Journal of Universal Computer Science, vol. 20, no. 9 (2014), 1259-1283
submitted: 25/11/13, accepted: 12/6/14, appeared: 1/9/14

On Alternative Approaches for Approximating the Transposition Distance[1]

Gustavo Rodrigues Galvão
(University of Campinas, Campinas, Brazil
ggalvao@ic.unicamp.br)

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

Abstract: We study the problem of sorting by transpositions, which consists in computing the minimum number of transpositions required to sort a permutation. This problem is NP-hard and the best approximation algorithms for solving it are based on a standard tool for attacking problems of this kind, the cycle graph. In an attempt to bypass it, some researches posed alternative approaches. In this paper, we address three algorithms yielded by such approaches: a 2.25-approximation algorithm based on breakpoint diagrams, a 3-approximation algorithm based on permutation codes, and a heuristic based on longest increasing subsequences. Regarding the 2.25-approximation algorithm, we show that previous experimental data on its approximation ratio are incorrect. Regarding the 3-approximation algorithm, we close a missing gap on the proof of its approximation ratio and we show a way to run it in $O(n \log n)$ time. Regarding the heuristic, we propose a minor adaptation that allow us to prove an approximation bound of 3. We present experimental data obtained by running these algorithms for all permutations with up to 13 elements and by running these algorithms and the best known algorithms based on the cycle graph for large permutations. The data indicate that the 2.25-approximation algorithm is the best of the algorithms based on alternative approaches and that it is the only one comparable to the algorithms based on the cycle graph.
Key Words: genome rearrangement, sorting by transpositions, approximation algorithms
Category: F.2.0, G.2.3

1 Introduction

A transposition is the rearrangement event that switches the location of two contiguous portions of a genome. The problem of computing the transposition distance between two genomes consists in finding the minimum number of transpositions needed to transform one genome into the other. Such problem finds application in comparative genomics because the transposition distance can be used to estimate the evolutionary distance between two genomes.

Representing the order of the genes in the genomes as permutations, that problem can be reduced to the combinatorial problem of finding the minimum

[1] An earlier version of this paper appeared in the Proceedings of BSB'2012 [Galvão and Dias 2012].

number of transpositions required to sort a permutation, which is referred to as the Problem of Sorting by Transpositions. This problem was recently proven to be NP-hard [Bulteau et al. 2012], therefore it is not likely that a polynomial-time algorithm exists. It was introduced by Bafna and Pevzner [Bafna and Pevzner 1998], who presented a 1.5-approximation algorithm that runs in quadratic time. Later, Elias and Hartman [Elias and Hartman 2006] improved the approximation bound to 1.375, maintaining quadratic time complexity. Recently, Dias and Dias [Dias and Dias 2010a, Dias and Dias 2010b] presented improved versions of Bafna and Pevzner's algorithm and Elias and Hartman's algorithm, which keep the original approximation ratios, and these have been the best known algorithms for the problem of sorting by transpositions.

These approximation algorithms, as well as others [Hartman and Shamir 2006, Christie 1999, Gu et al. 1999] with relatively low approximation ratios (*i.e.* less or equal than 1.5), are based on a structure named the cycle graph. This structure is regarded as complex by some authors, therefore they posed alternative approaches in order to bypass it. For a detailed literature survey, the reader is referred to the book of Fertin and colleagues [Fertin et al. 2009].

Walter, Dias, and Meidanis [Walter et al. 2000] presented a 2.25-approximation algorithm based on a structure named the breakpoint diagram that runs in $O(n^2)$ time. They ran some experiments in order to observe the approximation ratio of their algorithm in practice, but it was not conclusive. Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007] developed a 3-approximation algorithm based on permutation codes that runs in $O(n^2)$ time. According to them, although there exist better algorithms with respect to approximation ratio, their algorithm is faster than any existing one. Besides, their experimental results suggested that the approximation ratio of their algorithm may be lowered. Guyer, Heath, and Vergara [Guyer et al. 1997] devised a heuristic based on the longest increasing subsequence in a permutation that runs in $O(n^5 \log n)$ time. The experiments they performed suggested that it has the potential to produce near-optimal results.

In this work, we review these algorithms in order to improve their analyses, providing theoretical strengthening whenever possible, and to determine whether they are good alternatives to the algorithms based on the cycle graph. Regarding Walter, Dias, and Meidanis' algorithm [Walter et al. 2000], we show that previous experimental data on its approximation ratio are incorrect, and then we present new experimental data suggesting that its approximation ratio may be lowered to 2. Regarding Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007], we close a missing gap on the proof of its approximation ratio and we demonstrate a way to run it in $O(n \log n)$ time. On the other hand, we present experimental data that contradicts Benoît-Gagné and Hamel's hypothesis that its approximation ratio may be lowered. Regarding Guyer, Heath, and

Vergara's heuristic [Guyer et al. 1997], we propose a minor adaptation that allow us to prove an approximation bound of 3. On the other hand, we present experimental data that indicates this algorithm does not produce near-optimal results. Finally, we compare these three algorithms to the best known algorithms based on the cycle graph.

The remainder of this paper is organized as follows. In the next section, we give the basic definitions and notation of the paper. In Section 3, we briefly describe the algorithms studied in this paper, close a missing gap on Benoît-Gagné and Hamel's proof [Benoît-Gagné and Hamel 2007] for the approximation ratio of their algorithm (Lemma 3), and demonstrate that a constrained version of Guyer, Heath, and Vergara's heuristic [Guyer et al. 1997] still has an approximation bound of 3 (Theorem 8). In Section 4, we show how to compute the permutation codes in $O(n \log n)$ time, what allow us to implement Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007] in such a way that its running time becomes $O(n \log n)$. In Section 5, we present experimental results along with a discussion on the performance of the algorithms in practice. In the last section, we conclude the paper.

2 Preliminaries

We represent genomes as permutations, where genes appear as elements. A *permutation* π is a bijection of $\{1, 2, \ldots, n\}$ onto itself. The group of all permutations of $\{1,\ 2,\ \ldots,\ n\}$ is denoted by S_n and we write a permutation $\pi \in S_n$ as $\pi = (\pi_1\ \pi_2\ \ldots\ \pi_n)$. Sometimes, we extend it with two elements $\pi_0 = 0$ and $\pi_{n+1} = n + 1$. The extended permutation will still be called π.

A *transposition* is an operation $\rho(i,\ j,\ k)$, $1 \leq i < j < k \leq n+1$, that moves blocks of contiguous elements of a permutation $\pi \in S_n$ in such way that $\rho(i,\ j,\ k) \cdot (\pi_1\ \ldots\ \pi_{i-1}\ \underline{\pi_i\ \ldots\ \pi_{j-1}}\ \underline{\pi_j\ \ldots\ \pi_{k-1}}\ \pi_k\ \ldots\ \pi_n) = (\pi_1\ \ldots\ \pi_{i-1}\ \underline{\pi_j\ \ldots\ \pi_{k-1}}\ \underline{\pi_i\ \ldots\ \pi_{j-1}}\ \pi_k\ \ldots\ \pi_n)$. The Problem of Sorting by Transpositions consists in finding the minimum number of transpositions that transform a permutation $\pi \in S_n$ into the identity permutation $I_n = (1\ 2\ \ldots\ n)$. This number is known as the *transposition distance* of a permutation π and it is denoted by $d(\pi)$.

Given a permutation $\pi \in S_n$, a *breakpoint* is a pair of adjacent elements that are not consecutive, that is, a pair $(\pi_i,\ \pi_{i+1})$ such that $\pi_{i+1} - \pi_i \neq 1$, $0 \leq i \leq n$. The number of breakpoints of π is denoted by $b(\pi)$. Note that I_n is the only permutation in S_n having zero breakpoints. Since a transposition can remove at most three breakpoints, we can state the following lemma.

Lemma 1. *[Bafna and Pevzner 1998] For any permutation* $\pi \in S_n$, $d(\pi) \geq \frac{b(\pi)}{3}$.

A *strip* of a permutation π is a maximal series of consecutive elements without a breakpoint. We denote the number of strips in π by $s(\pi)$.

Example 1. Let $\pi = (0\ 4\ 5\ 2\ 3\ 1\ 6)$ be the extended permutation of $(4\ 5\ 2\ 3\ 1)$. We have that the pairs (0, 4), (5, 2), (3, 1), and (1, 6) are breakpoints, thus $b(\pi) = 4$. We also have that 4 5, 2 3, and 1 are strips of π, thus $s(\pi) = 3$.

Let $\pi \in S_n$, $\pi \neq I_n$, s_1 be the first strip of π, and s_m be the last strip of π, $m \leq n$. If we assume that $\pi_1 = 1$, we can transform π into a permutation $\sigma \in S_{n-|s_1|}$ such that $\sigma_i = \pi_i - |s_1|$, $i > |s_1|$. It is not hard to see that $d(\pi) = d(\sigma)$. An analogous argument can be used to show that we can transform π into a permutation $\gamma \in S_{n-|s_m|}$ such that $d(\pi) = d(\gamma)$ if $\pi_n = n$. These transformations are referred to as reductions, and we call *irreducible* any permutation in which such reductions cannot be applied. Furthermore, we denote by S_n^* the set formed by all irreducible permutations of S_n.

Lemma 2. *For any permutation* $\pi \in S_n^*$, $s(\pi) = b(\pi) - 1$.

Proof. Let $s_1, s_2, \ldots, s_{s(\pi)}$ be the strips of a permutation $\pi \in S_n^*$. The last element of strip s_i and the first element of strip s_{i+1}, $1 \leq i \leq s(\pi) - 1$, form a breakpoint, and the pairs (π_0, π_1) and (π_n, π_{n+1}) are always breakpoints on an irreducible permutation. Therefore, we have that $b(\pi) = s(\pi) + 1$ and the claim follows. □

Given a permutation $\pi \in S_n$, the *left and right codes of an element* π_i, denoted $lc(\pi_i)$ and $rc(\pi_i)$ respectively, are defined as $lc(\pi_i) = |\{\pi_j : \pi_j > \pi_i \text{ and } 1 \leq j \leq i-1\}|$ and $rc(\pi_i) = |\{\pi_j : \pi_j < \pi_i \text{ and } i+1 \leq j \leq n\}|$. The *left (resp., right) code of a permutation* π is then defined as the sequence of lc's (resp., rc's) of its elements, and it is denoted by $lc(\pi)$ (resp., $rc(\pi)$).

Let us call *plateau* any maximal length sequence of contiguous elements in a number sequence that have the same nonzero value. The number of plateaux in a code c is denoted p(c). We denote by $p(\pi)$ the minimum of $p(lc(\pi))$ and $p(rc(\pi))$. Note that I_n is the only permutation in S_n having zero plateaux.

Example 2. Let $\pi = (5\ 3\ 2\ 4\ 1)$. We have that $lc(\pi) = lc(\pi_1)\ lc(\pi_2)\ lc(\pi_3)\ lc(\pi_4)\ lc(\pi_5) = 0\ 1\ 2\ 1\ 4$, and $rc(\pi) = rc(\pi_1)\ rc(\pi_2)\ rc(\pi_3)\ rc(\pi_4)\ rc(\pi_5) = 4\ 2\ 1\ 1\ 0$. Then, $p(\pi) = \min\{p(lc(\pi)), p(rc(\pi))\} = \min\{4, 3\} = 3$.

An *increasing subsequence* of a permutation π is a subsequence $\pi_{i_1}\ \pi_{i_2} \ldots \pi_{i_j}$ of nonnecessarily contiguous elements of π such that for all k, $0 < k < j$, we have $i_k < i_{k+1}$ and $\pi_{i_k} < \pi_{i_{k+1}}$. A *longest increasing subsequence* is an increasing subsequence of π of maximum length. The set of the elements belonging to a longest increasing subsequence is denoted by LIS(π). It is easy to see that, for any $\pi \in S_n$, $|\text{LIS}(\pi)| = n$ if and only if $\pi = I_n$.

Example 3. Let $\pi = (2\ 3\ 1\ 5\ 4)$. The increasing subsequences 2 3 5 and 2 3 4 are maximal, therefore either LIS(π) = {2, 3, 5} or LIS(π) = {2, 3, 4}.

3 Algorithms

The following sections describe three algorithms for sorting by transpositions based on alternative approaches, namely Walter, Dias, and Meidanis' 2.25-approximation algorithm [Walter et al. 2000], Benoît-Gagné and Hamel's 3-approximation algorithm [Benoît-Gagné and Hamel 2007], and a constrained version of Guyer, Heath, and Vergara's heuristic [Guyer et al. 1997]. Moreover, Section 3.2 contains the missing proof for the approximation ratio of Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007] and Section 3.3 contains the demonstration that the constrained version of Guyer, Heath, and Vergara's heuristic has an approximation bound of 3.

3.1 Algorithm based on the breakpoint diagram

Walter, Dias, and Meidanis [Walter et al. 2000] developed an approximation algorithm based on breakpoints. We will not discuss in detail how this algorithm works because it relies on an extensive case by case analysis that is not relevant to the discussion we set in this paper. Nevertheless, we present below a sketch of this algorithm (Algorithm 1) to make it clear why it is a 2.25-approximation algorithm.

Algorithm 1: Sketch of the 2.25-approximation algorithm proposed by Walter, Dias, and Meidanis [Walter et al. 2000].

Data: A permutation $\pi \in S_n$.
Result: Number of transpositions applied for sorting π.

$d \leftarrow 0$;
while $\pi \neq I_n$ **do**
  if *there exists a transposition* $\rho(i, j, k)$ *that removes more than* 1 *breakpoint of* π **then**
    $\pi \leftarrow \rho(i, j, k) \cdot \pi$;
    $d \leftarrow d + 1$;
  else
    Find up to 3 transpositions that removes at least 4 breakpoints when applied on π;
    Apply on π the transpositions found and update d accordingly;
  end
end
return d;

Note that, in the worst case, Algorithm 1 removes 4 breakpoints applying 3 transpositions. Thus, denoting by $A_1(\pi)$ the number of transpositions applied by Algorithm 1 for sorting π, we have $A_1(\pi) \leq \frac{3}{4}b(\pi)$. Since $d(\pi) \geq \frac{b(\pi)}{3}$ (Lemma 1), we conclude that Algorithm 1 is a 2.25-approximation. As for its time complexity, Walter, Dias and Meidanis [Walter et al. 2000] showed that it runs in $O(n^2)$ time.

3.2 Algorithm based on permutation codes

Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007] showed that it is always possible to decrease by one unit the number of plateaux of a (right or left) code by applying a transposition. Since $p(\pi)$ represents the minimum value between $p(lc(\pi))$ and $p(rc(\pi))$, it is possible to sort π applying at most $p(\pi)$ transpositions. Thus, Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007] proposed a simple algorithm for approximating the transposition distance that only computes the value of $p(\pi)$. Such algorithm is described below (Algorithm 2).

Algorithm 2: Algorithm proposed by Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007].

Data: A permutation $\pi \in S_n$.
Result: Number of transpositions applied for sorting π.

Compute $lc(\pi)$;
Compute $rc(\pi)$;
$i \leftarrow p(lc(\pi))$;
$j \leftarrow p(rc(\pi))$;
$d \leftarrow \min\{i, j\}$;
return d;

Although Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007] stated that Algorithm 2 is a 3-approximation, we think that they did not provide a complete proof for such claim. That is, they proved that Algorithm 2 has the following approximation ratio

$$\frac{c \cdot p(\pi)}{b(\pi)}, \text{ where } c = \frac{3\lfloor \frac{b(\pi)}{3} \rfloor + b(\pi) \mod 3}{\lceil \frac{b(\pi)}{3} \rceil},$$

and they also proved that $c \leq 3$, but it lacked the proof that $p(\pi) \leq b(\pi)$. Such proof is given by Lemma 3.

Lemma 3. *Given a permutation $\pi \in S_n$, $\pi \neq I_n$, we have that $p(\pi) < b(\pi)$.*

Proof. Let $\pi \in S_n$, $\pi \neq I_n$, s_1 be the first strip of π, and s_m be the last strip of π, $m \leq n$. If $\pi_1 = 1$, then $lc(\pi_i) = rc(\pi_i) = 0$ for any element $\pi_i \in s_1$. Thus, the elements belonging to s_1 do not affect $p(\pi)$. The same can be observed for the elements belonging to s_m when $\pi_n = n$. It means that if we reduce π to σ, then $p(\pi) = p(\sigma)$. Since it is not hard to see that $b(\pi) = b(\sigma)$, we can restrict our analysis to irreducible permutations.

Let $\gamma \in S_n^*$, and let the series of consecutive elements $\gamma_i\ \gamma_{i+1}\ \ldots\ \gamma_j$ be a strip of γ. We have that $lc(\gamma_{k+1}) = lc(\gamma_k)$ and $rc(\gamma_{k+1}) = rc(\gamma_k)$, $i \leq k < j$. It means that, with respect to $lc(\gamma)$ and $rc(\gamma)$, the elements belonging to a strip of γ either have zero value or are contained in the same plateau. Therefore, $s(\gamma) \geq p(lc(\gamma))$ and $s(\gamma) \geq p(rc(\gamma))$, thus $s(\gamma) \geq p(\gamma)$. Since, by Lemma 2, $b(\gamma) > s(\gamma)$, the claim follows. $\square$

Regarding the time complexity of Algorithm 2, Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007] noted that $lc(\pi)$ and $rc(\pi)$ can be easily computed in $O(n^2)$ time, while $p(lc(\pi))$ and $p(rc(\pi))$ can be easily computed in $O(n)$ time. Therefore, they concluded that Algorithm 2 runs in $O(n^2)$ time. In Section 4, we show how to compute $lc(\pi)$ and $rc(\pi)$ in $O(n \log n)$ time.

3.3 Algorithm based on the longest increasing subsequence

Guyer, Heath, and Vergara [Guyer et al. 1997] developed a greedy algorithm based on the longest increasing subsequence of a permutation $\pi \in S_n$. At each iteration, the algorithm selects, from the $\binom{n+1}{3}$ possible transpositions, the transposition $\rho(i, j, k)$ such that $|\mathrm{LIS}(\rho(i, j, k) \cdot \pi)|$ is maximum. We say that a transposition satisfying this greedy choice is a greedy transposition. Since there may exist more than one greedy transposition, the performance of this algorithm may vary depending on the rule used to choose among greedy transpositions. Guyer, Heath, and Vergara [Guyer et al. 1997] neither pointed out any specific rule nor presented an approximation guarantee, therefore we decided to define a rule that could lead us to determine an approximation guarantee.

One might think of the rule that only greedy transpositions which remove breakpoints should be applied. Note that, using this rule, it would be trivial to prove an approximation bound of 3 due to Lemma 1. It is not true, however, that there always exists a greedy transposition satisfying this rule. For instance, among all the 56 permutations that can be obtained from permutation $\pi = (7\ 5\ 6\ 4\ 2\ 3\ 1)$ by applying a transposition, there are 8 permutations yielded by greedy transpositions, but none of them has less breakpoints than π.

We say that a transposition $\rho(i, j, k)$ does not cut a strip of a permutation π if the pairs of adjacent elements (π_{i-1}, π_i), (π_{j-1}, π_j) and (π_{k-1}, π_k) are breakpoints. The rule we considered is that only greedy transpositions which do not cut strips of permutations should be applied. Although it is possible that a greedy transposition cuts a strip of permutation (see Example 4), Lemma 6 shows that there is always a greedy transposition satisfying such a rule. Algorithm 3 is the resulting algorithm from this rule.

Algorithm 3: Constrained version of Guyer, Heath, and Vergara's heuristic [Guyer et al. 1997].

Data: A permutation $\pi \in S_n$.
Result: Number of transpositions applied for sorting π.

```
d ← 0;
while π ≠ I_n do
    d ← d + 1;
    ρ_d ← a transposition such that |LIS(ρ_d · π)| is maximum and that
        does not cut a strip of π;
    π ← ρ_d · π;
end
return d
```

Example 4. Let $\pi = (5\ 6\ 3\ 4\ 1\ 2)$. We have that $\rho(1, 3, 6)$ is a greedy transposition and it cuts the strip 1 2.

Lemma 4. *Let $\pi_1\ \pi_2\ \ldots\ \pi_r$ be the first strip of a permutation $\pi \in S_n$. If $\pi_1 = 1$, then a transposition $\rho(i, j, k)$ where $i \leq r$ cannot be a greedy transposition.*

Proof. Let $\rho(i, j, k)$ be a transposition where $i \leq r$ and let π' be the permutation $\pi' = \rho(i, j, k) \cdot \pi$. There are two cases to consider:

a) $j - 1 \leq r$. In this case, it is not hard to see that $|\text{LIS}(\pi')| \leq |\text{LIS}(\pi)|$ because the elements $\pi_i, \pi_{i+1}, \ldots, \pi_{j-1}$ are all smaller than the elements $\pi_j, \pi_{j+1}, \ldots, \pi_{k-1}$, therefore $\rho(i, j, k)$ could not be a greedy transposition.

b) $j - 1 > r$. In this case, we argue that, if $\rho(i, j, k)$ was a greedy transposition, then none of the elements $\pi_i, \pi_{i+1}, \ldots, \pi_r$ could belong to $\text{LIS}(\pi')$. For the sake of the contradiction, assume they could. Then none of the elements $\pi_j, \pi_{j+1}, \ldots, \pi_{k-1}$ could belong to $\text{LIS}(\pi')$ because they are greater than the formers. This implies that the elements in $\text{LIS}(\pi')$ would form an increasing subsequence in π, therefore $|\text{LIS}(\pi')| \leq |\text{LIS}(\pi)|$ and $\rho(i, j, k)$ could not be a greedy transposition. But if none of the elements $\pi_i, \pi_{i+1}, \ldots, \pi_r$ could belong to $\text{LIS}(\pi')$, then $\rho(i, j, k)$ could not be a greedy transposition since $|\text{LIS}(\rho(i + 1, j, k) \cdot \pi)| > |\text{LIS}(\pi')|$. □

Lemma 5. *Let $\pi_s\ \pi_{s+1} \ldots \pi_n$ be the last strip of a permutation $\pi \in S_n$. If $\pi_n = n$, then a transposition $\rho(i, j, k)$ where $k > s$ cannot be a greedy transposition.*

Proof. Analogous to the proof of Lemma 4. □

Lemma 6. *Given a permutation π, there exists a greedy transposition which does not cut any of its strips.*

Proof. Let $\rho(i, j, k)$ be a greedy transposition, and let π' be the permutation such that $\pi' = \rho(i, j, k) \cdot \pi$. If $\rho(i, j, k)$ does not cut a strip of π, then we are done. Otherwise, we have to basically consider three possibilities:

(a) (π_{i-1}, π_i) is not a breakpoint.

In this case let i' be the greatest integer such that $i' < i$ and $(\pi_{i'-1}, \pi_{i'})$ is a breakpoint (Lemma 4 guarantees that $i' \geq r$ when $\pi_1 = 1$), and let π'' be the permutation such that $\pi'' = \rho(i', j, k) \cdot \pi$. Then, we have four subcases to analyze:

(i) $\pi_i \in \mathrm{LIS}(\pi')$ and $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \in \mathrm{LIS}(\pi')$. In this subcase we have that the elements in $\mathrm{LIS}(\pi')$ form an increasing subsequence in π'', therefore $|\mathrm{LIS}(\pi'')| \geq |\mathrm{LIS}(\pi')|$. Since $\rho(i, j, k)$ is a greedy transposition by hypothesis, we have that $|\mathrm{LIS}(\pi'')| \leq |\mathrm{LIS}(\pi')|$. Therefore $|\mathrm{LIS}(\pi'')| = |\mathrm{LIS}(\pi')|$ and $\rho(i', j, k)$ is also a greedy transposition.

(ii) $\pi_i \in \mathrm{LIS}(\pi')$ and $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \notin \mathrm{LIS}(\pi')$. In this subcase we have that the elements in $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\}$ along with the elements in $\mathrm{LIS}(\pi')$ form an increasing subsequence in π'', therefore $|\mathrm{LIS}(\pi'')| > |\mathrm{LIS}(\pi')|$ and this contradicts our hypothesis that $\rho(i, j, k)$ is a greedy transposition.

(iii) $\pi_i \notin \mathrm{LIS}(\pi')$ and $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \in \mathrm{LIS}(\pi')$. In this subcase we have that $|\mathrm{LIS}(\rho(i+1, j, k) \cdot \pi)| > |\mathrm{LIS}(\pi')|$ and this contradicts our hypothesis that $\rho(i, j, k)$ is a greedy transposition.

(iv) $\pi_i \notin \mathrm{LIS}(\pi')$ and $\{\pi_{i'}, \pi_{i'+1}, \ldots, \pi_{i-1}\} \notin \mathrm{LIS}(\pi')$. The proof for this subcase is the same as that in subcase (a.i).

(b) (π_{j-1}, π_j) is not a breakpoint.

In this case let j' be the least integer such that $j < j'$ and $(\pi_{j'-1}, \pi_{j'})$ is a breakpoint, and let j'' be the greatest integer such that $j'' < j$ and $(\pi_{j''-1}, \pi_{j''})$ is a breakpoint. It may be the case that either $j' = k$ or $j'' = i$, but it is impossible that $j' = k$ and $j'' = i$, otherwise $\rho(i, j, k)$ would only move elements belonging to the same strip, therefore $|\mathrm{LIS}(\pi')| \leq |\mathrm{LIS}(\pi)|$ and $\rho(i, j, k)$ could not be a greedy transposition. Also note that a situation where

$\pi_{j-1} \in \text{LIS}(\pi')$ and $\pi_j \in \text{LIS}(\pi')$ is not possible given the definition of an increasing subsequence. Then, if we assume that $j' \neq k$ and let π'' be the permutation such that $\pi'' = \rho(i, j', k) \cdot \pi$, we have three subcases to analyze:

(i) $\pi_{j-1} \in \text{LIS}(\pi')$ and $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \notin \text{LIS}(\pi')$. This subcase is analogous to subcase (a.ii).

(ii) $\pi_{j-1} \notin \text{LIS}(\pi')$ and $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \in \text{LIS}(\pi')$. In this subcase we have that $|\text{LIS}(\rho(i, j-1, k) \cdot \pi)| > |\text{LIS}(\pi')|$ and this contradicts our hypothesis that $\rho(i, j, k)$ is a greedy transposition.

(iii) $\pi_{j-1} \notin \text{LIS}(\pi')$ and $\{\pi_j, \pi_{j+1}, \ldots, \pi_{j'-1}\} \notin \text{LIS}(\pi')$. This subcase is analogous to subcase (a.iv).

On the other hand, if we assume that $j'' \neq i$ and let π'' be the permutation such that $\pi'' = \rho(i, j'', k) \cdot \pi$, we also have three subcases to analyze:

(i) $\pi_j \in \text{LIS}(\pi')$ and $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \notin \text{LIS}(\pi')$. This subcase is analogous to subcase (a.ii).

(ii) $\pi_j \notin \text{LIS}(\pi')$ and $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \in \text{LIS}(\pi')$. In this subcase we have that $|\text{LIS}(\rho(i, j+1, k) \cdot \pi)| > |\text{LIS}(\pi')|$ and this contradicts our hypothesis that $\rho(i, j, k)$ is a greedy transposition.

(iii) $\pi_j \notin \text{LIS}(\pi')$ and $\{\pi_{j''}, \pi_{j''+1}, \ldots, \pi_{j-1}\} \notin \text{LIS}(\pi')$. This subcase is analogous to subcase (a.iv).

(c) (π_{k-1}, π_k) is not a breakpoint.

In this case let k' be the least integer such that $k < k'$ and $(\pi_{k'-1}, \pi_{k'})$ is a breakpoint (Lemma 5 guarantees that $k' \leq s$ when $\pi_n = n$), and let π'' be the permutation such that $\pi'' = \rho(i, j, k') \cdot \pi$. Then, we have four subcases to analyze:

(i) $\pi_{k-1} \in \text{LIS}(\pi')$ and $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \in \text{LIS}(\pi')$. This subcase is analogous to subcase (a.i).

(ii) $\pi_{k-1} \in \text{LIS}(\pi')$ and $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \notin \text{LIS}(\pi')$. This subcase is analogous to subcase (a.ii).

(iii) $\pi_{k-1} \notin \text{LIS}(\pi')$ and $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \in \text{LIS}(\pi')$. In this subcase we have that $|\text{LIS}(\rho(i, j, k-1) \cdot \pi)| > |\text{LIS}(\pi')|$ and this contradicts our hypothesis that $\rho(i, j, k)$ is a greedy transposition.

(iv) $\pi_{k-1} \notin \text{LIS}(\pi')$ and $\{\pi_k, \pi_{k+1}, \ldots, \pi_{k'-1}\} \notin \text{LIS}(\pi')$. This subcase is analogous to subcase (a.iv).

Although more than one possibility can occur at the same time, they are independent from each other, in such a way that in all possible cases, if transposition $\rho(i,\ j,\ k)$ cuts a strip of π, then it is possible to derive a greedy transposition which does not, thus the claim follows. $\square$

Based on the fact that Algorithm 3 does not apply greedy transpositions that cut strips, we are able to prove an upper bound on the number of transpositions it applies for sorting permutations (Lemma 7). Using this upper bound, it is possible to prove that Algorithm 3 is a 3-approximation (Theorem 8).

Lemma 7. *Let $A_3(\pi)$ be the number of transpositions applied by Algorithm 3 for sorting a permutation π. Then, we have $A_3(\pi) \leq s(\pi) - 1$.*

Proof. For helping us to determine an upper bound to the number of transpositions applied by Algorithm 3, we define a simple procedure, called *StripSum*, which sums the sizes of the strips of a permutation. It receives as input a permutation $\pi \in S_n$ and proceeds as follows. Firstly, it sorts all the strips of π with respect to their sizes, obtaining a list of strips $s^0,\ s^1,\ \ldots,\ s^{s(\pi)-1}$ such that $|s^i| \geq |s^{i+1}|$ for all i, $0 \leq i < s(\pi) - 1$. Secondly, it initializes a variable named SUM to $|s^0|$. Finally, starting from s^1, it iterates over the list of strips such that, at iteration i, the algorithm increases the value of SUM by $|s^i|$. Let SUM_i be the value of the variable SUM at iteration i, with $\mathrm{SUM}_0 = |s^0|$. Clearly, $\mathrm{SUM}_i = \mathrm{SUM}_{i-1} + |s^i|$ for all i, $1 \leq i \leq s(\pi) - 1$. Besides, when the algorithm stops, $\mathrm{SUM} = \mathrm{SUM}_{s(\pi)-1} = |s^0| + |s^1| + \cdots + |s^{s(\pi)}| = n$.

Now, assume that π was given as input to Algorithm 3 and let π^i be the permutation produced after i iterations, with $\pi^0 = \pi$. We can prove by induction that $|\mathrm{LIS}(\pi^i)| \geq \mathrm{SUM}_i$. For the base case, we have $|\mathrm{LIS}(\pi^0)| \geq \mathrm{SUM}_0$ because, by definition, $|\mathrm{LIS}(\pi^0)|$ must be equal or greater than the size of any strip of π^0. For the induction step, assume the claim holds for some $0 < i < A_3(\pi)$. Since Algorithm 3 never cuts a strip, all the strips of π^i are formed by strips of π^0. Let s' be the strip of greatest size among all strips of π^0 whose elements do not belong to a given $\mathrm{LIS}(\pi^i)$. We have that $|\mathrm{LIS}(\pi^{i+1})| \geq |\mathrm{LIS}(\pi^i)| + |s'|$ because it is possible to apply a transposition on π^i and obtain a new permutation containing an increasing subsequence formed by the elements of s' and $\mathrm{LIS}(\pi^i)$. If $|s'| \geq |s^{i+1}|$, then $|\mathrm{LIS}(\pi^{i+1})| \geq |\mathrm{LIS}(\pi^i)| + |s'| \geq \mathrm{SUM}_i + |s^{i+1}| = \mathrm{SUM}_{i+1}$. Otherwise, if $|s'| < |s^{i+1}|$, it means that the elements of all strips s^t, $0 \leq t \leq i+1$, belong to $\mathrm{LIS}(\pi^i)$, therefore $|\mathrm{LIS}(\pi^{i+1})| > |\mathrm{LIS}(\pi^i)| \geq \mathrm{SUM}_{i+1}$.

The inequality $|\mathrm{LIS}(\pi^i)| \geq \mathrm{SUM}_i$ implies that Algorithm 3 makes $|\mathrm{LIS}(\pi)|$ converge to n applying no more transpositions than the number of iterations that procedure *StripSum* performs. Since it performs $s(\pi) - 1$ iterations, the lemma follows. $\square$

Theorem 8. *Algorithm 3 is a 3-approximation.*

Proof. Lemma 4 guarantees that Algorithm 3 will never apply a transposition which moves the elements of the first strip of π when $\pi_1 = 1$. Similarly, Lemma 5 guarantees that Algorithm 3 will never apply a transposition which moves the elements of the last strip of π when $\pi_n = n$. For this reason, if we reduce permutation π to a permutation σ, it is not hard to see that $A_3(\pi) = A_3(\sigma)$. Thus, we can restrict our analysis to irreducible permutations.

Let $\gamma \in S_n^*$. By Lemma 7, we have that $A_3(\gamma) \leq s(\gamma) - 1$. Since, by Lemma 2, $s(\gamma) = b(\gamma) - 1$, we conclude that $A_3(\gamma) \leq b(\gamma) - 2$. It means that $A_3(\gamma) \leq 3d(\gamma)$ once $d(\gamma) \geq \frac{b(\gamma)}{3}$ (Lemma 1), and the theorem has been proved. □

Since there are $O(n^3)$ possible transpositions to consider per iteration, it takes $O(n\log n)$ time to determine a longest increasing subsequence of a permutation, it takes $O(1)$ time to determine whether a transposition cuts a strip of a permutation, and the while loop executes $O(n)$ times, we conclude that Algorithm 3 runs in $O(n^5\log n)$ time.

4 Computing Permutation Codes in $O(n \log n)$ Time

In this section, we describe how to compute the left and right codes of a permutation with n elements in $O(n \log n)$ time, what allow us to implement Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007] in such a way that its running time becomes $O(n \log n)$. We note that permutation codes are closely related to the Lehmer code [Lehmer 1960] (in fact, the Lehmer code is equivalent to the right code of a permutation) and there are known algorithms for computing the Lehmer code in $O(n \log n)$ time (see [Arndt 2010, page 235]).

Given a permutation $\pi \in S_n$, the left code of the element π_k in the interval $[i, j]$, denoted by $lc_{[i,j]}$, is defined as

$$lc_{[i,j]}(\pi_k) = |\{\pi_l : \pi_l > \pi_k \text{ and } i \leq l \leq k-1\}|$$

for all $i \leq k \leq j$. Similarly, the right code of the element π_k in the interval $[i, j]$, denoted by $rc_{[i,j]}$, is defined as

$$rc_{[i,j]}(\pi_k) = |\{\pi_l : \pi_l < \pi_k \text{ and } k+1 \leq l \leq j\}|$$

for all $i \leq k \leq j$. If $i = j$, then we define $lc_{[i,i]}(\pi_k) = rc_{[i,i]}(\pi_k) = 0$.

It is not hard to realize that $lc_{[i,j]}(\pi_k) = lc(\pi_k)$ and $rc_{[i,j]}(\pi_k) = rc(\pi_k)$ when $i = 1$ and $j = n$. By looking at the problem of computing the left/right code of the elements of a permutation $\pi \in S_n$ as the problem of computing the left/right code of such elements in the interval $[1, n]$, it becomes clearer that we can use a divide-and-conquer approach to solve it. Firstly, we compute recursively the left/right code of the elements $\pi_1\ \pi_2\ \dots\ \pi_m$ in the interval $[1, m]$ and the left/right code of the elements $\pi_{m+1}\ \pi_{m+2}\ \dots\ \pi_n$ in the interval $[m+1,$

$n]$ such that $m = \lfloor \frac{n+1}{2} \rfloor$. As for the base case, we have $lc_{[i,i]}(\pi_i) = rc_{[i,i]}(\pi_i) = 0$ for all $1 \leq i \leq n$. Once the left/right codes of these elements have been computed in the referred intervals, we have that:

- $lc_{[1,n]}(\pi_k) = lc_{[1,m]}(\pi_k)$ for all $1 \leq k \leq m$ and $lc_{[1,n]}(\pi_k) = lc_{[m+1,n]}(\pi_k) + |\{\pi_l : \pi_l > \pi_k \text{ and } 1 \leq l \leq m\}|$ for all $m+1 \leq k \leq n$;
- $rc_{[1,n]}(\pi_k) = rc_{[m+1,n]}(\pi_k)$ for all $m+1 \leq k \leq n$ and $rc_{[1,n]}(\pi_k) = rc_{[1,m]}(\pi_k) + |\{\pi_l : \pi_l < \pi_k \text{ and } m+1 \leq l \leq n\}|$ for all $1 \leq k \leq m$.

This means that, regarding the left code, the main question is how to efficiently compute $|\{\pi_l : \pi_l > \pi_k \text{ and } 1 \leq l \leq m\}|$ for every element π_k such that $m+1 \leq k \leq n$; regarding the right code, it is how to efficiently compute $|\{\pi_l : \pi_l < \pi_k \text{ and } m+1 \leq l \leq n\}|$ for every element π_k such that $1 \leq k \leq m$.

The answer for the above questions relies on mergesort, that is, it is possible to adapt the merge step of mergesort so that we can efficiently establish the order relations between the elements and, consequently, we can efficiently compute those values. Firstly, we present an algorithm, called *MergeLeftCodes*, that shows how to accomplish it regarding the left code.

Algorithm *MergeLeftCodes* receives four parameters as input: a vector L containing the elements $\pi_i, \pi_{i+1}, \ldots, \pi_j$ ordered in ascending order; a vector R containing the elements $\pi_{j+1}, \pi_{j+2}, \ldots, \pi_k$ ordered in ascending order; a vector LC such that $LC[e] = lc_{[i,j]}(\pi_e)$ for all $i \leq e \leq j$ and $LC[e] = lc_{[j+1,k]}(\pi_e)$ for all $j+1 \leq e \leq k$; and the inverse permutation of π, π^{-1}. As a result, it returns a vector M containing the elements of vectors L and R ordered in ascending order and updates vector LC in such a way that $LC[e] = lc_{[i,k]}(\pi_e)$ for all $i \leq e \leq k$.

We will prove the correctness of Algorithm *MergeLeftCodes* by proving that the following loop invariants hold for the while loop of lines 7-18:

- vector M contains the $m-1$ smallest elements of L and R ordered in ascending order;
- $LC[e] = lc_{[i,k]}(\pi_e)$, where $e = \pi^{-1}_{M[t]}$, for all $1 \leq t \leq m-1$;
- $L[l]$ and $R[r]$ are the smallest elements of vectors L and R that have not been copied to M.

It not hard to see that these loop invariants hold before the first iteration of the while loop of lines 7-18. At each iteration, we have to consider two possibilities: either $R[r] < L[l]$ or $R[r] > L[l]$.

If $R[r] < L[l]$, then $R[r]$ is the smallest element that has not been copied to M, therefore it is copied to M (line 9). Since the elements of L are to the left of the elements of R in the permutation π, it means that there exists $|L| - l$ elements greater than $M[m]$ and to its left in π considering just the elements of L (note that the element $n+1$ must not be considered). Given that $LC[e]$, e

Algorithm 4: MergeLeftCodes.

Data: Three vectors L, R and LC, and permutation $\pi^{-1} \in S_n$.
Result: Returns a vector containing the elements of L and R ordered in ascending order and updates vector LC in such a way that $LC[e] = lc_{[i,k]}(\pi_e)$ for all $i \leq e \leq k$.

1 $l \leftarrow 1$;
2 $r \leftarrow 1$;
3 $m \leftarrow 1$;
4 Let M be a vector of size $|L| + |R|$;
5 $R[|R| + 1] \leftarrow n + 1$;
6 $L[|L| + 1] \leftarrow n + 1$;
7 **while** $m \leq |M|$ **do**
8   **if** $R[r] < L[l]$ **then**
9         $M[m] \leftarrow R[r]$;
10         $e \leftarrow \pi^{-1}_{M[m]}$;
11         $LC[e] \leftarrow LC[e] + |L| - l$;
12         $r \leftarrow r + 1$;
13   **else**
14         $M[m] \leftarrow L[l]$;
15         $l \leftarrow l + 1$;
16   **end**
17     $m \leftarrow m + 1$;
18 **end**
19 **return** M;

$= \pi^{-1}_{M[m]}$, equals the number of elements greater than $M[m]$ and to its left in π considering just the elements of R, we conclude that $LC[e] + |L| - l$ equals the number of elements greater then $M[m]$ and to its left in π considering both the elements of L and R. In other words, we have $LC[e] = lc_{[i,k]}(\pi_e)$ after line 11. Finally, the variables r and m are incremented (lines 12 and 17), thus the loop invariants still hold.

If $R[r] > L[l]$, then $L[l]$ is the smallest element that has not been copied to M, therefore it is copied to M (line 14). Since the elements of L are to the left of the elements of R in the permutation π, we have $LC[e] = lc_{[i,k]}(\pi_e)$, $e = \pi^{-1}_{M[m]}$, therefore it is not necessary to update vector LC. Finally, the variables l and m are incremented (lines 15 and 17), thus the loop invariants still hold.

After the while loop of lines 7-18 terminates, it is clear that vector M will contain the elements of vectors L and R ordered in ascending order. Besides, vector LC will have been updated in such a way that $LC[e] = lc_{[i,k]}(\pi_e)$ for all

$i \le e \le k$.

As for the time complexity of Algorithm *MergeLeftCodes*, we have that each of the lines 1, 2, 3, 5, and 6 runs in time $O(1)$, line 4 runs in time $O(n)$, and the while loop executes $O(n)$ times. Therefore, Algorithm *MergeLeftCodes* runs in $O(n)$ time.

With Algorithm *MergeLeftCodes* in hand, the recursive algorithm that computes the left code of the elements $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$ in the interval $[i,\ j]$ can be trivially derived from the discussion held at the beginning of this section. This algorithm is called *RecursiveLeftCode* and it is presented below. In order to obtain the left code of a permutation π, we simply execute Algorithm *RecursiveLeftCode* as described by Algorithm 6.

Algorithm 5: RecursiveLeftCode.

Data: A permutation $\pi \in S_n$ and its inverse, π^{-1}, vector LC, and indexes i and j.

Result: Returns a vector containing the elements $\pi_i\ \pi_{i+1}\ \ldots\ \pi_j$ ordered in ascending order and updates vector LC in such a way that $LC[e] = lc_{[i,j]}(\pi_e)$ for all $i \le e \le j$.

$m \leftarrow \lfloor \frac{i+j}{2} \rfloor$;
if $i < j$ **then**
    $L \leftarrow$ `RecursiveLeftCode(`$\pi,\ \pi^{-1},\ LC,\ i,\ m$`)`;
    $R \leftarrow$ `RecursiveLeftCode(`$\pi,\ \pi^{-1},\ LC,\ m+1,\ j$`)`;
    $M \leftarrow$ `MergeLeftCodes(`$L,\ R,\ LC,\ \pi^{-1}$`)`;
else
    Let M be a vector of size 1;
    $M[1] \leftarrow \pi_i$;
    $LC[i] \leftarrow 0$;
end
return M;

The time complexity of Algorithm *RecursiveLeftCode* equals the time complexity of mergesort, which is $O(n \log n)$. Computing π^{-1} as well as creating a vector of size n takes $O(n)$ time, therefore Algorithm 6 runs in $O(n \log n)$ time.

In the case of the right code, the adaptation to the merge step of mergesort is very similar to the one made in the case of the left code, except that the elements are ordered in descending order rather than in ascending order. Moreover, it is not hard to see that the correctness and complexity analyses of the algorithms for computing the right code of a permutation are analogous to the ones performed for algorithms *MergeLeftCodes* and *RecursiveLeftCode*, therefore we omit them.

Algorithm 6: Computing the left code of a permutation.

Data: A permutation $\pi \in S_n$.

Result: Returns a vector containing the left codes of the elements of π.

```
Compute π^{-1};
Let LC be a vector of size n;
RecursiveLeftCode(π, π^{-1}, LC, 1, n);
return LC;
```

5 Experimental Results and Discussion

The following sections describe the experiments we have performed and discuss the results we have obtained. The algorithms described in this paper were implemented in Java, while the algorithms based on the cycle graph were implemented in Python (we used Dias and Dias [Dias and Dias 2010a, Dias and Dias 2010b] implementations). The experiments were performed on an Intel® Core™ i7-2600K CPU at 3.40GHz with 16GB of RAM running Ubuntu 12.04.2 LTS operating system.

5.1 Experiments on small permutations

The approximation algorithms presented in Section 3 were implemented and tested by their authors for verifying their performance in practice. One kind of test was to compare the distance computed by the algorithm with $d(\pi)$ for every $\pi \in S_n$ in order to obtain the real approximation ratio of the respective approximation algorithm for small permutations. More specifically, Walter, Dias, and Meidanis [Walter et al. 2000] ran this test for $1 \leq n \leq 11$, Benoît-Gagné and Hamel [Benoît-Gagné and Hamel 2007] ran it for $1 \leq n \leq 9$, and Guyer, Heath, and Vergara [Guyer et al. 1997] ran it just for $n = 6$.

We ran this kind of test for $1 \leq n \leq 13$ for all algorithms using GRAAu [Galvão and Dias 2014], and the results are presented in tables 1, 2, and 3, where n is the size of the permutations, *Max. Dist.* is the greatest distance outputted by the algorithm, *Avg. Dist.* is the average of the distances outputted by the algorithm, *Avg. Ratio* is the average of the ratios between the distance outputted by the algorithm and the transposition distance, *Max. Ratio* is the greatest ratio among all the ratios between the distance outputted by the algorithm and the transposition distance, and *Equals* is the percentage of distances outputted by the algorithm that is equal to the transposition distance.

Firstly, we note that the results obtained by Walter, Dias, and Meidanis for their 2.25-approximation algorithm are incorrect. For instance, the maximum value of $\frac{A_1(\pi)}{d(\pi)}$ they observed for $n = 11$ was $\frac{10}{5}$. But this result cannot be right

n	Max. Dist.	Avg. Dist.	Avg. Ratio	Max. Ratio	Equals
1	0	0.00	1.00	1.00	100.00%
2	1	0.50	1.00	1.00	100.00%
3	2	1.00	1.00	1.00	100.00%
4	3	1.54	1.00	1.00	100.00%
5	3	2.08	1.00	1.00	100.00%
6	4	2.61	1.00	1.33	99.17%
7	5	3.14	1.00	1.33	98.57%
8	6	3.66	1.01	1.50	97.12%
9	6	4.19	1.01	1.50	96.06%
10	7	4.70	1.01	1.50	94.15%
11	8	5.22	1.01	1.60	92.84%
12	9	5.73	1.02	1.60	90.68%
13	9	6.24	1.02	1.60	89.30%

Table 1: Results obtained from the audit of the implementation of Walter, Dias, and Meidanis' algorithm.

n	Max. Dist.	Avg. Dist.	Avg. Ratio	Max. Ratio	Equals
1	0	0.00	1.00	1.00	100.00%
2	1	0.50	1.00	1.00	100.00%
3	2	1.00	1.00	1.00	100.00%
4	3	1.54	1.00	1.00	100.00%
5	4	2.13	1.02	1.50	95.00%
6	5	2.75	1.06	1.67	85.00%
7	6	3.42	1.10	2.00	71.77%
8	7	4.13	1.14	2.00	56.41%
9	8	4.87	1.18	2.00	41.62%
10	9	5.63	1.22	2.25	28.80%
11	10	6.42	1.25	2.25	18.74%
12	11	7.22	1.29	2.25	11.57%
13	12	8.05	1.32	2.40	6.77%

Table 2: Results obtained from the audit of the implementation of Benoît-Gagné and Hamel's algorithm.

n	Max. Dist.	Avg. Dist.	Avg. Ratio	Max. Ratio	Equals
1	0	0.00	1.00	1.00	100.00%
2	1	0.50	1.00	1.00	100.00%
3	2	1.00	1.00	1.00	100.00%
4	3	1.54	1.00	1.00	100.00%
5	4	2.10	1.01	1.50	97.50%
6	5	2.67	1.03	1.50	92.78%
7	6	3.26	1.05	1.67	86.45%
8	7	3.86	1.06	1.67	77.93%
9	8	4.48	1.08	2.00	69.06%
10	9	5.10	1.10	2.00	58.94%
11	10	5.73	1.12	2.00	49.61%
12	11	6.38	1.14	2.00	40.23%
13	12	7.03	1.15	2.25	32.18%

Table 3: Results obtained from the audit of the implementation of Algorithm 3, which is a constrained version of Guyer, Heath, and Vergara's heuristic.

Permutation	**Transposition Sorting Sequence**
$\pi^5 = (16\ 9\ 4\ 11\ 6\ 15\ 8\ 2\ 12\ 7\ 5\ 3\ 14\ 13\ 10\ 1)$	$\rho(5,\ 9,\ 12)$, $\rho(1,\ 7,\ 10)$, $\rho(3,\ 9,\ 14)$, $\rho(5,\ 10,\ 17)$, $\rho(6,\ 10,\ 14)$, $\rho(1,\ 7,\ 11)$
$\pi^6 = (19\ 11\ 4\ 18\ 6\ 14\ 8\ 13\ 10\ 2\ 15\ 5\ 9\ 7\ 3\ 17\ 16\ 12\ 1)$	$\rho(4,\ 12,\ 17)$, $\rho(7,\ 12,\ 16)$, $\rho(6,\ 9,\ 15)$, $\rho(1,\ 5,\ 11)$, $\rho(3,\ 8,\ 20)$, $\rho(4,\ 13,\ 17)$, $\rho(1,\ 5,\ 13)$
$\pi^7 = (22\ 13\ 4\ 21\ 6\ 17\ 8\ 16\ 10\ 15\ 12\ 2\ 18\ 5\ 11\ 9\ 7\ 3\ 20\ 19\ 14\ 1)$	$\rho(4,\ 14,\ 20)$, $\rho(8,\ 13,\ 19)$, $\rho(7,\ 10,\ 18)$, $\rho(6,\ 9,\ 17)$, $\rho(1,\ 5,\ 11)$, $\rho(3,\ 8,\ 23)$, $\rho(4,\ 16,\ 20)$, $\rho(1,\ 5,\ 15)$

Table 4: Permutations π^m of size $3m+1$, $m \in \{5, 6, 7\}$, for which $\frac{p(\pi^m)}{d(\pi^m)} = \frac{3m}{m+1}$. Note that $d(\pi^m) \geq \frac{b(\pi^m)}{3} \geq m+1$.

because $A_1(\pi) \leq 9$ for every $\pi \in S_{11}$. Given a permutation $\pi \in S_n$, it is easy to see that $0 \leq b(\pi) \leq n+1$. As discussed in Section 3.1, $A_1(\pi) \leq \frac{3}{4}b(\pi)$, therefore $A_1(\pi) \leq \frac{3n+3}{4}$, and the claim follows.

The real approximation ratios observed for the 2.25-approximation algorithm seem to increase in a progression that converges to 2, that is, $\frac{2}{2}$, $\frac{4}{3}$, $\frac{6}{4}$, $\frac{8}{5}$, $\ldots$, $\frac{2k}{k+1}$. This may indicate that a deeper analysis of this algorithm could lead one to prove that it is in fact a 2-approximation.

Regarding the 3-approximation algorithm developed by Benoît-Gagné and Hamel, if we just consider the real approximation ratios obtained for $n \in \{7, 10, 13\}$, we can observe that they seem to follow the progression $\frac{6}{3}$, $\frac{9}{4}$, $\frac{12}{5}$, $\ldots$, $\frac{3k}{k+1}$. We ran further experiments to verify the strength of this assumption, and we found permutations π^m of size $3m+1$, $m \in \{5, 6, 7\}$, for which $\frac{p(\pi^m)}{d(\pi^m)} = \frac{3m}{m+1}$ (these permutations are presented in Table 4). Note that, for $m = 7$, the real approximation ratio of Benoît-Gagné and Hamel's algorithm equals $\frac{21}{8} = 2.625$. This is an indication that the approximation ratio of this algorithm may not be lowered, contradicting the hypothesis raised by Benoît-Gagné and Hamel that its approximation ratio "tends to a number significantly smaller than 3".

Figure 1 illustrates that, of the three algorithms, Walter, Dias, and Meidanis' algorithm has the best practical performance.

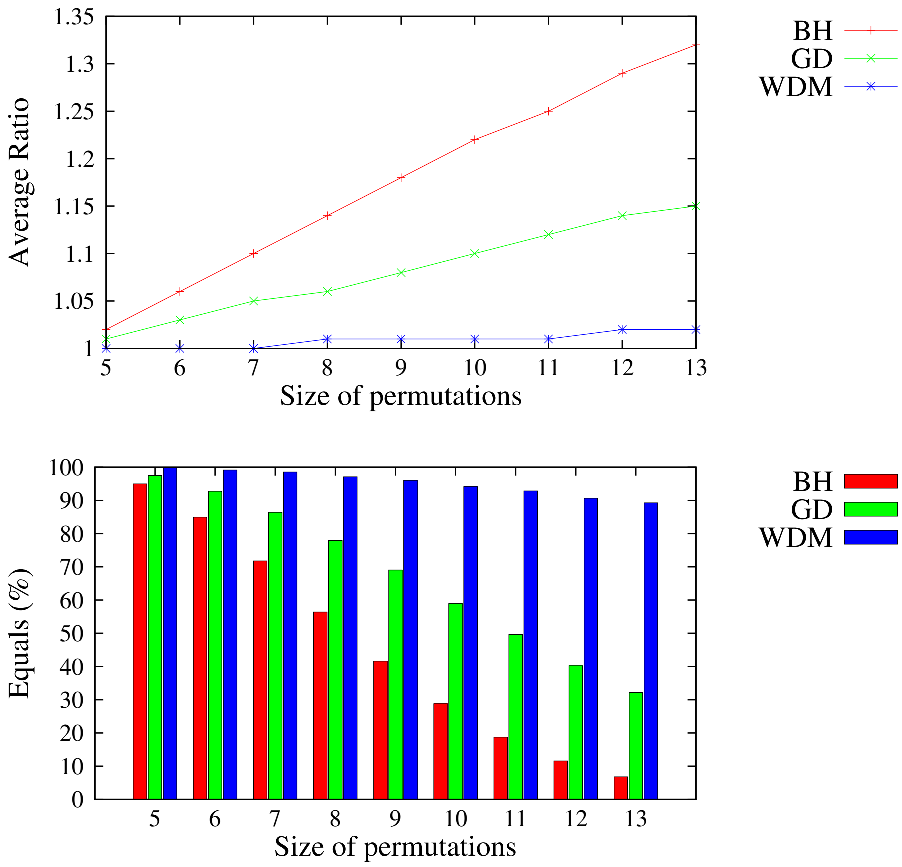

Figure 1: Comparison of Walter, Dias, and Meidanis' algorithm (WDM), Benoît-Gagné and Hamel's algorithm (BH), and the constrained version of Guyer, Heath, and Vergara's heuristic (GD) based on the results provided by GRAAu.

5.2 Experiments on large permutations

In order to investigate what happens in practice for large permutations and to compare the algorithms regarded in this paper against the best known algorithms based on the cycle graph (namely Bafna and Pevzner's 1.5-approximation algorithm [Bafna and Pevzner 1998], Elias and Hartman's 1.375-approximation algorithm [Elias and Hartman 2006], Dias and Dias' [Dias and Dias 2010a] extension of Bafna and Pevzner's algorithm [Bafna and Pevzner 1998], and Dias and Dias' [Dias and Dias 2010b] extension of Elias and Hartman's algorithm [Elias and Hartman 2006]) we tested all these algorithms on the same set of arbitrarily large permutations. This set consisted of 59,000 random permutations of sizes varying between 10 and 300 in intervals of 5, with 1,000 permutations of each size.

Figure 2 shows the average distance computed for all algorithms. As can be seen, these data corroborate with the data obtained for small permutations,

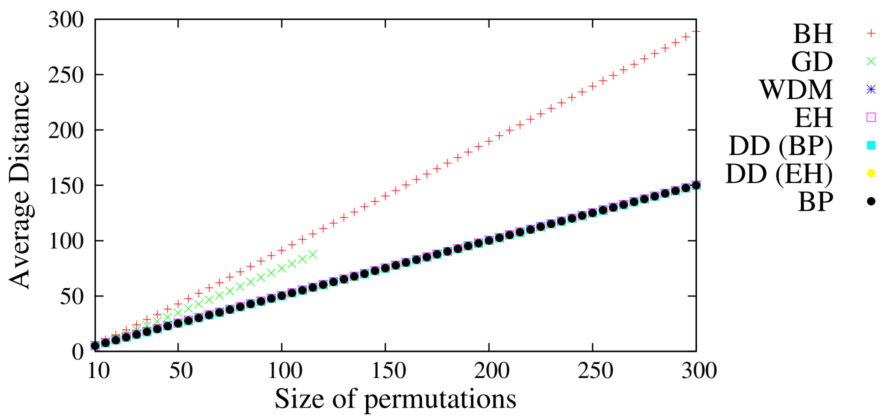

Figure 2: Comparison of Walter, Dias, and Meidanis' algorithm (WDM), Benoît-Gagné and Hamel's algorithm (BH), the constrained version of Guyer, Heath, and Vergara's heuristic (GD), Bafna and Pevzner's algorithm (BP), Elias and Hartman's algorithm (EH), and Dias and Dias' algorithms (DD (BP) and DD (EH)) based on the average distance. Due to time constraints, we could not compute the average distance of the constrained version of Guyer, Heath, and Vergara's heuristic for permutations with more than 115 elements (note that this algorithm runs in $O(n^5 \log n)$ time). The average distances computed for algorithms WDM, EH, DD (BP), DD (EH) and BP were about equal, therefore they are overlapping in the graph.

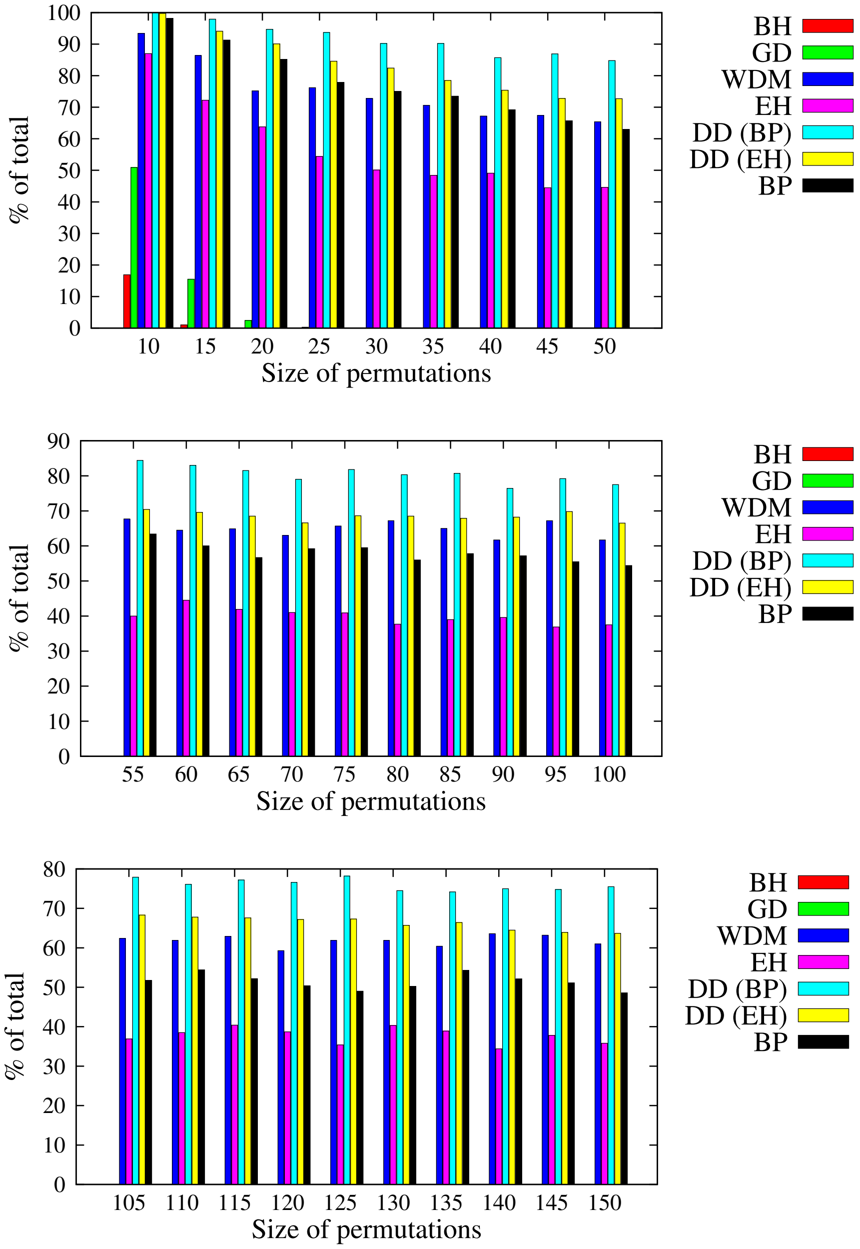

Figure 3: Relative number of times each algorithm provided the best distance. Note that more than one algorithm can have provided the best distance.

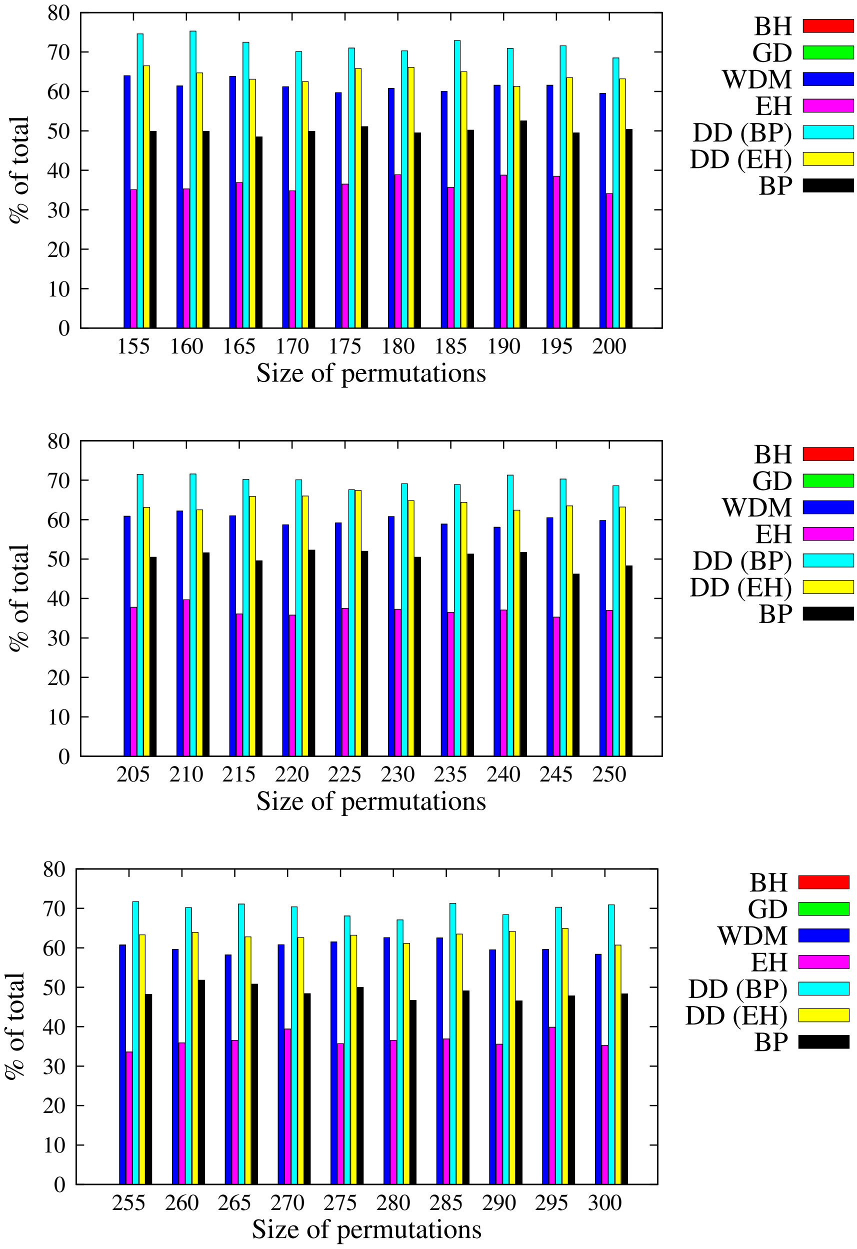

Figure 4: Relative number of times each algorithm provided the best distance. Note that more than one algorithm can have provided the best distance.

that is, of the three algorithms studied in this paper, Walter, Dias, and Meidanis' algorithm has the best practical performance. Figure 2 also shows that Walter, Dias, and Meidanis' algorithm provided results comparable to those provided by the algorithms based on the cycle graph. For the purpose of further verifying how good the algorithms studied in this paper performed in comparison to the algorithms based on the cycle graph, we computed how often each algorithm provided the best distance. The results are presented in figures 3 and 4.

We can notice that the results were consistently the same regardless of the size of the permutations. Benoît-Gagné and Hamel's algorithm and the constrained version of Guyer, Heath, and Vergara's heuristic provided the best distance less times than the other algorithms (for permutations with more than 20 elements, they did not provide the best distance even once). Walter, Dias, and Meidanis' algorithm provided the best distance more times than Bafna and Pevzner's algorithm and Elias and Hartman's algorithm, but less times than Dias and Dias' algorithms. Although Walter, Dias, and Meidanis' algorithm did not outperform Dias and Dias' algorithms, which are the best known algorithms for sorting by transpositions, it is remarkable that it outperformed two approximation algorithms with much better approximation ratios.

6 Conclusions

In this paper, we revisited three algorithms for the problem of sorting by transpositions: Walter, Dias, and Meidanis' 2.25-approximation algorithm [Walter et al. 2000], Benoît-Gagné and Hamel's 3-approximation algorithm [Benoît-Gagné and Hamel 2007], and Guyer, Heath, and Vergara's heuristic [Guyer et al. 1997]. These algorithms are based on alternative approaches to the cycle graph, which is the standard tool for tackling permutation sorting problems.

Regarding theoretical aspects, we closed a missing gap on the proof of the approximation ratio of Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007] and we demonstrated a way to run their algorithm in $O(n \log n)$ time. This latter reinforces Benoît-Gagné and Hamel's argument that, although there does exist better algorithms with respect to approximation ratio, their algorithm is fast. We proposed a minor adaptation to Guyer, Heath, and Vergara's heuristic [Guyer et al. 1997] that allowed us to prove an approximation bound of 3. Finally, with respect to Walter, Dias, and Meidanis' algorithm [Walter et al. 2000], we did not present any theoretical improvement, but we demonstrated that previous experimental data on its approximation ratio are incorrect.

Regarding practical aspects, we performed an experimental investigation of these three algorithms for small and large permutations. For the experiments on small permutations, we considered all permutations with up to 13 elements. To the best of our knowledge, this was the first time these algorithms were tested for all permutations with more that 11 elements. For the experiments on large permutations, we also taken into account approximation algorithms based on the cycle graph, namely Bafna and Pevzner's 1.5-approximation algorithm [Bafna and Pevzner 1998], Elias and Hartman's 1.375-approximation algorithm [Elias and Hartman 2006], Dias and Dias' [Dias and Dias 2010a] extension of Bafna and Pevzner's algorithm [Bafna and Pevzner 1998], and Dias and Dias' [Dias and Dias 2010b] extension of Elias and Hartman's algorithm [Elias and Hartman 2006]. The latter two are the best known algorithms for the problem of sorting by transpositions.

The experimental data yielded by the experiments on small permutations gave some insights on the approximation ratio of the algorithms under study. It indicated that the approximation ratio of Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007] may not be lowered, contradicting a first hypothesis [Benoît-Gagné and Hamel 2007] that it could be, and that the approximation ratio of Walter, Dias, and Meidanis' algorithm [Walter et al. 2000] may be lowered to 2. Unfortunately, we could not obtain any proof regarding the tightness of the approximation ratio of the studied algorithms.

Both the experiments on small and large permutations pointed out Walter, Dias, and Meidanis' algorithm [Walter et al. 2000] as the best algorithm out of the three algorithms based on alternative approaches. Moreover, the experiments on large permutations showed that Walter, Dias, and Meidanis' algorithm [Walter et al. 2000] provided results comparable to the ones provided by the algorithms based on the cycle graph. In fact, Walter, Dias, and Meidanis' algorithm [Walter et al. 2000] outperformed on average both Bafna and Pevzner's algorithm [Bafna and Pevzner 1998] and Elias and Hartman's algorithm [Elias and Hartman 2006], what is remarkable since these algorithms have much better approximation ratios.

We conclude that, although the algorithms based on alternative approaches have worse approximation ratios, Benoît-Gagné and Hamel's algorithm [Benoît-Gagné and Hamel 2007] is a good alternative due to its simplicity and its practical and asymptotic speed, while Walter, Dias, and Meidanis' algorithm [Walter et al. 2000] is a good alternative in terms of practical results. The constrained version of Guyer, Heath, and Vergara's heuristic [Guyer et al. 1997] proposed by us does not figure as a good alternative because it did not present good practical results and it has a prohibitive time complexity, just as the original heuristic.

Although the experimental data on small permutations suggested that none of the studied algorithms are promising alternatives in terms of approximation ratios, it is still not clear whether the approaches they rely on can or cannot yield algorithms with low approximation ratios. Therefore, searching for results that could help make progress on this question either way is an interesting direction to follow for future work.

Acknowledgements

This work was made possible by project fundings from CNPq to Zanoni Dias (numbers 306730/2012-0, 477692/2012-5, and 483370/2013-4). The authors also thank the Center for Computational Engineering and Sciences at Unicamp for financial support through the FAPESP/CEPID Grant 2013/08293-7. FAPESP and CNPq are Brazilian research funding agencies.

References

[Arndt 2010] Arndt, J.: Matters Computational: Ideas, Algorithms, Source Code; Springer (2010), page 235.

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

[Benoît-Gagné and Hamel 2007] Benoît-Gagné, M., Hamel, S.: "A new and faster method of sorting by transpositions"; Proc. CPM'2007; volume 4580 of LNCS, Springer-Verlag, Ontario (2007), 131–141.

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

[Christie 1999] Christie, D. A.: "Genome Rearrangement Problems"; Ph.D. thesis; University of Glasgow (1999).

[Dias and Dias 2010a] Dias, U., Dias, Z.: "Extending Bafna-Pevzner algorithm"; Proc. ISB'2010; ACM Press, Calicut (2010), 1–8.

[Dias and Dias 2010b] Dias, U., Dias, Z.: "An improved 1.375-approximation algorithm for the transposition distance problem"; Proc. BCB'2010; ACM Press, Niagara Falls (2010), 334–337.

[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.

[Fertin et al. 2009] Fertin, G., Labarre, A., Rusu, I., Tannier, E., Vialette, S.: Combinatorics of Genome Rearrangements; The MIT Press (2009).

[Galvão and Dias 2012] G. R. Galvão, Z. Dias.: "On the approximation ratio of algorithms for sorting by transpositions without using cycle graphs"; Proc. BSB'2012; volume 7049 of LNCS, Springer-Verlag, Campo Grande (2012), 25–36.

[Galvão and Dias 2014] Galvão, G. R., Dias, Z.: "An audit tool for genome rearrangement algorithms"; ACM Journal of Experimental Algorithmics; 19 (2014), Article 1.7, 1.1–1.34.

[Gu et al. 1999] Gu, Q., Peng, S., Chen, Q.: "Sorting permutations and its applications in genome analysis"; Lecture Notes on Mathematics in the Life Science; volume 26; 191–201.

[Guyer et al. 1997] Guyer, S. A., Heath, L. S., Vergara, J. P. C.: "Subsequence and run heuristics for sorting by transpositions"; Technical Report TR-97-20; Virginia Polytechnic Institute & State University (1997).

[Hartman and Shamir 2006] Hartman, T., Shamir, R.: "A simpler and faster 1.5-approximation algorithm for sorting by transpositions"; Information and Computation; 204 (2006), 2, 275–290.

[Lehmer 1960] Lehmer, D. H.: "Teaching combinatorial tricks to a computer"; Proceedings of Symposia in Applied Mathematics; 10 (1960), 179–193.

[Walter et al. 2000] Walter, M. E. M. T., Dias, Z., Meidanis, J.: "A new approach for approximating the transposition distance"; Proc. SPIRE'2000; IEEE Computer Society, Washington (2000), 199–208.