Search-Optimized Suffix-Tree Storage for Biological Applications

Srikanta J. Bedathur and Jayant R. Haritsa

Database Systems Lab, SERC,
Indian Institute of Science,
Bangalore 560012, India
{srikanta, haritsa}@dsl.serc.iisc.ernet.in

Abstract. Suffix-trees are popular indexing structures for various sequence processing problems in biological data management. We investigate here the possibility of enhancing the search efficiency of *disk-resident* suffix-trees through customized layouts of tree-nodes to disk-pages. Specifically, we propose a new layout strategy, called *Stellar*, that provides significantly improved search performance on a representative set of real genomic sequences. Further, Stellar supports both the standard root-to-leaf lookup queries as well as sophisticated sequence-search algorithms that exploit the suffix-links of suffix-trees. Our results are encouraging with regard to the ultimate objective of seamlessly integrating sequence processing in database engines.

1 Introduction

The *suffix-tree* is a highly popular mechanism for indexing exponentially growing biological sequence repositories [12,13]. Its appeal lies in its *linear* (in the size of the sequence) time and space complexity of construction, and its *linear* (in the size of the query) search complexity. A unique aspect of suffix-trees is that, unlike traditional database indexes whose size is typically a fraction of the database contents, their size is usually much 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 memory (with each DNA symbol represented with 2-bits), whereas the corresponding most space-economical suffix-tree occupies close to 25 GB. That is, it is often straightforward to host the sequence data in main memory, but the suffix-tree itself needs to be disk-resident.

This piquant size 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. Accordingly, there has been significant recent research activity to address this problem and design high-performance disk-resident suffix-trees [5,15,18,19]. However, these efforts have mainly focused on the *construction aspect*, that is, on how to build the tree efficiently on disk.[1] In this paper, we take the next logical step of exploring the *search aspect*, and investigating the

[1] Search performance is reported in [14,18], but not analyzed in detail.

associated efficiency concerns. 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, such that the search efficiency is improved. While layout strategies have been well-studied for a variety of data-structures [1,3,10,11,17,20], we are not aware of any work focusing 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 than those found in traditional index structures, since both tree-edges and special lateral connectors called suffix-links are involved.
– The presence of suffix-links turns suffix-trees into *cyclic* structures.
– Suffix-trees are not inherently balanced, unlike typical disk-resident index structures (e.g. B^+-trees).

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 the subsequent matches are found by following the suffix-links [7,9].

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 relationships between the suffix-links and the tree-edges under associated subtrees, to achieve high locality for both suffix-links and tree-edges. We quantify its effectiveness with a detailed performance study on a variety of real genomic sequences.

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 new suffix-tree layout that optimizes both kinds of traversals, thereby providing significantly improved search performance.

2 Sequence Search Using 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, while the directed dashed lines indicate *suffix-links*. The links play an important role in linear time construction of suffix-trees [16,22], and also in many search algorithms over suffix-trees [7,8,21]. Table 1 summarizes the terminology associated with suffix-trees used in the rest of the paper.

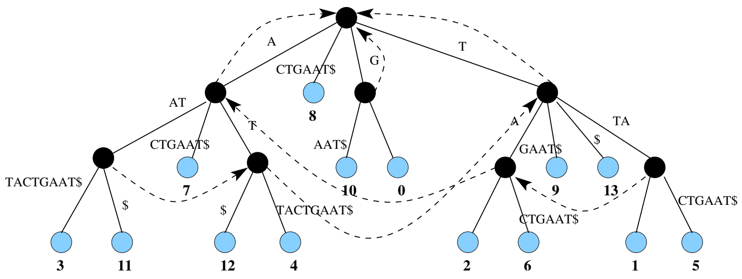


Fig. 1. Suffix-tree for the DNA fragment GTTAATTACTGAAT$

Table 1. Notation

S	Sequence of length n
Σ	Finite alphabet of symbols
\$	Delimiter symbol such that $\$ \notin \Sigma$
$S[i]$	Symbol at position i in S, drawn from Σ
$S[i \ldots j]$	Substring of S starting at position i and length $(j - i + 1)$
S_i	Suffix of the sequence S starting at position i
$sl(v)$	Suffix-link starting from the internal node v

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 locations, and so on. A particularly critical use of suffix-trees is in pre-processing a large genomics data repository and subsequently utilizing the index to efficiently answer 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 matching substrings are then used to generate *local alignments*, the regions of similarity between the sequences, through the use of various domain-specific heuristics.

In this paper, we use the *maximal common-substring search*, proposed in [7], as a representative search task over disk-resident suffix-trees. This task is defined as follows:

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 (that is, $j \geq \lambda$).* □

3 Suffix-Tree Layout

Suffix-trees, unlike popular index structures such as B-Trees [4], are not inherently balanced – their structure depends entirely on the combinatorial characteristics of the indexed sequence. Consider, for example, the suffix-tree shown in Figure 1 – here, leaf-node 8 is an immediate child of the root, whereas leaf-node 1 is at depth 3. In the worst-case, the tree can degenerate into a linear chain of internal nodes.

The fan-out of each internal node of a suffix-tree is upper-bounded by the size of the alphabet of the indexed sequence. Therefore, the common strategy of customizing the fanout to suit the disk-page size cannot be adopted here. This means that multiple nodes of a suffix-tree will be stored on a page, with nodes connected both within as well as across pages – it therefore becomes critical to choose the nodes that will be placed in the same disk-page in order to minimize the disk I/O cost incurred during search.

Earlier research on the layouts of disk-resident indexes [10] has considered the problem of packing trees in order to minimize the total disk accesses given a access distribution on the leaf nodes – that is, *average path-length minimization*, following the terminology of [10]. It has been shown that a heuristic-based linear-time algorithm, henceforth called SBFS, that does recursive localized breadth-first layout of the tree, not only outperforms classic tree-layout methods such as Breadth-first and Depth-first strategies, but also results in an I/O-cost that is within a small factor of an *optimal* quadratic-time layout algorithm.

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 visit-order 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.

3.1 Issues in Suffix-Tree Layout

The general problem of optimal graph layout is known to be NP-complete [11]. Even from a heuristic viewpoint, the storage layout of disk-resident suffix-trees introduces a variety of novel issues:

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

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

Due to these complexities, previously proposed layout strategies that are designed to work with either tree or DAG structures are not directly applicable in the context of suffix-trees. Nevertheless, to serve as a comparative yardstick, we investigate the efficacy of the SBFS strategy outlined above for laying out a suffix-tree on disk, by *ignoring* the suffix-links during the layout process.

3.2 Comparing the Quality of Layouts

The overall metric we use to evaluate the quality of layouts obtained using different storage strategies is to execute a representative set of queries over the suffix-trees laid out using these strategies and measure the number of disk accesses incurred. To gain more insight into the observed behavior, we also additionally measure the percentage of tree-edges and suffix-links whose source and target are both present in the same disk page – that is, the *structural localities* of the suffix tree layouts, discussed next.

Table 2 presents the structural locality results for suffix-trees built on a representative 25 Mbp long sequence drawn from Human Chromosome 2, hereafter referred to as HC2/25, with disk pagesize set to 4KB. The storage layouts evaluated here are: (1) **CO** (Creation Order), which corresponds to ordering the nodes as they are created during the construction (Ukkonen's construction algorithm [22] was used here); (2) **SBFS** layout discussed earlier; and (3) our new **Stellar** layout, described in detail in the next section.

Table 2. Structural Edge and Link Localities

Dataset	Storage	Suffix-Links	Tree Edges
Human Chromosome 2	CO	41.8%	0.2%
	SBFS	0.1%	77.5%
	Stellar	**40.0%**	**62.6%**

From the results, we first see that the CO-layout provides practically *no tree-edge locality* – only 0.2% of tree-edges are intra-page, while suffix-link locality is comparatively high – 42%. The SBFS-layout, on the other hand, represents the opposite extreme in structural locality, with 75-80% of tree-edges being intra-page, but less than 0.1% of suffix-links being local! Overall, these results indicate, as also confirmed by our other experiments, that the CO and SBFS layouts represent (negative) extremes in suffix-tree layout. The reasons for this behavior are explained in the extended version of this paper [6].

Finally, note that the structural localities for the Stellar layout in Table 2 indicate that its suffix-link locality (**40.0%**) is close to that of CO, while its tree-edge locality (**62.6%**) is comparable to that of SBFS – clearly *simultaneously* optimizing the locality of both connectors.

4 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. This relationship can be derived easily from well-known structural properties of suffix-trees [12]. Specifically, the property we use 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 nodes under v_2.

In other words, if two nodes are related through a suffix-link, then *all* the nodes under the source of this suffix-link have their suffix-link targets *only* in the subtree of the

```
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 suffix-tree under $r$

    $queue \longleftarrow r$; {push root into the BFS queue}
    $nodecount \leftarrow 0$; {initialize the counter}
    while $queue$ not $\emptyset$ do
      $r' \longleftarrow queue$; {remove head of the $queue$}
      if $r'$ not visited then
        mark $r'$ as visited and increment $nodecount$;
      for all $c$ such that $c$ is a child of $r'$ do
        $s \leftarrow sl(c)$;{$s$ is the suffix-link of $c$}
        if $c$ not visited AND $nodecount < B$ then
          mark $c$ as visited and increment $nodecount$;
        $queue \longleftarrow c$;
        if $s$ not visited AND $nodecount < B$ then
          mark $s$ as visited and increment $nodecount$;
        $queue \longleftarrow s$;
      if $nodecount \geq B$ then
        while $queue$ not $\emptyset$ do
          $m \longleftarrow queue$;
          Stellar($m$,$B$);
```

Fig. 2. Stellar Algorithm

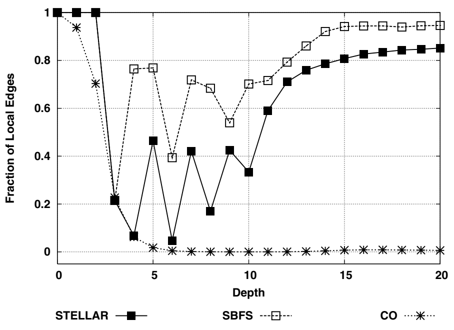
target. This property gives us a way to reconcile between the tree-edge and suffix-link localities in the suffix-tree.

The pseudocode of the Stellar algorithm, utilizing the above structural relationship, is presented in Figure 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 the subtree under a node and the subtree under the 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.

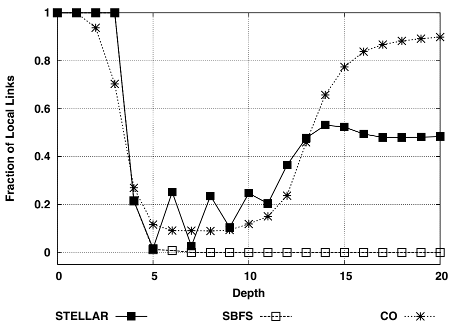
It is easy to observe that Stellar's complexity is linear in the size of the suffix-tree being processed – a node is visited only once during the top-down traversal of the tree. Additionally, it does not impose inordinate space overheads, as the only transient data structures required during the layout 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 100 nodes, even over DNA sequences exceeding 25Mbp.

4.1 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.



(a) Edge Locality



(b) Link Locality

Fig. 3. Depth-based Structural Localities

Figures 3(a) and (b) illustrate the locality distributions of tree-edges and suffix-links for the suffix-tree over the HC2/25 sequence under the three layout schemes. These values represent the number of intra-page tree-edges (resp. suffix-links) at every level in the suffix-tree as a fraction of all the tree-edges (resp. suffix-links) going out from that level. For example, there are a total of 2,417,879 outgoing edges from level 10, of which approximately 40% become intra-page under a Stellar layout.

As these graphs indicate, the tree-edge and suffix-link locality of all 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 locality metric. In the middle portion of the suffix-tree, due to the large number of tree nodes, the locality fraction (of both suffix-links as well as tree-edges) is lower than in the top and bottom parts of the tree under all the layouts.

While the above graphs were obtained with a pagesize of 4 KB, our experiments with larger page sizes such as 16 KB, also showed similar trends – details in [6].

5 Experimental Framework and Results

We now present the disk I/O results for evaluating the Maximal Common-substring Search query described in Section 2 for suffix-trees built over the HC2/25 sequence. Results over other datasets, including Protein sequence data, are available in [6].

Our suffix-tree implementation is based on an efficient array-based tree node representation suggested in [5], with 22.5 bytes per symbol. The disk page-size is set to 4KB – a typical value in most systems. A buffer pool of 8MB, which forms approximately 5% of the total index size, was used and managed using TOP-Q [5], a buffering policy designed for use with disk-resident suffix-trees.

5.1 Query Workload

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

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

Value of λ: The user-specified threshold, λ, serves as the lower-bound on the length of the match before all instances of the match are reported. The typical operational range of this parameter in a variety of DNA sequence retrieval software is between 9 and 50. Specifically, BLAST [2] uses a default value of 11 while MUMmer [9] sets it to 50.

We generated our query workload based on a collection of sequences from Expressed Sequence Tag (EST) database of GenBank. The EST-database contains 856,008 sequences with average sequence length of 357.6 basepairs. Using this base collection, we generated 3 length-restricted query collections, with lengths 50, 100, and 200, by randomly sampling fixed-length subsequences from each entry of the EST-database. In order to remove any remaining bias in the ordering of EST fragments, we sampled 10,000 sequences from each length-restricted query set to form three query collections, **hEST50**, **hEST100** and **hEST200**, used in our evaluation.

5.2 Utility of Disk Layout

The relative performance of maximal substring search over disk-resident suffix-tree laid out using Stellar, normalized to that with the CO layout, is shown in Figure 4. As these results indicate, the Stellar layout results in substantially reduced search costs as compared to CO. For example, at $\lambda = 11$, Stellar requires only 30–45% of the disk I/Os incurred by CO. Although this performance differential reduces with increasing value of λ, Stellar never incurs more than 75% of CO's disk accesses.

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 producing a large result set. As a result, the Stellar layout, with its larger fraction of intra-page tree-edges, clearly outperforms the CO layout which provides very little tree-edge locality.

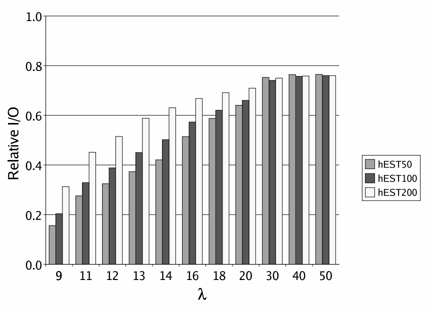


Fig. 4. Stellar Vs. CO

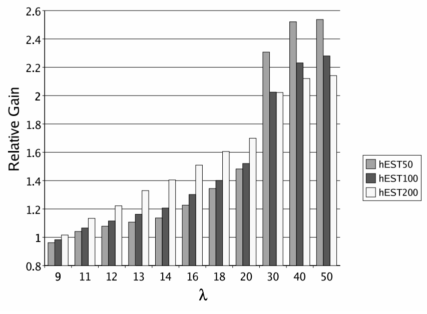


Fig. 5. Stellar Vs. SBFS

5.3 Relative Performance of Stellar and SBFS

We now turn our attention towards comparing the disk costs of Stellar and SBFS. In order to provide a normalized measure of performance, we measure their *relative performance gains* over that of the baseline CO layout. These statistics are shown in Figure 5, as a function of λ, and demonstrate that Stellar provides steadily increasing I/O gains with increasing values of λ. For example, at $\lambda = 11$, the performance gain of Stellar over SBFS is close to 20%, which increases to more than 50% at $\lambda = 16$.

In addition to these results, we also performed experiments to show that the suffix-link based searching over Stellar layouts require less than 50% disk I/O as compared to that required for searching *without suffix-links* over SBFS layouts. Details of these experiments are available in [6]. Note that the search performance of disk-resident suffix-trees constructed by the techniques of [15,18,19] is lower-bounded by the performance of the SBFS layout. As a consequence, Stellar-organized suffix-trees outperform the storage organizations produced by all these prior techniques.

6 Conclusions

Developing suffix-trees as a disk-resident sequence index structure has been an active research 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 addressed in this paper.

Specifically, we have evaluated the impact of the suffix-tree's disk layout on the I/O performance of common genomic search tasks, and shown through detailed empirical evidence that existing index layouts, such as Creation-Order (CO) and SBFS, are not effective. They provide locality for only one of the two traversal paths, tree-edges and suffix-links, used during suffix-tree searches, and practically zero locality for the other path.

To address this unsatisfactory state of affairs, we presented a layout strategy called Stellar that optimizes the locality of both tree-edges and suffix-links in the suffix-tree.

The layouts produced by Stellar show close to 40% suffix-link locality, and 60% tree-edge locality, providing an all-round performance that is comparable to the individual best performances.

Using real genomic DNA sequences drawn from the GenBank repository, and query-sets from the Human-EST collection, we showed that Stellar typically incurs only about 30-40% of the disk I/O incurred by a suffix-tree stored in creation order. Even in extreme cases, more than 25% disk costs are saved by Stellar. Furthermore, it provides close to *2-fold improvement* over the SBFS layout 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.

References

1. S. Alstrup et al. Efficient tree layout in a multilevel memory hierarchy. Technical Report arXiv:cs.DS/0211010v1, 2002.
2. S. Altschul et al. A Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3), 1990.
3. S. Baswana and S. Sen. Planar Graph Blocking for External Searching. *Algorithmica*, 34(3), 2002.
4. R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3), 1972.
5. S. Bedathur and J. Haritsa. Engineering a Fast Online Persistent Suffix Tree Construction. In *Proc. of the IEEE Intl. Conf. on Data Engg. (ICDE)*, 2004.
6. S. Bedathur and J. Haritsa. Search-Optimized Persistent Suffix-tree Storage for Biological Applications. Technical Report TR-2004-04, Database Systems Lab, Indian Institute of Science, 2004.
7. W. I. Chang and E. L. Lawler. Approximate String Matching in Sublinear Expected Time. In *Proc. of the IEEE Symp. on Found. of Comp. Sci. (FOCS)*, 1990.
8. A. L. Cobbs. Fast Approximate Matching using Suffix Trees. In *Proc. of the 6th Annual Symp. on Combinatorial Pattern Matching (CPM)*, 1995.
9. A. L. Delcher et al. Alignment of Whole Genomes. *Nucleic Acids Research*, 27(11), 1999.
10. A. A. Diwan et al. Clustering Techniques for Minimizing External Path Length. In *Proc. of the 22nd Intl. Conf. on Very Large Databases (VLDB)*, 1996.
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. P. Atkinson, and R. W. 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 *Proc. of the 27th Intl. Conf. on Very Large Databases (VLDB)*, 2001.
16. E. M. McCreight. A Space-Efficient Suffix Tree Construction Algorithm. *Jl. of the ACM (JACM)*, 23(2), 1976.
17. M. Nodine, M. Goodrich, and J. Vitter. Blocking for External Graph Searching. In *Proc. of the 12th ACM Symp. on Principles of Database Systems (PODS)*, 1993.

18. K.-B. Schürman and J. Stoye. Suffix Tree Construction and Storage with Limited Main Memory. Technical Report 2003-06, Universität Bielefeld, 2003.
19. S. Tata, R. A. Hankins, and J. M. Patel. Practical Suffix Tree Construction. In *Proc. of the 30th Intl. Conf. on Very Large Databases (VLDB)*, 2004.
20. S. Thite. Optimum Binary Search Trees on the Hierarchical Memory Model. Master's thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 2001.
21. E. Ukkonen. Approximate String Matching over Suffix Trees. In *Proc. of the 4th Annual Symp. on Combinatorial Pattern Matching (CPM)*, 1993.
22. E. Ukkonen. Online Construction of Suffix-trees. *Algorithmica*, 14(3), 1995.