

Suffix Vector: A Space-Efficient Suffix Tree Representation

Krisztián Monostori¹, Arkady Zaslavsky¹, and István Vajk²

¹ School of Computer Science and Software Engineering, Monash University, Melbourne
900 Dandendong Rd, Caulfield East VIC3145, Australia
{Krisztian.Monostori, Arkady.Zaslavsky}@csse.monash.edu.au

² Department of Automation and Applied Informatics,
Budapest University of Technology and Economics
1111 Budapest, Goldmann György tér 3. IV.em.433., Hungary
vajk@aut.bme.hu

Abstract. This paper introduces a new way of representing suffix trees. The basic idea behind the representation is that we are storing the nodes of the tree along with the string itself, thus edge labels can directly be read from the string. The new representation occupies less space than the best-known representation to date in case of English text and program files, though it requires slightly more space in case of DNA sequences. We also believe that our representation is clearer and thus implementing algorithms on it is easier. We also show that our representation is not only better in terms of space but it is also faster to retrieve information from the tree. We theoretically compare the running time of the matching statistics algorithm on both representations.

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 [2] lists over 40 applications of suffix trees and Gusfield [5] has a list of more than 20 applications.

Our main research area is identifying overlapping documents on the Internet [10] and we use a suffix tree structure to find the exact matching chunks among documents. Our application also shows that the applications of suffix trees are not limited to DNA-matching and basic string-matching problems but they can also be applied in other areas.

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 we store a start pointer and an end pointer. 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 [9], Weiner's [12], and Ukkonen's [11]. For a unifying view of these algorithms, see Giegerich and Kurtz's paper [4].

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 imple-

mentations are tailored to a specific problem. The original implementation, as it was suggested by McCreight [9], 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 the matching statistics [3] algorithm that we use to compare texts:

- suffix arrays [8] occupy 9 bytes for each input symbol
- level compressed tries [1] take 12 bytes for each input character
- suffix cactuses [6] require $10n$ bytes

Kurtz [7] 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.

In the next section we introduce suffix vectors: we show the basic idea abstracted from the actual implementation. In Section 3 we show how the new representation can be stored efficiently. Section 4 compares our representation to Kurtz's representation in terms of space and usability. Section 5 gives a high-level description of how the suffix vector can be constructed from a suffix tree. In Section 6 we summarize 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 describes the proposed alternative structure and in later sections we analyse worst-case space requirements.

First 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.

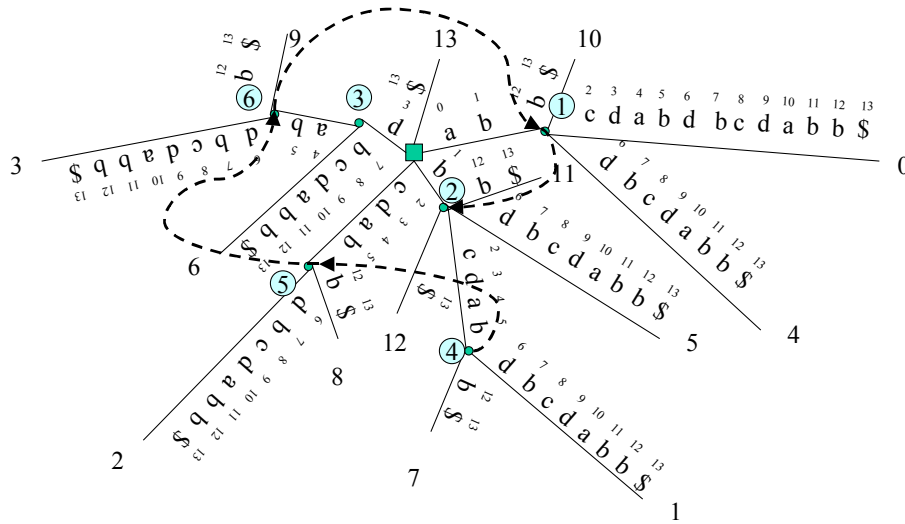


Figure 1. Suffix Tree of 'abcdabdbcdabb\$'

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,\$). 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 by 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 number in bold is the depth of the node, so in case of node 3, ‘7 x’ means that after matching one character (position 3 ‘d’) we can either follow the string itself (this is the edge pointing from node 3 to node 6), or we can jump to position 7 (this is leaf 6). The ‘x’ means that if we jump to position 7 there are no more nodes, that is a leaf. The second number in bold (5) says that if we reached this position after matching one character (‘d’) and we would follow matching ‘a’ (the “natural edge”), the next node is at position 5 (between 5 and 6). In the original tree the next node is 6, which is depicted by the third row of the box between 5 and 6. We need to be able to jump from one node to the next one for some algorithms. There are situations, which do not require this information. For example if we only need to find one occurrence of a pattern in the string we can find it without this information.

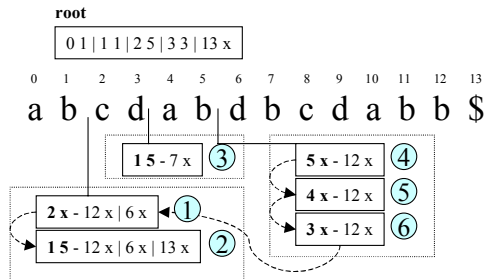


Figure 2. Suffix Vector

As one can see, each node has one corresponding row in one of the boxes. Node 1 is the first row in the box at position 1, node 2 is the second row in the box at position 1, node 3 (as discussed earlier) is the only row in the box at position 3, node 4 is the first row in the box at position 5, node 5 is the second row in the box at position 5, node 6 is the third row in the box at position 5. Every node is stored at the smallest possible index that is at the first occurrence of the string running into that node.

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 3, 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. 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, 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.

3 Space-Efficient Representation of a Suffix Vector

A naive representation of the suffix vector would store a pointer at each position in the string. These pointers may or may not be filled in. Each pointer would point to a box structure that may store multiple nodes at each position. We store the depth of the deepest node and the number of nodes in each box. We can have a pointer each to an array of next node pointers, suffix links, and first edges. Knowing the actual depth of the node we can calculate its position in the array from the deepest node value. The deepest node is stored at position 0. The edges may be represented as linked lists. Obviously this storage method is far from ideal. In the following subsection we propose a more space-efficient representation. In Section 3.2 we introduce the concept of large nodes and reduced nodes, which allow further space reduction.

3.1 Suffix Vector Physical Representation

We start with a few observations and then present the implementation details that utilise those observations.

Observation 1. The node depth of a deepest node can usually fit into 7 bits.

Large node depth values represent long repetitions in the string. These are very rare in English texts and also very unlikely in random texts. The representation does not limit the node depth but it stores it in one byte whenever it is possible. We use 7 bits because in the actual implementation the first bit is used for some other purposes.

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.

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 ■.

Figure 3 depicts the representation that utilises the observations and the theorem above.

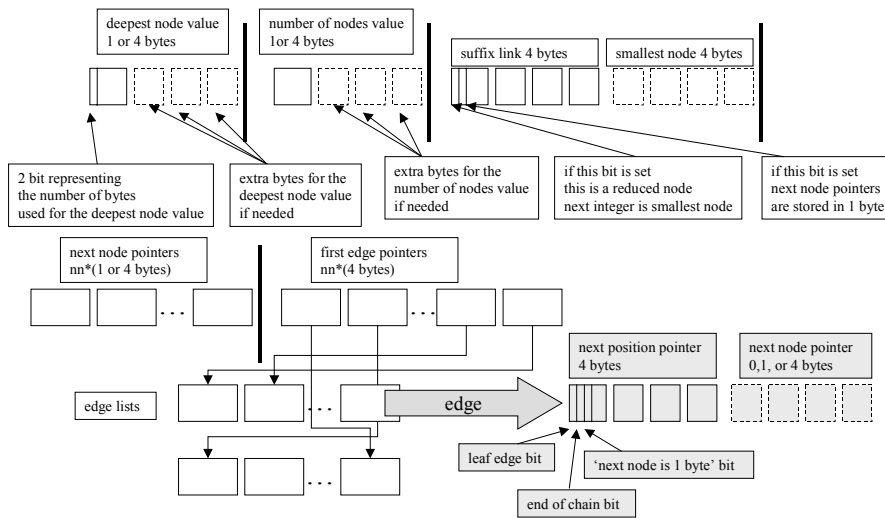


Figure 3. Space-efficient storage of a suffix vector

In each box we have to store three pieces of information that characterize the box rather than individual nodes, so this information must be stored once per box. The first value that we store is the deepest node value representing the depth of the deepest node stored at this position. From Observation 1 we know that the depth of the deepest node is usually very small, so storing it constantly in 4 bytes is a waste of storage space. We use the first bit to denote the number of bytes we need to store the deepest

node value (1 or 4 bytes). Let us denote this value by l . The best case for us is when the depth is under 128 because then it fits into the first byte (note that the first bit is used to flag the length of the field). It is very rare that chunks greater than 128 characters are repeated in any text. The number of nodes value uses the same number of bytes (l) based on Observation 2. It is possible that the number of nodes value is less than the deepest node value, thus it fits into one byte when the deepest node value does not fit into one byte but using another bit to flag this situation would unnecessarily complicate retrieval of data and would only save space rarely.

The next piece of information stored in the box is the suffix link value. From Theorem 1 it follows that every box needs to store at most one suffix link value. If the number of nodes value equals to the deepest node value it means that the depth of shortest node is one character. One-character-deep nodes do not need to store a suffix link. In this case it is not necessary to store a suffix link for the shortest node but we store one anyway because we use the first bit of the suffix link to flag whether this is a reduced node and the second bit to flag whether we have small next node pointers (1 byte) or large next node pointers (4 bytes). Reduced nodes are discussed later. For reduced nodes we need one more integer beside the suffix link, which tells us the depth of the shortest node. If the first bit is not set the suffix link is stored in one integer (the offset of the start position of the suffix link is 2^*l). If the second bit is set it means that all next node pointers can be stored in 1 byte, so the following pointers for next nodes occupy one byte. Let s be 1 if this is a reduced node, so we know that an extra integer is used for the smallest node and let n denote the length of the fields used for storing next node pointers (1 or 4 bytes).

The next thing that we are storing is the next node pointers. Next node pointers point to the node following from this position. Note that depending on the depth of the node stored at this position the next node value may vary, thus we have to store a next node pointer for each node represented at this position. From Observation 3 we know that edges are usually short, which means that we can save space by storing the length of an edge rather than the actual position of the next node. From the length of the edge we can calculate the actual position. As you will see in the ‘Performance Analysis’ section it is very rare to have edges longer than 256 characters, so we only consider two cases. If all the edges are short then next node pointers are stored in one byte, if any of the edges is long next node pointers are stored in 4 bytes. In the array we have as many pointers as the number of nodes and the size of the pointers depends on the size of edges as discussed above.

The next piece of information that we have to store in the array is the pointers to the list of first edges. The first of these pointers is located $(2^*l+4+s^*4+n^*number_of_nodes)$ bytes from the start position of the box. These are physical pointers to a given memory address. We need as many pointers as the number of nodes stored at this position. A pointer points to the address space where the list of edges running out of that node is stored. It is possible that a pointer points to an area and another pointer addresses edges within that area. These cases will be discussed in detail in the following subsection.

Each first edge pointer points to an area where the list of edges is stored. In the following we will discuss how a list of edges is represented. Each edge must store a next position pointer, which tells the next position in the string where we can follow the

matching of a pattern. We store this information in an integer (4 bytes). The 3 most significant bits of this value are saved for some additional information. The first bit flags whether this edge is a leaf or an intermediate edge. If it is a leaf there is no need to store a next node pointer, so in this case the edge is stored in one integer. The next bit flags whether this edge is the last one in the list or there are more edges to follow. Using this technique we do not need to store edges as a linked list connected by pointers, rather we can have a fix array and we check for each edge whether this is the last edge in the list. If it is not we know that the next integer stores another edge. The third bit flags the number of bytes used to store the next node pointer. We follow the same reasoning that we followed in case of the next node pointers for the box. Edges are usually short, so if they are shorter than 256 we store the length of the edge in one byte if they are longer then we store them in an integer (4bytes). The difference here is that we can decide for each next node pointer whether we need one or four bytes but in case of the next node pointers for the box it is determined for the whole array. By using this technique we can always determine the address of the next edge in the list in constant time from the first 3 bits or we learn that this is the last edge in the list. Let l be 0 if this is a leaf and 1 if it is not a leaf. Let n denote the number of bytes used for the next node pointer for the given edge. The number of bytes needed to store the given edge can be calculated using the following formula: $4+l*n$.

3.2 Reduced Nodes and Large Nodes

There are two observations we can make when we analyse the suffix vector of figure 2.

- the nodes at position 5 contain the same information
- some edges at node 1 have the same information

We can store the information of only one node at position 5, thus reducing the number of nodes we have to store. In order to retain memory integrity we set the number of nodes value to 1 at these nodes. It means that we have to store the actual smallest node value somewhere else in the data structure. It is stored after the suffix link (see section 3.1).

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. 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. Reduced nodes and large nodes are very important in the matching statistics algorithm [3]. 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 is applied 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.

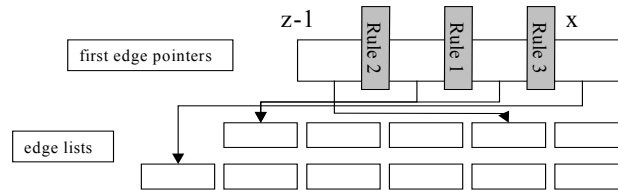


Figure 4. Large nodes

4 Performance Analysis

The most space-efficient suffix tree representation so far has been developed by Kurtz [7]. 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 tables show the space requirement of our and his representation as well as some statistical data about our representation that explain why our representation is better in some situations and why his representation is better than ours in other cases.

File name:	File size	Vector size	Bytes/symbol 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. Comparison of the total space requirement

Table 1 shows the total space requirement of the test files. For English texts our representation gives better results. The difference varies between 0.11 and 0.74 bytes per symbol. For program source code files our representation is significantly better than Kurtz's. The difference varies between 0.67 and 1.95 bytes per symbol. For K02402, which is a DNA sequence Kurtz's representation is better than ours by 0.23 bytes per symbol.

File name:	File size	Number of nodes	Number of large nodes	Number of reduced nodes	Number of boxes	Number of edges in the tree	Number of edges in the vector	Number of long edges - vector	Number of long depths
book2	610856	328825	81238	27928	85812	939682	505567	0	220
progl	71646	46512	6693	2242	6911	118159	66321	742	201
K02402	38095	24364	16574	1422	12706	62460	55614	0	0

Table 2. Statistical Data on Suffix Vectors

In table 2 we show some statistical data that explains why our representation is better in some cases and why Kurtz's representation is better in other cases. Firstly, let us analyse that part of the data that supports our observations. Observation 1 says that the node depth of the deepest node can usually be stored in one byte. If you consider the last column of the table you can see that this assumption is very true in practice. Observation 2 follows from Observation 1.

The results also support Observation 3. The number of long edges is 0 for most files. The largest number of long edges can be found in progl, which means that this file contains long overlaps. This is a program source code, so it is not a surprise that it has long overlaps within itself. The ratio of long edges is still small.

The main advantage of our algorithm is that redundant information is eliminated. If you compare the number of edges in the tree to the number of edges in the suffix vector you can see how many edge-representations can be saved by this technique. The number of reduced nodes also tells us that many nodes represent the same information and they are only stored once in our representation. The number of large nodes value is some way related to the number of edges value because the less the number of large nodes we have the less edges that we have to represent. Note that only edges of large nodes are explicitly represented.

Now we explain why we get much better results for program code. As you can see from the data in the table for program source code we have many nodes represented at each position (compare the number of nodes value to the number of boxes value). These nodes share some information and give a chance for longer sequence of small nodes.

For the DNA sequence we can see that we hardly save any edges and the number of large nodes are very close to the total number of nodes. It means that we have to represent most of the nodes and most of the edges. DNA sequences have more complicated suffix tree structures than natural English texts or program source codes. This complex structure can be slightly more efficiently represented by Kurtz's implementation.

5 Retrieving Information from the Suffix Vector

The efficiency of storing the tree is only one issue that we have to consider. Retrieving information from the tree is at least as important as the space requirement of the representation. In this section we compare the number of operations needed to retrieve the

information on nodes and edges in both representations. There are three basic operations on a suffix tree:

- getting the first edge running out of a node
- getting the next edge from the current edge in the list of edges
- following a suffix link

We divide operations into three categories:

- **Masking.** It is when we have a value (an integer or one byte) that stores multiple pieces of information and we have to mask some bits to retrieve the information we need. We denote the masking operation by **M**.
- **Comparison.** It is when we have to compare two values (two integers or two bytes) and based on the result we choose different execution paths. We denote the comparison operation by **C**.
- **Addition (Subtraction).** It is when we have to add (or subtract) two values. We denote the addition (subtraction) operation by **A**.

Due to space limitations here we are unable to analyse all operations, so we only give the actual number of steps needed in both representations for all operations. These figures may be verified by actually analysing each step. For each operation we give both the worst-case and best-case scenarios.

	Kurtz's representation		Suffix Vector	
	Worst case	Best case	Worst case	Best case
first edge	10M 5C 8A	6M 3C 2A	7M 5C 4A	6M 4C 3A
next edge	6M 4C 5A	1M 1C	5M 5C 2A	1C
suffix link	Alphabet size dependent	1C 1A	1C 1A	1C 1A

Table 3. Number of Steps Required to Obtain Information

Table 3 shows that both worst-case and best-case data are better in case of the suffix vector representation except for one situation. The best case of getting the first edge is better in case of Kurtz's representation. The worst case of getting the suffix link information in Kurtz's representation is when the suffix link is stored at the end of the edge list. It is only necessary if there is a node, whose depth is greater than $2^{10}-1$. This is a very rare situation. Neglecting this scenario the worst case is 3 maskings and 2 additions.

6 Building a Suffix Vector from a Suffix Tree

In this section we show that a suffix vector can be built from a suffix tree. It is not clear yet whether a direct construction algorithm can be used to build suffix vectors. Suffix vectors may still have their application, for example in case of our document overlap program. Firstly, the suffix vector representation uses fewer steps to retrieve the same information (see previous section). We have also analysed the number of steps saved by not analysing redundant information (see section 3.2), which showed that over 20% of the steps could be saved by using suffix vectors.

Before we give a high-level algorithm let us define some basic rules. Our first observation is that a suffix tree built by Ukkonen's algorithm will always have the lowest possible edge labels (note that edges are labelled by the beginning and end position). It is true because in the case of a given substring appearing multiple times in the string it will be inserted into the tree when the first occurrence is encountered (the tree is built from left to right) and later occurrences will already be in the tree, so they will not be added. When a node has to be represented in the vector it is inserted into the position that is defined by the end position of the edge running into the node. Let us define node-depth as the number of characters followed from the root to the node (note that this definition is the same as Kurtz's [7] but it is different from Gusfield's node-depth definition [5]).

The problem we have to face is that we do not want to recreate those arrays that contain next node pointers and point to the first edges. It would be good to know how many nodes we will have at a given position. This can be found in $O(n)$ time by a depth first search of the tree. If a node is found we have two options. This is the first node at a given position: a new box object is created in the position identified by the edge running into that node, the node counter is set to 1, and the deepest node value is set to the node-depth of the given node. Other information in the box is not filled at this stage. The other option is that the box object has been created in some previous step and in this case we only update its information. By the end of the traversal each box object will have the correct value for the number of nodes, and the depth of the deepest node. In the same run large nodes and reduced nodes can also be identified. With another $O(n)$ run we can go through the vector and create the actual boxes when every node is actually inserted: its position in the array is defined by `deepest_node_depth-current_node_depth` and the list of the edges is created. The next node values of edges are simply the difference between the end position and start position value of the edge. Note that we store the difference rather than the actual end position.

7 Conclusion and Future Work

In this paper we have proposed an alternative data structure for representing and storing suffix trees. We compared this representation to the best-known representation to date. We have found that the suffix vector representation is more space-efficient in case of semi-structured textual documents and program code while it is slightly worse in case of DNA sequences. This structure similarly to Kurtz's representation retains the versatility of suffix trees.

Future work will include analysis of how our representation can directly be constructed from scratch in $O(n)$ time. Our data structure, at this stage, has the advantage of simplicity over Kurtz's representation besides space efficiency, which is a very important argument. This advantage is demonstrated in the number of steps needed to obtain information from the tree. Our representation also eliminates some redundant information of a tree, which also saves some time on algorithms that run on the proposed data structure.

Acknowledgement

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

References

1. Anderson A., Nilson S. Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters*, 46:295-300, 1993.
2. Apostolico A. (1985). The Myriad Virtues of Subword Trees, in A. Apostolico and Z.Galli. *Combinatorial Algorithms on Words*. (Springer-Verlag Heidelberg) pp. 85-96.
3. Chang W.I., Lawler. E.L. Sublinear Approximate String Matching and Biological Applications. *Algorithmica* 12. pp. 327-344, 1994.
4. Giegerich R., Kurtz S. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, 19:331-353, 1997.
5. Gusfield D. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Cambridge University Press, 1997.
6. Kärkkäinen J. Suffix Cactus: A Cross between Suffix Tree and Suffix Array. In Proceedings of the Annual Symposium on combinatorial Pattern Matching (CPM'95), LNCS 937, pages 191-204, 1995.
7. Kurtz S.: Reducing the Space Requirement of Suffix Trees. *Software-Practice and Experience*, 29(13), pages 1149-1171, 1999.
8. Manber U., Myers E.W. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935-948, 1993.
9. McCreight. E. M. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262-272, 1976.
10. Monostori K., Schmidt H., Zaslavsky A. Document Overlap Detection System for Distributed Digital Libraries. *ACM Digital Libraries 2000 (DL00)*, 2-7 June, 2000 in San Antonio, Texas, USA.
11. Ukkonen. E. On-Line Construction of Suffix Trees. *Algorithmica* 14. pp 249-260, 1995
12. Weiner P. Linear Pattern Matching Algorithms. *Proceedings of the 14th IEEE Annual Symposium on switching and Automata Theory*, p1-11, The University of Iowa, 15-17 October, 1973.