Universität Ulm
Institut für Theoretische Informatik
Leiter: Prof. Dr. Uwe Schöning

Genome Rearrangement Algorithms

Dissertation

zur Erlangung des Doktorgrades Dr.rer.nat.
der Fakultät für Ingenieurwissenschaften und Informatik der Universität Ulm

vorgelegt von
Martin Bader
aus Friedrichshafen

2011

Amtierender Dekan:	Prof. Dr. Klaus Dietmayer
Gutachter:	Prof. Dr. Enno Ohlebusch Prof. Dr. Jacobo Torán Prof. Dr. Jens Stoye
Tag der Promotion:	

Summary

With the increasing amount of sequenced genomes, a comparison of species based on these data becomes more and more interesting. In contrast to the classical approach, where only point mutations were considered, genome rearrangement problems ignore small mutations and only consider large-scale mutations that change the gene order on the chromosomes. This makes these problems a powerful tool when studying organisms that have diverged millions of years ago, or very fast evolving genomes, like those of cancerous cells.
A large variety of problems arises from genome rearrangements, like the calculation of evolutionary distances, the reconstruction of evolutionary events and the gene order of hypothetical ancestors, or even the reconstruction of whole phylogenetic trees. Due to the high complexity of most of these problems, the algorithms are based on highly simplified models of the reality. For example, most algorithms consider only a single type or a small set of evolutionary events. Therefore, one biological problem results in several algorithmic problems, depending on the chosen model.
Genome rearrangement problems are often very challenging. For many problems, it is not known whether they can be solved exactly in polynomial time w.r.t. the size of the genomes (like *sorting by transpositions*), others are known to be NP-hard even for the most simple models, like the *median problem* or the *multiple genome rearrangement problem*. For some problems, their complexity depends on the chosen model. Many of the problems can be solved in polynomial time if every gene occurs in each genome exactly once, but become NP-hard as soon as duplications are allowed.
From a computer scientist's point of view, this raises both theoretical and algorithmic tasks. For each problem, it is desirable either to find an exact algorithm or to prove that the problem is NP-hard. In some cases, the examination of a simplified version of the problem and the connection between this simplified problem and the actual problem can be helpful (actually, this approach helped Hannenhalli and Pevzner to obtain their famous result about *sorting by reversals*). If a problem has been shown to

be NP-hard, still many instances of the problem can be solved exactly and efficiently by clever algorithms. Where even this strategy fails, we seek for efficient heuristic algorithms.
The major results contained in this thesis are as follows:

- We provide a linear time transformation from an arbitrary permutation into its *equivalent simple permutation*, and an $O(n \log n)$ time back-transformation. Transformations like these were used in several sorting-by-reversals algorithms, and can give new insight into other genome rearrangement problems.
- We prove the NP-completeness of the transposition median problem. As a direct consequence, also the reversal and transposition median problem is NP-complete if both operations are weighted equally.
- We provide an exact branch and bound algorithm for the transposition median problem and the weighted reversal and transposition median problem which is fast enough to be used in practice. As a byproduct, we improve Christie's exact algorithm for sorting by transpositions, and extend it to sorting by weighted reversals and transpositions.
- We introduce a new pairwise distance measure which also considers operations that change the genome content, namely tandem duplications and deletions. We develop a lower bound for this distance and a greedy algorithm based on this lower bound. We provide two versions of this algorithm, one for unichromosomal circular genomes and one for multichromosomal linear genomes.

Contents

Preface

Acknowledgements

First of all I want to thank Enno Ohlebusch for supervising me. I am grateful for his helpful comments and proofreading of all my papers, but also for his motivating words during the last years.
I also would like to thank Sophia Yancopoulos and Michal Ozery-Flato. The discussions we had on the RECOMB-CG workshop in San Diego laid the foundation for my work on genome rearrangement distances with duplications.
Furthermore, I would like to thank Guillaume Bourque and Glenn Tesler for providing `MGR`, Jijun Tang for providing `GRAPPA-TP` and `DCM-GRAPPA`, and Matthias Bernt for providing `amGRP`. All of them also helped me to find the best parameters for their programs.
Finally, I would like to thank Holger Dammertz, Fabian Wagner, Thomas Schnattinger, Dominikus Krüger, and my parents for proofreading this thesis.

List of Publications

- Gog, S. and Bader, M. Fast algorithms for transforming back and forth between a signed permutation and its equivalent simple permutation. *Journal of Computational Biology* 15(8):1-13, 2008.

  The contents of this paper are presented in Chapter 2. The transformation into an equivalent simple permutation was developed together with Simon Gog, who contributed the main ideas of the algorithm.

- Bader M., Abouelhoda, M.I., and Ohlebusch, E. A fast algorithm for the multiple genome rearrangement problem with weighted reversals and transpositions. *BMC*

Bioinformatics 9:516, 2008.

The contents of this paper are presented in Chapter 4. The algorithm was developed together with Mohamed Abouelhoda and Enno Ohlebusch, where Mohamed had the initial idea of using clouds instead of a median solver.

- Bader, M. Sorting by Reversals, Block Interchanges, Tandem Duplications, and Deletions. *BMC Bioinformatics* 10(Suppl 1):S9, 2009.

  The contents of this paper are presented in Section 5.1. However, the algorithm has been adapted to circular genomes, so as to make it more consistent with the rest of this thesis.

- Bader, M. On Reversal and Transposition Medians. In *Proceedings of World Academy of Science, Engineering and Technology* 54:667-675, 2009.

  The contents of this paper are presented in Section 3.3. Our median solver has been improved since the publication of the paper. These improvements are also described in this section.

- Bader, M. Genome rearrangements with duplications. *BMC Bioinformatics* 11(Suppl 1):S27, 2010.

  The contents of this paper are presented in Section 5.2.

- Bader, M. The Transposition Median Problem is NP-complete. To be published in *Theoretical Computer Science*, doi:10.1016/j.tcs.2010.12.009.

  The contents of this paper are presented in Section 3.2.

1 Introduction

In this chapter, we cover the biological background, some fundamental definitions and concepts of genome rearrangements, as well as results from related literature.
The chapter is organized as follows. In Section 1.1, some biological background and the motivation for genome rearrangement problems is provided. In Section 1.2, some fundamental definitions and common concepts of the algorithms are shown. In Section 1.3, the most important genome rearrangement based distance measures are introduced. In Sections 1.4 to 1.7, the topics covered by this thesis and the related work from existing literature are briefly summarized.

1.1 Biological background

For understanding *genome rearrangement problems*, a certain knowledge of biology and molecular genetics is necessary. Therefore, we will give a brief introduction to molecular genetics. This introduction only covers the subjects that are relevant for understanding genome rearrangement problems, additional information can be found in any textbook about molecular biology, like e.g. [Bro02].

1.1.1 The structure of the genome

The genetic information of an organism is stored in large molecules, called *deoxyribonucleic acid* (short DNA). These molecules consist of an ordered chain of nucleotides (i.e., there is a natural direction in reading them), and each nucleotide is characterized by its base, which is one of *adenine* (A), *cytosine* (C), *guanine* (G), and *thymine* (T). Thus, on an abstract level, a DNA molecule can be seen as a string over the alphabet $\{A, C, G, T\}$. Pairs of DNA molecules are combined to the famous double-helix structure, which has been discovered by Watson and Crick in 1953 [WC53]. The two chains of a double helix (called its *strands*) have opposite orientations and are connected

by hydrogen bonds between the bases of their single nucleotides. These bonds are (almost) always between two complementary bases (where adenine and thymine as well as cytosine and guanine are complementary). Therefore, each strand is the *reverse complement* of the other strand, i.e., it can be obtained by reading the bases of the strand backwards and replacing them by their complementary base.
A single double helix is called a *chromosome*. The whole genetic information of an organism is stored in its *genome*, which can consist of one or several chromosomes. The size of a genome is measured in *million base pairs* (short Mbp). Segments of a chromosome can encode proteins. Such a segment is called a *gene*. Depending on which strand is encoding the protein, we can assign an orientation to each gene. On an even higher level of abstraction than before, a chromosome is a string of oriented genes. Note that this orientation is only relative to the other genes on the same chromosome, because the strand which encodes genes with positive orientation can be chosen arbitrarily. Thus, a chromosome is equivalent to its *reverse complement*, which can be obtained by reading the chromosome backwards and inverting the orientation of every gene.

1.1.2 Genome dynamics

The genome of an organism is stored in each of its cells. Over time, a genome can change, due to replication errors and chemical or physical influences. Some of these changes destroy the functionality of the cell and let the cell die. Others change the behavior of the cell, and lead to malfunctions of the cell. This happens for example in cancer cells. However, most of the changes have no influence at all, and some can even result in benefits for the organism. These changes are inherited by child cells, leading to organisms with a slightly modified genome.
These modifications are classified into two different types. While *mutations* affect only a short segment of the genome (often only a single base pair), *genome rearrangements* are large-scale modifications of the genome. A few examples for genome rearrangements are *deletions* (where a whole segment of the chromosome is deleted), *duplications* (where an exact copy of a segment of the chromosome is inserted), or *inversions* (where a segment of the chromosome is excised, inverted in orientation, and reinserted).
The replication mechanism of DNA is combined with a very efficient error-fixing mechanism, therefore mutations are rare events in nature. For example, the overall error rate for replication in *E.coli* is 1 in 10^{10} to 1 in 10^{11} base pairs [Bro02]. With a genome length of about 4.65 Mbp, 10000 copies of the *E.coli* genome contain about 1 point mutation. Although this seems like a very small value, these point mutations soon accumulate, and are not suitable to compare species whose last common ancestor lived millions of years ago. Genome rearrangements are observed much less frequently than point mutations, among vertebrates there are about 0.2 to 2 rearrangements per million years [BBD$^+$99]. Therefore, it is often still possible to reconstruct these evolutionary events that happened between two highly diverged species.

1.1.3 Prokaryotic and eukaryotic DNA

In biology, organisms are classified into the three domains of *bacteria*, *archaea*, and *eukaryotes*. Although archaea are more closely related to eukaryotes than to bacteria [IKH+89], their cell and genome structure shares many properties with the bacteria. Therefore, bacteria and archaea are often united to the *prokaryotes*. The main difference of cell and genome structure between prokaryotes and eukaryotes can be summarized as follows.

Prokaryotes: Prokaryotic cells contain no nucleus. Their genome mostly consists of a single circular chromosome, i.e., the double helix builds a ring. Prokaryotic DNA is relatively small (most prokaryotes have a DNA smaller than 5 Mbp), but it contains only small non-coding regions, and only a few of its genes are duplicated. Prokaryotes can have additional small DNA molecules, called *plasmides*. The genes carried by the plasmides appear to be not necessary for the function of the organism, but can encode quite useful functions, like for example antibiotic resistance.

Eukaryotes: Eukaryotic cells contain a nucleus, the DNA molecules of eukaryotes are within this nucleus. The eukaryotic genome normally consists of several linear chromosomes. The DNA molecules are large compared to prokaryotic DNA (the whole human genome has about 3000 Mbp), and contain many non-coding or duplicated regions.
Eukaryotic cells often contain subunits called *organelles*, like *chloroplasts* and *mitochondria*. Organelles are found outside the cell nucleus and contain their own small genomes. The genome of an organelle is mostly a single circular DNA, and often highly conserved (i.e., it changes very slowly during evolution), and therefore is often used to compare more distant species.

Due to these differences, it is preferable to design an algorithm either for unichromosomal circular genomes, or for multichromosomal linear genomes. Considering duplicated segments is especially of interest for multichromosomal genomes.

1.1.4 Computational challenges and genome rearrangement problems

With the rapidly increasing amount of sequenced genomes, comparisons of species based on these data become more and more interesting. However, this task is very challenging, not only because of the size of the data but also because of the structural complexity of some problems. While there are also many interesting problems considering point mutations, we will cover in this thesis only problems that ignore small mutations and only consider large-scale genome rearrangements. These so-called *genome rearrangement problems* can roughly be classified into four different categories.

1. Calculate the evolutionary distance between two genomes.
2. Reconstruct the evolutionary events that happened between two species.
3. Reconstruct the gene order of the genome of a hypothetical ancestor of two species.
4. Reconstruct a phylogenetic tree for a set of genomes.

Of course, some problems cover more than just one task. For example, the *multiple genome rearrangement problem* asks for a phylogenetic tree of a set of genomes, and the gene orders of the hypothetical ancestors at the internal nodes of the tree.

1.2 Preliminaries

We will start with some basic definitions and concepts on which all algorithms in this work rely. As most of the algorithms work on unichromosomal genomes, we give the definitions for this case. In biology, most unichromosomal genomes are circular, therefore also our model will use circular genomes. Note that in the following, we use the distance formulas for circular genomes, which often differ by 1 from the formulas for linear genomes. Most of the formulas and algorithms can easily be adjusted to linear chromosomes, which will be discussed in Section 1.2.3.
For multichromosomal genomes, some of the definitions have to be adjusted, which will be done in Chapter 5.2. Furthermore, some of the concepts, like the *breakpoint graph*, have to be extended for the different problems. Thus, we give the basic definitions in this chapter, the required extensions will be defined in the corresponding chapters.

1.2.1 Elementary definitions

A *genome* $\pi = (\pi_1 \dots \pi_m)$ is a string over the alphabet $\Sigma_n = \{1, \dots, n\}$, in which the indices are cyclic (i.e., m is followed by 1, and genomes that differ solely by a cyclic shift of the elements are considered to be equivalent) and each element x has a positive or negative orientation (indicated by $\overrightarrow{x}$ or $\overleftarrow{x}$, respectively). The number of occurrences of an element x (with arbitrary orientation) in a genome π is called its *multiplicity*, and is denoted by $mult(x, \pi)$. In classical genome rearrangement problems, it is required that all genomes contain each element of Σ_n exactly once. To emphasize this condition, we call the genome a *permutation*. The set of all genomes over Σ_n (or, depending on the context, the set of all permutations over Σ_n) is denoted by Σ_n^*. We get the *inverse* of an element π_i (indicated by $-\pi_i$) by inverting its orientation. The *reflection* of a genome $\pi = (\pi_1 \dots \pi_m)$ is the genome $-\pi = (-\pi_m \dots -\pi_1)$. It is considered to be equivalent to π. Each element x in a genome can also be represented by the ordered set of its *extremities* x_t (the *tail*) and x_h (the *head*), where the ordering of the extremities is $x_t x_h$ if x has positive orientation, and $x_h x_t$ otherwise. For example, the genome

$$rev(3,7) \cdot (\overrightarrow{1}\ \overleftarrow{4}\ \overleftarrow{6}\ \overleftarrow{11}\ \overleftarrow{10}\ \overrightarrow{3}\ \overrightarrow{7}\ \overrightarrow{5}\ \overleftarrow{8}\ \overrightarrow{2}\ \overrightarrow{9}) = (\overrightarrow{1}\ \overleftarrow{4}\ \overleftarrow{3}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{5}\ \overleftarrow{8}\ \overrightarrow{2}\ \overrightarrow{9})$$

$$tp(4,9,1) \cdot (\overrightarrow{1}\ \overleftarrow{4}\ \overleftarrow{3}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{5}\ \overleftarrow{8}\ \overrightarrow{2}\ \overrightarrow{9}) = (\overrightarrow{1}\ \overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{8}\ \overrightarrow{2}\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{5})$$

$$itp(2,5,10) \cdot (\overrightarrow{1}\ \overleftarrow{4}\ \overleftarrow{3}\ \overleftarrow{8}\ \overrightarrow{2}\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{5}) = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5})$$

$$bi(3,6,9,1) \cdot (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}) = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11})$$

(a)

$$dcj(3,9,-) \cdot (\overrightarrow{1}\ \overrightarrow{2}\ \blacktriangle\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}\ \blacktriangle\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}) = (\overrightarrow{1}\ \overrightarrow{2}\ \blacktriangle\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}) + CI(\overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}\ \blacktriangle)$$

$$dcj(1,CI4,-)(\blacktriangle\ \overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}) + CI(\overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11}\ \blacktriangle\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}) = (\blacktriangle\ \overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \blacktriangle\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8}\ \overrightarrow{9}\ \overrightarrow{10}\ \overrightarrow{11})$$

(b)

Figure 1.1: (a) Examples for the operations. Note that indices are cyclic, thus the last parameter of the transposition and the block interchange corresponds to a cut after the last element. (b) The block interchange can be simulated by two DCJs. The first DCJ cuts out a circular intermediate (marked with CI), the second rejoins the genomes.

$\pi = (\overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{1}\ \overleftarrow{4})$ can also be written as $\pi = (2_t\ 2_h\ 3_t\ 3_h\ 1_t\ 1_h\ 4_h\ 4_t)$. This notation is called the *extremities notation*. Two extremities belonging to the same element are also called *co-elements*. The genome $id = (\overrightarrow{1} \dots \overrightarrow{n})$ is called the *identity genome*. An unordered pair of extremities $\{x_{t/h}, y_{t/h}\}$ is called an *adjacency* of two genomes π and ρ if $x_{t/h}$ and $y_{t/h}$ are adjacent in both π and ρ. If $x_{t/h}$ and $y_{t/h}$ are adjacent in π but not in ρ, it is called a *breakpoint* of π w.r.t. ρ. In the example above, $\{2_h, 3_t\}$ is an adjacency of π and id, while $\{3_h, 1_t\}$, $\{1_h, 4_h\}$, and $\{4_t, 2_t\}$ are breakpoints of π w.r.t. id. A *segment* $\pi_i \dots \pi_j$ is a consecutive sequence of elements of a genome. An *operation* op transforms a genome π into the genome $op \cdot \pi$. Note that we prefer this notation over the standard notation $op(\pi)$, because it better distinguishes between parameters of the operation and the genome where it is applied to, and because of its better readability when concatenating operations.
In this work, we will mainly consider the following operations (additional operations will be described in Chapter 5). Examples for these operations can be seen in Fig. 1.1.

- A *reversal* $rev(i,j)$ inverts the order of the elements of the segment $\pi_i \dots \pi_{j-1}$. Additionally, the orientation of each element is flipped, i.e., applying the reversal $rev(i,j)$ on π yields the genome $rev(i,j) \cdot \pi = (\pi_1 \dots \pi_{i-1}\ -\pi_{j-1} \dots -\pi_i\ \pi_j \dots \pi_m)$.

- A *transposition* $tp(i, j, k)$ cuts the segment $\pi_i \dots \pi_{j-1}$ out of π, and reinserts it before the element π_k, i.e., applying the transposition $tp(i, j, k)$ on π yields the genome $tp(i, j, k) \cdot \pi = (\pi_1 \dots \pi_{i-1}\ \pi_j \dots \pi_{k-1}\ \pi_i \dots \pi_{j-1}\ \pi_k \dots \pi_m)$.

- An *inverted transposition* $itp(i, j, k)$ has the same effect as a transposition, but additionally inverts the segment $\pi_i \dots \pi_{j-1}$ before reinserting, i.e., $itp(i, j, k) \cdot \pi = tp(i, j, k) \cdot rev(i, j) \cdot \pi$.

- A *block interchange* $bi(i, j, k, l)$ cuts the segments $\pi_i \dots \pi_{j-1}$ and $\pi_k \dots \pi_{l-1}$ out of a genome, and swaps their positions. That is, applying the block interchange $bi(i, j, k, l)$ on π yields the genome $bi(i, j, k, l) \cdot \pi = (\pi_1 \dots \pi_{i-1}\pi_k \dots \pi_{l-1}$ $\pi_j \dots \pi_{k-1}\pi_i \dots \pi_{j-1}\pi_l \dots \pi_m)$. Note that a transposition is a special case of a block interchange, where the segments are consecutive.

- A *double cut and join operation* $dcj(i, j, x)$ (with $i, j \in [1, m]$ and $x \in \{+, -\}$) cuts the genome π before the elements π_i and π_j, and rejoins the cut ends in two new pairs. If $x = +$, we rejoin such that the elements π_{i-1} and π_{j-1} as well as the elements π_i and π_j become adjacent. This is equivalent to the reversal of the segment $\pi_i \dots \pi_{j-1}$, i.e., $dcj(i, j, +) \cdot \pi = rev(i, j) \cdot \pi$. If $x = -$, we rejoin such that the elements π_{i-1} and π_j as well as the elements π_i and π_{j-1} become adjacent. This cuts π into the genomes $(\pi_1 \dots \pi_{i-1}\pi_j \dots \pi_m)$ and $(\pi_i \dots \pi_{j-1})$. The latter genome is called a *circular intermediate*, and can be absorbed by another double cut and join operation with one cutting point in each of the genomes. Depending on how we rejoin, those two operations are equivalent to either two consecutive reversals or to one block interchange.

A sequence of operations $op_1 \dots op_k$ applied on π yields the genome $op_k \cdot \dots \cdot op_1 \cdot \pi$. If a sequence of operations transforms a genome π into a genome ρ, we say it is a *sorting sequence* of π w.r.t. ρ. A genome π' with $\pi' = op_j \cdot \dots \cdot op_1 \cdot \pi$ for a $j \in [1, k]$ *lies* on the sorting sequence. A *weighting function* $w : Operations \to \mathbb{R}_0^+$ assigns each operation op a non-negative *weight* $w(op)$. The weight of a sequence of operations is the sum of the weights of its operations. A *genome rearrangement problem* is defined as follows. Given two genomes π and ρ, a set of operations, and a weighting function, find a sorting sequence of π w.r.t. ρ of minimal weight. The weight of this sequence is called the *distance* $d(\pi, \rho)$ between the genomes. For example, in the problem *sorting by reversals*, the set of operations consists solely of reversals, and the weighting function weights each reversal with 1, i.e., one searches for a minimal sequence of reversals that transforms π into ρ. The *reversal distance* $d_{rev}(\pi, \rho)$ is the minimum number of reversals required to transform π into ρ. For more rearrangement problems and distance measures, see Section 1.3.

1.2.2 Relevance of the operations

The first evidence of a reversal has already been found by Sturtevant in 1926 with the help of light microscopes [Stu26]. Shortly after, Dobzhansky and Sturtevant provided evidence of several reversals in giant chromosomes in strains of *Drosophila pseudoobscura* coming from different geographical regions [DS38]. Nowadays, reversals are considered to be the most frequent rearrangements in unichromosomal genomes, especially in plant mitochondrial DNA [PH88] and bacteria [Hug00]. But also transpositions and inverted transpositions play an important role in evolution, as it has been shown by several authors (see e.g. [DEEA02, KBH$^+$03]).

The relevance of block interchanges and the double cut and join operation (short DCJ) is still unclear. Although the creation of circular intermediates and their absorption by another DCJ is a well-known process in the immune response of higher animals [KM07] and has also been observed in the genome of the bacterium *Borrelia burgdorferi* [CPvV$^+$00], there is no evidence that these operations are relevant in evolution.

Despite of this question, the possibility of unifying several different operations by using DCJs is advantageous enough to study this operation, especially as DCJs can also simulate *translocations*, *fusions*, and *fissions* in multichromosomal genomes. The original algorithm for *sorting by double cut and join* by Yancopoulos et al. [YAF05] has been extended by Bergeron et al. to a universal framework where linear and circular chromosomes can co-exist [BMS06]. By restricting the set of allowed DCJs in this framework, one can adjust the set of simulated biological operations [LHWC06, BMS09]. This leads to simplified versions of existing algorithms [BMS09] or efficient algorithms for new models [LHWC06].

1.2.3 Circular versus linear genomes

Although most unichromosomal genomes are circular in reality, it is sometimes desirable to also have an algorithm for linear unichromosomal genomes. Fortunately, most of the problems for linear genomes are linearly equivalent to the corresponding problems for circular genomes, i.e., the algorithms for circular genomes can easily be modified such that they solve the problem for linear genomes without changing their time or space complexity.

The idea of this modification has been presented by Meidanis et al. in [MWD00]. They add a *boundary element* $\overrightarrow{0}$ at the end of the linear genomes, and assume that the genomes are now circular. As long as an operation does not affect a segment containing the boundary element, there is a corresponding operation for the linear problem. If an operation affects the boundary element, it has to be replaced by an equivalent operation that does not affect it, where two operations op and op' are considered to be equivalent if $op \cdot \pi$ and $op' \cdot \pi$ are considered to be equivalent (remember that two

circular genomes are equivalent if one is the cyclic shifted version or the reflection of the other). It has been proven in [MWD00] that for each reversal that affects the boundary element, there exists an equivalent reversal that does not affect it. Hartman and Sharan have shown that this also holds for transpositions [HS06], and the proof can easily be adjusted to block interchanges. For inverted transpositions, this only holds if the inverted segment does not contain the boundary element. Otherwise, the operation is equivalent to an operation called *revrev*, which consists of two reversals of consecutive segments [HS05]. In other words, the transformation works as long as we only use reversals, transpositions, and block interchanges. If also inverted transpositions are allowed, we also have to allow revrevs in the linear problem.
Note that, due to the use of circular genomes, the distance formulas in the following chapters may vary by 1 from the distance formulas from the original papers, i.e., we always provide the distance formulas for circular genomes.

1.2.4 The breakpoint graph

Most genome rearrangement algorithms rely on the *breakpoint graph*, which has first been introduced by Bafna and Pevzner [BP93][1] for genome rearrangement problems between two permutations π and ρ (i.e., both genomes contain each element in $\Sigma_n = \{1, \ldots, n\}$ exactly once). We will use a presentation similar to the one proposed in [SM97], which is more convenient for circular genomes. The breakpoint graph $BG(\pi, \rho) = (V, E_\pi \cup E_\rho)$ is an edge-colored multigraph (i.e., it may contain parallel edges) which visualizes the neighborhood relations of the elements in both π and ρ, and is defined as follows.

1. The set of nodes consists of all extremities of the genomes, i.e., $V = \{1_t, 1_h, \ldots, n_t, n_h\}$. The ordering of the elements in π defines an ordering of the nodes, i.e., we can compare two nodes by the $<$ operator. If we say that a node u is left (or right) of another node v, we mean that $u < v$ (or $v < u$). The *position* of a node v (denoted by $pos(v)$) is its position in this ordering, beginning with 0. When visualizing the breakpoint graph, its nodes are normally written counterclockwise ordered on a circle.

2. The set of *black edges* E_π consists of the neighborhood relation in π, i.e.,

$$E_\pi = \{(u, v) \mid u, v \text{ are adjacent in } \pi \text{ and are not co-elements}\}.$$

3. Analogously, the set of *gray edges* E_ρ consists of the neighborhood relation in ρ, i.e.,

$$E_\rho = \{(u, v) \mid u, v \text{ are adjacent in } \rho \text{ and are not co-elements}\}.$$

An example of a breakpoint graph can be seen in Fig. 1.2. Two edges (u, v) and (x, y)

[1] A more comprehensive version of this paper appeared as [BP96].

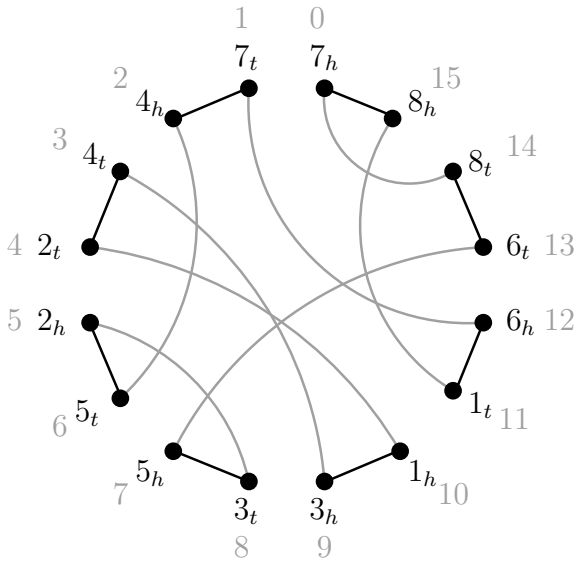


Figure 1.2: The breakpoint graph of $\pi = (\overleftarrow{7}\ \overleftarrow{4}\ \overrightarrow{2}\ \overrightarrow{5}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{6}\ \overrightarrow{8})$ and $\rho = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6}\ \overrightarrow{7}\ \overrightarrow{8})$. The black labels are the node labels, the gray numbers are the positions of the nodes.

(with $u < v$ and $x < y$) are *intersecting* if $u < x < v < y$ or $x < u < y < v$. Note that, due to the ordering of the nodes, only gray edges can intersect. As each node is incident to exactly one black and one gray edge, the breakpoint graph decomposes into cycles. Two cycles c_1 and c_2 *intersect* if there is a gray edge in c_1 which intersects with a gray edge in c_2. The *length of a cycle* $\ell(c)$ is the number of its black edges (or equivalently, gray edges). A cycle of length k is called a *k-cycle*. If $k \leq 2$, the cycle is a *short cycle*, otherwise it is a *long cycle*. If k is odd, the cycle is an *odd cycle*, otherwise it is an *even cycle*. The number of odd cycles in $BG(\pi, \rho)$ is denoted by $c_{odd}(\pi, \rho)$, the number of even cycles is denoted by $c_{even}(\pi, \rho)$. The overall number of cycles in $BG(\pi, \rho)$ is denoted by $c(\pi, \rho)$. It is often convenient to describe an operation by its effects on the breakpoint graph. As operations change the neighborhood relations in π, its main effect on the breakpoint graph is that some black edges are cut, and some new black edges may be inserted. We say that the operation *acts* on these edges.
Sometimes, it is advantageous to just consider a subgraph of the breakpoint graph (e.g., a single cycle). Such a subgraph is called a *configuration* of the breakpoint graph. To distinguish a configuration from the whole breakpoint graph, we draw its nodes on a straight line instead of on a circle.

1.3 Genome rearrangement problems and distance measures

The following distance measures and genome rearrangement problems have been extensively studied during the last decades. Note that all these problems were originally stated for genomes without duplicates (i.e., permutations), and cannot easily be extended to genomes with duplicates (except for the breakpoint distance).

- The *breakpoint distance* $d_{bp}(\pi, \rho)$ is the number of breakpoints of π w.r.t. ρ. It is trivial to compute, and therefore has been proposed by Sankoff and Blanchette to be used in the multiple genome rearrangement problem [SB98].

- The problem *sorting by reversals* (short SBR) is a genome rearrangement problem where the only allowed operations are reversals, all weighted equally. That is, one searches for a sorting sequence of a permutation π w.r.t. another permutation ρ, such that the sequence consists solely of reversals and has minimal length. The length of such a sequence is called the *reversal distance* $d_{rev}(\pi, \rho)$. In other words, $d_{rev}(\pi, \rho)$ is the minimum number of reversals that is required to transform π into ρ. The currently best algorithm for sorting by reversals has a running time of $O(n^{1.5})$ [TBS07, Han06], and the reversal distance can be computed in linear time [BMY01, BMS04].

- The problem *sorting by transpositions* (short SBT) is a genome rearrangement problem where the only allowed operations are transpositions, all weighted equally. The corresponding distance measure is called the *transposition distance* $d_{tp}(\pi, \rho)$. It is still unknown whether these problems can be solved in polynomial time, and the currently best algorithm provides an approximation guarantee of 1.375 [EH06].

- In the problem *sorting by weighted reversals and transpositions* (short SBwRT), the set of operations consists of reversals, transpositions, and inverted transpositions. Reversals are weighted by w_r, transpositions and inverted transpositions are weighted by w_t, where both values are fixed user-defined parameters. The corresponding distance measure is called the *weighted reversal and transposition distance* $d_{wrt}(\pi, \rho)$. SBwRT has been examined for different weight ratios $w_r : w_t$. As each transposition can be replaced by 3 reversals, and each inverted transposition can be replaced by 2 reversals, SBwRT is equivalent to SBR if $3w_r \leq w_t$. For other weight ratios, the complexity of the problem is still open, and only approximation algorithms exist. For the weight ratio $1 : 2$, Eriksen provided a $(1 + \varepsilon)$ approximation algorithm [Eri02]. Hartman and Sharan provided a 1.5-approximation algorithm for the weight ratio $1 : 1$ [HS05]. Bader and Ohlebusch devised an algorithm that can guarantee an approximation ratio of 1.5 for any weight ratio with $w_r \leq w_t \leq 2w_r$ [BO07]. For a more restricted version of the problem, where inverted transpositions are excluded, Walter et al. [WDM98] as

well as Lin and Xue [LX01] provided a 2-approximation algorithm for the weight ratio 1 : 1.
The question of which weight ratio results in the biologically most realistic results has been discussed in several works (see e.g. [BKS96, Eri03, Ber10, JA11]). However, the results also depend on the chosen algorithm (in [BKS96] and [Eri03], the parameters are optimized for the software tool `DERANGE II`), and all works consider only pairwise distances and not the effects on the multiple genome rearrangement problem.
Because transpositions and inverted transpositions can remove more breakpoints and create more cycles than reversals, a single transposition or inverted transposition can contribute more to the sorting scenario than a single reversal, therefore these operations are generally favored if all operations are weighted equally. On the other hand, if the weight ratio is $1 : 2$ ($w_r : w_t$) and one uses Eriksen's $(1+\varepsilon)$-approximation algorithm [Eri02], then the resulting sorting sequence consists solely of reversals in most cases (note that transpositions are only required if the breakpoint graph contains a hurdle, which is a rare event [Cap99]). In our opinion, as reversals are observed more frequently in biology than transpositions, a weight ratio that results in biologically meaningful sorting scenarios must therefore be somewhere between $1 : 1$ and $1 : 2$ ($w_r : w_t$).

- The problem *sorting by double cut and join* (short SBDCJ) is a genome rearrangement problem where the only allowed operations are DCJs, all weighted equally. The corresponding distance measure is called the *double cut and join distance* $d_{dcj}(\pi, \rho)$. Both problems can be solved in linear time [YAF05, BMS06].

1.4 Simple permutations

One of the most important and best studied problems in comparative genomics is SBR, where one searches for a minimum sequence of reversals that transforms a permutation π into another permutation ρ. The first polynomial time algorithm for this problem was presented by Hannenhalli and Pevzner in 1995 [HP95][2]. The algorithm was simplified several times [BH96, KST99], and the *reversal distance problem* (in which one is only interested in the number of required reversals) can be solved in linear time [BMY01, BMS04]. In 2004, Tannier and Sagot presented an algorithm for SBR that has subquadratic time complexity [TS04] (the algorithm was later improved by Tannier et al. [TBS07] and Han [Han06]). The algorithm of Hannenhalli and Pevzner [HP99] as well as the original algorithm of Tannier and Sagot [TS04] first transform π and ρ into *equivalent simple permutations* π_{simple} and ρ_{simple}, such that $d_{rev}(\pi_{simple}, \rho_{simple}) = d_{rev}(\pi, \rho)$ and $BG(\pi_{simple}, \rho_{simple})$ only contains short cycles.

[2] A more comprehensive version of this paper appeared as [HP99].

Then, they compute a sorting sequence of π_{simple} w.r.t. ρ_{simple}. As final step, this sorting sequence is transformed back into a sorting sequence of π w.r.t. ρ. In literature, there are several algorithms for the transformation into simple permutations [HP99, BH96], but all of them have at least quadratic time complexity (there is an unpublished linear time algorithm by Tannier and Sagot which uses another technique than our algorithm [TS07]). For the back transformation to get the sorting sequence of π w.r.t. ρ, there is no algorithm that performs better than the naive approach, which has a quadratic running time. Although Tannier et al. improved their algorithm such that it does no longer require simple permutations [TBS07], a fast algorithm for the transformation could be crucial for further investigations on genome rearrangements. In Chapter 2, we show how two permutations can be transformed into their equivalent simple permutations in linear time. Furthermore, we provide an $O(n \log n)$ time algorithm to transform a sorting sequence on the simple permutations into a sorting sequence on the original permutations. While the first algorithm is specific for SBR, the second can be easily adjusted to any genome rearrangement algorithm that works on simple permutations with padded elements, like the current state-of-the-art algorithms for SBT [HS06, EH06] and SBwRT [HS05, BO07].

1.5 On median problems

Due to the increasing amount of sequenced genomes, the problem of reconstructing phylogenetic trees based on these data is of great interest in computational biology. In the so-called *multiple genome rearrangement problem*, one searches for a phylogenetic tree describing the most "plausible" rearrangement scenario for multiple genomes. Formally, given k genomes (the *input genomes*) and a distance measure d, find a tree T with the k genomes as leaf nodes and assign ancestral genomes to internal nodes of T such that the tree is optimal w.r.t. d, i.e., the sum of rearrangement distances over all edges of the tree is minimal. If $k = 3$, i.e., one searches for a genome σ such that the sum of the distances from σ to three given genomes is minimized, we speak of the *median problem*, and an algorithm that tackles this problem (either exactly or approximatively) is called a *median solver*. The median problem is the simplest form of the multiple genome rearrangement problem, and median solvers are used in all current state-of-the-art algorithms for the multiple genome rearrangement problem as a subroutine. However, even this problem has been proven to be NP-complete for most distance measures. In the context of comparative genomics, the median problem has been intensively studied during the last decades for the following distance measures.

Breakpoint distance: The use of the breakpoint distance in the multiple genome rearrangement problem was first proposed by Sankoff and Blanchette [SB98]. They showed that the *breakpoint median problem* (short BMP) is a special case of the well-known *traveling salesman problem* (short TSP), and solved instances of

the BMP by using an algorithm for the TSP. Because the breakpoint distance is much easier to compute than the reversal distance, their software tool `BPAnalysis` could solve much larger instances than other approaches which tried to solve the RMP at that time. However, despite of the fact that most instances of the BMP can be solved very fast in practice, it was shown by Pe'er and Shamir that the BMP is NP-complete [PS98].

Reversal distance: Although the reversal distance can be computed in linear time [BMY01], its structure is more complex than the one of the breakpoint distance, making it much harder to find convenient algorithms for the *reversal median problem* (short RMP). The first attempt to solve the RMP was done by Hannenhalli et al. [HCKP95]. Their approach used an exhaustive search over a bounded search space, and therefore was limited to relatively close related genomes. Sankoff et al. [SSK96] used a heuristic algorithm, which allowed them to compute the median of more distant genomes. Still, this approach was very limited, and both approaches mentioned above "have been little used because of the computational difficulty in generalizing measures of genomic distance to more than two genomes" [SB98]. The breakthrough of the reversal median versus the breakpoint median was due to the exact algorithms by Siepel and Moret [SM01] and Caprara [Cap03], although it could be proven that the RMP is NP-complete [Cap03]. Today, almost all state-of-the-art algorithms for phylogenetic reconstruction based on the reversal distance use either Siepel's algorithm [MSTL02] or Caprara's algorithm [MSTL02, BAO08, BMM07] as a subroutine, where `GRAPPA` (which provides both algorithms) recommends the use of Caprara's algorithm [MT04]. An exception is `MGR` [BP02], which uses its own heuristic reversal median solver. There exist several speed improvements for this algorithm [AT07, STTM09], however all at the cost of accuracy.

Transposition distance: As the complexity of the transposition distance is still open, only few attempts to solve the *transposition median problem* (short TMP) can be found in literature. Sankoff et al. [SSK96] used a heuristic algorithm that is based on pairwise sorting scenarios, which were created by a greedy heuristic. A more recent approach is implemented in `GRAPPA-TP` [YZT08], which uses an extension of Siepel's median solver [SM01] and solves pairwise distances by a fast heuristic. Although it suggests itself from the results about the RMP that also the TMP is NP-complete, no attempts to prove this conjecture can be found in literature.

Weighted reversal and transposition distance: To the best of our knowledge, the only attempt to solve the *weighted reversal and transposition median problem* (short wRTMP) was due to Hannenhalli et al. [HCKP95], who weighted both operations equally. They used the same approach as for the RMP, but, due to the lack of an exact algorithm for the pairwise distance, they had to use an exhaustive search

to determine the bounds. Naturally, also this attempt was limited to very closely related genomes. From a theoretical view, the NP-completeness of the problem follows immediately from [Cap03] for the weight ratio $w_r : w_t = 1 : 2$. For other weight ratios, the problem has not been examined so far.

DCJ distance: Due to the similar properties of the reversal distance and the DCJ distance, Caprara's proof of the NP-completeness of the RMP [Cap03] is still valid for the *DCJ median problem* (this holds even for multichromosomal genomes, see [TZS08]), and his reversal median solver can easily be transformed into a DCJ median solver. Nowadays, almost all DCJ median solvers are based on Caprara's reversal median solver (an exception is [AS08]), and recent research has brought up speed-ups for unichromosomal genomes [XS08, Xu09b] as well as extensions to multichromosomal genomes [Xu09a, ZAT09].

In Chapter 3, we focus on median problems. The main results are a proof of the NP-completeness of the TMP and an exact branch-and-bound algorithm for the wRTMP and the TMP. Both results are an extension of Caprara's work on the RMP in [Cap03]. As a byproduct, we also improve Christie's exact algorithm for SBT [Chr98].

1.6 Phylogenetic reconstruction

The reconstruction of phylogenetic trees based on genome rearrangement motivated distance measures is an important field of research in comparative genomics. The algorithmic challenge is to find fast and accurate heuristics for the multiple genome rearrangement problem, as stated in the last section, for an arbitrary number of input genomes. During the last two decades, several approaches have been proposed, which can be classified into two main strategies.

Strategy 1: First create a tree topology, then assign ancestral genomes to internal nodes.

The first algorithm following this strategy was `BPAnalysis` [SB98], which iterated over all possible tree topologies, used a heuristic for initially assigning ancestral genomes to internal nodes, and then iteratively improved these genomes by using a median algorithm for the breakpoint distance. Mainly due to the iteration over all tree topologies, the algorithm was rather slow, therefore Cosner et al. replaced this iteration by a heuristic called `MPBE` [CJM+00a, CJM+00b]. Moret et al. provided a reimplementation of the original `BPAnalysis` algorithm called `GRAPPA`, which, combined with a bounding strategy for selecting the tree topology, resulted in a speed improvement of several orders of magnitude [MWB+01]. The algorithm has been further improved [MTWW02, MSTL02, LTM05], the current version `GRAPPA 2.0` [MT] is using a reversal median solver instead of a breakpoint median solver, and can easily be adapted to other distance measures [TMCd04, YZT07].

However, the heuristic to create the tree topology is still based on the breakpoint distance, and only a very recent paper provides a scoring method for tree topologies which is based on the DCJ distance [XM10].

Strategy 2: Start with an empty tree, iteratively add the input genomes. In each iteration step, immediately label any new internal node.
This strategy was first implemented in a software tool called `MGR` [BP02]. The heuristic for adding a genome to the tree is based on a solver for the RMP, but using other distances, like the reversal and translocation distance [BP02] or the DCJ distance [AS08], is also possible. Bernt et al. improved `MGR` by making use of the fact that the median of three genomes is not unique in most cases. By using a heuristic for selecting one of the medians, their software tool `amGRP` outperforms `MGR` in both accuracy and running time [BMM07].

In Chapter 4, we present a new algorithm for the multiple genome rearrangement problem that tries to construct an optimal phylogenetic tree under the weighted reversal and transposition distance. It consists of two phases, the construction and the improvement phase. The construction phase follows Strategy 2. In contrast to previous algorithms, no median solver is used in this phase, as solving instances of the wRTMP takes much longer than solving instances of the RMP. In the improvement phase, we use a median solver to improve the labelling of the internal nodes, as well as a new algorithm that improves the tree topology.

1.7 Genome rearrangements with duplications

In the classical approach to genome rearrangement problems, each gene appears exactly once in each genome, and operations cannot change the content of a genome. We call these operations (like reversals, transpositions, or DCJs) *classical operations*. While this approach leads to efficient algorithms, restricting the genes to be unique in each genome does not reflect the biological reality very well. In most genomes that have been studied, there are some genes that are present in two or more copies. This holds especially for the genomes of plants, and one of the most prominent genomes is the one of the flowering plant *Arabidopsis thaliana*, where large segments of the genome have been duplicated (see e.g. [BBG$^+$00]). There are various evolutionary events that can change the content of the genome, like duplications of single genes, horizontal gene transfer, or tandem duplications. For a nice overview in the context of comparative genomics, see [San01].
In a pioneering work, Sankoff tackled the challenge of genomes with duplicated genes with his "exemplar model" [San99], where the following problem was examined. Given two genomes with duplicated genes, identify in both genomes the "true exemplars" of each gene and remove all other genes, such that the rearrangement distance be-

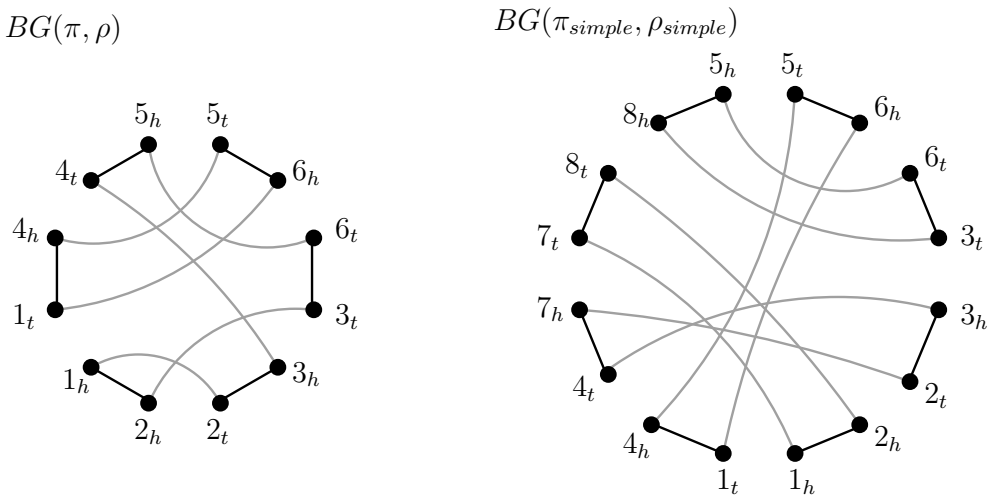
tween these modified genomes is minimized. This approach was later extended to the "matching model", where one searches for a maximum matching between the copies of each gene such that the genome rearrangement distance according to this matching is minimized [BCF04]. However, both approaches have been proven to be NP-hard for the breakpoint distance and the reversal distance [Bry00, BCF04, CZF+05]. Note that these approaches do not construct the evolutionary events that changed the genome contents, i.e., the set of operations is still restricted to the set of classical operations. The first approach that explicitly constructed duplication events was done by El-Mabrouk [EM02], where one searches for a hypothetical ancestor with unique gene content, such that the reversal and duplication distance from this ancestor to a given descendant (with duplicated genes) is minimized. This work has been further extended during the last years (see e.g. [BLEMG06, CZF+05, FCV+06]). Still, the duplications were technically limited to have the length of one element, and therefore the algorithms can only be applied if no large segmental duplication happened during evolution. One idea to overcome this problem was to simulate duplications by insertions, as it has been done in [EM01, MSM04, SMEDM08]. Recently, two new mathematical models for a genome rearrangement based distance measure were proposed [YF08, LRSM10]. Both take classical operations as well as duplications of arbitrary length into account ([YF08] weights duplications by length, while [LRSM10] uses a threshold for the length of a duplication). However, both models only provide a distance measure, and it is not possible to efficiently calculate a corresponding sorting scenario. To the best of our knowledge, the first work that allowed duplications of arbitrary segments was done by Ozery-Flato and Shamir [OFS07], who demonstrated that a simple greedy algorithm can find biologically realistic sorting scenarios for most karyotypes in the *Mitelman Database of Chromosome Abberations and Gene Fusions in Cancer* [MJM10]. Further simplifications of the model led to an algorithm with provable approximation ratio of 3 [OFS08] (note that the algorithm performs much better in practice).
In Chapter 5, we provide a heuristic algorithm for the following problem. Given an ancestral genome ρ with unique gene content and the genome of a descendant π with arbitrary gene content, find a shortest sorting sequence of ρ w.r.t. π. We first focus on the unichromosomal case, where the set of operations consists of reversals, block interchanges, tandem duplications, and deletions. Then, the approach is extended to multichromosomal genomes. This extends the set of operations by translocations, fusions, fissions, chromosome duplications, and chromosome deletions.

2 Simple Permutations

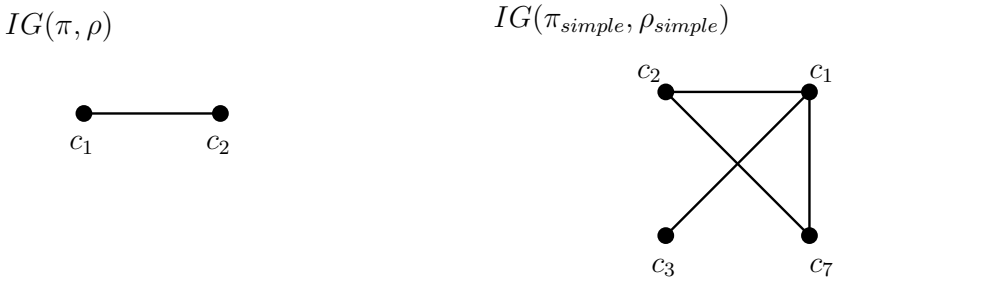
In this Chapter, we examine how we can transform forth and back between two permutations and their *equivalent simple permutations*, which are important steps in some algorithms from the literature (e.g. [HP99, TS04]).
The chapter is organized as follows. In Section 2.1, some fundamental definitions and results from the literature are given. In Section 2.2, we provide a linear time algorithm to transform two permutations π and ρ into their *equivalent simple permutations* π_{simple} and ρ_{simple} by padding elements to them. In Section 2.3, we show how a sorting sequence of π_{simple} w.r.t. ρ_{simple} can be transformed back into a sorting sequence of π w.r.t. ρ, the running time of our algorithm is $O(n \log n)$. While the former algorithm is specific for sorting by reversals, the latter can be adjusted to any genome rearrangement algorithm that works with padded elements, like e.g. [HS06, EH06, HS05, BO07].

2.1 Fundamental definitions and results

Let (π, ρ) be an instance of a *sorting by reversals* problem, i.e., both π and ρ are permutations of size n. The permutations π' and ρ' are called *extended permutations* of π and ρ if they were obtained by padding k elements $n+1, \ldots, n+k$ at arbitrary positions in both π and ρ. In the extended permutations, the elements $1, \ldots, n$ are called *original elements*, whereas the elements $n+1, \ldots, n+k$ are called *padded elements*. If the breakpoint graph $BG(\pi', \rho')$ contains only short cycles, π' and ρ' are called *simple permutations*. If it further holds that $d_{rev}(\pi', \rho') = d_{rev}(\pi, \rho)$, we say that π' and ρ' are *equivalent simple permutations* of π and ρ. We also denote equivalent simple permutations of π and ρ by π_{simple} and ρ_{simple}. For an example, see Fig. 2.1(a).
In order to understand equivalent simple permutations, we need to know some details about the reversal distance. A gray edge (u, v) in a breakpoint graph is *oriented* if $pos(v) - pos(u)$ is an even number, otherwise the edge is *unoriented*. A cycle is *oriented*



(a)



(b)

Figure 2.1: (a) The breakpoint graph of $\pi = (\overrightarrow{5}\ \overrightarrow{4}\ \overrightarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overrightarrow{6})$ and $\rho = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6})$, and the one of their equivalent simple permutations $\pi_{simple} = (\overrightarrow{5}\ \overleftarrow{8}\ \overrightarrow{7}\ \overrightarrow{4}\ \overrightarrow{1}\ \overleftarrow{2}\ \overleftarrow{3}\ \overrightarrow{6})$ and $\rho_{simple} = (\overrightarrow{1}\ \overrightarrow{7}\ \overrightarrow{2}\ \overrightarrow{8}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6})$. (b) The corresponding interleaving graphs. In the interleaving graphs, c_x denotes the cycle containing the node x_t. According to Theorem 2.1, $d_{rev}(\pi, \rho) = d_{rev}(\pi_{simple}, \rho_{simple}) = 4$.

if it contains at least one oriented gray edge, or if its length is 1. Otherwise, the cycle is *unoriented*. The *interleaving graph* $IG(\pi, \rho)$ is a graph (V, E) with set of nodes

$$V = \{c \mid c \text{ is a cycle in } BG(\pi, \rho)\}$$

and set of edges

$$E = \{(c_i, c_j) \mid c_i \text{ and } c_j \text{ are intersecting in } BG(\pi, \rho)\}.$$

Examples of interleaving graphs are depicted in Fig 2.1 (b).
In the context of sorting by reversals, a *component* is a connected component in $IG(\pi, \rho)$ in the graph theoretical manner, i.e., it is a set of nodes which are pairwise connected by paths in $IG(\pi, \rho)$. A component is *oriented* if it contains at least one oriented cycle, otherwise it is *unoriented*. Hannenhalli and Pevzner defined some special structures that depend on unoriented components, called *hurdles* and *fortress* (for details, see [HP99]). With these structures, they were able to prove the following theorem.

Theorem 2.1. *[HP99] The distance formula for the reversal distance is*

$$d_{rev}(\pi, \rho) = n - c(\pi, \rho) + h(\pi, \rho) + f(\pi, \rho)$$

where $h(\pi, \rho)$ is the number of hurdles in $IG(\pi, \rho)$, and $f(\pi, \rho)$ is the fortress indicator.

2.2 Transforming a permutation into its equivalent simple permutation

Before we describe our algorithm, we have to introduce a canonical labeling of the nodes and edges of the breakpoint graph, similar to the one in [Chr98]. Furthermore, we review the definition and results about (b, g)-splits from [HP99]. Then, we are ready to describe the data structures used for our algorithm, and the algorithm itself.

2.2.1 The canonical labeling of cycles

The algorithm performs several scanlines over the breakpoint graph which must start at the left node of a black edge. As the original ordering of the nodes starts with the right node of a black edge, we change the position of this node from 0 to $2n$. That is, this node is now the rightmost node and the node with position 1 is now the leftmost node of the breakpoint graph. Note that this can be done without creating any conflicts, because the "starting point" in a circular ordering can be chosen arbitrarily.
The canonical labeling of the nodes and edges of a cycle in the breakpoint graph is now defined as follows. The leftmost node of a cycle (i.e., the one with the least position) is labelled with v_1. Then, we follow the black edge to node v_2, then the gray edge to node v_3, and so on. The black edge from node v_{2i-1} to v_{2i} is labelled by b_i, the gray edge from node v_{2i} to node v_{2i+1} is labelled by g_i. An example of such a labeling is depicted in Fig. 2.2.

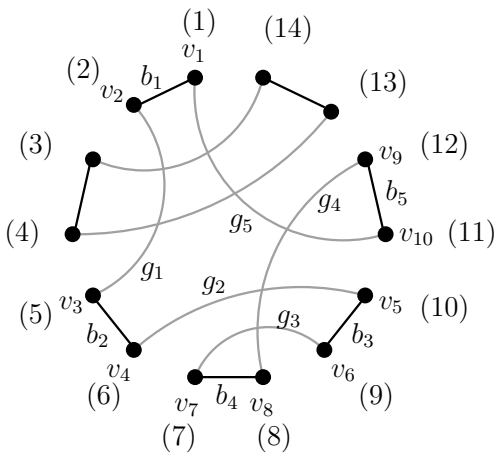


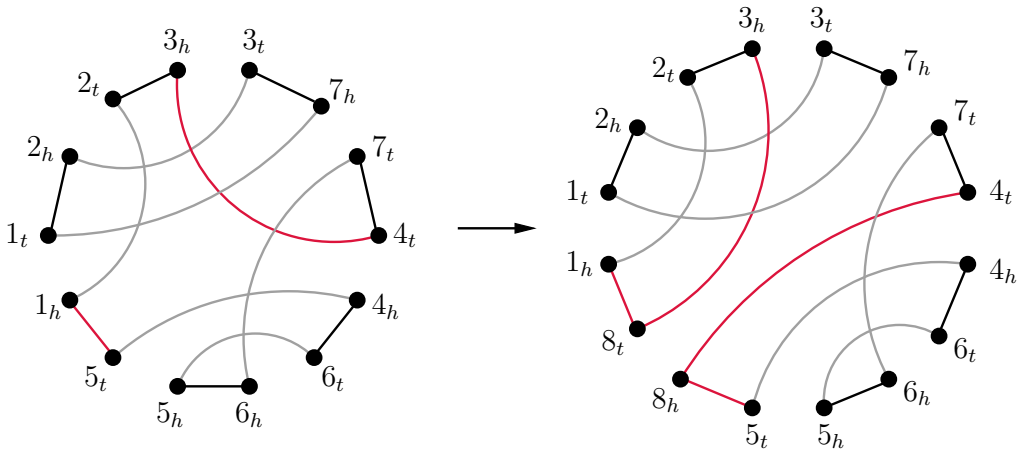
Figure 2.2: The canonical labeling of a cycle. The numbers in brackets are the node positions.

2.2.2 (b, g)-splits

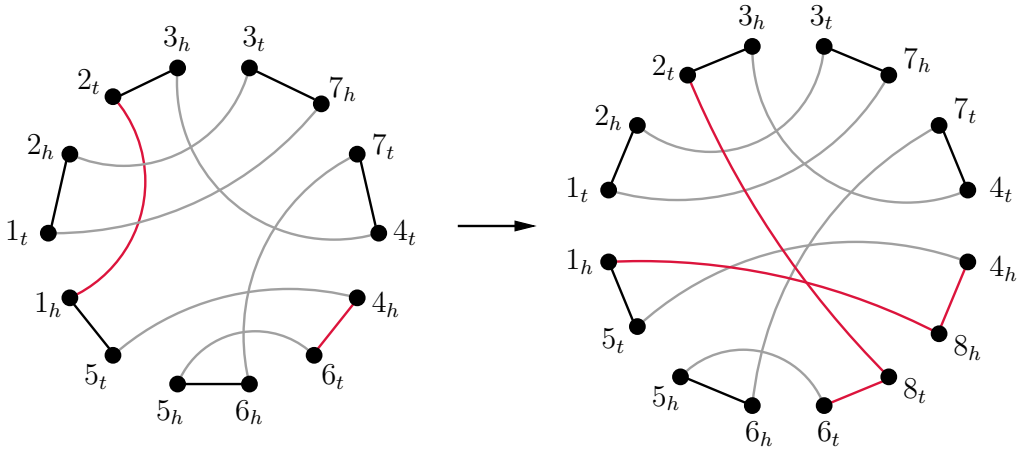
We first focus on the creation of simple permutations before we discuss the creation of equivalent simple permutations. If $BG^0 := BG(\pi, \rho)$ contains a long cycle, it is transformed into a new breakpoint graph BG^1 by "breaking" this cycle into two smaller ones. This step is repeated until the resulting breakpoint graph contains only short cycles, i.e., we get a breakpoint graph $BG^k = BG(\pi^k, \rho^k)$ such that π^k and ρ^k are simple permutations.
On the breakpoint graph, the "breaking of a cycle" can be described as follows. Let $b = (v_{b1}, v_{b2})$ be a black edge and let $g = (v_{g1}, v_{g2})$ be a gray edge which belongs to a cycle $c = (\dots, v_{b1}, v_{b2}, \dots, v_{g1}, v_{g2}, \dots)$ in a breakpoint graph BG^{i-1}. A (b, g)-*split* produces a new breakpoint graph BG^i which is obtained from BG^{i-1} as follows.

1. Remove the edges b and g.
2. Add two new nodes v_{xt} with label $(n + i)_t$ and v_{xh} with label $(n + i)_h$.
3. Add two new black edges (v_{b1}, v_{xt}) and (v_{xh}, v_{b2}).
4. Add two new gray edges (v_{g1}, v_{xh}) and (v_{xt}, v_{g2}).

Two examples of such splits are illustrated in Fig. 2.3. As a result of the split the cycles $(\dots, v_{b1}, v_{xt}, v_{g2}, \dots)$ and $(\dots, v_{b2}, v_{xh}, v_{g1}, \dots)$ are created.
In order to examine the effect of a (b, g)-split on the permutations, let π^{i-1} and ρ^{i-1} be permutations of size $n + i - 1$ with $BG(\pi^{i-1}, \rho^{i-1}) = BG^{i-1}$. The permutations



(a)



(b)

Figure 2.3: Two (b, g)-splits on the permutations $\pi = (\overrightarrow{3}\ \overrightarrow{2}\ \overrightarrow{1}\ \overrightarrow{5}\ \overleftarrow{6}\ \overleftarrow{4}\ \overrightarrow{7})$ and $\rho = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6}\ \overrightarrow{7})$. (a) The first split uses the edges $b = (1_h, 5_t)$ and $g = (4_t, 3_h)$. In π, the element $\overrightarrow{8}$ is placed between $\overrightarrow{1}$ and $\overrightarrow{5}$. In ρ, the element $\overrightarrow{8}$ is placed between $\overrightarrow{3}$ and $\overrightarrow{4}$. The split is unsafe, because it creates a new hurdle. (b) The second split uses the edges $b = (6_t, 4_h)$ and $g = (1_h, 2_t)$. In π, the element $\overrightarrow{8}$ is placed between $\overleftarrow{6}$ and $\overleftarrow{4}$. In ρ, the element $\overleftarrow{8}$ is placed between $\overrightarrow{1}$ and $\overrightarrow{2}$. The split is safe, and does not create any new components.

π^i and ρ^i with $BG(\pi^i, \rho^i) = BG^i$ are obtained by padding the element $(n + i)$ into π^{i-1} and ρ^{i-1}. In π^{i-1}, the element is padded between the elements which induced the black edge b in BG^{i-1}, the orientation is chosen such that it induces the black edges (v_{b1}, v_{xt}) and (v_{xh}, v_{b2}) in BG^i. In ρ^{i-1}, the element is padded between the elements that induced the gray edge g in BG^{i-1}, the orientation is chosen such that it induces the gray edges (v_{g1}, v_{xh}) and (v_{xt}, v_{g2}) in BG^i.
A (b, g)-split is *safe* if b and g are non-incident, and the split does not create a new hurdle, i.e., $h(\pi^{i-1}, \rho^{i-1}) = h(\pi^i, \rho^i)$. The first condition ensures that we do not produce a 1-cycle and a cycle with the same size as the old cycle. Because a split is acting on a long cycle, the first condition is easy to achieve. The second condition ensures that the distances $d_{rev}(\pi^{i-1}, \rho^{i-1})$ and $d_{rev}(\pi^i, \rho^i)$ are equal (note that a split increases both the size of the permutations and the number of cycles by one, and the fortress indicator cannot be changed without changing the number of hurdles). The following lemma and corollary show that to fulfill the second condition, it is sufficient to ensure that the resulting cycles belong to the same component.

Lemma 2.2. *[HP99] Let a (b, g)-split break a cycle c in $BG(\pi^{i-1}, \rho^{i-1})$ into cycles c_1 and c_2 in $BG(\pi^i, \rho^i)$. Then c is oriented if and only if c_1 or c_2 is oriented.*

In other words, if we do not split a component into two components, the orientation of the component is not changed.

Corollary 2.3. *Let a (b, g)-split break a cycle c in $BG(\pi^{i-1}, \rho^{i-1})$ into cycles c_1 and c_2 in $BG(\pi^i, \rho^i)$, such that c_1 and c_2 belong to the same component. Then, $h(\pi^{i-1}, \rho^{i-1}) = h(\pi^i, \rho^i)$.*

For a constructive proof of the existence of safe splits we need the following lemma.

Lemma 2.4. *[HP99] For every gray edge g that does not belong to a 1-cycle, there exists a gray edge f intersecting with g in $BG(\pi, \rho)$. If c is a cycle in $BG(\pi, \rho)$ and $f \notin c$ then f intersects with an even number of gray edges in c.*

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

Corollary 2.5. *Let c be a cycle of length $\ell(c)$ in $BG(\pi, \rho)$ with gray edges g_1 to $g_{\ell(c)}$. If these gray edges are pairwise non-intersecting, then there exists an index j with $1 \leq j < \ell(c)$ and a cycle $c' \neq c$ with a gray edge f, such that f intersects both g_j and $g_{\ell(c)}$.*

Proof. As c has no pairwise intersecting gray edges, $g_{\ell(c)}$ does not intersect with another gray edge of c. So Lemma 2.4 implies that $g_{\ell(c)}$ intersects with a gray edge f of another cycle c'. Because f is not in c, it intersects with an even number of gray edges in c. It follows that f intersects with at least one more gray edge g_j (with $1 \leq j < \ell(c)$) of c. □

Algorithm 2.1 (b, g)-split

```
1: function bg-split(b = (v_b1, v_b2), g = (v_g1, v_g2))
2:    create new nodes (n+1)_t, (n+1)_h
3:    v_b1.black = (n+1)_t; v_b2.black = (n+1)_h          {adjust black and gray edges}
4:    (n+1)_t.black = v_b1; (n+1)_h.black = v_b2
5:    v_g1.gray = (n+1)_h; v_g2.gray = (n+1)_t
6:    (n+1)_t.gray = v_g1; (n+1)_h.gray = v_g2
7:    (n+1)_t.position = v_b2.position; (n+1)_h.position = v_b1      {write positions}
8:    numCycles = numCycles + 1                          {update cycle information}
9:    for all v_x in new short cycle do
10:      v_x.cycle = numCycles
```

Theorem 2.6. *[HP99] If c is a long cycle in $BG(\pi, \rho)$, then there exists a safe (b, g)-split acting on a black and a gray edge of c.*

The proof given in [HP99] is constructive. However, the construction cannot transform the whole permutation into a simple permutation in linear time. Therefore, in Section 2.2.4, we provide an algorithm that achieves this goal.

2.2.3 The data structure

We represent the breakpoint graph as a linked list of $2n$ nodes. The data structure `node` for each node v consists of the three pointers `black` (pointing to the node connected with v by a black edge), `gray` (pointing to the node connected with v by a gray edge), and `co_element` (pointing to the co-element of v), and the two variables `position` (the position of the node in the breakpoint graph) and `cycle` (the index of the cycle where the node belongs to; the cycle indices can be arbitrary but must be consistent, i.e., the value of `cycle` of two nodes is equal if and only if they belong to the same cycle). We can initialize this data structure for every permutation in linear time. First, the initialization of `black`, `co_element`, and `position` can be done with a scan through the permutation. Second, for the initialization of `gray` we need the inverse permutation (mapping the nodes ordered by their label to their position) which can also be generated in linear time. Finally, we can initialize `cycle` by following the black and gray edges which also takes linear time.

Given a black edge $b = (v_{b1}, v_{b2})$ and a gray edge $g = (v_{g1}, v_{g2})$, a (b, g)-split can be performed in constant time (see Algorithm 2.1) if we assume that the split cycle contains a constant number of nodes, and disregard the problem that we have to update `position` for the new nodes and for all nodes that lie to the right of v_{b2}. Since our algorithm only uses (b, g)-splits to split 2-cycles from long cycles, the first condition is fulfilled. Furthermore, `position` is only required to determine if two edges of the same cycle intersect, thus it is sufficient if the relative positions of the nodes of each cycle

are correct, i.e., it must only hold that `v.position` < `w.position` if $v < w$ and both nodes belong to the same cycle. This information can be maintained in linear time if we set the positions of the new nodes v_{xt} and v_{xh} to the positions of the old nodes v_{b1} and v_{b2} which are now non-incident to v_{xt} or v_{xh}. After performing all splits, the breakpoint graph can easily be transformed into the simple permutations by following co-element pointers and black edges (for π_{simple}) or gray edges (for ρ_{simple}).

2.2.4 The algorithm

We now tackle the problem of transforming two permutations π and ρ into their equivalent simple permutations π_{simple} and ρ_{simple} in linear time. The algorithm has two processing phases.

Phase 1

Our goal in the first phase is to create short cycles or cycles that have no intersecting gray edges. We achieve this goal with a scanline algorithm. The algorithm requires two additional arrays: `left[j]` stores the leftmost node of each cycle c_j (i.e., the one with the least position), and `next[j]` stores the right node of the gray edge of cycle c_j which is currently checked for intersections. In both arrays, all variables are initialized with `UNDEF`. In the following, v_s denotes the current position of the scanline. Before we describe the algorithm, we first provide an invariant for the scanline.

Invariant: If $g_i = (v_{g1}, v_{g2})$ is a gray edge of the long cycle c with $i < \ell(c)$, and both nodes of g_i lie to the left of v_s (i.e., $v_{g1} < v_s$ and $v_{g2} < v_s$), then g_i does not intersect with any other gray edge of c.

It is clear that a cycle c has no intersecting edges if the invariant holds and the scanline passed the rightmost node of c_j, because $g_{\ell(c)}$ does also not intersect with a gray edge of c as the intersection relation is symmetric. Since v_s is initialized with the leftmost node of $BG(\pi, \rho)$, the invariant holds in the beginning. While the scanline has not reached the rightmost node, we repeat to analyze the following cases.

Case 1.1 v_s **is part of a short cycle**.
We move the scanline to the left node of the next black edge. As the invariant only considers long cycles, the invariant is certainly preserved.

Case 1.2 v_s **is part of a long cycle** c_j **and** `next[j] = UNDEF`.
That is, v_s is the leftmost node of cycle c_j. So we set `left[j] =` v_s. To check whether $g_1 = (v_2, v_3)$ intersects with another gray edge, we store the right node of g_1 (i.e., v_3) in `next[j]` and move v_s to the left node of the next black edge. Both nodes passed by the scanline (i.e., v_1 and v_2) are the left nodes of a gray

edge, so the set of gray edges which are located completely to the left of v_s is not changed and the invariant is preserved.

Case 1.3 v_s **is part of a long cycle** c_j **and** `next[j]` $\neq v_s$.
Let `next[j]` be the node v_{2k+1}, i.e., we check for a gray edge that intersects with g_k (going from node v_{2k} to node v_{2k+1}). Since $v_1 < v_{2k} < v_s < v_{2k+1}$, there must be a gray edge g_m belonging to c_j that intersects with g_k. We now distinguish three cases:

(a) g_k **is not** g_1 (see Fig. 2.4).
We perform a (b, g)-split with $b = b_{k+1}$ and $g = g_{k-1}$. That is, we split the 2-cycle $(v_{2k}, v_{2k+1}, v_{xt}, v_{2k-1})$ from c_j. This split is safe since g_k now lies in the 2-cycle that still intersects with g_m, which belongs to c_j, i.e., the component has not been split. The right node of the new edge g_{k-1} in c_j is v_{xh}, so we adjust `next[j]` to v_{xh}.

(b) g_k **is** g_1 **and** g_k **intersects with** $g_{\ell(c_j)}$ (see Fig. 2.5).
We perform a (b, g)-split with $b = b_1$ and $g = g_2$. That is, we split the 2-cycle (v_2, v_3, v_4, v_{xh}) from c_j. This split is safe since g_1 now lies in the 2-cycle that still intersects with $g_{\ell(c_j)}$, which belongs to c_j. After the split, $g_1 = (v_{xt}, v_5)$, so we set `next[j] = ` v_5. Note that v_5 cannot be left of v_s, as v_s is the leftmost node that belongs to c_j and has an index ≥ 4.

(c) g_k **is** g_1 **and** g_k **does not intersect with** $g_{\ell(c_j)}$ (see Fig. 2.6).
It follows that $g_m \neq g_{\ell(c_j)}$. We perform a (b, g)-split with $b = b_2$ and $g = g_{\ell(c_j)}$. That is, we split the 2-cycle (v_2, v_3, v_{xt}, v_1) from c_j. This split is safe since g_1 now lies in the 2-cycle that still intersects with g_m. As the old leftmost node and black edge of c_j lie in the 2-cycle we set `next[j] = UNDEF` which forces the re-initialization of `left[j]` with v_s.

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

Case 1.4 v_s **is part of a long cycle** c_j **and** `next[j] = ` v_s.
That is, we reach the right node of a gray edge g_k. It follows that g_k does not intersect with any other gray edge of c_j since we have not detected a node of c_j between the left and right node of g_k. Thus moving v_s to the right preserves the invariant. The next gray edge to check is $g_{k+1} = (v_{2(k+1)}, v_{2(k+1)+1})$, so we set `next[j]` to the right node of b_{k+1} and move v_s to the left node of the next black edge.

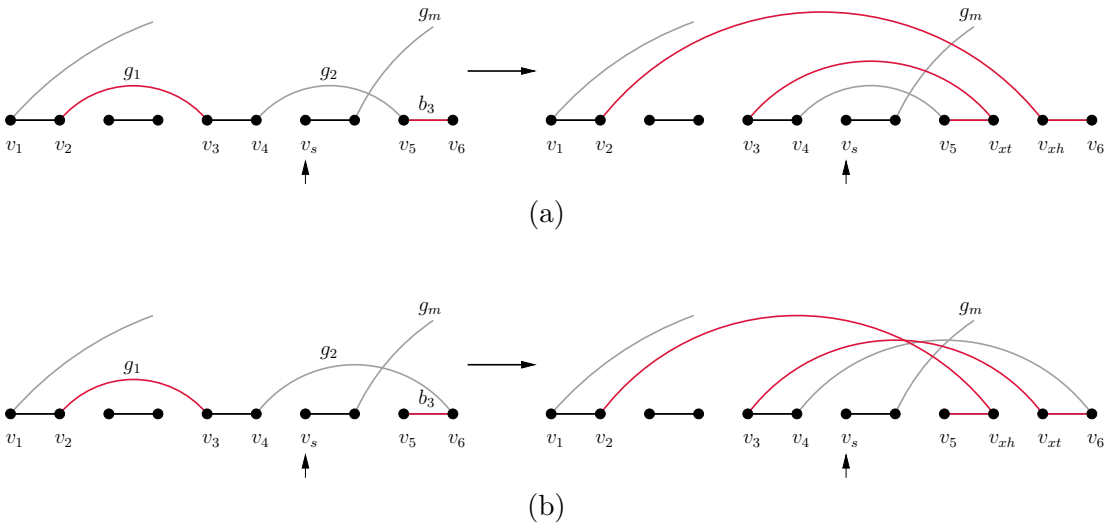


Figure 2.4: The (b, g)-split of Case 1.3(a) where $g_k = g_2$ intersects with g_m and (a) is unoriented or (b) is oriented.

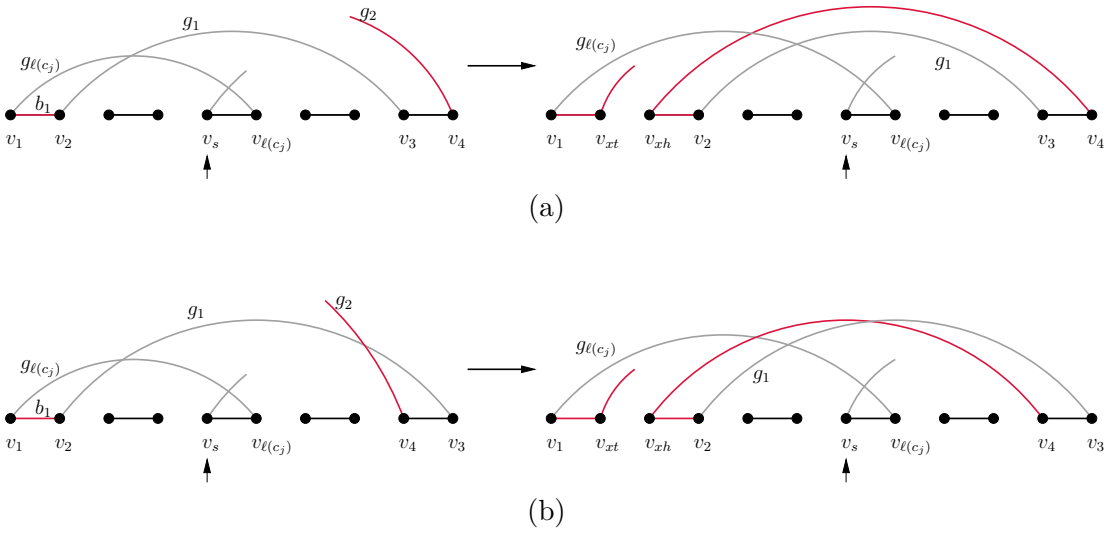


Figure 2.5: The (b, g)-split of Case 1.3(b) where $g_k = g_1$ intersects with $g_m = g_{\ell(c_j)}$ and (a) is unoriented or (b) is oriented.

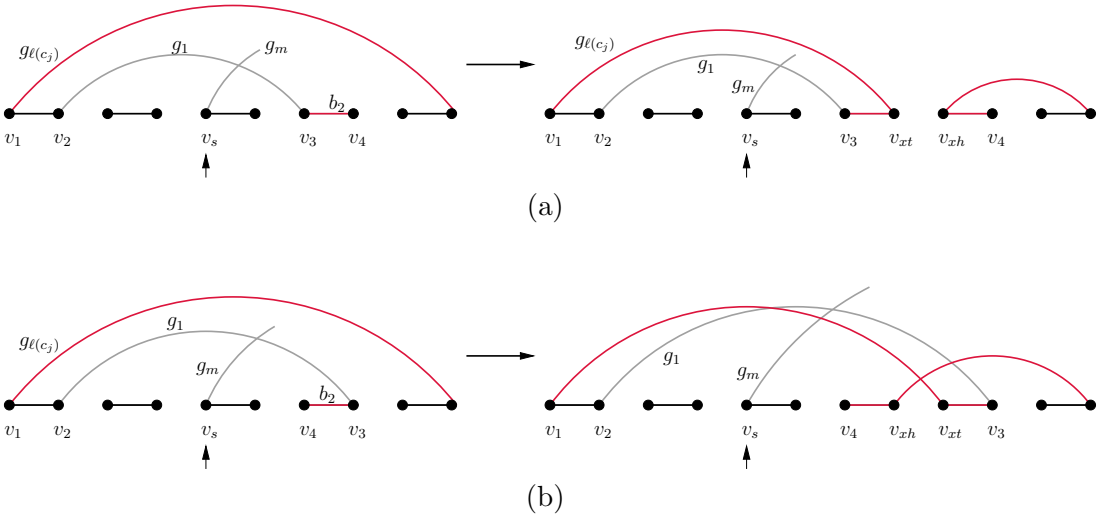


Figure 2.6: The (b, g)-split of Case 1.3(c) where $g_k = g_1$ intersects with $g_m \neq g_{\ell(c_j)}$ and (a) is unoriented or (b) is oriented.

Phase 2

After Phase 1 we can assure that there remain only short cycles and long cycles with pairwise non-intersecting gray edges. These long cycles have a special structure. The positions of the nodes $v_1, \ldots, v_{2\ell(c)}$ of a cycle c are strictly increasing and so the first $\ell(c)-1$ gray edges g_i lie one after another, and $g_{\ell(c)}$ connects the rightmost and leftmost node of c. As we know from Corollary 2.5 there exists a gray edge f of a cycle $c' \neq c$ that intersects with $g_{\ell(c)}$ and another gray edge g_k of c. In order to find for all cycles c_j such an edge f_j in linear time, we use a stack based algorithm. For each cycle c_j, I_{c_j} is the interval from the leftmost node to the rightmost node in c_j (in this phase of the algorithm, this is equivalent to the nodes of its edge $g_{\ell(c_j)}$). In each step of the algorithm, we maintain a stack of these intervals, such that each interval is contained in all other intervals that are below it on the stack (i.e., the topmost interval is contained in all other intervals on the stack). We scan the breakpoint graph from left to right. For each node v, we check whether its gray edge $f = (v, w)$ intersects with the topmost interval I_{c_j} of the stack. If so, we report the intersecting edges f and $g_{\ell(c_j)}$, pop I_{c_j} from the stack, check whether f intersects with the new top interval, and so on, until f does not intersect with the top interval. As the top interval is contained in all other intervals of the stack and Lemma 2.4 ensures that we find an intersecting edge before we reach the right end of the interval, f cannot intersect with any other interval on the stack. If v is the leftmost node of a cycle c_j, we push I_{c_j} on the stack (note that

this interval is equivalent to the gray edge $g_{\ell(c_j)}$, so it does not intersect with the topmost interval and is therefore contained in it). In all cases, we continue by moving the scanline one node to the right. The algorithm stops when we have reached the rightmost node. During the algorithm, we push the interval I_{c_j} of each cycle c_j on the stack, and pop this cycle when we reach a node v in I_j such that the gray edge (u, v) intersects with I_{c_j}. As this node must exist for each cycle, we find for each cycle c_j an edge that interleaves with $g_{\ell(c_j)}$.
Once we have found an edge f_j that intersects with the edge $g_{\ell(c_j)}$ of cycle c_j, we have to find another edge g_k of cycle c_j that intersects with f_j. This can be done by a scan over all gray edges of c_j. After determining these edges for all cycles, we distinguish two cases for a safe (b, g)-split:

Case 2.1 $g_k \neq g_{\ell(c_j)-1}$ (see Fig. 2.7(a)).
We perform the (b, g)-split on c_j with $b = b_1$ and $g = g_{\ell(c_j)-1}$. This splits c_j into $c_1 = (v_1, v_{xt}, v_{2\ell(c_j)-1}, v_{2\ell(c_j)})$ and $c_2 = (v_{xh}, v_2, \ldots, v_{2\ell(c_j)-2})$. As f_j intersects with $g_{\ell(c_j)}$, which is now part of c_1, and g_k, which is now part of c_2, the component structure remains the same.

Case 2.2 $g_k = g_{\ell(c_j)-1}$ (see Fig. 2.7(b)).
We perform the (b, g)-split on c_j with $b = b_{\ell(c_j)}$ and $g = g_1$. This splits c_j into $c_1 = (v_1, v_2, v_{xh}, v_{2\ell(c_j)})$ and $c_2 = (v_{xt}, v_3, v_4, \ldots, v_{2\ell(c_j)-1})$. As f_j intersects with $g_{\ell(c_j)}$, which is now part of c_1, and g_k, which is now part of c_2, the component structure remains the same.

In both cases, g_k is in cycle c_2 after the split, and f_j intersects with both g_k and the new gray edge $g_{\ell(c_2)}$. Thus we do not have to recalculate the edge g_k, and can repeat this step on c_2 until the remaining cycles are all 2-cycles.

Time complexity

In Phase 1, each step either moves the scanline further to the right (Cases 1.1, 1.2, and 1.4) or performs a (b, g)-split (Case 1.3). As at most n splits are required, and the resulting breakpoint graph has at most $2n$ black edges, we have to perform at most $3n$ steps. Each step can be performed in constant time, thus Phase 1 has linear running time. In Phase 2, the stack based algorithm to find the edges f_j pushes and pops each interval I_{c_j} exactly once on the stack, so these are $O(n)$ operations. Additionally, there are n gray edges checked for intersections, so this can also be done in linear time. Finding the edges g_k which intersect with the f_j takes $\sum_{j=1}^{c(\pi,\rho)} \ell(c_j) = O(n)$ time. Again, at most n (b, g)-splits must be performed, so the overall running time of Phase 2 is $O(n)$, and therefore the whole algorithm has a linear running time.

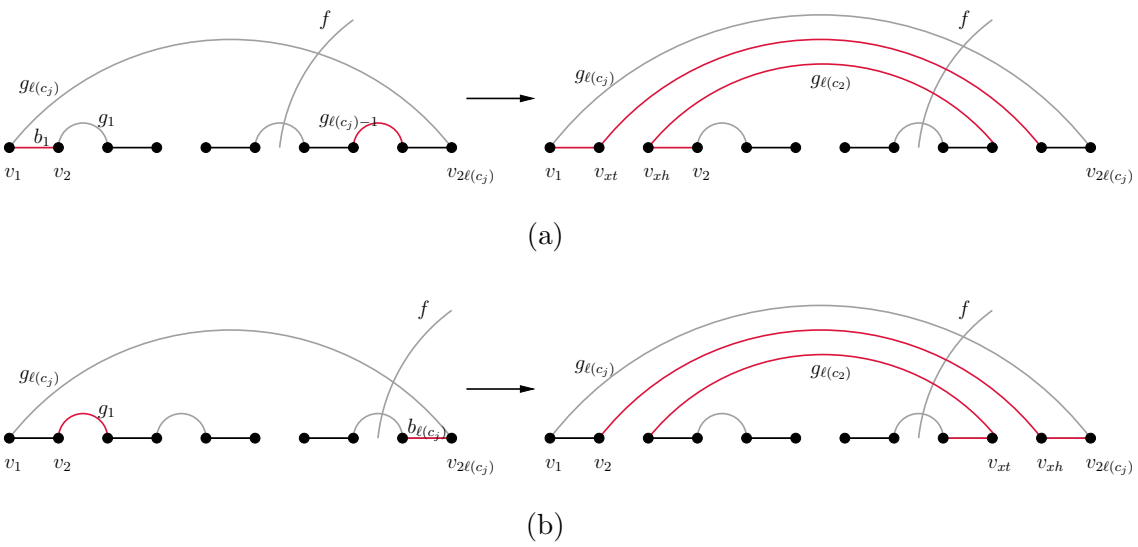


Figure 2.7: The (b, g)-split of (a) Case 2.1 and (b) Case 2.2.

2.3 Transforming back the simple permutation

In the previous section, we have shown how permutations π and ρ can be transformed into equivalent simple permutations π_{simple} and ρ_{simple}. After a sorting sequence of π_{simple} w.r.t. ρ_{simple} has been found, the remaining step is to transform it into a sorting sequence of π w.r.t. ρ. If one implements the algorithm of Hannenhalli and Pevzner [HP99] or Tannier and Sagot [TS04], the easiest way to specify a reversal is by its boundary elements. Thus, in the following, we assume that the reversals on π_{simple} are specified by their boundary elements and not by their positions. At the end of this section, we show that calculating the position of an element and vice versa can be done in logarithmic time with our data structure, which is fast enough to maintain the overall time complexity $O(n \log n)$ of the algorithm. In fact, we can simplify our data structure if the reversals on the simple permutations are specified by positions on π_{simple}.

In the naive approach, if we have a reversal $rev(x, y)$ on π_{simple} (where x and y are the boundary elements), we would scan π_{simple} beginning at x and y up to the next elements that are original elements. Then we must determine the position of these elements in π. As each of these operations requires $O(n)$ steps and $n + 1$ reversals must be performed in the worst case (i.e., the *reversal diameter* for circular permutations is $n + 1$, see [MWD00]), the whole algorithm would have quadratic running time. Thus, we now describe a data structure that supports the following two operations in logarithmic time. (1) Transform a reversal on π_{simple} into the corresponding reversal

on π in $O(\log n)$ time, and (2) update the data structure after a reversal. This allows us to transform a sorting sequence of π_{simple} w.r.t. ρ_{simple} into a sorting sequence of π w.r.t. ρ in $O(n \log n)$ time.

2.3.1 The data structure

The data structure is based on balanced binary search trees (short BBS trees), like splay trees, 2-3 trees, AVL trees, and red-black trees. The height of these trees is logarithmic in the number of their nodes, and they support concatenation of two trees and split into two trees in logarithmic time (for details on these algorithms, see [Cra72, Knu98]). In our examples, we use red-black trees (see e.g. [CLRS01]).
Let $\tilde{\pi}_1, \ldots, \tilde{\pi}_n$ be the elements in π_{simple} that correspond to the elements in π, i.e., the original elements. For $1 \leq i < n$, let I_i be the interval of padded elements that lie between $\tilde{\pi}_i$ and $\tilde{\pi}_{i+1}$ in π_{simple}. I_n is the interval of padded elements that lie between $\tilde{\pi}_n$ and $\tilde{\pi}_1$. Thus we can write $\pi_{simple} = (\tilde{\pi}_1\ I_1\ \tilde{\pi}_2\ I_2 \ldots \tilde{\pi}_{n-1}\ I_{n-1}\ \tilde{\pi}_n\ I_n)$. Note that each of these intervals may also be empty. During the algorithm, the positions of original elements and intervals will change, but original elements and intervals will always be alternating.
For each interval I_i, the order of its elements is stored in a BBS tree T_i. Each element in I_i is linked to a node in T_i. Additionally, each node in the tree has an orientation flag that indicates whether the subtree is inverted (i.e., we first have to read the right subtree in inverted order, then the element of the current node as inverted element, then the left subtree in inverted order) or not. This allows us to make a reversal of a whole subtree by just changing one flag. The real orientation of a node depends on its own orientation flag and the orientation flags of all its ancestors, i.e., if both the root node and its child node have a negative orientation flag, then the child node has a positive orientation.
The alternating order of intervals and original elements is stored in a further tree T_π, i.e., the nodes of this tree are either an original element or an interval of padded elements I_i. Each root node of a tree T_i is linked to the node I_i in T_π (see Fig. 2.8 for an example). For T_π, we use an order-statistics tree, which is a BBS tree where each node also stores the number of elements in its left and right subtree. Thus one can get the position of an element by a bottom-up traversal in logarithmic time. As original elements and intervals of padded elements are alternating in this tree, we can easily calculate the position of an original element in π if we know its position in T_π. Also in this tree, each node has the orientation flag, as described for the trees T_i.
The tree T_π is very similar to the tree proposed by Kaplan and Verbin for maintaining a permutation [KV03], with the difference that in their tree, each node corresponds to one element in the permutation, whereas the nodes in our tree either correspond to an original element or to an interval of padded elements. We will now show how we can efficiently perform the two operations on the data structure.

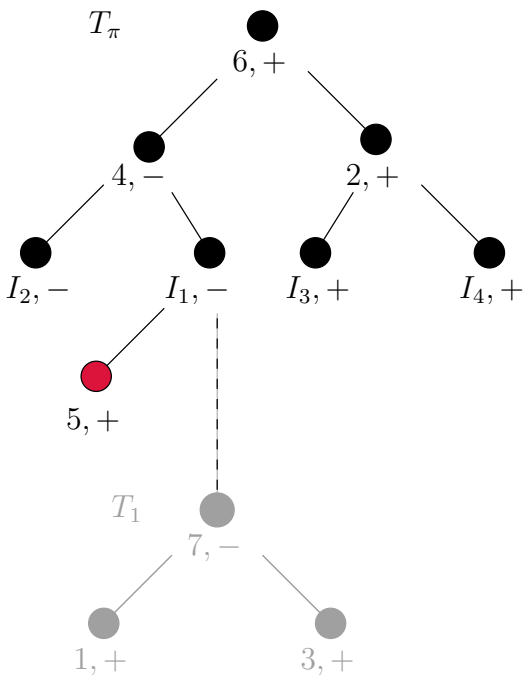


Figure 2.8: The data structure for $\pi_{simple} = (\overrightarrow{5}\ \overleftarrow{3}\ \overleftarrow{7}\ \overleftarrow{1}\ \overleftarrow{4}\ \overrightarrow{6}\ \overrightarrow{2})$, where 3, 7, and 1 are padded elements. The orientation of the nodes is indicated by the sign after its label. All interval trees except for T_1 are empty. Note that the negation of all elements in T_1 is done by an odd number of minus signs (namely 3) on the path from the nodes to the root of T_π.

2.3.2 Transforming a reversal on π_{simple} into a reversal on π

If there is a reversal on π_{simple} that is bounded by the elements x and y (lying in I_x and I_y), we traverse the corresponding trees T_x and T_y bottom-up, beginning at the corresponding nodes. This leads to two nodes in T_π, and we can also traverse this tree bottom-up to get the positions of the nodes in T_π (of course, if one of x and y is an original element, we start the tree traversal for this element directly in T_π). Having these positions, it is easy to transform them into the corresponding positions in π. As the depth of the trees is logarithmic in their size and therefore in n, this task can be done in $O(\log n)$ time.

2.3.3 Update of the data structure

Let us assume that there is a reversal bounded by the two padded elements x and y, where x lies in I_x, and y lies in I_y. W.l.o.g. I_x comes before I_y in the current permutation (otherwise the same effect can be achieved by inverting the segment from y to x and changing the sign of the root node of T_π). The reversal causes the following changes on the interval trees. If an interval I_z lies between I_x and I_y, the whole interval is inverted, i.e., the orientation flag on the root node of T_z must be changed. We cannot do this directly for each tree T_z as there are $O(n)$ trees in the worst case, but we can manage this by inverting the appropriate nodes in T_π, as we will show later (i.e., the orientation of interval I_z does not only depend on the orientation flag at the root node of T_z but also on the orientation flags on the path from I_z to the root node in T_π). Next, we split T_x into two trees T'_x and $\hat{T}_x$. Tree T'_x contains the elements of I_x that are not involved in the reversal, whereas $\hat{T}_x$ contains those that are involved. Analogously, we split T_y into the trees $\hat{T}_y$ (containing the elements of I_y that are involved in the reversal) and T'_y (containing the elements that are not involved). Now, we invert the orientation flag of the root nodes of $\hat{T}_x$ and $\hat{T}_y$ (this means an inversion of all elements in these trees), and concatenate T'_x and $\hat{T}_y$ (resulting in the updated tree T_x) as well as $\hat{T}_x$ and T'_y (resulting in the updated tree T_y). Note that the split and concatenation operations require only logarithmic time. Updating T_π works analogously, except that we have to split the tree into three trees T_l (left of inverted region), T_c (inverted region), and T_r (right of inverted region). Again, we invert the orientation flag at the root node of T_c, and merge the trees into the updated tree T_π. Note that this also affects the orientation of all intervals I_z that lie completely in the inverted region, as mentioned above.

We have described the algorithm for reversals that are bounded by two padded elements. If one of the bounding elements is an original element, the algorithm becomes even easier - we do not have to split the corresponding interval tree, everything else remains the same. For an example, see Fig. 2.9.

As mentioned above, we assumed that reversals on π_{simple} are specified by their boundary elements, because this is the most convenient way to implement the algorithm devised in [TS04]. If the reversals are specified in the usual way (i.e., by their positions), our algorithm still works, as we can get the corresponding elements with a top-down traversal of T_π. In fact, in this case we even do not need the interval trees T_i, it is sufficient to store the size of the intervals, which simplifies the algorithm.

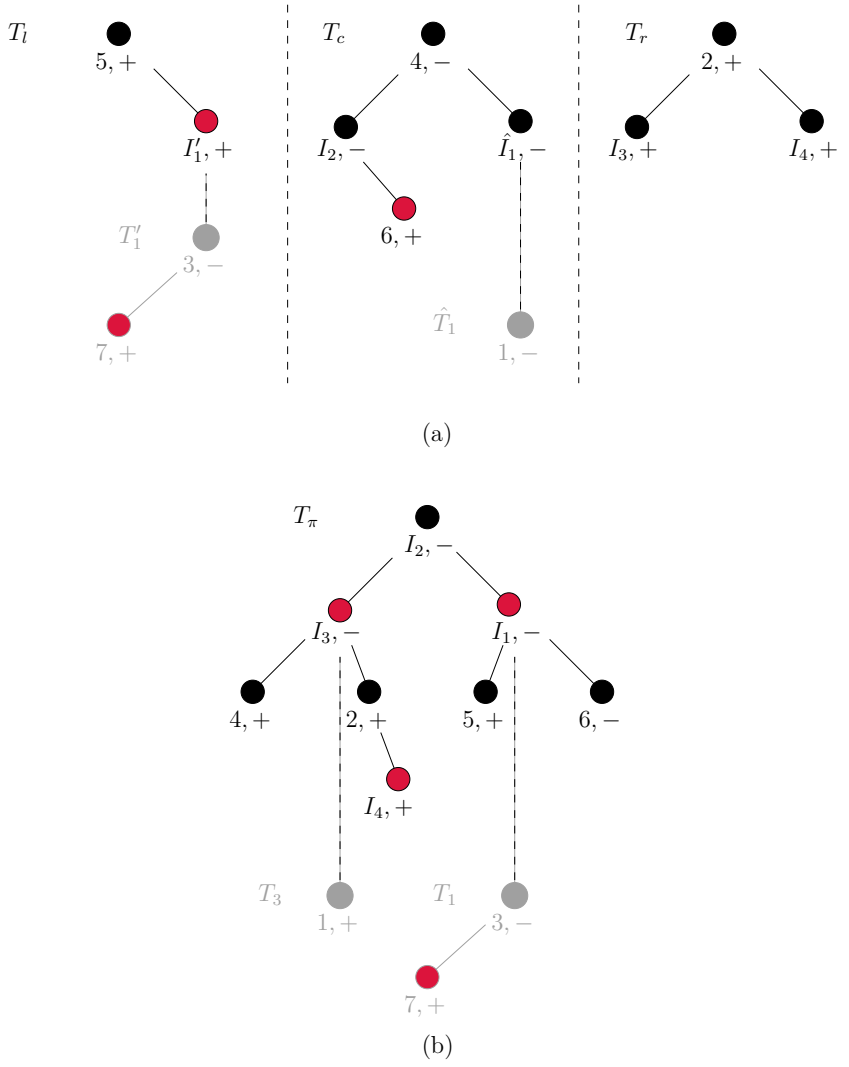


Figure 2.9: The effect of inverting the segment $\overleftarrow{1\ 4\ 6}$ in the example permutation of Fig. 2.8. (a) T_π is split into three trees T_l, T_c, and T_r. (b) The orientation flag of the root node of T_c is switched and the trees are merged, resulting in the updated tree T_π. Note that the orientation of a node depends on its own sign as well as on the signs of all its ancestors, e.g., the element 7 is inverted in the resulting permutation, as there are an odd number (namely 3) of minus signs on the path from the root node of T_π to node 7 in T_1. Also the ordering of the children depends on the sign of a node, e.g., I_1 comes before I_3 in the resulting permutation, as the root node I_2 has negative orientation.

3 On Median Problems

In this chapter, we prove the NP-completeness of the *transposition median problem* and the *weighted reversal and transposition median problem* for the weight ratios $1:1$ and $1:2$ ($w_r : w_t$). Furthermore, we describe a branch-and-bound algorithm to solve these problems exactly.
The chapter is organized as follows. In Section 3.1, some fundamental definitions and results from the literature are given. In Section 3.2, we prove the NP-completeness of the TMP. The proof is an extension of Caprara's proof of the NP-completeness of the RMP [Cap03]. The NP-completeness of the wRTMP follows directly from Caprara's proof for the weight ratio $1:2$, and from our proof for the weight ratio $1:1$. In Section 3.3, we provide an exact branch-and-bound algorithm for the TMP and the wRTMP that is fast enough to be used in practice. As a byproduct, this also includes an improved exact algorithm for the corresponding pairwise distances. In Section 3.4, possible extensions and open problems related to the results are discussed.

3.1 Fundamental definitions and results

In the *weighted reversal and transposition median problem* (short wRTMP), the input consists of three permutations π^1, π^2, and π^3. For an arbitrary permutation ρ, the value $\gamma_{wrt}(\rho) = \sum_{i=1}^{3} d_{wrt}(\rho, \pi^i)$ is called its *solution value*. The wRTMP now asks for a permutation τ such that its solution value $\gamma_{wrt}(\tau)$ is minimized. The *transposition median problem* (short TMP) is defined analogously, with the difference that the solution value is defined by $\gamma_{tp}(\rho) = \sum_{i=1}^{3} d_{tp}(\rho, \pi^i)$. Furthermore, as a transposition can never change the orientation of an element, the given permutations as well as τ must consist solely of elements with positive orientation.
In order to examine the TMP and the wRTMP, strong bounds for the corresponding pairwise distances are required. These bounds are provided by the following lemmata.

Lemma 3.1. *[BP98, BO07] A lower bound $lb_{tp}(\rho,\pi)$ for the transposition distance $d_{tp}(\rho,\pi)$ can be defined as follows.*

$$d_{tp}(\rho,\pi) \geq lb_{tp}(\rho,\pi), \text{ where } lb_{tp}(\rho,\pi) := \frac{n - c_{odd}(\rho,\pi)}{2}$$

A lower bound $lb_{wrt}(\rho,\pi)$ for the weighted reversal and transposition distance $d_{wrt}(\rho,\pi)$ can be defined as follows.

$$d_{wrt}(\rho,\pi) \geq lb_{wrt}(\rho,\pi), \text{ where } lb_{wrt}(\rho,\pi) := (n-(c_{odd}(\rho,\pi)+(2-\frac{2w_r}{w_t})c_{even}(\rho,\pi)))\frac{w_t}{2}$$

Lemma 3.2. *[BP98, BO07] An upper bound $ub_{tp}(\rho,\pi)$ for the transposition distance $d_{tp}(\rho,\pi)$ can be defined as follows.*

$$d_{tp}(\rho,\pi) \leq ub_{tp}(\rho,\pi), \text{ where } ub_{tp}(\rho,\pi) := 1.5 \cdot lb_{tp}(\rho,\pi)$$

An upper bound $ub_{wrt}(\rho,\pi)$ for the weighted reversal and transposition distance $d_{wrt}(\rho,\pi)$ can be defined as follows.

$$d_{wrt}(\rho,\pi) \leq ub_{wrt}(\rho,\pi), \text{ where } lb_{wrt}(\rho,\pi) := 1.5 \cdot lb_{wrt}(\rho,\pi)$$

In practice, the lower bounds are very tight in the majority of cases. This motivates the definition of the *weighted cycle median problem* (short wCMP) and the *odd cycle median problem* (short oCMP). Given three permutations π^1, π^2, and π^3, and two weights w_{odd} and w_{even}, the wCMP asks for a permutation τ such that its solution value $\gamma_{wc}(\tau) = \sum_{i=1}^{3}(w_{odd}c_{odd}(\tau,\pi^i) + w_{even}c_{even}(\tau,\pi^i))$ is maximized. It is easy to see that τ minimizes $\sum_{i=1}^{3} lb_{wrt}(\tau,\pi^i)$ if $w_{odd} = 1$ and $w_{even} = 2 - \frac{2w_r}{w_t}$. For the reasons given in Section 1.3, we assume that $w_{odd} = 1$ and $0 \leq w_{even} \leq 1$, which results in $w_r \leq w_t \leq 2w_r$. Analogously, the oCMP asks for a permutation τ that maximizes the solution value $\gamma_{oc}(\tau) = \sum_{i=1}^{3} c_{odd}(\tau,\pi^i)$. As the orientation of an element can never change in SBT, we further demand in the oCMP that all elements in π^1, π^2, π^3 as well as in τ have a positive orientation.
An input triple (π^1, π^2, π^3) is also called an *instance* of the corresponding problem, and a permutation τ that minimizes (or maximizes) the solution value is called a *solution* of the instance.
For the proof of the NP-completeness of the TMP, it is crucial to find permutations ρ and π such that $d_{tp}(\rho,\pi) = lb_{tp}(\rho,\pi)$. If this equation holds, we also say that ρ is *hurdle-free* w.r.t. π. Although there is currently no efficient algorithm known to decide whether a permutation ρ is hurdle-free w.r.t another permutation π in general, there are some special cases that can easily be decided.

Lemma 3.3. *Let ρ and π be two permutations. If there is a transposition $tp(i,j,k)$ such that $c_{odd}(tp(i,j,k)\cdot\rho,\pi) - c_{odd}(\rho,\pi) = 2$ and $tp(i,j,k)\cdot\rho$ is hurdle-free w.r.t. π, then also ρ is hurdle-free w.r.t. π.*

Proof.

$$\begin{aligned} lb_{tp}(\rho,\pi) &\leq d_{tp}(\rho,\pi) \\ &\leq d_{tp}(tp(i,j,k)\cdot\rho,\pi)+1 \\ &= \frac{n-c_{odd}(tp(i,j,k)\cdot\rho,\pi)}{2}+1 \\ &= \frac{n-c_{odd}(\rho,\pi)}{2} \\ &= lb_{tp}(\rho,\pi) \\ \Rightarrow d_{tp}(\rho,\pi) &= lb_{tp}(\rho,\pi) \end{aligned}$$

□

Lemma 3.4. *If the breakpoint graph of ρ and π contains only short cycles, then ρ is hurdle-free w.r.t. π.*

Proof. We prove this lemma by induction on the number of 2-cycles in the breakpoint graph. If it contains only 1-cycles, then $d_{tp}(\rho,\pi) = 0$, and ρ is clearly hurdle-free w.r.t. π. Otherwise, according to [BP96], there are two consecutive transpositions that transform two 2-cycles into four 1-cycles (note that the number of even cycles is always even, see [Chr98]), i.e., both transpositions increase the number of odd cycles by 2. As the resulting permutation is hurdle-free by induction hypothesis, the proposition follows by applying Lemma 3.3 twice. □

3.1.1 The multiple breakpoint graph

The key tool to examine median problems is the *multiple breakpoint graph* (short MB graph), due to [Cap03]. It is an extension of the classical breakpoint graph, as described in Section 1.2.4. Intuitively spoken, this graph represents the neighborhood relations of an arbitrary number of permutations. Before we give a mathematical definition of the MB graph, we first need some further definitions. Given a set of nodes $V = \{1_t, 1_h, \ldots, n_t, n_h\}$, a *matching* M of V is a set of edges such that each node in V is endpoint of at most one edge in M. If each node in V is endpoint of exactly one edge in M, the matching is called a *perfect matching*. The perfect matching associated with a permutation π is defined by

$$M(\pi) = \{(x,y) \mid x,y \text{ are adjacent in } \pi \text{ and are not co-elements}\}$$

If a perfect matching M is associated with a permutation, i.e., $M = M(\pi)$ for a permutation π, then M is called a *permutation matching*. Given permutations $\pi^1, \ldots, \pi^q$, the MB graph $MBG(\pi^1, \ldots, \pi^q) = (V, E)$ is an edge colored multigraph (i.e., it can contain *parallel* edges with common endpoints) with set of nodes $V = \{1_t, 1_h, \ldots, n_t, n_h\}$

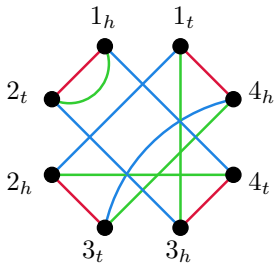


Figure 3.1: The MB graph for $\pi^1 = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4})$, $\pi^2 = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{4}\ \overrightarrow{3})$, and $\pi^3 = (\overrightarrow{1}\ \overrightarrow{4}\ \overrightarrow{3}\ \overrightarrow{2})$. The graph contains 2 odd red/green cycles, 2 odd green/blue cycles, and 2 even red/blue cycles.

and set of edges $E = M(\pi^1) \cup \cdots \cup M(\pi^q)$, where the edges of $M(\pi^i)$ have color i. In the following, let color 1 be red, let color 2 be green, and let color 3 be blue. For an example, see Fig. 3.1. The edges of two permutation matchings $M(\pi^i)$ and $M(\pi^j)$ decompose the MB graph into cycles, corresponding to the cycles in the breakpoint graph of π^i and π^j. Thus, in order to solve the oCMP or wCMP, we draw the MB graph $MBG(\pi^1, \pi^2, \pi^3)$ and search for a permutation matching $M(\tau)$ such that the corresponding weighted sum of cycles is maximized. Analogous to our previous definition, if M^i and M^j are perfect matchings, $c_{odd}(M^i, M^j)$ and $c_{even}(M^i, M^j)$ is the number of odd and even cycles in $(V, M^i \cup M^j)$. Note that this definition does not only apply to permutation matchings, but to any perfect matching.
In order to decide whether a given matching is a permutation matching, consider the perfect matching $H = \{(i_t, i_h) \mid 1 \leq i \leq n\}$, called the *base matching* of the MB graph.

Lemma 3.5. *[Cap03] A perfect matching M is a permutation matching if and only if $M \cup H$ defines a Hamiltonian cycle on the nodes of the MB graph (i.e., a cycle that visits every node of the graph exactly once).*

For a further examination of the MB graph, we need another important notion, introduced in [Cap03]. Given a perfect matching M of a set of nodes V and an edge $e = (u, v)$, M/e is defined as follows. If $e \in M$, $M/e = M \setminus \{e\}$. Otherwise, letting $(a, u), (b, v)$ be the two edges in M incident to u and v, $M/e = M \setminus \{(a, u), (b, v)\} \cup \{(a, b)\}$. Given an MB graph $G = (V, M(\pi^1) \cup \cdots \cup M(\pi^q))$, the *contraction* of an edge $e = (u, v)$ yields the graph $G/e = (V \setminus \{u, v\}, M(\pi^1)/e \cup \cdots \cup M(\pi^q)/e)$. For an example, see Fig. 3.2.

Lemma 3.6. *[Cap03] Given two perfect matchings M, L of V and an edge $e = (u, v) \in M$ with $e \notin L$, $M \cup L$ defines a Hamiltonian cycle of V if and only if $(M/e) \cup (L/e)$ defines a Hamiltonian cycle of $V \setminus \{u, v\}$.*

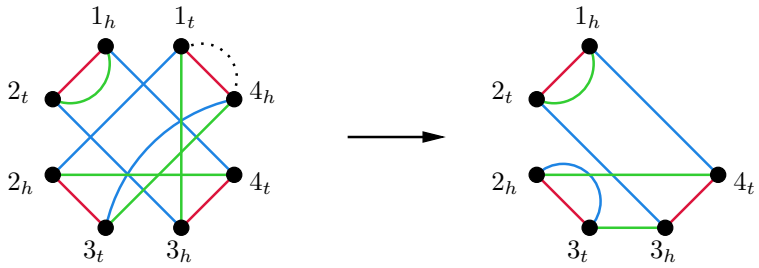


Figure 3.2: The contraction of the edge $(4_h, 1_t)$ in the left graph (dotted edge) yields the right graph.

In the proof of the NP-completeness of the TMP in Section 3.2, we construct a graph G and claim that this graph is isomorphic to an MB graph, i.e., the nodes of the graph can be labeled such that there are permutations $\pi^1, \ldots, \pi^q$ with $MBG(\pi^1, \ldots, \pi^q) = G$. In order to prove this claim, the following lemmata are required. The first of them has already been proven in [Cap03], except for the splitting of the set of nodes into V^t and V^h, which is not required in the RMP.

Lemma 3.7. *Let V^t and V^h be two disjoint sets of nodes, and let $G = (V^t \cup V^h, M^1 \cup M^2 \cup \cdots \cup M^q)$ be an edge-colored graph, where each M^i is a perfect matching, each edge in M^i has color i, and each edge connects a node in V^t with a node in V^h. Furthermore, let H be a perfect matching such that each edge in H connects a node in V^t with a node in V^h, and $H \cup M^i$ defines a Hamiltonian cycle of $V^t \cup V^h$ for $1 \leq i \leq q$. Then, there are permutations $\pi^1, \ldots, \pi^q$ such that G is isomorphic to the MB graph $MBG(\pi^1, \ldots, \pi^q)$, and all elements in $\pi^1, \ldots, \pi^q$ have a positive orientation.*

Proof. We give a constructive proof on how to create the permutations $\pi^1, \ldots, \pi^q$ such that G is isomorphic to the MB graph $MBG(\pi^1, \ldots, \pi^q)$. For this, set $n = |V^t|$. Now, arbitrarily label the nodes in V^t with $1_t, \ldots, n_t$. Label the nodes in V^h such that $H = \{(i_t, i_h) \mid 1 \leq i \leq n\}$. Let $j_{1t}, j_{1h}, j_{2t}, \ldots, j_{nh}, j_{1t}$ be the Hamiltonian cycle defined by $H \cup M^j$, starting at an arbitrary node $j_{1t} \in V_t$. Set $\pi^j = (\overrightarrow{j_1} \ \ldots \ \overrightarrow{j_n})$. It is clear to see that π^j is a valid permutation, and the perfect matching associated with π^j is M^j. Therefore, with the given node labeling, $MBG(\pi^1, \ldots, \pi^q) = G$, and H is the base matching of the MB graph. □

To simplify our argumentation, a perfect matching H that fulfills the conditions of Lemma 3.7 is called a *base matching* of G.

Lemma 3.8. *Let V^t and V^h be two disjoint sets of nodes, and let $G = (V = V^t \cup V^h, M^1 \cup M^2)$ be an edge-colored graph, where M^1 (M^2) is a perfect matching with*

red (green) edges, and each edge connects a node in V^t with a node in V^h. If $M^1 \cup M^2$ defines an even number of even cycles on V, then G has a base matching H.

Proof. We prove this lemma by an induction on the size of V^t. If $|V^t| = 1$, then the graph consists of just one parallel red and green edge, and there trivially exists a base matching H. For $|V^t| > 1$, we must distinguish two cases. If $M^1 \cup M^2$ defines at least two cycles on V, then there are nodes $u \in V^t$ and $v \in V^h$ such that these nodes are in different cycles. The contraction of $e = (u, v)$ merges these two cycles, and the resulting cycle is even if and only if exactly one of the merged cycles was even. In other words, the contraction of e reduces $|V^t|$ by 1 and does not change the parity of the number of even cycles. Due to the induction hypothesis, G/e has a base matching H'. According to Lemma 3.6, $H := H' \cup \{e\}$ is a base matching of G. The case where $M^1 \cup M^2$ defines just one cycle can be proven similarly. This cycle must be odd, and has at least length 3. Therefore, there are nodes $u \in V^t$ and $v \in V^h$ such that the edge $e = (u, v)$ is neither in M^1 nor in M^2. The contraction of e splits the cycle, and again the parity of the number of even cycles cannot be changed. With the same argumentation as above, it follows that the base matching H' of G/e can be extended to a base matching $H := H' \cup \{e\}$ of G. □

3.2 The transposition median problem is NP-complete

In order to prove the NP-completeness of the TMP, it is more convenient to formulate the TMP and the oCMP as decision problems. For this, let π^1, π^2, and π^3 be permutations, and let k be an integer. Then, $(\pi^1, \pi^2, \pi^3, k) \in TMP$ if and only if there is a permutation τ such that $\sum_{i=1}^{3} d_{tp}(\tau, \pi^i) \leq k$. Equivalently, $(\pi^1, \pi^2, \pi^3, k) \in oCMP$ if and only if there is a permutation τ such that $\sum_{i=1}^{3} c_{odd}(\tau, \pi^i) \geq k$.
As transpositions cannot change the orientation of an element, we will only consider permutations where all elements have a positive orientation. This slightly changes the definition of a permutation matching. That is, in this section, a matching M is a permutation matching if and only if there is a permutation π with $M = M(\pi)$, and all elements of π have a positive orientation.

Lemma 3.9. *$TMP \in NP$.*

Proof. The membership of TMP in NP can be shown by a simple "guess and check" argument. First, we nondeterministically "guess" a permutation τ and three sequences of transpositions $tp_1^1 \dots tp_{k_1}^1$, $tp_1^2 \dots tp_{k_2}^2$, and $tp_1^3 \dots tp_{k_3}^3$ with $\sum_{i=1}^{3} k_i \leq k$. Then, we check whether applying $tp_1^i \dots tp_{k_i}^i$ on π^i yields τ for $i \in \{1, 2, 3\}$. Naturally, this can hold only if $(\pi^1, \pi^2, \pi^3, k) \in TMP$. On the other hand, if $(\pi^1, \pi^2, \pi^3, k) \in TMP$, a nondeterministical machine will correctly "guess" the median τ and the sequences of transpositions. As the whole algorithm can be performed in polynomial time, it follows that $TMP \in NP$. □

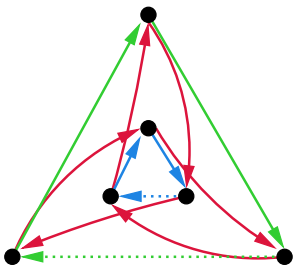


Figure 3.3: A cycle decomposition of a graph in $mdECD$. The marked edges are drawn dotted, the different cycles are drawn in different colors.

The proof of the NP-hardness of the TMP consists of several polynomial reductions, beginning at the well-known 3SAT problem. More precisely, we prove that $3SAT \leq_p mdECD \leq_p oCMP \leq_p TMP$, where mdECD is the *marked directed Eulerian cycle decomposition problem*, which is defined as follows. Let k be an integer, let $G = (V, E)$ be a directed graph, and let $E_k \subseteq E$ be a subset of its edges with $|E_k| = k$. The edges in E_k are called the *marked edges* of G. Then, $(G, E_k) \in mdECD$ if and only if G can be partitioned into edge-disjoint cycles such that each marked edge is in a different cycle. Note that the decomposition may contain cycles that do not contain a marked edge (see Fig. 3.3). This problem is a slight modification of the *Eulerian cycle decomposition problem* (short ECD), which has been proven to be NP-hard by Holyer [Hol81].

Lemma 3.10. *$mdECD$ is NP-hard.*

Proof. By following the proof for ECD in [Hol81] and simply directing the edges in the graph construction, one can prove that partitioning a directed graph into edge-disjoint cycles of length 3 is NP-hard (see also [AHK+07]). Furthermore, the edges of the graph can be partitioned into 3 groups such that each cycle of length 3 must contain one edge of each group, and this partitioning can be found in polynomial time. While Holyer used this fact to extend his proof to cycles of arbitrary length, we mark all edges of one group, i.e., each possible cycle of length 3 contains exactly one marked edge. This completes the proof for $k = |E|/3$. □

In the following, we will assume that for an mdECD instance (G, E_k), each node has the same in- and out-degree, and G is connected.

3.2.1 Reduction from mdECD to oCMP

In order to prove the NP-hardness of the oCMP, we first have to show that mdECD is NP-hard even when the in- and out-degree of all nodes is bounded by 2. Next, we provide a transformation from a directed graph G with bounded degree to an

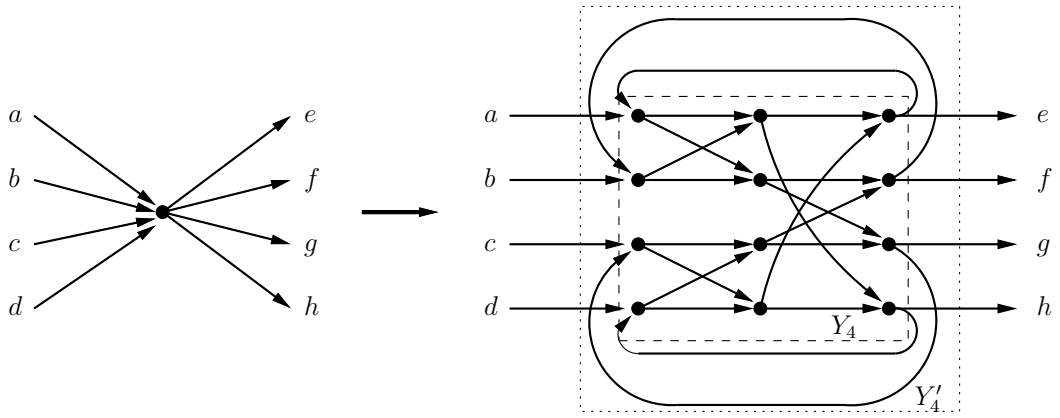


Figure 3.4: Transformation of a node v with degree 8 into a Y_4'.

MB graph $MBG(\pi^1, \pi^2, \pi^3)$ such that $(G, E_k) \in mdECD \Leftrightarrow (\pi^1, \pi^2, \pi^3, f(G, E_k)) \in oCMP$ (where $f(G, E_k)$ is a function that can be evaluated in polynomial time).
A *permutation network* is a directed graph Y_d where $2d$ of the nodes are labelled by $i_1, \ldots, i_d, o_1, \ldots, o_d$ (the input and output nodes). Furthermore, for each permutation σ in the symmetric group $\mathfrak{S}_d$, there are edge-disjoint paths $p_1, \ldots, p_d$ in Y_d such that path p_j goes from i_j to $o_{\sigma(j)}$.

Lemma 3.11. *[Wak68] For each d, a permutation network Y_d of size $O(d \log d)$ can be constructed in polynomial time. Furthermore, for each node v in Y_d, the following proposition holds.*

- $deg_{in}(v) = 0, deg_{out}(v) = 2$ *if v is an input node.*
- $deg_{out}(v) = 0, deg_{in}(v) = 2$ *if v is an output node.*
- $deg_{in}(v) = deg_{out}(v) = 2$ *if v is an inner node.*

By adding edges from the output nodes to the input nodes, it is possible to obtain a permutation network Y_d' where $deg_{out}(i) - deg_{in}(i) = 1$ for all input nodes i, and $deg_{in}(o) - deg_{out}(o) = 1$ for all output nodes o.
Let G be a directed graph with k marked edges. We obtain the graph G' by replacing each node v in G with degree $d > 4$ by a $Y_{d/2}'$. The incoming edges in v are arbitrarily connected to the input nodes of the corresponding $Y_{d/2}'$, and the outgoing edges in v are arbitrarily connected to its output nodes (see Fig. 3.4). Note that in G', all nodes v satisfy $deg_{in}(v) = deg_{out}(v) \leq 2$.

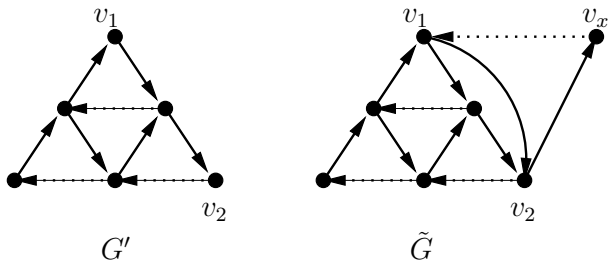


Figure 3.5: Transformation of $G' = (V, E)$ into $\tilde{G} = (\tilde{V}, \tilde{E})$, such that $|\tilde{V}| + |\tilde{E}| - (k+1)$ is odd. Marked edges are dotted.

Lemma 3.12. *$(G, E_k) \in mdECD$ if and only if $(G', E_k) \in mdECD$.*

Proof. If $(G, E_k) \in mdECD$, we can map the cycles in G to cycles in G' by adding the corresponding paths through the permutation network for each node in a cycle. As the paths through the permutation network are edge-disjoint, the cycles in G' also are edge-disjoint. Because all nodes v satisfy $deg_{in}(v) = deg_{out}(v)$, the remaining edges in the permutation network can be partitioned into edge-disjoint cycles. Thus, G' can be partitioned into edge-disjoint cycles and each marked edge is in a different cycle, i.e., $(G', E_k) \in mdECD$. On the other hand, if $(G', E_k) \in mdECD$, then we can remove the paths in the permutation networks from each cycle to obtain a cycle decomposition of G, i.e., $(G, E_k) \in mdECD$. □

The transformation from G to G' can be computed in polynomial time, i.e., the construction of G' describes a polynomial reduction from mdECD to mdECD with bounded node degrees.

Theorem 3.13. *mdECD is NP-hard even when the degree of all nodes is bounded by 4. Furthermore, the claim still holds for graphs where $|V| + |E| - k$ is odd.*

Proof. The first proposition directly follows from Lemmata 3.11 and 3.12. Now, assume that our transformation resulted in a graph $G' = (V, E)$ where $|V| + |E| - k$ is even. We further transform G' into $\tilde{G} = (\tilde{V}, \tilde{E})$ as follows (see also Fig. 3.5). Let v_1 and v_2 be two nodes of degree 2 that are not connected by an edge (if no such nodes exist, they can be created by splitting a non-marked edge without changing the parity of $|V| + |E| - k$). Create a new node v_x and set $\tilde{V} = V \cup \{v_x\}, \tilde{E} = E \cup \{(v_1, v_2), (v_2, v_x), (v_x, v_1)\}$, where (v_x, v_1) is a marked edge. As we added a cycle with one marked edge, it is clear to see that if $(G', E_k) \in mdECD$, then $(\tilde{G}, E_k \cup \{(v_x, v_1)\}) \in mdECD$. On the other hand, each cycle decomposition of $\tilde{G}$ can be modified such that the added edges form one cycle. This leads to a cycle decomposition of G', i.e., $(\tilde{G}, E_k \cup \{(v_x, v_1)\}) \in mdECD$ implies $(G', E_k) \in mdECD$. Together with the fact that $|\tilde{V}| + |\tilde{E}| - (k+1)$ is odd, the second proposition follows. □

Now, let $G = (V, E)$ be a directed graph with a set of k marked edges E_k, $deg_{in}(v) = deg_{out}(v) \leq 2\ \forall v \in V$, and $|V| + |E| - k$ odd. Let V_2 be the nodes with $deg(v) = 2$, and let V_4 be the nodes with $deg(v) = 4$. Let $\mathcal{E}$ be an Eulerian cycle in G (which clearly exists and can be computed in polynomial time, since a connected directed graph has an Eulerian cycle if and only if every vertex has an in-degree equals to its out-degree). We will now describe a polynomial transformation from G into a graph $G' = (V', E' = M^1 \cup M^2 \cup M^3)$, such that G' is isomorphic to an MB graph. The intuition behind this transformation is that many edges of the odd cycle median of G' are predetermined by this construction: each of its edges (in the following with color black) is parallel to a red or a green edge, thus the number of red/black and green/black cycles is fixed. The blue/black cycles correspond to cycles in G, and a blue/black cycle can only be odd if the corresponding cycle in G contains a marked edge. For a graphical representation of the transformation, see Fig. 3.6.

1. For each node $v \in V_2$, G' contains a subgraph W_2 with set of nodes $\{v_t, v_h\}$, and a parallel red and green edge (v_t, v_h). The node v_t is called the input node of W_2, and v_h is called the output node.

2. For each node $v \in V_4$, G' contains a subgraph W_4 with set of nodes $\{v_{1t}, v_{1h}, \ldots, v_{4t}, v_{4h}\}$, red edges $\{(v_{1t}, v_{3h}), (v_{2t}, v_{2h}), (v_{3t}, v_{1h}), (v_{4t}, v_{4h})\}$, green edges $\{(v_{1t}, v_{2h}), (v_{2t}, v_{1h}), (v_{3t}, v_{3h}), (v_{4t}, v_{4h})\}$, and blue edges $\{(v_{3t}, v_{4h}), (v_{4t}, v_{3h})\}$. The nodes v_{1t} and v_{2t} are called the input nodes of W_4, and v_{1h} and v_{2h} are called the output nodes.

3. For each edge $(u, v) \in E_k$ (i.e., the marked edges), there is a blue edge (u', v') in G' which connects the corresponding subgraphs of u and v. u' is always an output node of the corresponding subgraph, and v' is an input node of the corresponding subgraph.

4. For each edge $(u, v) \in E \setminus E_k$ (i.e., the non-marked edges), G' contains two nodes v_t, v_h, a parallel red and green edge (v_t, v_h), and two blue edges (u', v_t) and (v_h, v'), where u' is an output node of the subgraph corresponding to u, and v' is an input node of the subgraph corresponding to v.

5. The endpoints of a blue edge are always chosen such that each node is incident to exactly one blue edge. Furthermore, if $v \in V_4$ and $\mathcal{E}$ contains two consecutive edges $(u, v), (v, w)$, the corresponding blue edges are either of the form $(u', v_{1t}), (v_{2h}, w')$ or $(u', v_{2t}), (v_{1h}, w')$. Thus, the Eulerian cycle $\mathcal{E}$ is transformed into a cycle of alternating green and blue edges that passes each V_2 and V_4. This cycle contains one green edge for each V_2 and for each non-marked edge, and two green edges for each V_4.

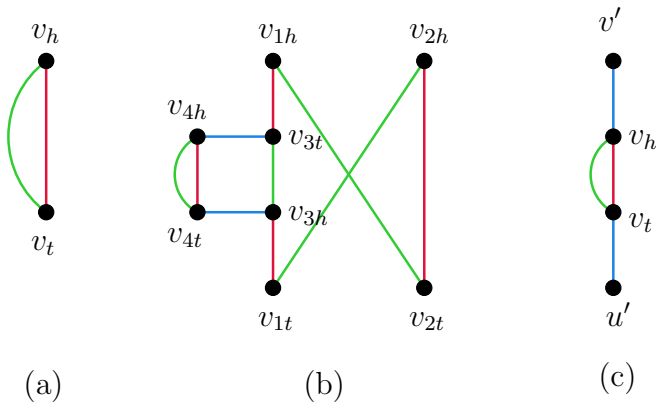


Figure 3.6: Transformation steps from a graph G to a graph G', such that G' is isomorphic to a MB graph. (a) a W_2 (b) a W_4 (c) transformation of a non-marked edge.

Lemma 3.14. *G' is isomorphic to an MB graph, and a base matching H of G' can be calculated in polynomial time.*

Proof. The set of nodes V' can be divided into V^t and V^h such that V^t contains all nodes $v_t, v_{1t}, v_{2t}, v_{3t}, v_{4t}$, and V^h contains all nodes $v_h, v_{1h}, v_{2h}, v_{3h}, v_{4h}$. Thus, all red, green, and blue edges connect a node in V^t with a node in V^h, and the edges of each color are a perfect matching of V'. Therefore, according to Lemma 3.7, it remains to show that there is a base matching H. This base matching can be built iteratively as follows. For each W_4 in G', we add the edges (v_{2t}, v_{3h}), (v_{3t}, v_{2h}), and (v_{4t}, v_{1h}) to H. Then, we contract these edges. Let $(u, v_{1t}), (v, v_{2t}), (w, v_{1h}), (x, v_{2h})$ be the incoming/outgoing blue edges of a W_4. Then, after the contraction, we have the blue edges $(u, v_{1t}), (x, v_{4h}), (v, w)$ and a parallel red and green edge (v_{1t}, v_{4h}). If we would now merge v_{1t} and v_{4h}, we would restore the Eulerian cycle $\mathcal{E}$. Therefore, we have now an Eulerian cycle of alternating blue and red/green edges. This cycle contains one green edge for each V_2, for each V_4, and for each non-marked edge (the second green edge of each V_4 has been absorbed by the contraction). This is equivalent to the number of vertices plus the number of non-marked edges in G. Because we assumed that in G, $|V| + |E| - k$ is odd, this cycle is also odd. Therefore, the preconditions of Lemma 3.8 are fulfilled, and we can continue with the algorithm devised there. □

For simplification, we assume that $G' = MBG(\pi^1, \pi^2, \pi^3)$, i.e., we ignore the incorrect labeling of the nodes and say that G' is an MB graph. We will now look at several perfect matchings M and examine whether they are permutation matchings and how much odd cycles are defined by $M^i \cup M$ for $1 \leq i \leq 3$. In the following, let the color of

these matchings be black. That is, if we speak of red/black odd cycles, we mean the odd cycles defined by $M^1 \cup M$.

Lemma 3.15. *Let G' be an MB graph with base matching H that has been constructed as described above. Then, every perfect matching M containing only edges parallel to red or green edges is a permutation matching.*

Proof. If we divide V' into V^t and V^h as described above, every edge of M connects a node in V^t with a node in V^h. From the proof of Lemma 3.7, it follows that if M is a permutation matching, the corresponding elements all have a positive orientation. According to Lemma 3.5, it remains to show that $H \cup M$ defines a Hamiltonian cycle on V'. For every W_4 in the MB graph, M must be parallel either to all green edges or to all red edges. Therefore, we can successively contract all edges of H that are in a W_4, and the contracted edges are never parallel to an edge in M (otherwise, according to Lemma 3.6, $H \cup M^1$ or $H \cup M^2$ would not define a Hamiltonian cycle). After contracting these edges, M^1, M^2, and M are identical. Because $H \cup M^1$ and $H \cup M^2$ define Hamiltonian cycles, also $H \cup M$ must define a Hamiltonian cycle. □

We call a perfect matching M *canonical* if, whenever there are two parallel edges in $M^1 \cup M^2 \cup M^3$, these edges are also parallel to an edge in M.

Lemma 3.16. *Given a perfect matching M on G', it is always possible to find a canonical matching M^c with $\sum_{i=1}^{3} c_{odd}(M^c, M^i) \geq \sum_{i=1}^{3} c_{odd}(M, M^i)$.*

Proof. Due to the construction of G', only red and green edges can be parallel. Let M be a perfect matching, and let u and v be two nodes that are connected by a red and a green edge, but not by a black edge. Assume that there are black edges (x, u) and (y, v). We replace these edges with the black edges (x, y) and (u, v). This transformation has the following three effects.

1. An even red/black cycle is split into two odd red/black cycles (1a), or an odd red/black cycle is split into an odd and an even red/black cycle (1b).

2. An even green/black cycle is split into two odd green/black cycles (2a), or an odd green/black cycle is split into an odd and an even green/black cycle (2b).

3. a blue/black cycle is split into two cycles (3a), the set of blue/black cycles remains unchanged (3b), two blue/black cycles are merged and at least one of the cycles was even (3c), or two odd blue/black cycles are merged into an even cycle (3d).

Effects 1a and 2a increase the number of odd cycles by 2. Effect 3d decreases the number of odd cycles by 2, all other effects do not decrease the number of odd cycles. Therefore, the transformation decreases the number of odd cycles if and only if effects 1b, 2b, and 3d occur simultaneously. In this case, the number of odd cycles is decreased by 2, and

the red/black, green/black, and blue/black cycle containing the black edge (x, y) are even cycles. Therefore, exchanging the endpoints of (x, y) with those of another black edge cannot remove any further odd cycle. However, such an exchange is possible such that the red/black cycle is split into two odd cycles, i.e., this operation increases the number of odd cycles by at least 2. Both operations together do not decrease the overall number of odd cycles. These steps can be repeated until M is a canonical matching. □

Lemma 3.17. *For every canonical matching M^c on G', $c_{odd}(M^c, M^3) \leq k$.*

Proof. By construction, G' contains pairs of blue edges that are separated by a parallel red and green edge. These pairs contain all blue edges, except some of those that correspond to a marked edge in G. Thus, every odd blue/black cycle must contain at least one blue edge corresponding to a marked edge in G. As there are only k marked edges in G, there can be at most k odd blue/black cycles if the black edges are a canonical matching. □

Lemma 3.18. *For every perfect matching M, $c_{odd}(M, M^1) + c_{odd}(M, M^2) \leq 2|V_2| + 6|V_4| + 2|E| - 2k$. The equality holds if and only if all black edges are parallel to a red or green edge.*

Proof. If e is a black edge in a red/black k-cycle, then let the *red score* of e be $1/k$ if k is odd, 0 otherwise. The *green score* is defined analogously for green/black cycles. The *score* of a black edge is the sum of its red and green score. Clearly, the number of red/black and green/black odd cycles is the sum of the scores of all black edges. If a red edge, a green edge, and a black edge are parallel, then the score of the black edge is 2, which maximizes this value. This score can be achieved by at most $|V_2| + |V_4| + |E| - k$ black edges, because this is the number of parallel red and green edges. The second best possible score is 4/3, and it is achieved if and only if a black edge is in a red/black 1-cycle and a green/black 3-cycle or vice versa. If all edges in a perfect matching are parallel to a red or green edge, the black edges in each W_4 are parallel to edges of the same color, forming four 1-cycles with this color and a 1-cycle and a 3-cycle with the edges of the other color. Thus, the number of black edges with score 2 is maximized, all other black edges have score 4/3, leading to an overall score of $2 \cdot \#W_2 + 2 \cdot \#W_4 + 2 \cdot \#\text{non-marked edges} + 4 \cdot \#W_4 = 2|V_2| + 6|V_4| + 2|E| - 2k$. If the matching contains a black edge that is neither parallel to a red nor to a green edge, then the score of this edge is $< 4/3$, and the sum of all scores can no longer be maximal. □

Theorem 3.19. *There is a permutation matching $M(\tau)$ with $\sum_{i=1}^{3} c_{odd}(\tau, \pi^i) \geq 2|V_2| + 6|V_4| + 2|E| - k$ if and only if $(G, E_k) \in mdECD$.*

Proof. According to Lemma 3.16, it is sufficient to consider canonical matchings. Together with Lemmata 3.17 and 3.18, it follows that the maximum number of odd

cycles is $2|V_2| + 6|V_4| + 2|E| - k$, and this can only be achieved if each black edge is parallel to a red or green edge. Then, all black edges in one W_4 must be parallel to edges of the same color. Depending on whether they are parallel to the red or the green edges, we get blue/black paths from v_{1t} to v_{1h} and from v_{2t} to v_{2h}, or from v_{1t} to v_{2h} and from v_{2t} to v_{1h}. Thus, there is a one-to-one correspondence between black/blue cycles in G' and cycles in G (except for possible even black/blue cycles that are completely within a W_4). A black/blue cycle in G' is odd if the corresponding cycle in G contains an odd number of marked edges. Therefore, if there is a permutation matching $M(\tau)$ with $\sum_{i=1}^{3} c_{odd}(\tau, \pi^i) = 2|V_2| + 6|V_4| + 2|E| - k$, it defines k blue/black odd cycles in G'. These cycles describe the partitioning of G into k edge-disjoint cycles such that each cycle contains a marked edge, i.e., $(G, E_k) \in mdECD$. On the other hand, such a partitioning of G can be used to obtain a permutation matching $M(\tau)$ with $\sum_{i=1}^{3} c_{odd}(\tau, \pi^i) = 2|V_2| + 6|V_4| + 2|E| - k$. □

Corollary 3.20. *oCMP is NP-complete.*

3.2.2 Reduction from oCMP to TMP

To prove the NP-hardness of the TMP, we describe a transformation from an MB graph $G = MBG(\pi^1, \pi^2, \pi^3)$ into an MB graph $\tilde{G} = MBG(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3)$, such that every permutation $\tilde{\tau}$ that minimizes $\sum_{i=1}^{3} d(\tilde{\tau}, \tilde{\pi}^i)$ also maximizes $\sum_{i=1}^{3} c_{odd}(\tilde{\tau}, \tilde{\pi}^i)$. Let $G = MBG(\pi^1, \pi^2, \pi^3) = (V, M(\pi^1) \cup M(\pi^2) \cup M(\pi^3))$ be an arbitrary MB graph with base matching H that satisfies the following condition.

Condition 3.1. *There is a permutation τ such that for all perfect matchings M, $\sum_{i=1}^{3} c_{odd}(M, M(\pi^i)) \leq \sum_{i=1}^{3} c_{odd}(M(\tau), M(\pi^i))$, and $M(\tau)$ is a canonical matching.*

Note that in the last section, if the starting problem is in mdECD, the resulting MB graph satisfies the condition.
First, we modify the MB graph such that τ is hurdle-free w.r.t. π^1. Although we do not know $M(\tau)$, we can presume that some edges of $M(\tau)$ are given due to the fact that it is a canonical matching. With these edges, we already get some red/black cycles and paths. Thus, there are red edges that are certainly not in a long red/black cycle. Let (u, v) be a red edge that might be in a long red/black cycle, and let (x, u), (v, y) be the adjacent edges of the base matching H. We transform the MB graph $G = (V, M^1 \cup M^2 \cup M^3)$ into an MB graph $\ddot{G} = MBG(\ddot{\pi}^1, \ddot{\pi}^2, \ddot{\pi}^3) = (\ddot{V}, \ddot{M}^1 \cup \ddot{M}^2 \cup \ddot{M}^3)$ with base matching $\ddot{H}$ as follows (for a graphical representation, see Fig. 3.7).

1. $\ddot{V} = V \cup \{a, b, c, d, e, f, g, h\}$.
2. $\ddot{H} = H \setminus \{(x, u), (v, y)\} \cup \{(x, a), (b, c), (d, e), (f, u), (v, g), (h, y)\}$
3. $\ddot{M}^1 = M(\pi^1) \cup \{(a, b), (c, d), (e, f), (g, h)\}$

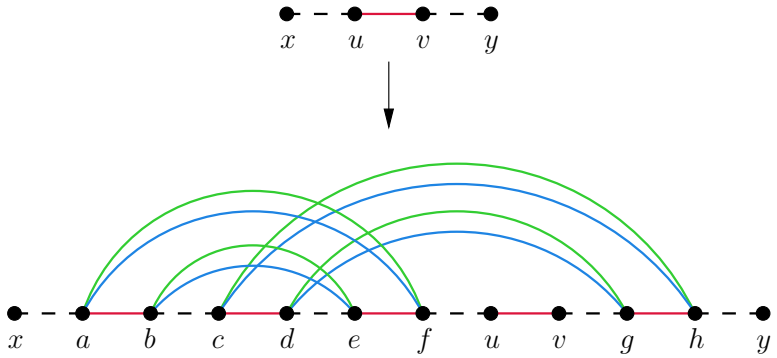


Figure 3.7: Transformation of a configuration of G containing a red edge (u, v) that might belong to a long red/black cycle. The base matching H is drawn as dashed lines.

4. Add green and blue edges $(a, f), (b, e), (c, h)$, and (d, g), i.e.,
   $\ddot{M}^2 = M(\pi^2) \cup \{(a, f), (b, e), (c, h), (d, g)\}$ and
   $\ddot{M}^3 = M(\pi^3) \cup \{(a, f), (b, e), (c, h), (d, g)\}$.

Lemma 3.21. *$\ddot{G}$ is a valid MB graph with base matching $\ddot{H}$.*

Proof. Let V^t and V^h be two disjoint sets of nodes with $V = V^t \cup V^h$ such that every edge in $M(\pi^1) \cup M(\pi^2) \cup M(\pi^3) \cup H$ connects a node in V^t with a node in V^h. W.l.o.g., assume that $u \in V^t$. If we set $\ddot{V}^t = V^t \cup \{a, c, e, g\}$ and $\ddot{V}^h = V^h \cup \{b, d, f, h\}$, then every edge in $\ddot{M}^1 \cup \ddot{M}^2 \cup \ddot{M}^3 \cup \ddot{H}$ connects a node in $\ddot{V}^t$ with a node in $\ddot{V}^h$. If we contract the red edges (a, b), (c, d), (e, f), (g, h), we get the Hamiltonian cycle $M(\pi^1) \cup H$ on V. According to Lemma 3.6, $\ddot{M}^1 \cup \ddot{H}$ defines a Hamiltonian cycle on $\ddot{V}$. The proof that also $\ddot{M}^2 \cup \ddot{H}$ and $\ddot{M}^3 \cup \ddot{H}$ define Hamiltonian cycles on $\ddot{V}$ is analogous, but with contracting the edges (a, f), (b, e), (c, h), (d, g). Thus, all preconditions of Lemma 3.7 are fulfilled, $\ddot{G}$ is a valid MB graph, and $\ddot{H}$ is a base matching of $\ddot{G}$. □

Lemma 3.22. *There is a one-to-one correspondence between canonical matchings $M(\rho)$ of G with $\sum_{i=1}^{3} c_{odd}(M(\rho), M(\pi^i)) = k$ and canonical matchings $M(\ddot{\rho})$ of $\ddot{G}$ with $\sum_{i=1}^{3} c_{odd}(M(\ddot{\rho}), M(\ddot{\pi}^i)) = k + 8$.*

Proof. $M(\ddot{\rho})$ must contain the edges (a, f), (b, e), (c, h), and (d, g) because it is canonical. These edges define 2 red/black 2-cycles, 4 green/black 1-cycles, and 4 blue/black 1-cycles (overall 8 odd and 2 even cycles). By contracting the edges (a, b), (c, d), (e, f), and (g, h), these cycles are removed, and the resulting graph is equivalent to G, thus each canonical matching $M(\rho)$ of G with $\sum_{i=1}^{3} c_{odd}(M(\rho), M(\pi^i)) = k$ corresponds to a canonical matching $M(\ddot{\rho})$ of $\ddot{G}$ with $\sum_{i=1}^{3} c_{odd}(M(\ddot{\rho}), M(\ddot{\pi}^i)) = k + 8$. □

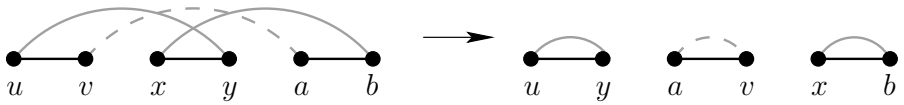


Figure 3.8: The effect of the transposition described in Lemma 3.24. The dashed line is a path of alternating gray and black edges. The operation splits the two 1-cycles with edges (u, y) and (x, b) from an l-cycle.

Lemma 3.23. *If ρ is hurdle-free w.r.t. π^2, then the corresponding permutation $\ddot{\rho}$ is also hurdle-free w.r.t. $\ddot{\pi}^2$.*

Proof. If one compares the breakpoint graph of ρ and π^2 with the one of $\ddot{\rho}$ and $\ddot{\pi}^2$, one can see that the transformation just added 4 1-cycles without changing the structure of any other cycle. Thus, for each sorting sequence that sorts ρ into π^2, there is an equivalent sorting sequence from $\ddot{\rho}$ to $\ddot{\pi}^2$. □

Of course, this lemma also holds for π^3. To make τ hurdle-free w.r.t. π^1, we repeat the transformation step for every red edge that might belong to a red/black l-cycle with $l \geq 3$. Let the resulting graph be $\hat{G} = MBG(\hat{\pi}^1, \hat{\pi}^2, \hat{\pi}^3)$. Before we can prove that every permutation that induces a canonical matching of $\hat{G}$ is hurdle-free w.r.t. π^1, we need one more lemma.

Lemma 3.24. *If a breakpoint graph contains black edges (u, v), (x, y), (a, b) (with $u < v$, $x < y$, and $a < b$) and intersecting gray edges $(u, y), (x, b)$, then a transposition acting on these black edges splits an l-cycle into an $(l - 2)$-cycle and two 1-cycles with edges (u, y) and (x, b).*

Proof. As the gray edges are intersecting, the ordering of the nodes must be $u < v < x < y < a < b$ or $a < b < u < v < x < y$ or $x < y < a < b < u < v$. In all cases, the transposition creates black edges (u, y), (x, b), and (a, v). The gray edges remain unchanged, thus the two 1-cycles with edges (u, y) and (x, b) are split from the l-cycle. For an illustration, see Fig. 3.8 □

Lemma 3.25. *Let $M(\hat{\rho})$ be a canonical matching of $\hat{G}$. Then, $\hat{\rho}$ is hurdle-free w.r.t. $\hat{\pi}^1$.*

Proof. Due to the construction rules, $\hat{\rho}$ and $\hat{\pi}^1$ have certain properties which can be best shown by their breakpoint graph. To keep the mapping from edges to permutations simple, we preserve the colors of the permutation matchings (black for $M(\rho)$ and red for $M(\pi^1)$) instead of using the colors of the original definition of the breakpoint graph (i.e., gray and black).

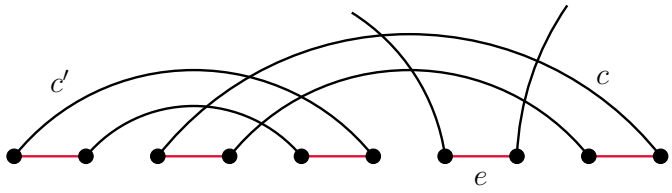


Figure 3.9: The configuration of the companion c of a red edge e. The black edges adjacent to e may also intersect. By construction, c intersects with another 2-cycle c'. However, c' is not a companion of a red edge.

For each red edge e that belongs to a long cycle, the adjacent black edges intersect with the black edges of a 2-cycle c. We call c the *companion* of e. The black edges of c neither intersect with a black edge of another long cycle, nor with a black edge of another companion. The configuration of an edge with its companion is illustrated in Fig. 3.9. We will now describe a sequence of transpositions that transforms $\hat{\pi}^1$ into $\hat{\rho}$ such that each transposition increases the number of odd cycles by 2. We start the sorting with an arbitrary red edge e of a long cycle. If the black edges adjacent to e intersect, we apply the transposition described in Lemma 3.24. This might destroy the companion of e, i.e., the intersection condition of the companion is no longer fulfilled. However, all other red edges in a long cycle still have a valid companion. Now, assume that the black edges adjacent to e do not intersect. Let f and g be the red edges connected to e by a black edge. Fig. 3.10 describes a sequence of three transpositions where each transposition increases the number of odd cycles by 2. The sequence uses the companions of f and g. Note that the sequence also works if the black edges adjacent to f or g intersect. After the sequence, all edges in a long cycle except e still have a companion. Thus, we can repeat this step (always starting with edge e) until e is in a short cycle, and then continue with another long cycle. When no long cycle remains, the resulting permutation is hurdle-free due to Lemma 3.4. □

We continue the transformation by performing equivalent steps for green and blue edges. Let $\tilde{G} = MBG(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3)$ be the resulting MB graph.

Theorem 3.26. *Let π^1, π^2, π^3 be permutations of size n, and let $G = MBG(\pi^1, \pi^2, \pi^3)$ be their MB graph satisfying Condition 3.1. Let $\tilde{G} = MBG(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3)$ be the MB graph obtained by transforming G as described above, and let m be the number of performed steps during the transformation. Then, $(\pi^1, \pi^2, \pi^3, k) \in oCMP$ if and only if $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, \frac{3n+4m-k}{2}) \in TMP$.*

Proof. As we have shown in Lemma 3.22, $(\pi^1, \pi^2, \pi^3, k) \in oCMP$ if and only if $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, k + 8m) \in oCMP$. The size of the permutations $\tilde{\pi}^1$, $\tilde{\pi}^2$, and $\tilde{\pi}^3$ is $n + 4m$.

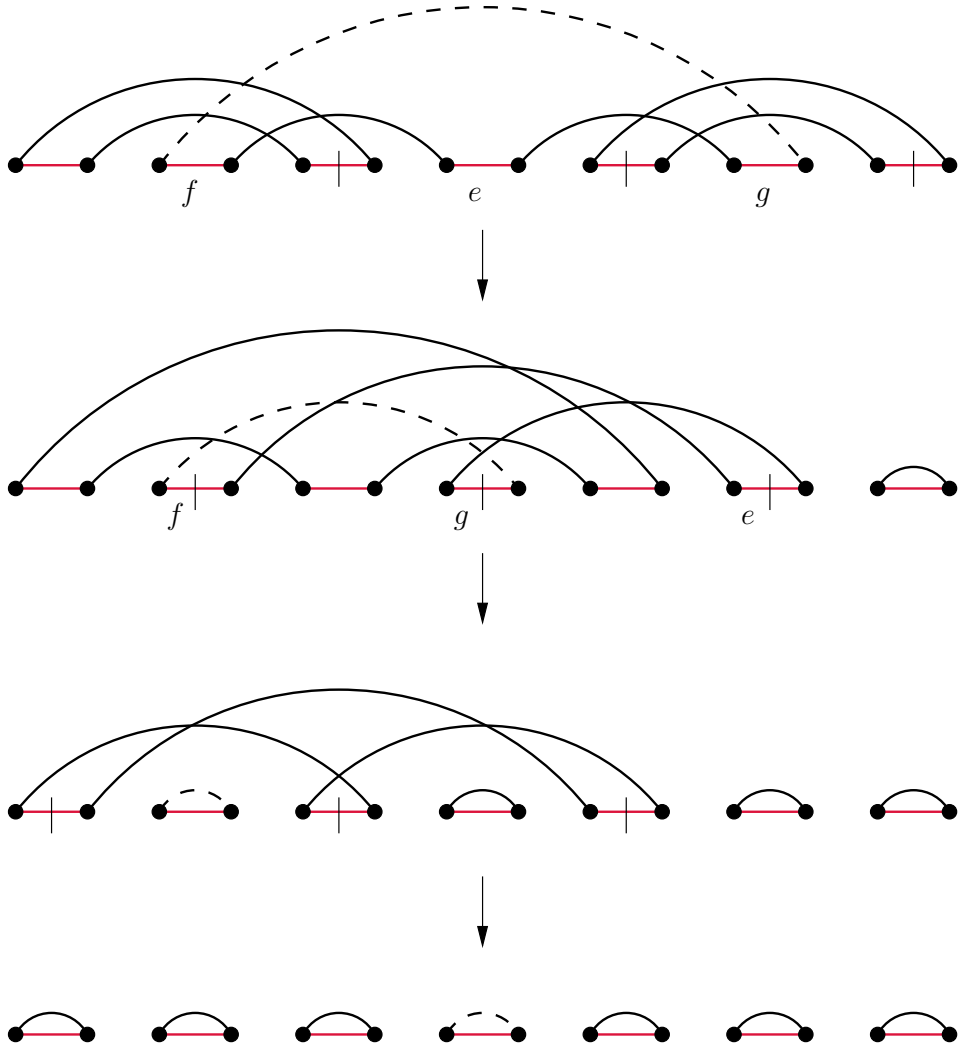


Figure 3.10: A sequence of 3 transpositions that can be applied when the black edges adjacent to e do not intersect. The 2-cycles belong to the companions of f and g. The dashed line is a path of alternating black and red edges. The vertical black lines indicate the red edges where the next transposition acts on. Note that the sequence also works if the black edges adjacent to f or g intersect.

Assume that there is a permutation matching $M(\tilde{\tau})$ of $\tilde{G}$ with $\sum_{i=1}^{3} c_{odd}(M(\tilde{\tau}), M(\tilde{\pi}^i)) \geq k+8m$. W.l.o.g. $M(\tilde{\tau})$ is a canonical matching, and due to Lemma 3.25, we can assume that $\tilde{\tau}$ is hurdle-free w.r.t. $\tilde{\pi}^1$, $\tilde{\pi}^2$, and $\tilde{\pi}^3$. Thus,

$$\begin{aligned}\sum_{i=1}^{3} d_{tp}(\tilde{\tau}, \tilde{\pi}^i) &= \sum_{i=1}^{3} \frac{n+4m-c_{odd}(\tilde{\tau}, \tilde{\pi}^i)}{2} \\ &= \frac{3n+12m-\sum_{i=1}^{3} c_{odd}(\tilde{\tau}, \tilde{\pi}^i)}{2} \\ &\leq \frac{3n+4m-k}{2}.\end{aligned}$$

On the other hand, let there be a permutation $\tilde{\tau}$ with $\sum_{i=1}^{3} d_{tp}(\tilde{\tau}, \tilde{\pi}^i) \leq \frac{3n+4m-k}{2}$. Then, we get (with Lemma 3.1)

$$\begin{aligned}\sum_{i=1}^{3} \frac{n+4m-c_{odd}(\tilde{\tau}, \tilde{\pi}^i)}{2} &= \frac{3n+12m-\sum_{i=1}^{3} c_{odd}(\tilde{\tau}, \tilde{\pi}^i)}{2} \\ &\leq \sum_{i=1}^{3} d_{tp}(\tilde{\tau}, \tilde{\pi}^i) \\ &\leq \frac{3n+4m-k}{2},\end{aligned}$$

and therefore

$$\sum_{i=1}^{3} c_{odd}(\tilde{\tau}, \tilde{\pi}^i) \geq k+8m.$$

In other words, $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, k+8m) \in oCMP$ if and only if $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, \frac{3n+4m-k}{2}) \in TMP$. □

Theorem 3.27. *TMP is NP-complete.*

Proof. Let (G, k) be an instance of an mdECD problem, let $(\pi^1, \pi^2, \pi^3, k')$ be the instance of an oCMP problem that we get after the reduction in Section 3.2.1, and let $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, \tilde{k})$ be the instance of a TMP problem that we get after the final reduction. If $(G, k) \in mdECD$, then $(\pi^1, \pi^2, \pi^3, k') \in oCMP$ and $MBG(\pi^1, \pi^2, \pi^3)$ satisfies Condition 3.1. Therefore, $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, \tilde{k}) \in TMP$.
If $(G, k) \notin mdECD$, then there is no perfect matching M on $MBG(\pi^1, \pi^2, \pi^3)$ with $\sum_{i=1}^{3} c_{odd}(M, M(\pi^i)) \geq k'$. Although $MBG(\pi^1, \pi^2, \pi^3)$ does not necessarily satisfy Condition 3.1, we can continue with the reduction as described above. From the proof of Theorem 3.26, it follows that if there is a permutation $\tilde{\tau}$ with $\sum_{i=1}^{3} d_{tp}(\tilde{\tau}, \pi^1) \leq \tilde{k}$, then $\sum_{i=1}^{3} c_{odd}(\tilde{\tau}, \tilde{\pi}^i) \geq k+8m$, and due to Lemma 3.22, there is a permutation matching $M(\tau)$ with $\sum_{i=1}^{3} c_{odd}(M(\tau), M(\pi^i)) \geq k'$, which is a contradiction. Therefore, $(\tilde{\pi}^1, \tilde{\pi}^2, \tilde{\pi}^3, \tilde{k}) \notin TMP$. □

3.3 A branch and bound algorithm

Despite the NP-hardness of the TMP, it is possible to solve many instances of the TMP and the wRTMP exactly by using a branch and bound algorithm. We will describe the algorithm for the wRTMP in Section 3.3.2, and discuss its adaption to the TMP in Section 3.3.3.
The algorithm is an extension of Caprara's median solver for the RMP [Cap03]. Its main idea is first to find a solution τ of the wCMP, and then to calculate its solution value $\gamma_{wrt}(\tau)$. For the calculation of the solution value, an algorithm for the pairwise distance d_{wrt} is required. This can either be done by the approximation algorithm devised in [BO07] (leading to a median solver with guaranteed approximation ratio of 1.5), or exactly by using another branch and bound algorithm.

3.3.1 Exact calculation of pairwise distances

Before we focus on the median solver, we first describe the exact algorithm for the pairwise distance $d_{wrt}(\pi, \rho)$. W.l.o.g., let ρ be the identity permutation, i.e., the task is to compute $d_{wrt}(\pi, id)$. The algorithm maintains a set S that contains triples $(\tilde{\pi}, d'_{wrt}(\pi, \tilde{\pi}), lb_{wrt}(\tilde{\pi}, id))$, where $\tilde{\pi}$ is a permutation, $d'_{wrt}(\pi, \tilde{\pi})$ is the weight of a sorting sequence of π w.r.t. $\tilde{\pi}$, and $lb_{wrt}(\tilde{\pi}, id)$ is the lower bound for the remaining distance towards id according to Lemma 3.1. Initially, S is set to $\{(\pi, 0, lb_{wrt}(\pi, id))\}$. In each step, one selects the triple $(\tilde{\pi}, d'_{wrt}(\pi, \tilde{\pi}), lb_{wrt}(\tilde{\pi}, id))$ from S where $d'_{wrt}(\pi, \tilde{\pi}) + lb_{wrt}(\tilde{\pi}, id)$ is minimized, and removes it from S. If $lb_{wrt}(\tilde{\pi}, id) > 0$, the triples $(op \cdot \tilde{\pi}, d'_{wrt}(\pi, \tilde{\pi}) + w(op), lb_{wrt}(op \cdot \tilde{\pi}, id))$ are added to S for each possible operation op. In other words, we add all permutations that can be reached from $\tilde{\pi}$ by a single operation, and maintain the information about the weight of the performed sequence and the lower bound towards id. We call this step *expanding* $\tilde{\pi}$. If $lb_{wrt}(\tilde{\pi}, id) = 0$, then the algorithm returns $d'_{wrt}(\pi, \tilde{\pi})$ and aborts.

Lemma 3.28. *If the algorithm selects a triple* $(\tilde{\pi}, d'_{wrt}(\pi, \tilde{\pi}), lb_{wrt}(\tilde{\pi}, id))$ *with* $lb_{wrt}(\tilde{\pi}, id) = 0$*, then* $d'_{wrt}(\pi, \tilde{\pi}) = d_{wrt}(\pi, id)$*.*

Proof. If $lb_{wrt}(\tilde{\pi}, id) = 0$, then $d_{wrt}(\tilde{\pi}, id) = 0$ due to the upper bound (see Lemma 3.2), i.e., $\tilde{\pi} = id$. The algorithm creates any possible sequence of operations, ordered by their weight. As the selected triple is the first one with $\tilde{\pi} = id$, the corresponding sequence of operations is a sorting sequence of minimal weight of π w.r.t. id, and therefore $d'_{wrt}(\pi, \tilde{\pi}) = d_{wrt}(\pi, id)$. □

If one is not only interested in the distance but also in the sorting sequence, it can easily be reconstructed by a traceback.
So far, the algorithm is just an ordinary branch and bound algorithm, and does not perform very well in practice. Thus, the algorithm is improved by a duplicate elimination. Because there are usually different optimal sequences to reach an intermediate

permutation, this permutation would be stored several times, and in the worst case the number of duplicates of a permutation can be exponential in the distance to the origin permutation. Therefore, it is first checked whether a permutation already has been reached on another sequence of operations before a new triple containing this permutation is created. Searching for a possible duplicate can be done quite efficiently by hashing techniques. The number of elements in S can be further decreased by working on the minimal permutations, which have been defined in [Chr98] as follows. Given a permutation π, the *minimal permutation* $gl(\pi)$ is obtained by "gluing" all adjacencies of π and id together, i.e., each segment of elements that is identical in π and id is replaced by a single element. For example, the permutations $\pi = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{4}\ \overrightarrow{3})$ and $\hat{\pi} = (\overrightarrow{1}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{2})$ have both the same minimal permutation $(\overrightarrow{1}\ \overrightarrow{3}\ \overrightarrow{2})$. The following lemma ensures that it is sufficient to search for an optimal sorting sequence between $gl(\pi)$ and id' to obtain an optimal sorting sequence between π and id, where id' is the identity permutation of same size as $gl(\pi)$.

Lemma 3.29. *[Chr98] Let π be a permutation and let $gl(\pi)$ be its minimal permutation. Let id be the identity permutation of same size as π, and let id' be the identity permutation of same size as $gl(\pi)$. Then, an optimal sorting sequence between $gl(\pi)$ and id' can easily be transformed into an optimal sorting sequence between π and id. Both sorting sequences have the same weight, i.e., $d_{wrt}(gl(\pi), id') = d_{wrt}(\pi, id)$.*

Note that the original lemma in [Chr98] only considered the transposition distance. However, the proof for the weighted reversal and transposition distance works analogously, thus this lemma holds for both distance measures. While Christie only used this proof to show that one never has to split adjacencies, we also use it for duplicate elimination. After performing an operation op on a permutation $\tilde{\pi}$, we store the triple $(gl(op \cdot \tilde{\pi}), d'_{wrt}(\pi, \tilde{\pi}) + w(op), lb_{wrt}(gl(op \cdot \tilde{\pi}), id'))$ instead of $(op \cdot \tilde{\pi}, d'_{wrt}(\pi, \tilde{\pi}) + w(op), lb_{wrt}(op \cdot \tilde{\pi}, id))$. Thus, two permutations that have the same minimal permutations are considered to be duplicates. The algorithm in pseudocode can be seen in Algorithm 3.1.

3.3.2 The median solver

We now describe an algorithm for the wCMP, and then extend it to the wRTMP. The formulation of the algorithm for the wCMP requires the extension of the MB graph to the *weighted MB graph*, that is a MB graph where each edge e has a weight $w(e)$, which is an odd integer (restricting the weights to be odd will simplify later proofs). Analogously, the matchings in the weighted MB graph are *weighted matchings*. The edges of two weighted matchings decompose the weighted MB graph into *paths* and *cycles*. The *length* of a path or a cycle is the sum of the weights of its edges. As a slight modification to the definition given in Section 3.1, a cycle is *even* if its length divided by 2 is an even number, otherwise it is *odd*. Note that a cycle always

Algorithm 3.1 An exact algorithm for the weighted reversal and transposition distance.

```
1: function d_wrt(π, id)
2:   π' = gl(π)
3:   S = {(π', 0, lb_wrt(π', id'))}
4:   while true do
5:     (π̃, d', lb) = S.selectMinimum()
6:     S = S \ {(π̃, d', lb)}
7:     if lb == 0 then
8:       return d'
9:     for all operations op do
10:      π' = gl(op · π̃)
11:      {insert with duplicate elimination}
12:      if S contains a triple (x, y, z) with x == π' then
13:        if y > d' + w(op) then
14:          update (x, y, z) to (x, d' + w(op), z)
15:      else
16:        S = S ∪ {(π', d' + w(op), lb(π', id'))}
```

consists of an even number of edges and each edge has an odd weight by definition, thus the length of the cycle is always an even number and dividing by 2 yields an integer value. Analogous to the previous definition, $c_{even}(M^i, M^j)$ and $c_{odd}(M^i, M^j)$ are the number of even and odd cycles defined by two matchings M^i and M^j. The *score* of two matchings M^i and M^j (which are not necessarily perfect) is defined by $\sigma(M^i, M^j) = c_{odd}(M^i, M^j) + (2 - \frac{2w_r}{w_t})c_{even}(M^i, M^j)$. For perfect matchings M^i and M^j, the *weighted cycle distance* is defined by $d_{wc}(M^i, M^j) = n - \sigma(M^i, M^j)$. Due to the weights, also the definition of M/e must be modified. Let $G = (V, E)$ be a weighted MB graph, let M be a perfect matching of V, and let $e = (u, v)$ be an edge on V. If $e \in M$, then $M/e = M \setminus \{e\}$. Otherwise, letting $(a, u), (b, v)$ be the two edges in M incident to u and v, $M/e = M \setminus \{(a, u), (b, v)\} \cup \{(a, b)\}$, and the weight of the new edge (a, b) can be calculated by $w((a, b)) = w((a, u)) + w((b, v)) + 1$. Note that $w((a, b))$ is odd, because both $w((a, u))$ and $w((b, v))$ are odd.

Lemma 3.30. *Let V be a set of nodes and let M^i, M^j, M^k, and M^l be perfect matchings of V. Then, $c_{odd}(M^i, M^k) - c_{odd}(M^i, M^l)$ is odd if and only if $c_{odd}(M^j, M^k) - c_{odd}(M^j, M^l)$ is odd.*

Proof. Let $w(M)$ denote the sum of the weights of the edges of a perfect matching M. The parity of the number of odd cycles between two perfect matchings must be the same as the overall weight of the edges divided by 2, i.e., for two perfect matchings M and M', $c_{odd}(M, M')$ is odd if and only if $\frac{w(M)+w(M')}{2}$ is odd. Therefore,

$$
\begin{aligned}
& c_{odd}(M^i, M^k) - c_{odd}(M^i, M^l) \text{ is odd} \\
\Leftrightarrow \quad & \frac{w(M^i) + w(M^k)}{2} - \frac{w(M^i) + w(M^l)}{2} \text{ is odd} \\
\Leftrightarrow \quad & \frac{w(M^k) - w(M^l)}{2} \text{ is odd} \\
\Leftrightarrow \quad & \frac{w(M^j) + w(M^k)}{2} - \frac{w(M^j) + w(M^l)}{2} \text{ is odd} \\
\Leftrightarrow \quad & c_{odd}(M^j, M^k) - c_{odd}(M^j, M^l) \text{ is odd}
\end{aligned}
$$

□

Lemma 3.31. *The* weighted cycle distance $n - \sigma(M^i, M^j)$ *on perfect matchings is a metric.*

Proof.

1. Positive definiteness: $n - \sigma(M^i, M^i) = 0$, because the graph decomposes into n odd cycles. For perfect matchings M^i, M^j with $M^i \neq M^j$, there must be at least one cycle with at least four edges, thus the overall number of cycles is less than n. As each cycle adds at most 1 to $\sigma(M^i, M^j)$, $\sigma(M^i, M^j) < n$ and $n - \sigma(M^i, M^j) > 0$.

2. Symmetry: This follows directly from the symmetry of $\sigma(M^i, M^j)$.

3. Triangle inequation: Let V be a set of nodes, and let M^i, M^j, and M^k be three perfect matchings of V. We show that $n - \sigma(M^i, M^k) + n - \sigma(M^k, M^j) \geq n - \sigma(M^i, M^j)$. For this, M^k is modified successively by the following rules.
   (a) If $(V, M^i \cup M^k)$ contains an even cycle with only two edges, change the weight of the corresponding edge in M^k such that the cycle becomes odd. This increases $\sigma(M^i, M^k)$ by $2\frac{w_r}{w_t} - 1$. In $(V, M^k \cup M^j)$, this either changes an even cycle into an odd cycle, or an odd cycle into an even cycle. Thus, $\sigma(M^i, M^k) + \sigma(M^k, M^j)$ does not decrease.
   (b) If $(V, M^i \cup M^k)$ contains a cycle with at least four edges, remove two of the edges of M^k and rejoin the endpoints such that the cycle is split into two cycles. Weight the new edges such that both cycles are odd cycles. If the original cycle was even, $\sigma(M^i, M^k)$ increases by $\frac{2w_r}{w_t}$. As the operation can effect at most two cycles in $(V, M^k \cup M^j)$, the worst possible effect on $\sigma(M^k, M^j)$ is that two odd cycles are merged into an even cycle. Thus, $\sigma(M^i, M^k) + \sigma(M^k, M^j)$ does not decrease. If the original cycle was odd, $\sigma(M^i, M^k)$ increases by 1, and the overall

number of odd cycles changes by 1. Due to Lemma 3.30, the parity of the number of odd cycles in $(V, M^k \cup M^j)$ must be changed by this modification, therefore the worst possible effect on $\sigma(M^k, M^j)$ is that two odd cycles are merged into one odd cycle. Thus, $\sigma(M^i, M^k) + \sigma(M^k, M^j)$ does not decrease.
(c) If none of the two rules above can be applied, M^i and M^k contain the same edges, but maybe with different weights. Change the weights of the edges of M^k such that they have the same weights as the edges in M^i. This step has no effect on the cycles, because all cycles in $(V, M^i \cup M^k)$ are already odd. Thus, $\sigma(M^i, M^k) + \sigma(M^k, M^j)$ remains unchanged.
The whole transformation transformed M^k into M^i without decreasing $\sigma(M^i, M^k)$ $+\sigma(M^k, M^j)$. Therefore, $n - \sigma(M^i, M^k) + n - \sigma(M^k, M^j) \geq n - \sigma(M^i, M^i) + n - \sigma(M^i, M^j) = n - \sigma(M^i, M^j)$.

□

The following lemma gives us a lower bound for the solution value of an instance of the wCMP.

Lemma 3.32. *Given four perfect matchings M^1, M^2, M^3, and M^τ, the following inequation holds.*

$$\sum_{i=1}^{3}(n - \sigma(M^\tau, M^i)) \geq \frac{3n}{2} - \sum_{i=1}^{2}\sum_{j=i+1}^{3}\frac{\sigma(M^i, M^j)}{2}$$

Proof. Using the triangle inequality given in Lemma 3.31, we get

$$\begin{aligned} & \frac{3n}{2} - \sum_{i=1}^{2}\sum_{j=i+1}^{3}\frac{\sigma(M^i, M^j)}{2} \\ = \; & \frac{1}{2}\sum_{i=1}^{2}\sum_{j=i+1}^{3}(n - \sigma(M^i, M^j)) \\ \leq \; & \sum_{i=1}^{3}(n - \sigma(M^\tau, M^i)) \end{aligned}$$

□

The following lemma gives us an even stronger lower bound if some of the edges in the solution M^ρ are already known. The idea is to contract these edges, and to calculate the lower bound for the remaining problem according to Lemma 3.32.

Lemma 3.33. *Let M^1, M^2, M^3, and M^τ be perfect matchings, let each edge in M^τ have weight 1, and let $e \in M^\tau$ be an edge. Then,*

$$\begin{aligned} & \sum_{i=1}^{3}(n - \sigma(M^\tau, M^i)) \\ = \; & 3 - \sum_{i=1}^{3}\sigma(M^i, \{e\}) + \sum_{i=1}^{3}(n - 1 - \sigma(M^\tau/e, M^i/e)) \end{aligned}$$

Proof. A cycle in $M^\tau \cup M^i$ is either absorbed by the contraction step, or it corresponds to a cycle in $M^\tau/e \cup M^i/e$ of the same length. In the first case, the absorbed cycle is equivalent to the cycle in $M^i \cup \{e\}$, and the sum of the scores of the absorbed cycles is $\sum_{i=1}^{3}\sigma(M^i, \{e\})$. As there are no new cycles in $M^\tau/e \cup M^i/e$, we get

$$\begin{aligned} & \sum_{i=1}^{3}(n - \sigma(M^\tau, M^i)) \\ = \; & -\sum_{i=1}^{3}\sigma(M^i, \{e\}) + \sum_{i=1}^{3}(n - \sigma(M^\tau/e, M^i/e)) \\ = \; & 3 - \sum_{i=1}^{3}\sigma(M^i, \{e\}) + \sum_{i=1}^{3}(n - 1 - \sigma(M^\tau/e, M^i/e)). \end{aligned}$$

□

By combining Lemmata 3.32 and 3.33, we get the following corollary.

Corollary 3.34. *Let M^1, M^2, M^3, and M^τ be perfect matchings, and let $M = \{e_1, \ldots, e_k\}$ be a subset of M^τ. Then,*

$$\sum_{i=1}^{3}(n-\sigma(M^\tau, M^i)) \geq 3|M| - \sum_{i=1}^{3}\sigma(M^i, M) + \frac{3(n-|M|)}{2} - \sum_{i=1}^{2}\sum_{j=i+1}^{3}\frac{\sigma(M^i/M, M^j/M)}{2}$$

with $M^i/M = (\ldots((M^i/e_1)/e_2)\ldots)/e_k$, and M^j/M analogously.

We are now ready to describe our branch and bound algorithm for the wCMP. A *partial solution* consists of a matching M that is not necessarily perfect, and the lower bound of M which can be calculated by the formula given in Corollary 3.34. The algorithm maintains a set S of partial solutions, consisting initially only of the empty partial solution $(\emptyset, \frac{3n}{2} - \sum_{i=1}^{2}\sum_{j=i+1}^{3}\frac{\sigma(M^i, M^j)}{2})$. In each step, the partial solution (M, lb) with the currently least lower bound is selected, removed from S, and expanded as follows. Let V' be the nodes of V such that no edge in M is incident to a node in V', and

let v_a be a fixed node in V'. Then, new partial solutions M' are created by setting $M' = M \cup (v_a, v_b)$ for all $v_b \in V', v_b \neq v_a$. Partial solutions M' that cannot be expanded to a permutation matching (i.e., $M' \cup H$ contains a cycle that is not a Hamiltonian cycle) are discarded. For all other partial solutions, the lower bounds are calculated, which can be done very efficiently by calculating the difference to the lower bound of M. In this calculation, only cycles that change due to the contraction of (v_a, v_b) have to be considered, which is much faster than to calculate the lower bound from scratch. The selection of the node $v_a \in V'$ can be done arbitrarily, and in fact Caprara uses always the node with the least index in his algorithm [Cap03]. However, a clever selection of v_a may reduce the number of partial solutions to examine, and thus also the running time and memory requirements of the algorithm. To keep the number of partial solutions small, we must increase the lower bounds of the partial solutions as fast as possible (remember that a partial solution with a lower bound greater than the true median will never be expanded). Therefore, a good strategy would be to select the node v_a such that the sum of the lower bounds of the new solutions is maximized (a similar strategy was proposed by Little et al. for the Traveling Salesman Problem [LMSK63]). Unfortunately, calculating these values in every extension step is too costly. Therefore, we calculate these values only before the first expansion step, and sort the nodes by these values. Then, in each expansion step, we select the first node in V' according to this ordering. As this ordering normally does not change very much if recalculated in each expansion step, we still get a good choice (even if not the optimal) for the node v_a in each step. In fact, experiments have show that using this strategy brings a speedup of factor $5 - 10$ against the naive approach.
The algorithm has found an optimal solution for the wCMP when the partial solution with the least lower bound is a perfect matching. It can easily be extended such that it can solve the wRTMP by adding the following step. Whenever the matching M^τ of the best partial solution is a perfect matching, create the corresponding permutation π^τ and test if $\sum_{i=1}^{3} d_{wrt}(\pi^\tau, \pi^i)$ is equal to the lower bound. In this case, an optimal solution is found. Otherwise, the lower bound for M^τ is increased, and the partial solution is reinserted into S. A further speed-up of the pairwise distance algorithm can be obtained by providing an upper bound (remember that we only want to test if the sum of the pairwise distances is equal to the lower bound, thus the pairwise distance algorithms can be aborted if the currently best results are above this bound). The algorithm in pseudocode can be seen in Algorithm 3.2.

3.3.3 Adaption to the TMP

The algorithm can easily be adapted to the TMP. If we set $w_r = w_t$, we get $n - \sigma(M^i, M^j) = n - c_{odd}(M^i, M^j)$, i.e., the weighted cycle distance matches the lower bound of the transposition distance. Therefore, all lemmata still hold. The only modification that has to be done is due to the fact that all elements must have a positive

Algorithm 3.2 An exact algorithm for the weighted reversal and transposition median.

1: **function** $median(\pi^1, \pi^2, \pi^3)$
2: $M = \emptyset$
3: $S = \{(M, lower\_bound(M(\pi^1), M(\pi^2), M(\pi^3), M))\}$
4: {calculate node order}
5: **for all** nodes v_i **do**
6: $score[i] = 0$
7: **for all** nodes $v_j \neq v_i$ **do**
8: **if** $M' \cup H$ contains no cycle **then**
9: $score[i] \mathrel{+}= lower\_bound(M(\pi^1), M(\pi^2), M(\pi^3), \{(v_i, v_j)\})$
10: order nodes by *score*
11: {perform branch and bound}
12: **while true do**
13: $(M, lb) = S.selectMinimum()$
14: $S = S \setminus \{(M, lb)\}$
15: **if** $M == M(\tau)$ for a permutation τ **then**
16: **if** $\sum_{i=1}^{3} d_{wrt}(\tau, \pi^i) == lb$ **then**
17: **return** τ
18: **else**
19: $S = S \cup (M, lb + 1)$
20: $a = argmax\{score[x] \mid \nexists y \text{ with } (v_x, v_y) \in M\}$ {select v_a}
21: **for all** $b \neq a$ with $\nexists y$ with $(v_b, v_y) \in M$ **do**
22: $M' = M \cup \{(v_a, v_b)\}$ {expand}
23: **if** $M' \cup H$ contains no non-Hamiltonian cycle **then**
24: $S = S \cup \{(M', lower\_bound(M(\pi^1), M(\pi^2), M(\pi^3), M'))\}$

orientation. Thus, when expanding a partial solution, we only consider edges of the form (x_t, y_h), because all other edges correspond to a change of the orientation of two adjacent elements in the corresponding permutation.

3.3.4 Experimental results

The algorithm was tested on artificial data. Tests were performed with different datasets to assess the performance under use of the transposition distance, and the weighted reversal and transposition distance with weight ratios $1:1$, $1:1.5$, and $1:2$ ($w_r : w_t$). Finally, we adjusted the algorithm such that it can solve the RMP, and created a corresponding dataset.
Each test case was generated as follows. First, we set $\pi^1 = \pi^2 = \pi^3 = id$, where id is the identity genome $(\overrightarrow{1} \ldots \overrightarrow{n})$, with $n \in \{37, 100\}$. This reflects the sizes of mitochondrial and chloroplast genomes. Then, we created l operations, where l was

varied from 10 to 100 in steps of 10. For each operation, it was randomly decided from an independent uniform distribution whether it should be applied to π^1, π^2, of π^3. In other words, the input genomes π^1, π^2, and π^3 were created out of id by three sequences of operations, the overall number of applied operations was l. For each operation, we first randomly decided the type of the operation, where reversals had the probability p_{rev}, transpositions had the probability p_{tp}, and inverted transpositions had the probability p_{itp}. Once the type of the operation was determined, the operation was drawn from a uniform distribution of all operations of this type. The probabilities p_{rev}, p_{tp}, and p_{itp} were chosen such that the expected frequency of the different operations reflect the parameters w_r and w_t when applying our median solver, i.e., it must hold that $p_{rev} = \frac{1}{w_r}/(\frac{1}{w_r} + \frac{1}{w_t}) = \frac{w_t}{w_r+w_t}$ and $p_{tp} + p_{itp} = \frac{1}{w_t}/(\frac{1}{w_r} + \frac{1}{w_t}) = \frac{w_r}{w_r+w_t}$. Thus, different datasets were created with $p_{tp} = 1, p_{rev} = p_{itp} = 0$ (corresponding to the transposition distance), $p_{rev} = 0.5, p_{tp} = p_{itp} = 0.25$ (corresponding to the weight ratio $w_r : w_t = 1 : 1$), $p_{rev} = 0.6, p_{tp} = p_{itp} = 0.2$ (corresponding to the weight ratio $w_r : w_t = 1 : 1.5$), $p_{rev} = \frac{2}{3}, p_{tp} = p_{itp} = \frac{1}{6}$ (corresponding to the weight ratio $w_r : w_t = 1 : 2$), and $p_{rev} = 1, p_{tp} = p_{itp} = 0$ (corresponding to the reversal distance). For all combinations of the parameters n, l, and the probabilities p_{rev}, p_{tp}, and p_{itp}, ten different test cases were created. All tests were performed on a standard PC (Intel 3.16 GHz Core2 Duo CPU with 4 GB RAM), the running time for each test case was limited to one hour.

An overview of the results is given in Tables 3.1 to 3.10 at the end of the chapter. In the tables, we list the number of solved test cases (out of 10) and average running time of the approximation algorithm (i.e., we used the approximation algorithms devised in [HS06, BO07] for recalculating the weights of the exact weighted cycle medians) and the exact algorithm for each combination of parameters, as well as the average gap and the maximum gap between the solution of the approximation algorithm and the exact algorithm. Of course, the gaps can only be computed for test cases which have been solved by both algorithms. Test cases where the time limit of 1 hour was exceeded were omitted when calculating the gaps, but taken into account and set to 1 hour when calculating the average running times. In some cases, the heap had to be pruned due to the memory limit of 4 GB. Although we only prune the currently worst solutions, there is the possibility that we miss the optimal solution. The column "heap pruned" indicates on how many test cases this heap pruning might have led to the loss of the optimal solution.
The algorithm shows slightly different behavior for $n = 37$ and $n = 100$. If n is set to 37, the running time slowly increases with increasing values of l. All instances could be solved within 1 hour, except for a few test cases where we used the transposition distance and the exact median solver. The weight ratio has a strong influence on the complexity of the problem, the running time decreases with increasing importance of reversals. That is, solving instances of the TMP takes the most time, followed by

instances of the wRTMP with weight ratios 1 : 1, 1 : 1.5, and 1 : 2. Solving instances of the RMP is even faster, all test cases could be solved within a few seconds. The same holds for the memory consumption. A comparison between the approximation algorithm and the exact algorithm shows that the approximation algorithm is significantly faster, and has a very good accuracy.
If n is set to 100, the algorithm behaves similarly. However, instead of the slow increment of the running time, there is a critical distance where running time and memory consumption drastically increase. Again, this value depends on the weight ratio, and varies from $l \approx 50$ for the TMP to $l \approx 90$ for the wRTMP at a weight ratio of 1 : 2. When solving instances of the RMP, this critical value was not reached in our experiments, i.e., all test cases could be solved within a few seconds.
A comparison with the software tool `GRAPPA-TP` [YZT07] on the instances of the TMP shows that this program reaches its limit much faster than our program. `GRAPPA-TP` needed in average more than 10 minutes for test cases with $l = 20$, and could not solve any test case with $l \geq 40$. Note that these problems could be solved in less than 1 minute by our approximation algorithm, which also had a higher accuracy than `GRAPPA-TP`.

3.4 Conclusion and open problems

We have proven the NP-completeness of the TMP, and developed an algorithm for the TMP and the wRTMP which is fast enough to be used in practice. Our implementation outperforms existing median solvers for the TMP, and, to the best of our knowledge, is the first program which can solve instances of the wRTMP exactly.

The proof of the NP-completeness of the TMP directly implies that also the wRTMP is NP-complete if both operations are weighted equally, i.e., the proof still holds if we replace the transposition distance by the weighted reversal and transposition distance in each step. For a weight ratio of 1 : 2 ($w_r : w_t$), the NP-completeness follows from [Cap03]. For all weight ratios in between, the complexity is still open.
A further open problem is whether the TMP is APX-hard or not. As the reversal median problem is APX-hard [Cap03], the former is more likely. Indeed, a careful examination of our proof shows that it also proves the APX-hardness of the TMP if there is an APX-hardness proof of the mdECD that holds even if the mdECD instance satisfies the following two conditions.

1. The degree of each node in the graph is bounded.
2. There is a constant $\alpha > 0$ with $k \geq \alpha \cdot |V|$

The first condition is required to keep the second condition valid when bounding the in- and out-degree to 2. The second condition is necessary to preserve the APX-hardness in

the reduction from oCMP to TMP. Unfortunately, Holyer's NP-hardness proof for ECD cannot be transformed into an APX-hardness proof by using an APX-hard variant of SAT, like MAX-2-SAT-3 [ACG$^+$99, BK99] (which would satisfy both conditions), as suboptimal solutions of the ECD can correspond to inconsistent variable assignments in the SAT formula. Also the APX-hardness proof for directed ECD in [SV05] does not help, as the size of the cycles grows with the size of the input, and therefore the second condition cannot be satisfied.

Another closely related problem is the *transposition median problem on the symmetric group* $\mathfrak{S}_n$ (short TM$\mathfrak{S}$), which has been extensively studied by Eriksen [Eri07, Eri09] (note that in this problem, transpositions are defined differently than in our problem). Although this problem is closely related to the DCJ median problem, its NP-hardness could not be proven so far, mainly because one has to deal with directed graphs [Eri09]. As our proof extends some steps of Caprara's proof to directed graphs, we hope that it can also give new insights into TM$\mathfrak{S}$.

Although the algorithm devised in Section 3.3 allows to solve small instances of TMP and wRTMP, there is still room for improvement. One possible speed improvement has been demonstrated by Rajan et al. [RXL$^+$10] for the RMP. They used a fast solver for the DCJ median problem which splits the MB graph into *adequate subgraphs* (for details, see [Xu08, Xu09b]). The solution value of the resulting median is reevaluated using the reversal distance. While this technique does not return all DCJ medians and may miss the true reversal median, it has proven itself as a good heuristic for the RMP. It remains an open question how good this heuristic is if the algorithm is adapted to the TMP or the wRTMP.

l	solved exactly	average time	heap pruned
10	10	0:00	0
20	10	0:00	0
30	10	0:00	0
40	10	0:00	0
50	10	0:00	0
60	10	0:01	1
70	10	0:02	0
80	10	0:03	2
90	10	0:05	5
100	10	0:07	4

l	solved exactly	average time	heap pruned
10	10	0:00	0
20	10	0:00	0
30	10	0:00	0
40	10	0:00	0
50	10	0:00	0
60	10	0:00	0
70	10	0:00	0
80	10	0:00	0
90	10	0:04	1
100	10	0:03	2

Table 3.1: reversal distance, $n = 37$ (left) and $n = 100$ (right).

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0	0	0:00	10	0:06	0
30	10	0	0	0:04	9	6:19	2
40	10	0.1	1	0:37	10	3:34	8
50	10	0.25	1	0:47	8	17:35	8
60	10	0.2	1	1:33	5	36:19	10
70	10	0.25	1	1:39	8	13:02	8
80	10	0	0	1:37	8	28:38	10
90	10	0.14	1	1:52	7	23:56	10
100	10	0	0	1:51	5	37:22	10

Table 3.2: $n = 37$, transposition distance

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0	0	0:00	10	0:00	0
30	10	0	0	0:00	10	0:00	0
40	10	0	0	0:22	8	12:29	6
50	7	0	0	23:16	2	48:17	9
60	4	0	0	48:43	1	56:07	9

Table 3.3: $n = 100$, transposition distance. No instance with $l \geq 70$ could be solved.

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0	0	0:01	10	0:00	0
30	10	0.2	1	0:15	10	0:04	1
40	10	0.2	1	0:59	10	1:27	8
50	10	0.3	2	2:43	10	3:58	9
60	10	0.3	1	4:56	10	8:17	10
70	10	0.3	1	7:44	10	12:29	10
80	10	0.1	1	6:29	10	12:53	10
90	10	0.4	1	9:23	10	16:00	10
100	10	0.4	1	9:42	10	15:43	10

Table 3.4: $n = 37$, $w_r = 1$, $w_t = 1$

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0	0	0:00	10	0:00	0
30	10	0	0	0:17	10	0:25	2
40	10	0	0	1:36	10	2:21	8
50	9	0.56	2	10:21	9	10:42	6
60	6	0.5	2	38:56	4	49:25	10

Table 3.5: $n = 100$, $w_r = 1$, $w_t = 1$. No instance with $l \geq 70$ could be solved.

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0.05	0.5	0:00	10	0:00	0
20	10	0	0	0:00	10	0:00	0
30	10	0.1	1	0:03	10	0:01	0
40	10	0	0	0:40	10	1:02	9
50	10	0.25	1	2:43	10	3:34	10
60	10	0.5	1.5	3:09	10	3:59	10
70	10	0.15	1	4:09	10	6:06	10
80	10	0.2	1	4:59	10	7:32	10
90	10	0.15	0.5	5:04	10	7:09	10
100	10	0	0	5:45	10	8:14	10

Table 3.6: $n = 37$, $w_r = 1$, $w_t = 1.5$

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0.05	0.5	0:00	10	0:00	0
30	10	0.05	0.5	0:00	10	0:00	0
40	10	0.1	0.5	0:02	10	0:02	1
50	10	0.15	1.5	2:22	10	3:28	8
60	10	0.3	1	12:08	10	17:01	10
70	10	0.35	2	31:31	8	41:12	10
80	7	0	0	51:11	2	58:27	10
90	1			59:52	0		
100	0				0		

Table 3.7: $n = 100$, $w_r = 1$, $w_t = 1.5$

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0	0	0:00	10	0:00	0
30	10	0	0	0:00	10	0:00	0
40	10	0	0	0:00	10	0:00	0
50	10	0	0	0:02	10	0:03	2
60	10	0	0	0:05	10	0:07	4
70	10	0	0	0:04	10	0:05	2
80	10	0	0	0:08	10	0:10	6
90	10	0	0	0:05	10	0:05	4
100	10	0	0	0:08	10	0:12	4

Table 3.8: $n = 37$, $w_r = 1$, $w_t = 2$

l	solved approx.	average gap	max. gap	average time	solved exactly	average time	heap pruned
10	10	0	0	0:00	10	0:00	0
20	10	0	0	0:00	10	0:00	0
30	10	0	0	0:00	10	0:00	0
40	10	0	0	0:00	10	0:00	0
50	10	0	0	0:00	10	0:00	0
60	10	0	0	0:00	10	0:00	0
70	10	0	0	0:26	10	0:25	3
80	10	0.2	1	6:00	10	7:21	6
90	10	0	0	16:32	8	24:57	10
100	3	0	0	52:13	1	55:34	10

Table 3.9: $n = 100$, $w_r = 1$, $w_t = 2$

l	solved	average gap	max. gap	average time
10	10	0	0	0:00
20	8	0.38	2	13:44
30	1	1	1	56:40

l	solved	average gap	max. gap	average time
10	10	0.3	2	0:02
20	10	0	0	12:39
30	5	0	0	44:11

Table 3.10: GRAPPA-TP, $n = 37$ (left) and $n = 100$ (right). No instance with $l \geq 40$ could be solved.

4 Phylogenetic Reconstruction

In this chapter, we present a heuristic algorithm that directly constructs a phylogenetic tree w.r.t. the weighted reversal and transposition distance. Experimental results on previously published datasets show that constructing phylogenetic trees in this way results in better trees than constructing the trees w.r.t. the reversal distance, and recalculating the weight with the weighted reversal and transposition distance.
The chapter is organized as follows. In Section 4.1, some fundamental definitions are given. In Section 4.2, the algorithm is described. The experimental results and a comparison to other available algorithms is given in Section 4.3. In Section 4.4, the results of the algorithm are summarized and other possibilities to create phylogenetic trees w.r.t. the weighted reversal and transposition distance are discussed.

4.1 Fundamental definitions

A *phylogenetic tree* of a set of genomes $P = \{\pi^1, \ldots, \pi^k\}$ (the *input genomes*) is a tree $T = (V, E)$, where V is the set of nodes and E is the set of edges of the tree. Each node is labelled by a genome π^i, and there is a bijection between the labels of leaves and the input genomes (i.e., any element of P is the label of exactly one leaf). The weight of an edge (π^i, π^j) is the distance $d(\pi^i, \pi^j)$. The weight of a tree $w(T)$ is the sum of the weights of its edges. Note that these definitions hold for any distance measure. In fact, our algorithm can be used for any distance measure as long as certain conditions are satisfied. If we want to emphasize that a certain distance measure is used, we write its indicator as subscript, e.g., $w_{rev}(T)$ is the weight of the tree under the reversal distance, and $w_{wrt}(T)$ is its weight under the weighted reversal and transposition distance. In the *multiple genome rearrangement problem*, given are a set of genomes P and a distance measure d, and the task is to find a phylogenetic tree of P with minimal weight under d.

4.2 The algorithm

The algorithm consists of two different phases. In the first phase, a fast heuristic is used to create a phylogenetic tree. In contrast to previous algorithms, this heuristic does not rely on a median solver. In the second phase, the tree is improved until it converges to a local optimum. Two different improvement algorithms are used: one improves the tree topology, while the other improves the labeling of internal nodes by using a median solver. These two algorithms can be run alternatingly until the tree does not improve any further. In practice, the topology of the tree created in the first phase is already very good, so the algorithm for improving the topology has to be run only once. The whole algorithm was designed for the weighted reversal and transposition distance, however, it can also be used for other rearrangement distances as it only requires an algorithm that finds an optimal sorting sequence between two permutations, or at least a good approximation algorithm. Only the second improvement algorithm requires a median solver.

4.2.1 Creating the tree

The tree T is created iteratively, beginning with a tree T_1 whose node set consists only of one arbitrary genome of the set of input genomes P, i.e., $T_1 = (V_1, E_1) = (\{\pi\}, \emptyset)$ with $\pi \in P$. In each step, we create the tree T_i from the tree T_{i-1} by choosing a genome $\pi^p \in P \setminus V_{i-1}$ that is not yet a node in the tree, and add this genome as leaf node. This includes an update of the set of edges, and can include the creation of a new internal node. The algorithm terminates when all input genomes are in the tree, i.e., $T = T_k$. Choosing the next genome $\pi^p \in P$ to be added to the tree T_{i-1}, as well as determining its ancestor node, is done by a heuristic that minimizes the weight of the resulting tree T_i. In contrast to previous algorithms, we do not use a median solver for this. Instead, we maintain for each edge $(\pi^i, \pi^j) \in E_{i-1}$ a set of genomes, called a *cloud* of the edge. These clouds can be seen as sets of candidate nodes for internal nodes. For a formal definition of a cloud, we first have to define the *δ-vicinity* of an edge. Let (π^i, π^j) be an edge in a phylogenetic tree, and let $\delta \in \mathbb{R}$. The δ-vicinity of (π^i, π^j) is defined by

$$vic_\delta(\pi^i, \pi^j) = \{\pi^c \in \Sigma_n^* \mid d(\pi^i, \pi^c) + d(\pi^c, \pi^j) \leq d(\pi^i, \pi^j) + \delta\}$$

Intuitively, this means that the δ-vicinity of an edge (π^i, π^j) contains all genomes that lie "in between" π^i and π^j. If $\pi^c \in vic_\delta(\pi^i, \pi^j)$, then splitting the edge into the two edges (π^i, π^c) and (π^c, π^j), and adding the edge (π^c, π^p) will increase the weight of the tree by at most $d(\pi^c, \pi^p) + \delta$. Thus, searching for new internal nodes in the vicinity of edges seems to be a good heuristic. However, even the 0-vicinity of an edge (π^i, π^j) can be of exponential size w.r.t. $d(\pi^i, \pi^j)$. Hence, it is not practicable to search the whole vicinities of the edges, and we have to restrict the search space somehow. In the following, we assume that δ is a small, fixed number. Then, any subset of $vic_\delta(\pi^i, \pi^j)$

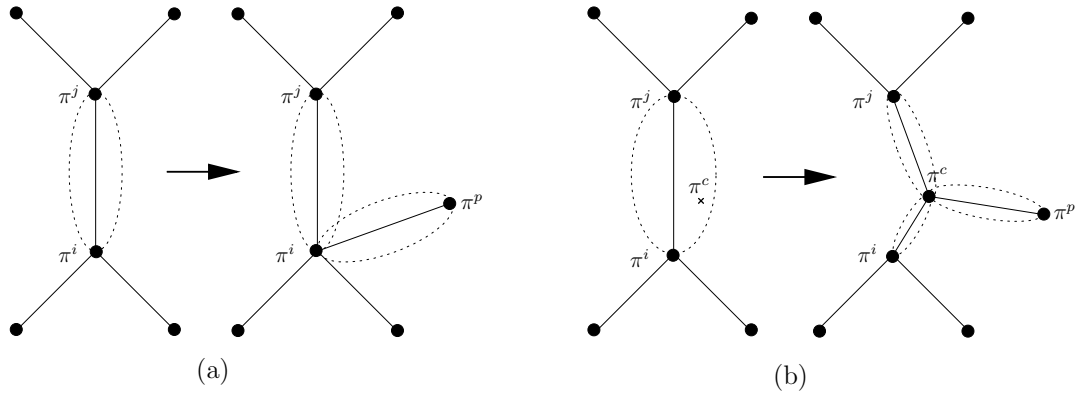


Figure 4.1: A new node π^p can either be added (a) to a node π^i in the tree or (b) to a node π^c in a cloud of an edge (π^i, π^j). In the latter case, we have to split the edge (π^i, π^j) into two edges (π^i, π^c) and (π^c, π^j). Clouds are removed/generated accordingly.

is called a *cloud* of the edge (π^i, π^j). Of course, this definition is quite general and does not reflect which cloud should be chosen. A heuristic for generating clouds is presented in the next section. In the following, $cloud(\pi^i, \pi^j)$ denotes the cloud which has been assigned to the edge (π^i, π^j).
Creating the tree T_i from T_{i-1} is done as follows. We choose an element $\pi^p \in P \setminus V_{i-1}$ and either (a) a node $\pi^i \in V_{i-1}$ or (b) an edge $(\pi^i, \pi^j) \in E_{i-1}$ and a genome $\pi^c \in cloud(\pi^i, \pi^j)$, such that the resulting tree T_i is of minimum weight. If (a) a node $\pi^i \in V_{i-1}$ is chosen, then the resulting tree is obtained by adding an edge from π^i to π^p, i.e., $V_i = V_{i-1} \cup \{\pi^p\}$ and $E_i = E_{i-1} \cup \{(\pi^i, \pi^p)\}$. If (b) an edge (π^i, π^j) and a genome $\pi^c \in cloud(\pi^i, \pi^j)$ is chosen, the resulting tree is obtained by replacing (π^i, π^j) with the two edges (π^i, π^c) and (π^c, π^j) and adding a new edge (π^c, π^p), i.e., $V_i = V_{i-1} \cup \{\pi^c, \pi^p\}$ and $E_i = E_{i-1} \cup \{(\pi^i, \pi^c), (\pi^c, \pi^j), (\pi^c, \pi^p)\} \setminus \{(\pi^i, \pi^j)\}$. An illustration can be found in Fig. 4.1. The weight of T_i can be calculated in Case (a) by $w(T_i) = w(T_{i-1}) + d(\pi^i, \pi^p)$ and in Case (b) by $w(T_i) = w(T_{i-1}) - d(\pi^i, \pi^j) + d(\pi^i, \pi^c) + d(\pi^c, \pi^j) + d(\pi^c, \pi^p)$. It should be pointed out that whenever we add an edge to the tree we also generate its cloud. Analogously, whenever we remove an edge from the tree we also delete its cloud. The resulting tree does not necessarily fulfill the definition of a phylogenetic tree, as an input genome may correspond to an internal node π instead of a leaf node. This can easily be fixed by creating an exact copy π' of π and adding the edge (π, π'), i.e., the input genome corresponds now to the leaf π'.
The algorithm in pseudocode for creating the tree can be seen in Algorithm 4.1.

Algorithm 4.1 Creating a phylogenetic tree

```
1: function createTree(P)
2:   select π ∈ P arbitrarily
3:   V_1 = {π}, E_1 = ∅
4:   for i = 2 to k do
5:     {find best tree update}
6:     bestWeight = ∞
7:     for all π^p ∈ P \ V_{i-1} do
8:       for all π^i ∈ V_{i-1} do
9:         if d(π^i, π^p) < bestWeight then
10:          bestWeight = d(π^i, π^p)
11:          bestInnerNode = π^i
12:          bestLeaf = π^p
13:          connectToCloud = false
14:      for all (π^i, π^j) ∈ E_{i-1} do
15:        for all π^c ∈ cloud(π^i, π^j) do
16:          if d(π^i, π^c) + d(π^c, π^j) + d(π^c, π^p) − d(π^i, π^j) < bestWeight then
17:            bestWeight = d(π^i, π^c) + d(π^c, π^j) + d(π^c, π^p) − d(π^i, π^j)
18:            bestEdge = (π^i, π^j)
19:            bestCloudNode = π^c
20:            bestLeaf = π^p
21:            connectToCloud = true
22:    {perform tree update}
23:    if connectToCloud then
24:      (π^i, π^j) = bestEdge
25:      V_i = V_{i-1} ∪ {bestCloudNode, bestLeaf}
26:      E_i = E_i ∪ {(π^i, bestCloudNode), (bestCloudNode, π^j),
27:           (bestCloudNode, bestLeaf)} \ {(π^i, π^j)}
28:    else
29:      V_i = V_{i-1} ∪ {bestLeaf}
30:      E_i = E_{i-1} ∪ {(bestInnerNode, bestLeaf)}
31:  return (V_k, E_k)
```

4.2.2 Creating the clouds

Quality and size of the clouds are crucial for the quality of the resulting tree and the running time of the algorithm. Let us consider the two extremes. If the clouds are empty, the algorithm is reduced to the Prim-Jarnik algorithm that finds a minimum spanning tree of P [Jar30, Pri57]. If the cloud of an edge (π^i, π^j) contains the whole vicinity of the edge, the size of the cloud is exponential w.r.t. $d(\pi^i, \pi^j)$. Our goal is to

Algorithm 4.2 Creating a cloud of the edge (π^i, π^j)

```
1: function createCloud(π^i, π^j)
2:    C_0 = {π^i}
3:    k = 1
4:    while π^j ∉ C_{k-1} do
5:       {if distances cannot be calculated exactly, use the lower bound instead}
6:       C_k = {op · π^c | π^c ∈ C_{k-1} and d(op · π^c, π^j) < d(π^c, π^j)}
7:       Reduce the size of C_k to a constant s
8:       k = k + 1
9:    return ⋃_{l=1}^{k-1} C_l
```

find, for each edge (π^i, π^j), a cloud of polynomial size w.r.t. $d(\pi^i, \pi^j)$ that provides a good coverage of $vic_\delta(\pi^i, \pi^j)$. The main idea of our heuristic is to generate different optimal or near optimal sorting sequences of π^i w.r.t. π^j and to select a subset of the genomes that lie on these sorting sequences as the cloud of the edge. To avoid duplicates in the cloud, our algorithm proceeds as follows. First, we define the set $C_0 = \{\pi^i\}$. Then, we iteratively generate the sets C_k out of the sets C_{k-1} by applying to each genome in C_{k-1} each operation that decreases the distance to π^j. For example, the first step yields the set $C_1 = \{op \cdot \pi^i \mid d(op \cdot \pi^i, \pi^j) < d(\pi^i, \pi^j)\}$. Then, we reduce the size of C_k by selecting a fixed number of disjoint genomes from C_k. As additional heuristic, we select the genomes from C_k that minimize the distance to the closest genome in the set of input genomes P that is not yet in the tree. These steps are repeated until we reach the genome π^j, i.e., $C_m = \{\pi^j\}$. As cloud, we use the union of the sets C_1 to C_{m-1}. The algorithm in pseudocode can be seen in Algorithm 4.2.

Note that the resulting cloud is a subset of $vic_0(\pi^i, \pi^j)$. However, this technique works only if we have a fast exact algorithm for the pairwise distance, thus it cannot be used for the weighted reversal and transposition distance directly. In this case, we therefore apply operations that decrease the lower bound instead of decreasing the real distance. Thus, the parameter δ depends on the approximation quality of the algorithm for generating the sorting sequence, and we cannot determine it exactly. However, the approximation quality of the algorithm is very good in practice [Bad05], so we can assume that δ is small.

4.2.3 Improving the topology

The construction phase may get trapped in a local minimum. To avoid this, it is followed by an improvement phase, which iteratively tries to find edges that are better than the existing ones. The input to the improvement algorithm is a tree $T' = (V', E')$ for P in conjunction with the clouds of the edges. The algorithm works as follows. First, we temporarily remove an edge $e \in E'$. This splits T' into two subtrees $T_1 = (V_1, E_1)$

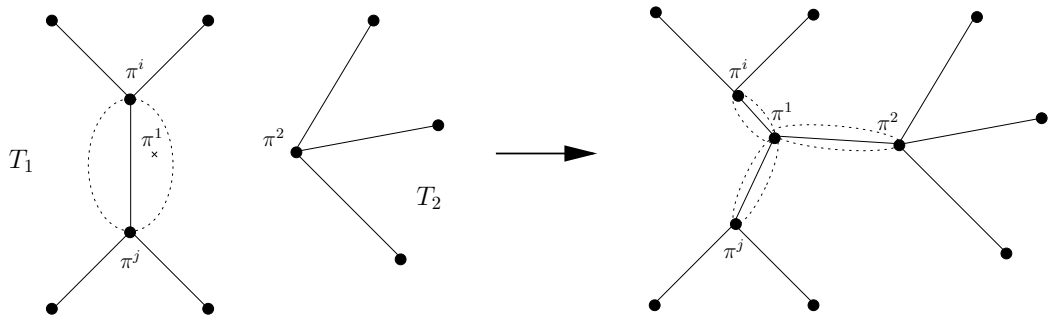


Figure 4.2: The subtrees T_1 and T_2 are reconnected by an edge (π^1, π^2), where $\pi^1 \in cloud(\pi^i, \pi^j)$ and $\pi^2 \in V_2$. The edge (π^i, π^j) must be split into two edges (π^i, π^1) and (π^1, π^j) before the new edge (π^1, π^2) is added. Clouds are removed/generated accordingly.

and $T_2 = (V_2, E_2)$. Then we search for a better edge (π^1, π^2) that reconnects these two subtrees as follows. π^1 is either a node in V_1 or a genome in the cloud of an edge $(\pi^i, \pi^j) \in E_1$. In the latter case, we alter the tree T_1 into a tree $\tilde{T_1} = (\tilde{V_1}, \tilde{E_1})$ by replacing the edge (π^i, π^j) with the two edges (π^i, π^1) and (π^1, π^j), i.e., $\tilde{V_1} = V_1 \cup \{\pi^1\}$ and $\tilde{E_1} = E_1 \cup \{(\pi^i, \pi^1), (\pi^1, \pi^j)\} \setminus \{(\pi^i, \pi^j)\}$. Otherwise, we set $\tilde{T_1} = T_1$. The tree $\tilde{T_2}$ is defined analogously to $\tilde{T_1}$ by distinguishing as to whether π^2 is a node in V_2 or a permutation in a cloud of an edge in E_2. The new tree $\overline{T} = (\overline{V}, \overline{E})$ is obtained by connecting $\tilde{T_1}$ and $\tilde{T_2}$ by the edge (π^1, π^2), i.e., $\overline{V} = \tilde{V_1} \cup \tilde{V_2}$ and $\overline{E} = \tilde{E_1} \cup \tilde{E_2} \cup \{(\pi^1, \pi^2)\}$. For an example, see Fig. 4.2. If $w(\overline{T}) < w(T')$, these changes are accepted (i.e., $T' := \overline{T}$), otherwise they are discarded. When the changes are accepted, we generate the clouds for the new edges and delete the clouds of removed edges. Note that searching for the edge (π^1, π^j) is done by an exhaustive search for π^1 in all nodes in V_1 and all genomes in the clouds of all edges in E_1, and similarly for π^2. The improvement step is repeated until no further improvement is found.

4.2.4 Improving internal nodes

Due to the tree construction algorithm, usually some of the internal nodes are not the median of their neighboring nodes, although they are very close to the median in most cases. The second improvement algorithm improves the tree by relabeling internal nodes until all internal nodes are the median of their neighbors. This algorithm has already been described in [SCL76], but for the sake of completeness, we give another description here.

In order to use a median solver to perform the improvement steps, we first have to ensure that each internal node is of degree 3, since the median solver is designed to

find the median of three nodes. Thus, for each internal node π with a degree of $k > 3$, we create a node π' which is an exact copy of π. We reconnect these two nodes such that π is connected to π' and two of its former neighbors, while π' is connected to π and the rest of the former neighbors of π. Thus π has now a degree of 3, and π' has a degree of $k - 1$. We repeat this step until all internal nodes have a degree of 3.
Now, for each internal node π, we calculate a median τ of its three neighboring nodes. If the sum of the distances of the neighboring nodes to τ is less than the sum of the distances of the neighboring nodes to π (i.e., π is not a median of its neighboring nodes), we replace π by τ. This improvement step is repeated until the tree does not improve any further, i.e., each node is the median of its neighboring nodes.

4.2.5 Implementation details

We implemented the algorithm for the reversal distance as well as for the weighted reversal and transposition distance. In the second improvement phase, we use our own reimplementation of Caprara's median solver [Cap03] for the reversal distance, which has been improved by the same selection strategy as described in Section 3.3.2. For the weighted reversal and transposition median, we use the algorithm devised in Section 3.3, the pairwise distances are calculated by the approximation algorithm devised in [BO07].
While the choice of the start node of the algorithm is arbitrary, the results may vary depending on the chosen start node. Therefore, we create a phylogenetic tree for every possible choice of a start node, and report the tree with the least weight.
Following exactly the description in the previous sections when implementing the algorithm results in a rather slow algorithm. Thus, in our implementation, we use the following two modifications which results in a severe speed improvement.

1. In the first phase, the exhaustive search can be accelerated by using a bounding technique. Let (π^i, π^j) be an edge in E_{i-1}, let π^c be a genome in $cloud(\pi^i, \pi^j)$, and let π^p be a genome in $P \setminus V_{i-1}$. From the triangle inequation, it follows that $d(\pi^i, \pi^p) + d(\pi^j, \pi^p) \leq d(\pi^i, \pi^j) + 2d(\pi^c, \pi^p)$, and therefore $d(\pi^c, \pi^p) \geq \frac{d(\pi^i,\pi^p)+d(\pi^j,\pi^p)-d(\pi^i,\pi^j)}{2}$. Thus, connecting π^p with a genome in $cloud(\pi^i, \pi^j)$ will increase the weight of the tree by at least $\frac{d(\pi^i,\pi^p)+d(\pi^j,\pi^p)-d(\pi^i,\pi^j)}{2}$. If this value is greater than the so far best found increment in weight, we do not have to search for connections from π^p to a genome in $cloud(\pi^i, \pi^j)$. A similar bounding strategy can be used when improving the tree topology.

2. When creating the clouds, it has turned out that even for the reversal distance, which is computable in linear time [BMY01], it is disadvantageous to use the real distance. Instead, we use a lower bound lb_{rev}, which is defined as follows.

$$lb_{rev}(\pi^i, \pi^j) = n - c(\pi^i, \pi^j)$$

In other words, when creating the set C_k out of the set C_{k-1}, we are looking for all operations that increment the number of cycles $c(\pi^i, \pi^j)$ in the breakpoint graph. An efficient way to determine these operations, even for the weighted reversal and transposition distance, can be found in [Mik03].

4.3 Experimental results

The accuracy and performance of our program, called `phylo`, has been evaluated on three different biological datasets. We compared our algorithm with the three state-of-the-art software tools `GRAPPA`, `MGR`, and `amGRP`. All tests were performed on a standard PC (Intel 3.16 GHz Core2 Duo CPU with 4 GB RAM).

4.3.1 Data sets

We tested our algorithm on the following three biological datasets, which can be considered as benchmarks for phylogenetic reconstruction algorithms based on genome rearrangements.

Campanulaceae: This dataset contains 13 chloroplast DNAs of the flowering plant family *Campanulaceae*, where each genome contains 105 elements. It was created by Cosner et al. as test case for their new method `MPBE` [CJM+00a], and at that time, it was ranked among the most challenging datasets for genome rearrangement based phylogenetic reconstruction.

Metazoan: This dataset contains 11 *metazoan* mitochondrial DNAs with 36 different elements. In the context of genome rearrangement algorithms, it was first used in [BKS99]. In the year 2002, Bourque and Pevzner published a tree with 150 reversals, showing that `MGR` outperforms `GRAPPA`, as `GRAPPA` was only able to find a tree with 175 reversals in more than 48 hours [BP02]. However, `GRAPPA` has been improved ever since, and the current version is now able to find a tree with 159 reversals in 39 seconds (see Section 4.3.4).

Protostomes: This dataset contains 62 *protostome* mitochondrial DNAs with 36 different elements. It was first published in [FSS06] and later adjusted in [BMM07] to be used as test scenario for `amGRP`. The increased amount of data and the larger genome distances make this dataset much more complicated than the Metazoan dataset.

4.3.2 Weight ratios

From the discussion in Section 1.3, it follows that in order to obtain biologically meaningful results, the weight ratio $w_r : w_t$ must be chosen somewhere between $1 : 1$

and 1 : 2. For other weight ratios, either transpositions are favored over reversals, or the sorting scenarios will contain virtually no transpositions. To cover the whole range of reasonable weightings, we performed tests using the weight ratios 1 : 1, 1 : 1.5, and 1 : 2. The weight w_r for reversals was fixed to 1, while the weight w_t for transpositions was set to the corresponding values.
For a better comparison with other tools, we also performed tests using the reversal distance.

4.3.3 Other tools using the reversal distance

We compared our results with the following three software tools, which are currently considered to be the state-of-the-art algorithms for the multiple genome rearrangement problem.

`GRAPPA`: We used the current version 2.0 [MT], which contains some serious improvements above older versions, especially it includes a reversal median solver. The best results were achieved with the parameters -t4 -T4 -n4 -e -m -a -C (for details see the `GRAPPA` manual). Using `DCM-GRAPPA` [LTM05] only improved the running times, but usually resulted in worse trees, so we do not provide the results from `DCM-GRAPPA` here.

`MGR`: We used the current version 2.01. The best results were achieved with the triplet resolution heuristic disabled. Note that the heuristics h3 and h5 are no longer available, thus we could not reproduce some of the results given in [BP02] and [BMM07].

`amGRP`: We used the version of April 2007. The best results were achieved with the "skewest" heuristic. As amGRP relies on randomness, we performed 50 runs for each data set, as suggested by the author [Ber07]. In Tables 4.1 to 4.3, the weight of the best found tree and the overall running time of all runs is provided. For more information about the variance of the output of `amGRP`, the reader is referred to [BMM07].

As all of these tools only create phylogenetic trees w.r.t. the reversal distance, a direct comparison is only possible for this distance measure. For the weighted reversal and transposition distance, we constructed the trees w.r.t. the reversal distance, and recalculated the weight of all edges with the weighted reversal and transposition distance, using the exact algorithm we provided in Section 3.3.1. Note that the same technique was used by Cosner et al. [CJM$^+$00a], where a tree was created w.r.t. the breakpoint distance, and the edge weights were recalculated using the software tool `DERANGE II` [BKS96].
As we did not automatize the recalculation of the tree weights, no running times are provided for these results.

4.3.4 Results

The results of our experiments are listed in Tables 4.1 to 4.3. An examination of the results shows that the accuracy is similar for all algorithms on the Campanulaceae dataset when using the reversal distance, and all algorithms were reasonable fast. In the more complicated Metazoan and Protostomes dataset, `phylo` and `amGRP` outperformed `GRAPPA` and `MGR` in terms of accuracy, in the latter dataset also in speed. While `phylo` and `amGRP` could solve this dataset in about 2 hours, `MGR` needed almost 9 hours, and `GRAPPA` did not terminate within 15 days before we aborted the program. The use of the weighted reversal and transposition distance drastically increased the running time, especially for the weight ratio $1 : 1$, but the algorithm still could solve every test case within one day. Despite of the increased running time, the results show that our approach is superior to constructing trees w.r.t. the reversal distance and recalculating the edge weights with the weighted reversal and transposition distance. While the results are similar for a weight ratio of $1 : 2$, our results are much better for other weight ratios. For example, the trees found by our algorithm for the Metazoan and Protostomes dataset at a weight ratio of $1 : 1$ have only about 80% of the weight of the trees found by `amGRP`.

4.4 Conclusion and discussion

We have developed a heuristic algorithm for the multiple genome rearrangement problem that directly uses the weighted reversal and transposition distance. Median calculations, which are the main bottleneck in other state-of-the-art algorithms, are avoided as long as possible and only used in the second improvement phase. The experimental results show that the approach is superior to the indirect approach, where trees are first created w.r.t. the reversal distance and then edge weights are recalculated with the weighted reversal and transposition distance.
Because of the very good accuracy and running time of `amGRP`, one might question whether it would be possible to simply replace its median solver for the RMP by a median solver for the wRTMP. However, in `amGRP`, almost the whole running time is used to solve instances of the RMP. Due to the higher complexity of the wRTMP, this technique would result in a significant increment of the running time of `amGRP`. The results in Section 3.3.4 suggest that this might be feasible for the weight ratio $w_r : w_t = 1 : 2$, where the wRTMP can be solved very fast. For any other weight ratio, solving instances of the wRTMP is too costly, and `amGRP` would be most likely much slower than our algorithm.

	phylo	GRAPPA	MGR	amGRP
d_{wrt}, $w_r = 1$, $w_t = 1$	38 (40 / 0 / 2) 0:59 (0:27 / 0:28 / 0:04)	46	47	41
d_{wrt}, $w_r = 1$, $w_t = 1.5$	50 (50.5 / 0.5 / 0) 0:32 (0:21 / 0:09 / 0:02)	55	56.5	52
d_{wrt}, $w_r = 1$, $w_t = 2$	62 (62 / 0 / 0) 0:22 (0:16 / 0:03 / 0:03)	64	64	63
d_{rev}	64 (65 / 1 / 0) 0:09 (0:04 / 0:04 / 0:01)	64 12:23	66 0:20	65 0:16

Table 4.1: Results for the Campanulaceae dataset. The entries consist of the weight of the best found tree (first line) and the running time in minutes of the algorithm (second line). For our algorithm, phylo, the values in brackets in the first line of each entry is the weight of the best tree after the construction phase, the decrement of the weight by improving the tree topology, and the decrement of weight by improving internal nodes. In the second line, the values in brackets are the running time for the construction phase and for the two improvement phases. For GRAPPA, MGR, and amGRP, running times are only provided for the reversal distance.

	phylo	GRAPPA	MGR	amGRP
d_{wrt}, $w_r = 1$, $w_t = 1$	85 (86 / 0 / 1) 159:35 (13:30 / 5:38 / 140:07)	114	108	106
d_{wrt}, $w_r = 1$, $w_t = 1.5$	119.5 (123.5 / 0 / 4) 45:16 (9:36 / 7:52 / 27:48)	137	130	125
d_{wrt}, $w_r = 1$, $w_t = 2$	146 (151 / 0 / 5) 12:55 (6:06 / 6:48 / 0:01)	159	151	144
d_{rev}	146 (148 / 0 / 2) 4:13 (1:39 / 2:09 / 0:25)	159 0:39	151 15:42	144 12:31

Table 4.2: Results for the Metazoan dataset. For details about the entries, see Table 4.1.

	phylo	MGR	amGRP
d_{wrt}, $w_r = 1$, $w_t = 1$	277 (283 / 1 / 5) 1404:23 (480:47 / 404:14 / 519:22)	364	348
d_{wrt}, $w_r = 1$, $w_t = 1.5$	389.5 (400 / 2 / 8.5) 1208:25 (347:19 / 423:37 / 507:29)	440	420.5
d_{wrt}, $w_r = 1$, $w_t = 2$	489 (499 / 5 / 5) 498:00 (256:34 / 233:22 / 8:04)	520	490
d_{rev}	503 (515 / 3 / 9) 123:24 (57:17 / 63:30 / 2:37)	528 536:15	500 108:33

Table 4.3: Results for the Protostomes dataset. For details about the entries, see Table 4.1. GRAPPA did not terminate within 15 days.

5 Rearrangement Distances with Duplications and Deletions

In this chapter, we focus on the following genome rearrangement problem. Given an ancestral genome ρ with unique gene content and the genome of a descendant π with arbitrary gene content, find a shortest sorting sequence of ρ w.r.t. π.
The chapter is organized as follows. In Section 5.1, we focus on the case where ρ and π are unichromosomal genomes, and the set of operations consists of reversals, block interchanges, tandem duplications, and deletions. We develop a lower bound for the distance, and a heuristic greedy algorithm that is based on this lower bound. In Section 5.2, the approach is extended to multichromosomal genomes. This extends the set of operations by translocations, fusions, fissions, chromosome duplications, and chromosome deletions. The accuracy of our approach is evaluated by experiments in Section 5.3. The limitations and further possible extensions of the algorithm are discussed in Section 5.4.

5.1 Sorting unichromosomal genomes

We now focus on a genome rearrangement problem for genomes with unequal gene content, i.e., elements can be duplicated or deleted. The algorithm is designed for unichromosomal genomes, i.e., ρ and π match our previous definition of a genome.

5.1.1 Problem definition

As the content of the ancestral genome ρ and the descendant π may differ, the set of operations must be enhanced by operations that can change the genome content. In addition to the classical operations, the following operations will be considered.

- A *tandem duplication* makes an exact copy of a segment of a genome, and inserts this copy exactly after the segment.
- A *transposition duplication* makes an exact copy of a segment, and inserts this copy at an arbitrary position in the genome. The copy may also be inverted.
- A *deletion* cuts a segment out of a genome.

We examine the following genome rearrangement problem. Given an ancestral genome ρ and the genome of a descendant π, find a sorting sequence of ρ w.r.t. π of minimal weight. The set of operations consists of reversals, tandem duplications, deletions (each with weight 1), and block interchanges (with weight 2). Transposition duplications are not explicitly allowed, but may be detected by our algorithm and simulated by a block interchange with a subsequent tandem duplication. For simplification, the ancestral genome ρ must contain each element of Σ_n exactly once (i.e., it is a permutation). The corresponding distance is denoted by $d_{uch}(\rho, \pi)$. Note that this distance is not a metric, because there is no inverse operation to a deletion.
As reversals can be simulated by a single DCJ operation and block interchanges can be simulated by two consecutive DCJs, our problem is equivalent to find a sorting sequence of minimum weight consisting of DCJs, tandem duplications, and deletions (each with weight 1), as long as we demand that if a DCJ creates a circular intermediate, it must be absorbed in the next step by another DCJ.

5.1.2 Idea of the algorithm

The main idea of the algorithm is to define a lower bound for $d_{uch}(\rho, \pi)$, and then to use a greedy algorithm that reduces this lower bound with each performed operation. The lower bound is based on the breakpoint graph, and uses the additional conditions that ρ satisfies. As these conditions are no longer valid if we apply an operation to ρ, we sort backwards, i.e., we create a sorting sequence of π w.r.t. ρ which consists of inverse operations. Note that simply restricting π instead of ρ would not work, because the operations are directed from the restricted ancestral genome to the unrestricted descendant, i.e., we would nevertheless have to invert the operations. As reversals and block interchanges can be simulated by DCJs, the task is to find a sorting sequence of π w.r.t. ρ consisting of DCJs, inverse tandem duplications, and inverse deletions. Note that the inverse of a DCJ is still a DCJ, while the inverse of a deletion is an insertion. To keep our original problem in mind, we will use the term "inverse deletion" instead of "insertion".

5.1.3 The breakpoint graph revisited

Although the breakpoint graph has been designed for permutations, we still can follow the instructions given in Section 1.2.4 for creating the breakpoint graph $BG(\rho, \pi)$, i.e.,

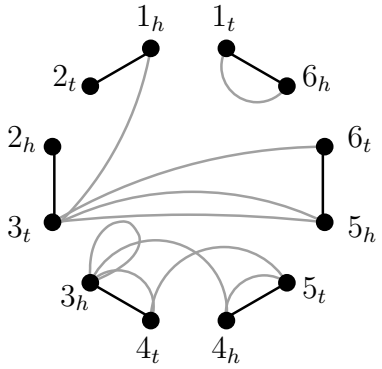


Figure 5.1: The breakpoint graph of $\rho = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6})$ and $\pi = (\overrightarrow{1}\ \overrightarrow{3}\ \overleftarrow{3}\ \overleftarrow{5}\ \overrightarrow{4}\ \overleftarrow{3}\ \overleftarrow{5}\ \overleftarrow{4}\ \overleftarrow{3}\ \overrightarrow{6})$. The edge $(3_t, 5_h)$ has a multiplicity of 2, all other edges have a multiplicity of 1. The edge $(3_h, 3_h)$ is a loop. The graph consists of 3 components, the edge $(1_h, 3_t)$ is a 1-bridge, the pair of edges $\{(3_h, 4_h), (4_t, 5_t)\}$ is a 2-bridge.

the set of black edges is

$$E_\rho = \{(u, v) \mid u, v \text{ are adjacent in } \rho \text{ and are not co-elements}\}$$

and the multiset of gray edges is

$$E_\pi = \{(u, v) \mid u, v \text{ are adjacent in } \pi \text{ and are not co-elements}\}$$

An example can be seen in Fig. 5.1. Note that two different descendants can have the same breakpoint graph. For example, if $\rho = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3})$, $\pi^a = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{1}\ \overleftarrow{3})$, and $\pi^b = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overleftarrow{1}\ \overleftarrow{3})$, then $BG(\rho, \pi^a) = BG(\rho, \pi^b)$. As π can have multiple occurrences of each element, also E_π can have identical edges, i.e., it is a multiset. The *multiplicity* of an edge (x, y) is the number of gray edges between x and y. An edge (x, x) is called a *loop*, the number of nodes x which are the endpoint of a loop (x, x) in $BG(\rho, \pi)$ is denoted by $L(\rho, \pi)$. The breakpoint graph does no longer decompose into cycles, but into connected *components*. These components are not to be confused with the components known from SBR and the Hannenhalli-Pezner Theory, but take on the role of cycles in SBR. The number of components in $BG(\rho, \pi)$ is denoted by $C(\rho, \pi)$. A gray edge is called a *1-bridge* if the removal of this edge increases $C(\rho, \pi)$. A pair of gray edges is called a *2-bridge* if none of them is a 1-bridge, and the removal of both edges increases $C(\rho, \pi)$.

5.1.4 A lower bound

We will now examine some properties of the breakpoint graph to derive a lower bound for $d_{uch}(\rho, \pi)$.

Lemma 5.1. *If $\pi = \rho$, then the breakpoint graph $BG(\rho, \pi)$ has n components and no loops. There is no genome π with $C(\rho, \pi) > n$.*

Proof. Each node is connected with another node by a black edge, therefore the maximum possible number of components in the breakpoint graph is n. If $\pi = \rho$, all gray edges are parallel to the black edges, therefore it has n components and no loops. □

The inverse operations can have the following effects on $C(\rho, \pi)$ and $L(\rho, \pi)$.

DCJ: A DCJ cuts the genome at two positions, and rejoins the cut ends. This has the following effect on the breakpoint graph. Two gray edges (u, v) and (x, y) are removed, and w.l.o.g. the gray edges (u, x) and (v, y) are added to the breakpoint graph. This can increase the number of components by at most 1. If one of the removed edges is a loop, all three vertices are in the same component after the operation, i.e., the number of components is not increased by this operation. As a DCJ removes only two edges, $L(\rho, \pi)$ can be decreased by at most 2.

Inverse tandem duplication: An inverse tandem duplication deletes the following gray edges. (a) Edges that are inside the duplicated segment. All these edges have a multiplicity ≥ 2, thus deleting these edges neither changes $C(\rho, \pi)$ nor $L(\rho, \pi)$. (b) The edge between the last node of the segment and the first node of the copy. This can increase the number of components by 1, or decrease $L(\rho, \pi)$ by 1 (but not both).

Inverse deletion: An inverse deletion splits the genome at one position and adds arbitrary elements. In the breakpoint graph, one gray edge is removed, and several gray edges are added. An inverse deletion can increase the number of components by 1, or remove one loop. As the number of components can only be increased if the removed edge is a 1-bridge, it cannot simultaneously increase $C(\rho, \pi)$ and decrease $L(\rho, \pi)$ by removing a loop.

These effects are also illustrated in Fig. 5.2.

Theorem 5.2. *A lower bound $lb_{uch}(\rho, \pi)$ for $d_{uch}(\rho, \pi)$ can be defined as follows.*

$$d_{uch}(\rho, \pi) \geq lb_{uch}(\rho, \pi), \text{ where } lb_{uch}(\rho, \pi) := n - C(\rho, \pi) + \sum_{C_i \in Components} \left\lceil \frac{L_i}{2} \right\rceil$$

and L_i is the number of nodes with a loop in component C_i.

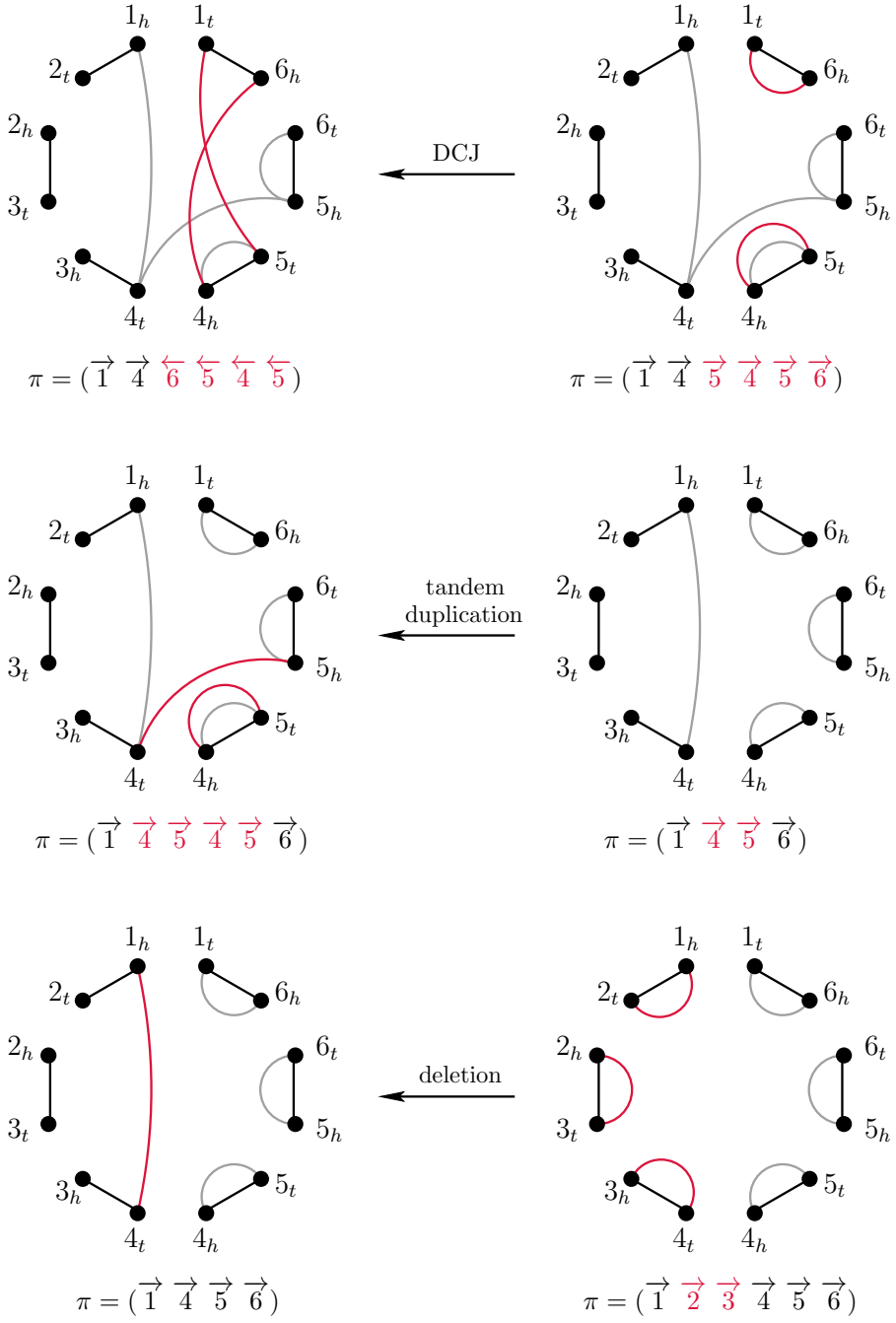


Figure 5.2: The effects of the different operations on the breakpoint graph. In all cases, $\rho = (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6})$.

Proof. According to Lemma 5.1, $n - C(\rho, \pi)$ components must be created and all loops must be removed by the sorting sequence. There is no inverse operation that simultaneously increases the number of components and decreases $L(\rho, \pi)$. An inverse operation can increase $C(\rho, \pi)$ by at most 1, and therefore decrease $lb_{uch}(\rho, \pi)$ by at most 1. For inverse operations that decrease $L(\rho, \pi)$, there are three cases. (a) It decreases $L(\rho, \pi)$ by 1. (b) It decreases $L(\rho, \pi)$ by 2, and both nodes where a loop is removed belong to the same component. (c) It decreases $L(\rho, \pi)$ by 2, and the nodes where a loop is removed belong to different components. It is easy to see that Cases (a) and (b) can decrease the lower bound by at most 1. Case (c) may decrease $\sum_{C_i \in Components} \left\lceil \frac{L_i}{2} \right\rceil$ by 2. However, this is only possible if the inverse operation is a DCJ that merges two components, i.e., also $C(\rho, \pi)$ is decreased by 1. □

Corollary 5.3. *If $\pi = \rho$, then $lb_{uch}(\rho, \pi) = 0$.*

Unfortunately, there are genomes $\pi \neq \rho$ with $lb_{uch}(\rho, \pi) = 0$, e.g., $\rho = (\overrightarrow{1}\,\overrightarrow{2}\,\overrightarrow{3})$ and $\pi = (\overrightarrow{1}\,\overrightarrow{2}\,\overrightarrow{3}\,\overrightarrow{1}\,\overrightarrow{2}\,\overrightarrow{3})$. Therefore, it is not sufficient to sort until the lower bound reaches 0. To overcome this problem, we introduce the *disruption measure* $\tau(\rho, \pi)$, which is based on the following two ideas.

1. π is similar to ρ if it has many adjacencies w.r.t. ρ.
2. π is similar to ρ if all elements have a similar multiplicity in π and ρ.

These ideas are reflected by the following definitions.

$$a(\rho, \pi) := \text{number of adjacencies in } \pi \text{ w.r.t. } \rho$$

$$m(\rho, \pi) := \sum_{x \in \{1,\ldots,n\}} |mult(x, \pi) - 1|$$

Removing one occurrence of a duplicated segment of length l (which intuitively brings π closer to ρ) decreases $m(\rho, \pi)$ by l, but may also decrease $a(\rho, \pi)$ by up to l. Thus, the disruption measure must weight the multiplicity of elements more than the number of breakpoints, leading to the following definition.

$$\tau_{uch}(\rho, \pi) = n - a(\rho, \pi) + 2 \cdot m(\rho, \pi)$$

Lemma 5.4. *If $\pi = \rho$, then $\tau_{uch}(\rho, \pi) = 0$. Otherwise, $\tau_{uch}(\rho, \pi) > 0$.*

Proof. For each adjacency $(x_{h/t}, y_{h/t})$, we can say that its contribution to $a(\rho, \pi)$ is shared between the two extremities. Thus, each occurrence of an element contributes at most 1 to $a(\rho, \pi)$. By adding the contribution of all occurrences of an element x, the contribution of x to $\tau_{uch}(\rho, \pi)$ is $\geq 2 \cdot |mult(x, \pi) - 1| - mult(x, \pi)$ which is minimized if $mult(x, \pi) = 1$. Thus, $\tau_{uch}(\rho, \pi) = 0$ if and only if all elements have a multiplicity of 1, and there are no breakpoints. In this case, π and ρ are identical. □

5.1.5 The algorithm

The algorithm uses a greedy strategy to sort the genome. In each step, it searches for inverse operations that decrease the lower bound, i.e., they either increase $C(\rho, \pi)$ or decrease $L(\rho, \pi)$, and check their effect on the lower bound. For selecting one of these operations, we define the *score* of an inverse operation iop as the tuple $\sigma_{uch}(\rho, \pi) = (\Delta lb_{uch}(\rho, \pi), \Delta\tau_{uch}(\rho, \pi))$, with $\Delta lb_{uch} = lb_{uch}(\rho, \pi) - lb_{uch}(\rho, iop \cdot \pi)$ and $\Delta\tau_{uch} = \tau_{uch}(\rho, \pi) - \tau_{uch}(\rho, iop \cdot \pi)$. The comparison operator between two scores is defined by $\sigma_{uch}(iop_1) > \sigma_{uch}(iop_2)$ if $\Delta lb_{uch}(iop_1) > \Delta lb_{uch}(iop_2) \vee (\Delta lb_{uch}(iop_1) = \Delta lb_{uch}(iop_2) \wedge \Delta\tau_{uch}(iop_1) > \Delta\tau_{uch}(iop_2))$. In each step, we select the inverse operation with the greatest score from the set of inverse operations that decrease the lower bound. If there is no such operation, we use additional heuristics to search for small sequences of inverse operations that decrease $\tau_{uch}(\rho, \pi)$ without increasing $lb_{uch}(\rho, \pi)$.
For simplification of the presentation of the algorithm, we assume in this section that ρ is the identity genome $(\overrightarrow{1} \ldots \overrightarrow{n})$.

Decreasing the lower bound

As a DCJ removes two gray edges and rejoins the endpoints with two new gray edges, it can only increase $C(\rho, \pi)$ if the removed edges are a 2-bridge, or two 1-bridges in the same component. If the DCJ rejoins the endpoints such that we get a linear genome and a circular intermediate, we need a lookahead to search for another DCJ that absorbs this circular intermediate. Those two DCJs are directly merged into two reversals or one block interchange with a weight of 2. Inverse tandem duplications can only remove one gray edge with a multiplicity of 1 (the one between the duplicated segments), thus an inverse tandem duplication increases $C(\rho, \pi)$ if and only if this edge is a 1-bridge. Inverse deletions remove only one gray edge, thus also an inverse deletion can increase $C(\rho, \pi)$ only if the removed edge is a 1-bridge. In summary, the main task in finding inverse operations that increase $C(\rho, \pi)$ is to find 1-bridges and 2-bridges in the breakpoint graph. This can be done in linear time (see e.g. [NI92, Tsi09]). For all 1-bridges, we additionally have to check if we can apply an inverse tandem duplication on it (i.e., check whether there are identical segments on both sides of the edge), and if we can apply an inverse deletion on it (i.e., check whether there is a segment that can be inserted such that no gray edge in the segment merges two components). For the latter case, practical tests have shown that there is such a segment in most cases, but it is better to only allow segments that contain no breakpoint w.r.t. ρ.
Finding inverse operations that decrease $L(\rho, \pi)$ is rather straightforward, as we just have to scan the breakpoint graph for loops with a multiplicity of 1 and find the corresponding position in the genome. An operation that decreases $L(\rho, \pi)$ can be an inverse tandem duplication or an inverse deletion that removes this loop, or a DCJ that removes two loops with a multiplicity of 1, or a DCJ that acts on a loop and another gray edge of the same component.

Heuristics for the remaining cases

If there is no inverse operation that decreases the lower bound, we search for inverse operations or small sequences of inverse operations that decrease $\tau_{uch}(\rho, \pi)$ without increasing $lb_{uch}(\rho, \pi)$. As an exhaustive search would be too expensive, we will use the following heuristics.
If there are two consecutive copies of the same segment, we can remove one of them by an inverse tandem duplication. As an inverse tandem duplication only removes gray edges, it can never increase the lower bound, but decreases $\tau_{uch}(\rho, \pi)$. This is different in the general case of an inverse transposition duplication, where the duplicated segments are separated by a non-empty segment in the genome. In this case, the removal of one of these segments (which can be simulated by a block interchange and an inverse tandem duplication) creates a new gray edge between the last element before the removed segment and the first element after the removed segment. If the corresponding nodes in the breakpoint graph are in the same component and not identical, we can safely apply this operation. Otherwise, it would decrease the number of components or create a new loop, i.e., there are inverse transposition duplications that increase the lower bound. However, if we have at least three copies of the segment, the situation is different.

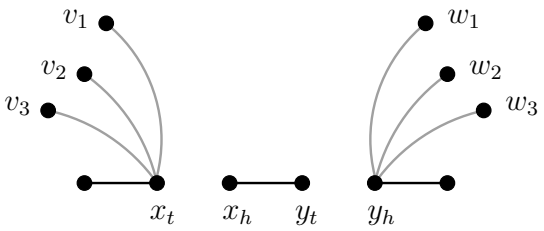
Lemma 5.5. *If there are three identical copies of a segment that are maximal (i.e., they cannot be extended in any direction such that still all three copies are identical), then there exists a sequence of inverse operations that removes two of these copies and does not increase the lower bound.*

Proof. Assume that the duplicated segment is of the form $\overrightarrow{x} \ldots \overrightarrow{y}$, i.e., the segment starts with x_t and ends with y_h. There are gray edges (v_1, x_t) and (y_h, w_1), (v_2, x_t) and (y_h, w_2), and (v_3, x_t) and (y_h, w_3) (from the elements enclosing the first, second, and third copy of the segment). Because the segment is maximal, we can assume w.l.o.g. that $w_1 \neq w_2$. As v_1, v_2, and v_3 are all adjacent to x_t, they must be in the same component, as well as w_1, w_2, and w_3. By deleting the first two segments, we remove the gray edges (v_1, x_t), (y_h, w_1), (v_2, x_t), and (y_h, w_2), and get the new gray edges (v_1, w_1) and (v_2, w_2). If this merges two components, the new gray edges are a 2-bridge, and we can apply a DCJ that replaces them by the gray edges (v_1, v_2) and (w_1, w_2). If $v_1 = v_2$ this can create a new loop. This loop can be removed by another DCJ between the edges (v_1, v_2) and (v_3, x_t) (note that $v_3 \neq v_1$ because the segments are maximal, and $v_1 \neq x_t$ because otherwise the loop was already there before the operation). In fact, the operations of the sequence can be arranged such that all DCJs are reversals, so we do not have to find appropriate follow-ups. An illustration of the sequence is depicted in Fig. 5.3. □

We now examine what we can do with elements with a multiplicity of at most 2. A first strategy would be to create adjacencies wherever this is possible without creating

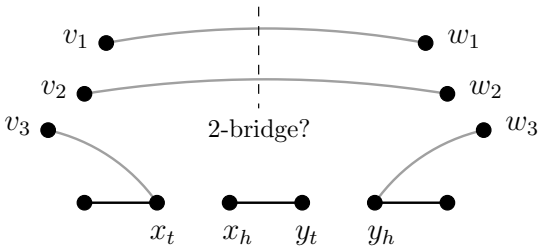
$\ldots \overrightarrow{a_1}\overrightarrow{x}\overrightarrow{y}\overrightarrow{b_1} \ldots \overrightarrow{a_2}\overrightarrow{x}\overrightarrow{y}\overrightarrow{b_2} \ldots \overrightarrow{a_3}\overrightarrow{x}\overrightarrow{y}\overrightarrow{b_3} \ldots$

Remove $2 \times \overrightarrow{x}\overrightarrow{y}$ by 2 block interchanges + 2 inverse tandem duplications



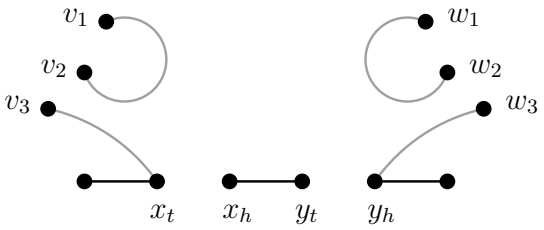
$\ldots \overrightarrow{a_1}\overrightarrow{b_1} \ldots \overrightarrow{a_2}\overrightarrow{b_2} \ldots \overrightarrow{a_3}\overrightarrow{x}\overrightarrow{y}\overrightarrow{b_3} \ldots$

If merge of components, apply reversal



$\ldots \overrightarrow{a_1}\overleftarrow{a_2} \ldots \overleftarrow{b_1}\overrightarrow{b_2} \ldots \overrightarrow{a_3}\overrightarrow{x}\overrightarrow{y}\overrightarrow{b_3} \ldots$

If $v_1 = v_2$, apply reversal



$\ldots \overrightarrow{a_1}\overleftarrow{a_3} \ldots \overleftarrow{b_2}\overrightarrow{b_1} \ldots \overrightarrow{a_2}\overrightarrow{x}\overrightarrow{y}\overrightarrow{b_3} \ldots$

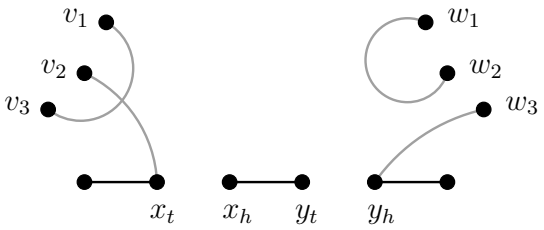


Figure 5.3: An example of a sequence that removes two of the segments $\overrightarrow{x}\overrightarrow{y}$ without increasing the lower bound. If some of the segments $\overrightarrow{x}\overrightarrow{y}$ are inverted, a similar sequence can be applied.

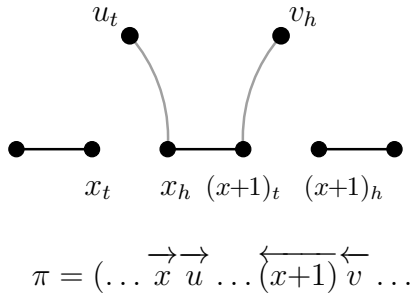


Figure 5.4: The configuration in which a DCJ can create an adjacency without creating a loop.

loops (note that creating adjacencies cannot decrease the number of components). As a precondition, there must be a black edge $(x_h, (x+1)_t)$ and gray edges $(x_h, u_{t/h})$ and $((x+1)_t, v_{t/h})$ with $u_{t/h} \neq v_{t/h}$ (see Fig. 5.4).

If there are no further adjacencies to create, and all elements have a multiplicity of at most 2, all possible cases for a black edge and its adjacent gray edges are depicted in Fig. 5.5. For all of these cases, there are sequences of inverse operations that decrease $\tau_{uch}(\rho, \pi)$ and do not increase $lb_{uch}(\rho, \pi)$. Some of these sequences require an inverse transposition duplication. In our algorithm, this will be simulated by a block interchange and an inverse tandem duplication.

Case (a): The genome is of the form $\pi = (\ldots \overrightarrow{u}\,\overrightarrow{x} \ldots)$, the element $x-1$ is missing. Let y be the largest element $< x$ that is not missing. We apply an inverse deletion of the elements $\overrightarrow{y+1}$ to $\overrightarrow{x-1}$ between $\overrightarrow{u}$ and $\overrightarrow{x}$, i.e., π becomes $(\ldots \overrightarrow{u}\,\overrightarrow{y+1} \ldots \overrightarrow{x-1}\,\overrightarrow{x} \ldots)$. The gray edge (x_t, u_h) is removed, the inserted gray edges are the edge $(u_h, (y+1)_t)$ and some edges corresponding to adjacencies, i.e., they are parallel to a black edge. The black edge $((x-1)_h, x_t)$ is split from the component, the edge $(u_h, (y+1)_t)$ may merge two components, so the overall number of components cannot be decreased. As the element $y+1$ was not present in π before the operation, the edge $(u_h, (y+1)_t)$ cannot be a loop, thus the lower bound is not increased by this operation.

Case (b): x is in a duplicated segment, w.l.o.g. the segment is left-maximal. We extend it to the right until it is also right-maximal, yielding the duplicated segment $\overrightarrow{x} \ldots \overrightarrow{y}$ (of course, y may also have negative orientation). Thus, the genome is of the form $\pi = (\ldots \overrightarrow{x-1}\,\overrightarrow{x} \ldots \overrightarrow{y}\,\overrightarrow{u} \ldots \overrightarrow{v}\,\overrightarrow{x} \ldots \overrightarrow{y}\,\overrightarrow{w} \ldots)$. The duplicated segments and the adjacent elements may also have a negative orientation. As the segment is right-maximal, $u_t \neq w_t$ or the segments have different orientation and touch each

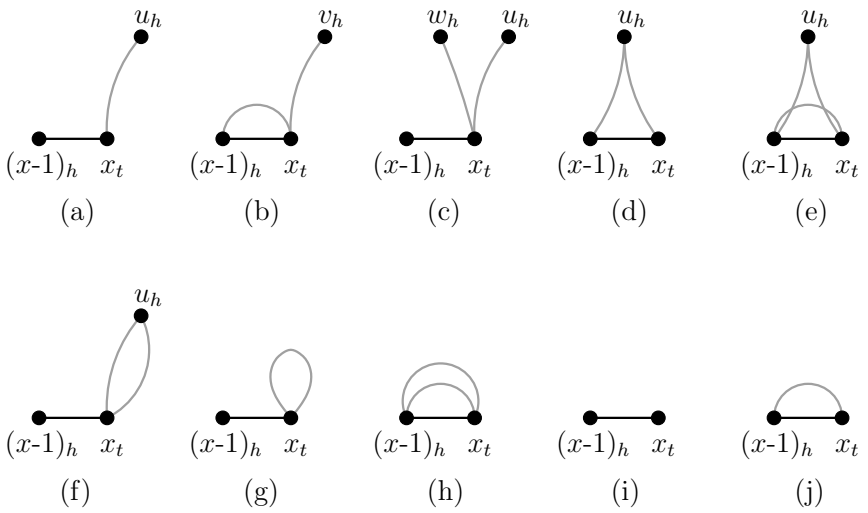


Figure 5.5: The different configurations in which each node is adjacent to at most 2 gray edges, and no adjacency can be created without creating a loop. In all cases, the picture shows a black edge and all its adjacent gray edges. Whether a node is the head or tail of an element may differ from the given labeling.

other, i.e., $\pi = (\dots \overrightarrow{x-1}\,\overrightarrow{x} \dots \overrightarrow{y}\,\overleftarrow{y} \dots \overleftarrow{x}\,\overleftarrow{v} \dots)$. In the former case, we remove the copy of the segment that is not adjacent to $x-1$, i.e., we remove the gray edges (v_h, x_t) and (y_h, w_t), and create the new gray edge (v_h, w_t). If $v_h = w_t$, the loop can be removed by a DCJ that acts on this edge and the edge (y_h, u_t). In the latter case, we remove the copy of x that is not adjacent to $x-1$, i.e., we remove the loop and the gray edge (v_h, x_t), and we create the gray edge (y_h, v_h). In both cases, the black edge $((x-1)_h, x_t)$ is split from the component, and adding one new gray edge can merge only two components, so the overall number of components does not decrease. Additionally, we do not create any loops, thus the lower bound does not increase.

Case (c): The genome is of the form $\pi = (\dots \overrightarrow{u}\,\overrightarrow{x} \dots \overrightarrow{y}\,\overrightarrow{v} \dots \overrightarrow{w}\,\overrightarrow{x} \dots \overrightarrow{y}\,\overrightarrow{z} \dots)$, where $\overrightarrow{x} \dots \overrightarrow{y}$ is a duplicated segment that cannot be extended in any direction (of course, the elements may also have a negative orientation). We remove the second copy of the duplicated segment. This removes the gray edges (w_h, x_t) and (y_h, z_t) and adds the gray edge (w_h, z_t). If this merges two components, then (u_h, x_t) and (w_h, z_t) are 1-bridges with disjoint endpoints (remember that there is no gray edge from node $(x-1)_h$), so a DCJ on these two edges splits the component again.

If $w_h = z_t$, this sequence would create a loop, so we do not apply it. Instead, we use the symmetrical case in which we remove the first copy of the duplicated segment. If both $w_h = z_t$ and $u_h = v_t$, we apply the first sequence, and remove the loop (w_h, z_t) by applying a DCJ on it and the gray edge (u_h, x_t). Note that there is the possibility that the first DCJ creates a circular intermediate that cannot be absorbed in the next step. In this case, we can apply the sequence for Case (a) twice, i.e., we add the same elements before both copies of x.

Case (d) and (e): If all elements have a multiplicity of at most 2, and there is no DCJ that creates an adjacency, then the black edge incident to u_h must correspond to Case (c), thus we can apply the according sequence.

Case (f), (g), (h), and (i): If any of the Cases (a) to (e) occurs in the breakpoint graph, apply the corresponding sequence. Now, assume that none of these cases occurs in the breakpoint graph. The elements on both sides of the black edges corresponding to Cases (f) to (i) have multiplicity 0 or 2. As the two gray edges starting from one node always go to the same node, the genome consists solely of two identical segments $\overrightarrow{x} \ldots \overrightarrow{y}$, i.e., it is of the form $\pi = (\overrightarrow{x} \ldots \overrightarrow{y}\,\overrightarrow{x} \ldots \overrightarrow{y})$ (if the breakpoint graph does not contain Case (g)) or $\pi = (\overrightarrow{x} \ldots \overrightarrow{y}\,\overleftarrow{y} \ldots \overleftarrow{x})$ (if the breakpoint graph contains Case (g)). We remove one of the segments, either by an inverse tandem duplication, or by a reversal and an inverse tandem duplication. While in the first case gray edges are only removed, the second case may merge two components with the new gray edge (y_h, x_t). However, as Case (g) may only occur between the two duplicated segments, all remaining loops of the breakpoint graph are removed, therefore the lower bound does not increase.

Case (j): If all black edges in the breakpoint graph correspond to Case (j), then $\pi = \rho$, and the algorithm stops.

All these sequences decrease $\tau_{uch}(\rho, \pi)$, a summary of their effects is listed in Table 5.1.

Completeness of the algorithm

If there is no inverse operation that decreases the lower bound, the algorithm searches for all inverse operations and sequences mentioned in the last paragraph, and again performs the one with the greatest score. The whole algorithm in pseudocode can be seen in Algorithm 5.1.

Theorem 5.6. *The algorithm terminates after a finite number of steps. When the algorithm terminates, π is transformed into ρ.*

Sequence	$\Delta m(\rho,\pi)$	$\Delta a(\rho,\pi)$	$\Delta \tau_{uch}(\rho,\pi)$
Inverse Tandem Duplication	$-l$	$\geq -l$	$\geq l$
Segments with multiplicity ≥ 3	$-2l$	$\geq -2l-1$	$\geq 2l-1$
Creating adjacencies	0	≥ 1	≥ 1
Case (a)	$-l$	l	$2l$
Case (b)	$-l$	$\geq -l$	$\geq l$
Case (c)	$-l$	$\geq -l$	$\geq l$
Case (c')	0	$2l$	$2l$
Case (f) - (i)	$-l$	$\geq -l$	$\geq l$

Table 5.1: Changes of $m(\rho,\pi)$, $a(\rho,\pi)$, and $\tau_{uch}(\rho,\pi)$ by applying the different sequences of operations described in this section. Case (c') is the case where we cannot solve Case (c) directly and have to apply the sequence for Case (a) twice. l denotes the length of inserted or removed segments. Note that for all sequences, $\Delta\tau_{uch} > 0$.

Algorithm 5.1 Heuristic algorithm for finding a sorting sequence of ρ w.r.t. π (unichromosomal case)

```
1: function findSequence(ρ, π)
2:   while (lb_uch(ρ, π), τ_uch(ρ, π)) ≠ (0, 0) do
3:     find all inverse operations that decrease lb_uch(ρ, π)
4:     if inverse operation found then
5:       apply inverse operation with maximal score
6:     else
7:       find inverse tandem duplications
8:       find inverse transposition duplications
9:       find sequences for segments with multiplicity ≥ 3
10:      find DCJs that create adjacencies
11:      find sequences for Cases (a) - (i)
12:      apply sequence with maximal score
13:  invert and output the sorting sequence
```

Proof. As long as $\tau_{uch}(\rho, \pi) > 0$, the algorithm finds and performs an inverse operation. Thus, if the algorithm stops, $\tau_{uch}(\rho, \pi) = 0$, and according to Lemma 5.4, $\pi = \rho$. As none of the applied operations increases the lower bound, only $lb_{uch}(\rho, \pi)$ inverse operations that decrease the lower bound can be applied. If there is no such operation, we apply sequences of inverse operations that decrease $\tau_{uch}(\rho, \pi)$. Because of $\tau_{uch}(\rho, \pi) \geq 0$, only a finite number of these sequences can be applied between two inverse operations that decrease the lower bound. Thus, the overall number of performed operations is finite. □

5.2 Sorting multichromosomal genomes

We now extend the algorithm to multichromosomal genomes. As in most species with a multichromosomal genome, the chromosomes are linear, also our model uses linear chromosomes. This requires an extension of our former definition of a genome, as well as the introduction of further operations.

5.2.1 Additional definitions

As the definition of a genome from Section 1.2 is not capable of modelling multichromosomal genomes, we need to alter this definition. A *chromosome* $\pi^i = (\pi^i_1 \dots \pi^i_k)$ is a string over the alphabet $\Sigma_n = \{1, \dots, n\}$, where each element has a positive or negative orientation (indicated by $\overrightarrow{x}$ or $\overleftarrow{x}$), i.e., a chromosome corresponds to the former definition of a genome, except for the fact that it is not circular. The *reflection* of a chromosome $\pi^i = (\pi^i_1 \dots \pi^i_n)$ is the chromosome $-\pi^i = (-\pi^i_n \dots - \pi^i_1)$, i.e., the order of the elements as well as the orientation of each element is inverted. π^i and $-\pi^i$ are considered to be equivalent. A *genome* $\pi = \{\pi^1, \dots, \pi^m\}$ is a multiset of chromosomes. The first and the last element of a chromosome written in extremities notation are its *telomeres*, $t(x_t, \pi)$ and $t(x_h, \pi)$ denote how often x_t and x_h are telomeres in π. If a telomere $x_{t/h}$ occurs in two genomes π and ρ, then $x_{t/h}$ is a *telomere adjacency* of π and ρ. If it occurs solely in π, $x_{t/h}$ is a *telomere breakpoint* of π w.r.t. ρ. Analogously, an unordered pair of extremities $\{x_{t/h}, y_{t/h}\}$ that are not co-elements is an *inner adjacency* of π and ρ if it occurs in both π and ρ, and an *inner breakpoint* of π w.r.t. ρ if it occurs solely in π.
Additionally to the previously defined operations, we consider the following operations.

- A *translocation* splits two chromosomes π^i and π^j at two positions s and t and rejoins the segments, yielding the chromosomes $(\pi^i_1 \dots \pi^i_{s-1} \pi^j_t \dots \pi^j_{l(j)})$ and $(\pi^j_1 \dots \pi^j_{t-1} \pi^i_s \dots \pi^i_{l(i)})$, where $l(i)$ is the length of chromosome π^i and $l(j)$ is the length of chromosome π^j.
- A *fission* cuts a chromosome into two chromosomes.

- A *fusion* concatenates two chromosomes.
- A *chromosome duplication* adds an exact copy of a chromosome to the genome.
- A *chromosome deletion* deletes a chromosome from a genome.

We now focus on the following genome rearrangement problem. Given an ancestral genome ρ and the genome of a descendant π, find a sorting sequence of ρ w.r.t. π of minimal weight. The set of operations consists of reversals, translocations, fissions, fusions, tandem duplications, deletions, chromosome duplications, chromosome deletions (each with weight 1), and transpositions (with weight 2). For simplification, the ancestral genome ρ must satisfy the following conditions.

1. Two chromosomes in ρ are either disjoint or identical.
2. Each element in Σ_n occurs in at least one chromosome of ρ.
3. Each element in Σ_n occurs at most once in each chromosome of ρ.

The corresponding distance is denoted by $d_{mch}(\rho, \pi)$. As in the unichromosomal case, also this distance is not a metric.
It has been shown in [YAF05] that reversals, translocations, fusions, and fissions can all be simulated by a single DCJ operation, while transpositions can be simulated by two consecutive DCJs. Therefore, it is sufficient to focus on DCJs and the operations that change the genome content, and demand that DCJs that create a circular intermediate are followed by another DCJ such that the combination of both DCJs corresponds to a transposition.

5.2.2 A further extension of the breakpoint graph

The main idea of the algorithm is the same as in the algorithm for the unichromosomal case: define a lower bound based on the breakpoint graph, and use a greedy heuristic to sort backwards from π to ρ. Technically, it would be possible to use the same definition of the breakpoint graph, but to obtain a strong lower bound, we need to modify it. The definition remains basically the same, with the exception that gray edges connect neighboring extremities in π only if none of these extremities is a telomere in ρ, i.e., $BG(\rho, \pi) = (V, E_\rho \cup E_\pi)$, with set of nodes $V = \{1_t, 1_h, \ldots, n_t, n_h\}$, the multiset of black edges is

$$E_\rho = \{(u, v) \mid u, v \text{ are adjacent in } \rho \text{ and are not co-elements}\},$$

and the multiset of gray edges is

$$E_\pi = \{(u, v) \mid u, v \text{ are adjacent in } \pi \text{ and are not co-elements, and } u \text{ and } v \text{ are not telomeres in } \rho\}.$$

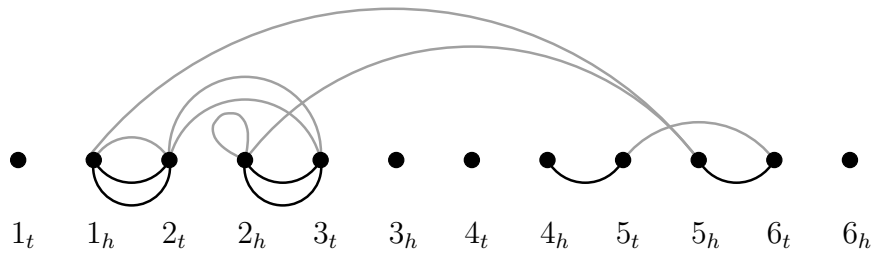


Figure 5.6: The breakpoint graph of $\rho = \{(\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}), (\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}), (\overrightarrow{4}\ \overrightarrow{5}\ \overrightarrow{6})\}$ and $\pi = \{(\overrightarrow{1}\ \overrightarrow{2}\ \overleftarrow{2}\ \overrightarrow{3}), (\overrightarrow{4}\ \overleftarrow{3}\ \overrightarrow{2}\ \overleftarrow{5}\ \overrightarrow{6}), (\overrightarrow{5}\ \overleftarrow{1})\}$. There is no gray edge between 4_h and 3_h because 3_h is a telomere in ρ.

Components, loops, 1-bridges, and 2-bridges are defined as in Section 5.1. To emphasize the fact that the chromosomes are linear and not circular, we draw the nodes of the breakpoint graph on a straight line. An example of a breakpoint graph is depicted in Fig. 5.6.

5.2.3 A lower bound

In the unichromosomal case, we were mainly interested in the properties of the breakpoint graph. As the breakpoint graph does not contain edges for telomeres, we additionally have to consider the amount of *incorrect telomeres* (denoted by $T(\rho, \pi)$), which is defined as follows.

$$T(\rho,\pi) = \sum_{x_{t/h} | t(x_{t/h},\rho) > 0} \max\{t(x_{t/h},\rho) - t(x_{t/h},\pi), 0\} + \sum_{x_{t/h} | t(x_{t/h},\rho) = 0} t(x_{t/h},\pi)$$

This formula is inspired by the following consideration. We are seeking a sequence of inverse operations that transforms π into ρ. If an extremity $x_{t/h}$ occurs k times as a telomere in ρ, but only $j < k$ times as a telomere in π, then this telomere must be created $k - j$ times in π (this results in the first summand of the formula). If an extremity $x_{t/h}$ is a telomere in π, but not in ρ, this telomere must be removed from π (this results in the second summand of the formula). For an illustration, see Fig. 5.7. Although it is possible to count the exact amount of telomeres which have to be removed or created (i.e., using the formula $\sum_{Extremities\ x_{t/h}} |t(x_{t/h},\rho) - t(x_{t/h},\pi)|$), this is disadvantageous in practice, because if $t(x_{t/h},\pi) > t(x_{t/h},\rho) > 0$, the algorithm cannot decide which of the telomeres in π correspond to those in ρ, and which have to be removed. Therefore, extremities $x_{t/h}$ with $t(x_{t/h},\pi) > t(x_{t/h},\rho) > 0$ are not considered in the formula.

$$\rho = (1_t 1_h 2_t 2_h)(1_t 1_h 2_t 2_h)(3_t 3_h 4_t 4_h)$$

$$\pi = (1_t 1_h 2_t 2_h 3_h 3_t)(3_t 3_h 4_t 4_h)(4_t 4_h)$$

Figure 5.7: Two genomes ρ and π written in extremities notation. Extremities accounting for $T(\rho, \pi)$ are drawn in red. The telomere 1_t in ρ accounts only once for $T(\rho, \pi)$, because there is one corresponding telomere in π for the other occurence of this telomere in ρ. Both telomeres 4_h in π do not account for $T(\rho, \pi)$, because 4_h is also a telomere in ρ.

Lemma 5.7. *If $\pi = \rho$, then $C(\rho, \pi) = n + ch(\rho)$, $T(\rho, \pi) = 0$, and $L(\rho, \pi) = 0$, where $ch(\rho)$ is the number of disjoint chromosomes in ρ. For all genomes π, it holds that $C(\rho, \pi) \leq n + ch(\rho)$, $L(\rho, \pi) \geq 0$, and $T(\rho, \pi) \geq 0$.*

Proof. The breakpoint graph has $2ch(\rho)$ nodes that are a telomere in ρ, and $2(n-ch(\rho))$ nodes that are not a telomere in ρ. The latter nodes are pairwise connected by black edges. Therefore, the maximum possible number of components is $2ch(\rho)+\frac{2(n-ch(\rho))}{2} = n+ch(\rho)$. It is easy to see that both $L(\rho, \pi)$ and $T(\rho, \pi)$ must always be ≥ 0. If $\pi = \rho$, all gray edges are parallel to the black edges, therefore it has $n + ch(\rho)$ components, no loops, and no incorrect telomeres. □

We now examine the effects of the inverse operations on $C(\rho, \pi)$, $L(\rho, \pi)$, and $T(\rho, \pi)$.

Inverse reversal, translocation, transposition: These operations can be simulated by one or two DCJs, thus it is sufficient to examine the effects of a DCJ (note that transpositions, which require two DCJs, have weight 2). We have already shown in the last section that a DCJ can increase the number of components by 1 and remove at most 2 loops, but not both simultaneously. This still holds for the modified breakpoint graph. A DCJ can only change $T(\rho, \pi)$ if one cut is at the end of a chromosome. Then, a telomere $x_{t/h}$ is removed and a new telomere $y_{t/h}$ is created. This decreases $T(\rho, \pi)$ by 1 or 2 if $x_{t/h}$ is not a telomere in ρ and $y_{t/h}$ is a telomere in ρ, otherwise the operation does not decrease $T(\rho, \pi)$. However, in the former case, the DCJ does not remove any gray edge: the cutting point at the telomere end does not correspond to any gray edge, and the inner breakpoint at $y_{t/h}$ does not induce a gray edge because $y_{t/h}$ is a telomere in ρ. Therefore, if a DCJ decreases $T(\rho, \pi)$, it neither increases $C(\rho, \pi)$ nor removes any loops.

Inverse fusion (fission): Splitting a chromosome can remove one gray edge, and therefore increase $C(\rho, \pi)$ by 1 or decrease $L(\rho, \pi)$ by 1. $T(\rho, \pi)$ decreases only if both new telomeres are also telomeres in ρ. In this case, no gray edge in the breakpoint

graph is removed, i.e., $C(\rho,\pi)$ and $L(\rho,\pi)$ remain unchanged. $T(\rho,\pi)$ is decreased by at most 2.

Inverse fission (fusion): Concatenating two chromosomes can decrease $T(\rho,\pi)$ by at most 2. This operation never removes a gray edge, thus $C(\rho,\pi)$ cannot be increased and $L(\rho,\pi)$ cannot be decreased.

Inverse tandem duplication: The observation from the previous section still holds, the operation can create one new component or remove one loop, but not both simultaneously. An inverse tandem duplication never changes the set of telomeres in π, and therefore cannot change $T(\rho,\pi)$.

Inverse deletion (insertion): The observation from the previous section still holds, the operation can create one new component or remove one loop, but not both simultaneously. An inverse deletion can only change $T(\rho,\pi)$ if the segment is inserted at a chromosome end. In this case, no gray edge is removed, i.e., $C(\rho,\pi)$ cannot be increased and $L(\rho,\pi)$ cannot be decreased. As the old chromosome end is no longer a telomere and one new telomere is added, $T(\rho,\pi)$ is decreased by at most 2.

Inverse chromosome duplication: This operation can decrease $T(\rho,\pi)$ by at most 2 (if the telomeres of the removed chromosome are not telomeres in ρ). Only gray edges with multiplicity ≥ 2 are removed, thus $C(\rho,\pi)$ and $L(\rho,\pi)$ remain unchanged.

Inverse chromosome deletion (chromosome insertion): This operation can decrease $T(\rho,\pi)$ by at most 2 (if the telomeres of the new chromosome are also telomeres in ρ). In the breakpoint graph, no gray edges are removed, i.e., $C(\rho,\pi)$ cannot be increased and $L(\rho,\pi)$ cannot be decreased.

Theorem 5.8. *A lower bound $lb_{mch}(\rho,\pi)$ for $d_{mch}(\rho,\pi)$ can be defined as follows.*

$$d_{mch}(\rho,\pi) \geq lb_{mch}(\rho,\pi),$$

$$\text{with } lb_{mch}(\rho,\pi) := n + ch(\rho) - C(\rho,\pi) + \left\lceil \frac{T(\rho,\pi)}{2} \right\rceil + \sum_{Components C_i} \left\lceil \frac{L_i}{2} \right\rceil$$

where L_i is the number of nodes with a loop in component C_i, and $ch(\rho)$ is the number of disjoint chromosomes in ρ.

Proof. An operation that decreases $T(\rho,\pi)$ will neither increase $C(\rho,\pi)$ nor decrease $L(\rho,\pi)$, therefore we can separate every sorting sequence into operations that decrease $CL(\rho,\pi) := n + ch(\rho) - C(\rho,\pi) + \sum_{Components C_i} \left\lceil \frac{L_i(\rho,\pi)}{2} \right\rceil$ and operations that decrease $T(\rho,\pi)$. If $\rho = \pi$, then $C(\rho,\pi) = n + ch(\rho)$ and $L(\rho,\pi) = 0$ (see Lemma 5.7), and therefore $CL(\rho,\pi) = 0$. As each operation decreases $CL(\rho,\pi)$ by at most one (the

proof from the previous section still holds), we need at least $CL(\rho, \pi)$ operations of the first kind. Furthermore, each operation decreases $T(\rho, \pi)$ by at most 2, so we need at least $\frac{T(\rho,\pi)}{2}$ operations of the second kind. Therefore, any sorting sequence must have at least $lb_{mch}(\rho, \pi)$ operations. □

Corollary 5.9. *If $\pi = \rho$, then $lb_{mch}(\rho, \pi) = 0$.*

Unfortunately, also in the multichromosomal case there are genomes $\pi \neq \rho$ with $lb_{mch}(\rho, \pi) = 0$, e.g., $\rho = \{(\overrightarrow{1}\ \overrightarrow{2}), (\overrightarrow{3}\ \overrightarrow{4})\}$ and $\pi = \{(\overrightarrow{1}\ \overrightarrow{2}\ \overrightarrow{3}\ \overrightarrow{4}), (\overrightarrow{1}\ \overrightarrow{2})\}$. Therefore, it is not sufficient to sort until the lower bound reaches 0, and again we have to define a *disruption measure* $\tau_{mch}(\rho, \pi)$. The intuition behind this disruption measure is similar to the one for the unichromosomal case. We use the following definitions.

$$ia(\rho, \pi) := 2 \cdot \text{number of inner adjacencies in } \pi \text{ w.r.t. } \rho$$

$$ta(\rho, \pi) := \text{number of telomere adjacencies in } \pi \text{ w.r.t. } \rho$$

$$m(\rho, \pi) := \sum_{\text{Element } x} |mult(x, \pi) - mult(x, \rho)|$$

$$\tau_{mch}(\rho, \pi) := ia(\rho, \rho) + ta(\rho, \rho) - ia(\rho, \pi) - ta(\rho, \pi) + 4 \cdot m(\rho, \pi)$$

Each inner adjacency $(x_{t/h}, y_{t/h})$ is weighted by 2, so both $x_{t/h}$ and $y_{t/h}$ can account 1 for $ia(\rho, \pi)$. A telomere adjacency consists just of a single extremity, which accounts 1 for $ta(\rho, \pi)$. $ia(\rho, \rho)$ and $ta(\rho, \rho)$ are a bias such that $\tau_{mch}(\rho, \pi) = 0$ if $\pi = \rho$. Removing a duplicated segment of length l can decrease $\tau_{mch}(\rho, \pi)$ by up to $l \cdot m(\rho, \pi)$, but it can also remove up to l inner adjacencies. To avoid the cancellation of the effects on $\tau_{mch}(\rho, \pi)$ if a duplicated segment with no breakpoints is removed, elements with incorrect multiplicity have twice the weight of an inner adjacency, which is analogous to the weights in the unichromosomal case. Therefore $m(\rho, \pi)$ is multiplied by 4. In our example, $ia(\rho, \rho) = 4$, $ta(\rho, \rho) = 4$, $ia(\rho, \pi) = 6$, $ta(\rho, \pi) = 4$, $m(\rho, \pi) = 2$, and therefore $\tau_{mch}(\rho, \pi) = 6$.

Lemma 5.10. *If $\rho = \pi$, then $\tau_{mch}(\rho, \pi) = 0$. Otherwise, $\tau_{mch}(\rho, \pi) > 0$.*

Proof. From the discussion above, it follows that $\tau_{mch}(\rho, \pi) \geq 0$, and the equality holds if $\rho = \pi$. In order to minimize $\tau_{mch}(\rho, \pi)$, it is necessary to minimize $m(\rho, \pi)$ and to maximize $ia(\rho, \pi)$ and $ta(\rho, \pi)$. $ia(\rho, \rho)$ and $ta(\rho, \rho)$ are independent of π and therefore fixed. Each extremity can account 1 for $ia(\rho, \pi)$ (if it is in an inner adjacency) or 1 for $ta(\rho, \pi)$ (if it is a telomere adjacency), therefore each element in π can account at most 2 for $ia(\rho, \pi) + ta(\rho, \pi)$, and this value is reached if and only if there is an adjacency on both sides of the element. Thus, the contribution to $\tau_{mch}(\rho, \pi)$ of all occurrences of an element x in π is minimized if $mult(x, \rho) = mult(x, \pi)$ and no extremity of x is part of any breakpoint. Every additional occurrence of x may increase $ia(\rho, \pi) + ta(\rho, \pi)$

by 2, but also increases $m(\rho, \pi)$ by 4 and therefore increases $\tau_{mch}(\rho, \pi)$ by at least 2. This means that $\tau_{mch}(\rho, \pi)$ is minimized if each element has the same multiplicity in ρ and π, and π contains neither inner breakpoints nor telomere breakpoints w.r.t. ρ. This is the case if and only if ρ and π are identical. In other words, if $\rho \neq \pi$, $\tau_{mch}(\rho, \pi)$ cannot be minimal, and therefore $\tau_{mch}(\rho, \pi) > 0$. □

5.2.4 The algorithm

The algorithm uses the same greedy strategy as in the unichromosomal case. In each step, it searches for all inverse operations that decrease the lower bound, i.e., they either increase $C(\rho, \pi)$, or decrease $L(\rho, \pi)$ or $T(\rho, \pi)$. From these operations, the one with the maximum *score* $\sigma_{mch}(\rho, \pi)$ is selected, where $\sigma_{mch}(\rho, \pi)$ is defined analogous to $\sigma_{uch}(\rho, \pi)$. If there is no such operation, we use additional heuristics to find operations that do not change the lower bound but have a positive score (i.e., $\sigma_{mch}(iop) > (0, 0)$). However, considering all possible cases is too complicated, therefore there is still the possibility that we do not find any operation with positive score. In this case, we use a fallback algorithm that is guaranteed to terminate.
For simplification of the presentation of the algorithm, we assume that all chromosomes of ρ consist of consecutive elements in Σ_n with positive orientation, i.e., they are of the form $(\overrightarrow{i}\,\overrightarrow{i+1}\ldots\overrightarrow{j-1}\,\overrightarrow{j})$.

Decreasing the lower bound

Finding operations that increase $C(\rho, \pi)$ can be done by finding 1-bridges and 2-bridges in the breakpoint graph and verifying additional preconditions, as shown in the last section. The only difference is that now there are DCJs that cut only one gray edge (or no gray edge at all). This is the case when the other cutting point is at the end of a chromosome, or at a breakpoint $(x_{t/h}, y_{t/h})$ where $x_{t/h}$ or $y_{t/h}$ is a telomere in ρ. Thus, we also have to consider DCJs that act only on one 1-bridge. Such a DCJ can be interpreted as a reversal, translocation, inverse fusion, or transposition. In the last case, we have to find a third cutting point in the same chromosome such that the resulting transposition still decreases the lower bound. Also finding operations that decrease $L(\rho, \pi)$ is straightforward and can be done as in the last section. The remaining task is to find operations that decrease $T(\rho, \pi)$. For this, we create a list of telomeres in π that are not telomeres in ρ, and another list of inner breakpoints in π where at least one of the adjacent extremities is a telomere in ρ. Operations that decrease $T(\rho, \pi)$ must act on one or two breakpoints of these lists, depending on the operation type. Creating the lists can be done by a linear scan over π, therefore all operations that decrease $T(\rho, \pi)$ can be found in quadratic time. The only exceptions are inverse deletions and inverse chromosome deletions, which may add segments of arbitrary content. As in the unichromosomal case, practical tests have shown that it is

better to only allow the insertion of segments without any breakpoints. This does not only lead to better sorting sequences, but also keeps the time complexity of finding the operations in $O(l(\pi)^2)$, where $l(\pi)$ is the number of elements in π.

Additional operations

If there is no inverse operation that decreases $lb_{mch}(\rho, \pi)$, we may still find inverse operations that do not change the lower bound but decrease $\tau_{mch}(\rho, \pi)$. Searching for all these operations would exceed our computing capacity, so we just search for the following subset of these operations that can be found easily.

- There are inverse tandem duplications and transposition duplications that do not change $lb_{mch}(\rho, \pi)$, but decrease $\tau_{mch}(\rho, \pi)$. We therefore search for identical consecutive segments that are maximal, i.e., they cannot be extended in any direction, and check the effect on $lb_{mch}(\rho, \pi)$ and $\tau_{mch}(\rho, \pi)$ if we remove one of them. This operation corresponds either to an inverse tandem duplication, or to an inverse transposition duplication, which can be simulated by a reversal or transposition and an inverse tandem duplication.
- Depending on the telomeres of a chromosome, the lower bound can remain unchanged during an inverse chromosome duplication, but $\tau_{mch}(\rho, \pi)$ can decrease. We therefore search for identical chromosomes and check the score of removing one of them.
- Inserting a segment of consecutive elements x with $mult(x, \rho) > mult(x, \pi)$ decreases $\tau_{mch}(\rho, \pi)$. We search for such segments of maximal length and try to insert them by an inverse deletion. Note that this is not always possible as this operation can increase the lower bound by merging two components.
- Creating inner or telomere adjacencies never increases the lower bound, but decreases $\tau_{mch}(\rho, \pi)$. We therefore search for DCJs and inverse fissions that create new adjacencies without splitting old ones.

The fallback algorithm

It is possible that neither there is an operation that decreases $lb_{mch}(\rho, \pi)$, nor do we find an operation that decreases $\tau_{mch}(\rho, \pi)$, so the main algorithm gets stuck. However, this case cannot occur if all elements have the same multiplicity in ρ and in π.

Lemma 5.11. *If $\pi \neq \rho$ and $mult(x, \pi) = mult(x, \rho)$ holds for all elements x, then there is an inverse operation with positive score.*

Proof. When the preconditions are fulfilled, there must be at least one breakpoint in π w.r.t. ρ. We have to distinguish three cases. (1) This is a telomere breakpoint. W.l.o.g.

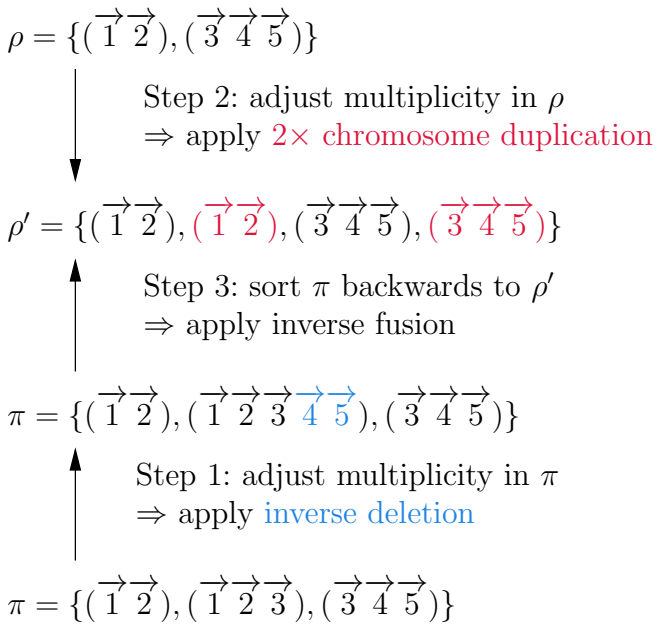


Figure 5.8: An example of the fallback algorithm. Note that this is a simplified example, in which the algorithm would find an operation with positive score, and not use the fallback algorithm.

a chromosome in π ends with x_h, but x_h is not a telomere in ρ. Then, $mult(x, \rho) = mult(x + 1, \rho)$ (as they are in the same chromosome), and therefore there must be another breakpoint including $(x+1)_t$. An operation that creates an adjacency between x_h and $(x+1)_t$ does not increase the lower bound, but decrease $\tau_{mch}(\rho, \pi)$ by at least 2. (2) The breakpoint is an inner breakpoint between two extremities that are telomeres in ρ. In this case, cutting the chromosome at the breakpoint with an inverse fusion does not increase the lower bound but decreases $\tau_{mch}(\rho, \pi)$ by 2, because it creates two telomere adjacencies. (3) The breakpoint is an inner breakpoint, and at least one of the adjacent extremities is not a telomere in ρ. W.l.o.g., the breakpoint is of the form (x_h, y_h), and x_h is not a telomere in ρ. Then, $mult(x, \rho) = mult(x + 1, \rho)$, thus there must be another breakpoint including $(x + 1)_t$. An operation that creates an adjacency between x_h and $(x + 1)_t$ does not increase the lower bound, but decreases $\tau_{mch}(\rho, \pi)$ by at least 2. □

The fallback algorithm first ensures that the precondition of the lemma holds. For each chromosome ρ^i in ρ, we determine the element x in ρ^i with the most occurrences in π. We then create maximal segments of consecutive elements $\overrightarrow{y}\ \overrightarrow{y+1}\ \ldots$ such

Algorithm 5.2 Heuristic algorithm for finding a sorting sequence of ρ w.r.t. π (multichromosomal case)

```
1: function findSequence(ρ, π)
2:   while (lb_mch(ρ, π), τ_mch(ρ, π)) ≠ (0, 0) do
3:     find all inverse operations that decrease lb_mch(ρ, π)
4:     if no inverse operation found then
5:       find inverse tandem duplications
6:       find inverse transposition duplications
7:       find inverse deletions of consecutive segments with mult(x, π) < mult(x, ρ)
8:       find DCJs that create adjacencies
9:     if inverse operation found then
10:      apply inverse operation with maximal score
11:    else
12:      use fallback algorithm
13:  invert and output the sorting sequence
```

that each element z in the segment belongs to ρ^i and $mult(z, \pi) < mult(x, \pi)$, and add this segment by an inverse deletion to π. Note that this can be done without creating new breakpoints. This step is repeated until all elements belonging to the same component in ρ have the same multiplicity in π. We then transform ρ into ρ' by applying chromosome duplications and chromosome deletions on ρ such that for each element x, $mult(x, \rho') = mult(x, \pi)$. Now, we apply our normal algorithm to sort π into ρ'. To ensure that the precondition of Lemma 5.11 is always fulfilled, we forbid operations that create or delete elements, i.e., any kind of duplication or deletion. Due to Lemma 5.11, the algorithm will find a sorting sequence that transforms π into ρ'. An example is depicted in Fig. 5.8. The whole algorithm in pseudocode is shown in Algorithm 5.2.

5.3 Experimental results

We tested both the unichromosomal and the multichromosomal algorithm on artificial data. The multichromosomal algorithm was also tested on cancer karyotypes from the "Mitelman Database of Chromosome Aberrations and Gene Fusions in Cancer" [MJM10].

5.3.1 Creating artificial data

The artificial data for testing the unichromosomal algorithm was created by applying random sequences of operations with an overall weight of αn on the identity genome of size n. The operations of the sequences are independently distributed, with tandem duplications and deletions having a probability of $\frac{1}{3}$, reversals having a probability of

$\frac{2}{9}$, and block interchanges having a probability of $\frac{1}{9}$ (thus the expected numbers of DCJs, tandem duplications, and deletions are equal). Once the type of an operation was determined, the operation was selected uniformly distributed among all operations of this type. As long deletions can cancel the effects of previous operations, deletions were restricted to have a length of at most 0.1 times the current genome length. To keep the size of the genome approximately constant, also tandem duplications were restricted to have a length of at most 0.1 times the current genome length.We created test cases for different values of n and α, namely $n \in \{20, 50, 80, 100\}$ and α from 0.1 to 1 in steps of 0.1. For each combination of the parameters n and α, we created 100 test cases.

The test cases for the multichromosomal algorithm were created in the same way. However, we now started with genomes with n different elements and c different chromosomes. Each chromosome has the same size, the ploidy (i.e., the number of identical copies) of the chromosomes is 1 or 2. When creating the sequences of operations, each type of operation had the same probability. Although these probabilities do not match the biological reality, this is still convenient to assess the performance of the algorithm. The parameter α which controls the weight of the sequences again was varied from 0.1 to 1 in steps of 0.1, the parameters n and c as well as the ploidy of the chromosomes were chosen such that they reflect the properties of biologically meaningful datasets.

To understand what "biologically meaningful" means, let us have a brief look on biological datasets. In most of them, elements do not represent single genes but *conserved segments* [NT84] or *synteny blocks* [PT03], i.e., regions of a chromosome that are highly conserved and do not contain breakpoints. These synteny blocks normally contain several genes. The amount n of different synteny blocks depends on the allowed dissimilarity between the synteny blocks as well as on the evolutionary distance between the genomes. For example, El-Mabrouk et al. [EMNS98] tested their algorithm on yeast genomes with 55 synteny blocks, Zheng et al. [ZZS06] identified 34 synteny blocks between rice, sorghum, and maize. Salse et al. [SPCD02] used 60 synteny blocks to compare *Arabidopsis thaliana* and rice. A recent comprehensive study of *Drosophila* genomes [Dro07] identified between 112 and 1406 synteny blocks, depending on the evolutionary distance of the species.

Our datasets reflect those parameters. Dataset 1 contains 16 chromosomes of ploidy 2 with a total of 64 elements, this approximately matches the yeast genome. Dataset 2 contains 12 chromosomes of ploidy 2 with a total of 36 elements, Dataset 3 contains 5 chromosomes of ploidy 2 with a total of 60 elements. These are realistic values for plant genomes. Dataset 4 contains 5 different chromosomes with a total of 200 elements, two of them with ploidy 1 and three of them with ploidy 2 (corresponding to two sex chromosomes and three diploid chromosomes). This reflects the values for closely related *Drosophila* species. Each dataset contains 100 different test cases for each choice of the parameter α. This approach allows us to evaluate our algorithm on a large set

of test cases and for many different evolutionary distances, and gives us much more robust result than just testing on a few biological datasets.

5.3.2 Results on artificial data

The results of the experiments using the algorithm for unichromosomal genomes are depicted in Figs. 5.9 to 5.12. Each figure shows

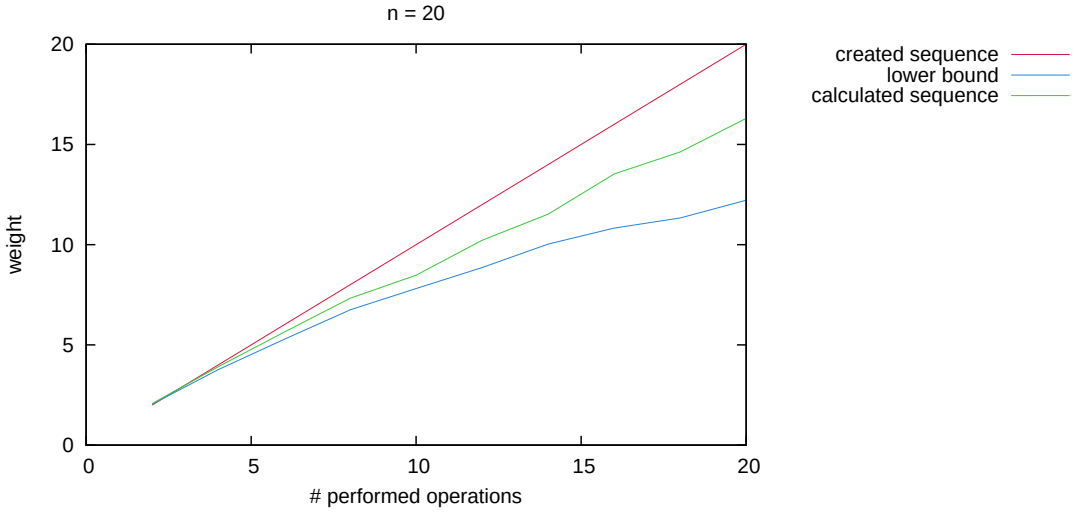
(a) the relation of the weight of the sequence used to create the test case (also known as the "true evolutionary distance" w_{true}), the lower bound lb_{uch}, and the weight of the sequence calculated by our algorithm (denoted by w_{calc}), as well as

(b) the relative frequency of the different types of operations found by our algorithm.

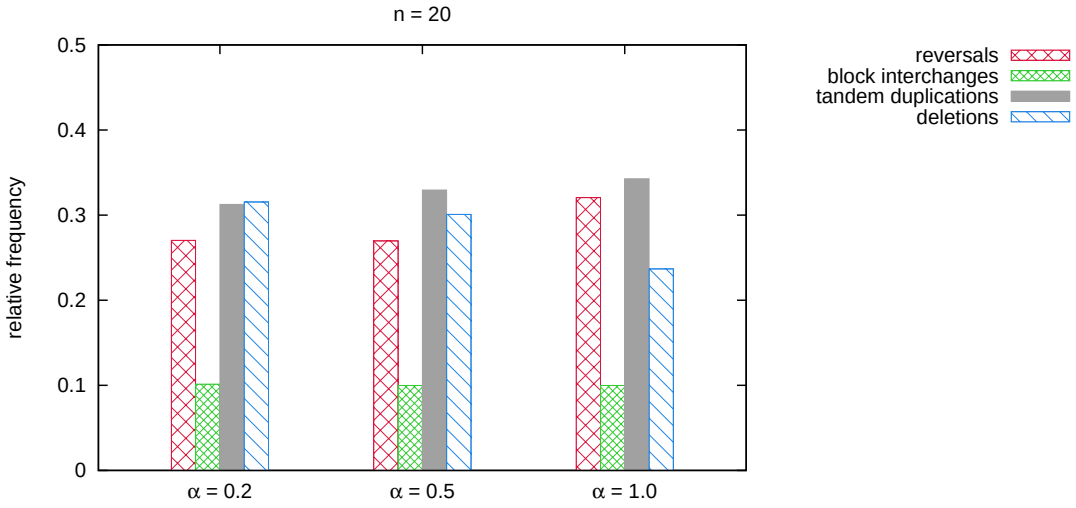
Each value is the average of all 100 test cases created with the same parameters.
The results show that, as long as w_{true} is close to the lower bound lb_{uch}, also w_{calc} is close to the lower bound, and on average less than w_{true}. As the coherence of lb_{uch} and w_{true} lessens for increasing values of n and α, the weights of the calculated sequences increase. Nevertheless, even for higher values of n and α, w_{calc} is still a good approximation of w_{true}. The analysis of the frequency of the different types of operations shows that at $\alpha = 0.2$, the frequencies are close to those used to create the test cases, only the number of reversals is a little bit overestimated (recall that when creating the test cases, the ratio of the operations was reversals : block interchanges : tandem duplications : deletions $= \frac{2}{9} : \frac{1}{9} : \frac{1}{3} : \frac{1}{3}$). With increasing values of α, the algorithm tends to overestimate the number of reversal and tandem duplications, and underestimates the number of block interchanges and deletions. This effect increases for increasing values of n.
The results of the algorithm for multichromosomal genomes are depicted in Figs. 5.13 to 5.16. Again, the true evolutionary distance and the distances calculated by our algorithm are close together. In the fourth diagram, an additional saturation effect can be observed, i.e., we can find a sorting sequence with weight $w_{algo} \lessapprox 120$ for most genomes. The analysis of the frequencies of the different types of operations shows that the algorithm tends to use many translocations and tandem duplications, while fusions and chromosome duplications are rarely used. Again, this effect increases for increasing values of α and n.

5.3.3 Evaluating the algorithm on cancer karyotypes

The "Mitelman Database of Chromosome Aberrations and Gene Fusions in Cancer" [MJM10] contains the descriptions of cancer karyotypes which have been manually collected from publications over the last twenty years. For our experiments, we used the version of May 14, 2009, which contains 56428 datasets. The data is represented in the ISCN format, which can be parsed by the software tool `CyDAS` [HBBR05]. From all

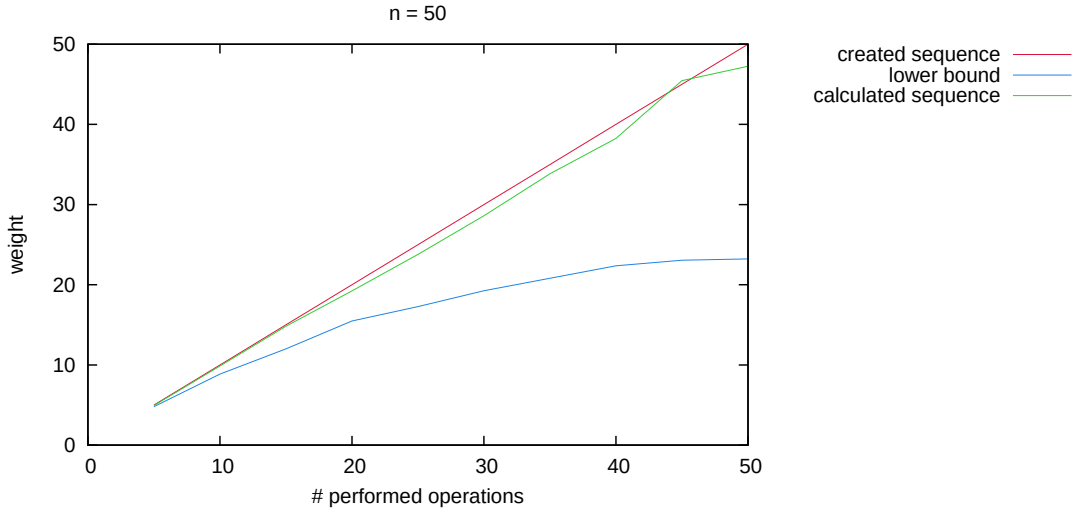


(a)

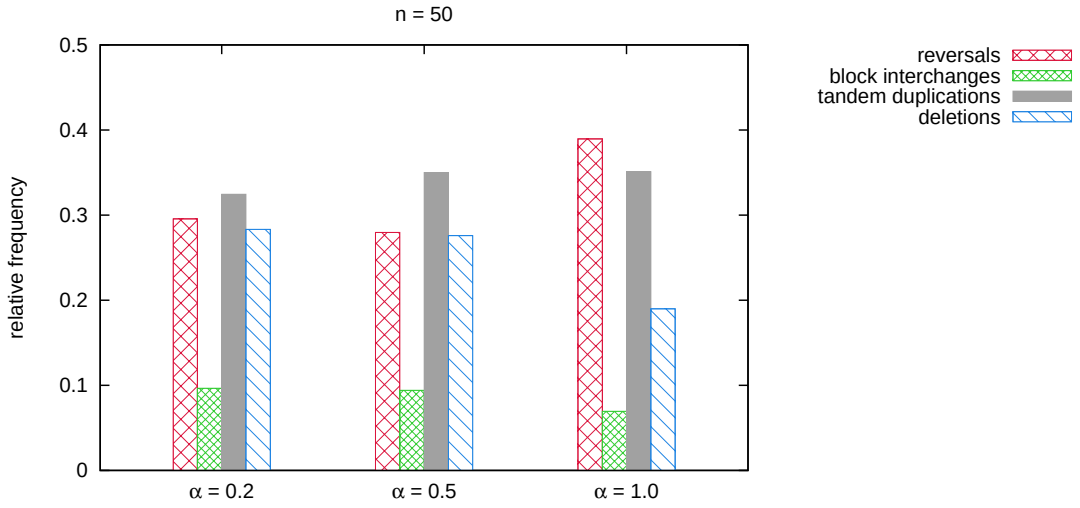


(b)

Figure 5.9: (a) Performance of the algorithm for unichromosomal genomes on the dataset with $n = 20$. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

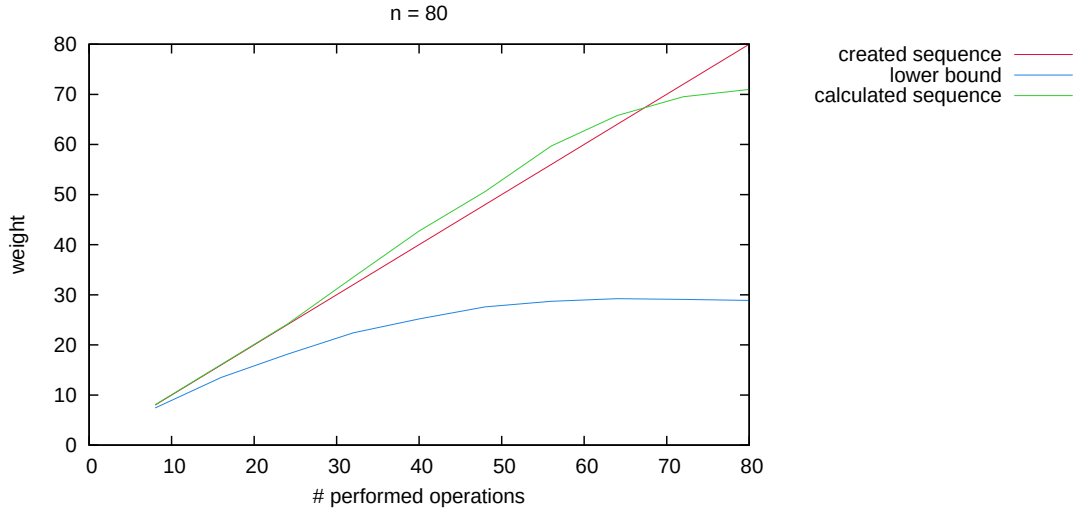


(a)

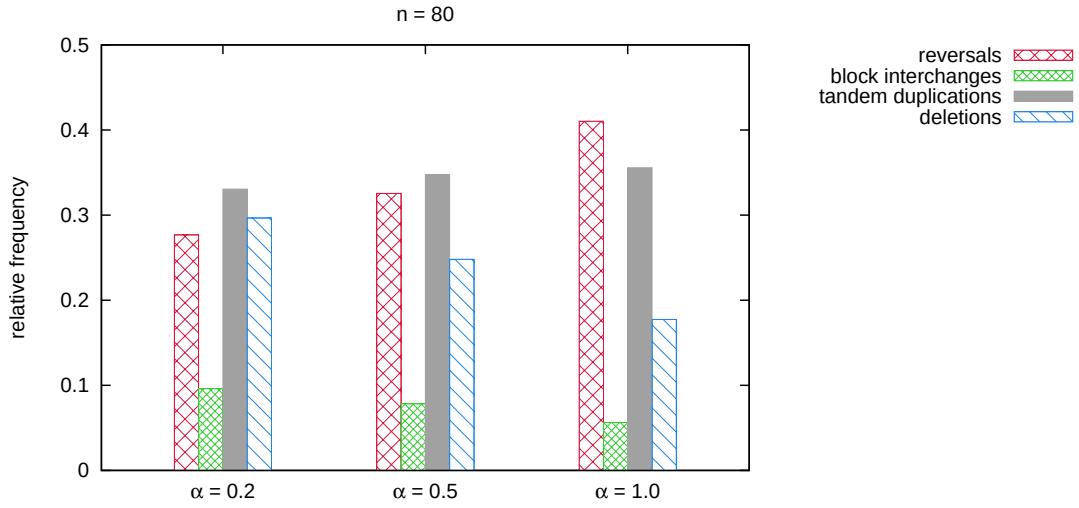


(b)

Figure 5.10: (a) Performance of the algorithm for unichromosomal genomes on the dataset with $n = 50$. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

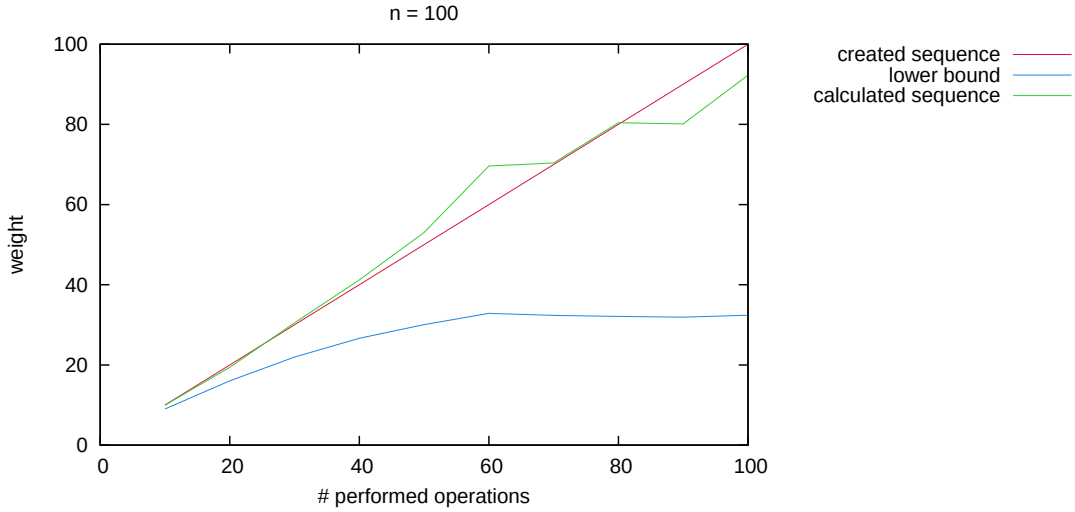


(a)

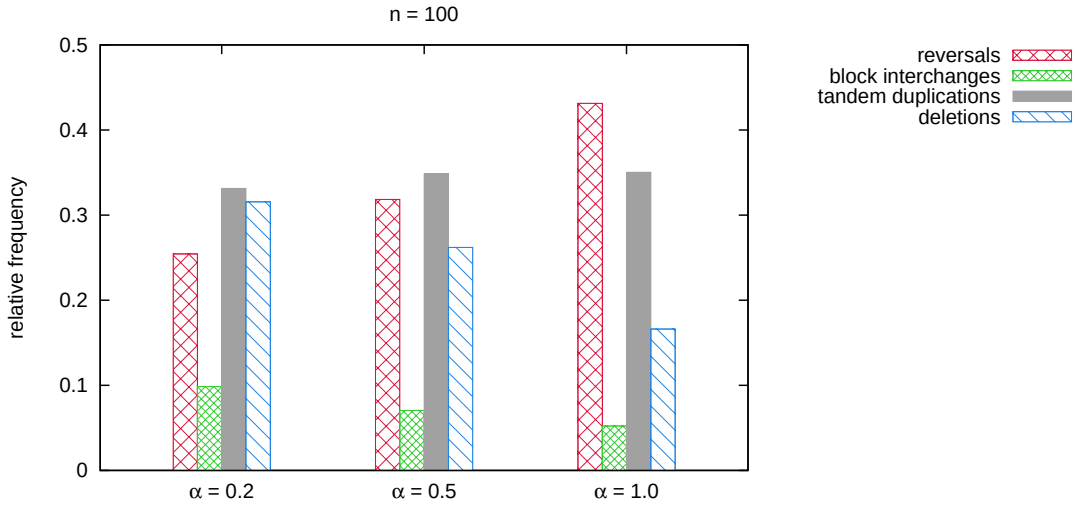


(b)

Figure 5.11: (a) Performance of the algorithm for unichromosomal genomes on the dataset with $n = 80$. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

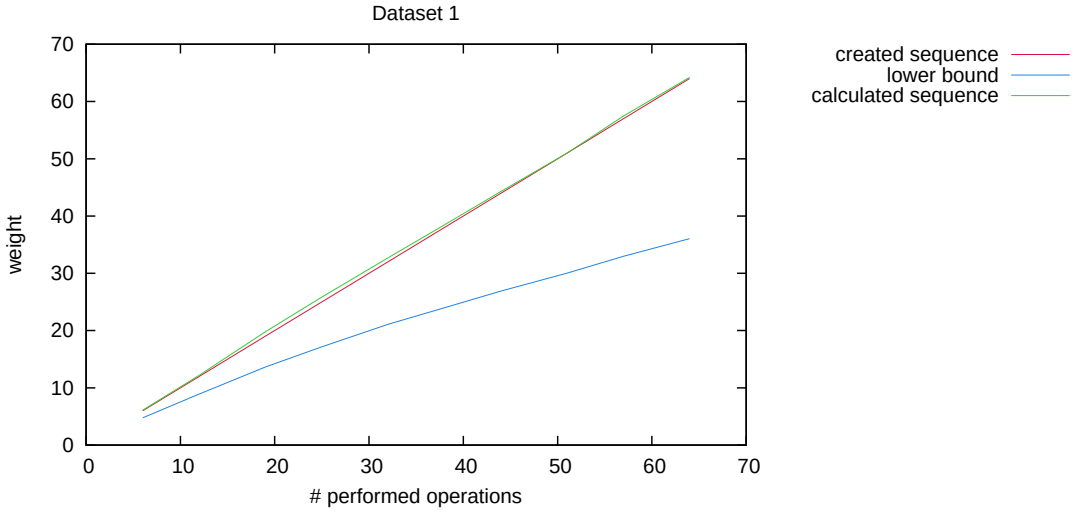


(a)

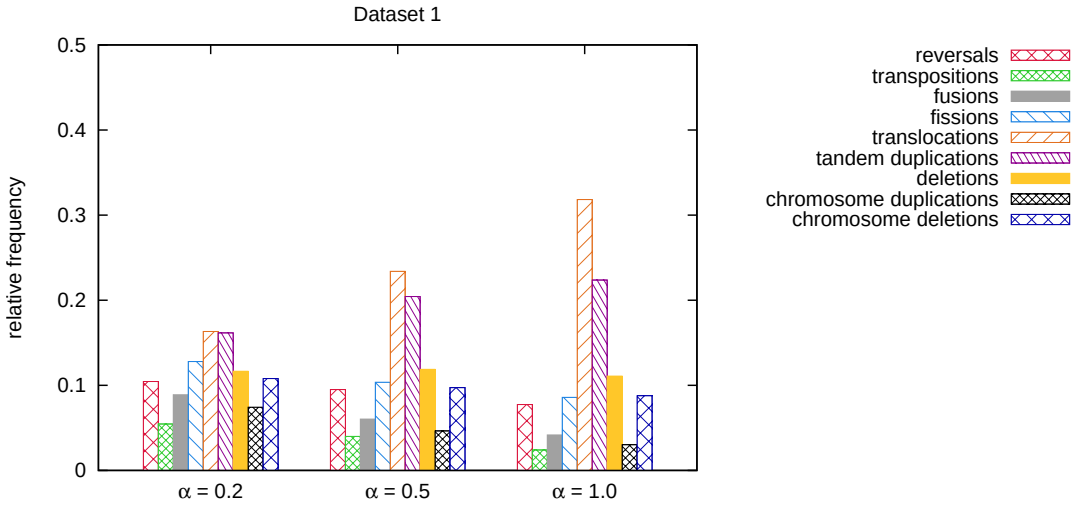


(b)

Figure 5.12: (a) Performance of the algorithm for unichromosomal genomes on the dataset with $n = 100$. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

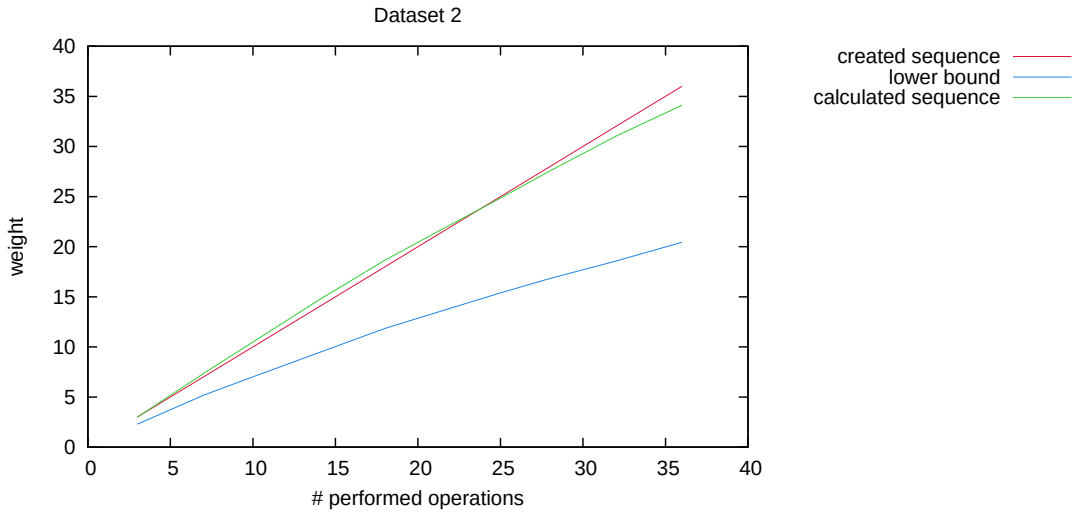


(a)

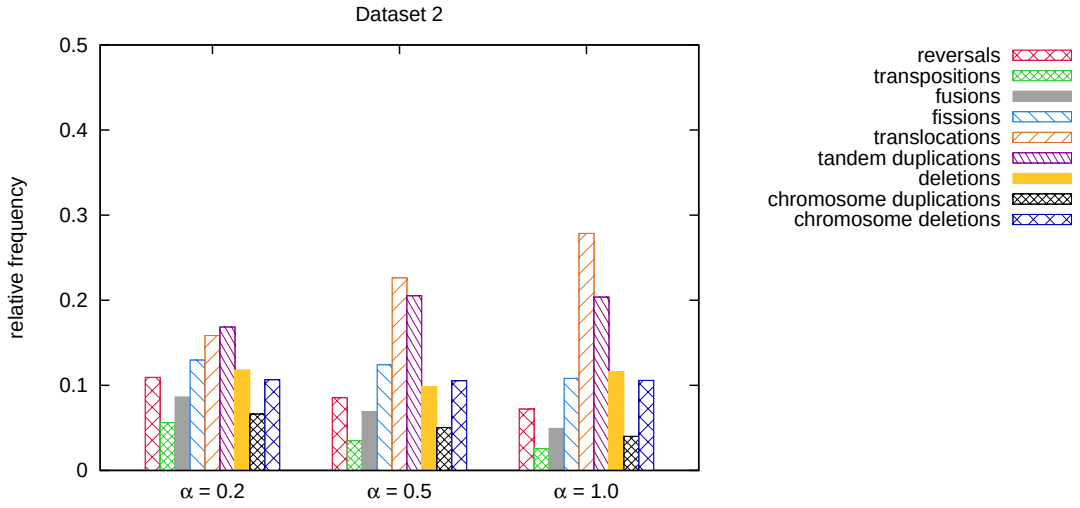


(b)

Figure 5.13: (a) Performance of the algorithm for multichromosomal genomes on Dataset 1. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

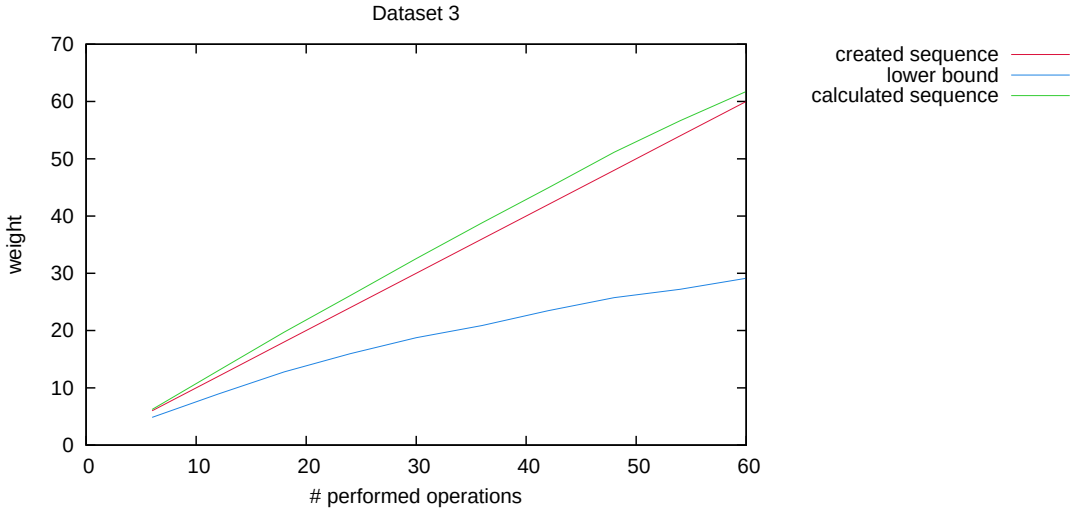


(a)

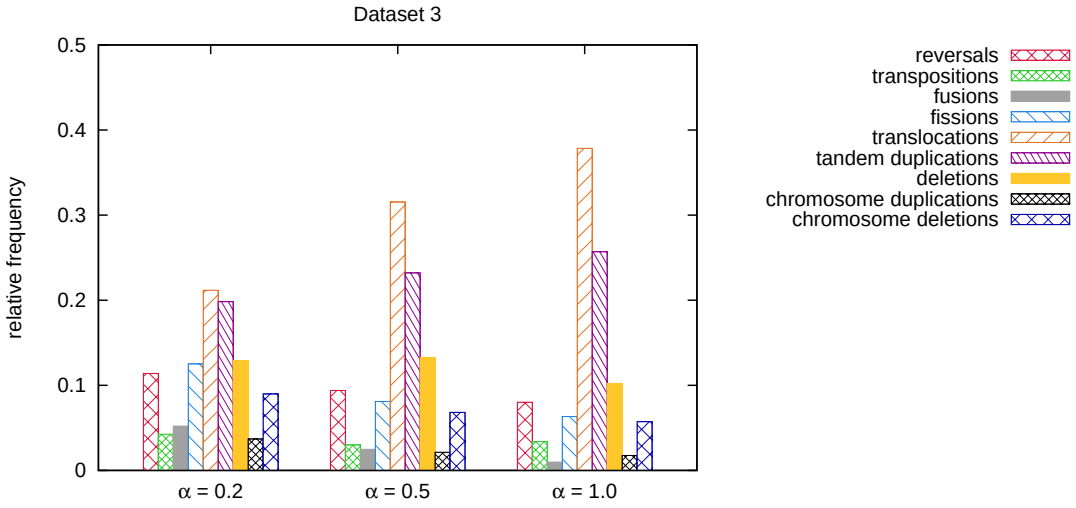


(b)

Figure 5.14: (a) Performance of the algorithm for multichromosomal genomes on Dataset 2. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

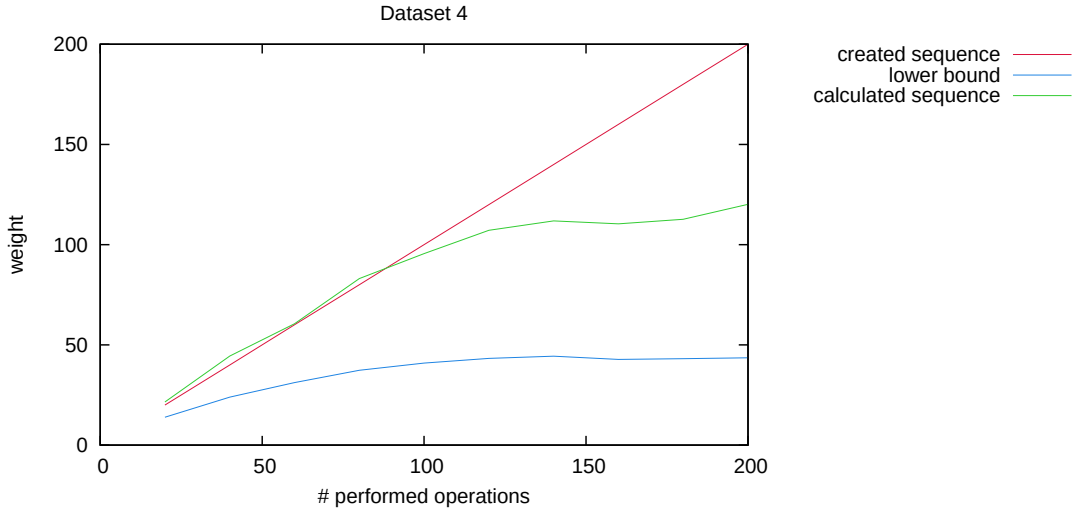


(a)

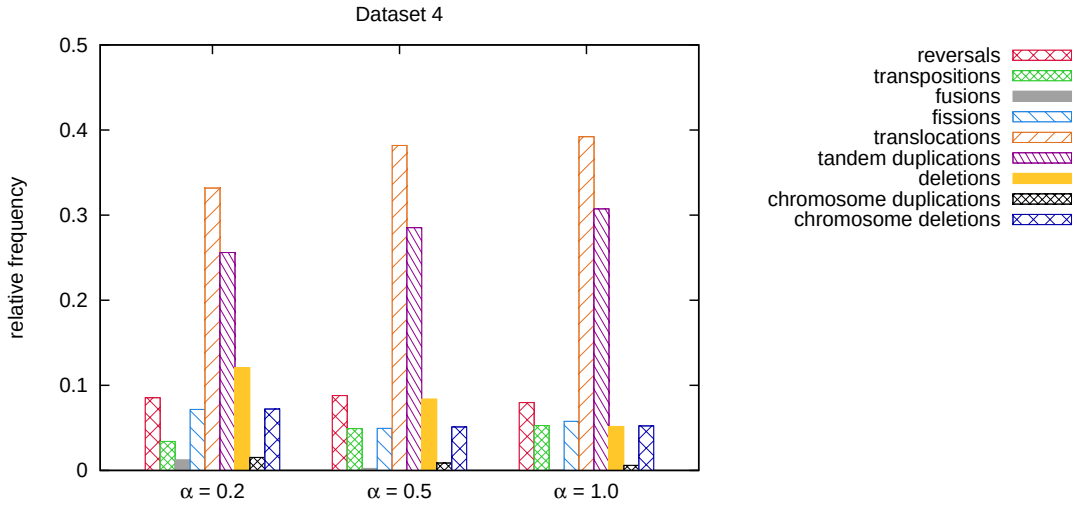


(b)

Figure 5.15: (a) Performance of the algorithm for multichromosomal genomes on Dataset 3. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.



(a)



(b)

Figure 5.16: (a) Performance of the algorithm for multichromosomal genomes on Dataset 4. (b) Relative frequencies of reconstructed operations on this dataset for three different values of α.

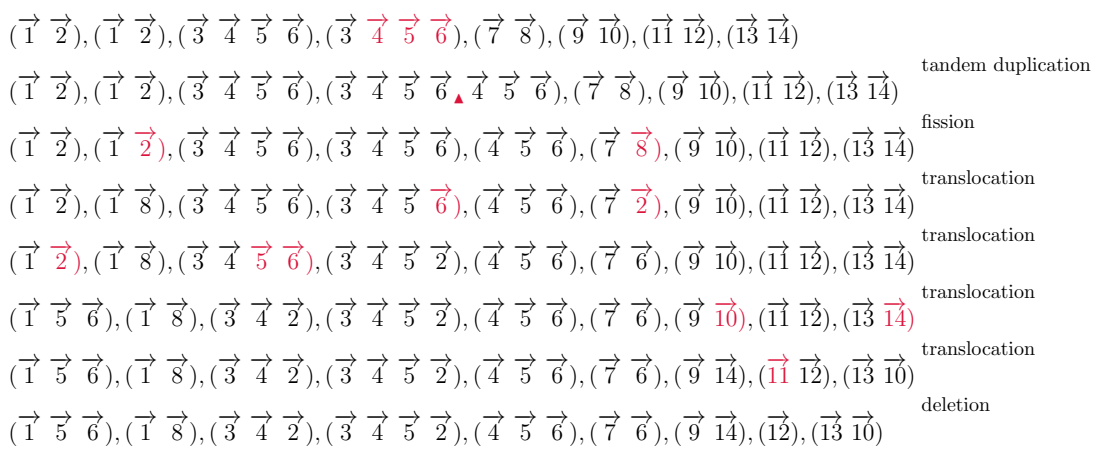


Figure 5.17: Sorting scenario for a cancer karyotype reported in [CPT+88]. For better readability, all chromosomes identical in ρ and π are omitted.

datasets which could be parsed by `CyDAS` without error (44064 datasets), all unknown elements were removed and all segments without breakpoint were compressed, i.e., if a set of consecutive elements contains no breakpoint in any chromosome, it can be represented as one element. The resulting datasets were used as input to our algorithm. Most of the scenarios are rather easy to reconstruct, the average lower bound is 2.63 and the average calculated weight is 4.08. However, there are some more complicated karyotypes, with rearrangement scenarios of over 50 operations. Exemplarily, the reconstructed scenario for one karyotype is shown in Fig. 5.17. This karyotype was reported in [CPT+88], and can be described by the ISCN formula[1]

$$47, XY, t(3;7)(q23;q22), t(3;7;9)(q23;q32;q22), +i(7)(q10), t(14;18)(q32;q21), del(17)(p11)$$

In principle, our algorithm correctly identified all operations. The triple translocation $t(3;7;9)(q23;q32;q22)$ and the new chromosome $+i(7)(q10)$ are not allowed operations in our model. Our algorithm replaced the triple translocation by two translocations, and the new chromosome by a tandem duplication with a subsequent fission, which are the best possible explanations within our model.

5.4 Conclusion and Discussion

We have developed algorithms for two genome rearrangement problems which consider a large variety of operations, including tandem duplications and deletions of segments of arbitrary size. Both algorithms are guaranteed to terminate, i.e., they find a sorting

[1] For details about the ISCN format, see [Mit95].

sequence for any possible pair of input genomes.
Genome rearrangement algorithms that also consider duplicated gene content is a rather new field of research, and although our results are promising, the algorithms should be seen as small steps towards an algorithm that produces biologically reliable results. We will now discuss some ideas which might improve the algorithms in future work.
One of the main drawbacks of the algorithms is that weights are chosen due to a mathematical model and do not reflect the biological reality. This leaves room for further investigations. For example, the algorithm could be improved by giving unwanted operations a larger weight or completely omit them. While adapting the theoretical model to other weights seems to be the obvious way to improve the algorithm, it might also be worth to examine how robust the results are w.r.t. the chosen weights. In other words, does the optimal rearrangement scenario change when we use other weights? Some observations in the genome can be explained best by a specific operation (e.g., a duplicated chromosome is most likely caused by a single chromosome duplication), no matter how this operation is weighted. Such observations are predominant in closely related genomes, and the corresponding operations can be reconstructed even with a wrong weighting scheme. In more diverged genomes, there are often different possible rearrangement scenarios, and the weighting scheme matters. Thus, further investigations should examine what the "critical distance" between two genomes is, i.e., up to which distance the optimal rearrangement scenario is mostly robust w.r.t. the weighting scheme.
Furthermore, there are many other possible improvements, both of theoretical and practical nature. For example, a tighter lower bound for the distances or even an upper bound would give us new insight into the problem. Also changing the model such that there are no restrictions to the ancestral genome are of great interest. A more practical improvement would be an improved heuristic or a postprocessing of the sorting sequence that removes redundant operations (like operations whose effects are cancelled by a subsequent deletion).

List of Tables

List of Figures

List of Algorithms

List of Abbreviations

BBS tree	balanced binary search tree
DCJ	double cut and join
DNA	deoxyribonucleic acid
ECD	Eulerian cycle decomposition problem
MB graph	multiple breakpoint graph
Mbp	million base pairs
mdECD	marked directed Eulerian cycle decomposition problem
oCMP	odd cycle median problem
RMP	reversal median problem
SBDCJ	sorting by double cut and join
SBR	sorting by reversals
SBT	sorting by transpositions
SBwRT	sorting by weighted reversals and transpositions
TM$\mathfrak{S}$	transposition median problem on the symmetric group $\mathfrak{S}_n$
TMP	transposition median problem
wCMP	weighted cycle median problem
wRTMP	weighted reversal and transposition median problem

Bibliography

[ACG+99] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Combinatorial optimization problems and their approximability properties*. Complexity and Approximation. Springer-Verlag, 1999.

[AHK+07] A. Amir, T. Hartman, O. Kapah, A. Levy, and E. Porat. On the cost of interchange rearrangement in strings. In *Proc. 15th Annual European Symposium on Algorithms*, volume 4698 of *Lecture Notes in Computer Science*, pages 99–110. Springer-Verlag, 2007.

[AS08] Z. Adam and D. Sankoff. The ABCs of MGR with DCJ. *Evolutionary Bioinformatics*, 4:69–74, 2008.

[AT07] W. Arndt and J. Tang. Improving inversion median computation using commuting reversals and cycle information. In *Proc. 5th Annual RECOMB Satellite Workshop on Comparative Genomics*, volume 4751 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 2007.

[Bad05] M. Bader. Sorting by weighted transpositions and reversals. Diploma thesis, Ulm University, 2005.

[BAO08] M. Bader, M.I. Abouelhoda, and E. Ohlebusch. A fast algorithm for the multiple genome rearrangement problem with weighted reversals and transpositions. *BMC Bioinformatics*, 9:516, 2008.

[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.

[BBG+00] G. Blanc, A. Barakat, R. Guyot, R. Cooke, and M. Delseny. Extensive duplication and reshuffling in the arabidopsis genome. *The Plant Cell*, 12:1093–1101, 2000.

[BCF04] G. Blin, C. Chauve, and G. Fertin. The breakpoint distance for signed sequences. In *Proc. 1st International Conference on Algorithms and Computational Methods for Biochemical and Evolutionary Networks*, pages 3–16, 2004.

[Ber07] M. Bernt. personal communication, 2007.

[Ber10] M. Bernt. *Gene order rearrangement methods for the reconstruction of phylogeny*. PhD thesis, Universität Leipzig, 2010.

[BH96] P. Berman and S. Hannenhalli. Fast sorting by reversals. In *Proc. 7th Symposium on Combinatorial Pattern Matching*, volume 1075 of *Lecture Notes in Computer Science*, pages 168–185. Springer-Verlag, 1996.

[BK99] P. Berman and M. Karpinski. On some tighter inapproximability results. Technical Report 23, DIMACS, 1999.

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

[BKS99] M. Blanchette, T. Kunisawa, and D. Sankoff. Gene order breakpoint evidence in animal mitochondrial phylogeny. *Journal of Molecular Evolution*, 49(2):193–203, 1999.

[BLEMG06] D. Bertrand, M. Lajoie, N. El-Mabrouk, and O. Gascuel. Evolution of tandemly repeated sequences through duplication and inversion. In *Proc. 4th RECOMB Comparative Genomics Satellite Workshop*, volume 4205 of *Lecture Notes in Computer Science*, pages 129–140. Springer-Verlag, 2006.

[BMM07] M. Bernt, D. Merkle, and M. Middendorf. Using median sets for inferring phylogenetic trees. *Bioinformatics*, 23:e129–e135, 2007.

[BMS04] A. Bergeron, J. Mixtacki, and J. Stoye. Reversal distance without hurdles and fortresses. In *Proc. 15th Annual Symposium on Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 388–399. Springer-Verlag, 2004.

[BMS06] A. Bergeron, J. Mixtacki, and J. Stoye. A unifying view of genome rearrangements. In *Proc. 6th Workshop on Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 163–173. Springer-Verlag, 2006.

[BMS09] A. Bergeron, J. Mixtacki, and J. Stoye. A new linear time algorithm to compute the genomic distance via the double cut and join distance. *Theoretical Computer Science*, 410(51):5300–5316, 2009.

[BMY01] D.A. Bader, B.M.E. Moret, and M. Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8:483–491, 2001.

[BO07] M. Bader and E. Ohlebusch. Sorting by weighted reversals, transpositions, and inverted transpositions. *Journal of Computational Biology*, 14(5):615–636, 2007.

[BP93] V. Bafna and P.A. Pevzner. Genome rearrangements and sorting by reversals. In *Proc. 34th IEEE Symposium on Foundations of Computer Science*, pages 148–157, 1993.

[BP96] V. Bafna and P.A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[BP98] V. Bafna and P.A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998.

[BP02] B. Bourque and P.A. Pevzner. Genome-scale evolution: Reconstructing gene orders in the ancestral species. *Genome Research*, 12(1):26–36, 2002.

[Bro02] T.A. Brown. *Genomes*. BIOS Scientific Publishers Ltd, second edition, 2002.

[Bry00] D. Bryant. The complexity of calculating exemplar distances. In *Proc. Workshop on Gene Order Dynamics, Comparative Maps, and Multigene Families*, pages 207–211. Kluwer Academic Publishers, 2000.

[Cap99] A. Caprara. On the tightness of the alternating-cycle lower bound for sorting by reversals. *Journal of Combinatorial Optimization*, 3:149–182, 1999.

[Cap03] A. Caprara. The reversal median problem. *INFORMS Journal on Computing*, 15(1):93–113, 2003.

[Chr98] D.A. Christie. *Genome Rearrangement Problems*. PhD thesis, University of Glasgow, 1998.

[CJM+00a] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.-S. Wang, T. Warnow, and S. Wyman. An empirical comparison between BPAnalysis and MPBE on the campanulaceae chloroplast genome dataset. In *Proc.*

Workshop on Gene Order Dynamics, Comparative Maps, and Multigene Families, pages 99–121. Kluwer Academic Publishers, 2000.

[CJM+00b] M.E. Cosner, R.K. Jansen, B.M.E. Moret, L.A. Raubeson, L.-S. Wang, T. Warnow, and S. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proc. 8th International Conference on Intelligent Systems for Molecular Biology*, pages 104–115. AAAI Press, 2000.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, second edition, 2001.

[CPT+88] F. Cabanillas, S. Pathak, J. Trujillo, J. Manning, R. Katz, P. McLaughlin, W.S. Welasquez, F.B. Hagemeister, A. Goodacre, A. Cork, J.J. Butler, and E.J. Freireich. Frequent nonrandom chromosome abnormalities in 27 patients with untreated large cell lymphoma and immunoblastic lymphoma. *Cancer Research*, 48:5557–5564, 1988.

[CPvV+00] S. Casjens, N. Palmer, R. van Vugt, W.M. Huang, B. Stevenson, P. Rosa, R. Lathigra, G. Sutton, J. Peterson, R.J. Dodson, D. Haft, E. Hickey, M. Gwinn, O. White, and C.M. Fraser. A bacterial genome in flux: the twelve linear and nine circular extrachromosomal DNAs in an infectious isolate of the lyme disease spirochete borrelia burgdorferi. *Molecular Microbiology*, 35(3):490–516, 2000.

[Cra72] C.A. Crane. *Linear lists and priority queues as balanced binary trees*. PhD thesis, Stanford University, 1972.

[CZF+05] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, and T. Jiang. The assignment of orthologous genes via genome rearrangement. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(4):302–315, 2005.

[DEEA02] D.A. Dalevi, N. Eriksen, K. Eriksson, and S.G.E Andersson. Measuring genome divergence in bacteria: A case study using chlamydia data. *Journal of Molecular Evolution*, 55:24–36, 2002.

[Dro07] Drosophila 12 Genomes Consortium. Evolution of genes and genomes on the drosophila phylogeny. *Nature*, 450:203–218, 2007.

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

[EH06] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(4):369–379, 2006.

[EM01] N. El-Mabrouk. Sorting signed permutations by reversals and insertions/deletions of contiguous segments. *Journal of Discrete Algorithms*, 1(1):105–122, 2001.

[EM02] N. El-Mabrouk. Reconstructing an ancestral genome using minimum segments duplications and reversals. *Journal of Computer and System Sciences*, 65:442–464, 2002.

[EMNS98] N El-Mabrouk, J. Nadeau, and D. Sankoff. Genome halving. In *Proc. 9th Annual Symposium on Combinatorial Pattern Matching*, volume 1448 of *Lecture Notes in Bioinformatics*, pages 235–250. Springer-Verlag, 1998.

[Eri02] N. Eriksen. $(1 + \varepsilon)$-approximation of sorting by reversals and transpositions. *Theoretical Computer Science*, 289(1):517–529, 2002.

[Eri03] N. Eriksen. *Combinatorial methods in comparative genomics*. PhD thesis, Royal Institute of Technology, Stockholm, 2003.

[Eri07] N. Eriksen. Reversal and transposition medians. *Theoretical Computer Science*, 374:111–126, 2007.

[Eri09] N. Eriksen. Median clouds and a fast transposition median solver. In *Proc. 21st International Conference on Formal Power Series and Algebraic Combinatorics*, pages 373–384, 2009.

[FCV+06] Z. Fu, X. Chen, V. Vacic, P. Nan, Y. Zhong, and T. Jiang. A parsiomony approach to genome-wide ortholog assignment. In *Proc. 10th Annual International Conference on Research in Computational Molecular Biology*, Lecture Notes in Computer Science, pages 578–594. Springer-Verlag, 2006.

[FSS06] G. Fritzsch, M. Schlegel, and P.F. Stadler. Alignments of mitochondrial genome arrangements: Applications to metazoan phylogeny. *Journal of Theoretical Biology*, 240(4):511–520, 2006.

[Han06] Y. Han. Improving the efficiency of sorting by reversals. In *Proc. International Conference on Bioinformatics and Computational Biology*, pages 406–409. CSREA Press, 2006.

[HBBR05] B. Hiller, J. Bradtke, H. Balz, and H. Rieder. CyDAS: a cytogenetic data analysis system. *Bioinformatics*, 21(7):1282–1283, 2005.

[HCKP95] S. Hannenhalli, C. Chappey, E.V. Koonin, and P.A. Pevzner. Genome sequence comparison and scenarios for gene rearrangements: A test case. *Genomics*, 30:299–311, 1995.

[Hol81] I. Holyer. The NP-completeness of some edge-partition problems. *SIAM Journal on Computing*, 10(4):713–717, 1981.

[HP95] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 178–189, 1995.

[HP99] S. Hannenhalli and P.A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM*, 46(1):1–27, 1999.

[HS05] T. Hartman and R. Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. *Journal of Computer and System Sciences*, 70(3):300–320, 2005.

[HS06] T. Hartman and R. Shamir. A simpler and faster 1.5-approximation algorithm for sorting by transpositions. *Information and Computation*, 204(2):275–290, 2006.

[Hug00] D. Hughes. Evaluating genome dynamics: the constraints on rearrangements within bacterial genomes. *Genome Biology*, 1(6):reviews0006–reviews0006.8, 2000.

[IKH+89] N. Iwabe, K.-I. Kuma, M. Hasegawa, S. Osawa, and T. Miyata. Evolutionary relationship of archaebacteria, eubacteria, and eukaryotes inferred from phylogenetic trees of duplicated genes. *Proceedings of the National Academy of Sciences of the United States of America*, 86:9355–9359, 1989.

[JA11] S. Jiang and M.A. Alekseyev. Weighted genomic distance can hardly impose a bound on the proportion of transpositions. To appear in *Proc. 15th Annual International Conference on Research in Computational Molecular Biology*, 2011.

[Jar30] V. Jarník. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 6:57–63, 1930.

[KBH+03] W.J. Kent, R. Baertsch, A. Hinrichs, W. Miller, and D. Haussler. Evolution's cauldron: Duplication, deletion, and rearrangement in the mouse and human genomes. *Proceedings of the National Academy of Sciences of the United States of America*, 100(20):11484–11489, 2003.

[KM07] F Kuttler and S. Mai. Formation of non-random extrachromosomal elements during development, differentiation and oncogenesis. *Seminars in Cancer Biology*, 17:56–64, 2007.

[Knu98] D.E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, second edition, 1998.

[KST99] H. Kaplan, R. Shamir, and R.E. Tarjan. A faster and simpler algorithm for sorting signed permutations by reversals. *SIAM Journal on Computing*, 29(3):880–892, 1999.

[KV03] H. Kaplan and E. Verbin. Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In *Proc. 14th Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 170–185. Springer-Verlag, 2003.

[LHWC06] C.L. Lu, Y.L. Huang, T.C. Wang, and H.-T. Chiu. Analysis of circular genome rearrangement by fusions, fissions and block-interchanges. *BMC Bioinformatics*, 7:295, 2006.

[LMSK63] J.D.C. Little, K.G. Murty, D.W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, 1963.

[LRSM10] Y. Lin, V. Rajan, K.M. Swenson, and B.M.E. Moret. Estimating true evolutionary distances under rearrangements, duplications, and losses. *BMC Bioinformatics*, 11(Suppl 1):S54, 2010.

[LTM05] T. Liu, J. Tang, and M.E. Moret. Quartet-based phylogeny reconstruction from gene orders. In *Proc. 11th Annual International Conference on Computing and Combinatorics*, volume 3595 of *Lecture Notes in Computer Science*, pages 63–73. Springer-Verlag, 2005.

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

[Mik03] I. Miklós. MCMC genome rearrangement. *Bioinformatics*, 19(Suppl 2):ii130–ii137, 2003.

[Mit95] F. Mitelman, editor. *Iscn 1995: An International System For Human Cytogenetic Nomenclature*. S. Karger AG, 1995.

[MJM10] F. Mitelman, B. Johansson, and F. Mertens, editors. *Mitelman Database of Chromosome Aberrations and Gene Fusions in Cancer*, http://cgap.nci.nih.gov/Chromosomes/Mitelman (Last checked 10/13/2010), 2010.

[MSM04] M. Marron, K.M. Swenson, and B.M.E. Moret. Genomic distances under deletions and insertions. *Theoretical Computer Science*, 325(3):347–360, 2004.

[MSTL02] B. Moret, A. Siepel, J. Tang, and T. Liu. Inversion medians outperform breakpoint medians in phylogeny reconstruction from gene-order data. In *Proc. 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 521–536. Springer-Verlag, 2002.

[MT] B.M.E. Moret and J. Tang. GRAPPA's homepage. World Wide Web, http://www.cs.unm.edu/∼moret/GRAPPA (Last checked 10/13/2010).

[MT04] B.M.E. Moret and J. Tang. *GRAPPA version 2.0 Manual*, 2004.

[MTWW02] B.M.E. Moret, J. Tang, L.-S. Wang, and T. Warnow. Steps towards accurate reconstructions of phylogenies from gene-order data. *Journal of Computer and System Sciences*, 65(3):508–525, 2002.

[MWB+01] B.M.E. Moret, S.K. Wyman, D.A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. 6th Pacific Symposium on Biocomputing*, pages 583–594, 2001.

[MWD00] J. Meidanis, M.E.M.T. Walter, and Z. Dias. Reversal distance of signed circular chromosomes. Technical Report IC-00-23, Institute of Computing, University of Campinas, 2000.

[NI92] H. Nagamochi and T. Ibaraki. A linear time algorithm for computing 3-edge-connected components in a multigraph. *Japan Journal of Industrial and Applied Mathematics*, 9:163–180, 1992.

[NT84] J.H. Nadeau and B.A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proceedings of the National Academy of Sciences of the United States of America*, 81(3):814–818, 1984.

[OFS07] M. Ozery-Flato and R. Shamir. On the frequency of genome rearrangement events in cancer karyotypes. In *Proc. 1st RECOMB Satellite Workshop in Computational Cancer Biology*, page 17, 2007.

[OFS08] M. Ozery-Flato and R. Shamir. Sorting cancer karyotypes by elementary operations. In *Proc. 6th Annual RECOMB Satellite Workshop on Comparative Genomics*, volume 5267 of *Lecture Notes in Bioinformatics*, pages 211–225. Springer-Verlag, 2008.

[PH88] J.D. Palmer and L.A. Herbon. Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence. *Journal of Molecular Evolution*, 27:87–97, 1988.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[PS98] I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. Technical Report TR98-071, Electronic Colloquium on Computational Complexity, 1998.

[PT03] P. Pevzner and G. Tesler. Genome rearrangements in mammalian evolution: Lessons from human and mouse genomes. *Genome Research*, 13:37–45, 2003.

[RXL+10] V. Rajan, A.W. Xu, Y. Lin, K.M. Swenson, and B.M.E. Moret. Heuristic for the inversion median problem. *BMC Bioinformatics*, 11(Suppl 1):S30, 2010.

[San99] D. Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15:909–917, 1999.

[San01] D. Sankoff. Gene and genome duplication. *Current Opinion in Genetics and Development*, 11:681–684, 2001.

[SB98] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5(3):555–570, 1998.

[SCL76] D. Sankoff, R.J. Cedergren, and G. Lapalme. Frequency of insertion-deletion, transversion, and transition in the evolution of 5S ribosomal RNA. *Journal of Molecular Evolution*, 7:133–149, 1976.

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

[SM01] A.C. Siepel and B.M.E. Moret. Finding an optimal inversion median: Experimental results. In *Proc. 1st Workshop on Algorithms*, volume 2149 of *Lecture Notes in Computer Science*, pages 189–203. Springer-Verlag, 2001.

[SMEDM08] K.M. Swenson, M. Marron, J.V. Earnest-DeYoung, and B.M.E. Moret. Approximating the true evolutionary distance between two genomes. *ACM Journal of Experimental Algorithmics*, 12:3.5:1–3.5:17, 2008.

[SPCD02] J. Salse, B. Piégu, R. Cooke, and M. Delseny. Synteny between arabidopsis thaliana and rice at the genome level: a tool to identify conservation in the ongoing rice genome sequencing project. *Nucleic Acids Research*, 30(11):2316–2328, 2002.

[SSK96] D. Sankoff, G. Sundaram, and J.D. Kececioglu. Steiner points in the space of genome rearrangements. *International Journal of Foundations of Computer Science*, 7(1):1–9, 1996.

[STTM09] K.M. Swenson, Y. To, J. Tang, and B.M.E. Moret. Maximum independent sets of commuting and noninterfering inversions. *BMC Bioinformatics*, 10(Suppl 1):S6, 2009.

[Stu26] A.H. Sturtevant. A crossover reducer in drosophila melanogaster due to inversion of a section of the third chromosome. *Biologisches Zentralblatt*, 46(12):697–702, 1926.

[SV05] M.R. Salavatipour and J. Verstraete. Disjoint cycles: Integrality gap, hardness, and approximation. In *Proc. 11th Conference on Integer Programming and Combinatorial Optimization*, volume 3509 of *Lecture Notes in Computer Science*, pages 51–65. Springer-Verlag, 2005.

[TBS07] E. Tannier, A. Bergeron, and M.-F. Sagot. Advances on sorting by reversals. *Discrete Applied Mathematics*, 155:881–888, 2007.

[TMCd04] J. Tang, B.M.E. Moret, L. Cui, and C.W. dePamphilis. Phylogenetic reconstruction from arbitrary gene-order data. In *Proc. 4th IEEE Conference on Bioinformatics and Bioengineering*, pages 592–599. IEEE Press, 2004.

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

[TS07] E. Tannier and M.-F. Sagot. personal communication, 2007.

[Tsi09] Y.H. Tsin. Yet another optimal algorithm for 3-edge-connectivity. *Journal of Discrete Algorithms*, 7:130–146, 2009.

[TZS08] E. Tannier, C. Zheng, and D. Sankoff. Multichromosomal genome median and halving problems. In *Proc. 8th Workshop on Algorithms in Bioinformatics*, volume 5251 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 2008.

[Wak68] A. Waksman. A permutation network. *Journal of the ACM*, 15(1):159–163, 1968.

[WC53] J.D. Watson and F.H.C. Crick. A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953.

[WDM98] M.E.M.T Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. In *Proc. Symposium on String Processing and Information Retrieval*, pages 96–102. IEEE Computer Society, 1998.

[XM10] A.W. Xu and B.M.E. Moret. Genome rearrangement analysis on high-resolution data. Submitted, 2010.

[XS08] A.W. Xu and D. Sankoff. Decomposition of multiple breakpoint graphs and rapid exact solutions. In *Proc. 8th Workshop on Algorithms in Bioinformatics*, volume 5251 of *Lecture Notes in Computer Science*, pages 25–37. Springer-Verlag, 2008.

[Xu08] A.W. Xu. A fast and exact algorithm for the median of three problem - a graph decomposition approach. In *Proc. 6th Annual RECOMB Satellite Workshop on Comparative Genomics*, volume 5267 of *Lecture Notes in Bioinformatics*, pages 184–197. Springer-Verlag, 2008.

[Xu09a] A.W. Xu. DCJ median problems on linear multichromosomal genomes: Graph representation and fast exact solutions. In *Proc. 7th Annual RECOMB Satellite Workshop on Comparative Genomics*, volume 5817 of *Lecture Notes in Bioinformatics*, pages 70–83. Springer-Verlag, 2009.

[Xu09b] A.W. Xu. A fast and exact algorithm for the median of three problem - a graph decomposition approach. *Journal of Computational Biology*, 16(10):1369–1381, 2009.

[YAF05] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

[YF08] S. Yancopoulos and R. Friedberg. Sorting genomes with insertions, deletions and duplications by DCJ. In *Proc. 6th Annual RECOMB Satellite Workshop on Comparative Genomics*, volume 5267 of *Lecture Notes in Bioinformatics*, pages 170–183. Springer-Verlag, 2008.

[YZT07] F. Yue, M. Zhang, and J. Tang. A heuristic for phylogenetic reconstruction using transposition. In *Proc. 7th IEEE Conference on Bioinformatics and Bioengineering*, pages 802–808, 2007.

[YZT08] F. Yue, M. Zhang, and J. Tang. Phylogenetic reconstruction from transpositions. *BMC Genomics*, 9(Suppl 2):S15, 2008.

[ZAT09] M. Zhang, W. Arndt, and J. Tang. An exact solver for the DCJ median problem. In *Proc. 14th Pacific Symposium on Biocomputing*, pages 138–149. World Scientific, 2009.

[ZZS06] C. Zheng, Q. Zhu, and D. Sankoff. Genome halving with an outgroup. *Evolutionary Bioinformatics*, 2:319–326, 2006.