# Distributed and Paged Suffix Trees for Large Genetic Databases

Raphaël Clifford and Marek Sergot

Department of Computing, Imperial College, London.
raphael@clifford.net, m.sergot@imperial.ac.uk

**Abstract.** We present two new variants of the suffix tree which allow much larger genome sequence databases to be handled efficiently. The method is based on a new linear time construction algorithm for "sparse" suffix trees, which are subtrees of the whole suffix tree. The new data structures are called the *paged suffix tree* (PST) and the *distributed suffix tree* (DST). Both tackle the memory bottleneck by constructing subtrees of the full suffix tree independently and are designed for single processor and distributed memory parallel computing environments (e.g. Beowulf clusters), respectively. The standard operations on suffix trees of biological importance are shown to be easily translatable to these new data structures. While none of these operations on the DST require inter-process communication, many have optimal expected parallel running times.

## 1 Introduction

The suffix tree is the key data structure of computational pattern matching which allows a multitude of sophisticated operations to be performed efficiently (see e.g. [3, 12]). In the field of bioinformatics this includes whole genome alignment [6], analysis of repetitive elements [17], and fast protein classification [7], amongst many others. However, the main obstacle to more widespread acceptance of these methods remains that of memory use. Suffix trees have high memory overheads, and the poor memory locality, both of their construction and of querying algorithms, make disk-based implementations highly problematic.

Generally, the existing approaches to tackling this memory bottleneck can be divided into two categories. On the one hand, there are those that attempt to improve the implementation of the suffix tree itself, at the cost of limiting the maximum problem size (see e.g. [16]). On the other hand, related data structures have been developed which have lower memory overheads at the cost of either restricting the range of queries that can be performed or increasing their time complexity (e.g. suffix arrays [18], level-compressed tries [2] and suffix cactuses [14], sparse suffix trees [15]).

To tackle significantly larger problem sizes (e.g. data that is hundreds of times larger than available RAM) a disk-based scheme would be desirable. However, as a result of the poor locality mentioned above, most existing applications of disk-based schemes assume that the order of node traversal to be performed at query time is known in advance. Ferragina and Grossi have studied the efficient external memory construction of string indices in general [8, 9, 10]. To the authors' knowledge, these methods have not yet been applied to large-scale bioinformatics problems.

We present here two new data structures for problems of intermediate size—that is, problems larger than can be handled by existing suffix tree/array methods but small enough that the input can be stored entirely in real memory—a range of at least an order of magnitude. To give some indication, the new methods allow us to store and analyse the whole human genome, perform cross species pattern matching on all available bacterial genomes at once, or search a large EST database, using a small cluster of standard PC's. The data structures are termed the *distributed suffix tree* (DST) and the *paged suffix tree* (PST). They are both based on a new extension of Ukkonen's suffix tree construction algorithm [19] which allows subtrees of a suffix tree to be constructed efficiently in space proportional to the size of the resultant data structure and not the whole suffix tree. This enables a suffix tree to be either distributed over a number of computing nodes (and queried in parallel) or for a single node to compute independent subtrees successively, querying each in turn. By effectively splitting the input string lexicographically (not into contiguous substrings) we show that all the most popular biologically inspired operations on suffix trees exhibit optimal or near optimal parallel speedups. Furthermore problems which would previously have been impossible to solve due to their size can now be tackled efficiently, either in parallel or serial and with modest hardware requirements.

The DST and PST construction algorithms have been implemented in C on an 8 processor distributed memory parallel computer, increasing by a factor of 7.65 the size of the largest database that could be indexed. Exact set matching and repeat finding procedures for random data have also been implemented and performed on the DST. The results, not shown here for space reasons, showed substantial speedups (with average efficiencies in excess of 90% and 99%, respectively) and exhibited good scalability, confirming the theoretical analysis [5]. For systematically biased genetic data, preliminary results show that simple load balancing schemes can successfully increase the parallel efficiency of biological operations to close to 90%.

The method is simple to apply. Almost any current bioinformatic technique that relies on suffix trees can be modified to take advantage of DSTs or PSTs, greatly extending the range of problem sizes that can be tackled. Also, complex or time consuming queries, such as the preliminary stages of matching all ESTs against the human genome, can be performed with optimal or near optimal efficiency in parallel. In the next section we first describe the new data structures and then present the construction algorithms. We then present the expected time efficiencies of a sample of operations on the DST and show empirical results using a snapshot of the sequencing of human chromosomes 21, 22, and X.

Construction algorithms for two previous types of (sparse) suffix tree have been considered in [15] and [1]. The first work considers suffix trees that contain only *evenly spaced* suffixes of the text. This allows pattern matching to be performed on a single, smaller, suffix tree at the cost of increasing the running time of the query. The definition of suffix links proposed there is not suitable for the class of sparse suffix trees we consider as it depends on properties of evenly spaced suffixes that do not hold in general. However, the construction algorithm presented here can be viewed as an extension of that work. The method in [1] uses a quite different word-oriented approach that relies on delimiters between "words" in the text. These delimiters may not overlap and so the method can not be directly applied to the problem of constructing subtrees of a suffix tree.

## 2 Distributed and Paged Suffix Trees

A suffix tree of input string $t$ is a compacted trie of the suffixes of $t$. We define a *sparse suffix tree* (SST) of input string $t$ to be a compacted trie of a subset of the suffixes of $t$. Here, we are interested in the special case where all the suffixes in this subset start with the same prefix $z$ and assume from now on that all SSTs are of this type. Both *distributed suffix trees* (DST) and *paged suffix trees* (PST) are simply collections of SSTs defined in this way. The most efficient use for a PST is for applications where the different SSTs can be queried successively and independently. In that case the SSTs will be constructed on a single processor only as they are needed and then discarded before the next one is required. We concentrate on the DST from now on to simplify the explanation and return to the PST in Section 2.4. The SSTs in a PST are defined in exactly the same way as they are for a DST.

Usually, a single SST will be held at each computing node and the union of the path labels of the leaves of these SSTs will be the full set of suffixes of $t$. In other words, every suffix of $t$ will be represented by exactly one SST at exactly one of the computing nodes. An example DST and the corresponding standard suffix tree are given in Figures 1 and 2.

In this case the prefixes for the 6 different SSTs are $aa$, $ac$, $ca$, $cc$, $a\$$ and $\$$. Each SST has been connected to a central root node. The *sparse suffix links* will be explained below but the most important feature is that in the standard suffix tree the suffix links can point the full width of the tree. In the DST the new links point only to nodes that are within the same SST. This allows the SSTs to be constructed independently without any inter-process communication.

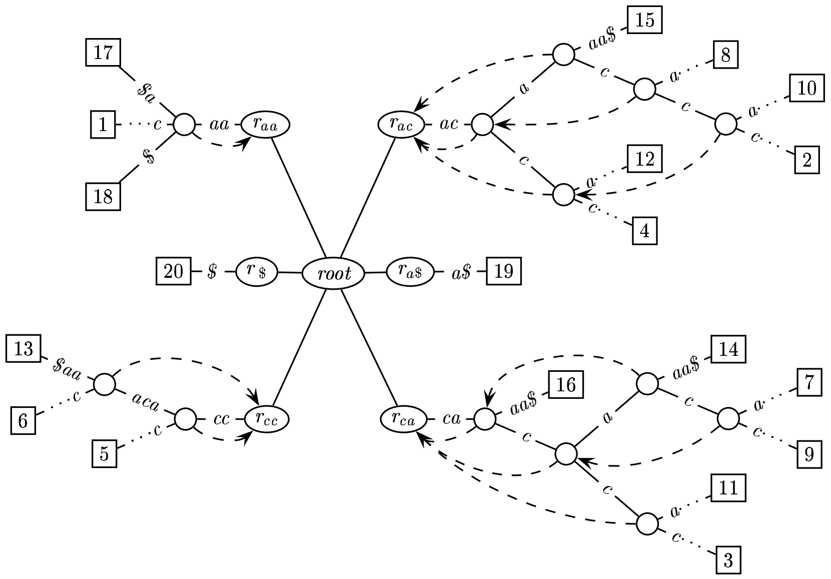

**Fig. 1.** The SSTs for $aacacccacacaccacaaa\$$ with their respective root nodes labelled $r_{aa}, r_{ac}, r_{ca}, r_{cc}, r_{a\$}$ and $r_\$$. The sparse suffix links for the valid sets $V_{aa}, V_{ac}, V_{ca}, V_{cc}, V_{a\$}$ and $V_\$$ are marked with dashed arrows. Note that the final suffixes, $a\$$ and $\$$, are included but typically will not be used.

## 2.1 Preliminaries

We call the input string $t$ and assume that $n = |t|$ throughout this paper. The characters of $t$ are drawn from an ordered alphabet $\Sigma$ and we let $\sigma = |\Sigma|$. $t[i, j]$ represents the substring of $t$ starting at position $i$ and terminating at position $j$, inclusively. A suffix of $t$ is said to be *repeated* if it occurs at least once as a non-suffix substring of $t$. We also need to be able to specify which suffixes of $t$ are to be included in an SST. This is done by considering a short prefix string $z$, and a set of start positions, $V_z$, for those suffixes of the input which have the string $z$ as a prefix. $V_z$ is also called the *valid set.* We say that a substring $s$ of $t$ is *valid* if $z$ is a prefix of $s$ and that an interval $I[i, j]$ is *valid* (with respect to $V_z$ and $t$) if $i \in V_z$. We say that $s$ is a *valid suffix for* for $I[i, j]$ if $s = t[k, j]$ for some $k \in V_z$ and $k > i$. Note that

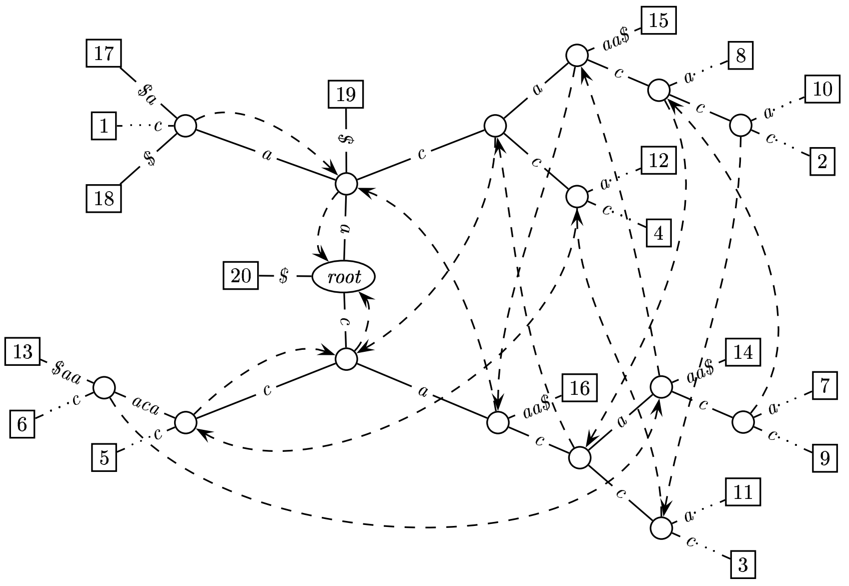

**Fig. 2.** The standard suffix tree of *aacacccacacaccacaaa*\$ with standard suffix links. This is for comparison with the merged tree in Figure 1. See the text for further explanation.

it is possible that a valid suffix for an interval may be shorter than the fixed prefix $z$, depending on the characters that follow directly in the input string.

Let $z$ be a string of length greater than or equal to 1. We say $(t, V_z)$ is an *input pair* if $V_z$ is the valid set. We write $sst(t, V_z)$ for the sparse suffix tree of $t$ using the valid set $V_z$. For a sparse suffix tree $T = sst(t, V_z)$ and an arbitrary set of strings $S$, we say that $T' = T$ *augmented by* $S$ if $T'$ is the sparse suffix tree of the valid suffixes of the input pair $(t, V_z)$ and the strings in $S$.

We call the concatenation of the edge labels on the path from the *root* to some position in the SST (either a node, or somewhere on an edge between two nodes) the *path label* of that position. To avoid confusion between strings and nodes, we use the notation $\overline{w}$ to label a node whose path label is $w$. A string which is a path label of some node is called *nodal*. A string which corresponds to any path label in the tree is said to *occur* in the tree.

*Example 1.* Consider the input

$$s = agacagc\$ \quad \text{with} \quad z = ag$$

The valid set is $V_z = \{1, 5\}$ and $sst(t, V_z)$ has two leaves labelled $\overline{agacagc}$ and $\overline{agc}$ and one branching node labelled $\overline{ag}$. The root has only one child.

## 2.2 Building the DST

We are able to construct the DST of string $t$ in $O(n)$ time in parallel with no communication overheads apart from the one-off cost of sending the input to the different nodes. This communication can be achieved on most modern local area networks by broadcasting the data to all nodes simultaneously, making the entire construction time $O(n)$. The time complexity can be achieved trivially, of course, by simply constructing the full suffix tree at each computing node using Ukkonen's algorithm [19] and then pruning it to remove unwanted nodes and edges. However, the assumption we make is that this will not be possible for large problems due to memory constraints and so a novel construction method is required.

To construct the DST in linear time we show how to construct an SST in linear time and simply run the algorithm in parallel for the different prefixes that are chosen. The resulting algorithm uses space at each node which is proportional to the size of the SST constructed, as required.

## Sparse suffix links

Suffix links play a critical role in the linear time construction of suffix trees. However, the standard definition is not sufficient for SSTs as, in general, a substring of $t$ and its longest suffix may have different prefixes. Moreover, suffixes that are shorter than the length of the prefix $z$ have to be considered separately. Online construction proceeds by reading in one character at a time from the input and stepping down in the tree until there is a mismatch. At this point a new edge or node is inserted and a jump is made in the tree to a new position, from which the process is continued. To perform this jump we need to consider the longest suffix of the current position which might require an additional edge or node in the tree, either using the current character or when more data is read in. Suffix links are used to perform this traversal and it is a fundamental property that if the current position is a node then there will be a corresponding node to which the jump should be made. A new definition of suffix links is required.

**Definition 1.** *Consider an input pair $(t, V_z)$ and sparse suffix tree $T = sst(t, V_z)$. Let $aw$ be nodal in $T$ and $v$ be the longest repeated suffix of $aw$ that occurs in $T$. A* sparse suffix link *or* ssl *is an unlabelled edge from $\overline{aw}$ to the* root *if $|v| < |z|$ and from $\overline{aw}$ to $\overline{v}$, otherwise.*

The following proposition shows that sparse suffix links are always well defined in a sparse suffix tree.

**Proposition 1.** *Consider an input pair $(t, V_z)$ and sparse suffix tree $T = sst(t, V_z)$. Let $aw$ be nodal in $T$ and $v$ be the longest repeated suffix of $aw$ that occurs in $T$. If $|v| \geq |z|$ then $v$ is nodal and therefore the sparse suffix link from $\overline{aw}$ to $\overline{v}$ is well defined.*

*Proof.* If $aw$ is nodal then there must be at least two occurrences of $aw$ in $t$, each with a different character directly to its right. $v$ must occur as a suffix of both occurrences. Therefore there are two occurrences of $v$ in $t$ with different characters directly to their right. If $|v| \geq |z|$ then $v$ must be valid (as $v$ occurs in $T$). This implies that $v$ is nodal in $T$, as required. □

We must also ensure that the total number of steps required by the construction algorithm is linear and that the correct positions can be visited by the sparse suffix links. A full proof is provided in [5]. Here we show which suffixes need to be inspected at each stage of construction and argue that this can be done efficiently.

The demonstration is an extension of that provided in [11] for standard suffix trees. First we need some further definitions. The valid set $V_z$ is always with respect to input string $t$. When the valid set is applied to a prefix $p$ of $t$, any start positions that are greater than $|p|$ are simply discarded.

**Definition 2.** *Consider input pair $(p, V_z)$ and a valid repeated suffix $s$ of $p$. Denote the set of such suffixes by $R(p, V_z)$. We define this set to include the empty string, $\epsilon$. Let $\alpha(p)$ be the longest suffix in $R(p, V_z)$.*

*Example 2.* Consider input string $t = aabaaa$ with prefix $p = aabaa$ and $V_{aa} = \{1, 4, 5\}$. $\alpha(p) = aa$ and $R(p, V_{aa}) = \{aa, \epsilon\}$.

It is an important property of $R(p, V_z)$ that all its elements must, by definition, either have length zero or be at least as long as the prefix string $z$.

**Theorem 1.** *Consider the input pair $(t, V_z)$ with $p$, a proper prefix of $t$. Let $a$ be the character in $t$ that directly follows the prefix $p$. Consider also the set, $S$, of valid suffixes, $sa$, for $I[1, |pa|]$ such that*

$$|\alpha(p)a| \geq |sa| > |\alpha(pa)|$$

*and $s \in R(p, V_z)$. Then $sst(pa, V_z) = sst(p, V_z)$* augmented by *$S$.*

*Proof.* Let $sa$ be a valid suffix for $I[1, |pa|]$. We need to insert this new suffix into the tree if and only if $s \in R(p, V_z)$ but $sa \notin R(pa, V_z)$ ($s = \epsilon$ is a special case). In this case $sa$ corresponds to a leaf in $sst(pa, V_z)$ but $s$ does not correspond to a leaf in $sst(p, V_z)$ so a new node is necessary.

1. If $|sa| > |\alpha(p)a|$ then $s \notin R(p, V_z)$. But $s$ is valid and thus corresponds to a leaf in $sst(p, V_z)$. In such a case $sa$ will correspond to the same leaf in $sst(pa, V_z)$ by the implicit growing of the corresponding open edge. No action is needed.
2. If $|\alpha(p)a| \geq |sa| > |\alpha(pa)|$ then $sa \notin R(pa, V_z)$. To determine whether any action is needed we consider the first part of the inequality which gives us $|\alpha(p)| \geq |s|$. In this case $s \in R(p, V_z)$ and therefore a new leaf $\overline{sa}$ must be added.
3. If $|\alpha(pa)| \geq |sa|$ then either $sa \in R(pa, V_z)$ or none of the non-suffix occurrences of $sa$ is valid. Recall that either $|s| \geq |z|$ or $|s| = \epsilon$. Consider the two cases:
   (a) $|s| \geq |z|$. $sa$ is a valid suffix for $I[1, |pa|]$ and therefore $z$ is a prefix of $sa$. As $sa$ is a suffix of $\alpha(pa)$ it now follows that $sa \in R(pa, V_z)$. No action is needed.
   (b) $s = \epsilon$. $sa$ is simply the first character of $z$ and must therefore occur in $p$ (as $\alpha(pa)$ has non-zero length). Therefore no action is needed.

So we need only consider case 2 to insert any new leaves required. Therefore only suffixes, $sa$, satisfying $|\alpha(p)a| \geq |sa| > |\alpha(pa)|$ where $s \in R(p, V_z)$ need to be inserted into $sst(p, V_z)$ as required. □

The positions in the tree of successive elements of $R(p, V_z)$ can be visited using sparse suffix links in an analogous way to the way positions of successive suffixes are visited in a standard suffix tree. The total number of these suffixes is $O(n)$ and, assuming that $|z|$ is constant, the total time taken to visit all their positions is also $O(n)$. Therefore, the whole construction algorithm runs in $O(n)$ time.

**Theorem 2.** *For a input pair $(t, V_z)$ with $|z| \geq 1$ , $sst(t, V_z)$ can be constructed in $O(n)$ time, where $n = |t|$.*

*Proof.* There are two main differences between our SST construction algorithm and the suffix tree construction algorithms of [19] and [15]. The first is that we use ssl's instead of suffix links and the second is that we must apply a different rule if we follow an ssl to the root. If we ignore this extra rule for a moment, the running time follows closely the reasoning that is presented in the previous papers and so we do not describe it further here. However, if an ssl is followed to the root then the position of the longest repeated prefix of the current suffix may be anywhere between the root and its child node. The position of the next suffix which must be inserted into the tree can trivially be found however, by simply stepping down in the tree, one character at a time. The characters of the next valid suffix in $t$ are used to step down until a mismatch is found. As the next valid suffix must, by definition, have $z$ as a prefix no mismatch will be found before the first child node is reached. Each time an ssl is followed the index of the current suffix being considered increases. This index is never decreased, so the total number of ssl's that are followed is $O(n)$. Therefore, the total number of one character steps taken down the tree is $O(n|z|)$. Assuming that $|z|$ is bounded above by a small constant, the total running time for the algorithm is therefore $O(n)$ as required. □

### 2.3 Experimental results

To test the sparse suffix tree construction algorithm we ran a series of experiments using random binary data and different prefix lengths. A fixed prefix of the desired length was chosen for each test. For the first test the prefix is set to "a", for the second "aa" and so on. As there is no inter-node communication requred in DST construction, the running times shown reflect the parallel running time for computing all the SSTs of a DST on a cluster of computers (excluding the time to broadcast the input to the different processors). The construction times for a PST can be calculted by multiplying the running time for each prefix by the number of prefixes of that length. 20 bytes/valid suffix were required for our implementation. More sophisticated implementations such as those described in [11] could significantly reduce this number. The purpose here is to compare the new and old data structures using the same implementation techniques.

The timings for different length prefixes are presented in Figure 3. The implementation is in C and was run on a 512MB 800MHz AMD system running Linux 2.2.19. The construction of the complete suffix tree slows drastically for inputs larger than 24.4 million characters. This is when real memory is exhausted. With prefix "aa" the maximum size is 47.8 million. With prefixes "aa" and "aaa" it grows to 94.4 million and 186.7 million respectively. So, using 8 SSTs and a binary alphabet, we are able to construct a DST or PST for problems approximately 7.65 times larger than before.

### 2.4 Operations on the DST and PST

Gusfield [12] provides what can be regarded as a canonical list of major bioinformatic techniques on suffix trees. The algorithms can be broadly classified into three categories according to how they perform on a DST. The first category is that of exact pattern matching algorithms which have very fast serial solutions. The only obstacle to their practical use is the size of the suffix tree that must be computed. A DST will allow a much larger text to be indexed and each search will require only one processor to perform a computation. This is in contrast to a segmentation of the text into contiguous sections which would require all processors to perform calculations for each query. The second category consists of operations such as the calculation of matching statistics [4] which can be performed on a DST but for which there is little parallel speedup. The third category are the algorithms which both benefit from being able to be run on larger data sets and which show optimal or near optimal speedup on a DST. It is perhaps surprising that all commonly used bioinformatic operations fall into either the first or the third category—that is, they perform traversals of the tree which can easily be translated to a DST without incurring any communication overheads. There is, of course, another class of algorithms outside this list which would require inter-process comunication if run on a DST. It is likely that these algorithms would also be impractical to run on a PST as only one SST will be in real memory at any given time. It is an open question which of them can be translated into efficient parallel algorithms on a DST.

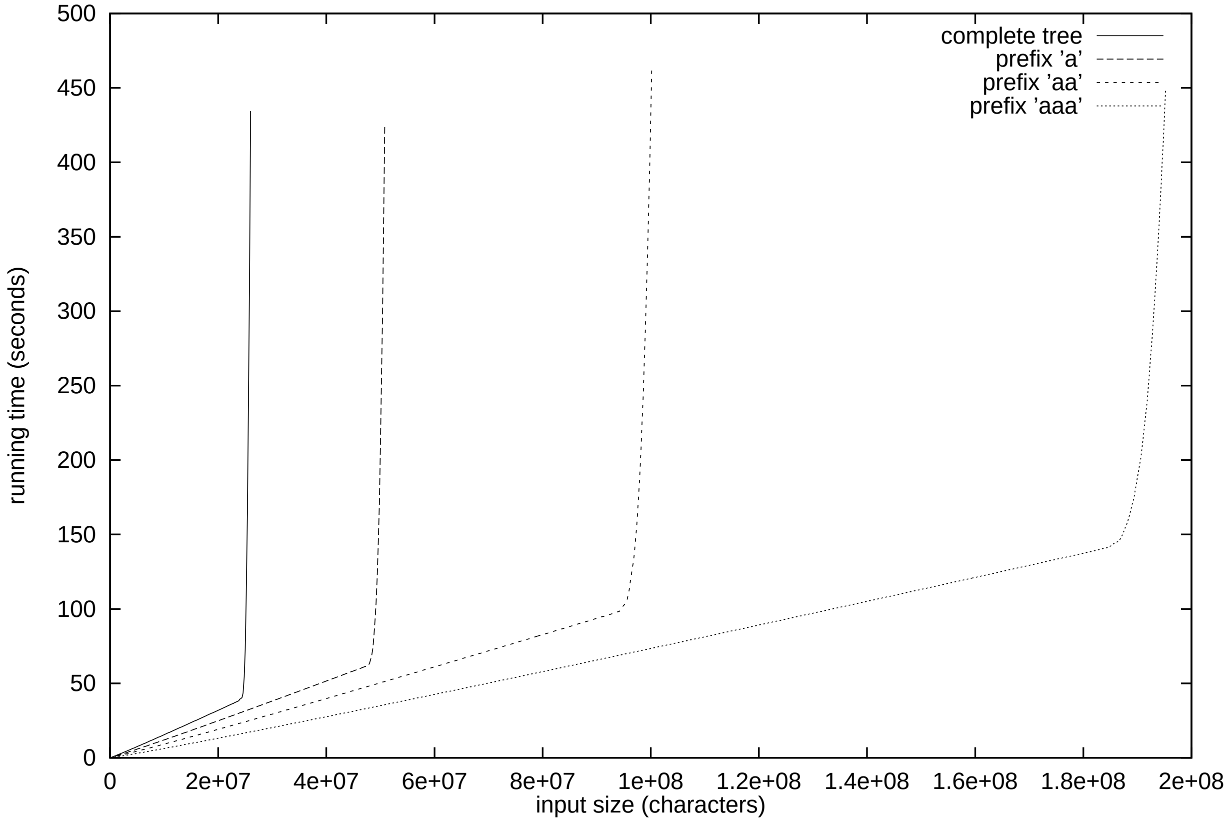

**Fig. 3.** A comparison of sparse suffix tree construction times using a binary alphabet. The sharp upturn in each line indicates the point at which real memory was exhausted.

We now summarise the analysis of five representative problems from the first and third categories described above. They are Longest Common Substring, Exact Local Matching (LCS, ELM), Maximal Repeat Finding, All Pairs Suffix-Prefix [13] and Exact Set Matching. Full descriptions along with their serial solutions using suffix trees can be found in Gusfield [12] and elsewhere. All five problems can clearly be solved using a PST with the running time complexity, excluding the time to construct the SSTs, exactly the same as for the serial case. We now examine their solutions on a DST.

Because we are interested in average case and not worst case analysis we make the commonly used assumption that the input characters are independent and uniformly distributed (i.u.d.). In practice, this assumption may not hold, of course; load balancing for systematically biased data is discussed at the conclusion.

We suppose that there are $k$ computing nodes and assume for simplicity that $k = \sigma^{|z|}$, where $\sigma$ is the alphabet size and $z$ is the fixed prefix. Table 1 compares the expected running times for the solution of these five problems using the fastest serial method (based on a standard suffix tree) and a parallel method (based on distributed suffix trees). The derivation of these results is sketched briefly below.

**Table 1.** Post-construction average time complexities for five different problems using standard and distributed suffix trees with $k$ computing nodes. $r$ is the number of strings for the All Pairs Suffix-Prefix problem and the number of patterns for Exact Set Matching.

| Problem | Expected Running Time | |
|---|---|---|
| | Standard ST (Serial) | Distributed ST (Parallel) |
| LCS and ELM | $O(n)$ | $O(n/k)$ |
| Maximal Repeat Finding | $O(n)$ | $O(n/k)$ |
| All Pairs Suffix-Prefix | $O(n + r^2)$ | $O((n + r^2)/k)$ |
| Exact Set Matching | $O(r \log n)$ | $O((r\log n)/k)$ |

## Longest Common Substring and Exact Local Matching

The serial solutions to these problems perform a full traversal of the suffix tree. As both the longest common substring and an exact local match of two strings will have the same prefix by definition, the problem can be solved by simply applying the serial solution to the SSTs in the DST in parallel. The average running time is therefore determined by the expectation of the size of the largest SST. Assuming i.u.d. characters in the input, this can be shown to be

$$\mathbb{E}\left[\max_z(|V_z|)\right] = \frac{n}{k} + o(n), \qquad \text{as } n \to \infty.$$

## Maximal Repeat Finding

The serial solution to this problem also performs a full traversal of the suffix tree. Any pair of maximal repeats will also have the same prefix by definition and so can be found in a single SST. Therefore the average running time is determined as above.

**All Pairs Suffix-Prefix**

**Definition 3.** *Given two strings $S_i$ and $S_j$, any suffix of $S_i$ that matches a prefix of $S_j$ is called a* suffix-prefix match *of $S_i$, $S_j$. Given a set of strings $\mathcal{S} = \{S_1, \ldots, S_r\}$ of total length $n$, the* all-pairs suffix-prefix problem *is the problem of finding, for each ordered pair $S_i$, $S_j$ in $\mathcal{S}$, the longest suffix-prefix match of $S_i$, $S_j$.*

The serial solution to this problem performs a depth-first search of a generalised suffix tree of the input strings recording each suffix-prefix match in one of $r$ stacks. Using a generalised DST the same problem can be solved in parallel in optimal expected time. A full traversal must be made at each SST and each suffix-prefix match has to be recorded. Call $r_z$ the number of strings in the input that have $z$ as a prefix. There can be no more than $r_z r$ suffix-prefix matches at each SST and therefore the running time at each SST is $O(|V_z| + r_z r)$. The time to completion is the maximum of these times and it can shown that

$$\mathbb{E}\left[\max_z(|V_z| + r_z r)\right] = \frac{n + r^2}{k} + o(n + r^2), \qquad \text{as } n, r \to \infty.$$

**Exact Set Matching**

In the exact set matching problem there is a set $\mathcal{P}$ consisting of $r$ patterns to be matched against a text $t$ of length $n$. For simplicity the patterns are assumed to be of equal length $m$ and, to ensure that the number of matches found by each string is bounded, we let $m = \log_\sigma(n/c_0)$. The running time calculation is complicated by the fact that the number of patterns sent to the relevant SST and the size of the associated SST are both subject to random variation. The following bound on the expected running time can be proved.

**Theorem 3.** *Suppose the characters in the input text are i.u.d with alphabet size $\sigma$ and that there are $r$ patterns each with length $m = \log_\sigma(n/c_0)$. Assume the characters in each of the patterns are also i.u.d. with alphabet size $\sigma$ and let $R$ be the maximum running time of the computing nodes in a DST using prefixes of equal length. Then for suitable constants $\lambda$ and $\gamma$,*

$$\mathbb{E}(R) \leqslant \frac{r}{k}(\lambda m + \gamma c_0) + o(r \log n), \qquad \textit{as } n, r \to \infty.$$

## 3 Discussion

The algorithms presented here were implemented and tested on random data. For systematically biased biological data the SSTs may not be as balanced as with random data. This is likely to decrease the work done at some computing nodes at the expense of increasing it at others, thereby reducing the overall parallel efficiency. Using a snapshot of the sequences available for human chromosomes 21 and 22 combined and that of chromosome X we were able to estimate the parallel efficiencies for maximal repeat finding and exact local matching on a DST. We found that the efficiencies were 90 and 82 percent for 4 computing nodes and 72 and 61 percent for 16 nodes. In order to increase these figures we introduced a simple load balancing scheme. An example of how it worked using 16 computing nodes follows.

Instead of considering the 16 different prefixes of length 2 for DNA data we consider all 64 prefixes of length 3. The number of substrings of the input which start with each prefix was counted and, using this information, the set of 64 prefixes was partitioned into 16 subsets as evenly as possible using a simple greedy heuristic. Each computing node was then associated with one of the 16 subsets and a variant of the SSTs described above was constructed. Instead of associating one prefix with each SST, the new structure has a set of prefixes associated with it. So, an SST represented all the suffixes in the input which started with any of the prefixes in the given set. In this way the size of the SSTs was evened out considerably, thereby increasing the overall efficiency. For example, we were able to increase the parallel efficiency of maximal repeat finding on chromosome X using 16 computing nodes from 61 to 89 percent. Simple load balancing schemes for the other problems listed above gave similar improvements in efficiency for real genetic data.

## 4 Acknowledgements

This work was undertaken while the first author was a PhD student at Imperial College London. We gratefully acknowledge his support through an EPSRC studentship. We should also like to thank Costas Iliopoulos, Wojtek Rytter, and the anonymous reviewers for a number of valuable suggestions.

# Bibliography

[1] A. Andersson, N. Larsson, Jesper, and K. Swanson. Suffix trees on words. In *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, LNCS 1075, pages 102–115. Springer-Verlag, 1996.

[2] A. Andersson and S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46:293–300, 1993.

[3] A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume F12 of *NATO ASI Series*, pages 85–96. Springer-Verlag, 1985.

[4] W. I. Chang and E. L. Lawler. Sublinear expected time approximate string matching and biological applications. *Algorithmica*, 12:327–344, 1994.

[5] R. Clifford. *Indexed strings for large-scale genomic analysis.* PhD thesis, Imperial College of Science Technology and Medicine, London, April 2001.

[6] A. Delcher, S. Kasif, R. Fleischmann, J. Peterson, O. White, and S. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[7] B. Dorohonceanu and C. Nevill-Manning. Accelerating protein classification using suffix trees. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 126–133, 2000.

[8] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 693–702, Las Vegas, Nevada, 1995.

[9] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proceedings of the Seventh Annual Symposium on Discrete Algorithms*, pages 373–382, Atlanta, Georgia, 1996.

[10] P. Ferragina and R. Grossi. The string B-Tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):238–280, 1999.

[11] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 1997.

[12] D. Gusfield. *Algorithms on strings, trees and sequences. Computer Science and Computational Biology.* Cambridge University Press, 1997.

[13] D. Gusfield, G. M. Landau, and D. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Information Processing Letters*, 41:181–185, 1992.

[14] J. Kärkkäinen. Suffix cactus : a cross between suffix tree and suffix array. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, LNCS 937, pages 191–204. Springer-Verlag, 1995.

[15] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *COCOON '96, Hong Kong*, LNCS 1090, pages 219–230. Springer-Verlag, 1996.

[16] S. Kurtz. Reducing the space requirement of suffix trees. Report 98–03. Technical report, Technische Fakultat, Universität Bielefeld, 1998.

[17] S. Kurtz and C. Schleiermacher. Reputer: Fast computation of maximal repeats in complete genomes. *Bioinformatics*, 15(5):426–427, 1999.

[18] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990.

[19] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.