Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. SP02031 Call Num:

General Guidelines:

- Library books should be used with great care.
- Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT

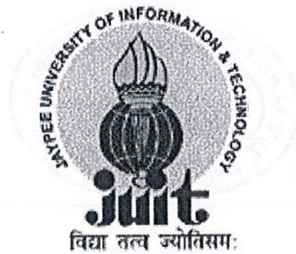
SP02031

SP2031

STUDIES
ON
"GENERALIZATION OF A SUFFIX TREE FOR RNA STRUCTURAL PATTERN MATCHING"

By

SHIKHA KAUSHAL-021505



Learning Resource Centre
JUIT
Waknaghat, Solan (H.P.)

MAY-2006

Submitted in partial fulfillment of the Degree of Bachelor of Technology

DEPARTMENT OF BIOINFORMATICS
JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY-WAKNAGHAT

Learning Resource Centre
JUIT
Waknaghat Solan (H.P.)

CERTIFICATE

This is to certify that the work entitled, **Studies on "Generalization of A Suffix Tree for RNA Structural Pattern"**, submitted by SHIKHA KAUSHAL in partial fulfillment for the award of the degree of Bachelor of Technology in Bioinformatics of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

(Dr. Soma Marla)
HOD
Department of Bioinformatics

ACKNOWLEDGEMENT

The Project **Studies on "Generalization of A Suffix Tree for RNA Structural Pattern"**, was successfully completed under the guidance of **Prof. Ashok Subramanian** Thus I sincerely thank Prof. Ashok Subramanian who provided his knowledge and help in the accomplishment of this project.

I also acknowledge the guidance of Dr Soma Marla, HOD, Bioinformatics

Thank You ...!

CONTENTS

Abstract..4

Introduction

Project Overview..5

Preface...6

RNA Structural Matching 6

SUFFIX TREES

Introduction to Suffix Trees..9

Short History...9

Basic Suffix Tree Definition..10

Ukkonen's linear-time suffix tree algorithm................................12

Implicit suffix trees..12

Description of Ukkonen's algorithm...................................14

Suffix Extension rules...15

Implementation and speedup...17

Suffix links..18

Single extension algorithm..22

What have been achieved so far...23

A simple implementation detail...30

Edge-label compression..30

Two more tricks..32

Single phase algorithm 35

Creating the true suffix tree...38

Suffix arrays-more space reduction..39

SUFFIX TREE TO SUFFIX ARRAY CONVERSION…………………………………………………….42

Future scope ……………………………………………………………………45

Conclusion………………………………………………………………………45

ABSTRACT

In molecular biology, it is said that two biological sequences tend to have similar properties if they have similar three-dimensional structures. Hence, it is very important to find not only similar sequences in their string sense, but also structurally similar sequences from databases. In this project a new data structure is studied that is a generalization of a parameterized suffix tree (p-suffix tree for short)

introduced by Baker. It is called the structural suffix tree or s-suffix tree for short. The s-suffix tree can be used for finding structurally related patterns of RNA or single-stranded DNA. Furthermore, an $O(n(\log|\Sigma| + \log|\Pi|))$ on-line algorithm is studied which is used for constructing it, where n is the sequence length, $|\Sigma|$ is the size of the normal alphabet, and $|\Pi|$ is that of the alphabet called "parameter," which is related to the structure of the sequence. As an algorithm for constructing the p-suffix tree, it is the first online algorithm, though the computing bound of the algorithm is the same as that of Kosaraju's best-known algorithm.

INTRODUCTION

PROJECT OVERVIEW

The object of the study is to build a better understanding of the proposed paper :

Generalization of a Suffix Tree for RNA Structural Pattern Matching.

This project also proposes a method to reduce the space complexity of the algorithm proposed in the paper i.e. by using suffix array in place of suffix tree. A method to convert a suffix tree to suffix array is also described

PREFACE

Why RNA Structural Matching?

Introduction.

The three-dimensional structure of a biological sequence plays a major role in determining its functions and properties, and sequences that have similar structures often have similar functions, even if the sequences themselves are not similar. Thus it is very important to predict, compare, or find structures of sequences in molecular biology, but they are very difficult and challenging tasks. On the other hand, stringological comparison of biological sequences without considering their structures is much easier. There are many efficient algorithms for it. For example, we can find frequently appearing sub strings in a sequence very efficiently using a very useful data structure called suffix trees. In this paper we strive to generalize the suffix trees to do structural analysis of RNA sequences as efficiently as we do ordinary string analysis.

Example of sequences that have high possibility of having the same structure

RNA sequences consist of four kinds of bases: A (adenine), U (uracil) ,C(cytosine), and G(guanine). A and U are said to be complements of each other, and C and G are also complementary bases. RNA and single-stranded DNA sequences often form some structures by combining two complementary base pairs. It is known that double-stranded DNA sequences sometimes from such structures by becoming single-stranded locally. Many computational studies have been done to predict RNA

secondary structures, comparing a new sequence with a known RNA structure, searching a known RNA or DNA structures from large databases, and so on. However, many of them are much slower than ordinary string comparison analyses, which are often done, in linear time. Our work finds a way out of it.

Sequence1: AUAUCGUAUGGCCGAGCC
Sequence2: CGCGUAGCGAAUUACAUU

FIG 1(a)

We consider the two RNA sequences shown in Figure 1(a). The two sequences are not at all similar to each other; there are no identical bases in identical positions. In sequence 1, A's are located at the 1st, 3rd, 8th, and 15th positions. In sequence 2, C's are located at the same position as A's in sequence1. Similarly, A's, U's,

and G's in sequence 2 are located at the same position as G's, C's and U's in sequence 1, respectively. Recall that A and U can combine with each other, and that C and G can also combine with each other. We then notice the following fact: if two bases in one of these sequences can combine with each other, then in the other, then in the other sequence, two bases at the same two positions are also able to combine with each other. This implies that a structure that can be formed by one of the sequences can also be formed by the other sequence. Thus there is a strong possibility that these two sequences have the same structure, and consequently may have similar properties. For example, Figure 1(b) shows one of the structures that can be formed by both sequences.

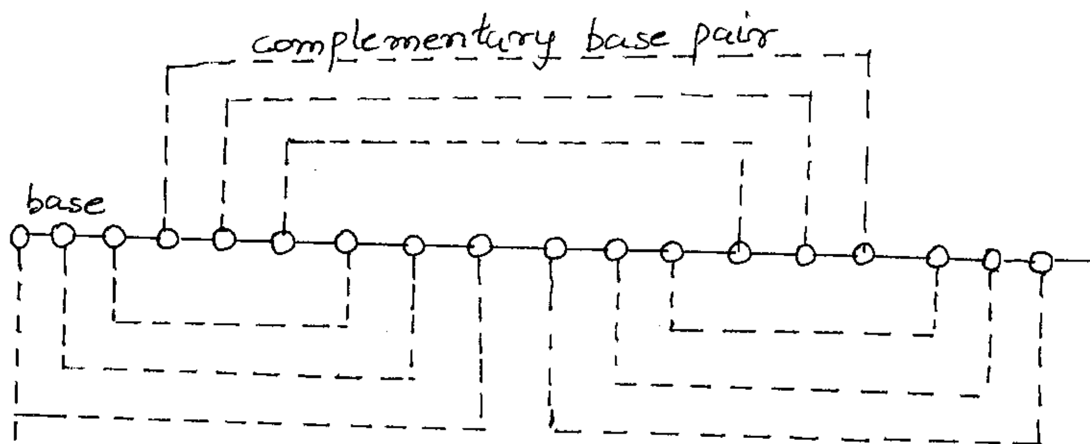


Fig 1(a)

Introduction to Suffix Trees

A suffix trees is a data structure that exposes the internal structure of a string in a deeper way than does the fundamental preprocessing used in string matching. Suffix trees can be used to solve the exact matching problem in linear time, but their real virtue comes from their use in linear-time solutions to many string problems more complex than exact matching. For example RNA structural matching .

A Short History

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

Basic Suffix Tree Definition

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

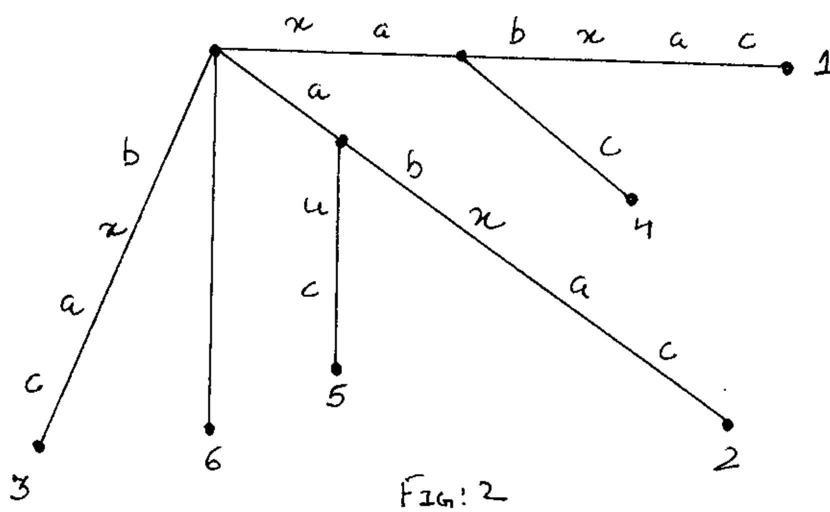


FIG: 2

For example, the suffix tree for the string *xabxac* is shown in Figure 2. The path from the root to the leaf numbered 1 spells out the full string S = *xabxac,* while the path to the leaf numbered 5 spells out the suffix ac, which starts in position 5 of S. As stated above, the definition of a suffix tree for S does not guarantee that a suffix tree for any string S actually exists. The problem is that if one *suffix* of S matches a *prefix* of another suffix of S then no suffix tree obeying the above definition is possible, since the

path for the first suffix would not end at a leaf. For example, if the last character of *xabxac* is removed, creating string *xabxa,* then suffix *xa* is a prefix of suffix *xabxa*, s o the path spelling out *xa* would not end at a leaf.

To avoid this problem, we assume (as was true in Figure 2) that the last character of S appears nowhere else in S. Then, no suffix of the resulting string can be a prefix of any other suffix. To achieve this in practice; we can add a character to the end of S that is not in the alphabet that string S is taken from. In this work we use $ for the "termination" character. When it is important to emphasize the fact that this termination character has been added, we will write it explicitly as in S$. Much of the time, however, this reminder will not be necessary and, unless explicitly stated otherwise, every string S is assumed to be extended with the termination symbol $, even if the symbol is not explicitly shown.

Definition: The *label of a path* from the root that ends at a *node* is the concatenation, in order, of the sub strings labeling the edges of that path. The *path-label of a node* is the label of the path from the root of T to that node.

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

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

For example, in Figure 2 string *xa* labels the internal node *w* (so node *w* has path-label *xa*), string a labels node *u*, and string *xabx* labels a path that ends inside edge *(w, 1)*, that

is, inside the leaf edge touching leaf 1.

Ukkonen's linear-time suffix tree algorithm

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

Implicit suffix trees

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

.

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

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

Definition. We denote the implicit suffix tree of the string S [1.. i] by Ii, for I from 1 to m.

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

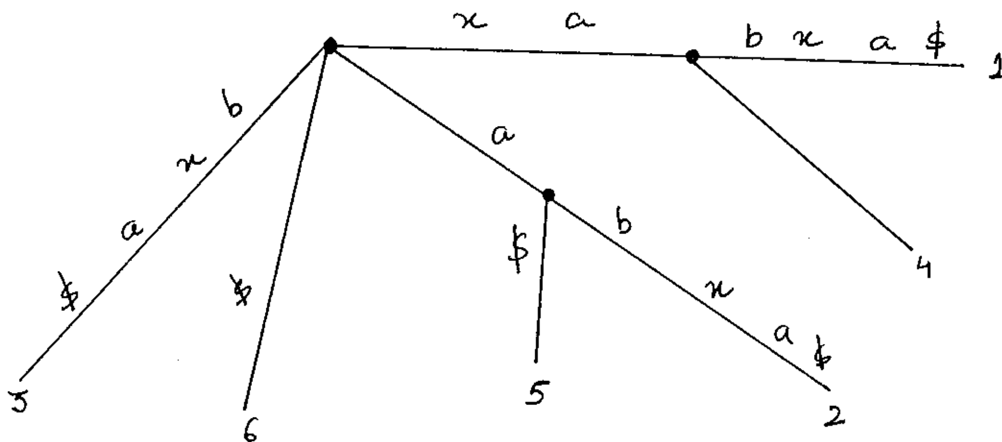


FIG 3

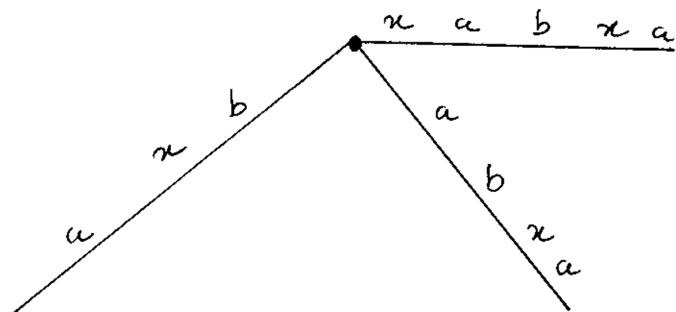


FIG 4

As an example, consider the suffix tree for string *xabxa$* shown in Figure 3. Suffix *xa* is a prefix of suffix *xabxa,* and similarly the string *a* is a prefix of *abxa.* Therefore, in the suffix tree for *xabxa* the edges leading to leaves 4 and 5 are labeled only with $. Removing these edges creates two nodes with only one child

each, and these are then removed as well. The resulting implicit suffix tree for *xabxa* is shown in Figure 4.

As another example, Figure 2 shows a tree built for the string *xabxac.* Since character *c* appears only at the end of the string, the tree in that figure is both suffix tree and an implicit suffix tree for the string.

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

Ukkonen's algorithm

Ukkonen's algorithm constructs an implicit suffix tree *Ii* for each prefix *S* [1..*i*] of *S* , starting from *I1* and incrementing *I* by one until *Im* is built. The true suffix tree for *S* is constructed from *Im*, and the time for the entire algorithm is *O (m).* We will explain Ukkonen's algorithm by first presenting an *O(m3)*-time method to build all trees *Ii* and then optimizing its implementation to obtain the claimed time bound

Description of Ukkonen's algorithm

Ukknone's algorithm is divided into *m phases.* In phase *i*+ 1, tree *Ii*+ 1 is constructed from *Ii*. Each phase *i*+ 1 is further divided into *i*+ 1 *extensions,* one for each of the *i*+ 1 suffixes of *S* [1..*i* + 1]. In extension *j* of phase *i*+ 1, the algorithm first finds the end of the path from the root labeled with substring *S* [*j*..*i*]. It then extends the substring by

adding the character S (i+ 1) to its end, unless S (i+1) already appears there. So in phase i+ 1, string S [1..i+ 1] is first put in the tree, followed by strings S [2..i + 1], S [3..i + 1],...(in extension 1,2,3... respectively). Extension i + 1 extends the *empty* suffix of S[1+ I], that is, it puts the single character string S (I + 1) into the tree (unless it is already there). Tree $I\iota$ is just the single edge labeled by character S (1). Procedurally, the algorithm is as follows:

High-level Ukkonen algorithm

Construct tree $I1$.

For $i$from 1 to m – 1 do

Begin {phase i. + 1}

For j from 1 to i+ 1

Begin {extension j}

Find the end of the path from the root labeled S [j..i] in the Current tree.

If needed, extend that path by adding character S (i+ 1),

end;

end;

Suffix extension rules

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

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

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

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

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

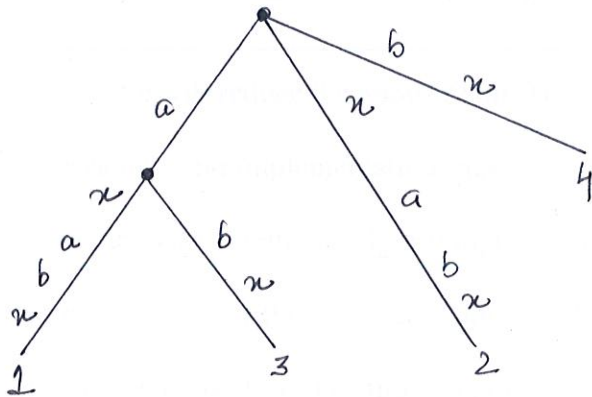


FIG 5

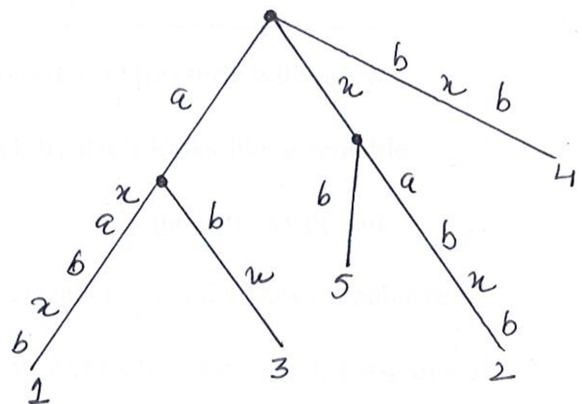


FIG 6

As an example, consider the implicit suffix tree for $S = axabx$ shown in Figure 5. The first four suffixes end at leaves, but the single character suffix x ends inside an edge. When a sixth character b is added to the string, the first four suffixes get extended by applications of rule 1, the fifth suffix gets extended by Rule 2, and the sixth by Rule 3.

The result is shown in Figure 6

Implementation and speedup

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

Naively we could find the end of any suffix ß in $O(|ß|)$ time by walking from the root of the current tree. By that approach, extension j of phase $i + 1$ would take $O(i + 1 - j)$ time, I_{i+1} could be created from I_i in $O(i^2)$ time, and I_m could be created in $O(m^3)$ time. It is easier to describe Ukkonen's $O(m)$ algorithm as a speedup of the $O(m^3)$ method above.

We will reduce the above $O(m^3)$ time bound to $O(m)$ time with a few observations and implementation tricks. Each trick by itself looks like a sensible heuristic to accelerate the algorithm, but acting individually these tricks do not necessarily reduce the worst-case bound. However, taken together, they do achieve a linear worst-case time.The most important element of the acceleration is the use of *suffix links.*

Suffix links: first implementation speedup

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

We will sometimes refer it a suffix link from *v* to *s(v)* as the pair (*v*,*s(v)*). For example, in Figure 3 let *v* be the node with path-label *xa* and let *s(v)* be the node whose path-label is the single character *a*. Then there exists a suffix link from node *v* to node *s(v)*. In this case, *a* is just a single character long.

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

Lemma 1.1. *If a new internal node v with path-label xa is added to the current tree in extension j of some phase i+* 1, *then either the path labeled a already ends at an internal node of the current tree or an internal node at the end of string a will be created (by the extension rules) in extension j* + 1 *in the same phase i* +1.

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

a. When a is continued only by c, extension rule 2 will create a node $s(v)$ at the end of
b. path a. When a is continued with two different characters, then there must
c. already be a node $s(v)$ at the end of path a. The *Lemma* is proved in either case.

Corollary 1.1. In Ukkonen's algorithm, any newly created internal node will have a suffix link from it by the end of the next extension.

PROOF The proof is inductive and is true for tree $I1$ since $I1$ contains no internal nodes. Suppose the claim is true through the end of phase i, and consider a single phase $i + 1$. By Lemma 1.1, when a new node v is created in extension j, the correct node $s(v)$ ending the suffix link from v will be found or created in extension $j + 1$. No new internal node gets created in the last extension of a phase (the extension handling the single character suffix $S(i + 1)$), so all suffix links from internal nodes created in phase $i + 1$ are known by the end of the phase and tree $Ii+1$ has all its suffix links.

Corollary 1.2 In any implicit suffix tree Ii if internal node v has path-label xa, then there is a node $s(v)$ of Ii with path-label a.

Following Corollary 1.1, all internal nodes in the changing tree will have suffix

links from them, except for the most recently added internal node, when will receive its suffix link by the end of the next extension. We now show how suffix links are used to speedup the implementation.

Following a trail of suffix links to build I_{i+1}

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

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

Let string $S[1..i]$ be xa, where x is a single character and a is a (possibly empty) sub string, and let $(v,1)$ be the tree-edge that enters leaf 1. The algorithm next must find the end of string $S[2..i] = a$ in the current tree derived from I_i. The key is that node v is either the root or it is an interior node of I_i. If it is the root, then to find the end of a the algorithm just walks down the tree following the path labeled a as in the naïve algorithm. But if v is an internal node, then by Corollary 1.2 (since v was in I_i) v has a suffix link

out of it to node $s(v)$. Further, since $s(v)$ has a path-label that is a prefix of string a, the end of string a must end in the subtree of $s(v)$. Consequently, in searching for the end of a in the current tree, the algorithm need not walk down the entire path from the root, but can instead begin the walk from node $s(v)$. That is the main point of including suffix links in the algorithm.

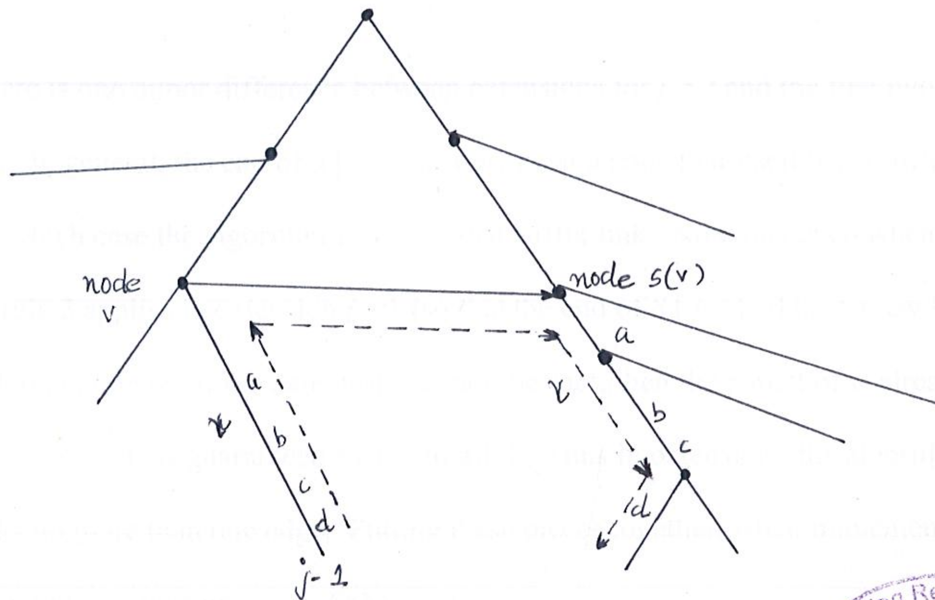


FIG 7

To describe the second extension in more detail, let y denote the edge-label on edge $(v,1)$. To find the end of a, walk up from leaf 1 to node v; follow the suffix link from v to $s(v)$; and walk from $s(v)$ down the path (which may be more than a single edge) labeled y. The end of that path is the end of a (figure 7). At the end of path a, the tree is updated following the suffix extension rules. This completely describes the first two extensions of phase $i+1$.

To extend any string $S[j..i+1]$ for $j > 2$, repeat the same general idea: starting at

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

There is one minor difference between extensions for $j > 2$ and the first two extensions. In general, the end of $S[j-1..i]$ may be at a node that itself has a suffix link from it, in which case the algorithm traverses that suffix link. Note that even when extension rule 2 applies in extension $j-1$ (so that the end of $S[j-1..i]$ is at a newly created internal node w), if the parent of w is not the root, then the parent of w already has a suffix link out of it, as guaranteed by Lemma 1.1. Thus in extension j the algorithm never walks up more than one edge. Putting these pieces together, when implemented using suffix links, extension $j \geq 2$ of phase $i+1$ is:

Single extension algorithm

Begin

1. Find the first node v at or above the end of $S[j-1..i]$ that either has a suffix link from it or is the root. This requires walking up at most one edge from the end of $S[j-1..i]$ in the current tree. Let y (possibly empty) denote the string between v and the end of $S[j-1..i]$.

2. If v is not the root, traverse the suffix link from v to node $s(v)$ and then walk down from $s(v)$ following the path for string y. If v is the root, then follow the path for $S[j..i]$ from the root (as in the naïve algorithm).

3. Using the extension rules, ensure that the string $S[j..i]S(I+1)$ is in the tree.

4. If a new internal node w was created in extension $j-1$ (by extension rule 2), then by Lemma 1.1, string a must end at node $s(w)$, the end node for the suffix link from w. Create the suffix link $(w,s(w))$ from w to $s(w)$.

End:

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

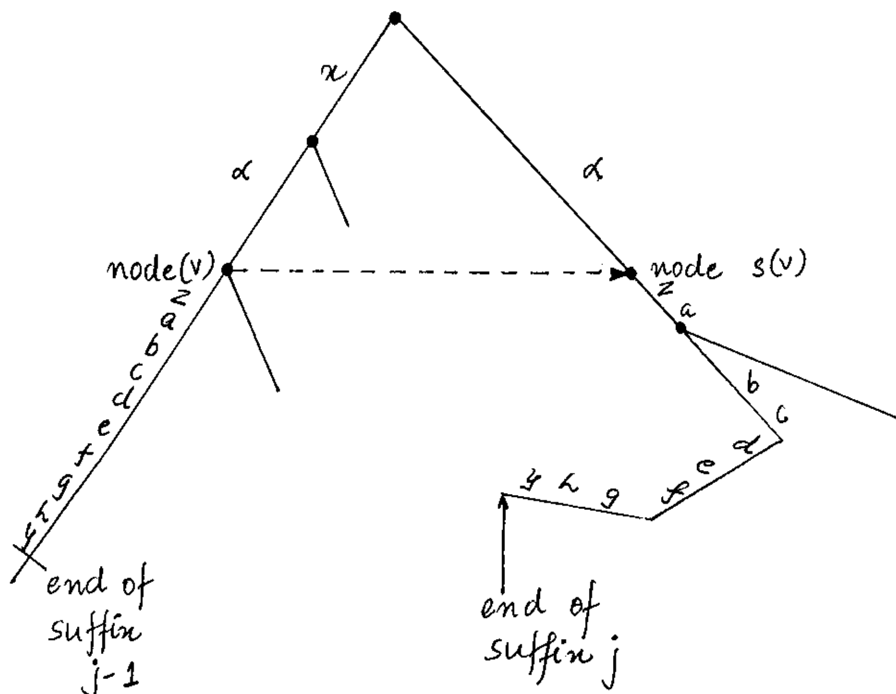


FIG 8

What has been achieved so far?

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

The answer is that as described, the use of suffix links does not yet improve the time bound. However, here we introduce a trick that will reduce the worst-case time for the algorithm to $O(m2)$.

Trick number 1: skip/count trick

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

<u>Trick 1</u> Let g denotes the length of y , and recall that no two labels of edges out of *s(v)* can start with the same character, so the first character of y must appear as first character on exactly one edge out of *s(v)*. Let $g`$ denote the number of characters on that edge. If $g`$ is less than g, then the algorithm does not need to look at any more of the characters on that edge: it simply skips to the node at the end of the edge. There it sets g to $g - g`$, sets a variable $h$ to $g` + 1$, and looks over the outgoing edges to find

the correct next edge (whose first character matches character *h* of *y*). In general, when the algorithm identifies the next edge on the path it compares the current value of *g* to the number of characters *g*` on that edge. When *g* is at least as large as *g*`, the algorithm skips to the node at the end of the edge, sets *g* – *g*`, sets *h to h* +*g*`, and finds the edge whose first character is character *h* of *y* and repeats. When an edge is reached where *g* is smaller than or equal to *g*` then the algorithm skips to character *g* on the edge and quits, assured that the *y* path from *s(v)* ends on that edge exactly *g* characters down its label.(see figure 8).

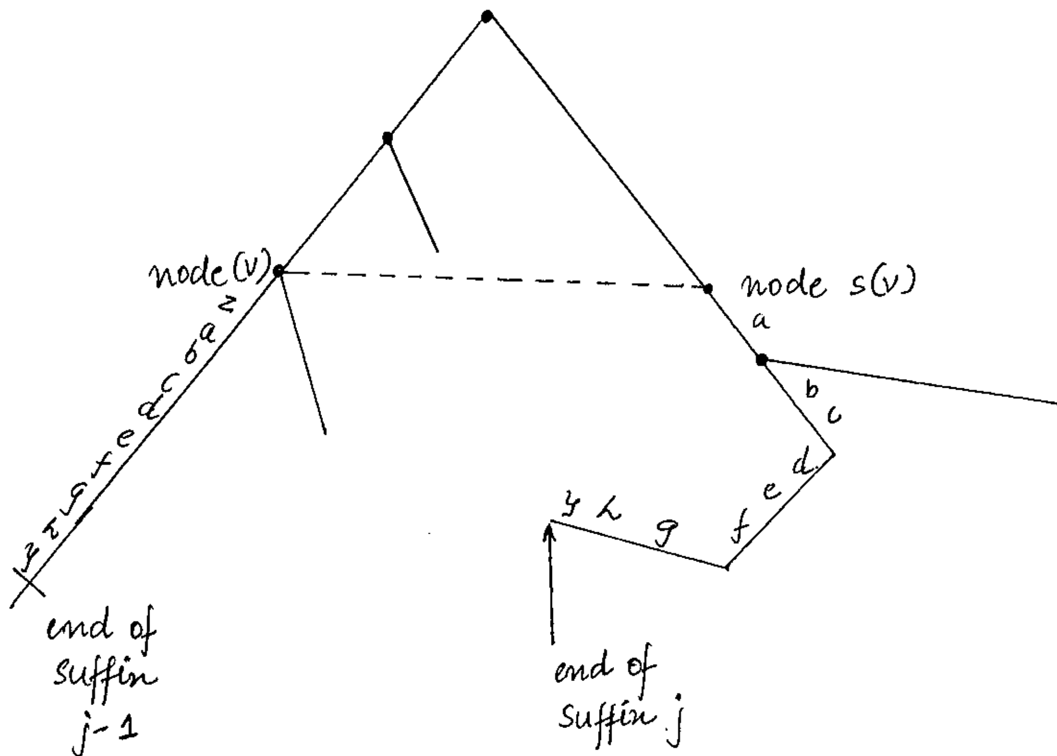


FIG 8

Assuming simple and obvious implementation details (such as knowing the number of characters on each edge, and being able, in constant time, to extract from *S* the character at any given position) the effect of using the skip/count trick is to move

from one node to the next node on the *y* path in *constant* time. The total time to traverse the path is then proportional to the number of *nodes* on it rather than the number of characters on it.

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

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

Lemma 1.2. *Let (v, (s(v)) be any suffix link traversed during Ukkonen's algorithm. At that moment, the node-depth of v is at most one greater than the node depth of s(v).*

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

ancestor for $s(v)$) is an internal ancestor whose path-label has length one (it has label x). Therefore, v can have node-depth at most one more than $s(v)$. (figure 9)

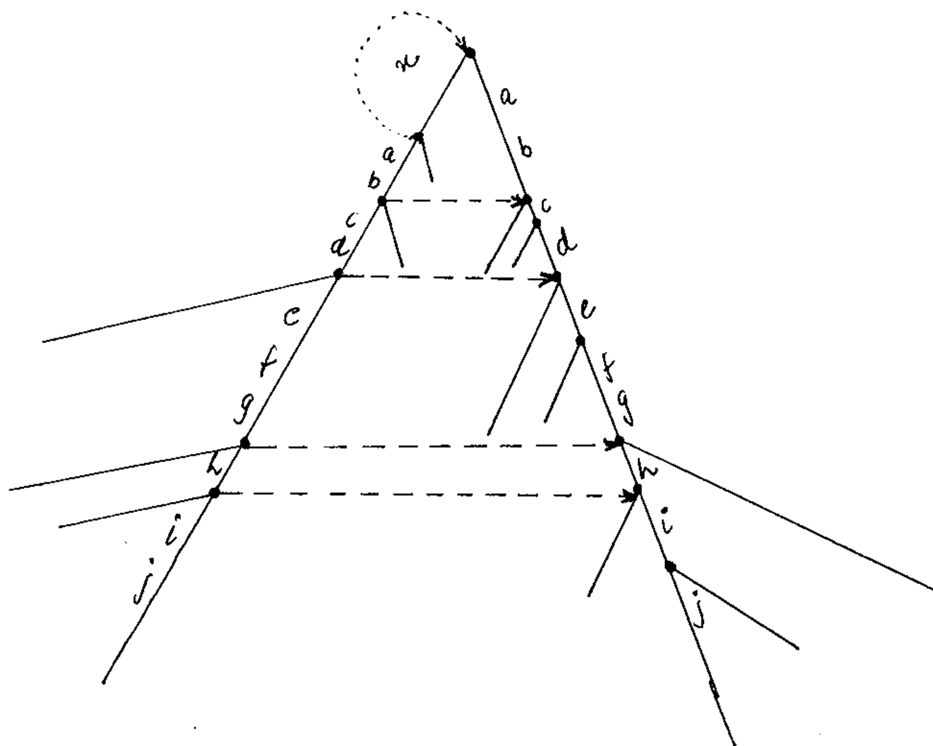


FIG 9

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

Theorem 1.1. *Using the skip/count trick, any phase of Ukkonen's algorithm takes $O(m)$ time.*

PROOF There are $i + 1 \leq m$ extension in phase i. In a single extension the algorithm walks up at most one edge to find a node with a suffix link, traverses one suffix link,

walks down some number of nodes, applies the suffix extension rules, and may be adds a suffix link. We have already established that all the operations other than the down walking take constant time per extension, so we only need to analyze the time for the down-walks. We do this by examining how the current node-depth can change over the phase.

The up-walk in any extension decreases the current node-depth by at most one (since it moves up at most one node), each suffix link traversal decreases the node-depth by at most another one (by Lemma 1.2), and each edge traversed in a down-walk moves to a node of greater node-depth. Thus over the entire phase the current node-depth is decremented at most $2m$ times, and since no node can have depth greater than m, the total possible increment to current node-depth is bounded by $3m$ over the entire phase. It follows that over the entire phase, the total number of edge traversals during down-walks is bounded by $3m$. Using the skip/count trick, the time per down-edge traversal is constant, so the total time in a phase for all the down walking is $O(m)$, and the theorem is proved.

There are m phases, so the following is immediate:

Corollary 1.3. Ukkonen's algorithm can be implemented with suffix links to run in $O(m^2)$ time.

Note that the $O(m^2)$ time bound for the algorithm was obtained by multiply the $O(m)$ time bound on a single phase by m (since there are m phases). This crude multiplication was necessary because the time analysis was directed to only a single

phase. What are needed are some changes to the implementation allowing a time analysis that crosses phase boundaries. That will be done shortly.

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

A simple implementation detail

We next establish an *O(m)* time bound for building a suffix tree. There is, however, one immediate barrier to the goal: The suffix tree require $\Phi(m^2)$ space. As described so far, the edge-labels of a suffix tree might contain more than $\Phi(m)$ characters in total. Since the time for the algorithm is at least as large as the size of its output, that many characters makes an *O(m)* time bound impossible.Consider the string = *abcdefghijklmnopqrstuvwxyz*. Every suffix begins with a distinct character; hence there are 26 edges out of the root and each is labeled with a complete suffix, requiring 26X27/2 characters in all. For strings longer than the alphabet size, some characters will repeat, but still one can construct strings of arbitrary length *m* so that the resulting edge-labels have more than $\Phi(m)$ characters in total. Thus, an *O (m)*-time algorithm for building suffix trees requires some alternate scheme to represent the edge-labels.

Edge-label compression

A simple, alternate scheme exists for edge labeling. Instead of explicitly writing a sub string on an edge of the trees, only write *a pair of indices* on the edge, specifying beginning and end positions of that sub string in *S* (see figure10). Since the algorithm has a copy of string *S*, it can locate any particular character in *S* in constant time given its position in the string. Therefore, we may describe any particular suffix tree algorithm as if edge-labels were explicit, and yet implement that algorithm with only

a constant number of symbols written on any edge (the index pair indicating the beginning and ending positions of a sub string).

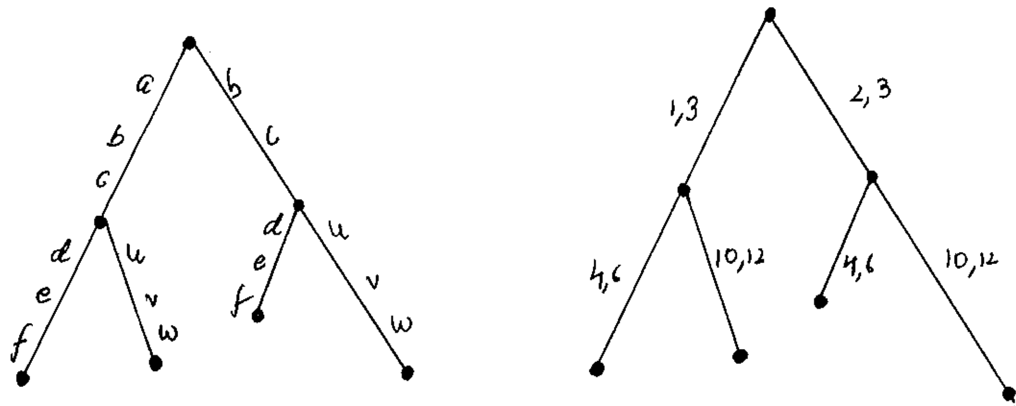


FIG 10

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

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

Two more little trick and we're done

Two more implementation tricks that come from two observations about the way the extension rules interact in successive extensions and phases. These tricks, plus Lemma 1.2, will lead immediately to the desired linear time bound.

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

When extension rule 3 applies, no work needs to be done since the suffix of interest is already in the tree. Moreover, a new suffix link needs to be added to the

tree only after an extension in which extension in which extension rule 2 applies. These facts and Observation 1 lead to the following implementation trick.

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

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

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

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

Now leaf 1 is created in phase 1, so in any phase I there is an initial sequence of consecutive extensions (starting with extension 1) where extension rule 1 or 2 applies. Let ji denote the last extension in this sequence. Since any application of rule 2 creates a new leaf, it follows from Observation 2 that $ji \leq ji{+}1$. That is, the initial

sequence of extensions where rule 1 or 2 applies cannot shrink in successive phases. This suggests an implementation trick that in phase $i+1$ avoids all explicit extensions 1 through ji. Instead, only constant time will be required to do those extensions implicitly.

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

Trick 3 In phase $i+1$, when a leaf edge is first created and would normally be labeled with sub string $S[p..i+1]$, instead of writing indices $(p,i+1)$ on the edge, write (p,e), where e is a symbol denoting "the current ene". Symbol e is a *global* index that is set to $i+1$ once in each phase. In phase $i+1$, since the algorithm knows that rule 1 will aply in extensions 1 through ji at least, it need do no additional explicit work to implement those ji extensions. Instead, it only does constant work to increment variable e, and then does explicit work for (some) extensions starting with extension $ji+1$.

1 2 3 4 5 6 7 8

8 9 10 11

11 12 13 14 15 16

16 17

Fig 10

The punch line

With Tricks 2 and 3, explicit extensions in phase $i + 1$ (using algorithm SEA) are then only required from extension $ji + 1$ until the first extension where rule 3 applies (or until extension $i + 1$ is implemented as follows:

Single phase algorithm: SPA

Begin

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

End

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

The key feature of algorithm SPA us that phase I + 2 will *begin* computing explicit extensions with extension j^*, where j^* was the *last* explicit extension computed in phase $I+1$. Therefore, two consecutive phases share *at most one* index (j^*) where an explicit extension is executed (figure 10). Moreover, phase $I+1$ ends knowing where string $S[j^*..i+1]$ ends, so the repeated extension of j^* in phase $I+2$ can execute the suffix extension rule for j^* without any up-walking, suffix link traversals, or node skipping. That means the first explicit extension in any phase only takes constant time. It is now easy to prove the main result.

Theorem 6.1.2. *Using suffix links and implementation tricks 1,2 and 3, Ukkonen's algorithm builds implicit suffix trees Ii through Im in O(m) total time.*

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

As the algorithm executes explicit extensions, consider anindex j corresponding to the explicit extension the algorithm is currently executing. Over the entire execution of the algorithm, j never decreases, but it does remain the same between two successive phases. Since there are only m phases, and j is bounded by m, the algorithm therefore executes only $2m$ explicit extensions. As established earlier, the

time for an explicit extension is a constant plus some time proportional to the number of node skips it does during the down-walk in that extension.

The bound the total number of node skips done during all the down-walks, we consider (similar to the proof of Theorem 1.1) how the current node-depth changes during successive extensions, evenextensions in different phases. The key is that the first explicit extension in any phase (after phase 1) begins with extension j^*, which was the last explicit extension in the previous phase. Therefore, the current node-depth does not change between the end of one extension and the beginning of the next. But (as detailed in the proof of Theorem 1.1), in each explicit extension the current node-depth is first reduced by at most two (up-walking one edge and traversing one suffix link), and thereafter the down-walk in that extension increases the current node-depth by one at each node skip. Since the maximum node-depth is m, and there are only $2m$ explicit extensions, it follows (as in the proof of Theorem 1.1) that the maximum number of node skips done during all the down-walking (and not just in a single phase) is bounded by $O(m)$. all work has been accounted for, and the theorem is proved.

Creating the true suffix tree

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

suffix arrays – more space reduction

When we talk about practicle implementation issues we observe that when alphabet size is included in the time and space bounds, the suffix tree for a string of length m either requires $\Phi(m|\Sigma|)$ space or the minimum of $O(m \log m)$ and $O(m \log|\Sigma|)$ time. Similarly, searching for a pattern P of length n using a suffix tree can be done in $O(n)$ time only if $\Phi(m|\Sigma|)$ space is used for the tree, or if we assume that up to $|\Sigma|$ character comparision cost only constant time. Otherwise, the search takes the minimum of $O(n \log m)$ and $O(n \log|\Sigma|)$ comparisions. For these reasons, a suffix tree may require too much space tobe practical in some applications. Hence a more space efficient approach is desired that still retains most of the advantates of searching with a suffix tree.

In the context of the substring problem where a fixed string T will be searched many times, the key issures are the time needed for the search and the space used by the fixed data structure representing T. the space used during the preprocessing of T is of less concern, although it should still be "reasonable".

Manber and Myers proposed a new data structure, called a *suffix array,* that is very space efficient and yet can be used to slove the exact matching problem or the sub string problem almost as efficiently as with a suffix tree. Suffix arrays are likely to be an important contribution to certain string problems in computational molecular biology, where the alphabet canbe large (we will discuss some of the reasons for large alphabets below). Interestingly, although the more formal notion of a suffix array and

the basic algorithms for building and using it were developed in Manber and Myers, many of the ideas were anticipated in the biological literature by Martinez.

After defining suffix arrays we show how to convert a suffix tree to a suffix array in linear time. It is important to be clear on the setting of the problem. String *T* will be held fixed for a long time, while *P* will vary. Therefore, the goal is to find a space-efficient representation for *T* (a suffix array) that will be held fixed and that facilitates search problems in *T*. However, the amount of space used during the construction of that representation is not so critical. In the exercises we consider a more space efficient way to build the representation itself.

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

That is, the suffix starting at position *Pos(1)* of *T* is the lexically smallest suffix, and in general suffix *Pos(i)* of *T* is lexically smaller than suffix *Pos(i+1)*.

An usual, we will affix a terminal symbol $ to the end of *S*, but now we interpret it to be lexically less than any other character in the alphabet. This is in contrast to its interpretation in the previous section. As an example of a suffix array, if *T* is *Mississippi*, then the suffix array *Pos* is 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3. Figure 12 lists the eleven suffixes in lexicographic order.

11: i
8: ippi
5: issippi
2: ississippi
1: mississippi
10: pi
9: ppi
7: sippi
4: sissippi
6: ssippi
3: ssissippi

FIG: 12

Notice that the suffix array holds only integers and hence contains no information about the alphabet used in string *T*. Therefore, the space required by suffix arrays is modest – for a string of length *m*, the array can be stored in exactly *m* computer words, assuming a word size of at least long *m* bits.

When augmented with an additional 2*m* values (called *Lcp* values and defined later), the suffix array can be used to find all the occurrences in *T* of a pattern *P* in *O* (*n* + log2 *m*) single-character comparision and book-keeping operations. Moreover, this bound is independent of the alphabet size. Since for most problems of interest log2 *m is O* (*n*), the substring problem is solved by using suffix arrays as efficiently as by using suffix trees.

Suffix tree to suffix array in linear time

We assume that sufficient space is available tobuild a suffix tree for *T* (this is done once during a preprocessing phase), but that the suffix tree cannot be kept intact to be used in the (many) subsequent searches for patterns in *T*. Instead, we convert the suffix tree to the more space efficient suffix array. Thus we develop an alternative, more space efficienc (but slower) method, for *building* a suffix array.

A suffix array for *T* canbe obtained from the suffix tree *T* for *T* by performing a "lexical" depth-first traversal of *T*. Once the suffix array is built, the suffix tree is discarded.

Definition. Define an edge (v, u) to be lexically less than an edge (v, w) if and only if the first character on the (v, u) edge is lexically less than the first character on (v, w). (In this application, the end of string character $ is lexically less than any other character).

Since no two edges out of *V* have labels beginning with the same character, there is a strict lexical ordering of the edges out of *v*. this ordering implies that the path from the root of *T* following the lexically smallest edge out of each encountered node leads to a leaf of *T* representing the lexically smallest suffix of *T*. More generally, a

depth-first traversal of *T* that traverses the edges out of each node *v* in their lexical order will encounter the leaves of *T* in the lexical order of the suffixes they represent. Suffix array *Pos* is therefore just the ordered list of suffix numbers encountered at the leaves of *T* during the lexical depth-first search. The suffix tree for *T* is constructed in linear time, and the traversal also takes only linear time, so we have the following:

Theorem 2.1. *The suffix array Pos for a string T of length m can be constructed in O(m) time.*

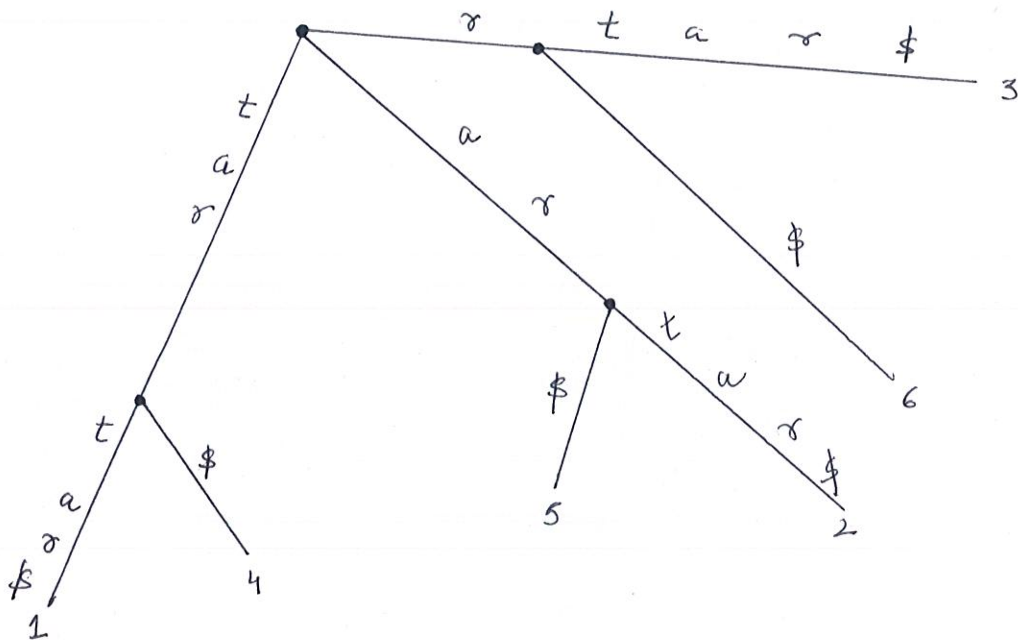


FIG 13

For example, the suffix tree for *T* = *tartar* is shown in figure 13. The lexical depth-first traversal visits the nodes in the order 5, 2, 6, 3, 4, 1, defining the values of array *Pos*.

As an implementation detail, if the branches out of each node of the tree are organized in a *sorted* linked list then the overhead to do a lexical depth-first search is the same as for any depth-first search. Every time the search must choose an edge out of a node v to traverse, it simply picks the next edge on v*'s* linked list.

FUTURE SCOPE

The Algorithm to convert Suffix trees to Suffix Arrays can be modified to convert the complex RNA s-matching algorithm(which uses suffix trees) to a simpler Algorithm which will have very less space complexity as it will be using Suffix Arrays.

CONCLUSION

Thus we conclude that for biological sequences, which generally are very long, use of Suffix Arrays is more economical as they reduce the space complexity of an algorithm remarkably and the time complexity is also not affected much.