Space Efficient Suffix Trees*

J. Ian Munro[1], *Venkatesh Raman*[2], *S. Srinivasa Rao*[2]
[1] Department of Computer Science, University of Waterloo, Canada N2L 3G1,
imunro@uwaterloo.ca
[2] The Institute of Mathematical Sciences, Chennai, India 600 113,
{vraman,ssrao}@imsc.ernet.in

Abstract

We give the first representation of a suffix tree that uses $n \lg n + O(n)$ bits of space and supports searching for a pattern string in the given text (from a fixed size alphabet) in $O(m)$ time, where n is the size of the text and m is the length of the pattern. The structure is quite simple and answers a question raised by Muthukrishnan in [22]. Previous compact representations of suffix trees had either a higher lower order term in space and had some expectation assumption [7], or required more time for searching [9]. When the size of the alphabet k is not viewed as a constant, this structure can be modified to use the same space but take $O(m \lg k)$ time for string searching or to use an additional $O(n \lg k)$ bits but take the same $O(m)$ time for searching. We then give several index structures for binary texts, with less space including

- a structure that uses a suffix array ($n\lceil \lg n \rceil$ bits) and an additional $o(n)$ bits,
- an indexing structure that takes $\frac{n}{2} \lg n + O(n)$ bits of space and
- an $o(n \lg n)$ bit structure which answers in $O(m)$ time, the decision question of whether a given pattern of length m occurs in the text.

Each of these structures uses a different technique, either in the storage scheme or in the search algorithm, in reducing the space requirement. The first one uses a suffix array, a sparse suffix tree and a table structure. Finding all the occurrences of a pattern using this structure takes $O(m+s)$ time, where s is the number of occurrences of the pattern in the text. The second structure constructs a sparse suffix tree for all the suffixes that start with the bit that occurs more number of times in the given binary text. The last structure uses an iterative algorithm to search for the pattern. This structure is the first $o(n \lg n)$ bit index to support the decision version of indexing queries in time linear in the length of the pattern. But this does not support the general indexing queries where we want to find the position of occurrence of the pattern.

Our main contribution is the development of techniques to use the succinct tree representation through balanced parentheses for suffix trees.

*A preliminary version of this paper has appeared in the proceedings of *18th Foundations of Software Technology and Theoretical Computer Science* conference, Lecture Notes in Computer Science, Springer Verlag **1530** (1998) pages 186-196.

1 Introduction and Motivation

Given a text string, full text indexing is the problem of preprocessing the text so that search for a pattern string can be done efficiently. While *inverted lists* [4, 18] and *signature files* [27] can be used for indexing texts that are structured as long sequences of words or keys, suffix trees and suffix arrays are much more directly assignable to the problem of full text search. A *suffix tree* [17] is a trie in which the leaves correspond to the suffixes of the text starting at each point in the given text. Standard representations of suffix trees for texts of length n take about $4n$ words or pointers, and the original text is retained, where each word takes $\lg n$ bits of storage; lg denotes the logarithm base 2. A search for a pattern of length m can be performed in $O(m)$ time (see Section 2 for details). A *suffix array* [16] is an array of pointers to the suffixes of the given text. This array of pointers is presented in lexicographic order of the suffixes referred to. A secondary array of longest common prefixes of some of those suffixes is used to aid in searching. This fairly standard representation of suffix array and the supporting structure takes about $2n$ words [9] and supports a search in $O(m + \lg n)$ time [16]. So in [22], Muthukrishnan asked whether there exists a data structure that uses only $n + o(n)$ words and answers indexing questions in $O(m)$ time. In this paper, we propose three such structures. The first one uses $n + O(n/\lg n)$ words, or equivalently $n \lg n + O(n)$ bits and supports string searching in $O(m)$ time. The second structure takes $n\lceil \lg n \rceil + o(n)$ bits of space and supports searching in $O(m)$ time. In these structures, all the occurrences of a pattern can be found in $O(m+s)$ time, where s is the number of occurrences of the pattern in the text. Then we give a structure that takes $\frac{n}{2} \lg n + O(n)$ bits of space and supports searching in $O(m)$ time. Then we give another structure to solve the decision problem, of finding whether a given pattern exists in the text, in $O(m)$ time using $o(n \lg n)$ bits. This structure does not support finding the position of an occurrence of the pattern, if it exists, in general.

Colussi and De Col [9] have reported a data structure that uses $n \lg n + O(n)$ bits, but a search in the structure takes $O(m + \lg \lg n)$ time. In [7], Clark and Munro gave a version of the suffix tree that uses n plus an expected $O(n \lg \lg n / \lg n)$ words under the assumption that the given binary (encoded) text is generated by a uniform symmetric random process and that the bit strings in the suffixes are independent. So our structures are not only more space efficient (in the lower order term), they require no assumption about the distribution of characters in the input string.

The starting point for our first representation is the $2n + o(n)$ bit encoding of an n node static binary tree[21]. This structure supports, in constant time, operations including move to either child or to the parent and also report the size of the subtree of a given node. We will, however, require a few more primitive navigational operations to support our suffix tree algorithms.

The next section reviews the suffix tree data structure. Section 3 reviews the succinct binary tree representation, describes algorithms to support the additional operations and explains how the binary tree representation can be used to obtain a space efficient suffix tree. In Section 4, we give a structure that uses a suffix array, a sparse suffix tree and a table structure, all taking $n\lceil \lg n \rceil + o(n)$ bits of space, which also supports indexing in $O(m)$ time. Section 5 gives a structure which takes $\frac{n}{2} \lg n + O(n)$ bits of space and answers indexing

queries in $O(m)$ time. Section 6 describes a structure to answer the decision version of the indexing queries (where given a text string and a pattern, we want to know whether the pattern occurs in the text) in $O(m)$ time, which takes $o(n \lg n)$ bits of space. Section 7 gives concluding remarks and lists some open problems.

Our model of computation is the standard unit cost RAM model where we assume that all the standard arithmetic and boolean operations on $\lg n$ bit words and reading and writing $\lg n$ bit strings can be performed in constant time. By a suffix array, we mean just an array of pointers to the suffixes of the given text in lexicographic order.

2 Suffix Trees

Suffix trees[1] are data structures that admit efficient online string searches. They have been applied to fundamental string problems such as finding the longest repeated substring [26], finding all squares or repetitions in a string [1], approximate string matching [15], and string comparison. They have also been used to address other types of problems such as text compression [24], compressing assembly code [10], inverted indices [5], and analysing genetic sequences [8].

A suffix tree for a text string x of length n is an ordered trie whose entries are all the suffixes of x. In order to ensure that each suffix has a minimal prefix that distinguishes it from the other suffixes, a special character '\$', not in Σ, is appended to x. At a leaf node we keep the starting position, in the text, of the suffix ending at that leaf. Since the sum of the lengths of all the suffixes could be $\Theta(n^2)$, the tree size (number of nodes in the tree) could also be of the same order. Various tricks have been employed to reduce the size of the tree to $O(n)$ [17, 19, 26]. One trick is to compress the nodes with single child, and store the starting position in the text, and the length of the compressed string at those nodes. Another trick is to store only the length (called the "skip value") of the compressed string at the nodes. For now, we will stick to the latter trick of keeping only the "skip value" at the compressed nodes. Since we have $n + 1$ leaf nodes (including the end-marker) and at most n internal nodes (as each internal node has at least two children), the tree has at most $2n + 1$ nodes.

Given a pattern p of length m and the suffix tree for a string x of length n, to search for p in x, we start at the root of the tree following the search string. At any node, we take the branch that matches the current character of the given pattern. At compressed nodes, we skip those many characters as specified by the "skip" value at that node, before comparing with the pattern. The search is continued until the pattern is exhausted or the current character of the pattern has no match at the current node. In the latter case, there is no copy of the pattern in the text. In the former case, if the search ends at a node, the position value stored in any leaf of the subtree rooted at the node (where the pattern is exhausted) gives a *possible* starting point of the pattern in the text. Since we skipped bits in the middle, we start at a position given by any of the leaves of the subtree rooted at the node where search has ended, and confirm if the pattern exists in the text starting from that position. Clearly the above procedure takes $O(m)$ time to check the existence of the pattern in the text.

The storage requirement of suffix trees comprises of three quantities[7]:

- the storage for the tree (trie),
- the storage for the skip values at the compressed nodes, and
- the position indices at the leaves.

The storage for the tree takes $2n \lg n$ bits if we use the usual pointer-node representation. The storage for the skip values at the compressed nodes will require at most $n\lceil \lg n \rceil$ bits. The position indices at the leaves take $(n+1)\lceil \lg n \rceil$ bits. Thus the total space requirement for the "standard representation" of a suffix tree is $4n \lg n + O(n)$ bits. In the next Section, we show how each of these components can be stored space efficiently. In particular, we give a suffix tree structure that takes $n \lg n + O(n)$ bits of space.

3 New Space Efficient Suffix Tree

In this Section, we review the succinct representation of binary trees and describe algorithms to support additional operations. Using this representation, we give a suffix tree structure that takes $n \lg n + O(n)$ bits of space.

Even if the given text is on a binary alphabet, its suffix tree will be a ternary tree (due to the extra \$ character). So first we will consider a binary alphabet by converting each symbol of the alphabet and the symbol \$ into binary. Thus, given a string x, we encode all the suffixes of $x\$$ in binary and construct a trie for them, which will be a binary tree. (This is the same as the PAT tree of Gonnet et al. [11].) This trie representing all the suffixes (in binary) of the text is a special case of a binary tree in which all the internal nodes have exactly two children. Since there are $n+1$ suffixes, there will be n internal nodes and $n+1$ external nodes in the trie.

3.1 Succinct Representation of Trees

A general rooted ordered tree on n nodes can be represented by a balanced string of $2n$ parentheses as follows: Perform a preorder traversal of the tree starting at the root. Write an open parenthesis when a node is first encountered, going down the tree, and a closing parenthesis while going up after traversing the subtree.

One can not use this procedure directly to represent a binary tree, as it is not possible to distinguish a node with a left child but no right child and one with a right child and no left child. So Munro and Raman[21] use the well known isomorphism between the class of binary trees and the class of rooted ordered trees to convert the given binary tree into a general rooted ordered tree and then represent the rooted ordered tree using the above parenthesis representation. In the ordered tree there is a root which does not correspond to any node in the binary tree. Beyond this, the left child of a node in the binary tree corresponds to the leftmost child of the corresponding node in the ordered tree, and the right child in the binary tree corresponds to the next sibling to the right in ordered tree. A node is represented, by convention, by its corresponding left parenthesis. See Figure 1. It is not obvious still, how the navigational operations can be performed in constant time. Munro and Raman[21] show that using $o(n)$ additional bits, the standard operations of *left child*, *right child* and *parent*

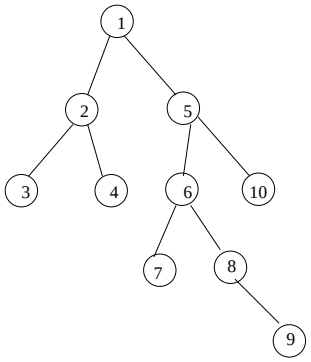


Fig 1.1: The Given Binary Tree on 10 Nodes

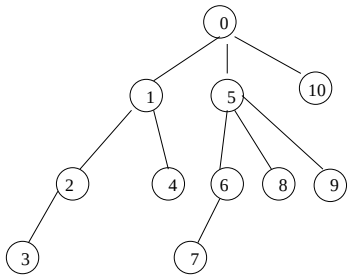


Fig 1.2: Equivalent Rooted Ordered Tree

0 1 2 3 3 2 4 4 1 5 6 7 7 6 8 8 9 9 5 10 10 0

(((( )) ( ))((( )) ( )( )) ( ))

Fig 1.3: The Parenthesis Representation

Figure 1: Binary tree representation using parenthesis sequence

of a given node can be found in constant time. They also show that given a node, the size of the subtree rooted at that node can be found in constant time.

To use this tree representation to represent a suffix tree, we need to support several additional operations in constant time. We represent a node in the binary tree by its preorder number. Let x be any node in the given binary tree. We define the following operations:

- $leafrank(x)$: return the number of leaves to the left of node x (in the preorder numbering)
- $leafselect(j)$: return the jth leaf in the left to right ordering of the leaves
- $leafsize(x)$: return the number of leaves in the subtree rooted at node x
- $leftmost(x)$: return the leftmost leaf in the subtree rooted at node x
- $rightmost(x)$: return the rightmost leaf in the subtree rooted at node x and

Beginning with Jacobson [13], much of the work on navigating succinct representations of trees [2, 3, 20, 21] has relied on the operations *rank* and *select* defined on binary strings.

- $rank(i)$: the number of 1's up to and including the position i and
- $select(i)$: the position of the ith 1.

The rank and select operations can be generalized as follows: Given a binary string of length n, and a pattern p which is a binary string of fixed length m, let $rank_p(i)$ be the number of (possibly overlapping) occurrences of pattern p up to and including the position i and $select_p(i)$ be the position of the ith occurrence of p in the given binary string. The following theorem is easy to prove.

Theorem 1: *Given a binary string of length n, and a binary pattern p of length at most $\epsilon \lg n$, where ϵ is any constant less than $1/2$, both $rank_p(i)$ and $select_p(i)$ can be supported in constant time using $o(n)$ bits, in addition to the space required for the given binary string.*

Proof: The algorithm to support the (original) *rank* operation [13, 20] uses the following basic idea: Divide the given bit string into blocks of size roughly $\lg^2 n$ each, and keep the rank information for the first element of every block. Within a block of size $\lg^2 n$ keep a recursive structure (storing the rank information with respect to the block). After a couple of levels, the block sizes are small enough that the number of distinct possible blocks is small enough to keep a precomputed table of answers in $o(n)$ bits. The precomputed table stores the number of occurrences of the pattern up to the position for each small block (of length roughly $(\lg n)/2$) and for each position in the block.

This structure can easily be adapted to keep the $rank_p$ information for every block, since the pattern length, say m, is less than the block size. The precomputed table, in this case, stores the number of occurrences for every possible triple consisting of a block b of size $(\lg n)/2$, a bit string x of size $m-1$, and a position i in the block. The table entry stores the number of occurrences of the pattern p in the prefix of length $|x|+i$ of the string xb. (This also takes care of occurrences of the pattern in a span of two consecutive blocks.) The space requirement for the table is bounded above by 2^m times the original space requirement (which is $o(n)$ since m is at most $\epsilon \lg n$, where ϵ is less than $1/2$).

The structure for computing *select* [6] uses three levels of auxiliary directories. The first auxiliary directory records the position of roughly every $(\lg n \lg\lg n)^t h$ one bit and hence requires at most $\frac{n}{\lg\lg n}$ bits. Let r be the size of a subrange between two values in the first auxiliary directory and consider the sub-directory for this range. If $r \geq (\lg n \lg\lg n)^2$ then we will explicitly store the positions of all the one bits, which requires at most $\frac{r}{\lg\lg n}$ bits. Otherwise, we re-subdivide the range and record the position, relative to the start of the range, of each $(\lg r \lg\lg n)^t h$ one bit in the second level auxiliary directory, which again takes at most $\frac{r}{\lg\lg n}$ bits.

After one more level, the block sizes will reduce to at most $(\lg\lg n)^4$. We can perform a select on a range of $(\lg\lg n)^4$ bits using a constant number of operations on regions of size $\lg n$ bits. Computing select on a small range of bits is again performed using table lookup. Let d be an integer greater than one. For each possible bit pattern of length $\frac{\lg n}{d}$ and each value i in the range $1 \ldots \frac{\lg n}{d}$ we record the position of the ith one in the bit pattern and in a separate table, the number of ones in the bit pattern. To compute select on a small range, we scan the range using the second table until we know which subrange contains the answer and use the first table to compute the answer. At most a constant number of subranges can be considered. The critical point is that we know where the appropriate directory bits at each level are located and how to interpret them based on the value of i and the preceding directory levels. The storage used for the auxiliary directories and the lookup tables is $\frac{3n}{\lceil \lg\lg n \rceil} + O(n^{1/d} \lg n \lg\lg n)$ which is $o(n)$ for $d \geq 2$.

This structure can easily be adapted to support the $select_p$ operation, as in the case of $rank_p$ operation, on the given binary string in constant time for any pattern p of length at most $\epsilon \lg n$, where ϵ is any constant less than $1/2$. □

Remark: When the pattern length m is $\omega(\lg n)$, we can find the maximum number of non-

overlapping occurrences of p up to any position (using $o(n)$ extra space) as follows: First divide the given bit string into blocks of size m. Then store with each block head (the first bit of the block), the answer for that position. Also if a pattern, that does not overlap with the previous occurrences of the pattern ends in that block, then the position where the pattern ends is also kept with the block head. There can be at most one such position in each block as the pattern length is more than the block size. Here we use the fact that the non-overlapping occurrences of the pattern obtained by starting with the leftmost occurrence of the pattern and picking the next available occurrence (in a greedy way) will give us the maximum number of occurrences.

Now, we prove the main theorem of this section.

Theorem 2: *A static binary tree on n nodes can be represented using $2n + o(n)$ bits such that given a node x, in addition to finding its parent, left child, right child and the size of the subtree rooted at node x, we can also support $leafrank(x)$, $leafselect(j)$, $leafsize(x)$, $leftmost(x)$, and $rightmost(x)$ operations in constant time.*

Proof: As before, we first convert the binary tree into an equivalent rooted ordered tree. The fact that *parent, left child, right child and the subtree size* are supported in constant time is already known [21].

Any leaf in the binary tree is a leaf in the general tree, but not vice versa. In fact, any leaf in the general tree is a leaf in the binary tree only if it is the last child of its parent. In other words, leaves in the binary tree correspond to the rightmost leaves in the general tree. In the parenthesis notation, a rightmost leaf corresponds to an open-close pair followed by a closing parenthesis. Thus to compute $leafrank(x)$ we need to find the $rank_p(x)$, where p is the pattern ()), in the parenthesis sequence corresponding to the tree. Also $leafsize(x)$ is the difference between $rank_p(x)$ and $rank_p(f(x))$ where $f(x)$ denotes the closing parenthesis corresponding to the parent of x. Similarly $leafselect(j)$ is nothing but $select_p(j)$ where p is the pattern ()). Hence from Theorem 1, these operations can be supported in constant time.

The leftmost leaf of the subtree rooted at a node in the binary tree is the leaf whose $leafrank$ is one more than the $leafrank$ of the given node. Thus it can be found using the expression: $leftmost(x) = select_p(rank_p(x) + 1)$, where p is the pattern ()), in constant time. The rightmost leaf of the subtree rooted at the node in the binary tree is the rightmost leaf of its parent in the general tree. Now the rightmost leaf of a node in the general tree is the leaf preceding the closing parenthesis of the given node. Thus $rightmost(x) = select_p(rank_p(close(parent(x)) - 1))$, where $close$ gives the position of the corresponding closing parenthesis of a given opening parenthesis, which takes constant time for evaluation (See [21] for details). □

3.2 Representing the Suffix Tree

Given a string x, we encode all the suffixes of $x\$$ in binary and construct a trie for them, which will be a binary tree on $2n + 1$ nodes. We represent this $2n + 1$ node binary trie with $4n + o(n)$ bits using the representation given in the Section 3.1. (As the external nodes are implicit and all internal nodes have two children, we could use only $2n + o(n)$ bits by

storing only the internal nodes of the tree. But listing the external nodes explicitly has some advantages for later modifications.) This will take care of the storage for the first component of suffix tree representation. Next, we show that the second component of the representation, the skip values, need not be stored explicitly and that it can be obtained online. So only the third component taking $n\lceil \lg n \rceil$ bits accounts for the higher order term and we get an $n \lg n + O(n)$ bit suffix tree structure.

We do not keep the skip values at all in our structure; we will determine them online whenever needed as detailed below.

To search for a pattern, we start at the root as before. Navigating in the suffix tree is possible in our tree representation. At each internal node, to find the skip value at a node we first go to the leftmost and rightmost leaves in the subtree rooted at that node. Then we start comparing the text starting at these positions until there is a disagreement. (We don't have to compare the suffixes from the starting position. We already know that they agree up to the portion of the string represented by the node. So we can start matching them from that position onwards.) Once we find the characters of disagreement, we find their binary encodings which may give raise to further agreement. The number of bits matched is the skip value at that node. Finding the leftmost or rightmost leaf of the subtree rooted at a node, in our tree representation takes constant time using the $leftmost(x)$ and $rightmost(x)$ operations of Theorem 2.

Now we continue the search as before. If the search terminates at a leaf node, then the pattern is compared with the suffix pointed to by the leaf to see if it matches. To find the suffix pointed to by the leaf, we first find the $leafrank$ of the leaf, and then find the value of that index in the array of pointers (suffix array). If the end of the pattern is encountered before we reach a leaf, then the suffix pointed to by a representative leaf from the subtree rooted at the node, at which the search has stopped, is compared with the pattern. This leaf can be found by the $leftmost(x)$ or $rightmost(x)$ operations of Theorem 2. The pattern matches the suffix if and only if all the suffixes in the subtree match the pattern.

If we are working over a k symbol alphabet, the time to find a skip value is $O(\lg k + the\ skip\ value)$. This is because once we find the characters where the disagreement happens, we find their binary representations and find further agreements. Now the sum of the skip values in the search is at most m. So the total time spent in figuring out skip values is only $O(m \lg k)$. Looking carefully at the calls to $rank$ and $select$, which are implicit in this approach and that of Clark and Munro[7], one can see that we have at most doubled the search cost by getting rid of the storage required for the skip values. This increase is due to the repeated $leftmost$ and $rightmost$ calls.

Once we confirm that the pattern exists in the text, the number of leaves in the subtree rooted at the node where the search ended, gives the number of occurrences of the pattern in the text. This can be found in constant time using the $leafsize$ operation of Theorem 2. Also, we can output all the occurrences in time proportional to the number of occurrences, using $rank_p$ and $select_p$ operations on the leaf nodes. Thus we have

Theorem 3: *A suffix tree for a text of length n can be represented using $n \lg n + O(n)$ bits such that given a pattern of size m, the number of occurrences of the pattern in the string can be found in $O(m \lg k)$ time where k is the size of the alphabet. Finding the positions of all*

the occurrences of the pattern requires $O(m+s)$ time, where s is the number of occurrences of the pattern in the text.

The above representation can be built in $O(n)$ time as once we build the suffix tree which takes $O(n)$ time [26], the succinct tree representation can be built in $O(n)$ time.

Benoit, Demaine, Munro and Raman[3] have extended the succinct representation for binary trees to represent an ordered k-ary tree using $2n + n\lceil \lg k \rceil + o(n)$ bits where all navigational operations, except 'finding the child labeled i' for some i, can be performed in constant time. Visiting the child labeled i, for any i, from any node in this structure takes $O(\lg\lg k/(\lg\lg n - \lg\lg k))$ time. This was later improved to $O(\lg\lg\lg k)$ in[23], where it is also shown that by using an additional $n\lceil \lg\lg k \rceil$ bits, we can support all the operations in *constant* time. Instead of converting the suffixes to binary, we can represent the k-ary suffix tree using these k-ary tree representations directly. Then we have

Theorem 4: *A suffix tree for a text of length n can be represented using $n(\lceil \lg n \rceil + \lceil \lg k \rceil + 2) + o(n)$ bits such that the number of occurrences of a given a pattern of length m, in the string can be found in $O(m \lg\lg\lg k)$ time, where k is the size of the alphabet.*

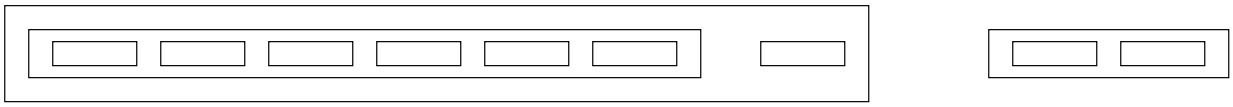
Theorem 5: *A suffix tree for a text of length n can be represented using $n(\lceil \lg n \rceil + \lceil \lg k \rceil + \lceil \lg\lg k \rceil + 2) + o(n)$ bits such that the number of occurrences of a given a pattern of length m, in the string, can be found in $O(m)$ time, where k is the size of the alphabet.*

Reporting all the occurrences of the pattern using these structures requires an additional $O(s)$ time, where s is the number of occurrences of the pattern in the text.

4 A Structure Using a Suffix Array and $o(n)$ Bits

In [9], Colussi and Col give a structure that takes $n \lg n + O(n)$ bits and takes $O(m + \lg\lg n)$ time to search for a pattern of length m. This structure has a sparse suffix tree for every $(\lg n)$th suffix and suffix arrays (with extra information about longest common prefixes to aid efficient searching) for each block of size $\lg n$ in the sorted array of suffixes of the given text. We first observe that with an additional $o(n)$ bits, this structure can be modified, as detailed below, into a structure that takes $O(m)$ time for searching. Note that the search time is $\omega(m)$ only when the pattern size m is $o(\lg\lg n)$. To take care of these small patterns, we store a precomputed table of answers. This table stores, for every string of length at most $\lg\lg n$, the positions, if any, in the suffix array of the first and the last suffix for which the string is a prefix. The difference between these two indices gives the number of occurrences of the string in the text. The table takes $O(2^{\lg\lg n} \lg n)$ or $O(\lg^2 n)$ bits. Given a pattern of length m, if $m \leq \lg\lg n$ then we look into the table and answer the query (in constant time). Otherwise we revert back to algorithm given in [9] to answer the query, which takes $O(m)$ time (since $m > \lg\lg n$). Though this gives another structure for indexing that takes $n \lg n + O(n)$ bits of space and supports searching in optimal time, the constant factor in the lower order term in space is higher than the structure given in Section 3.

In what follows we show that $o(n)$ bits are sufficient in addition to a suffix array to support searching in $O(m)$ time, in the case of a fixed size alphabet. For an arbitrary alphabet of size k, the space required will remain the same but the search time will be $O(m \lg k)$. We



001010 010110 011001 011001 100101 101010 011001 [3, 4]

Figure 2: Structure of each table entry and a typical entry in the table

will give the structure for the case of a fixed size alphabet. The general alphabet case is a straightforward extension of this structure, as explained in Section 3.

The key aspect of the structure is two levels built on top of a suffix array. First, we divide the suffix array into blocks of size b (to be fixed later) and build a suffix tree for every suffix starting at the first position in each block. This step is similar to the structures in [9, 14]. The first level consists of a suffix tree for every bth suffix in the suffix array. The second level consists of a table structure which gives the following information: given an array of b bit strings in lexicographically sorted order, each of length at most b and a pattern string of length at most b, it gives the first and last positions of the occurrences of the pattern, if it occurs, in the array. See Figure 2. There is a table entry for *every* bit string array of size b where each bit string is of size at most b and for every pattern string of length at most b. The table is stored in the lexicographic order of its entries.

The suffix tree is stored using the representation given in the previous section (without the leaf pointers). The space occupied by the tree is $O(n/b)$ bits. In the suffix tree, the leaf pointers are implicit (and can be computed using $leafrank$ operation described in Section 3). The space occupied by the table structure is at most $O(2^{b^2}2^b \lg b)$. Thus the overall space requirement for the structure (including $n\lceil \lg n \rceil$ bits for the suffix array) is $n\lceil \lg n \rceil + O(n/b) + O(2^{b^2+b} \lg b)$ bits. Thus we choose b such that b is the smallest integer with $n \leq 2^{b^2+b+\lg b+\lg\lg b}$. Note that b is $\Theta(\sqrt{\lg n})$ which makes the space complexity to be $n\lceil \lg n \rceil + O(n/\sqrt{\lg n})$ bits.

To search for a pattern string of length m, we first match the pattern in the suffix tree. If we have successfully matched the pattern in this tree, then all the nodes in the subtree rooted at the node where the search has ended, will have the pattern as a common prefix. To find the number of occurrences of the pattern, we have to find the first and last occurrences of the pattern which can be found respectively from the blocks before the leftmost leaf and after the rightmost leaf. If the pattern does not match completely, we will find the only block of the suffix array in which the pattern might occur depending on the next bit of the pattern. This block will be to the left of the leftmost leaf of the node where the search has ended if the next bit is a 0, and to the right of its rightmost leaf if this bit is a 1. Thus, in either case, we are left with (either one or two instances of) the problem of finding the first and last occurrences of the pattern in a block of length b in the suffix array.

Given an array of b strings, each of length (at most) b, in sorted order, and a pattern string of length (at most) b, the table has the beginning and ending positions of the occurrences of the pattern in the array. Now we read the first b bits of each of the suffixes in the sub-block, and the first b bits of the pattern and index into the table to find the first and last occurrences of the part of the pattern in the sub-block. If this gives a non-empty range, we will read the next b bits of each of the suffixes in this range and the next b bits of the pattern

and find the sub-range of suffixes that match the first $2b$ bits of the pattern. We will do this repeatedly until either the pattern is exhausted or the range has become empty (or a single suffix). The number of table look-up's is at most m/b and each table look-up takes $O(b)$ time. Here we use the fact that b is $O(\sqrt{\lg n})$ and that any substring of length at most $\lg n$ starting at a given position in the text can be read in constant time using *mod* and *div* operations. So the overall time to search for a pattern is $O(m)$. Thus we have

Theorem 6: *Given a text of length n from a fixed alphabet, there exists a data structure that uses $n\lceil \lg n \rceil + O(n/\sqrt{\lg n})$ bits such that given a pattern of size m, the number of occurrences of the pattern in the string can be found in $O(m)$ time. Finding all the occurrences of the pattern using this structure requires $O(m+s)$ time, where s is the number of occurrences of the pattern in the text.*

5 A Structure Using $\frac{n}{2}\lg n + O(n)$ Bits

We have presented a couple of structures using $n \lg n + O(n)$ bits of space. This section shows that we can even do better, by presenting a structure that takes at most $\frac{n}{2}\lg n + O(n)$ bits of space for a given binary string of length n and supports finding an occurrence of a pattern of length m, if it exists, in $O(m)$ time.

The main idea is to store only the suffixes starting with either a 0 bit or a 1 bit, whichever number is minimum and store some extra information to aid searching for patterns starting with the other bit.

Suppose there are more number of 0's than 1's in the given bit string $T[1 \ldots n]$. The structure consists of a sparse suffix tree for all the suffixes starting with 1. (Here, first we take all the suffixes with the end-markers, convert them into binary and then construct the suffix tree so that the resulting suffix tree is a binary tree with at most $n/2$ leaves.) We order the subtrees of a node such that the left subtree contains a leaf which has at least as many consecutive zeroes preceding its starting position as any other node in that subtree. In other words, the leftmost leaf of any node has the maximal number of consecutive zeroes preceding it among all the leaves of the subtree rooted at that node. The space occupied by the sparse suffix tree (using the representation given in Section 3) is at most $\frac{n}{2}\lg n + O(n)$ bits as there are at most $n/2$ suffixes starting with 1 (since there are more zeroes than ones).

Now given a pattern, if it starts with a 1, we can use the sparse suffix tree directly to find its occurrences. Otherwise, let the pattern be $0^l x$, where the first bit of x is 1. Search for x in the sparse suffix tree. If the search fails then the given pattern does not exist in the text. Otherwise, let v be the node at which the search has ended (i.e. the node corresponding to the string x in the sparse suffix tree) and let i be the position pointed to by the leftmost leaf of v. If $T[i-l \ldots i-1]$ is identical to 0^l, then the given pattern occurs at the position $i-l$. This takes $O(l)$ time, and hence the total time will be $O(m)$. Otherwise, there is no occurrence of the pattern in the given text T. Thus we have,

Theorem 7: *Given a binary text of length n, there exists a data structure that uses $\frac{n}{2}\lg n + O(n)$ bits of space that can be used to find an occurrence of a given pattern of length m, in $O(m)$ time.*

6 A Structure Using $o(n \lg n)$ Bits

Looking at the several variations of suffix trees, one is tempted to conjecture that $\Omega(n \lg n)$ bits are necessary to support pattern search in $O(m)$ time. In this section we disprove this conjecture for at least the decision version of the problem. We develop a structure that takes $O(n \lg n / \lg \lg n)$ bits and answers whether or not a pattern string of length m exists in the given text in $O(m)$ time [1]. The structure does not support finding such an occurrence, if it exists, in general, though in some cases it may be possible to do so.

One structure that solves this decision problem is a suffix tree without the pointers at the leaves. But in this case we have to store the skip values at the compressed nodes of the suffix tree. If we omit the skip values, then to find them online, we need the leaf pointers. Thus in either case, we need at least $n \lg n$ bits of space.

In this section, we describe a structure for binary texts, which can be easily extended to any fixed size alphabet. The structure we develop consists of a sparse suffix tree and two tables. The first table stores, for all binary strings of length at most b (to be fixed later), whether it appears as a substring in the given text string. A sparse suffix tree is constructed for every bth suffix of the given text string. We interpret the given binary text of length n as a string of length n/b over a 2^b-ary alphabet and construct a suffix tree for this string.

This suffix tree is stored using the cardinal tree representation given in [2, 3] which is a generalization of Jacobson's binary tree representation. In this representation, a cardinal tree of arity t with n nodes can be stored using $nt + o(n)$ bits and the tree navigational operations (like finding the parent or the ith child of a node) can be supported in constant time. In our case, the arity is 2^b and the number of nodes is $O(n/b)$. Thus the space required to store this suffix tree is $O(2^b n/b)$ bits. Note that by using a representation of a t-ary tree that takes $2n + n\lceil \lg t \rceil + n\lceil \lg \lg t \rceil + o(n)$ bits, we can support all the tree navigational operations in constant time[23]. But for our purposes, even the first representation taking $nt + o(n)$ bits is sufficient, since the space required for the tree is dominated by the space needed to store leaf pointers using $n \lg n/b$ bits.

The second table is indexed by a node v in the sparse suffix tree and two strings p and s of length at most b. The table stores a 1 if there exists a leaf in the subtree rooted at node v which points to a suffix such that $L_v s$ is a prefix of it and the substring p precedes that suffix (in the given text), and stores a 0 otherwise (here L_v is the string obtained by concatenating the edge labels in the path from root to the node v).

The space occupied by the first table is 2^b bits. The space occupied by the sparse suffix tree is $O(n \lg n/b)$ bits for the leaf pointers and $O(2^b n/b)$ bits for the tree representation. The space for the second table is $O(\frac{n}{b} 2^{2b})$ bits. Thus choosing b to be $\frac{1}{2} \lg \lg n$ will make the overall space occupied by the structure to be $O(n \lg n / \lg \lg n)$ bits.

To search for a given binary pattern p of length m, if its length is at most b then we can know the answer from the first table. Otherwise, we will repeat the following search procedure b times, varying i from 0 to $b-1$.

Let p_i be the prefix of p of length i and s_i be the suffix of p of length $(m - i - \lfloor \frac{m-i}{b} \rfloor b)$. Match the substring of p of length $(\lfloor \frac{m-i}{b} \rfloor b)$ starting at position i, in the suffix tree. If there is no match in the suffix tree then skip the current iteration. Otherwise, let v be the node

[1] The space bound was further improved to $O(n)$ bits in [12].

at which the match has ended. If the length of L_v is equal to $(\lfloor \frac{m-i}{b} \rfloor b)$, then find table entry corresponding to node v, prefix p_i and suffix s_i. If it is 1, output *yes* and halt; skip the current iteration, if the entry is 0. If L_v (defined earlier) has length more than $(\lfloor \frac{m-i}{b} \rfloor b)$ (this can happen when v is a compressed node), then find the $(\lfloor \frac{m-i}{b} \rfloor b + 1)^{st}$ character, say x, in L_v (a character here is an element of a 2^b sized alphabet and hence corresponds to a b bit string) and check if s_i is a prefix of x (note that s is a binary string of length at most b). If not skip the current iteration. Otherwise find table entry corresponding to node v, prefix p_i and suffix λ (the empty string). If it is 1, output *yes* and halt.

Each iteration of the above search procedure takes $O(m/b)$ time and thus the total time for searching for a pattern in this structure is $O(m)$, where m is the length of the pattern.

Theorem 8: *Given a text of length n from a fixed alphabet, there exists a data structure that uses $O(n \lg n / \lg \lg n)$ bits such that given a pattern of size m, it can be determined whether the pattern occurs in the string in $O(m)$ time.*

7 Conclusions

We have given several indexing structures including

- a suffix tree using $n\lceil \lg n \rceil + 4n + o(n)$ bits of storage,
- a structure using $n\lceil \lg n \rceil + o(n)$ bits, and
- a structure using $\frac{n}{2} \lg n + O(n)$ bits

where indexing queries can be answered in time linear in the length of the query string. The first two structures can also be used to find the number of occurrences of the pattern using no additional space. We have also given a structure that uses only $o(n \lg n)$ bits of space and answers whether a given pattern is a substring of the text in optimal $O(m)$ time. Building on our first structure and using other techniques, Grossi and Vitter[12] have developed an $O(n)$ bit index structure that can be used to search any binary pattern, (of length m) stored in $O(m/\lg n)$ words, in $o(m)$ time. Finding all the occurrences in this structure takes an additional $O(s \lg^{\epsilon} n)$ time, where s is the number of occurrences of the pattern in the text and ϵ is a fixed positive constant less than 1.

Some open problems that arise (remain) are:

- Constructing an efficient indexing structure when the given text resides in the secondary memory is the most important problem for large scale full text indexing. Unfortunately, while our structures are best suited to the situation in which the entire text resides in the main memory, the number of external memory accesses made in our structures is quite high (close to m), if the given text resides in the external memory. Indeed, this problem remains in all the standard representations of suffix trees where, an access to the text is required to find the compressed string at every compressed node. So constructing an $n \lg n + O(n)$ bit suffix tree for a text in external memory (i.e. one that uses as few external accesses as possible) is an interesting open problem.

- One obvious way to construct our suffix tree representations is to construct the usual suffix tree first and then construct the parenthesis representation of the tree from it. However this method uses more space during the construction phase than is required by the final structure. Can one avoid this problem?

- Finding all the occurrences of the pattern in $O(m + s)$ time with an index structure that takes $O(n)$ bits, where s is the number occurrences of the pattern, is still an open problem. The structure given by Grossi and Vitter [12] takes $O(m + s \lg^{\epsilon} n)$ time to find all the occurrences of the pattern, where ϵ is any fixed constant less than 1.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments.

References

[1] A. Apostolico and F. P. Preparata, Structural properties of the string statistics problem, *Journal of Computer and System Sciences* **31** (1985) 394-41.

[2] D. Benoit, "Compact tree representations", Master's Thesis, Department of Computer Science, University of Waterloo, Canada (1998).

[3] D. Benoit, E. D. Demaine, J. I. Munro and V. Raman, Representing trees of higher degree, *The Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS 99)*, LNCS 1663 (1999) 169-180.

[4] A. Blumer, J. Blumer, D. Haussler, R. McConnell and A. Ehrenfeucht, Complete inverted files for efficient text retrieval and analysis, *Journal of the ACM* **34**(3) (1987) 578-595.

[5] A. F. Cardenas, Analysis and performance of inverted data base structures, *Communications of the ACM* **18** (5) (1975) 253-263.

[6] D. Clark, "Compact Pat trees", Ph. D. Thesis, Department of Computer Science, University of Waterloo, Canada (1996).

[7] D. R. Clark and J. I. Munro, Efficient suffix trees on secondary storage, *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms* (1996) 383-391.

[8] B. Clift, D. Haussler, R. McConnell, T. D. Schneider, and G. D. Stormo, Sequence landscapes, *Nucleic Acids Research* **4** (1) (1986) 141-158.

[9] L. Colussi and Alessia De Col, A time and space efficient data structure for string searching on large texts, *Information Processing Letters* **58** (1996) 217-222.

[10] C. Fraser, A. Wendt, and E. W. Myers, Analysing and compressing assembly code, *Proceedings of the SIGPLAN Symposium on Compiler Construction* (1984) 117-121.

[11] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, New indices for text: PAT trees and PAT arrays, *Information Retrieval: Data Structures and Algorithms*, Frakes and Baeza-Yates Eds., Prentice-Hall, (1992) 66-82.

[12] R. Grossi and J. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, *Proceedings of the 32nd ACM Symposium on Theory of Computing* (2000) 397-406.

[13] G. Jacobson, Space-efficient static trees and graphs, *Proceedings of the 30th IEEE Symposium on Foundations of Computer Science* (1989) 549-554.

[14] J. Kärkkäinen and E. Ukkonen, Sparse suffix trees, *Proceedings of the Second Annual International Computing and Combinatorics Conference (COCOON 96)*, LNCS 1090 (1996) 219-230.

[15] G. M. Landau and U. Vishkin, Fast parallel and serial approximate string matching, *Journal of Algorithms* **10**(2) (1989) 157-169.

[16] U. Manber and G. Myers, Suffix arrays: a new method for on-line string searches, *SIAM Journal on Computing* **22**(5) (1993) 935-948.

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

[18] A. Moffat and J. Zobel, Self-indexing inverted files for fast text retrieval, *ACM Transactions on Information Systems* **14**(4) (1996) 349-379.

[19] D. R. Morrison, PATRICIA: practical algorithm to retrieve information coded in alphanumeric, *Journal of the ACM* **15** (1968) 514-534.

[20] J. I. Munro, Tables, *Proceedings of the 16th Foundations of Software Technology and Theoretical Computer Science conference*, LNCS 1180 (1996) 37-42.

[21] J. I. Munro and V. Raman, Succinct representation of balanced parentheses, static trees and planar graphs, *Proceedings of the IEEE Symposium on Foundations of Computer Science* (1997) 118-126.

[22] S. Muthukrishnan, Randomization in stringology, *Proceedings of the FST & TCS Preconference Workshop on Randomization*, (1997) 23-27.

[23] V. Raman and S. Srinivasa Rao, Static dictionaries supporting rank, *Proceedings of 10th International Symposium on Algorithms and Computation (ISAAC)*, LNCS 1741 (1999) 18-26.

[24] M. Rodeh, V. R. Pratt, and S. Even, Linear algorithm for data compression via string matching, *Journal of the ACM* **28**(1) (1991) 16-24.

[25] H. Shang, "Trie methods for text and spatial data structures on secondary storage", PhD Thesis, McGill University, (1995).

[26] P. Weiner, Linear pattern matching algorithm, *Proc. 14th IEEE Symposium on Switching and Automata Theory* (1973) 1-11.

[27] J. Zobel, A. Moffat and K. Ramamohanarao, Inverted files versus signature files for text indexing, *ACM Transactions on Database Systems* **23**(4) (1998) 453-490.