uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Robert Warren
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Ph.D. (Computer Science)
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

On Gene Duplication

TITRE DE LA THÈSE / TITLE OF THESIS

David Sankoff
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

Sylvia Boyd

Anil Maheshwari

Todd Wareham
Memorial U. of Newfoundland

Lucia Moura

Gary W. Slater
Le Doyen de la Faculte des études superieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

On Gene Duplication

Robert B. Warren

Thesis Submitted to the Faculty of Graduate and Postdoctoral Studies
In partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Science [1]

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

[1]The program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

Your file Votre référence
ISBN: 978-0-494-73896-2
Our file Notre référence
ISBN: 978-0-494-73896-2

Abstract

Due the sheer size and complexity of genomes, it is essential to develop automated methods to analyze them. To compare genomes, one distance measure that has been proposed is to determine the minimum number of evolutionary changes needed to transform one genome into another. In recent years, great progress has been made in this area with efficient exact algorithms that can transform one genome to another applying a wide range of evolutionary operations. However, gene duplications, a common occurrence and arguably the most important evolutionary operation, have proven to be one of the most difficult evolutionary operations to integrate.

We examine the most successful gene duplication algorithms: a family of algorithms that we call the *rearrangement-duplication* algorithms. Rather than compare two genomes, these algorithms attempt to efficiently remove the duplicates from a genome using the fewest number of duplications and other evolutionary operations.

In this thesis we give a complete survey of all the *genome halving* algorithms, a highly successful group of rearrangement-duplication algorithms that efficiently and exactly handle whole genome doubling (*tetraploidization*). We also introduce the *genome aliquoting* algorithms, a new variation on the genome halving problem, that attempts to handle unlimited scale whole genome duplications. As a new and challenging problem there are currently no efficient exact algorithms. However, early results include two approximation algorithms.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

How similar are two species? Since the dawn of civilization farmers have tried to answer this question using taxonomy, an enterprise that was later picked up by scientists. With the discovery of DNA and the realization that an organism's phenotype often has an analog on the genotype, researchers began to wonder if two species could be compared by examining their genomes.

Numerous algorithms have been developed to compare genomes ranging from exact algorithms and statistical methods to heuristic and approximation algorithms. All of the algorithms work on the same fundamental idea: they try to quantify how much evolution has occurred between organisms. By finding a plausible path of evolution between two organisms it is possible to make a statement as to the similarity of two organisms: the shorter the evolutionary path the more similar the organisms.

In this thesis, we will look at the family of *maximum parsimony algorithms*, a family that has yielded numerous exact algorithms (see Section 1.3). By exact algorithms we mean exact in the computational sense; no algorithm can accurately reconstruct the evolutionary history of a species. However, maximum parsimony algorithms can accurately reconstruct the shortest evolutionary path between two species which at least provides an upper bound on the similarity of the species.

One weakness that has plagued the exact maximum parsimony algorithms is their inability to handle gene duplicates. Most of the algorithms to date have been forced

to include the unrealistic precondition that the genomes to be compared must contain exactly one copy of every gene. The only exception has been the exact polynomial-time algorithms[EMS03, AP07a, WS09b, Mix08] that deal with a specific instance of *whole genome duplication*, called *tetraploidization*, where a genome has exactly two copies of every gene. In this thesis we examine and build on these algorithms in an effort to provide a broader framework for handling gene duplications with maximum parsimony algorithms.

1.1 Biology Background

In this section we give a brief overview of evolution and molecular biology in order to outline the biological motivation for this work.

1.1.1 Genomes

All living things are made up of *cells*. *Cells* by themselves have all the characteristics of life: they are born, eat, reproduce and, eventually, die. In fact, many organisms consist only of a single cell, *e.g.* bacteria. The workings of cells are extremely complex and beyond the scope of this thesis; our interest is in what the cell contains: the *genome*.

For an organism that is made up of trillions of cells all those cells must be working together in order for the organism to survive. One of the means in which this is accomplished is by all the cells working off a single blueprint for the organism and that blueprint is the organism's *genome*. The *genome* itself consists of one or more molecules of *deoxyribonucleic acid* (*DNA*) called *chromosomes*. *DNA* is made up of *nucleic acid* which consists of four types of *bases*: *adenine* (A), *thymine* (T), *guanine* (G) and *cystosine* (C). Each type of base is chemically attracted to another type: A and T are attracted to each other as are G and C. Thus, bases tend to pair up

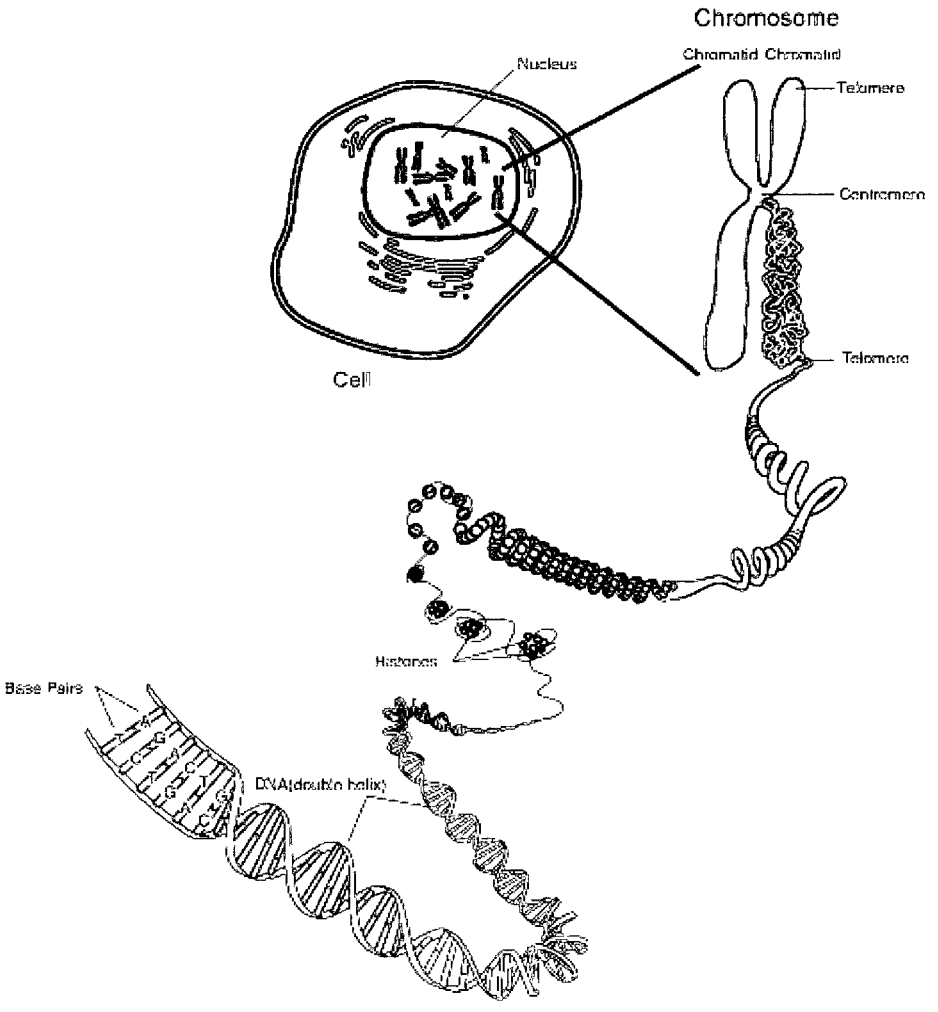


Figure 1.1: The figure depicts the organization of an organism's genome. The genome is contained in the nucleus of a cell. It is divided into chromosomes which are double stranded molecules of DNA. Finally, DNA consists of the amino acids A, G, T and C paired together across each strand. *Source:* National Institutes of Health, National Human Genome Research Institute, Division of Intramural Research.

into *base pairs* and, as a result, all the *chromosomes*, as *DNA* molecules, are divided into two connected *strands* of nucleic acids which entwine around each other forming a *double helix* structure with each *strand* being the complement of the other. Figure 1.1 clearly illustrates this arrangement.

Chemically, a nucleic acid consists of more than just bases. For stability, each base must not only be paired with its complementary base but also with sugar and

a phosphate group. As a result, each *strand* is composed of an alternating chain of a sugar, a base and a phosphate group. This implies a *directionality* of the *strand* going from phosphate to sugar or *vice versa*. The sugar side of a base is called *three prime*, denoted *3'*, and the phosphate group side is called *five prime*, denoted *5'*. Thus, a strand of DNA, or some subset of a strand, can be considered oriented from 5' to 3' or *vice versa*. The orientation of a strand is particularly useful in the context of a double helix as the two strands are complementary and, thus, if one strand is oriented from 5' to 3' then the other is oriented from 3' to 5'. Therefore, the orientation of the strands makes it easy to distinguish which strand is which in discussion.

As a set of instruction for a cell, DNA works on a principle identical to that of computer code, though the implementation differs significantly. While an exact interpretation of DNA still eludes researchers, what is apparent is that encoded within the DNA are regions of importance known as *genes*. A gene can appear on either strand of the DNA, however, each gene will only appear on one of the strands; a gene will not alternate code between strands. On the strand opposite to the gene is the gene's complement, which is gibberish and can be ignored.

There are two types of cells: *prokaryotes* and *eukaryotes*. *Prokaryotes* are thought to be earlier evolutions of the cell, with *eukaryotes* being the more modern variations. The differences between the two types of cells are extensive but there are two primary differences. First, *eukaryotes* are capable of combining to form *multicellular organisms*, *e.g.* humans, while *prokaryotes* cannot and, thus, are found exclusively in *unicellular organisms*, *e.g.* bacteria. Second, *eukaryotes* have a *nucleus*, a subunit that protects the genome, and *prokaryotes* do not. Since the cell in Figure 1.1 has a nucleus it is a eukaryote cell.

The genome does not have a monopoly on DNA; DNA is often founds in other parts of a cell as well. In nearly all eukaryotes DNA can be found in a subunit called a *mitochondrion*, which is responsible for supplying the cell with energy. Similarly, any eukaryote that derives energy from photosynthesis requires a subunit called a *chloro-*

plast that contains DNA. Both kinds of cells contain *plasmids*, virus-like symbiotic DNA which takes advantage of the host cell to reproduce.

Given our focus on DNA, the main difference between a eukaryotes and prokaryotes, mitochondria, chloroplasts and plasmids for us is that eukaryotes tend to have *linear-multichromosomal genomes* while the rest exclusively have a single circular molecule of DNA which, without a loss of generality, we will refer to as a *circular-unichromosomal genome.* A genome is said to be *multichromosomal* if and only if it contains more than one chromosome and *unichromosmal* otherwise. Chromosomes can be *circular* or *linear*; the two endpoints of a linear chromosome are called *telomeres.* In general, if there is more than one chromosome in a genome then they are all linear and, similarly, if there is only one chromosome in the genome then it is circular; mixed linear and circular chromosomes are not thought to be a stable configuration. However, recently, in bioinformatics, it has become popular to discuss mixed-multichromosomal genomes as it is one less constraint on the output, easing the computational burden[WS09b, Mix08, TZS09].

The discussion on multichromosomal versus unichromosomal genomes does not include the *ploidy* of the genome, *i.e.* the number of copies of a chromosome. For example, while some organisms are *haploids*, having only one copy of each chromosome, most mammals are *diploids*, meaning they have two copies of every chromosome, one from the mother and one from the father. It is possible for a mammal to inherit both copies from one or both the parents, becoming a *triploid* or a *tetraploid* respectively, but both are generally fatal in mammals[Ohn70]. On the other hand, it is very common for plants to have four or more copies of a chromosome[Gre05], for example, wheat is a *hexaploid* having six copies of each chromosome.

If there is more than one copy of a chromosome then copies tend to bond together into pairs. To accommodate this, the middle of a chromosome has a special molecular structure, this is called a *centromere.* For diploids with linear chromosomes, this gives rise to the familiar X shape commonly associated with chromosomes, see Figure 1.1

for an example.

1.1.2 Evolution

Cells are mortal and, thus, to ensure their survival, reproduce over time. Eukaryotic cells reproduce in two ways, *meiosis* and *mitosis*, while prokaryotic cells reproduce exclusively by *binary fission*. *Meiosis* is sexual reproduction between different members of the same species in order to produce offspring. *Mitosis* is when a cell attempts to create an exact copy of itself so that there will be another cell to take over after the current one dies. *Binary fission* is the same as mitosis but since prokaryotes are so physically different from eukaryotes the process of binary fission is very different from the process of mitosis. Regardless of the type of reproduction, changes in DNA inevitably result (intentionally in the case of meiosis, but accidentally in mitosis and binary fission), which can lead to evolution. In general, for multicellular organisms, meiosis is the main driver of evolution. Since in multicellular organisms changes caused by mitosis are not typically passed on to the next species, these changes tend to only cause diseases that are not passed onto the next generation. However, for unicellular organisms mitosis and binary fission are the drivers of evolution. Regardless of the scale or method of reproduction the end is always the same: an offspring with slightly different genetic material than their parent. Over generations these differences become significant and at some point the descendant diverges into a separate species: *speciation* has occurred.

Studying how organisms evolve over time is the study of the differences between organisms and, in particular, we are interested in large scale evolutionary differences between species rather than small scale ones that causes differences between individuals of the same species.

In 1970 [Ohn70] made the bold claim that duplications are the most important factor in evolution and a growing body of evidence seems to suggest that he

was correct[Gre05]. Most changes to genes are not positive and can cripple or kill an organism. An organism with "extra" copies of a gene has a gene that can be experimented on without danger. The result, in most cases, is that the duplicates lose functionality, what we refer to as *deleted*. Occasionally, however, the duplicate changes the function of the gene, either to a completely new function or to a specialization of the existing function. A duplication can duplicate a whole genome, a whole chromosome, a segment of a chromosome, a single gene or an even smaller amount of DNA. We examine each case below.

The largest and most spectacular evolutionary operation is *polyploidization*: the creation of a polyploid. During meiosis a diploid organism splits its genome among two haploids cells. These haploids combine with the haploids of the organism's sexual partner to produce a new diploid: combing elements from both organisms for a better chance of survival. However, occasionally one or more of the cells involved will fail to split into haploids but still manage to recombine resulting in a polyploid. For other types of organisms there is a similar process but the result is the same: an error in meiosis results in polyploidization. When members of different species interbreed and the result is a polyploid then it is called an *alloploid*. While normally inter-species children are infertile, alloploids generally can reproduce[Gre05]. More common[Gre05], however, are polyploids resulting from a pairing of the same species, called an *autoploid*. For our research, we consider only *autoploids*.

Less extreme than polyploidy is *polysomy*: the case where an organism has at least one extra chromosome. This seems to occur when, during meiosis, one (or more) chromosomes fail to separate. Unlike polyploidy, polysomy is not necessarily immediately fatal in mammals; for example, Down syndrome is a case of trisomy. Nevertheless, like polyploidy, polysomy doesn't appear to be a major evolutionary force in mammals as it is often fatal or the resulting organism is sterile[Ohn70]. However, for some species this can be a major way that duplications occur[Ohn70].

During mitosis, diploids frequently exchange the ends of the two copies of the

chromosomes. On a rare occasion, the ends it exchanges are slightly unequal, the result being one of the chromosomes ends up with two copies of the same gene. This type of duplication is called a *tandem duplication* and it has the interesting property that the duplicated gene appears adjacent to the original. In the more extreme case, instead of one gene being duplicated in this fashion a whole segment of a chromosome, potentially with dozens of genes duplicated. While the two segments are adjacent, it is not necessarily the case that any two genes are, as a result this type of duplication is called a *segment duplication.*

The duplication of small amounts of DNA, in the order of tens of base pairs, seem to occur frequently, especially at the *telomeres* (ends) and *centromeres* (middle) of (linear) chromosomes. These are not necessarily next to the original and, thus, are called *duplicative transpositions.* These duplications are too small to carry genes and, thus, are of little interest for our work.

Chromosomes occasionally break. This is usually caused by some outside force such as ionizing radiation or a chemical mutagen. In any event, the broken chromsomes will repair themselves although occasionally they repair themselves incorrectly changing the order of the genes. Beyond duplications, these *genome rearrangements* are another major force in evolution. The order of the genes can change within a chromosome by either a *reversal* or a *transposition.* Alternatively, the chromosome on which the gene is located can be changed by *translocation*, *fusion*, or *fission.*

When a chromosome is repaired after a break, the broken piece is occasionally reinserted on the wrong strand. Since the 5' side must be attached to a 3' side and *vice versa*, if there is more than one gene on the broken strand then not only do all the genes appear on the wrong strand but the order in which the genes appear along the chromosome is also reversed. This is called a *reversal.*

If a chromosome breaks into four pieces due to three breaks, it is possible that the chromosome gets put back together in the wrong order. Specifically, while the telomeres prevent the two end pieces from getting misplaced, the two middle pieces

can accidentally swap position. This is called a *transposition.*

Rather than a piece inside of a chromosome breaking, the ends of two chromosomes might break. In this case, during the repair process, the ends of the chromosomes might be accidentally swapped: a *translocation.* Again, like a reversal, because the 5' side must be attached a 3' side and *vice versa*, any genes on the ends not only end up on the wrong chromosome but also in the opposite order. Hence, a translocation is conceptually like a reversal between chromosomes.

When two chromosomes have *acrocentric* (off-centre) centromeres, a common special case of a translocation called a *Robertsonian translocation* can occur. During a Robertsonian translocation, the two chromosomes split at their centromeres such that two shorter ends without centromeres join together and the two longer ends with centromeres join together. Without a centromere, the two short ends are lost, only the joined long end remain, effectively resulting in a *fusion* between the two chromosomes. Fusions/Robertsonian translocations are often not fatal, and if no significant genetic material is contained in the short ends then the organism can be perfectly normal.

Centromere fission is essentially the opposite of a Robertsonian translocation. In this situation, the centromere splits in two with the two short ends joining together and the two long ends join together, each with half of the centromere in the middle. With half of a centromere they are still stable and, as a result, the two halves of a centromere become two new centromeres and the chromosome is permanently split in two. We refer to this process as *fission.*

The main contribution of rearrangement events towards evolution is that they can cause inter-organism sterility, hence, speciation. The reason for this is that it makes it difficult for recombination to occur between an organism with the rearrangement event and an organism without the rearrangement event during meiosis as it is more difficult to properly align DNA. This is not to say that recombination cannot occur if any difference in rearrangements occur; indeed, even between humans such events occur. However, rearrangement events make speciation more likely.

Important to our study is that genes are added, removed or change. While there is often more than one cause to any of the evolutionary operations that we have studied so far, the result is the same and, hence, we can study the evolutionary operation. Unfortunately, there aren't really any evolutionary operations which add, remove or alter genes. Instead, it occurs as a consequence of other evolutionary operations; most of the above operations could be a trigger. Nevertheless, it is worth studying *deletions*, *mutations* and *insertions* as events in their own right.

As previously mentioned, the *deletion* of a gene is the most common alteration to a gene. Much of the time this has no significant effect on the genome as it is the removal of unused DNA called *"junk DNA"*. However, occasionally a large section of a chromosome is deleted or a small amount is deleted from a gene causing the gene to malfunction (mutation) or cease to function altogether (deletion) resulting in the loss of one or more genes. Since the deletion may only delete part of a gene, even after the gene is deleted some of it can remain becoming new "junk DNA".

A new gene can be created through many processes although it is not as frequent an occurrence as gene deletion. Small amounts of DNA are constantly being added to the genome and occasionally this can reactivate some "junk DNA" turning into a new gene. Similarly, a reversal can sometimes join two "junk DNA" together such that a new gene is created. However, perhaps the most common method of gaining a new gene is through the duplication of an existing gene that takes on a new function. We call the creation of a new gene an *insertion*.

Similar to insertions are *mutations*, a change in function of an existing gene. These are often very hard to distinguish from insertions.

1.2 Informatics Background

In this section we give an overview of the mathematical and computer science notation that we will be using.

1.2.1 Algorithms

Computer science is the study of *algorithms*, step-by-step instructions that solve a problem. A problem can be anything although the most fundamental type of problem is that which has a "yes" or "no" answer, a *decision problem.* Of more interest to us are *optimization problems*, problems that involve maximizing or minimizing a particular value. But any optimization problem can be transformed into a decision problem by restating the problem from say "minimizing a value" to "does there exists a value less than or equal to k, for some constant k"; repeatedly asking the decision questions allows us to find the answer to the optimization problem and, conversely, knowing the optimal solution allows us to answer the decision problem. In computer science there are some semantic differences between decision problems and optimization problems that we would rather not concern ourselves with; as a result, while most of the problems in this thesis are stated as optimization problems we will consider them as decision problems.

While algorithms solve one specific problem, an algorithm can be used to indirectly solve other problems via a *reduction.* A *reduction* is an algorithm that transforms an instance of one problem into an instance of another problem. We say we have reduced the first problem to the second problem. Thus if there is an algorithm that solves instances of the second problem we can use it and the reduction to solve instances of the first problem.

A central concern of computer science is the efficiency of algorithms which is typically denoted in *big-O notation*, *e.g.* $O(n^2)$ or $O(2^n)$, which indicates that the efficiency of the algorithm has an asymptotic upper bound in proportion to the formula. Efficiency can refer to either running time or space (memory) consumed, *e.g.* an algorithm might run in $O(2^n)$ time and consume $O(n^2)$ space where n is the size of the input. Algorithms whose running time can be expressed, in big-O notation, as a polynomial in the size of the input are called *polynomial-time* algorithms. All

polynomial-time algorithms must consume at most a polynomial amount of space but the reverse is not necessarily true.

Problems can be divided into *complexity classes* based upon the properties of the algorithms that solve them. Any problem that has an algorithm that runs in polynomial time belongs to the class *P*. Problems that belong to class P are desirable as they are considered to have efficient solutions. There are many other complexity classes but there is only one other in which we are interested: the class *NP*.

A problem *belongs to class NP* if a possible solution to an instance of the problem can be verified to be an actual solution of the instance of the problem in polynomial time. Obviously, all problems in P also belong to NP, but, in general, membership in class NP itself is not interesting. Instead, we are interested in knowing if a problem is *NP-hard*, a problem is *NP-hard* if there exists a polynomial-time reduction from every problem in class NP to it. This is generally established by finding a reduction from a known NP-hard problem to a suspected NP-hard problem. A problem needn't be a member of class NP in order to be NP-hard, but problems that are both in NP and NP-hard are called *NP-complete*. NP-hard problems are undesirable as they are thought of as inefficient; to date there are no problems that are known to be both in P and NP-hard although such problems might exist (it is an open problem) in which case P=NP.Thus, to classify problems efficient or inefficient we attempt to establish if they are in P or if they are NP-hard, under the assumption that P$\neq$NP.

Occasionally, because an exact solution to the problem is difficult to find an inexact solution is sought instead. The exact definition of an inexact solution varies depending on the problem but for an optimization problem it often means a not necessarily optimal solution. However, to be of any use, such an unoptimal solution must typically be near-optimal. Any algorithm that gives an inexact solution to the problem is called a *heuristic*. A heuristic whose margin of error is bounded is called an *approximation algorithm*. Approximation algorithms are often defined by their bound such that an approximation algorithm with a bound ϵ is said to be an ϵ-approximation,

e.g. a maximization algorithm with a bound of $\log n$, where n is the size of the input to an algorithm, meaning that the solution returned by the algorithm falls somewhere between the optimal solution and $\log n$ times the optimal solution, is called a $\log n$-approximation. For a $\log n$-approximation for a minimization algorithm, where n is the size of the input to an algorithm, the solution similarly falls between the optimal solution and $1/\log n$ times the optimal solution. The boundary needn't be a function, it might be constant, *e.g.* a 2-approximation for a maximization algorithm returns an answer between the optimal solution and twice the optimal solution.

1.2.2 Sets and Sequences

A *set* is an unordered container of distinct elements. A set is denoted either by an italicized capital letter, *e.g.* a set S, or by listing the elements of the set between curly brackets, *e.g.* a set $\{x_1, x_2, \ldots, x_n\}$. Since a set is unordered, two sets are equivalent if and only if they contain the same elements, *e.g.* $\{x_1, x_2, x_3\} = \{x_3, x_2, x_1\} = \{x_1, x_3, x_2\}$. For any set A and element x, x is *in* A, denoted $x \in A$, or A *has* x if and only if x is one of the elements that makes up A. There are two special sets: the *empty set*, denoted $\emptyset$, which is a set with no elements, and the *universe*, denoted U, which is a set with all possible elements.

In addition to the equivalence operation mentioned above there are a number of other common set operations. Given two sets A and B, we say A is a *proper subset* of B, denoted $A \subset B$, if and only if B contains all the elements of A but A does not contain all the elements of B. A is a *subset* of B, denoted $A \subseteq B$ if and only if A is a proper subset of B or A is equivalent to B. The *union* of A and B, denoted $A \cup B$, is the set that contains all the elements of A and all the elements of B. The *intersection* of A and B, denoted $A \cap B$, is the set that contains only the elements that are in both A and B. The *complement* of A in B, denoted $B \setminus A$, is all the elements of B that are not in A. Given a set A, the *cardinality* of A, denoted $|A|$, is the number of

elements in A.

Computationally, because a set is unordered we can implement a set using a hash table, which allows the operations of accessing, adding and removing a single element to be performed with an average case of O(1) time complexity. If the universe is finite and small, as is the case for our algorithms, the time complexity can be improved to O(1) in the worst case at the increase of space complexity from O($|S|$) to O($|U|$) where S is a set. With this implementation, all the common operations between sets listed above can be performed in O($|U|$) time. The cardinality of a set is typically stored with the set and updated when set operations are performed. This sort of record keeping requires no significant overhead and allows the cardinality to be obtained in O(1) time.

Relaxing the definition of a set by removing the restriction that the elements be distinct results in a container known as a *multiset*. Conceptually a multiset is identical to a set, thus, we use the same notation as sets. However, some changes to the underlying behavior of the operations are necessary to accommodate multiple copies of each element.

Formally, a multiset is a set A consists of the pair (S, F), where S is a set called the *underlying set of elements*, and a F is a function that maps the elements in the set to non-negative natural numbers. Since accessing the function is so important, we define the shortcut function $\textsc{Multiplicity}_A = \mathrm{F}$. Thus, for each element x in the underlying set of elements of a multiset A, the *multiplicity* of x, *i.e.* the number of occurrences of x in the multiset, is the number $\textsc{Multiplicity}_A(x)$. To keep the notation of multisets consistent with sets, the expanded form of a multiset indicates each element's multiplicity by multiple copies of each element, *e.g.* a multiset $A = (S, \mathrm{F})$ where $S = \{x_1, x_2, x_3\}$ and $\mathrm{F}(x_1) = 2$, $\mathrm{F}(x_2) = 1$ and $\mathrm{F}(x_3) = 4$ expands to any permutation of $\{x_1, x_1, x_2, x_3, x_3, x_3, x_3\}$. An element that is absent from a multiset, such as x_4 in the previos example, is said to have a multiplicity of 0. Because they use the same notation, a sets and multisets can be compared by treating the set

as a multiset where all elements have a multiplicity of one. However, the result of any operations between a set and a multiset is a multiset.

Since multisets use the same notation as sets, they have the same operations as sets. The behavior of these operations is the same except that they must account for the multiplicity of the multiset's elements. In fact, performing an operation on a multiset is the same as performing the operation on its underlying set plus performing a related operation on the multiplicity function. Thus, given two multisets $A = (S, \mathrm{F})$ and $B = (S', \mathrm{F'})$, $A = B \equiv S = S' \wedge \forall x \in S\{\mathrm{F}(x) = \mathrm{F'}(x)\}$. Similarly, $A \subset B \equiv S \subset S' \vee \exists x \in S\{\mathrm{F}(x) < \mathrm{F'}(x)\}$. Using the above definitions for equality and proper subset, A is a subset of B if and only if A is a proper subset of B or A is equal to B. $A \cup B = (S'', \mathrm{F''})$ where $S'' = S \cup S$ and $\mathrm{F''}(x) = \mathrm{F}(x) + \mathrm{F'}(x)$, for all x in S''. Similarly. $A \cap B = (S'', \mathrm{F''})$ where $S'' = S \cap S$ and $\mathrm{F''}(x) = \min(\mathrm{F}(x), \mathrm{F'}(x))$, for all x in S''. Finally, $A \setminus B = (S'', \mathrm{F''})$ where $S'' = S \setminus S$ and $\mathrm{F''}(x) = \mathrm{F}(x) - \mathrm{F'}(x)$, for all x in S''. The cardinality of a multiset is the sum of the multiplicity of each of its elements.

A multiset can be implemented in the same manner as a set; it takes virtually no overhead in time and space to track the multiplicity of a multiset. Thus, the time complexity of multiset operations are the same as that of set operations.

Further relaxing the definition of a set to remove the unordered requirement results is in a container called a *sequence*. A sequence is very different from a set as the order of the elements is as important as the elements themselves. Thus, sequences use different notation then sets or mutlisets. A sequence is represented by a non-italicized capital letter, *e.g.* a sequence S, or as its elements enclosed in round brackets, *e.g.* a sequence $(x_1, x_2, \ldots, x_n)$. Since it is ordered, two sequences are equivalent if and only if they contain the same elements in the same order, *e.g.* $(x_1, x_2, x_1, x_3) \neq (x_3, x_1, x_2, x_1) \neq (x_3, x_2, x_1, x_1)$. Since sequences are ordered it is possible to access an element by its position. Given a sequence S, we denote S[i] to be the element in the i^{th} position of the sequence.

Aside from equality, the only operations for sequences that are needed are the *length* operation and the *concatenation* operation. The length of a sequence A, denoted |A|, is the total number of elements in the sequence. The concatenation of two sequences A and B, denoted A||B, merges the sequences such that all the elements of B appear after all the elements in A, *e.g.* A = (x_1, x_2, x_3) and B = (y_1, y_2) become A||B = $(x_1, x_2, x_3, y_1, y_2)$. We use the summation symbol $\sum$ to denote the concatenation of multiple sequences.

Maintaining a sequence is more complex than maintaining a set. Thus, a sequence can only be implemented so that either reading is optimized or writing is optimized but not both. We will assume that all sequences are optimized for writing so that adding an element takes constant time (we will refer to a sequence that is optimized for reading rather than writing as an *array*). It is worth noting that, in a sequence, even though concatenation adds m elements, it still takes O(1) time. However, reading takes O(n) time where n is the length of the sequence. Additionally, making modifications within a sequence, while only requiring O(1) time, requires finding a position within the sequence which requires a read increasing the time complexity to O(n) time. Reading a prefix or suffix of the sequence gives a slightly better time complexity, however, requiring only O(m) time where m is the length of the prefix or suffix. Thus, for reading the first or last element of the sequence, since it is a prefix or suffix of length one, requires O(1) time. Thus, we provide the special functions PREFIX(S, m) and SUFFIX(S, m) to find the prefix and suffix respectively where S is a sequence and m is the length of prefix or suffix of S to find.

1.2.3 Graphs

With the three fundamental containers it is now possible to define more complex data structures. A *graph* is one such data structure, represented by a calligraphic capital letter, *e.g.* a graph $\mathcal{G}$, defined as a pair of disjoint sets $\mathcal{G} = (V, E)$: a non-empty finite

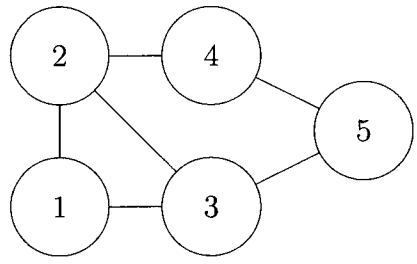


Figure 1.2: An example of a graph.

set V whose elements are called *vertices* and a set E of 2-element subsets of V called *edges*. For example, Figure 1.2 depicts a graph with vertices $V = \{1, 2, 3, 4, 5\}$ and edges $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}\}$. As a shorthand, we define the functions $\textsc{Vertices}(\mathcal{G})$ and $\textsc{Edges}(\mathcal{G})$ to refer to the set of vertices and edges in a graph $\mathcal{G}$ respectively. The cardinality of the set of vertices, called the *order*, is denoted $|\mathcal{G}|$ while the cardinality of the set of edges is denoted $\|\mathcal{G}\|$. If $u, v \in \textsc{Vertices}(\mathcal{G})$ and $e = \{u, v\} \in \textsc{Edges}(\mathcal{G})$ then we say that u and v are *the endpoints* of e, that they are *adjacent* to each other, and that they are *incident* with e or that e is *incident* with u or v. The set of vertices adjacent to a vertex u can be found using the function $\textsc{Adjacent}(u)$. The set of edges incident to a vertex u can be found using the function $\textsc{Incident}(u)$; the cardinality of this set is called the *degree* and can be found using the function $\textsc{Degree}(u)$. For example, in Figure 1.2 $\textsc{Adjacent}(2) = \{1, 3, 4\}$, $\textsc{Incident}(2) = \{\{1,2\}, \{2,3\}, \{2,4\}\}$ and $\textsc{Degree}(2) = 3$. A graph with no vertices (order 0) is called the *empty graph* $\mathcal{G} = (\emptyset, \emptyset)$ and is written as $\emptyset$.

Since a graph is just a pair of sets, two graphs can be compared like sets. A vertex v is said to be *in* a graph $\mathcal{G}$, denoted $v \in \mathcal{G}$ if and only if $v \in \textsc{Vertices}(\mathcal{G})$. Similarly, an edge e is said to be *in* a graph $\mathcal{G}$, denoted $e \in \mathcal{G}$ if and only if $e \in \textsc{Edges}(\mathcal{G})$. Two graphs are *equal* if and only if their sets are equal, *i.e.* $\mathcal{G} = \mathcal{G}'$ if and only if $\textsc{Vertices}(\mathcal{G}) = \textsc{Vertices}(\mathcal{G}')$ and $\textsc{Edges}(\mathcal{G}) = \textsc{Edges}(\mathcal{G}')$. Similarly, a graph is a *subgraph* of another graph if and only if one graph's sets are subsets of the other's, *i.e.* $\mathcal{G} \subseteq \mathcal{G}'$ if and only if $\textsc{Vertices}(\mathcal{G}) \subseteq \textsc{Vertices}(\mathcal{G}')$ and $\textsc{Edges}(\mathcal{G}) \subseteq \textsc{Edges}(\mathcal{G}')$

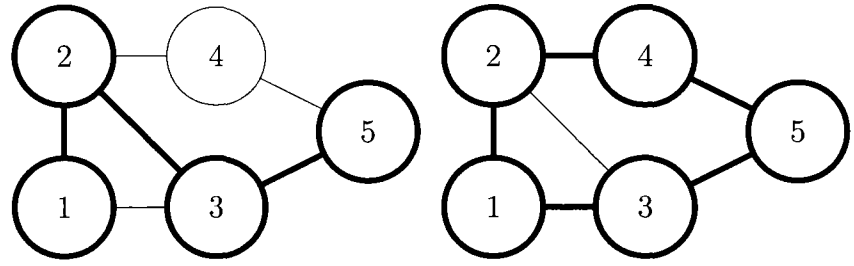


Figure 1.3: (a) The vertices and edges in bold compose one of many paths in this graph. (b) The vertices and edges in bold compose one of three cycles in this graph; it is the only Hamiltonian cycle in the graph.

and the edges in $\mathcal{G}$ must have their endpoints in $\textsc{Vertices}(\mathcal{G})$. A subgraph $\mathcal{G}$ of $\mathcal{G}'$ is *induced* if and only if for every edge $e \in \mathcal{G}'$ if both endpoints are in $\mathcal{G}$ then e must be in $\mathcal{G}$. A subgraph $\mathcal{G}$ of $\mathcal{G}'$ is *spanning* is $\textsc{Vertices}(\mathcal{G}) = \textsc{Vertices}(\mathcal{G}')$.

Graphs can be modified by adding or removing edges and vertices. Let $\mathcal{G} = (V, E)$ and $\mathcal{G}' = (V', E')$ be two graphs. The *union* of two graphs is the same as the union of both its sets, *i.e.* $\mathcal{G} \cup \mathcal{G}' = (V \cup V', E \cup E')$. Similarly, the *intersection* of the two graphs is the intersection of both its sets, *i.e.* $\mathcal{G} \cap \mathcal{G}' = (V \cap V', E \cap E')$. When $\mathcal{G} \cap \mathcal{G}' = \emptyset$ we say that they are *disjoint*. A set of vertices, $V'' \subseteq V$ can be *deleted* from $\mathcal{G}$, denoted $\mathcal{G} - V''$, meaning that we transform $\mathcal{G}$ into $\mathcal{G}'' = (V \setminus V'', E \setminus \bigcup_{v \in V''} \textsc{Incident}(v))$. When V'' is a singleton $\{v\}$, we can write $\mathcal{G} - v$ instead of $\mathcal{G} - \{v\}$. Similarly, a set of edges E'' where the endpoints of each edge are vertices in V, can be *added* or *removed* from $\mathcal{G}$, respectively denoted $\mathcal{G} + E''$ or $\mathcal{G} - E''$, by transforming $\mathcal{G}$ into $\mathcal{G}'' = (V, E \cup E'')$ or $\mathcal{G}'' = (V, E \setminus E'')$ respectively.

For a graph $\mathcal{G}$, a *walk* is a sequence W of vertices such that each vertex is connected to the next vertex by an edge in $\mathcal{G}$. The vertices and edges that form the walk are said to be *visited*. The first vertex in W is called the *start vertex* and the last is called the *end vertex*. All other vertices along the walk are called the *internal vertices*. A walk with an odd number of edges is called *odd*, while a walk with an even number of edges is called *even*. A *circuit* is a walk where the start and end

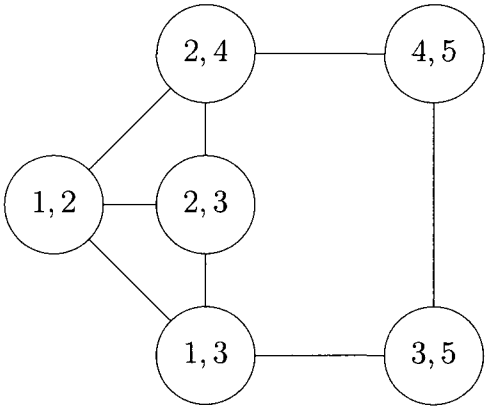


Figure 1.4: The line graph of the graph depicted in Figure 1.2.

vertices are the same. A walk is called *Eulerian* if and only if each edge in the graph is visited exactly once, *e.g.* the graph depicted in Figure 1.3a has an Eulerian walk ($\{3, 1, 2, 3, 5, 4, 2\}$) but no Eulerian circuit. A *path* is a walk where each vertex is visited at most once, *e.g.* $\{1, 2, 3, 5\}$ in Figure 1.3a. A *cycle* is a walk where each vertex is visited at most once except for the start and end vertices, which are the same, *e.g* $\{1, 2, 4, 5, 3, 1\}$ in Figure 1.3b. A path or cycle is called *Hamiltonian* if every vertex in the graph is visited exactly once, except, in the case of cycles, for the start and end vertices, see Figure 1.3b for an example. $\mathcal{G}$ is *connected* if and only if every vertex has a path to every other vertex, otherwise it is *disconnected.* A induced subgraph $\mathcal{G}$ is called a *component* of $\mathcal{G}$ if and only if it is connected and it is *maximal* (the addition of any other vertex from $\mathcal{G}$ will cause it to become disconnected). A vertex v that if deleted from the graph will cause the component to split into two or more components is called a *cut vertex.* A graph with no cycles is called a *forest.* A connected forest is called a *tree.* A tree with two vertices of degree 1 and all other vertices with degree 2 and is called a *path graph.* A graph where every vertex has degree 2 is called a *cycle graph.*

Of particular importance to the work in this thesis is a *line graph* which is effectively a graph where the vertices and edges have been switched. Formally:

Definition 1.2.1. *Given a graph $\mathcal{G}$, its* line graph, *denoted* LINEGRAPH($\mathcal{G}$), *is a graph such that:*

- *each vertex of* LINEGRAPH($\mathcal{G}$) *corresponds to an edge in $\mathcal{G}$;*
- *two vertices in* LINEGRAPH($\mathcal{G}$) *are adjacent if and only if their corresponding edges in $\mathcal{G}$ share a common endpoint;*

For example, given the graph $\mathcal{G} = (V, E)$ where $V = \{1, 2, 3, 4, 5\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{4, 5\}\}$, depicted in Figure 1.2, its line graph LINEGRAPH($\mathcal{G}$) $= (V', E')$ where $V' = E$ and $E' = \{\{1, 2\}, \{2, 3\}\}, \{\{1, 2\}, \{1, 3\}\}, \{\{1, 3\}, \{2, 3\}\}, \{\{1, 2\}, \{2, 4\}\}, \{\{2, 3\}, \{2, 4\}\}, \{\{1, 3\}, \{3, 5\}\}, \{\{2, 4\}, \{4, 5\}\}, \{\{3, 5\}, \{4, 5\}\}\}$, is depicted Figure 1.4.

A graph is typically implemented using the *adjacency-list representation*: a graph $\mathcal{G} = (V, E)$ is represented by an array of size $|V|$, one for each vertex $v \in V$, of lists, the list containing a pointer to the vertices in ADJACENT(v). The advantage of this representation is that it uses a small amount of space, $O(|V| + |E|)$ and it easy to add and remove edges. The drawback of this representation is that finding an edge takes $O($DEGREE$(v))$ time, where v is one of the endpoints of the edge, which is potentially $O(|V|)$. However, for sparse graphs this worst case will not occur and, thus, for our purposes all algorithms assume this representation.

Given a graph $\mathcal{G} = (V, E)$ and a set $I \subseteq V$, I is an *independent set* if and only if no two vertices I are adjacent in $\mathcal{G}$. A graph whose vertices can be divided into disjoint sets U and V such that U and V are independent sets is called a *bipartite graph.* A related famous problem in Computer Science is the *maximum independent set problem* defined as follows:

Definition 1.2.2. *Given a graph $\mathcal{G} = (V, E)$, the* maximum independent set problem *is to find an independent set with maximum cardinality out of all independent sets of $\mathcal{G}$.*

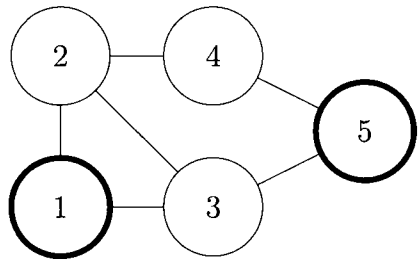


Figure 1.5: The bold vertices are in a maximum independent set for this graph.

Algorithm 1.1: RESTRICTEDMAXIMUMINDEPENDENTSET

Input: A graph $\mathcal{G}$ where all vertices have a degree of 2 or less.
Output: A maximum independent set I.

```
1  I ← ∅
2  partition G into components
3  foreach component C in G do
4      let VERTICES(C) = v_1, v_2, ..., v_n s.t. v_i is adjacent to v_{i+1} and v_1 ≠ v_n
5      for i ← 1 to n do
6          if i is odd then
7              add vertex v_i to the independent set I
8          end
9      end
10 end
11 return I
```

For example, a maximum independent set for the graph depicted in Figure 1.2, is $I = \{1, 5\}$, depicted in Figure 1.5. In this particular example there are numerous maximum independent sets.

The maximum independent set problem is well known to be NP-hard, even when the vertices of the graph are restricted to degree 3 [GJ90]. For graphs with vertices with degrees of at most 2, which consist of disjoint cycles and paths, the maximum independent set can easily be computed in linear time (see Algorithm 1.1).

Closely related to independent sets are *matchings*: set of independent edges. Formally, a matching is a set $M \subseteq E$ where no two edges in M share endpoints in a graph $\mathcal{G} = (V, E)$.

Definition 1.2.3. *Given a graph* $\mathcal{G} = (V, E)$, *the* maximum matching problem *is to*

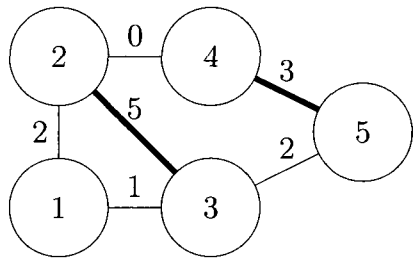


Figure 1.6: An example of a graph with weighted edges. Edges in bold correspond to the maximum weighted matching for this graph.

find a matching with the maximum cardinality out of all matchings in $\mathcal{G}$.

Unlike the maximum independent set problem, the maximum matching problem is in P with a famous polynomial time algorithm due to Edmonds[Edm65b].

Numbers assigned to the vertices or edges of a graph are called *weights*. The weight of a vertex v or an edge e can be found using the functions $\textsc{Weight}(v)$ and $\textsc{Weight}(e)$ respectively. For sets of vertices, the weight is equal to the sum of the weights of the vertices, *i.e.* $\textsc{Weight}(V) = \sum_{v \in V} \textsc{Weight}(v)$ where V is a set of vertices; the weights of sets of edges are the same. For example, Figure 1.6 depicts the graph with the set of edges $E = \{\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{3,5\}, \{4,5\}\}$ from Figure 1.2 modified into a graph with weighted edges by adding a weight function such that $\textsc{Weight}(\{1,2\}) = 2$, $\textsc{Weight}(\{1,3\}) = 1$, $\textsc{Weight}(\{2,3\}) = 5$, $\textsc{Weight}(\{2,4\}) = 0$, $\textsc{Weight}(\{4,5\}) = 3$ and $\textsc{Weight}(\{3,5\}) = 2$.

Definition 1.2.4. *Given a graph* $\mathcal{G} = (V, E)$ *with weighted edges, the* maximum weighted matching problem *is to find a matching* M *such that* $\textsc{Weight}(M)$ *is maximized.*

For example, the graph depicted in Figure 1.6 has a maximum weighted matching $M = \{\{2,3\}, \{4,5\}\}$. This is the only maximum weighted matching for this graph.

Like the maximum matching problem, the maximum weighted matching problem has a polynomial time algorithm, also due to Edmonds[Edm65a]. We define the

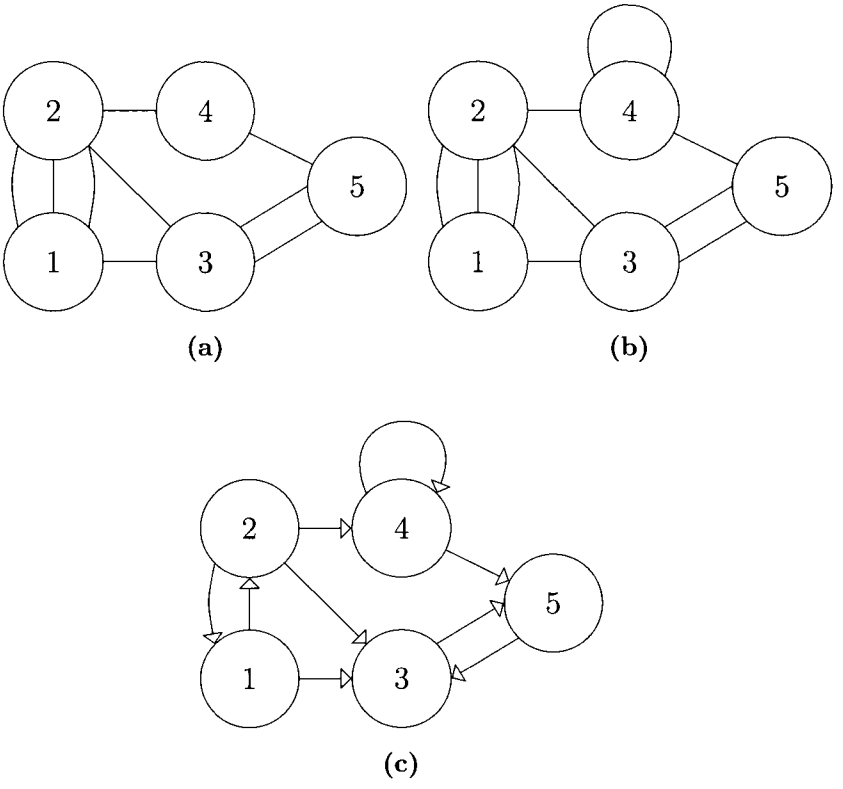


Figure 1.7: (a) An example of a multigraph. (b) An example of a pseudograph. (c) An example of a digraph.

function MAXIMUMWEIGHTMATCHING to compute the maximum weight matching of a graph, however, it is far too complex to give details here.

In addition to weights, graphs can also have extra information attached to their vertices and edges called *labels*. Given a vertex v and an edge e the functions LABEL(v) and LABEL(e) retrieve the label of v and e respectively.

Defining the edges of a graph as a set of 2-element sets has two important implications on the layout of a graph: there can be at most one edge between 2 vertices and a vertex can not have a *loop* (*i.e.* an edge to itself). However, sometimes these properties are desirable and, thus, we introduce two alternatives to a graph. A *multigraph* is the same as a graph except that rather than a set of edges it has a multiset of edges, *i.e.* a multiset of 2-element sets of vertices. Thus, a multiset permits multiple

edges between vertices, *e.g.* the mutligraph $\mathcal{G} = (V, E)$ with $V = \{1, 2, 3, 4, 5\}$ and edges $E = \{\{1, 2\}, \{1, 2\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{3, 5\}, \{3, 5\}, \{4, 5\}\}$, depicted in Figure 1.7a, has three edges between vertices 1 and 2 and two edges between vertices 3 and 5. Similarly, a *pseudograph* is the same as a multigraph except that rather than a multiset of 2-element sets of vertices for edges, it has a multiset of 2-element multisets of vertices. Like a multigraph, a pseudograph permits multiple edges between vertices but a pseudograph also allows loops, *e.g.* Figure 1.7b depicts a pseudograph with multiple edges as well as a loop at vertex 4: the edge $\{4, 4\}$. Both multigraphs and pseudographs use the same notation as sets and all the above definitions, operations, and problems for graphs can be applied to multigraphs and pseudographs.

All the edges in a graph, multigraph or pseudograph are *undirected*, meaning that they are symmetric, *i.e.* if v is adjacent to u then u is adjacent to v. Occasionally *directed* edges are desired. A graph with directed edges, or *digraph*, is defined as a graph except that instead of the edge set being a set of 2-element sets of vertices, the edge set is a set of 2-element sequences of vertices, thus, $(u, v) \neq (v, u)$ causing the edges to be directed. For example, Figure 1.7c depicts digraph $\mathcal{G} = (V, E)$ with vertices $V = \{1, 2, 3, 4, 5\}$ and edges $E = \{(1, 2), (2, 1), (3, 1), (3, 2), (4, 2), (5, 3), (3, 5), (4, 4), (5, 4)\}$. Because sequences are used, digraphs can have at most two edges between vertices, provided the edges have opposite directions.

Since the position of the elements in the sequence is important, we distinguish the elements of an edge $e = (u, v)$ by referring to the element in the first position, u, as the *head* and the element in the second position, v, as the *tail*. In general, all operations that work for graphs work for digraphs. However, the definition of *adjacency* is different: an edge (u, v) means that v is adjacent to u but not *vice versa*, unless there is also an edge (v, u). The degree of digraphs is also slightly different: digraphs have an *in-degree* and an *out-degree*. The *in-degree* of a vertex v, denoted InDegree(v) is number of edges in which v is the head. Similarly, the *out-degree* of a vertex v, denoted OutDegree(v) is the number of edges in which v is the tail.

The *degree* of a digraph is the sum of the in-degree and out-degree.

Similar to digraphs, *directed multigraphs* combines the features of a multigraph with those of a digraph, *i.e.* they can have multiple edges going the same direction between vertices. This is achieved by defining the edge set as a multiset of 2-element sequences. Directed multigraphs are otherwise similar to digraphs.

1.3 Bioinformatics Background

Bioinformatics is the application of computer science practices to biology problems. In order to bridge the gap between the two disciplines, biology concepts must be translated into computer science terms.

1.3.1 Genomes Redefined

We begin by defining a data structure to hold the biological data necessary for our biology problems: the genome. A *genome*, represented using a bold capital letter, *e.g.* a genome $\mathbb{G}$, can be defined as a sequence of *genes*, *e.g.* (a, b, c, d, e). However, there are two problems with this definition.

The first problem with the sequence representation is that this method fails to represent the chromosomes. Since chromosomes are unordered within a genome, we represent them as a set of sequences, *e.g.* $\{(a, b), (c, d, e)\}$. This representation assumes the chromosomes are *circular*. Linear chromosomes can be represented by introducing *caps* for the chromosomes: special characters to indicate the ends of the *chromosomes*. Typically, the same character is used to cap all the chromosomes ends but distinguished from each other by a subscript, *e.g.* $\mathbb{G} = \{(O_1, a, b, O_2), (O_3, c, d, e, O_4)\}$ represents two linear chromosomes, one with two genes and one with three genes. However, each genome must use a different character for capping in order to properly compared, *e.g* $\mathbb{H} = \{(P_1, a, b, c, d, e, P_2)\}$ can be compared with $\mathbb{G}$ from the previous

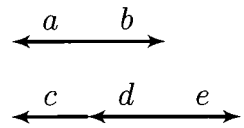


Figure 1.8: A graphical representation of a genome.

example.

The second problem with the sequence representation is that most of the algorithms in which we are interested are NP-hard using this representation[Cap97, ZW06]; there simply is not enough information to solve these problems. The missing information is the strand on which the gene is located. The sequence representation can be easily modified to include this information by assigning a sign to each gene. A gene with a negative sign is on the 3' to 5' strand while a gene with a positive sign is on the 5' to 3' strand, *e.g.* $\{(O_1, -a, +b, O_2), (O_3, -c, -d, +e, O_4)\}$ has two linear chromosomes, one with two genes a and b, where a is on the $3'$ to $5'$ strand and b is on the $5'$ to $3'$ strand, and the other with three genes, c, d and e, where c and d are on the $3'$ to $5'$ strand and e is on the $5'$ to $3'$ strand. Such genomes are called *signed genomes.* Genomes without the information on the strand are called *unsigned genomes.*

A genome can be represented graphically by a *genome digraph* where the edges correspond to genes and the direction of the edges correspond to the orientation of the gene. Vertices are often not depicted, however when they are they merely represent that two extremities of gene are adjacent. For example, Figure 1.8 depicts a genome $\mathbb{G} = \{(O_1, -a, +b, O_2), (O_3, -c, -d, +e, O_4)\}$. As Figure 1.8 clearly demonstrates, chromosomes are easily distinguished, without caps, as they correspond to components in the digraph.

For any genome $\mathbb{G}$, we provide several useful functions. The function GENES($\mathbb{G}$) retrieves the set of genes and only genes (no caps) that make up the genome, *e.g* for $\mathbb{G} = \{(O_1, -a, +b, O_2), (O_3, -c, -d, +e, O_4)\}$, GENES($\mathbb{G}$) $= \{a, b, c, d, e\}$. Observe

the set returned by GENES($\mathbb{G}$) is always unsigned. Similarly, we define CAPS($\mathbb{G}$) to be the set of caps (no genes) in the genome, *e.g.* continuing the previous example, CAPS($\mathbb{G}$) = $\{O_1, O_2, O_3, O_4\}$.

The genes themselves, captured in this representation, are rarely the important piece of information when studying genome rearrangements. Rather the relationship between the genes is important. To capture this information it is easier to break each gene a into its *extremities*: its *head* $\overrightarrow{a}$ and its *tail* $\overleftarrow{a}$. By placing the head before the tail we indicate that the gene is on the 3′ to 5′ strand and *vice versa* for the 5′ to 3′ strand.The function EXTREMITIES($\mathbb{G}$) gives the set of extremities that make up the genome, *e.g.* EXTREMITIES($\mathbb{G}$), where $\mathbb{G}$ is defined as previously, yields $\{\overrightarrow{a}, \overleftarrow{a}, \overleftarrow{b}, \overrightarrow{b}, \overrightarrow{c}, \overleftarrow{c}, \overrightarrow{d}, \overleftarrow{d}, \overleftarrow{e}, \overrightarrow{e}\}$.

The functions GENES and EXTREMITIES apply to any subsequence of a genome as well as a genome. They also apply to any graph that labels its vertices with data from a genome.

Using the extremities of the genes we can compare two genomes with the following graph:

Definition 1.3.1. *Given two genomes* $\mathbb{A}$ *and* $\mathbb{B}$ *with the same set of genes, the* breakpoint graph, *is a multigraph* $\mathcal{G} = (V, E)$ *where*

- V = EXTREMITIES($\mathbb{A}$)∪CAPS($\mathbb{A}$)∪CAPS($\mathbb{B}$) *(recall that* CAPS($\mathbb{A}$) *and* CAPS($\mathbb{B}$) *must be distinct);*
- $E = B \cup G$ *where* B *is the set of* black edges *and* G *is the set of* grey edges. *The set of* black edges *connects all the extremities and caps that are adjacent in* $\mathbb{A}$ *but not extremities of the same gene or a pair of caps. The set of* grey edges *is similarly defined but on genome* $\mathbb{B}$.

We provide the function BREAKPOINTGRAPH($\mathbb{A}, \mathbb{B}$) to construct a breakpoint multigraph from a pair of genomes.

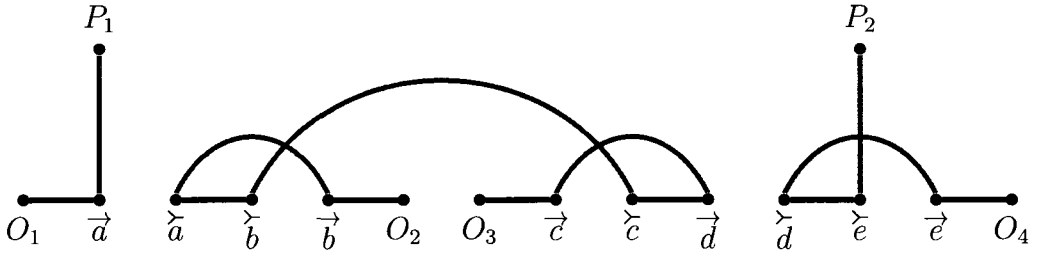


Figure 1.9: Example of a breakpoint graph. Black edges are black and grey edges are grey in the figure.

For example, Figure 1.9 depicts BREAKPOINTGRAPH($\mathbb{G}, \mathbb{H}$) where $\mathbb{G} = \{(O_1, -a, +b, O_2), (O_3, -c, -d, +e, O_4)\}$ and $\mathbb{H} = \{(P_1, -a, -b, +c, -d, -e, P_2)\}$. The set of vertices is $\{\vec{a}, \overleftarrow{a}, \vec{b}, \overleftarrow{b}, \vec{c}, \overleftarrow{c}, \vec{d}, \overleftarrow{d}, \vec{e}, \overleftarrow{e}, O_1, O_2, O_3, O_4, P_1, P_2\}$. The set of black edges is $\{\{O_1, \vec{a}\}, \{\overleftarrow{a}, \overleftarrow{b}\}, \{\vec{b}, O_2\}, \{O_3, \vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}, \overleftarrow{e}\}, \{\vec{e}, O_4\}\}$. The set of grey edges is $\{\{P_1, \vec{a}\}, \{\overleftarrow{a}, \vec{b}\}, \{\overleftarrow{b}, \overleftarrow{c}\}, \{\vec{c}, \vec{d}\}, \{\overleftarrow{d}, \vec{e}\}, \{\overleftarrow{e}, P_2\}\}$.

Generally, in the literature, the cap vertices from $\mathbb{A}$ and $\mathbb{B}$ are merged such that a breakpoint graph contains only cycles, a process called *capping*. Algorithms for capping can be quite complex and depend on the purpose for using the breakpoint graph. Thus, the only time we will use capping is in the context of genome halving and we will briefly discuss how to correctly match the caps in Section 2.1.3.

Most algorithms manipulating the breakpoint graph directly rather than using the sequence representation as the equivalent operations can be performed simply by changing black or grey edges. The flexibility of sets makes these operations very efficient and yet the order provided by the sequence is not lost as the edges preserve it. Thus, the breakpoint graph implies an even better representation for the genome: a set representation.

Given the sequence representation of a genome $\mathbb{G}$, the set representation of $\mathbb{G}$, introduced in [BMS06], is the edge set of BREAKPOINTGRAPH($\mathbb{G}, \emptyset$), *i.e.* the set of black edges. Thus, the elements of the set are pairs of extremities representing neighboring genes. With this representation capping is not necessary, thus, the elements

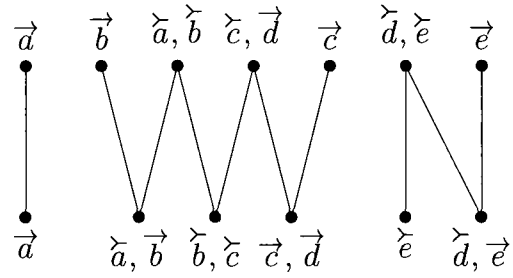


Figure 1.10: An example of an adjacency graph. This is the line graph of the breakpoint graph in Figure 1.9.

of the set can consist of two extremities, called *adjacencies*, or one extremity corresponding to the telomere of a linear chromosome, called a *telomere*. For example, for the genome $\mathbb{G} = \{(O_1, -a, +b, O_2), (O_3, -c, -d, +e, O_4)\}$, the set representation is $\mathbb{G} = \{\{\vec{a}\}, \{\overleftarrow{a}, \overleftarrow{b}\}, \{\vec{b}\}, \{\vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}, \overleftarrow{e}\}, \{\vec{e}\}\}$.

While we defined the set representation in terms of breakpoint graphs, set representations have their own counterpart to breakpoint graphs; a graph to assist with comparing two genomes, called an *adjacency graph*.

Definition 1.3.2. *An* adjacency graph *is the line graph of a breakpoint graph.*

The advantage of an adjacency graph over a breakpoint graph is that, even though it is line graph of a breakpoint graph with caps, it doesn't need any caps or capping as Figure 1.10 illustrates. We provide the function AdjacencyGraph($\mathbb{A}, \mathbb{B}$) as a shortcut for LineGraph(BreakpointGraph($\mathbb{A}, \mathbb{B}$)). Figure 1.10 is the adjacency graph for AdjacencyGraph($\mathbb{G}, \mathbb{H}$) where $\mathbb{G} = \{\{\vec{a}\}, \{\overleftarrow{a}, \overleftarrow{b}\}, \{\vec{b}\}, \{\vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}, \overleftarrow{e}\}, \{\vec{e}\}\}$ and $\mathbb{H} = \{\{\vec{a}\}, \{\overleftarrow{a}, \vec{b}\}, \{\overleftarrow{b}, \overleftarrow{c}\}, \{\vec{c}, \vec{d}\}, \{\overleftarrow{d}, \vec{e}\}, \{\overleftarrow{e}\}\}$.

In our study of genomes it is often advantageous to examine part of a genome in detail. Thus, we often discuss subsets, subsequences and subgraphs of the set, sequence and graph representation of a genome respectively. The functions Extremities and Caps function as normal, but only part of a gene (one extremity) may be present in the subset, subsequence or subgraph. However, Gene returns any gene that has at least one extremity in the subset, subsequence of subgraph respectively. It is im-

portant to observe that these sets, sequences and graphs are not necessarily genomes. To be a *genome* it must fulfill the following requirement:

Remark 1.3.3. *G is a* genome *if and only if for every gene $g \in$ GENES(G), $\vec{g}$ and $\overrightarrow{g}$ are in* EXTREMITIES(G).

If $\mathbb{H}$ is capped then CAPS(G) *must be even and it must be possible to infer a chromosome order from G, which is not always possible,* e.g *no chromosome order can be inferred from $G = \{(O_1, +a, O_2, +b)\}$ since if a is in chromosome then b is not and* vice versa.

1.3.2 Evolutionary Operations

With the above myriad of data structures that can be used to represent genomes we can now apply computer science to biology problems. In particular, within the field of comparative genomics, the goal is to develop some form of *metric* in order to measure the evolutionary *distance* between two genomes. We specifically study the following problem:

Definition 1.3.4 ([FLR+09])**.** *Given a set of genomes and a set of possible evolutionary operations, the* genome rearrangement problem *is to find the shortest sequence of operations transforming those genomes into one another.*

Since there are many different types of evolutionary operations the problem varies depending on which evolutionary operations are being studied. Computing the sequence of evolutionary operations is often called *sorting* by the evolutionary operations, *e.g.* if our set of evolutionary operations consists solely of reversals then the problem of finding the sequence of reversals is called sorting by reversals or reversal sorting. Similarly, as a shortcut when discussing the minimal distance under a set of evolutionary operations, we say the *distance* of the evolutionary operations, *e.g.* reversal distance.

As mentioned in Section 1.1, evolutionary operations are reversals, translocations, fusions, fissions, transpositions, insertions, deletions, mutations and duplications. One of the primary goals of comparative genomics is to develop an algorithm that solves the genome rearrangement problem that considers all the above operations. However, there are presently no algorithms that include all the evolutionary operations. This thesis is primarily concerned with the duplication operation, ideally in the company of other operations in order to further the goal of achieving an algorithm that considers all evolution operations. Thus, in this section we briefly survey the existing algorithm but withhold discussing the duplication operation until Section 1.3.3. For a more complete survey we refer the reader to [FLR+09].

An *insertion* is the addition of a sequence of genes to the genome at some arbitrary location within a chromosome, *e.g.* inserting $-f$, $-g$, $-h$ to the 3' strand of the genome $\{(O_1, -a, +b, O_2), (O_3, -c, -d, +e, O_4)\}$ between genes d and e creates $\{(O_1, -a, +b, O_2), (O_3, -c, -d, \underline{-f, -g, -h,} +e, O_4)\}$. A *deletion* is the removal of a sequence of genes from a chromosome, *e.g.* deleting d from $\{(O_1, -a, +b, O_2), (O_3, -c, \underline{-d,} +e, O_4)\}$ results in $\{(O_1, -a, +b, O_2), (O_3, -c, +e, O_4)\}$. There is a duality between insertion and deletion where, when comparing two genomes, an insertion in one genomes can be viewed as a deletion in the other and *vice versa.* Hence, computationally insertions and deletions cannot be distinguished, they are a matter of perspective, so we often refer to them as *insertions/deletions.*

A *mutation* occurs when a gene is replaced by another gene, *e.g.* the gene $-d$ in $\{(O_1, -a, +b, O_2), (O_3, -c, \underline{-d,} +e, O_4)\}$ could mutate into the gene $+f$ resulting in $\{(O_1, -a, +b, O_2), (O_3, -c, \underline{+f,} +e, O_4)\}$.

Insertions/deletions and mutations are generally considered together in what is known as the *longest common subsequence problem*, a famous problem with many practical applications. The earliest consideration of this problem is likely by [Lev65] in a non-biological context. [NW69] gave the first exact polynomial time algorithm to solve the problem, an $O(nm)$ time dynamic programming algorithm, where n

and m are the sizes of the sequences being compared. Later, [WSB76] refined the algorithm allowing for the simultaneous insertion/deletion of multiple elements (in our case, genes) at once. Unfortunately, the algorithm in [WSB76]is more complex, running in of $O(n^2m)$ time, though in some cases it can be improved to $O(nm)$ time[Got82, Wat83]. All these algorithms work generically on any sequence with biological applications focused on either amino acid sequences (DNA or RNA) and protein sequences. For our problem of insertions and deletions on genomes, the problem is simpler[SG89] with [San92] proposing a simple $O(n + m)$ time algorithm to compute the number of insertions/deletions but it does not account for mutations and, like the [NW69] algorithm, it does not account for the simultaneous insertion/deletion of multiple elements.

Reversals are performed by reversing the order of all the elements of the sequence between two points within a chromosome, *e.g* reversing the underlined part of $\{(O_1, -a, +b, O_2), (O_3, -c, \underline{-d, +e,} O_4)\}$ causes it to become $\{(O_1, -a, +b, O_2), (O_3, -c, \underline{-e, +d,} O_4)\}$. While it might be biologically possible, for our purposes a reversal cannot break up a gene. By definition, a reversal cannot break up a chromosome (*i.e.* a reversal cannot include a cap).

For 17 years, insertions/deletions and mutations were the only evolutionary operation studied. It wasn't until [WEHM82] that anybody proposed another evolutionary operation: the reversal. However, reversals went ignored until [KS95] who developed the first exact algorithm, albeit an exponential time branch and bound algorithm. [KS95] made two correct conjectures: that an algorithm for sorting by reversals could be refined to include transpositions and translocations (realized by [YAF05]) and that sorting by reversals is NP-hard (proven by [Cap97]). However, as it turns out the problem is only NP-hard on unsigned genomes. The signed version of the problem, proposed in [BP96], does have an algorithm that runs in $O(n^4)$ for computing the sequence of reversals and $O(n^2)$ for computing the distance, which was found by [HP99]. The time complexity has since improved to its current status

where the distance can be computed in $O(n)$ time[BMY01], while the sequence of rearrangements can be computed in $O(n^{3/2})$ time[FLR+09].

A *translocation* is essentially the same operation as a reversal except that it must break up a chromosome by including a pair of caps in different chromosomes, *e.g.* $\{(O_1, -a, +b, \underline{O_2}), (O_3, -c, -d, +e, O_4)\}$ becomes $\{(O_1, -a, +b, \underline{+c, O_3}), (O_2, -d, +e, O_4)\}$. Whole chromosomes can be ignored when determining the adjacency of caps, as reversing an entire chromosome is meaningless from a biological perspective since chromosomes are unoriented. Similarly, the first and last cap in the sequence can be considered adjacent as chromosomes are unordered. *Fusions* and *fissions* are special cases of translocations. A *fusion* occurs when a translocation removes the caps between two chromosomes resulting in a chromosome with the genes of both original chromosomes and another chromosome without any genes, called a *null chromosome*, *e.g.* $\{(O_1, -a, +b, \underline{O_2}), (O_3, -c, -d, +e, O_4)\}$ becomes $\{(O_1, -a, +b, \underline{-e, +d, +c, O_3}), (O_2, O_4)\}$, observe that O_2 and O_4 form a null chromosome. Conversely, a *fission* occurs when translocation splits a chromosome by moving some of its genes to a null chromosome, *e.g.* $\{\underline{(O_1, -a, +b,} -e, +d, +c, O_3), (O_2, \underline{O_4})\}$ becomes $\{\underline{(O_4,} -a, +b, -e, +d, +c, O_3), (O_2, \underline{-b, +a, O_1})\}$. To facilitate fusions and fissions it is often necessary for algorithms to introduce null chromosomes into a genome.

[KR95] formulated the idea of using translocations to solve the genome rearrangement problem. [Han96] with some corrections by [BMS05] produced the first polynomial time exact algorithm, an $O(n^3)$ time algorithm for computing both the sequence of translocations and the distance. The current best algorithm for computing the sequence of translocations, due to [OFS06], runs in $O(n^{3/2}\sqrt{\log(n)})$, while the best algorithm for computing the distance, due to [LQWZ04], runs in linear time. However, the genome rearrangement problem using translocations on an unsigned genome is NP-hard[ZW06].

Reversals are limited to single chromosomes while translocations are limited to

multiple chromosomes. Thus, it is only natural to want to combine the two operations to get results that are both more biologically accurate but which can also handle a wider variety of input. This variation of the genome rearrangement problem was introduced and solved in $O(n^4)$ time for both the distance and the sequence of rearrangements) on signed genomes by [HP95]. However, many errors were found in the algorithm but over many iterations an error-free algorithm that computes the sequence of rearrangements in $O(n^2)$ time and the distance in $O(n)$ time was produced[Tes02, OFS03, JN07]. [BMS08] greatly simplified the algorithm by adopting the adjacency graph and thereby avoiding capping, the source of most of the complexity and errors. The genome rearrangement problem with reversals and translocations and its counterpart for unichromosomal genomes, the genome rearrangement problem with reversals only, are commonly called the *HP distance problem* since Hannenhalli and Pevzner provided the first polynomial time algorithm for both of those instances of the problem[HP95, HP99]. Given two genomes $\mathbb{A}$ and $\mathbb{B}$ the function $\text{DISTANCE}_{HP}(\mathbb{A}, \mathbb{B})$ returns the HP distance between $\mathbb{A}$ and $\mathbb{B}$.

A *transposition* involves swapping two contiguous intervals within a chromosome, *e.g* swapping the two underlined intervals (O_1, a, b, O_2, O_3, $\underline{c}$, $\underline{d, e}$, O_4) produces (O_1, a, b, O_2, O_3,$\underline{d, e}$, $\underline{c}$, O_4). Since transpositions don't change the strand on which the gene is located, it only makes sense to consider the genome rearrangement problem using only transpositions on unsigned genomes. Introduced by [BP98], the complexity of the genome rearrangement problem using only transpositions has yet to be determined[FLR+09]. However numerous approximation algorithms do exist[FLR+09], the best of which is the 11/8-approximation algorithm due to [EH06]. But the approximation algorithm which is of the most interest is the algorithm that uses the *block-interchange* operation to approximate the transposition operation. A *block-interchange* is a generalization of a transposition[Chr96] and the approximation algorithm that uses the block-interchange operation to simulate the transposition operation is 2-approximation algorithm[LLCT05].

A *block-interchange* occurs when any two non-overlapping intervals of the chromosome are swapped, not necessarily contiguous ones, *e.g.* the underlined intervals can be swapped converting $\{(O_1, a, b, O_2), (O_3, \underline{c}, d, \underline{e}, O_4)\}$ into $\{(O_1, a, b, O_2), (O_3, \underline{e}, d, \underline{c}, O_4)\}$, even though the two intervals are not contiguous. [Chr96] introduced the block-interchange operation and provided an $O(n^2)$ time algorithm to compute the sequence of block-interchanges as well as an $O(n)$ time algorithm to compute the block-interchange distance. Better algorithms have since been found for computing the sequence of block-interchanges culminating in an $O(n + b \log b)$ time algorithm, where n is the number of genes and b is the number of block interchanges, proposed in [HHTL09].

Because of the limitations of transpositions (or block-interchanges), solutions to the genome rearrangement problem are limited to sets of unichromosomal unsigned genomes. Thus, it seems only natural to combine transpositions with either reversals or translocations to broaden the set of genomes. For transpositions and reversals numerous heuristics and approximation algorithms have been explored[FLR+09]. On the other hand, an $O(n^2)$ time algorithm for the genome rearrangement problem with block-interchanges and reversals has been found[MM07]. Moreover, combining transpositions and translocations has also been successful with [DM01] producing an exact $O(n^2)$ time algorithm for transpositions, fusions and fissions provided that transpositions are weighted twice that of fusions and fissions. [LHWC06] also produced an exact $O(n^2)$ time algorithm for block-interchanges, fusions and fissions. However, all of these approaches are limited to only circular chromosomes; so far no algorithm for any of these pairs of operations has been found that handles linear chromosomes. Thus, it is perhaps surprising that better algorithms exist for handling block-interchanges, reversals, translocations, fusions and fissions on signed multichromosomal genomes with any combination of linear and circular chromosomes.

[YAF05] observed that all the above operations are fundamentally combinations of two elemental operations: *cuts* and *joins*. A *cut* is an operation that splits an

adjacency into two telomeres, *i.e.* a cut on $\{x, y\}$ creates $\{x\}$ and $\{y\}$. A *join* is the reverse operation, it is the union of two telmores into an adjacency, *i.e.* $\{x\}$ and $\{y\}$ are joined into $\{x, y\}$. More over, [YAF05] showed that with at most two cuts and two joins any of the above operations could be simulated, though not necessarily with equal weights. Hence, the *double cut and join* operation: an operation that combines reversals, translocations, fusions, fissions and block-interchanges. Formally:

Definition 1.3.5 ([BMS06]). *The* double cut and join (DCJ) *operation acts on two adjacencies or telomeres u and v of a genome in one of the following three ways:*

- *If both $u = \{w, x\}$ and $v = \{y, z\}$ are adjacencies, these are replaced by the two adjacencies $\{w, y\}$ and $\{x, z\}$ or by the two adjacencies $\{w, z\}$ and $\{x, y\}$.*
- *If $u = \{w, x\}$ is an adjacency and $v = \{y\}$ is a telomere, these are replaced by $\{w, y\}$ and $\{x\}$ or by $\{x, y\}$ and $\{w\}$.*
- *If both $u = \{w\}$ and $v = \{x\}$ are telomeres, these are replaced by $\{w, x\}$.*

In addition, as an inverse of the last case, a single adjacency $\{w, x\}$ can be replaced by two telomeres $\{w\}$ and $\{x\}$.

The algorithm for DCJ proposed by [YAF05] runs in $O(n)$ time to compute the distance and $O(nb)$ time to compute the sequence of operations, where n is the number of genes and b is the number of block-interchanges. [YAF05] predicted, and [BMS06] proved, that if some of the constraints of the [YAF05] algorithm were relaxed then the sequence of operations could be computed in $O(n)$ time as well. Algorithm 1.2 depicts the pseudo-code for the faster [BMS06] algorithm. It sorts two genomes using DCJ in linear time and returns a sequence of operations that sorts genome $\mathbb{A}$ into genome $\mathbb{B}$. Thus, the *distance* between the two genomes is the cardinality of S. [BMS06] also provide an equation to compute the distance using an adjacency graph ([YAF05] provides a similar equation for breakpoint graphs):

$$|\mathrm{S}| = |\textsc{Genes}(\mathbb{A})| - c - \frac{i}{2} \tag{1.3.6}$$

Algorithm 1.2: DCJSORTING

Input: Two genomes $\mathbb{A}$ and $\mathbb{B}$.
Output: A sequence S of genomes depicting the progression of reversals, translocations and block interchanges required to sort $\mathbb{A}$ into $\mathbb{B}$.

```
S ← ∅
foreach adjacency (α = {x, y}) ∈ 𝔹 do
    let β be the element in 𝔸 that contains x
    let γ be the element in 𝔸 that contains y
    if β ≠ γ then
        replace β and γ in 𝔸 by {x, y} and (β \ {x}) ∪ (γ \ {y})
    end
    append 𝔸 to S
end
foreach telomere (α = {x}) ∈ 𝔹 do
    let β be the element in 𝔸 that contains x
    if β is an adjacency then
        replace β in 𝔸 by {x} and β \ {x}
    end
    append 𝔸 to S
end
return S
```

where c is the number of cycles and i is the number of *odd paths* (paths with an odd number of edges) in ADJACENCYGRAPH($\mathbb{A}, \mathbb{B}$). For example, the adjacency graph depicted in Figure 1.10 has two odd paths, no cycles and five genes. Equation 1.3.6 predicts 4 operations are required. Running Algorithm 1.2 produces:

$$\begin{aligned}
\mathbb{G} &= \{\{\vec{a}\},\underline{\{\overrightarrow{a},\overleftarrow{b}\}},\underline{\{\vec{b}\}},\{\vec{c}\},\{\overleftarrow{c},\vec{d}\},\{\overleftarrow{d},\overleftarrow{e}\},\{\vec{e}\}\}\\
&= \{\{\vec{a}\},\{\overrightarrow{a},\vec{b}\},\underline{\{\overleftarrow{b}\}},\{\vec{c}\},\underline{\{\overleftarrow{c},\vec{d}\}},\{\overleftarrow{d},\overleftarrow{e}\},\{\vec{e}\}\}\\
&= \{\{\vec{a}\},\{\overrightarrow{a},\vec{b}\},\{\overleftarrow{b},\overleftarrow{c}\},\underline{\{\vec{c}\}},\underline{\{\vec{d}\}},\{\overleftarrow{d},\overleftarrow{e}\},\{\vec{e}\}\}\\
&= \{\{\vec{a}\},\{\overrightarrow{a},\vec{b}\},\{\overleftarrow{b},\overleftarrow{c}\},\{\vec{c},\vec{d}\},\underline{\{\overleftarrow{d},\overleftarrow{e}\}},\underline{\{\vec{e}\}}\}\\
\mathbb{H} &= \{\{\vec{a}\},\{\overrightarrow{a},\vec{b}\},\{\overleftarrow{b},\overleftarrow{c}\},\{\vec{c},\vec{d}\},\{\overleftarrow{d},\vec{e}\},\{\overleftarrow{e}\}\}
\end{aligned}$$

a reversal, translocation, fusion and reversal: four operations.

The function DISTANCE$_{DCJ}$($\mathbb{A}, \mathbb{B}$) computes the DCJ distance between genomes

$\mathbb{A}$ and $\mathbb{B}$. Traditionally, algorithms for computing the distance have been faster than algorithms for computing the sequence of operations. However, DCJ is so fast that the two methods are equivalent as constructing an adjacency graph and finding the cycles and paths within it also takes linear time. Thus, $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B})$ could be implemented as $|\text{DCJSORTING}(\mathbb{A}, \mathbb{B})|$ or by finding the cycles and paths of the adjacency graph; both implementations are equivalent.

Theorem 1.3.7. *Given genomes* $\mathbb{A}$ *and* $\mathbb{B}$, $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B})$ *is a metric.*

Proof. In order to be a *metric* $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B})$ must always satisfy the following three properties: $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) \geq 0$ called the *positivity* property, $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) = \text{DISTANCE}_{DCJ}(\mathbb{B}, \mathbb{A})$ called the *symmetry* property and the *triangle inequality* property where $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{C}) \leq \text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) + \text{DISTANCE}_{DCJ}(\mathbb{B}, \mathbb{C})$. We consider the three cases below:

Positivity There are two edges in $\text{ADJACENCYGRAPH}(\mathbb{A}, \mathbb{B})$ for every gene (one for each extremity) in genome $\mathbb{A}$ (or $\mathbb{B}$). All the cycles of an adjacency graph have an even number of edges. Since all vertices in an adjacency graph have a degree of at most two, an edge cannot be in both a cycle and a path. If E is the set of edges in the adjacency graph then let e be the number of edges that are part of cycles and let f be the number of edges that are part of odd paths. We can now rewrite Equation 1.3.6 in terms of edges: $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) = \frac{|E|}{2} - \frac{e}{2} - \frac{f}{2}$. If all edges in the adjacency graph form cycles and odd paths then $e + f = |E|$, hence, $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) = \frac{|E|}{2} - \frac{|E|}{2} = 0$. If any edges form even paths then it follows that $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) > 0$. Therefore, $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) \geq 0$.

Symmetry $\text{ADJACENCYGRAPH}(\mathbb{A}, \mathbb{B})$ is a bipartite graph where the vertices be can partitioned into $U = \mathbb{A}$ and $V = \mathbb{B}$. Switching the two sets such that $U = \mathbb{B}$ and $V = \mathbb{A}$ doesn't change the edge or gene sets at all, thus, there are the same number of cycles and paths in $\text{ADJACENCYGRAPH}(\mathbb{B}, \mathbb{A})$. It follows from Equation 1.3.6 that $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) = \text{DISTANCE}_{DCJ}(\mathbb{B}, \mathbb{A})$.

Triangle Inequality Assume towards contradiction that $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{C}) > \text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) + \text{DISTANCE}_{DCJ}(\mathbb{B}, \mathbb{C})$. Let $S = \text{DCJSORTING}(\mathbb{A}, \mathbb{B})$ and $S' = \text{DCJSORTING}(\mathbb{B}, \mathbb{C})$. Clearly $S'' = S||S'$ is a sequence of DCJ operations that transforms $\mathbb{A}$ into $\mathbb{C}$. From our assumption $|S''| < \text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{C})$. However, this contradicts [BMS06, Lemma 2] (that Equation 1.3.6 is minimal). Therefore, $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{C}) \leq \text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) + \text{DISTANCE}_{DCJ}(\mathbb{B}, \mathbb{C})$. □

Since the double cut and join operation was introduced, it has become very popular thanks to its simplicity and universality; no other distance metric encompasses such a large number of operations. Capitalizing on its simplicity, [BMS08] modified the DCJ algorithm in order to eliminate block interchanges, framing HP distance in terms of DCJ distance. Capitalizing on the universality of cuts and joins, many[AP07b, FaM09, MS09] have experimented with their frequency but two cuts and two joins remains the most popular. However, while DCJ is nearly all encompassing, efficient and powerful, there is growing interest to add the remaining four evolution operations it is missing and create a true universal operation.

All evolutionary operations discussed above can be divided into two categories, *rearrangement operations* (reversals, translocations, fusions, fissions, transpositions and block-interchanges) and *alignment operations* (insertions, deletions and mutations), with different algorithmic approaches (greedy algorithms *vs* dynamic programming algorithms respectively). It follows that there are two schools of thought on combining these sets of operations: those that add rearrangement operations to the dynamic programming algorithms and those that add alignment operations to the greedy algorithms.

After [WEHM82] first proposed studying the reversal operation, the first work on reversals focused on adding to the only existing algorithms of the type at the time: the dynamic programming algorithms for the alignment operations. These attempts

were surprisingly successful with [SW92] creating a polynomial-time algorithm that computes the optimal reversals, insertions/deletions and mutations distance, three years before an algorithm that computes the reversal distance was found. Their algorithm is not without limitations: it only works in certain instances of the problem and it runs in $O(n^2m^2)$ time and $O(n^2m^2)$ space. Around the same time, [San92], working on the idea the insertions/deletions on genomes are simpler than on other sequences, attempted to combine reversals, insertions/deletions and transpositions but was only able to develop a heuristic. Beyond these two early efforts no other attempt has been made until recently when several improvements to the [SW92] algorithm have been made[GWN+03, VAL06]. The current best algorithm, from [VAL06], runs in $O(n^3)$ time and $O(nm)$ space, although it is still limited to certain instances of the problem.

The first work on adding alignment operations to the greedy algorithms is due to [EM00] who added deletions (her solution is asymmetric) to reversals. Her algorithm runs in $O(n^2)$ time, for computing the distance, but it was later improved to $O(n)$ time by [LMB03]. Despite these successes, adding symmetry (*i.e.* adding insertions) to the algorithm has proven difficult although [EM00] successfully created an approximation algorithm. The primary difficulty with adding insertions is that, when unrestricted, an insertion can be a duplication (if another copy of an existing genome is inserted). The [EM00] approximation algorithm avoided this problem, forbidding duplications, but an attempt by [MSM03] to allow duplications resulted in another approximation algorithm. Following in the footsteps of [EM00], [QLLX06] developed an approximation algorithm for sorting by translocations and deletions. [YF08] developed an interesting heuristic algorithm that adds insertion, deletions and duplications to DCJ. In this thesis, we focus on a few promising strategies to handle gene duplication under DCJ and HP distance.

1.3.3 On Gene Duplications

All the above rerarrangement algorithms make the assumption that there is only one copy of each gene in a genome. This, however, is a biologically unrealistic assumption as most genomes have genes with multiple copies, called gene *families*. A genome with a duplicated gene is called *ambiguous*, a name that implies the computation problems with such genomes. The ambiguity of genomes with duplicates is a problem both to represent them, the usual data structures for genomes is insufficient for the task, and with comparing them.

There is no ideal data structure for handling ambiguous genomes. While the sequence representation naturally handles duplicates gene, as sequences can have multiple copies of the same element, breakpoint graphs and the set representation can't accommodate it. But even with the sequence representation it is difficult to keep track of which gene is which while performing most evolutionary operations. Thus, regardless of which data structure is being used to represent the genome, the extremities of duplicated genes are usually arbitrarily distinguished by subscripts, *e.g.* if there are two copies of a gene a in genome $\mathbb{G}$ we can distinguish them as a_1 and a_2, thus, $\mathbb{G} = \{\{\overrightarrow{a_1}\}, \{\overrightarrow{a_1}, \overleftarrow{b}\}, \{\overrightarrow{b}\}, \{\overrightarrow{a_2}\}, \{\overleftarrow{a_2}, \overrightarrow{c}\}, \{\overleftarrow{c}, \overrightarrow{d}\}, \{\overleftarrow{d}, \overleftarrow{e}\}, \{\overrightarrow{e}\}\}$. Genomes that use this representation for duplicated genes are called *ordered genomes*. For ordered genomes, the functions GENES and EXTREMITIES return all copies of each gene, *e.g.* continuing the previous example, $\text{GENES}(\mathbb{G}) = \{a_1, a_2, b, c, d, e\}$ and $\text{EXTREMITIES}(\mathbb{G}) = \{\overrightarrow{a_1}, \overleftarrow{a_1}, \overrightarrow{a_2}, \overleftarrow{a_2}, \overrightarrow{b}, \overleftarrow{b}, \overrightarrow{c}, \overleftarrow{c}, \overrightarrow{d}, \overleftarrow{d}, \overrightarrow{e}, \overleftarrow{e}\}$.

For ambiguous genomes it is helpful to know the members of a gene family; thus, we define the function $\text{FAMILY}(a, \mathbb{A})$ to return all the members of the gene family containing a, including a itself, in the genome $\mathbb{A}$, *e.g.* $\text{FAMILY}(a_1, \mathbb{G})$, where $\mathbb{G}$ is the genome from the previous example, returns the set $\{a_1, a_2\}$. Given an extremity instead of a gene, the function family returns all the extremities in the family, *e.g.* $\text{FAMILY}(\overleftarrow{a_1}, \mathbb{G}) = \{\overrightarrow{a_1}, \overleftarrow{a_1}, \overrightarrow{a_2}, \overleftarrow{a_2}\}$. For unambiguous genomes or for genes without

duplicates (*singletons*) within an ambiguous genomes, the function FAMILY can be defined as well, however, in this case $\text{FAMILY}(a, \mathbb{A}) = \{a\}$ where a is a singleton in genome $\mathbb{A}$.

Ordered genomes are not, however, ideal for every situation. For ordered genomes we define equality such that $a_i = a_j$ even when $i \neq j$. It is also possible define other fundamental operations such as union and set minus. However, it is not possible to define the intersection of any sets that contain a_i and a_j since it is not possible to determine the subscript of the intersection, *i.e.* is $\{a_i\} \cap \{a_j\}$ equal to $\{a_i\}$ or $\{a_j\}$ when $i \neq j$? Unfortunately, as will be described in Section 2.1.1, the intersection is important to the work performed in Chapters 2 and 3. Thus, we provide another representation: an *unordered genome* is a relaxing of the set representation of a genome to be a multiset of multiset rather than a set of sets. The functions GENES, EXTREMITIES and FAMILY must return multisets under this representation, but otherwise behave the same as on ordered genomes, *e.g.* given an unordered genome $\mathbb{G} = \{\{\vec{a}\}, \{\overleftarrow{a}, \overrightarrow{b}\}, \{\vec{b}\}, \{\vec{a}\}, \{\overleftarrow{a}, \vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}, \overrightarrow{e}\}, \{\vec{e}\}\}$, $\text{GENES}(\mathbb{G}) = \{a, a, b, c, d, e\}$, $\text{EXTREMITIES}(\mathbb{G}) = \{\vec{a}, \overleftarrow{a}, \vec{a}, \overleftarrow{a}, \vec{b}, \overrightarrow{b}, \vec{c}, \overleftarrow{c}, \vec{d}, \overrightarrow{d}, \vec{e}, \overleftarrow{e}\}$ and $\text{FAMILY}(a, \mathbb{G}) = \{a, a\}$. While the intersection between two unordered genomes is well defined, it still has problems.

The problem with unordered genomes is that they are unordered, *i.e.* it is no longer possible to infer an order to the genes. For unordered genomes, breakpoint graphs and adjacency graphs lose their essential property that all vertices have a degree of at most 2, thus, the distance between two unordered genomes cannot be computed. Hence, we would like to use both representations; we would like to modify unordered genomes and analyze ordered genomes. Since the two representations are not equivalent, we distinguish the unordered genomes with a tilde, *i.e.* an unordered genome $\tilde{\mathbb{A}}$.

While converting from ordered genomes to unordered genomes, performed by the function $\text{UNORDER}(\mathbb{G})$, is easy, the reverse is difficult. This is because an unordered

genome with even one duplicated gene cannot be uniquely transformed back into an ordered genome: the information is permanently lost. There is, however, one important exception: a *perfect genome.*

A set $\mathbb{A}$ is *perfect* if and only if for all $\alpha, \beta \in A$ if, for an extremity x, $x \in \alpha$ and $x \in \beta$ then $\alpha = \beta$ and neither α nor β is of the form $\{x, x\}$; *e.g.* the genome $\tilde{\mathbb{G}}$ with duplicated genes a and b, $\tilde{\mathbb{G}} = \{\{\vec{a}\}, \{\overleftarrow{a}, \vec{b}\}, \{\overleftarrow{b}\}, \{\vec{a}\}, \{\overleftarrow{a}, \vec{b}\}, \{\overleftarrow{b}\}, \{\vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}, \overleftarrow{e}\}, \{\vec{e}\}\}$, is perfect while $\tilde{\mathbb{H}}$, also with duplicated genes a and b, defined $\tilde{\mathbb{H}} = \{\{\vec{a}\}, \{\overleftarrow{a}, \overleftarrow{a}\}, \{\vec{a}\}, \{\overleftarrow{b}\}, \{\vec{b}\}, \{\vec{b}\}, \{\overleftarrow{b}, \vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}, \overleftarrow{e}\}, \{\vec{e}\}\}$, is not perfect because $\{\overleftarrow{b}\} \neq \{\overleftarrow{b}, \vec{c}\}$ but both contain $\overleftarrow{b}$, and $\{\overleftarrow{a}, \overleftarrow{a}\}$ is of the form $\{x, x\}$ where $x = \overleftarrow{a}$. A set that is not perfect is called *rearranged.* Perfect and rearranged are similarly defined for sequences.

Clearly, any perfect unordered genome can be uniquely ordered. However, rarely is a genome considered on its own and more often it will be considered in the context of a known ordered rearranged genome. In this case, the perfect genome cannot be ordered arbitrarily as it may increase the distance. However, it can still be uniquely ordered such that the distance is minimized. Unfortunately, this is context specific, we can provide no generic algorithm to reorder the genome. Sections 2.1.6 and 3.1.4 describe how to solve this problem in the context of the genome halving problem and genome aliquoting problem respectively.

When comparing two genomes it is essential to ensure that they have the same set of genes. If a set A has at least one copy of every extremity of a set B then we say that A *covers* B.

Duplications have been studied in great detail in the context of phylogenetic trees. However, this is a different problem all together than the genome rearrangement problem and, while the two problems are closely related, it is beyond the scope of this thesis. We refer interested readers to a quick survey in [EM05].

The challenge of any genome rearrangement algorithm attempting to handle duplicate genes is determining the relationship between the similar genes in the two

genomes. When each genome has one copy of a gene the relationship between the two genes is easy: they both descend from the same ancestor, *i.e.* they are *orthologs*. But, for example, if genome $\mathbb{A}$ has two copies of a gene a, say a_1 and a_2, and another genome $\mathbb{B}$ has one copy of gene a, then what is the relationship between the genes? Clearly, either a_1 or a_2 is an ortholog of a, but which? It seems impossible to tell without trying all the cases. Even when all gene families in both genomes are the same size, *i.e.* the genomes are *balanced*, this problem persists. In spite of such seemingly insurmountable challenges, progress has been made.

The first attempt to handle gene duplications in the context of the genome rearrangement problem was by [San99]. Rather than attempt to directly compute the duplication distance between two genomes, or add it to a list of known evolutionary operations (at the time, reversals and translocations), [San99] attempted an indirect approach introducing the *exemplar problem*. Formally,

Definition 1.3.8. *The* exemplar problem *is defined as follows: given two genomes* $\mathbb{G}$ *and* $\mathbb{H}$ *with duplicated genes and distance function* $\text{DISTANCE}_{SORT}(\mathbb{G}, \mathbb{H})$, *construct two genomes with only one copy of each gene,* $\mathbb{G}'$ *and* $\mathbb{H}'$, *by removing all other copies from* $\mathbb{G}$ *and* $\mathbb{H}$ *respectively such that* $\text{DISTANCE}_{SORT}(\mathbb{G}', \mathbb{H}')$ *is minimized.*

The exemplar problem makes the assumption that the two genomes with duplicated genomes must have had ancestors without duplicate genomes in the past. This nicely avoids the problem of duplicates by ignoring the distance between the genomes and their unambiguous ancestors and instead taking advantage of existing algorithms for other evolutionary operations. Unfortunately, the problem was proven NP-hard for reversal distance by [Bry00]. Thus, at least for reversal distance but probably for other distances as well, the exemplar problem does not appear to be the ideal solution.

In an effort to find a more computationally tractable solution to gene duplications, [TM03, AFRV06] have tried relaxing the definition of the exemplar problem to

construct a pair of balanced ancestral genomes, rather than unambiguous ones. Unfortunately, this method has two formidable problems. First, as it turns out, it is no more tractable then the exemplar problem[FLR+09]. Second, even if it were tractable, there have been fewer attempts at comparing balanced genomes and reversal distance, both signed[RSW05] and unsigned[RSW05, CI01], transposition distance[RSW05], block-interchange distance[FLR+09] and DCJ[TZS09] have all been shown to be NP-hard on certain instances of the problem.

Given the problems with the indirect method and with comparing balanced genomes, it is perhaps surprising that the attempts at directly adding duplications to existing genome rearrangement algorithms[EM00, MSM03, YF08], discussed in the previous section, have produced a 10-approximation[MSM03] algorithm and a pair of heuristics[EM00, YF08] even though it remains to be seen if an exact algorithm is possible. But, despite the failures of both the indirect and direct methods to produce an exact algorithm, exact algorithms for the genome rearrangement problem with duplications do exist.

Closely related to the exemplar problem is the *rearrangement-duplication problem.* Instead of constructing ancestors with minimal distance from two genomes, the *rearrangement-duplication problem* constructs an ancestor with minimal distance to the original ambiguous genome. Formally,

Definition 1.3.9. *Given a genome* $\mathbb{A}$ *with duplicated genes and a rearrangement distance function* $\text{DISTANCE}_{SORT}(\mathbb{A}, \mathbb{B})$, *the* rearrangement-duplication problem *is to construct a genome* $\mathbb{B}$ *from which an unambiguous genome can be constructed using* $\text{DISTANCE}_{DUP}(\mathbb{A}, \mathbb{B})$ *duplication operations, such that* $\text{DISTANCE}_{SORT}(\mathbb{A}, \mathbb{B}) + \text{DISTANCE}_{DUP}(\mathbb{A}, \mathbb{B})$ *is minimized.*

At first the rearrangement-duplication problem doesn't seem to be a solution at all; how can the distances between the original genome and its ancestor be compared if the original has duplications and the ancestor does not? However, the fact that the

ancestor is constructed makes a difference as it overcomes the key limitation of any duplication algorithm: matching the duplicates. When constructing the ancestral genome, the algorithm, rather than nature, determines which gene of the ancestral genome corresponds to which gene in the original genome. Thus, it can record this information allowing the genes to be easily distinguished which then allows any of the algorithms discussed in the previous section to be used to determine the distance. Thus, the only challenge is ensuring that the constructed ancestral genome's distance is minimized.

The earliest attempts at this problem were very conservative focusing on the specific case where the ancestral genome is a *tetraploid.* This scenario makes a few convenient assumptions. First, it assumes that the ancestral genome is not only a tetraploid but where all the chromosomes of the tetraploid have exactly one copy of each gene. Second, it assumes no insertion/deletion, mutation or duplication events have occurred since the tetraploidization event, only rearrangement events. The resulting rearranged tetraploid is called a *duplicated genome* [1] and it has the important property that there are exactly two copies of each gene, *e.g.* the genome $\mathbb{G} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \overset{\succ}{b_1}\}, \{\vec{b_1}, \vec{d_1}\}, \{\vec{d_1}, \overset{\succ}{e_1}\}, \{\vec{e_1}, \vec{b_2}\}, \{\vec{b_2}\}, \{\vec{a_2}\}, \{\overset{\succ}{a_2}, \vec{c_1}\}, \{\overset{\succ}{c_1}, \overset{\succ}{c_2}\}, \{\vec{c_2}, \vec{d_2}\}, \{\overset{\succ}{d_2}, \overset{\succ}{e_2}\}, \{\vec{e_2}\}\}$ is an example of a duplicated genome. This problem of constructing a tetraploid from a duplicated genome is called the *genome halving problem.* For example, given the above $\mathbb{G}$, a solution to the genome halving problem would produce a tetraploid that is a possible ancestor of $\mathbb{G}$, such as $\{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \vec{c_2}\}, \{\overset{\succ}{c_2}\}, \{\vec{e_2}\}, \{\vec{d_2}, \vec{e_2}\}, \{\vec{b_2}, \overset{\succ}{d_2}\}, \{\vec{b_2}\}, \{\{\vec{a_2}\}, \{\overset{\succ}{a_2}, \vec{c_1}\}, \{\overset{\succ}{c_1}\}, \{\vec{e_1}\}, \{\vec{d_1}, \overset{\succ}{e_1}\}, \{\vec{b_1}, \overset{\succ}{d_1}\}, \{\vec{b_1}\}\}$.

Even though a tetraploid has two copies of every gene, rather than one, the

[1] As mentioned in Section 1.1 tetraploids are genomes with four copies of each chromosome. However, over time, many tetraploids begin to operate on the same principle as a diploid with two virtually identical "copies" of the genome, except that each copy has two copies of each chromosome. Just like we do with diploids, we simplify the problem by ignoring the second "copy" of the genome. Thus, for our purposes a teraploid has two copies of each chromosome. If it is important to compare all four copies then diploids and tetraploids behave like tetraploids and octoploids respectively in the bioinformatics literature.

genome halving problem is still a special case of the rearrangement-duplication problem. The reason is that it is easy to reduce a tetraploid to an unambiguous genome by simply keeping only one copy of each chromosome, a trivial task given that chromosomes do not affect one another.

Transforming a genome into the tetraploid can be done using only rearrangement operations, thanks to the assumptions above, and transforming the tetraploid into an unambiguous genome requires a single duplication event: the tetraploidization. Thus, in the genome halving problem $\text{DISTANCE}_{DUP}(\mathbb{A}, \mathbb{B}) = 1$ and, thus, the problem is simply a matter of constructing a genome $\mathbb{B}$ from genome $\mathbb{A}$ that minimizes $\text{DISTANCE}_{*}(\mathbb{A}, \mathbb{B})$ and is a tetraploid. This turns out to be computationally easy and produced numerous linear time algorithms for HP distance[EMS03, AP07a], DCJ distance[WS09b, BMS06] and *single cut or join* distance[FaM09]. Despite being computationally easy, the genome halving problem is conceptually very hard, which is why there are four algorithms for it; each an attempt to further simply the original algorithm by [EMS03]. This problem is explored in detail in Chapter 2.

Taking advantage of the computational simplicity of the genome halving problem, [WS09a] attempted to generalize the result to polyploids introducing the *genome aliquoting problem.* Unfortunately, going from two copies of a gene to three or more is non-trivial and [WS09a] were only able to develop a heuristic under DCJ distance discussed in detail in Chapter 3. For *single cut or join* distance, a simplified distance operation inspired by DCJ, [FaM09] showed that the problem has a polynomial time solution. However, the existence of a polynomial time algorithm for both HP distance and DCJ distance remains an open problem, see Chapter 4.

Following up from [EMS03], [EM02] modified her genome halving algorithm and provided an algorithm for the more general rearrangement-duplication problem but only for genomes with gene families of size 1 and 2. The algorithm doesn't handle tandem duplications or polyploidization, instead it focuses on a type of duplication called a *duplication-transposition.* The *duplication-transposition* allows a sequence of exist-

ing genes to be inserted anywhere in the genome, simplifying the problem. The result is an algorithm even simpler than the algorithm for the genome halving problem. However, as observed in [Kah07], the [EM02] analysis of difficulty of duplication-transposition is flawed: the problem is much more difficult than [EM02] concluded. Thus, [Kah07] suggested, published in a peer-reviewed journal in [KR08], an $O(n^4)$ time algorithm to calculate the duplication distance (a slightly different distance than duplication-transposition distance). Unfortunately, it is not compatible with the [EM02] algorithm and [Kah07, KR08] don't suggest a means for minimizing the rearrangement distance. Thus, their algorithm only minimizes $\text{DISTANCE}_{DUP}(\mathbb{A}, \mathbb{B})$. Thus, a complete solution to this problem remains open, see Chapter 4 for details.

1.4 Results

The original paper on genome halving[EMS03], at almost 40 pages, was very difficult to understand. However, the underlying concept behind the genome halving algorithm was surprisingly simple but it was obfuscated in both the pseudocode and in the proofs of correctness by unintuitive concepts like *hurdles* and *fortresses* carried over from the Hannenhalli-Pevzner algorithm[HP95] that it was built on.

The work in this thesis began with an effort to simplify the El-Mabrouk algorithm[EMS03] by using the new, more intuitive double-cut and join algorithm. While the algorithm had to be expanded to accommodate multiple circular chromosomes, this involved no overhead and the end result, outlined in Chapter 2, was a much more condensed, if not simpler, algorithm and accompanying proofs.

During our work in simplifying the El-Mabrouk algorithm, we realized that it might be possible to generalize this result to handle any polyploid, not just tetraploids. In Chapter 3 of this thesis we present two approximation algorithms for the genome aliquoting problem under DCJ distance. One is an exact polynomial-time algorithm for the genome aliquoting problem under *breakpoint distance* and the other is an exact

polynomial-time algorithm for the genome aliquoting problem under *single cut or join distance*.

Chapter 2

Genome Halving

The genome halving problem is one of the few duplication problems with a polynomial time exact algorithm. Unfortunately, the genome halving problem has an undeserved reputation as being too complex to be of practical value. Admittedly, while undeserved, this reputation is not untrue with many genome halving algorithms, especially the most practical, being very complex due the large number of variables it needs to optimize simultaneously.

In this chapter, we reveal that the genome halving problem can be simple and that all the tasks it needs to perform to be practical needn't be done simultaneously; they can be split into several simple algorithms that can be run sequentially. Furthermore, we compare and contrast the four genome halving algorithms created to date and illustrate how they are all merely minor evolutions and simplifications of the given algorithm.

2.1 Simplified Halving

There are many factors to consider when halving a genome that tend to obfuscate and complicate the algorithms. As a result, genome halving algorithms have a reputation for being complex[AP07a] even though the underlying principle is very simple. In this section we present a simplified, but limited, definition of the genome halving problem that illustrates the core idea behind the algorithms. We will then frame all

the genome halving algorithms in terms of this definition to illustrate that they are all essentially the same. In later sections, we will build the algorithms presented in this section into full genome halving algorithms.

Despite covering more evolutionary operations, the double cut and join algorithm is significantly simpler than other genome rearrangement algorithms because it is more general. With double cut and join there are no restrictions on the number or types of chromosomes that a genome can contain and with double cut and join there is always a rearrangement operation that will change the genome in a meaningful and optimal way. It is for these reasons that, in this section, we consider only the problem of genome halving with double cut and join. Furthermore, to take full advantage of double cut and join, we allow any genomes with any number and any mix of both types (circular and linear) of chromosomes. Thus, we will halve genomes regardless of how biologically realistic the genome may be. However, the resulting halving, even given biologically realistic input, may not itself be biologically realistic. In Section 2.3, we will examine methods to ensure more biologically realistic output.

Even genome halving using the general distance metric of double cut and join is deceptively complicated. For this reason, in this section, we further restrict the problem by considering only the class of *simple genomes*. A genome $\mathbb{G}$ is a *simple genome* if and only if there exists a subset A of $\mathbb{G}$ where A is perfect and covers $\mathbb{G}$. We can now define the problem:

Definition 2.1.1. *Given a duplicated simple genome* $\mathbb{G}$*, the* simplified genome halving problem *is to find a tetraploid* $\mathbb{H}$ *such that* $\textsc{Distance}_{DCJ}(\mathbb{G}, \mathbb{H})$ *is minimal.*

The simplified genome halving problem is simpler than the genome halving problem because it is easier to find a perfectly duplicated genome; we need only find a perfect and covering subset of a genome and apply Algorithm 2.1 to produce a tetraploid. The following theorem establishes the correctness of Algorithm 2.1.

Algorithm 2.1: DUPLICATE

Input: A subset $A \subset \tilde{\mathbb{G}}$ that is both perfect and covers $\tilde{\mathbb{G}}$, where $\tilde{\mathbb{G}}$ is an unordered duplicated genome $\tilde{\mathbb{G}}$.
Output: An unordered tetraploid $\tilde{\mathbb{H}}$.

```
1 foreach α ∈ A do
2     if MULTIPLICITY_A(α) ≠ 2 then
3         MULTIPLICITY_A(α) ← 2
4     end
5 end
```

Theorem 2.1.2. *Given $A \subseteq \tilde{\mathbb{G}}$ where $\tilde{\mathbb{G}}$ is a duplicated genome and A is perfect and covers $\tilde{\mathbb{G}}$, $\tilde{\mathbb{H}} = \text{DUPLICATE}(A)$ is a tetraploid ancestor of $\tilde{\mathbb{G}}$.*

Proof. Any perfectly duplicated genome is a tetraploid, thus, we need to prove that it is a perfectly duplicated genome where $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$. To accomplish this we must prove that $\tilde{\mathbb{H}}$ is perfect, $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$ and is a genome.

First we observe that since A covers $\tilde{\mathbb{G}}$ and $\tilde{\mathbb{H}}$ doesn't removes element from A, $\tilde{\mathbb{H}}$ covers A which implies that $\tilde{\mathbb{H}}$ also covers $\tilde{\mathbb{G}}$. Second, since A is perfect and all elements added to $\tilde{\mathbb{H}}$ are added to increase the multiplicity of the elements in A and, therefore, already in A, it follows that $\tilde{\mathbb{H}}$ is also perfect. Thus, we need only prove that $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$ and is a genome.

Since $A \subseteq \tilde{\mathbb{G}}$, every element in A and every element in $\text{EXTREMITIES}(A)$ has a multiplicity ≤ 2. Consider any two elements $\alpha, \beta \in A$ where $\alpha \neq \beta$. Since A is perfect, α and β share no extremities in common. Hence, the multiplicity of the extremities in α (or β) is equal to the multiplicity of α. Thus, since $\tilde{\mathbb{H}}$ is perfect and covers A and every element in $\tilde{\mathbb{H}}$ has a multiplicity of exactly 2, every element in $\text{EXTREMITIES}(\tilde{\mathbb{H}})$ must also have a multiplicity of exactly 2. $\tilde{\mathbb{G}}$ has exactly two copies of every extremity. Hence, because $\tilde{\mathbb{H}}$ covers $\tilde{\mathbb{G}}$ and both have two copies of every extremity they have the same set of extremities.

From Remark 1.3.3, to be a genome the genome $\tilde{\mathbb{H}}$ must have both $\overrightarrow{g}$ and $\overleftarrow{g}$ for

every $g \in \text{GENES}(\tilde{\mathbb{H}})$. Since A and $\tilde{\mathbb{H}}$ are derived from the unordered genome $\tilde{\mathbb{G}}$, this means that they are genomes if and only if $\text{MULTIPLICITY}(\vec{g}) = \text{MULTIPLICITY}(\overleftarrow{g})$. As previously established, every element of $\tilde{\mathbb{H}}$ has a multiplicity of 2. Since $\tilde{\mathbb{H}}$ covers $\tilde{\mathbb{G}}$, and $\tilde{\mathbb{G}}$ is a genome, it follows that $\text{MULTIPLICITY}(\vec{g}) = \text{MULTIPLICITY}(\overleftarrow{g}) = 2$ for $g \in \text{GENES}(\tilde{\mathbb{H}})$. Therefore, $\tilde{\mathbb{H}}$ is a genome. □

While it is not obvious, in Section 2.1.6 we will see that any perfectly duplicated genome constructed in this manner is optimal. Thus, given a simple genome $\mathbb{G}$ the problem of genome halving with double cut and join is reduced to finding a subset that is perfect and covers $\mathbb{G}$.

By just looking at the set representation of $\mathbb{G}$, finding a subset A that is perfect and covers $\mathbb{G}$ seems to be a hard problem. In fact, it can be done in linear time and is very obvious, given an appropriate data structure. In Section 2.1.1 we will introduce a data structure that simplifies the problem and also implies a method for identifying simple genomes in linear time. In the Sections 2.1.2, 2.1.3, 2.1.4, 2.1.5, we will examine the four algorithms for genome halving, in the order that they were discovered, in terms of the simplified genome halving problem.

2.1.1 Intersection Graphs

To solve the simplified genome halving problem an algorithm must find a subset of a genome that is both perfect and covers the genome. Of these two properties, a perfect subset is more important as a good perfect subset implies a covering subset, as the following theorem proves:

Lemma 2.1.3. *Given an unordered duplicated genome $\tilde{\mathbb{G}}$, any perfect subset $A \subset \tilde{\mathbb{G}}$ that has a multiplicity of 1 and $|\text{EXTREMITIES}(A)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$ also covers $\tilde{\mathbb{G}}$.*

Proof. Assume that there exists a perfect subset $A \subset \tilde{\mathbb{G}}$ that has a multiplicity of 1 and $|\text{EXTREMITIES}(A)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$. Since a duplicated genome has at

most two copies of each extremity, if $|\text{EXTREMITIES}(A)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$ then A must cover $\tilde{\mathbb{G}}$ unless there exists $x \in \text{EXTREMITIES}(A)$ such that has a multiplicity of 2. Assume towards contradiction that such an x exists.

Let $\alpha_i, \alpha_j \in A$ be the elements of A that contain x. There are three possibilities: $\alpha_i = \alpha_j$ and $i \neq j$, $\alpha_i \neq \alpha_j$ or $\alpha_i = \alpha_j$ and $i = j$. For the first case, α_i has a multiplicity of 2, α_j being the second copy, hence, there exists an element of A with a multiplicity of 2. The second case is where $\alpha_i \neq \alpha_j$, hence, A is not perfect. For the third case, since x has a multiplicity of 2 and both copies are in α_i (alternatively, α_j), $\alpha_i = \{x, x\}$, hence, A is not perfect. Thus, in all three cases there is a contradiction, hence, A covers $\tilde{\mathbb{G}}$. □

Thus, the problem of finding a perfect and covering set is reduced to the problem of finding a perfect set with multiplicity of 1 containing half of the extremities. In fact, the two are almost, but not quite, equivalent, as the following theorem shows:

Theorem 2.1.4. *Given an unordered duplicated genome $\tilde{\mathbb{G}}$, there exists a subset $A \subset \tilde{\mathbb{G}}$ that is perfect and covers $\tilde{\mathbb{G}}$ if and only if there exists a perfect subset $B \subset \tilde{\mathbb{G}}$ that has a multiplicity of 1 and $|\text{EXTREMITIES}(B)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$.*

Proof. Assume that there exists a perfect subset $B \subset \tilde{\mathbb{G}}$ that has a multiplicity of 1 and $|\text{EXTREMITIES}(B)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$. From Lemma 2.1.3 it follows that B covers $\tilde{\mathbb{G}}$. Hence, there exists a subset that is perfect and covers $\tilde{\mathbb{G}}$.

Assume that there exists a subset $A \subset \tilde{\mathbb{G}}$ that is perfect and covers $\tilde{\mathbb{G}}$. To prove that B exists we will construct B from A: let B be A except that extra copies of elements are removed such that every element in B has a multiplicity of 1. By definition, every element in B has a multiplicity of 1. From the definition of perfect, it is easy to see that the removal of an element cannot make a perfect set rearranged (not perfect), thus, since A is perfect B is perfect. Since A covers $\tilde{\mathbb{G}}$ and no extremities of A have been removed (only copies of extremities) then B covers $\tilde{\mathbb{G}}$.

Since B covers $\tilde{\mathbb{G}}$, every element in B as a multiplicity of 1, and $\tilde{\mathbb{G}}$ is a duplicated genome, $|\text{EXTREMITIES}(B)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$. □

The intersection between two elements in the genome is the most important property in determining if the adjacencies are perfect. To illustrate this point, we can redefine the definition of perfect entirely in terms of the intersection as follows: given two elements α_i and α_j from a genome $\mathbb{G}$, α_i and α_j are perfect if and only if $\alpha_i \cap \alpha_j = \emptyset$ or $|\alpha_i \cap \alpha_j| = |\alpha_i| = |\alpha_j|$ but $\alpha_i \neq \{x, x\}$, for some extremity x. Thus, we define the following data structure:

Definition 2.1.5. *We define an* intersection pseudograph, INTERSECTIONGRAPH($\tilde{\mathbb{G}}$), *of an unordered genome* $\tilde{\mathbb{G}}$ *as follows:*

- *create a vertex* v_i *for every* $\alpha_i \in \tilde{\mathbb{G}}$. *Assign a label function to* v_i *such that* LABEL(v_i) = α_i;
- *connect* v_i *and* v_j *with a number of undirected edges* $e = \{v_i, v_j\}$ *equal to* $|\alpha_i \cap \alpha_j|$ *where* $i \neq j$;
- *connect* v_i *to itself with undirected edge for each* $x_j, x_l \in \alpha_i$ *where* $x_j = x_l$ *and* $j \neq l$;

For example, Figure 2.1 depicts the intersection pseudograph of $\tilde{\mathbb{G}} = \{\{\overrightarrow{a}\}, \{\overleftarrow{a}, \overleftarrow{b}\}, \{\overrightarrow{b}, \overleftarrow{d}\}, \{\overrightarrow{d}, \overleftarrow{e}\}, \{\overrightarrow{e}, \overleftarrow{b}\}, \{\overrightarrow{b}\}, \{\overrightarrow{a}\}, \{\overleftarrow{a}, \overleftarrow{c}\}, \{\overrightarrow{c}\}, \{\overrightarrow{c}\}, \{\overleftarrow{c}, \overrightarrow{e}\}, \{\overleftarrow{e}, \overrightarrow{d}\}, \{\overrightarrow{d}\}\}$.

Because this is an intersection pseudograph, DEGREE(v_i) = $|\alpha_i|$. Since each element has at most two extremities, the degree of any vertex is at most two. Thus, it is easy to subdivide the intersection pseudograph into connected components and, furthermore, each component is either a cycle or a path. Additionally, observe that since the vertex set of both the intersection graph and an adjacency graph are labeled with the elements of the set representation of the genome they are identical sets.

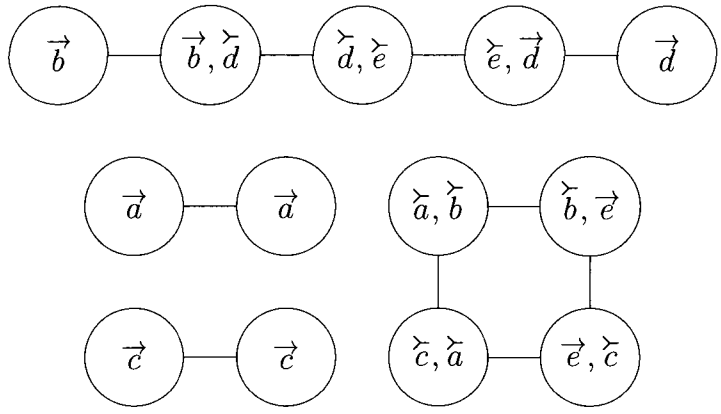


Figure 2.1: An example of an intersection pseudograph. The vertices represent the adjacencies and telomeres of an unordered genome while the edges represent the overlap between them.

The intersection graph shows the intersection which is useful to compute a perfect subset but how exactly can a perfect subset be computed? As we will now prove, the independent set, described in Section 1.2.3, is a perfect subset.

From the definition of an intersection and our intersection definition of perfect, we can immediately make the following helpful observations:

Observation 2.1.6. *Two elements α_i and α_j, where $i \neq j$, of a genome $\tilde{\mathbb{G}}$ are perfect if and only if their corresponding vertices v_i and v_j of* INTERSECTIONGRAPH($\tilde{\mathbb{G}}$) *are not adjacent or are the only vertices in their component.*

Observation 2.1.7. *Let α be an element of a genome $\tilde{\mathbb{G}}$ and let v the vertex labeled with α in* INTERSECTIONGRAPH($\tilde{\mathbb{G}}$). *α has a multiplicity of 2 in $\tilde{\mathbb{G}}$ if and only if v is a member of component of size 2.*

Combining Observations 2.1.6 and 2.1.7 yields an important theorem describing the relationship between an independent set of the intersection graph and perfect subsets of the genome:

Theorem 2.1.8. *I is an independent set of* INTERSECTIONGRAPH($\tilde{\mathbb{G}}$) *if and only if* LABEL(I) *is a perfect subset of $\tilde{\mathbb{G}}$ in which every element has a multiplicity of 1.*

Applying Theorem 2.1.8 to Lemma 2.1.3 gives us the following useful corollary:

Corollary 2.1.9. *Given an unordered duplicated genome* $\tilde{\mathbb{G}}$*, any independent set* I *of* INTERSECTIONGRAPH($\tilde{\mathbb{G}}$) *where* |EXTREMITIES(LABEL(I))| $\geq$ |EXTREMITIES($\tilde{\mathbb{G}}$)|/2, LABEL(I) *also covers* $\tilde{\mathbb{G}}$.

The other advantage of the intersection graph is that it can be used to determine if a genome is simple or not. In short, a genome is simple if and only if its intersection graph does not have an odd cycle. However, the full proof is similar to the algorithms discussed in the subsequent sections, thus, we will leave it until Section 2.1.6, after all the algorithms have been discussed. However, we can prove part of it now which will be needed to prove the remaining sections.

Lemma 2.1.10. *Given a genome* $\tilde{\mathbb{G}}$*, if* INTERSECTIONGRAPH($\tilde{\mathbb{G}}$) *has a component that is an odd cycle then the* INTERSECTIONGRAPH($\tilde{\mathbb{G}}$) *does not have an independent set* I *where* LABEL(I) *covers* $\tilde{\mathbb{G}}$.

Proof. Assume that INTERSECTIONGRAPH($\tilde{\mathbb{G}}$) has a component $\mathcal{C}$ that is an odd cycle. Since it is a cycle, each vertex is labeled with 2 extremities. However, because it odd, the independent set of $\mathcal{C}$ contains at most $(|\text{VERTICES}(\mathcal{C})| - 1)/2$ vertices. Thus, the number of extremities in the independent set is at most $|\text{VERTICES}(\mathcal{C})| - 1 <$ EXTREMITIES($\mathcal{C}$). From the definition of an intersection graph we know that the extremities of a component $\mathcal{C}$ are unique to $\mathcal{C}$, hence, there exists at least one extremity not covered by an independent set of $\tilde{\mathbb{G}}$. Therefore, there is no independent set that covers $\tilde{\mathbb{G}}$. □

Lemma 2.1.10 allows us to prove half of the relationship between odd cycles and simple genomes:

Theorem 2.1.11. *If the intersection graph of a unordered duplicated genome* $\tilde{\mathbb{G}}$ *has an odd cycle then* $\tilde{\mathbb{G}}$ *is not a simple genome.*

Proof. Assume that the intersection graph for $\tilde{\mathbb{G}}$ has an odd cycle. Let $A \subset \tilde{\mathbb{G}}$. Assume towards contradiction that A is perfect and covers $\tilde{\mathbb{G}}$. From Theorem 2.1.4 it follows that there exists a B that is perfect, has a multiplicity of 1 and $|\text{EXTREMITIES}(B)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$. Thus, from Theorem 2.1.8, it follows that B corresponds to an independent set I with $\text{LABEL}(I) = B$. From Lemma 2.1.10, B doesn't cover $\tilde{\mathbb{G}}$. However, because B doesn't cover $\tilde{\mathbb{G}}$ by modus tollens on Lemma 2.1.3, B isn't perfect, or doesn't have a multiplicity of 1, or $|B| < \text{EXTREMITIES}(\tilde{\mathbb{G}})/2$: a contradiction. Hence, A is not perfect or does not cover $\tilde{\mathbb{G}}$ and, therefore, any intersection graph with an odd cycle is not the intersection graph of a simple genome. □

2.1.2 El-Mabrouk-Sankoff Algorithm

The first genome halving algorithm developed is the El-Mabrouk-Sankoff algorithm outlined in [EMS03]. This algorithm performed genome halving on circular unichromosonal genomes and linear multi-chromosonal genomes using the reversal and translocation distance from [Tes02]. This method predates the set representation of a genome used in this thesis instead using a breakpoint graph to represent the genome. As a result, instead of an intersection graph it uses a *natural graph* which is a partitioning of the breakpoint graph into a set of lists of black edges.

Though it solves a more difficult problem, genome halving with reversals and translocations, the original El-Mabrouk-Sankoff algorithm will not solve the simplified genome halving problem without modifications. As mentioned before, there are many additional restrictions that have to be taken into account in more specific instances of the problem which cause the solution to be suboptimal in the more general case. However, by eliminating a few checks the El-Mabrouk-Sankoff algorithm can be used to solve the simplified genome halving problem.

Before the algorithm can be described, the essential data structure of El-Mabrouk-Sankoff approach, the natural graph, must be formally defined:

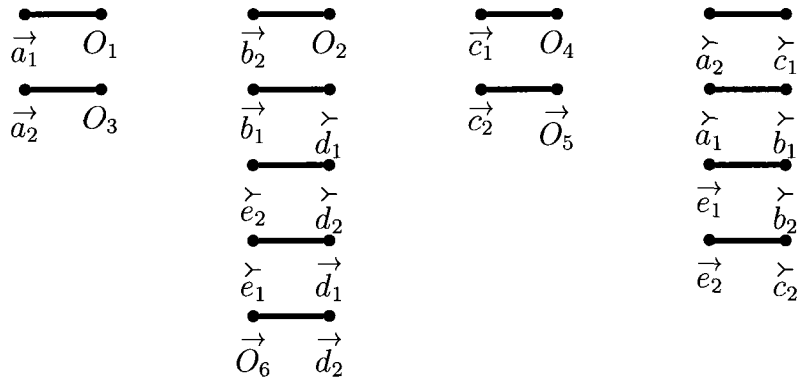


Figure 2.2: An example of a natural graph.

Definition 2.1.12. *Given a genome* $\mathbb{G}$, *let* $\mathcal{B} = \text{BREAKPOINTGRAPH}(\mathbb{G}, \emptyset)$. *The* natural graph *of* $\mathbb{G}$, *denoted* $\text{NATURALGRAPH}(\mathbb{G})$, *is a set of sequences* $NG = \{\text{N}_1, \text{N}_2, \ldots, \text{N}_m\}$ *of edges of* $\mathcal{B}$, *recursively defined, without a loss of generality, according to the following statement: for all extremities* x *and any cap or extremity* y, z, *if* $e = \{x_1, y\} \in \text{EDGES}(\mathcal{B})$ *is in* N_i *then so is any* $e' = \{x_2, z\} \in \text{EDGES}(\mathcal{B})$.

For example, Figure 2.2 depicts the natural graph of $\tilde{\mathbb{G}} = \{\{\vec{a}\}, \{\overleftarrow{a}, \overleftarrow{b}\}, \{\vec{b}, \overleftarrow{d}\}, \{\vec{d}, \overleftarrow{e}\}, \{\vec{e}, \overleftarrow{b}\}, \{\vec{b}\}, \{\vec{a}\}, \{\overleftarrow{a}, \overleftarrow{c}\}, \{\vec{c}\}, \{\vec{c}\}, \{\overleftarrow{c}, \vec{e}\}, \{\overleftarrow{e}, \overleftarrow{d}\}, \{\vec{d}\}\}$. Since Figure 2.2 uses the same genome as Figure 2.1, we can make some observations about their similarity: the sequences of the natural graph correspond exactly to the components of the intersection pseudograph. In fact, because the elements of the sequences of the natural graph corresponds to the edges of a breakpoint graph and vertices of the intersection pseudograph corresponds to vertices of an adjacency graph, the elements of the sequences correspond to the vertices of the intersection pseudograph (recall that the breakpoint graph is the line graph of the adjacency graph).

The order of the edges in the sequences of the natural graph is important, but, in Definition 2.1.12 we avoided defining it. Given the previous observation that the edges in the natural graph between natural graphs and intersection pseudographs we can now define an order:

Algorithm 2.2: ELMABROUKSANKOFF

Input: Duplicated simple genome $\mathbb{G}$.
Output: Set A that is perfect and covers UNORDER($\mathbb{G}$).

```
1 A ← ∅
2 NG ← NATURALGRAPH(𝔾)
3 foreach sequence N_i ∈ NG do
4     for j ← 1 to ⌊|N_i|/2⌋ do
5         add UNORDER(N_i[2 · j]) to A
6     end
7 end
8 return A
```

Definition 2.1.13. *Given a genome $\mathbb{G}$, elements e and f in the sequence $\mathrm{N}_i \in$* NATURALGRAPH($\mathbb{G}$) *are adjacent if and only if their corresponding vertices u and v in* INTERSECTIONGRAPH($\mathbb{G}$) *are adjacent.*

The natural graph depicted in Figure 2.2 is laid out in precisely this order.

The key to understanding the El-Mabrouk-Sankoff algorithm comes from [YAF05], the double cut and join distance algorithm for breakpoint graphs, in the form of a simple formula $\text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) = b - c$, where $\mathbb{G}$ and $\mathbb{H}$ are genomes and b is the number of black edges and c is the number of cycles in BREAKPOINTGRAPH($\mathbb{G}, \mathbb{H}$). In the case of genome halving, $\mathbb{G}$ and b fixed and, thus, the problem becomes that of constructing $\mathbb{H}$ such that c is maximized (therefore minimizing the distance). Since $\mathbb{H}$ is represented by gray edges in BREAKPOINTGRAPH($\mathbb{G}, \mathbb{H}$), the El-Mabrouk-Sankoff algorithm adds gray edges in such a manner as to maximize the number of cycles.

We take the same approach in Algorithm 2.2, however, in order to take advantage of the theorems and lemmas from Section 2.1.1 we will unorder the gray edges as we discover them. Since the breakpoint graph requires an ordered genome our algorithm requires that Algorithms 2.1 and 2.6 be applied before the breakpoint graph can be reconstructed. Thus, for now we will just prove that Algorithm 2.2 returns a perfect and covering set.

Theorem 2.1.14. *The set* $A = \text{ElMabroukSankoff}(\mathbb{G})$ *is perfect and covers* $\mathbb{G}$ *where* $\mathbb{G}$ *is a simple genome.*

Proof. Let $NG = \text{NaturalGraph}(\mathbb{G})$ containing sequence $\mathrm{N}_1, \mathrm{N}_2, \ldots, \mathrm{N}_m$. From Definition 2.1.13, two elements of a component N_i are adjacent if and only if their corresponding vertices are adjacent in the intersection graph. It follows from Observation 2.1.6 that since Algorithm 2.1 chooses every second element and therefore never chooses an adjacent element that A is perfect.

Similarly, by Observation 2.1.7, since Algorithm 2.1 never chooses adjacent elements it will never choose both elements of a sequence of $|\mathrm{N}_i| = 2$. Hence, all elements in A have multiplicity of 1.

Clearly, from Line 4, Algorithm 2.2 chooses at most $\left\lfloor \frac{|\sum_{1 \leq i \leq |NG|} \mathrm{N}_i|}{2} \right\rfloor$ elements. Since $\mathbb{G}$ is a simple genome, by modus tollens on Theorem 2.1.11 and Definition 2.1.13 that there are no sequences in NG that correspond to an odd cycle in $\mathcal{IG} = \text{IntersectionGraph}(\tilde{\mathbb{G}})$. Hence, if N_i corresponds to a cycle in $\mathcal{IG}$ then N_i has two extremities per element and, hence, A contains $|\mathrm{N}_i|$ our of $2 \cdot |\mathrm{N}_i|$ possible extremities: half of the extremities.

On the other hand, N_i can corresponds to a path in $\mathcal{IG}$ rather than a cycle. In this case, two elements of N_i, the first and last elements, $\mathrm{N}_i[1]$ and $\mathrm{N}_i[|\mathrm{N}_i|]$ respectively, contain only one extremity while all other elements contain 2. There are two cases: $|\mathrm{N}_i|$ is even or it is odd. If it is even, only $\text{Unorder}(\mathrm{N}_i[|\mathrm{N}_i|])$ is an element of A whereas if it is odd both $\text{Unorder}(\mathrm{N}_i[1])$ and $\text{Unorder}(\mathrm{N}_i[|\mathrm{N}_i|])$ are elements of A. Of the elements in N_i containing two extremities, in the case where N_i is even A contains $\frac{|\mathrm{N}_i|-2}{2}$ two extremity elements whereas, if it is odd, it contains $\frac{|\mathrm{N}_i|-3}{2}$ extremity elements. Thus, in either case, it contains $|\mathrm{N}_i| - 1$ extremities out of a possible $2 \cdot |\mathrm{N}_i| - 2$ extremities: half of the extremities.

Thus, on all three types of sequences, Algorithm 2.2 adds half of the extremities and, hence, A is perfect with a multiplicity of 1 and $\text{Extremities}(A) \geq$

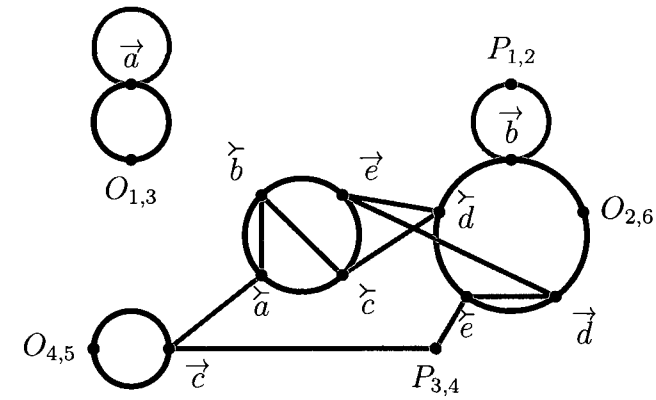
EXTREMITIES($\tilde{\mathbb{G}}$)/2. Therefore, by Lemma 2.1.3, A covers $\tilde{\mathbb{G}}$. □

Beyond being correct, the El-Mabrouk-Sankoff algorithm is fast. Computing the natural graph can be done in linear time. As mentioned in Section 1.2.2, we use sequences that are optimized for addition and deletion and, hence, random access takes linear time. However, the access in Algorithm 2.2 is not random access as it simply proceeds in order and, as a result, skipping every other element does not require random access. The UNORDER algorithm and the addition of 2-sets take constant time since there is only one element in $N_i[2 \cdot j]$. Thus, Algorithm 2.2 runs in linear time.

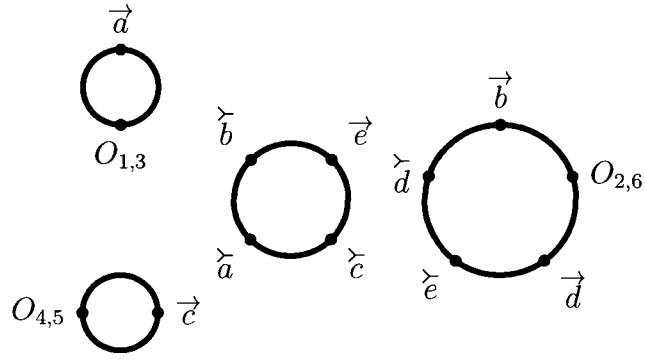
2.1.3 Alekseyev-Pevzner Algorithm

The Alekseyev-Pevzner algorithm, described in [AP07a], was the first attempt to refine and simplify the El-Mabrouk-Sankoff algorithm. They accomplished this by narrowing the ambitious scope of El-Mabrouk and Sankoff to circular unichromosomal genomes thereby focusing on reversal distance only. In this process of intensively studying the El-Mabrouk-Sankoff algorithm in order to improve it, Alekseyev and Pevzner found an error in the El-Mabrouk-Sankoff algorithm (not present in Algorithm 2.2) that occurs in the case of circular unichromosomal genomes and corrected in their algorithm. To keep consistent with the rest of this section and to demonstrate that all the genome halving algorithms are essentially the same, we present a generalized version of the Alekseyev-Pevzner algorithm to handle multichromosomal simple genomes and to use double cut and join rather than reversal distance.

The Alekseyev-Pevzner algorithm operates on data structure called the *contracted breakpoint graph*, which can be thought of as an intermediary between El-Mabrouk and Sankoff's natural graphs and the intersection graph previously discussed. In fact, the *contracted breakpoint graph* is exactly the breakpoint graph, as defined in Definition 1.3.1, but a breakpoint graph of unordered duplicated genomes.



(a)



(b)

Figure 2.3: (a) An example of a contracted breakpoint graph. (b) An example of a contracted breakpoint graph prior to the construction of the grey edges. Observe that it is the line graph of the intersection graph in Figure 2.1.

Our definition of a contracted breakpoint graph varies slightly from the Alekseyev-Pevzner definition in two ways. First, the Alekseyev-Pevzner contracted breakpoint graph depicts the *obverse edges* of the genome, edges between $\vec{x}$ and $\overleftarrow{x}$ of a gene x. As we will discuss below, in our overview of the Alekseyev-Pevzner algorithm, they needed these edges because their goal was to find a Hamiltonian cycle in this graph, thus, all the components needed to be connected. Our simplification of the problem doesn't require us to find a Hamiltonian cycle and, hence, we don't need the obverse edges. Second, Alekseyev-Pevzner do not cover the multichromosomal case and, thus, don't need to worry about capping. Since we do cover the multichromosomal case we

now have the problem of how to cap the contracted breakpoint graph.

Capping is used to compare chromosomes of different genomes. For genome halving capping simply involves matching the caps that are ends of the same sequence of a natural graph. Translating this to the Alekseyev-Pevzner algorithm, if there is a path consisting only of black or grey edges then the endpoints of that path, which are inevitably caps, are essentially duplicates and, hence, should be contracted together into one node, forming a cycle. The function CONTRACTEDBREAKPOINTGRAPH($\tilde{\mathbb{A}}$) creates the contracted breakpoint graph from an unordered duplicated genome $\tilde{\mathbb{A}}$.

For example, Figure 2.3a depicts the contracted breakpoint graph between the genomes $\tilde{\mathbb{G}} = \{(O_1, -a, +b, +d, +e, +b, O_2), (O_3, -a, +c, O_4),(O_5, -c, -e, +d, O_6)\}$ and $\tilde{\mathbb{H}} = \{(P_1, -b, +a, -a, -c, +b, P_2), (P_3, -c, +d, -e, -d, -e, P_4)\}$.

As discussed in Section 1.3.3, a breakpoint graph of an unordered duplicated genome, *i.e.* a contracted breakpoint graph, cannot be used to compute the rearrangement distance. However, Alekseyev and Pevzner discovered a means to transform a contracted breakpoint graph into a breakpoint graph; this is the essence of their genome halving algorithm.

The Alekseyev and Pevzner algorithm involves constructing an alternating black-obverse Hamiltonian cycle; the black edges used in this cycle correspond to adjacencies of a tetraploid, thus, they are also the grey edges of the graph. While the Hamiltonian cycle problem is, in general, NP-hard, Alekseyev and Pevzner show that the contracted breakpoint graph is a *de Bruijn graph* and, hence, detecting Hamiltonian cycles is polynomial time. However, they do not use the usual de Bruijn method to find the Hamiltonian cycle.

To find the alternating black-obverse Hamiltonian cycle, Alekseyev-Pevzner find a maximum matching of black edges for each component. Maximum matching is generally a polynomial time algorithm but, in this case, it is even easier as a maximum matching can be found using simple greedy algorithm. Since each vertex has one obverse edge, from the matching of black edges one or more Hamiltonian cycles are

Algorithm 2.3: AlekseyevPevzner

Input: Unordered duplicated simple genome $\tilde{\mathbb{G}}$.
Output: Set A that is perfect and covers $\tilde{\mathbb{G}}$.

```
1  A ← ∅
2  BG ← ContractedBreakpointGraph(G̃)
3  foreach component C in BG do
4      if there is a contracted cap u in Vertices(C) then
5          v_1 ← u
6      else
7          v_1 ← arbitrary vertex in Vertices(C)
8      end
9      from v_1 arbitrarily order Vertices(C) = v_2, ..., v_n s.t. v_i is adjacent to
       v_i + 1 and v_1 = v_n for all i where 1 ≤ i ≤ n and n = |C| + 1
10     for i ← 3 to n do
11         if i is odd then
12             add Label(v_i), Label(v_{i-1}) to A
13         end
14     end
15 end
16 return A
```

found. Alekseyev and Pevzner then merge the Hamiltonian cycles to form one single alternating Hamiltonian cycle for the entire graph.

The reason why the Alekseyev-Pevzner algorithm searches for a Hamiltonian cycle is to preserve the unichromosomal property of their genome. Since we simplified the problem by delegating this particular condition to Section 2.3.3, we will not concern ourselves with this here. Thus, our algorithm only needs to find a maximum matching, which is in fact equivalent to the independent set of the intersection graph since, aside from the capping, the contracted breakpoint graph is the line graph of the intersection graph.

Algorithm 2.3 computes the maximum matching and puts it into a format consistent with the theorems of Section 2.1.1. It is also consistent with the El-Mabrouk and Sankoff algorithm except for one small caveat: odd paths in the intersection

graph, which correspond to odd cycles in contracted breakpoint graph, require careful matching to avoid including the caps. In this case including the caps prevents a covering matching from being found but, in others cases, specifically the case of an even path in the intersection graph, caps are acceptable.

As we have mentioned previously, though have yet to prove, odd cycles in the intersection graph indicate a non-simple genome, a case that we avoid until Section 2.2. We avoid it because it is impossible to find an independent set (or matching, in this case) that is perfect and covering. However, since the contracted breakpoint graph contains only cycles, odd paths in the intersection graph become odd cycles with the same problem. Fortunately, missing caps don't violate the definition of covering, hence, rather than missing a gene the algorithm makes sure to miss the caps instead, thus, a matching that is both perfect and covering is found. This is identical to the El-Mabrouk and Sankoff algorithm as it also avoids caps in the case of odd size natural graphs.

Combined with Theorem 2.1.4, the following theorem proves the correctness of Algorithm 2.3:

Theorem 2.1.15. *Given a genome* $\tilde{\mathbb{G}}$, *let* $A = \textsc{AlekseyevPevzner}(\tilde{\mathbb{G}})$. A *is perfect and covers* $\tilde{\mathbb{G}}$.

Proof. The contracted breakpoint graph has only one copy of every extremity. From the loop on Line 10 in Algorithm 2.3 we can see that every extremity in a component $\mathcal{C}$ is added to A with two exceptions: the extremity $\textsc{Label}(v_1)$ and the extremity $\textsc{Label}(v_n)$, where $n = |\mathcal{C}|+1$. Interestingly, because all components in the contracted breakpoint graph are cycles and because of the way the ordering of vertices is defined in Algorithm 2.3, $v_1 = v_n$. Let $x = \textsc{Label}(v_1) = \textsc{Label}(v_n)$, $\textsc{Label}(v_1)$ is never added to A but when $|\mathcal{C}| + 1$ is odd $\textsc{Label}(v_n)$ is added to A, hence, x added to A at most once and x is added to A only when n is odd.

If a component $\mathcal{C}$ is an even cycle then $n = |\mathcal{C}| + 1$ is odd since $|\mathcal{C}|$ is even. Thus,

every extremity in $\mathcal{C}$ is in A, hence, they are covered.

An odd cycle $\mathcal{C}$ has exactly one contracted cap. Assume towards contradiction that $\mathcal{C}$ doesn't have a contracted cap as an element then it corresponds to an odd cycle in the intersection graph. However, because $\tilde{\mathbb{G}}$ is simple from Theorem 2.1.11 we know that the intersection graph doesn't have any odd cycles; a contradiction. Hence, $\mathcal{C}$ must have an element that is a contracted cap. Since $\mathcal{C}$ has a contracted cap it must correspond to a path in the intersection graph so it can have no more than one contracted cap since a path has exactly two endpoints.

If a component $\mathcal{C}$ is an odd cycle then the extremity $\textsc{Label}(v_1) = \textsc{Label}(v_n)$, where $n = |\mathcal{C}| + 1$, will not be included in A. However, from Line 5 in Algorithm 2.3 $v_1 = v_n$ is the contracted cap. Hence, all *extremities* of $\mathcal{C}$ are covered in A.

Therefore, A covers $\tilde{\mathbb{G}}$.

The contracted breakpoint graph has only one copy of every extremity, a fact which is mostly preserved in the ordering in Algorithm 2.3 except that $v_1 = v_n$, where $n = |\mathcal{C}|+1$. From the loop on Line 10 in Algorithm 2.3 we can see that it never chooses the same extremity twice so A is perfect so long as both $\textsc{Label}(v_1)$ and $\textsc{Label}(v_n)$ aren't added to A. $\textsc{Label}(v_1)$ is never added to A (the loop on Line 10 starts at 3). Therefore, A is perfect and has a multiplicity of 1.

We consider an adjacency in A of the form $\{x, O\}$, where x is an extremity and O is a cap, to be equal to $\{x\}$. Hence, every edge in the contracted breakpoint graph corresponds to an adjacency in $\tilde{\mathbb{G}}$. Since Algorithm 2.3 creates each adjacency in A using adjacent vertices v_i and v_{i-1}, each adjacency in A corresponds to an edge in contracted breakpoint graph and, hence, to an adjacency in $\tilde{\mathbb{G}}$. Therefore, A is a subset of $\tilde{\mathbb{G}}$. □

Algorithm 2.4: WARRENSANKOFF

Input: Unordered duplicated simple genome $\tilde{\mathbb{G}}$.
Output: Set A that is perfect and covers UNORDER($\tilde{\mathbb{G}}$).
1 $\mathcal{IG} \leftarrow$ INTERSECTIONGRAPH($\tilde{\mathbb{G}}$)
2 $I \leftarrow$ RESTRICTEDMAXIMUMINDEPENDENTSET($\mathcal{IG}$)
3 $A \leftarrow$ LABEL(I)
4 **return** A

2.1.4 Warren-Sankoff Algorithm

The Warren-Sankoff algorithm, outlined in [WS09b], was the first genome halving algorithm for the double cut and join distance. In fact, this is not true as the El-Mabrouk-Sankoff algorithm, which predates DCJ, is actually divided into two parts the first of which halves the genome using DCJ and the second of which "corrects" the first distance into the HP distance. Thus, the El-Mabrouk-Sankoff algorithm inadvertently solves the genome halving with DCJ problem as an intermediate step towards solving the genome halving with HP problem. More correctly, the Warren-Sankoff algorithm was the first adjacency graph based genome halving algorithm.

The primary purpose of the Warren-Sankoff algorithm was to simplify the very complex El-Mabrouk-Sankoff algorithm. To accomplish this three innovations where made: first, it uses the adjacency graph avoiding the need for caps; second, it is built using a well-studied problem, the independent set problem, on well-studied data structure, the intersection graph; and third, it accepts anything that is perfect and covering as a halving. If this seems familiar it is because, for this thesis, we have been examining all the genome halving algorithms in terms of the Warren-Sankoff algorithm and most of the proofs for the correctness of the algorithm have been already described in Section 2.1.1.

Algorithm 2.4 describes the Warren-Sankoff algorithm. Clearly, from Algorithm 2.4, $A = \text{LABEL}(I)$ where I is an independent set of INTERSECTIONGRAPH($\tilde{\mathbb{G}}$). From Corollary 2.1.9, to prove the correctness of Algorithm 2.4, we need only prove that A

contains half of the extremities.

Theorem 2.1.16. *Given a genome* $\tilde{\mathbb{G}}$, $|\text{EXTREMITIES}(A)| \geq |\text{EXTREMITIES}(\tilde{\mathbb{G}})|/2$, *where* $A = \text{WARRENSANKOFF}(\tilde{\mathbb{G}})$.

Proof. To determine the correctness of the Warren-Sankoff algorithm we must examine Algorithm 1.1: RESTRICTEDMAXIMUMINDEPENDENTSET in detail.

Clearly, from Line 5, Algorithm 1.1 chooses all the odd elements, so, half of the elements when there are an even number of vertices and one more than half when the number of vertices is odd. Since $\mathbb{G}$ is a simple genome, by modus tollens on Theorem 2.1.11 that there are no odd cycles in $\mathcal{IG} = \text{INTERSECTIONGRAPH}(\tilde{\mathbb{G}})$. Hence, if $\mathcal{C}$ is a cycle then $\mathcal{C}$ has two extremities per element and, hence, A contains $|\mathcal{C}|$ of the $2 \cdot |\mathcal{C}|$ possible extremities: half of the extremities.

On the other hand, $\mathcal{C}$ can be a path rather than a cycle. In this case, the start and end vertex, v_1 and $v_{|\mathcal{C}|}$ respectively, are labeled with one extremity although the remaining vertices are labeled with two extremities. There are two cases: $|\mathcal{C}|$ is even or it is odd. If it is even (an odd path), of v_1 and $v_{|\mathcal{C}|}$, only $\text{LABEL}(v_1)$ is an element of A whereas if it is odd (an even path) both $\text{LABEL}(v_1)$ and $\text{LABEL}(v_{|\mathcal{C}|})$ are in A. Of the remaining vertices, in the case where $|\mathcal{C}|$ is even A contains $\frac{|\mathcal{C}|-2}{2}$ vertices whereas, if it is odd, it contains $\frac{|\mathcal{C}|-3}{2}$ vertices. Thus, in either case, it contains $|\mathcal{C}| - 1$ extremities out of a possible $2 \cdot |\mathcal{C}| - 2$ extremities: half of the extremities.

Thus, from all three types of components Algorithm 2.4 takes half of the extremities. Therefore, the Warren-Sankoff algorithm is correct. □

For example, given a genome $\tilde{\mathbb{G}} = \{\{\overrightarrow{b}\}, \{\overleftarrow{b}, \overrightarrow{e}\}, \{\overleftarrow{e}, \overrightarrow{d}\}, \{\overleftarrow{d}, \overrightarrow{b}\}, \{\overleftarrow{b}, \overleftarrow{a}\}, \{\overrightarrow{a}\}, \{\overrightarrow{a}\}, \{\overleftarrow{a}, \overleftarrow{c}\}, \{\overrightarrow{c}\}, \{\overrightarrow{c}\}, \{\overleftarrow{c}, \overrightarrow{e}\}, \{\overleftarrow{e}, \overrightarrow{d}\}, \{\overrightarrow{d}\}\}$, the independent set of its intersection graph, depicted in Figure 2.4, forms the perfect and covering set $\text{LABEL}(I) = \{\{\overrightarrow{b}\}, \{\overleftarrow{e}, \overrightarrow{d}\}, \{\overrightarrow{d}\}\}, \{\overrightarrow{a}\}, \{\overleftarrow{b}, \overleftarrow{a}\}, \{\overleftarrow{c}, \overrightarrow{e}\}, \{\overrightarrow{c}\}\}$.

While at first this solution might seem to be much worse as the maximum independent set problem is, in general, an NP-hard problem, as discussed in Section 1.2.3,

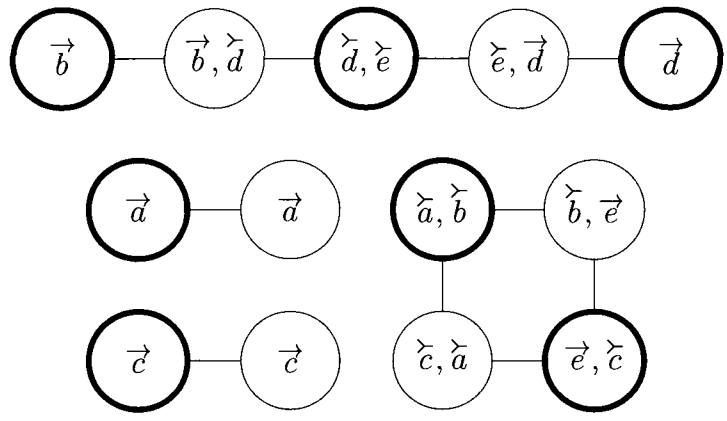


Figure 2.4: The result of the Warren-Sankoff algorithm. The vertices that form the independent set are bold.

in this case, the case where all vertices have a degree of at most 2, the independent set can be computed in linear time by Algorithm 1.1. In fact, computing the intersection graph and the transforming of the independent set into a genome via algorithm LABELS can also be in linear time meaning that, like the El-Mabrouk-Sankoff algorithm, this algorithm runs in linear time.

Unlike the other algorithms discusses in this chapter, because the Warren-Sankoff is built using well studied algorithms and data structures it can be easily generalized, although its generalization isn't necessarily a polynomial-time algorithm. Chapter 3 discusses a generalization of the Warren-Sankoff algorithm to the Genome Aliquoting problem in detail.

2.1.5 Mixtacki Algorithm

It is arguable that neither the Alekseyev-Pevzner algorithm nor Warren-Sankoff algorithm are simpler than El-Mabrouk-Sankoff algorithm; any gains towards simplicity were achieved thanks to a more limited scope, otherwise, as we have seen, all the algorithms are similar. However, the Mixtacki algorithm is simpler.

In essence, the Mixtacki algorithm is the Warren-Sankoff algorithm. No new data structures are provided, nor any substantial changes to the algorithm, nor any

Algorithm 2.5: MIXTACKI

Input: Unordered duplicated simple genome $\tilde{\mathbb{G}}$.
Output: Set A that is perfect and covers UNORDER($\tilde{\mathbb{G}}$).
1 $\mathcal{IG} \leftarrow$ INTERSECTIONGRAPH($\tilde{\mathbb{G}}$)
2 bipartition VERTICES($\mathcal{IG}$) into U and V s.t. U and V are independent sets
3 $A \leftarrow$ LABEL(U)
4 **return** A

new evolutionary operations are considered. Instead, Mixtacki offers a few key observations about the Warren-Sankoff approach that Warren and Sankoff overlooked but which greatly simplify the proofs and clarify the relationship between genome halving and DCJ.

What Mixtacki observed was that the intersection graph is a bipartite graph. Moreover, not only is it bipartite but it is bipartite in a very interesting and important way:

Observation 2.1.17. *The intersection graph of an unordered simple duplicated genome* $\tilde{\mathbb{G}}$ *is a bipartite graph such that each bipartition contains exactly one copy of every extremity,* i.e. *covers* $\tilde{\mathbb{G}}$.

This observation follows from Theorem 2.1.16, however, it also easily follows from the definition of intersection graph: since each edge connects both copies of an extremity, one copy of each extremity will appear on each side of the bipartition. It is also important to observe that this observation only applies to simple genomes, as previously mentioned but yet to be proven, only the intersection graphs of simple genomes are devoid of odd paths and, hence, only the intersection graphs of simple genomes are bipartite.

Algorithm 2.5 depicts the Mixtacki algorithm. Clearly, it is identical to the Warren-Sankoff algorithm except for how it computes the independent set. Its correctness immediately follows from Corollary 2.1.9 and Observation 2.1.17.

Beyond just providing an algorithm, Observation 2.1.17 implies what we have

avoided so far in this chapter: that this halving strategy produces a tetraploid with minimum double cut and join distance to the duplicated genome.

2.1.6 Optimality of Halving

In [EMS03, AP07a, WS09b] the proofs that their respective halving algorithms are optimal were convoluted at best. As a result, we have avoided them until this point. However, in addition to Observation 2.1.17, Mixtacki made another observation that almost trivializes the proofs that these genome halving algorithms are optimal:

Observation 2.1.18. *The intersection graph of a duplicated simple genome $\tilde{\mathbb{G}}$ is an adjacency graph of two non-duplicated genomes $\mathbb{U}$ and $\mathbb{V}$ where $\mathbb{U}$ and $\mathbb{V}$ are the bipartitions of the intersection graph.*

It follows that the DCJ operations needed to transform $\tilde{\mathbb{G}}$ into $\mathbb{U} \cup \mathbb{U}$ (or $\mathbb{U}||\mathbb{U}$ depending on wether or not we are considering the set representation or sequence representation of a genome respectively) is the same as the number of DCJ operations needed to transform $\mathbb{U}$ into $\mathbb{V}$. Thus, from Equation 1.3.6, the number of DCJ operations needed to transform $\tilde{\mathbb{G}}$ is

$$\textsc{Distance}_{DCJ}(\tilde{\mathbb{G}}, \mathbb{U} \cup \mathbb{U}) = \frac{|\textsc{Genes}(\tilde{\mathbb{G}})|}{2} - c - \left\lfloor \frac{\imath}{2} \right\rfloor \tag{2.1.19}$$

where c is the number of cycles and $\imath$ is the number of odd paths in the intersection graph of $\tilde{\mathbb{G}}$. Thus, the optimality of the Mixtacki algorithm follows from the optimality of DCJ. Similarly, it is easy to see that all the other algorithms follow the same pattern: they transform one part of the bipartition into the other, hence, they also construct maximally parsimonious genome.

Of course, while finding $\mathbb{U}$, a perfect and covering set, is the most difficult step in the algorithm, it is not the final step. After $\textsc{Duplicate}(\mathbb{U})$ is called, the new genome $\mathbb{U}\cup\mathbb{U}$ is an unordered genome; as discussed in Section 1.3.3 we cannot preserve

Algorithm 2.6: REORDER

Input: An unordered tetraploid $\mathbb{U} \cup \mathbb{U}$ and an ordered genome $\mathbb{G}$ such that $\mathbb{U} \subset \tilde{\mathbb{G}}$.

Output: An ordered tetraploid $\mathbb{H}$ such that $\text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) = \text{DISTANCE}_{DCJ}(\mathbb{U}, \mathbb{V})$ where $\mathbb{V} = \tilde{\mathbb{G}} \setminus \mathbb{U}$.

```
1  H ← ∅
2  foreach α ∈ G do
3      if UNORDER(α) ∈ U then
4          if α is an adjacency then
5              let {x_i, y_j} ← α where i, j ∈ {1, 2}
6              let ᾱ ← {x_{3−i}, y_{3−j}}
7          else
8              let {x_i} ← α where i ∈ {1, 2}
9              let ᾱ ← {x_{3−i}}
10         end
11         H ← H ∪ {α, ᾱ}
12     end
13 end
14 return H
```

the algorithms optimality by imposing an arbitrary order. Algorithm 2.6 REORDER correctly computes the optimal reordering as the following theorem proves:

Theorem 2.1.20. *Given an unordered tetraploid* $\mathbb{U} \cup \mathbb{U}$ *and an ordered genome* $\mathbb{G}$ *such that* $\mathbb{U} \subset \tilde{\mathbb{G}}$, *let ordered tetraploid* $\mathbb{H} = \text{REORDER}(\mathbb{U} \cup \mathbb{U}, \mathbb{G})$ *and let* $\mathbb{V} = \tilde{\mathbb{G}} \setminus \mathbb{U}$. $\text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) = \text{DISTANCE}_{DCJ}(\mathbb{U}, \mathbb{V})$.

Proof. We can view $\mathbb{G} = \mathbb{U}' \cup \mathbb{V}'$ where $\text{EXTREMITIES}(\mathbb{U}) \neq \text{EXTREMITIES}(\mathbb{V})$ as the extremities in the two partitions are distinguished by subscripts and where $\mathbb{U} = \text{UNORDER}(\mathbb{U}')$ and $\mathbb{V} = \text{UNORDER}(\mathbb{V}')$. Algorithm 2.6 distinguishes the two copies of $\mathbb{U}$ by creating $\mathbb{U}''$ and $\mathbb{U}^3$ such that $\text{EXTREMITIES}(\mathbb{U}'') = \text{EXTREMITIES}(\mathbb{U}')$ and $\text{EXTREMITIES}(\mathbb{U}^3) = \text{EXTREMITIES}(\mathbb{V}')$. Clearly, $\mathbb{U}''$ and $\mathbb{U}'$ are the same and, hence, the distance between them is 0. Similarly, DCJ distance between $\mathbb{U}^3$ and $\mathbb{V}'$ must be the same as the distance between their unordered counterparts $\mathbb{U}$ and $\mathbb{V}$. Therefore, $\text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) = \text{DISTANCE}_{DCJ}(\mathbb{U}, \mathbb{V})$. □

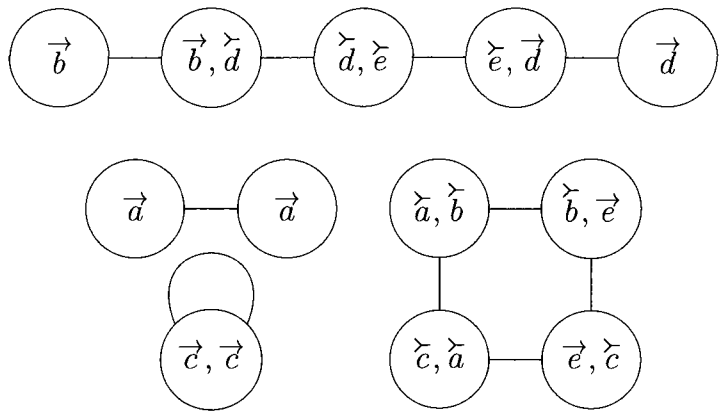


Figure 2.5: The intersection pseudograph of a non-simple genome. Observe that $\{\vec{c}, \vec{c}\}$ forms a odd cycle of size 1; odd cycles of any size denote a non-simple genome.

Having concluded our discussion of the algorithms, we will now complete the proof of the relationship between simple genomes and odd cycles which will be needed in the following section.

Theorem 2.1.21. *Given an unordered duplicated genome $\tilde{\mathbb{G}}$, if $\tilde{\mathbb{G}}$ is not a simple genome then* IntersectionGraph($\tilde{\mathbb{G}}$) *has a component C that is an odd cycle.*

Proof. Consider the contraposition that if IntersectionGraph($\tilde{\mathbb{G}}$) does not have a component C that is an odd cycle then $\tilde{\mathbb{G}}$ is a simple genome.

Assume that IntersectionGraph($\tilde{\mathbb{G}}$) does not have a component C that is an odd cycle. Then IntersectionGraph($\tilde{\mathbb{G}}$) is bipartite. Thus, from the previous algorithms, there must exist a set $A \subseteq \tilde{\mathbb{G}}$ that is perfect and covers $\tilde{\mathbb{G}}$. Hence $\tilde{\mathbb{G}}$ is a simple genome. □

Combining Theorems 2.1.11 and 2.1.21 it is now possible to easily distinguish if a genome is simple or not:

Corollary 2.1.22. *An unordered duplicated genome $\tilde{\mathbb{G}}$ is simple if and only if the intersection graph of $\tilde{\mathbb{G}}$ does not have a component that is an odd cycle.*

For example, genome $\mathbb{G} = \{\{\vec{a_1}\}, \{\overset{\succ}{c_1}, \overset{\succ}{a_1}\}, \{\vec{c_1}, \vec{c_2}\}, \{\vec{e_2}, \overset{\succ}{c_2}\}, \{\overset{\succ}{e_2}, \vec{d_1}\}, \{\vec{b_2}, \overset{\succ}{d_1}\},$

$\{\overrightarrow{b_2}, \overrightarrow{e_1}\}$, $\{\overrightarrow{d_2}, \overrightarrow{e_1}\}$, $\{\overrightarrow{d_2}\}$, $\{\overrightarrow{a_2}\}$, $\{\overrightarrow{a_2}, \overrightarrow{b_1}\}$, $\{\overrightarrow{b_1}\}\}$ is a non-simple genome as can be seen by the odd cycle in its intersection graph in Figure 2.5.

2.2 Generalized Genome Halving

From Corollary 2.1.22 it is clear that modifying the algorithms to handle any genome is simply a matter of developing an algorithm that handles odd cycle components in the intersection graph. Since, as written in their respective papers, all of the algorithms discusses so far are not limited to simple genomes there must be a solution for each of the algorithms. We will examine the solution presented in the Warren-Sankoff algorithm, in [WS09b], as it is the most general; all the other algorithms impose additional constraints while Warren-Sankoff ignore those constraints and instead aggressively optimize for parsimony. We will study the additional constraints imposed by other algorithms in Sections 2.3 and 2.4

Before discussing how to handle odd cycle components in the intersection graph, let us observe that if we eliminate the restriction that the input and output of all of the algorithms discussed in Section 2.1 be genome they still return the correct output. For example, given a subset G of a genome $\tilde{\mathbb{G}}$, Algorithm 2.4 returns a set A that is perfect and covers G but, since G is not necessarily a genome, neither A nor $H = \text{DUPLICATE}(A)$ are necessarily genomes. Thus, if we remove the elements that form the odd cycle components of the intersection graph from the genome, we can use the algorithms from Section 2.1. Thus, we can "halve" the odd cycle components in isolation and then recombine them with the results for the rest of the genome to get a tetraploid that covers the genome.

The main problem with odd cycles is that they cannot have a subset that is both perfect and covering. Clearly, then, the first restriction to lift is that we don't

[2]Any of the four genome halving algorithms discussed (Algorithms 2.2, 2.3, 2.4, 2.5) should be substituted for algorithm SIMPLEGENOMEHALVING in Algorithm 2.7.

Algorithm 2.7: GENOMEHALVING

Input: A duplicated genome $\mathbb{G}$.
Output: A tetraploid $\mathbb{H}$.

```
1 H̃ ← SIMPLEGENOMEHALVING(G)^2
2 H̃ ← DUPLICATE(H̃)
3 C ← ∅
4 foreach e ∈ EXTREMITIES(UNORDER(G)) \ EXTREMITIES(H̃) do
5     add {e1, e2} to C
6 end
7 H ← REORDER(H̃)
8 add C to H
9 return H
```

necessarily need to find a subset, the most parsimonious perfect and covering set is sufficient. The restriction that it must be a subset was to simplify the proofs because if there is a perfect and covering subset then for certain it has the most parsimonious distance. In fact, the El-Mabrouk algorithm as written in [EMS03] avoids the subset solutions.

While we argued that it was more important to find perfect subsets in Section 2.1.1, in this case it is more important to find a covering subset because if it is not covering then it is not a halving. That said, we cannot totally abandon our notion of perfect otherwise it is not a halving either. The one notion of perfect that can be abandoned is that we cannot have an adjacency of the form $\{x, x\}$, thus, we say an unordered duplicated genome $\tilde{\mathbb{A}}$ is *weakly perfect* if and only if for all $\alpha, \beta \in \tilde{\mathbb{A}}$, if $x \in \alpha$ and $x \in \beta$ then $\alpha = \beta$. We similarly defined *weakly perfect* for ordered genomes and any sets derived from them (either subsets or sets that use a subset of their extremities).

Algorithm 2.7 shows the complete genome halving algorithm that will halve any genome including non-simple genomes. It does this by adding each missing extremity x in an adjacency $\{x, x\}$ creating a weakly perfect halving. Before we can prove that this is optimal we must make a quick aside about DCJ.

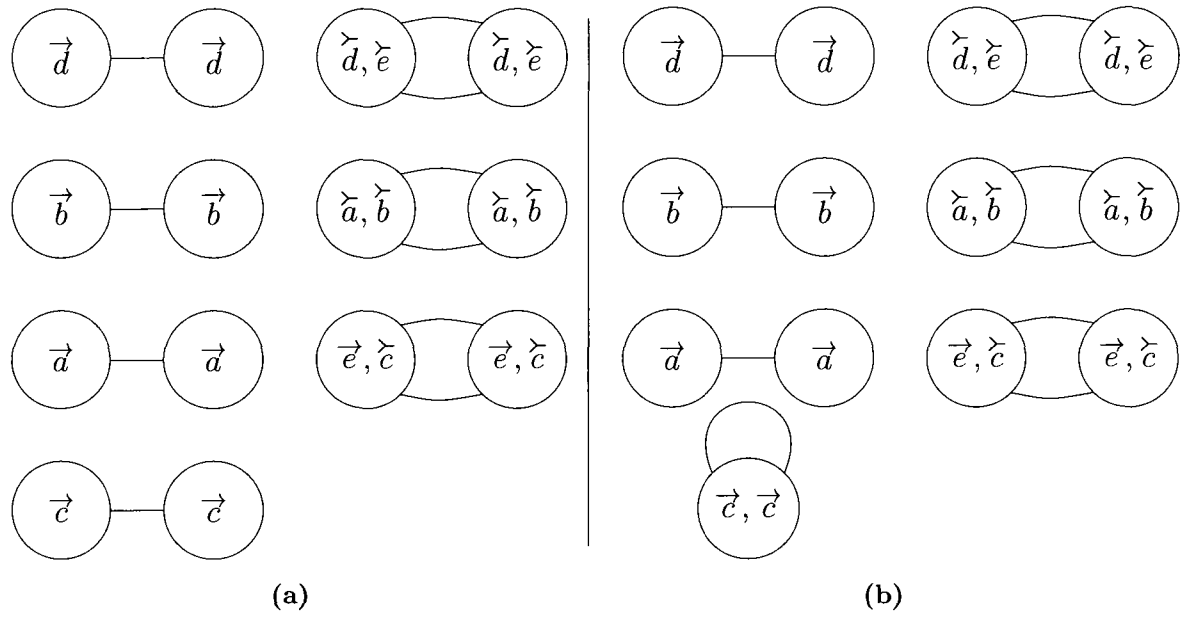


Figure 2.6: (a) The intersection pseudograph of a simple tetraploid. This tetraploid is a halving of the duplication genome depicted in Figure 2.1. (b) The intersection pseudograph of a non-simple tetraploid. This tetraploid is a halving of the duplication genome depicted in Figure 2.5.

When DCJ was defined in [BMS06] the following observation was made:

Observation 2.2.1. *Any graph where every vertex has a degree of at most* 2 *can be sorted by DCJ.*

The presence of odd cycles doesn't change that fact, even though adjacency graphs don't have odd cycles, we can still manipulate an intersection graph directly with DCJ (DCJ is often defined in terms of edges rather than extremities). Observation 2.2.1 is a generalization of Observation 2.1.18 and, like that observation, is also due to Mixtacki. Thus, we can count how many DCJ operations are needed to covert the intersection graph into the desired form to determine the distance.

As the following theorem proves, Algorithm 2.7 is an optimal solution:

Theorem 2.2.2. *Given a duplicated genome* $\mathbb{G}$, $\mathbb{H} = \textsc{GenomeHalving}(\mathbb{G})$ *is weakly perfect and covers* $\mathbb{G}$ *and* $\textsc{Distance}_{DCJ}(\mathbb{G}, \mathbb{H})$ *is minimal.*

Proof. The fact that $\mathbb{H}$ is both weakly perfect and covers $\mathbb{G}$ is trivial so we will focus on proving that it is minimal. The distance for the non-odd cycles is handled in the previous section so we will focus on the distance for odd cycles.

Observe that to create an even cycle of size 2 or an odd path of size 1 requires one DCJ operation from the odd cycle. Furthermore, since an odd cycle has an odd number of edges and each extremity corresponds to one edge, an odd cycle must have $2n+1$ extremities, for some natural number n. From $2n+1$ extremities, the maximum possible number of even cycles that can be created is n whereas the maximum possible number of odd paths is $2n+1$ but the last odd path doesn't require a DCJ operation meaning that $2n$ DCJ operations are required. Thus, we conclude that maximizing the number of even cycles requires the fewest number of DCJ operations.

Creating n even cycles leaves one extremity x in an adjacency $\{x, x\}$. By keeping $\{x, x\}$ the total number of DCJ operations is n. Since Algorithm 2.7 halves the genome by maximizing the number of even cycles and we explored all possibilities (except for the trivially rejected even paths option) we conclude that the distance is minimal. □

Since every odd cycle reorganizes $2n + 1$ extremities for $2n$ DCJ operations, for some natural number n, odd cycles decrease the DCJ distance. Therefore, given a genome $\mathbb{G}$ and $\mathbb{H} = \text{GENOMEHALVING}(\mathbb{G})$ the general genome halving distance formula is

$$\text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) = \frac{|\text{GENES}(\tilde{\mathbb{G}})|}{2} - c - \left\lceil \frac{i+o}{2} \right\rceil \tag{2.2.3}$$

where c is the number of cycles, i is the number of odd paths and o is the number of odd cycles in the adjacency graph of $\mathbb{G}$ and $\mathbb{H}$.

For example, the output for Algorithm 2.7 given the non-simple duplicated genome $\mathbb{G} = \{\{\vec{a_1}\}, \{\overset{\succ}{c_1}, \overset{\succ}{a_1}\}, \{\vec{c_1}, \vec{c_2}\}, \{\vec{e_2}, \overset{\succ}{c_2}\}, \{\overset{\succ}{e_2}, \vec{d_1}\}, \{\vec{b_2}, \overset{\succ}{d_1}\}, \{\overset{\succ}{b_2}, \vec{e_1}\}, \{\overset{\succ}{d_2}, \overset{\succ}{e_1}\}, \{\vec{d_2}\}, \{\vec{a_2}\}, \{\overset{\succ}{a_2}, \overset{\succ}{b_1}\}, \{\vec{b_1}\}$, with the intersection graph in Figure 2.5, as input is the tetraploid $\mathbb{H} = \{\{\vec{d_2}\}, \{\overset{\succ}{d_2}, \overset{\succ}{e_1}\}, \{\vec{e_1}, \overset{\succ}{c_1}\}, \{\vec{c_1}, \vec{c_2}\}, \{\vec{e_2}, \overset{\succ}{c_2}\}, \{\overset{\succ}{d_1}, \overset{\succ}{e_2}\}, \{\vec{d_1}\}, \{\vec{b_1}\}, \{\overset{\succ}{a_2}, \overset{\succ}{b_1}\},$

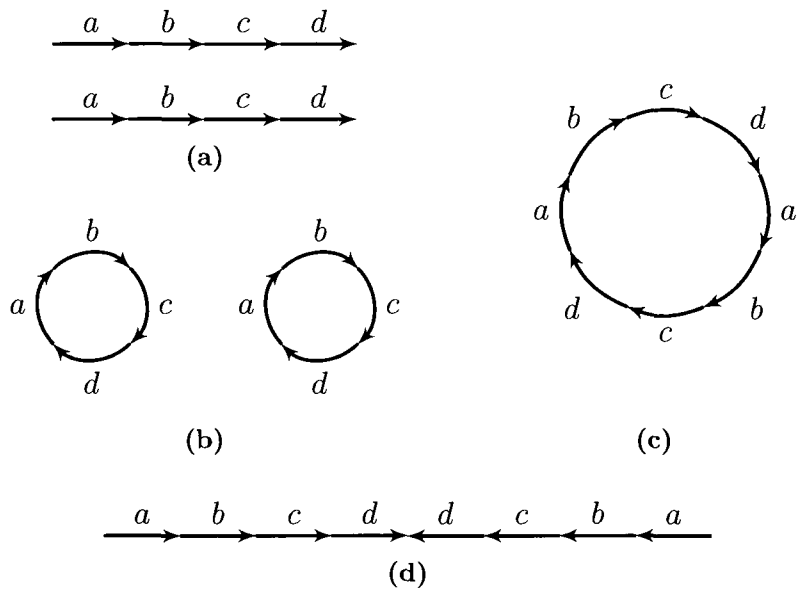


Figure 2.7: (a) A linear double chromosome halving. (b) A circular double chromosome halving. (c) A circular single chromosome halving. (d) A linear single chromosome halving. Examples of the various ways in which a chromosome can be considered halved.

$\{\overrightarrow{a_2}\}$, $\{\overrightarrow{b_2}\}$, $\{\overrightarrow{a_1}, \overrightarrow{b_2}\}$, $\{\overrightarrow{a_1}\}\}$. Observe that $\mathbb{H}$ still has an adjacency $\{\overrightarrow{c_1}, \overrightarrow{c_2}\}$ which corresponds to an odd cycle in its intersection graph depicted in Figure 2.6b.

While optimal from a computational perspective, as we will discuss in Section 2.3, the odd cycles can have undesirable points from a biological perspective and in many cases need to removed, which increases the distance.

2.3 Restricting Chromosomes

In all the previous sections, we have used a very liberal definition of halving: so long as the duplicate genes were separated into identical contiguous segments they are halved. Each contiguous segment must necessarily be part of a chromosome but, for each pair of identical contiguous segments, there are four possible chromosomal arrangements, which are depicted in Figure 2.7. Not all of these halvings are desirable and not all are desirable for every input, thus, in this section, we discuss methods for

eliminating undesirable halvings.

Most of the halvings depicted in Figure 2.7 are desirable at least some of the time. However, a *linear single halving*, depicted in Figure 2.7d, has no biological basis and, as such, is rarely, if ever, desirable. It is, however, an unfortunate consequence of the allowing weakly perfect halvings.

For the other possible halvings, in the case of mixed circular and linear input any of them can be considered desirable. However, if the input consists exclusively of linear chromosomes then, in general, it is desirable to get only linear chromosomes as output. Thus, in this case, only the linear double chromosome halving, depicted in Figure 2.7a, is acceptable. Similarly, in the common case of a circular unichromosomal genome as input, it is desirable to get a circular unichromosomal genome as output, thus, only circular single chromosome halving, depicted in Figure 2.7c, is desirable.

For the case of mixed circular and linear multichromosomal input, it is not clear what the desirable output should look like but to restrict it a more complex system is needed. Such a system has never been explored and, thus, is beyond the scope of this section.

2.3.1 Mixtacki Method

While what is a desirable halving is not clear in the case of mixed circular and linear input, what is clear is that chromosomes like the one depicted in Figure 2.7d are not desirable. In [Mix08], Mixtacki proposed a simple and elegant method for eliminating these chromosomes.

There is exactly one case where a tetraploid can have single linear chromosome halving: when it contains an adjacency of the form $\{x, x\}$ for some extremity x. There is no other possible case as any other adjacency that would merge two otherwise separate chromosomes would inevitably have a matching adjacency that, since the chromosomes are merged, would cause the duplicate segments to be circular.

Algorithm 2.8: ELIMINATELINEARSINGLECHROMOSOMEHALVINGS

Input: A tetraploid $\mathbb{G}$.

Output: A tetraploid $\mathbb{G}'$ without adjacencies of the form $\{x_1, x_2\}$ where x is an extremity.

```
1  G' ← ∅
2  β ← ∅
3  foreach adjacency α ∈ G do
4      if α is of the form {x1, x2} then
5          if β ≠ ∅ then
6              {x1, x2} ← α
7              {y1, y2} ← β
8              add {x1, y1} and {x2, y2} to G'
9              β ← ∅
10         else
11             β ← α
12         end
13     else
14         add α to G'
15     end
16 end
17 if β ≠ ∅ then
18     {y1, y2} ← β
19     add {y1} and {y2} to G'
20 end
21 return G'
```

Recall that the definition of perfect expressly forbids adjacencies of the form $\{x, x\}$ for some extremity x and in Section 2.2 weakly perfect was introduced to allow adjacencies to achieve the most parsimonious halving possible. Mixtacki's solution simply involves enforcing the definition of perfect to exclude adjacencies of the form $\{x, x\}$ and accept a less parsimonious halving, assuming, of course, that single chromosome "halvings" can even be considered halvings.

Forcing a halving to be perfect brings back the problem of how to handle odd cycles as, with any odd cycle regardless of size, there will inevitably be an extremity x that is uncovered without the inclusion of $\{x, x\}$. The naïve solution would be to

split $\{x, x\}$ into two telomeres but this is not the best solution as it costs extra one DCJ operation per odd cycle. Instead, it is better to take advantage of the double property of double cut and join and merge two odd cycles, if available, into one even cycle at a cost of one DCJ operation per two odd cycles. If there is an odd number of odd cycles, the remaining odd cycle will need to be converted into a pair of telomeres for one extra DCJ operation. Thus, this costs $\lceil \frac{o}{2} \rceil$ operations, where o is the number of odd cycles, exactly canceling out the benefit of leaving the odd cycles alone. Thus, the distance becomes the same as the distance for simple genomes in Equation 2.2.3.

For all its elegance there is, however, one minor drawback to the Mixtacki method. Not all adjacencies of the form $\{x, x\}$, for some extremity x, cause single linear chromosome halvings to occur, some cause single circular chromosome halvings. Thus, by eliminating adjacencies of the form $\{x, x\}$ we also eliminate some otherwise perfectly acceptable single circular chromosome halvings whose inclusion would slightly reduce the distance. This problem would be less severe if we simply forbade single circular chromosomes but, as previously mentioned, they still occur under the Mixtacki definition. Therefore, a better solution would be to either find a method to allow single circular chromosomes with adjacencies of the form $\{x, x\}$ while simultaneously forbidding single linear chromosomes or to eliminate single circular chromosomes. However, when mixed circular and linear chromosomes are given as input, no such method is known although such a method might work similarly to the strategies discussed below.

2.3.2 Linear Multichromosomal

If the input to the genome halving algorithm is a linear multichromosomal genome then the output should also be a linear multichromosomal genome consisting only of double linear chromosome halvings of the sort depicted in Figure 2.7a. The El-Mabrouk-Sankoff algorithm is the only algorithm that, as written, deals with this

particular case but does so within the genome halving algorithm complicating it. In [WS09b], it is mentioned that in some of the experiments they performed on their algorithm, Warren and Sankoff restricted their output to linear multichromosomal genomes but they do not mention how they accomplished this other than to say that they used the El-Mabrouk-Sankoff method. Not wanting to complicate the above genome halving algorithms we provide a post-processing algorithm that eliminates circular chromosomes if the input genome contains only linear chromosomes. This algorithm is based on a similar algorithm that Alekseyev and Pevzner provide to ensure that the genome produced by their algorithm is a circular unichromosomal genome.

To ensuring that all chromosomes are linear double chromosome halvings is to ensure that there are no linear single chromosome halvings. This can easily be accomplished, in a post-processing step, by using Algorithm 2.8, which is optimal in this situation since circular single chromosome halvings are not desirable either, although, we must still eliminate them.

Once all the linear single chromosome halvings are gone, next the algorithm eliminates the circular chromosomes. There are two strategies used in tandem for linearizing a genome: one is to cut circular chromosomes and the other is to merge circular chromosomes with linear chromosomes. One of these strategies is always available and one (or both) will always produce optimal results. Algorithm 2.9 optimally executes both these strategies, cutting circular chromosomes on Lines 17 & 19 and merging circular chromosomes with linear chromosomes on Lines 6 – 14. The following Theorem establishes the correctness of the algorithm:

Theorem 2.3.1. *Given a genome* $\mathbb{G}$ *and a tetraploid* $\mathbb{H} = \textsc{GenomeHalving}(\mathbb{G})$ *where* $\mathbb{G}$ *has only linear chromosomes but* $\mathbb{H}$ *has at least one circular chromosome, the tetraploid* $\mathbb{H}' = \textsc{LinearizeGenome}(\mathbb{G}, \mathbb{H})$*:*

1. *contains only linear double chromosome halvings;*

Algorithm 2.9: LINEARIZEGENOME

Input: Genome $\mathbb{G}$ and a tetraploid $\mathbb{H} = \text{GENOMEHALVING}(\mathbb{G})$ where $\mathbb{G}$ has only linear chromosomes but $\mathbb{H}$ has at least one circular chromosome.
Output: A tetraploid $\mathbb{H}'$ of $\mathbb{G}$ such that $\mathbb{H}'$ has only linear chromosomes.

1 $\mathbb{H}' \leftarrow$ ELIMINATELINEARSINGLECHROMOSOMEHALVINGS($\mathbb{H}$)
2 **foreach** circular chromosome $\mathbb{C}$ in $\mathbb{H}$ **do**
3 **if** there exists $\{x_i, y_j\} \in \mathbb{G}$ such that x_i is an extremity in $\mathbb{C}$ and y_j is an extremity in a linear chromosome $\mathbb{L}$ **then**
4 let $\alpha = \{x_i, u_k\}$ be the adjacency in $\mathbb{H}'$ that contains x_i
5 let β be the element in $\mathbb{H}'$ that contains y_i
6 **if** β is an adjacency **then**
7 let $\{y_i, v_l\} = \beta$
8 remove $\{x_i, u_k\}$, $\{x_{3-i}, u_{3-k}\}$, $\{y_i, v_l\}$ and $\{y_{3-i}, v_{3-l}\}$ from $\mathbb{H}'$
9 add $\{x_i, y_j\}$, $\{u_k, v_l\}$, $\{x_{3-i}, y_{3-j}\}$ and $\{u_{3-k}, v_{3-l}\}$ to $\mathbb{H}'$
10 **else**
11 let $\{y_i\} = \beta$
12 remove $\{x_i, u_k\}$, $\{x_{3-i}, u_{3-k}\}$, $\{y_i\}$ and $\{y_{3-i}\}$ from $\mathbb{H}'$
13 add $\{x_i, y_j\}$, $\{u_k\}$, $\{x_{3-i}, y_{3-j}\}$ and $\{u_{3-k}\}$ to $\mathbb{H}'$
14 **end**
15 **else**
16 let telomeres $\{x_i\}, \{y_j\} \in \mathbb{G}$ but not in $\mathbb{H}$
17 let $\{x_i, u_k\}$ and $\{y_j, v_l\}$ be the adjacencies in $\mathbb{H}$ that contain x_i and y_j
18 remove $\{x_i, u_k\}$, $\{x_{3-i}, u_{3-k}\}$, $\{y_i, v_l\}$ and $\{y_{3-i}, v_{3-l}\}$ from $\mathbb{H}'$
19 add $\{x_i\}$, $\{y_j\}$, $\{u_k, v_l\}$, $\{x_{3-i}\}$, $\{y_{3-j}\}$ and $\{u_{3-k}, v_{3-l}\}$ to $\mathbb{H}'$
20 **end**
21 **end**
22 **return** $\mathbb{H}'$

2. *has a distance to* $\mathbb{G}$ *equal to* $\frac{|\text{GENES}(\tilde{\mathbb{G}})|}{2} - c - \frac{i}{2}$ *where* c *is the number of cycles and* i *is the number of odd paths in the adjacency graph between them;*

Proof. Line 1 ensures that there are no linear single chromosome halvings. We claim that Algorithm 2.9 produces a genome that satisfies Condition 1 if:

1. if, at every iteration of the loop, the preconditions for one of the two conditional possibilities are satisfied. Thus, for the "if" case, described on Line 3, there exists an adjacency $\{x_i, y_j\}$ in $\mathbb{G}$ such that x_i is an extremity in a circular in $\mathbb{H}$

chromosome and y_j is an extremity in a linear chromosome in $\mathbb{H}$. The "else" case, described on Line 16, there must be two telomeres in $\mathbb{G}$ that are not telomeres in $\mathbb{H}$ and the telomeres are not duplicates of each other;

2. if the code in the "if" case, Lines 6 – 14, causes every pair of identical circular chromosomes or a single circular halved chromosome to merge with corresponding pair of identical linear chromosomes resulting in a pair of identical linear chromosomes;

3. if the code in the "else" case, Lines 17 & 19, cause a single halved circular chromosome every pair of identical circular chromosomes or a pair of identical circular chromosomes and a single circular halved chromosome or two pairs of identical circular chromosomes to linearize resulting in a pair of identical linear chromosomes;

We will prove each claim above in the same order that they are listed in:

1. Either there exists a telomeres $\{w_i\}$ that is in both $\mathbb{G}$ and $\mathbb{H}'$ or there doesn't. Assume that such a telomere does exist. Let $\mathbb{L} = (O_a, w_i, \ldots, x_j, y_l \ldots, z_k, O_b)$ be the chromosome beginning with the telomere $\{w_i\}$ in $\mathbb{H}$ with y_l being the first extremity that differs from the chromosome beginning with $\{w_i\}$ in $\mathbb{G}$. Either such an $\mathbb{L}$ exists or it doesn't; assume that is exists. Then there is an adjacency $\{x_j, y_l\}$ where the gene x_j is on a linear chromosome in $\mathbb{H}$ and the gene y_l is on a circular chromosome in $\mathbb{H}$.

   If there does not exist $\mathbb{L}$ with an extremity y_j that differs then all linear chromosomes in $\mathbb{H}$ are in $\mathbb{G}$ but since $\mathbb{H}$ has circular chromosomes and $\mathbb{G}$ has only linear chromosomes then there must exists a telomere $\{x_i\}$ that is in $\mathbb{G}$ but not in $\mathbb{H}$. By similar reasoning, if $\mathbb{H}$ has no linear chromosomes then there must exists a telomere $\{x_i\}$ that is in $\mathbb{G}$ but not in $\mathbb{H}$. Because $\{x_i\}$ corresponds to a linear chromosome that does not have any telomeres in $\mathbb{H}$ it must have

a second telomere not in $\mathbb{H}$. However, assume towards contradiction that the second telomere of the chromosome is $\{x_{3-i}\}$. Then the natural graph of $\mathbb{G}$ must consist of an odd path of size 1 with two vertices labeled $\{x_i\}$ and $\{x_{3-i}\}$. However, all of the genome halving algorithms add at least one vertex from an odd path to the tetraploid, which, in the case of an odd path of size 1, means both telomeres to be in $\mathbb{H}$ and, therefore, $\mathbb{H}'$. Therefore, there must exists two telomeres $\{x_i\}$ and $\{y_j\}$ that are in $\mathbb{G}$ but not in $\mathbb{H}'$.

2. Only one adjacency (or telomere) is altered in each of the four effected chromosomes, thus, all other genes remain linked in the same manner. Clearly, the change in the four adjacencies cause one linear chromosomes to merge with one of the circular chromosomes.

   If no telomeres are altered then there must still be four telomeres among two chromosomes and, hence, both chromosomes must be linear. Since both chromosomes were altered identically they also remain identical. Furthermore, if the genome was halved, merging these adjacencies doesn't disrupt the halving as one duplicate is never joined with the other duplicate.

   If telomeres in each of the linear chromosomes were altered then observe that a pair of new telomeres $\{u_k\}$ and $\{u_{3-k}\}$ are created in place. Thus, both chromosomes must still be linear.

   From Line 1 there must be two linear chromosomes and, hence, four telomeres. Thus, even though both linear chromosomes are merged with a single circular chromosome there must still be four telomeres and, hence, by similar reason as above there must be two identical linear chromosomes.

3. The initial number of circular chromosomes depends on wether of not $\{x_i\}$, $\{y_j\}$, $\{x_{3-i}\}$ and $\{y_{3-j}\}$ are in the same of different chromosomes. Since they will all be telomeres of the linear chromosomes we can be sure that $\{x_i\}$ and $\{x_{3-i}\}$

will be on different linear chromosomes and, similarly, for $\{y_j\}$ and $\{y_{3-j}\}$. The adjacency $\{u_k, v_l\}$ causes the chromosome containing $\{x_i\}$ and $\{y_j\}$ to merge, if they are on different chromosomes. Thus, by similar reason as case 2 all circular chromosomes must be merged into two identical linear chromosomes.

Since Algorithm 2.8 reduces the distance to $\frac{|\text{GENES}(\tilde{\mathbb{G}})|}{2} - c - \frac{i}{2}$, we claim Condition 2 is satisfied if Lines 6 – 14 and Lines 17 & 19 do not alter the number of odd paths and cycles in the adjacency graph.

Consider Lines 6 – 14 where x_i and y_j are both part of adjacencies in $\mathbb{H}'$. Since $\{x_i, y_j\}$ is not in $\mathbb{H}'$ there are at most three effected cycles in the adjacency graph: one containing $\{x_i, u_k\}$ and $\{y_j, v_l\}$, one containing $\{x_{3-i}, u_{3-k}\}$ and one containing $\{y_{3-j}, v_{3-l}\}$. By adding $\{x_i, y_j\}$ to $\mathbb{H}'$ we create one cycle of size 2. Since both $\{x_i, y_j\}$ are in $\mathbb{G}$ and $\mathbb{H}'$ we split apart the cycle (or odd path, or even path) containing $\{x_i, y_j\}$ in the natural graph, however, by adding $\{u_k, v_l\}$ to $\mathbb{H}'$ it must rejoin the cycle (or odd path, or even path). Adding $\{x_{3-i}, y_{3-j}\}$ and $\{u_{3-k}, v_{3-l}\}$ it merges the cycle (or odd path, or even path) one containing $\{x_{3-i}, u_{3-k}\}$ and one containing $\{y_{3-j}, v_{3-l}\}$ into one cycle (or odd path, or even path). Thus, the number of cycles, odd paths, and even paths doesn't change.

When x_i or y_j are part of telomeres in either $\mathbb{G}$ (Lines 17 & 19) or $\mathbb{H}'$ (Lines 6 – 14) then there are fewer cycles and more odd paths but the reasoning is similar. Hence, the distance does not change. □

Determining if a chromosome is circular or linear takes linear time. However, this needs only be done once provided we determine in advance which chromosomes are identical and which are linear, thus, this can be done before the loop. Algorithm 2.8 takes linear time, and the loop in Algorithm 2.9 takes linear time. Hence, the whole algorithm runs in linear time.

2.3.3 Circular Unichromosomal

The circular unichromosomal case is, in one sense, easier than the linear multichromosomal case because there are only circular chromosomes and the only strategy needed to create a unichromosomal genome is to merge the circular chromosomes until only one remains. However, in another sense, it poses a problem because adjacencies of the form $\{x, x\}$, for some extremity x, are permitted and even desirable in some, but not all, cases.

A single chromosome with more than two adjacencies of the form $\{x, x\}$ cannot possibly be halved. Unfortunately, this restriction doesn't apply to the input circular unichromosomal case, which can in fact have several adjacencies of that form or, alternatively, break down into numerous odd cycles on the intersection graph. Thus, keeping two adjacencies of the form $\{x, x\}$ is desirable, as it reduces the distance, but more than that is not desirable, as it cannot form a single unichromosomal genome. Thus, use Algorithm 2.8 to remove all but two of these adjacencies; We call this modified version of the algorithm ModifiedEliminateLinearSingleChromosomeHalvings. Because the input genome is circular the intersection graph only contains cycles, so there must be an even number of odd cycles so there is no risk of Algorithm 2.8 introducing telomeres to the genome.

Algorithm 2.10 describes how to transform the genome into a circular unichromosomal genome. The following theorem proves its correctness:

Theorem 2.3.2. *Given a genome* $\mathbb{G}$ *and a tetraploid* $\mathbb{H} = \text{GenomeHalving}(\mathbb{G})$ *where* $\mathbb{G}$ *is a circular unichromosomal genome but* $\mathbb{H}$ *has multiple circular chromosome, the tetraploid* $\mathbb{H}' = \text{MergeCircles}(\mathbb{G}, \mathbb{H})$*:*

1. *contains a circular unichromosomal genome;*

2. *has a distance to* $\mathbb{G}$ *equal to* $\frac{|\text{Genes}(\tilde{\mathbb{G}})|}{2} - c - \frac{\imath}{2} - 2$ *where* c *is the number of cycles and* $\imath$ *is the number of odd paths in the adjacency graph between them if the*

Algorithm 2.10: MERGECIRCLES

Input: Genome $\mathbb{G}$ and a tetraploid $\mathbb{H} = \text{GENOMEHALVING}(\mathbb{G})$ where $\mathbb{G}$ has only linear chromosomes but $\mathbb{H}$ has more than one circular chromosome.

Output: A tetraploid $\mathbb{H}'$ of $\mathbb{G}$ such that $\mathbb{H}'$ has only one circular chromosome.

```
1 ℍ' ← MODIFIEDELIMINATELINEARSINGLECHROMOSOMEHALVINGS(ℍ)
2 while there is more than circular chromosome in ℍ do
3     there exists {x_i, y_j} ∈ 𝔾 such that x_i is an extremity in a circular
      chromosome ℂ and y_j is an extremity in another circular chromosome ℂ'
4     let {x_i, u_k} be the adjacency in ℍ' that contains x_i
5     let {y_i, v_l} be the adjacency in ℍ' that contains y_i
6     remove {x_i, u_k}, {x_{3-i}, u_{3-k}}, {y_i, v_l} and {y_{3-i}, v_{3-l}} from ℍ'
7     add {x_i, y_j}, {u_k, v_l}, {x_{3-i}, y_{3-j}} and {u_{3-k}, v_{3-l}} to ℍ'
8 end
9 return ℍ'
```

intersection graph of $\mathbb{G}$ *contains two or more odd cycles or* $\frac{|\text{GENES}(\tilde{\mathbb{G}})|}{2} - c - \frac{i}{2}$ *if the intersection graph of* $\mathbb{G}$ *contains no odd cycles;*

Proof. Line 1 ensures that there are no linear single chromosome halvings. We claim that Condition 1 is satisfied if:

1. the precondition for every iteration of the loop is true. That is if every for every circular chromosome there exists an adjacency $\{x_i, y_j\}$ where x_i is an extremity in one circular chromosome in $\mathbb{H}$ and y_j is an extremity of another circular chromsome in $\mathbb{H}$;

2. at each invocation of Line 7 the number of circular chromosomes is reduced by at least one;

We will prove each claim above in the same order that they are listed in:

1. Since there is only one chromosome in $\mathbb{G}$ but there is more than one in $\mathbb{H}'$, there must be an adjacency $\{x_i, y_j\}$ that is in $\mathbb{G}$ but not in $\mathbb{H}'$ that merges the two genomes.

2. Only four adjacency mentioned are altered in each of the effected chromosomes, thus, all other genes remain linked in the same manner. Clearly, the change in the four adjacencies causes at least two circular chromosomes to merge into one. Furthermore, if the genome was halved merging these adjacencies doesn't disrupt the halving as one duplicate is never joined with the other duplicate.

Since the modified Algorithm 2.8 reduces the distance to $\frac{|\text{GENES}(\mathbb{G})|}{2} - c - \frac{i}{2} - 2$ or $\frac{|\text{GENES}(\mathbb{G})|}{2} - c - \frac{i}{2}$ depending on the number of odd cycles in the intersection graph, we claim Condition 2 is satisfied if Line 7 does not alter the number of cycles in the adjacency graph.

Since $\mathbb{G}$ and $\mathbb{H}'$ contain only adjacencies, their adjacency graph contains only cycles. Consider Line 7 where x_i and y_j are both part of adjacencies in $\mathbb{H}'$. Since $\{x_i, y_j\}$ is not in $\mathbb{H}'$ there are at most three effected cycles in the adjacency graph: one containing $\{x_i, u_k\}$ and $\{y_j, v_l\}$, one containing $\{x_{3-i}, u_{3-k}\}$ and one containing $\{y_{3-j}, v_{3-l}\}$. By adding $\{x_i, y_j\}$ to $\mathbb{H}'$ we create one cycle of size 2. Since both $\{x_i, y_j\}$ are in $\mathbb{G}$ and $\mathbb{H}'$ we split apart the cycle containing $\{x_i, y_j\}$ in the natural graph, however, by adding $\{u_k, v_l\}$ to $\mathbb{H}'$ it must rejoin the cycle. Adding $\{x_{3-i}, y_{3-j}\}$ and $\{u_{3-k}, v_{3-l}\}$ it merges the cycle one containing $\{x_{3-i}, u_{3-k}\}$ and one containing $\{y_{3-j}, v_{3-l}\}$ into one cycle. Thus, the number of cycles doesn't change. □

Computing the circular chromosomes takes linear time. However, this needs only be done once provided we determine in advance which chromosomes are identical, after that we need only keep track of the which chromosomes are merged, which can be done in constant time. Thus, we can compute the number of circular chromosomes prior to the loop. The modified version of Algorithm 2.8 takes linear time, and the loop in Algorithm 2.10 takes linear time. Hence, the whole algorithm runs in linear time.

2.4 Eliminating Block Interchanges

The only difference between HP sorting and DCJ sorting is the presence of block interchanges in DCJ. This difference is profound: where two parameters are needed to compute the DCJ distance (in the case of the original formula due to [YAF05]), the number of genes and the number of cycles, the HP distance also requires that *hurdles*, *greatest hurdles*, *super-hurdles*, *fortresses*, *knots*, *greatest knots*, *super-knots*, *fortress-of-knots*, *semi-knots*, *real-knots*, *greatest real-knots*, *super-real-knots*, *fortress-of-real-knots* and many more graph parameters be computed (we refer adventurous readers to [HP95] for the definitions); none of which are trivially computed. For simplicity, we will use the term hurdle to refer to any or all of these parameters.

While difficult to compute and even comprehend, the intuition behind all the graph parameters needed to compute HP distance is easy to understand. When sorting by reversals and translocation it is possible to arrive at situation where no rearrangement will bring one genome closer to the other. In this situation it takes two or more reversals and/or translocations before progress can resume. This is understandably a difficult situation as not just any reversal or translocation will improve the situation as there is often only one correct rearrangement and there is no obvious way to figure out what that correct rearrangement might be. There is, however, an unobvious way to find the correct rearrangement in this situation: it is through the use of hurdles.

Block interchanges simplify the problem because, with the addition of a block interchange, there is always an obvious correct move. In fact, block interchanges occur exactly in the place where are hurdles would be needed. Thus, to convert from DCJ distance to HP distance one must simply eliminate block interchanges. In the context of genome halving this means minimizing hurdles.

Unfortunately, eliminating the block interchanges is easier said then done. Despite recent advances, particular those outlined in [BMS08], it is still daunting. The

only true algorithm to handle HP distance in genome halving is due to [EMS03]; all other algorithms merely refer interested readers back to that paper. However, in this section, for the sake of completeness, we sketch the procedure for the interested reader.

2.4.1 Detecting Unoriented Components

Hurdles, knots, semi-knots, *etc* are all various cases of a layout of genes commonly known as an *unoriented component*. In this section we define *unoriented components* and there various configurations using the terminology from [BMS08]. We chose [BMS08] for two reasons: first, it uses the simplest set of terminology (fewest terms) and, second, it begins with DCJ distance and coverts it to HP distance, which is our problem.

Before we can define *unoriented component* we must first define what *component* means in the context of a genome:

Definition 2.4.1 ([BMS08]). *Given two genomes $\mathbb{A}$ and $\mathbb{B}$, an interval $(l, \ldots, r)$ of a genome $\mathbb{A}$ is a* component *relative to genome $\mathbb{B}$ if there exists an interval in genome $\mathbb{B}$:*

- *with the same* endpoints, i.e. *the first and last extremity in the interval must be the same in both genomes or must be a cap (not necessarily the same cap) in both intervals;*
- *with the same set of genes and both extremities of every gene must be within the interval;*
- *that is not the union of two such intervals;*

Two components may be nested or may share one gene in common but the definition prevents any other combination of overlap between them. However, it is

often important to study components on adjacency graphs, thus, we must distribute the adjacencies between the components such that each adjacency *properly belongs* to only one component. A component *contains* an adjacency if and only if both of the adjacencies extremities are in the component. An adjacency *properly belongs* to the smallest component that contains it, *i.e.* adjacencies properly belong to the most nested component and, the case of a shared gene, each component gets an adjacency containing one extremity of the gene that they share. As a consequence of this definition, the cycles and paths of the adjacency are not split across multiple components[BMS08].

Components can further be subdivided into *blocks* consisting of nested components and individual genes such that the concatenation of the blocks form the component.Given two genomes $\mathbb{A}$ and $\mathbb{B}$ and set of n components $C_1, \ldots, C_n$ between them, for any component C_i let $(1, \ldots, k)$ be a labeling of the blocks of C_i, where k is the number of blocks, by the order they appear in genome $\mathbb{B}$. The *associated permutation of* C_i is a permutation $(\rho_1, \ldots, \rho_k)$ of the labels $1, \ldots, k$ to match the order the blocks appear in genome $\mathbb{A}$ with negative signs in the permutation indicating that a genes has changed orientation.

From [BMS08], a non-trivial component C is an *oriented component* if and only if its associated permutation has both positive and negative elements or its adjacency graph has two even paths. Otherwise, C is an *unoriented component.*

Correcting unoriented components is problematic as correcting one unoriented component can interfere with another. Finding and identifying all these situations has lead to the extensive terminology often seen when discussing unoriented components. [BMS08] construct a nice tree data structure to solve this problem that also uses a slightly simpler terminology.

Two overlapping components are said to be *linked.* Successive linked components form a *chain* and a chain that cannot be extended in either direction are said to be *maximal.* We define a forest that represents the nesting of components and chaining

of components as follows:

Definition 2.4.2 ([BMS08]). *Given a chromosome* $\mathbb{X}$ *of a genome* $\mathbb{G}$ *and its relative to genome* $\mathbb{H}$, *define the forest* $\mathcal{F}_{\mathbb{X}}$ *by the following construction:*

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

We combine the above forests into a tree as follows:

Definition 2.4.3 ([BMS08]). *Suppose a genome* $\mathbb{G}$ *consists of chromosomes* $\{\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_k\}$. *The* tree $\mathcal{T}$ associated to the components of the genomes $\mathbb{G}$ and genome $\mathbb{H}$ *is given the following construction:*

1. *The root is a round vertex;*
2. *All trees of the set of forests* $\{\mathcal{F}_{\mathbb{X}_1}, \mathcal{F}_{\mathbb{X}_2}, \ldots, \mathcal{F}_{\mathbb{X}_k}\}$ *are children of the root;*

The round vertices are *colored* according to the following rules:

1. The root and all vertices corresponding to oriented components are *black*;
2. All vertices associated with unoriented components that contain a telomere are *white*;
3. All vertices associated with other unoriented components are *grey*;

A *long branch* in the tree is a branch with two or more vertices associated with unoriented components. A tree is called a *fortress* if it has an odd number of leaves all of them on long branches.

If the associated tree between genomes $\mathbb{G}$ and $\mathbb{H}$ with w white vertices and g grey vertices that isn't have a fortress then the HP distance is:

$$\text{DISTANCE}_{HP}(\mathbb{G}, \mathbb{H}) = \text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) + w + \left\lceil \frac{g}{2} \right\rceil \quad (2.4.4)$$

Similarly, if the associated tree is a fortress then the HP distance is:

$$\text{DISTANCE}_{HP}(\mathbb{G}, \mathbb{H}) = \text{DISTANCE}_{DCJ}(\mathbb{G}, \mathbb{H}) + w + \left\lceil \frac{g}{2} \right\rceil + 1 \quad (2.4.5)$$

2.4.2 Real Components

In the case of genome halving there isn't much that can be done to eliminate the effects of unoriented components. The problem has to do with the fact that there are two copies of every gene in the duplicated genome. Because there are two copies of every gene there are components within the duplicated genome, some of which may be unoriented. This is easy to see in simple genomes, which can easily be bipartitioned into two genomes, but occurs in non-simple genomes as well. Given a tetraploid $\mathbb{H}$ created from a duplicated genome $\mathbb{G}$, if $\mathbb{C}$ is a component between $\mathbb{H}$ and $\mathbb{G}$ and $\mathbb{C}$ is also a component between $\mathbb{G}$ and itself then it is called a *real component.*

Real components are unavoidable; there is no way to construct a tetraploid where each gene duplicate exactly matches each duplicate in the original duplicated genome because the gene duplicates in the original genome don't match. As a corollary, there is no way to avoid an unoriented real component either.

While nothing can be done about real components, not all components are real components and even real components can be mitigated. Both situations use the same algorithm, Algorithm 2.11. Algorithm 2.11 ensures that there is no more than one non-trivial component per sequence of duplicated genes. If a component is not a real component then Algorithm 2.11 eliminates it.

HP distance is only defined for linear multichromosomal genomes (linear unichromosomal genomes are included in this category) and circular unichromosomal genomes.

Algorithm 2.11: SPOIL

Input: A duplicated genome $\mathbb{G}$ and a tetraploid $\mathbb{H}$ derived from $\mathbb{G}$.
Output: A tetraploid $\mathbb{H}'$ with the number of components between $\mathbb{G}$ and $\mathbb{H}'$ minimized.

```
1  H' ← H
2  let C_1, ..., C_n be the components between G and H'
3  foreach component C_i do
4      let C̄_i be complement of C_i containing an extremity x_{3-j} for every
       extremity x_j in C_i
5      let S be the sequence in G contains the same extremities as C_i
6      let S̄ be complement of S containing an extremity x_{3-j} for every extremity
       x_j in C_i
7      remove C_i and C̄_i from H'
8      add S and S̄ to H'
9  end
10 return H'
```

While Algorithm 2.11 works for both cases, there is more to consider as each requires a slightly different strategy. Their strategies are outlined in their respective sections below.

2.4.3 Linear Multichromosomal Genomes

When the input genome is a linear multichromosomal genome there is more that can be done over Algorithm 2.11. This is because not all real components are real components in a linear multichromosomal genomes, some are *potential real components*, *potential components* for short.

A *potential component* is an otherwise *real component* that contains one or more telomeres. By merging chromosomes in the tetraploid it is possible to reduce the number of components while only slightly increasing the DCJ distance between the genomes. Since bad components containing telomeres increase the distance the most (they correspond to the white vertices described in Section 2.4.1), if done correctly, the increase in DCJ distance pays off with a reduction of bad components.

Algorithm 2.12: AMALGAMATE

Input: A duplicated genome $\mathbb{G}$ and a tetraploid $\mathbb{H}$ derived from $\mathbb{G}$.
Output: A tetraploid $\mathbb{H}'$ with the number of potential components between $\mathbb{G}$ and $\mathbb{H}'$ minimized.

```
1  H' ← H
2  let E_1, ..., E_e be the even components between G and H'
3  for i ← 1 to ⌊e/2⌋ do
4      let P_1, P_2 be the even paths associated with E_{2·i}
5      let P_3, P_4 be the even paths associated with E_{2·i−1}
6      H' ← MERGE(P_1, P_3)
7      H' ← MERGE(P_2, P_4)
8  end
9  if e is odd then
10     let P_1, P_2 be the even paths associated with E_e
11     H' ← MERGE(P_1, P_2)
12 end
13 sort the unoriented odd components from fewest associated graphs to most
   associated graphs
14 foreach unoriented odd component O_i between H' and G do
15     foreach associated path P_j do
16         let P_k be a path in an unoriented odd component O_l that has not yet
           been merged
17         H' ← MERGE(P_j, P_k)
18     end
19 end
20 return H'
```

A potential component can contain one or two telomeres. Interestingly any potential component must exactly correspond to one or two components, specifically paths, of the intersection graph[EMS03]. These components are called the *associated paths.* If the associated path is an even path then there are necessarily two associated paths[EMS03]; such potential components are called *even.* There can be one or two odd paths as associated paths; such potential components are called *odd.* [3]

Algorithm 2.12 optimally combines the linear chromosomes to minimize the num-

[3]This varies slightly from the El-Mabrouk-Sankoff notation. Out odd potential components are her even potential components and *vice versa.* This is because she counts vertices while we count edges

ber of potential components. As with most things concerning unoriented components, its proof of correctness is far too complex and, thus, we omit it. We refer brave readers can read [EMS03]. However, we will none-the-less give an overview of the algorithm with some of the rational as to why it is correct.

There are two parts to Algorithm 2.12. Lines 2 – 12 describe the first part. This part amalgamates two even components, and their chromosomes, by creating an adjacency that contains extremities from both components, a task we delegate to an algorithm MERGE. The algorithm MERGE is tedious so we avoid describing it other than to say that in the case of even paths there is always an optimal way to amalgamate the paths without increasing the DCJ distance.

Pairing up and amalgamating the even components eliminates all but one of them. If one remain we can amalgamate its two natural graphs together in such a way as to orient the components and keep the two chromosomes that the paths represent linear and separate without increasing the DCJ distance, but again, we omit the exact details.

While all potential unoriented even components can be merged and eliminated without increasing the distance, the same is not true for odd components. It is impossible to merge two odd components without increasing their DCJ distance. Thus, unlike even components where it doesn't matter if component is oriented or unoriented, with odd components it is only worth while to eliminate them unless they are unoriented.

The second part of the algorithm, Lines 13 – 19 describe how to handle unoriented odd components. This situation is a bit more complex since not all of the components have two associated paths. Of particular importance is Line 13, which insures that, when possible, if an associated graph remains unmerged it will be an associated graph belonging to a component with two associated graphs.

Obviously, if any unoriented odd components are present the distance does change inspite of the mitigating effects of algorithm 2.12. Providing an exact distance

formula is again complex so we omit it and again refer readers back to [EMS03].

2.4.4 Circular Unichromosomal Genomes

Circular unichromosomal genomes don't have telomeres so they don't have potential components. As a consequence, aside from Algorithm 2.11, nothing can be done about unoriented components.

However, there are still special considerations when eliminating the unoriented components of unichromosomal genomes. Specifically, unichromosomal genomes components have *special components*, which are components that contain both a gene x_i and its duplicate x_{3-i}. Special components are never real components but like real components special components cannot be avoided. Fortunately, because the output of a halving algorithm is a tetraploid, if there are special components there will be at most two of them.

Beyond being aware of the existence of special components there is nothing else that can be done. However, they can be easily predicated and computed into the distance formula prior to halving; see [EMS03] for details.

2.5 Summary

As we have seen, each genome halving algorithm, with the possible exception of final two which optimize for HP distance, are straightforward and simple. And if the algorithms for optimizing the genome halving for HP distance is complex it is only because HP distance is complex. The problem with many of the earlier attempts at genome halving is that they attempted to combine the steps as well as handling the exceptional case of odd cycles in the intersection graph along with the more general case of intersection graphs without odd cycles.

The genome halving algorithm forms the basis for all rearrangement-duplication

algorithms. With a solid understanding of this algorithm it is possible to modify it to handle other rearrangement-duplication problems. In the next chapter we introduce our contribution, the genome aliquoting problem, which is a generalization of the genome halving problem.

Chapter 3

Genome Aliquoting

Introduced just over a year ago, the genome aliquoting problem is the newest variation of the rearrangement-duplication problem. Despite being so new, two papers have already been published on the topic by separate sets of authors; the results of both are explored in this chapter. It is, however, too early to tell if it will become a popular problem.

The genome aliquoting problem generalizes genome halving to include genomes with gene families of size greater than 2, for which no exact algorithm is known for DCJ distance. The hope for genome aliquoting is that the exact algorithms for genome halving might somehow carry over, but unfortunately, this has not been the case. Under DCJ, the only known algorithms are approximations. However, for two other distance metrics, *breakpoint distance* and *single cut or join* distance, exact polynomial time algorithms are known.

3.1 Warren-Sankoff Algorithm

After building a genome halving algorithm using well-studied algorithms and data structures in [WS09b], Warren and Sankoff realized that it might in fact be possible to use that algorithm to transform any genome with gene families of equal size into a polyploid of corresponding ploidy. However, the Warren-Sankoff algorithm outlined in Algorithm 2.4 doesn't generalize easily. Undaunted, Warren and Sankoff refined

the approach in [WS09a] but ultimately were only able to produce a heuristic. In this section we improve upon those results by showing that the heuristic is an exact algorithm for *breakpoint distance* and an approximate algorithm for DCJ distance.

There are three problems that occur when running Algorithm 2.4 on polyploids. The first problem is in regards to step 2 of Algorithm 2.4: the computation of the independent set. As it is written, Algorithm 2.4 uses Algorithm 1.1, a variation of the general independent set algorithm optimized for graphs with a degree of at most 2. However, as the following example illustrates the vertices of an intersection graph of a polyploid don't necessarily have a degree of at most 2: consider the rearranged hexaploid depicted in Figure 3.1a, $\mathbb{G} = \{\{\overrightarrow{a_1}\}, \{\overrightarrow{a_1}, \overrightarrow{b_1}\}, \{\overrightarrow{b_1}, \overrightarrow{c_1}\}, \{\overrightarrow{c_1}, \overrightarrow{a_2}\}, \{\overrightarrow{a_2}, \overrightarrow{c_2}\}, \{\overrightarrow{c_2}, \overrightarrow{d_1}\}, \{\overrightarrow{d_1}\}, \{\overrightarrow{d_2}\}, \{\overrightarrow{d_2}, \overrightarrow{c_3}\}, \{\overrightarrow{c_3}, \overrightarrow{a_3}\}, \{\overrightarrow{a_3}, \overrightarrow{b_2}\}, \{\overrightarrow{b_2}\}, \{\overrightarrow{b_3}\}, \{\overrightarrow{b_3}, \overrightarrow{d_3}\}, \{\overrightarrow{d_3}\}\}$, with three chromosomes and four gene families of size 3, the vertices in its intersection graph, depicted in Figure 3.1b, have degrees of up to 4. In fact, given a genome that is a rearranged polyploid with each gene family of size p, it is easy to see that from the definition of an intersection graph, its vertices have a degree of either $p - 1$ or $2 \cdot (p - 1)$. Thus, since any generalization beyond genome halving will have a ploidy ≥ 3, Algorithm 2.4 is not accurate.

Of course, the use of Algorithm 1.1 is more of an optimization than a necessity. It is easy to use a more general independent set algorithm instead of Algorithm 1.1 and, moreover, this is more in keeping with the intent of the Warren-Sankoff algorithm. Unfortunately, this presents another problem: the general maximum independent set problem is NP-hard. Thus, with this minor modification, Algorithm 2.4 has gone from running in linear time to running in exponential time.

However, an exponential running time is the least of the Warren-Sankoff algorithm's problems. Two maximum independent sets can lead to very different distances. The problem is that, for genome halving, determining if a genome is perfect is boolean: a genome either is or is not perfect. However, for genome aliquoting this is no longer correct: some genomes are more perfect than others.

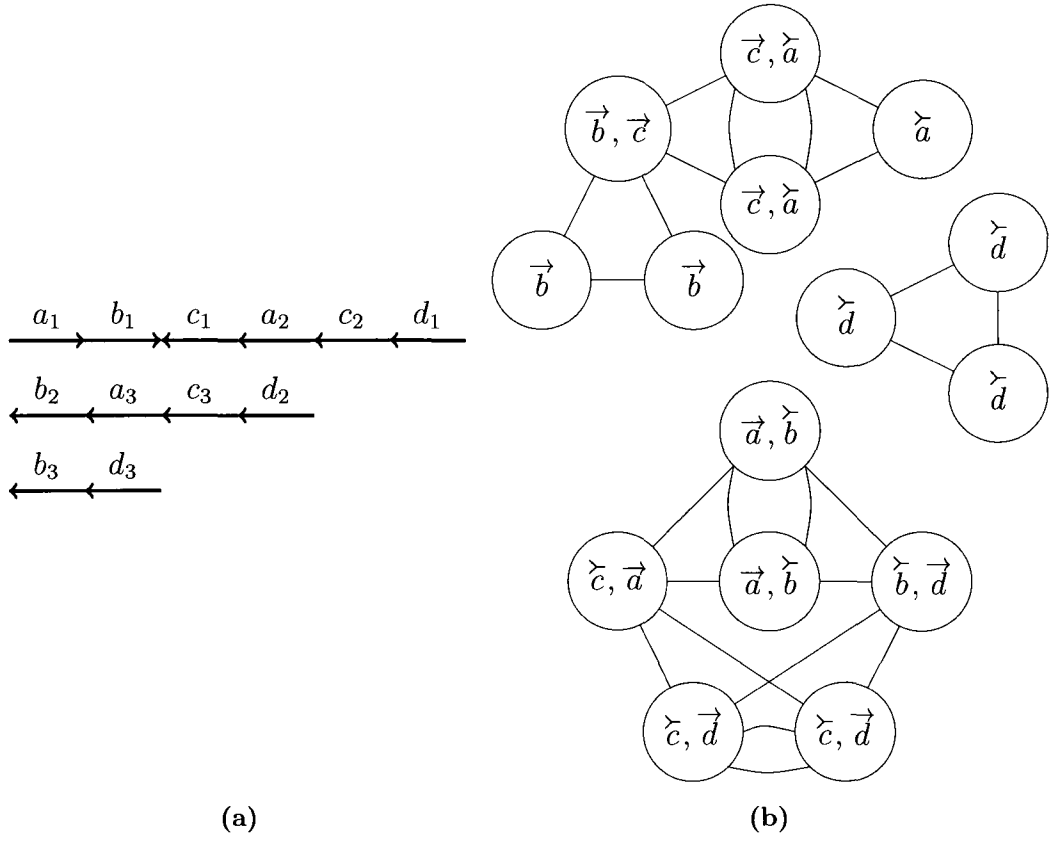


Figure 3.1: (a) A genome digraph of a rearranged duplicated genome with gene families of size 3. (b) An intersection graph of a rearranged duplicated genome with gene families of size 3, observe that every vertex has a degree of either 2 or 4.

For similar reasons, Algorithm 2.6 REORDER also no longer works. REORDER depends on the binary nature of tetraploids to correctly reconstruct the order of the genome. With polyploids a great deal of ambiguity is introduced and it can not necessarily be solved by an arbitrary ordering. Interestingly, this problem persists even if we avoid unordering the genomes to begin with; after aliquoting an ordered genome there are still ambiguous extremities.

3.1.1 Breakpoint Distance

It is not all bad news for genome aliquoting. With a slight modification Algorithm 2.4 can give an exact solution to genome aliquoting. Unfortunately, it solves a slightly different instance of the genome alquoting problem: it solves it for *breakpoint* distance rather than double cut or join distance.

Breakpoint distance, introduced by [SB97], is one of the simplest distance measures and is useful as an upper bound on other distance measures. Breakpoint distance is different from the other distances studied because it isn't linked to any specific evolutionary operations. As we will see, breakpoint distance and DCJ distance are closely related and extending the results for DCJ distance to include breakpoint distance is possible.

A *breakpoint* (*BP*) is a difference in adjacencies or telomeres between two genomes, *e.g.* in $\mathbb{G} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \overset{\succ}{b_1}\}, \{\vec{b_1}, \overset{\succ}{d_1}\}, \{\vec{d_1}, \overset{\succ}{e_1}\}, \{\vec{e_1}, \overset{\succ}{b_2}\}, \{\vec{b_2}\}, \{\vec{a_2}\}, \{\overset{\succ}{a_2}, \vec{c_1}\}, \{\overset{\succ}{c_1}, \overset{\succ}{c_2}\}, \{\vec{c_2}, \vec{d_2}\}, \{\overset{\succ}{d_2}, \overset{\succ}{e_2}\}, \{\vec{e_2}\}\}$ and $\mathbb{H} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \vec{b_1}\}, \{\overset{\succ}{b_1}, \vec{c_1}\}, \{\overset{\succ}{c_1}, \vec{d_1}\}, \{\overset{\succ}{d_1}, \vec{e_1}\}, \{\overset{\succ}{e_1}, \vec{a_2}\}, \{\overset{\succ}{a_2}, \vec{b_2}\}, \{\overset{\succ}{b_2}, \vec{c_2}\}, \{\overset{\succ}{c_2}, \vec{d_2}\}, \{\overset{\succ}{d_2}, \vec{e_2}\}, \{\overset{\succ}{e_2}\}\}, \{\vec{b_1}, \overset{\succ}{d_1}\}$ is a breakpoint in $\mathbb{G}$ since there is no equivalent element in $\mathbb{H}$ but $\{\vec{a_1}\}$ is not a breakpoint because it is in both genomes. Extending this notion to entire genomes we arrive at the following definition:

Definition 3.1.1. *The* breakpoint distance *between two genomes* $\mathbb{A}$ *and* $\mathbb{B}$ *is the number of breakpoints between* $\mathbb{A}$ *and* $\mathbb{B}$.

The breakpoint distance is easy to calculate. From [TZS09], given two genomes $\mathbb{A}$ and $\mathbb{B}$, let $\textsc{Adjacencies}(\mathbb{A}, \mathbb{B})$ be the set of adjacencies shared between $\mathbb{A}$ and $\mathbb{B}$ and let $\textsc{Telomeres}(\mathbb{A}, \mathbb{B})$ be the set of telomeres then the *breakpoint distance* between $\mathbb{A}$ and $\mathbb{B}$ is can be calculated using the following equation:

$$\textsc{Distance}_{BP}(\mathbb{A}, \mathbb{B}) = |\textsc{Genes}(\mathbb{A}, \mathbb{B})| - |\textsc{Adjacencies}(\mathbb{A}, \mathbb{B})| - \frac{|\textsc{Telomeres}(\mathbb{A}, \mathbb{B})|}{2} \quad (3.1.2)$$

For example, given two duplicated genomes $\mathbb{G} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \overset{\succ}{b_1}\}, \underline{\{\vec{b_1}, \overset{\succ}{d_1}\}}, \underline{\{\vec{d_1}, \overset{\succ}{e_1}\}}, \underline{\{\vec{e_1}, \overset{\succ}{b_2}\}}, \underline{\{\vec{b_2}\}}, \underline{\{\vec{a_2}\}}, \underline{\{\overset{\succ}{a_2}, \vec{c_1}\}}, \underline{\{\overset{\succ}{c_1}, \overset{\succ}{c_2}\}}, \underline{\{\vec{c_2}, \vec{d_2}\}}, \underline{\{\overset{\succ}{d_2}, \overset{\succ}{e_2}\}}, \underline{\{\vec{e_2}\}}\}$ and $\mathbb{H} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \vec{b_1}\}, \underline{\{\overset{\succ}{b_1}, \vec{c_1}\}}, \underline{\{\overset{\succ}{c_1}, \vec{d_1}\}}, \underline{\{\overset{\succ}{d_1}, \vec{e_1}\}}, \underline{\{\overset{\succ}{e_1}, \vec{a_2}\}}, \underline{\{\overset{\succ}{a_2}, \vec{b_2}\}}, \underline{\{\overset{\succ}{b_2}, \vec{c_2}\}}, \underline{\{\overset{\succ}{c_2}, \vec{d_2}\}}, \underline{\{\overset{\succ}{d_2}, \vec{e_2}\}}, \underline{\{\overset{\succ}{e_2}\}}\}$ the breakpoints are underlined; since $\text{ADJACENCIES}(\mathbb{A}, \mathbb{B}) = 1$ and $\text{TELOMERES}(\mathbb{G}, \mathbb{H}) = 1$ and $\text{GENES}(\mathbb{G}, \mathbb{H}) = 10$, $\text{DISTANCE}_{BP}(\mathbb{G}, \mathbb{H}) = 8.5$.

Genome halving with BP certainly has a polynomial time algorithm. [TZS09] state that since both HP distance and DCJ distance have proven polynomial time algorithms for mixed circular and linear multichromosomal genomes that it follows that BP distance must also have a polynomial time algorithm but they do not provide an algorithm. In fact, looking at Equation 3.1.2 and Equation 1.3.6 we can see that BP distance and DCJ distance are almost identical. In fact, BP distance is just DCJ distance except that it doesn't consider cycles (in the Adjacency Graph) with more than 2 edges and paths with more than 1 edge, *i.e.* it doesn't consider cycles and paths, only identical elements.

3.1.2 Generate Modified Clique Graphs

It is clear from the definition of breakpoint distance that to compute the optimal genome aliquoting for it we must maximize the number of similar adjacencies and telomeres between the given rearranged genome and the constructed perfectly duplicated genome. Thus, like the genome halving problem for DCJ, to solve the genome aliquoting problem for BP we must find a perfect subset of the genome. This was done by finding the maximum independent set during the genome halving problem, but as we already discussed, finding the maximum independent set is no longer a simple operation due to the higher degree of the intersection graph. Therefore, to efficiently solve the genome aliquoting problem for BP we must find an efficient way to find the maximum independent set.

While most of the useful properties of the intersection graph, including its in-

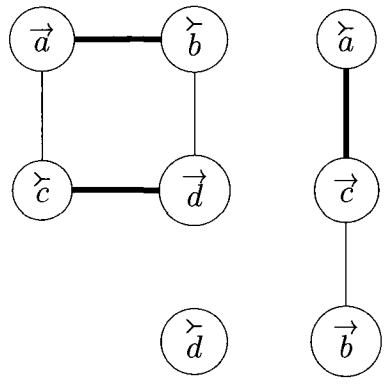


Figure 3.2: A clique graph describing the intersection graph depicted in Figure 3.1a. The three bold edges represent all the edges that belong to a maximum matching of this graph.

dependent set, are obfuscated when the ploidy of the graph is increased to three or more, one property is still very clear: the graph's cliques. Every extremity of the genome corresponds to one clique in the intersection graph. Thus, to capitalize on this, we create a clique graph from the intersection graph according to the following definition.

Definition 3.1.3. *Let $\mathcal{IG}$ be an intersection graph constructed from a genome $\mathbb{G}$. A* clique graph *$\mathcal{CG}$ is a graph whose vertices are the unique extremities of the unordered genome $\tilde{\mathbb{G}}$ and there exists an edge $\{u, v\}$ if there exists an adjacency $\{u, v\}$ in $\tilde{\mathbb{G}}$.*

Observant readers may notice that, in order to compute the clique graph, the intersection graph is not needed. We can construct the clique graph directly from the genome.

For example, consider the unordered genome $\tilde{\mathbb{G}} = \{\{\overleftarrow{a}\}, \{\vec{a}, \overleftarrow{b}\}, \{\vec{b}, \vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{a}, \vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}\}, \{\overleftarrow{d}\}, \{\vec{d}, \overleftarrow{c}\}, \{\vec{c}, \overleftarrow{a}\}, \{\vec{a}, \overleftarrow{b}\}, \{\vec{b}\}, \{\vec{b}\}, \{\overleftarrow{b}, \vec{d}\}, \{\overleftarrow{d}\}\}$ whose intersection graph is depicted in Figure 3.1b. The vertices of $\mathcal{CG}(\tilde{\mathbb{G}})$ consist of one copy of each extremity: $\{\overleftarrow{a}, \vec{a}, \overleftarrow{b}, \vec{b}, \overleftarrow{c}, \vec{c}, \overleftarrow{d}, \vec{d}\}$. The edges of $\mathcal{CG}(\tilde{\mathbb{G}})$ consist of the genome's adjacencies: $\{\{\vec{a}, \overleftarrow{b}\}, \{\vec{b}, \vec{c}\}, \{\overleftarrow{c}, \vec{a}\}, \{\overleftarrow{c}, \vec{d}\}, \{\vec{c}, \overleftarrow{a}\}, \{\overleftarrow{b}, \vec{d}\}\}$. The clique graph is a graph, thus, there can be only one copy of each edge. Thus, we have only one copy of $\{\vec{a}, \overleftarrow{b}\}$, $\{\overleftarrow{c}, \vec{d}\}$ and $\{\overleftarrow{a}, \vec{c}\}$, instead of two. $\mathcal{CG}(\tilde{\mathbb{G}})$ is depicted in Figure

3.2.

Since at most one vertex from any clique can belong to a maximum independent set of the intersection graph, a maximum independent set of the clique graph could also be used to construct a maximum independent set of the intersection graph: for each vertex in the independent set of the clique graph select any one vertex from its corresponding clique. Unfortunately, this does not solve our problem as finding the independent set of the clique graph is no easier than finding the independent set of the intersection graph. However, we have more information; the edges of the clique graph correspond to vertices in the intersection graph.

A maximum independent set of edges in the clique graph would also be a maximum independent set (of vertices) in the intersection graph. As discussed in Section 1.2.3, this is the maximum matching problem and it has a polynomial time algorithm. Thus, we can find the maximum independent set of the intersection graph in polynomial time but, unfortunately, there is still one important caveat: only those vertices that correspond to adjacencies in the genome are represented, those that correspond to telomeres would be missing.

For example, the bold edges depicted in Figure 3.2 correspond to a maximum matching. These edges, in turn, corresponds to the independent set $\{\{\overrightarrow{a}, \overleftarrow{b}\}, \{\overleftarrow{c}, \overrightarrow{d}\}, \{\overleftarrow{a}, \overrightarrow{c}\}\}$ in the intersection graph of Figure 3.1b. Unfortunately, this independent set is not a maximum, or even maximal, independent set as the vertices that correspond to telomeres $\overrightarrow{b}$ and $\overleftarrow{d}$ can be added to the independent set.

To solve this problem we can simply create a new vertex for cliques that contain telomeres. This new vertex would be connected to its corresponding clique (and the clique graph in general) by a single edge. Because edges in the intersection graph correspond to adjacencies that intersect, telomeres will not intersect with anybody outside of their clique and, thus, connecting the new vertex to its corresponding clique is consistent. In effect, we can imagine that every telomere is an actually an adjacency but with one extremity being "null", hence, we call these new vertices

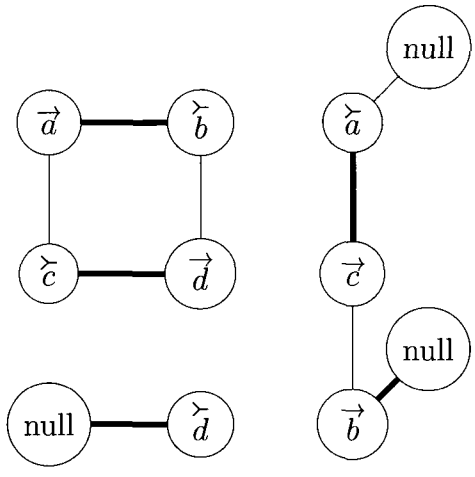


Figure 3.3: A modified clique graph describing the intersection graph depicted in Figure 3.1a. The five bold edges represent all the edges that belong to a maximum matching of this graph.

null vertices. We occasionally refer to all other vertices as *non-null vertices.* We call the clique graph with null-vertices a *modified clique graph* and provide the function $\mathcal{MCG}(\tilde{\mathbb{G}})$ that produces the modified clique graph of an unordered genome $\tilde{\mathbb{G}}$. Figure 3.3 depicts the modified clique graph of a genome.

Figure 3.3 extends Figure 3.2 to include null vertices that correspond to telomeres $\{\overrightarrow{a}\}$, $\{\overrightarrow{d}\}$ and $\{\overrightarrow{b}\}$. Now the maximum matching, represented by the bold edges, does correspond to a maximum independent set, $\{\{\overrightarrow{a}, \overrightarrow{b}\}, \{\overrightarrow{c}, \overrightarrow{d}\}, \{\overrightarrow{a}, \overrightarrow{c}\}, \{\overrightarrow{d}\}, \{\overrightarrow{b}\}\}$, of the intersection graph depicted in Figure 3.1b.

While each edge in the clique graph now corresponds to either and adjacency or telomere in the intersection graph and we can use this information to reconstruct the independent set, we still have a problem. As alluded to in the introduction, since we can no longer depend on the binary nature of tetraploids, some perfect sets are more perfect than others. From Equation 3.1.2 we can see that the distance is minimized by maximizing the number of adjacencies and telomeres present in both genomes, but a perfect set only ensures that any adjacencies or telomeres present with be present in both genomes; it provides no guarantees that such a set minimizes the breakpoint distance. Thus, we need a stronger type of perfect set, a *maximum perfect set* of a

genome $\tilde{\mathbb{G}}$, P, is a subset of $\tilde{\mathbb{G}}$ that is perfect and of maximum *weight*:

$$\text{WEIGHT}(P) = |\text{ADJACENCIES}(\tilde{\mathbb{G}}, P)| + \frac{|\text{TELOMERES}(\tilde{\mathbb{G}}, P)|}{2} \quad (3.1.4)$$

Using the maximum matching of the modified clique graph/independent set of the intersection graph to compute the perfect matching does not guarantee a maximum perfect set since it only selects at most one vertex from each clique in the intersection graph even though the adjacencies or telomeres of two or more vertices in the same clique might be perfect.

There are perfect sets, maximum perfect sets in particular, that do not correspond to independent sets of the intersection graph. However, every maximum perfect set does correspond to a set of vertices that do have an independent set of the intersection graph as a subset. Thus, we needn't abandon the idea of finding an independent set in order to find the maximum perfect sets, but we do need to find an independent set that is the subset of a set of vertices that corresponds to the maximum perfect set and then we need to find the missing vertices. Fortunately, this is easier then it seems.

Lemma 3.1.5. *The relationship between perfect distinct elements α and β of a genome $\tilde{\mathbb{G}}$ and the modified clique graph $\mathcal{MCG}(\tilde{\mathbb{G}})$ is as follows:*

1. *If α and β are adjacencies then they are perfect if and only if they correspond to the same edge in $\mathcal{MCG}(\tilde{\mathbb{G}})$ or there exists a matching that contains the corresponding edges of both elements;*

2. *If α and β are telomeres then they are always perfect and there exists a matching such that their corresponding edges are both elements;*

3. *If α is a telomere and β is an adjacency (or* vice versa*) then they are perfect if and only if there exists a matching that contains the corresponding edges of both elements;*

Proof. We examine each case in order below:

1. Assume that α and β are both adjacencies.

   If α and β are perfect then either $\alpha = \beta$ or $\alpha \cap \beta = \emptyset$ and neither α nor β are of the form $\{w, w\}$ for some extremity w.

   If $\alpha = \beta$ then let $\{x, y\} = \alpha = \beta$ where x and y are extremities and $x \neq y$. Since each extremity corresponds to exactly one clique and $x \neq y$, x and y correspond to two different vertices, u and v respectively, in the clique graph. From the definition of a clique graph, α corresponds to an edge $\{u, v\}$ in the clique graph. Similarly, β also corresponds to an edge $\{u, v\}$ in the clique graph. Hence, they both correspond to the same edge.

   If $\alpha \cap \beta = \emptyset$ then, since each extremity corresponds to a vertex and each element an edge in the clique graph, α and β correspond to edges with no vertices in common. Hence, the set containing the edge corresponding to α and the edge corresponding to β is a matching. Thus, there exists a matching containing the corresponding edges of both elements.

   Assume that either α and β correspond to the same edge in the clique graph or there exists a matching that contains the corresponding edges of both α and β.

   If α and β correspond to the same edge in the clique graph then let $\{u, v\}$ be the edge to which they correspond. Since each vertex in the clique graph corresponds to an extremity let x and y be the extremities to which u and v correspond respectively. It follows that $\alpha = \beta = \{x, y\}$. Since there are no self loops in a clique graph, $x \neq y$. Hence, α and β are perfect.

   Assume that there exists a matching that contains the corresponding edges of both α and β. Thus, the two edges share no vertices in common. Since, vertices correspond to extremities, α and β share no extremities in common. Hence, $\alpha \cap \beta = \emptyset$ meaning α and β are perfect.

2. Assume that α and β are both telomeres. By definition, they are both perfect. Exactly one edge connects a null vertex to the rest of the graph; every telomere corresponds to one of these edges. If α and β correspond to the same edge then there trivially exists a matching that contains it. If α and β correspond to different edges then, for certain, these edges don't share the same null vertex. Since there is exactly one non-null vertex for every null vertex and each null vertex is connected with a different non-null vertex, the non-null vertices to which α and β correspond must be different. Hence, the edges share no vertices in common and, thus, a set containing both edges is a matching. Therefore, there exists a matching that contains both α and β.

3. Assume, without a loss of generality that α is a telomere and β is an adjacency. If α and β are perfect then $\alpha \cap \beta = \emptyset$. For certain α and β correspond to different edges, since one is a telomere and the other is not. Thus, by similar reasoning to the case where α and β are adjacencies such that $\alpha \cap \beta = \emptyset$, there exists a matching containing the edges corresponding to α and β.

   Assume there exists a matching that contains the edges corresponding to α and β. Again, by the same reasoning to the similar case when α and β are adjacencies, α and β must be perfect. □

From Lemma 3.1.5 there are two criteria for determining if two adjacencies are perfect. We already account for the second criteria, that there exists a matching that contains both of their corresponding edges in our algorithm since our algorithm deduces a perfect set from a matching. However, our algorithm overlooks elements of the perfect set that correspond to the first case. To find such elements, we simply weight the edges according to how many adjacencies share it. We can then find the maximum weight matching, instead of the maximum matching, to account for both criteria at the same time.

Even though according to Lemma 3.1.5 the number of telomeres sharing the same edge doesn't seem to matter, since the maximum weight matching might contain both edges corresponding to telomeres and the edges corresponding to adjacencies it is important that telomeres have weights so that they can be accurately compared. However, there is one caveat to weighting the edges that correspond to telomeres. From Equation 3.1.2, telomeres are weighted half as much as adjacencies. Hence, the weights of the edges that correspond to telomeres should be half of that of those that correspond to adjacencies. Adding weights to Figure 3.3, based off the genome $\tilde{\mathbb{G}} = \{\{\overrightarrow{a}\}, \{\overrightarrow{a}, \overrightarrow{b}\}, \{\overrightarrow{b}, \overrightarrow{c}\}, \{\overrightarrow{c}, \overrightarrow{a}\}, \{\overrightarrow{a}, \overrightarrow{c}\}, \{\overrightarrow{c}, \overrightarrow{d}\}, \{\overrightarrow{d}\}, \{\overrightarrow{d}\}, \{\overrightarrow{d}, \overrightarrow{c}\}, \{\overrightarrow{c}, \overrightarrow{a}\}, \{\overrightarrow{a}, \overrightarrow{b}\}, \{\overrightarrow{b}\}, \{\overrightarrow{b}\}, \{\overrightarrow{b}, \overrightarrow{d}\}, \{\overrightarrow{d}\}\}$, gives us the *modified weighted clique graph* depicted in Figure 3.4. Since there are two copies of adjacencies $\{\overrightarrow{a}, \overrightarrow{b}\}$, $\{\overrightarrow{c}, \overrightarrow{d}\}$ and $\{\overrightarrow{c}, \overrightarrow{a}\}$, the edges to which they correspond have weights 2. There is only one copy of all other adjacencies, hence, those adjacencies have a weight 1. There are three copies of telomere $\{\overrightarrow{d}\}$ so the corresponding edge has a weight of 1.5, since telomeres have half the weight of an adjacency. Similarly, there are two copies of telomere $\{\overrightarrow{b}\}$ so that edge has a weight of 1. The only other telomere, $\{\overrightarrow{a}\}$, has a single instance in the genome, hence, its edge has a weight of .5. The function $\mathcal{MWCG}(\tilde{\mathbb{G}}$ produces the modified weighted clique graph of an unodered genome $\tilde{\mathbb{G}}$.

Even with all the additional information added to the clique graph some information is still missing. In particular, adjacencies of the form $\{u, u\}$ are not considered in this graph. This is intentional. As described for genome halving in Section 2.3.1, such information doesn't help in the aliquoting of the graph so the algorithm simply ignores it.

3.1.3 Compute Maximum Weight Matching

Algorithm 3.1 finds the maximum perfect matching by finding the maximum weight matching of the modified weighted clique graph. It follows from Lemma 3.1.5 that

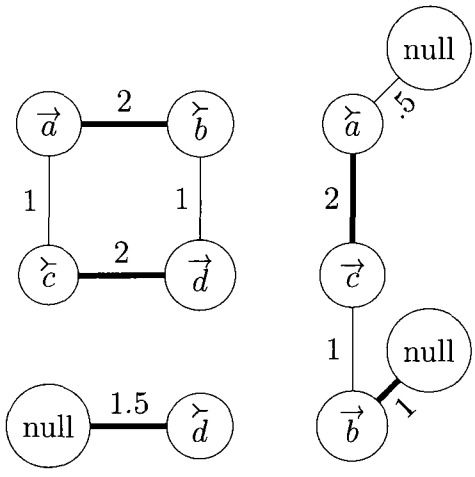


Figure 3.4: A modified weighted clique graph describing the intersection graph depicted in Figure 3.1a. The five bold edges represent all the edges that belong to the maximum weight matching of this graph.

Algorithm 3.1: PERFECTSET

Input: An unordered genome $\tilde{\mathbb{G}}$ and a matching M.
Output: A perfect set P.

```
1  P ← ∅
2  foreach edge e ∈ M do
3      let e = {u, v} for vertices u and v
4      if LABEL(v) = NULL then
5          add {LABEL(u)} to P
6      else if LABEL(u) = NULL then
7          add {LABEL(v)} to P
8      else
9          add {LABEL(u), LABEL(v)} to P
10     end
11 end
12 return P
```

Algorithm 3.1 constructs a perfect set, thus, to prove its correctness we need only prove that it is a maximum perfect set.

The modified weighted clique graph was defined such that the weight of the maximum weight matching is equal to the weight of the perfect set defined by Algorithm 3.1. We can formalize this relationship with the following trivial lemma:

Lemma 3.1.6. *Given an unordered genome $\tilde{\mathbb{G}}$, let M be a matching of $\mathcal{MWCG}(\tilde{\mathbb{G}})$ and let $P = \text{PERFECTSET}(\tilde{\mathbb{G}}, M)$:* $\text{WEIGHT}(P) = \text{WEIGHT}(M)$.

It is now possible to prove that maximum weight matchings correspond to maximum perfect sets:

Lemma 3.1.7. *Given a genome $\tilde{\mathbb{G}}$), let M be a matching of $\mathcal{MWCG}(\tilde{\mathbb{G}})$ and let $P = \text{PERFECTSET}(\tilde{\mathbb{G}}, M)$. If M is a maximum weighted matching then P must be a maximum perfect set.*

Proof. Assume towards contradiction that M is a maximum weight matching but P is not a maximum perfect set then there must exists a P' such that $\text{WEIGHT}(P) < \text{WEIGHT}(P')$. However, from Lemma 3.1.5 if follows that there exists a matching M' that corresponds to P'. From Lemma 3.1.6 it follows that $\text{WEIGHT}(M) < \text{WEIGHT}(M')$. But M is a maximum weight matching, a contradiction. □

For example, the maximum weight matching of the modified weighted clique graph depicted in Figure 3.4 is indicated by the bold edges in that figure. It corresponds to the maximum perfect set $\{\{\vec{a}, \overleftarrow{b}\}, \{\vec{a}, \overleftarrow{b}\}, \{\overleftarrow{c}, \vec{d}\},, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{a}, \vec{c}\}, \{\overleftarrow{a}, \vec{c}\}, \{\overleftarrow{d}\}, \{\overleftarrow{d}\}, \{\overleftarrow{d}\}, \{\vec{b}\}, \{\vec{b}\}\}$ of the genome $\tilde{\mathbb{G}} = \{\{\overleftarrow{a}\}, \{\vec{a}, \overleftarrow{b}\}, \{\vec{b}, \vec{c}\}, \{\overleftarrow{c}, \vec{a}\}, \{\overleftarrow{a}, \vec{c}\}, \{\overleftarrow{c}, \vec{d}\}, \{\overleftarrow{d}\}, \{\overleftarrow{d}\}, \{\vec{d}, \overleftarrow{c}\}, \{\vec{c}, \overleftarrow{a}\}, \{\vec{a}, \overleftarrow{b}\}, \{\vec{b}\}, \{\vec{b}\}, \{\overleftarrow{b}, \vec{d}\}, \{\overleftarrow{d}\}\}$.

Like with the genome halving problem, it is important that we find a subset of the genome that is both perfect and covering. Since the genome halving problem is a special case of the genome aliquoting problem it follows that because such a set does not necessarily exist in the genome halving problem it does not necessarily exist in the genome aliquoting problem. The solution to this problem in the genome aliquoting problem is also the same: do away with the restriction that it must be a subset.

Since we do not allow adjacencies of the form $\{u, u\}$, for some extremity u, into the aliquoted genome the solution is simple: instead of adding a null vertex with an edge to its corresponding non-null vertex for cliques that include telomeres in

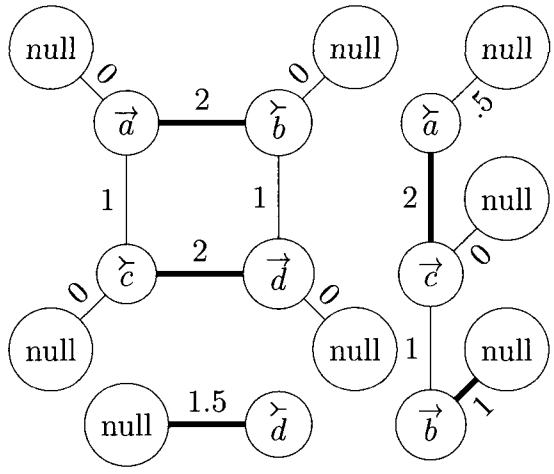


Figure 3.5: A fully modified weighted clique graph describing the intersection graph depicted in Figure 3.1a. The five bold edges represent all the edges that belong to the maximum weight matching of this graph.

the intersection graph, add such a vertex and edge for all cliques in the intersection graph (one for all non-null vertices in the clique graph). Since these edges do not correspond to any telomeres, their weight is 0. We call such a graph a *fully modified weighted clique graph*; $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ is the fully modified weighted clique graph of an unordered genome $\tilde{\mathbb{G}}$.

Without the restriction that the perfect set be a subset of $\tilde{\mathbb{G}}$, the idea of a maximum perfect set doesn't make any sense; we could add an infinite number of adjacencies and telomeres to the set if they don't have to come from the genome. Thus, we introduce a *quasi maximum perfect set*: a perfect set with a maximum perfect set as a subset but with additional elements not from $\tilde{\mathbb{G}}$. All maximum perfect sets are also quasi maximum perfect sets, although the reverse is not true.

While Lemma 3.1.7 doesn't apply to fully modified weighted clique graphs, however, since fully modified weighted clique graphs are not very different than modified weighted clique graphs, the result is essentially the same. However, because a fully modified weighted clique graph contains edges that don't correspond to elements of the genome the result might be a quasi maximum perfect set rather than a maximum

perfect set. Therefore, we generalize Lemma 3.1.7 as follows:

Lemma 3.1.8. *Given a genome $\tilde{\mathbb{G}}$, let M be a matching of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ and let $P = \textsc{PerfectSet}(\tilde{\mathbb{G}}, M)$. If M is a maximum weighted matching then P must be a quasi maximum perfect set.*

Proof. Since $\mathcal{MWCG}(\tilde{\mathbb{G}})$ is a subset of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$, let M' be the subset of M that contains only the edges in $\mathcal{MWCG}(\tilde{\mathbb{G}})$. Since all the edges that are in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ but not in $\mathcal{MWCG}(\tilde{\mathbb{G}})$ have weight 0, the edges in M' must be a maximum weight matching of $\mathcal{MWCG}(\tilde{\mathbb{G}})$.

Let $P' = \textsc{PerfectSet}(\tilde{\mathbb{G}}, M')$. Since M' is a maximum weight matching it follows from Lemma 3.1.7 that P' is a maximum perfect set. Since M' is a subset of M it follows that P' is a subset of P. Thus, P is a quasi maximum perfect set. □

Before we prove that there must exist a quasi maximum perfect set that covers the genome, it is important to understand what such a genome looks like. Non-null vertices are labeled with extremities in the genome (where as null vertices have a null label). Thus, any matching such that all of the non-null vertices in a graph are incident to an edge in the matching must correspond to a perfect set that covers the genome. Formally:

Lemma 3.1.9. *Given an unordered genome $\tilde{\mathbb{G}}$, let M be a maximum weight matching of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ and let $P = \textsc{PerfectSet}(\tilde{\mathbb{G}}, M)$. If all non-null vertices of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ are incident with edges in M then P covers $\tilde{\mathbb{G}}$.*

Proof. Each non-null vertex in the fully modified weighted clique graph is labeled with a single extremity from the genome such that all the extremities in the genome are used as a label of exactly one non-null vertex. Since an edge between two non-null vertices corresponds to an adjacency that contains the extremities on the label each incident vertex in P and an edge between a non-null vertex and a null vertex

corresponds to a telomere that contains the extremity on the label of the non-null vertex in P, all the extremities of $\tilde{\mathbb{G}}$ must be present in P. Hence, P covers $\tilde{\mathbb{G}}$. □

To be clear, the maximum weight matching in Lemma 3.1.9 is not the same as a *maximum weight perfect matching* nor is our definition of perfect in any way similar to the definition of perfect in *maximum weight perfect matching*. A *maximum weight perfect matching* refers to a maximum weight matching in which each vertex is incident with at least one edge in the matching. While at first this seems very similar to Lemma 3.1.9, the difference is that Lemma 3.1.9 is only concerned with *non-null* vertices that are incident with edges in the matching; null of vertices are of no interest.

Since our problem is different, we cannot take advantage of vast literature on maximum weight perfect matchings we must prove that there exists maximum weight matching that meets our criteria. Such a matching must exist, but not all maximum weight matchings meet our requirements:

Lemma 3.1.10. *Given an unordered genome $\tilde{\mathbb{G}}$, there exists M, a maximum weight matching of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$, such that every non-null vertex in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ is incident with an edge in M.*

Proof. Let M be a maximum weight matching of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$. If every non-null vertex in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ is incident with an edge in M. Otherwise, from M we will construct another maximum weight matching where every edge in the matching is incident with at least one non-null vertex.

Let u be a non-null vertex that is not incident with an edge in M. Let v be the null vertex adjacent to u; v must exist because, by definition, every non-null vertex in a fully modified weighted clique graph is incident with a unique null vertex. Since v is only adjacent with u, if u isn't incident with any edges in M then neither is v. Thus, if we add $\{u, v\}$ to M, M will still be a matching. Therefore, we construct M'

Algorithm 3.2: ExtendedMaximumWeightMatching

Input: A fully modified weighted clique graph $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ of a unordered genome $\tilde{\mathbb{G}}$.

Output: A maximum weight matching M of $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ where every non-null vertex in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ is incident with an edge in M.

```
1 M ← MaximumWeightMatching(FMWCG(G̃))
2 foreach non-null vertex v in FMWCG(G̃) do
3     if v is not incident with an edge in M then
4         let u be the null vertex adjacent to v
5         add {u, v} to M
6     end
7 end
8 return M
```

by adding all edges $\{u, v\}$ to M where u is a non-null vertex that is not incident with an edge in M and v is its adjacent to null vertex.

Since the weight all edges between non-null vertices and null vertices is greater than or equal to 0, and M' contains all the edges in M plus these edges, $\text{WEIGHT}(M') \geq \text{WEIGHT}(M)$. Because M is a maximum weight matching the weight cannot be greater, hence, $\text{WEIGHT}(M') = \text{WEIGHT}(M)$. Therefore, because M' is a matching with weight equal to the maximum weight matching, it must also be a maximum weight matching. Thus, there exists a maximum weight matching where each edge of the matching is incident with at least one non-null vertex. □

Even though not all maximum weight matchings meet our requirements, all maximum weight matchings can be transformed in linear time into a maximum weight matching that does correspond to a covering set. The proof of Lemma 3.1.10 does exactly this to prove that such a maximum weight matching exists, Algorithm 3.2 is a pseudo-code description of this process.

Combining Lemmas 3.1.8, 3.1.9 and 3.1.10 we can conclude the following theorem:

Algorithm 3.3: REPLICATE

Input: P quasi maximum perfect and covering set with respect to an unordered duplicated genome $\tilde{\mathbb{G}}$ and p, the number of copies of each gene present in $\tilde{\mathbb{G}}$.

Output: An unordered polyploid $\tilde{\mathbb{H}}$ with p copies of each gene.

```
1 foreach α ∈ P do
2     if MULTIPLICITY_P(α) ≠ p then
3         MULTIPLICITY_P(α) ← p
4     end
5 end
6 return P
```

Theorem 3.1.11. *Let* $M = \text{EXTENDEDMAXIMUMWEIGHTMATCHING}(\tilde{\mathbb{G}})$ *and* $P = \text{PERFECTSET}(\tilde{\mathbb{G}}, M)$, *where* $\tilde{\mathbb{G}}$ *is an unordered genome.* P *is a quasi maximum perfect and covering set of* $\tilde{\mathbb{G}}$.

3.1.4 Reorder Genes

To finish the genome aliquoting algorithm we must convert the perfect set into an ordered duplicated genome. Thus, we must "duplicate" the perfect set like in Section 2.1 and reorder the result like in Section 2.1.6. Unfortunately, the algorithms described in those sections, Algorithm 2.1 and Algorithm 2.6 respectively, cannot be directly applied to the aliquoting problem. Fortunately, like Algorithm 2.4 only minor modifications are needed to extend them to the genome aliquoting problem.

Algorithm 2.1 cannot be directly applied for the same reason that Algorithm 2.4 could not be directly applied: it assumes that there are only two copies of every gene. However, as Algorithm 3.3 illustrates, this is not a very serious problem and Algorithm 2.1 can be trivially extended to the general case by taking the number of duplicated genes as input.

To prove the correctness of the Algorithm 3.3 we will extend Theorem 2.1.2 with the following theorem:

Theorem 3.1.12. *Given* $\tilde{\mathbb{G}}$ *where* $\tilde{\mathbb{G}}$ *is a duplicated genome with* p *copies of every gene and* P *is a quasi maximum perfect and covering set with respect to* $\tilde{\mathbb{G}}$, $\tilde{\mathbb{H}} = \text{REPLICATE}(P, p)$ *is a polyploid with* p *copies of every chromosome such that* $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$.

Proof. There are three properties of $\tilde{\mathbb{H}}$ that we need to prove. First we need to prove that $\tilde{\mathbb{H}}$ is in fact an unordered genome. Second, that it is a polyploid with p copies of every chromosome. And third, that $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$.

The fact that $\tilde{\mathbb{H}}$ is unordered is trivial. $\tilde{\mathbb{H}}$ is a genome follows from the fact that $P \subseteq \tilde{\mathbb{H}}$ and P covers $\tilde{\mathbb{G}}$, a genome. $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$ also follows from this fact. Hence, $\tilde{\mathbb{H}}$ is an unordered genome and $\text{EXTREMITIES}(\tilde{\mathbb{H}}) = \text{EXTREMITIES}(\tilde{\mathbb{G}})$.

By definition, $\tilde{\mathbb{H}}$ is a polyploid with p copies of every chromosome if and only if it is perfect with p copies from every element. From Algorithm 3.3 it is clear that $\tilde{\mathbb{H}}$ has p copies of every element. Since P is perfect and $P \subseteq \tilde{\mathbb{H}}$ and, furthermore, the only difference between P and $\tilde{\mathbb{H}}$ is the multiplicity of the elements, which doesn't effect the property of perfect, $\tilde{\mathbb{H}}$ must also be perfect. □

Reordering the genome is even more tricky than it was in the halving problem, but, nevertheless we can still employ the same strategy as Algorithm 2.6, even though we cannot re-use the same algorithm. Thus, we introduce a new, more general, version of the REORDER algorithm in Algorithm 3.4.

Algorithm 3.4 works in two parts. First it determines which elements in the unordered genome are *similar* to elements in the ordered genome. By *similar*, we mean that when unordered the element from the ordered genome is identical to that of the unordered genome, *e.g.* an unordered adjacency $\{x, y\}$, where x and y are extremities, is similar to the ordered adjacency $\{x_2, y_4\}$. It adds all these similar adjacencies to the ordering of the unordered genome.

Algorithm 3.4: REORDER

Input: An unordered polyploid $\tilde{\mathbb{H}}$ with p chromosomes and an ordered genome $\mathbb{G}$ with p copies of every extremity where EXTREMITIES($\tilde{\mathbb{G}}$) = EXTREMITIES(UNORDER($\mathbb{G}$)).

Output: An ordered polyploid $\mathbb{H}$ such that DISTANCE$_{BP}(\mathbb{G}, \mathbb{H})$ is minimal.

```
1  ℍ ← ∅
2  foreach element α ∈ 𝔾 do
3      if UNORDER(α) ∈ ℍ̃ then
4          add α to ℍ
5      end
6  end
7  E ← EXTREMITIES(G) \ EXTREMITIES(H)
8  foreach element α ∈ ℍ̃ do
9      if α is an adjacency then
10         let {x, y} ← α where x, y are extremities
11         while there exists an i such that x_i ∈ E and a j such that y_j ∈ E do
12             add {x_i, y_j} to ℍ
13             remove x_i from E
14             remove y_j from E
15         end
16     else
17         let {x} ← α where x is an extremity
18         while there exists an i such that x_i ∈ E do
19             add {x_i} to ℍ
20             remove x_i from E
21         end
22     end
23 end
24 return ℍ
```

After finding similar adjacencies, Algorithm 3.4 determines which ordered extremities still have not already been used in the ordering of the genome. For the second part of the Algorithm 3.4, it uses the available ordered extremities to arbitrarily order the remaining adjacencies. To do this, we simply check the extremities of each (unique) element of the unordered genome and, for each unordered extremity in the element, we check to see if an unordered extremity is available. If so, we create

a new ordered element using the ordered extremities. If the element is an adjacency there will be two unordered extremities to order but they must have the same number of related ordered extremities remaining because the unordered genome is a polyploid.

The correctness of Algorithm 3.4 follows easily from Equation 3.1.2 for BP distance. The BP distance between two genomes is equal to the number of adjacencies and telomeres that the two genomes have in common, elements that they don't have in common don't count. Thus, if we maximize the number of elements that they have in common the rest of the genomes don't matter and can be ordered in any way that is convenient for us. Formally:

Theorem 3.1.13. *Given an unordered polyploid* $\tilde{\mathbb{H}}$ *with* p *chromosomes and an ordered genome* $\mathbb{G}$ *with* p *copies of every extremity where* $\textsc{Extremities}(\tilde{\mathbb{H}}) = \textsc{Extremities}(\textsc{Unorder}(\mathbb{G}))$, $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$ *where* $\mathbb{H} = \textsc{Reorder}(\tilde{\mathbb{H}}, \mathbb{G})$ *is minimal.*

Proof. $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$ is minimal if there doesn't exist another ordering of $\tilde{\mathbb{H}}$, say $\mathbb{H}'$, such that $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H}') < \textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$. Assume to towards contradiction that such an $\mathbb{H}'$ does exist.

From Equation 3.1.2 only adjacencies and telomeres that are the same in both genomes reduce the distance. Thus, because the distance between $\mathbb{G}$ and $\mathbb{H}'$ is lower than the distance between $\mathbb{G}$ and $\mathbb{H}$, $\mathbb{G}$ and $\mathbb{H}'$ must have at least one adjacency or telomere, say α, in common that are not in common between $\mathbb{G}$ and $\mathbb{H}$. Since $\textsc{Unorder}(\alpha)$ must be in $\tilde{\mathbb{H}}$, since $\mathbb{H}'$ is an ordering of $\tilde{\mathbb{H}}$. But from lines 2 – 6 in Algorithm 3.4, α must be in $\mathbb{H}$ too; a contradiction. Therefore, $\mathbb{H}$ is minimal. □

3.1.5 Implementation

While from Theorem 3.1.13 we can conclude that while Algorithm 3.4 optimally reorders the unordered genome given as input, it is ultimately the quality of the input genome that determines if the resulting reordered genome is an optimal aliquoting.

To complete our algorithm and our proof we will show that given a quasi maximum perfect and covering set as input, Algorithm 3.4 gives an optimal aliquoting as output.

Theorem 3.1.14. *Given $\mathbb{G}$ be an ordered rearranged genome with p copies of every gene, let P be a quasi maximum perfect and covering with respect to* $\textsc{Unorder}(\mathbb{G})$ *and let* $\tilde{\mathbb{H}} = \textsc{Replicate}(P, p)$*, then* $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$ *where* $\mathbb{H} = \textsc{Reorder}(\tilde{\mathbb{H}}, \mathbb{G})$ *is minimal.*

Proof. $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$ is minimal if there doesn't exist another genome $\mathbb{H}'$ such that $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H}') < \textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$. Assume to towards contradiction that such an $\mathbb{H}'$ does exist.

Let E be the set of elements that are identical between $\mathbb{H}$ and $\mathbb{G}$ and let E' be the set of elements that are identical between $\mathbb{H}'$ and $\mathbb{G}$. We observe that because $\mathbb{H}$ and $\mathbb{H}'$ are polyploids $\textsc{Unorder}(E)$ and $\textsc{Unorder}(E')$ must be a perfect sets of $\textsc{Unorder}(\mathbb{G})$.

From Equation 3.1.2 only adjacencies and telomeres that are the same in both genomes reduce the distance. Thus, since $\textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H}') < \textsc{Distance}_{BP}(\mathbb{G}, \mathbb{H})$, $\textsc{Weight}(\textsc{Unorder}(E')) > \textsc{Weight}(\textsc{Unorder}(E))$.

By definition, a quasi maximum perfect set must have a maximum perfect set as a subset; let $M \subseteq P$ be such a maximum perfect set of $\textsc{Unorder}(\mathbb{G})$. Since E is a subset of $\mathbb{H}$ it follows that $\textsc{Unorder}(E)$ must be a subset of $\tilde{\mathbb{H}}$. Thus, M and $\textsc{Unorder}(E)$ are subsets of $\tilde{\mathbb{H}}$ and, in fact, we will prove that $M = \textsc{Unorder}(E)$ by assuming towards contradiction that $M \neq \textsc{Unorder}(E)$.

$M \neq \textsc{Unorder}(E)$ if and only if the multiplicity of all elements in M is not equal to the multiplicity of all elements in $\textsc{Unorder}(E)$. Let α be an element in both M and $\textsc{Unorder}(E)$ but where $\textsc{Multiplicity}_M(\alpha) \neq \textsc{Multiplicity}_{\textsc{Unorder}(E)}(\alpha)$. Since M is maximum perfect set of $\textsc{Unorder}(\mathbb{G})$ all of its elements can have a multiplicity of at most p and, because M is a maximum, the multiplicity of α in M is equal to the multiplicity of α in $\textsc{Unorder}(\mathbb{G})$. Since $\textsc{Replicate}(P, p)$ creates

Algorithm 3.5: BREAKPOINTALIQUOTING

Input: An ordered rearranged genome $\mathbb{G}$ with p copies of each gene.
Output: An ordered polyploid $\mathbb{H}$ with p copies of each chromosome.

1 $\tilde{\mathbb{G}} \leftarrow$ UNORDER($\mathbb{G}$)
2 $M \leftarrow$ EXTENDEDMAXIMUMWEIGHTMATCHING($\mathcal{FMWCG}(\tilde{\mathbb{G}})$)
3 $P \leftarrow$ PERFECTSET($\tilde{\mathbb{G}}, M$)
4 $\tilde{\mathbb{H}} \leftarrow$ REPLICATE(P, p)
5 $\mathbb{H} \leftarrow$ REORDER($\tilde{\mathbb{H}}, \mathbb{G}$)
6 **return** $\mathbb{H}$

$\tilde{\mathbb{H}}$ by setting all the elements of P, which includes all of the elements of M as a subset P, to multiplicity p, the multiplicity of α in $\tilde{\mathbb{H}}$ is greater than or equal to that of M. It follows that the multiplicity of α in M is the multiplicity of α in UNORDER($\mathbb{G}$) $\cap\, \tilde{\mathbb{H}}$. From Lines 2 – 6 in Algorithm 3.4 REORDER($\tilde{\mathbb{H}}, \mathbb{G}$) we can conclude that UNORDER($\mathbb{G}$) $\cap\, \tilde{\mathbb{H}}$ = UNORDER($\mathbb{G} \cap \mathbb{H}$). By definition, $E = \mathbb{G} \cap \mathbb{H}$, hence, UNORDER($\mathbb{G} \cap \mathbb{H}$) = UNORDER($E$). Therefore, the multiplicity of α in M and the multiplicity of α in UNORDER(E) are equal; a contradiction. Thus, M = UNORDER(E).

WEIGHT(UNORDER(E')) > WEIGHT(M) follows from the the facts that M = UNORDER(E) and WEIGHT(UNORDER(E')) > WEIGHT(UNORDER(E)). But M is a maximum perfect set; a contradiction. Hence, $\mathbb{H}$ must be minimal. □

Algorithm 3.5 is the complete breakpoint aliquoting algorithm bringing together all the algorithms discussed in this section. It follows from Theorem 3.1.11 and Theorem 3.1.14 that Algorithm 3.5 produces the optimal breakpoint aliquoting.

With the breakpoint aliquoting algorithm complete we will now analyze its complexity. Below we examine each sub-algorithm in the order that they appear in Algorithm 3.5, the sum of which is complexity of our breakpoint aliquoting algorithm.

Given an ordered rearranged genome $\mathbb{G}$ as input into Algorithm 3.5, let $\tilde{\mathbb{G}}$ be its unordered version and let n the number of gene families and let p be the number

of members in each gene family (recall that all gene families are the same size) of $\tilde{\mathbb{G}}$. Then $2pn$ is the number of extremities in $\mathbb{G}$ and $4n$ is the number of vertices in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$. Each vertex in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$ can have at most $p-1$ edges, hence, there are at most $(2p-2)n$ edges in $\mathcal{FMWCG}(\tilde{\mathbb{G}})$.

UNORDER($\mathbb{G}$) runs in linear time in the number of extremities in $\mathbb{G}$, hence it runs in $O(p \cdot n)$ time.

EXTENDEDMAXIMUMWEIGHTMATCHING($\mathcal{FMWCG}(\tilde{\mathbb{G}})$) has two parts: finding the maximum weight matching and adding edges to ensure that every non-null vertex. If m' is the number of edges and n' is the number of vertices in a graph, the fastest algorithm for finding the maximum weight matching of the graph is $O(m' \cdot n' \log n')$ [LP09]. The later step runs in linear time in the number of non-null vertices, which is $2n$, hence the complexity of the algorithm is $O((4p-4)n^2 \log(2n) + 2n)$ time.

PERFECTSET($\tilde{\mathbb{G}}, M$) runs in linear time in the number of edges in the matching returned by extended maximum weight matching. Because we extend the matching so that each non-null vertex is incident with one edge, there is at most one edge in the matching for each non-null vertex. Since there are n non-null vertices the algorithm runs in $O(n)$ time.

REPLICATE(P, p) depends on the implementation of the set. Assuming that increasing the multiplicity of an element is a matter of incrementing an integer, then its running time is linear bounded by the number of elements in the perfect set which is bounded by the number of extremities in $\tilde{\mathbb{G}}$, hence, it runs in $O(n)$ time. If increasing the multiplicity of an element is a matter of adding elements to the set then the complexity is $O((p-1)n)$ since we might have to add $p-1$ copies of an element to the set. For the rest of the analysis we will assume that the running time is $O(n)$.

REORDER($\tilde{\mathbb{H}}, \mathbb{G}$) has two parts: finding the elements of $\mathbb{G}$ that directly correspond to $\tilde{\mathbb{H}}$ and then creating any missing elements in $\tilde{\mathbb{H}}$. The first part involves iterating through $\mathbb{G}$ and finding elements in $\tilde{\mathbb{H}}$. In an efficient set implementation,

finding elements in $\tilde{\mathbb{H}}$ can be done, on average, in constant time. Similarly, tracking the unused subscripts can be done in constant time. Thus, the only significant complexity to the first part comes from iterating through the elements of $\mathbb{G}$, which takes $O(p \cdot n)$ time. The second part requires iterating through the elements of $\tilde{\mathbb{H}}$, which takes $O(n)$ time, and then adding any unused elements, which, in the worse case, takes $O(p-1)$. Thus, the second part runs in $O((p-1)n)$ time. The total algorithm runs in $O(p \cdot n + (p-1)n)$ time.

Putting it all together, we can compute the complexity of Algorithm 3.5: $O(2n + 3p \cdot n + (4p-4)n^2 \log(2n))$ which simplifies to $O(pn^2 \log n)$. In general p is significantly less than n, however, theoretically p is not bounded by n so we cannot ignore it.

3.1.6 Double Cut and Join Distance

Double cut and join distance and breakpoint distance are closely related. BP distance counts the number of different elements between two genomes. For the most part this is the same as DCJ distance; it generally takes one DCJ operation to correct one difference between two genomes. However, sometimes a DCJ operation corrects two differences instead of one and the DCJ distance must take this into account; this is the only difference between the two distances and it is not very significant as this occurs infrequently. The following theorem proves the relationship between the two distance:

Theorem 3.1.15. *Given two genomes* $\mathbb{A}$ *and* $\mathbb{B}$*:*

$$\textsc{Distance}_{DCJ}(\mathbb{A}, \mathbb{B}) \leq \textsc{Distance}_{BP}(\mathbb{A}, \mathbb{B}) \leq 2 \cdot \textsc{Distance}_{DCJ}(\mathbb{A}, \mathbb{B}) \quad (3.1.16)$$

Proof. We will prove the equation in two parts: we prove $\textsc{Distance}_{DCJ}(\mathbb{A}, \mathbb{B}) \leq \textsc{Distance}_{BP}(\mathbb{A}, \mathbb{B})$ first and $\textsc{Distance}_{BP}(\mathbb{A}, \mathbb{B}) \leq 2 \cdot \textsc{Distance}_{DCJ}(\mathbb{A}, \mathbb{B})$ second.

First, for $\textsc{Distance}_{DCJ}(\mathbb{A}, \mathbb{B}) \leq \textsc{Distance}_{BP}(\mathbb{A}, \mathbb{B})$, observe that every adjacency shared between the two genomes causes a cycle of size 2 in the adjacency graph.

It follows that $|\text{ADJACENCIES}(\mathbb{A}, \mathbb{B})|$ is equal to the number of cycles of size 2 in the adjacency graph and, hence, less than or equal to the number of cycles in the adjacency graph. By similar reasoning, $|\text{TELOMERES}(\mathbb{A}, \mathbb{B})|$ is equal to the number of paths of size 1 in the adjacency graph and, hence, less than or equal to the number of paths in the adjacency graph. It follows that $|\text{ADJACENCIES}(\mathbb{A}, \mathbb{B})| + \frac{|\text{TELOMERES}(\mathbb{A},\mathbb{B})|}{2} \leq c + \frac{i}{2}$ where c is the number of cycles and i the number of odd paths in the adjacency graph. Since this factor decreases the distance, it follows that $\text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B}) \leq \text{DISTANCE}_{BP}(\mathbb{A}, \mathbb{B})$.

Second, for $\text{DISTANCE}_{BP}(\mathbb{A}, \mathbb{B}) \leq 2 \cdot \text{DISTANCE}_{DCJ}(\mathbb{A}, \mathbb{B})$, we will assume the most extreme case, where $|\text{ADJACENCIES}(\mathbb{A}, \mathbb{B})| + \frac{|\text{TELOMERES}(\mathbb{A},\mathbb{B})|}{2} = 0$ but the number of cycles and paths in the adjacency graph is otherwise maximal. This produces the worst possible BP distance, $|\text{GENES}(\mathbb{A}, \mathbb{B})|$, but it is otherwise a maximal DCJ distance.

To determine the maximum number of cycles and paths in the adjacency graph, observe that every edge in the adjacency graph corresponds to one extremity shared between the genomes. Since $|\text{ADJACENCIES}(\mathbb{A}, \mathbb{B})| + \frac{|\text{TELOMERES}(\mathbb{A},\mathbb{B})|}{2} = 0$ it follows that there are no cycles of size 2 and odd paths of size 1, so the smallest cycles and odd paths are of size 4 (since all cycles are even) and 3 respectively. Thus, the maximum number of cycles and odd paths is $\frac{|\text{GENES}(\mathbb{A},\mathbb{B})|}{2}$. Thus, in the worst case scenario, BP distance is equal to twice the DCJ distance. □

Because breakpoint distance can be equal to the double cut and join distance, it is possible that the genome that results from Algorithm 3.5 could not only be the optimal breakpoint aliquoting but it could also be the optimal double cut and join aliquoting. Because the breakpoint distance is bounded by twice the double cut and join distance, the distance between the original genome and its optimal breakpoint aliquoting is no more than twice the distance between the original genome and its optimal double cut and join distance. Thus, we can deduce the following:

Algorithm 3.6: SCJSORTING

Input: Two genomes $\mathbb{A}$ and $\mathbb{B}$.
Output: A sequence S of genomes depicting the progression of and joins required to sort $\mathbb{A}$ into $\mathbb{B}$.

```
1  S ← ∅
2  foreach adjacency (α = {x, y}) ∈ 𝔸 do
3      if α ∉ 𝔹 then
4          remove α from 𝔸
5          add {x} and {y} to 𝔸
6      end
7      add 𝔸 to S
8  end
9  foreach adjacency (α = {x, y}) ∈ 𝔹 do
10     if α ∉ 𝔸 then
11         remove {x} and {y} frm 𝔸
12         add α to 𝔸
13     end
14     add 𝔸 to S
15 end
16 return S
```

Theorem 3.1.17. *Algorithm 3.5 is a 2-approximation for the genome aliquoting problem using double cut and join distance.*

3.2 Feijão-Meidanis Algorithm

While Warren and Sankoff did the first genome aliquoting algorithm and provided the only known algorithm for genome aliquoting with double cut and join, it is not the only genome aliquoting algorithm. Recently, in [FaM09], Feijão and Meidanis proposed a new distance metric: *single cut or join* (*SCJ*). Based upon the DCJ realization that all rearrangements are just a series of cuts and joins, *single cut or join* takes this concept to its most elemental level counting each cut and join as one rearrangement operation towards the distance. The result is a shockingly easy sorting algorithm: for single cut or join, as can be seen in Algorithm 3.6, the optimal solution

is to simply perform all the cuts followed by all the joins.

The simplicity of SCJ simplifies some hard, and potentially hard (such as genome aliquoting), problems. In particular, the *median problem* has a simple polynomial time solution under single cut and join:

Definition 3.2.1. *The* median problem *is defined as follows: given n genomes $\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_n$ with the same set of genes and some distance function* DISTANCE_**, construct a genome $\mathbb{H}$, also with the same set of genes such that* $\text{DISTANCE}_*(\mathbb{H}, \mathbb{G}_1) + \text{DISTANCE}_*(\mathbb{H}, \mathbb{G}_2), \ldots, \text{DISTANCE}_*(\mathbb{H}, \mathbb{G}_n)$ *is minimized.*

The *median problem* is NP-hard under most distance metrics[TZS09] and might be the hardest rearrangement problem. As a result, most, if not all, other rearrangement problems can be reduced to instances of the median problem and solved using Algorithm 3.7. This is especially true for genome aliquoting problem which is very closely related to the median problem. Specifically, the genome aliquoting problem can be thought of as an instance of the median problem where the genomes have been mixed together. While this might at first seem to make the problem harder, as we will see, it is no harder and, in fact, the genome aliquoting problem is relaxation of a restriction from the median problem: the restriction that rearrangement operations cannot be performed between genomes. Like most cases where a restriction is relaxed it tends to simplify the problem, though there is no proof that this is the case in this scenario. But at least for SCJ distance it is no harder as [FaM09] solve the genome aliquoting problem by a reduction to the median problem.

To understand the reduction from the genome aliquoting problem to the median problem we must first understand the SCJ algorithm for the median problem. Algorithm 3.7 lists the pseudocode for the SCJ median algorithm. The algorithm is easy to understand once we make the following observation: in SCJ boths cut and joins are separate steps that cost an equal amount, however, cuts must come before joins since many joins are dependent on cuts and not *vice versa*, hence, it is almost always

Algorithm 3.7: SCJMEDIAN

Input: n genomes $\mathbb{G}_1, \mathbb{G}_2, \ldots, \mathbb{G}_n$.
Output: A genome $\mathbb{H}$ such that it minimizes $\text{DISTANCE}_{SCJ}(\mathbb{H}, \mathbb{G}_1) + \text{DISTANCE}_{SCJ}(\mathbb{H}, \mathbb{G}_2), \ldots, \text{DISTANCE}_{SCJ}(\mathbb{H}, \mathbb{G}_n)$.

```
1  H ← ∅
2  foreach adjacency α ∈ ⋃_{i←1}^{n} G_i do
3      if n − 2 · Σ_{i←1}^{n} |α ∩ G_i| < 0 then
4          add α to H
5      end
6  end
7  foreach extremity x ∈ (EXTREMITIES(G) \ EXTREMITIES(H)) do
8      add {x} to H
9  end
10 return H
```

better to do only cuts. Thus, when constructing a genome to minimize the distance in most circumstances it should just be telomeres as joins are too expensive. The only time that joins are cheaper than cuts is when more than half of the genomes share the same adjacency. Thus, in the firt loop of Algorithm 3.7 it checks for adjacencies shared between more than half of the genomes and adds them to the median genome. After that, in the second loop, for any extremities not covered, it adds them as telomeres to the median genome.

Algorithm 3.8 is the Feijão-Meidanis algorithm for genome aliquoting. The algorithm looks for adjacencies that appear multiple times and break them up over multiple genomes. If the number of occurrences of that adjacency is more than half the ploidy it should be in the aliquoting, thus, in Algorithm 3.8 it places repeat occurrences of the same adjacency over multiple genomes so that the median algorithm will select adjacencies that should be in the aliquoting. Note that the genomes created can be duplicated genomes themselves, it doesn't really matter so long as the number of occurrences in the input genome corresponds to the number of created genomes in which the adjacency appears.

Algorithm 3.8: SCJALIQUOTING

Input: A rearranged duplicated genome $\mathbb{G}$ with p copies of every gene.
Output: A polyploid $\mathbb{H}$ with a ploidy of p such that it minimizes $\text{DISTANCE}_{SCJ}(\mathbb{G}, \mathbb{H})$.

```
1  F_1, F_2, ..., F_p ← ∅
2  G̃ ← UNORDER(G)
3  foreach adjacency α ∈ G̃ do
4      for i ← 1 to MULTIPLICITY(α) do
5          add α to F_i
6      end
7  end
8  foreach genome F_i do
9      foreach extremity x ∈ (EXTREMITIES(G̃) \ EXTREMITIES(F_i)) do
10         add {x} to F_i
11     end
12 end
13 H' ← SCJMEDIAN(F_1, F_2, ..., F_p)
14 H̃ ← REPLICATE(H', p)
15 H ← REORDER(H̃, G)
16 return H
```

3.2.1 Feijão, Meidanis and DCJ

SCJ and DCJ are obviously very closely related. What is more, it is easy to convert between a sequence of SCJ operations and DCJ operations: just combine related cuts and joins. But this doesn't mean that, if we had an algorithm that converts SCJ operations into DCJ operations, we could use the Algorithm 3.8 to find the optimal genome aliquoting for DCJ.

The relationship between SCJ and DCJ, discussed in [FaM09], is obviously that one DCJ operation is at most 4 SCJ operations, specifically, two cuts and two joins. From this observation alone, we know that Algorithm 3.8 can't guarantee an optimal aliquoting, though it can be a 4-approximation algorithm. It is worth mentioning that it is possible, though unlikely, that the DCJ distance will be equal to the SCJ distance so Algorithm 3.8 is a 4-approximation although it will rarely give the optimal

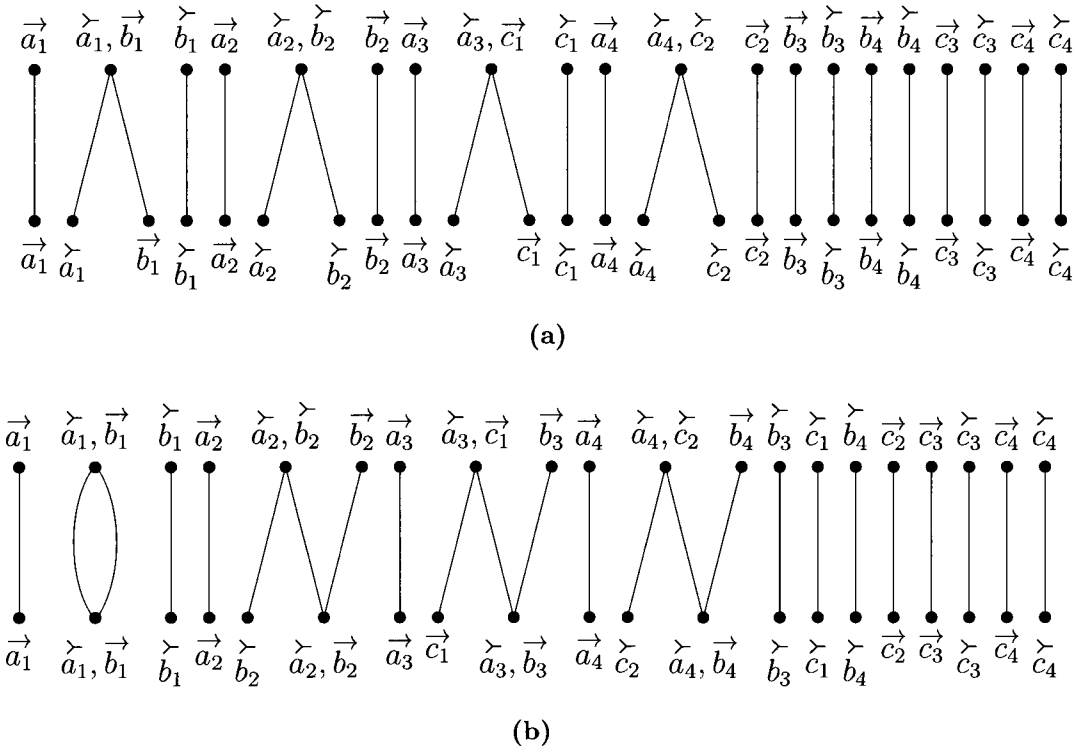


Figure 3.6: (a) An adjacency graph depicting an optimal SCJ aliquoting for a genome. (b) An adjacency graph depicting another aliquoting for the same genome that has a better DCJ distance than the optimal SCJ aliquoting but a worse SCJ distance.

answer.

The reason why Algorithm 3.8 will rarely produce an optimal solution for DCJ distance is because of its strong bias against joins. The reality is that most of the aliquoted genomes produced by Algorithm 3.8 consist solely of telomeres.

Not only is an aliquoted genome consisting solely of telomeres biologically unrealistic but in the case where an extremity is never a telomere in the original genome, for DCJ, it is more optimal to have that extremity as part of an adjacency rather then as part of a telomere, *e.g.* the genome $\mathbb{G} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}, \vec{b_1}\}, \{\overset{\succ}{b_1}\}, \{\vec{a_2}\}, \{\overset{\succ}{a_2}, \overset{\succ}{b_2}\}, \{\vec{b_2}\}, \{\vec{a_3}\}, \{\overset{\succ}{a_3}, \vec{c_1}\}, \{\overset{\succ}{c_1}\}, \{\vec{a_4}\}, \{\overset{\succ}{a_4}, \overset{\succ}{c_2}\}, \{\vec{c_2}\}, \{\vec{b_3}\}, \{\overset{\succ}{b_3}\}, \{\vec{b_4}\}, \{\overset{\succ}{b_4}\}, \{\vec{c_3}\}, \{\overset{\succ}{c_3}\}, \{\vec{c_4}\}, \{\overset{\succ}{c_4}\}\}$ is optimally aliquoted with SCJ into the genome $\mathbb{H} = \{\{\vec{a_1}\}, \{\overset{\succ}{a_1}\}, \{\vec{a_2}\}, \{\overset{\succ}{a_2}\}, \{\vec{a_3}\}, \{\overset{\succ}{a_3}\}, \{\vec{a_4}\}, \{\overset{\succ}{a_4}\}, \{\vec{b_1}\}, \{\overset{\succ}{b_1}\}, \{\vec{b_2}\}, \{\overset{\succ}{b_2}\}, \{\vec{b_3}\}, \{\overset{\succ}{b_3}\}, \{\vec{b_4}\}, \{\overset{\succ}{b_4}\}, \{\vec{c_1}\}, \{\overset{\succ}{c_1}\}, \{\vec{c_2}\},$

$\{\overleftarrow{c_2}\}$, $\{\overrightarrow{c_3}\}$, $\{\overleftarrow{c_3}\}$, $\{\overrightarrow{c_4}\}$, $\{\overleftarrow{c_4}\}\}$ that consists only of telomeres. The reason for this is that, if we choose one of the adjacencies in the original genome involving that extremity and add the necessary number of copies of it to the aliquoted genome, then the adjacency graph between the original genome and this aliquoted genome will contain at least one cycle (of size 2), *e.g.* under DCJ, the genome from the previous example would be better aliquoted into $\mathbb{H} = \{\{\overrightarrow{a_1}\}, \{\overleftarrow{a_1}, \overrightarrow{b_1}\}, \{\overleftarrow{b_1}\}, \{\overrightarrow{a_2}\}, \{\overleftarrow{a_2}, \overrightarrow{b_2}\}, \{\overleftarrow{b_2}\}, \{\overrightarrow{a_3}\}, \{\overleftarrow{a_3}, \overrightarrow{b_3}\}, \{\overleftarrow{b_3}\}, \{\overrightarrow{a_4}\}, \{\overleftarrow{a_4}, \overrightarrow{b_4}\}, \{\overleftarrow{b_4}\}, \{\overrightarrow{c_1}\}, \{\overleftarrow{c_1}\}, \{\overrightarrow{c_2}\}, \{\overleftarrow{c_2}\}, \{\overrightarrow{c_3}\}, \{\overleftarrow{c_3}\}, \{\overrightarrow{c_4}\}, \{\overleftarrow{c_4}\}\}$ that preserves on adjacency, specifically $\{\overleftarrow{a_1}, \overrightarrow{b_1}\}$, which gives the adjacency graph, Figure 3.6b, an cycle of size 2. Whereas, if the aliquoted genome uses only telomeres then the adjacency graph will, for any extremities that only make up adjacencies in the original genome, contain even cycles instead, *e.g.* the adjacency graph of an optimal (for SCJ) aliquoting, Figure 3.6a, depicting the adjacency graph of an optimal SCJ aliquoting, only contains even paths using extremity $\overleftarrow{a}$, which exclusively makes up adjacencies in the original genome. From Equation 1.3.6 it is clear that the aliquoted genome with a cycle is usually better than one with only even paths, *e.g.* the DCJ distance of Figure 3.6b with the adjacencies in the aliquoting is 7 $(16 - 1 - \frac{16}{2})$ which is better than the DCJ distance of Figure 3.6a, 8 $(16 - 0 - \frac{16}{2})$, despite Figure 3.6a being the optimal SCJ aliquoting.

Without extensive modifications, Algorithm 3.8 cannot overcome its weakness of avoiding adjacencies to get a closer bound for DCJ. There are invariably a few cases where SCJ avoids an adjacency when there is an equivalent SCJ distance with the adjacency. However, there are still many cases where this is not possible and where the optimal SCJ distance is not the optimal DCJ distance, like the genome depicted in Figure 3.6a. That said, there might be some post-processing algorithms that can be used to increase the number of adjacencies after the SCJ aliquoting but it is open problem wether or not such an algorithm can be used to find the optimal DCJ.

3.3 Summary

With some modifications, the same strategies that work for genome halving also work for genome aliquoting but, it seems, they only work for BP distance, which is one of the simplest distance metrics.

At least for one distance metric, SCJ distance, an algorithm that solves instances of the median problem can also be used to solve instances of the genome aliquoting problem. Unfortunately, for most distances the median problem is NP-hard for many distances, including BP distance (except in the case where the input genomes can be contain mixed circular and linear chromosomes)[TZS09]. However, for SCJ distance, a distance metric that is even simpler and more elementary than BP distance, the median problem is in P.

Therefore, there are two very different strategies for solving the aliquoting problem both of which are approximation algorithms for the genome aliquoting problem with DCJ distance. Genome aliquoting with BP distance has a tighter bound but genome aliquoting with SCJ distance is much faster and still has a constant bound.

Chapter 4

Conclusions and Future Work

To compare genomes that contain multiple copies of genes is difficult. Directly comparing them in polynomial time seems to be impossible. The first efficient algorithm was the El-Mabrouk-Sankoff genome halving algorithm. While it does not directly compare two ambiguous genomes, it eliminates the duplicates allowing two ambiguous genomes to be indirectly compared via the resulting unambiguous genomes.

The El-Mabrouk-Sankoff algorithm was difficult to understand spanning a series of algorithms attempting to simplify it. But, as we have hopefully shown in Chapter 2, when broken down, all the algorithms are essentially the same. Also, using DCJ distance instead of HP distance simplifies the problem as HP distance is more complex requiring the algorithm to account for additional factors.

Generalizing the El-Mabrouk-Sankoff algorithm into a universal solution to duplicates is not easy. In fact, just solving the generalized genome halving problem, the genome aliquoting problem, is challenging. Two efficient algorithm that solve genome aliquoting problem exist although both are for less interesting distances than DCJ distance. The Warren-Sankoff genome aliquoting algorithm, a generalization of the El-Mabrouk-Sankoff algorithm, can aliquote any polyploid in sub-cubic time for BP distance. The Feijão-Meidanis algorithm completely departs from the El-Mabrouk-Sankoff algorithm and can aliquote any polyploid in linear time for SCJ distance. Both BP distance and SCJ distance are closely related to DCJ distance. As a result,

their aliquoting algorithms serve as a 2-approximation and a 4-approximation for the genome aliquoting with DCJ problem respectively.

There are two obvious avenues for future work. The first is to find an exact algorithm for the genome aliquoting problem under DCJ distance, if such an algorithm even exists. We discuss the development of such an algorithm in Section 4.1 with a particular emphasis on what it would take to modify the Warren-Sankoff algorithm to be an exact DCJ distance genome aliquoting algorithm.

The second area for future research would be to generalize either the genome halving algorithms or the genome aliquoting algorithms to solve other types of duplication events, such as tandem duplications or segment duplications. Such a generalization has actually been attempted before by Nadia El-Mabrouk as an extension of her work on genome halving. We summarize her results in Section 4.2 as well as discuss what more can be done in this area especially in light of the new work into the genome aliquoting problem.

4.1 Genome Aliquoting with Double Cut and Join

There is a hidden part to the Warren-Sankoff genome halving algorithm that, because it is not obvious and not needed for breakpoint distance, we didn't generalize when we developed the Warren-Sankoff genome aliquoting algorithm. It goes back to the difference between breakpoint distance and double cut and join distance: breakpoint distance counts identical elements between the genomes but double cut and join distance not only counts identical elements between the genomes but also counts how many elements become identical when transforming one genome into the other. In many situations there is no difference as often it takes one double cut and join operation per difference to transform one genome into the other. But sometimes one double cut and join operation corrects two differences rather than one. Predicting these situations, and more importantly, maximizing them it the missing step between

finding the optimal aliquoting under breakpoint distance and the optimal aliquoting under double cut and join distance.

Determining whether or not a double and join operation corrects one or two differences between the genomes involves finding the cycles and paths in the adjacency graph between the genomes, this is basis for double cut and join distance formula (Equation 1.3.6) with even length cycles and odd length paths correcting exactly one extra difference. From Observation 2.1.18 the intersection graph of a duplicated genome with exactly two copies of every gene is an adjacency graph on which DCJ operations can be applied to transform half the genome to be identical with the other half. As a result, even length cycles and odd length paths in the intersection graph identify situations where DCJ operations correct two differences instead of just one.

Since there were only two copies of every gene in the genome halving problem, the benefit provided by the cycles and paths of the intersection graph can be implicitly determined, which is why none of the genome halving algorithms explicitly check for them. In the genome aliquoting problem, to get the best possible DCJ distance we must explicitly find the maximum number of even cycles and odd paths and modify the final genome accordingly.

Finding the cycles and paths in the intersection graph in which there are more than two copies of every gene is certainly a non-trivial problem. It may even be an NP-hard problem. We can offer up no efficient algorithm to solve this problem although we can offer three insights into the problem.

First, we observe that these cycles and paths must be *alternating* between vertices that correspond to matched edges and vertices that correspond to unmatched edges (we will call these vertices *matched* and *unmatched* respectively). Because of this, our cycles and paths can never contain an edge between two matched vertices or two unmatched vertices. Thus, we can remove such edges from intersection graph without effecting result. The graph that results will be bipartite, although unfortunately it does not have a degree of 2 and, hence, it is not an adjacency graph.

Second, cycles contain only vertices that correspond to adjacencies; only paths contain vertices that correspond to telomeres, of which they contain exactly two. While this was also true for the genome halving problem this observation is particularly important for the genome aliquoting problem because with all the extra edges it is theoretically possible to have a cycle that contains vertices that correspond to telomeres or paths with telomeres in the middle. Thus, for the genome aliquoting problem such cycles and paths must be discarded.

Third, we are seeking edge disjoint cycles and paths. Unfortunately, the cycles and paths must also be vertex disjoint but only with regards to unmatched vertices; matched vertices may be reused as often as desired. We can imagine the matched vertices being templates from which we will construct our final genome using the extremities of the unmatched vertices. Since the matched vertices are templates we can reuse the same template over several times. On the other hand, the unmatched vertices can only be used once. Additionally, we cannot just choose any edge for the cycles and paths. Each edge has an associated unordered extremity. No path or cycle can contain edges labelled with the same unordered extremity.

Once all the cycles and paths corresponding to the above criteria have been found we can use that information to partially reconstruct the aliquoted genome. Recall that Algorithm 3.4 REORDER was divided into two parts. In the first part it finds, for each matched vertex, a similar element in the original ordered genome; each matched vertex should correspond to an element of the original ordered genome. The second part arbitrarily orders the remainder of the unordered genome to create an aliquoted genome. It is in the second part where the errors creep in that prevent the BP aliquoting algorithm from being an exact DCJ aliquoting algorithm. To optimize the result for DCJ we must use the cycles and paths to reorder the unordered genome. Only if the cycles and paths fail to completely reorder the genome do we resort to arbitrarily ordering the genome.

In the intersection graph, each edge corresponds to an unordered extremity and

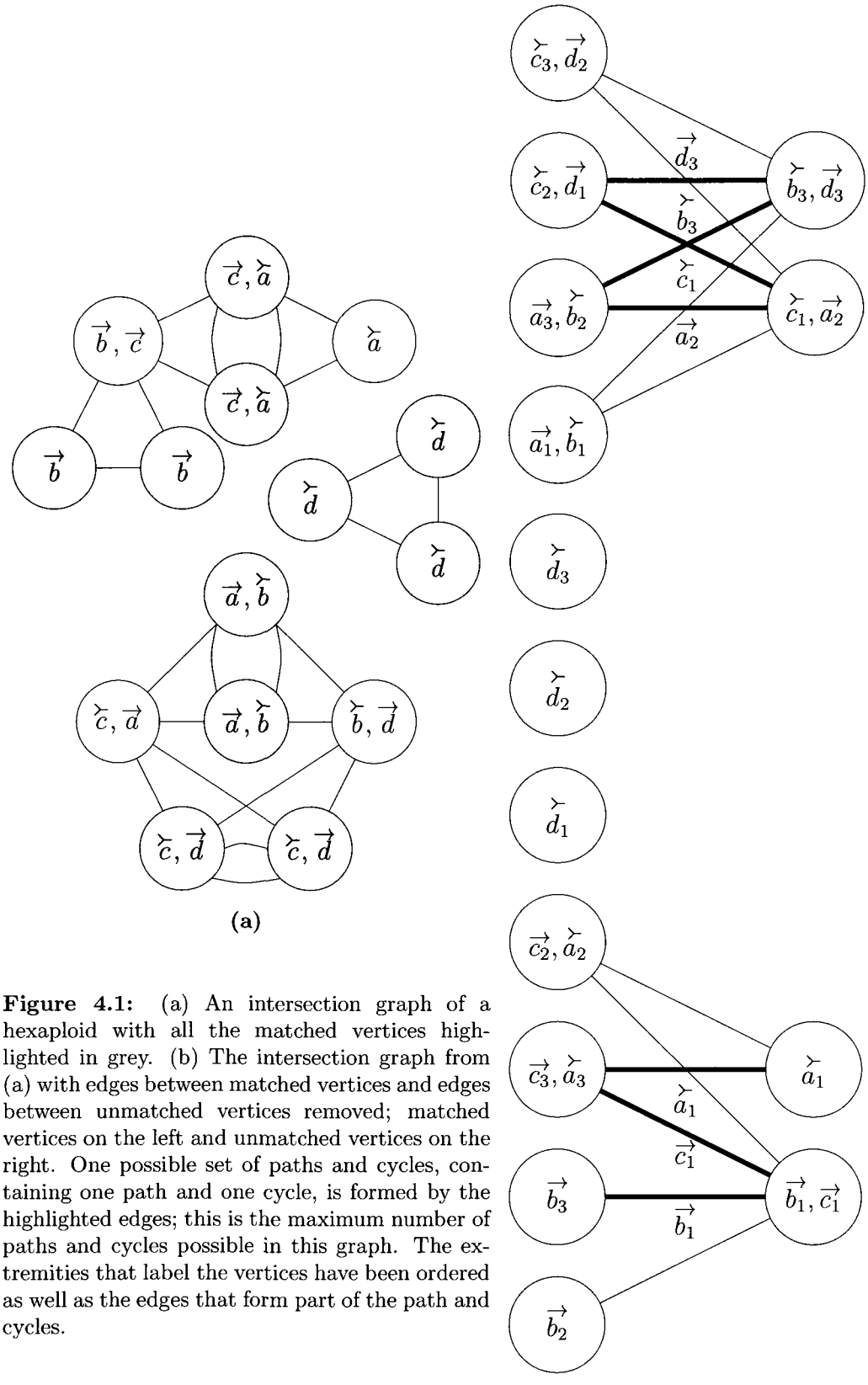


Figure 4.1: (a) An intersection graph of a hexaploid with all the matched vertices highlighted in grey. (b) The intersection graph from (a) with edges between matched vertices and edges between unmatched vertices removed; matched vertices on the left and unmatched vertices on the right. One possible set of paths and cycles, containing one path and one cycle, is formed by the highlighted edges; this is the maximum number of paths and cycles possible in this graph. The extremities that label the vertices have been ordered as well as the edges that form part of the path and cycles.

each vertex corresponds to an unordered element in the rearranged genome. Using the same method that Algorithm 3.4 finds reorderings for the matched vertices, we can find orderings for the unmatched vertices of the intersection graph too; it is possible to do this such that every vertex in the intersection graph corresponds to a unique ordered element of the rearranged genome. Ordering the extremity assigned to each edge is unfortunately more complex.

For each edge, the associated extremity corresponds to one of the extremities in the intersection of the unordered elements of the rearranged genome the correspond to the vertices to which the edge is incident. Unfortunately, when the element that correspond to the vertices are ordered their intersection is the empty set since the extremities can now be distinguished with different subscripts. Thus, assigning an ordered extremity to an edge that reflects the relationship between the two incident vertices associated ordered element is impossible. However, it is possible to assign an ordered extremity to an edge that will help us infer an ordering from the cycles and paths.

We can ignore edges between pairs of matched vertices and pairs of unmatched vertices as these are never part of a cycle or path. We have no need to perform any kind of operation on the matched vertices since Algorithm 3.4 already optimally reorders the element that correspond to these vertices. Thus, for each edge, if x is the unordered extremity that corresponds to the edge we assign the edge instead x_i where x_i is an element of the unmatched vertex to which the edge is incident.

Once we have an ordered extremity for each (relevant) edge, using the cycles and paths to improve the reordering of the genome is easy. For each matched vertex along the cycle or path, join the ordered extremities on the incident edges form the cycle or path and add the result to the ordered genome, *e.g.* in a path, if a matched vertex is incident with two edges that are also in the path with corresponding ordered extremities x_i and y_j respectively, add $\{x_i, y_j\}$ to the ordered genome. For paths, the start and end vertex will be incident with only one edge. However, recall that, by

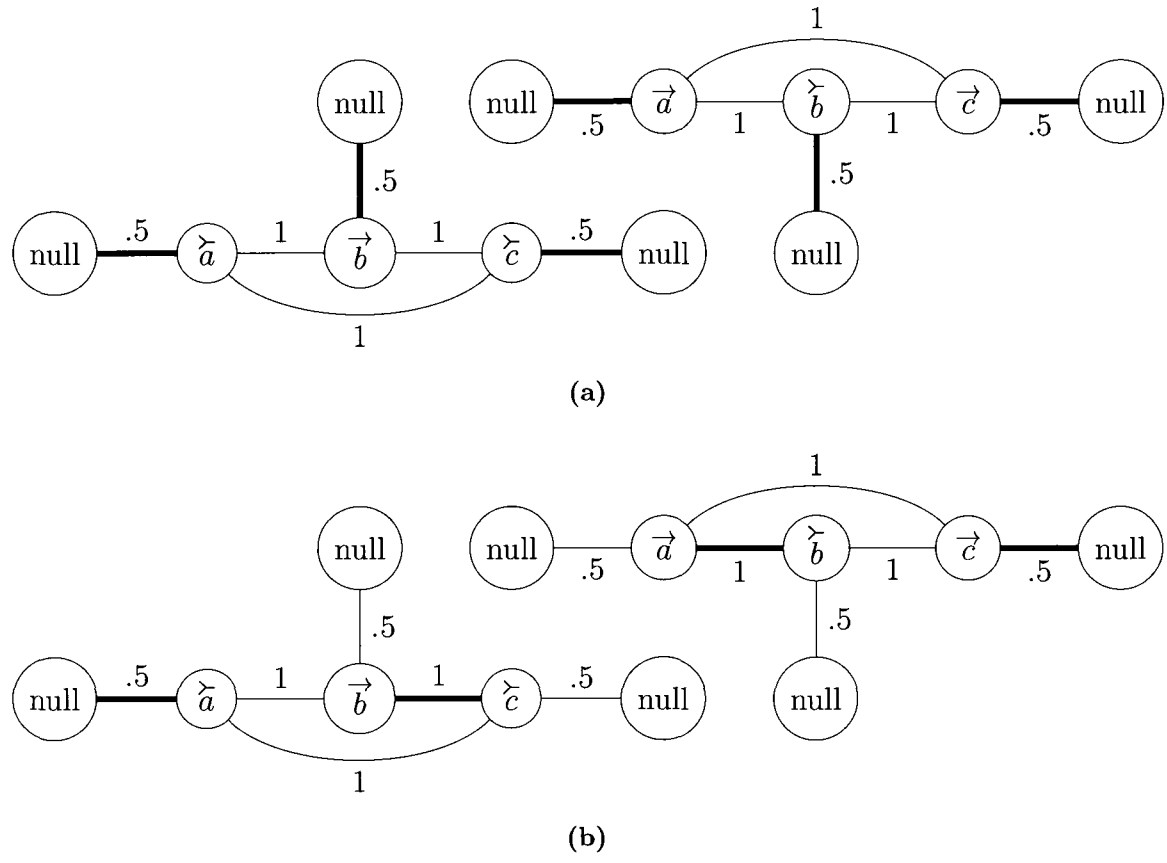


Figure 4.2: A pair of fully modified weighted clique graphs describing the same genome. Highlighted edges are edges that belong to the maximum weight matching; each graph describes a different maximum weight matching. Despite both being maximum weight matchings, the result genomes, even after optimizing with cycles and paths, produce genomes with completely different DCJ distances from the original.

definition, the start and end vertex must correspond to a telomere. Thus, in such a case we simply add the ordered extremity corresponding to the one incident edge to the ordered genome as a telomere, *e.g.* if a matched start vertex of a path is incident with an edge that corresponds to x_i then we add $\{x_i\}$ to the ordered genome.

Alas, even if it were possible to find an optimal selection of cycles and paths it is not possible to guarantee that the result is optimal. Consider the following counter-example: let $\mathbb{G}$ be the hexaploid $\{\{\overleftarrow{a_1}\}, \{\overrightarrow{a_1}, \overleftarrow{b_1}\}, \{\overrightarrow{b_1}\}, \{\overrightarrow{a_2}\}, \{\overleftarrow{a_2}, \overrightarrow{b_2}\}, \{\overleftarrow{b_2}, \overrightarrow{c_1}\}, \{\overleftarrow{c_1}\}, \{\overleftarrow{b_3}\}, \{\overrightarrow{b_3}, \overleftarrow{c_2}\}, \{\overrightarrow{c_2}, \overrightarrow{a_3}\}, \{\overleftarrow{a_3}, \overleftarrow{c_3}\}, \{\overrightarrow{c_3}\}\}$. The maximum weighted matching has a

weight of 3, and there are 16 ways of forming it, let us consider two such ways. Figure 4.2a depicts one maximum weight matching and Figure 4.2b depicts another. After performing a Warren-Sankoff genome aliquoting but accounting for cycles and paths the matching depicted in Figure 4.2a produces $\mathbb{H} = \{\{\overleftarrow{a_1}\}, \{\overrightarrow{a_1}\}, \{\overleftarrow{b_1}\}, \{\overrightarrow{b_1}\}, \{\overrightarrow{a_2}\}, \{\overleftarrow{a_2}\}, \{\overrightarrow{b_2}\}, \{\overleftarrow{b_2}\}, \{\overrightarrow{c_1}\}, \{\overleftarrow{c_1}\}, \{\overleftarrow{b_3}\}, \{\overrightarrow{b_3}\}, \{\overleftarrow{c_2}\}, \{\overrightarrow{c_2}\}, \{\overrightarrow{a_3}\}, \{\overleftarrow{a_3}\}, \{\overleftarrow{c_3}\}, \{\overrightarrow{c_3}\}\}$. Similarly, the matching depicted in Figure 4.2b produces $\mathbb{H}' = \{\{\overleftarrow{a_1}\}, \{\overrightarrow{a_1}, \overleftarrow{b_1}\}, \{\overrightarrow{b_1}, \overleftarrow{c_3}\}, \{\overrightarrow{c_3}\}, \{\overleftarrow{a_2}\}, \{\overrightarrow{a_2}, \overleftarrow{b_2}\}, \{\overrightarrow{b_2}, \overleftarrow{c_1}\}, \{\overrightarrow{c_1}\}, \{\overleftarrow{a_3}\}, \{\overrightarrow{a_3}, \overleftarrow{b_3}\}, \{\overrightarrow{b_3}, \overleftarrow{c_2}\}, \{\overrightarrow{c_2}\}\}$. While both have the same BP distance, 6, they have very different DCJ distances. The DCJ distance of the former is 6 while that of the latter is 4. The difference is that, while neither have any cycles in their intersection graphs, the later has 4 odd paths whereas the former only has even paths. Thus, different matches do change the available number of cycles and paths.

Since it is not possible to find the cycles and paths without first having a matching, to get the Warren-Sankoff algorithm to give the optimal DCJ distance for all possible maximum weight matchings we would have to check for an optimal selection of cycles and paths. The matching/cycle and path combination that produces the best overall result would most likely produce the optimal DCJ distance, although this remains to be proven. But clearly, this approach isn't practical; even if there was an efficient way to find the cycles and paths there certainly isn't a polynomial time algorithm that enumerates all possible maximum weight matchings simply because the number of matchings cannot be guaranteed to be a polynomial of the size of the input.

There may, however, be a polynomial time solution to the genome aliquoting problem with DCJ distance. Such an algorithm is unlikely to resemble the Warren-Sankoff algorithm, or for that matter, any of the genome halving algorithms. It isn't likely to resemble the Feijão-Meidanis algorithm either since that algorithm depends on a polynomial time solution to the median problem which doesn't exist under DCJ distance[TZS09].

4.2 General Duplications

In Section 1.3.3 we mentioned that the genome halving problem and genome aliquoting problem are just specific cases of the rearrangement-duplicate problem introduced by [EM00]. In that paper, El-Mabrouk used a variation of her genome halving algorithm to solve this problem; we look at this variation in Section 4.2.1. The question is: does this algorithm generalize from HP distance to DCJ distance? Similarly, El-Mabrouk's algorithm has the same limit of genome halving: gene families of size 2. Can we generalize this to gene families of any size?

4.2.1 El-Mabrouk Algorithm

Consider a scenario where only individual genes can be duplicated and when they are duplicated they can be inserted to any position in the genome (not just in adjacent positions like a tandem duplication). With this constraint solving the rearrangement-duplication problem is trivial: simply remove all but one copy of each gene. Under the restriction that only individual genes can be duplicated, this algorithm optimally transforms an ambiguous genome into an unambiguous one using a number of duplications equal to the number of genes removed and no rearrangement operations.

Unfortunately, real duplications often duplicate contiguous segments of genes rather than just one gene and real duplications rarely place the duplicated segment at any position in the genome. Either of these conditions complicates the problem so for her instance of the rearrangement-duplication problem El-Mabrouk relaxed the second condition: she allows a single duplication event to duplicate any contiguous segment and insert it anywhere in the genome. Such duplication events are called *duplication-transpositions* and, while they might occur in biology, they probably don't occur with the same frequency as the El-Mabrouk algorithm suggests. Nevertheless, the El-Mabrouk algorithm represents real progress towards an algorithm for the rearrangement-duplication problem that covers all duplication operations.

In addition to duplication-transpositions, for a rearrangement operation, the El-Mabrouk algorithm uses reversals. Because of this, the El-Mabrouk algorithm is restricted to unichromosomal genomes. El-Mabrouk further restricts the algorithm to circular unichromosomal genomes since it avoids capping and is more biologically realistic.

To avoid the complications introduced by reversals in our presentation of the algorithm we consider both reversals and block-interchanges. However, this does not require modifications to the algorithm since El-Mabrouk's algorithm, like the El-Mabrouk-Sankoff genome halving algorithm, is broken into two steps: one where it finds the solution for reversals and block-interchanges and a second step that removes the block-interchanges. We simply ignore the second step. However, the algorithm we present is still restricted to circular unichromosomal genomes. We discuss how to introduce translocations and handle mixed circular and linear multichromosomal genomes in Section 4.2.3.

The final restriction imposed by the El-Mabrouk algorithm is that it can only analyze *semi-ambiguous* genomes. A *semi-ambiguous* genome is an ambiguous genome where gene families are of size 1 or 2.

The strategy of the El-Mabrouk algorithm is based around the observation that the duplications and rearrangements can be performed as separate steps. Specifically, all the rearrangements can be performed first and then the duplications. Thus, exactly like the genome halving problem the algorithm constructs an intermediary genome that minimizes the duplication operations.

To understand the structure of the intermediary genome we must first understand the duplication-transpositions. A *repeat* is a maximum contiguous sequence of genes in an unordered ambiguous genome $\tilde{\mathbb{G}}$ that appears more than once, *e.g.* the genome $\tilde{\mathbb{G}} = \{\{\overrightarrow{y}, \overleftarrow{a}\}, \{\overrightarrow{a}, \overrightarrow{b}\}, \{\overleftarrow{b}, \overleftarrow{c}\}, \{\overrightarrow{c}, \overleftarrow{x}\}, \{\overrightarrow{x}, \overleftarrow{d}\}, \{\overrightarrow{d}, \overrightarrow{e}\}, \{\overleftarrow{e}, \overleftarrow{e}\}, \{\overrightarrow{e}, \overrightarrow{d}\}, \{\overleftarrow{d}, \overleftarrow{a}\}, \{\overrightarrow{a}, \overrightarrow{b}\}, \{\overleftarrow{b}, \overleftarrow{c}\}, \{\overrightarrow{c}, \overleftarrow{y}\}\}$, depicted in Figure 4.3, has two repeats $\{\{\overleftarrow{a}\}, \{\overrightarrow{a}, \overrightarrow{b}\}, \{\overleftarrow{b}, \overleftarrow{c}\}, \{\overrightarrow{c}\}\}$ and $\{\{\overleftarrow{e}\}, \{\overrightarrow{e}, \overrightarrow{d}\}, \{\overleftarrow{d}\}\}$. The orientation of the sequence doesn't mat-

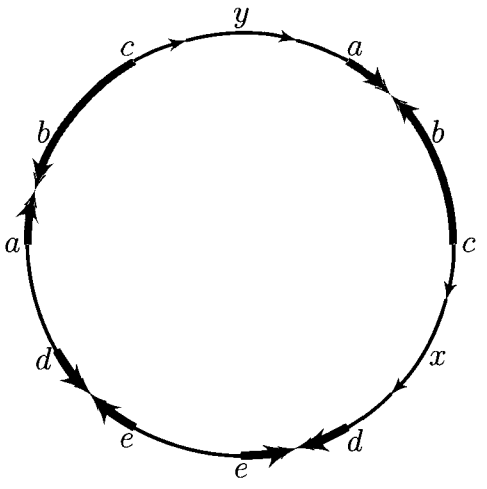


Figure 4.3: A genome with two repeats: one in bold black and another in bold gray.

ter, if a sequence and its reverse both appear in the genome then they are both repeats of each other, *e.g.* as can be seen in Figure 4.3, the sequence $\{\{\overleftarrow{e}\}, \{\overrightarrow{e}, \overrightarrow{d}\}, \{\overleftarrow{d}\}\}$ is oriented differently in both its repeats.

Repeats correspond to duplication-transposition operations. Once repeats are identified, transforming an ambiguous genome to an unambiguous genome is easy: just arbitrarily remove all but one instance of each repeat from the genome. According to [EM00], like when we restricted duplications to individual genes, this part is optimal. However, according to [Kah07], [EM00] is incorrect; the problem of removing duplicated segments is much more complex than removing individual genes. However, for now we will assume that [EM00] is correct. We will save the observations of [Kah07] and possible solutions until Section 4.2.2.

Since the El-Mabrouk algorithm is concerned with semi-ambiguous genomes, a repeat corresponds to exactly two segments in the genome, hence, each repeat corresponds to exactly one duplication. Therefore, the duplication distance is equal to the number of repeats. It follows that, to minimize the duplication distance, we must minimize the number of repeats.

The maximum number of repeats in a semi-ambiguous genome corresponds to

the number of gene families of size 2 as, in the worst case scenario, there has been one duplication event for each duplicated gene. It follows that minimizing the number of repeats is a matter of maximizing the size of each repeat.

Using rearrangement operations to maximize the size of the repeats is very similar to the genome halving problem. So similar, in fact, that the same algorithm can be used with just a few modifications. The key differences between this problem and the genome halving problem is that we needn't always perform a rearrangement operation as the genome doesn't need to be any particular state to remove the duplications and that the cost of the duplication operations is a variable. Hence, we must now consider if it is worth performing each rearrangement operation as it may be cheaper to just perform multiple duplications instead.

The intersection graph of the genome $\mathbb{G} = \{\{\overrightarrow{y}, \overrightarrow{a_1}\}, \{\overrightarrow{a_1}, \overrightarrow{b_1}\}, \{\overrightarrow{b_1}, \overrightarrow{x}\}, \{\overrightarrow{x}, \overrightarrow{h_1}\}, \{\overrightarrow{h_1}, \overrightarrow{f_1}\}, \{\overrightarrow{f_1}, \overrightarrow{e_1}\}, \{\overrightarrow{e_1}, \overrightarrow{g_1}\}, \{\overrightarrow{g_1}, \overrightarrow{c_1}\}, \{\overrightarrow{c_1}, \overrightarrow{a_2}\}, \{\overrightarrow{a_2}, \overrightarrow{b_2}\}. \{\overrightarrow{b_2}, \overrightarrow{d_1}\}, \{\overrightarrow{d_1}, \overrightarrow{h_2}\}, \{\overrightarrow{h_2}, \overrightarrow{z}\}, \{\overrightarrow{z}, \overrightarrow{d_2}\}, \{\overrightarrow{d_2}, \overrightarrow{e_2}\}, \{\overrightarrow{e_2}, \overrightarrow{g_2}\}, \{\overrightarrow{g_2}, \overrightarrow{c_2}\}, \{\overrightarrow{c_2}, \overrightarrow{f_2}\}, \{\overrightarrow{f_2}, \overrightarrow{y}\}\}$ is depicted in Figure 4.4a.

Like the El-Mabrouk-Sankoff genome halving algorithm, the El-Mabrouk algorithm uses a natural graph rather than an intersection graph, Figure 4.4b depicts the corresponding natural graph. Since the El-Mabrouk algorithm considers only circular unichromosomal genomes there are no telomeres. Despite this, the intersection graph still has both paths and cycles with the *singleton* genes, genes with gene families of size 1, serving as the endpoints of paths.

Unlike the genome halving problem, finding the maximum independent set of the intersection graph (or its corresponding set on the natural graph) is not the goal of the El-Mabrouk algorithm. There are two reasons for this: first, in most cases the maximum independent set causes too many rearrangement operations and second, duplicating a maximum independent set that includes singletons causes nonsensical results. However, some non-maximal independent sets do not cause these two problems, hence, the El-Mabrouk algorithm finds these independent sets.

Avoiding the second problem is simple: there are no acceptable independent sets

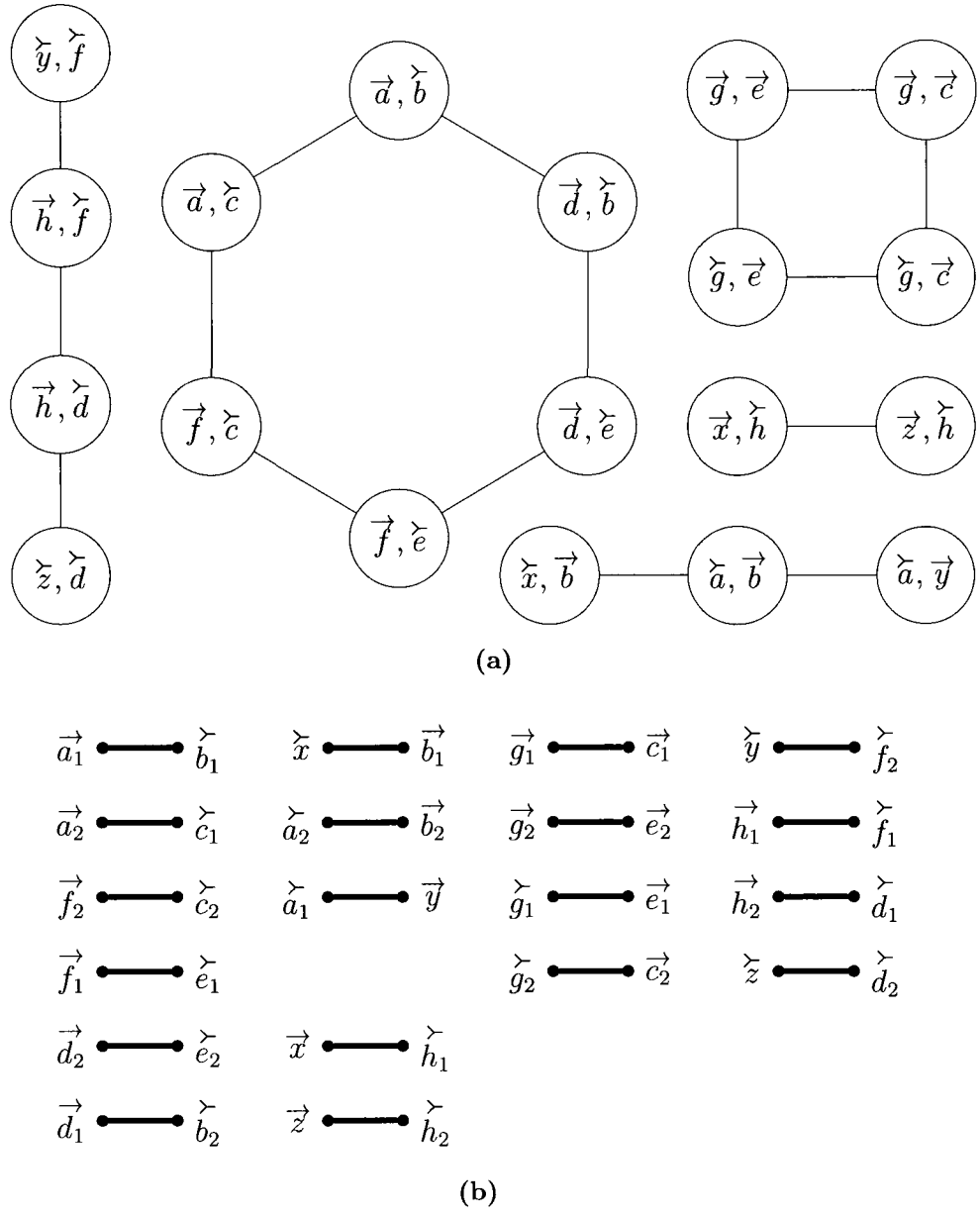


Figure 4.4: (a) The intersection graph of a semi-ambiguous genome. (b) The natural graph that corresponds to the intersection graph.

of paths (and all singletons in a path). While at first this solution seems extreme, there might be an acceptable independent set of the component that does not include the vertices labeled with singletons, but as it turns out there is no independent set that avoids the singletons and avoids the problem of causing too many rearrangements.

Algorithm 4.1 REARRANGEMENTDUPLICATION

Input Ambiguous genome $\mathbb{G}$
Output Set A that is perfect and covers UNORDER($\mathbb{G}$)

```
1  H ← ∅
2  NG ← NATURALGRAPH(G)
3  foreach sequence N_i ∈ NG do
4      {x, y} ← N_i[1]
5      if |N_i| is even and neither x nor y are singletons then
6          for j ← 1 to |N_i|/2 do
7              {x_k, y_l} ← N_i[2 j]
8              add N_i[2 · j] and {x_{3-k}, y_{3-l}} to H
9          end
10     else
11         add each element of N_i to H
12     end
13 end
14 return H
```

Beyond paths, odd cycles also don't have any acceptable independent sets, all independent sets cause too many rearrangements Thus, only components that have acceptable independent sets are even cycles

The proof as to why only even cycles have acceptable independent sets is beyond the scope of this work so we refer interested readers to [EM00, Lemmas 3 and 4] However, since it is not obvious, we will explain the intuition behind the proof However, before we can describe why this is the case let us first examine the algorithm

Algorithm 4 1 lists the pseudocode for El-Mabrouk algorithm It computes and duplicates the independent set for even cycles and doesn't modify all other components It is almost identical to Algorithm 2 2 except that it doesn't unorder the genome and it performs the duplication within the algorithm Like that algorithm, it is extremely efficient, running in linear time We refer readers interested in a proof of the algorithm's correctness back to [EM00]

For example, Algorithm 4 1 on genome $\mathbb{G} = \{\{\overrightarrow{y}, \overleftarrow{a_1}\}, \{\overrightarrow{a_1}, \overleftarrow{b_1}\}, \{\overrightarrow{b_1}, \overleftarrow{x}\}, \{\overrightarrow{x}, \overleftarrow{h_1}\}, \{\overrightarrow{h_1}, \overleftarrow{f_1}\}, \{\overrightarrow{f_1}, \overleftarrow{e_1}\}, \{\overrightarrow{e_1}, \overleftarrow{g_1}\}, \{\overrightarrow{g_1}, \overrightarrow{c_1}\}, \{\overleftarrow{c_1}, \overrightarrow{a_2}\}, \{\overleftarrow{a_2}, \overrightarrow{b_2}\}\}$ $\{\overleftarrow{b_2}, \overrightarrow{d_1}\}, \{\overleftarrow{d_1}, \overrightarrow{h_2}\}, \{\overleftarrow{h_2}, \overrightarrow{z}\},$

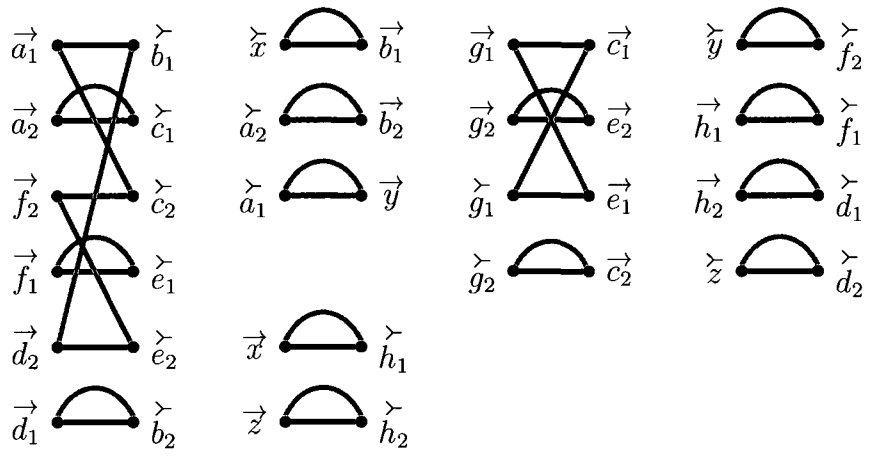


Figure 4.5: A breakpoint graph of a genome whose repeats have been minimized by Algorithm 4.1. Black edges correspond to the original genome while grey edges correspond to the genome created by Algorithm 4.1. Observe how few changes have been made to the genome as indicated by the large number of 2-cycles.

$\{\succ z, \vec{d}_2\}$, $\{\vec{d}_2, \succ e_2\}$, $\{\vec{e}_2, \vec{g}_2\}$, $\{\succ g_2, \vec{c}_2\}$, $\{\succ c_2, \vec{f}_2\}$, $\{\succ f_2, \succ y\}\}$ results in $\mathbb{H} = \{\{\vec{y}, \succ a_1\}$, $\{\vec{a}_1, \succ c_2\}$, $\{\vec{c}_2, \succ g_2\}$, $\{\vec{g}_2, \vec{e}_2\}$, $\{\succ e_2, \vec{f}_2\}$, $\{\succ f_2, \succ y\}$, $\{\vec{a}_2, \succ c_1\}$, $\{\vec{c}_1, \succ g_1\}$, $\{\vec{g}_1, \vec{e}_1\}$, $\{\succ e_1, \vec{f}_1\}$, $\{\succ f_1, \vec{h}_1\}$, $\{\succ h_1, \vec{x}\}$, $\{\succ x, \vec{b}_1\}$, $\{\succ b_1, \vec{d}_2\}$, $\{\succ d_2, \succ z\}$, $\{\vec{z}, \succ h_2\}$, $\{\vec{h}_2, \succ d_1\}$, $\{\vec{d}_1, \succ b_2\}$, $\{\vec{b}_2, \succ a_2\}\}$, their breakpoint graph is depicted in Figure 4.5. The duplication distance of $\mathbb{G}$ is 8, which is the worst possible scenario where every duplicated gene has its own repeat. However, using 3 rearrangement operations, Algorithm 4.1 reduced the duplication distance to 3, given an overall distance of 6, saving 2 operations. Observe, however, that $\mathbb{H}$ consists of 2 circular chromosomes, as opposed to 1. See Section 2.3 on how to resolve this problem.

Coming back to the question as to why independent sets can only be found for even cycles but not for odd cycles or paths, let us first understand that the rearrangement operations between the given genome and the constructed genome are not merely reversals and block interchanges but are, in fact, DCJ operations. Translocations don't occur since this is a circular unichromosomal genome, though, as can be seen in the previous example, they do, in fact, occur as errors and must be subsequently corrected.

The reason why even cycles have independent sets is because, in their case, fewer rearrangements operations combine more repeats. To understand why this only happens in the case of even cycles and not in any other cases we must first have an intuitive understanding of the DCJ distance formula.

Between two genomes, in this case the given genome and the constructed genome, every DCJ operation *finalizes* at least one adjacency, by which we mean that it modified an adjacency in the first genome to be identical an adjacency in the second genome. Occasionally, instead of just finalizing just one adjacency, it will finalize two. This is why even cycles are variables in Equation 1.3.6, for every even cycle there will be exactly one DCJ operation that finalizes two adjacencies instead of one. Otherwise, without this case, one DCJ operation would be required for every adjacency. Thus, in essence, every even cycle provides a "free" DCJ operation.

From Observation 2.2.1 we know that intersection graphs DCJ can be applied directly to any intersection graph. Thus, every even cycle in the intersection graph means a "free" DCJ operation which can be used to merge a repeat without increasing the distance. This is why for even cycles, and only even cycles, rearrangement operations have the advantage over duplications.

Equation 1.3.6 also mentions pairs of odd paths as another means of reducing the distance. While pairs of odd paths do provide "free" DCJ operations, this doesn't occur in this case. The reason why odd paths provide "DCJ" operations is that finalize an extra telomere, as opposed to an extra adjacency. Even though there are odd paths in the intersection graphs in this particular case, there are no telomeres since the chromosome is circular. Thus, odd paths in the intersection graphs of this particular instance of the rearrangement-duplication problem don't confer an advantage.

4.2.2 Error in El-Mabrouk Algorithm

El-Mabrouk claims that each repeat results in one duplication-transpositions operation. While at face value this logic seems unassailable, by flipping the problem around and looking at the problem from the biological perspective, transforming the constructed genome into the original genome *via* duplication-transpositions, reveals a fatal flaw. The flaw was revealed by [Kah07] by this simple counter-example:

Consider the given unordered ambiguous genome $\tilde{\mathbb{G}} = \{\{\overrightarrow{g}, \overleftarrow{a}\}, \{\overrightarrow{a}, \overleftarrow{b}\}, \{\overrightarrow{b}, \overleftarrow{d}\}, \{\overrightarrow{d}, \overleftarrow{e}\}, \{\overrightarrow{e}, \overleftarrow{c}\}, \{\overrightarrow{c}, \overleftarrow{d}\}, \{\overrightarrow{d}, \overleftarrow{b}\}, \{\overrightarrow{b}, \overleftarrow{c}\}, \{\overrightarrow{c}, \overleftarrow{e}\}, \{\overrightarrow{e}, \overleftarrow{f}\}, \{\overrightarrow{f}, \overleftarrow{g}\}\}$ whose intersection graph has no even cycles and, hence, Algorithm 4.1 doesn't modify. There are four repeats in $\tilde{\mathbb{G}}$: $\{\overleftarrow{b}, \overrightarrow{b}\}$, $\{\overleftarrow{c}, \overrightarrow{c}\}$, $\{\overleftarrow{d}, \overrightarrow{d}\}$ and $\{\overleftarrow{e}, \overrightarrow{e}\}$. Thus, the total distance is 4 according to El-Mabrouk. However, consider the unambiguous genome $\tilde{\mathbb{H}} = \{\{\overrightarrow{g}, \overleftarrow{a}\}, \{\overrightarrow{a}, \overleftarrow{b}\}, \{\overrightarrow{b}, \overleftarrow{c}\}, \{\overrightarrow{c}, \overleftarrow{d}\}, \{\overrightarrow{d}, \overleftarrow{e}\}, \{\overrightarrow{e}, \overleftarrow{f}\}, \{\overrightarrow{f}, \overleftarrow{g}\}\}$. We can transform $\tilde{\mathbb{H}}$ into $\tilde{\mathbb{G}}$ with three duplication-transpositions: $\{\overleftarrow{b}, \overrightarrow{b}, \overleftarrow{c}, \overrightarrow{c}\}$, $\{\overleftarrow{d}, \overrightarrow{d}\}$ and $\{\overleftarrow{e}, \overrightarrow{e}\}$. Thus, the El-Mabrouk solution is not optimal.

[Kah07] proposes another very slightly different duplication operation called simply a *duplication*. A *duplication* is identical to a duplication-transposition except that the source sequence and target sequence are separate, thus, it avoids the problem of a duplication breaking apart a repeat. Nevertheless, the algorithm for computing the duplication distance given in [Kah07] is quite complex with an $O(n^4)$ time complexity.

It remains an open problem if duplication distance can be combined with El-Mabrouk's rearrangement algorithm. Since the time complexity of duplication distance is much greater then that of El-Mabrouk's algorithm, extensive modifications are probably needed. Furthermore, [Kah07] proves that duplication distance is less than or equal to duplication-transposition distance, which may or may not be a problem.

An alternative open problem would be to find out if duplication-transposition distance can be fixed. [Kah07] discusses this possibility briefly and un-optimistically

concludes that duplication-transposition distance is much harder, although, the duplication distance at least provides a lower bound. Even if it turns out that the duplication-transposition distance can be solved some modifications may still be required to the rearrangement-duplication algorithm.

4.2.3 Rearrangement-Duplication with DCJ

El-Mabrouk limited her analysis to circular unichromosomal genomes and by removing the hurdle optimization procedures her algorithm certainly produces the optimal DCJ distance for that particular case. Extending her algorithm to handle linear chromosomes is simply a matter of determining how to handle paths ending in telomeres in the intersection graph. We conjecture that the optimal rearrangement is the same as for genome halving for odd paths ending with telomeres and otherwise to leave even paths ending with telomeres unarranged.

One curious case in the El-Mabrouk algorithm is the case of odd cycles in the intersection graph. For this case El-Mabrouk doesn't perform any rearrangements but, for genome halving algorithms in the circular unichromosomal case, up two odd cycles can contribute "free DCJ" operations. Does this advantage not occur in the rearrangement-duplication case or is this an error similar to the error found by [AP07a] in the El-Mabrouk-Sankoff genome halving algorithm for circular unichromosomal genomes?

4.2.4 Arbitrary Size Gene Families

In [EM00] El-Mabrouk proposed a heuristic for dealing with gene families on any size that relied on extra information beyond what can be obtained from the genome. Specifically, she used a *gene tree* for assistance. Despite the help of the extra information her algorithm is amazingly still only a heuristic.

Perhaps the most interesting area of future research within this section is to

see if either the Warren-Sankoff genome aliquoting algorithm or the Feijão-Meidanis genome aliquoting algorithm could be modified for this problem. The main difficulty in making such modifications would likely be in determining in which cases it is appropriate to perform rearrangements. However, it unlikely that the cases where rearrangements are appropriate when gene families are restricted to size 2 will change very much for gene families of arbitrary size so this may be very easy.

If either (or both) of the genome aliquoting algorithms can be modified to handle instances of the rearrangement-duplication problem it would be interesting to compare their performance with that of El-Mabrouk's algorithm. Beyond being a comparison of algorithms, such a comparison would also be comparing the benefit of having the extra information provided by a gene tree.

Appendix A

Bibliography

[AFRV06] Sébastien Angibaud, Guillaume Fertin, Irena Rusu, and Stéphane Vialette. How pseudo-boolean programming can help genome rearrangement distance computation. In Guillaume Bourque and Nadia El-Mabrouk, editors, *RECOMB CG '06: Proceedings of the fourth RECOMB satellite workshop on Comparative Genomics*, volume 4205 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 2006.

[AP07a] Max A. Alekseyev and Pavel A. Pevzner. Whole genome duplications and contracted breakpoint graphs. *SIAM Journal on Computing*, 36(6):1748–1763, 2007.

[AP07b] Max A. Alekseyev and Pavel A. Pevzner. Whole genome duplications, multi-break rearrangements, and genome halving problem. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *SODA '07: Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 665–679. Society for Industrial and Applied Mathematics, 2007.

[BMS05] Anne Bergeron, Julia Mixtacki, and Jens Stoye. On sorting by translocations. *Journal of Computational Biology*, pages 615–629, 2005.

[BMS06] Anne Bergeron, Julia Mixtacki, and Jens Stoye. A unifying view of

genome rearrangements. In Philipp Bücher and Bernard M.E. Moret, editors, *WABI '06: Proceedings of the sixth Workshop on Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 163–173. Springer-Verlag, 2006.

[BMS08] Anne Bergeron, Julia Mixtacki, and Jens Stoye. HP distance via double cut and join distance. In Paolo Ferragina and Gad M. Landau, editors, *CPM '08: Proceedings of the 19th Symposium of Combinatorial Pattern Matching*, volume 5029 of *Lecture Notes in Computer Science*, pages 56–68. Springer-Verlag, 2008.

[BMY01] David A. Bader, Bernard M.E. Moret, and Mi Yan. A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. *Journal of Computational Biology*, 8(5):483–491, 2001.

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

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

[Bry00] David Bryant. The complexity of calculating exemplar distances. In David Sankoff and Joseph H. Nadeau, editors, *Comparative Genomics: Empirical and Analytical Approaches to Gene Order Dynamics, Map Alignment, and the Evolution of Gene Families*, volume 1 of *Computational Biology Series*, pages 207–212. Kluwer Academic Pubs, Norwell, MA, USA, 2000.

[Cap97] Alberto Caprara. Sorting by reversals is difficult. In *RECOMB '97: Pro-*

ceedings of the first Conference on Research in Computational Molecular Biology, pages 75–85. Association for Computing Machinery, 1997.

[Chr96] David A. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169, 1996.

[CI01] David A. Christie and Robert W. Irving. Sorting strings by reversals and by transpositions. *SIAM Journal of Discrete Mathematics*, 14(2):193–206, 2001.

[DM01] Zanoni Dias and João Meidanis. Genome rearrangements distance by fusion, fission, and transposition is easy. In *SPIRE '01: Proceedings of the eighth International Symposium on String Processing and Information Retrieval*, pages 250–253. IEEE Computer Society, 2001.

[Edm65a] Jack Edmonds. Maximum matching and a polyhedron with 0, 1 vertices. *Journal of Research of the National Bureau of Standards*, 69(B):125–130, 1965.

[Edm65b] Jack Edmonds. Paths, trees and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

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

[EM00] Nadia El-Mabrouk. Genome rearrangement by reversals and insertions/deletions of contiguous segments. In David Sankoff and Raffaele Giancarlo, editors, *CPM '00: Proceedings of the 11th Symposium on Combinatorial Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 222–234. Springer-Verlag, 2000.

[EM02] Nadia El-Mabrouk. Reconstructing an ancestral genome using minimum segments duplications and reversals. *Journal of Computer and System Sciences*, 65(3):442–464, 2002.

[EM05] Nadia El-Mabrouk. Genome rearrangements with gene families. In Olivier Gascuel, editor, *Mathematics of Evolution and Phylogeny*, chapter 11, pages 291–320. Oxford University Press, New York, NY, USA, 2005.

[EMS03] Nadia El-Mabrouk and David Sankoff. The reconstruction of doubled genomes. *SIAM Journal on Computing*, 32:754–792, 2003.

[FaM09] Pedro Feijão and João Meidanis. SCJ: a novel rearrangement operation for which sorting, genome median and genome halving problems are easy. In Steven Salzberg and Tandy Warnow, editors, *WABI '09: Proceedings of the ninth International Workshop on Algorithms in Bioinformatics*, volume 5724 of *Lecture Notes in Computer Science*, pages 85–96. Springer-Verlag, 2009.

[FLR+09] Guillaume Fertin, Anthony Labarre, Irena Rusu, Éric Tannier, and Stéphane Vialette. *Combinatorics of Genome Rearrangements*. The MIT Press, Cambridge, Massachusetts, 2009.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[Got82] Osamu Gotoh. An improved algorithm for matching biological sequences. *Journal of Molecular Biology*, 162:705–708, 1982.

[Gre05] T. Ryan Gregory, editor. *The Evolution of the Genome*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2005.

[GWN+03] Yong Gao, Junfeng Wu, Robert Niewiadomski, Yang Wang, Zhi-Zhong Chen, and Guohui Lin. A space efficient algorithm for sequence alignment with inversions. In Tandy Warnow and Binhai Zhu, editors, *COCOON '03: Proceedings of the nineth International Computing and Combinatorics Conference*, volume 2697 of *Lecture Notes in Computer Science*, pages 57–67. Springer-Verlag, 2003.

[Han96] Sridhar Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71(1-3):137–151, 1996.

[HHTL09] Yen-Lin Huang, Cheng-Chen Huang, Chuan Yi Tang, and Chin Lung Lu. An improved algorithm for sorting by block-interchanges based on permutation groups. *Information Processing Letters*, Submitted, 2009.

[HP95] Sridhar Hannenhalli and Pavel A. Pevzner. Transforming men into mice. In *FOCS '95: Proceedings of the 36th Symposium on Foundations of Computer Science*, pages 581–592. IEEE Computer Society, 1995.

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

[JN07] Géraldine Jean and Macha Nikolski. Genome rearrangements: a correct algorithm for optimal capping. *Information Processing Letters*, 104(1):14–20, 2007.

[Kah07] Crystal L Kahn. Duplication distance. Technical report, Brown University, Providence, RI, USA, 2007.

[KR95] John D. Kececioglu and R. Ravi. Of mice and men: algorithms for evolutionary distances between genomes with translocation. In Guillaume

Bourque and Nadia El-Mabrouk, editors, *SODA '95: Proceedings of the sixth ACM-SIAM Symposium on Discrete algorithms*, pages 604–613. Society for Industrial and Applied Mathematics, 1995.

[KR08] Crystal L. Kahn and Benjamin J. Raphael. Analysis of segmental duplications via duplication distance. *Bioinformatics*, 24(16):i133–i138, 2008.

[KS95] John D. Kececioglu and David Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13:180–210, 1995.

[Lev65] Vladimir I. Levenshtein. Binary code capable of correcting deletions, insertions, and reversals. *Soviet Physics-Doklady*, 10(8):845–848, 1965.

[LHWC06] Chun Lung Lu, Yen-Lin Huang, Tsui Ching Wang, and Hsien-Tai Chiu. Analysis of circular genome rearrangement by fusions, fissions and block-interchanges. *BMC Bioinformatics*, 7:295, 2006.

[LLCT05] Ying Chih Lin, Chin Lung Lu, Hwan-You Chang, and Chuan Yi Tang. An efficient algorithm for sorting by block-interchanges and its application to evolution of vibrio species. *Journal of Computational Biology*, 12(1):102–112, 2005.

[LMB03] Tao Liu, Bernard M.E. Moret, and David A. Bader. An exact, linear-time algorithm for computing genomic distances under inversions and deletions. Technical report, University of New Mexico, 2003.

[LP09] László Lovász and Michael D. Plummer. *Matching Theory*. AMS Chelsea Publishing, Providence, Rhode Island, 2009.

[LQWZ04] Guojun Li, Xingqun Qi, Xiaoli Wang, and Binhai Zhu. A linear-time algorithm for computing translocation distance between signed genomes. In

Süleyman Cenk Sahinalp, S. Muthu Muthukrishnan, and Ugur Dogrusöz, editors, *CPM '04: Proceedings of the 15th Symposium on Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 323–332. Springer-Verlag, 2004.

[Mix08] Julia Mixtacki. Genome halving under DCJ revisited. In Xiaodong Hu and Jie Wang, editors, *COCOON '08: Proceedings of the 14th International Computing and Combinatorics Conference*, volume 5092 of *Lecture Notes in Computer Science*, pages 276–286. Springer-Verlag, 2008.

[MM07] Cleber Mira and João Meidanis. Sorting by block-interchanges and signed reversals. In *ITNG '07: Proceedings of the fourth International Conference on Information Technology*, pages 670–676. IEEE Computer Society, 2007.

[MS09] Paul Medvedev and Jens Stoye. Rearrangement models and single-cut operations. In Francesca Ciccarelli and István Miklós, editors, *RECOMB CG '09: Proceedings of the seventh RECOMB satellite workshop on Comparative Genomics*, volume 5817 of *Lecture Notes in Computer Science*, pages 84–97. Springer-Verlag, 2009.

[MSM03] Mark Marron, Krister M. Swenson, and Bernard M. E. Moret. Genomic distance under deletions and insertions. In Tandy Warnow and Binhai Zhu, editors, *COCOON '03: Proceedings of the nineth International Computing and Combinatorics Conference*, volume 2697 of *Lecture Notes in Computer Science*, pages 57–67. Springer-Verlag, 2003.

[NW69] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1969.

[OFS03] Michal Ozery-Flato and Ron Shamir. Two notes on genome rearrangement. *Journal of Bioinformatics and Computational Biology*, 1:71–94, 2003.

[OFS06] Michal Ozery-Flato and Ron Shamir. Sorting by translocations via reversal theory. In Guillaume Bourque and Nadia El-Mabrouk, editors, *RECOMB CG '06: Proceedings of the fourth RECOMB satellite workshop on Comparative Genomics*, volume 4205 of *Lecture Notes in Computer Science*, pages 87–98. Springer-Verlag, 2006.

[Ohn70] Susumu Ohno. *Evolution by Gene Duplication*. Springer-Verlag, Berlin / Heidelberg, 1970.

[QLLX06] Xingqin Qi, Guojun Li, Shuguang Li, and Ying Xu. Sorting genomes by translocations and deletions. In Peter Markstein and Ying Xu, editors, *CSB '06: Proceedings of the fifth International Conference on Computational Systems Bioinformatics*, pages 157–166. Imperial College Press, 2006.

[RSW05] A. Jamie Radcliffe, Alexander D. Scott, and Elizabeth L. Wilmer. Reversals and transpositions over finite alphabets. *SIAM Journal of Discrete Mathematics*, 19(1):224–244, 2005.

[San92] David Sankoff. Edit distances for genome comparisons based on non-local operations. In Alberto Apostolico, Maxime Crochemore, Zvi Galil, and Udi Manber, editors, *CPM '92: Proceedings of the third Symposium on Combinatorial Pattern Matching*, pages 121–135. Springer-Verlag, 1992.

[San99] David Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909–917, 1999.

[SB97] David Sankoff and Mathieu Blanchette. The median problem for breakpoints in comparative genomics. In Tao Jiang and Der-Tsai Lee, editors, *COCOON '97: Proceedings of the third International Conference on Computing and Combinatorics*, pages 251–264. Springer-Verlag, 1997.

[SG89] David Sankoff and Martin Goldstein. Probabilistic models of genome shuffling. *Bulletin of Mathematical Biology*, 51(1):117–124, 1989.

[SW92] Michael Schöniger and Michael S. Waterman. A local algorithm for DNA sequence alignment with inversions. *Bulletin of Mathematical Biology*, 54(4):521–536, 1992.

[Tes02] Glenn Tesler. Efficient algorithms for multichromosomal genome rearrangement. *Journal of Computer and Systems Sciences*, 65(3):587–609, 2002.

[TM03] Jijun Tang and Bernard M.E. Moret. Phylogenetic reconstruction from gene-rearrangement data with unequal gene content. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Michiel H. M. Smid, editors, *WADS '03: Proceedings of the 14th Workshop on Algorithms and Data Structures*, volume 2748 of *Lecture Notes in Computer Science*, pages 37–46. Springer-Verlag, 2003.

[TZS09] Eric Tannier, Chunfang Zheng, and David Sankoff. Multichromosomal median and halving problems under different genomic distances. *Bioinformatics*, 10:120, 2009.

[VAL06] Augusto F. Vellozo, Carlos E. R. Alves, and Alair Pereira do Lago. Alignment with non-overlapping inversions in $O(n^3)$-time. In Philipp Bücher and Bernard M. E. Moret, editors, *WABI '06: Proceedings of the sixth*

Workshop on Algorithms in Bioinformatics, volume 4175 of *Lecture Notes in Computer Science*, pages 186–196. Springer-Verlag, 2006.

[Wat83] Michael S. Waterman. Efficient sequence alignment algorithms. *Journal of Theoretical Biology*, 108:333–337, 1983.

[WEHM82] G. A. Watterson, W. J. Ewens, T. E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99:1 – 7, 1982.

[WS09a] Robert Warren and David Sankoff. Genome aliquoting with double cut and join. *BMC Bioinformatics*, 10(1):S2, 2009.

[WS09b] Robert Warren and David Sankoff. Genome halving with double cut and join. *Journal of Bioinformatics and Computational Biology*, 7(2):357–371, 2009.

[WSB76] Michael S. Waterman, T.F. Smith, and W.A. Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20:367–387, 1976.

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

[YF08] Sophia Yancopoulos and Richard Friedberg. Sorting genomes with insertions, deletions and duplications by DCJ. In Craig E. Nelson and Stéphane Vialette, editors, *RECOMB CG '08: Proceedings of the sixth RECOMB satellite workshop on Comparative Genomics*, volume 5267 of *Lecture Notes in Computer Science*, pages 170–183. Springer-Verlag, 2008.

[ZW06] Daming Zhu and Lusheng Wang. On the complexity of unsigned translocation distance. *Theoretical Computer Science*, 352(1):322–328, 2006.