

## Algorithms on Strings, Trees, and Sequences

Dan Gusfield

University of California, Davis

Cambridge University Press

1997

### 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  $I_i$  for  $i$  from 1 to  $m$ .

The implicit suffix tree for any string  $S$  will have fewer leaves than the suffix tree for string  $S\$$  if and only if at least one of the suffixes of  $S$  is a prefix of another suffix.



constructed from  $I_m$  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  $I_i$  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  $I_{i+1}$  is constructed from  $I_i$ . 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  $S[2..i + 1]$ ,  $S[3..i + 1]$ , ... (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  $I_i$  is just the single edge labeled by character  $S(1)$ . Procedurally, the algorithm is as follows:

#### High-level Ukkonen algorithm

Construct tree  $I_i$ .

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

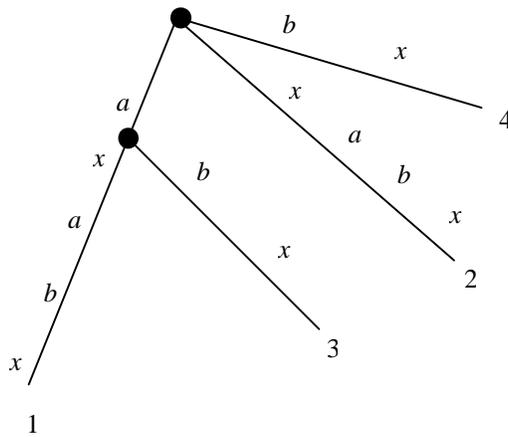
To turn this high-level description into an algorithm, we must specify exactly how to perform a *suffix extension*. Let  $S[j..i] = \beta$  be a suffix of  $S[1..i]$ . In extension  $j$ , when the algorithm finds the end of  $\beta$  in the current tree, it extends  $\beta$  to be sure the suffix  $\beta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

**Rule 1** In the current tree, path  $\beta$  ends at a leaf. That is, the path from the root labeled  $\beta$  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.

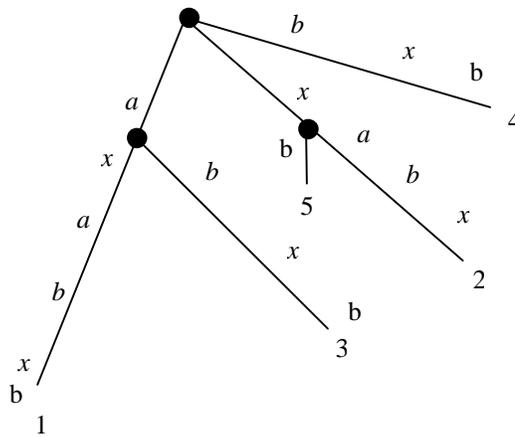
**Rule 2** No path from the end of string  $\beta$  starts with character  $S(i + 1)$ , but at least one labeled path continues from the end of  $\beta$ .

In this case, a new leaf edge starting from the end of  $\beta$  must be created and labeled with character  $S(i + 1)$ . A new node will also have to be created there if,  $\beta$  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  $\beta$  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  $\beta$  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  $\beta$  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,  $I_{i+1}$  could be created from  $I_i$  in  $O(i^2)$  time, and  $I_i$  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  $\alpha$  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  $\alpha$ , 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 (from page 95) let  $v$  be the node with path-label  $x\alpha$  and let  $s(v)$  be the node whose path-label is the single character  $\alpha$ . Then there exists a suffix link from node  $v$  to node  $s(v)$ . In this case, is just a single character long.

As a special case, if  $\alpha$  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  $\alpha$  already ends at an internal node of the current tree or an internal node at the end of string  $\alpha$  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  $\alpha$  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  $\alpha$  continues only with character  $c$  or it continues with some additional character. When  $\alpha$  is continued only by  $c$ , extension rule 2 will create a node  $s(v)$  at the end of path  $\alpha$ . When  $\alpha$  is continued with two different characters, then there must already be a node  $s(v)$  at the end of path  $\alpha$ . 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  $I_i$ , since  $I_i$  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  $I_{i+1}$  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  $I_i$ , if internal node  $v$  has path-label  $x\alpha$ , then there is a node  $s(v)$  of  $I_i$  with path-label  $\alpha$ .

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 $I_{i+1}$

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  $S[j..i]$  must end at a leaf of  $I_i$  since  $S[1..j]$  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 I 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  $\alpha$  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  $v$ . The key is that node  $v$  is either the root or it is an interior node  $I_i$ . If it is the root, then to find the end of  $\alpha$  the algorithm just walks down the tree following the path labeled  $\alpha$  as in the naive algorithm. But if  $v$  is an internal node, then by Corollary 6.1.2 (since  $v$  was in  $I_i$ )  $v$  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  $\alpha$ , the end of string  $\alpha$  must end at the subtree of  $s(v)$ . Consequently, in searching for the end of  $\alpha$  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  $\gamma$  denote the edge-label on edge  $(v, 1)$ . To find the end of  $\alpha$ , 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  $\gamma$ . The end of that path is the end of  $\alpha$  (see Figure 6.5). At the end of path  $\alpha$ , 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  $\gamma$  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  $\gamma$  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 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  $\gamma$ . Recall that there surely must be such a  $\gamma$  path from  $s(v)$ . Directly implemented, this walk along  $\gamma$  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  $\gamma$ , and recall that no two labels of edges out of  $s(v)$  can start with the same character, so the first character of  $\gamma$  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  $\gamma$ ). 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  $\gamma$  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  $\gamma$  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  $\gamma$  path in *constant* time.<sup>1</sup> 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  $s(v)$ .*

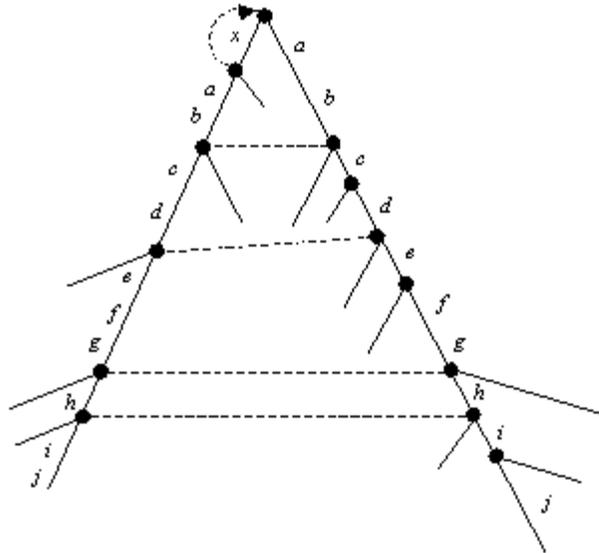
---

<sup>1</sup> Again, we are assuming a constant-sized alphabet.



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 tick, 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.

There are  $m$  phases, so the following is immediate:



**Figure 6.7:** For every node  $v$  on the path  $x\alpha$ , the corresponding node  $s(v)$  is on the path  $\alpha$ . However, the node-depth of  $s(v)$  can be one less than the node-depth of  $v$ ,  $h$  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  $abcdefg$  is five, whereas the node-depth of  $abodeig$  is six.

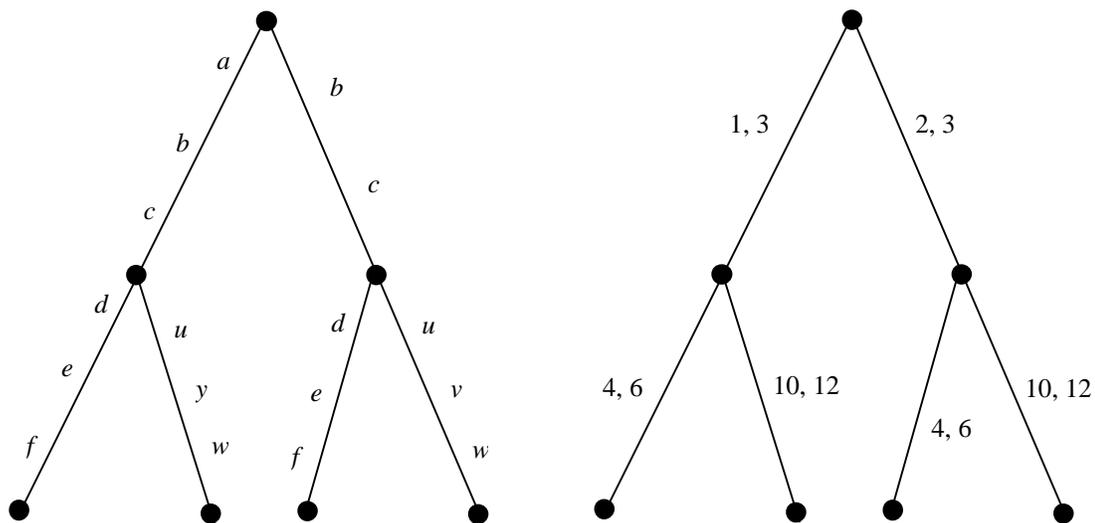
**Coronary 6.1.3.** Ukkonen's algorithm can be implemented with suffix links to run in  $O(m^2)$  time.

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  $S = abcdefghzjlmnopqrstuvwxyz$ . Every suffix begins with a distinct character; hence there are 26 edges out of the root and 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  $n$  so that the resulting edge-labels have more than  $e(m)$  characters in total. Thus, an  $O(m)$ -time algorithm for building suffix trees requires some alternate scheme to represent the edge-labels.



**Figure 6.8:** The left tree is a fragment of the suffix tree for strings  $S = abcdefabcuvw$ , 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.

#### 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.<sup>2</sup> 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.

### 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 characters  $s(i + 1)$ , and so the path labeled  $S[j + 1..i]$  does also, and rule 3 again applies in extensions  $j + 1, j + 2, \dots, i + 1$ .

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 phrase  $i + 1$  the first time that extension rule 3 applies. If this happens in extension  $k$ , 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  $j$  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.

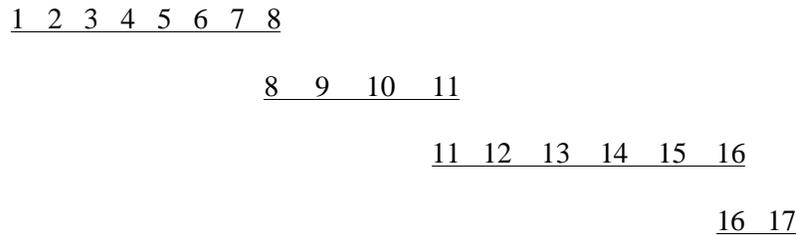
To describe the trick, recall that the label on any edge in an implicit suffix tree (or a suffix tree) can be represented by two indices  $p, q$  specifying the substring  $S[p..q]$ .

---

<sup>2</sup> 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.

Recall also that for any leaf edge of  $I_i$ , 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  $i$  through  $j$ , at least, it need do no additional explicit work to implement 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  $j_i + 1$ .



**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.

### 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  $S[j^*..j + 1]$  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  $I_i$  through  $I_m$ , 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  $\hat{j}$  corresponding to the explicit extension the algorithm is currently executing. Over the entire execution of the algorithm,  $\hat{j}$  never decreases, but it does remain the same between two successive phases.

Since there are only  $m$  phases, and  $\hat{j}$  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  $I_m$  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  $in$ . 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.*