Chapter 1

Suffix Trees

Suffix trees constitute a well understood, extremely elegant, but poorly appreciated, data structure with potentially many applications in language processing. A suffix tree is a data structure constructed from a text whose size is a linear function of the length of the text and which can also be constructed in linear time. That the size is linear in the length of the text—some 25 to 30 bytes per character of text—is fairly obvious from a simple characterization of the structure. That is can be built in linear time is more surprising.

In situations where it is expected that many searches of a long text will be required, so that preprocessing the text at reasonable cost should be easily repaid, it provides for matching of k different patterns constituting a string of length p when concatenated, in time $O(p+m)$, where m is the total number of matches found. It provides for locating interestingly long strings that occur in the text more than a given number of times. By virtue of this capability, it plays a crucial role in some text compression techniques because it allows repeated sequences of characters to be represented explicitly only once, other instances being replaced by a reference to that instance of the sequence. Suffix trees have also been used to detect plagiarism by simply concatenating the strings involved and looking for sufficiently long strings that occur in more than one of them.

Suffix trees were first introduced in [4], a paper which Donald Knuth characterized as "Algorithm of the Year 1973". A greatly simplified version of algorithm was proposed in [2] and [3]. Ukkonen provided the first linear-time online construction of suffix trees, now known as "Ukkonen's algorithm". This latter is generally regarded as the easiest to understand, and it may have marginal advantages in efficiency. It is the algorithm we

describe in this chapter.

We begin our exploration of suffix trees by considering a related but somewhat simpler structure called a *suffix trie*. A trie [1] is a very well-known data structure often used for storing words so that they can be looked up easily. We will refer to it by the alternative name *Digital Search Tree (DST)* to avoid hesitation about how it should be pronounced. A DST is a data structure that represents a set of strings. It is equivalent to a deterministic finite-state automaton whose transition diagram has the shape of a tree and in which the symbols encountered on a path from its initial state to a final state spell out a one of the strings in the set. In a suffix DST, that set consists of the suffixes of some text. We use the term "suffix" here, not in its morphological sense, but to refer to the set of strings each of which begins at a given character in a text, and extends all the way to the end. A text of n characters thus contains exactly $n+1$ suffixes, including the empty string.

Consider the text consisting of the single word "mississippi". The corresponding suffix DST is the one depicted in Fig. 1.1. The labels on every path from the root of the tree, on the left of the diagram, to a terminal node on the right spells out some suffix of the text, and there is such a path for every suffix. Since every substring of the text is a prefix of some suffix, it follows that every substring is spelled out by the characters on the path from the root to some node, terminal or nonterminal. An extra character, called a *sentinel*, which is known not to occur otherwise in the text, is generally added to the end to ensure that every suffix ends at a terminal node. We have used the dollar sign for this purpose.

The trouble with DST's is that they are totally impractical for large texts because of their sheer size and the time it takes to build them. However, these things are suddenly brought under control in the passage from DST's to suffix trees. This is accomplished in a small number of simple steps. The first consists in replacing every sequence of arcs and non-branching nodes by a single arc labelled with the corresponding sequence of characters. The second is to represent this sequence, not by the characters themselves, but by a pair of integers giving the beginning and end points of an occurrence of the string in the text. Nothing is lost by representing the tree in this way, because points represented by non-branching nodes can be reconstructed trivially from the branching nodes and pairs of integers on the edges. If we foresee any confusion, we will refer to trees represented in this way as *compact* trees. The results of performing this operation on the DST in Figure 1.1 are shown in Figure 1.2.

The importance of this compacting transformation cannot be overstated.

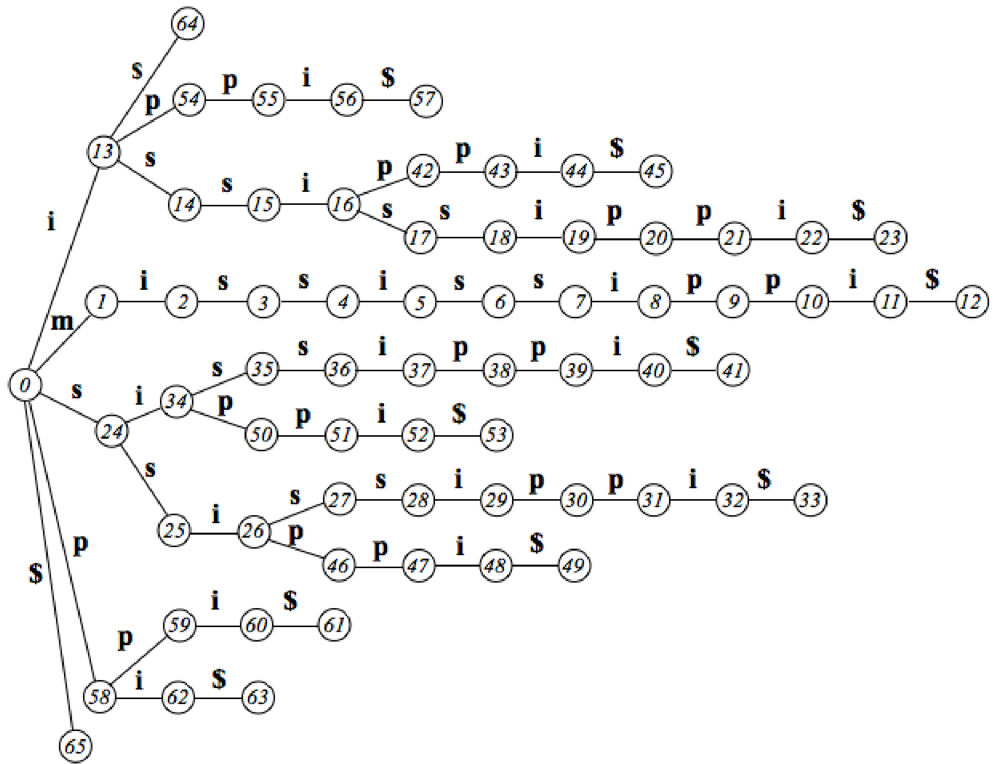

Figure 1.1: A Suffix Tree

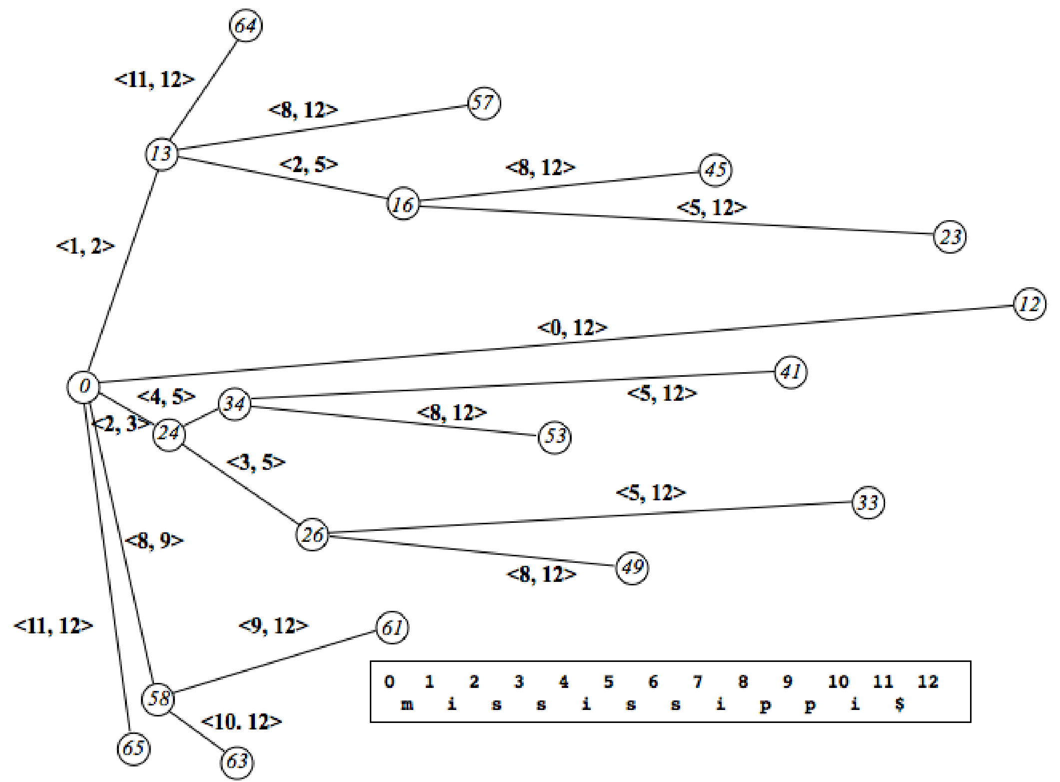

Figure 1.2: A Suffix Tree

The number of transitions in the uncompacted DST is

$$\binom{k+1}{2}$$

for a text of k characters. This is because a substring can begin or endat the point just before any one of the k characters, and also following the last character, for a total of $k+1$. The formula gives the number of ways of choosing two of these points, one at which the string will begin and the other at which it will end.

Every substring uniquely identifies a node in the DST. This is the node that is reached by starting at the root of the DST and crossing the transition labeled by each successive character in the string. The number of explicit nodes in the compressed tree is reduced to $2k-1$. To see why this is the case, suppose that the tree is being constructed by adding successively longer suffixes to the structure. Some prefix, possibly empty, of each new suffix, matches arcs that are already in the tree. At the point where the match cannot be continued, a new branch must be introduced, possibly turning a non-branching into a branching node. The remainder of the suffix is represented by newly introduced arcs and nodes among which there are no branches. It follows that no more branching nodes can be created than there are suffixes and, indeed, not even this many, because the first suffix put in the tree causes no branching.

A few words are in order on the question of how to read substrings out of the suffix tree represented in this particular way. It is generally necessary, when tracing a path from the root to a particular other node in the tree, to keep track of the length of the string traversed. This is simply a matter of adding the difference between each pair of text pointers encountered to a running count. The starting point of the string ending in a particular transition is the second of the numbers labeling that transition minus the current count.

The various occurrences of the substring ending at a particular node, whether branching, and therefore *explicit*, or non branching, and therefore *implicit*, can be enumerated straightforwardly and in a time that depends on the number of them that there are in the text and not on the actual length of the text. The method of doing this depends on the following observation: Just as the beginning of the substring corresponding to a sequence of arcs starting at the root can be determined by keeping track of the lengths of the segments covered by each transition, so the end points of the various occurrences of the substring represented by a particular node can be determined

by keeping track of the lengths of the segments from the node to the various terminal nodes reachable from it.

Consider, for example, the situation depicted in Fig. 1.3, representing a string of $q-p+s-r$ characters. There is an instance of this string beginning a position p in the text only if $q = r$. But, in any case, there is an instance of the string at position $s - (s - r) - (q - p = r - (q - p) = r - q + p$.

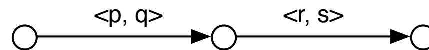

Figure 1.3: Computing substring locations

In general, to find all instances of a given substring, we find the locations of all the suffixes of which it is a prefix, and subtract the length of each of them from the length of the text. Consider the string "si" which traces out a path through the tree shown in Figure 1.2 over the edges $\langle 2, 3\rangle$ and $\langle 4, 5\rangle$. The length of the string is the sum of the lengths of the labels on these arcs, namely $3 - 2 + 5 - 4 = 2$. Since it ends at location 5 in the text, it begins at location $5 - 2 = 3$.

Suppose that the substring of interest is "s", reachable from the root over the single transition $\langle 2, 3\rangle$ and with paths to terminal nodes, including the substring itself, of lengths $(3-2)+(5-4)+(5-12) = 9$, $(3-2)+(5-4)+(12-8) = 4$, $(3-2)+(5-3)+(12-5) = 10$ and $(3-2)+(5-3)+(12-8) = 7$. This shows that it has four locations in the text at positions $12 - 9 = 3$, $12-4 = 8$, $12-10 = 2$ and $12-7 = 5$. Notice that, if a substring occurs in k locations, finding them in this way involves visiting no more than $2k - 1$ nodes, once the first occurrence has been located.

1.1 Building Suffix Trees—Ukkonen's Algorithm

1.1.1 Introduction

Ukkonen's algorithm has been characterized as an *on-line* algorithm because it considers progressively longer prefixes of the text, constructing a suffix tree for each of them on the basis of the one produced for the preceding prefix. One may quibble with this characterization on the grounds that, since no sentinel character is added until the rest of the text has been processed, the effect of adding the current character—namely that of ensuring that every suffix ends in a leaf node—will not have been realized. In general, therefore,

the intermediate trees will contain some suffixes ending at internal nodes that are not distinguished from other nodes. As we shall see, not only does this in no way detract from the effectiveness of the algorithm, it actually plays a crucial role in assuring its linear-time complexity.

The characters of the text are considered, one at a time, from left to right. When the character at position i is reached, a suffix tree for the first $i-1$ characters has been constructed, differing from a standard suffix tree only in that some suffixes may end at internal nodes. The task at this point is to is to modify the tree so that all the suffixes in it, except the one that consists of the empty string, now terminate in the new character. We were careful not to say that the new character is actually *added* to the end of each suffix because, since the number of suffixes in the tree is increased by one with each character that is processed, this could not result in a linear-time algorithm. It must clearly somehow be the case that many suffixes can show the effect of acquiring the new character without actually adding new arcs and nodes.

This is a achieved in two ways. One we have already mentioned. If a suffix is a prefix of a longer suffix, and continues to be so after the addition of the new character, then no new arcs are required in the tree to represent the addition. The character that one might expect to add is already in place. The other mechanism has to do with the representation of *leaf arcs*, that is, arcs that terminate in leaf nodes and whose labels are themselves suffixes of the text. Each arc is, as we have said, associated with a substring of the text. The label on the arc consists of a pair of integers giving the location in the text of the substring, and that of the character immediately following it. For leaf arcs, Ukkonen replaces the second of these integers by a special symbol, usually written as ∞. No information is lost as a result of this move because leaf nodes all end at the same place, namely the end of the text. But this on-line algorithm treats treats the text as though it were growing, and the tree being updated to reflect the addition of each new character. We must therefore think of ∞ as a variable that is incremented by 1, magically and without cost, after each new character is incorporated into the tree. These two devices will be principally responsible for ensuring that the tree is constructed in a time that is a linear function of the length of the text.

Since there is a one-to-one correspondence between the different substrings in the text and the nodes of the tree we can think of the strings as the names of the nodes, and we will refer to it as such. Notice that the subtree rooted in each node is itself a suffix tree containing just those suffixes of the text that are preceded by an instance of the name of the node. More

generally, if s_α and s_β are strings of characters, so that s_β names node n_β, and $s_\alpha s_\beta$ names node $n_{\alpha\beta}$, then the suffixes in the tree rooted at node $n_{\alpha\beta}$ are a subset of those in the tree rooted at n_β.

When adding a new character to a suffix tree, Ukkonen's algorithm visits nodes in order by decreasing lengths of the strings that name them and therefore by increasingly inclusive sets of suffixes in the trees rooted at then. If a is a character, α is a string, and the string $a\alpha$ names a node in the tree so-far constructed, then the algorithm will visit the node for α immediately after visiting this one. Most crucially, the suffixes in the tree rooted in the tree for $a\alpha$ will be a subset of those in the tree for the next node visited, namely the one named α.

1.1.2 BuildingDST's

We begin our description of Ukkonen's procedure by considering various states of affairs that might obtain in the midst of constructing some simple suffix DST's suffix trees using this algorithm. We will then consider the parallel situations in compact trees.

The following two properties of the algorithm follow directly from the observations just made:

1. **Property 1.** If the new character matches a leaf arc by virtue of the ∞ that is the second part of its label, then this must also have been true at all nodes visited previously with this character. Otherwise, there would have had to be an arc out of at least one of them that covered characters later in the text that the one currently being added.

2. **Property 2.** If there is an arc over a given character out of the node currently being visited, then there will be an arc over that character at all nodes visited subsequently.

Fig. 1.4 shows the DST that would exist after processing the characters "abcab". If this were a finite-state automaton, the black nodes would correspond to final states. These are the states in which a suffix ends and where new arcs might have to be added if the string were extended. Suppose, for example, that this were a stage in the construction of the DST for "abcabc". Adding "c" would give rise to the situation shown in Fig. 1.5.

In the diagrams, we have arranged the nodes that are at a given distance from the root in columns and we observe that just one of the nodes in each column is black. This is because a string of n characters contains $n + 1$

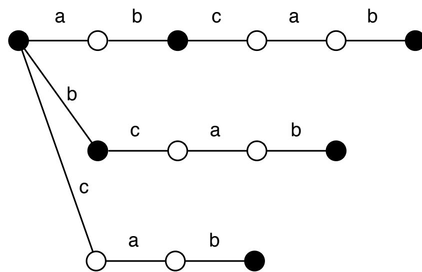

Figure 1.4: The Suffix DST for *abcab*

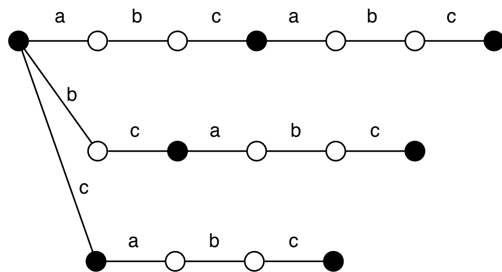

Figure 1.5: The Suffix DCT for *abcabc*

suffixes, one of which is the empty string represented by the root node, and each of which is of a different length.

While the tree in Fig. 1.4 contains 6 black nodes, only three new arcs and nodes were added in moving to the tree in Fig. 1.5. At the other three black nodes, there was already an arc labeled "c" so that it was sufficient to color the node at the beginning of the arc white, and color one at the end black. Notice also that, by virtue of property 2 of suffix trees, once it had been determined that no new arcs were required at the node reached from the root by following that arcs for "a" and "b", it was known that no nodes further to the left needed to be examined.

Suppose, now, that instead of adding "c" to the DST in Fig. 1.4, we add "d". This gives the DST in Fig. 1.6. This time, a substantive addition has to be made at each black node because no instances of "d" were added previously.

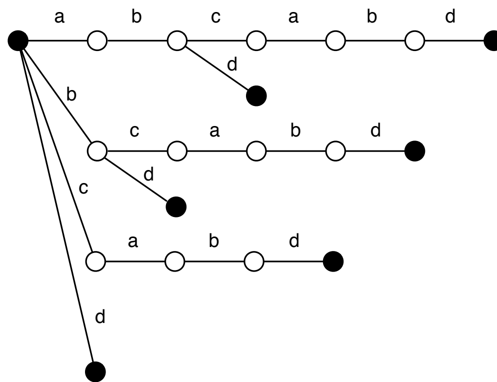

Figure 1.6: The Suffix DST for *abcabd*

Clearly, when a new character is added to a suffix DST, all the black nodes are affected, some by the addition of a new arc, and some only by causing some existing node to become black in the new DST. White nodes are not affected except inasmuch as some of them turn black.

1.1.3 Suffix Trees

When a suffix is added to a compact suffix tree, three cases have to be distinguished. Leaf arcs, even though they represent the end of a suffix, require no modification, because the end of the string is represented by a

special symbol ∞ which represents the end of the string, wherever that might be. When a new character is added, it is sufficient to reinterpret this symbol to incorporate it into these arcs. The second case is where a substantive change to the tree is required, namely the addition of a new arc and possibly a new node. The third case arises when the character is already in the s

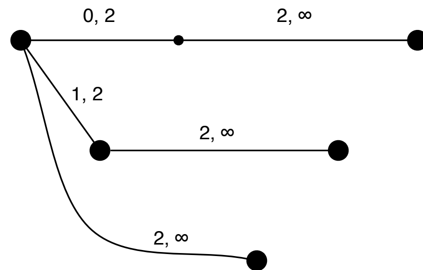

Figure 1.7: The Suffix Tree for *abcab*

The suffix tree for "abcab", now in its compacted form, is shown in 1.7. The configuration of black circles matches those in Fig. 1.4. The small one, however, is a virtual node with no explicit representation in the data structure. As we shall see, the program that constructs the tree can locate these when it needs to based on its internal state at that time. Adding a "c" to the end of each of the suffixes in this tree changes nothing that is an explicit part of the data structure. All that changes is information held by the program about where the small black circles are.

Consider now the suffix tree for the string "abbab" shown in 1.8. The two small black nodes on the top line correspond to the suffixes "a" and "ab". Adding the character "a" to this tree will illustrate all three of the ways in which suffixes are affected when new characters are introduced, as well as some crucial relations among these.

On the right of the diagram are three leaf nodes and therefore meet the first condition. The second component of their labels is the ∞ symbol, and the addition of the new character is therefore automatic. The next node to the left is a virtual node representing the suffix "ab". However, this instance of "ab" is followed by a "b" and the new character is "a", so a new node must be introduced, splitting the current arc and introducing an explicit node in place of the virtual one. The result of this operation can be seen in Fig. 1.9. The virtual node to the left of this, as well as the root node,

already have arcs labeled with "b", so no further action is required.

In this example, situations in which the introduction of a new character required no action because of the ∞ device occurred in the earliest steps which, because of property 1, is where they must occur. Situations where the character is already in place so the they require no action occur at the end, so that the process could be safely abandoned after the first of them is encountered. This is in accordance with property 2.

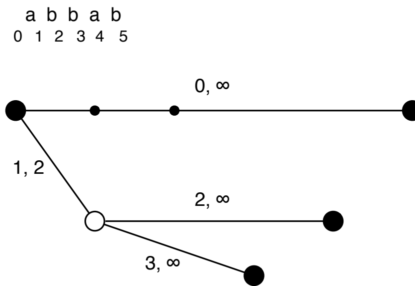

Figure 1.8: The Suffix Tree for *abbab*

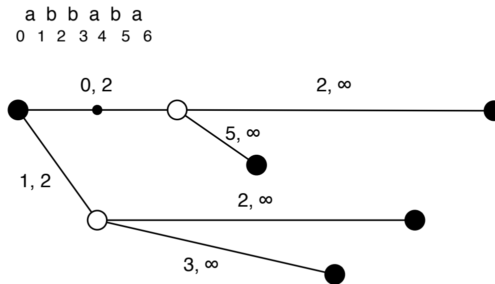

Figure 1.9: The Suffix Tree for *abbaba*

An arc is split, introducing an explicit, in place of a virtual, node when the character leaving the virtual nodes is different from the new one being introduced. After the arc has been split, the new character is provided for on a new leaf arc out of the new node. The next character to be accommodated in the tree will clearly match at a virtual node on this new arc by virtue of the ∞ marker that constitutes the second component of its label. We conclude that, when the next character is introduced, it will be automatically

accommodated by the ∞ devices on those arcs where the current character is so accommodated, and also on these newly added leaf arcs. Only after the new character has been found to match an existing character otherwise than by virtue of the ∞ does the possibility of further substantive changes to the tree arise. When treating the next character, we can therefore safely begin the examination of nodes starting at the explicit or virtual node beyond this first matching character.

We saw earlier that it is possible to stop the examination of nodes when the first one is encountered at which there is an explicit match. Now, we have seen that the first node that needs to be examined when the next character is treated can be determined at that point.

1.2 Reference Pairs

As we have remarked, it is in the nature of virtual nodes that they have no explicit representation in the data structures that our program will construct to represent suffix trees. The program, however, must have some way to refer to them, and this is usually done with what are referred to as *reference pairs*. A reference pair consists of an explicit node and a string. It is convenient, however, to refer to the string by a pair of integers, the first being a location in the text and the second being the length of the string so that, in all, the reference has three parts.

The pair refers to a node that can be found by starting at the given explicit node, and following the path through the tree whose characters spell out the string. In general, a given node may have a number of different reference pairs, and every node clearly has one referenced pair in which the explicit node is the root of the tree. The reason that there can be several pairs is that the path dictated by the string may pass through other explicit nodes. This is illustrated in Fig. 1.10. A virtual node at the position of the small black circle could be referred to as $\langle b, \langle 2, 1\rangle\rangle$ on the grounds that it is 1 character from the node b on a path beginning at position 2 in the text. On the other hand, it could be referred to as $\langle a, \langle 2, 2\rangle\rangle$ on the grounds that the 2 characters starting at position 2 in the text are "ba". Following the path labeled "b" from node a leads to node b, and the arc labeled "a" from their leads to the desired location. The former of these alternatives is the so-called *canonical* reference pair. It is the unique alternative that does not involve passing through another explicit node. It is the one in which the explicit node is the farthest to the right and the string is as short as possible. An important function in the program for building suffix trees will

be one that returns the canonical reference pair corresponding to one that may not be canonical. The canonical reference pair for an explicit node is clearly the node itself, a textual reference that is immaterial because it plays no role, and a length of 0.

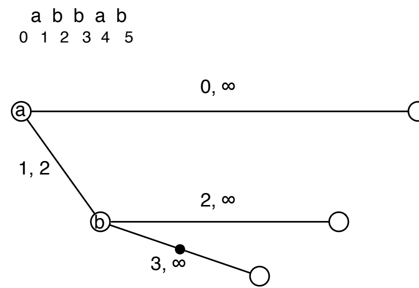

Figure 1.10: Reference Pairs

Ukkonen maeks crucial use of reference pairs and canonical reference pairs. The algorithm consists of two loops. The i-th iteration of the outer one adds the i-th character of the text to the end of each of the suffixes so far in the tree unless it is already there. The inner loop visits the ends of the current suffixes where substantive additions need to be made. The first of these locations is referred to as the *active point*. The sequence of such locations is known as the *boundary path*. The boundary path ends when a node is reached where the new character is already in place. This is know as the *end point*. As we have seen, the active point for the next iteration of the outer loop is found from the end point of the current loop by simply moving to the end of the arc where this first matching instance of the character was found. It remains to discuss how the moves from the end of a given suffix to the next shorter one are made.

If the name of the current node is $a\alpha$, where a is a character, and α a string, then we know that the name of the next node must be α and we know how to find a node with a given name, starting at the beginning of the tree. However, the time required to do this is, in the general case, proportional to the length of the text. To do it in this way would therefore negate the claim that the algorithm us linear. For explicit nodes, the solution to the problem is simply to maintain a set of explicit links from one node to another. In particular, node $a\alpha$ will incorporate a field, called the *suffix link* containing the location of node α. When a new node is created named α, it is invariably the case that the last node visited was one with a name of the form $a\alpha$, for

some a. We need therefore only keep track of this node from one iteration to the next in order to update it when needed.

In general, the boundary path is made up of virtual, as well as explicit, nodes. It is easy to see that, if a node on the path is explicit, then so must be the node following it and hence all the remaining nodes. If the node named $a\alpha$ is branching, and therefore explicit, then the node named α must clearly also be explicit by property 1. The reverse, however, is not the case. The successor on the boundary path of a node with reference pair $\langle x, s \rangle$, where x is named $a\alpha$ will be a node with reference pair $\langle y, s \rangle$ where y is named α. But what if x is the root, and is therefore named “”? If s is the empty string, then the end of the boundary path has been reached. Otherwise, the next node must presumably be one with the reference pair $\langle x, t \rangle$ where t is the string that results from removing the initial character from s. However, it is possible to avoid a special case here, and also at another place in the algorithm, by introducing a small modification to the tree.

We introduce a new node which we will refer to as the *base* node of the tree. It turns the structure into something which, strictly speaking, should no longer be called a tree, but no ill effects will flow from this. There will be an arc from the base node over every character in the alphabet to the root. Equivalently, there can be a single arc to the root whose label is capable of matching any possible character. The suffix link of the root points to this node.

No explanation of Ukkonen’s algorithm is complete that does not use the word “cacao” as a sample text. We use it now to illustrate the role of the base node. The first two steps of the process of constructing a suffix tree from this string are shown in Fig. 1.11. The base node is at the extreme left of each diagram. The node to the right is the one we shall continue to refer to as the *root* of the tree.

Diagram (i) shows the initial state, that is, the suffix tree for the empty string. The base node has an arc that will be followed over any character. The root has a suffix link that points back to the base node. The active point is $\langle$“$''$, 0, 0$\rangle$ where the second member of the triple is written as 0 only by convention. Its value is immaterial because the third member of the triple, which gives the length of the string, is 0.

Diagram(ii) shows the state of affairs after the first character, namely “c” has been added. As expected, an arc has been added at the root node. Standard procedure after making an addition, is to follow the suffix link and to consider whether the new character needs to be added there. When this question is asked of the base node, however, the answer is always “no” because the arc at the base node matches all characters. The base node is

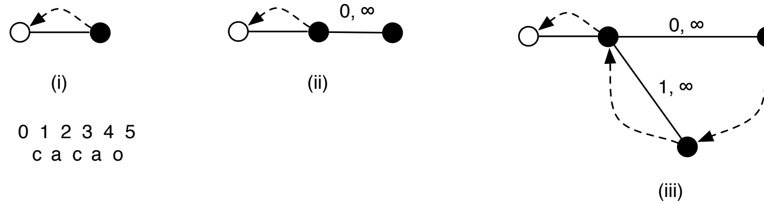

Figure 1.11: “cacao”—steps 1 and 2

therefore the end point and following the character that was found there takes us to the root, so this will be the active point for the next iteration of the outer loop.

The situation after the second character has been added is shown in diagram (iii). The sequence of events is just as in the case of the first character. The is no arc for the letter “a” at the root, so one is added. We then move to the base node, where the “a” matches the arc to the root. The active point is, once again, at the root.

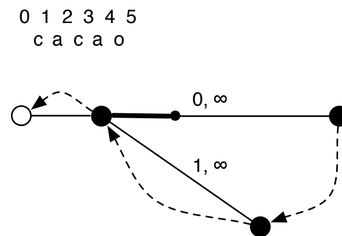

Figure 1.12: “cacao”—step 3

The second instance of the character “c” finds a match already at the root, and this is therefore the end point. The new active point will therefore be at the virtual node with the reference pair $\langle ``'' \langle c, 1\rangle\rangle$ which is shown with a small black circle in Fig. 1.12. The heavy black line leading to it shows the position of the last match of the most recent character.

The second instance of “a” also finds a match, ending at the virtual node $\langle ``'' \langle c, 2\rangle\rangle$, and the active point once again becomes the end point. The situation at this juncture is as depicted in Fig 1.13.

The next character, namely “o” does not match any arc at the new active point, so that the question arises of how to find the next point on the boundary path, given that the current one, $\langle$“”$\langle c, 2\rangle\rangle$, is virtual and the explicit node in its reference pair is “”. The existence of the base node and explicit suffix links makes the answer straightforward. The new node will be the one whose reference pair is derived from the current one by replacing the explicit node in it by the one at the end of that node's suffix link. In this case, that is the base node. This gives $\langle base\langle c, 2\rangle\rangle$ which, after canonization, becomes $\langle$“”$\langle c, 1\rangle\rangle$.

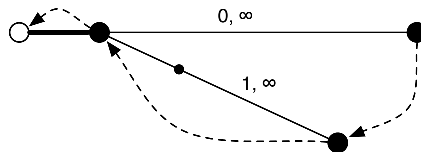

Figure 1.13: “cacao”—step 4

1.3 Segments

In the interest of minimizing the amount of space occupied by a suffix tree, without resorting to overly exotic devices, our implementation will depart from what the discussion up until this point suggests in some minor ways. The first concerns the representation of nodes and arcs, and the second, the labels that the arcs carry.

Since the object we are mainly concerned with is a rooted tree, we should expect there to be exactly one arc entering all nodes but the root. In fact the true root node is what we have been calling the *base* node, from which there are arcs over all possible characters to what we have been calling the *root* node. We will continue to use this terminology and will assign these nodes to special classes, to account for their special behavior. The remainder of the nodes in the structure do indeed have the property that exactly one arc enters them. This being the case, it is wasteful of space to treat arcs and edges as separate objects, with arcs containing pointers to nodes and nodes containing sets of arcs. It is more frugal to treat the arc and the node at which it terminates as one and the same object. To avoid confusion, we will refer to these objects as *segments*, but will still talk of nodes and arcs from

time to time in more informal contexts.

We will refer to the text from which a suffix tree is constructed as its *text*. Each arc, and therefore each segment (except those at the base node), has a *label* consisting of a pair of integers i and j, such that $0 \leq i < j < l$ where l is the length of the text. In other words, i and j are *indices* in the text, and i is strictly less than j. We will refer to these indices as the *left* and *right* parts of the label. These indices serve to associate the substring of the text beginning at character position i and ending at position $j - 1$ with the segment. Informally, we will sometimes refer to this character sequence as the label.

The objects that represent segments in our implementation will, in fact, have a place only for the left part of their labels. The right part will be obtained differently for internal and leaf segments. Since the substring covered by leaf segments all end at the end of the text, there is clearly no need for this to be represented explicitly on each of these segments. Since there is a strictly larger number of leaf segments in a tree than of any other kind, this represents a substantial savings. However, the details of how this should be implemented still leave some questions to be resolved.

If, as seems natural, there are to be classes to which the various kinds of segments will belong, then the ending point of the string associated with a leaf segment, which is simply the length of the string, could simply be a single variable in the class to which leaf segments belong. This has the apparent disadvantage, however, of making it impossible for more than one suffix tree to exist side by side because their leaf segments would presumably belong to the same class.

We solve this problem by calling on some of Ruby's less well known facilities. When a new Subtree object is created, a new class, which is a subclass of LeafSegment, is also created and stored in a class variable, called `@leaf_class`, of the new subtree. This class has a method called `text` which returns the text of this particular suffix tree. To create a new leaf segment, a suffix tree calls a method called `new_leaf` which returns a new instance of this special class. For the sake of parallelism, the classes of other segments in a suffix tree are also assigned to class variables, and new instances of them are created with calls to methods like `new_branch`.

It is also possible to dispense with the right part of the label on an internal segment. Observe that internal segments always arise as the result of splitting previously existing segments. The operation proceeds as shown in Fig. 1.16. After the split, the right part of label of the first segment must clearly be the same as the left part of the label of the second segment. Other segments may be added at the new node later but, if this first one is always

distinguishable in some way, then its left part can always be referred to in order to determine the right part of the first segment.

Figure 1.14: Segment Labels

The simplest way to store the successors of a given internal segment in such as way as to ensure that this first member of the set is distinguishable is to maintain a list of them with this distinguished member in the first position. The remaining members of the list might be stored in order by the characters that label them to that particular members could be found by binary search. It would suffice to check the first member first before initiating such a search in the remaining branches. Segments are either leaves or internal segments. The former belong to the `LeafSegment` class and the latter to the `InternalSegment` class. The `InternalSegment` class has two special subclasses, the `RootSegment` and `BaseSegment` classes. All of these are subclasses of the `Segment` class which has only the methods in program 1.

All segments have object variables `left` and `link` which contain the text position of the first character covered by the segment, and the segments suffix link. These are publically accessible (line **??**). They are set by the `initalize` method (lines **??** to **??**) to the argument supplied and `nil` respectively. The only other method that is shared by all segments is `to_s` (lines **??** to **??**) which returns a string giving the type of the segment, and the beginning and ending locations of the string it covers.

Program 1: class Segment

```
class Segment
  attr_accessor :link, :left                                #*a

  def initialize(left)                                      #*b
    @left = left
    @link = nil
  end                                                       #*c

  def ==(other)
```

```
10     self.type==other.type && @left==other.left
11   end
12
13   def to_s                                                   #*d
14     "<#type #left to #right>"
15   rescue
16     "<#type #left to ???>"
17   end                                                        #*e
18 end
```

The defining characteristic of an internal segment (see Program 2, lines **??** to **??**) is that they have branches. The `initialize` method provides and empty list of branches as the value of the `@branches` variable (line **??**) after calling the default `initialize` method (line **??**).

As we have alreay explained, the right-hand end of the sequence of characters covered by an internal segment is taken from the left-hand end of a distinguished member of its set of branches, namely the first branch that it acquired, and this occupies the first location on the list of branches. This is obtained by the `right` method (line **??**). It is assumed that this method will never be called before the first branch has been created.

The `find_index` method returns an integer giving the location in the array of branches from the segment whose label begins with the character at a given place in the text. It does this by calling the built-in method of the `Array` class, also called `find_index`(line **??**). This returns the first member of the array for which the block provided to it returns a **`true`** value. If there is no such member of the array, a `nil` value is returned. Notice that we have defined `find_index` in such a way that `@branches[i]` will not be evaluated if no appropriate member of the array is found.

The `each_index` method calls the block supplied to it for each branch from the segment, supplying it with the index of the branch(line **??**).

The `add_branch` method adds a new branch from the segment. If there is already a branch whose label begins with the same character, then that branch is replaced as the value of the method. If there is none such, then a `nil` value is returned, though this is immaterial for the algorithms presented here.

Program 2: class InternalSegment

```
1 class InternalSegment < Segment
2
3   attr_accessor :branches
4
```

```
5   def initialize(left)                                    #*a1
6     super                                                 #*a
7     @branches = []                                        #*b
8   end                                                     #*a2
9
10  def ==(other)
11    super && right == other.right
12  end
13
14  def right
15    @branches[0].left                                     #*c
16  end
17
18  def find_index(loc)
19    @branches.find_index|b| text[b.left]==text[loc]  #*d
20  end
21
22  def find_branch(loc)
23    (i = find_index(loc)) && @branches[i]                 #*e
24  end
25
26  def each_index
27    @branches.length.times|i| yield i                  #*f
28  end
29
30  def put(branch)
31    if i = find_index(branch.left)
32      @branches[i], old = branch, @branches[i]
33      return old
34    end
35    @branches.push(branch)
36    nil
37  end
38
39  # def copy
40  #   cp = self.class.new(left)
41  #   cp.link = link
42  #   cp.branches = Array.new(@branches)
43  #   cp
44  # end
```

```
45
46   def type
47     'IntSeg'
48   end
```

1.3.1 The SuffixTree object

Program 3: SuffixTree.init

```
 1 p .def initialize(text)
 2  d.def initialize(text, mon)
 3  d.  @mon = mon
 4 pd.  @text = text                                                     #*g
 5 pd.  [ @root_class = Class.new(RootSegment),
 6 pd.    @base_class = Class.new(BaseSegment),
 7 pd.    @branch_class = Class.new(InternalSegment),
 8 pd.    @leaf_class = Class.new(LeafSegment) ].each do |klass|         #*a
 9 pd.      klass.instance_exec(@text) do |txt|                          #*b
10 pd.        define_method(:text) do                                    #*c
11 pd.          txt
12 pd.        end                                                        #*d
13 pd.      end                                                          #*e
14 pd.    end                                                            #*f
15 pd.    @root=@root_class.new
16 pd.    @root.link = @base = @base_class.new(@root)
17 pd.    @base.link = @base
18 pd.end
```

Instances of the SuffixTree class are initialized by the method shown in 3. The instance variables `@root_class`, `@base_class`, `@branch_class`, and `@leaf_class` are assigned new classes which are subclasses of `RootSegment`, `BaseSegment`, `InternalSegment`, and `LeafSegment`. Calling the new method of `Class` with an argument that is the intended superclass has this effect. These four new classes are also collected into a list each member of which in turn becomes the value of the variable `klass` in a simple loop (lines **??** through **??**). All objects in Ruby, that is all members of the class `object`, have a method called `instance_exec` which executes the block supplied to it with the variable *self* bound to that object. Any arguments supplied to the method match, one for one, the arguments of the block. In the present case, the block is used to create a new method in the object, which is one of the four new classes. All that this method does is to return the text

that the current suffix-tree is attached to. This appears in the definition of the method as a constant. So, if `@text` is bound to "This is hardly worth making a suffix tree of", the new class that is bound to `@leaf_class` will have a method that is defined effectively as follows:

```
def text
 "This is hardly worth making a suffix tree of"
 end
```

We say that it is defined *effectively* in this way because the string is the same object as the string stored in the `SuffixTree` object as `@text`, and not a copy of it. The only other thing that happens in the initialization of the suffix tree is that the text of the tree is assigned to the `@text` variable (line **??**) of the tree.

A suffix tree is constructed from a text by calling the build method, with the string as an argument, on a newly created `SuffixTree` object. One may write:

```
st = SuffixTree.new.build(text)
```

because the `build` method returns the suffix tree.

Program 4: SuffixTree.build

```
1  p d.  def build
2   f .  def build_fiber
3    d.    show_text if @mon.monitoring('build')
4   fd.    k = 0
5  pfd.    current_segment, loc, offset = @root, 0, 0      #*d
6  pfd.    @text.length.times do |i|                       #*e
7  pfd.      old_branch = nil                              #*g
8    d.        # Insert character i at current_segment, offset characters down the
9    d.        # path beginning with the character at position loc in the text.
10   d.      if @mon.monitoring(['build'])
11   d.        puts " #i (#@text[i..i+30])"
12   d.        show_state(current_segment, loc, offset, '  Active point: ')
13   d.      end
14 pfd.      while true                                    #*h
15   d.        trace = "build#k"
16  fd.        k += 1
17   d.        show_state(current_segment, loc, offset, "  #k. Current segment: ") if @mon.m
18   d.        @mon.puts_if ['build', trace], "    old branch = #old_branch"
19 pfd.        current_segment, loc, offset =
20 pfd.          canonize(current_segment, loc, offset)
```

```
21 pfd.       if offset==0                                 #*k
22 pfd.         p = b = current_segment.find_branch(i)     #*l
23 pfd.       else
24 pfd.         b = current_segment.find_branch(loc)       #*m
25 pfd.         p = @text[b.left+offset]==@text[i]         #*n
26 pfd.       end
27 pfd.       if p                                         #*o
28                 # if offset==0 ? (b = current_segment.find_branch(i))
29                 #              : @text[(b = current_segment.find_branch(loc)).l
30 pfd.         loc = b.left                               #*o1
31   d.         show_state(current_segment, loc, offset, '    Branch exists: ') if
32 pfd.         offset += 1                                #*o2
33 pfd.         if old_branch                              #*o3
34 pfd.           old_branch.link = current_segment        #*o4
35   d.           @mon.puts_if ['build', trace], "    #old_branch --> #current_seg
36 pfd.         end
37 pfd.         break                                      #*j
38 pfd.       elsif offset>0                               #*p
39
40 pfd.         nb = branch = new_branch(b.left)               #*p1
41 pfd.         oldbr = current_segment.put(branch)        #*p2
42   d.          show_state(oldbr, loc, offset, '    Split: ') if @mon.monitoring(
43 pfd.         oldbr.left += offset                       #*p3
44 pfd.         branch.put(oldbr)                          #*p4
45 pfd.         branch.put(new_leaf(i))
46
47
48 pfd.       else                                         #*q
49 pfd.         (branch = current_segment).put(nb = new_leaf(i))  #*q1
50   d.         @mon.puts_if ['build', trace], "    Add branch #b at #current_segm
51 pfd.       end                                          #*r
52 pfd.       if old_branch                                #*r1
53 pfd.         old_branch.link = branch                   #*r2
54   d.         @mon.puts_if ['build', trace], "    #old_branch --> #branch" if @m
55 pfd.       end
56 pfd.       old_branch = branch
57 pfd.
58   d.       show(branches: [current_segment, nb], reset: true) if @mon.monitorin
59  f .       Fiber.yield(k)
60 pfd.       nb = nil
```

```
61 pfd.        current_segment = current_segment.link      #*s
62 pfd.      end # while true                              #*i
63 pfd.    end                                             #*f
64  f .    Fiber.yield nil
65 pfd.    return self                                     #*t
66 pfd.  end
67
68  f .  def build
69  f .    builder = Fiber.newbuild_fiber
70  f .    while val = builder.resume
71  f .      puts val
72  f .    end
73  f .  end
```

Every suffix tree contains a root and a base node, and these are created first (lines **??** and **??**) and the suffix link of the root is made to point to the base node(line **??**).

The current node, at which the next addition to the tree will be added if necessary, may be a virtual node. In the general case, it is therefore referred to by a node, and a string which is a substring of the text. The string can be represented by a location in the text where the string occursand its length is another integer. In our `build` method, these three values will be held in the following three variables:

`current_node`	an explicit node
`loc`	location of string in text
`offset`	string length

Table 1.1: Variables identifying the current segmenttab:a

If the node is explicit, then the third member of the triple will have the value 0. The current node is initially the root; the length is 0 and the branch is therefore irrelevant and is set to 0 by convention. These assignments are made in line **??** of the `build` method.

The main part of the building operation is contained in the loop in lines **??** to **??** which, on each iteration, adds the `i`-th character, to the partial suffixes that are already in the tree. This typically involves introducing a new explicit node in the middle of an existing edge—the *splitting* operation that we have already referred to—or introducing a new edge at an existing node. In either case, if the node in question is not the first one encountered

in the current iteration, it will generally be necessary to provide the node operated on in previous iteration with a suffix link that points to the current node. The identity of this node is therefore preserved as the value of the variable `old branch`. In the first iteration, there is no previous node, and the variable is therefore given the conventional value `nil` (line **??**).

The inner loop, extending from line **??** to **??**, follows the boundary path from the active point, established in line **??** to the end point. The loop is, in fact, terminated by the `break` instruction in line **??**.

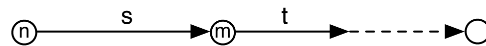

Figure 1.15: Canonization

The first move made in the inner loop is to *canonize* the reference to the current node. Recall that, in general, a reference to a node consists of an explicit node and a string. The virtual node is found by following the path through the tree that starts at the explicit node. However, as a result of splitting operations, some prefix of the string may lead to another explicit node. Suppose that reference to the virtual node is $\langle n, st\rangle$, where n is an explicit node, and st is the string that results from concatenating s and t. Suppose, furthermore, that the pair $\langle n, s\rangle$ identifies the explicit node m (see 1.15). In this case, the pair $\langle m, t\rangle$ identifies the same virtual node as the original pair $\langle n, st\rangle$. If this operation cannot be repeted on the new pair because no explicit node is encountered on the path from node m described by the string t, then $\langle m, t\rangle$ is canonical. In general, the canonical pair is found by repeating this operation until no further repetition is possible. The canonical pair is thus that member of the set of equivalent pairs with the shortest string. It is also the member in which the explicit node is furthest from the root of the tree.

Program 5: SuffixTree.canonize

```
1 def canonize(current_segment, loc, offset)
2   if offset>0                                          #*a
3     branch = current_segment.find_branch(loc)          #*c
4     span = branch.right-branch.left                    #*d
5     while offset>=span                                 #*e
6       loc += span                                      #*g
```

```
 7      offset -= span                              #*h
 8      current_segment = branch                    #*i
 9      branch = current_segment.find_branch(loc)   #*j
10      span = branch.right-branch.left             #*k
11    end                                           #*f
12  end
13  return current_segment, loc, offset             #*b
14 end
```

Canonization of node references is carried out by the SuffixTree's `canonize` method (Program 5). This is supplied with the three values needed to identify a node, namely an explicit node, `current_segment`, the location of a string in the text, `loc`, and the length of that string, `offset`. If `offset` is 0, the reference is to an explicit node and the string is empty. In this case, the value of `loc` is irrelevant, and the method returns the original triple of values (lines **??** and **??**). Otherwise the branch from the given node labeled with the first character of the string is located and asssigned to the variable `branch`(line **??**). If the triple is well formed, this branch must exist. The length of this branch—the difference between the locations of its left and right ends–is computed and assigned to the variable `span` (line **??**).

Canonization is an empty operation if the length of the string supplied to it (`offset`) is less than the length of the corresponding branch out of the given explicit node (`span`). The loop in lines **??** to **??** continually replaces the current explicit node by one further from the root, and the string by a suffix of the current string, until this condition is established. The location of the string in the text is moved forward by the length of the branch (`span`) in line **??**; the length of the string (őffset) is decreased by the same amount in line **??**; and the curent branch becomes the new node in line **??**. A new branch form the new node is located in line **??** together with its length in line **??**.

In line **??** of program 4, we may now take it that `current_segment`, `loc`, and `offset` constitute a canonical reference to an explicit or virtual node. Which of these is the case determines the details of the next operation, which determines whether the current suffix needs to be added at this node, or whether it is there already. If the value of `offset` is 0, indicating that the current node is explicit, then it is sufficient to determine whether there is already a branch at that node lobeled with the current character, that is, the character at text location `loc`. This is determined in line **??** and the branch itself is made the value of the variable `b`. The value of the variable `p` will be treated as Boolean. The value will be `nil` if no branch was found,

and this will be interpreted as **false**. Otherwise, the value will actually be the branch itself, and this will be interpreted as **true**.

If the value of `offset` is not 0, then a virtual node has been located, in which case `offset` determines the distance down the branch that must be examined to determine if the required character is already in place. The branch to be examined is determined by the value of `loc` , and the comparison with the character at the appropriate offset is made in line **??**. Once again, the varirable `p` is set in such a manner that it can be interpreted as a Boolean value showing whether the character was found.

We now come to the main decision that must be made concerning the current character of the text at the current node in the tree. There are three possibilities. The first is that the character is already in place, in which case the present iteration of the inner loop ends and a new active point is established for the next iteration of the outer loop. The second possibility is that a virtual node needs to be turned into an explicit node at which a new leaf node will be introduced. In other words, the current arc needs to be split The third possibility is that the current node is explicit and a new branch needs to be added from it.

The conditions required for the first of these possibilities obtain if the variable `p` has a value that can be interpreted as **true**, that is, any value other than `nil` (line **??**). In this situation, the current iteration of the inner loop ends, and new values of `loc` and `offset` are established in preparation for the next cycle of the outer loop. The function of the `loc` variable is to determine the branch out of the current node that will be followed to find the next node, explicit of virtual. As we have seen, when a character has been found to be in the tree already, the new active point is at the termination of the arc where it was found. We make take that node to be the one that is the value of the variable `b`(line **??**), even though this may changed later by canonicalization. Since the new active point is at the destination of this arc, the value of `offset` is incremented by 1 (line **??**). Finally, if there is an old branch, its suffix link is updated to point to the current branch (lines **??** and **??**). In cases where the inner loop continues, this is done in lines **??** and **??**, but that occurs at a point later in the loop that will not be reached in this case because of the `break` in line **??**.

If the character being added to the tree was not found, and the variable `p` has the value `nil`, and if `offset` has a value greater than 0, then the current node, which is virtual, must be made explicit so that it can be provided with a second branch. In other words, the current arc must be *split*. This is done in lines **??** to **??**.

An example that illustrates the situation is shown in Fig. 1.16. The

diagram labeled (i), on the left shows the sitution before the split, and the one on the right, labled (ii) shows the situation afterwards. The first four letters of the word *mississippi* have been put into the tree. The last of these—the second s in the string—was found to be already in the tree, so that the active point is now at the virtual node $\langle x, 2, 1\rangle$. The character in the next location on that branch is s, but the current character in the text is i. The variable `p` has the value **false**, the value of `offset` is 1, and we are at line **??**. A new internal segment is first created and assigned to the varialble `branch`. This then replaces the existing segment on which the virtual node is located out of the current segment (**??**).

The arc labeled $\langle 2, \infty\rangle$ is deleted from node **x** (line **??**) and becomes the value of the variable `b`. A new internal segment is created, whose left end is the same as that of the arc just deleted, namely 2, and the left end of this is increased by the value of `offset`, so that it becomes 3. The first of these becomes the arc from node x to y and the other becomes the second part of the split arc, labeled $\langle 3, \infty\rangle$. A new leaf segment is created for the new character (line **??**) and added to the current segment. Recall that the internal segments `put` method returns any edge that the new one replaces. There will always be such a branch in this case, and it may be either internal or a leaf segment. In either case, the character in the text that begins its label is moved forward by the value of `offset` and it is installed as a branch out of the new segment, `branch` (lines **??** and **??**).

The way the code is written, it is important that the original branch, which is split in this operation should become the second of the split pair because this makes unnecessary to distinguish the cases in which it is an internal or a leaf segment. Whichever class it belonged to before the split, it retains afterwards.

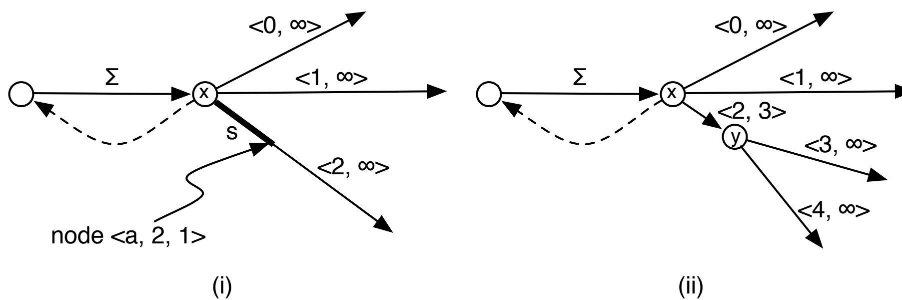

Figure 1.16: Splitting

The third, and last, case to consider is the one in which the varaible `p` has the value 0, indicating that current character is not, but `offset` also has the value 0, indicating that we are at an explicit node to which it is sufficient to add a new arc, and this is done in line **??**.

If the variable `old_branch` has a value other than `nil`, then the current iteration of the inner loop is not the first. In this case, a suffix link must generally be created from the node in the preceding iteration to the current node, which is done in line **??**. We say that this must *generally* be done because, if the current node is virtual, then it will already have been done in an earlier iteration, but there is nothing to be gained be distinguishing this case.

Finally, we move to a new iteration of the inner loop by following the suffix link from the current to a new node (line **??**). The method returns the subtree object itself (line **??**) so that it will be possible to cause a suffix tree to be built in the same expression in which the SuffixTree object is created.

node	level	terms	len	type	left	right
0	0		0	pre	-1	0
1	1		12	leaf	0	12
2	1		1	pre	1	2
3	2		4	pre	2	5
4	3		11	leaf	5	12
5	3		8	leaf	8	12
3	2	2	4	post	2	5
6	2		5	leaf	8	12
7	2		2	leaf	11	12
2	1	4	1	post	1	2
8	1		1	pre	2	3
9	2		3	pre	3	5
10	3		10	leaf	5	12
11	3		7	leaf	8	12
9	2	2	3	post	3	5
12	2		2	pre	4	5
13	3		9	leaf	5	12
14	3		6	leaf	8	12
12	2	2	2	post	4	5
8	1	4	1	post	2	3
15	1		1	pre	8	9
16	2		4	leaf	9	12
17	2		3	leaf	10	12
15	1	2	1	post	8	9
18	1		1	leaf	11	12
0	0	12	0	post	-1	0

Table 1.2: Variables identifying the current segmenttab:a

Chapter 2

Minimal Suffix Trees

One of the many ways of looking at a suffix tree is as deterministic finite-state machine. If only the leaf nodes are final, then it accepts just the suffixes of the text from which it was built. If all states, both implicit and explicit, are final, then it accepts all substrings of that text. Generally speaking, it will be possible, at least in principal to construct an automaton with a smaller number of states, that will accept these same strings. In fact, it is well known that every finite-state machine belongs to a family of equivalent automata exactly one of whose members contains less states than any of the others. However, the states and transitions that make up this minimal member of the family cannot be expected to have the shape of a tree. There are algorithms that construct the minimal member of the family to which an arbitrary finite-state machine belongs, and they proceed by conflating pairs of equivalent states until no such pairs remain. The prospect of finding a minimal equivalent for an arbitrary suffix tree is attractive because this is a data structure which tends, in interesting cases, to be large, and the prospect of doing it have therefore been one of the two two main approaches to the challenge of reducing the space that they require.

The other approach focuses on finding ever more ingeneous and exotic schemes for representing suffix trees in computer memory. This has reached the point of attempting to demonstate that the number of bits required for a particular scheme approaches the minimum possible from a purely informational-theoretic point of view. Needless to say, the single-minded pusuit of this goal is not always accompanied by comparable improvements in the algorithms that exploit the trees or the perspicacity of the scheme as a whole.

2.1 References

Bibliography

[1] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[2] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[3] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[4] P. Weiner. Linear pattern matching algorithm. *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.