How to achieve an equivalent simple permutation in linear time

Simon Gog and Martin Bader

University of Ulm, Institute of Theoretical Computer Science, 89069 Ulm, Germany
Email: `simon.gog@uni-ulm.de`, `martin.bader@uni-ulm.de`

Abstract. The problem of *Sorting signed permutations by reversals* is a well studied problem in computational biology. The first polynomial time algorithm was presented by Hannenhalli and Pevzner in 1995 [5]. The algorithm was improved several times, and nowadays the most efficient algorithm has a subquadratic running time [9, 8]. *Simple permutations* played an important role in the development of these algorithms. Although the latest result of Tannier et al. [8] does not require simple permutations the preliminary version of their algorithm [9] as well as the first polynomial time algorithm of Hannenhalli and Pevzner [5] use the structure of simple permutations. However, the latter algorithms require a precomputation that transforms a permutation into an *equivalent simple permutation*. To the best of our knowledge, all published algorithms for this transformation have at least a quadratic running time. For further investigations on genome rearrangement problems, the existence of a fast algorithm for the transformation could be crucial. In this paper, we present a linear time algorithm for the transformation.

1 Introduction

The problem of *Sorting signed permutations by reversals* (SBR) is motivated by a genome rearrangement problem in computational biology. The task of the problem is to transform the genome of one species into the genome of another species, containing the same set of genes but in different order. As transformation step, only *reversals* (also called *inversions*) are allowed, where a section of the genome is excised, reversed in orientation, and reinserted. This is motivated by the fact that reversals are the most frequent rearrangement operations in nature, especially for bacterial genomes. The problem can be easily transformed into the mathematical problem of sorting a *signed permutation* (i.e. a permutation of the integers 1 to n, where each element has an additional orientation) into the identity permutation. The elements represent the genes of the genome (or any other kind of marker), whereas the signs indicate the strandedness of the genes. As shorter rearrangement scenarios are biologically more plausible than longer ones, one is interested in a minimum sequence of reversals that transforms one permutation into the identity permutation.

SBR is a well studied problem in computational biology, and the first polynomial time algorithm was presented by Hannenhalli and Pevzner in 1995 [5]. The algorithm was simplified several times [4,6], and the *reversal distance problem* (in which one is only interested in the number of required reversals) can be solved in linear time [1,3]. In 2004, Tannier and Sagot presented an algorithm for SBR that has subquadratic time complexity [9]. This algorithm first transforms the given permutation π into an *equivalent simple permutation* $\hat{\pi}$ and then calculates a sorting for $\hat{\pi}$. This sorting is subsequently used to sort π. In literature, there are several algorithms for this transformation [5,4], but all of them have at least quadratic time complexity (there is an unpublished linear time algorithm by Tannier and Sagot which uses another technique than our algorithm, personal communication). Although Tannier et al. improved their algorithm such that it does no longer require simple permutations [8], a fast algorithm for the transformation could be crucial for further investigations on genome rearrangements. In this paper, we will provide a linear algorithm for transforming a permutation into an equivalent simple permutation.

2 Preliminaries

A *signed permutation* $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ is a permutation of the integers 1 to n, where each element π is assigned a positive ($\overrightarrow{\pi}$) or negative ($\overleftarrow{\pi}$) orientation. A *reversal* $\rho(i,j)$ reverses the order and flips the orientation of the elements between the i-th and j-th element of the permutation. For example, $\rho(3,5)$ transforms $\pi = \langle \overrightarrow{1}, \overrightarrow{2}, \boxed{\overleftarrow{5}, \overleftarrow{4}, \overleftarrow{3}}, \overrightarrow{6} \rangle$ into $id = \langle \overrightarrow{1}, \overrightarrow{2}, \overrightarrow{3}, \overrightarrow{4}, \overrightarrow{5}, \overrightarrow{6} \rangle$. The latter permutation is called identity permutation of size 6. The problem of sorting by reversals asks for a minimal sequence of reversals $\rho_1, \ldots, \rho_k$ that transforms a signed permutation π into the identity permutation. The length k of a minimal sequence is called the reversal distance $d(\pi)$.

The main tool for the solution of the problem of sorting by reversals is the *reality-desire diagram* (also called *breakpoint graph* [2,7]; see Fig. 1 for an example). The reality-desire diagram $RD(\pi)$ of a permutation $\pi = \langle \pi_1, \ldots, \pi_n \rangle$ can be constructed as follows. First, the elements of π are placed from left to right on a straight line. Second, each element x of π with positive orientation is replaced with the two nodes $2x-1$ and $2x$, while each element x with negative orientation is replaced with $2x$ and

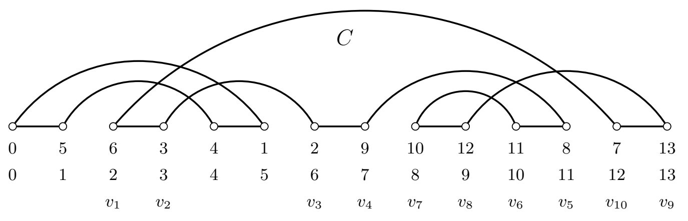


Fig. 1. A reality-desire diagram $RD(\pi)$ for $\pi = \langle \overrightarrow{3}, \overrightarrow{2}, \overrightarrow{1}, \overrightarrow{5}, \overleftarrow{6}, \overleftarrow{4} \rangle$. The first row of numbers are the labels of the nodes, the second are the positions. The third row contains the labeling of nodes of the long cycle C.

$2x-1$. We call these nodes *co-elements* of x where the first is called *left node* of x and the other the *right node* of x. Third, we add a single node labeled with 0 to the left of the left node of the first element and add a single node labeled with $2n+1$ to the right of the right node of the last element. Fourth, *reality edges* are drawn from the right node of π_i to the left node of π_{i+1} ($1 \leq i < n$), from node 0 to the left node of π_1, and from the right node of π_n to node $2n+1$. Fifth, *desire edges* are drawn from node $2i$ to node $2i+1$ ($0 \leq i \leq n$). We can interpret reality edges as the actual neighborhood relations in the permutation, and desire edges as the desired neighborhood relations. The *position* of a node v is its position in the diagram and denoted by $pos(v)$ (i.e. the leftmost node has the position 0, the node to its right has the position 1, and so on). As each node is assigned exactly one reality edge and one desire edge, the reality-desire diagram decomposes into cycles. The number of cycles in $RD(\pi)$ is denoted by $c(\pi)$. The *length* ℓ_j of a cycle C_j is the number of desire edges. If ℓ_j is smaller than 3 we call C_j a *short cycle*, otherwise a *long cycle*.

We label the nodes of a cycle C_j as follows. The leftmost node is called $v[j]_1$, then we follow the reality edge to node $v[j]_2$, then follow the desire desire edge to node $v[j]_3$, and so on. We label the reality edge from node $n[j]_{2i-1}$ to $n[j]_{2i}$ with $b[j]_i$ ($1 \leq i \leq \ell_j$) and the desire edge from node $n[j]_{2i}$ to $n[j]_{(2i+1)}$ with $g[j]_i$ ($1 \leq i < \ell_j$). The desire edge from node $n[j]_{2\ell}$ to $n[j]_1$ is labeled with $g[j]_{\ell_j}$. If the cycle index j of C_j is clear from the context we omit it.

A desire edge $g = (v_1, v_2)$ is called *oriented* if the positions of v_1 and v_2 in the diagram are both even or odd, otherwise we call g *unoriented*. A cycle which contains no oriented edges is called *unoriented*, otherwise *oriented*.

Two desire edges (v_1, v_2) and (w_1, w_2) *interleave* if the endpoints of the intervals $I_v = [pos(v_1), pos(v_2)]$ and $I_w = [pos(w_1), pos(w_2)]$ are alternating. Two cycles C_1 and C_2 are *interleaving* if there exist interleaving desire edges $f \in C_1$ and $g \in C_2$. A maximal set of interleaving cycles in $RD(\pi)$ is called a *component*. A component is *unoriented* if it contains no oriented cycles, otherwise it is *oriented*.

Hannenhalli and Pevzner found some special structures that depend on unoriented components called *hurdles* and *fortress*. The distance formula for the reversal distance is

$$d(\pi) = n + 1 - c(\pi) + h(\pi) + f(\pi)$$

where $h(\pi)$ is the number of hurdles in $RD(\pi)$ and $f(\pi)$ the indicator variable for a fortress (for details see [5]).

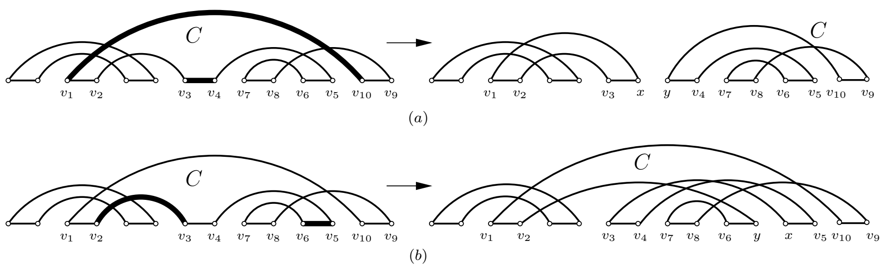
The original Hannenhalli-Pevzner algorithm [5] as well as the subquadratic algorithm of Tannier and Sagot [9] require a permutation whose reality-desire diagram contains only short cycles. Such a permutation is called a *simple permutation*. Hannenhalli and Pevzner showed that every permutation π can be transformed into an *equivalent simple permutation* $\hat{\pi}$, i.e. a simple permutation with $d(\hat{\pi}) = d(\pi)$, by padding additional elements to π. Moreover, a sorting sequence of $\hat{\pi}$ can be used to obtain a sorting sequence of π by ignoring the padded elements.

3 Creating equivalent simple permutations revisited

We first focus on the creation of simple permutations before we discuss the creation of equivalent simple permutations. If a permutation $\pi = \pi(0)$ has a long cycle, Hannenhalli and Pevzner [5] transform it into a new permutation $\pi(1)$ by ,,breaking" this cycle into two smaller ones. This step is repeated until a simple permutation $\pi(k)$ is achieved.

On the reality-desire diagram the ,,breaking of a cycle" can be described as follows. Let $b = (v_{b1}, v_{b2})$ be a reality edge and $g = (v_{g1}, v_{g2})$ a desire edge belonging to a cycle $C = (\ldots, v_{b1}, v_{b2}, \ldots, v_{g1}, v_{g2}, \ldots)$ in $RD(\pi(i))$. A (b, g)-*split* of $RD(\pi(i))$ produces a new diagram $\hat{RD}(\pi) = RD(\pi(i + 1))$ which is obtained from $RD(\pi(i))$ by:

1. removing edges b and g,
2. adding two new vertices x and y,
3. adding two new reality edges (v_{b1}, x) and (y, v_{b2}),
4. adding two new desire edges (v_{g1}, x) and (y, v_{g2}).



C
v_1 v_2 v_3 v_4 v_7 v_8 v_6 v_5 v_{10} v_9
C
v_1 v_2 v_3 x y v_4 v_7 v_8 v_6 v_5 v_{10} v_9
(a)
C
v_1 v_2 v_3 v_4 v_7 v_8 v_6 v_5 v_{10} v_9
C
v_1 v_2 v_3 v_4 v_7 v_8 v_6 y x v_5 v_{10} v_9
(b)

Fig. 2. (a) An unsafe (b, g)-split with $b = (v_3, v_4)$ and $g = (v_1, v_{10})$ that produces a new hurdle. (b) A safe (b, g)-split with $b = (v_5, v_6)$ and $g = (v_2, v_3)$, that does not produce any new components.

Two examples of such splits are illustrated in Fig. 2. As a result of the split the cycles $(\ldots, v_{b1}, x, v_{g1}, \ldots)$ and $(\ldots, v_{b2}, y, v_{g2}, \ldots)$ are created.

The effect of a (b, g)-split on the permutation can be described as follows. x and y are the nodes of a new element which lies between the consecutive elements previously connected by g. That is, we now consider *generalized permutations* which consists of arbitrary distinct reals instead of permutations of integers. Hannenhalli and Pevzner called the effects of a (b, g)-split on the permutation a (b, g)-padding. We will only use the term (b, g)-split as the two concepts are equivalent.

A (b, g)-split is *safe* if b and g are non-incident, and $\pi(i)$ and $\pi(i+1)$ have the same number of hurdles; i.e. $h(\pi(i)) = h(\pi(i + 1))$. The first condition assures that we do not produce a 1-cycle and a cycle with the same size as the old cycle. Because a split is acting on a long cycle, the first condition is easy to achieve. The second condition assures that the reversal distances of $\pi(i)$ and $\pi(i+1)$ are equal (note that a split increases both n and c by one, and the fortress indicator cannot be changed without changing the number of hurdles). The following lemma shows that to fulfill the second condition, it is sufficient to ensure that the resulting cycles belong to the same component.

Lemma 1 ([5]). *Let a (b, g)-split break a cycle C in $RD(\pi(i))$ into cycles C_1 and C_2 in $RD(\pi(i + 1))$. Then C is oriented if and only if C_1 or C_2 is oriented.*

In other words, if we do not split a component into two components, the orientation of the component is not changed. For the constructive proof of the existence of safe splits we need the following lemma.

Lemma 2 ([5]). *For every desire edge g that does not belong to a 1-cycle, there exists a desire edge f interleaving with g in $RD(\pi)$. If C is* *a cycle in $RD(\pi)$ and $f \notin C$ then f interleaves with an even number of desire edges in C.*

And for the linear time algorithm we need the following corollary.

Corollary 1. *Let C be a cycle of length $\ell > 1$ in $RD(\pi)$ with desire edges g_1 to g_ℓ. If these desire edges are pairwise non-interleaving, then there exists a g_j with $1 \leq j < \ell$ and a cycle $C' \neq C$ with a desire edge f, such that f interleaves both g_j and g_ℓ.*

Proof. As C has no pairwise interleaving desire edges, g_ℓ does not interleave with another desire edge of C. So Lemma 2 implies that g_ℓ interleaves with a desire edge f of another cycle C'. Because f is not in C, it interleaves with an even number of desire edges in C. It follows that f interleaves with at least one more desire edge g_j $(1 \leq j < \ell)$ of C.

Theorem 1 ([5]). *If $C = (\ldots, v_1, \ldots, v_{2\ell}, \ldots)$ is a long cycle in $RD(\pi)$, then there exists a safe (b, g)-split acting on C.*

The proof given in [5] is constructive. However, the construction cannot transform the whole permutation into a simple permutation in linear time (which is the goal of our paper). Therefore, in Section 5, we provide an algorithm that achieves this goal in linear time.

4 The data structure

We represent the reality-desire diagram as a linked list of $2n + 2$ nodes. The data structure `node` for each node v consists of the three pointers `reality` (pointing to the node connected with v by a reality edge), `desire` (pointing to the node connected with v by a desire edge), and `co_element` (pointing to the co-element of v), and the two variables `position` (the position w.r.t. the leftmost node in the diagram), and `cycle` (the index j of cycle C_j the node belongs to).

We can initialize this data structure for every permutation in linear time. First, the initialization of `reality`, `co_element`, and `position` can be done with a scan through the permutation. Second, for the initialization of `desire` we need the inverse permutation (mapping the nodes ordered by their label to their position) which can also be generated in linear time. Finally, we can initialize `cycle` by following the reality and desire edges which also takes linear time.

Given a reality edge $b = (v_{b1}, v_{b2})$ and a desire edge $g = (v_{g1}, v_{g2})$, a (b, g)-split can be performed in constant time (see Algorithm 1) if we disregard the problem that we have to update the position variables of the new nodes and all the nodes that lie to the right of b. Fortunately, we need `position` only to determine if two edges of the same cycle interleave, thus it is sufficient if the relative positions of the nodes of each cycle are correct. This information can be maintained if we set the positions of the new nodes x and y to the positions of the old nodes of b which are now non-incident to x or y. After performing all splits, the reality-desire diagram can easily be transformed into the simple permutation by following desire edges and co-element pointers.

Algorithm 1 (b,g)-split

```
1: function bg-split($b = (v_{b1}, v_{b2}), g = (v_{g1}, v_{g2})$)
2:     create new nodes $x$, $y$
3:     $v_{b1}.reality = x$; $v_{b2}.reality = y$ {adjust reality and desire edges}
4:     $x.reality = v_{b1}$; $y.reality = v_{b2}$
5:     $v_{g1}.desire = x$; $v_{g2}.desire = y$
6:     $x.desire = v_{g1}$; $y.desire = v_{g2}$
7:     $x.position = v_{b2}.position$; $y.position = v_{b1}$
8:     $return(x, y)$
```

5 The Algorithm

We now tackle the problem of transforming a permutation into an equivalent simple permutation in linear time. The algorithm has two processing phases.

Phase 1:
Our goal in the first phase is to create short cycles or cycles that have no interleaving desire edges. We achieve this goal with a scanline algorithm. The algorithm requires two additional arrays: `left[j]` stores the leftmost node of each cycle C_j and `next[j]` stores the right node of the desire edge we are currently checking for interleavings. In both arrays, all variables are initialized with `UNDEF`. In the following, v_s denotes the current position of the scanline. Before we describe the algorithm, we will first provide an invariant for the scanline.

Invariant: If g_i is a desire edge of the long cycle C_j with $i < \ell_j$, and both nodes of g_i lie to the left of v_s, then g_i does not intersect with any other desire edge of C_j.

It is clear that a cycle C_j has no interleaving edges if the invariant holds and the scanline passed the rightmost node of C_j: g_{ℓ_j} does also not interleave with a desire edge of C_j because the interleaving relation is symmetric. As v_s is initialized with the leftmost node of $RD(\pi)$, the invariant holds in the beginning. While the scanline has not reached the right end of the diagram, we repeat to analyze the following cases:

Case 1.1 v_s **is part of a short cycle**.

We move the scanline to the left node of the next reality edge. As the invariant only considers long cycles, the invariant is certainly preserved.

Case 1.2 v_s **is part of a long cycle** C_j **and** `next[j]=UNDEF`.

That is, v_s is the leftmost node of cycle C_j. So we set `left[j]=`v_s. To check whether $g_1 = (v_2, v_3)$ interleaves with another desire edge, we store the right node of g_1 in `next[j]` and move v_s to the left node of the next reality edge. Both nodes passed by the scanline (i.e. v_1 and v_2) are the left nodes of a desire edge, so the set of desire edges that lie completely to the left of v_s is not changed and the invariant is preserved.

Case 1.3 v_s **is part of a long cycle** C_j **and** `next[j]`$\neq v_s$.

Let `next[j]` be the node v_{2k+1}, i.e. we check for a desire edge that interleaves with g_k (going from node v_{2k} to node v_{2k+1}). As $pos(v_1) < pos(v_{2k}) < pos(v_s) < pos(v_{2k+1})$, there must be a desire edge g_m belonging to C_j that interleaves with g_k. We now distinguish three cases:

(a) g_k **is not** g_1 (for an example, see Fig. 3).
We perform a (b, g)-split with $b = b_{k+1}$ and $g = g_{k-1}$. That is, we split the 2-cycle $(v_{2k}, v_{2k+1}, x, v_{2k-1})$ from C_j. This split is save since g_k now lies in the 2-cycle that still interleaves with g_m, which belongs to C_j. The right node of the new g_{k-1} in C_j is y, so we adjust `next[j]` to y.

(b) g_k **is** g_1 **and** g_k **interleaves with** g_{ℓ_j} (see Fig. 4).
We perform a (b, g)-split with $b = b_1$ and $g = g_2$. That is, we split the 2-cycle (v_2, v_3, v_4, y) from C_j. This split is save since g_1 now lies in the 2-cycle that still interleaves with g_{ℓ_j}, which belongs to C_j. Now, $g_1 = (x, v_5)$, so we set `next[j]=`v_5. Note that v_5 cannot be to the left of v_s, as v_s is the leftmost node that belongs to C_j and has an index ≥ 4.

(c) g_k **is** g_1 **and** g_k **does not interleave with** g_{ℓ_j} (see Fig. 5).
It follows that $g_m \neq g_{\ell_j}$. We perform a (b, g)-split with $b = b_2$ and $g = g_{\ell_j}$. That is, we split the 2-cycle (v_2, v_3, x, v_1) from C_j. This

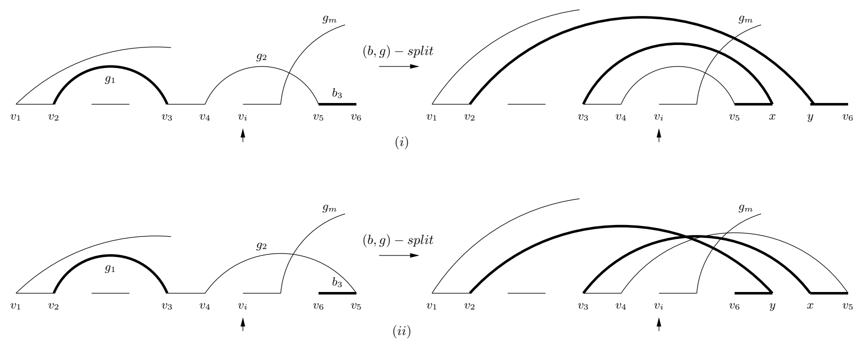

g_m
g_2
g_1
b_3
$(b, g) - split$
v_1 v_2 v_3 v_4 v_i v_5 v_6
v_1 v_2 v_3 v_4 v_i v_5 x y v_6
(i)
v_1 v_2 v_3 v_4 v_i v_6 v_5
v_1 v_2 v_3 v_4 v_i v_6 y x v_5
(ii)

Fig. 3. Case 1.3 (a): (i) $g_k = g_2$ is unoriented or (ii) oriented.

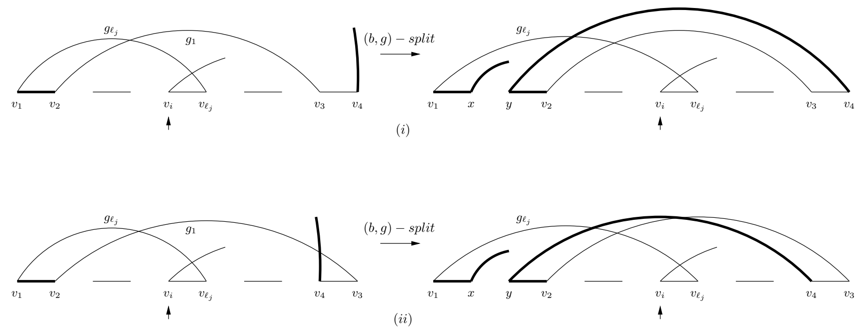

g_{ℓ_j}
g_1
$(b, g) - split$
v_1 v_2 v_i v_{ℓ_j} v_3 v_4
v_1 x y v_2 v_i v_{ℓ_j} v_3 v_4
(i)
v_1 v_2 v_i v_{ℓ_j} v_4 v_3
v_1 x y v_2 v_i v_{ℓ_j} v_4 v_3
(ii)

Fig. 4. Case 1.3 (b): (i) g_1 is unoriented or (ii) oriented.

split is save since g_1 now lies in the 2-cycle that still interleaves with g_m. As the old leftmost node and reality edge of C_j lie in the 2-cycle we set `next`[`j`] $= UNDEF$ which forces the re-initialization of `left`[`j`] with v_s and `next`[`j`].

In all of these cases, we do not create a desire edge that lies completely to the left of v_s, so the invariant is preserved.

Case 1.4 v_s **is part of a long cycle** C_j **and** `next[j]=`v_s.

That is, we reach the right node of a desire edge g_k. It follows that g_k does not interleave which any other desire edge of C_j since we have not detected a node of C_j between the left and right node of g_k. Thus moving v_s to the right preserves the invariant. The next desire edge to check is $g_{k+1} = (v_{2(k+1)}, v_{2(k+1)+1})$, so we set `next[j]` to the right node of g_{k+1} and move v_s to the left node of the next reality edge.

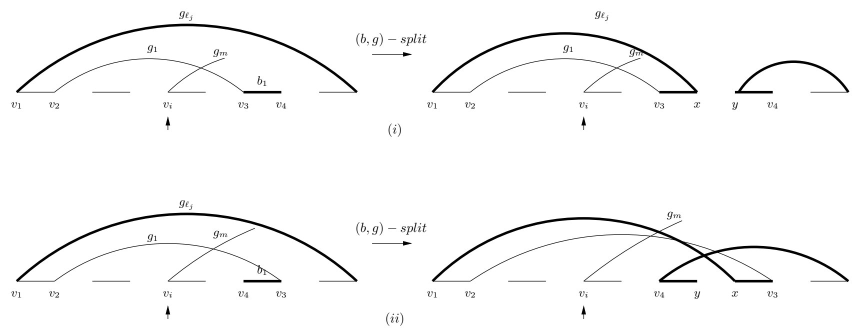


Fig. 5. Case 1.3 (c): (i) g_1 is unoriented or (ii) oriented.

We will now analyze the running time of the first phase. In each step we either move the scanline further right (cases 1.1, 1.2, and 1.4) or perform a save (b, g)-split (cases 1.3(a), 1.3(b), and 1.3(c)). As we can perform at most n splits and the resulting diagram can have at most $2n$ reality edges, we have to perform at most $3n$ steps. Each step takes constant time.

Phase 2 After phase 1 we can assure that there remain only short cycles and long cycles with pairwise non-interleaving desire edges. These long cycles have a special structure. The positions of the nodes $v_1, \ldots, v_{2\ell_j}$ of a cycle C_j are strictly increasing and so the first $\ell_j - 1$ desire edges g_i $(i < \ell_j)$ lie one after another. g_{ℓ_j} connects the leftmost and rightmost node of C_j. As we know from Corollary 1 there exists a desire edge f of a cycle $C' \neq C_j$ that interleaves with g_{ℓ_j} and another desire edge g_k of C_j.

We can detect this g_k by first determining a desire edge f which has a node in the interval $I_j = [pos(v_1), pos(v_{2\ell_j})]$ and interleaves with g_{ℓ_j}. Second, we get the g_i that interleaves with f by checking for every desire edge $\neq g_{\ell_j}$ whether it interleaves with f. As I is decomposed by the intervals of the desire edges in distinct areas, we get the corresponding g_i in at most ℓ_j steps.

Clearly, the second step takes $\sum_{j=1}^{c(\pi)} \ell_j = O(n)$ time. In the first step, we use a stack based algorithm to achieve a linear running time. In each step of the algorithm, the stack will contain a set of intervals I_j of cycles C_j, such that each interval on the stack is completely contained in all other intervals that are below it on the stack (i.e. the topmost interval is contained in all other intervals on the stack). We scan the reality-desire diagram from left to right. For each node v, we check whether its desire edge $f = (v, w)$ interleaves with the topmost interval I_j of the stack. If so, we report the interleaving edges f and g_{ℓ_j}, pop I_j from the stack, check whether f interleaves with the new top interval, and so on, until f does not interleave with the top interval. As the top interval is contained in all other intervals of the stack and Lemma 2 ensures that we find an interleaving edge before we reach the right end of the interval (i.e. v is contained in the topmost interval), f cannot interleave with any other interval on the stack. If v is the leftmost node of a cycle C_j, we push I_j on the stack (note that this interval is equivalent to the desire edge g_{ℓ_j}, so it does not interleave with the topmost interval and is therefore contained in it). In all cases, we continue by moving the scanline one node to the right. The algorithm stops when we have reached the right end of the diagram. During the algorithm, we push the interval I_j of each cycle C_j on the stack, and pop this cycle when we reach a node v in I_j such that the desire edge (v, w) interleaves with I_j. As this node must exist for each cycle (see Lemma 2), we find for each cycle C_j an edge that interleaves with g_{ℓ_j}.

After finding all g_k's we distinguish two cases for a save (b, g)-split:

Case 2.1 $g_{\ell_j - 1} \neq g_k$ (see Fig. 6(i)).
We perform the (b, g)-split on C with $b = (v_1, v_\ell)$ and $g = (v_3, v_4)$. We get $C_1 = (v_1, v_2, v_3, a)$ and $C_2 = (v_\ell, v_{\ell-1}, \ldots, v_4, b)$. As f interleaves with g_1 which is now part of C_1 and g_i which is now part of C_2 the component structure remains the same.

Case 2.2 $g_{\ell_j - 1} = g_k$ (see Fig. 6(ii)).
We perform the (b, g)-split on C with $b = (v_3, v_2)$ and $g = (v_\ell, v_{\ell-1})$. We get $C_1 = (v_1, v_2, b, v_\ell)$ and $C_2 = (a, v_3, v_4, \ldots, v_{\ell-1})$. As f interleaves with g_1 which is now part of C_1 and g_i which is now part of C_2 the component structure remains the same.

In both cases, g_k becomes a desire edge of the cycle C_2, and f intersects both g_k and $g_{\ell'}$ (where ℓ' is the length of C_2). Thus we do not have to recalculate the edge g_k, and can repeat this step on C_2 until the remaining cycles are all 2-cycles. The pseudo code of the whole algorithm is presented in Appendix A. An implementation in C++ can be obtained from the authors.

References

1. D. Bader, B. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8:483–491, 2001.

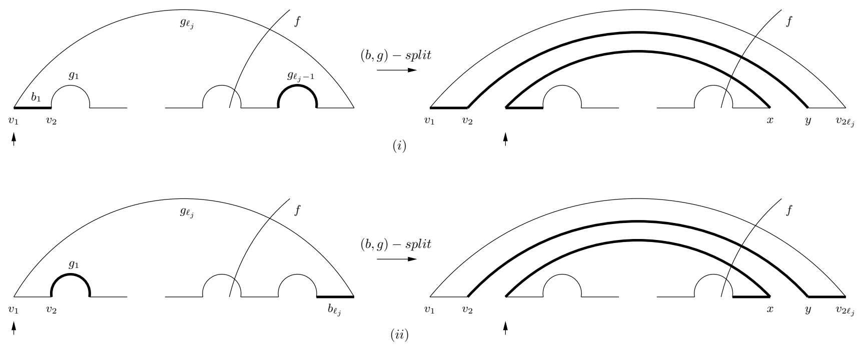


Fig. 6. (i) depicts Case 2.1 and (ii) Case 2.2.

2. V. Bafna and P. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.
3. A. Bergeron, J. Mixtacki, and J. Stoye. Reversal distance without hurdles and fortresses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching*, volume 3109 of *LNCS*, pages 388–399. Springer-Verlag, 2004.
4. P. Berman and S. Hannenhalli. Fast sorting by reversals. In *Proc. 7th Symposium on Combinatorial Pattern Matching*, volume 1075 of *LNCS*, pages 168–185. Springer-Verlag, 1996.
5. S. Hannenhalli and P. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, 1999.
6. H. Kaplan, R. Shamir, and R. Tarjan. A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, 29(3):880–892, 1999.
7. J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
8. E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155:881–888, 2007.
9. E. Tannier and M.-F. Sagot. Sorting by reversals in subquadratic time. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching*, volume 3109 of *LNCS*, pages 1–13. Springer-Verlag, 2004.

A Code

Algorithm 2 Equivalent transformation in a simple permutation into linear time

```
1: read $\pi$ and construct the reality-desire diagram $RD(\pi)$
2: mark and count cycles in $RD(\pi)$
3: $left[1..c(\pi)] := \{\text{undef}, \ldots, \text{undef}\}$; $next[1..c(\pi)] := \{\text{undef}, \ldots, \text{undef}\}$
4: set scanline $v_s$ to the leftmost node of $RD(\pi)$
5: while $v_s \neq nil$ do
6:    j:=$v_s.cycle$
7:    if $v_s$ is part of a short cycle then
8:       $v_s := v_s.reality.co\_element$
9:    else if $next[j]$ = undef then {we reach the leftmost point of cycle $C_j$}
10:      $left[j] := v_s$
11:      $next[j] := v_s.reality.desire$
12:      $v_s := v_s.reality.co\_element$
13:   else if $v_s = next[j]$ then {i.e. $g_i$ does not interleave with edge from $C_j$}
14:      $next[j] := v_s.reality.desire$
15:      $v_s := v_s.reality.co\_element$
16:   else if $g_k$ is not $g_1$ then
17:      (x,y):=bg-split($b_{k+1}$,$g_{k-1}$)
18:      $next[j] := y$
19:   else if $g_k$ interleaves with $g_{\ell_j}$ then
20:      (x,y):=bg-split($b_1$,$g_2$)
21:      $next[j] := v_5$
22:   else {$g_k$ does not interleave with $g_{\ell_j}$}
23:      bg-split($b_2$,$g_{\ell_j}$)
24:      $next[j]$ := undef

25: calculate the absolute position for each node in $RD(\pi')$
26: create stack ACTIVE_CYCLE
27: set scanline $v_s$ to the leftmost node of $RD(\pi')$
28: while $v_s \neq NIL$ do
29:   while ACTIVE_CYCLE is not empty do
30:      $g_\ell$:=ACTIVE_CYCLE.top
31:      if $(v_s, v_s.desire)$ or $(v_s.reality, v_s.reality.desire)$ interleaves with $g_\ell$ then
32:         determine $g_k$
33:         ACTIVE_CYCLE.pop
34:      else
35:         break
36:   if $v_s$ is the leftmost node of a long cycle then
37:      ACTIVE_CYCLE.push($(v_s, v_s.desire)$)
38:   $v_s := v_s.reality.co\_element$

39: for each node $v_i$ do
40:   if $v_i$ is the leftmost node of a long cycle $C_j$ then
41:      if $g_{\ell_j - 1} \neq g_k$ then
42:         bg-split($b_1$, $g_{\ell_j - 1}$)
43:      else
44:         bg-split($b_{\ell_j}$,$g_1$)
```