Parallel Construction of Large Suffix Trees on a PC Cluster

Chunxi Chen and Bertil Schmidt

School of Computer Engineering,Nanyang Technological University, Singapore
{pg03452644,asbschmidt}@ntu.edu.sg

Abstract. The suffix tree is a key data structure for biological sequence analysis. Even though efficient algorithms for suffix tree construction exist, for long DNA sequences such as whole human chromosomes, their run-time is still very high . In this paper we introduce a new parallel algorithm for suffix tree construction. This algorithm uses a new data structure call the *common prefix suffix tree* (CPST). Our parallel implementation on a PC cluster leads to significant run-time savings.

1 Introduction

The suffix tree is a compact trie of all suffixes over a string. It is a key data structure in the field of bioinformatics, since it permits very efficient solutions to many string based problems. Examples include exact and approximate substring matching, the longest common substring problem and the maximal repetitive structures problem [8]. Consequently, many widely used large-scale bioinformatics applications have achieved amazing performance using suffix trees, such as MUMmer [5], REPuter [15], and OASIS [17].

Several linear-time algorithms for suffix tree construction have been introduced (see [8] for a summary). Among them, Ukkonen's algorithm is most widely used. The key feature of Ukkonen's algorithm is to make use of *suffix links*, which allow the incremental construction of suffix trees. Unfortunately, these algorithms are impractical for constructing large size suffix trees because of high memory overheads. For example, the suffix tree of the whole human chromosomes of length 3 Giga base pairs (GBp) using the advanced space saving optimization requires 30 to 50 gigabytes of memory [14]. Therefore, new suffix tree construction approaches are required in bioinformatics because biological sequences typically have very large size and sequence datasets are growing at an exponential rate [20].

In order to tackle the memory bottleneck problem in constructing a large size suffix tree, researchers have tried several approaches. We summarize this research work into four categories:

1) `Space saving optimizations`. This approach exploits various kinds of redundancies in suffix trees to obtain more space efficiency[14]. However, the internal structure of suffix trees doesn't permit very significant space saving optimization without any sacrifice of suffix tree virtues.

2) `Disk-based approaches`. Disk-based approaches [7, 9, 19] hold the suffix trees in second memory. Unfortunately, suffix tree construction has poor memory locality since it requires a semi random walk over the tree as it is constructed [6]. Therefore, large-size suffix trees that will not fit in memory would take an unacceptably long time to construct and be accessed due to excessive page faulting.
3) `New data structures`. Another method is to develop alternative data structures which store less information than suffix trees and therefore have lower memory overheads. Some new data structures are *suffix array* [18], *level compressed trie* [1], *suffix binary search tree* [10], *suffix cactus* [12] and *PT-tree* [4]. This approach has the following two common shortcomings [14]. Firstly, they are specifically designed for certain applications and can not be adapted to other kinds of problems without severe performance degradation. Thus, they are not as versatile as suffix trees. Secondly, direct construction of these data structures is usually slower than suffix tree construction.
4) `Constructing suffix trees in parallel`. This approach uses the idea of processing sub-trees independently. Once all the sub-trees have been constructed it is possible to merge them together to form a complete suffix tree. We call this the *sub-tree idea*.

In this paper, we are using a PC cluster to parallelize suffix tree construction. A similar approach has been previously used in [3] and [2]. Unfortunately, [3] only gives some actual experiments on a binary alphabet, which is not relevant in practice; [2] constructs suffix trees not in a cluster, but a SMP machine with 4 CPUs and large memory. The main contributions of this paper are as follows:

1) `Presentation of a data structure with the corresponding` $O(n)$`-time construction method`. The data structure is called *common prefix suffix tree* (CPST). All suffixes in a CPST share a common prefix. A standard suffix tree can be divided into a number of CPSTs. Each CPST can be tackled independently by one node in a parallel environment. We present an algorithm that permits a linear time construction of CPSTs.
2) `Implementing the proposed method efficiently on a PC cluster`. The major difficulty of constructing a large suffix tree inside a cluster arises from the need to access the whole input sequence while constructing CPSTs. Our solution is to set aside several *data-servers* which hold the whole sequence. Processes constructing CPSTs then can access the sequence through communicating with these *data-servers*.

The rest of the paper is organized as follows. In Section 2, we provide the preliminaries of suffix trees. In Section 3, we give the description of the CPST and the algorithm for linear time construction. The parallel implementation on a PC cluster is described and evaluated in Section 4. Finally, Section 5 concludes our paper and with an outlook to further research.

2 Preliminaries

A suffix tree for a string S of length L is a rooted directed tree with exactly L leaves numbered 1 to L. For any leaf i, the concatenation of the edge labels on the path from the root to the leaf exactly spells out the suffix of S that starts from location i. Assume xs is a string over an alphabet Σ, where $x \in \Sigma$ and $s \in \Sigma^*$. In a suffix tree, for an internal node A with path-label (from the suffix tree root to the node) xs, there exists another node B with path-label s, Then the pointer from B to A is called a *suffix link*. The reason that *suffix links* are of interest is that they permit the suffix tree construction in linear time [8]. The suffix tree with corresponding *suffix links* for the string $S = accattgaagcgttaccagttat\$$ is shown in Figure 1.

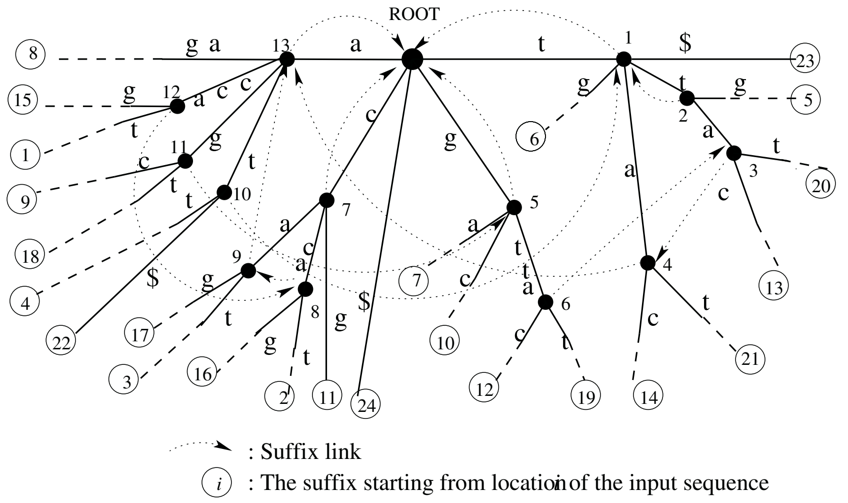


Fig. 1. The suffix tree of the sequence $accattgaagcgttaccagttat\$$.

3 Constructing a Suffix Tree with CPSTs

3.1 CPST: Common Prefix Suffix Tree

A way of dividing the problem (constructing suffix trees) into smaller sub-problems is to group the suffixes of a string first and then construct suffix trees for each suffix group. All suffixes of a string can be grouped according to the prefixes of each suffix. We define a *common prefix suffix tree* (CPST) of a string S to be a compact trie of a subset of the suffixes of the string which start with a same prefix (shown in Definition 1).

Definition 1. `Common prefix suffix tree (CPST)`*: For a given string S and a substring "compre" of S, a common prefix suffix tree, denoted as* $CPST(S, compre)$*, is the compact trie of all suffixes of S which start with compre.*

Figure 2 shows four CPSTs for the string $S = accattgaagcgttaccagttat\$$ and the common prefixes a, c, g, and t, i.e. $CPST(S, a)$, $CPST(S, c)$, $CPST(S, g)$, and $CPST(S, t)$. All CPSTs have fictitious connections to a root node. Considering the case $compre = g$, all suffixes in S starting with the g are $gaagcgttaccagttat\$$, $gcgttaccagttat\$$, $gttaccagttat\$$, and $gttat\$$. The trie of all these suffixes are presented by he $CPST(S, g)$.

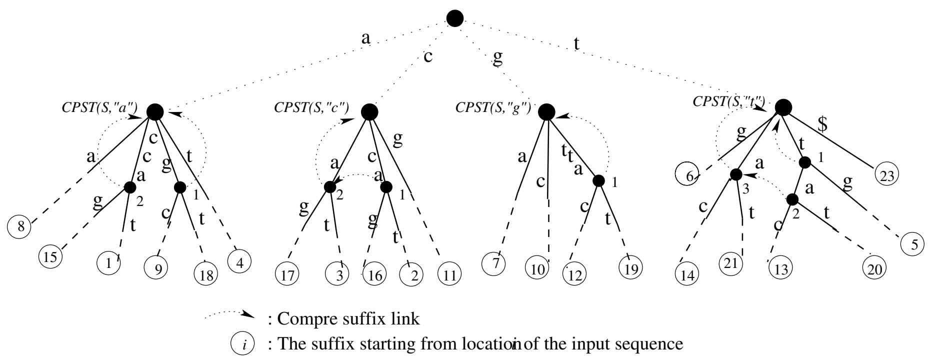


Fig. 2. The CPSTs of the sequence $accattgaagcgttaccagttat\$$. using $compre$s: a, c, g, and t.

3.2 Constructing CPSTs in Linear Time

A CPST is actually a subtree of a standard suffix tree. Once all the CPSTs of a given string have been constructed, the standard suffix tree of the string can be easily derived by concatenating the roots of every CPST with the virtual root of the standard suffix tree (shown in Figure 2).

Definition 2. `Suffix chain`*: For each internal node of a suffix tree, there exists a directed chain of suffix links starting from this node and ending at the root node. This directed chain is called suffix chain.*

For example in Figure 1, The *suffix chain* for internal node 3 is: $3 \to 4 \to 13 \to root$.

Definition 3. `Compre suffix link`*: Given are two internal nodes A and B inside the same CPST. We define a compre suffix link from A to B, if A and B are part of the same suffix chain in the standard suffix tree, where A is before B and no other internal node of the CPST lies between A and B on this suffix chain. We also define a compre suffix link from B to the root of the CPST, if no other node inside the CPST is part of the suffix chain between B and the root of the standard suffix tree.*

Let's consider the *suffix chain* illustrated in Figure 3: $F \to A \to C \to D \to B \to E \to root$. Node A and B belong to $CPST_2$. We draw a *compre suffix link* from A to B, since the part of *suffix chain* between A and B ($A \to C \to D \to B$) doesn't contain any other nodes of $CPST_2$. Additionally, there is a *compre suffix link* from B to the root of $CPST_2$.

In order to simplify the description, we use $s[i, j]$ to denote the substring of S starting at location i and ending at location j. $len(compre)$ denotes the length of $compre$. $pathlabel(N)$ denotes the characters on the path from the root to the internal node N. If suffix i starts with the prefix $compre$, we say suffix i is *valid* for $CPST(S, compre)$. The edge characters from node A to B is denoted as $e(A, B)$.

Theorem 1. *Given an internal node A of CPST(S, compre) with* $pathlabel(A) = S[l_1, l_2]$*. Assume suffix* l_3 *is the next valid suffix for CPST(S, compre) and* $l_3 +$

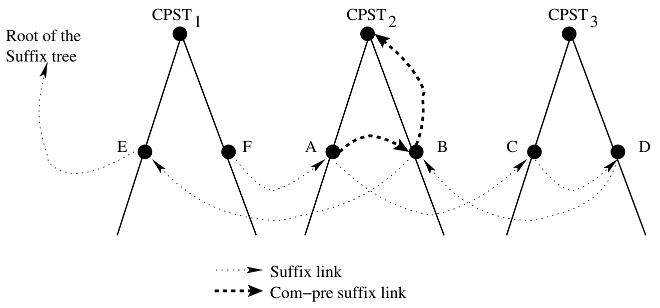
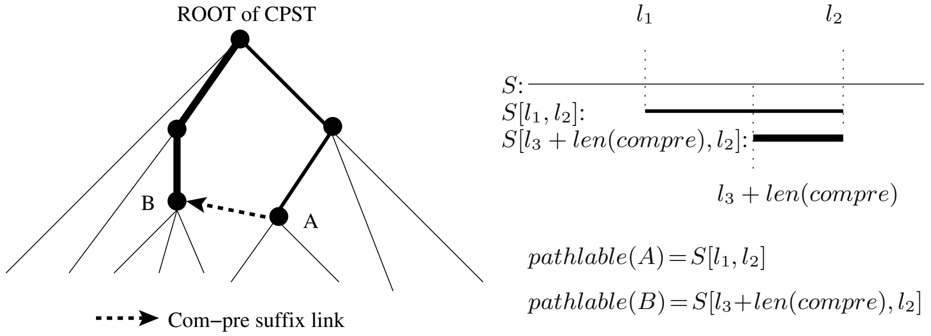


Fig. 3. An example for the *compre suffix link.*



ROOT of CPST
B
A
Com–pre suffix link
l_1
l_2
S:
$S[l_1, l_2]$:
$S[l_3 + len(compre), l_2]$:
$l_3 + len(compre)$
$pathlable(A) = S[l_1, l_2]$
$pathlable(B) = S[l_3 + len(compre), l_2]$

Fig. 4. The illustration for Theorem 1.

$len(compre) \leq l_2$. *Then there exists an internal node B with* $pathlabel(B) = S[l_3 + len(compre), l_2]$ *and a compre suffix link from A to B.*

Proof. The proof has two parts.

1. `Existence of` B. Since A is an internal node, there exists at least two substrings of S with $pathlabel(A) = S[l_1, l_2] = S[l_4, l_5]$. Hence, there are also two substrings of S with $S[l_3 + len(compre), l_2] = S[l_6, l_5]$. Therefore, there must be an internal node B with $pathlabel(B) = S[l_3 + len(compre), l_2]$.
2. `Existence of compre suffix link from` A `to` B. We show there exists a direct chain of *suffix links* from A to B by induction over $n = l_3 + len(compre) - l_1$. The claim then follows since A and B are inside the same $CPST$.

   `Basic Step:` $n = 1$. Obviously, there is a directed *suffix link* from A to B.
   `Inductive Step:` According to induction hypothesis, there is a directed chain of *suffix links* from A to a node C with $pathlabel(C) = S[l_3 + len(compre) - 1, l_2]$. Since there must also be a *suffix link* from C to B, it can be concluded that there is a directed chain of *suffix links* from A to B.

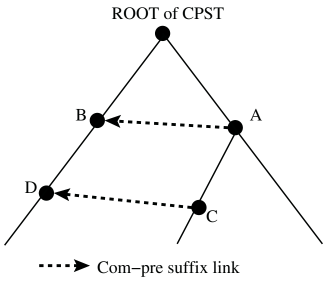


Fig. 5. The illustration for theorem 2.

Theorem 2. *Given an internal node C with parent node A inside CPST(S, compre). Assume there is a suffix link from A to another internal node B. Then there exists a node D below B with e(A, C) = e(B, D) and there is a compre suffix link form C to D.*

Proof. Let $pathlabel(A) = S[l_1, l_2]$ and $pathlabel(C) = S[l_1, l_4]$. Since there is a *compre suffix link* from A to B, it holds $pathlabel(B) = S[l_3, l_2]$, where l_3 is the next valid suffix for $CPST(S, compre)$ after l_1 and $l_3 < l_2$. With theorem 1 follows that it exist an internal node D with $pathlabel(D) = S[l_3, l_4]$ and a compre suffix link from C to D. obviously, D is below B and $e(A, C) = s[l_2 + 1, l_4] = e(B, D)$.

Our algorithm constructs a CPST through orderly inserting *valid* suffixes for the CPST. In [8], the introduction of $suffixlinks$ permits the usage of the skip/count trick which makes the Ukkonen's algorithm be in linear time. the $compre\_suffix\_link$ in CPSTs is the counterpart of $suffixlinks$ in standard suffix trees according to theorem 2. It can locate the next node in the CPST through using the skip/count trick instead of traversing the CPST from its root. Based on the definitions and theorems above, the algorithm of constructing $CPST(S, compre)$ can be described as follows:

```
CPST construction algorithm:
Input:    String $S = \alpha\$$, where $\alpha \in \Sigma^*$, $\$ \notin \Sigma$, and $\Sigma$ is a finite alphabet.
           Common prefix $compre \in \Sigma^*$ with $|compre| < |\alpha|$
Output:  $CPST(S, compre)$

$N = number\_of\_valid\_suffixes(S, compre)$;
IF $(N == 0)$ RETURN $(nil)$;
FOR $i = 1$ TO $N$ BEGIN
    $\nu(i)$ =starting position of the $i^{th}$ valid suffix in $S$;
END
$current\_node = CPST\_root$;
$theorem2\_flag\_node = current\_node.father.compre\_suffix\_link$;
FOR $i = 1$ TO $N$ BEGIN
    IF $((current\_node == CPST\_root)||(theorem2\_flag\_node == CPST\_root))$
        $new\_nodes\_info = traversal(current\_node, \nu(i), S)$;
    ELSE
        $new\_nodes\_info = skip\_count(theorem2\_flag\_node, current\_node.edge_labels)$;
    $create\_new\_nodes(new\_nodes\_info)$
    $create\_new\_CompreSuffixLink(new\_internal\_node, old\_internal\_node)$
    $current\_node = new\_internal\_node$;
    $theorem2\_flag\_node = current\_node.father.compre\_suffix\_link$;
END
RETURN ($CPST\_root$);
```

Theorem 3. *For an input sequence S and a substring compre, $CPST(S, compre)$ can be constructed in linear time.*

Proof. Our algorithm constructs a $CPST(S, compre)$ through orderly inserting *valid* suffixes for the CPST. Assume that the insertion of $valid$ suffix V_i results in a new internal node A with $pathlabel(A) = S[V_i+len(compre), l_i]$. The time complexity for this assertion is $O(l_i - V_i)$. For $valid$ suffixes whose starting locations are in the range $[V_i, l_i]$ (such as suffixes V_{i+1} and V_j), we can use the skip/count trick [8] to insert them according to theorem 2. The time complexity for these insertions using the skip/count strick are $O(m)$, where m is the number of suffixes whose starting locations are in the range $[V_i, l_i]$. Hence, the time complexity for inserting all $valid$ suffixes whose starting locations are in the range $[V_i, l_i]$ is $O(ll_i - V_i) + O(m)$. Obviously, it is linear to the length of the range. The whole input string is composed by these ranges. Therefore, the insertions of all $valid$ suffixes can be accomplished in linear time.

4 Parallel Implementation and Performance Evaluation

4.1 Input DNA Sequence

The DNA sequence used in this paper is human chromosome NC_000001.4 which is downloaded from [20]. The alphabet of actual DNA sequences consists of 16 characters, in which a, c, g, and t represent the four bases of DNA and r, y, w, s, m, k, b, d, h, and v represent undetermined base-pares. In the paper, we only consider the determined bases a, c, g, and t. For example, the human chromosome NC_000001.4 extracted by us is of length 222,827,884 bp.

4.2 Prefix Distribution in DNA Sequences

The purpose of presenting the new data structure called CPST is to divide a large-size suffix tree into a number of smaller size CPSTs first and then each CPST can be processed independently. This idea presumes that the suffix trees can be divided efficiently using CPST. However, this might not be possible for systematically biased sequences. Let's consider a worst case. For a sequence $S = aaaaaaaaaaaaaaaaaaaa$, all the suffixes of the sequence start with same prefix a. Thus, the idea of CPST is inefficient.

Fortunately, systematically biased sequences rarely occur in practice. The appearance of the 4 symbols $a, c, g, and\ g$ in actual DNA sequences is almost evenly distributed. This ensures that the number of DNA sequence suffixes starting with different possible prefixes are not severely imbalanced. Here we take human chromosome

NC_000001.4 length of 222,827,884 as an example. Table 1 shows that the number of suffixes starting with different prefixes are well balanced. This means the suffix tree can be divided efficiently into sub-problems using CPSTs.

Table 1. The number of suffixes of the human chromosome NC_000001.4 length of 222,827,884 (only consider a, c, g and t)which start with 1-letter and 2-letter prefixes.

$compre$	Num of suffixes	$compre$	Num of suffixes	$compre$	Num of suffixes	$compre$	Num of suffixes
a	64875254	c	46493994	g	46483769	t	64974866

$compre$	Num of suffixes	$compre$	Num of suffixes	$compre$	Num of suffixes	$compre$	Num of suffixes
aa	21191409	ac	11189673	ag	15878823	at	16615349
ca	16200299	cc	12132633	cg	2256627	ct	15904435
ga	13313713	gc	9838754	gg	12121539	gt	11209763
ta	14169833	tc	13332934	tg	16226780	tt	21245318

4.3 Parallelization Strategy

During the course of constructing CPSTs, the input string must reside in memory. This means every process in the parallel environment must allocate enough memory to hold the the input sequence first and then the remaindering memory can be allocated to construct CPSTs. Obviously the efficiency is low when the input sequence is large. This is the key reason some parallel implementations do not scale well.

If a cluster permits fast intra-cluster communication, it is possible that one or more nodes hold the input string while other nodes efficiently access the string by intra-cluster communication. We call this the *sharing input string idea*. Our implementation uses one or more *data-servers* which hold the whole input sequence. The processes constructing CPSTs (*constructors*) access the sequence through communication with these *data-servers*.

In order to decrease the communication between *dataservers* and *constructors*, we introduce the concept of *smallnode* and *largenode*. The communication between *dataservers* and *constructors* consists of two parts: 1) the *constructors* need to access the input string; and 2) the *constructors* need to get the edge labels of a node. The communication in Case 1 has good efficiency since the *constructors* can get a whole block of the substring a time. The highly frequent and low efficient communication comes from the Case 2, because the number of the nodes is large and the overhead of every communication is high. We classify the nodes of a CPST according to the lengths of their edge lables. The nodes whose edge-label lengths are larger than a criterion ($nodesize$) are called $largenodes$, or else called $smallnodes$. If the edge labels of the $smallnodes$ are kept in their CPSTs, the access to these $smallnodes$ doesn't need communication. Therefore, the communication in Case 2 will decrease.

4.4 Performance Evaluation

The cluster used in this paper consists of 10 nodes connected by a Gbit/s myrinet switch. Each node comprises two 2.6GHz CPUs and 1 Gigabytes RAM. In our experiments, the length of $compre$ is set to 2 and therefore the number of CPSTs is 16. The $nodesize$ is set as 10. The number of *constructors* are 4 times that of *dataservers*. Figure 6 shows

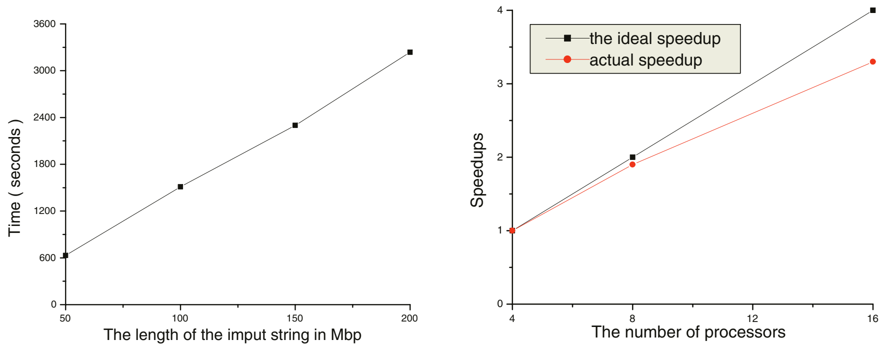


Fig. 6. The left part shows the runtimes for input strings with different lengths; The right part shows speedups using the input string length of 50M.

that the construction time of our implementation is in a linear relationship to the length of the input string. In addition, the speedup is almost linear.

5 Conclusion

The suffix tree is a key data structure for biological sequence analysis. However, construction of a suffix tree for long DNA sequences is made challenging by high memory overheads and poor memory locality. In this paper, we have introduced an efficient parallel algorithm for large-scale suffix tree construction using the CPST data structure. We have shown how a standard suffix tree can be divided into a number of CPSTs. Each CPST can then be processed independently by one cluster node. Our algorithm permits linear-time construction of CPSTs. In order to reduce space while constructing CPSTs inside a cluster, we use one or more *data-servers* which hold the whole sequence inside the cluster. *Constructors* access the input sequence through communicating with these *data-server*s. Our implementation can achieve linear space for a human chromosome DNA sequence.

References

1. A. Andersson and S. Nilsson, "Efficient Implementation of Suffix Trees", *Software-Practice and Experience*, 25(2), 129-141, 1995.
2. A.L. Brown. "Constructing Chromosome Scale Suffix Tree". *the 2nd Asia-Pacific Bioinformatics Conference*. New Zealand, 2004.
3. R. Clifford and M. Sergot. "Distributed and Paged Suffix Trees for Large Genetic Databases". *Journal of Discrete Algorithms*. Accepted.
4. L. Colussi and A. De Col, "A time and space efficient data structure for string searching on large texts", *Information Processing Letters*, 58(5), 217-222, 1996.
5. A. Delcher, A. Phillippy, J. Carlton, and S. Salzberg. "Fast Algorithms for Large-scale Genome Alignment and Comparision." *Nucleic Acids Research*, 30(11):2478-2483, 2002.

6. M. Farach, P. Ferragina, and S. Muthukrishnan. “Overcoming the Memory Bottleneck in Suffix Tree Construction”. *Proc. of IEEE Annual Symposium on Foundations of Computer Science*, 1998.
7. P. Ferragina and R. Grossi. “The string B-Tree: a new data structure for string search in external memory and its application.” *Journal of the ACM*, 46(2):238-280, 1999.
8. D. Gusfield. *Algorithms on strings, trees and sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
9. E. Hunt, M.P. Atkinson, and R.W. Irving. “A Database Index to Large Biological Sequences.” *The VLDB J.*, 7(3):139–148, 2001.
10. R.W. Irving, “Suffix Binary Search Trees”, *Research Report*, Department of Computer Science, University of Glasgow, 1996.
11. R. Japp. “Persistent Indexes for Data intensive applications”. *Twentieth British National conference on Databases*. Coventry, UK, Lecture Notes in computer Science, 2712.
12. J. Kärkkäinen, “Suffix Cactus: A Cross Between Suffix Tree and Suffix Array”, *Proc. of the Annual Symposium on Combinatorial Pattern Matching (CPM’95)*, LNCS 937, 191-204, 1995.
13. J. Kärkkäinen and E. Ukkonen. “Sparse Suffix Tree”. COCOON’96, LNCS1090, Hongkong, 1996.
14. S. Kurtz. “Reducing Space Requirement of Suffix Trees”. *Software Practice and Experience*, 29(13):1149–1171, 1999.
15. S. Kurtz and C. Schleiermacher. “REPuter: Fast Computation of Maximal Repeats in Complete Genomes.” *Bioinformatics*, 15(5):426-427, 1999.
16. U. Manber and E.W. Myers, “Sufix Arrays: A New Method for On-line String Searches”, *SIAM Journal on Computing*, 22(5), 935-948, 1993.
17. C. Meek, J. M. Patel, and S. Kasetty. “OASIS: An Online and Accurate Technique for Local-alignment Searches on Biological Sequences.” In *VLDB*, 2003.
18. G. Navarro, R. Baeza-Yates, and J. Tariho. “Indexing Methods for Approximate String Matching.” I*EEE Data Engineering Bulletin*, 24(4):19–27, 2001.
19. S. Tata, R.A. Hankins, J.M. Patel. “Practical Sufix Tree Construction.” in *proceedings of the 30th VLDB Conference*, Toronto, 2004.
20. The Growth of GenBank, NCBI, 2004. http://www.ncbi.nlm.nih.gov/genbank/
21. MPICH project: http://www-unix.mcs.anl.gov/mpi/mpich/