Basic Research in Computer Science

Exact Algorithms for Variants of Satisfiability and Colouring Problems

Bjarke Skjernaa

BRICS Dissertation Series **DS-04-5**

ISSN 1396-7002 **November 2004**

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/04/5/`

Exact Algorithms for Variants of Satisfiability and Colouring Problems

Bjarke Skjernaa

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

Exact Algorithms for Variants of Satisfiability and Colouring Problems

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Bjarke Skjernaa
6th September 2004

Abstract

This dissertation studies exact algorithms for various hard problems and give an overview of not only results but also the techniques used.

In the first part we focus on Satisfiability and several variants of Satisfiability. We present some historical techniques and results. Our main focus in the first part is on a particular variant of Satisfiability, Exact Satisfiability. Exact Satisfiability is the variant of Satisfiability where a clause is satisfied if it contains exactly one *true* literal. We present an algorithm solving Exact Satisfiability in time $O(2^{0.2325n})$, and an algorithm solving Exact 3-Satisfiability, the variant of Exact Satisfiability where each clause contains at most three literals, in time $O(2^{0.1379n})$. In these algorithms we use a new concept of *k-sparse* formulas, and we present a new technique called *splitting loosely connected formulas*, although we do not use the technique in the algorithms.

We present a new program which generates algorithms and corresponding upper bound proofs for variants of Satisfiability, and in particular Exact Satisfiability. The program uses several new techniques which we describe and compare to the techniques used in three other programs generating algorithms and upper bound proofs.

In the second part we focus on another class of NP-complete problems, colouring problems. We present several algorithms for 3-Colouring, and discuss k-Colouring in general. We also look at the problem of determining the chromatic number of a graph, which is the minimum number, k, such that the graph is k-colourable. We present a technique using maximal bipartite subgraphs to solve k-Colouring, and prove two bounds on the maximum possible number of maximal bipartite subgraphs of a graph with n vertices: a lower bound of $\Omega(1.5926^n)$ and an upper bound of $O(1.8612^n)$. We also present a recent improvement of the upper bound by Byskov and Eppstein, and show a number of applications of this in colouring problems.

In the last part of this dissertation we look at a class of problems unrelated to the above, Graph Distinguishability problems. DIST_k is the problem of determining if a graph can be coloured with k colours, such that no non-trivial automorphism of the graph preserves the colouring. We present some results on determining the distinguishing number of a graph, which is the minimum k, such that the graph is in DIST_k. We finish by proving a new result which shows that DIST_k can be reduced to DIST_l for various values of k and l.

Acknowledgements

First, I would like to thank my supervisors, Peter Bro Miltersen and Sven Skyum, for the guidance and assistance they have provided during the last four years.

A special thanks goes to all the people at the University of Aarhus, and especially the secretaries and my co-authors, Bolette Ammitzbøll Madsen and Jesper Makholm Byskov. It has really been a pleasure working with you.

I would also like to thank all the people I met while staying at the Max Planch Institut für Informatik in Saarbrücken, Germany, and in particular the secretaries for helping and making my stay very easy, and my office mate, Sylvain Pion, for helping me, when I needed it.

I am not sure I would have made it this far without my family and all my friends, so I would also like to thank my parents and my sisters, everybody in Frokostklubben (The Lunch Club), the people on the fourth floor of Skejbygårdkollegiet, and everyone in Århus Klatreklub (Aarhus Climbing Club). Thanks for all the time we have spent together, and I hope we will continue to spend a lot of time together.

Finally, I would like to thank all the people who have proofread my papers, and in particular Bolette for proofreading this dissertation.

Bjarke Skjernaa,
Århus, 6th September 2004.

Contents

Part I

Overview

Chapter 1

Introduction

The interest in algorithms for NP-complete problems have increased during the last five to ten years. The speed of computers has made it possible to solve interesting instances, but the impact of increase in the speed of computers is not very large. An algorithm running in time $O(2^n)$ will only be able to handle inputs of size one greater every time the speed of computers doubles. On the other hand, if one creates an algorithm for the same problem which only uses time $O(2^{n/10})$, it will be able to handle instances which are ten times greater. So although the increase in the speed of computers does matter, better algorithms play a big role in solving real instances.

In this dissertation I will present exact algorithms for Satisfiability, variants thereof and for colouring problems. I will also present a result about the distinguishing number of a graph.

The dissertation consists of two parts. In Part I, I present a survey of results and techniques related to my results, and a brief overview of my own results. The second part consists of the papers I have authored or co-authored during my PhD studies.

In Chapter 2 I look at SAT and k-SAT, and present some algorithms for solving SAT and k-SAT. I end the chapter by looking at several variants of Satisfiability. In Chapter 3 I consider a specific variant of Satisfiability, Exact Satisfiability, and give an overview of previous results and techniques, and present my new result. In Chapter 4 I present prior work in the area of automatic generation of algorithms and upper bound proofs and sumarises the differences between the prior work and my new program, which have mainly been used for generation algorithms for Exact Satisfiability.

Finally I present in Chapter 6 an overview of results related to the distinguishing number of a graph. I present my result which shows some reductions between different distinguishablity problems.

In Part II I present the following papers:

Chapter 7 New Algorithms for Exact Satisfiability [10]
This paper is co-authored by Jesper M. Byskov and Bolette A. Madsen, and is to appear in Theoretical Computer Science.

Chapter 8 Automated Generation of Branching Algorithms with Upper Bound

Proofs for Variants of SAT [52]
This paper is single authored and is not yet published.

Chapter 9 On the Number of Maximal Bipartite Subgraphs of a Graph [11]
This paper is co-authored by Jesper M. Byskov and Bolette A. Madsen, and is to appear in Journal of Graph Theory.

Chapter 10 Reductions among Graph Distinguishability Problems [54]
This paper is single authored and is not yet published.

Chapter 2

Satisfiability

2.1 Introduction

In this chapter we will look at Satisfiability, which is probably the most well studied NP-complete problem.

Satisfiability (SAT)	
Input:	A boolean formula $F(x_1, \ldots, x_n)$ in conjunctive normal form (CNF).
Output:	Does there exist an assignment to the variables $x_1, \ldots, x_n$, such that the formula evaluates to true: $\exists a_1, \ldots, a_n : F(a_1, \ldots, a_n)$.

It was the first problem proven to be NP-complete, and this was done by Stephen A. Cook in 1971:

Theorem 2.1 (Stephen A. Cook [12], Theorem 1) *If a set, S, of strings is accepted by some nondeterministic Turing machine within polynomial time, then S is P-reducible to $\{$DNF tautologies$\}$.*

Note, that Cook does not distinguish between a problem and the complement of the problem, and thus the theorem is normally rephrased just as

Theorem 2.2 (Cook's Theorem) *SAT is NP-complete.*

Most algorithms do not consider the full Satisfiability problem, but consider instead the restricted problem of k-Satisfiability.

k-Satisfiability (k-SAT)	
Input:	A boolean formula $F(x_1, \ldots, x_n)$ in conjunctive normal form. Each clause contains at most k literals.
Output:	Does there exist an assignment to the variables $x_1, \ldots, x_n$, such that the formula evaluates to true: $\exists a_1, \ldots, a_n : F(a_1, \ldots, a_n)$.

It is easy to reduce SAT to 3-SAT, but on the other hand 2-SAT is easily shown to be in P.

We will first present a bit of notation and then some algorithms for solving SAT and 3SAT. When talking about formulas it is important to distinguish between a variable and a literal. When we talk about a variable, x, it is all occurrences of x, both negated and unnegated, while when we speak about a literal, x, we only consider x and not $\bar{x}$, which is another literal. A clause of the formula is called a *k-clause* if it contains exactly k literals. If σ is a, possible partial, assignment to the variables of F, then we denote by $F[\sigma]$ the formula F where every variable defined by σ is replaced by the corresponding value. For SAT and 3SAT we will let a *reduced formula* be a formula that is either just a constant or does not contain any constants at all. A formula in CNF can be translated into an equivalent reduced formula by removing every clause containing the constant *true*, and removing the constant *false* from every clause. If the formula after this translation contains an empty clause, the entire formula is replaced by *false*, and if it does not contain any clauses at all, it is replaced by *true*.

As is common in papers on exact algorithms, we will throughout this dissertation use $O(f)$ to denote a function that is of the same order as f, ignoring polynomial factors. Note that when used on exponential functions this only differs from the normal definition if exact numbers are used, as we with the normal definition have: $O(p(n) \cdot c^{\alpha n}) \subseteq O(c^{(\alpha+\epsilon)n})$.

2.2 Branching algorithms

Many of the algorithms which we will present in this dissertation is branching algorithms, also known as Davis-Putnam-type algorithms [16, 17]. Branching algorithms solves a problem by branching into several smaller problems, in such a way that the solution to the original problem can be computed in polynomial time, given the solutions to the smaller problems. The algorithm recursively solves each of the smaller problems, and combines the solution to get a solution for the original problem. If an instance can be solved in polynomial time, e.g. if it is smaller than a fixed threshold, it is solved without branching. In this way the problems forms a tree, where the root is the original problem and the leaves are the problems which was solved directly. For a problem instance P we denote the tree with P as the root, $T(P)$. The running time of a branching algorithm for a given problem instance P is within a polynomial factor of the number of leaves in $T(P)$.

2.2.1 Analysing branching algorithms

Let $\mu(P)$ be a measure of the size of the problem instance P. If the problem instances are formulas, we generally use one of three different measures of the size: the number of variables, the number of clauses and the number of literals, but other measures could be used. Unless otherwise specified we use the number of variables.

The results in this section origins from a manuscript by Kullmann and Luckhardt [33]. If the algorithm when finding the solution for P constructs the problems $P_1, P_2, \ldots, P_k$, we let $t_i = \mu(P) - \mu(P_i)$ for $i \in [k]$, where $[k]$ denotes the set $\{1, 2, \ldots, k\}$. We call $t = (t_1, t_2, \ldots, t_k)$ the *branching vector* of this branch, and associate a *characteristic polynomial* with this branching vector

$$\chi(t)(x) = 1 - \sum_{i=1}^{k} x^{-t_i}. \tag{2.1}$$

The characteristic polynomial is strictly increasing and has thus exactly one positive root, which we call the *branching value* of t and denote by $\tau(t)$. By α_t we denote $\log \tau(t)$. As each internal node in $T(P)$ has an associated branching vector, we can define

$$\tau(T(P)) = \begin{cases} 1 & \text{if } T(P) \text{ only has one node,} \\ \max\limits_{t \text{ in } T(P)} \tau(t) & \text{otherwise.} \end{cases} \tag{2.2}$$

With this definition we can make an upper bound on the number of leaves in $T(P)$

Theorem 2.3 (Kullmann and Luckhardt [33], Theorem 1) *Let α be the largest of the α_ts for t in $T(P)$. We then have*

$$\textit{number of leaves in } T(P) \leq \tau(T(P))^{\mu(P)} = 2^{\alpha \cdot \mu(P)}. \tag{2.3}$$

Proof. The proof is by induction in the size of the tree. If $T(P)$ only has one node the bound is trivial. Assume the bound is true for $P_1, \ldots, P_k$. We get

$$\begin{aligned} \text{number of leaves in } T(P) &\leq \sum_{i=1}^{k} \tau(T(P_i))^{\mu(P_i)} = \sum_{i=1}^{k} \tau(T(P_i))^{\mu(P)-t_i} \\ &\leq \sum_{i=1}^{k} \tau(T(P))^{\mu(P)-t_i} = \tau(T(P))^{\mu(P)} \cdot \sum_{i=1}^{k} \tau(T(P))^{-t_i} \\ &= \tau(T(P))^{\mu(P)} \cdot (1 - \chi(t_1, \ldots, t_k)(\tau(T(P)))) \leq \tau(T(P))^{\mu(P)}. \end{aligned} \tag{2.4}$$

The last inequality follows from the fact that $\chi(t_1, \ldots, t_k)$ is increasing and $\tau(T(P)) \geq \tau(t_1, \ldots, t_k)$. The equality in (2.3) follows directly from the definition of $\tau(T(P))$ and α. □

Thus, to determine an upper bound on the running time of a branching algorithm one proves an upper bound on the branching value for every possible branch the algorithm can make. The worst branching value gives an upper bound on the overall running time.

We will use some simple properties of $\tau(t)$, which all follows directly from the definitions of $\chi(t)$ and $\tau(t)$

Lemma 2.1 (Kullmann and Luckhardt [33], part of Lemma 5.1) *$\tau(t)$ has the following properties:*

1. *Permutations of the entries in t do not affect $\tau(t)$,*
2. *$\tau(t)$ is strictly decreasing in each component (when t has more than one entry),*
3. *$\tau(t_1, \ldots, t_k) < \tau(t_1, \ldots, t_k, t_{k+1})$ for $t_{k+1} > 0$.*

2.3 Algorithms for Satisfiability

SAT is trivially solvable in time $O(2^n)$ by testing whether any of the 2^n assignments to $x_1, \ldots, x_n$ makes the expression evaluate to true. Unfortunately, no algorithm obtaining a better time complexity stated as a function of the number of variables is known.

2.3.1 k-SAT

In the case of k-SAT, it is easy to make a slight improvement to the trivial bound by noting that looking at a particular clause only $2^k - 1$ of the 2^k possible assignments to its variables are feasible (assuming the clause has k literals). Thus resulting in an $O((2^k - 1)^{n/k})$ algorithm.

To improve this Monien and Speckenmeyer, in one of the first and most simple algorithms for k-SAT, noted that instead of making $2^k - 1$ recursive calls with k variables fewer in each, one can do with k recursive calls with respectively $1, 2, \ldots, k$ variables fewer:

Boolean SolveMonien(F):
  if F is empty **then**
    return *true*
  end if
  if F contains the empty clause **then**
    return *false*
  end if
  Let $c = (l_1 \vee l_2 \vee \cdots \vee l_i)$ be an arbitrary clause of F
  for $j = 1, \ldots, i$ **do**
    if SolveMonien(reduce($F[l_1 = \mathit{false}, \ldots, l_{j-1} = \mathit{false}, l_j = \mathit{true}]$)) **then**
      return *true*
    end if
  end for
  return *false*

where reduce($\cdot$) reduces the expression as described on page 6. It is easy to see that F is satisfiable if and only if at least one of these sub-formulas are satisfiable. The algorithm has a branching vector of (1,2,. . . ,k). Denote by β_k the corresponding branching value. The algorithm shows that k-SAT can be solved in $O(\beta_k^n)$, which for $k = 3$ is $O(1.8393^n)$. Monien and Speckenmeyer also present a trick which improves this such that k-SAT is solved in $O(\beta_{k-1}^n)$. The observation is that in each step, it is possible either to eliminate at least one variable without branching (if a variable only occurs either negated or unnegated) or ensure that in the next step there is a clause with only $k - 1$ literals. Eliminating one variable followed by branching on a k-clause is better

than branching on a $(k-1)$-clause, and thus is the second case worst, and we get the claimed complexity. For 3-, 4-, and 5-SAT the result is $O(1.6181^n)$, $O(1.8393^n)$ and $O(1.9276^n)$.

In Table 2.1 we have summarised most results for k-SAT (an entry with value c indicates an algorithm running in time $O(c^n)$), and we will describe some of the newest improvements. Except from a few of the entries, the table is taken from a paper by Schöning [48], where the precise references can be found.

3-SAT	4-SAT	5-SAT	Type	Reference
1.849	-	-	det.	P. Pudlák, 1998
1.782	1.835	1.867	det.	R. Paturi, P. Pudlák, F. Zane, 1997
1.619	1.840	1.928	det.	B. Monien, E. Speckenmeyer, 1985 [37]
1.588	1.682	1.742	prob.	R. Paturi, P. Pudlák, F. Zane, 1997
1.579	-	-	det.	I. Schiermeyer, 1993
1.505	-	-	det.	O. Kullmann, 1999 [31]
1.5	1.6	1.667	prob.	U. Schöning, 1999
1.447	1.496	1.569	prob.	Paturi, Pudlák, Saks, Zane, 1998
1.362	1.476	-	prob.	Paturi, Pudlák, Saks, Zane, 1998
1.334	1.5	1.6	prob.	U. Schöning, 1999 [51]
1.481	1.6	1.667	det.	Dantsin et al., 2000 [15][a]
1.331	-	-	prob.	Schuler, Schöning, Watanabe, 2001 [50]
1.331	-	-	prob.	Hofmeister et al., 2002 [29]
1.329	-	-	prob.	S. Baumer and R. Schuler, 2004 [4]

[a]The result for 4- and 5-SAT was found independently by several people [14].

Table 2.1: History of k-SAT.

The algorithm presented by Schöning [51] is a Monte Carlo algorithm with one-sided error, which means that the upper bound on the running time is correct, but there may be a little probability of either false negative answers or false positive answers (but not both). In this case, the algorithm may possibly give false negative answers: given a formula, there is a little probability that the algorithm will claim that the formula is not satisfiable, even though it is. The probability of error can be made negligible by adjusting a parameter of the algorithm. The algorithm is extremely simple to describe:

Boolean SolveProb(F):
  repeat $f(n)$ **times**
    Guess an initial assignment uniformly at random.
    repeat $3n$ **times**
      if the formula is accepted by the actual assignment **then**
        return *true*
      end if
      Choose an arbitrary clause that is not satisfied. Pick uniformly at random one of the variables of the clause. Switch the value of the variable.
    end repeat
  end repeat
  return *false*

The error probability (and the running time) depends on the choice of $f(n)$. If the probability of succeeding in one iteration of the outer loop is denoted $p(n)$, choosing $f(n) = \frac{1}{p(n)}$ will make the error probability less than e^{-1}. This probability can be made arbitrary small by repeating the algorithm (choosing $f(n) = \frac{s}{p(n)}$, results in an error probability of at most e^{-s}). The running time of the algorithm is $O(f(n))$, so to determine the running time it will suffice to determine $p(n)$. Using Markov chains and the Ballot Theorem [24], it can be shown that $p(n) \geq (\frac{1}{2}(1+\frac{1}{k-1}))^n$. The crucial observation is that the probability that the Hamming distance to a particular solution (if one exists) decreases in a step is at least $\frac{1}{k}$.

The conclusion is that the algorithm solves k-SAT in $O((\frac{1}{2}(1+\frac{1}{k-1}))^{-n}) = O((2(1-\frac{1}{k}))^n)$, which for 3-SAT, 4-SAT and 5-SAT is $O((\frac{4}{3})^n), O((\frac{3}{2})^n)$ and $O((\frac{8}{5})^n)$. The algorithm has by several people independently been derandomised by using covering codes [14]. Although the derandomisation increases the running time to $O((2(1-\frac{1}{k+1}))^n)$, it is the best known deterministic result for k-SAT, for $k > 3$.

The algorithm of Schöning was slightly improved for 3-SAT in 2001 by Schuler, Schöning and Watanabe [50] to $O(1.3303^n)$. Their algorithm is a hybrid of the probabilistic algorithm of Schöning and a new deterministic algorithm. Given a set of clauses, $\mathcal{F}$, the algorithm finds an arbitrary maximal independent subset of the 3-clauses $\mathcal{C} \subseteq \mathcal{F}$ (no variable appears in more than one clause of $\mathcal{C}$, and every 3-clause in $\mathcal{F} \setminus \mathcal{C}$ shares a variable with at least one clause in $\mathcal{C}$). If every variable of $\mathcal{C}$ is assigned a value, the reduced formula is a 2-SAT instance, which can be solved in polynomial time. Thus the following deterministic algorithm will decide Satisfiability:

Boolean SolveSchuler(F):
  for every assignment, σ, to the variables of $\mathcal{C}$, that satisfy all clauses of $\mathcal{C}$ **do**
    if $F[\sigma]$ is satisfiable **then**
      return *true*
    end if
  end for
  return *false*

The number of truth assignments to the variables of $\mathcal{C}$ that satisfy all clauses of $\mathcal{C}$ is exactly $7^{|\mathcal{C}|}$, and can easily be generated with only a small overhead. The size of $\mathcal{C}$ can be $\frac{n}{3}$, which gives the time bound $O(7^{\frac{n}{3}}) \approx O(1.9129^n)$. This is almost as bad as the trivial solution, so how can this algorithm be of any use? The key here is that the "bad" instances are the instances in which $|\mathcal{C}|$ is large compared to n. Those "bad" instances are instead solved by a probabilistic algorithm almost identical to the one by Schöning. The difference is that instead of choosing the initial assignments uniformly at random, the assignments depend on $\mathcal{C}$. For a clause c in $\mathcal{C}$ the assignment to its three variables is done as follows

- With probability $\frac{1}{7}$ make all three literals of c true.
- With probability $\frac{2}{7}$ make exactly two of the three literals true (select two uniformly random).

- With probability $\frac{4}{7}$ make only one of the literals true (a uniformly distributed random one).

The variables not occurring in $\mathcal{C}$ are just assigned a uniformly distributed random value. It can be shown that the time for running the above algorithm is $O((\frac{4}{3})^{n-3|\mathcal{C}|}(\frac{7}{3})^{|\mathcal{C}|})$. As $\frac{7}{3} < (\frac{4}{3})^3$, this is worst when $|\mathcal{C}|$ is small. When $|\mathcal{C}| \approx 0.146652n$ both algorithms run in time $O(1.3303^n)$. So the algorithm finds a $\mathcal{C}$ and if $|\mathcal{C}| > 0.146652n$ the randomised algorithm is used, and otherwise the deterministic is used. This results in a worst case complexity of $O(1.3303^n)$. Together with Hofmeister [29] they later improved this algorithm to $O(1.3302^n)$. The currently best known algorithm for 3-SAT is by Baumer and Schuler [4] and it is also an improvement of the above algorithm; it obtains $O(1.3290^n)$.

The best deterministic algorithm for 3-SAT is an algorithm from 2000 by Dantsin et al. [15] which runs in time $O(1.481^n)$. It is an improvement of the derandomised version of the algorithm by Schöning.

2.3.2 SAT in general

As mentioned in the beginning there is no known algorithm solving general formulas in time $o(2^n)$. So for general formulas it is more interesting to measure algorithms by a different parameter. The parameters normally used for measuring the running time of an algorithm is the length of the formula, l, and the number of clauses, m. We will only mention the best known result, which is by Hirsch [27]. He presents both an algorithm running in time $O(2^{0.30897m}) \approx O(1.23881^m)$ and one running in time $O(2^{0.10299l}) \approx O(1.07399^l)$. Although the algorithms are not very complicated, we would, in order to describe them in details, have to introduce a lot of new concepts, so we will not do this. An important part of the correctness of the algorithms is *the black and white principle*: let F be a formula in CNF, and $P_F(l)$ be a property of a literal satisfying that not both a variable and its negation satisfy the property (i.e. $P_F(l) \Rightarrow \neg P_F(\bar{l})$). An example of such a property is "l occurs exactly twice, and $\bar{l}$ occurs at least three times in F". At least one of the statements

- There is a clause, C, in F that contains a literal satisfying P_F and the negation of any literal from C does not satisfy P_F.
- F is satisfiable if and only if it has a satisfying assignment assigning false to all literals satisfying P_F.

holds. This ensures that it is either possible to reduce the formula at no significant cost, the formula contains no literal satisfying the property, or there exist a clause of a special form. Both algorithms are branching algorithms. The reductions are not the same, because the measurements of size are not the same. The branching step for the first algorithm (the one expressing the running time as a function of the number of clauses) either finds a variable which will give a branching vector (two branches) better than $\tau(6,7,6,7) \approx 1.23881$ or a pair of variables which will give a branching vector (four branches) better than $\tau(6,7,6,7)$. Because of the properties of the reduction, one of those

cases is always possible, giving the claimed running time. The second algorithm just uses the first one if all clauses have at least 3 literals (which implies $m \leq \frac{1}{3}l$). Otherwise, there exists a variable which when branching on that variable will give a branching vector better than $\tau(5, 17) \approx 1.07361$. The case where the first algorithm is called is the worst, resulting in a time complexity of $O(\tau(6, 7, 6, 7)^{\frac{1}{3}l}) \approx O(1.0740^l)$.

2.4 Variants of Satisfiability

In this chapter we will look at different variants of Satisfiability, and especially 3-Satisfiability, in order to determine the complexity of the variants, and how hard they are compared to each other.

There are two main results concerning the complexity of SAT and variants of SAT. The first is Cook's Theorem from 1971 and the second is from 1978 by Thomas J. Schaefer [45]. Schaefer considers the variants of Satisfiability which can be described by a, possibly infinite, set of relations, each of which have some number of free variables, such that each instance is the conjunction of clauses, where each clause is one of the relations with specific variables. To take an example, 3SAT can be described by the four relations $R_0(x, y, z) \equiv x \vee y \vee z$, $R_1(x, y, z) \equiv \bar{x} \vee y \vee z$, $R_2(x, y, z) \equiv \bar{x} \vee \bar{y} \vee z$ and $R_3(x, y, z) \equiv \bar{x} \vee \bar{y} \vee \bar{z}$. Schaefer classifies all variants of this kind as either NP-complete or in P.

Theorem 2.4 (Schaefer [45], Dichotomy Theorem for Satisfiability) *Let S be a finite set of logical relations. If S satisfies any of the conditions (a)-(f) below, then SAT(S) is polynomial-time decidable. Otherwise, SAT(S) is NP-complete.*

(a) Every relation in S is satisfied when all variables are false.

(b) Every relation in S is satisfied when all variables are true.

(c) Every relation in S is definable by a CNF formula in which each conjunction has at most one negated variable.

(d) Every relation in S is definable by a CNF formula in which each conjunction has at most one unnegated variable.

(e) Every relation in S is definable by a CNF formula having at most 2 literals in each conjunction.

(f) Every relation in S is the set of solutions of a system of linear equations over the two-element field $\{0,1\}$ (corresponding to true and false).

SAT(S) is the variant of SAT defined by the relations in S. If a relation satisfies any of the conditions it is normally very easy to prove this, but it is not always straightforward to prove that a relation does not satisfy any of the conditions. Fortunately Schaefer also describes alternate methods for determining whether (a)-(f) holds for a given variant of SAT, and these methods

makes proving that a relation does not satisfy the condition very easy. We will here present the methods without proof.

In the following we will let R denote a relation and $V(R)$ denote the variables occurring in R. For an assignment σ we write $\sigma \in R$ if the assignment satisfies the relation, and for two (partial) assignments σ and σ', we let $\sigma[\sigma']$ be as σ except where σ' is defined, where it is like σ'.

It is trivial to determine whether a relation satisfies (a) or (b). R satisfies (c) if and only if for all subsets V of the variables of R, the following holds true

$$(\forall v \in V(R) \setminus V \exists \sigma \in R : \sigma(V) = \{\mathit{false}\} \wedge \sigma(v) = \mathit{true}) \Rightarrow [V \mapsto \mathit{false}, V(R) \setminus V \mapsto \mathit{true}] \in R, \quad (2.5)$$

i.e. if there exists a satisfying assignment with all variables in V set to *false* and such a partial assignment do not force any other variable to be *false*, then it is a satisfying assignment to assign the variables in V *false* and all other variables *true*.

By swapping *true* and *false* in the above one can determine if a relation satisfies (d). In order to determine whether R satisfies (e), we introduce the notion of a *change set* of an assignment. For an assignment $\sigma \in R$, $V \subseteq V(R)$ is a change set of σ if $\sigma[\forall v \in V : v \mapsto \neg\sigma(v)] \in R$, i.e. if the assignment is still a valid assignment if the values of all the variables in V are changed. R satisfies (e) if and only if for any valid assignment the set of change sets of the assignment is closed under intersection,

$$\forall \sigma \in R, \forall V_1, V_2 \in \mathrm{ChangeSet}(\sigma) : V_1 \cap V_2 \in \mathrm{ChangeSet}(\sigma). \quad (2.6)$$

Finally, R satisfies (f) exactly if the set of valid assignments are closed under the operation of taking exclusive or between three elements, i.e.

$$\forall \sigma_1, \sigma_2, \sigma_3 \in R, [\forall v \in V(R) : v \mapsto \sigma_1(v) \oplus \sigma_2(v) \oplus \sigma_3(v)] \in R. \quad (2.7)$$

In this paper we will, unless otherwise specified, only consider the variants of SAT which Schaefer considers, i.e. we will not work with maximisation problems, or problems where the number of occurrences of literals are restricted. With this restriction we have in Table 2.2 listed all the variants of 3SAT for which the order of the literals in the clause do not influence whether a clause is satisfied.

The complexities of all the problems in Table 2.2 can be proved by using the Dichotomy Theorem, and we will only prove it for one of the problems, X3SAT, which is the topic of the next chapter. For X3SAT there are four relations, which as for 3SAT only differs in the number of negated variables, and we only need to look at one in order to prove the NP-completeness of X3SAT: $R(x, y, z) \equiv (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y}) \wedge (\bar{x} \vee \bar{z}) \wedge (\bar{y} \vee \bar{z})$. (a) and (b) are not satisfied, and when V is chosen to be the empty set, (2.5) is not true (and neither is it with *true* and *false* swapped) so (c) and (d) are not satisfied either. For the assignment $[x \mapsto \mathit{true}; y, z \mapsto \mathit{false}]$ the only change sets are {x,y} and {x,z} and the set of change sets is thus not closed under intersection, which shows that (e) is not satisfied. Finally note that there are three different

Is a clause satisfied by ... *true* literals?				Name	Complexity
0	1	2	3		
no	no	no	no	FALSE	P
yes	no	no	no	ALLFALSE*	P
no	yes	no	no	XSAT (Exact)	NPC
no	no	yes	yes	STRONG SAT	P/NPC
yes	no	yes	no	EVEN SAT*	P
yes	no	no	yes	AESAT* (All Equal)	P
no	yes	yes	no	NAESAT (Not All Equal)	NPC
no	yes	yes	yes	SAT	NPC
yes	no	yes	yes	NXSAT* (Not Exact)	NPC
yes	yes	yes	yes	TRUE	P

Problems marked with * are problems for which we have not been able to find a name used elsewhere. Note that by negating all literals a clause with 0 *true* literals becomes a clause with 3 *true* literals and vice versa and a clause with 1 *true* literal becomes a clause with 2 *true* literals and vice versa. All problems thus have a corresponding negated problem which have the same complexity (in some cases, e.g. NAESAT, a problem is equal to its own negated problem). We have only included either a problem or its negation. As all the problems in the table can be generalised to clauses with more than three literals, we have stated the general problem. Note that the generalisation of Strong SAT which we consider is where all clauses must have at least two literals *true*, and this is decidable in polynomial time when the clauses are restricted to contain at most three literals, but is NP-complete if the clauses are allowed to contain four literals.

Table 2.2: Variants of 3SAT

valid assignments, but if we take the exclusive or between them it is not a valid assignment, so (f) does not hold either, and by the Dichotomy Theorem we thus have that X3SAT is NP-complete. Note that we in fact have proved that even restricted to only using unnegated variables X3SAT is NP-complete. Also note that this is by no means the easiest way to prove that X3SAT is NP-complete.

2.4.1 Relative hardness

Not all NP-complete problems are equally difficult. Let P_1 and P_2 be two NP-complete problems, and let μ_1 and μ_2 be functions measuring the size of P_1 instances respectively P_2 instances. Assume we have a reduction R, from P_1 to P_2, such that there exists two constants, k_1 and k_2, and for all instances I of P_1 we have

$$\mu_2(R(I)) \leq k_1 \cdot \mu_1(I) + k_2. \tag{2.8}$$

We then have that if there exists an $O(2^{\alpha \cdot \mu_2(I_2)})$ algorithm for P_2 then there exists a $O(2^{k_1 \cdot \alpha \cdot \mu_1(I_1)})$ algorithm for P_1.

In this section we will compare the hardness of the NP-complete variants of SAT, which we presented in the previous section. First we look at the problems

measured by the number of variables, as this is the most well studied measure.

If we only look at 3SAT variants, it is easy to reduce any variant to 3SAT without increasing the number of variables. This is due to the fact that each 3SAT clause only "prohibits" one of the eight assignments to the variables. We can thus conclude, that 3SAT is the hardest and will always have the worst time complexity of these. Furthermore X3SAT can be reduced to NAE3SAT without increasing the number of variables, and it can thus be considered easier than NAE3SAT. I have not been able to find any other reductions between variants of 3SAT, which do not increase the number of variables linearly in the number of clauses, so they do not satisfy (2.8).

All of STRONG SAT, NAESAT, SAT and NXSAT can be reduced to each other with only a constant *additive* overhead in the number of variables, and they thus have the same time complexity. XSAT can also be reduced to any of these with only a constant additive overhead, but I have not been able to reduce any of the other problems to XSAT without increasing the number of variables by at least two times the number of literals in the formula: in XSAT each clause prohibits many assignments, while in the other variants each clause only prohibits a few assignments. When we translate a clause to XSAT clauses, none of the variables from the original clause can occur together in the new clauses, and we thus need to introduce a lot of extra variables.

In Table 2.3 is listed the minimum value of k_1 for which I have found a reduction between various variants of 3SAT, when we measure the size by either the number of clauses or literals. The numbers coincide, except when reducing

From \ To	X3SAT	NAE3SAT	NX3SAT	3SAT
X3SAT	–	4	5	4/3
NAE3SAT	8	–	4	2
NX3SAT	11	6	–	3
3SAT	4	2	2	–

Table 2.3: k_1 when the measure is the number of clauses/literals

from X3SAT to 3SAT, where a three clause becomes one three clauses and three two clauses. The greatest blowup is when reducing to X3SAT, which could indicate that X3SAT is the variant which is easiest to solve. The same results for the general problems are stated in Table 2.4. Question marks denote that we have not been able to find a reduction which satisfied (2.8). In most of these cases the reductions we found increased the size according to the measure by a factor depending on l, the number of literals. Note that there are only three different kinds of reductions with only a constant blowup; NAESAT and SAT can be reduced to any other variant, except XSAT when the measure is clauses, and when measured by the number of literals all variants can be reduced to XSAT. An interesting fact which can be retrieved from the table is that with respect to the number of clauses SAT is easier than all other variants, except XSAT. Also, we can conclude that in most cases XSAT behaves differently from the other variants.

From \ To	XSAT	NAESAT	NXSAT	Strong SAT	SAT
XSAT	–	?	?	?	?
NAESAT	?/8	–	$2/\frac{8}{3}$	$2/\frac{8}{3}$	2
NXSAT	$?/\frac{44}{3}$	?	–	?	?
Strong SAT	?/6	?	?	–	?
SAT	?/4	$1/\frac{3}{2}$	$1/\frac{3}{2}$	$1/\frac{3}{2}$	–

Table 2.4: k_1 when the measure is the number of clauses/literals

Chapter 3

Exact Satisfiability

3.1 Introduction

In the last chapter we briefly introduced Exact Satisfiability

Exact Satisfiability (XSAT)	
Input:	A boolean formula $F(x_1, \ldots, x_n)$ in conjunctive normal form (CNF).
Output:	Does there exist an assignment to the variables $x_1, \ldots, x_n$, such that each clause contains exactly one *true* literal?

In this chapter we will go into a detailed study of this problem. The variant of XSAT where each clause contains at most k literals is denoted XkSAT.

As shown in the previous chapter, XSAT is NP-complete even when restricted to clauses containing at most three literals and all variables occurring only unnegated [45]. X2SAT is trivially solvable in polynomial time and here we prove that XSAT with all variables occurring at most twice can also be solved in polynomial time. The original proof is by Monien, Speckenmeyer and Vornberger [39],[1] but the proof we present here is by Porschen, Randerath and Speckenmeyer [42]. We reduce the problem to Perfect Matching. First we remove all variables which occur both negated and unnegated by resolution: (x, C) and $(\bar{x}, C')$ are replaced by the single clause (C, C'). We can then assume that the formula contains no negated variables. If the instance does not contain any unique variables, we can just make the graph where each vertex corresponds to a clause, and for every variable we connect the two vertices corresponding to the clauses containing the variable (thus two vertices are connected if the corresponding clauses share a variable):

$$V = \{v_C | C \in F\}, \tag{3.1}$$

$$E = \{e_v | v \in V(F), e_v = (v_{C_1}, v_{C_2}), v \in C_1 \cap C_2\}. \tag{3.2}$$

[1] They state that a generalised version of XSAT with variables occurring at most twice, called $MAX(\{\leq, =, \geq\}, \cdot, 2)$, reduces to Perfect Matching. The proof is in the technical report [38].

This graph has a perfect matching exactly if the formula can be exactly satisfied: If we have a perfect matching we can for each selected edge set the corresponding variable to *true*. We assign all other variables the value *false*. This is a satisfying assignment, as each clause has exactly one variable set to *true*. On the other hand, if we have a satisfying assignment, and we take all the edges corresponding to the *true* variables, we get a perfect matching.

If the instance has unique variables the graph we construct is a bit more complicated. It consists of two copies of the graph just described, and for each unique variable, we connect the two copies of the vertex corresponding to the clause containing the unique variable. See Fig. 3.1 for an example. Once again

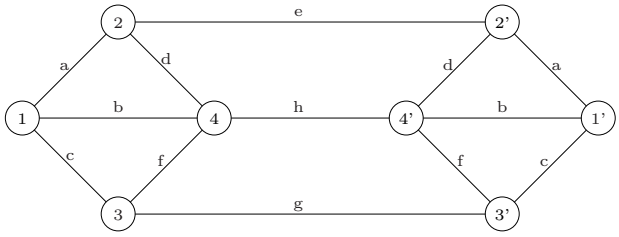


Figure 3.1: Example of reduction from XSAT, where each variable occurs at most twice, to Perfect Matching. The formula reduced is (a, b, c), (a, d, e), (c, f, g), (b, d, f, h).

there exists a perfect matching exactly if the formula is satisfiable: If we have a perfect matching then consider the selected edges in *one* of the copies and the selected edges between the two copies and set the variables corresponding to these edges to *true*, and the other variables to *false*. This gives a satisfying assignment. If we on the other hand have a satisfying assignment we can select all edges corresponding to the *true* variables (for each unique variable there is one corresponding edge and for the others there are two), and this is a perfect matching.

3.2 Algorithms for Exact Satisfiability

The first to present a nontrivial algorithm for XSAT was Schroeppel and Shamir [49], who in 1981 solved a class of problems, of which XSAT and Knapsack are the most prominent. Their algorithm uses $O(2^{n/2})$ time and it uses $O(2^{n/4})$ space. In the following we will describe this algorithm. Given a formula F with m clauses, we define for every partial assignment, σ, to the variables of F an m-tuple, $T(\sigma)$, such that the ith entry in $T(\sigma)$ is the number of *true* literals in the ith clause of F. Note that if σ_1 and σ_2 are two disjoint partial assignments, then $T(\sigma_1 \cup \sigma_2) = T(\sigma_1) + T(\sigma_2)$. In order to describe the algorithm we will start by describing an algorithm using $O(2^{n/2})$ time *and* space. The set of variables is divided into two sets, V_1 and V_2, of equal size. For each of these sets $T(\sigma)$ is calculated for every assignment to the variables of the set. This result in two sets of m-tuples, T_1 and T_2. The formula can be satisfied if and only if there exists two tuples $t_1 \in T_1$ and $t_2 \in T_2$, such that $t_1 + t_2 = (1, 1, \ldots, 1)$. By using

the lexicographic order on the tuples, we sort T_1 in increasing order and T_2 in decreasing order. By noting that if $t < t_1 + t_2$ and $t_1 < t_3$, then $t < t_3 + t_2$, and similar if $t > t_1 + t_2$ and $t_1 > t_3$, then $t > t_3 + t_2$, it is straightforward to prove that XSATsolve1, presented in Alg. 3.1, is correct. The time used by

```
/* T_1 is sorted in ascending order */
/* T_2 is sorted in descending order */
Tuple t_1 = T_1.first(), t_2 = T_2.first()
while t_1 ≠ NULL ∧ t_2 ≠ NULL do
  if t_1 + t_2 = (1, 1, ..., 1) then
    return true /* F is satisfiable */
  else if t_1 + t_2 < (1, 1, ..., 1) then
    t_1 = T_1.next(t_1)
  else
    t_2 = T_2.next(t_2)
  end if
end while
return false /* F is not satisfiable */
```

Algorithm 3.1. XSATsolve1

the algorithm is bounded by generating and sorting the two lists, both of which take time $O(2^{n/2})$, and the space used is also $O(2^{n/2})$. In order to decrease the space usage, note that we do not actually need to store T_1 and T_2, but only need to access the elements in sorted order. Here we present a method for generating T_1 in sorted order, using only $O(2^{n/4})$ space. T_2 can be handled similar. Split V_1 into two sets, V_{11} and V_{12}, of equal size, and calculate T_{11} and T_{12} in the same way as T_1 and T_2. Note that $T_1 = \{t_{11} + t_{12} | t_{11} \in T_{11}, t_{12} \in T_{12}\}$. Instead of generating T_1 we make a priority queue with all pairs (t_{11}, t_{12}), but we do not insert all elements at once. By using that if $t_{11} < t'_{11}$, then $t_{11} + t_{12} < t'_{11} + t_{12}$, we note that if $t_{11}, t'_{11} \in T_{11}$ and $t_{11} < t'_{11}$, then there is no need to add (t'_{11}, t_{12}) to the priority queue before (t_{11}, t_{12}) has been removed from the queue. Thus if T_{11} is sorted in ascending order, then initially we add $(T_{11}.first(), t_{12})$ for every $t_{12} \in T_{12}$, and when (t_{11}, t_{12}) is removed from the queue we add $(T_{11}.next(t_{11}), t_{12})$ to the queue. In this way we only use $O(2^{n/4})$ space for both of the lists and the priority queue. This algorithms is a proof of the fact that it is important to not just forget old algorithms, even though much better algorithms exists: recently Williams [56] used the ideas from this algorithm in the first algorithms solving MAX-2-SAT and MAX-CUT in $o(2^n)$

Also in 1981, Monien, Speckenmeyer and Vornberger [39] gave a polynomial space algorithm solving XSAT in time $O(2^{0.2441n})$, corresponding to a branching vector of $(11, 1)$. This was done with a pure branching algorithm. As the cases analysis are seven pages we will not describe it in this paper. In Chapter 7 we present the following result, which is the first, and so far the only, improvement of the result of Monien et al. for the general problem.

Theorem 3.1 (Theorem 7.1) *XSAT can be solved in time $O(2^{0.2325n})$, cor-*

responding to a branching vector of $(8, 2)$.

Note that several authors [13,18] have missed the result by Monien, Speckenmeyer and Vornberger and have thus presented algorithms for XSAT which they claim are improvements over the previous best known. Our result is obtained by making a more detailed case analysis, and by adding a new concept of k-*sparse* formulas. A formula is k-sparse if the number of variables occurring at least three times is at most n/k. To decide if a k-sparse formula is satisfiable, we can enumerate all possible truth assignments to these at most n/k variables; for each assignment, all variables in the remaining part of the formula occur at most twice, so we can decide in polynomial time, if it is satisfiable. The running time of this algorithm is $O(2^{n/k})$. To obtain our result for XSAT we use this with $k = 5$. It should be mentioned that Dahllöf, Jonnson and Beigel [13] independently have used the same technique, with $k = 9/2$, to get an inferior algorithm for XSAT

For X3SAT the general result of Monien et al. was not improved until 1999, when Drori and Peleg [18] presented an algorithm running in time $O(2^{0.2072n})$, corresponding to a branching vector of $(7, 4)$. In 2002 Hirsch and Kulikov [28] and Porschen, Randerath and Speckenmeyer [42] independently of each other found an algorithm running in time $O(2^{0.1626n})$, corresponding to a branching vector of $(9, 4)$. In 2004 Dahllöf, Jonnson and Beigel [13] presented an algorithm which they claimed was running in time $O(2^{0.1532n})$ (corresponding to $(10, 4)$), but as there is an error in the proof, the result is not valid. We will not describe the details of any of these algorithms, as they are all branching algorithms with increasingly complicated analyses. In Chapter 7 we present a new algorithm and prove the following theorem

Theorem 3.2 (Theorem 7.1) *X3SAT can be solved in time* $O(2^{0.1379n})$*, corresponding to a branching vector of* $(12, 4)$.

Our result is also obtained by adding quite a few new reductions and considering the cases in more detail, but does also use the concept of k-sparse, with $k = 15/2$.

In Table 3.1 all results for XSAT and X3SAT are summarised.

Paper	XSAT	X3SAT
Schroeppel and Shamir, 1981 [49]	$2^{n/2}$	
Monien, Speckenmeyer and Vornberger, 1981 [39]	$2^{0.2441n}$	
Drori and Peleg, 1999 [18]	$2^{0.3212n}$	$2^{0.2072n}$
Hirsch and Kulikov, 2002 [28]		$2^{0.1626n}$
Porschen, Randerath and Speckenmeyer, 2002 [42]		$2^{0.1626n}$
Dahllöf, Jonnson and Beigel, 2004 [13]	$2^{0.2519n}$	
Byskov, Madsen and Skjernaa, 2004 [10] (Chapter 3)	$2^{0.2325}$	$2^{0.1379n}$

Table 3.1: Results for XSAT and X3SAT.

Recently several variants of XSAT have been considered. Dahllöf, Jonnson and Beigel [13] considered the counting problems #XSAT and #X3SAT, which

are the problems of determining not just whether a formula can be exactly satisfied, but the number of assignments which exactly satisfies a formula. For #XSAT they present an algorithm running in time $O(4^{n/7}) \approx O(2^{0.2858n})$, and for #X3SAT they obtain a running time of $O(2^{n/5})$.

Another variant is to use other measures of the size than the number of variables, e.g. the number of literals or the number of clauses. For X3SAT there is no difference in measuring the number of literals and the number of clauses, as any formula with a clause with fewer than three literals can be reduced, and we can thus assume that every clause have size exactly three. It is very simple to get a branching vector of $(5, 3)$ with respect to clauses: if any clause contains more than one unique variable we can reduce the formula, and if two clauses share more than one variable we can also reduce the formula (these reduction can be found in Chapter 7). If every variable in the formula occurs at most twice, we can solve the problem in polynomial time. Otherwise we branch on a variable x occurring at least thrice. If x occurs both negated and unnegated another non-unique variable will get a value in both branches, and thus at least four clauses will be removed in each branch, giving a branching vector of $(4, 4)$ which is better than $(5, 3)$. If x only occurs positive or negative, we only remove the three clauses containing x in one branch, but in the other branch all variables occurring with x would get a value, and they must occur in at least two other clauses, which are then also removed (if they only occur in one other clause, these four clauses do not share any variables with the rest of the formula, and the formula can thus be reduced). This gives a branching vector of at least $(5, 3)$, corresponding to $O(2^{0.2557m})$, where m is the number of clauses. In Chapter 4 we present an automatically generated upper bound of $O(2^{0.2123m})$ corresponding to a branching vector of $(7, 3)$.

For XSAT there is a big difference between the measures. If we measures the number of literals, it is trivial to prove a branching vector of $(12, 3)$, and it is not hard to obtain a worst case branching vector of $(19, 3)$ by just branching on a variable occurring at least thrice. We will not show this, but it can be shown by an analysis similar to the one above, just with a few more cases.

On the other hand there does not exist any trivial bound on solving XSAT with respect to the number of clauses. Madsen [35] has shown an upper bound of $O(m!)$: for a permutation of the clauses, $c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}$, where $\pi \in \Pi_m$, we say that variable x_i occurs in an interval from j to j', with $j \leq j'$, if the clauses which contain x_i are exactly $c_{\pi(j)}, c_{\pi(j+1)}, \ldots, c_{\pi(j')}$. We denote this by $x_i \vDash \pi[j, j']$. If the formula is satisfiable, let T be a subset of the variables such that the formula is satisfied by setting the variables in T to *true* and all other variables to *false*. We can permute the clauses such that any variable in T occurs in an interval. Note that the intervals must be disjoint and cover all clauses. On the other hand if for some permutation of the clauses there is a subset of the variables which all occur in intervals, and the intervals are disjoint and cover all clauses the formula is satisfiable by setting the variables in the subset to *true*. For any permutation of the clauses we can in polynomial time decide if there is such a subset of the variables, e.g. by checking if there exists

a path from s to t in the following graph:

$$\begin{aligned}V =&\{s,t\} \cup \{v_x | x \in Var\}\\ E =&\{(s,v_x)|\exists j' : x \vDash \pi[1,j']\} \cup\\ &\{(v_{x_1},v_{x_2})|\exists j,j',j'' : x_1 \vDash \pi[j,j'] \wedge x_2 \vDash \pi[j'+1,j'']\} \cup\\ &\{(v_x,t)|\exists j : x \vDash \pi[j,m]\}\end{aligned}$$

It is thus possible to decide if a formula is satisfiable by trying all permutations of the clauses, which gives a running time of $O(m!)$.

Here we present a new algorithm which improves this to $O(2^m)$ at the cost of also using $O(2^m)$ space. First we show that every formula can be transformed to a formula where all variables only occur positive without increasing the number of clauses. If a variable x only occurs negative, we replace all occurrences of $\bar{x}$ with x, which does not change whether the formula is satisfiable. If a variable x occurs in two clauses (x, C) and $(\bar{x}, C')$, we can remove the variable from the formula by replacing every occurrence of x with C' and every occurrence of $\bar{x}$ by C. If we have a satisfying assignment to the original formula, we have that x is *true* if and only if C' contains exactly one *true* literal. We do thus not change whether a clause is satisfied by replacing x with C', and similar for $\bar{x}$ and C. On the other hand if we have a satisfying assignment to the new formula, exactly one literal in either C or C' is *true* (as there is a clause (C, C')). By setting x to *true* if and only if a literal in C' is *true*, we see that the original formula is also satisfiable. This reduction does not increase the number of clauses, and it does in fact decrease the number by at least one, as (x, C) and $(\bar{x}, C')$ becomes identical after the replacement. On a side note, the number of variables is also decreases by one. This reduction is due to Monien, Speckenmeyer and Vornberger [39].

The algorithm solving XSAT in time and space $O(2^m)$ for formulas without negations, is shown in Alg. 3.2. The main idea in the algorithm is to mark all subsets of the clauses, which satisfies a certain property. A subset S should be marked if and only if there exists an assignment such that all clauses in S have exactly one *true* literal, and all other clauses only contains *false* literals. We will show that XSATSOLVE does in fact mark exactly those clauses. First, we use induction to show that only subsets satisfying the property are marked. As all literals are unnegated, the empty set clearly satisfies the property. If S is marked and x does not occur in S, then does $S \cup \text{ClausesWith}(x)$, where $\text{ClausesWith}(x)$ are the clauses containing x, also satisfy the property and it is thus correct to mark it. On the other hand, if we assume there are subsets which satisfy the property, but which the algorithm does not mark, we can let S be a minimal, with respect to size, such subset. Now, S is not the empty set, so any assignment which proves that it satisfy the property, must have at least one variable, x, set to *true*. If we take such an assignment and change the value of x to *false*, the new assignment still have at most one literal *true* in each clause of the formula, and the set of clauses with one literal *true* is a subset of S. As S was chosen minimal, this subset is marked, but then S would also have been marked by the algorithm, which is a contradiction. We have now proved that the algorithm correctly marks all subsets which satisfy the property, and

```
Boolean XSATsolve(Formula F):
  Boolean satisfiable[SubsetsOf(F)] = {false, false, ...}
  satisfiable[∅] = true
  for s = 0 upto m do
    for all S ⊂ F, |S| = s do
      if satisfiable[S] then
        for all x ∈ Var(F) do
          if S ∩ ClausesWith(x) = ∅ then
            satisfiable[S ∪ ClausesWith(x)] = true
          end if
        end for
      end if
    end for
  end for
  return satisfiable[F]
```

Algorithm 3.2. XSATsolve

F is thus satisfiable exactly if F is marked. We have thus proved

Theorem 3.3 *XSAT can be solved in time $O(2^m)$ using space $O(2^m)$.*

3.3 Problems related to XSAT

There are two problems which are very closely related to XSAT, Exact Hitting Set

Exact Hitting Set	
Input:	A set S, and a set C of subsets of S: $C \subseteq 2^S$.
Output:	Does there exist a subset S' of S such that S' intersects every set in C in exactly one element: $$\exists S' \subseteq S : \forall c \in C : \lvert S' \cap c \rvert = 1.$$

and Exact Cover

Exact Cover	
Input:	A set S, and a set C of subsets of S: $C \subseteq 2^S$.
Output:	Does there exist a subset C' of C such that the union of the sets in C' is S and none of the sets in C' intersects: $$\exists C' \subseteq C : \bigcup_{c \in C'} c = S \wedge \forall c, c' \in C' : c \cap c' = \emptyset.$$

These two problems are dual to each other: if we have an instance (S, C) to either of the problems, $(C, \cup_{s \in S}\{\{c \in C | s \in c\}\})$ has the same solution, when

seen as an instance of the other problem. The second parameter is a subset for each element of S, and the subset is the subsets of C which contain the element.

These problems are related to XSAT, as Exact Hitting Set is more or less the same problem as XSAT, just with another representation of the input. If we have a formula without negations, each clause is just a subset of the set of variables, and seen in this way it is an instance of Exact Hitting Set, and vice versa. As was shown in the previous section we can transform a formula with negations to another formula without negations, without increasing the number of variables or clauses, and we can thus conclude that measured by the number of variables or clauses, XSAT is identical to Exact Hitting Set measured by the size of S or C.

3.4 Loosely connected formulas

In this section I will present the concept of *splitting k-loose formulas*, which we used in a preliminary version of the paper presented in Chapter 7, and which we believe could be used in other algorithms. The technique can be used for all variants of SAT described in Section 2.4, but does not work for variants like e.g. MAXSAT.

If a formula is disconnected, i.e. it consists of two parts not sharing any variables, we can check Satisfiability by checking it for the two parts independently. Here we present a technique for doing something similar if the formula is "almost disconnected". We say that a formula F is *k-loose*, if it is possible making the formula disconnected by removing all occurrences of at most k variables. In other words if there exists two subsets of the variables, V_1 and V_2, such that for every clause, all the variables in the clause are in V_1 or all the variables in the clause are in V_2, and $|V_1 \cap V_2| \leq k$. For a subset, V', of the variables in F, we let $F(V')$ denote the clauses of F, which only contains variables from V'. Let V_1 and V_2 satisfy the property above, and let $V_\cap = V_1 \cap V_2$. By checking satisfiability of $F(V_1)$ for all possible truth assignments to $V_\cap$, we can construct a "small" formula F', with variables from $V_\cap$, and possibly some new variables, such that for every assignment, σ, to $V_\cap$, $F(V_1)[\sigma]$ is satisfiable if and only if $F'[\sigma]$ is. Then F is satisfiable if and only if $F(V_2) \cup F'$ is. We can thus solve F by solving $F(V_1)$ 2^k times and then solve $F(V_2)$ with some additional variables and clauses.

For this splitting to be useful F' should be smaller than $F(V_1)$. For SAT it is always possible to construct F' without using any additional variables. For a variant of SAT, P, we will use $\nu_{\mathrm{P}}(k)$ to denote the maximum number of extra variables needed to encode any truth-table over k variables, and we thus have $|F'| \leq k + \nu_{\mathrm{P}}(k)$. There is no known easy way to calculate $\nu_{\mathrm{P}}(k)$. Trivially $\nu_{\mathrm{3SAT}}(0) = \nu_{\mathrm{3SAT}}(1) = \nu_{\mathrm{3SAT}}(2) = \nu_{\mathrm{3SAT}}(3) = 0$, and with the aid of a computer we have determined that $\nu_{\mathrm{3SAT}}(4) = 2$ and $\nu_{\mathrm{3SAT}}(5) = 3$. For XSAT and X3SAT we have determined that $\nu_{\mathrm{XSAT}}(1) = \nu_{\mathrm{X3SAT}}(1) = 0$, $\nu_{\mathrm{XSAT}}(2) = \nu_{\mathrm{X3SAT}}(2) = 1$ and $\nu_{\mathrm{XSAT}}(3) = \nu_{\mathrm{X3SAT}}(3) = 6$.

Theorem 3.4 *Given a formula, F, it is polynomial time decidable whether, for a positive constant k, F is k-loose with a "partitioning", V_1 and V_2, satis-*

fying $|V_1 \cap V_2| + \nu_P(k) < |V_1| \leq |V_2|$. *Furthermore, if* F *is* k*-loose with such a partitioning, performing loose splitting is better than any branching.*

Proof. A formula, F, is k-loose exactly if there exists V_1 and V_2 with $F = F(V_1) \cup F(V_2)$ and $|V_1 \cap V_2| \leq k$. We want to find such V_1 and V_2 which furthermore must satisfy $|V_1 \cap V_2| + \nu_{\mathrm{P}}(k) < |V_1| \leq |V_2|$.

In order to find V_1 and V_2, we construct the graph which has the variables as vertices and two vertices are connected if the two corresponding variables occur in a clause together. We need to check if removing at most k vertices can disconnect the graph into two parts with at least $\nu_{\mathrm{P}}(k)+1$ vertices in each. The existence of such a set of vertices $V_\cap$ can be determined in polynomial time by trying all possible subsets of size at most k, and V_1 and V_2 can easily be constructed from the connected components of the graph (and $V_\cap$) such that they both have size at least $\nu_{\mathrm{P}}(k) + |V_\cap| + 1$.

Let $V_\cap = V_1 \cap V_2$. When performing loose splitting we make 2^k recursive calls on a problem containing the variables from $V_1 \setminus V_\cap$ and one on a problem containing the variables from V_2 and possibly $\nu_{\mathrm{P}}(k)$ additional variables.

Now $|V_2| = n - (|V_1| - |V_\cap|)$ so the recursion for this branch is

$$T(n) = 2^k \cdot T(|V_1| - |V_\cap|) + T(n - (|V_1| - |V_\cap|) + \nu_{\mathrm{P}}(k)) + O(1).$$

Let N be some constant and suppose $|V_1| - |V_\cap| \leq N$; then we have 2^k recursive calls on a formula of size at most N, which is a constant, so these can be solved in constant time, and a single recursive call on a non-constant, but smaller formula. This is clearly better than any exponential running time. Now, if $|V_1| - |V_\cap| > N$ then also $|V_2| > N$, and thus $|V_1| - |V_\cap| < n - N$. We thus branch on 2^k subproblems that are at least N smaller and one that is $N - \nu_{\mathrm{P}}(k)$ smaller. By choosing N large enough we get that this is better than $2^{\alpha n}$ for any α. □

Chapter 4

Automated Generation of Algorithms

Automated generation of algorithms for NP-complete problems is a very new topic in computer science. Until recently there existed no work in the area, but in 2003 three groups independently published results within the area.

Jens Gramm, Jiong Guo, Falk Hüffner and Rolf Niedermeier [25] were the first to present a framework for automatic generation of branching algorithms for NP-hard problems. The class of problems they consider are graph modification problems. Graph modification problems are problems where a graph and an integer, k, are given, and the problem is to determine if the graph can be made to satisfy some local property, by performing at most k modifications, where each modification can be adding or deleting an edge or deleting a vertex. For a specific problem it is not necessarily all three kinds of modifications that are allowed.

They present two approaches, where the first, and most simple, just enumerates all subgraphs of a certain fixed size which do not satisfy the local property (i.e. subgraphs which have to be modified). For each subgraph they calculate all branching vectors which can be obtained by doing one or more branchings (i.e. they do not limit the level of branchings) on either performing or not performing a specific modification. In each branch they use a fixed set of problem specific reductions. Their second approach is a bit more like the approach we present in Chapter 8. They start with a minimal subgraph not satisfying the local property, and then expand the subgraph into more cases (corresponding to splitting the set of graphs containing the subgraph into several smaller sets). They repeat this until the subgraph has a certain fixed size, which they then find all possible branching vectors for. When doing the expansion they use some problem specific assumptions such that a subgraph is not expanded into all subgraphs of size one larger containing the original subgraph. The assumptions they use are assumptions which they can make on the subgraph, as if the subgraph does not satisfy the assumption it is possible to make a "good" branch. The assumptions can often be applied in different ways when expanding a subgraph and by checking all the possible expansions they can take the expansion that gives the best worst branching vector. As the only previous paper on automated generation of algorithms they have, to some extend, avoided duplicate cases by using dynamic programming and a test of graph isomorphism. With this method they avoid solving exactly the same

subgraph more than once, but they do not avoid solving cases which are just subcases of already handled cases. As the bottleneck in their program is finding all possible branching vectors (they do avoid finding some by using property (2) and (3) of Lemma 2.1), avoiding finding all branch vectors for subcases of already solved cases, could help removing this bottleneck.

The approach taken by Nikolenko and Sirotkin [41] is very simple. They want to prove an upper as a function of the number of clauses on solving SAT. But contrary to the other papers presented here they do not try to compete with the best known upper bound. Instead they try to prove an upper bound for a very simple algorithm, namely an algorithm which only uses unique literal elimination. Unique literal elimination is a reduction that states that if a variable only occurs either unnegated or negated, then the formula can be reduced by removing all clauses containing the variable. They prove a lower bound on $\Omega(3^{n/3})$, corresponding to $(3,3,3)$, and note that their program can thus assume that all variables are either (1,1)- or (1,2)-occurrences, as it would otherwise be possible to make a better branching. Their program then constructs an algorithm solving it in $O(3^{n/3})$, giving a tight bound.

The framework developed by Fedin and Kulikov [23] is the one that is most similar to the framework we use in Chapter 8. They consider three different problems, SAT, MAXSAT and $(n,3)$ MAXSAT (MAXSAT where each variable occurs at most thrice), and they consider three measures of the size, the number of variables, the number of clauses and the number of literals. They use a fixed set of reductions and also restrict the number of cases by using assumptions on which kind of (a,b)-occurrences can occur in the formula. An *(a,b)-occurrence* is a variable which occurs a times unnegated and b times negated or vice versa. We write $a+$ instead of a if it is at least a times. If they aim at proving a worst case branching value of α and branching on an $(a+,b+)$-occurrence immediately gives a better branching value, they assume the input does not contain any $(a+,b+)$-occurrences. In Chapter 8 we do not use such assumptions for two reasons. One reason is that one goal of the project was to see if a computer could prove new bounds without human aid, and another reason is that we do not aim at a fixed bound and adding assumptions would put a bound on how good bounds the program could prove. The bound with respect to the number of clauses Fedin and Kulikov obtain for SAT are actually bounded by their assumptions (they assume the formulas do not contain (2,4)-occurrences, as branching on these results in a branching value of 1.2721, but with this assumption all the cases they get have a branching value less than that). They do not mention if they have tried to use fewer assumptions such that the program had a chance of improving the bound.

In Chapter 8 we present a program generating algorithms and upper bound proofs of their running time, primarily for XSAT and X3SAT, but also for SAT. In Table 4.1 we present an overview over all the result obtained by the programs presented in this chapter, and as in the previous table, an entry c indicates a bound of $O(c^{\mu}(P))$. For the graph modification problems, the measure of size is the number of allowed modifications. Our primary goal was to see if a computer could prove bounds as good or better than humans without being taught a lot of reductions and assumptions. Thus in contrast to the other approaches

Paper	Problem	Previous best known result	Result
Gramm et al. [25]	Cluster Editing	2.27	1.92
Gramm et al. [25]	Cluster Deletion	1.77	1.53
Gramm et al. [25]	Cluster Vertex Deletion	2.27	1.26
Gramm et al. [25]	Triangle Edge Deletion	2.27	2.47
Gramm et al. [25]	Triangle Vertex Deletion	2.27	2.42
Gramm et al. [25]	Cograph Vertex Deletion	3.30	3.30
Skjernaa	X3SAT	1.1003	1.1058
Skjernaa	X3SAT, respect to m	(1.1939)	1.1586
Skjernaa	XSAT	1.1749	1.1975
Skjernaa	XSAT, respect to l	(1.0842)	1.0978
Skjernaa	SAT, respect to l	1.0740	1.0983
Fedin & Kulikov [23]	SAT, respect to m	1.2389	1.2721
Fedin & Kulikov [23]	SAT, respect to l	1.0740	1.0983[a]
Fedin & Kulikov [23]	MAXSAT, respect to m	1.341	1.3723
Fedin & Kulikov [23]	MAXSAT, respect to l	1.1058	1.1359
Fedin & Kulikov [23]	$(n, 3)$-MAXSAT	1.1058	1.2852
Fedin & Kulikov [23]	$(n, 3)$-MAXSAT, respect to m	(1.341)	1.2366
Fedin & Kulikov [23]	$(n, 3)$-MAXSAT, respect to l	(1.1058)	1.0983
Nikolenko & Sirotkin [41]	SAT, respect to m	1.2389	1.4423[b]

[a] In [23] they only state 1.1011. The bound 1.0983 can be found at `http://logic.pdmi.ras.ru/~kulikov/autoproofs.html`

[b] It is not really fair to compare the result to other results, as the purpose of their program was to obtain a bound on an algorithm only using pure literal elimination

Table 4.1: Results for automated generation of algorithms. Results in parenthesis are results where no previous results have been published, and the results stated are either very simple to obtain or the result for a more general problem

presented here, we use no preprogrammed reductions and the only assumption on the variable occurrences is that at least one variable must occur thrice, which is not a restriction, but only a mean to ensure that the algorithm does not try to solve instances where each variable occurs at most twice, as these can be solved in polynomial time. Instead of using preprogrammed reductions we let the program find reductions, by for each case trying all assignments, and note if a variable is forced to be either *true* or *false*, or if a pair of variables are forced to be either equal or different. It is also able to find reductions where a clause can be removed without changing the set of valid assignments. We do not limit the size of instances which we consider, and instead of trying to prove a specific branching value, it always take the worst case in the case analysis and tries to improve it. This is done by expanding the case into several subcases, creating a tree structure of cases. We then use the tree structure to avoid symmetrical cases and subcases of already solved cases. In Table 4.2 is a summary of the properties of each of the programs presented.

Paper	Problems	Flexible reductions	Flexible size	Flexible branching value	Flexible branching type	Without assumptions	Avoids symmetrical cases
[25]	Graph modification problems	%	%	✓	✓	%	(✓)
[23]	SAT, MAXSAT, $(n, 3)$-MAXSAT	%	✓	%	%	%	%
[41]	SAT	%	✓	%	%	%	%
[52]	X3SAT, XSAT, 3SAT, SAT	✓	✓	✓	%	✓	✓

Table 4.2: Comparison between programs for automatic generation of algorithms

Chapter 5

Colouring

5.1 Introduction

In this chapter we consider another NP-complete problem, which has received a lot of interest:

k-Colouring	
Input:	An undirected graph $G = (V, E)$.
Output:	Does there exist a colour assignment using k colours, such that every pair of neighbours have distinct colours: $\exists c : V \to [k]\ \forall (i, j) \in E : c(i) \neq c(j)$.

As with k-SAT, k-Colouring is in P, for $k \leq 2$, while it is NP-complete for $k \geq 3$. A *yes*-instance of k-Colouring is said to be *k-colourable*, and 2-colourable graphs are also called bipartite. The subgraph induced by a subset of the vertices $S \subseteq V$ is denoted $G[S]$ and if G' is a subgraph, we let $V(G')$ denote the vertices of G' and call $G[V \backslash V(G')]$ the *remaining graph*. A *maximal k-colourable subgraph* of a graph is an induced k-colourable subgraph, contained in no other induced k-colourable subgraph. If $k = 1$ or $k = 2$, we use the terminology *maximal independent set* or *maximal bipartite subgraph*, respectively. As a maximal independent set does not contain any edges we will represent it just as a set of vertices.

Closely related to k-Colouring is the problem of determining the chromatic number of a graph. The chromatic number of a graph G is the smallest k, such that G is k-colourable.

Chromatic number problem	
Input:	An undirected graph G.
Output:	The smallest k, such that G is k-colourable.

This problem, in its decision form, is also NP-complete. The chromatic number of G is denoted $\chi(G)$.

5.1.1 Constraint Satisfaction Problem

Constraint Satisfaction Problem (or just CSP) is not as well known as SAT or k-Colouring, but is in fact closely related to both, and we include it here as many of the algorithms solving k-Colouring solves CSP-instances as a part of the algorithm.

CSP is a bit more complicated to describe than the other problems presented so far, so first we give an informal description of it. An $(a, b)-$CSP consists of a set of "objects" and a set of constraints. Each object is to be assigned one of a possible values, but the assignment will have to satisfy the constraints. Each constraint consists of at most b object-value pairs, and in an assignment satisfying such a constraint, at least one of the b objects is not assigned the associated value.

(a, b)-Constraint Satisfaction Problem	
Input:	A pair (S, C) consisting of a set S of objects and a set C of constraints. $C \subseteq (S \times [a])^b$.
Output:	Does there exist a value assignment to the objects of S, such that the constraints of C are satisfied: $\exists f : S \to [a] : \forall((s_1, v_1), \ldots, (s_b, v_b)) \in C : \exists i \in [b] : f(s_i) \neq v_i$

The Constraint Satisfaction Problem is interesting because of its relation to Satisfiability and k-Colouring:

- $(2, k)$-CSP is equivalent to k-SAT: S corresponds to the variables, and each constraint corresponds to a k-clause.
- $(k, 2)$-CSP is a generalisation of k-Colouring: S corresponds to the set of vertices, and each constraint is prohibiting a certain colour-combination of two vertices (for k-Colouring each edge prohibits the k colour-combinations where the two end vertices obtain the same colour).

(a, b)-CSP is NP-complete when $a \geq 2$ and $b \geq 2$, but not both are equal to two, and in P otherwise. We will only present algorithms for solving CSP which are connected to colouring.

5.2 Algorithms for Colouring

In this section we will give some of the history of exact algorithms for colouring. Throughout this section $G = (V, E)$ will be an undirected graph, n will denote the number of vertices in the graph, and m the number of edges of the graph.

When looking at maximal k-colourable subgraphs it is often easier to look at a specific k-colouring guaranteed to exist by the following lemma, which is proved in Chapter 9:

Lemma 5.1 (Lemma 9.1) *Let M be a maximal k-colourable subgraph of a graph, $G = (V, E)$. Then the vertices of M can be split into colour classes $C_1, C_2, \ldots, C_k$ of non-increasing sizes s.t. for all i, j with $0 \leq i < j \leq k$, $G[C_{i+1} \cup C_{i+2} \cup \cdots \cup C_j]$ is a maximal $(j-i)$-colourable subgraph of $G[V \setminus (C_1 \cup C_2 \cup \cdots \cup C_i)]$.*

First it should be noted that the trivial algorithm for k-Colouring is running in time $O(k^n)$, but it is easy to improve this to $O((k-1)^n)$ by assigning colours to the vertices in a depth first order. Determining the chromatic number can, with the same improvement, be done in time $O((\chi(G)-1)^n)$, which in the worst case is $O((n-1)^n)$.

χ	3-col.	4-col.	5-col.	Type	Reference
2.4422	1.4422	2	-	det.	E. Lawler, 1976 [34]
-	1.4147	-	-	det.	I. Schiermeyer, 1994 [46]
-	1.3977	*1.5849*	*1.9376*	det.	I. Schiermeyer, 1996 [47][a]
-	1.3446	-	-	det.	Beigel and Eppstein, 1995 [5]
-	1.3553	1.8072	2.2590	prob.	D. Eppstein, 2001 [19]
-	1.3289	-	-	det.	D. Eppstein, 2001 [19]
2.4150	-	-	-	det.	D. Eppstein, 2001 [20]
2.4023	-	1.7504	2.1592	det.	J. Byskov, 2002 [8]
-	-	-	2.1020	det.	Byskov and Eppstein, 2004 [9]

[a] In Chapter 9 we show that these particular results for 4- and 5-colouring are incorrect.

Table 5.1: History of colouring.

In Table 5.1 is a summary of most results in the field of colouring (as with the previous tables, an entry with c, indicates an algorithm running in time $O(c^n)$). We will in the following sections present some of these results.

5.2.1 Chromatic number

Lawler [34] was the first to present a non-trivial algorithm for determining the chromatic number. He did so in 1976, by presenting a very simple algorithm which solved the problem in $O((1+\sqrt[3]{3})^n)$. The algorithm uses dynamic programming to calculate the chromatic number for all 2^n subgraphs. Using the fact (Lemma 5.1, with $i = 0$ and $j = 1$) that in a colouring, one of the colour classes can be assumed to be a maximal independent set of the graph, we obtain the recursive formula:

$$\chi(G[V']) = \begin{cases} 0 & \text{if } V' = \emptyset \\ 1 + \min\limits_{S \in MIS(V')} \{\chi(G[V' \setminus S])\} & \text{if } V' \neq \emptyset \end{cases}$$

$MIS(V')$ denotes the set of all maximal independent sets of the subgraph induced by V'. Thus to calculate $\chi(G[V'])$, given $\chi(G[V''])$ for all subsets V'' of V', we just calculate the set $MIS(V')$. The number of maximal independent sets in a graph with n vertices is at most $3^{n/3}$ [40], and it can be generated with

only a small polynomial overhead per maximal independent set [55]. The time used to calculate $\chi(V')$ for all subsets of V is thus:

$$\sum_{r=0}^{n} \binom{n}{r} 3^{\frac{r}{3}} = (1 + \sqrt[3]{3})^n$$

Lawler also noticed that in addition to the algorithm for finding the chromatic number, the method can be used to find a 3-colouring of a graph if one exists. By generating all maximal independent sets of a graph, and for each set S check whether the graph $G[V \setminus S]$ is 2-colourable, a running time of $O(3^{n/3})$ is obtained. Although not connected to the above algorithm, Lawler also mentions that 4-Colouring can be decided in $O(2^n)$, by forming all 2-partitions of the vertices, and for each, check if both sets are bipartite.

The algorithm by Lawler for determining the chromatic number was not improved until 2001, when Eppstein [20] presented an algorithm improving the time bound to $O((\frac{4}{3} + \frac{3^{4/3}}{4})^n)$. The algorithms compute the chromatic number in a very similar way, but Eppstein made a slight modification to the algorithm of Lawler, which at first does not improve the time complexity, but clears the path towards improvements. Eppstein computes an estimate of the chromatic number for every induced subgraph of the graph, but in contrast to Lawler, it is only guaranteed to be correct for all *maximal* k-colourable subgraphs. They both start with the small subgraphs, such that when they reach a subgraph G', the chromatic numbers of all induced subgraphs of G' have been calculated or estimated. But in contrast to Lawler, the chromatic number of G' has already been estimated when G' is reached. This is done by maintaining a table with upper bounds on the chromatic number for every induced subgraph of G. When $G[V']$ is reached, the upper bound on the chromatic number of $G[V' \cup S]$, for every S, where S is a maximal independent set of $G[V \setminus V']$, is set to the minimum of the previous upper bound and $\chi(G[V']) + 1$. This ensures that when a maximal k-colourable subgraph is reached its chromatic number is equal to the calculated upper bound. The time complexity of doing this is exactly the same as the one Lawler obtains.

Now to improve the time complexity, Eppstein notes that when doing the step for $G[V']$ it is not necessary to consider all maximal independent subsets of $G[V \setminus V']$; one can do with only considering the maximal independent subsets of $G[V \setminus V']$ of size at most $|V'|/\chi(G[V'])$ (Lemma 5.1 states that if $G[W]$ is maximal k-colourable, then it has a maximal $(k-1)$-colourable subgraph $G[W']$ of size at least $|W|(1 - \frac{1}{k})$, where $G[W \setminus W']$ is a maximal independent set of $G[V \setminus W']$). By precalculating the chromatic number for all the subgraphs with chromatic number less than or equal to 3, it is only necessary to do anything when reaching a subgraph with chromatic number greater than or equal to 3 (it is only necessary to compute upper bounds which are at least 4). Thus when doing the calculations for $G[V']$, it is enough to consider maximal independent subsets of size at most $|V'|/3$. To make this improvement useful, Eppstein shows that the number of maximal independent sets $I \subseteq V$ of size at most k is at most $3^{4k-n}4^{n-3k}$, and they can be generated with only a polynomial overhead. The time for precalculating the chromatic number for subgraphs with chromatic

number at most 3 is dominated by the time for computing all the maximal independent subsets which is (by the binomial formula)

$$\sum_{S\subseteq V} O\left(3^{\frac{4|S|}{3}-|V\setminus S|}4^{|V\setminus S|-|S|}\right) = O\left(\sum_{i=0}^{n}\binom{n}{i}3^{\frac{7i}{3}-n}4^{n-2i}\right)$$
$$= O\left(\left(\frac{4}{3}+\frac{3^{\frac{4}{3}}}{4}\right)^n\right)$$

It should be noted that the algorithm easily can be modified to also return the colouring, with only a constant factor in overhead. The currently best known algorithm for determining the chromatic number is by Byskov [8], and we will look at this result in Section 5.3. There is no known algorithm using only polynomial space and running in time $O(\alpha^n)$ for a constant α.

5.2.2 3-Colouring

As mentioned above, Lawler showed in 1976 that 3-Colouring can be solved in $O(mn3^{n/3})$. The result from 1994 by Schiermeyer [46] is a complicated case analysis, which we will not describe. In 1995, Beigel and Eppstein [5] presented an algorithm for 3-Colouring running in time $O(1.3446^n)$. The primary ingredient of the algorithm is to solve a Constraint Satisfaction Problem.

Beigel and Eppstein do not consider the general CSP, but only (3,2)-CSP, which they solve in $O(1.3803^n)$, where n is the number of objects. To obtain this complexity, they observe that if an object can only be assigned two different values, then it can be removed by a kind of resolution at the expense of some additional constraints. The algorithm is based on a case analysis, branching on different local configurations. The cases depend on how many constraints an object-value pair is part of, and on chains of constraints. The worst case being an object-value pair contained in (at least) three constraints, with three different objects.

To solve a problem containing such a configuration (say (v,i) occurs in three constraints with v_1, v_2 and v_3), it suffices to solve two subproblems: Either v is assigned i, or it is not. In the first case v_1, v_2 and v_3 is restricted to two values, and can be removed, resulting in a problem with four objects fewer. In the second case we can remove v from the problem (as it has only two possible values), and thus just have to solve a problem with one object fewer. This results in a branching vector of (4,1), which gives $O(1.3803^n)$. We will not describe how they use this to solve 3-Colouring in $O(1.3446^n)$, as the technique is almost identical to the technique presented by Eppstein in 2001 [19] (Eppstein improves it a little bit, but the main ingredients are the same).

The primary improvement by Eppstein is to improve the algorithm for solving (3,2)-CSP, from $O(1.3803^n)$ to $O(1.3645^n)$. Instead of solving (3,2)-CSP directly, he solves (4,2)-CSP, by case-analysis. The case analysis is somewhat similar to the case-analysis they did to solve (3,2)-CSP. It is not in itself very interesting, so we will not go into any details about it. But the proof of the complexity is very interesting, because he does not measure the size of a problem

in the straightforward way. If n_3 is the number of objects that can have three different values, and n_4 is the number of objects that can have four different values, it would be natural to say that the size n of the problem is n_3+n_4. On the other hand, it can be noted that an object that can have four different values, can be substituted by two objects, each of which can have three different values (two of the possible values correspond to values of the original object, while the last corresponds to "it is the other object that determines the value"). Therefore $n_3 + 2n_4$ would also be a way to measure the size of the problem, taking account of the fact that problems with many objects with four possible values, is probably more difficult than problems with few. Eppstein mentions the latter as the natural way of measuring the size, but uses instead $n_3 + (2 - \epsilon)n_4$. Note that no matter which of these measures are used, the size of a (3,2)-CSP instance is the same. Using this measure in the calculation of the workload, of each case, the running time depends on ϵ. Eppstein then proves that choosing ϵ such that $\tau(3 - \epsilon, 4 - \epsilon, 4 - \epsilon) = \tau(1 + \epsilon, 4)$ results in the best bound, and shows that this is the case when both values equal $\tau(4, 4, 5, 5) \approx 1.36443$, which happens when $\epsilon \approx 0.095543$.

As mentioned, 3-Colouring is a special case of (3,2)-CSP, and by translating a 3-Colouring instance directly to (3,2)-CSP, a running time of $O(1.3645^n)$ is obtained. To improve this, Beigel and Eppstein note that for any subset S of the vertices, 3-Colouring can be decided in $O(3^{|S|}1.3645^{n-|S\cup \mathrm{N}(S)|})$, where $\mathrm{N}(S)$ is the set of neighbours of S. This is obtained by trying all possible colourings for S, and translating the remaining problem to a (3,2)-CSP of size $n-|S|$. But in the translated problem all vertices in $\mathrm{N}(S)$ have only two different possible colours, and can thus be removed at no significant cost, resulting in a problem of size $n-|S\cup \mathrm{N}(S)|$, which can then be solved by the above algorithm. Having a small subset of the vertices, with a lot of neighbours will thus result in a low running time.

In order to find such a set, the papers suggest to find a set, X, which is maximal, such that no two vertices are neighbours, or have a neighbour in common. Now every vertex in $\mathrm{N}(X)$ has a unique neighbour in X, and every vertex in $V\setminus(X\cup \mathrm{N}(X))$ has a neighbour in $\mathrm{N}(X)$ (by maximality of X). If every vertex in $V \setminus (X \cup \mathrm{N}(X))$ is associated to an arbitrary vertex in $\mathrm{N}(X)$, a forest with the vertices of X as roots is formed. Choosing S to be X together with any vertex of $\mathrm{N}(X)$, which in the forest has at least 3 children (3 associated vertices from $V \setminus (X \cup \mathrm{N}(X))$), results in a good S. Both papers use this technique to chose S but also use some additional case analyses to avoid certain tree-"shapes" in the forest. We will not go into any details with this, just note that this leads to an $O(1.3289^n)$ algorithm for 3-Colouring and this is the currently best, known algorithm for 3-Colouring.

5.2.3 Colouring in general

Most effort in the field of colouring, has gone into solving 3-Colouring, but there are some results for the general case. The algorithm for determining the chromatic number, can of course be used for deciding k-Colouring, and it is in fact the best known algorithm when $k \geq 7$. In the paper by Eppstein mentioned

above [19], he also shows that k-Colouring can be decided by a probabilistic algorithm in time $O((0.4518k)^n)$. For each vertex, he randomly chooses 4 of the possible colours, and then translates the problem into a (4,2)-CSP, which he solves by the algorithm presented in the previous section. He repeats this sufficiently many times to make the probability of failure small.

Another algorithm that works in the general case is by Byskov [8]. It is an extension of the 3-Colouring algorithm Lawler [34] presented in 1976. As mentioned previously, Eppstein [20] shows that the number of maximal independent sets $I \subseteq V$ of size at most k is at most $3^{4k-n}4^{n-3k}$. Byskov notes that this result can be used to decide 4-Colouring in $O(1.7504^n)$: generate all maximal independent sets of the graph of size at least $\lceil \frac{n}{4} \rceil$, and use the 3-Colouring algorithm from Eppstein [19] on the remaining graph. The bound of Eppstein is used to bound the number of maximal independent sets of size exactly k. The time complexity for the algorithm is dependent on the time complexity of 3-Colouring, and can more generally be stated as $O(n \cdot 4^{n/4} \cdot T_3(\frac{3n}{4}))$, where $T_3(\cdot)$ is the time it takes to solve 3-Colouring.

The bound from Eppstein on the number of maximal independent sets of size at most k is tight for $\frac{n}{4} \leq k \leq \frac{n}{3}$. Byskov strengthens it to be tight for all k:

$$\mathcal{I}^{\leq k}(n) = \begin{cases} \lfloor \frac{n}{k} \rfloor^{k-(n \mod k)} (\lfloor \frac{n}{k} \rfloor + 1)^{n \mod k} & \text{if } k \leq \frac{n}{3} \\ 3^{\frac{n}{3}} & \text{if } k > \frac{n}{3} \wedge n \equiv 0 \pmod 3 \\ 4 \cdot 3^{\lfloor \frac{n}{3} \rfloor - 1} & \text{if } k > \frac{n}{3} \wedge n \equiv 1 \pmod 3 \\ 2 \cdot 3^{\lfloor \frac{n}{3} \rfloor} & \text{if } k > \frac{n}{3} \wedge n \equiv 2 \pmod 3 \end{cases}$$

where $\mathcal{I}^{\leq k}(n)$ denotes the maximum possible number of maximal independent sets of size at most k in graphs with n vertices. In addition, he shows that the number of maximal independent sets of size exactly k is at most $\lfloor \frac{n}{k} \rfloor^{k-(n \mod k)} (\lfloor \frac{n}{k} \rfloor + 1)^{n \mod k}$, and this bound is also tight for any k. It should be noted that the two bounds are equal when $k \leq \frac{n}{3}$. Using this result and the same method as he used for 4-Colouring (using the 4-Colouring algorithm as a black box instead of the 3-Colouring algorithm), he obtains an algorithm for 5-Colouring, running in time $O(2.1593^n)$ using the 3-Colouring algorithm by Eppstein (the 4-Colouring algorithm used as black box uses a 3-Colouring algorithm as black box), or in general $O(n \cdot 5^{n/5} \cdot T_4(\frac{4n}{5})) = O(n^2 \cdot 20^{n/5} \cdot T_3(\frac{3n}{5}))$.

Of course, the method can be extended to k-Colouring for any $k > 3$ (giving $O(n^{k-3}(\frac{k!}{3!})^{\frac{n}{k}} \cdot T_3(3n/k))$, but for $k = 6$ much better algorithms for 3-Colouring are required ($O(1.1824^n)$) for the algorithm to be competitive to the general algorithm [7] for determining the chromatic number. For $k > 6$ it will, no matter how fast 3-Colouring can be decided, not be competitive to the chromatic number algorithm.

5.3 Using maximal bipartite subgraphs

A possible way to improve these results is to find a bound on the number of maximal bipartite subgraphs of a graph. Lemma 5.1 ensures that if G is k-

colourable, it has a k-colouring consisting of a maximal bipartite subgraph such that the remaining graph is $(k-2)$-colourable. Thus a natural extension to the algorithm by Byskov [8] would be to generate all maximal bipartite subgraphs and check whether the remaining graphs are $(k-2)$-colourable. The time complexity of this algorithm is proportional to the time complexity of generating all maximal bipartite subgraphs plus the number of maximal bipartite subgraphs times the time complexity of checking $(k-2)$-colourability of the remaining graphs.

The time complexity of checking 4-colourability using the above algorithm is proportional to the time complexity of generating all maximal bipartite subgraphs, since 2-colourability can be checked in polynomial time. The algorithm by Byskov [8] for 5-Colouring is really just finding all maximal bipartite subgraphs of size at least $\frac{2}{5}n$. But it also finds some non-maximal ones (the union of two maximal independent sets is not necessarily a maximal bipartite subgraph: In the 6-cycle, two opposite vertices form a maximal independent set, but their union with a maximal independent set of the remaining graph does not form a maximal bipartite subgraph), and finds some of them multiple times (k copies of K_2 have only one maximal bipartite subgraph, but it may be decomposed into two maximal independent sets in 2^{k-1} different ways).

In Chapter 9, we show both a lower and upper bound on the maximum possible number of maximal bipartite subgraphs in a graph. The lower bound is shown by providing an infinite family of graphs with many maximal bipartite subgraphs. The infinite family consists of disconnected copies of a single graph, the k'th one having k copies. Our first lower bound used K_5 to generate the infinite family. The lower bound obtained by using this is $\binom{5}{2}^{n/5} = 10^{n/5} \approx 1.5849^n$, which was also observed by Schiermeyer [47] in 1996. Schiermeyer also states a matching upper bound, which we invalidate by using the graph in Fig. 5.1.

Theorem 5.1 (Theorem 9.1) *There exists an infinite family of graphs all having* $105^{n/10} \approx 1.5926^n$ *maximal bipartite subgraphs of size exactly* $\frac{2}{5}n$.

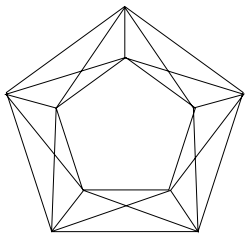


Figure 5.1: The graph used in Theorem 5.1.

We also present an upper bound on the number of maximal bipartite subgraphs.

Theorem 5.2 (Theorem 9.2) *Any graph contains at most* $n \cdot 12^{n/4} \approx n \cdot 1.8613^n$ *maximal bipartite subgraphs. Moreover, there is an algorithm that*

takes as input a graph and outputs all its maximal bipartite subgraphs in time $O(1.8613^n)$.

This upper bound is not good enough to improve any of the results for colouring, but recently Byskov and Epstein [9] presented an algorithm generating all maximal bipartite subgraphs in $O(1.7724^n)$. The algorithm is a branching algorithm with many cases so we will not describe it here. This improved bound results in the currently best known algorithm for 5-Colouring [9]:

Theorem 5.3 (Byskov and Eppstein [9], Theorem 2)
5-colourability can be checked in $O(2.1020^n)$ *using polynomial space.*

Proof. Find all maximal bipartite subgraphs of the graph and for all of size at least $2n/5$ check 3-Colouring of the remaining graphs by using the 3-Colouring algorithm of Beigel and Eppstein [5] running in time $O(1.3289^n)$. The total running time is $O(1.7724^n \cdot 1.3289^{3n/5}) \approx O(2.1020^n)$ □

In order to use this technique to improve the result for 4-Colouring, an upper bound of at most 1.75^n is required, and an algorithm generating all maximal bipartite subgraphs in this time should exist. It will suffice to show this upper bound for the number of maximal bipartite subgraphs of size greater than $\frac{1}{2}n$.

We do believe that the real upper bound is much closer to our lower bound than the upper bound by Byskov and Eppstein. If our lower bound is also an upper bound, it is possible to solve 5-Colouring in time $O(10^{n/5} \cdot T_3(\frac{3}{5}n))$, which with the 3-Colouring algorithm by Eppstein is $O(10^{n/5} \cdot 1.3289^{3n/5}) \approx O(1.8889^n)$.

Byskov and Eppstein also uses the upper bound on the number of maximal bipartite subgraphs to show the following:

Theorem 5.4 (Byskov and Eppstein [9], Theorem 3)
It is possible to find either all 3-colourable subgraphs or all maximal 3-colourable subgraphs of a graph in time $O(2.1809^n)$ *using* $O(2^n)$ *space.*

The algorithm is to for each maximal independent set I find all maximal bipartite subgraphs of $G[V \setminus I]$. From this result they conclude that 6-Colouring also can be solved in $O(2.1809^n)$ using $O(2^n)$ space, which is the first algorithm for 6-Colouring faster than the general algorithm for chromatic number.

In 2002, Byskov [7] used a weaker version of Theorem 5.4 (based on Theorem 5.2) to give an algorithm for determining the chromatic number in $O(2.4023^n)$. The main observation is that if we have a table with all maximal k-colourable subgraphs marked, then we can build a table with all $(k+1)$-colourable subgraphs marked in time $O(2.4023^n)$. This is done by for each maximal independent set I finding all k-colourable subgraph U of $G[V \setminus I]$, and marking $G[U \cup I]$ as $(k+1)$-colourable. This marks all maximal $(k+1)$-colourable subgraphs, and by doing a single traversal through the table all $(k+1)$-colourable subgraphs can be marked. All k-colourable subgraphs of $G[V \setminus I]$ can be found in time $O(2^{|V \setminus I|})$, by looking at the entries in the table corresponding to subgraphs of $G[V \setminus I]$, and we thus get a running time of

$$\sum_{I \in MIS(G)} 2^{|V \setminus I|} \leq \sum_{k=1}^{n} |MIS_{=k}(G)| \cdot 2^{n-k} \tag{5.1}$$

where $MIS_{=k}$ is the set of all maximal independent sets of size k in G. This sum can be shown to be $O(80^{n/5}) = O(2.4023^n)$.

Chapter 6

Graph Distinguishability

A classical problem presented by Frank Rubin [43] goes:

> A blind man keeps his keys on a circular ring. There are s distinct handle shapes that he can tell apart by feel, and he can purchase any key with any desired handle shape. Assume that all keys are symmetrical so that a rotation of the key ring about an axis in its plane is undetectable from examination of a single key. How many keys can he keep on the ring and still be able to select the proper key by feel?

In this chapter we will consider a generalised version of this problem called Graph Distinguishability. A graph is said to be k-distinguishable if it is possible to assign one of k different colours to each vertex of the graph in such a way that no non-trivial automorphism preserves the colour of all vertices. The above problem is the special case where $k = s$ and the graphs are cycles.

Let $G = (V, E)$ be a graph. The *distinguishing number*, $D(G)$, is the smallest number, k, for which G is k-distinguishable. If k is a positive integer, a *k-colouring* of G is a function mapping V to $[k]$. Note that a colouring in this chapter have no connection to the kind of colourings presented in the previous chapter. A *legal k-colouring*, c, is a k-colouring which destroys all non-trivial automorphisms:

$$\forall \phi \in \mathrm{Aut}(G) \setminus \{\mathrm{id}(V)\} : c \neq c \circ \phi. \tag{6.1}$$

The set of legal k-colourings of a graph G will be denoted $\mathrm{LC}_k(G)$.

Two legal k-colourings, c_1 and c_2, are called *indistinguishable* if there exists a $\phi \in \mathrm{Aut}(G)$ such that $c_1 = c_2 \circ \phi$.

DIST_k	
Input:	A graph $G = (V, E)$
Output:	Are there any legal k-colourings of G, i.e., is $\mathrm{LC}_k(G) \neq \emptyset$?

DIST_1 is also called *RIGID*. There has been considerable interest in the computational complexity of the problem of deciding the distinguishing number

of a graph. DIST_0 is trivial as it only contains the empty graph and RIGID is in coNP as a graph can be shown to be nonrigid by giving a nontrivial automorphism on the graph. For $k > 1$, DIST_k is not known to be in neither NP nor coNP, but it is easy to see that DIST_k is in Σ_2^P.

In 1996, Albertson and Collins [2] decided the distinguishing number for certain classes of graphs. They study the relation between the automorphism group of a graph and the distinguishing number of the graph, and prove the following results:

- $D(G) \leq 1 + \lfloor \log(|\mathrm{Aut}(G)|) \rfloor$,
- for any nontrivial finite group, Γ, there exists a graph, G, such that $\mathrm{Aut}(G) \cong \Gamma$ and $D(G) = 2$,
- if $\mathrm{Aut}(G)$ is nontrivial and either Abelian or Hamiltonian, then $D(G) = 2$,
- if $\mathrm{Aut}(G)$ is dihedral, then $D(G) \leq 3$,
- if $n \neq 3, 4, 5, 6, 10$ and $\mathrm{Aut}(G) \cong D_n$, where D_n is the automorphism group of the n-cycle, $D_n \cong \mathrm{Aut}(C_n)$, then $D(G) = 2$, and
- if $\mathrm{Aut}(G) \cong S_4$, then $D(G) = 2$ or $D(G) = 4$.

Two years later, in 1998, Russell and Sundaram [44] proved that DIST_k lies in $\Sigma_2^P \cap \mathrm{AM}$[1] and as Boppana, Håstad and Zachos [6] have proved that if coNP $\subset$ AM then the polynomial hierarchy collapses, they conclude that if it is coNP-hard the polynomial hierarchy collapses.

Recently Arvind and Nikhil [3] proved that for planar graphs the distinguishing number can be determined in polynomial time. The proof of this result consists of two parts. First they prove that $DIST_k$ can be determined in linear time for trees, and thus the distinguishing number can be found in polynomial time. The second, and most complicated part, reduces the problem of deciding the distinguishing number of a planar graph to the problem of deciding the distinguishing number of a tree. Here we will sketch the first part.

In linear time we find the *center* of the tree; i.e. the vertex which minimises the height of the tree when the vertex is chosen as root. There can be two such vertices, but in that case we insert a vertex on the edge between the two vertices. Then we have a unique center and the insertion does not change the distinguishing number of the tree. We then choose the center as the root of the tree, and use a result by Aho, Hopcroft and Ullman [1]: given a rooted tree T as input, we can in linear time compute a canonical numbering of the vertices of T by $\log n$ bit integers, such that two subtrees at the same level get the same label if and only if they are isomorphic. With such a labelling it is easy to make a recursive traversal of the tree and calculate the total number of distinguishable legal k-colourings for every subtree: for a single vertex (each

[1] Note that Russell and Sundaram repeatedly state that the problem lies in $\mathrm{AM} \subset \Sigma_2^P \cap \Pi_2^P$. This statement is somewhat misleading as it is a well known open problem if AM is in Σ_2^P (see, e.g., [26]). What they *do* show is that the problem is in AM and since it is clearly in Σ_2^P, it is in $\Sigma_2^P \cap \mathrm{AM}$ and as $\mathrm{AM} \subset \Pi_2^P$ thus in $\Sigma_2^P \cap \Pi_2^P$.

leaf), the number of distinguishable legal k-colourings is trivially k. If v has $\sum_{i=1}^{r} m_i$ children, $\{v_{ij}|1 \leq j \leq m_i, 1 \leq i \leq r\}$, such that the subtrees rooted at $\{v_{ij}|1 \leq j \leq m_i\}$ are all isomorphic and each can be legally k-coloured in l_i distinguishable ways, then the subtree rooted at v can be legally k-coloured in $k \cdot \prod_{i=1}^{r} \binom{l_i}{m_i}$ distinguishable ways.

In Chapter 10 we present a result on how to reduce DIST_l to DIST_k for various values of l and k:

Theorem 6.1 (Theorem 10.2) *If there exists a graph $H = (W, F)$ such that H can be legally k-coloured in l distinguishable ways, then $DIST_l$ can be reduced to $DIST_k$:*

$$DIST_l \prec DIST_k.$$

The idea in the proof is simple: by replacing every vertex in a graph G with a copy of H, we can check if G can be distinguished using l colours, by checking if the new graph can be distinguished using k colours. By using this theorem we also prove

Theorem 6.2 (Theorem 10.1) *RIGID is not harder than any other $DIST_k$, $k \geq 1$:*

$$RIGID \prec DIST_k.$$

Finally, we present different ways for constructing graphs with certain l and k values from smaller graphs, and show that DIST_l can be reduced to DIST_2 for l equal to 1, 2, 4, 6, 8, 10, 12, 14, 15, 16, 18, 20, 22, 24, 27, 28, 30, 32, 36 and 40.

Part II

Papers

Chapter 7

Algorithms for Exact Satisfiability

This chapter contains the paper "New Algorithms for Exact Satisfiability" [10]. The paper is co-authored by Jesper M. Byskov and Bolette A. Madsen, and is to appear in Theoretical Computer Science. Minor typographical changes have been made compared to the original paper.

Abstract

The Exact Satisfiability problem is to determine if a CNF-formula has a truth assignment satisfying exactly one literal in each clause; Exact 3-Satisfiability is the version in which each clause contains at most three literals. In this paper, we present algorithms for Exact Satisfiability and Exact 3-Satisfiability running in time $O(2^{0.2325n})$ and $O(2^{0.1379n})$, respectively. The previously best algorithms have running times $O(2^{0.2441n})$ for Exact Satisfiability (Monien, Speckenmeyer and Vornberger (1981)) and $O(2^{0.1532n})$ for Exact 3-Satisfiability (Dahllöf, Jonsson and Beigel (2004)). We extend the case analyses of these papers and observe that a formula not satisfying any of our cases has a small number of variables, for which we can try all possible truth assignments and for each such assignment solve the remaining part of the formula in polynomial time.

7.1 Introduction

The Exact Satisfiability (XSAT) problem is a variant of Satisfiability (SAT), where the difference is that in XSAT a clause is satisfied if *exactly* one of its literals is true. The Exact 3-Satisfiability (X3SAT) problem is the variant of XSAT in which each clause contains at most three literals. X3SAT is also called One-In-Three Satisfiability. XSAT is NP-complete even when restricted to clauses containing at most three literals and all variables occurring only unnegated [45]. XSAT with all variables occurring at most twice can be solved in polynomial time [39].[1]

XSAT can easily be solved in time $O(2^n)$ (we will ignore polynomial factors when stating running times, since these are all exponential) by enumerating all possible truth assignments to the n variables. In 1981, Schroeppel and

[1] They state that a generalised version of XSAT with variables occurring at most twice, called $MAX(\{\leq, =, \geq\}, \cdot, 2)$, reduces to Perfect Matching. The proof is in the technical report [38].

Shamir [49] were the first to give a faster algorithm. Their algorithm solves a class of problems, of which XSAT and Knapsack are the most prominent, in time $O(2^{n/2})$ and space $O(2^{n/4})$. The same year, Monien, Speckenmeyer and Vornberger [39] gave an algorithm solving only XSAT, but in time $O(2^{0.2441n})$[2] and polynomial space. This is the previously best algorithm for XSAT.

X3SAT can of course be solved by an algorithm solving XSAT, but in recent years faster algorithms for X3SAT have been designed. The first was by Drori and Peleg [18] and runs in time $O(2^{0.2072n})$. This was improved by Kulikov [30] and independently Porschen, Randerath and Speckenmeyer [42] in 2002 to obtain a running time of $O(2^{0.1626n})$. The previously best algorithm is by Dahllöf, Jonsson and Beigel [13] and runs in time $O(2^{0.1532n})$.

Except for the algorithm by Schroeppel and Shamir [49], all the algorithms mentioned above are *branch-and-reduce* algorithms. A branch-and-reduce algorithm branches by making recursive calls on smaller formulas, such that the original formula is satisfiable if and only if at least one of the smaller formulas is satisfiable. In each branch, the algorithm reduces the formula by replacing it with another formula that is satisfiable if and only if the original formula is and that contains fewer variables. Fast branch-and-reduce algorithms rely on good decisions about what to branch on and good reduction rules.

In this paper, we present new branch-and-reduce algorithms for XSAT and X3SAT running in time $O(2^{0.2325n})$ and $O(2^{0.1379n})$, respectively. We introduce new reductions for both XSAT and X3SAT and improve the case analyses by a more careful analysis of the worst cases, which for some cases involves splitting them into more cases. Our main improvement, however, lies in our handling of *sparse* formulas: if the number of variables occurring at least three times in the formula is small, we can enumerate all possible truth assignments to these variables. For each assignment, the remaining formula contains only variables occurring at most twice, so we can decide in polynomial time, if it is satisfiable [39].

7.2 Preliminaries

7.2.1 Definitions

We are given a set of variables, which we will denote by the letters x, y, z, w and u. A *literal* is either a variable x or the negation of a variable $\bar{x}$; we use $\tilde{x}$ to denote a literal that is either x or $\bar{x}$. A clause is a collection of literals, written as $(\tilde{x}_1, \ldots, \tilde{x}_l)$; we use the letter C to denote clauses. Sometimes, we will think of a clause as a set of literals (actually a multiset, since a clause can contain more than one of each literal); we use $(\tilde{x}, C)$ to denote a clause with $\tilde{x}$ and all the literals in C. A formula is a set of clauses usually written as $C_1 \wedge C_2 \wedge \cdots \wedge C_m$; we use the letter F to denote formulas. In intermediate steps of our algorithm we allow clauses to contain constants (*true* or *false*). The *size* of a formula is the number of literals and constants contained in the formula.

[2] Note that the journal version [39] only states the time $O(2^{n/4})$. The time complexity of $O(2^{0.2441n})$ is proved in the technical report [38].

XSAT is the problem of given a formula F with m clauses over n variables to decide, if there exists an assignment to all the variables, such that *exactly* one literal in each clause is true.

We let $V(F)$ denote the variables occurring in F. An (a, b)-*occurrence* is a variable occurring a times unnegated and b times negated or vice versa in F; a *unique* variable is a variable occurring only once. We will assume for simplicity, that when we look at a variable the first occurrence is unnegated.

We let $F[x \leftarrow y]$ where y is either a constant or a literal denote F with x replaced by y and $\bar{x}$ replaced by $\bar{y}$; similarly, we let $F[C \leftarrow \mathit{false}]$ denote F with all literals in C replaced by *false* and their negations by *true*.

In X3SAT, a *cycle* is a list of clauses $(y_1, \tilde{z}_1, z_2)$, $(y_2, \tilde{z}_2, z_3)$, $\ldots$, $(y_k, \tilde{z}_k, z_1)$ where neighbour clauses and the first and last clause share a variable and the z_is are different variables.

7.2.2 Branching

Our algorithms make recursive calls on formulas with fewer variables. If C is a clause in the formula and $C' \subsetneq C$ then in a satisfying assignment for the formula either all literals in C' are *false* or exactly one is *true*. We use three different types of branches: branching on C', meaning that the recursive calls are on $C' \wedge F$ (in this case the formula can be reduced immediately afterwards, such that there will be fewer variables and clauses) and $F[C' \leftarrow \mathit{false}]$; we will denote the first branch "setting C' to *true*" and the second "setting C' to *false*". We can also branch on two variables x_1, x_2 meaning that the recursive calls are on $F[x_1 \leftarrow \mathit{true}]$, $F[x_2 \leftarrow \mathit{true}]$ and $F[x_1, x_2 \leftarrow \mathit{false}]$. Finally, we can branch on x_1; the two branches are then $F[x_1 \leftarrow \mathit{true}]$ and $F[x_1 \leftarrow \mathit{false}]$.

Sparse formulas

We call a formula k-*sparse*, if the number of variables occurring at least three times is at most n/k. To decide if a k-sparse formula is satisfiable, we can enumerate all possible truth assignments to these at most n/k variables; for each assignment, all variables in the remaining part of the formula occur at most twice, so we can decide in polynomial time, if it is satisfiable. The total running time is $O(2^{n/k})$. We use this for XSAT with $k = 5$ and for X3SAT with $k = 15/2$.

7.2.3 Branching vectors

In each branch of the algorithm, we remove a certain number of variables using the reductions; then we get a recursion for the running time of the form $T(n) = T(n-t_1) + T(n-t_2) + \cdots + T(n-t_k)$. We call $t = (t_1, t_2, \ldots, t_k)$ the *branching vector* of this branch. The solution to the recursion is $T(n) = \alpha_t^n$, where α_t is the positive root of $1 - 1/x^{t_1} - 1/x^{t_2} - \cdots - 1/x^{t_k}$; we call α_t the *value* of t and the value of a branching vector is decreasing as a function of the entries in the vector. Proofs of these results can be found in a manuscript by Kullmann and Luckhardt [32]. The logarithms of the values of all branching vectors occurring in this paper are either stated in Table 7.1 or are smaller than one of them by

Table 7.1: Branching vectors and the logarithms of their values (rounded up).

(a) XSAT

t	$\log_2(\alpha_t)$	t	$\log_2(\alpha_t)$
$(12,1)$	0.2302	$(11,11,3)$	0.2216
$(8,2)$	0.2325	$(10,8,4)$	0.2325
$(6,3)$	0.2315	$(13,7,4)$	0.2258
$(5,4)$	0.2232		

(b) X3SAT

t	$\log_2(\alpha_t)$
$(12,4)$	0.1379
$(11,5)$	0.1317
$(9,6)$	0.1353
$(8,7)$	0.1336

monotonicity. The running time of the whole algorithm is $O(2^{\log_2 \alpha \cdot n})$, where α is the largest of the α_ts.

7.3 The algorithms

Both our algorithms have the following structure: first, the algorithm reduces the formula using the reductions from Section 7.3.1. If the reduced formula (we call a formula *reduced*, if none of the reductions from Section 7.3.1 are applicable) contains no clauses it is satisfied and if it contains an empty clause it cannot be satisfied. If the formula only contains variables that occur at most twice in the formula, the algorithm decides in polynomial time if the formula is satisfiable; otherwise, the algorithm branches depending on whether the formula contains certain subformulas. In each branch, the algorithm is called recursively on smaller formulas, which are obtained by trying all assignments to a few of the variables in the original formulas. In some cases, the algorithm applies some special reductions, which are not part of the reduction procedures, to the smaller formulas before making the recursive call. The algorithm branches on the first matching case, which means that when it is in one case, no part of the formula matches any previous cases. The cases are described for XSAT in Section 7.3.2 and for X3SAT in Section 7.3.3. For simplicity, we ignore symmetries when this does not lead to confusion, so if we have two variables y_1 and y_2 or two clauses C_1 and C_2 that occur symmetrically and one of them has a certain property, we will just assume it is either.

7.3.1 Reductions

In this section, we present the reductions that are used in our algorithms; first, we present some common reductions used in both algorithms and then specific ones for the two algorithms. The reduction procedure for XSAT uses reductions (7.1) to (7.13) and the reduction procedure for X3SAT uses (7.1) to (7.8) and (7.14) and (7.15). The reductions are applied repeatedly top-down until no reduction applies. If any of the reductions either assign a variable both *true* and *false* or a constant is assigned its opposite value, the reduction procedures replace the whole formula with an empty clause.

When we branch, we call the algorithms recursively on smaller formulas. We show how to apply specific reductions from the reduction procedure to remove

the stated number of variables. One can also show that applying the reductions top-down leads to the same or better branching vectors.

Common reductions

The common reductions are standard reductions also used by, e.g., Kulikov [30]. Here F denotes the entire left hand side of the reduction.

$$(true, C) \wedge F' \rightarrow F'[C \leftarrow false] \tag{7.1}$$
$$(false, C) \wedge F' \rightarrow C \wedge F' \tag{7.2}$$
$$(x) \wedge F' \rightarrow F'[x \leftarrow true] \tag{7.3}$$
$$(x, y) \wedge F' \rightarrow F'[y \leftarrow \bar{x}] \tag{7.4}$$
$$(x, x, C) \wedge F' \rightarrow F[x \leftarrow false] \tag{7.5}$$
$$(x, \bar{x}, C) \wedge F' \rightarrow F'[C \leftarrow false] \tag{7.6}$$
$$(x, y, C) \wedge (x, \bar{y}, C') \wedge F' \rightarrow F[x \leftarrow false] \tag{7.7}$$
$$(x, y, C) \wedge (\bar{x}, \bar{y}, C') \wedge F' \rightarrow F[y \leftarrow \bar{x}] \tag{7.8}$$

Reductions for XSAT

Reduction (7.9) removes any variable x that only occur unnegated and only in clauses with a unique variable or with literal y and y is in no clauses without x. This case can for instance be used if two unique variables occur in the same clause, if a variable only occurs in clauses with unique variables or two variables only occur in clauses with each other.

$$(x, y, C_1) \wedge \cdots \wedge (x, y, C_k) \wedge (x, u_1, C'_1) \wedge \cdots \wedge (x, u_l, C'_l) \wedge F' \rightarrow F[x \leftarrow false] \tag{7.9}$$
$$x, y \notin V(F'),\ u_i \text{ unique}$$

Reduction (7.9) is not used for X3SAT, as (7.14) or (7.15) can be used instead.

Reductions (7.10) and (7.11) are called resolution; resolution is a well-known technique for removing variables occurring both unnegated and negated and can also be used for solving SAT formulas. The idea is, that we can remove a variable x occurring both unnegated and negated and make all possible combinations of the clauses that contained x and the clauses that contained $\bar{x}$. If x is an (a, b)-occurrence, this will replace $a + b$ clauses with ab clauses, so we only use it for $(a, 1)$- and $(2, 2)$-occurrences, as this does not increase the number of clauses.

$$(\bar{x}, C) \wedge (x, C_1) \wedge \cdots \wedge (x, C_k) \wedge F' \rightarrow (C, C_1) \wedge \cdots \wedge (C, C_k) \wedge F' \tag{7.10}$$
$$x \notin V(F')$$

$$(\bar{x}, C_1) \wedge (\bar{x}, C_2) \wedge (x, C'_1) \wedge (x, C'_2) \wedge F' \rightarrow (C_1, C'_1) \wedge (C_1, C'_2) \wedge (C_2, C'_1) \wedge (C_2, C'_2) \wedge F' \tag{7.11}$$
$$x \notin V(F')$$

Resolution is not used for X3SAT, as it creates clauses with more than three variables.

Reduction (7.12) removes clauses that have another clause as a subset, and (7.13) reduces the formula if a clause shares all but one literal with another.

$$C \wedge \underset{C \subseteq C'}{C'} \wedge F' \;\rightarrow\; C \wedge F'[C' \setminus C \leftarrow \mathit{false}] \tag{7.12}$$

$$(x, C') \wedge (C, C') \wedge F' \;\rightarrow\; (x, C') \wedge (\bar{x}, C) \wedge F' \tag{7.13}$$

Reductions (7.12) and (7.13) are not used for X3SAT as (7.14) handles the same cases when the clauses have size three.

Lemma 7.1 *In a reduced XSAT formula, for all pairs of clauses, each has at least two variables that do not occur in the other.*

Proof. All clauses contain at least three literals by (7.3) and (7.4), so we only need to consider clauses having at least two variables in common. Common variables must occur the same (unnegated or negated) by (7.7) and (7.8). No clause is a subset of another by (7.12), and for any pair of clauses both have at least two literals that do not occur in the other by (7.13), so the lemma is true. □

Reductions for X3SAT

Reduction (7.14) is also a standard reduction, but we only use it for X3SAT. Reduction (7.15) reduces formulas containing two variables that only occur unnegated and only in clauses with a unique variable, except for one clause, where one occurs unnegated and the other negated.

$$(x, y, z_1) \wedge (x, y, z_2) \wedge F' \;\rightarrow\; (x, y, z_1) \wedge F'[z_2 \leftarrow z_1] \tag{7.14}$$

$$(\bar{x}_1, x_2, y) \wedge F' \;\rightarrow\; F[x_2 \leftarrow \mathit{false}] \tag{7.15}$$

x_1 and x_2 only occur unnegated and in clauses with a unique variable in F'

Lemma 7.2 *A reduced X3SAT formula contains no constants or 1- or 2-clauses, and no two clauses have more than one variable in common; also, no clause has more than one unique variable and all $(a, 0)$- and $(a, 1)$-occurrences that are not unique are in a clause with no unique variables.*

Proof. That there are no constants, 1- or 2-clauses or clauses sharing more than one variable follows directly from (7.1) to (7.4) and (7.7), (7.8) and (7.14). No clause has more than one unique variable by (7.15) with x_1 and x_2 unique, and all $(a, 0)$- and $(a, 1)$-occurrences that are not unique are in a clause with no unique variables, by (7.15) with x_1 or x_2 unique. □

Soundness and complexity

The following lemma states that the reductions are *sound*, that is, the reduced formula is satisfiable if and only if the original formula is.

Lemma 7.3 *Reductions* (7.1) *to* (7.15) *are sound.*

Proof. To prove that (7.1) to (7.8) and (7.10) to (7.14) are sound we just note, that exactly one literal from a clause must be *true* and exactly one of x and $\bar{x}$ must be *true*.

In (7.9), x can be assumed to be *false*, as a satisfying assignment with x *true* can be changed to a satisfying assignment with x *false* by making y and the unique variables *true* instead. Similarly in (7.15), a satisfying assignment with x_2 *true* must have x_1 *true*, and it can be changed by setting x_1 and x_2 *false* and all the unique variables occurring with them *true*. □

The next lemma shows, that during the reduction procedures the size of the formula is never larger than the maximum of the size of the original formula and $2mn$. We use this to show, that the reduction procedures run in polynomial time in the size of the formula.

Lemma 7.4 *When the reduction procedures run on a formula F with m clauses and n variables the intermediate formulas are never larger than* $\max(|F|, 2mn)$.

Proof. For X3SAT it is obvious that the size of the formula is never larger than $\max(|F|, 3m)$.

For XSAT, none of the reductions increase the number of variables or clauses in the formula. The reduction procedure first applies reductions (7.1) to (7.6), which all decrease the size of the formula. After it has run, the formula contains no constants and no variable occurs more than once in the same clause; thus, the size of the formula is at most mn.

The only reductions which can make the formula larger are resolution ((7.10) and (7.11)), but the number of literals in a clause after resolution is still bounded by $2n$ and will be reduced to n before we perform resolution again; thus, after the first applications of (7.1) to (7.6), the size of the formula is always bounded by $2mn$. □

Lemma 7.5 *The reduction procedures run in polynomial time in the size of the formula.*

Proof. For each reduction, the algorithm can in polynomial time in the size of the formula check whether it is applicable and if so apply it.

Resolution ((7.10) and (7.11)) removes a variable from the formula, so they can be applied at most n times, since no reduction add variables. All the other reductions reduce the size of the formula (for (7.13) just note that C' has size at least two by (7.4)). Since the size of the formula is at most $\max(|F|, 2mn)$, the reduction procedures run in polynomial time in $|F|$. □

7.3.2 The algorithm for XSAT

In this section, we present our algorithm for XSAT and show that it achieves a branching vector of $(8, 2)$ corresponding to a running time of $O(2^{0.2325n})$. The previously best algorithm is by Monien et al. [39] and has worst case branching vector $(11, 1)$ corresponding to a running time of $O(2^{0.2441n})$.

Variables occurring both unnegated and negated

If the formula F contains a variable occurring both unnegated and negated, it must occur at least three times unnegated and twice negated or vice versa; otherwise, it would have been removed by resolution ((7.10) or (7.11)). Let x be the corresponding literal occurring at least three times unnegated and twice negated as in the clauses in Fig. 7.1. The algorithm branches on x. By Lemma 7.1, C_1

(x, C_1)	$(\bar{x}, C_1')$	$x = \mathit{true} : C_1 = C_2 = C_3 = \mathit{false}$
(x, C_2)	$(\bar{x}, C_2')$	$x = \mathit{false} : C_1' = C_2' = \mathit{false}$
(x, C_3)		

Figure 7.1. Branching on at least a $(3, 2)$-occurrence x.

contains at least two variables not in C_2, C_2 contains at least two variables not in C_3 and C_3 contains at least two variables not in C_1 and since none of these variables can be the same, the three clauses contain at least six different variables in total. Similarly, C_1' and C_2' contain at least four different variables in total; thus, branching on x yields a branching vector of at least $(7, 5)$.

Two clauses having at least two variables in common

Suppose F contains two clauses having more than one variable in common as in Fig. 7.2, with C_1 and C_2 not having any variables in common and $|C| \geq 2$. By

(C, C_1)	$C = \mathit{true} : C_1 = C_2 = \mathit{false}$, this removes $\lvert C_1\rvert + \lvert C_2\rvert$
(C, C_2)	variables plus one if $\lvert C\rvert = 2$ (by (7.4))
	$C = \mathit{false}$: this removes $\lvert C\rvert$ variables plus one
	for each C_i with $\lvert C_i\rvert = 2$ (by (7.4))

Figure 7.2. Two clauses having at least two variables in common.

Lemma 7.1, also $|C_1|, |C_2| \geq 2$. If the two clauses are not two 5-clauses having exactly two variables in common, the algorithm branches on C as shown in the figure. Since C, C_1 and C_2 all have size at least two, this removes at least four variables when C is set to *true* and two when C is set to *false*. Now, if any of the clauses are 2-clauses this removes one extra variable in one branch and if they are at least 3-clauses we remove at least one more in the other branch. All in all we get branching vectors of at least $(5, 4)$, $(6, 3)$ or $(7, 2)$, but $(7, 2)$ only if we had two 5-clauses having two variables in common. Having excluded that case, which we deal with later, we have $(8, 2)$ as the worst case.

Variables occurring at least four times

If F contains a variable x occurring at least four times and either with at least eleven different variables or in a 3-clause, the algorithm branches on x. If it occurs with eleven different variables, this yields a branching vector of at least $(12, 1)$ and if x is in a 3-clause, we get a branching vector of at least $(9, 2)$, since

x must occur with at least eight different variables: there are two variables in the 3-clause, and these cannot be in the other clauses by Lemma 7.1. The three remaining clauses contain at least six different variables other than x by the same argument that C_1, C_2 and C_3 in Fig. 7.1 contain at least six different variables.

If F contains a variable x occurring at least four times that does not satisfy the previous case, we pick four of the clauses containing x. We want to count the number of different variables other than x in these four clauses. Since none of the clauses are 3-clauses by the previous case and only 5-clauses can share more than one variable, any clause that is not a 5-clause contains at least three variables not in the others. If there are any 5-clauses, the first one contains four variables other than x, a possible second 5-clause can contain one from the first, so it has at least three other variables, a third can contain one from each of the others so it contains at least two other variables and a fourth contains at least one. The only case with fewer than eleven variables is thus if all four clauses are 5-clauses that pairwise have two variables in common as in Fig. 7.3. Then

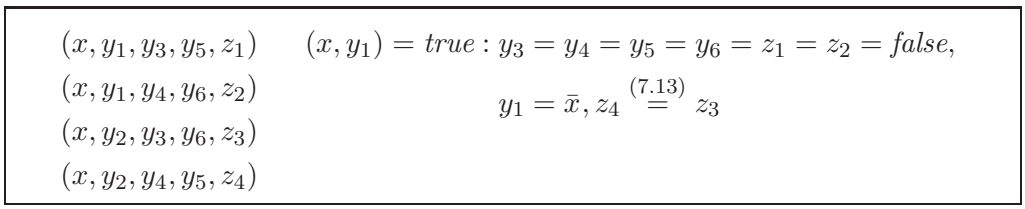


Figure 7.3. A $(4, 0)$-occurrence x with only ten variables.

the algorithm branches on (x, y_1). When it is set to *true*, y_1 is removed by (7.4) and the six other variables in the first two clauses are set to *false* by (7.6). The last two clauses reduce to (x, y_2, z_3) and (x, y_2, z_4), but then $z_3 = z_4$ by (7.13) and (7.4). So we get a branching vector of at least $(8, 2)$.

Lemma 7.6 *If a reduced formula only contains* $(1, 0)$*-,* $(2, 0)$*- and* $(3, 0)$*-occurrences and it contains two variables* x *and* y *that occur in a clause together and they both occur in a clause without the other, setting* (x, y) *to true removes both* x *and* y*.*

Proof. When the algorithm sets (x, y) to *true* then $y = \bar{x}$ by (7.4) and this makes x a $(1, 1)$-, $(2, 1)$- or $(2, 2)$-occurrence, which we remove by resolution ((7.10) or (7.11)). □

Remark 7.1 *If two variables that are not unique, occur together in a clause that is not a 5-clause, we can use Lemma 7.6 on these two variables as they cannot occur together in another clause. If we have a 4-clause* (y_1, y_2, y_3, y_4) *and we set* (y_1, y_2) *to false, this is the same as setting* (y_3, y_4) *to true, so we can use Lemma 7.6 on these two variables.*

Two 5-clauses having exactly two variables in common

Suppose F contains two 5-clauses having two variables in common as the first two clauses in Fig. 7.4(a). If both x_1 and x_2 occur in a clause without the

$(x_1, x_2, y_1, y_2, y_3)$	$(x_1, x_2, y_1, y_2, y_3)$
$(x_1, x_2, y_4, y_5, y_6)$	$(x_1, x_2, y_4, y_5, y_6)$
(x_1, y_1, C')	(x_1, z_1, z_2, z_3)
(a) The third clause contains y_1.	(b) The third clause contains no y_i.

Figure 7.4. Two 5-clauses having two variables in common.

other, the algorithm branches on (x_1, x_2). Setting it to *true* removes all eight variables in the two clauses by Lemma 7.6, so this yields a branching vector of at least $(8, 2)$.

By (7.9), at least one of x_1 and x_2 must occur in another clause, so assume we have another clause with x_1. Then x_1 is a $(3, 0)$-occurrence. The third clause with x_1 can have at most one variable in common with each of the first two clauses apart from x_1 and if there is such a variable, the third clause must be a 5-clause.

The third clause with x_1 contains y_1 as in Fig. 7.4(a). Now, C' contains three variables, one of which can be y_4, y_5 or y_6. If neither y_2 nor y_3 is unique, the algorithm branches on (y_2, y_3). Setting it to *true* makes x_1, x_2 and y_1 *false*. Now, if y_2 and y_3 do not occur together in another clause, setting (y_2, y_3) to *true* removes both y_2 and y_3 by Lemma 7.6 for a total of five variables. If y_2 and y_3 do occur in another clause together this clause must be a 5-clause that does not contain x_1, x_2 and y_1 so we remove at least three other variables and y_3 by (7.4) for at total of seven variables. Setting (y_2, y_3) to *false* removes y_2 and y_3, so we have the clauses in Fig. 7.5; then we apply the reductions shown in the figure and remove x_2 and y_1. In total, we get a branching vector of at least $(5, 4)$.

$$\begin{array}{l} (x_1, x_2, y_1) \\ (x_1, x_2, y_4, y_5, y_6) \\ (x_1, y_1, C') \end{array} \overset{2\times(7.13)}{\longrightarrow} \begin{array}{l} (x_1, x_2, y_1) \\ (\bar{y}_1, y_4, y_5, y_6) \\ (\bar{x}_2, C') \end{array} \overset{2\times(7.10)}{\longrightarrow} (x_1, y_4, y_5, y_6, C')$$

Figure 7.5. The clauses from Fig. 7.4(a) when $y_2 = y_3 =$ *false*.

In the other case, one of y_2 and y_3 (say y_3) is unique, but not y_2 by (7.9). Now, y_1 occurs in no other clauses; otherwise, x_1 and y_1 are two variables occurring twice together and both in a clause without the other, which is the first case above. The algorithm branches on y_2. Setting it to *true* removes at least seven variables, since y_2 occurs in another clause, which contains at least two variables not in the first clause with y_2 by Lemma 7.1. Setting y_2 to *false* leaves the three clauses in Fig. 7.6, where y_3 is unique, and x_1, x_2 and y_1 only occur in these three clauses; then we reduce the formula by replacing these three clauses by the two on the right: any truth assignment satisfying the original formula with x_1 *true* will satisfy the new formula if y_1 and x_2 are both changed to *true* and a satisfying assignment with x_1 *false* will also satisfy the new formula. On the other hand, a satisfying truth assignment to

$$\begin{array}{lcl} (x_1, x_2, y_1, y_3) & & (x_2, y_4, y_5, y_6) \\ (x_1, x_2, y_4, y_5, y_6) & \rightarrow & (y_1, C') \\ (x_1, y_1, C') & & \end{array}$$

Figure 7.6. Special reduction.

the new formula with both x_2 and y_1 *true* is a satisfying assignment to the original formula if x_1 is set to *true* and x_2, y_1 and y_3 are set to *false*. All other assignments satisfying the new formula can be changed to satisfy the original one by setting x_1 to *false* and choosing the value of y_3 such that (x_2, y_1, y_3) is satisfied. By using this reduction, both x_1 and y_3 are removed, so we get a branching vector of at least $(7, 3)$.

The third clause with x_1 contains none of the y_is. If the third clause is not a 4-clause, the algorithm branches on x_1. If the third clause is a 3-clause, this yields a branching vector of at least $(10, 2)$ and if it is at least a 5-clause, this yields a branching vector of at least $(12, 1)$.

The remaining case is depicted in Fig. 7.4(b); none of the z_is are unique by (7.9), since the two other clauses with x_1 contain x_2, which is in no other clauses. If one of z_1, z_2 and z_3 occurs three times, say z_1, the algorithm branches on x_1, z_1. Setting x_1 to *true* removes eleven variables and setting z_1 to *true* removes at least eight, since the first clause with z_1 is a 4-clause, so it does not share any variables other than z_1 with the other two clauses with z_1 and these two clauses each contain at least two variables not in the other by Lemma 7.1. Setting x_1 and z_1 to *false* removes all four variables in the third clause by Remark 7.1. So we get a branching vector of at least $(11, 8, 4)$.

In the last case, z_1, z_2 and z_3 all occur exactly twice in F; then the algorithm branches on (x_1, z_1). All four variables in the third clause are removed in both branches by Remark 7.1. Let the other clause containing z_1 be (z_1, C'). If it is a 3-clause, we remove an extra variable setting z_1 to *false*. If C' contains one of the y_is, then setting (x_1, z_1) to *true*, we get a clause where this y_i occurs twice after resolution ((7.10)), and the y_i is removed by (7.5). Both cases result in a branching vector of at least $(5, 4)$. Otherwise, (z_1, C') is at least a 4-clause containing none of the y_is, but in that case when we set (x_1, z_1) to *true* and apply (7.10), we get the clauses (C', x_2, y_1, y_2, y_3) and (C', x_2, y_4, y_5, y_6). These two clauses contain at least seven variables each and have all but three of them in common. We then further branch on (C', x_2) and get a branching vector of at least $(6, 4)$ (see Fig. 7.2). In total, we get a branching vector of at least $(10, 8, 4)$ ($(4, 4)$ followed by $(6, 4)$ in one branch).

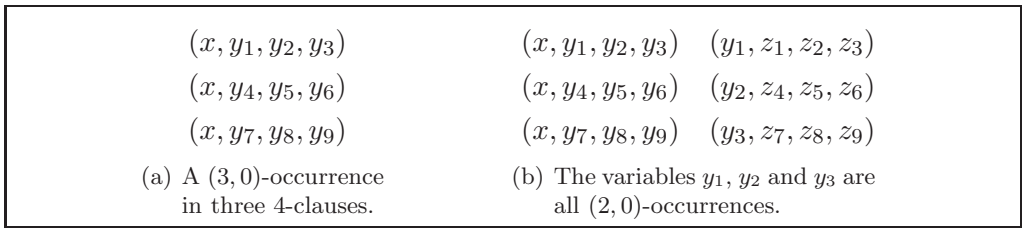
Variables occurring three times

When the algorithm reaches this point, every two clauses share at most one variable and the only variables occurring more than twice in F are $(3, 0)$-occurrences.

If F contains a variable occurring three times and not in three 4-clauses

or two 4-clauses and a 5-clause, the algorithm branches on it. This yields a branching vector of at least $(7,4)$, $(8,3)$, $(9,2)$ or $(12,1)$ depending on the number of 3-clauses the variable is in.

A $(3,0)$-occurrence in three 4-clauses. Suppose F contains a $(3,0)$-occurrence x, which is in three 4-clauses as in Fig. 7.7(a). Not all the clauses can



(a) A $(3,0)$-occurrence in three 4-clauses.

(b) The variables y_1, y_2 and y_3 are all $(2,0)$-occurrences.

Figure 7.7. A $(3,0)$-occurrence x in three 4-clauses.

contain a unique variable or F would have been reduced by (7.9), so assume that the first clause contains no unique variables. If y_1 occurs with at least four other variables, we branch on x, y_1. Setting x to *true* removes ten variables, setting y_1 to *true* removes at least eight and setting both to *false* removes four by Remark 7.1. In total, we get a branching vector of at least $(10,8,4)$.

We can assume now, that y_1, y_2 and y_3 are all $(2,0)$-occurrences and that their other clauses are at most 4-clauses; otherwise, one of the y_is would occur with at least four other variables, since clauses share at most one variable, but this is the previous case. If y_1 is in a 3-clause, the algorithm branches on (x, y_1); in both branches all variables in the clause with x and y_1 are removed by Remark 7.1. Setting (x, y_1) to *false* also removes one of the other variables from the 3-clause by (7.4). This yields a branching vector of at least $(5,4)$.

If none of the previous cases apply, the other clauses containing y_1, y_2 and y_3 must be the ones in Fig. 7.7(b), where some of the z_is can be one of y_4 to y_9 and some of them can be the same variable. By (7.9), at most one z_i from each clause is unique. Suppose two z_is from different clauses (say z_1 and z_4) are unique; then branching on (y_1, y_2) will remove x, y_1, y_2 and y_3 in both branches by Remark 7.1 and setting (y_1, y_2) to *true* also makes z_1 and z_4 end up in the same clause and one is removed by (7.9). This also yields a branching vector of at least $(5,4)$.

We can assume now, that say z_1, z_2 and z_3 are not unique and if any of them are a y_i, then z_1 is y_4. The algorithm branches on (y_1, z_1); in both branches y_1, z_1, z_2 and z_3 are removed by Remark 7.1. If z_1 was y_4, setting (y_1, z_1) to *true* makes $y_4 = \bar{y}_1$, so x is *false* by (7.7). This yields a branching vector of at least $(5,4)$. If z_1 is not y_4, setting (y_1, z_1) to *false* reduces the first clause with x to (x, y_2, y_3). The algorithm then branches on x, which yields a branching vector of at least $(9,3)$, since when x is set to *false*, y_2 and y_3 are removed by Remark 7.1. In total, this yields a branching vector of at least $(13,7,4)$.

A $(3,0)$-occurrence in two 4-clauses and a 5-clause. We have now removed all $(3,0)$-occurrences except those in two 4-clauses and one 5-clause.

If we have such a variable x and one of the 4-clauses contains another $(3,0)$-occurrence y_1 we branch on x, y_1. Setting one of the $(3,0)$-occurrences to *true* removes eleven variables and setting both to *false* removes three by (7.4), so we get a branching vector of at least $(11, 11, 3)$.

If we have a $(3,0)$-occurrence x in two 4-clauses and a 5-clause and one of the 4-clauses (x, y_1, y_2, y_3) contains only $(2,0)$-occurrences, i.e. no unique variables, other than x, we can branch as in the previous section with three 4-clauses: if one of the y_is, say y_1, is in a 5-clause the algorithm branches on x, y_1 and get $(11, 8, 4)$. If y_1 is in a 3-clause we branch on (x, y_1) and get $(5, 4)$ as before. Now, as in the previous section, we must have the clauses in Fig. 7.7(b), except that the last clause with x contains one more variable. The only branch which involves this 5-clause is the last, and there we just get $(10, 3)$ instead of $(9, 3)$ when branching on x after having branched on (y_1, z_1) so we get a branching vector of at least $(14, 7, 4)$ in total.

Sparse formulas. The only remaining case with variables occurring more than twice is a $(3,0)$-occurrence in two 4-clauses and one 5-clause, where both 4-clauses contain a unique variable and the other variables in the 4-clauses occur twice in F. Then F is 5-sparse, i.e., it contains at least four variables occurring at most twice for each variable occurring three times: we only count the variables in the 4-clauses. Each variable occurring three times occurs with two unique variables and four $(2,0)$-occurrences in the two 4-clauses. The $(2,0)$-occurrences might be in another clause with a variable occurring three times, but if this clause is a 4-clause it can contain at most one $(3,0)$-occurrence. Thus, we have at least two $(2,0)$-occurrences and two unique variables for each variable occurring at least three times, so the formula is 5-sparse and the algorithm solves the remaining formula in time $O(2^{n/5})$, where n is the number of remaining variables.

7.3.3 The algorithm for X3SAT

In this section, we give our algorithm for X3SAT and show that it achieves a branching vector of $(12, 4)$ corresponding to a running time of $O(2^{0.1379n})$. The previously best algorithm is by Dahllöf, Jonsson and Beigel [13] and has branching vector $(10, 4)$ corresponding to a running time of $O(2^{0.1532n})$.

Extra reductions. For X3SAT we have some extra reductions which are only needed to remove certain cycles. We do not use them in the reduction procedure, but rather apply them, when they are needed.

We are concerned with cycles because, if we have k clauses and a variable in each is set to *false*, we would normally remove another variable from each of the remaining 2-clauses by (7.4), but if some of the clauses form a cycle on the variables not set to *false*, we may remove one less variable. As an example, $F[z_1 \leftarrow z_2, z_2 \leftarrow z_3, \ldots, z_{k-1} \leftarrow z_k, z_k \leftarrow z_1]$ only removes $k-1$ variables from F.

Reductions (7.19) and (7.23) do not remove any variables and we will also refer to them as transformations. They are only used, when they allow us to apply another reduction or get a previous case: this means that we call the

algorithm recursively on the transformed formula; either a variable is removed by the reduction procedure or the algorithm will branch on one of the previous cases.

The first two reductions remove k-cycles with k or $k-1$ negations, the third some special k-cycles.

$$(y_1, \bar{z}_1, z_2) \wedge (y_2, \bar{z}_2, z_3) \wedge \cdots \wedge (y_k, \bar{z}_k, z_1) \wedge F' \rightarrow F[y_1, y_2, \ldots, y_k \leftarrow \mathit{false}] \tag{7.16}$$

$$(y_1, z_1, z_2) \wedge (y_2, \bar{z}_2, z_3) \wedge \cdots \wedge (y_k, \bar{z}_k, z_1) \wedge F' \rightarrow F[z_1 \leftarrow \mathit{false}] \tag{7.17}$$

$$(\tilde{y}_1, \tilde{z}_1, z_2) \wedge (\tilde{y}_2, \tilde{z}_2, z_3) \wedge \cdots \wedge (\tilde{y}_k, \tilde{z}_k, z_1) \wedge F' \rightarrow F[x \leftarrow \mathit{false}] \tag{7.18}$$
each y_i occurs unnegated in a clause with the literal x and the parities of k and the number of negations are different

If there is a 3-cycle with one negation, we can use (7.19) to either add a clause with the three variables not in the cycle or if all the four clauses are there remove any one of them. If the 3-cycle has a unique variable in the clause without the negated variable, we can remove this clause by (7.20). If u is not unique, but also occurs in $(\bar{u}, y_2, y_3)$, but in no other clauses, we can remove that clause by (7.19) and still use (7.20).

$$(y_1, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (y_3, \bar{z}_3, z_1) \wedge F' \leftrightarrow (y_1, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (y_3, \bar{z}_3, z_1) \wedge (\bar{y}_1, y_2, y_3) \wedge F' \tag{7.19}$$

$$(u, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (y_3, \bar{z}_3, z_1) \wedge F' \rightarrow (y_2, z_2, z_3) \wedge (y_3, \bar{z}_3, z_1) \wedge F' \quad (u \text{ unique}) \tag{7.20}$$

If two 3-cycles without negations share two clauses, we can reduce the formula by (7.21) or (7.22).

$$(y_1, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (y_3, z_3, z_1) \wedge (y_1, y_2, z_4) \wedge F' \rightarrow (z_1, z_2, z_3) \wedge F'[y_1 \leftarrow z_3, y_2 \leftarrow z_1, y_3 \leftarrow z_2, z_4 \leftarrow z_2] \tag{7.21}$$

$$(y, z_1, z_2) \wedge (y, z_3, z_4) \wedge (y, z_5, z_6) \wedge (z_1, z_3, z_5) \wedge F' \rightarrow F[y \leftarrow \mathit{false}] \tag{7.22}$$

If we have a 3-cycle with no negations and one of the variables is unique, we can transform the formula by (7.23).

$$(y_1, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (u, z_3, z_1) \wedge F' \rightarrow (y_1, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (\bar{y}_1, z_3, u) \wedge F' \quad (u \text{ unique}) \tag{7.23}$$

If there is a 4-cycle with two negations and the negated variables occur nowhere else, the formula can be reduced by (7.24) or (7.25) (w is a new variable).

$$(y_1, z_1, z_2) \wedge (y_2, \bar{z}_2, z_3) \wedge (y_3, z_3, z_4) \wedge (y_4, \bar{z}_4, z_1) \wedge F' \rightarrow (w, z_1, z_3) \wedge (\bar{w}, y_1, y_2) \wedge (\bar{w}, y_3, y_4) \wedge F' \quad (z_2, z_4 \notin V(F')) \tag{7.24}$$

$$(y_1, z_1, z_2) \wedge (y_2, z_2, z_3) \wedge (y_3, \bar{z}_3, z_4) \wedge (y_4, \bar{z}_4, z_1) \wedge F' \rightarrow (y_1, z_1, z_2) \wedge (\bar{y}_1, y_2, w) \wedge (\bar{w}, y_3, y_4) \wedge F' \quad (z_3, z_4 \notin V(F')) \tag{7.25}$$

Lemma 7.7 *Reductions* (7.16) *to* (7.25) *are sound.*

Proof. In (7.16), the z_is are either all *true* or all *false* or there would be a clause with two true literals. So all the y_is must be *false*.

In (7.17), if z_1 is *true* z_2 must be *false*, but then also z_3 must be *false* and the remaining z_is must be *false*; then in the last clause, both z_1 and $\bar{z}_k$ are *true*, which is a contradiction.

Reduction (7.18) is proved with a simple counting argument: let n_1 be the number of y_is that are negated and n_2 the number of z_is occurring negated. In a satisfying assignment with x *true*, n_1 of the clauses will be satisfied by the y_is, n_2 of the clauses will be satisfied by the z_is occurring negated, and an even number of clauses will be satisfied by the z_is occurring only unnegated. This is only possible, if the parities of k and $n_1 + n_2$ are the same.

To prove the soundness of (7.19), we prove that all assignments satisfying the left hand side of the reduction will also satisfy the right hand side (the opposite is trivial). If y_1 is *true*, z_1 and z_2 must be *false* and y_2 and y_3 must have different values, so $(\bar{y}_1, y_2, y_3)$ is satisfied. If y_1 is *false*, $z_2 = \bar{z}_1$ and y_2 and y_3 are both *false* by (7.8), so $(\bar{y}_1, y_2, y_3)$ is also satisfied in this case.

As u is unique in (7.20), the first clause just ensures that z_1 and z_2 are not both *true*, but this is also ensured by the two other clauses, as z_1 is in a clause with $\bar{z}_3$ and z_2 with z_3, so they cannot both be *true*.

In (7.21), setting $y_1 = \bar{z}_3$ leads to a contradiction: by the first and second clause z_2 is *false* and by the second and fourth clause y_2 is *false*, which makes z_3 *true*; now, z_1 should both be *false* (by the third clause) and *true* (by the first clause), so in a satisfying assignment $y_1 = z_3$; then $y_2 = z_1$ by the first and second clause, $z_4 = z_2$ by the second and fourth clause and $y_3 = z_2$ by the second and third clause. With these substitutions all four clauses have become (z_1, z_2, z_3), and three of the copies are discarded.

In (7.22), setting y to *true* will set all the z_is to false, but then the clause with only z_is is not satisfied; thus, y must be *false*.

In (7.23), the last clause on the left just ensures that not both z_1 and z_3 are *true* and the last clause on the right that not both $\bar{y}_1$ and z_3 are *true*. But by the first two clauses z_1 and z_3 are both *true* if and only if $\bar{y}_1$ and z_3 are both *true*, so we can replace the last clause on the left by the last clause on the right.

In (7.24), as z_2 does not occur elsewhere the first two clauses just ensure, that exactly one of y_1, y_2, z_1 and z_3 is *true*. This is also achieved by the clauses (w_1, z_1, z_3) and $(\bar{w}_1, y_1, y_2)$ (w_1 is a new variable). Similarly, the last two clauses can be replaced by (w_2, z_1, z_3) and $(\bar{w}_2, y_3, y_4)$, but then $w_1 = w_2$ by (7.14) and we get the three clauses on the right hand side of (7.24). Similarly in (7.25), the last three clauses just ensure that exactly one of y_2, y_3, y_4, z_1 and z_2 is *true*, but this can also be expressed by the clauses (w_1, z_1, z_2), $(\bar{w}_1, y_2, w_2)$ and $(\bar{w}_2, y_3, y_4)$; then $w_1 = y_1$ by (7.14), so we get the three clauses on the right hand side. □

General branching

Now, we state our algorithm for X3SAT. If we have an (a, b)-occurrence x occurring in the clauses in Fig. 7.8, we let $Y_1 = \{y_1, y_2, \dots\}$ be the set of variables that occur in a clause with x, $Y_2 = \{y'_1, y'_2, \dots\}$ those that occur in a clause with $\bar{x}$ and $Y = Y_1 \cup Y_2$. We let ys be variables in Y_1, y's be variables

(x, y_1, y_2)	$(\bar{x}, y'_1, y'_2)$	$x = \mathit{true} : y_1 = y_2 = \cdots = y_{2a-1} = y_{2a} = \mathit{false},$
(x, y_3, y_4)	$(\bar{x}, y'_3, y'_4)$	$y'_2 = \bar{y}'_1, y'_4 = \bar{y}'_3, \dots, y_{2b} = \bar{y}'_{2b-1}$
$\vdots$	$\vdots$	$x = \mathit{false} : y_2 = \bar{y}_1, y_4 = \bar{y}_3, \dots, y_{2b} = \bar{y}_{2b-1},$
(x, y_{2a-1}, y_{2a})	$(\bar{x}, y'_{2b-1}, y'_{2b})$	$y'_1 = y'_2 = \cdots = y'_{2a-1} = y'_{2a} = \mathit{false}$

Figure 7.8. Branching on an (a, b)-occurrence x.

in Y_2, zs be variables that are not x and not in Y and ws be variables that are not x.

If we branch on x as illustrated in Fig. 7.8, we get a branching vector of at least $(2a + b + 1, 2b + a + 1)$ from the above clauses. If $a + b \geq 5$, this yields a branching vector of at least $(11, 6)$, $(10, 7)$ or $(9, 8)$. For variables occurring fewer times, we also need to consider the other clauses in which the ys and y's occur. We start with a lemma showing some cases, in which we can reduce F.

Lemma 7.8 *If a reduced formula F contains a clause with three variables from Y that is not $(\bar{y}_1, y_3, y_5)$ or $(\bar{y}'_1, y'_3, y'_5)$ or if F contains a clause $(\bar{y}_1, \bar{y}_3, z_1)$, $(\bar{y}'_1, \bar{y}'_3, z_1)$ or $(\tilde{y}_1, \tilde{y}'_1, z_1)$, where at least one of y_1 and y'_1 is negated, F can be reduced.*

Proof. We only prove the lemma for clauses with at least as many variables from Y_1 as from Y_2. The result for clauses with less variables from Y_1 than Y_2 then follows by looking at $\bar{x}$ instead of x, as this swaps Y_1 and Y_2.

If F contains a clause with y_1 and y'_1 where at least one of them is negated, it contains the 3-cycle consisting of (x, y_1, y_2), $(\bar{x}, y'_1, y'_2)$ and $(\tilde{y}_1, \tilde{y}'_1, w)$, which has two or three negations and we reduce it by (7.16) or (7.17).

If F contains a clause with y_1 and y_3 where both of them are negated, it contains the 3-cycle consisting of (x, y_1, y_2), (x, y_3, y_4) and $(\bar{y}_1, \bar{y}_3, w)$, which has two negations and we reduce it by (7.17).

If F contains the clause (y_1, y'_1, y_3), then it contains the 3-cycle (x, y_1, y_2), $(\bar{x}, y'_1, y'_2)$ and (y_1, y'_1, y_3). We add the clause $(y_2, y'_2, \bar{y}_3)$ by (7.19) and have the first case. The only case left is if F contains the clause (y_1, y_3, y_5); then, x must be *false* by (7.22). □

If x is a $(3, 1)$-occurrence or a $(2, 2)$-occurrence, one of the variables in one clause with $\bar{x}$ say y'_1 must occur in another clause by Lemma 7.2 and the clause must be $(\tilde{y}'_1, w, z_1)$ by Lemma 7.8, since y'_1 cannot occur with two other y's as x only occurs negated in at most two clauses. From the clauses in Fig. 7.8, we get a branching vector of at least $(8, 6)$ or $(7, 7)$, but setting x to *false*, also removes z_1 by (7.1) or (7.4) and we get a branching vector of at least $(8, 7)$.

Now, we have removed all variables occurring at least four times in the formula, except $(4,0)$-occurrences.

Branching on $(2,1)$-occurrences

By Lemma 7.2, at least one y from each clause with x and two from one are in other clauses. We want to show, that in all cases we can either reduce F or branch on x and get a branching vector of at least $(8,7)$ or $(9,6)$. From the clauses with x, we get a branching vector of at least $(6,5)$ (see Fig. 7.8). We want to show, that we can always remove at least four more variables in total in the two branches from the other clauses with the variables from Y. First, we prove two lemmas showing, when we can reduce the formula.

Lemma 7.9 *If a reduced formula F contains the clause $(\tilde{y}_1, \tilde{y}'_1, z_1)$ and a clause containing z_1 and a variable from $\{y_1, y_2, y'_1, y'_2\}$ and these clauses are not (y_1, y'_1, z_1) and $(y_2, y'_2, \bar{z}_1)$, F can be reduced.*

Proof. By Lemma 7.8, if the first clause is not (y_1, y'_1, z_1), F can be reduced. If F does not contain the clause $(y_2, y'_2, \bar{z}_1)$, we add it by (7.19) (used on (x, y_1, y_2), $(\bar{x}, y'_1, y'_2)$ and (y_1, y'_1, z_1)) and since the second clause was not this one we have two clauses sharing at least two variables and we reduce F by (7.7), (7.8) or (7.14). □

(x, y_1, y_2)	(y_1, y'_1, z_1)	(x, y_1, y_2)	$(\bar{y}_1, y_3, z_1)$	(x, y_1, y_2)	(y_1, y_3, z_1)
(x, y_3, y_4)		(x, y_3, y_4)		(x, y_3, y_4)	
$(\bar{x}, y'_1, y'_2)$		$(\bar{x}, y'_1, y'_2)$		$(\bar{x}, y'_1, y'_2)$	
(a)		(b)		(c)	

Figure 7.9. A $(2,1)$-occurrence x and a clause with two variables from Y.

Lemma 7.10 *If a reduced formula F contains the clause $(\tilde{y}_1, \tilde{y}_3, z_1)$ and a clause containing z_1 and a variable from $\{y_1, y_2, y_3, y_4\}$ and these clauses are not $(\bar{y}_1, y_3, z_1)$ and $(y_2, \bar{y}_4, z_1)$, F can be reduced.*

Proof. By Lemma 7.8, at most one of y_1 and y_3 is negated. Suppose one is negated, then we have the clauses in Fig. 7.9(b). If the other clause is not $(y_2, \bar{y}_4, z_1)$, we add this clause by (7.19) (used on the three topmost clauses). Now, the other clause with z_1 and one of the ys will share at least two variables with one of the other two clauses with z_1 and we reduce by (7.7), (7.8) or (7.14).

Suppose we have the clauses in Fig. 7.9(c). By symmetry, we can assume that the second clause with z_1 is $(\tilde{y}_2, \tilde{z}_1, w)$. If neither y_2 nor z_1 is negated, we have two 3-cycles, which share two clauses and we can reduce F by (7.21) and if both are negated we have a 3-cycle with two negations, so we reduce F by (7.17). If only one is negated we have a 3-cycle with one negation and we add a clause with x, y_3 and w by (7.19) (used on (x, y_1, y_2), (y_1, y_3, z_1) and $(\tilde{y}_2, \tilde{z}_1, w)$), where one of x and y_3 is negated and then x and y_3 occur together

in two clauses and one of them is negated in one of the clauses, so we can reduce F by (7.7). □

Note, that we cannot have a clause with three variables from Y when x is a $(2,1)$-occurrence, and if two variables from Y occur together in a clause with a z, it must be in one of the combinations in Fig. 7.9 by Lemma 7.8. Now we first look at the three cases in Fig. 7.9, and then at the case where no two variables from Y occur together in a clause with a z.

F contains (y_1, y'_1, z_1) as in Fig. 7.9(a). Using (7.19) on the 3 clauses (x, y_1, y_2), $(\bar{x}, y'_1, y'_2)$ and (y_1, y'_1, z_1), we can transform F to contain one or both of (y_1, y'_1, z_1) and $(y_2, y'_2, \bar{z}_1)$.

Now, if there is only one other clause, C, with variables from Y, we can reduce F: C must contain either y_3 or y_4 by Lemma 7.2, since they would otherwise both be unique, but C does not contain both by (7.7), (7.8) and (7.14). Then C contains at most one of y_1, y_2, y'_1 and y'_2 by Lemma 7.8, since x is a $(2,1)$-occurrence, so we can choose to let F contain only one of the above clauses with z_1 such that at most three variables from Y occur in clauses without x. Then x occurs only in clauses with a unique variable and we can reduce F by Lemma 7.2.

Suppose that F contains two other clauses with variables from Y and the first contains z_2, the second contains z_3 and z_1, z_2 and z_3 are different variables; then branching on x yields $(8,7)$ or $(9,6)$: in both branches, y_1 or y'_1 is set to *false* in (y_1, y'_1, z_1), so z_1 is removed by (7.4). In each of the two other clauses with variables from Y, one variable is removed in one of the branches by (7.1) or (7.4).

Suppose, on the other hand, that F does not contain two such clauses with variables from Y and different zs, none of which are z_1. Now z_1 is in no more clauses with y_1, y_2, y'_1 or y'_2 by Lemma 7.9 and if it is in a clause with y_3, the clause contains no other variable from Y. Suppose F contains the clause $(\tilde{y}_3, \tilde{z}_1, z_2)$; then z_1 can be in no more clauses with variables from Y: if it is in a clause with y_4 that clause must contain a different z, but that is the previous case. F must contain another clause with a variable from Y and since it does not contain z_1 or any z_3, it must contain z_2 and two variables from Y. Using Lemmas 7.9 and 7.10, we have that these two variables must be one from Y_1 and one from Y_2 and two that do not already occur together. This clause must then be (y_1, y'_2, z_2), $(y_1, y'_2, \bar{z}_2)$, (y_2, y'_1, z_2) or $(y_2, y'_1, \bar{z}_2)$. All four cases are symmetric, so assume, we have the first clause, we can then add the last by (7.19). Now, we have the clauses in Fig. 7.10, but then we have a 3-cycle

(x, y_1, y_2)	(y_1, y'_1, z_1)	$(\tilde{y}_3, \tilde{z}_1, z_2)$	(y_1, y'_2, z_2)
(x, y_3, y_4)	$(y_2, y'_2, \bar{z}_1)$		$(y_2, y'_1, \bar{z}_2)$
$(\bar{x}, y'_1, y'_2)$			

Figure 7.10. A special case for $(2,1)$-occurrence x, in which we can reduce.

with two negations which we remove by (7.17): the 3-cycle contains the bottom

clause in the fourth column, the clause in the third column and one of the clauses in the second column (which one depends on whether z_1 is negated in $(\tilde{y}_3, \tilde{z}_1, z_2)$).

If the two other clauses with variables from Y do not contain z_1 and not two different zs, they must be of the form $(\bar{y}_1, y_3, z_2)$ and $(y_2, \bar{y}_4, z_2)$ or (y_3, y_1', z_2) and $(y_4, y_2', \bar{z}_2)$ by Lemmas 7.9 and 7.10, but then we remove one of the clauses by (7.19) and have the previous case with only one other clause with variables from Y. This completes all cases with the clauses in Fig. 7.9(a).

F contains $(\bar{y}_1, y_3, z_1)$ as in Fig. 7.9(b). If y_1 is only in the clause $(\bar{y}_1, y_3, z_1)$ and the one with x, we have a 3-cycle with one negation and we can apply (7.19) twice; first to add the clause $(y_2, \bar{y}_4, z_1)$ and then to remove the clause $(\bar{y}_1, y_3, z_1)$. Then y_1 is unique and occurs in the new 3-cycle with one negation (on y_4) and we can reduce F by (7.20). If y_1 is in another clause, it must be a $(2,1)$-occurrence and since x and y_3 are in a clause together, we have the previous case with y_1 acting as x.

F contains (y_1, y_3, z_1) as in Fig. 7.9(c). If y_2 is unique, we have a 3-cycle with no negations and a unique variable and we transform F by (7.23) and replace the clause (x, y_1, y_2) by $(x, \bar{z}_1, y_2)$, but then we have the previous case. Now, y_2 and by symmetry also y_4 must occur in another clause. If they occur together in a clause, they must occur unnegated or we have the previous case, but then we have two 3-cycles without negations, which share two clauses and we reduce F by (7.21). If they occur in different clauses, their clauses do not contain z_1 by Lemma 7.10 and no y' or negated y by the two previous cases, so they must contain two different zs. As before, this yields a branching vector of at least $(9,6)$ branching on x: when x is set to *true*, we remove z_1 and the two different zs occurring with y_2 and y_4, and either y_1' or y_2' also occurs with some z, which is removed when x is set to *false*.

F contains no clauses with two ys and a z. If no two variables from Y occur in the same clause without x, we get $(9,6)$ or $(8,7)$ branching on x: at least two of the variables in Y_1 and at least one of those in Y_2 must be in another clause, which removes at least two extra variables in the *true* branch and one in the *false* branch. Now, at least one more of the variables from Y must be in another clause, so in total we have four clauses with a variable from Y and two zs. This means that an extra variable is removed in one branch, unless there are three clauses with a variable from Y_1 and the zs in these three clauses form a 3-cycle, but then F can be reduced by Lemma 7.11.

Lemma 7.11 *If a reduced formula F contains a 3-cycle $(\tilde{y}_1, \tilde{z}_1, z_2)$, $(\tilde{y}_2, \tilde{z}_2, z_3)$ and $(\tilde{y}_3, \tilde{z}_3, z_1)$, F can be reduced.*

Proof. If more than one of the z_is are negated, we have a 3-cycle with two or three negations and F is reduced by (7.16) or (7.17), and if exactly one is negated, we use (7.19) on these three clauses to add a clause with y_1, y_2 and y_3, but y_1 and y_2 are already together in a clause with x, so we can reduce F

by (7.7), (7.8) or (7.14). If none of the z_is are negated we look at whether y_1 and y_2 are negated. If both are negated the clauses (x, y_1, y_2), $(\bar{y}_1, z_1, z_2)$ and $(\bar{y}_2, z_2, z_3)$ constitute a 3-cycle with two negations, which we reduce by (7.17) and if none of them are negated, we have two 3-cycles with no negations sharing two clauses and the formula is reduced by (7.21). If either y_1 or y_2 is negated, we add a clause with x, z_1 and z_3 by (7.19) (used on (x, y_1, y_2), $(\tilde{y}_1, z_1, z_2)$ and $(\tilde{y}_2, z_2, z_3)$) where one of z_1 and z_3 is negated, and then we have two clauses with both z_1 and z_3 and one of them is negated in the new clause, so we reduce the formula by (7.7). □

Branching on $(4, 0)$-occurrences

To get the desired branching vector for $(4, 0)$-occurrences, we show that if there are no variables occurring both unnegated and negated except $(1, 1)$-occurrences, we can extend Lemma 7.8 and reduce in all cases with a clause with three ys.

Lemma 7.12 *If a reduced formula F containing only $(a, 0)$- and $(1, 1)$-occurrences contains a clause $(\tilde{y}_1, \tilde{y}_3, \tilde{y}_5)$ or a clause $(\bar{y}_1, \tilde{y}_3, w)$, F can be reduced.*

Proof. The only case not covered by Lemma 7.8 is if F contains the clause $(\bar{y}_1, y_3, w)$, where w is either y_5 or a z. Now, y_1 must be a $(1, 1)$-occurrence, and we have a 3-cycle with one negation, so we apply (7.19) twice and replace this clause by $(y_2, \bar{y}_4, w)$ and then y_1 is unique and occurs in the new 3-cycle with one negation and we reduce F by (7.20). □

If x is a $(4, 0)$-occurrence, branching on x yields a branching vector of at least $(9, 5)$ from the clauses in Fig. 7.8. At least five of the ys must occur in another clause by Lemma 7.2, but then they must occur with at least two different zs or we reduce F by Lemmas 7.10 or 7.12. When we set x to *true* all the ys are *false* and at least two zs are removed, so we get a branching vector of at least $(11, 5)$.

Branching on $(3, 0)$-occurrences

When we look at a $(3, 0)$-occurrence x, we know by Lemma 7.12 that no three variables from Y occur together and if two occur together they must both occur unnegated.

F contains a clause with two ys and a z. Suppose F contains the clause (y_1, y_3, z_1) and two of y_2, y_4 and z_1 are unique, then we have a 3-cycle with no negations and a unique variable and we use (7.23) (with the other unique variable as y_2 in (7.23)) to get a 3-cycle with one negation and a unique variable and we remove one of the unique variables by (7.20). If only one of y_2, y_4 and z_1 is unique, we use (7.23) and get a $(2, 1)$-occurrence, which is a previous case. If z_1 is in a clause with any of the variables y_1, y_2, y_3 or y_4, we reduce F by Lemma 7.10. If two of y_1, y_2, y_3 and y_4 occur together in a second clause they

must both be unnegated by Lemma 7.12, but then we can also reduce F: if two ys, which already occur together in a clause with x, also occur together in a second clause, we reduce by (7.14). If two ys, which occur in different clauses with x, occur together we have two 3-cycles without negations sharing two clauses, and we reduce by (7.21) or (7.22). In the last case, we must have the clauses in Fig. 7.11, where z_2 is a new variable, and w can be either

(x, y_1, y_2)	(y_1, y_3, z_1)	$y_2 = \mathit{true} : x = y_1 = \mathit{false}, y_4 = \bar{y}_3, y_6 = \bar{y}_5, z_1 = \bar{y}_3$
(x, y_3, y_4)	$(\tilde{y}_2, w, z_2)$	$y_2 = \mathit{false} : y_1 = \bar{x}, y_3 \overset{(7.7)}{=} \mathit{false}, y_4 = \bar{x}, z_1 = \bar{y}_1$
(x, y_5, y_6)		$\tilde{y}_2 = \mathit{true} : w = z_2 = \mathit{false}$
		$\tilde{y}_2 = \mathit{false} : z_2 = \bar{w}$

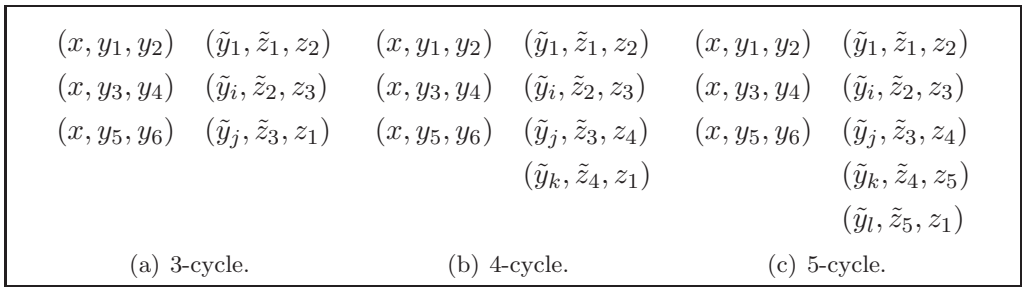
Figure 7.11. A clause with two ys. We branch on y_2.

y_5 (it cannot be negated by Lemma 7.12) or z_3 (another new variable). The algorithm branches on y_2. Let us first look at what happens in the first four clauses; setting y_2 to *true* removes the six variables depicted in Fig. 7.11, and setting y_2 to *false* makes $y_1 = \bar{x}$. Then we have the two clauses (x, y_3, y_4) and $(\bar{x}, y_3, z_1)$ which makes y_3 *false* by (7.7) and we remove the last two variables shown in the second line in Fig. 7.11 for at total of five variables. Now, we look at what happens with the clause $(\tilde{y}_2, w, z_2)$; when $\tilde{y}_2$ is set to *false*, we remove z_2, and when $\tilde{y}_2$ is set to *true*, both w and z_2 are set to *false*. This removes two additional variables in this branch: w is either y_5 or z_3, but neither has gotten a value in any of the branches. If w is y_5, the clause has two ys and z_2 cannot be unique, as this is the previous case. Now, either z_2 is in another clause or w is z_3 and is in another clause. This clause can at most contain one of y_1, y_2, y_3, y_4 or z_1 by Lemma 7.10, so we remove an extra variable from this clause when z_2 or z_3 is set to *false*. In total, we remove three variables in one branch and one in the other. This yields a branching vector of at least $(9, 6)$ or $(8, 7)$.

Cycles. At this point, the only clauses containing ys, except the clauses with x, contain only one y. By Lemma 7.2, at least four of the ys are not unique, so there must be at least four such clauses. We want to branch on x; when we set it to *true*, all the ys are set to *false* and the literals from clauses with unnegated ys are set to the negation of each other and the ones from clauses with negated ys are set to *false*. This removes at least as many extra variables as there are clauses with ys, unless some of these clauses form a cycle as in Fig. 7.12. We are only concerned with 3-, 4- and 5-cycles; we cannot have 2-cycles, as these would have been removed by (7.7), (7.8) or (7.14) and if we have at least a 6-cycle, we remove at least five zs, when we set x to *true*, but this yields a branching vector of at least $(12, 4)$, which is what we are after.

Lemma 7.13 *If a reduced formula does not satisfy any of the previous cases, the same y cannot occur twice in a 3-, 4- or 5-cycle.*

Proof. If a y occurs twice in a 3-, 4- or 5-cycle, it occurs at least three times in F so it must be a $(3, 0)$-occurrence. It cannot occur in two neighbouring clauses



(a) 3-cycle. (b) 4-cycle. (c) 5-cycle.

Figure 7.12. Cycles.

in the cycle by (7.7) and (7.14); since we are dealing with at most 5-cycles, it must then occur in two clauses $(y_1, \tilde{z}_1, z_2)$ and $(y_1, \tilde{z}_3, z_4)$, where z_2 and z_3 are together in another clause in the cycle. Then y_1 is a $(3, 0)$-occurrence with two of the variables it occurs with occurring together in another clause, but that is a previous case. □

Lemma 7.14 *If a reduced formula F containing only $(a, 0)$- and $(1, 1)$-occurrences contains two clauses $(\tilde{y}_i, z_1, z_2)$ and $(\tilde{y}_j, \tilde{z}_2, z_3)$, where at least two of y_i, y_j and z_2 are negated, we can reduce F.*

Proof. If y_i and y_j are from the same clause with x, that clause and the two clauses in the lemma form a 3-cycle with two or three negations, so we can reduce F by (7.16) or (7.17). If y_i and y_j are from different clauses with x, the clauses form a 4-cycle with two or three negations together with the two clauses where y_i and y_j occur with x. If there are three negations we reduce F by (7.17), and if there are two we reduce F by (7.24) or (7.25), since we do not have $(2, 1)$-occurrences. □

In the following, we show how to deal with the remaining cases of 3-, 4- and 5-cycles.

3-cycles. If F contains a 3-cycle as in Fig. 7.12(a), the ys must be from different clauses with x by Lemma 7.11. If the cycle contains more than one negated z_i, the formula is reduced by (7.16) or (7.17) and if there is exactly one negated z_i, we add a clause with three ys by (7.19). If this clause does not contain exactly one negation, we reduce F by Lemma 7.8; otherwise, the negated y has become a $(2, 1)$-occurrence, which is a previous case. Now, suppose that none of the z_is are negated; if at least two of the ys are negated, we reduce F by Lemma 7.14 and if none of the ys are negated we have a 3-cycle with no negations, and each of the ys occurring with x, so we reduce F by (7.18). So assume the cycle consists of the clauses in Fig. 7.13; then we branch on y_1 and get a branching vector of at least $(9, 7)$, as shown in the figure. The (7.7) above the equality means, that this follows from (7.7): since $z_2 = \bar{z}_1$, we get two clauses with z_1 and z_3 and z_1 is negated in one of them.

4-cycles. Suppose F contains a 4-cycle as in Fig. 7.12(b). If there are two or more negated z_is, we reduce the formula by (7.16), (7.17), (7.24) or (7.25),

$$\begin{array}{llll} (x,y_1,y_2) & (\bar{y}_1,z_1,z_2) & y_1 = \mathit{true} : x = y_2 = \mathit{false}, y_4 = \bar{y}_3, y_6 = \bar{y}_5, z_2 = \bar{z}_1, \\ (x,y_3,y_4) & (y_3,z_2,z_3) & \qquad z_3 \overset{(7.7)}{=} \mathit{false}, z_2 = \bar{y}_3, z_1 = \bar{y}_5 \\ (x,y_5,y_6) & (y_5,z_3,z_1) & y_1 = \mathit{false} : y_2 = \bar{x}, z_1 = z_2 = \mathit{false}, z_3 = \bar{y}_5 = \bar{y}_3, \\ & & \qquad y_4 \overset{(7.14)}{=} y_6 \end{array}$$

Figure 7.13. A 3-cycle with only y_1 negated. We branch on y_1.

since F contains no $(2,1)$-occurrences. If there is only one negated z_i, the two ys in the clauses with the negated variable must be unnegated by Lemma 7.14; let these clauses be (y_1, z_1, z_2) and $(y_i, \bar{z}_2, z_3)$. If an even number of the ys in the cycle are negated, we reduce the formula by (7.18); otherwise, there must be exactly one negated y and the cycle looks like the one in Fig. 7.14(a), but then $x = z_1$. Suppose $x = \bar{z}_1$; then we replace z_1 by $\bar{x}$ and get that both y_1

$$\begin{array}{llllll} (x,y_1,y_2) & (y_1,z_1,z_2) & (x,y_1,y_2) & (y_1,z_1,z_2) & (x,y_1,y_2) & (y_1,z_1,z_2) \\ (x,y_3,y_4) & (y_i,\bar{z}_2,z_3) & (x,y_3,y_4) & (y_2,z_2,z_3) & (x,y_3,y_4) & (y_3,z_2,z_3) \\ (x,y_5,y_6) & (\bar{y}_j,z_3,z_4) & (x,y_5,y_6) & (y_3,z_3,z_4) & (x,y_5,y_6) & (y_2,z_3,z_4) \\ & (y_k,z_4,z_1) & & (y_i,z_4,z_1) & & (y_4,z_4,z_1) \end{array}$$

(a) $x = z_1$. (b) $z_2 = z_4$. (c) Branch on x.

Figure 7.14. 4-cycles.

and y_k are in clauses with both x and $\bar{x}$, so they must be *false* by (7.7), but then $z_2 = x$, so y_i must also be *false* by (7.7). Now, both z_3 and z_4 must be equal to x, but then x must be *false* by (7.5) and y_j must also be *false*. This is a contradiction: since none of the ys are the same by Lemma 7.13, at least two of them are from the same clause with x, but they are all *false* and so is x.

Suppose that none of the z_is in the 4-cycle are negated. If an odd number of the ys in the cycle are negated we reduce F by (7.18) and if two ys in neighbouring clauses are negated, we reduce F by Lemma 7.14. If two ys in "opposite" clauses in the cycle are negated, x must be *false*: if x is *true*, all ys are set to *false* and by the two clauses with negated ys all the z_is are set to *false*, but then the other two ys must be *true*, a contradiction, so x must be *false*. In the remaining cases, none of the ys in the cycle are negated. If two ys in neighbouring clauses are from the same clause with x we have the cycle in Fig. 7.14(b), but then $z_2 = z_4$: suppose $z_2 = \bar{z}_4$; then z_1 and z_3 must be *false* by (7.7), but then $y_1 = y_2 = \bar{z}_2 = z_4 = \bar{y}_3$. Then y_1 and y_2 must be *false*, but then both x and y_3 must be *true*, a contradiction.

Lemma 7.15 *If a reduced formula F contains a 4-cycle as in Fig. 7.14(c), F is satisfiable iff F with the last clause in the cycle removed is satisfiable.*

Proof. It is trivially true, that if F is satisfiable, so is F with the last clause removed. Suppose that F without the last clause is satisfied. If x is *true*, all the ys are *false*, so from the other three clauses, we have that $z_1 = \bar{z}_2 = z_3 = \bar{z}_4$

so the fourth clause is satisfied. If x is *false*, $y_1 = \bar{y}_2$, so exactly one of the z_is are *true*; if it is one of z_1 or z_4, y_3 must be *true* by the second clause in the cycle so y_4 must be *false* and if it is one of z_2 or z_3, y_3 must be *false* and hence y_4 must be *true* by their common clause with x. In both cases, the last clause is satisfied. □

The only remaining 4-cycles have no negations and no y occurring more than once. Then at least two of the ys must be from the same clause with x and they are not in neighbour clauses in the cycle.

If the other two ys in the cycle are not from the same clause with x the 4-cycle must look like the one in Fig. 7.14(c), except y_4 is replaced by y_5. Then we add the clause (y_4, z_4, z_1) by Lemma 7.15 and get that $y_4 = y_5$ by (7.14).

If the other two ys in the cycle are also from the same clause with x, we have the 4-cycle in Fig. 7.14(c). Now y_5 or y_6 (say y_5) must also occur in a clause with two zs. If both these two zs are one of z_1, z_2, z_3 and z_4 they must be z_1 and z_3 or z_2 and z_4 by Lemma 7.2, but then we have a 3-cycle, which is a previous case. So y_5 must occur with a new variable z_5. If neither y_1 nor y_2 occurs in other clauses than the ones in the figure, our algorithm branches on x. Setting x to *true* removes all the ys and three of z_1, z_2, z_3 and z_4 plus another variable from the clause with y_5 for a total of eleven variables. When setting x to *false* the four clauses in the cycle turn into (y_1, z_1, z_2), (y_3, z_2, z_3), $(\bar{y}_1, z_3, z_4)$ and $(\bar{y}_3, z_4, z_1)$. The first of these can be removed by (7.19) and then y_1 is unique and the third can be removed by (7.23), which removes y_1 from the formula. This removes five variables for a branching vector of at least $(11, 5)$. If on the other hand y_1 occurs in another clause, the clause cannot contain z_1 and z_2 by Lemma 7.2 and if it contained z_3 or z_4, y_1 would be a $(3, 0)$-occurrence, with two of its ys occurring together (z_2 and z_3 or z_1 and z_4), which is a previous case. The third clause with y_1 may contain z_5, but must also contain another variable z_6. Branching on x then yields a branching vector of at least $(12, 4)$, as three of z_1, z_2, z_3 and z_4 as well as a variable from each of the clauses with y_1 and y_5 are removed when x is set to *true*.

F contains a clause with a negated variable from Y. There must be at least four clauses with variables from Y other than the ones with x. We branch on x; setting it to *false* removes four variables and setting it to *true* sets all the ys to *false* and in each of the other clauses with ys either sets one of the zs to the negation of the other or both to *false*. This removes at least twelve variables, as at least one y is negated and the zs do not form a 3- or 4-cycle, so we get a branching vector of at least $(12, 4)$.

5-cycles. If F contains a 5-cycle as in Fig. 7.12(c), then none of the ys are negated by the previous case; also, none of the ys in the cycle are the same by Lemma 7.13, so there must be four clauses in the 5-cycle, as in Fig. 7.15, that contain only ys from two clauses with x. The ys will always satisfy an even number of these clauses: if x is *true*, they satisfy zero and if x is *false* they satisfy two, as they pairwise become each others negation, since they were from only two different clauses with x. Now, let us look at z_2, z_3 and z_4. The

(x, y_1, y_2)	(y_1, z_1, z_2)	An even number of the z_is are negated : $z_1 = z_5$
(x, y_3, y_4)	$(y_i, \tilde{z}_2, z_3)$	An odd number of the z_is are negated : $z_1 = \bar{z}_5$
(x, y_5, y_6)	$(y_j, \tilde{z}_3, z_4)$	
	$(y_k, \tilde{z}_4, z_5)$	

Figure 7.15. Four clauses from a 5-cycle.

negated ones will always satisfy exactly one of the clauses and the unnegated will either satisfy zero or two; thus, if the number of negated z_is is odd, z_1 and z_5 must satisfy an odd number of the clauses for F to be satisfiable, so $z_1 = \bar{z}_5$ and if the number of negated z_is is even, $z_1 = z_5$ by the same argument. In both cases, we have reduced F.

F contains at least five clauses with ys and zs. Since the clauses do not form 3-, 4- or 5-cycles at this point, branching on x yields at least $(12, 4)$.

F contains exactly four clauses with ys and zs. As at least four variables from Y occur in clauses without x and no two variables from Y occur together (in a clause with a z), there must be exactly four clauses with a y and two zs. Furthermore, two of the variables from Y are unique, and the others are $(2, 0)$-occurrences; otherwise, there would be more than four clauses with variables from Y. Since there are no cycles, we have already seen how to get $(11, 4)$ branching on x. If the formula is $(15/2)$-sparse, we can solve the remaining formula in time $O(2^{2n/15}) = O(2^{0.13333n})$. We want to branch on formulas with too many $(3, 0)$-occurrences; we either prove that we remove an extra variable when branching on x or we branch on a different variable.

Suppose x_1 is a $(3, 0)$-occurrence occurring in the three first clauses in Fig. 7.16. The other clause with y_3 contains no unique variable by Lemma 7.2. If it contains a variable x_2, that only occurs in other clauses with unique variables as in Fig. 7.16, we branch on x_1; setting it to *true* removes eleven variables

(x_1, y_1, y_2)	(x_2, y_3, z)	All other clauses with x_2
(x_1, y_3, u_1)		contain a unique variable.
(x_1, y_5, u_2)		

Figure 7.16. Unique variables.

and setting it to *false* removes x_1, y_2, u_1 and u_2, but then y_3 is unique, so all clauses with x_2 contain a unique variable and we set x_2 to *false* by (7.15). Then also $z = \bar{y}_3$ and we get a branching vector of at least $(11, 6)$.

If another $(3, 0)$-occurrence x_2 occurs with one of the variables from Y it must be either y_1 or y_2: suppose x_2 is in a clause with y_3. Now, y_3 is a $(2, 0)$-occurrence with a unique variable in its clause with x_1, so its other clause cannot contain unique variables; then the remaining clauses with x_2 must contain unique variables, but that is the previous case. Suppose y_1 is in a clause with x_2, then that clause cannot contain a unique variable, as this would also

be the previous case (by looking at x_2 instead of x_1). The two other clauses with x_2 must then contain a unique variable, so they do not contain y_3 or y_5 and if they contain y_2, we have a $(3,0)$-occurrence with two of the variables it occurs with occurring together in another clause, which is a previous case. So we must have the clauses in Fig. 7.17 (to easier distinguish variables occurring with different $(3,0)$-occurrences, we use y' and y'' to denote variables occurring with other $(3,0)$-occurrences than x_1 in the rest of this section). Then we

(x_1, y_1, y_2)	(y_1, x_2, y_2')	(x_2, y_3', u_1')	$y_2 = \mathit{true} : x_1 = y_1 = z_3 = z_4 = \mathit{false},$
(x_1, y_3, u_1)	(y_2, z_3, z_4)	(x_2, y_5', u_2')	$u_1 = \bar{y}_3, u_2 = \bar{y}_5, y_2' = \bar{x}_2$
(x_1, y_5, u_2)	(y_3, z_5, z_6)		$y_2 = \mathit{false} : y_1 = \bar{x}_1, z_4 = \bar{z}_3, x_2 \overset{(7.15)}{=} \mathit{false},$
	(y_5, z_7, z_8)		$y_2' = \bar{y}_1, u_1' = \bar{y}_3', u_2' = \bar{y}_5'$

Figure 7.17. One variable from Y occurs with another $(3,0)$-occurrence.

branch on y_2; setting it to *true* removes eight variables as shown in Fig. 7.17 and setting it to *false*, we get $z_4 = \bar{z}_3$ and $y_1 = \bar{x}_1$ as shown in the figure. Then we have $\bar{x}_1$ in a clause with x_2 and both of them only occur unnegated in clauses with unique variables elsewhere, so x_2 is set to *false* by (7.15), and we remove the remaining variables shown in the second case in the figure for a branching vector of at least $(8,7)$.

Now, no variable from Y occurs with another $(3,0)$-occurrence. Suppose F contains a $(3,0)$-occurrence x_1 occurring in the clauses in the first two columns in Fig. 7.18. If both z_5 and z_6 occur with a $(3,0)$-occurrence, then z_5 and z_6 cannot be negated, as a $(3,0)$-occurrence where one of the variables it occurs with occurs negated in another clause is a previous case; also, the clause with z_5 and z_6 contains y_3, which is only in one other clause and that clause contains a unique variable, so if the clause with z_5 (or z_6) and the $(3,0)$-occurrence contains a unique variable, we have the case in Fig. 7.16 (with the $(3,0)$-occurrence as x_1 and z_5 as y_3), which we have handled. Now, z_5 and z_6 cannot occur in another clause together, so we must have all the clauses in Fig. 7.18. We branch on z_6;

(x_1, y_1, y_2)	(y_1, z_1, z_2)	(x_2, z_5, y_2')	(x_3, z_6, y_2'')
(x_1, y_3, u_1)	(y_2, z_3, z_4)	(x_2, y_3', u_1')	(x_3, y_3'', u_1'')
(x_1, y_5, u_2)	(y_3, z_5, z_6)	(x_2, y_5', u_2')	(x_3, y_5'', u_2'')
	(y_5, z_7, z_8)		

$z_6 = \mathit{true} : y_3 = z_5 = x_3 = y_2'' = \mathit{false}, u_1 = \bar{x}_1, y_2' = \bar{x}_2, u_1'' = \bar{y}_3'', u_2'' = \bar{y}_5''$

$z_6 = \mathit{false} : z_5 = \bar{y}_3, y_2'' = \bar{x}_3, x_2 \overset{(7.15)}{=} \mathit{false}, y_2' = \bar{z}_5, u_1' = \bar{y}_3', u_2' = \bar{y}_5'$

Figure 7.18. Both z_5 and z_6 occur with a $(3,0)$-occurrence.

setting it to *true* removes nine variables as shown in the figure, and setting z_6 to *false* sets $y_2'' = \bar{x}_3$ and $z_5 = \bar{y}_3$. Now, x_2 is in a clause with $\bar{y}_3$ and both variables only occur unnegated in clauses with unique variables elsewhere, so we

set x_2 to *false* by (7.15) and remove y'_2, u'_1 and u'_2. In total, we get a branching vector of at least $(9, 7)$.

Sparse formulas. Now, no y can occur with another $(3, 0)$-occurrence, so there are at least six variables occurring at most twice for each $(3, 0)$-occurrence. Also, as z_5 and z_6 (and by symmetry z_7 and z_8) do not both occur with a $(3, 0)$-occurrence, we can assume that neither z_5 nor z_7 occurs with a $(3, 0)$-occurrence. As they both occur at most twice, they are in clauses with at most four different variables; thus, we count each of them as one fourth of a variable for each of the $(3, 0)$-occurrences, whose y they occur with. This means, that there are at least six and a half variables occurring at most twice for each $(3, 0)$-occurrence; thus, the formula is $(15/2)$-sparse and we solve it in time $O(2^{2n/15}) = O(2^{0.13333n})$.

7.4 Conclusion

The main result of this paper is the following theorem.

Theorem 7.1 *The algorithms for XSAT and X3SAT run in time $O(2^{0.2325n})$ and $O(2^{0.1379n})$, respectively.*

Proof. When our algorithms are applied to a formula F with m clauses and n variables, the sizes of the intermediate formulas are never larger than the maximum of $|F|$ and $2mn$: they are never larger during the reduction procedures by Lemma 7.4 and after the reduction procedures have run, the formula has size at most mn. In some of the branches we add a clause, but when we call the algorithm recursively, the reduction procedure will remove a clause, so the size is never larger than $\max(|F|, 2mn)$, which is polynomial in the size of the original formula. Also, no reduction or branching adds variables. The number of recursive calls are at most $O(2^{0.2325n})$ and $O(2^{0.1379n})$, respectively, by Sections 7.3.2 and 7.3.3. For each recursive call, the reduction procedure runs in polynomial time in the size of the formula by Lemma 7.5 and we can in polynomial time decide, which case to branch on. Since we ignore the polynomial factors, we get the stated running times. □

Both our algorithms are extensions of known branch-and-reduce algorithms. One important addition to the algorithms are new reductions, which limit the number of possible structures of the formula. The other important addition is the concept of sparse formulas, which in certain situations enables us to simply enumerate all possible assignments to the variables we would otherwise branch on and leaves us with a problem that is solvable in polynomial time. One could hope that the concept of sparse formulas is also useful in other algorithms.

Acknowledgements

We would like to thank the anonymous referees for many helpful comments. We are also grateful to our supervisors Peter Bro Miltersen and Sven Skyum for their help with this paper.

Chapter 8

Automated Generation of Branching Algorithms

This chapter contains the paper "Automated Generation of Branching Algorithms with Upper Bound Proofs for Variants of SAT" [52].

Abstract

Automated generation of algorithms and corresponding upper bound proofs for NP-complete problems is a very new topic in computer science. In this paper we present a program generating algorithms and corresponding upper bound proofs for variants of Satisfiability with our main focus on Exact Satisfiability, the variant of Satisfiability where a clause is satisfied if exactly one of its literals is *true*. We describe several new techniques which we use in the program, e.g. a technique to avoid duplicate cases and a technique to find new reductions, and we present some results which we have obtained with the program.

8.1 Introduction

In recent years there has been an increased interest in algorithms generating algorithms and upper bound proofs of the corresponding running time for different problems in NP. In 2003 Nikolenko and Sirotkin [41] presented an automated proof of an upper bound of $O(2^{0.5284m})$, where m is the number of clauses, of an algorithm solving SAT. As Hirsch [27] had previous shown an algorithm with an upper bound of $O(2^{0.3090m})$ this was not an improvement for solving SAT, but they obtained the bound for a much simpler algorithm, which as the only reduction used pure literal elimination. Also in 2003, Gramm et al. [25] made automated generation of algorithms and corresponding upper bounds for various graph modification problems. For several of the problems they consider, the upper bound the program computes is better than the previous best known upper bound for the problem. In 2004 Fedin and Kulikov [23] considered the problems SAT, MAXSAT and $(n, 3)$-MAXSAT, the variant of MAXSAT where each variable occurs at most thrice, and made automatically generated algorithms and corresponding upper bound proofs. For $(n, 3)$-MAXSAT they improve the previous best known bound to $O(2^{0.3620n})$, where n is the number of variables in the formula.

In this paper we consider variants of SAT, and especially Exact Satisfiability (XSAT). XSAT is the variant of SAT where a clause is satisfied if *exactly* one literal is *true*. For more information on XSAT see e.g. [10]. We take a different approach than all the previous programs generating algorithms and proofs. In all previous programs the reductions they use are fixed. In this paper the program starts without any predefined reductions and instead the program finds reductions by itself. The program of Nikolenko and Sirotkin [41] and the program of Fedin and Kulikov [23] tries to prove a predetermined upper bound, and the program by Gramm et al. uses fixed sized graphs in the case analysis. The program presented in this paper on the other hand will run infinitely, and will always try to improve one of the worst known cases, outputting an algorithm whenever it finds an algorithm which it can prove better than the previous outputted algorithms. The main new ideas in the program is thus to automatically generate reductions, and to always improve the worst case in the analysis. Furthermore we present a technique to avoid solving cases which are subcases of already solved cases.

8.1.1 Definitions

We are given a set of variables, which we will denote by using the letters x and y. A *literal* is either a variable x or the negation of a variable $\bar{x}$; we use the letter l to denote a literal. A clause is a collection of literals, written as $(l_1, l_2, \ldots, l_3)$; we use the letter C to denote clauses. Sometimes, we will think of a clause as a set of literals (actually a multiset, since a clause can contain more than one of each literal). A formula is a set of clauses usually written as $C_1 \wedge C_2 \wedge \cdots \wedge C_m$; we use the letter F to denote formulas.

We will use three different measures in the upper bounds. The number of variables, which we denote n, the number of clauses, m, and the number of literals, l. For an integer k, we will use $[k]$ to denote the set $\{1, 2, \ldots, k\}$.

We will use $O(f)$ to denote a function that is of the same order as f, ignoring polynomial factors. Note that when used on exponential functions this only differs from the normal definition if exact numbers are used, as we with the normal definition have: $O(p(n) \cdot c^{\alpha n}) \subseteq O(c^{(\alpha+\epsilon)n})$.

8.1.2 Branching vectors

The algorithms the program in this paper produces are branching algorithms. This is also the case for all the previous programs mentioned in the introduction. Branching algorithms solves a problem by branching into several smaller problems in such a way that the solution to the original problem can be computed in polynomial time, given the solutions to the smaller problems. Thus, the algorithm recursively solves each of the smaller problems, and combines the solutions to get a solution for the original problem. We let $\mu(F)$ be a measure of the size of the formula F.

If the algorithm when finding the solution for F constructs the formulas $F_1, F_2, \ldots, F_k$, we let $t_i = \mu(F) - \mu(F_i)$ for $i \in [k]$ and we call $t = (t_1, t_2, \ldots, t_k)$ the *branching vector* of this branch. We then get a recursion for the running

time of the form $T(\mu(F)) = T(\mu(F)-t_1)+T(\mu(F)-t_2)+\cdots+T(\mu(F)-t_k)+O(1)$, which has the solution $T(\mu(F)) = \alpha_t^{\mu(F)}$, where α_t is the positive root of $1 - 1/x^{t_1} - 1/x^{t_2} - \cdots - 1/x^{t_k}$. We call α_t the *branching value* of t. The running time of the whole algorithm is $O(2^{\log_2 \alpha \cdot \mu(F)})$, where α is larger than α_t, for every branch vector t occurring in the algorithm. Proofs of these results can be found in a manuscript by Kullmann and Luckhardt [32].

8.2 Algorithm

8.2.1 class of formulas

Fedin and Kulikov [23] introduced the concepts of a *clause with unfixed length* and a *class of formulas*. In this paper we will extend the latter and we will use the term *open clause* instead of *clause with unfixed length*. An *open clause* containing the literals $l_1, l_2, \ldots, l_k$ is written $(l_1, l_2, \ldots, l_k, \ldots)$ (e.g. $(x_1, x_2, \bar{x}_3, \ldots)$) and describes a clause containing $l_1, l_2, \ldots, l_k$ and possible some additional literals. We thus say that a formula contains a specific open clause if it contains a clause containing all the literals in the open clause. An open clause is a generalisation of the normal notion of clause, and we will in the following use *clause* to denote either a normal clause or an open clause. If we want to emphasise that a clause is normal we will describe it as a *closed clause*.

A *pattern* is a set of clauses $C_1, C_2, \ldots, C_k$ and it *matches* all formulas which contain all the clauses (some of which can be open). We write a pattern as $C_1 \wedge C_2 \wedge \cdots \wedge C_k \cdots$. A *class of formulas* is defined by a tuple where the first component is a set of variables, and the second component is a pattern. The variables in the set of variables will be called *closed* and all other variables occuring in the pattern will be called *open*. The formulas belonging to a class of formulas defined by a set of variables and a pattern, are all formulas that match the pattern and where the closed variables only occur as specified by the pattern. More formally a formula $C'_1 \wedge C'_2 \wedge \cdots \wedge C'_{k'}$ belongs to the class of formulas $(V, C_1 \wedge C_2 \wedge \cdots \wedge C_k \cdots)$ if and only if there exists a surjective function, f, mapping $[k]$ to $[k']$ (corresponding to a mapping from the clauses of the patterns to the clauses of the formula) such that

- $C_i \subseteq C'_{f(i)}$ for all $i \in [k]$ where C_i is an open clause,
- $C_i = C'_{f(i)}$ for all $i \in [k]$ where C_i is a closed clause,
- $(C_i \setminus C'_{f(i)}) \cap V = \emptyset$ for all $i \in [k]$ and
- $C'_i \cap V = \emptyset$ for all $i \in [k'] \setminus Img(f)$, where $Img(f)$ denotes the image of f i.e. the set $\{f(i) | i \in [k]\}$.

When comparing classes of formulas, we will ignore variable naming, i.e. $(\emptyset, (x, x, y))$ describes the same class as $(\emptyset, (x, y, y))$. As classes of formulas are sets of formulas we will use set terminology. If one class of formulas is contained in another class of formulas we will say that it is a *subclass* of the other.

The classes of formulas Fedin and Kulikov [23] use are a special case of the classes described above, as they only have the pattern, and variables occurring in the pattern are not allowed to occur elsewhere in the formulas belonging to the class. This corresponds to choosing the first component of a class of formulas to be the set of all variables occurring in the pattern.

8.2.2 The overall structure

The program works by building a tree where each node is a class of formulas, and a corresponding branching value, i.e. the branching value which the program is able to obtain for all formulas in the class. The branching value will be set to one, if all formulas in the class can be reduced to smaller formulas in polynomial time. The structure of the program is shown in SOLVE.

```
SOLVE():
  Tree root = new Tree
  Priority Queue pq = new Priority Queue
  Real worst = ∞
  root.data = new Pair(new Class of Formulas(∅, (x_1, ...)),∞)
  pq.insert(root)
  while true do
    Tree node = pq.removeLargest()
    if node.data.second < worst then
      OUTPUTALGORITHM(root)
      worst = node.data.second
    end if
    node.children = SPLIT(node)
    for all Tree child in node.children do
      child.data.second = VALUE(child.data.first)
      if child.data.second > 1 then
        pq.insert(child)
      end if
    end for
  end while
```

The program repeatedly takes the class of formulas which has the maximum corresponding branching value. The function then splits the class of formulas into several smaller classes of formulas. The smaller classes are not necessarily disjoint, but their union are equal to the original class. The strategy used by SPLIT to split the class of formulas, may depend on the problem, and we will go into details in Section 8.4.

After the class has been split, the program calculates the corresponding branching values for each subclass, and if the class can not be reduced it is added to the priority queue.

This approach differs from the approach by Fedin and Kulikov in the way it builds the tree. They have a fixed branching value which they want to prove,

and then build the tree in a depth first manner, going deeper until they have a class of formulas for which they can prove the wanted branching value.

The function OUTPUTALGORITHM outputs all the cases the program has met, and for each it also outputs how to branch in order to get the worst case branching value, which have just been found.

8.3 Deciding the branching value

In this section we describe the implementation of VALUE. VALUE takes a class of formulas and first tries to decides if all formulas in the class can be reduced in polynomial time, i.e. all formulas can be replaced with smaller formulas without changing the satisfiability of them. If it determines that all formulas in the class can be reduced it returns this, and otherwise it gives an upper bound on the best branching value that can be obtained for all formulas in the class.

For simplicity we will call the variables occurring in the pattern $x_1, x_2, \ldots, x_v$ and let the closed variables be $x_1, x_2, \ldots, x_c$, where $c \leq v$. We use *full assignment* to describe an assignment to all the variables occurring in the pattern. An assignment which does not make any clause in the pattern unsatisfied, is called a *non-conflicting* assignment. Note that an open clause which is not unsatisfied is not necessarily satisfied (e.g. for XSAT (*false*, *false*, ...) is not unsatisfied but is not satisfied either).

First the program builds a data structure containing all full non-conflicting assignments. The data structure is just a binary tree, and nodes at level i (the root is level 0) corresponds to variable x_{v-i}. The left subtree corresponds to an assignment with x_{v-i} *true* and the right subtree corresponds to an assignment with x_{v-i} *false*. If there are any non-conflicting full assignments the tree has height $v + 1$, and each leaf corresponds to such an assignment. The algorithm FINDASSIGNMENT shows the overall structure of the algorithm used to build the binary tree.

```
Tree FindAssignment(Pattern pattern, Partial Assignment pa, int var):
  if var = 0 then
    return new Leaf
  end if
  Tree tree = new Tree;
  if not Unsatisfiable(pattern, pa[x_var ← true]) then
    tree.left = FindAssignment(pattern, pa[x_var ← true], var-1)
  end if
  if not Unsatisfiable(pattern, pa[x_var ← false]) then
    tree.right = FindAssignment(pattern, pa[x_var ← false], var-1)
  end if
  if tree.left = null and tree.right = null then
    return null
  end if
  return tree
```

The function UNSATISFIABLE(*Pattern*, *Partial Assignment*) returns whether there is a clause in the pattern which is unsatisfied by the partial assignment. If FINDASSIGNMENT is called with the pattern, the empty assignment and v it will return a binary tree with all assignments which do not make a clause unsatisfied.

8.3.1 Reducing

If the call to FINDASSIGNMENT returns the empty tree all formulas in the class of formulas are unsatisfied (as all assignments to the variables in the pattern makes a clause in the pattern unsatisfied), and we return that the formulas in the class can be reduced. In this section all full assignments will be non-conflicting unless stated otherwise, and we will thus just write *full assignment* instead of *full non-conflicting assignment*.

While building the binary tree we collect some auxiliary information, which can be used to decide whether all formulas in the class can be reduced.

1. For each variable, does there exists a full assignment with this variable set to *true* and does there exists a full assignment with this variable set to *false*.

2. For each pair of variables, does there exists a full assignment where these variables are equal and does there exists a full assignment where these variables are different.

3. For each closed variable, can every assignment to the open variables which can be extended to a full assignment, be extended to a full assignment with this closed variable set to *true* and/or *false*.

4. For each pair of variables, of which at least one is closed, can every assignment to the open variables which can be extended to a full assignment be extended to a full assignment where these variables are equal and/or different.

5. For each closed clause, does there exist an unsatisfying full assignment such that this clause is the only clause in the pattern which is unsatisfied.

In order to calculate (1) and (2) we update two tables each time we reach a leaf. After the binary tree has been build, we look at all variables and pairs. If there is a variable which in all full assignments has the same value, we can reduce all formulas in the class by setting this variable to this value. If there is a pair of variables which in all full assignments either are equal or are different, we can reduce the formula by replacing one of the variables with the other or the negation of the other.

We use the information from (3) to decide if we can set one of the closed variables to a constant, without changing whether the formulas can be satisfied or not. If all closed variables only occur in closed clauses and for some closed variable every assignment to the open variables which can be extended to a full assignment can be extended to a full assignment with the closed variable set

to e.g. *true*, we can set the closed variable to *true* without changing whether the formulas can be satisfied: take any formula in the class and any satisfying assignment to the formula. If we discard the assignments to the variables that are closed in the pattern, we know that we can set the closed variable to *true* and extend the assignment, such that none of the clauses in the pattern are unsatisfied. As a closed clause where all variables has been assigned is either satisfied or unsatisfied, all the closed clauses in the pattern are satisfied. As we have only changed the assignment to closed variables and they only occur in closed clauses in the formula, and all such clauses still are satisfied, all clauses in the formula must be satisfied, and we thus have a satisfying assignment, where the closed variable is *true*.

We use the information from (4) in the same way as we use the information from (3), by setting two variables equal or different instead of setting one to *true* or *false*, so we will not describe this in detail. Similarly calculating (4) is done the same way as (3) just using two dimensional tables instead of one dimensional, so for simplicity we will only describe how we calculate (3).

Calculating (3) is a bit more complicated than calculating (1) and (2). We use two tables to calculate (3), one which keeps the overall result and one which is used locally. In the global table each variable is initially marked as being able to be set to both *true* and *false*. For each node in level $v - c$ (corresponding to the first closed variable, and each node thus corresponds to an assignment to the open variables which can be extended to a full assignment) we use the local table to calculate information similar to the information in (1). Thus for each assignment to the open variables which can be extended to a full assignment, we calculate which values each of the closed variables can have in the full assignments. We use this information to update the global table. If a closed variable can have both values we do not change the entry in the global table, but if it can only have one value, we mark in the table that it can not be set to the opposite value. When the binary tree has been build, we can decide (3) from the global table.

We use (5) to decide if we can discard a clause. If there is a closed clause such that every unsatisfying full assignment which makes this clause unsatisfied also makes another clause unsatisfied we can remove this clause without changing the set of valid assignments to the formulas.

For efficiency reasons we do not actually calculate (5), but only a conservative approximation of the answer. If the answer for a clause is *yes*, then we decide this, but if the answer is *no* the program may find the answer to be *yes*. This is not a problem for the correctness as we only reduce if the answer is *no*, so it never makes an invalid reduction, but may miss a possible reduction.

We calculate the approximation of (5), by keeping a table with an entry per clause. Initially the answer is marked as *no* for each clause. Each time UNSATISFIABLE finds that the pattern has an unsatisfied clause with some partial assignment, and it only is one clause that is unsatisfied, it marks that the answer is *yes* for this clause. This will correctly mark all clauses for which the answer is *yes*, but for a clause there can be a partial assignment which only makes that clause unsatisfied, even though there are *no* full assignment only making that clause unsatisfied, thus incorrectly marking the clause as a clause

where the answer is *yes*.

Reducing to another case

Fedin and Kulikov points out several problems and possible extensions to their program. One problem is that they may solve the same classes of formulas more than once, as several of the generated classes may be identical, just by variable renaming. This is a problem for short proofs which potentially could be readable by humans, but suddenly becomes bloated because of duplicate cases, but is an even greater problem for large proofs as the size of the proof and thus the time required to compute it explodes in repeated calculations.

It is not difficult to show that deciding whether two classes of formulas are identical is computationally equivalent to deciding Graph Isomorphism and deciding if one class of formulas is a subclass of another class of formulas is computationally equivalent to deciding Subgraph Isomorphism, which is NP-complete. It is thus computationally very expensive to compare a class of formulas with all previously seen classes of formulas.

When we have a class of formulas, c, we want to check if the class is a subclass of any of the classes which are to the left of it in the tree. We first note that we do not have to compare c to every class of formulas which are left of it in the tree, as if c is a subclass of some class of formulas, c', then it is also a subclass of the parent of c' in the tree. We can thus check if c is a subclass of any of the classes left of it, by for each node on the path from c to the root, checking if c is a subclass of any of the siblings left of the node. If we find that c is a subclass of any of the classes which are left of it in the tree, we return that the formula can be reduced to another case. As the reductions always are to a case left of the current in the tree this does not lead to cyclic arguments in the proof produced.

The effect of these reductions to other cases are that the classes returned by Split are interpreted as being disjoint. If Split splits a class of formulas into $c_1, c_2, \ldots, c_k$ it gets interpreted as $c_1, c_2 \setminus c_1, \ldots, c_k \setminus \bigcup_{i=1}^{k-1} c_i$ because of the reductions.

8.3.2 Finding branching vectors

First phase

If we are unable to reduce, we try to find the best branching vector which can be achieved for all formulas in the class. This part of the program consists of two phases. In the first phase we calculate for each variable an upper bound on the branching value when branching on the variable and for each pair of variables an upper bound on the branching value when branching on whether the variables are equal or different. If this gives a better branching value than the worst case in the priority queue, we return this branching value.

We use the binary tree of full assignments not making any clause in the pattern unsatisfied to calculate the upper bounds. For each branch we traverse the binary tree, disregarding subtrees for which the partial assignment at the root of the subtree contradicts the branch. During the traversal we calculate

similar information to (1), (2), (3) and (4) and after the traversal we can use this information to calculate which variables we can set to constants, and which variables we can replace by other variables. If we are finding the branching values with respect to the number of variables, we use this information directly, and otherwise we us the information and the class of formulas to decide the number of literals or clauses which are removed by the assignment.

Second phase

If we did not find a branching vector better than the worst in the priority queue we go to the second phase, where we try to refine the branches by going into *branch depth* two. Thus for each branch from the first phase we try to branch further; thus getting branches where we either set two variables to a constant, one variable to a constant and two variables to be either equal or different or two pairs of variables either equal or different. Every branch can be described by one of the following cases:

(a) $x_i = \textit{true/false}$ and $x_j = \textit{true/false}$, where $i < j$,

(b) $x_i = \textit{true/false}$ and $x_{j_1} = / \neq x_{j_2}$, where $i \neq j_1$, $i \neq j_2$ and $j_1 < j_2$,

(c) $x_{i_1} = / \neq x_{i_2}$ and $x_{j_1} = / \neq x_{j_2}$, where $i_1 < i_2$, $i_1 < j_1 < j_2$ and $i_2 \neq j_1$.

The last condition in (c), $i_2 \neq j_1$, can be made, as a case with $i_2 = j_1$ is identical to a case where $i_2 = j_2$ instead (e.g. $x_{i_1} = x_{i_2}$ and $x_{i_2} \neq x_{j_2}$, where $i_1 < i_2 < j_2$, is the same as $x_{i_1} \neq x_{j_2}$ and $x_{i_2} \neq x_{j_2}$). The number of variables, literals or clauses the branch removes are calculated in the same way as for the simple branches. If the two conditions in a branch contradicts each other, i.e. there are no leafs in the binary tree for which both conditions hold true, we set the value to infinite, as infinite entries in a branching vector do not effect the value of the branching vector.

A lot of work can be saved by not just naively calculating all possible branches. Many branches can be found by looking at the calculated values for other branches. The conditions stated in (a) - (c) ensures that we do not have cases which are symmetrical, and neither cases which are also covered by a previous case. As each branch consists of two conditions, it may be the case that one condition implies something about the other condition, so there are four special cases of (a), eight of (b) and 15 of (c), where we can use the result from some other case instead of calculating it. We will not go in detail with all the special cases but only take one example from each of the three cases above.

(a) If we in the first phase found that $x_j = \textit{true}$ implies $x_i = \textit{false}$, the value of the branch $x_i = \textit{true}$ and $x_j = \textit{true}$ is infinite, and the value of the branch $x_i = \textit{false}$ and $x_j = \textit{true}$ is the same as the value for the branch $x_j = \textit{true}$ which we calculated in the first phase.

(b) If $x_i = \textit{true}$ implies $x_{j_2} = \textit{false}$ the branch $x_i = \textit{true}$ and $x_{j_1} = x_{j_2}$ has the same value as the branch $x_i = \textit{true}$ and $x_{j_1} = \textit{false}$.

(c) If $x_{i_1} = x_{i_2}$ implies $x_{i_1} = x_{j_1}$, the value for the branch $x_{i_1} = x_{i_2}$ and $x_{j_1} = x_{j_2}$ is, if $j_2 < i_2$, the same as the value for the branch $x_{i_1} = x_{i_2}$ and $x_{j_2} = x_{i_2}$, and if $j_2 > i_2$ the same as the value for the branch $x_{i_1} = x_{j_2}$ and $x_{i_2} = x_{j_2}$.

When the number of variables, literals or clauses each branch removes has been calculated, the values of all branching vectors which correspond to branches where we first branch on either a variable or a pair of variables, and in one of the branches branch further on a variable or a pair of variables are calculated. If any of the branching values are better than the worst case in the priority queue we return the corresponding branching. Otherwise we calculate the values of all vectors which corresponds to branches where we first branch on either a variable or a pair of variables, and in both branches branch further on a variable or a pair of variables (not necessarily the same in both branches). We then return the branching corresponding to the best branching value we have seen (i.e. not only branching values found in the last phase). As the second phase is very computationally expensive, the program provides the option of disabling it.

8.4 Strategy for Split

The strategy which should be used for splitting a class of formulas into subclasses depends on the problem. In our implementation of Split for XSAT and X3SAT we use the following strategy:

- If any clause is open, take such a clause (in fact there can be at most one) and split it into one class where the clause is closed, and one class for every possible addition of a literal (either a literal already occurring in the class or a new one) to the open clause.
- If all clauses are closed, we split in one class where the open variable with the smallest index is closed, one class where we add an open clause with the open variable with the smallest index, and one class where we add an open clause with the negation of the open variable with the smallest index.

The primary property of Split is that the union of all the subclasses equals the original class, and in both of these cases this is clearly the case. In order to optimise the program we actually break this property of Split by removing some of the subclasses in the first case. The subclasses we remove will all occur elsewhere in the tree which we build in Solve. We order the subclasses such that the first subclasses are the ones where the added literal has the smallest index, and the last subclass is the one where we add a new literal (which is assigned the next available index). With this ordering, we know that if we add a literal to the open clause, and the index of the literal is smaller than the index of another literal in the clause, the class is a subclass of some class which is to the left of this clause in the tree, and the subclass will thus be removed in

the next step of the program. There is thus no need to add the classes, so in the first case above we only add literals which have index larger or equal to the other literals in the open clause.

A strategy which works well for one problem, can work arbitrarily bad for another. If we used the strategy above for SPLIT while trying to prove a bound with respect to the number of clauses on 3SAT the program would always have a bad case on the form $(x \vee y_1 \vee y_2) \wedge (x \vee y_3 \vee y_4) \wedge \cdots \wedge (x \vee y_{2i-1} \vee y_{2i})$. For this case the best branching vector it can prove is $(i, 1, 1)$, which has branching value greater than 2, and it is thus not able to prove anything better than $O(2^m)$, which is trivial to obtain. In order to avoid this, SPLIT should either not only add clauses with the variable with the smallest index or it should use the knowledge that all variables can be assumed to occur both negated and unnegated. For SAT it would not be able to prove any bound at all with respect to the number of clauses, as there would always be a worst case on the form $(x_1, x_2, x_3, \ldots, x_i, \ldots)$.

8.5 Results

We have mainly tested the program on X3SAT, where the best known result is an algorithm from 2003 by Byskov, Madsen and Skjernaa [10] with upper bound $O(2^{0.137n})$ (branching vector (12,4)). When our program only tries simples branches (branch depth one) it finds an algorithm for which it can prove an upper bound of $O(2^{0.1450n})$, corresponding to a branching vector of (11,4), in 4.3 seconds. If we use the full algorithm, which uses branch depth two, it takes 114 seconds to prove this bound, and no case actually uses the ability to branch in depth two. In recent years there have been a lot of papers improving the upper bound (e.g. [18], [30], [42], [13]) and even though the algorithm our program finds does not improve the algorithm by Byskov et al., it does improve on all previous algorithms. For X3SAT there is no difference in giving an upper bound on the running time with respect to the number of clauses and giving an upper bound with respect to the number of literals, as in a reduced formula every clause contains exactly three literals. We are not aware of any previous results on upper bounds with respect to either of these measures. Our program is capable of making an algorithm with an upper bound of $O(2^{0.2124m})$ corresponding to a branching vector of (7,3). It reaches this result in just over one minute.

For XSAT our program proves an upper bound of $O(2^{0.2600n})$ corresponding to a branching vector of (10,1). The currently best known algorithm by Byskov, Madsen and Skjernaa has an upper bound of $O(2^{0.2441n})$, corresponding to a branching vector of (8,2).

There is no trivial upper bound with respect to the number of clauses on solving XSAT. Madsen [35,53] has shown an upper bound of $O(m!)$, and Skjernaa [53] have improved it to $O(2^m)$ at the cost of also using $O(2^m)$ space.

Neither with respect to the number of literals are any bound known, but it is not very hard to obtain (19,3). Our program has only proven a bound of (19,2), and it seems like another strategy for splitting is needed in order to

improve this in reasonable time.

We have also made the program generate an upper bound on solving Satisfiability with respect to the number of literals, but is has only proved a branching vector of (9,6), which is worse than the best upper bound by Hirsch [27], $O(2^{0.1030l})$, corresponding to (21,18,21,18). Fedin and Kulikov [23] have obtained the same bound as we do. Also here another strategy for splitting could be useful.

8.6 Limitations and future work

Although we believe that the approach we have taken in this paper can find improved bounds for several variants of SAT, it does have some limitations in the kind of reductions it is able to find. One of the reductions in the paper by Byskov, Madsen and Skjernaa [10] is (a reduction for X3SAT, reduction (24)):

$$(y_1, z_1, z_2) \wedge (y_2, \bar{z}_2, z_3) \wedge (y_3, z_3, z_4) \wedge (y_4, \bar{z}_4, z_1) \wedge F' \rightarrow \\ {\scriptstyle z_2, z_4 \notin V(F')} \qquad (w, z_1, z_3) \wedge (\bar{w}, y_1, y_2) \wedge (\bar{w}, y_3, y_4) \wedge F'.$$

Such a reduction can not be found by the program in this paper as it involves changing some clauses. In order to find such reductions the program would need to, given a class of formulas, search for a pattern which restricts the open variables in exactly the same way as the original pattern, but by using fewer closed variables. Doing this for even small formulas is extremely computationally expensive.

Also reductions like the unique literal rule (a reduction for SAT)

$$\underset{l \in Lit(F) \text{ and } \bar{l} \notin Lit(F)}{F} \rightarrow F[l \leftarrow true]$$

can not be found, as the left hand side of the reduction can not be described by a pattern. The program will of course be able to reduce such cases if l only occurs in closed clauses, but it will spend a lot of effort on such cases. If the program should handle such reductions it could be done by making Split avoid classes of formulas which is known to be reducible.

In this paper we have taken one extreme, in the sense that we did not teach the program any reductions, but let it find them by itself. As has been shown there are reductions which the program can not find, so it would be an obvious idea to extend the program with some predefined reductions, and see if it would make the program run faster (i.e. reach more cases in the same time as the original), have fewer cases or prove better bounds.

The algorithms for XSAT and X3SAT by Byskov, Madsen and Skjernaa use the concept of sparse formulas. A $k-$sparse formula is a formula where the number of variables which occur more than twice is at most n/k. If a formula is $k-$sparse it can be solved in time $O(2^{n/k})$. It would be interesting to extend our program such that it takes sparseness into consideration, in order to see if this can improve the bounds proven.

Chapter 9

Maximal Bipartite Subgraphs of a Graph

This chapter contains the paper "On the Number of Maximal Bipartite Subgraphs of a Graph" [11]. The paper is co-authored by Jesper M. Byskov and Bolette A. Madsen, and is to appear in Journal of Graph Theory. Minor typographical changes have been made compared to the original paper.

Abstract

We show new lower and upper bounds on the maximum number of maximal induced bipartite subgraphs of graphs with n vertices. We present an infinite family of graphs having $105^{n/10} \approx 1.5926^n$ such subgraphs, show an upper bound of $O(12^{n/4}) = O(1.8613^n)$ and give an algorithm that finds all maximal induced bipartite subgraphs in time within a polynomial factor of this bound. This algorithm is used in the construction of algorithms for checking k-colourability of a graph.

9.1 Introduction

In this paper, we show new lower and upper bounds on the maximum number of maximal bipartite subgraphs of a graph (all subgraphs in this paper are induced subgraphs). We provide an infinite family of graphs showing a lower bound of $105^{n/10} \approx 1.5926^n$ improving an earlier bound of $10^{n/5} \approx 1.5849^n$ by Schiermeyer [47]. Schiermeyer also claims an upper bound of $10^{n/5}$, which is invalidated by our new lower bound. Instead, we prove an upper bound of $O(12^{n/4}) = O(1.8613^n)$ and present an algorithm finding all maximal bipartite subgraphs in time within a polynomial factor of this bound. This can be used in an algorithm for deciding 4-colourability of graphs, and the running time is the same as that for finding all maximal bipartite subgraphs, as already noted by Schiermeyer [47]. For 5-colourability we match the best known running time of $O(2.1592^n)$ by Byskov [7].

9.2 Lower bound

We show a lower bound on the maximum number of maximal bipartite subgraphs in any graph by providing an infinite family of graphs with many maximal bipartite subgraphs. The infinite family consists of disconnected copies of a

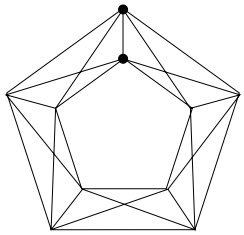
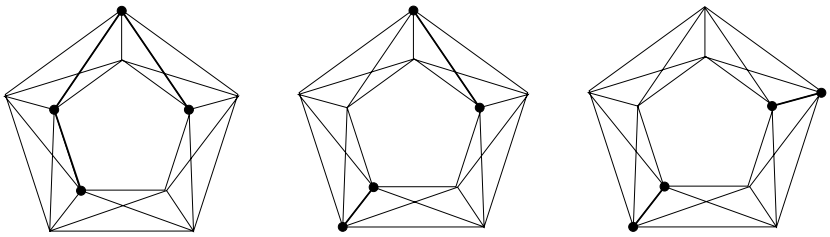


Figure 9.1. Generating graph with a pair marked.



(a) (b) (c)

Figure 9.2. The three different types of maximal bipartite subgraphs.

single graph, the k'th member having k copies. A maximal bipartite subgraph of a disconnected graph is exactly the union of one maximal bipartite subgraph of each connected component. Their number thus equals the product of the number of maximal bipartite subgraphs of each component. Schiermeyer [47] uses K_5 (having ten maximal bipartite subgraphs) to generate his infinite family resulting in a lower bound of $10^{n/5} \approx 1.5849^n$.

Theorem 9.1 *There exists an infinite family of graphs all having* $105^{n/10} \approx 1.5926^n$ *maximal bipartite subgraphs of size* $2n/5$.

Proof. The generating graph for our infinite family of graphs is given in Figure 9.1.[1] Let a *pair* denote a vertex on the outer 5-cycle and the nearest vertex on the inner 5-cycle (see Figure 9.1). The graph has $5 \cdot 2^4 = 80$ maximal bipartite subgraphs containing one vertex from four of the pairs (see Figure 9.2(a)), $5 \cdot 2^2 = 20$ containing one pair and one vertex from each of the opposite pairs (see Figure 9.2(b)) and five containing two pairs (see Figure 9.2(c)). In total it has 105 maximal bipartite subgraphs and yields a lower bound of $105^{n/10} \approx 1.5926^n$ using multiple copies. □

9.3 Upper bound

To show the upper bound, we first prove a lemma characterising maximal k-colourable subgraphs.

[1] This graph is also found in a list of counterexamples to graph conjectures [22] made by Graffiti [21], a program generating such conjectures automatically. We could not find any information about which conjecture it disproved.

Lemma 9.1 *Let M be a maximal k-colourable subgraph of a graph $G = (V, E)$. Then the vertices of M can be split into colour classes $C_1, C_2, \ldots, C_k$ of non-increasing sizes, such that for all i, j with $0 \leq i < j \leq k$, $G[C_{i+1} \cup C_{i+2} \cup \cdots \cup C_j]$ is a maximal $(j-i)$-colourable subgraph of $G[V \setminus (C_1 \cup C_2 \cup \cdots \cup C_i)]$.*

Proof. Look at all possible k-colourings of M having the colour classes sorted in non-increasing order. Label each with a list of the sizes of the colour classes in reverse order, i.e. the smallest one first. We pick one of the lexicographically smallest labelled colourings $C_1, C_2, \ldots, C_k$ and claim that it satisfies the conditions of the lemma.

Suppose conversely that there exist i, j with $0 \leq i < j \leq k$, such that $G[C_{i+1} \cup C_{i+2} \cup \cdots \cup C_j]$ is *not* a maximal $(j-i)$-colourable subgraph of $G[V \setminus (C_1 \cup C_2 \cup \cdots \cup C_i)]$. Since it is $(j-i)$-colourable, there exists a vertex v in $G[V \setminus (C_1 \cup C_2 \cup \cdots \cup C_j)]$, such that $G[C_{i+1} \cup C_{i+2} \cup \cdots \cup C_j \cup \{v\}]$ remains $(j-i)$-colourable. Now, $v \in M$; otherwise, $M \cup \{v\}$ is k-colourable, and this contradicts the maximality of M, so $v \in C_l$ for some $l > j$. Pick a $(j-i)$-colouring of $G[C_{i+1} \cup C_{i+2} \cup \cdots \cup C_j \cup \{v\}]$ together with the colouring $C_1, \ldots, C_i, C_{j+1}, \ldots, C_{l-1}, C_l \setminus \{v\}, C_{l+1}, \ldots, C_k$ of $G[V \setminus (C_{i+1} \cup C_{i+2} \cup \cdots \cup C_j \cup \{v\})]$. They form a k-colouring of M in which the l'th colour class is smaller than in the original colouring, and the succeeding ones are of the same size; thus the list of the sizes of the colour classes in reverse order is lexicographically smaller than the label of the original colouring. Since sorting the list only makes it smaller, the label of the new colouring is lexicographically smaller than the label of the original colouring. This is a contradiction, and therefore the lemma is proved. □

Theorem 9.2 *Any graph contains at most $O(12^{n/4}) = O(1.8613^n)$ maximal bipartite subgraphs. Moreover, there is an algorithm that takes as input a graph and outputs all its maximal bipartite subgraphs in time $O(1.8613^n)$.*

Proof. Let G be an arbitrary graph and M a maximal bipartite subgraph thereof. Using Lemma 9.1 twice, first with $i = 0$ and $j = 1$ and second with $i = 1$ and $j = 2$, we can assume that the vertices of M consist of a maximal independent set I_1 in G and a maximal independent set I_2 in $G[V \setminus I_1]$ having $|I_2| \leq |I_1|$. To find all maximal bipartite subgraphs our algorithm finds all maximal independent sets in G and for each finds all no larger maximal independent sets in the remaining graph. If their union is a maximal bipartite subgraph the algorithm outputs it.[2] Let $I_k(G)$ denote the set of all maximal independent sets of size at most k in G. Then the number of maximal bipartite subgraphs of G is at most

$$\sum_{k=1}^{n} \sum_{\substack{I \in I_k(G) \\ |I|=k}} |I_k(G[V \setminus I])| \leq \sum_{k=1}^{n} |I_k(G)| \cdot \max_{\substack{I \subseteq V \\ |I|=k}} |I_k(G[V \setminus I])|.$$

[2]The union is not necessarily a maximal bipartite subgraph. In the 6-cycle, two opposite vertices form a maximal independent set, but their union with a maximal independent set in the remaining graph does not form a maximal bipartite subgraph.

Eppstein [20] shows that $|I_k(G)| \leq 3^{4k-n}4^{n-3k}$, and Moon and Moser [40] show that any graph can have at most $3^{n/3}$ maximal independent sets in total. Splitting the sum in two and using the minimum of these two bounds, we get that the sum is at most

$$\sum_{k=1}^{\lfloor \frac{n}{4} \rfloor} 3^{4k-n}4^{n-3k}3^{4k-(n-k)}4^{(n-k)-3k} + \sum_{k=\lfloor \frac{n}{4} \rfloor+1}^{n} 3^{4k-n}4^{n-3k}3^{(n-k)/3}.$$

Moving the terms not depending on k outside the sums and using the fact that the sums are geometric series we get that the whole expression is $O(12^{n/4}) = O(1.8613^n)$.

All maximal independent sets in a graph can be found in time within a polynomial factor of their number, see e.g. Tsukiyama et al. [55]. Those of size at most k can be found in time $O(3^{4k-n}4^{n-3k})$ as shown by Eppstein [20]. Our algorithm uses these two algorithms to find all maximal independent sets in the graph, and for those of size $k \leq n/4$ it finds all maximal independent sets of size at most k in the remaining graphs, and for those of size $k > n/4$ it finds *all* maximal independent sets in the remaining graphs. The algorithm runs in time $O(1.8613^n)$. □

9.4 Colouring

Following Schiermeyer [47], we design an algorithm for deciding k-colourability of a graph. Suppose that $G = (V, E)$ is k-colourable. By setting $M = V$, $i = 0$ and $j = 2$ in Lemma 9.1 we obtain that the graph can be partitioned into a maximal bipartite subgraph and a $(k-2)$-colourable subgraph; thus the algorithm checks k-colourability of a graph by finding all maximal bipartite subgraphs and checking whether any of the remaining graphs are $(k-2)$-colourable. The time complexity of checking k-colourability is proportional to the time complexity of finding all maximal bipartite subgraphs times the time complexity of checking $(k-2)$-colourability of the remaining graphs.

The time complexity of checking 4-colourability using the above algorithm is within a polynomial factor of the time complexity of finding all maximal bipartite subgraphs, since 2-colourability can be checked in polynomial time. By Theorem 9.2, this is $O(1.8613^n)$. This is not competitive as Byskov [7] has a 4-colouring algorithm running in time $O(1.7504^n)$. To improve the running time of our 4-colouring algorithm we need a smaller upper bound on the number of maximal bipartite subgraphs and a fast algorithm for finding them.

Byskov [7] also shows that 5-colourability can be checked in time $O(2.1592^n)$. This is done (correctness follows from Lemma 9.1) by first finding a maximal independent set I_1 of size at least $n/5$ in G, then finding a maximal independent set I_2 in $G[V \setminus I_1]$ of size at least $(n-|I_1|)/4$ and finally checking 3-colourability of $G[V \setminus (I_1 \cup I_2)]$ using the fastest algorithm, which is by Eppstein [19]. This can be replaced by finding a maximal bipartite subgraph of the graph of size at least $2n/5$ and checking 3-colourability of the remaining graph which yields the same running time. A calculation similar to the one in the proof of Theorem 9.2

shows that the worst terms correspond to maximal bipartite subgraphs of size $2n/5$, and that there are at most $20^{n/5} \approx 1.8206^n$ of these. This means that better 5-colouring algorithms can be obtained by improving the bound on the number of maximal bipartite subgraphs of size $2n/5$ arbitrarily for instance by showing that the total number of maximal bipartite subgraphs is at most $o(20^{n/5})$. Since the time complexity of 5-colouring is $O(2.1592^n)$, we do not need to be able to find the maximal bipartite subgraphs faster to get better time bounds.

9.5 Conclusion

We have shown that there can be at least $105^{n/10} \approx 1.5926^n$ and at most $O(12^{n/4}) = O(1.8613^n)$ maximal bipartite subgraphs of a graph, and they can be found in time within a polynomial factor of our upper bound. Maximal bipartite subgraphs can be used in k-colouring algorithms, but to be competitive better upper bounds are needed.

We think that the true bound is in fact lower than our upper bound. The worst case graphs in the proof of our upper bound are those having $4^{n/4}$ maximal independent sets of size $n/4$. The only graphs achieving this are graphs consisting of a union of disconnected K_4s (by Eppstein [20]), and these have only $6^{n/4}$ maximal bipartite subgraphs. Also, if we use the same approach as in the proof of Theorem 9.2 to prove an upper bound on the number of maximal 3-colourable subgraphs of a graph we get a bound of $O(2.2680^n)$, which is clearly too high.

To prove our upper bound we use bounds on the number of maximal independent sets in a graph. These bounds are tight at least for the values where our expression attains its maximum. We found our lower bound by testing all graphs of size $n \leq 10$ on a computer. This becomes infeasible even for slightly larger n, since the number of graphs grows extremely fast. New ideas are thus needed to prove better lower and upper bounds.

Acknowledgements

We thank our supervisors Peter Bro Miltersen and Sven Skyum for many helpful comments and insights.

Chapter 10

Graph Distinguishability Problems

This chapter contains the paper "Reductions among Graph Distinguishability Problems" [54].

Abstract

The problem DIST_k is the problem of, given a graph, determining if there exists a colouring of the graph with at most k different colours such that no nontrivial automorphism preserves the colouring. In this paper, we prove how to reduce DIST_l to DIST_k for various values of l and k by showing that if there exists a graph which can be coloured in l "different" ways using k colours we can reduce DIST_l to DIST_k. From this we conclude that DIST_1 (which is also called RIGID) can be reduced to DIST_k for any $k > 0$. Furthermore, we present different ways for constructing graphs with certain l and k values from smaller graphs and show that DIST_l can be reduced to DIST_2 for l equal to 1, 2, 4, 6, 8, 10, 12, 14, 15, 16, 18, 20, 22, 24, 27, 28, 30, 32, 36 and 40.

10.1 Introduction

A classical problem presented by Frank Rubin [43] goes:

> A blind man keeps his keys on a circular ring. There are s distinct handle shapes that he can tell apart by feel, and he can purchase any key with any desired handle shape. Assume that all keys are symmetrical so that a rotation of the key ring about an axis in its plane is undetectable from examination of a single key. How many keys can he keep on the ring and still be able to select the proper key by feel?

In this paper we will consider a generalised version of this problem called Graph Distinguishability. A graph is said to be k-distinguishable if it is possible to assign one of k different colours to each vertex of the graph in such a way that no non-trivial automorphism preserves the colour of all vertices. The above problem is the special case where $k = s$ and the graphs are cycles. The distinguishing number of a graph is the smallest number, k, for which the graph is k-distinguishable.

Definition 10.1 *Let $G = (V, E)$ be a graph and let k be a positive integer. A* k-colouring *of G is a function mapping V to $\{1, 2, \ldots, k\}$. A* legal k-colouring *c is a k-colouring which destroys all non-trivial automorphisms:*

$$\forall \phi \in \mathrm{Aut}(G) \setminus \{\mathrm{id}(V)\} : c \neq c \circ \phi.$$

The set of legal k-colourings of a graph G will be denoted $\mathrm{LC}_k(G)$.

Definition 10.2 *The problem* DIST_k *($k \geq 0$) is: Given a graph $G = (V, E)$, are there any legal k-colourings of G, i.e., is* $\mathrm{LC}_k(G) \neq \emptyset$*? $DIST_1$ is also called* RIGID.

There has been considerable interest in the computational complexity of the problem of deciding the distinguishing number of a graph. DIST_0 is trivial as it only contains the empty graph and RIGID is in coNP as a graph can be shown to be nonrigid by giving a nontrivial automorphism on the graph. For $k > 1$, DIST_k is not known to be in neither NP nor coNP. It is easy to see that DIST_k is in Σ_2^P.

In 1996, Albertson and Collins [2] decided the distinguishing number for certain classes of graphs and in 1998, Russell and Sundaram [44] proved that DIST_k lies in $\Sigma_2^P \cap \mathrm{AM}$ and that if it is coNP-hard the polynomial hierarchy collapses[1]. Recently Arvind and Nikhil [3] proved that for planar graphs the distinguising number can be determined in polynomial time.

In the rest of this introduction, we will only consider legal colourings, but will omit the word *legal*. In section 10.3, we introduce the notion of unique colourings of a graph. This captures the fact that different colourings of a graph may be indistinguishable in the sense that there exists an automorphism of the graph taking one colouring to another. The unique colourings of a graph is thus only the colourings that can be distinguished. The first result in this paper is that we can add a "handle" (a number of new vertices connected to all vertices in the original graph) to a graph without changing the number of unique colourings. Adding a handle to a graph makes the graph connected and in the construction used in the proof of our main result we use multiple copies of a graph and the handle to ensure that vertices from different copies of the graph are not mixed.

Our main result states that if there exists a graph which have l unique k-colourings, then it is possible to reduce DIST_l to DIST_k. We use this result to prove that RIGID can be reduced to DIST_k for any $k > 0$.

In section 10.5, we present two methods for constructing graphs with a certain number of unique colourings. The first constructs a graph with $l_1 \cdot l_2$ unique colourings from two graphs with respectively l_1 and l_2 unique colourings and the second constructs from a graph which has l unique colourings a graph with $\binom{l}{a}$ unique colourings, where $1 \leq a \leq l$.

[1] Note that Russell and Sundaram repeatedly state that the problem lies in $\mathrm{AM} \subset \Sigma_2^P \cap \Pi_2^P$. This statement is somewhat misleading as it is a well known open problem if AM is in Σ_2^P (see, e.g., [26]). What they *do* show is that the problem is in AM and since it is clearly in Σ_2^P, it is in $\Sigma_2^P \cap \mathrm{AM}$ and as $\mathrm{AM} \subset \Pi_2^P$ thus in $\Sigma_2^P \cap \Pi_2^P$.

As our results depend on graphs having a specific number of unique k-colourings for some value of k, we take a closer look at the case $k = 2$ in section 10.6. In particular we show that we can construct graphs with 1, 2, 4, 6, 8, 10, 12, 14, 15, 16, 18, 20, 22, 24, 27, 28, 30, 32, 36 and 40 unique 2-colourings. We furthermore conjecture that it is not possible to construct a graph with 3 unique two-colourings. We conclude this paper by presenting some open problems.

As several of the proofs in this paper are quite large even though the ideas behind are simple, we try to sketch the idea of the proofs before actually stating the proofs. On a first read one might want to skip the proofs and this should not cause any trouble.

10.2 Notation

If $f : S \to T$ is a function mapping some set S into another set T and $S' \subseteq S$, then $f(S')$ is the set $\{f(s) : s \in S'\}$. The function $f|_{S'} : S' \to T$, where $S' \subseteq S$, denotes f restricted to the domain S' (i.e., $\forall s \in S' : f|_{S'}(s) = f(s)$). In this paper, it will always be the case that if $f|_{S'}$ is used and $S = T$ then $f(S') = S'$ (if $f : S \to S$ then $f|_{S'} : S' \to S'$).

We will use $f[s \mapsto t]$ where $s \in S$ and $t \in T$ to denote a function that acts as f except that s is mapped to t. If multiple values are redefined we will use $f[s_1 \mapsto t_1, \ldots, s_k \mapsto t_k]$, or if $S' \subset S$ $f[\forall s \in S' : s \mapsto g(s)]$. If g(s) is not just an expression but a function $g : S' \to T$ we will also use the shorthand $f[g]$ to denote the latter.

If g is a function mapping some set S' into another set T' and $S \cap S' = \emptyset$, then $f \cup g : S \cup S' \to T \cup T'$ will denote the function which acts as f on S and acts as g on S':

$$\forall s \in S \cup S' : (f \cup g)(s) = \begin{cases} f(s) & s \in S, \\ g(s) & s \in S'. \end{cases}$$

By $[k]$ we will denote the set $\{1, 2, \ldots, k\}$. If S is a set, $\mathrm{id}(S) : S \to S$ will denote the identity function defined on S and Σ_S will be the set of all permutations on S. If $G = (V, E)$ is a graph, we will let $\mathrm{V}(G)$ be the vertex set V and $\deg(v)$, where $v \in V$, will denote the degree of v in G. $\mathrm{Aut}(G)$ will be the set of automorphisms of G, i.e., $\phi \in \mathrm{Aut}(G) \Leftrightarrow (\phi \in \Sigma_V \wedge ((v_1, v_2) \in E \Leftrightarrow (\phi(v_1), \phi(v_2)) \in E))$. Note that if $\phi \in \Sigma_V$ and $(v_1, v_2) \in E \Rightarrow (\phi(v_1), \phi(v_2)) \in E$ then $\phi \in \mathrm{Aut}(G)$. $\mathrm{Aut}(G)$ will be called trivial if $\mathrm{Aut}(G) = \{\mathrm{id}(\mathrm{V}(G))\}$.

For simplicity, we will assume that graphs do not contain self loops and are not oriented, but all results are valid even for graphs with orientation and/or self loops.

10.3 Unique colourings

In this section we present formal definitions of unique colourings and handles and prove that adding handles does not change the number of unique colourings.

Definition 10.3 *For any graph G where $\mathrm{Aut}(G)$ is nontrivial some of the legal k-colourings of G are indistinguishable. We can thus define an equivalence relation on $\mathrm{LC}_k(G)$:*

$$c_1 \sim c_2 \quad \Leftrightarrow \quad \exists \phi \in \mathrm{Aut}(G) : c_1 = c_2 \circ \phi.$$

In order to look only at distinguishable colourings we define the set $\mathrm{ULC}_k(G)$ (unique legal k-colourings) *to be the equivalence classes of $\mathrm{LC}_k(G)$:*

$$\mathrm{ULC}_k(G) = \{[c]_G : c \in \mathrm{LC}_k(G)\}$$

where $[c]_G = \{c \circ \phi : \phi \in \mathrm{Aut}(G)\}$. In most cases G will be obvious from the context and we will just write $[c]$ (this should not be confused with the set $[k]$, where k is an integer).

If $\phi_1, \phi_2 \in \mathrm{Aut}(G)$ and $c \circ \phi_1 = c \circ \phi_2$, we get that $c = c \circ \phi_2 \circ \phi_1^{-1}$ and as $c \in \mathrm{LC}_k(G)$ we get that $\phi_2 \circ \phi_1^{-1} = \mathrm{id}(\mathrm{V}(G))$ which implies $\phi_1 = \phi_2$ and thus $\big|[c]\big| = |\mathrm{Aut}(G)|$. From this we get that $|\mathrm{ULC}_k(G)| = \frac{|\mathrm{LC}_k(G)|}{|\mathrm{Aut}(G)|}$.

Definition 10.4 *Let $G = (V, E)$ be an arbitrary graph and k a positive integer. Define the graph $\mathrm{handle}(G, k) = (V', E')$ to be G with k extra vertices, each connected to all the original vertices:*

$$\begin{aligned} V^h &= \big\{h_i : i \in [k]\big\}, \\ V' &= V \cup V^h, \\ E' &= E \cup V \times V^h. \end{aligned}$$

Note that G can easily be found from $\mathrm{handle}(G, k)$ if k is known.

Lemma 10.1 *The number of legal k-colourings of $\mathrm{handle}(\mathrm{handle}(G, k), k)$ is zero for any graph G. On the other hand, if $G \neq \mathrm{handle}(G', k)$ for all graphs G', the number of unique legal k-colourings of G and $\mathrm{handle}(G, k)$ is the same:*

$$|\mathrm{ULC}_k(G)| = |\mathrm{ULC}_k(\mathrm{handle}(G, k))|.$$

Intuitively, the lemma is correct since we can colour the handle and the rest of the graph separately. As the handle can only be legally coloured in one unique way (all k vertices have to have different colours), the number of unique legal colourings is the same for the handle graph as for the original graph. If a graph has two handles of size k they can not be distinguished with k colours and thus there are no legal k-colourings of the graph.

Proof. Let $G' = (V', E') = \mathrm{handle}(\mathrm{handle}(G, k), k)$. By construction of G' there exists V_1^h and V_2^h with $|V_i^h| = k$ such that

$$\begin{aligned} V' &= V \cup V_1^h \cup V_2^h, \\ E' &= E \cup V \times V_1^h \cup V \times V_2^h \cup V_1^h \times V_2^h \end{aligned}$$

where $G = (V, E)$. In a legal k-colouring of G' the vertices of V_i^h must have different colours as two vertices, $v, w \in V_i^h$, with the same colour would be

indistinguishable. Then there is no way to distinguish V_1^h from V_2^h. Each uses all colours exactly once, so there exists an automorphism which maps V_1^h to V_2^h and vice versa and preserves the colouring. This contradicts the definition of legal colouring and proves the first part of the lemma.

In the following let $G = (V, E)$ be such that $G \neq \text{handle}(G', k)$ for all graphs G'. Let H, W and F be such that $H = (W, F) = \text{handle}(G, k) = (V \cup V^h, E \cup E^h)$. First we prove a property of the automorphisms of H:

$$\forall \phi \in \text{Aut}(H) : \phi(V) = V \wedge \phi(V^h) = V^h. \tag{10.1}$$

If $\phi(V) = V$ then also $\phi(V^h) = V^h$ and it thus suffices to prove the former. Assume there exists a $\phi \in \text{Aut}(H)$ and a $v \in V$ such that $\phi(v) \in V^h$. Let W^h be the vertices which ϕ maps to V^h. The vertices in V^h are not connected to each other but are connected to all other vertices, and as ϕ is an automorphism this is also true for all vertices in W^h. From this we can conclude $W^h \subset V$. By setting $V' = V \setminus W^h$ we get that

$$E = \big(E \cap (V' \times V')\big) \cup (V' \times W^h).$$

This shows that $G = (V' \cup W^h, E) = \text{handle}\big((V', E \cap (V' \times V')), k\big)$ which contradicts the assumption.

From this property of $\text{Aut}(H)$ we get that if $\phi \in \text{Aut}(H)$ then $\phi|_V \in \text{Aut}(G)$. On the other hand we get from the structure of H that if $\phi \in \text{Aut}(G)$ and $\pi \in \Sigma_{V^h}$ then $(\phi \cup \pi) \in \text{Aut}(H)$. From the last part we get that if $c \in \text{LC}_k(H)$ then $c|_V \in \text{LC}_k(G)$.

Next we will prove

$$\forall c_1, c_2 \in \text{LC}_k(H) : [c_1]_H = [c_2]_H \quad \Leftrightarrow \quad [c_1|_V]_G = [c_2|_V]_G. \tag{10.2}$$

Let $c_1, c_2 \in \text{LC}_k(H)$ with $[c_1]_H = [c_2]_H$. By definition, there exists $\phi \in \text{Aut}(H)$ such that $c_1 = c_2 \circ \phi$ and thus $c_1|_V = c_2|_V \circ \phi|_V$. As $\phi|_V \in \text{Aut}(G)$ we get that $[c_1|_V]_G = [c_2|_V]_G$. On the other hand, if $c_1, c_2 \in \text{LC}_k(H)$ are such that $[c_1|_V]_G = [c_2|_V]_G$ then by definition there exists $\phi \in \text{Aut}(G)$ such that $c_1|_V = c_2|_V \circ \phi$. As the k vertices in V^h are indistinguishable, c_1 and c_2 both assign them different colours. We can thus define a permutation $\pi = (c_2|_{V^h})^{-1} \circ c_1|_{V^h} \in \Sigma_{V^h}$, and get that $c_1 = c_2 \circ (\phi \cup \pi)$, which concludes the proof of (10.2).

From (10.2) we get directly that $|\text{ULC}_k(G)| \geq |\text{ULC}_k(H)|$. Any k-colouring of G can be extended to a k-colouring of H by assigning the handle vertices k different colours (i.e., $\forall c \in \text{LC}_k(G)\ \exists c' \in \text{LC}_k(H) : c'|_V = c$) and from (10.2) we can thus also conclude $|\text{ULC}_k(G)| \leq |\text{ULC}_k(H)|$. Thus the number of unique legal k-colourings are the same for G and H. □

10.4 Reductions

In this section we will state and prove our primary result, but first we present one concrete result which can be obtained from the general result:

Theorem 10.1 *RIGID is not harder than any other $DIST_k$, $k \geq 1$:*

$$RIGID \prec DIST_k.$$

We will postpone the proof, to after we have proved our primary result:

Theorem 10.2 *If there exists a graph $H = (W, F)$ such that $l = |\mathrm{ULC}_k(H)|$ then $DIST_l$ can be reduced to $DIST_k$:*

$$DIST_l \prec DIST_k.$$

The idea in the reduction is simple: By replacing each vertex with a copy of H, we can simulate l colours with k colours. We use the handle construction to ensure that any automorphism of the graph maps each copy of H to a copy of H, and thus does not mix the vertices of different copies of H.

Proof. By Lemma 10.1 we can assume that $H = \mathrm{handle}(H', k)$, for some graph H'. If $H' = \mathrm{handle}(H'', k)$ for some graph H'', the reduction is trivial, because it is a reduction from DIST_0, which only contains the empty graph: If the input is the empty graph output the empty graph and otherwise output $(V, \emptyset)$ where $|V| = k+1$. This graph does not have a legal k-colouring, so it will be rejected. In the following we will thus assume that $H' \neq \mathrm{handle}(H'', k)$ for all graphs H''. Let $W^h = \{h_1, \ldots, h_k\} = W \setminus \mathrm{V}(H')$ be the handle vertices of H, and $W^{nh} = W \setminus W^h$ be the non-handle vertices of H.

Given a graph $G = (V, E)$ let $G' = (V', E')$ be the graph consisting of one copy of H for each vertex of G, and furthermore have all edges between the non-handle vertices of two copies of H if there is an edge between the corresponding vertices of G:

$$\begin{aligned}
V_v &= \{v_{v,w} : w \in W\} && v \in V,\\
V' &= \bigcup_{v \in V} V_v = \{v_{v,w} : v \in V, w \in W\}, &&\\
V_v^{nh} &= \left\{v_{v,w} : w \in W^{nh}\right\} && v \in V,\\
E_v &= \{v_{v,w_1} v_{v,w_2} : (w_1, w_2) \in F\} && v \in V,\\
E_{(v_1,v_2)} &= V_{v_1}^{nh} \times V_{v_2}^{nh} && (v_1, v_2) \in E,\\
E' &= \bigcup_{v \in V} E_v \cup \bigcup_{(v_1,v_2) \in E} E_{(v_1,v_2)}. &&
\end{aligned}$$

For simplicity we also define the handle-vertices $V_v^h = V_v \setminus V_v^{nh}$ for all $v \in V$. In Figure 10.1 is an illustration of the construction of G'. The remainder of this proof will be used to show that $G \in \mathrm{DIST}_l$ if and only if $G' \in \mathrm{DIST}_k$.

In the proof, we will use v to denote variables in V, v' to denote variables in V' and w to denote variables in W. If multiple variables from a set is needed we will use subscripts. Similarly, ϕ will denote an element from $\mathrm{Aut}(G)$, ϕ' an

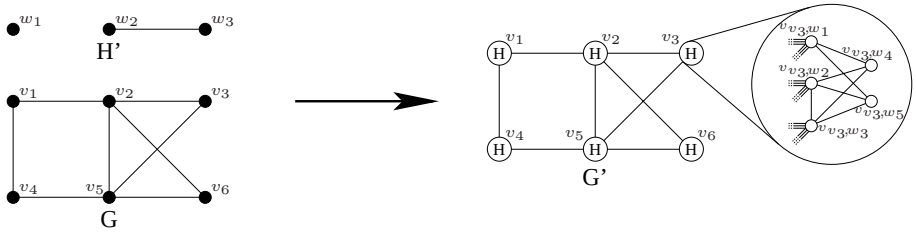


Figure 10.1. An example of the reduction in Theorem (10.2). Note that $H = \text{handle}(H', 2)$.

element from $\text{Aut}(G')$ and τ an element from $\text{Aut}(H)$. Finally, c will denote an l-colouring of G and c' and d respectively will denote k-colourings of G' and H. We will let $d_1, \ldots, d_l$ be such that $\text{ULC}_k(H) = \{[d_1], \ldots, [d_l]\}$. The following functions will be used to determine which vertices in V and W a vertex from V' origins from:

$$\begin{aligned} \text{G}(v_{v,w}) &= v, \\ \text{H}(v_{v,w}) &= w. \end{aligned}$$

The following properties follows directly from the construction of G' and the definitions of $\text{G}(v')$ and $\text{H}(v')$.

$$\forall v' \in V' : v' \in V_{\text{G}(v')}, \tag{10.3}$$

$$\forall v_1, v_2 \in V : V_{v_1} \cap V_{v_2} \neq \emptyset \Rightarrow v_1 = v_2, \tag{10.4}$$

$$\deg(v') = \begin{cases} \left|W^{nh}\right| \quad (= \deg(\text{H}(v'))) & \text{if } \text{H}(v') \in W^h, \\ \deg(\text{H}(v')) + \deg(\text{G}(v')) \cdot |W^{nh}| & \text{if } \text{H}(v') \in W^{nh}. \end{cases} \tag{10.5}$$

The idea in the first part of the proof is to show that every automorphism of G' keeps the copies of H together (property (10.6) below). The reason this is true is that the handles are kept together and they ensure that the rest of each H is kept together. We can thus look at the automorphisms as permutations of the vertices of G and we prove that these permutations are in fact automorphisms of G (property (10.7)). On the other hand if we look at a copy of H then an automorphism of G' maps it into another copy of H and this can be seen as a permutation on H. We prove that this permutation is an automorphism on H (property (10.8)). Finally we use this to prove that if we have a legal l-colouring of G we can colour G' legally by using d_i to colour the copies of H which correspond to vertices of G with colour i.

$$\forall \phi' \in \text{Aut}(G')\ \forall v_1', v_2' \in V' : \text{G}(v_1') = \text{G}(v_2') \Leftrightarrow \text{G}(\phi'(v_1')) = \text{G}(\phi'(v_2')), \tag{10.6}$$

$$\forall \phi' \in \text{Aut}(G')\ \exists \phi \in \text{Aut}(G)\ \forall v' \in V' : \phi(\text{G}(v')) = \text{G}(\phi'(v')), \tag{10.7}$$

$$\forall \phi' \in \text{Aut}(G')\ \forall v \in V\ \exists \tau \in \text{Aut}(H)\ \forall v' \in V_v : \tau(\text{H}(v')) = \text{H}(\phi'(v')). \tag{10.8}$$

In order to show (10.6) we will show

$$\forall \phi' \in \mathrm{Aut}(G')\ \forall v' \in V' : \mathrm{H}(v') \in W^h \Leftrightarrow \mathrm{H}(\phi'(v')) \in W^h, \tag{10.6a}$$

$$\forall \phi' \in \mathrm{Aut}(G')\ \forall v' \in V' : \phi'(V_{\mathrm{G}(v')}) = V_{\mathrm{G}(\phi'(v'))}. \tag{10.6b}$$

For any automorphism ϕ' of G' and $v' \in V'$ with $\mathrm{H}(v') \in W^h$ we have that $\deg(v') = \deg(\phi'(v'))$ and from (10.5) that $\deg(v') = |W^{nh}|$. If $\mathrm{H}(\phi'(v')) \in W^{nh}$ then by (10.5) we get that $\deg(\phi'(v')) = \deg(\mathrm{H}(\phi'(v'))) + \deg(\mathrm{G}(\phi'(v'))) \cdot |W^{nh}|$ and since $\deg(w) > 0$ for all vertices $w \in W$ then $\deg(\mathrm{G}(\phi'(v'))) = 0$. Thus $V_{\mathrm{G}(\phi'(v'))}$ is an isolated copy of H in G' and as H is connected, $V_{\mathrm{G}(v')}$ must also be an isolated copy of H. Then $\phi'(V_{\mathrm{G}(v')}) = V_{\mathrm{G}(\phi'(v'))}$ and thus $\tau = [\forall w \in W : w \mapsto \mathrm{H}(\phi'(v_{\mathrm{G}(v'),w}))]$ is a permutation of H. If $(w_1, w_2) \in F$ then we have from the construction of G' that $(v_{\mathrm{G}(v'),w_1}, v_{\mathrm{G}(v'),w_2}) \in E'$ and thus $(\phi'(v_{\mathrm{G}(v'),w_1}), \phi'(v_{\mathrm{G}(v'),w_2})) \in E'$. Using that $\phi'(V_{\mathrm{G}(v')}) = V_{\mathrm{G}(\phi'(v'))}$ we get that the edge is within a copy of H and from the construction of G' we then get that $(\tau(w_1), \tau(w_2)) \in F$, which shows that $\tau \in \mathrm{Aut}(H)$. But $\tau(\mathrm{H}(v')) = \mathrm{H}(\phi'(v_{\mathrm{G}(v'),\mathrm{H}(v')})) = \mathrm{H}(\phi'(v'))$ which contradicts (10.1), and thus proves $\mathrm{H}(\phi'(v')) \in W^h$ and we have proved the implication in (10.6a) from left to right. From right to left then follows from this and the fact that $(\phi')^{-1}$ is also an automorphism of G'.

Let $\phi' \in \mathrm{Aut}(G')$ and $v'_1 \in V_v^h$ for some $v \in V$, then the neighbours of v'_1 are V_v^{nh} and thus the vertices $\phi'(v'_1)$ have edges to are $\phi'(V_v^{nh})$. By (10.3) and (10.6a) $\phi'(v'_1) \in V^h_{\mathrm{G}(\phi'(v'_1))}$ and the vertices it has edges to are $V^{nh}_{\mathrm{G}(\phi'(v'_1))}$ so $\phi'(V^{nh}_{\mathrm{G}(v'_1)}) = V^{nh}_{\mathrm{G}(\phi'(v'_1))}$. On the other hand the only vertices that only have edges to $V^{nh}_{\mathrm{G}(v'_1)}$ are $V^h_{\mathrm{G}(v'_1)}$ and thus $\phi'(V^h_{\mathrm{G}(v'_1)})$ must be the only vertices that only have edges to $\phi'(V^{nh}_{\mathrm{G}(v'_1)}) = V^{nh}_{\mathrm{G}(\phi'(v'_1))}$. But the vertices that only have edges to $V^{nh}_{\mathrm{G}(\phi'(v'_1))}$ are $V^h_{\mathrm{G}(\phi'(v'_1))}$ so $\phi'(V^h_{\mathrm{G}(v'_1)}) = V^h_{\mathrm{G}(\phi'(v'_1))}$. Putting these two equalities together we get that $\phi'(V_{\mathrm{G}(v'_1)}) = V_{\mathrm{G}(\phi'(v'_1))}$. This shows (10.6b) when $v' \in V_v^h$ for some $v \in V$. Now, let $v'_2 \in V_v^{nh}$. As $v'_2 \in V_{\mathrm{G}(v'_2)}$ and $G(v'_1) = G(v'_2)$ we have that $\phi'(v'_2) \in \phi'(V_{\mathrm{G}(v'_2)}) = \phi'(V_{\mathrm{G}(v'_1)}) = V_{\mathrm{G}(\phi'(v'_1))}$ and from (10.3) $\phi'(v'_2) \in V_{\mathrm{G}(\phi'(v'_2))}$. By (10.4) we have that $\mathrm{G}(\phi'(v'_1)) = \mathrm{G}(\phi'(v'_2))$. This shows that $\phi'(V_{\mathrm{G}(v'_2)}) = V_{\mathrm{G}(\phi'(v'_2))}$, which concludes the proof of (10.6b).

Let $\phi' \in \mathrm{Aut}(G')$ and $v'_1, v'_2 \in V'$ with $\mathrm{G}(v'_1) = \mathrm{G}(v'_2)$. By (10.6b) we have that $V_{\mathrm{G}(\phi'(v'_1))} = \phi'(V_{\mathrm{G}(v'_1)}) = \phi'(V_{\mathrm{G}(v'_2)}) = V_{\mathrm{G}(\phi'(v'_2))}$ and from (10.4) we get that $\mathrm{G}(\phi'(v'_1)) = \mathrm{G}(\phi'(v'_2))$, which shows (10.6) from left to right. As in the proof of (10.6a) from right to left follows from the fact that $(\phi')^{-1}$ is also an automorphism of G' and we have proved (10.6).

To prove (10.7) we first show that for any $\phi' \in \mathrm{Aut}(G')$ there exists a function ϕ from V to V such that

$$\begin{array}{ccc} V' & \xrightarrow{\phi'} & V' \\ {\scriptstyle G}\downarrow & & \downarrow{\scriptstyle G} \\ V & \xrightarrow[\phi]{} & V \end{array}$$

commutes, i.e. such that $\forall v' \in V' : \phi(\mathrm{G}(v')) = \mathrm{G}(\phi'(v'))$. If no such function exist it must be the case that $\exists v'_1, v'_2 \in V' : \mathrm{G}(v'_1) = \mathrm{G}(v'_2) \wedge \mathrm{G}(\phi'(v'_1)) \neq \mathrm{G}(\phi'(v'_2))$, but (10.6) states that this can not be the case.

Let ϕ be a function making the diagram commute. As $\phi' \in \Sigma_{V'}$ and G is surjective we get that $\phi \in \Sigma_V$. To prove that $\phi \in \mathrm{Aut}(G)$ it thus suffices to prove that $\forall (v_1, v_2) \in E : (\phi(v_1), \phi(v_2)) \in E$. Let $(v_1, v_2) \in E$. By construction of G' we have that $V^{nh}_{v_1} \times V^{nh}_{v_2} \subseteq E'$ and as $\phi' \in \mathrm{Aut}(G')$ we get that $\phi'(V^{nh}_{v_1}) \times \phi'(V^{nh}_{v_2}) \subseteq E'$. Let $v'_1 \in V^{nh}_{v_1}$ and $v'_2 \in V^{nh}_{v_2}$. As there are no self loops in G we have that $v_1 \neq v_2$ and thus $\mathrm{G}(v'_1) \neq \mathrm{G}(v'_2)$ and by (10.6) $\mathrm{G}(\phi'(v'_1)) \neq \mathrm{G}(\phi'(v'_2))$. As $(\phi'(v'_1), \phi'(v'_2)) \in E'$ and $\mathrm{G}(\phi'(v'_1)) \neq \mathrm{G}(\phi'(v'_2))$ we get from the construction of G' that $(\mathrm{G}(\phi'(v'_1)), \mathrm{G}(\phi'(v'_2))) \in E$, and from the definition of ϕ we can conclude that $(\phi(\mathrm{G}(v'_1)), \phi(\mathrm{G}(v'_2))) \in E$ and thus $(\phi(v_1), \phi(v_2)) \in E$, which finishes the proof of (10.7).

The proof of (10.8) is structurally very similar to the proof of (10.7). Let $\phi' \in \mathrm{Aut}(G')$ and $v \in V$, and choose $\phi \in \mathrm{Aut}(G)$ such that it satisfies (10.7). As $H|_{V_v}$ is a bijection there exists a function τ from W to W such that

$$\begin{array}{ccc} V_v & \xrightarrow{\phi'} & V_{\phi(v)} \\ {\scriptstyle H}\downarrow & & \downarrow{\scriptstyle H} \\ W & \xrightarrow[\tau]{} & W \end{array}$$

commutes, i.e. such that $\forall v' \in V_v : \tau(\mathrm{H}(v')) = \mathrm{H}(\phi'(v'))$. As $(H|_{V_v})^{-1}$, $\phi'|_{V_v} : V_v \to V_{\phi(v)}$ and $H|_{V_{\phi(v)}}$ are all bijections then τ is a bijection, i.e., $\tau \in \Sigma_W$.

Let $(w_1, w_2) \in F$. Then $(v_{v,w_1}, v_{v,w_2}) \in E_v$ and by construction all edges between vertices in $V_{\phi(v)}$ are in $E_{\phi(v)}$ we thus get that $(\phi'(v_{v,w_1}), \phi'(v_{v,w_2})) \in E_{\phi(v)}$. From the definition of $E_{\phi(v)}$ we get that $(\mathrm{H}(\phi'(v_{v,w_1})), \mathrm{H}(\phi'(v_{v,w_2}))) \in F$ and by definition of τ: $(\mathrm{H}(\phi'(v_{v,w_1})), \mathrm{H}(\phi'(v_{v,w_2}))) = (\tau(\mathrm{H}(v_{v,w_1})), \tau(\mathrm{H}(v_{v,w_2}))) = (\tau(w_1), \tau(w_2))$. This shows that $(w_1, w_2) \in F \Rightarrow (\tau(w_1), \tau(w_2)) \in F$, which proves that $\tau \in \mathrm{Aut}(H)$ and thus completes the proof of (10.8).

We now prove the first part of the theorem

$$G \in \mathrm{DIST}_l \Rightarrow G' \in \mathrm{DIST}_k. \tag{10.9}$$

Let $c \in \mathrm{LC}_l(G)$ and define a colouring c' of G':

$$c'(v') = d_{c(\mathrm{G}(v'))}(\mathrm{H}(v')) \qquad\qquad v' \in V'. \tag{10.10}$$

Remember that $d_1, \ldots, d_l$ is chosen such that $\mathrm{ULC}_k(H) = \{[d_1], \ldots, [d_l]\}$, and thus c' is a k-colouring of G'. If we assume that c' is not a legal k-colouring of G' there exists a nontrivial automorphism ϕ' of G' such that $c' = c' \circ \phi'$. By (10.7) we know that there exists an automorphism ϕ of G such that $\forall v' \in V' : \phi(\mathrm{G}(v')) = \mathrm{G}(\phi'(v'))$. Let ϕ be such an automorphism. If $\phi = \mathrm{id}(V)$ we have that $\forall v \in V : \phi'(V_v) = V_v$ and there must be a $v \in V$ such that $\phi'|_{V_v} \neq \mathrm{id}(V_v)$. If $\phi = \mathrm{id}(V)$ take such a v and otherwise let v be such that $c(v) \neq c(\phi(v))$. Let

τ be as in (10.8), and note that if $\phi = \mathrm{id}(V)$ then by the choice of v we know that $\tau \neq \mathrm{id}(W)$. We have that

$$\begin{aligned}\forall w \in W : d_{c(v)}(w) &= c'(v_{v,w}) = c'(\phi'(v_{v,w})) = d_{c(\mathrm{G}(\phi'(v_{v,w})))}(\mathrm{H}(\phi'(v_{v,w}))) \\ &= d_{c(\phi(\mathrm{G}(v_{v,w})))}(\tau(\mathrm{H}(v_{v,w}))) = d_{c(\phi(v))}(\tau(w)) \\ &= (d_{c(\phi(v))} \circ \tau)(w).\end{aligned}$$

If $\phi = \mathrm{id}(V)$ this contradicts that $d_{c(v)} \in \mathrm{LC}_k(H)$. On the other hand, if $\phi \neq \mathrm{id}(V)$ then $c(v) \neq c(\phi(v))$ and by definition of $d_1, \ldots, d_l$, we have that $d_{c(v)}$ and $d_{c(\phi(v))}$ belongs to different equivalence classes, and this is also a contradiction. This completes the proof of (10.9).

The idea in the second part of the proof is that we first show that an automorphism on H can be taken to any of the copies of H in G' and is then an automorphism on G' (property (10.11)). We then use this to show that for a legal colouring of G' each copy of H is also coloured legally (property (10.12)). If we have a legal colouring of G' we then have that for each vertex of G, the corresponding copy of H is coloured with a legal colouring from $[d_i]$ for some i. We end the proof by showing that if we colour each vertex of G with the corresponding i we get a legal colouring of G.

As in the first part we will start the second part by proving some properties of G':

$$\forall \tau \in \mathrm{Aut}(H)\ \forall v \in V : \mathrm{id}(V')[\forall w \in W : v_{v,w} \mapsto v_{v,\tau(w)}] \in \mathrm{Aut}(G'), \quad (10.11)$$

$$\forall c' \in \mathrm{LC}_k(G')\ \forall v \in V\ \exists d \in \mathrm{LC}_k(H)\ \forall w \in W : d(w) = c'(v_{v,w}). \quad (10.12)$$

Let $\phi = \mathrm{id}(V')[\forall w \in W : v_{v,w} \mapsto v_{v,\tau(w)}]$ for some $\tau \in \mathrm{Aut}(H)$ and $v \in V$. As $\tau \in \Sigma_W$ we get that $\phi \in \Sigma_{V'}$. Every edge not containing a vertex from V_v is of course preserved by ϕ. Edges between vertices from V_v are also preserved as (V_v, E_v) are a copy of H and $\tau \in \mathrm{Aut}(H)$. Edges with one vertex from V_v must be an edge between the non-handle vertices of two copies of H. Then all edges between the non-handle vertices of the two copies are present in E' and by (10.1) τ maps V_v^{nh} to itself, so ϕ also preserves such edges. This completes the proof of (10.11).

Let $c' \in \mathrm{LC}_k(G')$ and $v \in V$. Define a function $d : W \to [k]$:

$$d(w) = c'(v_{v,w}) \qquad w \in W.$$

If d is not a legal k-colouring, i.e. $d \notin \mathrm{LC}_k(H)$, there exists a nontrivial automorphism $\tau \in \mathrm{Aut}(H) \setminus \{id(W)\}$ such that $d = d \circ \tau$. By (10.11) we get that $\phi = \mathrm{id}(V')[\forall w \in W : v_{v,w} \mapsto v_{v,\tau(w)}]$ is a (nontrivial) automorphism of G', but for all $v_{v_1,w} \in V'$ we get that :

$$c'(\phi(v_{v_1,w})) = \begin{cases} c'(v_{v_1,w}) & \text{if } v_1 \neq v, \\ c'(v_{v_1,\tau(w)}) = d(\tau(w)) = d(w) = c'(v_{v_1,w}) & \text{if } v_1 = v. \end{cases}$$

This shows that $c' = c' \circ \phi'$ which contradicts $c' \in \mathrm{LC}_k(G')$, so $d \in \mathrm{LC}_k(H)$ and (10.12) is proved.

For all legal k-colourings c' of G' and $v \in V$ we can thus define $d(c', v)$ to be the legal k-colouring of H such that $(d(c', v))(w) = c'(v_{v,w})$ for all $w \in W$. Furthermore, we can define a l-colouring of G, $c(c')$, such that

$$(c(c'))(v) = i \qquad \Leftrightarrow \qquad d(c', v) \in [d_i]. \tag{10.13}$$

This is well-defined as $d(c', v)$ is in exactly one of the unique colour classes of H. Finally, we will for all legal k-colourings c' of G' and all $v \in V$ define $\tau(c', v)$ such that

$$d_{(c(c'))(v)} = d(c', v) \circ \tau(c', v).$$

We will let $\tau^{-1}(c', v)$ be the inverse element of $\tau(c', v)$. Intuitively, we have for all legal k-colourings c' of G' and all $v \in V$ that c' restricted to V_v corresponds to $d(c', v)$, $c(c')$ is the colouring of G where v has colour i if $d(c', v)$ is in the i'th equivalence class of $\mathrm{ULC}_l(H)$, and $\tau(c', v)$ is the automorphism of H that takes $d(c', v)$ to the representative of the i'th equivalence class.

Next, we want to prove the existence of a certain automorphism of G'. The definition of the automorphism is somewhat awkward but it is the automorphism that will make the proof go through.

$$\forall c' \in \mathrm{LC}_k(G')\ \forall \phi \in \mathrm{Aut}(G)\ \exists \phi' \in \mathrm{Aut}(G')\ \forall v_{v,w} \in V' : \\ \phi'(v_{v,w}) = v_{\phi(v),(\tau(c',\phi(v))\circ\tau^{-1}(c',v))(w)}. \tag{10.14}$$

Let $c' \in \mathrm{LC}_k(G')$ and $\phi \in \mathrm{Aut}(G)$. For readability we will let τ_v be a shorthand for $\tau(c', \phi(v)) \circ \tau^{-1}(c', v)$. Define $\phi' : V' \to V'$ by:

$$\forall v_{v,w} \in V' : \phi'(v_{v,w}) = v_{\phi(v),\tau_v(w)}$$

As $\phi \in \Sigma_V$ and $\tau_v \in \Sigma_W$ for all $v \in V$ it follows that $\phi' \in \Sigma_{V'}$. Let $(v_{v_1,w_1}, v_{v_2,w_2}) \in E'$. If $v_1 \neq v_2$ then $(v_1, v_2) \in E$ and $(v_{v_1,w_1}, v_{v_2,w_2}) \in V^{nh}_{v_1} \times V^{nh}_{v_2}$. As ϕ is an automorphism of G, we have that $(\phi(v_1), \phi(v_2)) \in E$ and from (10.1) we get that $\forall \tau \in \mathrm{Aut}(H) : \tau(w_1), \tau(w_2) \in W^{nh}$, and thus $(\phi'(v_{v_1,w_1}), \phi'(v_{v_2,w_2})) = (v_{\phi(v_1),\tau_{v_1}(w_1)}, v_{\phi(v_2),(\tau_{v_2}(w_2)}) \in V^{nh}_{\phi(v_1)} \times V^{nh}_{\phi(v_2)} \subseteq E'$.

If, on the other hand, $v_1 = v_2$ we get from the construction of G' that $(w_1, w_2) \in F$. As $\tau_{v_1} \in \mathrm{Aut}(H)$ we get that $(\tau_{v_1}(w_1), \tau_{v_1}(w_2)) \in F$, and thus $(\phi'(v_{v_1,w_1}), \phi'(v_{v_2,w_2})) = \left(v_{\phi(v_1),\tau_{v_1}(w_1)}, v_{\phi(v_1),\tau_{v_1}(w_2)}\right) \in E'$. This completes the proof of (10.14).

We are now ready to prove the second part of the theorem:

$$G' \in \mathrm{DIST}_k \Rightarrow G \in \mathrm{DIST}_l. \tag{10.15}$$

Let $c' \in \mathrm{LC}_k(G')$, and let $c = c(c')$ be as defined in (10.13). If c is not a legal l-colouring of G, there must exist a nontrivial automorphism ϕ of G such that $c = c \circ \phi$. If this is the case let ϕ' be as in (10.14). We then get

$$\begin{aligned}
\forall v_{v,w} \in V' : c'(v_{v,w}) &= (d(c', v))(w) \\
&= \big(d_{c(v)} \circ \tau^{-1}(c', v)\big)(w) \\
&= \big(d_{c(\phi(v))} \circ \tau^{-1}(c', v)\big)(w) \\
&= \big(d(c', \phi(v)) \circ \tau(c', \phi(v)) \circ \tau^{-1}(c', v)\big)(w) \\
&= \big(d(c', \phi(v))\big)\Big(\big(\tau(c', \phi(v)) \circ \tau^{-1}(c', v)\big)(w)\Big) \\
&= c'(v_{\phi(v),(\tau(c',\phi(v))\circ\tau^{-1}(c',v))(w)}) \\
&= c'(\phi'(v_{v,w})).
\end{aligned}$$

So $c' = c' \circ \phi'$, which contradicts that $c' \in \mathrm{LC}_k(G')$, so c must be a legal l-colouring of G and the proof of the theorem is complete. □

With this result the proof of Theorem 10.1 is now easy.

ProofProof of Theorem 10.1. Consider the full graph on k vertices, $K_k = (V, E)$. In a legal k-colouring c of K_k, c must use all k colours, as if $v_1 \in V$ and $v_2 \in V$ are assigned the same colour, then $c = c \circ \mathrm{id}(V)[v_1 \mapsto v_2, v_2 \mapsto v_1]$, and $\mathrm{id}(V)[v_1 \mapsto v_2, v_2 \mapsto v_1]$ is a nontrivial automorphism of K_k as $\mathrm{Aut}(K_k) = \Sigma_V$. Thus c must use every colour exactly once. But as $\mathrm{Aut}(K_k) = \Sigma_V$ all the legal k-colourings of K_k are in the same equivalence class, and $|\mathrm{ULC}_k(K_k)| = 1$. Theorem 10.2 then states that $\mathrm{DIST}_1 \prec \mathrm{DIST}_k$. □

10.5 Constructing graphs

In this section we present some constructions to make graphs with certain numbers of unique colourings from other graphs.

Theorem 10.3 *If $|\mathrm{ULC}_k(G_1)| = l_1$ and $|\mathrm{ULC}_k(G_2)| = l_2$ for two graphs G_1 and G_2, then there exists a graph G' with $|\mathrm{ULC}_k(G')| = l_1 \cdot l_2$.*

The idea in the construction in this proof is to make a graph that consists of the two graphs without any edges between. We can then colour each graph independently.

Proof. We can assume that $l_1, l_2 > 0$ as the statement otherwise is trivial. If we have two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with $|\mathrm{ULC}_k(G_1)| = l_1$ and $|\mathrm{ULC}_k(G_2)| = l_2$, we can, by Lemma (10.1), assume that $G_1 = \mathrm{handle}(G_1', l_1)$ for some graph G_1' and $G_2 \neq \mathrm{handle}(G_2', l_2)$ for all graphs G_2', and furthermore that $|V(G_1)| > |V(G_2)|$ (if $|V(G_1)| \leq |V(G_2)|$ then $\mathrm{handle}(G_2, l_2)$ and G_1' would satisfy the conditions). We want to prove that the graph $G' = (V_1 \cup V_2, E_1 \cup E_2)$ has $|\mathrm{ULC}_k(G')| = l_1 \cdot l_2$.

As G_1 consists of just one connected component and $|V_1| > |V|/2$ all automorphisms ϕ' of G' must satisfy that $\phi'(V_1) = V_1$ and thus $\phi'|_{V_1} \in \mathrm{Aut}(G_1)$ and $\phi'|_{V_2} \in \mathrm{Aut}(G_2)$. From this follows that if $c_1 \in \mathrm{LC}_k(G_1)$ and $c_2 \in \mathrm{LC}_k(G_2)$ then $(c_1 \cup c_2) \in \mathrm{LC}_k(G')$. On the other hand, if $\phi_1 \in \mathrm{Aut}(G_1)$ and $\phi_2 \in \mathrm{Aut}(G_2)$ then $(\phi_1 \cup \phi_2) \in \mathrm{Aut}(G')$ and from this follows that if

$c \in \mathrm{LC}_k(G')$ then $c|_{V_1} \in \mathrm{LC}_k(G_1)$ and $c|_{V_2} \in \mathrm{LC}_k(G_2)$. This shows that $|\mathrm{Aut}(G')| = |\mathrm{Aut}(G_1)| \cdot |\mathrm{Aut}(G_2)|$ and $|\mathrm{LC}_k(G')| = |\mathrm{LC}_k(G_1)| \cdot |\mathrm{LC}_k(G_2)|$. Now, from the note after Definition 10.3 we get that

$$\begin{aligned} |\mathrm{ULC}_k(G')| &= \frac{|\mathrm{LC}_k(G')|}{|\mathrm{Aut}(G')|} \\ &= \frac{|\mathrm{LC}_k(G_1)| \cdot |\mathrm{LC}_k(G_2)|}{|\mathrm{Aut}(G_1)| \cdot |\mathrm{Aut}(G_2)|} \\ &= |\mathrm{ULC}_k(G_1)| \cdot |\mathrm{ULC}_k(G_2)| = l_1 \cdot l_2. \end{aligned}$$

□

Theorem 10.4 *Let G be a graph with $|\mathrm{ULC}_k(G)| = l$, then for any integer a with $1 \leq a \leq l$ there exists a graph G' with $|\mathrm{ULC}_k(G')| = \binom{l}{a}$.*

The idea is to make G' consist of a independent copies of G. We can then colour each copy independently, but two copies are not allowed to be coloured in the same way and two colourings of G' can not be distinguished if they use the same colourings for the copies. Thus the number of unique legal colourings of G' is the number of ways we can choose a out of l unique legal colourings of G, ignoring the order.

Proof. Let $G = (V, E)$ be such that $|\mathrm{ULC}_k(G)| = l$. We can assume that $l > 0$ as the claim in the theorem is otherwise empty. Also, we can assume that G is connected (otherwise just use handle(G, k)). Given an integer a with $1 \leq a \leq l$ let $G' = (V', E')$ be the graph consisting of a copies of G:

$$\begin{aligned} V'_i &= \{v_{v,i} : v \in V\} && i \in [a], \\ E'_i &= \{(v_{v_1,i}, v_{v_2.i}) : (v_1, v_2) \in E\} && i \in [a], \\ V' &= \bigcup_{i \in [a]} V'_i, \\ E' &= \bigcup_{i \in [a]} E'_i. \end{aligned}$$

We will prove the theorem by proving

$$|\mathrm{Aut}(G')| = a! \cdot |\mathrm{Aut}(G)|^a, \tag{10.16}$$

$$|\mathrm{LC}_k(G')| = \binom{l}{a} \cdot a! \cdot |\mathrm{Aut}(G)|^a. \tag{10.17}$$

As G is connected and there are no edges between the copies of G in G', any automorphism of G' will map each copy of G to a copy of G. Thus, if we consider how an automorphism of G' works on a single copy of G it will correspond to an automorphism of G. This shows that any automorphism of G' can be characterised by a permutation of the a copies of G and a list (thus, the order of the elements matters) with a automorphisms of G. On the other hand it is clear that any permutation of the a copies of G and list with a

automorphisms of G describes an automorphism of G' and for different choices of permutation and automorphisms of G the described automorphisms of G' are different. This proves (10.16).

From the characterisation of automorphisms of G' also follows that for $c' \in \mathrm{LC}_k(G')$ each copy of G is coloured legally. Otherwise the automorphism of G' which is trivial except that it maps an illegally coloured copy into itself by an automorphism of G proving that it is not legally coloured, would prove that $c' \notin \mathrm{LC}_k(G')$. Furthermore, two of the copies can not be coloured with colourings from the same colour class of G, e.g. c_1 and c_2 where $c_1 = c_2 \circ \phi$, as there would then be an automorphism of G' mapping one copy to the other and vice versa. From this we get that any legal k-colouring of G' can be defined by a list with a legal k-colourings of G from different colour classes. On the other hand if we colour the copies with a legal k-colourings of G from different colour classes it is a legal k-colouring of G'. If this was not the case there would be a nontrivial automorphism of G', which would preserve the colouring. Such an automorphism would either map one copy of G to another and the colourings of the two copies would thus be from the same colour class, or would nontrivially map one copy into itself, which can not be the case as the copy is itself legally couloured.

We can thus conclude that the number of legal k-colourings of G' are the same as the number of lists with a legal k-colourings of G from different colour classes. As each colour class has size $|\mathrm{Aut}(G)|$ we get

$$\begin{aligned}
|\mathrm{LC}_k(G')| &= \prod_{i=0}^{a-1}(|\mathrm{LC}_k(G)| - i|\mathrm{Aut}(G)|) = \prod_{i=0}^{a-1}(l|\mathrm{Aut}(G)| - i|\mathrm{Aut}(G)|) \\
&= |\mathrm{Aut}(G)|^a \cdot \prod_{i=0}^{a-1}(l-i) = |\mathrm{Aut}(G)|^a \cdot \frac{l!}{(l-a)!} \\
&= \binom{l}{a} \cdot a! \cdot |\mathrm{Aut}(G)|^a,
\end{aligned}$$

and we have thus proved (10.17). By combining (10.16) and (10.17) we get

$$|\mathrm{ULC}_k(G')| = \frac{|\mathrm{LC}_k(G')|}{|\mathrm{Aut}(G')|} = \frac{\binom{l}{a} \cdot a! \cdot |\mathrm{Aut}(G)|^a}{a! \cdot |\mathrm{Aut}(G)|^a} = \binom{l}{a},$$

which completes the proof of the theorem. □

10.6 Concrete reductions

In this section we will take a closer look at which reductions are actually possible. We will primarily focus on reductions to DIST_2. We will use R_k to denote the set $\{|\mathrm{ULC}_k(G)| : G \text{ is a graph}\}$, i.e., $l \in R_k$ exactly if we by the method presented in this paper can reduce DIST_l to DIST_k. It should be noted that this does of course not imply that if $l \notin R_k$ it is not possible to reduce DIST_l to DIST_k, just that it is not possible by the method presented in this paper.

We have from Theorem (10.1) that we can reduce RIGID to DIST_k for any $k > 0$, so $\forall k > 0 : 1 \in R_k$. As a single vertex graph has k unique k-colourings, we also get that $\forall k \geq 0 : k \in R_k$.

In the following we will examine R_2, and primarily focus on the small numbers (below 40) of R_2. As $2 \in R_2$ we get from Theorem (10.3) that $\forall i > 0 : 2^i \in R_2$. As $\binom{4}{2} = 6$ we get from (10.4) that $\{\binom{4}{2}, \binom{6}{2}, \binom{6}{3}, \binom{8}{2}\} = \{6, 15, 20, 28\} \subset R_2$, and then by Theorem (10.3) that $\{12, 24, 30, 40\} \subset R_2$. It is easy to see that these in fact are all the numbers below 40 that Theorem (10.3) and (10.4) implies from $2 \in R_2$.

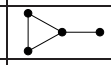
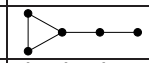
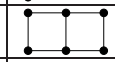
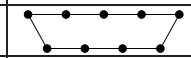
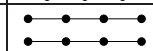
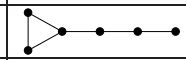
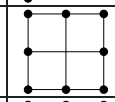
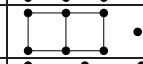
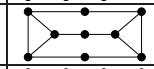
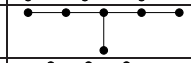
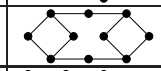
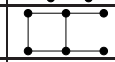
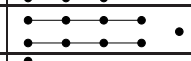
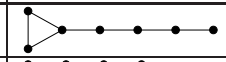
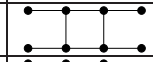
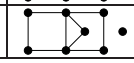
l	Graph	Proof
1		Theorem (10.1)
2		$k \in R_k$
4		Theorem (10.3) : $2 \cdot 2$
6		Theorem (10.4) : $\binom{4}{2}$
8		Theorem (10.3) : $2 \cdot 4$
10		
12		Theorem (10.3) : $2 \cdot 6$
14		
15		Theorem (10.4) : $\binom{6}{2}$
16		Theorem (10.3) : $2 \cdot 8$
18		
20		Theorem (10.4) : $\binom{6}{3}$
22		
24		Theorem (10.3) : $2 \cdot 12$
27		
28		Theorem (10.4) : $\binom{8}{2}$
30		Theorem (10.3) : $2 \cdot 15$
32		Theorem (10.3) : $2 \cdot 16$
36		Theorem (10.3) : $6 \cdot 6$
40		Theorem (10.3) : $2 \cdot 20$

Table 10.1: Graphs used in reductions to DIST_2

Table (10.1) contains numbers (below 40) in R_2 for which we have been able to find a graph proving that the number is in R_2. One could have hoped that Theorem (10.3) and (10.4) did describe how to get all possible reductions, but

as can be seen in the table this is not the case.

It should be mentioned that the graphs in Table (10.1) are all minimal with respect to the number of vertices, and thus not necessarily the graphs that come from the constructions in Theorem (10.3) and (10.4). When looking at the table one notices that graphs for most equal numbers can be created (the first we have not been able to create a graph for is 26), and that there are very few odd numbers. We do not believe this is a coincident and conjecture that $3 \notin R_2$. One could also get the suspicion that there are no primes (except for 2) in R_2, but this is not the case as the 12-cycle C_{12} has 127 unique legal 2-colourings.

The only numbers below 40 we have been able to prove to lie in R_3 are 1, 3, 9, 12 (C_5), 27, 36 and 37 (C_6). This makes us believe that R_3 is much sparser than R_2, but it may also just be an indication that the graphs needed to prove numbers in R_3 are generally larger (and thus more difficult to find and test) than the ones needed to find numbers in R_2.

10.7 Conclusion and open problems

In this paper we have provided a tool for making reductions from DIST_l to $DIST_k$ for various values of l and k. We gave some methods for constructing graphs with certain properties from smaller graphs.

Some of the graphs in Table (10.1) were found by generating all graphs [36] with at most 10 vertices and calculating the number of unique legal 2-colourings of each, but this method can of course only give a partial characterisation of R_2. A complete characterisation of R_2 (and R_k in general) would be of great interest. As we conjecture that the method can not be used for reducing DIST_3 to DIST_2, it would be very interesting to find other methods of reducing, in order to know whether for every $l \geq 0$ DIST_l can be reduced to DIST_2. Except when $l = 0$ or $l = 1$ all reductions one can obtain with the method presented in this paper have $k \leq l$ so it would also be interesting to know if it is possible to reduce DIST_l to $DIST_k$ for some values l and k where $1 < l < k$.

Acknowledgements

I would like to thank V. Arvind for presenting the problem to me during his visit at the University of Aarhus and my supervisors Sven Skyum and Peter Bro Miltersen. I would also like to thank Bolette A. Madsen and Jesper M. Byskov for many useful discussions and proofreading.

Bibliography

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] M. O. Albertson and K. L. Collins. Symmetry breaking in graphs. *Electronic Journal of Combinatorics*, 3 (#R18), 1996.

[3] V. Arvind and N. D. R. Symmetry breaking in trees and planar graphs by vertex coloring. To appear in *J. of Algorithms*.

[4] S. Baumer and R. Schuler. Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs. In E. Giunchiglia and A. Tacchella, editors, *Proc. 6th Conf. SAT 2003*, volume 2919 of *LNCS*, pages 150–161. Springer-Verlag, 2004.

[5] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3446^n)$: a no-MIS algorithm. In *Proc. 36th Symp. Foundations of Computer Science*, pages 444–453. IEEE, Oct. 1995.

[6] R. B. Boppana, J. Hastad, and S. Zachos. Does co-NP have short interactive proofs? *Inf. Process. Lett.*, 25(2):127–132, 1987.

[7] J. M. Byskov. Algorithms for k-colouring and finding maximal independent sets. In *Proc. 14th Symp. Discrete Algorithms*, pages 456–457. ACM and SIAM, Jan. 2003.

[8] J. M. Byskov. Enumerating maximal independent sets with applications to graph colouring. *Operations Research Letters*, 32(6):547–556, Nov. 2004.

[9] J. M. Byskov and D. Eppstein. An algorithm for enumerating maximal bipartite subgraphs. Unpublished, 2004.

[10] J. M. Byskov, B. A. Madsen, and B. Skjernaa. New algorithms for exact satisfiability. *Theoretical Comput. Sci.*, 2004. To appear.

[11] J. M. Byskov, B. A. Madsen, and B. Skjernaa. On the number of maximal bipartite subgraphs of a graph. *J. Graph Theory*, 2004. In press.

[12] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM Press, 1971.

[13] V. Dahllöf, P. Jonsson, and R. Beigel. Algorithms for four variants of the exact satisfiability problem. *Theoretical Comput. Sci.*, 320(2–3):373–394, June 2004.

[14] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. H. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Comput. Sci.*, 289(1):69–83, Oct. 2002.

[15] E. Dantsin, A. Goerdt, E. A. Hirsch, and U. Schöning. Deterministic algorithms for k -SAT based on covering codes and local search. In *Automata, Languages and Programming*, pages 236–247, 2000.

[16] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[17] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[18] L. Drori and D. Peleg. Faster exact solutions for some NP-hard problems. *Theoretical Comput. Sci.*, 287(2):473–499, Sept. 2002.

[19] D. Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *Proc. 12th Symp. Discrete Algorithms*, pages 329–337. ACM and SIAM, Jan. 2001.

[20] D. Eppstein. Small maximal independent sets and faster exact graph coloring. *J. Graph Algorithms & Applications*, 7(2):131–140, 2003.

[21] S. Fajtlowicz. Written on the wall.

[22] S. Fajtlowicz and S. Skiena. A database of counterexamples to conjectures by graffiti.

[23] S. S. Fedin and A. S. Kulikov. Automated proofs of upper bounds on the running time of splitting algorithms. In F. Dehne, R. Downey, and M. Fellows, editors, *Proc. Int. Worksh. Parameterized and Exact Computation*, volume 3162 of *LNCS*. Springer-Verlag, Sept. 2004.

[24] W. Feller. *An introduction to Probability Theory and its Applications*. Wiley, New York, 1968.

[25] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated generation of search tree algorithms for graph modification problems. *Algorithmica*, 39(4):321–347, May 2004.

[26] D. Gutfreund, R. Shaltiel, and A. Ta-Shma. Uniform hardness vs. randomness tradeoffs for arthur-merlin games. In *Eighteenth Annual IEEE Conference on Computational Complexity*, pages 33–47, 2003.

[27] E. A. Hirsch. New worst-case upper bounds for SAT. *J. Autom. Reason.*, 24(4):397–420, 2000.

[28] E. A. Hirsch and A. S. Kulikov. A $2^{n/6.15}$-time algorithm for X3SAT, 2002.

[29] T. Hofmeister, U. Schöning, R. Schuler, and O. Watanabe. A probabilistic 3-sat algorithm further improved. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science*, pages 192–202. Springer-Verlag, 2002.

[30] A. S. Kulikov. An upper bound $O(2^{0.16254n})$ for exact 3-satisfiability: A simpler proof. *Zapiski nauchnyh seminarov POMI*, 293:118–128, 2002. English translation to appear in *J. Mathematical Sciences*. `http://logic.pdmi.ras.ru/~kulikov/`.

[31] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Comput. Sci.*, 223(1–2):1–72, July 1999.

[32] O. Kullmann and H. Luckhardt. Deciding propositional tautologies: Algorithms and their complexity, 1997. Available at `http://cs-svr1.swan.ac.uk/~csoliver/tg.ps.gz`.

[33] O. Kullmann and H. Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages, Feb. 1998.

[34] E. L. Lawler. A note on the complexity of the chromatic number problem. *Inf. Process. Lett.*, 5(3):66–67, Aug. 1976.

[35] B. A. Madsen. Personal communication, 2003.

[36] B. D. McKay. Nauty v. 2.2. Nauty contains a program *geng* to generate all graphs with certain properties. It can be found at `http://cs.anu.edu.au/people/bdm/nauty/`.

[37] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Appl. Math.*, 10:287–295, 1985.

[38] B. Monien, E. Speckenmeyer, and O. Vornberger. Upper bounds for covering problems. Bericht 7, Universität Paderborn, 1980.

[39] B. Monien, E. Speckenmeyer, and O. Vornberger. Upper bounds for covering problems. *Methods of Operations Research*, 43:419–431, 1981.

[40] J. W. Moon and L. Moser. On cliques in graphs. *Israel J. Mathematics*, 3:23–28, 1965.

[41] S. I. Nikolenko and A. V. Sirotkin. Worst-case upper bounds for SAT: automated proof. In B. ten Cate, editor, *Proc. 8th ESSLLI Student Session*, pages 225–232, Aug. 2003. `http://logic.pdmi.ras.ru/~sergey/`.

[42] S. Porschen, B. Randerath, and E. Speckenmeyer. Exact 3-satisfiability is decidable in time $O(2^{0.16254})$. *Annals of Mathematics and Artificial Intelligence*, 2004. In press.

[43] F. Rubin. Problem 729. the blind man's keys. *Journal of Recreational Mathematics*, 11(2):128, 1979. (solution in volume 12(2), 1980).

[44] A. Russell and R. Sundaram. A note on the asymptotics and computational complexity of graph distinguishability. *Electronic Journal of Combinatorics*, 5 (#R23), 1998.

[45] T. J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th Symp. Theory of Computing*, pages 216–226. ACM, 1978.

[46] I. Schiermeyer. Deciding 3-colourability in less than $O(1.415^n)$ steps. In J. van Leeuwen, editor, *Proc. 19th Int. Worksh. Graph-Theoretic Concepts in Computer Science*, volume 790 of *LNCS*, pages 177–188. Springer-Verlag, 1994.

[47] I. Schiermeyer. Fast exact colouring algorithms. *Tatra Mountains Mathematical Publications*, 9:15–30, 1996.

[48] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, New York, NY, USA*, pages 410–414. IEEE Press, 1999.

[49] R. Schroeppel and A. Shamir. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM J. Comput.*, 10(3):456–464, Aug. 1981.

[50] R. Schuler, U. Schöning, and O. Watanabe. An improved randomized algorithm for 3-sat. Technical Report TR-C146, Dept. of Mathematical and Computing Sciences, Tokyo Inst. of Tech., 2001.

[51] U. Schöning. A probabilistic algorithm for k-SAT based on limited local search and restart. *Algorithmica*, 32(4):615–623, Jan. 2002.

[52] B. Skjernaa. Automated generation of branching algorithms with upper bound proofs for variants of SAT. Unpublished, 2004.

[53] B. Skjernaa. *Exact Algorithms for Variants of Satisfiability and Colouring Problems*. PhD thesis, University of Aarhus, 2004.

[54] B. Skjernaa. Reductions among graph distinguishability problems. Unpublished, 2004.

[55] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, Sept. 1977.

[56] R. Williams. A new algorithm for optimal constraint satisfaction and its implications. In *Proc. 31st Int. Coll. Automata, Languages and Programming*, volume 3142 of *LNCS*, pages 1227–1237. Springer-Verlag, July 2004.

Recent BRICS Dissertation Series Publications

DS-04-5 **Bjarke Skjernaa.** ***Exact Algorithms for Variants of Satisfiability and Colouring Problems*****. November 2004. PhD thesis. x+112 pp.**

DS-04-4 **Jesper Makholm Byskov.** ***Exact Algorithms for Graph Colouring and Exact Satisfiability*****. November 2004. PhD thesis.**

DS-04-3 **Jens Groth.** ***Honest Verifier Zero-knowledge Arguments Applied*****. October 2004. PhD thesis. x+112 pp.**

DS-04-2 **Alex Rune Berg.** ***Rigidity of Frameworks and Connectivity of Graphs*****. July 2004. PhD thesis. xii+173 pp.**

DS-04-1 **Bartosz Klin.** ***An Abstract Coalgebraic Approach to Process Equivalence for Well-Behaved Operational Semantics*****. May 2004. PhD thesis. x+152 pp.**

DS-03-14 **Daniele Varacca.** ***Probability, Nondeterminism and Concurrency: Two Denotational Models for Probabilistic Computation*****. November 2003. PhD thesis. xii+163 pp.**

DS-03-13 **Mikkel Nygaard.** ***Domain Theory for Concurrency*****. November 2003. PhD thesis. xiii+161 pp.**

DS-03-12 **Paulo B. Oliva.** ***Proof Mining in Subsystems of Analysis*****. September 2003. PhD thesis. xii+198 pp.**

DS-03-11 **Maciej Koprowski.** ***Cryptographic Protocols Based on Root Extracting*****. August 2003. PhD thesis. xii+138 pp.**

DS-03-10 **Serge Fehr.** ***Secure Multi-Player Protocols: Fundamentals, Generality, and Efficiency*****. August 2003. PhD thesis. xii+125 pp.**

DS-03-9 **Mads J. Jurik.** ***Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols*****. August 2003. PhD thesis. xii+117 pp.**

DS-03-8 **Jesper Buus Nielsen.** ***On Protocol Security in the Cryptographic Model*****. August 2003. PhD thesis. xiv+341 pp.**

DS-03-7 **Mario José Cáccamo.** ***A Formal Calculus for Categories*****. June 2003. PhD thesis. xiv+151.**

DS-03-6 **Rasmus K. Ursem.** ***Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization*****. June 2003. PhD thesis. xiv+183 pp.**