Sparse Suffix Trees*

Juha Kärkkäinen and Esko Ukkonen

Department of Computer Science, P.O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, Finland.
Email: {Juha.Karkkainen,Esko.Ukkonen}@cs.Helsinki.FI

Abstract. A sparse suffix tree is a suffix tree that represents only a subset of the suffixes of the text. This is in contrast to the standard suffix tree that represents all suffixes. By selecting a small enough subset, a sparse suffix tree can be made to fit the available storage, unfortunately at the cost of increased search times. The idea of sparse suffix trees goes back to PATRICIA tries. Evenly spaced sparse suffix trees represent every kth suffix of the text. In the paper, we give general construction and search algorithms for evenly spaced sparse suffix trees, and present their run time analysis, both in the worst and in the average case. The algorithms are further improved by using so-called dual suffix trees.

1 Introduction

Finding an index for a long text that makes fast string matching possible is one of the very central problems of text processing systems. Suffix trees offer a theoretically time-optimal solution. A suffix tree is a trie-like data structure that represents all suffixes of a text. It can be constructed in time linear in the length of the text [16, 13, 15]. With the help of the suffix tree it is possible to find all occurrences of a given string ('keyword') in the text in time that is linear in the length of the string and in the number of the occurrences. Because of such strong properties, suffix trees are used as essential building blocks in several string matching algorithms [4].

Although linear in size, a suffix tree can be too large to be really attractive in practical applications. The size depends on implementation details and the structure of the text, but will never be as low as $10n$ bytes, where n is the size of the text. Suffix arrays [11, 6] (size $5n$ bytes), level-compressed tries [2, 3] (size about $11n$ bytes), suffix cactuses [8] (size $9n$ bytes), and suffix binary search trees [7] (size about $10n$ bytes) are alternative smaller data structures with almost the same properties as the suffix tree. Their space requirement is still high for large texts. As the text in many applications (natural language processing, biocomputing) can be very long, such a high space requirement can make it impossible to accommodate the entire tree in the fast memory. In this case the slow secondary memory operations can, in practice, destroy the good theoretical performance. Hence there is a need to find small alternatives for suffix trees, even at the cost of increased search times.

* A work supported by the Academy of Finland.

In this paper we study *sparse suffix trees* (SSTs), suffix trees that contain only a subset of all the suffixes of the text. For example, if the text is natural language, one could want to represent only the suffixes that start from the beginning of each word (instead of each character) [6]. The idea already appears in [14]. Such an *unevenly spaced* SST can be constructed either through the full suffix tree, at the cost of extra space, or by brute force construction, at the cost of extra time. Recently, Andersson et. al [1] have presented fast construction algorithms working in small space.

Another natural variation is to represent every kth suffix for some fixed k. Such an *evenly spaced* SST can be constructed directly, in linear time, using a modified version of the classical suffix tree construction algorithm. Obviously, the size of an evenly spaced SST is $O(n/k)$ where n is the length of the text. By increasing k one can make the tree arbitrarily small, so that it can be stored into the available memory. Unfortunately, this can only happen at the cost of increased string matching times. There is a trade-off between the size of an SST and the search time for finding the occurrences of a given pattern string using the SST. We develop a search algorithm for an evenly spaced SSTs and show that its expected running time for a random text (in the uniform Bernoulli model) is

$$O\left(\frac{n}{c^m} + k \min\left\{m, \log_c \frac{n}{k}\right\} + \min\left\{mc^{k-1}, nc^{k-m}, \sqrt{mnc^{k-m}}\right\}\right),$$

where m is the length of the pattern, n is the length of the text, and c is the size of the alphabet.

A totally different, word-oriented approach to the indexing problem is described in [12]. We expect our approach to be useful in applications in which the strings to be searched from the text are relatively long and there is no natural word-like structure in the text.

2 Preliminaries

Let $T = t_0 t_1 \ldots t_{n-1}$ be a string over alphabet Σ. The *length* of T is $|T| = n$. A *substring* T_i^j of T is a string $t_i t_{i+1} \cdots t_{j-1}$ for some $0 \le i \le j \le n$. The string $T_i = T_i^n = t_i \ldots t_{n-1}$ is a *suffix* of string T and the string $T^j = T_0^j = t_0 \ldots t_{j-1}$ is a *prefix* of string T.

Let string T of length n be the *text* and string P of length m the *pattern*. The problem of *string matching* is to find the occurrences of string P as a substring of T. It can be solved in linear time by scanning text T using, e.g., the Knuth-Morris-Pratt algorithm [9]. For a large static text, a faster solution can be achieved by preprocessing the text.

A *suffix tree* of text T is a compacted trie for the suffixes of T. Fig. 1 shows an example of a suffix tree. A node v of suffix tree represents the string that is formed by catenating the strings within the nodes on the path from the root to node v inclusive. Each internal node has at least two children and the string within each child starts with a different character. Let $\overline{S}$ denote a node representing the string S. The same notation is also used for any point within the nodes, that is,

$\overline{S}$ denotes the point such that the path from root to that point spells out S. A node $\overline{R}$ is said to contain a point $\overline{S}$ if $\overline{S}$ lies within the node $\overline{R}$. As an example, the rightmost leaf in Fig. 1 is $\overline{\texttt{cca}}$ and it contains point $\overline{\texttt{cc}}$. There is no node $\overline{\texttt{cc}}$.

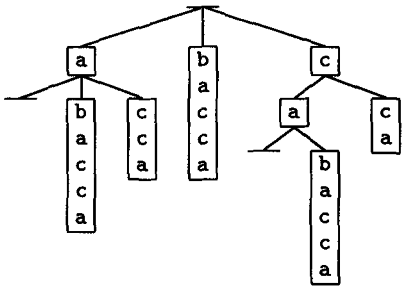

Fig. 1. The suffix tree of string cabacca.

Using suffix trees, all suffixes with prefix P can be found in time $O(m + l)$, where l is the number of suffixes in the result. This gives all the occurrences of P, because every substring of T is a prefix of some suffix, i.e., $T_i^j = (T_i)^j$.

The strings within the nodes of a suffix tree are represented by pairs of pointers to the text in constant space, making the size of the suffix tree linear in $|T|$. Suffix trees can be constructed in linear time [16, 13, 15]. The construction algorithms utilize extra pointers, called the *suffix links*, between the nodes. From an internal node $\overline{S}$ there is a suffix link to node $\overline{S_1}$.

3 Sparse Suffix Trees

As all internal nodes of a suffix tree have at least two children, the size of the suffix tree is linear in the number of leaves, i.e., the number of suffixes in the tree. A significant reduction in the size can be achieved by including only a subset of the suffixes. We call such a tree a *sparse suffix tree.* A suffix tree containing all suffixes is *full.* Fig. 2 shows two examples of sparse suffix trees.

Let us first consider a general sparse suffix tree containing some arbitrary subset of suffixes. For a sparse suffix tree SST, we call these suffixes the SST-*suffixes.* The starting points of these suffixes in the text are called the *suffix points.* A sparse suffix tree can be used to find efficiently all occurrences of a pattern P starting at the suffix points. However, an arbitrary sparse suffix tree may not be of much help in finding all the other occurrences.

To achieve sublinear full string matching over a sparse suffix tree SST, we need to put some restrictions on the set of SST-suffixes. One possibility is to put an upper limit on the distance between two adjacent suffix points. This will limit the distance from an occurrence to the closest suffix point. String matching over sparse suffix trees will be discussed further in Sect. 5.

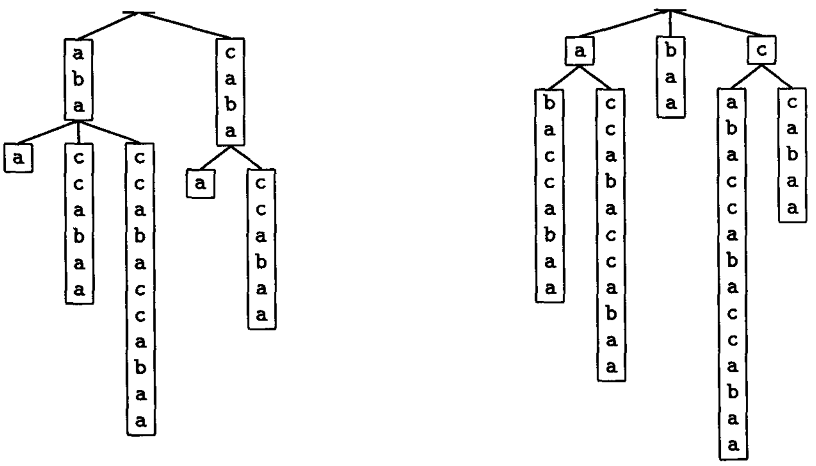

Fig. 2. Two sparse suffix trees over string **cabaccabaccabaa**. The tree on the left contains all suffixes that start after symbol **c**. On the right is a 3-spaced suffix tree.

Another drawback of general sparse suffix trees is their construction time. The linear time construction of full suffix trees relies heavily on suffix links. In a full suffix tree there is a suffix link from $\overline{S}$ to $\overline{S_1}$. A key fact is that the node $\overline{S_1}$ always exists (see Lemma 3 below). This is no more true in SSTs. As a result, the linear time construction algorithm does not work for SSTs. Of course, we can always construct the full suffix tree first and then prune it to get the sparse suffix tree. This can be done in $O(n)$ time, where n is the length of the text, but it also needs $O(n)$ space.

A trivial construction algorithm for sparse suffix trees adds the suffixes to the tree one at a time following the path from root to the point where the new leaf is added. The construction works in just $O(N)$ space for a sparse suffix tree of N suffixes. The construction time is, in the worst case, linear in the total length of the suffixes, which is $O(Nn)$. By the results of Apostolico and Szpankowski [5], the expected construction time for random text and randomly selected suffixes is $O(n+N\log N)$. Recently, Andersson et. al [1] have described a more complicated $O(n)$ time and $O(N)$ space construction algorithm.

4 Evenly Spaced Sparse Suffix Trees

An *evenly spaced sparse suffix tree* contains every kth suffix of the text for some positive integer k. We will also use the term *k-spaced suffix tree.* Fig. 2 shows an example of a 3-spaced suffix tree. The even spacing helps full string mathing over the tree as we will see in Sect. 5. In this section we will show how to modify the linear time construction algorithm for full suffix trees to work for evenly spaced suffix trees. Essentially the same algorithm was presented in different context in [10].

For the construction algorithm we need suffix links.

Definition 1. In a k-spaced suffix tree, there is a suffix link from an internal node $\overline{S}$ to root, if $|S| \leq k$, and to another internal node $\overline{S_k}$ otherwise.

Note that for $k = 1$, Definition 1 coincides with the definition of suffix links for full suffix trees.

The node $\overline{S_k}$ always exist as we shall next prove. We will need the following well-known lemma characterizing the set of internal nodes of suffix trees. The lemma holds for all suffix trees including general sparse suffix trees.

Lemma 2. *Let* SST *be a sparse suffix tree over text T. The tree* SST *has a node $\overline{S}$ if and only if there exist two* SST*-suffixes T_i and T_j such that the longest common prefix of T_i and T_j is S.*

Now we can show that the node pointed to by a suffix link in Definition 1 always exists.

Lemma 3. *If a k-spaced sparse suffix tree* SST *has an internal node $\overline{S}$, $|S| > k$, then* SST *has an internal node $\overline{S_k}$.*

Proof. By Lemma 2 there exists two SST-suffixes T_i and T_j such that S is the longest common prefix of the suffixes. Both suffixes must be at least as long as S, i.e., longer than k. Thus suffixes T_{i+k} and T_{j+k} also exists and, due to k-spacing, are SST-suffixes. The longest common prefix of T_{i+k} and T_{j+k} is S_k and thus SST has the node $\overline{S_k}$. □

The different definition of suffix links is the only major modification needed for the construction algorithm to work for evenly spaced sparse suffix trees. We will omit here the details of this quite complicated algorithm and refer to [13, 15]. The resulting algorithm constructs a k-spaced SST in $O(n)$ time and $O(n/k)$ space.

5 String Matching

Efficient string matching in full suffix trees utilizes the fact that every substring of the text is a prefix of some suffix. However, a sparse suffix tree SST does not contain all suffixes and thus we have to modify our approach to find those occurrences of the pattern that are not prefixes of an SST-suffix.

The situation for a single occurrence is depicted in Fig. 3. The start of the occurrence is between two suffix points. An obvious solution is to use a suffix starting at one of these suffix points to locate the occurrence. This gives us the following two basic methods for finding the occurrence of pattern P.

Method 1: Find a suffix T_i such that the pattern occurs in the suffix after some arbitrary prefix of length g, i.e., $T_{i+g}^{i+g+m} = P$.

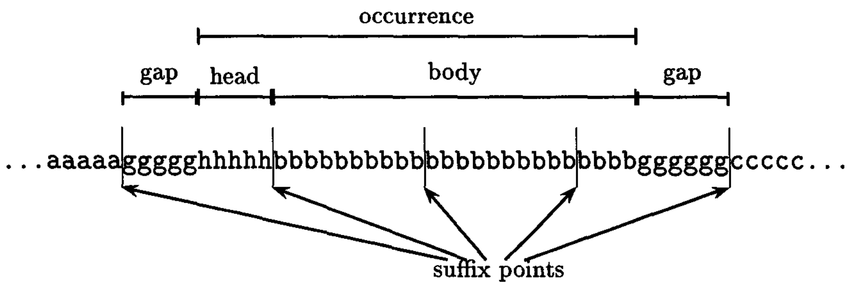

Fig. 3. An occurrence of a pattern in a text.

Method 2: Find a suffix T_i such that it has a prefix T_i^j that is a suffix P^h of the pattern P. Check whether $T_{i-h}^i = P_h$.

A problem with both methods is that, for one occurrence, there may be many suffixes satisfying the condition. Of course, we want to find the suffix with the closest starting point, but in an arbitrary sparse suffix tree it is not easy to limit searching to only the closest suffixes. However, a limit k on the maximum distance between adjacent suffix points guarantees that for each occurrence there is a suffix point within distance $k-1$ in each direction. Furthermore, in the case of k-spaced suffix trees there is exactly one suffix within that distance in each direction.

The second method has a more serious problem: There may be no suffix points at all within an occurrence. A limit k on the maximum distance between consecutive suffix points makes the method work for patterns of length at least k.

The two methods can be combined to form a more efficient method. The key observation is that, when the start of an occurrence is far away from the preceding suffix point, it is close to the following suffix point and vice versa. Especially effective this idea becomes with the k-spaced suffix tree. If g is the distance from the start of an occurrence to the preceding suffix point and h is the distance to the following suffix point, then $g + h = k$. The combined algorithm is presented in Fig. 4.

Let W_g be the width of SST at depth g, i.e., the number of different strings of length g that can be read starting from the root. For a single value of g for which Method 1 is selected, the loop starting at line 3 is executed W_g times. The check at line 4 takes $O(m)$ time in the worst case. The innermost loop at lines 5 and 6 is executed once for each occurrence found using Method 1. Thus the total time taken by the loop over the whole algorithm is at most linear in the size of the output. For a given g, using Method 1 would therefore take $O(mW_g)$ in the worst case (excluding the size of the output for a moment).

Let C_h be the number of occurrences of P_h starting at suffix points. For a single $h = k - g$ for which Method 2 is selected, the loop at line 8 is executed C_h times. The check at line 7 takes $O(m)$ time and each check at line 9 takes $O(h)$ time in the worst case. For a given g, Method 2 would therefore take $O(m +$

```
input:  $k$-spaced suffix tree SST over text $T$, pattern $P$
output: starting points of all occurrences of $P$ in $T$
(1)  for $g = 0$ to $k - 1$ do
(2)       Determine whether to use Method 1 or Method 2 (see text).
          Method 1:
(3)            for all points $\overline{G}$ of SST, $|G| = g$ do
(4)                 if point $\overline{GP}$ exist in SST then
(5)                      for all leafs $\overline{T_i}$ under $\overline{GP}$ do
(6)                           output $i + g$
          Method 2:
(7)            if point $\overline{P_{k-g}}$ exist in SST then
(8)                 for all leafs $\overline{T_i}$ under $\overline{P_{k-g}}$ do
(9)                      if $T^i_{i-k+g} = P^{k-g}$ then
(10)                          output $i - k + g$
```

Fig. 4. The combined string matching algorithm for k-spaced suffix trees.

$hC_h) = O(m + (k - g)C_{k-g})$ in the worst case.

To determine at line 2 which method to use, we must try to estimate the time that each method would take. Assume that we have precomputed and stored the values W_g, $g = 1, \ldots, k - 1$. Then we can compute in constant time a good estimate for the time requirement of Method 1. The value C_h is the size of the subtree under $\overline{P_h}$. The C_h's can be precomputed and stored in the nodes of the tree. Then an estimate of the complexity of Method 2 can be computed in constant time after the execution of line 7. Based on these estimates, we select Method 1 if $mW_g < (k-g)C_{k-g}$ and Method 2 otherwise. This gives an algorithm with total time complexity

$$O\left(l + km + \sum_{g=0}^{k-1} \min\{mW_g, (k - g)C_{k-g}\}\right) \tag{1}$$

for a fixed problem instance. Here l is the size of the output and km is the total time spend at line 7 which is executed every time to select the method.

It can be shown that the worst case running time of the algorithm is $\Theta(kn)$, but we will omit the analysis in this paper. Instead, we will analyze the average case behavior of the algorithm using the uniform Bernoulli model of randomness. This should give a better idea of the practical behavior of the algorithm.

It is not difficult to see that in the average case $W_g = O(\min\{c^g, n/k\})$ and $C_h = O(n/(kc^{m-h}))$, where $c = |\Sigma|$ is the size of the alphabet. The term within the summation in (1) can now be bounded by

$$\begin{aligned} &\min\{mc^g, mn/k, (k - g)nc^{k-g-m}/k\} \\ &\leq \min\{mc^g, nc^{k-g-m}\} = \min\{T_1(g), T_2(g)\}. \end{aligned}$$

Table 1. Active ranges of the terms of the bound in Theorem 4.

term	meaning	active range
$\frac{n}{c^m}$	size of output	$k-1 \le \log_c \frac{n}{m} - m < \infty$
mc^{k-1}	Method 1 always faster	$k-2 \le \log_c \frac{n}{m} - m < k-1$
$\sqrt{mnc^{k-m}}$	both methods used	$-k \le \log_c \frac{n}{m} - m < k-2$
nc^{k-m}	Method 2 always faster	$-m \le \log_c \frac{n}{m} - m < -k$

The sum is then asymptotically bounded by its largest term, because T_1 increases and T_2 decreases exponentially in g. If $T_1(g) \le T_2(g)$ (or $T_2(g) \le T_1(g)$) for all $g \in \{0, \dots, k-1\}$, then the largest term is $T_1(k-1)$ ($T_2(0)$). Otherwise, the minimum is bounded by $T_1(\hat{g})$, where $\hat{g}$ is such that $T_1(\hat{g}) = T_2(\hat{g})$, i.e., $\hat{g} = (\log_c(n/m) + k - m)/2$. This gives the upper bound

$$\min\left\{mc^{k-1}, nc^{k-m}, \sqrt{mnc^{k-m}}\right\}$$

for the largest term. After an average case analysis of the other terms of (1) (omitted here) we get the following theorem.

Theorem 4. *Let T be a random text of length n in the uniform Bernoulli model of randomness and let P be any pattern of length m. The expected running time of the string matching algorithm in Fig. 4 over a k-spaced sparse suffix tree of T is*

$$O\left(\frac{n}{c^m} + k\min\left\{m, \log_c \frac{n}{k}\right\} + \min\left\{mc^{k-1}, nc^{k-m}, \sqrt{mnc^{k-m}}\right\}\right).$$

The bound in the theorem is quite complicated. Table 1 tries to shed a little light on it. The table tells, for each term in the bound, the range of parameters (active range) for which the term dominates the other terms. The middle term of the bound representing the selection time is not included in the table. There are combinations of parameters for which the selection term dominates, but the areas of domination are more complicated and higher dimensional than the clean linear ranges in the table. These areas fall within the two bottom ranges of the table. As it is, Table 1 represents the bound with the selection term removed. This is not a big concern, though, because the selection term has little dependence on the most important parameter, the length of the text. From the table we can also see that for large enough text the size of output dominates the running time.

6 Dual Suffix Trees

A *reverse suffix tree* is the suffix tree of the reverse of a text. Let SST be a sparse suffix tree. Its *dual reverse tree* SST^R is a sparse reverse suffix tree that contains the reverse T^{i^R} of prefix T^i if and only if SST contains the suffix T_i. Fig. 5 illustrates the dual reverse tree.

A simple method to utilize dual trees is to use them as alternatives to each other in the same way as Method 1 and Method 2 were alternatives in the algorithm of Fig. 4. However, this could be done with any two sparse suffix trees. What we want to do is to utilize the complementary properties of dual trees. We can use an SST to find the occurrences of suffix P_h starting at a suffix point as was done in Method 2. Then, instead of looking into the text, we use the dual reverse tree to check which of the occurrences of P_h are preceded by P^h.

The answer to a basic string matching in a (sparse) suffix tree is defined by some subtree of the suffix tree and can be represented by the root of this subtree. With a left-to-right ordering of the leaves, the answer can also be represented as a subinterval of this ordering. The end points of the intervals can be precomputed and stored to the nodes. Thus, we can get a root of subtree representation or an interval representation of the answer in time linear in the length of the pattern regardless of the size of the answer.

Suppose that each leaf $\overline{T_i}$ of SST contains the index of leaf $\overline{T^{i^R}}$ in the ordering of the leaves of SST^R. We can then, for each leaf $\overline{T_i}$ of SST under point $\overline{P_h}$, check whether $\overline{T^{i^R}}$ is in the subinterval representing the occurrences of P^h. This check takes constant time instead of $O(h)$ time that the corresponding check takes with Method 2.

An even better method can be developed by utilizing the root of subtree representation of search result. The method is based on what we call *dual links* between nodes of the two trees.

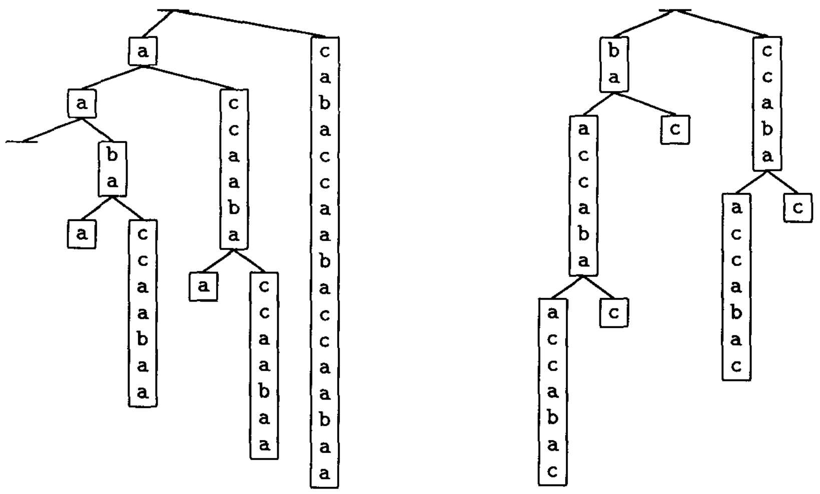

Fig. 5. The 3-spaced suffix tree SST for text cabaccaabaccaabaa and its dual reverse tree SST^R.

Definition 5. Let SST be a k-spaced sparse suffix tree and SST^R its dual reverse tree. Let $\overline{S}$ be a node of SST and let $t = k\lfloor |S|/k \rfloor$. Let $\overline{R}$ be the node of SST^R that contains point $\overline{S^{t^R}}$. A pointer from $\overline{S}$ to $\overline{R}$ is called a *dual link.*

Example 1. Consider the node $\overline{S} = \overline{\texttt{accaaba}}$ in SST in Fig. 5. Now $t = 6$, $S^6 =$ `accaab`, and the string $\texttt{accaab}^R$ = `baacca` is contained by the node $\overline{R} = \overline{\texttt{baaccaba}}$ in SST^R. Therefore, the dual link from $\overline{S}$ points to $\overline{R}$. □

The length of the string S^t is a multiple of k. This means that wherever S^t appears as a prefix of an SST-suffix it is also a suffix of an SST^R-prefix. Thus the subtree under $\overline{S^t}$ in SST and the subtree under $\overline{R}$ in SST^R represent the exactly same set of occurrences of S^t. Note that the node $\overline{S}$ may not contain the point $\overline{S^t}$. The dual link from such a node is not used in string matching but is needed for the construction.

The dual links can be set up by a breadth first traversal of SST after first constructing the two trees separately. The construction is based on the following lemma.

Lemma 6. *Let* SST *be a k-spaced sparse suffix tree and* SST^R *its dual reverse tree. Let $\overline{S}$ be a node of* SST, *$|S| \geq k$, and let $\overline{R}$ be the node pointed to by the dual link from $\overline{S}$. Then the dual link from node $\overline{S_k}$ points to an ancestor*[2] *$\overline{R^s}$ of $\overline{R}$ for some s.*

Proof. We first note that by Lemma 3 $\overline{S_k}$ is indeed a node. Let $t = k\lfloor |S|/k \rfloor$. Then, by Definition 5, $\overline{R}$ contains point $\overline{S^{t^R}}$ and the node $\overline{R^s}$ contains point $\overline{S_k^{t^R}}$. The string $S_k^{t^R}$ is a prefix of string S^{t^R} and thus $\overline{R^s}$ must be an ancestor of $\overline{R}$. □

The suffix link from node $\overline{S}$ points to node $\overline{S_k}$. Thus, starting from $\overline{S}$ we can find the node $\overline{R}$ by following first a suffix link, then a dual link and finally traveling down the tree SST^R. This takes at most $O(k)$ time. The dual link from $\overline{S_k}$ must be set before the dual link from $\overline{S}$, so the nodes must be processed in (at least roughly) breadth first order. The number of nodes is $O(n/k)$, thus the total time taken by the construction is $O(n)$.

A string matching algorithm using dual links is given in Fig. 6. We describe it using the terminology from Fig. 3. The value h is the length of the head. The algorithm first finds point $\overline{P_h}$ of SST representing the body of the occurrence and then extends it in all possible ways to reach a depth that is a multiple of k. Then, for each such extension, the algorithm finds the corresponding point in the reverse tree using a dual link and matches the head to complete the pattern.

Comparing this algorithm to Method 2, we notice that this algorithm does not search the whole tree under point $\overline{P_h}$. Instead, this algorithm goes through all points at depth g below $\overline{P_h}$. The number of those points is always at most, and can be significantly less than, the number of leaves under $\overline{P_h}$. Thus the algorithm is always at least as fast as Method 2.

[2] A node is an ancestor of itself.

```
input:  $k$-spaced suffix tree SST over text $T$, dual reverse tree SST$^R$
        pattern $P$ $(m = |P|)$
output: starting points of all occurrences of $P$ in $T$
(1)  Find and output occurrences starting from suffix points ($h = 0$).
(2)  for $h = 1$ to $k - 1$ do
(3)      if $h \geq m$ then use Method 1 over SST$^R$
(4)      else if point $\overline{P_h}$ exists in SST then
(5)          for all points $\overline{P_h G}$ of SST, $|G| = g = (-m + h) \bmod k$ do
(6)              Let $\overline{P_h G S}$ be the node containing point $\overline{P_h G}$.
(7)              Let $t = k\lfloor |S|/k \rfloor$.
(8)              Follow dual link from $\overline{P_h G S}$ to node $v$ of SST$^R$.
                     Node $v$ contains point $\overline{S^{t^R} G^R (P_h)^R}$.
(9)              if point $\overline{S^{t^R} G^R (P_h)^R P^{h^R}} = \overline{S^{t^R} G^R P^R}$ exist in SST$^R$ then
(10)                 for all leafs $T^{i^R}$ under $\overline{S^{t^R} G^R P^R}$ do
(11)                     output $i - m - g - t$
```

Fig. 6. String matching with dual suffix trees.

The algorithm can also be compared to Method 1 over the reverse tree SST^R. For both methods the wildcard string G is the string between the end of the occurrence and the following suffix point. Method 1 goes through all different G ending at a suffix point, while this algorithm goes through only those that are also preceded by P_h. Thus the algorithm is always at least as fast as Method 1 over the reverse tree.

We could combine Method 2 for SST and Method 1 for SST^R in the same way as the two methods for one tree were combined in the previous section. By the above comparisons the algorithm in Fig. 6 is always at least as fast as this combined algorithm. The difference may not be very large, however, because for small patterns Method 1 is close to the algorithm in Fig. 6 and for large patterns Method 2 is close to the algorithm.

References

1. A. Andersson, N. J. Larsson, and K. Swansson, *Suffix trees on words*, in Proc. 7th Symposium on Combinatorial Pattern Matching (CPM), 1996. To appear.
2. A. Andersson and S. Nilsson, *Improved behaviour of tries by adaptive branching*, Inf. Process. Lett., 46 (1993), pp. 295–300.
3. ———, *Efficient implementation of suffix trees*, Software—Practice and Experience, 25 (1995), pp. 129–141.
4. A. Apostolico, *The myriad virtues of subword trees*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, 1985, pp. 85–95.
5. A. Apostolico and W. Szpankowski, *Self-alignments in words and their applications*, Journal of Algorithms, 13 (1992), pp. 446–467.

6. G. H. GONNET, R. A. BAEZA-YATES, AND T. SNIDER, *Lexicographical indices for text: Inverted files vs.* PAT *trees*, Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.
7. R. W. IRVING, *Suffix binary search trees*, Technical report TR-1995-7, Computing Science Department, University of Glasgow, Apr. 1995.
8. J. KÄRKKÄINEN, *Suffix cactus: A cross between suffix tree and suffix array*, in Proc. 6th Symposium on Combinatorial Pattern Matching, CPM 95, 1995, pp. 191–204.
9. D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.
10. S. R. KOSARAJU AND A. L. DELCHER, *Large-scale assembly of DNA strings and space-efficient construction of suffix trees*, in Proc. 27th Annual ACM Symposium on Theory of Computing (STOC), 1995, pp. 169–177.
11. U. MANBER AND G. MYERS, *Suffix arrays: A new method for on-line string searches*, SIAM J. Comput., 22 (1993), pp. 935–948.
12. U. MANBER AND S. WU, *A two-level approach to information retrieval*, Technical Report TR 93-06, University of Arizona, 1993.
13. E. M. MCCREIGHT, *A space-economical suffix tree construction algorithm*, J. Assoc. Comput. Mach., 23 (1976), pp. 262–272.
14. D. R. MORRISON, *PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric*, J. Assoc. Comput. Mach., 15 (1968), pp. 514–534.
15. E. UKKONEN, *On-line construction of suffix-trees*, Algorithmica, 14 (1995), pp. 249–260.
16. P. WEINER, *Linear pattern matching algorithms*, in Proc. IEEE 14th Annual Symposium on Switching and Automata Theory, 1973, pp. 1–11.