

Suffix Vector: Space- and Time-Efficient Alternative To Suffix Trees

Krisztián Monostori, Arkady Zaslavsky, Heinz Schmidt

School of Computer Science and Software Engineering
Monash University, Melbourne
900 Dandenong Rd, Caulfield East, VIC 3145

{Krisztian.Monostori,Arkady.Zaslavsky,Heinz.Schmidt}@infotech.monash.edu.au

Abstract

Suffix trees are versatile data structures that are used for solving many string-matching problems. One of the main arguments against widespread usage of the structure is its space requirement. This paper describes a new structure called suffix vector, which is not only better in terms of storage space but also simpler than the most efficient suffix tree representation known to date. Alternatives of storage representations are discussed and a linear-time construction algorithm is also proposed in this paper. Space requirement of the suffix vector structure is compared to the space requirement of alternative suffix tree representations. We also make a theoretical comparison on the number of operations required to run algorithms on the suffix vector.

Keywords: Algorithm, data structure, suffix tree.

1 Introduction

Suffix tree is a data structure that stores all possible suffixes of a string. It is one of the most versatile data structures in the area of string matching. Apostolico (1985) lists over 40 applications of suffix trees and Gusfield (1997) has a list of more than 20 applications. Problems that can be solved by using suffix trees include exact matching, longest common substring of two strings, matching statistics, etc. (Gusfield 1997)

The main research area of our research group is to identify similar documents on the Internet (Monostori K., Schmidt H., and Zaslavsky A. 2000). A suffix tree is built on a document and other documents are compared to that suffix tree allowing to identify overlapping chunks. Our application shows that the possible applications of suffix trees are not confined to basic string-matching or DNA-matching problems.

If we stored the suffix tree in a naive way it would occupy $O(n^2)$ space because the overall length of suffixes is $n*(n-1)/2$. Instead of storing the actual characters of an edge in the tree a start pointer and an end pointer are stored and they refer to positions in the string. Since the number of edges and nodes are proportional to n , the overall space requirement of a suffix tree is $O(n)$. There are three basic algorithms to build a suffix tree:

McCreight's (McCreight 1976), Weiner's (Weiner 1973), and Ukkonen's (Ukkonen 1995). For a unifying view of these algorithms, see Giegerich and Kurtz's paper (1997).

One of the main arguments against suffix trees is the space requirement of the structure. There has been quite a work done in this area, though most of the implementations are tailored to a specific problem. The original implementation, as it was suggested by McCreight (1976), occupies 28 bytes for each input character in the worst case. Most of the implementations that aimed to improve the space-efficiency are not as versatile as the original suffix tree. None of the following alternative structures keep suffix link information in the tree, which is heavily utilised in many algorithms including the matching statistics algorithm (Chang and Lawler 1994) that we use to compare texts:

- suffix arrays (Manber and Myers 1993) occupy 9 bytes for each input symbol
- level compressed tries (Anderson and Nilson 1993) take 12 bytes for each input character
- suffix cactuses (Kärkkäinen 1995) require 10n bytes

Kurtz (1999) proposes a data structure that requires 10.1 bytes per input character on average for a collection of 42 files. This data structure retains all features of the suffix tree including suffix links. Kurtz compares his representation to many other competing representations and finds that his implementation is the most space-efficient for the collection of 42 files he used. Later in this paper we will have a more detailed comparison of our representation to Kurtz's one.

The next section introduces the suffix vector at a high level. Two alternatives of physical representations are described in Section 3. Section 4 compares the most space-efficient suffix vector representation to Kurtz's representation (Kurtz 1999) regarding both space and time. In Section 5 we describe a linear-time construction algorithm. In Section 6 we summarise our results and look at future work.

2 Suffix Vector

Suffix vectors are proposed in this paper as an alternative representation of suffix trees and directed acyclic graphs (DAG) derived from suffix trees. The basic idea of suffix vectors is based on the observation that we waste too much space on edge indices, so we store node information aligned with the original string. Hence, edge labels can be read directly from the string. This section

Copyright © 2001, Australian Computer Society, Inc. This paper appeared at the *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia. Conferences in Research and Practice in Information Technology, Vol. 4. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

- Monostori K., Zaslavsky A., Schmidt H. Suffix Vector: Space- and Time-Efficient Alternative To Suffix Trees. Proceedings of the 25th Australasian Computer Science Conference, Monash University, Melbourne, Victoria, 28 January - 1 February, 2002. pp 157-166, 2002.

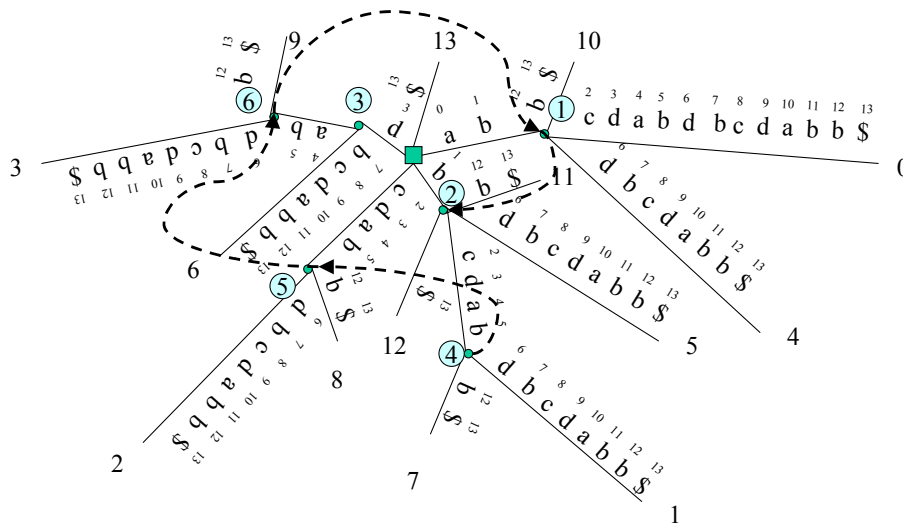


Figure 1. Suffix Tree of the example string 'abcdabdbcdabb\$'

describes the proposed alternative structure and in the following sections we analyse worst-case space requirements.

A suffix tree itself stores much of redundant information. Gusfield (1997) derives a DAG from a suffix tree, which eliminates identical subtrees. We will also show that there are other means of redundant information that can be eliminated in the suffix vector structure. Firstly we give a sample string and the suffix tree representation for that string. Let the string be $S = \text{'abcdabdbcdabb$'}$. '\$' is the unique termination symbol, which is needed, otherwise the suffix tree cannot be constructed. The suffix tree for that string is depicted in *figure 1*. Firstly we introduce a high-level suffix vector representation that is abstracted from the actual storage method. We show how the traversal of the tree works using this representation and later we show how we can efficiently represent this structure in memory. As we have already mentioned, the basic idea of our new representation is based on storing nodes and edges aligned with the string. *Figure 2* shows the new representation.

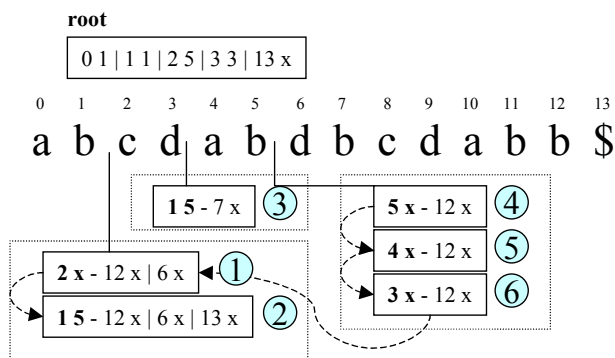


Figure 2. Suffix Vector

The root node is represented as a linked list and it shows where to start searching for a string. It has one pointer for each possible character in the string (a,b,c,d, \$). The root node looks like a root node in any other suffix tree representations. The edges store the start and end position of the string they represent. 'x' denotes a leaf edge.

Nodes in the original tree are represented as linked lists in the vector aligned with the string. For example node 3 in the original tree is represented in the box between position 3 and 4. Each node has a "natural edge", that is the continuation of the string, so the edge pointing from node 3 to node 6 is character 4 and 5 in the string. The first of the two numbers in bold represents the depth of the node that is the number of characters you encounter on the way to the given node from the root. The second number in bold gives you the position of the next node provided that you follow the natural edge. In case of node 3 if you have matched one character ('d') and you would follow by matching 'a', that is following the natural edge, the next node will be at position 5 (this is the edge pointing from node 3 to node 6). An edge has a start and an end pointer. The end pointer is depicted by x if the edge is a leaf. For some problems we need to be able to jump from one node to the next one, which means that we need to store information on the position of the next node but some problems, i.e. finding the first occurrence of a given string in another string, do not require this information, hence the end position may be omitted.

To see how the algorithm finds a string, let us follow the matching of 'dabb' in the string. We start from the root and find that we have to start at position 3. It is equivalent to analysing the edges running out of the root in the tree. After having matched 'd', we try to match 'a'. In the tree we have to check whether there is an edge starting with 'a' running out of node 5, in the suffix vector we match the next character. In this case it matches 'a' but if it did not match we should check the possible followings after having matched one character. We find this information in the first row of the box at position 3 (node 3). After 'a' we have to match 'b' on the edge in the tree and in the string in the suffix vector. They match, so we have to match the second 'b'. They do not match. We have followed 3 characters up to now, so we have to check the possible followings from here. We can see that having matched 3 characters we could follow at position 12 (node 6), that is leaf 9 in the tree. The 'x' depicts that this is a leaf node, so that is the only possible match. We have matched 4 characters up to position 12, so the start position is 9.

This example reveals another advantage of this representation that is there is no need to store leaf indices because they can be directly derived from the position: we simply have to subtract the number of characters matched up to that position.

2.1 Eliminating Redundancy in a Suffix Vector

As one can see, there is still redundant information in the representation shown in *figure 2*:

- nodes 4,5,6 are identical
- nodes 1 and 2 share some edges.

In the first case we can store only one node with the extra information that it represents nodes with depths 3 through 5 (see *figure 3*). We call these nodes *reduced nodes* because the information we have to store is reduced.

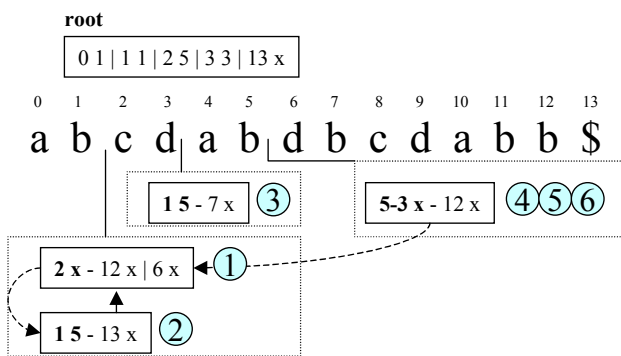


Figure 3. Eliminating Redundant Information

We can utilise the second observation by eliminating redundant edge information. It can be proven that the number of edges running out of a node monotonically increases as the node depth decreases for nodes stored at the same position. It is true because if we have x possible routes having matched y characters we will have at least x followings (the same ones) in case we have matched $y-1$ characters but others are possible. There are three cases:

- **Rule 1:** the node following in the list has as many edges as the previous one. In this case we simply set their first edge pointers to the same position.
- **Rule 2:** the node following in the list has the same edges as the previous node but it also has some extra edges. In this case the pointer of the previous node will point to the edge in the list of the second node where its own edges start.
- **Rule 3:** the node following in the list does not have the same edges as the previous node. In this case all the edges must be represented in a separate list. We call these nodes *large nodes*.

Figure 4 illustrates this concept and *figure 3* shows how it applies to our example string. The following theorem lets us to store only one suffix link for each box.

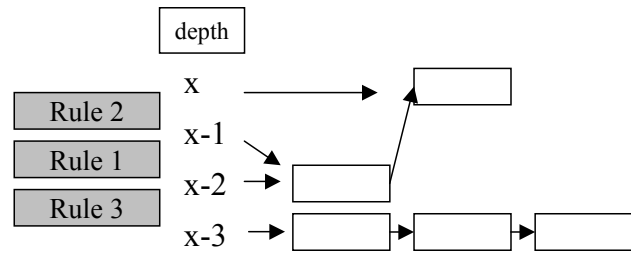


Figure 4. The Concept of Large Nodes

Theorem 1. The suffix link of a node always points to another node at the same position except for the last node (the node with the smallest node depth).

Let the label of *node v* be aw where a is a single character and w is a sequence of characters. If there is a suffix link between *node v* and *node z* then *node z* has a label w . Let x be the depth of *node v*. Then *node v* is listed at the given position (let this position be i). If *node v* is not the node with the smallest node depth then there is a *node y*, which has a node depth $x-1$ and its label is w by definition. The suffix link of *node v* points to a node with label w and there is only one such node in the suffix tree because node labels are unique. Therefore, *node v* must point to *node y*, thus the equality relation $node\ z \equiv node\ y$ follows \square .

Reduced nodes and large nodes are very important in the matching statistics algorithm (Chang and Lawler 1994). In case a reduced node is found we can save as many steps as the number of nodes represented because we do not need to analyse redundant information. In case Rule 1 applies when a suffix link is followed (note that this is the case when the suffix link is the next node in the list) we can save comparison again and when Rule 2 applies we only have to check until the beginning of the previous edge because after that all the edges have been checked in the previous step.

3 Physical Representation of a Suffix Vector

In this section we describe two alternative physical representations of a suffix vector. The first representation occupies more space but it can be directly built from a string in linear time. The second representation is the most compact physical representation but because of the actual physical structures used it cannot be directly built. However the first representation can be converted to the second one in linear time with minimal overhead. From here on we will refer to the first representation as the *general suffix vector* while the second one as the *compact suffix vector*.

In any suffix vector representation the building blocks are boxes. One box stores all the nodes represented at the same position. We have to be able to access those nodes in constant time otherwise the linearity of the algorithms run on the structure will be jeopardised. It means that in each box we have to store the number of nodes represented at that node as well as the depth of the deepest node. We also have to store a pointer to the first edge for each node and a next node value that stores the index of the next node in case the natural edge is

followed. Following from Theorem 1 we only need to store one suffix link per box.

We start with a few observations that are utilised in either or both of the representations described here.

Observation 1. The node depth of a deepest node can usually fit into 7 bits. (The reason for using 7 bits will become clear when we describe the compact representation.)

Large node depth values represent long repetitions in the string. These are very rare in English texts and also very unlikely in random texts. Representations should not limit the possible node depth values but they may allow storing this information in one byte whenever it is possible.

Observation 2. The number of nodes at a given position (in a given box) can usually fit into 7 bits.

This observation is a direct consequence of Observation 1 because we cannot have more nodes in a given box than the depth of the deepest node since nodes with the same node depths are stored at different positions.

Observation 3. The length of an edge can usually fit into 1 byte.

This observation follows the reasoning of the previous ones. Long edges mean long overlaps in the text and you cannot have many long overlaps. If you have many long overlaps it means that you have a long text, so the ratio of long edges is still small.

3.1 General Suffix Vector Representation

In this representation for each box we store the number of nodes represented in the box (4 bytes), the depth of the deepest node (4 bytes) an array of pointers to the first edges of each node (the length of the array equals to the number of nodes and each pointer is 4 bytes), an array of next node indices (same length and each index is 4 bytes) and a suffix link value. Using this representation the first edge of a given node can be accessed in constant time if we know the depth of the given node.

For storing edges we make use of Observation 3. We store the start position of the edge in 4 bytes but the 2 most significant bits have special meaning. If the first bit is set it means that the edge is a leaf and if the second bit is set it means that the next node pointer is stored in 1 byte rather than 4 bytes. It does not actually store the position of the next node but rather stores the length of the edge. We also have to store a 4-byte pointer that points to the next edge in the linked list. Using these two special bits

we can only use 30 bits to represent the start pointer, which limits the length of the string that we may be able to store. However it is not a real constraint because using 4-byte pointers we can address 4GB memory space and the suffix vector representation, as pointed out in the next section, uses at least 8 bytes per input symbol meaning that in a 2^{32} -byte memory space we can store a string with a length less than 2^{29} characters. The structure of a box in the general representation is shown in *figure 5*.

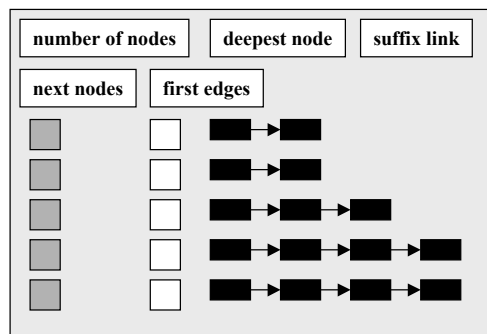


Figure 5. A Box in the General Suffix Vector

A reduced node would only store one edge-list and one next node value because they are the same for each node. In the general representation we do not make use of large nodes because they do not allow linear-time construction.

3.2 Compact Suffix Vector Representation

A compact suffix vector cannot be directly constructed in linear time but it allows storing our suffix vector in much smaller space (up to 40%). This representation is described in more details in (Monostori, Zaslavsky, and Vajk 2001).

‘Deepest node’ and ‘number of nodes’ values are stored in 1 byte whenever it is possible. The representation makes use of large nodes as well as reduced nodes. If the next node pointers fit into one byte each, only one byte is used to store them. Edges are represented in the same manner as in the case of a general suffix vector but instead of a linked list we have one byte array and a flag that indicates whether the given edge is the last edge in the list. If it is not the adjacent memory location stores the next edge. This edge representation is not dynamic, which prevents linear-time construction. The physical representation of a box is depicted in *figure 6*.

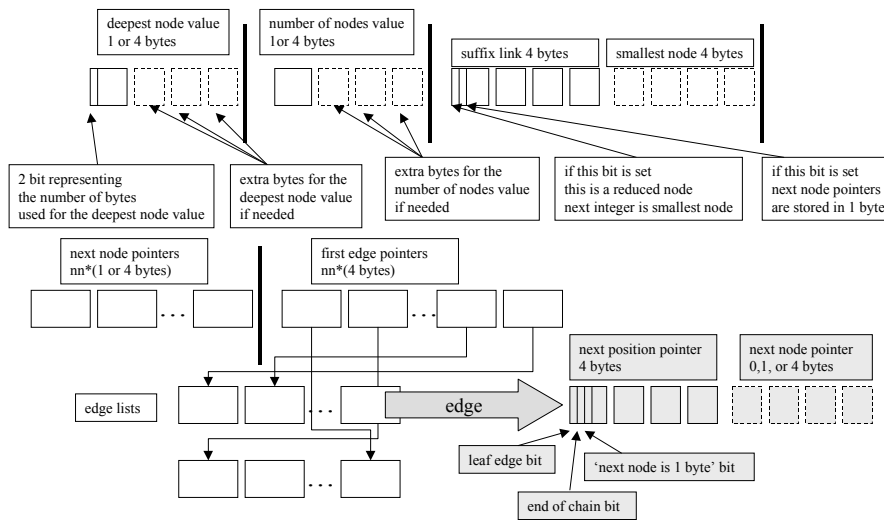


Figure 6. A Box in the Compact Suffix Vector Representation

4 Performance Analysis

The most space-efficient suffix tree representation so far has been developed by Kurtz (1999). He uses a collection of 42 files of different types to compare his representation to others. We compare our representation to his representation in this section but we only show the result to one file in each file group because of the space limits of this paper. Other files in the same group have similar results. In Kurtz's collection there are English text files (see book2 as an example), program code files (progl), and DNA sequences (K02402). We do not consider binary files because they are not commonly used in suffix tree applications. The following table shows the space requirement of our and his representation.

File name	File size	Vector size	Bytes/symbol Compact Suffix Vector	Bytes/symbol Kurtz
book2	610856	5454903	8.9299328	9.67
progl	71646	593135	8.2786897	10.22
K02402	38095	488504	12.82331	12.59

Table 1. Space-Requirement Comparison

One can see that the compact suffix vector representation performs slightly better in case of English text, significantly better in case of program code and slightly worse in case of a DNA sequence. A detailed explanation of these differences can be found in (Monostori, Zaslavsky, and Vajk 2001). The general suffix vector representation uses additional 4-5 bytes per input symbol.

Not only is the suffix vector representation better in terms of space but it also outperforms Kurtz's representation in time, that is running time of algorithms using the structures. The time-advantage is two-fold.

Firstly it is much simpler to retrieve information from the suffix vector representation because information on nodes in Kurtz's representation is scattered in memory while it is much straightforward to get the same information in either of the suffix vector representations described here. For a detailed analysis see (Monostori, Zaslavsky and Vajk 2001).

The other advantage of the suffix vector representation is that it stores less redundant information than Kurtz's representation. As described in Section 2.1 multiple checking of edges can be eliminated because of the nature suffix vectors are stored.

5 Building a Suffix Vector in Linear Time

As mentioned in Section 3 we are able to build a general suffix vector in linear time, which can then be converted to a compact representation in linear time. In Subsection 5.1 we describe a linear-time algorithm that builds a general suffix vector and Subsection 5.2 will describe how to convert a general suffix vector into a compact suffix vector.

5.1 Linear-Time Construction of a General Suffix Vector

Since a suffix vector is an alternative representation of a suffix tree we have developed an algorithm that is based on Ukkonen's (1995) linear-time suffix-tree construction algorithm. Every step in Ukkonen's algorithm has a corresponding step in our construction algorithm though building a general suffix vector is more complicated because we have to deal with reduced nodes and the fact that we store multiple nodes in one box makes things more complicated.

Ukkonen's algorithm builds the tree from left to right, which means that each edge label has the least possible indices. In phase i the algorithm makes sure that the implicit suffix tree T_i is complete by inserting suffixes $S[0..i]$ through $S[i..i]$. Ukkonen's algorithm in this form would not run in linear time. In each phase we only have to make a limited number of insertions. If we find that $S[j..i]$ is already present in T_{i-1} we can conclude that all suffixes from $S[j+1..i]$ through $S[i..i]$ are already present. With the automatic extensions of leaves we only have to start inserting the suffixes in phase $i+1$ at j . The steps spent on reaching the required positions between extensions adds up to not more than $3n$ (n is the number of characters in the string) by using suffix links. The above is only a rough sketch of Ukkonen's algorithm. For

detailed descriptions see (Ukkonen 1995 or Gusfield 1997).

We have already proven in Section 2.1 that suffix links point to nodes in the same box except for the suffix link of the node with the smallest node depth. Suffix links are updated during the construction algorithm. Once a node is created its suffix link must be directed to the next node created during the algorithm. These updates can also be performed in our representation.

In order for the construction algorithm to run in linear time we have to make sure that the size of a box is known when it is created because if a box needs to be extended it means that the information that is stored in the box must be copied to a new location in memory. The amount of data to be copied is proportional to the number of nodes stored in those boxes. In case we need to extend a box some time during the algorithm we have to make sure that the overall amount of data copied is proportional to the number of characters in the string for the entire algorithm.

A box may be extended in two directions. Let us suppose that currently we have x nodes at a position i and the depth of the deepest node is y . Of course, $x \leq y$. Let us suppose that we have to add a node with depth z . Either $z \leq y-x$ or $z > y$. We will show that the second case is possible only within one phase and that in the first case if $z-y > 1$ then all other nodes between z and y will also be added to the box in the same phase. The proof relies on the following observation.

Observation 4. Only one node may be represented at a given position with a given depth. In other words it is impossible to have two different nodes with the same depth represented in the same box.

Let us suppose that we have two nodes x and y with depth d represented at position i . Their incoming edges are the same because that is why they are stored at the same position. It also means that they are descendants of the same node with the same edge labels, which contradicts the suffix tree definition. Here we have to note that we utilise the fact that each node is stored at the smallest possible position.

Theorem 2. Once a phase is over and a box has been created it is impossible that a box must be extended downwards that is adding a node with less depth than the shallowest node.

Let us suppose that in phase i in the first extension j we have to create a node at position x with depth d . It means that $S[i-d..i]$ was not found in the vector but $S[i-d..i-1]$ could be found at $S[x-(d-1)..x]$. We have to create a node at x with one edge labelled (i, end) where ‘end’ means that this is a leaf edge. The phase is not over because the actual substring was not found.

In the next extension we have to make sure that $S[i-d+1..i]$ is in the vector. We know that $S[i-d+1..i-1]$ is in the vector. There are two possibilities: it is at either $S[x-(d-2)..x]$ or some other place say $S[z-(d-2)..z]$. If it is found at the same position we have to create a node with depth $d-1$ at x with an edge labelled (i, end) . If it is found at some different position z , it is sure that $z < x$ because each node is stored at the smallest possible position. If we

do not have a node with depth $d-1$ at z we have to create one. The case when there is a box there but not a node with the given depth is handled in Theorem 3. By the end of this extension we will surely have a node with depth $d-1$ at z and we know that $S[x-(d-2)..x]=S[z-(d-2)..z]$, thus any subsequent extension in any subsequent phase would create a node at z rather than x .

It is possible that we keep inserting nodes at x until the end of the phase. A phase can finish in two ways. It is finished because either $S[j_{last}..i]$ is found in the vector or $S[i..i]$ is inserted at the root. In the second case we have a box at x , which stores nodes with depth d through l , so it is obvious that no more nodes can be added below the shallowest node. In the first case it is definitely not found at $x+1$ because if it had been found at $x+1$ $S[j..i]$ would have been found at $x+1$, too. If it has not been found at $x+1$ it means that at some extension we jumped to another *node* z , which leads us back to the previous case \square .

Theorem 3. If we extend a box upwards that is adding a node with greater depth than the deepest node at the given position, by the end of the phase we will have a continuous range of nodes from the deepest node through ‘deepest node’ – ‘number of nodes’ + 1.

Let us suppose that at position x we have a box where the deepest node is d and in phase i at extension j we have to add a node with depth $d+e$. It means that the first occurrence of $S[i-(d+e)..i-1]$ is at $S[x-(d+e)+1..x]$. We also know that the first occurrence of $S[x-d+1..x]$ finishes at x otherwise we would not have a node with depth d at position x . In order to have a continuous range of nodes within the box we have to make sure that in this phase extensions $j+1$ through $j+e$ will also create a node at position x . There are two reasons not to create a node at x for these extensions: either the phase finishes before $j+e$ or the first occurrence of $S[i-(d+k)..i-1]$ is at position $z < x$ for some $0 < k < e$. Since in the first case each extension tries to find an edge with label $S[i]$ running out from x and there are no nodes with the given depth there the only possibility is the natural edge, that is $S[x+1]=S[i]$. If this was true we would not have started extension j because $S[j..i]=S[i-(d+e)..i]=S[x-(d+e)..x+1]$ would have held. It means that the phase cannot finish between extensions j and $j+e$. The second case is also impossible because we know that the first occurrence of $S[x-d..x]$ finishes at x and the first occurrence of $S[x-(d+e)..x]$ finishes at x , which means that all substrings between the two will finish at x \square .

We have shown that a box can only be extended upwards, so we have to prove that the overall number of steps involved in copying those nodes are proportional to the total number of characters. We may also need to copy nodes when a reduced node becomes a normal node but it involves creating nodes that have not been created yet only because we could store them at one place. Thus these copying activities create new nodes, which are included in the linear time-bound of Ukkonen’s algorithm.

Before proving that node extensions do not violate the linear running time of the algorithm we prove the

following corollary that is the result of the previous theorems.

Corollary 1. Every node starts off as a reduced node, the only possible difference between the nodes are next node positions.

As we have already shown in Theorem 3 we always get a continuous range of nodes. If a new node has to be created it means that a new edge running out of that node has to be created, too. This edge will be the same for each node at the given position: (i, end) . The next node pointer belonging to a given depth is determined by the next node pointer of the edge that was split in this extension. This may vary from depth to depth \square .

In our algorithm whenever a new box needs to be created we start with a reduced node and we split that reduced node only if it is necessary. It may become necessary because a new edge is added to some intermediate node or the next node pointer must be changed for some intermediate node. If any changes need to be made to the deepest node we can apply those changes to the reduced node because we are certain that those changes will need to be propagated to the rest of the nodes later in that same phase. The only thing we have to be careful with is that when we add some extra information to a reduced node, in the next extension we will try to add that same information to some intermediate node. We have to flag that this new information is only there because of the nature of the reduced node. It means that though the actual extension does not need to be done but it does not stop the phase as it would stop once an extension is not necessary.

Our rule is that whenever some extra information needs to be added to an intermediate node of a reduced node we split up that box and create a normal box with the given number of nodes. It may happen that later on that extra information will also appear at all nodes in which case it could become a reduced node again. We make sure that these reduced nodes are identified in the conversion phase when we convert our general representation to the compact representation. The conversion algorithm is described in the following subsection.

There is only one bit missing of proving that our algorithm is linear in time. We still have not proven that the copying steps involved in box extensions are proportional to the length of the string. We know from Theorem 2 that a node cannot be extended downwards. It means that the only possible extension is when we have a box with a deepest node of depth d , and a new node needs to be created with depth $d+z$ where $z>0$. The following theorem formalises these ideas.

Theorem 4. During the general suffix vector construction algorithm the overall number of steps l involved in copying nodes when a box with deepest node of depth d at position i must be extended because a new node with depth $d+z$ ($z>0$) must be added is proportional to the length n of string S the suffix vector is built on.

We suppose that we have a finite alphabet, which is a fair assumption considering that we are talking about characters stored in computer memory. If we have a finite

alphabet copying one node from one memory location to another can be done in constant time and the constant is determined by the size of the alphabet. Given this fact we have to prove that the number of nodes to be copied during the entire algorithm is proportional to n .

Here we prove that each action of copying a node can be associated with a unique leaf. Since the number of leaves are equal to the number of characters in the string it will make sure that no more than n (the length of the string) copying will be carried out during the course of construction. In the proof we use both the tree and the vector representation for demonstration, whichever is more expressive.

Let us suppose that we are in phase i at extension j , that is $S[j..i]$ must be added to the tree and we are currently in a position in the vector (tree), which has a label of $S[j..i-1]$. In *figure 7* the new node is depicted with a lighter colour and the details of the paths running to nodes are depicted by a single line. In the worst case we already have a $i-1-j+1$ nodes at our current position f in the vector. It means that $i-1-j+1$ nodes must be copied in this extension. We associate the copying of node $i-1-j+1$ with leaf $j+1$ ($j+1$ is the first character in the path to the node), node $i-1-j$ with $j+2$, etc. *Figure 7* depicts that in the tree it will mean that on the path running out of node labelled $S[j+1..i]$ we have to create a leaf with label $j+1$. It is created either in the next extension in the same phase or it is found in the vector (tree), but in some later phase it still has to be created because each position in the string must have an associated leaf in the tree.

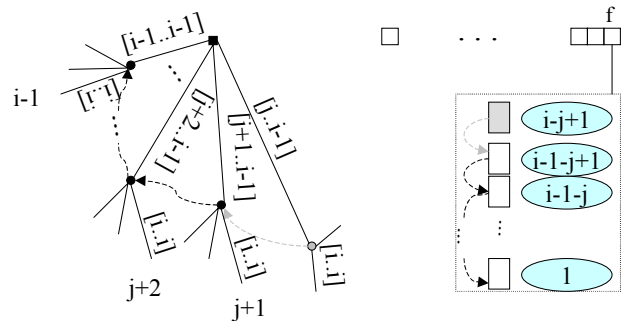


Figure 7. Associating copying activities with leaves

It may be clear from this mapping that every leaf has only one copying action associated with it but we give a formal proof below. Let us suppose that copying of two different nodes denoted by K_x and H_y , respectively, have the same leaf j associated with them (see *figure 8*). The leaf associated with copying a node is always in the subtree of the node, so both K_x and H_y must be on the path to leaf j . It means that at some stage we had to include $S[j-1..i]$ in the tree and $S[j-1..k]$ in the tree. Without the loss of generality we may assume that $i<k$, that is node H_y is deeper than node K_x . From Ukkonen's construction algorithm we know that it is only possible in two different phases. But if they are inserted in two different phases it means that $S[j-1..i]$ was found in the tree at phase i , extension $j-1$, that is it could not have possibly generated the copying of K_x , which contradicts with our initial assumption that it is associated with leaf j . Note that this proof also proves the situation when $K_x \equiv H_y$.

because the same node cannot be copied multiple times in the same extension.

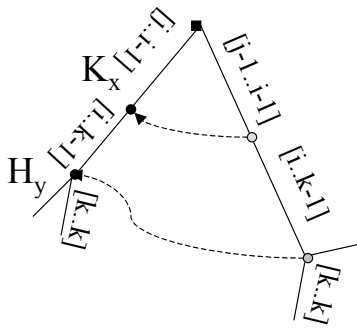


Figure 8. Different copying activities have different leaves associated with them

5.2 Converting a General Suffix Vector Into a Compact Suffix Vector

There are a few differences between the two representations. These differences are discussed in this subsection and a conversion method is described for each difference that converts the general representation to the compact representation. In order to make sure that the conversion algorithm works in linear time we prove that none of the nodes or edges are examined more than a constant number of times and none of the edges or nodes are copied more than once.

Edges are stored as linked lists in the general suffix vector while they are stored as fixed length byte-arrays in the compact representation. The conversion between the two is straightforward. We have to go through the linked list in one sweep to calculate the overall size of the byte array that stores the edges. A memory area must be allocated for the array and the edge information must be filled in.

The depth of the deepest node in the box and the number of nodes stored in the box can occupy 1 or 4 bytes each. The first bit of the first byte of the box object is reserved to flag whether these values are stored in 1 byte (the remaining 7 bits of the first byte) or in 4 bytes. This can be retrieved directly from the actual deepest node value in the general representation.

The next piece of information to be stored is the suffix link. The first two bits have special meaning. If the first bit is set this is a reduced node, which means that it represents more than one node but the information for all of those nodes are identical. Reduced nodes are flagged by a bit in the general representation, so this information can be retrieved directly. The second bit flags if all next node values can be stored in 1 byte rather than 4 bytes. If we go through all next node values stored in the given box we can set this flag accordingly. If this is not a reduced node we set the next node values by going through the next node values in the general representation. In case this is a reduced node the next piece of information to be stored is the depth of the node with the smallest depth. This information can directly be retrieved from the general representation. In case of a

reduced node there is only one next node value which can be set in the same way as described earlier.

The computationally most expensive part of the conversion is to define large nodes and set first edge pointers accordingly. The first node, the deepest one, is going to be a large node. For a notion of large nodes see Section 2.1. The way the general suffix tree is constructed ensures that if two consecutive nodes have common edges (edges with the same start and end pointer) they appear in the same order. If the difference between two edges is only some extra edges they can be represented in the same list of edges by setting the pointers accordingly (Rule 2 in Section 2.1). Of course, if all their edges are identical they can use the same list of edges, too. Comparing two edge-lists can be done in constant time because identical edges appear in the same order. We start comparing the edges of the deepest node with the node one less deep. If they can be represented in the same edge-list (Rule 1 or Rule 2) we compare the next node to the node one less deep than the deepest node. We continue until Rule 3 applies or we reach the shallowest node in the box. If Rule 3 applies at node with depth x we can create one edge-list for nodes down to depth $x+1$. The lower the node is in the list, that is the less deep the node is, the more edges it will have. It means that first we have to store those edges that appear in node with depth $x+1$ but not appear in node with depth $x+2$. Again this can be done in constant time because the order of the edges. These steps are repeated until we reach the node with the largest depth value.

From here on node with depth x will act as the deepest node and the steps described above are repeated. By the time we reach the shallowest node all first edge pointers will be set properly. *Figure 9* shows what the edge-lists look like by using large nodes.

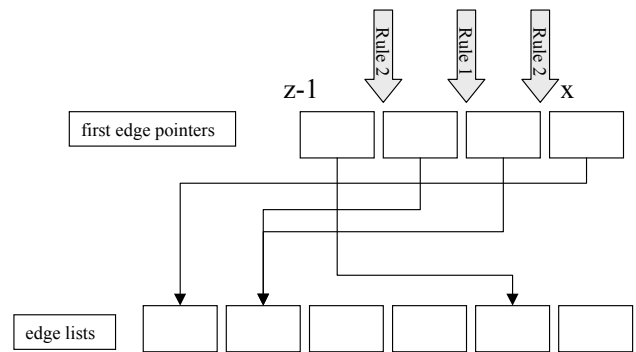


Figure 9. Utilising the Concept of Large Nodes

It is possible that by the end of this step we find a reduced node that was not originally flagged as reduced node. Two conditions must be met. Firstly all next node pointers must be equal and the deepest node must be the only large node and Rule 1 must apply between all consecutive nodes. If these criteria are met we can declare this node as a reduced node and change data accordingly.

The overhead of the conversion algorithm is very low. When general box information is copied the old data-space can be freed immediately. When next node pointers are copied the old space of next node pointers can immediately be freed, too. When copying edge-lists we

might need to store one common edge-list between the two representations, other edge-lists are stored in either of the representations but not both of them.

The following high-level pseudo code summarises the conversion algorithm:

```
For each box that is not empty
  Fill in box values
  x = deepest node
  While x is in Box
    d = x
    While d in Box and d is not large
      Decrement d
    End While
    Set Edges for x through d+1
    x = d
  End While
End For
```

6 Conclusion and Future Work

In this paper we have presented a new data structure called suffix vector, which is an alternative data structure for suffix trees. Suffix vectors store edge data aligned with the string, thus saving space over suffix tree representations. Two physical storage methods have been proposed: general suffix vector and compact suffix vector. Compact suffix vector uses the least possible space and it is more space-efficient on average than the best known suffix tree representation known to date. Not only is it more space-efficient but algorithms run faster on a suffix vector firstly because it is a simpler structure secondly because it ignores redundant edges and nodes.

A compact suffix tree cannot be constructed from a string in linear time because of the data structures used in the representation. A general suffix tree occupies approximately additional 4-5 bytes per input symbol but it can be constructed in linear time. The conversion between the two representations is done in linear time, too.

In the future we will experiment with different algorithms run on a suffix tree and a suffix vector and we will compare the two representations by means of the computational intensity of those algorithms both theoretically and practically.

We use suffix vectors in document-comparison applications where documents are compared in parallel on a cluster of workstations. Different parallel approaches will be examined in the future to find an optimal distribution of a suffix vector on nodes.

Acknowledgement

Support from Distributed Systems Technology Centre (DSTC Pty Ltd) for this project is thankfully acknowledged.

7 References

- ANDERSON A. and NILSON S. (1993): Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters*, 46:295-300.
- APOSTOLICO A. (1985): The Myriad Virtues of Subword Trees. In *Combinatorial Algorithms on Words*. 85-96. A. APOSTOLICO AND Z.GALLI. Springer-Verlag Heidelberg)
- CHANG W.I. and LAWLER. E.L. (1994): Sublinear Approximate String Matching and Biological Applications. *Algorithmica* 12:327-344.
- GIEGERICH R. and KURTZ S. (1997): From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica* 19:331-353.
- GUSFIELD D. (1997): *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Cambridge University Press.
- KÄRKKÄINEN J. (1995): Suffix Cactus: A Cross between Suffix Tree and Suffix Array. In *Proceedings of the Annual Symposium on combinatorial Pattern Matching (CPM'95)*, 191-204 LNCS 937.
- KURTZ S.(1999): Reducing the Space Requirement of Suffix Trees. *Software-Practice and Experience*, 29(13): 1149-1171.
- MANBER U. and MYERS E.W. (1993): Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing* 22(5):935-948.
- MCCREIGHT. E. M. (1976): A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM* 23(2):262-272.
- MONOSTORI K., SCHMIDT H., and ZASLAVSKY A. (2000): Document Overlap Detection System for Distributed Digital Libraries. In *Proceedings of the Fifth ACM Conferences on Digital Libraries (DL00)*, 2-7 June, 2000 in San Antonio, Texas, USA. 226-227.
- MONOSTORI K., ZASLAVSKY A., and VAJK I. (2001): Suffix Vector: A Space-Efficient Suffix Tree Representation. In *the Proceedings of the International Symposium on Algorithms and Computation, Christchurch, New Zealand, Dec 19-21*. To Appear.
- UKKONEN. E. (1995): On-Line Construction of Suffix Trees. *Algorithmica* 14:249-260.
- WEINER P. (1973): Linear Pattern Matching Algorithms. *Proceedings of the 14th IEEE Annual Symposium on switching and Automata Theory, The University of Iowa, 15-17 October*. 1-11.