Reoptimization of the Shortest Common Superstring Problem*
(Extended Abstract)

Davide Bilò[1], Hans-Joachim Böckenhauer[2], Dennis Komm[2], Richard Královič[2], Tobias Mömke[2], Sebastian Seibert[2], and Anna Zych[2]

[1] Department of Computer Science, University of L'Aquila
dbilo@inf.ethz.ch
[2] Department of Computer Science, ETH Zurich, Switzerland
{hjb, dennis.komm, richard.kralovic, tobias.moemke, sseibert, anna.zych}@inf.ethz.ch

Abstract. A reoptimization problem describes the following scenario: Given an instance of an optimization problem together with an optimal solution for it, we want to find a good solution for a locally modified instance.

In this paper, we deal with reoptimization variants of the shortest common superstring problem where the local modifications consist of adding or removing a single string. We show NP-hardness of these reoptimization problems and design several approximation algorithms for them.

1 Introduction

In classical algorithmics, one is interested in finding good feasible solutions to input instances about which nothing is known in advance. Unfortunately, many practically relevant problems are computationally hard, and so different approaches such as approximation algorithms or heuristics are used for computing good approximations for optimal solutions. In the real world, however, some extra knowledge about the instance at hand might be already known. The concept of reoptimization employs a special kind of additional knowledge: Under the assumption that we are given an instance of an optimization problem together with an optimal solution for it, we want to efficiently compute a good solution for a locally modified input instance.

This concept of reoptimization was mentioned for the first time in [13] in the context of postoptimality analysis for some scheduling problem. Postoptimality analysis deals with the related question of how much an instance may be altered without changing the set of optimal solutions, see, e. g., [17]. Since then, the concept of reoptimization has been successfully applied to various problems like

* This work was partially supported by SNF grant 200021-121745/1 and SBF grant C 06.0108 as part of the COST 293 (GRAAL) project funded by the European Union.

the traveling salesman problem [1, 3, 7, 10], the Steiner tree problem [4, 8, 11], the knapsack problem [2], and various covering problems [5]. A survey of reoptimization problems can be found in [9].

In this paper, we investigate some reoptimization variants of the *shortest common superstring problem*, SCS for short. Given a substring-free set of strings, the SCS asks for a shortest common superstring of S, i. e., for a minimum-length string containing all strings from S as substrings. The SCS is one of the most prominent hard problems in stringology with many applications, e. g., in computational biology where it is used for modeling certain aspects of the DNA fragment assembly problem (see, for instance, [6, 14] for more details). The SCS is known to be NP-hard [12] and even APX-hard [18]. Many approximation algorithms have been devised for the SCS, the best-known being a greedy algorithm proposed by Tarhio and Ukkonen [16] which can be proven to achieve an approximation ratio of 4, but is conjectured to be 2-approximative. The currently best known approximation algorithm achieves a ratio of 2.5 [15].

In this paper, we deal with reoptimizing the SCS under the local modifications of adding or removing a single string. Our main results are the following. We show that both reoptimization versions of the SCS are NP-hard and propose some approximation algorithms for them. First, we devise an iteration technique for improving the approximation ratio of any SCS algorithm in the presence of a long string in the input which might be of independent interest. Then, we use this iteration technique to design an algorithm for SCS reoptimization which gives an approximation ratio arbitrarily close to 1.6 for adding a string and a ratio arbitrarily close to 13/7 for removing a string. This algorithm uses some known approximation algorithm for the original SCS (without reoptimization), and its approximation ratio depends on the ratio of this SCS algorithm. Thus, any improvement over the best known ratio of 2.5 for the SCS immediately yields also an improvement of these reoptimization results. Since the running time of this iterative algorithm is rather high, we also analyze a simple and fast reoptimization algorithm for adding a string and prove an approximation ratio of 11/6 for it.

The paper is organized as follows. In Section 2, we formally define the reoptimization variants of the SCS and fix our notation. Section 3 is devoted to the hardness results, in Section 4, we present the iterative reoptimization algorithms, and Section 5 contains the analysis of the fast approximation algorithm for adding a string.

2 Preliminaries

We start with defining some notations for dealing with strings that we will use throughout the paper. By λ we denote the empty string. The concatenation of two strings s and t will be written as $s \cdot t$, or as st for short. Let s, t, x, and y be some (possibly empty) strings such that $t = xsy$. Then s is a *substring* of t, we write $s \sqsubseteq t$, and t is a *superstring* of s. If x is empty, we say that s is a *prefix* of t, if y is empty, then s is a *suffix* of t. We say that a set S of strings is *substring-free* if $s \not\sqsubseteq t$, for all $s, t \in S$.

For two strings s_1 and s_2, the *overlap* $\text{ov}(s_1, s_2)$ of s_1 and s_2 is the maximum-length *proper* suffix of s_1 which is also a *proper* prefix of s_2, i. e., we additionally require that $s_1, s_2 \not\sqsubseteq \text{ov}(s_1, s_2)$. The corresponding prefix of s_1, i. e., the string p such that $s_1 = p \cdot \text{ov}(s_1, s_2)$, is denoted by $\text{pref}(s_1, s_2)$. The *merge* of s_1 and s_2 is defined as $\text{merge}(s_1, s_2) := \text{pref}(s_1, s_2) \cdot s_2$. We inductively extend this notion of merge to more than two strings by defining

$$\text{merge}(s_1, \ldots, s_m) = \text{merge}(\text{merge}(s_1, \ldots, s_{m-1}), s_m).$$

We call a string s *periodic* with period π, if there exist a suffix $\underline{\pi}$ and a prefix $\overline{\pi}$ of the string π and some $k \in \mathbb{N}$ such that $s = \underline{\pi} \cdot \pi^k \cdot \overline{\pi}$. In this case, we also write $s \sqsubseteq \pi^\infty$.

The problem we are investigating in this paper is to find the shortest common superstring for a given set $S = \{s_1, \ldots, s_m\}$ of strings. If S is substring-free, then the shortest common superstring can be unambiguously described by the order in which the strings appear in it: If $s_{i_1}, \ldots, s_{i_m}$ is the order of appearance in a shortest superstring t, then $t = \text{merge}(s_{i_1}, \ldots, s_{i_m})$. This observation leads to the following formal definition of the problem.

Definition 1. *The* shortest common superstring problem*, SCS for short, is the following optimization problem: Given a substring-free set of strings $S = \{s_1, \ldots, s_m\}$, the feasible solutions are all permutations $(s_{i_1}, \ldots, s_{i_m})$ of S. For any feasible solution* $\text{Sol} = (s_{i_1}, \ldots, s_{i_m})$*, the cost is* $|\text{Sol}| = |\text{merge}(s_{i_1}, \ldots, s_{i_m})|$*, i. e., the length of the shortest superstring for S containing the strings from S in the order as given by* Sol*. The goal is to find a permutation minimizing the length of the corresponding superstring.*

In this paper, we deal with two reoptimization variants of the SCS. The local modifications we consider here are adding a string to our set of input strings or deleting one string from it. The corresponding reoptimization problem can be formally defined as follows.

Definition 2. *The input for the* SCS reoptimization problem with adding a string*, SCS+ for short, consists of a substring-free set $S_O = \{s_1, \ldots, s_m\}$ of strings, an optimal SCS-solution* Opt_O *for it, and a string s_{new} such that also $S_N = S_O \cup \{s_{new}\}$ is substring-free.*

Analogously, the input for the SCS reoptimization problem with removing a string*, SCS– for short, consists of a substring-free set of strings $S_O = \{s_1, \ldots, s_m\}$, an optimal SCS-solution* Opt_O *for it, and a string $s_{old} \in S_O$. In this case, $S_N = S_O \setminus \{s_{old}\}$. For both problems, the goal is to find an optimal SCS-solution* Opt_N *for S_N.*

In addition to the maximum overlap and merge as defined above, we also consider the overlap and merge inside a given solution. Let Sol be some solution for an SCS instance given by a set of strings S and let s and t be two strings from S which are not necessarily overlapping in Sol. Then $\mathrm{ov}_{\mathrm{Sol}}(s,t)$ denotes the overlap of s and t in Sol, and we use $\mathrm{merge}_{\mathrm{Sol}}(s,t) = \mathrm{merge}(s,\ldots,t)$ as an abbreviation for the merge of s and t together with all input strings lying between them in Sol. By $\mathrm{prefM}_{\mathrm{Sol}}(s,t)$, we denote the prefix of $\mathrm{merge}_{\mathrm{Sol}}(s,t)$ such that $\mathrm{prefM}_{\mathrm{Sol}}(s,t) \cdot t = \mathrm{merge}_{\mathrm{Sol}}(s,t)$. Note that s may be a proper prefix of $\mathrm{prefM}_{\mathrm{Sol}}(s,t)$. For $\mathrm{Sol} = \mathrm{Opt}_O$, we use the notations ov_O, merge_O, and prefM_O for $\mathrm{ov}_{\mathrm{Opt}_O}$, $\mathrm{merge}_{\mathrm{Opt}_O}$, and $\mathrm{prefM}_{\mathrm{Opt}_O}$, respectively. Analogously, we use ov_N, merge_N, and prefM_N for $\mathrm{Sol} = \mathrm{Opt}_N$. Note that, for two consecutive strings s and t inside some solution Sol, $\mathrm{merge}_{\mathrm{Sol}}(s,t) = \mathrm{merge}(s,t)$, but this equality does not necessarily hold for non-consecutive strings.

3 Hardness Results

In this section, we show that the considered reoptimization problems are NP-hard. Similarly to [9], we use a polynomial-time Turing reduction since we rely on repeatedly applying reoptimizations.

Theorem 1. *The problems SCS+ and SCS– are NP-hard.*

Proof. We split the reduction into several steps. Given an input instance I for SCS, we define a corresponding easily solvable instance I'. Then we show that I' is indeed solvable in polynomial time. Finally, we show how to use polynomially many reoptimization steps in order to transform the optimal solution for I' into an optimal solution for I.

For any SCS+ instance I, the easy instance I' consists of no strings. Obviously, the empty vector is an optimal solution for I'. Now, I' can be transformed into any instance I by adding all strings from I one after the other. Thus, SCS+ is NP-hard.

Now, let us consider the local modification of removing strings. Let I be an instance for SCS that consists of m strings $s_1, s_2, \ldots, s_m$. For any i, let s_i^f be the first symbol of s_i, let s_i^l be its last symbol, and let s_i^c be s_i without the first and last symbol. Without loss of generality, we exclude strings of length 1 since they cannot significantly increase the hardness of any input instance.

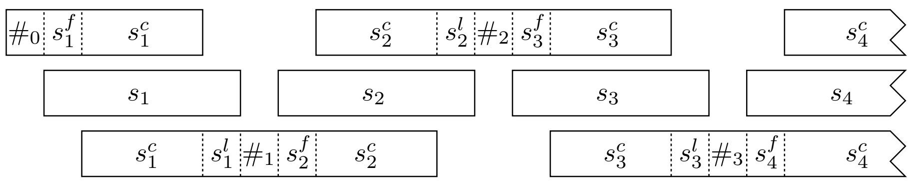


Fig. 1. An optimal solution for the easily solvable instance I'

Now, we construct I' as follows. Let $\#_0, \#_1, \ldots, \#_m$ be $m+1$ special symbols that do not appear in I. Then, we introduce the set of strings $S' :=$

$\{s'_0, s'_1, \ldots, s'_m\}$, where $s'_0 := \#_0 s_1^f s_1^c$, $s'_m := s_m^c s_m^l \#_m$, and $s'_i := s_i^c s_i^l \#_i s_{i+1}^f s_{i+1}^c$, for each $i \in \{1, \ldots, m-1\}$. Let the instance I' be the set of the strings from I together with the strings from S'. It is clear that $m+1$ local modifications, each removing one of the new strings, transform I' into I. Thus, it only remains to show that I' is efficiently solvable. To this end, we claim that no algorithm can do better than alternating the new and the old strings as depicted in Fig. 1.

We now formally prove the correctness of the construction above. First, observe that the constructed instance is substring-free. Now, let us only consider the strings from S'. We will show that at each position of any common superstring at most two of these strings can overlap. Suppose, conversely, that more than two of the strings overlap. Then there are pairwise disjoint numbers i,j, and k between 0 and m such that s'_i, s'_j, and s'_k overlap in at least one symbol. Let, without loss of generality, s'_i be the leftmost string and let s'_k be the rightmost string in an overlapping setting. But then each symbol of the middle string, s'_j, is overlapped by at least one of the other strings — a contradiction, because the symbol $\#_j$ only appears in s'_j.

Following the construction of S', the overall length $\sum_{i=0}^{m} |s'_i|$ of the strings from S' is $m+1+2\cdot\sum_{i=1}^{m}(|s_i|-1)$. Since only non-special symbols can overlap, any shortest superstring is at least $m+1+\sum_{i=1}^{m}(|s_i|-1)$ symbols long; otherwise, there would be some position in the superstring that overlaps with three strings from S'. Finally, we have to include the strings from I. To this end, we will show that adding m strings not containing special symbols to S' results in a lower bound of $m+1+\sum_{i=1}^{m}|s_i|$ on the length of any common superstring.

Note that, given a substring-free set of k strings, where w is the longest one, it cannot have a common superstring t_1 that is shorter than $|w|+(k-1)$, i. e.,

$$|t_1| \geq |w| + k - 1. \tag{1}$$

Similarly, given a substring-free set of $k+2$ strings exactly two of which contain special characters, namely $w_l = u_l \#_u u_r$ and $w_r = v_l \#_v v_r$, any common superstring starting with w_l has at least $|w_l| + k$ symbols and, analogously, any common superstring ending with w_r has at least $|w_r| + k$ symbols.

Given a common superstring t_2 which starts with w_l and ends with w_r, we have

$$|t_2| \geq |u_l| + 1 + |v_r| + 1 + \max\{|u_r|, |v_l|\} + k. \tag{2}$$

Now let us consider I'. Let t be a common superstring for I'. We decompose t into $w_0 w'_0 w_1 w'_1 w'_2 w_2 \ldots w_m w'_m w_{m+1}$ such that each w'_i consists of exactly one special symbol. Therefore, each string from I is contained in at least one of the strings between the special symbols. Let k_i be the number of strings from I that is contained in w_i. Then, according to (1) and (2), w_i is at least k_i symbols longer than the longer end of the two special strings belonging to w'_i and w'_{i+1}. Due to the estimation of the length of a shortest common superstring above, and since all m strings of I have to appear somewhere, i. e., $\sum_{i=0}^{m+1} k_i \geq m$, the length of t is at least

$$\sum_{i=1}^{m}(|s_i|-1)+m+1+\sum_{i=0}^{m+1}k_i \geq \sum_{i=1}^{m}(|s_i|-1)+m+1+m=\sum_{i=1}^{m}|s_i|+m+1.$$

But this is exactly the length of our constructed common superstring. Therefore, we conclude that SCS– is NP-hard.

4 Iterative Algorithms for Adding or Removing a String

Consider any polynomial approximation algorithm A for SCS with approximation ratio γ. We show how to construct a polynomial reoptimization algorithm for SCS+ with approximation ratio arbitrarily close to $(2\gamma-1)/\gamma$. Furthermore, we show a similar result for SCS– with approximation ratio $(3\gamma-1)/(\gamma+1)$. Since the best known polynomial approximation algorithm for SCS gives $\gamma = 2.5$, see [15], we obtain an approximation ratio arbitrarily close to $8/5 = 1.6$ for SCS+ and an approximation ratio arbitrarily close to $13/7 < 1.86$ for SCS–.

The core part of our reoptimization algorithms is an approximation algorithm for SCS that works well if the input instance contains at least one long string. More precisely, let $S = \{s_1, \ldots, s_m\}$ be an instance of SCS such that $\mu_0 \in S$ is a longest string in S, and let $|\mu_0| = \alpha_0|\mathrm{Opt}|$, for some $\alpha_0 > 0$, where Opt is an optimal solution of S.

The algorithm A_1 guesses the leftmost string l_1 and the rightmost string r_1 which overlap with μ_0 in the string corresponding to Opt, together with the respective overlap lengths. Afterwards, it computes a new instance S_1 by eliminating all substrings of $\mathrm{merge}_{\mathrm{Opt}}(l_1, \mu_0, r_1)$ from the instance S, calls the algorithm A on S_1 and appends l_1, μ_0, r_1 to the approximate solution returned by A.

Now we generalize A_1 by iterating this procedure k times. For arbitrary k, we construct a polynomial-time approximation algorithm A_k for SCS that computes a solution of length at most

$$\left(1+\frac{\gamma^k(\gamma-1)}{\gamma^k-1}(1-\alpha_0)\right)|\mathrm{Opt}|.$$

For every $i \in \{1, \ldots, k\}$, we define strings l_i, r_i, and μ_i as follows: Let l_i be the leftmost string that overlaps with μ_{i-1} in Opt. If there is no such string, $l_i := \mu_{i-1}$. Similarly, let r_i be the rightmost string that overlaps with μ_{i-1} in Opt. We define μ_i as $\mathrm{merge}_{\mathrm{Opt}}(l_i, \mu_{i-1}, r_i)$.

The algorithm A_k uses exhaustive search to find strings l_i, r_i and μ_i for every $i \in \{1, \ldots, k\}$. This can be done by assigning every possible string of S to l_i and r_i, and trying every possible overlap between l_i, μ_{i-1} and r_i. For every feasible candidate set of strings and for every i, the algorithm computes the candidate solution Sol_i corresponding to the string $\mathrm{merge}(u_i, \mu_i)$, where u_i is the string corresponding to the result of algorithm A on the input instance S_i obtained by removing all substrings of μ_i from S. Algorithm A_k then outputs the best solution among all candidate solutions.

Theorem 2. *Let n be the total length of all strings in S, i. e., $n = \sum_{j=1}^{m} |s_j|$. Algorithm A_k works in time $O(m^{2k}n^{2k}(kmn + kT(m,n)))$, where $T(m,n)$ is the time complexity of algorithm A on an input instance with at most m strings of total length at most n.*

Proof. Algorithm A_k needs to test all $O(m^{2k})$ possibilities for choosing $2k$ strings $l_1, r_1, \ldots, l_k, r_k$ from the m strings of S. For every such possibility, it must test all possible overlaps between the strings in order to obtain strings $\mu_1, \ldots, \mu_k$. Hence, the lengths of $2k$ overlaps must be tested. As the length of each overlap can be in the range from 0 to n, there are $O(n^{2k})$ possibilities. For each of the $O(m^{2k}n^{2k})$ possibilities, A_k tests if it is feasible (this can be done in time $O(n)$) and computes the corresponding k candidate solutions. To compute one candidate solution Sol_i, the instance S_i is prepared in time $O(mn)$ and algorithm A is executed in time $T(m,n)$. □

Theorem 3. *Algorithm A_k finds a solution of S of length at most*

$$\left(1 + \frac{\gamma^k(\gamma - 1)}{\gamma^k - 1}(1 - \alpha_0)\right) |\mathrm{Opt}|.$$

Proof. Assume that A_k outputs a solution of length greater than $(1+\beta)|\mathrm{Opt}|$, for some $\beta > 0$. In the analysis, we focus on the part of the computation of A_k where the correct assignment of strings l_i, r_i, and μ_i is analyzed. By our assumption, every candidate solution Sol_i has length greater than $(1+\beta)|\mathrm{Opt}|$. The solution Sol_i corresponds to the string $\mathrm{merge}(u_i, \mu_i)$, where $|\mu_i| = \alpha_i|\mathrm{Opt}|$, for some $\alpha_i > 0$, and u_i is the result of algorithm A on the input instance S_i. Hence, $|\mathrm{Sol}_i| \leq |u_i| + |\mu_i|$.

It is not difficult to check that, if we remove all substrings of μ_i from Opt, we obtain a feasible solution for S_i of length at most $|\mathrm{Opt}| - |\mu_{i-1}| = (1-\alpha_{i-1})|\mathrm{Opt}|$: By definition of μ_i, we have removed every string that overlapped with μ_{i-1}. Hence, $|u_i| \leq \gamma(1-\alpha_{i-1})|\mathrm{Opt}|$, and

$$(1+\beta)|\mathrm{Opt}| < |\mathrm{Sol}_i| \leq (\gamma(1-\alpha_{i-1}) + \alpha_i)|\mathrm{Opt}|. \tag{3}$$

Inequality (3) implies

$$\alpha_i > 1 + \beta - \gamma + \gamma\alpha_{i-1}. \tag{4}$$

Solving the system of recurrent equations (4) yields

$$\alpha_k > (1 + \beta - \gamma)\frac{\gamma^k - 1}{\gamma - 1} + \gamma^k\alpha_0. \tag{5}$$

Since μ_i is a substring of Opt for every i, it holds that $\alpha_k \leq 1$. Putting this together with (5) yields

$$\beta \leq \frac{\gamma^k(\gamma - 1)}{\gamma^k - 1}(1 - \alpha_0).$$

□

4.1 Reoptimization of SCS+

We now employ the iterative SCS algorithm described above for designing an approximation algorithm for SCS+. For every k, we define the algorithm A_k^+ for SCS+ as follows. Given an input instance S_O, its optimal solution Opt_O, and a new string s_{new}, the algorithm A_k^+ returns the solution Sol_1 corresponding to $\mathrm{merge}(\mathrm{Opt}_O, s_{new})$ or the solution Sol_2 computed by A_k for the input instance $S_N := S_O \cup \{s_{new}\}$, whichever is better.

Theorem 4. *Algorithm A_k^+ yields a solution of length at most*

$$\frac{2\gamma^{k+1} - \gamma^k - 1}{\gamma^{k+1} - 1}|\mathrm{Opt}_N|.$$

Proof. Let $|s_{new}| = \alpha|\mathrm{Opt}_N|$. Then $|\mathrm{Sol}_1| \leq (1+\alpha)|\mathrm{Opt}_N|$. Since S_N contains a string of length at least $\alpha|\mathrm{Opt}_N|$, Theorem 3 ensures that

$$|\mathrm{Sol}_2| \leq \left(1 + \frac{\gamma^k(\gamma-1)}{\gamma^k - 1}(1-\alpha)\right)|\mathrm{Opt}_N|.$$

Hence, the minimum of $|\mathrm{Sol}_1|$ and $|\mathrm{Sol}_2|$ is maximal if

$$(1+\alpha)|\mathrm{Opt}_N| = \left(1 + \frac{\gamma^k(\gamma-1)}{\gamma^k - 1}(1-\alpha)\right)|\mathrm{Opt}_N|,$$

which happens if

$$\alpha = \frac{\gamma^{k+1} - \gamma^k}{\gamma^{k+1} - 1}.$$

In this case, A_k^+ yields a solution of length at most

$$(1+\alpha)|\mathrm{Opt}_N| = \frac{2\gamma^{k+1} - \gamma^k - 1}{\gamma^{k+1} - 1}|\mathrm{Opt}_N|.$$

□

By choosing k sufficiently large, the approximation ratio of A_k^+ can be made arbitrarily close to $(2\gamma-1)/\gamma$. Algorithm A_k^+ is polynomial for every k, but the degree of the polynomial grows with k.

4.2 Reoptimization of SCS–

Similarily as for the case of SCS+, we define algorithm A_k^- for SCS– as follows. Given an input instance S_O, its optimal solution Opt_O and a string $s_{old} \in S_O$ to be removed, A_k^- returns the solution Sol_1 obtained from Opt_O by leaving out s_{old}, or the solution Sol_2 computed by A_k for input instance $S_N := S_O \setminus \{s_{old}\}$, whichever is better.

Theorem 5. *Algorithm A_k^- yields a solution of length at most*

$$\frac{3\gamma^{k+1} - \gamma^k - 2}{\gamma^{k+1} + \gamma^k - 2} |\mathrm{Opt}_N|.$$

Proof. Let $l \in S_O$ ($r \in S_O$) be the string that immediately precedes (follows) s_{old} in Opt_O, respectively. We focus on the case where both l and r exist, the other cases are analogous. It is easy to see that

$$|\mathrm{Sol}_1| \leq |\mathrm{Opt}_O| - |s_{old}| + |\mathrm{ov}(l, s_{old})| + |\mathrm{ov}(s_{old}, r)|.$$

Since augmenting Opt_N with s_{old} yields a feasible solution for S_O, we have $|\mathrm{Opt}_O| \leq |\mathrm{Opt}_N| + |s_{old}|$.

Without loss of generality, assume that $|\mathrm{ov}(s_{old}, r)| \leq |\mathrm{ov}(l, s_{old})| = \alpha|\mathrm{Opt}_N|$. Hence, $|\mathrm{Sol}_1| \leq (1 + 2\alpha)|\mathrm{Opt}_N|$. Furthermore, S_N contains the string l of length at least $\alpha|\mathrm{Opt}_N|$, so Theorem 3 ensures that

$$|\mathrm{Sol}_2| \leq \left(1 + \frac{\gamma^k(\gamma - 1)}{\gamma^k - 1}(1 - \alpha)\right) |\mathrm{Opt}_N|.$$

The minimum of $|\mathrm{Sol}_1|$ and $|\mathrm{Sol}_2|$ is maximal if

$$(1 + 2\alpha)|\mathrm{Opt}_N| = \left(1 + \frac{\gamma^k(\gamma - 1)}{\gamma^k - 1}(1 - \alpha)\right) |\mathrm{Opt}_N|,$$

which happens if

$$\alpha = \frac{\gamma^{k+1} - \gamma^k}{\gamma^{k+1} + \gamma^k - 2}.$$

In this case, A_k^- yields a solution of length at most

$$\frac{3\gamma^{k+1} - \gamma^k - 2}{\gamma^{k+1} + \gamma^k - 2} |\mathrm{Opt}_N|.$$

□

Similarly as in the case of SCS+, the approximation ratio of A_k^- can be made arbitrarily close to $(3\gamma - 1)/(\gamma + 1)$ by choosing k sufficiently large.

5 One-Cut Algorithm for Adding a String

In this section, we present a simple and fast algorithm OneCut for SCS+ and prove that it achieves an 11/6-approximation ratio. The algorithm cuts Opt_O at all positions one by one. Recall that the given optimal solution Opt_O is represented by an ordering of the input strings, thus cutting Opt_O at some position yields a partition of the input strings into two sub-orderings. The two corresponding strings are then merged with s_{new} in between. The algorithm returns a shortest of the strings obtained in this manner, see Algorithm 1.

Algorithm 1 OneCut

Input: A set of strings $S = \{s_1, \dots, s_m\}$, an optimal solution $\mathrm{Opt}_O = (s_1, \dots, s_m)$ for S, and a string s_{new}

for $i \in \{0, \dots, m\}$ **do**

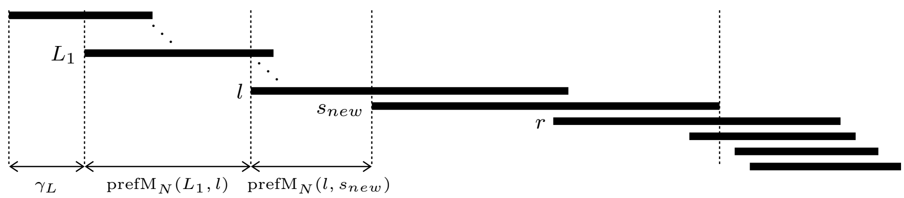
  Let $\mathrm{Solution}_i := (s_1, \dots, s_i, s_{new}, s_{i+1}, \dots, s_m)$.

Output: A best of the obtained solutions $\mathrm{Solution}_i$, for $0 \le i \le m$

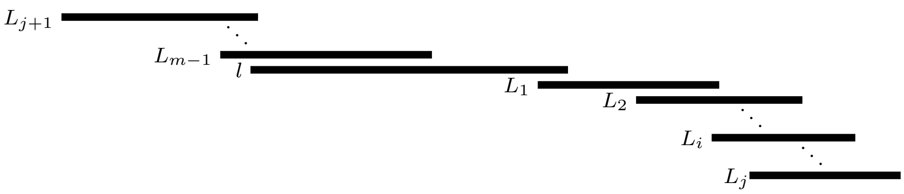
Theorem 6. *The algorithm* OneCut *is an* $11/6$*-approximation algorithm for SCS+ running in time* $\mathcal{O}(n \cdot m)$ *for inputs consisting of* m *strings of total length* n *over a constant-size alphabet.*

Proof sketch. We first analyze the running time of OneCut. Using suffix trees, we can compute all pairwise overlaps of $\{s_{new}, s_1, \dots, s_m\}$ in time $\mathcal{O}(n \cdot m)$, see e. g. [16]. Using these precomputed overlaps, each of the $m + 1$ iterations of OneCut can be performed in constant time. Thus, the overall running time of OneCut is also in $\mathcal{O}(n \cdot m)$.

We now show that OneCut provides an approximation ratio of 11/6 for SCS+. The proof is constructed in the following manner. One by one, we eliminate cases in which we can prove a ratio of 11/6 for OneCut, until all cases are covered. Each time we prove a ratio of 11/6 under some condition, we can deal in the following with the remaining cases under the assumption that this condition does not hold. In this way, we construct a list of assumptions which eventually lead to some final case. Due to the space limitations, the proofs of the lemmas are omitted in this extended abstract.



(a) The new optimal solution Opt_N (in the case that L_1 precedes l)



(b) The old optimal solution Opt_O (in the case that $L_i \neq \lambda$)

Fig. 2. The new and old optimal solution

Lemma 1. *If* $|s_{new}| \leq \frac{5}{6}|\mathrm{Opt}_N|$*, then the algorithm* OneCut *provides an* 11/6*-approximation ratio.*

Lemma 1 shows that the desired approximation ratio can be reached whenever the string s_{new} is short. This leads to the first assumption.

Assumption 1 $|s_{new}| > \frac{5}{6}|\mathrm{Opt}_N|$.

Let l be the string directly preceding s_{new} in Opt_N and let r be the direct successor of s_{new} in Opt_N (see Fig. 2 (a)). Lemma 2 proves that we may assume, without loss of generality, that l and r almost completely cover the string s_{new}.

Lemma 2. *If* OneCut *returns an* 11/6*-approximation for all instances where there is at most one letter from* s_{new} *not covered in* Opt_N *by either* l *or* r*, then it returns an* 11/6*-approximation in general.*

Assumption 2 *In* Opt_N*, at most one letter of the string* s_{new} *is not covered by either* l *or* r*.*

Lemma 3. *Assumption 2 implies that either* $|s_{new}| \leq \frac{1}{2}|\mathrm{Opt}_N| + |\mathrm{ov}(l, s_{new})|$ *or* $|s_{new}| \leq \frac{1}{2}|\mathrm{Opt}_N| + |\mathrm{ov}(s_{new}, r)|$*.*

By Lemma 3 and Assumption 2, without loss of generality, we may assume the following.

Assumption 3 $|s_{new}| \leq \frac{1}{2}|\mathrm{Opt}_N| + |\mathrm{ov}(l, s_{new})|$.

We now enumerate the strings in Opt_O according to the position of l as shown in Fig. 2 (b):

$$\mathrm{Opt}_O = (L_{j+1}, \dots, L_{m-1}, l, L_1, \dots, L_i, \dots, L_j)$$

Thus, let L_1 be the direct successor of l in Opt_O. If l has no successor in Opt_O, let $L_1 = \lambda$ be the empty string. In this case, the strings preceding l in Opt_O are $L_2, \dots, L_m$, and we may assume that L_1 is located at the end of Opt_O.

In Lemma 4, we resolve the case where L_1 follows s_{new} in Opt_N.

Lemma 4. *Under Assumptions 1 and 3, if* L_1 *is located after* s_{new} *in* Opt_N*, then* OneCut *returns an* 11/6*-approximation.*

If $L_1 = \lambda$, we may assume that it follows l in Opt_N. Thus, we can add the following assumption.

Assumption 4 $L_1 \neq \lambda$ *and* L_1 *precedes* s_{new} *in* Opt_N*.*

We define $\pi_L = AB$, where $A = \mathrm{prefM}_N(L_1, l)$ and $B = \mathrm{prefM}_O(l, L_1) = \mathrm{pref}(l, L_1)$. Note that $L_1 = (AB)^g p_1$ and $l = (BA)^h p_2$ for some natural numbers g, h, where p_1 and p_2 denote some prefixes of AB and BA, respectively (see Fig. 3). Thus, $L_1, l \sqsubseteq \pi_L^\infty$. Now let L_i be the first string after L_1 in Opt_O which is not periodic with period π_L, i.e., $L_i \not\sqsubseteq \pi_L^\infty$. If there is no such string, let $L_i = \lambda$ be the empty string. Let $L = \mathrm{merge}_O(l, L_{i-1})$. Let γ_L denote the prefix of Opt_N preceding L_1 (see Fig. 2 (a)).

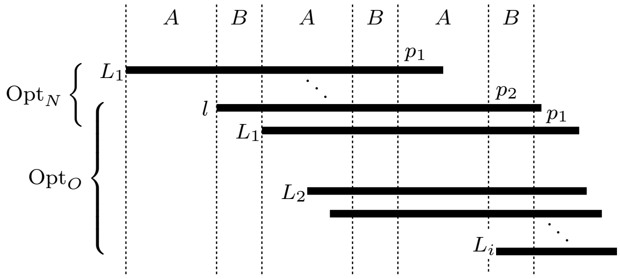


Fig. 3. Periodicity of l and L_1

Lemma 5. *Assumption 4 and* $|\pi_L| \geq \frac{1}{6}|\mathrm{Opt}_N| - |\gamma_L|$ *give approximation ratio* 11/6 *for* OneCut.

In Lemma 5, we have handled the case that the period π_L is relatively long, yielding the following assumption for the rest of the proof.

Assumption 5 $|\pi_L| + |\gamma_L| \leq \frac{1}{6}|\mathrm{Opt}_N|$.

The 11/6-approximation is proven in Lemma 6 for the case where L_i follows s_{new} in Opt_N.

Lemma 6. *Under Assumptions 1, 2, 3, 4, and 5, and if L_i follows s_{new} in* Opt_N*,* OneCut *is an* 11/6*-approximation algorithm for SCS+.*

Thus, we can make the following assumption for our final case. (In the case where $L_i = \lambda$, we may assume that L_i follows s_{new} in Opt_N.)

Assumption 6 $L_i \neq \lambda$ *and L_i precedes s_{new} in* Opt_N.

In the final case of the proof, as presented in Lemma 7, we will use Assumptions 1 to 6 to prove our claim for all remaining situations not previously dealt with.

Lemma 7. *Under Assumptions 1, 2, 3, 4, 5, and 6,* OneCut *provides an* 11/6*-approximation ratio for SCS+.*

This completes the proof of Theorem 6. □

We now show that the analysis in the proof of Theorem 6 is tight.

Theorem 7. *Algorithm* OneCut *cannot achieve an* $(\frac{11}{6} - \varepsilon)$*-approximation, for any* $\varepsilon > 0$.

Proof. For any $n \in \mathbb{N}$, we construct an input instance that consists of the following strings:

$$S_O = \{\vdash, xa^{n+2}x, a^{n+1}xa^{n+1}, a^n xa^{n+1}xa^n,$$
$$b^n yb^{n+1}yb^n, b^{n+1}yb^{n+1}, yb^{n+2}y, \dashv\}.$$

Obviously, arranging the strings in the order as presented forms an optimal solution Opt_O of length $6n + O(1)$:

$$\begin{array}{llllllllllll}
 & a^n & x & a^{n+1} & x & a^n & b^n & y & b^{n+1} & y & b^n & \\
 & a^{n+1} & x & a^{n+1} & & & & & b^{n+1} & y & b^{n+1} & \\
x & a^{n+2} & x & & & & & & & y & b^{n+2} & y \\
\vdash & & & & & & & & & & & \dashv
\end{array}$$

The corresponding superstring is $\vdash xa^{n+2}xa^{n+1}xa^nb^nyb^{n+1}yb^{n+2}y\dashv$. Let

$$s_{new} := b^{n-1}yb^{n+1}yb^n\#a^nxa^{n+1}xa^{n-1}.$$

It is easy to see that there is a solution for $S_N = S_O \cup \{s_{new}\}$ which has asymptotically the same length as Opt_O:

$$\begin{array}{lllllllllllll}
 & b^{n-1} & y & b^{n+1} & y & b^n & \# & a^n & x & a^{n+1} & x & a^{n-1} & \\
 & b^n & y & b^{n+1} & y & b^n & & a^n & x & a^{n+1} & x & a^n & \\
 & b^{n+1} & y & b^{n+1} & & & & & & a^{n+1} & x & a^{n+1} & \\
y & b^{n+2} & y & & & & & & & & x & a^{n+2} & x \\
\vdash & & & & & & & & & & & & \dashv
\end{array}$$

This new optimal solution Opt_N is obviously *unique* (except for the placement of the symbols $\vdash$ and $\dashv$ at the beginning or the end). Applying algorithm OneCut for inserting s_{new} into the instance when Opt_O is given, however, does not find a common superstring that is shorter than $11n + O(1)$ symbols.

Here, the crucial observation is that all strings in S_O need to be rearranged to construct Opt_N from Opt_O (which then means that no information is gained by the given additional knowledge). To be optimal, 7 cuts are necessary. Being allowed to only cut once, however, cannot yield a solution better than $11n+\mathcal{O}(1)$. Finally, we easily verify that $|\mathrm{Opt}_N| = 6n + \mathcal{O}(1)$. □

6 Conclusion

In this paper, we have investigated the reoptimization of SCS according to two different local modifications. Besides the results presented here, there is a straight-forward generalization of the algorithm OneCut. For any constant k, we can also allow k cuts. We expect that additional cuts lead to improved approximation ratios. It is not hard, however, to show some lower bounds on the approximation ratio for k cuts. Using the same hard instance as in the proof of Theorem 7, we can show that two cuts do not improve the approximation ratio. In general, for any $\varepsilon > 0$ and $k \geq 3$, we have constructed a hard input instance such that the approximation ratio of the k-cut algorithm is bounded from below by $1 + 2/(k+1) - \varepsilon$.

References

1. C. Archetti, L. Bertazzi, and M. G. Speranza. Reoptimizing the traveling salesman problem. *Networks*, 42(3):154–159, 2003.

2. C. Archetti, L. Bertazzi, and M. G. Speranza. Reoptimizing the 0-1 knapsack problem. Technical Report 267, University of Brescia, 2006.
3. G. Ausiello, B. Escoffier, J. Monnot, and V. T. Paschos. Reoptimization of minimum and maximum traveling salesman's tours. In *Proc. of the 10th Scandinavian Workshop on Algorithm Theory (SWAT 2006)*, volume 4059 of *LNCS*, pages 196–207. Springer, 2006.
4. D. Bilò, H.-J. Böckenhauer, J. Hromkovič, R. Královič, T. Mömke, P. Widmayer, and A. Zych. Reoptimization of Steiner trees. In *Proc. of the 11th Scandinavian Workshop on Algorithm Theory (SWAT 2008)*, volume 5124 of *LNCS*, pages 258–269. Springer, 2008.
5. D. Bilò, P. Widmayer, and A. Zych. Reoptimization of weighted graph and covering problems. In *Proc. of the 6th Workshop on Approximation and Online Algorithms (WAOA 2008)*, volume 5426 of *LNCS*, pages 201–213, 2009.
6. H.-J. Böckenhauer and D. Bongartz. *Algorithmic Aspects of Bioinformatics*. Natural Computing Series. Springer, 2007.
7. H.-J. Böckenhauer, L. Forlizzi, J. Hromkovič, J. Kneis, J. Kupke, G. Proietti, and P. Widmayer. Reusing optimal TSP solutions for locally modified input instances (extended abstract). In *Proc. of the Fourth IFIP International Conference on Theoretical Computer Science (IFIP TCS 2006)*, volume 209, pages 251–270. Springer, 2006.
8. H.-J. Böckenhauer, J. Hromkovič, R. Královič, T. Mömke, and P. Rossmanith. Reoptimization of Steiner trees: Changing the terminal set. *Theoretical Computer Science (to appear)*.
9. H.-J. Böckenhauer, J. Hromkovič, T. Mömke, and P. Widmayer. On the hardness of reoptimization. In *Proc. of the 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2008)*, volume 4910 of *LNCS*, pages 50–65. Springer, 2008.
10. H.-J. Böckenhauer and D. Komm. Reoptimization of the metric deadline TSP. In *Proc. of the 33rd International Symposium on Mathematical Foundations of Computer Science (MFCS 2008)*, volume 5162 of *LNCS*, pages 156–167. Springer, 2008.
11. B. Escoffier, M. Milanic, and V. T. Paschos. Simple and fast reoptimizations for the Steiner tree problem. Technical Report 2007-01, DIMACS, 2007.
12. J. Gallant, D. Maier, and J. A. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20(1):50–58, 1980.
13. M. W. Schäffter. Scheduling with forbidden sets. *Discrete Applied Mathematics*, 72(1-2):155–166, 1997.
14. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. Natural Computing Series. PWS Publishing Company, 1997.
15. Z. Sweedyk. A $2\frac{1}{2}$-approximation algorithm for shortest superstring. *SIAM Journal on Computing*, 29(3):954–986, 2000.
16. J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical Computer Science*, 57(1):131–145, 1988.
17. S. Van Hoesel and A. Wagelmans. On the complexity of postoptimality analysis of 0/1 programs. *Discrete Applied Mathematics*, 91(1-3):251–263, 1999.
18. V. Vassilevska. Explicit inapproximability bounds for the shortest superstring problem. In *Proc. of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS 2005)*, volume 3618 of *LNCS*, pages 793–800. Springer, 2005.