Sorting by weighted transpositions and reversals

Diploma thesis at the University of Ulm
Faculty of Computer Science

presented by:

Martin Bader

1. evaluator: *Prof. Dr. Enno Ohlebusch*
2. evaluator: *Dr. Mohamed Ibrahim Abouelhoda*

2005

Contents

Chapter 1

Introduction

1.1 Biological background

For understanding *genome rearrangements*, a certain knowledge of biology and molecular genetics is necessary. Therefore, we will now provide a brief introduction to molecular genetics. This introduction only covers the subjects that are necessary for understanding genome rearrangements, additional information can be found in any textbook about molecular biology. The main information of this chapter has been taken from [SM97] and [Bro02].

1.1.1 Cells, DNA, and Proteins

All biological life consists of *cells*, where we distinguish between *eukaryotes* and *prokaryotes*. Eukaryotes are organisms that consist of cells having a nucleus. For example, all vertebrates are eukaryotes. In prokaryotes, the cells do not have a nucleus. Prokaryotes consists of bacteria and archaea. Cells can have very different tasks in the organism, but each cell contains the whole genetic information of the organism. Cells spawn by cleavage, where the whole genetic information is passed.
The genetic information is stored in large molecules, called *deoxyribonucleic acid* (short **DNA**). The cells contain the DNA as a double helix, consisting of two **strands**. Each strand consists of a sequence of simple molecules. These molecules are called *nucleotides*, which consist of of the sugar-molecule *2'deoxyribose*, a phosphate group, and a base. The sugar-molecule contains five carbon atoms, which are labeled 1' to 5', as illustrated in Figure 1.1. The phosphate group is attached to the 5'-carbon of the sugar, and the base is attached to the 1'carbon. Two nucleotides are linked together by *phosphodiester bonds* between their 5'- and 3'-carbons [Bro02]. We can say that the chain consists of a backbone of the sugar molecules and the phosphate residues, and the bases are attached to this backbone (see Figure 1.2). With these bonds, we can assign a natural orientation to the chain. By convention it goes from the 5'-end to the 3'-end. There are four different bases which can be attached to the sugar molecule: adenine (A), guanine (G), cytosine (C), and thymine (T). The sequence of the bases of the strand codes the genetic information,

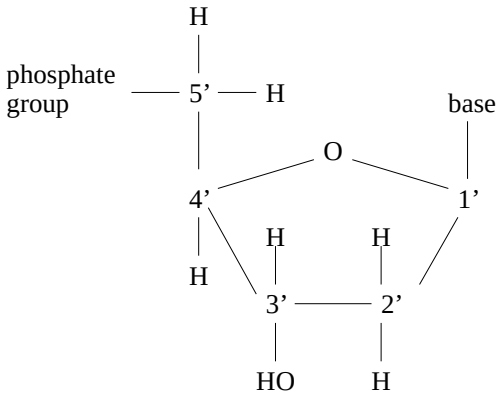


Figure 1.1: A single nucleotide. The carbon atoms are labelled 1' to 5'.

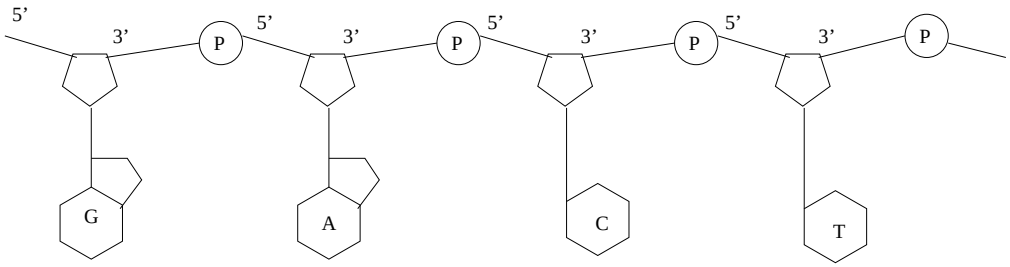


Figure 1.2: A schematic view on a single DNA strand.

so we write a strand by simply writing this sequence. As mentioned above, the DNA consists of two strands, building the famous double-helix. The reason for this is that the bases build hydrogen bonds to their complementary bases: Adenine binds to thymine, and cytosine binds to guanine. We also speak of the **Watson-Crick base pairs** A-T and C-G. The second strand (also called **lagging strand**) consists of a sequence of complementary bases to the first strand (the **leading strand**). The leading and the lagging strand have a different orientation, e.g. the 5'-end of the leading strand binds with the 3'-end of the lagging strand. Therefore, the lagging strand is the **reverse complement** of the leading strand: the bases are exchanged by their complement, and the order of the sequence of bases is inverted. For an illustrating example, see Figure 1.3.

A measure for the size of a DNA molecule is the number of these base pairs, abbreviated with **bp**. The abbreviations **kbp** (for kilo base pairs = 10^3 bp) and **Mbp** (for Mega base pairs = 10^6 bp) are also common. A single DNA-molecule is called **chromosome** [1], and an organism can contain one or more chromosomes. All chromosomes together are called the **genome**. For example, the human genome consists of 23 pairs of chromosomes. For eukaryotes, the genome is in the cell nucleus.

[1] also plasmids, mitochondrials, and chloroplasts are DNA molecules, but the main genetic data is stored in the chromosomes

leading strand: 5' ... ACCGTATGGAC ... 3'
lagging strand: 3' ... TGGCATACCTG ... 5'

read lagging strand from 5' to 3': $\Rightarrow$ ... GTCCATACGGT ...

Figure 1.3: When read from 5' to 3', the lagging strand is the reverse complement of the leading strand. Bases are replaced by their complement, and the order of the bases is inverted.

An important role of the genome is the coding of *proteins*. Proteins are chains of *amino acids*, and perform many different tasks in an organism (for example, proteins known as *enzymes* act as catalysts of chemical reactions). Each amino acid is coded by a triplet of bases, called **codon**. The whole protein is coded as a substring of the DNA, beginning with a start codon and terminated by a stop codon. But not the whole DNA is coding proteins. There are some contiguous stretches that do code proteins while others do not. The stretches that code proteins are called **genes**. As the sequence of base pairs on the leading strand is different from the sequence on the lagging strand, the coding information of one gene is on one of the strands. We therefore can say that a gene is on the leading strand or on the lagging strand.

1.1.2 Genome dynamics

As a result of various mutation or recombination events, genomes can change over time. Some of these changes destroy the functionality of the cell and let the cell die. Others change the behaviour of the cell, leading to malfunctions which can harm the organism (such as cancer cells). But many changes have no effect on the cell at all, and some can even increase the fitness of the organism. These changes are inherited by child cells, leading to organisms with a slightly changed genome.
Although mutations and recombinations both result in a change of the genome, we distinguish between these two events:

- A **mutation** is a change in the nucleotide sequence of a short region of the genome [Bro02]. Many mutations are **point mutations**, changing just a single base pair. We distinguish between three types of point mutations:

  Replacement: a single base pair has been replaced by another base pair.

  Insertion: an additional base pair has been inserted into the DNA molecule.

  Deletion: a base pair has been removed from the DNA molecule.

  Mutations can have different causes: physical and chemical influences (such as radiation) can interact with the DNA and change the structure of an individual nucleotide. Also the replication mechanism of the DNA can cause mutations. Although it is combined with an error-fixing mechanism, a few of the replication errors remain.

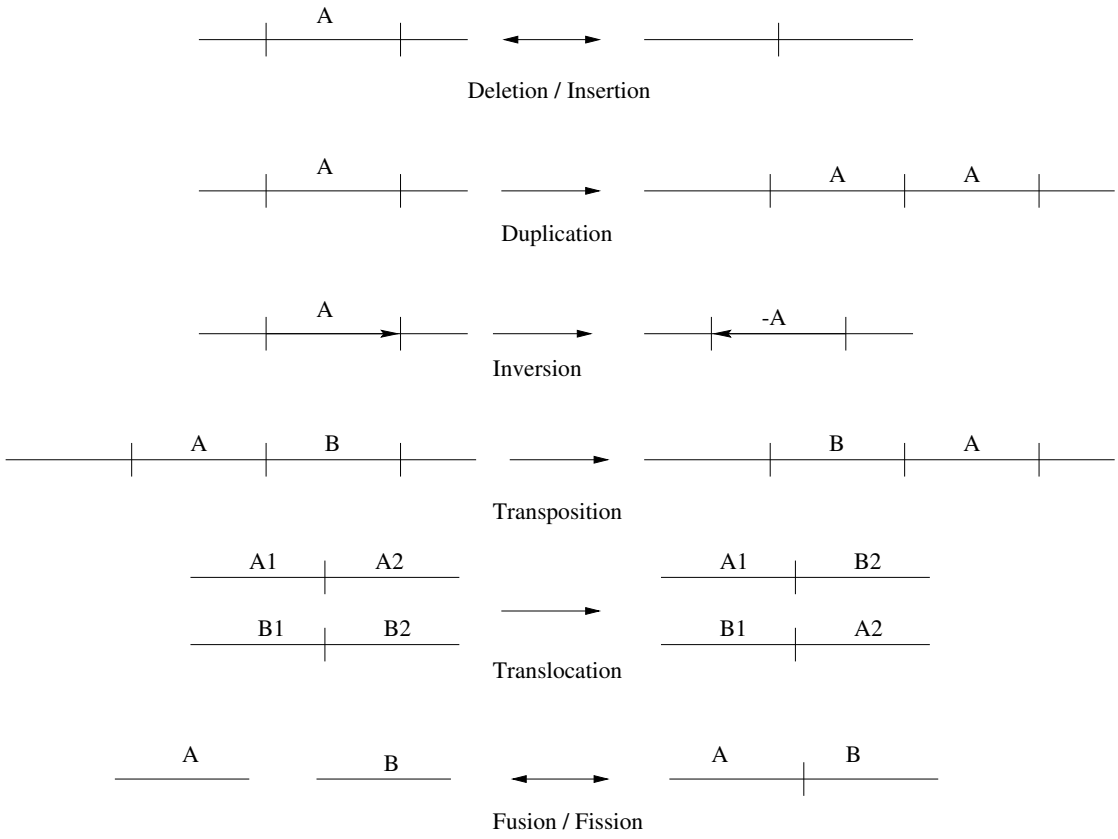


Figure 1.4: The possible genome rearrangements.

For *E.coli*, the overall error rate for replication is 1 in 10^{10} to 1 in 10^{11} base pairs [Bro02].

- A **recombination** is an event that changes the genome in a larger scale. Examples for recombination are the exchange of homologous chromosomes during meiosis, or a rearrangement of the genes in the DNA. We will focus on these **genome rearrangements**and distinguish between several possible types (see also Figure 1.4):

  Deletion: A segment of a chromosome is deleted, the genes on this segment are lost.

  Insertion: A new segment (with new genes) is inserted into a chromosome.

  Duplication: A segment of the chromosome is duplicated. Its genes appear now twice in the chromosome.

  Inversion: A segment of a chromosome is cut off from the chromosome and is inserted at the same position, but in the different direction. As the 5'→3'-orientation of both strands must be maintained, the leading strand of the cut out segment is inserted into the lagging strand of the chromosome and vice versa. The result of an inversion is that the order of the genes on the segment

is inverted, and genes from the leading strand are now on the lagging strand and vice versa. Although in biology this event is called inversion, in computer science is rather known as **reversal**.

Transposition: A segment of the chromosome is cut off and is inserted at another position in the chromosome. If the segment has also been inverted, we speak of an **inverted transposition** (in computer science, it is called **transreversal**).

Translocation: Two segments at the end of two different chromosomes are exchanged.

Fission: A chromosome is split up into two new chromosomes.

Fusion: Two chromosomes are linked together, building one new chromosome.

Genome rearrangements occur very rarely in nature. Among vertebrates there are about 0.2 to 2 rearrangements per million years [BBD+99]. Therefore measuring the number of genome rearrangements that are necessary for transforming the genome of one species into the genome of another species is a good approach to measure the evolutionary distance between these two species.

1.1.3 Prokaryotic DNA

As our algorithm is primarily designed to be applied to prokaryotic DNA, we will discuss some aspects about their DNA here. As mentioned above, prokaryote cells have no nucleus, and the DNA is moving freely in the cell. The size of these genomes is, compared to eukaryotes (the human genome has about 3000 Mbp), relatively small: most prokaryotes have a genome smaller than 5 Mbp. Mostly, the genome is a single circular DNA molecule (i.e. the double helix builds a ring). In a circular DNA molecule, one can not distinguish between the leading and the lagging strand: one can choose any strand as leading strand, and the other one is the lagging strand. Therefore, the molecule is equivalent to its *reflection*, which is the same molecule, but the definition of leading and lagging strand is exchanged.
Prokaryotes can have additional small DNA molecules, called *plasmids*. The genes carried by the plasmids appear to be not necessary for the function of the organism, but can be quite useful: For example, they can provide antibiotic resistance. The number of genes in the genome is also smaller than in eukaryotes, but not as much as expected by the difference in genome size: prokaryotic genomes have only small non-coding regions, and the number of repeats (i.e. DNA sequences that appear at different positions in the DNA) is low. In summary, the genetic information in prokaryotic genomes is much more packed than in eukaryotic genomes. If we look at genome rearrangements, the most frequent operation are inversions, but also transpositions can be observed. Especially inversions around the replication origin (the position in the DNA where the replication begins) are often observed.

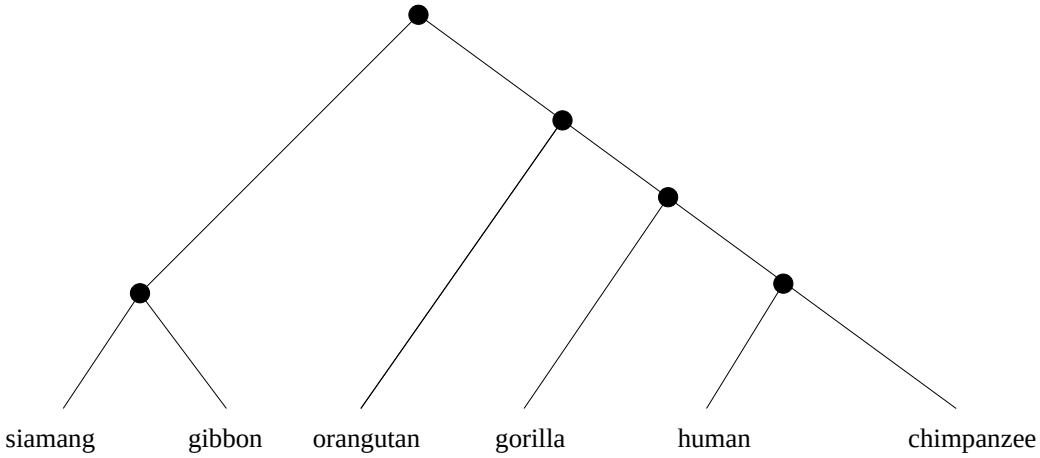


Figure 1.5: A phylogenetic tree for some primates [SM97].

1.1.4 Phylogenetic reconstruction

An important task in biology is to reconstruct the process of evolution and to predict the phylogeny of a given set of species. These relationships are normally represented in a tree, where each node of the tree represents a species. The leaves of the tree represent present-day species, inner nodes represent ancestors, and the edges represent the ancestor-descendant relation. These trees are called **phylogenetic trees**. An example can be seen in Figure 1.5.

Creating phylogenetic trees is a difficult task, for various reasons:

- Ancestors in the tree are often extinct species, and there is only uncertain data available. In many cases, even these are not available and we have to construct hypothetic ancestors from the data of the present-day species.
- For creating the tree, it is necessary to determine the evolutionary distance of the species. This can be a very hard task if the species in the set are very similar, and the distances can only be gained by a hypothesis.
- The tree structure is sometimes a too simplified view of the evolution. In reality, there can exist cross-links between similar species, building a graph rather than a tree. Prokaryotes also sometimes transfer genes from one species to another. In this case, we speak of lateral gene transfer. Lateral gene transfer has even been observed between bacteria and archaea. It can be difficult to distinguish if an identical gene in two species comes from a common ancestor, or from lateral gene transfer.

Although it is an interesting task to construct the tree if distances are known, this will not be discussed here. Instead, we will have a look at distance functions that use genomic data. The classical approach is to count the point mutations in homologous regions of two genomes. However, this method can only handle local mutations and ignores global

rearrangements (i.e. inversions, transpositions and so on) completely. In several cases, this approach is not satisfying, and counting the genome rearrangements that are necessary to transform one genome into another can result in a more realistic distance function. This approach can use homologous blocks instead of base pairs as data, and therefore reduce the amount of data drastically. Another advantage is that genome rearrangements occur rarely compared to point mutations, which can also increase the quality of the distance function.
One of the first phylogenetic trees based on a genome rearrangement distance was presented by Dobzhansky and Sturtevant in 1938, who studied chromosomes of the fruit fly *Drosophila pseudoobscura* in different regions of the western USA and Mexico [DS38]. Without sequenced genomes, they had to find inversions out of the threedimensional structure of the chromosomes: if an inversion has occured in a chromosome, but not in its homologous chromosome, the structure of these chromosomes builds a loop which can be seen with a microscope. However, this technique is limited to a small number of inversions. With the sequencing of genomes, the gene order can be determined more easily. Nowadays, it is of interest to determine non-trivial genome rearrangement distances as an evolutionary distance for phylogenetic reconstruction.

1.2 Previous works

The genome rearrangement problem - finding the most plausible sequence of genome rearrangements that transforms one genome into another - has become a challenging topic in the 1990's. So far all approaches use simplifications. Most algorithms work on single chromosomes, and the set of genome rearrangement operations is restricted. Except for a few exceptions all algorithms weight all operations equally and search for the shortest sequence that transforms one permutation into another (the source genome can be represented as a permutation of the genes of the target permutation, therefore we can reduce the problem to sorting of permutations). Even with these simplifications, the problem is still very hard: A. Caprara showed that *Sorting by reversals* is NP-hard [Cap97]. However, he did not consider the orientation of the genes (i.e. wether they are on the leading or on the lagging strand). If this orientation is considered, we have the problem *Sorting signed permutations by reversals*, and S. Hannenhalli and P.A. Pevzner showed that this problem can be solved in polynomial time [HP99]. The Hannenhalli-Pevzner theory was simplified [Ber05] and the running time of the algorithm has been improved several times. To date, a subquadratic time algorithm is available [TS04]. If we only ask for the *reversal distance* (where we are solely interested in the minimum number of required reversals but not in the sequence of reversals) it is solvable in linear time [BMS04]. If one restricts the set of operations to transpositions (T), to transpositions and reversals (T + R), or to transpositions, reversals, and transreversals (T + R + TR), the complexity of the problem is still unknown. There exist polynomial-time approximation algorithms, and the best of them are listed in the table below.

operations	T	T + R	T + R + TR
performance ratio	1.375	2	1.5
references	[EH05]	[WDM98, LX01]	[HS04]

Using different operations, it is also of interest to assign weights to the operations, and search for the sequence of operations with the lowest weight: in nature, for example, reversals are much more frequently observed than transpositions and should therefore have a smaller weight. Algorithms with equal weights for reversals and transpositions tend to favor transpositions (we will see the reason for this later). Consequently, the sequence of rearrangement operations returned by these algorithms will often significantly deviate from the "true" evolutionary history.
However, sorting signed permutations with weighted reversals and transpositions is poorly studied. To our knowledge, there are only two algorithms that tackle it. The first is a $(1+\varepsilon)$-approximation algorithm devised by Eriksen [Eri02]. It uses a weight proportion 2:1 (transposition:reversal) and has the tendency to use as much reversals as possible. The second algorithm is implemented in the software tool DERANGE II [BKS96]. It is a greedy algorithm that works on the breakpoint distance and can only guarantee an approximation ratio of 3.
In this work we will present a 1.5-approximation algorithm for any weight proportion between 1:1 and 2:1. Hence, our result closes the gap between the result of Hartman and Sharan [HS04] for the 1:1 proportion and that of Eriksen [Eri02] for the 2:1 proportion.

1.3 Structure of this work

In this work, we will develop a 1.5-approximation algorithm for sorting circular permutations by reversals, transpositions, and transreversals.
In Chapter 2, we will introduce some elementary definitions and give a mathematical definition of the problem. We will also prove some reductions that simplify the problem and provide all lemmata necessary to develop the algorithm. Furthermore, we show a non-trivial lower bound for the weighted distance of two permutations.
In Chapter 3, we begin with a short explanation of the idea of the algorithm and give an overview over the case analysis used in the algorithm. Then, we have a closer look at all cases of the analysis, and show how we can solve them. This provides us everything we need to implement the algorithm.
In Chapter 4, we show how to embed the case analysis into an algorithm, and discuss the running time of the algorithm. We show how the performance can be improved by combining the algorithm with a greedy strategy. For the comparison of the results of the algorithms, we also provide a branch and bound algorithm that can solve the sorting problem correctly. However, it can only handle small permutations.
In Chapter 5, we use random permutations to compare the algorithms. The algorithms tested are the approximation algorithm with and without the greedy strategy, the branch and bound algorithm, and the program DERANGE II developed by Blanchette et al [BKS96]. The running times as well as the approximation ratios will be compared for

test sets of different size.
Finally, in Chapter 6, we will outline the most significant results, and discuss the problems that are still open.

Chapter 2

Preliminaries

2.1 Elementary definitions

Before we begin to develop the algorithm, we need some elementary definitions:

Definition 1. *A **signed permutation** $\pi = (\pi_1 \dots \pi_n)$ is a permutation of $(1 \dots n)$, in which each element is labeled by plus or minus. If the indices are cyclic (that is, π_1 follows π_n), we speak of a **circular signed permutation**.*

As we will not discuss the problem of sorting unsigned permutations, every time we speak of a permutation, we mean a signed permutation.

Definition 2. *The **identity permutation** is the permutation*

$$id = (+1 \ +2 \ \dots \ +n)$$

where n is the number of elements in the permutation.

Definition 3. *The **reflection** of a permutation π is a permutation that can be obtained from π by inverting the order of elements and flipping the sign of each element:*

$$refl(\pi) = (-\pi_n - \pi_{n-1} \cdots - \pi_1)$$

For circular permutations, a permutation and its reflection are considered to be biologically equivalent.

Definition 4. *A **segment** $\pi_i \dots \pi_j$ (with $j \geq i$) of a permutation π is a consecutive sequence of elements in π, with π_i as first element and π_j as last element.*

There are three possible rearrangement operations on our permutation π:

Definition 5. *A **transposition** $t(i, j, k)$ (with $i < j$ and $k < i$ or $k > j$) is an operation that cuts the segment $\pi_i \dots \pi_{j-1}$ out of π, and inserts it before the element π_k.*

$$\begin{aligned}
t(3,6,7)\ (+6\ +1\ -5\ -3\ +4\ -2\ +7) &= (+6\ +1\ -2\ -5\ -3\ +4\ +7)\\
r(3,6)\ (+6\ +1\ -5\ -3\ +4\ -2\ +7) &= (+6\ +1\ -4\ +3\ +5\ -2\ +7)\\
tr(3,6,7)\ (+6\ +1\ -5\ -3\ +4\ -2\ +7) &= (+6\ +1\ -2\ -4\ +3\ +5\ +7)
\end{aligned}$$

Figure 2.1: Some example operations on a permutation

$$\begin{aligned}
tr(5,6,4)\ (+6\ +1\ -5\ -3\ +4\ -2\ +7) &= (+6\ +1\ -5\ -4\ -3\ -2\ +7)\\
t(1,2,7)\ (+6\ +1\ -5\ -4\ -3\ -2\ +7) &= (+1\ -5\ -4\ -3\ -2\ +6\ +7)\\
r(2,6)\ (+1\ -5\ -4\ -3\ -2\ +6\ +7) &= (+1\ +2\ +3\ +4\ +5\ +6\ +7)
\end{aligned}$$

Figure 2.2: A sequence that transforms the permutation into the identity permutation

Definition 6. *A **reversal** $r(i,j)$ (with $i<j$) is an operation that inverts the order of the elements of the segment $\pi_i \dots \pi_{j-1}$. Additionally the sign of each element in the segment will be flipped.*

Definition 7. *A **transreversal** $tr(i,j,k)$ (with $i<j$ and $k<i$ or $k>j$) is the concatenation $t(i,j,k) \circ r(i,j)$ of a reversal and a transposition. In other words, the segment $\pi_i \dots \pi_{j-1}$ will be cut out of π, inverted, and inserted before π_k*

Remark. If our operation is a transposition $t(i,j,k)$, we can regard the operation as the exchange of the segments $\pi_i, \dots, \pi_{j-1}$ and $\pi_j, \dots, \pi_{k-1}$. Hence $t(i,j,k)$ is equivalent to $t(j,k,i)$, and we can write every transposition as a transposition $t(i,j,k)$ with $i<j<k$.

Definition 8. *A **sequence** of operations $op_1, op_2, \dots, op_n$ consists of some consecutive operations on a permutation π, yielding the permutation $op_n \circ op_{n-1} \circ \dots \circ op_1(\pi)$. If the sequence transforms π into the identity permutation, it is called a **sorting sequence**.*

Figure 2.2 shows an example sequence that transforms a permutation into the identity permutation.

Definition 9. *The **weight** w is a function that returns for each operation a value ≥ 0, i.e. $w : \{op(i,j,k)\} \to \mathbb{R}^+$. The function can depend on the operation type and its parameters.*

This definition is biologically motivated: Some types of operations occur more often than others. Also the likelihood of operations acting on short segments is observed to be higher than that of operations acting on long segments. The weights should represent the likelihood of operations: The higher the likelihood, the smaller the weight. However, it is very difficult to devise an algorithm that can work with arbitrary weights, so we use a simplification: The weights depend only on the operation type, not on the parameters. We will use only two different constant weights:

- w_r is the weight of a reversal
- w_t is the weight for a transposition or a transreversal

This is motivated by the fact that a reversal splits the genome at two positions, while all other operations have to split the genome at three positions. For most applications, the w_r should be smaller than w_t, but not too small. The most realistic weight proportion depends on the species we want to compare. Our algorithm is designed for a weight ratio of $w_r : w_t = 1 : 1.5$ and works correctly for any ratio between $1 : 1$ and $1 : 2$.

Definition 10. *The weight w of a sequence of operations is the sum of the weights of the operations.*

We will focus our sight on circular permutations. In the next section we will show how we can adapt an algorithm that works on circular permutations, so that it also works on linear permutations.

Definition 11. *The problem **sorting circular permutations by weighted reversals and transpositions** is defined as follows: Given a permutation π, find a sequence of operations with minimum weight that transforms π into the identity permutation. All four types of operations are allowed.*
*The weight of this sequence is called the **weighted distance** $d_w(\pi)$.*

The sequence shown in Figure 2.2 has the weight $w = w_r + 2w_t$ and transforms the permutation into the identity permutation. In fact, if $\frac{w_r}{w_t} = \frac{2}{3}$, this sequence is optimal.

2.2 Linear vs. circular permutations

Our algorithm is designed to solve the problem on circular permutations. If we allow a new operation, called *revrev*, we can adapt it to linear permutations. We will follow the proof given in [HS04].

Definition 12. *A **revrev** $rr(i, j, k)$ (with $i < j < k$) is the concatenation $r(j, k) \circ r(i, j)$ of two reversals. In other words, the consecutive segments $\pi_i \dots \pi_{j-1}$ and $\pi_j \dots \pi_{k-1}$ will be inverted.*

As further restriction, the weight of a revrev must be the same as the weight of a transreversal.
For solving the sorting problem on a linear permutation $\pi_{lin} = \pi_1\pi_2 \dots \pi_n$, we begin by transforming it into a circular permutation π_{circ}. This is done by simply inserting a dummy element $\pi_0 = 0$ that is adjacent to π_1 and π_n, i.e.

$$\pi_{circ} = \pi_0\pi_1\pi_2 \dots \pi_n$$

Suppose that we have a sorting sequence for π_{circ}, and no operation of the sequence operates on π_0, i.e. π_0 is never on a segment that is inverted by a reversal or moved by a transposition. Then any operation of the sequence is also a valid operation on the linear permutation, and the sequence can be used to sort π_{lin}.

Lemma 13. *[HS04] Let π_x be an element of a circular permutation π_{circ}, and let op be an operation that operates on π_x. Then there exists an equivalent operation op′ (i.e. op and op′ have the same weight, op(π) is equivalent to op′(π)) that does not operate on π_x.*

Proof. For proving this, we must recall that a circular permutation is equivalent to its reflection. We will now distinguish between the possible cases:

- Let *op* be the reversal $r(i, j)$ with $i \leq x < j$ (so the reversal operates on π_x). Then the reversal $r(j, i)$ does not act on π_x, and $r(j, i)\pi_{circ}$ is the reflection of $r(i, j)\pi_{circ}$. For an illustrating example, see Figure 2.3.

- Let *op* be the transposition $t(i, j, k)$ with $i \leq x < j$. Let A be the segment $\pi_i \dots \pi_{j-1}$, let B be the segment $\pi_j \dots \pi_{k-1}$, and let C be the segment $\pi_k \dots \pi_{i-1}$. Looking at the circular permutation, the transposition exchanges the segments A and B, so the order of the segments is changed from ABC to BAC. As the permutation is cyclic, the order BAC is equivalent to ACB and CBA. These orders can be obtained from π_{circ} by the transpositions $t(j, k, i)$ and $t(k, i, j)$. Thus, $t(i, j, k)\pi_{circ}$, $t(j, k, i)\pi_{circ}$, and $t(k, i, j)\pi_{circ}$ are all equivalent, and the last two of these transpositions do not operate on π_x (see also Figure 2.3).

- Let *op* be the transreversal $tr(i, j, k)$ with $i \leq x < j$. Let A be the segment $\pi_i \dots \pi_{j-1}$, let B be the segment $\pi_j \dots \pi_{k-1}$, and let C be the segment $\pi_k \dots \pi_{i-1}$. The transreversal exchanges the segments A and B, and changes the orientation of A. Thus, the resulting order of segments is $+B\ -A\ +C$. Now, let us have a look at the revrev $rr(j, k, i)$. Applied on π_{circ}, it changes the orientation of the segments B and C, so the resulting order of the segments is $+A\ -B\ -C$. This is the (cyclic shifted) reflection of $+B\ -A\ +C$. Therefore, $tr(i, j, k)\pi_{circ}$ and $rr(j, k, i)\pi_{circ}$ are equivalent, and $rr(j, k, i)$ does not operate on π_x (see also Figure 2.3).

□

We can now transform a sequence for sorting π_{circ} into a sequence for π_{lin} by exchanging any operation that operates on the dummy element 0 by an equivalent operation. Exchanging an operation can shift the arguments of the following operations or even change their order, but it is easy to track these effects. As no operation operates on the dummy element, we can also assure that it is not inverted in the resulting permutation. Therefore, the resulting permutation is the identity permutation and not its reflection (note that these are not equivalent for linear permutations).
So far, we have shown that we can use a sorting sequence seq_{circ} of a circular permutation of size $n + 1$ to obtain a sorting sequence seq_{lin} of a linear permutation, such that both sequences have the same weight. We now have to show that seq_{lin} is optimal for π_{lin} if seq_{circ} is optimal for π_{circ}. For showing this, let us assume that seq_{lin} is optimal. Now, we generate a sequence seq_{circ} for sorting π_{circ} out of this sequence. Any operation of seq_{lin} is a valid operation for sorting the circular permutation, except revrevs (in the circular problem, revrevs are not allowed), which can be replaced by equivalent transreversals. It

$r(8,3)\pi = (-9\ -8\ +3\ +4\ +5\ +6\ +7\ -2\ -1\ -10)$ $r(3,8)\pi = (+1\ +2\ -7\ -6\ -5\ -4\ -3\ +8\ +9\ +10)$	$-A\ +B\ -A$ $+A\ -B\ +A$
$t(9,3,6)\pi = (+3\ +4\ +5\ +9\ +10\ +1\ +2\ +6\ +7\ +8)$ $t(3,6,9)\pi = (+1\ +2\ +6\ +7\ +8\ +3\ +4\ +5\ +9\ +10)$	$+B\ +A\ +C$ $+A\ +C\ +B\ +A$
$tr(9,3,6)\pi = (+3\ +4\ +5\ -2\ -1\ -10\ -9\ +6\ +7\ +8)$ $rr(3,6,9)\pi = (+1\ +2\ -5\ -4\ -3\ -8\ -7\ -6\ +9\ +10)$	$+B\ +A\ +C$ $+A\ -B\ -C\ +A$

Figure 2.3: Any operation that operates on a given element π_x can be replaced by an equivalent operation that does not act on π_x. On the left side, there are examples for replacing operations that act on π_1. In each block, both operations are equivalent. For all examples, the source permutation π is the identity permutation of size 10. On the right side, the view is simplified to segments. The order of the segments can be seen, and the sign indicates if a segment is inverted (-) or not (+). We work with circular permutations but as we can write them only linearly, elements at the beginning and the end of the permutation may belong to the same segment. This we indicate by writing the segment at the beginning and at the end (e.g $+A\ +C\ +B\ +A$).

is also easy to see that seq_{lin} sorts the circular permutation correctly, the dummy element 0 stays untouched until the permutation is sorted around it. Therefore, we can transform seq_{lin} into a sequence for circular permutations with the same weight. This shows that the weighted distance is the same for both π_{lin} and π_{circ}, and our transformation from seq_{circ} to seq_{lin} gives again an optimal sequence.
With this equivalence shown, we will now focus our sight on circular permutations. In the following, if we speak of a permutation, we mean a signed circular permutation.

2.3 The reality-desire diagram

The reality-desire diagram is a graph that helps us to analyse the permutation. It is a circular variation of the breakpoint graph first described in [BP96], and the representation we use is the same as in [SM97].
The main idea of the reality-desire diagram is to draw a graph whose edges represent the current neighbourhood relations of the elements in the permutation, and the desired neighbourhood relations (these are the neighbourhood relations of the sorted permutation). These neighbourhood relations must also take into account the sign of an element. In order to achieve this, we can view each element as a block with a start node and an end node. If an element is labelled with plus, the block is positively oriented, i.e. that the start node comes before the end node. Otherwise, the block is negatively oriented, i.e. the end node

Figure 2.4: Transforming each element into a block with a start node and an end node. In this example, x is labelled by plus, so the start node (marked with -) comes before the end node (marked with $+$). y is labelled by minus, so the end node comes before the start node.

comes before the start node. We will mark the start node of the block with a minus, and the end node with a plus (see also Figure 2.4). In the following, we will call the first of these nodes (this is the start node if π_i is labelled by plus, otherwise it is the end node) the *left node* of π_i, and the second the *right node*. For the current neighbourhood relations, it is easy to see that the right node of π_i is adjacent to the left node of π_{i+1}. In the sorted permutation, the end node of each element x must be adjacent to the start node of the element $x+1$. Now, we are ready to define the reality-desire diagram:

Definition 14. *The **reality-desire diagram** of a permutation π is the graph which we get by the following construction:*

- *First, we write the permutation counterclockwise on a circle.*
- *Each element π_i is replaced by the two nodes $-|\pi_i|$ (the start node) and $+|\pi_i|$ (the end node). If π_i was labelled by plus, the start node comes counterclockwise before the end node, otherwise the end node comes before the start node. Let us call the first of these nodes the left node of π_i, and the second the right node.*
- *For each element π_i, a **reality-edge** is added from the right node of π_i to the left node of π_{i+1} (indices are cyclic). Reality-edges are always drawn on the boundary of the circle.*
- *A desire-edge is added from the node $+x$ to the node $-(x+1)$ for each x from 1 to n (indices are cyclic). In the following, desire-edges will also be called **chords**. Chords are always drawn through the inside of the circle.*

Writing the permutation on a circle is the most convenient way to represent the neighbourhood relations of a circular permutation. By rawing the reality-edges on the edge of the circle and desire-edge through its inside, one can well distinguish between them. Additionally, in this representation, it is well-defined wether two chords do intersect, a fact that we will use later. An example of a reality-desire diagram can be found in Figure 2.5. We sometimes speak of elements in the reality-desire diagram. Although we do not draw them in the diagram, the position of an element x in the diagram is the position of the two nodes $+x$ and $-x$. If we say an element or a reality-edge precedes another element or reality-edge in the reality-desire diagram, we mean it precedes the other element counterclockwise.

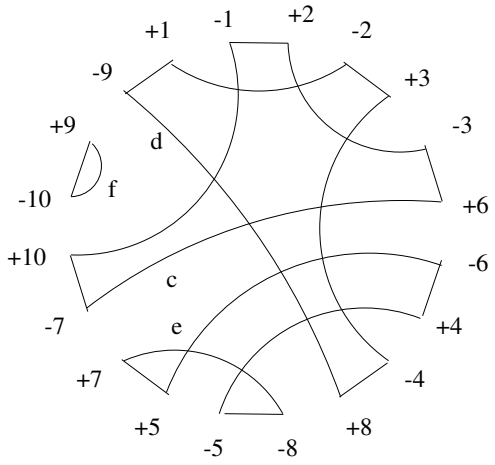


Figure 2.5: The reality-desire diagram of $\pi = (+1 + 9 + 10 + 7 - 5 + 8 + 4 + 6 + 3 + 2)$. The diagram decomposes into four cycles c, d, e, and f. The first three of these cycles are 3-cycles, and cycle f is an adjacency. Note that also in the permutation π, $\pi_2 = +9$ and $\pi_3 = +10$ form an adjacency.

As each node in the diagram has a degree of 2, it is easy to see that the diagram decomposes into cycles. Reality-edges represent the current neighbourhood relations, and desire-edges the neighbourhood relations of the sorted permutation. Therefore a reality-edge and a desire-edge connect the same two nodes in the reality-desire diagram if and only if the elements that induced these nodes form an adjacency.

Definition 15. *Let c be a cycle in a reality-desire diagram. Then the **length** l of a cycle is the number of reality-edges in the cycle, and c is called an **l-cycle**. If l is even, c is called an **even cycle**, otherwise it is called an **odd cycle**. If $l = 1$, we also say c is an **adjacency**.*

Definition 16. *Let π be a permutation and let $RDD(\pi)$ be its reality-desire diagram. Then $c(\pi)$ is the number of cycles, $c_{odd}(\pi)$ is the number of odd cycles and $c_{even}(\pi)$ is the number of even cycles in $RDD(\pi)$. If it is clear to which permutation we refer, we just write c, c_{odd} and c_{even}.*

Lemma 17. *Let π be a permutation with n elements. Then $c(\pi) = n$ if π is the identity permutation or its reflection. Otherwise, $c(\pi) \leq n - 1$.*

Proof. If π is the identity permutation or its reflection, π contains no breakpoints, and therefore all cycles in its reality-desire diagram have a length of 1. The sum of the lengths of the cycles must be n (as the diagram has n reality-edges), so $c(\pi) = n$. If π is neither the identity permutation nor its reflection, it contains at least one breakpoint, and therefore contains at least one cycle with length ≥ 2. As the sum of the cycle lengths is n, and each other cycle has a length of at least 1, the number of cycles cannot exceed $n - 1$. □

2.4 The effects of operations on the reality-desire diagram

The reality-desire diagram is a very useful tool to analyze the effects of operations on the permutation. Each operation changes the permutation, and therefore the reality-desire diagram. As the number of cycles c is maximized for the identity permutation or its reverse, it is an obvious strategy to try to increase c as fast as possible. However, the change of c per operation is bounded:

Lemma 18. *[GPS99] Let c be the number of cycles in the reality-desire diagram of a permutation before an operation, and let c' be the number of cycles after the operation. Then for any operation,*

$$\Delta c = c' - c \in [-2; +2]$$

Proof. Any operation splits the permutation at most at three positions. In the reality-desire diagram, at most three reality-edges will be split (we say the operation acts on these reality-edges), desire-edges always remain unchanged. So any operation can affect at most three cycles, altering them to some new cycles. As any operation is reversible, we also can get at most three new cycles, so the difference is in $[-2; +2]$. □

As a reversal splits a permutation at only two positions, a reversal cannot change the number of cycles by two:

Corollary 19. *Let c be the number of cycles in the reality-desire diagram of a permutation before a reversal, and let c' be the number of cycles after the reversal. Then for any reversal,*

$$\Delta c = c' - c \in [-1; +1]$$

Later it will be important to distinguish between even and odd cycles. Therefore, we will also proof a lemma about Δc_{odd}:

Lemma 20. *[GPS99] Let c_{odd} be the number of odd cycles in the reality-desire diagram of a permutation before an operation, and let c'_{odd} be the number of odd cycles after the operation. Then for any operation,*

$$\Delta c_{odd} = c'_{odd} - c_{odd} \in \{-2; 0; +2\}$$

Proof. Analogous to the proof of Lemma 18, our operation alters one to three cycles to some new cycles. Note that there is also the possibility that we have no odd cycle in the beginning or in the end, so $\Delta c_{odd} \in [-3; +3]$. Now let l be the sum of the cycle lengths:

$$l = \sum_c length(c)$$

If Δc_{odd} is an odd number, l will change from an even to an odd number or vice versa. This means that the permutation would change its size, what is obviously not possible. Therefore, $\Delta c_{odd} \in \{-2; 0; +2\}$. □

Note that a reversal can change the total number of cycles by at most one, but it can change the number of odd cycles by two: a reversal can split an even cycle into two odd cycles.
Another important observation is that a transposition or transreversal that acts on the reality-edges of exactly two cycles does not change the total number of cycles: A segment of one cycle is cut off, and inserted into the other cycle.
A complete list of possible changes in the reality-desire diagram, depending on the involved cycles, can be seen in Table 2.1. Additionally the change of a score σ is listed there. This score will be introduced in the next section.

2.5 A lower bound

Now we are ready to prove a lower bound for the weighted transposition and reversal distance. As we have seen in section 2.4, an operation can increase the number of odd cycles in the reality-desire diagram at most by two (see Lemma 20). The identity permutation has n odd cycles, so we need at least $\frac{n-c_{odd}}{2}$ operations to sort a permutation. Therefore our first estimation for the distance is

$$d_w \geq \frac{n - c_{odd}}{2} \min\{w_t, w_r\}$$

Now let us assume that $w_r \leq w_t \leq 2w_r$. Taking a look at table 2.1, we see that there are five possible moves that increase c_{odd} by two:

1. A reversal can split one even cycle into two odd cycles.
2. A transposition or a transreversal can split one odd cycle into three odd cycles.
3. A transposition or a transreversal can split one even cycle into two odd cycles and one even cycle.
4. A transposition or a transreversal can split one even cycle into two odd cycles.
5. A transposition or a transreversal can transform two even cycles into two odd cycles.

The cheapest of these operations is the reversal, but it requires an even cycle. Thus we should not only consider the number of odd cycles but also the number of even cycles in the reality-desire diagram. The score we will introduce is a linear combination of c_{odd} and c_{even}, with both coefficients positive. Therefore, we only have to focus on the first three of the operations mentioned above; the last two have a cost of w_t and decrement c_{even}, so they are certainly not as good as the second operation. We will now assume that a sequence where any move is one of the first three of the moves mentioned above is optimal (after defining the score and looking at the effects of the operations on the score, we will see that this assumption is correct). Therefore, these moves should have an equal effect on the score per weight of the move. This leads to the following definition of the score:

	e	o	ee	eo	oo	eee	eeo	eoo	ooo
r	e 0	o 0	e $2\frac{wr}{wt}-2$	o $2\frac{w_r}{w_t}-2$	e $-2\frac{w_r}{w_t}$				
	ee $2-2\frac{w_r}{w_t}$	eo $2-2\frac{w_r}{w_t}$							
	oo $2\frac{w_r}{w_t}$								
t/tr	e 0	o 0	ee 0	eo 0	ee $2-4\frac{w_r}{w_t}$	e $4\frac{w_r}{w_t}-4$	o $4\frac{w_r}{w_t}-4$	e -2	o -2
	ee $2-2\frac{w_r}{w_t}$	eo $2-2\frac{w_r}{w_t}$	oo $4\frac{w_r}{w_t}-2$		oo 0				
	oo $2\frac{w_r}{w_t}$	eeo $4-4\frac{w_r}{w_t}$							
	eee $4-4\frac{w_r}{w_t}$	ooo 2							
	eoo 2								

Table 2.1: This table lists all possibilities how an operation can change the cycles of a reality-desire diagram. The first row contains the cycles involved (where e is an even cycle and o is an odd cycle). The other rows give the possible resulting cycles, and the differences of the score σ induced by the operation. The first column determines whether the operation is a reversal (r) or a transposition or transreversal (t/tr).

Definition 21. *The score σ of a permutation is defined by*

$$\sigma = c_{odd} + (2 - \frac{2w_r}{w_t})c_{even}$$

where c_{odd} is the number of odd cycles and c_{even} is the number of even cycles in the reality-desire diagram of the permutation. The change of the score due to an operation or a sequence of operations is called $\Delta\sigma$.

The following lemma shows that the score is maximized for the identity permutation. This will be very useful for developing our approximation algorithm.

Lemma 22. *For any permutation π and weights w_r, w_t with $w_r \leq w_t \leq 2w_r$,*

- *$\sigma = n$ if π is the identity permutation or its reflection*
- *$\sigma \leq n - 1$ if π is neither the identity permutation nor its reflection*

Proof. If π is the identity permutation or its reflection, the reality-desire diagram consists of n adjacencies, so $\sigma = n$. Otherwise, the reality-desire diagram has at least one cycle with length ≥ 2. Therefore it has at most $n - 1$ cycles. An odd cycle adds 1 to the score, an even cycle adds $2 - \frac{2w_r}{w_t}$. With $w_t \leq 2w_r$, $2 - \frac{2w_r}{w_t} \leq 1$. Therefore, $\sigma \leq n - 1$. □

Now, we will use this score to show a better lower bound:

Theorem 23. *For each permutation and weights w_t, w_r with $w_r \leq w_t \leq 2w_r$, a lower bound for the weighted distance is*

$$d_w \geq c_{even}w_r + (\frac{n - c_{odd}}{2} - c_{even})w_t$$

Proof. In the beginning, the reality-desire diagram of our permutation has c_{odd} odd cycles and c_{even} even cycles. In the identity permutation, we have n odd cycles and no even cycle. If we have a look at the tuple (c_{odd}, c_{even}), the task is to change this tuple to $(n, 0)$ by changing the permutation, using reversals, transpositions and transreversals. It is hard to see a minimum distance between these two tuples, so we introduce a score σ as a linear combination of c_{odd} and c_{even}. If we use Definition 21, we can assure that for each operation, the change of the score per weight is at most $\frac{2}{w_t}$ (this can easily be verified by having a look at Table 2.1 and inserting the bounds for w_r and w_t). Our permutation has a score of $c_{odd} + (2 - \frac{2w_r}{w_t})c_{even}$ in the beginning, and the identity permutation has a score of n. The difference in the score is

$$n - (c_{odd} + (2 - \frac{2w_r}{w_t})c_{even})$$

Now we can estimate the weighted distance:

$$\begin{aligned} d_w &\geq (n - (c_{odd} + (2 - \frac{2w_r}{w_t})c_{even}))\frac{w_t}{2} \\ &= (n - c_{odd})\frac{w_t}{2} - (2 - \frac{2w_r}{w_t})c_{even}\frac{w_t}{2} \\ &= c_{even}w_r + (\frac{n - c_{odd}}{2} - c_{even})w_t \end{aligned}$$

□

2.6 Transforming into simple permutations

The algorithm which we develop in the next chapter will work with a case analysis on the cycles of the reality-desire diagram. However, this case analysis is rather complicated if we have to work with arbitrary cycle lengths. Therefore we will restrict cycle lengths to a maximum of 3. In this section we will show how to transform any permutation π into a permutation $\tilde{\pi}$ with restricted cycle lengths, and how we can use a sorting sequence of $\tilde{\pi}$ to obtain a sorting of π. Finally, we will prove that the sorting of π is a 1.5-approximation if the sorting of $\tilde{\pi}$ is a 1.5-approximation.

Definition 24. *A cycle in a reality-desire diagram is called a* ***short cycle*** *if its length is at most 3. Otherwise, it is called a* ***long cycle****.*

Definition 25. *A permutation is called a* ***simple permutation*** *if all cycles in its reality-desire diagram are short cycles.*

We will now provide an algorithm that transforms any long cycle into a short cycle by adding elements to the permutation. The algorithm has been described in [LX01]. However, we will give here a more constructive presentation of the algorithm.
Let c be a k-cycle with $k > 3$. Let $r_1, r_2, \ldots, r_k$ be the reality-edges of c in the order we pass them if we walk along the cycle in the reality-desire diagram. The starting edge is arbitrary. However, we will assume that we pass r_1 counterclockwise. Let v be the new element we want to insert into the permutation, and let x be the element that precedes r_1 in the reality-desire diagram. If x is labelled by plus (so r_1 starts at the node $+x$), the value of v is between x and $x+1$ (for the moment, we do not care that this value is not integer). Otherwise (so r_1 starts at node $-x$), the value of v is between $x-1$ and x. The effect of choosing this value is that the desire-edge from r_1 no longer goes to r_k, but to a reality-edge where the new element v will be inserted. Let y and z be the two elements beside r_3. We insert v between y and z. As a result of this, r_3 will be split into two reality-edges r_{3a} (that lies between y and v) and r_{3b} (that lies between v and z). Depending on the sign we choose for v, the desire-edge from r_1 will be connected either with r_{3a} or with r_{3b}. If r_{3a} is connected by a desire-edge with r_2, we choose the sign so that the desire-edge goes from r_1 to r_{3a}, obtaining a 3-cycle with the reality-edges r_1, r_2, and r_{3a}. Otherwise, we choose the sign so that the desire-edge goes from r_1 to r_{3b}, obtaining a 3-cycle with the reality-edges r_1, r_2, and r_{3b}. In both cases, the other of the reality-edges r_{3a} and r_{3b} will be connected by a desire-edge to r_k, obtaining a $k-2$-cycle with reality-edges r_{3a} (respectively r_3b), $r_4, \ldots, r_k$. Now, we can increase any element greater than v by one, and make v an integer value. We can repeat this until the whole permutation only contains short cycles. A single transformation step is illustrated in Figure 2.6. As the transformation from π into the simple permutation $\tilde{\pi}$ was done by just padding elements to the permutation, it is easy to see how we can use a sorting sequence $o\tilde{p}_1, o\tilde{p}_1, \ldots, o\tilde{p}_n$ of $\tilde{\pi}$ to get a sorting sequence $op_1, op_2, \ldots, op_n$ of π: we use the same sequence as we used for $\tilde{\pi}$, and ignore the padded elements. In some cases, the operations can change due to the padded elements:

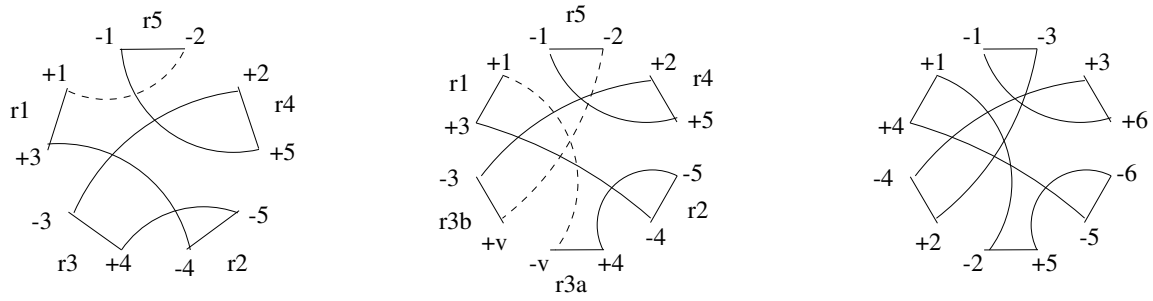


Figure 2.6: Transformation of a 5-cycle into two 3-cycles. The first picture is the reality-desire diagram of the original permutation. The reality-edges are labelled, and the element x is the element $+1$ in the permutation. The dashed desire-edge will be replaced by two new desire-edges. As x is labelled by plus, the value of v is between 1 and 2. In the second picture, the new element is inserted, and the reality-edge r_3 is split into two edges r_{3a} and r_{3b}. Note that the dashed desire-edge of the first picture is replaced by two other desire-edges, and the cycle is already split into two 3-cycles. The right picture is after the padding, when the elements have been shifted. The permutation $\pi = (+1\ -3\ -4\ +5\ -2)$ has been transformed into $\tilde{\pi} = (+1\ -4\ -2\ -5\ +6\ -3)$.

- If $o\tilde{p}_i = t(i,j,k)$ and $i = j$ or $i = k$ or $j = k$, the transposition just exchanges two segments. As we work with circular permutations, this has no effect on the result, so we can omit this operation.
- If $o\tilde{p}_i = tr(i,j,k)$ and $j = k$ or $k = i$, the transposition of the transreversal is no longer necessary. The operation is reduced to the reversal $op_i = r(i,j)$.
- If $o\tilde{p}_i = tr(i,j,k)$ and $i = j$, the transreversal just exchanges two segments, and we can omit this operation.
- If $o\tilde{p}_i = r(i,j)$ and $i = j$, this reversal does nothing, so it can be omitted.

Note that all of these changes reduce the weight of the operation, so that the weight of the sorting sequence of π is equal to or less than the sorting sequence of $\tilde{\pi}$.

Theorem 26. *Let π be a permutation and let $\tilde{\pi}$ be the simple permutation obtained by the algorithm above applied on π. Let*

$$b = c_{even}(\pi)w_r + (\frac{n - c_{odd}(\pi)}{2} - c_{even}(\pi))w_t$$

be the lower bound for the weighted distance of π, and let

$$\tilde{b} = c_{even}(\tilde{\pi})w_r + (\frac{n - c_{odd}(\tilde{\pi})}{2} - c_{even}(\tilde{\pi}))w_t$$

be the lower bound for the weighted distance of $\tilde{\pi}$ (as proven in Theorem 23). Then a sorting sequence of $\tilde{\pi}$ with weight $\tilde{w} \leq 1.5\tilde{b}$ can be transformed into a sorting sequence for π with weight $w \leq 1.5b$ by simply ignoring the padded elements in $\tilde{\pi}$.

Proof. Every transformation step adds one element to the permutation and splits a k-cycle (with $k > 3$) into a 3-cycle and a (k-2)-cycle. If k is even, the result of the split is an even and an odd cycle. If k is odd, the result of the split are two odd cycles. In both cases, c_{odd} is increased by 1, whereas c_{even} remains unchanged. Therefore, the bound $c_{even}w_r + (\frac{n-c_{odd}}{2} - c_{even})w_t$ is not changed by the transformation, and $b = \tilde{b}$. As ignoring the padded elements cannot increase the weight of an operation, we get

$$d_w \leq \tilde{d_w} \leq 1.5\tilde{b} = 1.5b$$

□

We have now seen that sorting any permutation can be reduced easily to sorting a simple permutation. Therefore we will only consider simple permutations in the rest of this work.

2.7 Some observations about cycles

As we have seen in the previous sections, a sorting sequence is optimal if it increases the number of odd cycles by two in every operation. Unfortunately, this is not possible in most cases. We will now examine the cycles and develop some lemmata to get a better characterisation of the cycles. First, we start with some definitions.

Definition 27. *A **configuration** is a subset of the cycles of the reality-desire diagram of a permutation.*

An example for a configuration can be seen in Figure 2.7. Note that each reality-desire diagram can be regarded as a configuration. This is not valid vice versa: there exist

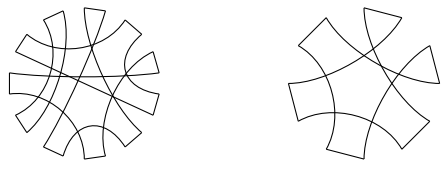


Figure 2.7: The cycles of a reality-desire diagram, and a subset of its cycles as configuration. The reality-desire diagram on the left belongs to the permutation $\pi = (+1\ -4\ -10\ -5\ +2\ -6\ -7\ -11\ +9\ +8\ -3)$; there exists no permutation with the configuration on the right picture being its reality-desire diagram.

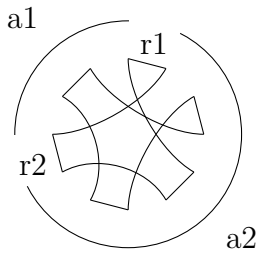


Figure 2.8: The reality edges r1 and r2 induce two arcs a1 and a2.

configurations which are not the reality-desire diagram of any permutation. The concept of the configuration is very useful as it helps us to examine the effect of different operations on only a few cycles.

Definition 28. *A **position** in a configuration is a position between two consecutive reality-edges in the configuration.*

If the configuration contains all cycles of the reality-desire diagram, a position in the configuration is equivalent to an element of the reality-desire diagram. If the configuration does not contain all cycles, then there are also some reality-edges that do not appear in the configuration, and several elements of the reality-desire diagram collapse to one position. For example, in Figure 2.7, the elements $+1$, -4, -10 collapse to one position in the configuration.

Definition 29. *An **arc** is a series of consecutive positions of a configuration, bounded by two reality-edges r_1 and r_2.*

Note that there are always two arcs between two reality-edges r_1 and r_2: One goes from r_1 counterclockwise to r_2, the other goes from r_1 clockwise to r_2. To make the definition well-defined, we always take the arc that goes counterclockwise from r_1 to r_2. An example can be seen in Figure 2.8.

Definition 30. *Two chords d_1 and d_2 are **intersecting** if they intersect in the reality-desire diagram. More precisely, the endpoints of the chords must alternate along the cycle in the configuration. Two cycles c_1 and c_2 are intersecting if a chord of c_1 intersects with a chord of c_2.*

Definition 31. *Two cycles c_1 and c_2 are interleaving if in the reality-desire diagram, the reality-edges of c_1 and c_2 are alternating. More precisely, between each pair of reality-edges of c_1 lies a reality-edge of c_2, and vice versa.*

Note that interleaving cycles must have the same size. Examples for intersections and interleaving cycles can be found in Figure 2.9.

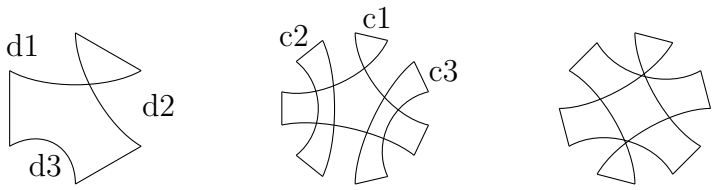


Figure 2.9: Examples for intersections. In the first picture, the chords d1 and d2 are intersecting. d3 does not intersect any chord. In the second picture, the cycle c1 is intersecting the cycles c2 and c3. c2 and c3 do not intersect each other. The last picture is an example for two interleaving cycles.

Definition 32. *An operation is called x_y-**move** if it increases the number of cycles by x, and the operation type is y (where $y = r$ is a reversal, $y = t$ a transposition, and $y = tr$ a transreversal).*

For example, a transposition that splits one cycle into three is a 2_t-move. A reversal that merges two cycles is a -1_r-move.

Definition 33. *A $m_1 m_2 \dots m_n$-sequence is a sequence of n consecutive operations, where the first is an m_1-move, the second an m_2-move and so on.*

For example, a $0_r 2_t 2_t$-sequence is a reversal which does not change the number of cycles, followed by two transpositions each of them increasing the number of cycles by two.

Definition 34. *A cycle c is called r-oriented if there exists a 1_r-move that acts on two of the reality-edges of c. Otherwise the cycle is called r-unoriented.*
A cycle c is called t-oriented if there exists a 2_t-move or a 2_{tr}-move that acts on three of the reality-edges of c. Otherwise the cycle is called t-unoriented.

This is the general definition of the orientation of cycles. As we will work only on simple permutations, we will have a closer look at 2-cycles and 3-cycles. With our definition of the score σ (see Definition 21), optimal moves are 1_r-moves on 2-cycles and 2_t-moves or 2_{tr}-moves on 3-cycles. These moves split the cycles into adjacencies. The gain of the score is $\Delta\sigma = \frac{2}{w_t}$. Any other move on 2- and 3-cycles yields a smaller increment of the score. For a better characterisation of the cycles, we need the definition of twists:

Definition 35. *A reality-edge is called **twisted** if its adjacent chords are intersecting. A cycle is called **k-twisted** if k of its reality-edges are twisted. For $k = 0$, we also say that the cycle is nontwisted.*

Definition 36. *A chord is called **twisted** if it intersects with another chord that belongs to the same cycle. Otherwise, it is called **nontwisted**.*

For example, a 1-twisted 3-cycle has two twisted chords, and one nontwisted chord (see also Figure 2.10).

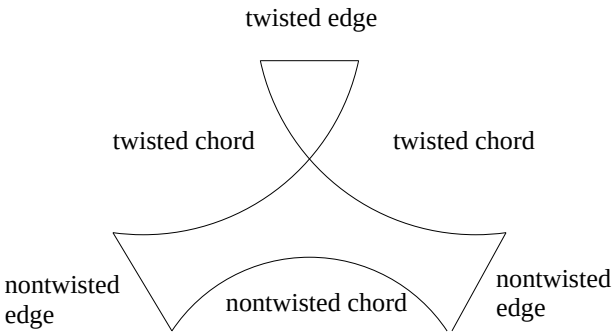


Figure 2.10: An example for twisted reality-edges and twisted chords.

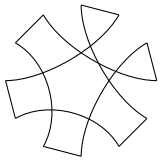


Figure 2.11: A 1-twisted pair.

Definition 37. *Two interleaving 1-twisted 3-cycles form a 1-twisted pair if their twisted reality-edges are consecutive in the configuration (see Figure 2.11).*

Lemma 38. *A 2-cycle is r-oriented iff it is 2-twisted.*

Proof. There are only two possible configurations of a 2-cycle, as illustrated in Figure 2.12. If the 2-cycle is 2-twisted, a reversal that acts on its reality-edges splits the cycle into two adjacencies. If the cycle is 0-twisted, a reversal that acts on its reality-edges does not change the shape of the cycle in the configuration. □

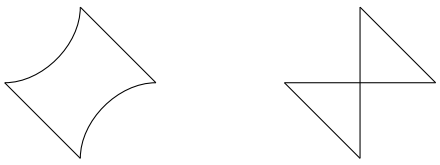


Figure 2.12: The two possible configurations of a 2-cycle. a) is nontwisted and r-unoriented, b) is 2-twisted and r-oriented.

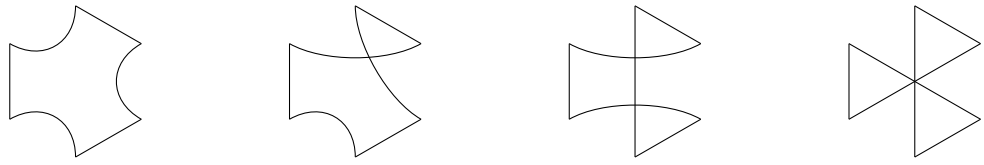


Figure 2.13: The possible configurations of a 3-cycle. a) and b) are t-unoriented, c) and d) are t-oriented.

Lemma 39. *[HS04] A 3-cycle is t-oriented iff it is 2-twisted or 3-twisted.*

Proof. There are four possible configurations of a 3-cycle, as illustrated in Figure 2.13. If the cycle is 3-twisted, a transposition that acts on its reality-edges is a 2_t-move. If the cycle is 2-twisted, a transreversal that acts on its reality-edges and reverses the segment between the twisted edges is a 2_{tr}-move. For 0-twisted and 1-twisted 3-cycles, there exists no move that splits the cycle into adjacencies. □

Removing oriented cycles results in a maximal gain of the score by a minimal weight. However, such a move can change the orientation of other cycles, making them non-oriented. We will now provide two lemmata about moves that keep some other cycles oriented. Both lemmata have been proven in [HS04].

Lemma 40. *Let π be a permutation and let c and d be two intersecting, non-interleaving cycles in its reality-desire-diagram with the following properties:*

- *c is a 2-twisted 3-cycle, d is a 1-twisted 3-cycle*
- *none of the arcs induced by the two twisted edges of c contains both nontwisted edges of d*

Then there exists a $2_{tr}2_{tr}$-sequence.

Proof. The transreversal that eliminates c either inverts one untwisted edge of d, or one untwisted edge and the twisted edge of d. In both cases, d becomes 2-twisted, and a second transreversal will eliminate it. Two examples can be seen in Figure 2.14. □

Lemma 41. *Let c and d be two 2-twisted, interleaving 3-cycles. These cycles admit a $2_{tr}2_{tr}$-sequence iff four of their twisted edges are consecutive in the configuration.*

Proof. There are two possible configurations for two interleaving 2-twisted 3-cycles, as illustrated in Figure 2.15. The first configuration fulfils the precondition, and admits a $2_{tr}2_{tr}$-sequence. The second configuration does not fulfil the precondition. Eliminating one 3-cycle makes the other nontwisted, so it cannot be eliminated with the next move. □

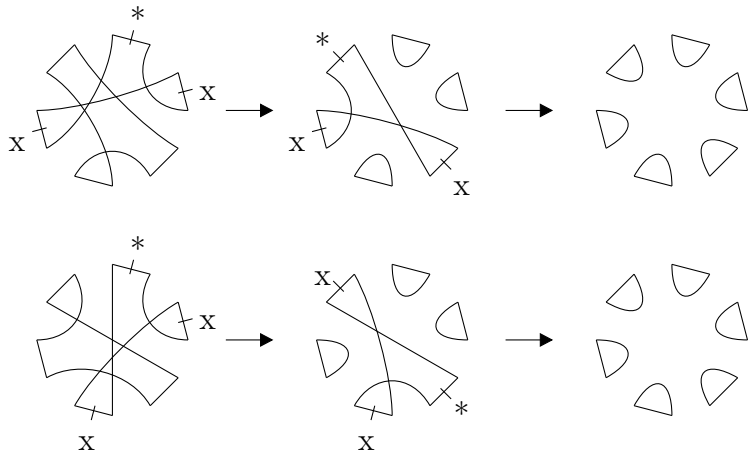


Figure 2.14: A 2-twisted 3-cycle and a 1-twisted 3-cycle are intersecting, as described in Lemma 40. Each transposition acts on the edges marked with * or x. The part between two x will be inverted.

As we cannot eliminate unoriented cycles directly, we first need a move that orientates the cycle. In most cases, this is a move that acts on the reality edges of another cycle that intersects with the unoriented cycle. We will now show that unoriented cycles must intersect with other cycles. For this proof, we first need the definition of the complement graph ([HS04]):

Definition 42. *The **complement graph** of a reality-desire diagram is defined as follows:*

- *The nodes and desire-edges are the same as in the reality-desire diagram.*
- *Adjacent reality-edges will be connected by new edges.*
- *The original reality-edges will be removed.*

An example for a complement graph can be seen in Figure 2.16. By construction, there is a new edge for each element i of a permutation, and the desire-edges connect these edges with the edges belonging to $i+1$ and $i-1$. Thus the complement graph consists of exactly one cycle.

Lemma 43. *If a cycle c has a nontwisted chord, then there exists another cycle d that intersects with the nontwisted chord of c.*

Proof. Let e be the nontwisted chord of c, and let a be the arc that lies between the reality-edges adjacent to e. Suppose that there is no other chord that intersects with e. The complement graph has at least two cycles: One with the chord e and all edges in the arc a, and the other with the edges outside of a. This is a contradiction. □

Definition 44. *Two arcs a_1, a_2 are called **adjacent** if the endpoints of a_1 are connected with the endpoints of a_2 by two chords.*

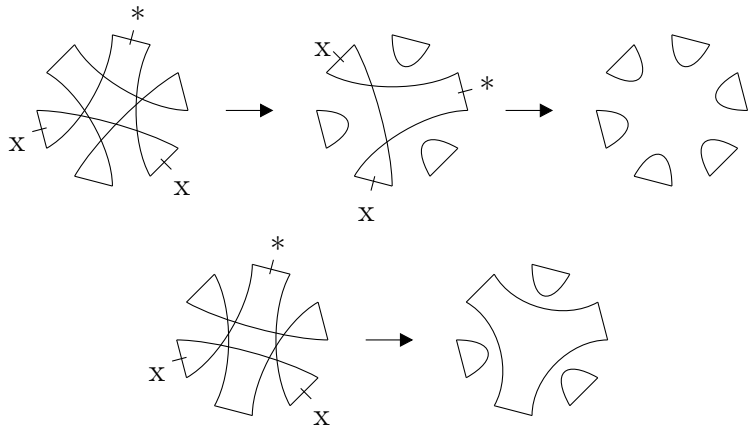


Figure 2.15: The two possible configurations for two interleaving 2-twisted 3-cycles. The first configuration admits a $2_{tr}2_{tr}$-sequence. For the second configuration, such a sequence is not possible.

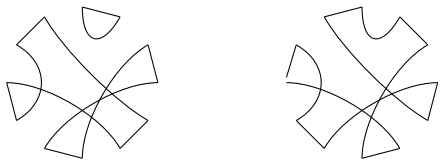


Figure 2.16: The reality-desire diagram of the permutation $\pi = (+1\ +4\ -3\ -5\ +2\ +6)$, and its complement graph.

Examples for adjacent arcs can be seen in Figure 2.17.

Lemma 45. *Let a_1 and a_2 be two disjoint (i.e. they have no common positions) adjacent arcs in a configuration, so that there is at least one position in the configuration that is not covered by a_1 or a_2. Then there exists a cycle in the reality-desire diagram with one reality-edge in a_1 or a_2, and another reality-edge that is neither in a_1 nor in a_2.*

Proof. This lemma has been proven in [HS04]. Suppose that no such cycle exists. Then in the complement graph, there is a cycle that only contains edges inside the arcs a_1 and a_2. As there is another position in the configuration not covered by a_1 or a_2, this cycle does not contain all edges, so there must be a second cycle. This is a contradiction. □

It is sometimes of interest to see the effect of a single transposition on a t-unoriented cycle. The following lemma describes how a transposition can orient such a cycle.

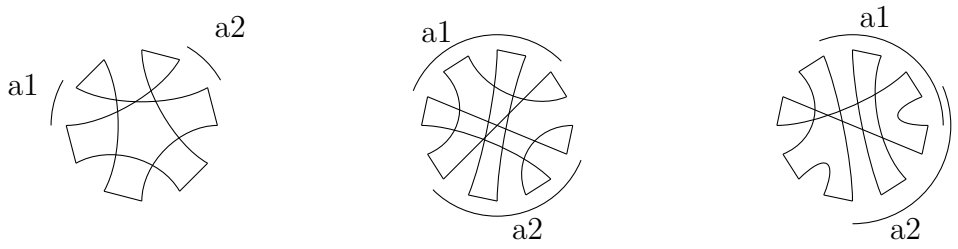


Figure 2.17: Examples for adjacent arcs. In each picture, the arcs a_1 and a_2 are adjacent. Note that adjacent arcs can overlap (right picture).

Figure 2.18: A transposition that acts on three reality-edges in different arcs induced by a t-unoriented cycle c orientates c.

Lemma 46. *Let c be a t-unoriented 3-cycle and let $a1$, $a2$ and $a3$ be the three arcs induced by the reality-edges of c. If a transposition acts on three reality-edges, so that one of them is in $a1$, one in $a2$, and one in $a3$, then c becomes t-oriented. More precisely, if c was nontwisted, it becomes 3-twisted, and if c was 1-twisted, it becomes 2-twisted.*

Proof. The transposition just moves a segment containing one of the reality-edges between the other reality-edges. The results of this move can be seen in Figure 2.18. □

Although our algorithm works on simple permutations, there are some cases where we have to build a 5-cycle. This 5-cycle will be split into a 3-cycle and two adjacencies in the following moves. We will now provide some lemmata that show when such a move is possible and how to construct such a 5-cycle.

Lemma 47. *Let c be a 5-cycle and let r_1 and r_2 be two reality-edges in c connected by a desire-edge. If we remove r_1 and r_2 and the adjacent desire-edges, close the remaining cycle with a new desire-edge, then we obtain a new cycle called c' (see Figure 2.19 for an example).*

- *There exists a 2_t-move that splits c into a 3-cycle and two adjacencies iff there exists a pair of reality-edges r_1, r_2, so that c' is a 3-twisted 3-cycle.*
- *There exists a 2_{tr}-move that splits c into a 3-cycle and two adjacencies iff there exists a pair of reality-edges r_1, r_2, so that c' is a 2-twisted 3-cycle.*

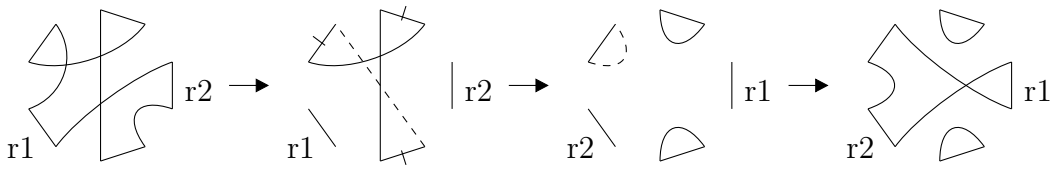


Figure 2.19: This 5-cycle allows a 2_t-move that splits the cycle into a 3-cycle and two adjacencies. First, we mark two reality-edges connected by a desire-edge with r_1 and r_2, remove them, and close the cycle by a new desire-edge (dashed line). Then, we perform the transposition (third picture). The last step is to reinsert the reality-edges r_1 and r_2.

Proof. An operation can only generate an adjacency if it acts on two reality-edges that are connected by a desire-edge. Therefore, to get two adjacencies, the operation must act on three reality-edges that are connected by desire-edges. We can now simulate every operation as follows: First, we remove the other two reality-edges (called r_1 and r_2) and their adjacent desire-edges, and close the cycle with a new desire-edge. This corresponds to c'. Now, we perform our operation on c'. As final step, we insert r_1 and r_2 into the cycle with the newly created desire-edge. An example can be seen in Figure 2.19.
If there exists a pair of reality-edges r_1 and r_2, so that c' is 3-twisted, a transposition will split c' into three adjacencies. If we insert r_1 and r_2 into the resulting cycles, we get a 3-cycle and two adjacencies. If there conversely does not exist such pair of reality-edges, there is also no transposition that creates three adjacencies. Therefore, any transposition that acts on three reality-edges that are connected by desire-edges can split c into at most two cycles.
For the 2_{tr}-move, the argumentation is the same, except that a 2_{tr}-move on a 3-cycle requires a 2-twisted 3-cycle. □

Lemma 48. *Let c be a t-unoriented 5-cycle and let d be a 3-cycle, so that between each pair of reality-edges of d, there is a reality-edge of c. Then a transposition that acts on the reality-edges of d makes c t-oriented, and there is a second move that splits c into a 3-cycle and two adjacencies.*

Proof. Let a_1, a_2, and a_3 be the three arcs induced by the reality-edges of d. In one of these arcs (without loss of generality, in a_1) is only one reality-edge of c. We call this edge c_1. Let c_0 and c_2 be the reality-edges that are directly connected to c_1 by a desire-edge. If c_0 is in arc a_2 and c_2 is in arc a_3 or vice versa, c_0, c_1, and c_2 form a triplet of reality-edges that are connected by desire-edges and all of them are in a different arc induced by the reality-edges of d. If c_0 and c_2 are in the same arc (without loss of generality, in a_2), then one of them must be connected to a reality-edge in a_3 (otherwise, there would be no reality-edge in a_3, a contradiction to our precondition). Again, there is a triplet of reality-edges that are connected by desire-edges and all are in a different arc induced by the reality-edges of d. If we remove the remaining two reality-edges and the adjacent desire-edges, and close the

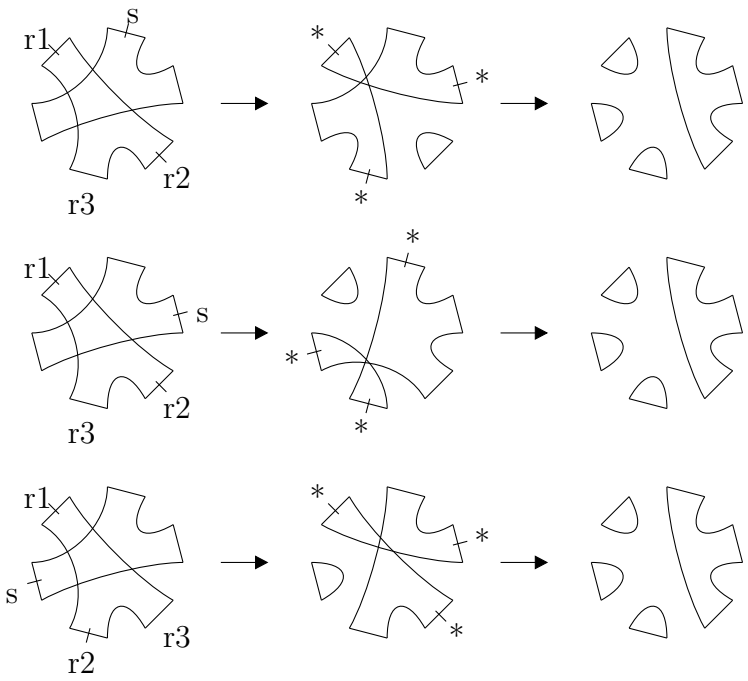


Figure 2.20: Depending on the position of s, there are three possibilities for choosing r_1, r_2, and r_3 so that the preconditions of Lemma 49 are fulfilled. In each case, a transposition that acts on r_1, r_2 and s transforms the cycles into a 5-cycle and an adjacency. The second transposition that transforms the 5-cycle into a 3-cycle and two adjacencies acts on the reality-edges marked with *.

cycle with a new desire-edge, we obtain a 3-cycle that must be t-unoriented according to Lemma 47. A transposition that acts on the reality-edges of d makes this 3-cycle t-oriented (see Lemma 46). Therefore, after the transposition, d fulfills the preconditions of Lemma 47 and can be split into a 3-cycle and two adjacencies in the next move. □

The next lemma shows how we can get an adjacency and a t-oriented 5-cycle out of two intersecting nontwisted 3-cycles, and therefore describes a 0_t2_t-move. It has been proven in [Har03].

Lemma 49. *Let c and d be two intersecting, non-interleaving nontwisted 3-cycles. Let r_1, r_2 and r_3 be the reality-edges of c and let s be any reality-edge of d. Let a_1 be the arc between r_1 and r_2, and a_2 the arc between r_3 and s. If a_1 and a_2 are overlapping, then a transposition that acts on the reality-edges r_1, r_2 and s transforms the cycles into a 5-cycle and an adjacency. A second transposition that transforms the 5-cycle into a 3-cycle and two adjacencies is possible.*

Proof. There are only three possible configurations of two nontwisted 3-cycles that fulfil the preconditions. For each configuration, the 0_t2_t-sequence is illustrated in Figure 2.20. □

Chapter 3

A 1.5-approximation

In this chapter, we provide a 1.5-approximation algorithm for sorting circular permutations by weighted transpositions and reversals. We first develop an algorithm that works on simple permutations. Then we use the results of section 2.6 to generalize it to arbitrary permutations. The algorithm is a 1.5-approximation for any constant weights w_t, w_r with $w_r \leq w_t \leq 2w_r$.

3.1 Algorithm overview

Given a simple permutation π of size n, the overall goal is to find a sorting sequence with weight w such that $w \leq 1.5d_w$. According to Theorem Theorem 23, we know that

$$\begin{aligned} d_w &\geq c_{even}w_r + (\frac{n - c_{odd}}{2} - c_{even})w_t \\ &= \frac{w_t}{2}(n - c_{odd} - 2c_{even} + 2c_{even}\frac{w_r}{w_t}) \\ &= \frac{w_t}{2}(n - \sigma) \end{aligned}$$

For any sorting sequence, $\Delta\sigma = n - \sigma$. Therefore, if the sorting sequence fulfills $\frac{\Delta\sigma}{w} \geq \frac{4}{3w_t}$, then

$$\begin{aligned} w &\leq \frac{3w_t}{4}\Delta\sigma \\ &= 1.5\frac{w_t}{2}(n - \sigma) \\ &\leq 1.5d_w \end{aligned}$$

We will compose the sorting sequence out of many short sequences, each fulfilling the condition $\frac{\Delta\sigma}{w} \geq \frac{4}{3w_t}$. It is easy to see that also the sorting sequence will fulfill the condition, and therefore provides a 1.5-approximation. The algorithm is working by searching such a short sequence, applying it to the permutation, and searching for the next short sequence, until the permutation is sorted. We will now discuss how we can find these short sequences.

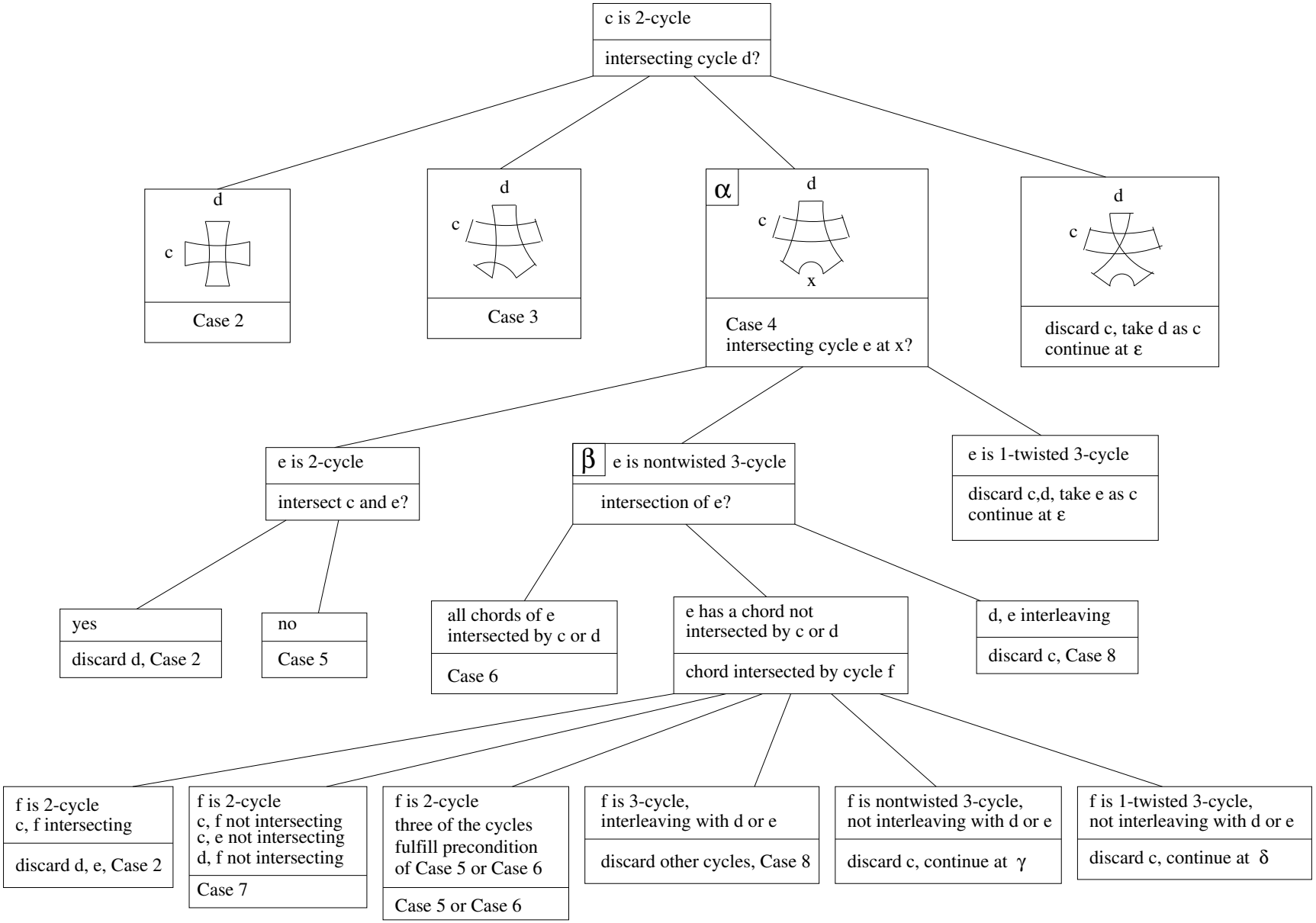


Table 3.1: The algorithm decision tree if we begin with an r-unoriented 2-cycle c. All cycles are considered to be r-unoriented 2-cycles or t-unoriented 3-cycles; otherwise the algorithm jumps directly to the trivial case (Case 1). Cross-references α and β can be found on this table, γ and δ on Table 3.2, ε, ζ and η on Table 3.3.

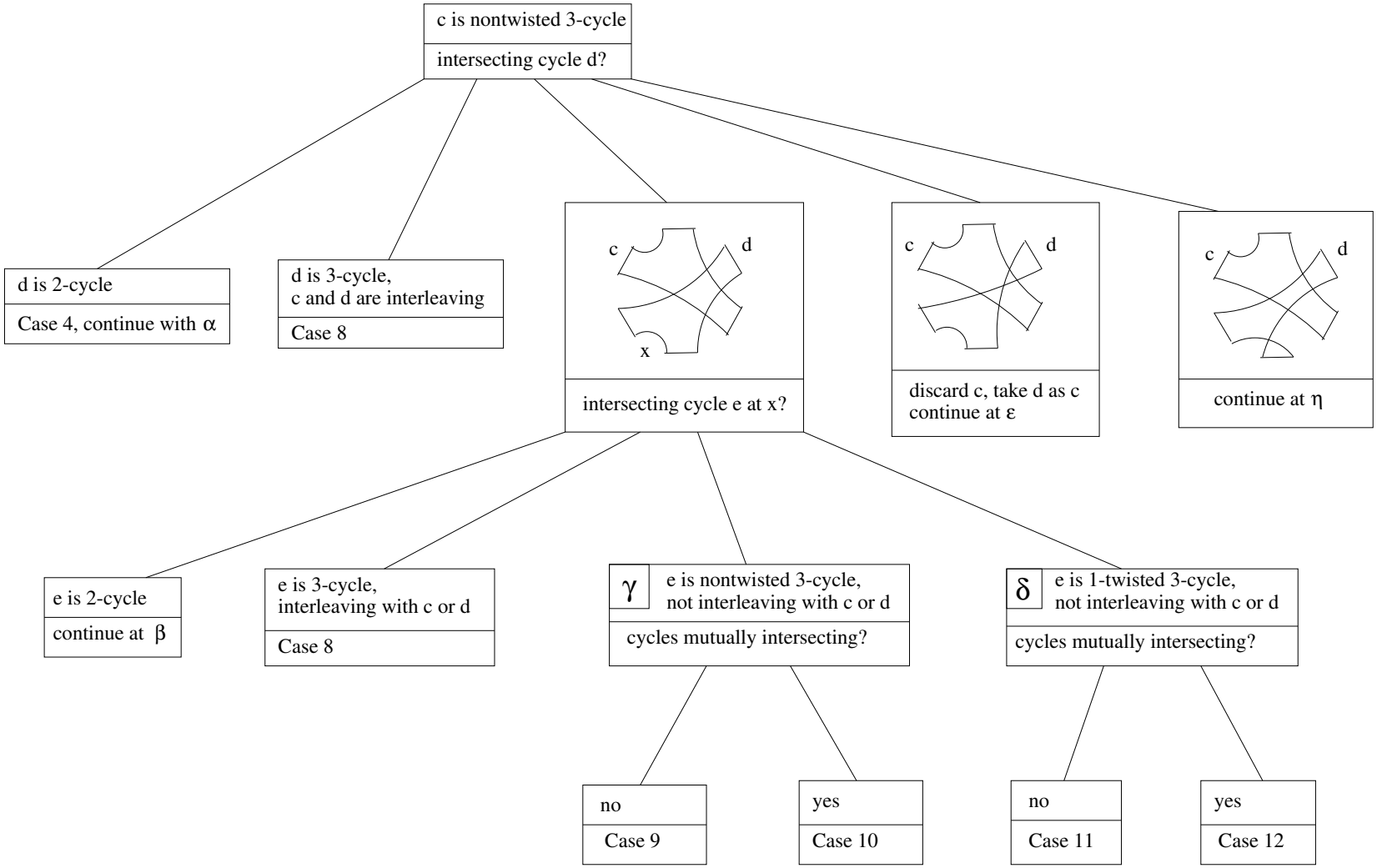


Table 3.2: The algorithm decision tree if we begin with a nontwisted 3-cycle c. All cycles are considered to be r-unoriented 2-cycles or t-unoriented 3-cycles; otherwise the algorithm jumps directly to the trivial case (Case 1). Cross-references α and β can be found on Table 3.1, γ and δ on this table, ε, ζ and η on Table 3.3.

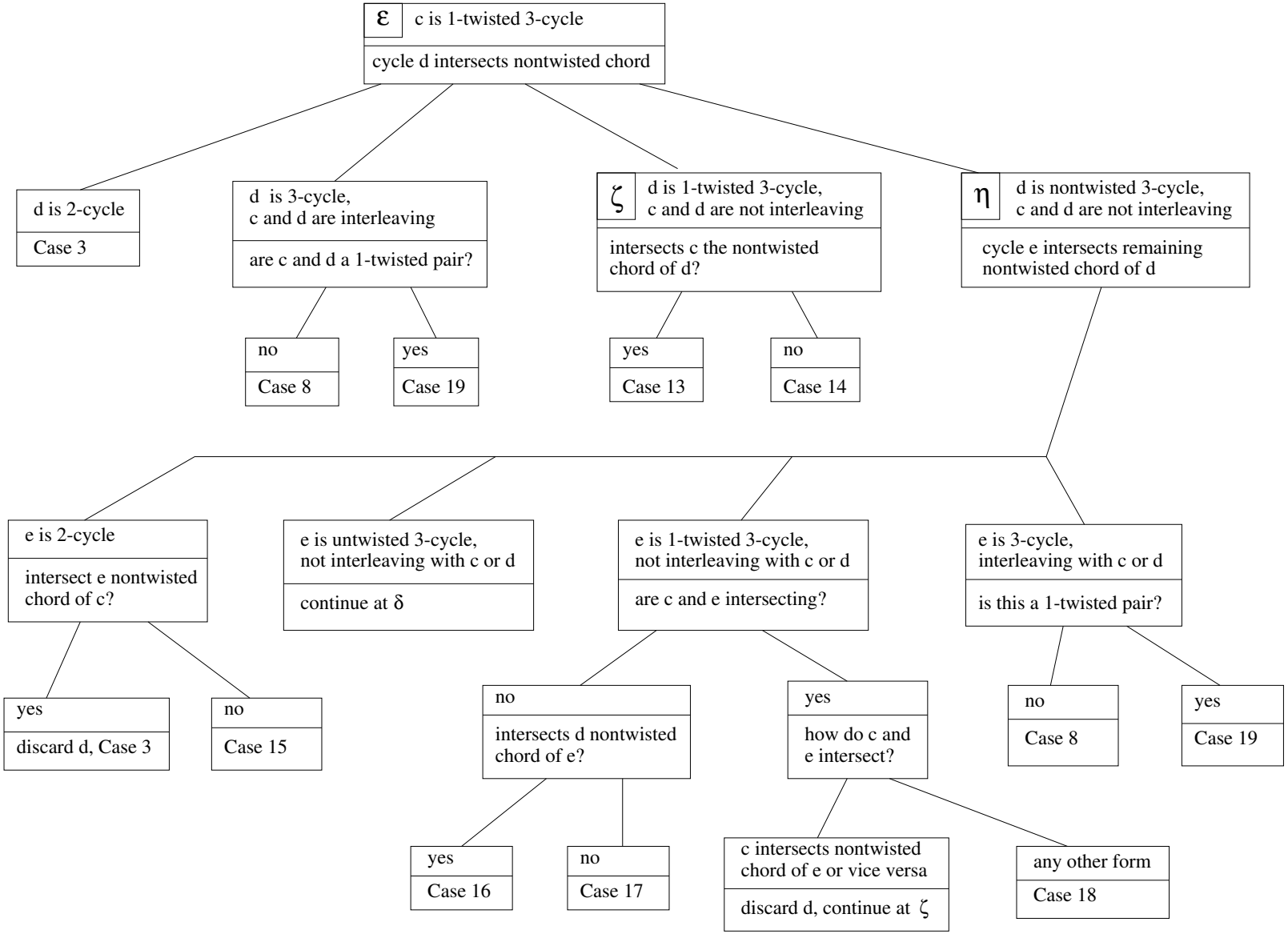


Table 3.3: The algorithm decision tree if we begin with a 1-twisted 3-cycle c. All cycles are considered to be r-unoriented 2-cycles or t-unoriented 3-cycles; otherwise the algorithm jumps directly to the trivial case (Case 1). Cross-references α and β can be found on Table 3.1, γ and δ on Table 3.2, ε, ζ and η on this table.

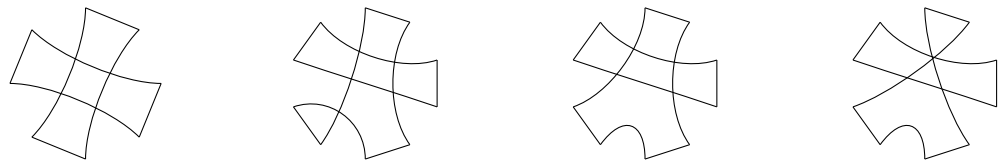


Figure 3.1: The four possible configurations of 2 intersecting cycles, where one of them is a 2-cycle.

The starting point is an arbitrary cycle c of length ≥ 2 in the reality-desire diagram. If c is an r-oriented 2-cycle or a t-oriented 3-cycle, the sequence consists of only one operation that eliminates this cycle (i.e. the operation cuts the cycle into adjacencies). For this operation, $\frac{\Delta\sigma}{w} = \frac{2}{w_t} > \frac{4}{3w_t}$, so we are within our bounds. If c is an r-unoriented 2-cycle or a t-unoriented 3-cycle, it contains a nontwisted edge (see Lemmata 38 and 39). According to Lemma 43, there is a cycle d that intersects c. Now we have a look at the configuration consisting only of the cycles c and d. Again, there are some cases where we can provide a sequence with $\frac{\Delta\sigma}{w} \geq \frac{4}{3w_t}$. In all other cases there must be a chord in the configuration that is not intersected by another chord of the configuration. We search for the cycle that intersects this chord, and extend the configuration by this cycle. We continue doing so until we get a configuration where we can provide a sequence. The whole decision tree can be seen in the Tables 3.1 to 3.3. In some cases we have to discard some old cycles of the configuration, but the depth of the decision tree is bounded. The maximal number of cycles in a configuration is four; if we have reached this configuration size, we can provide a sequence for any possible case. It is also clear that the algorithm always finds a sequence, until the permutation contains only adjacencies. At this point, the permutation is the identity permutation or its reflection, and the sorting is finished.
Now we will list all cases which we have to consider and ensure that we will find any of these cases.

The first case is the trivial case, where we need only one operation:

Case 1. c is an r-oriented 2-cycle or a t-oriented 3-cycle.

In any other case, we need a cycle d that intersects with c. Note that any cycle to which Case 1 does not apply must have a nontwisted chord. According to Lemma 43, this chord must intersect with another cycle d. Additionally we can assume that d is neither an r-oriented 2-cycle nor a t-oriented 3-cycle. Otherwise we could use d as our first cycle and apply Case 1.

If one of the cycles is a 2-cycle, there are four possible configurations, as illustrated in Figure 3.1.

Case 2. c and d are two intersecting 2-cycles (Figure 3.1 a)

Case 3. A 2-cycle intersects the nontwisted chord of a 1-twisted 3-cycle (Figure 3.1 b).

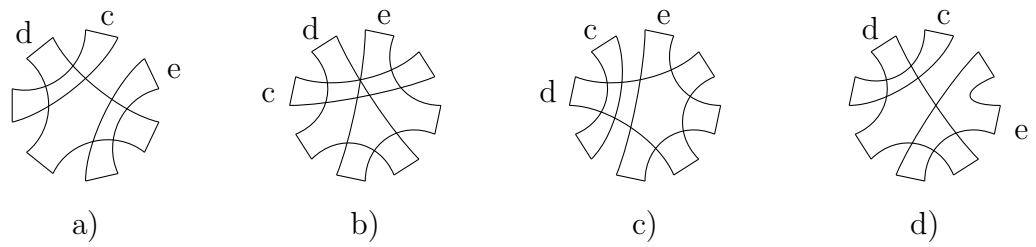


Figure 3.2: A 2-cycle c intersects with a nontwisted 3-cycle d, and another cycle e intersects with the nontwisted chord of d not intersected by c. If e is not 1-twisted, there are four general cases: a) e is a 2-cycle; b) e is a 3-cycle, and c intersects the nontwisted chord not intersected by d; c) e is a 3-cycle, d and e are interleaving; d) e is a 3-cycle and has a nontwisted chord intersected neither by c nor by d.

Case 4. A 2-cycle intersects with a nontwisted 3-cycle (Figure 3.1 c).

The remaining case, where the 2-cycle intersects with the twisted chords of the 3-cycle, can be ignored. In this case, we will use the 3-cycle as cycle c and discard the 2-cycle. So, we have a 1-twisted 3-cycle, and we start the searching algorithm again: There must exist a cycle d that intersects with the nontwisted chord of c.
Case 2 and Case 3 can be handled directly. In Case 4, there must be another cycle e that intersects the last chord of the 3-cycle. Several cases can occur: e is either a 2-cycle, a nontwisted 3-cycle or a 1-twisted 3-cycle. Again, if e is a 1-twisted 3-cycle, we will ignore the other cycles, and use e as cycle c. If e is a 2-cycle or a nontwisted 3-cycle, there are four general cases, as illustrated in Figure 3.2.
In the first of these cases, we can assume that the 2-cycles are not intersecting. Otherwise we can apply Case 2.

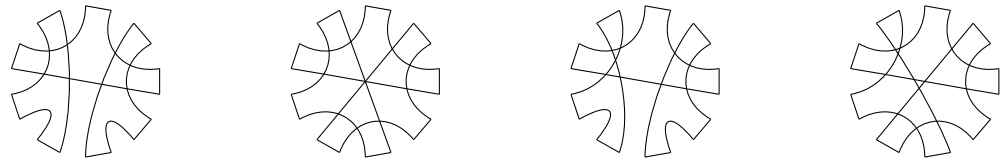
Case 5. c and e are 2-cycles, d is a nontwisted 3-cycle. c and e are not intersecting, and each nontwisted chord of d is intersected by c or e (Figure 3.2 a).

Case 6. c is a 2-cycle, d and e are intersecting nontwisted 3-cycles. c intersects the other nontwisted chords of d and e (Figure 3.2 b).

These two cases can be handled directly. The case where we have two interleaving 3-cycles will be discussed later (Figure 3.2 c). For the case illustrated in Figure 3.2 d), we need a fourth cycle f, that intersects with the last nontwisted chord of e. If f is a 2-cycle, this case can be handled directly:

Case 7. d and e are two intersecting nontwisted 3-cycles, and c and f are 2-cycles. c intersects with the nontwisted chord of d that is not intersected by e, and f intersects with the nontwisted chord of e that is not intersected by d. c is not intersecting with e or f, and f is not intersecting with c or d.

Note that we have to consider only the case where c and f are not intersecting. The other case already has been handled in Case 2. Also the case where one of the 2-cycles intersects both 3-cycles is not to be considered here. Any possible configuration can be



a) b) c) d)

Figure 3.3: The four main cases for three intersecting, non-interleaving 3-cycles, where at least two of them are nontwisted, and one of the nontwisted cycles intersects both other cycles.

reduced to Case 5 or to Case 6. If f is a 3-cycle, we discard the 2-cycle and apply any of the cases with three 3-cycles that are connected by intersections. These cases will be discussed later (Cases 9 to 12).
Now, we will have a look at the case that our first two cycles c and d are both 3-cycles. The first possible case is that we have two interleaving 3-cycles:

Case 8. c and d are interleaving 3-cycles, and do not form a 1-twisted pair.

As long as interleaving cycles do not form a 1-twisted pair, they can be handled easily. If they do, it is rather complicated, and we first must handle a lot of other cases, before we are able to solve this case. In the following cases, we assume that we have no interleaving 3-cycles, except if they form a 1-twisted pair.
If both c and d are nontwisted, d has a nontwisted chord that must intersect with a third cycle e. There are different possible configurations: e can either be a 2-cycle, a nontwisted 3-cycle, or a 1-twisted 3-cycle. If e is a 2-cycle, the possible cases already have been discussed above. If e is a 3-cycle, there are four possible cases:

Case 9. c, d and e are all nontwisted 3-cycles, and c does not intersect with e (Figure 3.3 a).

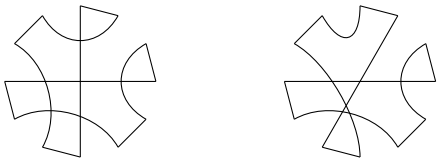
Case 10. c, d and e are mutually intersecting nontwisted 3-cycles (Figure 3.3 b).

Case 11. c and d are nontwisted 3-cycles, e is a 1-twisted 3-cycle. c and e are not intersecting (Figure 3.3 c).

Case 12. Two nontwisted 3-cycles and a 1-twisted 3-cycle are mutually intersecting (Figure 3.3 d).

Each of these cases can be handled directly. Some example configurations are illustrated in Figure 3.3.
Now, we begin with two 1-twisted cycles c and d. As we always search for cycles intersecting with the nontwisted chord of a given cycle, we can assume that d intersects the nontwisted chord of c. There are two possible cases:

Case 13. Two 1-twisted 3-cycles are intersecting, such that the nontwisted chord of each cycle is intersected by the other cycle (Figure 3.4 a).



a) b)

Figure 3.4: Two intersecting 1-twisted 3-cycles

Case 14. c and d are two intersecting 1-twisted 3-cycles. d intersects the nontwisted chord of c, but c does not intersect the nontwisted chord of d (Figure 3.4 b).

Both cases can be handled directly.
Now, let c be a 1-twisted 3-cycle, and d a nontwisted 3-cycle that intersects the nontwisted chord of c. There must be a third cycle e that intersects with the nontwisted chord of d not intersected by c. If e is a 2-cycle, there are two possibilities: Either the 2-cycle also intersects the nontwisted chord of c (then we can apply Case 3), or we have a case that we can handle directly, and an example configuration is illustrated in Figure 3.5:

Case 15. c is a 1-twisted 3-cycle, and d is a nontwisted 3-cycle that intersects the nontwisted chord of c. The nontwisted chord of d is intersected by a 2-cycle that does not intersect the nontwisted chord of c (Figure 3.5).

If e is a nontwisted 3-cycle, we have either Case 12 or Case 11. If e is a 1-twisted 3-cycle, and one of the 1-twisted 3-cycles intersects with the nontwisted chord of the other, we have either Case 13 or Case 14. There are three remaining cases, and example configurations are illustrated in Figure 3.6.

Case 16. c is a nontwisted 3-cycle, d and e are 1-twisted 3-cycles. c intersects with the nontwisted chords of d and e, d and e do not intersect each other (Figure 3.6 a).

Case 17. c is a nontwisted 3-cycle, d and e are 1-twisted 3-cycles. c intersects with the nontwisted chord of d and the twisted chord of e. d and e do not intersect each other (Figure 3.6 b).

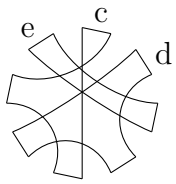
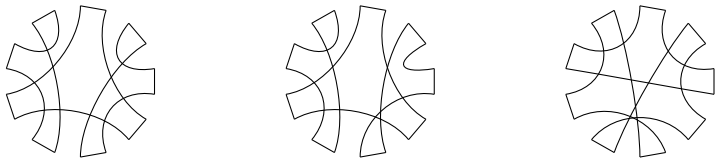


Figure 3.5: A 1-twisted 3-cycle c intersects a nontwisted 3-cycle d. The nontwisted chord of d not intersected by c is intersected by a 2-cycle e. e does not intersect the nontwisted chord of c.



a) b) c)

Figure 3.6: A nontwisted 3-cycle intersects two 1-twisted 3-cycles. These are some possible example configurations. Figure a) refers to Case 16, Figure b) to Case 17, Figure c) to Case 18

Case 18. c is a nontwisted 3-cycle, d and e are 1-twisted 3-cycles. c intersects with the nontwisted chord of d and any chord of e. d and e intersect with their twisted chords, but are not interleaving (Figure 3.6 c).

All these cases can be handled directly. Note that we have the precondition that c intersects with the nontwisted chord of d. The last case we have to consider is two interleaving 3-cycles that form a 1-twisted pair:

Case 19. Two 1-twisted 3-cycles form a 1-twisted pair.

Now, we have covered all possible cases. In the next section we will provide the sequences for them.

3.2 Sequences for the different cases

We will now provide sequences for the different cases. Each sequence has a weight w, and the gain of the score per weight $\frac{\Delta\sigma}{w}$ is at least $\frac{4}{3w_t}$ (with our bounds $w_r \leq w_t \leq 2w_r$). The sequences that only involve 3-cycles have been shown in [Har03] and [HS04], except the sequences for the Cases 11, 12 and 19, which we could simplify. Although the sequences by [Har03] and [HS04] have been developed for sorting by reversals and transpositions without weights, the gain of the score is high enough, so we can use them also for our purposes.
We will solve many of the cases by showing the sequences directly in all possible configurations for the case, where symmetric configurations are ommitted. In the figures, the reality-edges where the operations act on are marked. If three edges are marked with $*$, the move is a transposition. If two edges are marked with x and one with $*$, the move is a transreversal that inverts the segment between the x. A reversal is indicated by two edges marked with x.

The following two lemmata describe Case 1:

Lemma 50. *If π contains an r-oriented 2-cycle, there exists a reversal that increases c_{odd} by 2 and decreases c_{even} by 1.*

Remark. For this move, $\frac{\Delta\sigma}{w} = \frac{2}{w_t}$.

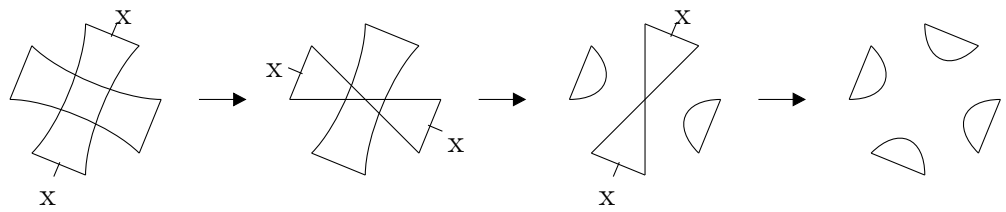


Figure 3.7: A $0_r1_r1_r$-sequence for two intersecting 2-cycles.

Proof. Just use the reversal that splits c into two cycles. The resulting cycles must be adjacencies. □

Lemma 51. *If π contains a t-oriented 3-cycle, then there exists a transposition or a transreversal that increases c_{odd} by 2 and does not change c_{even}.*

Remark. For this move, $\frac{\Delta\sigma}{w} = \frac{2}{w_t}$.

Proof. Just use the move that splits c into three cycles. The resulting cycles must be adjacencies. □

For all other cases, we assume that we only have r-unoriented 2-cycles and t-unoriented 3-cycles.
The next lemma describes Case 2:

Lemma 52. *If π contains two intersecting 2-cycles, then there exists a $0_r1_r1_r$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = -2$.*

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. The sequence is described in Figure 3.7. □

The next lemma describes Case 3:

Lemma 53. *If π contains a 2-cycle that intersects the nontwisted chord of a 1-twisted 3-cycle, then there exists a $1_r1_r1_r$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = -1$.*

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{2w_r+2w_t}{3w_rw_t}$.

Proof. The sequence is described in Figure 3.8. □

As we split Case 4 into several subcases, we do not need a lemma for the main case. We will rather provide some lemmata for the subcases. The next lemma describes Case 5:

Lemma 54. *Let π be a permutation and let c, d, and e be three cycles in its reality-desire diagram with the following properties:*

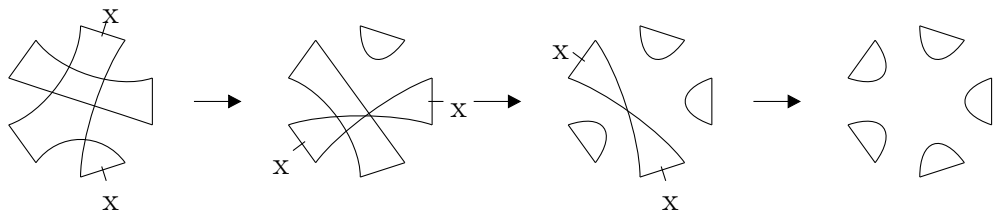


Figure 3.8: A $1_r1_r1_r$-sequence for a 2-cycle intersecting the nontwisted chord of a 1-twisted 3-cycle.

- *d is a nontwisted 3-cycle, c and e are nontwisted 2-cycles,*
- *each chord of d is intersected by c or e,*
- *c and e are not intersecting.*

Then there exists a $0_t2_t2_t$-sequence with $\Delta c_{odd} = 6$ and $\Delta c_{even} = -2$.

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{2}{3w_t} + \frac{4w_r}{3w_t^2}$.

Proof. The sequence is described in Figure 3.9. □

The next lemma describes Case 6:

Lemma 55. *Let π be a permutation and let c, d, and e be three cycles in its reality-desire diagram with the following properties:*

- *c is a nontwisted 2-cycle, d and e are intersecting, non-interleaving nontwisted 3-cycles,*
- *the chords of d and e not being intersected by the other 3-cycle are intersected by c.*

Then there exists a $0_r1_r2_{tr}$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = -1$.

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{2w_r+2w_t}{w_t(2w_r+w_t)}$.

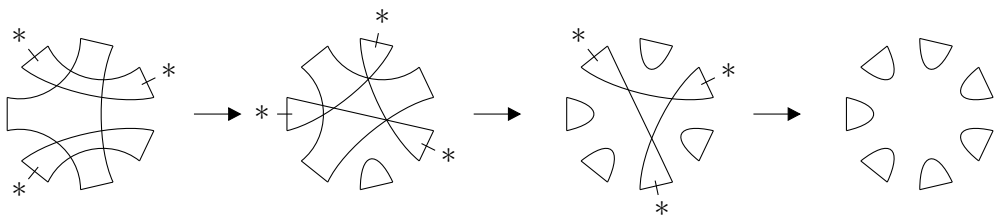


Figure 3.9: A $0_t2_t2_t$-sequence for two 2-cycles intersecting a nontwisted 3-cycle.

Proof. The sequence is described in Figure 3.10. □

The next lemma describes Case 7:

Lemma 56. *Let π be a permutation and let c, d, e, and f be cycles in its reality-desire diagram with the following properties:*

- *c and f are nontwisted 2-cycles, d and e are intersecting, non-interleaving nontwisted 3-cycles,*
- *c intersects the chord of d not intersected by e,*
- *f intersects the chord of e not intersected by d,*
- *c does not intersect with e or f,*
- *f does not intersect with c or d.*

Then there exists a $0_t2_t2_t2_t$-sequence with $\Delta c_{odd} = 8$ and $\Delta c_{even} = -2$, or a $0_t2_t2_t$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.

Remark. For the $0_t2_t2_t2_t$-sequence, $\frac{\Delta\sigma}{w} = \frac{1}{w_t} + \frac{w_r}{w_t^2}$. For the $0_t2_t2_t$-sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. There are four possible configurations. For each of them, a sequence is described in Figure 3.11. □

The next lemma describes Case 8. Two of the sequences for this case have been described in [Har03] and [HS04].

Lemma 57. *If π contains two interleaving t-unoriented 3-cycles that do not form a 1-twisted pair, then there exists a sequence with weight $w \leq 3w_t$, $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.*

Remark. For this sequence, $\frac{\Delta\sigma}{w} \geq \frac{4}{3w_t}$.

Proof. There are three cases that can occur. For all of them, the sequence is described in Figure 3.12. □

The next lemma describes Case 9. The sequences have been described in [Har03].

Lemma 58. *Let π be a permutation and let c, d, and e be nontwisted 3-cycles in its reality-desire diagram with the following properties:*

- *c intersects d and e, but does not interleave with them,*
- *d and e are not intersecting.*

Then there exists a $0_t2_t2_t$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

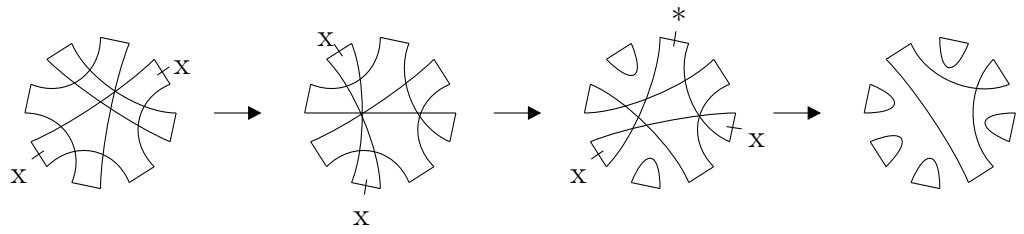


Figure 3.10: A $0_r1_r2_{tr}$-sequence for two intersecting nontwisted 3-cycles and a 2-cycle that intersects the remaining chords of the 3-cycles.

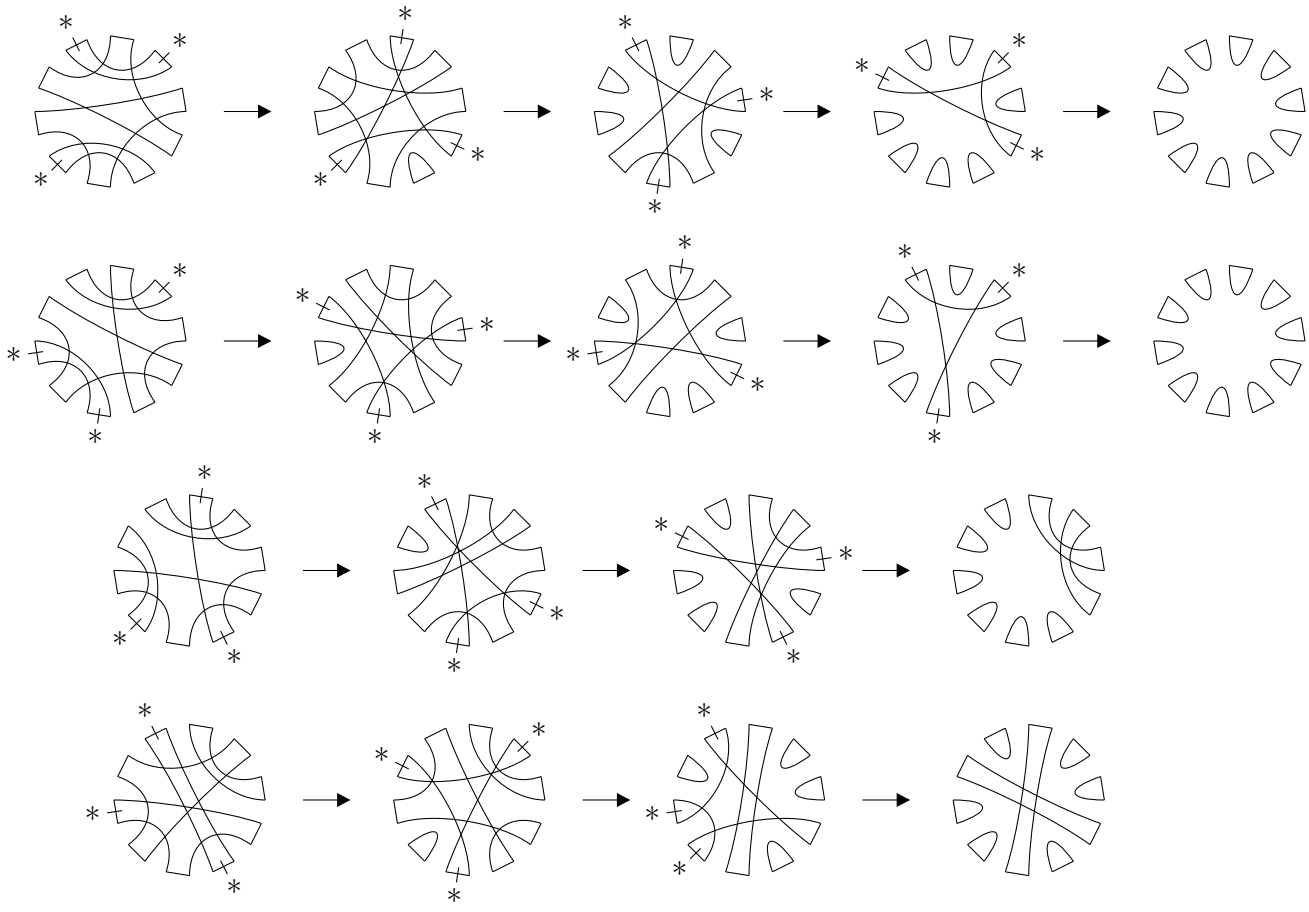


Figure 3.11: Sequences for two 2-cycles that intersect the remaining nontwisted chords of two intersecting nontwisted 3-cycles.

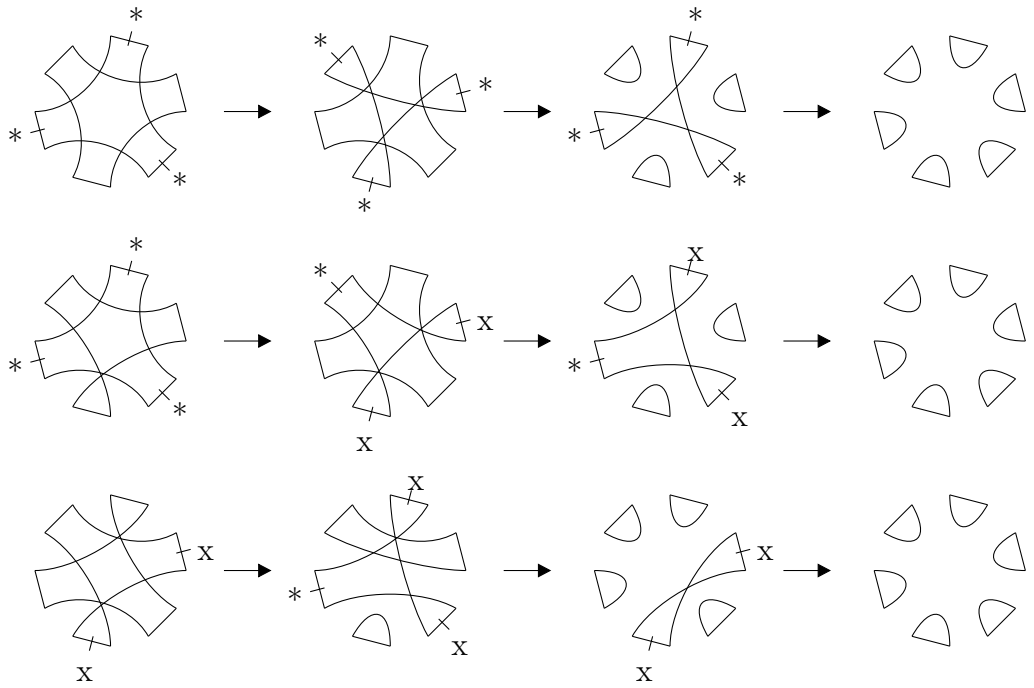


Figure 3.12: Sequences for two interleaving 3-cycles that do not form an 1-twisted pair.

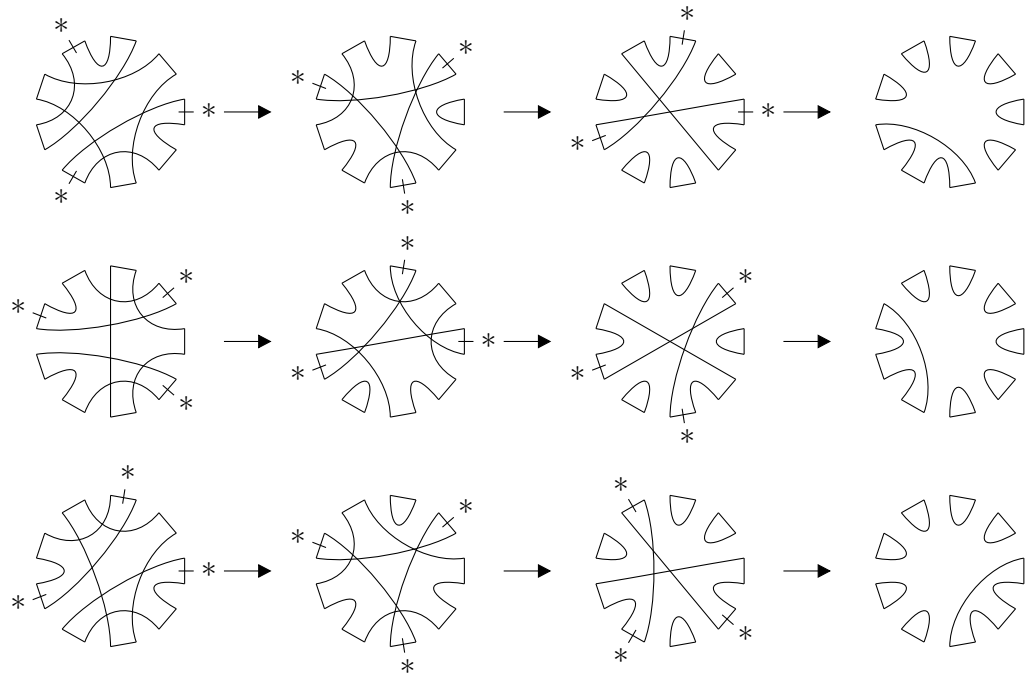


Figure 3.13: Sequences for three intersecting nontwisted 3-cycles.

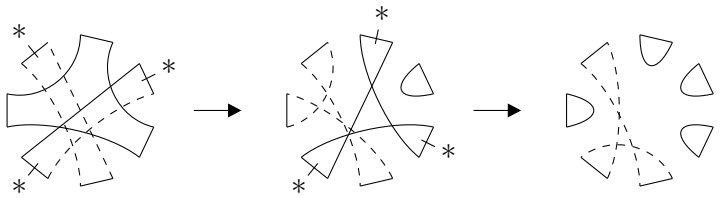


Figure 3.14: Sequence for three mutual intersecting nontwisted 3-cycles. Dashed lines either represent a single desire-edge, or two desire-edges separated by a reality-edge. As the starting configuration contains three 3-cycles, the cycle with the three dashed lines must be a 5-cycle. Lemma 49 ensures that we can perform a third transposition that splits this cycle into a 3-cycle and two adjacencies.

Proof. There are three possible configurations. For all of them, a sequence is described in Figure 3.13. □

The next lemma describes Case 10. The sequences have been described in [Har03].

Lemma 59. *If a permutation π contains three nontwisted non-interleaving 3-cycles that are mutually intersecting, then there exists a $0_t2_t2_t$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.*

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. The general case is illustrated in Figure 3.14. As all cycles are unoriented, the preconditions of Lemma 49 are fulfilled. We begin with a transposition as illustrated in the figure. The result is an adjacency, a 3-twisted 3-cycle, and a 5-cycle. According to Lemma 49, the 5-cycle allows a 2_t-move t that splits it into two adjacencies and a 3-cycle. The second move is a transposition that eliminates the 3-cycle. This move does not change the structure of the 5-cycle, so we can perform the transposition t as third move. The resulting configuration contains one 3-cycle and six adjacencies. □

The next lemma describes Case 11. We cannot use the proof shown in [HS04] here, as they start their sequences with the first move of the corresponding sequence of Case 9, such that the first transposition does not act on a twisted edge. However, there is a case where the first move must act on the twisted edge, and this case cannot be solved by using a symmetric configuration.

Lemma 60. *Let π be a permutation and let c, d, and e be three cycles in its reality-desire diagram with the following properties:*

- *c and d are nontwisted 3-cycles, e is a 1-twisted 3-cycle,*
- *d intersects c and e, but does not interleave with them,*
- *c and e are non-intersecting.*

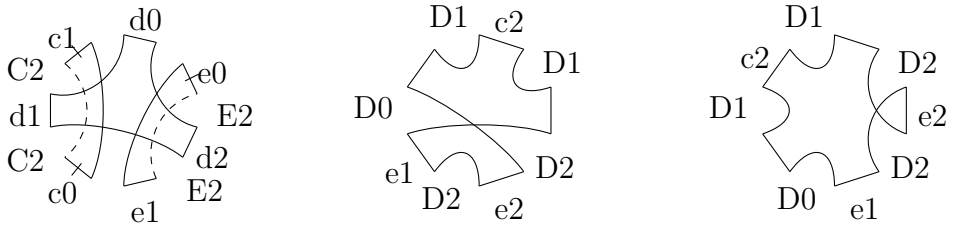


Figure 3.15: Left: The general case of Case 11. Dashed lines represent two desire-edges separated by a reality-edge. One of the edges e_0 and e_1 also can be twisted. c_2 must be at one of the positions marked with $C2$, and e_2 must be at one of the positions marked with $E2$. Middle: This is the resulting 5-cycle after the first transposition if e_0 was twisted. Right: This is the resulting 5-cycle after the first transposition if e_0 was not twisted. In both pictures, the possible positions for the reality-edges of d are marked with $D0$, $D1$, and $D2$.

Then there exists a $0_t2_t2_t$*-sequence or a* $0_t2_t2_{tr}$*-sequence with* $\Delta c_{odd} = 4$ *and* $\Delta c_{even} = 0$.

Remark. For these sequences, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. The general case is illustrated in the left picture of Figure 3.15. Dashed lines represent two desire-edges separated by a reality-edge. e_0 or e_1 also can be twisted. Note that c_2 (the third reality-edge of cycle c) must be at one of the positions marked with $C2$, and e_2 (the third reality-edge of cycle e) at one of the positions marked with $E2$. The first move is a transposition that acts on c_0, e_0, and c_1, cutting a part of the cycle c and inserting it into e. This move makes d 3-twisted (see Lemma 46) and transforms c and e into a 5-cycle and an adjacency. If e_0 was twisted, the configuration of the 5-cycle is the one described in the middle picture of Figure 3.15, otherwise it is the one described in the right picture (where also e_2 can be twisted instead of e_1). $D0$, $D1$, and $D2$ mark the possible positions for the reality-edges of cycle d. In any case, the 5-cycle is t-unoriented, and each pair of reality-edges of d is separated by a reality-edge of the 5-cycle. Therefore, the preconditions of Lemma 48 are fulfilled, and a transposition that acts on the edges of d (this is a 2_t-move) makes the 5-cycle t-oriented, allowing a second 2_t-move or a 2_{tr}-move. □

The next lemma describes Case 12.

Lemma 61. *Let* π *be a permutation and let* c, d*, and* e *be three cycles in its reality-desire diagram with the following properties:*

- *c and d are nontwisted 3-cycles, e is a 1-twisted 3-cycle,*
- *c, d and e are mutually intersecting,*
- *c, d and e are not interleaving.*

Then there exists a $0_t2_t2_t$*-sequence or a* $0_t2_t2_{tr}$*-sequence with* $\Delta c_{odd} = 4$ *and* $\Delta c_{even} = 0$.

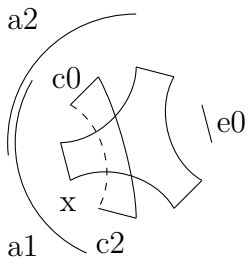


Figure 3.16: The general situation in Case 12. The dotted line represents two chords separated by the reality-edge c_1. c_1 must be in the arc a_1. At least one of the reality-edges e_1 and e_2 must be at the position marked with x. Cycle e has no reality-edge in arc a_2. For clarity, the chords of cycle e are not indicated in the diagram.

Figure 3.17: If both chords e_1 and e_2 were at the position marked with x in Figure 3.16, the resulting 5-cycle has one of these configurations after the second move. Left: This is the configuration if e_0 was twisted. The dotted line represents two chords separated by the reality-edge c_1, and c_1 must be at one of the positions marked with x. Right: This is the configuration if e_1 was twisted. If e_2 was twisted, the resulting configuration is symmetric.

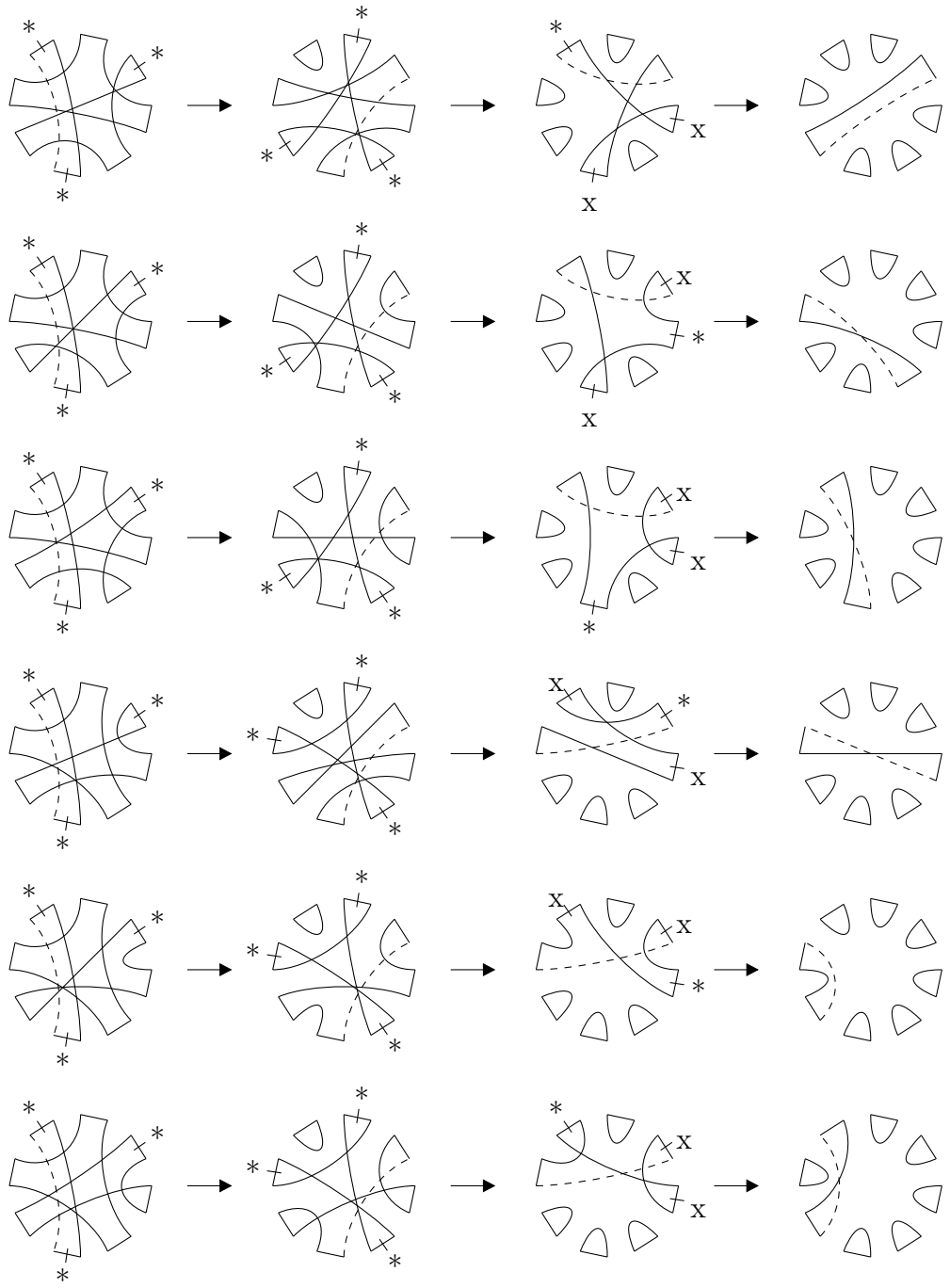


Figure 3.18: The sequences for the six possible configurations of Case 12 if cycle e has only one reality-edge in the arc between c_0 and c_2. The dotted line represents two chords separated by the reality-edge c_1.

Remark. For these sequences, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. The general situation is illustrated in Figure 3.16. The dotted line represents two chords separated by the reality-edge c_1. Because there exist many possibilities for the twist of cycle e, the chords of e are not in the diagram. As c is nontwisted, c_1 must be in arc a_1. At least one of the reality-edges of e is at the position marked with x. Without loss of generality, we can assume that no reality-edge of cycle e is in the arc a_2. The first move is always a transposition that acts on c_0, c_2, and e_0. This makes d 3-twisted, and the second move is the transposition that eliminates d. For the third move, we must distinguish between two cases for the positions of e_1 and e_2 in the starting configuration, which result in different configurations for the 5-cycle after the first transposition:

- Both e_1 and e_2 are at the position marked with x in Figure 3.16. If e_0 was twisted, the resulting configuration for the 5-cycle can be seen in the left picture of Figure 3.17. Otherwise, the configuration is that of the right picture (shown is the configuration where e_1 was twisted; if e_2 was twisted, the result is symmetric). Before the first move, the reality-edge c_1 must be at one of the positions marked with x in Figure 3.16. Note that it cannot be between e_1 and e_2 because c and e are non-interleaving. For both positions, the preconditions of Lemma 47 are fulfilled, allowing a 2_{tr}-move.
- If only one of e_1 and e_2 is at the position marked with x in Figure 3.16, then there are six possible configurations. For each of them, a sequence is described in Figure 3.18.

□

The next lemma describes Case 13.

Lemma 62. *If a permutation contains two intersecting, non-interleaving 1-twisted 3-cycles, and each of these cycles intersects with the nontwisted chord of the other cycle, then there exists a $1_r2_{tr}1_r$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.*

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{w_t+2w_r}$.

Proof. There are three possible configurations, and the sequences for these configurations are shown in Figure 3.19. □

The next lemma describes Case 14:

Lemma 63. *Let π be a permutation and let c and d be two cycles in its reality-desire-diagram with the following properties:*

- *c and d are 1-twisted 3-cycles,*
- *d intersects the nontwisted chord of c,*
- *c does not intersect the nontwisted chord of d.*

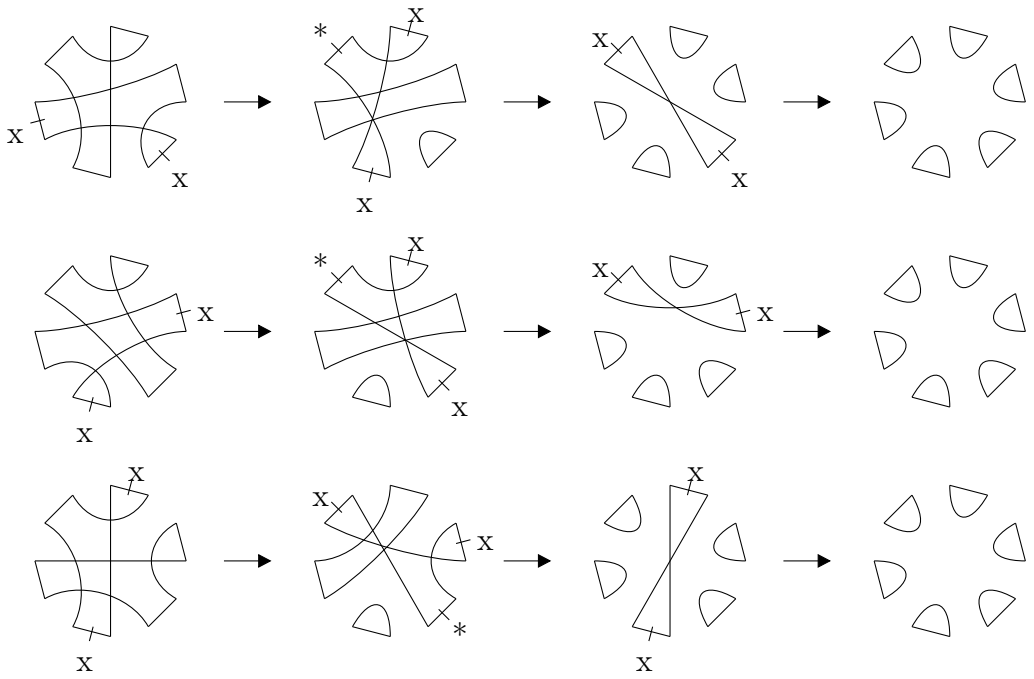


Figure 3.19: Sequences for two intersecting, non-interleaving 1-twisted 3-cycles, where each of them intersects with the nontwisted chord of the other cycle.

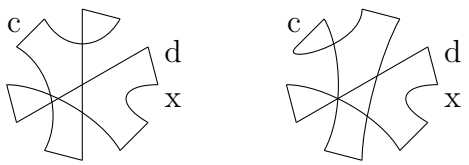


Figure 3.20: Two intersecting 1-twisted 3-cycles, and one of the nontwisted chords is not intersected by the other cycle. At position x, there must be a reality-edge of another cycle that intersects with the nontwisted chord.

Then we can either apply Case 3 or there exists a $0_r2_{tr}2_{tr}$*-sequence with* $\Delta c_{odd} = 4$ *and* $\Delta c_{even} = 0$ *or a* $1_r2_{tr}2_{tr}$*-sequence with* $\Delta c_{odd} = 4$ *and* $\Delta c_{even} = 1$*.*

Remark. For this sequence, either $\frac{\Delta\sigma}{w} = \frac{4}{2w_t+w_r}$ or $\frac{\Delta\sigma}{w} = \frac{6w_t-2w_r}{w_r(2w_t+w_r)}$.

Proof. There are two possible configurations, as described in Figure 3.20. According to Lemma 43, there must be a reality-edge of another cycle e at the position marked with an x, and this cycle must intersect with the nontwisted chord of d. If this cycle is a 2-cycle, we can apply Case 3. If e is a 3-cycle, we can use the proof given in [HS04]: We begin with a reversal on e that twists one of the nontwisted reality-edges of d. This is possible because e intersects the nontwisted chord of d. For the resulting configuration of c and d, there are four possibilities:

- c is 1-twisted and d is 2-twisted. In any case, the precondition of Lemma 40 is fulfilled, so we can apply a $2_{tr}2_{tr}$-sequence.
- c and d interleave and both cycles are 2-twisted. In any case, they have at least three consecutive twists. According to Lemma 41, a $2_{tr}2_{tr}$-sequence is possible.
- c and d are interleaving, c is nontwisted and d is 2-twisted. Eliminating d by a 2_{tr}-move makes c 2-twisted, so another 2_{tr}-move is possible.
- c and d are intersecting, and both are 2-twisted. In any case, c has no reality-edge between the two twisted reality-edges of d, so we can apply a 2_{tr}-move on c without reversing any part of d. Therefore, d remains 2-twisted, that allows another 2_{tr}-move.

In any case, our sequence will eliminate c and d. The first reversal is either a 0_r-move, or it splits e into a 2-cycle and an adjacency. In both cases c_{odd} remains unchanged. In the first case, also c_{even} remains unchanged. In the latter case, c_{even} increases by 1. The two transreversals increase c_{odd} by 4 and let c_{even} unchanged. □

The next lemma describes Case 15:

Lemma 64. *Let* π *be a permutation and let* c, d*, and* e *be three cycles in its reality-desire-diagram with the following properties:*

- c *is a 1-twisted 3-cycle,* d *is a nontwisted 3-cycle, and* e *is a nontwisted 2-cycle,*
- d *intersects the nontwisted chord of* c*,* c *and* d *are not interleaving,*
- e *intersects the chord of* d *not intersected by* c*,*
- e *does not intersect the nontwisted chord of* c*.*

Then there is a $0_t2_t2_t1_r$*-sequence, a* $0_t2_t2_{tr}1_r$*-sequence, or a* $1_r1_r2_{tr}1_r$*-sequence with* $\Delta c_{odd} = 6$ *and* $\Delta c_{even} = -1$*.*

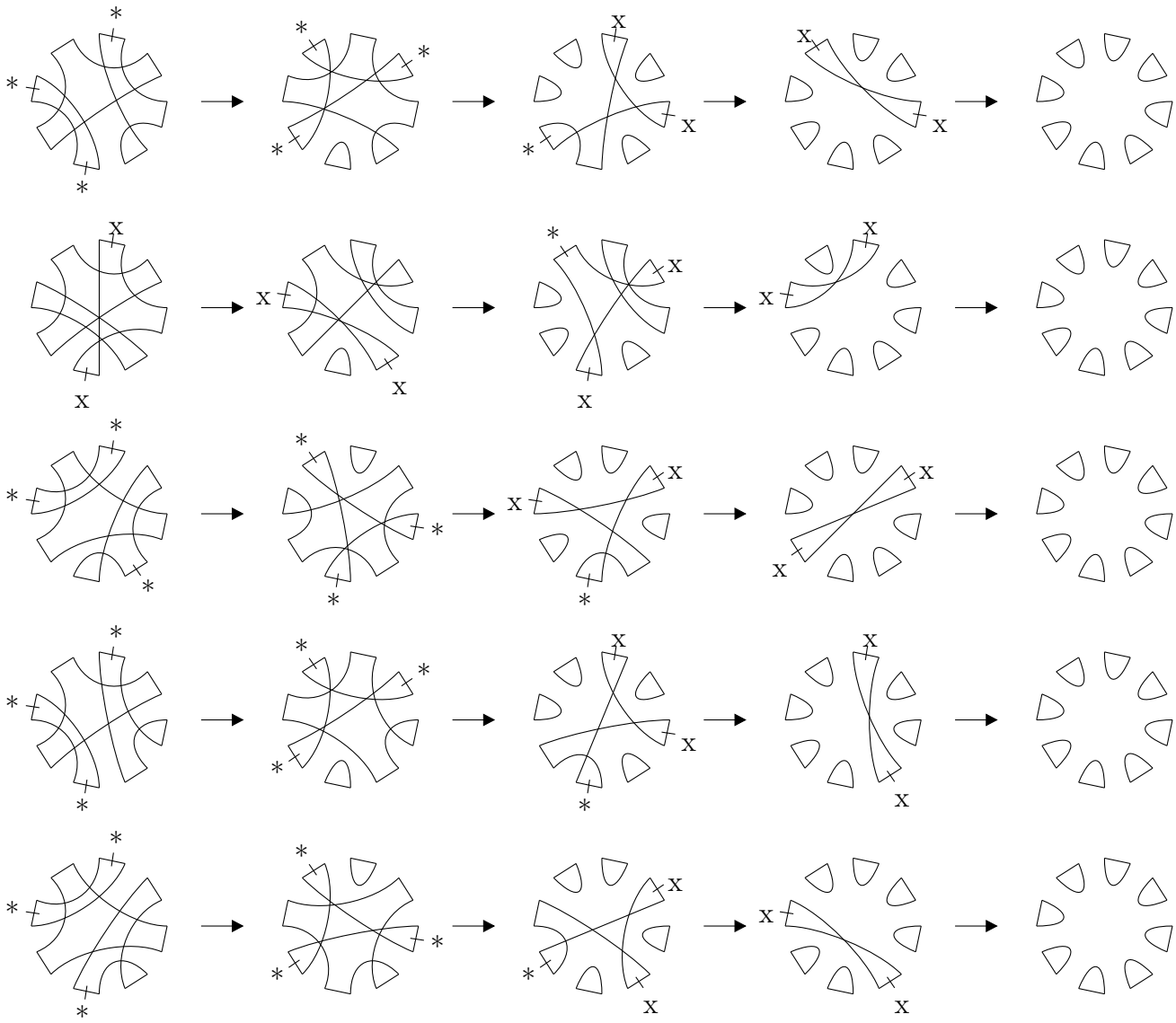


Figure 3.21: Sequences for a nontwisted 3-cycle intersecting the nontwisted chord of a 1-twisted cycle, and a 2-cycle intersecting the remaining nontwisted chord of the nontwisted cycle.

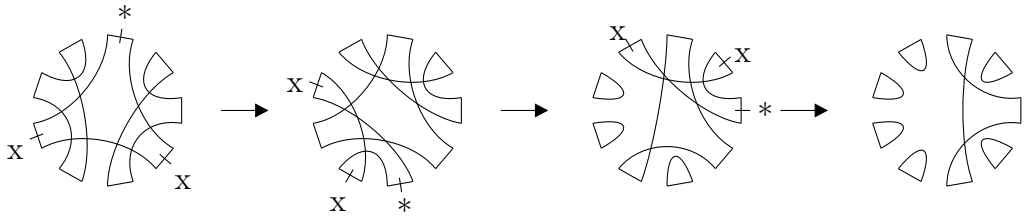


Figure 3.22: An example configuration for Lemma 65. Imagine the first transreversal as mirroring the part between the two x and inserting at * will help to see that the twisted cycles remain non-intersecting.

Remark. For the $0_t2_t2_t1_r$-sequence and the $0_t2_t2_{tr}1_r$-sequence, $\frac{\Delta\sigma}{w} = \frac{4w_t+2w_r}{w_t(3w_t+w_r)}$. For the $1_r1_r2_{tr}1_r$-sequence, $\frac{\Delta\sigma}{w} = \frac{4w_t+2w_r}{w_t(w_t+3w_r)}$.

Proof. There are five possible configurations. For each of them, a sequence is described in Figure 3.21. □

The next lemma describes Case 16. The sequences for this case have been described in [HS04].

Lemma 65. *Let π be a permutation and let c, d, and e be three cycles in its reality-desire diagram with the following properties:*

- *c is a nontwisted 3-cycle, d and e are 1-twisted 3-cycles,*
- *each chord of c is intersected by a nontwisted chord of d or e,*
- *c does not interleave with d or e,*
- *d and e are not intersecting.*

Then there exists a $0_{tr}2_{tr}2_{tr}$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. We begin with a transreversal that acts on the three reality-edges of c. There exists one chord in c that intersects with both d and e. If we choose the transreversal that inverts the arc below this chord, d and e remain non-intersecting, and become both 2-twisted. Both cycles can now be eliminated by another transreversal. An example sequence can be seen in Figure 3.22. □

The next lemma describes Case 17. The sequences for this case have been described in [HS04].

Lemma 66. *Let π be a permutation and c, d, and e be three cycles in its reality-desire diagram with the following properties:*

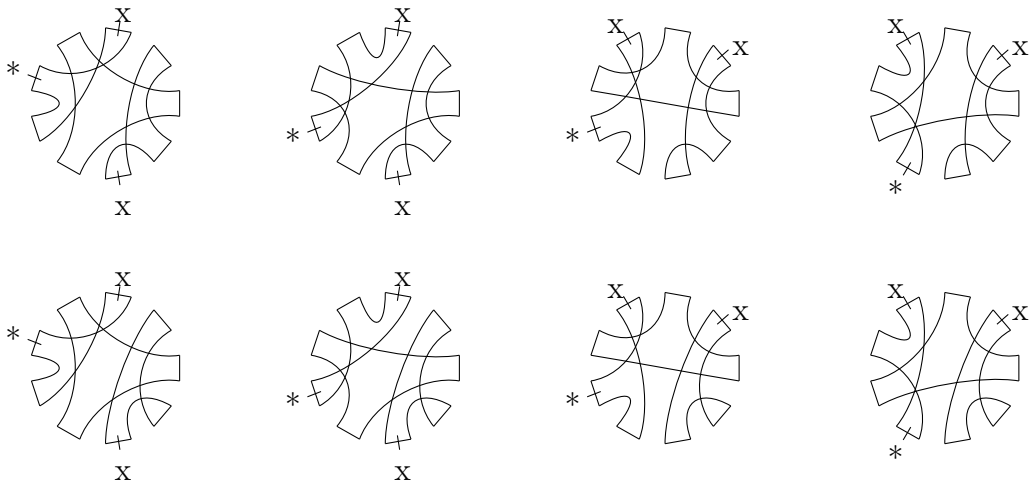


Figure 3.23: The possible configuration for the three 3-cycles of Lemma 66. The first move is marked in each diagram, the result is a 5-cycle, a t-oriented 3-cycle and an adjacency. After eliminating the 3-cycle, the 5-cycle is t-oriented and has one of the forms described in Figure 3.24.

- *c is a nontwisted 3-cycle, d and e are 1-twisted 3-cycles,*
- *c, d and e are not interleaving,*
- *c intersects the nontwisted chord of d,*
- *the chord of c that is not intersected by d intersects with the twisted chords of e; the nontwisted chord of e is not intersected by c,*
- *d and e are not intersecting.*

Then there exists a sequence with $w = 3w_t$, $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. There are eight possible configurations for this case. All these configurations are illustrated in Figure 3.23, and the first move of each sequence is marked there. The resulting configurations consist of a t-oriented 3-cycle, a 5-cycle, and an adjacency. The second move eliminates the 3-cycle and makes the 5-cycle t-oriented. All possible configurations for the 5-cycle are illustrated in Figure 3.24. The moves marked in this figure transform the 5-cycle into a 3-cycle and two adjacencies. □

The next lemma describes Case 18. The sequences for this case have been shown in [HS04].

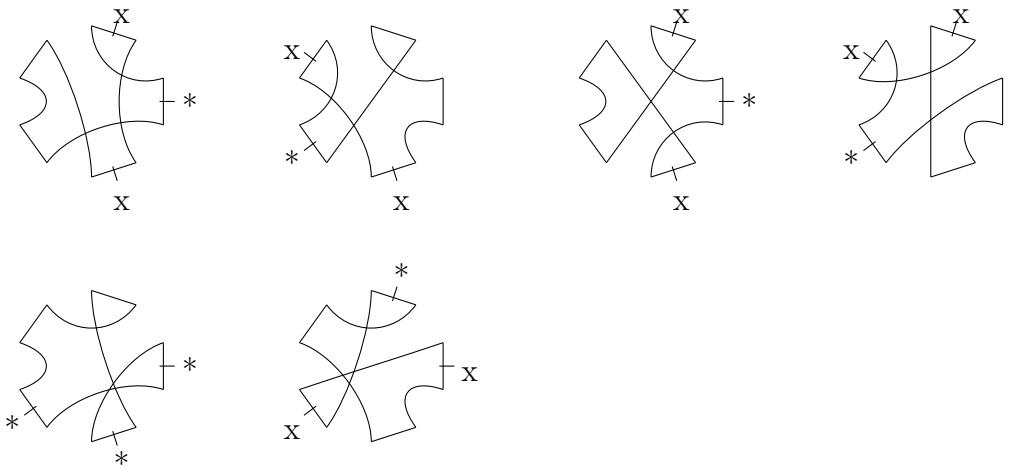


Figure 3.24: After eliminating the 3-cycle, the 5-cycle is t-oriented. All possible configurations are illustrated here. The marked moves transforms the cycles into a 3-cycle and two adjacencies.

Lemma 67. *Let π be a permutation and let c, d, and e be three cycles in its reality-desire diagram with the following properties:*

- *c is a nontwisted 3-cycle, d and e are 1-twisted 3-cycles,*
- *c is neither interleaving with d nor with e,*
- *c intersects the nontwisted chord of d,*
- *the chord of c that is not intersected by d intersects with any chord of e,*
- *d and e are not interleaving.*

Then there exists a $0_{tr}2_t2_{tr}$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$.

Remark. For this sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$.

Proof. There are three possible configurations for this case. For each of them, a sequence is described in Figure 3.25. □

The next lemma describes Case 19. It is an improvement of a lemma proven in [HS04], and some of the sequences have been taken from there.

Lemma 68. *If a permutation contains a 1-twisted pair, we can either apply Lemma 63, or we can apply a sequence with $w = 2w_t + w_r$ and $\Delta\sigma \geq 4$, or we can apply a $0_{tr}2_t2_t$-sequence or a $0_{tr}2_t2_{tr}$-sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = 0$, or we can apply a $0_r2_{tr}1_r$ sequence with $\Delta c_{odd} = 4$ and $\Delta c_{even} = -1$*

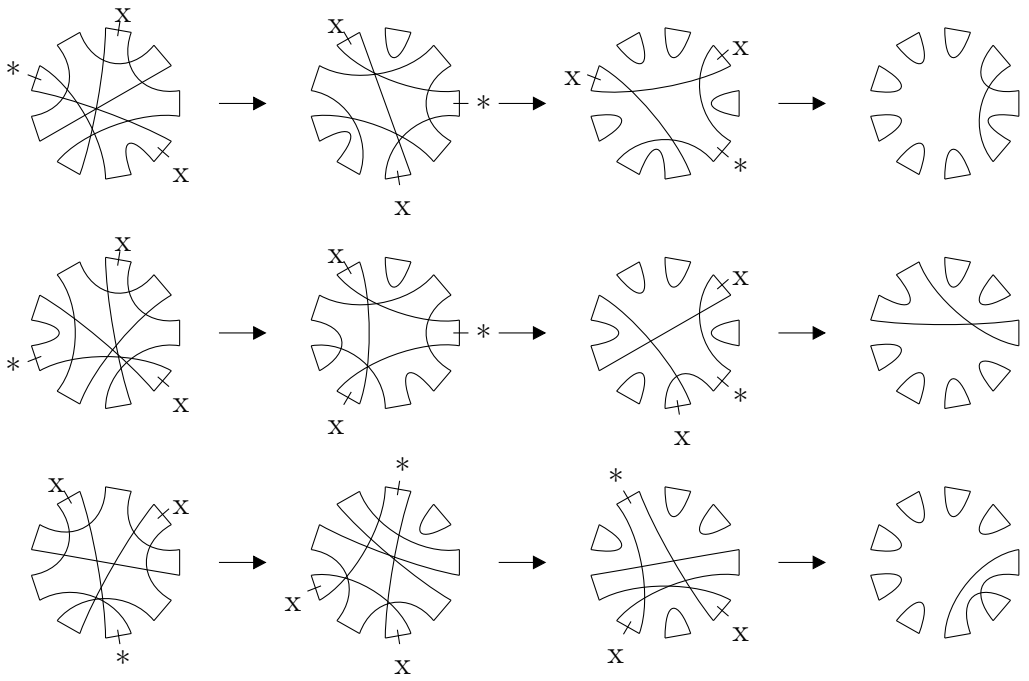


Figure 3.25: Sequences for the three possible configurations of Case 18.

Remark. For the sequences with $w = 2w_t + w_r$ and $\Delta\sigma \geq 4$, $\frac{\Delta\sigma}{w} \geq \frac{4}{2w_t+w_r}$. For the $0_{tr}2_t2_t$-sequence and $0_{tr}2_t2_{tr}$-sequence, $\frac{\Delta\sigma}{w} = \frac{4}{3w_t}$. For the $0_r2_{tr}1_r$-sequence, $\frac{\Delta\sigma}{w} = \frac{2w_r+2w_t}{w_t(2w_t+w_r)}$.

Proof. The starting configuration is illustrated in Figure 3.26. The arcs a_1 and a_2 in the figure are adjacent. According to Lemma 45, there must be a cycle e that has at least one reality-edge (let this be e_1) in one of the arcs, and at least one reality-edge (let this be e_2) that is in none of these arcs. Without loss of generality, we can assume that e_1 is in the arc a_1. Now, we must distinguish between the possible positions of e_2:

- If e_2 is in the arc a_3, we begin with a reversal that acts on e_1 and e_2. For all of the three possible positions of e_2, a sequence is described in Figure 3.27. Each sequence has the weight $w = 2w_t + w_r$. If the first reversal does not split the cycle e, the sequence increases c_{odd} by 4 and leaves c_{even} unchanged, therefore the gain in the score is 4. If the first reversal splits the cycle e, we either get one additional even cycle, or an even cycle will be split into two odd cycles. Both cases increase the score.

- If e_2 is not in the arc a_3, we can assume without loss of generality that e has no reality-edge in a_3. If e is 1-twisted and c or d intersects the nontwisted chord of e, then we can apply Lemma 63. Otherwise, there are six configurations we have to consider. For each of them, a sequence is described in Figure 3.28.

□

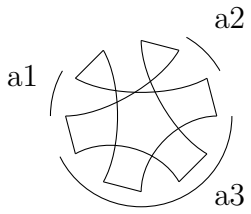


Figure 3.26: Two 1-twisted 3-cycles form a 1-twisted pair. The arcs a_1 and a_2 are adjacent, so there is a third cycle that has at least one reality-edge in a_1 or a_2, and at least one reality-edge that is neither in a_1 nor in a_2 (see Lemma 45). Depending on the position of this edge (in a_3 or between the two twisted reality-edges), we can choose a sequence for this case.

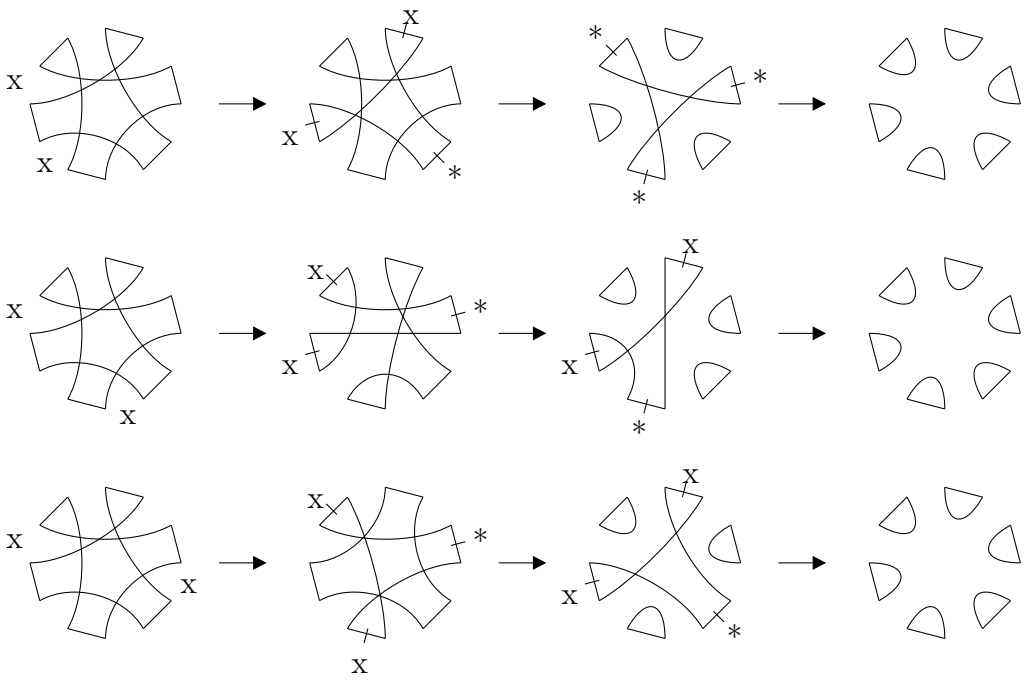


Figure 3.27: Sequences for eliminating a 1-twisted pair if there is a third cycle e that intersects at least one of the nontwisted chords of the 1-twisted pair. The first move is a reversal that acts on two reality-edges of e, the other two moves act on the reality-edges of the 1-twisted pair.

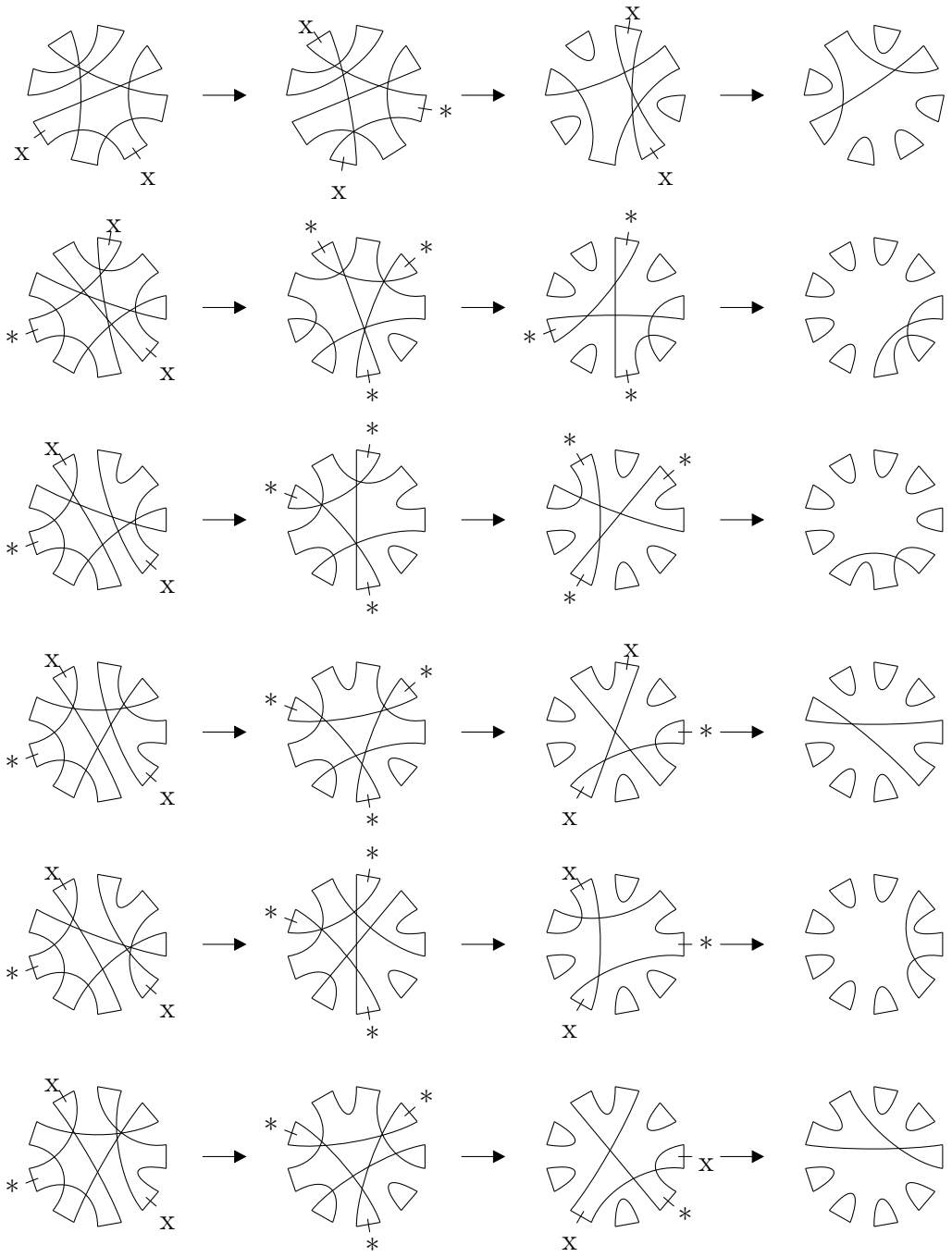


Figure 3.28: Sequences for two interleaving 3-cycles forming a 1-twisted pair. These are the sequences where we could not find a cycle that intersects one of the nontwisted chords of the 1-twisted pair.

Now, we have shown a sequence with $\frac{\Delta\sigma}{w} \geq \frac{3}{4w_t}$ for each case of the case analysis. These sequences are implemented in the algorithm, leading to a 1.5-approximation.

Chapter 4

The algorithms

4.1 The approximation algorithm

With the results of the previous chapters, we are now ready to develop the approximation algorithm, plus a further improvement that increases the approximation ratio in the practical tests.

4.1.1 The basic algorithm

The basic algorithm is designed to find a sequence s of operations that transforms a circular permutation π into another circular permutation $\hat{\pi}$. In the following, we will also call π the *source permutation* and $\hat{\pi}$ the *target permutation*. The weight of the sequence s shall be at most 1.5 times the weight of the optimal sequence that transforms π into $\hat{\pi}$. Due to the biological equivalence, it is also acceptable if the sequence sorts π into the reflection of $\hat{\pi}$. In cases where this is not acceptable, one can add a final reversal to the sequence in order to obtain the original target sequence. However, in this case, the approximation ratio of 1.5 cannot be guaranteed any longer. Practical tests show that most cases have a far better approximation ratio, and the last reversal does not push the ratio above 1.5 (see also Chapter 5).
The first step of the algorithm is to rename the elements of the permutations, so that the target permutation becomes the identity permutation. This may also change the orientation of some elements. An example can be seen in Figure 4.1. This step can easily be done in linear time and space: a linear walk over the target permutation creates a substitution table. Another linear walk over the source permutation renames the elements, using the substitution table.

The second step is the transformation of the source permutation into a simple permutation $\tilde{\pi}$. The algorithm is described in Section 2.6. It is important to store the mapping from the source permutation to the simple permutation, because we need this information to transform the sorting sequence of $\tilde{\pi}$ into a sorting sequence of π. Each insertion of a new element can be done in linear time. In the worst case, π contains only one cycle of length n. In this case, each step of the algorithm splits a 3-cycle from this cycle, and the

source $+1\ +7\ -3\ +2\ -5\ +6\ +4\ \rightarrow\ +5\ -2\ -1\ +7\ +4\ -3\ -6$
target $+3\ -7\ -6\ -5\ +1\ -4\ +2\ \rightarrow\ +1\ +2\ +3\ +4\ +5\ +6\ +7$

Substitution table:

+1	+2	+3	+4	+5	+6	+7
+5	+7	+1	-6	-4	-3	-2

Figure 4.1: First step: Rename the elements, so that the target permutation is the identity permutation.

length of the remaining cycle is decreased by 2. The algorithm stops when the remaining cycle has a length of 2 or 3. Therefore, the maximum number of splitting steps is $\frac{n-2}{2}$, and the whole splitting algorithm runs in $O(n^2)$. Note that the length of $\tilde{\pi}$ is $O(n)$.
The third step is searching for a sorting sequence for $\tilde{\pi}$. We begin the case analysis with the first cycle in $\tilde{\pi}$ that is not an adjacency, and then follow the decision tree illustrated in Tables 3.1 to 3.3. Note that although the tree contains some jumps to other positions, it does not contain any cycles, and therefore its depth can be bounded by a constant. Searching a cycle in the reality-desire diagram can always be done in linear time, so traversing the tree one time can be done in linear time. As the maximum number of operations is also linear in n, finding the sorting sequence can be done in $O(n^2)$.
The final step is to transform the sorting sequence of $\tilde{\pi}$ into a sorting sequence of π. As we have stored the mapping from π to $\tilde{\pi}$, the transformation of a single operation can be done in linear time (we just walk over π, and check wether the operation splits between the elements π_i and π_{i+1}). After each operation, the mapping must be updated. This can also be done in linear time, so the whole transformation can be done in $O(n^2)$.
We have shown that the whole algorithm has a running time of $O(n^2)$. The pseudo-code of the algorithm can be seen in Figure 4.2.

4.1.2 The Greedy algorithm

Although the basic algorithm fulfills the approximation ratio of 1.5 for any weight ratio from $w_r : w_t = 1 : 1$ to $w_r : w_t = 1 : 2$, it does not use the weights. In other words, changing the weight ratio will not change the resulting sequence. We will now combine the basic algorithm with a greedy strategy that takes the weight ratio into account. The algorithm is the same as the basic algorithm, except for the step where we choose the starting cycle for the case analyisis. Instead of using the first cycle in $\tilde{\pi}$ that is not an adjacency, we start the case analysis with each cycle in $\tilde{\pi}$. For each start cycle c_i, we get a sequence seq_i with weight w_i and a gain of the score $\Delta\sigma_i$. From these sequences, we apply the one where the ratio $\frac{\Delta\sigma_i}{w_i}$ is maximal. Note that we do not find the configuration with the best ratio $\frac{\Delta\sigma_i}{w_i}$ in any case; we are just varying the starting cycle of the case analysis, and take the first cycle we find when extending the configuration, and therefore can miss some configurations. Searching for any possible configuration would increase the running time of the algorithm too much.

```
void approx
{
    /* origin is the origin permutation
     * target is the target permutation
     * result is the resulting sequence; initially, it contains no operation

    // rename the elements
    substitutionTable = createSubstitutionTable(target);
    origin = subsititute(origin, substitutionTable);
    // create simple permutation
    simplePerm = createSimplePermutation(origin);
    // sort simplePerm
    while (simplePerm.numCycles != simplePerm.size)
    {
        cycle = getCycle(simplePerm);   // any cycle that is not an adjacency
        case = traverseDecisionTree(cycle); // the case at the end of the tree
        seq = solveCase(case);          // short sequence to solve this case
        simplePerm.performSequence(seq);
        result.push_back(seq);
    }
    // transform sequence for simplePerm into a sequence for origin
    transformSequence(result);
}
```

Figure 4.2: The basic approximation algorithm

As also the greedy algorithm uses only sequences that fulfill the approximation ratio of 1.5, it is easy to see that the whole algorithm is a 1.5-approximation. It is highly expected that with the greedy strategy that the results are better than the results of the basic algorithm. However, there are some cases where the result of the greedy algorithm is worse than that of the basic algorithm. These cases are very rare in practice, and the tests have shown that the greedy algorithm can improve the approximation ratio (see Chapter 5). The drawback of the greedy strategy is the longer running time. As the time complexity of traversing the decision tree is $O(n)$ and we start at each possible cycle, finding the best sequence seq_i has a time complexity of $O(n^2)$, and the whole algorithm has a time complexity of $O(n^3)$. The pseudo-code of the algorithm can be seen in Figure 4.3.

4.2 A branch and bound algorithm

For testing the approximation algorithm, it is not only of interest to test its approximation ratio against the lower bound. It would give much more insight to test its approximation ratio against the real weighted distance. Unfortunately, it is a hard task to determine this distance, and the complexity of the problem is unknown. In this section, we will provide a branch and bound algorithm that is able to calculate the weighted distance of small permutations. Although its running time is exponential in the worst case, the algorithm is far better than just testing all possible sequences up to a certain weight.
We begin by introducing the tree T_π, which contains all possible sortings of a permutation π:

Definition 69. *The* ***tree*** T_π *of a permutation* π *is defined as follows:*

- *Nodes in the tree are permutations* $\bar{\pi}$*, and the root node is the permutation* π*.*
- *If any node* $\bar{\pi}$ *is the identity permutation, it has no children. Otherwise, the children of* $\bar{\pi}$ *are the permutations* $op(\bar{\pi})$ *for each possible operation* op*.*
- *Each child* $op(\bar{\pi})$ *is connected by an edge with its parent node* $\bar{\pi}$*. The weight of the edge is* w_r *if* op *is a reversal, otherwise the weight is* w_t*.*
- *The weight* $w(\bar{\pi})$ *of a node* $\bar{\pi}$ *is the sum of the weights of the edges along the path from* $\bar{\pi}$ *to to the root node* π*.*

Each path in the tree corresponds to a sequence of operations. If the path starts at the root node π and ends at a leaf, the sequence is a sorting sequence for π. The weight of the leaf node is the weight of the sorting sequence. The branch and bound algorithm creates a part of the tree (the whole tree is infinitely large). If the algorithm reaches a leaf node, and it can be proven that any other possible leaf node has a higher weight, we have found the optimal sequence.
The main idea of the algorithm is to expand the node where we expect the leaf with the lowest weight. Therefore, we assign a bound to each node:

```
void approx
{
    /* origin is the origin permutation
     * target is the target permutation
     * result is the resulting sequence; initially, it contains no operation

    // rename the elements
    substitutionTable = createSubstitutionTable(target);
    origin = subsititute(origin, substitutionTable);
    // create simple permutation
    simplePerm = createSimplePermutation(origin);
    // sort simplePerm
    while (simplePerm.numCycles != simplePerm.size)
    {
        bestratio = 0;
        for (i = 0; i < n; i++)
        {
            cycle = getCycle(simplePerm, i);   // cycle with reality-edge i
            case = traverseDecisionTree(cycle); // the case at the end of the tree
            seq = solveCase(case);          // short sequence to solve this case
            // check if new best case
            if (seq.sigma / seq.weight > bestratio)
            {
                bestratio = seq.sigma / seq.weight
                bestseq = seq;
            }
        }
        simplePerm.performSequence(bestseq);
        result.push_back(bestseq);
    }
    // transform sequence for simplePerm into a sequence for origin
    transformSequence(result);
}
```

Figure 4.3: The greedy algorithm

Definition 70. *The bound $b(\pi)$ of a node π is defined as follows:*

$$b(\pi) = w(\pi) + c_{even}(\pi)w_r + (\frac{n - c_{odd}(\pi)}{2} - c_{even}(\pi))w_t$$

Lemma 71. *Let π be a permutation and let $\bar{\pi}$ its child in the tree. Then*

$$b(\pi) \leq b(\bar{\pi})$$

Proof. We begin with writing the bound in a slightly different way:

$$\begin{aligned} b(\pi) &= w(\pi) + c_{even}(\pi)w_r + (\frac{n - c_{odd}(\pi)}{2} - c_{even}(\pi))w_t \\ &= w(\pi) + \frac{w_t}{2}(n - c_{odd}(\pi) - 2c_{even}(\pi) + 2c_{even}(\pi)\frac{w_r}{w_t}) \\ &= w(\pi) + \frac{w_t}{2}(n - \sigma(\pi)) \end{aligned}$$

As we have seen in the proof of Theorem 23, $\frac{\Delta\sigma}{w} \leq \frac{2}{w_t}$. If the operation that transformed π into $\bar{\pi}$ has been a transposition or a transreversal, this would lead to

$$\begin{aligned} b(\pi) &= w(\pi) + \frac{w_t}{2}(n - \sigma(\pi)) \\ &\leq w(\pi) + \frac{w_t}{2}(n - \sigma(\bar{\pi}) + 2) \\ &= w(\pi) + w_t + \frac{w_t}{2}(n - \sigma(\bar{\pi})) \\ &= w(\bar{\pi}) + \frac{w_t}{2}(n - \sigma(\bar{\pi})) \\ &= b(\bar{\pi}) \end{aligned}$$

If the operation was a reversal, we have

$$\begin{aligned} b(\pi) &= w(\pi) + \frac{w_t}{2}(n - \sigma(\pi)) \\ &\leq w(\pi) + \frac{w_t}{2}(n - \sigma(\bar{\pi}) + 2\frac{w_r}{w_t}) \\ &= w(\pi) + w_r + \frac{w_t}{2}(n - \sigma(\bar{\pi})) \\ &= w(\bar{\pi}) + \frac{w_t}{2}(n - \sigma(\bar{\pi})) \\ &= b(\bar{\pi}) \end{aligned}$$

□

Note that the bound of a node is the weight we already used to come to this node, plus the minimum weight we use from here to the identity permutation (see Theorem 23).
For choosing the next node to expand, we always use the one with the lowest bound. If this

node is the identity permutation, the path from the root node to this node is an optimal sorting sequence, and we can stop the algorithm. To see this fact more clearly, let $b(\pi)$ be the bound of the identity permutation we just found. Any possible path to another leaf node $\hat{\pi}$ must pass by a node that is not yet expanded, and therefore has a bound $\geq b(\pi)$. Therefore, also $b(\hat{\pi}) \geq b(\pi)$. As for the leaf nodes, the bound is equal to the weight of the sorting sequence, there cannot exist any sorting sequence with a weight $< b(\pi)$. The whole algorithm is described in Figure 4.4.

The running time mainly depends on the size s of the tree. Using a heap, it is possible to insert new elements and getting the element with the best bound in $O(\log s)$ time. However, in the worst case, $s = O(n!)$, and therefore the time complexity of inserting an element or removing the top element is $O(n \log n)$. As we need this step for each element we want to insert, we get an overall time complexity of $O(n! \cdot n \cdot \log n)$, which is a very fast growing exponential function. Moreover, the space complexity is $O(n! \cdot n)$ (as we have to store the whole tree). However, the algorithm performs well for permutations of small sizes (see also Chapter 5).

```
void branchandbound
{
    /* heap is a heap of permutations, and initial contains no elements
     * perm is a permutation, and contains the weight of its tree node and its
     * bound as additional data
     * child is another permutation
     * origin is the permutation to sort
     * w_r and w_t are the weights for reversals and transpositions /
     * transreversals */

    heap.push(origin);
    while (true)
    {
        perm = heap.top();
        heap.pop();
        if (perm is identity)
        {
            outputSequence();
            return;
        }
        for (all reversals r(i, j))
        {
            child = r(i, j) perm;
            child.weight = perm.weight + w_r;
            child.calculateBound();
            heap.push(child);
        }
        for (all transpositions t(i, j, k))
        {
            child = t(i, j, k) perm;
            child.weight = perm.weight + w_t;
            child.calculateBound();
            heap.push(child);
        }
        for (all transreversals tr(i, j, k))
        {
            child = tr(i, j, k) perm;
            child.weight = perm.weight + w_t;
            child.calculateBound();
            heap.push(child);
        }
    }
}
```

Figure 4.4: The branch and bound algorithm in pseudocode. Printing the result can be done very efficient if we store for each node $\bar{\pi}$ the predecessor and the operation that led to the node $\bar{\pi}$.

Chapter 5

Practical results

Both the approximation algorithm and the branch and bound algorithm have been tested with different testing sets and different parameters for w_r and w_t. For each test run, we calculated the performance of the approximation, and also the running time of the different algorithms has been compared. The results were also compared with the output of DERANGE II, a program to solve sorting by weighted transpositions and reversals developed by Blanchette, Kunisawa, and Sankoff [BKS96].

5.1 The test sets

We tested the programs with two different kinds of permutations:

random permutations: These are test sets with uniformly distributed random signed permutations of fix size.

low distance permutations: These are test sets with a low distance to the identity permutation. They were generated by applying a small sequence of operations to the identity permutation. In practice, low distance permutations are much more interesting than random permutations, because we mostly want to compare the genomes of similar species, and it is highly expected that their genomes have a low distance.

For each kind of permutation, we have created test sets with different permutation sizes. The used permutation sizes are 8, 16, 50, 100, 500, 1000, and 4000. Each test set contains 100 test cases. Each of these sets has been tested with different ratios for $w_r : w_t$. We used the ratios $1 : 1$, $1 : 1.5$, and $1 : 2$.

5.1.1 How to generate low distance permutations

When generating low distance permutations, the first question is: What is the maximum distance where we speak of a low distance? Up to which distance can we say that two genomes are similar, and beyond which distance can we say that each similarity is purely

random? To solve this, we should examine the expected distance for a random permutation. We will do this by using some elementary results of Kolmogorov complexity:
First, let us count the number of different permutations of size n: There are $n!$ unsigned permutations, and for each element, we can use an arbitrary sign, so we have to multiply this value by 2^n. As we work with cyclic permutations, and we assume that a permutation is equivalent to its reflection, the permutations decompose into sets of equivalent permutations, and each set has a size of $2n$. Therefore, we have $k = \frac{2^n n!}{2n}$ different permutations. How many bits do we need to uniquely describe an element of a set with size k?

Lemma 72. *Let S be a set of size $k = 2^l$, and let $f : S \to \{0,1\}^*$ a function that assigns for each element $s \in S$ a unique bitstring. Then, the expected length of $f(s)$ for a random element $s \in_R S$ is*

$$E(|f(s)|) \geq \log k - 2$$

Proof. [Sch95] In the worst case, we have one element s with $|f(s)| = 0$, two with $|f(s)| = 1$, four with $|f(s)| = 2$, and so on (note that there do not exist more different bitstrings of these lengths). Finally, we have $\frac{k}{2}$ bitstrings of length $\log k - 1$, and one of length $\log k$. Building the sum over this, and dividing it by the size of S, we can estimate the length of $f(s)$:

$$\begin{aligned}
E(|f(s)|) &\geq \frac{1}{k}(\log k + \sum_{i=0}^{\log k - 1} i2^i) && (5.1)\\
&= \frac{\log k}{k} + \sum_{i=0}^{\log k - 1} i2^i && (5.2)\\
&= \frac{\log k}{k} + \sum_{j=1}^{\log k} (\log k - j)2^j && (5.3)\\
&\geq \frac{\log k}{k} + \sum_{j=1}^{\log k} \log k 2^{-j} - \sum_{h=0}^{\infty} h2^{-h} && (5.4)\\
&= \log k - \sum_{h=0}^{\infty} h2^{-h} && (5.5)\\
&= \log k - 2 && (5.6)
\end{aligned}$$

□

Using this formula, we see that the description of a random permutation of our set has an expected length of at least $\log k - 2 \approx n \log n$. Note that this holds for any possible unique description for the elements. Now, we describe a permutation π by the sorting sequence that our algorithm returns for π. For each operation in the sequence, we have to describe the operation type and the positions of at most three reality-edges on which the operation acts. To describe the type, two bits are sufficient. For each reality-edge, we need

$\log n$ bits. Therefore, for an operation, we need at most $3 \log n + 2$ bits. Now, let π be a random permutation, so that the description of π has a length of $n \log n$. Let $seq(\pi)$ be a sorting sequence of π, and let $|seq(\pi)|$ the number of operations in $seq(\pi)$. Then,

$$|seq(\pi)| \geq \frac{n \log n}{3 \log n + 2} \approx \frac{n}{3}$$

However, this estimation is not very close. The set of all sorting sequences with a length of $\frac{n}{3}$ leads to many duplicate permutations (imagine the second operation undoes the effect of the first operation). Therefore, we define that each permutation with a weighted distance of $\frac{n}{3} w_t$ to another permutation $\hat{\pi}$ has a low distance to $\hat{\pi}$.
When creating random sequences of operations, we should also pay attention to the weights w_r and w_t. If $w_r < w_t$, a random sequence should contain more reversals than transpositions and transreversals. Let s_r be a sequence that consists only of reversals, and s_t a sequence that consists only of transpositions and transreversals. When creating the random sequences, our goal is that if s_r and s_t have the same weight, the probability of getting s_r is the same as the probability getting s_t.

Lemma 73. *Let $P(r)$ be the probability that we choose a reversal, and let $P(t)$ the probability that we choose a transposition or a transreversal as operation in a random sequence. Then, the probability of getting a sequence with weight w that consists only of reversals is equal to the probability to get a sequence with the same weight w that consists only of transpositions and transreversals if the following equation holds:*

$$P(r)^{\frac{w_t}{w_r}} + P(r) - 1 = 0$$

Proof. Without loss of generality, we will assume that w_r and w_t are integer. A sequence s_r of w_t reversals has the same weight as a sequence s_t of w_r transpositions and transreversals (for both, $w = w_r w_t$). The probability of sequence s_r is $P(s_r) = P(r)^{w_t}$, and the probability of sequence s_t is $P(s_t) = P(t)^{w_r}$. With $P(s_r) = P(s - t)$, we get

$$P(r)^{\frac{w_t}{w_r}} = P(t)$$

As the sum of the probabilities must be 1, we have

$$P(t) = 1 - P(r)$$

Combining these equations leads to

$$P(r)^{\frac{w_t}{w_r}} + P(r) - 1 = 0$$

□

The algorithm for creating the low distance permutation works as follows: We begin with an empty sequence, and add one operation in each step. For each operation, we first decide if it is a reversal or not, using the probabilities $P(r)$ and $P(t)$. Note that we use

only three different ratios of $w_r : w_t$, so we can precalculate these values. If we decide to not take a reversal, the choice between transposition and transreversal is done with equal probabilities. The last step is to choose the reality-edges on which the operation acts. For this, we take a random function to pick one operation of the specfied type uniformly distributed out of all possible operations of this type. We keep adding random operations to the sequence until the weight of the sequence exceeds $\frac{n}{3}w_t$. The pseudo-code for this algorithm can be seen in Figure 5.1. Note that the resulting permutations have a maximum distance to the identity permutation of about $\frac{n}{3}w_t$, but they can also have a much smaller distance, because the inverse of the generated sequence is in general not the shortest sorting sequence.

5.2 The programs

approx: The 1.5-approximation algorithm, without using the greedy strategy.

greedy: The 1.5-approximation algorithm with the greedy strategy.

derange: A program to solve Sorting by weighted transpositions and reversals, developed by Blanchette, Kunisawa, and Sankoff [BKS96]. It works on the breakpoint distance and uses a greedy strategy with lookahead. It is also capable to use different weights for transpositions and transreversals, and can handle unsigned and linear permutations as well. In our tests, we used the recommended lookahead of 3.

branch: The branch and bound algorithm.

5.3 The test results

Each algorithm has been tested with each kind of test set, using the different weight ratios $w_r : w_t = 1 : 1$, $w_r : w_t = 1 : 1.5$, and $w_r : w_t = 1 : 2$. All tests were performed on a Sun Fire 280 with two processors (Ultra Sparc 3 CU, 1.015 GHz) and 6 GB RAM.
Due to its exponential running time, the branch and bound algorithm was just tested with permutations of size 8 and 16. DERANGE II was tested with permutations up to a size of 500. Above this size, the program crashed.
The test results are represented in the tables 5.1 to 5.24. The interpretation of the test results will be given in Section 5.4. In the following the column headings of the tables are explained.

size: Size of the permutations

dist: The averaged calculated weighted distance.

w/b **avg:** Average ratio $\frac{w}{b}$, where w is the calculated weight of a sequence, and $b = c_{even}w_r + (\frac{n-c_{odd}}{2} - c_{even})w_t$ is the lower bound for the weighted distance.

```
sequence createLowdist
{
    /* w_r and w_t are the weights for reversals and transpositions /
     *   transreversals
     * pr is the probability of choosing a reversal
     * seq is a sequence of operations (initial empty sequence)
     * op is an operation in the sequence
     * random() returns a float value uniformly distributed in [0; 1)
     * randreversal, randtransposition, randtransreversal return a random
     *   operation of the specified type, uniformly distributed */

    if (w_r / w_t == 1)
        pr = 0.5;                    // precalculated value
    else if (w_r / w_t == 1.0 / 1.5)
        pr = 0.5698;                 // precalculated value
    else if (w_r / w_t == 1.0 / 2.0)
        pr = 0.6180;                 // precalculated value
    while (w < n * w_t / 3)
    {
        if (random() < pr)
        {
            op = randreversal();
            w += w_r;
        }
        else
        {
            if (random() < 0.5)
                op = randtransposition();
            else
                op = randtransreversal();
            w += w_t;
        }
        seq.add(op);
    }
    return seq id;      // apply seq on the identity permutation
}
```

Figure 5.1: The algorithm to create low distance permutations.

w/b **worst:** The worst ratio $\frac{w}{b}$ over all 100 permutations of the test set, with w and b as described above.

w/d_w **avg:** Average ratio $\frac{w}{d_w}$, where w is the calculated weight of a sequence, and d_w is the weight of an optimal sorting sequence. As we need the branch and bound algorithm to calculate d_w, this values could only be calculated for permutations of size 8 and 16.

w/d_w **worst:** The worst ratio $\frac{w}{d_w}$ over all 100 permutations of the test set, with w and d_w as described above. As we need the branch and bound algorithm to calculate d_w, this values could only be calculated for permutations of size 8 and 16.

best: The number of test cases where the corresponding algorithm returned the shortest sorting sequence. If two algorithms return a sorting sequence with the same weight, both of them are considered to be the best for this test case. Therefore, the sum of this value for all algorithms can be greater than 100 for one test set. As the branch and bound algorithm always returns the best possible result, it is not considered in this statistic.

time: The average time for sorting one permutation. The format is minutes:seconds, with an accuracy of 1/10 second.

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	3.81	1.0992	1.333	1.0758	1.333	74	0:00.0
16	8.48	1.140	1.429	1.138	1.429	49	0:00.0
50	27.51	1.139	1.280			3	0:00.0
100	56.46	1.149	1.256			0	0:00.0
500	285.99	1.149	1.193			0	0:00.1
1000	573.27	1.150	1.204			0	0:00.5
4000	2297.57	1.150	1.164			0	0:07.7

Table 5.1: Random permutations, approx, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	4.95	1.0476	1.333	1.0328	1.333	75	0:00.0
16	11.43	1.0753	1.238	1.0747	1.238	42	0:00.0
50	37.94	1.0694	1.176			3	0:00.0
100	78.66	1.0786	1.158			0	0:00.0
500	401.84	1.0786	1.107			0	0:00.1
1000	805.66	1.0788	1.106			0	0:00.5
4000	3234.17	1.0792	1.0893			0	0:08.6

Table 5.2: Random permutations, approx, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	6.08	1.0175	1.333	1.00952	1.333	98	0:00.0
16	14.38	1.0405	1.167	1.0405	1.167	72	0:00.0
50	48.36	1.0335	1.133			48	0:00.0
100	100.86	1.0427	1.124			11	0:00.0
500	517.69	1.0435	1.0730			0	0:00.1
1000	1038.04	1.0432	1.0612			0	0:00.5
4000	4170.76	1.0440	1.0541			0	0:08.9

Table 5.3: Random permutations, approx, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	1.99	1.030	1.500	1.0200	1.500	96	0:00.0
16	4.24	1.0763	1.500	1.0730	1.500	77	0:00.0
50	12.64	1.143	1.364			17	0:00.0
100	25.81	1.152	1.318			0	0:00.0
500	127.26	1.153	1.234			0	0:00.0
1000	256.58	1.154	1.195			0	0:00.2
4000	1025.55	1.155	1.177			0	0:02.3

Table 5.4: Low distance permutations, approx, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	3.105	1.0427	1.5	1.0258	1.286	91	0:00.0
16	6.705	1.0505	1.286	1.0486	1.286	61	0:00.0
50	20.595	1.0699	1.220			13	0:00.0
100	42.92	1.0799	1.210			0	0:00.0
500	214.28	1.0817	1.143			0	0:00.1
1000	429.50	1.0805	1.133			0	0:00.2
4000	1721.99	1.0822	1.0975			0	0:03.3

Table 5.5: Low distance permutations, approx, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	4.77	1.0175	1.333	1.0150	1.333	95	0:00.0
16	9.59	1.0197	1.286	1.0197	1.286	89	0:00.0
50	31.74	1.0340	1.200			54	0:00.0
100	64.19	1.0379	1.127			23	0:00.0
500	326.78	1.0408	1.1762			0	0:00.1
1000	651.17	1.0382	1.0666			0	0:00.3
4000	2617.29	1.0402	1.0542			0	0:04.2

Table 5.6: Low distance permutations, approx, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	3.77	1.0875	1.333	1.0642	1.333	78	0:00.0
16	8.05	1.0830	1.333	1.0811	1.286	73	0:00.0
50	24.65	1.0207	1.0833			71	0:00.0
100	49.76	1.0130	1.0426			71	0:00.0
500	249.69	1.00285	1.0121			65	0:00.3
1000	499.30	1.00138	1.00402			100	0:01.1
4000	1999.17	1.000230	1.00100			100	0:19.2

Table 5.7: Random permutations, greedy, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	4.93	1.0432	1.333	1.0283	1.273	79	0:00.0
16	11.11	1.0452	1.222	1.0446	1.190	67	0:00.0
50	35.825	1.00994	1.0541			70	0:00.0
100	73.395	1.00645	1.0280			65	0:00.0
500	373.08	1.00144	1.00672			60	0:00.3
1000	747.36	1.000710	1.00269			100	0:01.1
4000	2997.23	1.000110	1.000670			100	0:18.8

Table 5.8: Random permutations, greedy, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	6.06	1.0142	1.333	1.00619	1.333	99	0:00.0
16	14.08	1.0186	1.154	1.0186	1.154	87	0:00.0
50	46.91	1.00249	1.0612			94	0:00.0
100	96.93	1.00207	1.0213			89	0:00.0
500	496.33	1.000440	1.00602			87	0:00.3
1000	995.28	1.000230	1.00401			100	0:01.0
4000	3995.31	1.000060	1.00100			100	0:14.8

Table 5.9: Random permutations, greedy, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	1.99	1.0300	1.500	1.0200	1.500	96	0:00.0
16	4.11	1.0473	1.333	1.0440	1.333	90	0:00.0
50	11.53	1.0433	1.200			69	0:00.0
100	22.84	1.0205	1.150			74	0:00.0
500	110.78	1.00388	1.0273			73	0:00.1
1000	222.76	1.00192	1.0177			100	0:00.3
4000	887.95	1.000440	1.00228			100	0:05.2

Table 5.10: Low distance permutations, greedy, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	3.11	1.0427	1.500	1.0258	1.286	91	0:00.0
16	6.57	1.0291	1.273	1.0273	1.273	82	0:00.0
50	19.62	1.0196	1.142			70	0:00.0
100	40.05	1.00769	1.050			75	0:00.0
500	198.50	1.00204	1.0101			67	0:00.1
1000	397.90	1.00103	1.00753			100	0:00.5
4000	1591.71	1.000280	1.00220			100	0:07.4

Table 5.11: Low distance permutations, greedy, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	4.74	1.0108	1.333	1.00833	1.333	97	0:00.0
16	9.50	1.0105	1.286	1.0105	1.286	94	0:00.0
50	30.88	1.00596	1.138			90	0:00.0
100	62.01	1.00260	1.0333			89	0:00.0
500	314.14	1.000600	1.00637			86	0:00.1
1000	627.46	1.000430	1.00480			100	0:00.6
4000	2516.33	1.000060	1.000800			100	0:08.2

Table 5.12: Low distance permutations, greedy, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	3.70	1.0783	1.333	1.0550	1.333	85	0:00.0
16	7.89	1.0638	1.333	1.0619	1.167	85	0:00.0
50	24.59	1.0184	1.0909			74	0:00.2
100	49.64	1.0107	1.0426			80	0:02.4
500	249.43	1.00181	1.00806			82	23:14.0

Table 5.13: Random permutations, derange, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	4.91	1.0424	1.375	1.0277	1.222	82	0:00.0
16	10.90	1.0257	1.167	1.0251	1.105	82	0:00.0
50	35.72	1.00695	1.0417			76	0:00.2
100	73.18	1.00345	1.0144			81	0:03.2
500	372.82	1.000720	1.00268			77	33:04.0

Table 5.14: Random permutations, derange, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	6.07	1.0173	1.500	1.00886	1.400	98	0:00.0
16	13.83	1.000830	1.0833	1.000830	1.0833	99	0:00.0
50	46.79	1.000	1.000			100	0:00.2
100	96.75	1.000210	1.0206			99	0:03.0
500	496.13	1.000040	1.00403			99	35:29.7

Table 5.15: Random permutations, derange, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	2.20	1.135	1.500	1.125	1.500	75	0:00.0
16	4.27	1.0993	1.500	1.0960	1.500	74	0:00.0
50	11.46	1.0383	1.200			74	0:00.0
100	22.68	1.0132	1.0526			83	0:00.0
500	110.63	1.00257	1.0188			83	0:01.6

Table 5.16: Low distance permutations, derange, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	3.24	1.0891	1.500	1.0722	1.333	66	0:00.0
16	6.65	1.0415	1.286	1.0396	1.286	69	0:00.0
50	19.62	1.0195	1.118			62	0:00.0
100	40.08	1.00848	1.0380			66	0:00.1
500	198.395	1.00151	1.00759			71	0:14.0

Table 5.17: Low distance permutations, derange, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	w/d_w avg	w/d_w worst	best	time
8	4.69	1.00250	1.250	1.000	1.000	100	0:00.0
16	9.41	1.000	1.000	1.000	1.000	100	0:00.0
50	30.7	1.000	1.000			100	0:00.0
100	61.85	1.000	1.000			100	0:00.3
500	313.95	1.000	1.000			100	0:45.8

Table 5.18: Low distance permutations, derange, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	time
8	3.53	1.0233	1.333	0:00.0
16	7.45	1.0067	1.167	0:07.8

Table 5.19: Random permutations, branch, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	time
8	4.78	1.0139	1.222	0:00.0
16	10.635	1.000560	1.0556	0:04.5

Table 5.20: Random permutations, branch, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	time
8	6.02	1.00750	1.250	0:00.0
16	13.82	1.000	1.000	0:10.3

Table 5.21: Random permutations, branch, $w_r : w_t = 1 : 2$

size	dist	w/b avg	w/b worst	time
8	1.95	1.0100	1.500	0:00.0
16	6.395	1.00183	1.100	0:00.3

Table 5.22: Low distance permutations, branch, $w_r : w_t = 1 : 1$

size	dist	w/b avg	w/b worst	time
8	3.02	1.0164	1.500	0:00.0
16	6.40	1.00183	1.100	0:00.3

Table 5.23: Low distance permutations, branch, $w_r : w_t = 1 : 1.5$

size	dist	w/b avg	w/b worst	time
8	4.69	1.00250	1.250	0:00.0
16	9.41	1.000	1.000	0:01.6

Table 5.24: Low distance permutations, branch, $w_r : w_t = 1 : 2$

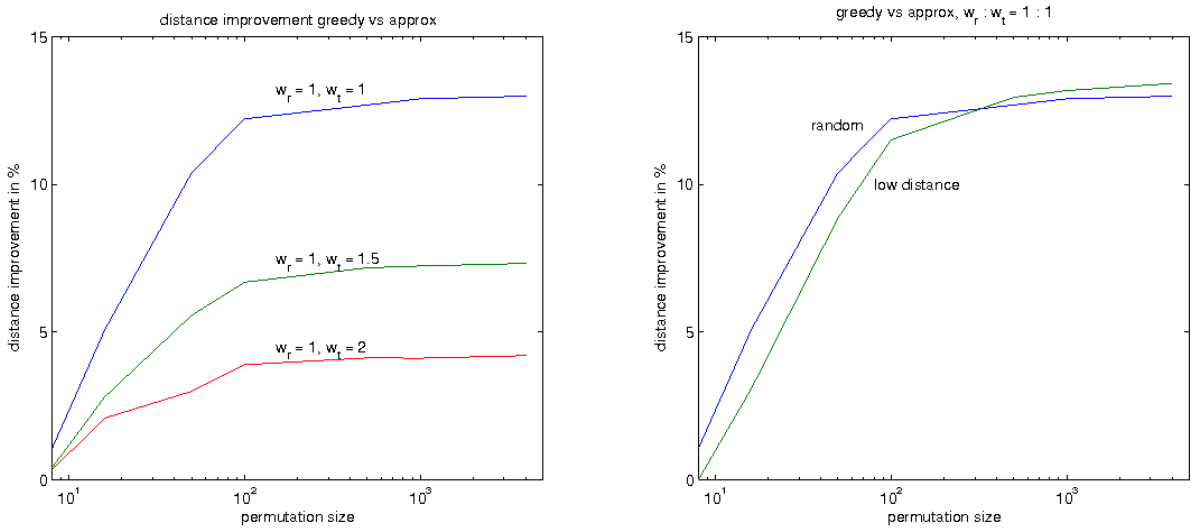


Figure 5.2: The improvement of the average distance due to the use of the greedy strategy. The left picture shows the improvement for random permutations and the different weight ratio. The right picture shows the improvement for a low distance permutation and a random permutation a t a weight ratio of $w_r : w_t = 1 : 1$. Note that the improvement depends mainly on the distance, what explains the lower improvement for the low distance permutations with small permutation sizes.

5.4 Interpretation of the test results

We will now have a closer look at the test results. First, we look at the average ratio $\frac{w}{b}$. For *approx*, this value is independent of the permutation size (except for very small permutations). It also shows no big differences between random permutations and low distance permutations, and depends only on the weight ratio $w_r : w_t$. We get the worst values for $w_r : w_t = 1 : 1$ (about 1.15), and they get better for a higher weight of w_t (up to about 1.04 for $w_r : w_t = 1 : 2$).
If we use *greedy*, the ratio $\frac{w}{b}$ becomes better for bigger permutations, and comes very close to 1. Also here, a high value for w_t decreases the ratio. If we compare the distances calculated by *approx* with the distances calculated by *greedy*, one can see that using the greedy strategy gives a significant improvement to the basic approximation algorithm. From the discussion above it is clear that this improvement becomes more significant for permutations of large size and low weights for w_t (the improvement is 13.42% for a permutation size of 4000, a weight ratio of 1:1, and low distance permutations). However, it is interesting to see that the improvement grows faster with increasing permutation size if we use low distance permutations instead of random permutations (see Figure 5.2). This means that the improvement through the greedy strategy is bigger if we examine low distance permutations. If we compare *derange* with *greedy*, we can see that the results

are very similar, with a slight advantage to *derange*. However, in no test set, the average distance *derange* gave was more than 2% better than the one given by *greedy*, and for big permutations, the difference becomes very small. *Derange* seems also to provide very bad results if the distances between the permutations are small (especially permutations of size 8 and low distance permutations). This has the following reason: while the greedy algorithm sorts a permutation to the identity or its reflection, DERANGE II always sorts to the identity permutation. Due to this, it is possible that the optimal sequence for DERANGE II needs one reversal more than the optimal sequence for the greedy algorithm. For small distances, this can adulterate the results significantly. For big distances, this disadvantage is not significant. It is also interesting to see that for low distance permutations and a weight ratio of $w_r : w_t = 1 : 2$, DERANGE II has found an optimal solution for all test cases ($\frac{w}{d_w} = 1$ or $\frac{w}{b} = 1$). However, DERANGE II is slow. The approximation algorithm (also with the slower greedy strategy) can solve much bigger permutations in the same time.

Chapter 6

Conclusion and open problems

We have developed an algorithm for sorting by weighted transpositions and reversals with a provable approximation performance of 1.5. In practice, it performs much better in the most cases. It has a low time complexity ($O(n^2)$) and therefore can handle bigger permutations than DERANGE II. However, there are still possible improvements and open questions:

- Although our algorithm is much faster than DERANGE II, the resulting sequences of DERANGE II are slightly better than ours. This is achieved by the lookahead used in DERANGE II. Therefore it would be of interest to implement a lookahead in the greedy version of our program. As a lookahead would increase the time complexity, the main task will be to find a good balance between the running time and the quality of the resulting sequences.

- Examining configurations with more cycles could improve the approximation ratio. Using this strategy, Elias and Hartman [EH05] recently succeeded in improving the performance ratio for sorting by transpositions from 1.5 to 1.375. It is highly expected that this strategy can also improve the performance ratio of sorting by weighted reversals, transpositions, and transreversals.

- Using a special data structure described in [KV03], it is possible to find the different cases in sublinear time ($O(n \log n)$). Although this will not improve the time complexity of the whole algorithm, it could at least speed up some parts of the algorithm.

- When adapting the algorithm to linear permutations, the operation *revrev* is needed. This operation is biologically only poorly motivated. Finding a way to avoid any revrev in the sorting sequence would result in an algorithm that corresponds better to the biological problem.

Bibliography

[BBD+99] D.W. Burt, C. Bruley, I.C. Dunn, C.T. Jones, A. Ramage, A.S. Law, D.R. Morrice, I.R. Paton, J. Smith, D. Windsor, A. Sazanov, R. Fries, and D. Waddington. The dynamics of chromosome evolution in birds and mammals. *Nature*, 402:411–413, 1999. 5

[Ber05] A. Bergeron. A Very Elementary Presentation of the Hannenhalli-Pevzner Theory. *Discrete Applied Mathematics*, 146(2):134–145, 2005. 7

[BKS96] M. Blanchette, T. Kunisawa, and D. Sankoff. Parametric genome rearrangement. *Gene*, 172:GC11–17, 1996. 8, 73, 76

[BMS04] A. Bergeron, J. Mixtacki, and J. Stoye. Reversal Distance without Hurdles and Fortresses. In *Combinatorial Pattern Matching 15th Annual Symposium, CPM 2004)*, volume Volume 3109 / 2004 of *Lecture Notes in Computer Science*, pages 388–399, 2004. 7

[BP96] V. Bafna and P.A. Pevzner. Genome Rearrangements and Sorting by Reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996. 15

[Bro02] T.A. Brown. *Genomes Second Edition.* BIOS Scientific Publishers Ltd, 2002. 1, 3, 4

[Cap97] A. Caprara. Sorting by reversals is difficult. In S. Istrail, P.A. Pevzner, and M. Waterman, editors, *Proc. 1st Annual Int. Conf. on Computational Molec. Biol. (RECOMB'97)*, pages 75–83. ACM, 1997. 7

[DS38] T.H. Dobzhansky and A.H. Sturtevant. Inversions in the chromosomes of Drosophila Pseudoobscura. *Genetics*, 23(1):28–64, January 1938. 7

[EH05] I. Elias and T. Hartman. A 1.375-Approximation Algorithm for Sorting by Transpositions. In *Proc. of 5th International Workshop on Algorithms in Bioinformatics*, volume 3692 of *Lecture Notes in Bioinformatics*, pages 204–215. Springer-Verlag, 2005. 8, 89

[Eri02] N. Eriksen. $(1 + \epsilon)$-approximation of Sorting by Reversals and Transpositions. *Theoretical Computer Science*, 289(1):517–529, 2002. 8

[GPS99] Q.-P. Gu, S. Peng, and H. Sudborough. A 2-Approximation Algorithms for Genome Rearrangements by Reversals and Transpositions. *Theoretical Computer Science*, 210(2):327–339, Jan 1999. 18

[Har03] T. Hartman. A Simpler 1.5-Approximation Algorithm for Sorting by Transpositions. In *CPM: 14th Symposium on Combinatorial Pattern Matching*, 2003. 33, 43, 46, 49

[HP99] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Jrnl. A.C.M.*, 46(1):1–27, Jan 1999. 36th Annual IEEE Symp. on Foundations of Comp. Sci., pp581-592, 1995. 7

[HS04] T. Hartman and R. Sharan. A 1.5-Approximation Algorithm for Sorting by Transpositions and Transreversals. In *WABI: International Workshop on Algorithms in Bioinformatics, WABI, LNCS*, 2004. 8, 13, 14, 28, 29, 30, 43, 46, 49, 55, 57, 58, 59

[KV03] H. Kaplan and E. Verbin. Efficient Data Structures and a New Randomized Approach for Sorting Signed Permutations by Reversals. In *Proc. of 14th Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 170–185. Springer-Verlag, 2003. 89

[LX01] G.-H. Lin and G. Xue. Signed genome rearrangement by reversals and transpositions: models and approximations. *TCS: Theoretical Computer Science*, 259:513–531, 2001. 8, 22

[Sch95] U. Schöning. *Perlen der Theoretischen Informatik*. BI Wissenschaftsverlag, Mannheim, 1995. 74

[SM97] J.C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, Boston, 1997. 1, 6, 15

[TS04] E. Tannier and M.-F. Sagot. Sorting by Reversals in Subquadratic Time. In *Proc. of the 15th Annual Symposium on Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2004. 7

[WDM98] M.E.M.T Walter, Z. Dias, and J. Meidanis. Reversal and Transposition Distance of Linear Chromosomes. In *Proc. of the Symposium on String Processing and Information Retrieval*, pages 96–102. IEEE Computer Society, 1998. 8

Erklärung

Martin Bader Matr.Nr.: 443062

Hiermit erkläre ich, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den