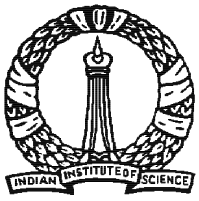
Search-Optimized Disk Layouts for Suffix-Tree Genomic Indexes

A Thesis
Submitted for the Degree of
Master of Science (Engineering)
in the Faculty of Engineering

By
Rajul D Bhavsar



Supercomputer Education and Research Centre
INDIAN INSTITUTE OF SCIENCE
BANGALORE – 560 012, INDIA

August, 2011

Acknowledgements

I would like to thank my advisor Prof. Jayant Haritsa, for allowing me an opportunity to work under him. I am also grateful to him for all his comments and suggestions, which are critical and useful for the completion of my thesis.

I would also like to thank all the DSLites for their help during various stages of my stay at DSL. I am thankful to my seniors Abhijit and Gourav for their help in initial days in the lab and also to Shivashankar for helping me in the project related issues. I am also grateful to my colleagues Atreyee, Abhirama, Ravi, Sourjya, and Harsh for their valuable suggestions. Many thanks to Anshuman, Mayuresh, Mahesh, Rakshit and Sreepathi, for their suggestions and solutions in various situations.

I am also thankful to Govindarajan Sir for his readiness to help me in every situation. Many thanks to whole SERC staff, particularly to Ms. Mallika, Mr. Madan, and Mr. Shekhar, for their help in various administrative issues. I specially thank Mr. Kiran, for his quick help during the allocation of resources on Regatta machine. I would also like to thank all the people from the Institute who directly or indirectly made this work possible.

Finally, it would not be possible without the support from my family members and friends, whose constant encouragement gave me the strength to reach upto this stage.

Abstract

Over the last decade, biological sequence repositories have been growing at an exponential rate. Sophisticated indexing techniques are required to facilitate efficient searching through these humongous genetic repositories. A particularly attractive index structure for such sequence processing is the classical *suffix-tree*, a vertically compressed trie structure built over the set of all suffixes of a sequence. Its attractiveness stems from its linearity properties – suffix-tree construction times are linear in the size of the indexed sequences, while search times are linear in the size of the query strings.

In practice, however, the promise of suffix-trees is not realized for extremely long sequences, such as the human genome, that run into the billions of characters. This is because suffix-trees, which are typically an order of magnitude *larger* than the indexed sequence, necessarily have to be *disk-resident* for such elongated sequences, and their traditional construction and traversal algorithms result in *random* disk accesses.

We investigate, in this thesis, *post-construction* techniques for disk-based suffix-tree storage optimization, with the objective of maximizing disk-reference locality during query processing. Specifically, we consider approaches based on (a) reorganizing the layouts, and (b) improving the physical structures of internal nodes of disk-resident suffix-trees. In marked contrast to prior techniques in the literature that modify the suffix-tree itself in order to gain performance (for example, by dropping the suffix-link edges), an important aspect of our work is that the logical structure is retained in pristine form, thereby retaining all the standard functionalities associated with these trees.

We begin by focusing on the layout reorganization, in which (i) complete reworking of node-to-block assignments, and (ii) resequencing of storage blocks, are carried out. While the classical suffix-tree construction algorithms deliver a depth-first layout, our node-to-

block assignments are based on combining the *breadth-first* layout approach advocated in the recent literature with the increased restrictions on the assignments of nodes to blocks, based on an analysis of node traversal patterns. Subsequently, in the second step, the optimized sequence of disk blocks is determined by making the physical distance between a pair of blocks commensurate with the estimated probabilities of visiting both these blocks during the search traversals.

The second part of our thesis is directed towards improving the physical structures of the suffix-tree internal nodes. In particular, we propose an embedding strategy whereby leaf nodes can be completely represented within their parent internal nodes, without requiring any space extension of the parent node's structure.

To quantitatively evaluate the benefits of our reorganized and restructured layouts, we have conducted extensive experiments on complete human genome sequences, with complex and computationally expensive user queries that involve finding the maximal common substring matches of the query strings. The experimental framework is instrumented to provide a variety of supporting statistics, such as intra-block localities, that help explain the observed behavior of the various suffix-tree layouts.

We show, for the first time, that the layout reorganization approach can be scaled to entire genomes, including the human genome. In the layout reorganization, with careful choice of node-to-block assignment condition and optimized sequence of blocks, search-time improvements ranging from 25% to 75% can be achieved with respect to the construction layouts on such genomes. While the layout reorganization does take considerable time, it is a one-time process whereas searches will be repeatedly invoked on this index.

The internalization of leaf nodes results in a 25% reduction in the suffix-tree space occupancy. More importantly, when applied to the construction layout, it provides search-time improvements ranging from 25% to 85%, and in conjunction with the reorganized layout, searches are speeded up by 50% to 90%.

Overall, our study and experimental results indicate that through careful choice of node implementations and layouts, the disk access locality of suffix-trees can be improved to the extent that upto an order-of-magnitude improvements in search-times may result relative to the classical implementations.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

DNA sequence[1] analysis is increasingly gaining importance in various scientific fields. The well-known usage of DNA sequence analysis is in the diagnosis of different diseases in the field of medical science[2]. Apart from this usage, DNA sequence analysis has found many other applications. For example, it is used in the field of bioarcheology to study migration of different population groups based on female genetic inheritance [35]. It is also used in the field of DNA forensics to identify crime and catastrophe victims. There are even efforts to build electronic micro-chips from DNA, due to its "reproducible, repetitive kinds of patterns" [36]. So, going ahead in the future, DNA sequence analysis will play a crucial role in almost every field of science.

On the other side of the DNA sequencing, the technological improvements have increased the accuracy of genomic sequence mapping. But at the same time, there is decrease in the cost and time duration for sequence mapping. Due to this, biological sequence repositories (like GenBank [11] and EMBL [10]) are continuously growing at an exponential rate. As a case in point, it has been estimated in [30] that the current growth rate for biological sequence repositories is around *36 Gbp*[3] *per month*. Compare it with the sequencing rate of the whole *Human Genome Project* [34], which took *13 years* and

[1]Text sequence consisting of characters - A, C, G and T, which represents the order of occurrence of nucleotide bases in DeoxyriboNucleic Acid.

[2]In 20th century, a disease had been identified based on symptoms, but in 21st century, it will be DNA sequence with which it will be diagnosed. [33]

[3]A Gbp is billion base-pairs and one base-pair refers to either of the 4 characters - A, C, G and T.

3 billion USD to sequence the human genome which is around *3 Gbp*. Today, efforts are going on to sequence the human genome in *5000 USD* [32]. This exponential decrease in costing is complementary to the exponentially growing sequence repositories. The newer and better technology has helped in continuing this increase in sequence mapping rate and decrease in cost.

These humongous genetic repositories require sophisticated indexing techniques to facilitate efficient searching on them. Due to lack of any particular structure in these genetic sequences (like words in a sentence), the classical *suffix-tree* is an attractive choice for indexing these sequences, as suffix-trees are full-text indexes [15], and suffix-trees are constructed in times which are linear in the size of indexed sequences. Also, searching on suffix-trees can be performed in times which are linear in the size of query strings. In practice, however, the promise of suffix-trees is not realized for extremely long sequences, such as the human genome, that run into the billions of characters. This is because suffix-trees, which are typically order-of-magnitude larger than their indexed sequences, necessarily have to be disk-resident for such elongated sequences, and their traditional construction algorithms result in random disk accesses.

Recently, there are many algorithms (such as [21, 12]) which can construct the genome-scale suffix-tree very quickly by localizing suffix-tree nodes. Using this criterion of localization, these algorithms have tried to avoid random disk accesses during suffix-tree construction. But once the suffix-tree is constructed, it is going to be used many times for various kind of search applications [13], which involve random disk accesses. These unavoidable random disk accesses create performance bottleneck, as they take orders of magnitude more time than random accesses to the main-memory. So, the more important question is - *How can we optimize storage for genome-scale disk-based suffix-trees such that search-time over them can be reduced?* In this thesis, we will attempt to address this question by proposing various search-optimized layouts for genome-scale disk-based suffix-trees, which reduce search-time by focusing on the optimization of random disk accesses incurred during the usage of suffix-trees for searching.

1.1 Motivation and Research Challenges

1.1.1 Motivation

1. Due to the exponential rate of sequence mapping, the sizes of various biological sequence repositories are currently in the order of Terabases[4] [31]. So, considering this humongous size, the efficient utilization of suffix-tree indexes (built over these sequence repositories) requires search-optimized storage, along with their efficient construction.

2. A DNA sequence is fairly static in nature and once a suffix-tree is constructed on it, then that suffix-tree will be used again and again for the purpose of searching only. So, the cost of one-time search-optimization of suffix-tree will yield better throughput, while using it for searching. This is necessary from the viewpoint of efficient sequence analysis.

3. There are many proposals (such as [21, 2, 12]) for efficiently constructing genome-scale disk-based suffix-trees. Whereas, there are very few efforts (such as [5, 28, 16]) to make suffix-trees search-optimized. Also, most of these efforts had not shown scalability of their search-optimization process for genome-scale suffix-trees.

1.1.2 Research Challenges

1. **Genome-Scale:** For the work on suffix-trees for genetic sequences to be useful, it should be done on genome-scale and not just at chromosome-scale. Otherwise, modern day computational biologists will find little use of it. But, the genome-scale suffix-tree is huge in size (typically around 100 GBs) and demands tremendous efforts for search-optimization.

2. **Structural Intactness:** In the process of search-optimization, we should be able to reduce the search-time without altering the logical structure of the suffix-tree.

[4] A Terabase refers to trillion base-pairs.

Because, if we modify the logical structure of the suffix-tree to suit our requirement, then we lose efficiency of various search algorithms, which rely on the original logical structure of the suffix-tree. For example, in [14], suffix-links are removed to achieve speedier construction of disk-based suffix-tree. But, in [21], it had been shown that the performance of exact match alignment anchor finding algorithm (used in [8]) with suffix-links is 2 to 5 times faster in comparison to the performance of the algorithm in which suffix-links are ignored.

1.2 Notations

Table 1.1 shows all the notations that are going to be used in this thesis. The used notations have either explicit or implicit meaning given at their usage locations, but this table describes all of them at one place.

Notation	Description
Σ	Finite set of possible alphabets used in a text sequence
S	Indexed sequence containing characters only from Σ
n	Number of characters present in S, i.e $\|S\|$
\$	Special termination character, not present in Σ
s_i, $S[i]$	Character at position i in S, must be drawn from Σ
$S[i \dots j]$	Substring of S starting at position i and length $(j - i + 1)$
S_i	Suffix of the sequence S starting at position i, i.e. $S[i \dots n]$
$\mathcal{I}$	Implicit suffix-tree constructed over the string S
$\mathcal{T}$	Suffix-tree constructed over the string S\$
l_i	Leaf in the suffix-tree $\mathcal{T}$ corresponding to the suffix S_i
$e(v)$	Incoming tree-edge to a node v of the suffix-tree $\mathcal{T}$
$sl(v)$	Suffix-link emanating from an internal node v
$E(v)$	Edge length of a node v, i.e. number of characters in edge-label of v
$L(v)$	Path length of a node v, the sum of edge lengths on the path from root to v
$\sigma(v)$	Substring $S[i \dots i + L(v)]$ associated with a node v in the suffix-tree (where i is the number associated with a leaf node l_i under v)
$parent(v)$	Parent internal node of a node v

Notation (cont.)	Description (cont.)
$child(v)$	Child node of an internal node v
$slNode(v)$	Suffix-link node of an internal node v
$slpNode(v)$	Suffix-link predecessor node of an internal node v
Q	Query string containing characters only from Σ
m	Number of characters present in Q, i.e $\|Q\|$
λ	Minimum threshold length for the search result
B	Suffix-tree storage block size
n_b	Total number of storage blocks of a suffix-tree
$\mathcal{PC}_{BS}$	Probabilistic cost of a sequence of blocks

Table 1.1: Notations

1.3 Background

In this section, we present an overview of the suffix-tree. We also review the popular algorithms used for the construction of in-memory suffix-trees as well as disk-based suffix-trees. Then, the search algorithm considered for the performance analysis is also discussed. The basic material for the background is obtained from [24].

1.3.1 Suffix-Tree

A *suffix-tree* is a highly compressed trie structure constructed over the set of all suffixes of a given text sequence. The usual compression in the suffix-tree is in a vertical direction (as shown in Figure 1.1), but a horizontal compression is also possible as mentioned in [20]. The various terms required to define a suffix-tree are described below:

Let $S = s_1 s_2 \dots s_n$ be a sequence of length n with each s_i drawn from an alphabet set Σ. A *substring* of S is a string $S[i \dots j] = s_i s_{i+1} \dots s_j$ for some $0 < i \leq j \leq n$. A *suffix* of S is a substring of S with $j = n$. In other words, a *suffix* is a part of the sequence starting at any location i in the sequence and continuing up to the end of the sequence. We represent a suffix starting at position i as S_i. Thus, there are exactly n suffixes from a sequence of length n, one for each position in the sequence.

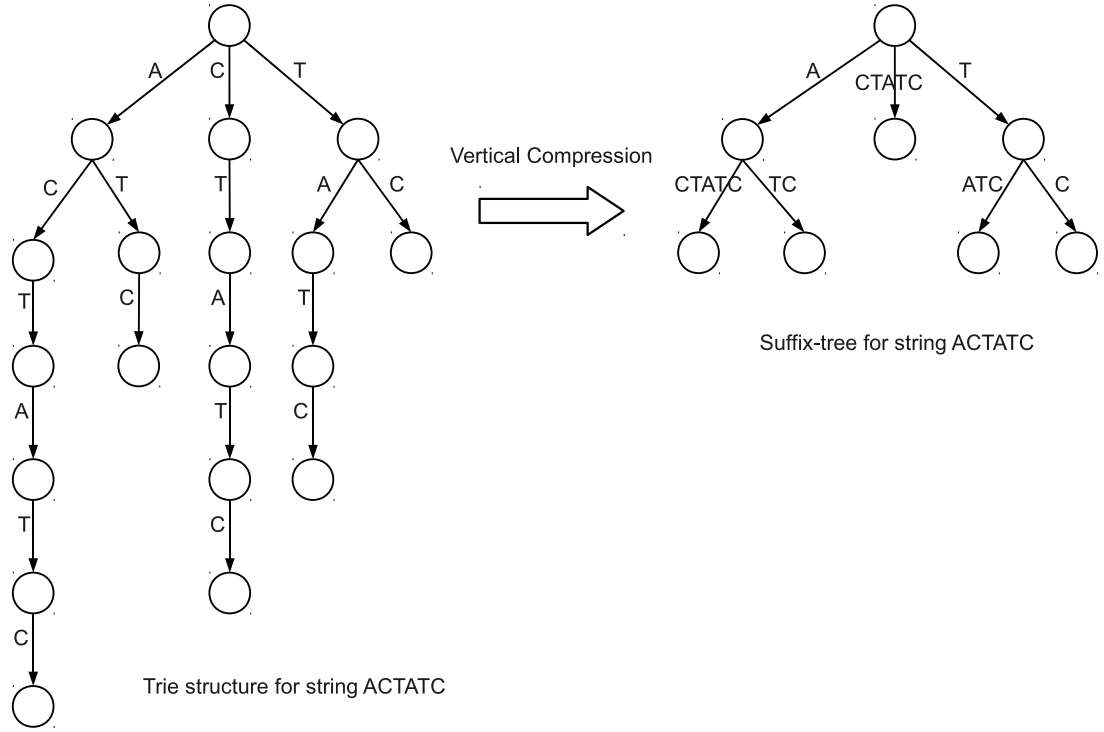


Figure 1.1: Vertical Compression of a Trie structure into a Suffix-Tree

Definition 1. ***(Suffix-Tree)*** *A suffix-tree for an n-character sequence S is a rooted directed tree with exactly n leaves numbered 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, i.e. S_i.*

Note that it is not possible to construct a suffix-tree for every sequence S as per Definition 1. If there is a suffix S_i that exactly matches another substring[5] $S[j \ldots k]$ (for $j \neq i$), then S_i ends at a non-leaf node in the suffix-tree constructed over S. So, the number of leaf nodes are not in one-to-one correspondence with the set of all suffixes of S (as is the case in Figure 1.1). Such a suffix-tree is called as *Implicit Suffix-Tree* (denoted by $\mathcal{I}$), as not every suffix can be exactly determined from this tree. In order to overcome

[5]This is the situation when a suffix of S is also a prefix of some other suffix of S.

this, a delimiter symbol (usually denoted by \$) is concatenated at the end of the indexed sequence. It is assumed that \$ does not appear anywhere else in the indexed sequence and $\$ \notin \Sigma$. With this assumption, it is guaranteed that there is a suffix-tree $\mathcal{T}$ (built over sequence $S\$$), in which every suffix of the sequence $S\$$ can be exactly located.

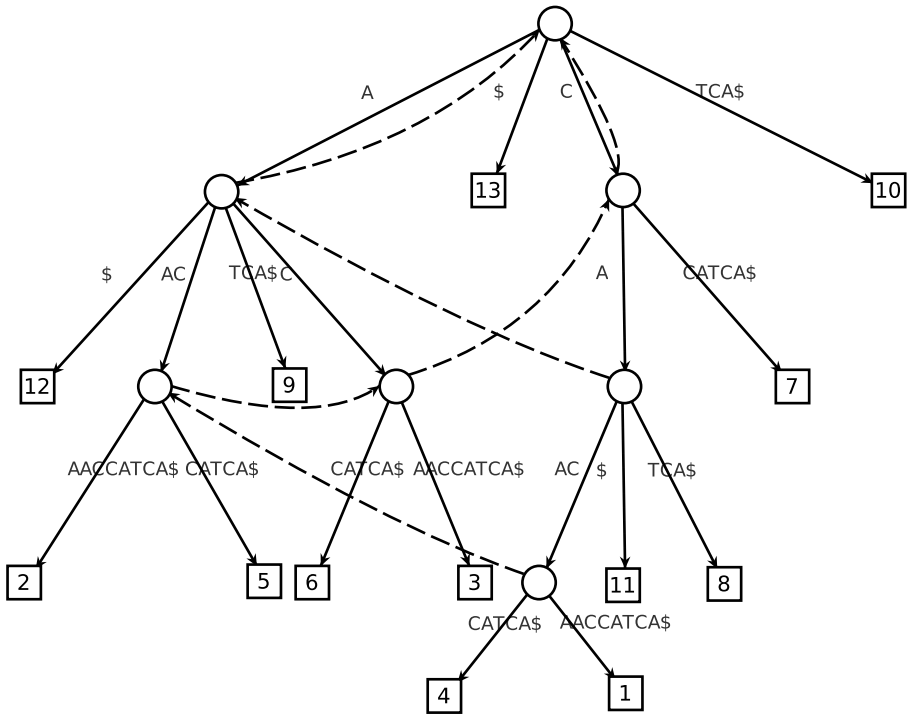


Figure 1.2: Suffix-Tree for DNA fragment CAACAACCATCA\$

The suffix-tree for a DNA fragment CAACAACCATCA\$ is shown in Figure 1.2. The circular nodes are internal nodes and the square nodes are leaf nodes. The number in each leaf node is the index i associated with the suffix corresponding to the leaf node. Note that the total number of leaf nodes are exactly same as the number of characters present in the sequence $S\$$. Solid lines indicate tree-edges and each tree-edge has an associated label, which is a substring of the sequence $S\$$. Dashed lines indicate suffix-links. The detailed explanation about suffix-links is given in the following section.

1.3.1.1 Suffix-Links

Although the definition given above is the commonly used one for suffix-trees, it does not incorporate one significant structural augmentation to the suffix-tree – namely, the notion of *suffix-links*. In practice, suffix-trees are augmented with additional edges called suffix-links that are necessary to achieve their linear time construction (in traditional construction algorithms, such as [27, 19, 26]) and also to significantly enhance the subsequent string searches (by improving the time-complexity of searching). 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. But, another kind of relationship exists between them, which is described in the following definition of a suffix-link.

Definition 2. ***(Suffix-Link)*** *Let $x\alpha$ denote an arbitrary string, where x denotes a single character and α denotes (a possibly empty) substring of the sequence to be indexed. 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. The entire set of suffix-links $sl(.)$, forms a tree with many source nodes, but with the root of $\mathcal{T}$ as the only sink node. In this alternative tree structure, there are exactly $L(v)$ edges on the path from an internal node v to the root node of $\mathcal{T}$. In Figure 1.2, the dashed lines between internal nodes of the tree are suffix-links, with the direction of the arrow indicating the pointer direction.

The suffix-links, in the present form, were first introduced by McCreight [19] 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 enables a much richer set of traversals over the suffix-tree resulting in many high-speed search algorithms. 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. Therefore, there are some proposals which resort to quadratic time construction of suffix-trees by completely dispensing with the suffix-links (such as [14, 25, 21]). But, some

of them introduce suffix-links at a later optional stage (which are described in following section). For the search algorithms such as Maximal Common Substring Search (refer Section 1.3.3 for details), the search time-complexity is linear in the size of the query string, when suffix-links are used. But, with the ignorance of suffix-links, the search time-complexity of these algorithms become quadratic in the size of query string. So, suffix-links are also required for efficient usage of suffix-tree for searching.

1.3.2 Construction of Disk-based Suffix-Trees

Prior to disk-based suffix-trees, the popularly used algorithms to construct (in-memory) suffix-trees were by Weiner [27], McCreight [19] and Ukkonen [26]. Each of these algorithms construct the suffix-tree in a time which is linear in $|S|$. However, all these algorithms required random accesses to the suffix-tree structure during its construction. This does not create problem while suffix-tree fits in main-memory. But, when the size of the suffix-tree exceeds the size of available main-memory, any of these construction algorithms has to use secondary storage (mainly disk), which introduces random disk accesses. The orders-of-magnitude difference between time taken for main-memory random accesses and disk random accesses causes these theoretically efficient algorithms to become practically infeasible for the construction of genome-level suffix-trees.

However, there are many efforts (starting from Hunt, et al. [14]), which tried to build genome-scale disk-based suffix-trees by completely ignoring methods suggested by traditional linear-time construction algorithms. On the other hand, there are some disk-based suffix-tree construction algorithms, who have not totally ignored the traditional linear time algorithms. For example, the proposals given in [4] and [21] used Ukkonen's algorithm or a variant of it as an intermediate stage. The choice of Ukkonen's algorithm over the Weiner's algorithm is due to its space-efficiency during the construction of suffix-tree. Also, it had been chosen over McCreight's algorithm due to its online nature and simpler mechanism for the suffix-tree construction. The high-level description of Ukkonen's algorithm is given in Figure 1.3.

As the Ukkonen's algorithm is online in nature, it incrementally builds the suffix-tree.

Algorithm Ukkonen (S)
Input:
S: Text sequence with n characters, i.e. $S[1 \dots n]$
Output:
$\mathcal{I}_n$: Implicit suffix-tree over the text sequence S

```
1.  I_1 ← Implicit suffix-tree for S[1...1]
2.  for i = 1 to n − 1 do
3.      for j = 1 to i do
4.          {LOCATE PHASE}
5.          Locate β = S[j...i] in I_i
6.          {INSERT PHASE}
7.          if β ends at a leaf then
8.              I_{i+1} ← add S[i + 1] to I_i
9.          else {β ends at an internal node, or at the middle of an edge}
10.             if from the end of β there is no path labeled S[i + 1] then
11.                 I_{i+1} ← split edge in I_i and add a new leaf
12.             else
13.                 I_{i+1} ← I_i{β already exists in I_i}
14.             end if
15.         end if
16.     end for
17. end for
```

Figure 1.3: Online Suffix-Tree Construction Algorithm of Ukkonen

The algorithm consists of two phases –

1. **Locate Phase:** In this phase, the appropriate edge in the temporary suffix-tree structure is located, to insert the character occurring just after the currently processed substring.

2. **Insert Phase:** In this phase, the next character from the sequence is inserted in the edge of the suffix-tree, which is determined earlier by *Locate Phase*. For this either a new edge and a new node are created or the edge-label of the located edge is expanded to include the new character.

With the sequence S consisting of n characters as input, the Ukkonen's algorithm outputs only an implicit suffix-tree over S. As we have seen earlier, in an implicit suffix-

tree, it is not necessary that every suffix of the sequence ends in a leaf. To obtain the suffix-tree where each suffix ends in a leaf, we have to add a terminal symbol \$ at the end of S and continue the algorithm with this $(n+1)^{th}$ character. The addition of terminal symbol \$ makes sure that no suffix will be a proper prefix of any other suffix of the same sequence. This is not the case in an implicit suffix-tree produced by Ukkonen's algorithm with just sequence S, if it ends with a non-unique character.

At first glance, the algorithm seems to have cubic complexity, i.e. $O(n^3)$, because locating a substring inside a suffix-tree requires scanning the whole sequence S in the worst-case. But, this is true only if the algorithm is implemented naively. When the algorithm is implemented with *few tricks*, it constructs suffix-tree in time which is linear in the size of the indexed sequence S, i.e. $O(n)$. For example, after processing of the current substring is completed, the next substring can be located within the suffix-tree (Locate Phase) in $O(1)$ time, by following the suffix-link from the current node. Other tricks for making algorithm linear-time can be found in [13].

Having looked at the traditional Ukkonen's algorithm, we now review the Trellis algorithm [21] which is the first disk-based algorithm to construct the genome-scale suffix-tree with suffix-links. The Trellis algorithm is a partition-based approach for constructing the disk-based suffix-tree. The different steps constituting the Trellis algorithm are described below:

1. **Prefix Creation Phase:** In the first step, variable length prefixes are determined. The main criterion in determining a prefix is that the suffix-tree partition corresponding to it, should fit in main-memory. Otherwise, the prefix will get extended further by appending more characters till the partition corresponding to each extended prefix fits in the main-memory. In this way, the suffix-tree partition size is controlled and the skew present in the sequence data is also handled.

2. **Partitioning Phase:** In this step, the indexed sequence S is partitioned and the suffix-tree for each of the sequence partition is built (which are called as *suffix subtrees*). Ukkonen's algorithm [26] is used in construction of these suffix subtrees. When these suffix subtrees are stored on the disk, they are segregated according to

the prefixes obtained in Step 1. Each segregated portion of the suffix subtrees is called a *prefix-based suffix subtree*.

3. **Merging Phase:** In this step, suffix subtrees corresponding to each variable length prefix are merged into a single suffix-tree partition. At the completion of this step, the partitioned suffix-tree without suffix-links is fully constructed. Here, the partitions of the suffix-tree are based on the prefixes obtained in Step 1. Whereas in Step 2, the sequence S is partitioned and suffix-tree is built for each substring partition. A point to note here is that the suffix-links that are created and used during Ukkonen's algorithm in step 2 are now discarded.

4. **Suffix-Link Recovery Phase:** Although this is an optional step, it is required to recover the suffix-links, whose crucial role in efficient searching is already highlighted in our earlier discussions. The Trellis algorithm recovers the suffix-links for each suffix-tree partition, one at a time. It starts with finding the suffix-link for the root node of a suffix-tree partition and then recursively finds the suffix-links for the child nodes in a depth-first manner.

A point to note here is that suffix-links are not fully utilized during the construction of suffix-tree and the recovery of suffix-links is optional. This is in contrast with the traditional algorithms, which heavily rely on the suffix-links for the suffix-tree construction, in order to get better theoretical bound. On the other hand, Trellis tries to reduce the active portion of the suffix-tree during construction by partitioning it. This smaller active portion of the suffix-tree is handled in-memory and thus avoids the random disk accesses. But, in achieving better localization during construction of the suffix-tree, the time-complexity of the Trellis algorithm becomes quadratic in $|S|$, i.e. $O(n^2)$.

1.3.3 Search Algorithm

The important application of the suffix-tree is that the searching of query string Q is performed in time that is linear in $|Q|$ and is not dependent on $|S|$, which is usually very large. That means the time-complexity for search is $O(m)$ and not $O(n)$, which is usually

the case with the methods other then the suffix-tree. There are many search algorithms available [13] for the suffix-tree, but in our work, we are focusing on the algorithm of *Maximal Common Substring Search* for the purpose of measuring performance improvement. The definition for the maximal common substring search is as follows:

Definition 3. ***(Maximal Common Substring Search)*** *Given a database sequence S, and a query sequence Q, the search which will locate all the occurrences of the longest matching substring of Q that is present in S, for each position in Q, is called the Maximal Common Substring Search. Usually, the matches having length greater than or equal to threshold λ is returned to the user, where the threshold λ is provided by the user.*

In other words, for each i, $1 \leq i \leq m$, locate all (i, j, k) triples where $j = max\ j'$ such that $Q[i \ldots i + j'] = S[k \ldots k + j']$ and $Q[i + j' + 1] \neq S[k + j' + 1]$ and $j \geq \lambda$.

The reason for the selection of this algorithm is that the first task of many well-known sequence matching tools (such as [7]) is to find maximal common substrings between two sequences. The algorithm of maximal common substring search was proposed in [6] and is also used in various search-optimization proposals, such as [5]. The algorithm for maximal common substring search is shown in Figure 1.4 and all the experimental search results presented in Chapter 5 are based on this algorithm.

In the algorithm, the function $charMatch$ returns the number of characters matched between the edge-label of the $leafChild$ and the portion of the query string still left to be matched. The $TraverseSubtree$ function visits all the nodes under the $currNode$ in depth-first manner (in order to find all occurrences of the maximally matched substring). The complexity of the algorithm is $O(m + occ)$, where occ is the number of times the *maximally matched* substring of $|Q|$ *occurs* in S. This algorithm heavily relies on the presence of suffix-links. In absence of the suffix-links desired linear time bound cannot be achieved and search has to begin from the root of the tree for every new substring of Q. And in such case, the worst-case search complexity will be quadratic in $|Q|$.

In the algorithm, bold lines indicate that random disk accesses have to be performed at those stages of the algorithm. Our focus in this thesis is to reduce the time incurred during these random disk accesses, as they form a dominant part of the search-time.

Maximal Common Substring Search Algorithm (S, $\mathcal{T}$, Q, λ)
Input:
S: Indexed sequence
$\mathcal{T}$: Suffix-tree over the indexed sequence S
Q: Query sequence
λ: Minimum match-length to be reported
Output:
$L = \{(q, l, d) \mid Q[q \ldots q+l] = S[d \ldots d+l], Q[q+l+1] \neq S[d+l+1], l \geq \lambda$, and l is maximal given $q\}$

1. $currNode \leftarrow$ root of $\mathcal{T}$; $matchedChars \leftarrow 0$; $L = \phi$;
2. for $i = 1$ to m do
3. $\quad mismatchFlag \leftarrow$ false; $leafMatchedChars \leftarrow 0$;
4. $\quad \mathcal{Q} \leftarrow Q[\ i \ldots m\ ]$
5. $\quad$ while $matchedChars < |\mathcal{Q}|$ do
6. $\quad\quad$ for $j = 1$ to $E(currNode)$ do
7. $\quad\quad\quad$ if $\mathcal{Q}[matchedChars + j] \neq currNode.edge[j]$ then
8. $\quad\quad\quad\quad mismatchFlag \leftarrow true$
9. $\quad\quad\quad\quad$ exit while loop (line 24)
10. $\quad\quad\quad$ end if
11. $\quad\quad$ end for
12. $\quad\quad matchedChars \leftarrow matchedChars + j$;
13. $\quad\quad parentNode \leftarrow currNode$;
14. $\quad\quad$ if currNode.child($\mathcal{Q}[matchedChars + 1]$) exist & is an internal node
15. $\quad\quad\quad$ **currNode $\leftarrow$ currNode.child($\mathcal{Q}$[matchedChars + 1]);**
16. $\quad\quad\quad$ continue to while loop (line 5)
17. $\quad\quad$ else if currNode.child($\mathcal{Q}[matchedChars + 1]$) exist & is a leaf node
18. $\quad\quad\quad$ **$leafChild \leftarrow$ currNode.child($\mathcal{Q}$[matchedChars + 1]);**
19. $\quad\quad\quad leafMatchedChars \leftarrow$ charMatch($leafChild$, $\mathcal{Q}[matchedChars+1 \ldots |\mathcal{Q}|]$);
20. $\quad\quad\quad$ exit from while loop (line 24)
21. $\quad\quad$ else
22. $\quad\quad\quad$ exit from while loop (line 24)
23. $\quad\quad$ end if
24. $\quad$ end while
25. $\quad$ if $(leafMatchedChars > 0)$ & $(matchedChars + leafMatchedChars \geq \lambda)$ then
26. $\quad\quad L \leftarrow L \cup (i, matchedChars + leafMatchedChars, leafChild.Start)$
27. $\quad$ else if $matchedChars \geq \lambda$ then
28. $\quad\quad$ **$L \leftarrow L \cup$ TraverseSubtree($currNode$)**
29. $\quad$ end if
30. $\quad$ if $mismatchFlag$ = true then
31. $\quad\quad$ **$currNode \leftarrow slNode(parentNode)$**
32. $\quad$ else
33. $\quad\quad$ **$currNode \leftarrow slNode(currNode)$**
34. $\quad$ end if
35. $\quad matchedChars \leftarrow matchedChars$ - 1;
36. end for

Figure 1.4: Algorithm – Maximal Common Substring Search

1.4 Thesis Contributions

The major contributions of this thesis are briefly described as follows:

- First, we show that the Stellar algorithm [5] (which had advocated a breadth-first or BFS approach and evaluated for chromosome-level sequences only) can be scaled to entire genome level, to produce the search-optimized layout for the disk-based suffix-tree. The resultant genome-scale layout by the Stellar algorithm also achieves significant reduction in the search-time, when it is compared with the search-time over the layout of the suffix-tree just after its construction.

- Secondly, we propose new layout reorganization algorithms, which are based on the Stellar algorithm. These reorganization algorithms introduce various conditions during the node-to-block[6] assignments. These conditions consider the properties of the suffix-tree nodes and blocks while rearranging them. Also, these reorganization algorithms try to determine the sequence of storage blocks, which reduces randomness in the disk accesses encountered during search traversal on the suffix-tree. The search-time over these resulting suffix-tree layouts not only outperforms the search-time over the original Stellar layout, but when it is compared with the search-time over the layout at the time of construction, the improvement is in the range of 25% to 75%.

- In our next step, we consider improving the physical structures of the suffix-tree internal nodes. In particular, we propose an embedding strategy whereby leaf nodes can be completely represented within their parent internal nodes, without requiring any space extension of the parent node's structure. Overall, this optimization results in a 25% reduction in the suffix-tree space occupancy. More importantly, when applied to the construction layout, it provides search-time improvements ranging from 25% to 85%, and when used in conjunction with the reorganized layouts, the speedup achieved in search-time is in the range of 50% to 90%.

[6]Here, a block refers to a storage block which is further explained in Section 3.2.1.

- Finally, we provide a variety of supporting statistics, such as intra-block localities, to help explain the observed behaviors of the various suffix-tree construction and search-optimized layouts.

1.4.1 Outline of Search-Optimization Process

In this section, we outline how the task of search-optimization of the disk-based suffix-tree layout has been accomplished. This is also represented diagrammatically in Figure 1.5.

- As the first step, we are using already constructed suffix-tree and then we try to make it search-optimized by rearranging its nodes and blocks. So, the process of search-optimization in our case is implemented as the post-construction processing of the suffix-tree. For the construction of the suffix-tree, we have used the Trellis algorithm [21], which constructs the suffix-tree with the suffix-links in a few hours on a typical desktop machine.

- The suffix-tree constructed by Trellis algorithm is in the form of multi-partitions, which is not suitable from the viewpoint of localization of both the tree-edges and suffix-links. So, we have merged all the partitions of the suffix-tree into one file. In doing so, we have not changed the internal storage of the nodes within the partition. We have just concatenated these partitions along with the top portion[7] of the suffix-tree. We call this layout of suffix-tree at this point of time as the *construction layout*, because node arrangement in this layout is the direct result of the suffix-tree construction algorithm.

- On the construction layout, we apply different algorithms for layout reorganization, which rearrange the nodes and blocks according to the conditions defined in the respective algorithms. So, in order to obtain search-optimized layout, we first apply node-to-block assignment, in which different layout conditions based on the properties of the suffix-tree nodes, such as number of characters present in the edge-label,

[7] In the multi-partition storage, the top portion of the suffix-tree is stored as a separate prefix file.

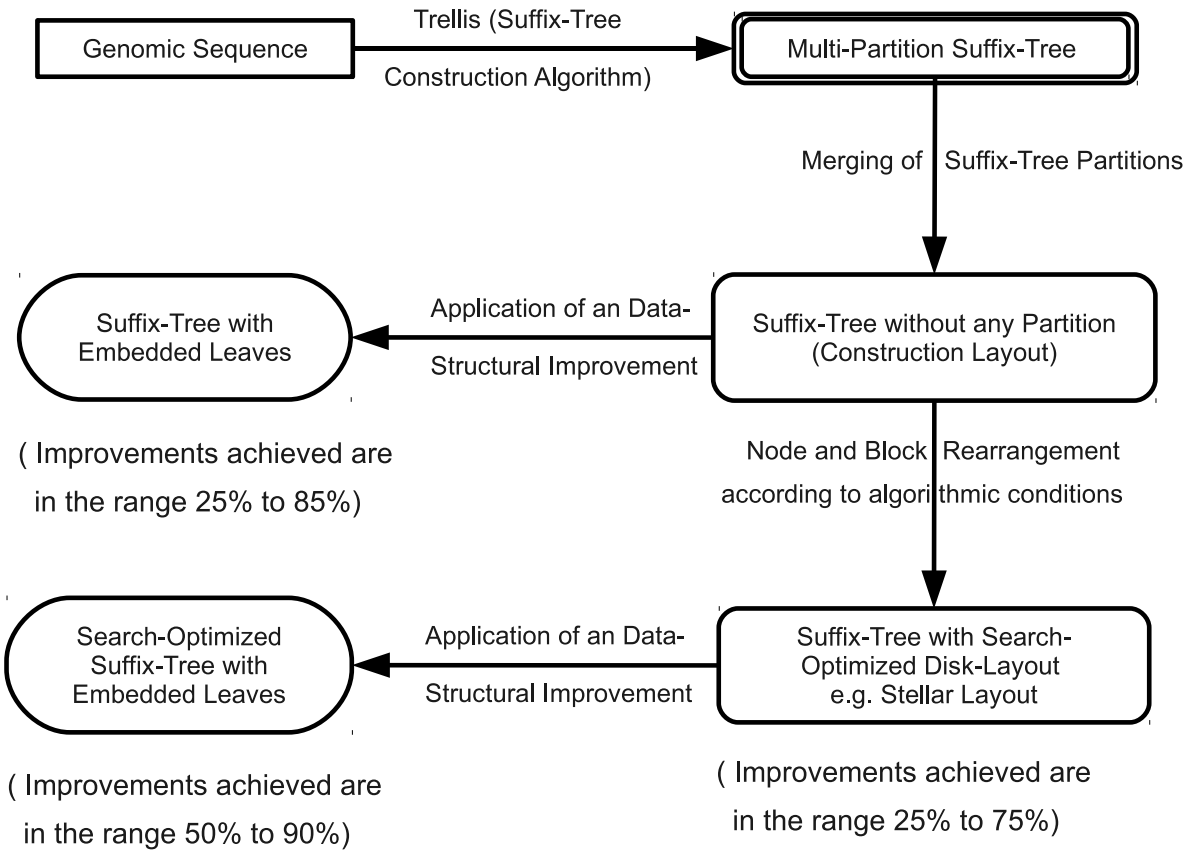


Figure 1.5: Outline of Search-Optimization Process

are considered. In the next step of layout reorganization, we apply inter-block optimization, which determines the optimized sequence of storage blocks by considering block-to-block traversal probabilities.

- Finally, the improvements in the physical structures of the suffix-tree internal nodes can be applied to various layouts. So, these improvements are applied to the construction layout, to the Stellar layout and also to the search-optimized layouts proposed by us. But, these improvements on different layouts are also considered as separate layouts in themselves, as they need separate storage and separate interface for searching over them. In the outline, we have mentioned one of these improvements which completely embeds leaf nodes in their parent internal nodes.

1.5 Organization

The remainder of this thesis is organized as follows: In Chapter 2, we review related research in the field of suffix-tree indexing. Details about implementation of various disk-based suffix-tree layouts proposed by us are presented in Chapter 3 and improvements related to physical structures of the suffix-tree internal nodes are illustrated in Chapter 4. The experimental evaluation is presented in Chapter 5. Finally, in Chapter 6, we summarize our conclusions and outline future research avenues.

Chapter 2

Survey of Related Research

The suffix-tree index exists for more than three decades. The initial construction algorithms for the suffix-tree index have emphasized on achieving theoretical efficiency. But in later years, the focus of suffix-tree construction algorithms have shifted to practical feasibility of suffix-tree construction, I/O efficiency and parallelization.

Although Weiner [27], McCrieght [19] and Ukkonen [26] gave linear complexity algorithms, the construction of the suffix-tree remained impractical for large sequences. For example, in the mid 90's, suffix-trees have found applications in computational genomics for exponentially growing genetic sequence repositories. For the genome-level sequences, the size of suffix-trees became so huge that it compelled the construction algorithms to use the disk. This is also necessary from the viewpoint of subsequent usage of suffix-tree for various search applications.

But, in [1], authors have predicted that the construction of the suffix-tree on secondary storage will remain impractical. The basis for their prediction was their observation that suffix-tree construction suddenly takes huge time, when the size of the suffix-tree exceeds available main-memory size. This sudden surge in construction time is due to random accesses made to the temporary tree structure during the construction process by the traditional construction algorithms. As the main-memory is designed for random accesses, the increase in construction time remained linear with the increase in indexed sequence size (till suffix-tree fits in main-memory). But, when the construction algorithm has to

use disk, the performance degrades due to random disk accesses. Because, disk accesses are already taking orders-of-magnitude more time than the main-memory accesses and this difference is more highlighted when accesses to the disk are random in nature (instead of serial). The random disk accesses cannot take advantage of the page cache maintained in the main-memory by the operating systems.

However, in recent years, the construction of the suffix-tree for large genome sequences become possible due to evolution in the construction algorithms. These algorithms have focused on achieving better locality of reference during construction of the suffix-tree, instead of getting better theoretical bounds (which is the main focus area of the traditional construction algorithms).

On the other side of the suffix-tree construction, the search aspect of the suffix-tree is largely ignored. The reason for this ignorance is the fact that till recent time the construction of the suffix-tree itself was very difficult for large sequences, such as the human genome sequence. Hence, the major focus remained on the efficient construction of the suffix-tree and not on the search-optimization of the suffix-tree storage.

So, we review the research work on the suffix-tree in two parts. In the first part, we see various efforts for the construction of disk-based suffix-trees. Then we go through different proposals for the search-optimization of disk-based suffix-trees. Although, our research focus in this thesis is always on the search-optimization of the suffix-tree, a quick review of the suffix-tree construction algorithms is also desirable.

2.1 Suffix-Tree Construction

In the last decade, there were numerous efforts for the construction of the disk-resident suffix-tree. Hunt, et al. [14] constructed one of the first disk-resident suffix-tree for human chromosomes, but with the removal of suffix-links to achieve better locality of reference. Hunt's method is also one of the first to introduce prefix-based partition method for the construction of disk-based suffix-tree. In their method, length of all the prefixes are incremented till the partition corresponding to each prefix, fits in the main-memory. Thus, all prefixes have same fixed length. So, this approach does not handle skew present

in the data which results in suffix-tree partitions of varying sizes. The complexity of Hunt's method is $O(n^2)$, where n is length of the sequence on which suffix-tree is to be constructed. After this initial effort, there are many efforts to build the suffix-tree on secondary storage in more and more efficient way. Here, we mention few of them, which have significantly enhanced the process of suffix-tree construction.

Bedathur, et al. [4], provided the buffering strategy *TOP-Q*, to build the disk-based suffix-tree with suffix-links for larger sequences. Although TOP-Q is built upon linear complexity algorithm of Ukkonen [26], their significant contribution is to show that complete suffix-trees can be built on larger sequences. They have also shown that the array representation of the suffix-tree nodes are superior than the space-efficient linked-list representation.

The approach of Tata, et al. [25] is similar to that of Hunt's, as they abandon the suffix-links for better locality of reference and their approach has the time-complexity $O(n^2)$. But, even with these loopholes, they are the first to scale upto human genome level. In their approach they are able to reduce massive random disk accesses and hence outperform the Hunt's and TOP-Q approaches for the construction of the suffix-tree.

The first complete genome-scale disk-based suffix-tree was constructed in the *Trellis* by Phoophakdee, et al. [21], in which they used partition and merge approach with variable length prefixes in order to handle the skew present in the genome sequence. The Trellis algorithm initially constructs suffix-tree without suffix-links. But, in the last optional phase, there is a choice to recover suffix-links. The detail mechanism for suffix-tree construction by the Trellis algorithm is already explained in Section 1.3.2. Like other partition-based approaches, the time-complexity of the Trellis algorithm is $O(n^2)$. But, due to its majority of operations are in-memory, it completely avoids the random disk accesses. Later on, they also come up with newer algorithm - *Trellis+* [22], which is not just speedier than Trellis with the same memory constraints, but, it also removes the restriction of keeping whole indexed sequence in the main-memory.

After Trellis, there are a few proposals for the construction of genome-scale suffix-trees. In *DiGeST* [2], Barsky, et al. claimed the construction of suffix-trees which can handle the

sequences greater then the human genome sequence. In the DiGeST method, the indexed sequence is first partitioned and each partition is then sorted lexicographically. After that, sorted partitions are merged using suffix-arrays to yield the final suffix-tree. But, again in their effort, they discarded suffix-links, and hence we are not considering it further, because of our focus on suffix-links for search applications. In continuing their effort to build suffix-tree for very large sequences, Barsky, et al. also come up with the algorithm of B^2ST [3], which can construct suffix-tree even for text sequences of size 12 GB, but again without suffix-links. They have also implemented an important enhancement in this effort – the indexed sequence need not be kept in the main-memory, during the construction of a suffix-tree.

Recently, *WAVEFRONT* [12] by Ghoting, et al., does speedier construction of the suffix-tree with limited memory. Unlike the "partition-and-merge" approach of the earlier efforts, WAVEFRONT constructs the suffix-tree by dividing it in disjoint tiles (which are similar to diagonal partitions of the suffix-tree, instead of usual vertical partitions). The WAVEFRONT algorithm outperforms Trellis algorithm when the indexed sequence does not fit fully in the main-memory. The parallel version of the WAVEFRONT algorithm constructs the suffix-tree in a few minutes on the high-performance systems.

2.2 Suffix-Tree Search-Optimization

In this section, we review various efforts for search-optimization of the suffix-tree. Although these efforts are less in number than the suffix-tree construction efforts, their role in search-time reduction is very important. This is because, after one-time construction, suffix-trees are going to be used many times for searching.

Bedathur, et al. [5] proposed *Stellar* layout, to optimize disk-based suffix-tree from search perspective. Their main contribution is to show that a balance between tree-edge localization and suffix-link localization can be achieved, which results in reduced I/O operations for search algorithms involving suffix-links. But, their approach of localizing tree-edges and suffix-links had not considered any suffix-tree property which is also important from the search perspective. Therefore, Stellar arbitrarily brings the tree-edges

and suffix-links together. Due to this, it achieves unnecessary localization at some places, while at other places, it has to content with lower localization in comparison to overall possible localization of the suffix-tree nodes. So, although Stellar achieves noticeable performance improvements, it is arbitrary in nature.

The efforts in [16], tried to give theoretically better disk-layout for suffix-tree. But, there main focus was to show the usefulness of secondary storage for the suffix-tree and to show how the suffix-tree can be updated efficiently. Both of these areas are not focused in our work. Because, DNA sequences are fairly static in nature and considering huge size of suffix-tree there is no other option then the secondary storage in the present time.

The efforts in [9], though not directly related to suffix-tree, provides the tree-layout which give near-optimal suffix-tree layout for exact match searches.

The work of Phoophakdee, et al. [21] showed that for exact match alignment anchor finding algorithm [8], the suffix-link version has better performance then the non-suffix-link version. But, they nowhere compare the search results involving suffix-links with the search results from other layouts.

The search-optimization effort in [23], improves the search-time after doing post-processing on the constructed suffix-tree. But, their medium of storage to have improvement in the search-time is suffix-array (which is a space-efficient alternative to the suffix-tree, proposed by Manber, et al. [18]) augmented with a small trie structure, which they called as *LOF-SA*. Our work is different from this approach as we have retained the suffix-tree structure for all kind of search-optimizations.

The CPS-tree proposed by Wong, et al. [28] is search-optimized version of suffix-tree which reduces the number of page-faults during the search traversal. It partitions the suffix-tree in various local trees, where each local tree gives priority to the nodes with maximum number of leaf nodes under it, for localization. It also implements forward links which limits the worst case logical block access to $O(log\ n)$, where n is number of characters in the indexed sequence. But, in this search-optimization effort, they just resort to exact-match searching which avoids complex traversal patterns by not considering suffix-links. In this effort, authors have not demonstrated scaling of the CPS-tree to the

human genome sequence.

Like many other search-optimization approaches (such as Stellar, LOF-SA), our approach is also a post-construction approach. So, we take already constructed suffix-tree from the disk and rearrange its nodes and blocks, to obtain the search-optimized suffix-tree layout. Our approach for search-optimization of suffix-tree through layout reorganization (Section 3.2) is independent of any particular suffix-tree construction algorithm. However, the suffix-tree search-optimization through improvements in physical node structure (Chapter 4) is dependent on the implemented data-structure to a certain extent.

Chapter 3

Search-Optimized Suffix-Tree Layouts

In this chapter, we propose search-optimized layouts for efficient storage of suffix-trees on disk, which give significant search performance improvements (as we quantitatively see later in Chapter 5). These new layouts are based on the Stellar layout, proposed in [5]. So, we also look into the details of the Stellar and other existing suffix-tree layouts. But, before moving onto the details of these existing as well as new layouts, we analyze the requirement for the search-optimization and different levels of implementation of search-optimization process.

3.1 Requirement for Search-Optimization of Suffix-Tree Storage

The disk access time is the major constituent of the search-time for the search performed on the disk-based suffix-tree. In our case, disk accesses took more than 99% of the total search time. This is because of the following two reasons:

1. The major task to be performed during the search traversal is to visit different nodes of the suffix-tree, which causes random accesses to the suffix-tree file which resides on

the disk. On the other hand, processing in terms of character comparison constitute very less fraction of the overall search-time (refer to algorithm in Figure 1.4).

2. There is orders-of-magnitude difference between main-memory access time and disk access time.

So, the disk access time remains a performance bottleneck in the searching over the disk-based suffix-tree.

The huge cost of these random disk accesses during search traversal is mainly attributed to the enormous size of the disk-based suffix-tree. Because, for smaller suffix-trees, the caching effect of the operating system would have reduced the impact of huge cost of random disk accesses. So, it is desirable to have some storage optimization within the disk-based suffix-tree which makes it more tolerant to randomness occurring in disk accesses and thus helps in reducing the search-time. As mentioned in Section 1.1.2, this process of storage optimization should not alter the logical structure of the suffix-tree. So, the alternative ways for achieving search-optimization are mentioned as follows:

1. Search-optimization of the suffix-tree through just *rearrangement* of various portions of it, i.e. *layout reorganization* of the suffix-tree.

2. Search-optimization of the suffix-tree through *improving physical structures of suffix-tree internal nodes* without affecting the logical structure of the suffix-tree.

The first option is the subject matter of this chapter, which totally focus on finding appropriate rearrangement of various suffix-tree portions, in order to make it search-optimized. While the search-optimization process using second option is explained in Chapter 4.

3.2 Reorganization of the Suffix-Tree Layout

The basic constituent of the suffix-tree is a node. But, from the perspective of the operating system, the suffix-tree stored on disk is just a collection of different disk pages.

So, in order to enhance the localization of nodes within a group of disk pages, the layout reorganization process can be performed at two different levels of the suffix-tree storage. These two levels are mentioned below:

1. **Node Level Rearrangement:** Finding an appropriate group of disk pages (which we call as *Block*) for a suffix-tree node.

2. **Block Level Rearrangement:** Finding the optimized sequence of blocks, which reduces randomness in disk accesses.

These two rearrangement levels are described (in more detail) in the following subsections:

3.2.1 Node Level Rearrangement: Node-to-Block Assignment

The main aim of the node rearrangement is to reduce search-time by physically localizing neighboring nodes[1]. This rearrangement of nodes should be done in such a way that the next node required during search traversal (which is either child or suffix-link node of the currently processed node) is either in the main-memory or at the least possible distance on the disk.

For this, we have to fix the size of a storage block which decides the extent of localization. Due to aggressive fetching of disk pages in advance by an operating system, we have set the size of a storage block to 64 kB. So, our first goal should be to increase the relevance of a node to the other nodes present in the same block, such that fetching of the extra amount of data by the operating system can be utilized meaningfully. In this way, we can also avoid bringing of extraneous pages into the main-memory as far as possible.

In order to localize nodes within a block, we have to consider properties of suffix-tree nodes which are important from the search perspective. For example, if a node have all the internal child nodes present, with their incoming edge representing just one character, then the suffix-link node of the current node will never be visited. So, there is no need

[1]Neighboring nodes are those nodes which are related with each either other by parent-child relationship or by suffix-link predecessor-follower relationship.

to place the suffix-link node of the current node in its vicinity. Similarly, we try to bring the most relevant nodes together by identifying certain properties (as we see in detail in Section 3.4.2), so that they can form a cluster of relevant nodes within a block. Now, if during the search traversal, any one of the node from this cluster gets visited, then there is a high probability that the next required node is obtained from the cluster itself. This criterion is also used in measuring the layout goodness, as we see later in Chapter 5.

3.2.2 Block Level Rearrangement: Determining Optimized Block Sequence

In the previous criterion of node level rearrangement, we looked at the question: how can the relevance of nodes within a block of a suffix-tree be increased? Now, we ask a broader question: How are the related blocks distributed within the suffix-tree file (which resides on the disk)? For example, if there is a high probability to go to a block from the current block, then that block should be placed physically nearer to the current block on the disk. Because, placing relevant blocks nearer to each other, reduces seek time and rotational latency[2], which are expensive operations in comparison to data transfer time of the disk, when disk accesses are random in nature [17].

A point to note here is that the process of determining optimized block sequence depends on how nodes are assigned to blocks. Because, this assigning of nodes determines how much these blocks are related with each other. Although most of the earlier efforts for search-optimization of suffix-trees had looked into node-to-block assignment, they have largely ignored the determination of optimized block sequence.

Both the above criteria for structural rearrangement play important role in the achievement of search-optimized storage for the suffix-tree.

[2]Assuming there is less or no fragmentation within the suffix-tree file.

3.3 Partition-based Suffix-Tree Layouts

Now, we analyze the storage of existing construction layouts, which used either partition (or tiling[3]) based approaches for the efficient construction of the suffix-tree. This type of storage mechanism has following advantages and disadvantages:

- **Advantages of Partition-based Layouts:**

  1. **Easier to Construct:** In order to take the advantage of locality of reference for efficient suffix-tree construction, various construction approaches have considered suffix-tree as combination of different parts. If the construction approaches ignore the locality of reference (like Ukkonen's algorithm [26]), then there is a huge increase in the random disk accesses, which in turn, drastically increase the construction time of the disk-based suffix-tree.
  2. **Possibility of Parallelization:** As the suffix-tree is divided into different parts, the parallel construction of these parts is also possible (as shown in the approach of [12]). Parallelizing can significantly reduce the suffix-tree construction time.

- **Disadvantages of Partition-based Layouts:**

  1. **Limitation to Localization:** In most of the approaches, the suffix-tree partitions are created vertically, considering tree-edges only. So, the localization of suffix-links within these suffix-tree partitions cannot be achieved as they span across the tree. Hence, with these kind of layouts, we can never get balance between tree-edge localization and suffix-link localization, if the storage mechanism for partitions are not going to be changed.
  2. **Complex Structure:** With this kind of storage mechanism, it is hard to determine the exact locations of suffix-link nodes across different partitions. First, we have to keep information about $\sigma(v)$ of an internal node v (during

[3]A different kind of division of the suffix-tree into disjoint parts, which is based on the start values of incoming edges to the suffix-tree internal nodes [12].

the search traversal) and then we have to perform extra calculations for determining partition of $slNode(v)$. This is in contrast to suffix-tree storage in one file, where it is straightforward to find the location of $slNode(v)$.

So, these partition-based layouts cannot be utilized directly to build search-optimized layouts, considering their limitation in achieving overall localization of nodes. In contrast to these partition-based layouts, all the search-optimized layouts that we have developed contain the suffix-tree in one file without dividing it into different partitions.

3.4 Search-Optimized Suffix-Tree Layouts

In the initial part of this section, understanding about various existing suffix-tree layouts have been provided. After that various types of search-optimized layouts proposed by us are mentioned, along with their implementation details.

3.4.1 Understanding Existing Suffix-Tree Layouts

Based on the purpose of layout origination, the existing suffix-tree layouts can be categorized into two types. These categories are mentioned as follows:

1. **Construction Layouts:** These are the layouts which are direct result of the suffix-tree construction. These type of layouts have focused on how to build the suffix-tree as quickly as possible and the handling of search aspect have been comparatively ignored. An example of this type of layout is the partition-based suffix-tree layout at the end of suffix-tree construction by the Trellis algorithm [21].

2. **Search-Optimized Layouts:** These type of existing layouts consist of layouts which have mainly focused on increasing the search performance of the suffix-tree, by optimizing its storage through a post-construction reorganization process. For example, SBFS [9] and Stellar [5], which can improve search performance by localizing nodes of a suffix-tree.

Now, we go through the details of existing layouts, which are mentioned as follows:

3.4.1.1 Trellis Layout

As we have seen in Section 1.3.2, each partition in the Trellis layout contains a different sub-tree of the suffix-tree which also corresponds to a unique variable length prefix. As each partition is processed within the main-memory, the Trellis algorithm had not considered any kind of block size for the disk storage of the suffix-tree. The nodes in a partition are arranged in a depth-first order, without considering any kind of suffix-tree node properties.

We have built a layout from the Trellis layout, which has whole suffix-tree stored in one file (as we have seen in Section 1.4.1). We call this layout as the *construction layout*, because it is the result of merging of the suffix-tree partitions, which are constructed by the Trellis algorithm directly from the genome sequence.

3.4.1.2 SBFS Layout

SBFS layout was proposed by Diwan et al. in [9]. The basic strategy in constructing the SBFS layout is to pack nodes in a block by doing localized breadth-first traversal in the source layout, until either the block is filled or there are no more nodes left to visit. This packing of nodes starts with the root node as the initial node for the first block. For the remaining blocks the initial nodes are provided by the FIFO queue of the nodes that are left to be visited (from the earlier breadth-first traversals). So, basically in this layout, SBFS algorithm tries to put the child nodes in the same block as that of the parent node.

For the suffix-tree, this layout has maximum possible localization of tree-edges, if suffix-links are completely ignored during the rearrangement process. But, when the suffix-links are taken into consideration, the theoretical bound given in [9] may not hold, as the suffix-tree becomes graph (and not just tree) after the consideration of suffix-links.

3.4.1.3 Stellar Layout

In Stellar layout [5], the localization criterion considers suffix-links at par with tree-edges and then rest of the approach for rearrangement of nodes is similar to SBFS. But, in the SBFS layout the child nodes are always available to place with the parent node during

rearrangement process (because, the considered suffix-tree structure is acyclic). Whereas, this is not the case with the Stellar layout, in which the child nodes may or may not be available for placement near the parent node (because, now the structure is a cyclic graph due to consideration given to suffix-links and so a node can be demanded from a suffix-link predecessor before its parent node). So, it seems that Stellar will give better performance in search involving suffix-links. Whereas, the SBFS will give better performance in the search not involving suffix-links (like exact match search).

In this layout, no particular criterion is considered for node localization. That is, Stellar tries to bring all the neighboring nodes[4] together without checking the degree of relevance between those nodes. This method of localization in which no properties of suffix-tree are taken into consideration, creates certain undesired node arrangement in the final layout. For example, consider the scenario in which an internal node v does not have any internal child present. This means node v has two or more leaf child nodes only. So, when the search traversal reaches node v, it is highly likely that $slNode(v)$ gets visited from v. Hence, we want to place $slNode(v)$ physically nearer to node v. But, there can be more link-predecessor nodes of $slNode(v)$ other than the node v, which also demand $slNode(v)$ for placing nearer to them. But when node v is the only link-predecessor of $slNode(v)$ then v should have gotten $slNode(v)$ nearer to it. We found that there are 13.5 million such cases in the suffix-tree built over the human genome sequence. But in Stellar we found that only 6 million $slNode(v)$ are localized to node v. This means that in *55%* of such cases, the Stellar layout had ignored the necessary and non-conflicting availability of suffix-link nodes for localization. This happens due to the first-come-first-serve approach adopted by the Stellar for localization of the suffix-tree nodes.

So, the usual search traversal patterns are ignored by the Stellar algorithm during the rearrangement of suffix-tree nodes.

[4]In this thesis, when we talk about neighborhood of a node, it is always logical neighborhood in which nodes are related with either parent-child relationship or suffix-link predecessor-follower relationship.

3.4.2 BFS Layouts: Node-to-Block Assignment

Now, we see the details about the variant of BFS layouts suggested by us. As described in Section 3.2, there are two levels in the reorganization of any layout, namely - localization based on node rearrangement and localization based on block rearrangement. In this section, layouts based on node rearrangement are proposed, whereas layouts based on block rearrangement are explained in Section 3.4.3.

A point to note here is that we are focusing only on the localization of internal nodes skipping the criteria of localization of leaf nodes. Although internal nodes are less in number than leaf nodes (about 70% to 80% of leaf nodes), still localization of internal nodes are given more attention from the search-optimization point of view. This is because of the following reasons:

1. During the search traversal, majority of accesses in the suffix-tree require internal nodes. The combination of tree-edge traversal and suffix-link traversal during visits to different internal nodes causes disk accesses to be random, and create performance bottleneck in search. On the other hand, accesses to leaf nodes are restricted by the sub-tree below a particular internal node. Because, unlike internal nodes, the leaf nodes have an in-degree of just one.

2. Usually, the size of internal nodes are greater then the size of leaf nodes (4 times larger in our case). So, even though internal nodes are less in number, the space occupied by them on disk is quite large in comparison to the space occupied by leaf node file (near to 3 times larger). Hence, from the localization point of view, larger file needs more consideration than the comparatively smaller file.

Before going into the details of various layouts based on node rearrangement, let us see the notion of *nodeType*, which is actually a numbering mechanism for combining two different information in one number.

The *nodeType* of an internal node v depends on two values:

1. Number of characters present in the edge-label of v (call it as *inChar*).

2. Number of internal child nodes of v (call it as *iChildCount*).

Based on these two entities, the value for the *nodeType* can be uniquely assigned as

$$nodeType(v) = (inChar - 1) * (|\Sigma| + 1) + iChildCount$$

For a DNA sequence, $|\Sigma|$ equals 4 (as the characters in DNA sequence are - A, C, G, T). So, for the suffix-tree built on a DNA sequence the *nodeType* of a node[5] v can be calculated as

$$nodeType(v) = (inChar - 1) * 5 + iChildCount$$

Now, let us see an example, how this notion can be useful for determination of the vicinity of a node in search-optimized layout. Inside the suffix-tree constructed over a DNA sequence, if the *nodeType* of a node v is 4 and the *nodeTypes* of all its child nodes are less than 5, then the suffix-link from v can never be visited. This is because, *nodeType* of v is 4 only when its edge-label has just 1 character and all 4 child nodes exist as internal nodes. The *nodeTypes* less than 5 represents single character in the edge-label of its child nodes. So, when the search traversal reaches v, the only character in its edge-label must have matched and since all 4 internal child nodes of v have just 1 character in their edge-labels, the search traversal must move on to any one of the child nodes. But, on the contrary, if some of the internal child node's edge-label contains more than 1 character (i.e., $nodeTye \geq 5$), then there are chances that the search traversal may mismatch characters on that edge and in that case the $slNode(v)$ from v will get visited.

So, *nodeType* is a single notion, which provides the information for both the characters in the edge-label to a node and the number of internal child nodes. Due to this reduction in dimensions of available information, the visual analysis of suffix-tree data becomes easier, as shown below.

The percentage of nodes having different *nodeTypes* for the suffix-tree built on the human genome sequence are shown in Figure 3.1[6]. In this figure, we can see that almost

[5]From now onwards, a node refers to an internal node, if not specified explicitly.

[6]If not mentioned specifically, the suffix-tree used in this thesis is always built on the human genome sequence.

60% of nodes have *nodeType* less than or equal to 4. This means that these are the nodes with just one character in their edge-label. But, in this 60% of nodes, majority of them have very few child nodes or no child node at all (for example, nodes with *nodeType* 0, 1 or 2). So, when search traversal reaches these kind of nodes, it is very likely that suffix-links from these nodes will be followed, instead of following tree-edges leading to their child nodes. So, for such nodes, during search-optimization process, suffix-link nodes are given more preference than their child nodes for localization.

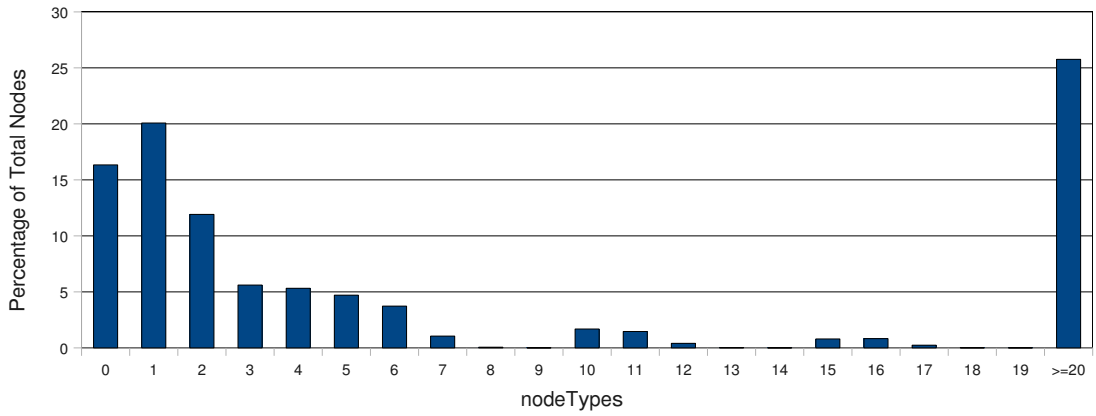


Figure 3.1: Percentage of Nodes vs. nodeType for the Suffix-Tree built over the Human Genome Sequence

Figure 3.1 does not include the classification of nodes according to different depth levels of suffix-tree which is provided in Figure 3.2. In Figure 3.2, we can observe that up to a certain level (i.e. level 13), the number of nodes present is very much less than the total nodes present in the suffix-tree. But, just after that level, majority of the suffix-tree nodes are accommodated within 5 to 6 levels. For the remaining levels of the suffix-tree, number of nodes goes on decreasing. So, our primary focus for the search-optimization is on the middle dense levels.

As we have seen in analysis of Figure 3.1, majority of nodes in suffix-tree have their *nodeTypes* as either 0, 1 or 2. According to Figure 3.2, these type of nodes are concentrated in the middle dense levels and require suffix-link nodes to be placed nearer to them. So, requirement of giving more preference to suffix-links than tree-edges starts at early levels in the suffix-tree. Also, with the increase in levels of suffix-tree, the percentage

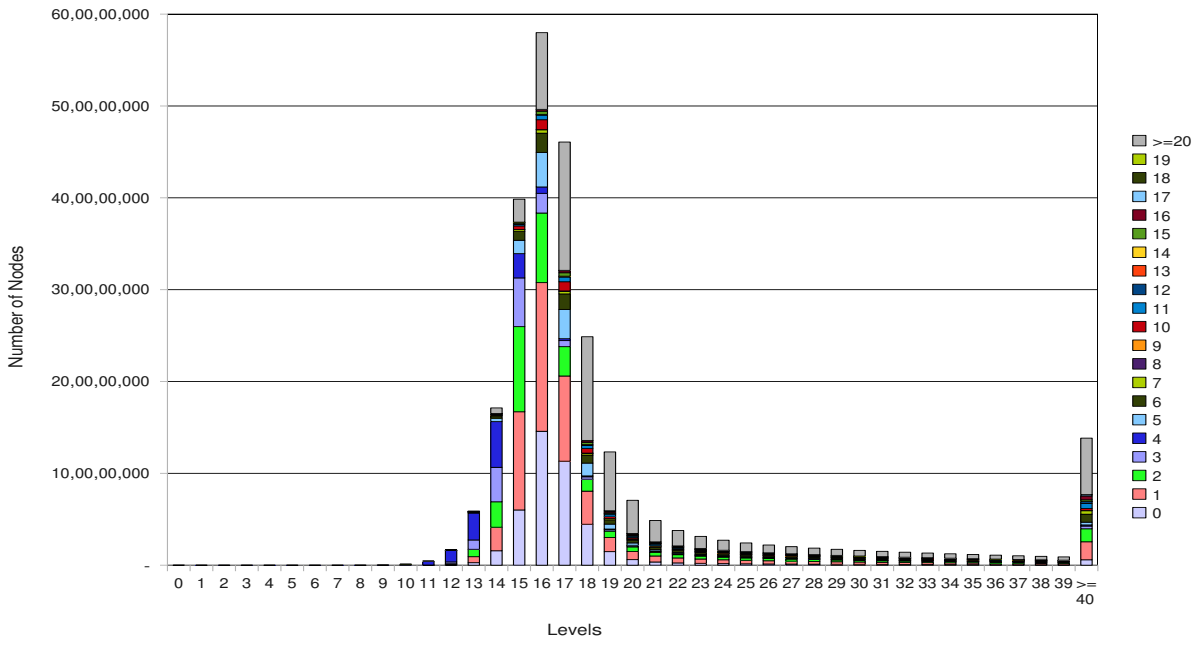


Figure 3.2: Number of Nodes vs. Tree Levels for each nodeTypes in the Suffix-Tree built over the Human Genome Sequence

of nodes with multi-character edge-labels increase. These multi-character edges help in determining whether a suffix-link node should be localized with its predecessor node or not. The analysis presented here is static in nature, as we have used only the suffix-tree structure for the analysis.

3.4.2.1 Common Rearrangement Function

After going through the analysis based on the numbering scheme of *nodeType*, let us now see, a common rearrangement function which is used in various layouts for actual rearrangement of nodes. This function is shown in Figure 3.3.

This function converts an existing suffix-tree layout into the desired layout by considering the block size and two bitmaps - $markedForLinkPredecessor$ and $followLink$. These two bitmaps play an important role in deciding the vicinity of a node.

The bitmap $markedForLinkPredecessor$ indicates that whether child node should be placed near to parent node or should it be left for its suffix-link predecessor node. Whereas bitmap $followLink$ indicates whether a node should bring its suffix-link node

Common Rearrangement Function($\mathcal{T}$, B, $markedForLinkPredecessor[\,]$, $followLink[\,]$ **)**
Input:

- $\mathcal{T}$: Suffix-tree given by the user
- B: Block size
- $markedForLinkPredecessor[\,]$: Bitmap indicating whether an internal node should be placed near to its parent node or near to its suffix-link predecessor node
- $followLink[\,]$: Bitmap indicating whether suffix-link node of an internal node should be brought near to it or not

Output:
Suffix-tree $\mathcal{T}_2$ which is similar to $\mathcal{T}$ in structure and semantics, but different in node arrangement

```
1.  GlobalQ.push(T.rootNode)
2.  currBlock ← T.rootNode
3.  currBlockSize ← sizeof(T.rootNode)
4.  markVisited(T.rootNode)
5.  while GlobalQ.notEmpty() do
6.      currNode ← GlobalQ.pop()
7.      LocalQ.push(currNode)
8.      while LocalQ.notEmpty() do
9.          currNode ← LocalQ.pop()
10.         for each childNode of currNode do
11.             if not markedForLinkPredecessor[childNode] then do
12.                 if notVisited(childNode) then do
13.                     currBlock ← currBlock + childNode
14.                     currBlockSize ← currBlockSize + sizeof(childNode)
15.                     LocalQ.push(childNode)
16.                     markVisited(childNode)
17.                 end if
18.             end if
19.         end for
20.         if followLink[currNode] & notVisited(linkNode) then do
21.             currBlock ← currBlock + linkNode
22.             currBlockSize ← currBlockSize + sizeof(linkNode)
23.             LocalQ.push(linkNode)
24.             markVisited(linkNode)
25.         end if
26.         if currBlockSize ≥ B then
27.             write nodes from currBlock up to size B onto the disk
28.             currBlock ← overflow nodes of currBlock
29.             currBlockSize ← currBlockSize − B
30.             transfer all nodes from LocalQ to GlobalQ
31.         end if
32.     end while
33. end while
```

Figure 3.3: Common Rearrangement Function

into its vicinity or not. These two bitmaps are created and initialized by the different layout algorithms according to their logic and passed on to this function. A point to note here is that each suffix-tree node is assigned to a block only once, during the rearrangement process of this function. Also, a node (which is pushed into the *localQ*) is considered only once for the visit to its child nodes and its suffix-link node. So, it is straightforward to observe that the time-complexity of this function is linear in the number of nodes of the suffix-tree. But, the number of nodes are linearly dependent on size of S (about 0.7 times $|S|$, for a typical human genome sequence [21]). Hence, the time-complexity for this function is linear in $|S|$, i.e. $O(n)$. The output of this function is a suffix-tree, which is same as input suffix-tree, but with different node arrangement in its storage.

3.4.2.2 1Cr4Cd Layout

As we have seen (in our analysis based on *nodeTypes*) that there are substantial chances of visiting suffix-link nodes even in the early levels of the suffix-tree. This layout generation algorithm tries to exploit this observation, by preferring suffix-link nodes over child nodes for localization with nodes not having *nodeTypes* as 4.

In this layout, all the child nodes of a node are placed near to it, if it has 1 character in its edge-label and all the 4 internal child nodes exist. Otherwise, all child nodes of v are left for their suffix-link predecessor nodes, if there are any. Due to this condition of *1 Character and 4 Children*, this layout is called as *1Cr4Cd*. As we can see in Figure 3.2 that the portion of suffix-tree till level 14 has significant number of nodes with $nodeType = 4$. So, in such cases, tree-edges are given full preference for localization and suffix-links are totally ignored. Hence, the top portion in this layout is similar to SBFS. On the other hand, the bottom portion has almost total preference to suffix-links for localization. However, the clear cut boundary between the two portions cannot be identified at a particular level, as in the middle levels both the criteria are followed. The algorithm for *1Cr4Cd* layout generation is presented in Figure 3.4.

1Cr4Cd Layout Generation($\mathcal{T}$, B)
Input:
$\mathcal{T}$: Suffix-tree given by the user
B: Block size
Output:
Suffix-tree $\mathcal{T}_2$ which is similar to $\mathcal{T}$ in structure and semantics, but different in node arrangement

```
1.  markedForLinkPredecessor[.] ← false          /* "[.]" represents all elements in an array */
2.  followLink[.] ← false
3.  existLinkPredecessor[.] ← false
4.  for each node n in suffix-tree T do
5.      existLinkPredecessor[n.link] ← true
6.  end for
7.  for each node n in suffix-tree T do
8.      if internalChildrenCount(n) ≠ 4 OR incomingEdgeCharCount(n) ≠ 1 then do
9.          for each childNode of n do
10.             if existLinkPredecessor[childNode] then do
11.                 markedForLinkPredecessor[childNode] ← true
12.             end if
13.         end for
14.         followLink[n] ← true
15.     end if
16. end for
17. Call CommonRerrangementFunction(T, B, markedForLinkPredecessor[ ], followLink[ ])
```

Figure 3.4: Algorithm – 1Cr4Cd Layout Generation

3.4.2.3 bfs-hybrid Layout

In this layout, we try to get the maximum possible localization for the tree-edges (as in SBFS), but at the same time, we are not ignoring localization of suffix-links, where it is indeed necessary. This means that, unless and until it becomes necessary to put a node adjacent to its suffix-link predecessor node, we try to put it with its parent node.

The name given to the layout is because of its resemblance with SBFS layout to a certain extent, as it gives more weightage to tree-edges for localization. At the same time, localization of suffix-links is also done wherever it is found to be necessary. This is illustrated in the algorithm for bfs-hybrid layout generation (refer Figure 3.5).

In the algorithm bfs-hybrid layout generation, *cCountArray* stores the minimum number of child pointers from all the suffix-link predecessor nodes (which is done in the first for loop). In the second for loop, appropriate values are assigned to the two arrays –

bfs-hybrid Layout Generation($\mathcal{T}$**,** B**,** $linkPredChild$ **)**
Input:
$\mathcal{T}$: Suffix-tree given by the user
B: Block size
$linkPredChild$: Minimum number of children of a node to demand its link node to be placed with it
Output:
Suffix-tree $\mathcal{T}_2$ which is similar to $\mathcal{T}$ in structure and semantics, but different in node arrangement

```
1.  markedForLinkPredecessor[.] ← false; followLink[.] ← false
2.  cCountArray[.] ← |Σ|        /*Array for storing least child count among all suffix-predecessor*/
3.  for each node n in suffix-tree T do
4.      if cCountArray[n.link] > internalChildrenCount(n) then do
5.          cCountArray[n.link] ← internalChildrenCount(n)
6.      end if
7.  end for
8.  for each node n in suffix-tree T do
9.      if internalChildrenCount(n) < linkPredChild then do
10.         if internalChildrenCount(n) = cCountArray[n.link] then do
11.             markedForLinkPredecessor[n.link] ← true
12.             followLink[n] ← true
13.             cCountArray[n.link] ← -1
14.         end if
15.     end if
16. end for
17. Call CommonRerrangementFunction(T, B, markedForLinkPredecessor[ ], followLink[ ])
```

Figure 3.5: Algorithm – bfs-hybrid Layout Generation

$markedForLinkPredecessor$ and $followLink$, using the values of $cCountArray$.

The significance of $linkPredChild$ is that a node v is allowed to have $slNode(v)$ adjacent to it only when it has fewer children than $linkPredChild$. So, if v has children less than $linkPredChild$ and its number of children equals the number in $cCountArray$ (which is present at the offset of $slNode(v)$), then v is marked to follow its suffix-link and the $slNode(v)$ is marked to be left for its suffix-link predecessor node (i.e. v).

The assignment of -1 in line 13 of the algorithm avoids the case of multiple suffix-link predecessor nodes claiming their common suffix-link node. This case can arise when there are multiple suffix-link predecessor nodes (of a common node) with same number of child nodes (which are also less than $linkPredChild$). If we assign -1 after the visiting the first predecessor node then the remaining predecessor nodes cannot claim already considered suffix-link node.

3.4.2.4 OneLinkIn Layout

The algorithm of OneLinkIn layout generation (refer Figure 3.6) is similar to the algorithm of bfs-hybrid layout generation presented in Figure 3.5. In both the cases the first condition for a node v to has $slNode(v)$ in its vicinity is that the number of child nodes of v should be fewer than $linkPredChild$. But, the difference between the two algorithms occurs in the second condition.

OneLinkIn Layout Generation($\mathcal{T}$**,** B**,** $linkPredChild$ **)**
Input:
$\mathcal{T}$: Suffix-tree given by the user
B: Block size
$linkPredChild$: Minimum number of children of a node to demand its link node to be placed with it
Output:
Suffix-tree $\mathcal{T}_2$ which is similar to $\mathcal{T}$ in structure and semantics, but different in node arrangement

1. $markedForLinkPredecessor[.] \leftarrow$ false
2. $followLink[.] \leftarrow$ false
3. $linkInDegreeArray[.] \leftarrow 0$ /*Array for storing link in-degree for all nodes*/
4. for each node n in suffix-tree $\mathcal{T}$ do
5. $\quad linkInDegreeArray[n.link] \leftarrow linkInDegreeArray[n.link] + 1$
6. end for
7. for each node n in suffix-tree $\mathcal{T}$ do
8. $\quad$ if $internalChildrenCount(n) < linkPredChild$ then do
9. $\quad\quad$ if $linkInDegreeArray[n.link] = 1$ then do
10. $\quad\quad\quad markedForLinkPredecessor[n.link] \leftarrow$ true
11. $\quad\quad\quad followLink[n] \leftarrow$ true
12. $\quad\quad$ end if
13. $\quad$ end if
14. end for
15. Call CommonRerrangementFunction($\mathcal{T}$, B, $markedForLinkPredecessor[\,]$, $followLink[\,]$)

Figure 3.6: Algorithm – OneLinkIn Layout Generation

In bfs-hybrid algorithm, $slNode(v)$ is placed near to v only when the number of children of v is fewer among all predecessor nodes of $slNode(v)$. Whereas in OneLinkIn algorithm, v should be the *only predecessor* of $slNode(v)$, to get $slNode(v)$ near to it. This condition also explains the name chosen for this layout. So, the OneLinkIn layout has more restricted condition for the localization of suffix-links than the condition for the bfs-hybrid layout, as nodes with more than one incoming link are not considered for link localization. In this way, the chances of finding $slNode(v)$ near to v increase, when there

is indeed high possibility of visiting $slNode(v)$ from v.

In the algorithm of OneLinkIn layout generation, $linkInDegreeArray$ stores the suffix-link in-degree of all nodes of the suffix-tree. The first for loop is for calculating and assigning these in-degree values to the array and rest of the algorithm is similar to the algorithm of bfs-hybrid layout generation (except for the condition on line 9 which is already discussed before).

In support of this algorithm, we found that there are 37% of total suffix-tree nodes that have just one incoming suffix-link (refer Figure 3.7). So, if suffix-link predecessor nodes of such nodes satisfy the condition of $linkPredChild$, then this algorithm ensures that they are placed in the neighborhood of their suffix-link predecessors, instead of placing them in the neighborhood of their parent.

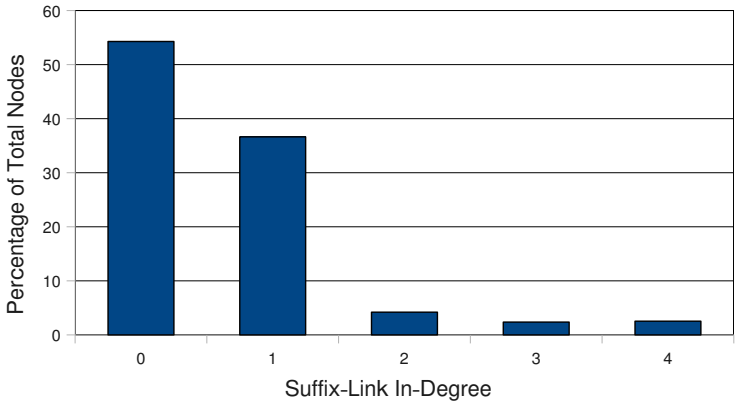


Figure 3.7: Percentage of Total Nodes vs. Suffix-Link In-Degree for the Suffix-Tree built over the Human Genome Sequence

3.4.2.5 Time-Complexity for Node-to-Block Assignment

All three layouts mentioned above consist of two for loops (iterated for each node of the suffix-tree) and a call to the $CommonRerrangementFunction$. The time-complexities of all these steps are linear in the number of suffix-tree nodes. So, the overall time-complexity of any node-to-block assignment algorithm is linear in the number of suffix-tree nodes. As we have seen earlier, the number of suffix-tree nodes are limited by the size of S, which makes the time-complexity of any node-to-block assignment algorithm as $O(n)$.

3.4.3 BFS Layouts: Block Sequence Determination

For the rearrangement of blocks (in order to get optimized sequence of blocks, as mentioned in Section 3.2.2), there can be two criteria for quantifying the relevance between any pair of blocks. These criteria consider either number of edges between pair of blocks or the overall probability of traversal between pair of blocks. These criteria are described as follows:

1. **Out-Degree of a block with respect to another block:** In this criterion, the consideration is given to *how many tree-edges and links are going from one block to another block*. But, the quantification of relevance between two blocks based on just the number of connecting tree-edges and links seems to be misguiding. Because, an edge from the initial node in a block have higher chances to get traversed, than the edge from the bottom part of that block. So, considering both the edges with equal weightage is not appropriate. This criteria can also fail under certain situations. For example, there can be many suffix-links from a block to another block, but they may not be used during search-traversal, if the *nodeTypes* of the majority of nodes are 4. So, we need the criterion which gives due weight to each edge connecting two blocks. This results in the formation of the basis for next criterion.

2. **Probability of Traversal from one block to another block:** This criterion considers the probability of going from one block b_s to another block b_d (i.e. $P(b_s, b_d)$), given that the search-traversal have already reached one of the independent nodes (which is defined shortly) in b_s. This probability of traversal can be measured as follows:

   (a) *Independent nodes* are the nodes whose parents or any of the suffix-link predecessors are not present in the current block b_s. We consider the probability of accessing these independent nodes as 1 (with respect to b_s). Because, accessing any independent node is not dependent on any other node present in b_s.

   (b) For an internal node v in block b_s, if $parent(v)$ is also in b_s, then the probability of visiting node v (given that the search-traversal has accessed $parent(v)$) can

be calculated as follows:

$$p_{v/parent(v)} = \frac{1}{|\Sigma|^{E(parent(v))-1}} * \frac{1}{|\Sigma|} \tag{3.1}$$

In order to understand formula 3.1, we need one basic information about how nodes in the suffix-tree are stored. In our implementation, an internal node v contains start and end positions of the substring represented by $e(v)$ (refer Section 4.1 for more details). This means access to v is necessary to know the characters on $e(v)$. So, a node v gets visited from $parent(v)$ only when $parent(v)$ gets accessed first and then all the characters on $e(parent(v))$ are matched with the characters from Q. The denominator in the first fraction (i.e. $|\Sigma|^{E(parent(v))-1}$) of formula 3.1 depicts this situation by indicating number of possible sequences of length $E(parent(v))-1$. The subtraction by 1 is due to the fact that the first character on $e(parent(v))$ is always matched before $parent(v)$ is visited. The denominator in the second fraction depicts equal chances for all the child nodes of $parent(v)$ to get visited.

From formula 3.1, it is obvious that the probability of visiting a node v from $parent(v)$ is limited by 0.25 (if $|\Sigma| = 4$). The reason for this limitation is that these probability calculations are done keeping in mind the arbitrariness of characters of query sequence Q (which are drawn from Σ) and not the number of children actually present in $parent(v)$.

In formula 3.1, we have made simplistic assumptions that all the characters are equally likely to occur in a sequence and their occurrences are independent of each other. But, this is not the case with real DNA sequences (as shown in [21]). Consider the table 3.1 for actual data regarding the percentage of occurrence of DNA characters within the human genome sequence.

So, if all the characters of S are given weights according to their occurrences (and the assumption of independent occurrences still holds), then formula 3.1 becomes

$$p_{v/parent(v)} = (f_A)^{Occ_A} * (f_C)^{Occ_C} * (f_G)^{Occ_G} * (f_T)^{Occ_T} * \frac{f_v}{f_{parent}} \tag{3.2}$$

Character	Percentage of Occurrence
A	29.51
C	20.53
G	20.52
T	29.44

Table 3.1: Percentage of occurrence of different characters in DNA sequence for Human Genome

where, for X = A, C, G or T, f_X indicates the fraction of X character in S, and Occ_X represents the number of times the characters X occurred in the substring represented by $e(parent(v))$. The terms f_{parent} and f_v represent the fraction of first character in the substring represented by $e(parent(v))$ and $e(v)$ respectively.

Till now, we have discussed traversal probability from a parent to its child node. But, the traversal probability from the suffix-link predecessor node also needs consideration. Unlike the visit from $parent(v)$, node v gets visited from $slpNode(v)$ only when there is a mismatch on the edge $e(child(slpNode(v)))$. This means that the grandchildren of $slpNode(v)$ should not be visited, if the link of $slpNode(v)$ has to be followed. So, the probability of visiting node v from $slpNode(v)$ is given as

$$p_{v/slpNode(v)} = 1 - \sum^{\forall g} p_{g/slpNode(v)} \tag{3.3}$$

where, g is a grandchildren of $slpNode(v)$. The probability $p_{g/slpNode(v)}$ can be calculated as the product of $p_{g/child(slpNode(v))}$ and $p_{child(slpNode(v))/slpNode(v)}$, where $child(slpNode(v))$ should be the $parent(g)$.

If v has $parent(v)$ and $slpNode(v)$ in the same block, then the probability to reach v is the summation of probabilities of reaching v from both the $parent(v)$ and $slpNode(v)$. When a tree-edge from v is going from one block to another block, the probability associated with that edge is the product of $p_{v/parent(v)}$ and $p_{parent(v)/indAncestor(parent(v))}$, where $indAncestor(parent(v))$ is the independent node which is also the ancestor of $parent(v)$. Similarly,the probability associated with a suffix-link to another block can calculated using above discussions.

Although these probabilities become smaller and smaller as we go deeper in a block, they play crucial role in assigning weights to edges which are at almost same level. For example, if a child of an independent node v is in another block then the $child(v)$ is surely going to have higher probability than some other node in the same block of v (which is neither an independent node nor its child node). But, when two grandchildren of v are located in two different blocks, the probabilities associated with them indicate that which one is going to carry more weightage from the viewpoint of visit by search traversal.

In order to calculate $P(b_s, b_d)$, we first find edges from b_s to b_d and probabilities associated with those edges. Then $P(b_s, b_d)$ is simply the summation of all such probabilities. When $P(b_d, b_s)$ is calculated, we just add it to $P(b_s, b_d)$ to obtain $P(b_s b_d)$[7] which is the probability of traversal in both direction. We have converted probabilities in both direction into one entity, as they jointly give indication of relevance between the two blocks. This $P(b_s b_d)$ is taken into consideration during determination of closeness of blocks b_s and b_d.

Now, in order to reduce the effect of assumption of independence that we have made earlier, we can consider the frequency of two or more characters taken together for calculation of traversal probabilities. But, we leave this fine-grained calculations as future work.

For the optimized rearrangement of blocks, we are considering the second option of block-to-block traversal probability. This block-to-block traversal probability helps us to determine the cost of a sequence of blocks, which is defined in the following definition:

Definition 4. ***Probabilistic Cost of a Block Sequence ($\mathcal{PC}_{BS}$):*** *In a sequence of block numbers, the summation over products of all block-to-block traversal probabilities and the inter-block distance between them is called as probabilistic cost of that block sequence.*

This measure of probabilistic cost is just a comparative and not an absolute one for the optimized block sequence. The lower value of probabilistic cost indicates better sequence

[7]Given that block number associated with b_s is less then block number associated with b_d.

of blocks. Because, probabilities associated with the blocks remain same, but the inter-block distances associated with them get changed after rearrangement of blocks. We want to bring blocks with high probabilities as close as possible for maximum reduction in probabilistic cost.

The algorithm for optimized block sequence determination is presented in Figure 3.8. In the algorithm, we always try to extend the optimized block sequence in such a way that increase in the probabilistic cost is least. The sequence determination starts with the pair of blocks with highest probability and then the pairs with next highest probabilities are considered. So, in this way, the approach of the algorithm is greedy, as it starts with local optimized sequence and continue to find overall optimized sequence.

3.4.3.1 Time-complexity for Block Sequence Determination

In the algorithm of Figure 3.8, we are considering calculation of block-to-block traversal probability for a pair of blocks as a basic operation, because the number of calculations to be done for it are restricted by the number of nodes in a block, which is a constant for the fixed block size.

Now, the first *for* loop (line 2) is iterated for each block in the suffix-tree. The nested *for* loop (line 3) is iterated for the number of connected blocks to the block of main *for* loop. But, the number of connected blocks to a block are limited by the size of the block and the cardinality of the alphabet set. For example, in our case, for block size of 64 kB, the maximum number of blocks that can be connected to a block (i.e. n_{mcb}) is 2048 * 5, where 2048 is maximum number of nodes accommodated in a block and 5 is the maximum out-degree of a node (including suffix-link pointer), which is $|\Sigma| + 1$. This is with the assumption that all the nodes in a block are independent nodes and all the pointers point to a different block. So, if we consider block size and cardinality of alphabet set as fixed (which is usually the case), then the time-complexity of the first *for* loop is linear in the number of blocks n_b (although constant factor is high).

For sorting of the list $\mathcal{L}$ (line 13), the worst-case entries in the list $\mathcal{L}$ could have been quadratic in n_b. But, we have seen that maximum connection from a block to other blocks

Optimized Block Sequence Determination ($\mathcal{T}$, n_b, B)
Input:
$\mathcal{T}$: Suffix-tree given by the user
n_b: Total number of blocks
B: Block size
Output:
Suffix-tree $\mathcal{T}_2$ which is similar to $\mathcal{T}$ in structure and semantics, but differ in arrangement of blocks

1. Create an empty list $\mathcal{L}$ for the insertion of triplets $\{b_s, b_d, P_{b_s b_d}\}$, where b_s and b_d refers to block numbers of two storage blocks of $\mathcal{T}$ such that $b_s < b_d$
2. for each storage block b_1 in $\mathcal{T}$ do
3. for each storage block b_2 connected to b_1 do
4. if $b_1 < b_2$ then do
5. Calculate $P_{b_1 b_2}$
6. $\mathcal{L}$.insert($\{b_1, b_2, P_{b_1 b_2}\}$)
7. else if there is no edge from b_2 to b_1 then do
8. Calculate $P_{b_2 b_1}$
9. $\mathcal{L}$.insert($\{b_2, b_1, P_{b_2 b_1}\}$)
10. end if
11. end for
12. end for
13. Sort elements of $\mathcal{L}$ according to block-to-block traversal probabilities
14. Create an empty master-list $\mathcal{L}_m$ for the storage of sub-lists of optimized block sequences
15. Create an empty sub-list α_{empty} in $\mathcal{L}_m$
16. for each triplet $\{b_1, b_2, P_{b_1 b_2}\}$ in list $\mathcal{L}$ do
17. if b_1 and $b_2 \notin$ any of the sub-list of $\mathcal{L}_m$ then do
18. Create new sub-list α_{new} such that $b_1, b_2 \in \alpha_{new}$
19. $\mathcal{L}_m.insert(\alpha_{new})$
20. else if $b_1 \in \mathcal{L}_m.\alpha_{sub1}$ and $b_2 \in \mathcal{L}_m.\alpha_{sub2}$ and $\mathcal{L}_m.\alpha_{sub1} \neq \mathcal{L}_m.\alpha_{sub2}$ then do
21. if $\mathcal{PC}_{BS}(\mathcal{L}_m.\alpha_{sub1}||\mathcal{L}_m.\alpha_{sub2}) < \mathcal{PC}_{BS}(\mathcal{L}_m.\alpha_{sub2}||\mathcal{L}_m.\alpha_{sub1})$ then do
/* || symbol indicates concatenation operation */
22. $\alpha_{result} = \mathcal{L}_m.\alpha_{sub1}||\mathcal{L}_m.\alpha_{sub2}$
23. else
24. $\alpha_{result} = \mathcal{L}_m.\alpha_{sub2}||\mathcal{L}_m.\alpha_{sub1}$
25. end if
26. $\mathcal{L}_m.remove(\alpha_{sub1})$; $\mathcal{L}_m.remove(\alpha_{sub2})$; $\mathcal{L}_m.insert(\alpha_{result})$
27. else if either b_1 or $b_2 \in$ any of the sub-list of $\mathcal{L}_m$ then do
28. if $b_1 \in \mathcal{L}_m.\alpha_{sub}$ then do
29. if $\mathcal{PC}_{BS}(\mathcal{L}_m.\alpha_{sub}||b_2) < \mathcal{PC}_{BS}(b_2||\mathcal{L}_m.\alpha_{sub})$ then $\mathcal{L}_m.\alpha_{sub} = \mathcal{L}_m.\alpha_{sub}||b_2$
30. else $\mathcal{L}_m.\alpha_{sub} = b_2||\mathcal{L}_m.\alpha_{sub}$
31. else if $b_2 \in \mathcal{L}_m.\alpha_{sub}$ then do
32. if $\mathcal{PC}_{BS}(\mathcal{L}_m.\alpha_{sub}||b_1) < \mathcal{PC}_{BS}(b_1||\mathcal{L}_m.\alpha_{sub})$ then $\mathcal{L}_m.\alpha_{sub} = \mathcal{L}_m.\alpha_{sub}||b_1$
33. else $\mathcal{L}_m.\alpha_{sub} = b_1||\mathcal{L}_m.\alpha_{sub}$
34. end if
35. end if
36. if $\mathcal{L}_m$ has only one sub-list α_{final} and $|\alpha_{final}| = n_b$ then exit for loop
37. end for
38. Rearrange the blocks of $\mathcal{T}$ according to the sequence in the α_{final}

Figure 3.8: Algorithm – Optimized Block Sequence Determination

are restricted by n_{mcb} (which is a constant number). So, the number of entries in $\mathcal{L}$ is linear in n_b. Hence, the time-complexity for sorting operation is $O(n_b\ log\ n_b)$.

In the third *for* loop (line 16), we are accessing each entry in the list $\mathcal{L}$, till all the blocks get consideration in the optimized block sequence. Now, in order to determine the presence of blocks (obtained from $\mathcal{L}$) in sub-lists of $\mathcal{L}_m$, the comparison operations required in the worst-case could be as many as total number of blocks. Also, during the calculation of $\mathcal{PC}_{BS}$ values, each pair of blocks from $\mathcal{L}$ is considered only once. But, it requires access to all blocks within a sub-list, to find all the connected blocks for a given block, and a sub-list can contain $O(n_b)$ blocks in the worst case. Hence, the time-complexity of this *for* loop is cubic in n_b. The remaining operations, like concatenation of two sub-lists or the creation of a sub-list, can be done in $O(1)$ time.

So, the overall time-complexity for the block sequence determination is $O({n_b}^3)$. Now, the total number of blocks n_b is linearly dependent on the total number of nodes. The value of n_b can be obtained by dividing the size of total nodes by the block size. But, as we have seen earlier, the total number of nodes is also limited by the size of S. Hence, the time-complexity for optimized block sequence determination becomes $O(n^3)$, where $n = |S|$.

In our work, we have used the Trellis algorithm for suffix-tree construction, whose time-complexity for construction is quadratic in the size of $|S|$, i.e. $O(n^2)$. So, the time-complexity for the generation of the search-optimized layout with node-to-block assignment becomes $O(n^2)$. Whereas the overall time-complexity for the generation of the search-optimized layout with optimized block sequence remains unchanged, which is $O(n^3)$. But, for our search-optimized layout generation, it is not necessary to use only Trellis algorithm. We could have used traditional linear time algorithm, such as Ukkonen's algorithm, for the construction of the suffix-tree (if practical construction time would not have been a major concern). This would have changed the overall time-complexity for the generation of the search-optimized layout with node-to-block assignment to $O(n)$. But, it would not affect the overall time-complexity for the generation of the search-optimized layout with optimized block sequence, which is cubic in n.

However, like many earlier disk-based suffix-tree construction approaches, our focus is on the practical aspect (of the search-time improvement) and not on the theoretical efficient algorithms (for various layout generation). So, even with this cubic complexity algorithm for the optimized block sequence determination, we have obtained substantial improvement in actual search-time (refer Chapter 5 for details), and the search algorithm for searching on these search-optimized layouts is still linear in the size of query sequence.

3.5 Implementation Issues

1. **Post-Construction Optimization:** Various search-optimized layouts proposed by us are implemented by rearranging nodes and blocks from already constructed suffix-tree. So, our approach is post-construction and we need the suffix-tree at least in construction layout to perform the process of layout reorganization. Our dependence (on the suffix-tree instead of the sequence on which it is built) is due to the requirement of advance knowledge of child nodes as well as suffix-link nodes, during rearrangement process.

2. **Dependency of Rearrangement Levels:** The two levels of the layout reorganization process mentioned in Section 3.2 are dependent on each other. We have to perform the block level rearrangement only after the node level rearrangement is done. This is due to the fact that the sequence of blocks generated by the block level rearrangement depends on how the relevant nodes are grouped together inside different blocks. If we do the reverse, then the node level rearrangement will nullify the optimization obtained by the block level rearrangement.

Chapter 4

Physical Structure Improvements of Internal Nodes

In this chapter, we propose a few improvements for the physical structures of suffix-tree internal nodes. These improvements have taken advantage of unused space and redundancy present in suffix-tree nodes by making data-structural changes in them. In Chapter 3, search-optimized layouts for suffix-trees have been constructed by just rearrangement of nodes and blocks. Whereas here, we are building search-optimized layouts by considering structural changes to suffix-tree nodes and so these changes are mentioned separately from Chapter 3. A point to note here is that although we are changing the data-structure used to implement suffix-tree nodes, we are not tampering with the logical structure of the suffix-tree itself.

4.1 Understanding Physical Structures of Suffix-Tree Nodes

In order to understand the improvements given in this chapter, the basic understanding of structures of suffix-tree nodes is desirable. In our implementation of suffix-tree, we are using suffix-tree node structures of Trellis [21]. In this implementation, suffix-tree nodes have array representation instead of linked-list representation. The array representation

of suffix-tree nodes is not only superior for the construction of the suffix-tree [4], but it also performs better during searching over the suffix-tree. This is because, parent to child node traversal always require maximum one access in array representation, which is not the case in linked-list representation.

As we have seen in Chapter 1, the suffix-tree has two kind of nodes - *internal nodes* and *leaf nodes*. Structures of these nodes for the suffix-tree built over a DNA sequence (where $|\Sigma| = 4$) are shown in Figures 4.1. All node sizes are fixed in our implementation. Due to the difference of what they represent, the two kind of nodes have different structures and so different files for their storage.

Start	End	$-Leaf Pointer	Child Pointer 1	Child Pointer 2	Child Pointer 3	Child Pointer 4	Suffix-Link Pointer
4 Bytes	4 Bytes	4 Bytes	4 Bytes	4 Bytes	4 Bytes	4 Bytes	4 Bytes

(a) Internal Node Structure

Suffix-ID	Actual Leaf Start
4 Bytes	4 Bytes

(b) Leaf Node Structure

Figure 4.1: Details of Suffix-Tree Node Structures

In Figure 4.1(a), the structure of an internal node (say v_i) is shown. In this structure, *Start* and *End* indicate the start and end positions of the substring of the indexed sequence S (i.e. $S[Start \ldots End]$), which is represented by the edge $e(v_i)$. Every *Child Pointer*, either points to an internal node, to a leaf node or to nothing (i.e. a *NULL pointer*). The *$-Leaf Pointer* (if it is not NULL) points to a child leaf node (say $v_{l_\$}$) only. The difference between *$-Leaf Pointer* and other *Child Pointers* pointing to leaf nodes is that the edge $e(v_{l_\$})$ represents only '$' as its substring. The number associated with each *Child Pointer* is indicative of the first character of the substring represented by respective $e(child(v_i))$. Finally, the *Suffix-Link Pointer* points to $slNode(v_i)$. So, in our case, the size of an internal node is 32 bytes.

The structure of a suffix-tree leaf node (say v_l) is given in Figure 4.1(b). It contains

two fields, which represent positions into S. The value of *Suffix-ID* (or equivalently i in S_i) indicates the position in S, from where a *suffix of* S starts and this suffix is fully represented by the substring $\sigma(v_l)$. The value of *Actual Leaf Start* indicates the position in S, which represents a *suffix of* S_i. This suffix of S_i is fully represented by the edge $e(v_l)$. To illustrate this with an example refer to Figure 1.2, where the leaf node labeled with number 3 has value of *Suffix-ID* as 3, whereas value of *Actual Leaf Start* as 5. The size of a leaf node is 8 bytes.

As genome-level sequences are less than 4 Gbp, the fields of size 4 bytes in suffix-tree nodes, which refer to various positions in the indexed sequence S (i.e. *Start*, *End* fields of an internal node and *Suffix-ID*, *Actual Leaf Start* fields of a leaf node), can indicate any position in S. Each *Child Pointer* points to its $child(v_i)$ (whether $child(v_i)$ is an internal node or a leaf node) by referring to the offset of $child(v_i)$ within its file. As we have appended a unique character at the end of S (refer Section 1.3.1 for details), the number of leaf nodes are guaranteed to be same as the number of characters present in S (i.e. less than 4 billion). Therefore, the size of *Child Pointer* (4 bytes) pointing to a leaf node is sufficient to point to any leaf node within the leaf node file. This pointer size is also sufficient to point to any internal node, as the number of internal nodes are around 0.7 times the number of leaf nodes [21]. Although the space in these pointer fields are sufficient to point to any node, it is not sufficient for the differentiation of node pointer types. That is, by referring just the value of a pointer, one cannot say whether the pointer is an internal node pointer or a leaf node pointer. For this purpose, a separate bitmap is required to distinguish between leaf node and internal node pointers. The implementation detail for this bitmap is given in Section 4.2.1.

4.2 Structural Improvements of Suffix-Tree Nodes

In this section, we suggest a few improvements for the structure of suffix-tree nodes. These improvements are based on a few observations and their subsequent analysis, which expose the inability of the structure of the suffix-tree and its nodes to fully utilize the space allocated to them. The improvements mentioned in this section attempt to utilize

this unused space present in suffix-tree nodes by filling it with useful information that can be utilized during the search traversal. With these improvements, the search traversal gets most of the required information from the internal node itself, instead of fetching it from the other parts of the same file or from a different file (such as the leaf node file). So, these improvements result in reduced disk accesses as well as reduced main-memory accesses. In addition to this, space reduction on disk is also achieved. These improvements are independent of suffix-tree layouts and can be applied to any suffix-tree layout discussed in Chapter 3.

4.2.1 Node Pointer Differentiation Bitmap

In Section 4.1, we mentioned the requirement of a bitmap to distinguish between an internal node pointer and a leaf node pointer. For this purpose, the bitmap requires 1-bit per each child pointer , i.e. $|\Sigma|$ bits per internal node. The suffix-tree on a DNA sequence requires 4-bits per node. Now, there are two options for storage of this bitmap - either it can be stored in a separate file or embedded in the related node itself. While the first option is straightforward, implementation of second option requires analysis of space within the suffix-tree nodes, which should be available for use in every internal node.

4.2.1.1 $-Leaf Pointer Usage Analysis

As we have seen in Section 1.3.1, a special character which is not present in the indexed sequence S (usually '$') is appended to the end of S. Although this unique character is appeared only once in S\$, the fan-out of internal-nodes has to be increased by one (we have indicated this pointer as *$-Leaf Pointer* in Figure 4.1(a)), as the number of different characters in the sequence S\$ is incremented by one.

As this character does not occur anywhere within S, there is not a single internal node v_i whose $e(v_i)$ contains this character. So, the tree-edges that contain this character must directly lead to a leaf node only, where this is always the last character in the substring represented by that tree-edge. If any tree-edge starts with this character, then that is the only character represented by that tree-edge. The edge in such case is represented by

$-Leaf Pointer and occurs when a proper suffix of S ends within the internal node of the suffix-tree. But, the quantitative measurement in Table 4.1 indicates that in real genome sequences, such suffixes occur less frequently, when compared with the total number of suffixes of S.

Indexed sequence	Total nodes in suffix-tree	Number of nodes having meaningful $-leaf pointer	Percentage of space wastage by $-leaf pointer
Chromosome - 22	24005631	10	99.9999
Chromosome - 16	54222324	16	99.9999
Chromosome - 01	154429037	14	99.9999
Human Genome	2641873380	786	99.9999

Table 4.1: $-Leaf Pointer Usage Measurement

Due to the huge unused space inside *$-Leaf Pointer* of internal nodes, it seems that the addition of unique terminal character is wasting space unnecessarily. But, from the perspective of correctness, addition of unique terminal character is necessary. Otherwise, the suffix which is prefix of another suffix will not get detected as a proper suffix of S. Detailed discussion about importance of *$-Leaf Pointer* is given Section 1.3.1.

4.2.1.2 Implementation of Bitmap

In order to utilize the unused space of *$-Leaf Pointer* across all internal nodes, the leaf nodes pointed by *$-Leaf Pointers* have to be separated from the main leaf node file to a different file (which we call as *"$-leaf file"*). As there are fewer number of nodes in the $-leaf file, the *$-Leaf Pointers* pointing to them do not require all bits of its 4-byte pointer. For example, in the suffix-tree on human genome sequence, 10-bits are sufficient for *$-Leaf Pointer*. The remaining space can be utilized for implementation of bitmap. However, in this implementation of bitmap we took only 4-bits in the most-significant part of a *$-Leaf Pointer* field. The remaining available bits can be used for further usage. For example, as shown in Section 4.2.3, this bitmap is further extended to 1 byte in order to accommodate improvement of embedded characters.

In our implementation, if a bit is 1 in the bitmap then the corresponding *Child Pointer* points to an internal node. On the other hand, if the bit is 0 then the corresponding *Child*

Pointer (if not NULL) points to a leaf node.

This bitmap is implemented in the same way across all the layouts that we have discussed in Chapter 3. This implementation changes the structure of internal nodes only. The structure of leaf nodes remains unaltered. The new structure for internal nodes is shown in Figure 4.2. This implementation saves space for storage of bitmap by embedding it inside internal nodes of the suffix-tree.

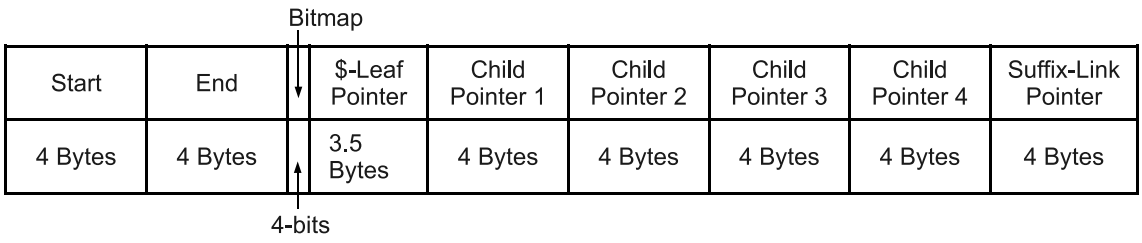


Figure 4.2: Details of Internal Node Structure with Bitmap

4.2.2 Embedded Leaf Nodes

For a leaf node v_l, if we store only the *Actual Leaf Start* instead of both - *Suffix-ID* and *Actual Leaf Start*, then from the explanation of Figure 4.1(b), we know that there is a problem in finding the actual suffix corresponding to a leaf node v_l, whose position in the sequence S is indicated by *Suffix-ID* field. But, the value of *Suffix-ID* can be calculated dynamically at the time of search by keeping track of how many characters are present on the path from the root node to $parent(v_l)$. Because, those characters precede the substring represented by $e(v_l)$ in the sequence S and when we subtract that number of characters from the start position (which is indicated by *Actual Leaf Start*) of the edge $e(v_l)$, we get the position of required suffix in S as indicated by the *Suffix-ID*.

Now, the size of *Actual Leaf Start* field of v_l is 4-bytes and the size of pointer to v_l is also 4 bytes. So, with our decision to store just *Actual Leaf Start*, we can replace the pointer to v_l in $parent(v_l)$ with its *Actual Leaf Start* value. This change altogether removes the requirement of leaf nodes and hence a separate leaf node file. In this way, all the leaves of a suffix-tree are absorbed by its corresponding parent internal nodes without increasing the size of the internal node file.

The removal of separate leaf node file not just reduced the space requirement of suffix-tree by *25%*, but it also decreases the amount of disk accesses as well as main-memory accesses required to fetch the leaf data during the search traversal. Because, part of the required leaf data (i.e. *Actual Leaf Start*) is already present in the internal node that we are accessing and remaining information related to *Suffix-ID* can be obtained from the search completed till that point (as mentioned earlier). Also, the size of the internal node is *32 bytes*, which will easily get accommodated within a processor cache-block whose minimum size is usually *64 bytes*, so that the chances of processor cache-miss on a leaf node access also diminishes. And thus, this improvement also increases the processor cache-consciousness of the suffix-tree layout.

The changes in the structure of an internal node for this improvement is illustrated in Figure 4.3. Note that instead of a leaf child pointer, *Actual Leaf Start* of respective leaf child node is stored.

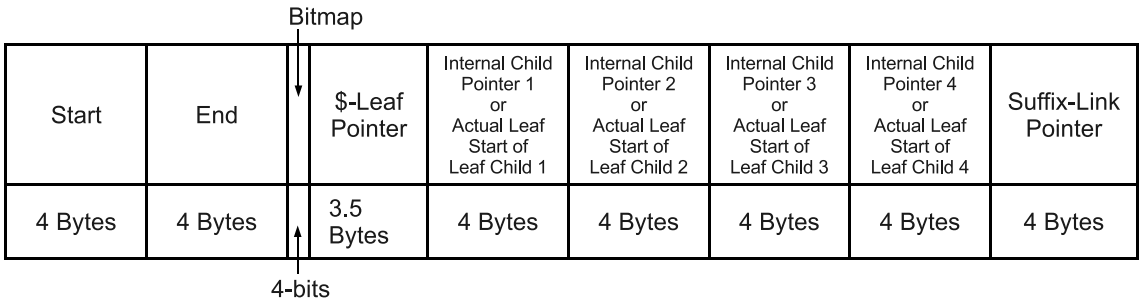


Figure 4.3: Details of Internal Node Structure with Embedded Leaf Nodes

4.2.3 Embedded Characters from the Indexed Sequence

In this improvement, we embed characters from the sequence S into internal nodes of the suffix-tree. The need for embedding of characters is due to two requirements:

1. During search traversal, access to S is required, when characters from Q are compared with the characters of substring represented by $e(v)$ of an internal node v.

2. The information about substring represented by $e(v)$ is placed in node v only. So, the characters of edge $e(child(v))$ cannot be known from v (except the first charac-

ter), without accessing $child(v)$. This may lead to unnecessary traversal to a child node, if a mismatch occurs during comparison of characters of Q with the substring represented by $e(child(v))$.

These requirements also lead us to two choices for embedding of characters (with respect to an internal node v). These choices describing which characters should be embedded are given below:

1. Characters from the substring represented by $e(v)$.
2. Characters from the substring represented by $e(child(v))$.

When the first choice is selected, then there are less chances that access to S is required for the matching of the substring represented by $e(v)$ with Q. So, even if we assume that whole sequence S is accommodated in main-memory, then also it improves search performance by increasing processor cache-hits. Because the required information for matching is already in the processor cache which is brought along with v.

The embedding strategy mentioned in the second choice, makes the search more I/O efficient as well as cache-conscious. Because, decision on whether to traverse $child(v)$[1] or $slNode(v)$ is taken mostly from the data present in v itself. Otherwise, this decision can be taken only after accessing $child(v)$, which may resides in the main-memory or on the disk.

So, the obvious choice is to go for the second option, mainly because of the possibility of reducing heavy cost of disk accesses during the search traversal. Also, the analysis done in Section 4.2.3.3 justifies our choice by showing that there are substantial number of cases, where implementation of second choice can help. A point to note here is that, we are not considering the case of embedding characters represented by $e(slNode(v))$. There are two reasons for ignoring it:

1. For an internal node v, the substring $\sigma(v)$ has one more character than the substring $\sigma(slNode(v))$. This difference of one character is at the start of substrings and not at

[1]Assuming that the required $child(v)$ for the search traversal is present.

the end of substrings. So, the end parts of $\sigma(v)$ and $\sigma(slNode(v))$ are same (unless one of v or $slNode(v)$ is the root node). That means, the characters represented by $e(slNode(v))$ are already matched with Q, as all characters on $e(v)$ are also matched (otherwise, search traversal would not have visited $slNode(v)$ from v).

2. If the search algorithm has to traverse $slNode(v)$ from v, then it cannot be avoided. This is in contrast to the case of a mismatch which occurs on the multi-character edge $e(child(v))$. In such a situation, if we knew that a mismatch is going to happen in advance, then we can avoid visiting $child(v)$ node.

Now, we have to find unused space in the suffix-tree internal nodes apart from unused space of *$-Leaf Pointer*. We have to also analyze this unused space for embedding of characters, which are mentioned in following two subsections.

4.2.3.1 Analysis of NULL Pointers

The NULL pointers present in the suffix-tree indirectly reduce the locality of nodes by wasting the space allocated to the suffix-tree nodes (if the node size is considered as fixed). Table 4.2 shows the average NULL pointers per internal node within the suffix-tree for various DNA sequences.

Indexed DNA sequence	Total nodes in suffix-tree	Total NULL pointers	Average NULL pointers per internal node
Chromosome - 22	24005631	36958263	1.539
Chromosome - 16	54222324	83676796	1.543
Chromosome - 01	154429037	237083043	1.535
Human Genome	2641873380	4306914607	1.630

Table 4.2: NULL Pointers present in Suffix-Tree for different DNA Sequences

In the suffix-tree for a DNA sequence, there are maximum 4 child pointers per internal node[2]. So, the maximum number of allowed NULL pointers per internal node is 2. Because, if there exist only one meaningful child pointer of node v then the child node

[2]We are not including *$-Leaf Pointer*, because in majority cases it is null and its unused space is already considered for a different case.

corresponding to that pointer gets pulled up and merged with v. So, the amount of NULL pointers per node is more than 80% of the permitted value (in case of suffix-tree built on the human genome sequence). Our aim is to utilize this unused space to embed the characters from S.

4.2.3.2 Analysis of Substring Length of Incoming Tree-Edge

For any internal node v, there is exactly one incoming tree-edge $e(v)$ (except the root node). This $e(v)$ is representative of some substring from S, which is indicated by *Start* and *End* fields of v (refer Section 4.1). The internal node classification (or alternatively incoming tree-edge classification, due to one-to-one correspondence of incoming tree-edges with internal nodes) according to the length of substring represented by $e(v)$ in the suffix-tree built on the human genome sequence is shown in Figure 4.4.

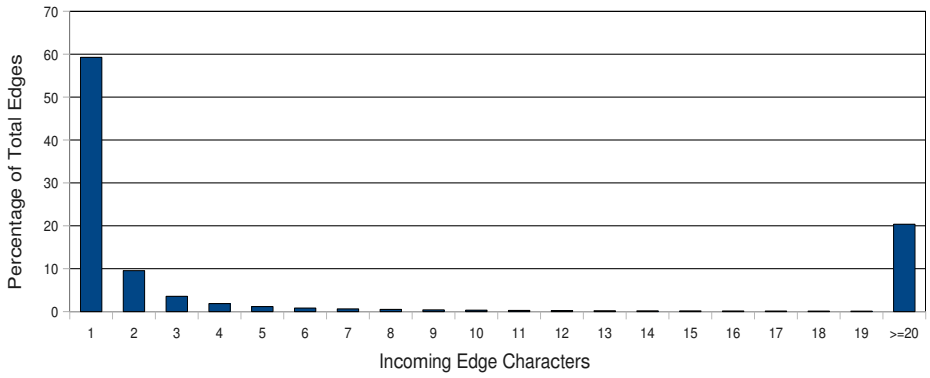


Figure 4.4: Incoming Tree-Edge Classification according to the Length of the Substring represented by it for the Suffix-Tree built over the Human Genome Sequence

The graph in Figure 4.4 indicates that there exist nearly 60% of total internal nodes in the suffix-tree built on human genome sequence, whose incoming tree-edges represent just one character. Now, in the structure of an internal node v, the length of substring represented by $e(v)$ is equal to $End - Start + 1$. So, if the length of $e(v)$ is just one character, then the value of *End* field becomes redundant (as the value of *End* field in such cases becomes equal to the value of *Start* field). Hence, the space allocated to *End* field remains unused in 60% of total internal nodes.

In this analysis, we have considered static structural property of the suffix-tree, i.e.

classification of incoming tree-edge according to the length of the substring represented by it. In the following section, we observe the effect of multi-character substring of incoming tree-edges on searching. So, in that analysis, our focus is on dynamic behavior of the suffix-tree during the search traversal.

4.2.3.3 Multi-Character Child Edge Traversal Analysis

As we have already discussed in the start of Section 4.2.3 that in some cases search traversal has to visit $slNode(v)$ even after visiting $child(v)$, due to mismatch on $e(child(v))$. These cases are quantitatively shown in Table 4.3, which are taken from a sample search. The search consists of different kinds of search strings which are partially drawn from the human genome sequence. The characters in search strings vary from 500 to 10000.

Percentage of search string taken from the indexed sequence	Total tree-edges traversed during the search	Number of tree-edges traversed unnecessarily	Percentage of unnecessary tree-edge traversal
0	1198194	177553	14.82
25	1106957	165680	14.97
50	982943	143678	14.61
75	835201	102249	12.24
100	616771	0	0

Table 4.3: Unnecessary Child Node Traversal

From Table 4.3, we can observe that when at most half part of a search string Q is drawn from the indexed sequence S, then the search traversal accesses nearly 15% of tree-edges unnecessarily. Although this constitutes fewer number of edge traversals (when compared with total tree-edge and suffix-link traversals), the disk accesses introduced by it not only take considerable time during the search traversal, but also increases randomness in the overall disk accesses that are performed during the search traversal. So, if we are able to avoid this unnecessary child node traversal, then the performance improvement can be achieved. But, not all unnecessary child node traversals can be avoided. It depends on how many characters can be embedded in the current node and how many characters get matched during the search. Also, if the child node is physically nearer to the parent node on the disk, then the performance improvement due to character embedding is not

noticeable, as even without this improvement there is no disk access involved for visiting the child node. Because, read-ahead of data done by operating systems can also bring the nearby nodes into the main-memory.

4.2.3.4 Implementation of Embedded Characters of Child Edge

For implementation of this improvement, there are two places within an internal node v, where characters from $e(child(v))$ can be embedded. The appropriateness of these two places are discussed below:

1. ***End* field of an Internal Node:** As analyzed in Section 4.2.3.2, there is lot of unused space in the *End* field of an internal node. But, we want to further analyze the effect of utilization of this space for embedding of characters.

   A detailed analysis of incoming edge substring length (Section 4.2.3.2), indicates that almost 82% of internal nodes have less then 64 characters in the substring represented by their incoming edge. So, in that case, if we decide to represent only the length of the substring and not the end position (as in original node structure of Figure 4.1(a)) in the *End* field, then we require only one byte (6-bits to indicate substring length and 2-bits to indicate overflow child pointer location). That leaves 3 bytes for character embedding in every internal node. But, for the remaining internal nodes with length ≥ 64, there must be some other vacant space (like NULL pointers) in the internal node itself, where we can extend the value of the substring length.

   Figure 4.5 shows that there exists a node $child(v)$ whose $e(child(v))$ edge's substring length is greater than or equal to 64 and also node v has all the internal child nodes present. So, in such cases, we may not find the void space within v to accommodate the extended value of the difference between its *End* and *Start* fields. The same situation will arise, even if the represented length inside *End* field is increased. And hence, we are not considering this option of embedding characters inside *End* field, even though the field has lot of unused space in it. Because, if we choose this option,

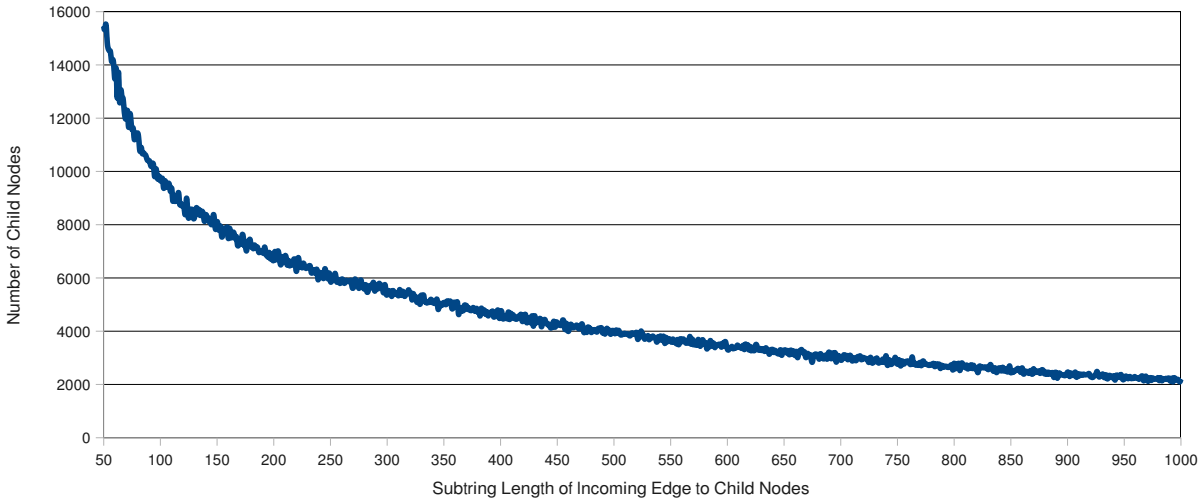


Figure 4.5: Number of Child Nodes with Substring Length of Incoming Edge between 50 to 1000 and whose Parent Nodes have all the Internal Child Nodes present

then suffix-tree will lose correctness (as the value of incoming edge's length cannot be represented correctly in all cases), which cannot be allowed.

2. **NULL Pointers:** As there are **1.6** NULL pointers per internal node of the suffix-tree built on human genome sequence (refer Section 4.2.3.1), we can utilize this unused space for embedding of characters from a child edge.

   Unlike the option of *End* field, embedding of characters in this option is done without affecting the correctness of the suffix-tree. Here, only unused space (in the form of NULL pointers) in an internal node v is considered for embedding. If NULL pointers are not present in v then we do not embed the characters from S.

   The number of NULL pointers in v (in the suffix-tree built over a DNA sequence) can vary from 0 to 2. Now, we briefly see how these 3 cases for NULL pointers are used to embed characters of $e(child(v))$.

   - **0 NULL Pointer in v:** In this case, there is no NULL pointer in v. So, no embedding of characters is done in v and it is left with its original content.
   - **1 NULL Pointer in v:** If there is just one NULL pointer in v, then we look for an internal $child(v)$ with maximum number of characters on edge $e(child(v))$.

If we find one, then we embed its characters in this NULL pointer.

- **2 NULL Pointers in v:** In this case, if there is only one internal $child(v)$ then the task of characters embedding is straightforward. Because, if $child(v)$ is having multiple characters on $e(child(v))$, then those characters are embedded in both the NULL pointers. But, when there are two internal $child(v)$ nodes, we have to choose either one of them or both of them for characters embedding. If only one of the two internal $child(v)$ nodes has multiple characters on its incoming edge, then that $child(v)$ is chosen for embedding of characters. Otherwise, both $child(v)$ nodes are chosen in order of occurrence of their pointers in v.

For the embedding of characters in NULL pointers, we need to extend the already present bitmap used for differentiation between internal node and leaf node pointer (see Section 4.1), to indicate the presence of embedded characters. So, the current form of bitmap (1-bit per pointer) is changed to 2-bits per pointer, in order to accommodate the third alternative of embedded characters. The mapping of these 2-bits in this implementation is as follows:

- 00 - NULL pointer
- 01 - Embedded characters
- 10 - Pointer to a leaf node
- 11 - Pointer to an internal node

The modified structure of an internal node is shown in following figure.

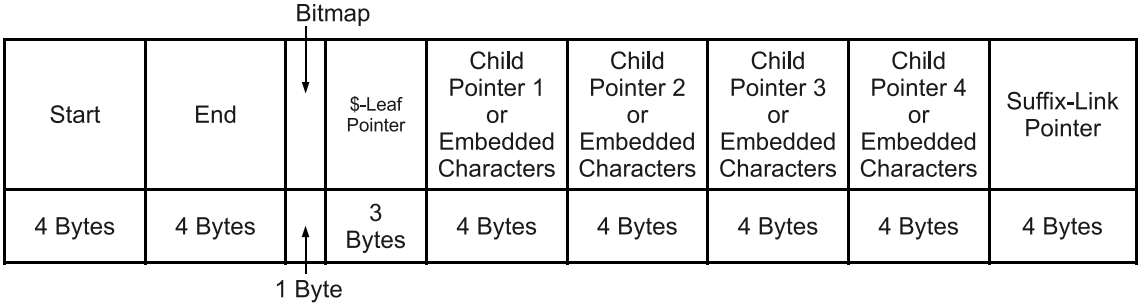


Figure 4.6: Details of Internal Node Structure with Embedded Child Characters

Now, when we store embedded characters in the field of *Child Pointer*, the bits of this field are used as shown in Figure 4.7. The first field (of 2 bits) shows for which child pointer the characters are embedded. The next field (of 4 bits) shows how many characters are embedded in this pointer. The remaining 26-bits are used to store embedded characters. With 2 bits per character (as we have $|\Sigma| = 4$), maximum of 13 characters can be stored.

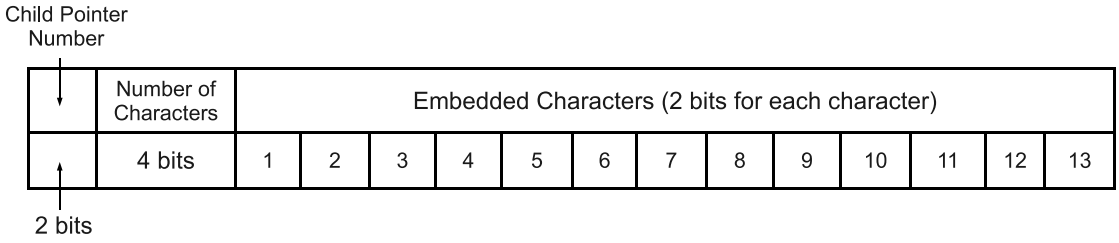


Figure 4.7: Details of NULL Child Pointer (of 4 Bytes) when Embedded with Child-Edge Characters

Chapter 5

Experimental Evaluation

Initially, in this chapter, we see the details about experimental setup which is used to carry out various experiments. Then, we provide the construction and layout modification timings for all layouts which are used for performance measurement. After that we go through the search results obtained from the layout reorganizations proposed by us (as suggested in Chapters 3) and compare them with the results obtained from the existing state-of-the-art layouts. Along with these results, we also analyze various supporting statistics which present justification for the result obtained. Towards the end of this chapter, we also analyze the significant reduction in search-time due to the improvements introduced in Chapter 4.

5.1 Experimental Setup

In the evaluation of experimental results, we have used two machines at different stages of experimentation. The details of these machines are given as follows:

1. **Machine R:** IBM P690 Reggata machine with AIX 5.2L operating system, powered with 32 IBM Power-4 processors operating at 1.9 Ghz, having 256 GB RAM and 1000 GB disk-space.

2. **Machine S:** Sun Ultra 24 machine with Ubuntu Linux 10.04 operating system, powered with Intel Core2 Extreme processor operating at 2 GHz, having 8 GB

RAM. In this machine, there are 4 Hitachi SATA hard-disks, each with rotational speed of 7200 rpm and 750 GB capacity.

Machine R is a high-end system with large main-memory. In our experiments, this machine is only used for the construction of various disk-based suffix-tree layouts. The reason for the usage of this machine in our experimentation is that due to its large main-memory, we can overcome heavy I/O costs incurred during the construction of various suffix-tree layouts. The resources involved in the construction of various suffix-tree layouts are of less concern to us, as we are focusing mainly on the search aspect of the suffix-tree. But, we are mentioning it over here for the sake of completeness of the experimentation. So, this machine is not used for any kind of further experimental analysis, such as searching on the suffix-tree. On the other hand, machine S is a typical desktop system which is used for all the measurement related to the search performance of already constructed disk-based suffix-tree layout.

All layout algorithms are implemented in C++ and compiled with the XL C++ version 5.0.2.0 on machine R and with the GNU g++ compiler version 4.4.3 on machine S. In all the cases compilation is done with the optimization flag '-O3'.

One last point related to the experimental environment is the file-system used. Although we have given consideration to the location of pages within the suffix-tree file, we have never considered the actual physical location of the pages of suffix-tree file on the disk. But, we have tried to make it as less fragmented as possible which ensures that the nearness of pages within the suffix-tree file also results in physical nearness of pages on disk. This requires an efficient file-system which is good for handling large files. For this purpose, all our search-time related performance measurements are done on *xfs* file system [29] which is also known for excellent bandwidth for data-transfer.

5.2 Construction Time of Various Suffix-Tree Layouts

Table 5.1 shows construction time of various suffix-tree layout algorithms on machine R. The construction time of various layouts (except Trellis layout) includes either reorgani-

zation of layout (Chapter 3) or improvement in physical structures of internal nodes of the suffix-tree (Chapter 4) or both. Although layout reorganizations had taken considerable time, their large reorganization time is not an obstacle in the process of search-optimization because search-optimization is a one time process, whereas search applications can be repeatedly invoked on these resultant search-optimized suffix-tree layouts.

Suffix-Tree Layout	Source Layout or Sequence	Time Taken (in hrs.)
Trellis (construction layout)	Human Genome Sequence	62.87
SBFS	Trellis	33.17
Stellar	Trellis	43.02
1Cr4Cd	Trellis	35.32
bfs-hybrid	Trellis	25.09
OneLinkIn	Trellis	30.02
bfs-hybrid-IBO	bfs-hybrid	103.8
Trellis-EL	Trellis	3.49
bfs-hybrid-IBO-EL	bfs-hybrid-IBO	5.10
Trellis-ECC	Trellis	8.48
bfs-hybrid-IBO-ECC	bfs-hybrid-IBO	8.93

Table 5.1: Construction Time of various Suffix-Tree Layouts on Machine R

In the table, we can see that the time taken for the Trellis construction layout is larger than the construction time on a typical desktop machine (as mentioned in [21] and also observed by us on machine S). But, this timing also includes the time taken for merging of suffix-tree partitions (which is nearly 33% of total time taken to build the final Trellis construction layout). Also, the Trellis algorithm is main-memory intensive program and machine R has the main-memory shared across all processor through a common shared bus. As the Trellis code is not parallelized, it is using one processor only. In such cases the main memory latency on machine R is 4 times more than the main-memory latency on machine S. So, a program with frequent memory accesses is bound to take more time on machine R than on machine S. Similarly, large time taken by bfs-hybrid-IBO is due to computationally intensive optimized block sequence determination which took 98% of the total time taken for the construction of bfs-hybrid-IBO layout.

5.3 Search Results

Before moving onto actual search results, we first see the data-set used for searching and the environment in which the search experiments are conducted.

5.3.1 About Search Experiment Data

- All our experiments are performed on the suffix-tree built on the actual DNA sequence and not on the randomly generated sequence. In the suffix-tree built on randomly generated sequence, the data skew is absent and it will not represent the real world scenario in which searching is done on actual genome sequences.

- In most of the prior techniques for search-optimization mentioned in the literature, the search data-set contains either randomly generated sequences or sequences which are fully drawn from the indexed sequence. In both the cases, the search results do not represent the real world searches performed on actual DNA sequences. Because, in randomly generated sequence data, each of the character have equal probability of occurrence within the sequence, which is not the case in real DNA sequence. On the other hand, the data fully drawn from indexed sequence does not mismatch at any particular point in the suffix-tree traversal during search traversal.

  So, all our experiments use different kind of search sequences, i.e. randomly generated sequences, sequences which are fully drawn from indexed sequence and search sequences which have partial similarity with the indexed sequence. In each case, 200 sequences of length 500 to 10,000 characters with a stride of 500 are considered. We have kept the value of minimum threshold length (λ) to 40.

- For all the cases in search results, we have kept indexed sequence in the main-memory. The results could have been affected, if it cannot be accommodated in the main-memory. But, we can ignore it due to the following reasons:

  1. The size of search sequence is comparatively lesser than the size of the suffix-tree. And with frequent accesses to it, its referred portions are more likely to

remain in the main-memory then the portions of the suffix-tree (due to the caching effect of the operating system).

2. In the suffix-tree node, the relative position of pointers depict the first character represented by the edge going to the child node (refer Section 4.2.3.2). So, if the edge has just one character (which is the case for about 60% of the tree-edges as indicated in Figure 4.4), then we may not even need to access the indexed sequence.

3. When the indexed sequence is not fully accommodated into the main-memory, the increase in search-time would have been in same proportion across the results of all layouts.

5.3.2 Search Experiment Environment

We have tried to perform all the search experiment in the same environment. So, apart from the same hardware and same system parameters, we have tried to retain exactly the same situation for all the search experiments. Some of the obviously looking precautions are mentioned as follows:

1. No major background process from the user is initiated during this time, so that there will not be any impact on the resources that is being used.

2. All the files are kept on the same disk. For example, a search experiment with internal node file and leaf file kept on different disk, performed significantly better than the experiment with all the files on the same disk.

3. We have cleared the cache after every sequence is searched. So that the page cached during searching of previous sequence do not affect the searching of the current sequence.

5.3.3 Search Results: Layout Reorganization

Here, we present the search results on the layouts which are modified by just rearrangement of nodes and blocks. The results are presented in Figures 5.1 to 5.5.

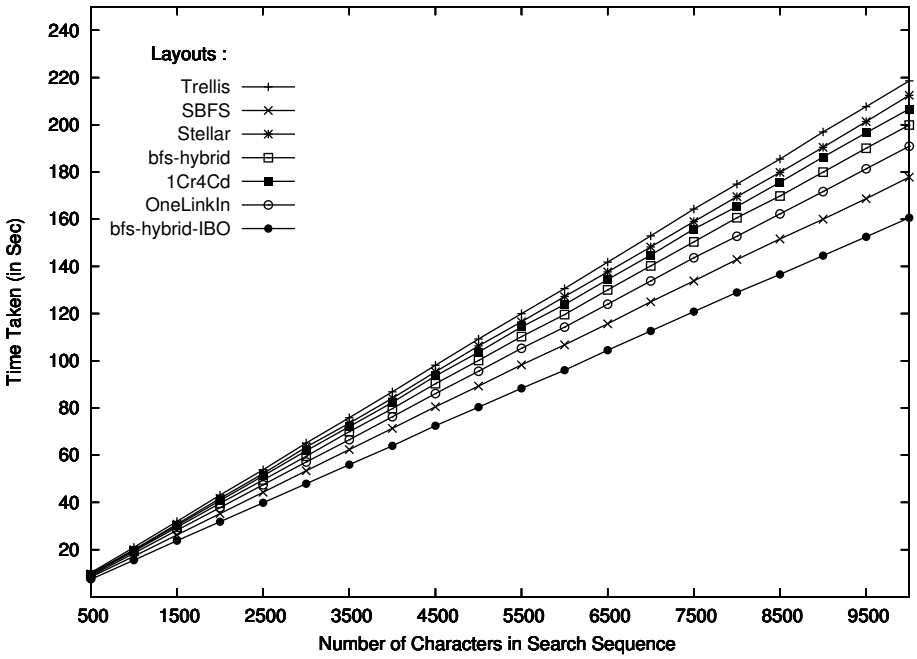


Figure 5.1: Layout Reorganization: Search-time for randomly generated search strings

In the results, we can observe a continuous increase in the search-time with the increase in the length of query strings. For example, in Figure 5.1, we get almost every result as a straight-line with positive slope. In this case the search strings are randomly generated and the average number of characters matched in this case is around 22 (which is lesser than our λ value of 40). So, the search traversal do not visit the sub-trees of the suffix-tree and it accessed nodes till middle dense levels of the suffix-tree (max level depth accessed is 15). But, even with only top portion access, all the layouts proposed by us outperform construction layout, whereas bfs-hybrid-IBO layout performed better then the existing layouts.

But, this trend of continuous increase in search-time is violated at a few places. This can be seen in the case of search strings whose 25% portion is similar to the indexed sequence (Figure 5.2). Let us analyze a case for the Trellis layout, when the length

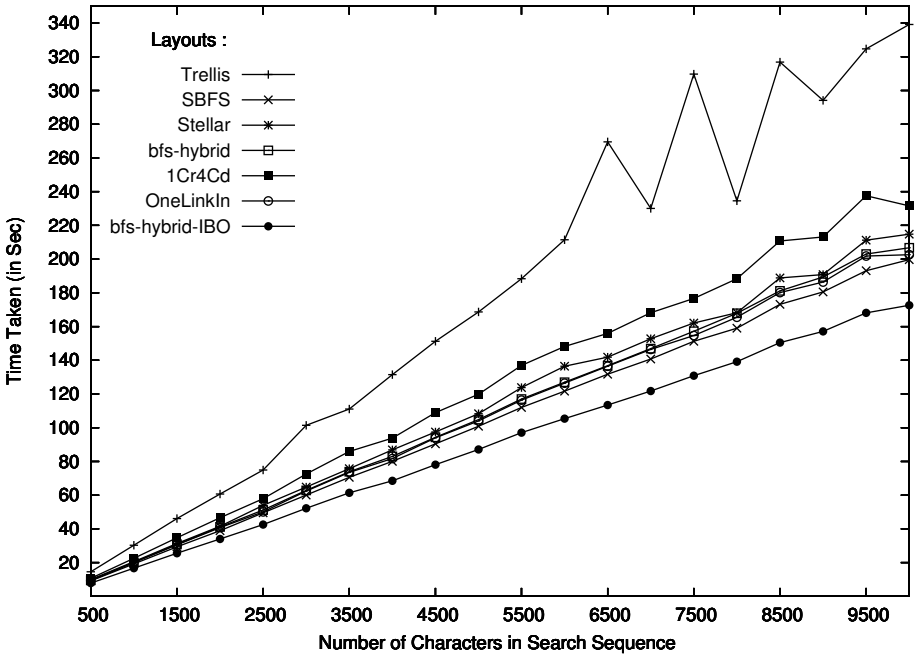


Figure 5.2: Layout Reorganization: Search-time for search strings with 25% similarity with the human genome sequence

of search string is increased from 8500 to 9000, the search-time is actually decreased. This seems to contradict the general sense of increase in search-time with the increase in string string length. But, after doing thorough analysis of this anomaly, we come to the conclusion that this case can happen with the searching over the suffix-tree. Let us call this situation as IDI (*Increase-Decrease-Increase*) situation in the search-time results.

There can be multiple reasons for this IDI situation to occur. For example, we have observed in this case that going from string length 8500 to string length 9000, the number of accessed leaf nodes actually decreased by 8%. And this decrease in leaf node access is common across many IDI situations that we have observed. Let us understand it, by taking an example from Figure 1.2. If we have to find the string AAC (for all its occurrences) in this tree, we have to visit 3 internal nodes and 2 leaf nodes. On the other hand if we try to find string CA (for all its occurrences), we have to visit 4 internal nodes and 4 leaf nodes. So, even with smaller length string, we have to traverse more nodes to locate it. And, if those nodes are far apart then the search-time may increase drastically. So, this reason for IDI situation, also leads us to the need of node rearrangement in the

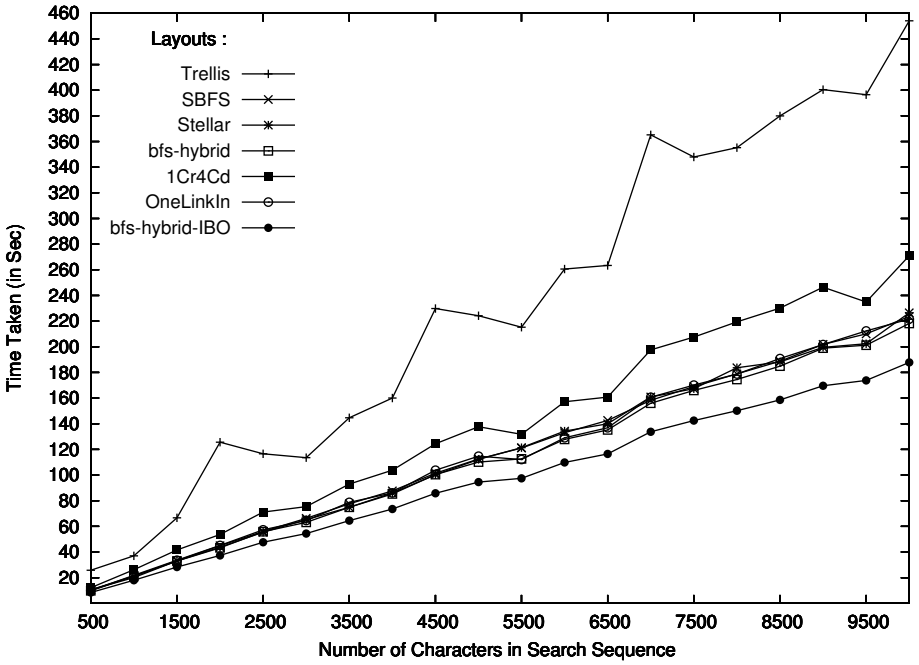


Figure 5.3: Layout Reorganization: Search-time for search strings with 50% similarity with the human genome sequence

suffix-tree layout. As we know that in Trellis the neighboring nodes are scattered (due to its DFS approach), this effect of IDI is more noticeable than the other layout, which have tried to bring the neighboring nodes as close as possible (by following certain criteria for localization).

Also, we have found that although the number of tree-edges traversed in case of sequence length 9000 is greater than the edges traversed in case of sequence length 8500, the percentage increase in the number of multi-character edges is more in the case of strings with length 9000. For example, going from 8000 to 8500, the percentage increase in multi-character edges is 11%, whereas in case of 8500 to 9000, this increase is 15%. Along with it, the average characters on multi-character edges in case of search strings with length 8000, 8500, 9000 are 1067, 221 and 1209 respectively. This means that more characters from search strings are considered with less number of edge traversals during searching.

This IDI situation is also visible in other layouts, but with lesser deviations (the reason for which has already been discussed). For example, in case of 1Cr4Cd layout, when search

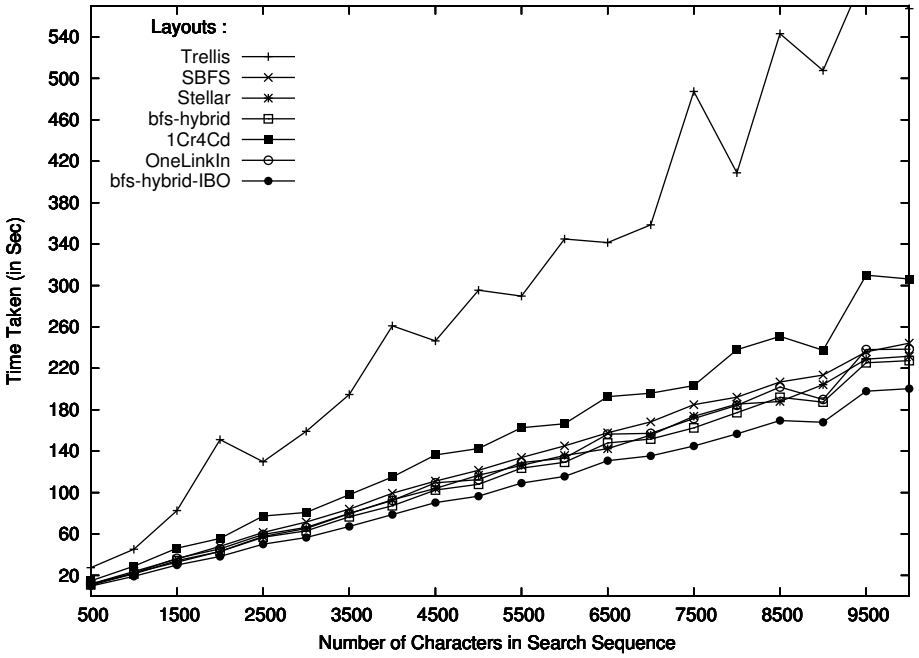


Figure 5.4: Layout Reorganization: Search-time for search strings with 75% similarity with the human genome sequence

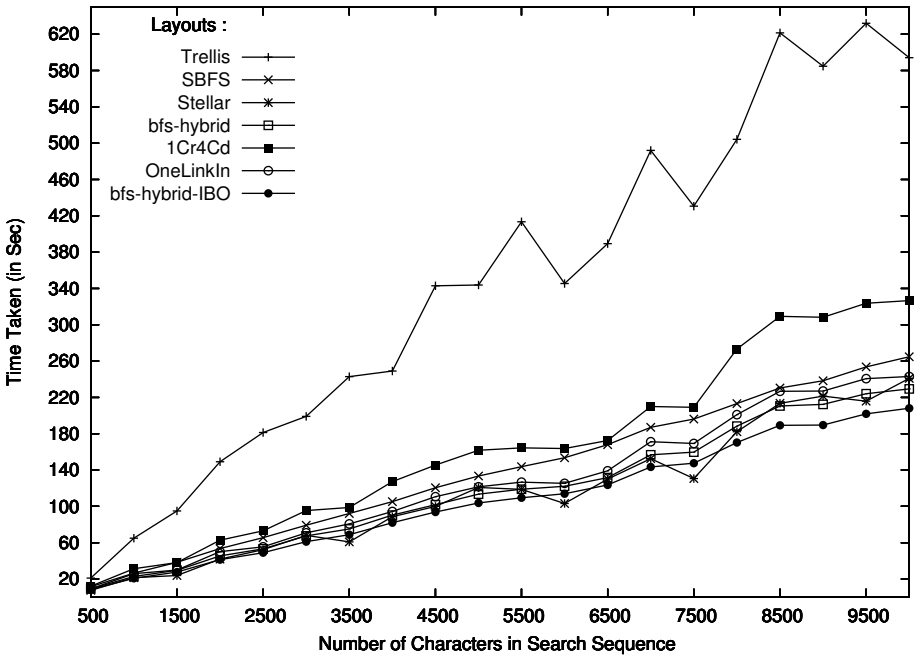


Figure 5.5: Layout Reorganization: Search-time for search strings fully drawn from human genome

strings have around 50% similarity with the indexed sequence (Figure 5.3), the search-time decreases when search string length is increased from 9000 to 9500. For this case also, we have observed similar deviations in traversal access pattern, as we have observed in the case of Trellis layout.

Let us now compare the performance of intra-block rearrangement layouts proposed by us with the existing layouts. Although for random search strings, these layouts have good performance, they are not consistent in their performance for different kind of search strings. For example, the bfs-hybrid layout, whose performance improves with the increase in the similarity portion, sometimes outperformed by the Stellar layout (for example, in Figure 5.5). This particular situation happened for the search strings which are fully drawn from the indexed sequence. In such cases mismatch on the tree-edges do not occur and suffix-links are visited only after leaf nodes get visited. Since, the Stellar had arbitrary localization of nodes, it is getting its localization exploited even in this situation (but, with the undesired effect of IDI). For the bfs-hybrid layout, the link nodes are given higher priority than the tree-edges for localization, when there is higher chances of mismatch (particularly in lower portion of the suffix-tree), which does not occur in this case.

For the remaining layouts with intra-block rearrangement, the search performance is not satisfactory when the search strings have more and more similarity with the indexed sequence. The reason given above is also applicable to the performance of the OneLinkIn layout. But, the consistent poor performance by 1Cr4Cd layout is due to its more weightage to suffix-links than tree-edges. In our search, the traversal ratio of tree-edges to suffix-links is around 70% to 30%. So, giving more weightage to suffix-links is actually not helping in reducing the search-time.

When a particular substring is located within the suffix-tree, whose length is more than the given threshold λ, the subtree below that node location is traversed in depth-first manner. So, it is surprising that all the layouts (existing as well as those proposed by us) consistently outperformed the construction time layout (whose nodes are store in depth-first manner). This shows the importance of search-optimization which is done by just rearranging nodes in breadth-first (or its variant) manner.

When inter-block optimization is applied to the bfs-hybrid layout (which is best among all the intra-block layouts), the layout of *bfs-hybrid-IBO*[1] is obtained, which significantly outperforms the construction layout for every kind of search strings. It also dominates state-of-the-art layouts in majority of the cases. The improvement in the search-time by this layout over the construction layout of Trellis is in the range of 25% to 75%. But, the average improvement over the different type of layouts is enlisted in Table 5.2. These results also highlight the need for rearrangement of blocks for search-optimization, which was not considered by the earlier techniques for search-optimization.

Improvement over Layout	Average Improvement in Percentage
Trellis	55.06%
SBFS	15.27%
Stellar	18.22%

Table 5.2: Performance Improvement by bfs-hybrid-IBO Layout over existing Layouts

For performance analysis of physical node structure improvement, we have chosen two extremely performing layouts – construction layout of Trellis and bfs-hybrid-IBO layout. Although physical structure improvements for suffix-tree internal nodes can be applied to any layout, we have restricted it to just two layouts, because of numerous combinations of layouts that will result from the application of these improvements. Now, we see the measurements for layout goodness, which also supports the results obtained in this section.

5.3.4 Quantification of Layout Goodness

In order to compare different layouts, based on their inherent characteristics, we need some quantification criteria for the layout goodness. Now, this quantification is also an indicative of the search performance measurements obtained earlier and depends either partially or fully on the properties of the layouts. It also depends on the type of search-data used, but it is independent of the machine on which it is generated or used. The various measurements for quantification of layout goodness are described as follows:

[1]IBO stands for Inter-Block Optimization.

5.3.4.1 Static Locality

In a disk-based suffix-tree layout, the *static locality* determines for every node in a block, how much percentage of its immediate child nodes or its immediate suffix-link node lie in the same block. The name of this criterion itself suggest the nature of the measurement performed on the suffix-tree layout. As this measurement is directly obtained from the layout, it is fully dependent on the properties of layout. This criteria was also used in [5] to show the goodness of a layout. The static localities for various layouts are shown in Figure 5.6.

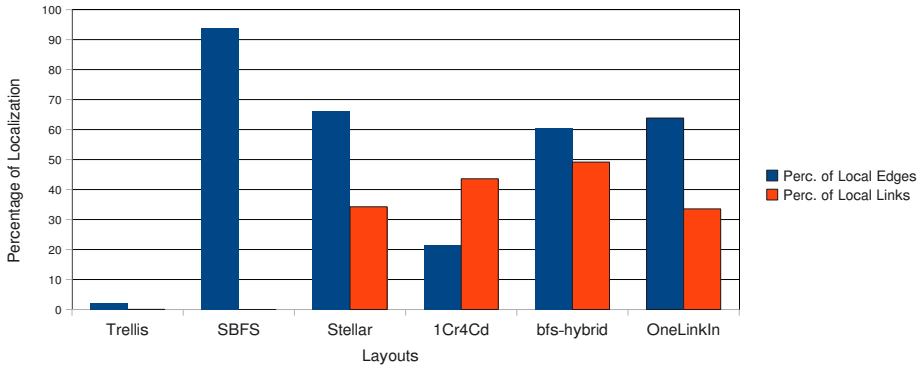


Figure 5.6: Static Locality for various Layouts for the Suffix-Tree constructed over the Human Genome Sequence

These measurements of static locality indicates the nature of node arrangement within the suffix-tree layouts. For example, while Trellis has nodes arranged in depth-first manner, it is actually localizing very few nodes. On the other hand, full emphasis on tree-edges by SBFS layout, gives it more than 90% static locality for tree-edges, while almost nil for suffix-links. The Stellar and the layouts proposed by us have emphasized both tree-edges and suffix-links, such that a balance is obtained between static locality for both kind of edges, which is necessary for complex traversal patterns involving both tree-edges and suffix-links.

Although the *static locality* gives good indication for tree-edge locality and suffix-link locality, we cannot take it as a final quantifying criteria for goodness. The reasons for this are illustrated as follows:

1. In this criterion, the localization of nodes within the blocks are considered, but no consideration is given to block sequence optimization (Section 3.4.3) which needs measurement for inter-block localization. This is the reason that we have not included *bfs-hybrid-IBO* layout in this analysis. Otherwise, this intra-block locality measurement would have given same measurement for *bfs-hybrid* and *bfs-hybrid-IBO*.

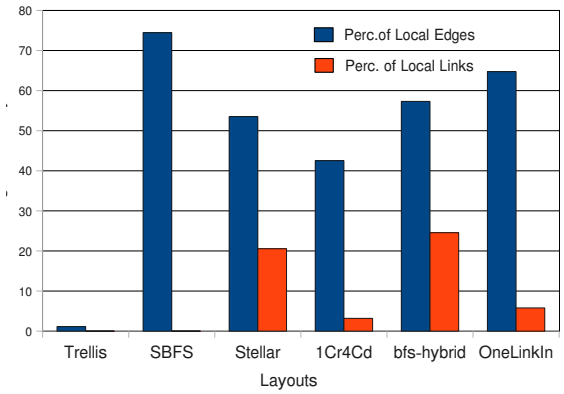
2. Although the percentage of local nodes are measured, it does not capture the relevance between localized nodes. For example, If a node with *nodeType* 4 has its suffix-link node local to it, then that is not of much use, because it is hardly going to get followed. This kind of relevance just skipped in this measurement.

In order to make *static locality* more effective, one more level of measurement can be added to its measurement. For example, instead of just analyzing whether immediate child nodes or suffix-link node are local or not (this is now the *First-Level of Static Locality* measurement), if we want *Second-Level of Static Locality*, then we also have to measure for the localization of child nodes and suffix-link nodes of immediate child nodes and immediate suffix-link node. This can be more accurate indication to the goodness of the layout. But, again it has same flaws as that of *First-Level of Static Locality* measurement.

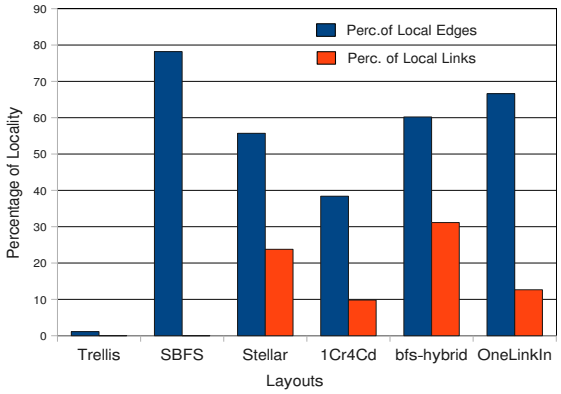
5.3.4.2 Dynamic Locality

Dynamic locality is similar to static locality, in a way that it also measures percentage of localization of immediate child and link nodes within a block. But, it is a representative of only a restricted subset of overall edges or links, which play important role in actual searching. This quantifying criterion is measured, when the actual search is being performed on the layout. So, measurement in this criterion is done at the time of searching, by determining how many of the child nodes or suffix-link node are local to a tree node.

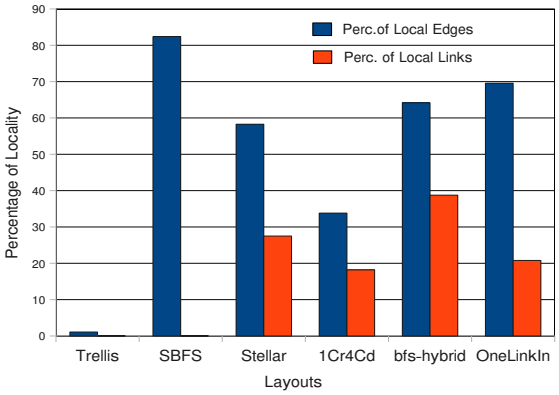
Dynamic locality depends on what kind of search strings are selected and based on that dynamic locality measurement changes. Few measurements of dynamic localities for



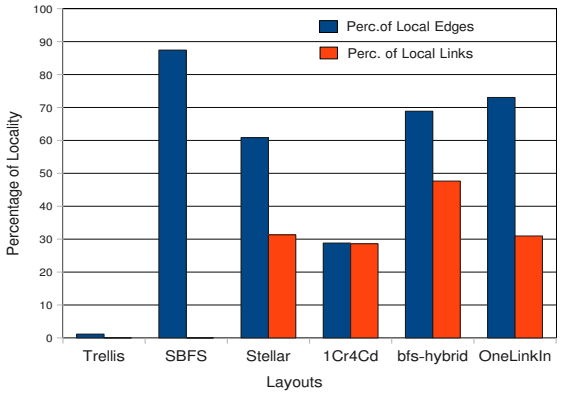
(a) Randomly generated search strings



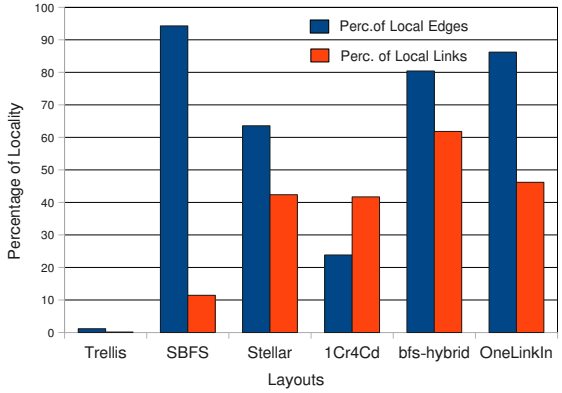
(b) Search strings with 25% part from human genome



(c) Search strings with 50% part from human genome



(d) Search strings with 75% part from human genome



(e) Search strings with 100% part from human genome

Figure 5.7: Dynamic Locality for various Layouts for different kind of Search Strings

various kinds of search strings are shown in Figures 5.7(a) to 5.7(e)[2]. In our case, for all the layouts, the dynamic locality measurements mimic static locality measurements (obtained in previous section). So, in that sense, static locality is also indicative of dynamic locality to some extent. For example, for Trellis layout, static as well as dynamic localities never exceeds 2% (for both tree-edge and suffix-link localization).

Since, dynamic locality represents only a subset of edges or links that have been considered in measuring the static locality, it need not always be lesser then the static locality which represents the average locality over whole of the suffix-tree. For example, when the search strings fully drawn from human genome sequence are used in searching over the SBFS layout, the dynamic locality obtained for tree-edges is 94.28%. Whereas the static locality of SBFS layout for tree-edges is 93.93%. Such situations with higher dynamic localities can arise when a particular path with higher locality is followed. And if the whole of search-string data is likely to follow localized path, then we can get overall dynamic locality either near to the overall static locality (or even higher in some cases). This also shows that the localization done statically in the suffix-tree layout is actually giving good localization at the time of search.

From the dynamic locality measurements, we can see the trends as we go from random search strings to search strings which are completely taken from the indexed sequence - *the edge locality as well as link locality increases* (except for edge locality for 1Cr4Cd Layout). The reason for increase in edge locality is that, as more part of search strings are taken from the indexed sequence, there is lesser and lesser chances of mismatch on the suffix-tree edges. And, if the localization criterion has given weightage to the tree-edges then tree-edge locality also increases. Note that 1Cr4Cd layout has not given weightage to tree-edges (except node with type 4 which constitutes only a smaller portion of the whole suffix-tree) and hence its tree-edge locality is bound to reduce. Similar reason can be given for the locality of the suffix-link as well.

[2]Each dynamic locality graph is corresponding to the search results of Figures 5.1 to 5.5

5.3.5 Search Results: Physical Structure Improvement of Internal Nodes

Here, we present the performance gain obtained by improving the physical structure of the suffix-tree internal nodes (refer Figure 5.8 to Figure 5.12). These improvements are implemented on the Trellis and the bfs-hybrid-IBO layout (as we have mentioned earlier in Section 5.3.3). The search data-set used in these experiments are same as the search data-set of previous experiments.

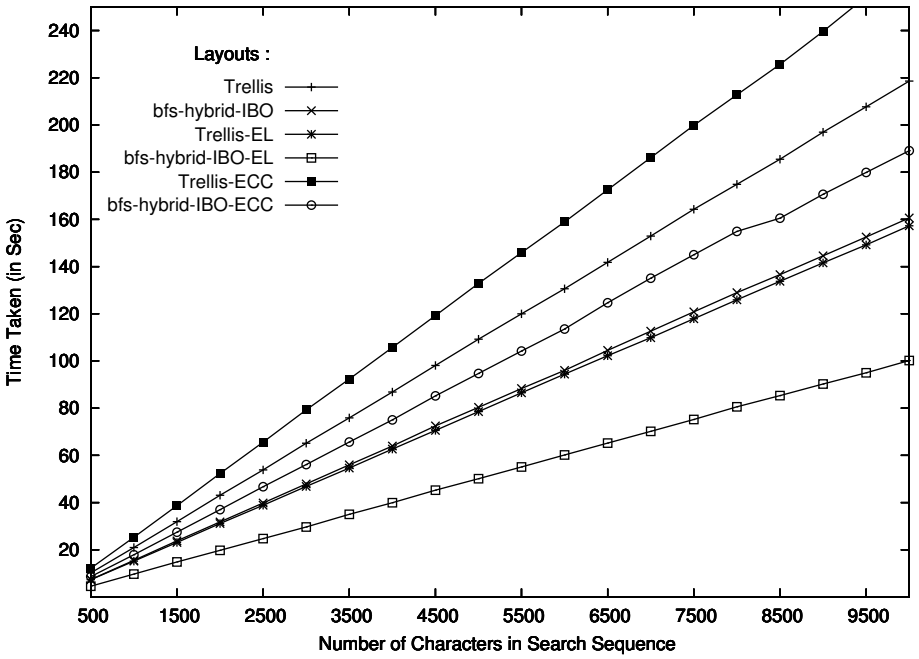


Figure 5.8: Node Structure Improvement: Search-time for randomly generated search strings

Now, we first look into the performance improvement obtained due to embedding of leaf nodes inside suffix-tree internal nodes (refer Section 4.2.2 for details). The two resulting layouts, i.e. Trellis-EL[3] (obtained from the Trellis layout) and bfs-hybrid-IBO-EL (obtained from the bfs-hybrid-IBO layout) consistently outperforms their parent layout. The improvement in search-time by Trellis-EL over Trellis Layout is in the range of 25% to 85%. Whereas the improvement by bfs-hybrid-IBO-EL over bfs-hybrid-IBO is in the

[3]Where EL stands for Embedded Leaf.

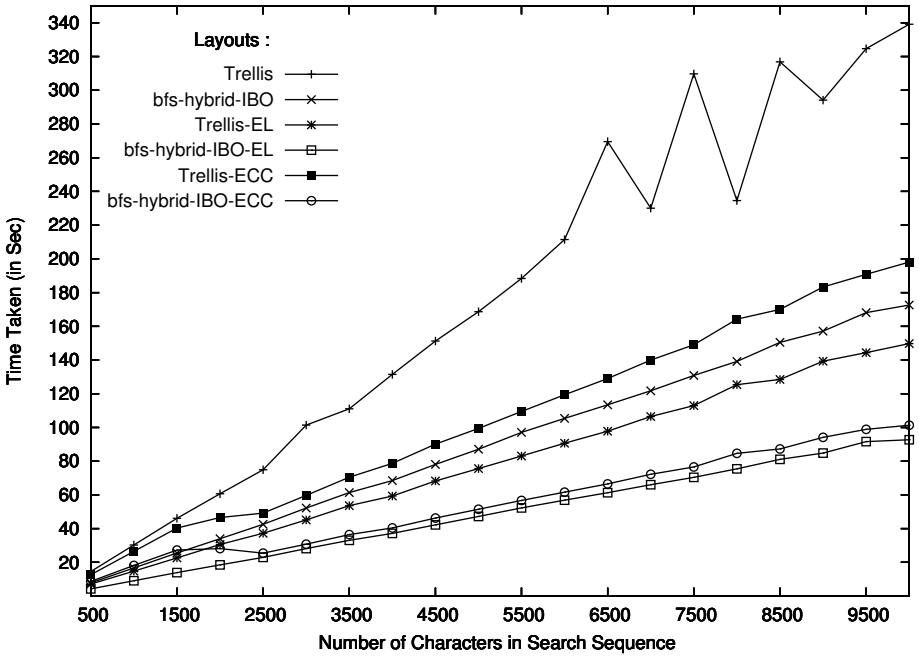


Figure 5.9: Node Structure Improvement: Search-time for search strings with 25% similarity with the human genome sequence

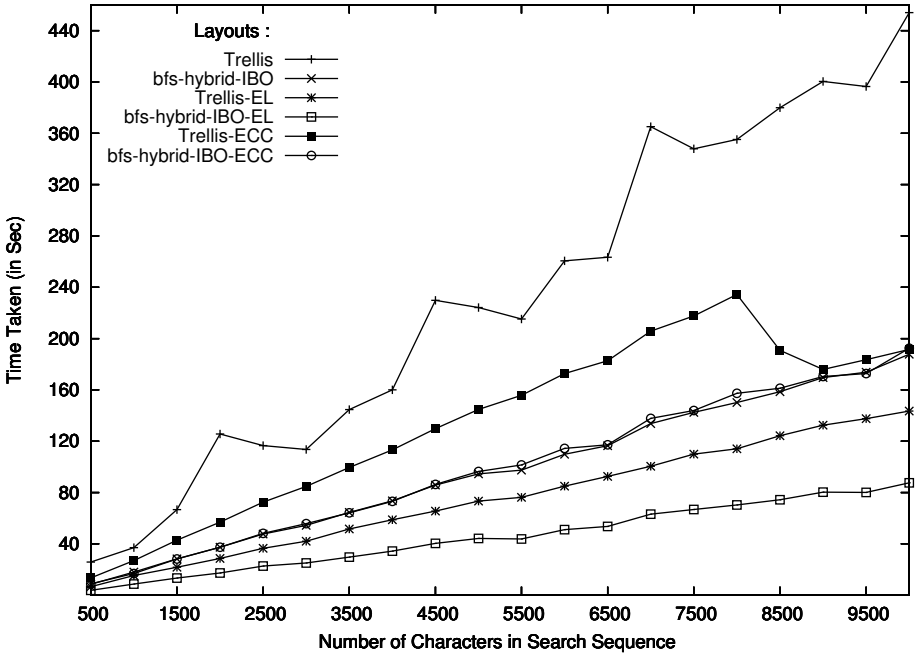


Figure 5.10: Node Structure Improvement: Search-time for search strings with 50% similarity with the human genome sequence

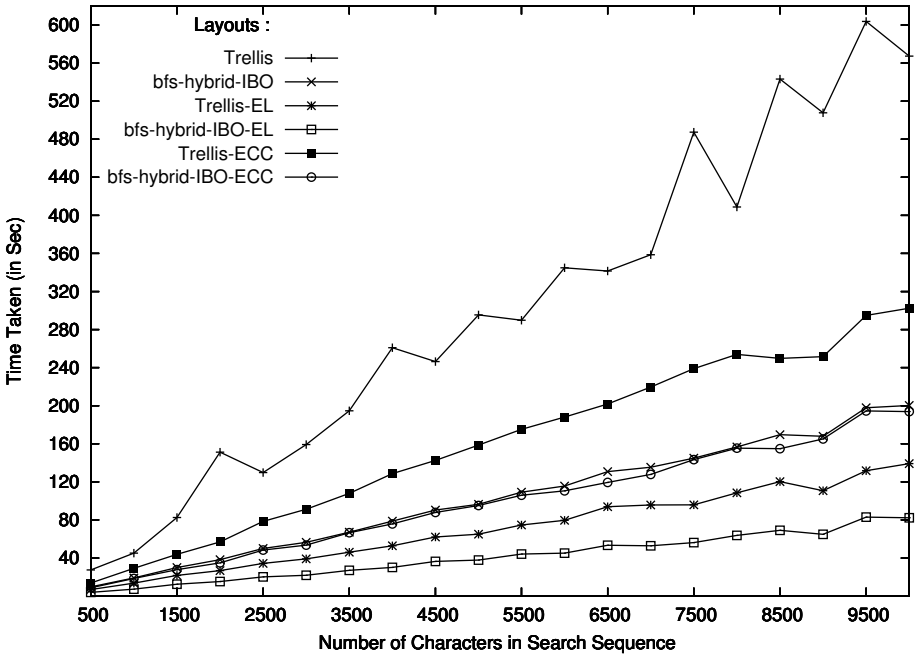


Figure 5.11: Node Structure Improvement: Search-time for search strings with 75% similarity with the human genome sequence

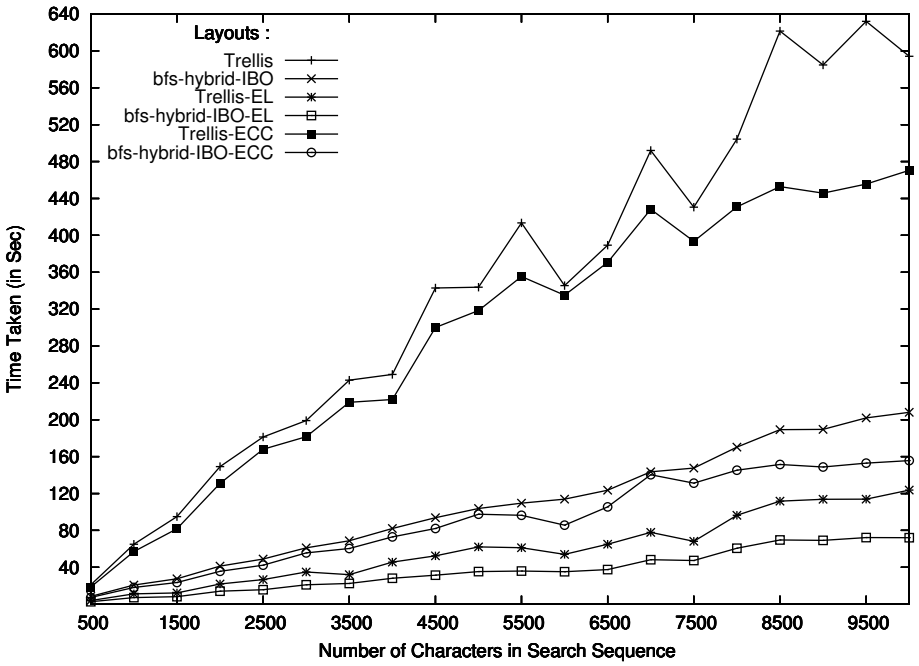


Figure 5.12: Node Structure Improvement: Search-time for search strings fully drawn from human genome

range of 35% to 70%. But, the interesting performance gain comes to our notice when performance of bfs-hybrid-IBO-EL is compared with the construction layout of the Trellis. In this case the improvement in the search-time is in the range of 50% to 90% – *an order of magnitude improvement.*

We now analyze the huge improvement in search-time obtained by the single improvement of embedding of leaf nodes. In this improvement, as the leaf nodes are totally contained within the parent internal node, the disk accesses as well as main-memory accesses are saved when the search traversal has to visit a leaf node.

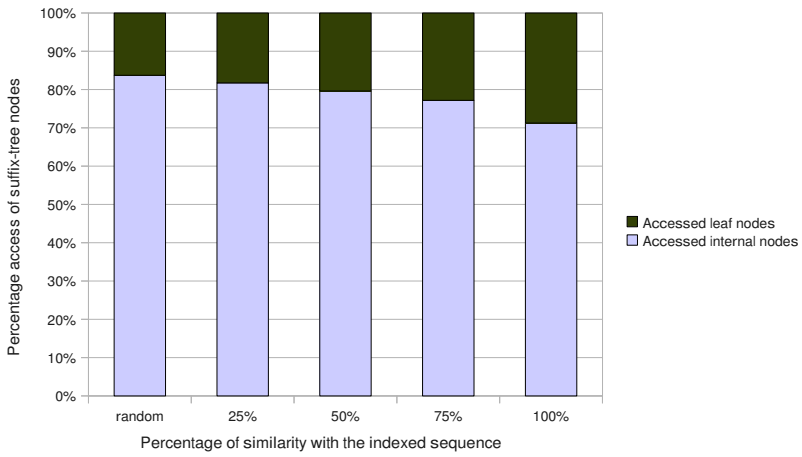


Figure 5.13: Percentage of Suffix-Tree Node Access : Internal node vs. Leaf node

As shown in Figure 5.13, the percentage of leaf node accesses varies between 16% to 29%. So, by embedding leaf into its parent internal node, we not only saved the disk accesses, but also achieved decrease in randomness that could have been introduced by leaf node access. For example, if internal node file and leaf node file is stored on the same disk then in order to get a leaf node (after an internal node fetching is done), the disk head has to move to an altogether different physical location on the disk. This operation involves a mechanical action (which is more time consuming). But with this improvement, we have tried to reduce it.

As the number of leaf nodes accessed increases with the increase in similarity of search strings with the indexed sequence (Figure 5.13), the difference in the search-time performance also increases. For example, performance improvement by bfs-hybrid-IBO-EL over

Trellis in case of random search strings is around 50%. But, for the same two layouts, the performance improvement in the case of search strings fully drawn from the indexed sequence is around 90%.

The effect of IDI situation is either very low or absent in this improvement, when we compare it to the effect of IDI situation of its parent layout. This is visible for the search string having 9000 characters in it and with 75% similarity with the indexed sequence (Figure 5.11). This is because, the leaf nodes are now embedded within the parent internal node.

The comparison of performance of *bfs-hybrid-IBO-EL* with the existing layouts is described in Table 5.3. Along with improvement in search-time performance, this embedding of leaf nodes also results in a 25% reduction in the suffix-tree space occupancy.

Improvement over Layout	Average Improvement in Percentage
Trellis	77.64%
SBFS	57.85%
Stellar	57.84%

Table 5.3: Performance Improvement by bfs-hybrid-IBO-EL layout over existing Layouts

Now, we look into the performance improvement obtained due to embedding of characters of the child edges into their parent internal nodes (refer Section 4.2.3 for details). The two resulting layouts, i.e. Trellis-ECC[4] (obtained from the Trellis layout) and bfs-hybrid-IBO-ECC (obtained from the bfs-hybrid-IBO layout) consistently outperforms their parent layout (except for the randomly generated search strings). The average improvement of Trellis-ECC over Trellis Layout is around 30%. Whereas the average improvement of bfs-hybrid-IBO-ECC over bfs-hybrid-IBO is 8%. But, when the performance of bfs-hybrid-IBO-ECC is compared with the performance of the construction layout of the Trellis, then the average improvement is 60%.

Now, we analyze the results obtained in this improvement. For the search strings generated randomly, layouts generated by this improvement have not performed better than their parent layouts. This is because, in that case the search traversal has not fully

[4]Where ECC stands for Embedded Child Characters.

explored the complex structure of the suffix-tree and hence the visit to multi-character edges is also limited. But, as the similarity portion (in the search strings) increases, the performance of these layouts improves over the performance of their parent layout. Because, in such scenarios, the mismatch on the child edge will help these layouts in reducing the search-time. In the search strings fully drawn from the indexed sequence, these layouts have slightly performed better than their parent layout. This is because, the search traversal has avoided access to the indexed sequence in many cases, by getting required information from the embedded characters of child edges.

The IDI situations in the results of these layouts also try to mimic the IDI situation of their parent layouts. But a peculiar case arise for Trellis-ECC for search strings with 50% similarity. In this case, the search-time decreases for query strings with length 8000 and more. We have found that the increase in edge traversal for sequence length 8500 to 10000 is mainly attributed to increase in multi-character edge traversal, whereas the increase in single character edge traversal is comparatively insignificant.

Hence, this physical structure improvement of embedded child edge characters of suffix-tree internal nodes also helps in reducing the search-time.

Chapter 6

Conclusions

Suffix-trees built over large sequences (such as the human genome sequence) are huge in size and require secondary storage (such as disk) for their construction and subsequent usage for varied search applications. There are several efforts which have focused on practically constructing genome-scale disk-based suffix-trees. On the other hand, scarcity of efforts for search-optimization of disk-based suffix-trees for search involving complex traversal patterns, limits the usage of suffix-trees for practical applications.

In this thesis, we have investigated the possibility of optimizing the storage of suffix-trees from search perspective. In our first step, we have reorganized suffix-tree layouts by assigning nodes to blocks and resequencing of blocks according to optimized sequence of blocks. In this reorganization, we have achieved significant reduction in the search-time, when this search-time is compared with search-time over various other layouts which include Trellis construction layout as well as state-of-the-art search-optimized layouts. Out of all the proposed layouts, the layout *bfs-hybrid-IBO* outperforms the Trellis construction layout by as much as *75%* in the best scenario. The average performance improvement by the *bfs-hybrid-IBO* layout over Trellis construction layout is *55%*. The same layout also outperforms the existing state-of-the-art layouts by a margin of *15%* on an average. While the layout reorganization does take considerable time, it is a one-time process whereas searches will be repeatedly invoked on this index.

In addition to the rearrangement of nodes and blocks, we also investigated the im-

provement in the physical structure of the internal nodes. We proposed two improvements in the structure of internal nodes. In the first improvement, we embedded the leaf nodes completely within the parent internal nodes without increasing their size. When this improvement is applied to the DFS layout of Trellis, it provides search-time improvements ranging from 25% to 85%, and when used in conjunction with aforementioned bfs-hybrid-IBO layout, the searches are speeded up by 50% to 90%. Additionally, this optimization results in a 25% reduction in the suffix-tree space occupancy.

In the second improvement of suffix-tree internal nodes, we have embedded the characters of the child edge into the parent internal node, if that child edge represents more than one character. When this improvement is applied to the Trellis construction layout, it provides 30% average search-time improvement. But, when this improvement is used in conjunction with bfs-hybrid-IBO layout, the average search-time improvement is 60% when compared with search-time of the construction layout.

Our experiments are conducted on complete human genome sequences that have in excess of three billion characters, with complex and computationally expensive user queries that involve finding the maximal common substring matches of the query strings on the genome database. The experimental framework is instrumented to provide a variety of supporting statistics, such as intra-block localities, to help explain the observed behaviors of the various suffix-tree construction and search algorithms.

In summary, our study and experimental results indicate that through careful choice of node implementations and layouts, the disk access locality of suffix-trees can be improved to the extent that upto an order-of-magnitude improvements in search-times may result relative to the classical implementations.

6.1 Future Work

Going ahead in the future, the work mentioned in this thesis can be extended in the following directions:

1. **Online Generation of Search-Optimized Layouts:** Currently, we are using

already constructed suffix-tree (in their construction layout form) and process them to one of the search-optimized layouts. These two steps can be merged into one, so that the search-optimized suffix-tree can be directly generated from the genome sequence.

2. **Cache-Consciousness:** In our work, we have not consciously tried to make the suffix-tree storage more cache-conscious. However, in the future, the gap between main-memory accesses and secondary storage accesses may decrease. For example, in the solid-state disks, the seek-time is very negligible. So, more focus on cache-consciousness will require in future, in order to reduce the search-time.

3. **Parallelization:** Although, the process of layout optimization seems to be following serial pattern, certain properties can be extracted from the suffix-tree, which can divide the huge computational task of layout reorganization among different processing elements, which will lead to reduced reorganization time for various layouts.

4. **Search Optimization for Suffix-Trees built on Protein Sequence:** In the current thesis, our focus is totally on the suffix-trees built on the DNA sequences, which can be extended to protein sequences as well. But, it will require careful choice of suffix-tree properties which are important from the search perspective. Because, the alphabet size in case of protein sequence is larger than the alphabet size for the DNA sequence. For example, protein data comprises of an alphabet made of 20 amino-acid symbols.

5. **Optimal Layout for a Search Data-Set:** For a particular search data-set, the sequence of accessed nodes should be noted. Then the existing layout should be modified to a new layout according to this sequence of accessed nodes such that the accesses required for the disk should remain at minimum level. This optimal layout for a search data-set can be used to further analyze the scope in search-time reduction.

References

[1] R. Baeza-Yates and G. Navarro, "A Hybrid Indexing Method for Approximate String Matching", *Journal of Discrete Algorithms*, 2000.

[2] M. Barsky, U. Stege, A. Thomo and C. Upton, "A New Method for Indexing Genomes Using On-Disk Suffix Trees", *In Proceedings of the 17th ACM Conference on Information and Knowledge Management*, 2008.

[3] M. Barsky, U. Stege, A. Thomo and C. Upton, "Suffix trees for very large genomic sequences", *In Proceedings of the 18th ACM Conference on Information and Knowledge Management*, 2009.

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

[5] S. Bedathur and J. Haritsa, "Search-Optimized Suffix-Tree Storage for Biological Applications", *In Proceedings of the IEEE International Conference on High Performance Computing*, 2005.

[6] W. I. Chang and E. L. Lawler, "Approximate String Matching in Sublinear Expected Time", *Inn Proceedings of the IEEE Annual Symposium on Foundations of Computer Science*, 1990.

[7] 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), 1999.

[8] A. L. Delcher, A. Phillippy, J. Carlton and S. Salzberg, "Fast algorithms for large-scale genome alignment and comparison", *Nucleic Acids Research*, 30(11):24782483, 2002.

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

[10] EMBL, `http://www.ebi.ac.uk/embl/`

[11] GenBank, `http://www.ncbi.nlm.nih.gov/Genbank/`

[12] A. Ghoting and K. Makarychev, "Serial and Parallel Methods for I/O Efficient Suffix Tree Construction", *In Proceedings of ACM SIGMOD International Conference on Management of Data*, 2009.

[13] D. Gusfield, "Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology", *Cambridge University Press*, 1997.

[14] E. Hunt, M. Atkinson and R. Irving, "A Database Index to Large Biological Sequences", *In Proceedings of 27th International Conference on Very Large Databases*, 2001.

[15] J. Karkkainen and S. Srinivasa Rao, "Full-Text Indexes in External Memory", *Algorithms for Memory Hierarchies*, 2003.

[16] P. Ko and S. Aluru, "Obtaining Provably Good Performance from Suffix Trees in Secondary Storage", *In Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching*, 2006.

[17] C. R. Lumb, J. Schindler, G. R. Ganger,D. F. Nagle and E. Riedel, "Towards higher disk head utilization: extracting free bandwidth from busy disk drives", *In Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, 2000.

[18] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches", *In Proceedings of the First Annual ACM-SIAM symposium on Discrete algorithms*, 1990.

[19] E. M. McCreight, "A Space-Efficient Suffix Tree Construction Algorithm", *Journal of the ACM (JACM), 23(2)*, 1976.

[20] N. Neelapala, R. Mittal and J. Haritsa, "SPINE: Putting Backbone into String Indexing", *In Proceedings of the IEEE International Conference on Data Engineering*, 2004.

[21] B. Phoophakdee and M. Zaki, "Genome-scale disk-based suffix tree indexing", *In Proceedings of ACM SIGMOD International Conference on Management of Data*, 2007.

[22] B. Phoophakdee and M. Zaki, "TRELLIS+: An effective approach for indexing genome-scale sequences using suffix trees", *In Proceedings of the Pacific Symposium on Biocomputing*, 2008.

[23] R. Sinha, S. Puglisi, A. Moffat and A. Turpin, "Improving Suffix Array Locality for Fast Pattern Matching on Disk", *In Proceedings of ACM SIGMOD International Conference on Management of Data*, 2008.

[24] Srikanta B. J., "BODHI: A Database Engine for Biological Applications", Ph.D. Thesis, Dept. of SERC, Indian Institute of Science, `http://dsl.serc.iisc.ernet.in/publications/thesis/srikanta.pdf`, April 2006.

[25] S. Tata, R. Hankins and J. Patel, "Practical suffix tree construction", *In Proceedings of 20th International Conference on Very Large Databases*, 2004.

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

[27] P. Weiner, "Linear Pattern Matching algorithms", *In Proceedings of the IEEE Symposium on Switching and Automata Theory*, 1973.

[28] S. Wong, W. Sung and L. Wong, "CPS-tree: A compact partitioned suffix tree for disk-based indexing on large genome sequences", *In Proceedings of the IEEE International Conference on Data Engineering*, 2007.

[29] XFS: A high-performance journaling filesystem, `http://oss.sgi.com/projects/xfs/`

[30] `http://docs.google.com/present/view?id=dczwht9g_3318qqfb6f5`

[31] `http://www.eurekalert.org/pub_releases/2008-07/wtsi-fhg063008.php`

[32] `http://www.eurekalert.org/pub_releases/2008-10/gb-cgl100408.php`

[33] `http://www.nature.com/nature/journal/v421/n6921/full/nature01402.html`

[34] `http://www.ornl.gov/sci/techresources/Human_Genome/home.shtml`

[35] http://www.ornl.gov/sci/techresources/Human_Genome/project/benefits.shtml

[36] http://www.reuters.com/article/rbssTechMediaTelecomNews/
idUSN16321068200908l6?rpc=60