Diplomarbeit

On the Algorithmic Tractability of Single Nucleotide Polymorphism (SNP) Analysis and Related Problems

Sebastian Wernicke

23. September 2003

Gutachter:
PD Dr. Rolf Niedermeier
Prof. Dr. Franz Oberwinkler

Betreuer:
PD Dr. Rolf Niedermeier
Dr. Jochen Alber Dr. Jens Gramm Dipl.-Inform. Jiong Guo

Nachwuchsgruppe Theoretische Informatik / Parametrisierte Algorithmen
Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Alle Stellen, die im Wortlaut oder dem Sinne nach anderen Werken entnommen sind, wurden durch Quellenangaben als Entlehnung kenntlich gemacht.

Tübingen, den 23. September 2003 *Sebastian Wernicke*

Contents

1 Introduction **1**
1.1 The Human Genome and SNPs . . . 1
1.2 Overview of this Work . . . 2

2 Biological Background and Motivation **5**
2.1 Basic Genetic Terminology . . . 5
2.2 An Introduction to SNPs . . . 7
2.3 Importance and Prospects of SNP Mapping . . . 9
2.3.1 SNPs in the Study of Population History . . . 9
2.3.2 SNPs and Pharmacogenetics . . . 11

3 Computer Science Preliminaries and Notation **15**
3.1 Notation for Matrices and Graphs . . . 15
3.2 Crash Course in Computational Complexity Theory . . . 16
3.2.1 Machine-Independent Analysis . . . 16
3.2.2 Running Time—Keeping Score . . . 18
3.2.3 Complexity Classes . . . 22
3.3 Fixed-Parameter Tractability (FPT) . . . 24
3.3.1 An Efficient Algorithm for Vertex Cover . . . 24
3.3.2 Formal Definition and Aspects of FPT . . . 27

4 Submatrix Removal Problems **31**
4.1 Definitions and Terminology . . . 31
4.2 A Reduction to d-Hitting Set . . . 35
4.2.1 Finding Forbidden Submatrices . . . 35
4.2.2 Approximability and Fixed-Parameter Tractability Results . . . 37
4.3 Hardness Results . . . 39
4.3.1 Overview of Results—Four Theorems . . . 39
4.3.2 Proofs for Theorems 4.13 and 4.14 . . . 40

4.3.3 Proof of Theorem 4.11 . . . 47
4.3.4 Proof of Theorem 4.12 . . . 49
4.4 Discussion and Future Extensions . . . 53

5 Perfect Phylogeny Problems 55
5.1 Phylogenetic Trees . . . 55
5.1.1 Introduction and Motivation . . . 55
5.1.2 Formal Definition . . . 56
5.2 Perfect Phylogeny Problems . . . 58
5.3 Relation to Forbidden Submatrix Problems . . . 60
5.4 Minimum Species Removal . . . 62
5.5 Minimum Character Removal . . . 66

6 Graph Bipartization 69
6.1 Introduction and Known Results . . . 69
6.2 Reducing Edge Bipartization to Vertex Bipartization . . . 72
6.3 A Branch&Bound Approach . . . 75
6.3.1 Initial Heuristics . . . 76
6.3.2 Data Reduction Rules . . . 79
6.4 Implementation and Comparison of the Algorithms . . . 89
6.4.1 Using the Program . . . 89
6.4.2 Some Implementation Details . . . 90
6.4.3 Tests and Test Results . . . 92

7 Using Graph Bipartization in SNP Analysis 99
7.1 Introduction and Overview of Results . . . 99
7.2 SNP Haplotype Assembly . . . 100
7.3 Inferring Haplotypes from Genotypes . . . 103
7.3.1 Minimum Genotype Removal . . . 109
7.3.2 Minimum Site Removal . . . 112
7.4 Testing Branch&Bound on SNP Data . . . 115

8 Conclusion 119
8.1 Summary of Results and Future Extensions . . . 119
8.2 Acknowledgments . . . 121

List of Figures 124

Bibliography 125

Chapter 1

Introduction

1.1 The Human Genome and SNPs

Throughout its life, an individual's hereditary potentials and limits are determined by its very own genes. Consequently, a lot of effort has been put into the Human Genome Project. As of April 2003, 95.8% of the human genome has been sequenced in a very high quality (see [NCBI03] for up-to-date information) and a goal has been set asking for a complete sequence due the end of this year as more and more chromosomes become fully mapped (see, e.g., [Heil03]). However—quoting from [WeHu02]— this achievement is merely the foundation for far deeper research as

> *"... we are ending the era of determining the sequence of the genetic code and entering the beginning of the age of deciphering the biology of life underlying that code."*

Hearing that a 95.8 percentile portion of the human genome has been sequenced, one is immediately bound to ask as to which human's genome we are actually referring to—after all, every human has a unique genetic markup. At the beginning of the sequencing process by the Human Genome Project, geneticists thought they would indeed have to make a choice as to who would be chosen to provide a reference sequence of the four nucleotides A, C, G, and T of his DNA.[1] One individual would surely constitute a "blueprint" of the human species, the study of genetic variation among our species would however gain little insight from this [Chak01].

The sequence we have obtained by the Human Genome Project fortunately was not obtained through making that kind of choice: Genetic variation among humans can—in almost every case—be traced back to variations that occur within a single nucleotide. Such a site where there are two different nucleotides to be found in two different DNAs, is commonly referred to as a Single Nucleotide Polymorphism (SNP, pronounced "snip") [HGSC01]. A simple definition is given by [Ston01]:

> *"... DNA is a linear combination of four nucleotides; compare two sequences, position by position, and wherever you come across different nucleotides at the same position, that's a SNP."*

[1] A thorough introduction to genetic terminology including SNPs is given in Chapter 2

During the Human Genome Project, 1.4 million sites of genetic variations have been mapped [SNP01] along a reference sequence composed of hundreds of different genomes. The reason why it was possible to combine such a multitude of different individuals' genomes into a coherent map of the human genome lies in the fact that DNA is mostly conserved around SNPs . The importance of SNPs is outlined e.g. in [Ston01] who refers to them as the *"bread and butter of DNA sequence variation"* for they are witnesses of unique past mutations in our genetic markup. Therefore, SNPs can give valuable hints about common evolutionary ancestors. But there is an even more severe economic influence: As genes are widely held responsible for the likeliness for the acquisition of certain diseases and the responsiveness to various medical treatments, SNPs can either be made directly responsible for such a variation or they may at least aid in the identification of the corresponding gene.[2]

In this work, we shall deal with topics from the field of theoretical bioinformatics that are connected to SNPs. Recent research [LBILS01, EHK03] has shown that all the useful applications and prospects of SNP data come at a price: Many computational problems arising during the acquisition and application of SNP data have been proven to be computationally "hard", meaning that they are widely believed to be impossible to solve in reasonable time .[3] However, there are techniques such as fixed-parameter tractability and data-reduction (both to be introduced in more detail throughout this work) that allow even "hard" problems to be solved efficiently in practical applications. This work explores the possible use of these techniques for computationally hard problems connected to SNP analysis.

1.2 Overview of this Work

The main part of this work (Chapters 2 to 7) can be divided into three parts:

- **Part 1 (Chapters 2 and 3): Introducing the Terminology**
  This work brings together two areas of science—biology and informatics—that have only recently been connected in the emerging (and vastly growing) research field of *bioinformatics*. In order to achieve a common basis for Parts 2 and 3 of this work, Part 1 intends to introduce the computer scientist to the relevant biological background and terminology (Chapter 2), and to familiarize the biologist with the relevant topics from theoretical computer science (Chapter 3).

  Chapter 2 first introduces some terminology from the field of genetics, thereby defining SNPs. We then motivate the analysis of SNPs by two applications: The analysis of evolutionary development and the field of pharmacogenetics. Especially the field of pharmacogenetics is capable of having an enormous impact on medicine and the pharmaceutical industry in the near future by using SNP data to predict the efficacy of medication.

  Chapter 3 gives a brief introduction to the field of computational complexity. We will see and motivate how algorithms are analyzed in theoretical computer science. This will lead to the definition of "complexity classes", introducing the class *NP* which includes computationally hard problems. Some of the hard problems in the class *NP* can be solved efficiently using the tool of *fixed-parameter tractability*, introduced at the end of this chapter.

[2] Section 2.3 gives a detailed introduction to the prospects of SNP mapping and analysis.
[3] Chapter 3 introduces the topic of computational hardness in more detail.

- **Part 2 (Chapters 4 and 5): Applying SNP Data (Perfect Phylogenies)**
  An important application of SNP data is in the analysis of the evolutionary history of species development (*phylogenetic analysis*). As will be made plausible in Chapter 5, using SNP data is—in many ways—superior to previous approaches of phylogenetic analysis. In order to analyze the development of species using SNP data, an underlying model of evolution must be specified. A popular model is the so-called *perfect phylogeny*, but the construction of this phylogeny is a computationally hard problem when there are inconsistencies (such as read-errors or an imperfect fit to the model of perfect phylogeny) in the underlying data.

  Chapter 4 analyzes the problem of "forbidden submatrix removal" which is closely connected to constructing perfect phylogenies—we will see in Chapter 5 that its computational complexity is directly related to that of constructing a perfect phylogeny from data which is partially erroneous. In this chapter, we analyze the algorithmic tractability of "forbidden submatrix removal", characterizing cases where this problem is *NP*-complete (being fixed-parameter tractable in general).

  Chapter 5 introduces the concept, motivation, and some known results for phylogenetic analysis. We then apply the results from Chapter 4 to perfect phylogeny problems, i.e., the problem of dealing with data-inconsistencies with respect to the underlying evolutionary model of perfect phylogeny. It will be shown that these problems are all fixed-parameter tractable and can be efficiently solved using existing algorithms.

- **Part 3 (Chapters 6 and 7): Obtaining SNP Data**
  Basically, obtaining SNP data requires sequencing two DNA strands and comparing them to each other. The problems lie in the details: Firstly, current techniques only allow sequences of at most 500 base pairs in length to be sequenced as a whole, and secondly, it is—in terms of cost and labor—often only possible to detect the presence of SNP sites rather than being able to tell which of the two DNAs contained which base. Part 3 of this work analyzes the computational complexity of these two problems by relating them to a graph-theoretic problem[4] called Graph Bipartization.

  Chapter 6 introduces the computationally hard problem of Graph Bipartization, stating some known results and showing the relative hardness of the two Graph Bipartization-problem variants Edge Bipartization and Vertex Bipartization (the latter one of which is proven to be at least as hard as the former one). Following this introduction, we develop and test practical algorithms for Graph Bipartization. These algorithms—although they require a long time even for medium-sized *general* graphs—prove to be efficient for the Graph Bipartization problems that arise during the acquisition of SNPs, even if these graphs contain a few hundred vertices.

  Chapter 7 introduces a formal definition of the computational problems of SNP analysis and proves their close relationship to Graph Bipartization. The last section of this chapter shows that the algorithms developed in Chapter 6 can be used to efficiently solve the presented problems by solving their corresponding Graph Bipartization problem.

This work is concluded by Chapter 8, presenting a summary of results and suggestions for future research related to this work.

[4] Graphs are introduced in Chapter 3

Chapter 2

Biological Background and Motivation

In this chapter, we establish some basic terminology from the field of genetics used throughout this work[1]. Afterwards, we introduce SNPs and current techniques used to detect and map them. The last section of this chapter provides an introduction to pharmacogenetics—the area that sparked economic and scientific interest in SNPs.

2.1 Basic Genetic Terminology

All living organisms encode their genetic information in the form of *deoxyribonucleic acid* (*DNA*, for short). DNA is a double-helix polymer where each strand is a long chain of polymerized monomer *nucleotides*.[2] Basically, these are four different nucleotides; to a deoxyribose sugar bonded with a phosphate, one of four possible bases is attached to form a nucleotide; these possible bases include the two *purines* (*adenine* and *guanine*) and the two *pyrimidines* (*cytosine* and *thymine*).[3] For abbreviation, the nucleotides in a strand of DNA are denoted by the first letter of their respective base (adenine by A, guanine by G, cytosine by C, and thymine by T). The nucleotides are joined to form a single strand of DNA by covalently[4] bonding a phosphate of one nucleotide with the sugar of the next, a strand starts with a sugar (this end is called the *3'-end*) and ends with a phosphate (called *5'-end*).[5] Two single strands of DNA are held together by hydrogen bonds between the bases of opposing nucleotides in the double-strand. These bonds specifically bind adenine with thymine and cytosine with guanine. The structure of DNA is shown in Figure 2.1. Although being

[1] For a more thorough introduction on genetics, see, e.g., [GMS00], [GGL02] or the chapters on genetics in biochemistry books such as [VoVo95] or [BJS02].

[2] In general, *polymer* designates the class of very large molecules (*macromolecules*) that are multiples of simpler chemical units called monomers.

[3] Actually, DNA has a lot less homogenous buildup than this because the nucleotides can be further modified by an organism at their deoxyribose sugar. For example, bacteria use such a modification to be able to distinguish their own DNA from foreign DNA coming, e.g., from a phage (bacterial virus). However, such modifications will not have to concern us in this work because even in the presence of them, the basic principles of DNA replication and translation hold.

[4] A covalent bond is the interatomic linkage that results from two atoms forming a common electron orbital.

[5] The terms 3' and 5' are due to the enumeration of the carbon atoms in the deoxyribose sugar.

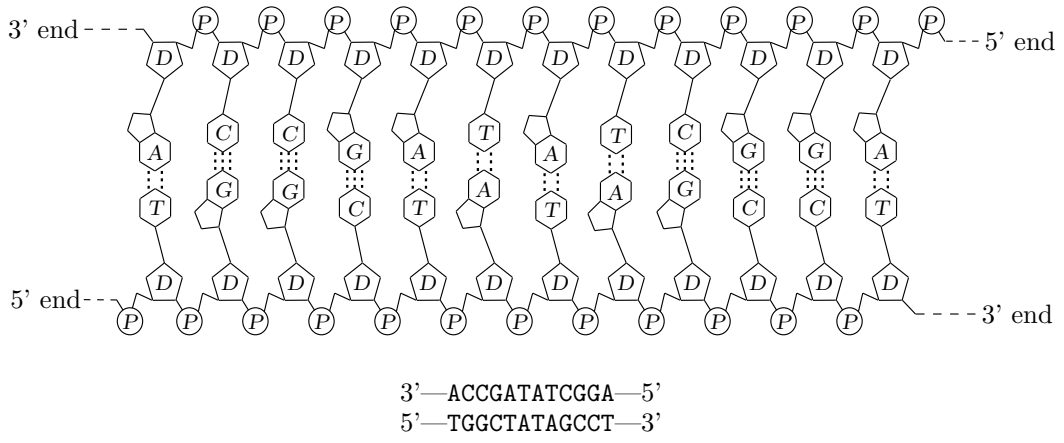


Figure 2.1: Chemical structure of DNA (above) and its abbreviated notation (below). The letters within the molecular structure above stand for phosphate (P), deoxyribose (D), adenine (A), guanine (G), cytosine (C), and thymine (T). The dashed vertical lines indicate hydrogen bonds.

just two strands of bases attached to a phosphate-sugar backbone, DNA can be extremely long by molecular measures.[6] In order for the DNA to fit into a single cell[7] whilst still being accessible for replication and transcription into RNA to make proteins, human DNA is organized into very dense complexes of proteins and DNA, called *chromosomes*. Humans have 22 pairs of autosomes[8] and one pair of sex chromosomes (with females carrying two "X" chromosomes and men one "X" and one "Y" chromosome) within—almost—each cell.

The complete DNA sequence of an organism is called its *genome*, its genetic constitution as a whole is called *genotype*[9]. Genetic areas of interest in a genome are called *loci*[10]. A *gene* is a unit of DNA that encodes hereditary information, i.e., the sequence of all proteins expressed by an organism, on a locus of an individual's chromosome.[11] Any one of two or more genes that may occur alternatively at a given locus on a chromosome is called an *allele*. A combination of alleles that is likely to be inherited as a whole and may be found on one chromosome is called a *haplotype*. The sequence of DNA within a gene determines the synthesis of proteins, experiments indicating that each gene is responsible for the synthesis of one protein. Each one of the 20 proteinogenic amino acids[12] is encoded by one or more triplets of bases. *Mutations*, disruptions altering the genetic information (and therefore in most cases the corresponding protein as well), may be due to deleting, inserting, replacing, or rearranging nucleotides in a gene; they are responsible for the unique individual genetic markup of organisms. As already mentioned above, a human cell contains two copies of every chromosome (excluding the gender-specific chromosomes X and Y), where one copy is inherited from each parent. Since each parent has its unique genetic markup, equivalent

[6] E.g., the diploid DNA of a human being has a total length of approximately 1.8m [VoVo95].

[7] With a few exceptions, every cell in a living organism contains its whole hereditary information.

[8] Autosomes are those chromosomes that control the inheritance of all characteristics except sex-linked ones

[9] The genotype of an organism is the basis for its *phenotype*, where phenotype denotes the two or more distinct forms of a characteristic within a population.

[10] Loci is the plural form of "locus".

[11] Note that the majority of DNA is presumed *not* to contain any genetic information [VoVo95].

[12] Proteins are basically chains of polymerized amino acids.

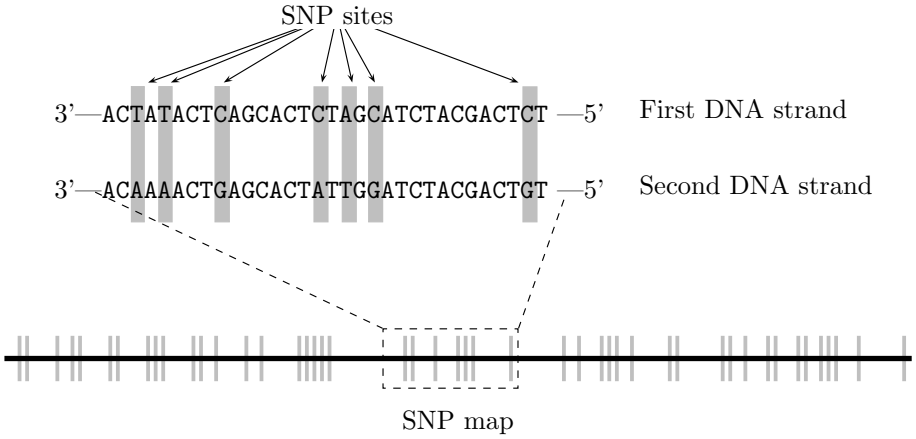


Figure 2.2: Mapping SNPs by comparison of two individuals' DNA sequence. Note that as mentioned in the text, a single nucleotide variation must occur in at least 1% of a population's individuals in order to be called "polymorphism" instead of "substitution".

genes in the two chromosomes may differ. Identical alleles on both chromosomes are referred to as being *homozygous*, different alleles are denoted *heterozygous*.

2.2 An Introduction to SNPs

A *polymorphism* is a region of the genome that varies between different individuals.[13] Consequently, a *single nucleotide polymorphism* (SNP, pronounced "Snip") is a genetic variation caused by the change of one single nucleotide (see Figure 2.2). These variations occur quite frequently among humans—on average, a SNP may be found approximately every $1.91 \cdot 10^3$ bases ("1.91 kilobases"), implying that over 90% of sequences longer than 20 kilobases will contain a SNP [Chak01]. SNPs are not evenly distributed across chromosomes, most genes contain just one or two SNPs. Currently, 93% of all genes are known to contain at least one SNP [SNP01]. Depending on whether they are found within genes or not, SNPs are either labeled cSNPs (*coding* SNPs) or ncSNPs (*non-coding*) SNPs. Generally, ncSNPs appear more frequently than cSNPs [Mu02]. Recall from the last section that more than one triplet of bases may encode a certain amino acid. Often, triplets that encode the same amino acid differ in a single nucleotide from each other. If a cSNP does not introduce an amino acid change in the encoded protein, it is named sSNP (*synonymous* SNP), and nsSNP (*non-synonymous* SNP) otherwise.[14] In the human genome, the ratio of sSNPs to snSNPs is approximately one to one [Carg99].

Before outlining some prospects and the scientific as well as economic impact of SNP analysis, we will now give a brief overview as to how SNPs are identified. In his survey on the usage of SNPs as a tool in human genetics, Gray [GCS00] names four methods for SNP detection:

[13] More precisely, a polymorphism has been defined as "*the least common allele occurring in 1% or greater of a population*" [Mare97], thereby distinguishing a polymorphism from a *substitution* that may occur in less than 1% of a population's individuals.

[14] For example, there is an nsSNP in the gene for the HLA-H protein, where a crucial disulfide bond is disrupted by changing the 282nd amino acid from cysteine to tyrosine, causing a metabolic disorder known as hereditary hemochromatosis [PSS97].

- *Identification of single strand conformation polymorphisms (SSCPs):* In this technique, DNA fragments of a locus containing the presumed SNP are amplified (e.g., multiplied into many identical fragments) using PCR amplification.[15] These fragments are then put on a polyacrylamide gel to which a current of diluting liquid is applied. Due to different folding of DNA fragments with different sequences, the speed of fragments will differ if they contain SNPs. The presence of SNPs may afterwards be confirmed by sequencing the respective patterns. This method is widely deprecated because of its low throughput and sometimes poor detection rate of about 70%.

- *Heteroduplex Analysis:* During PCR amplification of an individual that is heterozygous for a SNP, a heteroduplex[16] may be formed between two strands that are complementary to each other with exception of the SNP site. These heteroduplexes can then be detected either as a gel band (analogously to SSCP detection) or using high-performance liquid chromatography (HPLC). This SNP detection method combines reasonable throughput rates of 10 minutes per sample with a high detection rate between 95% and 100%.

- *Direct DNA sequencing:* This is the currently favored high-throughput method for detecting SNPs. According to [Carg99], almost a million base pairs can be analyzed in 48 hours with detection rates for heterozygotes ranging between 95% (using cheap "dye-terminator sequencing") and 100% (using a more expensive and laborious method known as "dye-primer sequencing"). Dye-terminator sequencing has been used by the SNP Consortium [Hold02] which published over 1.4 million SNPs in human DNA [SNP01]. Comparing equal loci in different versions of high-quality DNA sequences has recently led to an increase of *in silico* detection of SNPs.

- *Variant detector arrays (VDAs):* In this technique, glass chips with arrays of oligonucleotides are used to bind specific sequences derived in PCR amplification. VDA has a quality comparable to dye-terminator sequencing and is especially useful in rapidly scanning large amounts of DNA sequence.[17]

It should also be stressed that for SNP detection, an appropriate set of alleles from which SNPs are to be inferred needs to be chosen, as the different alleles occur with quite different frequencies in different populations (such as human ethnic groups).

In Chapter 7, we will be concerned with the algorithmic tractability of two problems that arise during the identification of SNPs: First, the sequencing of chromosomes in order to obtain haplotypes has to deal with some errors in the reading and assembling process of the DNA sequences (this will be discussed in more detail in Chapter 7). Second, haplotypes are—due to prohibitively high cost and labor—seldomly identified by sequencing single chromosomes. Rather, genotype information (both copies of a chromosome) is obtained, from which haplotypes can be inferred under certain assumptions. We will see in Chapter 7 that both problems are closely related (in a certain way even equivalent) to a problem called "graph bipartization" for which we will develop efficient algorithms in Chapter 6.

[15] The *polymerase chain reaction* (PCR, for short) can quickly and accurately make numerous identical copies of a specific DNA fragment. A PCR machine is capable of producing billions of copies of a DNA fragment in just a few hours. PCR is a widely used technique in diagnosing genetic diseases, detecting low levels of viral infection and for genetic fingerprinting in forensic medicine.

[16] A heteroduplex may either be a piece of DNA in which the two strands are different, or it is the product of annealing a piece of mRNA and the corresponding DNA strand.

[17] SNP identification through arrays is a rapidly growing market responsible for the recent development of biotechnology companies such as Affymetrix, Applied Biosystems, Marligen Biosciences, and Orchid Biosciences, among many others.

2.3 Importance and Prospects of SNP Mapping

SNPs are mainly useful for two areas of research: the study of population history (e.g., see [BBNE03]) and—an area of great economical significance—pharmacogenetics (e.g., see [Rose00]). In this section, we will introduce both applications. The algorithmic tractability of some problems in the study of population history using SNPs is dealt with in Chapter 5.

2.3.1 SNPs in the Study of Population History

SNPs often are a basis for various studies of population history (e.g., see [Tish96], [Tish00], and [Mu02]). Historically, such studies employed gene trees of non-recombining loci inherited from one parent such as mitochondrial DNA or the Y chromosome [Avis94]. A disadvantage of this approach is that such loci are subject to a lot of stochastic parameters, which in turn caused the requirement of vast amounts of loci to be analyzed to gain confidence over the results. The preference for SNPs as genetic markers arose from this problem, as SNPs provide a broad range of unlinked nuclear genetic markers and are thus able to capture "*a genome-wide picture of population history*" [Niel00]. Furthermore, SNPs are advantageous over previous methods such as using *microsatellites*[18] for population history studies because they show a very favorable mutation pattern and greatly simplify the task of unbiased sampling of genetic variation [BBNE03]:

- *Mutation pattern:* Microsatellites have a mutation rate of $\sim 10^{-4}$ per generation as opposed to the rate of 10^{-8} displayed by SNPs. This makes multiple mutations for a single SNP unlikely, therefore only two alleles exist of most SNPs. Such a property greatly facilitates populational analysis—e.g., we will make use of it in the algorithmic inference of haplotypes from genotypes in Chapter 7. Furthermore, mutations in SNPs are more evenly distributed than in microsatellites, where the mutation rate is often hard to estimate [BBNE03].
- *Unbiased Sampling:* Due to their uniform mutation rates, SNPs may be selected at random in populational studies, avoiding previous bias that arose due to the fact that often, only loci with well known mutation rates would be chosen for analysis. Furthermore, [BBNE03] suggests that cross-species analysis of SNPs can provide greater insight into the natural occurring rate of genome-wide variation than biased loci such as microsatellites.

SNPs have been of great interest in populational studies due to the phenomenon that often there is a collection of SNP sites where the individual SNPs are not independent of each other; rather, a phenomenon called *linkage disequilibrium* is observed.[19] This phenomenon refers to the fact that often, haplotype combinations of alleles at different loci are correlated in their appearance frequency, forming a so-called *linkage disequilibrium block (LD block)* [DRSHL01]. The size of LD blocks is often debated and ranges from size suggestions of a few kilobases (in empirical research [Dunn00] and computer simulation experiments [Krug99]) to more than a hundred thousand kilobases [Abec01]. Independent of the discussed size, the presence of linkage disequilibrium is believed to reflect the fact that haplotypes descended

[18] Microsatellites are stretches of DNA consisting of tandem repeats of a small, simple sequence of nucleotides, e.g., $GCTGCTGCT \ldots GCT$. Often in literature, microsatellites are also referred to simple tandem repeats (STRs).

[19] For the human genome, linkage disequilibrium was studied, e.g., in [Reic01].

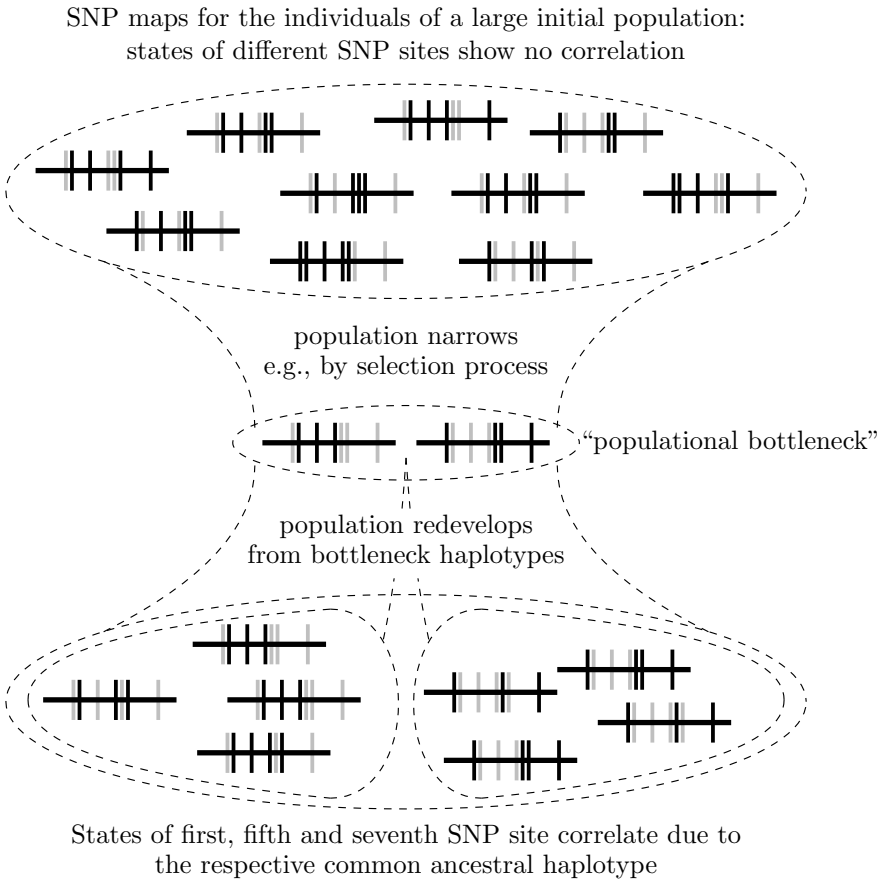


Figure 2.3: Development of linkage disequilibria in SNP sites: An initial, genetically diverse population is drastically reduced in its number of individuals (e.g., by selection processes in a unique environment). This causes a "populational bottleneck" where only a very few different haplotypes remain within a population. Redevelopment of a population based on this non-diverse genetic material causes linkage disequilibrium in the alleles, which decays over time due to mutations and recombination.

from common ancestral chromosomes; linkage disequilibrium may therefore also be an indicator for populational bottlenecks[20][Mu02]. The phenomenon of linkage disequilibrium relating to SNPs is illustrated in Figure 2.3.

Linkage disequilibrium of individuals' genes within a population "decays" with population history due to recombination [HaCl97]. It is believed that linkage disequilibrium around common alleles is a lot less frequent than around rare alleles, which are generally younger and thus less decayed by recombination [Watt77]. Using these assumptions, Reich *et al.* [Reic01] have shown that they can relate some linkage disequilibria to events such as the last glacial maximum 30 000-15 000 years ago, migration patterns in ancient Europe, or the dispersal of anatomically modern humans in Africa. The article mentions that for some

[20]A populational bottleneck is a period in population history where there are very few individuals in the population. These individuals then gave rise to the haplotypes found in a population today—conserved genetic patterns in the haplotypes can therefore be backwardly related to the respective ancestral individuals that lived during the bottleneck.

populations not as large as Europeans ([Reic01] mentions Yorubans as an example), the resolution of their linkage disequilibrium blocks is too coarse, but nevertheless—referring to studies such as [Tish96], [Tish00], or [Mate01]—*" [...] simultaneous assessment [of linkage disequilibria] at multiple regions of the genome provides an approach for studying history with potentially greater sensitivity to certain aspects of history than traditional methods [...]"* [Reic01]

Additionally to being able to gain deep biological insights into species development and population history, the study of linkage disequilibria and the search for common ancestors of species might also have high economical and health political impact. One example for this is the recent study of the malaria parasite *Plasmodium Falciparum* in [Mu02], this parasite has been of intense interest since it infects hundreds of millions of people each year, being responsible for almost 3 million annual deaths [Brem01].[21] An effective vaccine against malaria, for example, must trigger an immune response that is equivalent or superior to the one gained by contact with natural antigens. By finding some common SNP regions in different *Plasmodium Falciparum* populations, it is the hope of current research to build an accurate map of the ancestral relationship of various *Plasmodium Falciparum* strains. Such a map of ancestral relationships could help in identifying common antigens for immunizations [Gard98].

It was conjectured in [RLHA98] that the human malaria parasite experienced a populational bottleneck about 5000 years ago, further sparking the hope that it would be possible to find some common drug targets among malaria parasites. Although an extensive study of SNPs on the *Plasmodium Falciparum* genome carried out by Mu *et al.* [Mu02] have shown that this is probably not the case and *Plasmodium Falciparum* is rather a *"quite ancient and diverse"* population (with the most recent common ancestor being a few hundred thousand years old), it is still hoped that some more recent common ancestors of different strains can be found in order to obtain an assay of promising drug targets and vaccines:

> *"For the first time, a wealth of information is available [...] that comprise the life cycle of the malaria parasite, providing abundant opportunities for the study of [the ...] complex interactions that result in disease."* [Gard98]

Although the genetic sequence and the insights gained by SNPs alone are no cure for malaria and other widespread diseases, they seem to be a promising start.

The study of populational history based on traits of individuals (which—among others—may be the presence of highly correlated SNP sites) and its algorithmic tractability is studied in Chapter 5 of this work, where a special model of analysis called perfect phylogeny will be employed. As will be seen in Chapter 5, SNPs provide very good data for this model due to their very low mutation rate.

2.3.2 SNPs and Pharmacogenetics

The understanding of SNPs is believed to be a key to the research area known as *pharmacogenetics*. Using SNPs in pharmacogenetics is of immense economical interest to pharmaceutical companies. It has led to the founding and funding (hundreds of millions of US dollars) of the SNP Consortium [Hold02], a joint effort of major pharmaceutical companies such as Bayer, Bristol-Myers Squibb, Glaxo Wellcome, Aventis, Novartis, Pfizer, Roche, SmithKline

[21] 2001-2010 has been named the "Malaria Rollback Decade" by the WHO [WHO03] to emphasize efforts being made in limiting the widespread of malaria.

Beecham, and Zeneca. Interdisciplinary connections of the SNP Consortium include IBM and Motorola.

The problem with a lot of drug therapies is the possibility of adverse drug reactions by patients: Research by Lazarou, Pomeranz, and Corey [LPC98] suggests that, in 1994, such reactions were responsible for millions of hospitalizations and almost a hundred thousand deaths. This value is not likely to have improved lately and is hindering the introduction of new medications that are effective in most patients but pose unbearable risks: For example, the quite effective anticonvulsant drug Lamictal© by Glaxo Wellcome is only reluctantly prescribed because of a potentially fatal skin rash that arises as a side effect in five percent of all patients taking the drug [Maso99]. The problem of the different effects drugs exert on patients has long been known and studied, already over a hundred years ago Sir William Osler[22] reflected:

> *"If it were not for the great variability among individuals, medicine might as well be a science and not an art."* (as cited by [Rose00])

Pharmacogenetics is an area of research that studies how genetic variation influences a patients responsiveness and responses to drugs (a good introduction to pharmacogenetics is, e.g., [Rose00]), thereby trying to give physicians the possibility of using *objective* data about a patient's likeliness to react to prescribed drugs in a predictive way. The basic idea in pharmacogenetics is to build a *profile* of an individual's genetic variations in order to predict effectiveness and side-effects of drugs. As was discussed above—since genetic variation is mainly due to SNPs—a hope of pharmacogenetics relies on building an accurate map of SNP haplotypes.

Roughly speaking, the hope is to identify linkage disequilibrium loci around certain genes that are susceptible for causing a certain adverse reaction to drugs. The same technique has already been applied in the study of individuals' susceptibility to certain complex genetic diseases such as Alzheimer's disease: In an analysis of polymorphisms on the *ApoE* gene locus on chromosome 19, Martin *et al.* [Mart00] reported the detection of those SNPs in linkage equilibrium that are associated with Alzheimer's disease. A number of other studies successfully related the susceptibility for complex genetic diseases such as migraine with aura, psoriasis, and insulin-independent diabetes mellitus to certain SNPs in linkage disequilibrium [Rose00]. Now, just as linkage disequilibria can be related to the susceptibility for diseases, they can also be related to certain drug reactions. Two good examples for this are, e.g., patient's reactions to nortriptyline and beta-2-agonists:

- Nortriptyline is a medication against depression which is converted to an inactive compound within the body by drug metabolizing enzymes called the cytochrome P450 enzymes. Specifically, an enzyme labeled CYP2D6 is a key in inactivating nortryptiline and removing the inactivated substance from the body—except in some people that have variations in their CYP2D6 encoding gene. These variations may lead to two undesired effects [DeVa94]: People referred to as "ultra metabolizers" have a variation that causes the synthesis of too much CYP2D6 in their body, thus inactivating so much nortriptyline that these people are likely to receive insufficient antidepressant effects from nortriptyline. A more dangerous variation is found in people referred to as "poor metabolizers" who do not synthesize sufficient amounts of CYP2D6—these

[22] Osler, Sir William, Baronet (*1849, †1919). Canadian physician and professor of medicine who played a key role in the transformation of the curriculum of medical education in the late 19th and early 20th century.

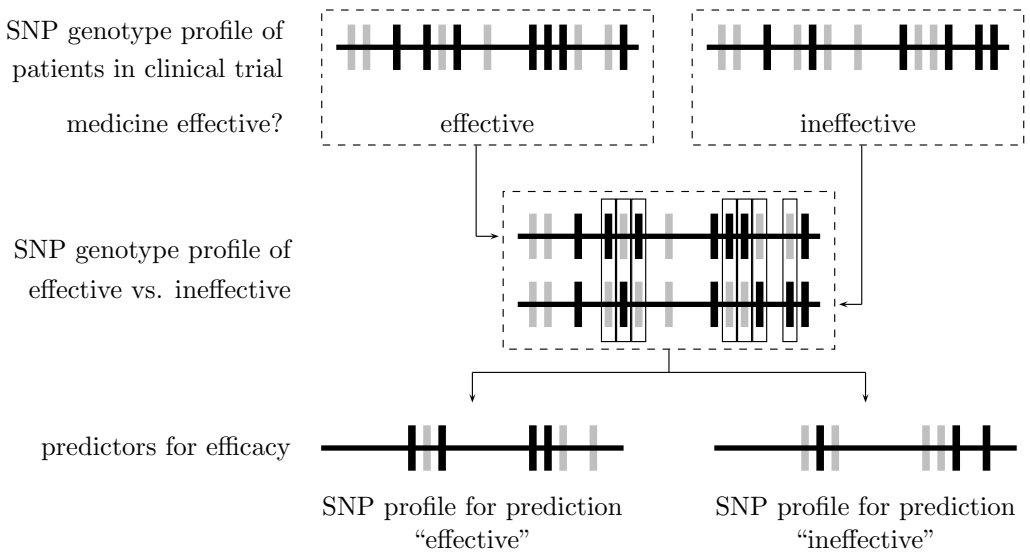


Figure 2.4: Profiling SNPs in pharmacogenetics: If there is a section of the SNP genotype profile that proves to be different in patients where a drug is effective as opposed to patients where a drug shows no efficacy or undesired side-effects, this region can be used to predict the effectiveness and potential risks due to side effects in a patient before a drug is prescribed.

are likely to experience toxic side effects due to accumulation of nortriptyline in their body. Genetic testing for variation in the gene for the CYP2D6 enzyme could avoid both scenarios.

- Beta-2-antagonists such as *albuterol* are important to the treatment of asthma. Interacting with the beta-2-adrenergic receptors in the lung, they cause the freeing of airways by inducing muscle relaxation in the lung muscles. A SNP in the gene encoding the beta-2-adrenergic receptor causes the carriers of one SNP variant to express fewer of these receptors, therefore receiving little relief of asthma symptoms upon a standard dose of albuterol [Ligg97]. Testing for presence of the specified SNP in patients can allow to clearly identify those 45% in the North American population[23] who can only poorly control their asthma by beta-2-antagonists.

It is clear that the prospects of being able to predict the efficacy of a drug whilst minimizing the risk of side-effects is of great interest to the pharmaceutical industry, which could then—as Roses proposes in [Rose00]—create efficacy profiles for patients (see Figure 2.4) already in phase II[24] clinical trials of medication. Abbreviated SNP profiles[25] could be used to record

[23]This figure should be similar for Europeans.

[24]Clinical trials are divided into five steps: *Preclinical Research* includes controlled experiments using a new substance in animals and test tubes and may take several years. *Phase I trials* are first tests of the investigated drug on humans. Doses are gradually increased to ensure safety. *Phase II trials* will gather information about the actual efficacy of a drug. *Phase III trials* studies a drugs effects with respect to gender, age, race, etc. A successful phase III trial leads to the admission of a drug to the public market. Occasionally, *phase IV trials* are conducted that are—in principle—phase III trials on an even broader variety of patients.

adverse reactions in patients and thus be able to predetect even the most rare adverse events.

Furthermore, the development of new, more effective drugs can be facilitated: The parallel developments of drugs targeting specific symptoms is facilitated because patients who do not respond to a certain medication can be profiled in early clinical trial stages. Additionally, the development of medications which are highly effective in only a comparably small part of a population (e.g., a medication with 30% response rate) become profitable as they may be specifically prescribed to patients to whom the respective substance will be effective.

Pharmacogenetics relying on SNP linkage analysis seems to be a promising start to replacing trial-and-error prescriptions with specifically targeted medical therapies.

[25] *Abbreviated SNP profiles* contain only the SNP information of a patient that is relevant for a drug efficacy prediction. The introduction of abbreviated profiles plays an important role in the discussion about the fear of "individual DNA profiling" because they cannot be backwardly related to a patient.

Chapter 3

Computer Science Preliminaries and Notation

The first section of this chapter introduces the notation used throughout this work, followed by a brief introduction to those ideas in computational complexity that are important to this work. Especially, the last section focuses on fixed-parameter tractability, laying a foundation for the computational complexity analysis in the following chapters.

3.1 Notation for Matrices and Graphs

Matrices. By an $n \times m$ matrix A we are referring to a rectangular arrangement of $n \cdot m$ elements into n rows and m columns. By a_{ij} we designate the element in A that may be found at the jth position of the ith row. We will use the terms A and (a_{ij}) synonymously.

Graphs. A graph consists of *vertices* and *edges*, where a vertex is an object with a name and other properties (such as a color) and an edge is the connection of two vertices. We will denote a graph G with n vertices and m edges by (V, E) where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\} \subseteq V \times V$. In this work, only *simple*, *undirected* graphs are considered, meaning

- a vertex cannot be connected to itself by an edge,
- no two vertices may be connected by more than one edge, and
- an edge leading from a vertex u to a vertex w also leads from w to u.

By the term *subgraph* $G' = (V', E')$ we are referring to a graph with $V' \subseteq V$ and $E' = E \cap (V' \times V')$. Given a set V', $G' = (V', E')$ with $E' = E \cap (V' \times V')$ is called the subgraph *induced* by V' in G.

A vertex v is said to have *degree* d—denoted by $\mathit{degree}(v) = d$—if there are exactly d edges in G that are adjacent to v.

A *path* p of *length* ℓ in a graph $G = (V, E)$ is a sequence of $\ell + 1$ distinct vertices $v_1 v_2 \ldots v_{\ell+1}$ in G such that for each $1 \leq i \leq \ell$, v_i and v_{i+1} are connected by an edge. A *cycle* of *length* ℓ in G is a sequence of ℓ vertices $v_1 v_2 \ldots v_\ell v_1$ in G such that $v_1 \ldots v_\ell$ is a path in G

and $\{v_\ell, v_1\} \in E$. We call $G = (V, E)$ a *tree* if it contains no cycles; a tree containing a specially designated node[1]—called the *root* of the tree—is called *rooted*. Nodes of degree 1 in a rooted tree are called leafs. A graph $G = (V, E)$ is called *connected* if any two vertices $u, v \in V$ are connected by a path in G. A subgraph of G that is maximally connected with respect to its number of vertices is called a *connected component*.

A graph $G = (V, E)$ is called *bipartite* if we can divide the set V of vertices into two disjoint subsets V_1 and V_2 such that E contains neither edges between vertices in V_1 nor edges between vertices in V_2. The graph G is called *planar* if it can be embedded into an (Euclidian) plane without any intersecting edges.

In this work—especially in Chapter 6—we will be using the following set of operations on graphs:

- **Subgraph removal.** Let $V' \subseteq V$ be a subgraph of $G = (V, E)$. By $G \setminus V'$ we will denote the subgraph that is induced in G by $V \setminus V'$.
- **Vertex deletion.** Let u be a vertex in a graph $G = (V, E)$. By $G \setminus \{u\}$, we denote the graph that is induced in G by $V \setminus \{u\}$.
- **Edge deletion.** Let e be an edge in a graph $G = (V, E)$. By $G \setminus \{e\}$, we denote the graph $G = (V, E')$ with $E' = E \setminus \{e\}$.

A *vertex separator* in a graph $G = (V, E)$ is a set $\mathcal{V} \subseteq V$ of vertices in G such that $G \setminus \mathcal{V}$ is not connected. If $|\mathcal{V}| = k$, we call $\mathcal{V}$ a *vertex separator of order* k. The definition of an *edge separator* $\mathcal{E} \subseteq E$ *of order* k is analogous.

3.2 Crash Course in Computational Complexity Theory

Generally speaking, *"an algorithm is a procedure to accomplish a specific task. It is the idea behind any computer program"* [Skie98]. The first goal for any algorithm is to be *effective*, i.e. providing correct solutions to a given problem, however, an effective algorithm is of little use if it is not *efficient*, i.e., requires more resources (especially time) to solve a problem than can be provided. Computational complexity theory deals with the amount of resources—the two most important of which are time and memory[2]—required to solve a certain computational problem by an algorithm. A brief introduction to analyzing the time complexity of algorithms is given in [Skie98], a very thorough treatment of complexity theory may be found, e.g., in [Papa94]. This section will introduce some basic terminology from computational complexity theory that will be used throughout this work.

3.2.1 Machine-Independent Analysis

Imagine that we are given an algorithm called $\mathcal{A}$ (in any programming language) that solves a certain problem $\mathcal{P}$ when given an input $\mathcal{I}$, called an *instance* of $\mathcal{P}$. We would now like to analyze the performance—especially concerning speed—of this algorithm. The most obvious way of this would be to run $\mathcal{A}$ on a lot of instances of $\mathcal{P}$ and measure the time it takes for $\mathcal{A}$

[1] In order to distinguish tree from graphs more easily throughout this work, we will use the term "vertex" for general graphs and the synonymous "node" for vertices in trees.

[2] Since only the notion of time complexity is important for this work, we shall omit space complexity in the following introduction.

to complete its task each time. However, with this approach, we quickly run into a multitude of problems, the most crucial of which are that the absolute time measured is influenced by the actual machines computer architecture[3], absolute time values are only useful for one particular type of machine, we can seldomly test the algorithm on all conceivable instances, and a purely practical analysis provides no indication about an algorithm's maximal (*worst-case*) running time.

Complexity theory tries to avoid these problems arising from a direct machine analysis by analyzing computational problems in a more formal and mathematical way. This analysis is machine-independent whilst still trying to incorporate the fundamental workings of a modern computer. Traditionally, complexity theory relies on the *Turing Machine* as its model of computation which is, however, a quite abstract model of computation lacking any close relationship to modern computers.[4] For this work, we do not require the many special features offered by Turing Machines and shall therefore rely on another model of computation that is sufficiently precise for our analysis, far more intuitive and more closely related to a "real" computer than a Turing Machine.[5] This model is the *RAM model of computation*, which Skiena [Skie98] describes quite vividly as a computer

> *"where each 'simple' operation (+,-,*,=,IF,call) takes exactly* 1 *time step [and] loops and subroutines are* not *considered simple operations. Instead, they are the composition of many single-step operations. ... Each memory access takes exactly one time step, and we have as much memory as we need."*

Using this model, the computational time of an algorithm is given by simply counting the number of time steps it takes the RAM machine to execute it on a given problem instance. The advantage of the RAM model lies in the fact that it captures the essential behavior of a modern computer without introducing any fickle parameters such as memory bandwidth, actual processor speed, and memory access time, just to name a few. The next subsection demonstrates the usage of this model in the analysis of an algorithm's time complexity. This, however, requires a last step of formalization: Besides the machine model, the term "problem" needs to be specified.[6]

Computational problems may be stated in many ways, the most important of which—at least in the context of this work—are *decision problems* (the output can be just either "YES" or "NO") and *optimization problems* (the output is a solution which is minimal/maximal in some respect). Most of computational complexity theory solely deals with *decision problems* because almost any "reasonable" way of stating a problem can be transformed into a decision problem.[7] Although a whole branch of theoretical informatics—*computability*—has evolved concerning the existence of decision problems that are *undecidable* (i.e., not algorithmically

[3] A modern computer's performance is, e.g., influenced by its processor, memory bandwidth, techniques such as pipelining and caching, the operating system, the programming language and its compiler, etc. Due to these many factors it is sometimes even difficult to obtain consistent results on a single, defined machine.

[4] There are many reasons why Turing Machines are nevertheless used in computational complexity theory: For example, requirements such as memory and time are very easy to define for a Turing Machine and can be analyzed with great accuracy. Furthermore, Turing Machines can simulate other machine models—one Turing Machine can, in theory, even simulate an infinite number of Turing Machines. The simulation of an algorithm on the RAM model (which will be introduced shortly) by a Turing Machine requires only polynomially more time than its execution on directly on the RAM (the term "polynomially more time" will also be defined more precisely later on in this chapter).

[5] A quite thorough analysis of different machine models can be found in Chapter 2 of [Papa94].

[6] An algorithm is by definition already given in a formal fashion.

[7] E.g., instead of asking "What are all prime numbers between 0 and 280581?" we can solve the decision problems "Is 1 a prime?" ("NO"), "Is 2 a prime?" ("YES"), ..., "Is 280580 a prime?" ("NO") separately.

solvable) by computers, we can assume for this work that we are always given *decidable* decision problems.

Analyzing decision problems is closely related to the fact that in complexity theory, a problem is generally formulated as a *language* $\mathcal{L}$ and asking (i.e., *deciding* by an algorithm) whether a given instance $\mathcal{I}$ is part of that language. Both the language $\mathcal{L}$ and the instance $\mathcal{I}$ are a subset of Σ^* for an *alphabet* Σ, where Σ is a finite set of *symbols* and Σ^* is the set of all *words* that may be generated by concatenation of symbols from Σ, including the *empty word* ϵ which contains no symbols at all.[8] Expressing a given problem as a language is—in most practical cases—quite straightforward.[9] For this work, the computational complexity for solving a problem can be seen as equivalent to the complexity of answering the corresponding decidability question. It should be noted that stating a problem in form of a language presents this particular problem in a very abstract form. Neither an algorithm for solving the problem is given nor any obvious hint about the time complexity of solving this problem. In order to deal with this, we will introduce the model of *complexity classes* and *reductions* later on in this chapter.

For the sake of simplifying the discussion in this work, we will refrain from stating problems in the form of a language. Instead, we will simply assume that the given problems may be stated in the form of a language. Furthermore, instead of asking whether an instance $\mathcal{I}$ is in $\mathcal{L}$, we shall directly deal the object x that $\mathcal{I}$ represents (such as a word, number, graph, etc.)[10]. We then call x an *instance* of the *problem* $\mathcal{P} =$ "$\mathcal{I} \in \mathcal{L}$?".

3.2.2 Running Time—Keeping Score

We discussed at the beginning of the last subsection that, given an algorithm for a problem $\mathcal{P}$, knowing how this algorithm performs on certain instances x of $\mathcal{P}$ is of little use. Rather, in order to understand the quality of an algorithm, it is vital to know how it performs on *any* conceivable instance x of $\mathcal{P}$ and express this performance in an intuitive way. This is done by introducing three new ideas: Analyzing how the running time of algorithms *scales* with the problem size, distinguishing between *worst*, *best* and *average-case complexity* (emphasizing on worst-case complexity), and analyzing the scaling of the algorithm in its *asymptotic behavior*.

The first idea is based on the intuitive observation that an algorithm should generally take longer time to run as the presented instance becomes larger. For instance, a graph problem on a general[11] graph consisting of ten vertices should be easier to solve than the same problem for a general graph with a thousand vertices. Therefore, if x is an instance of a problem $\mathcal{P}$, the running time t of an algorithm is expressed as a mathematical *function* f

[8] For example, if $\Sigma = \{0, 1\}$, $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$.

[9] E.g., the problem of deciding whether a given number is a prime number would have to be expressed as $\mathcal{L}_{prime} = \{p \in \{0,1\}^* \mid p \text{ is the binary representation of a prime number}\}$ (with $\Sigma = \{0, 1\}$) and then asking "given $\mathcal{I} \in \Sigma^*$, is $\mathcal{I} \in \mathcal{L}_{prime}$?".

[10] We shall assume for this work that such a representation, i.e, the *encoding* of x as $\mathcal{I}$ is always a valid one, meaning we do not need to worry about any cases where $\mathcal{I}$ does not encode a valid object x.

[11] By "general" we mean that the given graph has no special properties that greatly simplify the solution of the respective problem.

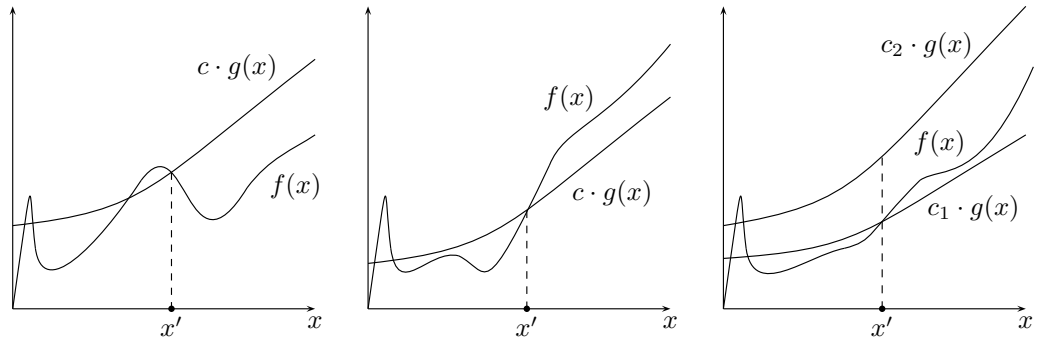


Figure 3.1: A function $f(x)$ and its bounds in O-notation: From left to right, $g(x)$ is an upper, lower and tight bound on $f(x)$. Note how the respective bounding property of g only needs to be true for all $x > x'$.

of $n := |x|$, the size of x:[12]

$$t_{algorithm}(x) = f(|x|) = f(n)$$

Given a fixed size n for the input instance x and an algorithm $\mathcal{A}$ that runs with x as an input, we distinguish between the *best-case*, *average-case*, and *worst-case* running time:

$$t_{best}(\mathcal{A}, n) = \min_{x\,:\,|x|=n} t_{\mathcal{A}}(x), \quad t_{avg}(\mathcal{A}, n) = \frac{\sum_{x\,:\,|x|=n} t_{\mathcal{A}}(x)}{|\{x \mid |x| = n\}|}, \text{ and } \quad t_{worst}(\mathcal{A}, n) = \max_{x\,:\,|x|=n} t_{\mathcal{A}}(x).$$

Most of the time, only the worst-case complexity of an algorithm is interesting since average-case and—especially—best-case complexity provide no information whatsoever about the running time that $\mathcal{A}$ might have when presented with any instance x. E.g., for average-time complexity the problem here lies in the definition of "average": There may be some problems which are rather easy to solve on many instances, but this is of no use if we should—consciously or not—be dealing just with hard instances during the application of the algorithm.[13] Albeit open to criticism about being too pessimistic, the worst-case complexity of an algorithm seems to be the most useful measure for its performance.

The computational RAM model introduced in the last subsection provided a way to measure the running time of a given algorithm $\mathcal{A}$ exact to a single time unit. This degree of accuracy is not useful as for such exact counts the function f that measures the running time of $\mathcal{A}$ will often get very complicated and unintuitive to analyze.[14] Moreover, as we are interested in the performance of an algorithm on a real-world machine and not on the hypothetical RAM, measuring the running time of $\mathcal{A}$ down to the last time unit finds no application. Therefore,

[12] Later on, we will analyze algorithms in more detail using various parameters of the input. For example, the running time of a graph algorithm may depend on the number of edges as well as on the number of vertices in the graph. The number of edges in the graph is lower than $|V|^2$, but explicitly using the number of edges provides a better analysis. In order to simplify the discussion, however, we will for now assume that there is just a single input size parameter given.

[13] Moreover, the corresponding mathematical analysis of average-case complexity even for simple algorithms and a clear definition of "average case" is often highly involved and complicated.

[14] Moreover, there are often trivial steps depending on the notation of the algorithm (such as initialization of variables) that require just a little constant amount of time and are thus not interesting for a general performance-analysis.

computational complexity theory, instead of directly analyzing f, rather analyzes the upper and lower bounds of f using the O-Notation (pronounced "Big Oh Notation"):

Definition 3.1 (O-Notation):
Given two functions $f : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$ and $g : \mathbb{R}_{>0} \to \mathbb{R}_{>0}$. We will say that

- *$f = O(g)$ if there exist a constant $c \in \mathbb{R}$ and an $x' \in \mathbb{R}$ such that for all $x > x'$, $f(x) \leq c \cdot g(x)$ (g is an* upper bound *for f).*
- *$f = \Omega(g)$ if there exist a constant $c \in \mathbb{R}$ and an $x' \in \mathbb{R}$ such that for all $x > x'$, $f(x) \geq c \cdot g(x)$ (g is a* lower bound *for f).*
- *$f = \Theta(g)$ if there exist two constants $c_1, c_2 \in \mathbb{R}$ with $c_1 \leq c_2$ and an $x' \in \mathbb{R}$ such that for all $x > x'$, $c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x)$ (g is a* tight bound *for f).*

This notation is illustrated by Figure 3.1.[15]

Analogously to preferring the worst-case complexity over the best- and average-case complexities when analyzing the performance of an algorithm, it is common practice to provide an upper bound for an algorithm's running time instead of a lower or tight one.[16]

To provide an example on how to determine the running time of an algorithm and express it in O-Notation, we will now analyze an algorithm for a well-known problem in computational complexity, called Vertex Cover.

Definition 3.2 (Vertex Cover)
Input: *A graph $G = (V, E)$ and a parameter k.*
Question: *Is it possible to choose a set $V' \subseteq V$ with $|V'| \leq k$ such that every edge in E has at least one endpoint in V'?*

In Figure 3.2, a graph and one of its vertex covers is given in order to illustrate this definition. A very trivial algorithm $\mathcal{A}_{\mathrm{VC}trivial}$ for this would be to simply try all possible solutions of size k and see whether one of these hypothetical solutions is indeed a vertex cover for the given graph:

Algorithm: Trivial algorithm $\mathcal{A}_{\mathrm{VC}trivial}$ for Vertex Cover
Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
Output: "Yes" if G has a vertex cover of size k, "No" otherwise

```
01  for every k-sized subset V' of V do
02      if V' is a vertex cover for G
03          return "Yes"
04  return "No"
```

[15] For a concrete example, consider the function $f(x) = x^4 + x^2 + x - \ln x + 1234$. If $x \geq 6$, we have

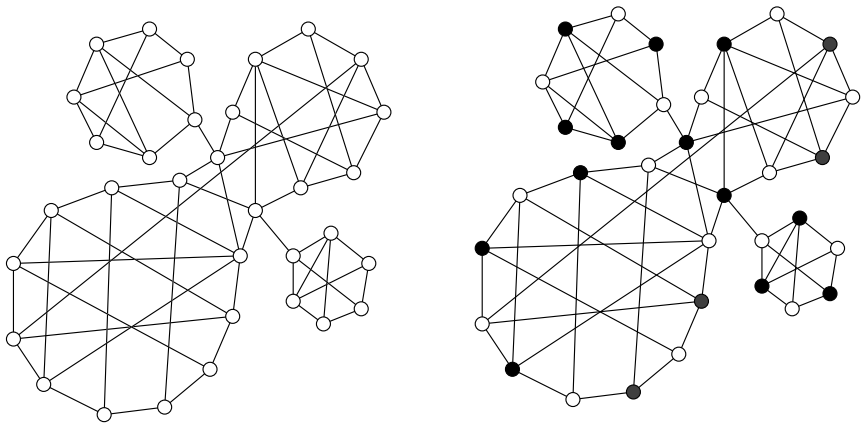
$$f(x) = x^4 + x^2 + x - \ln x + 1234 < x^4 + x^2 + x - \ln x + 6^4 < 5 \cdot x^4 =: 5 \cdot g(x),$$

$$f(x) = x^4 + x^2 + x - \ln x + 1234 > x^4 > 1 \cdot x^3 =: 1 \cdot g(x), \text{ and}$$

$$1 \cdot g(x) := 1 \cdot x^4 < f(x) = x^4 + x^2 - \ln x + 1234 < 5 \cdot x^4 = 5 \cdot g(x)$$

and therefore $f(x) = O(x^4)$, $f(x) = \Omega(x^3)$, and $f(x) = \Theta(x^4)$.

[16] This is mainly due to the fact that tight bounds on the running time of algorithms are often neither intuitive nor easy to grasp mathematically.



A graph $G = (V, E)$

A VERTEX COVER for G (black vertices)

Figure 3.2: A graph $G = (V, E)$ and a vertex cover of size 17 (black vertices) for G. Note how for each edge in G, at least one of its endpoints is in the given vertex cover. The shown vertex cover for G is optimal in the sense that there is no vertex cover for G with fewer than 17 vertices (this was verified using a computer program).

We will now analyze the running time of $\mathcal{A}_{\text{VC}trivial}$ in terms of the number of vertices ($|V|$) and the number of edges ($|E|$) in G.[17] Let us start with lines *03* and *04*: Since both terminate $\mathcal{A}_{\text{VC}trivial}$, they are executed at most once, and thus do not play a role in the asymptotic running time of $\mathcal{A}_{\text{VC}trivial}$. Line *02* can be executed by calling the following subroutine: Iterate over all edges of G, and check for every edge whether at least one of its endpoints is in V'. If we have been clever enough and marked those vertices that are in V' during the execution of line *01*, executing this line only requires $O(|E|)$ running time. For the seemingly most difficult line to analyze, line *01*, we make use of the machine-independency of our analysis by using an algorithm for generating subsets from the extensive available literature on algorithms (e.g., [CLRS01], [Knut97]).[18] In [Knut03], we can find an algorithm that generates all k-sized subsets V' of V in

$$O(\binom{|V|}{k}) = O\left(\frac{|V|!}{k!(|V|-k)!}\right) = O\left(\frac{1}{k!}\underbrace{|V| \cdot (|V|-1) \cdots (|V|-k+1)}_{k \text{ factors}}\right) = O(|V|^k)$$

time on a RAM-like machine. For finding the total running time of $\mathcal{A}_{\text{VC}trivial}$, it is now sufficient to observe that line *01* causes line *02* to be executed once for each of the at most $\binom{|V|}{k}$ subsets generated. Taking into account the time requirements of line *02*, the total running

[17] A quick glance at the algorithm demonstrates the advantage of all the conventions we have introduced above. E.g., if we were not to use the O-Notation for the worst-case bound we are about to determine, we would explicitly have to look at the exact number of steps a RAM needs to generate a subset in line *01*, to store G, to determine whether V' is a vertex cover of G, and so on.

[18] Note that the RAM models in literature do not necessarily need to be defined precisely the way we have. E.g., in the RAM model used in [Knut97], some computing steps take more than one unit of time. However, this is not important for the performance of an algorithm in O-notation: Assume, for example, that each simple computational operation would consume four time units instead of one on a machine RAM$'$ as opposed to our RAM model. If there is an algorithm that, e.g., requires $O(n^3)$ time on the RAM, it would require $O((4n)^3) = O(64n^3) = O(n^3)$ time on the RAM$'$.

time for $\mathcal{A}_{\mathrm{VC}trivial}$ is therefore bounded by

$$O(|V|^k \cdot |E|).$$

This upper bound is quite unsatisfactory for practical applications, for it implies an enormous worst-case running time even for small graphs and small k.[19] Note that from the discussion so far, it is not clear wether this is due to a poorly designed algorithm we have come up with or it is a result of some "inherent complexity" of VERTEX COVER. The next subsection and the following section will demonstrate that actually both is true, that is, VERTEX COVER is believed to be "hard to solve" (we will define this more precisely in the next subsection) but there are ways of "taming" this inherent complexity, as will be shown in Section 3.3.

3.2.3 Complexity Classes

In the previous subsection, we have given an algorithm to solve VERTEX COVER that was quite impractical for large input graphs. However, it was not clear whether this problem is hard to solve in general or if we just haven't come up with a good algorithm. We would now like to know wether there is a better algorithm for VERTEX COVER than the one presented, or—even better—know the fastest possible algorithm for VERTEX COVER (i.e., the minimum time complexity of VERTEX COVER). The first request is *comparably* easy to come by, we just have to look for an algorithm with a better worst-case running time than the one presented. The latter however is a lot harder to deal with, because in finding a *lower bound* for the time complexity of VERTEX COVER it is necessary to consider *every thinkable algorithm* for VERTEX COVER—even algorithms that have not yet been found.[20] However, there is another way to approach the problem of complexity bounds using *reductions* and *complexity classes*.

Reductions will allow us to divide problems into different "classes of difficulty". The idea behind this is the following: Although not knowing how hard an individual problem might be, we can relate problems to each other so that we know they are both "equally hard" to solve, meaning if there is a fast algorithm for one problem, there must be one for the other problem, too. A collection of such related problems is called a *complexity class* (a more formal definition will follow shortly). Problems are grouped together in complexity classes by finding a computationally "cheap"[21] transformation between instances of one problem and the other. Then, loosely speaking, if we know that if one of the two problems turns out to be easy to solve, we also know that the second problem is easy to solve, because we can apply the algorithm for the easy problem to transformed instances of the other one. In a more formal fashion:

Definition 3.3 (POLYNOMIAL TIME REDUCTION):
Given two languages $\mathcal{L}_1 \subseteq \Sigma_1^*$ and $\mathcal{L}_2 \subseteq \Sigma_2^*$. We call $\mathcal{L}_1 \subseteq \Sigma_1^*$ polynomial-time reducible *to $\mathcal{L}_2 \subseteq \Sigma_2^*$ (designated $\mathcal{L}_1 \leq_{poly} \mathcal{L}_2$) if there is a function $\mathcal{R}$ from Σ_1^* to Σ_2^* that can be computed in polynomial time on any $x \in \Sigma_1^*$ and*

$$x \in \mathcal{L}_1 \Leftrightarrow \mathcal{R}(x) \in \mathcal{L}_2.$$

[19] For example, finding out if a graph with 75 vertices and 200 edges has a vertex cover of size 10 would require $c \cdot 10^{21}$ steps on our RAM where c is some constant ≥ 1 omitted on the O-notation.

[20] Except for a few very rare cases of problems (such as sorting), the question of lower complexity bounds therefore generally remains unanswered.

[21] In our context, this will imply a polynomial running time with respect to the original instance's size.

In complexity theory, there are a lot of more specialized reductions, some of which we will get to know in Section 3.3, that are more "delicate" in the sense that they impose stricter requirements on $\mathcal{R}$ than just being computable in polynomial time. However, we shall work just with polynomial time reductions for the rest of this section.

The concept of polynomial time reduction may be used to build a hierarchy of computational problems. This hierarchy consists of classes $\mathcal{C}$. In each class, we can find those problems that are solvable using the resources allowed by the respective class. Furthermore, we introduce the concept of *completeness* to identify those problems in a class that are computationally as hard to solve as any other problem in that class. In this way, if a problem that is complete for a class should prove to be "easy" to solve, we know the same to be true for all other problems that are in $\mathcal{C}$.

Definition 3.4 (Complexity Class Hardness and Completeness):
Let $\mathcal{C}$ be a complexity class. A language $\mathcal{L}$ is called $\mathcal{C}$-hard *if all languages in $\mathcal{C}$ can be reduced in polynomial time to $\mathcal{L}$. We call $\mathcal{L}$* $\mathcal{C}$-complete*, if $\mathcal{L}$ is $\mathcal{C}$-hard and in $\mathcal{C}$.*

There is a vast number of complexity classes known today (see, for example, [Aaro03]), each of them grouping together problems with various properties. Two of the first classes that were developed and are of much interest for this work are *P* and *NP*.

Definition 3.5 (P and NP):
The complexity class P is the class of computational problems that can be solved in polynomial time on a deterministic Turing Machine.
The complexity class NP is the class of computational problems that can be solved in polynomial time on a non*deterministic Turing Machine.*

Although our definition uses the term "polynomial" to describe all problems in *NP*, it is widely believed that all *NP*-complete problems are only solvable in exponential time. The reason for this is the computational model underlying the definition: A nondeterministic Turing Machine is a very unrealistic model of computation, being able to—vaguely speaking—correctly "guess" the solution to a problem and then only needing to verify its correctness (the process of checking must then be done in polynomial time). However, all computers known today are deterministic, and therefore they have to "emulate" the guessing steps of the nondeterministic Turing Machine in order to find a solution to an *NP*-complete problem. This emulation is done by simply checking all possibilities for a possible solution which takes—in worst-case complexity—an exponential amount of time.

It must be stressed that *no proof whatsoever* has been given that problems in *NP* are at best solvable in exponential time. All we have stated is a plausibility argument: There are thousands of problems known to be *NP*-complete (even the rather outdated list of [GaJo79] lists hundreds of *NP*-complete problems), finding a polynomial time algorithm for *just one* *NP*-complete problem would show that *all* *NP*-complete problems are polynomially solvable, but this has not happened in spite of over 25 years of research so far. There are therefore two important things to be remembered throughout this work:

- When saying that a problem is harder than another one, we are always referring to *relative* complexity bounds, i.e., we are saying that if a "harder" problem should turn out to be efficiently solvable, so will the easier problem (but not vice-versa).

- Sometimes we will use an argument such as "...—since this would imply that $P = NP$" in our proofs, which is based on the unlikeliness of $P = NP$. It would, however, be more correct—albeit unusual—to write "Unless $P = NP$, the following holds true: ...".

The VERTEX COVER problem posed at the beginning of the previous subsection has been proven to be *NP*-complete in [GaJo79], where it is shown that VERTEX COVER is even *NP*-complete for planar graphs where each vertex has a degree of at most 3[22]. A long time, an *NP*-completeness proof for a problem was taken as a synonym for "unsolvable already for moderate input sizes" (coining the term "intractable"). However, this is not true in general, as the next section demonstrates.

3.3 Fixed-Parameter Tractability (FPT)

We have seen in the previous section how algorithms can be analyzed machine-independently by observing how they scale with the size of their respective input. This size we named n. We have also seen the class *NP*, reasoning that problems complete for this class most probably have a worst-case running time that is exponential, i.e., an *NP*-complete problem can only be solved in

$$\Omega(a^n)$$

time for some $a > 1$. Since this usually implies unreasonably high running times for large n, problems that are *NP*-hard are also referred to as being *intractable*. We have also stated in the last section—citing from [GaJo79]—that the problem VERTEX COVER (see Definition 3.2) is *NP*-complete. The *NP*-completeness of VERTEX COVER implies that it is most probably only solvable in $O(a^n)$ time where n is the size of the input instance and a is some constant. However, this definition provides us with two loopholes:

- We have made no statement about the size of a. A small a could lead to algorithms that are fast enough even for a fairly large n.
- We have made no good use of the fact that—besides the size of the input graph—any instance of VERTEX COVER contains a parameter k that might be restricted to a small value. What if we could restrict the exponential complexity of VERTEX COVER to the parameter k which is given along with the input graph according to Definition 3.2 ?

It might seem at first that observing these two "loopholes" is just splitting hairs in an imprecise definition, but in this section, we shall use exactly them to develop a more efficient algorithm for VERTEX COVER than the trivial $O(|V|^k \cdot |E|)$ algorithm used as an example for complexity analysis in the last section. After that, a short introduction to parameterized complexity theory is given.

3.3.1 An Efficient Algorithm for VERTEX COVER

At the end of this section, we will have improved the $O(|V|^k \cdot |E|)$ algorithm for VERTEX COVER given in the previous section to an $O(2^k \cdot |E|)$ algorithm. The strategy for this will be quite straightforward.

[22] Recall the definitions of "planar" and "degree" from Section 3.1.

Recall Definition 3.2. If a given graph G has a vertex cover of size k, then we can choose k vertices in G such that *every edge* in G includes *at least one vertex from the cover*. This means, if we were to search for a vertex cover V' for G we can simply pick any edge $e = \{v_a, v_b\}$ in G, and then, knowing that either v_a or v_b *must* be in the vertex cover, consider two distinct cases for V': Either V' contains v_a or V' contains v_b. For each of these cases, we would then look at the uncovered edges, pick one, and again consider the two cases for putting a vertex of that edge into V' (the common term for this is to *branch* into those two cases). This *recursive* algorithm leads to a tree-like structure searching for vertex covers of size k for G—depicted in Figure 3.3—that is commonly referred to as a *search tree*. Note that for each level down the search tree, we have one vertex less left to form a vertex cover for G. If we cannot find a vertex cover for G in the kth level of the search tree, then, as we have tried all possibilities of a vertex cover for G, G has no vertex cover of size k.

The described algorithm can be rewritten in a more formal fashion:

Algorithm: search tree algorithm $\mathcal{A}_{tree}$ for Vertex Cover
Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and a parameter k
Output: "Yes" if G has a vertex cover of size k, "No" otherwise

```
01 if G contains no edges then
02     return "Yes"
03 if G contains edges and k = 0 then
04     return "No"
05 pick an edge e = {v_a, v_b} from G
06 V'_a ← V \ {v_a}
07 E'_a ← E \ {e ∈ E | v_a is an endpoint of e}
08 if A_tree with G'_a := (V'_a, E'_a) and k − 1 as inputs returns "Yes" then
09     return "Yes"
10 V'_b ← V \ {v_b}
11 E'_b ← E \ {e ∈ E | v_b is an endpoint of e}
12 if A_tree with G'_b := (V'_b, E'_b) and k − 1 as inputs returns "Yes" then
13     return "Yes"
14 return "No"
```

This algorithm is illustrated in Figure 3.3. So what is the running time of this algorithm? Without the recursion (i.e., calling $\mathcal{A}_{tree}$ as a subprocedure), $\mathcal{A}_{tree}$ would require $O(|E|)$ time to generate G'_a and G'_b, since each edge must be looked at to see if it is adjacent to v_a or v_b, respectively (we shall assume that deleting a vertex from G takes constant time). The algorithm $\mathcal{A}_{tree}$ calls $\mathcal{A}_{tree}$ (with a different input, especially, k is decreased by one) at most two times. Each of those calls again calls $\mathcal{A}_{tree}$ at most two times, and so on, until the algorithm is called with $k = 0$. This means, in a worst-case analysis, $\mathcal{A}_{tree}$ is called

$$\underbrace{2}_{\text{initial } k} \cdot \underbrace{2}_{k-1} \cdot \underbrace{2}_{k-2} \cdots \underbrace{2}_{k-k+1} \cdot \underbrace{1}_{k-k=0} = 2^k$$

times. Each call itself takes—as mentioned above—$O(|E|)$ time which means in total, $\mathcal{A}_{tree}$ requires at most

$$O(2^k|E|)$$

time to solve a given instance (G, k) of Vertex Cover, a fairly large improvement compared to the trivial algorithm proposed in the last section, and moreover, the exponential part in the running time of $\mathcal{A}_{tree}$ is independent of the size of G. This makes Vertex Cover a

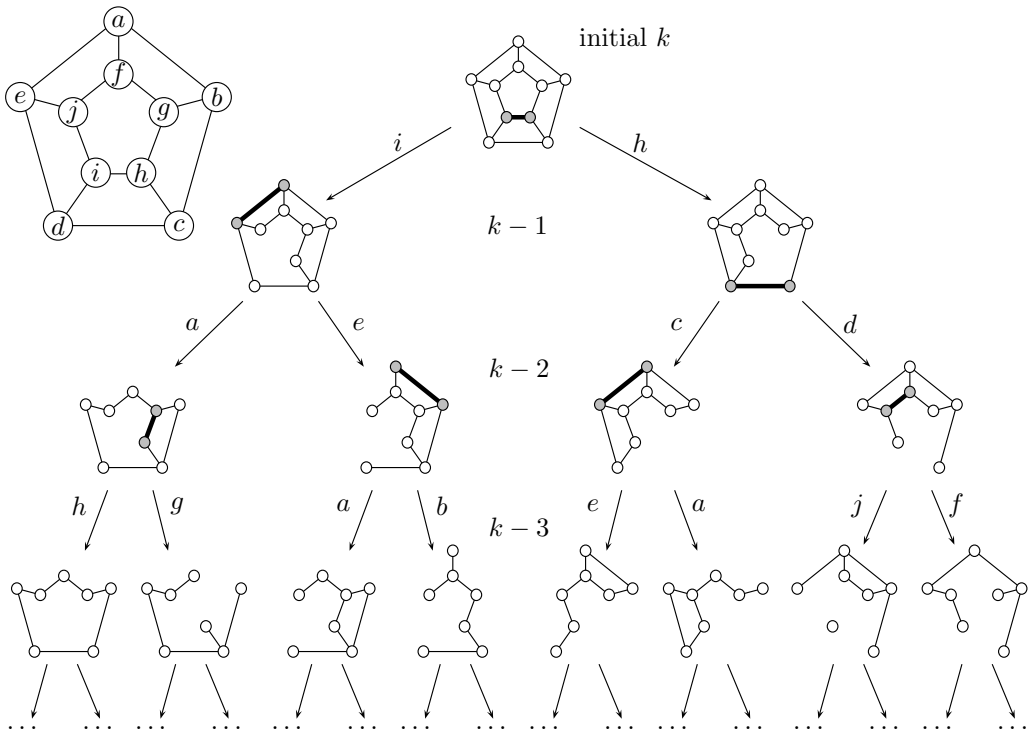


Figure 3.3: The search tree for finding a vertex cover of size k for a given graph: Given the graph G and a parameter $k > 3$, the above figure demonstrates how a search tree algorithm would try to find a vertex cover of size k for G. In each node of the search tree, an edge $e = \{u, v\}$ that is not yet covered is chosen from G (designated by coloring the adjacent vertices of e grey) and the algorithm branches into two cases: Either, u is in the vertex cover or v. The respective vertex from G is chosen into the vertex cover (and can then, along with its adjacent edges which are now covered, be removed from G), k decreased, and the algorithm proceeds if no vertex cover for G has been found yet and we have not yet chosen k vertices into the cover.

fixed-parameter tractable problem, because, as long as k is constant, the time required to solve Vertex Cover on (G, k) using $\mathcal{A}_{tree}$ is polynomial (for Vertex Cover, even linear) with respect to the size of the input graph G.

Note that $\mathcal{A}_{tree}$ is not the optimal fixed-parameter algorithm for Vertex Cover known today. In [CKJ01] and [NiRo03$_b$], $O(1.29^k)$-algorithms for solving Vertex Cover are given. This is done by optimizing the search tree: Instead of branching into two subcases and decreasing k by one each time the algorithm is called recursively, the algorithm may branch into more complex cases, allowing it to decrease k by more than 1 in some branches of the tree. Using the mathematical tool of *recursion analysis*, it can be analyzed how these complex cases decrease the base of the exponent in the algorithm. It should furthermore be noted that the algorithm uses the technique of *problem kernel reduction*[23] on Vertex

[23]Kernel reductions are based on the idea that using the parameter k, we can already decide for some parts of the input instance how they will add to the solution of the problem. A problem kernel reduction causes the input instance to be smaller than $f(k)$ for some f whilst being computable in polynomial time.

COVER: Loosely speaking, for some vertices in a given graph G, one can, given the parameter k, determine that they necessarily have to be in a vertex cover of size k—should one exist—for G and, before recursing, already choose these vertices into the vertex cover, decreasing k. A third technique called *interleaving* first introduced in [NiRo00] also applies problem kernel reduction during the recursion to decrease k even more in various branches of the search tree.

Unfortunately, the concept of fixed-parameter tractability is believed to be only applicable to a portion of *NP*-complete problems, which we will point out in the next subsection, after introducing some formalisms of fixed-parameter tractability.

3.3.2 Formal Definition and Aspects of FPT

It is obvious that restricting the exponential complexity of an *NP*-complete problem to a parameter k can only be done for those problems where k is given.[24] We call such a problem where a parameter k is given a *parameterized problem.*

Definition 3.6 (PARAMETERIZED PROBLEM):
A parameterized problem *is a language* $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$. *For every* $(x, k) \in \mathcal{L}$, *we call* k *the* parameter.

As we have already mentioned several times, we want to "restrict the complexity" of a problem $\mathcal{P}$ in order for the "hard part of $\mathcal{P}$" to be only dependent on k. In the context of parameterized complexity this means that we want to have an algorithm for a problem whose running time grows no more than polynomially[25] with the input size $|x|$.

Definition 3.7 (FIXED-PARAMETER TRACTABILITY):
A parameterized problem $\mathcal{L}$ *is called* fixed-parameter tractable *if there exists an algorithm that solves the decision problem* $\mathcal{P} =$ "$(x, k) \in \mathcal{L}?$" *in*

$$f(k) \cdot |x|^{O(1)}$$

time where $f(k)$ *is an arbitrary function solely dependent on* k.

When introducing the complexity class *NP*, a *polynomial time reduction* was used for reducing problems to each other in order to show their relative hardness to each other. Recall that we were not able to make any *absolute* statement about the time resources that (deterministic) algorithms require for solving *NP*-complete problems; instead we just showed that if one *NP*-complete problem is solvable in polynomial time, so are all *NP*-complete problems. Analogously, we are now seeking for a reduction that allows for relative complexity statements such as "if problem $\mathcal{L}_1$ is fixed-parameter tractable, so is problem $\mathcal{L}_2$". We would therefore like to find a reduction from an instance (x, k) of one parameterized problem to another parameterized problem that preserves some properties concerning the parameter;

It can be shown that for every fixed-parameter tractable problem, there exists a kernel reduction [DoFe99].

[24] It should be noted that the applications of parameterized complexity are not restricted in any way to *NP* or any other complexity class. However, since this work will solely deal with fixed-parameter tractability in the context of *NP*-complete problems, we will sometimes restrict our discussion to such problems for reasons of simplification.

[25] Although higher degree polynomials such as $|x|^{1000}$ would cause an impractically high running time, the term "polynomial" is often used synonymously with "efficient" because empirically, almost every polynomially solvable problem has shown to be solvable in $O(n^3)$ time or faster.

i.e., if the reduction is to make statements about the fixed-parameter tractability depending on the fixed-parameter tractability of the problem we perform the reduction from, it is clear that besides being computable in polynomial time with respect to the input size, this reduction must also keep the parameter of the instance yielded by the reduction independent from the size of x.

Definition 3.8 (Parameterized Reduction):
A parameterized problem $\mathcal{L}_1 \subseteq \Sigma_1^* \times \mathbb{N}$ *is said to be* fixed-parameter reducible *to another parameterized problem* $\mathcal{L}_2 \subseteq \Sigma_2^* \times \mathbb{N}$ *if there exist computable functions* $\psi : \mathbb{N} \to \mathbb{N}, k \mapsto k'$ *and* $\Phi : \Sigma_1^* \times \mathbb{N} \to \Sigma_2^*, (x,k) \mapsto x'$ *such that*

1. *for some function* f, Φ *is computable in time* $f(k) \cdot |x|^{O(1)}$, *and*
2. $(x,k) \in \mathcal{L}_1 \Leftrightarrow (x',k') \in \mathcal{L}_2$.

If $\mathcal{L}_1$ *is fixed-parameter reducible to* $\mathcal{L}_2$, *we write* $\mathcal{L}_1 \leq_\Psi \mathcal{L}_2$.

Using this reduction, Downey and Fellows developed a theory of computational complexity classes for classifying parameterized problems in [DoFe99], thereby introducing the classes

$$FPT \text{ and } W[1],\ W[2],\ \ldots,\ W[P]$$

where

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[P]$$

and P is some polynomial. Analogous to the "*P=NP*" problem, there is no proof whether these inclusions are strict or there is an $i \geq 1$ for which $FPT{=}W[i]$. For the purposes of this work, it will be sufficient to know that it is widely believed that $FPT{\neq}W[1]$ and that problems in the W-classes—as opposed to those in FPT—are not fixed-parameter tractable (e.g., [ADF95] and [CaJu01] provide some strong indications to this). Again, as in the case with *NP*-complete problems, the fixed-parameter *in*tractability of *W[1]*-complete problems has not been proven but is extremely likely to be true due to some consequences that would arise if this were not the case.

The above definition of a parameterized reduction—which closely follows the one given in [DoFe99]—may not be useful for practical applications. The reason for this is that k' may be impractically large compared to k, and Φ may have high demands in computational time. For example, the definition would technically allow for

$$k' = 10^{10^{10^k}} \text{ and } f(k) \cdot |x|^{O(1)} = k^{k^{k^k}} \cdot |x|^{O(1)}.$$

This might still be interesting from a theorist's point of view as it does not contradict the fact that even using these values for k' and the computational time of Φ means that if $\mathcal{L}_1$ is fixed-parameter tractable, so is $\mathcal{L}_2$. But bearing in mind that we introduced the whole concept of fixed-parameter tractability with the goal of developing efficient algorithms for hard problems, the definition of a parameterized reduction clearly needs refinement. We would rather desire a reduction that tells us that if $\mathcal{L}_1$ is fixed-parameter tractable and can therefore be dealt with by an *efficient* algorithm, the same holds true for $\mathcal{L}_2$. Using "computable in polynomial time" synonymous for "efficient" (as was already explained above), we arrive at the following definition:

Definition 3.9 (Parameter-Preserving Reduction):
Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be parameterized problems. We call $\mathcal{L}_1$ parameter-preserving reducible *to $\mathcal{L}_2$ if there is a parameterized reduction from $\mathcal{L}_1$ to $\mathcal{L}_2$ that*

1. *is computable in time $k^{O(1)} \cdot |x|^{O(1)}$ and*
2. *preserves the parameter k, that is $k = k'$.*

If $\mathcal{L}_1$ is parameter-preserving reducible to $\mathcal{L}_2$, we write $\mathcal{L}_1 \leq_{id} \mathcal{L}_2$.

Definition 3.10 (Parameter-Equivalence):
Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be parameterized problems. We will call $\mathcal{L}_1$ and $\mathcal{L}_2$ parameter-equivalent *if $\mathcal{L}_1 \leq_{id} \mathcal{L}_2$ and $\mathcal{L}_2 \leq_{id} \mathcal{L}_1$.*

Parameter-equivalence between problems expresses a strong linkage that easily allows the transfer of approximation and exact (fixed-parameter) results from one problem to its parameter-equivalents. We will use this definition to show some close links between two problems called Row Deletion and d-Hitting Set in the next chapter.

Concluding this chapter, it should be noted that parameterized complexity is an ongoing field of research with still many open problems to explore— *"it will need many people to join this fixed-parameter track."* [Nied02]

Chapter 4

Submatrix Removal Problems

Motivated by the problems analyzed in the next chapter, this chapter analyzes the computational complexity of the following problem called Row Deletion(B): Given a matrix A, avoid any permutation of a small matrix B (called the "forbidden submatrix") to occur in A by removing the smallest possible number of rows from A. This problem will play an important role in the next chapter, where it will be shown that the construction of a so-called *perfect phylogeny* (a model of evolutionary development) can only be constructed for a set of species if a matrix representing the species' characters avoids the induction of the matrix "Σ"[1].

In this chapter, we demonstrate a very close linkage between Row Deletion(B) and a problem called d-Hitting Set (to be defined in Section 4.1). Section 4.2 shows that all occurrences of a given forbidden submatrix in a larger matrix can be found in polynomial time, leading to a direct reduction from Row Deletion(B) to d-Hitting Set where d is determined solely by B. Section 4.3 analyzes the converse reduction from d-Hitting Set to Row Deletion(B) in order to determine a lower computational complexity bound for Row Deletion(B), characterizing a set of forbidden submatrices for which Row Deletion(B) is *NP*-complete (see overview of results in Subsection 4.3.1). Some ideas for extending the framework of Section 4.3 conclude this chapter in Section 4.4.

4.1 Definitions and Terminology

The problems in this chapter will consider finite matrices with entries from a finite alphabet $\mathcal{A}$. W.l.o.g., we consider all matrices in this chapter to contain entries from the alphabet $\mathcal{A} = \{0, \ldots, \ell - 1\}$.[2] We shall call such a matrix *ℓ-ary*. A matrix A will be referred to as a *permutation* of a matrix A' when A and A' are have the same size and A can be transformed into A' by a (finite) series of row- and column-swappings. This allows for the following definition:

Definition 4.1 (Induction of B, B-Freeness):
Let A be an $n \times m$ matrix and B be an $r \times s$ matrix with $1 \leq r \leq n$ and $1 \leq s \leq m$. We

[1] The naming for $\Sigma := \left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ is explained in more detail in the next chapter.

[2] We assume $\ell > 1$, since the problem would otherwise be trivial.

will say that A induces *B if A contains r rows and s columns such that the corresponding submatrix of A is a permutation of B. We refer to A as being B-*free *if A does not induce B.*

With this terminology a formal definition of the general SUBMATRIX OCCURRENCE problem is obtained.

Definition 4.2 (SUBMATRIX OCCURRENCE *Problem*)
Input: *An $n \times m$ matrix A and an $r \times s$ matrix B ($1 \leq r \leq n$ and $1 \leq s \leq m$).*
Question: *Does A induce B?*

Removing data from a matrix A in order to avoid the induction of B may be done in three ways: By deleting rows, columns, or both from A. Removing rows from a matrix A so that it becomes B-free leads to the definition of the following problem:[3]

Definition 4.3 (ROW DELETION(B) *Problem*)
Input: *An matrix A and a parameter k.*
Question: *Is it possible to delete at most k rows in A such that the resulting matrix does not induce B?*

This chapter will not explicitly analyze reductions and parameterized relationships for the analogously definable COLUMN DELETION(B) problem since all results that we obtain for ROW DELETION(B) are easily transferred to COLUMN DELETION(B) due to reasons of symmetry.

Allowing for both row and column deletion in a matrix A to avoid a forbidden submatrix B is also not considered due to the application in Chapter 5 for which the results from this chapter are developed: In Chapter 5, it will be shown that being able to construct a tree depicting the evolutionary relationship of species depends on avoiding the forbidden submatrix "Σ" (for more details, see Definition 5.5) mentioned on page 31 in a matrix A that represents some biological properties of the species. Removing rows from A will correspond to removing species, removing columns will correspond to removing characteristics. In order to be able to construct such a tree, allowing both row- and column-deletion ("ROW AND COLUMN DELETION(B)") seems to be too powerful if the results are supposed to have a biological meaning. For example, consider the matrices

$$A := \begin{pmatrix} 1&1&1&1&1&1&1 \\ 1&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1 \\ 0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&1 \\ 0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&1 \\ 0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&1 \\ 0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&1 \\ 0&0&0&0&0&1&0 \end{pmatrix} \quad \text{and} \quad A' := \begin{pmatrix} 1&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&1&0&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&1&0 \\ 0&0&0&0&0&1 \\ 0&0&0&0&0&1 \end{pmatrix}$$

where A' is obtained from A by deleting the first row and last column. Note that A' does not induce Σ whilst A does so many times. If we were trying to delete just rows from A in order to make this matrix Σ-free, we would see that at least seven rows (e.g., all those that contain a 1 in the last column) are necessary to achieve this. Analogously, for COLUMN

[3] A variant to the forbidden submatrix problems presented and analyzed in this chapter is discussed in detail in [KRW95]. In [KRW95], the problem is to avoid a *fixed permutation* of the forbidden submatrix B to be induced in a larger matrix A by permuting the rows of A. This problem is proven to be *NP*-complete in general.

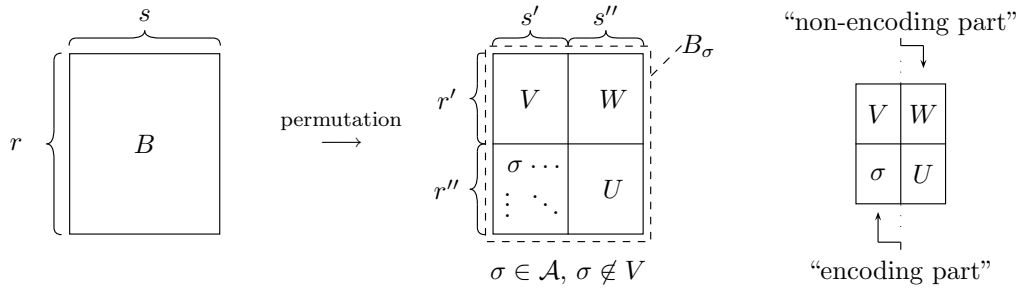


Figure 4.1: General scheme for the σ-decomposition of a matrix B over the alphabet $\mathcal{A}$.

DELETION(B), we would require the deletion of at least five columns from A. As we can see from this example, allowing freely for the deletion of rows and columns in a matrix might produce very "cheap" solutions (in terms of the total number of rows and columns deleted) and thus force the data to fit the evolutionary model proposed although it originally does not comply with this model at all. If—to compensate for this "data force-fitting" effect—we were to explicitly restrict the number of deletable rows and columns individually, the problem probably becomes harder: Consider, for example, the forbidden submatrix $B := (1)$ over a binary alphabet. ROW DELETION(B) and COLUMN DELETION(B) are easily solvable in polynomial time—simply remove all rows (or columns, respectively) that contain a 1. However, allowing for both column- and row-deletion where each the number of deletable rows and columns is restricted individually is already *NP*-complete (this problem, known as CONSTRAINT BIPARTITE VERTEX COVER, is treated, e.g., in [FeNi01]).

The main results of this chapter are obtained by showing a close linkage between ROW DELETION(B) and the d-HITTING SET problem

Definition 4.4 (d-HITTING SET PROBLEM):
Input: *A collection $\mathcal{C}$ of subsets of size d of a finite set $\mathcal{S}$ and an integer k.*
Question: *Is there a subset $\mathcal{S}' \subseteq \mathcal{S}$ with $|\mathcal{S}'| \leq k$ that contains at least one element from each subset in $\mathcal{C}$?*

Already for $d = 2$, d-HITTING SET is *NP*-complete [DoFe99]. It is obvious that for a given d, d-HITTING SET can be solved by a search algorithm where the search tree has size $O(d^k)$. In [NiRo00], techniques using successive problem kernel reductions when traversing the search tree (called "interleaving") are introduced. These may be applied to obtain an algorithm with $O(d^k + n)$ running time for d-HITTING SET, which is thus fixed-parameter tractable. The best algorithm for the general d-HITTING SET problem known has a worst-time complexity of $O((d - 1 + O(d^{-1}))^k + n)$ [NiRo03$_a$]; for 2-HITTING SET and 3-HITTING SET, there exist even better algorithms that will be introduced later in this work.

The proofs presented in Section 4.3 will all rely on a special decomposition of the forbidden submatrix B, which we will call a *σ-decomposition*, illustrated in Figure 4.1 and formally stated in the following definition:

Definition 4.5 (σ-DECOMPOSITION)
Given an ℓ-ary $r \times s$ matrix $B = (b_{ij})$ over the alphabet $\mathcal{A}$. A permutation B_σ of B is called a σ-decomposition *of B if there exists a $\sigma \in \mathcal{A}$ and there exist r', r'', s', s'' with $r' + r'' = r, s' + s'' = s$ such that*

1. $r' > 0$ *and* $s' > 0$,
2. $\forall\ 1 \leq i \leq r', 1 \leq j \leq s' :\ b_{ij} \neq \sigma$ *(call this upper left submatrix* V*) and*
3. $\forall\ r' < i \leq r, 1 \leq j \leq s' :\ b_{ij} = \sigma$.

The upper right $r' \times s''$ *submatrix* $(b_{ij})_{1 \leq i \leq r', s' < j \leq s}$ *of* B_σ *is called* W*, the lower right* $r'' \times s''$ *submatrix* $(b_{ij})_{r' < i \leq r, s' < j \leq s}$ *is referred to as* U.

The left part $(b_{ij})_{1 \leq i \leq r, 1 \leq j \leq s'}$ *of* B_σ *(the one containing* V*) is called the* encoding part *of* B_σ*. The right part* $(b_{ij})_{1 \leq i \leq r, s' < j \leq s}$ *of* B_σ *(the one consisting of* W *and* U*) is called the* non-encoding part *of* B_σ.

Note that for the rest of this chapter, we will often use the names V, W, and U for a σ-decomposition of B in accordance with this definition.

A σ-decomposition of B can easily be generated in polynomial time by choosing a symbol σ from the alphabet $\mathcal{A}$ and a column c in B. Then, the rows in B are permuted such that all σ's in c are moved to the bottom of B. Then, c is swapped with the first column of B. For example, consider the matrix

$$B := \begin{pmatrix} 2&9&0&7&0&2 \\ 2&1&0&5&8&1 \\ 2&9&0&7&0&1 \\ 0&2&1&1&7&9 \end{pmatrix}$$

for which we now generate a σ-decomposition using 7 as σ and the 4th column as c:

$$\begin{pmatrix} 2&9&0&7&0&2 \\ 2&1&0&5&8&1 \\ 2&9&0&7&0&1 \\ 0&2&1&1&7&9 \end{pmatrix} \overset{\text{permute rows}}{\Longrightarrow} \begin{pmatrix} 2&1&0&5&8&1 \\ 0&2&1&1&7&9 \\ 2&9&0&7&0&2 \\ 2&9&0&7&0&1 \end{pmatrix} \overset{\text{swap columns}}{\Longrightarrow} \begin{pmatrix} 5&2&1&0&8&1 \\ 1&0&2&1&7&9 \\ 7&2&9&0&0&2 \\ 7&2&9&0&0&1 \end{pmatrix} =: B_\sigma$$

In this decomposition of B we have (according to Definition 4.5)

$$V = \begin{pmatrix} 5 \\ 1 \end{pmatrix},\ W = \begin{pmatrix} 2&1&0&8&1 \\ 0&2&1&7&9 \end{pmatrix},\ \text{and } U = \begin{pmatrix} 2&9&0&0&2 \\ 2&9&0&0&1 \end{pmatrix}.$$

Note that in the following hardness proofs of Section 4.3, the lower relative hardness bound of Row Deletion(B) will mainly depend on the height of V. For some of the following proofs, we are therefore seeking to maximize the height of V—we therefore define the *maximal σ-Decomposition* of B:

Definition 4.6 (Maximal σ-Decomposition)
A σ*-Decomposition* B_σ *of* B *is called* maximal *if there exists no* $\sigma' \in \mathcal{A}$ *such that there is a* σ'*-Decomposition* $B_{\sigma'}$ *of* B*, where* V *in* $B_{\sigma'}$ *is higher than in* B_σ*. We denote a maximal decomposition by* $B_{\sigma\text{-}max}$.

As an example, a maximal decomposition of the matrix $B = \begin{pmatrix} 1&0&1&1&0 \\ 0&1&0&1&1 \\ 1&1&1&1&0 \\ 0&1&0&0&1 \end{pmatrix}$ would be

$$B_{\sigma\text{-max}} = \begin{pmatrix} 1&1&0&1&0 \\ 1&0&1&0&1 \\ 1&1&1&1&0 \\ 0&0&1&0&1 \end{pmatrix}$$

with $\sigma = 0$ where $V = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$ has height 3.

Finding a maximal decomposition for a given B only requires a little more effort than finding any decomposition: We simply iterate over all ℓ symbols of the alphabet $\mathcal{A}$ of B and record for each symbol σ how many times σ appears in each column c of B, searching for a σ that appears least frequently in one column of B. Putting this in an algorithmic form:

Algorithm: Finding a maximal σ-Decomposition
Input: An ℓ-ary $r \times s$ matrix B
Output: A maximal σ-Decomposition of B

```
01  n ← ∞
02  σ_max ← 0
02  c_max ← 0
03  for σ ← 0 ... ℓ − 1 do
04      for each column c in B do
05          if σ appears n′ < n times in c then
06              n ← n′
07              σ_max ← σ
08              c_max ← c
09  permute B so that all σ_max's in c_max are at the bottom of c_max
10  swap the first column of B with c_max
```

Note that there may be more than one σ which allows a maximal decomposition of the forbidden submatrix. Furthermore, although there may be σ-decompositions for a given forbidden submatrix B that fulfill the prerequisites of Theorems 4.11 to 4.14 presented in Section 4.3.1, this might not be true for any of the maximal decompositions of B.

4.2 A Reduction to d-Hitting Set

This section is divided into two parts. The first part gives a polynomial-time algorithm for finding all inductions of a fixed forbidden submatrix in a larger matrix. This will lead to approximability and fixed-parameter tractability results presented in Subsection 4.2.2.

4.2.1 Finding Forbidden Submatrices

Before considering which rows to delete from a given input matrix A in order to make A B-free, we obviously have to actually *find* those sets of rows that are responsible for the induction of B in the first place.

An algorithm to achieve this proceeds as follows: In order to find all inductions of B in A, we first generate all q (at most $r!$) distinguishable row-permutations of B. We denote the ith permutation thus obtained by $\Pi_i(B)$. Each $\Pi_i(B)$ contains s column vectors called $\pi_{i1}, \ldots, \pi_{is}$. After having generated the permutations, for each combination of r rows in A, we iterate over the column vectors in A induced by these rows. If one of these vectors is equal to a π_{ij}, we mark that π_{ij}.[4] If, in one column iteration, all π_{ij} have been marked for a particular i, we know that these r rows induce B. Such an r-sized set of rows can then be added to the generated output collection $\mathcal{C}$. For the following formal description of the algorithm, we label the rows in A by 1 through n.

Algorithm: Finding all inductions of B in A
Input: A matrix A from the same alphabet as B
Output: A collection $\mathcal{C}$ of r-sized sets of rows in A

[4]Care must be taken if for some i, j', and j'' we have $\pi_{ij'} = \pi_{ij''}$. In the formal description of the algorithm, this is achieved by lines *09*, *10*, and *11* which allow each π_{ij} to be marked at most once and break the iteration over j (line *07*) once a certain π_{ij} has been marked.

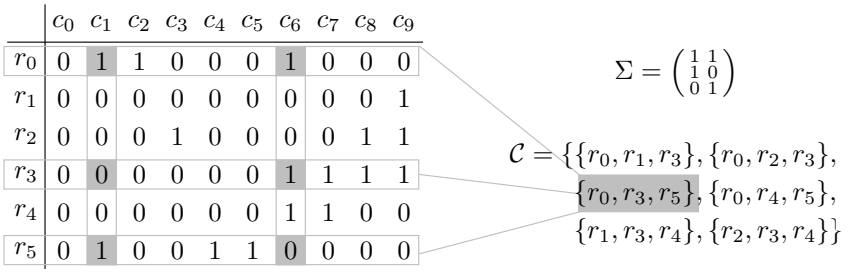


Figure 4.2: Finding all sets of rows that induce a forbidden submatrix: In the binary matrix on the left, we want to find all rows that induce the forbidden submatrix Σ. The corresponding output of row sets as generated by the algorithm presented in Section 4.2.1 is shown on the right. The grey underlay shows, as an example, how B is induced by the rows r_0, r_3, r_5 in columns c_1 and c_6 and thus leads to the addition of $\{r_0, r_3, r_5\}$ to $\mathcal{C}$.

where the rows in each set induce B.

```
01  Generate the column vectors π_{i1}, ..., π_{is} as described above
02  C ← ∅
03  for every r-sized subset S of {1, ..., n} do
04      create array V[q][s] filled with "0"s
05      for l ← 1 ... m do
06          for i ← 1 ... q do
07              for j ← 1 ... s do
08                  if the rows in S in the l-th column induce π_{ij} then
09                      if V[i][j] ≠ 1 then
10                          V[i][j] ← 1
11                          break
12      if ∃i : ∀1 ≤ j ≤ s : V[i][j] = 1 then
13          C ← C ∪ {S}
```

The output of this algorithm is illustrated in Figure 4.2.

The algorithm runs in polynomial time with respect to the input matrix A—given that the forbidden submatrix B is fixed—as the next theorem shows.

Theorem 4.7 *Given an $n \times m$ matrix A and a fixed $r \times s$ forbidden submatrix B (where $1 \leq r \leq n$ and $1 \leq s \leq m$). Then we can find all r-sized sets of rows in A that induce B in $O(n^r m)$—i.e., polynomial—time.*

Proof In order for the algorithm presented in this subsection to run in polynomial time, it is important to stress that B is not part of the input of a Row Deletion(B) problem. The running time of the above algorithm is $O(q \cdot s) = O(r! \cdot s)$ for the generation of the column vectors in each of the $\Pi_i(B)$ in line *01*. The running time of the inner loop in lines *04* through *11* is bounded by $O(s \cdot q + m \cdot q \cdot s) = O(s \cdot r! + m \cdot r! \cdot s)$, the if-statement in line *12* requires $O(q \cdot s) = O(r! \cdot s)$ time steps for execution. Lines *04* through *13* are executed due to the for-loop in line *03*, which adds a multiplying factor of $O(n^r)$. Thus, the total running time is

$$O\Big(\underbrace{(r! \cdot s)}_{\text{line } 01} + \underbrace{(n^r)}_{\text{line } 03} \cdot (\underbrace{(s \cdot r! + m \cdot r! \cdot s)}_{\text{lines } 04 \text{ to } 11} + \underbrace{(r! \cdot s)}_{\text{line } 12}\Big).$$

For a fixed B, r and s are constant, i.e., independent of the size of the input matrix A, and the running time is therefore

$$O(n^r m)$$

which is polynomial in the size of the input matrix A. □

It should be noted that the algorithm presented above will of course not always be directly applied to a given SUBMATRIX OCCURRENCE(B) problem since the factor q—although being constant for a given B—can get impractically large already if $r \approx 10$ (as it is only bounded by $r!$). This factor should thus not be omitted. However, this is just a worst-case estimation that does not make any use of special properties B might have. Such a property might be that two columns in B are permutations of each other, which leads to a significant reduction of q. This will not touch the $O(n^r m)$ bound given by Theorem 4.7, however, for practical applications, significantly improve the constant factors neglected in the O-notation. Such an example will be given in Theorem 5.19 where it will be shown how all inductions of the matrix Σ in a binary matrix can be found by testing for the presence of two out of the three column vectors $(1\ 1\ 0)^T$, $(1\ 0\ 1)^T$, and $(0\ 1\ 1)^T$ instead for all six distinguishable row permutations of B. This roughly halves the running time compared to the algorithm of this section.

4.2.2 Approximability and Fixed-Parameter Tractability Results

By Theorem 4.7, we can find all sets of rows in a matrix A that induce a forbidden submatrix B of size $r \times s$ in polynomial time. Thus, according to the definition of the ROW DELETION(B) problem, from each such set at least one row has to be removed from A in order for A to become B-free. Note how therefore, ROW DELETION(B) is closely related to r-HITTING SET.

Corollary 4.8 *Given an $n \times m$ matrix A and an integer k as an input instance of* ROW DELETION*(B) where B is an $r \times s$ matrix ($1 \leq r \leq n$ and $1 \leq s \leq m$). Then, this instance is parameter-preserving reducible to an instance $(\mathcal{C}, \mathcal{S}, k)$ of* r-HITTING SET.

Proof The idea behind this reduction is the following: We will—in polynomial time with respect to k and the size of A—find all inductions of B in A. If there is a set of rows in A that induces B, we delete at least one of the rows from that set in A to avoid this particular induction. This is the analogy to r-HITTING SET, where we must similarly choose at least one element from each subset in a given collection.

Thus, given an instance of ROW DELETION(B), we can generate an instance of r-HITTING SET by setting $\mathcal{S}$ equal the set of rows in A and using all r-sized sets of rows that induce B (as generated by the algorithm presented in Section 4.2.1) as $\mathcal{C}$. The parameter, k, is directly preserved. (An example for the reduction is given after the proof.)

Since the parameter, k, is preserved throughout the reduction, only the equivalence of solutions remains to be shown:

Let $\mathcal{S}' \subseteq \mathcal{S}$ be a solution of size k to the r-HITTING SET problem $(\mathcal{C}, \mathcal{S}, k)$ generated by the reduction. Now, delete the rows in A that correspond to the elements in $\mathcal{S}'$, yielding A'. Assume that B were still induced in A' by a set I of rows. Then, the rows in I did induce B in A, meaning a set containing these rows was put into $\mathcal{C}$. But one row of I must then have been deleted since $\mathcal{S}'$ is a valid solution to $(\mathcal{C}, \mathcal{S}, k)$, a contradiction. Therefore, B cannot be induced by A' anymore.

If, on the other hand, A can be made B-free by deleting k rows, then for each induction of B in A by some rows, at least one of those rows must have been deleted. This implies, that, by choosing the elements corresponding to the deleted rows as a solution $\mathcal{S}' \subseteq \mathcal{S}$ to the generated r-Hitting Set instance, we have chosen at least one element from every set in $\mathcal{C}$, making $\mathcal{S}'$ a valid solution of size k. □

Let us illustrate the reduction of the above corollary by the following example. Let the forbidden submatrix be Σ and

$$A := \begin{pmatrix} 0&1&1&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&1&1 \\ 0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&1&1&0&0 \\ 0&1&0&0&1&1&0&0&0&0 \end{pmatrix}$$

be the input matrix to Row Deletion(B), consisting of the six rows $r_0 \dots r_5$ and ten columns $c_0 \dots c_9$. Since B is of height 3, the algorithm from Section 4.2.1 will generate a 3-Hitting Set instance. Iterating over every possible combination of three rows in A, we find that B is induced by the rows r_0, r_1, r_3 (in columns c_6 and c_9), r_0, r_2, r_3 (columns c_6 and c_8), etc. Thus, the generated 3-Hitting Set instance consists of $\mathcal{S} = \{r_0, \dots, r_5\}$ and $\mathcal{C} = \{\{r_0, r_1, r_3\}, \{r_0, r_2, r_3\}, \dots\}$. The complete $\mathcal{C}$ was already given in Figure 4.2, where it is also shown (grey underlays) how, as an example, B is induced by the rows r_0, r_3, r_5 in columns c_1 and c_6. Observe that A can be made B-free by deleting, e.g., the first and last row, which leads to

$$A' := \begin{pmatrix} 0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&1&0&0&0&0&1&1 \\ 0&0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&1&1&0&0 \end{pmatrix}.$$

Choosing r_0 and r_4 accordingly solves the 3-Hitting Set instance.

It might seem awkward that we have performed a reduction to a known *NP*-complete problem without having proven that Row Deletion(B) is *NP*-complete in the first place. The dependency between B and the hardness of Row Deletion(B) are further analyzed in the next section, where some structures for B are given for which Row Deletion(B) is *NP*-complete.

With the reduction of Row Deletion(B) to d-Hitting Set from Corollary 4.8, we can immediately deduce that Row Deletion(B) is fixed-parameter tractable:

Corollary 4.9 *Given B,* Row Deletion*(B) is fixed-parameter tractable.*

Proof Corollary 4.8 gave a parameter-preserving (and therefore also parameterized) reduction from Row Deletion(B) to d-Hitting Set with a fixed d (equivalent to the height r of the forbidden submatrix B). □

We can also obtain an analogous approximability result:

Corollary 4.10 Row Deletion*(B) for a given $r \times s$ matrix B may be approximated to a factor of r in polynomial time.*

Proof We first perform the reduction from Row Deletion(B) to the corresponding d-Hitting Set problem. The corollary directly follows from the facts that k is preserved in the reduction and that d-Hitting Set can be approximated to a factor of d by subsequently choosing all elements from a set in $\mathcal{C}$ until we have a valid solution.[5] □

[5]The approximation factor is due to the observation that for every set from $\mathcal{C}$, we must choose at least one element due to the definition of d-Hitting Set.

So far, we have shown that a given Row Deletion(B) problem is not harder than its corresponding d-Hitting Set problem. However, no lower complexity bound[6] for the hardness of Row Deletion(B) has been given. In Chapter 3, we have introduced the "parameter-preserving" reduction for finding close relationships concerning the fixed-parameter complexity of problems. Such a reduction will be used in the following discussion to exploit some relationships between Row Deletion(B) and d-Hitting Set depending on the structure of B.

4.3 Hardness Results

The main results of this section concerning the hardness of Row Deletion(B) depending on the structure of B are summarized in Subsection 4.3.1. The proofs for Theorems 4.11 to 4.14 are provided in the subsequent Subsections 4.3.2, 4.3.3, and 4.3.4.

4.3.1 Overview of Results—Four Theorems

The main results of this section are as follows:

Theorem 4.11 *(Proof on page 47)*
Let B be a forbidden submatrix of size $r \times s$ with a σ-decomposition B_σ where the submatrix V (of height r') of B_σ is not induced in the non-encoding part of B_σ. Then there exists a parameter-preserving reduction from r'-Hitting Set to Row Deletion(B).

If V *is* induced in the non-encoding part of B_σ, but we can find one column vector of V which is induced there *at most once* the following hardness result for Row Deletion(B): can be achieved:[7]

Theorem 4.12 *(Proof on page 53)*
If the $r \times s$-submatrix B has a σ-decomposition B_σ where the submatrix V of height r' has a column vector v that is induced at most once in the non-encoding part of B_σ, then r'-Hitting Set is parameter-preserving reducible to Row Deletion(B).

If neither the prerequisites for Theorems 4.11 nor those for Theorem 4.12 can be fulfilled, there are two more subcases for which a hardness result can be established:

Theorem 4.13 *(Proof on page 43)*
Let B be a forbidden $r \times s$-submatrix with a σ-decomposition B_σ where all entries of U are equal to σ and V contains r' rows. Then r'-Hitting Set is parameter-preserving reducible to Row Deletion(B).

Theorem 4.14 *(Proof on page 44)*
Let B be a forbidden $r \times s$-submatrix with a σ-decomposition B_σ where all entries of W are equal to σ and V contains r' rows. Then r'-Hitting Set is parameter-preserving reducible to Row Deletion(B).

[6] Recall that Chapter 3, we stressed that in complexity theory, computational bounds are always relative, meaning they are only statements such as "if problem $\mathcal{L}_1$ is easy, $\mathcal{L}_2$ is easy as well".

[7] Note that Theorem 4.11 and Theorem 4.12 are in some sense orthogonal, since it is possible to construct a submatrix B which fulfills the prerequisites of Theorem 4.11, but does not those of Theorem 4.12, and vice versa.

The *NP*-completeness of d-Hitting Set for $d \geq 2$ leads to the following corollary:

Corollary 4.15 *If a forbidden submatrix B has a σ.decomposition that fulfills the prerequisites from any of the Theorems 4.11-4.14 and where V has height 2 or greater,* Row Deletion*(B) is NP-complete.*

For all other cases, i.e., if B does not fulfill any of the prerequisites[8] from Theorems 4.11 to 4.14, no general statement on the problem's complexity has been established so far (note the conjecture in Section 4.4).

The proofs for Theorems 4.13 and 4.14 will be presented in the following subsection, followed by separate subsections for the more involved proofs of Theorem 4.11 and Theorem 4.12 in Subsections 4.3.3 and 4.3.4, respectively. Following each proof, an example is given to illustrate its main ideas.

4.3.2 Proofs for Theorems 4.13 and 4.14

The key idea behind all of the following reductions will be to use the matrix V of a given σ-decomposition of the forbidden submatrix B to encode a given d-Hitting Set instance into an instance of Row Deletion(B) (i.e., a matrix A) and use σ as a "filling-symbol" to prevent unwanted inductions of B in A. This idea is illustrated in more detail in the proof of the following Lemma:

Lemma 4.16 *Let B be a forbidden submatrix of size $r \times s$ with a σ-decomposition B_σ where $r = r'$ ($r'' = 0$). Then r-*Hitting Set *can be parameter-preservingly reduced to* Row Deletion*(B).*

Proof Let $(\mathcal{C}, \mathcal{S}, k)$ be an instance of r-Hitting Set, where $\mathcal{S} = \{1, \ldots, n\}$. The idea behind the reduction is the following: We will create a matrix A of size $n \times (s \cdot |\mathcal{C}|)$ where each row corresponds to an element in $\mathcal{S}$. For each set C in $\mathcal{C}$, we will encode a B into a set of s consecutive columns of A, using the rows that correspond to the elements in C. We then have to show that there is a "1:1-correspondence" between deleting a row in A and choosing an element from $\mathcal{S}$. More precisely, we have to show that when we delete a row in A corresponding to the element $z \in \mathcal{S}$, exactly those induced B in A are destroyed that were encoded due to sets from $\mathcal{C}$ that contained z.

The encoding of B is done by the following algorithm:

Algorithm: r-Hitting Set to Row Deletion(B), Lemma 4.16
Input: An instance $(\mathcal{C}, \mathcal{S}, k)$ of r-Hitting Set
Output: An instance of Row Deletion(B) which is parameter-equivalent to the given r-Hitting Set instance

```
01  create an n × s · |C| matrix A = (a_{i,j}) filled with σ's
02  col ← 0
03  for each set C ∈ C do
04      row ← 1
05      for each i ∈ C do
06          for  c ← 1 ... s do
```

[8] An example of such a matrix would be $B = \left(\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{smallmatrix}\right)$ over the alphabet $\Sigma = \{0, 1\}$.

```
07          a_{i,col+c} ← b_{row,c}
08        row ← row + 1
09      col ← col + s
10      return (A, k)
```

Line *07* of the code may be interpreted as follows: For the *row*th element z_i of the *col*th set $C \in \mathcal{C}$, the *row*th row of B is written into the *col*th set of s consecutive columns in A into the z_ith row (i.e. the row in A that corresponds to i).

For every set C in the collection $\mathcal{C}$, B is induced in the rows of A corresponding to the elements of C (lines *05* through *07*). The other entries of A remain filled with σ's. This reduction can be computed in polynomial time with respect to the input, i.e.,

$$O(|\mathcal{C}| \cdot r \cdot s).$$

Having established that the reduction can be computed in polynomial time whilst preserving the parameter k, only the equivalence of solutions remains to be shown.

Assume that we have a solution $\mathcal{S}' \subset \mathcal{S}$ of size k to a given r-Hitting Set instance. We then delete the rows in A that correspond to the elements of $\mathcal{S}'$, obtaining A'. Now, assume that B is still induced in A'. Then, the $r \times s'$ submatrix V in the decomposition of B is also induced in A'.

Claim: If V is induced by a set of columns in A', none of these columns contains less than r symbols different from σ.
Proof: Having a column in A' with less than r symbols different from σ inducing a column of V is a contradiction to the fact that V has height r and does not contain σ.

The claim implies that there is at least one column c in A' that contains exactly r symbols different from σ (note that due to the encoding of the algorithm, no column of A' may contain more than r symbols different from σ). If this is the case, then there is a set in $\mathcal{C}$ from the r-Hitting Set instance that was encoded into A for which none of the encoding rows have been deleted. But then $\mathcal{S}'$ contains no element of this set, a contradiction to the assumption that $\mathcal{S}'$ is a solution to the given r-Hitting Set instance.

Now, we prove the reverse direction of the above: Assume that by deleting k rows in A we can make the resulting matrix A' B-free. Then, from each set of rows that induces B in A, at least one row must have been deleted. This implies that from each B which was encoded into A, at least one row has been deleted. We claim that the set $\mathcal{S}' \subseteq \mathcal{S}$ consisting of those k elements in $\mathcal{S}$ for which the corresponding rows in A have been deleted, solves the given r-Hitting Set instance. If this were not the case, there would be a set $C \in \mathcal{C}$ for which $C \cap \mathcal{S}' = \emptyset$. For every set in $\mathcal{C}$, the algorithm given above encoded a matrix B into A. The existence of a $C \in \mathcal{C}$ for which $C \cap \mathcal{S}' = \emptyset$ then implies, however, that in A', all rows of this particular encoding are still present, a contradiction to the assumption that A' is B-free. □

Let us illustrate the proof of the above lemma by an explicit example (a more general scheme for the above proof is provided in Figure 4.3). Assume that we are given the forbidden submatrix $B := \binom{1}{1}$ over a binary alphabet. A σ-Decomposition of B is $B_\sigma := \binom{1}{1}$ with $\sigma = 0$. Now, for the example, we will take the following instance of 2-Hitting Set[9]: $\mathcal{S} = \{1,2,3,4,5\}$, $\mathcal{C} = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,5\},\{3,4\}\}$, the parameter k is arbitrary as it is preserved by the reduction. The algorithm given in the above proof will then initialize

[9] 2-Hitting Set is more commonly known as Vertex Cover.

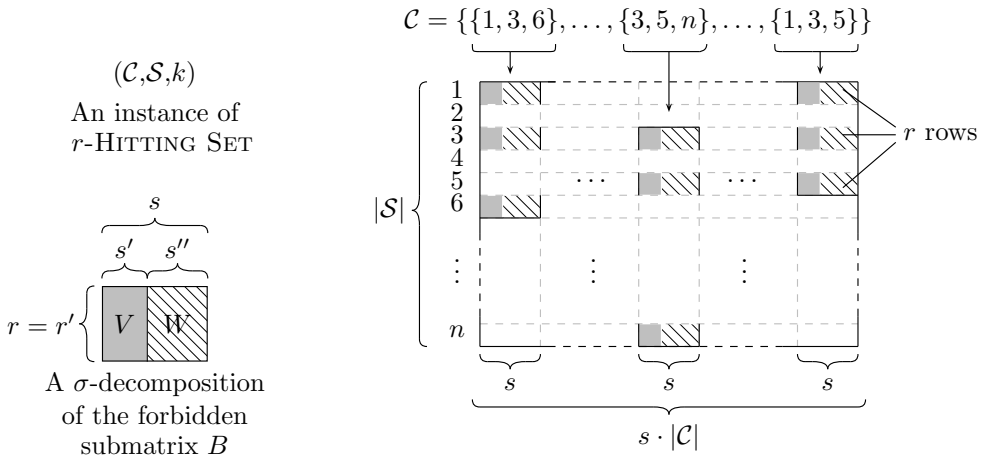


Figure 4.3: Reduction from r-Hitting Set to Row Deletion(B) used in the proof of Lemma 4.16: Given a r-Hitting Set instance and a decomposition of B where V has the same height as B. Then, the algorithm from the proof of Lemma 4.16 gives a parameter-preserving reduction from r-Hitting Set to Row Deletion(B) (output of this algorithm is the right matrix). All entries of the output matrix not containing any part of V or W are equal to σ.

a $|\mathcal{S}| \times s \cdot |\mathcal{C}|$ (i.e., $5 \times 1 \cdot 6$) matrix A, fill it with zeros, and then iteratively (lines 04 through 09) encode each set of $\mathcal{C}$:

$$A = \begin{pmatrix} 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix} \overset{\text{encode } \{1,2\}}{\Longrightarrow} A = \begin{pmatrix} 1&0&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix} \Longrightarrow$$

$$\overset{\text{encode } \{1,3\}}{\Longrightarrow} A = \begin{pmatrix} 1&1&0&0&0&0 \\ 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix} \overset{\text{encode } \{1,4\}}{\Longrightarrow} A = \begin{pmatrix} 1&1&1&0&0&0 \\ 1&0&0&0&0&0 \\ 0&1&0&0&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix} \Longrightarrow$$

$$\overset{\text{encode } \{2,3\}}{\Longrightarrow} A = \begin{pmatrix} 1&1&1&0&0&0 \\ 1&0&0&1&0&0 \\ 0&1&0&1&0&0 \\ 0&0&1&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix} \Longrightarrow \ldots \Longrightarrow A = \begin{pmatrix} 1&1&1&0&0&0 \\ 1&0&0&1&1&0 \\ 0&1&0&1&0&1 \\ 0&0&1&0&0&1 \\ 0&0&0&0&1&0 \end{pmatrix}$$

An optimal solution of size 3 to the encoded instance of 2-Hitting Set is, e.g., $\mathcal{S}' = \{2, 3, 4\}$. Deleting the corresponding rows in A leaves us with

$$A' = \begin{pmatrix} 1&1&1&0&0&0 \\ 0&0&0&0&1&0 \end{pmatrix}.$$

Note that each column in A' contains less than two 1's and can therefore not participate in an induction of B—this is exactly the argument employed in the above proof. It also illustrates the purpose of the σ-Decomposition: In our example, 0 was chosen as σ and thus is not part of V in the σ-Decomposition of B. The advantage of V having the same height as B and not containing σ's is that, during the encoding of the given r-Hitting Set instance, it is easy to avoid unwanted inductions of V in A and therefore ensure that destroying all induced V in A will also destroy all induced B.[10] The main reason for the subsequent proofs

[10]For instance in the above example, we simply had to ensure that from every column containing two 1's, at least one 1 is deleted.

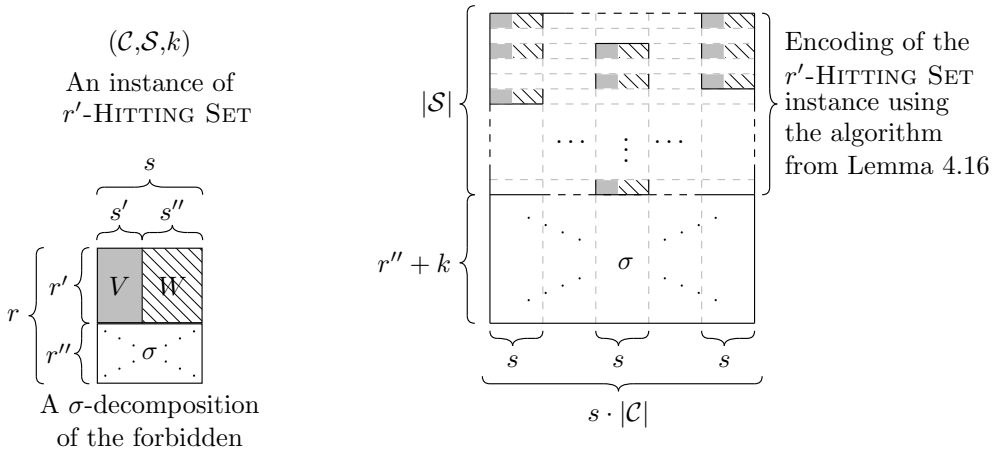


Figure 4.4: Reduction from r'-Hitting Set to Row Deletion(B) used in the proof of Theorem 4.13: Given an r'-Hitting Set instance and a decomposition of B where V is of height r' and the bottom r'' rows contain just σ's as their entries. Then the algorithm from the proof of Theorem 4.13 gives a parameter-preserving reduction from r'-Hitting Set to Row Deletion(B) (output of this algorithm is the right matrix). All entries of the output matrix that are not explicitly determined by V or W in the figure are equal to σ.

in this and the following subsection becoming quite involved is to ensure that no unwanted inductions of B occur in the output matrix A. For the proof of Theorem 4.13, however, Lemma 4.16 is easily extended:

Proof of Theorem 4.13 Let $(\mathcal{C}, \mathcal{S}, k)$ be an instance of r'-Hitting Set, where $\mathcal{S} = \{1, \ldots, n\}$. First, encode the given r'-Hitting Set instance into a matrix A' using the algorithm from Lemma 4.16 and the first r' rows of B_σ as the forbidden submatrix. Then, $r'' + k$ rows[11] containing just σ-entries are added to the bottom of A', yielding A.

In close analogy to the proof of Lemma 4.16, a solution $\mathcal{S}' \subset \mathcal{S}$ to the r'-Hitting Set-problem gives a solution to Row Deletion(B) on A: Deleting those rows in A that correspond to the elements in $\mathcal{S}'$ inhibits all inductions of V in A. Proving the reverse direction, we need to delete at least one row from every encoded V in A in order to make A B-free (note that this cannot be done by deleting the bottom rows as there are too many with respect to k). □

An overview for the encoding employed in the above proof is given in Figure 4.4. A σ-decomposition of a matrix that would fulfill the conditions of Theorem 4.13 is, e.g.,

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

with $\sigma = 0$, and $U = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$. Due to the close analogy to Lemma 4.16, we shall not give an explicit example for the encoding.

[11] Recall from Definition 4.5 that r'' is the height of the submatrix U in the σ-Decomposition of B

The proof of Theorem 4.14 requires a more involving extension of the ideas presented in Lemma 4.16.

Proof of Theorem 4.14 The construction of the input matrix A to Row Deletion(B) is best explained as follows: Imagine A to be composed of four submatrices (see also Figure 4.5 for an illustration); the upper left submatrix is generated by the algorithm from Lemma 4.16 to encode a given instance of r'-Hitting Set using V as the forbidden submatrix. The lower right matrix contains U in some form (to which we will come later on in this proof). The other two submatrices contain just σ's as entries. Note that then each V induced in the upper left submatrix of A induces B with any induced U in the lower right submatrix of A. If we ensure that the induction of B in A cannot be prevented by deleting rows from the encoded U and that V is not induced in the lower right submatrix of A, an argument similar to the one used to prove Lemma 4.16 can then be employed to prove this lemma.

Now, let $(\mathcal{C}, \mathcal{S}, k)$ be an instance of r'-Hitting Set with $\mathcal{S} = \{1, \ldots, n\}$. The given r'-Hitting Set instance is first encoded into a matrix A'' using the algorithm from Lemma 4.16 and V as the forbidden submatrix. Then, $s'' \cdot (k+1)$ columns containing just σ-entries are added to the right of A'', yielding A'. To A', $r'' \cdot (k+1)$ rows containing just σ-entries are added to its bottom; we thus obtain A.

As already explained above, we now need to write some U's into A in order to induce B in A. For these U we will require that the following two be fulfilled:

a) V is not induced in the resulting matrix by the entries of the written U's (this is to ensure that B is not induced in the right part of A).

b) No induced B can be destroyed by deleting at most k of the lower $r'' \cdot (k+1)$ rows of A (this will ensure the equivalence of solutions later on in the proof).

In order to fulfill these two conditions, two cases are distinguished:

I. The matrix U does not induce V:

We will write $k+1$ U into the lower right submatrix of A in a diagonal pattern. Writing U more than k times secures condition b) stated above, the pattern will ensure that a) is fulfilled.

Writing U into A is done by applying the following algorithm to A (using the lower right submatrix $U = (u_{i,j})$ of B_σ):

Algorithm: Encoding U, Theorem 4.14, case I
Input: The values of $|\mathcal{C}|$ and k of an instance $(\mathcal{C}, \mathcal{S}, k)$ of r'-Hitting Set, the matrices $A = (a_{i,j})$ and $U = (u_{i,j})$
Output: Matrix A modified accordingly

```
01  for  h ← 0 ... k
02      for  i ← 1 ... r''
03          for  j ← 1 ... s''
04              a_{|S|+h·r''+i, |C|·s'+h·s''+j} ← u_{i,j}
```

This algorithm encodes U $k+1$ times into A (see Figure 4.5), each in a different set of rows. Any induced B in A can therefore only be destroyed by deleting rows that

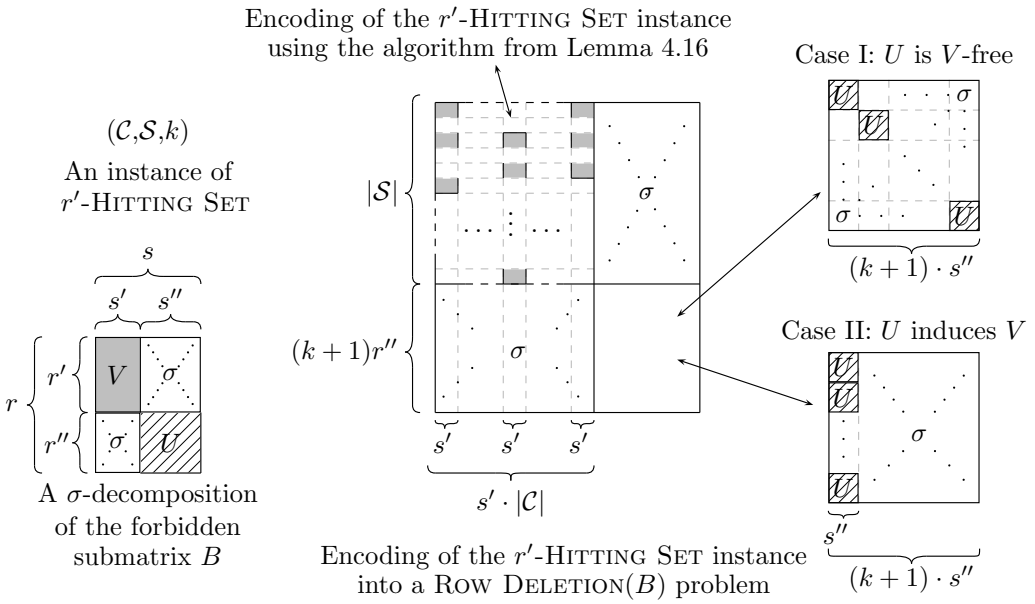


Figure 4.5: Reduction from r'-Hitting Set to Row Deletion(B) used in the proof of Theorem 4.14: Given a r'-Hitting Set instance and a decomposition of B where W contains just σ's. Then the algorithms from the proof of Theorem 4.14 (with Cases I and II) give a parameter-preserving reduction from r'-Hitting Set to Row Deletion(B).

induce the respective V. Therefore, if we have a solution for Row Deletion(B) on A of size at most k, at least one row from each encoded matrix V must have been deleted, which is also a solution to the original r'-Hitting Set problem (see Lemma 4.16 for a more detailed explanation of this argument). Now, if there is a solution of size k to the r'-Hitting Set problem, deleting the corresponding rows in A will also destroy all inductions of V in A. Since V is not induced by any U (note that multiple U cannot induce V because of the way they are encoded), A is V-free after the deletion and therefore also B-free.

II. The matrix U induces V:

The strategy for enclosing the U into A in this case is the following: We will fill the lower right submatrix in $\mathcal{C}$ of A with s'' columns containing just U (see Figure 4.5). Note that then, although both V and U are induced in the lower right submatrix of A, B is not induced there because this submatrix contains only s'' columns with entries other than σ's and for s (the width of B), $s > s''$ holds. The final argument in the proof of this case is that if V is not induced in the upper left submatrix of A, U cannot be induced there either (since U induces V) and therefore, A would then be B-free.

Writing U into A is done by applying the following algorithm to A (using the lower right submatrix $U = (u_{i,j})$ of B_σ):

Algorithm: Encoding U, Theorem 4.14, case II
Input: The values of $|\mathcal{C}|$ and k of an instance
$(\mathcal{C}, \mathcal{S}, k)$ of r'-Hitting Set, the

matrices $A = (a_{i,j})$ and $U = (u_{i,j})$
Output: Matrix A modified accordingly

```
01 for h ← 0 ... k
02     for i ← 1 ... r''
03         for j ← 1 ... s''
04             a_{|S|+h·r''+i,|C|·s'+j} ← u_{i,j}
```

This algorithm encodes U $k+1$ times into the lower right submatrix $A_{\mathrm{LR}} := (a_{ij})$ of A where $i = |\mathcal{S}| \ldots |\mathcal{S}| + (k+1) \cdot r''$ and $j = |\mathcal{C}| \cdot s', \ldots, |\mathcal{C}| \cdot s' + (k+1) \cdot s''$. Figure 4.5 provides an illustration for this. Although V is induced in the Us written into A_{LR}, B is not induced there because $s > s''$. Therefore, A is B-free if the upper left submatrix of A (designated A'' at the beginning of this proof) induces neither V nor U.

The rest of our argument is similar to the proof of Lemma 4.16: Since U induces V, A'' fulfills this condition if it does not induce V. Therefore, if we have a solution for Row Deletion(B) on A of size at most k, at least one row from each encoded matrix V must have been deleted, which is also a solution to the original r'-Hitting Set problem. If, on the other hand, there is a solution of size k to the r'-Hitting Set problem, deleting the corresponding rows in A will also destroy all induced B in A by destroying all V in A''.

Summarizing the two cases, we have proven the theorem. □

The general scheme for the above proof is illustrated in Figure 4.5; we shall now furthermore illustrate it with the following example:

Let there be two forbidden submatrices B_{I} and B_{II} with

$$B_{\mathrm{I}} := \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 2 \end{pmatrix} \text{ and } B_{\mathrm{II}} := \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}.$$

For $\sigma = 0$, matrix B_{I} meets case I of the above proof with $V = \binom{1}{1}$, $U = \binom{1}{2}$ and B_{II} meets case II with $V = \binom{1}{1}$, $U = \binom{1}{1}$. Using the above matrices, we encode the instance $\mathcal{C} = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,5\},\{3,4\}\}$, $\mathcal{S} = \{1,2,3,4,5\}$, and $k = 3$ of 2-Hitting Set. We perform two parameter-preserving reductions from this instance to two instance A_{I} and A_{II} of Row Deletion(B_{I}) and Row Deletion(B_{II}), respectively. Both A_{I} and A_{II} will be of size $(|\mathcal{S}| + (k+1) \cdot r'') \times (s' \cdot |\mathcal{C}| + (k+1) \cdot s'')$ (i.e., 13×10). The upper right 5×6 submatrices of A_{I} and A_{II} are generated by the algorithm from Lemma 4.16 using V as the forbidden submatrix (note that V are the same for B_{I} and B_{II}) and equal to

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

For a more detailed explanation as to how we derive this matrix, the reader is deferred to the corresponding example of Lemma 4.16. Filling the lower left and upper right submatrices of A_{I} and A_{II} with σ's and encoding the respective U, we arrive at

$$A_{\mathrm{I}} := \begin{pmatrix} 1&1&1&0&0&0&0&0&0&0 \\ 1&0&0&1&1&0&0&0&0&0 \\ 0&1&0&1&0&1&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&2&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&2&0&0 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&2&0 \\ 0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&2 \end{pmatrix} \text{ and } A_{\mathrm{II}} := \begin{pmatrix} 1&1&1&0&0&0&0&0&0&0 \\ 1&0&0&1&1&0&0&0&0&0 \\ 0&1&0&1&0&1&0&0&0&0 \\ 0&0&1&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \end{pmatrix}.$$

Note that after deleting at most 3 out of the bottom 8 rows, the respective forbidden submatrices are still induced. A solution of size 3 to the encoded instance of 2-Hitting Set is, e.g., $\mathcal{S}' = \{2, 3, 4\}$. Deleting the corresponding rows in each A_{I} and A_{II} leaves us with

$$A'_{\mathrm{I}} := \begin{pmatrix} 1&1&1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&2&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&2&0&0 \\ 0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&2&0 \\ 0&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&2 \end{pmatrix} \text{ and } A'_{\mathrm{II}} := \begin{pmatrix} 1&1&1&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0 \end{pmatrix},$$

where the respective B_{I} and B_{II} are no longer induced.

4.3.3 Proof of Theorem 4.11

Proof of Theorem 4.11 This reduction from r'-Hitting Set to an instance (A, k) of Row Deletion(B) is very similar to the one used in the proof of Theorem 4.14, except that we additionally have to encode W appropriately into A to ensure that B is indeed induced in conjunction with the encoded V and U. Let $(\mathcal{C}, \mathcal{S}, k)$ be a given instance of r'-Hitting Set, where $\mathcal{S} = \{1, \ldots, n\}$.

Firstly, the given r'-Hitting Set instance is encoded into a matrix A' using the algorithm from Lemma 4.16 and V as the forbidden submatrix. Then, the given instance of r'-Hitting Set is encoded into a second matrix A'', this time using W as the forbidden submatrix. A third matrix A''' is generated by writing matrices $U = (u_{i,j})$ from the decomposition of B in a diagonal fashion into it (in a similar way as was done in the first case in the proof of Theorem 4.14). This is done by applying the following algorithm to a $(|\mathcal{C}| \cdot r'') \times (|\mathcal{C}| \cdot s'')$ matrix $A''' = (a'''_{i,j})$ filled with σ's:

Algorithm: Encoding U into A''' according to Theorem 4.12
Input: An instance $(\mathcal{C}, \mathcal{S}, k)$ of r'-Hitting Set,
the matrices $A''' = (a'''_{i,j})$ and $U = (u_{i,j})$
Output: Matrix A''' modified accordingly

```
01 for h ← 0 ... |C|
02     for i ← 1 ... r''
03         for j ← 1 ... s''
04             a'''_{h·r''+i, h·s''+j} ← u_{i,j}
```

The final matrix A we use for the reduction is then composed by fitting together four submatrices: The upper left submatrix is A', the upper right A'', the lower left contains just σ and the lower right is A'''. Figure 4.6 illustrates the resulting matrix A.

The parameter k is preserved by the reduction, which can be carried out in polynomial time with respect to the input size (since the four components of A can each be constructed in polynomial time). The equivalence of solutions remains to be shown:

Note that deleting a row corresponding to an element in $\mathcal{S}$ will always destroy more encoded submatrices than deleting any other row. Assume that by deleting at most k rows in A, we can make A B-free. Then there exists such a solution where only rows corresponding to elements in $\mathcal{S}$ have been deleted. From every induced V in the upper part of A, at least one row must have been deleted. This directly implies that by choosing those elements from $\mathcal{S}$ corresponding to the deleted rows as the elements of $\mathcal{S}'$, we have chosen at least one element

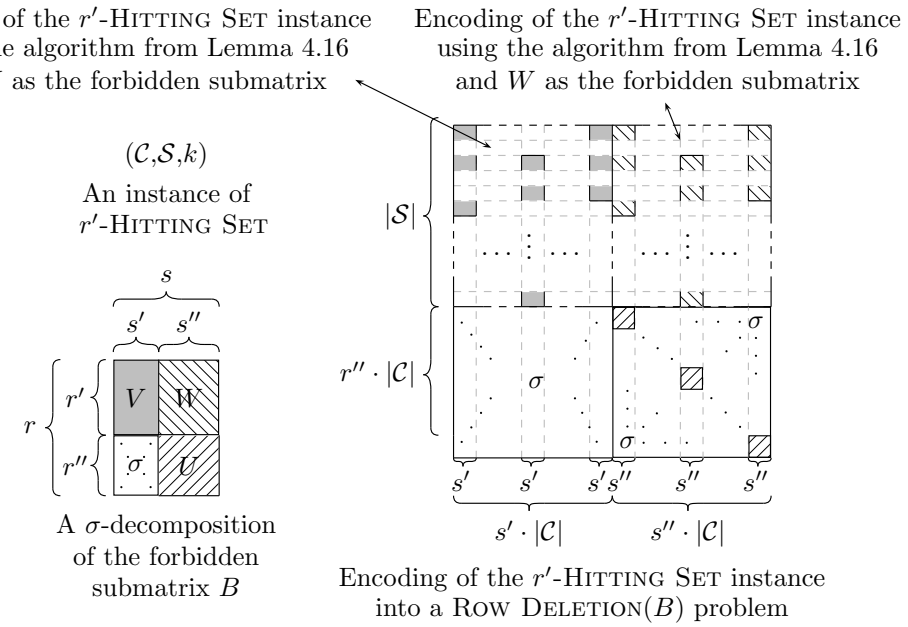


Figure 4.6: Reduction from r'-Hitting Set to Row Deletion(B) used in the proof of Theorem 4.11: Given a r'-Hitting Set instance and a σ-decomposition B_σ of B where the part of B_σ containing U and W does not induce V. Then Case I of the two constructions used in the proof of Theorem 4.14 gives a parameter-preserving reduction from r-Hitting Set to Row Deletion(B).

from every encoded set from the r'-Hitting Set instance. Thus, $\mathcal{S}'$ is a valid solution of size at most k to the given r'-Hitting Set instance.

Now, assume that we have a solution $\mathcal{S}' \subset \mathcal{S}$ of size k for the given r'-Hitting Set and delete the corresponding[12] rows in A, obtaining A_{del}. Then, from each V encoded into the upper left part (designated A' in the construction) of A, at least one row has been deleted. Every column in A' contains less than r' σ's after the deletion. Since V does not contain the symbol σ, this also implies that V is not induced in A' after the deletion. Therefore, if V were still to be induced in A_{del}, this would mean that V was already induced in the right part (designated A'' and A''' in the construction) of A. Let us call this part A_{right}.

The induction of V in A_{right} is, however, impossible for the following reason: Recall that V was not induced in the non-encoding part of B_σ. If V is to be induced in A_{right}, this implies that $s' \geq 2$, because if $s' = 1$, then V is induced in a single column of A_{right} and therefore also induced in B.

Furthermore, the induction would have to include at least two columns in A_{right} that were generated by the encoding of two *different* sets of $\mathcal{C}$. If these two columns are indeed part of an induced V in A_{right}, they must induce at least r' row vectors with symbols different from σ in both columns. Now observe the row vectors induced by two columns c_1 and c_2 in A_{right} that were generated by the encoding of two different sets of $\mathcal{C}$:

- Since two sets in $\mathcal{C}$ differ in at least one element, in the upper n rows of A_{right} a symbol

[12]Recall that in the first n rows of A, the ith row corresponds to the ith element in $\mathcal{S}$

different from σ in the first column can meet a symbol different from σ in the second column at most $r' - 1$ times.

- In the rows below the first n rows of A_{right}, the U's corresponding to the encoding of the two sets from $\mathcal{C}$ have been written into A_{right} in the diagonal fashion as illustrated in Figure 4.6. Therefore, a symbol different from σ is always met by a σ in the other column in any of the lower rows.

Thus, for all rows in A_{right}, c_1 and c_2 only induce at most $r' - 1 < r'$ row vectors with both symbols different from σ and therefore cannot induce V.

Thus, if we have a solution $\mathcal{S}' \subset \mathcal{S}$ of size k for the given r'-Hitting Set and delete the corresponding rows in A, V is not induced in the resulting matrix anymore and therefore, B is not induced there anymore; indicating that we have a valid solution of size k to Row Deletion(B). □

Let us illustrate the above reduction by an example: Take the forbidden submatrix $B := \left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 2 \end{smallmatrix}\right)$ over a ternary alphabet. Taking $\sigma = 0$, we see that this matrix is already a σ-decomposition according to the prerequisites of Theorem 4.12, since the right column of B does not induce the left one. As in our previous examples, we shall again create an instance of Row Deletion(B) from the 2-Hitting Set problem $\mathcal{S} = \{1, 2, 3, 4, 5\}$, $\mathcal{C} = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 5\}, \{3, 4\}\}$, and arbitrary k. Using the construction for the matrices A', A'', and A''' (that will later be put together to construct the instance of Row Deletion(B)) given in the proof, we obtain

$$A' = \begin{pmatrix} 1&1&1&0&0&0 \\ 1&0&0&1&1&0 \\ 0&1&0&1&0&1 \\ 0&0&1&0&0&1 \\ 0&0&0&0&1&0 \end{pmatrix}, \; A'' = \begin{pmatrix} 1&1&1&0&0&0 \\ 0&0&0&1&1&0 \\ 0&0&0&0&0&1 \\ 0&0&0&0&0&0 \\ 0&0&0&0&0&0 \end{pmatrix}, \text{ and } A''' = \begin{pmatrix} 2&0&0&0&0&0 \\ 0&2&0&0&0&0 \\ 0&0&2&0&0&0 \\ 0&0&0&2&0&0 \\ 0&0&0&0&2&0 \\ 0&0&0&0&0&2 \end{pmatrix}.$$

Putting these matrices together as $\left(\begin{smallmatrix} A' & A'' \\ 0 & A''' \end{smallmatrix}\right)$, we finally get

$$A = \begin{pmatrix} 1&1&1&0&0&0&1&1&1&0&0&0 \\ 1&0&0&1&1&0&0&0&0&1&1&0 \\ 0&1&0&1&0&1&0&0&0&0&0&1 \\ 0&0&1&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&2&0&0&0&0&0 \\ 0&0&0&0&0&0&0&2&0&0&0&0 \\ 0&0&0&0&0&0&0&0&2&0&0&0 \\ 0&0&0&0&0&0&0&0&0&2&0&0 \\ 0&0&0&0&0&0&0&0&0&0&2&0 \\ 0&0&0&0&0&0&0&0&0&0&0&2 \end{pmatrix}.$$

As mentioned in the previous examples, deleting the second through fourth row of this matrix should constitute a solution to Row Deletion(B), which is also the case this time as is easy to check.

4.3.4 Proof of Theorem 4.12

The proof of Theorem 4.12 will mainly be a generalization of the following Lemma 4.17 concerning a general observation for binary forbidden submatrices of size $r \times 2$. Lemma 4.17 is then generalized to any $r \times 2$ matrix by Lemma 4.18, from which we deduce Theorem 4.12.

Lemma 4.17 *Let B be a binary $r \times 2$ matrix. If, for a $\sigma \in \{0, 1\}$, B contains a column with d symbols different from σ, d-Hitting Set is parameter-preserving reducible to* Row Deletion*(B).*

$$B_\sigma = \begin{bmatrix} 1 & 1 \\ \vdots & \vdots \\ 1 & 1 \\ 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \\ \vdots & \vdots \\ 0 & 1 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \end{bmatrix} \begin{matrix} \} i_{(11)} \\ \\ \} i_{(10)} \\ \\ \} i_{(01)} \\ \\ \} i_{(00)} \end{matrix} \begin{matrix} \} r' \\ \\ \\ \\ \} r'' \end{matrix} \qquad V = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \begin{matrix} \} i_{(11)} \\ \\ \} i_{(10)} \end{matrix} \qquad W = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \} i_{(11)} \\ \\ \} i_{(10)} \end{matrix} \qquad U = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \} i_{(01)} \\ \\ \} i_{(00)} \end{matrix}$$

Figure 4.7: Sorted decomposition of a binary matrix B according to the proof of Lemma 4.17. To the right of B, the decomposition submatrices V, W, and U are shown.

Proof We will assume w.l.o.g. that we are given a maximal[13] σ-Decomposition $B_{\sigma\text{-max}}$ of B.[14] Furthermore, we assume w.l.o.g. that for the maximal decomposition, $\sigma = 0$.

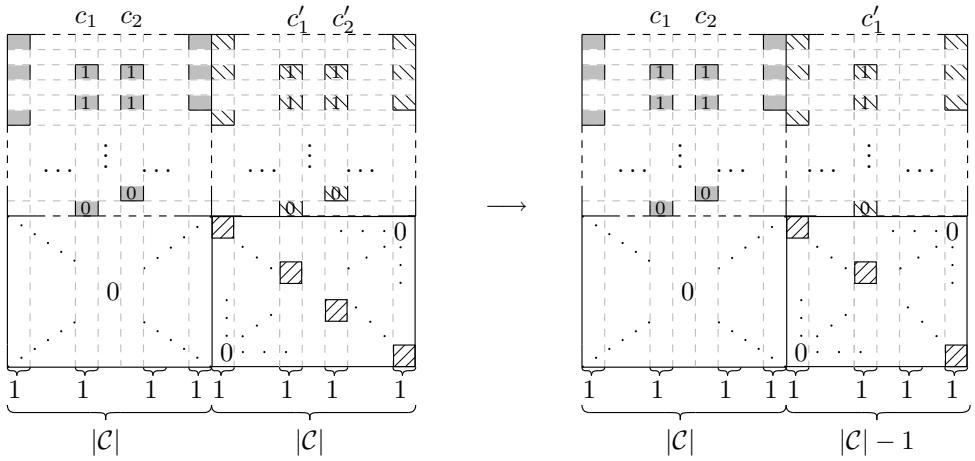
Note that since B_σ is only of width 2, the decomposition-matrices V, W, and U are all column vectors. The following observations can be made for B_σ:

- None of the two columns in B may contain a 0 more than r' times: Otherwise, we could choose $\sigma = 1$, obtaining a higher V in the corresponding decomposition of B—a contradiction to our assumption that the given decomposition is maximal.
- The right column may not contain more 1's than the left column in the decomposition because then the right part of B would be the encoding part in a maximal σ-decomposition of B.

The second observation leaves us with two cases to consider for the given σ-decomposition of B: Either, the right column contains less than r' or exactly r' 1's. The first case is already handled by Theorem 4.14, because if the second column in the decompositions contains less than r' 1's, it cannot induce V since V contains exactly r' 1's. Therefore, from now on we let B be a binary $r \times 2$ matrix where each row contains exactly r' entries equal to 1. Furthermore, we assume that B is sorted such that—from top to bottom—the first $i_{(11)}$ row vectors in B are equal to $(1\ 1)$, the next $i_{(10)}$ row vectors are equal to $(1\ 0)$, the next $i_{(01)}$ row vectors are equal to $(0\ 1)$, and the bottom $i_{(00)}$ row vectors are equal to $(0\ 0)$. The resulting matrix B_σ and the submatrices V, W, and U of such a sorted decomposition are shown in Figure 4.7.

[13] Recall Definition 4.6: This means that there is no $\sigma' \neq \sigma$ in $\mathcal{A}$ for which we can decompose B such that the decomposition-submatrix V is higher than the one in the decomposition for σ.

[14] Considering only one maximal σ-Decomposition can be justified as follows: Let r'_1 be the height of V in a maximal σ-Decomposition of a forbidden submatrix B, and let r'_2 be the height of V in any σ-Decomposition of B. Then, clearly, $r'_1 \geq r'_2$. Thus, by considering a maximal σ-Decomposition, we claim that r'_2-Hitting Set is at most as hard as r'_1-Hitting Set. This claim is easy to show: For any given instance $(\mathcal{C}, \mathcal{S}, k)$ of r'_1-Hitting Set, add $r'_2 - r'_1$ "dummy symbols" $\{d_{i,1}, \ldots, d_{i,r'_2 - r'_1}\}$ with $d_{i,j} \cap \mathcal{S} = \emptyset$ to every clause $C_i \in \mathcal{C}$ in order to obtain $\mathcal{C}'$. Set $\mathcal{S}' := \mathcal{S} \cup \{d_{i,j} \mid d_{i,j} \text{ is a dummy symbol}\}$. Then, clearly, $(\mathcal{C}', \mathcal{S}', k)$ is an instance of r'_1-Hitting Set that has a solution of size k if and only if $(\mathcal{C}, \mathcal{S}, k)$—as an instance of r'_2-Hitting Set—has one.



Matrix A' as generated by the scheme used in the proof of Theorem 4.11

Matrix A' after performing the merge operation for columns c'_1 and c'_2

Figure 4.8: An illustration for one step of the merge operation used in the proof of Theorem 4.12

Note that a) this is a maximal decomposition of B, b) $i_{(11)} + i_{(10)} = r'$, and that c) $i_{(10)} = i_{(01)}$. Additionally, we can assume that $i_{(11)} > 0$, because for the case $i_{(11)} = 0$, this lemma has already been proven by Case II of Theorem 4.14.

Given an instance $(\mathcal{C}, \mathcal{S}, k)$ of r'-Hitting Set where $\mathcal{S} = \{1, \ldots, n\}$. Then, the output matrix A that is a parameter-equivalent instance of Row Deletion(B) is constructed in two steps:

1. Use the construction employed in the proof of Theorem 4.11 to obtain a matrix A' from $(\mathcal{C}, \mathcal{S}, k)$ and B.

2. Perform the following "merge-operation" on A': While there are two columns in the right part of A' that induce B, arbitrarily choose one of these two columns and delete it (this operation is illustrated in Figure 4.8). Call the resulting matrix A.

Performing the merge-operation is justified as follows: Let two columns c_1 and c_2 in the left part of A' induce B with two columns c'_1 and c'_2 in the right part of A', respectively. If c'_1 and c'_2 induce B, they have to induce the row vector $(\,1\;1\,)$ exactly i_{11} times. Note that by means of construction for A', this row vector can only be induced in the upper part of A'. Note that additionally, each column in the upper right part of A' contains exactly i_{11} 1's. Hence, if c'_1 and c'_2 induce the row vector $(\,1\;1\,)$ exactly i_{11} times, their upper n entries must be identical. But then, observe how c_1 also induces B together with c'_2 and c_2 induces B together with c'_1. Therefore, after the merge-operation, c_1 and c_2 still induce B with some column in the right part of A' $(*)$.

We now claim that Row Deletion(B) has a solution of size k on A if the original r'-Hitting Set instance has a solution of size k. To prove this claim, the same arguments as in the proof of Theorem 4.11 can be employed due to $(*)$ provided we can show that there exists always an optimal solution to Row Deletion(B) on A that does not involve the deletion of any of the bottom $r'' \cdot |\mathcal{C}|$: In order to see this, note that the following holds true

after the merge-operation: If a set $M := c'_a, c'_b, \ldots$ was merged to a single column c_m, note that c_m contains r' 1's, some of which are to be found in the top n rows of A. Therefore, a B induced by c_m and some column in the left part of A that can be destroyed by deleting a 1 from the bottom $r'' \cdot |\mathcal{C}|$ rows of A can also be destroyed by deleting a 1 from c_m in the top n rows of A.

Since k is directly preserved by the reduction, we have proven the lemma. □

Let us illustrate the reduction in the above proof by the following example (the general scheme is illustrated in Figure 4.8): Let $\Sigma = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$ be given the forbidden submatrix . This is already a maximal σ-Decomposition with $V = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\sigma = 0$. For the reduction from 2-Hitting Set to Row Deletion(B) in this example, we shall use an instance of 2-Hitting Set where $\mathcal{C} = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,5\},\{3,4\}\}$, $\mathcal{S} = \{1,2,3,4,5\}$, and $k = 3$. Then A' is generated by the algorithm from Theorem 4.12, yielding

$$A' = \begin{pmatrix}
1&1&1&0&0&0&1&1&1&0&0&0\\
1&0&0&1&1&0&0&0&0&1&1&0\\
0&1&0&1&0&1&0&0&0&0&0&1\\
0&0&1&0&0&1&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0\\
0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&1
\end{pmatrix}.$$

After performing the merge-operation (the 7th, 8th and 9th column are merged and the 10th and 11th column are merged), we have

$$A = \begin{pmatrix}
1&1&1&0&0&0&1&0&0\\
1&0&0&1&1&0&0&1&0\\
0&1&0&1&0&1&0&0&1\\
0&0&1&0&0&1&0&0&0\\
0&0&0&0&1&0&0&0&0\\
0&0&0&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0\\
0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1
\end{pmatrix}.$$

Note how after the merge operation, each one of the six left columns induces the forbidden submatrix with exactly one of the three rightmost columns and that the three rightmost columns do not induce Σ by themselves.

We can easily generalize the above Lemma for larger alphabets:

Lemma 4.18 *Let B be an ℓ-ary $r \times 2$ matrix over the alphabet $\mathcal{A}$. If, for a $\sigma \in \mathcal{A}$, B contains a column with d symbols different from σ, d-Hitting Set is parameter-preserving reducible to Row Deletion(B).*

Proof In principle, we use the same reduction as in the proof of Lemma 4.17, again employing the merge-operation. The merge-operations correctness for an alphabet of size 2 or greater is justified as follows: In the proof of Lemma 4.17, we considered the i_{11} rows of B which induced the row vector $(1\ 1)$. Now, given a maximal decomposition B_σ of a forbidden submatrix of size $r \times 2$, let $i_{\not\sigma\not\sigma}$ be the number of rows in B_σ that do not contain σ. Then, if after the generation of A' according to Theorem 4.11, if B is induced by two columns c_1 and c_2 in the right part of A', the upper n rows of these columns must induce $i_{\not\sigma\not\sigma}$ row vectors that do not contain σ. But then c_1 and c_2's top n rows are bound to be identical, because they contain only $i_{\not\sigma\not\sigma}$ symbols different from σ and W is not permuted during the encoding process. Since the top n rows of c_1 and c_2 are identical if they induce B, we can perform the merge-operation. The rest of the argument is analogous to the proof of Theorem 4.11. □

A closer look at the proof of Lemma 4.18 shows that we can relax the conditions imposed upon B in that B need not be restricted to a width of 2 as long as V contains a column vector that is induced at most once in the non-encoding part of B. This directly leads to Theorem 4.12.

Proof of Theorem 4.12 Recall the reduction from the proof of Lemma 4.18. The key point was that after the reduction, B was not induced in the right part of A, designated A_r. According to the prerequisites for this Theorem, the submatrix V of height r' has a column vector v that is induced at most once in the non-encoding part of B_σ. Denote the $r \times 2$ submatrix of B_σ that contains the v induced in the encoding and the one induced in the non-encoding part of B_σ by $B_{\sigma,v}$. If $B_{\sigma,v}$ were the forbidden submatrix, we can—as in Lemma 4.18—reduce r'-Hitting Set to Row Deletion$(B_{\sigma,v})$. Now, consider the following reduction to encode an r'-Hitting Set instance into an Row Deletion(B) instance: We simply use the algorithm from Lemma 4.18 and $B_{\sigma,v}$ as the forbidden submatrix but, instead of only writing the entries of $B_{\sigma,v}$ into those rows of A determined by the algorithm, we write the respective parts of the whole matrix B_σ into A. □

The reduction used in the above proof is illustrated by the following example, very similar to the example used to illustrate Lemma 4.17: Given the forbidden submatrix $B = \left(\begin{smallmatrix} 1 & 2 & 3 & 1 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 3 & 1 \end{smallmatrix}\right)$. This is a maximal σ-Decomposition of B with $V = \left(\begin{smallmatrix} 1 & 2 \\ 1 & 2 \end{smallmatrix}\right)$ and $\sigma = 0$. Note that—as required by Theorem 4.12—the column vector $\left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right)$ in V is induced just once in the non-encoding part of B. Now, using the same instance[15] of 2-Hitting Set as in the illustration of Lemma 4.17, we generate A:

$$A' = \left(\begin{smallmatrix}
1&2&1&2&1&2&0&0&0&0&0&0&3&1&3&1&3&1&0&0&0&0&0&0\\
1&2&0&0&0&0&1&2&1&2&0&0&0&0&0&0&0&0&3&1&3&1&0&0\\
0&0&1&2&0&0&1&2&0&0&1&2&0&0&0&0&0&0&0&0&0&0&3&1\\
0&0&0&0&0&0&0&0&0&0&1&2&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&1&2&0&0&1&2&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&3&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1
\end{smallmatrix}\right).$$

Merging the appropriate columns, we obtain from this

$$A = \left(\begin{smallmatrix}
1&2&1&2&1&2&0&0&0&0&0&0&3&1&0&0&0&0\\
1&2&0&0&0&0&1&2&1&2&0&0&0&0&3&1&0&0\\
0&0&1&2&0&0&1&2&0&0&1&2&0&0&0&0&3&1\\
0&0&0&0&0&0&0&0&0&0&1&2&0&0&0&0&0&0\\
0&0&0&0&1&2&0&0&1&2&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&3&1
\end{smallmatrix}\right).$$

Notice how after the merge-operation, B is not induced in the right part of A (the six rightmost columns) but the inductions of B by columns in the left part of A' are preserved.

4.4 Discussion and Future Extensions

Theorems 4.11 to 4.14 have shown that for many forbidden submatrices, the following conjecture holds true:

Conjecture 4.19 *If r' is the height of the submatrix V in a σ-decomposition of the forbidden submatrix B, r'-Hitting Set is parameter-preserving reducible to Row Deletion(B).*

[15] $\mathcal{C} = \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,5\},\{3,4\}\}$, $\mathcal{S} = \{1,2,3,4,5\}$, and $k = 3$.

Although we have proven many special cases of this conjecture (which is more than sufficient for the discussion in this work), a general proof would provide a very interesting structural result. Ideas for a proof of the open problem would be an inductive extension of Lemma 4.17 for more than one induction of V in the non-encoding part of a σ-Decomposition of B. Another idea would be to find a proof for the intuitive statement that if B contains a submatrix B' for which Row Deletion(B') is known to be at least as hard as a certain d-Hitting Set, then Row Deletion(B) is as hard as d-Hitting Set as well. The problem in such proofs will most likely be to find a way to write W and U of a given σ-Decomposition into certain parts of A so that we can always be sure that an induction of V is inhibited in these parts. During the preparation of this work such a construction has been possible for many forbidden submatrices using the technique from Lemma 4.17 even if the forbidden submatrices did not fulfill the necessary prerequisites, so this should be a promising start for a possible proof of the conjecture.

Note that—in accordance with the above conjecture—Row Deletion(B) need not be NP-hard for all non-trivial B, as the matrix $B = \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ over the alphabet $\Sigma = \{0, 1\}$ shows. To see that this problem is solvable in polynomial time, observe that a B-free matrix A has the property that all its columns consist either solely of 1's or solely of 0's, i.e., all rows of A are identical.[16] This implies that the minimum number of rows that need to be deleted in order to make an $n \times m$-matrix B-free is equal to $n - x$, where x denotes the size of the largest set of identical rows in A, which can be determined in polynomial time.[17]

Another interesting area of research would be to close the relative hardness gap left by the results in this chapter: For an $r \times s$ forbidden submatrix B with an $r' \times s'$ submatrix V in the maximal σ-Decomposition, we have shown in Section 4.2.1 that Row Deletion(B) is not harder to solve than r-Hitting Set from a parameterized point of view. In this section we have conjectured that it is at least as hard to solve as r'-Hitting Set. But it seems likely that there are more efficient (especially parameterized) algorithms for r'-Hitting Set than for r-Hitting Set when $r' < r$—e.g., see Theorem 5.17 in the following chapter. It would be interesting to find out how this "gap" is closed and how this—if possible—can be related to the structure of B.

[16] This was pointed out by my advisor Jiong Guo.

[17] Note, however, that Row Deletion(B) for $B = \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ over the alphabet $\Sigma = \{0, 1\}$ already is NP-complete by Theorem 4.11, since B trivially has a 0-decomposition with $V = B$, and $W = U = \emptyset$.

Chapter 5

Perfect Phylogeny Problems

This chapter shows an application of the previous chapter's analysis of Row Deletion and Column Deletion problems in an area of biology commonly known as phylogenetics. We will first introduce the concept of phylogenetic trees, especially focusing on perfect phylogenies. A phylogenetic tree is a tree that depicts the evolutionary history from an imaginary "ancestral species" to a given set of species. This requires ordering the species depending on how closely they are related to each other (i.e., which sets of species have taken separate evolutionary pathways earlier than others). Ordering in a phylogeny is always done according to a presumed model—in our case this is *perfect phylogeny*, introduced in Section 5.2. We will then find out that the construction of a phylogenetic tree is only possible when—in a matrix representation of the input data—certain submatrices do not appear. As it turns out, the problem of avoiding these submatrices by removing species or some of their information from the data used to construct a perfect phylogeny, are exactly the Row Deletion and Column Deletion problems analyzed in Chapter 4.

5.1 Phylogenetic Trees

5.1.1 Introduction and Motivation

Since Darwin introduced the theory of evolution, it has always been the desire of biologists to infer the ancestral relationships of present-day species. Given a collection of species, a *phylogenetic analysis* will try to determine their evolutionary relationship. This is done by constructing a tree that displays the process of evolution as a sequence of branching events; i.e., a common ancestor is divided into distinct species by a speciation event. A good general introduction to (computational) phylogeny may be found, e.g., in [DEKM98], [Fels03], and [SeSt03]. Besides for applications like those mentioned in Section 2.3, the study of phylogenetics is important to research on—often fundamental—questions in areas such as conservation genetics, epidemology, ecology, medicine, and even non-biological fields of research such as linguistics [SeSt03]. Before the availability of molecular data, the inference of phylogenies was based on physical characteristics expressed by species.[1] However,

[1] As we will soon see, a group of species due to common physical characteristics is called a *clade*. There are some views—such as in [KFHW98]—that the resulting trees of a clade-based analysis should be referred to as a *cladogram* to distinguish it from an evolutionary tree which, in addition, contains an implicit time axis of speciation events.

the observation of physical characteristics is often quite misleading: E.g., in 1952, Robertson and Reeve [RoRe52] performed a selection experiment to change the wing size in two *Drosophila* populations. However, a closer examination revealed that whereas in one population the number of cells per wing increased, the other population developed larger wing cells, demonstrating a case of *homoplasy*[2]. Of course, a close examination of the wings would have shown that obviously different genetic markups must be responsible for the expression of a common characteristic, but there are more subtle examples. The main point is that physical characteristic are often subject to change due to selection processes and therefore, although some species may show the same characteristics, this does not imply a close ancestral relationship. As Page and Holmes stated in [PaHo98]: "*homoplasy is a poor indicator of evolutionary relationships, because similarity does not reflect shared ancestry.*"

According to [SeSt03], the field of phylogenetics "*was revolutionized by the arrival of molecular data*" which allows a much better distinction of different evolutionary pathways that have led to the expression of different characteristics: Each such pathway contains a multitude of "genetic traces" called *markers* that may serve as an indicator for common or separate paths of evolution. However, even with genetic data available, there are still problems phylogenetics has to deal with. For example, the problem of homoplasy as in the *Drosophila* experiment of Robertson and Reeve can also be encountered at a genetic level if proteins for similar functions have evolved in different species. The most promising markers today for phylogenetic analysis seem to be SNPs[3], as has already been mentioned in Chapter 2. Since SNPs show very slow rates of mutations and the area around them is often highly conserved, they are very well suited as a basis for inferring ancestral relationships; most of the time, only two different bases at a SNP site can be found throughout a population, which is then a good indicator for speciation events and common ancestral relationships.

This work will only be concerned with computational problems arising from the study of so-called perfect phylogenies (a more thorough introduction to this model follows in the next subsection), however, there is a multitude of other approaches such as "maximum likelihood" or "minimum parsimony". A more thorough introduction to these and some more distantly related topics can be found, e.g., in [KFHW98] and [PaHo98], which are not as much focused on computational biology as the references already recommended above.

5.1.2 Formal Definition

This subsection introduces the predominant phylogenetic model for this chapter, the species-character model[4], which leads us to perfect phylogenies. A good survey article of problems related to perfect phylogeny is, e.g., [Fern01]. In the species-character model, each species is described by m different *characters*. Each character may take one of up to ℓ different *states*. Let $\sigma(i,j) \in \{0,\ldots,\ell-1\}$ be the state of the jth character for the species s_i. Each species is characterized by the states of its m characters, we therefore introduce a *character vector* for each species that is a row vector containing all the characters of a certain species.[5] For convenience, the input data may be written as an $n \times m$ matrix $A = (a_{i,j})$ with $a_{i,j} = \sigma(i,j)$,

[2]Formally, the term "homoplasy" refers to a correspondence between parts of an organism acquired due to parallel evolution or convergence.

[3]An indication for the importance of SNPs is that almost every article on evolutionary relationships in widely recognized scientific magazines such as *Nature* or *Science* obtains its results from the analysis of SNPs.

[4]Some literature also refers to this model as the *cladistic model*, where "clade" is defined as a group of biological species that includes all descendants of a common ancestor.

[5]For example, if, for a species s_i, we have $\sigma(i,1) = 1$, $\sigma(i,2) = 2$, $\sigma(i,3) = 3$, $\sigma(i,4) = 2$, and $\sigma(i,5) = 1$, its character vector would be (1 2 3 2 1).

the *character-state matrix*. Each row r_i in A corresponds to the species s_i and the column c_j to the jth character of all species. For example, if we have three species s_1, s_2, and s_3 with respective character vectors $(\,0\;9\;7\;0\;1\,)$, $(\,9\;5\;1\;4\;2\,)$, and $(\,3\;6\;4\;8\;5\,)$, we would write

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

Note that there are variants of phylogenetic problems where not all entries of A are known (i.e., some states of some characters are missing). These entries will be represented by a "?" character.[6]

The phylogenetic tree we seek to construct from the given species and character data is called a *perfect phylogeny*, which is the following tree-graph:

Definition 5.1 (Perfect Phylogeny)
Given a set $\mathcal{S}_{leaf}$ of i species, each of which is described by exactly one of ℓ different states for each of its m characters. Let $\mathcal{S}_{internal}$ be a set of "ancestral" species (disjoint with $\mathcal{S}_{leaf}$) for which we may assign character states. We then set $\mathcal{S} := \mathcal{S}_{leaf} \cup \mathcal{S}_{internal}$. A tree $T = (V, E)$ is called a perfect phylogeny if it fulfills the following properties:

1. *T has $|\mathcal{S}_{leaf}|$ leafs. There exists a bijection $\Phi : \mathcal{S} \to V$ between the species in $\mathcal{S}$ and the vertices of T such that every species from $\mathcal{S}_{leaf}$ is mapped to a leaf in T and every ancestral species is mapped to an internal node.*

2. *Let $\mathcal{S}_{j\lambda} \subseteq \mathcal{S}$ be the set of species for which the jth character takes state λ. Then, for every $1 \leq j \leq m$ and $0 \leq \lambda < l$, the set $\{\Phi^{-1}(s) | s \in \mathcal{S}_{j\lambda}\}$ of vertices forms a connected component in T (in other words, all species that share a common state for a certain character induce a subtree in T).*

An example for a set of species and a perfect phylogeny for their characteristics is given in Figure 5.1.

In the introduction to this chapter, we have already mentioned the problem of homoplasy in phylogenetic analysis. In principle, a perfect phylogeny is a phylogenetic model that does not allow for homoplasy, leading to the elegant mathematical descriptions that we have seen. However, as Fernández-Baca observes in his survey [Fern01], "*elegance comes at a price*", and there are only very few examples in biology and linguistics where perfect phylogenies occur naturally (e.g., see [NRO99] and [BLM03]). However, if we assume that only a very few species or characters are violating a cladistic model in a given set of data, the analysis from Section 5.4 onwards in this chapter will provide efficient algorithms and a complexity analysis for finding and removing a minimal set of violating parts of the data.

Perfect Phylogeny is a special case of the so-called Character Compatibility problem, introduced in [MeEs85]: For this problem, every character that satisfies condition 2 of the above definition ("all species that share a common state for a certain character induce a subtree in T") is called "true" (following [Esta78]). The Character Compatibility problem then asks for a *maximum* number of characters to be true as opposed to Perfect Phylogeny, where *all* characters must be true.

[6] The problems associated with incomplete input data are usually harder than their equivalents with complete data. E.g., the Undirected Perfect Phylogeny problem that will be introduced shortly is already *NP*-complete for binary characters when the state of some characters is unknown [Stee92].

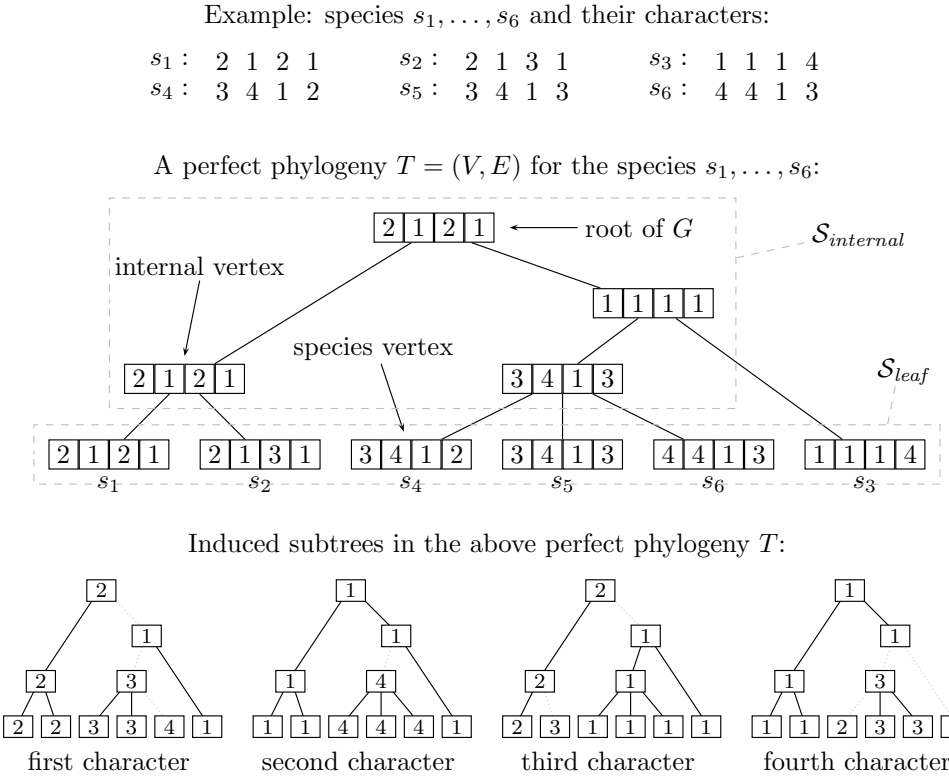


Figure 5.1: An example of a perfect phylogeny for a set of species: For six species s_1 through s_6, four characters and their states are given. The tree $T = (V, E)$ then represents a perfect phylogeny for these species (notice how all species that share a common state for a certain character induce a subtree in T). The leafs in T each correspond to a species from the set $\mathcal{S}_{leaf}$ of species, the internal vertices to the set $\mathcal{S}_{internal}$ of "ancestral" species. Sometimes in literature, just the root of T is referred to as being *the* "ancestral species".

5.2 Perfect Phylogeny Problems

There is quite a multitude of perfect phylogeny problems (see, for example, [BFW92], [Gusf91], [PSS02$_a$], [PSS02$_b$]). This section introduces these problems, giving an overview of the most important variants and related results concerning the computational complexity. As we will see, this complexity is mainly determined by the maximum number of different states that characters may take.

The most general formulation of perfect phylogeny problems is k-Perfect Phylogeny. The input for a k-Perfect Phylogeny problem consists of a set $\mathcal{S} = \{s_1, \ldots, s_n\}$ of n *species*, each described by m different *characters* where each character may take one of k different *states*. As was already explained on page 57, this instance is often given as a k-ary $n \times m$ matrix A. Seeing k as the classifying parameter, we directly obtain the definition of the k-Perfect Phylogeny problem:

Definition 5.2 (k-Perfect Phylogeny)
Input: *Given a set $\mathcal{S}$ of n species, each of which described by exactly one of k different*

states for each of its m characters.
Question: *Does there exist a perfect phylogeny for the species in $\mathcal{S}$?*

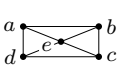
Buneman demonstrated in [Bune74] that this problem is equivalent to a graph-theoretic problem called Triangulating Colored Graphs. A graph is called *triangulated* if there is no cycle of four or more vertices in G that does not induce a cycle of size 3 as a subgraph.[7] It is obvious that every graph can be triangulated by adding enough edges to it. However, the Triangulating Colored Graphs problem asks whether a given graph G, where each vertex is associated with one of k colors, can be triangulated by adding edges such that the resulting graph has *no edges between vertices of equal color*. If this is possible, the resulting graph is called *properly colored triangulated*. We shall only sketch the idea behind the reduction from k-Perfect Phylogeny to Triangulating Colored Graphs by Buneman, more details may be found, e.g., in [Bune74] or [Fern01]. For the proof, the character-state matrix A is first transformed into a so-called *character-state intersection graph* G_A. In this graph, each entry a_{ij} of A corresponds to a vertex, the color of the respective vertex represents the one of k states found for the ith species at its jth character in A. Two vertices are connected by an edge if their represented character states occur together in a species. A theorem by Gavril [Gavr74] states that a triangulated graph is the intersection graph of a family of subtrees of a tree, leading to the result that a properly colored triangulated intersection graph represents a phylogenetic tree.

The equivalence of k-Perfect Phylogeny and Triangulating Colored Graphs later allowed Bodlaender *et al.* to show that the k-Perfect Phylogeny problem is *NP*-complete if k is not fixed to a constant [BFW92]. If k is constant, for $k = 3$ k-Perfect Phylogeny is solvable in polynomial, i.e., $O(nm^2)$ time according to [DrSt92]; for $k = 4$, [KaWa94] provides an $O(n^2m)$ algorithm. Seeing k as a parameter, k-Perfect Phylogeny is fixed-parameter tractable and can be solved in $O(4^k nm^2)$ time [KaWa97] (a big improvement over a previous algorithm stated in [AgFe93]).

Along with the *NP*-completeness proof for k-Perfect Phylogeny in [BFW92], another result concerning k-Perfect Phylogeny on graphs with *bounded treewidth* was shown. Many hard graph problems become polynomially-time solvable on tree graphs. For some of these problems, this property can be used to develop efficient algorithms on graphs with bounded treewidth.[8] Roughly speaking, the treewidth of a graph is a measure of how "treelike" the structure of a graph is. A *tree-decomposition* of a graph is a rooted tree representation of the graph, that divides it into some subgraphs, the treewidth then becomes a measure for the size of these subgraphs. If a problem is finite-state for bounded treewidth, it means that we can use the tree-decomposition of a graph to efficiently solve a hard problem on it by computing partial solutions for the children of each node of the tree starting from the leafs, using the fact that a solution computed for the two children of an inner node (which is from a *finite* set of possible solutions, hence the term *finite* state) can be used to efficiently arrive at a solution for the inner node itself. However, it was shown in [BFW92] that k-Perfect Phylogeny is not finite-state for bounded treewidth if $k \geq 4$. This result implies that for $k \geq 3$, these dynamic programming strategies are not applicable.

In this work, we will mainly deal with the 2-Perfect Phylogeny problem and therefore use Definition 5.3 for means of abbreviation:

[7] e.g., the graph 

is triangulated whereas 

is not because the vertices a, b, c and d induce a cycle of size 4 which by itself contains no cycle of size 3 as a subgraph.

[8] For a more detailed survey on this topic, see, e.g., [Bodl97] or [Bodl88].

Definition 5.3 (PERFECT PHYLOGENY)
Input: *An instance of* 2-PERFECT PHYLOGENY
Question: *Does there exist a perfect phylogeny for the species in* $\mathcal{S}$*?*

The PERFECT PHYLOGENY problem[9] may be reformulated as follows: For a set of species with binary character states we wish to construct a phylogenetic tree, such that for each character, there exists at most one edge that partitions the tree into two subtrees—one of which has state 1, the other one state 0 for that character in every one of its nodes. Many times in literature, the following variant of PERFECT PHYLOGENY is discussed.

Definition 5.4 (DIRECTED PERFECT PHYLOGENY)
Input: *An instance of* 2-PERFECT PHYLOGENY
Question: *Does there exist a perfect phylogeny for the species in* $\mathcal{S}$ *such that the root is the all-zero vector?*

The main difference between (UNDIRECTED) PERFECT PHYLOGENY and DIRECTED PERFECT PHYLOGENY lies in the fact that in DIRECTED PERFECT PHYLOGENY, we allow only changes from 0 to 1 (i.e., the gaining of characters) throughout evolution. Gusfield demonstrates in [Gusf91] that the DIRECTED PERFECT PHYLOGENY problem is by far easier than general k-PERFECT PHYLOGENY to solve, it is in fact linear-time solvable with respect to the size of the input matrix A (he furthermore gives a proof that this algorithm is time-optimal). Since a linear time reduction from DIRECTED PERFECT PHYLOGENY to PERFECT PHYLOGENY will follow from Theorem 5.7, (2-)PERFECT PHYLOGENY is therefore also linear-time solvable.

5.3 Relation to Forbidden Submatrix Problems

This section will show that we can easily determine whether a given binary species-character matrix A implies that the represented species have evolved according to a perfect phylogeny. This will be done by testing for the induction of a certain submatrix that must not be induced in a perfect phylogeny. If we determine that the species have not evolved according to a perfect phylogeny but assume that only a few characters or species in the dataset represented by A are responsible for this, we might try to delete as few rows or columns from A as possible (which is equivalent to removing species or characters from the dataset represented by A) in order to find out which portion of the data fits a perfect phylogeny. This is a submatrix removal problem, leading us to the application of the results from Chapter 4 in the next section.

It is easy to determine whether the data represented by a given species-character matrix A allows for the construction of a perfect phylogeny by testing for the induction of the so-called Σ-matrix in A.

Definition 5.5 (Σ-MATRIX): *A binary matrix A is called* Σ-matrix[10]*, if it is a permutation*

[9] Sometimes PERFECT PHYLOGENY is called UNDIRECTED PERFECT PHYLOGENY in order to distinguish it from the DIRECTED PERFECT PHYLOGENY problem we shall introduce shortly.

[10] The name for this matrix originates from [PSS02$_a$]: When a binary $n \times m$ character-state matrix A is interpreted as the adjacency-matrix of a bipartite graph $G = \{V_1, V_2, E\}$, where $V_1 = \{c_1, \ldots, c_n\}$, $V_2 = \{s_1, \ldots, s_m\}$ and c_i is connected with s_j if and only if $a_{ij=1}$, the presence of a Σ matrix leads to the structure  (with the left vertices representing characters and the right vertices states) in the graph implicated by A, which looks like a mirrored Σ.

of the 3×2 *matrix* $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$*, i.e. it is equal to one of the matrices*

$$\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{smallmatrix}\right), \left(\begin{smallmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{smallmatrix}\right), \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{smallmatrix}\right), \left(\begin{smallmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{smallmatrix}\right), \left(\begin{smallmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{smallmatrix}\right), \text{ or } \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{smallmatrix}\right).$$

The following theorem has been proven independently by many authors :

Theorem 5.6 *[For a proof see, e.g., [Meac83] and [EJM75]] Given a binary character-state matrix A. There exists a directed perfect phylogeny for the species represented in A if and only if A is* Σ*-free.* □

Comparing Directed Perfect Phylogeny to Perfect Phylogeny, it is a very strong assumption that characters as we have labeled them have only been gained throughout evolution. This assumption is bound to fail often as labeling of the character states as 0 and 1 is arbitrary.[11] When given m characters, the general Perfect Phylogeny problem would allow 2^m choices for the character states in the root. Fortunately, if we do not care about relabeling of the state labels, the following theorem by McMorris (see page 135 of [Morr77] for the proof[12]) simplifies the choice of a root vector to a task that may be performed in $O(mn)$ time, i.e., linear time with respect to the size of the binary input matrix A.

Theorem 5.7 *[Morr77] If there is a perfect phylogeny for a binary matrix A, there is a perfect phylogeny for A where each character in the root takes the state that the majority of individuals takes for that character.* □

We can then simply invert all those characters where the proposed root contains a 1 to obtain a Directed Perfect Phylogeny instance from a Perfect Phylogeny one. However, this inversion is only possible if certain conditions are satisfied by the input matrix A as shall be explored in the next theorem. For this theorem, we will rely on the following notation: We denote by $\mathcal{L}(M)$ an operation on a binary matrix M with two columns. The operation $\mathcal{L}(M)$ inverts the left column. Analogously to this we define the operations $\mathcal{R}(M)$ (inverting the right column) and $\mathcal{LR}(M)$ (inverting the whole matrix). We immediately observe that if M_Σ is a Σ-matrix, then $\mathcal{L}(M_\Sigma)$, $\mathcal{R}(M_\Sigma)$, and $\mathcal{LR}(M_\Sigma)$ are not Σ-matrices. However, the restrictions of the following theorem apply.

Definition 5.8 (Extended Σ-Matrix): *A binary matrix A is called* extended Σ-matrix*, or* EΣM *for short, if it is a permutation of the* 4×2 *matrix* $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{smallmatrix}\right)$.

Theorem 5.9 *Let A be a binary matrix with two columns that induce a* Σ*-matrix. Then A can be made* Σ*-free by column inversion if and only if those two columns do not induce an extended* Σ*-matrix.*

Proof Let $S(M)$ be the set of induced row-vectors in a binary matrix M. Since A induces a Σ-matrix, there is a submatrix M induced by two columns of A that contains the row-vectors $(1,1)$, $(1,0)$ and $(0,1)$. If M does not induce the row-vector $(0,0)$, then

$$S(\mathcal{L}(M)) = \{(0,1),(0,0),(1,0)\},$$

[11] I.e., losing a trait may also be considered as a "gain" of that particular loss in evolutionary history.

[12] Note that as in most older books concerning perfect phylogenies, the terminology and notation used is quite different from the one used today and in this work.

$$S(\mathcal{R}(M)) = \{(1,0),(1,1),(0,0)\} \text{ and}$$

$$S(\mathcal{LR}(M)) = \{(0,0),(0,1),(1,0)\}.$$

None of the above S contains all row vectors of a Σ-matrix—this immediately implies that M can be made Σ-free by any of the three inversion operations. If, however, $S(M) = \{(0,0),(0,1),(1,0),(1,1)\}$, then

$$S(\mathcal{L}(M)) = \{(1,0),(1,1),(0,0),(0,1)\},$$

$$S(\mathcal{R}(M)) = \{(0,1),(0,0),(1,1),(1,0)\} \text{ and}$$

$$S(\mathcal{LR}(M)) = \{(1,1),(1,0),(0,1),(0,0)\}$$

meaning that M—and therefore A—cannot be made Σ-free by inversion-operations on its columns. □

The question for a minimal column inversion does not seem to be very reasonable: If it is clear whether a trait has been gained or lost during the evolution of the given species, an inversion of the respective column would not be appropriate with respect to its biological implications—inversion of the character labels should come at no cost.

Having seen that a matrix cannot be made Σ-free just by column inversion alone if it induces an EΣM, the next two sections of this chapter will employ results from Chapter 4 to analyze the computational complexity of removing a minimum number of rows or columns from a given binary matrix A such that the resulting matrix is EΣM-free.

5.4 Minimum Species Removal

When constructing a perfect phylogeny for a set of species—assuming that these have evolved according to a perfect phylogeny—there might be a few species in the dataset that prevent such a construction. These species will therefore cause the induction of EΣMs in the species-character matrix, and must therefore be removed from the dataset in order to permit the construction of a perfect phylogeny for the remaining individuals. Since we assume that the given species actually *have* evolved in a perfect phylogeny, we require this removal to consist of as few species as possible.

Definition 5.10 (Minimum Species Removal):
Input: *A binary $n \times m$ matrix M and an integer k.*
Question: *Is it possible to delete at most k rows in M so that the resulting matrix is EΣM-free?*

As we can see from Theorem 5.9, the species must not induce any EΣMs in the character-species matrix A, since species/genotypes are rows in A, Minimum Species Removal is a Row Deletion problem with the EΣM as its forbidden submatrix. This immediately leads to the following result:

Theorem 5.11 2-Hitting Set *is parameter-preservingly reducible to* Minimum Species Removal.

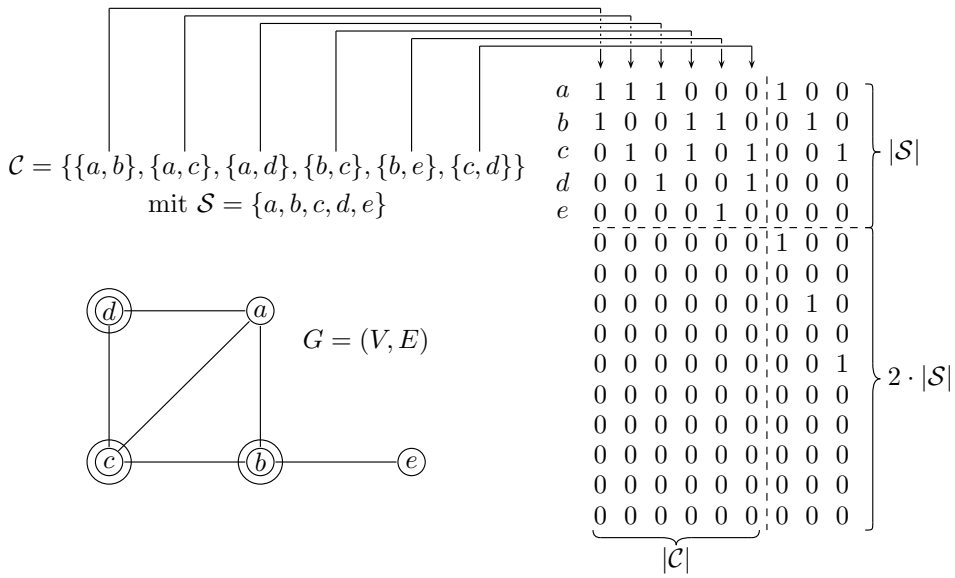


Figure 5.2: Parameterized reduction of 2-Hitting Set (Vertex Cover) to Minimum Species Removal. An instance of 2-Hitting Set is given as collection $\mathcal{C}$ of two-element subsets of $\mathcal{S}$, for illustration purposes, the equivalent Vertex Cover-graph G with an optimal solution is also shown. The matrix on the right is an equivalent instance of Minimum Species Removal to the given 2-Hitting Set instance.

Proof Follows directly from Theorem 4.12: Choosing 0 as σ, $\left(\begin{smallmatrix}1&1\\1&0\\0&1\\0&0\end{smallmatrix}\right)$ is a maximal σ-decomposition for the EΣM where $V := \left(\begin{smallmatrix}1\\1\end{smallmatrix}\right)$ is induced only once in the non-encoding part (right column) of the decomposition. Since V has height 2, this theorem follows from Theorem 4.12. □

Figure 5.2 gives an example for a reduction of 2-Hitting Set to Minimum Species Removal. The above theorem also proves that Minimum Species Removal is *NP*-hard. Furthermore, Minimum Species Removal is *NP*-complete and fixed-parameter tractable by using Theorem 4.7.

Theorem 5.12 Minimum Species Removal *can be parameter-preservingly reduced to* 4-Hitting Set.

Proof Follows directly from Theorem 4.7. since the EΣM has four rows. □

Note that this reduction does not directly yield an efficient algorithm for solving Minimum Species Removal—a trivial search tree for 4-Hitting Set has size $O(4^k)$, which grows quite rapidly. A much better algorithm is given in [NiRo03$_a$] which implies a search tree size of $O(3.30^k)$. However, Minimum Species Removal can be solved even faster already by a trivial algorithm, as the next theorem—Theorem 5.17—will show. For the proof of this theorem, we will first introduce two useful lemmata and some new notations.

In order to better analyze the size of a search tree, one can use the mathematical tool of *recurrence analysis*. A detailed introduction to the analysis of recurrences can be found,

e.g., in [GKP94] and [GrKu90]. For our purposes, the following is crucial: Note that in the search tree, each time we traverse an edge, k is decreased by one. For some problems, however, it is possible to decrease k by more than one. This leads to the general notion of a *branching vector*, sometimes also referred to as a *branching tuple*:

Definition 5.13 (Branching Vector)
If every node of a search tree T branches into one of s different subcases, and for each subcase, we reduce the parameter by a respective $d_i > 0$, we call $b := (d_1, \ldots, d_s)$ the branching vector *for T.*

The following result from recurrence analysis states that we can use the branching vector to determine the size $\tau(b)$ of the corresponding search tree:

Lemma 5.14 *[Titc76] If a search tree has a branching vector $b := (d_1, \ldots, d_s)$, its total size is $\tau(b) = O(|\alpha|^k)$, where α is the solution of $z^d - z^{d-d_1} - \cdots - z^{d-d_s} = 0$ (where $d = \max_i d_i$) with the largest absolute value (largest root, for short). In our context, this root is real.* □

For example, the trivial search tree for solving Vertex Cover in Section 3.3 branches into two subcases at each node, each time reducing the parameter by one. Therefore, we have $s = 2$ and $d_1 = d_2 = d = 1$. The value of α is therefore the largest root of $z^1 - z^0 - z^0 = 0 \Rightarrow z - 2 = 0$, which is obviously 2. So, according to the above lemma, the search tree for our trivial Vertex Cover algorithm has size $O(2^k)$, in accordance with the observations from Section 3.3. A second lemma by Kullmann (Lemma 8.5 of [Kull99]) allows us to make a statement about the relative size of two trees by comparing their branching vectors.

Lemma 5.15 *[Kull99] Let there be given two branching vectors $b_1 = (d_1, \ldots, d_s)$ and $b_2 = (d'_1, \ldots, d'_s)$. If $\min(d_1, d_2) > \min(d'_1, d'_2)$, $d_1 + d_2 \leq d'_1 + d'_2$, and for every $i > 2$, we have $d_i = d'_i$, then $\tau(b) < \tau(b')$. This also holds true for any two positions that differ in two branching vectors if all other positions are equal. Furthermore, $\tau(b)$ remains constant when b is permuted.* □

Corollary 5.16 *Given a branching vector $b = (d_1, \ldots, d_s)$. Then, $\tau(b) < \tau(b')$ where b' is the the branching vector $(\sum_{i=1}^{s} d_i - (s-1), \underbrace{1, \ldots, 1}_{s-1})$.*

Proof Observe that if $a \geq b > 1$, we have $\tau(\ldots, a+1, \ldots, b-1, \ldots) > \tau(\ldots, a, \ldots, b, \ldots)$ due to Lemma 5.15, because then $\min(a+1, b-1) < \min(a, b)$ and $(a+1)+(b-1) = a+b$. W.l.o.g., we can assume that $d_1 = \max_i d_i$ because Lemma 5.15 allows us to permute the branching vector without changing the value of the respective τ.

From this observation we can see by repeated application of Lemma 5.15 that

$$\tau(\sum_{i=1}^{s} d_i - (s-1), 1, 1, \cdots, 1) > \tau(\sum_{i=1}^{s} d_i - (s-1) - 1, 2, 1, \cdots, 1) >$$

$$> \tau(\sum_{i=1}^{s} d_i - (s-1) - 2, 3, 1, \cdots, 1) > \cdots > \tau(\sum_{i=1}^{s} d_i - (s-1) - (d_2 - 1), d_2, 1, \cdots, 1) >$$

$$> \cdots > \tau(\sum_{i=1}^{s} d_i - (s-1) - (d_2 - 1) - (d_3 - 1), d_2, d_3, \cdots, 1) > \cdots >$$

$$> \tau(\sum_{i=1}^{s} d_i - (s-1) - \sum_{i=2}^{s} (d_i - 1), d_2, d_3, \cdots, d_s) =$$

$$= \tau(d_1 - (s-1) + (s-1), d_2, d_3, \cdots, d_s) = \tau(d_1, d_2, d_3, \cdots, d_s).$$

Note that this lemma demonstrates how a tree gets larger the more "unbalanced" it is. □

Using the last corollary and Lemma 5.14, we can now show that the trivial search tree algorithm for MINIMUM SPECIES REMOVAL has a search tree of size $O(3^k)$.

Theorem 5.17 *There exists a fixed-parameter algorithm for* MINIMUM SPECIES REMOVAL *with a search tree of size* $O(3^k)$.

Proof Let (A, k) be a given instance of MINIMUM SPECIES REMOVAL. We will construct a simple algorithm for solving MINIMUM SPECIES REMOVAL, analogously to the one that was used to solve VERTEX COVER in Section 3.3: Looking for EΣMs in the input matrix A, each time an induction of an EΣM is encountered in two columns c_1 and c_2 of A, the algorithm branches into four cases:

I. Delete all i_{11} rows in A that induce the row vector $(1\ 1)$ in c_1 and c_2, obtaining A'. Then, proceed with the algorithm on $(A', k - i_{11})$.

II. Delete all i_{10} rows in A that induce the row vector $(1\ 0)$ in c_1 and c_2, obtaining A'. Then, proceed with the algorithm on $(A', k - i_{10})$.

III. Delete all i_{01} rows in A that induce the row vector $(0\ 1)$ in c_1 and c_2, obtaining A'. Then, proceed with the algorithm on $(A', k - i_{01})$.

IV. Delete all i_{00} rows in A that induce the row vector $(0\ 0)$ in c_1 and c_2, obtaining A'. Then, proceed with the algorithm on $(A', k - i_{00})$.

The corresponding search tree has size $\tau(i_{11}, i_{10}, i_{01}, i_{00})$. From Corollary 5.16, we know that

$$\tau(i_{11}, i_{10}, i_{01}, i_{00}) < \tau(i_{11} + i_{10} + i_{01} + i_{00} - 3, 1, 1, 1)$$

Note that for any two columns c_1 and c_2 in the input matrix A, any row vector induced by these two columns is either $(1\ 1)$, $(1\ 0)$, $(0\ 1)$, or $(0\ 0)$. Therefore, $i_{11} + i_{10} + i_{01} + i_{00} = n$ (where n is the height of A) and

$$\tau(i_{11}, i_{10}, i_{01}, i_{00}) < \tau(n - 3, 1, 1, 1)$$

By using Lemma 5.14, we can deduce from this that the size of the trivial search tree is bounded by $O(\alpha^k)$, where α is the largest root of

$$z^{n-3} - z^{n-3-(n-3)} - z^{n-3-1} - z^{n-3-1} - z^{n-3-1} = 0 \Longrightarrow z^{n-3} - 3z^{n-4} - 1 = 0$$

The largest real root of this polynomial is smaller than $3 + \frac{1}{3^{n-4}}$ because for $z > 3 + \frac{1}{3^{n-4}}$, we have

$$z^{n-3} - 3z^{n-4} - 1 > \left(3 + \frac{1}{3^{n-4}}\right)^{n-3} - 3\left(3 + \frac{1}{3^{n-4}}\right)^{n-4} - 1 =$$

$$= \left(\frac{3^{n-3}+1}{3^{n-4}}\right)^{n-3} - 3\left(\frac{3^{n-3}+1}{3^{n-4}}\right)^{n-4} - 1 = \left(\frac{3^{n-3}+1}{3^{n-4}}\right)^{n-4} \cdot \left(\frac{3^{n-3}+1}{3^{n-4}} - 3\right) - 1 =$$

$$= \left(\frac{3^{n-3}+1}{3^{n-4}}\right)^{n-4} \cdot \left(\frac{3^{n-3}+1-3^{n-3}}{3^{n-4}}\right) - 1 = \left(\frac{3^{n-3}+1}{3^{n-4}}\right)^{n-4} \cdot \frac{1}{3^{n-4}} - 1 =$$

$$= \left(\frac{3^{n-3}+1}{3^{n-3}}\right)^{n-4} - 1 = \underbrace{\left(1+\frac{1}{3^{n-3}}\right)^{n-4}}_{>1} - 1 > 0$$

Now, assuming $k \leq n-4$,[13] we obtain

$$\alpha < 3 + \frac{1}{3^{n-4}} \leq 3 + \frac{1}{3^k}$$

Using $k > 0 \Longrightarrow \frac{k}{3^{k+1}} < 0.125$ $(*)$ and the well-known inequality $0 < i < 1, n > 0 \Longrightarrow (1+i)^n \leq e^{i \cdot n}$ $(**)$, we obtain from this

$$\alpha < 3 + \frac{1}{3^k} = 3 \cdot \left(1 + \frac{1}{3^{k+1}}\right) = 3 \cdot \left(\left(1 + \frac{1}{3^{k+1}}\right)^k\right)^{\frac{1}{k}} \overset{(**)}{\leq}$$

$$\leq 3 \cdot \left(e^{\frac{k}{3^{k+1}}}\right)^{\frac{1}{k}} \overset{(*)}{<} 3 \cdot \left(e^{0.125}\right)^{\frac{1}{k}} < 3 \cdot 1.14^{\frac{1}{k}}.$$

This implies a search tree size of $O(\left(3 \cdot 1.14^{\frac{1}{k}}\right)^k) = O(3^k \cdot 1.14) = O(3^k)$. □

As we have seen, Minimum Species Removal can be solved using a search tree of size $O(3^k)$—this narrows the gap between the minimum hardness boundary for Minimum Species Removal that has been shown at the beginning of this section[14] and the best algorithm for Minimum Species Removal. However, more research would be needed to determine whether this gap can be narrowed even further to bring the computational complexity of Minimum Species Removal even closer to Vertex Cover, which we have proven to be a lower computational complexity bound for this problem.

5.5 Minimum Character Removal

Instead of looking for the minimum number of species that prevent the construction of a phylogeny from the given dataset, one might also be interested in the minimum number of sites responsible for these conflicts (by inducing EΣMs). This problem is formally stated in Definition 5.18, using n as the number of species and m as the number of considered characters for each species.

Definition 5.18 (Minimum Character Removal):
Input: *A binary $n \times m$ matrix A and an integer k.*
Question: *Is it possible to delete at most k columns in A so that the resulting matrix is EΣM-free?*

[13] Observe that if $k > n-4$, we can use an $O(n^3)$ algorithm to try all combinations of three rows that are *not* deleted from the input matrix in order to make it Σ-free.

[14] Recall that this was the Vertex Cover, or 2-Hitting Set problem.

Using the general framework given in Section 4.2.1, the following algorithm efficiently finds all pairs of columns in a given $n \times m$ input matrix A that induce an EΣM in $O(n^2m)$ time, leading to an instance of 2-Hitting Set. For the algorithm, let c_i denote the ith column of the input matrix A.

Algorithm: Finding all EΣM inducing columns
Input: A binary $n \times m$ matrix A
Output: A set $\mathcal{C}$ containing all pairs of columns in A that induce an EΣM

```
01  C ← ∅
02  for i ← 1...n do
03      for j ← i+1...n do
04          v1, v2, v3, v4 ← false
05          for l ← 1...m do
06              if (a_li, a_lj) = (1,1) then v1 ← true
07              if (a_li, a_lj) = (1,0) then v2 ← true
08              if (a_li, a_lj) = (0,1) then v3 ← true
09              if (a_li, a_lj) = (0,0) then v4 ← true
10          if v1, v2, v3, and v4 are all true then
11              C ← C ∪ {(c_i, c_j)}
12  return C
```

As was already mentioned, the constant in the O-notation of the given algorithm is much smaller than the one implied by the general algorithm given in Section 4.2.1 due to the fact that in lines *06* to *09*, we only check whether all four row vectors induced by an EΣM are present in a pair of columns in A—an important performance gain for practical applications.

The equivalence of Minimum Character Removal to 2-Hitting Set is directly given by the results from Chapter 4:

Theorem 5.19 Minimum Character Removal *is parameter-equivalent to* 2-Hitting Set.

Proof The results of Chapter 4 that concerned Row Deletion also hold—for reasons of symmetry—for Column Deletion problems. According to Theorem 4.7, Minimum Character Removal is parameter-preservingly reducible to 2-Hitting Set. Since the row vector $(\,1\;1\,)$ is only induced once in an EΣM, we can deduce from Theorem 4.12 that 2-Hitting Set is also parameter-preservingly reducible to Minimum Character Removal. (The latter reduction is illustrated in Figure 5.3.) □

The Vertex Cover problem (equivalent to 2-Hitting Set) which we have introduced in Chapter 3 and already mentioned in the last section is probably the best analyzed problem in parameterized complexity theory. Successive improvements of parameterized algorithms for this problem have led to algorithms with a $O(1.29^k + km)$ worst-case running time (see [NiRo03$_b$] and [CKJ01]). Combining this algorithm with the parameterized reduction from Minimum Character Removal to 2-Hitting Set given at the beginning of this section (which runs in $O(m^2n)$ time), we obtain corollary 5.20.

Corollary 5.20 Minimum Character Removal *can be solved in* $O(m^2n + 1.29^k + km)$ *running time.*

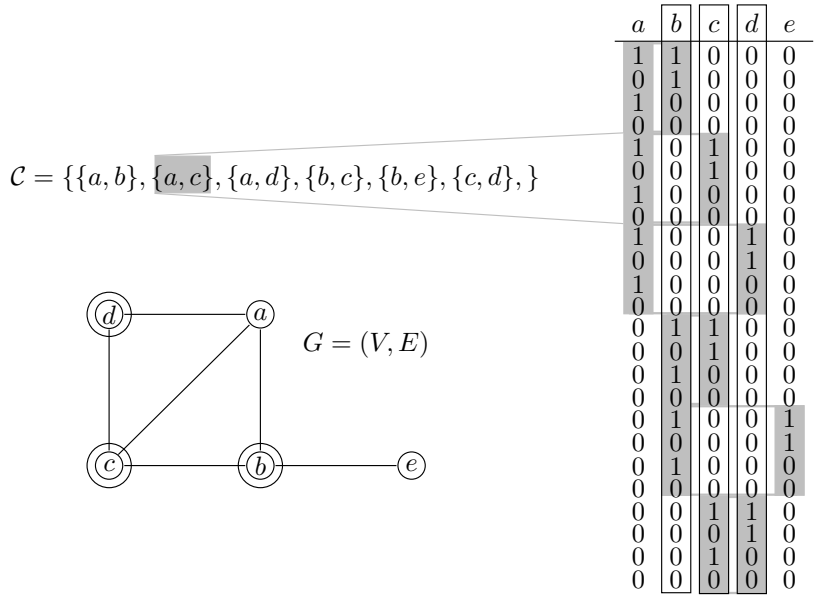


Figure 5.3: An instance of 2-Hitting Set and the corresponding Minimum Character Removal problem. For illustration purposes, the 2-Hitting Set problem is shown with its corresponding Vertex Cover instance. An optimal solution is marked in the graph and the binary matrix.

Proof The running time is obtained by adding the running time given in [NiRo03$_b$] for solving the Vertex Cover problem to the running time required for the reduction from Minimum Character Removal to Vertex Cover.. □

This directly implies that Minimum Character Removal can be solved efficiently in practice for a reasonable dataset: If a number of species evolved according to a phylogenetic model, then the number of characters that contradict this model (i.e., k) should be relatively small. On the downside, the very close relationship between Minimum Character Removal and 2-Hitting Set implied by Theorem 5.19 combined with the fact that the known algorithms for Vertex Cover might not be improved much further suggests that it might not be possible to solve Minimum Character Removal much faster than with the given $O(m^2n + 1.29^k + km)$-algorithm.

This chapter concludes the first part of this work dealing with forbidden submatrix problems and their application in the inference of phylogenies. The next two chapters will be concerned with another set of interesting problems that arise when we try to obtain SNP data from the genetic information of an organism. Just as the problem of inferring perfect phylogenies led to the more general problem of submatrix removal, the problems arising in SNP analysis can be related to a problem called Graph Bipartization.

Chapter 6

Graph Bipartization

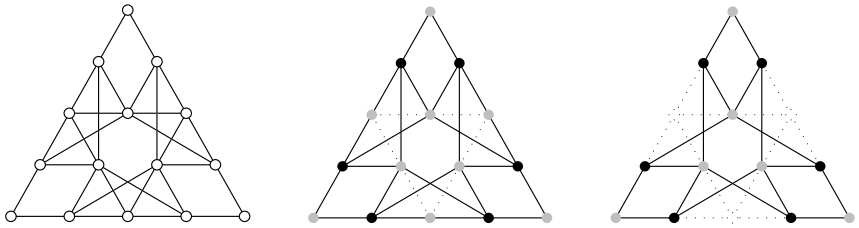
The preceding two chapters have analyzed forbidden submatrix problems and their relationship to Perfect Phylogeny problems. We have already mentioned that among their many qualities, SNPs seem to be a promising source for data that can be used to construct perfect phylogenies. This and the following chapter are more directly concerned with SNPs in that they will deal with computational problems that arise during actually *obtaining* SNP data. This chapter introduces the Graph Bipartization problem, which we shall show—in Chapter 7—to be closely linked to problems arising in the acquisition of SNP data.

Graph Bipartization is the following problem: Given a graph G, remove as few vertices (or edges) as possible from it so that it becomes bipartite[1]. Bipartization by either vertex- or edge-deletion is *NP*-complete [Yann81]. However, making use of the fact that in our applications—those presented in Chapter 7—the graphs that we are to bipartize will most likely be rather small and "almost" bipartite, we develop an exact algorithm for solving Graph Bipartization on these graphs. An implementation of this algorithm is presented in Section 6.4. In this section, we also test the algorithm on random graphs, seeing that the Graph Bipartization problem is already intractable for moderately sized *random* graphs. However, as will be shown in Section 7.4 of the next chapter, the developed algorithms generally allow the bipartization of graphs obtained during SNP analysis even if they contain a few hundred vertices.

6.1 Introduction and Known Results

Besides its applications in computational biology—some of which are presented in Chapter 7—the task of bipartizing a graph is also important for a range of non-biological problems such as, e.g., VLSI chip design [CRS94]. The reason for this is that many problems can be traced back to so called *conflict-graphs*, where the vertices of a graph represent certain data and edges represent data-dependencies. The presence of odd cycles (later in this section we show that a graph is bipartite if and only if it does not contain such cycles) in the conflict-graph indicates that certain data-dependencies contradict each other. For example—as is explained in more detail in Chapter 7—the bipartiteness of a graph constructed from a set of given genotypes determines whether this set of genotypes can be resolved into the underlying haplotypes. Logically, if data conflicts are presumed to be present in a set of

[1] "Bipartiteness" is defined in Section 3.1.



A graph G | G bipartized by deleting six edges | G bipartized by deleting three vertices

Figure 6.1: A graph G and its optimal bipartization by edge and vertex deletion. The bipartiteness of the middle and right graph are illustrated by coloring the vertices black and grey such that no two equally shaded vertices are connected by an edge. Notice that such a shading is not possible for the graph on the left, since it contains cycles of odd length (e.g., 3).

data only due to some flaws (e.g., read or measurement errors), the question for a low-cost way to bipartize the given graph directly arises. We may define bipartization in two ways, formulating the Edge Bipartization and the Vertex Bipartization problem.

Definition 6.1 (Edge Bipartization *Problem*)
Input: *Given a graph $G = (V, E)$ and an integer k.*
Question: *Can G be made bipartite (i.e., free of odd cycles) by deleting at most k edges?*

Definition 6.2 (Vertex Bipartization *Problem*)
Input: *Given a graph $G = (V, E)$ and an integer k.*
Question: *Can G be made bipartite (i.e., free of odd cycles) by deleting at most k vertices?*

By Graph Bipartization we will denote a problem that is either Edge Bipartization or Vertex Bipartization.

An example for both Edge Bipartization and Vertex Bipartization is given in Figure 6.1. The key graph-theoretic idea behind bipartization of a given graph is that a graph is bipartite if and only if it does not contain any odd cycles (see, e.g., the bipartized graph in Figure 6.1): Assume that a given graph $G = (V_1, V_2, E)$ is bipartite and has an odd cycle. Since G contains only edges between vertices in V_1 and V_2, if the cycle has an odd length and starts at a vertex in V_1, it would end in a vertex in V_2, which clearly cannot be. Now assume that a graph has no odd cycles. Then we can simply put an arbitrary vertex of G into the first partition, all of its neighbors in the second, the neighbors of the neighbors in the first again and so on. Assume now that there are two vertices u and v in G that are connected by an edge e and have been put into the same partition (which would inhibit the partitioning process). Then, there is a path p from u to v of even length in the already partitioned graph because the already partitioned graph is connected. Since e connects u and v, p and e form an odd cycle in the graph, a contradiction.

Both Edge Bipartization and Vertex Bipartization have been proven to be *NP*-complete (e.g., see [Yann78], [Yann81] and [LeYa80]) for general graphs. For planar graphs, Edge Bipartization is solvable in polynomial time [Hadl75] whereas Vertex Bipartization on planar graphs is generally *NP*-complete (it becomes solvable in polynomial time when the maximum vertex degree does not exceed 3) [CNR89, Karp72]. For any graph, we

can find a set of edges in polynomial time that is larger by a factor of at most $O(\log |V|)$ compared to the smallest possible set of edges[2] that bipartizes a given graph [GVY96]. However, it has been demonstrated in [PaYa91] that both Edge Bipartization and Vertex Bipartization are *MAX-SNP*-hard[3], meaning that there is no PTAS[4] for these two problems unless $P = NP$ [ALMSS92]. It remains an open question whether Graph Bipartization problems can be approximated within a constant factor [GVY96].

The fixed-parameter tractability of Edge Bipartization and Vertex Bipartization is an open question. A conjecture by Khot and Raman [KhRa02] concerning *parametric duality*[5] states that for *NP*-complete problems that are in *FPT*, their parametric dual is often not in *FPT*. This is however a simply empirical observation that should at most be taken as a hint.

Vertex Bipartization has a parametric dual called MaxCut.

Definition 6.3 (MaxCut *Problem*)
Input: *A graph $G = (V, E)$ and a parameter k.*
Question: *Can V be partitioned into two disjoint subsets V_1 and V_2 such that there are at least k edges in G from vertices in V_1 to vertices in V_2?*

The fixed-parameter tractability of MaxCut is shown through a reduction to Max2Sat (e.g., see [PoTu95] for details).

Definition 6.4 (Max2Sat *Problem*)
Input: *A boolean formula F in 2-CNF and a parameter k.*[6]
Question: *Are k or more clauses in F satisfiable?*

It is obvious that if a graph can be bipartized by deleting at most k edges from it, it has a maximum cut of size at least $|E| - k$. An instance of MaxCut is reduced to Max2Sat by a translation of a given graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ into a boolean formula F with n literals $\{\ell_1, \ldots, \ell_n\}$ and two clauses C_{j1} and C_{j2} for each edge $e_j = \{v_a, v_b\}$, where $C_{j1} = (\ell_a \vee \ell_b)$ and $C_{j2} = (\neg\ell_a \vee \neg\ell_b)$. By developing an algorithm to solve Max2Sat with n clauses in $n^{O(1)} \cdot 1.15^n$ time, Gramm, Hirsch, Niedermeier and Rossmanith [GHNR03] have also provided an $n^{O(1)} \cdot 1.26^m$ algorithm for solving MaxCut on a graph with m edges. In the practical test of the developed branch&bound algorithms for Vertex Bipartization and Edge Bipartization in (Sections 6.4 and 7.4), we will use a Max2Sat-solving program developed by Gramm to compare our algorithms against.

In the above definition of Vertex Bipartization and Edge Bipartization, both problems were given as *decision* problems. However, in the next chapter, we will be interested in

[2]The *optimization* variants of Graph Bipartization problems will be introduced shortly.

[3]Note that second part of the name for this class is an abbreviation for "Strict *NP*" and has no connection with SNPs. It is therefore pronounced "S-N-P" instead of "Snip". More information on the class *MAX-SNP* can be found, e.g., in Chapter 13 of [Papa94].

[4]PTAS stands for "Polynomial Time Approximation Scheme". An approximation scheme for a problem $\mathcal{P}$ is a set of approximation algorithms $\mathcal{A}_\epsilon$, $\epsilon > 0$, where the algorithm $\mathcal{A}_\epsilon$ approximates $\mathcal{P}$ to a factor of $1 + \epsilon$. If every $\mathcal{A}_\epsilon$ has a running time that is polynomial with respect to the input length, we are referring to a *polynomial time* approximation scheme.

[5]For a parameterized language $\mathcal{L}_1$, its parametric dual is $\mathcal{L}_2 := \{(x, k) \mid (x, |x| - k) \in \mathcal{L}_1\}$. For more details, see, e.g. [PrSl03].

[6]A boolean formula is in 2-CNF, if it can be written as $(\ell_1 \vee \ell_2) \wedge (\ell_3 \vee \ell 4) \ldots$ where the ℓ_i are either boolean literals or their negation. A good introduction to boolean formulas and boolean logic may be found, e.g., in Chapter 4 of [Papa94]

finding a *minimum number* of edges or vertices to bipartize a given graph, i.e., the smallest k for which GRAPH BIPARTIZATION returns "YES" on a given graph.

Definition 6.5 (opt-EDGE BIPARTIZATION *Problem*)
Input: *Given a graph* $G = (V, E)$.
Output: *The smallest integer* k *for which* $(G, k) \in$ EDGE BIPARTIZATION.

Definition 6.6 (opt-VERTEX BIPARTIZATION *Problem*)
Input: *Given a graph* $G = (V, E)$.
Output: *The smallest integer* k *for which* $(G, k) \in$ VERTEX BIPARTIZATION.

By opt-GRAPH BIPARTIZATION we denote a problem that is either opt-EDGE BIPARTIZATION or opt-VERTEX BIPARTIZATION. Note that any algorithm solving opt-GRAPH BIPARTIZATION can be used to solve GRAPH BIPARTIZATION as well (for a given input graph G, simply calculate the optimal k and output "YES", if for a given k', $k' \geq k$) and the running time of an algorithm for opt-GRAPH BIPARTIZATION is at most polynomially worse than the running time for the respective GRAPH BIPARTIZATION problem.[7]

6.2 A Parameter-Preserving Reduction from EDGE BIPARTIZATION to VERTEX BIPARTIZATION

Using a parameter-preserving reduction, this section shows that VERTEX BIPARTIZATION is at least as hard to solve as EDGE BIPARTIZATION. It should be noted that this proof supports the general observation from [Yann81] that vertex-deletion problems in order to achieve a certain graph-property generally appear to be harder than their edge-deletion equivalents.

Theorem 6.7 EDGE BIPARTIZATION *is parameter-preserving reducible to* VERTEX BIPARTIZATION.

Proof Let $G_1 = (V_1, E_1)$ be an instance of EDGE BIPARTIZATION and $G_2 = (V_2, E_2)$ be the parameter-equivalent instance of VERTEX BIPARTIZATION we wish to construct. Let $V_1 = \{v_1, \ldots, v_n\}$ and $E_1 = \{e_1, \ldots, e_m\}$. The new graph G_2 will consist of two different types of vertices:

- Type I: This set contains $k + 1$ duplicates of every vertex in V_1:

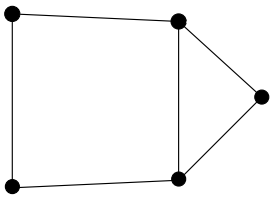
$$V_{\mathrm{I}} := \{u_{ij} \mid v_i \in V_1, 0 \leq j \leq k\}$$

- Type II: This set contains two vertices for every edge in E_1:

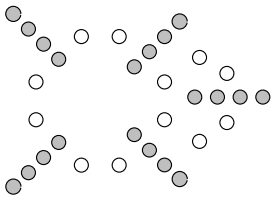
$$V_{\mathrm{II}} := \{w_{\ell 1}, w_{\ell 2} \mid e_\ell \in E_1\}$$

Having defined these sets of vertices, we set $V_2 := V_{\mathrm{I}} \cup V_{\mathrm{II}}$. Note that vertices of type I are all named by a u and those of type II by a w. The edges in E_2 also consist of two sets:

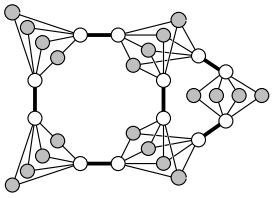
[7] I.e., if there is an algorithm for VERTEX BIPARTIZATION, we can test for all $0 \leq k \leq |V|$ on a given graph G whether $(G, k) \in$ VERTEX BIPARTIZATION and output the smallest k for which this is true.



The original graph G_1
for construction of G_2



The vertices of V_2
($k = 3$ in this example)



G_2 with all edges and
vertices from E_2 and V_2

Vertex-Types:
● : Type I
○ : Type II

Edge-Types:
━ : Type I
– : Type II

Figure 6.2: Construction of a VERTEX BIPARTIZATION instance from an EDGE BIPARTIZATION instance. On the top left, we see the original graph that is an input for VERTEX BIPARTIZATION. Then, on the top right, the vertices from the set V_{I} and V_{II} (see text for details) are displayed. The final output graph of the reduction, where also all edges from the sets E_{I} and E_{II} (again, see text for details) have been added, is shown. Note that although the deletion of one single edge would be sufficient to bipartize G, we set $k = 3$ in this example to better illustrate the duplication of the vertices in G_1 by the vertices of type I.

- E_{I}: For each edge in the original graph, there is a pair of vertices of type II. Each such pair is connected by an edge from this set:

$$E_{\mathrm{I}} := \{\ \{w_{\ell 1}, w_{\ell 2}\} \mid e_\ell \in E_1\ \}$$

- E_{II}: Note that E_{I} contains an edge for every edge in E_1. If two vertices in G_1 were connected by an edge e_ℓ, then E_{II} will connect the corresponding vertices in V_{I} using the edge $\{w_{\ell 1}, w_{\ell 2}\} \in E_{\mathrm{I}}$:

$$E_{\mathrm{II}} := \{\ \{u_{aj}, w_{\ell 1}\}, \{u_{bj}, w_{\ell 2}\} \mid e_\ell = \{v_a, v_b\} \in E_1, 0 \leq j \leq k\}$$

In analogy to the vertices in V_2, we set $E_2 := E_{\mathrm{I}} \cup E_{\mathrm{II}}$. Figure 6.2 gives an example for the construction of G_2.

The idea behind the reduction is as follows: If two vertices v_a and v_b are directly connected in G_1 by an edge e_ℓ, they are now connected by a path $u_{aj} w_{\ell 1} w_{\ell 2} u_{bj}$ of length 3 in G_2. Note that this conserves all odd and even cycles, for an even cycle of length $2n$ is now represented as an even cycle of length $3 \cdot 2n = 6n$ and an odd cycle of length $2n + 1$ as an odd cycle of length $3 \cdot (2n + 1) = 6n + 3$. Deleting either $w_{\ell 1}$ or $w_{\ell 2}$ from the path directly corresponds to the deletion of an edge in G_1 concerning the destruction of cycles. Note that if V_{I} were to contain less than k duplicates of every vertex from V_1, an algorithm that solves the VERTEX BIPARTIZATION problem on G_2 might also consider deleting all of the vertices in V_{I} that

correspond to a certain $v_i \in V_1$ in order to destroy an odd cycle—an operation for which there is no equivalent edge deletion. The $k+1$ duplicates therefore "secure" each vertex from deletion. It remains to be shown that the duplication process conserves all cycles from G_1 in G_2:

For any vertex $u_i \in V_\mathrm{I}$, let $S_i := \{u_{ij} \mid 0 \leq j \leq k\}$. Then for any S_i, $N(S_i)$ shall be defined as the set of vertices adjacent to all vertices in S_i.[8] We now prove that any path between a vertex $w \in N(S_i)$ and $w' \in N(S_i)$ that contains solely vertices from S_i and $N(S_i)$ is of even length $(*)$. This is not hard to show since such a path p of length l between a vertex $w \in N(S_i)$ and $w' \in N(S_i)$ that contains solely vertices from S_i and $N(S_i)$ can be decomposed into $p = ws_1x_1s_2x_2 \ldots x_{\frac{l}{2}-1}s_{\frac{l}{2}}w'$ where all $x \in N(S_i)$ and all $s \in S_i$. Thus, l must be even.

We now show that $(G_1, k) \in$ Edge Bipartization $\Leftrightarrow (G_2, k) \in$ Vertex Bipartization.

"$\Rightarrow$": Assume that G_1 can be bipartized by deleting at most k edges, yielding G_1'. Then, for each of those edges e_ℓ, we delete one of the corresponding vertices $w_{\ell\bullet}$ in G_2 (see the Definition of V_II). Now, assume that after this deletion process, there is still an odd cycle c in G_2. This cycle c must contain a vertex of type II because by $(*)$, cycles containing just vertices of type I are even. Call this vertex w_c. The cycle c may be interpreted as a path p of odd length in G_2 from a vertex w_c to itself. This path will contain edges of both types I and II. It follows from $(*)$ that there is an even number of edges of type II in p. After having deleted the vertices in G_2 that corresponded to edges deleted in G_1, there is exactly one edge of type II in G_2 for every edge in G_1'. Thus, if c is odd, the number of edges of type II in p must also be odd since $E_2 := E_\mathrm{I} \cup E_\mathrm{II}$ and the other edge types are present an even number of times. This however implies that there is an odd cycle in G_1', a contradiction to our initial assumption that G_1' is bipartite.

"$\Leftarrow$": Assume that we can bipartize G_2 by deleting at most k vertices from it, yielding G_2'. The following can be seen from the construction: If an odd cycle was induced in the original graph G_1, it will be induced in G_2 and contain vertices of both types I and II. It is then impossible to bipartize G_2 by deleting vertices of type I, because for each vertex that was part of the odd cycle in G_1, there are now $(k+1)$ copies present in G_2. We may therefore assume w.l.o.g. that only vertices of type II are removed from G_2. Removing a vertex of type II from G_2 will implicitly delete an edge from E_I. Each edge in E_I corresponds to exactly one edge in G_1. Now, delete all edges in G_1 corresponding to the vertices that were removed in G_2, obtaining G_1'. If this does not bipartize G_1 (as we intend), this means that there is still an odd cycle—call this cycle c—present in G_1'. However, then those edges in E_I that correspond to the edges in c are still present in G_2'. The vertices of type II in G_2' adjacent to those edges can therefore not have been removed from G_2. But if, for the odd cycle c, none of the corresponding edges or vertices have been removed in G_2', G_2 must also contain that odd cycle and therefore cannot be bipartite—a contradiction.

Performing the construction of G_2 requires $O(|E_1| + (k+1) \cdot |V_1|) \leq O(|E_1| + |E_1| \cdot |V_1|)$ time for adding the vertices and $O(|E_1| + (2k+2) \cdot |V_1|) \leq O(|E_1| + 2|E_1||V_1|)$ time for the edge construction—it can thus be carried out in polynomial time. Note that the parameter k is directly preserved by the reduction. □

The theorem implies that, from a parameterized point of view, Vertex Bipartization is at least as hard to solve as Edge Bipartization.

[8] Note that for any S_i, $N(S_i) \subseteq V_\mathrm{II}$ and that if a vertex is adjacent to one vertex in S_i, it is adjacent to all vertices in S_i due to the construction.

6.3 A Branch&Bound Approach

As was mentioned in Section 6.1, this section will develop algorithms for solving the *optimization* variant of a given Graph Bipartization problem.

Having shown that Vertex Bipartization and Edge Bipartization are *NP*-complete, not approximable within a constant factor, and might not even be fixed-parameter tractable, we also know that the respective optimization variants opt-Edge Bipartization and opt-Vertex Bipartization are also hard to solve. In this section, we will develop efficient algorithms for opt-Graph Bipartization using a technique known as *branch&bound* based on the idea that for the applications developed in Chapter 7, the given input graphs should be relatively small and "almost bipartite."

Branch&bound works as follows: As was noted in Chapter 3, *NP*-complete problems can be solved by a computer that correctly guesses a solution to the problem and then deterministically verifies it. As we do not have access to a machine with such "oracle" capabilities, we have to try all possible solutions—branch&bound is, in a way, an attempt to perform this trial and error procedure more "intelligently."

We shall illustrate the principle of branch&bound using opt-Vertex Bipartization as an example. For a given opt-Vertex Bipartization instance, trying all possible solutions means we have to find all possibilities to bipartize G and see which is the minimal number of vertices necessary to bipartize G. There are $2^{|V|}$ subgraphs induced by G, as each vertex can either be deleted or kept in the graph. Bipartizing the graph by deleting vertices is equivalent to looking for an induced bipartite subgraph in G that contains as many vertices as possible. Assume that we have enumerated the n vertices of a graph G as $v_1, \ldots, v_n$. Then we can use the following algorithm to generate all $2^{|V|} = 2^n$ subgraphs of G by a branching algorithm similar to the one introduced in Section 3.3: We select a vertex v from G and branch into two subcases, where in the first subcase v is kept in the graph and in the second case v is removed from G. For the resulting subgraphs, the algorithm is applied recursively (note that once we have decided to keep v in the graph, it cannot be removed by the algorithm on the resulting subgraph). Initially, the algorithm is called with G and n as inputs.

Algorithm: Enumerating subgraphs
Input: A graph $G = (V, E)$ with labeled vertices
$V = v_1, \ldots, v_n$ and a number i
Output: All 2^n induced subgraphs of G

```
01 if i = 0 then
02     output G
03 call this algorithm with G and i − 1 as inputs
04 call this algorithm with G \ {v_i} and i − 1 as inputs
```

Observe that this algorithm can be depicted in a search tree structure as in Section 3.3. In order to find an optimal solution to opt-Vertex Bipartization on G we have to look for a leaf in the tree that yields a bipartite graph with as many vertices as possible. So far, the branch&bound search seems just to be a trial and error approach with exponential running time. We will not be able to loose the exponential worst-case running time, however, there are two ways to (hopefully significantly) speed up our search:

- The key to branch&bound is the following idea: If a partial solution cannot do better than the best solution so far it is not further explored in the search tree. Assume

that we have already found a solution to OPT-VERTEX BIPARTIZATION on G that uses only $i < n$ vertices. If we are at an inner node N in the tree where we have already deleted i vertices from G, the solutions generated by traversing further into the tree from N can never be better than the one we have already found. We therefore do not need to traverse into that subtree and can directly look at another branch in the tree leaving from N's parent node. By always keeping the best solution found so far in memory, we therefore do not need to traverse the whole tree to see if there is a better one, but only look into "promising" solutions. In order to further speed up the search in its initial phase, a *heuristic* is employed to get a (hopefully good) initial solution. A heuristic[9] is an algorithm that finds a solution to a problem but does not guarantee optimality.[10] Special heuristics for OPT-EDGE BIPARTIZATION and OPT-VERTEX BIPARTIZATION will be developed later on in this chapter.

- There are possibly certain structures in the graph G for which we can deterministically predict how they will be solved in an optimal solution to OPT-VERTEX BIPARTIZATION on G. For example, vertices of degree 1 cannot be part of a cycle and therefore do not need to be considered for deletion when traversing the search tree. We refer to such operations as *data reduction*. We can "pre-solve" such structures before performing the actual branch&bound procedure, thereby reducing the amount of input data. Data reduction can also be employed during the search tree traversal.

Using these techniques, we can generally—but without any guarantees—speed up an exhaustive search for an optimal solution to OPT-GRAPH BIPARTIZATION on G. This is exactly what we will later see in the implementation of the algorithms in Sections 6.4 and 7.4: While the algorithms are generally too slow to bipartize random graphs even of moderate size and average vertex degree, they are generally capable of solving GRAPH BIPARTIZATION for graphs from SNP analysis even if these graphs contain a few hundred vertices.[11]

The development of an efficient branch&bound algorithm for OPT-GRAPH BIPARTIZATION is divided into the following two subsections: The next subsection will present the heuristics for obtaining the initial bound for the respective OPT-GRAPH BIPARTIZATION problem, followed by the subsection presenting the data reduction rules applied. An implementation of the developed algorithms in the *Java* programming language is presented and analyzed in Section 6.4.

6.3.1 Initial Heuristics

For the OPT-EDGE BIPARTIZATION-heuristic we will use an algorithm developed by Schröder, May, Vrto, and Sýkora [SMVS97][12]. This algorithm is very efficient, easy to implement, and—according to [SMVS97]—often produces results that are very close to the optimal solution (although there is no guarantee for the quality of the solution). The algorithm works as follows on a given graph G: First, the vertices of G are colored randomly red and blue. Then, as long as there is a vertex v in G that has more neighbors colored equally to it

[9] The word heuristic originates from the greek word $\epsilon\dot{\upsilon}\rho\acute{\iota}\sigma\kappa\epsilon\iota\nu$, meaning "to discover."

[10] There are many heuristics—including the ones that we will use—that do not make statements even about how close the generated solution is to an optimal one.

[11] The significance of this result will become clear later in this section when discussing Reduction Rule 3: A complete enumeration of all possible solutions to a GRAPH BIPARTIZATION problem would have to consider over $2^{100} < 10^{30}$ subgraphs for a graph containing 100 vertices.

[12] The title of this paper, "Approximation algorithms for the Vertex Bipartization Problem", is quite misleading as this paper only deals with heuristics for EDGE BIPARTIZATION.

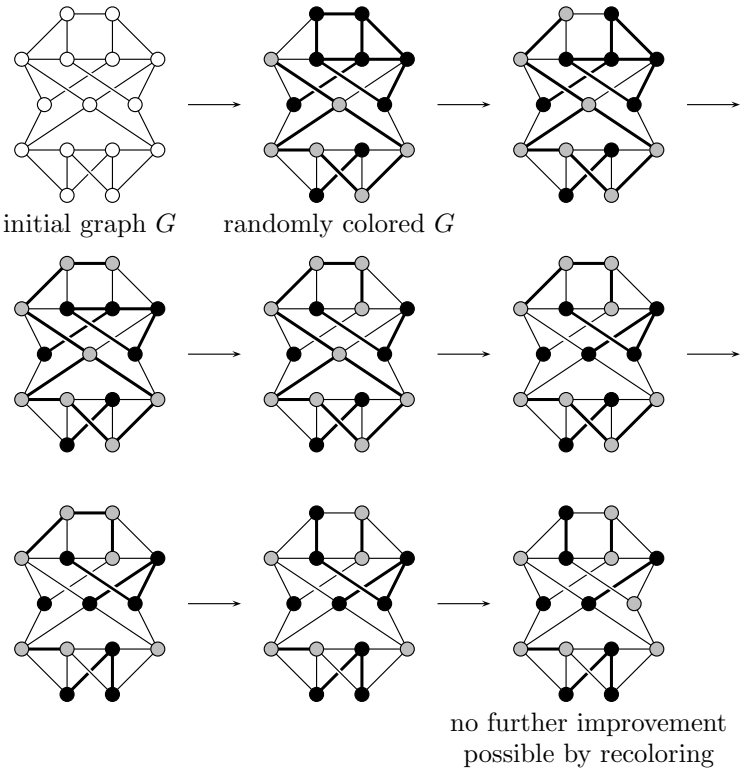


Figure 6.3: Illustration of the recoloring heuristic for Edge Bipartization used to obtain an upper bound for the branch&bound algorithm. Starting with a randomly colored graph, if there are vertices in G whose recoloring reduces the number of conflict edges in G, these vertices are recolored. The detailed algorithm is provided in the text. Note that the recoloring heuristic yields a suboptimal solution (six edges instead of four) to Edge Bipartization in this example.

than neighbors colored differently to it, the respective v is *recolored* (hence, this algorithm is referred to as the *recoloring heuristic* for Edge Bipartization). We will refer to an edge that connects two equally colored vertices as a *conflict edge*. Written in pseudocode, the algorithm looks like this:

Algorithm: Recoloring heuristic for Edge Bipartization from [SMVS97]
Input: A graph $G = (V, E)$
Output: An upper bound for Edge Bipartization on G

```
01  randomly color vertices in G red or blue
02  while there is a vertex v in G with more neighbors
      colored equally than differently to it do
03    change the color of v
04  return number of conflict edges in G
```

The algorithm's worst-case running time is bounded by $O(|V| \cdot |E|)$ since each execution of line *03* decreases the total number of conflict edges in the graph by at least one. However, practical performance of the algorithm (see Section 6.4) has shown to be a lot better than

suggested by this bound. The algorithm is illustrated in Figure 6.3.

The heuristic for determining the upper bound of OPT-VERTEX BIPARTIZATIONon G is as follows: The graph is traversed using depth-first search.[13] In the heuristic algorithm, an initial vertex is chosen arbitrarily. From then on, we are always traversing from a visited vertex to a not-visited one during the search. The heuristic will color the graph during traversal; the initial vertex is given the color red, and from then on, each vertex that is visited from a red vertex is colored blue and vice-versa. During this coloring process, conflicts may arise if a vertex is given a color that one of its neighbors already has (this leads to *conflict edges* between equally colored vertices). In this case, if the newly colored vertex causes more than one conflict edge, it is deleted from G. If the newly colored vertex causes one conflict edge e_c, the endpoint of e_c with the higher degree is removed from G. In a more formal form:

Algorithm: OPT-VERTEX BIPARTIZATION heuristic
Input: A graph $G = (V, E)$
Output: An upper bound for OPT-VERTEX BIPARTIZATION on G

```
01  cost ← 0
02  choose v ∈ V arbitrarily
03      color(v) ← red
04          for each vertex u visited from a colored vertex w
              during depth-first search traversal of G do
05              if color(w) = red then
06                  color(u) ← blue
07              if color(w) = blue then
08                  color(u) ← red
09              if u has more than one neighbor colored equally to it then
10                  G ← G \ {u}
11                  cost ← cost + 1
12              if u has exactly one neighbor x colored equally to u then
13                  z ← vertex from {u, x} with maximal degree
14                  G ← G \ {z}
15                  cost ← cost + 1
16  return cost
```

The idea behind this heuristic is the following: The graph is greedily colored as if it were bipartite. Once this cannot be done anymore, two cases are distinguished: If the newly colored vertex u is the cause of more than one conflict edge, either it or all equally colored neighbors need to be removed from G. The locally cheaper solution, i.e., the deletion of u, is chosen. If the coloring of u causes just one conflict edge with another vertex x in G, the vertex with the higher degree is deleted because this implicitly removes as many edges—potential conflict edges—as possible from G. The algorithm is illustrated in Figure 6.4. As the algorithm executes a depth-first search algorithm exactly once, its running time is $O(|E|)$ (assuming that marking vertices as "deleted" takes constant time).

[13] Depth-first search is an algorithm to efficiently traverse all vertices in a connected graph in $O(|E|)$ time using a recursive procedure. See, e.g., [CLRS01] or [Skie98] for details.

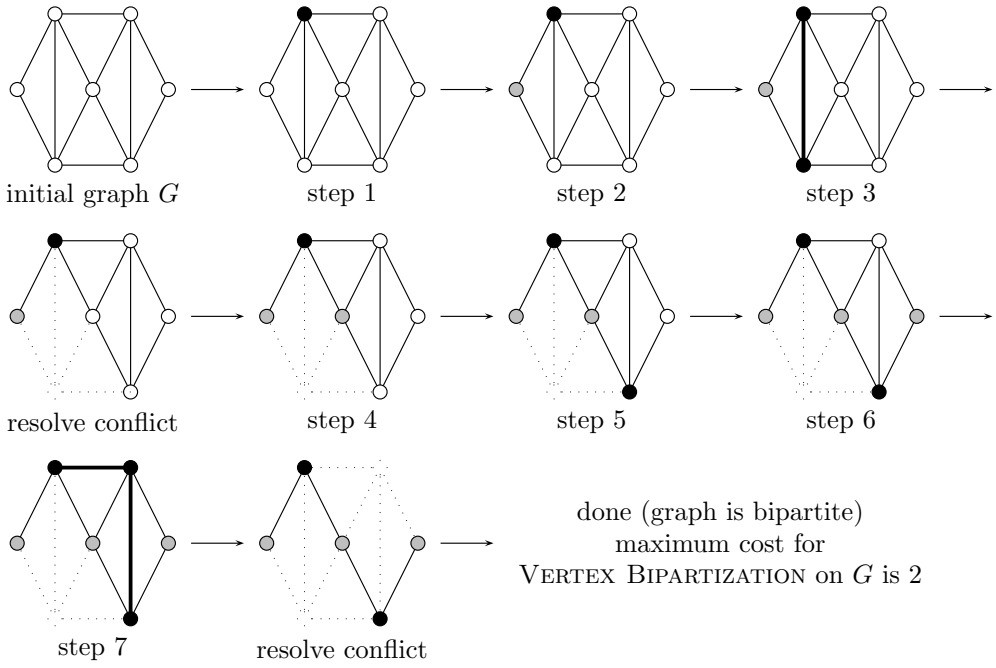


Figure 6.4: Illustration of the heuristic for opt-Vertex Bipartization used prior to branch&bound. Starting from an initial vertex in a graph G which is colored black, all vertices in the graph are colored alternatively grey and black during a depth-first search traversal of G. If conflict edges (drawn in bold) arise, the newly colored vertex or one of its neighbors is deleted according to the rules given in the text. For the displayed proceeding of the algorithm, a solution involving the deletion of 2 vertices from G is found (in this case, this is even an optimal solution).

6.3.2 Data Reduction Rules

The following data reduction rules can be applied to a graph G before and while performing a branch&bound search for an optimal solution to opt-Vertex Bipartization or opt-Edge Bipartization. For all reductions, we will use the following notation and agreements: The reduction rules are always applied to a graph $G = (V, E)$, G is not assumed to be connected, however, all reduction rules presented apply only to connected components.[14] If G induces a connected bipartite subgraph $G' = (V', E')$, we denote by V'_1 and V'_2 (where $V'_1 \cap V'_2 = \emptyset$ and $V' = V'_1 \cup V'_2$) the two subsets into which V' can be divided such that E' contains no edges between vertices in V'_1 and vertices in V'_2. For simplifying the discussion, we shall refer to vertices in one of the two subsets (either V'_1 or V'_2) as being colored *red* and those in the other subset as being colored *blue* (analogously to the previous section).

In order to simplify the discussion of data reduction for opt-Edge Bipartization, we will allow edges to be given a *weight* $\omega : E \to \mathbb{N}$. This weight has the following meaning: Initially, each edge in the graph has weight 1, however, if it is observed during data reduction that a certain structure (e.g., a set of paths) between two vertices u and v can only be removed

[14] If G is not connected, we should not solve Edge Bipartization or opt-Vertex Bipartization on G as a whole but on its connected components separately as this is much more efficient (see the discussion of Reduction Rule 3).

from the given graph by deleting at least i edges between u and v, this substructure is replaced by an edge of weight i—representing the fact that the deletion of this edge in the new graph actually represents the deletion of i edges in the original graph.

For OPT-VERTEX BIPARTIZATION, we introduce the following modification: As will be shown in the following reduction rules, it is sometimes possible to predetermine for a vertex v that there is an optimal solution to OPT-VERTEX BIPARTIZATION on the given graph that does not include v. In this case we allow v to be marked as "not considered for deletion", meaning that during branch&bound, there is no branching considering the deletion of v—rather, v is always kept in the graph. We denote such a marking of a vertex v by calling v "undeletable". Note that a vertex marked as "un*deletable*" is still considered for *removal* by reduction rules that might apply to it.

It is important to note that there are two main types of reduction besides the special case for OPT-VERTEX BIPARTIZATION just mentioned: The first one involves *removing* some parts of a graph because these parts do not play any role in finding an optimal solution. The second one involves choosing a certain edge e or vertex v for *deletion* because it is clear that there must exist an optimal solution containing e or v, respectively. We will emphasize the difference between "deletion" and "removal" by increasing a variable *count*—reflecting the increase in the cost of the solution—each time a deletion is performed.

It is important to recognize that through the application of an individual reduction rule, other rules may become applicable, therefore, the following reduction rules should be used iteratively on the input graph until no further modification is possible. In principle, the order of execution of the individual reduction rules is not important in the sense that the execution of one rule renders another rule inapplicable. For best performance, however, Reduction Rules 1 and 2 should be executed first as they are quick and may quickly reduce the given graph's size. This should then be followed by Rules 3 and 4. These rules may split a connected component in the input graph into two or more smaller connected components, so that the following, computationally more expensive rules can be carried out on each component separately, which greatly increases their efficiency (see also the discussion of Reduction Rule 3).

The following rules are all applicable to both EDGE BIPARTIZATION and VERTEX BIPARTIZATION unless explicitly stated.

Reduction Rule 1 (REMOVING BIPARTITE COMPONENTS): *If G induces a connected bipartite component C, remove C.*

Correctness The correctness of this algorithm is obvious. However, it is presented here because some of the later reduction rules as well as the branch&bound procedure itself might cause a bipartite connected component to be induced in G.

Running time: All bipartite connected components in the input graph can be found using a depth-first search algorithm that colors the vertices in G while detecting all connected components. This takes $O(|E|)$ time.

Reduction Rule 2 (REMOVING VERTICES OF LOW DEGREE: *Remove all vertices of degree 1 from G. For* OPT-VERTEX BIPARTIZATION*, mark all vertices v of degree 2 as undeletable.*[15]

[15]Odd cycles consisting just of vertices of degree 2 are handled by Reduction Rule 5.

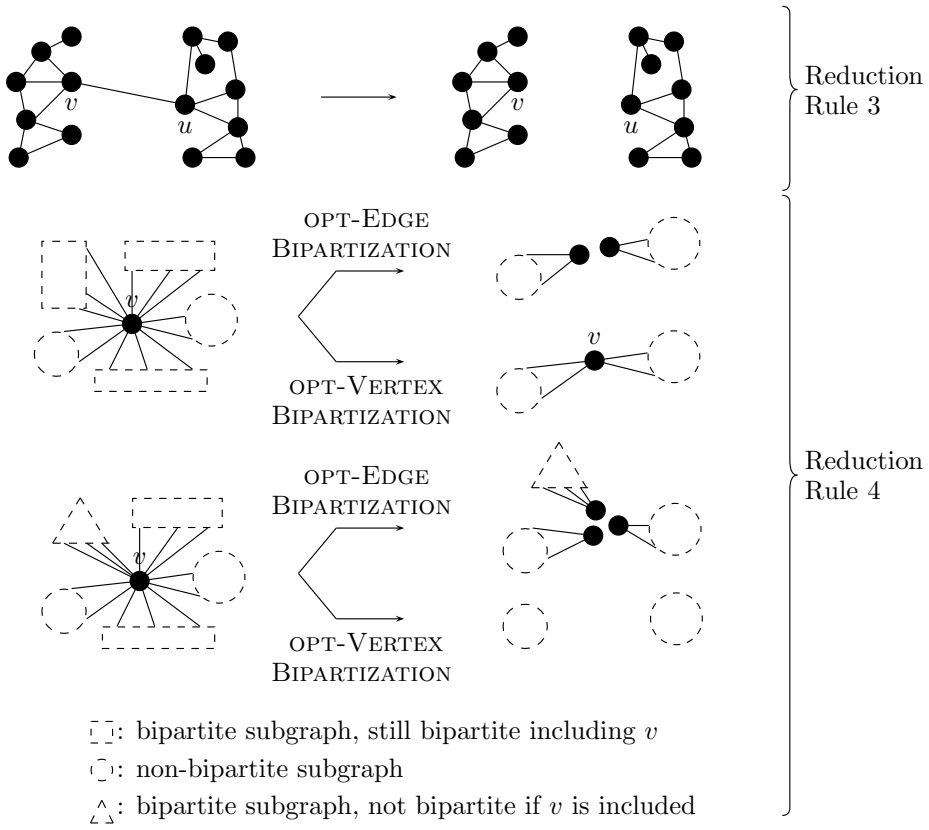


Figure 6.5: Examples for Reduction Rules 3 and 4. A detailed description is given in the text.

Correctness and Running Time: The algorithm—clearly executable in $O(|V|)$ time—is correct for vertices of degree 2 since any odd cycle in G that includes v must also include both its neighbors. Therefore, for any optimal solution to OPT-VERTEX BIPARTIZATION on G including v there is a corresponding optimal solution including one of v's neighbors. Vertices of degree 1 can impossibly induce a cycle in G and can therefore be removed.

In the implementation of the branch&bound algorithms, we will also introduce the following optimization for OPT-EDGE BIPARTIZATION: A vertex v of degree 2 is only colored if both of its neighbors already have been colored. This is due to the fact that we can then color v deterministically.[16]

Reduction Rule 3 (SPLITTING THE GRAPH I): *Let $e \in E$ be an edge-separator*[17] *of order* 1 *in $G = (V, E)$. Then, remove e from G.*

Example: See Figure 6.5 for an illustration of this reduction.

Correctness: If e is an edge that connects two otherwise disconnected components in G, then e may clearly not be member of a cycle in G. Therefore, any union of two optimal

[16] If both neighbors are colored equally, we color v in the opposite color, otherwise we color v so that the weight of the edge that connects v with its equally colored neighbor has minimum wight.

[17] Separators are defined in Section 3.1.

solutions an OPT-GRAPH BIPARTIZATION problem on G_1 and G_2 yields an optimal solution for $G = G_1 \cup G_2 \cup \{e\}$.

Application of this rule can significantly reduce the running time of the branch&bound algorithm: As we have already seen previously, branch&bound needs (in a worst-case estimation) to check $O(2^{|V|})$ different solutions to solve OPT-EDGE BIPARTIZATION or OPT-VERTEX BIPARTIZATION on a graph G. However, if there is an edge e in G whose removal splits G into two nonempty components $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with

$$V_1 \cap V_2 = \emptyset, V_1 \cup V_2 = V,$$

then, solving OPT-EDGE BIPARTIZATION or OPT-VERTEX BIPARTIZATION for both components separately only requires looking at

$$2^{|V_1|} + 2^{|V_2|} \leq 2 \cdot 2^{\max\{|V_1|,|V_2|\}} \leq 2^{|V|}$$

solutions. Note that the gain in computational speed through this rule increases the more equally sized the two components are.

Running time: In order to find all separating edges that comply with this reduction rule, we need to iterate over all $e = \{u, v\} \in E$ and then, for every such e, test in $O(|E|)$ time (using depth-first-search), if there is a path from u to v that does not lead over e. Thus, the total running time for applying this rule is $O(|E|^2)$.

Reduction Rule 4 (SPLITTING THE GRAPH II): *Let $v \in V$ be a vertex-separator of order 1 in $G = (V, E)$ whose removal splits G into n connected Components $C_1, \ldots, C_n$. Then,*

- *for* OPT-EDGE BIPARTIZATION*: If v is already colored, replace G by the n connected components $G_1, \ldots, G_n$ where G_i is the subgraph induced in G by the vertices of $C_i \cup \{v\}$. Remove all G_i that are bipartite.*[18]
- *for* OPT-VERTEX BIPARTIZATION*: If, for a certain C_i, the subgraph induced in G by the vertices of $C_i \cup \{v\}$ is bipartite, remove C_i from G. If a C_i itself is bipartite, but the subgraph induced by the vertices in $C_i \cup \{v\}$ is not, delete v, and increase* count *by one.*

Example: See Figure 6.5 for an illustration of the different cases of this reduction.

Correctness: For OPT-EDGE BIPARTIZATION, the correctness of the reduction rule is clear: If v is a vertex-separator of order 1 for G, any odd cycle including v does not include vertices from two different C_i. For OPT-VERTEX BIPARTIZATION, there are two cases to consider: On the one hand, if a connected component C_i is bipartite and $C_i \cup \{v\}$ is bipartite as well, there are no odd cycles in G that include a vertex from C_i, for that reason C_i may be removed from G. On the other hand, if a component C_i is itself bipartite, but $C_i \cup \{v\}$ is not, then there exists at least one odd cycle in G consisting of v and some vertices in C_i. Note however that all such odd cycles that can be destroyed by deleting a vertex from C_i may also be destroyed by deleting v.

Running time: In order to find all vertex-separators of order 1 in G, we iterate over the vertices in G, each time testing by depth-first-search whether deletion of v results in at least two connected components. This takes a total of $O(|V||E|)$ time. For each of these

[18] If v is not colored, we branch into two subcases where v is either colored red or blue before applying this reduction rule.

components, we test in the case of OPT-VERTEX BIPARTIZATION whether it is bipartite and whether v only has edges to vertices of one color for each component. Thus, the algorithm takes $O(|V||E|)$ time for OPT-EDGE BIPARTIZATION and $O(|V|^2|E|)$ time for OPT-VERTEX BIPARTIZATION.

Reduction Rule 5 (MAKING SIMPLE PATHS SHORTER): *Let u and v be two vertices in G that are connected by a path $p = (w_1 \dots w_l)$ of length ℓ in G; u and v may additionally be connected by an edge $e \in E$. If every w_i has degree 2, then remove all w from G and*

- *connect u and v by a new edge $\{u, v\}$ if ℓ is even and u and v are not already connected by an edge in E.*
- *connect each u and v to a new vertex z by two edges $\{u, z\}$ and $\{v, z\}$ if ℓ is odd.*

For OPT-EDGE BIPARTIZATION, *two cases must be distinguished for the adjustment of the edge-weights:*

I. If ℓ is even, $\{u, v\}$ has a weight equal to the minimum weight edge in p plus—if u and v were connected by an edge e before the reduction—the weight of e.

II. If ℓ is odd, both $\{u, z\}$ and $\{v, z\}$ have a weight equal to the minimum weight edge in p.

If there is a connected component in G that is a cycle of size 3, delete the lowest weight edge (or—in the case of OPT-VERTEX BIPARTIZATION*—an arbitrary vertex) from the cycle and remove the rest of the component.*

Example: See Figure 6.6 for an example of the cases of this reduction. In the figure, paths of length 3 (top) and four (bottom) and their respective reduction are shown as examples.

Correctness: Note that by replacing the path p by a shorter one, all odd cycles in G are preserved.

Correctness for OPT-EDGE BIPARTIZATION: Since the removal of a single edge from p destroys all cycles that include p, there is no optimal solution to OPT-EDGE BIPARTIZATION on G that includes the removal of two or more edges from p. The adjustment of the edge-weights assures that we replace p by an edge whose weight is equal to the lowest weight edge that would destroy p. Adding the weight of the edge e connecting u and v (if such an edge exists in G) in Case I is justified by the observation that in order to "disconnect" u and v, we have to delete an edge in p as well as e itself (any odd cycle including p implicitly defines an odd cycle that contains e instead of p).

Correctness for OPT-VERTEX BIPARTIZATION: Assume that there is at least one cycle in G that contains the path p between u and v. If there is an optimal solution to OPT-VERTEX BIPARTIZATION on G that includes the removal of a vertex from p (thus "disconnecting" u and v), then there must also be one that includes either u or v, because deleting either u or v from G clearly "disconnects" u and v just as the deletion of any other vertex from p would. Moreover, cycles that include either u or v but not p are destroyed alongside.

Running time: This reduction rule can be implemented very efficiently in $O(|V|)$ time: To find all simple paths in the graph, we simply iterate over each vertex u in G. If the respective u has degree 2, we check which of its neighbors v and w has degree 2. For each neighbor that has degree 2, we check whether its neighbor (other than u) has degree 2 and so

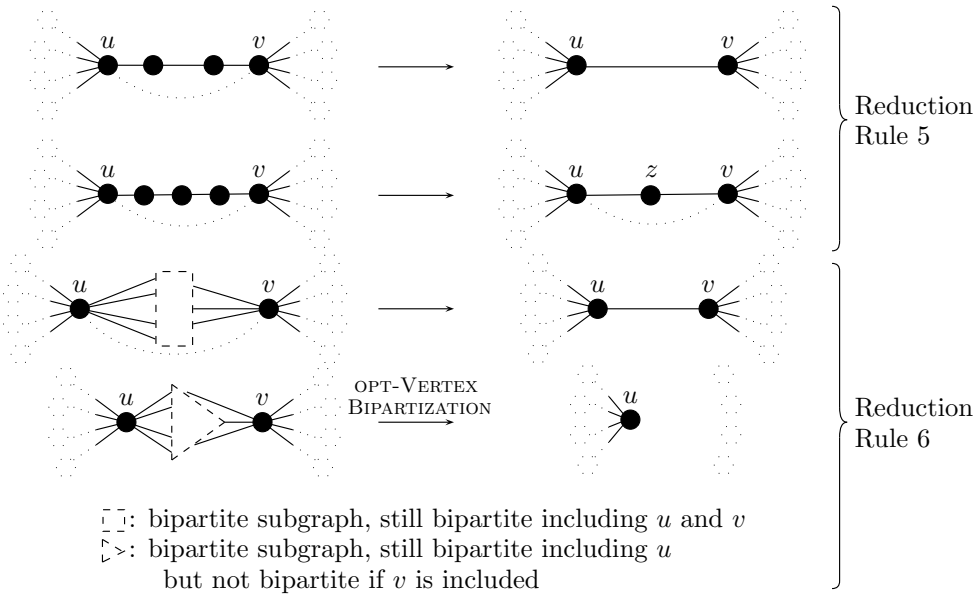


Figure 6.6: Illustrations for Reduction Rules 5 and 6. Detailed descriptions for the corresponding rules are provided in the text.

on, extending the path until we reach to end-vertices with a degree different from two, then applying the reduction. If this rule is applied for opt-Edge Bipartization, we additionally record the lowest weight edge in p while elongating p (this only requires a constant amount of additional time for each vertex added to p). Each vertex of degree 2 is only looked at once, any other step can be carried out in constant time. Hence the algorithm takes $O(|V|)$ time to find all simple paths in the graph.

Reduction Rule 6 (Vertex Separators of Order 2): *Let $\{u, v\} \subseteq V$ be a vertex-separator of order 2 in G, where C is a connected component induced in G by deletion of u and v.*

- *If $C \cup \{u, v\}$ is bipartite, replace C in G by a path of even length with edges e and e' if all paths from u to v using just vertices in C are of even length, else replace C by an edge e''. For* opt-Edge Bipartization*, the weight of e, e' and e'' is equal to the weight of the minimum cut in $G' = C \cup \{u, v\}$ between u and v.*[19]
- *For* opt-Vertex Bipartization*: If $G' = C \cup \{u, v\}$ is not bipartite but $C \cup \{u\}$ is, delete v from G, increasing* count *by one. Additionally, remove C from G. Proceed analogously if G' is not bipartite but $C \cup \{v\}$ is.*

Example: See Figure 6.6 for an example of the cases of this reduction.

Correctness: If $G' = C \cup \{u, v\}$ is bipartite, this subgraph contains no odd cycles. Therefore, if there are odd cycles in G that include vertices from C, these will always include u and v as well. This already justifies the replacement by paths for opt-Vertex Bipartization,

[19]The *minimum cut* between two vertices u and v in a graph is the edge separator of minimum weight whose deletion puts u and v into two disjoint connected components.

as for any optimal solution including the deletion of a vertex w from C there is a corresponding optimal solution including the deletion of v or u instead of w. For OPT-EDGE BIPARTIZATION, if there is an optimal solution that includes the deletion of edges from G', then all paths between u and v must be destroyed by such a deletion. The minimum sum of edge-weights necessary in order to achieve this is exactly the *minimum cut* between u and v. The correctness of the second special case of this reduction rule for OPT-VERTEX BIPARTIZATION is shown as follows: If G' is not bipartite, at least one vertex from G' must be deleted in order to bipartize G. Since $C \cup \{u\}$ is bipartite, deleting v leads to an optimal solution as any odd cycle in G that includes vertices from C can be destroyed by deleting v. We can remove C afterwards due to Reduction Rule 4, as u is a vertex separator of order 1 in G after the deletion of v.

Running time: Finding all C in G to which this reduction rule may be applied can be done in $O(|V|^2 \cdot |E|)$ time by iterating over all possible pairs $u, v \in V$ and then testing if the removal of u and v yields a bipartite connected component in G. For OPT-VERTEX BIPARTIZATION, $O(|V|^2 \cdot |E|)$ is then also the total running time for this reduction rule, as finding out whether all paths between u and v are even or odd can be done in constant time once we have found a bipartite component C in $G \setminus \{u, v\}$—we just have to look if both u and v are connected to equally colored vertices in C (in which case all paths are of even length) or to differently colored ones (in which case all paths between u and v via C would be of odd length). For OPT-EDGE BIPARTIZATION, we have to additionally perform the edge weight adjustment. The minimum cut between u and v in G' can be found, e.g., using the Ford-Fulkerson algorithm first presented in [FoFu62].[20] The running time for the Ford-Fulkerson algorithm is bounded by $O(|E| \cdot |V|)$ in this case[21], leading to a worst-case running time of $O(|V|^3 \cdot |E|^2)$ for this reduction rule when applied to OPT-EDGE BIPARTIZATION.

Reduction Rule 7 (SIMPLIFYING CYCLES OF SIZE 3): *For* OPT-VERTEX BIPARTIZATION*: Let there be three vertices u, v, and w in G that form a cycle of size 3, where w has degree 2. Let the degree of v be smaller than that of u. If the degree of v is at most 3, delete u from G, increasing count by one. (Note that afterwards, v and w can be removed from G due to Reduction Rule 2.)*

Example: Figure 6.7 provides an illustration for this reduction. Note that after the deletion of u, Reduction Rule 3 can be applied to the resulting graph.

Correctness: Since u, v, and w form an odd cycle c in G, at least one of these three vertices must be removed from G in order to bipartize G. Consider any odd cycle $c' \neq c$ in G. Observe that any c' that can be destroyed by removing w from G can also be removed by deleting either u or v from G. Now, if the degree of v is at most 3, any odd cycle other than c in G that includes v must include u as well. Therefore, there must be an optimal solution to OPT-VERTEX BIPARTIZATION on G that includes the removal of v.

Running time: This reduction rule can be carried out in $O(|V||E|)$ time using the following algorithm:

[20] The work by Ford and Fulkerson in [FoFu62] explores the important algorithmic area of *network flows*. The Ford-Fulkerson algorithm finds a *maximum flow* F in a graph G between two vertices u and v—the value of $|F|$ is equal to the minimum cut separating u and v according to the famous *max-flow min-cut theorem*.

[21] If F is a minimum cut for a graph G, the running time of the Ford-Fulkerson algorithm is bounded by $O(|E| \cdot |F|)$ according to [CLRS01]. Note that the minimum cut between u and v cannot be larger than $\max\{\mathit{degree}(u), \mathit{degree}(v)\} < |V|$, leading to the total $O(|E| \cdot |V|)$ worst-case running time.

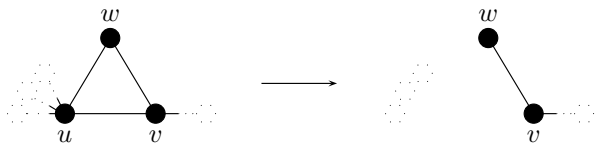


Figure 6.7: An example for Reduction Rule 7. Note that after applying this rule—i.e., after deleting u and increasing *count* by one—v and w may be removed from G due to Reduction Rule 2

Algorithm: OPT-VERTEX BIPARTIZATION, Reduction Rule 5
Input: A graph $G = (V, E)$
Output: G modified according to Reduction Rule 5

```
01  for each w ∈ V of degree 2 do
02      v ← neighbor of w with lower degree
03      u ← neighbor of w with higher degree
04      if {u, v} ∈ E then
05          if degree(v) ≤ 3 then
06              G ← G \ {u}
07              count ← count + 1
```

Reduction Rule 8 (SIMPLIFYING CYCLES OF SIZE 4): *Let there be a cycle c of size 4 in G whose vertices do not induce a cycle of size 3. If c contains exactly two vertices of degree 2 that are not connected,*[22] *then remove those two vertices and their edges from G. Add a new vertex z to G and connect the remaining two vertices from c—denote them by u and v—to z via two edges $e' := \{u, z\}$ and $e'' := \{v, z\}$. For* OPT-EDGE BIPARTIZATION*, the edge-weights are adjusted in the obvious way: In c, u and v are connected via two distinct paths. The weight of e' and e'' is set to the sum of the lowest weight edge in one path and the lowest weight edge in the other.*

Example: Figure 6.8 illustrates the different cases of this reduction rule.

Correctness: Since only the vertices u and v are connected to $G \setminus c$ and c is of even length, any odd cycle in G that contains vertices from c also includes u and v. This justifies the replacement of c for OPT-VERTEX BIPARTIZATION. For OPT-EDGE BIPARTIZATION, it is additionally important to note that in order to "disconnect" u and v we must remove an edge from each of the two distinct paths between u and v via c.

Running time: In order for this reduction to run fast, we need to find a way to detect cycles of size 4 in G that comply with this reduction rule. This can be accomplished in $O(|V|^2)$ time using the following algorithm:

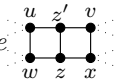
Algorithm: Finding cycles of size 4 for Reduction Rule 8
Input: A graph $G = (V, E)$
Output: All cycles of size 4 in G that comply with the requirements of Reduction Rule 8

```
01  for each w ∈ V do
```

[22] Note that the case where there are two vertices of degree 2 in c that are connected by an edge leads to the structure  which is already handled by Reduction Rule 5.

```
02    if w has degree 2 then
03        u ← first neighbor of w
04        v ← second neighbor of w
05        for each neighbor z of u do
06            if z has degree 2 an z ≠ w then
07                if z has u and v as neighbors then
08                    output {w, u, z, v}
```

For applying the reduction rule, instead of outputting the cycles we perform the actual reduction (this takes a constant amount of time). The algorithm requires $O(|V|)$ time each time lines *05* to *08* are called (we iterate over $O(|V|)$ vertices, note that lines *06* to *08* require only a constant amount of time since the degree of z is bounded by 2). Lines *02* and *03* obviously require a constant amount of time. Lines *02* to *08* are called $O(|V|)$ times by line *01*; therefore, the total running time of this algorithm is $O(|V|) \cdot O(|V|) = O(|V|^2)$.

Reduction Rule 9 (Simplifying Grids): *For* opt-Vertex Bipartization*: Let there be two cycles $c_1 = uwzz'$ and $c_2 = vxzz'$ in G (this leads to the ∞-like structure* 

). If z and z' have degree 3, u and v are not connected, and w and x are not connected by an edge, remove z, z', and their adjacent edges from G and add two edges $\{u, x\}$ and $\{w, v\}$ to G.

Example: Figure 6.8 provides an illustration for this reduction rule.

Correctness: Suppose that there is an optimal solution to Vertex Bipartization on G that includes the deletion of z. Now, the solution can only be optimal if one of the five other vertices w, u, v, x, or z' is included as well (note that otherwise, the deletion of z is redundant as there are still paths between u, w, and v, x). We can now simply show that independently of which of these five vertices is included in the solution, there is always a corresponding optimal solution that does not include the deletion of z:

- If the additional vertex is w, delete x instead of z.
- If the additional vertex is u, delete w instead of z.
- If the additional vertex is z', delete u and w instead of z and z'.
- If the additional vertex is v, delete x instead of z.
- If the additional vertex is x, delete w instead of z.

For each of these "deletion-substitutions," the following holds true: Any two vertices that were "disconnected" using the original solution are also disconnected in the new solution. Therefore, for any optimal solution to Vertex Bipartization that includes either z, z', or both, we have shown that there must then exist an equally sized optimal solution to Vertex Bipartization on G that includes neither the deletion of z nor that of z'.

Running time: In order to be able to efficiently carry out this reduction rule, an efficient algorithm for identifying the grid-like structure formed by c_1 and c_2 is needed. The following algorithm can find all such structures in a graph in $O(|V| + |E|)$ time by iterating over all vertices in G in $O(|V|)$ time to find a vertex z of degree three. For each neighbor z' of z

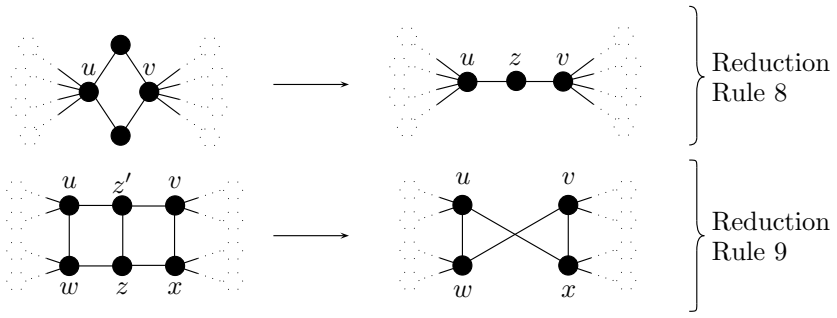


Figure 6.8: An illustration for Reduction Rules 8 and 9.

that has degree 3 as well, we check in $O(|E|)$ time whether the neighbors of z and z' can be grouped into two pairs where the vertices in each pair are connected by an edge.

By Reduction Rule 9, we conclude our presentation of reduction rules. The presented rules have been chosen mainly because they can be implemented rather efficiently and are expected to be applicable to a wide range of possible input graphs. Some rules have been given only for opt-Vertex Bipartization, because the reduction does not allow a correct representation of the edge-weights: For example, consider the grid in Reduction Rule 9. Deletion of the newly inserted edge $\{u, x\}$ after the reduction has the same effect as deleting $\{u, z'\}$ and $\{z, x\}$ in the original graph. The analogue holds true for the inserted edge $\{w, v\}$. So far, the edge-weights for $\{u, x\}$ and $\{w, v\}$ seem obvious, however, they are unable to handle the case where the only optimal solution to opt-Edge Bipartization on the input graph G involves the deletion of $\{z, z'\}$.

There are still some possibilities for future extensions of the algorithm. For example, for cliques[23] in the input graph, it is more efficient to perform the branching on which vertices/edges are *not* deleted rather than on which *are deleted*. This is due to the observation that for a clique of size k, at least $k - 2$ vertices need to be deleted in order to bipartize that clique; for opt-Edge Bipartization, at most $\frac{k^2}{4}$ of the $k^2 - k$ edges can be kept in the graph, because a bipartite graph with k vertices may not contain more than $\max_{i \geq 0} \frac{k-i}{2} \cdot \frac{k+i}{2} = \frac{k^2}{4}$ edges. The problem with a reduction rule based on cliques is that we need to find an efficient way to find large cliques—but finding a maximum clique in a graph is not only *NP*-complete but also hard to approximate (similar to Graph Bipartization, there exists no *PTAS*[24] for clique detection unless *P=NP*). Moreover, since the graphs in this work originate from biological sources and edges represent *conflicts* or *flaws* in the original data, we would not expect to see large cliques occurring in the input graph to our branch&bound algorithm.

Extensions of the above reduction rules to edge- or vertex-separators of a higher order than 2 do not seem useful and/or possible: Finding large edge- or vertex-separators in a graph should be possible e.g., using maximum-flow techniques (see Reduction Rule 6), but the main argument employed above that "an odd cycle in G must contain either all or none of the separator elements" does not work for higher order separators because there might be odd cycles in the input graph that contain only a subset of the separator.

[23] A clique in a graph is a subgraph where every vertex is connected to all other vertices by an edge. A clique of size k therefore contains $k(k-1) = k^2 - k$ edges.

[24] Refer to footnote 4 on page 71 for a definition of *PTAS*.

6.4 Implementation and Comparison of the Algorithms

It was already emphasized at the beginning of the last section that branch&bound algorithms provide no guarantees that their running time is better than an exhaustive search of all possible solutions to a given problem. Therefore, in order to determine the efficiency of the developed Graph Bipartization-algorithms, they were implemented in the *Java*[25] programming language (version 1.4.1) and tested on random graphs as well as graphs corresponding to the SNP-related problems presented in the next chapter. In this section, we first introduce the usage of the software-package, followed by some implementation details (e.g., the data structure for representing the graph). This section is concluded by the results from testing the implementation on various random graphs, the results concerning SNP-related problems are presented in Section 7.4. As we will see then, although the algorithms do not allow us to efficiently solve Graph Bipartization on random graphs in general, they are efficient for solving the graphs arising during the analysis of SNPs.

6.4.1 Using the Program

The bipartization-software was designed with the experimental testing of the algorithms in mind. It therefore provides detailed statistics concerning the size of the search tree, various running-times, and reduction rule usage. Whilst the output files are provided in a very readable format, the user-interface is only a command-line interface (rather than a graphical one) in order to simplify automatic batch testing for many graphs and keep the measurements free from interferences with *Java*'s rather slow graphical output capabilities.

Input File Format: The format for the input file is straightforward and should easily be convertible to other graph formats such as that of the *LEDA* graph-library[26]. The file specifying the graph must be written in plain ASCII-text and is build up as follows (text that must literally appear in the input file is written in a `typewriter`-font):

01 `# Graph name`
02 The name of the graph
03 `# Number of Vertices`
04 The number n of vertices in the graph.
05 `# Number of Edges`
06 The number m of edges in the graph.
07 `# Vertex Names`
08 The names for the graph's vertices, each one in a separate line.
⋮
08+n `# Edges`
09+n For each edge in the graph, a separate line contains the connected vertices' names separated by a space character.
⋮
09+n+m `# EOF`

[25] A good overview of *Java* is, e.g., [Flan02]. Sun Microsystems™ provide a free software-development kit (SDK) for *Java* on their webpage [SuMi03].

[26] *LEDA*, a library written in the C++-programming language, is widely regarded as one of the *"the best-designed general-purpose graph data structure[s] currently available"* [Skie98]. More information about *LEDA* can be found in [MeNä99].

Note that the vertex names in lines *08* to *08+n−1* may not contain any spaces (such as the simple space character or a tabstop).

Starting the Program: The program is started by calling

```
java Bipartize infile outfile [E V R H<int> A]
```

where `infile` is the name of the file containing the graph that is to be bipartized, and `outfile` specifies the name of the file to which the results from the branch&bound program will be written to. Additionally, the following flags are recognized:

- `E` makes the program solve OPT-EDGE BIPARTIZATION on the input-graph.
- `V` makes the program solve OPT-VERTEX BIPARTIZATION on the input-graph.
- `R` turns off data reduction during branch & bound (the initial data reduction is still performed).
- `H` is only valid for OPT-EDGE BIPARTIZATION and must be directly followed by an integer. This integer specifies the number of runs for the initial (randomized) recoloring heuristic. Since the heuristic for OPT-VERTEX BIPARTIZATION is deterministic, this flag has no effect when used together with the `H` flag.
- `A` terminates the program after the initial heuristic was executed, writing the approximate solution to the output file.

Note that exactly one of the flags `E` and `V` must be specified by the user, all other flags may be specified at will.

Output: The detailed results of the bipartization as well as statistics concerning, e.g., search tree size and usage of reduction rules, are written to the specified logfile.

6.4.2 Some Implementation Details

In this subsection, we introduce the basic implementation-scheme and some details concerning, e.g., the used data structures of the GRAPH BIPARTIZATION-software package. The implementation consists of 16 classes with a total of about 2600 lines of code; the relations of the central 11 classes are provided as a *UML* diagram[27] in Figure 6.9 which shows the basic structure of the implementation.

The most important fact to notice in the implementation is that vertices and edges in the graph can only be created but never deleted using the methods provided by the Graph class itself. Methods for deleting vertices and edges from the graph are only provided by the GraphModifier class, which can be created from a Graph using the Graph.modifier() method. Once an element from the graph has been deleted by a GraphModifier, it cannot be restored directly but only through using the GraphModifier.undoStep() method. This ensures that elements in the graph can only be undeleted in the reverse order that they were

[27] *UML* is a standard for showing class-relations for programs written in object-oriented languages such as *Java* or *C++*. The standard is specified by the Object Management Group [OMG03]. A good introduction to *UML* is, e.g., [Page99].

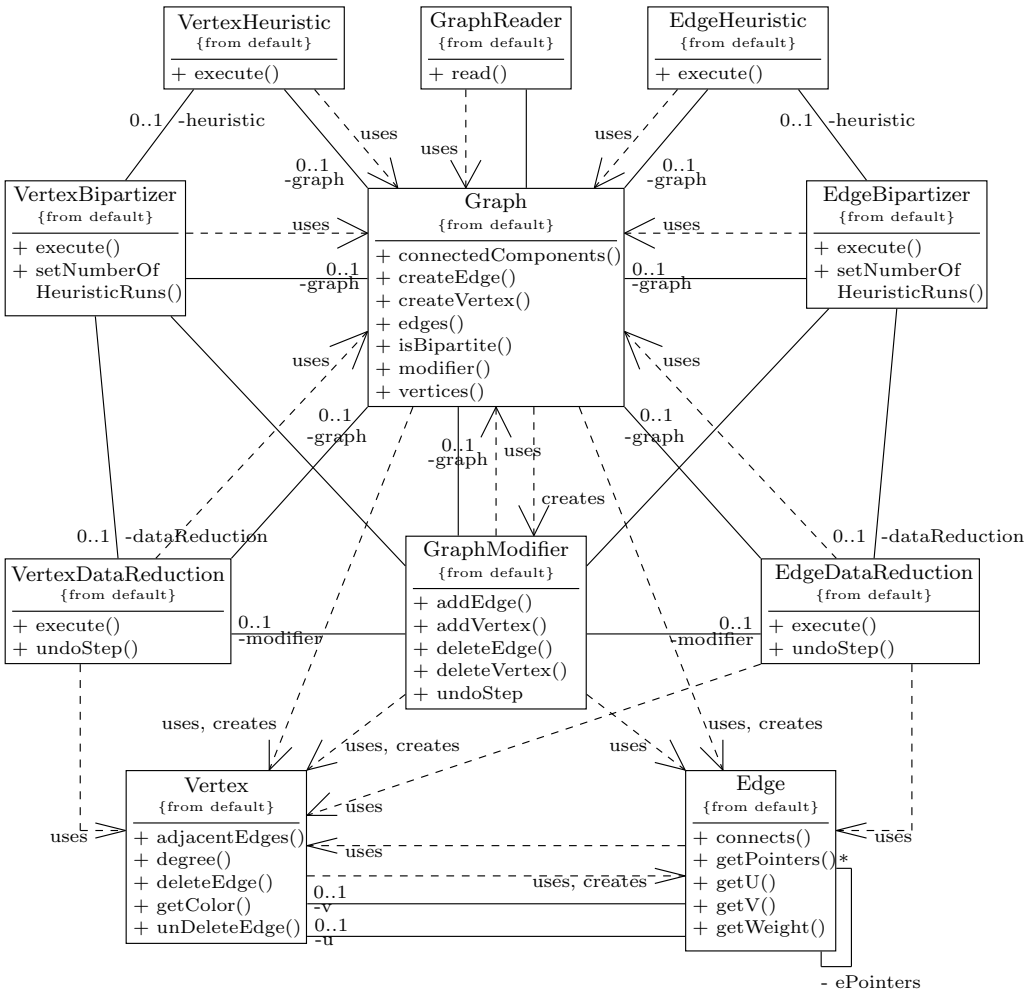


Figure 6.9: UML-diagram for the implementation of the branch&bound algorithm. Note that for each class, only the "relevant" (meaning relevant to understanding the basic program function) portion of public methods is shown in order to keep the diagram readable.

deleted, i.e., if an element a was deleted earlier than an element b it can only be undeleted once b has been undeleted. Observe that this deletion-scheme is sufficient for the graph-modifications performed by a branch&bound algorithm on the graph,[28] allowing us to work with only a single copy of the graph in memory[29] and thus avoiding the computational cost in terms of time and memory required for creating many duplicates of the graph.

The most important benefit of the GraphModifier's delete-undelete scheme is the speedup of the deletion and undeletion itself. Making use of the fact that the insertion of vertices and edges into the graph is only performed during initialization of the branch&bound algorithm, the implementation trades a computationally more time-expensive graph-creation process for the gain of computational speed during the branch&bound process itself: After the

[28] Recall the algorithm presented on page 75.

[29] Keeping the information necessary for the undelete-operation can be done very efficiently, as we will see shortly

graph has been created, a vertex v can be deleted and undeleted in $O(degree(v))$ time from the graph, edges can even be deleted and undeleted in $O(1)$ time. In order to achieve this, a double-linked list is constructed for the graph elements whose individual elements can be accessed in constant time via a HashMap.[30] Due to the stack-like structure of the deletion-undeletion scheme, pointers in the double-linked list can be modified in constant time because we can be sure that before an undelete-operation is performed for a specific list element, the list has exactly the same buildup it had right after the corresponding deletion. Since the graph elements are never really deleted but only shut off from being accessed, storing the undelete-information only requires to store pointers to the deleted elements, which never occupies more memory than the graph itself.

Some other details to notice in the implementation are the following:

- The VertexBipartizer class provides a method setNumberOfHeuristicRuns() only to maintain a common interface with the EdgeBipartizer class—since the heuristic for opt-Vertex Bipartization is deterministic, executing it more than once would be redundant.
- Each edge can be assigned an array of edge-pointers by the Edge-constructor. This is important for the opt-Edge Bipartization reduction rules, because these reduction rules may replace multiple edges with a single new one. This new edge then contains a reference to the edges it represents by its edge pointer, which greatly simplifies the output of results. If no edge-pointers are specified during construction, an edge only has a pointer to itself.
- Since each vertex maintains a double-linked list of its adjacent edges, the class Vertex provides the delete() and unDelete() methods to manipulate this list when an adjacent edge or vertex is removed from the graph.

Some possible future improvements to the presented implementation will be discussed towards the end of this section following the presentation of the practical tests' results.

6.4.3 Tests and Test Results

In this subsection, we present results concerning the performance of the branch&bound implementation on random graphs. Results concerning the performance on graphs related to the SNP problems introduced in the next chapter are given in Section 7.4.

Methodology The practical experiments on random graphs can be divided into two parts: In the first part, random graphs (*RGs*) were generated and then bipartized. In the second part, random *bipartite* graphs (*RBGs*) were generated, followed by an addition of a fixed number of *de-bipartizing* edges or vertices, where de-bipartizing edges connect two red or two blue vertices and de-bipartizing vertices are connected to both some red and some blue vertices.

Both the RGs and the RBGs were generated by the following algorithm that is used to generate random graphs in *LEDA* [MeNä99]: First, a list of all possible edges in the respective graph is created. Then, the desired number of edges in the graph is chosen randomly

[30] See, e.g., [CLRS01] for more details on hash-based data structures

from this list and inserted into the graph. The de-bipartizing elements were inserted analogously.[31] Using these random graphs, the following measurements were made for opt-Edge Bipartization and opt-Vertex Bipartization:

- The running-time and search tree size when bipartizing a RG with 20 vertices, relative to the average vertex degree. Each measurement was performed twice, once with and once without the application of data reduction during branch & bound (initial data reduction was always performed).
- The running-time and search tree size when bipartizing a RG with an average vertex degree of 3, relative to the number of vertices. Each measurement was performed twice, once with and once without the application of data reduction during branch&bound (initial data reduction was always performed).
- The running-time and search tree size when bipartizing a RBG with 20 vertices, relative to the average vertex degree and the number of de-bipartizing elements.
- The running-time and search tree size when bipartizing a RBG with an average vertex degree of 3, relative to the number of vertices and the number of de-bipartizing elements.

Additionally, we evaluate the average usage and performance of the reduction rules and the performance of the initial heuristics.

Since in a first run of the experiments, the reduction rules based on separators (Rules 3, 4, and 6) were almost never applicable (although computationally expensive), it seems that the general structure of random graphs makes separators of small order very improbable (see the discussion towards at the end of this section). The respective reduction results were therefore switched off for all measurements in order to facilitate the testing of more instances.

With the purpose of obtaining a good estimation of the average-case running-time and search tree size during bipartization, each measurement was performed on 10 different RGs/RBGs with the same parameters (number of edges, vertices and de-bipartizing elements), leading to a total of approximately 6 000 bipartized graphs. All results were obtained on a machine equipped with a 2.4 GHz Intel® Pentium® IV Processor and 512 MB physical memory, running Red Hat™ Linux. The software was compiled using the *Java* SDK 1.4.1 from Sun Microsystems™ [SuMi03].

Results The recoloring heuristic almost always found a solution that is within 20% of an optimal solution (with the average difference being around 15%) when executed 10 times. This figure can be decreased further to an average difference of less than 10% by executing the heuristic approximately 25 times.[32] The heuristic for opt-Vertex Bipartization did not perform as well although satisfactory: It almost always finds a solution that is within 50% of the optimal solution, with the average heuristic solution being about 40% larger than an optimal one. It should be noted that the opt-Vertex Bipartization heuristic is

[31] Note that a graph with, e.g., n added de-bipartizing edges may still have a solution to opt-Edge Bipartization that is smaller than n because it might be bipartized more efficiently by deleting other edges other than the inserted ones. In the following results, we will say that a graph has n added de-bipartizing elements if only if the solution to the corresponding opt-Graph Bipartization problem really has size n.

[32] Cases where the recoloring heuristic might not perform well are discussed in [SMVS97]. Interestingly, some of these cases are handled by the reduction rules from this chapter.

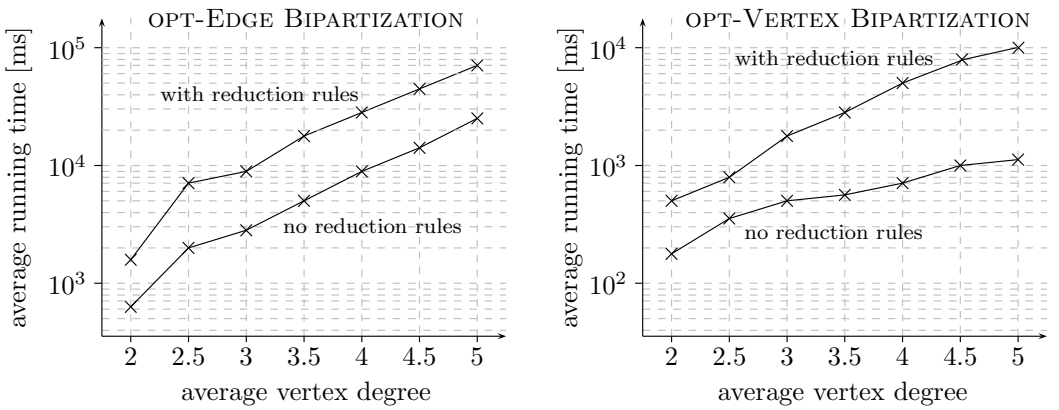


Figure 6.10: Running time for the implemented Graph Bipartization-algorithms on a graph with 20 vertices and varying average vertex degree. Note how the overhead due to the data reduction algorithms causes the total running time to increase up to 3 and 10 times for opt-Edge Bipartization and opt-Vertex Bipartization, respectively.

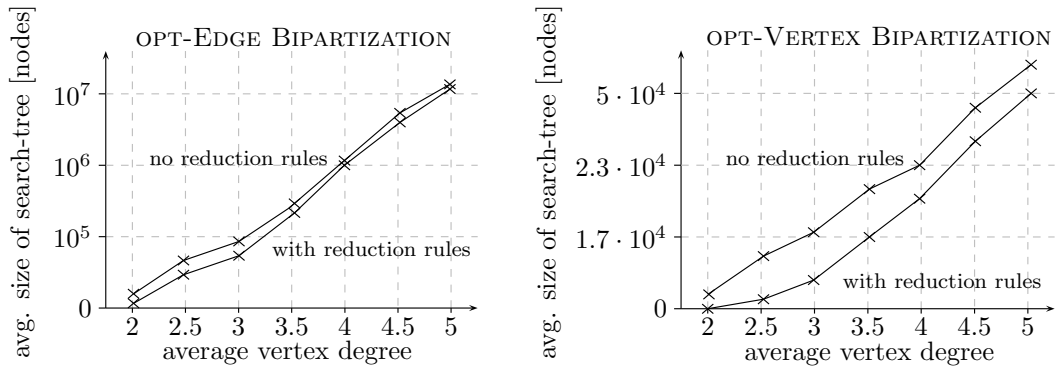


Figure 6.11: Search tree size for the implemented Graph Bipartization-algorithms on a graph with 20 vertices and varying average vertex degree. The reduction rules turn out to be most effective for graphs with an average vertex degree of 3, especially in the case of opt-Vertex Bipartization.

deterministic and therefore only executed once, a randomized variant that can be executed multiple times would probably perform as well as the recoloring heuristic.

The running time required for opt-Edge Bipartization and opt-Vertex Bipartization on a graph with 20 vertices and varying average vertex degree is shown in Figure 6.10, the corresponding search tree sizes are given in Figure 6.11. Since the search tree sizes correlate with the running time of the algorithms, only the measured running times of the subsequent experiments are shown in this work.

Figure 6.12 displays the exponential increase in running time for Graph Bipartization when the number of total vertices in the RG increases linearly. In Figures 6.13 and 6.14, we show the measured running times for the experiments on RBGs.

The average reduction rule usage was measured to be as follows:

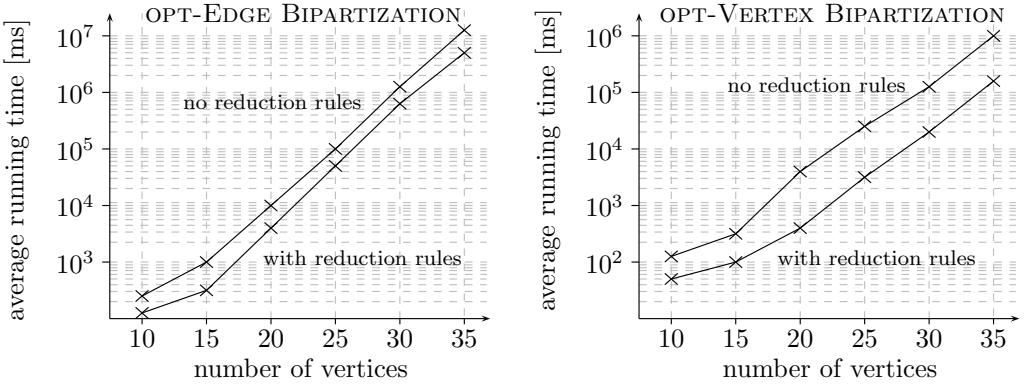


Figure 6.12: Running time for the implemented Graph Bipartization-algorithms on a graph with average vertex degree 3 and a varying total number of vertices.

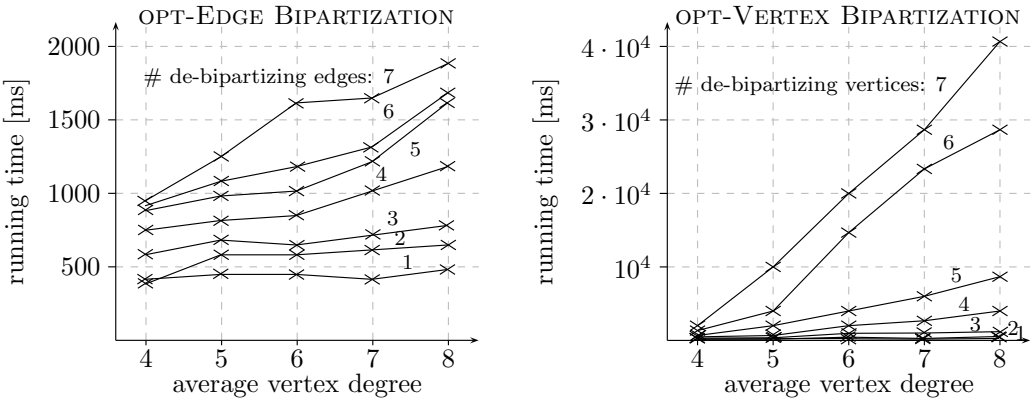


Figure 6.13: Running time for the implemented Graph Bipartization-algorithms on graphs with 20 vertices and varying average vertex degree. The respective running times increase only linearly with the average vertex degree for a fixed number of de-bipartizing elements.

Rule	opt-Edge Bipartization	opt-Vertex Bipartization
1	6.1%	25.9%
2	68.1%	33.7%
3	(off)	(off)
4	(off)	(off)
5	15.2%	8.6%
6	(off)	(off)
7	not applicable	10.8%
8	10.7%	5.5%
9	not applicable	5.6%

Discussion The first observation we can make in Figure 6.10 is that although all reduction rules that were turned on during the measurements were used (with a relative usage of more than 5%), the running times for both the opt-Edge Bipartization and the opt-Vertex Bipartization solving program are slower by a factor of almost ten with reduction rules

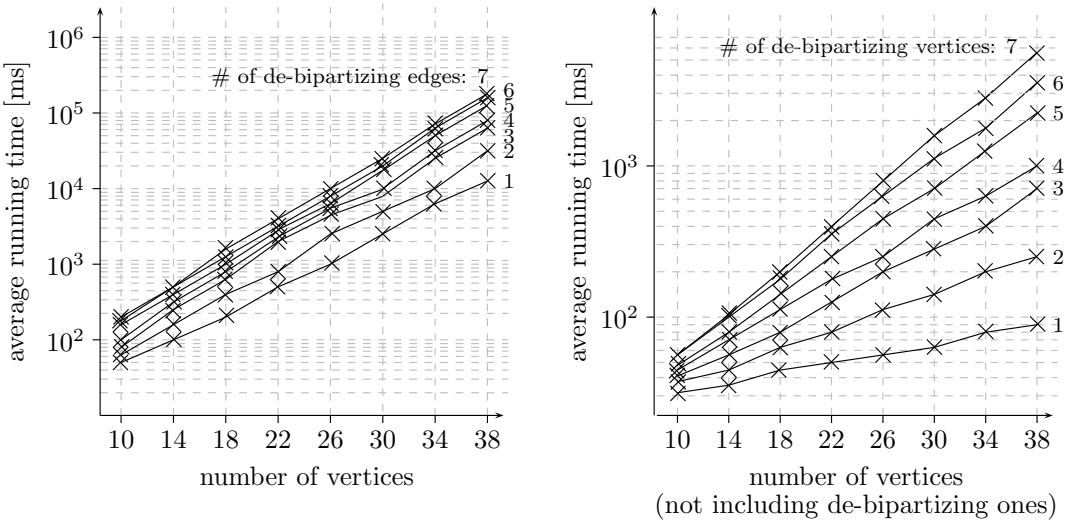


Figure 6.14: Running time for the implemented Graph Bipartization-algorithms on graphs with an average vertex degree of 3 relative to the total number of vertices.

turned on. This can be explained by looking at the corresponding tree-size in Figure 6.11: The reduction rules are only capable of at best halving the total search tree size, but—as is to be expected from the runtime-analysis in the previous section—they consume more time than they can make up for by this reduction. Another interesting fact to note in Figure 6.10 is the exponential increase in running time with a linear increase of the average vertex degree—at first sight this would not be expected since the algorithms' and reduction rules' running time did not indicate such a behavior in our previous run-time analysis. At a second glance, however, note that the more "dense" the RGs become in terms of edges, the average number of deleted edges/vertices required to bipartize the graph increases as well.[33]

Figure 6.14 shows that—as was to be expected—the average running time for our algorithms increases exponentially with the total number of vertices in the RBG. Note, however, that this increase is not as bad as would be expected: In a graph with 5 more vertices than a given graph, a complete search of all possible solutions to a given Graph Bipartization problem would approximately take $2^5 = 32$ times as long as for the original graph. The observed increase however is by only about a factor of 10 (instead of 32) for 5 additional vertices. This figure does not even consider that, as in the previous experiments, the average size of the optimal solution increases with the total number of vertices. Once we take account of this fact by looking at the algorithms' running times when the number of de-bipartizing elements is kept constant (Figure 6.14), we see that the observed increase-factor is even lower: For Vertex Bipartization, it varies from about 1.3 for 5 additional vertices (1 de-bipartizing vertex) to 2 for 5 additional vertices (7 de-bipartizing vertices).

In Figure 6.13 we can see—as was to be expected— that for a fixed number of de-bipartizing elements, the algorithms' running times increase only linearly with the average vertex degree (i.e., total number of edges).

Overall, the developed algorithms show a rather long running time already for moderately sized RGs and RBGs. In some preliminary tests for opt-Edge Bipartization, the

[33] E.g., in the experiments for opt-Edge Bipartization on a RG with 20 vertices, an increase in the average vertex degree by 0.5 increased the size of the average optimal solution by approximately 4 edges.

Max2Sat software by Gramm mentioned at the beginning of this chapter outperformed the opt-Edge Bipartization algorithm by a factor of up to 10 on the random graph instances. Bear in mind however, that graphs corresponding to SNP problems may have some special properties we might not expect in random graphs:[34] Intuitively, it seems likely that the special structures such as separators, long paths, etc. are not present initially in the RGs and RBGs constructed, but rather emerge close to the leafs of the search tree, when the analyzed graph is small and many edges/vertices have already been removed from it. In order to test this hypothesis, a variable was added to the implementation counting how close to the leaf the reduction rules concerning separators could be applied, it was possible to see that these reduction rules are—if at all—mostly applicable about 3 to 4 levels up from a leaf on a RG with 20 vertices and an average vertex degree of 3. The application of reduction rules (especially those separating the graph into several small subgraphs) is however most effective close to the root of the search tree.

As we shall see in Section 7.4, the reduction rules developed in this section can—although being hardly applicable to random graphs—significantly reduce the running time of opt-Edge Bipartization and opt-Vertex Bipartization on graphs obtained from the SNP analysis problems of the next chapter, enabling us to bipartize these graphs even when they contain a few hundred vertices.

[34]The reasons for this will be discussed later in Section 7.4.

Chapter 7

Using Graph Bipartization in SNP Analysis

In the previous chapter, algorithms for opt-Vertex Bipartization as well as opt-Edge Bipartization were developed. It was already mentioned then that graph bipartization has a broad field of applications. This chapter will discuss two recently posed problems in SNP analysis that will turn out to be closely related to Graph Bipartization. The first problem—analyzed in Section 7.2—concerns the problem of selecting fragments from a diploid[1] DNA in order to obtain two consistent haplotypes during sequencing. Section 7.3 concerns a more indirect approach to detecting SNPs: Based on the assumption that SNPs have evolved according to a perfect phylogeny (see Chapter 5), we use a set of genotypes which are then *resolved* into the—presumed—underlying haplotypes of the genotype set. At the end of this chapter, we test the Graph Bipartization algorithms developed in the last chapter on graphs arising from SNP problems, showing that they are often capable of efficiently solving these graphs even if they contain a few hundred vertices.

7.1 Introduction and Overview of Results

Modern DNA sequencing techniques are only capable of sequencing DNA fragments of at most 1 000 bases in length. If the DNA of a diploid organism is sequenced we obtain slightly different fragments at SNP sites. However, for reassembly of the fragments it is vital to be able to distinguish fragment differences due to SNP sites in the diploid source sequence from those that are caused by errors in the sequencing process. In order to separate the two from each other, we will use a minimality argument (i.e, most fragments are presumed to have been read correctly) proposed in [RBIL02], leading to the Minimum SNP Removal and Minimum Fragment Removal problem (Definitions 7.2 and 7.3, respectively). The latter problem is shown to be parameter equivalent to Vertex Bipartization in Corollary 7.6. For the former problem we show that this problem is at least as hard as Edge Bipartization (Theorem 7.5). For reasons that are discussed at the end of Section 7.2, a reduction from Minimum Fragment Removal does not appear to be possible, leaving it an open problem to find an upper hardness bound for Minimum SNP Removal.

[1] Recall from Chapter 2 a human cell contains two copies of each non-sex chromosome.

Section 7.3 uses a different approach to obtain SNP data: For current sequencing techniques, it is often infeasible (e.g., due to prohibitively high cost and labor) to directly sequence haplotype data. Instead, genotype data is obtained, i.e., we know for certain sites that a SNP occurs but cannot tell which haplotype actually shows which base. Given a whole set of genotypes, however, it has been proposed to infer the haplotypes that probably cause the observed genotypes by assuming that all haplotypes must have evolved according to a perfect phylogeny [Gusf02] (the reasons to justify this assumption are given in Section 7.3). Analogously to the fragment removal problems, we analyze the complexity of the problems that arise when dealing with reading errors and/or the case where some haplotypes have not evolved in a perfect phylogeny. It will be shown that the Minimum Genotype Removal problem (Definition 7.8) is at least as hard as Edge Bipartization (Theorem 7.9) and the Minimum Site Removal problem (Definition 7.10) is parameter-equivalent to Vertex Bipartization (Theorem 7.11). Due to analogue reasons as in the case of Minimum SNP Removal, the existence of a parameterized or parameter-preserving reduction from Minimum Genotype Removal *to* Edge Bipartization remains open.

Concluding this chapter, we test the algorithms for Vertex Bipartization that were developed in the last chapter on some instances of Minimum Fragment Removal and Minimum Genotype Removal, showing that these two problems can generally be efficiently solved even if the corresponding Vertex Bipartization instance contains a few hundred vertices.

7.2 SNP Haplotype Assembly

Current methods in DNA sequencing are not powerful enough to sequence a whole strand of DNA but only fragments of at most 1 000 bases in length. In order to still be able to sequence longer strands of DNA, a technique called *shotgun sequencing*—invented by Sanger around 1980 (e.g., see [SCHHP82])—is employed.[2] Shotgun sequencing proceeds as follows: The source sequence is first copied many times using PCR[3] and then more or less randomly broken into fragments, e.g., using DNA cutting enzymes called endonucleases. These fragments are then sequenced individually.[4] Afterwards, the resulting small sequences need to be re*assembled* to obtain the whole DNA strand's sequence. Assembly is a computationally quite involving task that has, e.g., to deal with read errors of the fragments obscuring a correct alignment or with repeated regions in the DNA strand that produce very similar fragments. In order to overcome some of this difficulties, fragments are often generated in pairs and therefore contain some extra information concerning their relative distance on the source sequence (this technique is known as *double barrel shotgun sequencing* [RBWL95, WeMy97]).

Recall that the human genome is diploid. Therefore, when sequencing a human genotype, due to the high similarity in the two chromosomes and errors in the sequencing and assembly process, it is often not possible to directly infer the pair of haplotypes that gives rise to the observed genotype from the given fragments. Since for a certain location on the genome, a fragment can take at most two different values (e.g., in a heterozygous SNP site), a fragment from the sequencing read will from now on be represented as a string over the alphabet $\{0, 1, ?\}$. The symbols 0 and 1 will be used to represent the two different possibilities

[2] For more information on shotgun sequencing see, e.g., [WeMy97] or Chapter 7 of [Wate95].

[3] For more information on PCR, refer to the footnote concerning PCR on page 8.

[4] Details on the sequencing of individual fragments such as sequencing by restriction endonucleases, chemical cleavage, or the chain-terminator method may be found, e.g., in Chapter 28 of [VoVo95].

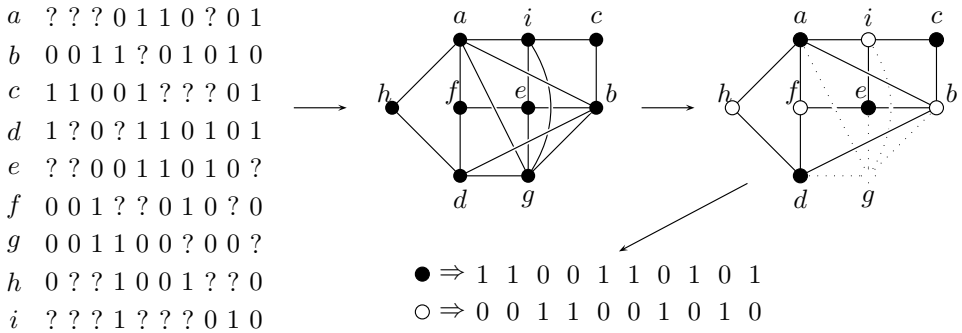


Figure 7.1: An example for Minimum Fragment Removal on a set of fragments: The upper left matrix represents the reads of 9 fragments $a, \ldots, f$. From this matrix, the fragment conflict graph G_F is constructed (first arrow). Then, G_F is bipartized by removing the vertex g. From the resulting bipartite graph, we can directly infer which fragments belong to which haplotype from the coloring of the respective vertices. The resulting haplotypes when corresponding vertices are merged are shown below the conflict graphs (third arrow).

for a base at a certain SNP site. If a read for a certain site has not been made,[5] it is represented by a "?". In order to obtain (*infer*) the underlying pair of haplotypes from a set F of fragments, [LBILS01] introduces the so-called *fragment-conflict graph* G_F.

Definition 7.1 (Fragment-Conflict Graph):
Given a set F of fragments, the fragment-conflict graph G_F represents each fragment as a vertex, and connects two fragments a and b by an edge if there exists a SNP site for which a and b have explicitly different[6] *reads.*

It is clear that in order to infer a pair of haplotypes from the given fragments, we must be able to partition the set of vertices in G_F into two subsets such that there exist no conflicts (i.e., edges) between two vertices in the same subset. This is exactly the case when G_F is bipartite; which we shall use to expose the following two problems' relationships to Graph Bipartization:

Definition 7.2 (Minimum SNP Removal):
Given a set of fragments, what is the minimum number of SNP sites that need to be ignored in order for the fragments to be resolvable into two haplotypes?

Definition 7.3 (Minimum Fragment Removal):
Given a set of fragments, what is the minimum number of fragments that need to be removed from the set in order for the remaining fragments to be resolvable into two haplotypes?

An example for Minimum Fragment Removal and the resulting haplotypes is given in Figure 7.1. We have already shown that in order to be able to reassemble the individual fragments from the sequencing process, the corresponding conflict-graph must be bipartite.

[5] Note that this will be the case for many sites, as the length of a read is generally a lot shorter than that of the actual sequence

[6] I.e., if either a or b is a "?", the two fragments are not connected.

In the remaining part of this section, we show that this leads to a close linkage between Graph Bipartization and Minimum SNP Removal/Minimum Fragment Removal, using the following lemma:

Lemma 7.4 *For any given graph $G = (V, E)$, it is possible to construct a set F of fragments over the alphabet $\{0, 1, ?\}$ in polynomial time such that the conflict graph G_F is isomorphic*[7] *to G.*

Proof Let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. Then, we construct n fragments (each one corresponding to a certain vertex) of length m, such that the ith site in the fragment representing v_i has value "1", all fragments representing vertices to which v_i is connected by an edge take value "0" for site i, and all other vertices take value "?". Then, if we construct G_F for the fragments, we obtain a graph with n vertices where two vertices are in conflict (i.e., connected by an edge) if and only if they were connected by an edge in the original graph. □

Theorem 7.5 Vertex Bipartization *is parameter-preservingly reducible to* Minimum Fragment Removal. Edge Bipartization *is parameter-preservingly reducible to* Minimum SNP Removal.

Proof In the reduction of Lemma 7.4, we obtain a set of fragments where there is exactly one fragment for every vertex in the original graph and exactly one SNP site corresponding to every edge in the graph. This "1:1-relationship" ensures that if the set of fragments can be resolved into haplotypes by removing at most k fragments (SNP-sites) from it, then the corresponding conflict graph (which is isomorphic to the graph we want to bipartize) can be bipartized by deleting at most k vertices (edges) from it. Note that the reduction in Lemma 7.4 is clearly computable in polynomial time with respect to the size of the input graph. □

From the above theorem, we can deduce the following corollary:

Corollary 7.6 Minimum Fragment Removal *is parameter-equivalent to* Vertex Bipartization.

Proof The parameter-preserving reduction from Vertex Bipartization to Minimum Fragment Removal is given by Theorem 7.5. For the reverse direction, note that each fragment in a given Minimum Fragment Removal instance always corresponds to exactly one vertex in the fragment-conflict graph. Therefore, if the conflict graph can be bipartized by deleting at most k vertices from it, deletion of the corresponding fragments in the given Minimum Fragment Removal instance allows the fragments of this instance to be resolved into haplotypes. □

The above corollary 7.6 allows us to use the Vertex Bipartization-algorithm from Chapter 6 to efficiently solve this problem, as will be shown later in Section 7.4.

Note that a result similar to Corollary 7.6 for Minimum SNP Removal (i.e., the reduction to Edge Bipartization) is all but obvious and therefore remains an open problem in the

[7] Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *isomorphic* if there is a bijection $\Phi : V_1 \to V_2$ such that $\{v_a, v_b\} \in E_1 \Leftrightarrow \{\Phi(v_a), \Phi(v_b)\} \in E_2$.

scope of this work: E.g., observe the edge $\{b, g\}$ in Figure 7.1. The 9th SNP site is "responsible" for this conflict edge. However, observe how by ignoring this site for the construction of the conflict graph (in order to delete $\{b, g\}$), we implicitly delete the edge $\{h, i\}$ as well. There are other problems as well, e.g., multiple SNP sites causing the same edge in the conflict graph (e.g., the edge $\{b, g\}$ is caused by the 3rd, 4th, and 7th SNP site).

7.3 Inferring Haplotypes from Genotypes

At the beginning of this work we have introduced SNPs, variations in a single nucleotide along multiple copies of the same DNA segment of different organisms and within the haplotypes of a diploid organism. Today, sequencing techniques that are commonly used in practice yield genotype instead of haplotype information. The direct inference of haplotypes is—although possible [PBH01]—often not considered for practical use due to cost and speed considerations. The problem that arises with this is the following: Techniques that only provide genotype instead of haplotype information give the bases for both SNPs at a certain site but do not specify the chromosome on which each of them appears. This is no problem if two haplotypes are homozygous (i.e., they contain the same base at the given site). If, however, the genotype is heterozygous, problems arise as becomes clear from the following example: Consider a genotype sequence $TXXG$, where X stands for a SNP in the individual haplotypes that contains A in one haplotype and C in the other. Then, the observed genotype can be resolved into haplotypes in four different ways:

$$TXXG \longrightarrow \begin{matrix} TAAG \\ TCCG \end{matrix}, \quad TXXG \longrightarrow \begin{matrix} TCCG \\ TAAG \end{matrix},$$

$$TXXG \longrightarrow \begin{matrix} TCAG \\ TACG \end{matrix}, \quad \text{or} \quad TXXG \longrightarrow \begin{matrix} TACG \\ TCAG \end{matrix}.$$

Observe that as in the above example, a SNP site in a single genotype may show at most two different bases. We have already mentioned in Chapter 2 that even throughout a whole set of genotypes, SNPs often occur in only two variations (this is confirmed by the data used for practical tests in Section 7.4). For each SNP site, we shall therefore label one of the two possible variations "0" and the other one "1", allowing us to represent genotype information for a SNP site by one of three states:

- "0" represents a site where a 0 occurs in both haplotypes,
- "1" represents a site where a 1 occurs in both haplotypes, and
- "2" represents a site where a 0 occurs in one haplotype and a 1 in the other.

A genotype representing m different SNP sites can then be written as a row vector (*genome vector*) with m entries from the alphabet $\{0, 1, 2\}$. A *haplotype vector* (in this context, we will often use the term "haplotype" for means of abbreviation) is a row vector with entries from the alphabet $\{0, 1\}$. This section will deal with the problem that, given n genotypes of length m, we would like to find the corresponding haplotypes to those genotypes such that for each observed genotype the result provides two haplotypes that *explain* this genotype. As has been illustrated above, this is not possible without any further constraints (recall that there were four ways to resolve even a single genotype into two haplotypes). It has been suggested as a reasonable constraint that the resulting haplotypes must obey a model of perfect phylogeny [Gusf02]. This model will predict the correct haplotypes for the given genotypes under two assumptions:

1. The most arguable assumption is that there is an absence of recombination. Under this assumption, each haplotype sequence is derived from a single ancestor in the previous generation and their evolutionary history will form a tree structure [Huds90]. The absence of recombination is justified by Gusfield by the biological observation that often, in long blocks of genetic data, recombination seldomly occurs. It seems that including recombination into the models for inferring haplotypes will make the corresponding computational problems a lot harder to solve as a whole new multitude of combinatorial unknowns (i.e., the sites of recombination) are introduced.

2. The so-called "infinite sites" assumption: Since we cannot detect multiple mutations that occurred during evolution at a single SNP site, the assumption is made that at most one mutation has occurred within the evolutionary timeline for that SNP. According to [Gusf02], empirical data supports this assumption.

Gusfield [Gusf02] furthermore provides three reasons to justify a model of perfect phylogeny, the most convincing of which is a success in practical applications: The program PHASE, one of the first available programs for inferring haplotypes from genotypes, generates data most effectively, i.e. the predicted haplotypes are often correct, if it is assumed that the generated haplotypes obey a model of perfect phylogeny [SSD01].

An efficient algorithm to solve the problem of inferring haplotypes from genotypes in order to obtain a perfect phylogeny was given by Eskin, Halperin, and Karp in [EHK03] (experimental studies of the algorithms are given in [HaEs03]). This algorithm directly relates the inference of haplotypes from genotypes to Graph Bipartization problems, as this section will show. For a formal definition of the problem, we assume as was justified above that there are only two possible nucleotides for any given SNP site[8], one of these two possibilities is labeled 1, the other 0. A genotype (made up of two haplotypes) is then represented as a row vector representing m sites with entries from the alphabet $\{0, 1, 2\}$ as already introduced above. For example, if the haplotypes $(\,0\,1\,0\,1\,1\,1\,0\,1\,0\,0\,1\,0\,1\,1\,)$ and $(\,1\,0\,0\,1\,1\,0\,0\,0\,1\,1\,1\,1\,1\,0\,)$ occur in a genome we would only see the row vector $(\,1\,2\,0\,1\,1\,2\,0\,2\,2\,2\,1\,2\,1\,2\,)$. Each pair of haplotype vectors that generates the given genome vector is called *compatible* with that genome vector. There is, as has been pointed out at the beginning of this section, quite a multitude of ways to resolve a genome vector into two compatible haplotype vectors (hence the restriction to a perfect phylogeny of the individual haplotypes). In this work, we will be seeking for a way to solve the following problem:

Definition 7.7 (Perfect Phylogeny Haplotype *Problem*)
Input: *A set of n genotype-representing row vectors, each of length m.*
Question: *Is it possible to resolve the genotype vectors into a set of haplotype vectors such that the haplotypes have evolved according to a perfect phylogeny scheme?*

For example, given the genotype vectors $(\,2\,0\,1\,)$, $(\,0\,2\,1\,)$, and $(\,1\,1\,1\,)$, can the underlying haplotypes have evolved according to a perfect phylogeny? As we will see later in this chapter, this is not the case (see Footnote 9 on page 105). The algorithm from [EHK03] is introduced in very much detail in the rest of this section because it will provide a basis for the hardness proofs of Theorems 7.9 and 7.11. In order to present the algorithm, we will start with some necessary terminology and observations.

If we are given n genome vectors of length m (i.e., representing m site), we may write this input as an $n \times m$ matrix A (often referred to as *"SNP matrix"* in the literature). For

[8] According to the authors of [HaEs03], this is sufficient for most cases of polymorphic sites.

developing an algorithm to solve Perfect Phylogeny Haplotype on a given SNP matrix, the following observations are crucial: Given the row vectors induced by two columns c_1 and c_2 (i.e., two SNP sites) in A. Based on the these row vectors, we can make the following observations regarding c_1 and c_2:

1. A row vector that does not contain a 2 can unambiguously be resolved into two identical haplotype vectors (e.g., $(0\ 1)$ must be resolved into $(0\ 1)$ and $(0\ 1)$).
2. A row vector $(0\ 2)$ must be resolved into two haplotype vectors $(0\ 0)$ and $(0\ 1)$. The analogue holds true for the row vectors $(1\ 2)$, $(2\ 0)$, and $(2\ 1)$.
3. The row vector $(2\ 2)$ can be resolved in exactly four ways, either

$$(2\ 2) \longrightarrow \begin{pmatrix} 1\ 1 \\ 0\ 0 \end{pmatrix}, \quad (2\ 2) \longrightarrow \begin{pmatrix} 0\ 0 \\ 1\ 1 \end{pmatrix}, \quad (2\ 2) \longrightarrow \begin{pmatrix} 0\ 1 \\ 1\ 0 \end{pmatrix}, \quad \text{or} \quad (2\ 2) \longrightarrow \begin{pmatrix} 1\ 0 \\ 0\ 1 \end{pmatrix}.$$

   The first two ways are called *equal resolution*, the second two *unequal resolution* of c_1 and c_2.

Following [EHK03], instead of saying that a row vector "can be resolved into" certain vectors we shall say that it *induces* those vectors from now on.

Due to the assumption that the inferred haplotypes have evolved according to a perfect phylogeny, note that two columns cannot be resolved equally and unequally for two different genomes: If this were the case, the resulting haplotypes would induce the row vectors $(0\ 0)$, $(0\ 1)$, $(1\ 0)$, and $(1\ 1)$—meaning they would induce an EΣM (see Definition 5.8). In Theorem 5.9, we have already shown that it would then be impossible to construct a perfect phylogeny for the haplotype vectors, even if inversion of the labels 0 and 1 were allowed.[9] More generally, if the given row vectors imply in any way that the genotype vectors must be resolved into haplotypes that induce the row vectors $(0\ 0)$, $(0\ 1)$, $(1\ 0)$, and $(1\ 1)$, there can be no perfect phylogeny for the haplotypes underlying the given genotypes. For example, this would be the case if two columns in the genotype matrix induce the row vectors $(0\ 2)$ and $(1\ 2)$.

We will now introduce the actual algorithm for Perfect Phylogeny Haplotype, which is proven in [EHK03] to run in $O(nm^2)$ time for n genotype vectors considering m sites. Let the genotype vectors be given as an $n \times m$ input matrix A. First, it is checked whether there are two columns in A that induce an EΣM. This must not be the case, since then is no perfect phylogeny for the haplotypes. Following this check, each column where the first row not containing a 2 contains a 1 is inverted—this inversion operation is done so that we can assume a *directed* perfect phylogeny[10] for the evolving of the haplotypes [EHK03]. The main key to the algorithm in [EHK03] is the resolvance of genotypes into haplotypes by determining where to place individual haplotypes with respect to each other in the perfect phylogeny. For this placement, [EHK03] develops the following intuitive notation concerning the relationship of two columns c_1 and c_2 in A:

- Column c_1 is said to *strongly dominate* column c_2 if c_1 together with c_2 induces the row vectors $(0\ 0)$, $(1\ 0)$, and $(1\ 1)$. The term "strong domination of c_1 over c_2" reflects the fact that since we have a directed perfect phylogeny, the mutation represented in c_1 took place in the evolutionary tree earlier than in c_2. This relationship between c_1 and c_2 is designated by writing $c_1 \succ c_2$.

[9] This answers the question posed in the example on page 104, because the presented genotype vectors induce an EΣM in the first two SNP sites.

[10] Recall the corresponding definitions in Section 5.2.

- Two columns c_1 and c_2 are called *siblings* if they induce the row vectors $(\,0\;0\,)$, $(\,1\;0\,)$ and $(\,0\;1\,)$. A sibling relationship between two columns denotes the fact that the mutation in site c_1 and the one in site c_2 took place independently, i.e. in different branches of the evolutionary tree. A sibling relationship between c_1 and c_2 is written as $c_1 \sim c_2$.

- Column c_1 is said to *weakly dominate* column c_2 if c_1 induces the row vectors $(\,0\;0\,)$ and $(\,1\;0\,)$ with c_2. For a weakly dominating c_1, we cannot directly determine whether the mutation in c_1 took place before the one in c_2 or in a different branch. However, we *do* know that this mutation can not have taken place after the one in c_2 in the same evolutionary branch. A weak domination of c_2 by c_1 is designated by writing $c_1 \succeq c_2$.

Using these relationships, the algorithm proceeds on A as follows, either creating a matrix B containing the haplotypes that explain the genotype matrix A or determining that A cannot be resolved into haplotypes that follow a perfect phylogeny:

1. Delete any columns in A that have a 2 in exactly the same rows and moreover induce $(\,1\;1\,)$. Call the matrix thus obtained A'. This may be done because the $(\,2\;2\,)$ row vectors in these columns will have to be resolved equally in order to be able produce a perfect phylogeny from the haplotypes.[11]

2. Choose a pivot column $\tilde{c}$ in A' so that for each column $c \neq \tilde{c}$ in A, either $\tilde{c} \succ c$, $\tilde{c} \sim c$, or $\tilde{c} \succeq c$ holds (such a column can always be found [EHK03]).[12]

3. Resolve every column in A' that is strongly dominated by $\tilde{c}$ equally with $\tilde{c}$ and every column in A' that is a sibling with $\tilde{c}$ unequally with $\tilde{c}$.

   For resolving the weakly dominated columns, the following graph $G = (V, E)$ with labeled edges is constructed:

   - The vertex set V is the set of sites.
   - There is an edge labeled 1 between the vertex representing $\tilde{c}$ and every vertex that represents a site that is weakly dominated by $\tilde{c}$.
   - If there are two sites c_1 and c_2 different from $\tilde{c}$ such that these three columns induce the row vector $(\,2\;2\;2\,)$ and c_1 and c_2 induce $(\,1\;1\,)$, the vertices in V representing c_1 and c_2 are joined by an edge labeled 0.
   - If there are two sites c_1 and c_2 different from $\tilde{c}$ such that these three columns induce the row vector $(\,2\;2\;2\,)$ and c_1 and c_2 are siblings, the vertices in V representing c_1 and c_2 are joined by an edge labeled 1.

   The columns are then resolved equally if the two respective vertices in G are connected by an edge labeled 0 and unequally if they are joined by an edge labeled 1. Conflicts may only arise if G contains a cycle that has an odd number of edges labeled 1, in which case the algorithm terminates with failure. Components in G from which there is no path to the vertex representing $\tilde{c}$ may be resolved arbitrarily with respect to $\tilde{c}$. If no columns can be resolved with respect to $\tilde{c}$ in this step, call the resulting matrix A'' and proceed to the next step.

[11] Note how an unequal resolution leads to the appearance of a Σ-matrix in the resulting haplotype matrix.

[12] As mentioned above, the key to this algorithm is to determine how different haplotypes are related in a perfect phylogeny due to comparison of different sites. The next step will now resolve individual columns with respect to the pivot column.

4. If A'' still contains unresolved haplotypes, the algorithm is recursively applied, starting from Step 1. Otherwise, A'' is put out as the resulting haplotype matrix B.

As mentioned above, the correctness of this algorithm as well as its running time is proven in [EHK03]. We shall now illustrate the algorithm by an example, using the genotype matrix

$$A := \begin{pmatrix} 2&0&2&0&0&1&0 \\ 2&2&1&2&0&1&2 \\ 2&0&1&2&2&2&0 \\ 0&2&1&0&2&2&2 \\ 1&0&1&2&0&1&0 \\ 0&0&1&0&1&2&0 \end{pmatrix}$$

as an input. The columns of A are referred to as $c_1, \ldots, c_7$. Since c_3 and c_6 contain a 1 in their first row not containing a 2, the first step is to invert c_3 and c_6, yielding

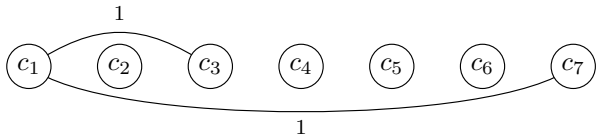
$$A' := \begin{pmatrix} 2&0&2&0&0&0&0 \\ 2&2&0&2&0&0&2 \\ 2&0&0&2&2&2&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix}.$$

No columns need to be deleted in Step 1. Now, for Step 2, c_1 is chosen as the pivot column. Using the terminology given on page 105, we have $c_1 \sim c_2$, $c_1 \succeq c_3$, $c_1 \succ c_4$, $c_1 \sim c_5$, $c_1 \sim c_6$, and $c_1 \succeq c_7$.[13] Step 3 of the algorithm given above can immediately resolve the pivot c_1 with c_2 (unequally), with c_4 (equally), with c_5 (unequally) and with c_6 (unequally), obtaining

$$\begin{pmatrix} 2&0&2&0&0&0&0 \\ 0&1&0&2&0&0&2 \\ 1&0&0&2&0&0&2 \\ 2&0&0&2&2&2&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix}, \text{ then } \begin{pmatrix} 2&0&2&0&0&0&0 \\ 0&1&0&0&0&0&2 \\ 1&0&0&1&0&0&2 \\ 0&0&0&0&2&2&0 \\ 1&0&0&1&2&2&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix},$$

$$\text{then } \begin{pmatrix} 2&0&2&0&0&0&0 \\ 0&1&0&0&0&0&2 \\ 1&0&0&1&0&0&2 \\ 0&0&0&0&1&2&0 \\ 1&0&0&1&0&2&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix}, \text{ and finally } \begin{pmatrix} 2&0&2&0&0&0&0 \\ 0&1&0&0&0&0&2 \\ 1&0&0&1&0&0&2 \\ 0&0&0&0&1&1&0 \\ 1&0&0&1&0&0&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix}.$$

The arrows indicate the resolvance of a genotype into two haplotypes.[14] Moving on to the graph construction in Step 3 of the algorithm, the following graph is constructed:



This graph is bipartite, so we can resolve c_1 unequally with c_3 and then unequally with c_7, obtaining

$$\begin{pmatrix} 0&0&1&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&2 \\ 1&0&0&1&0&0&2 \\ 0&0&0&0&1&1&0 \\ 1&0&0&1&0&0&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix}, \text{ and then } \begin{pmatrix} 0&0&1&0&0&0&0 \\ 1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1 \\ 1&0&0&1&0&0&0 \\ 0&0&0&0&1&1&0 \\ 1&0&0&1&0&0&0 \\ 0&2&0&0&2&2&2 \\ 1&0&0&2&0&0&0 \\ 0&0&0&0&1&2&0 \end{pmatrix}$$

[13] E.g., c_1 and c_7 induce the row vectors $(1\,0)$ (first row), $(0\,0)$ (seventh row), and $(0\,1)$ (fourth row).

[14] Note that a genotype is only explicitly splitted into two haplotypes if the respective columns used for the resolvance induce the row vector $(2\,2)$.

After the first round of the algorithm, the resulting matrix still contains rows with more than one 2. Hence, we have to apply the algorithm recursively to the last result and choose c_2 as the pivot column. We then have $c_2 \sim c_4$, $c_2 \sim c_5$, $c_2 \sim c_6$, and $c_2 \succ c_7$. Using these relationships, we can fully resolve our genotype matrix into

$$\begin{pmatrix} 0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,2\,2 \\ 0\,1\,0\,0\,0\,2\,2 \\ 1\,0\,0\,2\,0\,0\,0 \\ 0\,0\,0\,0\,1\,2\,0 \end{pmatrix}, \text{ then } \begin{pmatrix} 0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,2 \\ 0\,1\,0\,0\,0\,0\,2 \\ 1\,0\,0\,2\,0\,0\,0 \\ 0\,0\,0\,0\,1\,2\,0 \end{pmatrix}, \text{ and finally } \begin{pmatrix} 0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,2\,0\,0\,0 \\ 0\,0\,0\,0\,1\,2\,0 \end{pmatrix}$$

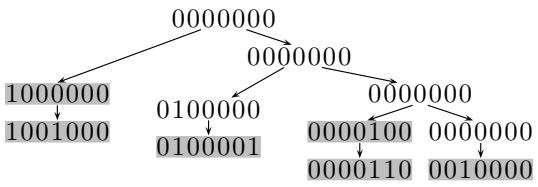
which implies the final haplotype matrix

$$\begin{pmatrix} 0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,0\,0\,0\,0 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \end{pmatrix} \text{ or, with duplicate haplotypes removed, } B := \begin{pmatrix} 0\,0\,1\,0\,0\,0\,0 \\ 1\,0\,0\,0\,0\,0\,0 \\ 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,0\,0 \\ 0\,0\,0\,0\,1\,1\,0 \end{pmatrix}$$

Observe that the haplotypes in B explain all genotypes in A':

$$\begin{aligned} 2\,0\,2\,0\,0\,0\,0 &= 0\,0\,1\,0\,0\,0\,0 + 1\,0\,0\,0\,0\,0\,0 \\ 2\,2\,0\,2\,0\,0\,2 &= 0\,1\,0\,0\,0\,0\,1 + 1\,0\,0\,1\,0\,0\,0 \\ 2\,0\,0\,2\,2\,2\,0 &= 0\,0\,0\,0\,1\,1\,0 + 1\,0\,0\,1\,0\,0\,0 \\ 0\,2\,0\,0\,2\,2\,2 &= 0\,0\,0\,0\,1\,1\,0 + 0\,1\,0\,0\,0\,0\,1 \\ 1\,0\,0\,2\,0\,0\,0 &= 1\,0\,0\,0\,0\,0\,0 + 1\,0\,0\,1\,0\,0\,0 \\ 0\,0\,0\,0\,1\,2\,0 &= 0\,0\,0\,0\,1\,0\,0 + 0\,0\,0\,0\,1\,1\,0 \end{aligned}$$

Furthermore, the explaining haplotypes have evolved according to a directed perfect phylogeny:



As we have seen, the algorithm given in [EHK03] provides a way to construct the haplotypes from given genotypes in polynomial time. However, real biological data will often not fit the model of a directed perfect phylogeny. If the algorithm determines that there exists no valid resolution of the given haplotypes, we might therefore ask what minimum number of sites or genotypes would have to be removed in order to fit the model of a perfect phylogeny. If the input matrix is just binary, i.e., it contains no entry equal to 2, we know from Chapter 5 that the haplotypes will only form a perfect phylogeny if they do not induce an EΣM (see Definition 5.8). If they do induce an EΣM, we have given some efficient fixed-parameter algorithms for removing a minimum number of species (genotypes) or characters (sites) from the input matrix so that we are able to infer a perfect phylogeny in Chapter 5. We have already seen in this chapter that if the input matrix contains entries equal to 2, the

problem of being able to construct a perfect phylogeny can—besides the avoidance of an induced EΣM—be related to a graph not containing a cycle of odd length[15] (where an edge may have a length of either zero or one). We can directly relate this problem to Graph Bipartization by simply replacing each edge of length zero with a path of length two.[16]

We will show in the following two subsections that the problems Minimum Genotype Removal and Minimum Site Removal are—from a parameterized point of view—at least as hard as Edge Bipartization and Vertex Bipartization, respectively. For Minimum Site Removal, we will even show that it is parameter-equivalent to Vertex Bipartization.

7.3.1 Minimum Genotype Removal

Note that this subsection makes extensive use of the terminology introduced in Section 7.3. The algorithm presented there to solve the Perfect Phylogeny Haplotype problem (see Definition 7.7) was not always able to find a haplotype-resolution for the given genotypes because a graph constructed from the input data proved not to be bipartite. We therefore seek a minimal number of genotypes to be removed from the dataset in order for the haplotypes to be constructible.

Definition 7.8 (Minimum Genotype Removal *Problem*)
Input: *A ternary matrix A of dimension $n \times m$ and an integer k.*
Question: *Is it possible to delete at most k rows in A so that the genotypes represented in the resulting matrix A' can be resolved into haplotypes that have evolved according to a perfect phylogeny?*

In [EHK03], the authors give a proof for the MAX-SNP-hardness of the Minimum Genotype Removal problem on ternary matrices, meaning there is no PTAS for this problem unless $P = NP$ (see the footnote on page 71 for the definition of PTAS). Moreover, a proof is given that an α-approximation algorithm for the Minimum Genotype Removal problem implies the α-approximability of the Edge Bipartization problem, for which no such approximation is known (see Chapter 6). With some slight modifications, the approximation preserving reduction from [EHK03] can be turned into a parameter-preserving reduction.

Theorem 7.9 Edge Bipartization *is parameter-preservingly reducible to* Minimum Genotype Removal.

Proof The reduction relies on the following idea: The presented algorithm for Perfect Phylogeny Haplotype constructs a graph from the given genotypes, where the vertices of the graph correspond to sites and the edges to the relationship of the sites determined by the individual genotypes. The graph was bipartite if and only if these genotypes could be resolved into haplotypes that constitute a perfect phylogeny. Given an instance (G, k) of Edge Bipartization, we will now simply build a matrix A consisting of genotypes that will force the algorithm to construct a graph that is isomorphic[17] to G. Furthermore, each

[15] These lengths are not to be confused with the edge-weights used in the opt-Edge Bipartization reduction rules.

[16] Note that this does not significantly increase the running time of the branch&bound algorithms from the previous chapter due to Reduction Rule 2 presented on page 80.

[17] Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are called *isomorphic* if there is a bijection $\Phi : V_1 \to V_2$ such that $\{v_a, v_b\} \in E_1 \Leftrightarrow \{\Phi(v_a), \Phi(v_b)\} \in E_2$.

genotype in A that is suited for deletion will correspond to exactly one edge in G. Then, removing a genotype from A corresponds directly to removing an edge from G.

For a given instance of Edge Bipartization, let $G = (V, E)$ be its graph where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$ (note that $n = |V|$ and $m = |E|$). An $m(n+1) \times (n+1)$ matrix $A = (a_{i,j})$ is now constructed, starting with a matrix of zeros and then applying the following algorithm:

Algorithm: Edge Bipartization to Minimum Genotype Removal
Input: A graph $G = (V, E)$ and a parameter k
Output: A parameter-equivalent instance A of Minimum Genotype Removal

01 **for** $i \leftarrow 1 \ldots m(n+1)$ **do**
02 $\quad a_{i,n+1} \leftarrow 2$
03 **for** $i \leftarrow 1 \ldots n$ **do**
04 $\quad$ **for** $j \leftarrow 1 \ldots m-1$ **do**
05 $\quad\quad a_{im-j,i} \leftarrow 2$
06 **for** each edge $e_i = \{v_a, v_b\}$ **do**
07 $\quad a_{i+mn,a} \leftarrow 2$
08 $\quad a_{i+mn,b} \leftarrow 2$

For illustration purposes, Figure 7.2 shows a graph and the corresponding matrix A generated by this algorithm, which works as follows: Lines *01* and *02* of the algorithm write a 2 into every row of the last column of A. By lines *03* to *05*, we ensure that each pair of the first n columns induces the row vectors $(1\ 0)$ and $(0\ 1)$. Therefore, each of the first n columns are pairwise siblings and will hence have to be resolved unequally according to the Perfect Phylogeny Haplotype-algorithm. Lines *06-08* encode the actual graph into A. During the following proof, we will say that an edge in G corresponds to a row in the matrix A if two 2-entries were inserted in that row due to lines *06-08* of the algorithm.

As was mentioned above, the first n columns are pairwise siblings. Column $(n+1)$ contains just 2's and weakly dominates all other columns, it will therefore be chosen as the pivot column c for the Perfect Phylogeny Haplotype algorithm. The algorithm will then proceed as follows: Since all columns are weakly dominated by c, the graph G_{algo} constructed for resolving the weak domination relationship between c and the first n columns of A will contain a vertex for each column. This includes a vertex for c, however, this vertex may be omitted since it will not be connected to any other component in G_{algo}.[18] So, in total, we have as many vertices in G as in the original graph G from which A was constructed. Two vertices u and v are connected in G_{algo} if and only if they were connected in G, and the corresponding columns have to be resolved unequally. Note that since for each edge in G, we have a row in A, the constructed graph G_{algo} is isomorphic to G. Then, as is proven in [EHK03], we can only resolve the graph if it does not contain any odd cycles, i.e., if it is bipartite.

Now assume that there exists a fixed-parameter algorithm for the Minimum Genotype Removal problem. For a given instance (G, k) of Edge Bipartization we can construct the matrix A using the algorithm above in polynomial time, more precisely we need

$$O(|V| + |V| \cdot |E| + |E|)$$

[18] Note that only strong domination or a sibling relationship will result in a connection.

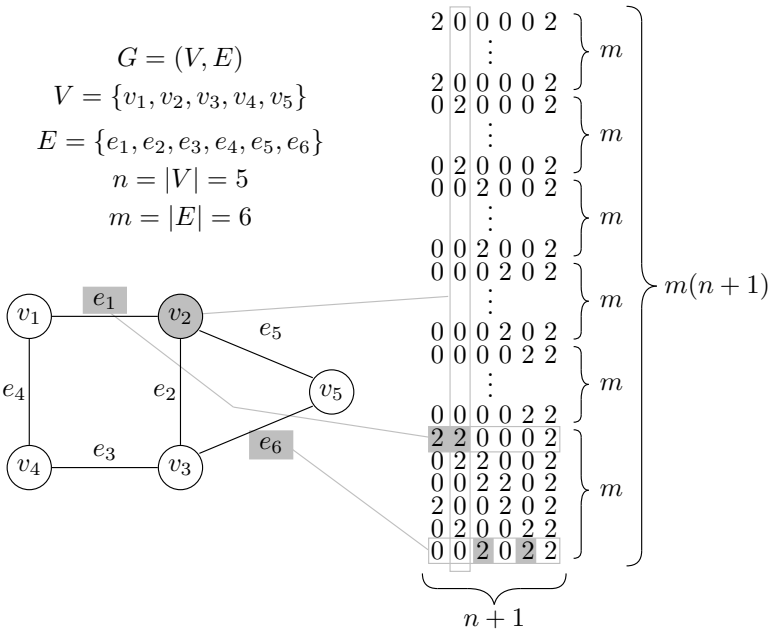


Figure 7.2: Minimum Genotype Removal matrix A (right) to solve the Edge Bipartization problem on the given graph G. Note how the correspondence between the elements of G and the resulting genotype matrix have been emphasized by grey underlays: Edge e_6 corresponds to the last row of A, since e_6 connects v_3 and v_5, a 2 has been written into the 3rd and 5th column of that row. Analogously, e_1 connects v_1 and v_2. Each vertex of G is represented by a column in A, emphasized in this figure by marking the second column and showing its correspondence to v_2. Note the interesting observation that we do not need entries equal to 1 for the reduction.

time with respect to the input graph size. If there exists, as a solution to the Edge Bipartization problem, a set $I \subseteq \{1, \ldots, m\}$ with $|I| \leq k$ such that removing the edges $\{e_i \mid i \in I\}$ will bipartize G, then deletion of the corresponding rows (genotypes) in A will lead to a matrix A' that can be resolved to fit the phylogenetic haplotype model. Conversely, if the deletion of at most k rows in A yields a resolvable matrix then there are two cases to be considered for each deleted row:

1. The row does not correspond to an edge. Since that row only contains at most two 2's, one in the pivot column and an additional one in some other row, deleting it will not touch the construction of G_{algo}, and can thus be considered obsolete. Note that for each column, there are at least m rows containing a 2 in that column and in the last one. Thus, it is impossible to force two columns to be resolved equally by deleting rows, since $k < m - 2$ in order for the original Edge Bipartization problem not to be trivial.

2. The row corresponds to an edge e. Then G_{algo} will be isomorphic to $G \setminus \{e\}$.

Thus, if deletion of a certain set of rows yields a matrix A' from A that fits the phylogenetic model, then deletion of the corresponding edges will bipartize G. Note that the parameter k is preserved by the reduction. □

Note that in the proof, we constructed the matrix A such that every genotype in A represents at most one edge in G_{algo}. However, this must not be the case vice-versa,[19] rendering the existence of a parameter-preserving reduction from Minimum Genotype Removal to Edge Bipartization an open problem (in close analogy to the existence of such a reduction for Minimum SNP Removal as shown in the previous section).

Instead of removing genotypes from the input matrix to Perfect Phylogeny Haplotype, one might rather be interested in removing a minimum number of sites. This is considered in the next subsection.

7.3.2 Minimum Site Removal

As a motivation for this subsection, we employ—in principle—the same reasons as in Section 5.5. Again, only a few sites in the genotypes may be responsible for inhibiting the construction of a phylogeny for the haplotypes. This directly leads to the Minimum Site Removal problem.

Definition 7.10 (Minimum Site Removal *Problem*)
Input: *A ternary genotype matrix M of size $n \times m$ and an integer k.*
Question: *Is it possible to delete at most k columns in M so that the resulting matrix can be resolved into haplotypes that have evolved according to a perfect phylogeny?*

We have shown in the last subsection that the analogue problem, i.e., removing genotypes from a given Perfect Phylogeny Haplotype input matrix, is at least as hard as Edge Bipartization. In the following Theorem 7.11, we will show that a similar result can be obtained for Minimum Site Removal, which we show to be parameter-equivalent to Vertex Bipartization.

Theorem 7.11 Vertex Bipartization *is parameter-equivalent to* Minimum Site Removal.

Proof In the introduction of the Perfect Phylogeny Haplotype-algorithm it was shown that the problem of resolving the genotypes of an input matrix A into haplotypes depends on the bipartization of a graph G_{algo} that is constructed from the input data. This graph was constructed so that every site in the genotype matrix was represented by a vertex in G_{algo}. Hence, deleting a vertex in G_{algo} directly corresponds to the removal of a site in A; thus, if G_{algo} can be bipartized by removing at most k vertices from it, removing the corresponding sites in A will allow the genotypes in the resulting matrix to be resolved into haplotypes by the Perfect Phylogeny Haplotype-algorithm.

We now give a reduction from Vertex Bipartization to Minimum Site Removal, analogously to the proof of Theorem 7.9.

The reduction is based on the following idea: Each of the first n columns in the constructed matrix A will represent a vertex in the given graph $G = (V, E)$ (with $|V| = n$ and $|E| = m$) that we wish to bipartize by vertex deletion. By the first n genotypes in A we will ensure that the first n columns have to be resolved unequally with respect to each other, because

[19] E.g., observe in Step 3 of the Perfect Phylogeny Haplotype algorithm that if a single genotype is responsible for making the pivot column weakly dominate a set of i columns, this genotype is responsible for i edges in the corresponding graph G_{algo}.

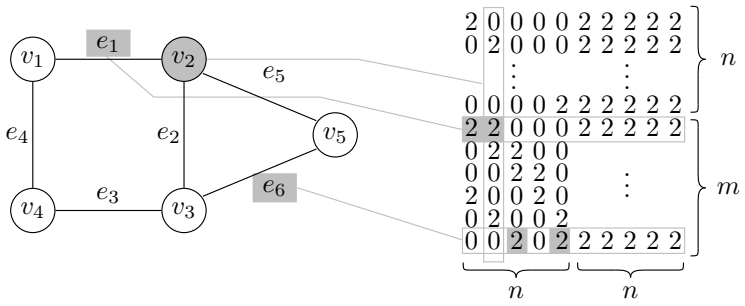


Figure 7.3: Minimum Site Removal matrix to solve the Vertex Bipartization problem on a given graph. To see the correspondence between columns in the matrix and vertices in the graph as well as edges in the graph and rows in the matrix, the following relationships have been marked: Edge e_6 corresponds to the last row of the matrix; since it connects v_3 and v_5, a 2 has been written into the 3rd and 5th column of that row. In an analogue way, e_1 (mth row from the bottom) connects v_1 (first column) and v_2 (second column). Each vertex of the graph is represented by a column in the matrix, emphasized in this figure by marking the second column and showing its correspondence to v_2. Note the interesting property of the reduction that it does not require any entries to be equal to 1.

they induce the row vectors $(1\,0)$ and $(0\,1)$, i.e., are pairwise siblings. With m further genotypes, we represent the m edges of G such that the deletion of a column c in A will also destroy any represented edges that have vertex corresponding to c as an endpoint.

Given a graph $G = (V, E)$ that we wish to bipartize by deleting at most k vertices, a ternary matrix A is constructed using an algorithm quite similar to the one given in the proof of Theorem 7.9. Let $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. To an $(m+n) \times 2n$ matrix A of zeroes, the following algorithm is applied in order to perform the reduction

Algorithm: Vertex Bipartization to Minimum Site Removal
Input: A graph $G = (V, E)$
Output: A parameter-equivalent instance A of Minimum Site Removal

01 **for** $i \leftarrow 1 \ldots n+m$ **do**
02 **for** $j \leftarrow n+1 \ldots 2n$ **do**
03 $a_{i,j} \leftarrow 2$
04 **for** $i \leftarrow 1 \ldots n$ **do**
05 $a_{i,i} \leftarrow 2$
06 **for** each edge $e_i = \{v_j, v_k\}$ **do**
07 $a_{i+n,j} \leftarrow 2$
08 $a_{i+n,k} \leftarrow 2$

In Figure 7.3, an example for the construction is given. The algorithm runs in

$$O(|V| \cdot (|V| + |E|) + |V| + |E|)$$

time and is thus polynomially bounded by the size of the input graph. It functions as follows: Lines *01* through *03* write a 2 into every row of the rightmost n columns of A. Lines *04* and *05* write 2's diagonally in the upper $n \times n$ submatrix of A, causing each of the first n

columns to be siblings to each other—they will therefore have to be resolved unequally by the PERFECT PHYLOGENY HAPLOTYPE-algorithm. Lines *06-08* encode the actual graph G into A.

In the matrix A, the first n columns are pairwise siblings and any of the n rightmost columns contains just 2s. Thus, any one of these columns (we will refer to these columns as "fill columns" from now on) weakly dominates all of the n leftmost columns. Now pick a fill column c. With any of the first n columns, c only induces the pairs $(2, 0)$ and $(2, 2)$ as row vectors (hence the weak domination). With any of the last columns, c only induces the pair $(2, 2)$, thus these columns may be resolved arbitrarily. Analogously to the proof of Theorem 7.11, the graph G_{algo} constructed by the PERFECT PHYLOGENY HAPLOTYPE-algorithm in order to resolve the weakly dominated sites will therefore be isomorphic to G and each one of the first n columns corresponds to a vertex in G.

The PERFECT PHYLOGENY HAPLOTYPE-algorithm will then proceed as follows: Since all of the first n columns are weakly dominated by c, the graph G_{algo} will contain a vertex for each of these columns. Note that there will be n vertices for the last n columns in A, however, these may be omitted since the resolvance between any two fill columns is arbitrary and a fill column weakly dominates any column in the left half of A. Using the same arguments as in the proof of Theorem 7.9, we can see that G_{algo} is isomorphic to G and the resulting haplotype model can only be resolved if G_{algo} (alternatively G) is bipartite.

Assume that there exists a fixed-parameter algorithm for the MINIMUM SITE REMOVAL problem. For a given instance (G, k) of VERTEX BIPARTIZATION we first construct A in polynomial time (see above). Furthermore, let there exists a solution to the given VERTEX BIPARTIZATION instance, i.e., there exists a set $I \subseteq \{1, \ldots, n\}$ with $|I| \leq k$ such that removing the vertices $\{v_i \mid i \in I\}$ will bipartize G. Then the deletion of the corresponding columns $\{c_i \mid i \in I\}$ in A will lead to a matrix A' that fits the phylogenetic haplotype model. Conversely, if the deletion of at most k columns in A leads to a resolvable matrix then there are two cases to be considered:

1. A column in the right half of A is deleted. Since this column does not contain any additional information as it is identical to any other fill column this does not affect the resolvability of the MINIMUM SITE REMOVAL problem on A. Note that there are n fill columns, and since $k < n - 2$ we cannot delete all of these columns.

2. A column in the left half of A is deleted (let this be the ith column where $1 \leq i \leq n$). This removes a 2 from the ith row and any 2 in the lower half of A that indicated an edge of v_i. Removing the 2 from the ith row does not affect the solvability of A, since that row directly induces the zero-root and does not have to be resolved. Removing a 2 in a column that corresponded to any edge $e = \{v_i, v_j\}$ for any j causes this row to be identical to the jth row in A. The deletion of a column removes a vertex from G_{algo}, and therefore, if G_{algo} can be bipartized by column deletion in A, G is bipartizable by deleting the corresponding vertices.

Concluding, if deletion of a certain set of columns yields a matrix A' from A that fits the phylogenetic model, then deletion of the corresponding vertices will bipartize G. Note that the parameter k is directly preserved by the reduction. □

We have seen that MINIMUM GENOTYPE REMOVAL is at least as hard to solve as EDGE BIPARTIZATION—deeper research will be needed to determine the relative hardness of MINIMUM GENOTYPE REMOVAL to MINIMUM SITE REMOVAL and their possible/impossible fixed-parameter tractability.

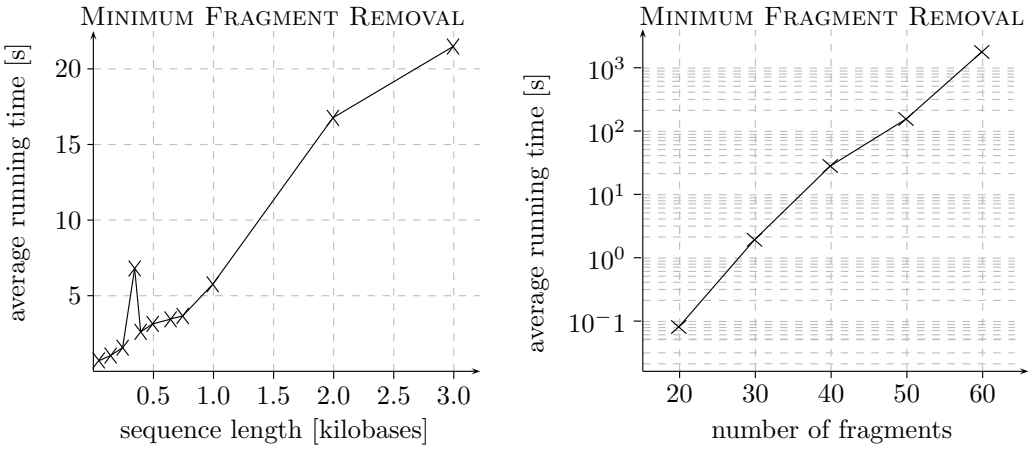


Figure 7.4: Performance of the Vertex Bipartization algorithm on various Minimum Fragment Removal instances. On the left, the average running time for Vertex Bipartization on a fragment conflict graph is shown relative to the source sequence's length ℓ (the number of fragments is kept at $\frac{\ell}{10}$). On the right, for a source sequence of length $\ell = 300$, the average running time of Vertex Bipartization on the fragment conflict graph is shown relative to the total number of fragments.

7.4 Testing Branch&Bound on SNP Data

In this chapter, we have seen that the problems Minimum Site Removal and Minimum Fragment Removal are parameter-equivalent to Vertex Bipartization. Using the reductions presented in this chapter, instances for both problems were transformed into Vertex Bipartization-instances, on which the bipartization-software package presented in Section 6.4 could be tested. In general, the software performed quite well, being able to solve instances for both problems that lead to graphs containing almost a hundred or (in the case of Minimum Fragment Removal) even hundreds of vertices

Methodology Due to the inavailability of "raw" read data from genotype-sequencing experiments, the Minimum Fragment Removal instances were generated artificially: Starting with two copies s_1 and s_2 of a random sequence of length ℓ over the alphabet $\{A, C, G, T\}$, sequence s_2 was altered in 5% of its bases in order to simulate the presence of SNPs. Then, n fragments of length 50 were read randomly from both sequences, introducing read errors at a rate of 0.5%. Two experiments were made:

- Varying ℓ between 100 and 3000 with $n = \frac{\ell}{10}$.
- Varying n between 20 and 60 with $\ell = 300$.

Each measurement was performed on 10 different random graphs for each set of parameters.

In order to test the performance of the Vertex Bipartization algorithm for instances of Minimum Site Removal, SNP-haplotype data from 50 samples of African Americans and 42 samples of unrelated Japanese and Chinese origin (both available at [Whit03]) were first converted into genotype data and then transformed into a graph using an algorithm that basically follows the procedure of the Perfect Phylogeny Haplotype-algorithm

introduced in the last section. "Holes" in the data from [Whit03], i.e., SNPs for which the state is specified as "unknown", were simply discarded during the graph-construction.

The machine on which the results of this section were obtained is the same as in the experiments from Section 6.4.

In some preliminary tests, all reduction rules except for Rule 6 (the rule involving vertex separators of order two) could often be applied to the graph. Since Rule 6 has by far the longest running time, it was switched off in the experiments. Performing measurements for the average running time with reduction rules turned off was not possible: Although sometimes, the same phenomenon as in the experiments of Section 6.4 occurred (i.e., the program was faster with reduction rules switched off), some instances turned out to be unsolvable in reasonable time with the reduction rules turned off. [20]

Results Figure 7.4 shows the results from the Minimum Fragment Removal experiments, Figure 7.5 shows the results from the Minimum Site Removal experiments. The average reduction rule usage was measured as follows:

Rule	Minimum Fragment Removal	Minimum Site Removal
1	36.4%	15.9%
2	30.5%	66.7%
3	3.5%	0.2%
4	4.9%	0.3%
5	2.0%	2.1%
6	(off)	(off)
7	10.72%	9.3%
8	10.5%	4.7%
9	1.5%	0.7%

Note that this usage does not reflect the relative gain in running time. Although Reduction Rules 3 and 4 are not used very often, they should, as was discussed Subsection 6.3.2, provide a majority of the gain in running time.

Discussion For Minimum Fragment Removal, problem instances based on sequences of a few thousand base pairs in length can be solved in acceptable time on average—if the relative number of fragments (compared to the sequence's total length) is not too high (e.g., around 5 as in the experiments shown left in Figure 7.4): While the average running time increases roughly linearly with a longer sequence, it grows exponentially when the ration $\frac{n}{\ell}$ is increased.

It should be noted that the term "on average" must be used with care here: As the peak in Figure 7.4 already indicates, the measured times had a very high variance, with some larger instance being solvable in under a minute, others in many hours for the same parameters of sequence and fragment length. However, note that in the average case, even instances with a few hundred fragments (=vertices in the corresponding conflict graph) can be solved efficiently. The reason why—compared to random graphs—the reduction rules are quite effective for Minimum Fragment Removal-conflict graphs, lies in the structure of these graphs: Note that each fragment covers only a rather small portion of the source sequence.

[20] E.g., one instance of Minimum Fragment Removal that could be solved in about an hour with reduction rules could not be solved within a day when the reduction rules were switched off.

sample name	population	# vertices	# edges	size of optimal solution [vertices]	running time [s]
10a	Japanese	57	117	3	11.3
11a	Japanese	51	212	5	48.7
12b	Japanese	48	91	1	0.8
13a	Japanese	78	210	6	2 498.8
14a	Japanese	72	107	4	7.1
15a	Japanese	45	55	1	0.07
16a	Japanese	14	10	0	0.03
17a	Japanese	81	322	Not solvable in < 10 hours	
18a	Japanese	73	296	Not solvable in < 10 hours	
19a	Japanese	101	172	3	6.2
20a	Japanese	241	640	Not solvable in < 10 hours	
21a	Japanese	33	102	9	2.1
22a	Japanese	76	391	9	476.5
10a	Afr.-American	45	143	5	6.2
11a	Afr.-American	45	191	9	359.3
12b	Afr.-American	19	14	1	0.02
13a	Afr.-American	57	235	10	58.3
14a	Afr.-American	63	397	Not solvable in < 10 hours	
15a	Afr.-American	47	139	6	224.2
16a	Afr.-American	12	11	0	0.01

Figure 7.5: Performance of the Vertex Bipartization algorithm on various Minimum Site Removal instances. The Minimum Site Removal instances were generated from data freely available in [Whit03]. If the bipartization on the test machine took longer than 10 hours, it was canceled.

Fragments that cover different parts of the source sequence are never connected by an edge in the conflict graph—this seems to significantly increase the likeliness of separators. Unless—by chance—many fragments cover the same area of the source sequence, the corresponding Minimum Fragment Removal problems should therefore be efficiently solvable in practice using the Vertex Bipartization-algorithm from this work.

Due to the rather small sample size, it is hard to make a general statement about the solvability of Vertex Bipartization instances derived from Minimum Site Removal problems. Nevertheless many of the analyzed instances were solvable in under one hour, even some in which the graph contained as much as 70 vertices. Referring to the results from Section 6.4, this is a tremendous improvement. E.g., Sample 13a of the African-American population led to a graph with 57 vertices and an average vertex degree of more than 4—from the results of Section 6.4, we would have expected all but a bipartization in under a minute, especially with the optimal solution containing as much as 10 vertices. Looking at the corresponding usage of reduction rules, it seems that exactly those larger graphs were bipartizable, to which Reduction Rules 3 and 4 could be applied. However, precisely characterizing the instances of Minimum Site Removal which—although they lead to rather large graphs—can be efficiently solved remains an open problem for future research. Overall, the results indicate that the developed Vertex Bipartization algorithms are efficient enough to solve even some larger instances of Minimum Site Removal and Minimum Fragment Removal.

Chapter 8

Conclusion

In this chapter, we give a brief recapitulation of this work, recalling the most important results. This summary is followed by some suggestions for related future research.

8.1 Summary of Results and Future Extensions

In Chapters 2 and 3, we provided a basic introduction, the motivation, and necessary terminology for this work—ranging from the prospects of SNPs in pharmacogenetics and phylogenetic analysis (Chapter 2) to a crash course in computational complexity, the analysis of algorithms, and fixed-parameter tractability (Chapter 3). Briefly recalling the results of the chapters following this bio-informatic introduction, we have shown that...

- ...the problem of Submatrix Occurrence is solvable in polynomial time (Chapter 4).
- ...we can prove that the problem of Row Deletion(B) is *NP*-complete for submatrices B which have special σ-decompositions (Chapter 4).
- ...the problem of Row Deletion(B) is fixed-parameter tractable and has a constant approximation factor for any forbidden submatrix B (Chapter 4).
- ...data correction in order to be able to construct a perfect phylogeny can be formulated as a Row Deletion(B) or Column Deletion(B) problem with B being the "extended Σ-matrix"—or EΣM, for short (Chapter 5).
- ...data correction in order to be able to construct a perfect phylogeny is *NP*-complete and fixed-parameter tractable (Chapter 5).
- ...the search tree of a trivial algorithm for eliminating all EΣMs in a binary matrix by row deletion has size $O(3^k)$ instead of the intuitively expected $O(4^k)$ (Chapter 5).
- ...the problem of Vertex Bipartization is at least as hard as Edge Bipartization, for there exists a parameter-preserving reduction from the latter to the former (Chapter 6).
- ...data reduction rules can be designed to efficiently solve Graph Bipartization-problems on graphs related to SNP-analysis (Chapters 6 and 7).

- ... the reassembly of genotype-fragments into the underlying haplotypes is closely related to Graph Bipartization (Chapter 7).
- ... data correction during the inference of haplotypes from genotypes is at least as hard as Edge Bipartization (Chapter 7).
- ... data correction on the SNP sites during the inference of haplotypes from genotypes is parameter-equivalent to Vertex Bipartization (Chapter 7).

During the preparation of this work, some interesting questions arose for which time and space did not allow a further investigation. However, they might serve as a starting point for future research:

- We have shown forbidden submatrices to be a generalization of the Minimum Species Removal and Minimum Character Removal problem. Since any forbidden substructure problem on bipartite graphs can be stated as a forbidden submatrix problem on the respective graphs adjacency matrix: What other problems can be related to the removal of forbidden submatrices? Are there problems that can be related to the removal of ℓ-ary matrices?
- Can problems related to k-Perfect Phylogeny be connected to forbidden submatrix problems?
- Is Conjecture 4.19 true? What are examples for matrices for which we do not yet know whether this conjecture is true and how can they be characterized in general?
- Can the computational complexity of other submatrix-removal strategies (such as the modification of individual entries or Row and Column Deletion) be determined using the framework developed in this work?
- The problem of finding a minimum "feedback-vertex set" (a set that contains at least one vertex from every cycle in the graph) is known to be fixed-parameter tractable [DoFe99]—can it be connected to Graph Bipartization? Furthermore, we expect the Minimum Feedback Vertex Set problem to have a lot of biological problems closely related to it (e.g., finding key reactions in metabolic networks)—it therefore deserve some further research of its own.
- The developed reduction rules are only efficient for the presented problems relating to SNPs. The importance of Graph Bipartization in many areas of research (as was mentioned in Chapter 6) could be a motivation to develop an efficient all-purpose software package to solve Graph Bipartization (especially Vertex Bipartization) problems.

As we have seen, bringing together biology and (theoretical) computer science is a fruitful undertaking for both areas, with biological problems giving impulses for the systematic analysis of previously unstudied computational problems (such as the Row Deletion problem in this work), and computer science providing tools and insights about chances and limits in various areas of biological research. This is the core idea behind the field of bioinformatics, from which we can hope to see more of this mutual nourishment in the near future as more and more people join in on connecting the life- and the computer sciences.

8.2 Acknowledgments

My advisors Rolf Niedermeier, Jochen Alber, Jens Gramm, and Jiong Guo introduced me to SNPs, spent many hours with me discussing my work, provided a lot of helpful advice, time and again proof-read my drafts, always had some interesting material at hand, and created a most friendly, memorable, and stimulating working atmosphere. I have benefited from their efforts far beyond this work. Additionally, I wish to thank Eva Anderl for her support, encouragement, countless waking and sleeping hours spent on reading my drafts, and for giving me the relieving insight that even a studied linguist can sometimes despair of the English relative clause and the commata it causes.

List of Figures

2.1 Chemical structure of DNA and its abbreviated notation 6
2.2 Mapping SNPs by comparison of two individuals' DNA sequence 7
2.3 Development of linkage disequilibria in SNP sites 10
2.4 SNP profiling in pharmacogenetics 13

3.1 A function $f(x)$ and its bounds in O-notation. 19
3.2 A graph G and an optimal vertex cover for G. 21
3.3 The search tree for finding a vertex cover of size k in a given graph 26

4.1 General scheme for the σ-decomposition of a matrix B 33
4.2 Finding all sets of rows that induce a forbidden submatrix. 36
4.3 Reduction from r-Hitting Set to Row Deletion(B) (Lemma 4.16) 42
4.4 Reduction from r'-Hitting Set to Row Deletion(B) (Theorem 4.13) . . . 43
4.5 Reduction from r'-Hitting Set to Row Deletion(B) (Theorem 4.14) . . . 45
4.6 Reduction from r'-Hitting Set to Row Deletion(B) (Theorem 4.11) . . . 48
4.7 Sorted decomposition of a binary matrix (Lemma 4.17) 50
4.8 The merge operation for the proof of Theorem 4.12 51

5.1 An example of a perfect phylogeny for a set of species 58
5.2 Parameterized reduction of 2-Hitting Set to Minimum Species Removal. 63
5.3 An instance of 2-Hitting Set and the corresponding Minimum Character Removal problem. 68

6.1 A graph G and its optimal bipartization by edge and vertex deletion. 70
6.2 Construction of a Vertex Bipartization instance from an Edge Bipartization instance. 73
6.3 Illustration of the recoloring heuristic for Edge Bipartization 77
6.4 Illustration of the heuristic for opt-Vertex Bipartization 79
6.5 Reduction Rules 3 and 4 81
6.6 Reduction Rules 5 and 6 84

6.7 Reduction Rule 7 86
6.8 Reduction Rules 8 and 9 88
6.9 UML-diagram for the implementation of the branch&bound algorithm 91
6.10 Running time for Graph Bipartization on a graph with 20 vertices 94
6.11 Search tree size for Graph Bipartization on a graph with 20 vertices . . . 94
6.12 Running time for Graph Bipartization relative to the number of vertices . 95
6.13 Running time for Graph Bipartization relative to the number of de-bipartizing elements (varying vertex degree) 95
6.14 Running time for Graph Bipartization relative to the number of de-bipartizing elements (varying number of vertices) 96

7.1 Minimum Fragment Removal on a set of fragments 101
7.2 Minimum Genotype Removal matrix to solve the Edge Bipartization problem on a given graph 111
7.3 Minimum Site Removal matrix to solve the Vertex Bipartization problem on a given graph 113
7.4 Performance of the Vertex Bipartization algorithm on various Minimum Fragment Removal instances 115
7.5 Performance of the Vertex Bipartization algorithm on various Minimum Site Removal instances 117

Bibliography

[Aaro03] S. Aaronson. *The Complexity Zoo*. http://www.cs.berkeley.edu/~aaronson/zoo.html, 2003 → 23

[Abec01] G. R. Abecasis. Extent and distribution of linkage disequilibrium in three genomic regions. *American Journal of Humane Genetics*, 68:191-197, 2001 → 9

[ADF95] K. A. Abrahamson, R. G. Downey, and M. R. Fellows. Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE analogs. *Annals of Pure and Applied Logic*, 73:235-276, 1995 → 28

[AgFe93] R. Agarwala and D. Fernández-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal on Computing*, 23(6):1216-1224, 1993 → 59

[ALMSS92] S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM*, 45:501-555, 1998 → 71

[Avis94] J. C. Avise. *Molecular Markers, Natural History and Evolution*, Chapman & Hall, 1994 → 9

[BBNE03] R. Brumfield, P. Beerli, D. Nickerson, and S. Edwards. The utility of single nucleotide polymorphisms in inferences of population history. Submitted for publication, 2003 → 9

[BFW92] H. L. Bodlaender, M. R. Fellows, and T. J. Warnow. Two strikes against perfect phylogeny. In *Proceedings of the 19th ICALP*, Springer-Verlag LNCS 623, 273-283, 1992 → 58, 59

[BJS02] J. M. Berg, J. L. Tymoczko, and L. Stryer. *Biochemistry*. W. H. Freeman, 2002 → 5

[BLM03] C. H. Bennett, M. Li, and B. Ma. Chain Letters & Evolutionary Histories. *Scientific American* 6:64-69, 2003 → 57

[Bodl88] H. L. Bodlaender. Dynamic programming algorithms on graphs with bounded treewidth. In *Proceedings of the 15th ICALP*, Springer-Verlag LNCS 317, 105-119, 1988 → 59

[Bodl97] H. L. Bodlaender. Treewidth: algorithmic techniques and results. *Mathematical Foundations of Computer Science '97*, 19-36, Springer, 1997 → 59

[Brem01] J. G. Breman. The ears of the hippopotamus: manifestations, determinants, and estimates of the malaria burden. *American Journal of Tropical Medical Hygiene*, 64:1-11, 2001 → 11

[Bune74] P. Buneman. A characterization of rigid circuit graphs. *Discrete Mathematics*, 9:205-212, 1974 → 59

[CaJu01] L. Cai and D. Juedes. Subexponential parameterized algorithms collapse the W-hierarchy. In *Proceedings of the 28th ICALP*, Springer-Verlag LNCS 2076, 273-284, 2001[1] → 28

[Carg99] M. Cargill *et al.* Characterization of single nucleotide polymorphisms in coding regions of human genes. *Nature Genetics*, 22:231-238, 1999 → 7, 8

[Chak01] A. Chakravarti. . . . to a future of genetic medicine. *Nature*, 409:822-823, 2001 → 1, 7

[CRS94] J. D. Cho, S. Raje, and M. Sarrafzadeh. Approximation algorithm on multiway maxcut partitioning. In *Proceedings of ESA '94*, Springer-Verlag LNCS 855:148-158, 1994 → 69

[CKJ01] J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280-301, 2001 → 26, 67

[CNR89] H. Cho, K. Nakajima, and C. S. Rim. Graph bipartization and via minimization. *SIAM Journal on Computing*, 2(1), 38-47, 1989 → 70

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms* (Second Edition). MIT Press, 2001 → 21, 78, 85, 92

[DEKM98] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis.* Cambridge University Press, 1998 → 55

[DeVa94] C. L. DeVane. Pharmacogenetics and drug metabolism of newer antidepressant agents. *Journal of Clinical Psychiatry*, 55:38-47, 1994 → 12

[DoFe99] R. G. Downey and M. R. Fellows. *Parameterized Complexity.* Monographs in Computer Science, Springer-Verlag, 1999 → 27, 28, 33, 120

[DRSHL01] M. J. Daly, J. D. Rioux, S. F. Schaffner, T. J. Hudson, and E. S. Lander. High-resolution haplotype structure in the human genome. *Nature Genetics*, 29(2):229-32, 2001 → 9

[DrSt92] A. Dress and M. Steel. Convex tree realizations of partition. *Applied Mathematics Letters* 5:3-6, 1992 → 59

[Dunn00] A. M. Dunning *et al.* The extent of human linkage disequilibrium in four populations with distinct demographic histories. *American Journal of Humane Genetics*, 67:1544-1554, 2000 → 9

[1] Some major flaws in this work have been pointed out in R.G. Downey, M.R. Fellows, R. Niedermeier, and P. Rossmanith (eds.). *Parameterized Complexity.* Dagstuhl-Report No. 316, 2001. A revised version of this paper is to appear under the title "On the existence of subexponential-time parameterized algorithms" in *Journal of Computer and System Sciences.*

[EHK03] E. Eskin, E. Halperin, and R. M. Karp. Efficient reconstruction of haplotype structure via perfect phylogeny. To appear in *Journal of Bioinformatics and Computational Biology (JBCB)*, 2003 → 2, 104, 105, 106, 107, 108, 109, 110

[EJM75] G. Estabrook, C. Johnson, and F. McMorris. An idealized concept of the true cladistic character. *Mathematical Bioscience*, 23:263-272, 1975 → 61

[Esta78] G. F. Estabrook. Some concepts for the estimation of evolutionary relationships in systematic botany. *Systematic Botany*, 3(2):146-158, 1978 → 57

[Fels03] J. Felsenstein. *Inferring Phylogenetics*, Sinauer Associates Incorporated, 2003 → 55

[FeNi01] H. Fernau and R. Niedermeier. An efficient exact algorithm for constraint bipartite vertex cover. *Journal of Algorithms* 38(2):374-410, 2001 → 33

[Fern01] D. Fernández-Baca. The perfect phylogeny problem. In *Steiner Trees in Industry*, Kluwer Academic Press, 2001 → 56, 57, 59

[Flan02] D. Flanagan. *Java in a Nutshell. A Desktop Quick Reference*, O'Reilly Publishers, 2002 → 89

[FoFu62] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*, Princeton University Press, 1962 → 85

[GaJo79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979 → 23, 24

[Gard98] M. J. Gardner *et al.* Chromosome 2 sequence of the human malaria parasite plasmodium falciparum. *Science*, 282:1126-1132, 1998 → 11

[Gavr74] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory Series B*, 16:47-56, 1974 → 59

[GCS00] I. C. Gray, D. A. Campbell, and N. K. Spurr. Single nucleotide polymorphisms as tools in human genetics. *Human Molecular Genetics*, 9(16):2403-2408, 2000 → 7

[GGL02] A. J. F. Griffiths, W. M. Gelbart, and R. C. Lewontin. *Modern Genetic Analysis: Integrating Genes and Genomes*, W. H. Freeman, 2002 → 5

[GHNR03] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for MAX2SAT with application to MAXCUT. *Discrete Applied Mathematics*, 130(2):139-155, 2003. → 71

[GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*, Addison-Wesley, 1994 → 64

[GMS00] A. J. F. Griffiths, J. H. Miller, and D. T. Suzuki. *An Introduction to Genetic Analysis*, W. H. Freeman, 2000 → 5

[GrKu90] D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms*, Birkhäuser Verlag, 1990 → 64

[Gusf91] D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21:19-28, 1991 → 58, 60

[Gusf02] D. Gusfield. Haplotyping as perfect phylogeny: Conceptual framework and efficient solutions [Extended Abstract]. *Proceedings of the 6th ACM International Conference on Computational Molecular Biology (RECOMB 2002)*, 166-175, 2002 → 100, 103, 104

[GVY96] N. Garg, V. V. Vazirani, and M. Yannakakis. Approximate max-flow min-(multi)cut theorems and their applications. *SIAM Journal on Computing*, 25:235-251, 1996 → 71

[HaCl97] D. L. Hartl and A. G. Clark. *Principles of Population Genetics*, Sinnauer, 1997 → 10

[Hadl75] F. O. Hadlock, Finding a maximum cut of a planar graph in polynomial time. *SIAM Journal on Computing* 4(3), 221-225, 1975 → 70

[HaEs03] E. Halperin and E. Eskin. Large scale recovery of haplotypes from genotype data using imperfect phylogeny. *Technical Report*, The Hebrew University of Jerusalem, School of Engineering and Computer Science, 2003 → 104

[Heil03] R. Heilig *et al.* The DNA sequence and comparative analysis of human chromosome 14. *Nature*, 421:601-607, 2003 → 1

[HGSC01] The International Human Genome Sequencing Consortium (approx. 100 authors). Initial sequencing and analysis of the human genome. *Nature*, 409:860-921 → 1

[Hold02] A. L. Holden. The SNP Consortium: Summary of a private consortium effort to develop an applied map of the human genome. *BioTechniques* 32:22-26, 2002 → 8, 11

[Huds90] R. Hudson. Gene genealogies and the coalescent process. *Oxford Survey of Evolutionary Biology*, 7:1-44, 1990 → 104

[Karp72] R. M. Karp. Reducibility among combinatorial problems. *R. E. Miller and J. W. Thatcher (eds.): Complexity of Computer Computations*, 85-103, 1972 → 70

[KaWa94] S. Kannan and T. Warnow. Inferring evolutionary history from DNA sequences. *SIAM Journal on Computing*, 23:713-737, 1994 → 59

[KaWa97] S. Kannan and T. Warnow. A fast algorithm for the computation and enumeration of perfect phylogenies. *SIAM Journal on Computing*, 26(6):1749-1763, 1997 → 59

[KFHW98] I. J. Kitching, P. L. Forey, C. J. Humphries and D. M. Williams. *Cladistics: The Theory and Practice of Parsimony Analysis*, Oxford University Press, 1998 → 55, 56

[Knut97] D. E. Knuth. *The Art of Computer Programming, Volumes I-III Boxed Set: Fundamental Algorithms, Seminumerical Algorithms, Sorting and Searching* (Third Edition). Addison-Wesley, 1997 → 21

[Knut03] D. E. Knuth. *The Art of Computer Programming, Pre-Fascicle 2A: A draft of Section 7.2.1.1: Generating all n-Tuples*, Zeroth printing (revision 6), available via http://www-cs-faculty.stanford.edu/~knuth/taocp.html, 2003 → 21

[KhRa02] S. Khot and V. Raman. Parameterized complexity of finding subgraphs with hereditary properties. *Theoretical Computer Science* 289(2):997-1008, 2002 → 71

[Krug99] L. Kruglyak. Prospects for whole-genome linkage disequilibrium mapping of common disease genes. *Nature Genetics*, 22:139-144, 1999 → 9

[KRW95] B. Klinz, R. Rudolf, and G. J. Woeginger. Permuting matrices to avoid forbidden submatrices. *Discrete Applied Mathematics*, 60:223-248, 1995 → 32

[Kull99] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science* 223:1-72, 1999 → 64

[LBILS01] G. Lancia, V. Bafna, S. Istrail, R. Lippert and R. Schwartz. SNPs problems, complexity, and algorithms. In *Proceedings of ESA 2001*, Springer-Verlag LNCS 2161, 182-193, 2001 → 2, 101

[LeYa80] J. M. Lewis and M. Yannakakis. The node-deletion problem for hereditary properties is *NP*-complete. *Journal of Computer and System Sciences*, 20:219-230, 1980 → 70

[Ligg97] S. B. Liggett. Polymorphisms of the beta 2-adrenergic receptor and asthma. *American Journal of Respiratory Critical Care Medicine*, 156:156-162, 1997 → 13

[LPC98] J. Lazarou, B. H. Pomeranz, and P. N. Corey. Incidence of adverse drug reactions in hospitalized patients. *Journal of the American Medical Society.* 279:1200-1205, 1998 → 12

[Mare97] D. Marez *et al.* Polymorphism of the cytochrome P450 CYP2D6 gene in a European population: characterization of 48 mutations and 53 alleles, their frequencies and evolution. *Pharmacogenetics*, 7:193-202, 1997 → 7

[Mart00] E. R. Martin *et al.* Analysis of association at single nucleotide polymorphisms in the ApoE region. *Genomics*, 63:7-12, 2000 → 12

[Maso99] E. Masood. Glaxo Wellcome is already using map data, *Nature*, 398:546, 1999 → 12

[Mate01] E. Mateu *et al.* Worldwide genetic analysis of the CFTR region. *American Journal of Humane Genetics*, 68:103-117, 2001 → 11

[Meac83] C. A. Meacham. Theoretical and computational considerations of the compatibility of qualitative taxonomic characters. *Nato ASI Series Vol. G1 on Numerical Taxonomy*, Springer, 1983 → 61

[MeEs85] C. A. Meacham and G. F. Estabrook. Compatibility methods in systematics. *Annual Review of Ecology and Systematics*, 16:431-446, 1985 → 57

[MeNä99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, 1999 → 89, 92

[Morr77] F. McMorris. On the compatibility of binary qualitative taxonomic characters. *Bulletin of Mathematical Biology*, 39:133-138, 1977 → 61

[Mu02] J. Mu *et al.* Chromosome-wide SNPs reveal an ancient origin for plasmodium falciparum.[2] *Nature*, 418:323-326, 2002 → 7, 9, 10, 11

[NCBI03] *National Center for Biotechnology Information (NCBI)*, information available via http://www.ncbi.nlm.nih.gov/, 2003 → 1

[Nied02] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Habilitationsschrift, Universität Tübingen, 2002 → 29

[Niel00] R. Nielsen. Estimation of population parameters and recombination rates using single nucleotide polymorphisms. *Genetics*, 154:931-942, 2000 → 9

[NiRo00] R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73:125-129, 2000 → 27, 33

[NiRo03$_a$] R. Niedermeier and P. Rossmanith. An efficient fixed-parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, to appear 2003 → 33, 63

[NiRo03$_b$] R. Niedermeier and P. Rossmanith. On efficient fixed-parameter algorithms for weighted vertex cover. *Journal of Algorithms*, 47:63-77, 2003 → 26, 67, 68

[NRO99] M. Nikaido, A. P. Rooney, and N. Okada. Phylogenetic relationships among certiodactyls based on insertions of short and long interpersed elements. Hyppopotamuses are the closest extant relatives of whales. *Proceedings of the Natinal Academy of Sciences (USA)*, 96:10261-10266, 1999 → 57

[OMG03] *The Object Management Group UML Ressource Page.* http://www.omg.org/uml/, 2003 → 90

[PaHo98] R. D. M. Page and E. C. Holmes. *Molecular Evolution: A Phylogenetic Approach*, Blackwell Science, 1998 → 56

[Papa94] C. H. Papadimitriou. *Computational Complexity.* Addsion-Wesley, 1994. → 16, 17, 71

[PaYa91] C. H. Papadimitriou and M. Yannakakis. Optimization, approximation and complexity classes. *Journal of Computer and System Sciences*, 43:425-440, 1991 → 71

[PBH01] N. Patil, A. J. Berno, D. A. Hinds *et al.* Blocks of limited haplotype diversity revealed by high-resolution scanning of human chromosome 21. *Science*, 294:1719-1723, 2001 → 103

[Page99] M. Page-Jones. *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, 1999 → 90

[PoTu95] S. Poljak and Z. Tusza. Maximum cuts and large bipartite subgraphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 20:181-244, 1995 → 71

[PrSl03] E. Prieto and C. Sloper. Either/or: Using vertex cover structure in designing FPT-algorithms - the case of k-internal spanning tree. In *Proceedings of WADS 2003*, 2003 → 71

[2] A correction of some details appears in *Nature* 419:487, 2002.

[PSS97] S. Parkkila, W. S. Sly, R. C. Schatzmann *et al.* The hemochromatosis founder mutation in HLA-H disrupts 2-microglobulin interaction and cell surface expression. *Journal of Biological Chemistry*, 22:14025-14028, 1997 → 7

[PSS02$_a$] I. Pe'er, R. Shamir, and R. Sharan. Incomplete directed perfect phylogeny. In *Proceedings of 11th CPM*, Springer-Verlag LNCS 1848, 143-153, 2000 → 58, 60

[PSS02$_b$] I. Pe'er, R. Shamir, and R. Sharan. On the generality of phylogenies from incomplete directed characters. In *Proceedings of 8th SWAT 2002*, Springer-Verlag LNCS 2368, 358-367, 2002 → 58

[RBIL02] R. Rizzi, V. Bafna, S. Istrail and G. Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. In *Proceedings of WABI 2002*, Springer-Verlag LNCS 2452, 29-43, 2002 → 99

[RBWL95] J. C. Roach, C. Boysen, K. Wang, and L. Hood. Pairwise end sequencing: a unified approach to genomic mapping and sequencing. *Genomics*, 26(2):345-353, 1995 → 100

[Reic01] D. E. Reich *et al.* Linkage disequilibrium in the human genome. *Nature*, 411:199-204, 2001 → 9, 10, 11

[RLHA98] S. M. Rich, M. C. Licht, R.R. Hudson, and F. J. Ayala. Malaria's Eve: evidence of a recent populational bottleneck throughout the world's populations of plasmodium falciparum. *Proceedings of the National Academy of Sciences (USA)*, 95:4425-4430, 1998 → 11

[RoRe52] F. W. Robertson and E. Reeve. Studies in quantitative inheritance. 1. The effects of selection on wing and thorax length in Drosophila melanogaster. *Journal of Genetics* 50:414-448, 1952 → 56

[Rose00] A. D. Roses. Pharmacogenetics and the practice of medicine. *Nature*, 405:857-865, 2000 → 9, 12, 13

[SCHHP82] F. Sanger, A. R. Coulson, G. F. Hong, D. F. Hill, G. B. Petersen. Nucleotide sequence of bacteriophage lambda DNA. *Journal of Molecular Biology* 162:729-773, 1982 → 100

[SeSt03] C. Semple and M. Steel. *Phylogenetics*, Oxford Lecture Series in Mathematics and Its Applications, Oxford University Press, 2003 → 55, 56

[Skie98] S. S. Skiena. *The Algorithm Design Manual.* Springer-Verlag, 1998 → 16, 17, 78, 89

[SMVS97] H. Schröder, A. E. May, I. Vrto, and O. Sýkora: Approximation algorithms for the vertex bipartization problem. In *Proceedings of the 24th SOFSEM*, Springer-Verlag LNCS 1338, 547-554, 1997 → 76, 77, 93

[SNP01] The International SNP Map Working Group (approx. 40 authors). A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. *Nature*, 409:928-933 → 2, 7, 8

[SSD01] M. Stephens, N. Smith, and P. Donnelly. A new statistical method for haplotype reconstruction from population data. *American Journal of Humane Genetics*, 68:978-989, 2001 → 104

[Stee92] M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9:91-116, 1992 → 57

[Ston01] M. Stoneking. From the evolutionary past... . *Nature*, 409:821-822, 2001 → 1, 2

[SuMi03] *The Sun Microsystems^{TM}Java Technology Source*, http://java.sun.com/, 2003 → 89, 93

[Tish96] D. A. Tishkoff *et al.* Global patterns of linkage disequilibrium at the CD4 locus and modern human origins. *Science*, 271:1380-1387, 1996 → 9, 11

[Tish00] D. A. Tishkoff *et al.* Short tandem-repeat polymorphism/Alu haplotype variation at the PLAT locus: Implications for modern human origins. *American Journal of Human Genetics*, 67:901-925, 2000 → 9, 11

[Titc76] E. C. Titchmarsh. *Theory of Functions*, Oxford University Press, 1976 → 64

[VoVo95] D. Voet and J. Voet. *Biochemistry.* John Wiley & Sons, 1995 → 5, 6, 100

[Wate95] M. S. Waterman. *Introduction to Computational Biology.* Chapman & Hall, 1995 → 100

[Watt77] G. A. Watterson. Is the most frequent allele the oldest?. *Theoretical Population Biology*, 11:141-160, 1977 → 10

[WeHu02] M. P. Weiner and T. J. Hudson. Introduction to SNPs: Discovery of markers for disease. *BioTechniques*, 32:4-13, 2002 → 1

[WeMy97] J. Weber and E. Myers. Human whole genome shotgun sequencing. *Genome Research*, 7:401-409, 1997 → 100

[Whit03] The Whitehead Institute - Center for Genome Research. *The Structure of Haplotype Blocks in the Human Genome.* http://www-genome.wi.mit.edu/mpg/hapmap/hapstruc.html#data, 2003 → 115, 116, 117

[WHO03] The World Health Organization. *Fact Sheet 94: Malaria.* available via http://www.who.int/inf-fs/en/fact094.html, 2003 → 11

[Yann78] M. Yannakakis. Node-and edge-deletion *NP*-complete problems. In *Proceedings of the 10th annual ACM Symposium on Theory of Computing*, 253-264, 1978 → 70

[Yann81] M. Yannakakis. Edge-deletion problems. *SIAM Journal on Computing*, 10(2):297-309, 1981 → 69, 70, 72