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 7.10), 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*

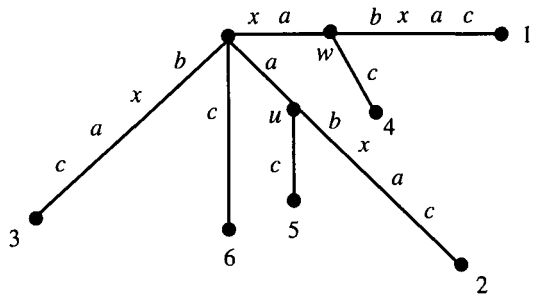

Figure 5.1: Suffix tree for string *xabxac*. The node labels *u* and *w* on the two interior nodes will be used later.

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

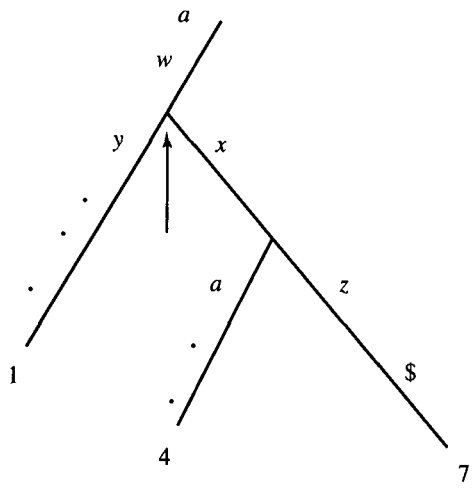

Figure 5.2: Three occurrences of *aw* in *awyawxawxz*. Their starting positions number the leaves in the subtree of the node with path-label *aw*.

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

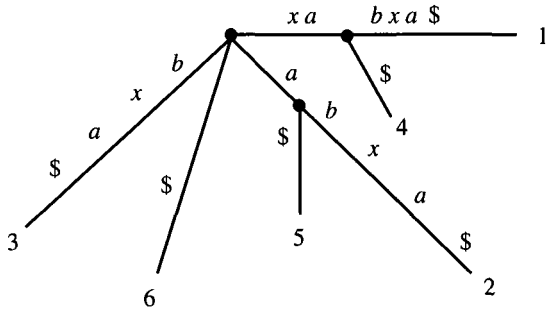

Figure 6.1: Suffix tree for string *xabxa* $.

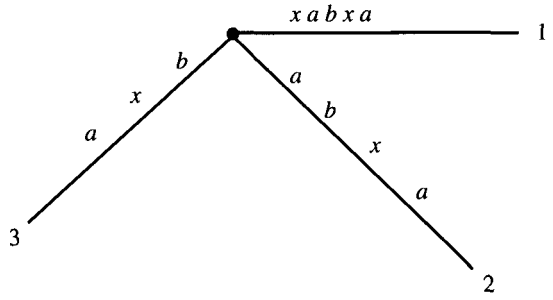

Figure 6.2: Implicit suffix tree for string *xabxa*.

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 $\mathcal{T}_i$ 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 $\mathcal{T}_{i+1}$ is constructed from $\mathcal{T}_i$. 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 $S[2..i+1]$, $S[3..i+1]$, ... (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 $\mathcal{T}_1$ is just the single edge labeled by character $S(1)$. Procedurally, the algorithm is as follows:

High-level Ukkonen algorithm

```
Construct tree T_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.

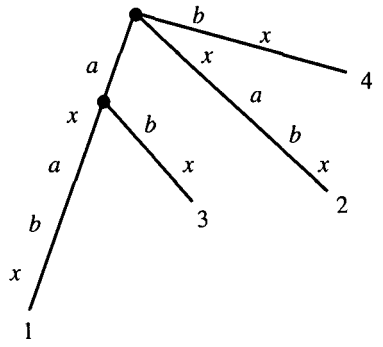

Figure 6.3: Implicit suffix tree for string *axabx* before the sixth character, *b*, is added.

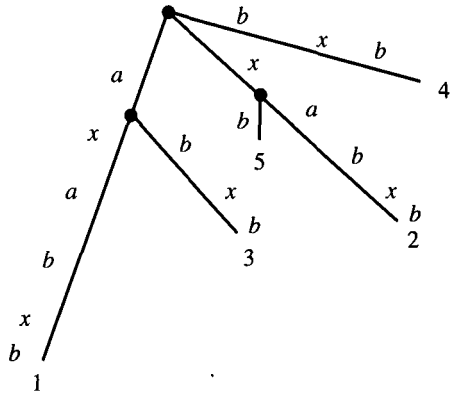

Figure 6.4: Extended implicit suffix tree after the addition of character *b*.

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

bound. However, taken together, they do achieve a linear worst-case time. The most important element of the acceleration is the use of *suffix links*.

Suffix links: first implementation speedup

Definition Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes a (possibly empty) substring. For an internal node v with path-label $x\alpha$, *if* there is another node $s(v)$ with path-label α, then a pointer from v to $s(v)$ is called a *suffix link*.

We will sometimes refer to a suffix link from v to $s(v)$ as the pair $(v, s(v))$. For example, in Figure 6.1 (on page 95) let v be the node with path-label xa and let $s(v)$ be the node whose path-label is the single character a. Then there exists a suffix link from node v to node $s(v)$. In this case, α is just a single character long.

As a special case, if α is empty, then the suffix link from an internal node with path-label $x\alpha$ goes to the root node. The root node itself is not considered internal and has no suffix link from it.

Although definition of suffix links does not imply that every internal node of an implicit suffix tree has a suffix link from it, it will, in fact, have one. We actually establish something stronger in the following lemmas and corollaries.

Lemma 6.1.1. *If a new internal node v with path-label $x\alpha$ is added to the current tree in extension j of some phase $i+1$, then either the path labeled α already ends at an internal node of the current tree or an internal node at the end of string α will be created (by the extension rules) in extension $j+1$ in the same phase $i+1$.*

PROOF A new internal node v is created in extension j (of phase $i+1$) only when extension rule 2 applies. That means that in extension j, the path labeled $x\alpha$ continued with some character other than $S(i+1)$, say c. Thus, in extension $j+1$, there is a path labeled α in the tree and it certainly has a continuation with character c (although possibly with other characters as well). There are then two cases to consider: Either the path labeled α continues only with character c or it continues with some additional character. When α is continued only by c, extension rule 2 will create a node $s(v)$ at the end of path α. When α is continued with two different characters, then there must already be a node $s(v)$ at the end of path α. The Lemma is proved in either case. □

Corollary 6.1.1. In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

PROOF The proof is inductive and is true for tree $\mathcal{I}_1$ since $\mathcal{I}_1$ contains no internal nodes. Suppose the claim is true through the end of phase i, and consider a single phase $i+1$. By Lemma 6.1.1, when a new node v is created in extension j, the correct node $s(v)$ ending the suffix link from v will be found or created in extension $j+1$. No new internal node gets created in the last extension of a phase (the extension handling the single character suffix $S(i+1)$), so all suffix links from internal nodes created in phase $i+1$ are known by the end of the phase and tree $\mathcal{I}_{i+1}$ has all its suffix links. □

Corollary 6.1.1 is similar to Theorem 6.2.5, which will be discussed during the treatment of Weiner's algorithm, and states an important fact about implicit suffix trees and ultimately about suffix trees. For emphasis, we restate the corollary in slightly different language.

Corollary 6.1.2. In any implicit suffix tree $\mathcal{I}_i$, if internal node v has path-label $x\alpha$, then *there is* a node $s(v)$ of $\mathcal{I}_i$ with path-label α.

Following Corollary 6.1.1, all internal nodes in the changing tree will have suffix links from them, except for the most recently added internal node, which will receive its suffix link by the end of the next extension. We now show how suffix links are used to speed up the implementation.

Following a trail of suffix links to build $\mathcal{T}_{i+1}$

Recall that in phase $i+1$ the algorithm locates suffix $S[j..i]$ of $S[1..i]$ in extension j, for j increasing from 1 to $i+1$. Naively, this is accomplished by matching the string $S[j..i]$ along a path from the root in the current tree. Suffix links can shortcut this walk and each extension. The first two extensions (for $j=1$ and $j=2$) in any phase $i+1$ are the easiest to describe.

The end of the full string $S[1..i]$ must end at a leaf of $\mathcal{T}_i$ since $S[1..i]$ is the longest string represented in that tree. That makes it easy to find the end of that suffix (as the trees are constructed, we can keep a pointer to the leaf corresponding to the current full string $S[1..i]$), and its suffix extension is handled by Rule 1 of the extension rules. So the first extension of any phase is special and only takes constant time since the algorithm has a pointer to the end of the current full string.

Let string $S[1..i]$ be $x\alpha$, where x is a single character and α is a (possibly empty) substring, and let $(v, 1)$ be the tree-edge that enters leaf 1. The algorithm next must find the end of string $S[2..i] = \alpha$ in the current tree derived from $\mathcal{T}_i$. The key is that node v is either the root or it is an interior node of $\mathcal{T}_i$. If it is the root, then to find the end of α the algorithm just walks down the tree following the path labeled α as in the naive algorithm. But if v is an internal node, then by Corollary 6.1.2 (since v was in $\mathcal{T}_i$) v has a suffix link out of it to node $s(v)$. Further, since $s(v)$ has a path-label that is a prefix of string α, the end of string α must end in the subtree of $s(v)$. Consequently, in searching for the end of α in the current tree, the algorithm need not walk down the entire path from the root, but can instead begin the walk from node $s(v)$. That is the main point of including suffix links in the algorithm.

To describe the second extension in more detail, let γ denote the edge-label on edge $(v, 1)$. To find the end of α, walk up from leaf 1 to node v; follow the suffix link from v to $s(v)$; and walk from $s(v)$ down the path (which may be more than a single edge) labeled γ. The end of that path is the end of α (see Figure 6.5). At the end of path α, the tree is updated following the suffix extension rules. This completely describes the first two extensions of phase $i+1$.

To extend any string $S[j..i]$ to $S[j..i+1]$ for $j > 2$, repeat the same general idea: Starting at the end of string $S[j-1..i]$ in the current tree, walk up at most one node to either the root or to a node v that has a suffix link from it; let γ be the edge-label of that edge; assuming v is not the root, traverse the suffix link from v to $s(v)$; then walk down the tree from $s(v)$, following a path labeled γ to the end of $S[j..i]$; finally, extend the suffix to $S[j..i+1]$ according to the extension rules.

There is one minor difference between extensions for $j > 2$ and the first two extensions. In general, the end of $S[j-1..i]$ may be at a node that itself has a suffix link from it, in which case the algorithm traverses that suffix link. Note that even when extension rule 2 applies in extension $j-1$ (so that the end of $S[j-1..i]$ is at a newly created internal node w), if the parent of w is not the root, then the parent of w already has a suffix link out of it, as guaranteed by Lemma 6.1.1. Thus in extension j the algorithm never walks up more than one edge.

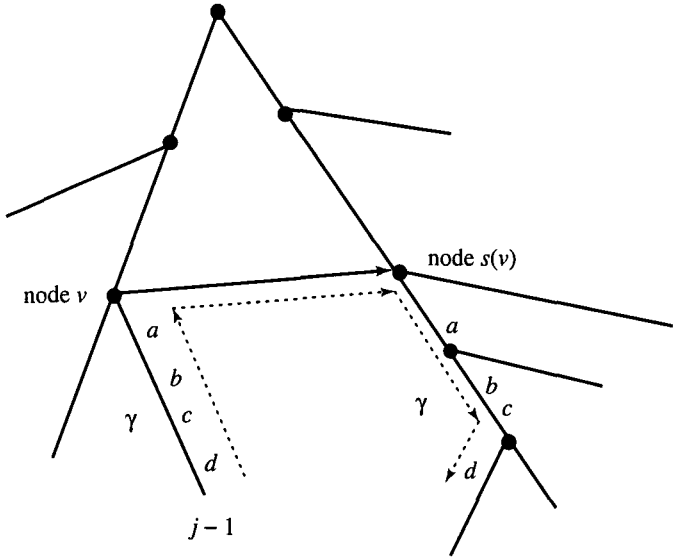

Figure 6.5: Extension $j > 1$ in phase $i+1$. Walk up atmost one edge (labeled γ) from the end of the path labeled $S[j-1..i]$ to node v; then follow the suffix link to $s(v)$; then walk down the path specifying substring γ; then apply the appropriate extension rule to insert suffix $S[j..i+1]$.

Single extension algorithm: SEA

Putting these pieces together, when implemented using suffix links, extension $j \geq 2$ of phase $i+1$ is:

Single extension algorithm

Begin

1. Find the first node v at or above the end of $S[j-1..i]$ that either has a suffix link from it or is the root. This requires walking up at most one edge from the end of $S[j-1..i]$ in the current tree. Let γ (possibly empty) denote the string between v and the end of $S[j-1..i]$.
2. If v is not the root, traverse the suffix link from v to node $s(v)$ and then walk down from $s(v)$ following the path for string γ. If v is the root, then follow the path for $S[j..i]$ from the root (as in the naive algorithm).
3. Using the extension rules, ensure that the string $S[j..i]S(i+1)$ is in the tree.
4. If a new internal node w was created in extension $j-1$ (by extension rule 2), then by Lemma 6.1.1, string α must end at node $s(w)$, the end node for the suffix link from w. Create the suffix link $(w, s(w))$ from w to $s(w)$.

End.

Assuming the algorithm keeps a pointer to the current full string $S[1..i]$, the first extension of phase $i+1$ need not do any up or down walking. Furthermore, the first extension of phase $i+1$ always applies suffix extension rule 1.

What has been achieved so far?

The use of suffix links is clearly a practical improvement over walking from the root in each extension, as done in the naive algorithm. But does their use improve the worst-case running time?

The answer is that as described, the use of suffix links does not yet improve the time bound. However, here we introduce a trick that will reduce the worst-case time for the

algorithm to $O(m^2)$. This trick will also be central in other algorithms to build and use suffix trees.

Trick number 1: skip/count trick

In Step 2 of extension $j+1$ the algorithm walks *down* from node $s(v)$ along a path labeled γ. Recall that there surely must be such a γ path from $s(v)$. Directly implemented, this walk along γ takes time proportional to $|\gamma|$, the *number of characters* on that path. But a simple trick, called the *skip/count trick*, will reduce the traversal time to something proportional to the *number of nodes* on the path. It will then follow that the time for all the down walks in a phase is at most $O(m)$.

Trick 1 Let g denote the length of γ, and recall that no two labels of edges out of $s(v)$ can start with the same character, so the first character of γ must appear as the first character on exactly one edge out of $s(v)$. Let g' denote the number of characters on that edge. If g' is less than g, then the algorithm does not need to look at any more of the characters on that edge; it simply skips to the node at the end of the edge. There it sets g to $g - g'$, sets a variable h to $g' + 1$, and looks over the outgoing edges to find the correct next edge (whose first character matches character h of γ). In general, when the algorithm identifies the next edge on the path it compares the current value of g to the number of characters g' on that edge. When g is at least as large as g', the algorithm skips to the node at the end of the edge, sets g to $g - g'$, sets h to $h + g'$, and finds the edge whose first character is character h of γ and repeats. When an edge is reached where g is smaller than or equal to g', then the algorithm skips to character g on the edge and quits, assured that the γ path from $s(v)$ ends on that edge exactly g characters down its label. (See Figure 6.6).

Assuming simple and obvious implementation details (such as knowing the number of characters on each edge, and being able, in constant time, to extract from S the character at any given position) the effect of using the skip/count trick is to move from one node to the next node on the γ path in *constant* time.[1] The total time to traverse the path is then proportional to the number of *nodes* on it rather than the number of characters on it.

This is a useful heuristic, but what does it buy in terms of worst-case bounds? The next lemma leads immediately to the answer.

Definition Define the *node-depth* of a node u to be the number of *nodes* on the path from the root to u.

Lemma 6.1.2. *Let $(v, s(v))$ be any suffix link traversed during Ukkonen's algorithm. At that moment, the node-depth of v is at most one greater than the node depth of $s(v)$.*

PROOF When edge $(v, s(v))$ is traversed, any internal ancestor of v, which has path-label $x\beta$ say, has a suffix link to a node with path-label β. But $x\beta$ is a prefix of the path to v, so β is a prefix of the path to $s(v)$ and it follows that the suffix link from any internal ancestor of v goes to an ancestor of $s(v)$. Moreover, if β is nonempty then the node labeled by β is an internal node. And, because the node-depths of any two ancestors of v must differ, each ancestor of v has a suffix link to a distinct ancestor of $s(v)$. It follows that the node-depth of $s(v)$ is at least one (for the root) plus the number of internal ancestors of v who have path-labels more than one character long. The only extra ancestor that v can have (without a corresponding ancestor for $s(v)$) is an internal ancestor whose path-label

[1] Again, we are assuming a constant-sized alphabet.

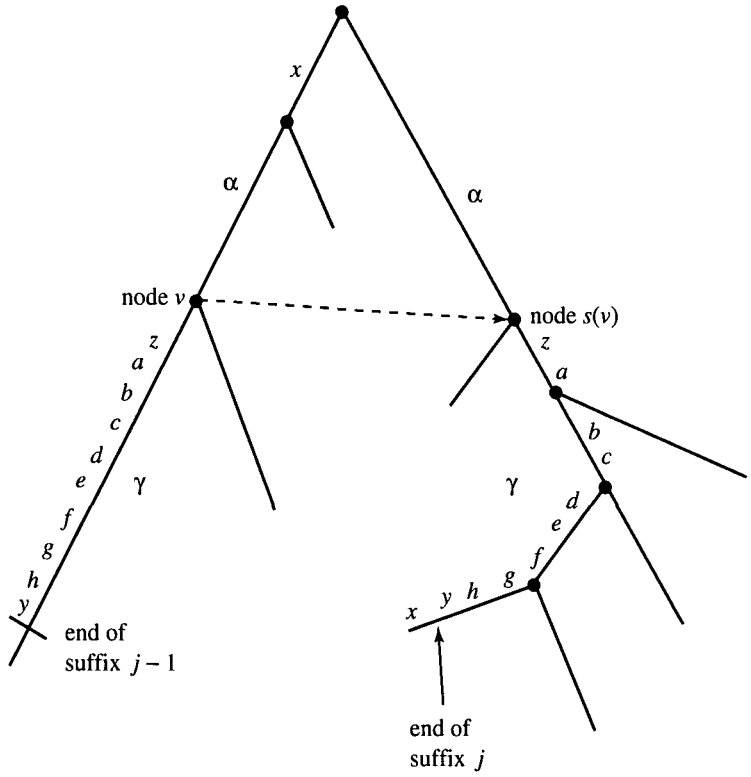

Figure 6.6: The skip/count trick. In phase $i + 1$, substring γ has length ten. There is a copy of substring γ out of node $s(v)$; it is found three characters down the last edge, after four node skips are executed.

has length one (it has label x). Therefore, v can have node-depth at most one more than $s(v)$. (See Figure 6.7). □

Definition As the algorithm proceeds, the *current node-depth* of the algorithm is the node depth of the node most recently visited by the algorithm.

Theorem 6.1.1. *Using the skip/count trick, any phase of Ukkonen's algorithm takes $O(m)$ time.*

PROOF There are $i + 1 \leq m$ extensions in phase i. In a single extension the algorithm walks up at most one edge to find a node with a suffix link, traverses one suffix link, walks down some number of nodes, applies the suffix extension rules, and maybe adds a suffix link. We have already established that all the operations other than the down-walking take constant time per extension, so we only need to analyze the time for the down-walks. We do this by examining how the current node-depth can change over the phase.

The up-walk in any extension decreases the current node-depth by at most one (since it moves up at most one node), each suffix link traversal decreases the node-depth by at most another one (by Lemma 6.1.2), and each edge traversed in a down-walk moves to a node of greater node-depth. Thus over the entire phase the current node-depth is decremented at most $2m$ times, and since no node can have depth greater than m, the total possible increment to current node-depth is bounded by $3m$ over the entire phase. It follows that over the entire phase, the total number of edge traversals during down-walks is bounded by $3m$. Using the skip/count trick, the time per down-edge traversal is constant, so the total time in a phase for all the down-walking is $O(m)$, and the theorem is proved. □

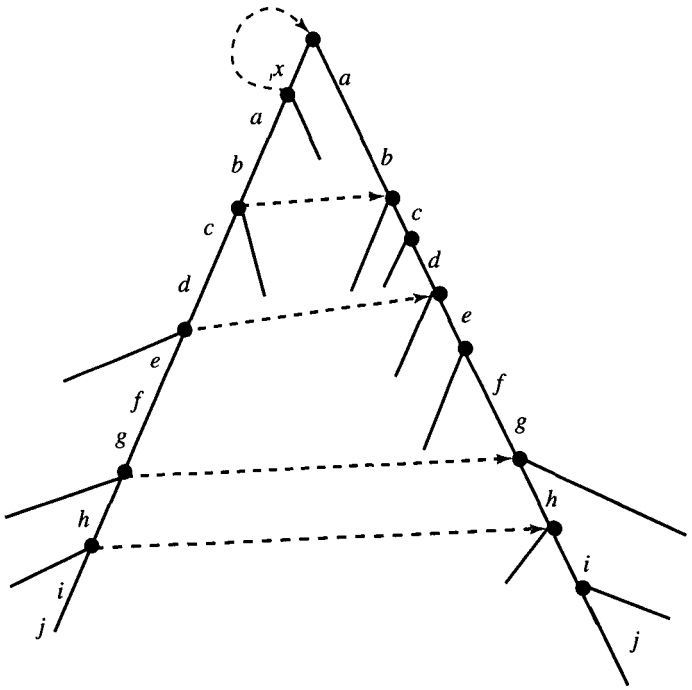

Figure 6.7: For every node v on the path $x\alpha$, the corresponding node $s(v)$ is on the path α. However, the node-depth of $s(v)$ can be one less than the node-depth of v, it can be equal, or it can be greater. For example, the node labeled *xab* has node-depth two, whereas the node-depth of *ab* is one. The node-depth of the node labeled *xabcdefg* is four, whereas the node-depth of *abcdefg* is five.

There are m phases, so the following is immediate:

Corollary 6.1.3. Ukkonen's algorithm can be implemented with suffix links to run in $O(m^2)$ time.

Note that the $O(m^2)$ time bound for the algorithm was obtained by multiplying the $O(m)$ time bound on a single phase by m (since there are m phases). This crude multiplication was necessary because the time analysis was directed to only a single phase. What is needed are some changes to the implementation allowing a time analysis that crosses phase boundaries. That will be done shortly.

At this point the reader may be a bit weary because we seem to have made no progress, since we started with a naive $O(m^2)$ method. Why all the work just to come back to the same time bound? The answer is that although we have made no progress on the time bound, we have made great conceptual progress so that with only a few more easy details, the time will fall to $O(m)$. In particular, we will need one simple implementation detail and two more little tricks.

6.1.4. A simple implementation detail

We next establish an $O(m)$ time bound for building a suffix tree. There is, however, one immediate barrier to that goal: The suffix tree may require $\Theta(m^2)$ space. As described so far, the edge-labels of a suffix tree might contain more than $\Theta(m)$ characters in total. Since the time for the algorithm is at least as large as the size of its output, that many characters makes an $O(m)$ time bound impossible. Consider the string $S = abcdefghijklmnopqrstuvwxyz$. Every suffix begins with a distinct character; hence there are 26 edges out of the root and

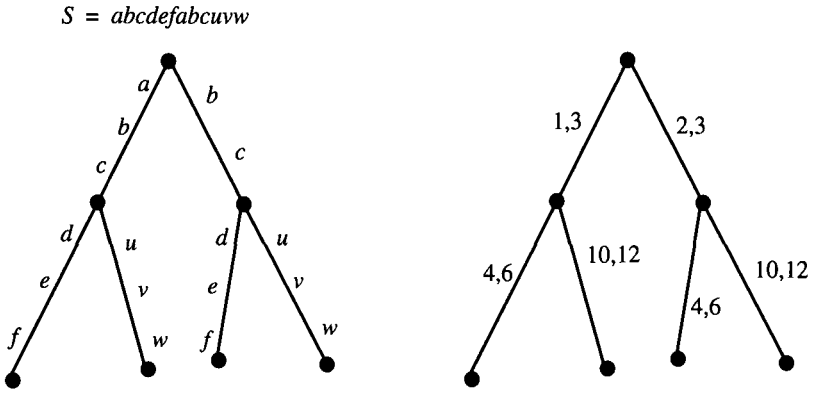

Figure 6.8: The left tree is a fragment of the suffix tree for string $S = abcdefabcuvw$, with the edge-labels written explicitly. The right tree shows the edge-labels compressed. Note that that edge with label 2, 3 could also have been labeled 8, 9.

each is labeled with a complete suffix, requiring $26 \times 27/2$ characters in all. For strings longer than the alphabet size, some characters will repeat, but still one can construct strings of arbitrary length m so that the resulting edge-labels have more than $\Theta(m)$ characters in total. Thus, an $O(m)$-time algorithm for building suffix trees requires some alternate scheme to represent the edge-labels.

Edge-label compression

A simple, alternate scheme exists for edge labeling. Instead of explicitly writing a substring on an edge of the tree, only write a *pair of indices* on the edge, specifying beginning and end positions of that substring in S (see Figure 6.8). Since the algorithm has a copy of string S, it can locate any particular character in S in constant time given its position in the string. Therefore, we may describe any particular suffix tree algorithm as if edge-labels were explicit, and yet implement that algorithm with only a constant number of symbols written on any edge (the index pair indicating the beginning and ending positions of a substring).

For example, in Ukkonen's algorithm when matching along an edge, the algorithm uses the index pair written on an edge to retrieve the needed characters from S and then performs the comparisons on those characters. The extension rules are also easily implemented with this labeling scheme. When extension rule 2 applies in a phase $i+1$, label the newly created edge with the index pair $(i+1, i+1)$, and when extension rule 1 applies (on a leaf edge), change the index pair on that leaf edge from (p, q) to $(p, q+1)$. It is easy to see inductively that q had to be i and hence the new label $(p, i+1)$ represents the correct new substring for that leaf edge.

By using an index pair to specify an edge-label, only two numbers are written on any edge, and since the number of edges is at most $2m - 1$, the suffix tree uses only $O(m)$ symbols and requires only $O(m)$ space. This makes it more plausible that the tree can actually be built in $O(m)$ time.[2] Although the fully implemented algorithm will not explicitly write a substring on an edge, we will still find it convenient to talk about "the substring or label on an edge or path" as if the explicit substring was written there.

[2] We make the standard RAM model assumption that a number with up to $\log m$ bits can be read, written, or compared in constant time.

6.1.5. Two more little tricks and we're done

We present two more implementation tricks that come from two observations about the way the extension rules interact in successive extensions and phases. These tricks, plus Lemma 6.1.2, will lead immediately to the desired linear time bound.

Observation 1: Rule 3 is a show stopper In any phase, if suffix extension rule 3 applies in extension j, it will also apply in all further extensions ($j+1$ to $i+1$) until the end of the phase. The reason is that when rule 3 applies, the path labeled $S[j..i]$ in the current tree must continue with character $S(i+1)$, and so the path labeled $S[j+1..i]$ does also, and rule 3 again applies in extensions $j+1, j+2, \ldots, i+1$.

When extension rule 3 applies, no work needs to be done since the suffix of interest is already in the tree. Moreover, a new suffix link needs to be added to the tree only after an extension in which extension rule 2 applies. These facts and Observation 1 lead to the following implementation trick.

Trick 2 End any phase $i+1$ the first time that extension rule 3 applies. If this happens in extension j, then there is no need to explicitly find the end of any string $S[k..i]$ for $k > j$.

The extensions in phase $i+1$ that are "done" after the first execution of rule 3 are said to be done *implicitly*. This is in contrast to any extension j where the end of $S[j..i]$ is explicitly found. An extension of that kind is called an *explicit* extension.

Trick 2 is clearly a good heuristic to reduce work, but it's not clear if it leads to a better worst-case time bound. For that we need one more observation and trick.

Observation 2: Once a leaf, always a leaf That is, if at some point in Ukkonen's algorithm a leaf is created and labeled j (for the suffix starting at position j of S), then that leaf will remain a leaf in all successive trees created during the algorithm. This is true because the algorithm has no mechanism for extending a leaf edge beyond its current leaf. In more detail, once there is a leaf labeled j, extension rule 1 will always apply to extension j in any successive phase. So once a leaf, always a leaf.

Now leaf 1 is created in phase 1, so in any phase i there is an initial sequence of consecutive extensions (starting with extension 1) where extension rule 1 or 2 applies. Let j_i denote the last extension in this sequence. Since any application of rule 2 creates a new leaf, it follows from Observation 2 that $j_i \leq j_{i+1}$. That is, the initial sequence of extensions where rule 1 or 2 applies cannot shrink in successive phases. This suggests an implementation trick that in phase $i+1$ avoids all explicit extensions 1 through j_i. Instead, only constant time will be required to do those extensions implicitly.

To describe the trick, recall that the label on any edge in an implicit suffix tree (or a suffix tree) can be represented by two indices p, q specifying the substring $S[p..q]$. Recall also that for any leaf edge of $\mathcal{I}_i$, index q is equal to i and in phase $i+1$ index q gets incremented to $i+1$, reflecting the addition of character $S(i+1)$ to the end of each suffix.

Trick 3 In phase $i+1$, when a leaf edge is first created and would normally be labeled with substring $S[p..i+1]$, instead of writing indices $(p, i+1)$ on the edge, write (p, e), where e is a symbol denoting "the current end". Symbol e is a *global* index that is set to $i+1$ once in each phase. In phase $i+1$, since the algorithm knows that rule 1 will apply in extensions 1 through j_i at least, it need do no additional explicit work to implement

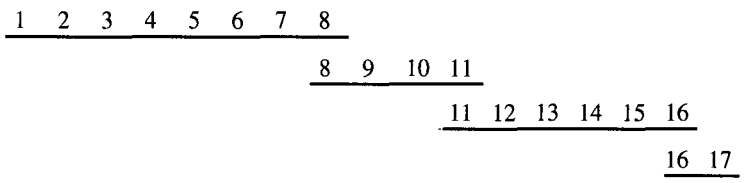

Figure 6.9: Cartoon of a possible execution of Ukkonen's algorithm. Each line represents a phase of the algorithm, and each number represents an explicit extension executed by the algorithm. In this cartoon there are four phases and seventeen explicit extensions. In any two consecutive phases, there is at most one index where the same explicit extension is executed in both phases.

those j_i extensions. Instead, it only does constant work to increment variable e, and then does explicit work for (some) extensions starting with extension $j_i + 1$.

The punch line

With Tricks 2 and 3, explicit extensions in phase $i + 1$ (using algorithm SEA) are then only required from extension $j_i + 1$ until the first extension where rule 3 applies (or until extension $i + 1$ is done). All other extensions (before and after those explicit extensions) are done implicitly. Summarizing this, phase $i + 1$ is implemented as follows:

Single phase algorithm: SPA

Begin

1. Increment index e to $i + 1$. (By Trick 3 this correctly implements all implicit extensions 1 through j_i.)
2. Explicitly compute successive extensions (using algorithm SEA) starting at $j_i + 1$ until reaching the first extension j^* where rule 3 applies or until all extensions are done in this phase. (By Trick 2, this correctly implements all the additional implicit extensions $j^* + 1$ through $i + 1$.)
3. Set j_{i+1} to $j^* - 1$, to prepare for the next phase.

End

Step 3 correctly sets j_{i+1} because the initial sequence of extensions where extension rule 1 or 2 applies must end at the point where rule 3 first applies.

The key feature of algorithm *SPA* is that phase $i + 2$ will *begin* computing explicit extensions with extension j^*, where j^* was the *last* explicit extension computed in phase $i + 1$. Therefore, two consecutive phases share *at most one* index (j^*) where an explicit extension is executed (see Figure 6.9). Moreover, phase $i + 1$ ends knowing where string $S[j^*..i + 1]$ ends, so the repeated extension of j^* in phase $i + 2$ can execute the suffix extension rule for j^* without any up-walking, suffix link traversals, or node skipping. That means the first explicit extension in any phase only takes constant time. It is now easy to prove the main result.

Theorem 6.1.2. *Using suffix links and implementation tricks* 1, 2, *and* 3, *Ukkonen's algorithm builds implicit suffix trees* $\mathcal{I}_1$ *through* $\mathcal{I}_m$ *in* $O(m)$ *total time.*

PROOF The time for all the implicit extensions in any phase is constant and so is $O(m)$ over the entire algorithm.

As the algorithm executes explicit extensions, consider an index $\bar{j}$ corresponding to the explicit extension the algorithm is currently executing. Over the entire execution of the algorithm, $\bar{j}$ never decreases, but it does remain the same between two successive phases.

Since there are only m phases, and $\bar{j}$ is bounded by m, the algorithm therefore executes only $2m$ explicit extensions. As established earlier, the time for an explicit extension is a constant plus some time proportional to the number of node skips it does during the down-walk in that extension.

To bound the total number of node skips done during all the down-walks, we consider (similar to the proof of Theorem 6.1.1) how the current node-depth changes during successive extensions, even extensions in different phases. The key is that the first explicit extension in any phase (after phase 1) begins with extension j^*, which was the last explicit extension in the previous phase. Therefore, the current node-depth does not change between the end of one extension and the beginning of the next. But (as detailed in the proof of Theorem 6.1.1), in each explicit extension the current node-depth is first reduced by at most two (up-walking one edge and traversing one suffix link), and thereafter the down-walk in that extension increases the current node-depth by one at each node skip. Since the maximum node-depth is m, and there are only $2m$ explicit extensions, it follows (as in the proof of Theorem 6.1.1) that the maximum number of node skips done during all the down-walking (and not just in a single phase) is bounded by $O(m)$. All work has been accounted for, and the theorem is proved. □

6.1.6. Creating the true suffix tree

The final implicit suffix tree $\mathcal{I}_m$ can be converted to a true suffix tree in $O(m)$ time. First, add a string terminal symbol \$ to the end of S and let Ukkonen's algorithm continue with this character. The effect is that no suffix is now a prefix of any other suffix, so the execution of Ukkonen's algorithm results in an implicit suffix tree in which each suffix ends at a leaf and so is explicitly represented. The only other change needed is to replace each index e on every leaf edge with the number m. This is achieved by an $O(m)$-time traversal of the tree, visiting each leaf edge. When these modifications have been made, the resulting tree is a true suffix tree.

In summary,

Theorem 6.1.3. *Ukkonen's algorithm builds a true suffix tree for S, along with all its suffix links in $O(m)$ time.*

6.2. Weiner's linear-time suffix tree algorithm

Unlike Ukkonen's algorithm, Weiner's algorithm starts with the entire string S. However, like Ukkonen's algorithm, it enters one suffix at a time into a growing tree, although in a very different order. In particular, it first enters string $S(m)\$$ into the tree, then string $S[m-1..m]\$, \ldots,$ and finally, it enters the entire string $S\$$ into the tree.

Definition Suff_i denotes the suffix $S[i..m]$ of S starting in position i.

For example, Suff_1 is the entire string S, and Suff_m is the single character $S(m)$.

Definition Define $\mathcal{T}_i$ to be the tree that has $m-i+2$ leaves numbered i through $m+1$ such that the path from the root to any leaf j ($i \leq j \leq m+1$) has label $\text{Suff}_j\$$. That is, $\mathcal{T}_i$ is a tree encoding all and only the suffixes of string $S[i..m]\$$, so it is a suffix tree of string $S[i..m]\$$.

Weiner's algorithm constructs trees from $\mathcal{T}_{m+1}$ *down* to $\mathcal{T}_1$ (i.e., in decreasing order of i). We will first implement the method in a straightforward inefficient way. This will