5

A Partition-Based Suffix Tree Construction and Its Applications

Hongwei Huo[1] and Vojislav Stojkovic[2]
[1] *School of Computer Science and Technology, Xidian University, Xi'an*
[2] *Computer Science Department, Morgan State University, Baltimore*
[1]*China*
[2]*USA*

1. Introduction

A suffix tree (also called suffix trie, PAT tree or, position tree) is a powerful data structure that presents the suffixes of a given string in a way that allows a fast implementation of important string operations. The idea behind suffix trees is to assign to each symbol of a string an index corresponding to its position in the string. The first symbol in the string will have the index 1, the last symbol in the string will have the index n, where n = number of symbols in the string. These indexes instead of actual objects are used for the suffix tree construction. Suffix trees provide efficient access to all substrings of a string. They are used in string processing (such as string search, the longest repeated substring, the longest common substring, the longest palindrome, etc), text processing (such as editing, free-text search, etc), data compression, data clustering in search machines, etc.

Suffix trees are important and popular data structures for processing long DNA sequences. Suffix trees are often used for efficient solving a variety computational biology and/or bioinformatics problems (such as searching for patterns in DNA or protein sequences, exact and approximate sequence matching, repeat finding, anchor finding in genome alignment, etc).

A suffix tree displays the internal structure of a string in a deeper way. It can be constructed and represented in time and space proportional to the length of a sequence. A suffix tree requires affordable amount of memory. It can be fitted completely in the main memory of the present desktop computers. The linear construction time and space and the short search time are good features of suffix trees. They increase the importance of suffix trees. A suffix tree construction process is space demanding and may be a fatal in the case of a suffix tree to handle a huge number of long DNA sequences. Increasing the number of sequences to be handled, due to random access, causes degrades of the suffix tree construction process performance that uses suffix links. Thus, some approaches completely abandon the use of suffix link and give up the theoretically superior linear construction time for a quadratic time algorithm with better locality of reference.

2. Previous work

Weiner [1] gave the first linear time algorithm for suffix tree construction. McCreight [2] built a more space efficient algorithm for suffix tree construction in linear time. It has a

Source: Advances in Greedy Algorithms, Book edited by: Witold Bednorz,
ISBN 978-953-7619-27-5, pp. 586, November 2008, I-Tech, Vienna, Austria

readable account for suffix tree construction while processing a string from right to left. Ukkonon [3] developed a conceptually different linear-time algorithm for suffix tree construction that includes all advantages of McCreight's algorithm but also allows a much simpler explanation. It is a left-to-right on-line algorithm. Ukkonon's algorithm maintains at each step a suffix tree for a string S, where S is $c_1 \ldots c_i \ldots c_n$, as the index i is increasing from 1 to n. Many improvements in suffix tree construction have been done during the last decades. The early algorithms for suffix tree construction have been focused on developing algorithms in linear space. These algorithms are adapted to a small input size and the entire-complete suffix tree can be constructed in the memory. Unfortunately, these algorithms are less space efficient, because they suffer from a poor locality of memory reference. Cache processor architectures have a hard job to store memory references in the secondary memory. One moment there are too many data to be loaded into the memory that causes the missing a lot of cache and more disk swapping. Thus, how to develop a practical algorithm for suffix tree construction is still an important problem.

Suffix trees are not only used in the substring processing problems. They are used also in the complex genome-scale computational problems. For example, MUMmer [4, 5] is a system for the genome alignment, which uses as its main structure suffix trees to align two closely relative genomes. Due to the advantages of suffix trees, MUMmer provides the faster, simpler, and more systematic way to solve the hard genome alignment problem. REPuter [6, 7] is another popular software tool for the efficient computing of exact repeats and palindromes in the entire genome. It uses an efficient and compact suffix tree to locate exact repeats in linear time and space.

Although suffix trees have these superior features, they are not widely used in the real string processing software. The main reason for that is that the space consumption of a suffix tree is still quite large despite the asymptotically linear space [3]. Therefore, several researchers/scientists have developed the alternative index structures, which store less information than suffix trees, but they are more space efficient [8]. The most known index structures are suffix arrays, level compressed tries, suffix binary search trees, [4]. Index structures have to be tailed for some string matching problems and cannot be adapted to other kinds of problems without loss of performance. Also, the traditional string methods cannot be directly used in the DNA sequences because they are too complex to be treated. The reducing the space requirement of suffix trees is still an important problem in the genome processing.

In order to overcome these disadvantages, we propose a new algorithm for suffix tree construction for DNA sequences based on the partitioning strategies and use of the common prefixes to construct the independent subtrees [9]. The experiments show that the proposed algorithm is more memory-efficient and it has a better performance on the average running time.

3. Suffix tree

3.1 Definition

Definition 1. A suffix tree for a string S of n-characters, where n>=1, is a tree with n leaves numbered from 0 to n-1. Each internal node, other than the root, has at least two children. Each edge has an edge-label that is a nonempty substring of the string S. All edges exit from a same node have edge-labels beginning with different characters.

The most important characteristic of a suffix tree for a string S is that for each leaf i, where 0 <= i <= n-1, the concatenation of edge-labels on the path from the root to the leaf i is the ith suffix of the string S. The ith suffix of a string S is the suffix of the string S that starts at the position i.

Suffix trees can be constructed in linear time and space [1~3]. Some suffix tree construction algorithms that use suffix links require $O(n)$ construction time, where n is the length of a string. A suffix link is a link from one internal node to another internal node. Often, leaves of a suffix tree are labeled by leaf-labels. A leaf-label is the starting position of the suffix that ends at this leaf.

The Fig. 1 shows the suffix tree for the string S = *ATTAGTACA*\$. The \$ character represents the end of the string S and it is count as the part of the string S. Dashed lines represent suffix links.

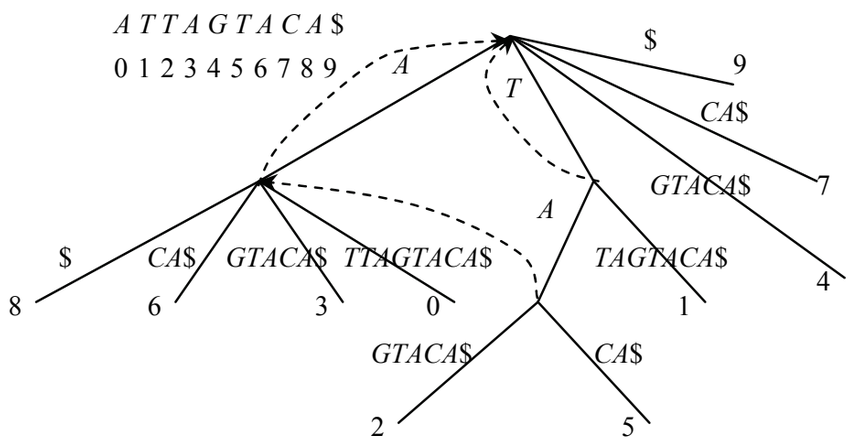


Fig. 1. The suffix tree for the string S = *ATTAGTACA*\$

3.2 Space requirements

The important characteristic of the suffix tree T for a string S, $T(S)$, is that $T(S)$ can be stored in $O(n)$ space, where n is the length of the string S.

The idea is the following:

- $T(S)$ has exactly n leaves, where n is the length of the string S.
- Since each internal node of $T(S)$ is a branching node, $T(S)$ has at most n internal nodes.
- Since in/at each node, except the root, enters/ends exactly one edge, $T(S)$ has at most $2n$ edges.
- Since each edge-label is a substring of S\$, it can be represented in constant space by a pair (start, end) points into S\$.

4. Partition-based suffix tree construction

4.1 Analysis

If a memory access mechanism has temporal and/or spatial locality features then the processor may use one or more caches to speed up access to the memory. Linear time, suffix tree construction algorithms, such as McCreight's algorithm [2] and Ukkonen's algorithm

[3], require many random accesses to the memory for suffix trees and links. In Ukkonen's algorithm, cache misses happen, when the algorithm makes a traversal via suffix links to reach another new subtree to check its children nodes. Such traversals cause random memory accesses at the very distant memory locations. In addition, each memory access visits memory with the higher probability because the address space span is too large to fit into the memory.

Kurtz's algorithm [8, 14], optimizes the space requirements for the McCreight's algorithm. Kurtz's algorithm divides the internal nodes into large nodes and small nodes to store the suffix tree information based on the relation of head position values. During the construction of internal nodes, there are many short or long small-large chains, which are sequences of small nodes followed by one large node. In a small-large chain, values of head position, the depth and suffix link of all small nodes can be derived from the large node at the end of chain. Therefore, with the bit optimization technique, Kurtz's algorithm uses four integers for one large node, two integers for one small node and one integer for each leaf node. Therefore, what a small-large chain is longer than more space is saved.

After analyzing, we find that a small-large chain is formed only if all nodes in the chain are series of new nodes created consecutively while series of suffixes are added into the suffix one by one.

DNA sequences are not only well known for their repetitive structures but also they are well known for their small-sized alphabet sequences that have high possibility of repetition. Therefore, applying Kurtz's algorithm on DNA sequences may not get advantage on small nodes but produces more large nodes.

4.2 Algorithm

Based on the properties of suffix trees, we can:

- in advance put together some suffixes of a branching node
- during the top-down suffix tree construction merge step by step the common prefixes of suffixes
- generate the internal branching nodes with the common prefix such as an edge-label and the responding leaf nodes
- finish the construction of the various branching nodes under the branch.

We propose the new ST-PTD (Suffix Tree Partition and Top-Down) algorithm for construction of a suffix tree for a DNA sequence. The ST-PTD algorithm uses partition and top-down techniques. Due to partition, a large input to the suffix tree construction is allowable. The construction of each subtree in the memory is independent.

The ST-PTD algorithm consists of two phases: partition suffixes and subtree construction. The algorithm is shown in Fig. 2.

Algorithm ST-PTD (S, *prefixlen*)
// Phase 1: Preprocessing
1. Scan the string S and partition suffixes based on the first *prefixlen* symbols of each suffix
// Phase 2. Construct suffix tree
2. **for** each partition P_i **do**
3. $R \leftarrow$ sorting(P_i)
4. **do**
5. **if** $|R| = 1$ **then**
6. create a leaf l

7. $ST_i \leftarrow ST_i \cup \{l\}$
8. **else**
9. *lcp* = finding-LCP(*R*)
10. create a branch node in the ST_i
11. add the *X* to *R*, *X* being the set of remaining suffixes from *R* after splitting off the longest common prefix
12. sorting(*R*)
13. **while** (!empty(*R*))
14. Merge{ST_i}

Fig. 2. The ST-PTD algorithm

In the preprocessing step, the suffixes of the input string *S* is partitioned into $|\sum|^{prefixlen}$ parts, where $\sum$ is an alphabet and $|\sum|$ is the size of the alphabet $\sum$. In the case of DNA sequences $\sum$ = {A, C, G, T} and $|\sum|$ = 4. *prefixlen* is the depth of partitioning. The partition procedure is as follows. First, we scan the input string from left to right. At each index position *i*, the *prefixlen* subsequent characters are used to determine one of the $|\sum|^{prefixlen}$ partitions and the index *i* is then recorded to the calculated partition. At the end of the scan, each partition will contain the suffix pointers for suffixes that all have the same prefix of size *prefixlen*. In the case of DNA sequences, we can assume that the internal nodes close to the root are highly repetitive and have the small alphabet - we can take value of *prefixlen* to be the $\log_4$ (*Seqlen*–1). However, when the value of *prefixlen* is large than 7, the running time for partition phase for large dataset, such as genome, is costly and can not bring the obvious advantages to the algorithm, thus we take the value of *prefixlen* to be the ($\log_4$(*Seqlen*–1))/2. In the suffix tree construction step, for each partition, the algorithm performs an independent construction of the respective suffix tree branch. The algorithm does not need to start at the root of the suffix tree but directly in the node that is found at some depth.

4.3 Space requirements

The space requirement measures how many bytes one character uses on average.

We use the DNA sequences from the NCBI web site to compare the space requirement of the Kurtz's algorithm [8] with the space requirement of the ST-PTD algorithm. The numbers given in the Table 1 refer to the space required for the construction. They do not include the *n* bytes used to store the input string.

Name	Length	Kurtz's algorithm	The ST-PTD algorithm	Saving
AC008583	122493	12.62	11.79	0.0658
AC135393	38480	12.39	11.85	0.0436
BC044746	4897	12.61	11.72	0.0706
J03071	11427	12.32	13.68	-0.1104
M13438	2657	12.50	11.59	0.0728
M26434	56737	12.52	12.03	0.0391
M64239	94647	12.62	11.72	0.0713
V00662	16569	12.69	11.74	0.0749
X14112	152261	12.58	11.87	0.0564
ecoli	4668239	12.56	11.72	0.0669
[Average]	516841	12.541	11.971	0.0451

Table 1. The space requirements of Kurtz's algorithm and the ST-PTD algorithm

Table 1 shows the space requirement for each sequence.
- The first column of the Table 1 contains the names of DNA sequences.
- The second column of the Table 1 contains the lengths of DNA sequences.
- The third column of the Table 1 contains the space requirement of Kurtz' algorithm.
- The fourth column of the Table 1 contains the space requirement of the ST-PTD algorithm.
- The fifth column of the Table 1 contains the savings.

The ST-PTD algorithm compared with Kurtz's algorithm saves about 4.55% in space.
There is no relationship between space needs and the length of sequence. However, the DNA sequence, such as J03071, has a great effect on the space demand.

4.4 Running time

Kurtz's algorithm and the ST-PTD algorithm have been implemented in the C programming language and compiled with the GCC compiler. To learn and show the impact of the memory on the algorithms, we ran/executed the programs on two different platforms config1 and config2. Config1 consisted of the Intel Pentium 4.3 GHZ processor, 512M RAM, and the Red Hat Linux 9 operating system. Config2 consisted of the Intel Pentium III 1.3 GHHZ processor, 128 RAM, and the Fedora 4 operating system.
The experimental results are shown in Table 2. The running time is in seconds and throughout is the ratio of time multiplied by 10^6 and sequence length. The dark shaded areas show the better throughout. '-' shows the running time more than 1 hour.
We used in both algorithms arrays as the main data structures to get the higher efficiency in time. Unfortunately, arrays limit the size of data they deal with. However, we still used arrays, because Kurtz's algorithm in which we used linked lists to implement DNA sequences takes 1176.02 seconds (about 20 minutes) for the sequence B_anthracis_Mslice of 317k length and over four hours for the sequence ecoil of 4.6M length.
Although Kurtz's algorithm requires O(n) time in the worst case and the ST-PTD algorithm requires O(n^2) time, the ST-PTD algorithm is a little faster than Kurtz's algorithm on the average running time. This shows that the locality of memory reference has the great influence on the running time of both algorithms. The partition strategies and the sequence structure also had the impact on the performance of both algorithms. For example, the difference induced by the unbalanced partitions on the sequence influenza slice is obvious.
The ST-PTD algorithm has greater advantages on Kurtz's algorithm for the lower configuration due to its partition phase. The partition phase decreases the size of the set of problems we are processing so that we can deal with the larger size of data.
Comparing the running time of both algorithms in different configurations, we can see that memory is still one of the bottlenecks affecting the performances of both algorithms. Suffix trees are indeed very space greedy. In addition, compared with Kurtz's algorithm, the ST-PTD algorithm is easier to understand and implement. Also, the ST-PTD algorithm is easier to be parallelized because the construction of each subtree is independent.

5. Some applications of suffix trees

5.1 Exact string matching

The *exact string matching problem* is: Given a string/sequence S and a pattern string P. Find all positions of the pattern P in *the string S*.

		Config 1				Config 2			
		Kurtz's algorithm		The ST-PTD algorithm		Kurtz's algorithm		The ST-PTD algorithm	
Sequence	Length	time	tput	time	tput	time	tput	time	tput
J03071	11427	0.06	5.25	0.10	8.75	0.06	5.25	0.12	10.50
V00662	16569	0.01	0.60	0.02	1.21	0.01	0.60	0.02	1.21
AC135393	38480	0.2	5.20	0.94	24.43	0.26	6.76	1.54	40.02
M26434	56737	0.04	0.71	0.05	0.88	0.06	1.06	0.07	1.23
M64239	94647	0.07	0.74	0.08	0.85	0.11	1.16	0.12	1.27
AC008583	122493	0.09	0.73	0.11	0.90	0.14	1.14	0.15	1.22
X14112	152261	0.11	0.72	0.14	0.92	0.20	1.31	0.21	1.38
B_anthracis_Mslice	317829	0.34	1.07	0.31	0.98	0.46	1.45	0.45	1.42
H.sapiens chr.10 slice1	1119913	1.28	1.14	1.27	1.13	1.59	1.42	2.70	2.41
H.sapiens chr.10 slice2	2099930	2.62	1.25	2.53	1.20	3.41	1.62	5.32	2.53
H.sapiens chr.10 slice3	3149930	3.98	1.26	3.98	1.26	23.31	7.40	8.45	2.68
H.sapiens chr.10 slice4	4199930	5.56	1.32	5.13	1.22	-	-	11.74	2.80
ecoli	4668239	7.19	1.54	5.79	1.24	-	-	13.69	2.93
H.sapiens chr.10 slice5	4899930	6.25	1.28	6.08	1.24	-	-	14.08	2.87
H.sapiens chr.10 slice6	5250000	6.62	1.26	7.74	1.47	-	-	15.39	2.93
H.sapiens chr.10 slice7	5600000	7.03	1.26	7.04	1.26	-	-	16.61	2.97
influenza slice	5918744	5.16	0.87	46.07	7.78	-	-	71.15	12.02
H.sapiens chr.10 slice8	6019975	7.66	1.27	21.94	3.64	-	-	38.44	6.39
H.sapiens chr.10 slice9	6300000	8.2	1.30	7.92	1.26	-	-	18.78	2.98
H.sapiens chr.10 slice10	6999930	9.67	1.38	9.04	1.29	-	-	21.30	3.04
H.sapiens chr.10 slice11	8400000	10.71	1.28	11.52	1.37	-	-	26.55	3.16
H.sapiens chr.10 slice12	9100000	12.92	1.42	13.53	1.49	-	-	28.65	3.15
Arabidopsis thaliana chr. 4	9835812	44.01	4.47	30.33	3.08	-	-	-	-
H. sapiens chr. 10 slice13	10500000	79.13	7.54	25.89	2.47	-	-	-	-
[Average]		8.42	2.13	7.98	2.02				

Table 2. The running time and throughout of Kurtz's algorithm and ST-PTD

The exact string matching problem can be solved using the suffix tree on the following elegant way:

- Construct the suffix tree for the string S, $T(S)$.
- Traverse – top-down pass through $T(S)$ from the root further into $T(S)$, guided by the characters of P, as long as there is a continuation in $T(S)$ that corresponds to the letters of P.
- If this search stops before the end of P is reached, P does not occur in S.
- If P can be spelled out completely, then P occurs in S. Moreover, the numbers at the leaves below the end point of this search tell all the positions in S where P occurs.

Suppose that $S = ATTAGTACA\$$ is a string. The suffix tree for the string S, $T(S)$, is shown in Fig. 1.

Suppose that $P = TAA$ is a pattern. After reading the first two characters of P, T and A, we will arrive to the branching node $\underline{TA}$. Because, the edge A is not outgoing from the branch node $\underline{TA}$, we cannot continue with the matching P against $T(S)$. In other words, P does not occur in $T(S)$. Therefore, P is not the substring of S.

Suppose that $P = ATA$ is a pattern. Follow the first edge from the root to the node $\underline{A}$. The node $\underline{A}$ has the edge $TTAGTACA\$$ leading to the leaf $\underline{0}$. The next character to be read in P is the last character in P - $\underline{A}$. $\underline{A}$ does not match the next character $\underline{T}$ of the edge $TTAGTACA\$$. Therefore, P does not occur in $T(S)$ that is P is not the substring of S.

If we can find that P occurs in $T(S)$, then we can also find the positions in S where P occurs. Suppose that $P = TA$ is a pattern and assume $T(S)$ of Fig. 1. Following the second edge from the root, we will reach to the branching node $\underline{TA}$. Therefore, P is the substring of S. The leaf numbers in the subtree below the branching node $\underline{TA}$ are 2 and 5. Therefore, TA starts in S at positions 2 and 5.

The time complexity of this algorithm is as follows. The construction of $T(S)$ takes $O(n)$ time, where n is the length of S. The search for occurrences of P takes $O(m + z)$ time, where m is the length of P and z is the number of occurrences of P in S. (Note that z can be larger than m, but not larger than n.) Hence, the asymptotic time for the complete search is the same as for the optimal on-line string matching algorithms such as Boyer-Moore or Knuth-Morris-Pratt, $O(n + m)$.

5.2 Exact set matching

The *exact set matching problem* is: Given an array of strings/sequences $(S_k) = S_1, S_2, \ldots, S_k$ and a pattern string P. Find all positions of the pattern P in the sequence (S_k).

The exact set matching problem can be solved using the suffix trees on the following straightforward way:

- Concatenate the strings $S_1, S_2, \ldots, S_k$ separated by the unique separator symbols $\$_i$, where i =1, ..., k-1, into the string S, $S = S_1\$_1S_2\$_2 \ldots \$_{k-1}S_k$.
  The string S is called the generalized string of the strings $S_1, S_2, \ldots, S_k$.
- Construct the suffix tree for the generalized string S, $T(S)$.

The suffix tree for the generalized string S, $T(S)$, is called the generalized suffix tree for the generalized string S. Leaves of the generalized suffix tree are labeled with pairs of the form (sequence number, start position). Often, the labels of leaf-edges are cut of after the first separator symbol. The pattern search is performed as in the standard suffix tree. Again, the pattern search takes $O(m + z)$ time to find all z occurrences of a pattern P of the length m.

Suppose that $S = BABAB\$_1AAB\$_2$ is a generalized string. The corresponding generalized suffix tree is shown in Fig. 2.

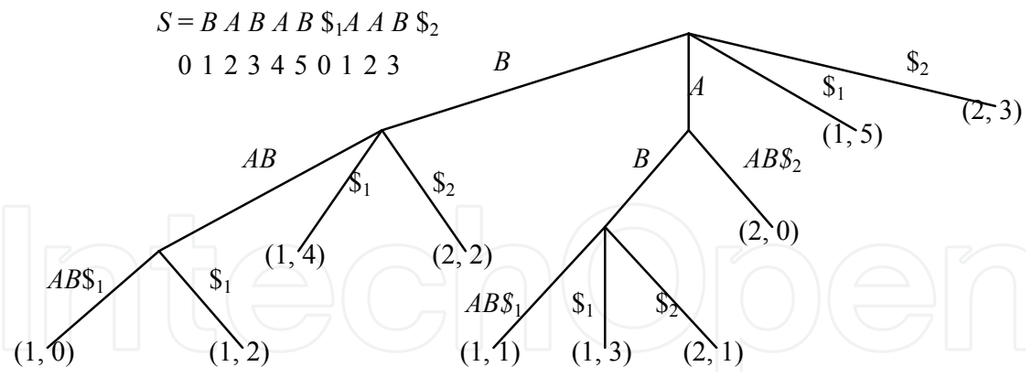


Fig. 2. The generalized suffix tree for the string S_1=*BABAB*\$1 and the string S_2=*AAB*\$2.

The advantage of constructing an index becomes clear when several searches are performed on the same string, as the following table shows (z denotes the output size).

-	- search for one pattern of length m	- search for k patterns of length m
- on-line algorithms	- $O(n + m)$	- $O(k(n + m))$
- suffix-tree algorithm	- $O(n + m)$	- $O(n + km + z)$

5.3 Minimal unique substrings

The *minimal unique substrings problem* is: Given a string S and a constant (a positive integer) l. Find - enumerate all substrings u of the string S that satisfy the following properties:

- u occurs exactly once in S (uniqueness)
- all proper prefixes of u occur at least twice in S (minimality)
- the length of u is greater or equal than l.

Suppose that S = *ATTAGTACA*\$ is a string and l = 2 is a constant. The minimal unique substrings of S are *TAG* and *AT* (see Fig. 1). The substring *ATT,* for example, is not the minimal unique substring of S, since the proper prefix *AT* of *ATT* is already unique, that is the minimality condition is not satisfied.

To solve the minimal unique substrings problem, exploit the following two properties of the suffix tree for the string S:

- if a string w occurs at least twice in the string S, there are at least two suffixes in S for which w is a proper prefix. Therefore, in the suffix tree $T(S)$, w corresponds to a path ending with an edge to a branching node;
- if a string w occurs only once in the string S, there is only one suffix in S for which w is a prefix. Therefore, in the suffix tree $T(S)$, w corresponds to a path ending with an edge to a leaf.

According to the second property, we can find the unique strings by looking at the paths ending on edges to a leaf. So, if we have reached a branching node, say w, then we only have to enumerate the leaf edges outgoing from w. Suppose w->y is the edge from the branching

node w leading to the leaf y, labeled av where a is the first character on that edge. Then wa occurs only once in S, i.e. it is unique. Moreover, w corresponds to the path leading to w and by the second property, w occurs at least twice. Finally, we only have to check if the length of wa is larger or equal than l.

The suffix tree based algorithm to solve the minimal unique substrings problem is very simple.

Let us apply the algorithm to the suffix tree of Fig. 1. Assume that $l = 2$. We can skip the root, since it would result in strings, which are too short. Let us consider the branching node reached by the edge from the root labeled TA. Then $w = TA$ and with the first character G of the label of the second edge we obtain the minimal unique substring TAG. The other solution AT can be found by looking at the other branching node reached by the label A from the root together with its zero-numbered edge.

The running time of the minimal unique substrings algorithm is linear in the number of nodes and edges in the suffix tree, since we have to visit each of these only once and for each we do a constant amount of work. The algorithm runs in linear time since the suffix tree can be constructed in linear time and there are $O(n)$ nodes and edges in the suffix tree. This is optimal, since the running time is linear in the size of its input.

The minimal unique substrings problem has applications in primer design.

5.4 Maximal unique match

The standard dynamic programming algorithm to compute the optimal alignment of two sequences of the length m and the length n requires $O(mn)$ steps. This is too slow for the cases when sequences have hundreds of thousands or millions characters.

There are algorithms that allow aligning of two genomes under the assumption that genomes are similar. Genomes are similar if they have long identical subsequences. Identical subsequences, called MUMs (Maximal Unique Matches), are almost certainly part of high quality and efficient alignment of two genomes. The first step of the maximal unique match algorithm is to find MUMs. MUMs are taken as the fixed part of alignment. The remaining parts of genomes (the parts not included in MUMs) are aligned with traditional dynamic programming methods.

In this section, we will show how to compute MUMs in linear time. This is very important for the applicability of the maximal unique match algorithm, the MUM algorithm. We do not consider how to compute the final alignment.

The *maximal unique match problem, the MUM problem,* is: Given two sequences S, $S' \in \Sigma^*$ (the genomes) and a constant (a positive integer) l. Find all subsequences u with the following properties:

- $|u| \geq l$.
- u occurs exactly once in S and exactly once in S' (uniqueness).
- for any character a neither ua nor au occurs both in S and in S' (maximality).

Suppose that $S = CCTTCGT$ *is a string*, $S' = CTGTCGT$ *is another string*, and $l = 2$ is a constant. There are two maximal unique matches CT and $TCGT$. Consider an optimal alignment of these two sequences (assuming the same costs for insertions, deletions, and replacements):

CCT-TCGT
-CTGTCGT

Clearly, two MUMs *CT* and *TCGT* are parts of this alignment.
To compute MUMs, we first have to construct the suffix tree for the concatenation of the two sequences S and S'. To prevent any match that occurs on the borderline between S and S', we put the unique symbol # between S and S', i.e. we construct the suffix tree for the string X, where $X = S\#S'$. A MUM, say u, must occur exactly twice in X, once in S and once in S'. Therefore, u corresponds to the path in the suffix tree $T(X)$ ending with an edge to a branching node. Since u is the right-maximal by definition (i.e. for any symbol a, ua does not occur both in S and in S'), u must even correspond to the branching node. In other words, for each MUM u there is the branching node $\underline{u}$ in the suffix tree for the string X.
Since u occurs twice in X, there are exactly two leaves in the subtree below u. The subtree can contain no branching node, hence there are two leaf edges outgoing from u. One edge must lead to the leaf, say v, that corresponds to a suffix starting in S and the other edge must lead to the leaf, say w, that corresponds to a suffix starting in S'. For the given branching node u, the existence of exactly two such leaf edges can be checked easily. What remains is to verify left-maximality, i.e. to check if there is the character a such that au occurs both in S and in S'. Suppose that the leaf v has the leaf number i and the leaf w has the leaf number j. Then u is left maximal, if and only if $i = 0$ or $X_{i-1} \neq X_{j-1}$. In other words, we only have to look at the two positions immediately to the left of two positions in X where u starts.

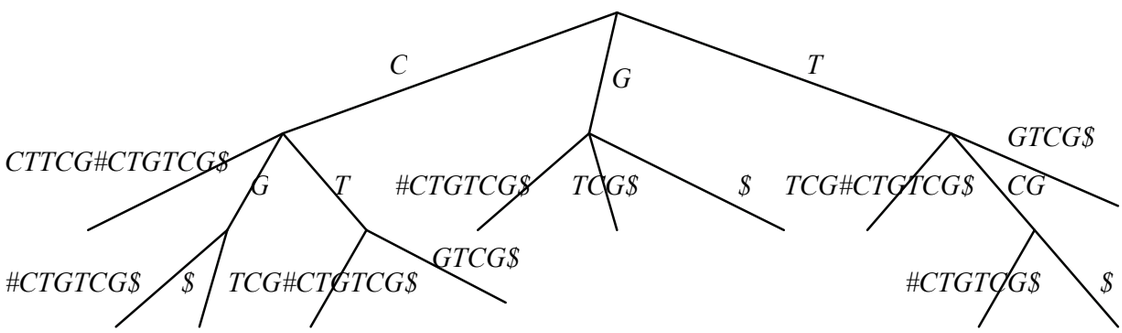


Fig. 3. The suffix tree for *CCTTCG#CTGTCG$* without the leaf edges from the root

Suppose that $S = CCTTCG$ is a string, $S' = CTGTCG$ is another string, and $l = 2$ is a constant. Consider the suffix tree for the string $S\#S'$. This is shown in Fig. 3. Obviously, the string *TCG* occurs once in S and S', since there are two corresponding edges from the branching node $\underline{TCG}$. Comparing the character G and the character T immediately to the left of the occurrences of *TCG* in S and S' verifies the left maximality. The string *CT* also occurs once in S and once in S', as verified by the two leaf edges from $\underline{CT}$. The left-maximality is obvious, since the character C and the character # to the left of the occurrences are different.

5.5 Maximal repeat

If a sequence S may be represented by the array $S_0 \ldots S_{n-1}$, then the sequence S is indexed from 0 to n-1. If a sequence S is indexed, then a subsequence $S_i \ldots S_j$ of the string S may be represented by the pair (i, j). A pair (l, r), where $l = (i, j)$ and $r = (i', j')$, is a repeat, if $i < i'$ and $S_i \ldots S_j = S_{i'} \ldots S_{j'}$. l is the left instance of the repeat and r is the right instance of the repeat.

Note that the left instance of the repeat and the right instance of the repeat may overlap.
Suppose that $S = GAGCTCGAGC$ is a string. The string S contains the following repeats of the length $l \geq 2$:

-	((0, 3), (6, 9))	-	*GAGC*
-	((0, 2), (6, 8))	-	*GAG*
-	((0, 1), (6, 7))	-	*GA*
-	((1, 3), (7, 9))	-	*AGC*
-	((2, 3), (8, 9))	-	*GC*

The example shows that shorter repeats are often contained in longer repeats. To remove redundancy, we restrict to maximal repeats. A repeat is the maximal if it is the left maximal and the right maximal. These notions are formally defined as follows: The repeat $((i, j), (i', j'))$ is the *left maximal* if and only if $i\text{-}1 < 0$ or $S_{i-1} \neq S_{i'-1}$. The repeat $((i, j), (i', j'))$ is the *right maximal* if and only if $j'+1 > n\text{-}1$ or $S_{j+1} \neq S_{j'+1}$.
From now, we will restrict ourselves to the maximal repeats. All repeats, which are not the maximal repeats, can be obtained from the maximal repeats. In the example above, the last four repeats can be extended to the left or to the right. Hence, only the first repeat is maximal.
In the following, we will present an algorithm to compute all maximal repeats of a given sequence. It works in two phases. In the first phase, the leaves of the suffix tree are annotated. In the second phase, the repeats are output while simultaneously the branching nodes are annotated.
We will show how the algorithm to compute all maximal repeats works for the string $\$_1GCGC\$_2GGCG\$_3$. The corresponding suffix tree (with some unimportant edges left out) is shown in Fig. 4.
Suppose that:

- a string S of the length n over the alphabet $\sum$ such that the first and the last character of S both occur exactly once and
- the suffix tree for a string S is given (as in Fig. 4).

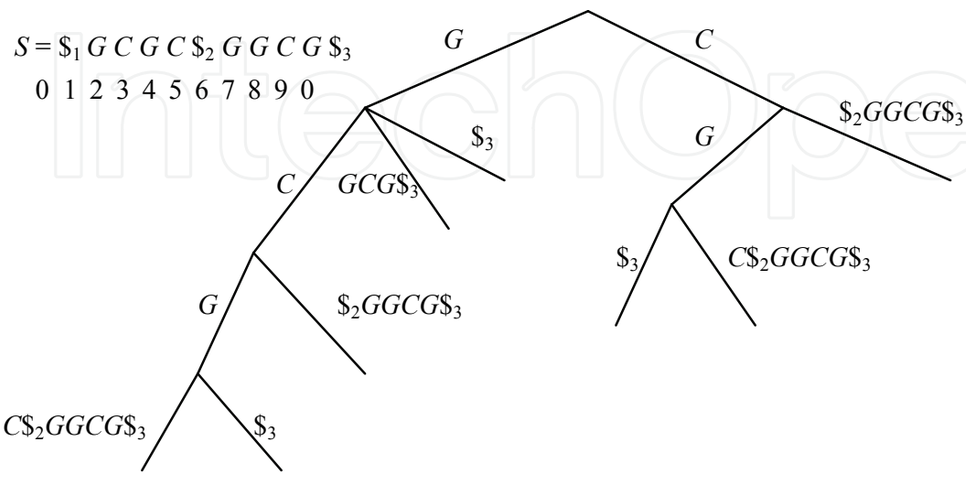


Fig. 4. The suffix tree for the string $\$_1GCGC\$_2GGCG\$_3$.

We can ignore leaf edges from the root, since the root corresponds to the repeats of the length zero, and we are not interested in these.
In the first phase, the algorithm annotates each leaf of the suffix tree: if $v = S_i \ldots S_n$, then the leaf $\underline{v}$ is annotated by the pair (a, i), where i is the starting position of the suffix v and $a = S_{i-1}$ is the character to the immediate left to the position i. (a, i) is the *leaf annotation* of $\underline{v}$ and it is denoted $A(\underline{v}, S_{i-1}) = \{i\}$. We assume that $A(\underline{v}, c) = \phi$ (the empty set) for all characters $c \in \Sigma$ different from S_{i-1}. This assumption holds in general (also for branching nodes), whenever there is no annotation (c, j) for some j.
The leaf annotation of the suffix tree for the string S in Fig. 4, is shown in Fig. 5.
Leaf edges from the root are not shown. These edges are not important for the algorithm.

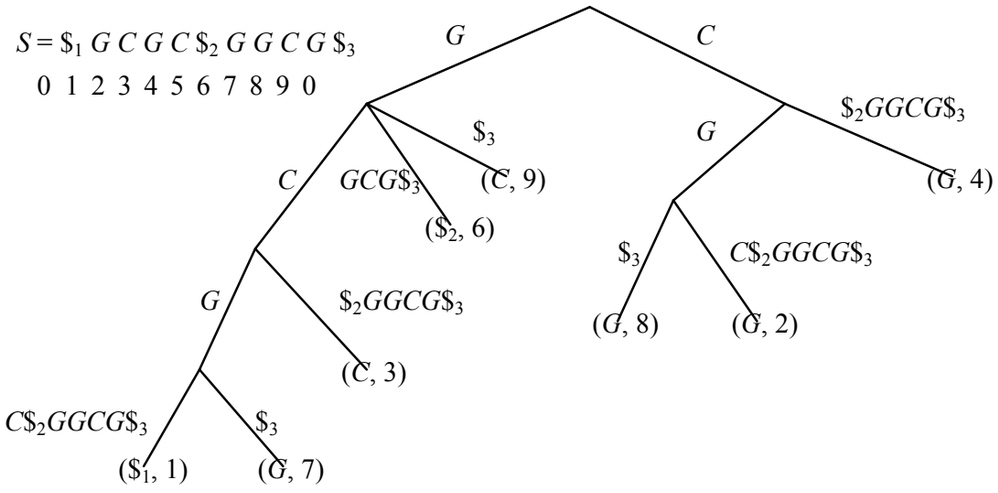


Fig. 5. The suffix tree for the string $\$_1GCGC\$_2GGCG\$_3$ with leaf annotation.

The leaf annotation gives the character upon which we decide the left-maximality of a repeat, plus the position where the repeated string occurs. We have only to combine this information at the branching nodes appropriately. This is done in the second phase of the algorithm: In a bottom-up traversal, the repeats are output and simultaneously the annotation for the branching nodes is computed. A bottom-up traversal means that a branching node is visited only after all nodes in the subtree below that node have been visited. Each edge, say $\underline{w} \to \underline{v}$ with a label au, is processed as follows: At first repeats (for w) are output by combining the annotation already computed for the node $\underline{w}$ with the complete annotation stored for $\underline{v}$ (this was already computed due to the bottom-up strategy). In particular, we output all pairs $((i, i + q\text{-}1), (j, j + q\text{-}1))$, where

- q is the depth of node $\underline{w}$, i.e. $q = |w|$,
- $i \in A(\underline{w}, c)$ and $j \in A(\underline{v}, c')$ for some characters $c \neq c'$,
- $A(\underline{w}, c)$ is the annotation already computed for $\underline{w}$ w.r.t. character c and $A(\underline{v}, c')$ is the annotation stored for node $\underline{v}$ w.r.t. character c'.

The second condition means that only those positions are combined which have different characters to the left. It guarantees the left-maximality of repeats. Recall that we consider

processing the edge $\underline{w} \rightarrow \underline{v}$ with the label au. The annotation already computed for $\underline{w}$ was inherited along edges outgoing from $\underline{w}$, which are different from $\underline{w} \rightarrow \underline{v}$ with the label au. The first character of the label of an edge, say c, is different from a. Now since w is the repeated substring, c and a are characters to the right of w. Consequently only those positions are combined which have different characters to the right. In other words, the algorithm also guarantees the right maximality of repeats.

As soon as for the current edge the repeats are output, the algorithm computes the union $A(\underline{w}, c) \cup A(\underline{v}, c)$ of all characters c, i.e. the annotation is inherited from the node $\underline{v}$ to the node $\underline{w}$. In this way, after processing all edges outgoing from $\underline{w}$, this node is annotated by the set of positions where w occurs, and this set is divided into (possibly empty) disjoint subsets $A(\underline{w}, c_1),\ldots, A(\underline{w}, c_r)$, where $\sum = \{c_1,\ldots,c_r\}$.

Fig. 6 shows the annotation for a large part of the previous suffix tree and some repeats. The bottom up traversal of the suffix tree for the string $\$_1GCGC\$_2GGCG\$_3$ begins with the node $\underline{GCG}$ of depth 4, before it visits the node $\underline{GC}$ of depth 2. The maximal repeats for the string $\underline{GC}$ are computed as follows: The algorithm starts by processing the first edge outgoing from $\underline{GC}$. Since initially, there is no annotation for $\underline{GC}$, no repeat is output, and $\underline{GC}$ is annotated by $(C, 3)$. Then the second edge is processed. This means that the annotation $(\$_1, 1)$ and $(G, 7)$ for $\underline{GCG}$ is combined with the annotation $(C, 3)$. This give repeats ((1, 2), (3, 4)) and ((3, 4), (7, 8)). The final annotation for $\underline{GC}$ becomes $(C, 3)$, $(G, 7)$, $(\$_1, 1)$ which also can be read as $A(\underline{GC}, C) = \{3\}$, $A(\underline{GC}, G) = \{7\}$, and $A(\underline{GC}, \$_1) = \{1\}$.

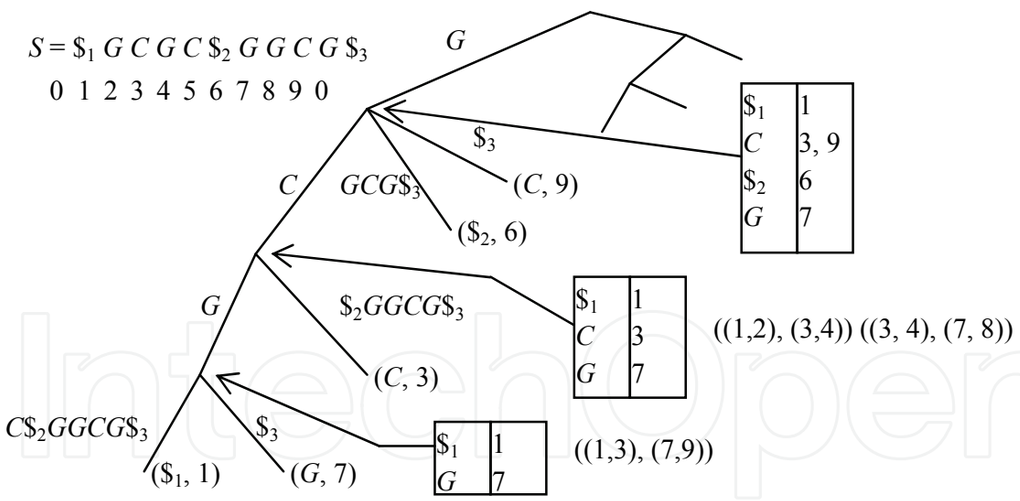


Fig. 6. The annotation for a large part of the suffix tree of Fig. 5 and some repeats.

Let us now consider the running time of the maximal repeats algorithm. Traversing the suffix tree bottom-up can be done in time linear in the number of nodes. Follow the paths in the suffix tree, each node is visited only once. Two operations are performed during the traversal: the output of repeats and the combination of annotations. If the annotation for each node is stored in linked lists, then the output operation can be implemented such it runs in time linear in the number of repeats. Combining annotations only involves linking

lists together. This can be done in time linear in the number of nodes visited during the traversal. Recall, that the suffix tree can be constructed in $O(n)$ time. Therefore, the algorithm requires $O(n + z)$ time, where n is the length of the input string and z is the number of repeats.

To analyze the space consumption of the maximal repeats algorithm, the annotations for all nodes do not have to be stored all at once. As soon as a node and its father have been processed, the annotations are no longer needed. The consequence is - the annotation only requires $O(n)$ space. Therefore, the space consumption of the algorithm is $O(n)$.

The maximal repeats algorithm is optimal, since its space and time requirement are linear in the size of the input plus the output.

6. References

[1] P. Weiner, "Linear Pattern Matching Algorithms," Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, pp1-11, 1973

[2] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *Journal of Algorithms*, Vol. 23, No. 2, pp262-272, 1976

[3] E. Ukkonen, "On-line Construction of Suffix Trees," Algorithmica, Vol. 14, No. 3, pp249-260, 1995

[4] Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White and Steven L. Salzberg, "Alignment of whole genomes," *Nucleic Acids Research*, Vol. 27, pp. 2369–2376, 1999

[5] Aurthur L. Delcher, Adam Phillippy, Jane Carlton and Steven L. Salzberg, "Fast algorithms for large-scale genome alignment and comparison," *Nucleic Acids Research*, Vol. 30, pp. 2478–2483, 2002

[6] S. Kurtz and Chris Schleiermacher, "REPuter fast computation of maximal repeats in complete genomes," *Bioinformatics*, Vol. 15, No. 5, pp.426-427, 1999

[7] Stefan Kurtz, Jomuna V. Choudhuri, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye and Robert Giegerich, "REPuter the manifold applications of repeat analysis on a genomic," *Nucleic Acids Research*, Vol. 29, No.22, pp. 4633–4642, 2002

[8] Stefan Kurtz, "Reducing the space requirement of suffix trees," *Software Pract. Experience*, Vol. 29, pp. 1149-1171, 1999

[9] Hongwei Huo and Vojislav Stojkovic, "A Suffix Tree Construction Algorithm for DNA Sequences," IEEE 7th International Symposium on BioInformatics & BioEngineering. Harvard School of Medicine, Boston, MA, October 14-17, Vol. II, pp. 1178-1182, 2007.

[10] Stefan Kurtz, "Foundations of sequence analysis," lecture notes for a course in the winter semester, 2001

[11] D. E. Knuth, J. H. Morris, and V. B. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, 1977, Vol. 6, pp. 323-350, 1997

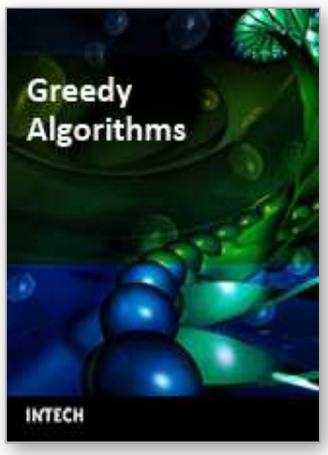
[12] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, 1977,Vol. 20, pp. 762-772, 1997

[13] Yun-Ching Chen & Suh-Yin Lee, "Parsimony-spaced suffix trees for DNA sequences," *ISMSE'03*, Nov, pp.250-256, 2003.

[14] Giegerich, R., Kurtz, S., Stoye, J.,"Efficient implementation of lazy suffix trees," *Soft. Pract. Exp*. Vol. 33,1035–1049, 2003

[15] Schurmann, K.-B., Stoye, J.,"Suffix-tree construction and storage with limited main memory," Technical Report 2003-06, 2003, University of Bielefeld, Germany.

[16] Dan Gusfield, "Algorithms on strings, trees, and sequences," Cambridge University Press, 1997



Greedy Algorithms

Edited by Witold Bednorz

ISBN 978-953-7619-27-5
Hard cover, 586 pages
Publisher InTech
Published online 01, November, 2008
Published in print edition November, 2008

Each chapter comprises a separate study on some optimization problem giving both an introductory look into the theory the problem comes from and some new developments invented by author(s). Usually some elementary knowledge is assumed, yet all the required facts are quoted mostly in examples, remarks or theorems.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Hongwei Huo and Vojislav Stojkovic (2008). A Partition-Based Suffix Tree Construction and Its Applications, Greedy Algorithms, Witold Bednorz (Ed.), ISBN: 978-953-7619-27-5, InTech, Available from: http://www.intechopen.com/books/greedy_algorithms/a_partition-based_suffix_tree_construction_and_its_applications

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen