Search-Optimized Persistent Suffix Tree Storage for Biological Applications

Srikanta J. Bedathur Jayant R. Haritsa

Technical Report TR-2004-04

Database Systems Lab
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560012, India

`http://dsl.serc.iisc.ernet.in`

Search-Optimized Persistent Suffix Tree Storage for Biological Applications

Abstract

The suffix tree is a well known and popular indexing structure for various sequence processing problems arising in biological data management. However, unlike traditional indexing structures, suffix trees are orders of magnitude larger than the underlying data. Moreover, their construction and search algorithms are extremely inefficient when implemented directly on disk. Recently, we have shown that it is possible to significantly speedup the on-disk construction of suffix trees through a careful choice of buffering policy and physical representation of suffix tree nodes.

In this paper, we explore the problem of performing efficient searches on disk-resident suffix trees. Specifically, we investigate the gains that can be achieved through customized node-to-page layout strategies. Through detailed experimentation on real-life genomic sequences, we demonstrate that a new layout, called *Stellar*, provides significantly improved search performance. The key feature of Stellar is that in addition to standard root-to-leaf lookup queries, it also supports complex sequence search algorithms that exploit the suffix link feature of suffix trees. These results are encouraging with regard to the ultimate objective of seamlessly integrating sequence processing in database engines.

1 Introduction

In the bio-informatics domain, a highly popular indexing mechanism for providing fast access to biological sequence repositories, which are growing at an exponential rate, is the *suffix tree* [12, 13]. Its appeal lies in its *linear* (in the size of the sequence) time and space complexity for construction, and its *linear* (in the size of the query) search complexity.

A unique aspect of suffix trees is that, unlike traditional database indexes which are typically a fraction of the database contents, their size is larger than the underlying sequence data. In fact, standard implementations of suffix trees require in excess of *an order of magnitude* more space than the indexed data! As a case in point, the entire 3 Gbp of Human Genome is fully representable in about 1 GB (with each DNA symbol represented with 2-bits), whereas the corresponding most space economical suffix-tree occupies close to 25 GB ($= 3\ Gbp \times 8.5\ bytes$). That is, it is straightforward to host the sequence data in main memory, but the suffix-tree itself needs to be disk-resident!

This piquant situation is rendered even worse due to *suffix-trees not being disk-friendly*, as a consequence of the random traversals across tree nodes induced by the standard construction and

search algorithms. There has been significant recent research activity to address this problem and design high-performance disk-resident suffix trees [3, 15, 20, 22]. However, virtually all these efforts have focussed on the *construction aspect*, that is, on how to build the tree efficiently on disk. In this paper, we take the next step of considering the *search aspect*, that is, on how to efficiently search disk-resident suffix trees.

Specifically, our focus is on whether it is possible to optimize the *layout* of the suffix tree with regard to the assignment of tree nodes to disk pages. While layout has been well-studied in the database literature for access structures such as kdb-trees, Quad-trees etc., we are not aware of any similar work on suffix trees. Further, carrying out this study for suffix trees poses *new* problems arising out of the following:

- The patterns of search traversals over suffix-trees are much more complex since both tree edges and special connectors called suffix links (detailed in Section 2) are involved.
- Presence of suffix links turn the suffix tree into a *cyclic* structure.
- Suffix trees are not inherently balanced data structures, unlike typical secondary memory index structures.

Our experiments with a variety of real genomic sequences against representative query workloads demonstrate that the currently available layout choices are *extreme* – they either optimize "vertical" traversal through the tree edges, or optimize "horizontal" traversal through the suffix links. But, sequence search algorithms typically need to traverse *both edges and links* – for example, to find all maximal matching substrings between the database sequence and a query, tree edges are used to walk down the tree matching the query sequence along the way, and then subsequent matches are found by following the suffix links [6, 4].

Given the above motivation for designing a holistic algorithm that optimizes the layout for both kinds of traversals, we present in this paper **Stellar** (Suffix Tree Edge and Link Locality AmplifieR), an algorithm that attempts to achieve this goal. Stellar is a linear-time, top-down strategy that utilizes the structural relationship between suffix links and the tree edges under associated subtrees to achieve high locality of both suffix links and tree edges. We quantify its effectiveness with a detailed performance study.

In summary, the contributions of this paper are as follows:

1. Demonstrating that standard layouts of suffix-trees optimize only either edge traversals or link traversals, resulting in slow searches of genomic sequences;
2. Presenting Stellar, a suffix-tree layout that optimizes both kinds of traversals, thereby providing significantly improved search performance.

In this paper, we compare the search performance of Stellar layout to the layout produced by the Ukkonen's construction algorithm [25], and by the disk-resident suffix-tree construction algorithm proposed by Ela Hunt et al. [15]. In both cases, we show that a careful layout of the suffix-tree is extremely beneficial in terms of search performance.

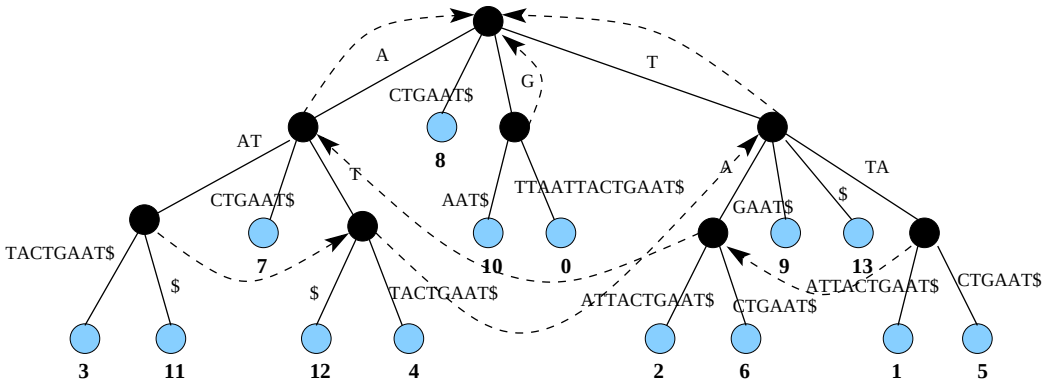


Figure 1: Suffix tree for the DNA fragment `GTTAATTACTGAAT$`

Organization

The rest of the paper is organized as follows: Section 2 provides a brief introduction to suffix trees and Section 3 introduces sequence search algorithms based on suffix trees. Then, in Section 4, we present existing index layout strategies, discuss issues in their application to suffix trees and present results to show their ineffectiveness. The design of our new Stellar layout algorithm is given in Section 5. The experimental setup is described in Section 6 before highlighting the results of our experimental analysis in Section 7. In Section 8, we move on to the analysis of sequence search algorithms in the absence of suffix-links in the tree, and present results showing the importance of suffix-links from the efficiency perspective. Results of our experiments over protein sequence data are presented in Section 9. We review related work in Section 10, and finally, summarize our results and outline our future research directions in Section 11.

2 Background on Suffix Trees

A suffix tree of a string is a compacted trie over all the suffixes of the string. For example, consider the suffix tree constructed over a DNA fragment, S = "`GTTAATTACTGAAT$`" shown in Figure 1 (the internal nodes of the tree are filled in dark and the leaf nodes are lightly shaded). The solid edges between nodes represent tree edges, and directed dashed lines show *suffix links*. The suffix links play an important role in linear time construction of suffix trees [17, 25] as well as in many search algorithms over suffix tree [5, 24, 4].

For the ease of exposition, we briefly present some terminology associated with suffix trees. Let $S[0..N]$ be the string indexed by the tree T. The leaf node corresponding to the i-th suffix, $S[i..N]$, is represented as l_i. An internal node, v, has an associated length $L(v)$ which is the sum of edge lengths on the path from root to v. We represent by $\sigma(v)$, the path label of v, to represent the substring $S[i..i + L(v)]$ where l_i is any leaf under v. A suffix link $sl(v) = w$ exists for every node v in the suffix tree such that if $\sigma(v) = a\alpha$, then $\sigma(w) = \alpha$, where a is a non-null symbol from the alphabet and α is a substring (possibly null) of the string. Note that $sl(v)$ is defined for *every* node in the suffix tree. And, more importantly, $sl(.)$ – the entire set of suffix links, form a tree rooted at the root of T, with the depth of any node v in this $sl(.)$ tree being

$L(v)$.

Since the time Weiner [27] introduced the suffix-tree data structure and a linear time algorithm for its construction, there has been growing interest in more space- and time-efficient algorithms for construction of suffix-trees. Conceptually different and space efficient algorithms to build suffix-trees in linear time have been given by McCreight [17], and later, by Ukkonen [25]. Further, McCreight also suggested the use of linked-list implementation of nodes for reducing the space overhead of the suffix-trees. All these algorithms make use of suffix-links to achieve linear-time construction and are implemented for various constant-sized alphabet datasets.

However, until recently, suffix-trees were not considered for persistent construction and maintenance as linear-time suffix-tree construction algorithms show very poor performance. The main bottleneck in the direct application of these algorithms for persistent suffix-trees is considered to be the random seeks induced during construction [8]. It has also been noted in [15] that suffix-links utilized by all these algorithms traverse the suffix-tree "horizontally", while edges span the tree "vertically". Thus, atleast one of them is expected to result in random access of memory. This is true only if the tree is stored on disk using depth-wise traversal of either edges of the tree or links of the tree. But this storage pattern is not feasible during the on-line construction of persistent suffix-trees. Therefore, in practice, *both* edges and suffix-links show non-local access patterns. In fact, even in a recent work [18], it was reported that whenever the dataset is large enough suffix-trees are not a viable option of indexing, since the memory is too small to hold the index completely.

2.1 Construction of Disk-resident Suffix-tree

In [15], a phased construction approach to building suffix-trees was proposed, wherein they use an asymptotically quadratic algorithm for construction of suffix-trees without suffix-links. Their technique is based on a mapping of all suffixes of a given database sequence D to *disjoint* set of partitions, $p_w \in P$, such that suffixes with equal prefix w are mapped to the same partition $p_w = \{wx | wx = s \text{ and } s \text{ is a suffix of } D\}$. The prefix length, w, is chosen such that the size of each suffix-tree partition does not exceed the available main memory size. Otherwise the algorithm fails. Since their phased approach involves constructing parts of the suffix-tree within memory and writing it to the disk completely, their empiricaly evaluation showed its superiority over traditional linear-time suffix-tree construction algorithms. But, their results on the performance of traditional construction algorithms do not consider the effects of paging policies and the storage management issues. In fact, in [16], it has been reported that a possible bottleneck could be the choice of checkpointing scheme of PJama, the underlying persistent mechanism used in [15].

One of the drawbacks of the above approach is that it is not sensitive to the skew in the distribution of symbols of the sequence. Due to this, the size of the partitions of the tree that are built within memory could vary resulting in a non-optimal utilization of available memory. In order to overcome this, an extended partition generation scheme was proposed in [20]. An initial pass over the complete sequence is made in order to estimate the cardinality of each partition. Based on these estimates, the prefix-length used to prepare the partitions is tuned to improve the main memory utilization.

Recently, in [22], a sophisticated partition based technique called *Top Down Disk-based* (TDD)

was introduced, that takes into account the effects of buffering policies during the suffix-tree construction. The TDD technique is a well-tuned combination of the partitioning approach introduced in [15] and an earlier main-memory algorithm called *wotd* (write-only top-down) [10] that was shown to incur fewer cache-misses on modern processors.

All of the above persistent suffix-tree construction algorithms suffer from a common significant drawback – the suffix-tree built using these algorithms is completely devoid of suffix-links. Although a large number of algorithms over suffix-trees do not exploit the presence of suffix-links, some of the critical applications of suffix-trees in bioinformatics such as MUMmer [6] depend on the availability of suffix-links. Therefore it is important to explore ways to speed up the suffix-tree construction without affecting their structure in any way. This was the aim of our previous work [3], which introduced a novel memory paging algorithm called TOP-Q that takes into account the probabilistic access pattern over the suffix-tree during its construction using the traditional linear time algorithms, such as those of Ukkonen or McCreight, to reduce the amount of disk I/O incurred.

3 Suffix-tree based String Searching

Suffix trees are useful in a large number of sequence search tasks [12], such as exact matching of pattern strings, identification of prefix-suffix pairs over a collection of sequences, common sub-string location, and so on. A particularly critical use of suffix trees is in pre-processing a large genomics data repository and utilizing the index to efficiently answer a large number of similarity searches. In these searches, the suffix tree index is used to quickly locate all common substrings between the database and the given query string. These are subsequently used to generate "local alignments", the regions of similarity between the sequences, through the use of various domain-specific heuristics.

Many of the high-speed algorithms designed for use with suffix trees exploit the presence of inter-node paths involving both tree edges and suffix links to the fullest. For example, techniques that significantly accelerate the accurate dynamic programming (DP) based approximate string matching [5, 24], exploit the suffix links to reduce the number of DP computations required.

At this point, we hasten to add that there are equivalent algorithms that use *only tree edges* for every algorithm that uses suffix links. However, they could be orders of magnitude *slower* than these high-speed algorithms, thus rendering them impractical in many genomics applications.

In this paper, we focus on the task of identifying *all common maximal substrings* between a large database sequence D that has been preprocessed into a suffix tree $\mathcal{T}$, and a pattern string Q, subject to the constraint that the match is longer than a user-specified cutoff length λ. This is an extremely important search task since it forms the backbone of many *sequence alignment* algorithms such as BLAST [2] and MUMmer [6] that have become extremely popular among genomics researchers.

Definition 1 (Maximal Common-substring Search) *Given a database sequence S, and a query sequence Q, locate $Q[i \ldots i+j]$ and $S[k \ldots k+j]$, such that, $1 \leq i \leq |Q|$, $1 \leq k \leq |S|$, $Q[i \ldots i+j] = S[k \ldots k+j]$ and $Q[i+j+1] \neq S[k+j+1]$. In practice, it is desired that only*

matches that satisfy a user-defined minimum threshold length, λ, are reported. In other words, $j \geq \lambda$. □

As an example, consider the following database sequence – GTTAATTACTGAAT$ which has been preprocessed into a suffix tree shown in Figure 1. In this suffix tree, the solid lines between nodes are the tree-edges, and the dashed edges indicate the suffix-links. The numbers at the bottom of the leaf nodes represent the start index of the suffix associated with them. Now, given the query sequence CTAATGACT and with λ set to 3, the desired common maximal substrings between the database sequence and the query sequence are: {TAAT, AAT, TGA, ACT}. Note that although CT is a common substring, it is not reported since it does not satisfy the match length restriction.

3.1 Searching for Maximal Common Substrings

Now, we describe the MaximalSubsringSearch algorithm, that utilizes the suffix links to locate the maximal common substrings quickly. This was proposed in [4] as a first step in their approximate string matching algorithm, which has been implemented in the popular genomics tool MUMmer [6].

The basic idea of the MaximalSubstringSearch algorithm is to locate the longest match between Q and D by walking down the suffix tree $\mathcal{T}$ using symbols of Q – matching them "letter-at-a-time". Subsequent longest matches are found by following the suffix links and going down the tree at the target of the traversed suffix link. A brief pseudo-code of the algorithm is provided in Algorithm 1 – detailed description of the algorithm, available in [4], has been omitted due to space constraints. In the rest of the paper, this search strategy is referred to as *LST algorithm* to indicate that it is a suffix-link based algorithm.

3.2 Locating Common Substrings without Suffix-links

The MaximalSubstringSearch algorithm presented above depends heavily on the availability of suffix-links in the disk-resident suffix tree. However, there are situations where suffix-links are completely *absent* from the suffix tree index. For e.g., when the disk-resident suffix tree is constructed using the techniques presented in [15] or [22], which completely dispense with suffix-links from the suffix trees in order to achieve faster disk-resident construction.

In order to locate all the maximal common substrings between D and Q when D has been processed into $\mathcal{T}'$, a suffix tree without suffix links, we use the observation that every common substring must result in a prefix match between corresponding suffixes in D and Q. This leads us to the following algorithm – use each suffix of Q to walk down the suffix tree $\mathcal{T}'$ from the root node, until either the suffix is completely located or there is a mismatch. If the length matched is greater than the value of λ, then add to the output set $\mathcal{L}$ all the leaf nodes under the current location. Follow this process for all the suffixes at positions from 0 to $|Q| - \lambda + 1$. We refer to this algorithm as *UST algorithm* in the rest of the paper.

Algorithm 1 Maximal Common Substring Search

MaximalSubstringSearch ($\mathcal{T}$, Q, λ, D)

Input

$\mathcal{T}$: Suffix-tree over the database sequence D

Q : Query string

λ : Minimum match-length to be reported

D : Database sequence

Output

$\mathcal{L} = \{(l, q, d) \mid Q[q \ldots q+l] = D[d \ldots d+l],\ Q[q+l+1] \neq D[d+l+1],\ l \geq \lambda\}$

Complexity

$O(|Q| + loc)$, where loc is the number of locations of match.

```
1: v ← root of T; j ← 0; k ← 0; L = φ
2: for i = 0 to |Q| do
3:    (v', j) ← StepDown(v, Q[i...]) {v' is the node at which matching has stopped, and j is the
      length of the match}
4:    if j ≥ λ then
5:       L = L ∪ TraverseSubtree(v')
6:    end if
7:    if IsLeaf (v') = true then
8:       k = v'.edgelen − j
9:       v' = v'.parent
10:   end if
11:   v = v'.suffixlink
12:   v = SkipDown(v, k, Q[i...]) {Use the skip-count trick [12] to traverse without comparisons}
13: end for
```

4 Suffix Tree Layout on Disk

Suffix trees, unlike popular secondary memory index structures such as B$^+$-Trees and R*-Trees, are not inherently balanced data structures – their structure depends entirely on the combinatorial characteristics of the sequence being indexed. In addition, the fan-out degree of suffix tree nodes cannot be varied in tune with the disk-page size. The fan-out of each internal node of a suffix tree is upper-bounded by the size of the alphabet of the indexed sequence. Hence, many nodes of a disk-resident suffix tree will be stored on a page, with nodes interconnected within as well as across pages. Therefore it becomes critical to choose the nodes that will be placed in the same disk-page in order to reduce the overall disk I/O cost of traversing the suffix tree during search.

Earlier research on the layout of disk-resident indexes [7] has shown that a heuristic-based linear-time algorithm, henceforth called SBFS, that does recursive localized breadth-first layout of the tree outperforms other commonly considered tree layout methods such as Breadth-first and Depth-first strategies. Through empirical studies they also show that SBFS is not too far from an optimal layout algorithm that has time complexity *quadratic* in the number of nodes in

the tree.

The basic idea behind the SBFS packing strategy is to recursively perform many local breadth-first traversals, beginning from the root of the tree, packing nodes in the order of visiting them into disk pages. Once enough nodes have been visited to fill a page, or there are no more nodes to be visited, the nodes visited so far are assigned to a page. Each of the remaining nodes in the BFS queue then becomes the root of a separate SBFS traversal. The recursion terminates when all nodes have been visited.

4.1 Issues in Disk-Resident Suffix Tree Layout

The search algorithms over suffix trees exhibit complex traversal patterns, significantly different from those commonly found in traditional indexing structures. In typical index structures the queries are mostly lookup queries involving root-to-leaf traversals. On the other hand, searching over suffix trees involves simultaneous use of two intertwined tree structures, one due to the tree edges and the other due to suffix links. Thus, the layout strategy also should take into account the two "orthogonal" traversal paths during suffix tree based search.

In addition to the issue of complex search traversal patterns, suffix trees exhibit higher inherent structural complexity than typical tree index structures due to the presence of *cyclic substructures*. As pointed out in Section 2, the collection of tree edges as well as the collection of suffix links in a suffix tree form two separate rooted tree structures. Also note that in the tree structure induced by the collection of suffix links, the links between nodes are *reversed* from the natural "parent-to-leaf" direction. That is, there exists a directed path starting at any internal node to the root of the suffix tree (also the root of the tree induced by suffix link collection), via a chain of suffix links. And, from the root node any of the internal nodes are reachable through a chain of tree edges, thus completing a cyclic path.

Due to these complexities, none of the previously proposed layout strategies that are designed to work with either tree structures or DAG (directed acyclic graph) structures are directly applicable in the context of suffix trees. Nevertheless, we investigate the efficacy of SBFS strategy outlined above for laying out a disk-resident suffix tree, by *ignoring* the suffix links during the layout process.

4.2 Comparing the Quality of Layouts

Before we can evaluate different layout strategies, it is required to develop a metric that can effectively capture the structural variations between suffix trees laid out with alternate layouts. One straightforward way to evaluate the quality of layouts obtained using different storage strategies is to execute a number of queries over the suffix trees laid out using these strategies and measure the disk I/O cost. However, this evaluation depends heavily on the characteristics of the query workload. It does not immediately reveal to us the structural properties of the layout that could affect general search workloads.

The overall efficiency of a disk layout depends on the amount of inter-connectivity of nodes within a disk page. The nodes in the suffix tree are interconnected through either tree edges or suffix links (or both). Hence, it is possible to capture the structural effectiveness of a layout

Dataset	Storage	Suffix-Links	Tree Edges
Human Chromosome 2	CO	41.8%	0.2%
	SBFS	0.1%	77.5%
	Stellar	**40.0%**	**62.6%**
C. elegans Chromosome 2	CO	39.7%	0.3%
	SBFS	0.01%	76.4%
	Stellar	**39.6%**	**61.6%**
Drosopila Melanogaster genome	CO	33.1%	0.006%
	SBFS	0.0%	68.5%
	Stellar	**38.8%**	**59.2%**
Symmetric Bernoulli	CO	27.6%	0.0%
	SBFS	0.0%	69.8%
	Stellar	**38.8%**	**57.7%**

Table 1: Static Edge and Link Localities

strategy through the amounts of suffix tree edges and suffix links that are entirely *within a page*, i.e., the source-target pairs are placed in the same diskpage.

Using this metric, we evaluated the following two layouts for suffix trees: (i) CO (Creation Order) – the "natural" layout produced at the end of suffix tree construction using the algorithm of Ukkonen [25], by ordering the nodes as they are created during the construction process, and (ii) the SBFS layout. Table 1 summarizes the results of this evaluation, providing for each layout strategy, the number of tree edges and suffix links that are *intra-page* as a fraction of the total number of tree edges and suffix links in the suffix tree. These values were obtained with suffix trees built on a 25 million basepair(Mbp) length DNA sequence drawn from Human Chromosome 2, and 15Mbp length of C. elegans Chromosome 2, with disk pagesize set to 4KB.

As can be observed from these results, CO and SBFS layouts represent (negative) extremes in disk-resident suffix tree layout. The CO layout exhibits practically *no tree edge locality* while providing good suffix link locality. The SBFS strategy is at the other end of the spectrum, with a large fraction of tree edges contained within a page and virtually *no suffix link locality*.

In contrast, results for the suffix trees ordered through our Stellar layout, also presented in Table 1, show suffix link locality close to that of CO and tree edge locality comparable to that of SBFS – clearly optimizing both forms of connections in the suffix tree.

Before we move on to the description of Stellar, we explore the reasons for this extreme behavior of CO and SBFS layouts:

CO Layout: During the suffix tree construction, two successive internal nodes v_1 and v_2 are created typically as follows:

1. Traverse the suffix link of the $parent(v_1)$ to reach $ancestor(v_2)$, and
2. Walk down the tree from $ancestor(v_2)$ using tree edges, until a mismatch in the tree edge results in the creation of v_2.

And, most importantly, the nodes v_1 and v_2 are related to each other through a suffix link, since they correspond to consecutive suffixes of the sequence processed so far by the online construction. Due to this sequencing of tree node creation, a large fraction of suffix links in the tree tend to be contained within a page.

Although we have considered the construction algorithm of Ukkonen [25] in our discussion, we performed similar experiments with McCreight's construction [17] as well, and found that the results are identical.

SBFS Layout: The SBFS ordering, in contrast, is designed to cluster the tree nodes related through tree edges into a diskpage. In a suffix tree, the nodes related through a tree edge share a common prefix – the path label of the parent is common to that of child. Thus, SBFS layout translates into a preferential clustering of suffix tree nodes that correspond to substrings with common prefix. However, the nodes with common prefix have very low probability of also being related through a suffix link – a situation that could occur only due to consecutive run of a symbol in the sequence. Thus, the suffix link locality of SBFS is extremely poor.

4.3 Search Utilization of Links and Edges

The results shown in Table 1 represent only the static structural property of different layouts. It is also important to consider the relative *utilization* of suffix links and tree edges during searches, and whether the search tasks require combined locality of both forms of connections.

Figure 2 shows, for different query collections (described in Section 6), relative utilization of tree edges over that of suffix links during maximal substring search as the value λ is varied in the typical operational region. Note that we also include the match-location reporting phase, which uses only tree edges to traverse the subtree under the match.

These graphs demonstrate that although searches involve more traversals of tree edges than suffix links for lower values of minimum match length, the differential is within a small constant factor. Further, as the λ value increases, the utilization of tree edges converges to be within a factor of 2 of the suffix links used. Therefore, the number of suffix links traversed is comparable to the number of suffix tree edges used during searches over the suffix tree – suggesting that the search algorithms benefit by simultaneously improving the number of intra-page tree edges as well as suffix links.

5 Design of Stellar

The design of Stellar is based upon the relationship between nodes connected through a suffix link and the tree edges under them, which can be easily derived based on the well-known structural properties of suffix-trees [12]. The property we exploit is as follows:

Property 1 *If $v_2 = sl(v_1)$, then all the suffix links originating from the nodes under v_1 point only to the nodes under v_2.*

Proof: Let the path label of v_1 be defined as $\sigma(v_1) = x\alpha$, where x is a symbol from the given alphabet Σ, and α is a non-empty substring of the string being indexed. Then, $\sigma(v_2) = \alpha$, since

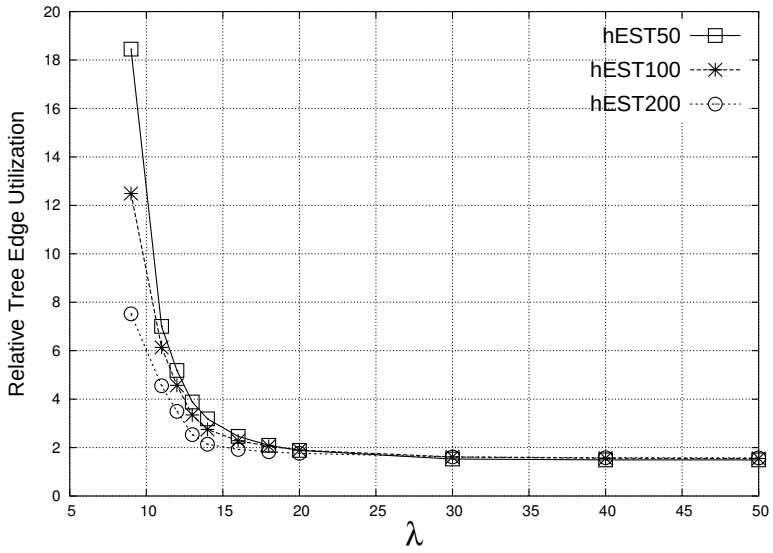


Figure 2: Relative Edge Utilization

$v_2 = sl(v_1)$.
Now, consider the subtree under node v_1, and note that the path labels of all nodes under v_1 have a common prefix defined by $\sigma(v_1)$. Therefore, we have, $\forall u \in descendent(v_1), \sigma(u) = x\alpha\beta$, where β is a non-empty substring of the indexed string. Similarly, $\forall u' \in descendent(v_2), \sigma(u') = \alpha\beta$. Further, if $u \in descendent(v_1)$ then, $\sigma(sl(u)) = \alpha\beta$. By definition of a suffix tree, there is only one path leading from root whose label is α and that is v_2. Hence the $sl(u)$ has to be under the subtree rooted at v_2. This completes the proof. □

In other words, if two nodes are related through a suffix link, then the *all* the nodes under the source of the suffix link have their suffix link targets *only* in the subtree of the target. This property gives us a way to reconcile between the edge and suffix link locality in the suffix tree.

5.1 Stellar Algorithm

The Stellar algorithm exploits the above derived property of suffix tree nodes to bring together the nodes in two subtrees whose root nodes are related through a suffix link. A pseudocode of Stellar algorithm is presented in Algorithm 2. The algorithm starts the suffix tree traversal at the root of the suffix tree, and recursively traverses the subtree below. When a node is visited, the suffix link target of the node is visited next, if not already visited through the tree edges. Thus an internal node and its suffix link target are treated as a "buddy" pair, and are scheduled for recursive traversal in sequence. This results in subtree under a node and the subtree under corresponding suffix link target to be recursively processed in succession – resulting in a large fraction of suffix links that span these two subtrees to be intra-page, in addition to the tree edges of each subtree. When enough nodes have been visited to fill a page, each node in the queue is scheduled for a separate recursive Stellar traversal, until all the nodes have been processed.

It is easy to observe that Stellar's complexity is linear in the size of the suffix tree being processed – a node is visited only once during the top-down traversal of the tree. Additionally, it

Algorithm 2 Stellar Algorithm

Stellar (r, B)

Input

r : Root of the subtree to be traversed

B : Capacity of the disk-page in terms of no. of nodes

Output

An ordering of the subtree under r

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

does not impose inordinate space overheads, as the only transient data structures required during the layout process are a queue of node ids, and a bit flag for each node of the tree indicating whether it has been visited or not. In our experiments we found that the queue never needs to hold ids of more than 15% of the nodes in the tree during the execution of Stellar.

In order to visually contrast the node clusterings produced by Stellar, SBFS and CO, consider the *intra-page connectivity* diagram of a suffix tree laid out using each of these algorithms, presented in Figure 3. These diagrams map the *intra-page tree edges* as solid lines and *intra-page suffix links* using dashed lines. The nodes of the suffix tree are presented in their order of distance from root, with their physical address as $< pageid : recid >$ pair within the oval corresponding to the node. The suffix tree presented here is built over a toy 100 basepair DNA sequence, with disk pagesize set to hold 5 nodes.

A visual inspection of these diagrams reveals that Stellar with 14 intra-page tree edges and 22 suffix links, generates tree layouts that exhibit better overall locality.

5.2 Level-wise Locality Variation

In addition to the overall locality of tree edges and suffix links obtained by the layout schemes, it is also critical to consider the distribution of such locality improvements in the suffix tree. If most of the locality gains are restricted only to a small portion of the tree that may not be accessed frequently by the search process then the effectiveness of locality improvements is significantly reduced.

Figure 4 illustrates the distribution of locality of tree edges and suffix links for the suffix tree over Human Chromosome 2 dataset under different layout schemes, including Stellar. These values represent the amount of local tree-edges (suffix links) at every level in the suffix-tree as a fraction of all the tree-edges (suffix links) present at that level.

As these graphs indicate, the tree-edge and suffix-link locality of all the three layouts are comparable at the top portion of the suffix tree. However, as the depth of the suffix-tree increases the suffix-link locality of CO layout outperforms SBFS significantly, while at the same time SBFS shows significantly better tree-edge locality over CO. On the other hand, the Stellar algorithm shows a steady locality comparable to the best within the tree-edge or suffix-link metric. In some cases, Stellar outperforms the suffix-link locality of CO layout (in the middle part of the suffix-tree).

5.3 Impact of Pagesize Variation

One of the important factors that could impact the locality property of layout algorithms is the size of the diskpage. With increasing pagesize one can hold more number of tree nodes within a page, which in turn could potentially result in more number of tree edges and suffix links local to the page. If such indeed is the case, then the performance of the layout can be easily improved by treating a set of consecutive pages as a logical set and applying buffering policies over this set.

Therefore, we evaluated the tree edge and suffix link localities of all the three layouts, with varying size of diskpage. The results are shown in Figure 5. As these graphs show, the superiority

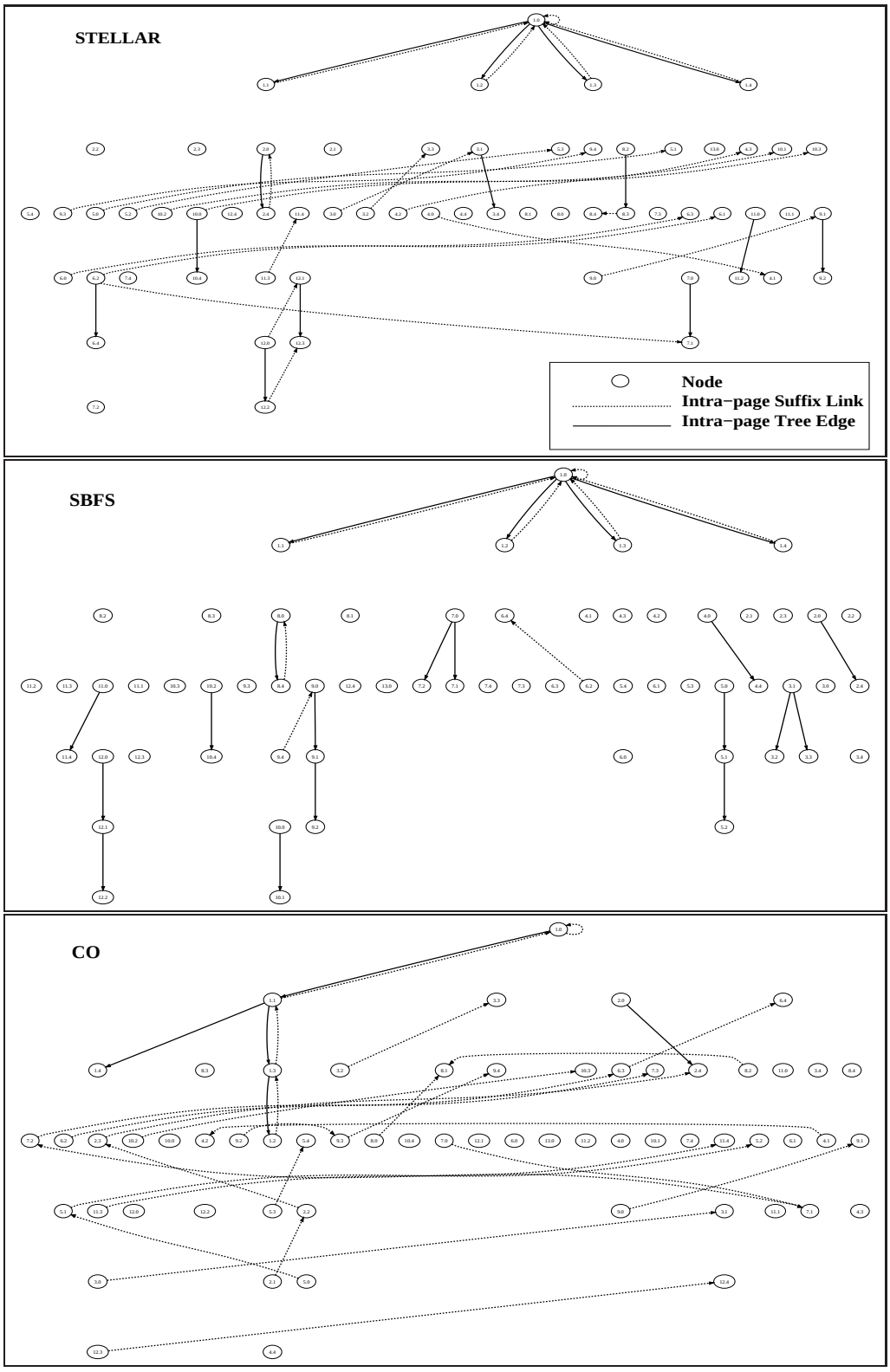
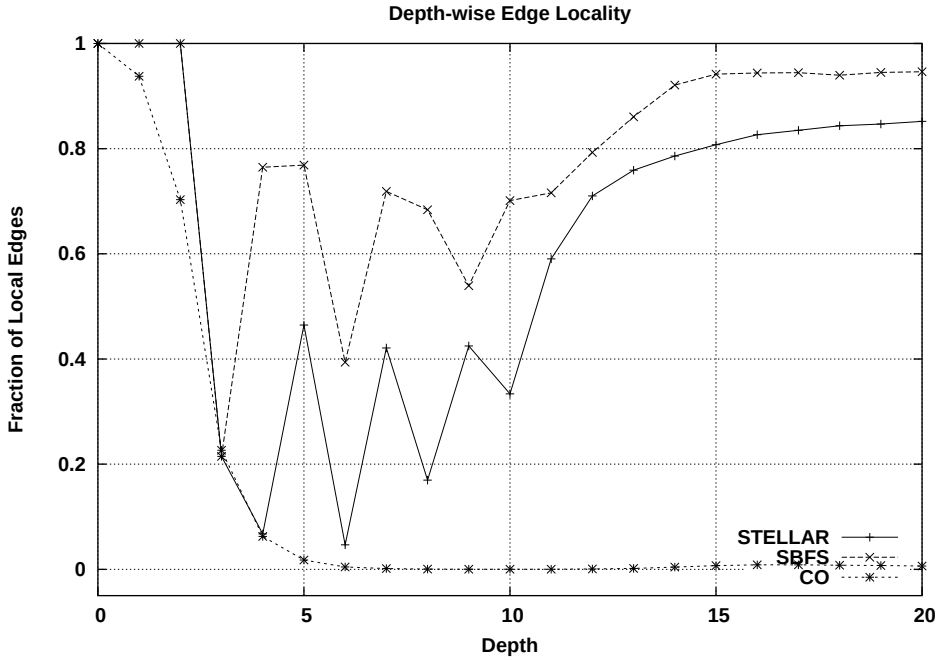
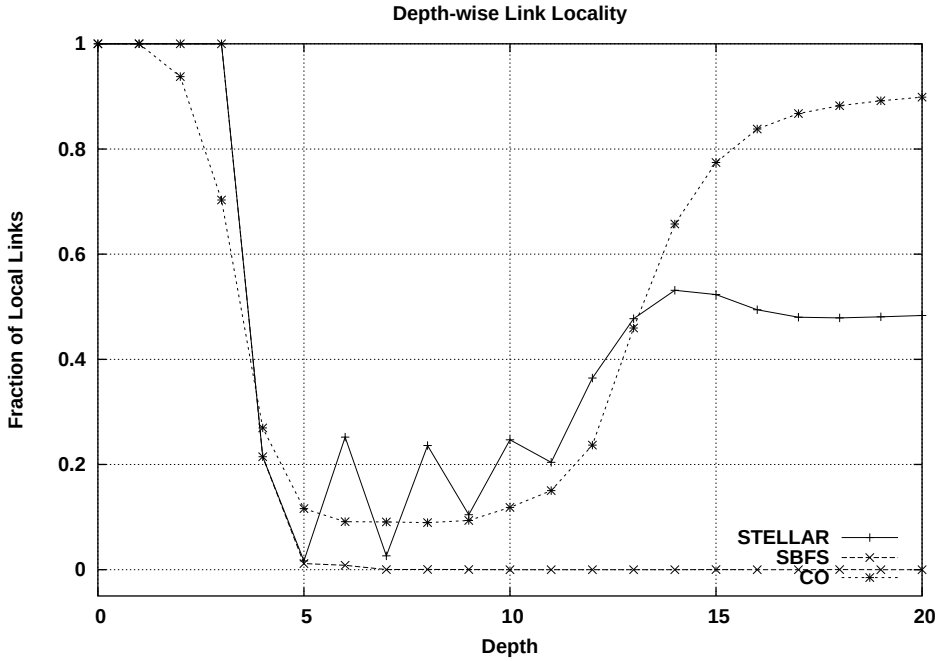


Figure 3: Intra-page Connectivity under Stellar, SBFS and CO Layouts



(a) Edge Locality



(b) Link Locality

Figure 4: Locality with varying Depth - Human Chromosome 2

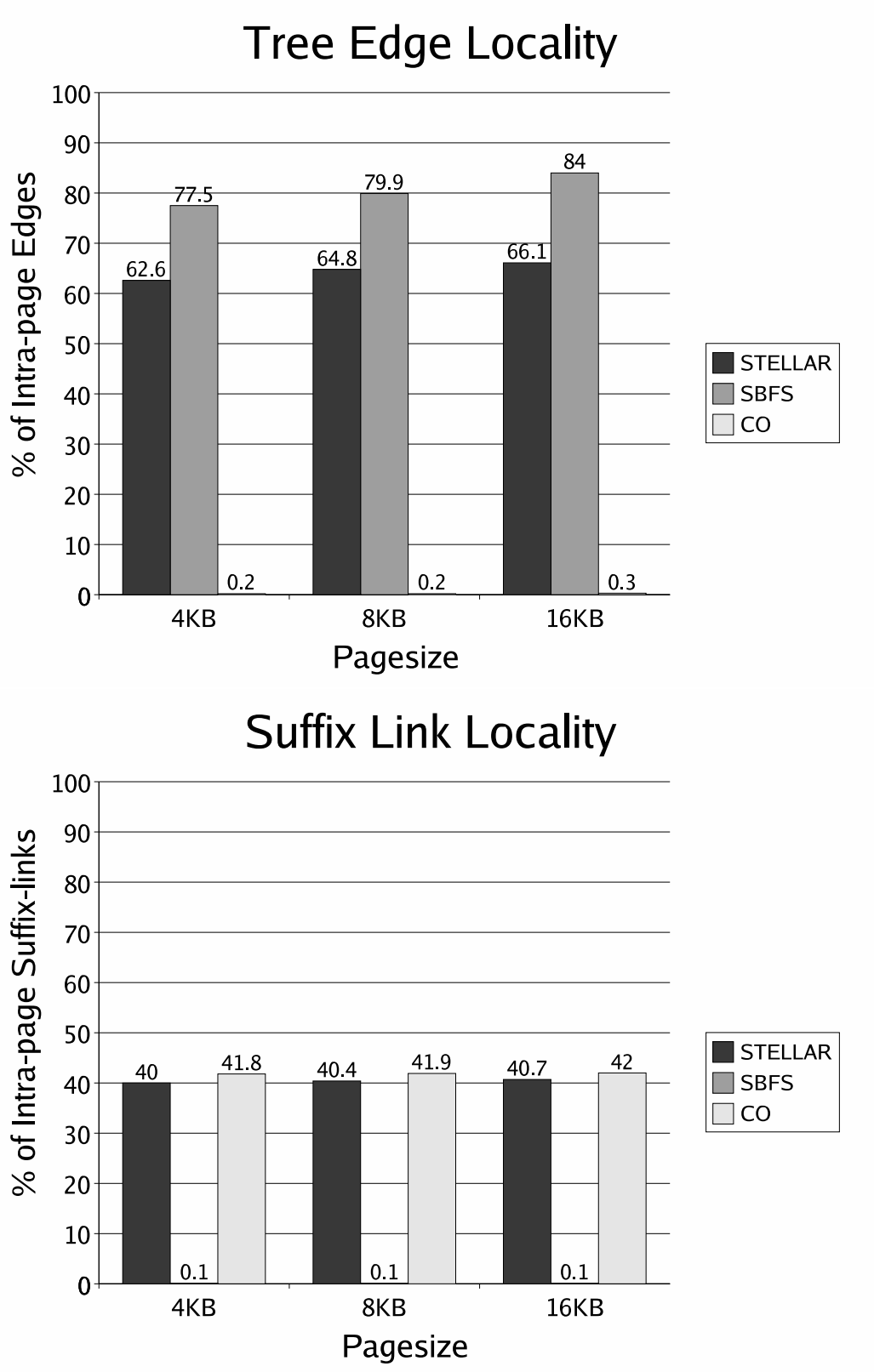


Figure 5: Locality with varying Pagesize

of Stellar in both forms of locality over other layout strategies is independent of variations in the pagesize.

6 Evaluation Framework

In this section, we present the experimental setup used for evaluation of different suffix tree layouts during maximal substring searches between genomic sequences.

In our evaluation, we used a variety of real-world DNA sequences available from GenBank [9] repository as the datasets to build disk-resident suffix trees.

6.1 Implementation Strategy

The suffix tree implementation used in our experiments is based on the efficient tree node representation proposed in our previous work [3].

Suffix tree nodes are densely packed into fixed size pages before they are committed to the disk. The pages on disk are either internal pages or leaf pages, depending on whether they store internal nodes or leaf nodes of the tree.

During construction, the storage of both internal nodes and leaf nodes is in their order of creation. Each page is committed immediately to the disk, as soon as all the space in the page is utilized. Once the suffix-tree is completely constructed, the resulting suffix-tree is traversed in the required storage order and the reordered suffix-tree is built during this post-construction process.

6.2 Query Collections

In order to evaluate the performance of search algorithms and the tree layout strategies presented so far, we need to pay attention to the following characteristics of the query sequences that have considerable impact on the search process.

Query length: The length of the query impacts the overall time for locating all the matching subsequences, as it directly determines the total number of iterations (number of suffix link traversals) during maximal substring location. In addition, increasing length of the query could also result in larger number of matches, increasing the overhead due to reporting of results.

Value of λ: As mentioned earlier, the maximal substring search algorithm takes as input a user-specified threshold λ, that serves as the lower-bound on the length of a match before all instances of the match are reported. The typical operational region of this parameter in genomic DNA sequence retrieval software is between 9 and 50, which is used in our experiments to demonstrate the utility of Stellar algorithm.

For DNA sequence searches, we used a collection of sequences from Expressed Sequence Tag (EST) database available from GenBank, as the base query collection. The Human-EST collection consists of 856,008 sequences with average length of each sequence being about 357.6

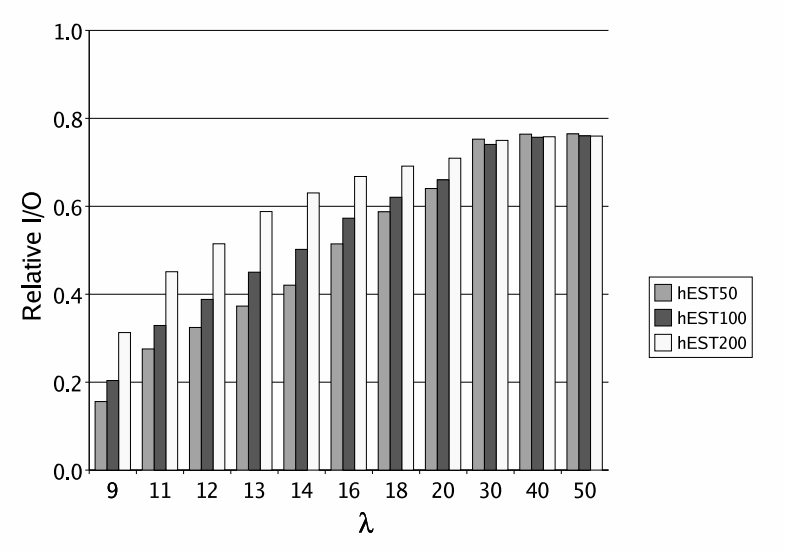


Figure 6: Performance of Stellar over CO

basepairs. The ESTs have been found to be extremely useful in high-throughput location of genes, genome mapping, etc., and form a key data collection in genomics research. Using this Human-EST collection, we generated length-restricted query collections of lengths 50, 100, and 200, by randomly sampling fixed-length sequences from each of these sequences. In order to remove any further bias in ordering of EST fragments generated, we randomly sampled 10,000 queries in each set to form three query collections, hEST50, hEST100 and hEST200.

7 Experimental Results

In this section, we present results of our empirical evaluation of various disk layout strategies for disk-resident suffix trees during maximal substring search task. A buffer pool of 8MB, which forms approximately 5% of the total size of the suffix tree, was used and managed using TOPQ [3], a buffering policy specifically designed for use with suffix trees.

7.1 Utility of Disk Layout

The relative performance of maximal substring search over disk-resident suffix tree laid out using Stellar against the CO layout is shown in Figure 6.

As these results indicate, Stellar layout results in a small fraction of the disk I/O performed during search, when compared to the I/O incurred suffix tree in CO layout. With increasing value of λ the performance differential between Stellar and CO reduces, but further investigation revealed that Stellar never incurs more than 76% of disk I/O than that of CO layout.

When λ values are in the lower end of operational spectrum, e.g. set to 9, the overall I/O cost of search is dominated by the overhead due to reporting of all results. As a result of this, Stellar layout with larger fraction of local tree edges clearly outperforms the CO layout which

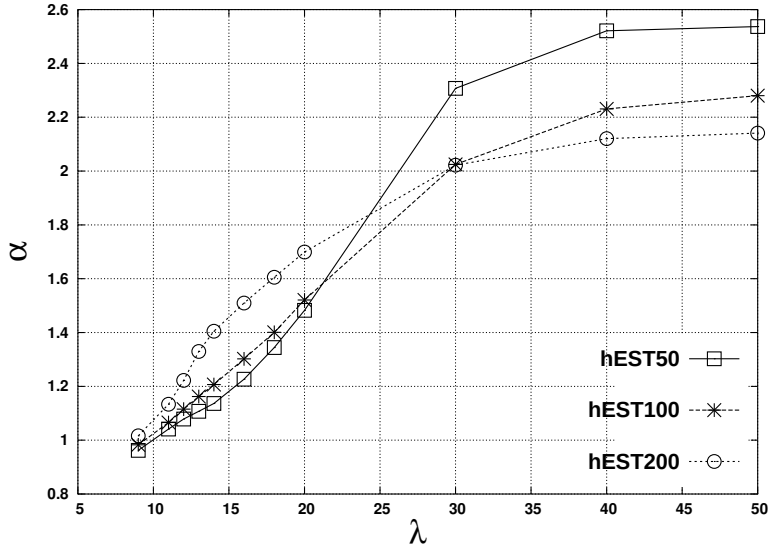


Figure 7: Relative Gains of Stellar over SBFS

practically provides no tree edge locality.

7.2 Performance of Stellar over SBFS

We now turn our attention towards comparison of disk I/O performance of suffix tree layout schemes of Stellar and SBFS. In order to provide a normalized measure of performance for both the disk layout strategies, we measure their *relative performance gains* over the base disk I/O cost of searching over the suffix tree in CO layout. Thus, the metric used for the performance comparison between SBFS and Stellar is as follows:

$$\alpha = \frac{I/O_{CO} - I/O_{Stellar}}{I/O_{CO} - I/O_{SBFS}} \tag{1}$$

where, I/O_x indicates the disk I/O cost of the maximal substring search algorithm over a suffix tree stored using x layout strategy.

Using the performance metric α defined in Equation 1 above, the relative performance of Stellar and SBFS with increasing values of λ is shown in Figure 7. As these graphs demonstrate, Stellar layout provides steadily increasing I/O gains with increasing values of λ. For example, at $\lambda = 11$, performance gain of Stellar over SBFS is close to 20%, which increases to more than 50% at $\lambda = 16$.

Note that, although it is not surprising to see that Stellar outperforms the layout produced by SBFS, it is interesting to see that it is possible to generate an efficient layout that trades off between tree-edge and suffix-link traversals.

7.3 Cardinality Evaluations

In many uses of suffix trees, it is enough to know the *cardinality* of matches rather than the identities of all the matches. For such uses of suffix trees, it is interesting to see the behavior of

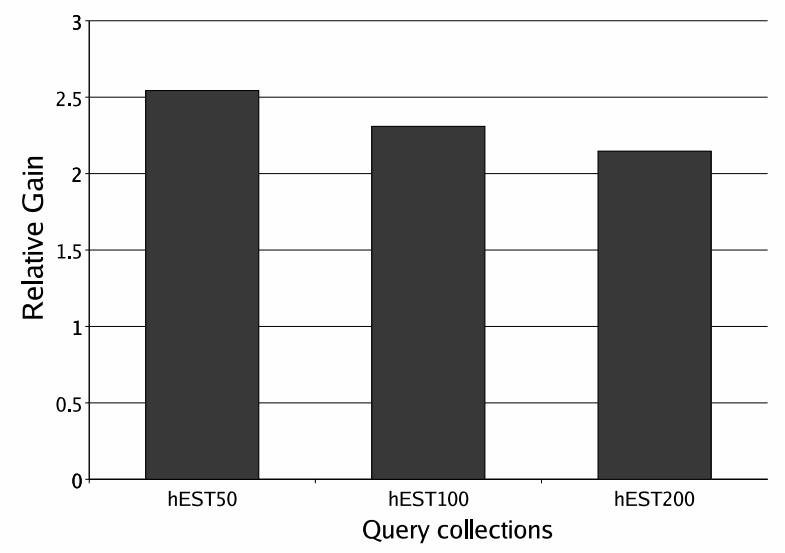


Figure 8: Cardinality Evaluation Performance

disk-layout strategies when the I/O cost associated with the result reporting phase is neglected. The important point here is that the performance of maximal substring search algorithm *without* the result reporting phase is independent of the value of λ – this lower bound is used only to *decide* if the subtree below has to be traversed to report all the matches.

Figure 8 shows the α values for the three EST query collections we have considered. These results demonstrate that the performance of Stellar is significantly better than SBFS – with more than 2-folds improvement in I/O gains, when subtree traversals are not needed to report all the result identities.

8 Search Performance over USTs

So far, we have studied the performance of maximal substring search over a disk-resident suffix-tree that provides suffix links to traverse across the tree efficiently. However, it has been previously suggested that presence of the suffix links in the tree results in poor performance of suffix-tree construction, and techniques for construction disk-resident suffix-trees completely dispensing with the suffix links have been proposed [15, 20, 22]. We call these suffix-trees as USTs (un-linked suffix-trees).

In this section, we first study the impact of disk layout on the search performance over USTs, and then compare their performance against disk-resident suffix-trees that have suffix links, stored using Stellar. These experiments were conducted after incorporating additional index space optimization by removing the 4-byte suffix link field in each internal node (bringing the indexing overhead to about 20.0 bytes per symbol). We used the disk-resident suffix-tree construction technique presented in [15], and the creation ordering of nodes generated during this construction comprises the baseline layout – similar to the CO-layout presented earlier.

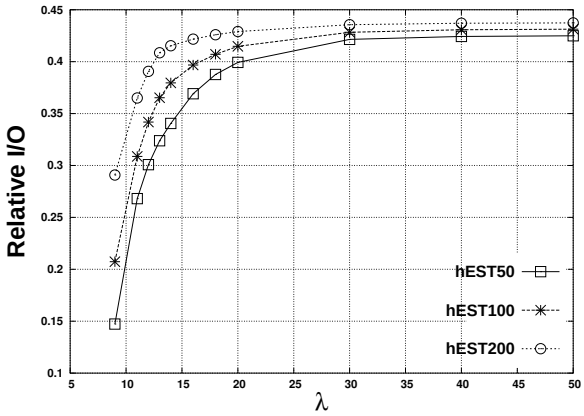


Figure 9: Gains due to SBFS over UST

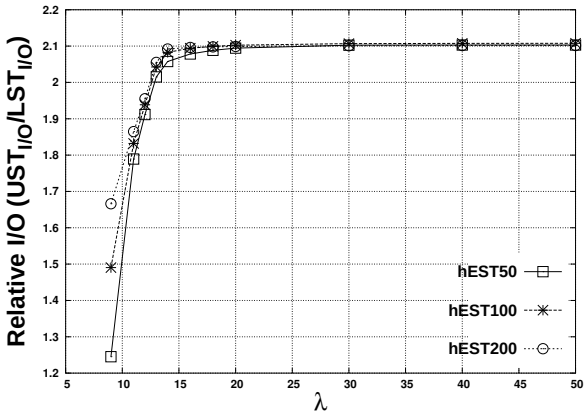


Figure 10: UST Vs. LST

8.1 Impact of Layout on UST Search Performance

In the absence of suffix links, the search performance of all the suffix-tree layouts generated by the disk-resident construction techniques of [15, 20, 22] is *lower bounded* by the SBFS layout strategy, as it optimizes the root-to-leaf traversals. Therefore, we compared the performance gains obtained for UST based maximal substring searches by the SBFS strategy over the storage order produced by construction algorithm of [15]. Figure 9 plots the impact of disk layout, with increasing values of λ, over UST built over Human Chromosome II dataset.

These results show that a careful layout of disk-resident USTs saves more than 50% of disk accesses, and in the commonly used values of λ – ranging from 9 to 15, the I/O savings are in the range of 60-70%.

8.2 Search Utility of Suffix Links

We now turn our attention to relative I/O performance of search tasks over disk-resident UST and suffix-tree with suffix links (i.e., LSTs – linked suffix-trees), thereby quantifying the search utility of suffix links. We compare the performance of UST laid out using SBFS strategy against the performance of LST laid out using Stellar layout strategy. Note that these layout strategies are optimized for the respective structural variants of suffix-trees.

Figure 10 presents the relative I/O incurred due to searching with UST as opposed to searching with LST, with increasing values of λ. As these graphs illustrate, searching over LST clearly provides distinct advantages over performing the same task with UST. Despite the superior space economy of USTs, it incurs more than 70% extra disk reads compared to LSTs. As the value of λ increases, the performance gap widens – at λ set to 20, UST incurs more than *2 times* the disk I/O than LST. These results show the need to retain suffix links in the suffix-trees, contrary to the disk-resident suffix-tree construction and maintenance recommended in [15, 14, 20, 22].

Dataset	Storage	Suffix-Links	Tree Edges
SPROT	CO	49.2%	0.2%
	SBFS	0.1%	56.1%
	Stellar	**31.6%**	**49.6%**

Table 2: Locality with Protein Dataset

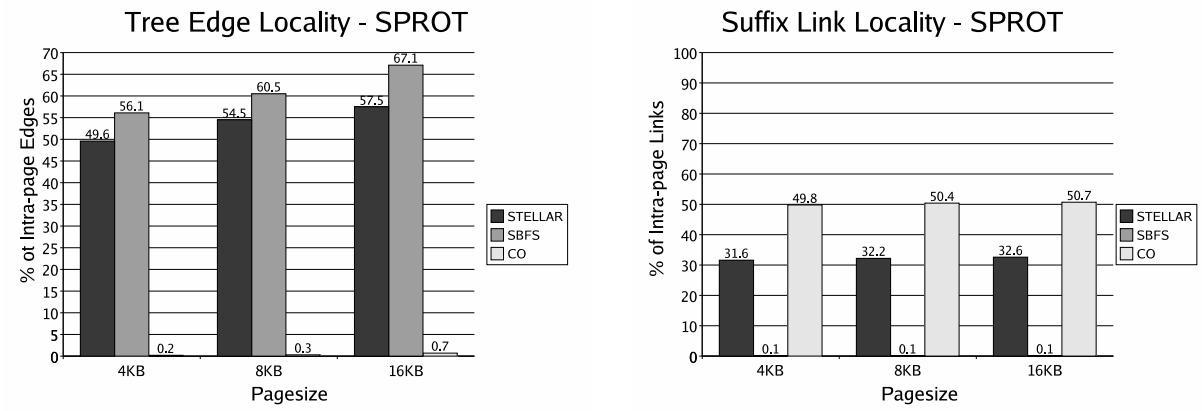


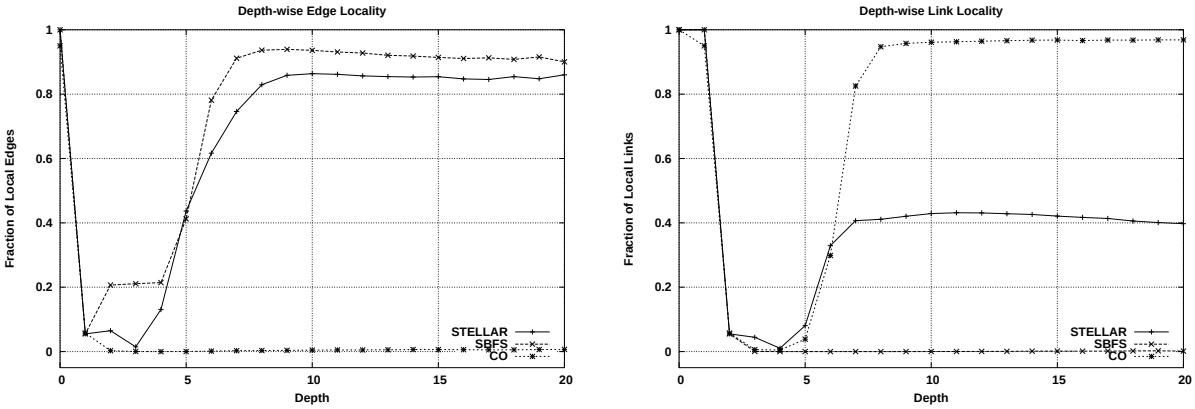
Figure 11: Locality with varying Pagesize - SPROT

9 Experiments with Protein Data

Now we turn our attention to the performance of Stellar and other tree layout algorithms for suffix trees over Protein sequences. We use a 25×10^6 length amino-acid sequence dataset, SPROT, derived from SwissPROT collection of protein sequences [21]. Since most of the individual proteins sequences are short, the SPROT dataset was generated by concatenating the sequences together.

The locality results for the suffix tree over SPROT dataset, under different disk layout strategies is shown in Table 2. As these number demonstrate, Stellar shows locality profile similar to that was shown in the case of DNA sequences earlier. Similarly, the variation of the locality with the size of the diskpage is illustrated in Figure 11. The graphs in Figure 12 illustrate the depth-wise distribution of the locality profile throughout the suffix tree. In contrast to the similar graphs for DNA sequences in Figure 4 earlier, the locality distribution displays a sharp variation at depths ranging from 0 to 5.

It should be noted that due to the choice of array-based representation of the suffix tree, the size of the internal node is significantly larger (93 bytes as opposed to 29 bytes for DNA), leading to much smaller packing density of nodes. This results in lower values of locality since there is less scope for packing the nodes involved in the tree-edge (suffix link) in the same page.



(a) Edge Locality (b) Link Locality

Figure 12: Locality with varying Depth - SPROT

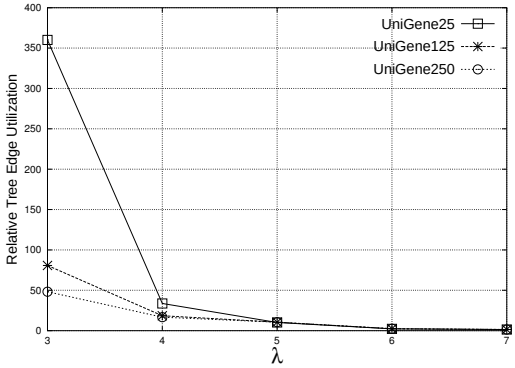


Figure 13: Relative Edge Utilization - SPROT

9.1 Protein Substring Searches

For the protein sequence searches, a amino-acid query set of 10,000 randomly sampled sequences from translated human UniGene non-redundant set of gene-oriented clusters was chosen [26].

Although the common search task over protein sequence database is maximal substring location, the parameters used in these search tasks differ significantly from those used in DNA database search, to account for the larger alphabet. Specifically, the λ values are much smaller – the BLASTP package uses default value of 3. As a result, the utilization of suffix links during the search process is significantly less, and the search cost is dominated by the use of tree edges for reporting of results.

The relative utilization of tree edges with respect to suffix links during the substring search task is illustrated in Figure 13. As these graphs indicate the tree edge utility is significantly higher in the *typical operating range* of λ values. Thus, higher tree-edge locality is beneficial in the case of protein datasets.

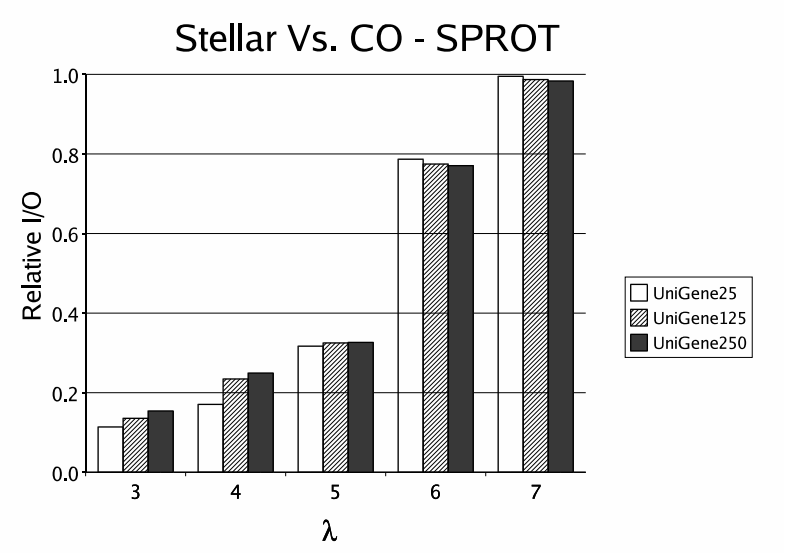


Figure 14: Performance of Stellar over CO - SPROT

Impact of Disk Layout The relative performance of maximal substring search over disk-resident suffix tree on SPROT dataset, laid out using Stellar against the CO layout is shown in Figure 14.

Stellar Vs. SBFS As we already pointed out, the suffix-link locality is less critical in the operating range of λ values for protein substring similarity search. Thus, it seems natural to expect SBFS which localizes the tree-edges only to perform superior to Stellar which clusters suffix-links as well.

The relative gains of Stellar over SBFS are shown in Figure 15. As these results indicate, the performance of Stellar is comparably close to that of SBFS even during the adverse value-range of λ. With increasing λ values, the Stellar improves over SBFS.

10 Related Work

In this section, we briefly summarize the earlier work in the area of disk layout schemes for *skewed* search trees.

There has been a growing interest in considering the layout of various data-structures ranging from binary search trees [23], general index trees and DAG structures [7, 11], and some special graph structures [19]. Recently, there is a surge of interest in the data structures for multi-level memory hierarchies, with minimal knowledge about the block and memory-level sizes – these are called *cache oblivious data structures*. A recent cache-oblivious tree layout algorithm was proposed in [1], which extends the results of Gil and Itai [11] to work in the context of multi-level memory hierarchies. We refer the reader to the above paper for further references in this area.

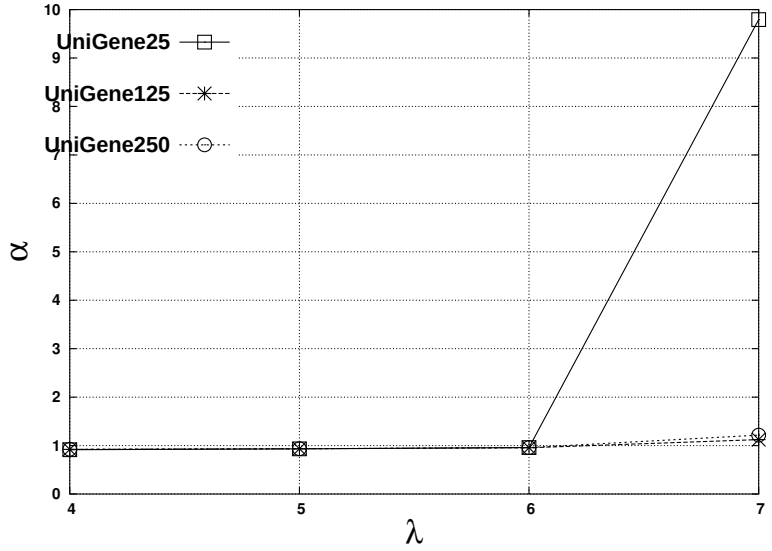


Figure 15: Relative Gains of Stellar over SBFS - SPROT

The earliest work in this area we are aware of is that of Diwan et al. [7]. They provide algorithms for minimization of external path length (defined as the number of edges to be traversed in a root to leaf path) in digital tree structures such as trie, kdb-trees etc., given a uniform access probability over leaf nodes of the tree. The worst-case external path length minimization is achieved through a $O(N)$ algorithm that does a bottom-up packing of nodes into pages. However, for average path-length minimization problem (which is more relevant with a large number of queries being posed over the search tree), they have a much more complex, dynamic programming based algorithm that has $O(kN^2)$ time complexity, where k is the page capacity and N is the number of nodes in the tree. Both these algorithms are shown to be optimal. However, empirical results in [7] indicate that a simpler technique of SBFS yields the same average path-lengths in most of the cases.

Another related work is [11], where they consider the issue of packing trees efficiently given the "*access weights*" associated with each node in the tree and provide optimal dynamic programming based packing algorithm. However, their algorithm is $O(BN^2)$, where B is the page size (in terms of no. of nodes), and N is the number of nodes in the tree. Although their techniques are applicable in the suffix tree packing as well, the quadratic costs associated with their algorithm makes it seem impractical in the case of suffix trees. Additionally, the algorithms of [7] and [11] do not provide good page utilization (both guarantee only 50% utilization). In the case of disk-resident suffix trees, already burdened with significant space overheads, it is impractical to resort to techniques that place additional demands on space. Therefore, we focus on achieving better disk layout schemes that guarantee 100% space utilization.

11 Conclusions and Future Work

Developing suffix trees as a disk-resident sequence index structure has been an active area in recent times, and many techniques have been proposed to significantly improve the construction time. However, there has been virtually no research on evaluating and optimizing the search performance of these disk-resident suffix trees, the topic we have addressed here in detail.

Specifically, we have evaluated the impact of suffix tree layout on disk on the I/O performance of common genomic search tasks, and shown through detailed empirical evidence that existing index layout algorithms are not effective for storing suffix trees on disk. The layouts produced through these algorithms provide locality for only one of the two traversal paths used during suffix tree searches, and practically zero locality for the other path.

Addressing this unsatisfactory state of affairs in disk-resident suffix tree layouts, we presented a layout strategy called Stellar, that optimizes the locality feature of both tree edges and suffix links in the suffix tree. The layouts produced by Stellar show more than 40% suffix link locality, and close to 65% tree edge locality, thus combining the strengths of the two extreme layout schemes considered before.

Using real genomic DNA sequences drawn from GenBank repository, and querysets from Human-EST collection, we showed that Stellar incurs only about 30-40% of the disk I/O incurred by a suffix tree stored in its creation order. Even in extreme cases, more than 25% disk costs are saved by laying out the suffix tree through Stellar. Furthermore, Stellar shows almost *2-fold improvement* over SBFS index layout strategy in terms of disk I/O saved. The relative performance of Stellar significantly improves with increasing values of λ (the minimum match length), thus highlighting the applicability of Stellar in full-genome alignment software such as MUMmer, where values of λ are typically in the range 20-50.

As part of our future research, we plan to extend the work presented in this paper, by designing a composite packing strategy that combines the advantages offered by Stellar during the search phase, and the performance of SBFS for the result reporting phase. Another research direction we intend to follow is to investigate if the reordering of disk-resident suffix trees can be achieved *during* their construction itself, and avoid the post-construction step altogether. We also plan to implement these disk layout strategies within a database system, and evaluate the performance gains over a richer variety of datasets.

References

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

[2] S. Altschul, W. Gish, W. Miller, E. W. Myers, and D. Lipman. A Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3), 1990.

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

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

[5] A. L. Cobbs. Fast Approximate Matching using Suffix Trees. In *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, 1995.

[6] 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.

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

[8] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the Memory Bottleneck in Suffix Tree Construction. In *Proceedings of the IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, 1998.

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

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

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

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

[13] D. Gusfield. Suffix Trees Come of Age in Bioinformatics (Invited Talk). In *IEEE Bioinformatics Conference (CSB)*, 2002.

[14] E. Hunt, M. Atkinson, and R. Irving. Database Indexing for Large DNA and Protein Sequence Collections. *VLDB Journal*, 7(3), 2001.

[15] E. Hunt, M. P. Atkinson, and R. W. Irving. A Database Index to Large Biological Sequences. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[16] R. Japp. First Year Report. Master's thesis, University of Glasgow, July 2001.

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

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

[19] Mark Nodine, Michael Goodrich, and Jeffrey Vitter. Blocking for External Graph Searching. In *Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, 1993.

[20] K.-B. Schürman and J. Stoye. Suffix Tree Construction and Storage with Limited Main Memory. Technical Report 2003-06, Universität Bielefeld, 2003.

[21] SWISS-PROT Protein Knowledgebase. http://www.expasy.org/sprot/.

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

[23] Sripad Thite. Optimum Binary Search Trees on the Hierarchical Memory Model. Master's thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2001.

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

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

[26] UniGene: Organized view of the transcriptome. ftp://ftp.ncbi.nih.gov/repository/UniGene/.

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