ANALYZING CACHE CONSCIOUSNESS OF MAIN MEMORY SUFFIX TREES

A Project Report

Submitted in partial fulfilment of the requirements for the Degree of

Master of Engineering

in

Internet Science and Engineering

by

Shivashankar J

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

JULY 2008

Acknowledgements

Many thanks to my guide Prof. Jayant Haritsa and all my labmates for their guidance, support and cooperation in the course of this work.

Abstract

A Suffix Tree *is a tree structure which exposes the internal structure of a string in a deeper way helping to solve problems on strings quickly. With increasing size of main memory even big suffix trees can be fitted into main memory. The Ukkonen construction algorithm used in constructing the suffix tree does not produce tree layout which is cache friendly. In this report, we explore the possibility of finding cache conscious layouts and data structures so that the algorithms applied on suffix tree can be executed faster. We show that the length of present cache lines in modern processors are too insufficient to exploit spatial locality for suffix tree search algorithms. We analyze the effects of different data structures used for suffix tree searches in depth, and find the most efficient structure that is cache conscious. The cache conscious data structures perform search faster by approximately 70% as compared to standard implementations.*

Contents

List of Figures

Keywords

Suffix tree, Ukkonen, Trellis, Cache conscious, Stellar.

Chapter 1

Introduction

1.1 Suffix Tree

A suffix tree is a tree data structure that represents the suffixes of a given string such that it helps fast implementation of many important string operations. Some of the string algorithms applied with a suffix tree represent the lower bounds for those operations.

Given a string S, the suffix tree is a tree whose *edges* represent *strings* (can be one character or more than one character), and every suffix of S (there are n suffixes in a string of length n) corresponds to *one and only one path* from the tree's root to some leaf. The path-label of a node refers to the label of the path from the root of T to that node. Thus every node can be uniquely represented as its path label.

To construct a suffix tree for string S, it takes linear space and time in the length of the string. There are a number of algorithms in literature like

- Weiner algorithm [8]
- McCreight algorithm [9]
- Ukkonen algorithm. [10]

Wiener was the first to show that suffix trees can be built in linear time. His algorithm assumes that the entire string is known at the start of the algorithm and does a *right to* *left scan* and builds the suffix tree. Ukkonen differs in its approach that it scans *left to right* and constructs the suffix tree using *lesser space* than Weiner's algorithm.

Post construction, a number of operations can be performed efficiently. Example search algorithms include

- locating a substring in S
- searching all occurrences of a substring in S
- matching statistics or maximal substring search
- longest common substring problem
- least common ancestor of any two nodes

Its also important to note that the size of the suffix tree is on an average around 20 times the size of the string. So if an input string is 1 MB long (or the number of characters is 1 million), the suffix tree is typically around 20 MB.

There are two kinds of nodes in a suffix tree, internal nodes and leaf nodes. The *internal node* has at least 2 children and a maximum of $|\Sigma|$ children, where $|\Sigma|$ represents the alphabet size. The edges are labelled with the string they represent. A *suffix link* is an important addition to the internal node information to make the construction of suffix tree linear. Let xα be an arbitrary string, where x denotes a single character and α denotes a possibly empty substring. For an internal node v with path label xα, if there is another node s(v) with path-label of α, then a pointer/edge from v to s(v) is called a suffix link. Leaf nodes do not contain suffix links.

An example suffix tree is shown in Figure 1.1. The suffix tree is the representation of the string 'GTTAATTACTGAAT\$'. The solid lines represent tree edges from parent to children. The suffix links of internal nodes are represented as dashed lines. Once a suffix tree is constructed, a lot of string operations become simple and straight forward. For example, in the suffix tree in figure 1.1, if we need to find the presence of substring 'TACT' then we see if there is a path from the root with label 'TACT'. If it exists, then the substring exists and otherwise it doesn't exist.

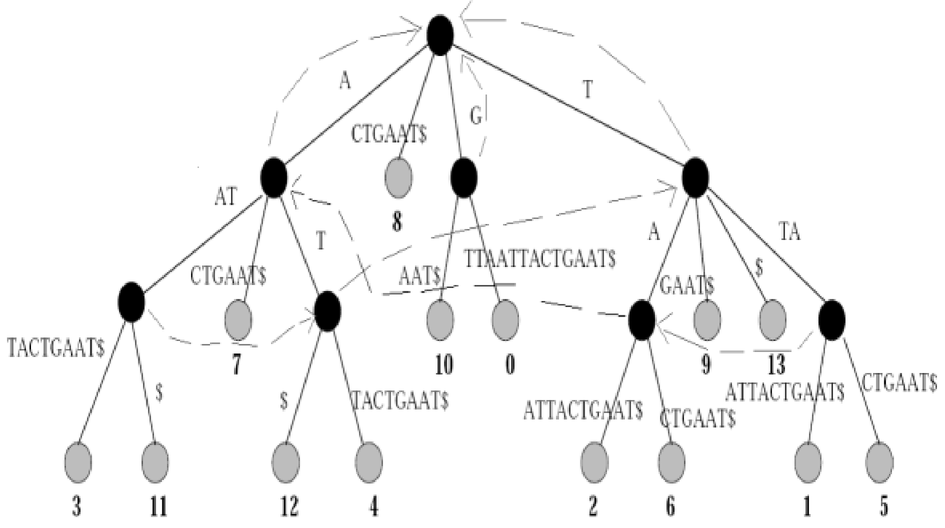


Figure 1.1: suffix tree example

1.2 Organization of this thesis

Chapter 2 gives details about how the construction is achieved in linear time and space emphasizing on the Ukkonen algorithm. It also gives details about two algorithms applied on the suffix tree - exact substring search and maximal substring search. Chapter 3 discusses related work like trellis algorithm [5], cache conscious prefix tree [20], etc. Chapter 4 discusses work done (layout changes) and experimental results. Chapter 5 discusses the data structure modifications made to the suffix tree and its experimental results. The last chapter discusses the conclusion, future work and ideas.

Chapter 2

Literature

2.1 Ukkonen algorithm and construction of suffix trees

Fig 2.1 gives a summary of Ukkonen algorithm at a high level. If no optimizations or tricks are applied, the method looks to be a O(n^3) time algorithm. A lot of optimizations have been applied by Ukkonen to make this algorithm an O(n) algorithm.

The algorithm is divided into n phases. In each phase $i+1$, tree st_{i+1} is constructed from st_i. Each phase i is further divided into i extensions. In extension j of phase $i+1$ the algorithm first finds the end of the path from the root labeled with substring InputSeq[j..i]. It then extends the substring by adding the InputSeq[i+1] to its end, unless it's already there.

The most important optimizations used by the Ukkonen algorithm are summarized here

- *Use of suffix links:*

  Normally every node in a tree has pointers to its parent and its children. But a suffix tree node has a special type of pointer, in addition to the parent and child pointer, which is the suffix link. If a internal node has a path label of $x\alpha$ from the root, then the suffix link of this node will point to the internal node which has a

```
**UkkonenSuffixTreeConstruction (InputSeq[n])**
Input:
InputSeq[n] (The string for which tree has to be built)
Let $st_0$ be the implicit suffix-tree for InputSeq[0]
outerloop:
for i = 0 to n do
  j = 0
  innerloop:
  while j less than (i + 1) do
  find the node $N_{ij}$ whose path-label is InputSeq[j..i] in $st_i$
  if $N_{ij}$ ends at a leaf $l_k$
  then
  Extend $l_k$ by adding character $s_{i+1}$
  else
  if (from the end of $N_{ij}$ there is no path labeled $s_{i+1}$)
  then
  $st_{i+1}$ = split edge in $st_i$ and add a new leaf
  else
  $st_{i+1}$ =$st_i$
  end while (innerloop)
end for (outer loop)
```

Figure 2.1: Ukkonen Algorithm

path α from the root. Intuitively, this reduces the need to come from root to the node represented by α and can be reached just by traversing the suffix link. This reduces the time complexity to O(n^2) algorithm

- *Skip and count trick:*

  This trick is based on the observation that when the suffix link of the present node is not available and we have to move to its parent, follow the suffix link and then come down to the correct node which is the corresponding suffix link of the original node then we can blindly count the number edge length of the correct child to be followed rather than comparing the characters of an edge one by one.

- *Representation of edge as start index and end index:*

  Consider a string from a-z, where no characters are duplicated. The alphabet size is 26 and string size is also 26. If we have to represent the edge labels by the actual string, there are 26*27/2 characters in all (including $ symbol at the end). This means that if we try to represent the characters of every edge in the suffix tree in the nodes, then the algoithm will be O(n^2) as the input itself is O(n^2). So each edge is represented as start index and end index, thus making all node size uniform, also reducing the need to store these combinations at the node level. The problem that arises because of this representation is that, the original string must be present along with the suffix tree for any post construction algorithm to work.

- *Representing the intermediate end index by a special symbol 'e':*

  We note that every time in the main loop of the algorithm, a lot of nodes are only extended by only one character, i.e the end index is updated by one. Instead of doing this for every loop, we do this only once at the end. In the intermediate stages the end index is just represented by a special symbol say 'e' which is updated to the length of the string after the outer loop of the algorithm finishes.

Due to all these enhancements and improvements, the Ukkonen algorithm reduces to a linear time algorithm.Ukkonen algorithm is the most popular way of constructing a suffix tree till date due to the following salient points:

- It is an online incremental algorithm.
- It scans the string from left to right unlike other linear time algorithms like Wiener algorithm which goes from right to left.
- It is also a linear space algorithm. No temporary extra memory is required.

Problems

Though the Ukkonen algorithm has lots of advantages, it has its shortcomings. Some of them are:

- The use of suffix links make the traversals cache unfriendly. It is observed that while travelling from one node to its suffix link the locality of reference is lost as the node of action shifts from one part of the tree to a different part of the tree. Trying to put the suffix links together (we can view the tree as a tree whose edge labels are suffix links), we lose locality of reference with respect to the normal tree edges. Many papers (e.g Stellar[4]) exist which try to maintain locality of suffix link and tree edges at the disk page level.
- The creation order of the internal nodes are random and the nodes are created such that the siblings of a node are created in a non localized way. This is because an internal node can be created at any time of the algorithm whenever a need to split an edge arises. This is completely independent to any split that may happen to the edge that arises from the sibling of a node. Also because of the inherent nature of the suffix tree, the tree is unbalanced and leaves occur at different levels of the tree.

2.2 Algorithms used on suffix trees

This section brings out different algorithms which can be used on a suffix tree. It explains two of the algorithms, one which uses suffix link and the other which doesn't use suffix link.

For the following discussion, assume the length of the original string is n and the suffix tree is built over this string.

Algorithms that do not use suffix links

- Exact string matching:- Given a query string of length 'm', it can be checked in O(m) time whether this is a substring that occurs in the original string
- Exact string matching, finding all occurrences:- This algorithm is similar to the previous algorithm but finds all the occurrences of the query string in the original string
- Largest Common substring of two strings:- One method of finding this is that a single suffix tree consisting of the two strings are built (this is called a generalized suffix tree). Then using this generalized suffix tree, the largest common substring that occurs in both the strings can be found out. One more method of finding the solution uses the suffix links.

Algorithms that use suffix links

- Matching statistics:- The original string of length n. The query string is of length m. The objective of this method is to produce at each index of the query string, a number which is called the matching statistics at that index represented by ms(i), i ranging from 1 to m. ms(i) denotes the maximal substring that occurs somewhere in the original string that matches with the query string starting at i.
- Longest common substring algorithm using suffix links:- Compute the matching statistics after constructing the suffix tree for one of the strings (instead of two used by the algo without suffix links) and then make a linear scan over the matching statistics obtained and the largest value gives the longest common substring for the two strings. The advantage is that this is more space efficient than the other algorithm which doesn't use suffix links.

Now we explain two of the algorithms in detail

Exact string matching, finding all occurrences

In this algorithm, using the query string of length m we traverse down the suffix tree of the first string. If before all the characters of the query string are matched, we get a mismatch, then the query string does not occur in the original string as a substring. If the query string does match and ends at a leaf node, then the substring occurs exactly once in the original string and its starting position can be got from the leaf node. If the match ends at an internal node, then the number of times the query string occurs in the original string is the number of leaves under the internal node and the index at which they occur can be obtained by travelling down the subtree till the leaves.

Exact string matching can also be done using the Knuth Morris Pratt [21] algorithm. But there are some advantages of using suffix tree over KMP algorithm.The salient features of this algorithm are (for simplicity we will consider first occurrence):

- The construction of the suffix tree and the search algorithm (first occurrence) takes O(m+n) time. This is the best known exact search algorithm. The KMP algorithm also achieves the same bounds.
- The advantage over KMP algorithm is that this involves a one time construction and so given 'l' queries this can find solution in O(n+lm) whereas the KMP algorithm will have to do it in O(l(m+n)).
- If the initial string on which query will be done is known earlier, then suffix tree is ideal for such a scenario. Since the suffix tree can be constructed initially and then any query can be answered in linear time in its size as it arrives.

The second algorithm that will be explained uses suffix link

Maximal substring search

This algorithm starts at the root and tries to find matching statistics (i.e the maximal substring in the original string that matches with the query string starting from some index 'i') for each position of the original string. It prints the match only if the matched string is greater than a user given threshold.

Maximal Substring search

```
Algorithm
MaximalSubstringSearch (S, T , Q, $\lambda$)
Input:
S : Database sequence
T : Suffix-tree over the database sequence S
Q : Query string
$\lambda$ : Minimum match-length to be reported
Output:
L = triplets (len, q, d) such that Q[q ... q + len] = S[d ... d + len]; Q[q + len
+ 1] != S[d + len + 1]; l greater than $\lambda$ and len is maximal given q
Algo:
v = root of T ; j = 0; k = 0; L = NULL
for i = 0 to $\|Q\|$ do
  (v', j) = StepDown(v,Q[i..]) (//v' is the node at which matching has stopped,
      and j is the length of the match)
  if j greater than $\lambda$ then
  L = L Union TraverseSubtree(v')
  end if
  if IsLeaf (v') = true then
  k = v'.edgelen - j
  v' = v'.parent
  end if
  v = v'.suffixlink
  v = SkipDown(v, k,Q[i...]) //Use the skip-count trick to traverse without
      comparisons
end for
```

Figure 2.2: Maximal substring search Algorithm

It traverses from the root till the first mismatch occurs which gives matching statistic number for index 1 (represented as ms(1)). Next it traverses the suffix link and then reaches the node whose path label is α which is the substring starting from index 2 till ms(1) in the query string. From here it starts and tries to match any more characters from the query string. Whenever a mismatch occurs,the maximal string matched so far is printed if it exceeds the threshold length specified. The algorithm is shown in Figure 2.2.

The complexity of the maximal substring match is O($|Q|$ + loc),where $|Q|$ is the length of the string queried and loc is the number of locations of match.

Since this algorithm used both suffix link and tree edges while searching, this will be considered as a representative algorithm for all search algorithms of the suffix tree for the analysis and experiments that follow.

2.2.1 Stellar Algorithm

This algorithm changes the layout of nodes of a suffix tree such that it makes search faster. It is a disk based algorithm. The algorithm starts the suffix-tree traversal at the root of the suffix-tree, and recursively traverses the subtree below. When a node is visited, the suffix-link target of the node is visited next, if not already visited through the tree-edges. Thus an internal node and its suffix-link target treated as a pair, and are scheduled for recursive traversal in sequence. This results in subtree under a node and the subtree under corresponding suffix-link target to be recursively processed in succession resulting in a large fraction of suffix-links that span these two subtrees to be intra-page, in addition to the tree-edges of each subtree. When enough nodes have been visited to fill a page, each node in the queue is scheduled for a separate recursive Stellar traversal, until all the nodes have been processed.

2.2.2 Trellis Algorithm

The Trellis algorithm is a disk based algorithm characterized by the fact that

- It constructs a disk based suffix tree for very large DNA strings like the human genome and
- It retains the suffix links

A few other algorithms have been proposed in literature which also builds disk based suffix trees ([11, 7]) but does not include suffix links and so cannot be used in algorithms like maximal substring search for fast implementations.

2.2.3 Related work in cache conscious data structures

In [20], FP tree mining is explored w.r.t cache consciousness. Cache-conscious prefix tree is proposed to address poor data locality and instruction level parallelism. The resulting tree improves spatial locality and also enhances the benefits from hardware cache line prefetching. A tiling strategy is used to improve temporal locality. The result is an overall speedup of up to 3.2 when compared with state-of-the-art implementations. They also show how the algorithms can be improved further by realizing a non-naive thread-based decomposition that targets simultaneously multi-threaded processors which ensures cache re-use between threads that are co-scheduled at a fine granularity.

In [13], B+-Trees are shown to be not cache conscious as their utilization of a cache line is low since half of the space is used to store child pointers. A new indexing technique is proposed called Cache Sensitive B+-Trees" (CSB+-Trees). It is a variant of B+-Trees that stores all the child nodes of any given node contiguously, and keeps only the address of the first child in each node. The rest of the children can be found by adding an offset to that address. Since only one child pointer is stored explicitly, the utilization of a cache line is high.

Chapter 3

Layout Changes

3.1 WORK DONE and EXPERIMENTAL RESULTS

The work carried out involves two aspects

- Analyzing the layout of the nodes in the main memory and finding the cache conscious characteristics of the different layouts
- Changing the data structure of the node and making it more cache friendly

3.1.1 Layouts

Four different layouts were analyzed

- Default Ukkonen Creation Order Layout
- Breadth First Layout
- Trellis Layout
- Stellar Layout

Creation order Layout

Here the default layout of the suffix tree when the Ukkonen algorithm is applied on a string is called the Creation order layout. As the construction of the tree proceeds, an edge of the tree connecting two nodes may be broken and a new node inserted in between but in the layout, the new node would be at the end, thus spatially separated from its parent and also its child.

Breadth First Layout

Here, after construction of the tree using Ukkonen algorithms, a post facto reorganization is done so that the siblings are grouped together.

Trellis Layout

In main memory, Trellis Layout is basically a Depth First Layout, i.e during the post facto reorganization, a depth first traversal is done on the ukkonen layout and the nodes are rearranged.

Stellar Layout

In this layout, depending on the parameter 'nodes per page' (here *page* refers to just a logical grouping of nodes), both siblings and the suffix link neighbours are brought as close as possible. As soon as a node is placed, its suffix link neighbour if not already placed is allocated the neighbouring location. Both the nodes' children and its suffix neighbours' children are placed in the queue to be further processed. This is also a post facto construction.

Experiments

For all the layouts, the L1 and L2 cache misses were measured. The maximal substring search algorithm was run on the suffix tree built from a 10 MB character file consisting of 1 million DNA string characters and the address traces were recorded in a file. This trace was fed to Dinero cache simulator[6]. The size of the suffix tree is 556 MB. We need twice this memory to change the layout. Also extra memory is required to map the addresses of the nodes to the new address for the changed layout. The results are shown in the following figure 3.1, 3.2 and 3.3. Since we have only simulated the cache

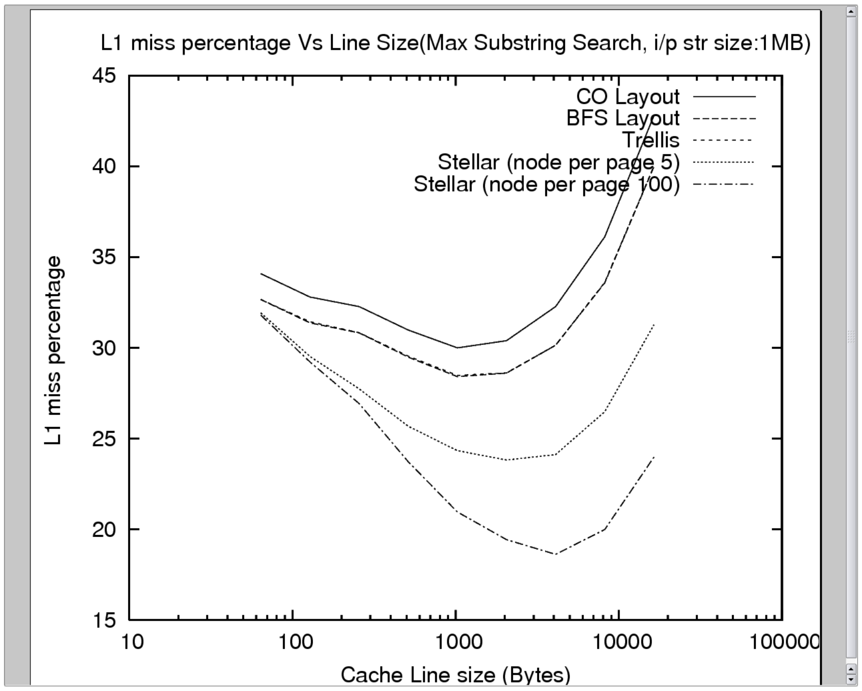


Figure 3.1: L1 misses of different layouts (Trellis and BFS Layout have almost the same values)

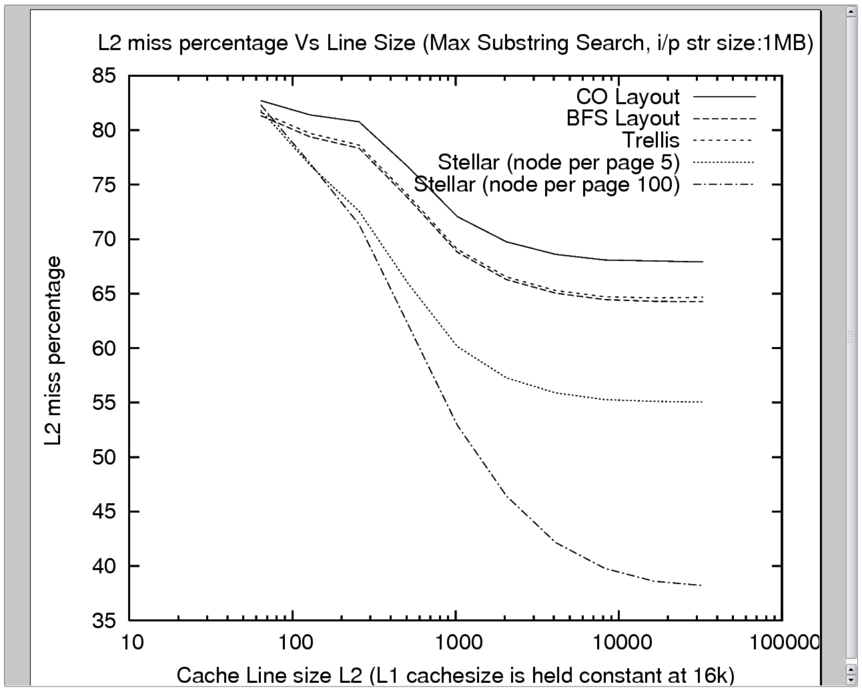


Figure 3.2: L2 misses of different layouts

parameters, it is not possible to study the impact of these cache improvements on the overall application time.

In the plot of Figure 3.1, the size of the cache line (which is a parameter to be given to the Dinero simulator) is varied from a small size to the total size of the cache itself. Here the total size of the cache is assumed to be 16k. We observe that initially, the cache miss rates decrease as the cache line size is increased and the spatial locality property is exploited more and more. Observe that the X-axis exponentially increases and Y-axis has values from 18% to 40% for Figure 3.1, and 40% to 80% for Figure 3.2.

As the size of the line becomes comparable to the size of the cache, the number of lines in the cache decreases. Due to this, the cache miss again begins to rise in Figure 3.2. In case of L2 misses, the increasing cache line size decreases the L2 cache miss rates. As the cache lines considered are much smaller than the total L2 size(8 MB), there is no rise of cache miss rates with increasing cache line size as seen in the previous case.

The maximum impact is at 1k - 4k range. This means the cache lines have to be

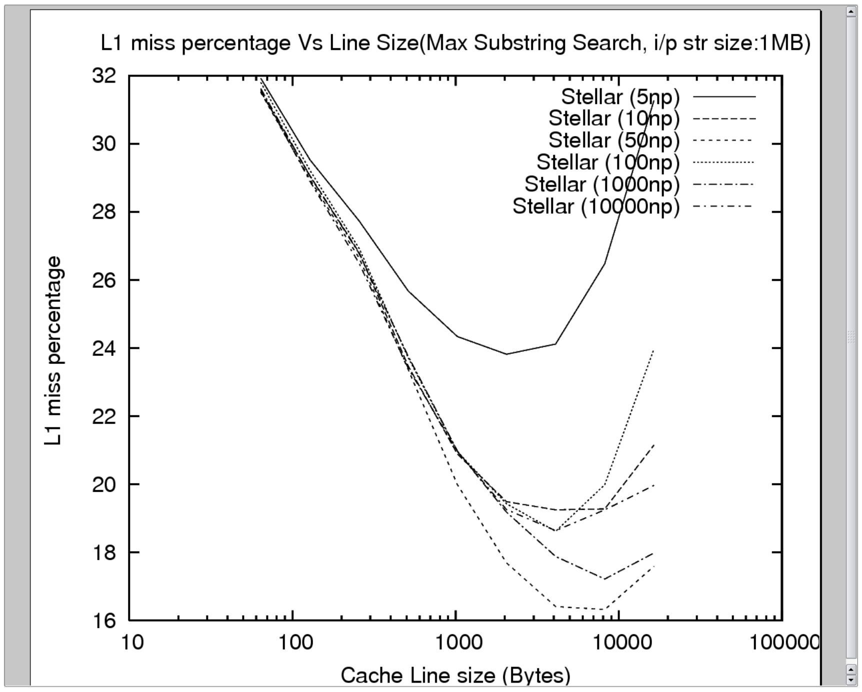


Figure 3.3: L1 misses of stellar layout with varying 'nodes per page'

much larger than what the present processors offer. The Intel pentium processors have around 64 bytes cache line size and AMD processors have around 128k cache line size. We observe that only if the cache line size is around 1k, or around 6% of the total cache size, then we can get significantly lower cache misses for the different layouts. Otherwise the cache miss rate tend to remain almost the same. So the present cache line sizes are too small in order to exploit the spatial locality while searching in suffix tree.

Also, we observe that the Stellar layout will have a much lesser cache miss as compared to the Breadth First and Creation Order Layout. This concurs with the results obtained in [1] for the disk level layout. This is not surprising, as the page size at the disk to main memory level is around 4k in modern systems, and we see that Stellar will outperform other layouts as shown in the graphs.

We now explore the optimal node per page at which the stellar layout gives the best performance in terms of cache misses. 'Node per page' is a parameter given to the algorithm so that the logical grouping of nodes can be done based on this parameter.

If 'node per page' is 100, it means that for a logical grouping of 100 nodes, the suffix link and tree edge locality is maximally preserved within the group, i.e most of the tree edges and suffix links are within the group itself. Figure 3.3 gives the comparison of cache miss rates for different 'nodes per page' or λ . We observe that small value of λ like 5 tends to give a non optimal performance, but a λ value of 50 or more gives a much better performance. The optimal value is around 50 to 100. The other values also come very close to the optimal performance as seen in the plot. This implies that if λ value is sufficiently large, then locality property is exploited efficiently by the cache lines.

The reason that the best performance of the stellar layout is achieved when the node per page parameter is around 50 to 100 nodes, is probably due to the fact that when the logical group size is equal to the cache line size, then the spatial locality propety is exploited best. Assuming that each internal node is around 25 bytes, then each logical group occupies around 1250 bytes for 50 node per group or around 2500 bytes for 100 node per group. When the cache line size is also around 1k to 4k bytes then the stellar layout gives the minimum L1 and L2 cache misses.

Now we explore the other direction of trying to obtain a cache friendly layout, i.e by modifying the data structures.

Chapter 4

Data Structure Modifications

4.1 Linked List Vs Array Implementation

Previous works have proposed both linked list representation and array based implementation of the nodes that form the suffix tree. For the DNA based suffix trees the node layout of linked list and array implementation is given in Figure 4.1.

The linked list representation is useful as the node size of internal node reduces very significantly. The size of the internal node for an linked list representation is 20 bytes. This can be further reduced to around 17 bytes as shown in [14] by exploiting redundant information. But the major problem in a linked list representation is that for search algorithm, traversing the tree becomes inefficient. For example to reach the 4th child we must traverse the first three siblings and then reach the fourth child of a node using the sibling pointer. This overhead is significant and increases the cache misses significantly as we show in the experiments.

4.1.1 Experiments and Results

Here we show that the linked list representation leads to more cache accesses (or memory accesses). So though the node space reduces when linked list is used, as more nodes have to be visited, the search algorithms have higher memory accesses making linked list not a very attractive scheme. This same observation was made in [1] for the disk level layout.

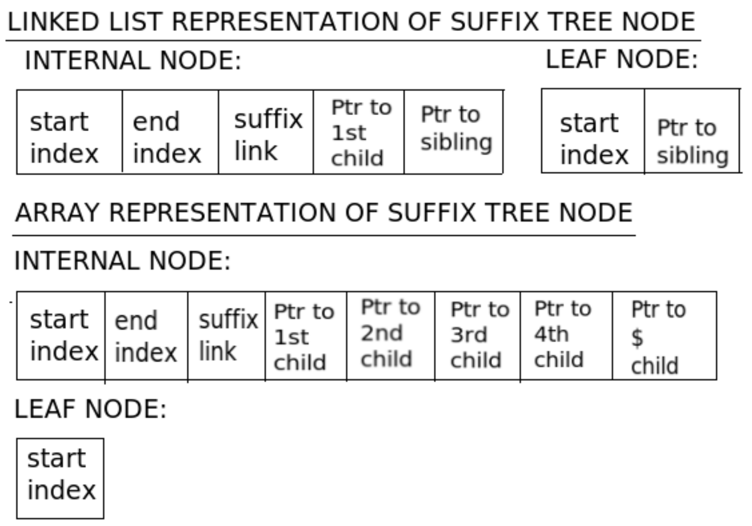


Figure 4.1: Node Representations

	Linked List	Array
Total Reads	143 million	51 million
L1 misses	24 million	19 million

The above table gives the cache misses for array and linked list representation. The suffix tree was built on a 1MB DNA string. 10000 queries were fired for maximal substring search. The resulting address trace is fed into the dinero cache simulator with parameters set to both intel and AMD processor cache parameters. We observe that the number of reads in case of an array is 35.6% of the total reads in case of linked list. This manifests in the search time being longer for linked list implementation.

The above table shows the difference in speed of execution of 10000 query strings for maximal substring search on input string size of varying size. The reason for considering maximal substring search is that it traverses both suffix links and tree edges thus can be considered as a representative algorithm for a variety of search algorithms used on suffix trees. It is clear that array based implementation is better than the linked list counterpart.

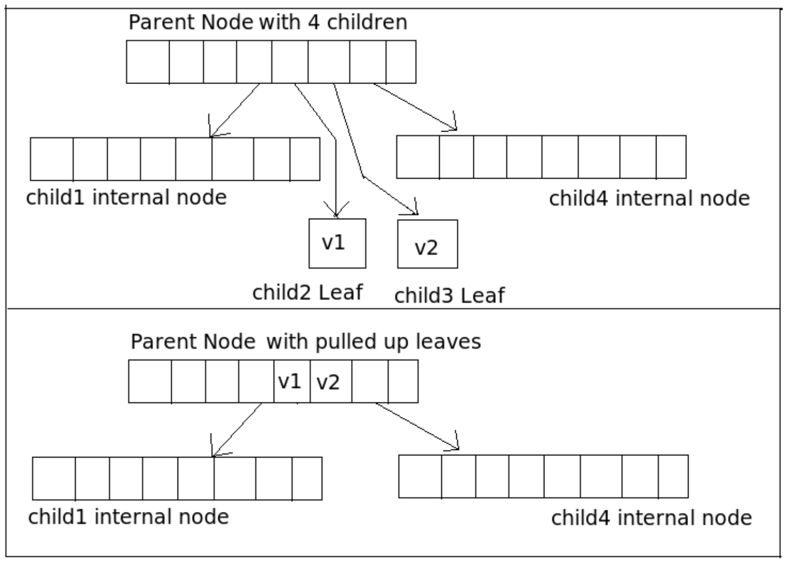
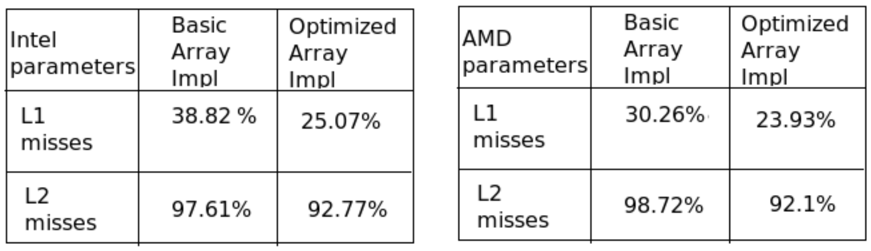


Figure 4.2: Nodes with leaf information in parent node itself

4.2 Array Implementation - pulled up leaves

As we have shown that array implementation performs better than linked list implementation, we now focus our attention to how to reduce the node size in an array based implementation. One of the methods is to remove the leaf nodes itself and instead encapsulate that information in the pointer space of the parent node where originally the pointer to the leaf node was present. Now given a pointer field of an internal node, we need to distinguish between a pointer to another internal node and a leaf's start value stored in the pointer space itself. For this a information byte is added to every internal node. Since a node can have five children at the maximum (including the $ child) we reserve 5 bits in the info byte to indicate whether the corresponding pointer space is a pointer to an internal node or the leaf node itself. Figure 4.2 illustrates the concept.

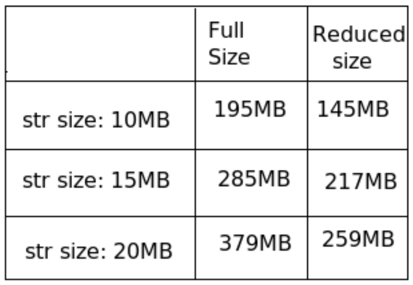
This kind of a tree is called a position tree as mentioned in [2]. This concept helps in reducing the space occupied by the suffix tree by a significant extent. Instead a 4 byte leaf, we replace it with a 1 byte info byte on every internal node. So at least 3 times the number of leaf nodes (which is equal to the number of characters) is saved.



Intel parameters	Basic Array Impl	Optimized Array Impl
L1 misses	38.82 %	25.07%
L2 misses	97.61%	92.77%

AMD parameters	Basic Array Impl	Optimized Array Impl
L1 misses	30.26%	23.93%
L2 misses	98.72%	92.1%

Figure 4.3: Cache Miss Rates



	Full Size	Reduced size
str size: 10MB	195MB	145MB
str size: 15MB	285MB	217MB
str size: 20MB	379MB	259MB

Figure 4.4: Size of Full tree and the reduced tree (after elimination of non null pointers)

Further, observe that this elimination of storage space for leaf node cannot be done if it was a linked list implementation as the sibling pointer has to be present in a leaf node implementation of a linked list suffix tree. Both Intel cache parameters and AMD cache parameters were input to Dinero simulator and the results are shown in Figure 4.3. The cache miss analysis is presented in the next subsection.

	Linked List	Array
Time taken (str size: 10MB)	11.6 sec	9.8 sec
Time taken (str size: 15MB)	14.1 sec	10.9 sec
Time taken (str size: 20MB)	16.8 sec	14.1 sec

4.3 Array Implementation - Retaining only non null pointers

In an array based implementation, most of the node space is reserved for pointers to children as there are five pointers to 'A', 'C', 'G', 'T' and '$' respectively. In most of the cases, the number of children of a node may be less than the maximum. When a suffix tree was built over a sample 10MB string, then we observe that 60% of the internal nodes had only two children, 15% had 3 children. So instead of allocating space for all children, only those pointers with non null pointers are retained. This helps us to save space for the suffix tree.

As we see from the table in Figure 4.4 that as the size of the suffix tree increases, significant amount of space can be saved by removing the non null pointers. The saved space can be as much as 25% of the total size of the tree. We need to add a information byte at the start of the node. For every internal node, the last five bits of the information byte indicate whether the corresponding pointers to children for that node exist or not. If the bit is set to 0, then the pointer is null and otherwise the corresponding pointer exists. To access the correct pointer, the previous bits have to be evaluated. For example, if the fourth child is present, then the bits for the first, second and third child are evaluated and depending on this information, the correct offset to the fourth child pointer is calculated.

Now we analyze the reduction in cache misses because of pulled up leaves and non null pointers. The table in Figure 4.3 gives details of the cache misses.

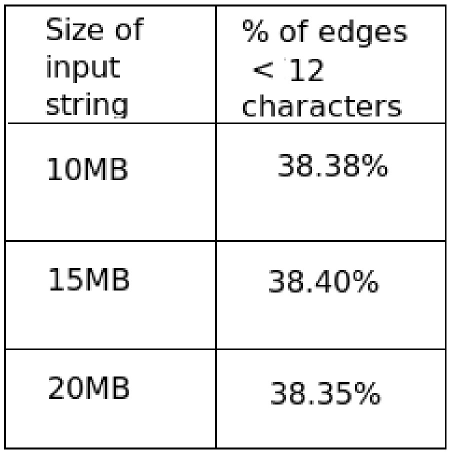
The first table in Figure 4.3 represents simulations for Intel cache parameters with cache size of 512k and cache line size of 64bytes and associativity 4. The second table represents simulations for AMD cache parameters with cache size of 1024k with cache line size of 128bytes and associativity 2. Both the experiments had address traces of maximal substring search of 10000 queries run on a suffix tree with total of 10 million characters long string.

We observe that the L1 misses comes down from 38% to 25% and in the second case from 30.26% to 23.97% and similarly the L2 misses also reduce. The reduced cache misses helps in making the search algorithm faster. The analysis of speed of execution is given in the last subsection.

4.4 Array Implementation - Encode characters instead of end index

In this section, we will analyze the impact of encoding characters directly within the node rather than having two indices, start and end, which points to the offset in the original string represented by the corresponding edge of the suffix tree. If we store the offsets and not the characters directly, the disadvantage is that for every access of the node, in order to find the substring represented by the edge coming into the node, we have to refer to the original string. The reason for representing this information as an offset is that it helps in having a constant sized field rather than a variable one for representing information whose length can vary between 1 and potentially the size of the string itself. When we analyze the suffix tree, we find that most of the edges have edgelength less than twelve. The table in Figure 4.5 gives the details. We see that around 40% of the edges in the suffix tree have less than or equal to 12 characters. Every time node is accessed, instead of accessing the original string to find out the characters, the characters themselves can be encoded instead of the end index field. The first 24 bits can be used to store this information. The next 8 bits is used to store the length of the edge (0 to 24). When the node is accessed along with the start information, the characters are also fetched and this helps us to save memory accesses to the original string.

We analyze the cache miss reduction due to the two improvements, i.e the elimination of leaf nodes by placing the information in the parent node and the encoding of character information in the node itself. The table below shows the cache misses in the basic array implementation with no optimizations and the fully optimized data structure implementation.



Size of input string	% of edges < 12 characters
10MB	38.38%
15MB	38.40%
20MB	38.35%

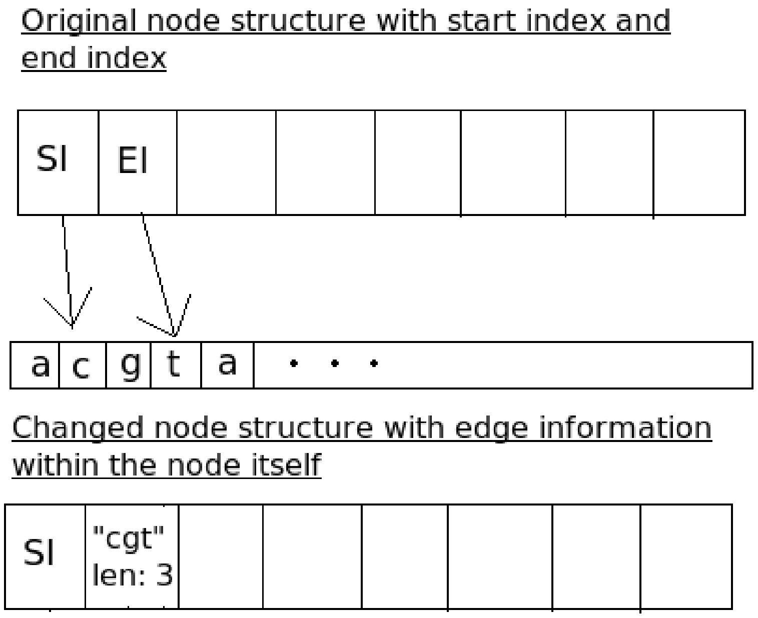

Original node structure with start index and end index
SI
EI
a
c
g
t
a
• • •
Changed node structure with edge information within the node itself
SI
"cgt"
len: 3

Figure 4.5: Edge information inside node

(Intel params)	Basic Array Impl	Optimized
L1 cache misses	38.82%	22.56%
L2 cache misses	97.61%	91.05%

(AMD params)	Basic Array Impl	Optimized
L1 cache misses	30.26%	21.26%
L2 cache misses	98.72%	91.58%

The experiments had address traces of maximal substring search of 10000 queries run on a suffix tree with total of 10 million characters long string. We observe for both intel cache parameters and AMD cache parameters the cache miss percentage is less for the optimized data structure as compared to the base array implementation. We observe that the L1 misses show significant reduction, but there is not much reduction in L2 misses. The L2 misses are still high. The most likely reason for this is that the spatial and temporal property are exploited at the L1 level itself and the L2 cache is not able to exploit any locality in the filtered memory accesses. In the next subsection we analyze the impact of reduced cache misses on speed of execution of the search algorithm.

We can compare this results to the optimal cache miss rate if the sequence of access for a given query string is already known. This means, we assume that the query string is already known and in such a case try to align the nodes so that it gives minimal cache misses. Assuming that every internal node is around 25 bytes and cache line is 128 bytes, 5 nodes can be fitted into a cache line. This implies that one in five nodes will have a miss and so the cache miss will be around 20% in the optimal case. Also note that the cache misses in the two tables also include the top portion of the tree being reused from the caches due to temporal property across query strings. That is the reason why the L2 cache miss shows around 90% miss, when in theory, for just one query string, there will be no spatial locality property that can be exploited by the L2 cache and cache miss would be 100%.

4.5 Padding

In this work, we assume that whenever a node is accessed then the whole node falls within the same cache line itself. But there may be cases when the node gets loaded in such a way that one part of the node is in one cache line and the other part is in the other cache line. To prevent this we use the concept of padding. As the nodes are being assigned memory space we check to see if it falls withing a cache line or would fall in the boundary of two cache lines. For the latter case, we allot a padding memory space so that the node being alloted falls in a new cache line.

The results after padding were similar to the results presented in the previous section. This means that the number of nodes that fell on the cache line boundaries are small compared to the total number of nodes.

4.6 Improvements w.r.t speed of execution

We compare the time taken by different implementations for the search algorithm. The suffix tree is built over 10MB, 15MB, 20MB string and then maximal substring search is applied. The time taken for the search algorithm is recorded. The graph in Figure 4.6 shows the details. The different implementations are

- Linked List Implementation
- Basic Array Implementation
- Trellis Implementation
- Optimized Implementation

The trellis implementation is basically a DFS based layout and also has leaf information in its parent. It was originally developed as a disk layout in [5] and here its used as a main memory layout for comparison with the rest of the implementations.

The graph clearly indicates that the time taken by the optimized implementation brings down the execution time by approximately 70% as compared to the linked list implementation and also performs better than the other implementations. The reduction has been achieved because of the reduced cache misses.

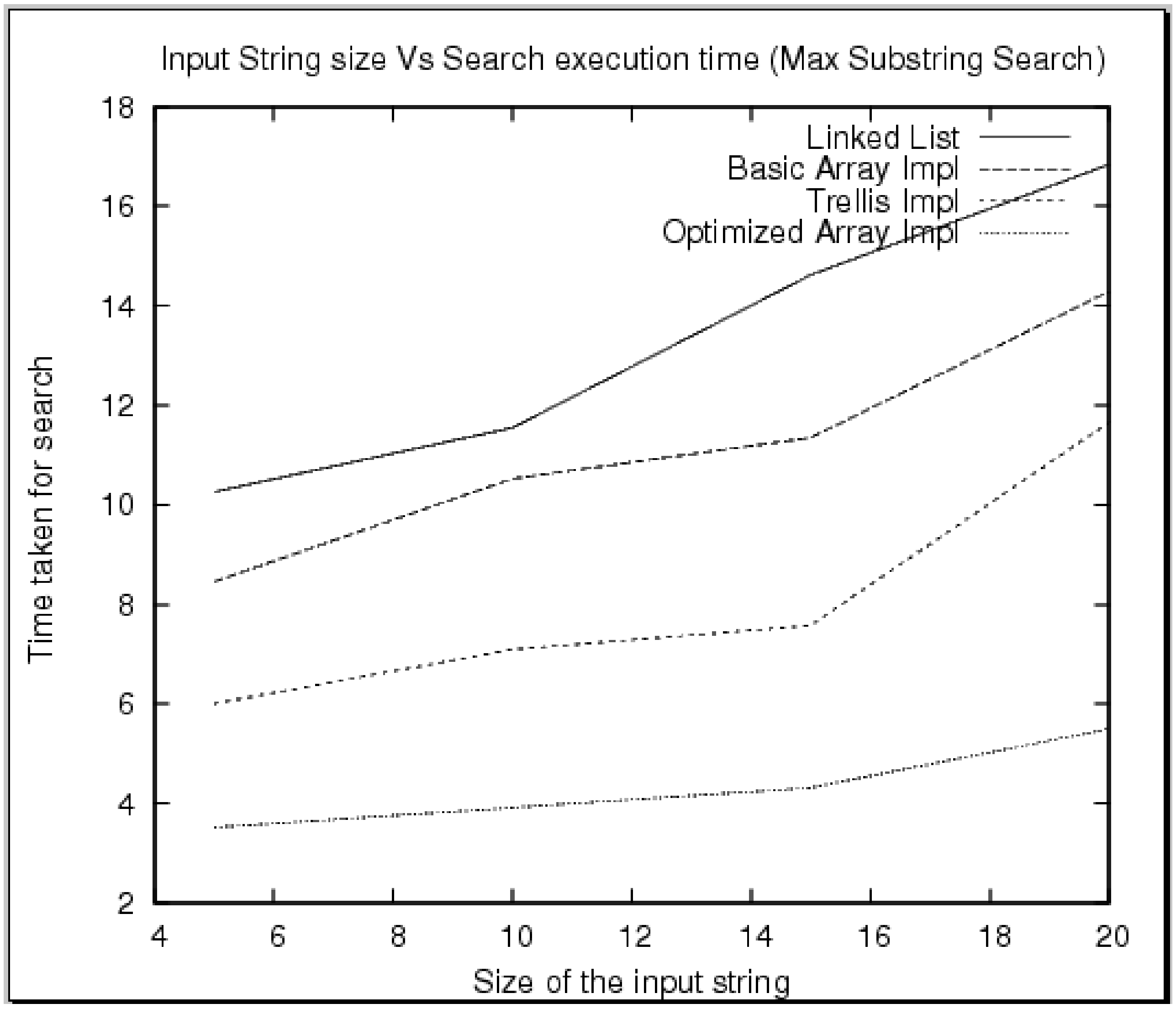


Figure 4.6: Execution Time Vs Size of input string

Chapter 5

Conclusions

In this report, we have analyzed the cache consciousness of suffix trees. Different techniques like layout changes, data structure changes were implemented. In layout change, Creation order layout, Breadth First Layout, Trellis Layout, Stellar layout were experimented with. However the cache line sizes were shown to be too small to exploit spatial locality for the search algorithms. In data structure changes, first we showed that linked list implementation though conserves space per node, is not ideal for searching as more nodes need to be traversed as compared to the array implementation. Three changes were done to the basic array implementation to make it more cache conscious. The leaf information was placed in the parent node itself, the null pointers were eliminated and the substring representing edges were directly encoded in the node itself rather than placing the offsets to the original string. Due to these changes we obtain around 70% gain in search time in the optimized version as compared to linked list or basic array implementation.

As future work, the following points can be considered.

- Grouping of nodes so that only one pointer points to a group of contiguously allocated nodes
- Other search algorithms can be implemented and checked if they give the similar results to maximal substring search.

- The implementions can be extended to disk level layouts and checked if the results hold good for disk to main memory hierarchy
- With reduced sized nodes, the behaviour of different layouts can be experimented

References

[1] Srikanta B.J. BODHI: A Database Engine for Biological Applications - PHD Thesis, DSL Lab, 2006.

[2] Dan Gusfield book. Algorithms on Strings, Trees and Sequences.

[3] Srikanta Bethadur and Jayant Haritsa. Engineering a fast online persistent suffix Tree Construction, ICDE 04.

[4] Srikanta Bethadur and Jayant Haritsa. Search Optimized Persistant Suffix Storage, HiPC 05.

[5] Benjarath and Mohammad Zaki. Genome-scale Disk-based Suffix Tree Indexing, SIGMOD 2007.

[6] Dinero Cache Simulator download page. http://pages.cs.wisc.edu/ markhill/DineroIV/

[7] S. Tata, R. Hankins, and J. Patel. Practical suffix tree construction.

[8] P. Weiner. "Linear pattern matching algorithm". IEEE Symposium on Switching and Automata Theory, 1973.

[9] McCreight. A space–economical suffix tree construction algorithm , ACM Journal 1976.

[10] E. Ukkonen. On-line construction of suffix trees. Algorithmica, 1995.

[11] Ela Hunt et al. Persistent Suffix Trees and Suffix Binary Search Trees as DNA Sequence Indexes, 2000.

[12] Anastassia Ailamaki et al. Weaving Relations for Cache Performance, VLDB 2001.

[13] J. Rao and K. Ross. Making B+ Trees Cache Conscious in Main Memory, SIGMOD 2000.

[14] Stefan Kurtz. Reducing the space requirements of Suffix Trees, Software -Experience and Practice, 1999.

[15] Amol Ghoting et al. Cache Conscious Frequent Pattern Mining, VLDB 2005.

[16] A. Ailamaki, D. J. DeWitt, M. Hill, and D. Wood. Dbms on a modern processor: Where does time go?, VLDB 1999.

[17] Naresh Neelapala, Jayant Haritsa. SPINE: Putting backbone into String indexing, ICDE 2004.

[18] Vtune performance analyser download page. - http://www.intel.com/cd/software/products/asmo-na/eng/vtune.

[19] Home page of valgrind simulator. http://valgrind.org.

[20] Amol Ghoting et al. Cache-conscious Frequent Pattern Mining on a Modern Processor, VLDB 2005.

[21] Knuth Morris Pratt algorithm. http://en.wikipedia.org/wiki/Knuth_Morris_Pratt_algorithm.