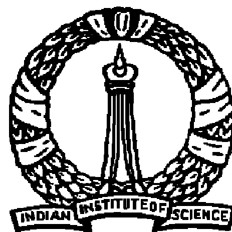
Online Construction of Search-Friendly Persistent Suffix-tree Layouts

A Project Report
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Engineering
in
Computer Science and Engineering

by

Akinapelli Sandeep



Department of Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

JULY 2006

Dedicated

To

Parents and Family

For their unconditional love and support

Acknowledgments

I take this opportunity to express my deepest gratitude to my guide Prof. Jayant Haritsa for his excellent guidance and persistent encouragement throughout the course of this work. I am thankful to B.J. Srikanta for his help and valuable suggestions at various stages of my project work. I am also indebted to my fellow DSLites Akshat, Aslam, Gopal and Prateem for their wonderful company and help. Several people have helped me in one way or the other and made my stay in the institute a pleasant and unforgettable experience. I sincerely thank all my friends who helped me directly or indirectly in completing this project.

Abstract

Sequence databases enjoy widespread use in modern life science applications and querying sequences is a common and critical operation in many of those applications. The suffix tree is a well known data structure that can be used to evaluate a wide variety of queries on sequence data sets. suffix trees are well-known to be not easily amenable to persistent implementation and usage, since the search procedures used in various applications exhibit complex traversal patterns since suffix links are also involved and hence induces severe disk thrashing. Stellar is a recently proposed mechanism for creating an efficient disk layout for searching on sequence databases. The generation of stellar layout is done post facto i.e., after building the suffix tree using other algorithms. This is not only time consuming, but also involves lot of additional disk I/O. In this thesis we propose a new algorithm which we call GULP that directly produces an approximate stellar layout. Many construction algorithm construct suffix trees without suffix links to avoid skew problem. Unlike these approaches GULP constructs suffix trees with suffix links and does not exhibit skew problems. Our experimental results on different real genomic sequences indicate that GULP beats the ukkonen construction algorithm by orders of magnitude.

The suffix tree construction and layout algorithm uses fixed node structures. However the layout produced by GULP/Stellar is amenable to node compression which can reduce the amount of disk space used by suffix tree. We propose a compression scheme that exploits the localities of the layout, which not only reduces the amount of disk space required for storing the tree but also improves the search performance over existing layout. we show that the combined effect of these optimizations results in substantially improved index construction and search times.

Contents

Acknowledgments **i**

Abstract **ii**

List of Tables **v**

List of Figures **vi**

1 Introduction **1**
1.1 Suffix Tree 2
1.1.1 Suffix links 3
1.2 Searching Suffix Tree 5
1.2.1 Locating Maximal Common Substrings over UST 6

2 Related Work **8**
2.1 Suffix Tree Layout 10
2.2 Issues in Suffix-Tree Layout 10
2.3 Stellar Layout 12
2.4 Complexity of traversal patterns 12

3 Online Construction Algorithm **15**
3.1 Node expansion 18
3.2 Suffix link computation 19
3.3 Time Complexity 22
3.4 Experimental Framework and Results 23
3.4.1 Query Workload 23
3.4.2 Construction times 25
3.4.3 Structural Localities of the Layouts 25
3.4.4 Search Performance 26
3.4.5 Conversion to Stellar 28
3.4.6 Comparison With TDD 28

4 Compact Suffix Tree **31**
4.1 Motivation 31
4.2 Compressed Node Structure 33

4.3 Compressed Stellar Layout 34
4.4 Experimental Framework and Results 36
4.4.1 Search Performance of CS-Layout 37

5 Conclusions **40**

Bibliography **41**

List of Tables

1.1 Notations 5

3.1 Structural Edge and Link localities 26

4.1 Structural Edge and Link localities of CS-Layout 37

List of Figures

1.1 Suffix-tree over a DNA fragment GTTAATTACTGAAT$ 3
1.2 Suffix-tree over a DNA fragment GTTAATTACTGAAT$ with suffix links 4

2.1 Stellar Algorithm 11
2.2 Traversal Pattern of Maximal Substring using construction order layout 13
2.3 Traversal Pattern of Maximal Substring using Stellar layout 14

3.1 Node structures used in GULP 16
3.2 GULP algorithm 17
3.3 Problem in Basic Algorithm 19
3.4 GULP algorithm with Node expansion rule 21
3.5 The traversal of *getSuffixLink()* 22
3.6 Construction times 24
3.7 Search times of Stellar layout vs GULP 27
3.8 Disk I/O of Stellar layout vs GULP 27
3.9 Performance of conversion to Stellar layout 28
3.10 Search Performance of GULP and TDD on hEST100 29
3.11 Search Performance of GULP and TDD on hEST200 29

4.1 Distribution of length field 32
4.2 Compressed Node structures 33
4.3 PageMapper 35
4.4 Compressed Stellar layout Algorithm 36
4.5 Disk space savings using compressed scheme 36
4.6 Search times of Stellar layout vs CS-Layout 38
4.7 Disk I/O of Stellar layout vs CS-Layout 38

Chapter 1

Introduction

With the advent of high throughput genome sequencing techniques, biological sequence data is being generated at speeds exceeding the growth of modern day computational speeds. Recently GenBank [38] crossed the 100 Gbp mark, with sequences from over 165,000 organism. Likewise, the Swissprot [39] protein sequence database currently records about 75 million amino acids, from over 200,000 sequences. This is expected to grow exponentially with ever expanding applications of genome sequence analysis. One of the most important computational tasks on this voluminous amount of sequence data is that of efficiently locating all the matches in the database to a given pattern sequence. As a consequence of the enormous datasize and exponential growth rate, it has become critical to have effective data structures and efficient algorithms for storing, querying and analyzing these sequence data.

In these tasks, the matching criteria could be either exact or inexact based on a similarity metric. The area of bioinformatics, which has developed almost independently of database research, has considered the suffix-tree data structure (along with its variants such as suffix-array, DAWG, etc.) as the defacto standard for preprocessing of the genomic sequences. This is mainly due to the adaptability of suffix-tree to solve many sequence processing problems that are otherwise computationally extremely hard to solve. In addition, suffix-trees have linear time and space complexity, which make them attractive for use with large scale sequence processing tasks. A suffix-tree is a data structure that exposes the internal structure of a sequence in a deeper way than any other datastructure such as inverted index.

1.1 Suffix Tree

Let $S = s_1 s_2 .. s_n$ be a sequence of length n with each s_i is drawn from an alphabet $\sum$. A substring of the string S is a string $S[i..j] = s_i s_{i+1} ... s_j$ for some $1 \leq i \leq j \leq n$. A suffix of the string S is a substring such that j = n ; i.e., it is a part of the string starting at any location i in the string, continuing upto the end of the string. We represent a suffix starting at position i as S_i. Thus, there are exactly n suffixes from a string of length n, one for each position in the string.

Definition 1.1 *A suffix-tree T_S for an n-character string S is a rooted directed tree with exactly n leaves numbered from 1 to n. Each internal node other than the root has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix-tree is that for any leaf numbered i, the concatenation of the edge-labels on the path from the root to that leaf exactly spells out the suffix of S that starts at position i. That is, it spells out S_i.*

If there is a suffix S_i that *exactly* matches another *substring*, $S[j \ldots k]$ for $j \neq i$, then S_i ends at a non-leaf. In order to overcome this, a *delimiter* symbol, denoted by $, is concatenated with the string. It is assumed that $ does not appear anywhere else in the string, and $ $\notin \Sigma$. With this assumption, it is guaranteed that there is a clear disambiguation between leaf and internal nodes. The leaf node corresponding to the i-th suffix, S_i, is represented as l_i. An internal node, v, has an associated length $L(v)$, which is the sum of edge lengths on the path from root to v. We represent by $\sigma(v)$, the string at v, to represent the substring $S[i..i + L(v)]$, where l_i is any leaf under v.

The suffix-tree for a DNA fragment `GTTAATTACTGAAT$` is shown in Figure 1.1. The dark nodes are the internal nodes and the lightly shaded nodes are the leaf nodes. Each edge has an associated label, which is a substring of the string $S$$, and entry under each leaf node is the index i associated with the suffix corresponding to the leaf node.

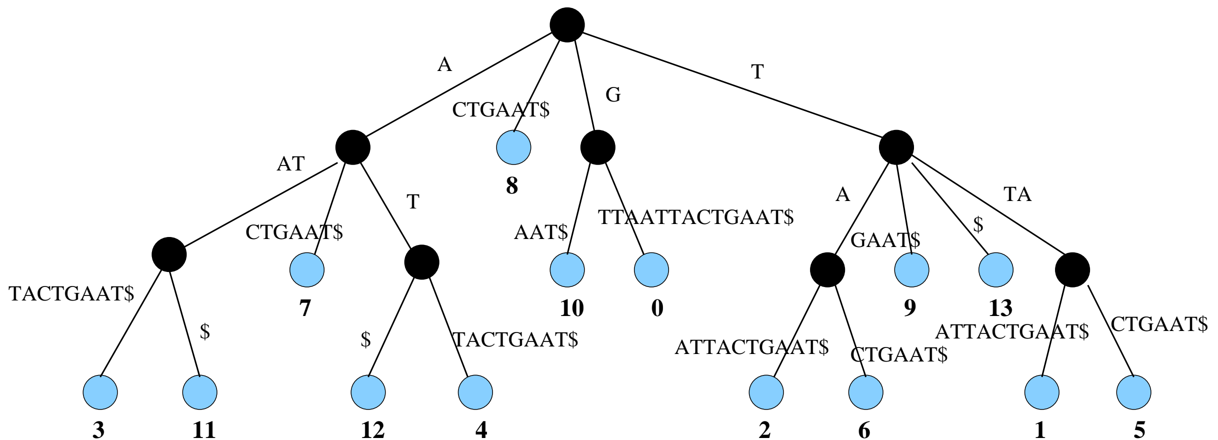


Figure 1.1: Suffix-tree over a DNA fragment GTTAATTACTGAAT$

1.1.1 Suffix links

One important structural augmentation to suffix tree are the suffix links which are crucial for linear time construction of suffix trees. Suffix links are edges (or pointers) that span across the suffix tree, between two internal nodes which may not be related through a parent-sibling relationship.

Definition 1.2 *Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denote a possibly empty substring. For an internal node v with path-label $x\alpha$, if there is another node s(v) with path-label α, then a pointer/edge from v to s(v) is called a suffix-link.*

The suffix-link of the root of a suffix-tree is defined to be pointing to itself. Other than this, the suffix-links are well defined for all the internal nodes. Figure 1.2 illustrates the suffix-tree with suffix-links, built over a genome fragment GTTAATTACTGAAT$. The dotted lines between internal nodes of the tree are the suffix-links, with the direction of the arrow indicating the pointer direction.

The suffix-links, in the present form, were first introduced by McCreight[27] and since then they are implicitly assumed to be present in the suffix-tree. In addition to the linear time construction, the presence of these links enable a much richer set of traversals over the suffix-tree resulting in many high-speed search algorithms [9, 35]. On the other hand, suffix-links are also considered a source of additional space overhead, and more significantly, a reason for poor locality properties of suffix-tree construction and search algorithms.

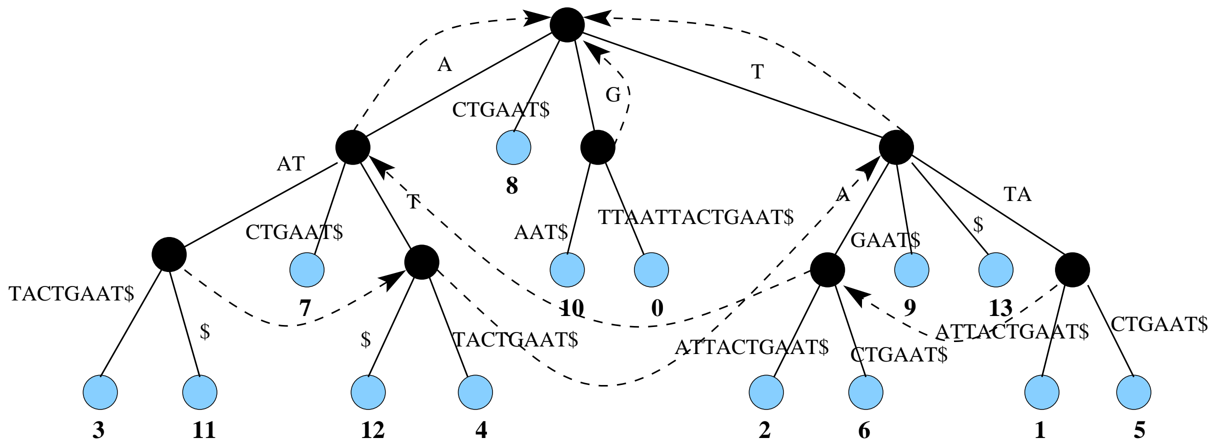


Figure 1.2: Suffix-tree over a DNA fragment `GTTAATTACTGAAT$` with suffix links

In this Report, we distinguish between these two structural variants of suffix-trees based on the presence of suffix-links as follows:

Un-linked Suffix-Tree (UST). This is the suffix-tree that strictly adheres to Definition 1. The suffix-links in the UST have been either dropped post-construction, or the tree has been constructed using algorithms that do not need suffix-links.

Linked Suffix-Tree (LST). In contrast to USTs, the LSTs retain the suffix-links in the tree providing much richer traversals across the suffix-tree.

We focus mainly on the construction and search performance of persistent version of LSTs. Hence, unless mentioned explicitly, we use the terms LSTs and suffix-trees interchangeably.

For ease of reference, Table 1.1 summarizes the terminology associated with suffix-trees, used in this thesis.

Unlike the traditional database indexes whose size is typically a fraction of the database contents, suffix-tree size is usually much larger than the underlying sequence data. As an example, the entire 3 Gbp of Human Genome is fully representable in about 1 GB memory where as the most space economical suffix tree occupies close to 25 GB. Though we can host the entire sequence in memory, suffix tree needs to be disk resident.

S	Sequence of length N
Σ	Finite alphabet of symbols
$	Delimiter symbol such that $\$ \notin \Sigma$
s_i	Symbol at position i in S, drawn from Σ
$S[i \ldots j]$	Substring of S starting at position i and length $(j - i + 1)$
S_i	Suffix of the sequence S starting at position i
T_S	Suffix-tree built for the sequence $S\$$
l_i	Leaf in the suffix-tree T_S corresponding to the suffix S_i
$L(v)$	Path length of a node v, the sum of edge lengths on the path from root to v
$\sigma(v)$	Substring $S[i \ldots i + L(v)]$ associated with node v in the suffix-tree
$sl(v)$	Suffix-link of the internal node v

Table 1.1: Notations

1.2 Searching Suffix Tree

Suffix tree is an important data structure for indexing text string since we can search for patterns in the data efficiently and the search complexity is independent of the original text string size. There exist many practical applications that rely on suffix tree, especially for processing biological sequence data [21]. As various genome sequencing projects are ongoing and more genome sequences are made known, the application of suffix tree on biological research is expected to increase. Biological applications use various search procedures among which, some searches involve traversing only the tree-edges, while some other searches involve traversing both the tree-edges and the suffix links. Two of the applications are listed below.

Definition 1.3 ***Exact String Matching:*** *Given a text T and a pattern P the exact string matching problem is to find all occurrences of P in text T. After building the suffix trees the problem can be solved in $O(|P|)$, where $|P|$ is the size of the pattern.*

Exact string matching is used in matching against a DNA or protein library of known patterns.The problem of finding which STSs (sequence tagged sites, a DNA string) are contained in an anonymous DNA is a good example for an exact string matching problem. This search algorithm involves traversing only the tree-edges of the suffix tree. Since this search does not involve suffix links USTs and LSTs serve the same purpose.

Definition 1.4 ***Maximal Substring Search:*** *Given a database sequence S, and a query sequence Q locate all k such that such that* $1 \leq i, i+j \leq |Q|$, $1 \leq k, k+j \leq |S|$, $j > 0$ *and* $Q[i..i+j] = S[k..k+j]$ *and* $Q[i+j+1] \neq S[k+j+1]$ *and there exist no* $p > j$*,such that* $Q[i..i+p] = S[k..k+p]$. *In practice it is desired that only matches that satisfy a user defined minimum threshold length,* λ *are reported that is (*$j \geq \lambda$*).*

To further illustrate this, consider the database sequence GTTAATTACTGAAT$. Now, given a query sequence CTAATGACT, with threshold λ set to 3, the desired common maximal substrings between the database sequence and the query sequence are: $\{$TAAT(3), AAT(4,12), TGA(10), ACT(8)$\}$. Note that although CT is a common substring, it is not reported since it does not satisfy the match length restriction.

This search algorithm involves traversing both the tree-edges and the suffix-links. Maximal Substring search has been implemented in various popular genomic softwares like MUMMER and BLAST. Since this kind of traversal needs suffix links, these kind of searches can not be carried out efficiently on USTs. The difference in the performance can be orders of magnitude due to the large size of suffix trees.

1.2.1 Locating Maximal Common Substrings over UST

In order to locate all the maximal common substrings between S and Q when S has been processed into an UST (Unlinked Suffix-Tree) , we use the observation that every common substring must result in a prefix match between corresponding suffixes in S and Q. This leads us to the following algorithm use each suffix of Q to walk down the suffix-tree from the root node, until either the suffix is completely located or there is a mismatch. If the length matched is greater than the value of λ, then add to the output set, L, all the leaf nodes under the current location. Follow this process for all the suffixes at positions from 0 to $|Q| - \lambda + 1$. We refer to this algorithm as MSS_{UST} in the rest of the thesis.

Genome databases are growing in order of gigabytes and hence maintaining suffix trees becomes an important issue. There are two immediate problems: The first problem is on constructing the suffix tree efficiently. Many suffix tree construction algorithms have been proposed over the years[27, 34, 37]. These algorithms will scale provided the entire tree fits

in the memory. Also, the use of suffix links, which are a key feature in obtaining the linear construction time, can result in poor locality of reference [23, 32]. To address this issue, several disk-based suffix tree algorithms have been proposed in the last few years. Some of these approaches [23, 32] completely abandon the use of suffix links and sacrifice the theoretically superior linear construction time in exchange for a quadratic time algorithm with better locality of reference. However, due to the abandoning of suffix links many fast string processing algorithms that make use of suffix links, such as computing matching statistics [9], tandem repeats[22], structural motifs[8] and genome alignment[15] are rendered unusable.

The second problem is on accessing the suffix tree. There are a number of disk based representations of suffix trees [11, 23, 29, 32] proposed in the literature. However, these disk-based suffix trees either fail to support all general suffix tree operations or have high I/O disk access for certain operations.

Optimizing the suffix trees for search aspect poses new problems:

- The patterns of search traversals over suffix trees are much more complex than those found in traditional index structure, since both tree edges and suffix links are involved.
- The presence of suffix links turns suffix trees into cyclic structure.
- Unlike the typical indexing structures like B-tree, suffix trees are not inherently balanced.

This thesis focuses on addressing the two problems simultaneously by designing a search-conscious and efficient construction algorithm over DNA sequences. We also propose a compression scheme, having a practical and efficient suffix tree implementation on disk that supports various suffix tree operations efficiently. We use optimized bit-packing scheme based on several observations to yield a compact suffix tree which takes less disk space and also provides good search performance.

Chapter 2

Related Work

Although much attention was given on how to efficiently construct the suffix tree, search performance of suffix tree has drawn very little attention. Hunt et.al. [23] gave a disk-based suffix tree construction for DNA and protein sequences and shows that dynamic programming over the suffix tree can be efficiently executed with less than 0.3% of the expected matrix being evaluated. The drawback of their approach is that, their suffix tree is large, requiring 21N to 65N bytes depending on implementation. Tata et. al. [32] gave an improved top-down disk-based construction algorithm named TDD for suffix tree that can scale up efficiently to large text sequence while using a fixed memory space. But all these techniques involves constructing UST(unlinked suffix tree) i.e., suffix tree without suffix links.

There exists I/O efficient implementations of suffix tree for exact string match queries which includes String B-Tree(SB-Tree) [17] and compact Pat Tree(CPT)[11]. SB-Tree applies the structure of B-tree over string to give a well-balanced tree structure that promises $O(log_B N)$ worst case I/O accesses to traverse the tree from the root to a leaf node. As SB-tree does not explicitly preserve the suffix tree structure, it is not obvious if SB-tree can be extended to handle complex query like approximate search efficiently. This limits the usefulness of SB-tree in practice. OASIS [29] is a dynamic programming A* search driven algorithm for local alignment on protein sequences that surpass the performance using smith Waterman algorithm [31] for exhaustive search. None of the above results address the issue of I/O efficiency in handling suffix tree operations which is the main focus of this thesis.

Suffix array(SA) is a reduced version of a suffix tree that basically stores the suffix positions in a list by sorting the suffixes in lexicographical order. It uses only 4N bytes space at the cost of additional $O(logn)$ time factor for binary search in matching a given string to the text as the access of suffix array does not display a regular pattern, the disk I/O cost for having the suffix on disk can be as high as $O(Nlogn)$. The naive suffix arrays does not support complex searches like maximal substring search. Abouelhoda et.al. [1] have proposed enhanced suffix arrays and showed how suffix arrays can be extended to make provision for all complex search patterns, but they haven't studied the disk performance of their structure.

Tree layout was well studied in the literature [2, 16, 20] but all of those studies are restricted to conventional trees, where there are no structures like suffix-links and also for the search procedures where search starts from root and ends at leaves. Diwan et. al. [16] have considered the problem of packing trees in order to minimize the total disk accesses given an access distribution on leaf nodes-i.e., average path-length minimization. They have shown that a heuristic-based linear-time algorithm henceforth called SBFS that does recursive localized breadth-first layout of the tree not only outperforms classic tree-layout methods such as Breadth-first and Depth-first strategies, but also results in an I/O-cost that is within a small factor of the optimal quadratic-time layout algorithm. Suffix tree with links can be viewed as a graph layout problem, but it has been proven that graph layout is NP-Complete [16]. Moreover due to the huge size of suffix trees those algorithms become impractical.

It has been shown[6] that Ukkonen algorithm when run produces static layouts on the disk which exhibits good link locality but poor edge locality. Stellar is a recently proposed layout algorithm [6], which tries to localize the suffix-links and tree-edges within the same page. Stellar does a recursive BFS starting from root and at every node all its children and the suffix-links of the children are packed into a single page. Stellar is a post-facto technique. i.e, After the tree is built Stellar is applied on the tree to produce a new layout on the disk. This technique involves lot of random DISK I/O. In this paper we present a new approach to efficiently construct the LST(Linked Suffix tree) i.e., suffix tree with links, at the same time producing a good layout on the disk.

2.1 Suffix Tree Layout

Suffix-trees, unlike popular index structure such as B-Trees, are not inherently balanced. Their structure depends entirely on the combinatorial characteristics of the indexed sequence. In the worst-case the tree can degenerate into a linear chain of internal nodes.

The fan-out of each internal node of a suffix tree is upper bounded by the size of the alphabet of the indexed sequence. Therefore the common strategy of customizing the fan-out to suit the disk-page size cannot be adopted here. This means, multiple nodes of a suffix-tree will be stored on a page with nodes connected both within as well as across pages. Therefore it is very important to have a layout of nodes on the external memory for efficiently processing the search procedures.

2.2 Issues in Suffix-Tree Layout

The storage layout of suffix trees introduces a variety of novel issues.

Structural complexity: Suffix-trees exhibit greater inherent structural complexity than typical tree index structures due to the presence of cyclic sub-structures. Specifically the collection of tree-edges as well as the collection of suffix-links in a suffix-tree form two separate tree structures, with a common root. Also note that in the tree structure induced by the collection of suffix-links, the traversal direction between nodes are reversed from the natural parent-to-leaf direction. That is there exists a directed path starting at any internal node to the root of the suffix-tree, via a chain of suffix-links and from the root node, any of the internal nodes are reachable through a chain of tree-edges, thus completing a cyclic path.

Complex Traversal Patterns: In typical index structures, the queries are mostly lookup searches involving root-to-leaf traversals. But some search algorithms over suffix-trees exhibit complex traversal patterns, involving simultaneous use of tree-edges and suffix-links. Thus the layout strategy has to take into account the two *orthogonal* traversal paths during search.

Algorithm 2.1 Stellar (r, B)
Input
r : Root of the subtree to be traversed
B : Capacity of the disk-page in terms of no. of nodes
Output
An ordering of the subtree under r

1: $queue \longleftarrow r$; {push root into the BFS queue}
2: $nodecount \leftarrow 0$; {initialize the counter}
3: **while** $queue$ not $\emptyset$ **do**
4: $r' \longleftarrow queue$; {remove head of the $queue$}
5: **if** r' not visited **then**
6: mark r' as visited and increment $nodecount$;
7: **end if**
8: **for all** c such that c is a child of r' **do**
9: $s \leftarrow sl(c)$;{s is the suffix-link of c}
10: **if** c not visited AND $nodecount < B$ **then**
11: mark c as visited and increment $nodecount$;
12: **end if**
13: $queue \longleftarrow c$;
14: **if** s not visited AND $nodecount < B$ **then**
15: mark s as visited and increment $nodecount$;
16: $queue \longleftarrow s$;
17: **end if**
18: **end for**
19: **if** $nodecount \geq B$ **then**
20: **while** $queue$ not $\emptyset$ **do**
21: $m \longleftarrow queue$;
22: Stellar(m,B);
23: **end while**
24: **end if**
25: **end while**

Figure 2.1: Stellar Algorithm

2.3 Stellar Layout

Stellar is a recently proposed[6] layout mechanism that organizes the suffix tree in the disk in order to efficiently process various search procedures on the suffix tree. Stellar utilizes the structural relationships of the tree and does a recursive BFS starting with root node and tries to localize edges and links. A pseudocode of Stellar algorithm is presented in Algorithm 2.1. 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 are treated as a "buddy" 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.

The complexity of the Stellar algorithm is linear in terms of the number of nodes the suffix tree contained. However Stellar is a post facto algorithm, i.e., Stellar layout is applied only after the suffix tree has been built using other well known algorithms like Ukkonen. The disadvantages of this approach is we need additional disk space that accounts for the following two factors. We need to keep both the source tree and the tree that is produced by stellar layout in the disk. The mapping structures that are needed to produce the stellar layout is linear in the size of suffix tree and hence needs to be disk-resident. Also the conversion time to stellar is expensive compared to the construction time of the suffix tree. Hence we need an online algorithm that directly produces the stellar layout.

2.4 Complexity of traversal patterns

To Understand the complexity of the traversal pattern, we have plotted the behavior of the maximal substring search in Figures 2.2 and 2.3. The X-axis represents the order of accesses and the Y-axis represents the current logical block number, that is being accessed while the

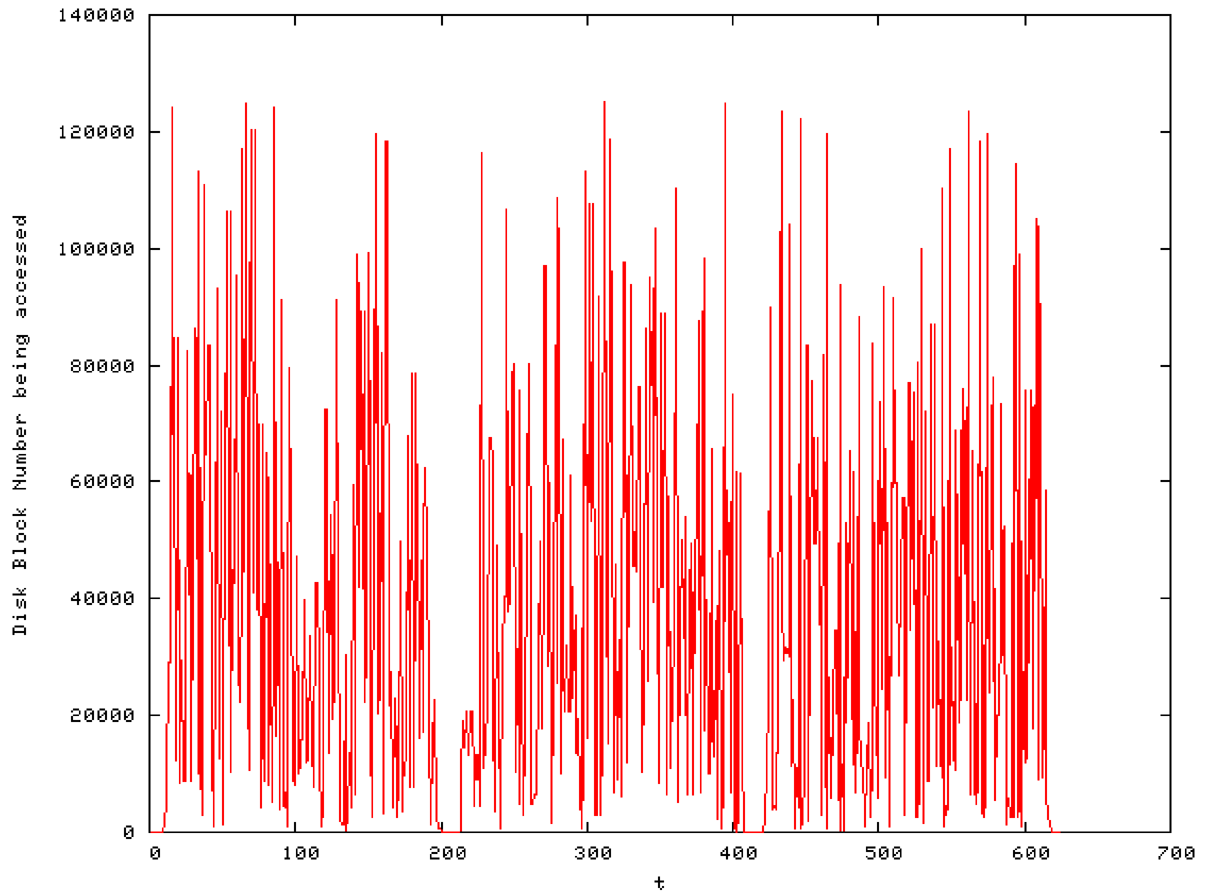


Figure 2.2: Traversal Pattern of Maximal Substring using construction order layout

search process is going on. The Figure 2.2 shows the traversal pattern using the default layout. i.e., The layout produced by the Ukkonen algorithm, which we call CO(construction order) layout. As evident from the figure, the traversal pattern is quite complex and pseudo random. Each of the peaks in the figure represents a different query of length 100. It is observed that the variations in each of the peak is high and also there is no correlation across queries. Unless we have a good layout on disk, the search process induces random disk I/O. The Figure 2.3 shows the traversal pattern of the maximal substring search when the tree is laid in the disk using stellar layout. There are two observations that can made from this figure. The width of the peaks is less indicating the less amount of Disk I/O. Also, the variations among each of the peaks is also small, which implies that the tree produced by stellar layout induces less random disk I/O compared to the CO layout. Hence it is very important to have a good layout of suffix tree on the disk in-order to efficiently process the sequence queries.

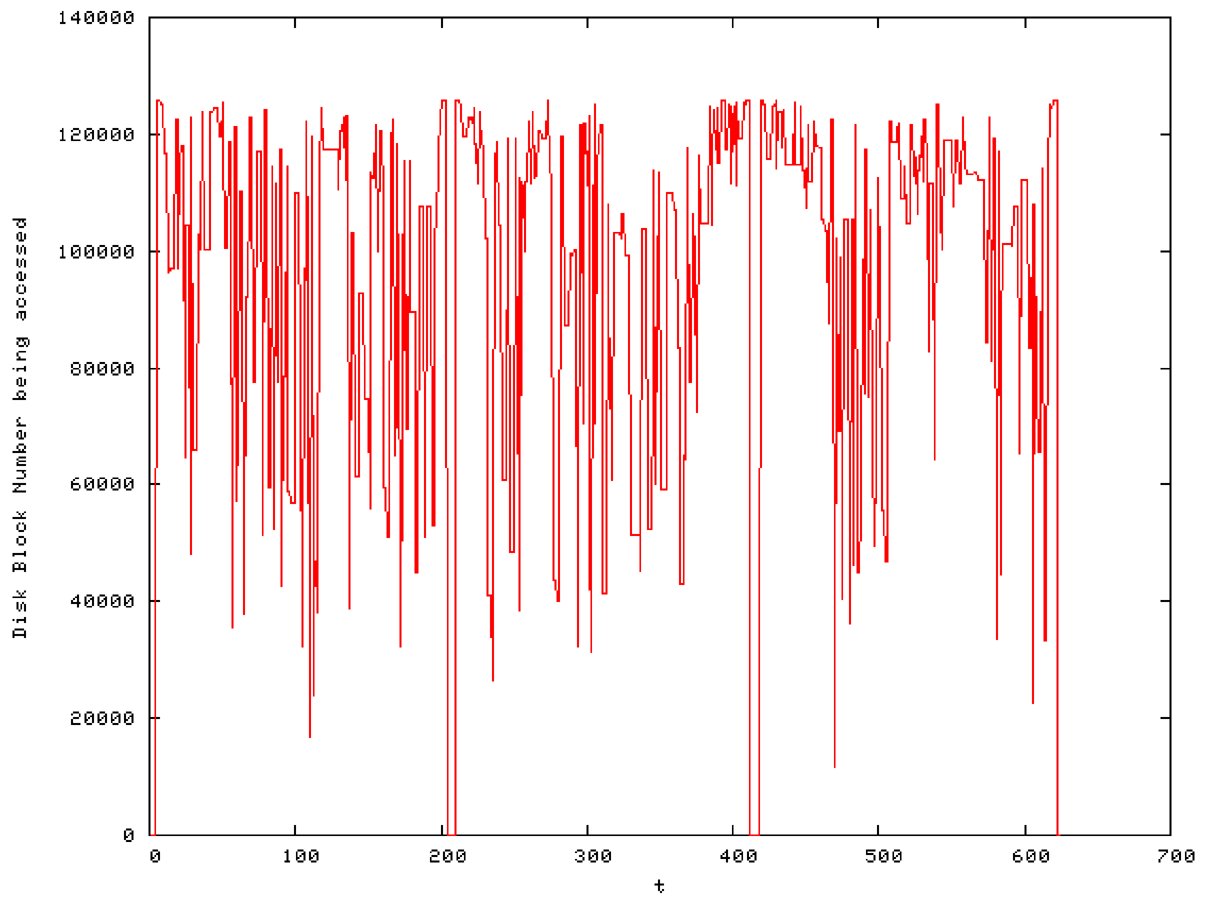


Figure 2.3: Traversal Pattern of Maximal Substring using Stellar layout

Chapter 3

Online Construction Algorithm

We now present a new algorithm which we call GULP (**G**enerating s**U**ffix trees with **L**inks using **P**refix partitioning) for construction of LST. The objectives of this new technique are two-fold. 1) To construct the LST efficiently 2)Come up with good disk layout while constructing the suffix tree itself in order to minimize the disk I/O during the search process.

The suffix tree is represented using the node structure shown in Figure 3.1. We use the array based implementation which is shown to be more efficient than the linked list implementation[4]. As shown in the figure, there are 3 different node structures namely Leaf node, Internal node and Auxiliary node. Leaf node contains 2 fields 1)Character Label, the first character of the incoming edge 2)Begin offset, which is an index into the string that denotes the suffix starting at that position.

The internal node has 5 fields. The character label is the the first character of the incoming edge label. The begin offset, which is an index into the string that represents the starting position of the edge label in the sequence string. The length field holds the length of the edge label of the incoming edge. The begin and length field together used to identify the edge label of the incoming edge. The suffix link field holds the pointer to the suffix link of the node. The child pointers store the pointers to children of the node. For DNA alphabet it has entries for each letter of the alphabet A,C,G,T.

The auxiliary node structure is a special structure used by GULP. The size of the auxiliary node is dependent on the alphabet size of the sequence. For an alphabet of size $|\Sigma|$ auxiliary

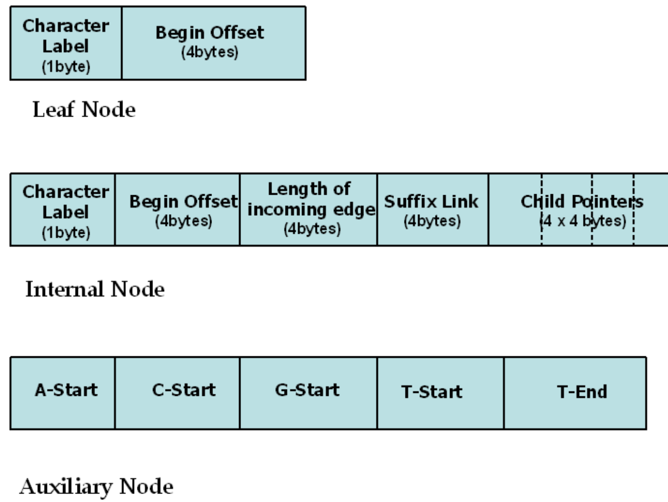


Figure 3.1: Node structures used in GULP

node contains $|\Sigma + 1|$ fields. Every internal node will be associated with an auxiliary node. Leaf nodes are not associated with any auxiliary nodes. The auxiliary nodes is used to identify the partitions(which will be discussed later) of all the children of a given internal node. Note that auxiliary nodes are not an integral part of the suffix tree. The auxiliary nodes are used only while constructing the tree and hence are discarded after building the tree completely. Hence we use a different buffer pool for auxiliary nodes.

The GULP consists of two phases. In the first phase we partition all the suffixes of input string into $|\Sigma|$ partitions, where $|\Sigma|$ is the alphabet size of the string. A partition essentially is the set of positions of the given sequence string. We use the stable, linear time counting sort[1] algorithm to create these partitions. The sequence string is scanned twice, once for counting the number of elements belong to each partition and then actually to create the partitions. At the end of the sorting, each partition contains the set of positions where the corresponding character appears. To further illustrate the partition step, consider the following example. Partitioning the string ATTAGTACA$ would create four partitions {0,3,6,8}, {7}, {4},{1,2,5} for each of character A,C,G and T respectively. The partition for T is {1,2,5} and represents the suffixes {TTAGTACA$, TAGTACA$, TACA$}. Each of these partitions represents the 4 subtrees of

[1]counting sort is linear for a constant alphabet size.

```
Algorithm: GULP (r, B)
r : Root of the subtree
B : Capacity of the disk-page in term in terms of no.of nodes

queue ⟵ r {push root into the BFS queue}
while queue ≠ φ
    r′ ⟵ queue
    getAllChildren(r′,queue)
    if the number of nodes that got created exceeds one
    page then
        while queue ≠ φ do
            m ⟵ queue
            GULP (m, B)
         end while
    end if
end while

procedure: getAllChildren(n,q)
    for each alphabet a check whether there exists a child
    starting with alphabet a
        if such a child is not already created then
        create the child
        if child is an internal node
            sort the partition mapped by this node and
            update the auxiliary node
            queue ⟵ child
        sl = getSuffixLink(n,child,q);
        if the suffix link is created in this iteration then
            queue ⟵ sl
end procedure

procedure: getSuffixLink(p,n,q)
    l = suffix link of p
    e = edgestring of n
    start with l and traverse down with each character
    in the e until the desired node found.
    if the node doesn't exist create one.
    update the suffix link of n
end procedure
```

Figure 3.2: GULP algorithm

the suffix tree starting at root.

Clearly the partitioning phase assumes that the sequence and the length of the sequence is well known in advance. In contrast to this Ukkonen is an incremental algorithm and hence the complete sequence need not be known well in advance. However the biological data, which people deal with are all known in advance, and also to the best of our knowledge there are no streaming sequences in biological domain and hence it is not a major pitfall to assume that we know the entire sequence before processing it.

3.1 Node expansion

The heart of the GULP algorithm is the node expansion, where a given internal node is expanded to create the desired children. Consider the case where we want to expand the node n to create an i-th child($1 <=$ i $<= |\Sigma|$). If the i-th partition of the node n is empty then it means that it doesn't have any i-th child. If the i-th partition of the node contains only one element then it is a leaf. If it contains more than one element we need to do more work to determine whether it is an internal node or leaf node. Consider the example string above where the root node has four partitions. Let us say we want to create a T-th child for the root node. The T-th partition for this node $\{$1,2,5$\}$. We add one to each offset to produce new partition $\{$2,3,6$\}$. We now scan the input string to determine which characters that each of these offsets points to. The characters at these positions are T,A and A. Since we have found two different characters the tree is bifurcated here with two different paths one starting with A and the other starting with T. Hence the T-th child of the root node is an internal node. Also just after one increment we found the internal node and hence the length of edge label of this child is one. If we don't find different characters we repeat this process of increment and scan until we found different characters or we end up with partition of size 1 (we eliminate offsets from end if they exceed the size of the sequence).

Now that we have determined a new internal node (T-th child of root node) we need to update the auxiliary node structure corresponding to this node. we will now sort the partition $\{$2,3,6$\}$ using the stable counting sorting algorithm. This results in two partitions $\{$2$\}$ and $\{$3,6$\}$.

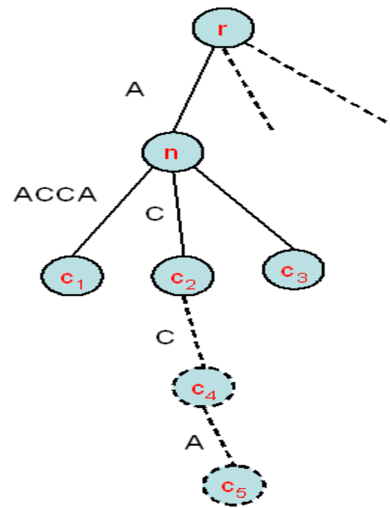


Figure 3.3: Problem in Basic Algorithm

We update the auxiliary nodes to reflect these partitions. Now recursively treating each of these partitions as the whole sequence, we can build the entire suffix tree. The pseudo-code for the GULP is shown in Figure 3.2.

3.2 Suffix link computation

We compute the suffix links for each of the internal nodes. After the node expansion we compute the suffix link for the created node by the following procedure. Let n be the node for which suffix link has to be computed and p be its parent. Let s_p be the suffix link of the parent p. We traverse down through the path starting from node s_p until we found the desired node or we create a node if it doesn't exist. The algorithm for doing it is shown in Figure 3.2. With such an arrangement we can produce the Stellar layout while constructing the suffix tree.

The main problem with this approach is at a particular point in time, to determine the suffix link of a node n we are assuming that the suffix link for the parent of node n has already been determined. Consider the scenario in Figure 3.3. The algorithm starts by expanding all children of node n and creates children c_1, c_2 and c_3. Now in order to determine the suffix link of the node c_1 we start with suffix link of node n which is the root node r. We traverse down from

root until path ACCA is finished and thus in the process creating nodes c_2, c_4 and c_5. When the algorithm reaches c_4 at a later point in time and we want to determine the suffix link of c_5, the suffix link for node c_4 is not present, thus contradicting the assumption. The possible solutions to overcome this is 1) by expanding the node structure to include the parent link 2)Sacrifice the exact Stellar layout and allow variations in the layout to avoid nodes with missing links. The first approach increases the node size significantly and hence defeats the purpose of producing a search-efficient layout. Hence we follow the second approach. We use the following expansion rule to avoid nodes without suffix links.

Node expansion rule: When a node n is created by expanding its parent, the suffix link of the node n is immediately computed before creating any of its siblings or its children.

The new algorithm using the expansion rule is shown in Figure 3.4. The following Lemma shows that using expansion rule we never reach a node with missing suffix link. The proof uses the traversal pattern shown in Figure 3.5.

Lemma 1: The procedure *getSuffixLink (parent, child)* never reaches a node whose parent doesn't have a suffix link and it does terminate.

Proof: Let α be the path length of the child node c and let l be the edge length of parent to child. i.e., $n \rightarrow c$. It is easy to see that, the cumulative edge length of $s_1..s_c$ is exactly l and $L(s_c)$ is exactly $\alpha - 1$. Each of the intermediate nodes s_1 to s_c are either already been created or created on the fly. Let s_i be the newly created node. Using expansion rule, we compute the suffix link for this newly created node s_i. Clearly the path length of the newly created node is strictly less than path length of c. If the suffix link of this node s_i falls on the path from root to c it occurs before c in the path (since $L(c) > L(sl(s_i))$). Similarly the suffix links of intermediate nodes (s_3,s_4,.. etc..) if falls on the path from root to s_i it will necessarily occurs before s_i in the path from root to s_i. Recursively, in the computation of suffix links for each of these intermediate nodes either there suffix links are already occurred or created a fresh and thus follows the same rule. Also it is clear that $L(c) > L(s_i) > L(s_4)$.... etc.. This procedure will terminate when the suffix links of all intermediate nodes are already occurred in the tree or when the root node is reached.

Due to the expansion rule, we persistently store the intermediate nodes that got created in

```
Algorithm: GULP (r, B)
r : Root of the subtree
B : Capacity of the disk-page in term in terms of no.of nodes

queue ⟵ r {push root into the BFS queue}
while queue ≠ φ
    r′ ⟵ queue
    getAllChildren(r′,queue)
    if the number of nodes that got created exceeds one
    page then
        while queue ≠ φ do
            m ⟵ queue
            GULP (m, B)
        end while
    end if
end while

procedure: getAllChildren(n,q)
    for each alphabet a check whether there exists a child
    starting with alphabet a
        if such a child is not already created then
        create the child
        if child is an internal node
            sort the part of array mapped by this node and
            update the auxiliary node
            queue ⟵ child
        sl = getSuffixLink(n,child,q);
        if the suffix link is created in this iteration then
            queue ⟵ sl
end procedure

procedure:getSuffixLink(p,n,q)
    l = suffix link of p
    e = edgestring of n
    start with l and traverse down with
    each character in the e
    if a node doesn't exist with that character create one
    lets call it i and let its parent be i_p
        getSuffixLink(i,i_p,q);
        queue ⟵ i
    update the suffix link of n
end procedure
```

Figure 3.4: GULP algorithm with Node expansion rule

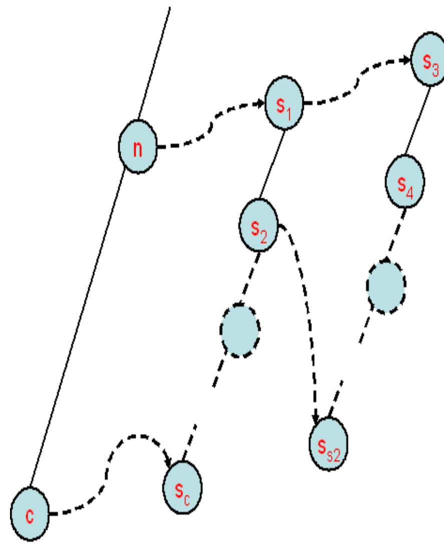


Figure 3.5: The traversal of *getSuffixLink()*

the procedure *getSuffixLink()*. Due to these intermediate nodes we get an approximate Stellar layout rather than exact layout. The number of intermediate nodes that will be created depends upon the edge length of the initial child. As the edge length increase we have more chances of getting more intermediate nodes. The edge lengths are large at deeper levels of the tree and are small for nodes near to the root.

3.3 Time Complexity

The time complexity of GULP is determined by the two procedures *getAllChildren()* and *getSuffixLink()*. The procedure *getAllChildren()* sorts the partition mapped by the corresponding internal node. We use the linear time complexity stable counting sort algorithm for sorting the partitions. Each of the partitions in the worst case can represent the entire suffixes of the string. Thus the maximum size taken by each of these partitions is order of length of the sequence, N. So the time complexity of the sorting procedure is $O(N)$.

The time complexity of the procedure *getSuffixLink()* is determined by the number of intermediate nodes that got created in the process. While traversing down through the parent's suffix link if a node is created we incur additional time, but this time spent in creating additional nodes

is amortized by the fact that we doesn't need to create these nodes again. Once the sorting of partition is done for a particular node the sorting doesn't happen for the same node again. If no nodes are created in this process, the time incurred is proportional to the length of the incoming edge of the node for which we are computing the suffix link. Thus the time complexity of this procedure is $O(l)$, where l is length of the incoming edge. In Section 4.1 we see that the maximum length of an edge for a dataset of 25Mbp is only 8152. Given that the edge length is small ($l << N$) the time complexity of the getSuffixLink is $O(1)$.

The main procedure GULP calls *getAllChildren()* until the complete suffix tree is built and this procedure incurs a time of $O(N)$ for each node it creates. Thus the overall time complexity of the GULP is $O(N^2)$, where N is the length of the data sequence.

3.4 Experimental Framework and Results

In this section, we present the experimental evaluation of different disk based suffix tree methods. We first compare the construction time with Ukkonen algorithm using TOP-Q buffering strategy for different data sets. Since Stellar is already proven to be outperforming other layouts [6], we evaluate the search performance of the new layout with Stellar layout.

We compare the performance of maximal substring search procedures described in section 1.2, for suffix trees built over the HC2/25 sequence(25Mbp of Human Chromosome 2), with Stellar layout.

The suffix tree implementation used here is based on an efficient array based tree node representation suggested in [4] with 22.5 bytes per symbol. The disk page size is set to 4KB. For maximal substring search on HC2/25, a buffer pool of 25 MB, which forms approximately 5% of the total suffix tree, was used.

3.4.1 Query Workload

The cost of the search process is considerably affected by the following query workload characteristics:

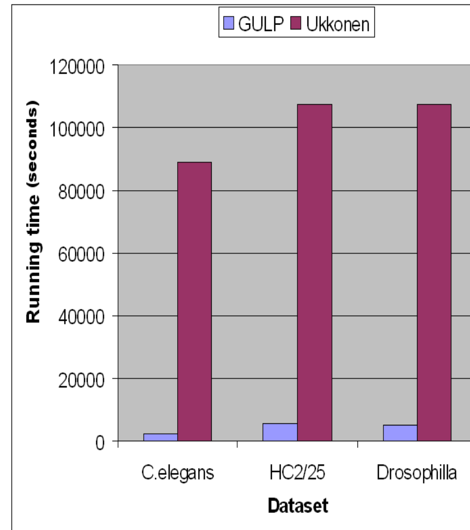


Figure 3.6: Construction times

Query length: The length of the query directly determines the total number of iterations required for locating all the maximal substrings. Further, the increased query length may result in a larger number of matches, increasing the cost of reporting results.

Value of λ: The user-specified threshold λ serves as the lower-bound on the length of the match before all instances of the match are reported. The typical operational range of this parameter in a variety of DNA sequence retrieval software is between 9 and 50. We have set it to 11, the default value used by BLAST.

We generated our query workload based on a collection of sequences from Expressed Sequence Tag (EST) database of GenBank [38]. The EST-database contains 856008 sequences with average sequence length of 357.6 basepairs. Using this base collection we generated three length restricted query collections, with lengths 50, 100 and 200, by randomly sampling fixed-length subsequences from each entry of the EST-database. In order to remove any remaining bias in the ordering of EST fragments, we sampled 10,000 sequences from each length-restricted query set to form three query collection **hEST50**, **hEST100** and **hEST200** used in our evaluations.

3.4.2 Construction times

In this section we compare the construction time of our new algorithm with Ukkonen algorithm using TOP-Q [4] buffering strategy. Figure 3.6 shows the comparison between the construction times of Ukkonen and GULP . We used a buffer size of 10% of the estimated tree size with disk page size being set to 4KB. GULP uses a set of different buffer pools, one for each of its datastructure, but we made it sure that the total amount of memory used doesn't exceed the alloted space.

GULP uses LRU as its buffering policy over TOP-Q. The reasons for this choice are as follows. TOP-Q buffering policy is designed with a view that during any point in the construction process the edges will be splitted and new nodes are created in between. This is not the case with GULP . Also TOP-Q gives priorities to the pages whose average depth is lower. This is because there are edges which will be get splitted to form new nodes and the chance of this split occurring is high at pages with lower average depth. Unlike this, in our algorithm, once the pages at the top are updated completely they are not accessed again. The access pattern of GULP is completely different than Ukkonen. Also by using LRU, we got the buffer hit ratio of around 97-99%. Hence we used LRU buffering policy for our algorithm.

We measured the wall clock time of both the algorithms on 3 different data sets and plotted them. The experiments are carried out on Intel Pentium 4, 2GHz machine with 256KB cache and 1GB of RAM, running Redhat Linux 8.0. The programs are written in C++ and compiled using gcc 2.93 with level-3 optimizations. Our new algorithm performs orders of magnitude better than the Ukkonen. In the worst case GULP is 20 times faster than the Ukkonen and in the best case it is 40 times better than the Ukkonen.

3.4.3 Structural Localities of the Layouts

Table 3.1 shows the structural localities of suffix trees generated using different algorithms on 3 different data sets. The data set HC2/25 represents the 25Mbp long sequence drawn from Human chromosome 2. All the localities were obtained with a disk page size of 4KB. The layouts evaluated here are 1)CO(creation order) refers to the layout of the tree obtained when the tree is created using Ukkonen algorithm [34]. 2)SBFS layout as discussed earlier [16].

Data set	Storage layout	Tree edges	Suffix-links
HC2/25	CO	0.13%	41.7%
	SBFS	73.3%	0.1%
	Stellar	62.5%	39.9%
	GULP	55.9%	50.9%
C.elegans	CO	0.3%	39.78%
	SBFS	72.2%	0.01%
	Stellar	61.6%	39.6%
	GULP	55.18%	49.8%
Drosophilla	CO	0.06%	33.07%
	SBFS	68.49%	0.00%
	Stellar	59.23%	38.8%
	GULP	54.55%	44.94%

Table 3.1: Structural Edge and Link localities

3)The Stellar layout [6] 4)The layout produced by our new algorithm GULP .

From the results, we first see that the CO-layout provides practically no tree-edge locality-only 0.3% of the tree-edges are intra-page. while suffix link locality is comparatively high-42%. The SBFS layout, on the other hand represents the opposite extreme in structural locality, with 70-75% of tree-edges being intra page. But less than 0.1% of suffix links being local. Stellar has suffix link locality close to CO and tree-edge link locality comparable to SBFS. Our algorithm produces edge locality comparable to that of Stellar but high link locality than any other. However maximal substring search use more tree-edges compared to suffix links[5] and hence search procedures may be slightly expensive.

3.4.4 Search Performance

Figure 3.8 shows the comparison in search performance of GULP layout with Stellar. As indicated above, a buffer pool of 25 MB, which forms approximately 5% of the total suffix tree, was used. The x-axis represents the value of λ used in the search process. The y-axis shows the amount of disk I/O incurred in the search process. It is clearly evident that Stellar is incurring less disk I/O than GULP. However the amount of disk I/O incurred by GULP is always within 10-15% of disk I/O incurred by Stellar.

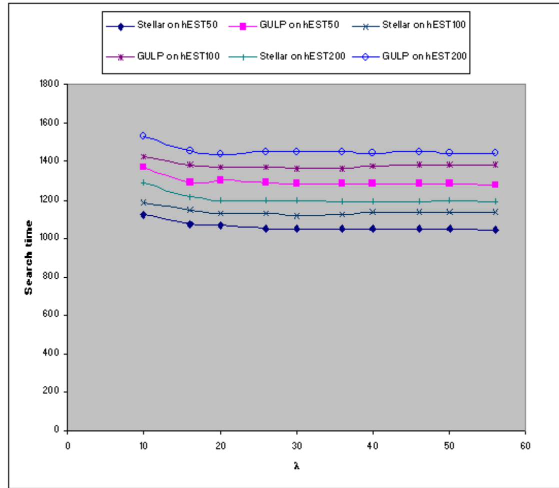


Figure 3.7: Search times of Stellar layout vs GULP

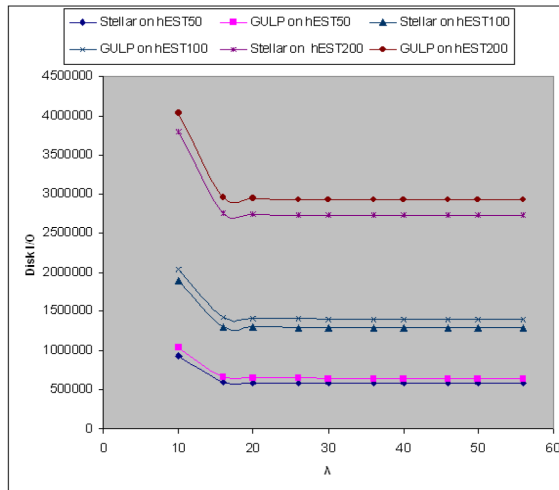


Figure 3.8: Disk I/O of Stellar layout vs GULP

Figure 3.7 provides insight into the actual search time that was involved in the search process. The x-axis represents the value of lambda used and the y-axis represents the wall clock time spent in search process. As Stellar is incurring less disk I/O its response time is also less, but in this case also the time spent by GULP is always within 10-15% of the time spent by Stellar.

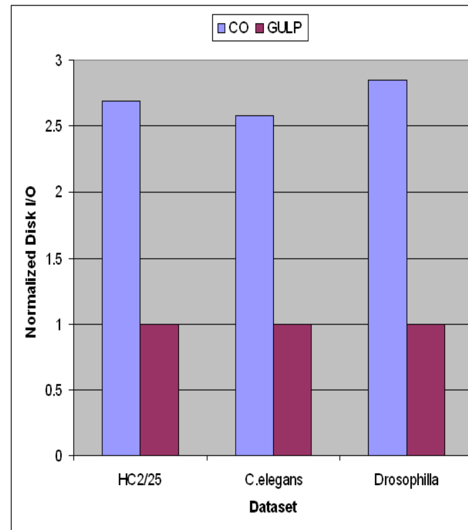


Figure 3.9: Performance of conversion to Stellar layout

3.4.5 Conversion to Stellar

Since construction is a one time process, it is really the search performance that matters. Since GULP incurs 10-15% more disk I/O than Stellar, it may not be a good layout in terms of search process. Since we already have an approximate Stellar layout in hand its a good idea to measure the performance of conversion from CO to Stellar to with that of GULP to Stellar. Figure 3.4.5 shows the comparison of amount of disk I/O incurred while converting to Stellar layout using the same buffersize as construction. On the average GULP takes 2.5 times less disk I/O than Stellar. Finally, the process of constructing GULP and conversion to Stellar is 5-10 times faster than that of CO and Stellar.

3.4.6 Comparison With TDD

Tata et. al. [32] proposed TDD technique for efficiently constructing the USTs. As of now, TDD is the fastest known algorithm for constructing the UST. TDD produces suffix trees with no suffix links. Since there are no suffix links the suffix tree produced by TDD occupies less space compared to suffix trees with links. The TDD produces suffix tree in a depth first manner. We compared the construction time of GULP with TDD. The TDD construction times were

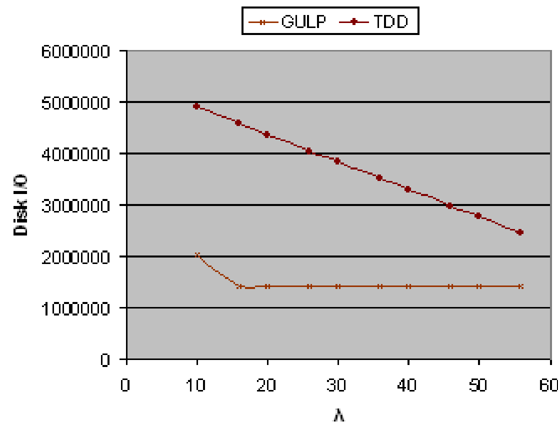


Figure 3.10: Search Performance of GULP and TDD on hEST100

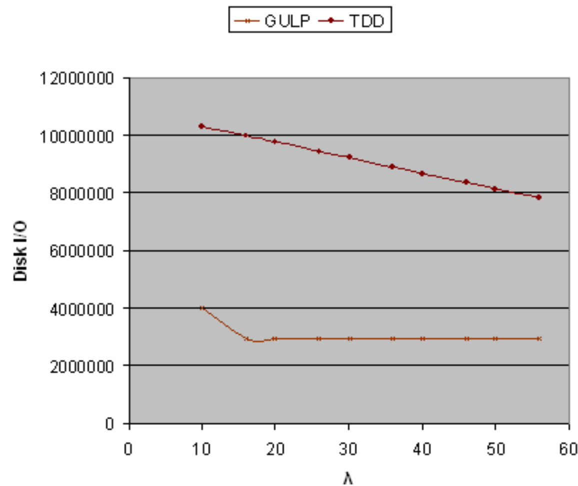


Figure 3.11: Search Performance of GULP and TDD on hEST200

much faster than GULP. TDD produced suffix trees 3-5 times faster compared to GULP, but construction is a one time process and after constructing the tree once, we use several search procedures on the already constructed tree. Since the tree produced by TDD doesn't contain suffix links, we used the MSS_{UST} algorithm described in Section 1.2 for maximal substring search. TDD algorithm uses 4 different datastructures during construction and 2 datastructures for search process. The two datastructures used for the search process are 1)String, for accessing the original data string and 2)Tree for accessing the suffix tree itself. The memory for String

structure was allotted to host the entire string in memory. The memory alloted for tree structure is same as that of GULP. Hence we measured the amount of disk I/O incurred in accessing the suffix tree only. Figures 3.10 and 3.11 shows the comparison of GULP with TDD. The x-axis shows the threshold(λ) value used in maximal substring search and the y-axis shows the amount of disk I/O occurred during the search process. For hEST100 on the average GULP incurs only 41% of the disk I/O and for hEST200 GULP incurs only 33% of the disk I/O of that of GULP. Clearly as the length of the query increases the TDD has to process many subqueries(Due to the absence of suffix links) and hence incurring additional disk I/O. For alignment of whole genomes the query length is huge and hence using of USTs may be impractical.

Chapter 4

Compact Suffix Tree

The second part of our work deals with compressing the nodes of the suffix tree for reducing the disk space required to store the tree. It turned out that the compression scheme not only reduced the amount of disk space needed but also improved the performance of search procedures. The suffix tree algorithms typically uses a fixed node structure. The node structure used by Stellar and as well as GULP consist of an implementation where the amount of memory occupied by each of the internal nodes or leaf nodes is fixed. But both the Stellar and GULP exhibits good edge and link localities which can be exploited to reduce the amount disk space required by the suffix tree. We designed our compression scheme based on the following set of observations.

4.1 Motivation

Observation 1: The edges in the tree can be broadly classified into two categories; local edges, those which points to the nodes in the same block and external edges , those which points to the nodes that are present outside the current block. The Stellar and GULP use two separate disk files: one for internal nodes and one for leaf nodes. For a sequence of length N this results in exactly N edges to be interpage (external). Also due to the fixed node structure we use the same amount of memory for representing the local edges and as well as the external edges. For the local edges it is enough to store the offset with in the block to identify the node. With high edge and link localities we have more local edges than external edges and thus a scope for

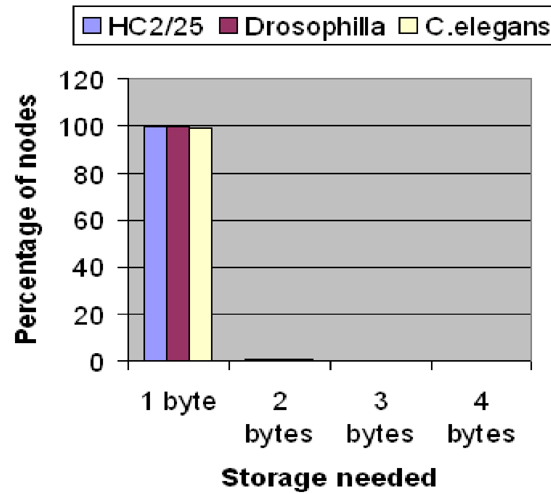


Figure 4.1: Distribution of length field

compression.

Observation2: The array implementation of a node structure for a sequence of alphabet size $|\Sigma|$ consists of $|\Sigma|$ pointers for each internal node each of which points to a child starting with that corresponding alphabet.

To fully utilize each of these pointers we need a scenario where each level is fully utilized upto leaf levels. Assuming all internal nodes directly above the leaves are fully utilized, thus it has $\frac{N}{|\Sigma|}$ nodes. Proceeding similarly we need $\frac{N}{|\Sigma|^2}$ at next level and so on. Adding them up, it is easy to see for a suffix tree built on a sequence of length N, in order to fully utilize each of these pointers the number of internal nodes required is $\frac{N}{|\Sigma|-1}$. In the case of DNA alphabet, for a sequence of length N this number is 0.33N. Typically the number of internal nodes in a suffix tree built on a DNA sequence ranges between 0.6N to 0.7N. That means when using an array implementation, approximately half of the child pointers doesn't carry useful information and are set to null.

Observation3: The node structure uses 4 bytes to represent the length of an edge. This is because the maximum length of an edge is atmost the length of the sequence. The plot in Figure 4.1 shows the distribution of value of length field among all internal nodes for three different datasets. We found that the maximum length of any edge among these datasets is 8152. Also on

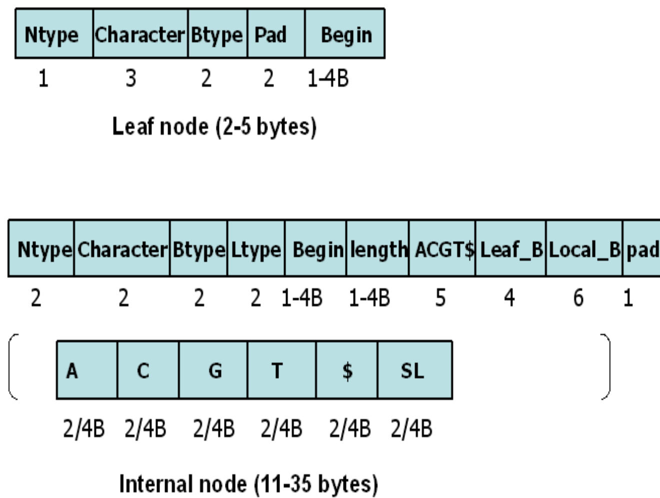


Figure 4.2: Compressed Node structures

average, 99.5% of the nodes have their length fields less than 256 and hence can be represented by using just 1 byte. In all the cases, there are no edges whose length field occupies more than 2 bytes.

4.2 Compressed Node Structure

Based on the above observations we designed a flexible node structure which is shown in Figure 4.2. The number below the field specifies the number of bits(or bytes) that field occupies. The first field Ntype identifies the type of node which is set to 0 for leaf nodes and 1 for internal nodes. The DNA has alphabet of size 4 and with the terminating symbol '$ ' it is 5, hence we use 3 bits for encoding the character. The field Btype identifies the number of bytes occupied by the begin field of that node. The Btype occupies 2 bits thus represents 4 different values one for each of the 1 to 4 bytes. The leaf node has a variable length begin field which occupies 1 to 4 bytes depending on the value of btype. Thus the leaf node occupies 2 to 5 bytes. The leaf node has a pad field of 2 bits which is set to 0. This pad field can serve several different purposes. If we have suffix tree built on larger alphabet sizes we can extend the character field to represent all the alphabets without extending the size of leaf node. Secondly, while doing different kinds

of traversals on the tree we can use pad field to mark the visited nodes.

The internal node has all the fields as that of leaf node and some extra fields. The field Ltype denotes the number of bytes occupied by the length filed of the internal nodes. Each of the 4 different values represents the 1 to 4 bytes occupied by the length field. The length field is thus variable and hence occupies 1 to 4 bytes. The next 2 bytes consists of bitmap vector which is composed of three fields ACGT,Leaf_B and Local_B. The field ACGT is a bitmap which denotes which among the children, whose edge label starts with A,C,G,T and $ are present in the suffix tree. Suppose an internal node has a children whose edge label starts with 'A' then the corresponding bit(The first bit in this case) of ACGT is set to 1 otherwise it is cleared to 0. The Leaf_B bitmap is used to differentiate between leaf nodes and internal nodes among the all its children. Suppose an internal node has a children whose edge label starts with 'A' and is a leaf then the corresponding bit in Leaf_B is set to 1 cleared otherwise. If an internal node has a child whose edge label starts with $ then we know its guaranteed to be a leaf and hence we don't use any bits to represent the $ child. The field Local_B is the bitarray to mark the edges and the suffix link which are local. Then the rest of the fields of internal nodes are dependent on these bitmask. We use A,C,G,T and $ fields for each of the children and sl for suffix link of the the node. Each of them occupies 2 to 4 bytes depending on whether local or external. We doesn't use any space if a children is not present.

Since we use 2 bytes for local edges which is the offset in the current block we can have a page size of maximum 64KB. The internal node occupies a minium of 11 bytes and a maximum of 35 bytes. When an internal node has all its children which are external, the worst case increase in the node size compared to a fixed array implementation is only 2 bytes.

4.3 Compressed Stellar Layout

With this node structure we use the Stellar algorithm to convert the suffix tree to Stellar layout. We call this new layout as CS-layout (Compressed Stellar layout). Unlike Stellar layout CS-layout use a single disk file for both internal and leaf nodes. We use a two-phase algorithm here for converting a given tree to CS-Layout. In the first phase of the algorithm The nodes

```
PageMapper(n,curpage)
;n is node in the suffix tree.
;curpage is the current block no.

rs = remaining space in current block
if n is not visited then
    page(n) ← curpage
    if sl(n) belongs to current page then
        set n's suffix link as local and reclaim space
    endif
endif
queue ← n
while queue not empty
    n ← queue
    for each child c of n
        if c is not visited then
            if rs > sizeof(c) then
                page(c) ← curpage
                set n's child as local and reclaim space
                queue ← c
            endif
            s ← sl(c)
            if s is not visited
                if rs > sizeof(s) then
                    page(s) ← curpage
                    set c's suffix link as local and
                        reclaim space
                    queue ← c
            endif
        endif
    end for
end while
```

Figure 4.3: PageMapper

are allocated to the physical blocks. In Phase 2 we update all the children pointers and suffix links. Using the Stellar algorithm it is complicated to foresee which edges will be local and which will be external hence we use the PageMapper algorithm presented in Figure 4.3 at each page. At first we don't directly allocate the memory required by the each node. The algorithm first allots the node to the current page assuming that the node occupies its maximum size. In successive iterations of the loop when its children (or suffix link) are alloted to the same page all such edges are set to local and the space savings are reclaimed. When the PageMapper finally terminates we will have a set of nodes which are mapped to current page. At this stage we also know the local edges(or links) of the node. Now we actually allocate the nodes within the page.

```
CStellar(r)
;r is the root node of current subtree

PageMapper(r,cur_block)
Allocate all the nodes mapped to this page in the cur_block
while queue is not empty
do
    n ← queue
    CStellar(q);
end while
```

Figure 4.4: Compressed Stellar layout Algorithm

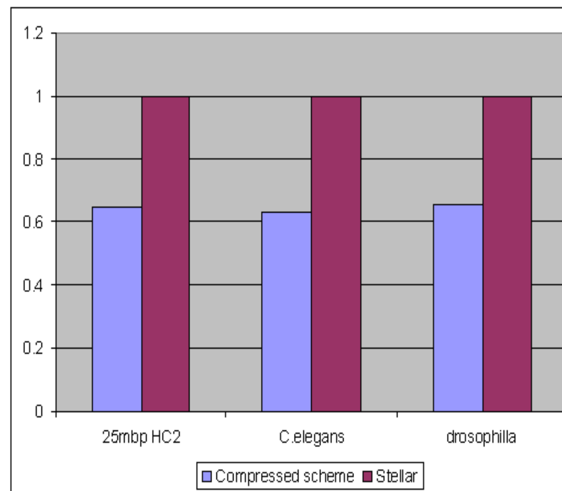


Figure 4.5: Disk space savings using compressed scheme

The algorithm for CS-Layout is shown in the Figure 4.4.

4.4 Experimental Framework and Results

We ran the experiments on three different datasets using this compressed node structures. The Figure 4.5 shows the amount of disk space needed using this compression scheme. The disk space is normalized by setting the amount of space occupied by Stellar to 1. On the average this scheme requires about 65% of the space occupied by the original tree.

The reduction in the amount of disk space may not help to efficiently process the query

Data set	Storage layout	Tree edges	Suffix-links
HC2/25	Stellar	62.5%	39.9%
	CS-Layout	77.3%	40.0%
C.elegans	Stellar	60.7%	38.9%
	CS-Layout	77.1%	39.5%

Table 4.1: Structural Edge and Link localities of CS-Layout

unless it exhibits a good layout in the disk. The Table 4.1 shows the static structural localities. The locality is the ratio of the local edges to that of total edges. On the average, the compression scheme achieves edge locality of around 77% and link locality around 40%.

4.4.1 Search Performance of CS-Layout

To analyze the efficiency of CS-Layout we use the same query set as section 3.4.1. We present results for suffix-trees built over a 25Mbp sequence drawn from Human Chromosome II. All experiments were conducted with disk page size set to 4K bytes, a typical pagesize in today's systems. Since Stellar layout is already shown to be outperforming other layouts, in this section we provide the comparison only with the Stellar layout. We ignore the physical organization of the files on disks and every page read from file is assumed to take a constant time to perform. If a block to access is not in the buffer, a page fault occurs and a new page of 4k bytes containing the required block is fetched into the buffer.

The experiments are carried out on Intel Pentium 4, 2GHz machine with 256KB cache and 1GB of RAM, running Redhat Linux 8.0. The programs are written in C++ and compiled using gcc 2.93 with level3 optimizations. We use a buffer-pool of 25MB approximately equal to 5% of the tree size. For the Stellar layout we use a split of 2:1 across internal nodes and leaf nodes as suggested in [5].

The graph in Figure 4.7 shows the amount of disk I/O spent with varying lambda values across 3 different query sets namely hEST50, hEST100 and hEST200. From the figure it is evident that,the CS-Layout demands less DISK I/O than Stellar layout consistently. The best case being CS-Layout requires only 62% of DISK I/O used by Stellar and the worst case is

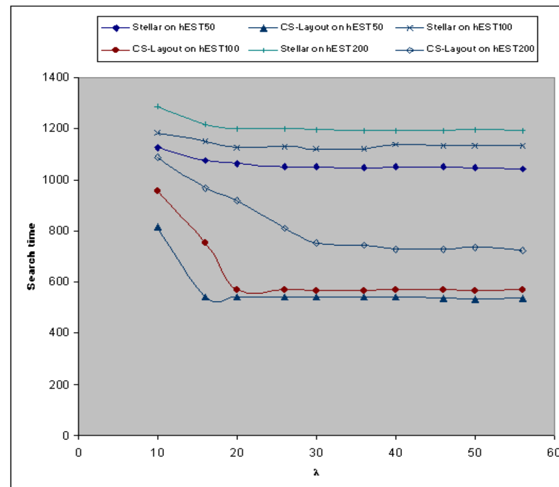


Figure 4.6: Search times of Stellar layout vs CS-Layout

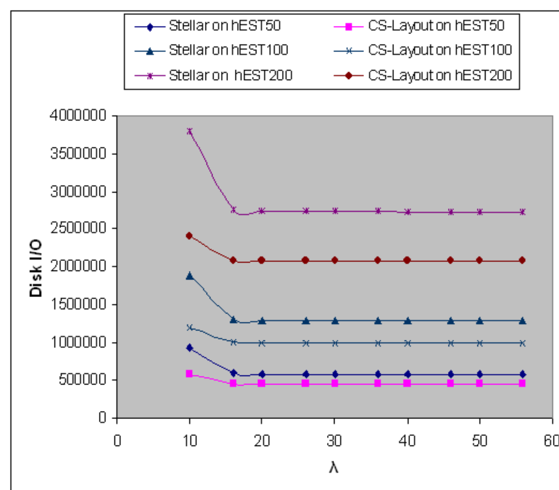


Figure 4.7: Disk I/O of Stellar layout vs CS-Layout

when CS-Layout requires as high as 79% of the DISK I/O as that of Stellar. On the average CS-Layout requires about 70-75% disk I/O as that of Stellar. The amount of DISK I/O required is more when the lambda value is higher. At lambda=11 which is default value used by BLAST the CS-Layout require on the average of 65% DISK I/O to that of Stellar.

Since response time is final way to assess the value of goodness of the layout, we have also measured the time taken by each of them. Figure 4.4.1 shows the search performance of CS-Layout vs Stellar layout. The x-axis is the lambda value used and y-axis represents the wall clock time incurred in search process. CS-Layout clearly beats the Stellar layout in terms of the search time spent also.

Chapter 5

Conclusions

In this thesis we presented an efficient and search-conscious algorithm called GULP for the persistent construction of suffix tree with links. The algorithm we presented not only runs faster than contemporary techniques but also produces the suffix tree which exhibit good disk layout. To the best of our knowledge, GULP is the first construction algorithm designed with a view of producing good disk layout. We also presented a compression scheme called CS-Layout which exploit the layouts and other factors to condense the suffix tree without losing any information. The CS-Layout not only uses less disk space but also hastens the search process. The maximal substring search process uses tree edges more than that of links. As a part of future work we are trying to look forward for techniques or heuristics which increases the edge locality of resulting suffix tree. We also try to investigate the possibility of directly producing the compressed layout using GULP.

Bibliography

[1] M. I. Abouelhoda, S. Kurtz and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53-86,2004.

[2] S. Alstrup, M. Bender, E. Demaine, M. Farach-Colton, J. Munro, T. Rauhe and M. Thorup. Efficient tree layout in a multilevel memory hierarchy. Technical Report arXiv:cs.DS/0211010v1, 2002.

[3] S. F. Altschul, W. Gish, W. Miller, W. Myers, D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology* 215:403-410,1990.

[4] S. Bedathur and J. Haritsa. Engineering a Fast Online Suffix Tree Construction In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2004.

[5] S.Bedathur BODHI: A Database Engine for Biological Applications, Phd.Thesis, Indian Institute of Science, 2005.

[6] S. Bedathur and J. Haritsa. Search-Optimized Persistent Suffix-tree Storage for Biological Applications. In P*roceedings of 12th IEEE International Conference on High Performance Computing (HiPC)*, December 2005.

[7] A. L. Cobbs. Fast approximate matching using suffix trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching(CPM)*,1995.

[8] A. Carvalho, A. Freitas, A. Oliveira and M. Sagot.Efficient extraction of structured motifs using box-links. In *Proceedings of the 11th Conference on String Processing and Information Retrieval*, volume 11,2004.

[9] W. I. Chang and E. L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1990.

[10] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica*, 12:327-344, 1994.

[11] David R.Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383-391, ACM Press 1996.

[12] C. F. Cheung, J. X. Yu and H. Lu. Constructing suffix tree for gigabyte sequences with megabyte memory. *IEEE Transactions on Knowledge and Data Engineering*,17(1):90-105, 2005.

[13] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White and S. L. Salzberg. Alignment of Whole Genomes. *Nucleic Acids Research.* 27(11):2369–2376,1999.

[14] E. D. Demaine, J. Iacono and S. Langerman. Worst-Case Optimal Tree Layout in a Memory Hierarchy. CoRR cs.DS/0410048: (2004).

[15] A.L. Delcher, A. Phillippy, J. Carlton and S.L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research*, 30(11):2478-2483, 2002.

[16] A. A. Diwan, S. Rane, S. Seshadri and S. Sudarshan. Clustering Techniques for Minimizing External Path Length. *In Proceedings of the 22nd International Conference on Very Large Databases (VLDB)*, 1996.

[17] P. Ferragina and R. Grossi. The string B-tree: A new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236-280, 1999.

[18] J. Gil and A. Itai. How to pack trees. *Journal of Algorithms*, 32(2):108-132, 1999.

[19] R. Giegerich, S. Kurtz and J. Stoye. Efficient Implementation of Lazy Suffix Trees. In *Proceedings of the Third Workshop on Algorithmic Engineering (WAE 99)*, 1999.

[20] J. Gil and A. Itai. How to Pack Trees. *Journal of Algorithms*,32(2), 1999.

[21] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, Cambridge, 1997.

[22] D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69:525-546, 2004.

[23] E. Hunt, M.P. Atkinson and R. W. Irving. Database indexing for large DNA and protein sequence collections. *The VLDB journal*, 11:256-271,2002.

[24] E. Hunt, M.P. Atkinson and R. W. Irving. A Database indexing to large biological sequences. In *Proceedings of the 27th International conference on Very Large Databases (VLDB)*, 2001.

[25] H. Hyyr, G. Navarro: A Practical Index for Genome Searching. *SPIRE* 2003: 341-349.

[26] S.Kurtz. Reducing the space requirement of suffix trees. *Software-Practice and Experience*, 13:1149-1171, 1999.

[27] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262-272, 1976.

[28] U.Mamber and G.Myers. Suffix arrays: A new method for online string searches. *SIAM journal of computing*, 22(5):935-948, 1993.

[29] C.Meek, J.M.Patel and S.Kasetty. OASIS: An online and accurate technique for local alignment searches on biological sequences. In *Proceedings of the 29th International conference on Very Large Databases(VLDB)*. 910-921, 2003.

[30] G.Navarro. A guided tour to approximate string matching. *ACM computing surveys*, 33(1):31-88, March 2001.

[31] T. Smith and M. waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*. 147:195-197, 1981.

[32] S. Tata, R. A. Hankins and J. M. Patel. Practical Suffix Tree Construction. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB)*,2004.

[33] Y. Tian, S. Tata, R. A. Hankins, J. M. Patel, Practical Methods for Constructing Suffix Trees, *The VLDB Journal* 14(3) Sep 2005,pp 281-299.

[34] E. Ukkonen. Online Construction of Suffix-trees. *Algorithmica*, 14(3), 1995.

[35] E. Ukkonen. Approximate String Matching over Suffix Trees. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 1993.

[36] H. E. Williams and J. Zobel. Indexing and Retrieval for Genomic Databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(1), 2002.

[37] P. Weiner. Linear pattern matching algorithms. In *14th IEEE Symposium on Switching and Automata Theory*, 1973.

[38] *GenBank. http://www.ncbi.nlm.nih.gov/Genbank.*

[39] *SwissProt.: http://www.expasy.org/sprot.*