Approximating Reversal Distance for Strings with Bounded Number of Duplicates

Petr Kolman*
Charles University in Prague
Faculty of Mathematics and Physics
Department of Applied Mathematics
kolman@kam.mff.cuni.cz

Tomasz Waleń†
Warsaw University
Faculty of Mathematics, Informatics and Mechanics
walen@mimuw.edu.pl

May 25, 2006

Abstract

For a string $A = a_1 \dots a_n$, a *reversal* $\rho(i,j)$, $1 \le i \le j \le n$, transforms the string A into a string $A' = a_1 \dots a_{i-1} a_j a_{j-1} \dots a_i a_{j+1} \dots a_n$, that is, the reversal $\rho(i,j)$ reverses the order of symbols in the substring $a_i \dots a_j$ of A. In a case of signed strings, where each symbol is given a sign $+$ or $-$, the reversal operation also flips the sign of each symbol in the reversed substring. Given two strings, A and B, signed or unsigned, *sorting by reversals* (SBR) is the problem of finding the minimum number of reversals that transform the string A into the string B.

Traditionally, the problem was studied for permutations, that is, for strings in which every symbol appears exactly once. We consider a generalization of the problem, k-SBR, and allow each symbol to appear at most k times in each string, for some $k \ge 1$. The main result of the paper is an $O(k^2)$-approximation algorithm running in time $O(n)$. For instances with $3 < k \le O(\sqrt{\log n \log^* n})$, this is the best known approximation algorithm for k-SBR and, moreover, it is faster than the previous best approximation algorithm.

Key words. Approximation algorithms, String comparison, Edit distance, Sorting by reversals, Minimum common string partition.

1 Introduction

For a string $A = a_1 \dots a_n$, a *reversal* $\rho(i,j)$, $1 \le i \le j \le n$, transforms the string A into a string $A' = a_1 \dots a_{i-1} a_j a_{j-1} \dots a_i a_{j+1} \dots a_n$, that is, the reversal $\rho(i,j)$ reverses the order of symbols in the substring $a_i \dots a_j$ of A. In a case of signed strings, where each symbol is given a sign $+$ or $-$, the reversal operation also flips the sign of each symbol in the reversed substring. Given two strings, A and B, signed or unsigned, *sorting by reversals* (SBR) is the problem of finding the minimum number of reversals that transform the string A into the string B; this number, denoted by $\mathsf{SBR}(A,B)$, is called the *reversal distance* of A and B.

*Research done in part while visiting University of California at Riverside. Supported by NSF grants CCR-0208856 and ACI-0085910 and by project 1M0021620808 of MŠMT ČR.

†Partially supported by the Polish Scientific Research Committee (KBN) under grant GR-1946.

A necessary and sufficient condition for A and B to have a finite reversal distance is that each letter appears the same number of times in A and B (for the signed version, we count together the occurrences of a letter with positive and negative signs). We call such strings *related.*

To give an example, $A = abcabc$ and $B = bcbaac$ are related strings and $\rho(3,6), \rho(1,4)$ is a sequence of reversals that turns A into B, therefore $\mathsf{SBR}(A,B) \leq 2$. Similarly, $\rho(1,4), \rho(4,4)$ turns $A' = +a-c-b-a+b+c$ into $B' = +a+b+c+a+b+c$ and thus, $\mathsf{SBR}(A',B') \leq 2$.

In this paper we study a variant of the problem, denoted by k-SBR, in which each symbol is allowed to appear at most k times in each string. Our particular interest is in the case of small values of k. The main contribution is an $O(k^2)$-approximation algorithm for k-SBR running in time $O(n)$. In particular, for $k = O(1)$ which is of interest for comparisons of genomic sequences, we have a linear time $O(1)$-approximation algorithm.

Preliminary version of this work was presented at the 30^{th} International Symposium on Mathematical Foundations of Computer Science [15].

1.1 Terminology

For notational simplicity, we allow a few symbols to have slightly different meanings for signed and unsigned strings. For a string $P = a_1 \ldots a_n$, we denote by $-P$ the result of a reversal $\rho(1,n)$ of P (e.g., for $P = +a+b-d$, we have $-P = +d-b-a$). We use two different equivalence relations. Two strings $A = a_1a_2 \ldots a_n$ and $B = b_1b_2 \ldots b_n$, signed or unsigned, are *identical*, $A = B$, if $a_i = b_i$ for each $i \in [n]$. In a case of signed strings, by $a_i = b_i$ we mean also equality of the signs. Signed or unsigned strings A and B are *congruent*, $A \cong B$, if $A = B$ or $A = -B$.

The length of a string A is denoted by $|A|$. A substring S of A is a *proper substring* if $|S| < |A|$. A *partition* of a string A is a sequence $\mathcal{P} = (P_1, P_2, \ldots, P_m)$ of strings whose concatenation is equal to A, that is, $P_1P_2 \ldots P_m = A$. The strings P_i are called the *blocks* of $\mathcal{P}$ and their number is the *size* of the partition. Given a partition $\mathcal{P} = (P_1, P_2, \ldots, P_m)$, of a string A, a pair $l, l+1$ is a *break* of the partition $\mathcal{P}$ if $l = \sum_{j=1}^{i} |P_j|$ for some $i \in [m-1]$. Informally, a break of a partition $\mathcal{P}$ of A is a pair of letters that are consecutive in A but are not consecutive in $\mathcal{P}$.

For two strings A and B, we say that S is a *common substring with respect to the relation* $=$ or $\cong$, respectively, if S is a substring of A and there exists a substring R of B such that $S = R$ or $S \cong R$, respectively, or S is a substring of B and there exists a substring R of A such that $S = R$ or $S \cong R$, respectively. When not necessary, we will often avoid specifying the relation and will talk only about a common substring. If S is a common substring of A and B, we use notations S^A and S^B to distinguish between the occurrences of S (or $-S$) in A and B; if S occurs more than once in A then S^A refers to an arbitrary but fixed occurrence of S in A and analogous convention applies for the string B. Given two partitions $\mathcal{A} = (A_1, \ldots, A_m)$ and $\mathcal{B} = (B_1, \ldots, B_{m'})$, a common substring of $\mathcal{A}$ and $\mathcal{B}$ is a string S such that S is a common substring of A_i and B_j, for some indices i, j.

1.2 Related work

String comparison is a fundamental problem in computer science with applications in text processing, data compression or computational biology. The problem of sorting by reversals drew a lot of attention in the last years as a useful tool for comparison of genomic sequences [1, 4, 6, 14]. In that application, the letters in the strings represent different genes and the reversal distance measures the similarity of two genomic sequences. A common assumption that a genome contains only one copy of each gene is unwarranted for genomes with multi-gene families such as the human genome [16]. On the other hand, a weaker assumption that a genome contains at most $k = O(1)$

copies of each gene is often warranted (cf. [9]). That is why k-SBR is of interest. In this subsection we will briefly mention the most relevant known results.

Under the assumption that every symbol appears in each input string exactly once, we have the well known problem of permutation sorting by reversals. The problem 1-SBR is solvable in polynomial time for strings with signs [1, 14] but is NP-hard [4] and even MAX-SNP hard [3] for strings without signs; the best known approximation ratio for the unsigned 1-SBR is 1.375 by an algorithm of Berman et al. [2]. A recent result of Chen et al. [5] shows that the signed k-SBR is NP-hard even for $k = 2$ (the unsigned k-SBR is obviously NP-hard for all $k \geq 2$). There are $O(1)$-approximation algorithms for signed 2-SBR and 3-SBR [5, 7, 13]. The best approximation ratio for the general signed SBR is $O(\log n \log^* n)$, using an $O(n \log^* n)$-time[1] algorithm for edit distance problem with block moves [8] (see bellow for further details).

Instead of bounding the number of duplicates, there is another way to restrict the general problem of sorting by reversals with duplicates: bound the size of the alphabet. Unsigned SBR with unary alphabet is trivial; the NP-hardness of unsigned SBR with binary alphabet was proved by Christie and Irving [6].

Closely related is a *minimum common string partition* problem (MCSP). Given a partition $\mathcal{P}$ of a string A and a partition $\mathcal{Q}$ of a string B, we say that the pair $\pi = \langle \mathcal{P}, \mathcal{Q} \rangle$ is a *common partition* of A and B with respect to the relation $\mathsf{Rel} \in \{=, \cong\}$, if there exists a permutation σ on $[m]$ such that for each $i \in [m]$, $(P_i, Q_{\sigma(i)}) \in \mathsf{Rel}$. The minimum common string partition problem is to find a common partition of A, B with the minimum size, denoted by $\mathsf{MCSP}(A, B)$. The restricted version of MCSP, where each letter occurs at most k times in each input string, is denoted by k-MCSP.

Similarly as for SBR, there is a signed and an unsigned variant of the problem. In *unsigned* MCSP, the input consists of two unsigned strings, and the relation $=$ is used; in *signed* MCSP, the input consists of two signed strings and the relation $\cong$ is used. For unsigned strings, we define yet another variant of the problem, *reversed* MCSP (RMCSP), in which the (unsigned) strings are compared by the relation $\cong$.

The signed MCSP problem was introduced by Chen et al. [5] as a tool for dealing with SBR. They observed that for any two related signed strings A and B, $\mathsf{MCSP}(A, B)$ and $\mathsf{SBR}(A, B)$ differ only by a constant multiplicative factor: given a partition $(P_1, \ldots, P_m)$ of A, $(Q_1, \ldots, Q_m)$ of B and the permutation σ on $[m]$ such that $P_i \cong Q_{\sigma(i)}$ for each $i \in [m]$, it is possible to move the block $P_{\sigma^{-1}(1)} \cong Q_1$ to the beginning of A by one reversal and then, if necessary, to reverse it by one more reversal; similarly it is possible to move the block $P_{\sigma^{-1}(2)} \cong Q_2$ to its right position in the first string by at most two reversals without affecting the block $P_{\sigma^{-1}(1)} \cong Q_1$ at the beginning of the string, etc. On the other hand, a reversal "breaks" at most two pairs of consecutive letters in the string and thus, from a sequence of m reversals, we derive a common partition with at most $2m$ breaks. Analogous observation applies for related unsigned strings and the problems RMCSP and SBR.

For $k \geq 2$, k-MCSP is NP-hard, and even APX-hard [13]. Due to the close relation between signed SBR and signed MCSP, the known approximation ratios for signed MCSP are within a constant factor of the approximation ratios for signed SBR: $O(1)$ approximation ratios for 2-MCSP and 3-MCSP [7, 13], $O(\log n \log^* n)$ approximation ratio for the general MCSP [8].

Chrobak et al. [7] analyzed the behavior of a natural greedy heuristic for MCSP: start with the two strings A and B and iteratively, find the longest common substring of A and B that does not overlap previously marked substrings, and mark this substring. They showed that though GREEDY is a 3-approximation algorithm for 2-MCSP, even for 4-MCSP its approximation ratio is $\Omega(\log n)$. For general MCSP, both signed and unsigned, the approximation ratio is between $\Omega(n^{0.43})$ and

[1] $\log^* n = \min\{k \in \mathbb{N} : g(k) \geq n\}$ for a function g defined by $g(1) = 2$, and $g(k) = 2^{g(k-1)}$ for every integer $k > 1$.

$O(n^{0.67})$. It is worth noting that two algorithms described in this paper are simple modifications of GREEDY, yet their approximation ratios for k-MCSP are better, namely $O(k^2)$, in contrast to the $\Omega(\log n)$ of GREEDY for $k \geq 4$.

In the *edit distance* (ED) problem, a set of string operations is given (e.g., DELETE, INSERT or CHANGE a character, SUBSTRING_MOVE or SUBSTRING_REVERSAL) and the task is to find the minimum number of operations needed to convert one string into the other. SBR can be also viewed as an edit distance problem where the only operation is SUBSTRING_REVERSAL and the input strings are related. For any two related strings A and B, $\mathsf{MCSP}(A, B)$ differs by a constant multiplicative factor from the edit distance of A and B with only SUBSTRING_MOVE operations, and the edit distance using only SUBSTRING_MOVE operations differs also by a constant multiplicative factor from the edit distance with operations {INSERT, DELETE a character, SUBSTRING_MOVE} [17].

On the other hand, MCSP can be utilized for approximating the edit distance even for *unrelated* strings. To give an example, consider ED with operations {INSERT, DELETE a character, SUBSTRING_MOVE}: given strings A and B, let $B - A$ denote the multiset of letters that have more occurrences in B than in A (i.e., if x has x_A occurrences in A and x_B occurrences in B then there are $\max\{0, x_B - x_A\}$ copies of x in $B - A$) and analogously for $A - B$. Then $|A - B| + |B - A|$ is a lower bound on the edit distance $\mathsf{ED}(A, B)$. Let A' denote a concatenation of the string A with all letters from $B - A$ (in any order), and similarly, let B' denote a concatenation of B with all letters from $A - B$; we observe that $\mathsf{ED}(A', B') \leq 2\mathsf{ED}(A, B)$. Exploiting the above mentioned relation between ED and MCSP for related strings we obtain $\mathsf{ED}(A, B) = \Theta(1) \cdot (|A - B| + |B - A| + \mathsf{MCSP}(A', B'))$.

For the edit distance problem with operations {INSERT, DELETE a character, SUBSTRING_MOVE}, Cormode and Muthukrishnan [8] described an $O(n \log^* n)$-time $O(\log n \log^* n)$-approximation algorithm which yields, by the relations described above, the $O(\log n \log^* n)$-approximation for SBR mentioned earlier in this subsection.

The edit distance problem with a different set of string operations was studied by Ergun et al. [10]. For several edit distance problems that allow SUBSTRING_DELETION, they describe an $O(1)$ approximation algorithm. This is in contrast to the above mentioned known approximations of edit distance *without* SUBSTRING DELETION where the best approximation ratio is of order $\Omega(\log n \log^* n)$.

The rest of the paper is organized as follows. In Section 2 we describe how to modify GREEDY to get the $O(k^2)$ approximation for (reversed) k-MCSP and thus, for k-SBR. Section 3 explains how to further improve the algorithm to work in linear time.

2 REFINED GREEDY: $O(k^2)$-approximation

In the previous section, we briefly described GREEDY algorithm and we recalled that its approximation ratio for k-MCSP and k-SBR, for any $k \geq 4$, is $\Omega(\log n)$. In this section, we show that a simple modification of GREEDY, called REFINED GREEDY, has an $O(k^2)$ approximation ratio for k-MCSP, which implies also an $O(k^2)$ approximation ratio for k-SBR.

A few more terms are needed. A *duo* is a string of length two. To *cut* a duo $a_i a_{i+1}$ of a block $P = a_j \dots a_k$ of a partition of A, for some $j \leq i < k$, means to replace the block P in the partition by two blocks $P_1 = a_j \dots a_i$ and $P_2 = a_{i+1} \dots a_k$. For a substring $S = a_i \dots a_j$ of $A = a_1 \dots a_n$, if $i > 1$ we say that $a_{i-1} a_i$ is a *(left) boundary* duo of S, and similarly, if $j < n$ $a_j a_{j+1}$ is a *(right) boundary* duo of S.

For unsigned k-MCSP the algorithm is the following:

Algorithm REFINED GREEDY
Input: two related strings A and B
$\mathcal{A} \leftarrow (A)$, $\mathcal{B} \leftarrow (B)$
while there are unmarked blocks in $\mathcal{A}$ and $\mathcal{B}$ **do**
$S \leftarrow$ longest common substring of $\mathcal{A}$, $\mathcal{B}$ that does not overlap previously marked blocks
mark S^A in $\mathcal{A}$ and S^B in $\mathcal{B}$
cut the boundary duos of S^A in $\mathcal{A}$ and the boundary duos of S^B in $\mathcal{B}$
cut in unmarked blocks of $\mathcal{A}$ and $\mathcal{B}$ *all* occurrences of duos $\delta \in \Phi$, where Φ is the set of boundary duos of S^A and S^B
Output: $(\mathcal{A}, \mathcal{B})$

To extend the algorithm for signed k-MCSP and for k-RMCSP, apart from considering common substrings with respect to the other equivalence relation $\cong$, the difference is that in the cutting steps, we cut not only all occurrences of $\delta \in \Phi$ but also all occurrences of $-\delta$.

To give an example, consider an instance of 4-MCSP,

$$A = abxyuvafxyuvddddhefxyuvebxyuvgggg \ ,$$
$$B = abxyuvddddafxyuvhefxyuvggggebxyuv \ .$$

REFINED GREEDY first marks substring $S_1 = xyuvdddd$ (we use overline to denote marking in this example) and cuts all unmarked occurrences of duos from $\Phi = \{fx, dh, bx, da\}$. In the second iteration, REFINED GREEDY looks for the longest unmarked substring in partitions $\mathcal{A} = (ab, xyuvaf, \overline{xyuvdddd}, hef, xyuveb, xyuvgggg)$ and $\mathcal{B} = (ab, \overline{xyuvdddd}, af, xyuvhef, xyuvggggeb, xyuv)$, marks substring $S_2 = xyuvgggg$ and cuts duos from $\Phi = \{ge\}$. In the third iteration, the algorithm looks for the longest unmarked substring in partitions $\mathcal{A} = (ab, xyuvaf, \overline{xyuvdddd}, hef, xyuveb, \overline{xyuvgggg})$ and $\mathcal{B} = (ab, \overline{xyuvdddd}, af, xyuvhef, \overline{xyuvgggg}, eb, xyuv)$, marks substring $S_3 = xyuv$ and cuts duos from $\Phi = \{xa, ch\}$. Eventually, REFINED GREEDY outputs the common partition

$$\begin{aligned}\mathcal{P} = \langle &(ab, xyuv, af, xyuvdddd, hef, xyuv, eb, xyuvgggg), \\ &(ab, xyuvdddd, af, xyuv, hef, xyuvgggg, eb, xyuv)\rangle \ .\end{aligned}$$

The optimal common partition has six blocks:

$$\begin{aligned}\mathcal{P}_{\text{OPT}} = \langle &(abxyuv, afxyuv, dddd, hefxyuv, ebxyuv, gggg), \\ &(abxyuv, dddd, afxyuv, hefxyuv, gggg, ebxyuv)\rangle \ .\end{aligned}$$

Compared to GREEDY, on a very high level, the advantage of REFINED GREEDY is that by introducing additional cuts in each step, the algorithm confines the propagation of "mistakes" that are caused by the the greedy choice of a common substring.

Theorem 2.1 REFINED GREEDY *is* $2k^2$*-approximation algorithm for unsigned and signed* k-MCSP *and* $2(2k-1)^2$*-approximation for* k-RMCSP.

Proof: The output of the algorithm is clearly a common partition. We only have to prove the bound on its quality. For simplicity of the presentation, we prove the claim in detail for the unsigned k-MCSP and then we briefly outline the necessary modifications for signed k-MCSP and for k-RMCSP.

For technical reasons, it will be convenient to extend the notions of a partition and a common partition from strings to sequences of strings. A *partition of the sequence* of strings $\mathcal{A} = (A_1, \ldots, A_l)$ is a sequence of strings $A_{1,1}, \ldots, A_{1,k_1}, A_{2,1}, \ldots,\ A_{2,k_2} \ldots, A_{l,1}, \ldots, A_{l,k_l}$, such that $A_i = A_{i,1} \ldots, A_{i,k_i}$ for $i \in [l]$. For two sequences of strings, the common partition is defined analogously as for two strings.

Observation 2.2 *Let $(\mathcal{Q}, \mathcal{R})$ be a common partition of sequences of strings $\mathcal{A}$ and $\mathcal{B}$, and let δ be any duo that appears in $\mathcal{Q}$ and $\mathcal{R}$. Let $\mathcal{Q}'$ denote the partition of $\mathcal{A}$ that is obtained from $\mathcal{Q}$ by cutting all occurrences of the duo δ, and let $\mathcal{R}'$ denote the partition of $\mathcal{B}$ that is obtained from $\mathcal{R}$ by cutting all occurrences of the duo δ. Then, $(\mathcal{Q}', \mathcal{R}')$ is a common partition of $\mathcal{A}$ and $\mathcal{B}$.*

Proof: Since $\mathcal{Q}$ is a permutation of $\mathcal{R}$, every block P from $\mathcal{Q}$ that contains δ appears also in $\mathcal{R}$, and vice versa. Thus, if we cut all occurrences of δ in $\mathcal{Q}$ and $\mathcal{R}$, the resulting new partitions $\mathcal{Q}'$ and $\mathcal{R}'$ will be again permutations of each other. □

Let $\pi = (\mathcal{P}, \mathcal{Q})$ be a minimum common partition of A and B, m be its size and let Δ be the set of all boundary duos of blocks in $\mathcal{P}$ and in $\mathcal{Q}$. Let T denote the number of steps of the algorithm. We are going to iteratively construct common partitions π_i of A and B that will help us to estimate the size of the common partition found by Refined Greedy. We define π_1 as the common partition derived from π by cutting *all* occurrences of all duos in Δ (the fact that π_1 is a partition follows from Observation 2.2). The breaks in π_1 are called *initial* breaks. For k-MCSP instances, the number of blocks in π_1 is at most k times greater than the number of blocks in π (if a letter a appears as a leftmost letter in a block of π then there are at most k blocks in π_1 with a as a leftmost letter) and therefore the number of initial breaks is at most $km - 1$.

Let S_i denote the substring that Refined Greedy used in iteration i and let Φ_i be the set of boundary duos of S_i^A and S_i^B. For iteration $i \geq 1$ of Refined Greedy, we define π_{i+1} as the common partition derived from π_i by cutting all occurrences of all duos in Φ_i. For ease of reference, we denote the sets $\mathcal{A}$ and $\mathcal{B}$ at the beginning of iteration i by $\mathcal{A}_i$ and $\mathcal{B}_i$, and by s_i the first position of S_i^A in A, by t_i the last position of S_i^A in A, by s_i' the first position of S_i^B in B, and by t_i' the last position of S_i^B in B.

Observation 2.3 *For every iteration i and for every $0 \leq l < |S_i| - 1$: the pair $s_i + l, s_i + l + 1$ is an initial break of A if and only if the pair $s_i' + l, s_i' + l + 1$ is an initial break of B.*

Proof: The observations follow from the definition of π_1 and initial breaks: if one occurrence of a duo is cut in π_1, then all occurrences of this duo are cut. □

Given a break $l, l+1$ of a partition of A, and a substring $S = a_i \ldots a_j$ of A, we say that the substring S *goes over the break* $l, l+1$ if $i \leq l < j$. Observation 2.3 can be informally stated like this: If the block S_i^A goes over one or more initial breaks, then the block S_i^B goes over the same number of initial breaks, and, moreover, the relative positions of the initial breaks in S_i^A and S_i^B are the same.

Let $\mathcal{A}_i' \subseteq \mathcal{A}_i$ and $\mathcal{B}_i' \subseteq \mathcal{B}_i$ denote the subsets of unmarked strings of $\mathcal{A}_i$ and $\mathcal{B}_i$, resp., at the beginning of phase i, and let π_i' denote the restriction of π_i to $\mathcal{A}_i'$ and $\mathcal{B}_i'$. Observation 2.3 implies the following important claim.

Observation 2.4 *For every i, π_i' is a common partition of $\mathcal{A}_i'$ and $\mathcal{B}_i'$.*

Proof: The proof is by induction. For $i = 1$, nothing is marked, $\mathcal{A}_1' = \{A\}$, $\mathcal{B}_1' = \{B\}$, $\pi_1' = \pi_1$ and the claim is obvious. For $i > 1$, Observations 2.2 and 2.3 imply that the blocks from π_i

corresponding to the newly marked block S^A_{i-1} are the same as the blocks from π_i corresponding to the newly marked block S^B_{i-1}. Observing that outside S^A_{i-1} and S^B_{i-1}, cuts of the same duos (i.e., duos from Φ_{i-1}) are used to obtain π'_i from π'_{i-1} and $(\mathcal{A}'_i, \mathcal{B}'_i)$ from $(\mathcal{A}'_{i-1}, \mathcal{B}'_{i-1})$, the proof is completed. □

Lemma 2.5 *For every i,*

- *the block $S_i = a_{s_i} \dots a_{t_i}$ is an entire block in π_i, or*
- *the block S_i goes over an initial break.*

Proof: The lemma follows from Observation 2.4 and from the greedy nature of Refined Greedy: for every common substring S of $\mathcal{A}'_i$ and $\mathcal{B}'_i$ not satisfying any of the conditions in the lemma, there exists another longer common substring S' of $\mathcal{A}'_i$ and $\mathcal{B}'_i$ such that S is a proper substring of S'. □

Lemma 2.5 provides us a tool for bounding the number of breaks in the common partition π_T after the last step of the algorithm. If Refined Greedy chooses for S_i an entire block of π_i, then $\pi_{i+1} = \pi_i$ and there are no new breaks in π_{i+1}, compared to π_i. If Refined Greedy chooses for S_i a substring that goes over an initial break, then we need at most $2k$ cuts (in the string A) to get π_{i+1} from π_i and we charge all of them to the initial break that S^A_i goes over. After the last step of the algorithm, there are at most $(2k+1)(km-1)$ breaks in π_T (recall that $km-1$ is an upper bound on the number of initial breaks).

By construction, the common partition π_T is a refinement of the common partition $\langle \mathcal{A}_T, \mathcal{B}_T \rangle$ computed by Refined Greedy. Thus, the number of blocks in π_T is an upper bound on the number of blocks in $\langle \mathcal{A}_T, \mathcal{B}_T \rangle$ and the approximation ratio of the algorithm is at most $k(2k+1)$. To slightly improve the ratio we observe that if Refined Greedy has chosen in L steps substrings that go over initial breaks (note that $L \leq km-1$), then there are at most $2kL + (km-1-L) \leq 2k^2m-1$ breaks in $\langle \mathcal{A}_T, \mathcal{B}_T \rangle$ which implies the desired approximation ratio.

For signed k-MCSP and k-RMCSP we only need to adjust the proof to reflect the thing that now a substring S from A can be matched with a substring R from B even if $S \neq R$ but $S = -R$. Thus, in Observation 2.2 we cut not only all occurrences of duo δ but also all occurrences of duo $-\delta$. To get the common partition π_1 from π, for each $\delta \in \Delta$ we cut all occurrences of δ as well as all occurrences of $-\delta$; for signed k-MCSP the number of breaks in π_1 increases again at most k times, for k-RMCSP it increases at most $2k-1$ times. In Observation 2.3, we distinguish whether $S^A_i = S^B_i$ or $S^A_i = -S^B_i$. In the later case, we count the relative positions of the initial breaks in S^B_i backwards (i.e., the claim is: $s_i + l, s_i + l + 1$ is an initial break of A if and only if the pair $t'_i - l - 1, t'_i - l$ is an initial break of B); the former case is as before. For signed k-MCSP, the number of duos cut in $\mathcal{A}$ in one iteration is at most $2k$, for k-RMCSP it is at most $2(2k-1)$. □

Considering the relation between signed MCSP and signed SBR, and between RMCSP and unsigned SBR, we get the following theorem.

Theorem 2.6 *There exists a polynomial time $4k^2$-approximation algorithm for signed k-SBR, and $8(2k-1)^2$-approximation algorithm for unsigned k-SBR.*

Concerning the running time of Refined Greedy, we just note that a naive straightforward implementation of the algorithm runs in time $O(n^3)$.

3 Educated Greedy

In the previous analysis we never used the fact that S_i was the *longest* common substring; we only used that S_i was never a proper substring of another common block of unmarked letters (proof of Lemma 2.5). Based on this observation, here we present two implementations of a faster algorithm Educated Greedy. As in the case of Refined Greedy, we describe them in detail for unsigned k-MCSP; the necessary modifications for signed k-MCSP and k-RMCSP are the same as before. We start by giving a naive implementation of Educated Greedy running in time $O(k^2 n)$. Then, in the next subsection we describe a linear time implementation of Educated Greedy; crucial components of this implementation are linear-time algorithms for construction of suffix trees [11] and for (a special case of) disjoint set union problem [12]. The two algorithms for MCSP have the same structure described bellow and the difference is in their choice of a common substring S_i in each iteration (function Find–Unextendable):

Algorithm Educated Greedy
Input: two related strings $A = a_1 \dots a_n$ and $B = b_1 \dots b_n$
  $\mathcal{A} \leftarrow (A)$, $\mathcal{B} \leftarrow (B)$
  $i = 1$
  while $i \leq n$ **do**
    $S \leftarrow$ Find–Unextendable(a_i)
    mark S^A in $\mathcal{A}$ and S^B in $\mathcal{B}$
    cut the boundary duos of S^A in $\mathcal{A}$ and the boundary duos of S^B in $\mathcal{B}$
    cut in $\mathcal{A}$ and $\mathcal{B}$ *all* unmarked occurrences of duos $\delta \in \Phi$, where Φ is
      the set of boundary duos of S^A and S^B
    $i \leftarrow \min\{j : j \geq i \text{ and } j \text{ unmarked}\}$
Output: $(\mathcal{A}, \mathcal{B})$

Given an nonempty string C, the function Find–Unextendable(C) finds an unmarked common substring S of $\mathcal{A}$ and $\mathcal{B}$ such that C is a substring of S and S is not a proper substring of any unmarked common substring of $\mathcal{A}$ and $\mathcal{B}$.

3.1 Naive implementation

A naive implementation of the function Find–Unextendable(C) is done as follows: check all k^2 possibilities for the positions of the desired substrings S^A and S^B; this requires time $O(k^2|S|)$. Since all other operations over all iterations require time $O(n)$, the total running time of the algorithm is $\sum_i O(k^2|S_i|) = O(k^2 n)$.

Concerning the correctness of the algorithm, we already observed earlier that the proof of Lemma 2.5 is the only place in the proof of Theorem 2.1 that refers to the choice of the common substrings S_i used by Refined Greedy. However, as mentioned above, the proof only needs the fact that S_i cannot be extended on either side. Thus, Lemma 2.5 holds also for the choices of Educated Greedy and the $O(k^2)$ approximation ratio follows by the same reasoning as for Refined Greedy. We get he following theorem.

Theorem 3.1 *The naive implementation of* Educated Greedy *runs in time $O(k^2 n)$ and computes an $O(k^2)$-approximation for unsigned and signed k-*MCSP*, k-*RMCSP *and k-*SBR.

3.2 Fast implementation

The most time consuming part of the naive implementation is the search for the unextendable common substring of $\mathcal{A}$, $\mathcal{B}$ containing a given letter; in the worst case it takes time $\Omega(k^2|S|)$ in a single iteration to find the substring S. With an additional data structure we show how to implement this step in (amortized) time $O(|S|)$, yielding a linear time algorithm. From now on, let X denote the concatenation of A, \$ and B, where \$ is a character that does not appear in A and B, that is, $X = A\$B$; observe that every suffix of B is also a suffix of X. We assume that the characters of A and B are represented by integers of size $O(\log n)$ bits.

The idea is simple. We observe that given the strings A and B, a common substring Z of A and B, and a suffix tree of $X = A\$B$, one can easily find a common substring Z' of A and B such that Z is a prefix of Z' and Z' is not a proper prefix of any common substring of A and B (first, descent from the root of the suffix tree along edges with labels corresponding to Z, and then descent along any edges such that their subtree has at least one leave corresponding to a suffix of X starting in the A part of X and at least one leave corresponding to a suffix of X starting in the B part of X). To find an unextendable common substring of A and B containing a letter a (i.e., containing a substring $Z = a$), we first find a common substring Z' that starts with a and that is not a prefix of any longer common substring. Then we find a common substring S' of $-A$ and $-B$ that has $-Z'$ as a prefix and that is not a prefix of any longer common substring of $-A$ and $-B$; $-S'$ is the common substring we are looking for (cf. Figure 1). The rest of the subsection deals with the technical problems arising from the fact that after the first iteration of EDUCATED GREEDY some duos in A and B are broken and some letters are marked and cannot be used for S.

We are going to describe a function EXTEND$(\mathcal{A}, \mathcal{B}, Z)$ that finds in partitions $\mathcal{A}$ and $\mathcal{B}$ an unmarked common substring Z' such that Z is a prefix of Z' and Z' is not a prefix of any longer unmarked common substring. Fast implementation of EXTEND is a key component of a fast implementation of FIND–UNEXTENDABLE:

function FIND–UNEXTENDABLE(Z)
  $Z' \leftarrow$ EXTEND$(\mathcal{A}, \mathcal{B}, Z)$
  $S' \leftarrow$ EXTEND$(-\mathcal{A}, -\mathcal{B}, -Z')$
return$(-S')$

For a string Z, we denote by $Z[i \rightarrow]$ the suffix of Z starting at position i. A *rank* of a suffix T of Z, denoted RANK$_Z(T)$, is the number (rank) of the suffix T in the lexicographical order of all suffixes of Z. The ranks of all suffixes of Z can be computed in linear time as a by-product of the construction of the suffix tree of Z [11]. Observe that the ranks of the suffixes corresponding to leaves of any node of the suffix tree form an interval.

Throughout the runtime of EDUCATED GREEDY we maintain a tree $\mathcal{T}$; at the beginning of the algorithm the tree $\mathcal{T}$ is the suffix tree of the string $X = A\$B$ and in later iterations, $\mathcal{T}$ is a subtree of the original suffix tree of X. Recall that in a suffix tree of X every edge has a label corresponding to a substring of X. For a node u of $\mathcal{T}$ we denote be L_u the concatenation of the labels of all edges

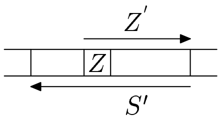

Figure 1: Construction of unextendable common substring S by function FIND–UNEXTENDABLE.

on the path from root to u. For each node u of the original tree $\mathcal{T}$ we also compute an interval (i_u, j_u) of ranks of suffixes of X that correspond to leaves of the subtree of $\mathcal{T}$ rooted in the node u.

In the following description of EXTEND, we work with the tree $\mathcal{T}$ of X; whenever we refer to (a label of) an edge or a vertex, we refer to (a label of) an edge or a vertex in the tree $\mathcal{T}$. For a string Z we denote by z_{-1} the last letter in Z.

function EXTEND$(\mathcal{A}, \mathcal{B}, Z)$
(1) $(u, v) \leftarrow$ an edge such that L_u is a proper prefix of Z and Z is a prefix of L_v
  if $Z = L_v$ **then**
    foreach $w \in \text{children}(v)$ **do**
      $y_1 \dots y_r \leftarrow$ a label of (v, w)
      if $z_{-1}y_1$ is not a broken duo **and** EXISTS(A, w) **and** EXISTS(B, w) **then**
        $Z \leftarrow Zy_1$
        goto (1)
(8)   **else** remove the edge (v, w) from $\mathcal{T}$
    return(Z)
  else
    $y_1 \dots y_r \leftarrow$ a label of (u, v)
    $i \leftarrow$ an index such that $L_u y_1 \dots y_i = Z$
    while $i < r$ **do**
      if $y_i y_{i+1}$ is not a broken duo **then**
        $Z \leftarrow Zy_{i+1}$
        $i \leftarrow i + 1$
      else return(Z)
    goto (1)

The last part of the algorithm to be filled in is the description of the function EXISTS(Y, v) that answers the question whether in the subtree of v exists a leaf corresponding to a suffix of X starting in the Y part of X (for $Y = A$ or $Y = B$) such that the first letter of the suffix is unmarked. We employ an algorithm for (a special case of) the disjoint set union problem [12] for this task. In the version of the problem that is relevant to our setting, the sets correspond to disjoint intervals of integers such that their union is the interval $1, \dots, m$. The task is to perform an intermixed sequence of two operations that access and modify the set of intervals:

- FIND(h) - returns the largest element from the interval containing h;
- UNION(h) - creates a new interval that is the union of the interval containing h and the consecutive interval (i.e., the interval containing FIND$(h)+1$) and destroys the two old intervals; if there is no consecutive interval, the operation does nothing.

Gabow and Tarjan [12] describe an algorithm that executes a sequence of m UNION and FIND operations in time $O(m)$; the amortized time for a single operation is $O(1)$.

In our application we use $m = 2n + 1$ and we work with two systems of intervals: $\mathcal{S}_A$ for the string A and $\mathcal{S}_B$ for the string B. In $\mathcal{S}_A$, resp. $\mathcal{S}_B$, an interval $\{l, l+1, \dots, r\}$ represents the fact that the first letter of a suffix with rank r is not marked and that there is no suffix of X starting in the A part, resp. B part, with rank at least l but less than r. The initialization of $\mathcal{S}_A$ and $\mathcal{S}_B$ is done as follows. Let χ_A and χ_B be two binary arrays of length m such that $\chi_A[i] = 1$ if the suffix of X with rank i starts in the A part of X and $\chi_B[i] = 1$ if the suffix of X with rank i starts in the B part of X. We set $\mathcal{S}_A = \{\{l, l+1, \dots, r\} \mid \chi_A[r] = 1, \chi_A[i] = 0 \text{ for } i = l, \dots, r-1, \text{ and either } l =$

1 or $\chi_A[l-1]=1\}$; the system $\mathcal{S}_B$ is defined analogously. Such an initialization is computed in linear time as a byproduct of the construction of the suffix tree of X. With the structures $\mathcal{S}_A$ and $\mathcal{S}_B$ we implement the function EXISTS as follows ($Y = A, B$):

function EXISTS(Y, u)
    $(i_u, j_u) \leftarrow$ the minimal rank and the maximal rank of the suffixes in the subtree of u
    if $\mathcal{S}_Y$.FIND$(i_u) > j_u$ **then return** (False)
    else return (True)

To keep the structures $\mathcal{S}_A$ and $\mathcal{S}_B$ up-to-date we perform an operation UNION(RANK($X[i \rightarrow]$)) whenever we mark a letter a_i in A, we perform an operation UNION(RANK($X[n+1+i \rightarrow]$)) whenever we mark a letter b_i in B. This finishes the description of data structures for the function EXTEND$(\mathcal{A}, \mathcal{B}, Z)$; for EXTEND$(-\mathcal{A}, -\mathcal{B}, -Z')$ we maintain analogous data structures for the strings $-A, -B$ and $-X$.

Theorem 3.2 *The fast implementation of* EDUCATED GREEDY *runs in time $O(n)$ and computes an $O(k^2)$-approximation for unsigned and signed k-*MCSP*, k-*RMCSP *and k-*SBR.

Proof: With the help of the data structure for the disjoint set union problem, every iteration requires amortized time $O(|S_i|)$ *plus* the time spent on removing edges from $\mathcal{T}$ (line (8)). Since every edge is removed at most once and the number of edges in the original suffix tree of X is $O(n)$, the total running time is also $O(n)$. The correctness of the fast implementation follows from correctness of the naive implementation. □

4 Conclusion

We presented an $O(k^2)$-approximation algorithm for k-MCSP and k-SBR running in time $O(n)$. A challenging open problem is whether there exists a (simple and fast) $O(k)$-approximation algorithm for k-MCSP and k-SBR? Another interesting question concerns the best possible approximation ratio for the general MCSP and SBR: is it possible to get below the $O(\log n \log^* n)$ upper bound in polynomial time?

References

[1] A. Bergeron, J. Mixtacki, and J. Stoye. Reversal distance without hurdles and fortresses. In *Proceedings of 15th Annual Combinatorial Pattern Matching Symposium (CPM)*, volume 3109 of *Lecture Notes in Computer Science*, pages 388–399. Springer-Verlag, 2004.

[2] P. Berman, S. Hannenhalli, and M. Karpinski. 1.375-approximation algorithm for sorting by reversals. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, volume 2461 of *Lecture Notes in Computer Science*, pages 200–210, 2002.

[3] P. Berman and M. Karpinski. On some tighter inapproximability results. In *Proceedings of the of 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 200–209, 1999.

[4] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the First International Conference on Computational Molecular Biology*, pages 75–83. ACM Press, 1997.

[5] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, and T. Jiang. Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(4):302–315, 2005.

[6] D. A. Christie and R. W. Irving. Sorting strings by reversals and by transpositions. *SIAM Journal on Discrete Mathematics*, 14(2):193–206, 2001.

[7] M. Chrobak, P. Kolman, and J. Sgall. The greedy algorithm for the minimum common string partition problem. In *Proceedings of the 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pages 84–95, 2004. To appear in ACM Transactions on Algorithms.

[8] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *Proceedings of the 13th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 667–676, 2002.

[9] N. El-Mabrouk. Reconstructing an ancestral genome using minimum segments duplications and reversals. *Journal of Computer and System Sciences*, 65(3):442–464, 2002.

[10] F. Ergun, S. Muthukrishnan, and S. C. Sahinalp. Comparing sequences with segment rearrangements. In *Proceedings of the 23rd Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2914 of *Lecture Notes in Computer Science*, pages 183–194, 2003.

[11] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

[12] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 246–251, 1983.

[13] A. Goldstein, P. Kolman, and J. Zheng. Minimum Common String Partition Problem: Hardness and Approximations. In *Proceedings of the 15th International Symposium on Algorithms and Computation (ISAAC)*, volume 3341 of *Lecture Notes in Computer Science*, pages 484–495, 2004.

[14] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, 1999.

[15] P. Kolman. Approximating reversal distance for strings with bounded number of duplicates. In *Proceedings of the 30th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 3618 of *Lecture Notes in Computer Science*, pages 580–590, 2005.

[16] D. Sankoff and N. El-Mabrouk. Genome rearrangement. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 135–155. The MIT Press, 2002.

[17] D. Shapira and J. A. Storer. Edit distance with move operations. In *13th Symposium on Combinatorial Pattern Matching (CPM)*, volume 2373 of *Lecture Notes in Computer Science*, pages 85–98, 2002.