10

THE INVERSION DISTANCE PROBLEM

Anne Bergeron, Julia Mixtacki, and Jens Stoye

Among the many genome rearrangement operations, signed inversions stand out for many biological and computational reasons. Inversions, also known as reversals, are widely identified as one of the common rearrangement operations on chromosomes, they are basic to the understanding of more complex operations such as translocations, and they offer many computational challenges. From the first formulation of the inversion distance problem, ca. 1992, to its first polynomial solution in 1995, to the several simplifications of the solution in recent years, there is not yet a simple, complete, and elementary treatment of the subject. This is the goal of this chapter.

10.1 Introduction and biological background

In the last 10 years, beginning with Sankoff [20], many papers have been devoted to the subject of computing the *inversion distance* between two permutations. An *inversion* of an interval from p_i to p_j transforms a permutation P into P':

$$\begin{array}{rl} P = & (p_1 \quad \cdots \quad \underline{p_i \quad p_{i+1} \cdots \quad p_j} \quad \cdots \quad p_n), \\ P' = & (p_1 \quad \cdots \quad p_j \quad \cdots \quad p_{i+1} \; p_i \quad \cdots \quad p_n). \end{array}$$

The inversion distance between two permutations is the minimum number of inversions that transform one into the other.

From a problem of unknown complexity, it eventually graduated to an NP-hard problem [9], but an interesting variant was proven to be polynomial [12]. In the *signed* version of the problem, each element of the permutation has a plus or minus sign, and an inversion of an interval from p_i to p_j transforms P to P':

$$\begin{array}{rl} P = & (p_1 \quad \cdots \quad \underline{p_i \quad p_{i+1} \cdots \quad p_j} \quad \cdots \quad p_n), \\ P' = & (p_1 \quad \cdots \quad -p_j \quad \cdots \quad -p_{i+1} \; -p_i \quad \cdots \quad p_n). \end{array}$$

Permutations, and their inversions, are useful tools in the comparative study of genomes. The genome of a species can be thought of as a set of ordered

sequences of genes—the *chromosomes*—each gene having an orientation given by its location on the DNA double strand. Different species often share similar genes that were inherited from common ancestors. However, these genes have been shuffled by evolutionary events that modified the content of chromosomes, the order of genes within a particular chromosome, and/or the orientation of a gene. Assigning the same index to similar genes appearing along a chromosome in two different species, and using negative signs to model changes in orientation, yields two signed permutations. The inversion distance between these permutations can thus be used to compare species.

Computing the inversion distance of signed permutations is a delicate task since some inversions unexpectedly affect deep structures in permutations. In 1995, Hannenhalli and Pevzner proposed the first polynomial algorithm to solve it [12], developing along the way a theory of how and why some permutations were particularly resistant to sorting by inversions. It is of no surprise that the label *fortress* was assigned to specially acute cases.

Hannenhalli and Pevzner relied on several intermediate constructions that have been subsequently simplified [7, 13], but grasping all the details remained a challenge. Before Bergeron [3], all the criteria given for choosing a *safe* inversion involved the construction of an associated permutation on $2n$ points, and the analysis of cycles and/or connected component of the graph associated with this permutation.

Moreover, most papers tended to mix two different problems, as pointed out in references [1, 13]: the computation of the *number* of necessary inversions, and the reconstruction of one possible sequence of inversions that realizes this number. The first problem was finally proved to be of linear time complexity [1], but this approach still used many of the Hannenhalli–Pevzner constructions. However, the existence of a linear-time solution was a strong incentive to try to present the computation in an elementary way, which led to the recognition of the central role played by *subpermutations* in the theory [4, 6, 11].

In this chapter, we present an elementary treatment of the sorting by inversions problem. We give a complete proof of the Hannenhalli–Pevzner *duality theorem* in terms of the elements of a given signed permutation, efficient, and simple algorithms to compute the inversion distance, and simple procedures for the construction of optimal inversion sequences.

In the next section, we introduce the basic definitions and describe the sorting by inversions problem. In Section 10.3 we introduce several concepts, such as cycles and components, which are central to the solution of this problem. The relations between components are used to construct a tree associated to a signed permutation. This tree is the basis of a simple proof of the Hannenhalli–Pevzner *duality theorem* presented in Section 10.4. Finally, in Section 10.5 we present algorithms to identify the components, to count the number of cycles, and to construct the tree associated to a signed permutation.

The last section contains a glossary of the terminology used in this chapter.

10.2 Definitions and examples

A *signed permutation* is a permutation on the set of integers $\{0, 1, 2, \ldots, n\}$ in which each element has a sign, positive or negative. For convenience,[1] we will assume that all permutations begin with 0 and end with n. For example:

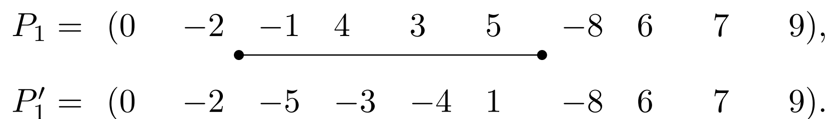
$$P_1 = \begin{pmatrix} 0 & -2 & -1 & 4 & 3 & 5 & -8 & 6 & 7 & 9 \end{pmatrix}.$$

Since integers represent genes and signs represent the orientation of a gene on a particular chromosome, we will refer to the underlying gene as an *unsigned element* of the permutation.

A *point* $p \cdot q$ is defined by a pair of consecutive elements in the permutation. For example, $0 \cdot -2$ and $-2 \cdot -1$ are the first two points of P_1. When a point is of the form $i \cdot i+1$, or $-(i+1) \cdot -i$, it is called an *adjacency*, otherwise it is called a *breakpoint*. For example, P_1 has two adjacencies, $-2 \cdot -1$ and $6 \cdot 7$. All other points of P_1 are breakpoints.

We will make an extensive use of intervals of consecutive elements in a permutation. An interval is easily defined by giving its *endpoints*. The *elements* of the interval are the elements between the two endpoints. When the two endpoints are equal, the interval contains no elements. A non-empty interval can also be specified by giving its first and last element, such as $(i \ldots j)$, called the *bounding elements* of the interval.

An *inversion* of an interval of a signed permutation is the operation that consists of inverting the order of the elements of the interval, while changing their signs. For example, the inversion of the interval of P_1 whose endpoints are $-2 \cdot -1$ and $5 \cdot -8$ yields the permutation P_1':



The inversion of an interval modifies the points of a signed permutation in various ways. Points $p \cdot q$ that are inside the interval are transformed to $-q \cdot -p$, the endpoints of the interval exchange their flanking elements, and points that are outside the interval are unaffected.

The *inversion distance* $d(P)$ of a permutation P is the minimum number of inversions needed to transform P into the identity permutation. Finding one sequence of inversions that realizes this distance is called the *sorting by inversions problem*. For example, $d(P_1) = 5$, and Fig. 10.1 shows a sequence of inversions that realizes this distance.

A sequence of inversions, applied to a permutation P, is called an *optimal sorting sequence* if it transforms P into the identity permutation, and if its length

[1]This assumption simplifies the theory and is coherent with biological applications in which whole chromosomes do not have a global orientation: only local changes of orientation are relevant.

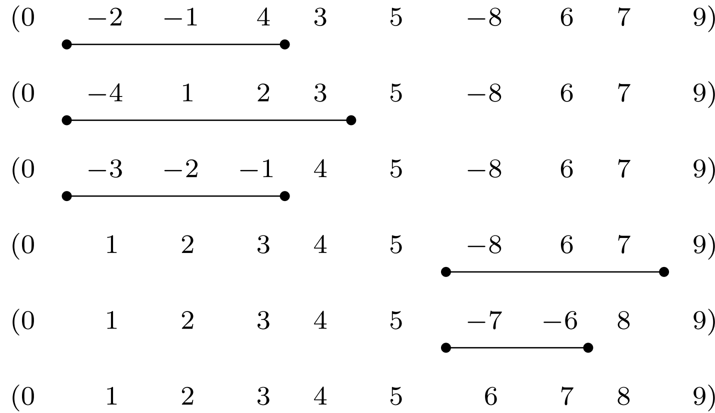


FIG. 10.1. Sorting $P_1 = (0\ {-2}\ {-1}\ 4\ 3\ 5\ {-8}\ 6\ 7\ 9)$ by inversions.

$P =$	$(0\ {-2}\ {-1}\ 4\ 3\ 5\ {-8}\ 6\ 7\ 9)$	$Q^{-1} \circ P =$	$(0\ 1\ {-3}\ {-7}\ {-6}\ 2\ {-8}\ 4\ 5\ 9)$
	$(0\ {-2}\ {-1}\ 4\ 3\ 5\ {-7}\ {-6}\ 8\ 9)$		$(0\ 1\ {-3}\ {-7}\ {-6}\ 2\ {-5}\ {-4}\ 8\ 9)$
	$(0\ {-2}\ {-1}\ {-5}\ {-3}\ {-4}\ {-7}\ {-6}\ 8\ 9)$		$(0\ 1\ {-3}\ {-2}\ 6\ 7\ {-5}\ {-4}\ 8\ 9)$
	$(0\ {-2}\ {-1}\ {-5}\ 6\ 7\ 4\ 3\ 8\ 9)$		$(0\ 1\ {-3}\ {-2}\ 4\ 5\ {-7}\ {-6}\ 8\ 9)$
	$(0\ {-2}\ 5\ 1\ 6\ 7\ 4\ 3\ 8\ 9)$		$(0\ 1\ 2\ 3\ 4\ 5\ {-7}\ {-6}\ 8\ 9)$
$Q =$	$(0\ {-2}\ 5\ 1\ 6\ 7\ {-3}\ {-4}\ 8\ 9)$	$Q^{-1} \circ Q =$	$(0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$

FIG. 10.2. Transforming permutation $P_1 = (0\ {-2}\ {-1}\ 4\ 3\ 5\ {-8}\ 6\ 7\ 9)$ into permutation $Q = (0\ {-2}\ 5\ 1\ 6\ 7\ {-3}\ {-4}\ 8\ 9)$ is simulated by transforming permutation $Q^{-1} \circ P_1$ into $Q^{-1} \circ Q$, where $Q^{-1} = (0\ 3\ {-1}\ {-6}\ {-7}\ 2\ 4\ 5\ 8\ 9)$.

is $d(P)$. An inversion that belongs to an optimal sorting sequence is called a *sorting inversion.*

In general, the *inversion distance* between two arbitrary permutations P and Q is the minimum number of inversions that transform one into the other. One can always reduce this problem to a problem of inversion distance to the identity permutation by composing[2] the permutations P and Q with the inverse permutation of one of them, say Q^{-1}. Any sequence of inversions that transforms $Q^{-1} \circ P$ into $Q^{-1} \circ Q$ can be applied to the original problem. An example is given in Fig. 10.2.

Historical notes. Surprisingly, inversions of segments of chromosomes have been identified in close species by Sturtevant [23] early in the last century. It then took decades of biological experiments to accumulate sufficient data to compare gene order of a vast array of species. For simple chromosomes, such as mitochondria, the sequence of genes is now known for several hundred species. See Chapter 9, this volume, for more details.

[2]Here, composition is understood as the standard composition of functions. Dealing with signed permutations requires the additional axiom that $P(-a) = -P(a)$.

In 1982, Watterson *et al.* [26] first formulated the problem of finding the minimum number of inversions required to bring one configuration of genes into another. It took more than 10 years until Kececioglu and Sankoff [14] developed the first approximation algorithm for the problem of sorting an unsigned permutation by inversions. They also conjectured that this problem is NP-hard. Indeed, this was shown in 1997 by Caprara [9]. Bafna and Pevzner [2] initiated the study of signed permutations in order to model the orientation of genes. In 1995, Hannenhalli and Pevzner [12] gave the first polynomial–time algorithm for the problem of sorting a signed permutation by inversions using the concepts developed by Bafna and Pevzner. A clear distinction between the problem of computing the inversion distance and finding an optimal sorting sequence was worked out by Kaplan *et al.* [13] and Bader *et al.* [1]. Currently, the most efficient algorithms to solve the inversion distance problem are linear, while the most efficient algorithms to find optimal sorting sequences are not [19, 24].

Since many optimal sorting sequences exist, recently Siepel [22] studied the problem of finding all optimal sequences and gave a polynomial–time algorithm to find all sorting inversions of a permutation.

10.3 Anatomy of a signed permutation

In the following, we define several concepts central to the analysis of signed permutations, and study the effect of inversions on these structures. First, we consider the elementary intervals and cycles in Sections 10.3.1 and 10.3.2, and then we treat the components of a permutation in Sections 10.3.3 and 10.3.4.

10.3.1 *Elementary intervals and cycles*

Let P be a signed permutation on the set $\{0, 1, 2, \ldots, n\}$ that begins with 0 and ends with n. Any element i of P, $0 < i < n$, has a *right* and a *left* point.

Definition 10.1 *For each pair of unsigned elements* $(k, k+1)$, $0 \leq k < n$, *define the* elementary interval I_k *associated to the pair to be the interval whose endpoints are:*

1. *The right point of* k, *if* k *is positive, otherwise its left point.*
2. *The left point of* $k+1$, *if* $k+1$ *is positive, otherwise its right point.*

Elements k *and* $k+1$ *are called the* extremities *of the elementary interval.*

An elementary interval can contain zero, one, or both of its extremities. For example, in Fig. 10.3, interval I_0 contains one of its extremities, interval I_3 contains both, and interval I_5 contains none. Empty elementary intervals, such as I_1 and I_6, correspond to adjacencies in the permutation.

When the extremities of an elementary interval have different signs, the interval is said to be *oriented*, otherwise it is *unoriented*. Oriented intervals are exactly those intervals that contain one of their extremities.

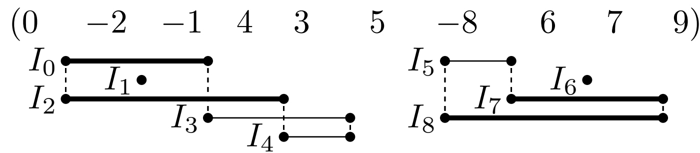


FIG. 10.3. Elementary intervals and cycles of a permutation. Oriented intervals are represented by thick lines, and unoriented intervals by thin lines. Vertical dashed lines join intervals that meet at breakpoints, tracing the cycles.

Oriented intervals play a crucial role in the problem of sorting by inversions since they can be used to create adjacencies. Namely, we have:

Proposition 10.2 *Inverting an oriented interval I_k creates, in the resulting permutation, either the adjacency $k \cdot k+1$ or the adjacency $-(k+1) \cdot -k$.*

Proof Suppose that k is positive, then $k+1$ must be negative for the interval I_k to be oriented. If $k+1$ succeeds k, then the interval will contain $k+1$ but not k, and inverting it will create the adjacency $k \cdot k+1$. If $k+1$ precedes k, then the interval will contain k but not $k+1$, and inverting it will create the adjacency $-(k+1) \cdot -k$. The case when k is negative is treated similarly. □

For example, inverting the oriented elementary interval I_8 in permutation P_1 of Fig. 10.3 creates the adjacency $8 \cdot 9$.

When a point is the endpoint of two elementary intervals, these are said to *meet* at that point.

Proposition 10.3 *Exactly two elementary intervals meet at each breakpoint of a permutation.*

Proof From Definition 10.1, the right and left point of each element of the permutation is used once as an endpoint of an elementary interval, thus each breakpoint is used twice. □

Therefore, by Proposition 10.3, starting from an arbitrary breakpoint, one can follow elementary intervals on a unique path that eventually comes back to the original breakpoint. More formally:

Definition 10.4 *A* cycle *is a sequence $b_1, b_2, \ldots, b_k$ of points such that two successive points are the endpoints of an elementary interval, including b_k and b_1. Adjacencies define* trivial *cycles consisting of a single point.*

For example, as shown in Fig. 10.3, permutation P_1 has four cycles, two of them are trivial, and the other two contain, respectively, 4 and 3 breakpoints.

Cycles are conveniently defined with breakpoints, but one can always focus on the elementary intervals that are defined by the breakpoints of a cycle. The following property, on the number of oriented intervals of a cycle, will be useful to prove results on the number of cycles of a permutation.

Lemma 10.5 *A cycle always contains an even number of oriented intervals.*

Proof Let J_i be the interval that connects b_i to the next breakpoint in a cycle $b_1, b_2, \ldots, b_k$. Define e_i to be the number of extremities of J_i contained in it, either 0, 1, or 2, and consider the sum: $E = \sum_{i=1}^{k} e_i$. We will show that E is an even number, implying that the number of oriented intervals is even.

The idea is to construct the sum E by considering the contribution of each breakpoint of the cycle. Follow the breakpoints in the order $b_1, b_2, \ldots, b_k$. A given breakpoint can either join two disjoint intervals, or two stacked intervals. In this last case, the breakpoint is a *turning* point of the cycle. Each turning point $p \cdot q$ contributes 1 to the number E, since either p or q is inside both intervals, and the other is outside both intervals. Each breakpoint $p \cdot q$ that joins two disjoint intervals contributes 0 or 2 to the number E, since p is inside its interval if and only if q is. However, the number of turning points of a cycle must be even, therefore E is even. □

A last fundamental relation between elementary intervals is the *overlap* relation.

Definition 10.6 *Two elementary intervals I and J* overlap *if each contains exactly one of the extremities of the other.*

The overlap relation is often easily detectable, like the overlap of the intervals I_2 and I_1 in Fig. 10.4. Intervals that meet at a breakpoint can overlap or not. For example, intervals I_0 and I_2 overlap since I_0 contains element -3, and I_2 contains element 1; on the other hand intervals I_0 and I_3 do not overlap, despite the fact that they meet at breakpoint $0 \cdot 4$.

A common way to represent the overlap relation between elementary intervals is the *overlap graph* $\mathcal{O}$ with black and white vertices standing, respectively, for oriented and unoriented elementary intervals. Two vertices are connected

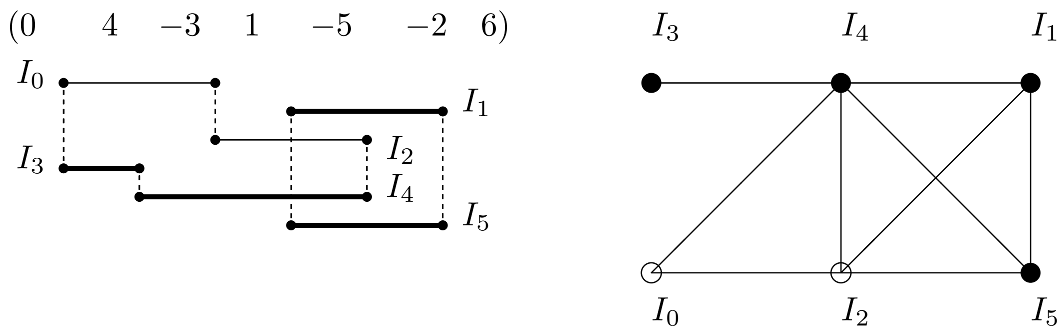


FIG. 10.4. A permutation and its overlap graph $\mathcal{O}$. Only two elementary intervals are unoriented, I_0 and I_2, corresponding to white vertices of the graph $\mathcal{O}$. Intervals I_0 and I_2 overlap since I_0 contains element -3, and I_2 contains element 1; on the other hand intervals I_0 and I_3 do not overlap, despite the fact that they meet at breakpoint $0 \cdot 4$.

in $\mathcal{O}$ if and only if the corresponding intervals overlap. The right hand side of Fig. 10.4 gives an example of such a graph.

10.3.2 *Effects of an inversion on elementary intervals and cycles*

One of the cornerstones of the sorting by inversions problem is to study the effects of an inversion on elementary intervals and cycles. The first result, due to reference [15], is the effect of an inversion on the number of cycles. It is based on the fact that, for all points except the endpoints of an inversion, the elementary intervals that meet at those points will still meet at that point after the inversion.

Proposition 10.7 *An inversion can only modify the number of cycles by* $+1$, 0, *or* -1.

Proof An inversion exchanges the elements of two points of a permutation. If these two points belong to the same cycle, then either the cycle is split in two, or is conserved but with different breakpoints. If the two points belong to different cycles, then these cycles are merged. Figure 10.5 gives an illustration of the three cases. □

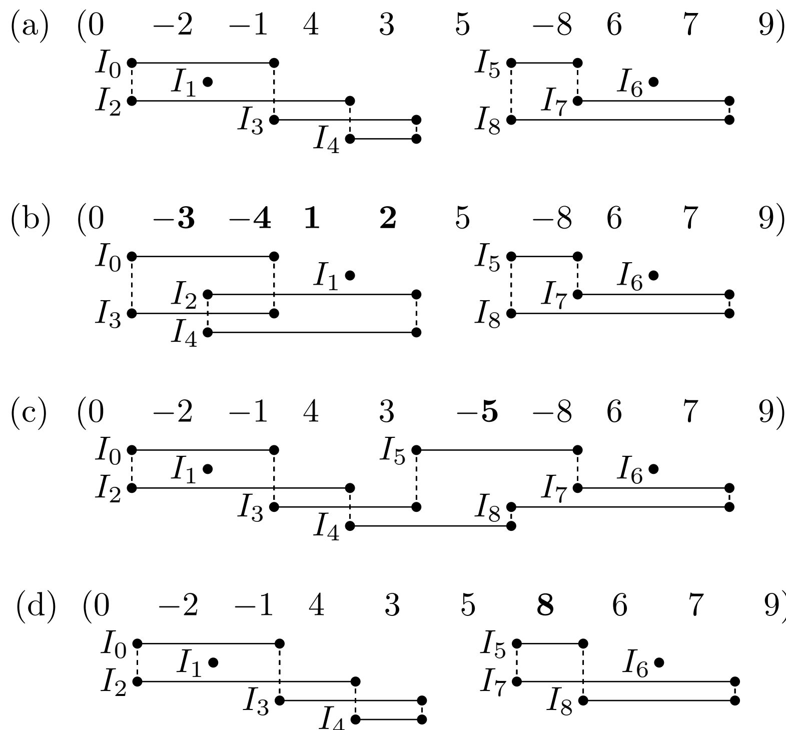


FIG. 10.5. Effects of inversions on cycles. The original permutation, again P_1, is shown in (a). In (b), the inversion of interval $(-2, -1, 4, 3)$ splits the cycle of length 4 of the original permutation. In (c), the inversion of element 5 merges the two long cycles of the original permutation. Finally, in (d), the inversion of element 8 leaves the number of cycles unchanged.

By Propositions 10.2 and 10.7, inverting an oriented interval always splits a cycle, since an adjacency is a trivial cycle. The identity permutation on the set $\{0, 1, 2, \dots, n\}$ is the only one with n cycles, all adjacencies. Since at most one cycle can be added by an inversion, Proposition 10.7 implies a first lower bound to the inversion distance of a permutation:

Lemma 10.8 *Let c be the number of cycles of a signed permutation P on the set $\{0, 1, 2, \dots, n\}$. Then $d(P) \geq n - c$.*

The next important observation is an easy consequence of the overlap relation. If I and J overlap, then inverting the interval I will change the orientation of J, since only one extremity of J will change sign.

When two intervals J and K overlap an interval I, the effect of inverting I complements the overlap relation between J and K: if J and K overlapped before the inversion, they do not overlap after it; if J and K did not overlap before the inversion, they overlap after it.

Formally, we have:

Proposition 10.9 *Let G_I be the subgraph of the overlap graph formed by vertex I and its adjacent vertices. Consider the inversion of elementary interval I.*

1. *If I is unoriented, the effect on the overlap graph is to change the colour of all vertices in $G_I - \{I\}$, and complement the edges of $G_I - \{I\}$.*
2. *If I is oriented, the effect on the overlap graph is to change the colour of all vertices in G_I, and complement the edges of G_I.*

Proof 1. If the elementary interval I is unoriented, either both or none of the extremities of I are contained in the interval I, thus inverting the interval I does not change the orientation of the vertex I. Let vertex J be adjacent to I, then I contains exactly one of the extremities of J, and inverting the interval I changes the sign of one extremity of J. Thus, J changes orientation. If vertices J and K are adjacent to I, then one extremity of J and one of K are contained in I. If J and K are overlapping, then inverting the elementary interval I will invert the order of the extremities of J and K that are contained in I. The elementary intervals J and K will either be disjoint, or one will be contained in the other. Thus, they are not overlapping in the resulting permutation. A similar argument shows that if J and K are not overlapping, then they will overlap after the inversion.

2. Inverting the oriented elementary interval I creates the isolated vertex I, since it creates an adjacency by Proposition 10.2. Thus each edge incident to I is erased. The complementation of the edges and the orientation of $G_I - \{I\}$ is similar to the unoriented case. □

10.3.3 *Components*

Elementary intervals and cycles are organized in higher structures called *components*. These were first identified in reference [11] as *subpermutations* since

$$P_2 = (0\ -3\ \ 1\ \ 2\ \ 4\ \ 6\ \ 5\ \ 7\ -15\ -13\ -14\ -12\ -10\ -11\ -9\ \ 8\ \ 16).$$

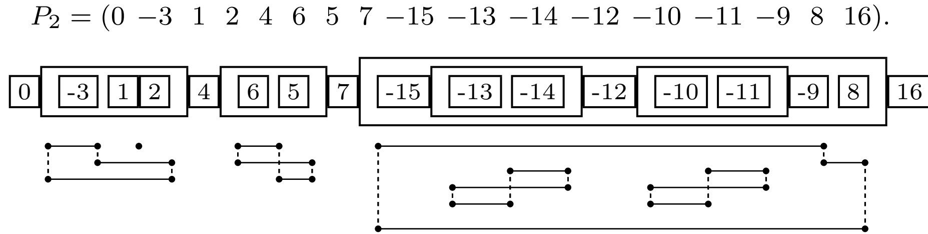


FIG. 10.6. A permutation and the boxed representation of its components. Endpoints of elementary intervals, and thus cycles, belong to exactly one component.

they are intervals that contain a permutation of a set of consecutive integers, and later studied in more detail in reference [4] as *framed common intervals.*

Definition 10.10 *Let P be a signed permutation on the set $\{0, 1, 2, \ldots, n\}$. A* component *of P is 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. Components with positive, respectively negative, bounding elements are referred to as* direct*, respectively* reversed*, components.*

For example, consider the permutation P_2 of Fig. 10.6. It has six components: four of them are direct, $(0 \ldots 4)$, $(4 \ldots 7)$, $(7 \ldots 16)$, and $(1 \ldots 2)$; and two of them are reversed, $(-15 \ldots -12)$ and $(-12 \ldots -9)$. Note that a component, such as the adjacency $1 \cdot 2$, can contain only two elements.

Components of a permutation can be represented by a boxed diagram, such as in Fig. 10.6, in which bounding elements of each component have been boxed, and elements between them are enclosed in a rectangle. Elements which are not bounding elements of any component are also boxed.

Components organize hierarchically the points, elementary intervals, and cycles of a permutation.

Definition 10.11 *A point $p \cdot q$* belongs *to the smallest component that contains both p and q.*

Note that this does not prevent the elements p and q to belong, separately, to other components, such as point $7 \cdot -15$ in the permutation of Fig. 10.6.

Proposition 10.12 *The endpoints of an elementary interval belong to the same component, thus all the points of a cycle belong to the same component.*

Proof Consider an elementary interval I_k and any component C of the form

$$(i \ldots i + j) \quad \text{or} \quad (-(i + j) \ldots -i),$$

such that $i \leq k < i + j$. We will show that both endpoints of I_k are contained in C. This is obvious if k is different from i and $k + 1$ is different from $i + j$, since both k and $k + 1$ will be in the interior of the component. If $k = i$, then k and i have the same sign, and the first endpoint of I_k belongs to the component.

If $k+1 = i+j$, then $k+1$ and $i+j$ have the same sign, and the second endpoint of I_k belongs to the component.

Thus endpoints of I_k are either both contained, or not, in any given component, and the result follows. □

A component can have more than one cycle. For example, the permutation of Fig. 10.4 has one component $(0 \dots 6)$ consisting of two cycles. Finally, components can be classified according to the nature of the points they contain:

Definition 10.13 *The* sign *of a point $p \cdot q$ is positive if both p and q are positive, it is negative if both p and q are negative. A component is* unoriented *if it has one or more breakpoints, and all of them have the same sign, otherwise the component is* oriented.

For example, the unoriented components of the permutation of Fig. 10.6 are $(4 \dots 7)$, $(-15 \dots -12)$, and $(-12 \dots -9)$. All the elementary intervals whose endpoints belong to the same unoriented component are unoriented intervals. Therefore, it is impossible to create an adjacency in an unoriented component with only one inversion. On the other hand, an oriented component contains at least one oriented interval, thus at least two, by Lemma 10.5 and Proposition 10.12.

In order to optimally solve the sorting problem, it is necessary to understand the relationship between the components of a permutation. The following definitions and propositions establish these relationships.

Proposition 10.14 ([6]) *Two different components of a permutation are either disjoint, nested with different endpoints, or overlapping on one element.*

Proof First note that two components that share an endpoint must be both direct or both reversed.

Consider two direct components C and C' of the form

$$C = (i \dots i+j) \quad \text{and} \quad C' = (i' \dots i'+j').$$

Suppose the components C and C' are nested with $i = i'$ and $j' < j$. Since C' is a component, it contains all unsigned elements between its bounding elements i' and $i'+j'$, and hence the interval $(i'+j' \dots i+j)$ contains all unsigned elements between $i'+j'$ and $i+j$. This contradicts the fact that the component C is not the union of two shorter components. The case where the components C and C' are reversed can be treated similarly.

Suppose that the components $C = (i \dots i+j)$ and $C' = (i' \dots i'+j')$ are direct and overlap with more than one element. We can assume that

$$i < i' < i+j < i'+j'.$$

Since all unsigned elements between i' and $i'+j'$ are greater than i', the interval $(i \dots i')$ must contain all unsigned elements between i and i'. Thus, C is the

union of two shorter components, which leads to a contradiction. Again, the reverse case follows by a similar argument. □

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*. Note that a maximal chain may consist of a single component. If one component of a chain is nested in a component A, then all other components of the chain are also nested in A.

The nesting and linking relations between components turn out to play a major role in the sorting by inversions problem. Another way of representing these relations is by using the following tree:

Definition 10.15 *Given a permutation P on the set $\{0, 1, \ldots, n\}$ and its components, define the tree T_P by the following construction:*

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

For example, Fig. 10.7 represents the tree T_{P_2} associated to permutation P_2 of Fig. 10.6.

It is easy to see that, if the permutation begins with 0 and ends with n, the resulting graph is a single tree with a square node as root. The tree is similar to the PQ-tree used in different context such as the consecutive ones test [8]. The following properties of paths in T_P are elementary consequences of the definition of T_P.

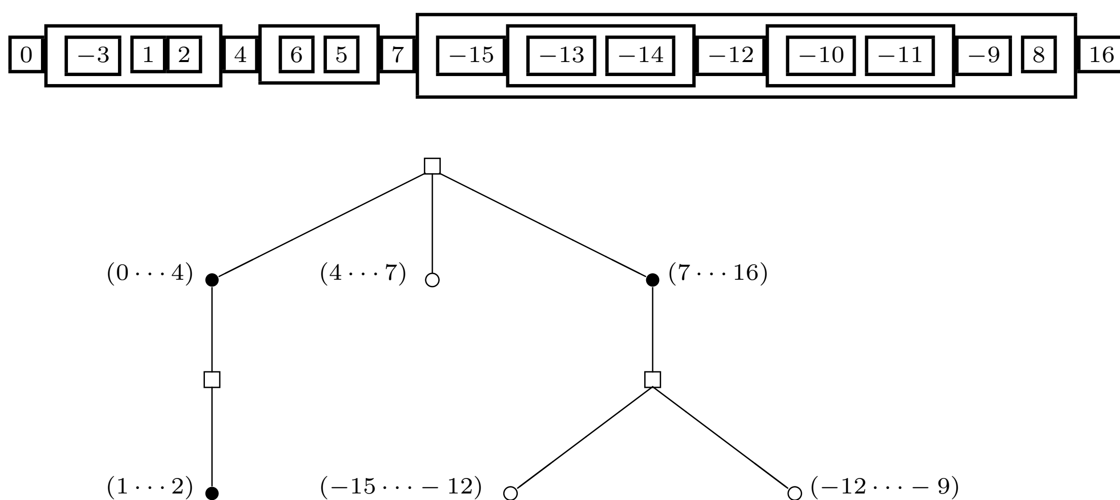


FIG. 10.7. The tree T_{P_2} associated to permutation P_2 of Fig. 10.6. White round nodes correspond to unoriented components, and black round nodes correspond to oriented components.

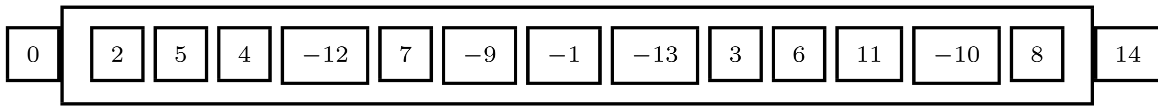
Proposition 10.16 *Let C be a component on the (unique) path joining components A and B in T_P, then C contains either A or B, or both.*

1. *If C contains both A and B, it is unique.*
2. *No component of the path contains both A and B if and only if A and B are included in two components that are in the same chain.*

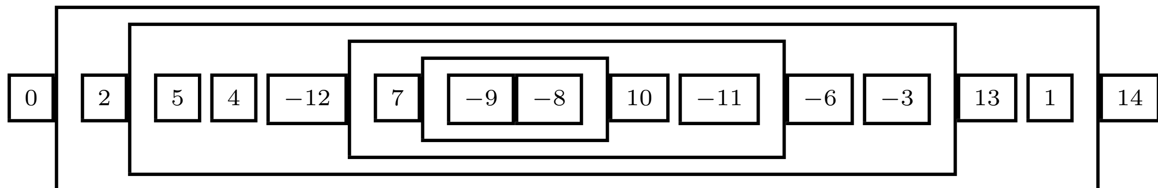
Proof Consider the smallest component D that contains components A and B. If it is on the path that joins A and B, then any other component that contains A and B is an ancestor of D, therefore not on the path. If D is not on the path that joins A and B, then the least common ancestor of components A and B is a square node q that is a child of the round node representing D, thus A and B are included in two components that are in the chain represented by q. □

10.3.4 *Effects of an inversion on components*

We saw, in Proposition 10.7, that an inversion can modify the number of cycles of a permutation by at most 1. On the other hand, an inversion can create or destroy any number of components. For example, inverting the interval $(-1, \ldots, 8)$ in the following permutation



creates the adjacency $-9 \cdot -8$ and yields a permutation with four new components:



As we will see in the next section, creating oriented components, or adjacencies, is generally considered a good move towards optimally sorting a permutation. However, the creation of unoriented components should be avoided. Luckily, few inversions have that effect.

The next three propositions describe the effects of inversions whose endpoints are in unoriented components. These are classical results from the Hannenhalli–Pevzner theory.

Proposition 10.17 *If a component C is unoriented, no inversion with its two endpoints in C can split one of its cycles, nor create a new component.*

Proof First note that Lemma 10.5 implies that the number of positive—or negative—extremities of intervals of a cycle must be even, since each oriented interval has a positive and a negative extremity.

If a component C is unoriented, then all the breakpoints of its cycles have the same sign. An inversion with its two endpoints in one of the cycles of C will introduce exactly two new breakpoints which are neither positive nor negative. If a cycle of C is split, those two breakpoints must belong to different cycles c_1 and c_2. In each of these cycles, the remaining breakpoints are either positive or negative. Thus, the number of positive extremities of the intervals of c_1 and of c_2 would be odd numbers.

Suppose an inversion creates a new component D, then one bounding element of D has to be inside the inverted interval, and the other one outside the inverted interval, otherwise the component D would have existed before the inversion. Therefore, the bounding elements of the component D have different sign, which contradicts the definition of a component. □

Proposition 10.18 *If a component C is unoriented, the inversion of an elementary interval whose endpoints belong to C orients C, and leaves the number of cycles of the permutation unchanged.*

Proof Inverting an elementary interval changes the sign of the elements of the inverted interval. Therefore, component C will be oriented. Since the endpoints of an elementary interval belong to the same cycle, the inversion cannot merge cycles. By Proposition 10.17, the inversion of I cannot split a cycle. Therefore, the number of cycles remains unchanged. □

Orienting a component as in Proposition 10.18 is called *cutting* the component. Such an inversion is seldom a sorting inversion since it is possible, with a single inversion, to get rid of more than one unoriented component. The following proposition describes how to *merge* several components, and the relation of this operation to paths in T_P.

Proposition 10.19 *An inversion that has its two endpoints in different components A and B destroys, or orients, all components on the path from A to B in T_P, without creating new unoriented components.*

Proof Note first that an inversion with endpoints in different components A and B must merge two cycles, one from each component, into a new cycle c. If A and B are unoriented, cycle c contains at least one oriented interval.

Suppose that a new component D is created by such an inversion, then the bounding elements of D must be both outside the inverted interval. Indeed, if both bounding elements of D are inside the inverted interval, D existed in the original permutation. If one bounding element of D is outside the interval, then component D must contain at least one endpoint of the inverted interval in order to be affected by the inversion. Since the two endpoints of the inverted interval belong to the same cycle c, the second endpoint of the interval must also be in component D, thus the second bounding element of D is also outside the interval.

Thus, the only component eventually created by an inversion with endpoints in different components is the union of two or more linked components. Since

linked components have bounding elements with the same sign, the sign of the former links will be different from the sign of the bounding elements of the new component, thus it will be oriented.

By Proposition 10.16, if there is a component C on the path from A to B and that contains both, then A and B are not included in linked components, thus no new component can be created by the inversion. Since C is the smallest component that contains the new cycle c, C will be oriented.

Finally, suppose that a component C is on the path from A to B and contains either A or B, but not both. Then the inversion changes the sign of one of the bounding elements of C, and C will be destroyed. □

Proposition 10.19 thus states that one can get rid of many unoriented components with only one inversion. This idea is exploited in the next section to compute the inversion distance of a permutation.

Historical notes. In 1984, Nadeau and Taylor [18] introduced the notion of breakpoints of a permutation. One decade later, Kececioglu and Sankoff [14] brought in the *breakpoint graph* in their analysis of the sorting by inversions problem. Later, Bafna and Pevzner [2] extended the breakpoint graph to signed permutations.

The most common version of the breakpoint graph[3] is based on an unsigned permutation of $2n$ elements defined as follows: replace any positive element x of a signed permutation by $2x-1, 2x$ and any negative element $-x$ by $2x, 2x-1$. The breakpoint graph is an edge-coloured graph whose set of vertices are the elements $(p_0, \ldots, p_{2n-1})$ of this unsigned permutation.

For each $0 \leq i < n$, vertices p_{2i} and p_{2i+1} are joined by a black edge, and elements $2i$ and $2i+1$ of the permutation are joined by a grey edge. Thus, each vertex of the breakpoint graph has exactly two incident edges. This allows the unique decomposition of the breakpoint graph into *cycles*. The *support* of a grey edge is the interval of elements between and including the endpoints. Two grey edges *overlap* if their supports intersect without proper containment. The *overlap graph* is the graph whose vertices are the grey edges of the breakpoint graph and whose edges join overlapping grey edges.

In the traditional analysis of the sorting by inversions problem, the cycles of the breakpoint graph, and the connected components of the overlap graph, play an important role. The elementary intervals, cycles and overlap graph of this section are equivalent to the traditional concepts, but directly defined on the elements of the permutation. The components of Definition 10.10 correspond to the connected components of the overlap graph.

It is also worth mentioning that Setubal and Meidanis [21] obtained many combinatorial results on the effects of inversions on a permutation, generalizing results such as Proposition 10.17.

[3]For a more detailed presentation of the breakpoint graph, see Chapter 11, this volume.

10.4 The Hannenhalli–Pevzner duality theorem

In this section, we develop a formula for computing the inversion distance of a signed permutation. There are two basically different problems: the contribution of oriented components to the total distance is treated in Section 10.4.1, and the general formula is given in Section 10.4.2.

10.4.1 *Sorting oriented components*

We will show that sorting oriented components can be done by choosing oriented inversions that do not create new unoriented components. For example, the inversion of the oriented interval I_3 in the following permutation creates a new unoriented component (0 2 1 3). In the resulting positive permutation, no inversion can create an adjacency, or split a cycle.

$$(0 \quad 2 \quad -3 \quad -1 \quad 4),$$

$$(0 \quad 2 \quad 1 \quad 3 \quad 4).$$

However, one can invert the oriented interval I_0, and the resulting component(s) remain oriented, thus allowing the sorting process to continue.

$$(0 \quad 2 \quad -3 \quad -1 \quad 4),$$

$$(0 \quad 1 \quad 3 \quad -2 \quad 4).$$

Choosing oriented inversions that do not create new unoriented components, called *safe* inversions, can be done by trial and error: choose an oriented inversion, perform it, then test for the presence of new unoriented components. However, it is possible to do much better. Several different criteria exist in the literature, and we give here the simplest one, which also provides a proof of existence of safe inversions in any oriented component.

Definition 10.20 *The* score *of an inversion is the number of oriented elementary intervals in the resulting permutation.*

Theorem 10.21 ([3]) *The inversion of an oriented elementary interval of maximal score does not create new unoriented components.*

Proof Consider a permutation P and its overlap graph. Suppose that vertex I has maximal score, and that the inversion induced by I creates a new unoriented component C containing more than one vertex. At least one of the vertices in C must have been adjacent to I, since the only edges affected by the inversion are those connecting vertices adjacent to I.

Let J be a vertex formerly adjacent to I and contained in C, thus J is oriented in P.

By Proposition 10.9, the scores of I and J can be written as:

$$score(I) = T + U - O - 1,$$

$$score(J) = T + U' - O' - 1,$$

where T is the total number of oriented vertices in the overlap graph, U and O are the numbers of unoriented, respectively oriented, vertices adjacent to I, and U' and O' are the numbers of unoriented, respectively oriented, vertices adjacent to J.

All unoriented vertices formerly adjacent to I must have been adjacent to J. Indeed, an unoriented vertex adjacent to I and not to J will become oriented, and connected to J, contrary to the assumption that C is unoriented. Thus, $U' \geq U$.

All oriented vertices formerly adjacent to J must have been adjacent to I. If this was not the case, an oriented vertex adjacent to J but not to I would remain oriented, again contradicting the fact that C is unoriented. Thus, $O' \leq O$.

Now, if both $O' = O$ and $U' = U$, vertices I and J have the same set of vertices, and complementing the subgraph of I and its adjacent vertices will isolate both I and J. Therefore, we must have $score(J) > score(I)$, which is a contradiction. □

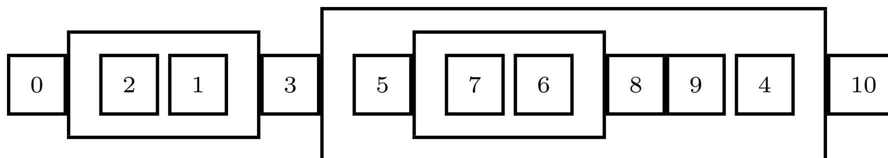
Corollary 10.22 *If a permutation P on the set $\{0, \ldots, n\}$ has only oriented components and c cycles, then $d(P) = n - c$.*

Proof By Lemma 10.8, we have $d(P) \geq n - c$ since any inversion adds at most one cycle, and the identity permutation has n cycles. Any oriented inversion adds one cycle, thus Theorem 10.21 guarantees that there will be always enough oriented inversions to sort the permutation. □

Corollary 10.22 implies that it is possible to compute the inversion distance of some permutations without actually sorting them: counting cycles is the important step, and is easily done, as we will show in Section 10.5. It is in this respect that the problem of computing the inversion distance differs from the problem of finding an optimal sorting sequence. There is no need to identify safe inversions in order to compute the distance.

10.4.2 *Computing the inversion distance*

In the preceding section, we have determined the number of inversions needed to sort a permutation which contains only oriented components. If a permutation has unoriented components, we first have to orient or destroy them. It is desirable to use as few inversions as possible for this task. Consider, for example, the following permutation which has three unoriented components. It is possible to get rid of all three of them by inverting the interval $(1 \ldots 7)$ that merges the two components $(0 \ldots 3)$ and $(5 \ldots 8)$.



In the following, we will use the tree T_P defined in Section 10.3.3 in order to compute the minimum number of inversions required to orient unoriented

components of a given permutation. The basic idea is to cover the unoriented components of T_P with paths that indicate which pairs of components should be merged together.

Definition 10.23 *A* cover $\mathcal{C}$ *of* T_P *is a collection of paths joining all the unoriented components of* P*, and such that each terminal node of a path belongs to a unique path.*

By Propositions 10.18 and 10.19, each cover of T_P describes a sequence of inversions that orients all the components of P. A path that contains two or more unoriented components, called a *long* path, corresponds to merging the two components at its terminal nodes. In Fig. 10.7, for example, a path joining components $(4 \ldots 7)$ and $(-12 \ldots -9)$ would destroy these components, along with component $(7 \ldots 16)$. A path that contains only one component, a *short* path, corresponds to cutting the component.

The *cost* of a cover is defined to be the sum of the costs of its paths, given that:

(1) the cost of a short path is 1;
(2) the cost of a long path is 2.

An *optimal* cover is a cover of minimal cost. Define t as the cost of any optimal cover of T_P.

The following theorem shows that the cost of an optimal cover is precisely the number of extra inversions needed to optimally sort a signed permutation containing unoriented components.

Theorem 10.24 ([5]) *If a permutation* P *on the set* $\{0, \ldots, n\}$ *has* c *cycles, and the associated tree* T_P *has minimal cost* t*, then we have*

$$d(P) = n - c + t.$$

Proof We first show $d(P) \leq n - c + t$. Let $\mathcal{C}$ be an optimal cover of T_P. Apply to P the sequence of m merges and q cuts induced by the cover $\mathcal{C}$. Note that $t = 2m + q$. By Proposition 10.12, the resulting permutation P' has $c - m$ cycles, since merging two components always merges two cycles, and cutting components does not change the number of cycles. Thus, by Corollary 10.22, $d(P') = n - c + m$. Since $m + q$ inversions were applied to P, we have:

$$d(P) \leq d(P') + (m + q) = n - c + 2m + q = n - c + t.$$

In order to show that $d(P) \geq n - c + t$, consider any sequence of length d that optimally sorts the permutation. By Proposition 10.7, d can be written as

$$d = s + m + q,$$

where s is the number of inversions that split cycles, m is the number of inversions that merge cycles, and q is the number of inversions that do not change the number of cycles. Since the m inversions remove m cycles, and the s inversions add

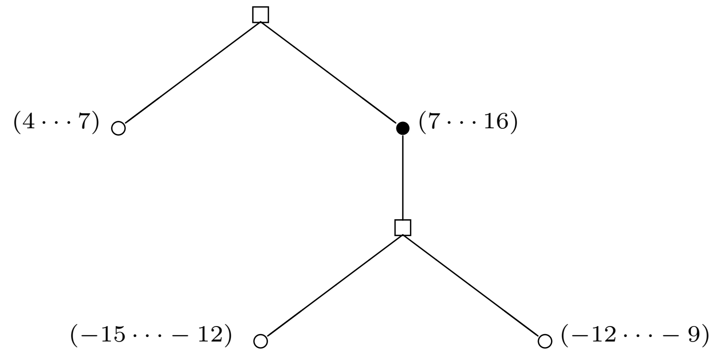


FIG. 10.8. The tree T' associated to the tree T_{P_2} of Fig. 10.7.

s cycles, we must have:

$$c - m + s = n, \quad \text{implying } d = n - c + 2m + q.$$

The sequence of d inversions induces a cover of T_P. Indeed, any inversion that merges a group of components traces a path in T_P, of which we keep the shortest segment that includes all unoriented components of the group. Of these paths, suppose that m_1 are long paths, and m_2 are short paths. Clearly we have $m_1 + m_2 \leq m$. The $q' \leq q$ remaining unoriented components are all cut. Thus

$$2m_1 + m_2 + q' \leq 2m_1 + 2m_2 + q' \leq 2m + q.$$

Since we have $t \leq 2m_1 + m_2 + q'$, we get $d \geq n - c + t$. □

The last task is to give an explicit formula for t. Let T' be the smallest unrooted subtree of T_P that contains all unoriented components of P. Formally, T' is obtained by recursively removing from T_P all dangling oriented components and square nodes. All leaves of T' will thus be unoriented components, while internal round nodes may still represent oriented components. For example, the tree T' of Fig. 10.8 is obtained from the tree T_{P_2} of Fig. 10.7. It contains three unoriented components and one oriented one.

Define a *branch* of a tree as the set of nodes from a leaf up to, but excluding, the next node of degree ≥3. A *short* branch of T' contains one unoriented component, and a *long* branch contains two or more unoriented components. For example, the tree of Fig. 10.8 has three branches, and all of them are short. We have:

Theorem 10.25 *Let T' be the unrooted subtree of T_P that contains all the unoriented components as defined above.*

1. *If T' has $2k$ leaves, then $t = 2k$.*
2. *If T' has $2k + 1$ leaves, one of them on a short branch, then $t = 2k + 1$.*
3. *If T' has $2k + 1$ leaves, none of them on a short branch, then $t = 2k + 2$.*

Proof Let $\mathcal{C}$ be an optimal cover of T', with m long paths and q shorts ones. By joining any pair of short paths into a long one, $\mathcal{C}$ can be transformed into an optimal cover with $q = 0$ or 1.

Any optimal cover has only one path on a given branch, since if there were two, one could merge the two paths and lower the cost. Thus if a tree has only long branches, there always exists an optimal cover with $q = 0$.

Since a long path covers at most two leaves, we have $t = 2m + q \geq l$, where l is the number of leaves of T'. Thus cases (1) and (2) are lower bounds. But if $q = 0$, then t must be even, and case (3) is also a lower bound.

To complete the proof, it is thus sufficient to exhibit a cover achieving these lower bounds. Suppose that $l = 2k$. If $k = 1$, the result is obvious. For $k > 1$, suppose T' has at least two nodes of degree ≥ 3. Consider any path in T' that contains two of these nodes, and that connects two leaves A and B. The branches connecting A and B to the tree T' are incident to different nodes of T'. Thus cutting these two branches yields a tree with $2k - 2$ leaves. If the tree T' has only one node of degree ≥ 3, the degree of this node must be at least 4, since the tree has at least four leaves. In this case, cutting any two branches yields a tree with $2k - 2$ leaves.

If $l = 2k + 1$ and one of the leaves is on a short branch, select this branch as a short path, and apply the above argument to the rest of the tree. If there is no short branch, select a long branch as a first (long) path. □

For example, the permutation

$$P_2 = (0\ -3\ \ 1\ \ 2\ \ 4\ \ 6\ \ 5\ \ 7\ -15\ -13\ -14\ -12\ -10\ -11\ -9\ \ 8\ \ 16)$$

has 6 cycles, as shown in Fig. 10.6. Its associated tree T', see Fig. 10.8, can be covered by one long path and one short path, since it has three leaves, all of them on short branches. Thus:

$$d(P_2) = n - c + t = 16 - 6 + 3 = 13.$$

Historical notes. There exist different criteria to choose a safe inversion. Hannenhalli and Pevzner [12] proved the existence of a safe inversion in any oriented component. Their algorithm suggests an exhaustive search for a safe inversion by trial and error, and runs in $\mathcal{O}(n^3)$ time. Berman and Hannenhalli [7] halved the number of candidates for every successive trial and bounded the number of trials by $\mathcal{O}(\log(n))$ yielding an algorithm to find a safe inversion in $\mathcal{O}(n\alpha(n))$ time, where $\alpha(n)$ is the inverse Ackermann function. Kaplan *et al.* [13] introduced the concept of a *happy clique* and developed an algorithm that finds a safe inversion in $\mathcal{O}(n)$ time. Bergeron [3] worked with an adjacency matrix to represent the overlap graph, with an additional score vector. The search for a safe inversion is simply the vertex with maximal score, and the update of the overlap graph is done with bit-vector operations.

The inversion distance formula given in Theorem 10.24 was first developed by Hannenhalli and Pevzner [12] in 1995. They introduced the notions of *hurdles*

and *fortresses* in order to express the inversion distance in terms of breakpoints, cycles, and hurdles.

In the literature the notion of *hurdle* is handled in various ways: Hannenhalli and Pevzner [12] define *minimal hurdles* as unoriented components which are minimal with respect to the order induced by span inclusion. In addition, the greatest element is a hurdle, called *greatest hurdle*, if it does not separate any two minimal hurdles. Kaplan *et al.* [13] do not distinguish between minimal and greatest hurdles since they order the elements of unoriented components on a circle. They define a *hurdle* as an unoriented connected component whose elements occur consecutively on the circle. Regardless of the precise definition of a hurdle, hurdles can be classified as follows: A *simple hurdle* is defined as a hurdle whose elimination decreases the number of hurdles, otherwise the hurdle is called a *super-hurdle*. A *fortress* is a permutation that has an odd number of hurdles, all of which are super-hurdles.

Let P be a permutation on the set $\{0,\ldots,n\}$, Hannenhalli and Pevzner proved the following:

$$d(P) = \begin{cases} n-c+h+1, & \text{if } P \text{ is a fortress,} \\ n-c+h, & \text{otherwise.} \end{cases}$$

where c is the number of cycles and h is the number of hurdles of permutation P.

10.5 Algorithms

In this section, we present algorithms to compute the inversion distance of a permutation P based on Theorems 10.24 and 10.25. The overall procedure consists of three parts. First, the number of cycles c is computed by a left-to-right scan of P, then the components of P are computed by an algorithm originally presented in reference [4], and finally the tree T_P is created by a simple pass over the components of P, followed by a trimming procedure yielding T'.

The number of cycles is computed in linear time by Algorithm 1. The idea is to mark each point of P as follows. The points of P are processed in left-to-right order, and each time an unmarked point is detected, all points on its cycle are marked, and the number of cycles is incremented by one. Adjacencies are treated as a limiting case. In order to do this efficiently, we need to know the endpoints of each elementary interval, and the pair of intervals that meet at each point. Figure 10.9 gives an example, along with tables containing the necessary information.

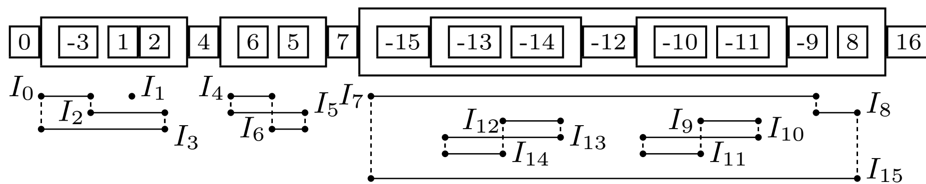
The second part of the overall procedure is the computation of the components, shown in Algorithm 2. The input of this algorithm is a signed permutation P, separated into an array of unsigned elements $\pi = (\pi_0, \pi_1, \ldots, \pi_n)$ and an array of signs $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_n)$.

Direct and reversed components are identified independently. Here we trace the algorithm only for direct components. In order to find these components, an array M is used, defined as follows: $M[i]$ is the nearest unsigned element of π that precedes π_i, and is greater than π_i, and n if no such element exists.

Algorithm 1 (Compute the number of cycles)

```
1: a point π_{p−1} · π_p is represented by the index p of its right element
2: marked[1, . . . , n] is an array of n boolean values, initially set to FALSE

3: c ← 0 (* counter for the number of cycles *)
4: for p ← 1, . . . , n do
5:    if not marked[p] then
6:       i ← one of the two intervals meeting at point p
7:       while not marked[p] do
8:          marked[p] ← TRUE
9:          i ← the interval meeting i at point p
10:         p ← the other endpoint of i
11:      end while
12:      c ← c + 1
13:   end if
14: end for
```



Elementary interval	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9	I_{10}	I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
First endpoint	1	3	4	1	5	7	6	8	16	14	12	13	11	9	10	8
Second endpoint	2	3	2	4	6	5	7	15	15	13	14	12	10	11	9	16

Point	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
First interval	I_0	I_0	I_1	I_2	I_4	I_4	I_5	I_7	I_{13}	I_{12}	I_{12}	I_{10}	I_9	I_9	I_7	I_8
Second interval	I_3	I_2	I_1	I_3	I_5	I_6	I_6	I_{15}	I_{14}	I_{14}	I_{13}	I_{11}	I_{11}	I_{10}	I_8	I_{15}

FIG. 10.9. Detecting cycles in permutation P_2 using Algorithm 1. Starting at the first point of P_2, we identify the cycle consisting of the elementary intervals I_0, I_2, and I_3. The next iteration is skipped because the second point was marked during the traversal of the first cycle. Eventually, all six cycles are recovered.

For example, the array M of permutation P_2 is:

$$P_2 = (0 \;\; -3 \;\; 1 \;\; 2 \;\; 4 \;\; 6 \;\; 5 \;\; 7 \;\; -15 \;\; -13 \;\; -14 \;\; -12 \;\; -10 \;\; -11 \;\; -9 \;\; 8 \;\; 16),$$
$$M = (16 \;\; 16 \;\; 3 \;\; 3 \;\; 16 \;\; 16 \;\; 6 \;\; 16 \;\; 16 \;\; 15 \;\; 15 \;\; 14 \;\; 12 \;\; 12 \;\; 11 \;\; 9 \;\; 16).$$

M is computed using a stack M_1 as shown in lines 5–10 of Algorithm 2.

To find the direct components (lines 11–14 of Algorithm 2), a second stack S_1 stores potential left boundary elements s, which are then tested by the following criterion: $(\pi_s \ldots \pi_i)$ is a direct component if and only if:

Algorithm 2 (Find the components of signed permutation $P = (\pi, \sigma)$)

1: M_1 and M_2 are stacks of integers; initially M_1 contains n and M_2 contains 0
2: S_1 and S_2 are stacks of integers; initially S_1 contains 0 and S_2 contains 0
3: $M[0] \leftarrow n$, $m[0] \leftarrow 0$
4: **for** $i \leftarrow 1, \ldots, n$ **do**
(* Compute the $M[i]$ *)
5: **if** $\pi[i-1] > \pi[i]$ **then**
6: push $\pi[i-1]$ on M_1
7: **else**
8: pop from M_1 all entries that are smaller than $\pi[i]$
9: **end if**
10: $M[i] \leftarrow$ the top element of M_1
(* Find direct components *)
11: pop the top element s from S_1 as long as $\pi[s] > \pi[i]$ or $M[s] < \pi[i]$
12: **if** $\sigma[i] = +$ **and** $M[i] = M[s]$ **and** $i - s = \pi[i] - \pi[s]$ **then**
13: report the component $(\pi_s \ldots \pi_i)$
14: **end if**
(* Compute the $m[i]$ *)
15: **if** $\pi[i-1] < \pi[i]$ **then**
16: push $\pi[i-1]$ on M_2
17: **else**
18: pop from M_2 all entries that are larger than $\pi[i]$
19: **end if**
20: $m[i] \leftarrow$ the top element of M_2
(* Find reversed components *)
21: pop the top element s from S_2 as long as $(\pi[s] < \pi[i]$ or $m[s] > \pi[i])$ and $s > 0$
22: **if** $\sigma[i] = -$ **and** $m[i] = m[s]$ **and** $i - s = \pi[s] - \pi[i]$ **then**
23: report the component $(-\pi_s \ldots -\pi_i)$
24: **end if**
(* Update stacks *)
25: **if** $\sigma[i] = +$ **then**
26: push i on S_1
27: **else**
28: push i on S_2
29: **end if**
30: **end for**

(1) both σ_s and σ_i are 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 $M[i] = M[s]$, and
(3) no element “between” π_s and π_i is missing, that is, $i - s = \pi_i - \pi_s$.

For example, the component $(4 \ldots 7)$ will be found in iteration $i = 7$ because:

(1) both 4 and 7 are positive,

(2) all elements between 4 and 7 are greater than 4 (since element 4 is still stacked on S_1 when $i = 7$) and smaller than 7 (since $M[4] = 16 = M[7]$), and
(3) $i - s = 7 - 4 = \pi_i - \pi_s$.

Similarly, for the detection of reversed components, we use a stack M_2 to compute $m[i]$, the nearest unsigned element of π that precedes π_i and is smaller than π_i, and a stack S_2 that stores potential left boundary elements of reversed components.

The classification of components as oriented or unoriented can be done by a slight modification of Algorithm 2, without affecting the running time. We need an extra array o to store the signs of the points of the permutation P (for ease of notation shifted down by one position). For $0 \le i < n$, the entries of the array o are initially defined as follows:

$$o[i] = \begin{cases} +, & \text{if } \sigma_i = + \text{ and } \sigma_{i+1} = +, \\ -, & \text{if } \sigma_i = - \text{ and } \sigma_{i+1} = -, \\ 0, & \text{otherwise.} \end{cases}$$

For example, the initial array o of permutation P_2 is:

$$o = (0 \quad 0 \quad + \quad + \quad + \quad + \quad + \quad 0 \quad - \quad - \quad - \quad - \quad - \quad - \quad 0 \quad +).$$

Now we define a function $f\colon \{-,0,+\}^2 \to \{-,0,+\}$ as:

$$f(x_1, x_2) = \begin{cases} x_1, & \text{if } x_1 = x_2, \\ 0, & \text{otherwise.} \end{cases}$$

Then, in the modified algorithm, whenever an index s is removed from the stack such that index r becomes the top of the stack, $o[r]$ will be replaced by $f(o[r], o[s])$. We also replace the entry of the left bounding element of an identified direct component by $+$, and the entry of the left bounding element of an identified reversed component by $-$. This way, when a direct component $(\pi_s \dots \pi_i)$ is reported in line 13 of Algorithm 2, the signs of all its points are folded by repeated application of function f to the leftmost point s of the component. Its orientation can easily then be derived: $(\pi_s \dots \pi_i)$ is unoriented if and only if

(1) $s + 1 \neq i$ (the component contains one or more breakpoints); and
(2) $o[s]$ equals $+$ or $-$ (all its points have the same sign).

The correctness of this algorithm follows from the fact that all the indices of elements of an unoriented component are stacked on the same stack, and that all its points have the same sign. If a component C contains other components, these will be identified before C, and are treated as single positive or negative elements. Since the bounding elements of oriented components have the same sign, each oriented component has at least two points for which $o(i) = 0$, and at least one index on each stack for which $o(i) = 0$.

In order to understand the third part of the overall procedure, note that Algorithm 2 reports the components in left-to-right order with respect to their

Algorithm 3 (Construct T_P from the components $C_1, \ldots, C_k$ of P)

1: create a square node q, the root of T_P, and a round node p as the child of q
2: **for** $i \leftarrow 1, \ldots, n-1$ **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:     $p \leftarrow$ parent of q
10:     $q \leftarrow$ parent of p
11:   **end if**
12: **end for**

right bounding element. For each index i, $0 \leq i \leq n$, 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_P in one left-to-right scan along the permutation. Initially one square root node and one round node representing the component with left bounding element 0 are 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.

To generate tree T' from tree T_P, a bottom-up traversal of T_P 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 inversion distance: 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 formula from Theorem 10.25 to obtain t, and the formula from Theorem 10.24 to obtain d.

Altogether we have:

Theorem 10.26 *Using Algorithms 1, 2, and 3, the inversion distance $d(P)$ of a permutation P on the set $\{0, \ldots, n\}$ can be computed in linear time $O(n)$.*

Historical notes. Traditionally, the inversion distance is computed by using the formula of Hannenhalli and Pevzner. As the hurdles and fortresses are detectable from connected component analysis, the most delicate part is to compute the connected components. The existing algorithms solve this problem in different ways. The initial algorithm of Hannenhalli and Pevzner [12], restricted to the computation of the inversion distance, runs in quadratic time by constructing the overlap graph. In 1996, Berman and Hannenhalli [7] developed a faster algorithm for computing the connected components, yielding an algorithm

to compute $d(P)$ in $\mathcal{O}(n \cdot \alpha(n))$ time. They used a Union/Find structure to maintain the connected components of the overlap graph, without constructing the graph itself. In 2001, Bader *et al.* [1] gave the first linear time algorithm for computing the inversion distance. By scanning the permutation twice, their algorithm constructs another graph, called the *overlap forest*, which has exactly one tree per connected component of the overlap graph.

10.6 Conclusion

This chapter gave an elementary presentation of the results of the classical Hannenhalli–Pevzner theory on the inversion distance problem. Most of the results are obtained by working directly on the elements of the permutation, instead of relying on intermediate constructions. This effort yielded a simpler equation for the distance, an increased understanding of the effects of inversions on a permutation, and the development of very elementary algorithms.

Looking at the problem from this point of view led to some interesting variants of genome comparison tools. The concept of *conserved intervals* [4, 6], for example, can be used to measure the similarity of a set of permutations. It is a direct offspring of the crucial role played by components in the inversion problem.

This work is also a first step in the simplification of the problem of comparing multi-chromosomal genomes. Rearrangement operations between these genomes include, among others, inversions, translocations, fusions, and fissions of chromosomes. The algorithmic treatment of this problem relies on the properties of the sorting by inversions problem, and currently involves half a dozen parameters [12]. The initial solution contained gaps that took years to be closed [19, 25]. A linear algorithm for the translocation distance problem [11] is given in reference [16].

Another crucial extension is the ability to handle insertions, deletions, and duplications of genes. This extension is much harder, but much more important for biological applications. Indeed, one of the main driving forces of genome evolution are segment duplications. Recent work on this problem can be found in [10, 17], and is surveyed in the Chapters 11 and 12, this volume.

Glossary

adjacency	pair of consecutive integers, Section 10.2
bounding elements	first and last elements of an interval, Section 10.2
branch	set of nodes from a leaf to the next node of degree ≥ 3, Section 10.4
breakpoint	a point that is not an adjacency, Section 10.2
chain	a sequence of components overlapping on one element, Section 10.3.3

component	Definition 10.10
cover	Definition 10.23
cycle	Definition 10.4
direct component	Definition 10.10
elementary interval	Definition 10.1
endpoints	first and last points of an interval, Section 10.2
extremities	Definition 10.1
long branch	branch containing more than one unoriented component, Section 10.4
oriented component	Definition 10.13
oriented interval	elementary interval whose inversion creates an adjacency, Section 10.3.1
overlapping intervals	Definition 10.6
point	pair of consecutive elements, Section 10.2
reversed component	Definition 10.10
safe inversion	oriented inversion that does not create new unoriented components, Section 10.4.1
score	Definition 10.20
sign of a point	Definition 10.13
short branch	branch containing one unoriented component, Section 10.4
sorting inversion	an inversion that belongs to an optimal sorting sequence, Section 10.2
sorting sequence	sequence of inversions that transform a permutation into the identity permutation, Section 10.2
tree T_P	Definition 10.15
unoriented component	Definition 10.13
unoriented interval	elementary interval whose inversion does not create an adjacency, Section 10.3.1

References

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

[2] Bafna, V. and Pevzner, P.A. (1996). Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, **25**(2), 272–289.

[3] Bergeron, A. (2001). A very elementary presentation of the Hannenhalli–Pevzner theory. In *Proc. of 12th Symposium on Combinatorial Pattern Matching (CPM'01)* (ed. A. Amihood and G.M. Landau), Volume 2089 of Lecture Notes in Computer Science, pp. 106–117. Springer-Verlag, Berlin.

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

[5] Bergeron, A., Mixtacki, J., and Stoye, J. (2004). Reversal distance without hurdles and fortresses. In *Proc. of 15th Symposium on Combinatorial Pattern Matching (CPM'04)* (ed. S.C. Sahinalp, S. Muthukrishnan and U. Dogrusoz), Volume 3109 of Lecture Notes in Computer Science, pp. 388–399. Springer-Verlag, Berlin.

[6] Bergeron, A. and Stoye, J. (2003). On the similarity of sets of permutations and its applications to genome comparison. In *Proc. of 9th Conference on Computing and Combinatorics (COCOON'03)* (ed. T. Warnow and B. Zhu), Volume 2697 of Lecture Notes in Computer Science, pp. 68–79. Springer-Verlag, Berlin.

[7] Berman, P. and Hannenhalli, S. (1996). Fast sorting by reversal. In *Proc. of 7th Combinatorial Pattern Matching (CPM'96)* (ed. D.S. Hirschberg and E.W. Myers), Volume 1075 of Lecture Notes in Computer Science, pp. 168–185. Springer-Verlag, Berlin.

[8] Booth, K.S. and Lueker, G.S. (1976). Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. *Journal of Computer and System Sciences*, **13**(3), 335–379.

[9] Caprara, A. (1997). Sorting by reversals is difficult. In *Proc. of 1st Conference on Computational Molecular Biology (RECOMB'97)* (ed. M. Waterman), pp. 75–83. ACM Press, New York.

[10] El-Mabrouk, N. (2000). Genome rearrangement by reversals and insertions/deletions of contiguous segments. In *Proc. of 11th Conference on Combinatorial Pattern Matching (CPM'00)* (ed. R. Giancarlo and D. Sankoff), Volume 1848 of Lecture Notes in Computer Science, pp. 222–234. Springer-Verlag, Berlin.

[11] Hannenhalli, S. (1996). Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, **71**(1–3), 137–151.

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

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

[14] Kececioglu, J. and Sankoff, D. (1993). Exact and approximation algorithms for the inversion distance between two chromosomes. In *Proc. of 4th Conference on Combinatorial Pattern Matching (CPM'93)* (ed. A. Apostolico, M. Crochemore, Z. Galil and U. Manber), Volume 684 of Lecture Notes in Computer Science, pp. 87–105. Springer-Verlag, Berlin.

[15] Kececioglu, J.D. and Sankoff, D. (1994). Efficient bounds for oriented chromosome inversion distance. In *Proc. of 5th Conference on Combinatorial Pattern Matching (CPM'94)* (ed. M. Crochemore and D. Gusfield), Volume 807 of Lecture Notes in Computer Science, pp. 307–325. Springer-Verlag, Berlin.

[16] Li, G., Qi, X., Wang, X., and Zhu, B. (2004). A linear-time algorithm for computing translocation distance between signed genomes. In *Proc. of 15th Symposium on Combinatorial Pattern Matching (CPM'04)* (ed. S.C. Sahinalp, S. Muthukrishnan and U. Dogrusoz), Volume 3109 of Lecture Notes in Computer Science, pp. 323–332. Springer-Verlag, Berlin.

[17] Marron, M., Swenson, K., and Moret, B. (2003). Genomic distances under deletions and insertions. In *Proc. of 9th Conference on Computing and Combinatorics (COCOON'03)* (ed. T. Warnow and B. Zhu), Volume 2697 of Lecture Notes in Computer Science, pp. 537–547. Springer-Verlag, Berlin.

[18] Nadeau, J.H. and Taylor, B.A. (1984). Lengths of chromosomal segments conserved since divergence of man and mouse. *Proceedings of the National Academy of Sciences USA*, **81**, 814–818.

[19] Ozery-Flato, M. and Shamir, R. (2003). Two notes on genome rearrangements. *Journal of Bioinformatics and Computational Biology*, **1**(1), 71–94.

[20] Sankoff, D. (1992). Edit distances for genome comparison based on non-local operations. In *Proc. of 3rd Conference on Combinatorial Pattern Matching (CPM'92)* (ed. A. Apostolico, M. Crochemore, Z. Galil, and U. Manber), Volume 644 of Lecture Notes in Computer Science, pp. 121–135. Springer-Verlag, Berlin.

[21] Setubal, J. and Meidanis, J. (1997). *Introduction to Computational Molecular Biology.* PWS Publishing, Boston.

[22] Siepel, A. (2002). An algorithm to find all sorting reversals. In *Proc. of 2nd Conference on Computational Molecular Biology (RECOMB'02)* (ed. G. Myers, s. Hannenhalli, S. Istrail, P. Pevzner and M. Waterman), pp. 281–290. ACM Press, New York.

[23] Sturtevant, A.H. (1926). A crossover reducer in drosophila melanogaster due to inversion of a section of the third chromosome. *Biologisches Zentralblatt*, **46**(12), 697–702.

[24] Tannier, E. and Sagot, M.F. (2004). Sorting by reversals in subquadratic time. In *Proc. of 15th Symposium on Combinatorial Pattern Matching (CPM'04)* (ed. S.C. Sahinalp S. Muthukrishnan and U. Dogrusoz), Volume 3109 of Lecture Notes in Computer Science, pp. 1–13. Springer-Verlag, Berlin.

[25] Tesler, G. (2002). Efficient algorithms for multichromosomal genome rearrangements. *Journal of Computer and System Sciences*, **65**(3), 587–609.

[26] Watterson, G.A., Ewens, W.J., and Hall, T.E. (1982). The chromosome inversion problem. *Journal of Theoretical Biology*, **99**(1), 1–7.