The generalised k-Truncated Suffix Tree for time-and space-efficient searches in multiple DNA or protein sequences

Marcel H. Schulz

Institute für Medizinische Genetik,
Charité Universitätsmedizin Berlin,
Augustenburger Platz 1,
13353 Berlin, Germany
E-mail: marcel.schulz@molgen.mpg.de

International Max Planck Research School
for Computational Biology and Scientific Computing,
Berlin, Germany

Sebastian Bauer and Peter N. Robinson*

Institute für Medizinische Genetik,
Charité Universitätsmedizin Berlin,
Augustenburger Platz 1,
13353 Berlin, Germany
E-mail: sebastian.bauer@charite.de
E-mail: peter.robinson@charite.de
*Corresponding author

Abstract: Efficient searching for specific subsequences in a set of longer sequences is an important component of many bioinformatics algorithms. Generalised suffix trees and suffix arrays allow searches for a pattern of length n in time proportional to n independent of the length of the sequences, and are thus attractive for a variety of applications. Here, we present an algorithm termed the generalised k-Truncated Suffix Tree (kTST), that represents an adaption of Ukkonen's linear-time suffix tree construction algorithm. The kTST algorithm creates a k-deep tree in linear time that allows rapid searches for short patterns of length of up to k characters. The kTST can offer advantages in computational time and memory usage for searches for short sequences in DNA or protein sequences compared to other suffix-based algorithms.

Keywords: suffix tree; biological sequence analysis; suffix array; bioinformatics.

Reference to this paper should be made as follows: Schulz, M.H., Bauer, S. and Robinson, P.N. (2008) 'The generalised k-Truncated Suffix Tree for time- and space-efficient searches in multiple DNA or protein sequences', *Int. J. Bioinformatics Research and Applications*, Vol. 4, No. 1, pp.81–95.

Biographical notes: Marcel H. Schulz is now a PhD student at the Max Planck Institute for Molecular Genetics in Berlin, Germany. Previously, he gained his BA in Bioinformatics at the Free University in Berlin and joined the International Max Planck Research School for Computational Biology and Scientific Computing in Berlin. His interests include algorithms for molecular biology and probabilistic models.

Sebastian Bauer is a PhD student at the Institute of Medical Genetics, Charité Universitätsmedizin in Berlin, Germany. He received his Master's Degree in Computer Science from the Technical University of Ilmenau. His main interests are in the fields of discrete algorithms and graphical probability models.

Peter N. Robinson is a Physician and Research Scientist at the Institute of Medical Genetics, Charité Universitätsmedizin in Berlin, Germany. He gained his BA in Mathematics and MSc in Computer Science from the Columbia University in New York City and his MD from the University of Pennsylvania in Philadelphia. His research interests include both wetlab molecular genetics and computational biology.

1 Introduction

Finding specific subsequences[1] in longer DNA or amino-acid sequences is one of the fundamental tasks in computational genomics. The suffix tree is an important data structure whose primary characteristic is that after it has been constructed by preprocessing of the target string or strings in linear time and space, any number of strings can be sought in the target in time proportional only to the length of the search string, independent of the length of the target string. This is a remarkable result considering that important string search algorithms such as Boyer-Moore and Knuth-Morris-Prath require time proportional to the length of the (potentially huge) target sequence for each pattern search (Gusfield, 1997).

A disadvantage of many implementations of the suffix tree algorithm involves the relatively high space (memory) requirements (Kurtz, 1999); additionally, lack of locality of reference can lead to excessive page faulting with reduced performance for very large trees that do not entirely fit into physical memory (Giegerich and Kurtz, 1995). However, optimised implementations of suffix trees can reduce the amount of memory per character of input sequence to roughly 10–20 bytes (Kurtz, 1999).

A suffix array for single string of length m refers to a lexicographically ordered array of all the suffixes of the string. The original implementation required $O(m \log m)$ construction time but was 2–5 times more space efficient than available suffix tree implementations (Manber and Myers, 1990). Depending on the implementation, searching for substrings of length k in suffix arrays could be performed in $O(k \log m)$ or $O(k + \log m)$ runtime, which is asymptotically slower than with suffix trees. More recently, linear-time construction algorithms were presented (Kärkkäinen and Sanders, 2003; Kim et al., 2003; Ko and Aluru, 2003). Enhanced suffix arrays that keep track of more information about the target strings than the standard suffix array can be used

to replace the suffix tree in nearly all applications while retaining linear construction and search times (Abouelhoda et al., 2004).

Kurtz and colleagues presented a 'lazy' Write-Only Top-Down (WOTD) construction for suffix trees that evaluates subtrees only when they are traversed for the first time (Giegerich et al., 2003). This is a fast and space-efficient algorithm that can be regarded as an algorithm of first choice for many applications. WOTD does not use explicit suffix links, and at the moment is not available in an implementation for the analysis of multiple input strings.

Each of the above algorithms is attractive for certain applications. In the present work, we describe the kTST, which is adapted to searching for short DNA or protein subsequences in multiple target sequences. This data structure was motivated by the desire to search a set of thousands of short sequences within a set of larger sequences in an efficient manner, as is for instance required for many types of promoter analysis (DNA) or string kernels (DNA or protein sequences). In the context of searching for short patterns in single or multiple sequences, the kTST described here provides significant savings in space and time compared to standard generalised suffix trees. Two algorithms for construction of truncated suffix trees have been previously presented (Na et al., 2003; Allali and Sagot, 2004), but executables or source code are not available. The present work represents a simplification and generalisation of these algorithms, as will be discussed below. Source code and more detailed description of the kTST algorithm is available at the project homepage[2] under the terms of the GNU Lesser General Public License.

2 Algorithm

We describe an online algorithm for the construction of a kTST based on Ukkonen's algorithm for linear time construction of suffix trees (Ukkonen, 1995). The kTST construction algorithm was implemented by modifications to our version of Ukkonen's algorithm which is based upon the strmat package (Gusfield, 1997; http://www.cs.ucdavis.edu/∼gusfield/strmat.html). In the following sections we review Ukonnen's algorithm for suffix tree construction and then demonstrate how the kTST differs from it.

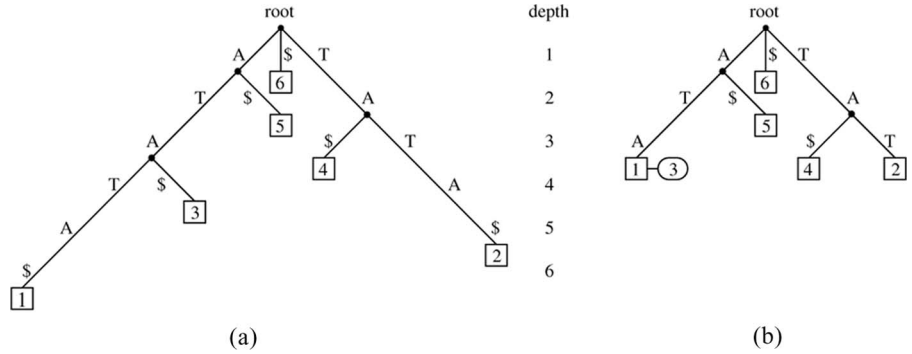
2.1 The suffix tree: background and definitions

Let Σ be a finite alphabet. A suffix tree for an m-character string s over Σ is a rooted directed tree with exactly m leaves numbered 1 to m, corresponding to each suffix of s. Each internal node has at least two children and each edge is labelled with a nonempty substring of s. No two edges out of the same node are allowed to have edge-labels beginning with the same character. For any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of s that starts at position i, i.e., the substring $s_{i..m}$. A unique termination character $\$ \notin \Sigma$ is added to the end of the string to ensure that no suffix is a prefix of any other suffix (Gusfield, 1997).

Figure 1(a) shows the suffix tree for $s = ATATA\$$. Every leaf, indicated as a square, holds the start position of a suffix. Searching for a pattern P can be done in time proportional to the length of P, by traversing the suffix tree from the root, matching

the characters of P in turn. For instance, searching for $P = \text{ATA}$ ends in depth 3; the positions of P in the original string are given by the numbers of the leaves in the subtree beneath the node (see Figure 1(a)). If any character in P does not match on the way down, the pattern is not contained in the tree.

Figure 1 Comparison between standard and truncated suffix trees. A standard suffix tree (a) and a 3-truncated suffix tree (b) were constructed from the string ATATA\$. Both trees store identical information concerning the locations of 3-mer subsequences in the input string; for instance, ATA occurs twice in ATATA\$ and can be found in the standard suffix tree by following the path from the root labelled 'ATA'. Every leaf in the tree below this point corresponds to a match for 'ATA' (leaves 1 and 3). The procedure is similar for the 3-truncated suffix tree: follow the path from the root to leaf 1, and then examine the list of IntLeaves to find the match at position 3. Note that the 3-truncated suffix tree is not intended to be used to find substrings of length greater than 3. The IntLeaves are displayed as an oval connected to the node by a horizontal line



2.2 *Ukkonen's algorithm for suffix tree construction*

Given a string s of length m, a naive construction algorithm for suffix trees would have $O(m^2)$ time complexity, which would be unacceptable for most realistic applications in computational biology. Several more complex linear time construction algorithms are known, of which the most widely cited is due to Ukkonen (1995).

Ukkonen's algorithm works its way from left to right across the string s in a series of m *phases*. In phase i, the substring $s_{1..i}$ and all of its suffixes $s_{2..i}, s_{3..i}, \ldots, s_i$ are inserted into the suffix tree (unless they already appear there). Each phase is thus divided into *extensions*, such that after extension j the string $s_{j..i}$ is included in the tree. Algorithm 1 summarises the core procedure.

The workflow of an extension j of phase $i+1$ can be conceptually divided into two parts. In part A (lines 6–21), Ukkonen's procedure inserts the substring $s_{j..i+1}$ into the suffix tree by extending substring $\beta = s_{j..i}$ (which is already present in the tree) by means of one of the following rules:

Rule 1: The path β from the root extends a leaf of the current tree. To update the tree, simply add the character s_{i+1} to the end of the leaf's label.

Rule 2: The path β does not end at a leaf and no path continuing from β starts with the character s_{i+1}. Then, a new leaf starting from β is created and labelled with s_{i+1}:

(a) if β ends inside an edge a new node is created, to which the new leaf is appended

(b) Otherwise β ends inside a node and the leaf is added as a child of this node

Rule 3: Some path starting from the end of β starts with the character s_{i+1}. In this case, βs_{i+1} is already in the tree and no more work needs be done.

Once a leaf is created by rule 2 and labelled j for the suffix starting at position j of s, then the leaf will remain a leaf in all successive trees created by the algorithm. Therefore, an implementation of Ukkonen's algorithm can avoid the use of rule 1 by (conceptually) inserting the complete suffix into the tree whenever rule 2 applies.[3]

Part B (lines 22–41) is concerned with preparing for the next extension, i.e., finding the substring $s_{j+1..i}$ in the suffix tree. A key idea in Ukkonen's algorithm is the suffix link, which was originally proposed by McCreight (1976). This is a pointer added during the construction of the algorithm that represents a kind of 'short-cut'. Every internal node N_1 of a suffix tree has a suffix link emerging from it such that if $path(N_1)$ is $x\alpha$ (where $x \in \Sigma$), then the suffix link of N_1 points to a node N_2 with $path(N_2) = \alpha$. This allows a trail of suffix links to be followed without having to traverse back to the root each time, thus avoiding redundant character comparisons. The node pointed to by the suffix link of node N is denoted by $N.slink$.

Following the extension, we may have to walk up a path γ to the nearest node N before taking advantage of a suffix link. After getting to $N.slink$ we have to go back down the path γ in order to reach the appropriate place in the tree to apply one of the extension rules to insert the suffix $s_{j..i+1}$. The *skip and count trick* is an efficient way of going down this path. As γ is guaranteed to be already present in the tree we only need to look for the child whose first character matches the first character of γ. We then skip down to the next node or to the end of γ (which may also end in the middle of an edge). We can move from one node to the next with constant-time operations. Therefore, the total traversal time is proportional to the number of nodes going down γ rather than the number of characters. Pseudocode of the whole procedure is given in Algorithm 1.

One further aspect of suffix tree construction needs to be discussed to motivate the kTST. The above descriptions and the pseudocode all refer to the construction of a suffix tree for a single string. In order to construct a generalised version of the suffix tree, which can contain an arbitrary number of strings, every leaf should additionally store an identifier of the source string. Another important change is the use of so-called *IntLeaves*. They are added to a leaf as a linked list as soon as another suffix is encountered which has already been inserted to the tree and merely need to store which strings contain the suffix and what position they have in the respective strings.[4] This leads to a new 'rule' for the insertion of suffixes into generalised suffix trees that we will also make use of for the kTST:

Rule 4: Whenever the insertion of a suffix ends in a node of the tree, create an *IntLeaf* to record string id and position of the suffix and add it to the node.

Algorithm 1: Alterations to the Ukkonen algorithm for the kTST are shown in gray.
ADDSTRING(T,s): Add string s to suffix tree T

```
1:  m ← length of s
2:  lastNode ← root of T, node ← lastNode
3:  j ← 1, depth ← 0
4:  for i ← 1 while i ≤ m + 1 do                                   ▶ phase i
5:    for j while j ≤ i and i ≤ m                                  ▶ extension j

                          Part A: Insert suffix s_{j..i+1} into T
6:      if s_{i+1} isn't contained in tree at current position
7:        if s_{j..i} doesn't end at directly at node then
8:          node ← SPLITEDGE
9:        end if
10:       Add leaf to node with edge label starting with s_{i+1}   ▶ Rule 2/2*
11:       lastNode.slink ← node, lastNode ← node
12:     else
13:       Move down one character along edge                       ▶ Rule 3
14:       depth ← depth +1
15:       if depth = k then
16:         Add new IntLeaf                                        ▶ Rule 4
17:         i ← i + 1
18:       else
19:         break ▶ Leave inner loop, begin next phase
20:       end if
21:     end if

          Part B: Prepare for next extension: update current position to s_{j+1..i}
22:     if node ≠ root of T
23:       if s_{j..i} ends directly at node and node has a suffix link then
24:         node ← node.slink
25:         depth ← depth − 1
26:       else
27:         xη ← label between current position and node.parent    ▶ x ∈ Σ
28:         node ← node.parent
29:         depth ← depth − 1
30:         if node ≠ root of T
31:           node ← node.slink
32:           γ ← xη
33:         else
34:           γ ← η
35:         end if
36:         Use skip&count to move back down via γ                 ▶ This alters node
37:         if current position is node and lastNode has no suffix link then
38:           lastNode.slink ← node, lastNode ← node
39:         end if
40:       end if
41:     end if

42:   end for
43: end for
```

2.3 Properties of k-Truncated Suffix Trees

A standard suffix tree for any given s has a maximal string-depth[5] corresponding exactly to the length of s. For many purposes, much of the information in the deeper realms of the suffix tree is not of direct interest. For standard suffix trees, existential searches for substrings of fixed length k (k-mers or k-factors) do not require traversals deeper than k. But if one wants to know all the positions in which the substring was found (enumerative query), then traversal of the subtree below the position of the match is required.

The kTST contains all suffixes of all k-mers of the input string. For instance, the first k-mer of s is $s_{1..k}$ and the corresponding suffixes are $s_{2..k}, \ldots, s_{k-1..k}, s_k$. In contrast, the suffixes in a standard suffix tree are always suffixes of the original input string. Figure 1(b) shows the 3-truncated suffix tree for $s = \text{ATATA\$}$ and $k = 3$.

Similar to the suffix tree, searching for up to k-length patterns can be performed in time proportional to the length of the pattern, irregardless of the length of the input string. In case of multiple matches of some k-mer subsequence, additional IntLeaves analogous to those described above are added to internal nodes or leaves and displayed.

Note that searching for the pattern $P = \text{ATA}$ yields identical results in both trees but that the k-truncated suffix tree requires less memory. In addition, there is no need to traverse any subtree to find all occurrences of the pattern.

2.4 k-Truncated Generalised Suffix Tree construction

There are several conceivable strategies for building a truncated suffix tree. It is possible to delete subtrees from a completely constructed suffix tree. Obviously, this is very unattractive with respect to memory during the construction phase. As an adaptation of the naive suffix construction algorithm, one can also build the tree by inserting every k-mer of the input strings into the tree resulting in a time bound of $O(km)$. We instead propose an approach inspired by Ukkonen's suffix tree construction algorithm.

Only a few simple changes to Ukkonen's algorithm (shown in gray in Algorithm 1) are required to construct the kTST while maintaining time and space linearity. The kTST is constructed from left to right by moving a window of size of not greater than k over the input sequence s: $s_{1..k}, s_{2..k+1}, \ldots, s_{i..i+k-1}, \ldots, s_m$. The current string depth is recorded in the variable *depth* (Algorithm 1, lines 3, 14, 25 and 29). This has two key consequences.

Firstly, in the standard Ukkonen algorithm, whenever a new leaf is created the entire suffix is (conceptually) inserted. In the kTST, a substring of up to k characters is inserted. This leads to a modified version of rule 2:

Rule 2*: Whenever a new leaf is created, only the k-mer truncated suffix is entered into the tree.

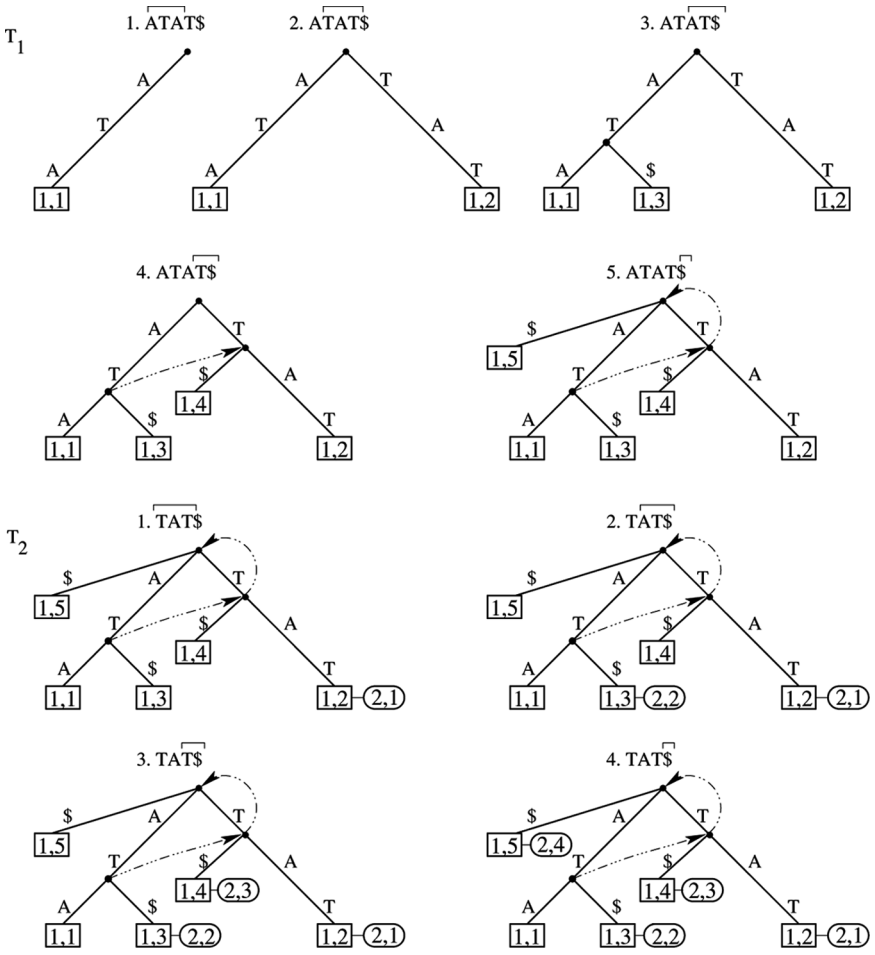
Secondly, in the kTST, once the algorithm has reached depth k (line 15–18) we now insert the k-mer string $s_j..s_{j+k}$ into the tree as, by definition, the kTST does not contain substrings with more than k characters. Then, the next string to be inserted is $s_{j+1}..s_{j+k+1}$ which requires advancing to the next extension (j), so that the inner loop is not exited by the `break` command as in the standard algorithm. Instead, the phase variable i is incremented by one and the algorithm continues through part B in order

to arrive at $s_{j+1}..s_{j+k}$ at the end of the extension. Note that this string exists within the tree, as it has been inserted in the previous phase.

Finally, we note that rule 4 functions identically as in the standard suffix tree, but it is used much more often in the kTST, because the chance of having identical k-mer subsequences is much higher than the chance of having identical suffixes of arbitrary length. That is, the kTST will tend to have a much higher number of IntLeaves and a smaller number of nodes than the suffix tree. Since IntLeaves require less memory than nodes, the net result is usually a significant improvement in time and memory efficiency.

Figure 2 offers an example of how a kTST is constructed. Two strings, $T_1 = \text{ATAT\$}$ and $T_2 = \text{TAT\$}$, are inserted by consecutive calls to the procedure ADDSTRING (Algorithm 1).

Figure 2 Construction of the generalised 3-truncated suffix tree for two strings $T_1 = \text{ATAT\$}$ and $T_2 = \text{TAT\$}$. Leaves are shown as squares and IntLeaves as ovals. In each case the first number refers to the string and the second to the position in the string at which a given substring was found. Suffix links are shown as dashed lines. See text for details



To insert T_1, the first two substrings of length k, ATA and TAT, are inserted directly by rule 2* (line 10). For AT$, we have to walk down the edge and create a new node at the diverging edge-label by rule 2 (lines 8–10). The next substring T$ is added after walking up to the root (line 26) and one character down (line 36), and create a node for T$ by rule 2 (lines 8–10). This is the first time when a suffix link is created (line 11). The last substring of T_1, $, is represented by leaf $(1, 5)$ out of the root (rule 2, line 10).

It is instructive to consider the steps involved in inserting the second string T_2 (which is a substring of T_1) into the suffix tree. To insert the first k-mer subsequence TAT we walk down the edge from the root that begins with the character 'T' (line 13). When we arrive at the leaf $(1, 2)$, which records the k-mer substring beginning at position 2 of the first string T_1, we add an IntLeaf $(2, 1)$ to record the occurrence of the pattern TAT at position 1 of the second string T_2 (rule 4, line 16). In the following step, we add the substring AT$ to the tree and add the IntLeaf $(2, 2)$ to the tree in a similar fashion. To add T$ to the tree, we do not need to return to the root but can follow the suffix link from the parent node (lines 23–24), and then add another IntLeaf $(2, 3)$ by rule 4. In the final step, we follow the suffix link to the root and follow the edge labelled with the character $ and finally add an IntLeaf $(2, 4)$.

By use of the ideas of Ukkonen's suffix tree construction algorithm including suffix links, the skip-count trick and edge-label compression, construction of a k-truncated suffix tree algorithm can be achieved with a time and space bound linear in the length of the input sequences. The proof given in Gusfield (1997), Allali and Sagot (2004) and Ukkonen (1995) applies equally to our algorithm, and we present empirical results to support this further on.

3 Results and discussion

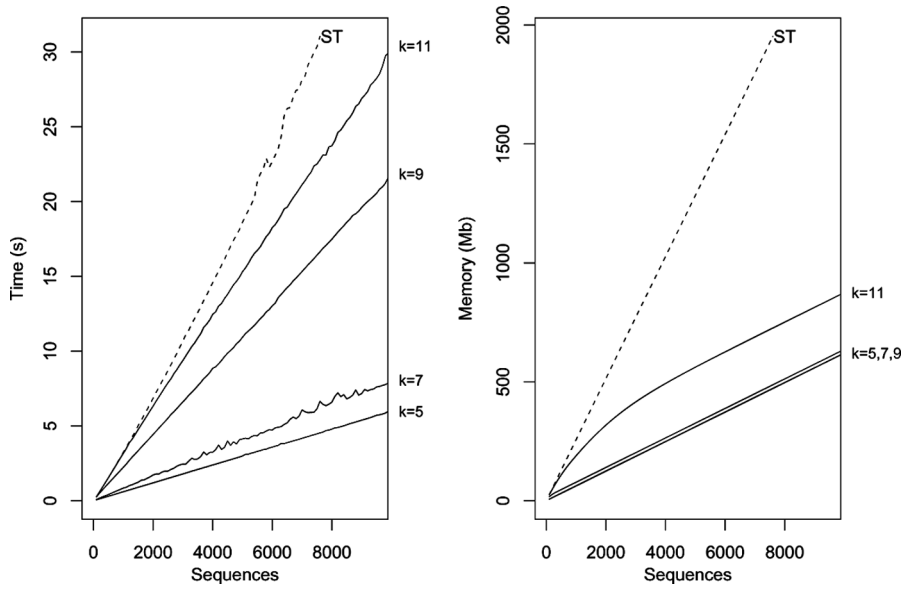
In this work, we present the kTST, a data structure based on the suffix tree that allows linear time searching for up to k-mer search strings. We implemented a standard generalised suffix tree based on Ukkonen's algorithm and used this as a benchmark against which to test our implementation of the kTST. The implementations were identical except for the modifications of the tree construction algorithm we designed for the kTST. All tests were performed on a desktop PC with a 3.0 GHz Pentium IV CPU with 1MB cache and 2 Gb memory with a Debian Linux system using a 2.6 kernel.

Time and memory requirements for tree construction were compared by building generalised trees for n random DNA sequences with equal nucleotide frequencies of 5000 nucleotides each, where $n = 100, 200, \ldots, 10,000$. kTSTs with $k = 5, 7, 9, 11$ were compared with a standard suffix tree with respect to memory usage and time of execution. Figure 3 shows that the truncated suffix trees have a time and space advantage over standard trees at every value of k and for all number of sequences tested. As k increases, the behaviour of the truncated suffix tree with respect to time and memory requirements approaches that of a standard suffix tree. Note that the curves for the standard suffix tree (dotted line) are incomplete, because excessive page faulting (thrashing) caused the operating system to terminate the program for more than 7800 sequences.

The kTST achieves faster enumerative search times for cases where multiple instances of a given k-mer sequence are present in the tree because they are stored in a linked list of IntLeaves. The full suffix tree requires traversing the subtree beginning

at the position representing the k-mer string in the tree. This advantage becomes somewhat less as k is increased because the proportion of IntLeaves to nodes is reduced (Figure 4).

Figure 3 Runtimes and memory usage for the construction of standard and k-truncated suffix trees for $k = 5$, 7, 9, and 11. Up to 10,000 sequences with 5000 random nucleotides (equal nucleotide frequencies) were analysed. The runtime and memory performance for increasing number of sequences are shown

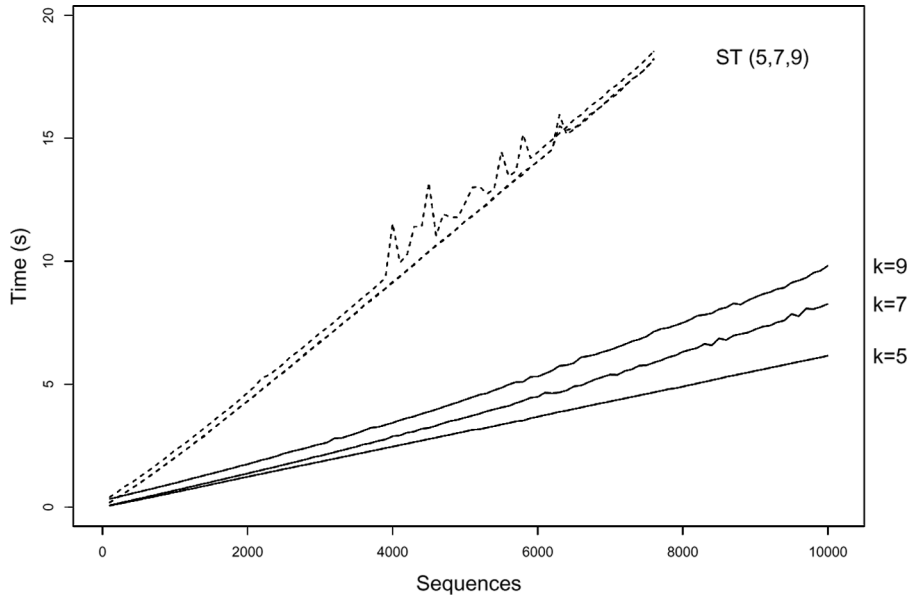


For the purpose of comparison with the kTST, we implemented the enhanced Suffix Array (eSA) and adapted it for multiple strings to create a generalised eSA by concatenating all input strings and creating another array to store the string index for every position. For construction of the eSA, we used the deep shallow sorting algorithm (Manzini and Ferragina, 2004), which clearly outperforms linear time bounded algorithms in practice (Puglisi et al., 2005). We performed similar tests as described above to test the time and space complexity of the generalised eSA. We note that while it was possible to perform a direct comparison of the characteristics of the truncated and standard suffix trees because the underlying implementation was identical except for the modifications needed to construct a truncated tree, the results of the comparison of the kTST against the eSA need to be interpreted with caution because there are many ways of implementing the eSA with different time and space trade-offs. For instance, the memory saving tricks of Abouelhoda et al. (2004) are implemented in gsuffix and can be optionally enabled within the library, but we did not use them for the comparisons shown here because of the slower running times. Given these caveats, the kTST showed competitive results with respect to speed of construction and memory requirements (Table 1).

We note that for simple applications on DNA sequences, a direct-access table algorithm is a good alternative. Averaged over all tests (from 100 to 10,000 sequences of 5000 nucleotides each) and $k = 5, 7, 9, 11$, the ratio of the build time for the kTST

compared to the direct-access table was 3.03, the ratio for the time needed to search for all possible k-mer sequences was 1.31, and the ratio for the memory usage was 1.33. Similarly, hashing is a reasonable alternative for DNA sequences. For the purposes of comparison, we implemented one-level hashing with the sdbm hash function. In general, this implementation performs competitively for DNA sequences with small values of k. For protein sequences and larger values of k, search speed deteriorated dramatically with increasing amounts of data, because it is necessary to search through the entire linked list belonging to the hash bucket (data not shown).

Figure 4 Search times to search enumeratively for all 4^k possible k-mers for $k = 5$, 7 and 9. The k-truncated suffix trees are shown as solid lines, and the standard suffix trees with dashed lines. The sequences were the same as those used in Figure 3. Testing could not be continued beyond 7800 sequences for the standard suffix tree because excessive page faulting caused the operating system to terminate the program



In the C language, the data type `int` is generally represented by four bytes and a `short` by two bytes (although this is compiler dependent and can differ from machine to machine). An `unsigned short` can thus hold $2^{16} = 65536$ distinct values compared to $2^{32} = 4294967296$ values for a four-byte `int`, In order to construct a suffix tree with up to 65536 different sequences, each of which is up to 65535 characters long, it is sufficient to use a two-byte `short`. This is an option in our implementation. The two-byte version of the kTST showed reduced memory usage (on average, 74% of that of the four-byte version on the data analysed in Figures 3 and 4) and identical or slightly better build and search times (see also Table 1).

A tree built from strings over the DNA alphabet contains at most 4^k nodes. For all k-mers that occur more than once, a linked list of IntLeaves is created. For large datasets, much of the information is thus held in the IntLeaves, and nodes can have

a linked list of up to hundreds of IntLeaves representing the sequences and positions of all the occurrences of the k-mer sequence of the node. It can be more efficient to store multiple occurrences within a single 'multi-IntLeaf', because of two reasons. Firstly, we save some space because less pointers must be stored. Secondly, there is more locality of reference and less cache misses, resulting in faster execution times. As expected, multi-IntLeaves were advantageous especially for large amounts of input data (see Table 1).

Table 1 Results of analysis of several typical biological datasets with the suffix algorithms implemented in gsuffix. Building and search time is given in seconds, memory consumption is given in MB. A 6-TST was tested for Human chromosomes 7 and 12 with and without 20-fold multi-IntLeaves. Search time was measured by searching for all possible 6-mer DNA sequences. The best result in each test is shown in italic. Proteome refers to the set of all known and predicted human proteins downloaded from Ensembl (http://www.ensembl.org). To test the search time, 100,000 random 6-mer protein sequences were generated and sought in the input sequences. Finally, 2000 nucleotides of 20647 upstream sequences for RefSeq genes were downloaded from the UCSC Genome Bioinformatics Site (http://genome.ucsc.edu), and a 6-TST was again compared to standard suffix trees and enhanced suffix arrays. A '–' indicates that the operating sc terminated the test because of lack of memory

	Chromosome 7			*Chromosome 12*		
Datastructure	*Build*	*Search*	*Memory*	*Build*	*Search*	*Memory*
kTST	*33.09*	21.13	1817.17	*27.56*	17.12	1517.33
kTST (20-fold multi)	37.74	*3.53*	*1272.40*	32.11	*2.95*	*1062.52*
ST (4-byte)	–	–	–	–	–	–
ST (2-byte)	–	–	–	–	–	–
SA	–	–	–	–	–	–

	Proteome			*Upstream sequences (2000 nt)*		
Datastructure	*Build*	*Search*	*Memory*	*Build*	*Search*	*Memory*
kTST (4-byte)	50.00	*0.59*	444.36	8.92	5.34	473.49
kTST (2-byte)	49.53	*0.59*	*395.61*	*8.68*	5.28	*315.77*
ST (4-byte)	43.37	0.62	697.93	49.79	17.80	1867.92
ST (2-byte)	43.18	0.61	678.38	47.45	17.62	1823.36
SA	*19.35*	0.64	467.95	49.04	*4.14*	1142.70

The kTST algorithm has similarities to an algorithm proposed by Na and colleagues for single strings in the context of data compression (Na et al., 2003) and to the at-most k-deep factor tree proposed by Allali and Sagot (2004). In contrast to Na et al. (2003), the kTST and its construction algorithm allows for enumerative queries in multiple target sequences. The main difference between the kTST algorithm presented here and the at-most k-deep factor tree is that the latter algorithm relies on automatically extending the leaves by one character after each phase and holding the leaves in a queue until the paths they represent reach a length of k. The kTST avoids this extra work by means of directly extending the leaves to a length of k when they are first created.

Another difference between the two algorithms is that to date, a generalisation of the at-most k-deep factor tree has been introduced for the strings but not the position of the k-mers within the input strings. The generalisation scheme involves saving a boolean array with one slot for each input string (Carvalho et al., 2004), which is not efficient for high numbers of input sequences and large values of k. In contrast, the kTST stores both the string id and the position using a linked list of IntLeaves, as described above. Source code or executables for Na et al. (2003) and Allali and Sagot (2004) were not available for direct comparisons.

Table 2 provides an overview of salient properties of the several important string algorithms that can be used to search for substrings in multiple input strings. The enhanced suffix array (Abouelhoda et al., 2004) and WOTD (Giegerich et al., 2003) are the fastest and most flexible algorithms for general searches. The suffix tree can be used to efficiently implement a number of algorithms of great importance in string-based analysis of biological sequences, such as searching for the longest common subsequence in two or more sequences, or searching for tandem repeats (Gusfield, 1997). The kTST can be used for many of these algorithms as well, with the limitation in maximal length to k. The kTST offers some advantages in speed and memory for certain datasets. For simple searches for k-mer subsequences in DNA sequences, the direct-access table is a quick and simple alternative to suffix-based algorithms, however, this algorithm as well as hashing based algorithms are less flexible than the suffix based algorithms for applications that might search for subsequences of varying length (up to k) or might take advantage of suffix links or of the tree structure inherent in the kTST or standard suffix tree (which can also easily be simulated with the eSA).

Table 2 Properties of the String Algorithms Implemented in suffix: *DNA sequences*, *Amino acid sequences*, *ASCII Sequences*: The algorithm can (+) or can not (–) be used to analyse the respective type of sequence ('*' indicates that suffix links can be simulated but are not explicitly used in the data structure). *Dynamic*: The data structure can be dynamically changed after initial construction, e.g., new strings can be added. *Searches for 1-k-mer subsequences*: The data structure can be used to search for substrings with a length from 1 to k characters. *Searches for arbitrary strings.* The data structure can be used to search for sequences of any length. *Suffix algorithms.* The data structure can be used to implement algorithms taking advantage of the suffix tree. Abbreviations: ar: array implementation of suffix tree/kTST; ll: linked-list implementation; DAT: direct-access table; eSA: enhanced suffix array

Property	*kTST-ar*	*kTST-ll*	*ST-ar*	*ST-ll*	*eSA*	*DAT*	*Hash*
DNA sequences	+	+	+	+	+	+	+
Amino acid sequences	–	+	–	+	+	+	+
ASCII strings	–	+	–	+	+	+	+
Dynamic	+	+	+	+	–	+	+
Searches for 1- to k-mer subsequences	+	+	+	+	+	–	–
Searches for arbitrary strings	–	–	+	+	+	–	–
Suffix algorithms	+	+	+	+	*	–	–

Source: Abouelhoda et al. (2004)

We have implemented the kTST, standard suffix trees and the enhanced suffix array in a C library. As examples of applications for which the kTST is useful,

we have implemented a non-parametric procedure for finding k-mer substrings in promoter sequences that are significantly associated with expression differences measured by microarrays (Mootha et al., 2004) and a k-spectrum kernel for support vector machine analysis of distant protein homologs (Leslie et al., 2002). The library and the applications are available at the project homepage `http://gsuffix.sourceforge.net`.

References

Abouelhoda, M.I., Kurtz, S. and Ohlebusch, E. (2004) 'Replacing suffix trees with enhanced suffix arrays', *Journal of Discrete Algorithms*, Vol. 2, pp.53–86.

Allali, J. and Sagot, M-F. (2004) *The at Most k-Deep Factor Tree*, Technical Report No. 2004–2003, Institut Gaspard Monge, Université de Marne la Vallée.

Carvalho, A.M., Oliveira, A.L., Freitas, A.T. and Sagot, M-F. (2004) 'A parallel algorithm for the extraction of structured motifs', *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, Nicosia, Cyprus, pp.147–153.

Giegerich, R. and Kurtz, S. (1995) 'A comparison of imperative and purely functional suffix tree constructions', ESOP'94: Selected papers of ESOP'94, *the 5th European symposium on Programming*, Edinburgh, Scotland, pp.187–218.

Giegerich, R., Kurtz, S. and Stoye, J. (2003) 'Efficient implementation of lazy suffix trees software', *Practice and Experience*, Vol. 33, pp.1035–1049.

Gusfield, D. (1997) *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, New York, NY, USA.

Kärkkäinen, J. and Sanders, P. (2003) 'Simple linear work suffix array construction', *Proc. 30th International Conference on Automata, Languages and Programming*, The Netherlands, pp.943–955.

Kim, D.K., Sim, J.S., Park, H. and Park, K. (2003) 'Linear-time construction of suffix arrays', *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, pp.186–199.

Ko, P. and Aluru, S. (2003) 'Space efficient linear time construction of suffix arrays', *Lecture Notes in Computer Science*, Vol. 2676, pp.200–210.

Kurtz, S. (1999) 'Reducing the space requirement of suffix trees', *Softw. Pract. Exper.*, Vol. 29, pp.1149–1171.

Leslie, C., Eskin, E. and Noble, W.S. (2002) 'The spectrum kernel: a string kernel for SVM protein classification', *Pac. Symp. Biocomput.*, pp.564–575.

Manber, U. and Myers, G. (1990) 'Suffix arrays: a new method for on-line string searches', *SODA'90: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco, California, USA, pp.319–327.

Manzini, G. and Ferragina, P. (2004) 'Engineering a lightweight suffix array construction algorithm', *Algorithmica*, Vol. 40, pp.33–50.

McCreight, E.M. (1976) 'A space-economical suffix tree construction algorithm', *J. ACM*, Vol. 23, pp.262–272.

Mootha, V.K., Handschin, C., Arlow, D., Xie, X., St. Pierre, J., Sihag, S., Yang, W., Altshuler, D., Puigserver, P., Patterson, N., Willy, P.J., Schulman, I.G., Heyman, R.A., Lander, E.S. and Spiegelman, B.M. (2004) 'Errα and Gabpa/b specify PGC-1α-dependent oxidative phosphorylation gene expression that is altered in diabetic muscle', *Proc. Natl. Acad. Sci. USA*, Vol. 101, pp.6570–6575.

Na, J.C., Apostolico, A., Iliopoulos, C.S. and Park, K. (2003) 'Truncated suffix trees and their application to data compression', *Theor. Comput. Sci.*, Vol. 304, pp.87–101.

Puglisi, S.J., Smyth, W.F. and Turpin, A. (2005) 'The performance of linear time suffix sorting algorithms', *Data Compression Conference (DCC'05)*, Snowbird, Utah, USA, pp.358–367.

Ukkonen, E. (1995) 'On-line construction of suffix trees', *Algorithmica*, Vol. 14, pp.249–260.

Notes

[1]We use the term subsequence in the sense of a substring of contiguous characters in this paper in conformance with accepted usage in molecular biology.

[2]http://gsuffix.sourceforge.net/

[3]Ukkonen's algorithm is online, so the entire suffix is not available when insertion is performed; instead, an open edge is created which grows with the input string. In C, this is implemented with a pointer. In the following text, we will mean this when we write that an entire suffix is conceptually inserted into the string. In fact, this is one of the key ideas in Ukkonen's algorithm – the insertion of an open edge – without which linearity would not be possible.

[4]Alternatively, all input strings can be concatenated into a single string whereby individual sequences are separated from one another by a symbol that does not occur in the sequences themselves. By using binary search, retrieving the sequence and position within it for searching then requires $O(\log N)$ post processing, where N is the number of input sequences. Our implementation currently stores the string ids explicitly to minimise search time, but could be easily adapted for the other method.

[5]We use 'string-depth' to refer to the number of characters on the path from the root of the tree to a leaf, as opposed to the usual notion of depth that denotes the number of edges on the path from the root to a leaf.

Website

http://gsuffix.sourceforge.net