5

Introduction to Suffix Trees

A suffix tree is a data structure that exposes the internal structure of a string in a deeper way than does the fundamental preprocessing discussed in Section 1.3. Suffix trees can be used to solve the exact matching problem in linear time (achieving the same worst-case bound that the Knuth-Morris-Pratt and the Boyer–Moore algorithms achieve), but their real virtue comes from their use in linear-time solutions to many string problems more complex than exact matching. Moreover (as we will detail in Chapter 9), suffix trees provide a bridge between *exact* matching problems, the focus of Part I, and *inexact* matching problems that are the focus of Part III.

The classic application for suffix trees is the *substring problem*. One is first given a text T of length m. After $O(m)$, or linear, preprocessing time, one must be prepared to take in any unknown string S of length n and in $O(n)$ time either find an occurrence of S in T or determine that S is not contained in T. That is, the allowed preprocessing takes time proportional to the length of the text, but thereafter, the search for S must be done in time proportional to the length of S, *independent* of the length of T. These bounds are achieved with the use of a suffix tree. The suffix tree for the text is built in $O(m)$ time during a preprocessing stage; thereafter, whenever a string of length $O(n)$ is input, the algorithm searches for it in $O(n)$ time using that suffix tree.

The $O(m)$ preprocessing and $O(n)$ search result for the substring problem is very surprising and extremely useful. In typical applications, a long sequence of requested strings will be input after the suffix tree is built, so the linear time bound for each search is important. That bound is *not* achievable by the Knuth-Morris-Pratt or Boyer–Moore methods – those methods would preprocess each requested string on input, and then take $\Theta(m)$ (worst-case) time to search for the string in the text. Because m may be huge compared to n, those algorithms would be impractical on any but trivial-sized texts.

Often the text is a fixed *set* of strings, for example, a collection of STSs or ESTs (see Sections 3.5.1 and 3.5.1), so that the substring problem is to determine whether the input string is a substring of any of the fixed strings. Suffix trees work nicely to efficiently solve this problem as well. Superficially, this case of multiple text strings resembles the *dictionary* problem discussed in the context of the Aho–Corasick algorithm. Thus it is natural to expect that the Aho–Corasick algorithm could be applied. However, the Aho–Corasick method does not solve the substring problem in the desired time bounds, because it will only determine if the new string is a *full* string in the dictionary, not whether it is a substring of a string in the dictionary.

After presenting the algorithms, several applications and extensions will be discussed in Chapter 7. Then a remarkable result, *the constant-time least common ancestor method,* will be presented in Chapter 8. That method greatly amplifies the utility of suffix trees, as will be illustrated by additional applications in Chapter 9. Some of those applications provide a bridge to inexact matching; more applications of suffix trees will be discussed in Part III, where the focus is on inexact matching.

5.1. A short history

The first linear-time algorithm for constructing suffix trees was given by Weiner [473] in 1973, although he called his tree a position tree. A different, more space efficient algorithm to build suffix trees in linear time was given by McCreight [318] a few years later. More recently, Ukkonen [438] developed a conceptually different linear-time algorithm for building suffix trees that has all the advantages of McCreight's algorithm (and when properly viewed can be seen as a variant of McCreight's algorithm) but allows a much simpler explanation.

Although more than twenty years have passed since Weiner's original result (which Knuth is claimed to have called "the algorithm of 1973" [24]), suffix trees have not made it into the mainstream of computer science education, and they have generally received less attention and use than might have been expected. This is probably because the two original papers of the 1970s have a reputation for being extremely difficult to understand. That reputation is well deserved but unfortunate, because the algorithms, although nontrivial, are no more complicated than are many widely taught methods. And, when implemented well, the algorithms are practical and allow efficient solutions to many complex string problems. We know of no other single data structure (other than those essentially equivalent to suffix trees) that allows efficient solutions to such a wide range of complex string problems.

Chapter 6 fully develops the linear-time algorithms of Ukkonen and Weiner and then briefly mentions the high-level organization of McCreight's algorithm and its relationship to Ukkonen's algorithm. Our approach is to introduce each algorithm at a high level, giving simple, *inefficient* implementations. Those implementations are then incrementally improved to achieve linear running times. We believe that the expositions and analyses given here, particularly for Weiner's algorithm, are much simpler and clearer than in the original papers, and we hope that these expositions result in a wider use of suffix trees in practice.

5.2. Basic definitions

When describing how to build a suffix tree for an arbitrary string, we will refer to the generic string S of length m. We do not use P or T (denoting pattern and text) because suffix trees are used in a wide range of applications where the input string sometimes plays the role of a pattern, sometimes a text, sometimes both, and sometimes neither. As usual the alphabet is assumed finite and known. After discussing suffix tree algorithms for a single string S, we will generalize the suffix tree to handle sets of strings.

Definition A suffix tree $\mathcal{T}$ for an m-character string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i..m]$.

For example, the suffix tree for the string *xabxac* is shown in Figure 5.1. The path from the root to the leaf numbered 1 spells out the full string $S = xabxac$, while the path to the leaf numbered 5 spells out the suffix ac, which starts in position 5 of S.

As stated above, the definition of a suffix tree for S does not guarantee that a suffix tree for any string S actually exists. The problem is that if one *suffix* of S matches a *prefix*

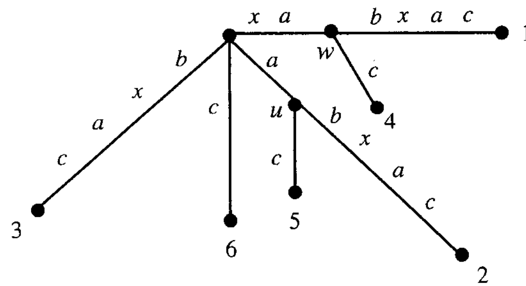

Figure 5.1: Suffix tree for string *xabxac*. The node labels *u* and *w* on the two interior nodes will be used later.

of another suffix of S then no suffix tree obeying the above definition is possible, since the path for the first suffix would not end at a leaf. For example, if the last character of $xabxac$ is removed, creating string $xabxa$, then suffix xa is a prefix of suffix $xabxa$, so the path spelling out xa would not end at a leaf.

To avoid this problem, we assume (as was true in Figure 5.1) that the last character of S appears nowhere else in S. Then, no suffix of the resulting string can be a prefix of any other suffix. To achieve this in practice, we can add a character to the end of S that is not in the alphabet that string S is taken from. In this book we use \$ for the "termination" character. When it is important to emphasize the fact that this termination character has been added, we will write it explicitly as in $S\$$. Much of the time, however, this reminder will not be necessary and, unless explicitly stated otherwise, every string S is assumed to be extended with the termination symbol \$, even if the symbol is not explicitly shown.

A suffix tree is related to the keyword tree (without backpointers) considered in Section 3.4. Given string S, if set $\mathcal{P}$ is defined to be the m suffixes of S, then the suffix tree for S can be obtained from the keyword tree for $\mathcal{P}$ by merging any path of nonbranching nodes into a single edge. The simple algorithm given in Section 3.4 for building keyword trees could be used to construct a suffix tree for S in $O(m^2)$ time, rather than the $O(m)$ bound we will establish.

Definition The *label of a path* from the root that ends at a *node* is the concatenation, in order, of the substrings labeling the edges of that path. The *path-label of a node* is the label of the path from the root of $\mathcal{T}$ to that node.

Definition For any node v in a suffix tree, the *string-depth* of v is the number of characters in v's label.

Definition A path that ends in the middle of an edge (u, v) splits the label on (u, v) at a designated point. Define the label of such a path as the label of u concatenated with the characters on edge (u, v) down to the designated split point.

For example, in Figure 5.1 string xa labels the internal node w (so node w has path-label xa), string a labels node u, and string $xabx$ labels a path that ends inside edge $(w, 1)$, that is, inside the leaf edge touching leaf 1.

5.3. A motivating example

Before diving into the details of the methods to construct suffix trees, let's look at how a suffix tree for a string is used to solve the exact match problem: Given a pattern P of

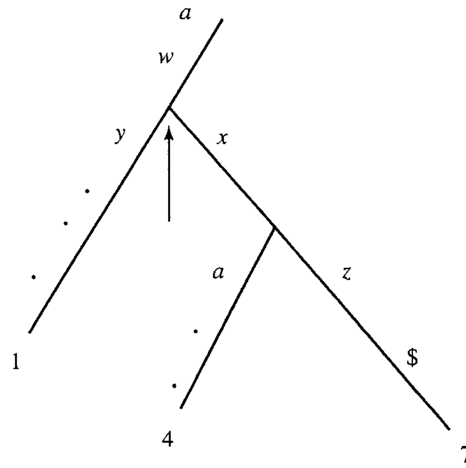

Figure 5.2: Three occurrences of *aw* in *awyawxawxz*. Their starting positions number the leaves in the subtree of the node with path-label *aw*.

length n and a text T of length m, find all occurrences of P in T in $O(n + m)$ time. We have already seen several solutions to this problem. Suffix trees provide another approach:

> Build a suffix tree $\mathcal{T}$ for text T in $O(m)$ time. Then, match the characters of P along the unique path in $\mathcal{T}$ until either P is exhausted or no more matches are possible. In the latter case, P does not appear anywhere in T. In the former case, every leaf in the subtree below the point of the last match is numbered with a starting location of P in T, and every starting location of P in T numbers such a leaf.

The key to understanding the former case (when all of P matches a path in T) is to note that P occurs in T starting at position j if and only if P occurs as a prefix of $T[j..m]$. But that happens if and only if string P labels an initial part of the path from the root to leaf j. It is the initial path that will be followed by the matching algorithm.

The matching path is unique because no two edges out of a common node can have edge-labels beginning with the same character. And, because we have assumed a finite alphabet, the work at each node takes constant time and so the time to match P to a path is proportional to the length of P.

For example, Figure 5.2 shows a fragment of the suffix tree for string $T = awyawxawxz$. Pattern $P = aw$ appears three times in T starting at locations 1, 4, and 7. Pattern P matches a path down to the point shown by an arrow, and as required, the leaves below that point are numbered 1, 4, and 7.

If P fully matches some path in the tree, the algorithm can find all the starting positions of P in T by traversing the subtree below the end of the matching path, collecting position numbers written at the leaves. All occurrences of P in T can therefore be found in $O(n+m)$ time. This is the same overall time bound achieved by several algorithms considered in Part I, but the distribution of work is different. Those earlier algorithms spend $O(n)$ time for preprocessing P and then $O(m)$ time for the search. In contrast, the suffix tree approach spends $O(m)$ preprocessing time and then $O(n + k)$ search time, where k is the number of occurrences of P in T.

To collect the k starting positions of P, traverse the subtree at the end of the matching path using any linear-time traversal (depth-first say), and note the leaf numbers encountered. Since every internal node has at least two children, the number of leaves encountered

is proportional to the number of edges traversed, so the time for the traversal is $O(k)$, even though the total string-depth of those $O(k)$ edges may be arbitrarily larger than k.

If only a single occurrence of P is required, and the preprocessing is extended a bit, then the search time can be reduced from $O(n + k)$ to $O(n)$ time. The idea is to write at each node one number (say the smallest) of a leaf in its subtree. This can be achieved in $O(m)$ time in the preprocessing stage by a depth-first traversal of $\mathcal{T}$. The details are straightforward and are left to the reader. Then, in the search stage, the number written on the node at or below the end of the match gives one starting position of P in T.

In Section 7.2.1 we will again consider the relative advantages of methods that preprocess the text versus methods that preprocess the pattern(s). Later, in Section 7.8, we will also show how to use a suffix tree to solve the exact matching problem using $O(n)$ preprocessing and $O(m)$ search time, achieving the same bounds as in the algorithms presented in Part I.

5.4. A naive algorithm to build a suffix tree

To further solidify the definition of a suffix tree and develop the reader's intuition, we present a straightforward algorithm to build a suffix tree for string S. This naive method first enters a single edge for suffix $S[1..m]\$$ (the entire string) into the tree; then it successively enters suffix $S[i..m]\$$ into the growing tree, for i increasing from 2 to m. We let N_i denote the intermediate tree that encodes all the suffixes from 1 to i.

In detail, tree N_1 consists of a single edge between the root of the tree and a leaf labeled 1. The edge is labeled with the string $S\$$. Tree N_{i+1} is constructed from N_i as follows: Starting at the root of N_i find the longest path from the root whose label matches a prefix of $S[i + 1..m]\$$. This path is found by successively comparing and matching characters in suffix $S[i + 1..m]\$$ to characters along a unique path from the root, until no further matches are possible. The matching path is unique because no two edges out of a node can have labels that begin with the same character. At some point, no further matches are possible because no suffix of $S\$$ is a prefix of any other suffix of $S\$$. When that point is reached, the algorithm is either at a node, w say, or it is in the middle of an edge. If it is in the middle of an edge, (u, v) say, then it breaks edge (u, v) into two edges by inserting a new node, called w, just after the last character on the edge that matched a character in $S[i + 1..m]$ and just before the first character on the edge that mismatched. The new edge (u, w) is labeled with the part of the (u, v) label that matched with $S[i + 1..m]$, and the new edge (w, v) is labeled with the remaining part of the (u, v) label. Then (whether a new node w was created or whether one already existed at the point where the match ended), the algorithm creates a new edge $(w, i + 1)$ running from w to a new leaf labeled $i + 1$, and it labels the new edge with the unmatched part of suffix $S[i + 1..m]\$$.

The tree now contains a unique path from the root to leaf $i + 1$, and this path has the label $S[i + 1..m]\$$. Note that all edges out of the new node w have labels that begin with different first characters, and so it follows inductively that no two edges out of a node have labels with the same first character.

Assuming, as usual, a bounded-size alphabet, the above naive method takes $O(m^2)$ time to build a suffix tree for the string S of length m.

6

Linear-Time Construction of Suffix Trees

We will present two methods for constructing suffix trees in detail, Ukkonen's method and Weiner's method. Weiner was the first to show that suffix trees can be built in linear time, and his method is presented both for its historical importance and for some different technical ideas that it contains. However, Ukkonen's method is equally fast and uses far less space (i.e., memory) in practice than Weiner's method. Hence Ukkonen is the method of choice for most problems requiring the construction of a suffix tree. We also believe that Ukkonen's method is easier to understand. Therefore, it will be presented first. A reader who wishes to study only one method is advised to concentrate on it. However, our development of Weiner's method does not depend on understanding Ukkonen's algorithm, and the two algorithms can be read independently (with one small shared section noted in the description of Weiner's method).

6.1. Ukkonen's linear-time suffix tree algorithm

Esko Ukkonen [438] devised a linear-time algorithm for constructing a suffix tree that may be the conceptually easiest linear-time construction algorithm. This algorithm has a space-saving improvement over Weiner's algorithm (which was achieved first in the development of McCreight's algorithm), and it has a certain "on-line" property that may be useful in some situations. We will describe that on-line property but emphasize that the main virtue of Ukkonen's algorithm is the simplicity of its description, proof, and time analysis. The simplicity comes because the algorithm can be developed as a simple but inefficient method, followed by "common-sense" implementation tricks that establish a better worst-case running time. We believe that this less direct exposition is more understandable, as each step is simple to grasp.

6.1.1. Implicit suffix trees

Ukkonen's algorithm constructs a sequence of *implicit* suffix trees, the last of which is converted to a true suffix tree of the string S.

Definition An *implicit suffix tree* for string S is a tree obtained from the suffix tree for $S\$$ by removing every copy of the terminal symbol \$ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children.

An implicit suffix tree for a prefix $S[1..i]$ of S is similarly defined by taking the suffix tree for $S[1..i]\$$ and deleting \$ symbols, edges, and nodes as above.

Definition We denote the implicit suffix tree of the string $S[1..i]$ by $\mathcal{I}_i$, for i from 1 to m.

The implicit suffix tree for any string S will have fewer leaves than the suffix tree for

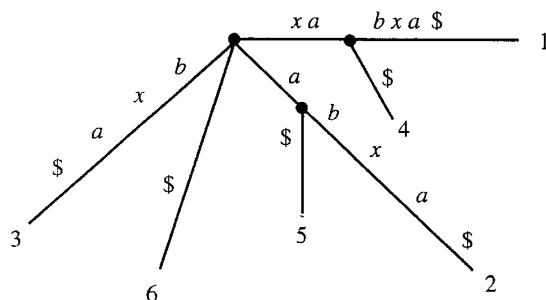

Figure 6.1: Suffix tree for string *xabxa* $.

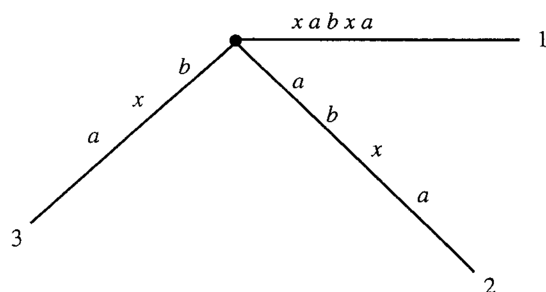

Figure 6.2: Implicit suffix tree for string *xabxa*.

string $S\$$ if and only if at least one of the suffixes of S is a prefix of another suffix. The terminal symbol \$ was added to the end of S precisely to avoid this situation. However, if S ends with a character that appears nowhere else in S, then the implicit suffix tree of S will have a leaf for each suffix and will hence be a true suffix tree.

As an example, consider the suffix tree for string $xabxa\$$ shown in Figure 6.1. Suffix xa is a prefix of suffix $xabxa$, and similarly the string a is a prefix of $abxa$. Therefore, in the suffix tree for $xabxa$ the edges leading to leaves 4 and 5 are labeled only with \$. Removing these edges creates two nodes with only one child each, and these are then removed as well. The resulting implicit suffix tree for $xabxa$ is shown in Figure 6.2. As another example, Figure 5.1 on page 91 shows a tree built for the string $xabxac$. Since character c appears only at the end of the string, the tree in that figure is both a suffix tree and an implicit suffix tree for the string.

Even though an implicit suffix tree may not have a leaf for each suffix, it does encode all the suffixes of S – each suffix is spelled out by the characters on some path from the root of the implicit suffix tree. However, if the path does not end at a leaf, there will be no marker to indicate the path's end. Thus implicit suffix trees, on their own, are somewhat less informative than true suffix trees. We will use them just as a tool in Ukkonen's algorithm to finally obtain the true suffix tree for S.

6.1.2. Ukkonen's algorithm at a high level

Ukkonen's algorithm constructs an implicit suffix tree $\mathcal{I}_i$ for each prefix $S[1..i]$ of S, starting from $\mathcal{I}_1$ and incrementing i by one until $\mathcal{I}_m$ is built. The true suffix tree for S is constructed from $\mathcal{I}_m$, and the time for the entire algorithm is $O(m)$. We will explain

Ukkonen's algorithm by first presenting an $O(m^3)$-time method to build all trees $\mathcal{I}_i$ and then optimizing its implementation to obtain the claimed time bound.

High-level description of Ukkonen's algorithm

Ukkonen's algorithm is divided into *m phases*. In phase $i+1$, tree $\mathcal{I}_{i+1}$ is constructed from $\mathcal{I}_i$. Each phase $i+1$ is further divided into $i+1$ *extensions*, one for each of the $i+1$ suffixes of $S[1..i+1]$. In *extension j* of phase $i+1$, the algorithm first finds the end of the path from the root labeled with substring $S[j..i]$. It then extends the substring by adding the character $S(i+1)$ to its end, unless $S(i+1)$ already appears there. So in phase $i+1$, string $S[1..i+1]$ is first put in the tree, followed by strings $S[2..i+1]$, $S[3..i+1], \ldots$ (in extensions 1,2, 3, ..., respectively). Extension $i+1$ of phase $i+1$ extends the *empty* suffix of $S[1..i]$, that is, it puts the single character string $S(i+1)$ into the tree (unless it is already there). Tree $\mathcal{I}_1$ is just the single edge labeled by character $S(1)$. Procedurally, the algorithm is as follows:

High-level Ukkonen algorithm

Construct tree $\mathcal{I}_1$.
For i from 1 to $m-1$ do
begin {phase $i+1$}
    For j from 1 to $i+1$
        begin {extension j}
        Find the end of the path from the root labeled $S[j..i]$ in the
        current tree. If needed, extend that path by adding character $S(i+1)$,
        thus assuring that string $S[j..i+1]$ is in the tree.
        end;
end;

Suffix extension rules

To turn this high-level description into an algorithm, we must specify exactly how to perform a *suffix extension*. Let $S[j..i] = \beta$ be a suffix of $S[1..i]$. In extension j, when the algorithm finds the end of β in the current tree, it extends β to be sure the suffix $\beta S(i+1)$ is in the tree. It does this according to one of the following three rules:

Rule 1 In the current tree, path β ends at a leaf. That is, the path from the root labeled β extends to the end of some leaf edge. To update the tree, character $S(i+1)$ is added to the end of the label on that leaf edge.

Rule 2 No path from the end of string β starts with character $S(i+1)$, but at least one labeled path continues from the end of β.

In this case, a new leaf edge starting from the end of β must be created and labeled with character $S(i+1)$. A new node will also have to be created there if β ends inside an edge. The leaf at the end of the new leaf edge is given the number j.

Rule 3 Some path from the end of string β starts with character $S(i+1)$. In this case the string $\beta S(i+1)$ is already in the current tree, so (remembering that in an implicit suffix tree the end of a suffix need not be explicitly marked) we do nothing.

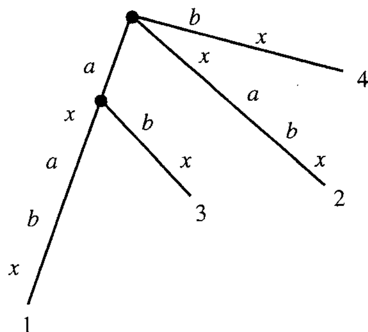

Figure 6.3: Implicit suffix tree for string *axabx* before the sixth character, *b*, is added.

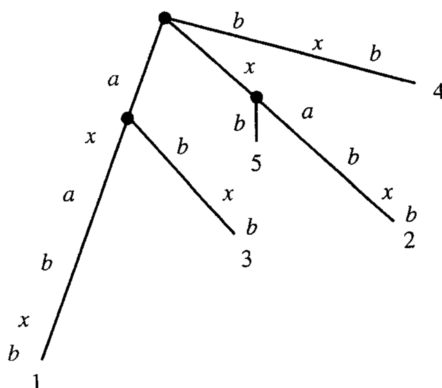

Figure 6.4: Extended implicit suffix tree after the addition of character *b*.

As an example, consider the implicit suffix tree for $S = axabx$ shown in Figure 6.3. The first four suffixes end at leaves, but the single character suffix x ends inside an edge. When a sixth character b is added to the string, the first four suffixes get extended by applications of Rule 1, the fifth suffix gets extended by rule 2, and the sixth by rule 3. The result is shown in Figure 6.4.

6.1.3. Implementation and speedup

Using the suffix extension rules given above, once the end of a suffix β of $S[1..i]$ has been found in the current tree, only constant time is needed to execute the extension rules (to ensure that suffix $\beta S(i+1)$ is in the tree). The key issue in implementing Ukkonen's algorithm then is how to locate the ends of all the $i+1$ suffixes of $S[1..i]$.

Naively we could find the end of any suffix β in $O(|\beta|)$ time by walking from the root of the current tree. By that approach, extension j of phase $i+1$ would take $O(i+1-j)$ time, $\mathcal{I}_{i+1}$ could be created from $\mathcal{I}_i$ in $O(i^2)$ time, and $\mathcal{I}_m$ could be created in $O(m^3)$ time. This algorithm may seem rather foolish since we already know a straightforward algorithm to build a suffix tree in $O(m^2)$ time (and another is discussed in the exercises), but it is easier to describe Ukkonen's $O(m)$ algorithm as a speedup of the $O(m^3)$ method above.

We will reduce the above $O(m^3)$ time bound to $O(m)$ time with a few observations and implementation tricks. Each trick by itself looks like a sensible heuristic to accelerate the algorithm, but acting individually these tricks do not necessarily reduce the worst-case