Theoretical Computer Science 542 (2014) 98–107

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Quick greedy computation for minimum common string partition

Isaac Goldstein, Moshe Lewenstein *,1

Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel

ARTICLE INFO

Article history:
Received 21 November 2013
Received in revised form 1 May 2014
Accepted 10 May 2014
Communicated by M. Crochemore

Keywords:
Strings
Approximation algorithm
Pattern matching

ABSTRACT

In the *minimum common string partition* problem one is given two strings S and T with the same character statistics and one seeks for the smallest partition of S into substrings so that T can also be partitioned into the same substring multiset. The problem is fundamental in several variants of edit distance with block operations, e.g. signed reversal distance with duplicates and edit distance with moves.
The minimum common string partition problem is known to be NP-complete and the best approximation algorithm known has an approximation factor of $O(\log n \log^* n)$. Since the minimum common string partition problem is of utmost practical importance one seeks a heuristic that will (1) usually have a low approximation factor and (2) will run fast.
A simple greedy algorithm is known, which iteratively choose non-overlapping longest common substrings of the input strings. This algorithm has been well-studied from an approximation point of view and it has been shown to have a bad worst case approximation factor. However, all the bad approximation factors presented so far stem from complicated recursive construction. In practice the greedy algorithm seems to have small approximation factors. However, the best current implementation of greedy runs in quadratic time.
We propose a novel method to implement greedy in linear time.

1. Introduction

The classical edit distance is the number of edit operations, insertions, deletions, character exchanges and (sometimes) swaps, needed to transform one string into another. All the edit operations work on single characters, besides swap which operates on two. Motivation for block operations has stimulated a large collection of new problems, some of which have a completely different flavor than the original edit distance problem.

Sankoff and Kruskal [21] and Tichy [23] were the first to consider block operations together with simple character operations. Lopresti and Tomkins [19] suggested several distance measures for block operations in the same spirit as they were defined for the case of single characters. The subject of block edit distance gained an increasing interest in the last decade especially due to its important impact on the computational biology field. In the research of the homology of structures there is an interest in finding organisms' characteristics that are derived from a common ancestor. The task of identifying the orthologs, especially direct descendants of ancestral genes in current species is a fundamental problem in computational biology. A promising approach to solving this problem is to take into account not only local mutations but

* Corresponding author.
E-mail addresses: goldshi@cs.biu.ac.il (I. Goldstein), moshe@cs.biu.ac.il (M. Lewenstein).

1 This research was supported by BSF grant 2010437 and GIF grant 1147/2011.

http://dx.doi.org/10.1016/j.tcs.2014.05.006
0304-3975/© 2014 Published by Elsevier B.V.

also global genome rearrangements events. This approach is strongly connected to the area of edit distance where block operations are allowed, since those genes can be represented as sequences of characters with block operations applied to them [4].

1.1. Sorting by reversals and MCSP

Chen et al. [4] studied the problem of signed reversal distance with duplicates (SRDD) which is a slight extension of the well-known sorting by reversals (SBR) problem. They showed that this problem is NP-hard even for $k = 2$, where k is the maximum number of occurrences of any character in the input strings (it is worth noting that in the unsigned case the problem is NP-hard even if $k = 1$. However, for the signed case there is a polynomial algorithm for solving SBR for $k = 1$, see [14]). Moreover, they pointed out that this problem is closely related to the minimum common string partition problem (MCSP). In the MCSP we are given two strings S and T and we need to find a partition of each of these strings to substrings, so that the set of substrings of S and T are the same. We define this more fully in Section 2.

1.2. Edit distance with moves and MCSP

The problem of edit distance with moves was studied by Cormode and Muthukrishnan [8]. In this variant of the classical edit distance, moving a substring is allowed in addition to deleting and inserting single characters. They showed the problem to be NP-complete and suggested an approximation algorithm with an approximation factor of $O(\log n \log^* n)$ and running time $O(n \log^* n)$. This is still the best approximation algorithm given for the problem. Lately there has been a renewed interest in approximating NP-complete problems for substring, subsequence and supersequence problems [1,7,12,13].

Shapira and Storer [22] observed that the problem, while recursive, can be transformed into a non-recursive version with a constant-factor cost in the approximation. Moreover, they showed that the problem can be transformed into a version in which moves only are allowed. It has been observed that this is in essence a reduction to the MCSP problem, although not specifically noted so in the paper.

Using Shapira and Storer's observations one can use Cormode and Muthukrishnan [8] approximation algorithm for MCSP as well. This is currently the best known approximation.

1.3. Restricted versions of MCSP

Goldstein et al. [11] proved that even the simple case of 2-MCSP (MCSP in which each character occurs at most twice) is NP-hard. Moreover, they showed that it is APX-hard. They also gave an 1.1037-approximation algorithm for this problem which improved the 1.5 ratio showed by Chen et al. [4]. For 3-MCSP a 4-approximation algorithm was given [11]. Kolman [17] proposed an $O(k^2)$-approximation algorithm for k-MCSP with $O(nk)$ running time for general k, where each symbol can occur at most k times. Kolman and Walen [18] improved this result and gave a $O(k)$-approximation algorithm with $O(n)$ running time.

Due to the close relation between the MCSP problem and the SBR problem (as shown by Chen et al. [4]) the approximation factor for these restricted versions of MCSP also applies to parallel restricted versions of SBR with just a constant factor penalty. Instead of restricting the problem by the number of each symbol occurrences, the MCSP problem and the related problem of SBR could be also restricted by the alphabet size. Unsigned SBR for unary alphabet could be solved trivially. However, Christie and Irving [5] showed that for binary alphabet unsigned SBR becomes NP-hard. Similar NP-hardness results for MCSP incorporating alphabet sizes greater than 1 were given by Jiang et al. [15].

1.4. MCSP parameterized

Recently there have been several attempts to parameterize the problem. Damaschke [9] showed that the problem of MCSP is fixed-parameter tractable (FPT) on both k, the size of the optimal solution to the MCSP problem of the input strings, and r, the repetition number of a string x, which is defined as the maximum i so that $x = uv^iw$ holds for some strings u, v, w, with nonempty v. Jiang et al. [15] presents an FPT algorithm for the d-MCSP problem (MCSP where each symbol appears at most d times) that runs in $O((d!)k)$ time. They also investigated the case in which the partition is x-balanced, which requires the length of each block not to be more than x away of the average length. They devised a FPT algorithm with parameters k and x which runs in $O((2x)kk!n)$ time for this case. Following their work, Bulteau et al. [2] gave an FTP algorithm that run in $O(d^{2k}kn)$ time. Their algorithm also applies to a generalized version of the MCSP problem. Finally, Bulteau and Komusiewicz [3] showed a FTP algorithm parametrized only by k.

1.5. MCSP and the GREEDY algorithm

A simple greedy algorithm, to be denoted by GREEDY, was suggested by Shapira and Storer [22]. They showed that for large classes of inputs the algorithms have a logarithmic approximation factor.

Chrobak et al. [6] demonstrated that in the general case the approximation ratio of GREEDY for the MCSP is not better than $\Omega(n^{0.43})$. In addition, for the special case of 4-MCSP it was found that it is at least $\Omega(\log n)$. Nevertheless, they showed it has an $O(n^{0.69})$ approximation factor for the general case, and a 3-approximation factor for 2-MCSP. The lower bound on GREEDY's performance for the general case was later increased to $\Omega(n^{0.46})$ by Kaplan and Shafrir [16].

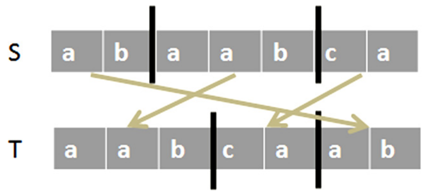


Fig. 1. An example of the minimum common string partition problem for input strings: $S = abaabca$, $T = aabcaab$ and the optimal solution for this instance.

Algorithm 1 GREEDY(S, T).

```
1: Set $P$ to be the initial (empty) partition of $S$ and $Q$ to be the initial (empty) partition of $T$.
2: Initially all letters in $S$ and $T$ are unmarked and $P$, $Q$ are empty.
3: while there are unmarked letters in $T$ do
4:     $LCS \leftarrow$ longest common substring of $S$ and $T$ that does not contain marked letters.
5:     $LCS_S, LCS_T \leftarrow$ occurrences of $LCS$ in $S$ and $T$, respectively. All letters in these occurrences must be unmarked.
6:     Designate $LCS_S$ as a block of $P$ in $S$ and $LCS_T$ as a block of $Q$ in $T$.
7:     Mark all letters in $LCS_S$ and $LCS_T$.
8: end while
9: output $(P, Q)$
```

1.6. Our results

Since the edit distance with moves and the signed reversal distance with duplicates are used in practice it is of interest to construct an algorithm for MCSP that also (1) gives good approximation factors and (2) runs fast.

The approximation algorithm of [8] is the best in the worst-case sense, but the approximation factor is inherent in the method. On the other hand, GREEDY, while it can be bad in the approximation factor sense, as mentioned above, in practice, GREEDY seems to approximate quite well. The worst-case examples are constructed from recursive definitions.

Shapira and Storer presented an implementation of GREEDY that runs in $O(n^2)$ time. We will show a novel algorithm for implementing the GREEDY that runs in $O(n)$ time.

2. Problem definitions and preliminaries

Given a string S, $|S|$ is the length of S. Throughout this paper we denote $n = |S|$. An integer i is a *location* or a *position* in S if $i = 1, \dots, |S|$. The substring $S[i..j]$ of S, for any two positions $i \le j$, is the substring of S that begins at index i and ends at index j. The *suffix* S_i of S is the substring $S[i..n]$.

The *suffix tree* [10,20,24,25] of a string S, denoted ST(S), is a compact tree of all the suffixes of $S\$$, i.e. S concatenated with a delimiter symbol $\$ \notin \Sigma$. ST($S$) requires $O(n)$ space. Algorithms for the construction of a suffix tree enable $O(n)$ preprocessing time when $|\Sigma|$ is constant (where Σ is the alphabet set), and $O(n \log \min(n, |\Sigma|))$ time when $|\Sigma|$ is not. A more sophisticated construction of the suffix tree can be constructed in linear time even for alphabets drawn from a polynomially-sized range, see [10]. For a node v in a suffix tree we say that it *represents* a string s, if s is given by the concatenation of the edge labels along the path from the root to node v. The length of s is called the *string-depth* of v.

Let the longest common substring, for short *LCS*, of S and T be denoted by $LCS(S, T)$. Moreover, if $LCS(S, T)$ appears at location i of S and location j of T then we say that it *appears* at (i, j).

We say that two strings S and T over alphabet Σ are *equi-statistic* if for every $\sigma \in \Sigma$ the number of appearances of σ in S and T is the same. Clearly, this implies that $|S| = |T|$.

A *partition* of a string $S = s_1, \dots, s_n$ is a sequence of strings (also called blocks) $R_1, R_2, \dots, R_l$ such that $S = R_1 \cdot R_2 \cdot \dots \cdot R_l$, where $\cdot$ denotes concatenation. A *common string partition* of two strings S and T is a partition $R_1, \dots, R_l$ of S such that there is a permutation $\pi : \{1, \dots, l\} \to \{1, \dots, l\}$ such that $R_{\pi(1)}, \dots, R_{\pi(l)}$ is a partition of T. Note that if S and T are equi-statistic there is a simple common string partition of S and T; that is $R_i = s_i$ for every $1 \le i \le n$. Moreover, for any two strings that have a common string partition it is easy to verify that they are equi-statistic. Nevertheless, for equi-statistic strings S and T there may be various different common string partitions.

The *minimum common string partition problem* is defined as follows (Fig. 1).

Input: Two equi-statistic strings S and T.
Output: A common string partition $R_1, \dots, R_l$ such that l is minimized.

The GREEDY algorithm for the minimum common string partition problem is presented in Algorithm 1. We assume that the input strings S and T are equi-statistic.

Informally speaking, when given two input strings S and T, the algorithm iteratively finds the longest common substring of both strings, each time marking its letters (in both strings) so they will not be chosen again. Those iteratively chosen longest common substrings make the complete common partition of S and T.

3. Naive GREEDY implementation

When implementing the GREEDY method it is natural to use suffix trees. Specifically, start by constructing a generalized suffix tree for S and T, or in other words a suffix tree for $S\#T\$$, where $\#, \$ \notin \Sigma$, to be denoted by $\mathrm{ST}(S, T)$. To find the *LCS* of S and T seek for the deepest node v which contains leaves in its subtree which represent suffixes from both S and T. This can be done by a traversal of the suffix tree in order. The substring which node v represents is the *LCS*.

Once the *LCS* is found one can reiterate the process, but beforehand x needs to be removed. One way to handle this is to replace x (in both S and T) with a new symbol not in the alphabet and to start from scratch. The time this process takes is $O(n^2)$ because in each round the suffix tree needs to be built from scratch and there can be $O(n)$ rounds. In essence this is the implementation of the GREEDY method in [22].

4. Toward fast GREEDY implementation

Since we desire to reduce the time to have a more efficient implementation, we do not want to construct such a suffix tree from scratch. Rather we want to fix the given suffix tree to reflect the changes that have been made. This is challenging. We now give the outline of how we shall do this.

4.1. The first round

Let $x = LCS(S, T)$ and assume that it appears at (i, j). More specifically, $S = S[1 \ldots i-1] \cdot x \cdot S[i + |x|, n]$ and $T = T[1 \ldots j-1] \cdot x \cdot T[j + |x|, n]$ or, in suffix terms, x is the prefix of suffix S_i and x is also the prefix of T_j. Once x is found it is necessary to update the suffix tree to reflect the fact that the appearances of x cannot be used again, not x itself and not any part of it. This means that we need to create a generalized suffix tree for $S[1 \ldots i-1]$, $S[i + |x|, n]$, $T[1 \ldots j-1]$ and $T[j + |x|, n]$. However, the problem is that when transforming the generalized suffix tree for S and T into the generalized suffix tree for $S[1 \ldots i-1]$, $S[i + |x|, n]$, $T[1 \ldots j-1]$ and $T[j + |x|, n]$, all the suffixes $S_1, \ldots, S_{i-1}$ and $T_1, \ldots, T_{j-1}$ are affected. This means that the transformation can be too costly, namely $O(n)$ which is the same as reconstructing the suffix tree from scratch.

Nevertheless, it turns out that the number of suffixes that affect the future runs of GREEDY can be bounded by the following observation.

Observation 1. *Let $x = LCS(S, T)$. Denote $k = |x|$. Assume that x appears at (i, j). Let y be any other common substring of S and T. Say, y appears at (i', j'). If $i' \notin [i-k+1, i+k-1]$ then the appearances of y and x do not intersect in S. Likewise, if $j' \notin [j-k+1, j+k-1]$ then the appearances of y and x do not intersect in T.*

It follows from the lemma that if the *LCS* x appears at (i, j) and is of length $k = |x|$ then the first $i - k$ suffixes of S and the first $j - k$ suffixes of T, while they technically should be shortened because of the removal of x, they need not be actually shortened because they will not affect the GREEDY method. Therefore, when preparing the generalized suffix tree for the next step it is sufficient to take care of the $2k - 1$ suffixes from S in the range $[i - k + 1, i + k - 1]$ and the $2k - 1$ suffixes from T in the range $[j - k + 1, j + k - 1]$. The suffixes of S in the range $[i, i + k - 1]$ need to be removed completely, as they overlap the removed x and the suffixes of S in the range $[i - k + 1, i - 1]$ need to be shortened. More specifically, each S_r, where $r \in [i - k + 1, i - 1]$ needs to be shortened to length $i - r$. Likewise, the suffixes of T in the range $[j, j + k - 1]$ need to be removed completely and the suffixes of T in the range $[j - k + 1, j - 1]$ need to be shortened appropriately.

4.2. The subsequent rounds

The described method for transforming the generalized suffix tree upon finding the *LCS* of S and T holds for the next rounds as well. Specifically, in the next round we will find an *LCS* y in the generalized suffix tree for $S[1 \ldots i-1]$, $S[i + |x|, n]$, $T[1 \ldots j-1]$ and $T[j + |x|, n]$, which will, once again, be the deepest node v with suffixes from S and T in its subtree. The string y represented by v will be the *LCS* (in the 2nd round) and will appear at (i', j') if the leaf representing $S_{i'}$ is in the subtree of v and the leaf representing $T_{j'}$ is also in the subtree of v. Removing y will split $S[1 \ldots i-1]$, $S[i + |x|, n]$, $T[1 \ldots j-1]$ and $T[j + |x|, n]$ into (at most) six substrings, (at most) three from S and (at most) three from T. We say "at most" because y splits a fragment into two only if it is in the middle of a fragment. Otherwise, if y is at the end, or beginning the fragment will not be split after removing y. If y is the whole fragment then there is one fragment less. The suffix tree will need to be updated again. Obviously, Observation 1 holds also here. This process is repeated iteratively and in round r, S and T will each be fragmented into (at most) $r + 1$ substrings and we will need to find the *LCS* that appears both in the fragments of S and the fragments of T. We denote the *LCS* in round r with $LCS^{[r]}$, which is also called the *current LCS* when dealing with the rth round. We point out that $\sum_r |LCS^{[r]}| = n$. Hence, if we can solve each round r in time $O(|LCS^{[r]}|)$ then we will have overall time $O(n)$.

Therefore, we define the four distinct sets of suffixes described above to be handled in round r of the GREEDY procedure (Fig. 2):

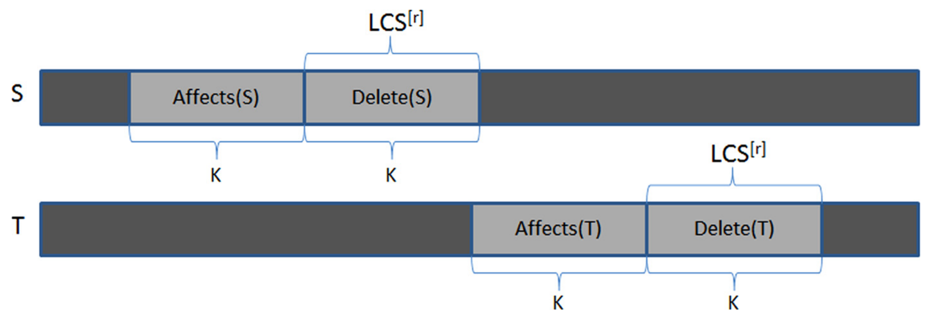


Fig. 2. The 4 sets of *Affects(S), Affects(T), Delete(S), Delete(T)* for $LCS^{[r]}$. $|LCS^{[r]}| = k$.

1. ***Delete(T)*:** all T_i such that i is one of the $|LCS^{[r]}|$ locations where $LCS^{[r]}$ appears in T.
2. ***Affects(T)*:** all T_i such that i is one of the $|LCS^{[r]}| - 1$ locations preceding the appearance of $LCS^{[r]}$ in T.
3. ***Delete(S)*:** all S_i such that i is one of the $|LCS^{[r]}|$ locations where $LCS^{[r]}$ appears in S.
4. ***Affects(S)*:** all S_i such that i is one of the $|LCS^{[r]}| - 1$ locations preceding the appearance of $LCS^{[r]}$ in S.

Now we are left with two major tasks. First, we need to *update* the suffix tree to reflect the changes described for the four above mentioned groups of suffixes. Second, in each round we will need to *find* $LCS^{[r]}$. The challenging part for the first problem is to fix the suffix tree for *Affects(T)* and *Affects(S)* since it requires putting those suffixes in new places within the suffix tree. The challenging part of the second problem is to find $LCS^{[r]}$ in the limited time we have, namely in $O(|LCS^{[r]}|)$ time. This will require two data structures, for otherwise we will need to restart from scratch every time. The data structures need to be adapted according to the changes in the suffix tree.

5. Quick suffix tree updating

Say we are just finished with round r and moving to round $r+1$. Currently S and T are each fragmented into at most $r+1$ substrings. We need to update the suffix tree to correspond to $LCS^{[r]}$. This fragments the text beyond the previous round by, at most, one more fragment for each of S and T. Let the suffix tree before round r be denoted by $\mathrm{ST}(S,T)^{[r]}$. Recall, that we will be updating only the suffixes from the four previously described suffix sets.

For any suffix in any of these sets we will need to remove it from the suffix tree. We will later need to reposition those from *Affects(S)* and *Affects(T)*. Removing a suffix is actually quite straightforward, albeit a bit technical. We reach the appropriate leaf v representing the suffix by a pointer from an array representing the string (S or T). v's parent is then checked whether it has two children or more. If it has more it is sufficient to remove v and the edge connecting it to its parent. If v's parent has only two sons, one of which is v, in order to maintain the suffix tree in its compressed format one needs to phase out the parent of v by putting an edge between the grandparent and sibling of v. The labeling of the new edge needs to be fixed, which is easily done from the current labels.

So, what really needs to be handled is entering the truncated suffixes of *Affects(S)* and *Affects(T)* into the suffix tree. We treat *Affects(T)* and *Affects(S)* in the same manner but separately. The update we will do is done concurrently for all the members of *Affects(T)*.

We will refer to the suffix tree construction of Weiner [25]. Weiner constructs the suffix tree from the shortest suffix to the longest. Say the process is done on string Q of length q. Then the suffixes are entered into the compressed trie of suffixes (which will become a suffix tree) in the order of suffixes \$, $Q[q]\$$, $Q[q-1\dots q]\$$, $\dots$, $Q[2\dots q]\$$, $Q\$$. The suffixes are inserted with the help of suffix pointers, which point from node v representing x to node v' which represent ax, if such a node exists. Weiner shows that the whole process is linear using an amortized argument that goes along the following lines. The amortized process assumes that we have just entered suffix x and need to enter suffix ax. If x has a pointer to ax then we use it and we enter the suffix ax as its child. If not, then we walk up the path of root-to-leaf-representing-x until we reach a node v representing x' a prefix of x for which there is a pointer to ax' and do as mentioned before. The argument is that the overall walk is linear in the number of suffixes = the length of $Q = q$.

This suits our purpose well. Since we have just found $LCS^{[r]}$ we are allowed to spend $k = |LCS^{[r]}|$ time in work for reinserting all the suffixes in *Affects(T)*. If $x = LCS^{[r]}$ then $y = y_1, .., y_{k-1}$ the substring (of length $k-1$) that appears just before x in T is what *Affects(T)* is constructed from. Therefore, the suffixes of *Affects(T)*, after being truncated, are $y_{k-1}, y_{k-2}y_{k-1}, y_{k-3}y_{k-2}y_{k-1}, \dots, y$. We traverse the suffix tree and enter these suffixes in the described order using the Weiner scheme. The time arguments of Weiner shows that all these suffixes can be inserted in $O(|LCS^{[r]}|)$ time. The same work is done for *Affects(S)*. Hence, we conclude:

Lemma 1. *Updating the suffix tree* $\mathrm{ST}(S,T)^{[r]}$ *in round r can be done in* $O(|LCS^{[r]}|)$ *time.*

What still needs to be done is to actually find the $LCS^{[r]}$. We will do so in the next section.

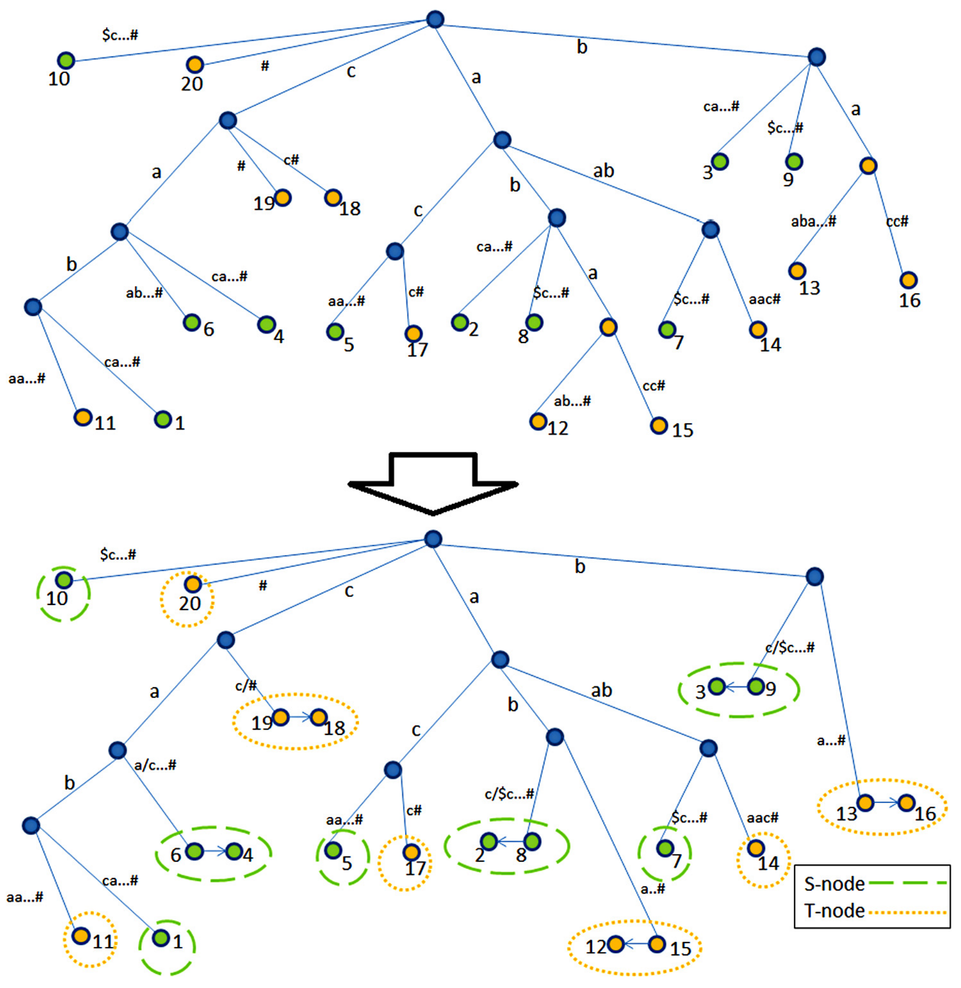


Fig. 3. Suffix tree for strings $S = cabcacaab$ and $T = cabaabacc$ and the corresponding suffix skeleton tree (at the bottom).

6. Fast LCS finding

The process of quickly finding the current *LCS* is based on two data structures which we will define: the *suffix skeleton tree* and the *LPB array* (Longest Prefix Bucket Array). The idea behind these two data structures is to maintain the minimum data needed to fulfill our mission.

6.1. Suffix skeleton tree

The suffix skeleton tree is a compact version of a suffix tree. Our goal is to keep only those nodes in the suffix tree that represent the points of contact of substrings from both S and T and remove all other nodes. For a more formal description we give some definitions. Given $\mathrm{ST}(S, T)$ the suffix tree of $S\#T\$$ ($\#, \$ \notin \Sigma$), we define an *essential node* to be a node that has in its subtree leaves representing substrings from both S and T. Each essential node may have essential and non-essential nodes as its children. We wish to contract our suffix tree so it will contain only essential nodes and all other nodes will be removed. That being said, we do not want to lose the information on the suffixes represented by the leaves of our suffix tree. Therefore, we define two special child nodes for each essential node v: An *S-node* which is a node that contains all the leaves representing a suffix from S that v is their lowest essential ancestor (if there is no such leaf then v has no S-node child). A *T-node* which is defined in the same manner as an S-node but for T's suffixes. Each of these two special nodes contains a linked-list of leaf nodes representing substrings that share the same longest common substring with the other input string. An example is given in Fig. 3.

Using these definitions, the *suffix skeleton tree* of S and T is the suffix tree contracted to essential nodes, S-nodes and T-nodes only. Note that all the nodes that are not essential are not part of the suffix skeleton tree themselves, but the information about the suffixes in their subtree is kept in the S-nodes and T-nodes of the suffix skeleton tree.

The construction of the suffix skeleton tree starts with a regular suffix tree. Afterwards, we traverse the tree in post-order and mark each node with one of three colors:

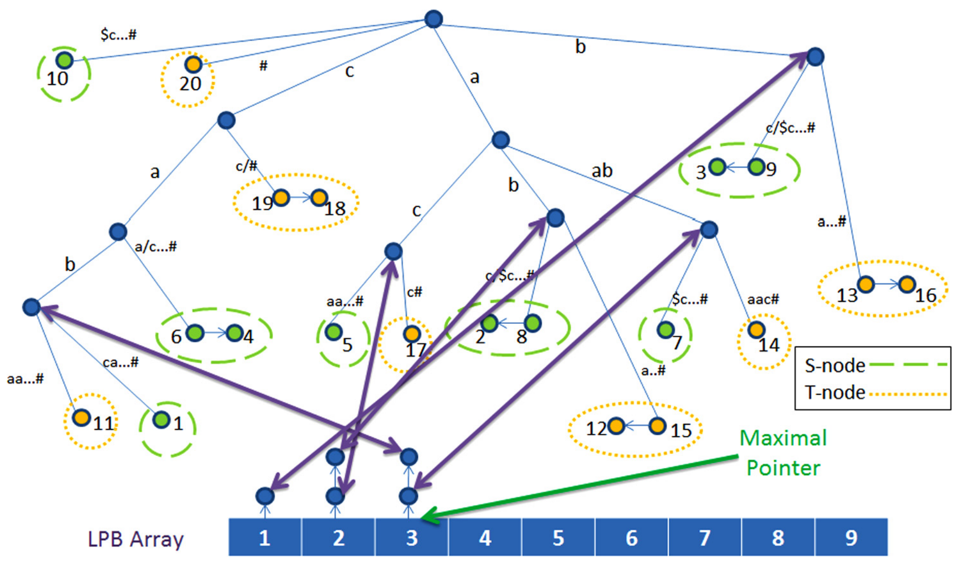


Fig. 4. LPB array for the suffix skeleton tree from Fig. 3.

Algorithm 2 FindingCurrentLCS(S, T, *suffix skeleton tree*, *LPB array*).

```
1: Use the maximal-pointer to find the largest index $i$ in our LPB array containing a non-empty bucket.
2: First pointer of the linked-list in $LPB[i]$ leads to essential node $v$.
3: By definition of our LPB array, $v$ has both an S-node and a T-node.
4: Choose one suffix from each special node.
5: The string on root-to-$v$ path is the current $LCS$, which is also the prefix of both chosen suffixes.
```

- *Green* – if the leaves in its subtree represent suffixes from S only.
- *Yellow* – if the leaves in its subtree represent suffixes from T only.
- *Blue* – if the leaves in its subtree represent suffixes from both input strings.

The blue nodes are our essential nodes. So, we traverse the tree again removing all the green and yellow nodes combining them in a linked-list kept in two special nodes – the S-node and the T-node – of their corresponding lowest *essential* ancestor.

That being said, we conclude that by a linear time suffix tree construction and two tree traversals we can have our first tool for finding the *LCS* quickly – the suffix skeleton tree.

6.2. *LPB array*

The second ingredient of our quick *LCS* finding is the LPB array. This is an array of n buckets, where n is the length of each of our two input strings. We denote the ith bucket in the array by $LPB[i]$. $LPB[i]$ contains a linked-list of pointers to essential nodes with string-depth i, that have both an S-node and a T-node as their children. It should be noted that some of the buckets might be empty. Moreover, we maintain a special pointer which we call the *maximal pointer* that points to the largest index having a non-empty bucket in our array.

Building such an array can be done by first allocating the n buckets and then traversing the suffix skeleton tree and adding the pointers to each of the essential nodes that have both an S-node and a T-node as children. These pointers can be added to the beginning of the list of their proper bucket according to the string-depth of their essential node which we maintain during our traversal. Therefore, the LPB array can be constructed by a linear-time procedure.

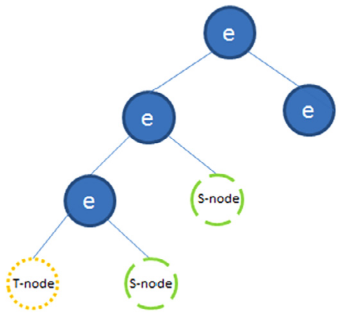
This LPB array gives us a way to quickly pick essential nodes having both an S-node and a T-node in a sorted manner. An example of this data structure is given in Fig. 4.

6.3. *Finding the LCS*

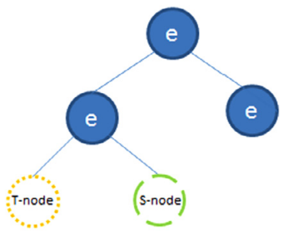
Armed with these two data structures – the suffix skeleton tree and the LBP array – we can now show how to quickly find the current *LCS* of our input strings in the rth round.

The formal description of this procedure is given in Algorithm 2.

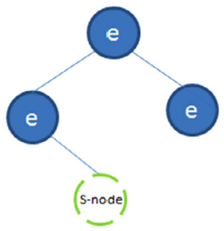
The last thing we need to show regarding this process is how to update the two data structures we define. The need for updating is due to the changes done to our strings following the *LCS* finding, which include the deletion of some suffixes



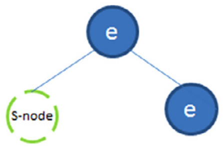
(a) *Suffix is deleted from a T-node. The T-node becomes empty*



(b) *First parent compression.*



(c) *A merge of S-nodes*



(d) *Second parent compression. The process ends.*

Fig. 5. An example of a deletion process with parent compressions.

and the shortening of some others (those suffixes are the members of the 4 groups we define in Section 4). These updates are the subject of the next subsection.

6.4. Updating auxiliary data structures

6.4.1. Suffix skeleton tree updates

We first describe how to update the suffix skeleton tree.

To do that quickly we maintain a pointer from each of our suffixes (in an array representing them) to the node representing it which is contained in the linked-list of some special node (S-node or T-node). These pointers can be added during the traversals we do for constructing the suffix skeleton tree and need not change the fact that this process is done in linear-time.

6.4.1.1. Deletion When *deleting* a suffix from our input strings, we can use the pointers we created to quickly get the node representing the suffix in the list maintained by the S-node or T-node. Then, we could delete that node from our tree. Yet, the deletion process might not be finished. It could be the case that our special node becomes empty as a result of our node deletion. We then need to remove the special node, which might cause its parent to be no longer essential. In this situation we merge the single special node it has with the corresponding special node that is its sibling (it might not exist, in this case we use the special node that remains as the new sibling by itself). Then, we can delete our node, which might eventually, recursively, cause its parent to not be essential. We keep following the same procedure until we reach an essential node that remains essential. We call the process of merging the special nodes and deleting the essential node that stops being essential *parent compression* (Fig. 5). The cost of our deletion operation is the number of parent compressions we do, as each parent compression can be done in constant time by removing a node and merging two linked list of special nodes. At first glance, it seems as if this process can be quite expensive, but we should observe that each parent compression involves a **node deletion**. Therefore, as a consequence of the fact that there are no more than $O(n)$ nodes in our suffix skeleton tree and we do not add more than $O(n)$ nodes during the shortening process in all rounds together, we deduce that the cost of all the deletions during the whole algorithm is no more than $O(n)$.

6.4.1.2. Shortening Updating our suffix skeleton tree as a result of the *shortening* operation (caused by each member of *Affects(S)* or *Affects(T)*) demands first deleting all the nodes representing the shortened substring. This is done as we have just explained for the deletion operation. Afterwards, we need to reinsert the shortened version of the substring. This can be done in a similar manner to what we described for the regular suffix tree in Section 5, which means we use a partial Wiener construction. At each stage of reinsertion we do just local changes at the node we reach, adding a new node to one of the special nodes linked-list. This, of course, can be done in constant time. Therefore, as we observe in Section 4 the overall cost of the shortening operation is $O(n)$.

6.4.2. *LPB array updates*

Now, we show how to update the LPB array quickly. We maintain for each pointer in the array that points to an essential node a pointer in the opposite direction. This way when we delete a node from our suffix skeleton tree we can also delete the pointer to that node from the LBP array in a constant time. A shortening operation starts with the described deletion. Then, we insert a new pointer into the proper bucket in the LPB array. A new pointer is inserted only in case the parent of the special node to which we have added the shortened substring did not have both an S-node and a T-node before the reinsertion but has them now. We maintain the string-depth of each essential node during the suffix skeleton tree construction in order to help us find the right bucket in constant time. Adding a new pointer to the array is also done in constant time as we just insert the newly created pointer to the front of the linked-list of the bucket.

7. Fast-GREEDY

Putting all the pieces together, we have a *Fast-GREEDY* implementation of the GREEDY scheme which is outlined in Algorithm 3.

Algorithm 3 Fast-GREEDY(S, T).

```
1: Initialization
2: Construct a suffix tree of $S\#T\$$
3: Traverse the tree and mark each node denoting whether it has suffixes from $S$, $T$ or both.
4: Create a suffix skeleton tree using the marking.
5: Allocate an empty LPB array of length $n = |S| = |T|$.
6: Traverse the tree and fill the LPB array's buckets.
7: Processing
8: while input strings are not empty do
9:     Find the current LCS using maximal pointer of the LPB array.
10:    Delete occurrences of current LCS from both $S$ and $T$ after outputting their starting indices.
11:    Update our data structures following the changes to the 4 groups: $Delete(T)$, $Affects(T)$, $Delete(S)$, $Affects(S)$.
12: end while
```

Hence,

Theorem 1. *The GREEDY algorithm for the Minimum Common String Partition problem can be implemented in $O(n)$ time for strings S and T of length n each.*

References

[1] Nikhil Bansal, Moshe Lewenstein, Bin Ma, Kaizhong Zhang, On the longest common rigid subsequence problem, Algorithmica 56 (2) (2010) 270–280.
[2] Laurent Bulteau, Guillaume Fertin, Christian Komusiewicz, Irena Rusu, A fixed-parameter algorithm for minimum common string partition with few duplications, in: WABI, 2013, pp. 244–258.
[3] Laurent Bulteau, Christian Komusiewicz, Minimum common string partition parameterized by partition size is fixed-parameter tractable, in: SODA, 2014, pp. 102–121.
[4] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, T. Jiang, Assignment of orthologous genes via genome rearrangement, IEEE/ACM Trans. Comput. Biology Bioinform. 2 (4) (2005) 302–315.
[5] D.A. Christie, R.W. Irving, Sorting strings by reversals and by transpositions, SIAM J. Discrete Math. 14 (2) (2001) 193–206.
[6] Marek Chrobak, Petr Kolman, Jiri Sgall, The greedy algorithm for the minimum common string partition problem, ACM Trans. Algorithms 1 (2) (2005) 350–366.
[7] Raphaël Clifford, Zvi Gotthilf, Moshe Lewenstein, Alexandru Popa, Restricted common superstring and restricted common supersequence, in: Symposium on Combinatorial Pattern Matching, CPM, 2011, pp. 467–478.
[8] G. Cormode, S. Muthukrishnan, The string edit distance matching problem with moves, in: Proceedings of the Symposium on Discrete Algorithms, SODA 2002, 2002, pp. 667–676.
[9] P. Damaschke, Minimum common string partition parameterized, in: Proceedings of the Workshop on Algorithms in Bioinformatics, WABI 2008, 2008, pp. 87–98.
[10] M. Farach, Optimal suffix tree construction with large alphabets, in: Proceedings of the Symposium on Foundations of Computer Science, FOCS 1997, 1997, pp. 137–143.
[11] A. Goldstein, P. Kolman, J. Zheng, Minimum common string partition problem: hardness and approximations, Electron. J. Combin. 12 (2005).
[12] Zvi Gotthilf, Danny Hermelin, Gad M. Landau, Moshe Lewenstein, Restricted *LCS*, in: Symposium on String Processing and Information Retrieval, SPIRE, 2010, pp. 250–257.
[13] Zvi Gotthilf, Moshe Lewenstein, Alexandru Popa, On shortest common superstring and swap permutations, in: Symposium on String Processing and Information Retrieval (SPIRE), 2010, pp. 270–278.
[14] S. Hannenhalli, P.A. Pevzner, Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals, J. ACM 46 (1) (1999) 1–27.
[15] Haitao Jiang, Binhai Zhu, Daming Zhu, Hong Zhu, Minimum common string partition revisited, J. Comb. Optim. 23 (4) (2012) 519–527.
[16] Haim Kaplan, Nira Shafrir, The greedy algorithm for edit distance with moves, Inf. Process. Lett. 97 (1) (2006) 23–27.
[17] P. Kolman, Approximating reversal distance for strings with bounded number of duplicates, in: Proceedings of the Symposium on Mathematical Foundations of Computer Science, MFCS 2005, 2005, pp. 580–590.

[18] P. Kolman, T. Walen, Reversal distance for strings with duplicates: linear time approximation using hitting set, Electron. J. Combin. 14 (1) (2007).
[19] D. Lopresti, A. Tomkins, Block edit models for approximate string matching, Theoret. Comput. Sci. 181 (1) (1997) 159–179.
[20] E.M. McCreight, A space-economical suffix tree construction algorithm, J. ACM 23 (2) (1976) 262–272.
[21] D. Sankoff, J.B. Kruskal (Eds.), Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison, Addison-Wesley, Reading, MA, 1983.
[22] D. Shapira, J.A. Storer, Edit distance with move operations, J. Discrete Algorithms 5 (2) (2007) 380–392.
[23] W.F. Tichy, The string-to-string correction problem with block moves, ACM Trans. Comput. Syst. 2 (4) (1984) 309–321.
[24] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.
[25] P. Weiner, Linear pattern matching algorithms, in: Proceedings of the Symposium on Switching and Automata Theory, 1973, pp. 1–11.