Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information and Computation 204 (2006) 275–290
Information and Computation
www.elsevier.com/locate/ic

A simpler and faster 1.5-approximation algorithm for sorting by transpositions ☆

Tzvika Hartman [a,*], Ron Shamir [b,1]

[a] *Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel*
[b] *School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel*

Received 1 March 2005; revised 25 August 2005
Available online 5 December 2005

Abstract

An important problem in genome rearrangements is sorting permutations by transpositions. The complexity of the problem is still open, and two rather complicated 1.5-approximation algorithms for sorting linear permutations are known (Bafna and Pevzner, 98 and Christie, 99). The fastest known algorithm is the quadratic algorithm of Bafna and Pevzner. In this paper, we observe that the problem of sorting circular permutations by transpositions is equivalent to the problem of sorting linear permutations by transpositions. Hence, all algorithms for sorting linear permutations by transpositions can be used to sort circular permutations. Our main result is a new $O(n^{3/2}\sqrt{\log n})$ 1.5-approximation algorithm, which is considerably simpler than the previous ones, and whose analysis is significantly less involved.

Keywords: Computational biology; Genome rearrangements; Sorting permutations by transpositions

1. Introduction

When trying to determine evolutionary distance between two organisms using genomic data, one wishes to reconstruct the sequence of evolutionary events that have occurred, transforming

☆ A preliminary version containing some of the results of this work has appeared in the proceedings of *CPM2003* [13].

* Corresponding author. Fax: +972 8 9459105.
E-mail addresses: tzvi.hartman@weizmann.ac.il (T. Hartman), rshamir@tau.ac.il (R. Shamir).
[1] Fax: +972 3 640 5384.

0890-5401/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.ic.2005.09.002

one genome into the other. One of the most promising ways to trace the evolutionary events is to compare the order of appearance of identical (or orthologous) genes in two different genomes. In the 1980s, evidence was found that certain species have essentially the same set of genes, but their gene order differs [21,16]. This suggests that global rearrangement events (such as reversals and transpositions of genome segments) can be used to trace the evolutionary path between genomes. Such rare events may provide more accurate and robust clues to the evolution than local point mutations (i.e., insertions, deletions, and substitutions of nucleotides).

In the last decade, a large body of work was devoted to genome rearrangement problems. Genomes are represented by permutations, where each element stands for a gene. Circular genomes (such as bacterial and mitochondrial genomes) are represented by circular permutations. The basic task is, given two permutations, to find a shortest sequence of rearrangement operations that transforms one permutation into the other. Assuming that one of the permutations is the identity permutation, the problem is to find the shortest way of sorting a permutation using a given rearrangement operation (or set of operations). For more background on genome rearrangements refer to [24,22,23,25].

The problem of sorting permutations by reversals has been studied extensively. It was shown to be NP-hard [6], and several approximation algorithms have been suggested [2,7,5]. On the other hand, for signed permutations (every element of the permutation has a sign, + or −, which represents the direction of the gene), a polynomial algorithm for sorting by reversals was first given by Hannenhalli and Pevzner [12]. Subsequent work improved the running time of the algorithm, and simplified the underlying theory [17,4,1]. The problems of sorting signed permutations by reversals were shown to be equivalent for linear and circular permutations [20].

There has been less progress on the problem of sorting by transpositions. A transposition is a rearrangement operation, in which a segment is cut out of the permutation, and pasted in a different location. The complexity of sorting by transpositions is still open. It was first studied by Bafna and Pevzner [3], who devised a rather complicated 1.5-approximation algorithm, which runs in quadratic time. Christie [8] gave a somewhat simpler $O(n^4)$ algorithm with the same approximation ratio. An $O(n^3)$ implementation of this algorithm, along with heuristics that improve its performance, were given in [28]. Eriksson et al. [10] provided an algorithm that sorts any given permutation on n elements by at most $2n/3$ transpositions, but has no approximation guarantee. The problem of sorting by both reversals and transpositions was addressed in [27,11,19,14].

In this paper, we study the problem of sorting permutations by transpositions. First, we prove that the problem of sorting circular permutations by transpositions is equivalent to the problem of sorting linear permutations by transpositions. Hence, all algorithms for sorting linear permutations by transpositions can be used to sort circular permutations. Then, we derive our main result: a new $O(n^{3/2}\sqrt{\log n})$ 1.5-approximation algorithm, which is considerably simpler than the previous ones [3,8], and achieves better running time. Moreover, the analysis of the algorithm is significantly less involved, and provides a good starting point for studying related open problems. The improvement in the running time of the algorithm is achieved by exploiting an efficient data structure introduced by Kaplan and Verbin [18] in the context of sorting by reversals.

The paper is organized as follows. In Section 2, we first prove the equivalence between the problem of sorting linear and circular permutations by transpositions. Then, we review some classical genome rearrangement results, and show that every permutation can be transformed into a so-called simple permutation. Our main result, a new and simple quadratic 1.5-approximation algorithm for

sorting permutations by transpositions, is introduced in Section 3. In Section 4, we describe the efficient data structure that allows an $O(n^{3/2}\sqrt{\log n})$ implementation of the algorithm. We conclude in Section 5 with a short discussion and some open problems.

2. Preliminaries

2.1. Linear and circular permutations

Let $\pi = [\pi_1 \dots \pi_n]$ be a permutation on n elements. Define a *segment* A in π as a consecutive sequence of elements $\pi_i, \dots, \pi_k$ $(k \geqslant i)$. Two segments $A = \pi_i, \dots, \pi_k$ and $B = \pi_j, \dots, \pi_l$ are *contiguous* if $j = k + 1$ or $i = l + 1$. A *transposition* τ on π is the exchange of two disjoint contiguous segments (Fig. 1a). If the segments are $A = \pi_i, \dots, \pi_{j-1}$ and $B = \pi_j, \dots, \pi_{k-1}$, then by performing τ on π, the resulting permutation, denoted $\tau \cdot \pi$, is $[\pi_1 \dots \pi_{i-1} \pi_j \dots \pi_{k-1} \pi_i \dots \pi_{j-1} \pi_k \dots \pi_n]$ (note that the end segments can be empty if $i = 1$ or $k - 1 = n$). We shall say that τ *cuts* π *before* positions i, j, and k. We say that τ *involves* index l if $i \leqslant l < k$, i.e., if l belongs to one of the two exchanged segments.

In circular permutations, one can define a transposition analogously as the exchange of two contiguous segments. Note that here the indices are cyclic, so the disjointness of the exchanged segments is a meaningful requirement. The transposition partitions a circular permutation into three segments, as opposed to at most four in a linear permutation (see Fig. 1). Note that for circular permutations, we can assume w.l.o.g. that all three segments are non-empty, since otherwise the original and transformed permutations are the same. Since there are only two cyclic orders on three segments, and each two of the three segments are contiguous, the transposition can be represented by exchanging any two of them. Note that the number of possible transpositions on a linear n-permutation is $\binom{n+1}{3}$, since there are $n + 1$ possible cut points of segments. In contrast, in a circular n-permutation there are only $\binom{n}{3}$ possibilities.

The problem of finding a shortest sequence of transpositions, which transforms a (linear or circular) permutation into the identity permutation, is called *sorting by transpositions*. The

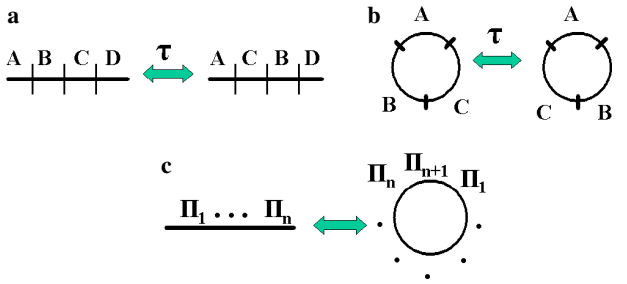

Fig. 1. (a) A transposition τ, which is applied on a linear permutation, and exchanges segments B and C. (b) A transposition τ, which is applied on a circular permutation. τ can be viewed as exchanging A and B, or B and C, or A and C. (c) A one-to-one transformation between linear and circular permutations. In the circular permutation, a new element, π_{n+1}, is introduced.

transposition distance of a permutation π, denoted by $d(\pi)$, is the length of the shortest sorting sequence.

Theorem 1. *The problem of sorting linear permutations by transpositions is linearly equivalent to the problem of sorting circular permutations by transpositions.*

Proof. Given a linear n-permutation, circularize it by adding an $n+1$'st element $\pi_{n+1} = x$, and closing the circle (see Fig. 1c). Call the new circular permutation π^c. By the discussion above, any transposition on π^c can be represented by the two segments that do not include x. Hence, there is an optimal sequence of transpositions that sorts π^c, and none of them involves x. The same sequence can be viewed as a sequence of transpositions on the linear permutation π, by ignoring x. This implies that $d(\pi) \leqslant d(\pi^c)$. On the other hand, any sequence of transpositions on π is also a sequence of transpositions on π^c, so $d(\pi^c) \leqslant d(\pi)$. Hence, $d(\pi) = d(\pi^c)$. Moreover, an optimal sequence for π^c provides an optimal sequence for π.

For the other direction, starting with a circular permutation, we can linearize it by removing an arbitrary element, which plays a role of x above (see Fig. 1c). The same arguments imply that an optimal solution for the linear permutation translates to an optimal solution for the circular one. □

In the rest of the paper, we will discuss only circular permutations. As implied by Theorem 1, all the results on circular permutations hold also for linear ones. We prefer to work with circular permutations since it simplifies the analysis.

2.2. The circular breakpoint graph

We transform a permutation π on n elements into a permutation $f(\pi)$ on $2n$ elements, by replacing each element i by two elements $2i-1, 2i$. On the doubled permutation $f(\pi)$, we allow only transpositions that cut before odd positions. This ensures that no transposition cuts between $2i-1$ and $2i$, and therefore every transposition on π can be mimicked by a transposition on $f(\pi)$. We call such transpositions *legal*. We now define the circular breakpoint graph, which is the circular version of the breakpoint graph [2]. Throughout, in both indices and elements, we identify $2n+1$ and 1.

Definition 1. Let $\pi = (\pi_1 \dots \pi_n)$ be a circular permutation, and $f(\pi) = \pi' = (\pi'_1 \dots \pi'_{2n})$. The *breakpoint graph* $G(\pi)$ is an edge-colored graph on $2n$ vertices $\{1, 2, \dots, 2n\}$. For every $1 \leqslant i \leqslant n$, π'_{2i} is joined to π'_{2i+1} by a black edge (denoted by b_i), and $2i$ is joined to $2i+1$ by a gray edge.

Note that unlike previous studies of transpositions [3,8], we chose to double the number of vertices and work with an undirected graph, as done in the signed case [2]. It is convenient to draw the breakpoint graph on a circle, such that black edges are on the circumference and gray edges are chords (see Fig. 2A). We shall use this representation throughout the paper.

Since the degree of each vertex is exactly 2, the graph uniquely decomposes into cycles. Denote the number of cycles in $G(\pi)$ by $c(\pi)$. The *length* of a cycle is the number of black edges it contains. A *k-cycle* is a cycle of length k, and it is *odd* if k is odd. The number of odd cycles is denoted by $c_{\text{odd}}(\pi)$. Define $\Delta c(\pi, \tau) = c(\tau \cdot \pi) - c(\pi)$, and $\Delta c_{\text{odd}}(\pi, \tau) = c_{\text{odd}}(\tau \cdot \pi) - c_{\text{odd}}(\pi)$.

Bafna and Pevzner proved the following useful lemma (This—and other results we quote—was proved for linear permutations, but holds also for circular ones).

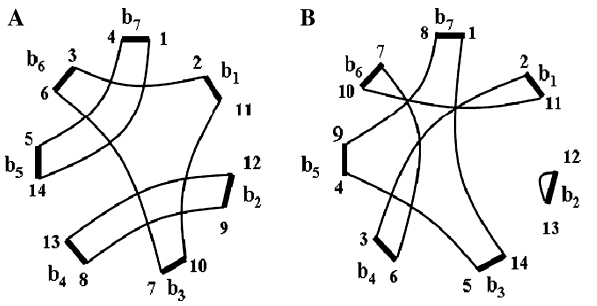

Fig. 2. (A) The circular breakpoint graph of the permutation $\pi = (1\ 6\ 5\ 4\ 7\ 3\ 2)$, for which $f(\pi) = (1\ 2\ 11\ 12\ 9\ 10\ 7\ 8\ 13\ 14\ 5\ 6\ 3\ 4)$. Black edges are represented as thick lines on the circumference, and gray edges are chords. (B) The circular breakpoint of π after applying the transposition that acts on black edges b_2, b_4, and b_7.

Lemma 2 (Bafna and Pevzner [3]). *For all permutations π and transpositions τ, it holds that $\Delta c(\pi, \tau) \in \{-2, 0, 2\}$, and $\Delta c_{\text{odd}}(\pi, \tau) \in \{-2, 0, 2\}$.*

Let $n(\pi)$ denote the number of black edges in $G(\pi)$. The maximum number of cycles is obtained iff π is the identity permutation. In that case, there are $n(\pi)$ cycles, and all of them are odd (in particular, they are all of length 1) . Starting with π with c_{odd} odd cycles, Lemma 2 implies the following lower bound on $d(\pi)$.

Theorem 3 (Bafna and Pevzner [3]). *For all permutations π, $d(\pi) \geqslant (n(\pi) - c_{\text{odd}}(\pi))/2$.*

By definition, every legal transposition must cut three black edges. The transposition that cuts black edges b_i, b_j, and b_k is said to *act on* these edges (see Fig. 2B). A transposition τ is a *k-transposition* if $\Delta c_{\text{odd}}(\pi, \tau) = k$. A cycle is called *oriented* if there is a 2-transposition that acts on three of its black edges; otherwise, it is *unoriented.*

Observation 4. *There are only two possible configurations of 3-cycles that can be obtained by legal transpositions.*

The two possibilities are shown in Fig. 3. It is easy to verify that the left 3-cycle is unoriented, and the right one is oriented.

Given a cyclic sequence of elements $i_1, \ldots, i_k$, an *arc* is an interval in the cyclic order, i.e., a set of contiguous elements in the sequence. The pair (i_j, i_l) $(j \neq l)$ defines two disjoint arcs: $i_j, \ldots, i_{l-1}$ and $i_l, \ldots, i_{j-1}$. Similarly, a triplet defines a partition of the cycle into three disjoint arcs. We say that two pairs of black edges (a, b) and (c, d) are *intersecting* if a and b belong to different arcs defined by the pair (c, d). A pair of black edges intersects with cycle C, if it intersects with a pair of black

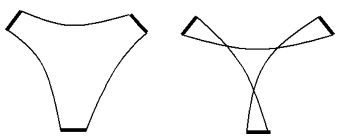

Fig. 3. The only two possible configurations of 3-cycles. The left one is unoriented, and the right one is oriented.

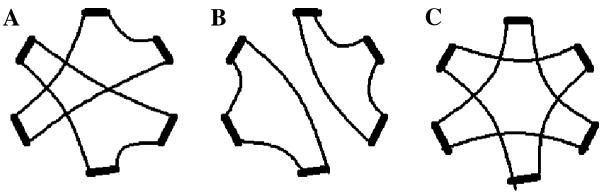

Fig. 4. (A) Intersecting 3-cycles. (B) Non-intersecting 3-cycles. (C) Interleaving 3-cycles.

edges that belong to C. Cycles C and D intersect if there is a pair of black edges in C that intersect with D (see Fig. 4A). Two triplets of black edges are *interleaving* if each of the edges of one triple belongs to a different arc of the second triple. Two 3-cycles are interleaving if their edges interleave (see Fig. 4C).

Throughout the paper, we use the term permutation also when referring to the breakpoint graph of the permutation (as will be clear from the context). For example, when we say that π contains an oriented cycle, we mean that $G(\pi)$ contains an oriented cycle.

2.3. Transformation into equivalent simple permutations

A k-cycle in the breakpoint graph is called *short* if $k \leqslant 3$; otherwise, it is called *long*. A breakpoint graph is called *simple* if it contains only short cycles. A permutation π is called *simple* if $G(\pi)$ is simple. Following [12,19], we show how to transform an arbitrary permutation into a simple one, while maintaining the lower bound of Theorem 3.

Let $b = (v_b, w_b)$ be a black edge and let $g = (v_g, w_g)$ be a gray edge belonging to the same cycle $C = (\ldots, v_b, w_b, \ldots, w_g, v_g, \ldots)$ in $G(\pi)$. A (g, b)-*split* of $G(\pi)$ is a sequence of operations on $G(\pi)$, resulting in a new graph $\hat{G}(\pi)$ with one more cycle, as follows:

- Removing edges b and g.
- Adding two new vertices v and w.
- Adding two new black edges (v_b, v) and (w, w_b).
- Adding two new gray edges (w_g, w) and (v, v_g).

Fig. 5 shows a (g, b)-split transforming a cycle C in $G(\pi)$ into two cycles C_1 and C_2 in $\hat{G}(\pi)$. Note that the order of the nodes of each edge along the cycle is important, as other orders may not split the cycle. Hannenhalli and Pevzner [12] show that for every (g, b)-split on a permutation π of n elements, there is a permutation $\hat{\pi}$ of $n + 1$ elements, that is obtained by inserting an element into

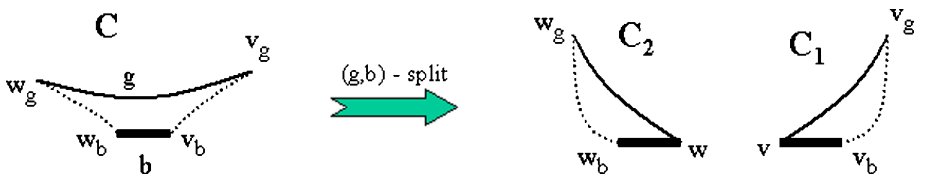

Fig. 5. A (g, b)-split. A dashed line indicates a path.

π, such that $\hat{G}(\pi) = G(\hat{\pi})$. Thus, a (g,b)-split can be viewed as a transformation from π to $\hat{\pi}$. A (g,b)-split is called *safe* if $n(\pi) - c_{\text{odd}}(\pi) = n(\hat{\pi}) - c_{\text{odd}}(\hat{\pi})$, i.e., if it maintains the lower bound of Theorem 3.

Lemma 5 (Lin and Xue [19]). *Every permutation can be transformed into a simple one by safe splits.*

Proof. Let π be a permutation that contains a long cycle C. Let b_1 be a black edge in C. Denote by b_2 and b_3 the black edges that are connected to b_1 via a gray edge. Let g be the gray edge that is connected to b_2 but not to b_1. Then a (g,b_3)-split breaks C into a 3-cycle and a $(k-2)$-cycle in $\hat{\pi}$. Clearly, $n(\hat{\pi}) = n(\pi) + 1$, and $c_{\text{odd}}(\hat{\pi}) = c_{\text{odd}}(\pi) + 1$, so the split is safe. This process can be repeated until a simple permutation is eventually obtained. □

We say that permutation π is *equivalent* to permutation $\hat{\pi}$ if $n(\pi) - c_{\text{odd}}(\pi) = n(\hat{\pi}) - c_{\text{odd}}(\hat{\pi})$.

Lemma 6 (Hannenhalli and Pevzner [12]). *Let $\hat{\pi}$ be a simple permutation that is equivalent to π, then every sorting of $\hat{\pi}$ mimics a sorting of π with the same number of operations.*

In the following, we show how to sort a simple permutation by transpositions. We prove that the number of transpositions is within a factor of 1.5 from the lower bound of Theorem 3. Thus, we obtain a 1.5-approximation algorithm for sorting simple permutations. The above discussion implies that this algorithm translates into a 1.5-approximation algorithm for an arbitrary permutation: Transform the permutation into an equivalent simple permutation (Lemma 5), sort it, and then mimic the sorting on the original permutation (Lemma 6).

3. The algorithm

In this section, we provide a 1.5-approximation algorithm for sorting permutations by transpositions. We first develop an algorithm for simple permutations, and then use the results of Section 2.3 to prove the general case. Recall that the breakpoint graph of a simple permutation contains only 1-, 2- and 3-cycles. Our goal is to obtain a graph with 1-cycles only, which is the breakpoint graph of the identity permutation. Thus, the sorting can be viewed as a process of transforming the 2- and 3-cycles into 1-cycles.

First we deal with the case that the permutation contains a 2-cycle:

Lemma 7 (Christie [8]). *If π is a permutation that contains a 2-cycle, then there exists a 2-transposition on π.*

By definition, an oriented 3-cycle can be eliminated by a 2-transposition that acts on its black edges. Suppose from now on that all 2-cycles were eliminated by applying Lemma 7, and all oriented 3-cycles were eliminated. The only remaining problem is how to handle unoriented 3-cycles. This is the case we analyze henceforth.

A *(0,2,2)-sequence* is a sequence of three transpositions, of which the first is a 0-transposition, and the next two are 2-transpositions. Note that a $(0,2,2)$-sequence increases the number of odd cycles by 4 out of 6 that are the maximum possible in 3 steps, and thus a series of $(0,2,2)$-sequences preserves a 1.5 approximation ratio. We shall show below that such a sequence is always possible.

Lemma 8. *Let π be a permutation that contains two interleaving unoriented 3-cycles. Then, there exists a $(0,2,2)$-sequence of transpositions on π.*

Proof. The $(0,2,2)$-sequence is described in Fig. 6. □

Lemma 9. *Let C and D be two intersecting unoriented 3-cycles that are not interleaving. Then, there exists a transposition which transforms C and D into a 1-cycle and an oriented 5-cycle.*

Proof. Let c_1, c_2, and c_3 be the three black edges of C. Assume, without loss of generality, that (c_1, c_2) intersects with D. We shall in fact prove a stronger statement, namely, for any choice of a black edge $d \in D$ such that (d, c_3) intersects with (c_1, c_2), the transposition on c_1, c_2, and d satisfies the lemma. There are three possible cases to consider, which are shown in Fig. 7. In each case, the

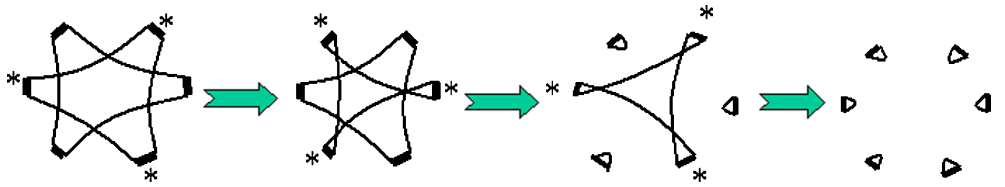

Fig. 6. A $(0,2,2)$-sequence of transpositions for two interleaving unoriented 3-cycles. At each step the transposition acts on the three black edges that are marked by a star.

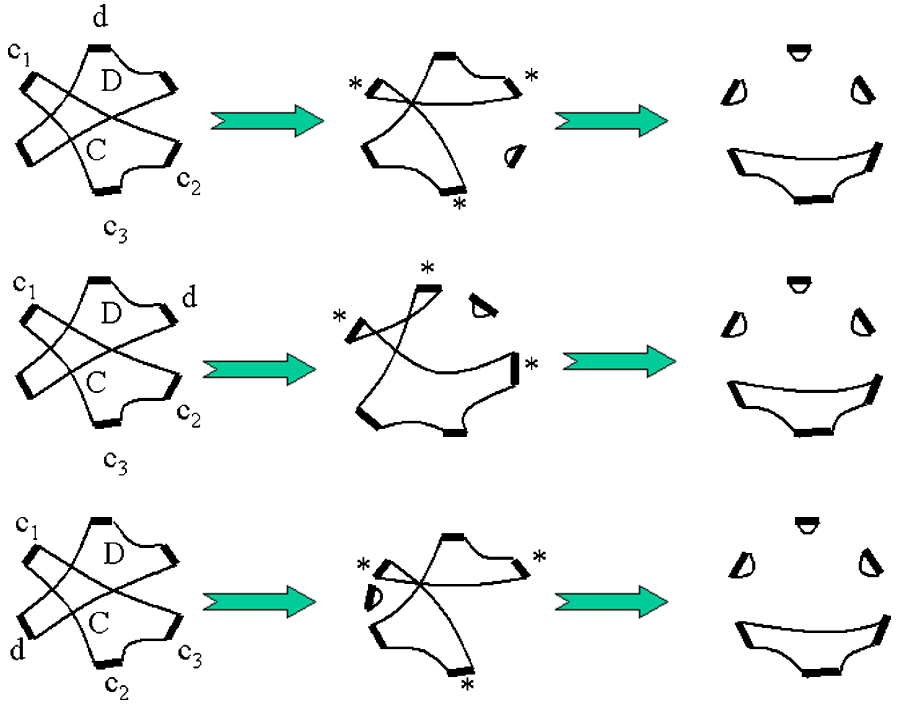

Fig. 7. The three possible cases of two intersecting unoriented 3-cycles that are not interleaving. In each case, the transposition that acts on edges c_1, c_2, and d, transforms C and D into a 1-cycle and an oriented 5-cycle.

first transposition, which acts on c_1, c_2, and d, transforms 3-cycles C and D into a 1-cycle and a 5-cycle. Then, to show that the 5-cycle is oriented, a 2-transposition which acts on three of its edges is shown. □

We say that cycle E is *shattered* by cycles C and D if every pair of edges in E intersects with a pair of edges in C or with a pair of edges in D (see Fig. 8).

Lemma 10. *Let π be a permutation that contains three unoriented 3-cycles C, D, and E, such that E is shattered by C and D. Then, there exists a $(0,2,2)$-sequence of transpositions on π.*

Proof. If two of the three cycles are interleaving, the (0,2,2)-sequence follows from Lemma 8. Otherwise, there are two general cases:

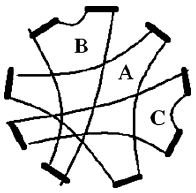

Fig. 8. Cycle shattering. Cycle A is shattered by B and C. Cycle C is not shattered by A and B.

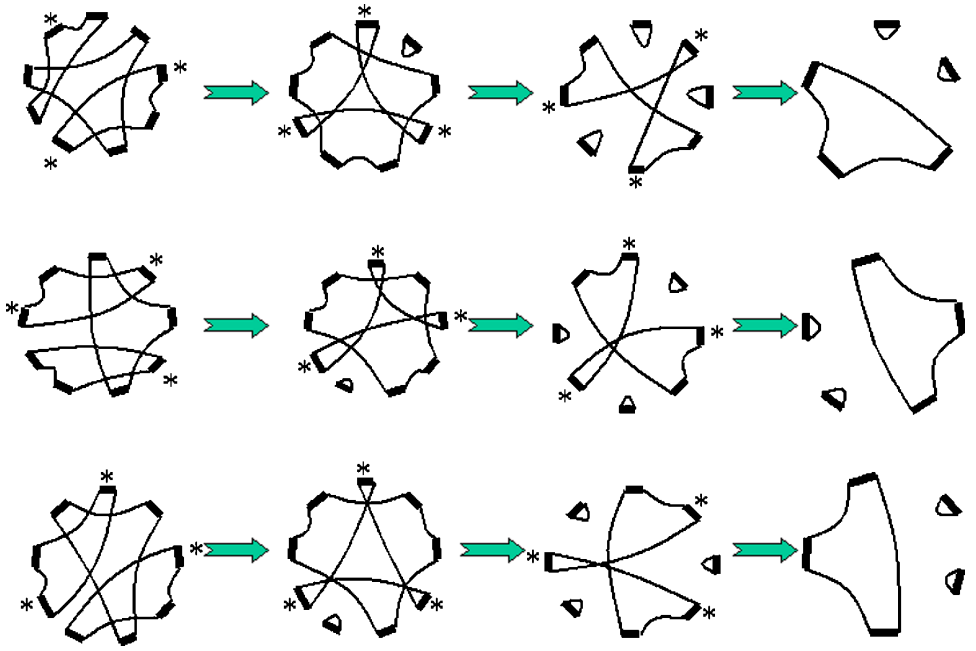

Fig. 9. The three possible cases of three unoriented 3-cycles, such that one of them is shattered by the other two, no pair is interleaving and two of them are non-intersecting. In each case, a $(0,2,2)$-sequence of transpositions is shown. For simplicity, every 1-cycle is shown only when it is formed and not in subsequent graphs (since it is not affected by transpositions in later steps).

(1) Two out of the three cycles are non-intersecting. In this case, there are three possible configurations of the cycles, which are shown in Fig. 9. For every sub-case, a $(0,2,2)$-sequence is shown.
(2) The three cycles are mutually intersecting. The general case is illustrated in Fig. 10. Since cycles C and D are unoriented, the condition of the proof of Lemma 9 is fulfilled. Thus, we can apply a 0-transposition that acts on edges c_1, c_2, and d, and obtain a new oriented cycle F. Now we apply a 2-transposition on E (which has also become oriented). Cycle F remains oriented, since the latter transposition does not change its structure. Thus, another 2-transposition is possible on the edges of F, which completes the $(0,2,2)$-sequence. □

A pair of black edges is said to be *connected* if they are connected by a gray edge.

Lemma 11 (Bafna and Pevzner [3]). *Let (b_i, b_j) be a connected pair in an unoriented cycle. Then, (b_i, b_j) intersects with some other cycle.*

We are now ready to present the full algorithm. It is described in Fig. 11. Note that in steps 2–3 it is impossible to create a long cycle, and thus the permutation remains simple throughout the

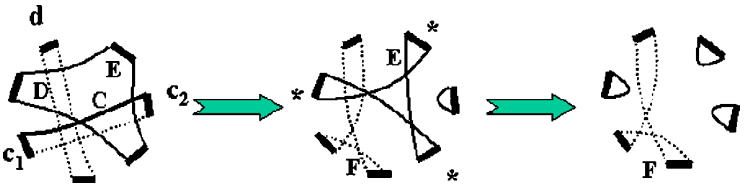

Fig. 10. Three mutually intersecting unoriented cycles such that no pair is interleaving, and one is shattered by the other two. A dashed line represents either a single gray edge or a path of length 3. Note that edges c_1 and c_2 are connected by a single gray edge.

Algorithm *Sort* (π)

(1) Transform permutation π into an equivalent simple permutation $\hat{\pi}$ (Lemma 5).
(2) While $G(\hat{\pi})$ contains a 2-cycle, apply a 2-transposition (Lemma 7).
(3) While $G(\hat{\pi})$ contains a 3-cycle C, do:
- Pick a connected pair of black edges c from cycle C. If C is oriented - apply a 2-transposition.
- Otherwise, pick a connected pair d (from some cycle D) that intersects with c (the existence of d is guaranteed by Lemma 11). If C and D are interleaving - apply a $(0,2,2)$-sequence (Lemma 8).
- Otherwise, there is a connected pair c' in C that does not intersect with D. Pick a connected pair e (from some cycle E) that intersects with c' (the existence of e is guaranteed by Lemma 11). As cycle C is shattered by cycles D and E, apply a $(0,2,2)$-sequence (Lemma 10).

(4) Mimic the sorting of π using the sorting of $\hat{\pi}$ (Lemma 6).

Fig. 11. 1.5-approximation algorithm for sorting by transpositions.

algorithm. Note also that in step 3 we do not create 2-cycles, and hence, there is no need to iterate over step 2.

Performing only steps 2–3 is an algorithm in its own, and is denoted by *Algorithm SortSimple*. The following lemma claims that SortSimple is a quadratic time 1.5-approximation algorithm for sorting simple permutations.

Lemma 12. *Algorithm SortSimple is a 1.5-approximation algorithm for simple permutations, and it runs in time* $O(n^2)$.

Proof. The sequence of transpositions that is generated by the algorithm contains only 2-transpositions and $(0,2,2)$-sequences of transpositions. Therefore, every sequence of three transpositions increases the number of odd cycles by at least 4 out of 6 possible in 3 steps (as implied from the lower bound of Theorem 3). Hence, the approximation ratio is 1.5.

We now analyze the running time of the algorithm. Step 2 can be done in linear time. The number of iterations in step 3 is linear, since every iteration we break a 3-cycle into three 1-cycles. The main operations in each iteration is to find a connected pair that intersects with a given pair, and to apply a transposition (the other operations can be done in constant time). These operations can be done trivially in linear time. Hence, the algorithm is quadratic. □

Now we are ready to prove the correctness of Algorithm *Sort*.

Theorem 13. *Algorithm Sort is a 1.5-approximation algorithm for general permutations, and it runs in time* $O(n^2)$.

Proof. By Lemma 12, we are guaranteed that $alg(\hat{\pi}) \leqslant 1.5 \cdot d(\hat{\pi})$, where $alg(\hat{\pi})$ is the number of transpositions used by Algorithm SortSimple to sort $\hat{\pi}$. Thus, by Theorem 3,

$$alg(\hat{\pi}) \leqslant 1.5d(\hat{\pi}) \leqslant 1.5\left(\frac{n(\hat{\pi}) - c_{\text{odd}}(\hat{\pi})}{2}\right) = 1.5\left(\frac{n(\pi) - c_{\text{odd}}(\pi)}{2}\right) \leqslant 1.5 \cdot d(\pi)$$

Using Lemma 6, we can sort π by $alg(\hat{\pi})$ transpositions, which implies an approximation ratio of 1.5.

Since steps 1 and 4 can be done in linear time, Lemma 12 implies that the running time of Algorithm Sort is $O(n^2)$. □

4. An $O(n^{3/2}\sqrt{\log n})$ implementation of the algorithm

In this section, we exploit a special data structure in order to speed-up the algorithm. As discussed in the proof of Lemma 12, the main operations in each iteration of the algorithm are finding an arbitrary connected pair that intersects with a given connected pair, and applying a transposition. In the sequel, we describe a data structure that allows to perform these operations in sub-linear time. This data structure is similar to the one introduced by Kaplan and Verbin [18], although here the required operations are slightly different. For completeness we give here a full description of the data structure.

By Theorem 1, the data structure can be presented for linear permutations (we prefer doing that since it makes the presentation clearer). We consider the doubled permutation $f(\pi)$ (see Section

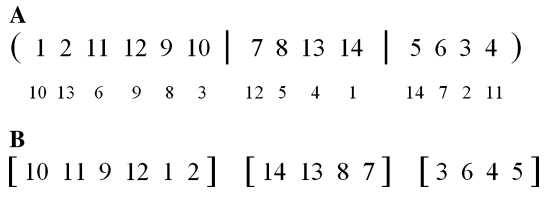

Fig. 12. (A) Partition of the permutation $\pi = (1\ 2\ 11\ 12\ 9\ 10\ 7\ 8\ 13\ 14\ 5\ 6\ 3\ 4)$ into blocks. Below each element, the location of its mate in π is indicated. (B) The internal order of each block (according to the order of the mates in π).

2.2), which is denoted here simply by π. A connected pair of black edges (b_1, b_2) is represented by the pair $(2i, 2i+1)$ which corresponds to the gray edge that connects b_1 and b_2. Thus, π is a union of disjoint pairs. Two elements that form a pair are called *mates*. We need a data structure that supports the following operations in sub-linear time:

- *IntersectionQuery*(π, e_1, e_2): Find a pair that intersects in π with the pair of elements (e_1, e_2).
- *Transp*(π, e_1, e_2, e_3): Apply on π a transposition that cuts before elements e_1, e_2 and e_3.

A query is said to *act* on the elements e_1 and e_2. Similarly, a transposition acts on elements e_1, e_2, and e_3. The *location* of an element e is its number in a left-to-right ordering of π, and is denoted by $loc(e)$.

Now we describe the data structure. The permutation π is divided into $\Theta(\sqrt{\frac{n}{\log n}})$ blocks of size $\Theta(\sqrt{n \log n})$ each, which are maintained in a list. The elements in each block are ordered according to the order of their mates in π (see example in Fig. 12). Attached to each block is a splay tree [26] in which the elements of the block are maintained. This data structure is a balanced binary search tree that is re-balanced via rotations, and supports split and concatenate operations in logarithmic time. [1] We also maintain a lookup table that contains for each element a pointer to its block.

For simplicity, we assume that queries and transpositions act only on elements that are on block boundaries (Lemma 15 will show why we can make this assumption). More specifically, assume that e_1 and e_2 (for transpositions also e_3) are all first elements in their blocks.

Lemma 14. *Operations IntersectionQuery and Transp can be performed in time* $O(\sqrt{n \log n})$, *assuming that they act only on elements that are on block boundaries.*

Proof.

- Intersection Query(π, e_1, e_2): Let B_1 and B_2 be the blocks that contain e_1 and e_2, respectively, (these blocks are found by using the lookup table) and assume w.l.o.g that B_1 is located before B_2. For each block that is before B_1 or after B_2 do the following. Find, by binary search, the first element that is located after $loc(e_1)$ and the element that is located right before $loc(e_2)$. Split

[1] As in [18], we use splay trees since the implementation of the split and concatenate operations is very elegant. Therefore, our running times will be amortized. Amortization can be avoided by using some other type of search tree.

the corresponding tree in the locations of these two elements, and consider the subtree that is bounded by these two elements. If this subtree is not empty, then pick an arbitrary element in it. By construction, this element and its mate are intersecting with (e_1, e_2), i.e., the query is answered. Otherwise, continue to the next block. The split operation is done in logarithmic time. Since there are $O(\sqrt{\frac{n}{\log n}})$ blocks of size $\Theta(\sqrt{n \log n})$, the total time is $O(\sqrt{n \log n})$.

- Transposition(π, e_1, e_2, e_3):
  - Let B_1, B_2, and B_3 be the blocks that contain e_1, e_2, and e_3. Apply a straightforward transposition on the permutation of all blocks, that acts on B_1, B_2, and B_3. Time: $O(\sqrt{\frac{n}{\log n}})$.
  - The order of the elements in π is changed and since the elements in each block are ordered according to the order of their mates in π, we must update the order of the elements in each block. For each block, split the corresponding tree in the three elements that are located eight before e_1, e_2, and e_3, coming up with four trees which are denoted by T_1, T_2, T_3, and T_4, respectively. Now concatenate the trees in the order T_1, T_3, T_2, and T_4, yielding the new permutation. Split and concatenation are logarithmic operations that are applied to $O(\sqrt{\frac{n}{\log n}})$ blocks of size $\Theta(\sqrt{n \log n})$ and therefore, the total time is $O(\sqrt{n \log n})$. □

The following lemma is based on [18], and shows why we can assume that all operations act only on elements that are at block boundaries.

Lemma 15. *Suppose that it is possible to perform IntersectionQuery and Transp in time* $O(\sqrt{n \log n})$, *assuming that the operations act only on block boundaries. Then, it is possible to perform these operations with the same time complexity, even if they act on arbitrary elements.*

Proof. The idea is to (1) add for each operation a pre-processing step that splits up to three blocks, such that in the new partition of blocks the operation acts only on boundaries of blocks; (2) apply procedures *IntersectionQuery* and *Transp* on these blocks, and (3) add a post-processing step that ensures that at the end of the operation the blocks are of size between $\frac{\sqrt{n \log n}}{2}$ and $2\sqrt{n \log n}$, and hence, there are still $\Theta(\sqrt{n \log n})$ blocks. We now describe steps (1) and (3), and show that they can be performed in time $O(\sqrt{n \log n})$.

- Pre-processing: Splitting a block is done by splitting the corresponding splay tree in the appropriate location(s), and updating the lookup table accordingly. The splitting locations are the elements that are right before $loc(e_i)$ for $i = 1, 2, 3$, and are found by binary search (for queries we split in two locations). The number of splits is at most three, each split is logarithmic and the number of elements to update is $O(\sqrt{n \log n})$. Thus, this step can be done in time $O(\sqrt{n \log n})$.
- Post-processing: Due to the pre-processing step, there may be up to six blocks that are smaller than $\frac{\sqrt{n \log n}}{2}$. If there is such a block, we first concatenate it to the preceding block (if it is the first block, we concatenate to the last one). Now, it is possible that a block that is bigger than $2\sqrt{n \log n}$ is created. However, since the size of this new block is bounded by $2.5\sqrt{n \log n}$, another single split in this block ensures that the new blocks are of legal size. Concatenating two blocks is done by concatenating the corresponding splay trees, and updating the lookup table accordingly.

Since the total number of splits and concatenations in this step is constant and the number of updates is $O(\sqrt{n \log n})$, it can be performed in time $O(\sqrt{n \log n})$. □

By combining Lemmas 14 and 15 we get.

Corollary 16. *Step 3 of Algorithm Sort can be implemented in time* $O(\sqrt{n \log n})$.

Now we are ready for the main result of this section .

Theorem 17. *Algorithm Sort guarantees a 1.5-approximation ratio to sorting by transpositions, and runs in time* $O(n^{3/2}\sqrt{\log n})$.

Proof. The number of iterations in the algorithm is at most n. By Corollary 16, each iteration can be implemented in time $O(\sqrt{n \log n})$. Thus, the whole algorithm runs in time $O(n^{3/2}\sqrt{\log n})$. □

5. Discussion and subsequent work

In this paper, we studied the problem of sorting permutations by transpositions, simplified the underlying theory, and presented a $O(n^{3/2}\sqrt{\log n})$ 1.5-approximation algorithm for the problem . We believe that this is an important step towards solving some related open problems. The main open problem is to determine the complexity of sorting by transpositions. Devising algorithms with better approximation ratio and/or faster running time is also desirable. Another direction, which is more biologically relevant, is to consider algorithms for sorting permutations by a set of rearrangement operations (such as reversals, transpositions and translocations).

An implementation of our algorithm and a comparison to the previous ones was done in [15,29]. Hartman and Sharan [14] considered the related problem of sorting by transpositions and transreversals (a rearrangement operation which is a combination a transposition and a reversal). Based on the simplified theory described above, they developed a 1.5-approximation algorithm for the problem. The approximation ratio of sorting by transpositions was recently improved to 1.375 by Elias and Hartman [9].

Acknowledgments

Special thanks to Haim Kaplan, who pointed out the applicability of the data structure introduced in [18] for our algorithm, and to Elad Verbin for helpful discussions on this data structure. We thank Roded Sharan for fruitful discussions, and Vineet Bafna for help in understanding the complexity of the Bafna–Pevzner Algorithm [3]. This work was supported in part by the Israel Science Foundation (grant 309/02).

References

[1] D.A. Bader, B.M.E. Moret, M. Yan, A linear-time algorithm for computing inversion distance between signed permutations with an experimental study, J. Comput. Biol. 8 (5) (2001) 483–491.

[2] V. Bafna, P.A. Pevzner, Genome rearragements and sorting by reversals, SIAM J. Comput. 25 (2) (1996) 272–289.
[3] V. Bafna, P.A. Pevzner, Sorting by transpositions, SIAM J. Discrete Math. 11 (2) (1998) 224–240.
[4] A. Bergeron. A very elementary presentation of the Hannenhalli–Pevzner theory, in: Proceedings of 12th Annual Symposium on Combinatorial Pattern Matching (CPM '01), 2001.
[5] P. Berman, S. Hannanhalli, M. Karpinski, 1.375-approximation algorithm for sorting by reversals, in: Proceedings of 10th European Symposium on Algorithms (ESA'02), Lecture Notes in Computer Science, vol. 2461, Springer, 2002, pp. 200–210.
[6] A. Caprara, Sorting permutations by reversals and Eulerian cycle decompositions, SIAM J. Discrete Math. 12 (1) (1999) 91–110.
[7] D.A. Christie, A3/2-approximation algorithm for sorting by reversals, in: Proceedings of Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 98), ACM Press, 1998, pp. 244–252.
[8] D.A. Christie, Genome Rearrangement Problems. PhD thesis, University of Glasgow, 1999.
[9] I. Elias, T. Hartman, A 1.375-approximation algorithm for sorting by transpositions, in: Proceedings of 5th Workshop on Algorithms in Bioinformatics (WABI'05), LNBI 3692, 2005, pp. 204–215.
[10] H. Eriksson, K. Eriksson, J. Karlander, L. Svensson, J. Wastlund, Sorting a bridge hand, Discrete Math. 241 (1–3) (2001) 289–300.
[11] Q.P. Gu, S. Peng, H. Sudborough, A 2-approximation algorithm for genome rearrangements by reversals and transpositions, Theor. Comput. Sci. 210 (2) (1999) 327–339.
[12] S. Hannenhalli, P. Pevzner, Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals, J. ACM 46 (1999) 1–27 (Preliminary version in Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing 1995 (STOC 95), pp. 178–189)..
[13] T. Hartman, A simpler 1.5-approximation algorithm for sorting by transpositions, in: Proceedings 14th Annual Symposium on Combinaotrial Pattern Matching (CPM '03), Springer, 2003, pp. 156–169.
[14] T. Hartman, R. Sharan, A 1.5-approximation algorithm for sorting by transpositions and transreversals, J. Comput. Syst. Sci. 70 (2005) 300–320.
[15] M.I. Honda, Implementation of the Algorithm of Hartman for the Problem of Sorting by Transpositions, Master Thesis, 2004.
[16] S.B. Hoot, J.D. Palmer, Structural rearrangements, including parallel inversions, within the chloroplast genome of Anemone and related genera, J. Mol. Evol. 38 (1994) 274–281.
[17] H. Kaplan, R. Shamir, R.E. Tarjan, Faster and simpler algorithm for sorting signed permutations by reversals, SIAM J. Comput. 29 (3) (2000) 880–892 (Preliminary version in Proceedings of the eighth annual ACM-SIAM Symposium on Discrete Algorithms1997 (SODA 97), ACM Press, pp. 344–351).
[18] H. Kaplan, E. Verbin, Efficient data structures and a new randomized approach for sorting signed permutations by reversals, in: Proceedings 14th Annual Symposium on Combinaotrial Pattern Matching (CPM '03), Springer, 2003, pp. 170–185.
[19] G.H. Lin, G. Xue, Signed genome rearrangements by reversals and transpositions: models and approximations, Theor. Comput. Sci. 259 (2001) 513–531.
[20] J. Meidanis, M.E. Walter, Z. Dias, Reversal distance of signed circular chromosomes, Technical report IC-00-23, Institute of Computing, University of Campinas, December 2000.
[21] J.D. Palmer, L.A. Herbon, Tricircular mitochondrial genomes of Brassica and Raphanus: reversal of repeat configurations by inversion, Nucleic Acids Res. 14 (1986) 9755–9764.
[22] P.A. Pevzner, Computational Molecular Biology: An Algorithmic Approach, MIT Press, Cambridge, MA, 2000.
[23] D. Sankoff, N. El-Mabrouk, Genome rearrangement, in: T. Jiang, T. Smith, Y. Xu, M.Q. Zhang (Eds.), Current Topics in Computational Molecular Biology, MIT Press, Cambridge, MA, 2002.
[24] J. Setubal, J. Meidanis, Introduction to Computational Biology, PWS Publishing, 1997.
[25] R. Shamir, Algorithms in molecular biology: lecture notes. 2002. Available from: <http://www.math.tau.ac.il/~rshamir/algmb/01/algmb01.html>.
[26] D.D. Sleator, R.E. Tarjan, Self-adjusting binary search trees, J. Assoc. Comput. Mach. 32 (1985) 652–686.
[27] M.E. Walter, Z. Dias, J. Meidanis, Reversal and transposition distance of linear chromosomes, in: String Processing and Information Retrieval: A South American Symposium (SPIRE 98), 1998.

[28] M.E. Walter, L. Reginaldo, A.F. Curado, A.G. Oliveira, Working on the problem of sorting by transpositions on genome rearrangements, in: Proceedings of 14th Annual Symposium on Combinatorial Pattern Matching (CPM '03), Springer, 2003, pp. 372–383.

[29] M.E. Walter, M.C. Sobrinho, E.T.G. Oliviera, L.S. Soares, A.G. Oliviera, T.E.S. Martins, T.G. Fonseca, Improving the algorithm of Bafna and Pevzver for the problem of sorting by transpositions: a practical approach, in: Proceedings of Second Brazilian Workshop on Bioinformatics (WOB'03), 2003.