A New Linear Time Algorithm to Compute the Genomic Distance Via the Double Cut and Join Distance

Anne Bergeron[a], Julia Mixtacki[b,1], Jens Stoye[*,b]

[a] *Département d'informatique, Université du Québec à Montréal, Canada*
[b] *Universität Bielefeld, Technische Fakultät, AG Genominformatik, Germany.*

Abstract

The genomic distance problem in the Hannenhalli-Pevzner (HP) theory is the following: Given two genomes whose chromosomes are linear, calculate the minimum number of translocations, fusions, fissions and inversions that transform one genome into the other. This paper presents a new distance formula based on a simple tree structure that captures all the delicate features of this problem in a unifying way, and a linear-time algorithm for computing this distance.

Key words: comparative genomics, genome rearrangements, genomic distance computation, sorting by translocations, fusions, fissions and inversions

1. Introduction

A *genome* is a collection of large linear or circular double-stranded molecules called *chromosomes* described by their ordered sequences of *nucleotides*. These are smaller building blocks that come in four shapes, usually designated by the four letters A, C, G and T. Following a century old tradition [10], a contiguous segment of a chromosome that carries some inherited marker is called a *gene*. The biological notion of gene has been under constant revision in the last decades, but, in this paper, we will be interested in the combinatorial properties of segments of chromosomes.

When two genomes are compared, it is common to find *homologous* genes between them. Homologous genes are defined as genes that share a common ancestor and identifying homologous genes is a complex process that uses numerous algorithmic and biological techniques. A detailed discussion of this subject is outside the scope of this paper, and the interested reader is refered to [11]. Homologous genes may, or may not, belong to the same chromosomes in different genomes, and their relative *orientation* may differ. Differences in relative orientations appear when during the course of evolution segments of chromosomes are broken, and repaired in the reverse direction. Homologous genes are assigned a common label, and a sign to reflect relative orientation, allowing us to represent genomes as sets of signed sequences such as:

[*] Corresponding author
Email addresses: bergeron.anne@uqam.ca (Anne Bergeron), julia.mixtacki@uni-bielefeld.de (Julia Mixtacki), stoye@techfak.uni-bielefeld.de (Jens Stoye)

[1] Supported by the group "Combinatorial Search Algorithms in Bioinformatics" funded by the Sofja Kovalevskaja Award 2004 of the Alexander von Humboldt Foundation and the Bundesministerium für Bildung und Forschung.

Dagstuhl Seminar Proceedings 10231
Structure Discovery in Biology: Motifs, Networks & Phylogenies
http://drops.dagstuhl.de/opus/volltexte/2010/2689

$$\begin{aligned} \text{Genome}A &= \{(\circ, 1, 2, 3, 4, \circ), (\circ, 5, 6, \circ)\} \\ \text{Genome}B &= \{(\circ, 1, -2, 6, \circ), (\circ, 5, \circ), (3, -4)\} \end{aligned}$$

The symbol '$\circ$' is used to indicate that the corresponding chromosome is a linear molecule, and, borrowing from the biological terminology, is referred to as a *telomere* marker. This kind of representation of genomes is unique up to chromosome flipping, in which the order and orientation of all genes in a chromosome are reversed, or circular permutations of circular chromosomes. For example, $(\circ, 1, -2, 6, \circ) \equiv (\circ, -6, 2, -1, \circ)$, and $(3, -4) \equiv (-4, 3) \equiv (4, -3) \equiv (-3, 4)$. In the above example, genome A has two linear chromosomes, and genome B has two linear and one circular chromosome. In this example, up to its sign, each gene appears once in each genome. This is a considerable, but useful simplification with respect to biological reality, and we will assume this is the case throughout the paper.

Several edit operations have been proposed to explain the differences between genomes. In the case of linear genomes, that is, genomes that have only linear chromosomes, the two main operations are *inversions* inside chromosomes and *translocations* between chromosomes. In the first case, the order and sign of consecutive genes in a chromosome are reversed, transforming, for example $(\circ, 1, 2, 3, 4, 5, \circ)$ into $(\circ, 1, 2, -4, -3, 5, \circ)$. In the second case, two chromosomes exchange prefix or suffix of their sequences, transforming, for example $(\circ, 1, 2, 3, 4, 5, \circ)$ and $(\circ, 6, 7, 8, 9, \circ)$ into either:

$$(\circ, 1, 2, 8, 9, \circ) \text{ and } (\circ, 6, 7, 3, 4, 5, \circ)$$

or

$$(\circ, 1, 2, -7, -6, \circ) \text{ and } (\circ - 9, -8, 3, 4, 5, \circ).$$

Special cases of translocations, such as fusions or fissions, are obtained by allowing cuts to occur next to telomere markers and adding empty chromosomes.

The *genomic distance problem* for linear genomes is to compute the minimum number of inversions and translocations that are necessary to transform one genome into the other. The first solution to this problem was given by Hannenhalli and Pevzner [12] in 1995. Their distance formula, called the general *HP theorem*, requires preprocessing steps such as *capping* and *concatenation* and involves seven parameters. In the last decade, different authors pointed to problems in the original formula and in the algorithm given by Hannenhalli and Pevzner. Their algorithm was first corrected by Tesler [17]. In 2003, Ozery-Flato and Shamir [16] found a counter-example and modified one of the parameters of the distance formula. Very recently, another correction was presented by Jean and Nikolski [14]. Unfortunately, the last two recent results have not resulted in simpler presentations of the material, and the only available software tool is *GRIMM* implemented by Glenn Tesler [18].

In contrast to this rather complicated distance measure, Yancopoulos *et al.* [19] presented a general genome model that includes linear and circular chromosomes and introduced a new operation called *double cut and join* (or shortly DCJ) operation. In addition to inversions and translocations, the DCJ operation also models more complicated rearrangement operations like transpositions and block-interchanges. Beside the simple distance computation, the sorting algorithm is also basic and efficient [8].

In this paper we will show how the rearrangement model considered in the HP theory can be integrated in the more general DCJ model. Specifically, the HP distance can be expressed as

$$d_{HP} = d_{DCJ} + t$$

where t represents the extra cost of not resorting to "forbidden" DCJ operations. The extra cost can easily be computed in linear time by a tree data structure associated to a genome.

The next section recalls the results on the DCJ distance. In Section 3, we establish the conditions under which the HP and DCJ distances are equal, which corresponds to the case where $t = 0$. The general case $t \geq 0$ is treated in Section 4, where we introduce the basic concepts and the tree needed for the computation of the HP distance, and we give a new proof and formula for the Hannenhalli-Pevzner theorem. The algorithms for the distance computation are described in Section 5. Finally, Section 6 presents the conclusion.

2. The Double Cut and Join (DCJ) Model

Let A and B be two linear genomes on the same set of N genes. An *interval* $(l, \ldots, r)$ in a genome is a set of consecutive genes or telomere markers within a chromosome; the set $\{l, -r\}$ is the set of *extremities* of the interval – note that $\circ = -\circ$. An *adjacency* is an interval of length 2, an adjacency that contains a telomere marker is called a *telomere*. Each gene g is the extremity of two adjacencies, one as $+g$, and one as $-g$, in both genomes A and B. Given the set of adjacencies of a linear genome, it is thus possible to uniquely reconstruct each of its chromosomes by starting with telomeres, and listing the successive adjacencies that share extremities of the same gene.

Definition 1. Given two adjacencies $\{x, -y\}$ and $\{u, -v\}$ in a genome G, a *double cut and join* (DCJ) operation applied on these adjacencies produces a genome in which adjacencies $\{x, -y\}$ and $\{u, -v\}$ are replaced either by $\{x, -v\}$ and $\{u, -y\}$ or by $\{x, u\}$ and $\{-y, -v\}$.

A simple interpretation of this operation in terms of consecutive segments ab and cd of chromosomes is making two cuts, one between a and b, and the other between c and d. There are then three possible 'repairs': the recently cut extremity a can be joined with either b, c or d, and the two remaining extremities are joined together. Of the three possibilities, the first one yields the original genome, but the two others change the set of adjacencies. The precise formulation of Definition 1 takes care of the changes in gene orientations.

DCJ operations are used to model both inversions and translocations in linear genomes. For translocations, the two choices given in the example of the Introduction correspond exactly to the choices in Definition 1. However, for inversions, only one choice yields linear chromosomes: for example, applying a DCJ operation on the two adjacencies $\{2, -3\}$ and $\{4, -5\}$ of chromosome $(\circ, 1, 2, 3, 4, 5, \circ)$ may yield adjacencies $\{2, 4\}$ and $\{-3, -5\}$, thus chromosome:

$$(\circ, 1, 2, -4, -3, 5, \circ),$$

but it could also yield adjacencies $\{2, -5\}$ and $\{-3, -4\}$. In this case, the resulting genome would contain:

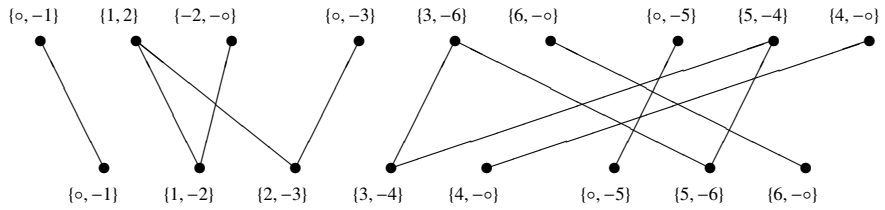
$$(\circ, 1, 2, 5, \circ), (-3, -4),$$

in which the second chromosome is a circular chromosome. Thus DCJ operations are more general than the usual rearrangement operations on linear genomes.

The following basic construct is associated to a pair of genomes, each represented by its set of adjacencies.

Definition 2. The *adjacency graph* $AG(A, B)$ is a graph whose vertices are the adjacencies of genomes A and B. Each gene g defines two edges, one connecting the two adjacencies of genome A and B in which g appears as extremity $+g$, and one connecting the two adjacencies in which g appears as extremity $-g$.

Since adjacencies that are telomeres have only one gene, the vertices of the adjacency graph will have degree one or two, thus the graph is a union of *paths* and *cycles*. Paths of odd length, called *odd paths*, connect telomeres of different genomes, and paths of even length, the *even paths*, connect telomeres of the same genome. For example, the adjacency graph of genomes $A = \{(\circ, 1, -2, \circ), (\circ, 3, 6, \circ), (\circ, 5, 4, \circ)\}$ and $B = \{(\circ, 1, 2, 3, 4, \circ), (\circ, 5, 6, \circ)\}$ has four odd paths, one cycle and one even path:



It is interesting to note that a *DCJ operation* applied to two adjacencies of the same genome disconnects the incident edges of the adjacency graph, and reconnects them in one of the possible other ways. The *DCJ distance* between genomes A and B, $d_{DCJ}(A, B)$, is the minimum number of DCJ operations necessary to transform genome A into genome B. We have:

Theorem 1 ([8]). *Let A and B be two genomes defined on the same set of N genes, then we have*

$$d_{DCJ}(A, B) \;=\; N - (C + I/2)$$

where C is the number of cycles and I the number of odd paths in $AG(A, B)$. An optimal sorting sequence can be found in $O(N)$ time.

A DCJ operation that reduces the DCJ distance by 1 is called *DCJ-sorting*. Using Theorem 1, we have the following property of DCJ-sorting operations, using the fact that a DCJ operation acts on at most two paths or cycles, and produces at most one new path or cycle:

Corollary 1. *A DCJ-sorting operation acts on a single path or cycle, or on two even paths of the adjacency graph.*

As we saw earlier, some DCJ operations can create intermediate circular chromosomes, even if both genomes A and B are linear, and we will want to avoid them in the HP model. The following definition is a generalization of a classical concept in rearrangement theory, *oriented operations*:

Definition 3. A DCJ-sorting operation is *oriented* if it does not create circular chromosomes.

For two linear genomes, oriented operations are necessarily inversions, translocations, fusions or fissions. These operations are also called *HP operations*, and the HP distance $d_{HP}(A, B)$ is the minimum number of HP operations needed to transform genome A into genome B. Since DCJ operations are more general than HP operations, the following lower bound is immediate:

Proposition 1. *For two linear genomes A and B, the following inequality holds:*

$$d_{DCJ}(A, B) \leq d_{HP}(A, B).$$

The non-negative number

$$t = d_H P - d_D CJ$$

thus counts the extra rearrangement operations that are necessary in order to avoid the creation of circular chromosomes. In the next section, we characterize genomes for which the two distances are equal, that is $t = 0$.

3. Components and Oriented Sorting

In this section, we introduce the notion of *components*. They roughly correspond to the classical concept of connected components of the overlap graph, but in the context of adjacency graphs, we prove that they are unions of paths and cycles.

3.1. Basic Definitions

Definition 4. Given two genomes A and B, an interval $(l, \ldots, r)$ of genome A is a *component* relative to genome B if there exists an interval in genome B:

a) with the same extremities,
b) with the same set of genes, and
c) that is not the union of two such intervals.

Example 1. Let

$$\begin{aligned} A &= \{(\circ, 2, 1, 3, 5, 4, \circ), (\circ, 6, 7, -11, -9, -10, -8, 12, 16, \circ), (\circ, 15, 14, -13, 17, \circ)\}, \\ B &= \{(\circ, 1, 2, 3, 4, 5, \circ), (\circ, 6, 7, 8, 9, 10, 11, 12, \circ), (\circ, 13, 14, 15, \circ), (\circ, 16, 17, \circ)\}. \end{aligned}$$

The components of genome A relative to genome B are: $(\circ, 2, 1, 3)$, $(3, 5, 4, \circ)$, $(\circ, 6)$, $(6, 7)$, $(-11, -9, -10, -8)$, $(7, -11, -9, -10, -8, 12)$, $(\circ, 15, 14, -13)$ and $(17, \circ)$.

Note that components of length 2 are the same adjacencies in both genomes, possibly up to flipping of a chromosome. These are called *trivial components*. All other components are *non-trivial*.

Two components are *nested* if one is included in the other and their extremities are different. As the following lemma shows, two components cannot share a telomere:

Lemma 1. *If* $(\circ, \ldots, r_1)$ *and* $(\circ, \ldots, r_1, \ldots, r_2)$ *are two components, then* $r_1 = r_2$*, and if* $(l_1, \ldots, l_2, \ldots, \circ)$ *and* $(l_2, \ldots, \circ)$ *are two components then* $l_1 = l_2$*.*

Proof. Suppose that $(\circ, \ldots, r_1)$ and $(\circ, \ldots, r_1, \ldots, r_2)$ are two components. Since the corresponding intervals in genome B, $(\circ, \ldots, r_1)$ and $(\circ, \ldots, r_2)$, share the same gene content, the interval $(r_1, \ldots, r_2)$ shares the same gene content in both genomes, thus $(r_1, \ldots, r_2)$ is a component, and $(\circ, \ldots, r_1, \ldots, r_2)$ is the union of two components, a contradiction. The second statement has a similar proof. □

It is further known that two components can not overlap on two or more elements. We thus have the following generalization of a statement from [9]:

Proposition 2. *Two components are either disjoint, nested, or overlap on exactly one gene.*

Proposition 2 implies that components can be partially ordered by inclusion, and that overlapping components will have the same parent, if it exists. An adjacency *properly belongs* to the smallest component that contains it.

Definition 5. The *adjacency graph of a component C* is the subgraph of the adjacency graph of genomes A and B induced by the adjacencies that properly belong to C.

An important property of the adjacency graph is the following:

Proposition 3. *The adjacency graph of a component is a union of paths and cycles of the adjacency graph of genomes A and B.*

Proof. Let $C = (l, \ldots, r)$ be a component. Since it has the same gene content and the same extremities as the corresponding interval in genome B, all edges of the adjacency graph that are within the interval $(l, \ldots, r)$ in genome A will also be within the interval $(l, \ldots, r)$ in genome A. Thus all these edges form a union of paths and cycles of the adjacency graph of genomes A and B.

Each component that is nested in C is also a union of paths and cycles of the adjacency graph of genomes A and B, and none of them contains an adjacency that properly belongs to C. We can thus remove them without compromising the connectivity of the adjacency graph of C. □

3.2. Oriented Sorting

In this section we will characterize genomes for which the DCJ distance and the HP distance are equal. We first consider sorting involving only one component. In particular, we apply well known results from the inversion theory that are obtained by working on permutations.

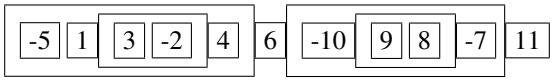
Since the naming and orientation of genes is relative, we can always assume that all genes in a chromosome of genome B are positive and in increasing order. The proper adjacencies of a component $C = (l, \ldots, r)$ induce a partition in the corresponding intervals of genomes A and B into sub-intervals that we will subsequently call *blocks*.

If we label the blocks in the interval of C in genome B with numbers from 1 to k, the corresponding blocks of the interval of C in genome A will be a signed permutation $(p_1, \ldots, p_k)$ of the *elements* $\{1, \ldots, k\}$. We will call this permutation – or its reverse – the *permutation associated to the component C*.

Consider for example the following two genomes:

$$\begin{aligned} A &= \{(\circ\ -5, 1, 3, -2, 4, 6, -10, 9, 8, -7, 11\ \circ)\}, \\ B &= \{(\circ\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,\ 9, 10, 11\ \circ)\}. \end{aligned}$$

The components of A with respect to B can easily be seen in the following diagram:



The component $(\circ, \ldots, 6)$ consists of three blocks: the gene -5, the component $(1, \ldots, 4)$ and the gene 6. Thus, the permutation associated to the component $(\circ, \ldots, 6)$ is $(-2, 1, 3)$. For the other three non-trivial components, the associated permutations are $(1, 3, -2, 4)$, $(-4, 3, 2, -1)$ and $(1, -2, 3)$.

As stated in Corollary 1, a DCJ-sorting operation increases either the number of cycles or the number of odd paths. When the elements of the permutation associated to a component have both positive and negative signs, then there exists a pair of consecutive elements with opposite signs. This implies that there exists a DCJ-sorting inversion that increases the number of cycles by creating an adjacency of the form $(i, i+1)$ or $(-(i+1), -i)$. For instance, the permutation $(-2, 1, 3)$ associated to component $(\circ, \ldots, 6)$ in the above example admits the DCJ-sorting inversion of element 1, creating the adjacency of elements $(-2, -1)$. This corresponds to the adjacency of genes $(-5, -4)$ in the component $(\circ, \ldots, 6)$, yielding a new genome

$$A' = \{(\circ\ -5, -4, 2, -3, -1, 6, -10, 9, 8, -7, 11\ \circ)\}.$$

Components whose associated permutations have only elements with the same sign are more intricate. We will see later that some of them can be optimally sorted by DCJ-sorting operations, others not. For example, consider the pair of genomes:

$$A = \{(\circ, 4, 3, 2, 1, \circ)\} \quad \text{and} \quad B = \{(\circ, 1, 2, 3, 4, \circ)\},$$

whose associated permutation is $(4, 3, 2, 1)$. There exists a DCJ-sorting operation that is a path fission increasing the number of odd paths. The DCJ distance is 4, and it can be optimally sorted by inverting each of the four genes. However, we have:

Lemma 2. *If all elements of the permutation associated to a component have the same sign, then no inversion acting on one of its paths or cycles can create a new cycle.*

Proof. By possibly flipping the chromosome, we can assume that all the elements of the permutation are positive. Suppose that an inversion is applied to two adjacencies $(+i, +j)$ and $(+k, +l)$ in a single path or cycle of the component, and that this creates a new cycle. The new adjacencies will be $(+i, -k)$ and $(-j, +l)$, where at most one of $+i$ and $+l$ can be a telomere. If both of these new adjacencies belong to the same path or cycle, there was no creation of a new cycle. Suppose that the adjacency $(+i, -k)$ belongs to the new cycle, then all other adjacencies of this cycle existed in the original component, and are composed of positive elements. This, however, is impossible by the construction of the adjacency graph. □

Definition 6. A component is *oriented* if there exists an oriented DCJ-sorting operation that acts on the vertices of its adjacency graph, otherwise it is *unoriented*.

Oriented components are characterized by the following:

Proposition 4. *A component is oriented if and only if either its associated permutation has positive and negative elements, or its adjacency graph has two even paths.*

Proof. If the associated permutation has positive and negative elements, then there is at least one change of signs between blocks labeled by consecutive integers. Thus, there exists an inversion that creates an adjacency in genome B, thus a new cycle, and the inversion is DCJ-sorting. If there are two even paths, then one of them must be a path from genome A to genome A, and the other one must be a path from genome B to genome B. An inversion in genome A that acts on one adjacency in each path creates two odd paths, thus is DCJ-sorting.

In order to show the converse, suppose that all elements of the associated permutation are positive, and all paths are odd. By Corollary 1, a DCJ-sorting operation must act on a single

path or cycle. This operation cannot be a translocation or a fusion since all paths and cycles of a component are within the same chromosome. This operation cannot be an inversion, since inversions that create new cycles are ruled out by Lemma 2, inversions acting on a single odd path cannot augment the number of odd paths, and inversions acting on cycles never create paths. Finally, this operation cannot be a fission: a fission acting on a cycle creates an even path; and a fission acting on an odd path must circularize one of the chromosome parts in order to be DCJ-sorting, otherwise it would be split into an even path and an odd path. □

As a consequence of Proposition 4, we have $d_{DCJ}(A, B) < d_{HP}(A, B)$ in the presence of unoriented components, since all DCJ-sorting operations will create circular chromosomes. On the other hand, well-known results from the Hannehalli-Pevzner theory show that, when all components admit a sorting inversion, then it is possible to create a new cycle at each step of the sorting process with HP operations, without creating unoriented components [13]. We will see in the next propositions that the same type of result can be obtained in this context. According to Proposition 4, it will be useful to treat components with paths and components without paths separately.

Definition 7. Components whose two extremities are both genes, are called *real* components. Components that contain one or two telomeres are *semi-real* components.

First, let us consider oriented real components that are well studied in the context of sorting by inversions. In [13], it was first shown that oriented real components can be sorted optimally by oriented inversions. This relies on the fact that among all possible oriented inversions there always exists one that does not create new unoriented components. Such an inversion is called a *safe* inversion and can be found by trial and error. More sophisticated techniques to efficiently find safe inversions can be found in [2, 3, 15].

Proposition 5 ([13]). *An oriented real component has an oriented DCJ-sorting operation that does not create new unoriented components.*

As a consequence of Proposition 5, it is possible to sort real components with oriented DCJ-sorting inversions as shown in the next proposition.

Proposition 6. *A real component can be sorted with oriented DCJ-sorting operations if and only if it is oriented.*

Proof. If a real component can be sorted by oriented DCJ-sorting operations, then, by definition, the component is oriented. If a real component is oriented, then there exists an oriented DCJ-sorting inversion and Proposition 5 guarantees that there will always be enough oriented DCJ-sorting inversions to sort the component. □

Now, we come to the semi-real components. First, we consider semi-real components whose associated permutation has positive and negative elements. As it turns out, these components can be treated in the same way as oriented real components by adding extra genes.

Proposition 7. *An oriented semi-real component whose associated permutation has positive and negative elements can be sorted with oriented DCJ-sorting operations.*

Proof. We will show that such components can be embedded in oriented real components with the same DCJ distance. Then, we can sort the component with oriented DCJ-sorting operations. Let $(p_1, \ldots, p_k)$ be the permutation of the *elements* $\{1, \ldots, k\}$ associated to a semi-real component. Suppose first that $p_k = k$, then we must have $p_1 \neq 1$, otherwise the component would be real. This implies that, in the adjacency graph of $(\circ, p_1, \ldots, p_k, \ldots)$ and $(\circ, 1, \ldots, k, \ldots)$, there is an odd path that joins the telomeres at p_1 and 1. Adding a gene number 0 next to both telomeres transforms the odd path into a cycle and an odd path of length 1, thus preserves the DCJ distance. The new component is real. If $p_1 = 1$, a similar arguments holds.

If both $p_1 \neq 1$ and $p_k \neq k$, then the telomere at 1 is either joined by an odd path to p_1 or p_k. In the second case, reverse the permutation and rename the elements. Thus we can assume that the telomere at 1 is joined to p_1, and the telomere at k is joined, also by an odd path, to p_k. Adding a gene 0 next to both telomeres 1 and p_1, and a gene $k+1$ next to both telomeres k and p_k transforms the two odd paths into cycles, and adds two odd paths of length 1, thus also preserves the DCJ distance. □

Oriented components must sometimes be sorted with fissions. This is the case for semi-real components with even paths, but care must be taken on which fission to apply. Such components have two paths, one of them connecting two telomeres of A and one connecting two telomeres of B, as in genomes:

$$A = \{(\circ, 4, 2, 1, 3, \circ)\} \quad \text{and} \quad B = \{(\circ, 1, 2, 3, 4, \circ)\}.$$

A fission creating telomere $(4, \circ)$ would create the new semi-real component $(1, \ldots, 3)$, but the fission creating telomere $(\circ, 1)$ is safe. In general, we have:

Proposition 8. *An oriented semi-real component whose adjacency graph has two even paths can be sorted with oriented DCJ-sorting operations.*

Proof. Let $C = (l, \ldots, r)$ be a semi-real component with two even paths. Consider the permutation $(p_1, \ldots, p_k)$ associated to C. If the permutation is oriented, then it is possible to sort the component with oriented DCJ-sorting operations by Proposition 7.

If the permutation is unoriented, then all genes p_1 to p_k have the same sign. Without loss of generality, we assume that they are all positive. The path connecting the two telomeres of genome B implies two possible fissions: fission F_k creating telomere $(k, \circ)$ and fission F_1 creating $(\circ, 1)$. If one of these fissions creates a new unoriented component, then it must be a semi-real component, otherwise it must have existed before the fission. This implies that if F_k creates a new component, this component contains element 1, and if F_1 creates a new component, this component contains element k.

Suppose that $p_i = 1$ precedes $p_j = k$. If fission F_k creates a new component, it must contain element 1, but since the interval also contains element k, it must contain all elements. The same argument applies to fission F_1.

Now suppose that $p_i = k$ precedes $p_j = 1$, and that both F_k and F_1 create new components. Then, one chromosome of genome A is given by

$$(\circ, p_1, \ldots, p_{i-1},\ k,\ p_{i+1}, \ldots, p_{j-1},\ 1,\ p_{j+1}, \ldots, p_k, \circ).$$

Let $(l_1, \ldots, r_1)$ and $(l_2, \ldots, r_2)$ be unoriented components created by fissions F_k, respectively F_1. Since fission F_k yields the chromosomes

$$(\circ, p_1, \ldots, p_{i-1},\ k, \circ) \text{ and } (\circ,\ p_{i+1}, \ldots, p_{j-1},\ 1,\ p_{j+1}, \ldots, p_k, \circ),$$

the newly created component $(l_1, \ldots, r_1)$ must contain element 1 and be in the beginning of the second chromosome, otherwise it would have existed before. On the other hand, fission F_1 creates chromosomes:

$$(\circ, p_1, \ldots, p_{i-1},\ k,\ p_{i+1}, \ldots, p_{j-1}, \circ) \text{ and } (\circ, 1,\ p_{j+1}, \ldots, p_k, \circ).$$

By a similar argumentation, the component $(l_2, \ldots, r_2)$ is at the end of the first chromosome and contains element k. This means that the genes in the set $\{p_{i+1}, \ldots, p_{j-1}\}$ are in the beginning and at the end of a chromosome of B. But, since each gene occurs exactly once, the set $\{p_{i+1}, \ldots, p_{j-1}\}$ must be empty, implying that $i + 1 = j$. Thus, both fissions create the following two chromosomes

$$(\circ, p_1, \ldots, p_{i-1},\ k, \circ) \text{ and } (\circ, 1,\ p_{j+1}, \ldots, p_k, \circ).$$

But, a component $(l_1, \ldots, r_1)$ in the end of the first chromosome would be a real component. This contradicts the fact that the component was newly created by the fission. Also, contradicting our assumption, $(l_2, \ldots, r_2)$ cannot be created by a fission. Thus, we have shown that at least one of the two fissions does not create any new unoriented components. □

Until now, we have dealt only with single components. Now, we will turn to the general problem of sorting linear genomes. In addition to inversions, this requires DCJ operations that change the gene content of one or more chromosomes of genome A, which are translocations, fusions and fissions, called *generalized translocations*. Generalized translocations never create circular chromosomes, thus we have:

Proposition 9. *Given two linear genomes A and B, if at least one adjacency of A does not belong to a component, then there exists at least one sorting generalized translocation.*

Proof. Suppose that there exist no sorting generalized translocations, i.e., the only oriented DCJ operations are inversions. This implies that all other possible DCJ-sorting operations create circular chromosomes, thus act on a single chromosome of genome A. Thus all chromosomes of genome A have a corresponding chromosome in genome B with the same gene content.

When two chromosomes in genome A and B have the same gene content, either the whole chromosome is a component, or it is a union of components, since the whole interval satisfies conditions a) and b) of Definition 4. Thus all adjacencies belong to a component. □

The next proposition is the key, it says that for any generalized translocation that creates an unoriented component there always exists an alternative DCJ-sorting operation that does not. This statement, already proven in the context of sorting by internal translocations [7], is here extended to the general case of multi-chromosomal genomes.

Proposition 10. *Given two linear genomes A and B, if a sorting generalized translocation creates an unoriented component, then there exists another sorting generalized translocation that does not.*

Proof. Consider all sorting generalized translocations, and let C be a smallest unoriented component created by any of them, let's say translocation T. We will show that there is always an alternative generalized translocation that does not create unoriented components, or C is not the smallest. We will distinguish three cases:

1. $C = (l, \ldots, r)$

2. $C = (l, \ldots, \circ)$ [or the symmetric case $(\circ, \ldots, r)$]

3. $C = (\circ, \ldots, \circ)$

In each case, we will distinguish whether a) elements of C belong to different chromosomes of genome A, or b) they all belong to the same chromosome.

Case 1.a In this case, genome A can be written as

$$(\circ, \ldots, l, \ldots, a, \ldots, \circ) \text{ and } (\circ, \ldots, b, \ldots, r, \ldots, \circ)$$

and component C as $(l, \ldots, a, b, \ldots, r)$. Since components must have more than two elements, we may assume that $l \neq a$. The case $b \neq r$ is symmetrical.

The goal is to find an alternative translocation that creates an adjacency $(M, M+1)$ such that $M \in (l, \ldots, a)$ but different from a, and $M+1 \in (b, \ldots, r)$. If $a = \max(l, \ldots, a)$, choose M as the smallest element of $(l, \ldots, a)$ such that $M+1$ is in $(b, \ldots, r)$. Such an element M always exists, otherwise $(l, \ldots, a)$ would already be a component in genome A, thus C would be the union of two smaller components. If $a \neq \max(l, \ldots, a)$, set M to $\max(l, \ldots, a)$.

The two chromosomes of genome A can therefore be written as

$$(\circ, \ldots, l, \ldots, M, \ldots, a, \ldots, \circ) \text{ and } (\circ, \ldots, b, \ldots, M+1, \ldots, r, \ldots, \circ).$$

Applying the translocation that creates adjacency $(M, M+1)$ results in a genome A'' with chromosomes:

$$(\circ, \ldots, l, \ldots, M, M+1, \ldots, r, \ldots, \circ) \text{ and } (\circ, \ldots, b, \ldots, a, \ldots, \circ).$$

Since both cuts of the new translocation are between elements of the interval $(l, \ldots, r)$, and elements of these intervals are on different chromosomes in genome A', any newly created component would be strictly shorter than $(l, \ldots, r)$. The new translocation is sorting since it creates a cycle corresponding to the adjacency $(M, M+1)$.

Case 1.b If all elements of the interval $C = (l, \ldots, r)$ were in the same chromosome in genome A, C was already a real component in genome A.

Case 2.a In this case, genome A can be written as

$$(\circ, \ldots, l, \ldots, a, \ldots, \circ) \text{ and } (\circ, \ldots, b, \ldots, \circ)$$

and component C as $(l, \ldots, a, b, \ldots, \circ)$.

If $l = a$, then b must be different from $a+1$, otherwise C would be the union of two smaller components, thus setting $M = a$ provides an alternative translocation.

If $l \neq a$, the argument is similar to Case 1.a.

Case 2.b In this case, genome A can be written as

$$(\circ, \ldots, l, \ldots, a, \ldots, \circ) \text{ and } (\circ, \ldots, \circ)$$

and component C as $(l, \ldots, a, \circ)$. We have $a \neq \max(l, \ldots, a)$, otherwise C would have already been a real component in genome A. Let $M = \max(l, \ldots, a)$, and consider the generalized translocation that creates telomere $(M, \circ)$ instead of telomere $(a, \circ)$.

We must first prove that this new translocation is sorting. Since $(l, \ldots, a, \circ)$ is a semi-real component, this implies that both adjacencies $(a, \circ)$ in genome A' and $(M, \circ)$ in genome B are on the same odd path of the new adjacency graph of A' and B, thus were on the same path P of the adjacency graph of genomes A and B, implying that both translocations acted on two adjacencies of path P, sharing one of them. Since T either creates a cycle or two new odd paths, it is also

true for the new translocation because it creates a path of length 1 defined by the telomere $(M, \circ)$ in both genomes A' and B.

Finally, any newly created component must contain M and not a, thus would be strictly shorter than C.

Case 3.a In this case, genome A can be written as

$$(\circ, \ldots, a, \ldots, \circ) \text{ and } (\circ, \ldots, b, \ldots, \circ)$$

and component C as $(\circ, \ldots, a, b, \ldots, \circ)$.

Suppose that interval $(\circ, \ldots, a)$ has at least two elements, thus can be written $(\circ, a', \ldots, a)$.

If there exists an element M different from a in interval $(\circ, a', \ldots, a)$ such that $M + 1$ belongs to $(b, \ldots, \circ)$, creating the adjacency $(M, M + 1)$ provides an alternative translocation.

Else, every element, except a, has its successor in $(\circ, a', \ldots, a)$, implying that $a = \max(\circ, a', \ldots, a)$. Furthermore, we have $a' \neq \min(\circ, a', \ldots, a)$, since this would make $(\circ, a', \ldots, a)$ a real component in genome A. Let $m = \min(\circ, a', \ldots, a)$. Element m cannot be the minimum of interval $C = (\circ, \ldots, a, b, \ldots, \circ)$, since, this time, this would make $(\circ, a', \ldots, a)$ a semi-real component in genome A. Thus $m - 1$ belongs to interval $(b, \ldots, \circ)$, and creating the adjacency $(m - 1, m)$ provides an alternative translocation.

Both alternative translocations have at least one cut between two elements of interval $(a', \ldots, a)$, thus any new unoriented component would be shorter than C.

The argument is similar if interval $(b, \ldots, \circ)$ has at least two elements.

Case 3.b Here we suppose that a becomes telomere $(a, \circ)$. The case when a becomes telomere $(\circ, a)$ is symmetrical. In this case, genome A can be written as

$$(\circ, \ldots, a, \ldots, \circ) \text{ and } (\circ, \ldots, \circ)$$

and component C as $(\circ, \ldots, a, \circ)$. Again, we have $a \neq \max(\circ, \ldots, a)$, otherwise C would have already been a semi-real component in genome A. The rest of the argument is similar to Case 2.b. □

Finally, we have all necessary ingredients for the main result of this section.

Theorem 2. *Given two linear genomes A and B, $d_{HP}(A, B) = d_{DCJ}(A, B)$ if and only if there are no unoriented components.*

Proof. The "if" part comes from the fact that we can sort a genome without unoriented components with DCJ-sorting operations (Propositions 7, 8 and 10), adding the fact that semi-real components whose graphs have even paths can be "destroyed" by fissions. The "only if" part comes from the fact that if there are unoriented components, then $d_{DCJ}(A, B) < d_{HP}(A, B)$, since we showed in Proposition 4 that all DCJ-sorting operations create circular chromosomes in these cases. □

4. Computing the General HP Distance

In this section we will show that, given the DCJ distance d_{DCJ}, one can express the Hannenhalli-Pevzner distance d_{HP} in the form

$$d_{HP} = d_{DCJ} + t,$$

where t represents the additional cost of not resorting to unoriented DCJ operations. First, we describe how to destroy unoriented components in Section 4.1 and after that, in Section 4.2, we compute the additional cost from the inclusion and linking tree of the unoriented components.

4.1. Destroying Unoriented Components

Destroying unoriented components can done by applying a DCJ operation either on one component in order to orient it, or on two components on the same chromosome in order to merge them, and possibly others, into a single oriented component. When a DCJ is applied to components on different chromosomes, the components are destroyed together with all components that contain them. By using the nesting and linking relationship between components, one can minimize the number of operations necessary to destroy unoriented components.

When two components overlap on one element, we say that they are *linked*. Successive linked components form a *chain*. A chain that cannot be extended to the left or right is called *maximal*. We represent the nesting and linking relations between components of a chromosome in the following way:

Definition 8. Given a chromosome X of genome A and its components relative to genome B, define the forest F_X by the following construction:

1. Each non-trivial component is represented by a round node.
2. Each maximal chain that contains non-trivial components is represented by a square node whose (ordered) children are the round nodes that represent the non-trivial components of this chain.
3. A square node is the child of the smallest component that contains this chain.

Now, we define a tree associated to the components of a genome by combining the forests of all chromosomes into one rooted tree:

Definition 9. Suppose genome A consists of chromosomes $\{X_1, X_2, \ldots, X_K\}$. The tree T associated to the components of genome A relative to genome B is given by the following construction:

1. The root is a round node.
2. All trees of the set of forests $\{F_{X_1}, F_{X_2}, \ldots, F_{X_K}\}$ are children of the root.

The round nodes of T are *painted* according to the following classification:

1. The root and all nodes corresponding to oriented components are painted *black*.
2. Nodes corresponding to unoriented components that do not contain telomeres are painted *white*.
3. Nodes corresponding to unoriented components that contain one or two telomeres are painted *grey*.

The tree associated to the components of the genomes A and B of Example 1 is shown in Fig. 1. Note that grey nodes are never included into other components.

The following two propositions are general remarks on components and are useful to show how to destroy unoriented components.

Proposition 11 ([7]). *A translocation acting on two components cannot create new components.*

Proposition 12 ([6]). *An inversion acting on two components $\mathcal{A}$ and $\mathcal{B}$ creates a new component $\mathcal{D}$ if and only if $\mathcal{A}$ and $\mathcal{B}$ are included in linked components.*

Now, we have all necessary results to get rid of unoriented components. The following two propositions are straightforward generalizations of well-known results from the sorting by inversion theory [5]. We will start by looking at one single unoriented component.

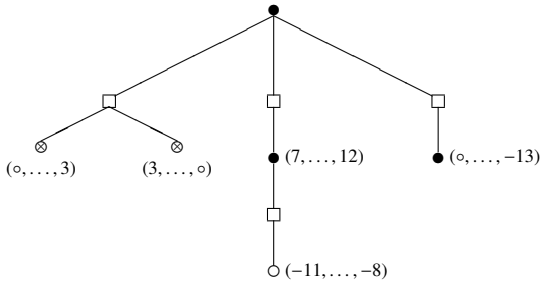


Figure 1: The tree T associated to the genomes A and B of Example 1 has two grey leaves, one white leaf and one black leaf.

Proposition 13. *If a component C is unoriented, any inversion between adjacencies of the same cycle or the same path of C orients C, and leaves the number of cycles and paths of the adjacency graph of C unchanged.*

Orienting a component as in Proposition 13 is called *cutting* the component. Note that this operation leaves the DCJ distance unchanged, and does not create new components.

It is possible to destroy more than one unoriented component with a DCJ operation acting on two unoriented components. The following proposition describes how to *merge* several components, and the relations of this operation to paths in the tree T.

Proposition 14. *There always exists an HP operation acting on adjacencies of two different unoriented components $\mathcal{A}$ and $\mathcal{B}$ that destroys, or orients, all components on the path from $\mathcal{A}$ to $\mathcal{B}$ in the tree T, without creating new unoriented components or circular chromosomes.*

Proof. If $\mathcal{A}$ and $\mathcal{B}$ are in different chromosomes, then we apply a translocation to destroy $\mathcal{A}$ and $\mathcal{B}$. As stated in Proposition 11, such a translocation cannot create new unoriented components. Moreover, all components that contain either $\mathcal{A}$ or $\mathcal{B}$ are also destroyed by the translocation. Thus, all components on the path are destroyed.

If $\mathcal{A}$ and $\mathcal{B}$ are in the same chromosome, then we apply an inversion to destroy $\mathcal{A}$ and $\mathcal{B}$. By Proposition 12, the only component eventually created by an inversion is the union of two or more linked components. Since linked components have extremities with the same sign, the sign of the links will be different from the sign of the extremities of the new component, thus it will be oriented.

Now, if the components $\mathcal{A}$ and $\mathcal{B}$ are not included in linked components, then the lowest common ancestor must be a round node, either the root, or a component C that is on the path from $\mathcal{A}$ to $\mathcal{B}$ and that contains both. In the latter case, components $\mathcal{A}$ and $\mathcal{B}$ must be real, thus the inversion merges two cycles, one from each component, into one new cycle. If $\mathcal{A}$ and $\mathcal{B}$ are unoriented, the new cycle contains at least one oriented adjacency. Since C is the smallest component that contains the new cycle, C will be oriented.

Finally, for any component C on the path from $\mathcal{A}$ to $\mathcal{B}$ and that contains either $\mathcal{A}$ or $\mathcal{B}$, but not both, the inversion changes the sign of one of the extremities of C, and C will be destroyed. □

If the HP operation acts on two odd paths, thus on grey components, then merging the two components can be done without changing the number of odd paths, and the DCJ distance is

unchanged. If the HP operation involves at least one cycle, then merging two components decreases the number of cycles by one, and the DCJ distance will increase by 1 in the resulting pair of genomes.

4.2. Unoriented Sorting

Let T be the tree associated to the components of genome A relative to genome B, and let T' be the smallest subtree of T that contains all the unoriented components, that is, the white and grey nodes.

Definition 10. A *cover* of T' is a collection of paths joining all the unoriented components, such that each terminal node of a path belongs to a unique path.

A path that contains two or more white or grey components, or one white and one grey component, is called a *long* path. A path that contains only one white or one grey component, is a *short* path.

The *cost* of a cover is defined to be the sum of the costs of its paths, where the cost of a path is the increase in distance caused by destroying the unoriented components along the path. More precisely, the cost of a path represents the sum of one extra HP operation and the variation of the DCJ distance. Using the remarks following Propositions 13 and 14, we have:

1. The cost of a short path is 1.
2. The cost of a long path with only two gray components is 1.
3. The cost of all other long paths is 2.

An *optimal* cover is a cover of minimal cost. Define t as the cost of any optimal cover of T'. We first establish that t is the difference between the two distances, using the following terminology:

Definition 11. Given genomes A and B, we call a DCJ operation applied to genome A

- *proper*, if it decreases $d_{DCJ}(A, B)$ by one, i.e. $\Delta(C + I/2) = 1$,
- *improper*, if $d_{DCJ}(A, B)$ remains unchanged, i.e. $\Delta(C + I/2) = 0$, and
- *bad*, if it increases $d_{DCJ}(A, B)$ by one, i.e. $\Delta(C + I/2) = -1$.

Theorem 3. *If t is the cost of an optimal cover of T', the smallest subtree of T that contains all the unoriented components of genome A relative to genome B, then:*

$$d_{HP}(A, B) = d_{DCJ}(A, B) + t.$$

Proof. First, we will show that $d_{HP}(A, B) \leq d_{DCJ}(A, B) + t$. Consider any cover of the tree T'. Let

- ww be the number of long paths with only white components,
- wg be the number of long paths with white and grey components,
- gg be the number of long paths with only grey components,
- w be the number of short paths with one white component,

- g be the number of short paths with one grey component.

Clearly, we have that the cost t' of this cover is $t' = 2ww + 2wg + gg + w + g$.

Suppose that the adjacency graph $AG(A, B)$ has C cycles and I odd paths. Applying $ww + wg$ bad DCJ operations and $gg + w + g$ improper DCJ operations yields a genome A'. Since each bad DCJ operation merges two cycles or one cycle and a path, the number of cycles in $AG(A', B)$ is $C - ww - wg$. Note that the number of odd paths remains unchanged. Therefore, by Theorem 2, we have that

$$\begin{aligned} d_{HP}(A, B) &\leq d_{HP}(A', B) + ww + wg + gg + w + g \\ &= N - (C + \frac{I}{2}) + 2ww + 2wg + gg + w + g \\ &= d_{DCJ}(A, B) + t'. \end{aligned}$$

Thus, since the above equation is true for any cover, we have: $d_{HP}(A, B) \leq d_{DCJ}(A, B) + t$.

The fact that $d_{HP}(A, B) \geq d_{DCJ}(A, B) + t$ is a consequence of the fact that an optimal sorting with HP operation necessarily induces a cover of T' since all unoriented components are eventually destroyed. □

It remains to establish a closed formula for t. A first easy but significant result on the size of t is the following lower bound. Let w be the number of white leaves and g be the number of grey leaves in T'. Since destroying a white leaf costs at least 1 and destroying a grey leaf costs at least $1/2$, and t is an integer, we have:

$$w + \left\lceil \frac{g}{2} \right\rceil \leq t.$$

It is quite remarkable, as was observed in the original paper on HP distance [12], that this bound is at most within one rearrangement operation from the optimal solution, i.e. $t = w + \left\lceil \frac{g}{2} \right\rceil$ or $t = w + \left\lceil \frac{g}{2} \right\rceil + 1$.

A branch in a tree is called a *long branch* if it has two or more unoriented components. A tree is called a *fortress* if it has an odd number of leaves, all of them on long branches. A standard theorem of the sorting by inversion theory states that the minimal cost to cover a tree that is not a fortress is ℓ, the number of leaves of the tree, and $\ell + 1$ in the case of a fortress [13].

The closed formula for t is now given in the following two theorems.

Theorem 4. *Let w be the number of white leaves and g be the number of grey leaves in T', the smallest subtree of T that contains all the unoriented components of genome A relative to genome B. If the root of T' has more than one child with white leaves, then the minimal cost of a cover of T' is:*

$$\begin{array}{ll} t = w + \lceil \frac{g}{2} \rceil & \textit{if the smallest subtree } T'' \textit{ that contains all the white leaves} \\ & \textit{of } T' \textit{ is not a fortress, or } g \textit{ is odd,} \\ t = w + \lceil \frac{g}{2} \rceil + 1 & \textit{otherwise.} \end{array}$$

Proof. If the subtree T'' is not a fortress then it admits a cover of cost w, and pairing the maximum number of grey nodes yields a cover of T' costing $w + \lceil \frac{g}{2} \rceil$. If the subtree T'' is a fortress, then one of its white leaves is not paired with another leaf since the number of leaves is odd. A cover of T' can be obtained by pairing this white leaf with a grey leaf, which exists if g is odd.

The resulting cost will be again $w + \lceil \frac{g}{2} \rceil$ which equals the lower bound, and thus the cover is optimal.

If the subtree is a fortress and g is even, we can construct a cover costing $(w + 1) + g/2$, using the cover of the fortress and pairing the grey nodes. To show that this cost is minimal, suppose that k grey nodes are paired with k white nodes, and the remaining white and grey nodes are paired separately. If k is even, then the cost of such a cover would be $(w - k + 1) + (g - k)/2 + 2k$, which is greater than or equal to $(w + 1) + g/2$. If k is odd, then the cost of this cover is $(w - k) + (g - k + 1)/2 + 2k$, which is again greater than or equal to $(w + 1) + g/2$. □

When all the white leaves belong to a single child of the root, the situation is more delicate. Define a *junior fortress* as a tree with an odd number of white leaves, all of them on long branches, except one that is alone on its branch, called the *top* of the fortress. We have the following:

Theorem 5. *Let $w > 0$ be the number of white leaves and $g > 0$ be the number of grey leaves in T', the smallest subtree of T that contains all the unoriented components of genome A relative to genome B. If the root of T' has only one child c with white leaves then the minimal cost of a cover of T' is:*

$$\begin{array}{ll} t = w + \lceil \frac{g}{2} \rceil & \textit{if } g \textit{ is odd and the subtree } T_c \textit{ that is rooted at } c \\ & \textit{is neither a fortress nor a junior fortress with } c \textit{ as its top,} \\ t = w + \lceil \frac{g}{2} \rceil + 1 & \textit{otherwise.} \end{array}$$

Proof. Suppose first that $g = 1$, then the only grey leaf either belongs to T_c or not. In the first case, this grey leaf must be the child c implying that T_c is not a junior fortress. If T_c is not a fortress, then there exists a cover with minimal cost equal to the number of leaves of T_c, which is given by $w + \lceil \frac{g}{2} \rceil$, since $g = 1$. If T_c is a fortress, then the minimal cost of a cover is $w + \lceil \frac{g}{2} \rceil + 1$.

In the other case, i.e. the grey leaf does not belong to T_c, then if T_c is a fortress or a junior fortress with c as its top, the whole tree T' is a fortress with $w + \lceil \frac{g}{2} \rceil$ leaves, yielding a cost of $w + \lceil \frac{g}{2} \rceil + 1$. Otherwise, if T_c is neither a fortress nor a junior fortress, then T' can not be a fortress, and hence can be destroyed with cost $w + 1 = w + \lceil \frac{g}{2} \rceil$.

The same argumentation holds for any $g > 1$ if g is odd.

Now, we consider the case $g = 2$. If T_c is a fortress, two of the white leaves in T_c can be paired with the two grey leaves outside T_c at cost 4. This eliminates the two grey leaves, two of the long white branches, and the branch containing c. The remaining $w - 2$ long branches are paired at cost $w - 2$. Together, this gives a cover of cost $4 + w - 2 = w + \lceil \frac{g}{2} \rceil + 1$. This is optimal since the cost of T' is the same as for T_c. If T_c is not a fortress, we do not need to pair white and grey leaves. T_c can be covered with cost $w + 1$ and the g grey leaves are paired with cost $\lceil \frac{g}{2} \rceil$, giving again a total cost of $w + \lceil \frac{g}{2} \rceil + 1$.

If $g > 2$ and g is even, it is always possible to pair the grey leaves, as long as there are more than two left, and then apply the case $g = 2$. This gives the same cost $w + \lceil \frac{g}{2} \rceil + 1$. □

For example, the genomes A and B of Example 1 have $N = 17$ genes. The adjacency graph $AG(A, B)$ has $C = 3$ cycles and $I = 6$ odd paths. After removing the dangling black leaf, the tree T' has $g = 2$ grey leaves and $w = 1$ white leaf (see Fig. 1). Therefore, by Theorem 5, we have $t = 2$ and thus

$$d_{HP}(A, B) = N - (C + \frac{I}{2}) + t = 17 - (3 + 3) + 2 = 13.$$

5. Algorithms

The goal of this section is to present a linear time algorithm to compute the HP distance between two linear genomes based on Theorem 3. We first give an overview of this algorithm, and then present the individual steps in more detail in Sections 5.1–5.3.

Assume that two linear genomes A and B are given, then the algorithm consists of five steps:

1. Construct the adjacency graph $AG(A, B)$;
2. Compute the cycles and paths of $AG(A, B)$;
3. Compute the components of A with respect to B;
4. Construct the trees T and T' associated to genomes A and B;
5. Compute the cost t of an optimal cover of T'.

5.1. Construction of the Adjacency Graph

The first two steps can be solved in linear time as shown in [1, 8]. The construction of the adjacency graph and the DCJ distance computation are given in Algorithm 1 in [8]. By a simple extension of the data structure described there, one can also store for each adjacency whether it belongs to a cycle, an even or an odd path.

5.2. Identification and Classification of Components

For two multichromosomal genomes A and B defined on the set of genes $\{1, \ldots, N\}$, we give an algorithm to compute the components of genome A with respect to B. In the following, we modify the algorithm for unichromosomal genomes [6] such that not only real, but also semi-real components are identified.

Equivalently to Definition 4, a *real* component can be described as an interval from i to $(i + j)$ or from $-(i + j)$ to $-i$, for some $j > 0$, whose set of unsigned elements is $\{i, \ldots, i + j\}$, and that is not the union of two such intervals [6]. The elements i and $i + j$ are called the *bounding elements*. If the bounding elements have positive sign, then the component is called *direct*, otherwise the component is a *reversed* component.

In order to extend the notion of direct and reversed components to semi-real components, we replace the telomere markers by *caps* such that the left and the right end of a chromosome are different and that they also have a sign. More concretely, we transform the genome A into two strings: First, we add 0 at the beginning and $N + 1$ at the end of each chromosome of genome A and chain its chromosomes into a string P^+. Similarly, we add $-(N + 1)$ at the beginning and -0 at the end of each chromosome of genome A and chain its chromosomes into a string P^-. If genome A consists of K chromosomes, then both strings P^+ and P^- have length $l = N + 2K$.

Compared to the uni-chromosomal case, we need to do some extra work in order to identify components: The chromosomes of B are numbered consecutively and an array c stores for each gene in A its chromosome number in B. More precisely, if $P^+[i] = P^-[i]$, then $c[i]$ is the chromosome number in B of the element at index i. Otherwise, at index i is a chromosome end and $c[i]$ gets the chromosome number of its adjacent gene.

Now, with the help of array c, we are able to describe components of genome A and B as intervals in the strings P^+ and P^-. Depending whether the bounding elements are caps or genes, there are eight different types of components: four direct components shown at the left and four reversed components shown at the right of Fig. 2.

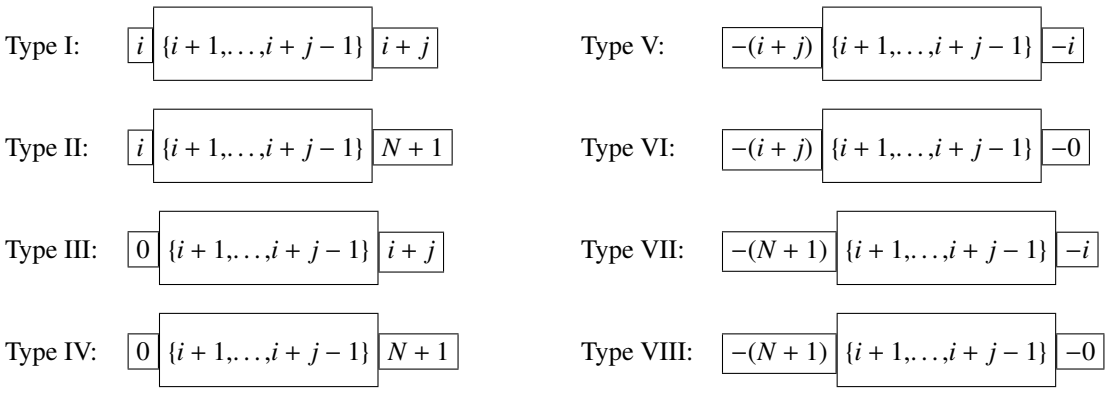


Figure 2: **Left:** Direct components. **Right:** Reversed components.

Example 2. Consider the genomes A and B of Example 1 with $N = 17$ genes:

$$\begin{aligned} A &= \{(\circ, 2, 1, 3, 5, 4, \circ), (\circ, 6, 7, -11, -9, -10, -8, 12, 16, \circ), (\circ, 15, 14, -13, 17, \circ)\}, \\ B &= \{(\circ, 1, 2, 3, 4, 5, \circ), (\circ, 6, 7, 8, 9, 10, 11, 12\circ), (\circ, 13, 14, 15, \circ), (\circ, 16, 17, \circ)\}. \end{aligned}$$

By replacing the telomere markers by the caps 0 and 18 (or -18 and -0), we get the string P^+ (or P^- respectively) and the array c that stores the chromosome in B for each gene:

P^+	:	0	2	1	3	5	4	18	0	6	7	−11	−9	−10	−8	12	16	18	0	15	14	−13	17	18
P^-	:	−18	2	1	3	5	4	−0	−18	6	7	−11	−9	−10	−8	12	16	−0	−18	15	14	−13	17	−0
c	:	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2	4	4	3	3	3	3	4	4

The components of genome A relative to genome B can be classified according to their bounding elements. There are six direct components: $(0, \ldots, 3)$ and $(0, 6)$ are of type III, $(3, \ldots, 18)$ and $(17, 18)$ are of type II, and $(6, 7)$ and $(7, \ldots, 12)$ are of type I. The two remaining components are reversed, the real component $(-11, \ldots, -8)$ of type V and the semi-real component $(-18, \ldots, -13)$ of type VII.

5.2.1. Preprocessing

The input for the component identification algorithm is a string of length l, either P^+ or P^-, separated into an array of unsigned elements $\pi = (\pi_0, \pi_1, \ldots, \pi_{l-1})$ and an array of signs $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_{l-1})$.

For each string P^+ and P^-, two further arrays M and m are computed beforehand. The array M is defined as follows: $M[i]$ is the nearest unsigned element of π that precedes π_i, and is greater than π_i, and $N + 1$ if no such element exists, or if index i is a cap. Similarly, $m[i]$ is the nearest unsigned element of π that precedes π_i and is smaller than π_i, and 0 if no such element exists or if index i is a cap.

Like in the unichromosomal case [6], the arrays M and m can efficiently be computed using two stacks M_1 and M_2, as shown for P^+ in Algorithm 1. For example, the arrays pi^+, σ^+, M and m of the string P^+ of Example 2 are given by:

π^+	:	0	2	1	3	5	4	18	0	6	7	11	9	10	8	12	16	18	0	15	14	13	17	18
σ^+	:	+	+	+	+	+	+	+	+	+	+	−	−	−	−	+	+	+	+	+	+	−	+	+
M	:	18	18	2	18	18	5	0	18	18	18	18	11	11	10	18	18	0	18	18	15	14	18	0
m	:	0	0	0	1	3	3	0	0	0	6	7	7	9	7	8	12	0	0	0	0	0	13	0

The corresponding arrays for the other string P^- are computed analogously, and we get:

π^-	:	18	2	1	3	5	4	0	18	6	7	11	9	10	8	12	16	0	18	15	14	13	17	0
σ^-	:	−	+	+	+	+	+	−	−	+	+	−	−	−	−	+	+	−	−	+	+	−	+	−
M	:	18	3	3	5	18	18	18	18	7	11	12	10	12	12	16	18	18	18	17	17	17	18	18
m	:	0	1	0	0	4	0	0	0	0	0	9	8	8	0	0	0	0	0	14	13	0	0	0

Algorithm 1 (Preprocessing for $P^+ = (\pi^+, \sigma^+)$, adapted from [6])

M_1 and M_2 are stacks of integers; initially M_1 contains $N + 1$ and M_2 contains 0

for $i \leftarrow 0, \ldots, l - 1$ **do**
  $M[i] \leftarrow N + 1$
  $m[i] \leftarrow 0$
  i++
  while $\pi^+[i] \neq N + 1$ **do**

    (* Compute the entry $M[i]$ *)
    if $\pi^+[i - 1] > \pi^+[i]$ **then**
      push $\pi^+[i - 1]$ on M_1
    else
      pop from M_1 all entries that are smaller than $\pi^+[i]$
    end if
    $M[i] \leftarrow$ the top element of M_1

    (* Compute the entry $m[i]$ *)
    if $\pi^+[i - 1] < \pi^+[i]$ **then**
      push $\pi^+[i - 1]$ on M_2
    else
      pop from M_2 all entries that are larger than $\pi[i]$
    end if
    $m[i] \leftarrow$ the top element of M_2

    i++
  end while
  $M[i] \leftarrow N + 1$
  $m[i] \leftarrow 0$
end for

5.2.2. Component Identification

The component identification algorithm consists of two phases. In the first phase, we identify the direct components of types I, III and IV and the reversed components of type VII by going from left to right through string P^+, and then, in the second phase, we compute the reversed components of types V, VI and VIII and the direct components of type III by going from right to left through string P^-. Thus, in both phases, direct as well as reversed components are reported.

To find components in Phase 1 (or Phase 2), we systematically test for each index i from 0 to $l-1$ (or from $l-1$ to 0, respectively), whether there is a component with right bounding element π_i, i.e. an interval of the form $(\pi_s, \ldots, \pi_i)$. Potential positive starting positions for direct components are stored in a stack S_1 and negative starting positions for reversed components in a stack S_2.

Four additional arrays are required for the component identification: Min and Max for direct components, and min and max for reversed components. Concretely, if j is the top of stack S_1, then $Min[j]$ is the minimum and $Max[j]$ is the maximum between j and the current index i. In the same way, the arrays min and max are defined using the stack S_2. Since direct and reversed components are reported in both phases, all four arrays and both stacks S_1 and S_2 are updated in both phases.

The component identification in Phase 1, shown as pseudocode in Algorithm 2, is as follows: First, let us consider direct real components. Additionally to the three conditions described in [6], we require that all elements of a component must be on the same chromosome in B. In line 12 of Algorithm 2, when a left bounding element s is tested, σ_s is always positive because s is the top of the stack S_1. Thus, $(\pi_s, \ldots, \pi_i)$ is a component of type I if and only if:

1. σ_i is positive,
2. all elements between π_s and π_i in π are greater than π_s and smaller than π_i, the latter being equivalent to the simple test $M[i] = M[s]$,
3. $\pi_s \neq 0$ and $Max[s] - Min[s] = i - s$, and
4. all elements from s to i are on the same chromosome in B.

It should be mentioned that the third condition is equivalent, but slightly different to the one given in [6]. For real components, we have that $Max[s] - Min[s] = \pi_i - \pi_s$. The reason for testing $Max[s] - Min[s] = i - s$ instead of testing $\pi_i - \pi_s = i - s$ is that the latter one is not extendable to semi-real components where one or both bounding elements are caps.

Now, we turn from real to semi-real components. Our strategy in Phase 1 is to report semi-real components whose left bounding element is a cap. These are direct components of type III and reversed components of type VII. Semi-real components whose right bounding element is a cap will be reported in Phase 2.

First, let us consider components of type III. Compared to real components, two conditions are modified: Since $\pi_s = 0$, all elements between π_s and π_i are greater than π_s, simplifying the second condition. The third condition is modified in two ways. First, the number of genes in the interval is one less than for real components because the left bounding element is a cap. Second, the minimum of the interval $(s+1, \ldots, i)$, $Min[s]$, must be a gene at the left end of a chromosome in B. Bringing this together, the two conditions are replaced by:

(2) all elements between π_s and π_i in π are smaller than π_i, which is equivalent to the simple test $M[i] = M[s]$,
(3) $\pi_s = 0$ and $Max[s] - Min[s] = i - s - 1$ and $Min[s]$ is a left chromosome end in B.

In line 13 of Algorithm 2, components of type I are reported together with components of type III since their conditions are the same, except for the second and the third condition (see line 12).

The identification of components of type VII is similar to the one of type III. To switch from direct to reversed components, we change the sign of σ_i and replace Max and Min by max and min in the four conditions (see line 19). As a consequence, the following four conditions have to be tested:

1. σ_i is negative,
2. all elements between π_s and π_i in π are greater than π_i, which is equivalent to the simple test $m[i] = m[s]$,
3. $\pi_s = 0$ and $max[s] - min[s] = i - s - 1$ and $max[s]$ is at a right chromosome end in B, and
4. all elements from s to i are on the same chromosome in B.

Finally, at the end of a chromosome, when $\pi_i = N + 1$, semi-real components of type IV whose bounding elements are both caps and have positive sign are identified. As long as we have $\pi_s \neq 0$ for the top element of stack S_1, we remove s (line 30) and update Min and Max (line 31). Then, we test whether the whole chromosome is a semi-real component. In particular, we have that $\pi_s = 0$ and $\pi_i = N + 1$. Obviously, all elements between π_s and π_i are greater than π_s and smaller than π_i, making the second condition meaningless. Since both bounding elements are caps, the number of genes is two less than for real components. Furthermore, there must exist a chromosome in B of the form $(\circ, Min[s], \ldots, Max[s], \circ)$. Altogether, we have to test the following conditions in line 33:

1. σ_{s+1} is positive,
2. (not applicable)
3. $Max[s] - Min[s] = i - s - 2$ and $Min[s]$ is at a left chromosome end in B and $Max[s]$ is at a right chromosome end of B,
4. all elements from s to i are on the same chromosome in B,
5. the component is not a chain of shorter components.

In Phase 2, we apply Algorithm 2 to string P^- instead of P^+. By going backwards through P^-, we report the remaining components in the following order: First, the components of types V and VI are found in the same way as the components of types I and III in Phase 1. Then, by switching from reversed to direct, we identify all components of type II. Finally, at the end of each chromosome, we test whether it is a component of type VIII.

5.2.3. Component Classification

Up to now, Algorithm 2 reports all components of genome A with respect to genome B, and it remains to classify which of them are unoriented and which are not. For the classification of components of types I-III and of types V-VII, it is sufficient to test whether all elements of the component have the same sign. As shown in [6], this can be done by a slight modification of Algorithm 2, without affecting the running time. The classification for the components of types IV and VIII is more delicate and requires the computation of the paths and cycles that belong to the component.

Recall that a component of type IV (or VIII) is unoriented if its elements are positive (respectively negative) and its adjacency graph does not contain an even path. The idea is to mark in an additional boolean array whether an adjacency belongs to an even path or not. This information

Algorithm 2 (Phase 1: Find components of $P^+ = (\pi, \sigma)$)

1: **for** $i \leftarrow 0, \ldots, l$ **do**
2: S_1 and S_2 are stacks of integers; initially S_1 contains i and S_2 contains i
3: $Min[i] \leftarrow N$, $Max[i] \leftarrow 0$, $min[i] \leftarrow N$, $max[i] \leftarrow 0$
4: **while** $\pi[i] \neq N + 1$ **do**

(* Update minima and maxima *)
5: $Min[i] \leftarrow \pi[i]$, Min[top element of S_1] $\leftarrow \min(Min$[top element of S_1], $\pi[i])$
6: $Max[i] \leftarrow \pi[i]$, Max[top element of S_1] $\leftarrow \max(Max$[top element of S_1], $\pi[i])$
7: update similarly min and max using stack S_2

(* Find components of types I and III *)
8: **while** $\pi[s] > \pi[i]$ **or** $M[s] < \pi[i]$ **do**
9: pop the top element s from S_1
10: update Min and Max as in line 5 and 6
11: **end while**
12: **if** $\sigma[i] = +$ **and** $M[i] = M[s]$ **and** (($\pi[s] \neq 0$ **and** $Max[s] - Min[s] = i - s$) **or** ($\pi[s] = 0$ **and** $Max[s] - Min[s] = i - s - 1$ **and** $Min[s]$ is at a left chromosome end in B)) **and** all elements from s to i are on the same chromosome in B **then**
13: report the component $(\pi_s \ldots \pi_i)$
14: **end if**

(* Find components of type VII *)
15: **while** ($\pi[s] < \pi[i]$ or $m[s] > \pi[i]$) **and** $\pi[s] > 0$ **do**
16: pop the top element s from S_2
17: update min and max as in line 7
18: **end while**
19: **if** $\sigma[i] = -$ **and** $m[i] = m[s]$ **and** ($\pi[s] = 0$ **and** $max[s] - min[s] = i - s - 1$ **and** $max[s]$ is at a right chromosome end in B) **and** all elements from s to i are on the same chromosome in B **then**
20: report the component $(\pi_s \ldots \pi_i)$
21: **end if**

(* Update stacks *)
22: **if** $\sigma[i] = +$ **then**
23: push i on S_1
24: **else**
25: push i on S_2
26: **end if**
27: i++
28: **end while**

(* Find components of type IV *)
29: **while** $\pi[s] \neq 0$ **do**
30: pop the top element s from S_1
31: update Min and Max as in lines 5 and 6
32: **end while**
33: **if** $\sigma[s + 1] = +$ **and** $Max[s] - Min[s] = i - s - 2$ **and** $Min[s]$ is at a left chromosome end in B **and** $Max[s]$ is at the right chromosome end of B **and** all elements from s to i are on the same chromosome in B **and** this chromosome is not a chain of shorter components **then**
34: report the component $(\pi_s \ldots \pi_i)$
35: **end if**
36: **end for**

has been obtained in Step 2 of the overall algorithm. If a semi-real component $(\pi_s \dots \pi_i)$ is reported in line 34 of Algorithm 2, we test in constant time whether the two adjacencies $(\circ, \pi_{s+1})$ and $(\pi_{i-1}, \circ)$ belong to an even path or not.

Summarizing, we have that:

Theorem 6. *All components of two linear genomes on the set of genes* $\{1, \dots, N\}$ *can be found and classified as oriented or unoriented with a modified version of Algorithm 2 in* $O(N)$ *time and space.*

5.3. Tree Construction and Distance Computation

Now that we have an algorithm for the component identification, we can describe Steps 4 and 5 of the overall algorithm.

Given the components, we construct the tree T as follows: For each index i, $0 \leq i \leq l$, at most one component can start at position i, and at most one component can end at position i. Hence, it is possible to create a data structure that tells, in constant time, if there is a component beginning or ending at position i and, if so, reports such components. Given this data structure, it is a simple procedure to construct the tree T in one left-to-right scan along the permutation. Initially one round root node is created. Then, for each additional component, a new round node p is created as the child of a new or an existing square node q, depending if p is the first component in a chain or not. For details, see Algorithm 3.

Algorithm 3 (Construct T from the components of A with respect to B)

```
 1: create a round node p, the root of T
 2: for i ← 1, ..., l − 1 do
 3:    if there is a component C starting at position i then
 4:       if there is no component ending at position i then
 5:          create a new square node q as a child of p
 6:       end if
 7:       create a new round node p (representing C) as a child of q
 8:    else if there is a component ending at position i then
 9:       p ← parent of q
10:       q ← parent of p
11:    end if
12: end for
```

Finally, we compute the extra cost. To generate tree T' from tree T, a bottom-up traversal of T recursively removes all dangling round leaves that represent oriented components, and square nodes, including the root if it has degree 1. Given the tree T', it is easy to compute the cover cost: Perform a depth-first traversal of T' and count the number of leaves and the number of long and short branches, including the root if it has degree 1. Then use the formulas from Theorems 4 and 5 to obtain t, and the formula from Theorem 3 to obtain the distance d_{HP}.

Altogether we have:

Theorem 7. *The HP distance* $d_{HP}(A, B)$ *of two genomes A and B on the set* $\{1, \dots, N\}$ *can be computed in linear time* $O(N)$.

6. Conclusion

In this paper, we have given an alternative formula for the Hannenhalli-Pevzner genomic distance equation. It requires only a few parameters that can efficiently be computed from the genomes and from simple graph structures derived from the genomes. Traditionally used concepts that were sometimes hard to access, like weak-fortresses-of-semi-real-knots, are bypassed.

References

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

[2] A. Bergeron. A very elementary presentation of the Hannenhalli-Pevzner theory. In *Proceedings of CPM 2001*, volume 2089 of *LNCS*, pages 106–117. Springer Verlag, 2001.

[3] P. Berman and S. Hannenhalli Fast sorting by reversal. In *CPM 1996 Proceedings*, volume 1075 of *LNCS*, pages 168–185. Springer Verlag, 1996.

[4] A. Bergeron, S. Heber, and J. Stoye. Common intervals and sorting by reversals: A marriage of necessity *Bioinformatics.*, 18 (Suppl. 2): S54-S63, 2002.

[5] A. Bergeron, J. Mixtacki, and J. Stoye. Reversal distance without hurdles and fortresses. In *Proceedings of CPM 2004*, volume 3109 of *LNCS*, pages 388–399. Springer Verlag, 2004.

[6] A. Bergeron, J. Mixtacki, and J. Stoye. The inversion distance problem. In O. Gascuel, editor, *Mathematics of Evolution and Phylogeny*, chapter 10, pages 262–290. Oxford University Press, Oxford, UK, 2005.

[7] A. Bergeron, J. Mixtacki, and J. Stoye. On sorting by translocations. *J. Comput. Biol.*, 13(2):567–578, 2006.

[8] A. Bergeron, J. Mixtacki, and J. Stoye. A unifying view of genome rearrangements. In *Proceedings of WABI 2006*, volume 4175 of *LNBI*, pages 163–173. Springer Verlag, 2006.

[9] A. Bergeron and J. Stoye. On the similarity of sets of permutations and its applications to genome comparison. In T. Warnow and B. Zhu, editors, *Proceedings of COCOON 2003*, volume 2697 of *LNCS*, pages 68–79. Springer Verlag, 2003.

[10] T. Dobzhansky and A. T. Sturtevant. Inversions in the Chromosomes of *Drosophila pseudoobscura*. *Genetics*, 23:28–64, 1938.

[11] W. M. Fitch. Homology: a personal view on some of the problems. *Trends in Genetics*,16(5) 227–231, 2000.

[12] S. Hannenhalli and P. A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of FOCS 1995*, pages 581–592. IEEE Press, 1995.

[13] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, 1999.

[14] G. Jean and M. Nikolski. Genome rearrangements: A correct algorithm for optimal capping. *Inf. Process. Lett.*, 104:14–20, 2007.

[15] H. Kaplan, R. Shamir, and R. E. Tarjan. A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM J. Computing*, 29(3):880–892, 1999.

[16] M. Ozery-Flato and R. Shamir. Two notes on genome rearrangements. *J. Bioinf. Comput. Biol.*, 1(1):71–94, 2003.

[17] G. Tesler. Efficient algorithms for multichromosomal genome rearrangements. *J. Comput. Syst. Sci.*, 65(3):587–609, 2002.

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

[19] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.