# Algorithms for Transforming Strings by Reversals*

**Gustavo da S. Teixeira**[1], **Carla N. Lintzmayer**[1]

[1]Center for Mathematics, Computing and Cognition – Federal University of ABC
Santo André, São Paulo, Brazil

teixeira.gustavo@aluno.ufabc.edu.br, carla.negri@ufabc.edu.br

***Abstract.*** *A reversal is an operation that cuts a segment of a string and reverses it. The problem of Transforming Strings by Reversals (TSbR) consists of, given two strings, finding the minimum number of reversals that transform one string into the other. TSbR is NP-hard and there are not many algorithmic results for it. In this work, we propose eight practical algorithms for TSbR and compare them, experimentally.*

## 1. Introduction

To estimate the evolutionary distance between two organisms, due to the principle of parsimony, it is common to use the minimum number of large-scale mutations that affect the genome of one organism and transform it into the genome of another. One such possible mutation is a reversal, which is a well-studied genome rearrangement operation.

We can represent a genome as a sequence of integers. When genes are not repeated, we represent it as a permutation. Otherwise, the representation is given by a string. Also, when the orientation of the genes is known, each element has a "$+$" or "$-$" sign, and the permutation or string is called "signed".

A *reversal* is a rearrangement operation that reverses the order of a segment in a string. Sorting Permutations by Reversals (SbR) and Transforming Strings by Reversals (TSbR) consist of finding the minimum number of reversals that sort a given permutation and that transform one given string into another, respectively.

Signed SbR is polynomial [Hannenhalli and Pevzner 1999], while the unsigned SbR is NP-hard with a 1.375-approximation [Berman et al. 2002] as best result. As for TSbR, it is NP-hard [Chen et al. 2005], and there exists a $\Theta(k)$-approximation [Kolman and Waleń 2007] for when each symbol occurs at most $k$ times derived from a $\Theta(k)$-approximation for the Minimum Common String Partition problem (MCSP) [Chen et al. 2005]. A simple greedy heuristic for the MCSP when each symbol occurs at most $k$ times has approximation ratio $\Omega(n^{0.43})$ and $O(n^{0.67})$, for any $k$ and strings of size $n$ [Chrobak et al. 2004]. [Siqueira et al. 2020] compared the $\Theta(k)$-approximation with some heuristics of their own to deal with TSbR only in the particular case of $k = 2$. The approximation algorithm obtained the worst results in their tests.

To the best of our knowledge, apart from these algorithmic results, no other are known. Note, specially, the lack of existence of algorithms that directly handle TSbR. The approximations algorithms (which are primarily for MCSP), despite having linear time, are not so simple to implement. Our goal, therefore, is to find good practical algorithms for TSbR. We present eight algorithms, and compare all of them through practical experiments, considering sets of strings in which each symbol can occur more than twice.

*The authors thank the São Paulo Research Foundation, FAPESP, proc. 2019/13312-7.

The rest of this text is organized as follows: Section 2 presents important definitions and notations; Section 3 describes each of the proposed and implemented algorithms; Section 4 presents the experimental results and a comparative analysis of the algorithms; Section 5 concludes the work and gives suggestions for future work.

## 2. Definitions and notation

A *permutation* is a tuple $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, of length $|\pi| = n$, with $\pi_i \in \{1, 2, \ldots, n\}$ and $|\pi_i| \neq |\pi_j| \iff i \neq j$. In a *signed permutation*, each element has an associated "+" or "−" sign. In an *unsigned permutation*, signals are omitted. The *identity permutation* is given by $\iota = (1, 2, 3, \ldots, n)$ and is the goal of the sorting problems.

A *string* $S = s_1 s_2 \ldots s_n$ is a sequence of (possibly repeated) elements of an *alphabet* $\Sigma = \{0, 1, 2, \ldots, k-1\}$. In a *signed string*, each element has an associated "+" or "−" sign. In an *unsigned string*, signals are omitted. Note that, by definition, a permutation is a string in which the elements do not repeat. The *length* of a string $S$, denoted by $|S|$, is the number of positions for symbols it has.

The *longest common prefix* (resp. *suffix*) between two strings $S$ and $T$ is the maximal substring $L = s_i s_{i+1} \ldots s_{f-1} s_f$ such that $i = 1$ (resp. $f = n$), and, for all $i \leq j \leq f$, $s_j = t_j$, and its length is denoted by $lcp(S, T)$ (resp. $lcs(S, T)$).

We denote by $f(S, x)$ the number of occurrences of a symbol $x$ in the string $S$. Also, $f(S, S')$ denotes the number of occurrences of substring $S'$ in $S$.

Two strings $S$ and $T$ are *balanced* if $|S| = |T|$, they are described over the same alphabet $\Sigma$, and $f(S, x) = f(T, x)$ for all $x \in \Sigma$. We only consider balanced strings.

A *reversal* $\rho(i, j)$, with $1 \leq i < j \leq n$, is an operation that transforms a string when applied to a string $S = s_1 s_2 \ldots s_n$ into the string $S \cdot \rho(i, j) = s_1 \ldots s_{i-1}$ $\underline{s_j\ s_{j-1} \ldots s_{i+1}\ s_i}$ $s_{j+1} \ldots s_n$. If the string is signed, the reversal also changes the sign of each element of the reversed segment.

The *reverse string* of $S$ is the string $S^R = S \cdot \rho(1, n)$. Two strings $S = s_1 \ldots s_n$ and $T = t_1 \ldots t_n$ are *identical*, denoted by $S = T$, if $s_i = t_i$ for all $i \in \{1, \ldots, n\}$. They are *congruent*, denoted by $S \cong T$, whether $S = T$ or $S = T^R$. We say that $R$ is a *common substring, with respect to the equivalence relation* $(=)$, if $R$ is a substring of $S$ and is a substring of $T$. Also, $R$ is a *common substring, with respect to the congruence relation* $(\cong)$, if $R$ is a substring of $S$ and there is a substring $R'$ of $T$, or if $R$ is a substring of $T$ and there is a substring $R'$ of $S$, such that $R \cong R'$.

The problem of *Transforming Strings by Reversals (TSbR)* consists of, given two strings, finding the minimum number of reversals that transform one string into the other. This value is called the *(reversal) distance* and it is denoted by $d_\rho(S, T)$. For example, for $S = 01021323$ and $T = 02321310$, we have $d_\rho(S, T) = 3$, as the sequence of strings $0\underline{1021323}$, $02\underline{312013}$, $023\underline{12}310$ show, where each reversed segment is underlined.

The concept of *reversal breakpoints*, widely used in permutations, was adapted for strings considering their duos, which are substrings of length 2. The idea is to count duos that appear in different amounts in the strings. Let $\delta(x) = x$, if $x > 0$, and $\delta(x) = 0$, otherwise. Consider special elements $w < 0$ and $z > |\Sigma| - 1$, added to the beginning and end, respectively, of both strings. The

*number of reversal breakpoints* between two strings $S$ and $T$, denoted by $b_\rho(S,T)$, is given by $b_\rho(S,T) = \sum_{w \leq x < y \leq z} \delta\left(f(S,xy) + f(S,yx) - f(T,xy) - f(T,yx)\right) + \sum_{x \in \Sigma} \delta\left(f(S,xx) - f(T,xx)\right)$.

For example, for $S = 01021323$ and $T = 02321310$, we have $b_\rho(S,T) = 2$, since $S$ has one occurrence of duo $01/10$ and one of duo $3z/z3$ more than $T$, and $T$ has one occurrence of duo $13/31$ and one of duo $0z/z0$ more than $S$.

A *partition* of a string $S = s_1 \dots s_n$ is a sequence $\mathcal{P} = (P_1, P_2, \dots, P_m)$ of strings whose concatenation $P_1 P_2 \dots P_m$ results in $S$. Each $P_i$, with $1 \leq i \leq m$, is called *part* of $\mathcal{P}$, and the number of parts in a partition is its *size*, denoted by $|\mathcal{P}|$.

Given a partition $\mathcal{P} = (P_1, \dots, P_m)$ of a string $S$ and a partition $\mathcal{Q} = (Q_1, \dots, Q_m)$ of a string $T$, we say that the pair $\pi = (\mathcal{P}, \mathcal{Q})$ is a *common partition* of $S$ and $T$, with respect to the relation $\texttt{Rel} \in \{=, \cong\}$, if there is a permutation $\sigma$ of $(1, \dots, m)$ such that $(P_i, Q_{\sigma(i)}) \in \texttt{Rel}$, for each $i \in \{1, \dots, m\}$. For example, for $S = 30132210$ and $T = 30231021$, one possible common partition with respect to the relation $\cong$ is $\pi = (\mathcal{P}, \mathcal{Q})$, with $\mathcal{P} = (3, 0132, 21, 0)$ and $\mathcal{Q} = (3, 0, 2310, 21)$.

The Minimum Common String Partition Problem (MCSP) consists of, given two unsigned strings $S$ and $T$, finding a common partition of minimum size between $S$ and $T$ with respect to the relation $=$. The Reverse MCSP (RMCSP) has the same objective, but with respect to the relation $\cong$.

## 3. Implemented algorithms

In the following subsections, we describe the eight algorithms tested in this work.

### 3.1. Improved Selection Sort

The simplest algorithm for transforming a string into another compatible string is similar to the *Selection Sort* algorithm. Given two strings $S$ and $T$, of length $n$, the idea is to go through the elements of $S$ from the first position, $i = 1$, and, if $s_i \neq t_i$, find the minimum index $j > i$ such that $sj = t_i$ and apply $\rho(i,\, j)$ to $S$.

One can see that, in the worst case, $n-1$ reversals will be necessary to perform the transformation. Note that this process is equivalent to saying that, each reversal applied will increase by at least one the length of the longest common prefix between $S$ and $T$. The algorithm stops when $lcp(S,T) = n$, which implies $S = T$.

Note that we can have, in a given iteration, several reversals that increase the value of $lcp(S,T)$, but the algorithm applies the first one it finds. Instead, the algorithm could choose, among all the reversals applicable to $S$, the one that most increases the value of $lcp(S,T)$. We can also look for reversals that instead of increasing the value of $lcp(S,T)$, most increase the value of $lcs(S,T)$, since increasing the common suffix is also a way to obtain identical strings. In addition, we do not need to apply reversals only to $S$ during transformation, but we can also apply reversals to $T$.

Thus, the final version of the algorithm will choose, at each iteration, the reversal that most increases the value of $lcp(S,T) + lcs(S,T)$, among all the reversals applicable to $S$ or $T$, and apply it in the respective string. This final version of the algorithm will be called SELECTION.

For example, for $S = 30132210$ and $T = 30231021$, the SELECTION algorithm would apply the following sequence of 3 reversals, where the reversed segments are underlined: 30$\underline{132}$210, 30231$\underline{210}$, 30231$\underline{201}$.

Each iteration takes time $O(n^2)$ to find and apply the reversal that most increases the value of $lcp(S,T) + lcs(S,T)$. In the worst case, $n-1$ iterations will be required. Thus, SELECTION takes time $O(n^3)$.

### 3.2. Removing breakpoints

Note that each reversal applied to a string can remove at most 2 breakpoints [Chen et al. 2005]. Since $S = T$ implies $b_\rho(S,T) = 0$, one strategy is to successively eliminate breakpoints. However, one obstacle to this strategy is the fact that $b_\rho(S,T) = 0$ does not imply $S = T$. Thus, if we eliminate breakpoints until there are no more breakpoints between the strings, we will not necessarily have transformed one string into the other. Furthermore, in permutations, we know that there is an optimal sequence of reversals that never increases the number of reversal breakpoints [Hannenhalli and Pevzner 1996], but this result is not valid for strings, as one can see from $S = 01021323$ and $T = 01321023$, where $b_\rho(S,T) = 0$ and $d_\rho(S,T) = 2$.

Thus, the strategy of the second proposed algorithm consists of seeking, in $S$ and $T$, the reversal that most eliminates breakpoints between the strings and, if there is no reversal that eliminates breakpoints, apply the reversal that most increases the value of $lcp(S,T) + lcs(S,T)$. This second algorithm will be called BREAKS.

For each iteration, finding the reversal that most eliminates breakpoints takes time $O(n^2)$ and, if there is no reversal in this condition, looking for the reversal that most increases the value of $lcp(S,T) + lcs(S,T)$ takes time $O(n^2)$. In the worst case, the algorithm will apply $n$ reversals, thus the total time is $O(n^3)$.

### 3.3. Selection Sort from RMCSP

The third algorithm proposed here consists of, initially, finding a common partition through the GREEDY algorithm, proposed by Chrobak *et al.* [Chrobak et al. 2004], for the RMCSP. The idea is, from the common partition $\pi = (\mathcal{P}, \mathcal{Q})$ found, sort the parts using, again, an algorithm similar to the Selection Sort.

Starting with $i = 1$, whenever $P_i \neq Q_i$, we have two possibilities, for some $j > i$. Either $P_j = Q_i^R$, in which case one reversal over $S$ places $P_j$ in its correct position, or $P_j = Q_i$, in which case two reversals over $S$ place $P_j$ in its final position. In the end, we will have $\mathcal{P} = \mathcal{Q}$ and, consequently, $S = T$. This algorithm will be called MCSP1.

For example, for $S = 30132210$ and $T = 30231021$, the common partition given by GREEDY is $\pi = (\mathcal{P}, \mathcal{Q})$, where $\mathcal{P} = (3, 0132, 21, 0)$ and $\mathcal{Q} = (3, 0, 2310, 21)$. The first reversal applied places part 0 of $\mathcal{P}$ in the second position, that is, 3$\underline{0132210}$. Then, we have $\mathcal{P}' = (3, 0, 12, 2310)$. Now, we must place the part 2310 of $\mathcal{P}'$ in the third position, which requires 2 reversals. That is, 30$\underline{122310}$ and 30$\underline{0132}$21. Now, we have $\mathcal{P}'' = (3, 0, 2310, 21) = \mathcal{Q}$, so the algorithms stops after 3 reversals.

The GREEDY algorithm takes time $O(n^3)$ to find a common partition between strings. From the common partition, the algorithm that places the parts in the correct positions takes time $O(n^2)$. Thus, MCSP1 has total time $O(n^3)$.

### 3.4. Signed permutations from RMCSP

The fourth algorithm again starts with a common partition $\pi = (\mathcal{P}, \mathcal{Q})$ returned by the GREEDY algorithm, and then transforms each one into a signed permutation. Each part $Q_i$ from $\mathcal{Q}$ is renamed to $+i$ (which will result in $\mathcal{Q}$ being equal to the identity permutation), and one part $P_j$ in $\mathcal{P}$ is renamed to $+i$ if $P_j = Q_i$, or to $-i$ if $P_j = Q_i^R$.

Consider the same example of strings $S = 30132210$ and $T = 30231021$ as before and their common partition $\pi = (\mathcal{P}, \mathcal{Q})$ given by GREEDY. After converting $\pi$ into a pair of signed permutations, we will have the identity permutation $\iota = (+1, +2, +3, +4)$ for $\mathcal{Q}$ and permutation $\pi = (+1, -3, +4, +2)$ for $\mathcal{P}$.

After this process, we will have an instance of the signed SbR problem, which we can solve optimally [Hannenhalli and Pevzner 1999]. We then return such distance. This algorithm will be called MCSP2.

Renaming the common partition as a signed permutation takes time $O(n^2)$. Calculating the reversal distance for signed permutations, without the optimal sequence of reversals itself, takes time $O(n)$. Therefore, MCSP2 also has total time $O(n^3)$ due to the GREEDY algorithm.

### 3.5. Mapping strings into permutations

Let $S = s_1\, s_2 \ldots s_n$ be a string described over an alphabet $\Sigma$, with $|\Sigma| = k$, such that $f(S, i) = a_i$ for all $i \in \Sigma$. A *mapping* $m$ of $S$ can be represented by a set of $k$ sequences $m_i$, with $0 \leq i < k$, each one being a permutation of the set $[a_i] = \{1, 2, \ldots, a_i\}$. The element at position $j$ of a sequence $m_i$, denoted by $m_{(i,j)}$, represents the label of the $j$-th occurrence of the symbol $i$ in $S$. The permutation resulting from applying the labels of $m$ to $S$ will be denoted by $S^m$. A *trivial mapping* $m$ of $S$ is a mapping where each element $m_{(i,j)} = j$, which is equivalent to saying that each sequence $m_i$ increasingly ordered.

For example, one possible mapping $m$ for $S = 12323013$ could be such that $m_0 = (1)$, $m_1 = (2, 1)$, $m_2 = (1, 2)$, $m_3 = (2, 3, 1)$. The resulting permutation will be $S^m = (1_2, 2_1, 3_2, 2_2, 3_3, 0_1, 1_1, 3_1) = (3, 4, 7, 5, 8, 1, 2, 6)$. For this same string $S$, the trivial mapping $z$ would be such that $z_0 = (1)$, $z_1 = (1, 2)$, $z_2 = (1, 2)$, $z_3 = (1, 2, 3)$, and the permutation resulting from it would be $S^z = (1_1, 2_1, 3_1, 2_2, 3_2, 0_1, 1_2, 3_3) = (2, 4, 6, 5, 7, 1, 3, 8)$.

For two strings $S$ and $T$, if the reversal distance between the permutations $S^p$ and $T^q$ is $d(S^p, T^q)$, then we know that $d(S, T) \leq d(S^p, T^q)$, for any mappings $p$ and $q$. Also, there is at least one pair of mappings $p$ and $q$ such that $d(S, T) = d(S^p, T^q)$.

Finally, the algorithm consists of, given two strings $S$ and $T$ and an integer $N$, generating $N$ random mappings for $S$ and a trivial mapping $t$ for $T$, and calculating, using the greedy 2-approximation for SbR, which we call KS95 [Kececioglu and Sankoff 1995], the number of reversals that transforms each of the permutations generated from $S$ into the permutation $T^t$. From the $N$ solutions, the algorithm returns the one with the minimum cost. This algorithm will be called MAPS1 and it was given by [Siqueira et al. 2020].

Generating each of the random mappings takes time $O(n)$ and, for each mapping, calculating a solution with KS95 takes time $O(n^2)$.

### 3.6. Mappings and breakpoints

Another proposed algorithm is a variation of the MAPS1 algorithm that, instead of calculating a number of reversals for each mapping using KS95, calculates the number of reversal breakpoints of the permutations generated by the mappings. At the end, it returns the number of reversals calculated by KS95 for the permutation that has the minimum number of reversal breakpoints.

This idea is based on the fact that fewer breakpoints, in general, imply shorter distances, and that applying KS95 is more time consuming than calculating the number of breakpoints. This algorithm will be called MAPS2.

Generating each of the random mappings takes time $O(n)$ and, for each mapping, calculating the number of breakpoints takes time $O(n)$.

### 3.7. BRKGA implementation for TSbR

The next algorithm consists of the implementation of the metaheuristic Biased Random-Key Genetic Algorithm (BRKGA) [Gonçalves and Resende 2010] for TSbR.

The proposed implementation receives as input a pair of strings $S$ and $T$, the population size $p$, the number $g$ of generations, the number $q_e$ of individuals in the elite population, the number $q_t$ of mutants in the population, and a real number $\rho_e \in (\frac{1}{2}, 1)$, representing the probability that a descendant will inherit parts of his parent belonging to the elite in the *crossover* process.

**Decoder and Initial Population.** Initially, the algorithm will generate $p$ trivial mappings for the string $S$ and a single trivial mapping for $T$. For each element $m_{(i,j)}$ of each sequence $m_i$ of each trivial mapping $m$ generated for $S$, a random key will be assigned, represented by a real number in the interval $[0, 1]$. The decoder will work by sorting the elements of each of the $m_i$ sequences of each mapping $m$ in non-decreasing order of the random keys associated with each element. This process is analogous to shuffling the elements of each of the trivial mapping sequences, and we can see that the mapping resulting from this process remains a valid mapping for $S$. Each resulting mapping will represent an individual in the initial population.

**Fitness Function.** For a decoded mapping $m$ of $S$, a solution is calculated using KS95 with the permutation $S^m$ and the permutation generated by the trivial mapping of $T$ as input. Thus, the lower the value of the solution, the greater its fitness value.

**Elite.** The elite of the current generation is formed by the $q_e$ individuals with the highest fitness values. They are copied to the next generation.

**Mutation.** We randomly generate $q_t$ individuals (mutants) for the next generation.

**Crossovers.** The number $q_c$ of individuals that must be generated by crossover is $p - q_e - q_t$. To generate each one, we randomly select an elite parent and a non-elite parent and, to form each sequence $m_i$ of the descendant, we draw a real number $r$ in the $[0, 1]$ range. If $r \leq \rho_e$, the sequence inherited by the descendant will be the sequence $m_i$ of the elite parent. Otherwise, the descendant will inherit sequence $m_i$ from the non-elite parent. Note that we chose to inherit each sequence $m_i$, and not each element $m_{(i,j)}$. The reason for this is the fact that the decoder, as proposed, rearranges the elements $m_{(i,j)}$ of each sequence $m_i$ of individuals. Thus, if we chose to draw a number for each element $m_{(i,j)}$,

the decoder could transform one of the sequences $m_i$ of the descendant into a sequence that would not represent the inheritance of characteristics from either parent.

After creating $g$ generations of $p$ individuals, the algorithm will return as the solution for the TSbR the lowest value (or highest fitness value) of the solutions associated with individuals of generation $g$. This algorithm will be called BRKGA1. Note that creating one individual takes time $O(n)$, while calculating its fitness value (the solution calculated by KS95) takes time $O(n^2)$.

### 3.8. BRKGA and breakpoints

In the same way that we did with the mappings, we proposed a variation, less time consuming, for the BRKGA1 algorithm. It consists of changing the fitness function so that the fitness value is the number of reversal breakpoints from the associated $S$ mapping. The smaller the number of reversal breakpoints, the greater the individual's fitness value.

The rest of the algorithm remains unchanged, except that it returns the number of reversals calculated by KS95 for an individual with the highest fitness value, i.e., the least number of reversal breakpoints. This algorithm will be called BRKGA2. Now, note that calculating the fitness value of an individual takes time $O(n)$.

## 4. Experimental results

In order to compare, in practice, the quality of the solutions of the proposed algorithms, described in the previous sections, 11 sets, each with 100 inputs, were generated for experiments. Each input consists of two strings, $S$ and $T$, randomly generated as follows: an identity string $S$ is created and its elements are shuffled uniformly (all possible permutations of the elements in this string are equally likely to occur); after shuffling, we copy the resulting string $S$ to the string $T$, now obtaining two identical strings; finally, $d_{est}$ reversals are randomly applied to the string $S$, where $d_{est}$ is an integer defined for each of the sets of 100 inputs.

It is worth mentioning that, since $d_{est}$ reversals were applied to create each of the inputs, we know that the optimal reversal distance between the strings of each pair will necessarily be less than or equal to $d_{est}$.

Let $n$ be the length of the strings $S$ and $T$, described over an alphabet $\Sigma$, where $|\Sigma| = k$. In all sets defined for the tests, each pair of strings $S$ and $T$ will have the same number of occurrences for each symbol, i.e., $f(S, i) = f(S, j), \forall\, i, j \in \Sigma$. We will then denote the number of occurrences of each symbol in each of the strings only by an integer $f$. Consequently, for each of the eleven sets, we will have $n = kf$.

To form the 11 sets, we let $f = 5$ for nine of them, and $f = 2$ for the remaining two sets. In the nine sets where $f = 5$, we define $n \in \{20, 30, 40, 50, 60, 70, 80, 90, 100\}$. In the two sets where $f = 2$, we define $n \in \{50, 100\}$.

In the algorithms that receive additional parameters, apart from the pair of strings $S$ and $T$, we searched several sets of values for each of the parameters, in order to obtain a configuration that returned, on average, better results. For MAPS1 and MAPS2, the only additional parameter is the number $N$ of random mappings generated for the string $S$. We set $N = 100000$. BRKGA1 and BRKGA2 have five additional parameters, set as follows: $(p, g) \in \{(100, 1000), (200, 500), (500, 200), (1000, 100)\}$, $q_e \in \{0.15p,$

**Table 1. Experimental results of the proposed algorithms.**

| $n$ | $k$ | $f$ | $d_{est}$ | SELECTION | BREAKS | MCSP1 | MCSP2 | MAPS1 | MAPS2 | BRKGA1 | BRKGA2 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 20 | 4 | 5 | 10 | 6.76 | 9.35 | 11.14 | 7.86 | 7.37 | 8.49 | 5.67 | 6.85 |
| 30 | 6 | 5 | 10 | 11.10 | 15.36 | 16.29 | 11.38 | 14.04 | 15.50 | 8.48 | 9.80 |
| 40 | 8 | 5 | 15 | 17.04 | 24.46 | 24.40 | 16.78 | 21.87 | 24.15 | 12.97 | 14.99 |
| 50 | 10 | 5 | 15 | 20.73 | 28.87 | 27.37 | 18.94 | 29.87 | 32.00 | 14.87 | 17.51 |
| 60 | 12 | 5 | 20 | 28.45 | 40.30 | 35.73 | 24.52 | 38.57 | 41.43 | 19.98 | 23.13 |
| 70 | 14 | 5 | 20 | 31.48 | 42.21 | 38.39 | 25.82 | 46.76 | 49.87 | 21.16 | 24.21 |
| 80 | 16 | 5 | 25 | 38.99 | 52.76 | 47.37 | 32.08 | 55.65 | 59.00 | 27.21 | 30.65 |
| 90 | 18 | 5 | 25 | 41.23 | 55.94 | 48.39 | 32.18 | 64.23 | 67.06 | 28.90 | 31.42 |
| 100 | 20 | 5 | 25 | 44.23 | 56.96 | 51.30 | 33.60 | 72.89 | 75.81 | 31.56 | 33.71 |
| 50 | 25 | 2 | 15 | 21.96 | 19.33 | 26.89 | 18.18 | 17.96 | 19.36 | 14.61 | 15.74 |
| 100 | 50 | 2 | 25 | 42.49 | 32.99 | 49.85 | 32.02 | 44.52 | 46.93 | 25.61 | 27.71 |
| ERROR AVERAGE (%) | | | | 40.38 | 76.24 | 77.90 | 20.27 | 86.78 | 99.55 | −1.52 | 10.80 |

$0.2p$, $0.3p$, $0.4p\}$, $q_t \in \{0.15p$, $0.2p$, $0.3p$, $0.4p\}$ and $\rho_e \in \{0.55, 0.6, 0.65, 0.7\}$. All combinations were tested in a set of 100 entries where $n = 50$, and the best one was $p = 1000$, $g = 100$, $q_e = 0.3p$, $q_t = 0.3p$, and $\rho_e = 0.6$. For MCSP2, we used the GRIMM tool [Tesler 2002] to calculate the optimal solutions for the signed SbR.

Table 1 shows the experimental results obtained for each of the 11 sets of strings, one per line. The first four columns represent the values that define the equivalence class of the strings in each set: the length $n$, the size $k$ of the alphabet, the number of occurrences $f$ of each of the symbols and the number of reversals $d_{est}$ applied to one of the strings of each pair. The error of an algorithm is given by $(d_{alg} - d_{est})/d_{est}$, where $d_{alg}$ is the average distance estimated by the algorithm for the strings of a given set, which is the value in each cell. The average of the errors, given in the line ERROR AVERAGE, represents how far from $d_{est}$ the algorithm estimated the distance to the 1100 tested instances. The lower the value in ERROR AVERAGE, the better the performance of the algorithm in the tested sets, and a negative value in this line means that the algorithm returned, on average, better solutions (of less value) than $d_{est}$. Recall that negative values are possible since $d_{est}$ can be indeed greater than the real distance value.

When analyzing the results of Table 1, we can see that MAPS1 and MAPS2 algorithms were the ones that most distanced, on average, from the $d_{est}$ for the tested instances. Although BREAKS and MCSP1 showed worse results than MAPS1 and MAPS2 for some sets with strings of short length, this scenario was reversed in the sets with $f = 5$ and $n \geq 50$. Even when generating 100 thousand random mappings for each instance, the results showed that the growth of $n$ and/or $f$ values greatly increases the search space for random mappings, i.e., the number of mappings $m$ is very small compared to the total number of possible mappings for the string being mapped.

For a string $S$ such that $f(S, i) = f$ for all $i \in \Sigma$, the number of possible mappings $M$ is given by $M = f!f! \cdots f! = (f!)^k$. Calculating this value for the tested sets with the highest values of $n$ and $f = 5$, we can see that 100 thousand mappings represents an extremely small value in relation to the number of possible mappings.

The BREAKS and MCSP1 algorithms were the ones with the worst results after MAPS1 and MAPS2, which indicates that the choices of reversals of these algorithms have somewhat naive criteria. In the case of BREAKS, the problem may lie in the fact that we are often unable to eliminate breakpoints and have to use criterion of increasing the

length of common extremes. In the case of MCSP1, the problem is clearer, as we wanted to demonstrate the difference between the easiest choices of reversals (criterion of sorting the parts of the common partition via Selection Sort) and the modeling of the common partitions into signed permutations, which can be solved optimally.

The algorithm that follows, in ascending order of the average quality of the solutions, is the SELECTION algorithm. An interesting fact observed in its results was that the averages of the values of its solutions were not lower whenever the value of $f$ was lower, in the sets with $n = 50$ and $n = 100$, having occurred only in the last case.

Between the first two algorithms, SELECTION was much better than BREAKS when $f = 5$. However, for both sets with $f = 2$, BREAKS algorithm took advantage of the first. This result is interesting because removing breakpoints works better in permutations, which we can see as strings where $f = 1$. An example of this is the KS95 algorithm, which uses this strategy in permutations and has a well-defined approximation factor.

The use of the exact polynomial time algorithm for signed permutations was probably the reason why MCSP2 was the third best algorithm in our experiments, worse only than the two BRKGA implementations.

The two versions of BRKGA took advantage over the other algorithms and, between the two, the first one returned shorter distances, as already expected, because the fitness value is the solution itself by KS95.

It is important to highlight that the growth, even if not so expressive, in the values of $n$ and $f$ can dramatically increase the search space for mappings, as revealed by calculating the total number of possible mappings for a string. Thus, for high values of these two variables, even the BRKGA with a well-adjusted configuration may need many generations to sufficiently explore the solution space, which can make this method unfeasible in terms of time consumption, when looking for quality solutions similar to those returned in the tests performed in this work.

Precisely for this reason, it is worth mentioning that, for strings with many occurrences of each symbol, or of very long lengths, the second version of BRKGA may be more interesting, since the lower cost in relation to time allows the number of generations (and/or the size of the populations) created to be greater than what would be achieved with the same time consumption of the first version.

Finally, we will comment on the execution time of the algorithms in practice. Due to some difficulties we had, not all the results presented in Table 1 had their time recorded. Thus, we performed some new tests in order to evaluate that, on a specific set of 100 strings, with $n = 50$, $f = 5$ and $d_{est} = 15$, and calculated the average time, in milliseconds, that each one took to return its solution for each instance of this set: SELECTION $= 4.85$ ms; BREAKS $= 28.44$ ms; MAPS1 $= 129261.80$ ms; MAPS2 $= 564.36$ ms; BRKGA1 $= 103389.63$ ms; BRKGA2 $= 1153.40$ ms; MCSP1 $= 34.35$ ms. For MCSP2, because we used the GRIMM tool, we were unable to calculate its exact execution time; however, when timing the approximate time spent to solve all instances of the test set, we conclude that the time of this algorithm is close to that of MAPS2.

When analyzing the execution times in the test set, one can see that even for strings with $n = 50$, MAPS1 and BRKGA1 take a long time to return their solutions,

when compared to the other algorithms, because they need a very large number of mappings/individuals to return good quality solutions, having to calculate an approximate solution, using KS95, for each of these mappings/individuals, as already mentioned.

## 5. Conclusions

In this work, we proposed and compared several practical algorithms for the problem of Transforming Strings by Reversals and the algorithm with the best results, on average, was the implementation of the BRKGA metaheuristic. However, the BRKGA version that returned the best solutions, BRKGA1, is very time consuming when strings are too long. In the future, it would be interesting to study structural aspects of the problem, in order to develop algorithms with well-defined and smaller approximation factors than those already existing in the literature. In addition, this work considered only the unsigned version of TSbR, and obtaining algorithmic results for the signed version of the problem should also be one of the objectives for future work.

## References


Berman, P., Hannenhalli, S., and Karpinski, M. (2002). 1.375-Approximation Algorithm for Sorting by Reversals. In Möhring, R. and Raman, R., editors, *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'2002)*, volume 2461 of *Lecture Notes in Computer Science*, pages 200–210. Springer-Verlag, Berlin, Germany.

Chen, X., Zheng, J., Fu, Z., Nan, P., Zhong, Y., Lonardi, S., and Jiang, T. (2005). Assignment of Orthologous Genes via Genome Rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(4):302–315.

Chrobak, M., Kolman, P., and Sgall, J. (2004). The Greedy Algorithm for the Minimum Common String Partition Problem. In Jansen, K., Khanna, S., Rolim, J. D. P., and Ron, D., editors, *Proceedings of the 7th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'2004), and 8th International Workshop on Randomization and Computation (RANDOM'2004)*, pages 84–95, Berlin, Heidelberg. Springer.

Gonçalves, J. F. and Resende, M. G. C. (2010). Biased random-key genetic algorithms for combinatorial optimization. *Journal of Heuristics*, 17(5):487–525.

Hannenhalli, S. and Pevzner, P. (1996). To Cut ... or Not to Cut (Applications of Comparative Physical Maps in Molecular Evolution). In Tardos, E., editor, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'1996)*, pages 304–313, Philadelphia, PA, USA. Society for Industrial and Applied Mathematics.

Hannenhalli, S. and Pevzner, P. A. (1999). Transforming Cabbage into Turnip: Polynomial Algorithm for Sorting Signed Permutations by Reversals. *Journal of the ACM*, 46(1):1–27.

Kececioglu, J. D. and Sankoff, D. (1995). Exact and Approximation Algorithms for Sorting by Reversals, with Application to Genome Rearrangement. *Algorithmica*, 13:180–210.

Kolman, P. and Waleń, T. (2007). Reversal Distance for Strings with Duplicates: Linear Time Approximation Using Hitting Set. In Erlebach, T. and Kaklamanis, C., editors, *Proceedings of the 4th International Workshop on Approximation and Online Algorithms (WAOA'2006)*, pages 279–289, Berlin, Heidelberg. Springer.

Siqueira, G., Brito, K. L., Dias, U., and Dias, Z. (2020). Heuristics for reversal distance between genomes with duplicated genes. In *International Conference on Algorithms for Computational Biology*, pages 29–40. Springer.

Tesler, G. (2002). GRIMM: Genome Rearrangements Web Server. *Bioinformatics*, 18(3):492–493.