle implementazioni, memorizzare molta informazione in corrispondenza di ogni nodo attivo dell'albero delle decisioni, ciò può consentire di risparmiare molta memoria, specialmente nel caso in cui i nodi dell'albero delle decisioni abbiano molti figli. Inoltre, in questo modo i problemi esaminati in sequenza mentre si scende in profondità nell'albero delle decisioni—ma non necessariamente quando si risale, ossia si effettua un *backtrack*—sono molto simili tra loro, il che può essere utile qualora si utlizzino tecniche di riottimizzazione (si veda il paragrafo precedente) nel rilassamento e/o nell'euristica. Infine, è più facile mantenere l'informazione relativa ad ogni singolo sottoproblema in "forma differenziale", ossia memorizzare, per ogni nodo dell'albero delle decisioni, solamente le differenze tra il problema corrispondente al nodo ed il problema corrispondente al padre: ciò può consentire di risparmiare ulteriormente memoria. Si noti che se si utilizza una visita depth-first può essere cruciale selezionare oculatamente l'ordine con cui vengono esaminati i figli di ogni nodo: infatti, generare un figlio prima di un altro significa esplorare tutto il sottoalbero corrispondente al primo prima del sottoalbero corrispondente al secondo, per cui è necessario cercare di indirizzare la ricerca prima verso i figli "più promettenti".

Le regole di visita basate sull'informazione utilizzano informazione sui nodi in Q per definire il prossimo nodo da estrarre; in altri termini, corrispondono ad implementare Q come una coda di priorità. La strategia con informazione più usata è la *best-first*, in cui ad ogni nodo viene associato il valore della valutazione superiore prodotta dal corrispondente rilassamento e viene selezionato il nodo con valore maggiore, che corrisponde al sottoalbero "più promettente", ossia nel quale si dovrebbero avere maggiori probabilità di incontrare una soluzione ottima. Si noti che ciò richiede una modifica dell'algoritmo 7.1, in quanto il rilassamento deve essere risolto immediatamente dopo la generazione di un nodo e prima del suo inserimento in Q; naturalmente, il nodo non viene inserito se risulta potato dalla valutazione superiore, ma il controllo deve comunque essere ripetuto quando il nodo viene estratto da Q, in quanto la valutazione inferiore potrebbe essere cambiata nel frattempo. La strategia best-first è molto usata in quanto solitamente riesce ad indirizzare la visita verso le zone "più promettenti" dell'albero delle decisioni, meglio di quanto non accada utilizzando strategie topologiche. Inoltre, il Lemma 7.2 mostra che il nodo attivo avente il maggior valore della valutazione superiore associata è quello che determina la valutazione superiore corrente per l'intero algoritmo; visitare per primo quel nodo corrisponde dunque a cercare di far diminuire la valutazione superiore, il che, in particolare qualora si usi una regola di terminazione approssimata, può rendere più efficiente l'approccio. Per contro, la strategia best-first è più complessa da implementare, e tipicamente accade che vengano esaminati in sequenza nodi "lontani" nell'albero delle decisioni, che corrispondono a problemi "molto diversi" tra loro; questo può rendere meno efficaci, o più onerose da implementare, le tecniche di riottimizzazione. Pertanto nelle implementazioni efficienti si usano anche tecniche miste. Ad esempio, è possibile effettuare una visita tendenzialmente depth-first dell'albero delle decisioni, ma controllando il valore del gap relativo $(\bar{z}(P') - \underline{z}(P'))/\underline{z}(P')$ dei nodi generati: se il valore diminuisce "abbastanza" discendendo nell'albero delle decisioni (ad esempio, almeno dello 0,1% per ogni decisione presa) si continua con la visita depth-first, altrimenti si usa una qualche strategia con informazione per selezionare un nuovo nodo "promettente" da cui continuare la ricerca.

7.2.3 Regole di dominanza

Un modo ulteriore per "potare" nodi dell'albero delle decisioni, rispetto a quelli indicati nello schema generale, consiste nell'individuare ed applicare regole di dominanza. Si consideri un nodo attivo (P') tale che esista un diverso nodo (P''), attivo o già esaminato, per il quale si possa affermare con sicurezza che $z(P') \leq z(P'')$: in questo caso si dice che (P'') domina (P'). È chiaro che non è necessario che l'algoritmo esplori il nodo (P') purchè venga esplorato (P''); (P') può quindi essere *potato per dominanza*. Si noti che la potatura basata sulla valutazione superiore è un caso particolare della potatura basata sulla dominanza: in tale caso, infatti, nella visita è stato generato un nodo (P'') tale che $z \leq z(P'')$, e quindi la relazione $\bar{z}(P') \leq z$ implica che (P') sia dominato da (P'').

Una relazione tra (P') e (P'') che permetta di affermare che (P'') domina (P') viene detta *regola di dominanza*. Per l'osservazione precedente, la normale "potatura" dei nodi basata sul valore della loro valutazione superiore è quindi una particolare regola di dominanza. Esistono diversi modi possibili per costruire regole di dominanza tra sottoproblemi di uno stesso problema di *OC*. Alcune regole di dominanza verranno illustrate nel contesto degli algoritmi di programmazione dinamica, discussi nel paragrafo 7.4. Pertanto, per il momento ci limitiamo a notare come il principale problema dell'applicazione delle regole di dominanza all'interno di un algoritmo di enumerazione implicita sia la necessità di verificare la regola tra un nodo (P'), ad esempio quello appena estratto da Q, e tutto l'insieme dei nodi attivi e dei nodi già visitati: ciò può risultare molto oneroso. Infatti, le regole di dominanza si sfruttano di solito quando sia possibile organizzare la visita dell'albero delle decisioni in modo tale che il loro controllo sia effettuabile in modo efficiente, come accade, appunto, nel caso degli algoritmi discussi nel §7.4.

7.2.4 Regole di branching

Per uno stesso problema di *OC* si possono usare diverse regole di branching. La selezione della regola di branching può avere un forte impatto sull'efficienza di un algoritmo di enumerazione implicita, pertanto la scelta delle regola deve essere compiuta con oculatezza. In generale, affinchè l'algoritmo termini occorre solamente che la regola sia *completa*, ossia tale che $X' = X'_1 \cup \ldots \cup X'_k$, e che ciascuno dei sottoinsiemi X'_i sia un sottoinsieme proprio di X'. In generale, però, alcune regole di branching possono avere proprietà che le rendono particolarmente attraenti, quali:

- partizionare lo spazio delle soluzioni, ossia garantire che $X'_i \cap X'_j = \emptyset \;\; \forall i, j$;
- equidividere X', ossia garantire che la cardinalità di tutti i sottoinsiemi X'_i sia approssimativamente uguale;
- garantire che la soluzione ottima del rilassamento utilizzato per X' *non* sia ammissibile per i rilassamenti utilizzati per ciascun X'_i, in modo tale da rendere possibile una decrescita stretta della valutazione superiore;
- essere "compatibile" con il rilassamento e l'euristica utilizzata, ossia non distruggere la struttura che li rende efficienti;
- generare "pochi" figli per ogni nodo, ossia tipicamente un numero costante che non dipende dalla dimensione del problema.

Discutiamo adesso brevemente le precedenti proprietà. Ovviamente, una regola di branching che partizioni lo spazio delle soluzioni garantisce che esista un solo cammino nell'albero delle decisioni per ogni soluzione. Quando ciò non accade, come nell'esempio in Figura 7.2, la stessa soluzione può essere visitata più volte, il che chiaramente corrisponde ad uno spreco di risorse.

Una regola di branching che equisuddivida gli insiemi garantisce che il corrispondente albero delle decisioni sia "bilanciato", ossia che la lunghezza di tutti i cammini dalla radice alle foglie sia approssimativamente uguale, e quindi "relativamente corto". Ad esempio, l'albero delle decisioni per un problema con n variabili binarie costruito come in Figura 7.1 ha altezza n, anche se ha 2^n foglie. Si consideri, come esempio estremo di una regola che costruisce alberi "sbilanciati", una regola di branching che costantemente selezioni un elemento $\bar{x} \in X'$ e costruisca due figli di X', uno corrispondente a $X'_1 = \{\, \bar{x} \,\}$ e l'altro corrispondente a $X'_2 = X' \setminus X'_1$: chiaramente il corrispondente albero delle decisioni ha una profondità pari alla cardinalità di X, ossia tipicamente esponenziale nella dimensione del problema (tale regola oltretutto sarebbe difficilmente compatibile con un qualsiasi rilassamento ed euristica). Equisuddividere fa in modo che, in linea di massima, tutti i problemi allo stesso livello dell'albero delle decisioni siano "ugualmente difficili" e "significativamente più facili" di quelli del livello precedente. Si pensi ad esempio alla regola di branching "sbilanciata" appena discussa: uno dei due figli corrisponde ad un problema estremamente facile, l'altro ad un problema che ha all'incirca le stesse soluzioni ammissibili di quello originario, e quindi è in generale altrettanto difficile.
L'equisuddivisione dei sottoproblemi è particolarmente importante qualora si usi una strategia depth-first, che visita un sottoalbero finchè non ne ha completato l'esplorazione. Si pensi, come caso estremo, alla strategia "sbilanciata" in cui al nodo radice viene selezionata in X'_1 l'unica soluzione ottima del problema, ma l'algoritmo inizia a visitare X'_2. Un esempio più realistico di regola "non bilanciata" è il seguente: si consideri un problema di Programmazione 0/1 che contenga un vincolo "tipo semiassegnamento" (normalmente chiamato *SOS*, da Special-Ordered Set)

$$\textstyle\sum_{i \in S} x_i = 1 \;\; .$$

Chiaramente, in questo caso è possibile utilizzare la regola di branching "standard" che seleziona una delle variabili e la fissa a 0 oppure ad 1; tale regola risulta però sbilanciata. Infatti, per via del vincolo di semiassegnamento fissare ad 1 una variabile x_i corrisponde di fatto a fissare a 0 tutte le variabili x_j per $j \in S$ (si veda il paragrafo 7.2.5); per contro, fissare x_i ha 0 non ha, in generale, un analogo effetto. Di conseguenza, X'_0 contiene tutte le soluzioni di X' in cui una variabile è fissata a 0, mentre X'_1 contiene tutte le soluzioni di X' in cui $|S|-1$ variabili sono fissate a 0. X'_0 contiene quindi probabilmente più soluzioni di X'_1. Una strategia "bilanciata" per lo stesso problema potrebbe essere implementata come segue: si partiziona S in due sottoinsiemi S_1 ed S_2, di cardinalità approssimativamente uguale, e si costruiscono i due insiemi X'_1 ed X'_2 in cui sono fissate a 0 rispettivamente tutte le variabili di S_1 ed S_2. Questa regola—o sue generalizzazioni per vincoli che definiscono altri tipi di Special Ordered Set—si dimostra normalmente migliore della strategia "standard". Si noti che questa strategia partiziona lo spazio delle soluzioni: qualsiasi soluzione ammissibile del problema ha $x_i = 1$ per un qualche indice $i \in S$, quindi la soluzione appartiene esattamente ad uno tra X'_1 ed X'_2. Ciò non sarebbe vero nel caso di vincoli di tipo $\sum_{i \in S} x_i \leq 1$.

Garantire che la soluzione ottima del rilassamento utilizzato per X' *non* sia ammissibile per i rilassamenti utilizzati per ciascun X'_i è necessario se si vuole avere una speranza che l'operazione di branching produca una decrescita stretta della valutazione superiore associata al nodo, e quindi possibilmente della valutazione superiore complessiva, nel caso—tipico—in cui la funzione obiettivo non cambi. Sia infatti $\bar{x} \in \bar{X}'$ la soluzione ottima del rilassamento al nodo X', e supponiamo per semplcità che tutti i rilassamenti usino la funzione obiettivo $c()$ del problema originale (P): se $\bar{x} \in \bar{X}'_i$ per qualche i, allora ovviamente $\bar{x}$ è una soluzione ottima del rilassamento al nodo $\bar{X}'_i$.

Esercizio 7.4 *Si dimostri l'affermazione precedente.*

Di conseguenza, la valutazione superiore $\bar{z}(P') = c(\bar{x})$ di $z(P')$, ottenuta risolvendo il rilassamento al nodo X', non viene migliorata dal branching: infatti si ha $\bar{z}(P'_i) = c(\bar{x})$ per almeno un i, e quindi $\max_i \; \bar{z}(P'_i) \geq c(\bar{x})$. Per questo è importante che la regola di branching renda inammissibile la soluzione ottima del rilassamento. In effetti, questo è esattamente il modo in cui normalmente si costruisce una regola di branching: si esamina la soluzione ottima del rilassamento e si prendono decisioni sulle variabili in modo da renderla inammissibile. Ciò sarà illustrato più in dettaglio nel seguito.

La compatibilità della regola di branching con il rilassamento e l'euristica è fondamentale: prendere decisioni su un problema significa modificarlo, e, come abbiamo visto, modificare un problema di OC "facile" può renderlo "difficile". Illustriamo questa problematica con un esempio. Si consideri il problema del cammino minimo vincolato introdotto nel paragrafo 6.2.1.2. Un ovvio rilassamento per questo problema è il rilassamento per eliminazione di vincoli, o quello Lagrangiano, che produce un problema di cammino minimo. Si consideri quindi la regola di branching "standard" che prende decisioni su una singola variabile del problema, ossia che seleziona un arco (i, j) e definisce X'_0 come l'insieme di tutti i cammini in X' che non contengono (i, j) e X'_1 come l'insieme di tutti i cammini in X' che contengono (i, j). Si consideri quindi il rilassamento da risolvere ad un generico nodo dell'albero delle decisioni: si tratta di determinare un cammino di costo minimo che non contiene un certo insieme di archi, e che contiene un altro insieme di archi. Mentre la prima richiesta non ha alcun impatto sulla struttura del problema—è sufficiente eliminare tali archi dal grafo sul quale si calcola il cammino minimo—la seconda richiesta ne altera profondamente la difficoltà: determinare un cammino di costo minimo tra tutti quelli che contengono un insieme prefissato di archi è un problema $\mathcal{NP}$-arduo[2]. Quindi, questa regola di branching non può essere utilizzata in combinazione con questo rilassamento. È bene notare che in moltissimi casi è possibile adattare facilmente euristiche e rilassamenti affinchè lavorino su problemi in cui sono state prese decisioni. Ad esempio, per il problema dello zaino è immediato adattare il rilassamento e le euristiche viste al caso in cui alcune variabili sono fissate a 0 (basta eliminare i corrispondenti oggetti dalla lista) o a 1 (basta eliminare i corrispondenti oggetti dalla lista diminuendo opportunamente la capacità dello zaino). Ciò corrisponde al fatto che i problemi di OC molto spesso hanno la proprietà di *auto-riducibilità*, per cui il problema (P) in cui alcune variabili sono fissate può essere riformulato come un'istanza diversa (più piccola) dello stesso problema.

Esercizio 7.5 *Si discuta come l'auto-riducibilità di (CMST), descritta nel paragrafo 4.3, possa essere utilizzata per adattare le euristiche ed i rilassamenti presentati affinchè lavorino su un'istanza del*

[2] Il problema può essere risolto con una complessità polinomiale nella dimensione del grafo ma esponenziale nella cardinalità dell'insieme di archi fissati.

problema sulla quale sono state prese decisioni (alcuni archi appartengono/non appartengono all'albero ottimo); si noti che l'operazione di "aggregazione" di nodi illustrata nel paragrafo 4.3 deve essere applicata ricorsivamente, diventando un'operazione di aggregazione di interi sottoalberi in un singolo nodo.

Esercizio 7.6 *Si discuta come adattare il rilassamento (MS1-T) per il problema (TSP) al caso in cui si sia deciso che alcuni archi fanno parte del ciclo hamiltoniano ed altri non ne fanno parte (suggerimento: è sufficiente eliminare dal grafo i secondi ed "aggregare" i nodi corrispondenti ai primi, avendo però cura di operare ricorsivamente e rispettando la struttura dei cammini creati dagli archi già fissati).*

Infine, una regola di branching che generi "pochi" figli evita alcuni problemi. Innanzitutto, una regola che generi, ad esempio, solo due figli per ogni nodo è più semplice da implementare. Viceversa, una regola che genera "molti" figli per ogni nodo può essere complessa da implementare, e può risultare costosa sia in termini di memoria che di tempo. Infatti, se ogni nodo genera "molti" figli l'insieme Q cresce rapidamente; quindi cresce la quantità di memoria necessaria per memorizzare le descrizioni dei nodi attivi, e le operazioni su Q possono diventare costose. Inoltre, generare molti figli ha intrinsecamente un costo che può essere rilevante; come esempio estremo si pensi ad una regola che genera, per ogni nodo, un numero di figli esponenziale nella dimensione del problema. Questo è in un certo senso il requisito "duale" rispetto a quello dell'equisuddivisione degli insiemi: mentre l'equisuddivisione garantisce che l'albero non sia troppo profondo, ossia che esista un cammino "ragionevolmente corto" dalla radice a ciascuna soluzione, il criterio relativo ai pochi figli garantisce che l'albero non sia troppo poco profondo, ossia che lo sforzo necessario per esaminare tutti i figli di un dato nodo sia limitato.

7.2.5 Preprocessing

Le operazioni di "pretrattamento" (preprocessing) del problema sono tutte quelle manipolazioni sui dati del problema che consentono di ottenere rapidamente informazione utile a rendere più efficiente il processo risolutivo. Spesso le operazioni di preprocessing sfruttano proprietà apparentemente banali ed ovvie del problema; alcuni semplici esempi sono:

- nel problema dello zaino, qualsiasi oggetto i con $a_i > b$ non può far parte della soluzione ottima, ossia si può porre $x_i = 0$;

- nel problema dello zaino, per qualsiasi oggetto i con $c_i \leq 0$ ed $a_i \geq 0$ si può porre $x_i = 0$, e, viceversa, per qualsiasi oggetto i con $c_i \geq 0$ ed $a_i \leq 0$ si può porre $x_i = 1$;

- nel problema del (CMST), nessun lato $\{i, j\}$ tale che $c_{ij} \geq \max\{ c_{ri} , c_{rj} \}$ (dove r è la radice) può far parte dell'albero ottimo;

- nel problema del (CMST), se un nodo i ha peso $b_i = Q$ (o, più in generale, $b_i + \max_{j \neq i}\{ b_j \} > Q$) il lato $\{r, i\}$ deve necessariamente far parte dell'albero ottimo.

Può apparire sorprendente che sia necessario verificare questo tipo di condizioni; in prima approssimazione sembrerebbe ragionevole che, ad esempio, non venissero forniti in input ad un problema dello zaino oggetti che da soli non entrano nello zaino. Ciò però non sempre accade, per almeno due motivi:

1. l'istanza del problema può essere stata generata a partire da dati complessi con una procedura complessa, e può essere di dimensioni tali da non rendere possibile una sua verifica puntuale da parte di un esperto;

2. l'istanza del problema può essere stata generata a partire da un'istanza diversa, tipicamente prendendo alcune decisioni o rilassando alcuni vincoli.

La seconda osservazione è particolarmente importante nel contesto degli algoritmi enumerativi, specialmente nei nodi diversi dalla radice. Infatti, le procedure di preprocessing applicate al problema (P') corrispondente al nodo dell'albero delle decisioni attualmente visitato possono essere viste come un modo per *sfruttare implicazioni logiche delle decisioni prese* (nel cammino dell'albero delle decisioni che raggiunge il nodo) per *estrarre informazione utile su* (P'). Ad esempio, nel caso dello zaino, denotando con S l'insieme degli indici delle variabili fissate ad 1 in (P'), si ha che qualsiasi oggetto con $a_i > b - \sum_{j \in S} a_j$ non può più essere inserito nello zaino, anche se $a_i \leq b$ e quindi l'oggetto è, in linea di principio, un input "ragionevole" per l'istanza: è solamente quando vengono prese alcune decisioni che si può "dedurre" che tale oggetto non è più utilizzabile. Analogamente, nel caso del (CMST) fissare alcuni archi come facenti parte dell'albero ottimo richiede di aggregare i sottoalberi così formati trasformandoli in nodi (si veda l'esercizio 7.5): il peso di tali nodi (dato dalla somma dei pesi dei nodi che compongono il sottoalbero) può quindi diventare abbastanza grande da permettere di applicare le operazioni di preprocessing precedentemente indicate.

Esercizio 7.7 *Per tutti i problemi di* OC *incontrati finora si discutano opportune operazioni di preprocessing.*

Le operazioni di preprocessing sono dunque uno dei modi possibili per estrarre efficientemente informazione sul problema, o meglio sui sottoproblemi corrispondenti ai nodi durante la visita dell'albero delle decisioni. In un certo senso, quindi, si può dire che anche la computazione della valutazione superiore o il fissaggio per costi ridotti (si veda il paragrafo 6.1.2.2) sono operazioni di preprocessing, in quanto consentono di ottenere informazione sul sottoproblema (sicuramente non contiene una soluzione ottima, sicuramente alcune variabili devono essere fissate in certi modi). Di solito si considerano operazioni di preprocessing solo quelle che usano relazioni "ovvie" e sono molto semplici da implementare, anche se questa distinzione non è chiara, e ci sono alcune operazioni di preprocessing molto sofisticate e potenzialmente computazionalmente costose.

Uno degli ambiti in cui le operazioni di preprocessing sono state maggiormente sviluppate è quello degli algoritmi di enumerazione implicita sviluppati per risolvere qualsiasi problema formulato come *PLI*. Normalmente questi algoritmi usano solutori generali ed efficienti per la *PL* per risolvere il rilassamento continuo del modello, e saranno discussi più in dettaglio nel paragrafo 7.3.1. Poichè non hanno nessuna informazione sulla struttura combinatoria del problema da risolvere, questi algoritmi si concentrano sui vincoli della rappresentazione *PLI* e tentano di ricavare informazione da questi. Si consideri ad esempio un singolo vincolo lineare $\sum_{i=1}^{n} a_i x_i \leq b$ per cui tutte le variabili x_i siano binarie (ossia un *vincolo di zaino*). Chiamiamo $P = \{\, i \,:\, a_i > 0 \,\}$ e $N = \{\, i \,:\, a_i < 0 \,\}$, e definiamo $\underline{b} = \sum_{i \in N} a_i$. Per qualsiasi $j \in P$, se si ha $a_j + \underline{b} > b$ allora sicuramente deve essere $x_j = 0$. Questa è un'ovvia generalizzazione delle operazioni di preprocessing viste nel caso del problema dello zaino; come discusso in precedenza, ed illustrato nel paragrafo 7.3.1, questo tipo di relazione può non essere vera per il vincolo nella formulazione iniziale del problema, ma diventarlo dopo che sono state prese alcune decisioni. In generale, si può ottenere da queste tecniche informazione sul problema in forma diversa, e più generale, del fissare variabili. Con la notazione precedente, ad esempio, si ha:

- se $a_i + a_j + \underline{b} > b$, allora il vincolo $x_i + x_j \leq 1$ è soddisfatto da tutte le soluzioni ammissibili del problema;
- se $a_i + (\underline{b} - a_j) > b$ per $j \in N$, allora il vincolo $x_i \leq x_j$ (corrispondente all'implicazione logica "$x_i = 1 \Longrightarrow x_j = 1$") è soddisfatto da tutte le soluzioni ammissibili del problema.

Esercizio 7.8 *Si estendano le relazioni precedenti al caso di insiemi di variabili di cardinalità maggiore di due.*

Queste relazioni permettono di derivare vincoli lineari che sono "logicamente implicati" dai vincoli originali e dall'integralità delle variabili. Questi vincoli, se aggiunti alla formulazione, possono migliorare la qualità della valutazione superiore ottenuta dal rilassamento continuo.

Esempio 7.1: Preprocessing per la *PLI*
Si consideri ad esempio il problema

$$\max\{\ 2x_1 + x_2\ :\ 27x_1 + 14x_2 \leq 40\ ,\ x_i \in \{0,1\}\quad i = 1,2\ \}\ .$$

In questo caso $P = \{1,2\}$, $N = \emptyset$, $\underline{b} = 0$; applicando la prima relazione si ha che il vincolo $x_1 + x_2 \leq 1$ deve essere soddisfatto da tutte le soluzioni ammissibili del problema. Aggiungere il vincolo alla formulazione non cambia quindi l'insieme delle soluzioni ammissibili, ma cambia il rilassamento continuo. Infatti, è facile verificare che la soluzione ottima del rilassamento continuo senza il vincolo è $[\,40/27\,,\,0\,]$, che corrisponde ad una valutazione superiore sull'ottimo del problema di 80/27; invece aggiungendo il vincolo la soluzione ottima del rilassamento continuo diviene $[\,1\,,\,0\,]$, che è intera e quindi ottima per il problema originario.

Esercizio 7.9 *Si verifichino le affermazioni precedenti disegnando la regione ammissibile del rilassamento continuo del problema nell'esempio con e senza il vincolo* $x_1 + x_2 \leq 1$.

Queste operazioni di preprocessing sono quindi un caso particolare delle *tecniche poliedrali* discusse al paragrafo 7.5, in quanto, come l'esempio precedente mostra chiaramente, cercano di ottenere una più accurata descrizione dell'inviluppo convesso dell'insieme delle soluzioni intere del problema. Per ulteriori dettagli sulle tecniche di preprocessing sviluppate la *PLI* si rimanda alla letteratura citata.

Per terminare questa sezione notiamo che gli algoritmi di enumerazione implicita si prestano bene ad essere implementati su architetture perallele; a parte l'uso di algoritmi paralleli per risolvere euristica e rilassamento, che dipende dal particolare problema affrontato, è chiaramente sempre possibile effettuare in parallelo la visita dell'albero delle decisioni, valutando più nodi attivi contemporaneamente, uno su ciascun processore. In questo caso Q è una struttura dati distribuita, e le operazioni di aggiornamento della valutazione inferiore z richiedono comunicazione tra i vari processori. Lo sviluppo di algoritmi di enumerazione implicita massicciamente paralleli non ha ancora raggiunto uno stadio di piena maturità, e quindi non discuteremo ulteriormente questo punto.

7.3 Esempi di algoritmi enumerativi

In questa sezione discuteremo alcuni esempi di algoritmi di enumerazione implicita applicati a specifici problemi di *OC*. Per ogni problema discuteremo varie possibilità per implementare i vari aspetti cruciali dell'algoritmo (rilassamento ed euristica, regola di branching, strategia di visita, ...) esaminando i prò ed i contro di alcune alternative diverse. Mostreremo poi il funzionamento dell'algoritmo su un esempio.

7.3.1 Il problema dello zaino

Per il problema dello zaino possiamo implementare un algoritmo di enumerazione implicita caratterizzato dai seguenti componenti:

- rilassamento continuo;
- euristica greedy CUD;
- branching "standard" sull'unica variabile frazionaria;
- visita depth-first;
- preprocessing corrispondente nel fissare $x_i = 0$ se $a_i > \bar{b}$, dove $\bar{b}$ è la capacità residua dello zaino che tiene in conto delle variabili fissate a 1, e fissaggio basato sui costi ridotti.

La scelta del rilassamento continuo e dell'euristica CUD è dettata dal fatto che sono entrambe efficienti e "collaborano"; infatti, una volta ordinati gli oggetti dello zaino per costo unitario non crescente all'inizio dell'algoritmo, entrambe possono essere risolti in $O(n)$, e riutilizzando per l'euristica parte del lavoro svolto nel rilassamento. Il rilassamento continuo permette anche il fissaggio basato sui costi ridotti. Poichè la soluzione del rilassamento è intera tranne per la variabile "critica" h (la prima variabile a non entrare completamente nello zaino) al più, ed ottima se x_h è intera, l'ovvia regola di branching è quella di fissare x_h rispettivamente a 0 ed 1 nei figli del nodo. Questa strategia partiziona lo spazio delle soluzioni, costruisce esattamente due figli per nodo e rende inammissibile la soluzione ottima del rilassamento. La strategia è anche "teoricamente" bilanciata, ma in pratica ciò può non risultare del tutto vero. Infatti, in molte istanze del problema dello zaino la soluzione ottima contiene "pochi" oggetti, per cui fissare un oggetto come facente parte della soluzione può avere un effetto maggiore che escluderlo dalla soluzione (può esservi un numero minore di soluzioni ammissibili nel primo caso che nel secondo). La visita depth-first permette di implementare l'algoritmo in modo molto efficiente. Ogni volta che si scende (o si sale) di un livello nell'albero delle decisioni la capacità residua $\bar{b}$ può diminuire (aumentare), oppure rimanere costante. La soluzione ottima del rilassamento può essere determinata molto efficientemente partendo dalla soluzione ottima del rilassamento nel nodo padre. Per via della potenziale asimmetria tra il fissare una variabile a 0 e ad 1, può essere consigliabile generare (e quindi visitare) per primo il nodo in cui la variabile viene fissata ad 1.

Esercizio 7.10 *Si discuta nei dettagli come implementare questo algoritmo di enumerazione implicita in modo che risulti il più efficiente possibile, o meglio ancora lo si implementi e si verifichi nella pratica l'impatto delle diverse scelte implementative.*

Esempio 7.2: B&B per lo zaino

Si consideri la seguente istanza del problema dello zaino:

$$\begin{array}{rrcrcrcrcl} \max & 11x_1 & + & 8x_2 & + & 7x_3 & + & 6x_4 & & \\ & 5x_1 & + & 4x_2 & + & 4x_3 & + & 4x_4 & \leq & 12 \\ & x_1 & , & x_2 & , & x_3 & , & x_4 & \in & \{0,1\} \end{array} .$$

Ad ogni nodo dell'albero delle decisioni denotiamo con $\bar{x}$ la soluzione ottima del rilassamento continuo (si noti che gli oggetti sono già ordinati per costo unitario non crescente) e $\bar{z}$ la corrispondente valutazione superiore, con $\underline{x}$ la soluzione determinata dall'euristica CUD e $\underline{z}$ la corrispondente valutazione inferiore, e con c^* il vettore dei costi ridotti; si noti che

$$c_i^* = c_i - y^* a_i = c_i - \frac{c_h}{a_h} a_i = \left(\frac{c_i}{a_i} - \frac{c_h}{a_h} \right) a_i$$

(si vedano i paragrafi 5.1.2.1 e 6.1.2.2). Al nodo radice dell'albero delle decisioni abbiamo quindi

$$\bar{x} = [\,1\,,\,1\,,\,0.75\,,\,0\,] \quad , \quad \bar{z} = 24.25 \quad , \quad \underline{x} = [\,1\,,\,1\,,\,0\,,\,0\,] \quad , \quad \underline{z} = z = 19 \ .$$

Il gap assoluto al nodo radice è quindi di 5.25. Essendo $c^* = [\,2.25\,,\,1\,,\,0\,,\,-1\,]$, nessun costo ridotto è in valore assoluto maggiore del gap è non è quindi possibile fissare variabili. Occorre quindi effettuare l'operazione di branching sulla variabile frazionaria x_3.

Il successivo nodo estratto da Q corrisponde quindi al problema in cui $x_3 = 1$, ossia

$$\begin{array}{rrcrcrcl} 7 + \max & 11x_1 & + & 8x_2 & + & 6x_4 & & \\ & 5x_1 & + & 4x_2 & + & 4x_4 & \leq & 8 \\ & x_1 & , & x_2 & , & x_4 & \in & \{0,1\} \end{array} .$$

Il rilassamento continuo e l'euristica forniscono rispettivamente

$$\bar{x} = [\,1\,,\,0.75\,,\,1\,,\,0\,] \quad , \quad \bar{z} = 24 \quad , \quad \underline{x} = [\,1\,,\,0\,,\,1\,,\,0\,] \quad , \quad \underline{z} = 18 < z = 19$$

per cui il gap al nodo scende a 5 (la soluzione candidata non viene modificata); dato che $c^* = [\,1\,,\,0\,,\,*\,,\,-2\,]$ (la variabile x_3 è fissata e quindi non compare nel rilassamento continuo, pertanto non ha costo ridotto), nessuna variabile viene fissata ed è necessario effettuare l'operazione di branching sulla variabile frazionaria x_2.

Il successivo nodo estratto da Q corrisponde quindi al problema in cui $x_2 = x_3 = 1$, $\bar{b} = 4$. Poichè $a_1 = 5 > \bar{b}$ viene quindi fissata anche $x_1 = 0$. Il rilassamento continuo e l'euristica forniscono quindi

$$\bar{x} = \underline{x} = [\,0\,,\,1\,,\,1\,,\,1\,] \quad , \quad \bar{z} = \underline{z} = z = 21$$

ed il nodo viene potato per ottimalità, mentre viene aggiornata la soluzione candidata. Si noti che non effettuare il preprocessing, ossia non fissare $x_1 = 0$, avrebbe comportato ottenere $\bar{x} = [\,0.80\,,\,1\,,\,1\,,\,0\,]$ e $\bar{z} = 23.8$ e quindi non poter potare il nodo.

Si esegue quindi un "backtrack" e si passa ad esaminare il problema in cui $x_2 = 0$, $x_3 = 1$, per cui il rilassamento continuo e l'euristica forniscono

$$\bar{x} = [\,1\,,\,0\,,\,1\,,\,0.75\,] \quad , \quad \bar{z} = 22.5 \quad , \quad \underline{x} = [\,1\,,\,0\,,\,1\,,\,0\,] \quad , \quad \underline{z} = 18 < z = 21$$

corrispondenti ad un gap di 1.5. Si noti che in questo momento si ha una valutazione superiore sull'ottimo del problema in cui $x_3 = 1$ pari a $\max\{\,21\,,\,22.5\,\} = 22.5$, ossia migliore della valutazione superiore di 24 ottenuta dal rilassamento

continuo. Poichè si ha $c^* = [\,2.75\,,\,*\,,\,*\,,\,0\,]$, è possibile fissare ad 1 la variabile x_1: infatti, $22.5 - 2.75 = 19.75 < 21$. Si noti che 19.75 è una valutazione superiore del valore del rilassamento continuo in cui $x_1 = 0$; tale valutazione è molto approssimata (ma corretta) in quanto la soluzione ottima di tale rilassamento è $[\,0\,,\,0\,,\,1\,,\,1\,]$ che ha costo 13. Fissare $x_1 = 1$ lascia $\bar{b} = 3$, e quindi porta a fissare anche $x_4 = 0$. Siamo così giunti ad una foglia dell'albero delle decisioni, ossia a dimostrare che nessuna soluzione nel sottoalbero corrispondente a $x_2 = 0$, $x_3 = 1$ ha costo migliore di 18. È così terminata l'esplorazione dell'intero sottoalbero corrispondente ad $x_3 = 1$, per il quale si ha adesso $\bar{z} = \underline{z} = 21$.
Si esegue quindi un "backtrack" e si passa ad esaminare il problema in cui $x_3 = 0$, per cui il rilassamento continuo e l'euristica forniscono

$$\bar{x} = [\,1\,,\,1\,,\,0\,,\,0.75\,] \quad , \quad \bar{z} = 23.5 \quad , \quad \underline{x} = [\,1\,,\,1\,,\,0\,,\,0\,] \quad , \quad \underline{z} = 19 < z = 21 \;\;;$$

abbiamo quindi $z(P) \leq \max\{\,21\,,\,23.5\,\} = 23.5$, e quindi possiamo affermare che la soluzione candidata ha un gap massimo di 2.5 rispetto alla soluzione ottima. Poichè si ha $c^* = [\,2.75\,,\,2\,,\,*\,,\,0\,]$, è possibile fissare $x_1 = 1$: infatti, $23.5 - 2.75 = 20.75 < 21$. Ancora una volta si noti che 20.75 è una valutazione superiore (approssimata) del valore del rilassamento continuo in cui $x_1 = 0$, che ha soluzione ottima $[\,0\,,\,1\,,\,0\,,\,1\,]$ con costo 14. Comunque questo non permette di terminare l'esplorazione, per cui si deve eseguire il branching sulla variabile frazionaria x_4.
Il nodo successivo corrisponde al problema in cui $x_1 = x_4 = 1$, $x_3 = 0$. Poichè $\bar{b} = 3 > a_2 = 4$ si può fissare $x_2 = 0$ e si è raggiunta la foglia dell'albero delle decisioni corrispondente alla soluzione $[\,1\,,\,0\,,\,0\,,\,1\,]$ di costo $17 < z = 21$. Poiché non ci sono più decisioni da prendere si esegue un "backtrack", ossia si passa ad esaminare il nodo successivo.
Il nodo corrisponde al problema in cui $x_1 = 1$, $x_3 = x_4 = 0$. In questo caso si ha

$$\bar{x} = \underline{x} = [\,1\,,\,1\,,\,0\,,\,0\,] \quad , \quad \bar{z} = \underline{z} = z = 19$$

ed il nodo viene potato per ottimalità. A questo punto abbiamo $Q = \emptyset$, e $z(P) \leq \max\{\,21\,,\,19\,\} = 21$; infatti, la soluzione candidata $x = [\,0\,,\,1\,,\,1\,,\,1\,]$ è ottima.

Questo algoritmo di enumerazione implicita è un caso particolare del *Branch and Bound basato sulla PL*, che è uno tra gli algoritmi più generali per risolvere problemi di *OC*. L'algoritmo risolve in tempo finito qualsiasi problema di Programmazione Lineare Intera (o mista)

$$(P) \quad \max\{\, cx \;:\; Ax \leq b \;,\; x \in \mathbb{Z}^n \,\}$$

utilizzando un metodo di enumerazione implicita che usa il rilassamento continuo per produrre valutazioni superiori. Il branching viene effettuato sulle variabili frazionarie: scelta una variabile x_i frazionaria nella soluzione ottima x^* del rilassamento continuo, si costruiscono i due figli del nodo nell'albero delle decisioni semplicemente aggiungendo alla formulazione del problema i vincoli

$$x_i \leq \lfloor\, x_i^* \,\rfloor \qquad \text{oppure} \qquad x_i \geq \lceil\, x_i^* \,\rceil \;.$$

Questi vincoli equipartizionano (in teoria) l'insieme ammissibile e sono soddisfatti da tutte le soluzioni intere ma non da x^*. Si noti che questa regola di branching generalizza il fissare a 0 oppure 1 una variabile binaria che abbia valore frazionario nel rilassamento continuo. In presenza di strutture specifiche nel modello si possono comunque utilizzare anche regole di branching diverse, come per il caso degli Special Ordered Sets descritti nel paragrafo 7.2.4. Sfruttando le opportunamente le capacità di riottimizzazione degli algoritmi del simplesso (si veda il paragrafo 7.2.1) e strategie di visita opportune si può implementare questo algoritmo in modo che sia in grado di valutare in modo piuttosto efficiente ciascun nodo nell'albero delle decisioni.
I problemi principali di questo algoritmo sono la mancanza di euristiche generali per la *PLI* e la debolezza delle valutazioni superiori ottenute dal rilassamento continuo in molti casi (si veda il paragrafo 6.1.1). Per il primo problema, le tecniche di arrotondamento (si veda il paragrafo 6.1.2.1) generali spesso hanno scarsa efficacia; recentemente sono quindi state sviluppate euristiche più sofisticate (*feasibility pump*, *RINS*, ...) che hanno invece dimostrato buone prestazioni in molti casi, contribuendo a migliorare in modo significativo le prestazioni dei solutori "general-purpose" basati su queste tecniche. Per il secondo problema sono molto utili le tecniche poliedrali discusse al paragrafo 7.5. Altri problemi di questo approccio riguardano il fatto che gli algoritmi per la *PL* sono comunque relativamente costosi, sia in tempo che in memoria, quando la dimensione del modello cresce, ad esempio rispetto ad altri tipi di problema come quelli di flusso su rete.
In generale il B&B basato sulla *PL* riesce a risolvere efficientemente problemi di piccole dimensioni, ed in diversi casi anche problemi di dimensioni superiori. Esistono implementazioni estremamente

sofisticate ed efficienti dell'approccio che, sfruttando lo stato dell'arte nel campo della *PL* e delle tecniche poliedrali generali, riescono spesso a risolvere problemi di *PLI* di dimensioni rilevanti senza necessità di interventi da parte dell'utente. Per contro, in moltissimi casi i problemi di *OC* derivanti da modelli realistici di problemi decisionali concreti non vengono risolti con sufficiente efficienza anche dai migliori di questi approcci, almeno non senza un rilevante intervento da parte di esperti in grado di sfruttare al meglio le proprietà del problema da risolvere.

7.3.2 Il problema del commesso viaggiatore

Discutiamo adesso un algoritmo di enumerazione implicita per il problema del commesso viaggiatore che utilizzi (MS1-T) come rilassamento e "twice around MST" come euristica. Questa versione base potrebbe poi essere modificata utilizzando

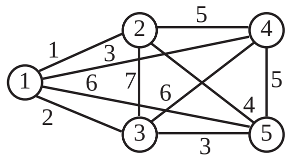
- tecniche di ricerca locale per migliorare la soluzione determinata dall'euristica;
- il rilassamento Lagrangiano avente (MS1-T) come sottoproblema, possibilmente trasformando quindi l'euristica in un euristica Lagrangiana.

Per la regola di branching supponiamo quindi di avere a disposizione un 1-albero ottimo T^*: dobbiamo prendere decisioni sul problema in modo da rendere T^* non ammissibile. Se ogni nodo in T^* ha esattamente due lati incidenti allora T^* è un ciclo Hamiltoniano, e quindi è la soluzione ottima del problema; possiamo quindi supporre che esista un nodo i in cui incidano più di due lati di T^*. Sia $L = \{ l_1 , \dots , l_k \}$ $(k > 2)$ l'insieme di questi lati. Un semplice modo per effettuare il branching corrisponde a selezionare un qualsiasi $l_i \in L$ ed a costruire due figli, uno nel quale si è deciso che l_i faccia parte del ciclo Hamiltoniano e l'altro in cui si è deciso che non ne faccia parte. Questa regola è semplice da implementare, potenzialmente equipartiziona l'insieme (ma si noti che ogni nodo ha solo due lati incidenti nel ciclo e molti lati incidenti che non appartengono al ciclo, quindi in uno dei due figli viene presa una decisione "più forte") e crea esattamente due figli per nodo. Per contro, nel figlio in cui si decide che il lato fa parte del ciclo, la soluzione ottima del rilassamento non cambia.
Una regola che assicura l'inammissibilità di T^* in tutti i figli è la seguente. Vengono costruiti $k(k-1)/2+k+1$ figli del nodo. Nei primi $k(k-1)/2$ vengono fissati come appartenenti al ciclo Hamiltoniano tutte le possibili coppie di lati nell'insieme L; di conseguienza, una volta fissata una qualsiasi coppia tutti gli altri lati incidenti in i vengono fissati come non appartenenti al ciclo. Nei successivi k figli si fissa uno dei lati $l_i \in L$ come appartenente al ciclo e tutti gli altri come non appartenenti. Nell'ultimo figlio si fissano tutti i lati in L come non appartenenti al ciclo. Questa regola partiziona l'insieme, ma ha diversi svantaggi: è piuttosto complessa da implementare, fissa lati come appartenenti al ciclo e quindi costringe ed operazioni complesse per adattare euristica e rilassamento (si veda l'esempio 7.6) e può produrre molti figli.
Una regola di branching ancora diversa prevede di costruire k figli del nodo: in ciascuno dei figli si fissa uno dei lati $l_i \in L$ come *non* appartenente al ciclo, e non si prende alcuna altra decisione. Questa regola assicura l'inammissibilità di T^*, è relativamente semplice da implementare, fissa solamente lati come non appartenenti al ciclo (il che semplifica molto l'adattamento di rilassamento ed euristica) e produce un numero minore di figli rispetto alla precedente. Per contro non partiziona l'insieme: qualsiasi ciclo che non contenga nessuno degli archi in L appartiene a tutti i figli. Comunque quest'ultima regola di branching è probabilmente la più ragionevole di quelle proposte; si noti che, per limitare ulteriormente il numero di figli da generare, è possibile scegliere, tra tutti i nodi in cui incidano più di due lati di T^*, quello in cui ne incide il minor numero. Dato che comunque il numero di figli può essere alto, può essere ragionevole utilizzare una strategia di visita di tipo depth-first. Per questo algoritmo non proponiamo nessuna specifica tecnica di preprocessing.

Esercizio 7.11 *Si propongano e discutano altre regole di branching e tecniche di preprocessing per questo algoritmo. Si implementi l'algoritmo e si verifichi nella pratica l'impatto delle diverse scelte implementative.*

Esempio 7.3: B&B per il (TSP)

Si consideri l'istanza del problema (TSP) illustrata in figura qui accanto. Al nodo radice si ottiene l'1-albero mostrato in figura 7.3(a) (l'arco tratteggiato è quello aggiunto al (MST)), di costo $\underline{z} = 13$, e quindi l'euristica ottiene il ciclo mostrato a fianco, di costo $\bar{z} = 21$. Si ha quindi un gap di 8, e bisogna effettuare l'operazione di branching. Per questo si seleziona il nodo 1, che ha tre archi incidenti nell'1-albero, e si costruiscono tre figli, nei quali sono escluso dal ciclo rispettivamente il lato $\{1, 2\}$, il lato $\{1, 4\}$ ed il lato $\{1, 3\}$.



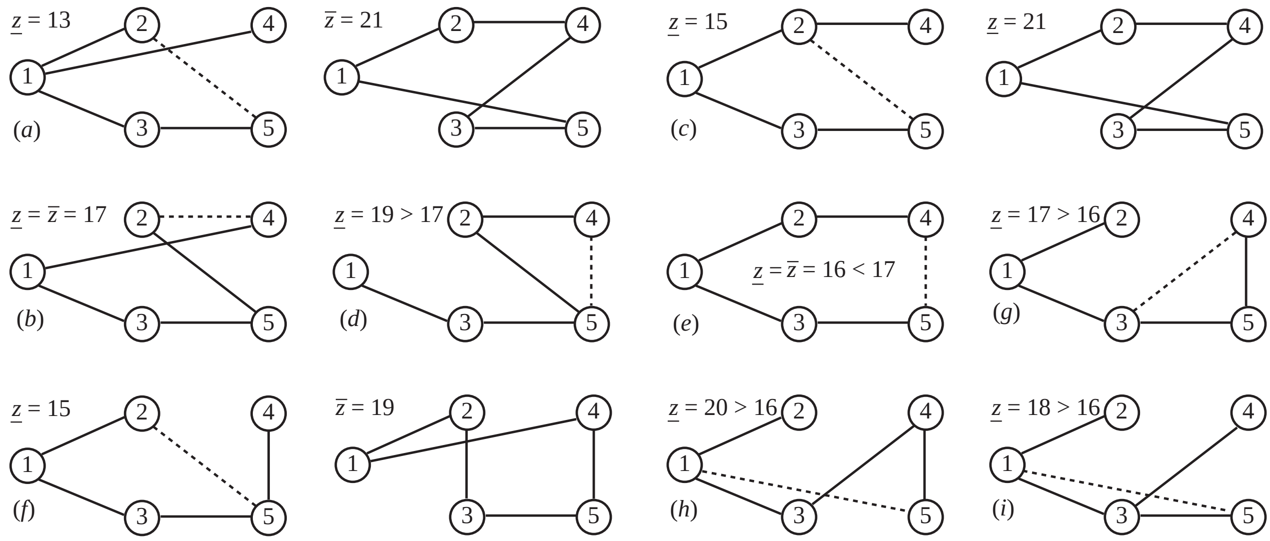


Figura 7.3: Algoritmo *B&B* per il (TSP)

$\boxed{x_{12} = 0}$: l'1-albero corrispondente, mostrato in figura 7.3(b), è un ciclo Hamiltoniano di costo 17. Viene quindi aggiornata la soluzione candidata, ed il nodo è potato per ottimalità.

$\boxed{x_{14} = 0}$: si ottengono l'1-albero ed il ciclo Hamiltoniano mostrati in figura 7.3(c); poiché il problema non è stato risolto e $\underline{z} = 15 < z = 17$, si effettua l'operazione di branching. Per questo si seleziona il nodo 2, che ha tre archi incidenti nell'1-albero, e si costruiscono i tre figli nei quali sono esclusi dal ciclo rispettivamente i lati $\{1, 2\}$, $\{2, 5\}$ e $\{2, 4\}$.

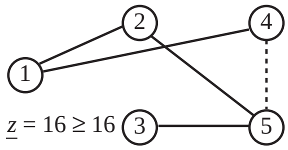
$\boxed{x_{14} = x_{12} = 0}$: si ottiene l'1-albero mostrato in figura 7.3(d), di costo $19 > z = 17$, e quindi il nodo viene potato dalla valutazione superiore.

$\boxed{x_{14} = x_{25} = 0}$: si ottiene l'1-albero mostrato in figura 7.3(e), che è un ciclo Hamiltoniano di costo 16; viene quindi aggiornata la soluzione candidata, ed il nodo è potato per ottimalità.

$\boxed{x_{14} = x_{24} = 0}$: si ottengono l'1-albero ed il ciclo Hamiltoniano mostrati in figura 7.3(f); poiché il problema non è stato risolto e $\underline{z} = 15 < z = 16$, si effettua l'operazione di branching. Per questo si seleziona il nodo 5, che ha tre archi incidenti nell'1-albero, e si costruiscono i tre figli nei quali sono esclusi dal ciclo rispettivamente i lati $\{2, 5\}$, $\{3, 5\}$ e $\{4, 5\}$.

Per tutti e tre i nodi appena generati si ottengono 1-alberi, mostrati rispettivamente in figura 7.3(g), 7.3(h) e 7.3(i), che hanno costo maggiore di $z = 16$; tutti e tre i nodi vengono quindi potati dalla valutazione superiore, e si risale nell'albero delle decisioni (si esegue un "backtrack") di due livelli.

$\boxed{x_{13} = 0}$: si ottiene l'1-albero mostrato in figura 7.3(e); poiché il costo dell'1-albero è pari a $z = 16$, non possono esistere in questo sottoalbero dell'albero delle decisioni soluzioni di costo inferiore a 16 (si noti che in principio potrebbero essercene di costo uguale) ed il nodo viene potato dalla valutazione superiore. Siccome $Q = \emptyset$ l'algoritmo termina, dimostrando che la soluzione candidata, mostrata qui accanto, è ottima per il problema.



7.3.3 Il problema del cammino minimo vincolato

Per questo problema, introdotto nel paragrafo 6.2.1.2, analizzeremo un algoritmo enumerativo che usa come rilassamento un problema di cammino minimo, ossia il rilassamento per eliminazione del vincolo sulla lunghezza dei cammini. Analoghe considerazioni potrebbero poi essere fatte per il corrispondente rilassamento Lagrangiano. In questo caso decidiamo di non utilizzare nessuna euristica: le soluzioni ammissibili verranno fornite dal rilassamento stesso, o al limite dalla visita di foglie dell'albero delle decisioni. Ciò suggerisce l'utilizzo della regola di visita depth-first.

Dobbiamo quindi decidere un'opportuna regola di branching. Il rilassamento produce un cammino P tra l'origine e la destinazione: se il cammino rispetta il vincolo di lunghezza allora è ottimo e non occorre effettuare l'operazione di branching. Possiamo quindi supporre di avere, al momento di dover effettuare il branching, un cammino P di lunghezza maggiore della soglia L. Abbiamo già discusso come la regola di branching "standard", ossia selezionare un arco di P e fissarlo come appartenente o no al cammino ottimo, non sia applicabile, in quanto la sua applicazione ripetuta porterebbe a dover risolvere nei nodi interni dell'albero delle decisioni problemi di cammino minimo vincolato (in cui il vincolo è di passare per un insieme di archi dato) sostanzialmente difficili come il problema che si vuole risolvere. Inoltre questa regola, al momento in cui fissa l'arco come appartenente al cammino, non rende inammissibile la soluzione ottima del rilassamento. Una regola di branching alternativa può essere ottenuta come nell'esempio precedente: si costruiscono tanti figli quanti sono gli archi di P, ed in ciascun figlio si fissa come non appartenente al cammino ottimo uno degli archi. Questa regola rende inammissibile la soluzione ottima del rilassamento, ma ha gli inconvenienti già sottolineati nel caso dell'esempio precedente: può produrre un numero di figli elevato, e soprattutto non partiziona l'insieme ammissibile. Un vantaggio di questa regola è invece che, decidendo solo per l'esclusione di archi, rende molto facile adattare il rilassamento a lavorare nei nodi interni: è sufficiente eliminare gli archi dal grafo su cui si calcola il cammino minimo.

Per questo problema possiamo però proporre una regola alternativa che produce un numero di figli inferiore ed ha il vantaggio di partizionare l'insieme delle soluzioni, anche se al costo di complicare l'implementazione del rilassamento. La regola funziona come segue: dato il cammino P tra r e t, che viola il vincolo sulla massima lunghezza, si costruisce un primo figlio in cui il primo arco del cammino è fissato come non appartenente al cammino ottimo. Nel secondo figlio si fissa invece il primo arco del cammino come appartenente al cammino ottimo, ed il secondo arco del cammino come non appartenente al cammino ottimo. Nel terzo figlio si fissano i primi due archi del cammino come appartenenti al cammino ottimo ed il terzo come non appartenente, e così via finché non si giunge al primo arco del cammino tale che il sottocammino che parte da r e contiene tutti i primi archi di P, fino a quello, viola il vincolo di lunghezza. L'ultimo figlio è quindi quello in cui l'arco "critico" è fissato come non appartenente al cammino ottimo e tutti i precedenti sono fissati come appartenenti: non è necessario generare altri figli, perchè nessun cammino ammissibile può contenere altri sottocammini di P che comprendono r. Questa regola di branching è un caso particolare di una regola che si può applicare per problemi con variabili $x_i \in \{0, 1\}$, in cui nella soluzione ottima x^* del rilassamento si abbia che l'insieme $S = \{i : x^*_i = 1\} = \{i_1, i_2, \ldots i_k\}$ contiene "troppe" variabili. La regola di branching crea un figlio con $x_{i_1} = 0$, un figlio con $x_{i_1} = 1$ e $x_{i_2} = 0$, un figlio con $x_{i_1} = x_{i_2} = 1$ e $x_{i_3} = 0$ e così via. Nel caso di (CSP) non è necessario generare tutti i figli in quanto alcuni di essi sono certamente non ammissibili. Si noti che questa regola non è bilanciata. Questa regola di branching fissa archi come appartenenti al cammino, e quindi rischierebbe di incorrere negli stessi problemi di incompatibilità col rilassamento della regola standard. Ma in questo caso non sono fissati archi qualsiasi: vengono sempre fissati interi sottocammini che partono da r. È quindi possibile adattare il rilassamento semplicemente rimuovendo tutti i nodi del cammino fissato dal grafo, tranne il nodo terminale, e rendendo il nodo terminale del cammino la radice del cammino minimo. Contemporaneamente occorre aggiornare la soglia di massima lunghezza L sottraendo ad essa la lunghezza del sottocammino fissato: a questo punto si può anche applicare un'operazione di preprocessing eliminando dal grafo tutti gli archi che abbiano $l_{ij} > L'$, dove L' è la soglia aggiornata.

Esempio 7.4: B&B per (CSP)

Applichiamo l'algoritmo *B&B* all'istanza dell'Esempio 6.1. Al nodo radice si ha la situazione rappresentata in figura 7.4(a): il cammino P (archi evidenziati) ha costo 3 ma lunghezza $3 > L = 2$. Poiché non si è determinata nessuna soluzione ammissibile si ha $z = +\infty$, e quindi un gap infinito: si applica quindi l'operazione di branching.

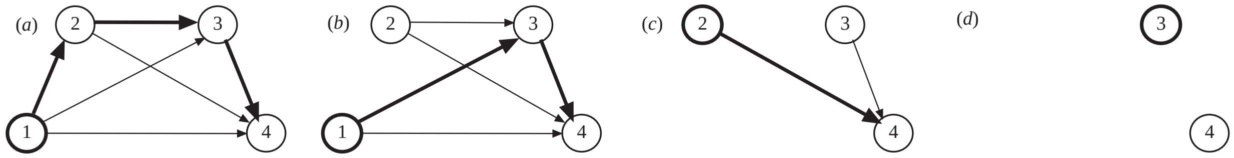


Figura 7.4: Algoritmo *B&B* per il (CSP)

$\boxed{x_{12} = 0}$: il corrispondente cammino minimo è mostrato in figura 7.4(b); il cammino ha costo 4 ed è ammissibile, per cui si pone $z = 4$ ed il nodo viene potato per ottimalità.

$\boxed{x_{12} = 1,\ x_{23} = 0}$: si sposta la radice r al nodo 2 ponendo $L = 1$, ottenendo il cammino ottimo illustrato in figura 7.4(c) che ha costo $5 > z = 4$: il nodo viene quindi potato dalla valutazione inferiore.

$\boxed{x_{12} = x_{23} = 1,\ x_{34} = 0}$: si sposta la radice r al nodo 3 e si pone $L = 0$, come illustrato in figura 7.4(d); sul grafo non esistono cammini tra 3 e 4, quindi il nodo viene potato per inammissibilità. Essendo Q vuoto l'algoritmo termina riportando come soluzione ottima il cammino di costo 4 mostrato in figura 7.4(b).

Esercizio 7.12 *Si discuta se e come sia possibile adattare gli algoritmi per il problema dei cammini minimi studiati nel paragrafo 3.2 affinché sfruttino la conoscenza del cammino e delle etichette ottime corrispondenti al padre per risolvere in modo più efficiente i rilassamenti nei nodi figli.*

Esercizio 7.13 *Si propongano algoritmi di enumerazione implicita per tutti i problemi di* OC *visti in queste dispense, proponendo per ciascuno uno o più rilassamenti, euristiche, regole di branching, operazioni di preprocessing e così via, e discutendo i pró ed i contro delle diverse alternative.*

7.4 Programmazione dinamica

Work in progress.

7.5 Tecniche poliedrali

Work in progress.

Riferimenti Bibliografici

F. Maffioli "**Elementi di Programmazione Matematica**", *Casa Editrice Ambrosiana*, 2000.

A. Sassano "**Modelli e Algoritmi della Ricerca Operativa**", *FrancoAngeli*, 1999.

L. Wolsey "**Integer Programming**", *Wiley-Interscience*, 1998.

K. Marriott, P.J. Stuckey "**Programming with Constraints: An Introduction**", *MIT Press*. 1998.

Appendice A

Algoritmi e complessità

In questa appendice vogliamo brevemente richiamare alcuni concetti fondamentali della teoria della complessità computazionale, utili per meglio comprendere la diversa "difficoltà" della soluzione dei problemi di ottimizzazione.

A.1 Modelli computazionali

Una volta che un problema P sia stato formulato, deve essere risolto: siamo quindi interessati alla messa a punto di strumenti di calcolo che, data una qualsiasi istanza p, siano in grado di fornirne una soluzione in un tempo finito. Tali strumenti di calcolo si chiamano *algoritmi*. Un algoritmo che risolve P può essere definit come una sequenza finita di istruzioni che, applicata ad una qualsiasi istanza p di P, si arresta dopo un numero finito di passi (ovvero di computazioni elementari), fornendo una soluzione di p oppure indicando che p non ha soluzioni ammissibili. Per poter studiare gli algoritmi dal punto di vista della loro efficienza, o complessità computazionale, è necessario definire un modello computazionale: classici modelli computazionali sono la Macchina di Turing (storicamente il primo proposto), la R.A.M. (Random Access Machine), la Macchina a Registri (MR), etc.
La Macchina a Registri è un buon compromesso tra semplicità e versatilità: una MR consiste di un numero (finito ma non limitato) di registri, ciascuno dei quali può contenere un singolo numero intero, e di un programma, ossia di una sequenza finita di istruzioni del tipo

- incrementa il registro k e salta all'istruzione j;
- decrementa il registro k e salta all'istruzione j;
- se il registro k contiene 0 salta all'instruzione j, altrimenti salta all'istruzione h.

Si tratta di una macchina *sequenziale e deterministica*, poiché il comportamento futuro della macchina è univocamente determinato dalla sua configurazione presente. Una MR è un buon modello astratto di un calcolatore elettronico, ed è quindi in grado di compiere tutte le computazioni possibili in un qualunque sistema di calcolo attualmente noto (se si eccettuano i computer quantistici, la cui implementabilità è comunque ancora da dimostrare). D'altra parte, si può dimostrare che la classe delle funzioni computabili da una MR è equivalente alla classe delle funzioni computabili da una Macchina di Turing, il che, secondo la *Tesi di Church*, implica che le MR siano presumibilmente in grado di calcolare qualsiasi funzione effettivamente computabile con procedimenti algoritmici.

A.2 Misure di complessità

Dato un problema P, una sua istanza p, e un algoritmo A che risolve P, indichiamo con costo (o complessità) di A applicato a p una misura delle risorse utilizzate dalle computazioni che A esegue su una macchina MR per determinare la soluzione di p. Le risorse, in principio, sono di due tipi, *memoria occupata e tempo di calcolo*: nell'ipotesi che tutte le operazioni elementari abbiano la stessa durata, il tempo di calcolo può essere espresso come numero di operazioni elementari effettuate dall'algoritmo. Poiché molto spesso la risorsa più critica è il tempo di calcolo, nel seguito useremo soprattutto questa come misura della complessità degli algoritmi.
Dato un algoritmo, è opportuno disporre di una misura di complessità che consenta una valutazione sintetica della sua bontà ed eventualmente un suo agevole confronto con algoritmi alternativi. Conoscere la complessità di A per ognuna delle istanze di P non è possibile (l'insieme delle istanze di un problema è normalmente infinito), né sarebbe di utilità pratica: si cerca allora di esprimere la complessità come una funzione $g(n)$ della dimensione, n, dell'istanza cui viene applicato l'algoritmo. Poiché, per ogni dimensione, si hanno in generale molte istanze di quella dimensione, si sceglie $g(n)$ come il costo necessario per risolvere la più difficile tra le istanze di dimensione n: si parla allora di *complessità nel caso peggiore*.
Bisogna naturalmente definire in modo preciso il significato di dimensione di una istanza: chiameremo dimensione di p una misura del numero di bit necessari per rappresentare, con una codifica "ragionevolmente" compatta, i dati che definiscono p, cioè una misura della lunghezza del suo input. Per esempio, in un grafo con n nodi e m archi i nodi possono essere rappresentati dagli interi tra 1 ed n e gli archi per mezzo di una lista contenente m coppie di interi (l'arco che

collega i nodi i e j è rappresentato dalla coppia (i, j)): trascurando le costanti moltiplicative, potremo allora assumere come misura della dimensione della codifica del grafo, al variare del numero dei nodi e degli archi, la funzione $m \log n$, dato che interi positivi e non superiori ad n possono essere rappresentati con $\log n$ bit[1]. Nel seguito, per semplicità, oltre alle costanti moltiplicative trascureremo anche le funzioni sublineari, come la funzione logaritmo; diremo allora che m è la lunghezza dell'input per un grafo con m archi. Nelle ipotesi fatte, la misura della lunghezza dell'input non varia se usiamo una codifica in base $b > 2$: se invece si usasse una codifica unaria, la lunghezza dell'input nell'esempio in questione diventerebbe nm, aumentando considerevolmente.
A questo punto la funzione $g(n)$, introdotta precedentemente, risulta definita in modo sufficientemente rigoroso: in pratica essa continua però ad essere di difficile uso come misura della complessità, dato che risulta difficile, se non praticamente impossibile, la valutazione di $g(n)$ per ogni dato valore di n. Questo problema si risolve sostituendo alla $g(n)$ il suo ordine di grandezza: si parlerà allora di *complessità asintotica*. Data una funzione $g(x)$, diremo che:

1. $g(x)$ è $O(f(x))$ se esistono due costanti c_1 e c_2 per cui, per ogni x, è $g(x) \leq c_1 f(x) + c_2$;
2. $g(x)$ è $\Omega(f(x))$ se $f(x)$ è $O(g(x))$;
3. $g(x)$ è $\Theta(f(x))$ se $g(x)$ è allo stesso tempo $O(f(x))$ e $\Omega(f(x))$.

Sia $g(x)$ il numero di operazioni elementari che vengono effettuate dall'algoritmo A applicato alla più difficile istanza, tra tutte quelle che hanno lunghezza di input x, di un dato problema P: diremo che la complessità di A è un $O(f(x))$ se $g(x)$ è un $O(f(x))$; analogamente, diremo che la complessità di A è un $\Omega(f(x))$ o un $\Theta(f(x))$ se $g(x)$ è un $\Omega(f(x))$ o un $\Theta(f(x))$.

A.3 Problemi trattabili e problemi intrattabili

Chiameremo *trattabili* i problemi per cui esistono algoritmi la cui complessità sia un $O(p(x))$, con $p(x)$ un polinomio in x, e *intrattabili* i problemi per cui un tale algoritmo non esiste: le seguenti tabelle chiariscono il perché di questa distinzione. In questa tabella vengono forniti, per diverse funzioni di complessità f, i tempi di esecuzione (in secondi, ove non diversamente specificato) per alcuni valori di n su un calcolatore che richieda $1e^{-6}$ secondi per effettuare un'operazione elementare.

f	10	20	40	60
n	$1e^{-5}$	$2e^{-5}$	$4e^{-5}$	$6e^{-5}$
n^3	$1e^{-3}$	$8e^{-3}$	$7e^{-2}$	$2e^{-1}$
n^5	$1e^{-1}$	3.2	1.7 min.	13 min.
2^n	$1e^{-3}$	1	13 giorni	36600 anni
3^n	$6e^{-2}$	1 ora	$4e^5$ anni	$1e^{13}$ anni

In questa tabella vengono invece indicati i miglioramenti ottenibili, in termini di dimensioni delle istanze risolvibili, per diverse funzioni di complessità, al migliorare della tecnologia dei calcolatori: con x_i abbiamo indicato la dimensione di un'istanza risolvibile oggi in un minuto per la i-esima funzione di complessità.

f	**Computer odierno**	**100 volte più veloce**	**10000 volte più veloce**
n	x_1	$100x_1$	$10000x_1$
n^3	x_2	$4.6x_2$	$21.5x_2$
n^5	x_3	$2.5x_3$	$6.3x_3$
2^n	x_4	$x_4 + 6.6$	$x_4 + 13.2$
3^n	x_5	$x_5 + 4.2$	$x_5 + 8.4$

Molti problemi di rilevante importanza pratica sono trattabili: sono problemi per i quali disponiamo di efficienti algoritmi di complessità polinomiale. Per potere effettuare una più rigorosa classificazione dei diversi problemi, facciamo riferimento a problemi in forma decisionale.

A.3.1 Le classi $\mathcal{P}$ e $\mathcal{NP}$

Una prima importante classe di problemi è la classe $\mathcal{NP}$, costituita da tutti i problemi decisionali il cui problema di certificato associato può essere risolto in tempo polinomiale. In altri termini, i problemi in $\mathcal{NP}$ sono quelli per cui è possibile verificare efficientemente una risposta "si", perché è possibile decidere in tempo polinomiale se una soluzione x è ammissibile per il problema. Ad esempio, il problema della soddisfattibilità proposizionale (SAT), introdotto in 1.2.3.1, è un problema in $\mathcal{NP}$: dato un qualunque assegnamento di valori di verità, è possibile verificare se tale assegnamento rende vera la formula in tempo lineare nella sua dimensione.
Equivalentemente, si può definire $\mathcal{NP}$ come la classe di tutti i problemi decisionali risolubili in tempo polinomiale da una MR nondeterministica ($\mathcal{NP}$ va infatti inteso come $\mathcal{P}$olinomiale nel calcolo $\mathcal{N}$ondeterministico). Una MR nondeterministica è il modello di calcolo (astratto) in cui una MR, qualora si trovi ad affrontare un'operazione di salto condizionale,

[1] In generale, tranne quando sarà detto il contrario, se i dati di un problema sono costituiti da n numeri considereremo come limitato da $\log n$ il numero di bits necessari per la codifica in binario dei numeri stessi.

può eseguire *contemporaneamente* entrambi rami dell'operazione, e questo ricorsivamente per un qualsiasi numero di operazioni. In altre parole, i problemi in $\mathcal{NP}$ sono quelli per cui esiste una computazione di lunghezza polinomiale che può portare a costruire una soluzione ammissibile, se esiste, ma questa computazione può essere "nascosta" entro un insieme esponenziale di computazioni analoghe tra le quali, in generale, non si sa come discriminare.
Ad esempio, per SAT si può immaginare una MR nondeterministica che costruisca l'assegnamento di valori di verità ai letterali con una computazione in cui, sequenzialmente, viene assegnato il valore di verità a ciascun letterale, in un certo ordine prestabilito, in base ad una certa condizione logica. Questo identifica un *albero di computazione* che descrive tutte le possibili esecuzioni della MR, e le cui foglie sono tutti i 2^n possibili assegnamenti di valori di verità agli n letterali della formula. Se la formula è soddisfattibile, allora esiste un cammino nell'albero di computazione, di lunghezza lineare nel numero di letterali, che porta a costruire esattamente un certificato del problema, ossia un assegnamento di valori di verità che soddisfa la formula.

Un sottoinsieme della classe $\mathcal{NP}$ è la classe $\mathcal{P}$, costituita da tutti i *(problemi polinomiali)*, quei problemi decisionali per i quali esistono algoritmi di complessità polinomiale che li risolvono. Una domanda particolarmente importante è se esistano problemi in $\mathcal{NP}$ che non appartengano anche a $\mathcal{P}$. A questa domanda non si è a tutt'oggi stati in grado di rispondere, ma si ritiene fortemente probabile sia effettivamente $\mathcal{P} \neq \mathcal{NP}$; una breve giustificazione di questo sarà data nel paragrafo successivo.

A.3.2 Problemi $\mathcal{NP}$-completi e problemi $\mathcal{NP}$-ardui

Molti problemi, anche se apparentemente notevolmente diversi, possono tuttavia essere ricondotti l'uno all'altro; dati due problemi decisionali, P e Q, diciamo che P *si riduce in tempo polinomiale a* Q, e scriveremo $P \propto Q$, se, supponendo l'esistenza di un algoritmo A_Q che risolva Q in tempo costante (indipendente dalla lunghezza dell'input), esiste un algoritmo che risolve P in tempo polinomiale utilizzando come sottoprogramma A_Q: parliamo in tal caso di *riduzione polinomiale di* P a Q. Ad esempio, SAT si riduce polinomialmente alla *PLI*, come mostrato nel paragrafo 1.2.3.1. Quindi, se esistesse un algoritmo polinomiale per la *PLI* allora esisterebbe un algoritmo polinomiale per SAT: data la formula in forma normale congiuntiva, basterebbe produrre (in tempo polinomiale) il problema di *PLI* corrispondente, applicare l'algoritmo a tale problema e rispondere "si" o "no" a seconda della risposta ottenuta. Possiamo quindi affermare che $SAT \propto PLI$. È facile verificare che la relazione $\propto$ ha le seguenti proprietà:

1. è riflessiva: $A \propto A$;
2. è transitiva: $A \propto B$ e $B \propto C \Rightarrow A \propto C$;
3. se $A \propto B$ e $A \notin P$, allora $B \notin P$;
4. se $A \propto B$ e $B \in P$, allora $A \in P$.

Possiamo definire adesso la classe dei problemi $\mathcal{NP}$-completi: un problema A è detto $\mathcal{NP}$*-completo* se $A \in \mathcal{NP}$ e se per ogni $B \in \mathcal{NP}$ si ha che $B \propto A$. La classe dei problemi $\mathcal{NP}$-completi costituisce un sottoinsieme di $\mathcal{NP}$ di particolare importanza; un fondamentale teorema, dovuto a Cook (1971), garantisce che ogni problema $P \in \mathcal{NP}$ si riduce polinomialmente a SAT, ossia che tale classe non è vuota. I problemi $\mathcal{NP}$-completi hanno la proprietà che se esiste per uno di essi un algoritmo polinomiale, allora necessariamente tutti i problemi in $\mathcal{NP}$ sono risolubili polinomialmente, e quindi è $\mathcal{P} = \mathcal{NP}$; in un certo senso, tali problemi sono i "più difficili" tra i problemi in $\mathcal{NP}$. Un problema che abbia come caso particolare un problema $\mathcal{NP}$-completo ha la proprietà di essere almeno tanto difficile quanto i problemi $\mathcal{NP}$-completi (a meno di una funzione moltiplicativa polinomiale): un problema di questo tipo si dice $\mathcal{NP}$-arduo. Si noti che un problema $\mathcal{NP}$-arduo può anche non appartenere a $\mathcal{NP}$. Ad esempio, sono $\mathcal{NP}$-ardui i problemi di ottimizzazione la cui versione decisionale è un problema $\mathcal{NP}$-completo: infatti, come si è già visto, è sempre possibile ricondurre un problema decisionale ad un problema di ottimizzazione con una opportuna scelta della funzione obiettivo. Ad esempio, il problema della *PLI* è $\mathcal{NP}$-arduo, poichè ad esso si riduce SAT, che è $\mathcal{NP}$-completo.
Fino ad oggi, sono stati trovati moltissimi problemi $\mathcal{NP}$-ardui; si può affermare che la grande maggioranza dei problemi combinatori siano $\mathcal{NP}$-ardui. Nonostante tutti gli sforzi dei ricercatori, non è stato possibile determinare per nessuno di essi un algoritmo polinomiale (il che avrebbe fornito algoritmi polinomiali per tutti i problemi della classe); questo fa ritenere che non esistano algoritmi polinomiali per i problemi $\mathcal{NP}$-completi, ossia che sia $\mathcal{P} \neq \mathcal{NP}$. Ad oggi non si conosce nessuna dimostrazione formale di questa ipotesi, ma essa è suffragata da molti indizi. Ad esempio, $\mathcal{P}$ ed $\mathcal{NP}$ sono solo i primi elementi di una *gerarchia polinomiale* di classi di problemi, sempre "più difficili", che collasserebbero tutte sulla sola classe $\mathcal{P}$ qualora fosse $\mathcal{P} = \mathcal{NP}$, il che fa ritenere questa eventualità altamente improbabile.

Per riassumere, possiamo dire che, allo stato delle conoscenze attuali, tutti i problemi $\mathcal{NP}$-ardui sono presumibilmente intrattabili. Naturalmente, ciò non significa che non sia in molti casi possibile costruire algoritmi in grado di risolvere efficientemente istanze di problemi $\mathcal{NP}$-ardui di dimensione significativa (quella richiesta dalle applicazioni reali); questo non rientra comunque nel campo della teoria della complessità computazionale ma piuttosto nel campo dell'Ottimizzazione Combinatoria (si veda il Capitolo 4 e seguenti).

A.3.3 Complessità ed approssimazione

Esistono molti altri risultati interessanti della teoria della complessità computazionale che non rientrano nello scopo di queste note. Risultati molto importanti permettono, ad esempio, di caratterizzare classi di problemi per cui non solo il problema originario, ma anche ottenere una soluzione approssimata sia un problema intrattabile.

Ricordiamo che un algoritmo euristico si dice *ε-approssimato* se produce una soluzione ε-ottima per ogni istanza. Uno *schema di approssimazione* è un algoritmo che, oltre all'istanza p di P, prende in input anche un parametro $\varepsilon > 0$ e produce una soluzione $\bar{x}$ che è ε-ottima; sostanzialmente uno schema di approssimazione è famiglia infinita di algoritmi ε-approssimati, uno per ogni valore di ε. Tipicamente, la complessità di un tale algoritmo sarà anche una funzione dell'approssimazione ε voluta (crescente con $1/\varepsilon$): uno schema di approssimazione è detto:

- *polinomiale*, se la sua complessità è polinomiale nella dimensione dell'istanza p;
- *pienamente polinomiale*, se la sua complessità è polinomiale nella dimensione di p ed in $1/\varepsilon$.

Accenniamo brevemente qui ad un noto risultato che riguarda la difficoltà di costruire schemi di approssimazione pienamente polinomiali per un problema di ottimizzazione.

Esistono problemi $\mathcal{NP}$-completi che sono risolvibili polinomialmente nella lunghezza dell'input se codificati in unario, mentre non lo sono con codifiche più compatte: un esempio è il problema dello zaino (1.2.2.1). Tali problemi sono in un certo senso più facili di quelli che richiedono algoritmi esponenziali indipendentemente dalla codifica dell'input. Viene detto *pseudopolinomiale* un algoritmo che risolva uno di tali problemi in tempo polinomiale nella lunghezza dell'input codificato in unario; un problema è quindi detto *pseudopolinomiale* se può essere risolto per mezzo di un algoritmo pseudopolinomiale. Viene chiamato $\mathcal{NP}$-completo *in senso forte* un problema $\mathcal{NP}$-completo per cui non esistono algoritmi pseudopolinomiali; analogamente, viene chiamato $\mathcal{NP}$-arduo *in senso forte* un problema di ottimizzazione che abbia come versione decisionale un problema $\mathcal{NP}$-completo in senso forte. I problemi $\mathcal{NP}$-completi in senso forte sono tipicamente quelli che "non contengono numeri": ad esempio, SAT è un problema $\mathcal{NP}$-completo in senso forte. Un diverso esempio è il problema del commesso viaggiatore (TSP) (si veda il §1.2.2.3), che resta $\mathcal{NP}$-arduo anche se i costi degli archi sono limitati a due soli possibili valori, ad esempio "0" e "1". Infatti, dato un algoritmo per TSP con costi 0/1 è possibile risolvere (in modo ovvio) il problema del ciclo Hamiltoniano su un grafo, che è notoriamente $\mathcal{NP}$-completo.
È possibile dimostrare che l'esistenza di algoritmi pseudopolinomiali per un problema è equivalente all'esistenza di *schemi di approssimazione pienamente polinomiali* per il problema; in altri termini, nessun problema $\mathcal{NP}$-arduo in senso forte (la grande maggioranza) ammette schemi di approssimazione pienamente polinomiali (a meno che $\mathcal{P} = \mathcal{NP}$). Di conseguenza, per la maggior parte dei problemi $\mathcal{NP}$-ardui è difficile non solo risolvere il problema originario, ma anche una sua approssimazione arbitraria.

Risultati di questo tipo sono stati dimostrati anche per approssimazioni con errore relativo fissato. Ad esempio, è stata definita un'operazione di riduzione simile a $\propto$ che preserva l'approssimabilità, ossia tale che se esiste un algoritmo ε-approssimato per un un certo problema A ed un altro problema B si riduce ad A, allora esiste un algoritmo ε-approssimato (possibilmente per un diverso ε) anche per B. Sfruttando alcuni risultati che mostrano come per certi problemi non possano esistere algoritmi ε-approssimati con ε più piccolo di una specifica soglia (a meno che $\mathcal{P} = \mathcal{NP}$), si dimostra che per una grande classe di problemi di ottimizzazione non possono esistere algoritmi di approssimazione polinomiali con precisione arbitrariamente piccola.

Appendice B

Grafi e Reti

In questa appendice richiamiamo i principali concetti relativi a grafi e reti; descriviamo inoltre alcune classi di strutture dati che possono essere utilizzate per implementare efficientemente algoritmi su grafi e reti.

B.1 I grafi: notazione e nomenclatura

Un grafo, $G = (N, A)$, è una coppia di insiemi: N è un insieme finito e non vuoto di elementi, mentre A è un insieme finito di coppie di elementi distinti di N.

B.1.1 Grafi, nodi, archi

N è detto *insieme dei nodi*, e usualmente viene indicato mediante i primi $n = |N|$ numeri naturali: $N = \{\, 1\,,\, 2\,,\, \ldots\,,\, n \,\}$. L'insieme A, detto *insieme degli archi*, è in generale indicato con $A = \{\, a_1\,,\, a_2\,,\, \ldots\,,\, a_m \,\}$ $(m = |A|)$. Nel seguito un arco verrà indifferentemente denotato o da un nome che lo individua, ad esempio a_k, oppure da una coppia di nodi. Si distingue il caso in cui la coppia sia ordinata, e quindi si indica con (i, j), oppure non sia ordinata: in questo caso si indica con $\{i, j\}$. Nel primo caso l'arco è detto *orientato*, altrimenti è detto *non orientato*, oppure *lato*. I nodi i e j sono detti *estremi* dell'arco (i, j) (lato $\{i, j\}$), che è *incidente* in essi; in questo caso si dirà che i nodi i e j sono *adiacenti*. Se l'arco è orientato, allora i è la *coda* e j è la *testa* di (i, j); l'arco è *uscente* da i e *entrante* in j. Un grafo i cui archi sono tutti non orientati è detto *non orientato* o *simmetrico*, ed è detto *orientato* se tutti i suoi archi sono orientati (si noti che esistono grafi *misti*). Per rimarcare la differenza tra i due casi, a volte i nodi di un grafo simmetrico vengono chiamati *vertici*, ed il grafo si indica con $G = (V, E)$, essendo E l'insieme dei *lati*.

In un certo senso i grafi orientati sono una generalizzazione dei grafi non orientati: infatti un arco non orientato può essere rappresentato per mezzo di una coppia di archi orientati, come indicato in figura B.1. In figura B.2 sono rappresentati un grafo non orientato (a) ed un grafo orientato (b). Nel seguito pertanto i principali concetti verranno dati solo in termini di grafi orientati in quanto la loro estensione ai grafi non orientati è in genere immediata.

Figura B.1: Equivalenza tra un arco non orientato ed una coppia di archi orientati

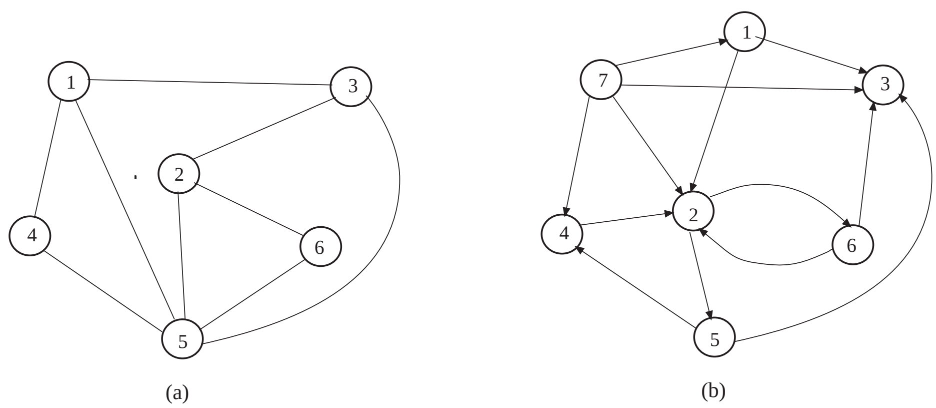


Figura B.2: Esempi di grafi

Esercizio B.1 *Indicare i nodi adiacenti al nodo* 5 *e gli archi incidenti nel nodo* 1 *del grafo in figura B.2(a).*

Esercizio B.2 *Indicare i nodi adiacenti al nodo* 5 *e gli archi incidenti nel nodo* 1 *del grafo in figura B.2(b).*

Una *rete* è un grafo ai cui nodi e/o archi sono associati dei pesi. Tali pesi possono avere significati diversi a seconda del contesto; ad esempio, se il grafo rappresenta una rete idraulica, i pesi associati agli archi possono rappresentare la portata (numero di litri per unità di tempo) e il flusso (effettiva quantità di acqua che fluisce nell'arco), mentre i pesi associati ai nodi possono rappresentare la quantità di acqua immessa nella rete o da essa estratta. I grafi rappresentano la struttura topologica delle reti; nel seguito faremo uso indifferentemente dei termini "grafo" e "rete".

Dato un grafo (orientato o non orientato), per ogni nodo $i \in N$, si indica con $N(i)$ l'*insieme dei nodi adiacenti* ad i e con $S(i)$ l'*insieme degli archi incidenti* in i; $|S(i)|$ è detto il *grado* del nodo i ed è indicato con g_i. Dato un grafo orientato, per ogni nodo $i \in N$, si indica con $FN(i)$ e $BN(i)$ rispettivamente l'*insieme dei nodi successori* e l'*insieme dei nodi predecessori*

$$FN(i) = \{ \, j \in N \; : \; \exists (i,j) \in A \, \} \qquad , \qquad BN(i) = \{ \, j \in N \; : \; \exists (j,i) \in A \, \} \;\; ;$$

mentre con $FS(i)$ e $BS(i)$ si indicano rispettivamente l'*insiemi degli archi uscenti* da i, o *stella uscente* di i, e l'*insieme degli archi entranti* in i, o *stella entrante* di i:

$$FS(i) = \{ \, (x,y) \in A \; : \; x = i \, \} \qquad , \qquad BS(i) = \{ \, (x,y) \in A \; : \; y = i \, \} \;\; .$$

Valgono le relazioni $FN(i) \cup BN(i) = N(i)$ e $FS(i) \cup BS(i) = S(i)$; $|FN(i)| = |FS(i)|$ e $|BN(i)| = |BS(i)|$ sono detti rispettivamente *grado uscente* e *grado entrante* di i.

Esercizio B.3 *Indicare la stella uscente e la stella entrante del nodo* 2 *del grafo in figura B.2(b).*

Dato un grafo $G = (N, A)$, il grafo $G' = (N, A')$, con $A' \subset A$, è detto *grafo parziale* di G; il grafo $G'' = (N'', A'')$, con $N'' \subset N$ e $A'' = \{(i, j) \in A : i, j \in N''\}$, è detto *grafo indotto* da N''. Un grafo $G^* = (N^*, A^*)$, è un *sottografo* di G se A^* è contenuto in A e N^* contiene tutti i nodi estremi degli archi di A^*.

Esercizio B.4 *Disegnare il grafo parziale* $G' = (N, A')$ *del grafo* $G = (N, A)$ *in figura B.2(b), dove* $A' = \{(1, 2), (2, 5), (2, 6), (4, 2), (5, 3), (7, 3)\}$.

Esercizio B.5 *Disegnare il grafo* $G'' = (N'', A'')$ *indotto da* $N'' = \{2, 4, 5, 6\}$ *sul grafo* $G = (N, A)$ *in figura B.2(b).*

Esercizio B.6 *Disegnare il sottografo* $G^* = (N^*, A^*)$ *del grafo* $G = (N, A)$ *in figura B.2(b), dove* $A^* = \{(1, 3), (7, 2), (7, 3)\}$.

B.1.2 Cammini, cicli

Dato un grafo orientato G, un *cammino* tra il nodo i_0 ed il nodo i_q è una sequenza di nodi e di archi

$$C = \{ \; i_0 \; , \; a_1 \; , \; i_1 \; , \; a_2 \; , \; i_2 \; , \; \ldots \; , \; i_{q-1} \; , \; a_q \; , \; i_q \; \}$$

in cui per ogni $h = 1, \ldots, q$, a_h è incidente in i_{h-1} e i_h. Se, per $h = 1, \ldots, q$, è $a_h = (i_{h-1}, i_h)$, allora C è un *cammino orientato* da i_0 a i_q. Il nodo i_0 è detto l'*origine* e i_q è detto la *destinazione* di C. Una sottosequenza di C è detta *sottocammino*. Se $i_0 = i_q$, C è detto un *ciclo* (eventualmente orientato). Un cammino (ciclo) non contenente cicli come sottocammini (propri) è detto un *cammino* (*ciclo*) *semplice* (eventualmente orientato). Nei cammini (cicli) che non sono semplici vi è ripetizione di nodi; essi possono avere anche ripetizioni di archi (si veda la figura B.3). Quando non si ingenerano equivoci, un cammino può essere descritto mediante la sola sequenze dei suoi nodi o dei suoi archi.

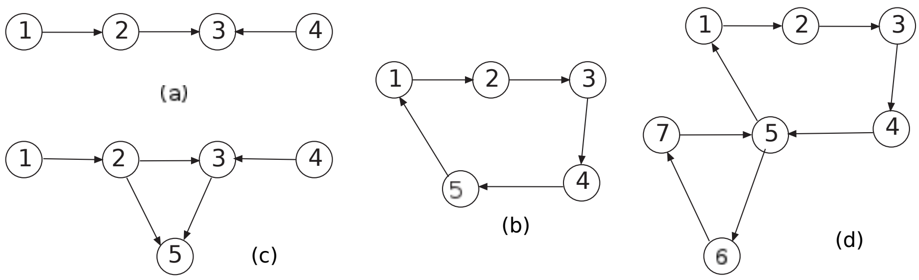


Figura B.3: (a) un cammino semplice; (b) un ciclo orientato semplice; (c) un cammino non semplice con ripetizione di archi; (d) un ciclo orientato non semplice senza ripetizione di archi

Esercizio B.7 *Individuare un cammino tra i nodi* 2 *e* 4 *sul grafo* $G = (N, A)$ *in figura B.2(a); dire se è semplice.*

Esercizio B.8 *Individuare un ciclo orientato semplice formato da tre nodi e tre archi sul grafo $G = (N, A)$ in figura B.2(b).*

Un ciclo semplice formato da n archi è detto *ciclo Hamiltoniano*; esso passa per ogni nodo del grafo una e una sola volta. Un ciclo senza ripetizione di archi formato da m archi è detto *ciclo Euleriano*; esso passa attraverso ciascun arco del grafo una e una sola volta. Si può dimostrare che un grafo possiede un ciclo Euleriano se e solo se il grado di ogni nodo è pari, e che un grafo orientato possiede un ciclo Euleriano orientato se e solo se le cardinalità della stella uscente e della stella entrante di ogni nodo sono uguali.

Esercizio B.9 *Dimostrare l'affermazione precedente.*

Esercizio B.10 *Individuare, se esiste, un ciclo Hamiltoniano sul grafo $G = (N, A)$ in figura B.2(a).*

Due nodi i e j sono *connessi* se esiste un cammino tra di essi; in un grafo orientato si dice che j è *connesso a* i se esiste un cammino orientato da i a j.

Esercizio B.11 *Elencare tutti i nodi che sono connessi al nodo 2 sul grafo $G = (N, A)$ in figura B.2(b).*

B.1.3 Tagli e connettività

Dato un grafo $G = (N, A)$, una partizione di N in due sottoinsiemi non vuoti N' e N'' è detta un *taglio* del grafo e viene indicata con (N', N''). I due insiemi N' ed N'' sono detti le *rive* del taglio, e l'insieme degli archi aventi un estremo in N' e l'altro in N'' è detto *insieme degli archi del taglio* e verrà denotato con $A(N', N'') = \{ (i, j) \in A \, : \, i \in N' \, , \, j \in N''$ oppure $j \in N' \, , \, i \in N'' \}$.

Esercizio B.12 *Sia dato il taglio $(\{1, 3, 5\}, \{2, 4, 6\})$ del grafo $G = (N, A)$ in figura B.2(a), fornire l'insieme degli archi del taglio.*

Se il grafo è orientato, per ogni taglio si possono distinguere due insiemi di archi del taglio, l'insieme $A^+(N', N'')$ detto degli *archi diretti* del taglio e l'insieme $A^-(N', N'')$ detto degli *archi inversi* del taglio:

$$A^+(N', N'') = \{ (i, j) \in A \, : \, i \in N' \, , \, j \in N'' \} \qquad , \qquad A^-(N', N'') = \{ (i, j) \in A \, : \, j \in N' \, , \, i \in N'' \} \ .$$

Ovviamente, $A(N', N'') = A^+(N', N'') \cup A^-(N', N'')$.

Esercizio B.13 *Sia dato il taglio del grafo $G = (N, A)$ in Figura B.2(b) individuato dagli insiemi $N' = \{1, 3, 5, 7\}$ e $N'' = \{2, 4, 6\}$, fornire gli insiemi $A^+(N', N'')$ e $A^-(N', N'')$.*

Utilizzando il concetto di taglio possiamo ridefinire la relazione di connessione fra nodi. Due nodi i e j sono connessi se non esiste un taglio (N', N'') tale che sia $i \in N'$, $j \in N''$ e $A(N', N'') = \emptyset$.

Esercizio B.14 *Dimostrare l'equivalenza delle due definizioni di connessione, basate sui cammini e sui tagli.*

Analogamente, in un grafo orientato j è connesso a i se non esiste un taglio (N', N'') tale che sia $i \in N'$, $j \in N''$ e $A^+(N', N'') = \emptyset$. Un *grafo connesso* è un grafo in cui tutte le coppie di nodi sono connesse, altrimenti è detto *non connesso*. Un *grafo fortemente connesso* è un grafo in cui per ogni coppia di nodi i, j si ha che j è connesso a i, cioè esiste un cammino orientato da i a j. Ad esempio tutti i grafi in Figura B.3 sono connessi, mentre solamente i grafi (b) e (d) sono fortemente connessi. Dato un grafo non connesso $G = (N, A)$, ogni grafo connesso indotto da un sottoinsieme di N, massimale rispetto alla connessione, è detto *componente connessa* di G. Ad esempio il grafo di Figura B.4 contiene due componenti connesse $G' = (N', A')$ e $G'' = (N'', A'')$, con $N' = \{1, 2, 3\}$, $A' = \{(1, 2), (1, 3), (2, 3)\}$, $N'' = \{4, 5, 6, 7\}$, $A'' = \{(4, 7), (6, 5), (6, 7), (7, 5)\}$.

Un grafo è detto *completo* se esiste un arco tra ogni coppia di nodi. Pertanto un grafo non orientato completo possiede $m = n(n-1)/2$ archi, mentre un grafo orientato completo possiede $m = n(n-1)$ archi.

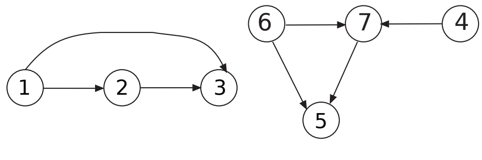


Figura B.4: Due componenti connesse

Esercizio B.15 *Disegnare un grafo completo non orientato con 4 nodi e un grafo completo orientato con tre nodi.*

Un grafo $G = (N, A)$ è detto *bipartito* se esiste un taglio (N', N'') il cui insieme degli archi $A(N', N'')$ coincide con A, cioè se è possibile partizionare N in N' e N'' in modo tale che tutti gli archi abbiano un estremo in N' e l'altro in N''. Un grafo (orientato) bipartito è detto completo se esiste un arco (orientato) per ogni coppia di nodi non appartenenti al medesimo sottoinsieme del taglio; indicando con $n' = |N'|$ e $n'' = |N''|$ le cardinalità dei due sottoinsiemi, il numero di archi è $m = n'n''$ se il grafo è non orientato, mentre è $m = 2n'n''$ nel caso sia orientato.

Esercizio B.16 *Dimostrare le affermazioni fatte sul numero di archi dei grafi completi e dei grafi bipartiti completi (orientati e non).*

Esercizio B.17 *Disegnare un grafo bipartito completo non orientato con 8 nodi in cui $n' = 3$ e $n'' = 5$.*

B.1.4 Alberi

Un grafo connesso e privo di cicli è detto *albero*. Un albero $T = (N, A)$, con $|N| = n$, è tale che $|A| = n - 1$. Sono equivalenti alla precedente le seguenti definizioni: un albero è un grafo connesso con $n - 1$ archi; un albero è un grafo privo di cicli con $n - 1$ archi.

Esercizio B.18 *Dimostrare l'equivalenza delle definizioni date.*

Un *albero radicato* è un albero in cui sia stato selezionato un nodo, detto *radice dell'albero*; in un tale albero i nodi possono essere ordinati per "livelli" in modo ricorsivo: la radice è posta al livello 0 e i nodi adiacenti ad essa sono posti al livello 1; al livello $k + 1$ appartengono i nodi che non appartengono al livello $k - 1$ e che sono adiacenti ai nodi del livello k. Nel grafo di figura B.5, se si assume il nodo 1 come radice, al livello 0 appartiene il nodo 1, al livello 1 si trovano i nodi 2 e 3, a livello 2 si trovano i nodi 4, 5, 6 e 7, mentre a livello 3 si trovano i nodi 8 e 9.

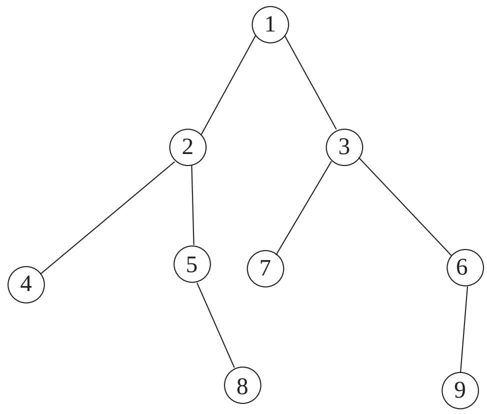


Figura B.5: Un albero

Esercizio B.19 *Disegnare l'albero in figura B.5 radicato nel nodo* 3*; individuare i nodi a livello* 2.

Ogni arco di un albero radicato connette due nodi appartenenti a livelli adiacenti. Per ciascun arco (i, j) con i a livello k e j a livello $k + 1$, i è detto *padre* di j e questi *figlio* di i; la radice non ha padre. Nodi aventi lo stesso padre sono detti *fratelli*. Un nodo senza figli è detto *foglia* dell'albero radicato.
Esiste un solo cammino tra la radice e qualsiasi nodo dell'albero; la lunghezza (in numero di archi) di tale cammino è uguale al livello cui appartiene il nodo destinazione del cammino. Un nodo i che appartiene al cammino dalla radice ad un nodo j è detto un *antenato* di j e questi un *discendente* di i (ogni nodo è antenato e discendente di sé stesso). Un sottoalbero $T(j)$ di un albero radicato è il grafo indotto dall'insieme dei nodi discendenti di j; in altri termini $T(j)$ è formato da j, da tutti gli altri suoi discendenti e da tutti gli archi dell'albero tra questi nodi. Ad esempio, nell'albero di figura B.5, considerato radicato in 1, il padre di 3 è 1; i nodi 4 e 5 sono fratelli; l'insieme delle foglie è $\{4, 7, 8, 9\}$; l'insieme degli antenati di 3 è $\{1, 3\}$ e quello dei discendenti è $\{3, 6, 7, 9\}$. Inoltre, il sottoalbero $T(2)$, disegnato in figura B.6, dell'albero di figura B.5, radicato in 1, contiene i nodi $2, 4, 5$ e 8.

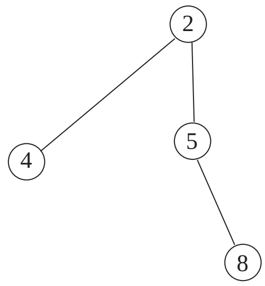


Figura B.6: Un sottoalbero

Esercizio B.20 *Individuare sull'albero disegnato per l'esercizio precedente i figli della radice, gli antenati di* 6 *e i discendenti di* 1*; disegnare il sottoalbero* $T(6)$.

Un albero radicato, i cui archi sono orientati, è detto *orientato* se tutti i suoi archi sono orientati dal padre verso il figlio (o dal figlio verso il padre). Dato un grafo $G = (N, A)$, un suo grafo parziale $T = (N, A_T)$ che sia un albero è detto *albero di copertura* (*spanning tree*) di G; nel grafo di figura B.7 è evidenziato un albero di copertura. Ogni arco $(i, j) \in A$, non appartenente a T, forma con gli archi di T un unico ciclo che indicheremo con $C_T(i, j)$. Inoltre, l'eliminazione di un arco $(i, j) \in A_T$ divide l'albero T in due sottoalberi $T_i = (N_i, A_i)$ e $T_j = (N_j, A_j)$, individuando un taglio (N_i, N_j). Gli archi del taglio sono quelli dell'insieme

$$A(N_i, N_j) = \{(k, l) \in A : k \in N_i, l \in N_j \text{ oppure } l \in N_i, k \in N_j\} \;;$$

cioè, essi sono (i, j) stesso e tutti gli archi non appartenenti a T che, quando aggiunti all'albero, formano un ciclo contenente (i, j). Un grafo le cui componenti connesse sono alberi è detto *foresta*.

Esercizio B.21 *Disegnare una foresta del grafo in figura B.4.*

B.2 Rappresentazione di grafi ed alberi

In questo paragrafo verranno presentate le strutture dei dati fondamentali per la rappresentazione dei grafi e delle reti, e verranno introdotti alcuni algoritmi elementari che serviranno come strumenti di base per la costruzione di algoritmi per problemi di ottimizzazione su reti.

B.2.1 Matrici di incidenza e liste di adiacenza

Dato un grafo orientato $G = (N, A)$, la sua *matrice di incidenza* $E = [e_{ik}]$ è una matrice $n \times m$ (le righe corrispondono ai nodi e le colonne agli archi), così definita:

$$e_{ik} = \begin{cases} -1 & \text{se } i \text{ è la coda dell'arco } a_k \\ 1 & \text{se } i \text{ è la testa dell'arco } a_k \\ 0 & \text{altrimenti.} \end{cases}$$

La matrice di incidenza ha due soli elementi diversi da 0 per ogni colonna: un -1 ed un 1. Un esempio di matrice di incidenza è riportato in figura B.8. La *lista di adiacenza* per stelle uscenti di un grafo orientato è la sequenza $\{FS(1), FS(2), \ldots, FS(n)\}$ degli insiemi degli archi uscenti dai nodi del grafo. Nell'esempio in figura B.8, la lista di adiacenza è $\{\{a_1, a_2\}, \{a_4, a_8\}, \{a_3, a_5, a_6\}, \{a_9, a_{10}\}, \{a_7\}, \emptyset\}$. In modo analogo si definisce la lista di adiacenza per stelle entranti $\{BS(1), BS(2), \ldots, BS(n)\}$.

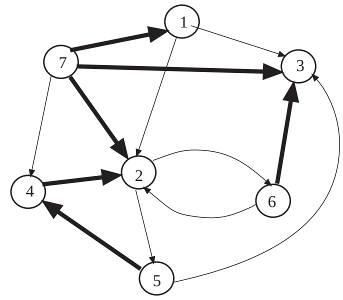
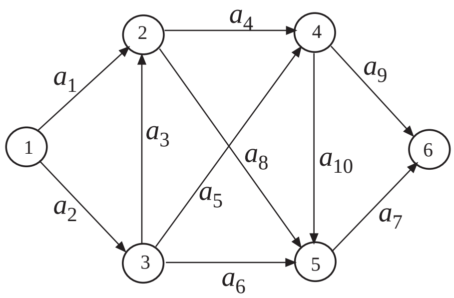


Figura B.7: Un grafo albero di copertura



	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}
1	-1	-1	0	0	0	0	0	0	0	0
2	1	0	1	-1	0	0	0	-1	0	0
3	0	1	-1	0	-1	-1	0	0	0	0
4	0	0	0	1	1	0	0	0	-1	-1
5	0	0	0	0	0	1	-1	1	0	1
6	0	0	0	0	0	0	1	0	1	0

Figura B.8: Un grafo e la sua matrice di incidenza

Le liste di adiacenza consentono una efficiente memorizzazione dei grafi. Ad esempio in figura B.9 è riportata una possibile rappresentazione del grafo di figura B.8 che utilizza un sistema di liste. In particolare:

1. I nodi sono rappresentati per mezzo di una lista in cui ogni record corrisponde ad un nodo e contiene quattro campi: il nome del nodo, il puntatore al nodo successivo nella lista, il puntatore al primo arco della stella uscente, il puntatore al primo arco della stella entrante; un puntatore al primo nodo consente di iniziare la scansione.
2. Gli archi sono rappresentati per mezzo di una lista in cui ogni record corrisponde ad un arco e contiene sei campi: il nome dell'arco, il puntatore all'arco successivo, la coda, la testa, il puntatore all'arco successivo della stella uscente cui esso appartiene, il puntatore all'arco successivo della stella entrante cui esso appartiene; anche qui c'è un puntatore al primo arco.

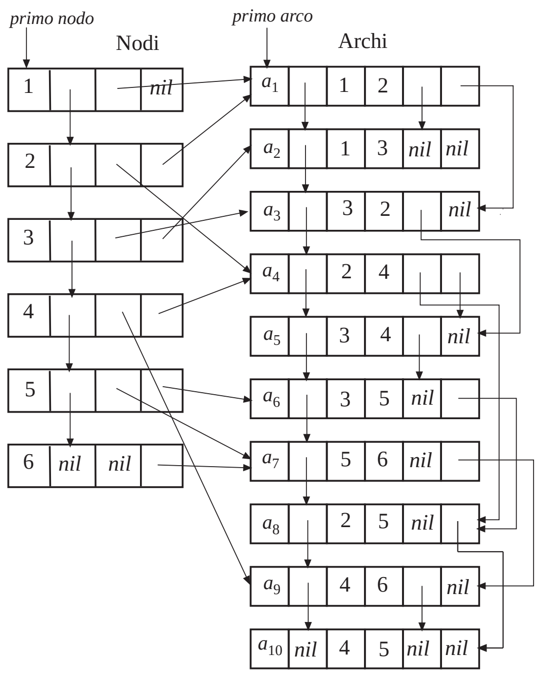


Figura B.9: Liste di adiacenza

Tale struttura dati permette agevolmente sia la scansione dei nodi e degli archi, sia la scansione degli archi di una stessa stella uscente od entrante quando si conosca il nodo relativo. Essa permette anche inserimenti e rimozioni di nodi e/o di archi. La struttura dati può essere ulteriormente ampliata: ad esempio, a ciascun record di nodo e/o di arco possono essere aggiunti nuovi campi contenenti i pesi o le variabili di interesse. Un altro esempio di ampliamento della struttura consiste nell'introduzione di puntatori inversi dei puntatori sopra definiti; la presenza di tali puntatori consente la rimozione di un nodo o di un arco senza dover scorrere la lista alla ricerca del record precedente.

Esercizio B.22 *Scrivere una procedura che, utilizzando la struttura per liste di adiacenza descritta sopra, calcoli il grado entrante ed il grado uscente di ciascun nodo; se ne valuti la complessità.*

Esercizio B.23 *Scrivere una procedura che, utilizzando la struttura per liste di adiacenza descritta sopra, inserisca un nuovo nodo $n + 1$; se ne valuti la complessità.*

Esercizio B.24 *Scrivere una procedura che, utilizzando la struttura per liste di adiacenza descritta sopra, inserisca un nuovo arco a_{m+1}; se ne valuti la complessità.*

Esercizio B.25 *Scrivere una procedura che, utilizzando la struttura per liste di adiacenza descritta sopra, rimuova un nodo i e tutti gli archi incidenti in esso; se ne valuti la complessità.*

Esercizio B.26 *Risolvere il problema dell'esercizio precedente utilizzando una struttura ampliata con i puntatori inversi.*

Spesso, nella risoluzione di problemi su grafi, le rimozioni di nodi ed archi sono temporanee in quanto un particolare sottoproblema da risolvere è relativo ad una data porzione del grafo originario e sottoproblemi successivi sono relativi ad altre porzioni; non si vuole pertanto distruggere la struttura di dati che descrive il grafo originario. In tal caso, la rimozione fisica dei record relativi a nodi ed archi risulta inutilmente costosa. È più conveniente invece affiancare, in ciascun record, ai puntatori 'statici' che definiscono il grafo originario, nuovi puntatori 'dinamici' che definiscono il sottografo corrente, ed operare gli opportuni aggiornamenti su di essi.

Liste di adiacenza mediante vettori di puntatori

La struttura per liste di adiacenza può essere semplificata quando non si prevedono aggiunte o rimozioni di nodi e/o archi. In tal caso le liste a puntatori dei nodi e degli archi possono essere agevolmente realizzate mediante vettori facendo corrispondere l'indice del nodo o dell'arco con l'indice della componente del vettore contenente le informazioni relative al nodo o all'arco. Per realizzare la lista di adiacenza per stelle uscenti è sufficiente disporre di un vettore, $P\_FS[\cdot]$, ad $n+1$ componenti, una per ogni nodo più una ausiliaria, e di un vettore, $H\_Arc[\cdot]$, ad m componenti, una per ogni arco. L'elemento i-esimo del primo vettore ($i = 1, \ldots, n$) contiene il puntatore al primo arco della stella uscente del nodo i, mentre l'elemento $n+1$ punta all'arco fittizio $m+1$. L'elemento k-esimo del secondo vettore ($k = 1, \ldots, m$) contiene il nodo testa dell'arco k; gli archi sono ordinati per stelle uscenti. Per conoscere la stella uscente del nodo i basta effettuare una scansione del vettore $H\_Arc[\cdot]$ tra la posizione $P\_FS[i]$ e la posizione $P\_FS[i+1]-1$, ottenendo le teste degli archi aventi i come coda. La stella uscente è vuota se $P\_FS[i] = P\_FS[i+1]$. L'occupazione di memoria di questa rappresentazione della lista di adiacenza è $m+n+1$. Ad esempio per rappresentare il grafo di figura B.8 i due vettori $P\_FS[\cdot]$ e $H\_Arc[\cdot]$ assumono i seguenti valori:

i	1	2	3	4	5	6	7
$P\_FS[i]$	1	3	5	8	10	11	11

k	1	2	3	4	5	6	7	8	9	10
$H\_Arc[k]$	2	3	4	5	2	4	5	6	5	6

Analogamente si può realizzare la lista di adiacenza per stelle entranti.

Esercizio B.27 *Costruire la lista di adiacenza (per stelle entranti) del grafo in figura B.8.*

B.2.2 Rappresentazione di alberi: la funzione predecessore

Un albero radicato di radice r è rappresentato mediante la funzione predecessore p:

$$p_j = \begin{cases} i & \text{se } i \text{ è padre di } j \\ 0 & \text{se } j \text{ è la radice } (j = r) \end{cases}$$

(alternativamente si può porre $p_r = r$). Se gli archi dell'albero radicato sono orientati, e ci interessa memorizzare l'orientamento dell'arco che connette un nodo con il padre, basta porre $p_j = -i$ se i è padre di j e l'arco tra essi è (j, i). Osserviamo che la funzione predecessore può essere convenientemente inserita nella struttura dati descritta nel paragrafo B.2.1, inserendo ulteriori campi in ciascun record della lista corrispondente all'insieme dei nodi. Infatti, per il record corrispondente al nodo i è sufficiente aggiungere un campo contenente il puntatore al nodo p_i, un campo (booleano) per l'orientamento dell'arco (i, p_i) o (p_i, i) ed eventualmente un campo contenente il puntatore a tale arco nell'insieme degli archi.

B.2.3 Visite di un albero

Un'operazione particolarmente rilevante è la visita di un albero. A seconda dell'ordine con cui i nodi (e gli archi) vengono visitati avremo diversi tipi di visita. Si dice *visita anticipata* di un albero $T = (N_T, A_T)$, di radice r e definito dalla funzione predecessore p, una visita dei nodi secondo la regola "un nodo i viene visitato solo se tutti i nodi appartenenti all'unico cammino in T tra r e i sono stati visitati", cioè un nodo può essere visitato solo dopo che sono stati visitati tutti i suoi antenati. Pertanto la visita inizia dalla radice dell'albero e termina in una sua foglia. Osserviamo che la visita anticipata visita anche gli archi di T. Infatti, quando viene visitato un nodo $i \neq r$, viene anche implicitamente visitato l'arco $(i, -p_i)$ (o (p_i, i)); quindi la visita anticipata induce un ordinamento sui nodi e sugli archi di T. Una visita anticipata è definita per mezzo di una funzione, $va(\cdot)$, che associa ad ogni nodo i il nodo che verrà visitato dopo i attraverso una visita anticipata di T; inoltre, $va(\cdot)$ associa all'ultimo nodo visitato il primo della visita. Dato un albero ci sono diverse funzioni $va(\cdot)$ che realizzano una visita anticipata. In figura B.10 viene fornito un esempio di visita anticipata; essa permette di visitare consecutivamente i nodi di ciascun sottoalbero.

Definiamo *visita posticipata* di T una visita dei nodi secondo la seguente regola: "un nodo i viene visitato solo se tutti i suoi nodi figli sono stati visitati", cioè un nodo può essere visitato solo dopo che sono stati visitati tutti gli altri suoi discendenti. Analogamente alla funzione $va(\cdot)$, possiamo definire la funzione di visita posticipata, $vp(\cdot)$. Una particolare visita posticipata è data dalla funzione inversa di $va(\cdot)$: $va^{-1}(j) = i \Leftrightarrow va(i) = j$. Con riferimento all'esempio in figura B.10 si ha:

i	1	2	3	4	5	6	7	8	9
$va(i)$	2	4	7	5	8	9	6	3	1
$va^{-1}(i)$	9	1	8	2	4	7	3	5	6

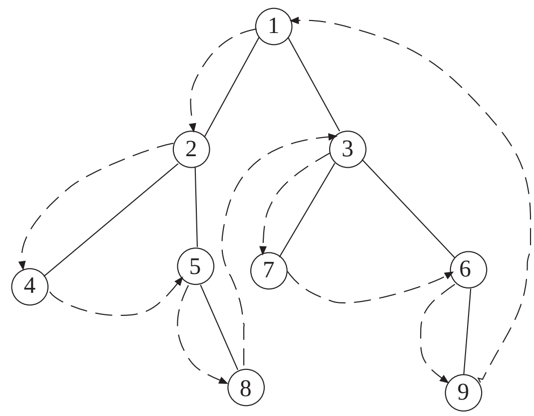


Figura B.10: La funzione $va(\cdot)$ della visita anticipata

B.2.4 Livello dei nodi di un albero

La funzione *livello*, $lev(\cdot)$, dei nodi di un albero associa ad ogni nodo i il suo livello, cioè il numero di archi dell'unico cammino nell'albero tra la radice r e i; $lev(\cdot)$ può essere facilmente calcolata, date le funzioni predecessore p e visita anticipata $va(\cdot)$, mediante la seguente procedura Livello:

```
procedure Livello( r , p , va , lev ) {
  lev[r] = 0; u = va[r];
  while( u ≠ r ) do { lev[u] = lev[p[u]] + 1; u = va[u]; }
  }
```

Procedura 2.1: Calcolo della funzione *livello*

Esercizio B.28 *Determinare $lev(\cdot)$ per l'albero in figura B.10.*

B.3 Visita di un grafo

Gli *algoritmi di visita* sono strumenti che consentono di individuare degli insiemi di nodi o delle porzioni di grafo che soddisfano particolari proprietà. Nel seguito descriveremo prima la versione base della procedura di visita, che risolve il problema di determinare, dato un grafo orientato $G = (N, A)$, l'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato nodo r. Mostreremo poi come tale versione di base possa essere utilizzata o adattata per risolvere problemi diversi. La procedura *Visita* riceve in input il grafo orientato $G = (N, A)$ ed un nodo origine o radice r, e determina i nodi raggiungibili da r per mezzo di cammini orientati. Tali cammini individuano un albero orientato $T_r = (N_r, A_r)$ che viene fornito in output per mezzo di un vettore di predecessori $p[\cdot]$ (o funzione predecessore, si veda B.2.2). Al termine della procedura, ogni nodo i tale che $p[i] = 0$ non è stato raggiunto nella visita. Per il suo funzionamento la procedura si avvale di un insieme, Q, che contiene i *nodi candidati*, cioè quei nodi che sono già stati raggiunti nell'esplorazione ma ancora non sono stati utilizzati per proseguirla. Ad ogni passo la procedura seleziona uno dei nodi in Q e prosegue la visita del grafo a partire da tale nodo; la correttezza dell'algoritmo non dipende dal modo in cui il nodo è selezionato, ossia dall'implementazione della funzione *Select*.

```
procedure Visita ( G , r , p ) {
  Q = { r }; foreach( i ∈ N ) do p[i] = 0;
  do { i = Select(Q); Q = Q \ { i };
        foreach( (i, j) ∈ FS(i) ) do
          if( p[j] == 0 ) then { p[j] = i; Q = Q ∪ { j }; }
        } while( Q ≠ ∅ );
  }
```

Procedura 2.2: Visita di un grafo

Ogni nodo i del grafo viene inserito in Q solamente la prima volta che viene raggiunto, quindi non si avranno più di n inserzioni e rimozioni di nodi da Q, e ogni arco (i, j) verrà esaminato al più una volta, se e quando i viene estratto da Q. Pertanto il numero globale di ripetizioni delle operazioni effettuate nel ciclo "**do** ... **while**" è limitato superiormente da m. Supponendo che le operazioni di gestione di Q abbiano complessità $O(1)$, si ha che la complessità di *Visita* è $O(m)$.

Esercizio B.29 *Realizzare la procedura* Visita *per grafi memorizzati mediante liste di adiacenza.*

B.3.1 Implementazioni della procedura di visita

La correttezza della procedura di visita descritta nel paragrafo precedente è indipendente da:

- l'ordine con cui vengono esaminati gli archi della FS del nodo i estratto da Q, e
- l'ordine con cui vengono estratti i nodi da Q, ossia in che modo l'insieme viene implementato.

Questo significa che, indipendentemente dall'implementazione di queste due operazioni, si ottengono comunque tutti i nodi raggiungibili da r per mezzo di cammini orientati. Implementazioni diverse possono però fornire, al termine della procedura, insiemi di cammini diversi. Non ci soffermeremo sull'effetto dell'ordinamento dei nodi nelle FS, che di solito, in pratica, dipende dai dettagli dell'implementazione delle strutture dati con cui è rappresentato il grafo, e spesso, in ultima analisi, dall'ordine con cui sono descritti gli archi nell'input della procedura. Per semplicità, nel seguito assumeremo che le FS siano ordinate in senso crescente degli indici dei nodi testa.
Per quanto riguarda l'implementazione di Q, scelte diverse possono avere un impatto rilevante sull'insieme dei cammini individuati. In particolare, le implementazioni di Q come fila (*queue*) e pila (*stack*) corrispondono rispettivamente alle strategie di esplorazione del grafo note come *visita a ventaglio* (bfs, da *breadth-first search*) e *visita a scandaglio* (dfs, da *depth-first search*). Si noti che tutti i nodi inseriti in Q in quanto raggiungibili da uno stesso nodo i saranno "figli" di i nell'insieme di cammini determinato. In una visita *bfs*, tutti i figli di uno stesso nodo i vengono inseriti consecutivamente in Q, e quindi estratti consecutivamente da Q: di conseguenza, i discendenti di tali nodi possono essere estratti solamente dopo che l'ultimo di questi nodi è stato estratto, ossia i "fratelli" vengono visitati prima dei figli. In una visita *dfs*, i "figli" del nodo estratto i vengono inseriti in cima alla pila, e quindi saranno estratti (visitati) prima dei "fratelli" di i che sono ancora in Q al momento in cui i viene estratto. Queste due strategie tendono a costruire insiemi di cammini con proprietà abbastanza diverse: in particolare, la visita a ventaglio (Q implementato come fila) tende a costruire cammini "corti", mentre la visita a scandaglio (Q implementato come pila) tende a costruire cammini "lunghi".

Esempio B.1: Diversi tipi di visita di un grafo
Applichiamo la procedura *Visita* al grafo in figura B.11(a) partendo dal nodo $r = 1$. Se Q è implementato mediante

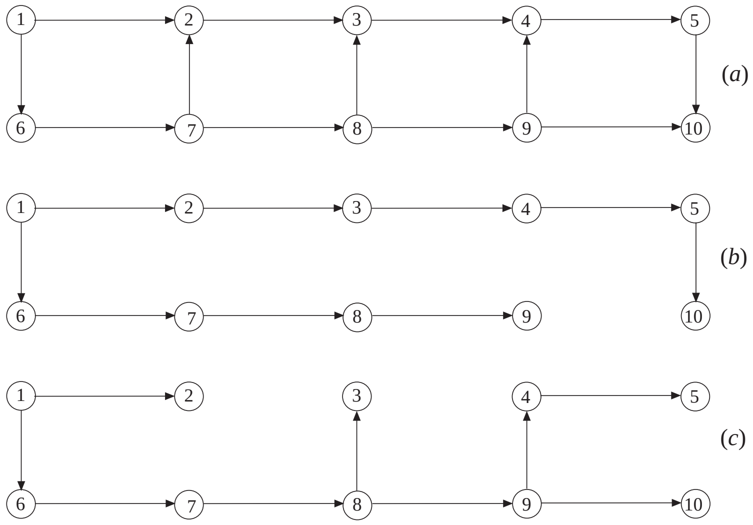


Figura B.11: Applicazioni della procedura *Visita*

una fila (*queue*), il risultato, in termini di cammini determinati, è mostrato in figura B.11(b), L'ordine di inserzione in (e rimozione da) Q è: $1, 2, 6, 3, 7, 4, 8, 5, 9, 10$. La rimozione di 1 da Q causa l'inserimento di 2 e 6 mediante gli archi $(1, 2)$ e $(1, 6)$; la rimozione di 2 causa l'inserimento di 3 mediante l'arco $(2, 3)$; la rimozione di 6 causa l'inserimento di 7 mediante l'arco $(6, 7)$, ecc. La funzione predecessore è definita dal vettore $p = [0, 1, 2, 3, 4, 1, 6, 7, 8, 5]$. In questo caso il risultato è che tutti i nodi sono raggiungibili da r. Diverso è il risultato, mostrato in figura B.11(c), se Q è implementato mediante una pila (*stack*). In questo caso 2 e 6, "figli" di 1, vengono esaminati in ordine inverso (rispetto al caso precedente) in quanto 6 è inserito in Q dopo 2, e quindi ne è estratto prima. Inoltre, quando 6 viene rimosso da Q segue l'inserzione di 7 in cima alla pila, e quindi l'esplorazione prosegue da 7, "figlio" di 6, piuttosto che da suo "fratello" 2, e così via. Si noti come i cammini prodotti in questo caso possano essere diversi e più lunghi di quelli prodotti nel caso precedente; in particolare, nel caso (c) i cammini da 1 a 4 e da 1 a 5 sono formati rispettivamente da 5 e 6 archi, mentre nel caso (b) erano formati rispettivamente da 3 e 4 archi.

In effetti, è possibile dimostrare il seguente interessante risultato:

Teorema B.1 *La procedura di visita in cui Q è implementato come fila determina, per ogni nodo i raggiungibile da r, un cammino orientato da r a i di lunghezza minima in termini di numero di archi.*

Dimostrazione Per ogni nodo i raggiungibile da r, denotiamo con $d(i)$ la "distanza" di i da r, ossia la lunghezza (in termini di numero di archi) del più corto cammino tra i ed r ($d(r) = 0$). Dimostriamo per induzione che ogni nodo i con $d(i) = k > 0$ è inserito in Q—e quindi, siccome Q è una fila, estratto da Q—dopo tutti i nodi h con $d(h) = k - 1$ e prima di qualsiasi nodo j con $d(j) > k$ (se ne esistono); inoltre, al momento di essere inseriti in Q il loro predecessore $p[i] = h$ ha $d(h) = k - 1$ (ossia il cammino determinato dalla visita ha lunghezza pari a k). Questo è certamente vero per

$k = 1$: un nodo i ha distanza 1 da r se e solo se $(r, i) \in FS(r)$, e tutti questi nodi sono esaminati e posti in Q alla prima iterazione della visita, quando viene esaminato r. Per il passo induttivo, supponiamo che la proprietà sia vera per k e dimostriamo che è vera per $k+1$. Dall'ipotesi induttiva segue immediatamente che quando il *primo* nodo i con $d(i) = k$ è estratto da Q, tutti i nodi h che erano stati estratti da Q in precedenza avevano $d(h) < k$: quindi, in quel momento tutti i nodi j con $d(j) > k$ non sono ancora stati inseriti in Q, e quindi hanno $p[j] = 0$. Un nodo j ha $d(j) = k + 1$ se e solo se esiste almeno un nodo i con $d(i) = k$ tale che $(i, j) \in A$; siccome Q è una fila, dall'ipotesi induttiva segue che tutti i nodi con $d(j) = k + 1$ sono inseriti in coda alla fila nelle iterazioni della visita in cui sono estratti da Q ed esaminati tutti i nodi i con $d(i) = k$, e che il loro predecessore è uno di tali nodi. Da questo segue che la proprietà è vera anche per $k + 1$, e quindi il teorema. ◇

Quindi, la procedura di visita è, in alcuni casi, in grado di calcolare insiemi di cammini che utilizzano il minimo numero di archi: nel paragrafo 3.2 vedremo come affrontare problemi di cammini minimi di tipo più generale.

B.3.2 Usi della procedura di visita

La procedura sopra descritta può essere modificata per risolvere anche altri problemi, tra i quali citiamo i seguenti:

- determinare l'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato *insieme* $R \subset N$ di nodi;
- determinare l'insieme dei nodi a partire dai quali un dato nodo r è raggiungibile per mezzo di un cammino orientato;
- determinare l'insieme dei nodi raggiungibili per mezzo di un cammino *non orientato* a partire da un dato nodo r, o, equivalentemente, determinare l'insieme dei nodi raggiungibili a partire da un dato nodo r su un grafo *non orientato*;
- individuare se un grafo è aciclico e, se non lo è, determinare un ciclo del grafo;
- determinare le componenti connesse di un grafo;
- determinare se un grafo è bipartito.

Tali problemi sono risolubili con piccole modifiche alla procedura *Visita*, e/o applicando la procedura ad un opportuno grafo ottenuto a partire da quello originario.
Ad esempio, supponiamo di voler determinare l'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato insieme R di nodi; è facile verificare che questo problema può essere risolto mediante un'applicazione della procedura *Visita* al grafo $G' = (N', A')$ in cui $N' = N \cup \{s\}$, dove s è un nodo fittizio che funge da "super-radice", e $A' = A \cup \{\, (s, r) \,:\, r \in R \,\}$. Per il problema di determinare l'insieme dei nodi a partire dai quali r è raggiungibile per mezzo di un cammino orientato, invece, è sufficiente applicare la procedura *Visita*, con la stessa radice, al grafo $G' = (N, A')$ che ha gli stessi nodi del grafo originario G ma i cui archi sono "invertiti" rispetto a quelli di G, ossia tale che $A' = \{\, (j, i) \,:\, (i, j) \in A \,\}$. Analogamente, se si vuole determinare l'insieme dei nodi raggiungibili da r per mezzo di un cammino non orientato, è sufficiente applicare la procedura al grafo $G' = (N, A')$ in cui $A' = A \cup \{\, (j, i) \,:\, (i, j) \in A \,\}$. Si noti che, negli esempi precedenti, è possibile evitare di costruire una rappresentazione del grafo G' modificando opportunamente la procedura *Visita* in modo che possa lavorare direttamente sulle strutture dati che descrivono il grafo originario G. Ad esempio, nel caso della determinazione dell'insieme dei nodi raggiungibili per mezzo di un cammino orientato a partire da un dato insieme R di nodi, è solamente necessario sostituire le istruzioni

$$p[r] = 0;\ Q = \{r\}; \qquad \text{con} \qquad \textbf{for each}(\ r \in R\ )\ \textbf{do}\ \{\ p[r] = 0;\ Q = R;\ \}\ .$$

In altri termini, basta inizializzare tutti i nodi di R come nodi radice e porli tutti in Q all'inizio dell'algoritmo. Per il problema di determinare l'insieme dei nodi a partire dai quali r è raggiungibile per mezzo di un cammino orientato, invece, è sufficiente modificare l'istruzione

$$\textbf{for each}(\ (i, j) \in FS(i)\ )\ \textbf{do} \ldots \qquad \text{in} \qquad \textbf{for each}(\ (j, i) \in BS(i)\ )\ \textbf{do} \ldots$$

Analogamente, per il problema della raggiungibilità attraverso cammini non orientati è sufficiente esaminare sia gli archi $(i, j) \in FS(i)$ che gli archi $(j, i) \in BS(i)$ corrispondenti al nodo i estratto da Q.

Esercizio B.30 *Ovviamente, le operazioni precedenti si possono combinare: ad esempio, si discuta come modificare la procedura di visita per determinare l'insieme di nodi a partire dai quali almeno uno dei nodi in $R \subset N$ è raggiungibile mediante un cammino orientato.*

Altri problemi possono essere risolti con l'uso ripetuto della procedura di visita o con modifiche minori alla stessa. Alcuni di questi problemi sono descritti nei seguenti esercizi.

Esercizio B.31 *Si proponga un algoritmo di complessità $O(m)$, basato sulla procedura di visita, che determini il numero di componenti connesse di un grafo non orientato, fornendone lo pseudo-codice; si noti che in un grafo non orientato $FS(i)$ e $BS(i)$ non sono definite, è definita solamente la stella $S(i)$ degli archi incidenti nel nodo i (suggerimento: la visita a partire da un qualunque nodo i determina la compenente connessa di cui i fa parte; al termine della visita i nodi delle altre componenti connesse hanno predecessore nullo).*

Esercizio B.32 *Si modifichi l'algoritmo dell'esercizio precedente in modo tale che, con la stessa complessità, produca un vettore* $cc[\cdot]$ *tale che* $cc[i] = k$ *se e solo se il nodo* i *appartiene alla* k*-esima componente connessa.*

Esercizio B.33 *Si costruisca una versione modificata della procedura di visita (fornendo lo in pseudo-codice) che risolva il problema di determinare se un dato grafo non orientato e connesso sia aciclico, ossia se contenga oppure no cicli. Nel caso che il grafo non sia aciclico, si richiede che la procedura produca in output (come "certificato") un ciclo del grafo. Si discuta la complessità di tale procedura.*

Esercizio B.34 *Si adatti l'algoritmo dell'esercizio precedente al caso di un grafo non connesso. Si discuta la complessità di tali procedure.*

Esercizio B.35 *Fornire un algoritmo di visita, di complessità* $O(m)$*, per verificare se un dato grafo non orientato, eventualmente non connesso, sia bipartito.*

Esercizio B.36 *Fornire un algoritmo di visita, di complessità* $O(m)$*, per verificare se un grafo, orientato e connesso, sia fortemente connesso (suggerimento: è sufficiente verificare che un arbitrario nodo* r *del grafo è connesso a tutti gli altri nodi e questi sono connessi ad* r *mediante cammini orientati).*

B.4 Albero di copertura di costo minimo

Riprendiamo il problema dell'*albero di copertura di costo minimo* (MST), già presentato nel paragrafo 1.2.2.2. Dato un grafo non orientato $G = (V, E)$ con costi c_{ij} associati ai lati, consideriamo il problema della determinazione di un albero di copertura $T = (V, E_T)$ tale che sia minimo il costo di T, definito come $c(T) = \sum_{\{i,j\} \in E_T} c_{ij}$. Senza alcuna perdita di generalità possiamo assumere che G sia connesso e che i costi siano positivi: la connessione può infatti essere ottenuta con l'aggiunta di archi a costo opportunamente grande ($M = c_{max} + 1$), mentre la positività dei costi può essere ottenuta sommando al costo di ogni arco una opportuna costante C (di conseguenza, il costo di ogni soluzione viene incrementato del valore $(n-1)C$).
Il problema (MST) può essere risolto per mezzo di un algoritmo di tipo *greedy*. Presenteremo qui uno schema algoritmico molto generale, *Greedy-MST*, per la sua risoluzione. Tale algoritmo costruisce l'albero incrementalmente, mantenendo ad ogni passo due insiemi di lati: S, l'insieme dei lati già inseriti nell'albero, e R, l'insieme dei lati scartati, cioè l'insieme dei lati che certamente non verranno inseriti nell'albero. S e R, che all'inizio sono vuoti, vengono aggiornati per mezzo delle seguenti operazioni:

- *Inserzione*: seleziona in G un taglio (V', V'') tale che $S \cap A(V', V'') = \emptyset$, ed aggiungi ad S un lato $\{u, v\}$ per cui $c_{uv} = \min\{\, c_{ij} \,:\, \{i, j\} \in A(V', V'') \setminus R \,\}$.
- *Cancellazione*: seleziona in G un ciclo C tale che $C \cap R = \emptyset$, e aggiungi a R un lato $\{u, v\} \in C \setminus S$ per cui $c_{uv} = \max\{\, c_{ij} \,:\, \{i, j\} \in C \setminus S \,\}$.

```
procedure Greedy-MST( G , c , S )  {
  S = ∅; R = ∅;
  do { applica Inserzione o Cancellazione
       } while( S ∪ R ⊊ E and |S| < n − 1; )
}
```

Procedura 2.3: Algoritmo *greedy* per il problema (MST)

L'algoritmo termina quando nessuna delle due operazioni è più applicabile, cioè quando risulta $S \cup R = E$, oppure quando sono stati inseriti $n-1$ lati in S, cioè quando il grafo parziale definito da S è un albero. Quando l'algoritmo termina, S definisce una soluzione ottima del problema; vale infatti la seguente proprietà:

Lemma B.1 *Se la coppia di insiemi* (S, R) *è tale per cui esiste in* G *un albero di copertura di costo minimo* $T = (V, E_T)$ *con* $S \subseteq E_T$ *e* $R \cap E_T = \emptyset$*, allora l'applicazione di una qualsiasi delle operazioni di* inserzione *o di* cancellazione *produce una nuova coppia* (S, R) *che gode della stessa proprietà.*

Dimostrazione Cominciamo col dimostrare che la tesi vale per l'operazione di *inserzione*. Sia T un albero ottimo tale che E_T contiene S e $R \cap E_T = \emptyset$; siano (V', V'') e $\{u, v\}$, rispettivamente, il taglio ed il lato selezionati dall'operazione di *inserzione*. Se $\{u, v\} \in E_T$ allora la proprietà è soddisfatta. Altrimenti, essendo T connesso, esiste almeno un lato $\{k, l\}$ dell'unico cammino che connette u e v in T che appartenga ad $A(V', V'')$. Sostituendo $\{u, v\}$ a $\{k, l\}$ in E_T, si ottiene un nuovo albero di copertura T'; essendo $c_{uv} \leq c_{kl}$, si ha $c(T') \leq c(T)$. Siccome per ipotesi T è ottimo, anche T' lo è (e quindi, $c_{uv} = c_{kl}$); T' contiene il lato selezionato $\{u, v\}$, e quindi la proprietà è soddisfatta da T' (anche se non da T).
Consideriamo ora l'operazione di *cancellazione*. Sia T un albero ottimo per cui $E_T \cap R = \emptyset$, e siano C e $\{u, v\}$, rispettivamente, il ciclo e l'arco selezionati dall'operazione di *cancellazione*. Se T non contiene $\{u, v\}$, allora la proprietà è soddisfatta. Altrimenti, cancellando $\{u, v\}$ dall'albero si ottengono due sottoalberi $T' = (V', E'_T)$ e $T'' = (V'', E''_T)$; tra gli archi del ciclo C deve necessariamente esistere un arco $\{k, l\} \notin E_T$ che appartenga a $A(V', V'')$, con $c_{kl} \leq c_{uv}$.

Sostituendo $\{u, v\}$ con $\{k, l\}$ in T si ottiene un nuovo albero, anch'esso ottimo, che soddisfa la proprietà (si ha infatti $c_{kl} = c_{uv}$). $\diamond$

Dal lemma precedente si ricava immediatamente per induzione che l'algoritmo *Greedy-MST* termina fornendo un albero di copertura di costo minimo (basta osservare che all'inizio S e R sono vuoti, e quindi godono banalmente della proprietà richiesta). Si noti che la correttezza dell'algoritmo non dipende dall'ordine con cui vengono realizzate le operazioni di *inserzione* e *cancellazione*. In effetti, esistono diverse possibili implementazioni dell'algoritmo *Greedy-MST* che si distinguono per l'ordine con cui vengono effettuate tali operazioni: nel seguito presenteremo due di tali algoritmi.

B.4.1 Algoritmo di Kruskal

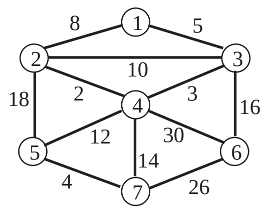
In questo algoritmo, i lati del grafo vengono inizialmente ordinati in ordine di costo non decrescente. Seguendo tale ordinamento, ad ogni iterazione viene selezionato il primo arco $\{u, v\}$ non ancora esaminato: se $\{u, v\}$ non forma alcun ciclo con gli archi in S, allora esso viene inserito in S, cioè si applica l'operazione di *inserzione*, altrimenti l'arco viene inserito in R, cioè si applica l'operazione di *cancellazione*. Le due operazioni sono applicabili legittimamente. Infatti, nel primo caso, la non esistenza di cicli nel grafo parziale (V, S) garantisce l'esistenza di un taglio (V', V'') con $u \in V'$ e $v \in V''$, tale che $\{\, (i, j) \,:\, i \in V' \,,\, j \in V'' \,\} \cap S = \emptyset$; inoltre, poiché i lati non ancora selezionati hanno un costo non minore di c_{uv}, è vero che $\{u, v\}$ è un lato di costo minimo fra quelli del taglio. Nel secondo caso, tutti i lati del ciclo C appartengono ad S tranne $\{u, v\}$: quindi $\{i, j\}$ è il lato di costo massimo tra quelli in $C \setminus S$, essendo l'unico. Si noti comunque che $\{u, v\}$, essendo stato selezionato dopo tutti gli altri lati di C, è anche il lato di costo massimo fra tutti quelli del ciclo. Lo pseudo-codice dell'algoritmo è il seguente, dove la funzione *Sort*(E) restituisce l'insieme E ordinato per costo non decrescente, e la funzione*Component*(E , u , v) risponde *true* se u e v appartengono alla stessa componente connessa del grafo parziale definito da S, cioè se $\{u, v\}$ induce un ciclo su tale grafo, e *false* altrimenti.

procedure *Kruskal*(G , c , S) {
  $S = \emptyset$; $R = \emptyset$; $X = Sort(\ A\ )$;
  do { estrai da X il primo lato $\{u, v\}$;
    if(*Component*(E , u , v))
      then $R = R \cup \{\, \{u, v\} \,\}$; /* cancellazione */
      else $S = S \cup \{\, \{u, v\} \,\}$; /* inserzione */
    } **while**($|S| < n - 1$);
}

Procedura 2.4: Algoritmo di Kruskal per il problema (MST)

L'operazione più critica dell'algoritmo è il controllo di esistenza di un ciclo comprendente il lato $\{u, v\}$, cioè se due nodi u e v appartengono alla stessa componente connessa o meno. È possibile predisporre delle opportune strutture di dati che permettano di effettuare in modo efficiente questo controllo. Con l'ausilio di tali strutture di dati si può ottenere una complessità computazionale pari a $O(m \log n)$, data dal costo dell'ordinamento degli archi.

Esempio B.2: Esecuzione dell'algoritmo di Kruskal



Applichiamo ad l'algoritmo di Kruskal al grafo in figura qui accanto; in figura B.12 sono riportati i passi effettuati, riportando i lati inseriti in S ed il costo $c(S)$. L'ordinamento iniziale fornisce $X = \{$ {2,4}, {3,4}, {5,7}, {1,3}, {1,2}, {2,3}, {4,5}, {4,7}, {3,6}, {2,5}, {6,7}, {4,6} $\}$. Si noti che il quinto lato inserito, $\{4, 5\}$, ha costo $c_{45} = 12$; tale inserimento avviene dopo la cancellazione dei lati $\{1, 2\}$ e $\{2, 3\}$ in quanto $\{1, 2\}$ forma ciclo con $\{2, 4\}$, $\{3, 4\}$ e $\{1, 3\}$, mentre $\{2, 3\}$ forma ciclo con $\{2, 4\}$ e $\{3, 4\}$. Analogamente, prima dell'inserzione di $\{3, 6\}$ vi è stata la cancellazione del lato $\{4, 7\}$ in quanto forma ciclo con $\{5, 7\}$ e $\{4, 5\}$. Con l'inserzione di $\{3, 6\}$ il grafo parziale definito da S diviene connesso ($|S| = 6$), e quindi un albero di copertura di costo minimo; gli ultimi tre archi nell'insieme X non vengono esaminati.

Esercizio B.37 *Applicare* Kruskal *al grafo di figura B.13; fornire ad ogni iterazione la foresta $T = (V, S)$ ed il lato esaminato indicando se viene eliminato o inserito.*

Esercizio B.38 *Se il grafo G non è connesso, non esiste un albero di copertura per G; esiste però una foresta di alberi di copertura, e quindi alberi di copertura di costo minimo, per ciascuna delle componenti connesse. Si discuta come modificare l'algoritmo di Kruskal, senza aumentarne la complessità, in modo tale che determini una foresta di alberi di copertura di costo minimo per tutte le componenti connesse di G.*

B.4.2 Algoritmo di Prim

L'*algoritmo di Prim* effettua solamente l'operazione di *inserzione*, mentre l'operazione di *cancellazione* viene effettuata implicitamente. Per effettuare l'operazione di *inserzione*, viene costruito ed aggiornato ad ogni iterazione un taglio (V', V'') con la caratteristica che (V', S) è un albero di copertura di costo minimo per il grafo indotto da V'. Ad ogni iterazione viene selezionato ed inserito in S un lato a costo minimo fra quelli che hanno un estremo in V' e l'altro in

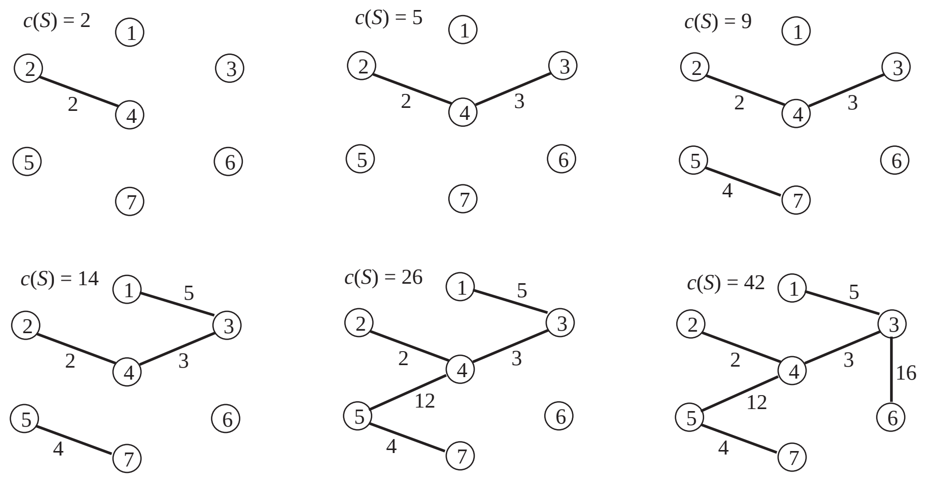


Figura B.12: Passi effettuati dall'algoritmo di Kruskal

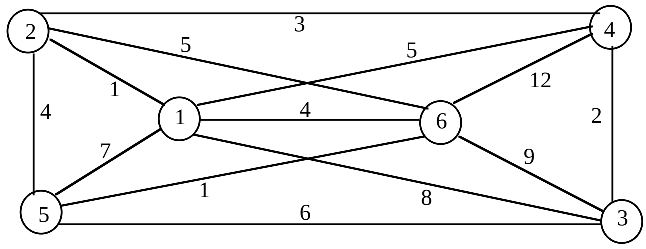


Figura B.13: Un'istanza di (MST)

V'', cioè un lato appartenente all'insieme $A(V', V'')$ degli archi del taglio. L'insieme V' viene inizializzato con un nodo arbitrario r: $V' = \{r\}$ e $V'' = V \setminus \{r\}$, pertanto $S = \emptyset$. Introduciamo un lato fittizio $\{r, i\}$, per ogni $i \in V \setminus \{r\}$, di costo opportunamente grande $c_{ri} = M$. Se G è connesso, dopo l'inserimento di $n - 1$ archi in S, l'insieme V' coincide con N e quindi $T = (V, S)$ è un albero di copertura di costo minimo per G. Altrimenti si otterrà un taglio (V', V'') i cui archi sono tutti fittizi; in tal caso l'algoritmo si arresta connettendo questi nodi ad r mediante gli archi fittizi. Per memorizzare la porzione di albero corrente definita da (V', S), e alla fine l'albero di copertura T, utilizziamo un vettore di predecessori $p[\cdot]$.
Per effettuare l'operazione di inserzione si deve determinare un lato di costo minimo appartenente all'insieme corrente $A(V', V'')$. Al fine di rendere efficiente questa operazione, che in generale avrebbe complessità in tempo $O(m)$, memorizziamo, per ogni nodo $j \in V''$, sia il lato $\{i, j\} \in S(i) \cap A(V', V")$ di costo minimo, utilizzando il vettore di predecessori ($p[j] = i$) che il suo costo, utilizzando un vettore di etichette $d[\cdot]$ ($d[j] = c_{ij}$). In tal modo è possibile determinare, in tempo $O(n)$, il lato di costo minimo in $A(V', V'')$ selezionando un nodo u di etichetta minima tra i nodi in V'', che viene inserito in V'. Lo spostamento di u da V'' a V' equivale all'inserimento di $(p[u], u)$ in S. Si noti che, per aggiornare i valori $p[j]$ e $d[j]$ per ogni nodo $j \in V''$, è sufficiente esaminare ciascun lato $\{u, v\} \in S(u)$ tale che $v \in V''$ e verificare se esso non sia il lato del taglio incidente in v di costo minimo. Infatti, basta scegliere il "lato di costo minore" tra $\{u, v\}$ e $\{p[v], v\}$, cioè basta controllare se $c_{uv} < d_v$. In caso affermativo $\{u, v\}$ risulta migliore di $\{p[v], v\}$ e viene sostituito ad esso ponendo $p[v] = u$ e $d_v = c_{uv}$; altrimenti non si effettua nessun cambiamento. Si noti che, nel primo caso, si ha un'operazione di cancellazione implicita del lato $\{p[v], v\}$, mentre nel secondo viene cancellato il lato $\{u, v\}$. Per ragioni implementative, viene utilizzato un insieme Q dei nodi candidati che contenga tutti e soli i nodi di V'' per i quali esista almeno un lato del taglio incidente in essi; in tal modo la ricerca del nodo di etichetta minima viene effettuata in Q. Si noti che i nodi $j \in V'' \setminus Q$ sono tutti e soli i nodi con etichetta arbitrariamente grande ($d[j] = M$).
Lo pseudo-codice evidenzia la forte somiglianza con l'algoritmo *SPT.S* descritto nel paragrafo 3.2.3. In esso viene modificata la condizione di scelta dell'arco: al posto della condizione di Bellman si inserisce la condizione di "lato di costo minore". Inoltre, per indicare che un nodo i appartiene all'insieme V', e quindi che non devono essere più effettuate inserzioni di lati incidenti in esso, si pone $d[j] = -M$.
È facile verificare che, nel caso in cui Q sia implementata come una lista non ordinata, la complessità dell'algoritmo di Prim è $O(n^2)$, come quella del suo corrispondente per (SPT). Infatti, anche in questo caso non si avranno più di n estrazioni di nodi da Q, e ogni estrazione ha costo $O(n)$ per la scansione di Q per determinare il nodo di etichetta minima. Le operazioni relative ai lati hanno complessità costante e saranno ripetute al più due volte per ciascun lato: quindi, globalmente costano $O(m) = O(n^2)$ poiché $m < n^2$. Se Q è implementato come uno *Heap* la complessità è invece $O(m \log n)$ (si veda il paragrafo 3.2.4).

```
procedure Prim( G , c , r , p )  {
  foreach( i ∈ V ) do { p[i] = r; d[i] = M; }
  d[r] = −M; Q = { r };
  do {  seleziona u in Q tale che d[u] = min{ d[j] : j ∈ Q };
        d[u] = −M; Q = Q \ {u};
        foreach( {u, v} ∈ FS(u) ) do
          if( c[u, v] < d[v] ) then {  d[v] = c[u, v]; p[v] = u;
                                        if( v ∉ Q ) then Q = Q ∪ { v };
                                      }
      } while( Q ≠ ∅ );
}
```

Procedura 2.5: Algoritmo di *Prim*

Esempio B.3: Esecuzione dell'algoritmo di Prim
In figura B.14 sono rappresentate le soluzioni al termine delle iterazioni dell'algoritmo di Prim applicato al grafo dell'Esempio B.2. I lati evidenziati sono quelli inseriti in S, e sono incidenti in nodi i tali che $p[i]$ è fissato e $d[i] = -M = -31$, mentre i lati tratteggiati sono quelli $(p[v], v)$ candidati ad essere inseriti in S, per ogni $v \in Q$ (accanto al nodo è riportata la sua etichetta). La linea tratteggiata indica il taglio (V', V''). I nodi evidenziati sono quelli di etichetta minima che verranno estratti da Q all'iterazione successiva. La sequenza dei valori delle etichette dei nodi rimossi da Q è $0, 5, 3, 2, 12, 4, 16$. Ad ogni iterazione viene riportato il costo $c(S)$ della soluzione corrente; al termine $c(S) = 42$, cioè la somma delle etichette dei nodi al momento della loro estrazione da Q; infatti, esse rappresentano i costi dei lati che formano l'albero.

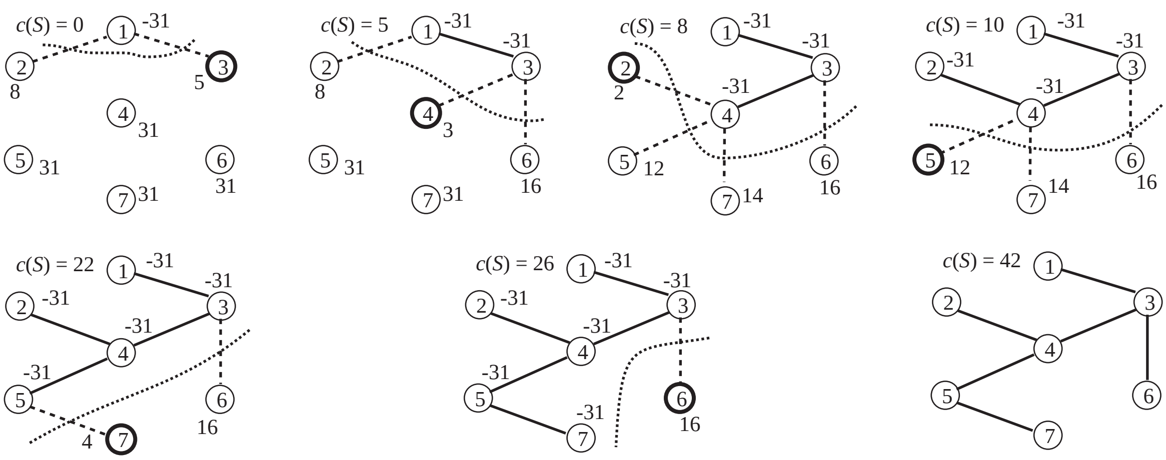


Figura B.14: Passi effettuati dall'algoritmo di Prim

Esercizio B.39 *Applicare* Prim *al grafo di figura B.13; fornire ad ogni iterazione l'albero parziale* $T = (V', E_T)$, *l'insieme* Q *e il nodo* u *selezionato.*

Esercizio B.40 *Si discuta come modificare l'algoritmo di Prim, senza aumentarne la complessità, in modo tale che, anche se applicato ad un grafo* G *non è connesso, determini alberi di copertura di costo minimo per tutte le componenti connesse di* G.

B.4.3 Albero di copertura bottleneck

Un diverso problema di albero ottimo, strettamente correlato con (MST), è quello in cui si richiede di minimizzare non il costo dell'albero di copertura $T = (V, E_T)$ ma il suo *valore bottleneck (collo di bottiglia)* $V(T) = \max\{ c_{ij} : (i, j) \in E_T \}$, ossia il massimo costo degli archi utilizzati. Vale il seguente lemma:

Lemma B.2 *Un albero di copertura di costo minimo è anche un albero bottleneck.*

Dimostrazione Sia $T' = (V, E_{T'})$ un albero di copertura di costo minimo e sia $\{u, v\}$ un suo lato di costo massimo. La rimozione di $\{u, v\}$ da T' genera un taglio (V', V'') del grafo; per l'ottimalità di T' non esistono lati del taglio di costo inferiore a c_{uv} e quindi il valore $V(T)$ di qualsiasi albero bottleneck T è tale che $V(T) \geq c_{uv}$. Da $V(T') = c_{uv}$ segue che T' è un albero bottleneck. $\diamond$

Si noti che, in generale, non è vero il viceversa; infatti, dato un albero di copertura di costo minimo (e quindi anche bottleneck) T', di valore bottleneck $V(T')$, se si sostituisce un qualsiasi lato $\{i, j\}$ con un lato $\{p, q\}$ appartenente al taglio indotto da $\{i, j\}$, purché $c_{ij} < c_{pq} \leq V(T')$, si ottiene un nuovo albero T, che è bottleneck ma non più di costo minimo. Abbiamo pertanto mostrato che i due problemi non sono equivalenti, ma che l'insieme delle soluzioni del problema dell'albero bottleneck contiene l'insieme delle soluzioni del (MST); pertanto per risolvere il primo problema è sufficiente risolvere il secondo.