

Constructing Chromosome Scale Suffix Trees

Dr A.L. Brown

School of Computer Science
University of Adelaide
South Australia 5005, Australia
fred@cs.adelaide.edu.au

Abstract

Suffix trees have been the focus of significant research interest as they permit very efficient solutions to a range of string and sequence searching problems. Given a suffix tree that encodes a particular string, it is possible to solve problems such as searching for a specific pattern in time proportional to the length of the pattern rather than the length of the string. Suffix trees can also support inexact matching by dramatically improving the performance of dynamic programming. Therefore, suffix trees may enable a number of large scale bioinformatics problems to be solved more efficiently than previously thought. However, these benefits presume that a suffix tree of sufficient scale can be constructed. An inherent difficulty in suffix tree construction is that the tree construction requires a semi random walk over the tree as it is constructed. Therefore very large trees that will not fit in memory could take an unacceptably long time to construct due to excessive page faulting. In this paper we present a linear time construction algorithm that can construct suffix trees larger than memory using discrete sub-trees. The sub-trees can be constructed in parallel. The performance of the algorithm is evaluated using suffix trees constructed for chromosomes 1 and 12 of the human genome.

Keywords: algorithms, parallel algorithms, pattern matching, suffix trees, chromosomes, human genome.

1 Introduction

Given a string or sequence of symbols it is possible to construct an index by combining all possible suffixes of the string or sequence in a searchable data structure. The suffixes of a string are all the contiguous substrings of the original that end with the last symbol. If a special symbol that does not occur anywhere in the string is appended to the end, then all suffixes are unique. If all the suffixes are entered into a trie and the shared structure compressed we have a data structure known as a suffix tree (Weiner 1973, McCreight 1976). For example, consider the string "xabxab". If we add a unique symbol to the end, \$, then this consists of 7 unique suffixes: "xabxab\$", "abxab\$", "bxab\$", "xab\$", "ab\$", "b\$" and "\$". If we enter these

into a trie and remove redundant nodes we get the following suffix tree.

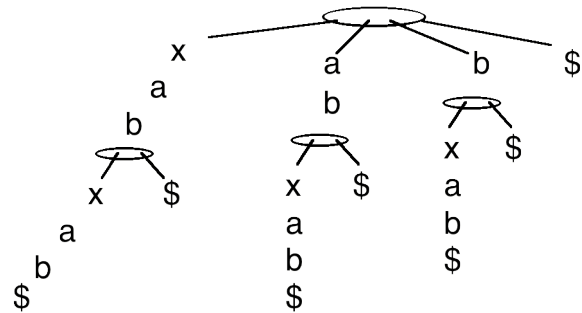


Figure 1: Suffix tree for the string "xabxab\$"

In recent years suffix trees have been the focus of significant research interest as they may permit very efficient solutions to a range of string and sequence searching problems (Gusfield 1997). Given a suffix tree that encodes a particular string, it is possible to solve problems such as searching for a specific pattern in time proportional to the length of the pattern rather than the length of the string. Every pattern present in a string is necessarily at the start of one of the suffixes and therefore starts at the root of the suffix tree. Suffix trees can also support inexact matching by dramatically improving the performance of dynamic programming (Hunt 2003). However, these benefits presume that the initial construction of a suffix tree of the required scale is possible. Gusfield (1997) presents a good introduction to the myriad of uses for suffix trees and also introduces alternative approaches to their construction.

An inherent difficulty in suffix tree construction is that the construction requires a semi random walk over the tree as it is constructed. Therefore large trees that will not fit in memory could take an unacceptably long time to construct due to excessive page faulting (Baeza-Yates and Navarro 2000). One solution that has appeared effective is to use a naive algorithm with effectively $O(N \log N)$ performance to construct discrete sub-trees and thereby achieve the locality necessary to avoid problems with virtual memory (Hunt, Atkinson and Irving 2002). However, this approach discards some internal structure that is essential for some suffix tree based algorithms. In this paper we present an algorithm that adapts McCreight's linear time construction algorithm (McCreight 1976) so that it can also construct suffix trees larger than memory using discrete sub-trees whilst retaining the internal tree structure. The performance of these approaches is compared by constructing suffix trees for chromosomes 1 and 12 of the human genome.

2 Suffix Tree Structure

Suffix trees exhibit a number of interesting structural properties that can be used to achieve both efficient construction and compact sizes (Kurtz 1999). For example, consider the suffix tree from Figure 1. It consists of a root node, three internal nodes and seven leaf nodes. All nodes except for the root node have a single incoming edge labelled by part of a suffix. If this suffix tree is constructed by entering each suffix in turn, from longest to shortest, the following can be observed.

The first three suffixes entered do not overlap and are represented by leaf nodes directly connected to the root, see Figure 2.

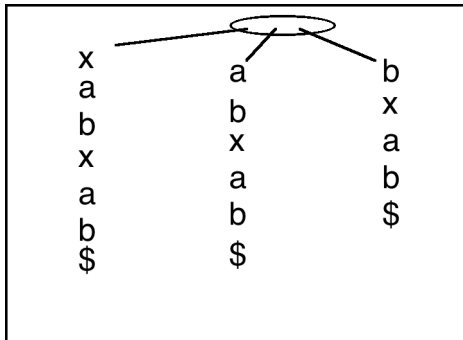


Figure 2: Before adding suffix "ab\$"

The next suffix to be entered "xab\$" shares the first 3 characters of the suffix "xabxab\$" resulting in that edge being split by the addition of an internal node, see Figure 3.

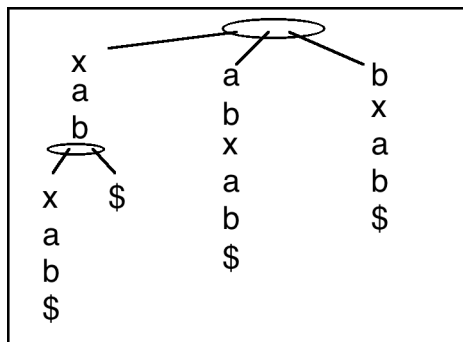


Figure 3: After adding suffix "ab\$"

The next suffix "ab\$" similarly splits the suffix "abxab\$", Figure 4, and the following one "b\$" splits "b\$abc\$", Figure 5. The final suffix "\$" forms a new leaf node attached to the root giving the final tree shown in Figure 1.

The construction algorithm implied by the previous paragraph is to enter each suffix into a suffix tree in order, from longest to shortest. To add a suffix we walk from the tree root towards the leaf nodes matching the characters of the suffix to an existing path in the tree. Since we have a unique symbol terminating the original string, all suffixes also must be unique. Therefore the traversal of the suffix tree will eventually reach a point where the next character of an existing path differs from the next character in the suffix. For example, when entering the suffix "xab\$" we traversed an existing path as far as "xab" before finding a difference. At this point we need to add a new branch to the suffix tree that

encodes the remainder of the suffix being entered. In this example we split the edge "xabxab\$" into the edge "xab" leading to a new internal node with leaf nodes labelled "xab\$" and "\$". This transformed Figure 2 into Figure 3. In those cases where the difference occurs at an existing internal node, the new branch becomes a new child for that node.

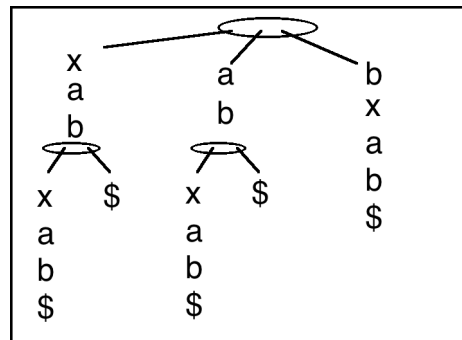


Figure 4: Adding suffix "ab\$"

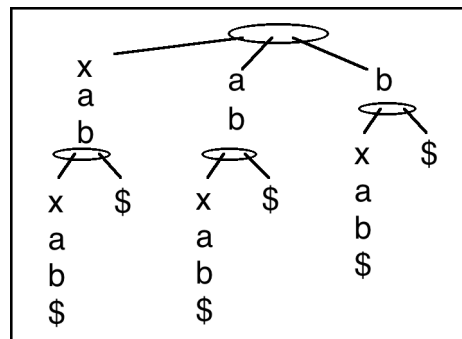


Figure 5: Adding suffix "b\$"

This suffix insertion algorithm works regardless of the order in which suffixes are entered into the tree. Furthermore, the apparent $O(N^2)$ performance is usually nearer $O(N \log N)$ in practice. It is also amenable to a simple but significant optimisation if the suffixes are entered in order from longest to shortest. The key to inserting a suffix is to identify where a new branch must be added. For a new branch to be added, the path prior to the branching point must already be present in the suffix tree. Consider our example when inserting the suffix "xab\$", the path "xab" was traversed before reaching the branching point, that is, it was already present. If we have inserted the suffixes in order then we also know that the path "ab" is already in the suffix tree. Therefore, when we attempt to add the suffix after "xab\$" which is "ab\$" then there will be a new branching node 2 characters along the existing path "ab".

In a more complex example where \$ is not the last symbol, we could find that another suffix starting with "ab\$" has already been added. If this happens then the next suffix will cause a new branch at a deeper part of the tree. However, the optimisation still works because the first part of the path already exists and we are able to traverse it more quickly. In general the optimisation allows subsequent suffixes to be added more quickly since not all characters on an existing path must be checked, we need only check the first character of each edge on the path.

the suffix tree are also contiguous substrings of the original and can be represented by start and end indices into the original string. This enables suffix trees to be represented in $O(N)$ space. However, the space overhead per character indexed by a suffix tree can still be significant. Many research papers report sizes of between 12 and 28 bytes per character indexed for real data (Kurtz 1999).

The internal structure of suffix trees does permit some significant space saving optimisations to be employed. For example, if a suffix tree is constructed that does not require suffix links, then no node is referred to by more than one other node. This permits a very efficient vector representation of internal nodes to be used. If a vector must grow as a result of adding a new branch then we already have access to the only reference to the node and it can be immediately updated to point to the new larger vector. If the old vector is placed on a free list it can be reused later in the tree construction.

A further optimisation is to note that leaf nodes can be represented by a single index into the original string. If string indices can be differentiated from node addresses, then only internal nodes need be represented in a suffix tree. Coupled with the efficient vector representation this can yield compact tree representations. For example, using 32-bit values for string indices and node addresses it is possible to construct suffix trees over DNA with only 9.3 bytes per character. This is as good as the best published result to date (Japp 2003). With an additional compression step such as the suffix cactus (Kärkkäinen 1995), significantly smaller representations are possible.

Where suffix links are required to be present to support tree comparison algorithms such as matching statistics (Gusfield 1997), significant space optimisations are also possible. The suffix links themselves form an alternative tree structure. Unfortunately they may result in internal nodes being referred to multiple times. There will still only be one parent node but an internal node could still be the target of up to K suffix links for an alphabet of size K . This would effectively rule out the efficient vector representation without additional indirect addressing overheads. However, using a linked list representation considerable space savings have been demonstrated (Kurtz 1999).

Even with the advances in storage representations, suffix trees are still very much larger than the strings they index. This is particularly problematic if very large strings are to be indexed such as DNA sequences. In these cases the suffix tree structures could be many times the size of physical memory, a suffix tree for an entire genome consisting of 3 billion characters could require 30 to 50 gigabytes of memory. This is in addition to a copy of the original sequence that is usually encoded using one byte per character. Coupled with the semi-random traversal of suffix tree construction the space overhead is a significant problem. However, one further aspect of suffix tree structure can be used to address this.

If we consider any suffix tree, all suffixes starting with a particular prefix, say "xab" in our first example, will all be found in the same sub-tree. Therefore, if a suffix tree were to be constructed one sub-tree at a time it may be

possible to achieve the locality of reference required to construct trees larger than memory without incurring prohibitively expensive page faulting.

One option is to construct a top-compressed suffix tree (Japp 2003), that is, save each completed sub-tree in a file for later retrieval and maintain a table of patterns describing the sub-trees. Alternatively, if there is sufficient virtual memory, each new sub-tree simply causes the previous sub-tree to be paged out since it is no longer required during construction. However, for either of these approaches to work each sub-tree must be discrete, that is there must be no links between them.

$O(N)$ algorithms rely on the presence of suffix links that would effectively link the sub-trees together in a semi-random fashion. To date only the naïve algorithmic approach has been employed to create suffix trees larger than memory (Hunt, Atkinson and Irving 2002, Japp 2003). In this paper we first outline the naïve algorithm and then demonstrate how to construct discrete sub-trees using suffix links within the sub-trees.

3 Partitioned $O(N^2)$ Algorithm

As noted above the naïve $O(N^2)$ algorithm is able to insert suffixes in any order. So, if we ignore the optimised insertion, it is possible to construct sub-trees independently. For example, if we wish to create the sub-tree for all suffixes starting with the pattern "xab" we simply scan the source string inserting each of these suffixes. If we insert this subset of suffixes in order from longest to shortest, it is also possible to implement the insertion optimisation.

Note that the result of inserting a new suffix into a suffix tree is the creation of a new branch. If this branch is inserted say M characters from the root, we know that in the complete suffix tree there are M further branches where each branch is one character closer to the root than the previous one. If we only insert the subset of suffixes starting with a given pattern we can still calculate the location of the next branching point. If the next suffix in the sub-tree starts N characters after the one just inserted, where $N < M$, this results in a new branch being added a distance of at least $M-N$ characters from the root. The first $M-N$ characters of the path from the root are known to be present and can be traversed without inspecting every character.

The key to successfully constructing a very large suffix tree with this approach is to carefully select the sub-trees we wish to construct so that they will fit into memory. As will be described later this may prove problematic for some data sets but, if a suitable partitioning can be found, the approach could be used to construct suffix trees significantly larger than memory. A limiting factor is how much free memory is available for sub-tree construction once we have made an in-memory copy of the sequence.

4 Partitioned $O(N)$ Algorithm

The development of the $O(N)$ algorithm was based on explicitly linking together branching nodes for which the naïve algorithm's insertion optimisation worked. These suffix links were then used to optimise identifying the location of branching nodes inserted later in suffix tree

construction. Since the naïve algorithm's insertion optimisation still works in the creation of discrete sub-trees it is also possible to use the suffix link optimisation in the creation of discrete sub-trees. However, only a subset of the suffix links that occur in a complete suffix tree are created.

The key to using suffix links in constructing discrete sub-trees lies in a property of the optimisation itself. If the insertion of a suffix results in a new branching node M characters from the root node then we have started a new chain of suffix links. If the new branching node is D characters deeper than the last branching node traversed, we can calculate the location of the next node on the chain of suffix links. We simply follow the suffix link of the new node's immediate parent and then traverse D characters down the tree to find the location of the next node on the chain. The important point to note is that following the suffix link still left us D characters from the next node on the chain. This is true for each subsequent insertion of a suffix until a suffix link takes us to the root node itself. Therefore, if a chain of suffix links only included nodes from a discrete sub-tree, it would not be necessary to know how many suffix links from the complete suffix tree are missing in order to locate the next node on the chain.

Using this information we can now adapt our $O(N)$ algorithm to work on discrete sub-trees. To achieve this we must ensure that no suffixes are added to the tree unless they start with the desired pattern. Three cases must be considered.

The first case is when the algorithm attempts to identify the location of the next branch by starting a traversal from the root node. When this occurs we first check that the suffix to be entered starts with the correct pattern and only proceed if it does. For each traversal we know the last node entered and how far to traverse to find the target of its suffix link. When a traversal has progressed this far we either find an existing node or the location for a new branch. Either way, we fill in the suffix link for the last node. If we found an existing node we must traverse further down the tree to find the location of the new branch. In those cases where a suffix does not start with the correct pattern we move onto the next suffix and start again. This results in the expected depth of the next node on the current chain of suffix links being one closer to the root node. When this depth reaches 0 the last node is given a suffix link pointing to the root node.

The second case is when a new branching node has been added and the immediate parent's suffix link does not refer to the root node. In this case our $O(N)$ algorithm runs unaltered. As noted above the parent's suffix link may skip some of the chain of suffix links that would be present in a complete suffix tree, but that does not affect the calculation of the location of the next node on the new suffix chain.

The third case is when the immediate parent is not the root node and its suffix link refers to the root node. Consider the problem of inserting a suffix into a sub-tree starting with the pattern "xab" as illustrated in Figure 9. If we enter a new suffix that involves traversing existing branching nodes then we will add a new branch, $N1$,

below an existing branching node $B1$. The next branching node to be added, $N2$, will be located by following $B1$'s suffix link to node $B2$, then walking down the tree to $N2$'s final position. $N2$ then becomes the target for $N1$'s suffix link. Subsequent suffix insertions will continue the new chain of suffix links $N1, N2$, etc in parallel with $B1, B2$, etc.

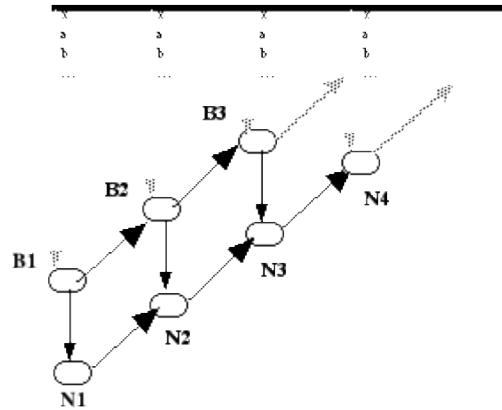


Figure 9: Inserting a sequence of nodes $N1, N2, \dots$

The third case arises in our example if $B3$ has a suffix link pointing to the root node. We know that branching nodes $B1, B2$ and $B3$ all lie on paths that start with "xab" since we are building the sub-tree for "xab". We also know that $B3$ may not be the last branching node on its suffix chain in the complete tree but it is the last branching node on the chain whose suffix starts with "xab". In the complete suffix tree there could be an additional node on the suffix chain with a path from the root node labelled "xa" or "x". If there is then this node could lie at the start of a path labelled "xab" between the root and $N4$.

To complete the chain of suffix links $N1, N2, N3, N4$ we now need to identify the next suffix that would be entered by the naïve algorithm's insertion optimisation that also starts with "xab" and how deep a traversal is needed to locate $N4$.

This can be achieved by simulating the missing "xa" node. If it were present there would be a path two longer than the path length from $B3$ to $N3$ between the root and $N4$. We know which part of the source string matched the path $B3$ to $N3$ so we can continue the algorithm by attempting to insert the suffix starting two characters earlier. If the missing "xa" node does exist then this suffix may start "xab" and be part of the sub-tree. The insertion is performed by a traversal starting from the root node. This is the first case we considered and includes a check of whether or not the suffix starts with "xab". In general when case three arises we go back by the length of the pattern less one character and resume with case one.

Once all the discrete sub-trees have been constructed it is possible to merge these together to form a complete suffix tree. Alternatively, the discrete sub-trees can remain separate to form a top-compressed suffix tree. In evaluating this algorithm we have chosen to form a top-compressed suffix tree to avoid the need for the final merging stage.

Given that the suffix link based optimisation gives $O(1)$ performance for most suffix insertions, the total time to construct a complete suffix tree from discrete sub-trees should approach $O(N)$ in practice. To test this hypothesis we have constructed a top-compressed suffix tree for chromosomes 1 and 12 of the human genome.

5 Evaluation

The sequence used to demonstrate the construction of suffix trees larger than memory is based on build 32 of the human genome. The latest build is available from the website of the US National Center for Biotechnology Information:

<http://www.ncbi.nlm.nih.gov/genome/guide/human/>

To form a single sequence we have concatenated all 24 chromosomes together from 1 to 22 and X and Y. Within each chromosome we have replaced each gap with a single don't know character, N, for each base pair in the gap. This has resulted in a single DNA sequence of 3,044,091,776 characters. The alphabet of the sequence consists of 16 characters. We have 'A', 'C', 'G', 'T' representing the four bases of DNA. The remaining characters represent all possible combinations of bases where the biological data is unclear, ie 'R', 'Y', 'W', 'S', 'M' and 'K' represent cases where the base-pair is one of two possibilities, 'B', 'D', 'H' and 'V' represent cases where the base-pair is one of three possibilities and 'N' represents a don't know. The symbol '\$' is appended to the end to ensure all suffixes of the sequence are unique.

The tree representations used for each algorithm were based on the following C data structure:

```
typedef struct tree_node
{
    unsigned int flags:28 ;
    unsigned int nchildren:4 ;
    unsigned int start ;
    Tree suffix_link ; /*O(N)*/
    Tree children[2 /*upto16*/] ;
} Node, *Tree ;
```

For each node, storage was dynamically allocated for a structure that held the correct number of children. The storage used for the $O(N)$ algorithm included a suffix link pointer in each structure which was not present for the $O(N^2)$ algorithm. Approximately half of all nodes only have two children and never need to grow in size.

When a node has to grow in size in the $O(N^2)$ algorithm, the original storage is made available for reuse and a new structure with space for one additional child pointer is created.

When a node has to grow in size in the $O(N)$ algorithm, additional storage is allocated to hold the suffix link and all but the first two child pointers. The original suffix link pointer then becomes a pointer to the additional space. When the overflow space needs to grow the original overflow space is made available for reuse.

The pointers are represented by unsigned 32-bit values with a flag in the header indicating whether a pointer is a string index or a pointer to an internal node. This allows leaf nodes to be discarded, it is sufficient to record the start of the edge leading to a leaf node using an index into the original sequence. This data structure allows

sequences of up to 4G characters to be encoded as a suffix tree using sub-trees of up to 4GB in size.

The total storage overhead for the $O(N^2)$ algorithm is approximately 12 bytes per internal node plus 4 bytes per suffix. Since there are about 2 internal nodes per 3 characters indexed this represents a total size overhead of approximately 12 bytes per character indexed. For the $O(N)$ algorithm we have the addition of a 4 byte suffix link to each node and an overflow pointer in half of the nodes. This results in a total size overhead of approximately 16 bytes per character indexed.

The most difficult part of the experiment is deciding how to partition the suffix tree to be constructed. If the DNA is random then we could predict the average suffix tree size for different lengths of pattern. For example, since most DNA consists of 'A', 'C', 'G', or 'T' we would expect a factor of four reduction in total tree size for each character in a pattern. So for patterns of length one there should be 750 million suffixes per tree, for length two, 190 million suffixes per tree and for length three, 48 million suffixes per tree. The genome sequence constructed was scanned and the number of suffixes in each sub-tree for patterns of lengths 1, 2, 3 and 4 were counted. The larger sub-trees are shown in Table 1.

Pattern	Number of Suffixes
A	840,784,386
C	582,244,178
G	582,588,852
T	842,055,822
N	196,416,625
TT	279,144,917
AA	278,345,707
AT	219,939,771
TG	206,916,931
NNN	196,373,011
TTT	109,637,932
AAA	109,225,200
ATT	71,020,931
AAT	70,910,770
NNNN	196,366,239
TTTT	44,209,037

Table 1: Suffix tree sizes by pattern

Since our suffix tree representation is up to 16 bytes per character indexed, the maximum sub-tree size required for patterns of length 1, 2 and 3 are approximately 12.5GB, 4.2GB and 2.9GB respectively. Even if the genome is compressed to 4 bits per character to reduce its size from 2.84GB to 1.42GB, this still represents main memory requirements of approximately 13.9GB, 5.6GB and 4.3GB respectively. This data presents a good example of the limitations of the sub-tree approach. If the data partitions evenly then very large suffix trees can be constructed. However, in reality, data may have interesting features that prevent this.

5.1 The Experiments

We do not have ready access to systems with large memories so the initial experiments reported here are working with sequences for chromosomes 1 and 12 of the human genome. These are 246,122,627 and 134,207,505

characters in length respectively. The sub-tree partitioning chosen was based on patterns consisting of combinations of ‘A’, ‘C’, ‘G’ and ‘T’. All suffixes that started with a pattern containing any other character were placed in a single sub-tree. For patterns of length 1 this resulted in 5 sub-trees, one each for suffixes starting ‘A’, ‘C’, ‘G’ or ‘T’ and a sub-tree for everything else. For patterns of length 2 this resulted in 17 sub-trees, one sub-tree for suffixes starting with each of the 16 combinations of ‘A’, ‘C’, ‘G’ or ‘T’ and a sub-tree for everything else.

The test environment we used is a Sun Microsystems E420R, with four 450Mhz UltraSparc2 CPUs, 4GB of main memory, two mirrored 9GB SCSI disks and running Solaris 8. The GNU C Compiler version 3.0.2 was used with the "-O5" flag to enable optimisation. A log file was generated during each run that recorded the results of a *getusage* system call after every one million suffix insertions and at the end of constructing each sub-tree. The graphs presented below only detail the performance during suffix tree construction. That is they measure the construction from just prior to inserting the first suffix to just after the insertion of the final suffix. Since our interest is purely the behaviour of the construction algorithms, the memory allocated to each sub-tree was reclaimed once completed. For a production system the sub-trees would be saved to file for later reuse rather than being discarded.

5.2 The Results

The graph shown in Figure 10 compares the time taken to enter suffixes from the chromosome 1 sequence into a partitioned suffix tree using our linear algorithm with pattern lengths of length 0, 1 and 2. The vertical axis is in seconds of elapsed time and the horizontal axis is millions of suffixes entered. Note that the curve for constructing a complete suffix tree, pattern length 0, is incomplete due to exhausting the host system’s virtual memory. The complete suffix tree for chromosome 1 requires 3.6GB of memory. The peak memory requirement for a sub-tree using the length 1 and length 2 patterns was 980MB and 425MB respectively, in addition to a 4bit per character copy of the sequence.

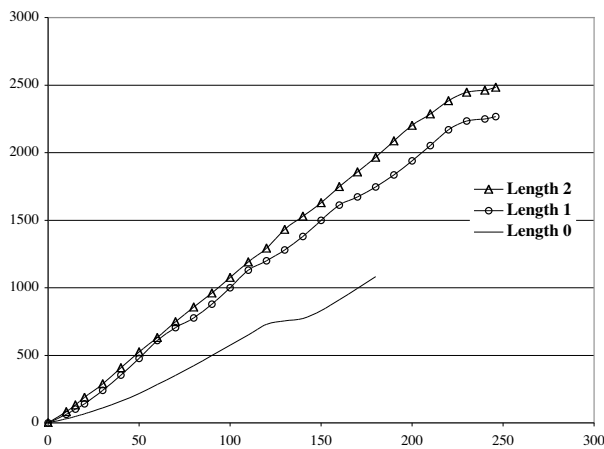


Figure 10: Chromosome 1 Linear Algorithm

The graph shown in Figure 11 compares the time taken to enter suffixes from the chromosome 12 sequence into a partitioned suffix tree using our linear algorithm with pattern lengths of length 0, 1 and 2. The vertical axis is in

seconds of elapsed time and the horizontal axis is millions of suffixes entered. The complete suffix tree is 2,031MB in size. The peak memory requirement for a sub-tree using the length 1 and length 2 patterns was 580MB and 192MB respectively, in addition to a 4bit per character copy of the sequence.

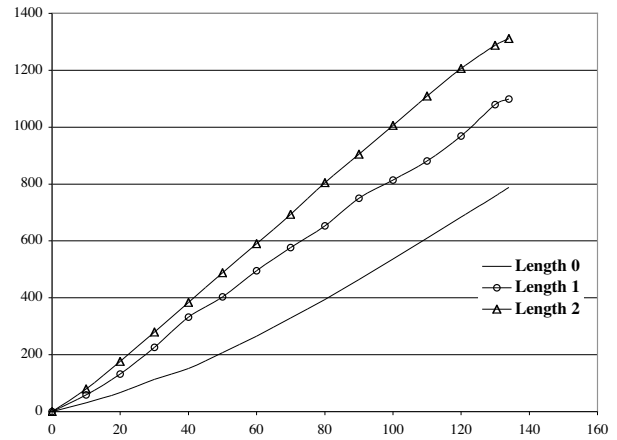


Figure 11: Chromosome 12 Linear Algorithm

The graph shown in Figure 12 compares the time taken to enter suffixes from the chromosome 12 sequence into a partitioned suffix tree using our linear algorithm and the naïve algorithm. In both cases a pattern length of 2 was used. The vertical axis is log base 10 of elapsed time in seconds and the horizontal axis is millions of suffixes entered. The peak memory requirement for a sub-tree using the naïve algorithm was 148MB. The linear algorithm required 1,312 seconds to construct the partitioned suffix tree whereas the naïve algorithm required 57,102 seconds.

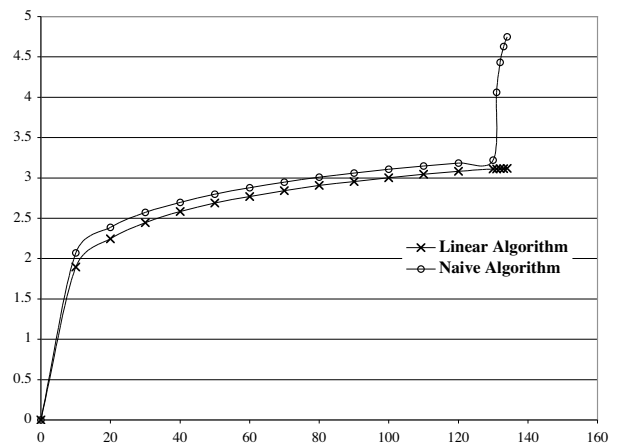


Figure 12: Chromosome 12, Linear and Naive Algorithms

5.3 Analysis

The experiments reported above demonstrate that the new partitioned suffix tree construction algorithm can construct suffix trees larger than memory in linear time. The experiments also highlight some data dependent limitations of this approach one of which appears to render the naïve algorithm ineffective.

Figures 10 and 11 illustrate the linear time behaviour of the partitioned algorithm. As would be expected an increase in pattern length increases the total time taken since the input sequence must be scanned an increasing number of times but this need not be so. The sub-trees

being constructed are all independent and could be constructed in parallel. Parallel implementations of the naïve algorithm are also possible. The speed up that can be achieved is of course dependent on the largest sub-tree that must be constructed. This in turn is a consequence of the chosen partitioning.

From the brief analysis of the single sequence constructed from the human genome it is apparent that real data may not partition evenly. For example, in a randomly constructed sequence of 15 different characters, a 3,044,091,776 character sequence partitioned using patterns of length 4 should consist of 50,625 sub-trees of about 60,000 suffixes. However, table 1 shows that even with a pattern length of 4 our DNA sequence would still have one sub-tree of 200 million suffixes. In effect, finer grain partitioning does not necessarily increase the scale of suffix tree that can be constructed. The limitation is clearly data dependent.

A further limiting factor is that the sequence being indexed must also be resident in memory. For large sequences such as the entire human genome this severely limits the memory available to hold individual sub-trees. If the problematic partitioning is repeated for other DNA sequences it may not be possible to build suffix trees over DNA that are more than 5 to 10 times the size of memory.

Figure 12 illustrates the relative performance of the new linear algorithm in comparison to the naïve algorithm. The rationale for developing a partitioned $O(N)$ algorithm is that for very large scale data an $O(N)$ construction algorithm should be important. However, the graphs presented show that, in some cases, there may not be a significant performance difference between the optimised partitioned $O(N^2)$ algorithm and the partitioned $O(N)$ algorithm.

If we ignore the last four million suffixes entered for chromosome 12, both algorithms have very similar performance and both exhibit a linear growth rate. It is also worth noting that others have observed similar comparative behaviour for non-partitioned implementations of these algorithms in Java (Hunt, Atkinson and Irving 2002).

The key difference between the two algorithms arises when the last few suffixes are entered. The last sub-tree constructed includes all patterns involving unknown bases of which there are a very large number. These unknowns occur in numerous long sequences and result in the naïve algorithm exhibiting worst case performance as indicated by the vertical spike in Figure 12. In contrast long sequences of the same character result in the new algorithm exhibiting best case performance as indicated by the flattened tails on the curves in Figure 10.

The extent to which this disparate behaviour would occur in practical applications is unclear. Clearly the chromosome sequences constructed for these experiments have exposed the problem but this may not be a biologically appropriate way of assembling the data.

6 Conclusions

In this paper we have illustrated an $O(N)$ suffix tree construction algorithm based on suffix links that can

construct suffix trees many times larger than physical memory. The preliminary experimental results suggest that the partitioned construction algorithm exhibits $O(N)$ performance on real data. The results also demonstrate that a naïve suffix tree construction algorithm with a simple insertion optimisation can exhibit similar performance. However, it is clear that certain sequences of characters can effectively defeat the naïve algorithm by forcing it to exhibit worst case performance.

In addition to the linear growth rate the new partitioned construction algorithm can be easily parallelised with the speed up only being limited by the relative size of the largest sub-tree that must be constructed.

Future work will consider a wide range of different space reduction techniques that can be employed in the construction of suffix trees (Gusfield 1997, Hunt 2003, Japp 2003, Kärkkäinen 1995, Kurtz 1999). We will also consider new techniques based on some of the interesting structural properties of suffix trees (Kurtz 1999, Monostori, Zaslavsky and Schmidt, 2002).

7 References

- Baeza-Yates, R. and Navarro, G. (2000): A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, **1** (1): 205-239.
- Gusfield, D. (1997): *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge, Cambridge University Press.
- Hunt, E. (2003): The Suffix Sequoia Index for Approximate String Matching. Technical Report TR-2003-135, Department of Computer Science, University of Glasgow, Glasgow.
- Hunt, E., Atkinson, M.P. and Irving, R.W. (2002): Database indexing for large DNA and protein sequence collections. *The VLDB Journal*, **11**: 256-271.
- Japp, R. (2003): Persistent Indexes for Data Intensive Applications. *Twentieth British National Conference on Databases*, Coventry, UK, Lecture Notes in Computer Science, **2712**, Springer.
- Kärkkäinen (1995): Suffix Cactus: A Cross Between Suffix Tree and Suffix Array. In *Proc., Sixth Symposium on Combinatorial Pattern Matching (CPM'95)*, Lecture Notes in Computer Science **937**: 191-204, Springer.
- Kurtz, S. (1999): Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, **29** (13), 1149-1171.
- McCreight E.M. (1976): A Space-Economical Suffix Tree Construction Algorithm. *Journal of the Association of Computing Machinery*, **23**(2):262-272.
- Monostori, K., Zaslavsky, A. and Schmidt, H. (2002): Suffix Vector: Space and Time Efficient Alternative to Suffix Trees, *Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*, Melbourne, Australia. *Conferences in Research Practice in Information Technology*, **4**: 157-165, Australian Computer Society.
- Ukkonen, E. (1995): On-line Construction of Suffix Trees. *Algorithmica*, **14** (3): 249-260.
- Weiner, P. (1973): Linear Pattern Matching Algorithm. In *FOCS73*, 1-11.