Lecture 2: November 5, 2003

Lecturer: Ron Shamir *Scribe: Igor Ulitsky and Uri Ron*

2.1 Exact String Matching

2.1.1 Introduction

This lecture introduces the problem of locating a given pattern in a string and the standard solutions to that problem. First, the problem is presented along with the key definitions and the biological motivation. Then a classical algorithm is mentioned. Most of the discussion revolves around the *Suffix Tree* data structure and how it can be used to solve the exact matching problem. For the *Suffix Tree* construction, *Ukkonen's* linear-time algorithm is presented and explained along with a time and space complexity analysis. At the end, various biological applications of the suffix trees are presented.

2.1.2 Basic String Definitions

- **Definition** *String* S - any ordered sequence of characters written contiguously from left to right.

- **Definition** *Substring of* S - $S[i..j]$ is the string composed of characters i through j $i \leq j$ of S.

  Example For a string $S =$ "*ComputationalGenomics*" the string "*omics*" is a substring, as well as "*utationalGeno*", whereas "*tatiomics*" is not (gaps and jumps aren't allowed).

- **Definition** *Subsequence of* S - a string composed of characters $i_1 < i_2 < i_3 < ... < i_k$ of S. Example: For $S =$ "*ComputationalGenomics*" , both "*omics*" and "*tatiomics*" are considered subsequences.

Remarks:

2.1.3 Introduction

- The empty string is considered both a substring and a subsequence of any string

- Those are Computer Science terms with will be used for this discussion. In Biology terminology the subsequence is contiguous.

centering

2.1.4 The Exact String/Pattern Matching Problem

Problem 2.1 (String/Pattern Matching)
INPUT: A text S and a pattern p_i
QUESTION: Is p_i a substring of S

Possible solutions or the problem include of course the naive string-matching algorithm which checks the match of all the substrings of S with p_i. The complexity in this case would be $O(|S||p_i|)$.

Besides the naive algorithm, the *Knuth-Moriss-Pratt 1977 (KMP)* string matching algorithm provides a linear-time solution in $O(|S| + |pi|)$ for matching a text with a single pattern. The problem with KMP algorithm arises with the need for matching multiple patterns with a given text. In such case, $O(n|S| + \sum_{i=1}^{n} |pi|)$ time is needed in order to match n patterns $p_1, ..., p_n$ to the same string S.

A solution that solves the problem for n queries with a much better complexity is the *Suffix Tree* solution, which can execute n pattern queries on S in $O(|S| + \sum_{i=1}^{n} |p_i|)$.

2.1.5 The Biological Motivation for a Multiple-Query Solution

The biological motivation for performing multiple queries on a given text can be seen in queries made on entire genome sequences. For instance, searching for n patterns of 100-1000bp(average size of a gene) in the human genome sized $3^9 bp$ can produce a very costly process. This problem involves a rarely changing text (the genome database is updated approximately monthly) and many queries of various patterns to be matched in the genome (for example, cDNA sequences).

Actually, the sequences sought for are not exact-matching strings, due to the certain variance found in DNA sequences. A fine example for **exact** string-matching comes from the field of DNA-chips.

Definition *DNA chip* is a microchip that holds single-strand DNA probes and can recognize complementry single-stranded DNA or mRNA from samples being tested.

Every probe in a DNA chip can contain a sequence of approximately 25bp representing a single gene. The sequence of the probe needs to be as unique as possible to the gene that it

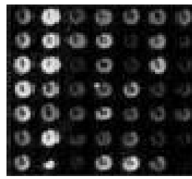

Figure 2.1: A DNA chip

represents. The probe serves as a fingerprint of the gene and therefore it shouldn't be found anywhere else in the genome.

The human genome contains approximately $30K$ genes, and let's consider length of a gene as 1000bp. In order to find enough unique 20-80bp fingerprints, all the $\approx 25bp$-long substrings of the gene must be sought in all the coding sequences of the genome (in order to eventually select the best candidates). In order to solve the problem for n genes we'll require to look for a $\sum_{i=1}^{n} |pi| = 30K * 1000 * 25$ in a text sized $|S| = 30K * 1000$. Using KMP algorithm, such a complex query $(n = 30,000 * 1000)$ will take approximately $30K * 1000 * 30K * 1000 + 30K * 1000 * 25$ operations, which sums up to about $9 * 10^{15}$ operations. The Suffix Trees algorithm will need only $30K * 1000 + 30K * 1000 * 25 = 7.8 * 10^{8}$ operations.

2.2 Suffix Trees

Both the *KMP* and the *Suffix Tree* algorithms perform **preprocessing** of the input before the search. The *KMP* algorithm preprocesses the patterns (by calculating the Π function), while the suffix tree algorithm preprocesses the given text by building a *suffix tree* for it (in $O(|S|)$ time). After the tree has been constructed each pattern is sought in $O(|p_i|)$ time.

2.2.1 Basic Definitions

Definition *Prefix of a string S* - substring of S beginning at the first position of S

Definition *Suffix of a string S* - substring of S ending at the last position of S

Example For a text $S = AACTAG$
Possible **prefixes** : $AACTAG, AACTA, AACT, AAC, AA, A$

Possible **suffixes** : $AACTAG, ACTAG, CTAG, TAG, AG, G$

Lemma 2.1 *A pattern P is a substring of a string S if and only if P is a prefix of some suffix of S*

2.2.2 Trie Structure

Definition A *trie* is a tree representing a set of strings by character labels on its edges. Every path from root to leaf represents a different string - thus every string corresponds to a leaf of the trie. Besides, no two edges outgoing from the same node are labeled the same.

Example A simple trie is shown in Figure 2.2.

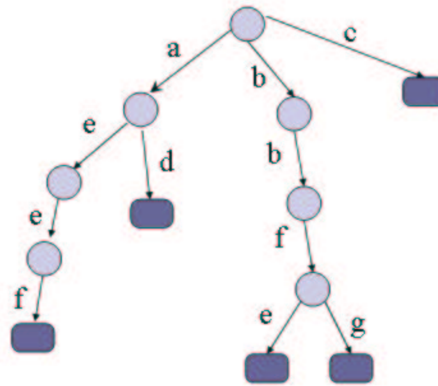

Figure 2.2: This is a *trie* representing the strings : *aeef, ad, bbfe, bbfg, c*

Definition A *Compressed Trie* is a trie with strings instead of characters on the edges. A compressed trie can be obtained from an ordinary trie by removing the unary nodes, and merging the appropriate edges.

Example Figure 2.3 presents a compressed trie corresponding to the ordinary trie shown in Figure 2.2.

2.2.3 Constructing a Suffix Tree Structure

Definition A suffix tree is a compressed trie representing all the suffixes of a string S. Thus the suffix tree is :

1. A rooted tree T with $|S| = m$ leaves numbered from 1 to m. As explained above, every leaf represents a suffix of S. The leaf number represents the starting point of the suffix in the original string S.

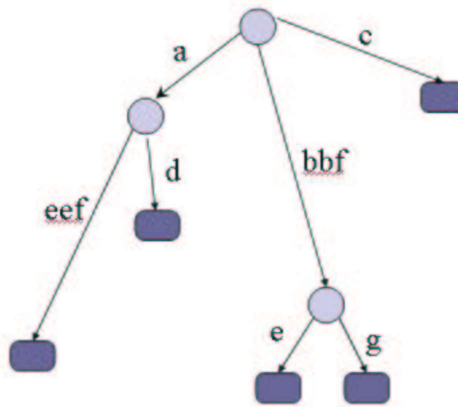

Figure 2.3: A compressed trie

2. Each internal node of T, excluding the root, has ≥ 2 children (as in any compressed trie).

3. Each edge of T is labeled with a nonempty substring of S.

4. All edges out of a node must have edge-labels starting with different characters (as in any trie).

Given the above definitions, for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i spells out the suffix of S, namely $S[i..m]$, that starts at position i.

Example A simple suffix tree for a string "$xabxac$" is shown in Figure 2.4:

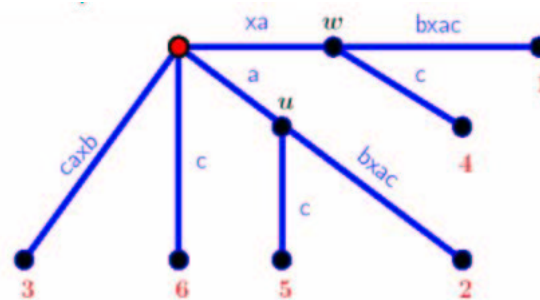

Figure 2.4: A suffix tree of the string "xabxac"

2.2.4 Guaranteeing Existence of a Suffix Tree

The definition above does not guarantee that a suffix tree for any string S actually exists. A problem can arise if one *suffix* of S matches a *prefix* of another suffix of S - since the path for the first suffix would not end with a leaf. Actually it is enough for the last character of the string to appear somewhere else in the string, in order to make this problem appear.

Example For $S =$ "*xabxa*" and $S_1 =$ "*xabxa*", $S_4 =$ "*xa*" - the suffix "*xa*" does not end with a leave - thus this tree is not a proper suffix tree. This is shown in Figure 2.5.

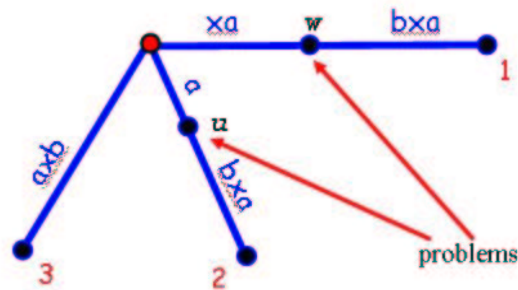

Figure 2.5: The problem of building a suffix tree for the string "*xabxa*".

To avoid this problem we must add a unique character to the end of S, a character that does not exist in the alphabet of S. The *$* sign will be used as such a string termination character in further discussion. This will eliminate the possibility of a suffix of S being a prefix of another suffix.

Example The previous example with *$* terminating character added is shown in Figure 2.6 .

2.2.5 Usage of a Suffix Tree for Pattern Matching

In order to match a substring of a text S with a pattern p we descend down a suffix tree. We begin at the root, and move down one level at a time, matching subsequent characters from the pattern with the labels on the edges. If the end of the pattern has been reached **before** reaching a leaf in the tree - the pattern matches a prefix of some suffix of S - hence it matches a *substring* of S. This can be proved easily using lemma 2.1

Since the alphabet we're using is finite, the number of comparisons required at each node of the tree is limited. Thus the complexity of locating a pattern in a constructed Suffix Tree is linear to the size of the pattern - $O(|p_i|)$. This brings us to the main complexity challenge of the algorithm - the building of the suffix tree.

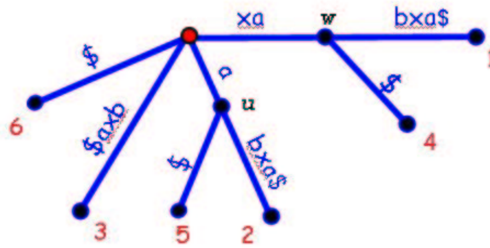

Figure 2.6: A suffix tree of the string "$xabaxc$$".

2.2.6 The naive algorithm to build a suffix tree

1. Create a single-edge tree for a suffix $S[1..m]\$$ (the entire string) - the leaf will be labeled as 1.
2. Successively enter suffixes $S[i..m]\$$ into the growing tree for $i = 2..m$ and label the leaf created at step i with i (since it represents the suffix $S[i..m]$).

Example The algorithm is illustrated using building of suffix tree for a string "$S = abab$" in Figure 2.7

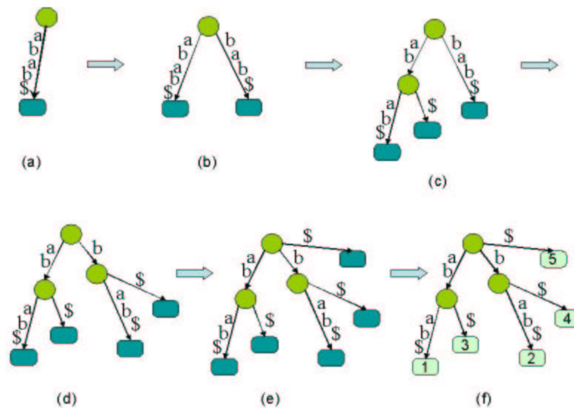

Figure 2.7: A construction of a suffix tree for "$S = abab$": (a) Putting the longest suffix $abab$$; (b) Adding the suffix $bab$$; (c) Adding the suffix $ab$$; (d) Adding the suffix $b$$; (e) Adding the suffix $; (f) Labeling each leaf with the starting point of the corresponding suffix .

Complexity Analysis of the Naive Algorithm

For each suffix $S[i..m]$ the building time is proportional to its length. As there are $m+1$ suffixes, and the length of each suffix does not exceed m, the overall complexity can take $O(m^2)$. In this analysis, as above, we assume the alphabet to be finite.
This complexity does not provide a suffice solution for complex problems presented above. This leads us to the main part of our discussion - a linear time suffix tree construction algorithm.

2.3 Ukkonen's Linear-time Suffix Tree Algorithm

2.3.1 Historical Background

The most significant works in the field of construction of suffix trees in linear time are :

- **Weiner's algorithm** [FOCS, 1973] - referred by Knuth as "The algorithm of 1973", solves the problem in a linear time, but uses too much space.
- **McCreight's algorithm** [JACM, 1976] - solves the problem in linear time but uses quadratic space, although it is much more readable.
- **Ukkonen's algorithm** [Algorithmica, 1995] - taking much less space than the above, and still solving the problem in linear time. Further discussion will focus on Ukkonen's Algorithm.

2.3.2 Implicit Suffix Trees

Ukkonen's algorithm is based on the construction of a series of *implicit suffix trees* , only the last of which is converted into an true suffix tree of the given string.

Definition An implicit suffix tree for string S is obtained from a true suffix tree for $S\$$ by the following 3 stages :

1. Removing every copy of the terminal symbol $ from the edge labels of the tree.
2. Removing any edge that contains no label after the removal.
3. Removing every node that remained with only one child.

A similar process can be performed on a suffix tree of a prefix $S[1..i]\$$ creating an implicit suffix tree for $S[1..i]$.

Definition An implicit tree for $S[1..I]$ will be denoted in further discussion as I_i .

Example Figure 2.8 shows the construction of an implicit suffix tree for the string $S =$ "$xabxa\$$" which contains the suffixes "$xabxa\$$", "$abxa\$$", "$bxa\$$", "$xa\$$", "$a\$$", $\$$"

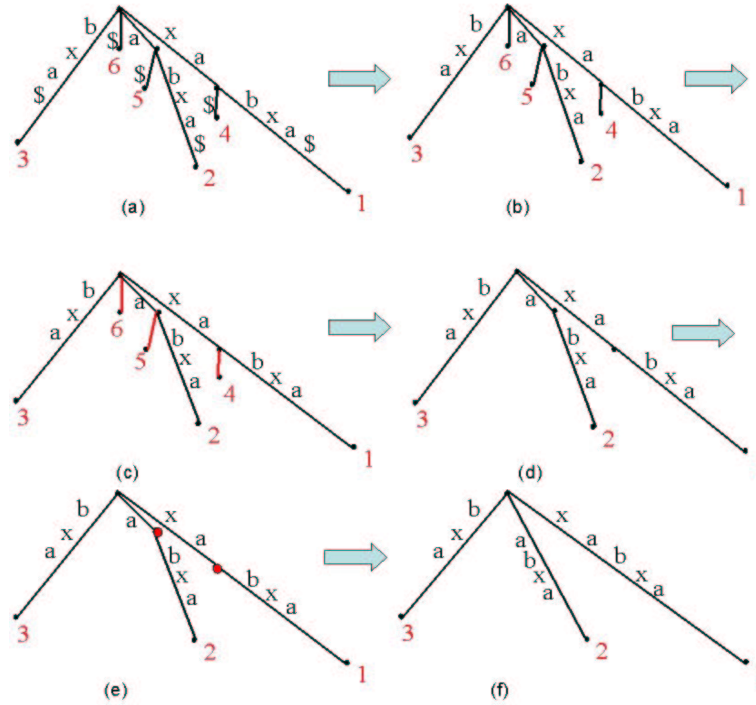

Figure 2.8: A construction of the implicit suffix tree for "$S = xabxa$": (a) The suffix tree for $S =$ "$xabxa\$$"; (b) Removing the \$; (c) Removing unlabeled edges; (d) After the removal of unlabeled edges; (e) Removing internal nodes with only one child; (f) The final implicit tree

Even though an implicit suffix tree does not contain a separate leaf for each suffix, it still includes all the suffixes of S, each suffix spelled out by the characters on some path from the root of the tree. However, if the path does not end with a leaf, it contains no marker indicating that the path has ended. Therefore, implicit suffix trees, on their own, are somewhat less informative than true suffix trees. They will be used just as a tool in the algorithm in order to obtain a true suffix tree for S at its end.

2.3.3 Ukkonen's Algorithm (UA)

The algorithm constructs an implicit suffix tree I_i for each prefix $S[1..i]$ of S, starting from I_1 and incrementing i by 1, until I_m is constructed. We will first present an intuitive implementation whose time complexity will be $O(m^3)$ and then optimize it to obtain an $O(m)$ time-bound algorithm.

High Level Description:

UA is divided into m *phases* . It begins by construction of I_1, which is a single edge labeled by $S(1)$. Afterwards, in phase $i+1$ tree I_{i+1} is constructed from I_i. In each phase $i+1$, $i+1$ *extensions* are made, one for each suffix of $S[1..i+1]$. The rules for these extensions will be described below. In extension j of phase $i+1$, the end of the path from the root labeled with the substring $S[j..i]$ is found and extended by adding character $S(i+1)$ to its end according to the extension rules.
The last extension (extension $i+1$ of phase $i+1$) extends the empty suffix of $S[1..i]$, inserting the single character string $S(i+1)$ into the tree (unless it is already there).

UA high-level algorithm:

1. **Construct** I_1

2. **For** $i = 1..m-1$ **do**
   begin {phase $i+1$}
       For $j = 1..i+1$ **do**
       begin {extension j}
         Find the end of the path from the root whose label is $S[j..i]$ in I_i and
         extend it with character $S(i+1)$.
       end {extension j}
   end {phase $i+1$}

3. **Convert** I_m **into a true suffix tree** S

Example For $S =$ "$xabxacd\$$" :

1. In the first stage (phase $i+1=1$) we construct I_1 - which contains only "x"

2. Phase $i+1=2$ - we construct I_2 by extending "x" to "xa" and adding "a"

3. Phase $i+1=3$ - we construct I_3 by extending "xa" to "xab" ,"a" to "ab" and adding "b"

4. etc.

2.3.4 Extension Rules

This section describes how to perform every *extension* stage described in the above high-level algorithm. The goal of extension j is to make sure that the suffix $S[j..i+1]$ is in the tree. This is accomplished by the following three rules :

Rule 1: If $S[j..i]$ ends at a leaf:
Add character $S(i+1)$ to the end of the label on the leaf's edge.

Rule 2: If $S[j..i]$ does not end at a leaf, and the following character after $S(i)$ is not $S(i+1)$: Split a new leaf edge for character $S(i+1)$. If $S[j..i]$ ends in the middle of an edge, create a new *internal node*. The leaf at the end of the new edge is given the number j.

Rule 3: Otherwise, $S[j..i+1]$ is already in the tree, so no update is necessary, since in an implicit suffix tree, there's no need for explicit mark at the end of each suffix.

Example Figure 2.9 shows the construction of an implicit suffix tree for the string $S =$ ”*xabxa$*” which contains the suffixes ”*xabxa$*”, ”*abxa$*”, ”*bxa$*”, ”*xa$*”, ”*a$*”, $”

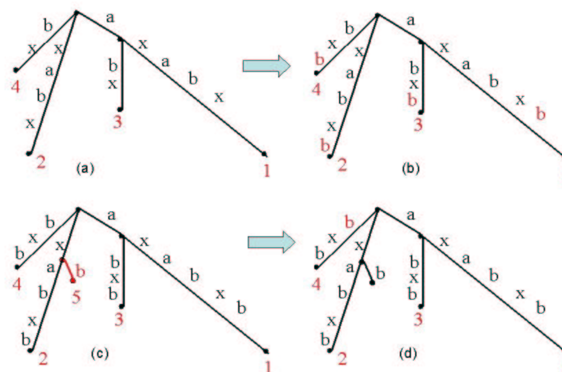

Figure 2.9: Building an implicit tree for *axabxb* from tree for *axabx*: (a) The original tree for *axabx*; (b) Rule 1 applies while adding suffixes *axabxb*, *xabxb*, *abxb* and *bxb* - at a leaf node; (c) Rule 2 applies when adding suffix *xb* - adding a leaf edge and an interior node; (d) Rule 3 applies when adding *b*: already in tree - do nothing.

Example Figure 2.10 shows an example of the complete execution of the simple UA of construction of a suffix tree for $S =$ ”*axabxc*”

2.3.5 Naive Implementation Complexity Analysis

- Following the extension rules, an extension takes only a *constant time* ($O(1)$) once the end of $S[i..j]$ has been found in the suffix tree.

- This makes the major problem in implementing UA is locating the end of the suffix $S[i..j]$ in the tree.

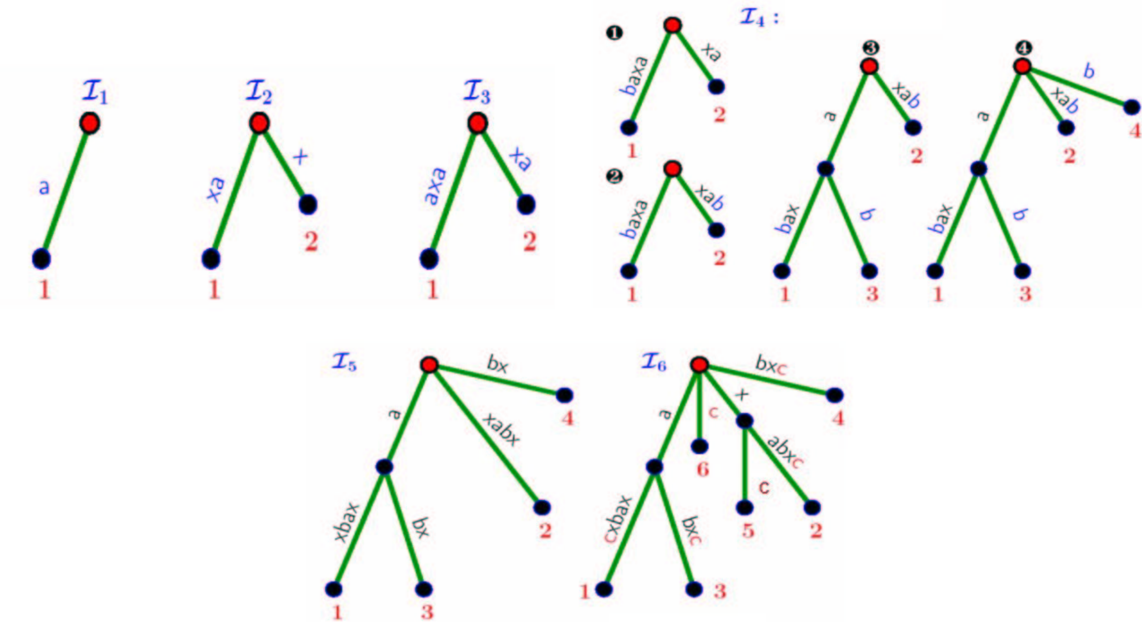

Figure 2.10: Performing the full UA for the string *axabxc*

- *Naive implementation* of this problem is walking down from the root of the current tree. This walk would take $O(i - j)$ time, and I_{i+1} would be created for I_i in $O(i^2)$ time, making the total complexity of the construction $O(m^3)$. This makes the whole idea of UA seem rather foolish, as a simpler algorithm for building a suffix tree in $O(m^2)$ has been presented above.

- However, Ukkonen's linear-time algorithm is based on this high-level approach and reduces the $O(m^3)$ time bound to $O(m)$ using the following tricks and observations:

  - Suffix links
  - Skip and count trick
  - Edge-label compression
  - A stopper
  - Once a leaf, always a leaf

  Each trick by itself does not necessarily reduce the worst-case bound. However, taken together, they achieve the linear worst-case time. The most important element of the acceleration is the use of *suffix links.*

2.3.6 Suffix Links

Definition A *path-label* of the node v is the substring of S constructed by concatenating the labels on the path from the root to v on the suffix tree.

Definition Consider a character x and a (possibly empty) string α. Let v be the node whose path-label is xα and let $s(v)$ be another node with a path-label α. A pointer from v to $s(v)$ (*the edge* $(v, s(v))$) is called a *suffix link*.

Example Figure 2.11 presents an example of 2 suffix links (denoted by arrows) in a tree.

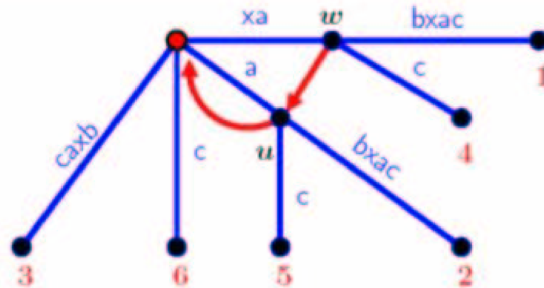

Figure 2.11: The suffix links shown in this figure are : (1) from w to u where $x\alpha = xa$ and $\alpha = a$; (2) from u to the root where $x = a$ and α is the empty string.

Remarks :

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

2. Though the definition itself does not imply that every internal node has a suffix link from it, it will, in fact, have one.

Example Figure 2.12 shows an example of all the suffix links in a suffix tree for the string "*ACACACAC*".

Lemma 2.2 *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 3 possible scenarios can occur:*

- *The path labeled α already ends at an internal node of the tree.*

- *The internal node labeled α will be created in extension $j+1$ in phase $i+1$ by the extension rules.*

Figure 2.12: Suffix tree for the string "$ACACACAC$" along with the suffix links.

- *String α is empty and $s(v)$ is the root.*

Proof: A new internal node can be created only when *extension rule 2* applies. That means that in extension j the path labeled xα continued with some character $c \neq S(i+1)$. Thus in extension $j+1$ there is a path p labeled α (that continues after character c). Now, there are 2 options to be considered (shown in Figure 2.13) :

- path p continues with only the character c, and then extension rule 2 will create a node s(v) at the end of path α.
- path p continues with 2 different characters, then there surely is a node s(v) at the end of p.

Figure 2.13: Two possible options for creating a new internal node.

■

Corollary 2.3 *Any newly created internal node of an implicit suffix tree will have a suffix link from it by the end of the next extension. This can be proved by induction (See [?] section 6.1.3 page 98).*

Corollary 2.4 *In any implicit suffix tree I_i, if an internal node v has path-label $x\alpha$, then* there is *a node $s(v)$ of I_i with path-label α.*

Constructing I_{i+1} using suffix links

As described above, in phase $i+1$, extension j, UA locates suffix $S[j..i]$ of $S[1..i]$. In the naive implementation this was achieved by matching the string $S[j..i]$ along a path from the root. Suffix links can shorten this walk, thus reducing the complexity (As shown in Figure 2.14) .

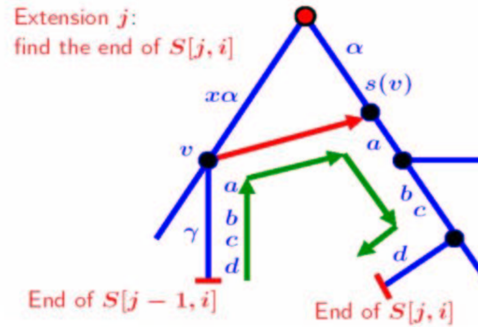

Figure 2.14: Location of the end of $S[j..i]$ using suffix links.

We will start by describing the first 2 extensions ($j=1, j=2$), as they are the most coherent.

For $j=1$, $S[1..i]$ must end with a leaf, since it is the longest string in the implicit tree I_i. That makes the end of this suffix easier to find. We can keep a pointer to the leaf corresponding to the current full string $S[1..i]$ as the trees are constructed. Its suffix extension is handled using Rule 1, so the first extension of each phase ($j=1$) only takes constant time.

For $j=2$, we look for the end of $S[2..i]$. Let's denote $S[1..i]=x\alpha$ and then $S[2..i]=\alpha$. Let $(v,1)$ be the tree-edge that enters leaf 1. There are two options as to node v :

- If v is the root then we descend from the root to find α using the naive algorithm.
- Otherwise, v is an internal node, then by Corollary 2.4, since v was in I_i, v has a suffix link out of it to node $s(v)$. We can follow that link and then descend from $s(v)$ in order

to find the rest of α. Thus, in order to find the end of α in the 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.

When performing extension j , we begin where extension $j-1$ has concluded - that is at the end of $S[j-1..i]$. We perform the following stages in order to find the end of $S[j..i]$:

- We'll begin by finding the first node v at/above $S[j-1..i]$ that has suffix link or is the root. Let's denote as γ the string between v and the end of $S[j-1..i]$.
- If v is an internal node (not the root), we'll use the suffix link to get to $s(v)$ and descend from there to the end of $S[j..i]$.
- If v is the root, we'll descend from there.
- Upon reaching the end of $S[j..i]$, we extend the suffix by $S(i+1)$, according to the extension rules.

It should be noted at this point that the algorithm never has to walk up more than one edge in the beginning of every extension:

- If there is a node at the end of $S[j-1..i]$ (where we began extension j) had a suffix link out of it, the algorithm would transverse that link and get to $s(v)$.
- Otherwise, the end of $S[j-1..i]$ is a node (let's call it w), newly created by *rule 2* in the previous extension. If the parent of w is not the root, than it already has a suffix link out of it (according to Lemma 2.2) - which the algorithm will use.

That alone is a somewhat improval of the algorithm, but does not yet improve the time bound. That leads us to a trick that will, along with the suffix links reduce the worst-case time to $O(m^2)$.

2.3.7 Skip and Count Trick

Problem: We've improved the "walking up" part of the algorithm, but "walking down" the path labeled γ from s(v) to the end of $S[j..i]$, still takes time proportional to the number of characters - $|\gamma|$.

Solution: We'll make the running time of "walking down" proportional to the **number of nodes** on the path. This will make the total time of all the *down walks* in a phase $O(m)$.

We'll add a counter to every edge indicating the number of characters on its label. If there are a couple of possible edges to walk down, we simply look at the first character of every edge and decide which edge to follow (as done before). However, since we already

know the number of characters in γ and the length of the edge where we begin to descend, we can decide by a simple comparison, whether we need to stop in the middle of that edge, or skip directly to the node at the end of it. The algorithm will continue skipping edges this way, until the desired edge that contains the end of γ is found. We can then extract the desired character from that edge in $O(1)$ time, since we know the number of characters of γ we have yet to find, by the time we reach it. The total time to traverse the path of γ is then proportional to the number of *nodes* on it, rather than the number of characters on it. We'll now see how this affects the complexity.

Example Figure 2.15 shows the implementation of the skip and count trick to find the end of suffix $S[i..j]$

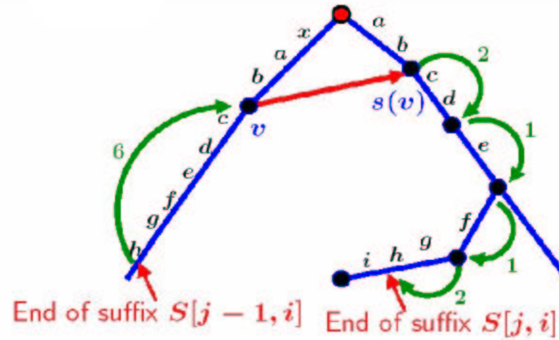

Figure 2.15: The skip and count trick

Definition *Node Depth* of a node v or $ND(v)$ is the number of nodes on the path from the root to v.

Lemma 2.5 *When the suffix link* $(v, s(v))$ *is traversed in UA,* $ND(v) \leq ND(s(v)) + 1$

Proof: When suffix link $(v, s(v))$ is traversed, any internal ancestor of v which has path-label, denoted $x\beta$, has a suffix link to a node with path-label β. But $x\beta$ is a prefix of a path to v, so β is a prefix of the path $s(v)$. This means that a suffix link from any internal ancestor of v goes to some ancestor of $s(v)$. Moreover, if $\beta \neq \emptyset$ then β labels a path to an internal node. Because the NDs of any 2 ancestor of v must differ, each ancestor of v has a suffix link to a distinct ancestor of $s(v)$.
Thus: $ND(v) \leq 1$ (for the root) plus the number of internal ancestors of v, who have path labels ≥ 1 character long. The only ancestor that v can have (without a corresponding ancestor for $s(v)$) is an internal ancestor whose path-label has length 1 (it has label x in the

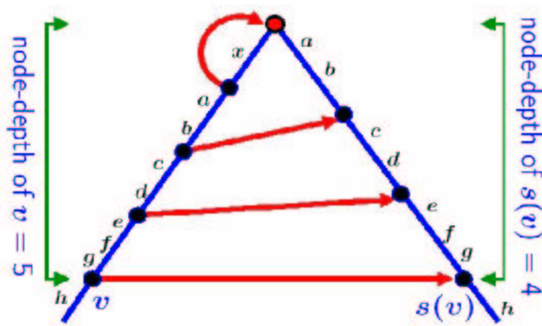

Figure 2.16: Suffix links from the path to v to the path to $s(v)$

following illustration). Therefore, $ND(v) \leq ND(s(v)) + 1$. For example, see Figure 2.16 ■

Definition The *Current Node Depth* of the algorithm is the node depth of the node most recently visited by the algorithm.

Theorem 2.6 *Using the skip and count trick, any phase of Ukkonen's algorithm takes $O(m)$ time.*

Proof:

- We've already shown that when implementing suffix links, all the up-walks, along with traversing and creating links take $O(1)$ per extension, summing to $O(m)$ per phase of the algorithm. It remains to show that the down-walks of the phase can be completed in $O(m)$ as well.
- Also, since it has been shown that every up-walk of the algorithm only moves one node up, the current node-depth can change only by ≤ 1 at any extension.
- Suffix link traversal also decreases the ND by at most one, as has been shown by Lemma 2.5.
- Over the entire phase the current node depth is decremented at most $2m$ times, since the number of extensions is $\leq m$ and every extension involves no more than one step up and one suffix link traversal. As every step walking down, using the skip and count trick, increases ND by one and since no node can have depth $> m$ the total possible increment to current node depth is $\leq 3m$ over the entire phase. It follows that over the entire phase, the total number of edge traversals during down-walks is bounded by $3m$.

- Since using the skip and count trick the time per down-edge traversal is constant, the total time for down-walks in a phase is $O(m)$, bounding the entire running-time of a phase to $O(m)$.

■

Since overall the algorithm performs m phases:

Corollary 2.7 *Using suffix links along with the skip and count trick Ukkonen's algorithm can be implemented in $O(m^2)$ time.*

This implementation is still not very interesting, as it still does not provide with a linear-time solution. So far, all the changes that have been made were directed to a single phase in the algorithm. The final solution will be provided by a few more changes to the implementation allowing a time analysis crossing phase boundaries.

2.3.8 Edge-label compression

Another problem that has been ignored so far involves the space complexity of the algorithm. The problem with the implementation of the algorithm as it has been described so far, is that the suffix tree may require $\theta(m^2)$ space. Since the time for the algorithm is at least as large as the size of its input, that much space makes the $O(m)$ bound impossible.

Consider the string "$S = abcdefghijklmnopqrstuvwxyz$". Every suffix begin with a distinct character; hence there are 26 edges out of the root, each labeled with a complete suffix, requiring $26 * 27/2$ in all characters. For strings longer in length than the size of the alphabet, some characters will repeat, of course, but strings of arbitrary length m are still possible, requiring edge-labels longer than $\theta(m)$ characters in total. This calls for some alternate way to present the edge-labels.

Definition For *Edge Label Compression* instead of writing an explicit substring (which is contiguous) on every edge, write a *pair of indices* on the edge, specifying beginning and end position of that substring in the original string.

Example Figure 2.17 shows an edge-label compression performed on a suffix tree for $xabxa$.

Using the two indices, since the algorithm has a copy of the original string S for random-access, it can easily extract the desired substring when needed. This allows us to construct a suffix tree with only a constant number of symbols written on any edge.

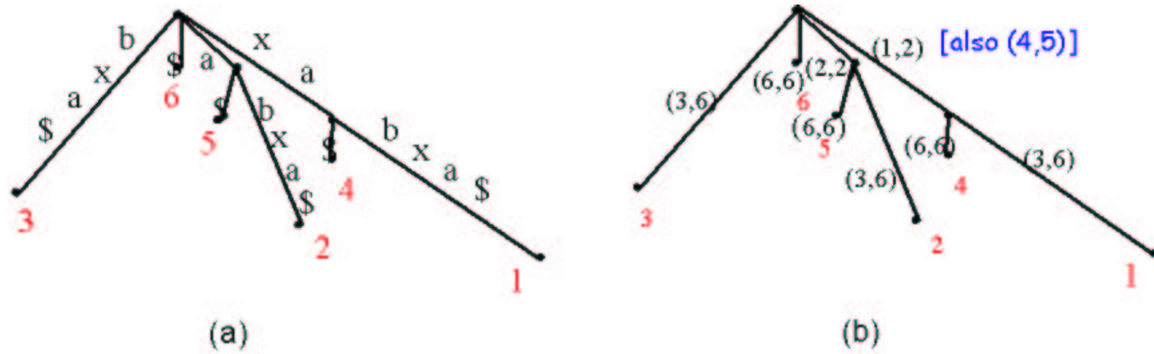

Figure 2.17: Edge-label compression of the suffix tree for the string *xabxa* shown in (a) results in a suffix tree shown in (b).

Implementing the labeling scheme in UA

The labeling scheme described above can be easily implemented in Ukkonen's algorithm. While traversing down the edges during the algorithm, the index pair on each edge is used to fetch the needed characters from S.

The extensions can be carried out in the following way:

- When *extension rule 1* applies in phase $i+1$, we needed to extend an existing edge by a single character. When using edge-label compression, we simply change the index pair from (p, q) to $(p, q+1)$.
- When *rule 2* applies, a new edge was created for a new leaf and labeled with the character $S(i+1)$. When implementing compression, we label the new edge with $(i+1, i+1)$
- When *rule 3* applies, no change is necessary, with or without compression.

By using the index pairs to label the edges, only 2 numbers are written on any edge. Since the number of edges in a tree with m leafs is at most $2m-1$, that makes the total number of characters needed to represent a suffix tree $O(m)$, thus removing the space complexity barrier.

2.3.9 Rule 3 is a stopper

Observation: In phase $i+1$, once *extension rule 3* applies in extension j, it will also apply in all further extension of that phase ($j+1$ to $i+1$). In general, if a string is already included in the suffix tree, all its suffixes are also already included. In the algorithm, when *rule 3*

applies in phase $i+1$ extension j, that means that the suffix $S[j..i+1]$ is already in the tree. Following the above, this means that for every $k > j$, $S[k..i+1]$ is also already included. Remember there are no changes made when Rule 3 applies, and also no new suffix link needs to be made from this point until the end of the phase (suffix links are added only when *rule 2* applies).

This leads us to the following implementation trick: end phase $i+1$ at the first time that extension rule 3 applies.

Definition The extensions in phase $i+1$ that are supposed to be done (without the trick) after first execution of rule 3 are referred to as *implicit*. Any other extension j (where the end of $S[j..i]$ is explicitly found) is called an *explicit* extension.

This seems like a good heuristic, but only after one more observation and trick the desired time bound is reached.

2.3.10 Once a Leaf, Always a Leaf

Observation: if at some point during the execution of the algorithm, a new leaf is created and labeled j, it will remain a leaf until the conclusion of the algorithm. That is true, since there is no mechanism for extending a leaf edge beyond its current leaf in the algorithm. Actually, if the leaf has been added in phase i, in all the successive phases, *extension rule 1* will apply. Thus, *once a leaf, always a leaf.*

2.3.11 Bringing it all together

Now, in every phase i there's a sequence of extensions (starting from 1) when rules 1 and 2 will apply (until rule 3 applies and we cease the phase). Let j_i denote the last extension in that sequence. Remember, that any application of *rule 2* in the sequence creates a new leaf. That means that in phase $i+1$ in *all* the first j_i extensions we'll need to extend existing leaf edges, by applying *rule 1*. From the above observation, in phase $i+1$ there are j_i initial applications of *rule 1*, hence $j_i \leq j_{i+1}$. That is, since the number of leaves in the tree keeps growing, and we need to extend all the leaf edges at every phase, the sequence of initial extensions can not shrink. This leads us to the following trick:

In phase $i+1$, when leaf p is added and its leaf edge is first created, it would normally be labeled $S[p..(i+1)]$ (which we would compress as $(p, i+1)$). Instead of doing that, we label it with (p, e), where e is a symbol denoting *"the current end"*. Symbol e is a *global* index, which can be set to $i+1$ once in each phase in a constant time ($O(1)$ for all the leaf edges).

In phase $i+1$ since *rule 1* will apply in extensions $1..j_i$ at least, all those extensions can be completed in $O(1)$ time by incrementing e and then continuing with *explicit* work for extensions j_i+1 until rule 3 applies.

Putting all this together we can summarize the work in each phase as follows:

Single Phase Algorithm: SPA

1. Increment e to $i+1$. This takes care of extensions $1..j_i$.

2. Explicitly compute successive extensions (starting from j_{i+1}, as described, implementing rules 1 and 2, until the first extension j^* where *rule 3* applies or until the end of the phase. This takes care of all the needed extensions in the phase.

3. Set j_{i+1} to $(j^* - 1)$, for the use of the next phase.

This way, phase $i+2$ will begin its explicit work with extension j^* which was the *last* explicit extension performed by phase $i+1$. Therefore, *at most one index* where an explicit extension is performed is shared by two consecutive phases (as seen in Figure 2.18). Besides, phase $i+1$ ends at the end of $S[j^*..i+1]$, which is exactly where extension j^* of phase $i+2$ needs to be done. This allow the first extension of any phase to be done without up-walking, link traversal and down-walking - in $O(1)$ time.

	...	11	12	13	14	15	16	17	18	...
Phase i		♣	♣	♣						
Phase i+1				♣	♣	♣	♣	♣		
Phase i+2								♣	♣	

Figure 2.18: Explicit extensions made in phases i, $i+1$ and $i+2$ is denoted by ♣.

Theorem 2.8 *Using suffix links and implementation tricks 1,2 and 3, Ukkonen's algorithm builds implicit suffix trees I_1 through I_m in $O(m)$ total time.*

Proof: The time for all the implicit extensions in a phase is constant, summing to $O(m)$ over the entire algorithm. ■

Example Figure 2.19 shows an example of the entire algorithm so far for constructing a series of implicit suffix trees for the string *axaxbb*$.

2.3.12 Creating the final true suffix tree

It remains to show that the last of the series of the implicit suffix trees - I_m can be converted to a true suffix tree in $O(m)$ time. This is done utilizing the following stages :

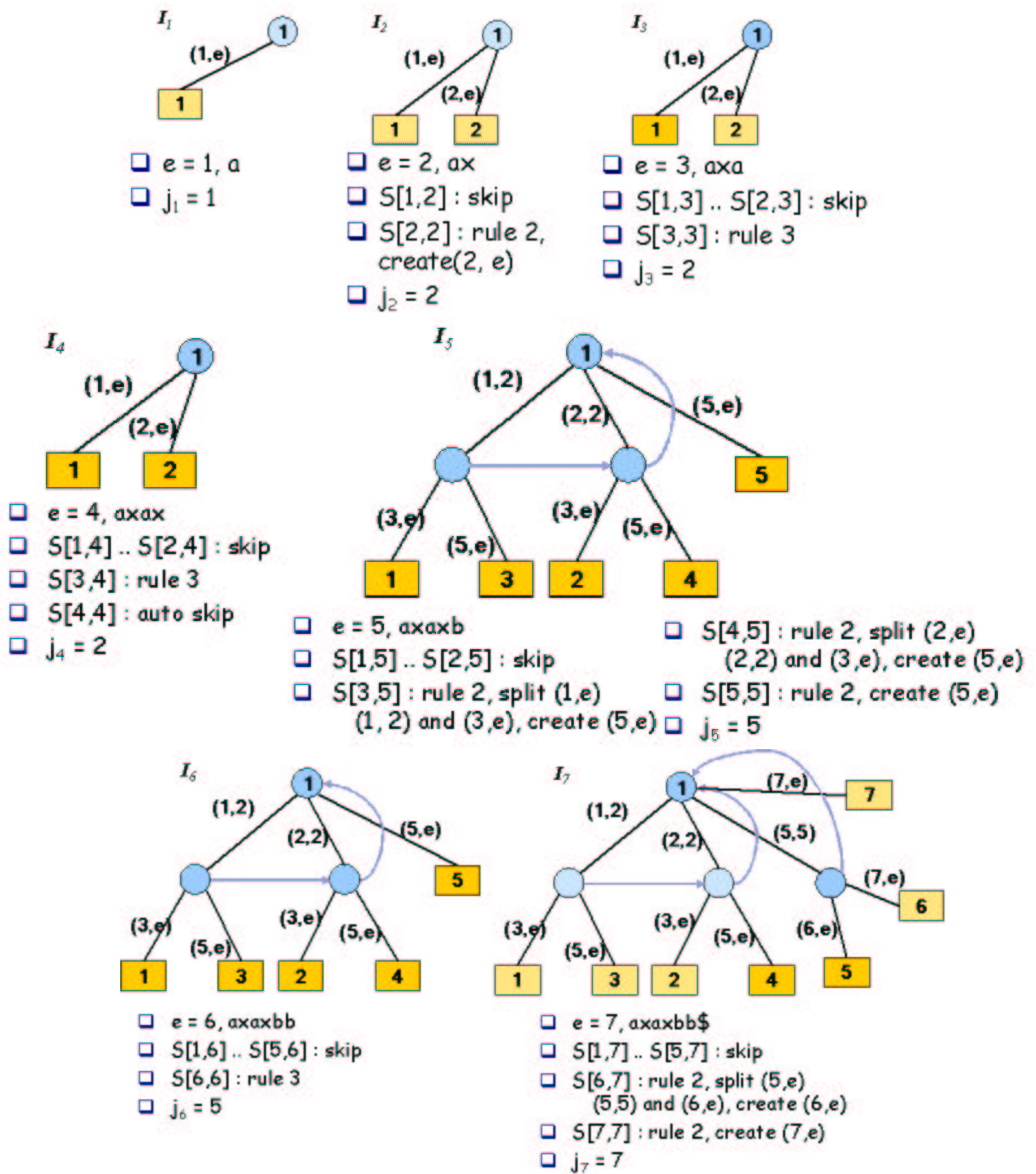

Figure 2.19: Explicit extensions made in phases i, $i+1$ and $i+2$ is denoted by ♣.

- Add a string terminal symbol \$ to the end of S and let the algorithm make another phase with this character. This makes no suffix in the tree a prefix of another suffix, so that each suffix now ends with a leaf and is explicitly represented.
- Replace every index e on every leaf edge with the number m. This can be done by $O(m)$ traversal of the tree, visiting every leaf edge.

The result of the above modification is a true suffix tree. Thus, all the three stages of Ukkonen's Algorithm can be done in $O(m)$ time, leading us to:

Theorem 2.9 *Ukkonen's algorithm builds a true suffix tree for S, along with all its suffix links in $O(m)$ time.*

2.4 Implementation Issues

Serious problems can arise while implementing a suffix tree when the size of the alphabet grows. If the tree can't be kept entirely in the computer's memory, the use of suffix links, which gave us the linear time-bound, will cause a lot of paging, thus making the running time significantly slower. There are a few basic choices that can be made as to how to implement every node of the suffix tree data structure, every choice having its benefits and disadvantages.

1. **An array** of the size $\theta(\Sigma)$ at every non-leaf node. The cell indexed by character x has a pointer to a child of v if there is an edge out of v starting with x, and *null* otherwise. The problem with this implementation is that for a large alphabet (such as amino-acids for a protein sequence, where $|\Sigma| = 20$), this structure will take an impractical amount of space.

2. For every node v, we can use **a linked list** of the characters that appear at the beginning of the edge labels out of v. While traversing the tree, the algorithm will search the list for the right character. This list can be kept in a sorted order, reducing the average time of search for a given string. The problem is that at the worst-case scenario, every node visit will cost an extra $|\Sigma|$ time. If the number of children of v is large, little space will be saved in comparison to the array implementation, while significantly degrading performance.

3. The list at the node v can be also implemented using a balanced tree, making a compromise between space and speed. This will make the additional searches take $O(\log k)$ time and every node will take $O(k)$ space, where k denotes the number of children of v in the tree. The implementation, however, is a much more complex one, thus making it relevant only for a fairly large k (making the difference between k and $\log k$ a significant one).

4. A hashing scheme can also be used to implement every node. The challenge in this case is to find a scheme that will balance space with speed. When m and $|\Sigma|$ are fairly large, hashing can be very attractive, at least for some of the nodes.

5. When m and $|\Sigma|$ are large enough, the best design is probably a mixture of the above choices. Since nodes near the root of the suffix tree tend to have the most children (the root, for example, has a child for every character appearing in the tree), an array scheme is the most sensible application for them. On the other hand, for nodes in the middle of the tree, hashing or balanced trees may be the best choices, as they have a smaller k.

Sometimes, when the alphabet size is fairly large, it is explicitly presented in the time and space bounds of the algorithm, making construction of the suffix tree $O(m \log |\Sigma|)$, using $\theta(m|\Sigma|)$ space when m is the size of the string. It should be noted, that this is usually not the case in computational biology, as the size of the alphabet is usually rather small.

2.5 Applications Of Suffix Trees

2.5.1 Exact String Matching & Counting Number of Pattern Occurrences

As described in the Introduction, the suffix trees algorithm is very useful for problems of matching multiple distinct patterns with a rarely-changing text (in biology - a long DNA sequence such as an entire genome). It could also be applied for finding all occurrences of a certain pattern in that text. Given a text T containing $|T| = n$ characters, we preprocess it by constructing a suffix tree in $O(n)$ time using the described algorithm. We then can use a pattern P to traverse the tree accordingly (as shown in Figure 2.20)

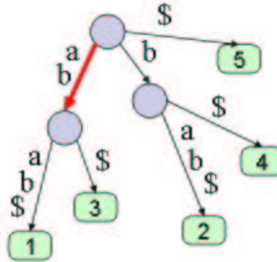

Figure 2.20: Matching the pattern *ab* in the suffix tree of *abab*

If we succeed to traverse the entire pattern, it occurs in the text. Besides, the number of leaves in the subtree below the node we've reached in the tree indicates the number of the

occurrences of the pattern in the text. This allows us to find all k occurrences in $O(n+k)$ time.

2.5.2 Generalized Suffix Tree

Definition Given a set of strings S a *generalized suffix tree* of S is a compressed trie of all the suffixes of all strings $s \in S$.

In order to associate each suffix with a unique string in S, a **different** special terminating character (such as \$) should is added to each s.

Example Figure 2.21 shows a generalized suffix tree for string $s_1 =$ "*abab*" and $s_2 =$ "*aab*", using a terminating character \$ for s_1 and # for s_2.

Figure 2.21: A generalized suffix tree

The generalized suffix tree can be used for matching a pattern against a database of strings - for example, genomes from multiple organisms, or a set of different chromosomes.

2.5.3 Longest Common Substring (of two strings)

In order to find a longest common substring of two string we first build a generalized suffix tree for the strings. Every node with a leaf descendant from s_1 and a leaf descendant from s_2 represents a common substring of s_1 and s_2. Such a node with a maximal string depth represents the **longest** common substring.

Example Figure 2.22 shows an example of locating the longest common substring of "*abab*" and "*aab*" ("*ab*"). The node with the maximal string depth is marked with blue.

Figure 2.22: Using a generalized suffix tree for locating a longest common substring of two strings.

Example Figure 2.23 shows another example of locating the longest common substring, this time for the string *xabxac* and *abx*.

Figure 2.23: Locating the longest common substring of *xabxac* and *abx*.

2.5.4 Lowest Common Ancestor and the Longest Common Prefix

Definition In a rooted tree T, the *lowest common ancestor (LCA)* of two nodes x and y is the deepest node in T that is the ancestor of both x and y.

Example Figure 2.24 shows an example of a node that is the LCA of 2 designated leaves

Further applications of the suffix tree can be made possible if the tree is preprocessed to find the lowest common ancestors. This preprocessing won't be described here, but it can be made in a *linear* time, and will allow the location of the LCA in a *constant* time.

A sample application of a suffix tree preprocessed for locating LCAs is location of *longest common prefix* of 2 suffixes. This can be done by simply locating the LCA of the leaves representing those suffixes, and reading the edge labels from the root to that node.

Example Figure 2.24 shows an example of s suffix tree preprocessed for locating LCAs used to find the longest common prefix of all the suffixes of the string *abab*.

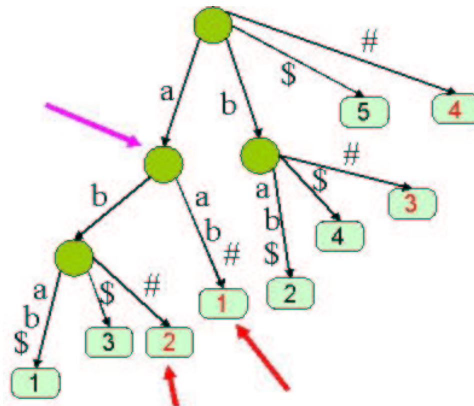

Figure 2.24: Locating the longest prefix of all the suffixes of *abab*.

2.5.5 Finding Maximal Palindromes

Definition An even-length substring s of S if a *maximal palindrome* of radius k, if starting in the middle of s, it reads the same in both directions for k characters but not for any $k' > k$. An odd length maximal palindrome s is similarly defined after excluding the middle character of s.

Example For $S = aabactgaaccaat$, substrings aa, aba and $aaccaa$ are maximal palindromes of radius 1,2 and 3 respectively.

Any other palindrome found in the sequence will be contained within some maximal palindrome, so the most interest is in finding them.
Observation: The maximal palindrome with center between $i-1$ and i is the LCA of the suffix at position i and the suffix at the position $m-i+1$ of s^r (s reversed).

Maximal Palindromes Algorithm

In order to find the *maximal palindromes* of the string S:

1. Prepare a generalized suffix tree for S and S^r.

2. For every i, find the LCA of suffix i of S and suffix $m - i + 1$ of S^r.

Example Figure 2.25 shows the generalized suffix tree for $S = cbaaba\$$ and $S^r = abaabc\#$. In this tree we need to check 8 pairs of leaves - $i = 1..7$ of the $\$$ leaves with $j = 7..1$ of $\#$ leaves accordingly (for instance check leaf pair $i = 1, j = 7$).

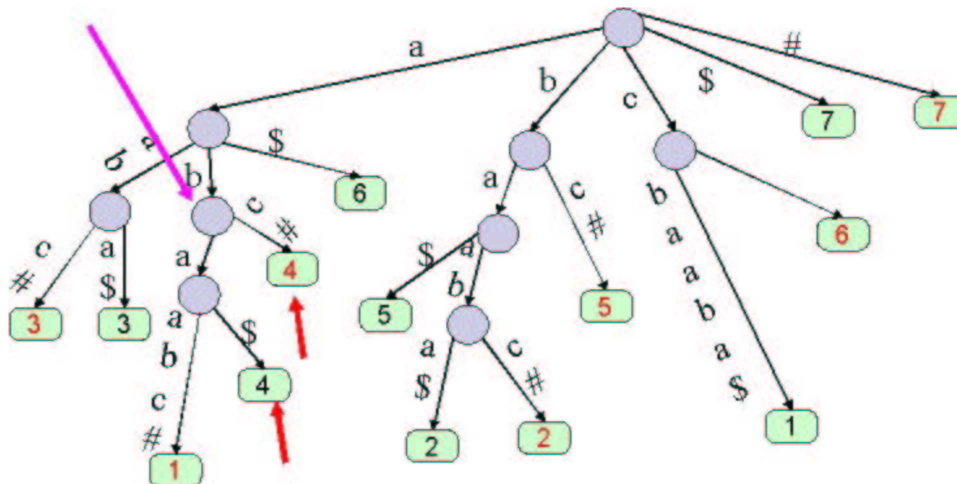

Figure 2.25: A generalized suffix tree for finding maximal palindromes in the string *cbaaba*. The red arrows indicate the only pair of matching leaves from the $ and the # suffixes that have an LCA (designated by the purple arrow).

2.6 Suffix Arrays

As described in Section 2.4, although suffix trees use linear space, the constant can be rather large as the alphabet and the size of the string grow larger, making the use of the suffix trees impractical. Hence, a more space efficient algorithm is desired that would still retain most of the advantages of searching with a suffix tree.

A *suffix array* is a data structure which can be used to solve the exact matching problem and the substring problem almost as efficiently as the suffix tree, while using very little space.

Definition Given an m-character string S, a *suffix array* for S, called Pos, is an array of integers in the range 1 to m, specifying the lexicographic order of the m suffixes of the string S.

That means that $Pos[1]$ will hold the starting position of the least lexicographically significant suffix of S, and every $Pos[i]$ would indicate a suffix less significant than $Pos[i+1]$.

Example For $S = abab$, the suffixes sorted lexicographically are $ab, abab, b, bab$. Thus the suffix array for S would be

3	1	4	2

.

Notice that since the suffix array contains only integers, it contains no information about the alphabet used in the string T. Thus, for a string of the length m, the structure requires only $O(m)$ computer words.

Construction of a Suffix Array from a Suffix Tree

A suffix array for the string S can be constructed efficiently in the following way:

1. Construct a suffix tree for S in a linear time (using Ukkonen's algorithm, for example). Here we assume that enough space is available to construct it once in the preprocessing stage.

2. Traverse the tree using the *DFS (depth-first)* graph traversal algorithm, lexicographically picking edges outgoing from each node, and fill the suffix array Pos. A lexicographical order on the edges exists, as no two edges out of a node begin with the same character. This will fill Pos with suffixes in their lexicographical order.

Since both constructing the suffix tree and traversing it using DFS take linear time, this way the suffix array for S can be constructed in $O(m)$ time.

Example Figure 2.26 shows a suffix array structure for the string $S = mississippi$.

Searching for a Pattern Using a Suffix Array

Once a suffix array has been constructed for a string, a pattern P can be easily searched in it. If P occurs in S then all its locations will appear consecutively in Pos. This means that in order to find all the occurrences of P in S all we need is to perform a simple binary search over the suffix array to find the lexicographically first occurences, and then check the next cells in the array for additional matches.

If we assume that the comparison itself between the strings takes time proportional to the length of the pattern $O(n)$ this makes the cost of all the binary search $O(n \log m)$.

Figure 2.26: A suffix array for the string *mississippi*