Journal of Universal Computer Science, vol. 23, no. 9 (2017), 868-906
submitted: 2/9/16, accepted: 26/9/17, appeared: 28/9/17 © *J.UCS*

On the Sorting by Reversals and Transpositions Problem

Andre Rodrigues Oliveira
(Institute of Computing - University of Campinas, Campinas, Brazil
andrero@ic.unicamp.br)

Ulisses Dias
(School of Technology - University of Campinas, Limeira, Brazil
ulisses@ft.unicamp.br)

Zanoni Dias
(Institute of Computing - University of Campinas, Campinas, Brazil
zanoni@ic.unicamp.br)

Abstract: Reversals and transpositions are two classic genome rearrangement operations. Reversals occur when a chromosome breaks at two locations called breakpoints and the DNA between the breakpoints is reversed. Transpositions occur when two adjacent blocks of elements exchange position. This paper presents a polynomial-time approximation algorithm for the Sorting by Reversals and Transpositions Problem. Our algorithm applies to both signed and unsigned versions of the problem, and it treats both cases in a unified manner. We prove an approximation factor of 2 for signed permutations and $2k$ for the unsigned case, where k is the approximation factor of the algorithm used for cycle decomposition, but in our practical experiments our algorithm found results with approximation ratio better than 1.5 in more than 99% of the signed permutations and better than 1.8 in more than 97% of the unsigned permutations. Our analysis also shows that our algorithm outperforms any other approach known to date.
Key Words: approximation algorithms, genome rearrangement, sorting by reversals and transpositions
Category: F.2.0, G.2.3

1 Introduction

Reversals and transpositions are classic genome rearrangement operations. Reversals occur when a chromosome breaks at two locations called breakpoints and the DNA between the breakpoints is reversed. Transpositions occur when two adjacent blocks of elements exchange position.

We represent genomes as sequences of segments called syntenic blocks, which we assume to be shared by the genomes being compared. Let n be the total number of shared segments. We assign a unique number in the set $\{1, \ldots, n\}$ to each segment such that chromosomes can be regarded as permutations. When segments shared by two genomes have different orientations, we assign different signs ('+' or '$-$') to their unique numbers. When the orientation of the segments is not considered, the unique numbers have no signs.

A sequence of shared segments can be seen as a permutation of their unique numbers, and the permutation is how we represent the genome itself. We sort a given permutation by applying successive operations that transform it into a permutation where all elements are positive and in ascending order. The main goal is to find the minimum number of such operations, which represents a most parsimonious scenario.

Reversals and transpositions lead to two classic genome rearrangement problems called "The Sorting by Reversals Problem" and "The Sorting by Transpositions Problem", respectively. The former has polynomial algorithms in the signed version [Bergeron 2005, Hannenhalli and Pevzner 1999, Tannier et al. 2007], and the unsigned version is NP-Hard [Caprara 1999] and the best approximation factor known is 1.375 [Berman et al. 2002]. The sorting by transpositions problem has only the unsigned version and it is also NP-Hard [Bulteau et al. 2012]. The best approximation factor known to date is 1.375 [Elias and Hartman 2006].

Several efforts have been made to develop algorithms that consider both reversals and transpositions. The first algorithms were developed by Walter, Dias, and Meidanis [Walter et al. 1998]. They presented a 2-approximation algorithm for the signed version and a 3-approximation algorithm for the unsigned version of "The Sorting by Reversals and Transpositions Problem". The signed version uses a structure called the cycle graph whereas the unsigned version uses breakpoints. The concept of breakpoints was further explored by Dias *et al.* to create a greedy algorithm for unsigned permutations [Dias et al. 2014a].

In 1999, Gu *et al.* [Gu et al. 1999] introduced a genome rearrangement operation called inverted transposition. This operation breaks the chromosome at two locations, then reverses the DNA between these locations and places it in another position in the same chromosome. Gu *et al.* gave a 2-approximation algorithm for the minimal number of operation needed to sort a signed permutation by reversals, transpositions, and inverted transpositions. In 2002, Eriksen [Eriksen 2002] claimed that an algorithm looking for the minimal number of such operations will produce a solution heavily biased toward transpositions. Thus, they gave an approximation algorithm with factor $1 + \epsilon$ when the weight of reversals is 1 and the weights of transpositions and inverted transpositions are 2.

Rahman, Shatabda, and Hasan [Rahman et al. 2008] used the cycle graph to present a $2k$-approximation algorithm for the problem of sorting unsigned permutations by reversals and transpositions, where k is the approximation of the algorithm used for cycle decomposition [Caprara 1999]. We adapted this algorithm for signed permutations, which resulted in a 2-approximation algorithm [Dias et al. 2014b]. At present the best known approximation ratio achievable for the cycle decomposition is $k = \frac{17}{12} + \epsilon \approx 1.4167 + \epsilon$, for any positive ϵ, published by Chen [Chen 2013]. This algorithm is a modification on Lin and Jiang's approximation algorithm that has $k = \frac{5073-15\sqrt{1201}}{3208} + \epsilon \approx 1.4193 + \epsilon$, for any positive ϵ [Lin and Jiang 2004]. We used the original Lin and Jiang's algorithm in our implementation because of its simplicity.

Here, we present an algorithm for the problem of sorting permutations by reversals and transpositions. Our algorithm applies to both signed and unsigned versions, and it treats both cases in a unified manner. Our algorithm does not improve the best approximation factors of 2 and $2k$ for signed and unsigned permutations, respectively. However, the only step which does not guarantee an approximation ratio better than 1.8 and $1.8k$ for signed and unsigned permutations, respectively, occurs in rare situations as shown by practical experiments.

This paper is organized as follows. Section 2 defines the notation we use throughout the paper and provides a formal presentation for our problem. Section 3 describes a basic algorithm that can be used to optimally sort a large number of instances, but it does not provide answers for a particular set. We study this set and provide an approach to compute a 2.0-approximate value in Section 4. In Section 5, we carry out a practical analysis that shows that our algorithm outperforms other methods.

2 Background

Throughout this paper, we represent a genome with n conserved blocks as a *permutation* $\pi = (\pi_1\ \pi_2\ \dots\ \pi_n)$, $\pi_i \in \mathbb{Z}$, $1 \leq |\pi_i| \leq n$, and $|\pi_i| \neq |\pi_j|$ for all $i \neq j$. The permutation such that all elements are positive and in ascending order is called *identity*, and we represent it as $\iota = (1\ 2\ \dots\ n)$.

Two kinds of permutations are possible: signed and unsigned. For a *signed permutation*, every element π_i has a plus $(+)$ or minus $(-)$ sign that indicates the orientation of the block that it represents. For an *unsigned permutation*, the block orientation is unknown. Therefore, only the order of the blocks is represented.

A *reversal* is an operation $\rho_r(i, j)$, $1 \leq i \leq j \leq n$, that reverses the order of $\pi[i..j]$ for any permutation π. This operation also changes the signs of each element between π_i and π_j if π is a signed permutation. When π is unsigned, reversals $\rho_r(i, j)$ such that $i = j$ are not considered, since they do not change the permutation. Therefore, if π is an unsigned permutation we have $(\pi_1\ \dots\ \pi_{i-1}\ \underline{\pi_i\ \pi_{i+1}\ \dots\ \pi_j}\ \pi_{j+1}\ \dots \pi_n) \cdot \rho_r(i, j) = (\pi_1\ \dots\ \pi_{i-1}\ \underline{\pi_j\ \dots\ \pi_{i+1}\ \pi_i}\ \pi_{j+1}\ \dots \pi_n)$. In a similar way, if π is a signed permutation we have $(\pi_1\ \dots \pi_{i-1}\ \underline{\pi_i\ \pi_{i+1}\ \dots\ \pi_j}\ \pi_{j+1}\ \dots \pi_n) \cdot \rho_r(i, j) = (\pi_1\ \dots\ \pi_{i-1}\ \underline{-\pi_j\ \dots\ -\pi_{i+1}\ -\pi_i}\ \pi_{j+1}\ \dots \pi_n)$.

A *transposition* is an operation $\rho_t(i, j, k)$, $1 \leq i < j < k \leq n+1$, that moves a block of contiguous elements $\pi_i \dots \pi_{j-1}$ to a new location between π_{k-1} and π_k. Therefore, $(\pi_1\ \dots\ \pi_{i-1}\ \underline{\pi_i\ \dots\ \pi_{j-1}}\ \underline{\pi_j\ \dots\ \pi_{k-1}}\ \pi_k\ \dots\ \pi_n) \cdot \rho_t(i, j, k) = (\pi_1\ \dots\ \pi_{i-1}\ \underline{\pi_j\ \dots\ \pi_{k-1}}\ \underline{\pi_i\ \dots\ \pi_{j-1}}\ \pi_k\ \dots\ \pi_n)$. Observe that a transposition does not affect the sign of any element and thus it has the same impact on permutations no matter if they are signed or unsigned.

A *model* M is the set of allowed operations that can be applied to an arbitrary permutation π. The *distance* $d_M(\pi)$ is the minimum number d of operations $\rho_1, \rho_2, \ldots, \rho_d \in M$ such that $\pi \cdot \rho_1 \cdot \rho_2 \cdot \ldots \cdot \rho_d = \iota$. Therefore, the *reversal distance* $d_r(\pi)$ is the minimum number of reversals that transform π into the identity permutation, while the *transposition distance* $d_t(\pi)$ is the minimum number of transpositions that transform π into the identity permutation. When the model includes both reversals and transpositions, we define the *reversal and transposition distance* $d_{rt}(\pi)$, which is the minimum number of reversals and transpositions that transform π into the identity permutation.

2.1 Cycle Graph

We map any signed permutation π to a *cycle graph* [Bafna and Pevzner 1998] $G(\pi)$ as follows. For each element π_i we add two vertices to $G(\pi)$, namely, $-\pi_i$ and $+\pi_i$. We also add the vertices 0 and $-(n+1)$. Therefore, the vertex set of $G(\pi)$ is $\{-n, \ldots, -2, -1, 1, 2, \ldots, n\} \cup \{0, -(n+1)\}$. Two sets of edges can be defined: the *gray edge* set is $\{(+(i-1), -i) : 1 \leq i \leq n+1\}$, and the *black edge* set is $\{(-\pi_i, +\pi_{i-1}) : 1 \leq i \leq n+1\}$. The black edges link elements that are side by side in π and the gray edges link elements that are side by side in ι. For example, the permutation $\pi = (+5 \ \ -3 \ \ -4 \ \ +2 \ \ +1)$ generates the vertex set $\{0, -5, +5, +3, -3, +4, -4, -2, +2, -1, +1, -6\}$ and the edges shown in Figure 1.

Observe that we draw the same graph in two forms: linear and circular. The *linear form* is illustrated in Figure 1(a), where vertices are drawn on a horizontal line in the same order as the elements of π. The black edges are drawn as linear edges, and the gray edges are drawn as arcs. The *circular form* is illustrated in Figure 1(b), where we close together the two endpoints of the line in the first form to make a circle putting the black edges on the circumference of the circle and the gray edges inside the circle.

For each element $\pi_i \in \pi$, the vertex $-\pi_i \in G(\pi)$ is drawn before the vertex π_i as one can observe in Figure 1(a) by reading the vertices from left to right. This convention is useful to retrieve the signs of elements in the original permutation. For instance, we can see that in the cycle graph the vertex -5 happens before $+5$ and hence 5 is positive in the permutation π, whereas $+3$ appears before -3 and hence 3 is negative in the permutation π.

When π is unsigned, we can map it to a signed permutation π' by assigning arbitrary signs to each element π_i. In the end, we associate $G(\pi')$ with π. Therefore, π can be mapped to several cycle graphs and the best graph for our purposes is the one that maximizes the number of cycles. We postpone the definition of cycles until Section 2.2; what is important now is the fact that finding the best graph is an NP-hard problem [Caprara 1999]. Thus, we rely on approximations to find cycle graphs for unsigned permutations. The best approximation

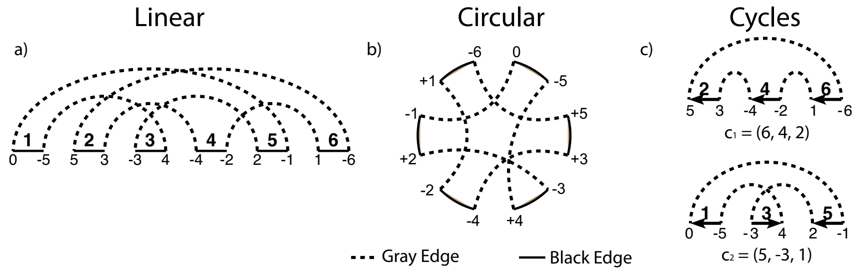

Figure 1: Cycle graph $\mathbf{G}(\pi)$ for $\pi = (+\mathbf{5}\ \ -\mathbf{3}\ \ -\mathbf{4}\ +\mathbf{2}\ +\mathbf{1})$ shown in different representations in (a) and (b). In (c) we present the cycles $c_1 = (-6, 1, -2, -4, 3, 5)$ and $c_2 = (-1, 2, -3, 4, -5, 0)$, which can also be represented as $c_1 = (6, 4, 2)$ and $c_2 = (5, -3, 1)$ given the labels of the black edges in the cycles.

algorithm has approximation factor $1.4167 + \epsilon$, for $\epsilon > 0$ [Chen 2013].

Since both signed and unsigned permutations can be mapped to cycle graphs, we hereafter treat both cases in a unified manner. We will apply reversals and transpositions directly on the cycle graph, and that will be enough to sort the permutation (signed or unsigned) that underlies it.

2.2 Cycles

Each vertex in $G(\pi)$ is incident on one gray edge and one black edge, which allows a unique decomposition of edges in cycles of alternating colors. In Figure 1, we have two cycles: $c_1 = (-6, 1, -2, -4, 3, 5)$ and $c_2 = (-1, 2, -3, 4, -5, 0)$. A cycle c can be written in many possible ways depending on the choice of the first vertex and on the direction we traverse the edges. To make the representation unique, we assume that the first vertex v_1 in the cycle is the "rightmost" element (terms like "rightmost" or "leftmost" refer to the linear form of drawing the cycle graph). In addition, we assume that the second element is the vertex that is linked to v_1 by a black edge. The cycle representations of c_1 and c_2 presented in Figure 1(c) keep these restrictions in mind.

We simplify our cycle representation by first numbering the black edges of the cycle graph $G(\pi)$ from 1 to $n + 1$. We assign label i to a black edge that links $-\pi_i$ to $+\pi_{i-1}$. Let $(v_1, v_2, v_3, v_4, \ldots)$ be a cycle written using the rules previously described; the simplified representation lists the labels for the black edges $\{(v_1, v_2),\ (v_3, v_4), \ldots\}$ in order, and we use '+' or '$-$' to recognize how black edges are traversed. If the black edge is traversed from right to left then it is assigned a '+' sign and we say it is a positive edge; otherwise, it is assigned a '$-$' sign and we say it is a negative edge. Since v_1 is the rightmost element in the cycle, it is easy to see that (v_1, v_2) is always traversed from right to left, so the first black edge in our simplified representation is always positive.

Figure 1(c) presents the cycles c_1 and c_2 in the simplified notation. Keep in mind that the signs are not assigned to a black edge itself, but to its label in the cycle representation (observe the black edge labeled as 3 in c_2). In addition, the cycle graph is not a digraph and a black edge $B = (a, b)$ can also be written as $B = (b, a)$. We are using arrows in Figure 1(c) only to clarify how the black edges are being traversed inside the cycle.

We say that two or more black edges are *convergent* if they are in the same cycle and have the same sign. We say that two black edges are *divergent* if they are in the same cycle and have different signs.

We denote the number of cycles in $G(\pi)$ as $c(\pi)$. The *size* of a cycle is the number of black edges in it. A cycle is *odd* if its number of black edges is odd and it is *even* otherwise. The number of odd cycles is denoted by $c_{odd}(\pi)$. The identity is the only permutation such that every cycle has only one black edge. Therefore, let $b(G(\pi)) = n+1$ be the number of black edges in $G(\pi)$; the sequence of operations that sort π must increase the number of odd cycles from $c_{odd}(\pi)$ to $b(G(\pi))$.

Let $\Delta c(\rho) = c(\pi \cdot \rho) - c(\pi)$ be the variation in the number of cycles, and let $\Delta c_{odd}(\rho) = c_{odd}(\pi \cdot \rho) - c_{odd}(\pi)$ be the variation in the number of odd cycles when the operation ρ is applied. Then, $\Delta c(\rho_t) \in \{2, 1, 0, -1, -2\}$ for any transposition ρ_t and $\Delta c(\rho_r) \in \{1, 0, -1\}$ for any reversal ρ_r [Walter et al. 1998].

Bafna and Pevzner proved that $\Delta c_{odd}(\rho_t) \in \{2, 0, -2\}$ for the sorting by transposition problem [Bafna and Pevzner 1998], which can be seen as a particular case in the sorting by reversals and transpositions problem where every cycle has only positive black edges. The proof presented by Bafna and Pevzner still holds for transpositions $\rho_t(i, j, k)$ if the black edges i, j or k that are in the same cycle have the same signs, which leads us to Lemma 1.

Lemma 1. *Let $\rho_t(i, j, k)$ be a transposition such that the black edges i, j, and k are convergent. Then, $\Delta c_{odd}(\rho_t) \in \{2, 0, -2\}$.*

A transposition $\rho_t(i, j, k)$ *acts* on edges i, j, and k, $(i < j < k)$; and a reversal $\rho_r(i, j)$ *acts* on edges i and j, $(i < j)$. The next four lemmas further discuss the impact of operations on the number of cycles in a cycle graph.

Lemma 2. *Let $\rho_t(i, j, k)$ be a transposition that acts on two black edges that are divergent. The third black edge may be in the same cycle or in a different one. Then, $\Delta c(\rho_t) \leq 1$.*

Proof. Let i, j, and k be of the form $i = (v_{i_1}, v_{i_2})$, $j = (v_{j_1}, v_{j_2})$, and $k = (v_{k_1}, v_{k_2})$. After the transposition, these black edges will be destroyed and three will be created of the form (v_{i_1}, v_{j_2}), (v_{j_1}, v_{k_2}), and (v_{k_1}, v_{i_2}).

First we consider the case where the third edge k is in the same cycle as i and j. Since we have three edges and one pair diverges, there is a pair that converges. Let us assume without loss of generality that i and j are divergents and j and k are convergents. Therefore, there is a path between v_{i_1} and v_{j_1} that contains neither v_{i_2} nor v_{j_2}, and a path between v_{i_1} and v_{k_1} that contains neither v_{i_2} nor v_{k_2}.

1. If the path between v_{i_1} and v_{j_1} does not contain k, the newly created black edges (v_{i_1}, v_{j_2}) and (v_{j_1}, v_{k_2}) are in the same cycle.

2. If the path between v_{i_1} and v_{j_1} contains k, so the path between v_{i_1} and v_{k_1} does not contain j. Therefore, the newly created black edges (v_{i_1}, v_{j_2}) and (v_{k_1}, v_{i_2}) are in the same cycle.

These cases lead us to the conclusion that ρ_t will remove the cycle from $G(\pi)$ and add at most two new cycles in $G(\pi)$.

Now we consider the case where the third edge k is in a different cycle. After we break the black edges i, j, and k, there will be three distincts paths that do not share any vertice: (i) $v_{i_1} \dots v_{j_1}$, (ii) $v_{i_2} \dots v_{j_2}$, and (iii) $v_{k_1} \dots v_{k_2}$. The newly create black edges (v_{i_1}, v_{j_2}), (v_{j_1}, v_{k_2}), and (v_{k_1}, v_{i_2}) will join these three paths in a single cycle. $\square$

Lemma 3. *Let $\rho_t(i, j, k)$ be a transposition that acts on two black edges that are divergent. The third black edge may be in the same cycle or in a different one. Then $\Delta c_{odd}(\rho_t) \leq 2$.*

Proof. The proof from Lemma 2 shows that one or two cycles are removed from $G(\pi)$ and at most two new cycles are added to $G(\pi)$. Therefore, the best we can do is to assume that the removed cycles are even and that the added cycles are odd, which would lead to at most $\Delta c_{odd}(\rho_t) = 2$. $\square$

Hannenhalli and Pevzner proved that $\Delta c(\rho_r) \in \{1, 0, -1\}$ for any reversal ρ_r [Hannenhalli and Pevzner 1999]. We state Lemma 4 about the effect of ρ_r in the number of odd cycles.

Lemma 4. *For any reversal $\rho_r(i, j)$, $\Delta c_{odd}(\rho_r) \in \{2, 0, -2\}$.*

Proof. If i and j are black edges in different cycles, then the reversal will remove two cycles from $G(\pi)$ and add a new cycle. If both the removed cycles are odd, then the new cycle is even and $\Delta c_{odd} = -2$. If at least one removed cycle is even, then $\Delta c_{odd} = 0$.

If i and j are black edges in the same cycle, then $\rho_r(i, j)$ will remove one cycle and add one or two cycles. If it adds only one cycle, then $\Delta c_{odd} = 0$. Otherwise, (i) if the removed cycle is odd, then $\Delta c_{odd} = 0$, and (ii) if the removed cycle is even, then the new cycles must be both odd ($\Delta c_{odd} = 2$) or both even ($\Delta c_{odd} = 0$). □

Lemmas 1, 3 and 4 lead to Lemma 5.

Lemma 5. $\Delta c_{odd}(\rho_t) \leq 2$ *and* $\Delta c_{odd}(\rho_r) \leq 2$.

The sequence of operations that sorts a non-identity permutation π increases the number of odd cycles from $c_{odd}(\pi)$ to $b(G(\pi))$. Since, from Lemma 5, the maximum increase due to a single operation is 2, a lower bound on $d_{rt}(\pi)$ follows.

Theorem 6. $d_{rt}(\pi) \geq \frac{b(G(\pi)) - c_{odd}(\pi)}{2}$.

2.3 Complement Graph

Assume that we have a cycle graph in linear form computed from an unknown permutation. To obtain the permutation, we start by assigning the label 0 to the leftmost vertex. After that, we can safely assign the label -1 to the vertex linked to 0 by a gray edge. The two vertices 0 and -1 are linked to each other by a gray edge and linked to other two vertices by black edges. We can make no assumption about the labels of these vertices based solely on the black edges. However, we know that a vertex e is always side by side with $-e$ and hence we assign labels to two new vertices (1 and $-(n + 1)$). Next, we follow the gray edges linked to these two new vertices and repeat the same procedure.

In summary, our procedure has two steps: (i) find the label of a vertex x based on a gray edge and (ii) assign the label $-x$ to the element that is side by side with x but is not linked to it by a black edge. The procedure stops when we eventually reach the rightmost element.

A graph called *the complement graph* and hereafter referred to as $\bar{G}(\pi)$ highlights this procedure. We derive *the complement graph* from the cycle graph by removing the black edges and inserting the complement edges linking x and $-x$ for every $x \in \{1 \dots n\}$. We also place a complement edge linking 0 and $-(n+1)$.

In Figure 2 we illustrate the steps to build the complement graph. The cycle graph $G(\pi)$ for permutation $\pi = (+3\ -2\ -1\ +4\ -5)$ is given. We remove from $G(\pi)$ the black edges leading to the graph shown in Figure 2(b), which contains only gray edges. In Figure 2(c) we add the complement graphs linking x and $-x$ for every $x \in \{1 \dots n\}$; and we rearrange gray edges so a Hamiltonian cycle is easily noticeable.

The same steps can be made for any permutation and, by construction, it will always generate a Hamiltonian cycle in the complement graph.

2.4 Configuration and Component

Let $G(\pi) = (V, E)$ be a cycle graph. A *configuration* is a graph $C = (V', E')$ built from $G(\pi)$ by removing one or more cycles. Let U be the set of vertices

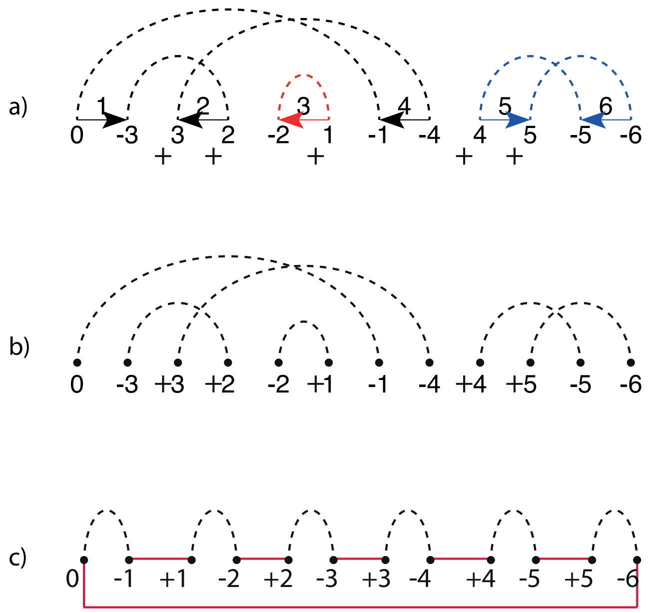

Figure 2: Steps to build the complement graph. In (a) we show the cycle graph $\mathbf{G}(\pi)$ for $\pi = (+\mathbf{3}\ -\mathbf{2}\ -\mathbf{1}\ +\mathbf{4}\ -\mathbf{5})$. In (b) we remove the black edges. In (c) we place the remaining gray edges side by side, and we add complement edges linking x and $-x$ for every $x \in \{1 \ldots n\}$. It is clear that there is a Hamiltonian path from 0 to $-(n+1)$ in the complement graph. Since we also include a complement edge linking 0 and $-(n+1)$, the complement graph has a Hamiltonian cycle.

in the cycles we are removing, and let F be set of edges incident on elements of U, $V' = V - U$ and $E' = E - F$. We say that C is a *subset* of $G(\pi)$, which is represented as $C \subset G(\pi)$. The subset relation may be generalized to any configuration, we say that $C_2 \subset C_1$ if C_2 is obtained after removing cycles from C_1.

Some black edges were removed from $G(\pi)$ to build C, so we need to reassign labels to remaining ones in C such that each black edge receives a label in the range $\{1..b(C)\}$, where $b(C)$ now represents the number of black edges in the configuration C. Note that we keep the relative ordering of black edges unchanged, meaning that if two black edges were labeled b_1 and b_2 in $G(\pi)$ and are now labeled as b'_1 and b'_2 in C, respectively, then $b_1 < b_2$ if and only if $b'_1 < b'_2$.

Theorem 6 also holds for configurations since it uses properties that are common both for cycle graphs and configurations. Let $b(C)$ be the number of black edges in C and c_{odd} be the number of odd cycles in C. Theorem 7 is the Theorem 6 counterpart for configurations.

Theorem 7. $d_{rt}(C) \geq \frac{b(C)-c_{odd}(C)}{2}$.

A *complement configuration* $\bar{C}$ for C is build in a similar way we built complement graphs for $G(\pi)$. Let b_1 be a black edge in C, $1 \leq b_1 \leq b(C)-1$; we build complement edges linking the rightmost vertex of b_1 with the leftmost vertex of b_1+1 and linking the leftmost vertex of the black edge labeled as 1 with the rightmost vertex of the black edge labels as $b(C)$.

We say that a configuration C is *complete* if its complement graph $\bar{C}$ has one Hamiltonian cycle; otherwise it is *incomplete*. When C is complete, it is possible to find a permutation σ such that C is equal to $G(\sigma)$. That is not possible when C is incomplete, because $\bar{G}(\sigma)$ has one Hamiltonian cycle for every permutation σ as shown in Section 2.3.

Figure 3 depicts an incomplete configuration both in linear and circular forms. It is incomplete because $\bar{C}$ shown in Figure 3(c) and Figure 3(d) has two cycles instead of a single Hamiltonian cycle.

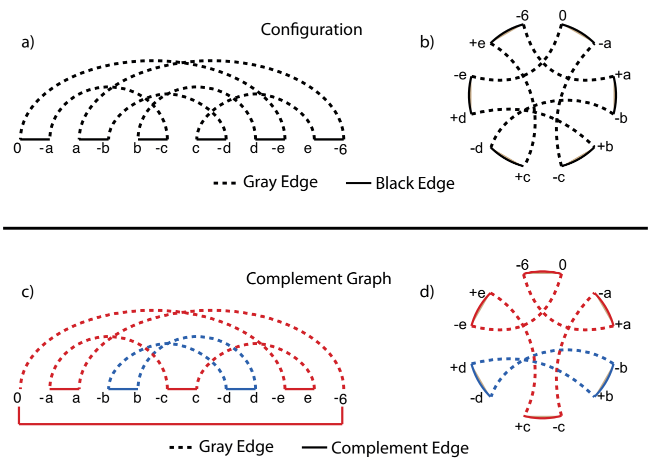

Figure 3: We present an incomplete configuration in linear (a) and circular (b) forms. Therefore, this configuration cannot be built from any permutation. The configuration is incomplete because its complement graph, as shown in (c) and (d), has two cycles instead of a Hamiltonian cycle.

Let g_1 be a gray edge that links the black edges labeled as x_1 and y_1, and let g_2 be a gray edge that links the black edges labeled as x_2 and y_2. Let us assume, without loss of generality, that $x_1 < y_1$ and $x_2 < y_2$. We say that g_1 *intersects* g_2 if $x_1 < x_2 < y_1 < y_2$, or in the case that $x_1 < x_2 = y_1 < y_2$, g_1 must link to the rightmost vertex in y_1.

We say that two cycles c and d *intersect* if there is a gray edge in c that intersects with a gray edge in d. One configuration C is *connected* if, for any two cycles c_1 and c_k, there are cycles $c_2, \ldots, c_{k-1}$ such that c_i intersects with c_{i+1}, for $1 \leq i \leq k-1$.

A configuration C is a *component* if it is complete and connected. Let C be a component in a cycle graph $G(\pi)$; we can always find a permutation σ such that $G(\sigma) = C$. Therefore, if we consider the component as a whole, it is also a small cycle graph for another permutation.

2.5 Equivalence Class

All configurations can be rotated to the right or left using the circular form, which will lead to different permutations with similar cycle graphs as we can see in Figure 4. The cycle graphs built from permutations $(-2\ +1)$, $(-1\ -2)$, and $(+2\ -1)$ are *equivalent*, because one can be reached from the other by cyclically shifting the graph in the circular form as shown in Figure 4(a). This equivalence is hard to observe if we examine the graph only in the linear form shown in Figure 4(b). We say that these graphs (and as a consequence the permutations that lead to them) are in the same *equivalence class*.

Since these permutations have the same graph in the circular form, the sequence of operations that sorts one of them can be adapted to sort another. Let π and σ be two permutations in the same equivalence class, each operation that affects $G(\pi)$ has an equivalent in $G(\sigma)$. We first identify the black edges acted upon by the operation in $G(\pi)$ and locate their counterparts in $G(\sigma)$. After that, we identify the operation that acts on these black edges in σ. Since we can do this to any operation, we know π and σ have the same distance.

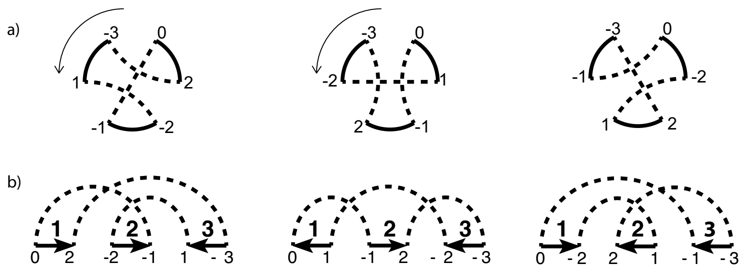

Figure 4: The cycle graphs built from permutations $(-2\ +1)$, $(-1\ -2)$ and $(+2\ -1)$ are in the same equivalence class, since one can be reached from the others by cyclically shifting the graph in the circular form as shown in (a). This equivalence is hard to observe if we look to the same graph as depicted in (b).

As an example, we depict in Figure 5(a) one sorting sequence for an entire class of configurations. We save the configuration as a list of cycles and the black edges that are affected by the operations. Figures 5(b), 5(c), and 5(d) illustrates how one adapt the generic sorting to fit individual permutations. The transposition always affect the three black edges, so it is $\rho_t(1, 2, 3)$ in all the examples. The reversal applied after the transposition differs for each case, so we need to keep track of which black edge in the generic configuration corresponds to each black edge in the cycle graph for permutations. In our examples, the reversal applied on edges y and z are mapped to the reversals $\rho_r(2, 3)$ in Figure 5(b), $\rho_r(1, 3)$ in Figure 5(c) and $\rho_r(1, 2)$ in Figure 5(d).

2.6 Simple Permutations

A cycle is *short* if it has 3 or fewer black edges; otherwise it is *long*. A cycle graph is *simple* if it has only short cycles, and we say a permutation is *simple* if its cycle graph is simple. Transforming a permutation π into a simple permutation $\hat{\pi}$ includes the addition of new elements to break long cycles.

We break long cycles as follows. Let b_1 be a black edge in C and denote by b_2 the black edge connected to b_1 via a gray edge, and b_3 the black edge connected to b_2 via a gray edge. Let g be the gray edge connected to b_1 but not to b_2. Then, we break the edges b_3 and g to insert two new vertices in the graph. Assuming $b_3 = (v_b, w_b)$ and $g = (w_g, v_g)$, as shown in Figure 6, we remove the edges b_3 and g. After that, we add two new vertices v and w between the endpoints of the former black edge b_3, and we create two new black edges $(v_b,\ v)$ and $(w,\ w_b)$. Finally, we add the gray edges $(w_g,\ w)$ and $(v,\ v_g)$.

Each time we add a cycle we are increasing both the number of black edges and the number of odd cycles. Therefore, the transformation guarantees that $d_{rt}(\pi) \leq d_{rt}(\hat{\pi})$ and that both π and $\hat{\pi}$ have the same lower bound in Theorem 6, which allows us to make assertions about the approximation factor. We repeat the procedure until no cycle of size greater than 3 is found.

A sequence that sorts $\hat{\pi}$ can be adapted to sort π with the same number of operations by ignoring the elements added to $\hat{\pi}$. For example, if a reversal acts on the elements $[a, b, c, d, e]$ such that b, d were added to $\hat{\pi}$ and are not in π, then the reversal should be adapted to π by making it affect only $[a, c, e]$.

3 The Algorithm

Our algorithm focus on applying moves that increase the number of odd cycles, which differs from previous approaches that centered the analysis on the overall number of cycles [Rahman et al. 2008, Walter et al. 1998]. We identify in Lemmas 8, 9, and 10 three major characteristics that make it feasible to increase the number of odd cycles by 2.

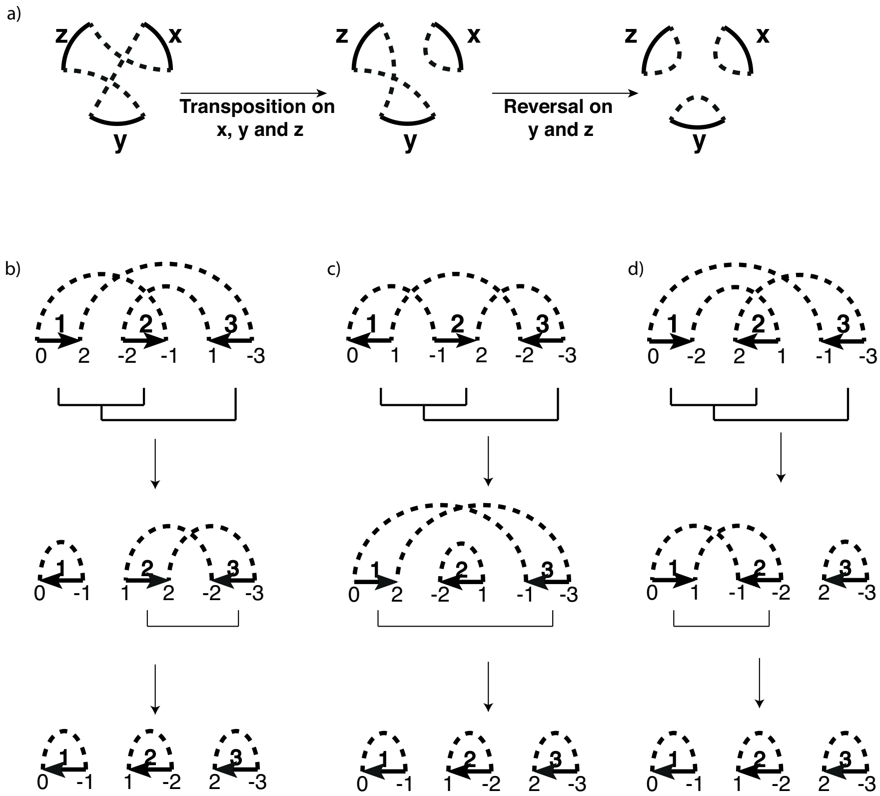

Figure 5: In (a) we show a sequence of two operations (one transposition followed by one reversal) applied to a generic configuration. In (b) we show the cycle graph for the permutation $\pi = (-2\ 1)$ which is in the class, and we map the sorting sequence for the class into a sorting sequence for π. In (c) and (d) we do the same for two other permutation: $(-1, -2)$ and $(2, -1)$.

Consider a triple of black edges x, y, and z belonging to the same cycle c in $G(\pi)$. Reading c induces a cyclic order on x, y, z, and, among three possible representations of this order, we choose as canonical the one such that x is the rightmost black edge among the three edges. We hereafter assume that triples of black edges will be in the canonical order.

We define the distance between any two black edges b_1 and b_2 in the same cycle as the number of gray edges between b_1 and b_2 when traversing c in the canonical order, denoted as $dist(b_1, b_2)$.

Lemma 8. *Let x, y, z, be a triple (in the canonical order) of black edges in c.*

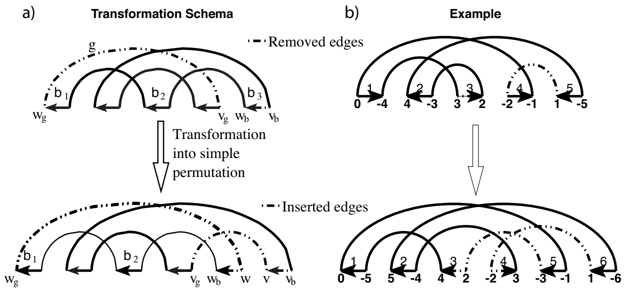

Figure 6: Procedure to transform a permutation into a simple permutation. We show in (a) the procedure itself and in (b) an example using the input permutation $\pi = (+4\ +3\ -2\ +1)$, which outputs $\hat{\pi} = (+5\ +4\ -2\ -3\ +1)$.

If (i) x, y, z have the same sign in c, (ii) $y < z < x$, and (iii) at least two distances in the set $\{dist(x, y), dist(y, z), dist(z, x)\}$ are odd, then the transposition $\rho_t(y, z, x)$ increases by 2 the number of odd cycles.

Proof. A transposition $\rho_t(y, z, x)$ replaces c with three new cycles c_1, c_2 and c_3 whose sizes are $dist(x, y)$, $dist(y, z)$ and $dist(z, x)$, respectively. Two of the new cycles will be odd if c is even. Otherwise, three odd cycles will be created. In any case, the number of odd cycles will be increased by 2. □

Lemma 9. *Let x,y be two black edges in an even cycle $c = (\ldots, -x, \ldots, y, \ldots)$ such that $dist(x, y)$ is odd. The reversal $\rho_r(x, y-1)$ increases by 2 the number of odd cycles if $0 < x < y$. Otherwise, if $0 < y < x$, the reversal $\rho_r(y, x-1)$ increases by 2 the number of odd cycles.*

Proof. Since x and y are divergent black edges, the reversal that breaks both edges will split the cycle in two. One of the cycles will have size $dist(x, y)$, which is odd as stated by the lemma itself. Since c is even, then the other new cycle with the remaining edges must be odd. So the reversal will remove from $G(\pi)$ an even cycle and it will add two odd cycles to it. □

Lemma 10. *Let x, y be two black edges in the even cycle $c = (\ldots, x, \ldots, y, \ldots)$ such that $dist(x, y)$ is odd and let z be a black edge in another even cycle d. The transposition that acts on x, y and z increases by 2 the number of odd cycles.*

Proof. The transposition will remove the cycles c, and d from $G(\pi)$ and it will add two new cycles c' and d' such that c' is smaller than c by $dist(x, y)$ black edges and d' is greater than d by $dist(x, y)$ black edges. Since c and d are even and $dist(x, y)$ is odd, then c' and d' are odd. Therefore, this transposition increases by 2 the number of odd cycles. □

Our algorithm starts by applying several steps supported by Lemmas 8, 9, and 10. Step i is executed only if steps 1 to $i-1$ are not possible for the current configuration. Algorithm 1 presents the pseudocode. The algorithm stops in two situations: (i) if none of the steps can be applied or (ii) if we succeed in sorting the input permutation.

Algorithm 1: BasicAlgorithm(π)

```
1  distance ← 0
2  flag ← false
3  while π ≠ ι and not flag do
4      flag ← true
5      i ← 1
6      while i ≤ 6 and flag do
           // Line below uses six steps explained in Section 3
7          candidates ← find_candidates(π, Step i)
8          if length(candidates) ≥ 1 then
9              ρ ←select_candidate(candidates)
10             π ← π · ρ
11             distance ← distance+1
12             flag ← false
13         i ← i + 1
14 return π, distance
```

We now detail the steps that were mentioned in Algorithm 1, Line 6.

Step 1: Lemma 8 explains how to break one cycle c of a given size s to create three new cycles whose sizes are a, b and c such that $a+b+c=s$. If c has three black edges, then the three new cycles will have only one black edge each.

Step 2: Lemma 9 explains how to break one even cycle c of a given size s to create two odd cycles whose sizes are a and b such that $a+b=s$. If c has two black edges, the two new cycles will have only one black edge each.

Step 3: Lemma 8 explains the characteristics that a triple of black edges x, y, z in a cycle c of size s should have in order to find a transposition that increases by two the number of odd cycles. One of the characteristics states that at least two distances in the set $\{dist(x,y),\ dist(y,z),\ dist(z,x)\}$ are odd. If we further constrain this property to guarantee that these two odd distances are in fact equal to 1, we guarantee that we will generate three new cycles whose sizes are 1, 1, and $s-2$. Therefore, this step searches for cycles of any size such that two distances in the set $\{dist(x,y)$, $dist(y,z)$, $dist(z,x)\}$ are exactly one.

Step 4: This step looks for transpositions that satisfy Lemma 8, which are those transpositions that increase by two the number of odd cycles while also increasing by two the overall number of cycles. Every triple x, y, z that satisfies Lemma 8 is a candidate for this step and one of the characteristics states that at least two distances in the set $\{dist(x,y)$, $dist(y,z)$, $dist(z,x)\}$ are odd. Therefore, this step searches for cycles of any size such that two distances in the set $\{dist(x,y)$, $dist(y,z)$, $dist(z,x)\}$ are exactly odd.

Step 5: This step looks for reversals that satisfy Lemma 9, which are those reversals that increase by two the number of odd cycles while increasing by one the number of cycles. Therefore, this step searches for an even cycle with divergent edges, and if that can be found, we apply a reversal that increases by two the number of odd cycles.

Step 6: This step looks for transpositions that satisfy Lemma 10, which are those transpositions that transform two even cycles into two odd cycles. This step is important, because it guarantees that after applying the basic algorithm the resulting permutation has at most one even cycle. In addition, this even cycle has no divergent edges; otherwise, the previous step would have had a candidate. Therefore, this step searches for two even cycles, and if that can be found, we apply a transposition that increases by two the number of odd cycles while keeping the overall number of cycles unchanged.

Algorithm 1 runs in $O(n^4)$ time. The worst case scenario for Steps 1, 3 and 4 occurs if the cycle graph has only one cycle. In that case, identifying the black edges x, y, z that satisfy Lemma 8 may take $O(n^3)$ time. Step 2 runs in $O(n)$ time, since we just need two edges with different orientations. Steps 5 and 6 run in $O(n)$ time if any reversal that satisfies Lemma 9 and any transposition that satisfies Lemma 10 are allowed. However, both steps may take $O(n^3)$ time, if one decides to list all such operations to filter the best one. Since we need at most $O(n)$ operations to sort the input permutation, the stated complexity follows.

These six steps are sufficient to optimally sort many random permutations. Unfortunately, some permutations have cycle graphs where none of the six steps can be applied. In these cases, Algorithm 1 will finish before reaching the identity permutation.

To deal with these cases, we generated a database with enough configurations to guarantee that we can find a sequence of operations in the database in any situation such that Algorithm 1 stops prematurely. These operations bring us close to the identity permutation, and Algorithm 1 can be executed again after applying them. This behavior is described in Algorithm 2. We provide all the details regarding how the database is created and used in Section 4.

Algorithm 2: CompleteAlgorithm(π)

```
distance ← 0
while π ≠ ι do
    σ, opCount← BasicAlgorithm(π)
    if opCount> 0 then
        π ← σ
        distance ← distance+opCount
    if π ≠ ι then
        sequence ← DatabaseSearch(π) // Algorithm 3
        π ← applyOperations(sequence)
        distance ←distance+length(sequence)
return distance
```

Algorithm 2 makes a call to Algorithm 1. This call may be followed by a database search that executes in $O(n)$ time. Therefore, Algorithm 2 also runs in $O(n^4)$ time.

A better explanation for the database search is given in Section 4.3 and the pseudocode is presented in Algorithm 3.

4 The Database

Algorithm 1 provides a rearrangement sequence for many permutations, but it has no answer for permutations π whose cycle graphs have all of the following properties: (i) there is no triple of the form required by Lemma 8, otherwise Step 1 or Step 4 would be applied; (ii) no black edge inside an even cycle is traversed from left to right, otherwise Step 2 or Step 5 would be applied; (iii) there is at most one even cycle, otherwise Step 3 or Step 6 would be applied.

Our first step to deal with this issue is to transform the cycle graph $G(\pi)$ into a simple cycle graph $G(\hat{\pi})$. The transformation maintains the three properties as shown by Lemma 11 with the addition of a fourth property that states that $G(\hat{\pi})$ has only short cycles.

Lemma 11. *The operation that transforms a permutation π into a simple permutation $\hat{\pi}$ maintains the properties (i), (ii), and (iii).*

Proof. We prove that $\hat{\pi}$ maintains each property as follows.

- Property (i) asserts that $G(\pi)$ has no triple of the form required by Lemma 8. Let b_3 be the black edge that the transformation will break, b_2 be a black edge linked to b_3, and b_1 be the black edge linked to b_2 that is not b_3. Two new cycles will be created such that one of them has the form (b'_3, b_2, b_1). If this cycle has the triple required by Lemma 8, than the triple (b_3, b_2, b_1) in $G(\pi)$ would have the same property because b'_3 is placed in $G(\hat{\pi})$ in the same position b_3 was in $G(\pi)$ relative to b_2 and b_1.

  In regard to the second cycle, let $(a_1, a_2, \ldots, a_m)$ be the black edges in the cycle being broken in $G(\pi)$ such that $a_i \notin \{b_1, b_2, b_3\}$ for $1 \leq i \leq m$. Two black edges b_1, b_2 are removed from the cycle and the black edge b''_3 replaces b_3 in the same reading order and in the same position. Therefore, if $dist(a_i, a_j)$, was odd (even) before the transformation, it should be odd (even) after the transformation, so there should be no triple (a_i, a_j, a_k), $1 \leq i, j, k \leq m$, as required by Lemma 8 unless it existed before the transformation. In addition, if $dist(a_i, b_3)$, $1 \leq i \leq m$, was odd (even) before the transformation, $dist(a_i, b''_3)$ should be odd (even) after the transformation, and since b''_3 replaces b_3, there should be no triple (a_i, a_j, b''_3) as required by Lemma 8 in $G(\hat{\pi})$ unless (a_i, a_j, b_3) had the required property in $G(\pi)$.

- Property (ii) asserts that no black edge inside an even cycle is traversed from left to right in $G(\pi)$. This property is maintained because the transformation breaks one black edge b_3 and creates two black edges b'_3 and b''_3 in different cycles. Let b_2 and b_4 be the black edges linked to b_3 by gray edges in $G(\pi)$ and let us assume that b_2 goes to the same cycle as b'_3, and b_4 goes to the same cycle as b''_3 in $G(\hat{\pi})$. Since b_2 and b_3 are convergent in $G(\pi)$, b_2 and b'_3 are convergent in $G(\hat{\pi})$. Analogously, since b_3 and b_4 are convergent in $G(\pi)$, b''_3 and b_4 are convergent in $G(\hat{\pi})$. As in $G(\pi)$ every black edge is traversed from right to left, this situation cannot be changed by the transformation into a simple permutation.

- Property (iii) asserts that $G(\pi)$ has at most one even cycle. This property is maintained because each transformation breaks a cycle of size l and creates two new cycles such that one of them has size 3 and the other has size $l - 2$. Therefore the number of even cycles in $G(\hat{\pi})$ is the same as the number of even cycles in $G(\pi)$.

$\square$

Until the end of this section, we assume that every permutation we consider complies with the four properties stated for $G(\hat{\pi})$.

The key point is how we build a database that contains a sequence of operations for all possible permutations of this form. We address this issue in Section 4.1 by describing a procedure to build the database. Section 4.2 details the outcome of using this procedure. In Section 4.3, we show how we use the database to sort any input permutation.

4.1 Building the Database

Only a limited number of cycles may compose $G(\hat{\pi})$ to satisfy all the four constraints: it may have at most one even cycle of the form $(2, 1)$ and one or more cycles in the set $\{(3, 2, 1), (3, -2, 1), (3, -1, 2), (3, 2, -1), (3, 1, -2), (3, -2, -1), (3, -1, -2)\}$. The cycles in the set $\{(3, -2, -1), (3, 1, -2), (3, -1, 2)\}$ have features in common because one can be found from the others by rotation as shown in Figure 4. Therefore, these cycles are in the same equivalence class. The same happens with the set $\{(3, -1, -2), (3, -2, 1), (3, 2, -1)\}$.

Previous sections stated that we will select several configurations and include them in a database with sequences of operations that sort them. Some considerations about the approximation factors of these sequences need to be made. Let $s = \langle\rho_1, \rho_2, \ldots, \rho_d\rangle$ be a sequence of operations such that $\pi \cdot \rho_1 \cdot \rho_2 \cdot \ldots \cdot \rho_d = \sigma$ and each ρ_i can be either a reversal or a transposition. Note that the best we can do is to create $2d$ odd cycles, since we are using d operations from π to σ. Therefore, the approximation for this sequence is $\frac{2d}{c_{odd}(\sigma) - c_{odd}(\pi)}$. Observe that we do not enforce $\sigma = \iota$.

Since we made no restriction on the number of cycles, there is an infinite number of cycle graphs. Although our database cannot store all of them, it should have enough information to build a sequence of operations for any of them. We accomplish this task by storing connected configurations instead of cycle graphs. Remember that configurations were defined as subsets of cycle graphs. Therefore, if we have a configuration C and we know a sequence of operations that can be applied on C, then we can adapt this sequence of operations so we can use it on any cycle graph $G(\pi)$ such that $C \subset G(\pi)$. Thus, we need only to guarantee that for each cycle graph that is not in the database we have at least one configuration that is a subset of the cycle graph. In this case, a sequence of operations for the cycle graph can be computed from the sequence of operations for the configuration.

We start an enumeration process with the smallest configurations possible, which are configurations having a single cycle, and we try to find sequences of operations that guarantee a given approximation bound using a branch-and-bound algorithm. If a sequence of operations exists for a configuration C, we do not need to consider any other configuration of cycle graph C' such that $C \subset C'$. On the other hand, if the sequence of operations does not exist for C, then we expand our analysis by considering in the next iteration configurations C' one cycle greater than C such that $C \subset C'$.

An *extension* is the process of creating a configuration C' from a configuration C by adding one cycle c to it. This process is done by inserting the black edges from c somewhere in the configuration C such that at least one gray edge of c

intersects with at least one gray edge that is already in C. This restriction is important to force C' to be always a connected configuration.

Observe that inserting c in C will destroy complement edges in $\bar{C}$ and create new ones. Therefore, this process may transform incomplete configurations into components and vice versa.

If it is possible to create a configuration B by adding one or more cycles to a configuration A, then we say that B is extended from A. Let A be a not complete configuration and let B be a component extended from A. Since $\bar{B}$ has a single Hamiltonian cycle and $\bar{A}$ has several cycles $c_1, c_2, \ldots, c_k$, $k > 1$, we know that each of these cycles should be joined by the extensions to reach B. Otherwise, there is an incomplete configuration B' such that $A \subset B' \subset B$.

From the discussion above it follows that there are two types of extensions that are sufficient for building any component, which are (i) extensions of incomplete configurations that join together cycles in the complement graph and (ii) extensions of components that have no further restriction apart from the previous one that stated that gray edges should cross in the extension process to build connected configurations.

Let us assume we have an incomplete configuration A, and we want to extend it; we select the smaller cycle c_i in $\bar{A}$ and consider all the cases where at least one complement edge of c_i and at least one complement edge outside c_i are affected. If A is a complete configuration, we consider every position between two black edges when adding a cycle to A.

Figure 7 provides an example. The initial configuration is $C = \{(3, -2, 1)\}$ and we will use the cycle $c = (3, -2, -1)$ to extend C. First, we compute the complement graph $\bar{C}$ and since $\bar{C}$ has two cycles, we consider the smaller cycle, which has only one complement edge (a, b). When we add c to the configuration C, we guarantee that one or two black edges from c will be added inside the arc created by a and b and that one or two black edges from c will be added outside this same arc. That leads to 5 extended configurations as shown in Figure 7. After that, we filter the configurations that are in the same equivalence class and compute one representative for each class. In the example shown in Figure 7, the configurations $\{(6, -4, 3), (5, -2, -1)\}$ and $\{(6, -5, 3), (4, -2, -1)\}$ are equivalent.

Assume, for instance, that we reach a situation where all the extended configurations have sorting sequences with the desired approximation bound. We do not need to further extend any configuration and the algorithm stops.

4.2 Individual Databases

We did not try to create one large database with all configurations. Instead of that, we decided to create a set of small databases with particular features. They

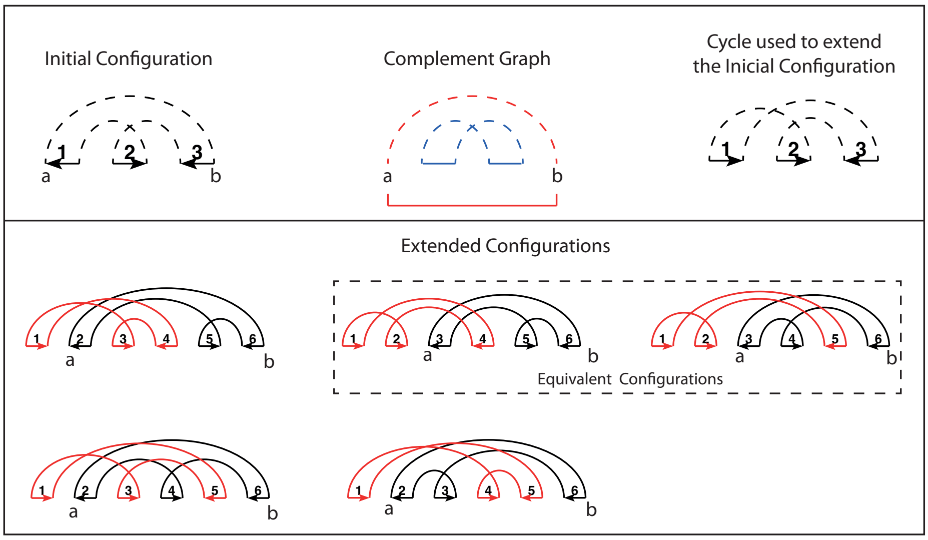

Figure 7: Extension process. We compute the complement graph, and we break only the smallest cycle in it (shown in red). The extension generates 5 configurations, and two of them are equivalent.

differ from each other by the initial configurations placed in the database and the set of cycles that were used to extend configurations in each iterative step.

Convergent cycle (1.375-Database): In this database, there is only one initial configuration $C = \{(3, 2, 1)\}$ and only one cycle $c = (3, 2, 1)$ used for extensions. Therefore, this database contains configurations that are a set of one or more convergent cycles. Every component in this database can be computed from signed permutations with no negative signs. Thus, we can sort these permutations using only transpositions, similarly to Elias and Hartman [Elias and Hartman 2006], who developed an algorithm using a method which is also based on creating databases. Therefore, we simply converted their database to the format we need.

Table 1 shows that this database has a total number of 4,002,165 configurations having a sequence of operations with 1.375 approximation factor, and the number of cycles in those configurations ranges from 3 to 13.

The intuition to build this database is that it comprises the instances to the well studied Sorting by Transposition Problem and these instances have been sorted with operations that are valid to our problems (transpositions). Therefore, we can simply use the sorting sequences and treat these instances as a particular case of our problem.

Even cycle (1.667-Database): In this database, there is only one initial configuration $C = \{(2, 1)\}$, which has one even cycle. In addition we use the following cycles during the extensions: $\{(3, 2, 1)$, $(3, -2, 1)$, $(3, -1, 2)$, $(3, 2, -1)$, $(3, 1, -2)$, $(3, -2, -1)$, $(3, -1, -2)\}$. This database is the only database that has an even cycle. We computed a sequence for every configuration in this database and it is guaranteed that a sequence with approximation factor up to 1.667 can be found for any cycle graph having an even cycle.

Table 1 shows that this database has a total number of 1,806 configurations having sequences of operations, and the number of cycles in those configurations ranges from 2 to 4. Since all 4 cycle configurations found in the process have a sequence of operations with approximation factor up to 1.667, we did not extend this database to 5 cycle configurations.

The intuition to build this database is the fact that Step 6 in Algorithm 1 needs two even cycles. Since we already start with one even cycle, we just need to create another one, which can be achieved by reverting an odd divergent cycle. This database stores the optimum way of doing this in various situations.

Odd cycles (1.8-Database): We consider as initial, the configurations that have divergent cycles: $\{(3, -2, 1)$, $(3, -1, 2)$, $(3, 2, -1)$, $(3, 1, -2)$, $(3, -2, -1)$, $(3, -1, -2)\}$. These cycles plus the cycle $(3, 2, 1)$ are also used for extensions. In this database, we did not allow configurations with more than 4 cycles, and we set our branch-and-bound algorithm to look for 1.8-sequences. We found some configurations that had no 1.8-sequences, so we decided not to include them in the database.

Table 1 shows that this database has a total number of 443,748 configurations with 1.8-sequences, and the number of cycles in those configurations ranges from 2 to 4. If we try to extend this database to 5 cycle configurations, the number of instances will increase to more than 37 million, which is prohibitive. In this case, we decided to limit the number of extensions to generate configurations with at most 4 cycles.

The intuition to build this database is that in most situations the 1.8 approximation factor is plausible if we consider a small set of cycles that form a configuration. Therefore, we filter these configuration and guarantee that 2.0-approximation steps will be only used in a very limited amount of cases.

Odd cycles (2.0-Database): This database has the same initial configurations and extension set as the previous one. Here, however, we set our branch-and-bound algorithm to look for 2.0-sequences. The one cycle configuration

$\{(3,-2,-1)\}$ has a sequence of two operations $\langle(1,2),(1,1)\rangle$ with an approximation factor of 2.0, which drastically reduces the number of configurations in comparison with the 1.8-Database. Note that $(3,-2,-1)$, $(3,-1,2)$ and $(3,1,-2)$ are in the same equivalence class. It was possible to find a 2.0-approximation sequence for every cycle graph, which leads to the theoretical approximation bound of the entire algorithm.

Table 1 shows that this database has a total number of 257 configurations with 2.0 approximation ratio sequences, and the number of cycles in those configurations ranges from 1 to 3. Since all the 3 cycle configurations have a sequence of operations, the database is complete.

Database	Number of cycles	Number of configurations	Total
1.375-Database	3	1	4,002,165
	4	22	
	5	5,696	
	6	53,898	
	7	377,877	
	8	1,450,662	
	9	1,077,521	
	10	1,034,940	
	11	1,140	
	12	264	
	13	144	
1.667-Database	2	2	1,806
	3	161	
	4	1,643	
1.8-Database	2	6	443,748
	3	1,468	
	4	442,274	
2.0-Database	1	1	257
	2	6	
	3	250	

Table 1: Number of configurations with a valid sequence for each database.

4.3 Using Database Search to Sort Any Permutation

Algorithm 1 presented in Section 3 does not provide a sequence to sort every input permutation. In this case, Algorithm 2 shows that the database search is a final step that guarantees that any permutation will be sorted. Let π be the (signed or unsigned) permutation that cannot be sorted by previous steps; we first compute the cycle graph $G(\pi)$. The cycle graph may have several components, but since our database was built based on connected configurations, we simply select the first component and start our database search.

Algorithm 3 describes the procedure that queries the database. The algorithm receives a permutation as input, and computes a cycle graph which can be built either for signed or unsigned permutations.

We first try to use the 1.375-Database by selecting the convergent cycles that create connected configurations from the input cycle graph. If we can find the configuration in the database, we immediately return the sequence of operations. This step is shown in Algorithm 3, lines 3–8.

Since we have several databases, it may occur that we need to search for configurations in all of them based on some arbitrary order. We decided to search in databases that guarantee smaller approximation ratios first. In the worst case scenario, we will find a configuration in the 2.0-Database. Therefore, the complete algorithm has a 2.0 theoretical approximation bound.

5 Experimental Results

We have implemented our algorithm, and we ran it on three sets of permutations.

Constant number of mutations and variable permutation size:
This experiment uses permutations of size n ranging from 10 to 100. For each permutation, we apply 20 operations (10 reversals and 10 transpositions). Section 5.1 discuss the results obtained from this experiment.

Fixed mutation rate and variable permutation size:
This experiment uses permutations of size n ranging from 10 to 100 with the rate of rearrangement operations fixed on $0.2 \times n$. Section 5.2 discuss the results obtained from this experiment.

Fixed permutation size and variable mutation rate:
This experiment uses permutations of size 100 and we made the rate of genome rearrangements in the range 10 to 100. Section 5.3 discuss the results obtained from this experiment.

In Section 5.4, we considered the instances from the three experiments and computed the expected approximation ratio of our algorithm.

Algorithm 3: DatabaseSearch(π)

```
G(π) ← CycleGraphDecomposition(π)
Component ← GetFirstComponent(G(π))
Configuration ← ∅
Cycle ← Component.GetNextConvergentCycle()
while Cycle ≠ ∅ do
    Configuration ← Configuration ∪ Cycle
    Sequence ← QueryDatabase(Configuration, 1.375-Database)
    if Sequence ≠ ∅ then
        return Sequence
    Cycle ← Component.GetNextConvergentCycle()
Configuration ← ∅
Cycle ← Component.GetNextEvenCycle()
while Cycle ≠ ∅ do
    Configuration ← Configuration ∪ Cycle
    Sequence ← QueryDatabase(Configuration, 1.667-Database)
    if Sequence ≠ ∅ then
        return Sequence
    Cycle ← Component.GetNextCycle()
Configuration ← ∅
Cycle ← Component.GetNextDivergentCycle()
count ← 0
/* We fixed the number of extensions for the 1.8-Database   */
while Cycle ≠ ∅ and count < 5 do
    Configuration ← Configuration ∪ Cycle
    Sequence ← QueryDatabase(Configuration, 1.8-Database)
    if Sequence ≠ ∅ then
        return Sequence
    count ← count + 1
    Cycle ← Component.GetNextCycle()
Configuration ← ∅
Cycle ← Component.GetNextDivergentCycle()
while Cycle ≠ ∅ do
    Configuration ← Configuration ∪ Cycle
    Sequence ← QueryDatabase(Configuration, 2.0-Database)
    if Sequence ≠ ∅ then
        return Sequence
    Cycle ← Component.GetNextCycle()
```

The main goal of our analysis is to compare our algorithm against other methods that provide valid sequences to sort the input permutation. First, we have implemented the algorithms presented by Walter, Dias and Meidanis [Walter et al. 1998], and we refer to these algorithms as **WDM** on discussion sections. The authors presented two algorithms; one is intended for signed permutations, guarantees an approximation factor of 2 and has $O(n^3)$ time complexity; the other is intended for unsigned permutations with an approximation factor of 3 and has $O(n^2)$ time complexity.

In 2008, Rahman, Shatabda and Hasan [Rahman et al. 2008] presented a $2k$-approximation algorithm for unsigned permutations, where k is the approximation ratio of the algorithm used for cycle decomposition [Caprara 1999]. We adapted this algorithm to run on signed permutations by removing the cycle decomposition step. Instead, we use the unique cycle graph for the input permutation as described in Section 2. Therefore, for signed permutations, the approximation factor for the algorithm is 2, and it has $O(n^2)$ time complexity. We refer to both the original and the adapted versions as **RSH**.

In 2014, Dias, Galvão, Lintzmayer, and Dias [Dias et al. 2014a] presented a greedy algorithm for the sorting by reversals and transpositions problem on unsigned permutations. The algorithm uses the notion of breakpoints, which are elements that are side by side in the input permutation but not in the identity permutation. We were able to apply the same ideas on signed permutations by adapting the notion of breakpoints. We refer to both versions of this algorithm as **DGLD**. In both cases, the algorithm runs in $O(n^2)$ and guarantees the approximation factor 3.

We will denote as **REV** the optimum solution for the sorting by reversals problem on signed permutations [Hannenhalli and Pevzner 1999]. Since our problem accepts both transpositions and reversals, a sequence that applies only reversals is a valid solution. We used the implementation provided by the program `GRIMM` [Tesler 2002], which runs in $O(n)$ time. It is possible to prove that **REV** is a 3-approximation for our problem, since the effect of any transposition can be reproduced by 3 reversals.

To run **REV** on unsigned permutations, we first use the cycle graph decomposition algorithm developed by Lin and Jiang [Lin and Jiang 2004], which has an approximation factor of $1.4193 + \epsilon$, for any positive ϵ. After that, we obtain the signed permutation associated with that cycle graph, and we optimally sort this signed permutation. The post-processing step requires us to remove reversals $\rho(i, j)$ such that $i = j$.

The same way we are using the sorting by reversals problem to obtain valid solutions, we can use approximation algorithms for the sorting by transpositions problem. For the unsigned case, we could use any of the approximation algorithms developed for the sorting by transpositions problem, since transpositions do not change signs. However, for signed permutations π, a pre-processing step is mandatory. Thus, we first identify all maximal subsequences of negative elements, and we reverse each one to find the permutation π' whose elements are positive. After that, we can sort π' as if it were unsigned.

The sorting by transpositions problem has several approximation algorithms. We decided to use the algorithm devised by Dias and Dias [Dias and Dias 2013], because it provides the best results in a practical analysis [Dias et al. 2014a], even knowing its approximation guarantee is 1.5 against the 1.375-approximation algorithm presented by Elias and Hartman [Elias and Hartman 2006]. The Dias and Dias algorithm runs in $O(n^5)$ time. The heuristic that reverses maximal subsequences of negative elements and later uses a sequence of transpositions will be called **TRANS**.

Sections 5.1, 5.2, and 5.3 show the results on different simulated experiments designed to evaluate the algorithms.

5.1 Constant number of mutations and variable permutation size

In this experiment, we used a set composed of 190,000 signed and 190,000 unsigned permutations. Permutation sizes range from 10 to 100 in intervals of 5, with 10,000 permutations of each size. The permutations were added to the data set as follows: starting from the identity permutation, we apply 10 reversals and 10 transpositions at random. Therefore, we know that 20 is an upper bound for the distance of the permutations in our set, and we can use the upper bound for comparison purposes.

Table 2 and Table 3 show the average distance for signed and unsigned permutations, respectively. In both cases, our algorithm returns the best results. If we take into account that 20 operations (10 reversals and 10 transpositions) were used to create the permutations, our results are very good, since our algorithm usually finds sequences with fewer operations. Our average is 16.82 operations when we consider the 190,000 signed permutations, and 16.20 when we consider the 190,000 unsigned ones. **DGLD** is the second best algorithm with averages of 18.19 and 16.74 operations on signed and unsigned permutations, respectively. **RSH** also can return sequences with fewer operations than the 20 used to generate the scenario, the averages are 19.69 operations for signed permutations and 18.79 operations for unsigned permutations. Other algorithms return more than 20 operations on average.

Further comparing our algorithm against the other algorithms for the sorting by reversals and transpositions problem, we observed that our sequences have on average 10% fewer operations than **RSH** and **WDM**, no matter the permutation size. Besides, for permutations having 20 or more elements, our algorithm returned sequences that are about 30% smaller than **WDM**.

n	10	20	30	40	50	60	70	80	90	100
DGLD	6.90	12.59	16.17	18.34	19.66	20.50	20.94	21.26	21.52	21.68
WDM	8.38	16.51	22.00	25.25	27.00	27.94	28.45	28.77	28.98	29.13
RSH	7.26	13.89	18.24	20.62	21.71	22.19	22.37	22.49	22.58	22.66
REV	8.76	16.95	22.50	25.75	27.51	28.48	28.97	29.29	29.49	29.62
TRANS	7.38	13.81	18.32	21.36	23.54	25.11	26.29	27.13	27.93	28.55
OUR	**6.37**	**11.48**	**14.85**	**16.91**	**18.13**	**18.91**	**19.39**	**19.75**	**20.01**	**20.23**

Table 2: Average distance for signed permutations. For conciseness, we are not showing results for permutations with size in range {15,25,. . . ,95}. Bold numbers highlight the best results.

n	10	20	30	40	50	60	70	80	90	100
DGLD	**4.48**	**9.45**	**13.26**	16.00	17.93	19.31	20.33	21.06	21.55	21.95
WDM	5.31	12.73	18.71	23.06	26.24	28.62	30.44	31.89	32.92	33.85
RSH	4.98	10.61	15.21	18.55	20.69	22.09	22.91	23.32	23.46	23.43
REV	5.76	12.49	17.97	21.93	24.62	26.46	27.65	28.40	28.83	29.15
TRANS	4.66	9.49	13.76	17.63	21.22	24.76	28.14	31.56	34.92	38.36
OUR	4.74	9.66	13.28	**15.79**	**17.50**	**18.67**	**19.45**	**20.00**	**20.30**	**20.54**

Table 3: Average distance for unsigned permutations. For conciseness, we are not showing results for permutations with size in range {15,25,. . . ,95}. Bold numbers highlight the best results.

When we consider the algorithms **REV** and **TRANS** in the analysis, we conclude that **TRANS** performed well on small permutations. Observe that the cause for **TRANS** being better than **REV** on small permutations is not really the size of the permutation. The real reason is that we are always using 10 reversals and 10 transpositions to generate permutations, so small permutations have the highest mutation rate. In this case, pairs of elements that are side by side in permutations of size 10 are likely not side by side in the identity permutation. Transpositions act on at most three pairs whereas reversals acts on at most two, which explains why **TRANS** are more appropriate in these cases.

For the unsigned case, our approach was the best for permutations longer than 30 elements. For small permutations having the highest mutation rate, our results have 6% more operations than **DGLD** and **TRANS**. Observe that the good performance of **TRANS** happens on small permutations only, which is expected because they are more appropriate in these cases (high mutation rate) as already discussed.

Another aspect we analyzed in our experiments is the approximation ratio (average and maximum). We used the lower bound presented in Theorem 6. On signed permutations, our algorithm always obtained the lowest average approximation ratio, and it was never higher than 1.4 as shown in Figure 8. Besides, on permutations having more than 50 elements, the approximation ratio was lower than 1.2. **DGLD** also presented a good performance on average with an approximation factor that was about 10% greater than ours. The algorithms **REV** and **WDM** presented average approximations higher than 1.6.

For each signed permutation, we recorded the algorithms that compute the smallest sequence of operations. This information is used to find how often each algorithm returns the best result, which is shown in Figure 8. Our algorithm outperforms the others in this aspect, since we can find the smallest sequence of operations in about 80% to 90% of the cases. **REV**, **TRANS**, and **WDM** almost never find a solution better than the other algorithms.

The same analysis for unsigned permutations shows that our algorithm outperforms the others again. On average, our algorithm returns the best approximation factors as shown in Figure 9, which were always between 1.5 and 1.6. **DGLD** also obtained good approximation factors on average. This is particularly true for permutation having up to 30 elements, where they performed better than any other algorithm. **RSH** presented approximations between 1.5 and 1.9.

Figure 9 also shows the number of times each algorithm return the smallest sequence of operation. Our algorithm does this in more than 80% of the cases for permutations with more than 30 elements. **TRANS** and **DGLD** performed best for small permutations that have the highest mutation rate, but only the latter shows good results for longer permutations. **RSH**, **WDM**, and **REV** did not perform well in this aspect.

5.2 Fixed mutation rate and variable permutation size

In the previous experiment, we kept the number of mutations constant and changed the size of the permutations. Therefore, small permutations are much more scrambled than longer permutations because the rate of mutations is higher. This experiment changes the size of the permutations but keeps the mutation rate fixed. Therefore, we have a better measure of how the size impacts on algorithm performance.

Let n be the size of the permutation, we designed n to range from 10 to 100 with the rate of rearrangement operations fixed on 0.2. Therefore, if the permutation has size 10 we apply one reversal and one transposition. If the permutation has size 20 we apply two reversals and two transpositions and so on.

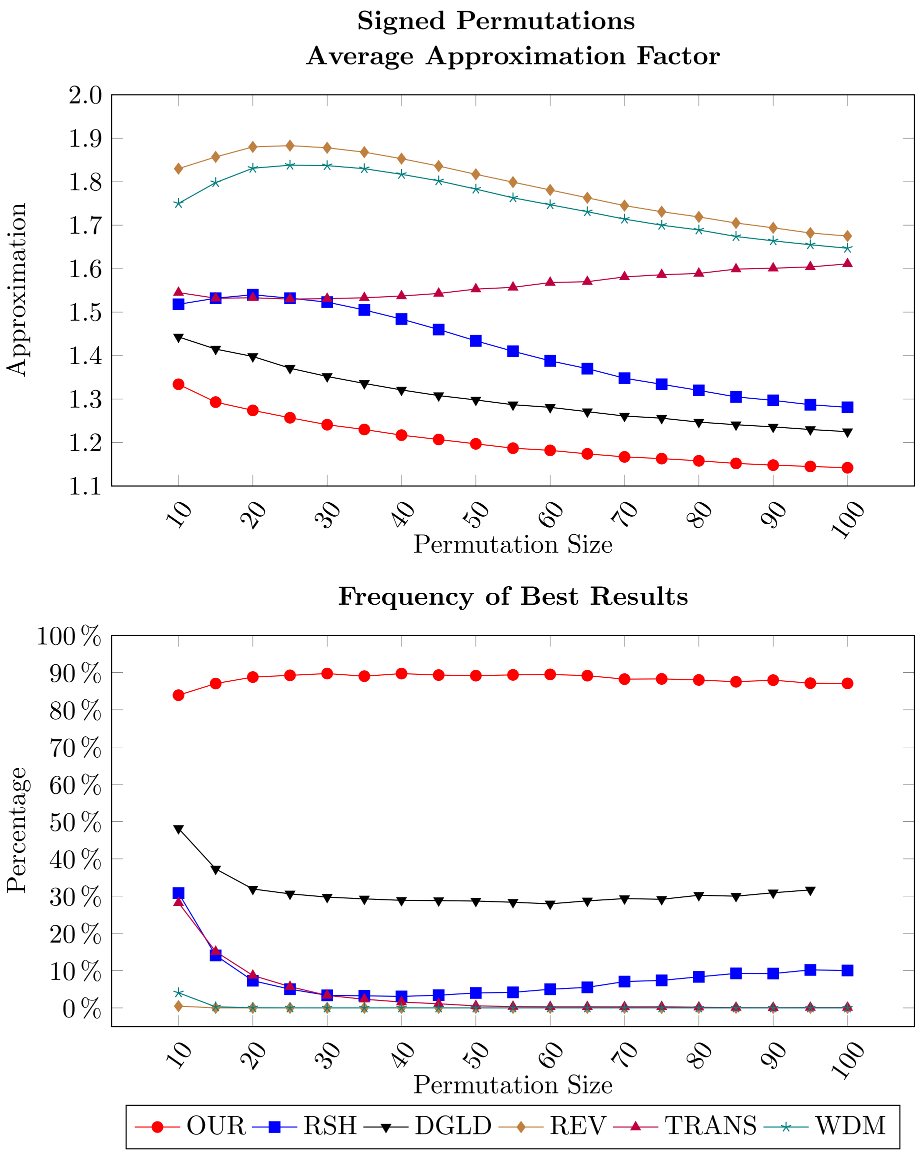

Figure 8: Average approximation factor on signed permutations and percentage of signed permutations in which each algorithm found the best result.

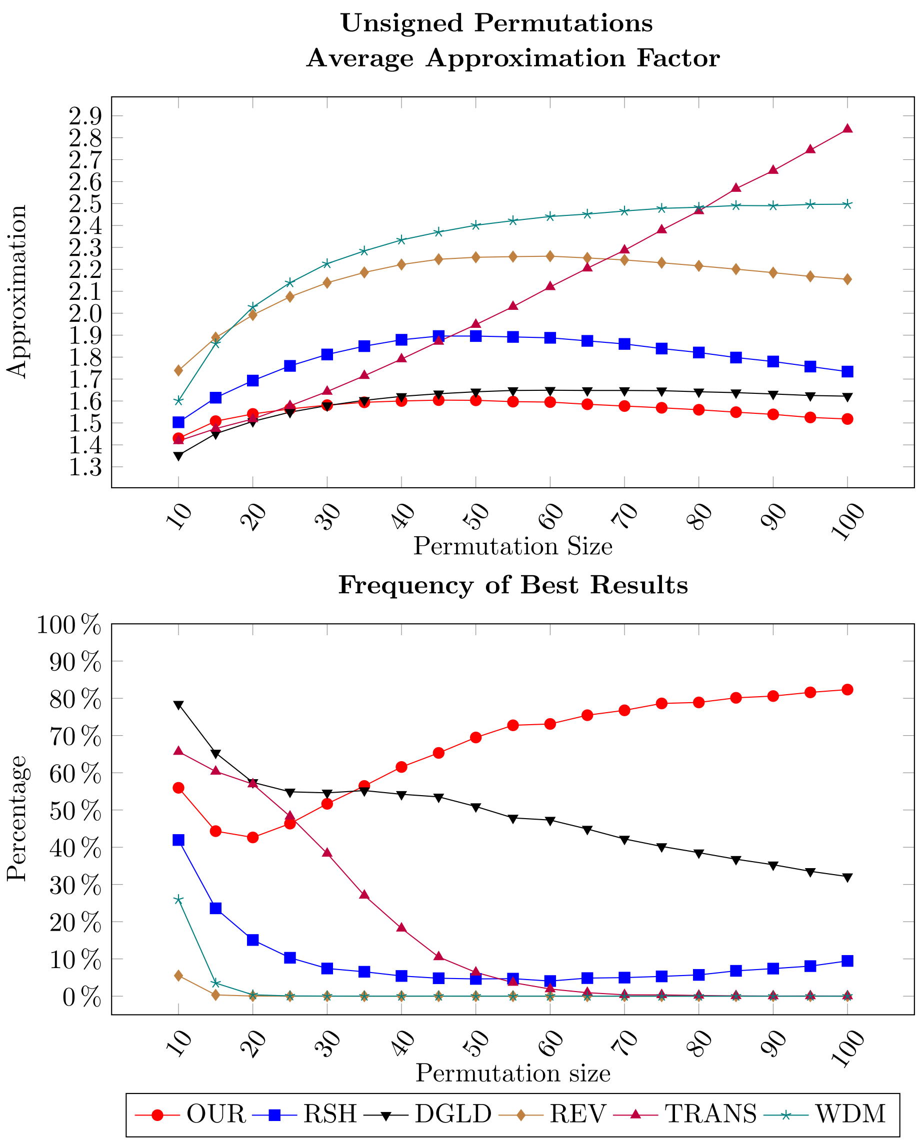

Figure 9: Average approximation factor for unsigned permutations and percentage of unsigned permutations in which each algorithm found the best result.

Figures 10 and 11 show how the algorithms behaves on unsigned and signed permutations, respectively. We observe few changes, which gives us a clear conclusion. Our algorithm outperforms the others and returns the best results in most of the cases. Our average approximation ratio converged to a number lower than 1.6 on unsigned permutations and to a number lower than 1.2 on signed permutations.

5.3 Fixed permutation size and variable mutation rate

This last experiment was designed to measure how the mutation rate impacts algorithm results. This information complements what we have discussed in previous experiments and isolates the likely cause of variation in the quality of the solution provided by each algorithm.

We created a set of random permutations having 100 elements. Each permutation was built by applying operations in the identity permutation and the number of operations changed from 10 to 100 in intervals of 10. Therefore, the mutation rage changed from $0.1 \times n$ to n. Half of the operations are reversals, and half are transpositions.

Figures 12 and 13 indicates that **TRANS** behavior is strongly influenced by the rate of mutations. The higher the rate of mutations, the better the results provided by **TRANS**. On unsigned permutations, **TRANS** outperforms the other algorithms when we set the mutation rate to a number higher than $0.9 \times n$, which is a very extreme case. In this situation, different operations act on the same places several times, which is very unrealistic. In fact, the sorting sequence returned by the algorithms are much smaller than the scenario we used to create the permutation. We also observe that the solutions provided by **TRANS** are very poor at smaller rates.

On signed permutations, the deviation observed is not large enough and **TRANS** do not surpass the others on higer rates. In fact, our algorithm is clearly better in any mutation rate.

5.4 Expected Approximation Ratio

We tested our algorithm with 30,000 permutations of each size between 10 and 100 for signed and unsigned permutations, resulting in a total of 300,000 signed and 300,000 unsigned permutations.

We considered the instances from the three practical experiments and computed the expected approximation ratio of our algorithm. In Figure 14, the X-axis shows the percentage of permutations used in our tests that are under an approximation factor given by the Y-axis. The figure shows results for both signed and unsigned permutations. We observe that our algorithm found results

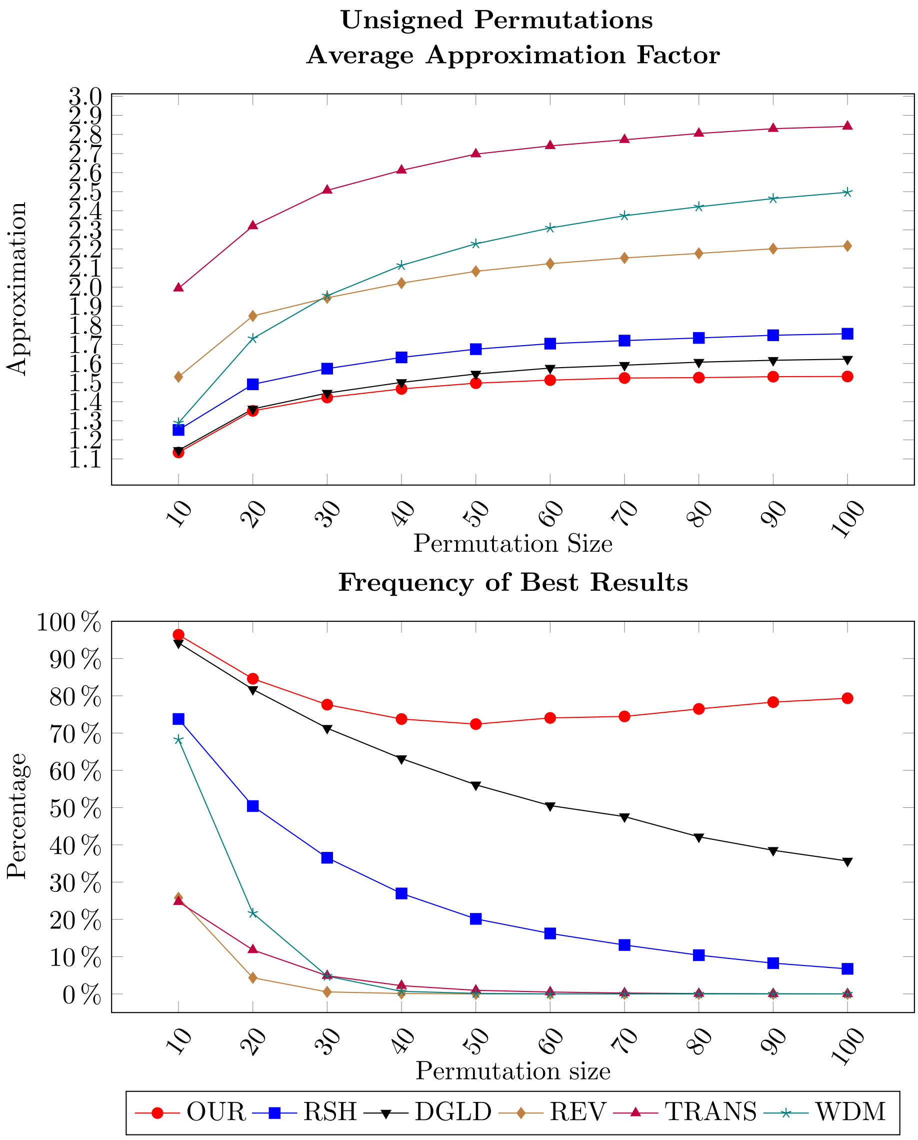

Figure 10: Average approximation factor for unsigned permutations with $0.2 \times n$ operations applied and percentage of unsigned permutations in which each algorithm found the best result.

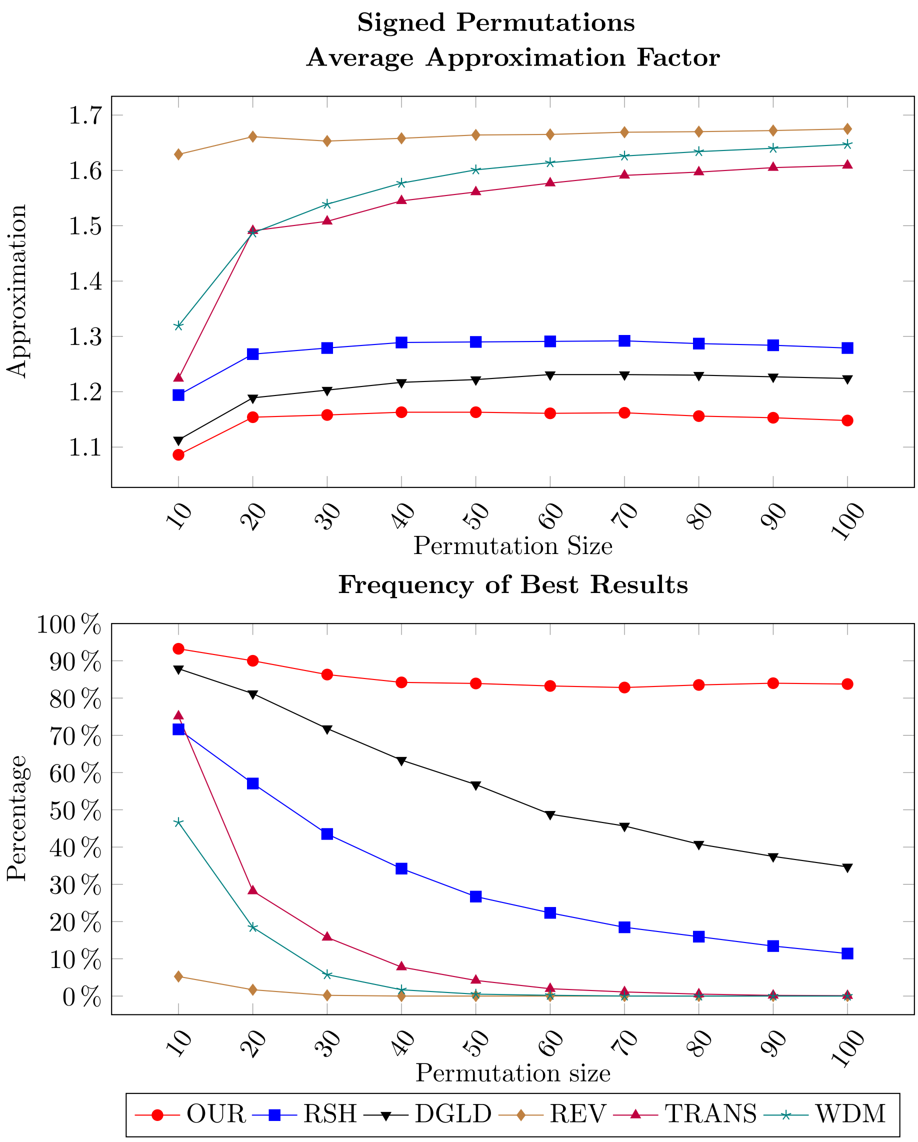

Figure 11: Average approximation factor for signed permutations with $0.2 \times n$ operations applied and percentage of signed permutations in which each algorithm found the best result.

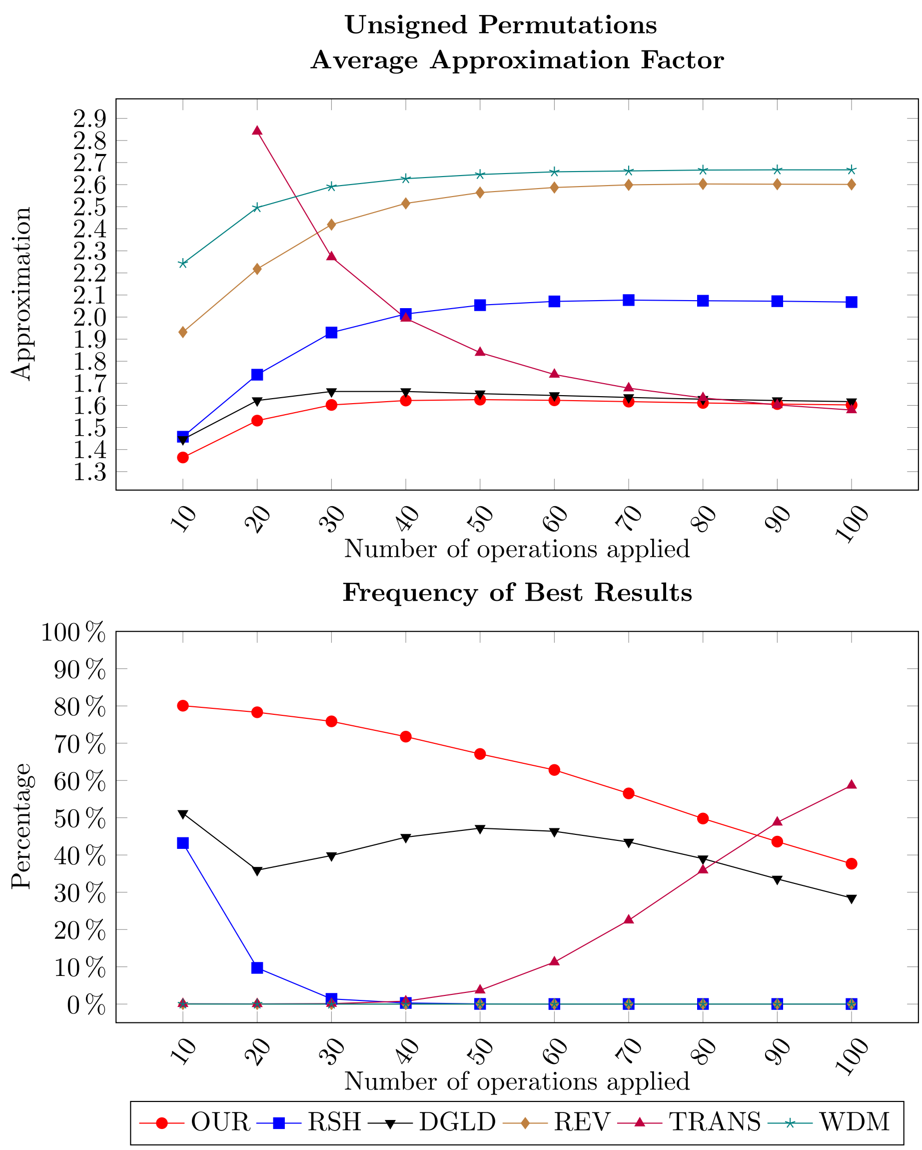

Figure 12: Average approximation factor for unsigned permutations of size 100 and percentage of unsigned permutations in which each algorithm found the best result.

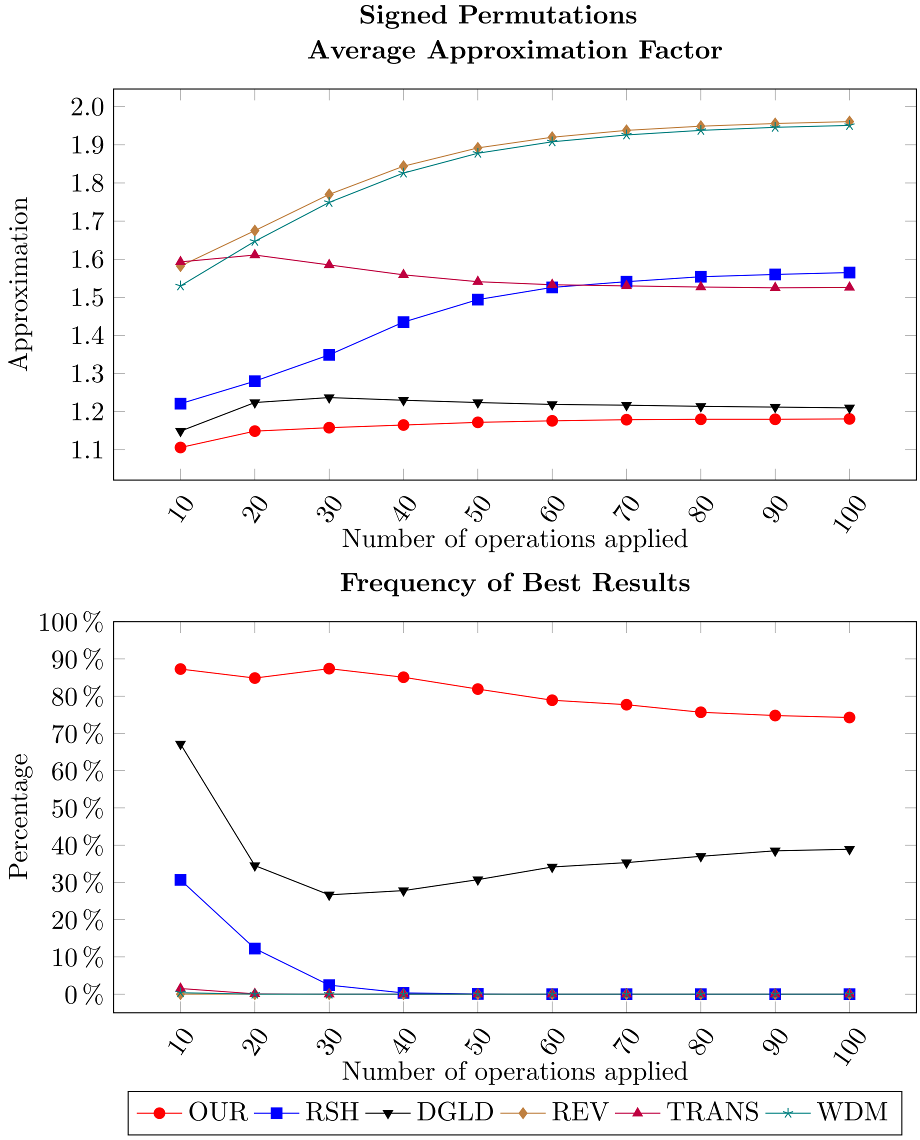

Figure 13: Average approximation factor for signed permutations of size 100 and percentage of signed permutations in which each algorithm found the best result.

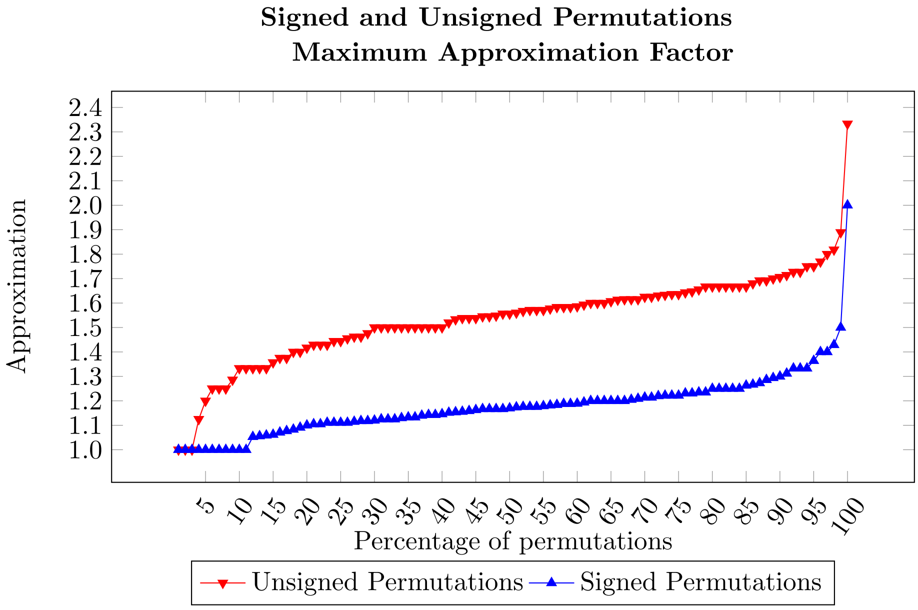

Figure 14: Percentage of permutations used in our tests that are under a given approximation factor for signed and unsigned permutations. The X-axis shows the percentage of permutations used in our tests that are under an approximation factor given by the Y-axis.

with an approximation ratio better than 1.5 in more than 99% of the signed permutations and better than 1.8 in more than 97% of the unsigned permutations. The worst case scenario, which includes those permutations such that no sorting sequence better than a 2 approximation was found, is a very rare situation.

6 Conclusions

In this paper, we presented an algorithm for the problem of sorting permutations by reversals and transpositions, two classic operations in the genome rearrangement field. Our algorithm applies to both the signed and unsigned versions of the problem, and it treats both cases in a unified manner.

We performed a theoretical analysis, and we proved that our algorithm has an approximation factor of 2 for signed permutations and $2k$ for unsigned permutations, where k is the approximation factor for the cycle graph decomposition problem. Therefore, our approximation factor equals previous approximation bounds. Our theoretical proof uses an enumeration of cases that populate several databases. Each database has a different approximation factor.

Our analysis on a group of around 300,000 signed and 300,000 unsigned instances shows that our algorithm outperforms any other approach known to date. We conclude that on larger permutations our algorithm becomes increasingly superior to the others. For signed permutations, our algorithm returned the best answer in about 80% to 90% of the cases. For unsigned permutations longer than or equal to 40 elements, our algorithm returned the best result in about 60% of the cases.

For future considerations, we intend to study the complexity of the sorting by reversals and transpositions problem. We also intend to study the diameter, which is the greatest distance between any two permutations having the same number of elements. Every sorting by genome rearrangement problem has a diameter problem associated with it, and few researchers have considered the diameter regarding reversals and transpositions.

Acknowledgments

The authors acknowledge the support from the International Cooperation Program CAPES/COFECUB Foundation under grant 831/15, the CNPq under grants 425340/2016-3 and 400487/2016-0, and also the São Paulo Research Foundation (FAPESP) under grants 2013/08293-7, 2014/19401-8, and 2015/11937-9.

References

[Bafna and Pevzner 1998] Bafna, V., Pevzner, P. A.: "Sorting by transpositions"; SIAM Journal on Discrete Mathematics; 11 (1998), 2, 224–240.

[Bergeron 2005] Bergeron, A.: "A very elementary presentation of the Hannenhalli-Pevzner theory"; Discrete Applied Mathematics; 146 (2005), 134–145.

[Berman et al. 2002] Berman, P., Hannenhalli, S., Karpinski, M.: "1.375-approximation algorithm for sorting by reversals"; Proceedings of the 10th European Symposium on Algorithms (ESA'2002); 200–210; Rome, Italy, 2002.

[Bulteau et al. 2012] Bulteau, L., Fertin, G., Rusu, I.: "Sorting by Transpositions is Difficult"; SIAM Journal on Computing; 26 (2012), 3, 1148–1180.

[Caprara 1999] Caprara, A.: "Sorting permutations by reversals and Eulerian cycle decompositions"; SIAM Journal on Discrete Mathematics; 12 (1999), 1, 91–110.

[Chen 2013] Chen, X.: "On sorting unsigned permutations by double-cut-and-joins"; Journal of Combinatorial Optimization; 25 (2013), 3, 339–351.

[Dias and Dias 2013] Dias, U., Dias, Z.: "Heuristics for the transposition distance problem"; Journal Of Bioinformatics And Computational Biology; 11 (2013), 5, 1350013.

[Dias et al. 2014a] Dias, U., Galvão, G. R., Lintzmayer, C. N., Dias, Z.: "A general heuristic for genome rearrangement problems"; Journal of Bioinformatics and Computational Biology; 12 (2014a), 03, 1450012.

[Dias et al. 2014b] Dias, U., Oliveira, A. R., Dias, Z.: "An improved algorithm for the sorting by reversals and transpositions problem"; Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics; BCB'14; 400–409; 2014b.

[Elias and Hartman 2006] Elias, I., Hartman, T.: "A 1.375-approximation algorithm for sorting by transpositions"; IEEE/ACM Transactions on Computational Biology and Bioinformatics; 3 (2006), 4, 369–379.

[Eriksen 2002] Eriksen, N.: "$(1+\epsilon)$-Approximation of Sorting by Reversals and Transpositions"; Theoretical Computer Science; 289 (2002), 1, 517–529.

[Gu et al. 1999] Gu, Q-P., Peng, S., Sudborough, I.H.: "A 2-Approximation Algorithm for Genome Rearrangements by Reversals and Transpositions"; Theoretical Computer Science; 210 (1999), 2, 327–339.

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

[Lin and Jiang 2004] Lin, G., Jiang, T.: "A further improved approximation algorithm for breakpoint graph decomposition"; Journal of Combinatorial Optimization; 8 (2004), 2, 183–194.

[Rahman et al. 2008] Rahman, A., Shatabda, S., Hasan, M.: "An approximation algorithm for sorting by reversals and transpositions"; Journal of Discrete Algorithms; 6 (2008), 3, 449 – 457.

[Tannier et al. 2007] Tannier, E., Bergeron, A., Sagot, M. F.: "Advances on sorting by reversals"; Discrete Applied Mathematics; 155 (2007), 6-7, 881–888.

[Tesler 2002] Tesler, G.: "GRIMM: genome rearrangements web server"; Bioinformatics; 18 (2002), 3, 492–493.

[Walter et al. 1998] Walter, M. E. M. T., Dias, Z., Meidanis, J.: "Reversal and transposition distance of linear chromosomes"; Proceedings of the String Processing and Information Retrieval (SPIRE'1998); 96–102; Santa Cruz, Bolivia, 1998.