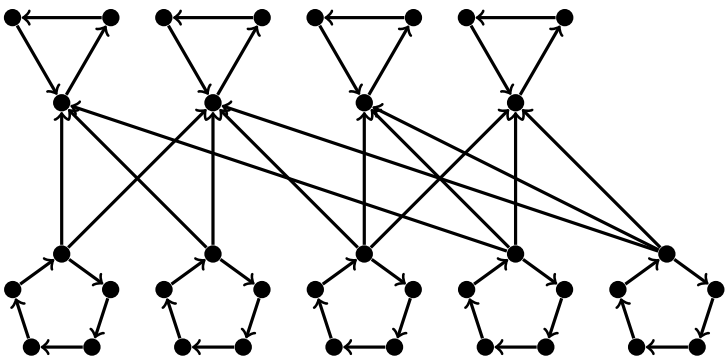
Bipartite Graphs and the Decomposition of Systems of Equations



Matthijs Bomhoff

Bipartite Graphs and the Decomposition of Systems of Equations

Matthijs Bomhoff

Dissertation committee

Chairman / Secretary	prof.dr.ir. A.J. Mouthaan	Univ. Twente, EWI
Supervisor	prof.dr. M.J. Uetz	Univ. Twente, EWI
Assistant Supervisors	dr. G.J. Still	Univ. Twente, EWI
	dr. W. Kern	Univ. Twente, EWI
Members	prof.dr.ir. H.J. Broersma	Univ. Twente, EWI
	prof.dr.ir. B.J. Geurts	Univ. Twente, EWI
	dr. H.L. Bodlaender	Univ. Utrecht
	dr. N. Bansal	Technische Univ. Eindhoven
	prof.dr. G. Schäfer	Centrum Wiskunde & Informatica / Vrije Univ. Amsterdam

The author gratefully acknowledges the support of the Innovation-Oriented Research Programme 'Integral Product Creation and Realization (IOP IPCR)' of the Netherlands Ministry of Economic Affairs.

Ph.D. thesis, University of Twente, Enschede, the Netherlands

ISBN 978-90-365-3476-5
DOI 10.3990/1.9789036534765

BIPARTITE GRAPHS AND THE DECOMPOSITION OF SYSTEMS OF EQUATIONS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 23 januari 2013 om 14:45 uur

door

Matthijs Jacobus Bomhoff
geboren op 19 februari 1982
te Emmen

Dit proefschrift is goedgekeurd door
prof. dr. M.J. Uetz (promotor)
dr. G.J. Still (assistent-promotor)
dr. W. Kern (assistent-promotor)

Voor mijn ouders

Voor Gerda

Abstract

Solving large systems of equations is a problem often encountered in engineering disciplines. However, as such systems grow, the effort required for finding a solution to them increases as well. In order to be able to cope with ever larger systems of equations, some form of decomposition is needed. By decomposing a large system into smaller subsystems, the total effort required for finding a solution may decrease. However, whether this is really the case of course depends on the additional effort required for obtaining the decomposition itself. In this thesis several aspects of the difficulty of obtaining such decompositions are explored.

The first problem discussed in this thesis is that of the decomposition of a system of non-linear equations. After describing the well-known conditions for consistency in systems of equations, the decomposition problem for under-specified systems is analyzed. This analysis, based on the existing notion of *free square blocks*, leads to several $W[1]$-hardness results. The most important of which is the proof that finding a decomposition into subsystems where the largest subsystem is as small as possible is $W[1]$-hard. This implies that even if the size of such a largest subsystem is bounded by a constant, the problem is still not solvable in polynomial time under the current working hypothesis that $W[1] \neq FPT$. As a by-product of these results two open problems regarding crown structures for the vertex cover problem are settled.

Having investigated the non-linear case, attention is then shifted to the case of systems of linear equations. Such systems are commonly represented as matrices upon which pivot operations are performed. In such operations preserving sparsity of the input matrix is often beneficial to limit both storage and processing requirements. The choice of pivot elements can strongly influence the level of sparsity that is preserved. Changing a zero value in a matrix to a non-zero value is called *fill-in*. An important role in preserving sparsity is played by bisimplicial edges in the bipartite graph that corresponds naturally to a matrix. Such bisimplicial edges correspond to pivots in the matrix that completely avoid fill-in. In this thesis a new $O(nm)$ algorithm for finding such bisimplicial edges for an $n \times n$ matrix with m non-zero values is described and analyzed on a common class

of random matrices. It is shown that the expected running-time of the algorithm on matrices corresponding to bipartite graphs from the $G_{n,n,p}$ model is $O\left(n^2\right)$, which is linear in the input size.

After analyzing single pivots that avoid turning zero elements into non-zero elements, the class of matrices that allow Gaussian elimination using only such pivots is discussed. Such matrices correspond to *prefect elimination bipartite graphs*. Several algorithms are known for the recognition of this class of graphs or matrices, but most of them are based on matrix multiplication implying sparse input matrices may still result in dense matrices along the way. In this thesis two new algorithms for the recognition of matrices allowing Gaussian elimination while completely preserving sparsity are described. One of them is an adaption of an existing algorithm with a $O(nm)$ time complexity in $\Theta\left(n^2\right)$ space. The other is a completely new algorithm designed for efficient handling of sparse instances in $O\left(m^2\right)$ time and $\Theta(m)$ space.

The final problem discussed in this thesis is a more fine-grained variation on the traditional Gaussian elimination where instead of using a single element to clear an entire column, a new pivot element is picked for each non-zero element in the matrix that needs to be cleared. It is shown that this approach to pivoting allows the complete preservation of sparsity on a larger class of graphs or matrices, called *perfect partial elimination bipartite graphs*. However, it is unfortunately also shown that recognition of such matrices is NP-hard, immediately implying that minimizing the amount of fill-in using such pivots can most probably not be done in polynomial time.

Samenvatting

Bipartiete grafen en de decompositie van stelsels van vergelijkingen

Het oplossen van grote stelsels van vergelijkingen is een veel voorkomend probleem in technische vakgebieden. Naarmate zulke stelsels groter worden, wordt het vinden van oplossingen meer en meer werk. Een vorm van decompositie is dan ook noodzakelijk om met steeds grotere stelsels om te kunnen gaan. Door grote stelsels op te splitsen in kleinere deelstelsels kan de totale hoeveelheid werk voor het vinden van oplossingen afnemen. Echter, of dit echt het geval is, is uiteraard afhankelijk van de extra moeite die het kost om de decompositie zelf te bepalen. In dit proefschrift worden enkele aspecten van het bepalen van zulke decomposities verkend.

Het eerste probleem dat behandeld wordt, is de decompositie van stelsels van niet-lineaire vergelijkingen. Nadat de bekende voorwaarden voor consistentie in stelsels van vergelijkingen beschreven zijn, wordt het decompositieprobleem voor ondergespecificeerde stelsels van vergelijkingen geanalyseerd. Deze analyse, gebaseerd op het concept van *free square blocks* leidt tot enkele $W[1]$-hardheidsbewijzen. De belangrijkste hiervan is het bewijs dat het vinden van een decompositie van een stelsel vergelijkingen waarbij het grootste deelstelsel zo klein mogelijk is, $W[1]$-hard is. Onder de aanname dat $W[1] \neq FPT$ impliceert dit dat dit probleem zelfs als de grootte van het grootste deelstelsel van bovenaf begrensd is door een constante, niet oplosbaar is in polynomiale tijd. Als bijeffect van deze resultaten zijn ook twee open vragen betreffende *crown structures* voor het *vertex cover* probleem opgelost.

Na het analyseren van het niet-lineaire geval volgt de bestudering van stelsels van lineaire vergelijkingen. Zulke stelsels worden vaak behandeld als matrices waarop pivot-operaties uitgevoerd worden. Bij deze operaties is het behouden van nullen in de invoermatrix over het algemeen gewenst om de eisen voor zowel opslag als verwerking te beperken. De keuze van pivot-elementen kan een grote invloed hebben op de mate waarin nullen behouden blijven. Het veranderen van een nul in een niet-nul-waarde in een matrix wordt *fill-in* genoemd. Een belangrijke rol in het behouden van

nullen is weggelegd voor *bisimplicial* zijden in de bipartiete graaf die op natuurlijke wijze correspondeert met de matrix. Zulke bisimplicial zijden corresponderen met pivots in de matrix die fill-in geheel voorkomen. In dit proefschrift wordt een nieuw $O(nm)$ algoritme voor het vinden van zulke bisimplicial zijden voor een $n \times n$ matrix met m niet-nul-elementen beschreven en geanalyseerd op een veel voorkomende klasse van random matrices. Deze analyse toont aan dat de te verwachten looptijd van het algoritme $O\left(n^2\right)$ is op matrices die corresponderen met bipartiete grafen uit het $G_{n,n,p}$-model, en daarmee lineair in de grootte van de invoer.

Volgend op de analyse van individuele pivots die fill-in voorkomen, wordt de klasse van matrices behandeld waarop Gauss-eliminatie mogelijk is door enkel dergelijke pivots te gebruiken. Deze matrices corresponderen met *perfect elimination bipartite graphs*. Enkele algoritmes voor het herkennen van zulke grafen of matrices waren al bekend, maar de meeste daarvan zijn gebaseerd op matrixvermenigvuldiging, wat betekent dat ijle invoermatrices nog steeds onderweg kunnen leiden tot een veel vollere matrix. In dit proefschrift worden twee nieuwe algoritmes beschreven voor het herkennen van matrices waarop Gauss-eliminatie mogelijk is waarbij alle fill-in voorkomen wordt. Het eerste is een aanpassing van een bestaand algoritme met een tijdscomplexiteit van $O(nm)$ in $\Theta\left(n^2\right)$ geheugen. Het tweede is een volledig nieuw algoritme specifiek ontworpen voor het efficiënt herkennen van ijle matrices in $O\left(m^2\right)$ tijd en $\Theta(m)$ geheugen.

Het laatste probleem dat besproken wordt, is een variant van Gauss-eliminatie waarbij in plaats van een enkel element voor het vegen van een kolom, er voor ieder niet-nul-element dat geveegd moet worden een nieuw pivot-element gekozen wordt. Deze aanpak voor pivoteren maakt het volledig behoud van nul-elementen mogelijk op een grotere klasse van grafen of matrices, genaamd *perfect partial elimination bipartite graphs*. Helaas wordt ook bewezen dat het herkennen van zulke matrices NP-hard is, wat direct impliceert dat het minimaliseren van fill-in bij gebruik van dergelijke pivot-elementen waarschijnlijk niet in polynomiale tijd mogelijk is.

Dankwoord

Allereerst wil ik graag mijn promotor Marc Uetz en mijn twee begeleiders Georg Still en Walter Kern bedanken. Ik ben zeer dankbaar voor de kans om in deeltijd te promoveren en het vertrouwen dat zij daarin hadden. De combinatie van goede inhoudelijke begeleiding, gezellige samenwerking en de vrijheid om mijn eigen weg te vinden, heb ik erg gewaardeerd. Met veel plezier denk ik terug aan de talloze keren dat ik bij Georg of Walter aanklopte om mijn zoveelste inval te bespreken voor een algoritme of een reductie. Jullie eindeloze geduld, enthousiasme en motivatie heeft mij meer geholpen dan jullie waarschijnlijk zelf beseffen.

Ook met de andere leden van de vakgroep DMMP heb ik prettig samengewerkt. Hoewel ik vaak afwezig was – parttime en ook nog eens vaak thuis aan het werk – was het altijd leuk om te horen waar anderen mee bezig waren en om wiskundige problemen met elkaar te delen. Een bijzondere vermelding voor Hajo Broersma is hier op zijn plek. Door zijn vakken heb ik grafentheorie leren kennen als fascinerend en elegant gebied binnen de wiskunde. Afstuderen bij hem is er niet van gekomen, maar ik ben blij en trots dat hij wel in mijn promotiecommissie plaats heeft willen nemen.

Het project waarbinnen mijn onderzoek plaatsvond, is een samenwerking tussen werktuigbouwkunde en wiskunde. Met mijn collega's binnen het project uit zowel Enschede als Delft heb ik vele interessante discussies gevoerd. Hiervan heb ik niet alleen inhoudelijk veel geleerd, maar ik werd hierdoor vaak ook gewezen op mijn eigen soms beperkte blik op problemen. De vele vrijdagmiddagen besteed aan allerhande onderwerpen vormden een ware verrijking.

In de meeste dankwoorden komt op een gegeven moment een overgang van collega's naar vrienden en familie. Aan beide zijden van deze overgang hoort voor mij Paul van der Vet. In 2004 heeft hij als een van mijn afstudeerbegeleiders bij de studie Informatica de dappere taak op zich genomen om mij als eigenwijze student bij te brengen wat het doen van wetenschappelijk onderzoek inhoudt. Voor het feit dat dit hem gelukt is op een manier die vervolgens leidde tot een goede vriendschappelijke band, heb ik – zeker achteraf bezien – niets dan bewondering. Gedurende de jaren hebben we veel interessante gesprekken gevoerd en ook na mijn afstuderen ben ik

je naast vriend toch ook altijd als mentor in de wetenschappelijke wereld blijven zien. Bedankt voor het openen van mijn ogen voor de meerwaarde die een gedegen methodologie biedt aan wetenschappelijk onderzoek! Laten we binnenkort weer eens een kopje koffie drinken.

Naast parttime werken aan mijn promotie heb ik volgend op mijn studie nog vier jaar extra parttime gewerkt bij Quarantainenet voordat ik daar fulltime aan de slag ging. Mijn collega's en vrienden daar ben ik dankbaar voor de flexibiliteit gedurende deze periode waarin ik soms voor de lunch nog niet wist waar ik na de lunch aan het werk zou zijn. Het vanzelfsprekende begrip dat zij hadden voor mijn keuze om de wetenschap niet na mijn studie al vaarwel te zeggen, heb ik zeer gewaardeerd!

Ook mijn vrienden met wie ik niet dagelijks samenwerk, verdienen een dankbetuiging. Zij hebben de afgelopen jaren mij zowel geholpen om mijn gedachten te verzetten door voor afleiding te zorgen, als om mijn gedachten te ordenen door geduldig te luisteren wanneer ik een verhaal aan ze kwijt moest. In het bijzonder wil ik kort stilstaan bij mijn paranimfen. Allereerst Stefan, met wie ik samen de studie Informatica doorlopen heb. Na ons beider afstuderen zijn we ieder een eigen kant op gegaan, maar dat heeft geen afbreuk gedaan aan onze interessante gesprekken over wetenschap en tal van andere onderwerpen. Ik ben blij dat ik mijn academische zoektocht met je heb kunnen delen. En daarnaast Jorik, die ik ook tijdens mijn studie in Enschede heb leren kennen. De wijze waarop jij me hebt geholpen mijn werk opzij te zetten wanneer ik dat het meeste nodig had en mij regelmatig hebt geholpen mijn zorgen te relativeren, is ongekend.

Voorts wil ik graag mijn ouders bedanken. Naast hun steun en motivatie gedurende mijn studie en promotietraject, heeft hun opvoeding mij gemaakt tot wie ik ben. Zij hebben mij altijd gestimuleerd om nieuwsgierig te zijn, vragen te blijven stellen en met zelfvertrouwen de wereld te ontdekken. Iets mooiers hadden jullie mij niet mee kunnen geven.

En ten slotte verdient mijn vrouw Gerda alle dank en lof van de wereld. Zij heeft mij aangemoedigd de kans op een promotieplek te grijpen en heeft de gehele periode achter mij gestaan wanneer ik worstelde met onzekerheid of twijfels. Voor de steun, het vertrouwen en een rustig en liefdevol thuis in soms hectische tijden ben ik je enorm veel dank verschuldigd.

Contents

Chapter 1

Introduction

In this thesis several problems concerning the structure of systems of equations are discussed. Such problems occur naturally in many fields and understanding them better may help to improve our ways of solving large systems, for example in industrial applications. This introductory chapter starts by describing the context and motivation for such problems. After this a short, informal description of the concept of computational complexity is given as this forms the basis of the new results. Subsequently, the problems considered in this thesis are briefly introduced as well as the main contribution of this thesis. The sections of this first chapter have been written with non-specialist readers in mind: The subjects are mainly discussed in a non-formal way with a focus on explaining the motivation behind, and a basic intuition for, the concepts involved. The chapter ends by outlining the structure of the remainder of the thesis.

1.1 Constraint Solving

The research of this thesis was carried out in the context of the *Smart Synthesis Tools* (SST) project. This section describes the goals of that project as well as how the research described in this thesis relates to them. The problem of finding a feasible solution to a set of mathematical constraints is one that often occurs in technical disciplines. For example, constraint solving is often a part of mechanical engineering design problems, where many physical requirements have to be met. However, not only industrial problems require solutions to large constraint solving problems. Another area where constraint solving has been getting increased attention is the field of robotics and computer vision. An example application in this field

is the translation of one or more two-dimensional images from cameras mounted on a robot to a three-dimensional model of its surroundings. As such problems become both larger and more common place, automatically finding solutions to them within a reasonable amount of time becomes ever more important.

The use of software and computers to assist designers in finding solutions to design problems is not new: Several decades ago the use of *Computer-Aided Design* (CAD) programs became common practice. The focus of these software systems however was mainly on drawing technical designs and performing calculations on designs based on such drawings. This meant the burden of coming up with the right values for parameters such as distances or weights in the concrete design of a product was still on the human engineers. More recently, automation of this task has come into focus as a subject for academic research projects. The SST project of which the work described in this thesis is a part, is such a research project. The general goal of the Smart Synthesis Tools project is described as follows [1] :

> The objective is a further development of syntheses based design tools, of which several prototypes already have been built in Twente. Synthesis is seen in this context as the process of creating solutions from a set of (incomplete) specifications of the required behavior. The solutions are completely defined and optimal configured designs.
>
> Experiences with the existing prototypes are very promising. They show that it is possible to generate optimal solutions for engineering problems, in significantly shortened time: up to ten times faster than with the current way of creating designs.
>
> For a designer, the biggest gain can be achieved with the selection of a good concept. The research focuses on the development and integration of synthesis tools into a multidisciplinary design support system that can be applied at this concept level of design.
>
> The tools will not, like a wishing well, invent new products, but they will assist engineers take the right decisions early in the process. They also will generate- and evaluate many solutions and help the engineer gain insight in his solution space.

From a mathematical point of view, the phrase "creating solutions from a set of (incomplete) specifications" readily connects to solving potentially large systems of equations and inequalities. This however still leaves us with a large number of highly diverse mathematical subjects. The next section will describe which of these aspects form the focus of this thesis.

1.2 Structural Approach

The approach chosen by the SST project consists of generating a great number of solutions to a design problem using some intelligence and presenting a suitable subset of these generated solutions to a human designer for further evaluation and selection. The input for the system in general consists of a set of degrees of freedom, or *variables*, that can be assigned a value, and a set of constraints or rules that govern the relation between the variables and as such define the allowed designs. In some cases the rules also provide for a mechanism to add additional complexity to an intermediate result in the form of additional variables or constraints that can be dynamically added during the design process. However, we will mostly disregard such situations in this thesis and focus on sets of variables and constraints that are known beforehand. Based on such a fixed set of constraints and variables corresponding to a design problem, the software envisioned as ultimate goal of the SST project generates different combinations of assignments of values to the variables that correspond to different feasible designs for this problem.

Now let us consider generating a single design from this set of constraints and variables. Without going into too much technical detail, we can state that the required effort for finding a solution grows as the set of constraints and variables grows. To be better able to cope with large sets of constraints and variables, from both a software engineering and a human point of view, it is often desirable to be able to decompose a problem into subproblems that can be solved independently. However, truly independent subproblems are often not possible. In that case we can still look for subproblems that can be solved in a given order where the values obtained in the first problem can be substituted into the next as constants and so on, reducing the number of relevant variables in each step. Such structural decomposition problems are an important part of this thesis.

In Chapter 2 we describe the mathematical foundation of our structural analysis in terms of graph theory and bipartite graphs in particular. In that chapter we will see how this decomposition is based purely on the structure of the set of constraints and variables and not on the exact values of any constants that occur in them, either from the start, or as values that have been obtained during the solving of a previous subproblem. This implies that a structural analysis of a design problem is not only useful for the process of generating a single design, but can rather be reused for every iteration of the generating process. Summarizing, the focus on the structural aspects of the mathematical models underlying these design problems is motivated by the following main reasons:

- the same sets of constraints and variables have to be solved many times with variations only in the values of the constants that occur in

the constraints;

- decomposition of a set of constraints and variables helps a human designer by facilitating piece-wise analysis of a model instead of handling the entire model at once;
- if in the future we want to also consider models where the sets of constraints and variables are augmented during the design process itself, structural analysis and decomposition may help guide the incremental build-up of the structure as it may not be required to consider an entire intermediate structure at once.

As in this thesis the focus is on structural rather than numerical aspects of constraint solving and we try to consider general problems regarding decompositions rather than specific case studies, we need some way to compare and assess the usefulness of such decomposition techniques in general. The notion of computational complexity described in the next section lends itself well to exactly such a purpose.

1.3 Computational Complexity

In this section we try to motivate the concept of computational complexity that forms the basis of much of our analysis later on. Most of the problems discussed in this thesis deal with different forms of decompositions of systems of equations in an attempt to speed up the process of finding a solution to such a system by subsequently solving separate (smaller) parts instead of solving everything at once. From a practical point of view, one of the most important aspects to consider is thus whether putting effort into obtaining such a decomposition will actually lead to a reduction in the overall effort required: If the preprocessing required for the decomposition and the effort for solving the separate parts is bigger than the effort required to solve the entire system at once, then putting effort into obtaining a decomposition may not be worthwhile.

To determine whether computing a decomposition may pay off in practice, we could take a number of example problems and simply compare two implementations of a solver on them; one using decomposition as a preprocessing step and one without. However, this would only provide us with insights limited to the combination of these specific example problems and implementations. Chances are that the differences we could measure now for currently relevant problems would no longer be relevant in the not too distant future as faster computers become available. To overcome this, we instead focus most of our analysis on how the required effort grows as problems themselves become larger and larger. This section gives a high-level introduction into the main concepts of this area of mathematics called *computational complexity*.

Most descriptions here aim for an intuitive understanding of the most important concepts and as such are not very formal in a mathematical sense. For example: A formal definition of many of the concepts regarding complexity is usually based on the notion of some computing model such as a Turing machine, but this is far beyond the scope of this thesis. For a more formal and in-depth treatment of this subject, the reader is referred to the standard work by Garey and Johnson [2].

1.3.1 Problems, Instances and Algorithms

Before we can address the subject of complexity itself, we first need to define some common concepts regarding problems and algorithms. In everyday speech what we call a 'problem' is usually a very concrete question to which we have to find an answer. For example: Finding a route home from some unfamiliar place. To solve this problem we may lookup our present location on a map and then try to plot a course in the general direction of our house. If we find ourselves in a different place the next day and again have the desire to return home, we may use the same procedure to get there, even though our starting point is different. (Better yet: the fact that most readers will find this example both familiar and trivial tells us that the procedure outlined here will probably also work for different houses and persons, implying an even higher degree of generality.)

In a mathematical sense we call the generic question 'find a route from one given location to another' a *problem* and the actual question 'find a route from here to my house' an *instance* of that problem. A problem in general thus consists of a description of what constitutes an instance and a question regarding such an instance. A general procedure for solving a problem (such as consulting a map) can then be described and later applied to any of the individual instances even if we don't know the specific details of the instance (the given locations) beforehand. Such a prescription of steps that can be applied to each problem instance to obtain a solution for that instance is called an *algorithm*. We only consider an algorithm as a solution to a problem, and call it an algorithm for that problem, if the algorithm is able to solve all possible instances of the problem.

1.3.2 Complexity of Algorithms

When we talk about the complexity of an algorithm, we usually mean its *running time*, the time it takes to solve instances. This is also known as its *time complexity*. This complexity can roughly be described as an upper bound on the number of 'elementary steps' an algorithm for a specific problem needs to solve an instance of that problem expressed as a function of the *size* of that particular instance. Steps in an algorithm are considered 'elementary' if we may assume them to take some constant unit of time at

most to complete. Formally defining what steps qualify as elementary is out of the scope of this short introduction, but we will try to give an intuitive indication by means of a few examples. The following operations can often be considered elementary:

- Simple mathematical operations, such as addition, subtraction, multiplication, division (assuming their operands are not unreasonably large).
- Simple decisions based on a limited number of values, for example 'compare these two numbers and skip the next step if the first number is at least as large as the second'.
- Using or updating the value of a simple variable (usually a value in a computer's memory).

The following operations are examples of steps that are usually not elementary, because the time they take to complete is typically strongly dependent on (the size of) their operands:

- Finding the lowest number from a list of numbers.
- Sorting a list of numbers.
- Updating all of the elements in a given matrix to a specific value.

Besides the notion of elementary steps or operations, our description of time complexity also hinges on the way we define the size of a problem instance. Defining this concept of size of an instance is not easy in general terms. However, nearly all the problems we discuss in this thesis are problems from combinatorial optimization and in particular from graph theory. This means an instance of a problem is often represented by a graph, so the size of the instance is usually conveniently expressed in terms of the number of vertices or the number of edges or a combination of both. Analogously, when considering systems of equations, reasonable parameters for describing the size of a problem instance could be: The number of equations, the number of variables and the number of 'occurrences' of variables in equations.

For the sake of simplicity it is often advantageous to express the size of an instance using a single parameter that is dominant for the size. For example, the size of a problem instance consisting of a connected graph is often simply taken to be equal to its number of edges.

Assuming that for a given problem we have found a suitable parameter n to measure the size of an instance, we want to analyze the running time of a certain algorithm for this problem. Here we encounter two more complications: Firstly, it is often impractical to assess the exact number of steps for all possible instances, even for a fixed value of n. And secondly, even if

notation	definition
f(n) = $O(g(n))$	$\exists k > 0, n_0 \forall n > n_0 : f(n) \leq k \cdot g(n)$
f(n) = $\Omega(g(n))$	$\exists k > 0, n_0 \forall n > n_0 : k \cdot g(n) \leq f(n)$
f(n) = $\Theta(g(n))$	$\exists k_1, k_2 > 0, n_0 \forall n > n_0 : k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$
f(n) = $o(g(n))$	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0 : \lvert f(n) \rvert \leq \varepsilon \cdot \lvert g(n) \rvert$
f(n) = $\omega(g(n))$	$\forall k > 0 \exists n_0 \forall n > n_0 : k \cdot \lvert g(n) \rvert \leq \lvert f(n) \rvert$

Table 1.1: Notations for asymptotic behavior.

we are able to obtain an exact formula for the required number of steps for the 'hardest' instance at every given size n, such a formula is probably too unwieldy for the purpose of comparing different algorithms. Besides, what we are usually only interested in is what happens for really large values of n: If we increase some n by a factor 10, what may we expect from the time the algorithm needs to solve such a bigger instance? Is it also multiplied by 10? or will it increase by a factor 100? By this same reasoning, constant factors are rarely of interest: If the instances we want to solve are large enough, an algorithm requiring $1000 \cdot n^2$ steps will still be faster than an algorithm requiring n^3 steps. Such considerations motivate the use of so-called *asymptotic behavior* in time complexity analysis. Roughly put, this means we compare algorithms based on the term that dominates an upper bound on their running time when n goes to infinity, while disregarding constant factors. In formal notation: If $f(n)$ denotes the time required to run a certain algorithm on instances of size n, we write $f(n) = O(g(n))$ if for n going to infinity, $f(n)$ is bounded above by $g(n)$ up to some constant factor. This notation together with some other notations for asymptotic behavior are shown with their mathematical definitions in Table 1.1. (Taking absolute values has only been added for the two notations where we actually use it.)

Besides analyzing the amount of time required by an algorithm to solve a problem, we may also be interested in the amount of space it requires to store intermediate results. This is for example relevant if we want to implement the algorithm in a computer program. A program can probably be written for any computer processor, fast or slow, and the choice of processor will only affect the amount of time we have to wait for an answer. However, if the computer we want to use does not have enough memory to store the algorithm's intermediate results, no amount of additional waiting will help; we will simply have to either add more memory or use a different algorithm. The analysis of the amount of space required by an algorithm, its *space complexity*, is again usually performed by investigating its asymptotic behavior and expressed using the same notations from Table 1.1.

1.3.3 Complexity of Problems

Analyzing the time complexity of different algorithms for a problem can give us an impression of how hard the problem itself is in an absolute sense. Intuitively: If we can find a fast algorithm to solve a problem, the problem is apparently not that hard to solve in general (even though finding a fast algorithm may not have been very easy). However, for many problems no acceptably fast algorithms have been found yet and it is often not even known whether such algorithms exist. In many cases it would thus be nice to get an idea of how hard a problem is before investing a lot of effort into the search for a good algorithm that may not even exist. This motivates the analysis of the 'inherent' complexity of problems instead of only analyzing the complexity of known algorithms that solve them.

Furthermore, comparing the complexity of problems where the instances or solutions have different structures is hard to do: The solution to an instance of one problem may be a single number, whereas another problem has entire graphs as solutions to its instances. In part to facilitate the comparison and categorization of different problems, we often analyze *decision problems*. A decision problem is a problem for which the answer to an instance is simply 'yes' or 'no'. For example: Instead of asking for the shortest route between two points on a map, we ask whether there exists a route with a length of at most some value, say 25 kilometers. The instances of a decision problem to which the answer is 'yes' are referred to as *yes-instances* and the other instances are referred to as *no-instances*. Optimization problems are at least as hard as their related decision problems; simply finding the shortest route will tell us if it is shorter than 25 kilometers. So by analyzing decision problems, we can at least obtain a lower bound on the hardness of their non-decision relatives. In the remainder of our discussion on complexity we will assume all problems are decision problems.

An important notion in the categorization of the complexity of problems is that of *polynomial running time* of an algorithm: A running time bounded above by some polynomial of the instance size. A problem that can be solved by such an algorithm is said to be solvable in *polynomial time*. The class of decision problems that can be solved in polynomial time is simply called P. Problems in this class are often considered to be efficiently solvable.

Another important complexity class is NP. NP contains all decision problems with the following characteristic: For each yes-instance of the problem there exists a so-called *certificate* that can be used to verify that the instance is a yes-instance in polynomial time. Clearly, all decision problems that are in P, are also in NP as we can determine whether an instance is a yes-instance in polynomial time even without using a certificate. The converse question is one of the most famous open questions in

mathematics: It is still unknown whether all problems in NP can also be solved in polynomial time, which would imply P= NP. Also, it is important to note that membership in NP does not automatically imply there is also a certificate for every no-instance that can be verified in polynomial time. (The class of problems that allow verification of no-instances using some certificate in polynomial time is called co-NP.)

Establishing membership in the classes P and NP is usually done by providing a polynomial time algorithm for respectively solving the problem or verifying yes-instances using a certificate. Membership in these classes however mainly shows what we *can* achieve (polynomial time solutions or verification of yes-instances); it does not impart a notion of hardness to a problem since $P \subseteq NP$. In order to come to such a concept of hardness, we first need a way to describe that one problem B is 'at least as hard' as another problem A. We say a problem A *reduces* to a problem B, if we can provide a polynomial time algorithm to construct an instance of B for each instance of A in such a way that the constructed instance of B is a yes-instance for B if and only if the original instance of A is a yes-instance for A. This construction procedure is also known as a *(Karp) reduction* from A to B [3]. Clearly, if we have an algorithm to solve B in polynomial time, and we have a polynomial time algorithm to convert instances of A to instances of B while preserving their yes/no-status, we can also solve A in polynomial time by combining these two algorithms. This implies that if we have a reduction from A to B and we can show B is a member of P, then A must be a member of P as well.

In 1971, Stephen A. Cook proved that all problems in NP can be reduced to the problem Satisfiability [4]. In other words: if a polynomial algorithm to solve Satisfiability is ever found, then all problems in NP can be solved in polynomial time, effectively showing P= NP. Problems such as Satisfiability to which all problems in NP can be reduced are called NP-hard. NP-hard problems are not necessarily members of NP themselves, but if they are in NP, they are called NP-complete. Since Satisfiability was proven to be NP-complete, the same has been shown for many other problems through (indirect) reductions from Satisfiability. Even though it is still unknown whether NP contains problems that are not in P, the working hypothesis among most mathematicians is that NP-hard problems are not in P. Under this assumption, proving that a problem is NP-hard implies searching for a polynomial time algorithm is destined to be fruitless. Proofs of NP-hardness thus often play an important role in deciding where to concentrate future research efforts, even though such results are not themselves immediately useable in practical applications.

1.3.4 Parameterized Complexity

Fortunately, not all has been lost for the application-minded once a problem has been shown to be NP-hard. It is sometimes for example still possible to impose additional constraints on the instances to achieve polynomial time solvability. Another promising approach is that of *parameterized complexity* [5]. Beyond considering just the size of a problem instance as leading for the definition of complexity, it is sometimes possible to define additional parameters of the instances or the question of the problem. The time complexity of such problems can sometimes be split into a multiplication of a part dependent only on the parameter, and a part dependent only on the general size of the input. If we have an additional parameter k on top of the instance size n in our problem formulation and our problem can be solved in time $O(f(k)\text{poly}(n))$ (where $\text{poly}(n)$ denotes a polynomial of n) then by fixing the value of k the problem becomes solvable in polynomial time even though this is not the case in general. The fixed parameter value in this case only influences the constant the polynomial of the input size is multiplied by, instead of, e.g., the exponent of this polynomial which would be a lot worse for large values of n from a practical point of view. Consider for example the Vertex Cover problem discussed in the introductory chapter of the seminal work on parameterized complexity by Downey and Fellows [5]: For any fixed value of k we can decide in linear time whether a graph of size n has a vertex cover of size k, whereas the ordinary, non-parameterized version of this problem is NP-complete [2]. It is also possible that more than one additional parameter is identified, but in this thesis we restrict ourselves to a single additional parameter. Parameterized problems that can be solved in time $O(f(k)\text{poly}(n))$ for fixed values of k are called *fixed parameter tractable* and the set of all such problems is denoted by FPT. Analogous to the contrast between P and NP-hard, there are also complexity classes containing problems for which fixed parameter tractability is considered unlikely. One of these classes is $W[1]$-hard. Unfortunately, explaining how this class is defined is beyond this introduction. The working hypothesis is that FPT$\neq W[1]$, so $W[1]$-hard problems are strictly harder than those in FPT in the sense that they will not admit a separation of their running time into a multiplication of a 'general' function of k and a polynomial of n for fixed parameter values. The concepts of membership and hardness are also relevant for parameterized complexity, although the reductions are slightly more involved as they also have to consider possible changes in the parameter values between instances of two problems. Once a problem has been proven to be $W[1]$-hard on top of being NP-hard, there is even less hope for finding efficient algorithms for it.

Having introduced the relevant concepts of complexity, we can now proceed to describe the main subjects of this thesis in the next two sections.

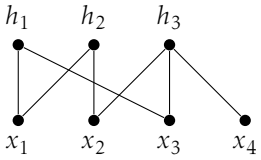


Figure 1.1: The structure of the example system of equations.

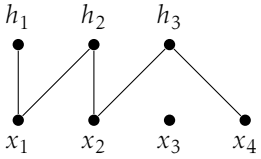


Figure 1.2: The remaining structure after substituting a value for x_3.

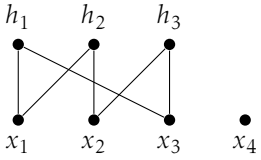


Figure 1.3: The remaining structure after substituting a value for x_4.

1.4 Equations and Decomposition

In this section we give a short introduction and motivation for the research described in Chapter 3. Based on the mathematical foundations of our structural analysis described in Chapter 2 we analyze the general hardness of decomposing a system of equations into parts that can be solved sequentially. To illustrate this problem consider the system of three equations in four variables given by

$$\begin{aligned} h_1(x_1, x_3) &= 0 \\ h_2(x_1, x_2) &= 0 \\ h_3(x_2, x_3, x_4) &= 0\,. \end{aligned}$$

In this system of equations, the exact form of the functions h_i is not relevant, but we assume that they are explicitly dependent on the variables

shown here. The *structure* of this system of equations, i.e., which equation explicitly depends on which variables, is shown in Figure 1.1. Our example system of equations is under-specified as it has one more variable than it has equations. This means we could solve it by assigning a value to one of the variables and solve the remaining equations and variables after that. Let us consider assigning a value to variable x_3 and substituting this value into the equations. The structure of the remaining system of equations is shown in Figure 1.2. The remaining system of equations then allows us to proceed as follows: Equation h_1 now only depends on x_1, so we can use it to obtain a value for x_1. By substituting this value into h_2, we can then easily solve x_2. And finally by substituting the value we find for x_2 into h_3 we are left with one equation in one variable x_4 which can then be solved. (Of course all of this only holds under the assumption that the intermediate systems of equations actually have solutions.)

To illustrate the usefulness of structural analysis, let us consider the original system of equations again and this time start by assigning a value to the variable x_4. The structure of the remaining system after substitution of this value into the equations is shown in Figure 1.3. From this figure it is immediately clear that no equation in the remaining system depends on only a single variable, so in order to solve this remaining system we are forced to consider multiple equations and variables at once.

This simple example shows how the choices we make when solving a system of equations can immediately influence the structure of what remains and how that structure in turn can make the remainder of the solving process easier or harder by forcing us to consider larger parts at once. In Chapter 3 we analyze the computational complexity of several aspects of this general structural decomposition problem. The research described in Chapter 3 is joint work with Georg Still and Walter Kern [6].

1.5 Linear Equations and Elimination

Chapters 4 and 5 describe the second part of the research results in this thesis. These chapters discuss Gaussian elimination and more specifically the process of picking the right pivots to avoid turning zero elements of a matrix into non-zero elements during the elimination process. Gaussian elimination is a classic and still very useful procedure for solving linear equations represented by a matrix. When applying Gaussian elimination to large but sparse matrices the selection of the pivot elements becomes central to reducing both the required computational effort and storage.

To illustrate this, consider the matrix shown in Figure 1.4. If we want to perform Gaussian elimination on this matrix, we have to start by picking a non-zero element and use it to clear the other non-zero elements from its column by subtracting appropriate multiples of its row from the other

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 3 & 2 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 5 & 0 & 1 & 0 \end{pmatrix}$$

Figure 1.4: An example matrix for Gaussian elimination.

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & -1 & -3 & -3 \\ 0 & -4 & -3 & -4 \\ 0 & -5 & -4 & -5 \end{pmatrix}$$

Figure 1.5: The example matrix after using element $(1,1)$ as pivot.

$$\begin{pmatrix} -3 & 1 & 0 & 1 \\ 3 & 2 & 0 & 0 \\ 4 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Figure 1.6: The example matrix after using element $(3,3)$ as pivot instead.

rows. For example if we pick the top-left element $(1,1)$ from our example matrix and use it to clear the left-most column, we end up with the matrix shown in Figure 1.5. Although the matrix we have obtained has only a single non-zero value in the left-most column, every previously zero element in the other columns has been turned into a non-zero. Our choice of pivot has effectively decreased the number of zero elements, reducing the sparsity of the matrix.

In our small example matrix it may not be a big problem to lose a few zero elements, but if we have to process a very large, sparse matrix on a computer losing a lot of sparsity may be detrimental to the overall storage requirements of our program. However by picking another pivot to start with we can do substantially better. Let us consider the original matrix

again but this time we use the element (3,3) as a pivot to clear its column. After this operation we end up with the matrix shown in Figure 1.6. This time we have cleared the third column except for the pivot element itself and we have not turned a single zero into a non-zero along the way. By repeatedly picking the right pivots we can even perform the complete Gaussian elimination procedure on our example matrix without turning a single zero into a non-zero along the way. This example shows how picking the right pivots can help in keeping computing time and storage requirements down for performing Gaussian elimination on sparse matrices.

In Chapter 4 the selection of pivots that completely avoid turning zero elements into non-zero elements is discussed. The chapter first describes the notion of a bisimplicial edge that is closely related to such pivots. After that a new algorithm to find such pivots is described and analyzed on a common class of random matrices.

In Chapter 5 our analysis is expanded from picking a single pivot to the entire Gaussian elimination procedure. This chapter contains two new algorithms that have been devised for the recognition of matrices that allow so-called perfect elimination – elimination without turning any zero into a non-zero. After the description and analysis of these algorithms we turn our attention to a modified version of the Gaussian elimination procedure: one with more fine-grained pivot selection where a new pivot is chosen for every non-zero element that has to be turned into a zero. We round off the chapter by analyzing the computational complexity of this natural generalization of Gaussian elimination for the case where we wish to avoid turning zero elements into non-zeros completely.

The algorithm for finding bisimplicial edges and its analysis in Chapter 4 is joint work with Bodo Manthey [7]. The analysis of the Perfect Partial Elimination problem described in Chapter 5 is joint work with Georg Still and Walter Kern [9].

1.6 Contribution

The main contributions of the original research described in Chapters 3, 4 and 5 of this thesis are:

- Parameterized complexity results regarding several problems related to the decomposition of under-specified systems of equations.
- Proofs of $W[1]$-completeness for two problems regarding crown structures.
- A new deterministic algorithm for finding bisimplicial edges in bipartite graphs.

- Two algorithms for the recognition of perfect elimination bipartite graphs. (One is an adaption of existing work of Goh and Rotem, the other is completely new.)
- A proof of NP-completeness of the recognition of the class of perfect partial elimination bipartite graphs related to a natural fine-grained generalization of Gaussian elimination.

The description of these contributions in this thesis is based on the following publications:

[6] Matthijs Bomhoff, Walter Kern, and Georg Still. On bounded block decomposition problems for under-specified systems of equations. *Journal of Computer and System Sciences*, 78(1):336–347, 2012.

[7] Matthijs Bomhoff and Bodo Manthey. Bisimplicial edges in bipartite graphs. *Discrete Applied Mathematics (CTW2010)*, 2011. in press, DOI: 10.1016/j.dam.2011.03.004.

[8] Matthijs Bomhoff. Recognizing sparse perfect elimination bipartite graphs. In Alexander Kulikov and Nikolay Vereshchagin, editors, *Computer Science – Theory and Applications*, volume 6651 of *Lecture Notes in Computer Science*, pages 443–455. Springer, Berlin, 2011.

[9] Matthijs Bomhoff, Walter Kern, and Georg Still. A note on perfect partial elimination. Technical report, University of Twente, 2011. *Submitted to Discrete Mathematics*.

Although Chapters 3, 4 and 5 do occasionally contain references to other material in the thesis, effort has been put into making each of these chapters a more or less self-contained unit while avoiding too much duplication. This hopefully permits the occasional reader to select and read those parts that are of interest to him or her.

1.7 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 describes the mathematical foundations of the structural approach to constraint solving problems. It introduces the relevant concepts from graph theory and shows how the analysis of bipartite graphs and matchings leads to meaningful results on constraint solving. Chapter 3 discusses the structural decomposition of non-linear, under-specified systems of equations. It starts by describing the Dulmage-Mendelsohn decomposition for bipartite graphs which forms the basis of the analysis. The use of this decomposition for under-specified systems is subsequently analyzed from a parameterized

complexity point of view, leading to several hardness-results. The chapter also discusses some new findings regarding crown structures, derived from the decomposition problem analysis. In Chapter 4 the focus is shifted to linear systems of equations and pivot operations related to Gaussian elimination on such systems in particular. The chapter starts with an introduction into structural analysis of pivot selection for avoiding fill-in during Gaussian elimination. Following the introduction, a new algorithm for pivot selection is presented together with a probabilistic analysis of its performance on a class of random instances. Chapter 5 continues the investigation of linear systems of equations and expands the topic of pivot selection to that of sequences of such pivots that lead to perfect elimination. New algorithms for determining whether a perfect elimination sequence exists in a sparse instance are presented. Finally, it is shown that adapting the traditional Gaussian elimination process to a more flexible procedure using partial pivots makes the problem NP-hard. Chapter 6 discusses the results and briefly describes their impact in the context of constraint solving applications. It also contains suggestions for future research both from a theoretical and from an applied point of view.

Chapter 2

Mathematical Concepts

This chapter introduces many of the general mathematical concepts that form the foundation of the work in this thesis. The focus of the first part of the chapter is on briefly refreshing concepts from graph theory and establishing a consistent terminology regarding them. The second part of the chapter describes the theoretical groundwork behind using graph theory as a tool for structural analysis for systems of equations. This chapter contains no new results and its contents are not meant to be exhaustive, they merely serve to outline the common framework of mathematics that underlies the research presented in the other chapters.

2.1 Graph Theory Concepts

The structural analysis of systems of equations discussed in this thesis is based on a graph theoretical representation of the structure of such systems. Graph theory provides simple yet powerful concepts for structural analysis, but many of these concepts are defined slightly differently by different authors. This section briefly defines the common concepts from graph theory that reoccur frequently in the remainder of this thesis. For a more in-depth treatment of these concepts, the reader is referred to introductory books on the subject (for example the classical work by Berge on graphs and hypergraphs [10]).

A *graph* G is a a tuple (V, E) consisting of a set V of objects called *vertices*, and a set E of unordered pairs of vertices, called *edges*. Although infinite graphs are not by definition excluded, in this thesis all graphs have a finite number of vertices and edges. The unordered pair of vertices x and y is usually written as xy. The vertex and edge sets of a graph G are denoted by $V(G)$ and $E(G)$ respectively. Alternatively, we sometimes also

use V_G and E_G. As E is a set, it can contain each unordered pair of vertices at most once, a graph with this property is sometimes called a *simple graph*. In a given graph $G = (V, E)$, two vertices x, y are called *adjacent* if $xy \in E$. The edge xy is said to *join*, or to be *between*, the vertices x and y. x and y are called the *endpoints* of the edge xy. We say the endpoints are *incident* to the edge. Two edges sharing an endpoint are also said to be *adjacent*. The number of edges incident to a vertex x is called its *degree* and is often written as $\delta(v)$. All vertices adjacent to a vertex x together are called its *neighbors*. The set of neighbors of vertex x is denoted by $\Gamma(x)$. The notion of neighbors can also be extended to a set of vertices: For a subset $V' \subseteq V$ we denote by $\Gamma(V')$ the set of vertices that are adjacent to at least one vertex in V' and are not in V' themselves.

A graph H is called a *subgraph* of a graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Note how by definition of a graph, the edge set of H can only contain edges between vertices of H. Subgraphs are sometimes written using set notation: $H \subseteq G$. A subgraph H of a graph G is called a *proper subgraph* if G contains at least one more vertex or edge than H, in set notation this is written as $H \subset G$.

A subgraph H of a graph G is called an *induced subgraph* if there is no subgraph H' of G such that $V(H') = V(H)$ and $|E(H')| > |E(H)|$. H is said to be induced by its vertex set $V(H)$. A subgraph of a graph G induced by the vertex set X is denoted by $G[X]$

A *walk* W in a graph G is a sequence of edges where it is possible to assign an orientation to each of the edges denoting one of its endpoints as 'head' and the other as 'tail' in such a way that for every pair of consecutive edges, the head of the first coincides with the tail of the second (multiple occurrences of the same edge in the sequence may get different orientations). The number of edges in the sequence is called the *length* of the walk. For every vertex x of G, let us define $\delta_W(x)$ as the number of edges of W incident to x, counting duplicates according to their multiplicity. If $\delta_W(x)$ is even for every x, the walk is called *closed*, otherwise it is called *open*. A closed walk is also called a *tour*. In an open walk, the two vertices of G with $\delta_W(x)$ odd are called the *endpoints* of the walk. All vertices with $\delta_W(x)$ even are said to be *traversed* by the walk. A walk is called *simple* if $\delta_W(x) \leq 2$ for every x. A simple walk is called a *path* if it is open, and a *cycle* if it is closed.

Two vertices x and y are said to be *connected* if there exists a path with x and y as endpoints. Furthermore, we consider every vertex connected to itself. A graph G is called connected if each pair of vertices of G is connected. Clearly, connectivity as a relation is reflexive, symmetric and transitive. As such, it partitions the vertices of a graph G into equivalence classes. Each equivalence class induces a subgraph of G that is connected. These induced subgraphs are called the *(connected) components* of G.

A graph $G = (V, E)$ is called *bipartite* if V can be partitioned into two

sets X and Y such that each edge in E has one endpoint in X and one endpoint in Y. X and Y are then called the *vertex classes* of G. If the partitioning of the vertices into vertex classes is explicit, we use the notation $G = (X, Y, E)$.

A graph $G = (V, E)$ is called a *complete graph* if E contains all possible edges between V: $E = \{uv \mid u, v \in V\}$. A complete subgraph is also called a *clique*. If $G = (X, Y, E)$ is a bipartite graph and E contains every possible edge between X and Y, $E = \{xy \mid x \in X, y \in Y\}$, then G is called a *complete bipartite graph*. A complete bipartite subgraph is also known as a *biclique*.

A *directed graph* (sometimes *digraph*) is a tuple (V, A) consisting of a set of vertices V and a set of ordered pairs of vertices A, called *arcs*. As A is a set, every ordered pair of vertices can occur in A at most once. An arc (x, y) is said to be directed from x (the *tail*) to y (the *head*).

A *directed walk* in a directed graph $G = (V, A)$ is a sequence of arcs where for every pair of consecutive arcs, the head of the first arc coincides with the tail of the second arc. A directed walk is called *closed* if the tail of its first arc is equal to the head of its last arc and *open* otherwise. A directed walk is called *simple* if every vertex in V occurs at most once as head and at most once as tail. Every vertex that occurs as both head and tail in a directed path is said to be *traversed*. An open simple directed walk is called a *directed path*, a closed simple directed walk is called a *directed cycle*.

Two vertices x and y of a directed graph $G = (V, A)$ are called *strongly connected* if there is at least one directed path from x to y and one directed path from y to x. Furthermore, we define every vertex to be strongly connected to itself, so strong connectivity in a directed graph, like connectivity in an ordinary graph, partitions the vertex set into equivalence classes. The subgraphs induced by the equivalent classes of the strong connectivity relation are called *strongly connected components*.

2.2 Matchings

A key role in our graph theoretical analysis of systems of equations is played by the concept of a matching in a graph. This section introduces matchings and provides a couple of results that will be used later on.

A *matching* $M \subseteq E$ of a graph $G = (V, E)$ is a set of edges such that every vertex of G is incident to at most one edge in M. A vertex x is said to be *covered* by a matching M if there is an edge $e \in M$ such that x is incident to e. A set of vertices is said to be covered by a matching if every vertex of the set is covered by the matching. A matching M of G is called *maximal* if no matching M' of G exists such that $M \subset M'$. M is called a *maximum* matching of G if there is no matching M' of G with $|M| < |M'|$. A matching M of G is called *perfect* if it covers $V(G)$. A perfect matching is always maximum and a maximum matching is always maximal. The converse is

not necessarily true. Given a matching M, a path or a cycle is called *M-alternating* if its edges are alternately in M and not in M. A path is called *M-augmenting* if it is M-alternating and its endpoints are not identical and both not covered by M.

Given two sets of edges, M_1 and M_2, we may consider the set of edges that are a member of exactly one of both sets. We call this the *symmetrical difference* of M_1 and M_2 and denote it by $M_1 \Delta M_2$. In set notation: $M_1 \Delta M_2 = (M_1 \cup M_2) \setminus (M_1 \cap M_2)$. The symmetrical difference of two matchings of a bipartite graph has a number of useful properties.

Lemma 2.1. *Let $G = (U, V, E)$ be a bipartite graph and let M_1 and M_2 be two matchings of G, then the following hold:*

1. *The edges of $M_1 \Delta M_2$ together form only paths and cycles in G. In other words: No vertex of G is incident to more than two edges of $M_1 \Delta M_2$.*
2. *If M_1 and M_2 are both perfect matchings of G, $M_1 \Delta M_2$ consists only of cycles. In other words: Every vertex of G is incident to either zero or two edges of $M_1 \Delta M_2$.*

Proof. By definition of a matching each vertex of G can be incident to at most one vertex of M_1. As the same holds for M_2, each vertex of G can be incident to at most two edges in $M_1 \cup M_2$. And as we have that $M_1 \cup M_2 \supseteq M_1 \Delta M_2$, this proves property 1. Property 2 is proven by a simple extension of this reasoning: for each vertex x of G a perfect matching M_1 contains exactly one edge incident to it. The same holds for M_2. If the edges in M_1 and M_2 that are incident to x are equal, then this edge is not a member of $M_1 \Delta M_2$ and so the symmetrical difference contains no edges incident to x. Otherwise, if M_1 contains a different edge incident to x than M_2, then both edges must be in $M_1 \Delta M_2$. This holds for every vertex x of G, so all vertices of G are incident to either zero or two edges of $M_1 \Delta M_2$. This completes the proof. □

Another useful application of the symmetrical difference is the symmetrical difference between a matching M and an M-augmenting path or M-alternating cycle: The symmetrical difference in this case always leads to a new matching. In the case of an M-augmenting path, the new matching contains exactly one more edge than M, hence the name M-augmenting.

We end this section by introducing two famous theorems regarding matchings in bipartite graphs. The reader is referred to [11] and [12] for a more in-depth treatment of these theorems, as well as their proofs.

The first of the two theorems, Kőnig's Minimax Theorem, relates vertex covers to matchings in bipartite graphs. A *vertex cover V'* of a graph $G = (V, E)$ is a subset of V such that every edge in E is incident to at least one vertex in V'.

Theorem 2.2 (Kőnig's Minimax Theorem). *(see e.g. [12]) In a bipartite graph, the cardinality of a maximum matching is equal to the cardinality of a minimum vertex cover.*

The second, P. Hall's Theorem, gives a condition that is both necessary and sufficient for the existence of a matching covering all the vertices in one of the vertex classes in a bipartite graph.

Theorem 2.3 (P. Hall's Theorem [13]). *Let $G = (U, V, E)$ be a bipartite graph. Then G has a matching covering V if and only if $|\Gamma(X)| \geq |X|$ for all $X \subseteq V$.*

2.3 Systems of Equations and Bipartite Graphs

In this thesis, we model the structure of a system of equations by a bipartite graph. This section explains this representation as well as the structural implications we can derive from it regarding the original system of equations. This usage of bipartite graphs for the analysis of systems of equations is not new and has been described before by many authors in different contexts in both papers (e.g. [14, 15, 16]) and books (e.g. [17, 18]) on this subject. The description in this section is an adapted version of that in a paper by Still et al. on the meaning of the concept of consistency [19].

We consider a system of m equations in n unknowns of the form

$$h_i(x) = 0, \quad i \in I := \{1, \ldots, m\}, x = (x_1, \ldots, x_n). \tag{2.1}$$

Instead of the actual form of the equations $h_i(x)$, we are interested in the structure of this system of equations, i.e., which equations depend explicitly on which variables. This structure can be represented by a bipartite graph with one vertex class representing the equations, the other vertex class representing the variables and the edges between the two classes representing the explicit occurrence of the variables in the equations. This bipartite graph G and its edge set E are thus constructed as

$$E := \left\{(i, j) \in I \times J \mid h_i(x) \text{ depends explicitly on } x_j\right\} \tag{2.2}$$

$$G := (I, J, E). \tag{2.3}$$

Here the index sets I and J of respectively the equations and the variables are used as their vertex sets in the graph G. We will illustrate this construction by an example.

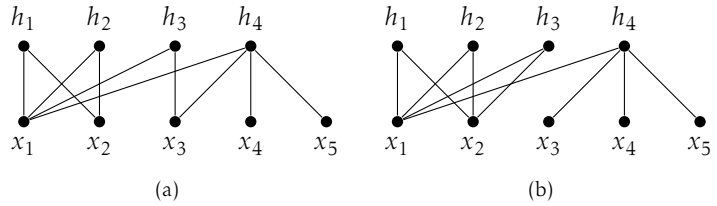


Figure 2.1: Bipartite graphs corresponding to example systems.

Consider the following system of 4 equations in 5 unknowns:

$$\begin{aligned} h_1(x) &= x_1 - x_2 &&= 0 \\ h_2(x) &= x_1 + x_2 - 4 &&= 0 \\ h_3(x) &= x_1^2 - x_3 &&= 0 \\ h_4(x) &= x_1 + x_3 + x_4 + x_5 &&= 0 \end{aligned}$$

The corresponding bipartite graph of the structure of this system is shown in Figure 2.1(a). To solve this system, we first obtain $x_1 = x_2 = 2$ from equations h_1 and h_2. h_3 then leads to $x_3 = 4$ and we can finally use h_4 and either freely choose a value for x_4 and solve x_5 or choose a value for x_5 and solve x_4. This freedom is directly attributable to the fact that we have more variables than equations.

Now let us consider a slightly adapted version of our example system of equations where h_3 no longer depends on x_3 but depends on x_2 instead. This leads to the following system of equations:

$$\begin{aligned} h_1(x) &= x_1 - x_2 &&= 0 \\ h_2(x) &= x_1 + x_2 - 4 &&= 0 \\ h_3(x) &= x_1^2 - x_2 &&= 0 \\ h_4(x) &= x_1 + x_3 + x_4 + x_5 &&= 0 \end{aligned}$$

The adapted bipartite graph corresponding to this system is shown in Figure 2.1(b). Again, we may start solving the system by obtaining $x_1 = x_2 = 2$ from equations h_1 and h_2. However substituting these values into h_3 now leads to a contradiction. Clearly, there is no valid solution to this example system.

The problem in this example is that the value of a variable can be obtained simultaneously through solving different parts of the system leading to conflicting results. To avoid such structural inconsistency, we can

impose a structural constraint. A moment of reflection shows that the following condition should hold assuming there are no redundant constraints: For any subset of the equations $I_0 \subseteq I$ the number of variables appearing in these equations, should not be smaller than the number of equations in the subset, i.e., the cardinality $|I_0|$ of I_0. (Example 2.11 at the end of this chapter illustrates how redundant constraints can complicate matters.) In the bipartite graph G this condition translates to

$$|\Gamma(I_0)| \geq |I_0| \quad \forall I_0 \subseteq I\,.$$

By Theorem 2.3 this implies we can avoid structural inconsistency in a system of equations by requiring that the associated bipartite graph G has a (maximum) matching covering I. Note how this condition immediately implies that the number of variables n must be at least as large as the number of equations m. In what follows, we will call a system of equations *(structurally) consistent*, if its associated bipartite graph G has a matching covering all of the vertices corresponding to the equations of the system.

In a given consistent system of equations $h_i(x) = 0$ with associated bipartite graph $G = (I, J, E)$, a matching M covering I naturally induces a partition of the variables x_j. If we denote by $B = \{j \in J \mid j \text{ is covered by } M\}$ the set of vertices corresponding to variables that are covered by M, then $x_B = \left\{x_j \mid j \in B\right\}$ and $x_F = \left\{x_j \mid j \notin B\right\}$ together form a partition of the variables x_j. The $n - m$ variables in x_F are called *free variables* for the system with respect to the particular matching M. According to the consistency concept the values for these variables can be chosen so that for any choice of x_F we are left with a structurally consistent system $\tilde{h}_i = 0$ with an equal number of equations and variables:

$$\tilde{h}_i(x_B) := h_i(x_B, x_F) = 0, \quad i = 1, \ldots, m\,.$$

If the general system $h_i(x) = 0$ has a solution (which may of course not even be the case) it is important to note that not all possible choices for values of x_F may lead to a system $\tilde{h}_i = 0$ that actually has a solution (consider for example the system $h(x) = e^{x_1} - x_2 = 0$ with $x_F = \{x_2\}$ and $x_2 = 0$). Furthermore, the set of possible maximum matchings of G dictates the possible partitions of x_j: partitions of x_j for which no suitable maximum matching exists can not lead to structural consistency.

From the preceding discussion, we know that the bipartite graph $G = (I, J, E)$ associated with a system of equations must have a maximum matching covering I in order for the system to be structurally consistent. Although different choices for this matching may result in different partitions of x_j into free and non-free variables, the free variables must all be assigned a value before we can meaningfully determine values for all of the non-free variables. Motivated by this, we restrict the remainder of our

analysis of the concept of consistency to the case where the number of equations is equal to the number of variables, i.e., $m = n$.

Formally, this means we consider a system of n equations in n unknowns,

$$h_i(x) = 0, i \in I := \{1, \dots, n\}, \quad x = (x_1, \dots, x_n) \in \mathbb{R}^n \tag{2.4}$$

with a structure given by a bipartite graph $G = (I, J, E)$ with $|I| = |J|$.

Definition 2.4. Let the structure of (2.4) be given by the bipartite graph $G = (I, J, E)$. We then call the system (2.4) consistent if G has a perfect matching.

Remark 2.5. Sometimes, instead of consistency the notion *structural solvability* is used. We again point out that our notion of consistency does not imply solvability. For example the following system is structurally consistent, but does not have a solution:

$$\begin{aligned} h_1(x) = x_1 + x_2 - 1 &= 0 \\ h_2(x) = x_1 + x_2 - 2 &= 0 \end{aligned}$$

Furthermore, consistency as defined in terms of the bipartite graph G does not say something about a concrete instance of a system of equations, but rather has a meaning for the whole class of systems characterized by the structure of G.

In the next section we further analyze this concept of consistency for linear systems of equations.

2.4 Consistency for Linear Equations

For the case of linear equations we restrict ourselves to systems of equations of the form

$$Ax - b = 0, \quad A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n \tag{2.5}$$

and consider the structure of the corresponding bipartite graph $G = (I, J, E)$. This bipartite graph allows us to define an entire class of systems of equations.

Definition 2.6. Given a bipartite graph $G = (I, J, E)$ we introduce the corresponding class of structured matrices

$$M_G = \left\{ A \in \mathbb{R}^{n \times n} \mid a_{ij} = 0 \text{ for } (i, j) \notin E \right\} \cong \mathbb{R}^{|E|}.$$

This bipartite graph and associated class of matrices also lead to a problem class P_G regarding systems of equations of the form (2.5),

$$P_G: \quad \text{solve } Ax = b \text{ with } A \in M_G, b \in \mathbb{R}^n.$$

We call the problem class P_G *consistent* if G allows a perfect matching (a matching covering I) and *inconsistent* otherwise.

For the special case of linear systems of equations, the concept of consistency is closely related to that of the *non-singularity* of the matrix $A \in M_G$. Non-singularity of A implies that the system $Ax = b$ has a unique solution. In particular, in what follows we will show that for almost all $A \in M_G$ the matrix A is non-singular if and only if P_G is consistent.

The basic implication of the notion of consistency for linear systems of equations is given by

Theorem 2.7. *The set M_G contains a non-singular matrix A_0 if and only if G allows a perfect matching.*

Proof. Let M be a perfect matching in the bipartite graph $G = (I, J, E)$. Then the matrix $A_0 \in M_G$ given by

$$A_0 = (a_{ij}), \quad a_{ij} = \begin{cases} 1 \text{ if (ij)} \in M \\ 0 \text{ otherwise} \end{cases}$$

is a permutation matrix with $\det A_0 = \pm 1$, i.e., A_0 is a non-singular matrix.

For the proof in the other direction, assume that G does not allow a perfect matching and let $A \in M_G$ be an arbitrary matrix of the class M_G. Then by Theorem 2.3 there exists a subset $I_0 \subset I$ with $|\Gamma(I_0)| < |I_0|$. Let $J_0 = \Gamma(I_0)$, $r = |I_0|$ and $k = |J_0|$. The r rows of A corresponding to I_0 must have all zero values in at least $n - k > n - r$ columns, implying that these rows are linearly dependent, which in turn implies A must be singular. □

The following lemma will be useful to us in refining the meaning of consistency for the classes P_G. We have not been able to find a proper proof of this lemma in the available literature, even though the result itself seems rather intuitive. The proof below is an adapted version of a proof by Caron and Traynor found in a seemingly unpublished note [20].

Lemma 2.8. *Let $p : \mathbb{R}^K \to \mathbb{R}$ be a polynomial mapping, $p \neq 0$ and denote by λ_K the K-dimensional Lebesgue measure. Then the subset of $\mathbb{R}^K$ mapped to zero by p has Lebesgue measure zero, i.e., $\lambda_K \left\{ x \in \mathbb{R}^K \mid p(x) = 0 \right\} = 0$.*

Proof. The proof proceeds by induction on K, the dimension of the vector x of variables. If $K = 1$ and $p(x) : \mathbb{R} \to \mathbb{R}$ is a non-zero polynomial, then clearly $p(x)$ has a finite set of roots with zero measure: $\lambda_1 \{x \in \mathbb{R} \mid p(x) = 0\} =$

0. Now suppose the result is true for polynomials in $K-1$ variables and let $p(x) = p(x_1, x_2, \ldots, x_K)$ be a non-zero polynomial of K variables with maximum degree M. For indices $i_1, i_2, \ldots, i_K \in \{0, \ldots, M\}$ and constants $\alpha_{i_1,i_2,\ldots,i_K} \in \mathbb{R}$ we may write $p(x)$ as

$$p(x) = p(x_1, x_2, \ldots, x_K) = \sum_{i_1,i_2,\ldots,i_K} \alpha_{i_1,i_2,\ldots,i_K} \cdot x_1^{i_1} x_2^{i_2} \ldots x_K^{i_K} .$$

Now let $q_{i_2,\ldots,i_K}(x_1) = \sum_{i_1} a_{i_1,i_2,\ldots,i_K} \cdot x_1^{i_1}$ then we may also write $p(x)$ as

$$p(x) = p(x_1, x_2, \ldots, x_K) = \sum_{i_2,\ldots,i_K} q_{i_2,\ldots,i_K}(x_1) \cdot x_2^{i_2} \ldots x_K^{i_K} .$$

Since p is not the zero polynomial, there is a set of indices $(i_2, \ldots, i_K)$ for which $q_{i_2,\ldots,i_K}(x_1) \neq 0$. As each non-zero $q_{i_2,\ldots,i_K}(x_1)$ is a polynomial of only a single variable, the set of its roots is finite. Let us denote by N the set of all values $x_1 \in \mathbb{R}$ such that $p(x)$ is zero for all values of $(x_2, \ldots, x_K)$. I.e., $N := \{x_1 \in \mathbb{R} \mid p(x_1, x_2, \ldots, x_K) = 0, \text{ for all } x_2, \ldots, x_K\}$. If for some value $x_1 \in N$ we have that $p(x) = 0$ for all values of $(x_2, \ldots, x_K)$ then we must have that each $q_{i_2,\ldots,i_K}(x_1) = 0$. So N is the intersection over all indices $(i_2, \ldots, i_K)$ of the sets of roots of $q_{i_2,\ldots,i_K}$. Clearly, N is finite and thus

$$\lambda_1(N) = 0 . \tag{2.6}$$

Now denote by $N^c := \mathbb{R} \setminus N$ the complement of N and consider a fixed $x_1 \in N^c$. For such a value of x_1, we know by our induction hypothesis

$$\lambda_{K-1}\left\{(x_2, \ldots, x_K) \in \mathbb{R}^{K-1} \mid p(x_1, x_2, \ldots, x_K) = 0\right\} = 0 . \tag{2.7}$$

Combining the two, we obtain

$$\begin{aligned}
\lambda_K \{x \in \mathbb{R} \mid p(x) = 0\} &= \int \lambda_{K-1} \{(x_2, \ldots, x_K) \mid p(x_1, x_2, \ldots, x_K) = 0\} \, dx_1 \\
&= \int_N \lambda_{K-1} \{(x_2, \ldots, x_K) \mid p(x_1, x_2, \ldots, x_K) = 0\} \, dx_1 \\
&\quad + \int_{N^c} \lambda_{K-1} \{(x_2, \ldots, x_K) \mid p(x_1, x_2, \ldots, x_K) = 0\} \, dx_1 \\
&= 0 .
\end{aligned}$$

□

Let us consider the polynomial mapping p on $M_G \cong \mathbb{R}^{|E|}$ given by

$$p : M_G \to \mathbb{R}, \quad p(A) = \det A .$$

According to Theorem 2.7 we have $p \neq 0$ if and only if G allows a perfect matching. Together with Lemma 2.8 this leads to the following corollary.

Corollary 2.9. *The subset $M_G^0 = \{A \in M_G \mid \det A = 0\} \subseteq M_G$ of singular matrices has (Lebesgue) measure zero if and only if G allows a perfect matching.*

Now let us consider the set $M_G^r = M_G \setminus M_G^0$ of regular matrices. By Corollary 2.9 this set has full Lebesgue measure if G allows a perfect matching and by Theorem 2.7 it is empty otherwise. For a given G this means that for almost all $A \in M_G$ the system $Ax = b$ is uniquely solvable if and only if G allows a perfect matching. If G has a perfect matching and $A_0 \in M_G$ is an arbitrary matrix with $p(A_0) = \det A_0 \neq 0$, then by the continuity of $p(A)$ clearly there exists some ε-neighborhood of A_0 in which $p(A) \neq 0$. This shows M_G^r is an open subset of M_G if G has a perfect matching. Furthermore, if G has a perfect matching and $A_0 \in M_G$ is a matrix such that $p(A_0) = 0$, then by an ε-perturbation of A_0 we obtain a matrix $\tilde{A}_0$ with $p(\tilde{A}_0) \neq 0$, showing M_G^r is also dense in M_G. These results are summarized in the main theorem of this section.

Theorem 2.10. *The set M_G^r is open and dense in M_G if and only if G allows a perfect matching.*

Simply put, Theorem 2.10 states that consistency as defined as the existence of a perfect matching in G corresponds exactly to the systems of equations $Ax = b$ with $A \in M_G$ for which we may expect to find a (unique) solution due to the regularity of the matrix A.

For the case of non-linear equations a similar, albeit significantly more involved, analysis of the concept of consistency can be performed. Such an analysis is however too far from the main subject of this thesis. We will restrict ourselves here to mentioning that also in the non-linear case a perfect matching of the bipartite graph representing the structure of the system of equations corresponds to the analogous concept of consistency. The interested reader is referred to the paper of Still et al. [19] for further details.

We round off this chapter with an example to illustrate the limitations of structural analysis for the concept of consistency.

Example 2.11. Consider the following system of three linear equations in two variables x_1, x_2 where $\alpha \in \mathbb{R}$ and $\beta \in \mathbb{R} \setminus \{0\}$ are fixed parameters:

$$x_1 + x_2 = 1 \tag{2.8}$$
$$x_1 + x_2 = 1 + \alpha \tag{2.9}$$
$$x_1 + \beta x_2 = 1\,. \tag{2.10}$$

As $\beta \neq 0$, the bipartite graph corresponding to the structure of this system of equations is shown in Figure 2.2.

Structurally, this system is clearly over-specified, as the number of equations is larger than the number of variables. However, if we look beyond

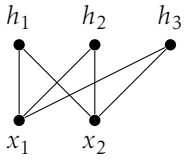


Figure 2.2: Example over-specified system of linear equations.

the structure, we see three significantly different cases can arise based on the values of the parameters α and β.

- If $\alpha \neq 0$, then equations 2.8 and 2.9 are contradictory and the system has no solution.
- If $\alpha = 0$ and $\beta \neq 1$, then equations 2.8 and 2.9 are identical and equation 2.10 is different so the system has a unique solution.
- If $\alpha = 0$ and $\beta = 1$, then equations 2.8, 2.9 and 2.10 are identical and the system has an infinite number of solutions.

Example 2.11 shows the limitations of our notion of (structural) consistency, even for the relatively simple case of linear equations: Even though structural analysis tells us the system is over-specified, depending on the actual values of α and β it can still have zero, one or an infinite number of solutions.

Chapter 3

Bipartite Graphs and Decompositions

When faced with a large problem (more formally: a large instance) that requires a solution, a sensible approach is often to consider breaking up the problem into smaller subproblems that can be solved separately and subsequently be combined into a single solution of the overall problem. In this chapter we analyze the viability of such an approach to solving general systems of non-linear equations. The chapter first describes the Dulmage-Mendelsohn decomposition, a well-known decomposition for bipartite graphs that naturally corresponds to a decomposition of a system of equations. This part of the chapter contains no new results, it merely provides the foundation for what follows. Subsequently we show that decomposing an under-specified system of equations into subsystems that are as small as possible is $W[1]$-hard. Based on these new results we can also answer a couple of hitherto open questions regarding crown structures for the parameterized vertex cover problem.

3.1 The Decomposition of Bipartite Graphs

The previous chapter discussed bipartite graphs as a representation of the structure of systems of equations. It was shown how, based on the bipartite graph, consistency or structural solvability can be determined using matchings. The structural information in the bipartite graph can, however, not only be used to determine consistency; it can also be used to find possible decompositions of a system into subsystems that can be solved separately.

The decomposition of bipartite graphs that allows us to find such subsystems was first described by Dulmage and Mendelsohn and called the *canonical decomposition* [14]. This decomposition is sometimes also simply called the Dulmage-Mendelsohn decomposition, the name we will also use. The first description by Dulmage and Mendelsohn of their decomposition was not based on matchings, but rather on the related concept of *(external) covers* and focused mainly on applications in matrices [21]. Later, the decomposition has also been described in terms of matchings (see e.g. [22, 11]), which is also the route we take below. We present a proof constructed from scratch mainly to focus on structural properties of the decomposition, rather than its construction. However, our proof has of course been influenced by previous work on the subject.

The canonical decomposition as described by Dulmage and Mendelsohn actually consists of two decompositions: A coarse and a fine decomposition. The coarse decomposition decomposes an arbitrary bipartite graph into three (possibly empty) parts. The fine decomposition further decomposes one of these parts into a number of irreducible blocks. Both decompositions are discussed in detail below together with several of their properties that will be of use to us in describing the relation to systems of equations.

3.2 The Coarse Dulmage-Mendelsohn Decomposition

Let $G = (U, V, E)$ be a connected bipartite graph and let $\mathcal{M}$ be the set of maximum matchings of G. The *coarse Dulmage-Mendelsohn decomposition* of G is defined using a partition of the vertices of G over three sets: D is the set of vertices v for which there is at least one maximum matching of G not covering v. A is the set of their neighbors and C is the set of remaining vertices. Formally:

$$D := \{x \in U \cup V | \exists M \in \mathcal{M} : x \text{ is not covered by } M\}$$
$$A := \Gamma(D) \setminus D$$
$$C := (U \cup V) \setminus (D \cup A)$$

An example of this partition is shown in Figure 3.1. For notational convenience, let $D_U = D \cap U$, etc. Using the sets D, A and C, we define the following subgraphs of G that form its coarse Dulmage-Mendelsohn decomposition:

$$G_1 := G[C]$$
$$G_2 := G[D_U \cup A_V]$$
$$G_3 := G[D_V \cup A_U]$$

An example of this decomposition is shown in Figure 3.2. We proceed by proving some useful properties of the coarse decomposition.

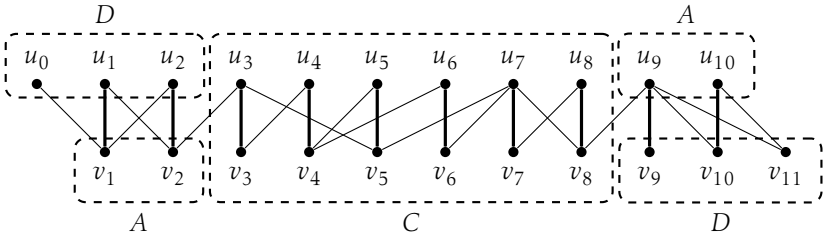


Figure 3.1: Example sets A, D and C for the coarse Dulmage-Mendelsohn decomposition.

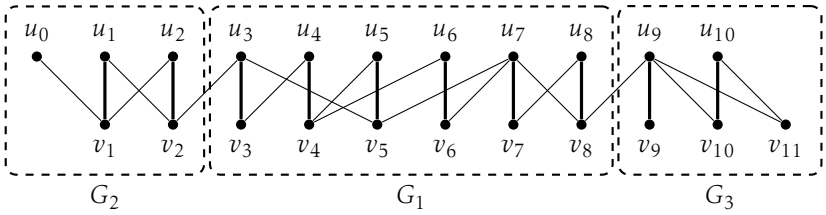


Figure 3.2: Example coarse Dulmage-Mendelsohn decomposition.

Lemma 3.1. *There is no edge in E connecting two vertices $x, y \in D$.*

Proof. As G is bipartite, we only have to consider the case where x and y are in different vertex classes. Assume without loss of generality that $x \in D_U$ and $y \in D_V$. To prove our claim, we show there exists a maximum matching M^* that covers neither x nor y, thus contradicting the existence of an edge xy.

Let M be a maximum matching which does not cover x and let M' be a maximum matching which does not cover y. If either M or M' does not cover x and y, we have found M^*, so let's assume this is not the case. In $M \Delta M'$, y must be the endpoint of some M-alternating path P of even length. As x also has degree one in $M \Delta M'$, we know P neither traverses nor ends in x. $M \Delta P$ can be used as the required matching M^*. □

Lemma 3.2. *In any maximum matching M, every vertex $x \in A$ is matched to a vertex $y \in D$.*

Proof. Assume there exists a maximum matching M and a vertex $x \in A$ such that x is matched to $y' \in A \cup C$. We will derive a contradiction. By construction of the decomposition, x has a neighbor $y \in D$. If y is not

covered in M, $M\Delta\{yx, xy'\}$ is a maximum matching that does not cover y', leading to a contradiction. So assume y is covered by M and let M' be a maximum matching in which y is not covered. $M\Delta M'$ contains an M-alternating path P of even length from a vertex y'' which is not covered in M to y. If P contains xy', let P' be the first part of P from y'' to y'. As y, y' and y'' are all in the same vertex class, P' must be of even length and its last edge must be xy', so $M\Delta P'$ is a maximum matching that does not cover y', leading to a contradiction. Finally, if P does not contain xy', $M\Delta P$ is again a maximum matching containing the edge xy' and in which y is not covered, leading to a contradiction. □

Lemmata 3.1 and 3.2 together trivially lead to the following corollary describing the structure of every maximum matching of G.

Corollary 3.3. *Any maximum matching consists only of edges between A and D and edges connecting vertices of C to other vertices of C.*

A possible maximum matching for our example bipartite graph is shown using vertical, thick lines in Figure 3.2.

3.3 The Fine Dulmage-Mendelsohn Decomposition

Part G_1 of the coarse Dulmage-Mendelsohn decomposition can be decomposed further using the so-called *fine Dulmage-Mendelsohn decomposition*. Before we describe the fine decomposition itself, we first define the concept of irreducible bipartite graphs.

Definition 3.4. A bipartite graph $G = (U, V, E)$ is called *irreducible* if it is connected and for each of its edges $e \in E$ there exists a perfect matching M of G such that $e \in M$.

The following lemma describes a useful property of irreducible bipartite graphs.

Lemma 3.5. *Let $G = (U, V, E)$ be an irreducible bipartite graph and let M be an arbitrary perfect matching of G. For each $u \in U, v \in V$ there exists an M-alternating path between u and v starting and ending with an edge in M.*

Proof. Let $u \in U$ and let $S \subseteq U \cup V$ be the set consisting of u and all vertices reachable from u by an M-alternating path starting with an edge in M. As M is a perfect matching, $|S \cap U| = |S \cap V|$ and each vertex in S is matched to another vertex in S. Now assume S is a proper subset of $U \cup V$. As G is connected, there must be an edge xy with $x \in S$ and $y \in (U \cup V) \setminus S$. Edge xy cannot be in M, so $x \notin V$ as that would imply there was an M-alternating path from u to y, contradicting $y \in (U \cup V) \setminus S$. Hence we must have $x \in U \cap S$ and $y \in V \setminus S$. Now let M' be a perfect matching containing

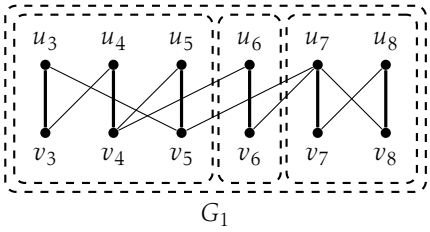


Figure 3.3: Example fine Dulmage-Mendelsohn decomposition.

xy. By a simple counting argument, M' must also contain at least one edge $u'v'$ with $u' \in U \setminus S$ and $v' \in V \cap S$. However, that would in turn imply the existence of an M-alternating path from u to u' contradicting $u' \in U \setminus S$. We conclude that S cannot be a proper subset of $U \cup V$, so $S = U \cup V$, proving our lemma. □

The fine decomposition splits a connected bipartite graph with a perfect matching into irreducible components. It is defined as follows: Let $G = (U, V, E)$ be a connected bipartite graph with a perfect matching. Construct the subgraph $H = (U, V, E')$ consisting of those edges of G that are contained in some perfect matching. In other words, if $\mathcal{M}$ denotes the set of all perfect matchings of G, $E' = \{e \in E \mid \exists M \in \mathcal{M} : e \in M\}$. The fine decomposition of G now consists of the components H_i of H. An example of the fine decomposition of G_1 from our course decomposition example is shown in Figure 3.3.

Clearly, the fine decomposition is well-defined and unique. We proceed by proving some properties of the fine decomposition that help us in decomposing systems of equations into smaller subsystems.

Lemma 3.6. *Let $G = (U, V, E)$ be a connected bipartite graph with a perfect matching and let $H = (U, V, E')$ with components H_i be its fine decomposition. For each edge $uv \in E \setminus E'$ with $u \in H_i, v \in H_j$ we have $i \neq j$.*

Proof. Assume to the contrary that there exists some edge $uv \in E \setminus E'$ with $u, v \in H_i$ for some i. From Lemma 3.5 we know that for every maximum matching M of G, there exists an M-alternating path from u to v starting and ending with an edge in M. Together with uv this path forms an M-alternating cycle C, so $M \Delta C$ forms another perfect matching of G with $uv \in M \Delta C$, contradicting $uv \in E \setminus E'$. Ergo, each edge $uv \in E \setminus E'$ must connect two different components H_i and H_j. □

Theorem 3.7. *Let $G = (U, V, E)$ be a connected bipartite graph with a perfect matching and let $H = (U, V, E')$ with irreducible components H_i be its fine decomposition. Then there is no sequence of irreducible components $H_1, H_2, \ldots, H_c$ such that there exists a set of edges $\{u_1v_2, u_2v_3, \ldots, u_cv_1\} \subseteq E \setminus E'$ with $u_i, v_i \in H_i$.*

Proof. Assume to the contrary that a sequence of irreducible components and corresponding edges exists. Let M be a perfect matching of G. From Lemma 3.5 we know that for each i there is an M-alternating path in H_i from u_i to v_i starting and ending with an edge in M. These paths combined with the edges $\{u_1v_2, u_2v_3, \ldots, u_cv_1\}$ form an M-alternating cycle C. So $M \Delta C$ is a perfect matching of G contradicting that the edges of $\{u_1v_2, u_2v_3, \ldots, u_cv_1\} \subseteq E \setminus E'$ are not in any perfect matching of G. □

3.4 Constructing the Dulmage-Mendelsohn Decomposition

The preceding discussion of the Dulmage-Mendelsohn decomposition focused on structural properties. In this section we approach the decomposition from a more constructive point of view and describe how it can be determined for arbitrary bipartite graphs in polynomial time.

From the definition of the coarse decomposition in Section 3.2 it is clear that once we know which vertices are in the set D, determining the rest of the decomposition is easy. Fortunately, our structural analysis of the decomposition leads to a practical way to determine the vertices in this set.

Theorem 3.8. *Let M be a maximum matching of G, D consists of exactly those vertices of G that are either not covered by M or can be reached by M-alternating paths of even length starting from the vertices not covered by M.*

Proof. It is clear that all vertices not covered by M, are in D. Now consider any M-alternating path P of even length starting at a vertex not covered by M. We know from Lemma 3.1 that the first edge of P (not in M) leads to a vertex in A, and from Corollary 3.3 that the second edge (in M) leads back to D and so on. So the other endpoint of P must also be in D. For the converse, it is clear that all vertices not covered by M are in D, so consider a vertex x in D which is covered by M. By definition, there exists a maximum matching M' which does not cover x. $M \Delta M'$ thus contains an M-alternating path of even length from a vertex not covered by M to x. □

Theorem 3.8 provides us with an easy way to obtain the coarse Dulmage-Mendelsohn decomposition for a given bipartite graph G: starting from a maximum matching M of G, we first determine D using M-alternating

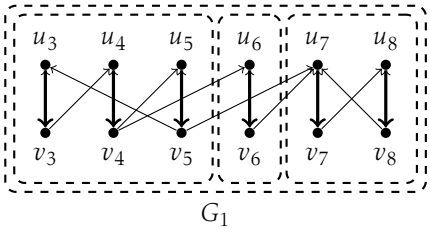


Figure 3.4: Example fine Dulmage-Mendelsohn decomposition.

paths of even length from vertices not covered by M. A and C then immediately follow from the definition of the decomposition. Finding a maximum matching of G can be done in time $O\left(|E|\sqrt{|U|+|V|}\right)$ using the algorithm of Hopcroft and Karp [23] . Determining which vertices can be reached by M-alternating paths of even length starting from the vertices not covered by M can also be done in polynomial time, for example using a breadth-first search from each of these vertices.

In what follows, we will show that the fine decomposition can subsequently also be obtained in polynomial time using a directed bipartite graph constructed from the part G_1 of the coarse decomposition.

Theorem 3.9. *Let $G = (U, V, E)$ be a connected bipartite graph graph with a perfect matching M. Construct the directed graph G' from G by orienting all edges of G from V to U. Also add to G' additional edges corresponding to the edges in M directed from U to V. The irreducible components of G correspond to the strongly connected components of G'.*

Proof. Let $u \in H_i$ and $v \in H_j$ with $i \neq j$. By Theorem 3.7 we know there is no directed cycle connecting two or more irreducible components, so if u and v are in different components of the fine decomposition of G, they are not strongly connected in G'.

For the converse, let $u, v \in H_i$. If u is matched to v in M, they are clearly strongly connected. So assume this is not the case and u and v are matched to v' and u' respectively in M. By Lemma 3.6 there exists an M-alternating path from u to v in G that corresponds to a directed path in G'. Furthermore, such a path also exists from u' to v'. Together with the directed edges from v' to u and from v to u', these paths form a directed cycle, showing that u and v are strongly connected. □

An example of the directed graph G' constructed from part G_1 of our example decomposition is shown in Figure 3.4. Finding a perfect matching in G (for example by the algorithm of Hopcroft and Karp), constructing

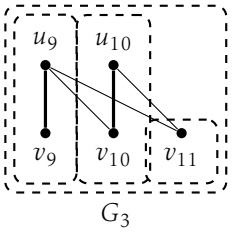


Figure 3.5: Possible decomposition of G_3.

the corresponding directed graph G' as described in Theorem 3.9 and determining its strongly connected components (for example using Tarjan's algorithm [24]) can be done in polynomial time.

For a more in-depth description of the algorithmic and complexity aspects of the Dulmage-Mendelsohn decomposition the reader is referred to existing literature on the matter, for example work by Ait-Aoudia et al. discussing the application of the Dulmage-Mendelsohn decomposition in the reduction of constraint systems for geometric modeling [15].

3.5 The Dulmage-Mendelsohn Decomposition and Systems of Equations

The Dulmage-Mendelsohn decomposition of a bipartite graph $G = (U, V, E)$ corresponding to a system of equations has a natural interpretation: If U represents the equations and V represents the variables, G_2 corresponds to the over-constrained part of the system, G_1 corresponds to the well-constrained part, and G_3 corresponds to the under-constrained part. In the remainder, we will assume G_2 to be empty, such that the system of equations is consistent and lacks an over-constrained part.

The fine decomposition of G_1 corresponds to a decomposition of the corresponding part of the system of equations into subsystems that can be solved successively in an order determined by the connections between the components. These connections represent the dependency of equations in one component on variables in another component and thus logically dictate the order of solving of the components.

Solving a large system of equations at once is often harder than solving its subsystems separately, so obtaining a decomposition into subsystems that can be solved in order can be beneficial. For the well-constrained part G_1, the fine Dulmage-Mendelsohn decomposition corresponds to the unique decomposition of G_1 into irreducible subsystems that can be solved separately. For the under-constraint part G_3 however, this decomposition

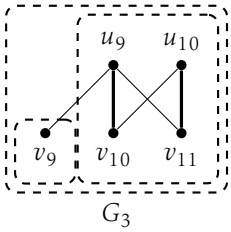


Figure 3.6: Alternative decomposition of G_3.

is no longer unique. Each maximum matching of G_3 leaves one or more vertices corresponding to variables uncovered. By assigning a value to these variables, the remainder of the system corresponding to G_3 becomes well-constrained and gets a perfect matching. It can then be decomposed using the fine decomposition and subsequently solved. However, different maximum matchings and their resulting uncovered variables can lead to different decompositions as can be seen in Figures 3.5 and 3.6.

The remainder of this chapter describes our research [6] on the complexity of finding an optimal decomposition, i.e., one in which the largest subsystem is as small as possible. This problem is not new: It has been studied before for example by Bliek et al. [25]. However, as far as we know, no investigation of its parameterized complexity has been undertaken before. Even though this problem is only relevant for the decomposition of G_3, we will usually simply consider the more general case of a bipartite graph $G = (U, V, E)$ with a maximum matching covering U. In the remainder of this chapter, G_1, G_2 and G_3 will be used to denote other (sub)graphs of G and no longer necessarily correspond to the parts of the coarse Dulmage-Mendelsohn decomposition. The decision problem corresponding to finding a decomposition where the largest subsystem is as small as possible can be stated as follows.

Bounded Block Decomposition
Instance: A bipartite graph $G = (U, V, E)$, an integer k
Question: Is there a partition of $U = U_1 \cup U_2 \cdots \cup U_n$ and $V = V_1 \cup V_2 \cdots \cup V_n \cup V_{n+1}$ such that for each $1 \leq i \leq n$, $G_i = G[U_i, V_i]$ is a bipartite graph with a perfect matching, $\Gamma(V_i) \subseteq \bigcup_{j=1}^{i} U_j$ and $|U_i| = |V_i| \leq k$?

Note that for any yes-instance of this decision problem, we must have $|U| \leq |V|$ and G contains a matching covering all of U. Furthermore, if G is not connected then we may analyze its components separately, so without loss of generality we will assume G is connected. For bipartite graphs

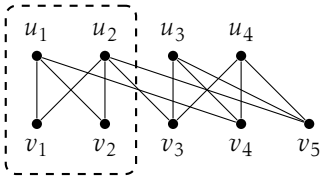


Figure 3.7: An example of a free square block.

$G = (U, V, E)$ with $|U| = |V|$ containing a perfect matching, a partition as described in the definition of the Bounded Block Decomposition problem is equal to the fine Dulmage-Mendelsohn decomposition, so Bounded Block Decomposition on such graphs is decidable in polynomial time. Unfortunately, for $|U| < |V|$ we will see that Bounded Block Decomposition is in general a harder problem.

We proceed by discussing some special properties of the part G_1, as well as the relation between this part, the entire decomposition and how it can be used to solve the corresponding system of equations.

3.6 The Free Square Block Problem

Before analyzing the difficulty of the Bounded Block Decomposition problem itself, we first turn our attention to the problem of finding a single small subsystem that can be solved after solving the remainder of a system of equations. We first introduce the concept of a free square block [25] and show some of its useful mathematical properties. Then we describe the decision problem related to finding the smallest free square block in a given bipartite graph as well as alternative mathematical interpretations of this problem. This forms the basis for the complexity analysis of the entire decomposition problem.

Definition 3.10. Let $G = (U, V, E)$ be a bipartite graph. A *free square block* A of G is a non-empty induced subgraph of G such that $|U_A| = |V_A|$, and $\Gamma_G(V_A) \subseteq U_A$.

Translated back to the application of systems of equations, the last requirement states that no variable in A may occur in an equation which is not part of A, so A can be solved after solving the remainder of the system. Figure 3.7 shows an example of a free square block.

We proceed by proving several useful properties of free square blocks.

Theorem 3.11. *Let $G = (U, V, E)$ be a connected bipartite graph with $1 \leq |U| \leq |V|$. There exists a non-empty induced subgraph $A \subseteq G$ with a perfect*

matching, such that $\Gamma_G(V_A) \subseteq U_A$*, i.e., G contains a free square block with a perfect matching.*

Proof. Consider a minimum vertex cover C of G with $U_C \neq \emptyset$ (the connectedness combined with $|U| \leq |V|$ guarantees such a vertex cover to exist) and a maximum matching M (of equal cardinality by König's theorem, Theorem 2.2). Let $U_A = U_C$ and assume U_A is matched to $V_A \subseteq V$. Now construct the induced subgraph $A = G[U_A \cup V_A]$. By construction, A has a perfect matching. Furthermore, as C is a vertex cover, we must have $\Gamma_G(V_A) \subseteq U_A$. □

It is convenient to introduce the concepts of minimal and minimum free square blocks. A *minimal free square block* $A \subseteq G$ is a free square block of G that contains no smaller free square block. Similarly, a *minimum free square block* A of G is a free square block of minimum size among the free square blocks of G.

Corollary 3.12. *A minimal free square block has a perfect matching.*

Proof. Assume to the contrary that A is a minimal free square block of $G = (U, V, E)$ that contains no perfect matching. Then by Hall's theorem (Theorem 2.3) there is a strict subset $V' \subset V_A$ such that $|\Gamma(V')| < |V'|$. However, in that case we know from Theorem 3.11 that $G[\Gamma(V') \cup V']$ contains a smaller free square block that is necessarily also a free square block of G, contradicting the minimality of A. □

Minimal free square blocks are of interest for solving systems of equations: They correspond to subsystems that cannot be decomposed further, can be solved after solving the rest of the system, and are consistent due to the existence of a perfect matching. Bliek et al. [25] have described an algorithm called OpenPlan to decompose systems of equations using free square blocks. This algorithm finds a smallest (w.r.t. the number of vertices) free square block in the bipartite graph representation of a system of equations and marks it as a subsystem that can be solved last. By iteratively applying this procedure until only variables are left, the algorithm comes up with an optimal decomposition, i.e., one in which the size of the largest subsystem is as small as possible. As finding the smallest free square block in a graph forms the core of this algorithm, we decided to further investigate the tractability of this problem, as well as that of the decomposition problem itself. In the analysis of the following sections, we

consider the following natural parameterization of the decision problem regarding the existence of free square blocks of a given size.

k-Free Square Block
Instance: A bipartite graph $G = (U, V, E)$, a positive integer k
Parameter: k
Question: Does G contain a free square block of size k?

Although our analysis is based on graph theory, this problem can also naturally be expressed in terms of hypergraphs or systems of distinct representatives. The following problem regarding hypergraphs (see e.g. [10]) is equivalent to the k-Free Square Block problem:

Instance: A hypergraph $H = (V, \mathcal{E})$, a positive integer k
Parameter: k
Question: Is there a subgraph $H' \subseteq H$ with $|V(H')| = |\mathcal{E}(H')| = k$?

Another formulation uses the notion of a *system of distinct representatives* (see e.g. [13, 26]). Let $\mathcal{F} = (S_1, \ldots, S_n)$ be a family of subsets of a finite set S, a sequence $F = (f_1, \ldots, f_n)$ is called a *system of distinct representatives*, or *SDR*, if all elements of F are distinct, and $f_i \in S_i$ for $i = 1, 2, \ldots, n$. In this context, the k-Free Square Block problem is equivalent to the following decision problem:

Instance: A set S, a family $\mathcal{F}$ of subsets of S, a positive integer k
Parameter: k
Question: Is there a subset $S' \subseteq S$ and a subset $\mathcal{F}' \subseteq \mathcal{F}$, such that $|S'| = |\mathcal{F}'| = k$ and $\bigcup_{F \in \mathcal{F}'} F = S'$ and $\mathcal{F}'$ has an SDR with respect to S'?

3.7 k-Free Square Block is $W[1]$-complete

We proceed by studying the k-Free Square Block problem to establish its parameterized complexity. The main results are two proofs by reduction that together establish the $W[1]$-completeness of the problem. The Dulmage-Mendelsohn decomposition and related problems have been studied before from a parameterized complexity point of view, for example in the context of variations on vertex cover problems (see e.g. [27, 28]), but the parameterized approach to the specific problems we study seems to be new.

Bliek et al. note that the *smallest free block problem* is expected to be NP-hard [25]. Later, the problem is stated to be NP-hard [29] as being the 'dual' of the minimum dense problem which asks for a minimum subgraph with at least a certain ratio between edges and vertices (see e.g. [30]), however, this duality is not immediately obvious. Furthermore, NP-completeness

is not always the end of the line, as parameterized versions of (decision) problems can sometimes be solved efficiently even though their non-parameterized versions are NP-hard. A nice example of this is given in the introductory chapter of [5] that discusses the (minimum) Vertex Cover problem which is known to be NP-complete and its parameterized version k-Vertex Cover that asks if a vertex cover of size k exists. The latter version is *fixed parameter tractable*, i.e., can be solved in time $O(f(k)\text{poly}(n))$ (where n denotes the number of vertices). So the question in our case is: Is there an efficient parameterized algorithm to find a small minimal free square block of parameterized (maximum) size? In this section, we show the k-Free Square Block problem is complete for the $W[1]$-class of decision problems. The proof of $W[1]$-hardness also shows NP-hardness and is based on a reduction from the $W[1]$-hard problem k-Clique (see e.g. [31]):

k-Clique
Instance: A graph $G = (V, E)$, a positive integer k
Parameter: k
Question: Is there a set of k vertices $V' \subseteq V$ that forms a complete subgraph of G (that is, a clique of size k)?

This result, formalized in Theorem 3.13, intuitively means that even for a small fixed value of k (the size of the free square block), the problem is not tractable, i.e., solvable in time polynomial in the size of the graph.

Theorem 3.13. *k-Free Square Block is $W[1]$-hard.*

Proof. The proof of $W[1]$-hardness is accomplished by showing how an arbitrary instance (G, k) of k-Clique can be converted in polynomial time into an instance (G', k') of k-Free Square Block in such a way that the latter is a yes-instance of k-Free Square Block if and only if the former is a yes-instance of k-Clique (a uniform reduction in the sense of [32]). We only prove this for odd values of k; any instance of k-Clique with k even can easily be converted into an equivalent instance of $(k+1)$-Clique by simply adding one extra vertex and connecting it to all the other vertices.

Let (G, k) with $G = (V, E)$ be an instance of k-Clique with k odd and construct a bipartite graph $G' = (U', V', E')$ as follows: Let V' contain one vertex for each of the edges in E. Let U' contain $\frac{k-1}{2}$ copies of each of the vertices in V. And let E' contain an edge between $u' \in U'$ and $v' \in V'$ if and only if the edge of G corresponding to v' is incident to the vertex corresponding to u'. (As each of the vertices of G is duplicated $\frac{k-1}{2}$ times, this means every $v' \in V'$ has degree $k-1$.)

Free square blocks of G' correspond to (dense) subgraphs of G that contain $\frac{k-1}{2}$ times as many edges as vertices. The smallest (in terms of vertices) possible subgraph of G with this ratio is a k-clique, so G' contains a free

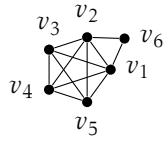


Figure 3.8: Example graph containing a 5-clique.

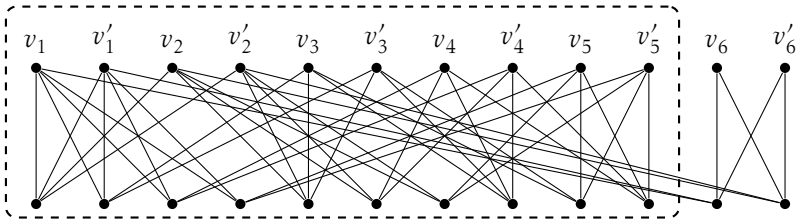


Figure 3.9: Free square block of size 10 corresponding to a 5-clique.

square block of size $k' = \frac{k(k-1)}{2}$ if and only if G contains a clique of size k; smaller free square blocks of G' can never exist as G cannot contain smaller subgraphs with this ratio.

We have thus created an instance (G', k') of k-Free Square Block that is a yes-instance if and only if the original (G, k) formed a yes-instance of k-Clique, proving k-Free Square Block to be $W[1]$-hard. □

As a free square block of size k' in the constructed instance of the k-Free Square Block problem has to be minimum if it exists, this also shows that the k-Minimum Free Square Block problem determining whether a bipartite graph contains a minimum free square block of size k is $W[1]$-hard.

As an example of the construction of G' in this proof, consider the graph G shown in Figure 3.8. Clearly G contains a 5-clique. We now construct the corresponding bipartite graph G' according to the procedure outlined in the proof of Theorem 3.13 as shown in Figure 3.9 (every vertex in the original graph is duplicated). The free square block corresponding to the 5-clique is clearly recognizable in the bipartite graph G'.

Remark 3.14. By Corollary 3.12 we know any free square block of size k' in G' has a perfect matching. This can also be seen by observing the following: Consider an Euler walk $v_1 e_1 v_2 e_2 \dots v_1$ in a k-clique on an odd number of vertices. Such a tour contains every e_i exactly once and every v_i exactly $\frac{k-1}{2}$ times. The edges in G' corresponding to $v_1 e_1$, $v_2 e_2$ etc. together lead

to a perfect matching in the free square block of G' corresponding to the k-clique.

After establishing the $W[1]$-hardness of the k-Free Square Block problem, in essence providing a lower bound on its difficulty, we now proceed to show also membership in $W[1]$. For the proof, we will use a reduction to the parameterized decision problem t-Threshold Stable Set known to be $W[1]$-complete [5].

t-Threshold Stable Set
Instance: A directed graph $G = (V, A)$, a positive integer k
Parameter: k
Question: Does G have a t-threshold stable set (cf. below) of size k?

A *t-threshold stable set* is a set of vertices $S \subseteq V$ such that, with some fixed t, for every vertex v of $V \setminus S$, there are fewer than t vertices $u \in S$ with $uv \in A$.

For our purpose we will only use $t = 1$, effectively reducing the problem to the following:

1-Threshold Stable Set
Instance: A directed graph $G = (V, A)$, a positive integer k
Parameter: k
Question: Is there a subset $S \subseteq V$ of size k such that $\Gamma(S) \subseteq S$

A useful property of a 1-Threshold Stable Set S is that for a strongly connected component $C \subseteq G$ we have either $C \subseteq G[S]$ or $C \cap S = \emptyset$. Using this, we can prove the following theorem:

Theorem 3.15. *k-Free Square Block is $W[1]$-complete.*

Proof. We construct a uniform reduction from k-Free Square Block to 1-Threshold Stable Set. Let p and q be two distinct prime numbers each greater than k. Given an instance $G = (U, V, E)$ of k-Free Square Block, we construct a directed graph G' that is an instance of 1-Threshold Stable Set (with parameter value k') as follows: First direct all edges from V to U. Then replace each vertex of U by a strongly connected component on p vertices, for example a p-cycle. Replace each vertex of V by a strongly connected component on q vertices (e.g., a q-cycle). The directed graph G' that is obtained contains a 1-threshold stable set of size $k' = kp + kq$ if and only if G contains a free square block of size k. This can be verified as follows: A free square block $A = (U', V', E') \subseteq G$ of size k has $|U'| = |V'| = k$. The union of U', V' and all of the vertices in their strongly connected components form a 1-threshold stable set of size $kp + kq$ as there are no outgoing arcs from this set to the rest of G'. Conversely, if we can find

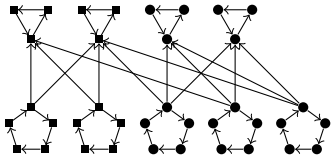


Figure 3.10: 1-Threshold stable set instance, stable set denoted by square vertices ($k = 2$; $p = 3$, $q = 5$, $k' = 16$).

a 1-threshold stable set S of size $kp + kq$ in G', then $|S \cap U| = |S \cap V| = k$ and there are no outgoing arcs from $S \cap V$ to $U \setminus (S \cap U)$, showing that $G[S \cap (U \cup V)]$ is in effect a free square block of size k. □

Figure 3.10 shows an example of this construction for a bipartite graph with a free square block of size $k = 2$.

Finally, we show that requiring G to have a (maximum) matching covering U, as is likely the case in consistent systems of equations, does not make the problem easier. To this end, construct a bipartite graph $G' = (U \cup U', V \cup V', E \cup E')$ where V' contains $|U| + 1$ additional vertices, U' contains a single new vertex, and $E' = \{uv \mid u \in U \cup U', v \in V'\}$, i.e., we add $|U| + 1$ vertices to V and connect each of them to all vertices in U as well as to a single new vertex. By this construction, G' contains $K_{|U|+1,|U|+1}$ (a complete bipartite graph with $|U|+1$ vertices in each of its classes) as a subgraph, so G' also contains a perfect matching covering $U \cup U'$. However, this construction does not add any new free square blocks of size smaller than $|U| + 1$. Passing from G to G' if necessary shows that the above decision problem remains $W[1]$-complete if restricted to instances where U is covered by a maximum matching.

3.8 Bounded Block Decomposition is $W[1]$-hard

After analyzing the complexity of the k-Free Square Block problem we turn our attention to the complexity of finding an entire decomposition of a bipartite graph. We consider the natural parameterization of the Bounded

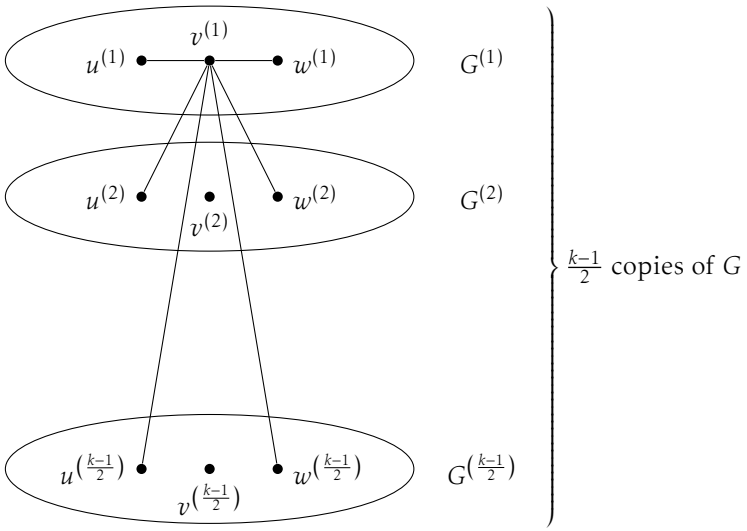


Figure 3.11: Construction of $\bar{G}$ (only edges incident to $v^{(1)}$ are shown).

Block Decomposition problem:

Bounded Block Decomposition
Instance: A bipartite graph $G = (U, V, E)$, an integer k
Parameter: k
Question: Is there a partition of $U = U_1 \cup U_2 \cdots \cup U_n$ and $V = V_1 \cup V_2 \cdots \cup V_n \cup V_{n+1}$ such that for each $1 \leq i \leq n$, $G_i = G[V_i \cup U_i]$ is a bipartite graph with a perfect matching, $\Gamma(V_i) \subseteq \bigcup_{j=1}^{i} U_j$ and $|U_i| = |V_i| \leq k$?

The main result of this chapter is the next theorem.

Theorem 3.16. Bounded Block Decomposition *is* $W[1]$*-hard.*

For the proof of this theorem, we first require a construction procedure. Given a graph $G = (V, E)$ and an odd k with $k > 1$ consider the graph $\bar{G} = \left(\bar{V}, \bar{E}\right)$ consisting of $\frac{k-1}{2}$ copies $(G^{(1)}, G^{(2)}, \ldots, G^{(\frac{k-1}{2})})$ of G with edges between all pairs of vertices $u^{(i)}$ and $v^{(j)}$ iff $uv \in E$. So a vertex $v^{(i)}$ is adjacent to its neighbors in $G^{(i)}$ as well as to all the copies of its neighbors (see also Figure 3.11). Clearly $\Gamma\left(v^{(i)}\right) = \Gamma\left(v^{(j)}\right)$ holds for any two copies $v^{(i)}$ and $v^{(j)}$ of the same vertex $v \in V$. Using this construction we can prove Theorem 3.16.

Proof of Theorem 3.16. The proof consists of a reduction from k-Clique. Let $G = (V, E)$ and k be an instance of k-Clique. To avoid a few corner-cases in the reduction, we assume G is connected, $|E| \geq |V|$, $k > 1$ and k is odd. We start by constructing the graph $\bar{G} = (\bar{V}, \bar{E})$ using G and k. This construction can clearly be performed in polynomial time. First, we claim that $\bar{G}$ contains a clique of size k iff G contains a clique of size k. Clearly, any clique of G has $\frac{k-1}{2}$ corresponding copies in $\bar{G}$. Conversely, if $\bar{S} \subset \bar{V}$ induces a clique in $\bar{G}$, then at most a single copy of any vertex $v \in V$ can be in $\bar{S}$. Replacing every vertex $v^{(i)} \in \bar{S}$ by $v^{(1)}$ (its copy in $G^{(1)}$), we obtain a clique in $G^{(1)}$, which corresponds to a clique in G. We have thus shown that $(\bar{G}, k)$ and (G, k) are equivalent as instances of k-Clique. Using $\bar{G}$ we now construct a bipartite graph $G' = (U', V', E')$ as in the proof of Theorem 3.13: U' contains $\frac{k-1}{2}$ copies of each vertex in $\bar{V}$, V' is equal to $\bar{E}$, and E' contains an edge between $u' \in U'$ and $v' \in V'$ iff the edge corresponding to v' in $\bar{G}$ is incident to the vertex in $\bar{G}$ that u' corresponds to. Free square blocks of G' correspond to subgraphs of $\bar{G}$ that contain $\frac{k-1}{2}$ times more edges than vertices. The smallest (in terms of vertices) possible subgraph of G with this ratio is a k-Clique, so G' contains a free square block of size $\binom{k}{2}$ if and only if $\bar{G}$ contains a clique of size k; smaller free square blocks of G' can never exist as $\bar{G}$ cannot contain smaller subgraphs with this ratio. So if G' has a Bounded Block Decomposition with block size bounded by $\binom{k}{2}$, then the first free square block in this decomposition must correspond to a k-clique in $\bar{G}$ and thus to a k-clique in G. For the converse, assume G, and thus $\bar{G}$, contains a k-clique; pick one such clique. By construction, G' contains $\frac{k-1}{2}$ disjoint free square blocks of size $\binom{k}{2}$ corresponding to this clique in G. After removing these blocks from G', the remainder of G' can be decomposed into free square blocks of size $\frac{k-1}{2}$ as follows: Pick a vertex $v^{(1)} \in \bar{V}$ that is not yet part of the decomposition and has a neighbor $w^{(1)} \in \bar{V}$ that is already part of the decomposition. The $\frac{k-1}{2}$ copies of $v^{(1)}$ in U' and the vertices in V' corresponding to the $\frac{k-1}{2}$ edges connecting $v^{(1)}$ to the copies $w^{(i)}$ of $w^{(1)}$ in $\bar{G}$ together form a free square block of size $\frac{k-1}{2}$ in the remainder of G'. Remove this free square block from G' and repeat this operation for $v^{(2)} \dots v^{(\frac{k-1}{2})}$ in $\bar{G}$. Keep constructing free square blocks in this way until all vertices in U' are part of a decomposition. Due to the connectedness of $\bar{G}$, we can keep picking vertices to induce the next block until the decomposition is complete. All blocks in this decomposition have a size bounded by $\binom{k}{2}$. By transforming an instance (G, k) of k-Clique to a corresponding instance $(G', \binom{k}{2})$ of Bounded Block Decomposition, we have shown a uniform reduction from k-Clique to Bounded Block Decomposition, proving that Bounded Block Decomposition is $W[1]$-hard. □

Theorem 3.16 shows that obtaining an optimal decomposition – one where the largest block is as small as possible – is a hard problem even if we

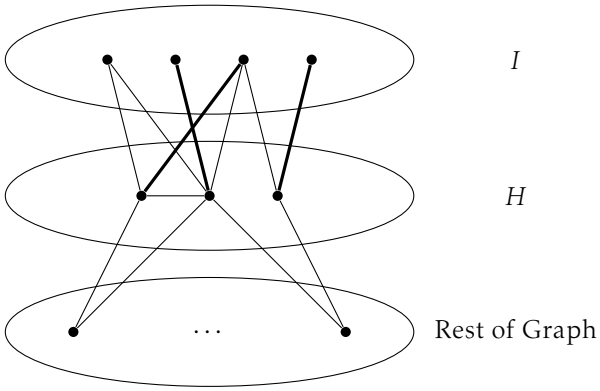


Figure 3.12: An example crown structure.

parameterize the size of the largest block. Before discussing the practical implications of this, we will first digress slightly by describing how our results on free square blocks have enabled us to settle two open questions regarding another problem in parameterized complexity.

3.9 Relation to Crowns

It was pointed out to us by a reviewer of the first draft of our paper on free square block problems [6] that they seem closely related to a special case of *crown structures*, a reduction mechanism for kernelization of the parameterized Vertex Cover problem [33, 34]. In this section we briefly discuss crown structures and their relation to the free square block problems we have considered in the preceding sections. After showing the relation between the two, we use our results on free square blocks to settle two open questions regarding crowns.

We start by giving a few definitions, adapted from [35]: A *crown* is an ordered pair (I, H) of subsets of vertices from a graph G that satisfies the following criteria: (1) $I \neq \emptyset$ is an independent set of G, (2) $H = \Gamma(I)$, and (3) there exists a matching M on the edges connecting I and H such that all vertices of H are covered. This implies $|I| \geq |H|$. H is called the *head* of the crown. An example of a crown is shown in Figure 3.12. A crown (I, H) is called a *straight crown* if it satisfies the condition $|I| = |H|$, otherwise it is called a *flared crown*. A crown that is contained in another crown is called a *subcrown*. The *size* or *order* of a crown is the number of vertices in $I \cup H$. (Note how this definition of the size of a crown is different from our previous definition of the size of a square block!)

Crowns are used to reduce the size of a problem instance of the Vertex Cover problem by exploiting the fact that if G is a graph with a crown (I, H), then there is a vertex cover of G of minimum size that contains all the vertices in H and none of the vertices in I [35]. By applying this reduction rule, a smaller instance of Vertex Cover can be solved instead of the original instance. It has been shown that finding a non-trivial crown in a graph G can be done in polynomial time. Finding a crown of maximum order is also polynomially solvable [35].

The remainder of this section is dedicated to establishing the $W[1]$-hardness of two parameterized decision problems related to crowns. The first problem we consider is the natural parameterization of the Sized Crown problem previously proven to be NP-complete by Sloper [36]. This decision problem involves determining if a graph contains a crown of a certain size.

Sized Crown
Instance: A graph $G = (V, E)$, a positive integer k
Parameter: k
Question: Does G contain a crown (I, H) with $|I \cup H| = k$?

The second problem we consider is the parameterized decision problem Minimum Crown, regarding the identification of a crown of minimum order [35].

Minimum Crown
Instance: A graph $G = (V, E)$, a positive integer k
Parameter: k
Question: Does G contain a minimum crown (I, H) with $|I \cup H| = k$?

The parameterized complexity of these problems is mentioned as an open problem by respectively Sloper [36] and Abu-Khzam et al. [35] and to our knowledge these problems have not been solved before.

To facilitate our discussion, we define a few more terms: We call a crown a *minimal crown* if it contains no smaller subcrown. A crown with the minimum number of vertices over all crowns is called a *minimum crown*. The following useful lemma enables us to consider only straight crowns if we search for crowns of minimum order as it implies that *minimal* crowns have to be straight [35].

Lemma 3.17 ([35]). *If (I, H) is a flared crown then there is another crown (I', H) that is straight and $I' \subset I$.*

We start our analysis by establishing a few additional characteristics of crowns in bipartite graphs and their relationship to free square blocks.

Lemma 3.18. *If (I,H) is a minimal crown of a bipartite graph $G=(U,V,E)$ then either $H \subseteq U$ or $H \subseteq V$.*

Proof. Assume to the contrary that H contains vertices from both U and V. Due to the existence of a perfect matching between I and H, we know that I also contains vertices from both U and V. Now let $H_U = H \cap U$ and $I_V = I \cap V$. Clearly, I_V is an independent set of G, $H_U = \Gamma(I_V)$, and there exists a perfect matching between H_U and I_V. So (I_V, H_U) is a strict subcrown of (I,H) contradicting the minimality of (I,H). □

Lemma 3.19. *The following properties define the relation between minimal crowns and free square blocks in a bipartite graph $G=(U,V,E)$:*

(i) A minimal free square block A corresponds to a straight crown (V_A, U_A).

(ii) A minimal straight crown (V_A, U_A) induces a free square block A of G.

(iii) The number of vertices in a minimum free square block is equal to the number of vertices of the smallest straight crown (I,H) with $H \subseteq U$.

Proof. (i) As G is a bipartite graph, (1) V_A is an independent set. (2) $U_A = \Gamma(V_A)$ and as a minimal free square block has a perfect matching, we have that (3) there exists a matching M on the edges connecting V_A and U_A such that all vertices of U_A are covered.

(ii) A straight crown by definition has $|V_A| = |U_A|$ and $U_A = \Gamma(V_A)$, so $A = G[V_A \cup U_A]$ is a free square block of G.

(iii) Immediate from (i) and (ii).

□

We now come to our main result on crowns.

Theorem 3.20. Minimum Crown *is $W[1]$-hard.*

Proof. Let $G=(U,V,E)$ and k be an instance of the k-Minimum Free Square Block problem and construct a new bipartite graph G' as follows: Let $K_{k+1,k+1} = (U^*, V^*, E^*)$ be a complete bipartite graph with $k+1$ vertices in each of its vertex classes. Construct $G' = (U', V', E')$ as $U' = U \cup U^*$, $V' = V \cup V^*$ and $E' = E \cup E^* \cup \{uv \mid u \in U, v \in V^*\}$. In other words, G' consists of G and $K_{k+1,k+1}$ and an edge between every pair (u,v) with $u \in U$ and $v \in V^*$.

Clearly, any crown (I,H) of G' with $I \cap (U^* \cup V^*) \neq \emptyset$ must have $|\Gamma(I)| = |H| \geq k+1$. Furthermore, due to its construction, any minimal crown (I,H) in G' with $H \subseteq U'$ must have $V^* \subseteq I$ and thus $|I| \geq k+1$. So this construction introduces no new crowns of size k or less in G'.

G' contains a minimum crown of size k if and only if G contains a minimum crown of size $2k$ with its head in U. Such crowns correspond exactly to minimum free square blocks of G. As the k-Minimum Free Square Block problem is $W[1]$-hard, and the above construction is a uniform reduction in the sense of [32], Minimum Crown is also $W[1]$-hard. □

As an immediate consequence Sized Crown is $W[1]$-hard as well.

3.10 Concluding Remarks

In this chapter we have described how the bipartite graph representation of a system of (non-linear) equations can be used to decompose such a system. Using the Dulmage-Mendelsohn decomposition of a bipartite graph we can determine the under-, well- and over-constrained parts of the system as well as split the well-constrained part into irreducible subsystems in a canonical way. Based on this, Bliek et al. have shown how free square blocks can be used to decompose the under-specified part of a system into irreducible blocks as well, however, this decomposition is no longer unique. They have also shown how finding an optimal decomposition, i.e., one in which the largest remaining block is as small as possible, can be achieved by iteratively finding a free square block of minimum size.

Although it has been shown before that the problem of finding minimum free square blocks was NP-hard, our analysis also establishes its parameterized complexity as $W[1]$-complete. This implies that under the working hypothesis that $FPT \neq W[1]$ (see [37]), it is even intractable to determine whether a free square block exists bounded by some fixed size.

Furthermore, we have also shown that the entire decomposition problem Bounded Block Decomposition is also $W[1]$-hard. This shows that the hardness of the free square block problem can most likely not simply be mitigated by resorting to another approach for obtaining an optimal decomposition.

Chapter 4

Pivoting and Bisimplicial Edges

Gaussian pivoting is a common operation on matrices and in particular on matrices representing systems of linear equations. In processes involving pivots we often have a set of candidate elements, of which we can select one to use as pivot. Common selection criteria are the likeliness of preserving numerical stability during the remainder of the process (in case of Gaussian elimination) or rate of progress towards a goal (in case of the Simplex method). In this chapter we focus on another criterion: preserving sparsity of the matrix. The chapter starts with an introduction into Gaussian pivots that preserve sparsity and their relation to bisimplicial edges in bipartite graphs. After describing previously known algorithms for finding bisimplicial edges in bipartite graphs, we introduce a new, simple algorithm based on counting arguments. We subsequently show that our new algorithm has the same time complexity for sparse matrices as the best known algorithm. Furthermore, it also has a significantly smaller expected running time on a natural class of random dense matrices.

4.1 Gaussian Pivots

Gaussian elimination is the classic algorithm for solving systems of linear equations. It has been known for centuries, but is still in use today. The core of the algorithm consists of using elementary row operations to reduce a matrix to triangular form. To simplify both notation and discussion in this chapter we only consider square matrices. We denote by n the number of rows or columns of these matrices and by m the number of non-zero elements. Furthermore, we assume each row or column contains at least one non-zero element. During each round of the elimination process, a non-

zero element of the remaining matrix, the *pivot*, is picked. The row and column of the pivot element are called the *pivot row* and *pivot column* respectively. Using elementary row operations, all other non-zero elements of the pivot column are *cleared* by subtracting a multiple of the pivot row from their rows.

More formally: If M is a real-valued matrix and $M_{k,l} \neq 0$ is used as a Gaussian pivot, the elements of the matrix M' we obtain after pivoting on (k,l) are given by (4.1).

$$M'_{i,j} = \begin{cases} M_{i,j} - \frac{M_{i,l}}{M_{k,l}} M_{k,j} & \text{if } i \neq k \text{ and } M_{i,l} \neq 0 \\ \frac{M_{i,j}}{M_{k,l}} & \text{if } i = k \\ M_{i,j} & \text{otherwise} \end{cases} \tag{4.1}$$

All elements of M with $i \neq k$ and $j = l$ are turned into zeroes in M'. Unfortunately, other elements of M that are zero may be turned into non-zeroes in M' in columns other than the pivot column. This phenomenon is called *fill-in*. When working with sparse matrices, avoiding fill-in can be very beneficial to preserve the reduced space requirements of most sparse matrix representations. The remainder of this chapter focuses on the selection of Gaussian pivots that preserve sparsity.

As our focus in this thesis is on structural rather than numerical analysis, we assume that subtracting a multiple of the pivot row k from another row i with a non-zero value in the pivot column l only turns the element in the pivot column into a zero. In other words: Only the elements of the pivot column that we want to clear are turned into zeroes and no 'accidental cancelations' take place in other columns. Clearly, this holds for almost all real-valued matrices. Under this regularity-assumption the actual values in the matrix are no longer of importance; only the structure of the matrix with respect to zero and non-zero values is relevant to our analysis.

To represent the sparsity structure of the matrix, we pass from a real-valued matrix to a $\{0,1\}$-matrix containing a 1 for exactly those elements that have a non-zero value in the original matrix. Under our assumption changes to the sparsity structure of the original matrix due to Gaussian pivots can now be simply modeled using zeroes and ones. Formally, if we have a $\{0,1\}$-matrix M representing the structure of a real-valued matrix and we pivot on the non-zero element (k,l), the elements of the matrix M' we obtain after pivoting are given by (4.2).

$$M'_{i,j} = \begin{cases} 0 & \text{if } j = l \text{ and } i \neq k \text{ and } M_{i,l} = 1 \\ \max\left(M_{i,j}, M_{k,j}\right) & \text{if } j \neq l \text{ and } i \neq k \text{ and } M_{i,l} = 1 \\ M_{i,j} & \text{otherwise} \end{cases} \tag{4.2}$$

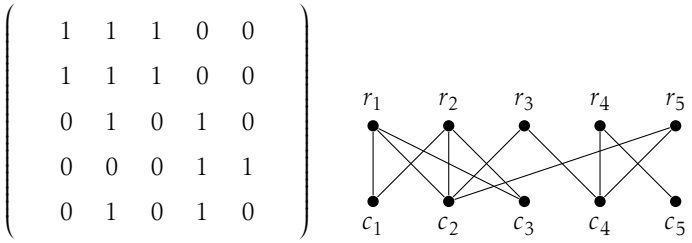


Figure 4.1: An example of a $\{0,1\}$-matrix M and its bipartite graph $G[M]$.

Note how in the $\{0,1\}$-matrix representation the subtraction operation has been replaced by taking the maximum of two matrix elements. In particular the row operation turns a zero in a non-pivot row into a non-zero (causing fill-in) if the element of the pivot row in the corresponding column is a non-zero. Let us introduce the $\leq$ relation on matrix rows. We write $M_{k,*} \leq M_{i,*}$ to indicate that, for any column j of M, $M_{k,j} = 1$ implies $M_{i,j} = 1$. If $M_{k,*} \leq M_{i,*}$ then row $M_{i,*}$ is said to *majorize* row $M_{k,*}$. Clearly every row majorizes itself. In order to avoid fill-in completely when pivoting on an element (k,l), we must have that $M_{k,*} \leq M_{i,*}$ for every row i with $M_{i,l} = 1$.

We may consider the $\{0,1\}$-matrix M as the biadjacency matrix of a bipartite graph $G[M]$ where the rows and columns of the matrix form the two vertex classes of the graph and the non-zero elements correspond to the edges between them. An example of a $\{0,1\}$-matrix and its corresponding bipartite graph is shown in Figure 4.1.

In this chapter we will analyze this graph representation $G[M]$ to find pivots in M that avoid fill-in. The next section describes how a bisimplicial edge in $G[M]$ between i and j corresponds to a pivot $M_{i,j}$ that avoids fill-in and vice versa. This correspondence was first discussed in detail by Golumbic and Goss [16]. Finding bisimplicial edges in $G[M]$ can therefore be useful when we are asked to find pivots avoiding fill-in in M.

After introducing bisimplicial edges, we will derive some additional characteristics that lead to a new algorithm for finding a bisimplicial edge in a bipartite graph. This algorithm is subsequently extended to a new algorithm for finding all bisimplicial edges in the bipartite graph. Finding all bisimplicial edges, instead of just one, can be beneficial in practice where pivot selection may be subject to a number of additional criteria besides preserving sparsity. Not every pivot might for example preserve numerical stability.

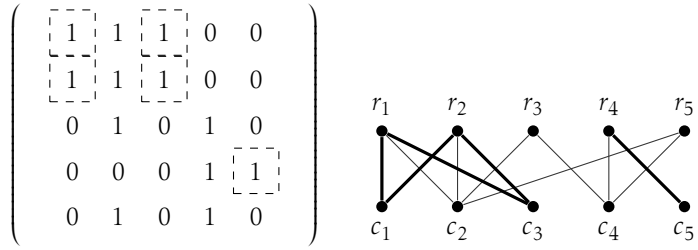


Figure 4.2: Bisimplicial edges in M and its bipartite graph $G[M]$ (bisimplicial edges are bold, the corresponding matrix entries are dashed).

4.2 Bisimplicial Edges

In this section we describe the concept of bisimplicial edges, derive some characteristics of these edges and discuss two existing as well as our new algorithm for finding them.

Definition 4.1. An edge uv of a bipartite graph $G = (U, V, E)$ is called *bisimplicial*, if the subgraph induced by the neighbors of its endpoints $G[\Gamma(u) \cup \Gamma(v)]$ is a complete bipartite graph.

Clearly, we can determine in $O(m)$ time if an edge (u, v) is bisimplicial: If $|\Gamma(u)| \cdot |\Gamma(v)| > m$, then (u, v) cannot be bisimplicial. Else, we check the presence of all edges in $\Gamma(u) \times \Gamma(v)$. So a naive algorithm to find a bisimplicial edge in a bipartite graph G, if one exists, takes $O\left(m^2\right)$ time. The bisimplicial edges in our example matrix M and associated graph $G[M]$ are shown in Figure 4.2.

Goh and Rotem [38] have presented a faster method for finding bisimplicial edges based on the following theorem.

Theorem 4.2 (Goh, Rotem [38]). *Let M be an $n \times n$ $\{0,1\}$-matrix and let $G[M]$ be its corresponding bipartite graph. Let ℓ_i be the number of rows in M that majorize row i and let s_j be the sum of the entries in column j of M. Then $M_{i,j} = 1$ and $\ell_i = s_j$ iff the corresponding edge of $G[M]$ is bisimplicial.*

The values ℓ_i can be easily determined using the matrix $Q = MM^T : \ell_i$ is equal to the number of elements in the row $Q_{i,*}$ that are equal to $Q_{i,i}$ (including $Q_{i,i}$ itself). Once the matrix Q is computed, finding a bisimplicial edge can be done in $O\left(n^2\right)$ operations. Computation of the matrix Q can be performed in either $O(nm)$ or $O(n^\omega)$ depending on the algorithm that is

used (where $\omega \leq 2.376$ is the matrix multiplication exponent [39]). However, fast matrix multiplication, e.g., using the algorithm of Coppersmith and Winograd, has huge hidden constants, which makes it impractical for applications.

The remaining sections of this chapter describe our research [7] on a different approach that has been published recently. Our approach first selects a set of candidate edges. The candidate edges are not necessarily bisimplicial and not all bisimplicial edges are marked as candidates. However, knowing which candidates, if any, are bisimplicial allows us to quickly find all other bisimplicial edges as well. By bounding the number of candidates, we achieve an improved expected running-time. The following observation is the basis of our candidate selection procedure.

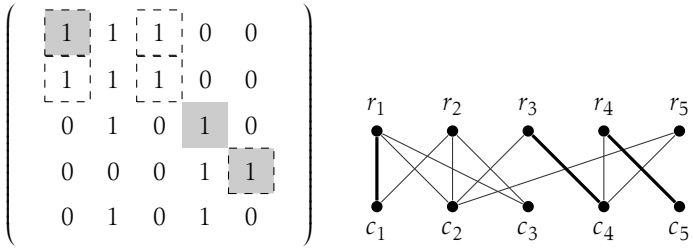
Lemma 4.3. *If an edge uv of a bipartite graph $G = (U, V, E)$ is bisimplicial, we must have $\delta(u) = \min_{u' \in \Gamma(v)} \delta(u')$ and $\delta(v) = \min_{v' \in \Gamma(u)} \delta(v')$.*

Proof. Let $uv \in E$ be a bisimplicial edge, and let $A = G[\Gamma(u) \cup \Gamma(v)]$ be the complete bipartite graph it induces. Now assume that there is a vertex $u' \in U_A$ with $\delta(u') < \delta(u)$. Then there must be a $v' \in V_A$ with $u'v' \notin E_A$. But this would mean A is not a complete bipartite graph, leading to a contradiction. □

Translated to the matrix M, this means that if $M_{i,j} = 1$, it can only correspond to a bisimplicial edge if row i has a minimal number of 1s over all the rows that have a 1 in column j and column j has a minimal number of 1s over all the columns having a 1 in row i. In what follows, we will call the row (column) in M with the minimal number of 1s over all the rows (columns) in M the *smallest* row (column). Using this observation, we construct the following algorithm to pick candidate edges that may be bisimplicial.

Algorithm 4.4. *Perform the following steps:*

1. *Determine the row and column sums for each row i and column j of M.*
2. *Determine for each row i the index c_i of the smallest column among those with $M_{i,c_i} = 1$ (breaking ties by favoring the lowest index); or let $c_i = 0$ if row i has no 1.*
3. *Determine for each column j the index r_j of the smallest row among those with $M_{r_j,j} = 1$ (breaking ties by favoring the lowest index); or let $r_j = 0$ if column j has no 1.*
4. *Mark $M_{i,j}$ as a candidate edge if $c_i = j$ and $r_j = i$.*



(a) (b)

Figure 4.3: Selected candidate edges in M (shaded) and its bipartite graph $G[M]$ (bold).

Clearly, all steps in Algorithm 4.4 can be performed in $O\left(n^2\right)$ time. Furthermore, the last step will mark at most n candidate edges and at least 1. The reason that we have at least one candidate edge is as follows: Let i be the smallest row with the smallest index. Row i will select a column j. Due to the tie-breaking mechanism, column j will also select row i, which leads to a candidate. The candidate edges marked by this algorithm in our example matrix M are shown in Figure 4.3.

The following lemmata establish a few more characteristics of the candidate edges.

Lemma 4.5. *Let i, j, j' be such that the following properties hold:*

1. *$M_{i,j} = 1$ and $M_{i,j'} = 1$ and*
2. *columns j and j' contain an equal number of* 1*s and*
3. *(i, j) is bisimplicial.*

Then (i, j') is also bisimplicial and columns j and j' are identical. Due to symmetry, the same holds if we exchange the roles of rows and columns.

Proof. If columns j and j' are not identical, but contain an equal number of 1s, then there is some row i' such that $M_{i',j} = 1$ and $M_{i',j'} = 0$. In that case (i, j) cannot be bisimplicial, so columns j and j' have to be identical. But then (i, j) and (i, j') both have to be bisimplicial due to symmetry. □

Remark 4.6. Lemma 4.5 can be applied repeatedly across rows and columns: If (i, j) is bisimplicial and $M_{i,j'} = 1$ and $M_{i',j} = 1$ and rows i and i' contain an equal number of 1s and columns j and j' contain an equal number of 1s, then $(i, j'), (i', j)$ and (i', j') are all bisimplicial.

Lemma 4.7. *If (i', j') is bisimplicial, then there are $i \leq i'$ and $j \leq j'$ such that rows i and i' are identical, columns j and j' are identical, and (i, j) is bisimplicial and selected as a candidate by Algorithm 4.4.*

Proof. Let $j \leq j'$ be the column with (1) the lowest index, (2) $M_{i',j} = 1$, and (3) an equal number of 1s to column j'. As (i', j') is bisimplicial, we know three things from Lemma 4.3 and Lemma 4.5: First, (i', j) is also bisimplicial. Second, columns j and j' are identical. Third, columns j and j' are smallest columns in row i'. Due to symmetry, there is also such a row $i \leq i'$ equal to row i' with the lowest index and (i, j') bisimplicial. As (i', j) and (i, j') are bisimplicial, rows i and i' are identical and columns j and j' are identical, also (i, j) must be bisimplicial. Furthermore, by construction, column j must be the smallest column in row i with the lowest index, and row i must be the smallest row in column j with the lowest index. Thus, (i, j) is selected as a candidate. □

Using Algorithm 4.4 as a subroutine, we can construct Algorithm 4.8 below to find all bisimplicial edges of $G[M]$.

Algorithm 4.8. *Perform the following steps:*

1. *Determine candidates using Algorithm 4.4.*
2. *Test each candidate for bisimpliciality.*
3. *For each candidate (i, j) marked as bisimplicial, mark also each (i', j') as bisimplicial for each row i' with an equal number of non-zeroes as row i and $M_{i',j} = 1$ and column j' with an equal number of non-zeroes as column j and $M_{i,j'} = 1$.*

Theorem 4.9. *Algorithm 4.8 finds all bisimplicial edges in time $O\left(n^3\right)$.*

Proof. Step 1 marks up to n candidates in time $O\left(n^2\right)$. Each of these candidates can be checked for bisimpliciality in time $O\left(n^2\right)$, so step 2 can be completed in time $O\left(n^3\right)$. Finally, step 3 marks all non-candidate bisimplicial edges as can be seen from Lemma 4.5 and Lemma 4.7. For a single candidate (i, j) that is found to be bisimplicial, all relevant rows i' and columns j' can be found in time $O(n)$ using the row and column sums computed in the first step of Algorithm 4.4. A total of $O\left(n^2\right)$ additional edges can be marked as bisimplicial during this step and every non-candidate edge is considered at most once. Thus, this step can also be completed in time $O\left(n^2\right)$. □

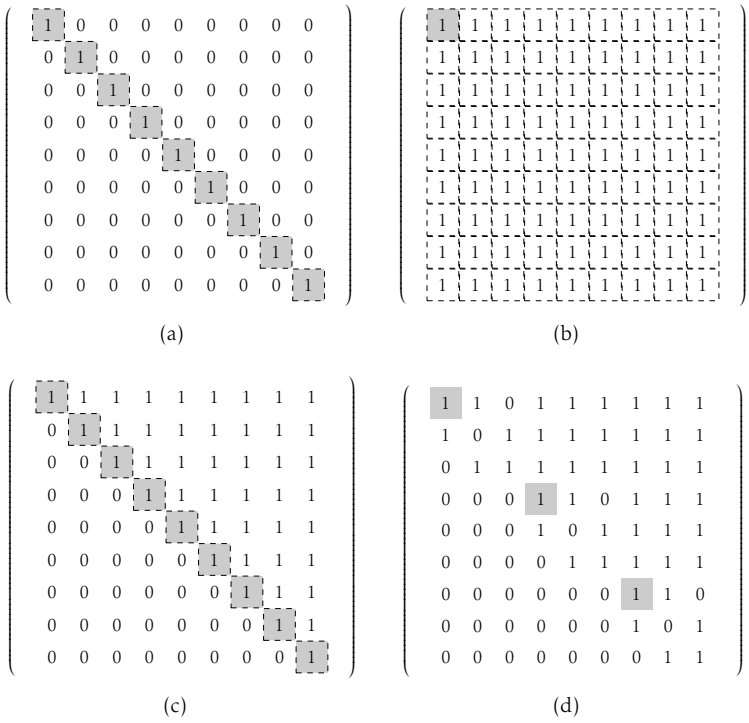


Figure 4.4: Several example matrices with bisimplicial (dashed) and candidate (shaded) elements.

To give a bit more insight into the working of Algorithm 4.8, Figure 4.4 shows several example matrices with their bisimplicial and candidate edges: Figures 4.4(a) and 4.4(c) show situations in which candidates and bisimplicial edges are the same. Figure 4.4(b) illustrates how a single candidate can be used to identify all edges as bisimplicial. Figure 4.4(d) shows how an arbitrarily large matrix can be constructed with $n/3$ candidates and no bisimplicial edges at all.

The running-time of Algorithm 4.8 is dominated by step 2 in which we have to check all candidates in $O\left(n^2\right)$ time each. As we can find up to n candidates, this leads to a worst-case running-time of $O\left(n^3\right)$. In the next section, we present an improved running-time analysis for sparse instances. After that, we show that our algorithm performs significantly better on a common class of random bipartite graphs. The reason for this is that our algorithm will usually select only a few candidate edges.

4.3 Sparse Matrices

Algorithm 4.8 can be implemented such that it makes use of any sparsity in the matrix M. This section describes how a running-time of $O(Cm)$ can be obtained, where C denotes the number of candidates found in the first phase of the algorithm. As $C \leq n$, the running-time is bounded by $O(nm)$. We assume the input matrix M is provided in the form of adjacency lists of the corresponding graph $G[M]$: For every row (column) we have a list of columns (rows) where non-zero elements occur.

The first step of Algorithm 4.8 consists of running Algorithm 4.4, which selects the candidates. Algorithm 4.4 itself consists of three steps. The first step, determining the row and column sums, can be completed in time $O(m)$ by simply traversing the lists. The same holds for the second step: by traversing the adjacency lists the values of c_i and r_j can be determined in time $O(m)$. Constructing the actual set of candidates from these values can subsequently be done in time $O(n)$. In total, Algorithm 4.4 determines the set of candidates in time $O(m)$. After this time, the number C of candidates is known.

Checking a single candidate can be done in time $O(m)$. Thus, the second step of Algorithm 4.8, which consists of checking all candidates for bisimpliciality, can be performed in time $O(Cm)$.

Finally, we analyze the third step of Algorithm 4.8, marking the remainder of the bisimplicial edges. For each bisimplicial candidate (i, j), we have to find all rows i' identical to row i and columns j' identical to column j. Due to Lemma 4.5, we can simply traverse the adjacency lists for row i and column j and check the column and row sums. As every row and every column contains at most one candidate, all adjacency lists are traversed at most once. Thus, this takes at most time $O(m)$ for all candidates together. For each candidate, once all relevant rows i' and columns j' have been determined, we have to mark all combinations (i', j') as bisimplicial. As every edge is considered at most once during this process, this can also be completed in time $O(m)$.

Summarizing, the total running-time of Algorithm 4.8 is $O(Cm)$ where C is bounded from above by n and known in time $O(m)$ after the first phase of the algorithm has been completed.

4.4 A First Bound on the Number of Candidate Edges

In the previous sections we have analyzed the worst-case performance of Algorithm 4.8. However the time required to find all bisimplicial edges in an instance largely depends on the number of candidates that are found during the first phase of the algorithm. We now proceed by showing that for a common model of random bipartite graphs, the $G_{n,n,p}$ model, the

number of candidates is significantly lower than n, both with high probability and in expectation. In this section we derive a logarithmic bound on the expected number of candidates for a fixed value of p in the $G_{n,n,p}$ model. The two subsequent sections further improve this bound on the number of candidates to a constant. This leads to an improved expected running time for our algorithm on such instances.

An instance of the $G_{n,n,p}$ model consists of a bipartite graph $G = (U, V, E)$ with n vertices in each of its the vertex classes U and V. Edges between U and V are drawn independently with probability p from the set of all n^2 possible edges. The corresponding stochastic biadjacency matrix is a $\{0,1\}$-matrix of $n \times n$ elements where each element $M_{i,j}$ is a one with probability p and a zero with probability $1-p$, independent of other elements. In this section we consider random bipartite graphs in the $G_{n,n,p}$ model for a fixed value of $p \in (0,1)$.

Let X_i be the (random) i-th row of M and let $|X_i|$ be the (random) sum of its elements. If we order the X_i vectors according to the number of 1s they contain (favoring the lower index i in case of a tie), we denote by $X_{(1)}$ the row with the least number of 1s, by $X_{(2)}$ the row with the second-to-least number of 1s, and so on.

Lemma 4.10. *Let* $\varepsilon = 2 \cdot \sqrt{\frac{\log n}{pn}}$. *Then*

$$\mathbb{P}\left[\left|X_{(1)}\right| < (1-\varepsilon)pn\right] \leq \frac{1}{n}.$$

Proof. For each $i \in \{1, \ldots, n\}$ we have by Chernoff's bound [40]

$$\mathbb{P}\left[|X_i| < (1-\varepsilon)pn\right] < e^{-np\varepsilon^2/2} = e^{-2\log n} = \frac{1}{n^2}.$$

By a union bound over all rows, we get

$$\mathbb{P}\left[\left|X_{(1)}\right| < (1-\varepsilon)pn\right] \leq n\mathbb{P}\left[X_i < (1-\varepsilon)pn\right] \leq \frac{1}{n}.$$

□

Lemma 4.11. *For* $k = o\left(\sqrt{n}/\log n\right)$, *we have*

$$\mathbb{P}[C > k] \leq (1 + o(1)) \cdot n(1-p)^k + \frac{1}{n}.$$

Proof. Choose $\varepsilon = 2\sqrt{\frac{\log n}{pn}}$ as in Lemma 4.10. We have for each column j

$$\begin{aligned}&\mathbb{P}\left[\text{Column } j \text{ has no 1 in rows } X_{(1)}, \ldots, X_{(k)} \mid \left|X_{(1)}\right| \geq (1-\varepsilon)pn\right] \\ &\leq (1-(1-\varepsilon)p)^k = (1-p+\varepsilon p)^k.\end{aligned}$$

So by again using a union bound, the probability that in this case, any column does not have a 1 in the k smallest rows is bounded from above by

$$\begin{aligned}&\mathbb{P}\left[\exists j : \text{Column } j \text{ has no 1 in rows } X_{(1)},\dots,X_{(k)} \mid \left|X_{(1)}\right| \geq (1-\varepsilon)pn\right]\\ &\leq n(1-p+\varepsilon p)^k\,.\end{aligned}$$

If all columns have at least one 1 in rows $X_{(1)},\dots,X_{(k)}$, all candidates selected by Algorithm 4.4 must be among these k rows, as they contain the smallest number of 1s over all the rows in M. Since each row from $X_{(1)},\dots,X_{(k)}$ contributes at most 1 candidate, Algorithm 4.4 selects at most k candidates in this case.

By Lemma 4.10, the probability that $\left|X_{(1)}\right| < (1-\varepsilon)pn$, i.e., the smallest row contains too few 1s so the case outlined above does not hold, is bounded from above by $1/n$. Combining these two cases, we get

$$\begin{aligned}\mathbb{P}[C > k] &\leq n(1-p+\varepsilon p)^k + \frac{1}{n}\\ &\leq n\left((1-p)^k + \sum_{i=1}^{k}\binom{k}{i}(\varepsilon p)^i(1-p)^{k-i}\right) + \frac{1}{n}\\ &\leq n\left((1-p)^k + (1-p)^k\cdot\sum_{i=1}^{k}\left(\frac{k\varepsilon p}{1-p}\right)^i\right) + \frac{1}{n}\\ &\leq n(1-p)^k\left(1 + \sum_{i=1}^{k}\left(\frac{k\varepsilon p}{1-p}\right)^i\right) + \frac{1}{n}\,.\end{aligned}$$

By our choice of ε and k, we have that $\varepsilon = o(1)$ and $k = o(1)$. The lemma now follows because $\frac{k\varepsilon p}{1-p} = o(1)$ and thus $\sum_{i=1}^{k}\left(\frac{k\varepsilon p}{1-p}\right)^i = o(1)$. □

From Lemma 4.11 we know that with high probability at most $O(\log n)$ candidates are selected: If k is not $O(\log n)$, then the term $n(1-p)^k$ dominates the first part of the expression above and diminishes the probability that $C > k$. This bound also holds in expectation: We use $k = 2\log_{(1-p)}\frac{1}{n} = 2\log_{1/(1-p)} n$. If the number of candidate edges exceeds k, then we use the worst-case bound of n. This gives us

$$\begin{aligned}\mathbb{E}[C] &\leq k + n\mathbb{P}[C > k]\\ &\leq k + n^2(1+o(1))(1-p)^k + 1\\ &\leq k + 2 + o(1)\\ &\leq (2+o(1))\log_{1/(1-p)} n\end{aligned}$$

and also leads to the following theorem and corollary.

Theorem 4.12. *Fix $p \in (0,1)$ and consider random instances in the $G_{n,n,p}$ model. With a probability of $1 - O\left(\frac{1}{n}\right)$ and in expectation, Algorithm 4.4 selects at most $(2 + o(1)) \log_{1-p} \frac{1}{n}$ candidates.*

Corollary 4.13. *For any fixed $p \in (0,1)$, Algorithm 4.8 has an expected running-time of $O\left(n^2 \log_{1/(1-p)} n\right)$ on instances drawn according to $G_{n,n,p}$.*

4.5 Isolating Lemma for Binomial Distributions

The previous section has shown that the expected running-time for our new algorithm is far better than the worst case of $O\left(n^3\right)$. In this section we further improve the expected running-time.

The tie-breaking of Algorithm 4.4 always chooses the row or column with the lowest index. Thus, the probability of the event that a nonzero element (i, j) of M becomes a candidate edge depends also on the number of rows (or columns) that actually have the minimum number of 1s among the rows in column j (or row i).

In this section we analyze the number of rows (or columns) that attain the minimum number of 1s. At first glance, one might argue as follows: The number of 1s per row are independent random variables with binomial distribution. Thus, according to Chernoff's bound, the number of 1s in each row is $np \pm O(\sqrt{n})$ with high probability (see also Lemma 4.10). Hence, we have roughly np random variables that assume values in an interval of size roughly $O\left(\sqrt{n}\right)$. From this, we would expect that the minimum is assumed by roughly $O\left(\sqrt{n}\right)$ random variables.

This bound is not very helpful to us, but fortunately it is also far too pessimistic. It turns out that, although relatively many random variables fall into a relatively small interval, the minimum itself is usually unique: Below we will show that the probability that the minimum is unique is $1 - o(1)$. This resembles the famous isolating lemma [41]. On top of that we also show that the expected number of random variables that assume the minimum value is $1 + o(1)$. The following lemma forms the basis of our proof and captures most of the intuition.

Lemma 4.14. *Let $k \in \mathbb{N}$, and let $X_1, \ldots, X_k$ be independent and identically distributed random variables with values in $\mathbb{Z}$. Let $Y = \min\{X_1, \ldots, X_k\}$, and let $Z = |\{i \mid X_i = Y\}|$ be the number of random variables that assume the minimum value. Let $t \in \mathbb{Z}$, $q \in (0,1)$, and $c \in (0,1)$ such that the following properties hold:*

1. *$\mathbb{P}[X_i \leq t] \leq q$ for any $i \in \{1, \ldots, k\}$.*
2. *For every $s > t$, we have $\mathbb{P}[X_i = s \mid X_i \leq s] \leq c$.*

Then

$$\mathbb{E}[Z] \leq \frac{1}{1-c} + k^2 q\,.$$

Proof. By a union bound over the k events $X_i \leq t$, we have

$$\mathbb{P}[Y \leq t] \leq kq\,.$$

If indeed $Y \leq t$ then we use the trivial upper bound of $Z \leq k$. This contributes the term $k^2 q$. Otherwise, we consider $X_1, X_2, \ldots, X_k$ one after the other. Let $Y_i = \min\{X_1, \ldots, X_i\}$. Let $Y_0 = \infty$ for consistency. Clearly, we have $Y_k = Y$. For every $i \in \{1, \ldots, k\}$, we let an adversary decide whether $X_i \leq Y_{i-1}$ or $X_i > Y_{i-1}$.

Fix any $\ell \in \mathbb{N}$, and let $j_0, j_1, \ldots, j_\ell$ be the last $\ell+1$ positions for which the adversary has chosen $X_{j_i} \leq Y_{j_i-1}$. By our choice of j_i, we have $Y_{j_i-1} = Y_{j_{i-1}}$, because between $j_i - 1$ and j_{i-1} our adversary has only chosen $X_i > Y_{i-1}$. The crucial observation is that $Z \geq \ell + 1$ if and only if $X_{j_i} = Y_{j_{i-1}}$ for all $i \in \{1, \ldots, \ell\}$. By independence and assumption, the probability of this is bounded from above by c^ℓ. Overall, we obtain

$$\begin{aligned}
\mathbb{E}[Z] &\leq \sum_{i=1}^{\infty} \left(i(c^{i-1} - c^i) \right) + k^2 q \\
&= \sum_{i=0}^{\infty} c^i + k^2 q \\
&= \frac{1}{1-c} + k^2 q
\end{aligned}$$

as claimed. □

To actually get the result we require for the specific case of binomial random variables, we show that the value for c from Lemma 4.14 can be chosen arbitrarily small. Intuitively, this is because for binomial distributions with large values of n, adjacent values not too far from the mean have approximately the same probability.

Lemma 4.15. *Fix any $p \in (0,1)$. Let $X_1, \ldots, X_k \sim \mathrm{Binom}(n,p)$ be independent random variables distributed according to a binomial distribution with parameters n and p, and assume $k = O(n)$. Let $Y = \min\{X_1, \ldots, X_k\}$, and let $Z = |\{i \mid X_i = Y\}|$. Then*

$$\mathbb{E}[Z] \leq 1 + o(1)\,.$$

Proof. We show that the value for c in Lemma 4.14 can be chosen as $c = o(1)$, provided that n is sufficiently large.

Let $t = np - a$ for $a = \sqrt{n}\log n$. Now by applying Chernoff's bound [40] using $\varepsilon = \frac{\sqrt{n}\log n}{np}$ we get for any i:

$$\begin{aligned}\mathbb{P}[|X_i| < (1-\varepsilon)np] &= \mathbb{P}\left[|X_i| < np - \sqrt{n}\log n\right] = \mathbb{P}[|X_i| < t] \\ &< e^{-np\varepsilon^2/2} = e^{-np\frac{n\log^2 n}{n^2p^2}/2} \\ &= e^{\frac{-\log^2 n}{2p}} = \left(e^{-\log n}\right)^{\frac{\log n}{2p}} \\ &= \left(\frac{1}{n}\right)^{\frac{\log n}{2p}} = o\left(\frac{1}{k^2}\right).\end{aligned}$$

Thus, we can choose $q = o(1/k^2)$ for the application of Lemma 4.14.

Now we choose a slowly growing, integer valued $x = x(n) = \omega(1)$ with $x = o(a)$. We will give further constraints for the function x later on. Our goal is to show that it is possible to choose $c = 2/x = o(1)$. This together with our choice of q yields

$$\mathbb{E}[Z] \leq \frac{1}{1-2/x} + qk^2 = 1 + \frac{2/x}{1-2/x} + qk^2 = 1 + o(1)$$

as claimed.

Now fix any $s > t$ and let n be sufficiently large such that $p > \frac{2a}{n} = \frac{2\log n}{\sqrt{n}} > \frac{a+x}{n}$. We have

$$\begin{aligned}\mathbb{P}[X_i = s \mid X_i \leq s] &\leq \frac{\mathbb{P}[X_i = s]}{\mathbb{P}[X_i \in \{s, s-1, \ldots, s-x+1\}]} \\ &= \frac{\binom{n}{s}p^s(1-p)^{n-s}}{\sum_{\ell=s-x+1}^{s}\binom{n}{\ell}p^\ell(1-p)^{n-\ell}} \\ &= \frac{n!\cdot p^s(1-p)^{n-s}}{s!\cdot(n-s)!\cdot\sum_{\ell=s-x+1}^{s}\frac{n!}{\ell!\cdot(n-\ell)!}p^\ell(1-p)^{n-\ell}} \\ &= \frac{1}{\sum_{\ell=s-x+1}^{s}\frac{(1-p)^{s-\ell}}{p^{s-\ell}}\cdot\prod_{i=\ell+1}^{s}\frac{i}{n-i}}. \qquad (4.3)\end{aligned}$$

Let us estimate the product within the summation in the denominator:

$$\prod_{i=\ell+1}^{s}\frac{i}{n-i} \geq \left(\frac{s-x}{n-s+x}\right)^{s-\ell} \geq \left(\frac{t-x}{n-t+x}\right)^{s-\ell} = \left(\frac{p-\frac{a+x}{n}}{1-p+\frac{a+x}{n}}\right)^{s-\ell}.$$

We now bound the base of the exponentiation in the right-hand side by $(1-\varepsilon)\frac{p}{1-p}$ for some $\varepsilon > 0$. We start by deriving a bound on this ε as follows:

$$\begin{aligned}
\frac{p - \frac{a+x}{n}}{1-p+\frac{a+x}{n}} &\geq (1-\varepsilon)\frac{p}{1-p} \\
\Leftrightarrow p - \frac{a+x}{n} - p^2 + p\frac{a+x}{n} &\geq (1-\varepsilon)\left(p - p^2 + p\frac{a+x}{n}\right) \\
\Leftrightarrow -\frac{a+x}{n} &\geq -\varepsilon\left(p - p^2 + p\frac{a+x}{n}\right) \\
\Leftrightarrow \varepsilon &\geq \frac{\frac{a+x}{n}}{p - p^2 + p\frac{a+x}{n}}\,.
\end{aligned}$$

So in particular, if we let $\varepsilon = \frac{\frac{a+x}{n}}{p-p^2} = \frac{a+x}{n}\left(\frac{1}{1-p} + \frac{1}{p}\right)$, we obtain

$$\left(\frac{p - \frac{a+x}{n}}{1-p+\frac{a+x}{n}}\right)^{s-\ell} \geq \left((1-\varepsilon)\frac{p}{1-p}\right)^{s-\ell}.$$

Plugging this into (4.3) and observing that $s - \ell \leq x$ then yields

$$\mathbb{P}[X_i = s \mid X_i \leq s] \leq \frac{1}{\sum_{\ell=s-x+1}^{s}(1-\varepsilon)^{s-\ell}} \leq \frac{1}{x \cdot (1-\varepsilon)^x}\,.$$

As we want to have $c = 2/x$, we have to bound this right-hand side by $2/x$.

$$\begin{aligned}
\frac{1}{x \cdot (1-\varepsilon)^x} &\leq \frac{2}{x} \\
\Leftrightarrow \frac{1}{(1-\varepsilon)^x} &\leq 2
\end{aligned}$$

By making use of the inequality $(1+\alpha) \leq e^{\alpha}$ which holds for any $\alpha \in \mathbb{R}$, we can instead consider the following sufficient condition:

$$(e^{-\varepsilon})^{-x} = e^{\varepsilon x} \leq 2\,.$$

Now by expanding ε we rewrite this as:

$$e^{x \cdot \frac{a+x}{n} \cdot \frac{1}{(1-p)p}} \leq 2\,.$$

Let us assume that we can find an x with $x = o(a)$ (we shall soon see this is

the case) then we can simplify this to

$$\begin{aligned} e^{x\cdot\frac{a+x}{n}\cdot\frac{1}{(1-p)p}} &\leq e^{x\cdot\frac{2a}{n}\cdot\frac{1}{(1-p)p}} \\ &= e^{x\cdot\frac{2\sqrt{n}\log n}{n}\cdot\frac{1}{(1-p)p}} \\ &= e^{x\frac{2\log n}{\sqrt{n}(1-p)p}} \leq 2 \\ \Leftrightarrow x\frac{2\log n}{\sqrt{n}(1-p)p} &\leq \log 2 \\ \Leftrightarrow x &\leq \frac{\log 2\sqrt{n}(1-p)p}{2\log n} \end{aligned}$$

So in particular we can use $x = \left\lfloor \frac{\log 2\sqrt{n}(1-p)p}{2\log n} \right\rfloor$ which has the desired properties $x = o(a)$ and $x = \omega(1)$. This completes the proof. □

4.6 Constant Bound for the Number of Candidates

Theorem 4.16. *Fix any $p \in (0,1)$, and let C be the (random) number of candidates if we draw instances according to $G_{n,n,p}$. Then*

$$\mathbb{E}[C] \leq \frac{1+o(1)}{p}.$$

Proof. Similar to Lemma 4.10, for $p' = (1-\varepsilon)p$ and $\varepsilon = \frac{\log n}{\sqrt{n}}$, the probability that some row or column in M contains less than np' 1s is $o(1/n)$ by Chernoff's bound [40]. If some row or column of M does have fewer 1s, we simply assume that we have n candidates. This adds only $o(1)$ to our final expected value, which is negligible. For the remainder of the proof we may thus assume all rows and columns contain at least np' 1s.

We proceed by bounding the probability that a row i contains a candidate. To establish an upper bound on this probability, we introduce a game on an unknown matrix M in which our adversary aims to increase the probability of row i containing a candidate as much as possible. For any fixed i, let us consider an unknown $n \times n$ matrix M and let our adversary pick a column j. We set $M_{i,j} = 1$ and let our adversary place additional 1s in column j so that it contains at least np' 1s. The other elements of M (i.e., those not in column j) are subsequently each assigned a 1 with probability p. Based on our assumption, every row and column now contains at least np' 1s. We now determine an upper bound on the maximum probability our adversary can achieve of row i containing a candidate.

The number and placement of 1s in column j is the only element of the game our adversary can influence to maximize the probability of row

i containing a candidate. Thus, the optimal strategy is to maximize the probability of (i, j) becoming a candidate. In order to do this, the number of 1s in column j has to be as small as possible (to force row i to select column j), so we may assume our adversary places no more than $np' - 1$ additional 1s for a total of np'. We assume row i thus selects column j.

Now let Z again be a random variable denoting the number of rows containing the smallest number of 1s among all rows having a 1 in column j. Recall that $\mathbb{E}[Z] \leq 1 + o(1)$ by Lemma 4.15. The probability of column j selecting row i in our algorithm is now bounded from above by $\mathbb{E}[Z]/np'$, which implies the probability of (i, j) becoming a candidate is also bounded by this probability. Plugging this in, we get

$$\begin{aligned}
\mathbb{E}[C] &\leq \sum_i \mathbb{P}[\text{row } i \text{ contains a candidate}] + o(1) \\
&= \sum_i \frac{\mathbb{E}[Z]}{np'} + o(1) = \frac{\sum_i \mathbb{E}[Z]}{np'} + o(1) = \frac{n\mathbb{E}[Z]}{np'} + o(1) \\
&= \frac{\mathbb{E}[Z] + o(1)}{\left(1 - \frac{\log n}{\sqrt{n}}\right)p} = \frac{1}{p} \cdot \frac{1 + o(1)}{\left(1 - \frac{\log n}{\sqrt{n}}\right)} \\
&= \frac{1}{p} \cdot \frac{1 - \frac{\log n}{\sqrt{n}} + \frac{\log n}{\sqrt{n}} + o(1)}{\left(1 - \frac{\log n}{\sqrt{n}}\right)} = \frac{1}{p} \cdot \left(1 + \frac{\frac{\log n}{\sqrt{n}}}{\left(1 - \frac{\log n}{\sqrt{n}}\right)} + o(1)\right) \\
&= \frac{1}{p} \cdot \left(1 + \frac{\log n}{\sqrt{n} - \log n} + o(1)\right) = \frac{1 + o(1)}{p}.
\end{aligned}$$

□

For any fixed $p \in (0, 1)$, we have a constant bound on the expected number of candidates. This implies the expected running-time of Algorithm 4.8 on random instances of $G_{n,n,p}$ is $O\left(n^2\right)$. This expected running-time is linear in the input size.

4.7 Concluding Remarks

Avoiding fill-in while performing Gaussian elimination is related to finding bisimplicial edges in bipartite graphs. Existing algorithms to find bisimplicial edges are based on matrix multiplication. Their running-time is dominated by the matrix multiplication exponent ($\omega \leq 2.376$). We have presented a new algorithm to find such pivots that is not based on matrix multiplication. Instead, our algorithm selects a limited number of candidate edges, checks them for bisimpliciality, and finds all other bisimplicial edges based on that. The worst-case running-time of our algorithm is

$O\left(n^3\right)$, but the expected running-time for random $G_{n,n,p}$ instances for fixed values of p is $O\left(n^2\right)$, which is linear in the input size. The main reason for this difference is that the expected number of candidates is only $\frac{1+o(1)}{p}$.

Besides improving on the expected running-time on random instances, our new algorithm is also very easy to implement in an efficient way. The running-time can be brought down easily to $O(Cm)$, where the number of candidates C is known after time $O(m)$ and is bounded from above by n. Thus, we have a worst-case running-time of $O(nm)$. The combination of ease of efficient implementation and a linear bound on the average-case running-time makes our algorithm very practical.

Chapter 5

Perfect Elimination

In the previous chapter the problem of finding pivots that avoid fill-in has been discussed. A natural extension of this problem is to ask for matrices that allow the entire Gaussian elimination procedure to be performed without fill-in. The class of such matrices is closely related to the class of perfect elimination bipartite graphs. In this chapter we discuss the recognition of these classes of graphs and matrices. Besides existing algorithms for this problem, we present two new algorithms aimed at recognizing sparse instances. In the second part of this chapter a related problem is investigated: In order to allow perfect elimination on more matrices a more fine-grained elimination procedure is proposed and it is shown that this procedure allows the avoidance of fill-in on more matrices. Unfortunately this comes at a price as we also show recognition of graphs and matrices that allow perfect elimination using this more fine-grained procedure is NP-hard.

5.1 Perfect Elimination

Performing Gaussian elimination on sparse matrices may have the unfortunate side effect of turning zeros into nonzero values, possibly even leading to a dense matrix along the way. Clearly, this can be undesirable, for example when working with very large sparse matrices. A natural question therefore is to ask when we can avoid fill-in during the elimination process. Recognizing matrices where fill-in can be avoided and selecting appropriate pivots can decrease the required effort and space for Gaussian elimination. For several special cases, such as symmetric (positive definite) matrices or pivots chosen along the main diagonal, this problem has been treated extensively in literature (see e.g. [42, 43, 44, 45, 46]).

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 4 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Figure 5.1: Sparse M may lead to dense $Q = MM^T$.

The general case of avoiding fill-in on square nonsingular matrices was first treated in detail by Golumbic and Goss [16]. They describe the correspondence between matrices that allow Gaussian elimination without fill-in and the class of *perfect elimination bipartite graphs* which will be described in more detail in the next section. This class of bipartite graphs is characterized by an elimination scheme based on the notion of bisimplicial edges as discussed in the previous chapter.

The correspondence between perfect elimination bipartite graphs and matrices is mainly of practical value for sparse instances: the original motivation for investigating this class of graphs is preserving sparsity during Gaussian elimination on their associated matrices by avoiding fill-in. For the specific case of pivots chosen on the diagonal, Rose and Tarjan [46] have described two algorithms for finding perfect elimination orderings. Their algorithms represent the common trade-off between time and space. One is faster but needs more space, the other is slower but requires storage proportional to the number of nonzero elements. However, for the general case it appears that efficient algorithms for the recognition of sparse instances have not yet been investigated. The focus in literature so far seems to be only on time complexity for dense instances. The best known algorithms for the general case are based on a matrix multiplication which may well result in a dense matrix, see e.g. Figure 5.1.

The first part of this chapter describes two algorithms for the recognition of perfect elimination bipartite graphs that we have constructed with special focus on sparse instances and have published recently [8]. The second part describes a more fine-grained variant of Gaussian elimination that has been analyzed by us [9].

In this chapter we again work under the assumption that subtracting a multiple of a row from another will always turn at most one element from nonzero to zero. We also use the same $\{0,1\}$-matrix representation of M for our instances as in the previous chapter. Furthermore, we also assume each row and column contains at least one non-zero element, as this is quite a natural requirement and it somewhat simplifies our discussion. Again, we denote by n the number of rows (or columns) of our input matrix and by m the number of non-zero elements in it.

5.2 Perfect Elimination Bipartite Graphs

Recall from the previous chapter that an edge uv of a bipartite graph is called *bisimplicial* if the neighbors of its endpoints $\Gamma(u) \cup \Gamma(v)$ induce a complete bipartite graph. Bisimplicial edges in $G[M]$ correspond to pivots that avoid fill-in in M. Using this notion Golumbic and Goss [16] first defined the class of perfect elimination bipartite graphs which corresponds to the class of matrices that allow elimination without any fill-in as follows:

Definition 5.1. A bipartite graph $G = (U, V, E)$ is called *perfect elimination bipartite*, if there exists a sequence of pairwise nonadjacent edges $[u_1v_1, \ldots, u_nv_n]$ such that u_iv_i is a bisimplicial edge of $G - \{u_1, v_1, \ldots, u_{i-1}, v_{i-1}\}$ for each i and $G - \{u_1, v_1, \ldots, u_n, v_n\}$ is empty. Such a sequence of edges is called a *(perfect elimination) scheme*.

This definition is based on the following theorem:

Theorem 5.2 ([16]). *If uv is a bisimplicial edge of a perfect elimination bipartite graph $G = (U, V, E)$, then $G - \{u, v\}$ is also a perfect elimination bipartite graph.*

This theorem immediately implies a simple $O\left(n^5\right)$ algorithm for the recognition of perfect elimination bipartite graphs that also leads to an elimination scheme in case the graph is perfect elimination bipartite. We will proceed by describing this algorithm more formally as we will subsequently improve on it.

Let us introduce the notion of row and column sets R_i and C_j defined as follows:

$$R_i = \left\{ j \in \{1 \ldots n\} | M_{i,j} \neq 0 \right\}$$
$$C_j = \left\{ i \in \{1 \ldots n\} | M_{i,j} \neq 0 \right\}$$

In other words: R_i contains the column numbers of elements in row i that have a nonzero value in M. Using these, we can describe the algorithm by Golumbic and Goss, shown in Algorithm 1. The algorithm basically performs n iterations, during each of which all remaining edges are completely checked for bisimpliciality.

Goh and Rotem [38] have presented a faster recognition algorithm based on the notion of majorizing rows. (This has also been described in Chapter 4, Theorem 4.2, it is repeated here to keep this chapter self-contained.) A row $M_{a,*}$ is said to *majorize* a row $M_{b,*}$ if for each $1 \leq j \leq n$ we have $M_{a,j} \geq M_{b,j}$. Note how according to this definition, every row majorizes itself. Using this, they came up with the following theorem regarding bisimplicial edges:

Algorithm 1 Original recognition algorithm by Golumbic and Goss.

```
1: I ← {1...n}
2: J ← {1...n}
3: while I ≠ ∅ do
4:    f ← false
5:    for all (i, j) ∈ I × J do
6:       if M_{i,j} = 1 then
7:          g ← true
8:          for all (k, l) ∈ (C_j ∩ I) × (R_i ∩ J) do
9:             if M_{k,l} = 0 then
10:               g ← false
11:            end if
12:         end for
13:         if g = true then
14:            f = true, x ← i, y ← j
15:         end if
16:      end if
17:   end for
18:   if f = false then
19:      return false {G[M] is not perfect elimination bipartite}
20:   end if
21:   I ← I \ x
22:   J ← J \ y
23: end while
24: return true {G[M] is perfect elimination bipartite}
```

Theorem 5.3. *[38] Let M be an $n \times n$ $\{0,1\}$-matrix representing a bipartite graph $G = (U, V, E)$. Let ℓ_i be the number of rows in M that majorize row i and let s_j be the sum of the entries in column j of M. Then $M_{i,j} = 1$ and $\ell_i = s_j$ iff the edge $u_i v_j$ is a bisimplicial edge of G.*

The values ℓ_i can be easily determined using the matrix $Q = MM^T$: ℓ_i is equal to the number of elements in the row $Q_{i,*}$ that are equal to $Q_{i,i}$ (including $Q_{i,i}$ itself). Once the matrix Q has been computed (this can of course be done in time $O\left(n^3\right)$), finding a bisimplicial edge can be done in $O\left(n^2\right)$ operations. If a bisimplicial edge uv is found, Q can be updated in $O\left(n^2\right)$ operations to the matrix Q' associated with $G' = G - \{u, v\}$ for the next iteration. After at most n iterations the algorithm terminates. The total time complexity of the algorithm is $O\left(n^3\right)$, a significant improvement over the $O\left(n^5\right)$ naive implementation. This algorithm is shown in Algorithm 2 (The notation M^{ij} is used to denote the (i, j) minor of M, i.e.,

the matrix obtained after removing row i and column j). As it needs to compute and store the matrix Q, the space complexity of this algorithm is $\Omega\left(n^2\right)$.

Algorithm 2 Recognition algorithm by Goh and Rotem.

```
1: simplicial_found ← true
2: compute the matrix Q = (Q_{i,j}) where Q = MM^T
3: ∀j ∈ {1...n} : s_j ← Σ_{i=1}^{n} M_{i,j}
4: while there exists an s_j ≠ 0 and simplicial_found do
5:    ∀i ∈ {1...n} : let ℓ_i be the number of entries in row i of Q that are
      equal to Q_{i,i}
6:    if there exists a nonzero entry M_{i,j} in M where s_j = ℓ_i then
7:       Compute the matrix D = (d_{k,l}) where d_{k,l} = M_{k,j} · M_{l,j}
8:       Q ← (Q − D)^{ii} {Q is now equal to (M^{ij})(M^{ij})^T}
9:       ∀k ∈ {1...n} : s_k ← s_k − M_{i,k}
10:      s_j ← 0
11:   else
12:      simplicial_found ← false
13:   end if
14: end while
15: return simplicial_found
```

More recently, Spinrad [47] obtained an improved algorithm with time complexity $O\left(n^3/\log n\right)$ using a notion of edges that may soon become suitable pivots during subsequent iterations as well as the faster matrix multiplication algorithm by Coppersmith and Winograd [39].

5.3 Goh-Rotem on Sparse Instances

By adapting the way calculations are performed, as well as the data structures, we have obtained a new implementation [8] of Algorithm 2 with time complexity $O(nm)$: an improvement for sparse graphs. Using the row and column sets we determine the matrix $Q = MM^T$ as

$$Q_{i,j} = \left|R_i \cap R_j\right| \ . \tag{5.1}$$

Based on this new formulation, we arrive at the following lemma that will be used below to analyze the time complexity of our new algorithm:

Lemma 5.4. *An upper bound on the sum of the elements in Q is given by*

$$\sum_{i,j} Q_{i,j} \leq nm \ . \tag{5.2}$$

Proof.

$$\sum_{i,j} Q_{i,j} = \sum_{i,j} \left|R_i \cap R_j\right| \leq \sum_i \sum_j \left|R_j\right| = nm \tag{5.3}$$

□

Besides the matrix Q, we require an additional $n \times (n+1)$ matrix B. To simplify notation, we number the columns of B starting at 0 instead of at 1. The values of B are defined by

$$B_{i,k} := \left|\left\{j \mid j \in \{1 \dots n\}, Q_{i,j} = k\right\}\right| \ . \tag{5.4}$$

I.e., $B_{i,k}$ contains the number of elements in row i of Q that have the value k. After computation of Q, the matrix B can be computed in time $O\left(n^2\right)$. Furthermore, without increasing the time complexity of the new algorithm, we can keep B up to date if we perform any updates to elements of Q. Using B, we can easily determine the value of ℓ_i as

$$\ell_i = B_{i,Q_{i,i}} \ . \tag{5.5}$$

Using our set-based calculation of Q and the new matrix B, we can adapt the original algorithm by Goh and Rotem and arrive at our new version shown in Algorithm 3. Apart from our use of the sets I and J to denote the rows and columns that are still part of M during the current iteration instead of taking minors of the involved matrices, the working of the algorithm is still basically identical to Algorithm 2. However, the additional bookkeeping of B and the upper bound on the sum of the elements in Q enable us to achieve an improved time complexity for sparse instances.

Theorem 5.5. *The time complexity of Algorithm 3 is $O(nm)$.*

Proof. From Lemma 5.4 we know that the sum of the elements of Q is bounded by $O(nm)$. This implies the initialization of the matrices Q and B in the loop on line 3 can be completed within time $O(nm)$. This leaves us with the task of establishing the same bound on the main loop from line 8 on down. Clearly, the main loop is executed up to n times, either finding and processing a pivot, or returning false during each iteration. Within the main loop, the first loop on line 10 processes each of the $O(m)$ edges in constant time. If a suitable pivot is found, we first update the R_i and C_j sets in lines 20 and 21. This can be done in time $O(m)$.

After that, we have to update the matrices Q and B in the loop on line 27. Every iteration of this inner loop decreases some element of Q by one. As none of the elements are decreased below zero, Lemma 5.4 again gives us a bound of $O(nm)$ on the number of iterations of this inner loop over the course of the entire algorithm.

Algorithm 3 Adapted Goh-Rotem algorithm.

Require: $Q = 0$ {Q is an $n \times n$-matrix}
Require: $B = 0$ {B is an $n \times (n+1)$-matrix}
1: $I \leftarrow \{1 \dots n\}$
2: $J \leftarrow \{1 \dots n\}$
3: **for all** $(i, j) \in I \times J$ **do**
4: $\quad Q_{i,j} \leftarrow \left|R_i \cap R_j\right|$
5: $\quad B_{i,Q_{i,j}} \leftarrow B_{i,Q_{i,j}} + 1$
6: **end for**
7: $\forall j \in J : s_j \leftarrow \left|C_j\right|$
8: **while** $I \neq \emptyset$ **do**
9: $\quad f \leftarrow$ **false**
10: $\quad$ **for all** $i \in I$ **do**
11: $\quad\quad$ **for all** $j \in R_i$ **do**
12: $\quad\quad\quad$ **if** $B_{i,Q_{i,i}} = s_j$ **then**
13: $\quad\quad\quad\quad f \leftarrow$ **true**, $x \leftarrow i, y \leftarrow j$
14: $\quad\quad\quad$ **end if**
15: $\quad\quad$ **end for**
16: $\quad$ **end for**
17: $\quad$ **if** $f =$ **false then**
18: $\quad\quad$ **return false** {$G[M]$ is not perfect elimination bipartite}
19: $\quad$ **end if**
20: $\quad \forall i \in I : R_i \leftarrow R_i \setminus y$
21: $\quad$ **for all** $j \in J$ **do**
22: $\quad\quad$ **if** $x \in C_j$ **then**
23: $\quad\quad\quad s_j \leftarrow s_j - 1$
24: $\quad\quad$ **end if**
25: $\quad\quad C_j \leftarrow C_j \setminus x$
26: $\quad$ **end for**
27: $\quad$ **for all** $(i, j) \in C_y \times C_y$ **do**
28: $\quad\quad B_{i,Q_{i,j}} \leftarrow B_{i,Q_{i,j}} - 1$
29: $\quad\quad Q_{i,j} \leftarrow Q_{i,j} - 1$
30: $\quad\quad B_{i,Q_{i,j}} \leftarrow B_{i,Q_{i,j}} + 1$
31: $\quad$ **end for**
32: $\quad \forall i \in I : B_{i,Q_{i,x}} \leftarrow B_{i,Q_{i,x}} - 1$
33: $\quad I \leftarrow I \setminus x$
34: $\quad J \leftarrow J \setminus y$
35: **end while**
36: **return true** {$G[M]$ is perfect elimination bipartite}

Finally, the loop on line 32 decreases $O(n)$ values of B after which I and J are updated to reflect the removal of the pivot row and column; all of this can be done in time $O(n)$.

So for both the initialization and the iteration phase of the algorithm we obtain a bound of $O(nm)$ on the time complexity. □

The space complexity of Algorithm 3 is $\Theta\left(n^2\right)$ as we need to compute and store the matrices Q and B.

5.4 Avoiding Matrix Multiplication

A possible disadvantage of recognition algorithms based on matrix multiplication is the amount of space required to store the result of the matrix multiplication. Even if an original sparse matrix M is stored efficiently using $\Theta(m)$ space, the result of the multiplication may be a dense matrix requiring $\Omega\left(n^2\right)$ space (see Figure 5.1). Avoiding matrix multiplication thus seems to be required in order to improve the space complexity. To do this, we started over from the algorithm originally presented by Golumbic and Goss for the recognition of perfect elimination bipartite graphs. Algorithm 1 proceeds in up to n iterations. In every iteration, every edge is checked against possibly all other edges to determine if it is bisimplicial. To check an edge uv for bisimplicity, we need to verify that $G[M]$ contains all edges $u'v'$ with $u' \in \Gamma(v)$ and $v' \in \Gamma(u)$. By performing this every iteration, we obtain a time complexity of $O\left(n^5\right)$.

The idea behind our new algorithm is as follows: in Algorithm 1 we check every remaining edge uv against possibly all other edges during every iteration. However, we can shave a factor n from the time complexity if we are checking uv and find an edge $u'v'$ as present in $G[M]$ during some iteration, we avoid checking it for uv again in subsequent iterations. A naive algorithm based on this notion is described in Algorithm 4. Assuming the use of suitable data structures, the time complexity of this algorithm is $O\left(n^2m\right)$. Unfortunately, by precomputing for every edge e the set of possible edges E_e that need to be checked, we require a lot more space, instead of less.

Observing the usage of the sets E_e we see they are all constructed at the beginning and processed one element at a time in arbitrary order. The element under consideration is either removed from the set and followed by another element, or it leads to the conclusion that e is not bisimplicial in the matrix that remains in the current iteration and it will be considered again later. If we impose a specific order on the processing of the edges $e' \in E_e$ we can do away with precomputing and storing the entire sets E_e and only store the element e' currently under consideration for each edge e.

Algorithm 4 A $O\left(n^2 m\right)$ recognition algorithm.

```
 1: I ← {1...n}
 2: J ← {1...n}
 3: ∀e = (i, j) ∈ E : E_e = C_j × R_i
 4: while I ≠ ∅ do
 5:    f ← false
 6:    for all e = (i, j) ∈ E do
 7:       g ← true
 8:       if i ∉ I ∨ j ∉ J then
 9:          g ← false
10:       end if
11:       while (E_e ≠ ∅) ∧ (g = true) do
12:          e' = (i', j') ← arbitrary_element(E_e)
13:          if i' ∉ I ∨ j' ∉ J then
14:             E_e ← E_e \ e'
15:          else if M_{i',j'} = 1 then
16:             E_e ← E_e \ e'
17:          else
18:             g ← false
19:          end if
20:       end while
21:       if g = true then
22:          f ← true, x ← i, y ← j
23:       end if
24:    end for
25:    if f = false then
26:       return false {G[M] is not perfect elimination bipartite}
27:    end if
28:    I ← I \ x
29:    J ← J \ y
30: end while
31: return true {G[M] is perfect elimination bipartite}
```

To implement this, we again represent M using the sets R_i and C_j, but this time we store them as sorted lists, as shown in Figure 5.2(a). To perform a pivot and remove the associated row and column, we simply adjust the links in the row and column lists to skip over the removed row and column, as shown in Figure 5.2(b) for a pivot on $(3,4)$. Clearly, such a pivot operation can be implemented in time $O(m)$, as we can simply pass over all the elements in each of the lists and adjust the links as we pass them. This representation requires $\Theta(m)$ space.

To check if an element $M_{i,j}$ corresponds to a bisimplicial edge in $G[M]$,

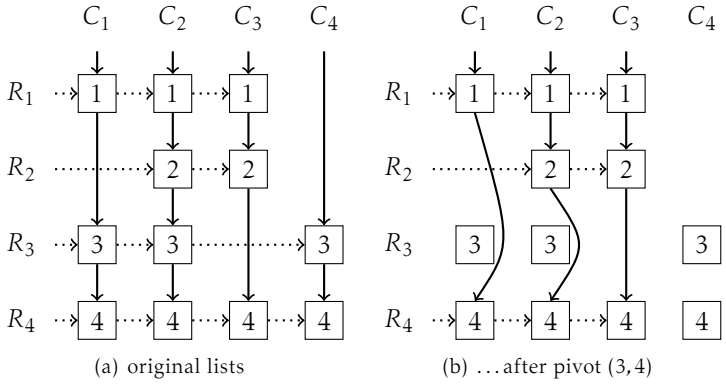


Figure 5.2: Row and column lists for example matrix M from Figure 4.1(a).

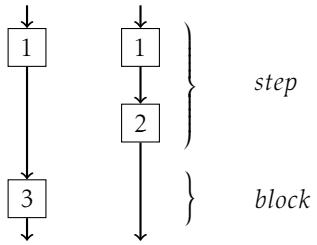


Figure 5.3: Steps and blocks.

we have to test if all edges between the neighbors of its endpoints exist. In terms of the column sets of the matrix M, this means that for every column $k \in R_i$, we must have that $C_j \subseteq C_k$. If we use the sorted list representation, the number of comparisons for each edge e is bounded by $O(m)$. Every comparison has one of three possible outcomes (see Figure 5.3):

1. C_j and C_k both contain the row number: the required edge is present, we can continue checking the next row number
2. C_k contains a number not present in C_j: an additional edge is present, we can continue checking the next row number
3. C_j contains a number not present in C_k: a required edge is missing: e is not bisimplicial in the current matrix

We call the first two cases *steps* (as they can be repeated during a single iteration) and call the third case a *block* (as it ends the checks for e during

this iteration). For a single edge e, steps can occur $O(m)$ times during the algorithm, whereas blocks are limited by $O(n)$ as they can occur only once per iteration. If there are no more comparisons left for any edge e that still remains in M at some point during the algorithm, we have found a suitable pivot. After removing the pivot row and column from M, we simply proceed checking the remaining edges starting at the point where they blocked during the previous iteration. We continue this process until either we have found a complete elimination scheme or we cannot find a bisimplicial edge anymore. This procedure is described in Algorithm 5.

Theorem 5.6. *The time complexity of Algorithm 5 is* $O\left(m^2\right)$.

Proof. It is possible to construct our sorted list representation in time $O\left(n^2\right)$. Other initialization, such as the state of the comparisons for every edge e, can easily be done within the same time. Following the initialization, we perform up to n iterations, during each of which we perform a pivot on our rows and columns lists in time $O(m)$ for a total of $O(nm)$. During the entire algorithm we perform $O(m)$ steps and $O(n)$ blocks for each of the m edges, leading to an overall time complexity of $O\left(m^2\right)$. □

Theorem 5.7. *The space complexity of Algorithm 5 is* $\Theta(m)$.

Proof. Our sorted lists representation of M contains m edges. For each edge we need $\Theta(1)$ space to store its progress with respect to its comparisons against its required neighbors for a total of $\Theta(m)$. Finally, we have to store the sets I and J to keep track of the rows and columns that still remain, both require $\Theta(n)$ space. In total, we thus obtain a space complexity of $\Theta(m)$. □

After establishing its running time and space requirements, we end this section by adapting our new algorithm to the special case of finding a perfect elimination ordering allowing only pivots on the diagonal of the matrix. Rose and Tarjan have studied this problem and have presented two algorithms for it also focusing on the trade-off between time and space requirements [46]. One of their algorithms has a time complexity of $O(nm)$ and uses $\Theta(nm)$ space, the other one has a time complexity of $O\left(n^2m\right)$ but uses only $\Theta(m)$ space.

It is not hard to see that our algorithm can be adapted to consider only a subset of all the edges as pivots: this simply means we only process steps and blocks for these edges while ignoring the other edges. If we test only c edges as allowed pivots in this way, the running time of our algorithm is $O(cm+nm)$ while the space complexity remains $\Theta(m)$. By only allowing pivots on the diagonal ($c = n$) instead of anywhere ($c = m$), we get a time

Algorithm 5 A new $O\left(m^2\right)$ recognition algorithm using $\Theta(m)$ space.

```
1: I ← {1...n}
2: J ← {1...n}
3: Construct R_i and C_j representation
4: while I ≠ ∅ do
5:    f ← false
6:    for all e = (i, j) ∈ E do
7:       g ← true
8:       if i ∉ I ∨ j ∉ J then
9:          g ← false
10:      end if
11:      while g = true and we are not done checking edges do
12:         e' = (i', j') ← the current edge to check
13:         if i' ∉ I ∨ j' ∉ J then
14:            Proceed to the next edge to check (if any) {This can only hap-
               pen during the first iteration of this inner loop}
15:         else if e' blocks then
16:            g ← false
17:         else
18:            Proceed to the next edge to check (if any)
19:         end if
20:      end while
21:      if g = true then
22:         f ← true, x ← i, y ← j
23:      end if
24:   end for
25:   if f = false then
26:      return false {G[M] is not perfect elimination bipartite}
27:   end if
28:   Update R_i and C_j links to perform pivot round (x, y)
29:   I ← I \ x
30:   J ← J \ y
31: end while
32: return true {G[M] is perfect elimination bipartite}
```

complexity of $O(nm)$ for this restricted case. We thus obtain a single algorithm that combines the best time complexity of Rose and Tarjan with their best space complexity for this restricted problem.

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

(a) (b)

Figure 5.4: Two matrices, both without a perfect elimination scheme, but (a) has a perfect partial elimination scheme whereas (b) does not.

5.5 Perfect Partial Elimination

The previous sections focused on the recognition of graphs and matrices that allowed complete avoidance of fill-in under Gaussian elimination. It is characteristic for the Gaussian elimination algorithm that in each iteration a pivot element is picked and used to clear its entire column. If we want to avoid fill-in completely, this limits the set of matrices we can apply this method to. In order to achieve perfect elimination for a broader set of matrices, we can use a more fine-grained elimination process in which we eliminate single nonzero values instead of entire columns at a time. Rose and Tarjan already mentioned such *partial elimination* as an alternative to ordinary Gaussian elimination for this reason [46]. In the subsequent sections we describe our analysis [9] of the complexity of this alternative process for perfect elimination.

In the remainder of this chapter we consider partial pivots, i.e., selecting a new pivot element for every single nonzero element that we zero. For example, the matrix in Figure 5.4(a) can be reduced to diagonal form by partial pivots without fill-in – although no perfect elimination scheme exists. Clearly, there are also matrices like the one in Figure 5.4(b) for which even partial pivoting cannot avoid fill-in. It is thus natural to ask ourselves which matrices allow perfect elimination using such partial pivots.

To answer this question, reconsider the elimination of edges in the corresponding bipartite graph. As the partial elimination steps involve only row operations zeroing single nonzero elements in our matrix, we first look at elements that can be zeroed this way. Recall that U denotes the vertex class corresponding to the rows of the matrix and V denotes the vertex class corresponding to the columns of the matrix.

Definition 5.8. An edge uv of a bipartite graph $G = (U, V, E)$ is called *disposable* if there exists another edge $u'v$ such that $\Gamma(u') \subseteq \Gamma(u)$.

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

Figure 5.5: The disposable elements (*indicated by squares*) of our example matrix.

$$\begin{pmatrix} 10 & 1 & 0 & 0 \\ 2 & 9 & 0 & 0 \\ 3 & 0 & 8 & 5 \\ 0 & 4 & 6 & 7 \end{pmatrix}$$

Figure 5.6: An example perfect partial elimination scheme.

Before defining the analogous concept for a $\{0,1\}$-matrix M, we reiterate the $\leq$ relation on matrix rows. We write $M_{i',*} \leq M_{i,*}$ to denote that for any column j of M, $M_{i',j} = 1$ implies $M_{i,j} = 1$. Note how this captures the same notion expressed by $\Gamma(u') \subseteq \Gamma(u)$ in the bipartite graph case. Defining disposable elements in the $\{0,1\}$-matrix M is now rather straight-forward: a nonzero element $M_{i,j}$ of a row $M_{i,*}$ is called *disposable* if there is another row $M_{i',*} \leq M_{i,*}$ such that $M_{i',j}$ is also nonzero. (Element $M_{i',j}$ can be used as a partial pivot to clear element $M_{i,j}$.) Figure 5.5 shows the disposable elements of our example matrix. Clearly, disposable elements play an important role in the characterization of the matrices that allow perfect partial elimination schemes. We now come to the definition of the class of bipartite graphs associated to these matrices:

Definition 5.9. A bipartite graph $G = (U, V, E)$ with $|U| \geq |V| = n$ and $|E| = m$ is called *perfect partial elimination bipartite* if there exists an ordering of the edges denoted by $E = \{e_1, \ldots, e_m\}$ such that $\{e_{m-n+1}, \ldots, e_m\}$ together form a maximum matching of G covering V and for each $i \in \{1, \ldots, m-n\}$, e_i is a disposable edge of $G_i := \left(U, V, \left\{e_j \in E \mid j \geq i\right\}\right)$. We call this ordering a *perfect partial elimination scheme*.

An example of a perfect partial elimination scheme for the $\{0,1\}$-matrix of Figure 5.4(a) is shown in Figure 5.6.

As our partial pivots only involve single edge operations, it is no longer required for the two vertex classes to be of equal size. Clearly, the notion of a perfect partial elimination scheme for G carries over readily to the corresponding $\{0,1\}$-matrix, which also no longer has to be square.

Having defined the class of matrices and bipartite graphs that allow perfect partial elimination, the logical next question to ask is how hard it is to recognize members of this class. The corresponding decision problem can be stated as follows:

Perfect Partial Elimination
Instance: A $\{0,1\}$-matrix M
Question: Does M have a perfect partial elimination scheme?

5.6 Perfect Partial Elimination is NP-hard

The proof is by reduction from Satisfiability [4]. We start by briefly defining this problem, using the terminology and notation from Garey and Johnson [2]. An instance S of Satisfiability consists of a set U of boolean variables and a set C of clauses defined over U. For every variable $u_i \in U$, u_i and $\bar{u}_i$ are called *literals* over U. A *truth assignment* (T,F) is a partition of the variables U. Under a given truth assignment, the literal u_i is true if and only if $u_i \in T$, otherwise it is false. Similarly, the literal $\bar{u}_i$ is true if and only if $u_i \in F$. The clauses in C are disjunctions of the literals over U. A given truth assignment is called *satisfying* for C if every clause in C contains at least one literal that is true under the truth assignment. The decision problem is now stated as follows:

Satisfiability
Instance: Set U of variables, collection C of clauses over U.
Question: Is there a satisfying truth assignment for C?

Using this we prove the NP-hardness of Perfect Partial Elimination.

Theorem 5.10. Perfect Partial Elimination *is NP-complete.*

Proof. As a given elimination sequence for perfect partial elimination can clearly be verified in polynomial time, we only show NP-hardness using a reduction from Satisfiability. For a given instance $S = (U,C)$ of Satisfiability, we construct a corresponding $\{0,1\}$-matrix M_S such that M_S has a perfect partial elimination scheme if and only if S has a truth assignment for U that is satisfying for C. To simplify the reduction, we will assume w.l.o.g. that the instance S has at least two clauses, all clauses contain at least one literal, and C contains no tautologies, i.e., there is no i such that some clause contains both u_i and $\bar{u}_i$.

The matrix M_S has $4|U|+|C|+1$ rows and $2|U|+|C|+2$ columns. For the description of the construction procedure, it is convenient to label the rows and columns.

For every variable u_i in U we have two columns labeled u_i and $\bar{u}_i$. These columns are used to represent the variables, their truth assignments and their occurrences in the clauses. For every clause $c_i \in C$, we have a column labeled c_i. These columns are used to keep the individual clauses separated while linking them together in the overall satisfiability requirement. We also have two auxiliary columns labeled a and b, which are used to limit the possible subtractions between rows.

The rows of M_S are partitioned into five sets: the first two sets each contain one row per variable and are denoted V and W. Subtractions between rows of these sets are used to represent possible truth assignments for U. The third set D is used mainly to clear matrix elements that are no longer required for the elimination process themselves. The fourth set of rows, K, represents the clauses of S and links them to their literals. The final set R contains only a single row and encodes the requirement that all clauses must be satisfied by the truth assignment.

Having introduced the rows and columns that together form the constructed matrix M_S, we will now describe the values of the elements of M_S. The row set V contains a single row v_i for every variable u_i. Each such row contains two ones in the columns corresponding to u_i and $\bar{u}_i$ and zeros everywhere else. The rows w_i in W are identical to the rows v_i, except for an additional 1-entry in column a. The set D contains two rows for each variable u_i: one for each of the two corresponding literals u_i and $\bar{u}_i$. Each row in D has a one in the corresponding literal column and a one in column b and zeros everywhere else. The rows in K each correspond to a clause in C. Row k_i has a one in every column corresponding to a literal occurring in c_i, as well as a one in the column corresponding to c_i itself. All rows in K also have a one in the columns a and b and zeros elsewhere. Finally, the set R contains only a single row r with ones in all columns c_i as well as in the b column, and zeros everywhere else. An example of this construction for $S = (U, C)$ with $U = \{u_1, u_2, u_3, u_4\}$ and $C = \{\{u_1, u_2\}, \{u_1, \bar{u}_2, u_3, \bar{u}_4\}, \{u_1, \bar{u}_3, u_4\}\}$ is shown in Figure 5.7 where the nonzero entries have been numbered according to the elimination scheme that will be described next. To complete the reduction, we have to show that M_S has a perfect partial elimination scheme if and only if S is satisfiable. We first show how a satisfying truth assignment for S leads to a perfect partial elimination scheme for M_S.

Let (T, F) be a satisfying truth assignment for S. For every $u_i \in T$, we use the corresponding row in V to clear the element in the $\bar{u}_i$ column of the corresponding row in W. Similarly, for every $u_i \in F$, we use the corresponding row in V to clear the element in the u_i column of the corresponding row in W. The modified rows in W now represent the truth assignment

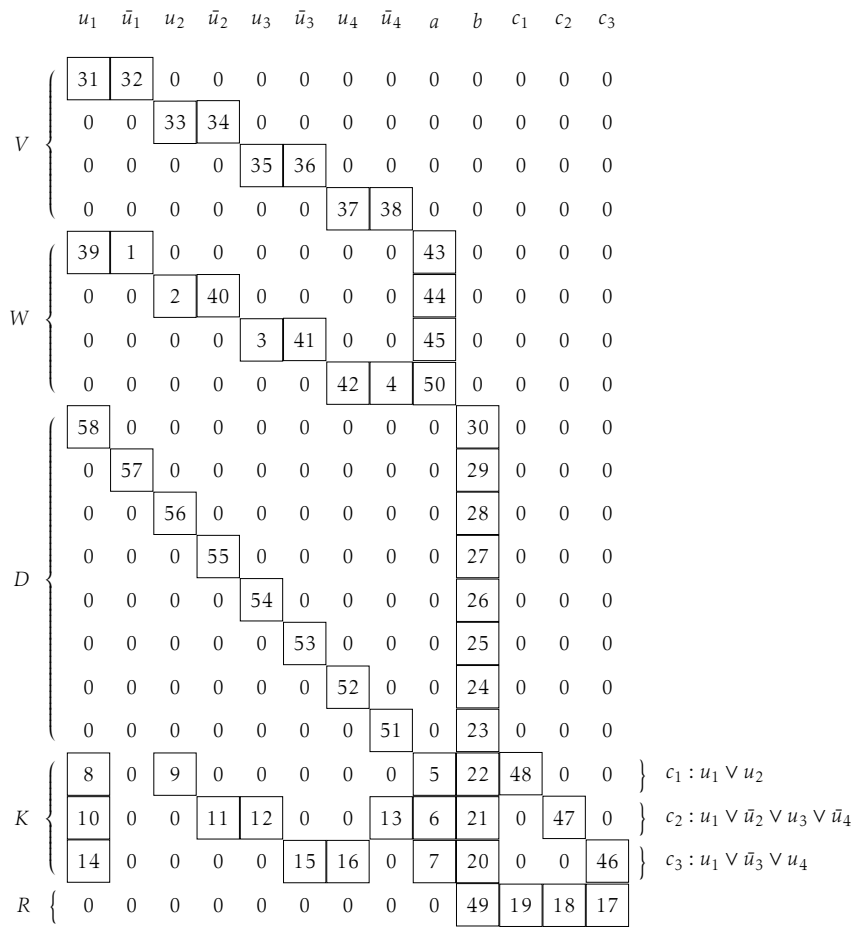


Figure 5.7: An example perfect partial elimination scheme for the construction used in the NP-hardness proof of Perfect Partial Elimination (*nonzero entries emphasized by squares*).

(T, F). As (T, F) is a satisfying truth assignment for S, we can find a row w_j for each clause row k_l such that $w_j \leq k_l$. We use these rows to clear all elements in the a column of K. Next, the rows from D are used to clear all the elements in the u_i and $\bar{u}_i$ columns of K. The only nonzero elements remaining in K are now the column b and the diagonal in the c_i columns. For every clause c_i we now have that $k_i \leq r$. The rows of K are now used to clear all c_i columns of r, so that the only nonzero element of r that remains is in column b. The remainder of the elimination scheme is rather

straightforward: The current row r can be used to clear column b in row sets D and K. The resulting rows in D can then zero all of the literal entries in other rows, leading to a matrix in which for each column there is a row with only a single 1-entry in exactly that column. After that, completion of the elimination scheme is a trivial task. Figure 5.7 shows an example of such an elimination scheme where the numbers in the figure represent the ordering from Definition 5.9.

It remains to show that no perfect partial elimination scheme exists if S is not satisfiable. The construction of M_S guarantees that a perfect partial elimination scheme, if one exists, must proceed over three distinct phases that can be characterized as follows:

I During the first phase, only rows from the row sets V, W and D can be used as pivots.

II During the second phase, rows from the row sets R and K can also be used as pivots, but only on other rows from R and K.

III During the final phase, rows from the row sets R and K can also be used as pivots on other rows of M_S.

We proceed by showing why during each phase at least one pivot must be performed before the next phase is reached and why each perfect partial elimination scheme contains pivots in each phase.

Due to the structure of the c_i columns and the assumption that there are at least two clauses, the row sets R and K cannot immediately be used as a pivot on M_S, so initially we can use only rows from the row sets V, W and D as pivots. This is phase I. Using only these rows as pivots, no row with a single 1-entry can be obtained.

Rows of R and K each contain at least a single 1-entry in one of the c_i columns. This means that before they can be used as pivots on rows from V, W or D, at least one row of R and K must be used as a pivot on another row of R and K. When such pivots have become possible, we have reached phase II.

To ultimately clear all but a single 1-entry in each of the rows of V, W and D as required for a perfect partial elimination scheme, at least one of the rows of R and K must be used as a pivot on the rows of V, W and D. When such pivots become possible, we have reached phase III.

Before a row from K can be used as a pivot on either another row of K or R, all of its 1-entries in the columns a, u_i and $\bar{u}_i$ must be cleared. The 1-entries in the u_i and $\bar{u}_i$ columns can be cleared using rows from D as pivots. The 1-entry in the a column can only be cleared by using a row from W after clearing a 1-entry in either the u_i or $\bar{u}_i$ column of that row. This means that for each row k_i of K that we want to use as a pivot, we have to pick some row from W that, after clearing one of its own 1-entries,

can be used to clear the 1-entry in the a column of k_i. If there is some way to do this for every row of K, the modified rows from W again represent a satisfying truth assignment, so we have to assume that for every possible usage of the rows from W to clear the a column of the rows of K, there is at least one row for which we can not clear the 1-entry in the a column. If there is more than one row for which this holds, we can never reach phase III, as none of the rows of K can be applied as pivots to other rows of K and after using the other rows as pivots, row r will always have at least two different 1-entries left in the c_i columns, so it can also never be used as a pivot. So let us assume there is exactly one row of K for which we can not clear the 1-entry in the a column. We refer to this row as z. In this case we can use all the other rows of K to clear all but a single 1-entry in the c_i columns of row r. Row r can subsequently be used to clear the last 1-entry in the c_i columns of z. (Row r could also be used to clear the 1 entry in column b instead, but that would clearly block row z from being used as a pivot.) The a column of row z still contains a 1-entry, so in order to use it as a pivot on a row from V, W or D and reach phase III, we will have to subtract it from another row with a 1-entry in the a column. The only rows eligible as operand for such a subtraction are the rows of W. Note that up to this point no row with only a single 1-entry could have been created.

By our assumption, none of the rows of W could be subtracted from z, as this would have enabled us to clear the 1-entry in the a column. This implies that in order to subtract z from any row of W, we have to clear all but a single 1-entry from row z first. This is true as z must contain fewer 1-entries than any row of W we want to subtract it from, because if z contains the same 1-entries, then the subtraction could have been performed the other way around which is impossible by our assumption. Now assume we have been able to clear all 1-entries of z except for the 1-entry in column a and one other 1-entry. To clear any of these two remaining 1-entries, we require another row with either a single 1-entry, none of which have been created yet, or a row with exactly the same two 1-entries as z, which again contradicts our assumption, as this row must be a row of W. It is therefore impossible to reach phase III if S is not satisfiable and therefore a perfect partial elimination scheme only exists if S is satisfiable. □

Remark 5.11. The problem does not become easier if we restrict ourselves to square matrices. Indeed, we can augment the matrix M_S with additional 'all ones' columns. A moment of reflection shows that these additional columns neither prevent a perfect partial elimination scheme if S has a satisfying truth assignment, nor do they allow such a scheme if S does not have a satisfying truth assignment.

5.7 Concluding Remarks

In this chapter we have analyzed two problems related to avoiding fill-in during Gaussian elimination: Recognition of sparse perfect elimination bipartite graphs and perfect partial elimination bipartite graphs.

In current literature, the fastest known algorithm for the recognition of general perfect elimination bipartite graphs is the algorithm by Spinrad [47] with a time complexity of $O\left(n^3/\log n\right)$. We have presented two new algorithms focused specifically on sparse instances. Our first algorithm is an adaption of the algorithm by Goh and Rotem with a time complexity of $O(nm)$, leading to an improvement for instances with $m = o\left(n^2/\log n\right)$ (all but the densest instances). The second algorithm we have presented is not based on some form of matrix multiplication and is as such able to do away with the $\Omega\left(n^2\right)$ space complexity associated with it. This algorithm has a time complexity of $O\left(m^2\right)$ and a space complexity of just $\Theta(m)$. For instances with $m = o\left(n\sqrt{n\log n}\right)$ this algorithm is faster than the algorithm by Spinrad while requiring less space. We have also shown how the restricted problem where only pivots on the diagonal are allowed can be solved in time $O(nm)$ using an adapted version of our algorithm.

When performing Gaussian elimination on sparse matrices, the choice of pivots is critical to preserving sparsity. Ideally, during elimination not a single zero element is turned into a nonzero. However, by being restricted to a single pivot per column, some possibilities for preserving sparsity may be missed. By clearing single elements at a time instead of performing pivots on entire columns at once, a more fine-grained variant on Gaussian elimination can achieve perfect elimination on a larger class of matrices and their associated bipartite graphs. However, we have shown that determining whether a matrix allows such a perfect partial elimination scheme is NP-hard. Intuitively, perfect partial elimination is harder than ordinary perfect elimination because clearing the wrong element can lead to a dead end in the elimination process, whereas with ordinary perfect elimination we can not go wrong by pivoting on any bisimplicial edge (cf. Theorem 5.2).

Chapter 6

Conclusions & Recommendations

In this chapter we briefly summarize the most important results from the previous chapters and outline their practical implications. For each of the main problems described in this thesis we also try to outline meaningful directions for further research.

While reading the conclusions and recommendations below, bear in mind that the results have been obtained while considering only the structural consistency of systems of equations. As discussed in Chapter 2 these results may not readily translate to individual problems, but they do offer more insight into the consistency of the classes of problems sharing the same structure (as defined by the bipartite graphs we use).

6.1 Decomposition

Conclusion The main result of Chapter 3 is the $W[1]$-hardness of finding a decomposition into small blocks. This leads to a negative outlook on a divide and conquer approach for constraint solving in the general case.

However, there are two important points to bear in mind regarding these results: First of all, in the application described in Chapter 1 lots of alternative solutions are generated for a given system of equations. If the structure of the system remains the same, we may analyze it once up front and reuse the decomposition we obtain in each iteration. If some computationally feasible way is found to determine a decomposition that is not necessarily optimal but rather 'good enough', a lot can still be gained by using it. And secondly, although the $W[1]$-hardness does provide us with an indication that a polynomial time algorithm may very well not exist even for the parameterized case, non-polynomial algorithms with passable complexities may still lead to usable exact algorithms.

Recommendations As already briefly touched upon in the conclusions, our first recommendation for further research lies in the development of exact algorithms for the decomposition problems described in Chapter 3.

Furthermore, the generality of our results may leave some room for improvements in more specific cases. The parameterization of our general case only involves the size of the largest blocks in the decomposition. Alternative or more extensive parameterizations of the decomposition problem might lead to further improvements: It may for example be interesting to also include the difference d between the number of equations and the number of variables of the entire system as a parameter and see whether that could be used to achieve polynomial solvability when we also fix the value of d. This will however not always be enough. For example a paper on the application of systems of equations in 3D scene reconstruction [48] mentions a system that contains 251 equations and 427 variables, a considerable difference. Another approach might be bounding the degree of the vertices by an additional parameter and trying to improve tractability that way. Further research in both directions is clearly required.

From a more application-oriented point of view, an interesting subject for further investigation would be heuristics to find small minimal free square blocks. Bliek et al. discuss one possible heuristic approach [25], however the performance of this approach has to our knowledge not been analyzed satisfactorily.

Another line of further investigation might lay in additional conditions on the system of equations. It might be possible to construct a good decomposition efficiently if more structural constraints can be placed on the system of equations, for example by restricting the problem instances to certain classes of bipartite graphs. Investigating such conditions and the corresponding algorithms for their decomposition definitely warrants additional research.

6.2 Bisimplicial Edges

Conclusion In Chapter 4 a new algorithm to find bisimplicial edges in bipartite graphs has been described. While this algorithm itself has the same worst case performance as existing algorithms, we have shown the expected performance on at least one class of random instances is significantly better. Furthermore, the basis of our algorithm is formed by several characteristics of bisimplicial edges that have been derived using a counting argument. It seems that these characteristics, while simple to deduce, have not been used previously in existing algorithms for finding either single bisimplicial edges or complete elimination schemes.

Recommendations Further research into possible applications might lead to improved algorithms for these problems. In particular we ask whether it would be possible to extend our algorithm for finding bisimplicial edges into a new algorithm for the recognition of perfect elimination bipartite graphs without simply running the entire algorithm for every single bisimplicial edge in a perfect elimination scheme.

6.3 Perfect Elimination

Conclusion In the first part of Chapter 5 we have presented two new algorithms for the recognition of perfect elimination bipartite graphs. Both are aimed at the efficient recognition of sparse instances, the trade-off between the two is in the amount of space required, respectively $\Theta\left(n^2\right)$ and $\Theta\left(m\right)$.

Recommendations Besides improving time and space complexity, another interesting aspect of algorithmic performance is the possibility of parallelization. From the algorithm of Goh and Rotem, it is not too hard to see that finding a single bisimplicial edge can be done in polylog time given a polynomial number of processors: matrix multiplication can be performed in polylog time [49] as well as the post-processing to determine the values of ℓ_i and check the individual matrix elements for bisimpliciality. Our new algorithm can be parallelized on $O\left(m^2\right)$ processors to find a bisimplicial edge in polylog time as all checks for all edges can be performed in parallel and subsequently combined in polylog time to find a bisimplicial edge if one exists. It is however unclear if it is also possible to use a polynomial number of processors to run the entire recognition process in polylog time: all currently known recognition algorithms are based on finding an elimination sequence of n bisimplicial edges and this appears to be an inherently sequential process. A fundamentally different approach might be necessary in order to achieve more parallelism and obtain a polylog time approach for the entire recognition process.

Another subject for further investigation is that of minimizing fill-in when it cannot be avoided completely. For symmetric positive definite matrices with pivots chosen along the main diagonal, minimizing the fill-in in the associated chordal graphs has been shown to be NP-hard [50]. Furthermore, for fill-in in chordal graphs an approximation algorithm has been developed [51]. As far as we know, the complexity of minimizing fill-in for general matrices and bipartite graphs is unknown. Considering the practical applications of minimum elimination orderings, obtaining results on the complexity in the general case, as well as either a polynomial time algorithm or an approximation algorithm for minimizing fill-in seem to be good topics for further research.

6.4 Perfect Partial Elimination

Conclusion In the second part of Chapter 5 we have analyzed a variation on Gaussian elimination where we no longer pick a single pivot element for clearing an entire column, but rather pick a separate pivot element for each individual element of the matrix we want to turn into a zero. We have shown that this more fine-grained approach to elimination allows us to avoid fill-in during reduction to triangular form for a larger class of matrices than ordinary Gaussian elimination, however, there is still a set of matrices left that can not be completely reduced while avoiding fill-in using this approach.

Unfortunately, even determining whether avoiding fill-in completely is possible for a given matrix using partial elimination pivots turned out to be NP-hard. As this implies that minimizing fill-in is also NP-hard, our new approach is not immediately suited to practical applications.

Recommendations However, as our analysis only treats the general case, it may be interesting to also investigate whether more restricted classes of matrices do admit a polynomial time algorithm for partial elimination. Another subject for further research could be parameterized versions of the Perfect Partial Elimination decision problem, for example bounding the degree of vertices in the bipartite graph.

Furthermore, it might be possible that a more fine-grained heuristic elimination procedure may prove to be worthwhile. In the context of this thesis we have not investigated any heuristic methods but rather elected to focus on the complexity of the problems in the general case. However, heuristic methods for reduction to triangular form while preserving sparsity has received attention in literature before and further research in this area for example based on our fine-grained approach would definitely be interesting from the point of view of practical applications.

Bibliography

[1] Project smart synthesis tools. `http://www.opm.ctw.utwente.nl/research/design_engineering/Project%20Smart%20Synthesis%20Tools.doc/index.html`. Online; accessed 18-November-2012.

[2] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, 1979.

[3] Richard M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, 1971. ACM.

[5] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, New York, 1999.

[6] Matthijs Bomhoff, Walter Kern, and Georg Still. On bounded block decomposition problems for under-specified systems of equations. *Journal of Computer and System Sciences*, 78(1):336–347, 2012.

[7] Matthijs Bomhoff and Bodo Manthey. Bisimplicial edges in bipartite graphs. *Discrete Applied Mathematics*, 2011. in press, DOI: 10.1016/j.dam.2011.03.004.

[8] Matthijs Bomhoff. Recognizing sparse perfect elimination bipartite graphs. In Alexander Kulikov and Nikolay Vereshchagin, editors, *Computer Science – Theory and Applications*, volume 6651 of *Lecture Notes in Computer Science*, pages 443–455. Springer, Berlin, 2011.

[9] Matthijs Bomhoff, Walter Kern, and Georg Still. A note on perfect partial elimination. Technical report, University of Twente, 2011.

[10] Claude Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, Amsterdam, 2nd edition, 1976.

[11] L. Lovász and M. D. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986.

[12] Armen S. Asratian, Tristan M. J. Denley, and Roland Häggkvist. *Bipartite Graphs and their Applications*. Cambridge University Press, Cambridge, 1998.

[13] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, 10(1):26–30, 1935.

[14] A. L. Dulmage and N. S. Mendelsohn. Coverings of bipartite graphs. *Canadian Journal of Mathematics*, 10:517–534, 1958.

[15] S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of constraint systems. In *Proceedings of Compugraphics 1993*, pages 83–92, 1993.

[16] Martin Charles Golumbic and Clinton F. Goss. Perfect elimination and chordal bipartite graphs. *Journal of Graph Theory*, 2(2):155–163, 1978.

[17] Kazuo Murota. *Matrices and Matroids for Systems Analysis*. Springer, Berlin, 2000.

[18] Kazuo Murota. *Systems Analysis by Graphs and Matroids*. Springer, Berlin, 1987.

[19] Georg Still, Walter Kern, Jaap Koelewijn, and Matthijs Bomhoff. Consistency of a system of equations: What does that mean? Technical report, University of Twente, 2010.

[20] Richard Caron and Tim Traynor. The zero set of a polynomial. `http://www.uwindsor.ca/math/sites/uwindsor.ca.math/files/05-03.pdf`. Online; accessed 18-November-2012.

[21] A. L. Dulmage and N. S. Mendelsohn. On the inversion of sparse matrices. *Mathematics of Computation*, 16(80):494–496, 1962.

[22] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Transactions on Mathematical Software*, 16(4):303–324, 1990.

[23] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[24] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[25] Christian Bliek, Bertrand Neveu, and Gilles Trombettoni. Using graph decomposition for solving continuous CSPs. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 102–116, Berlin, 1998. Springer.

[26] H. B. Mann and H. J. Ryser. Systems of distinct representatives. *The American Mathematical Monthly*, 60(6):397–401, 1953.

[27] Jianer Chen and Iyad A. Kanj. Constrained minimum vertex cover in bipartite graphs: complexity and parameterized algorithms. *Journal of Computer and System Sciences*, 67(4):833–847, 2003.

[28] Miroslav Chlebík and Janka Chlebíková. Crown reductions for the minimum weighted vertex cover problem. *Discrete Applied Mathematics*, 156(3):292–312, 2008.

[29] Christophe Jermann, Gilles Trombettoni, Bertrand Neveu, and Pascal Mathis. Decomposition of geometric constraint systems: a survey. *International Journal of Computational Geometry and Applications*, 16(05n06):379–414, 2006.

[30] Andrew Lomonosov. *Graph and Combinatorial Algorithms for Geometric Constraint Solving*. PhD thesis, University of Florida, Gainesville, 2004.

[31] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theoretical Computer Science*, 141(1-2):109–131, 1995.

[32] R. G. Downey and M. R. Fellows. Fixed-parameter intractability. In *Proceedings of the Seventh Annual Structure in Complexity Theory Conference, 1992*, pages 36–49, Boston, 1992. IEEE Computer Society.

[33] Benny Chor, Mike Fellows, and David Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In Juraj Hromkovič, Manfred Nagl, and Bernhard Westfechtel, editors, *Proceedings of WG 2004*, volume 3353 of *Lecture Notes in Computer Science*, pages 257–269, Berlin, 2005. Springer.

[34] Faisal N. Abu-Khzam, Rebecca L. Collins, Michael R. Fellows, Michael A. Langston, W. Henry Suters, and Chris T. Symons. Kernelization algorithms for the vertex cover problem: Theory and experiments. In Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick, editors, *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments (ALENEX 2004)*, pages 62–69, 2004.

[35] Faisal N. Abu-Khzam, Michael R. Fellows, Michael A. Langston, and W. Henry Suters. Crown structures for vertex cover kernelization. *Theory of Computing Systems*, 41(3):411–430, 2007.

[36] Christian Sloper. *Techniques in Parameterized Algorithm Design*. PhD thesis, The University of Bergen, Norway, 2006.

[37] Karl A. Abrahamson, Rodney G. Downey, and Michael R. Fellows. Fixed-parameter tractability and completeness IV: On completeness for $W[P]$ and PSPACE analogues. *Annals of Pure and Applied Logic*, 73(3):235–276, 1995.

[38] L. Goh and D. Rotem. Recognition of perfect elimination bipartite graphs. *Information Processing Letters*, 15(4):179–182, 1982.

[39] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[40] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, 1995.

[41] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[42] D. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In Ronald Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, New York, 1972.

[43] Loren Haskins and Donald J. Rose. Toward characterization of perfect elimination digraphs. *SIAM Journal on Computing*, 2(4):217–224, 1973.

[44] D. J. Kleitman. A note on perfect elimination digraphs. *SIAM Journal on Computing*, 3(4):280–282, 1974.

[45] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[46] Donald J. Rose and Robert Endre Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34(1):176–197, 1978.

[47] Jeremy P. Spinrad. Recognizing quasi-triangulated graphs. *Discrete Applied Mathematics*, 138(1-2):203–213, 2004.

[48] Marta Wilczkowiak, Gilles Trombettoni, Christophe Jermann, Peter Sturm, and Edmond Boyer. Scene modeling based on constraint system decomposition techniques. In *Proceedings of the Ninth IEEE International Conference on Computer Vision (ICCV 2003)*, volume 2, pages 1004–1010. IEEE Computer Society, 2003.

[49] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, 1994.

[50] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2(1):77–79, 1981.

[51] Assaf Natanzon, Ron Shamir, and Roded Sharan. A polynomial approximation algorithm for the minimum fill-in problem. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 41–47, New York, 1998. ACM.

About the Author

Matthijs Jacobus Bomhoff was born on February 19, 1982 in Emmen. Here he attended primary and secondary education where his interest in mathematics was kindled. After graduating in 1999 he decided to move to Enschede to study both Applied Mathematics and Computer Science at the University of Twente. In 2004 he completed his MSc in Computer Science with a master's thesis titled *The Best of Intentions* on the possible usage of user intentions in information retrieval. Together with four friends he founded Quarantainenet B.V. – a company specialized in network management and security – while working on his master's degree in Applied Mathematics. He finished his studies in mathematics in 2007 with a master's thesis titled *Additional Conditions on Full Components in Rectilinear Steiner Minimal Trees*.

Following the completion of his studies, he was fortunate enough to obtain a part-time PhD position at the Discrete Mathematics and Mathematical Programming group of the University of Twente, allowing him to combine work at his own company with further academical education in mathematics. His work as a PhD candidate ultimately resulted in this thesis.