Linköping Studies in Science and Technology

Dissertation No. 1013

Exact Algorithms for Exact Satisfiability Problems

by

Vilhelm Dahllöf

Linköpings universitet
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköping 2006

ISBN 91-85523-97-6
ISSN 0345-7524
Printed by LiU-tryck, Linköping 2006

Exact Algorithms for Exact Satisfiability Problems

Vilhelm Dahllöf

Abstract

This thesis presents exact means for solving a family of *NP*-hard problems. Starting with the well-studied Exact Satisfiability problem (XSAT) parents, siblings and daughters are derived and studied, each with interesting practical and theoretical properties. While developing exact algorithms to solve the problems, we gain new insights into their structure and mutual similarities and differences.

Given a Boolean formula in CNF, the XSAT problem asks for an assignment to the variables such that each clause contains exactly one true literal. For this problem we present an $\mathcal{O}(1.1730^n)$ time algorithm, where n is the number of variables. XSAT is a special case of the General Exact Satisfiability problem which asks for an assignment such that in each clause exactly i literals be true. For this problem we present an algorithm which runs in $\mathcal{O}\left(2^{(1-\varepsilon)n}\right)$ time, with $0 < \varepsilon < 1$ for any fixed i; for $i = 2, 3$ and 4 we have running times in $\mathcal{O}(1.4511^n)$, $\mathcal{O}(1.6214^n)$ and $\mathcal{O}(1.6848^n)$ respectively.

For the counting problems we present an $\mathcal{O}(1.2190^n)$ time algorithm which counts the number of models of an XSAT instance. We also present algorithms for $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$, two well studied Boolean problems. The algorithms have running times in $\mathcal{O}(1.2561^n)$ and $\mathcal{O}(1.6737^n)$ respectively.

Finally, we study optimisation problems: As a variant of the Maximum Exact Satisfiability problem, consider the problem of finding an assignment exactly satisfying a maximum number of clauses while the rest are left with no true literal. This problem is reducible to $\#2\text{SAT}_w$ without introducing new variables and thus is solvable in $\mathcal{O}(1.2561^n)$ time. Another interesting optimisation problem is to find two XSAT models which differ in as many variables as possible. This problem is shown to be solvable in $\mathcal{O}(1.8348^n)$ time.

Acknowledgements

Most of the PhD theses I have seen have had long and entertaining lists of people being thanked in eloquent and witty ways. Having a problem finding the right words I decided to turn to an old Swedish dissertation for inspiration. My choice was *Beiträge zur Kenntnis der Biochemie der Acetonkörper* (in those days, Swedes usually wrote their dissertations in German) by N.O. Engfeldt [31]. I was sure it would contain an enumeration of colleagues, wives, parents, friends, the house maid and the dog, all receiving laud in ornate sentences. However, nothing of the kind was found. The preface related the story behind the thesis and only the last paragraph contained some dry words of thanks. After some moments of disappointment, I decided to do similarly. After all, there is a fine line between distributing thanks to the whole world and thanking no-one.

I thank my supervisor Peter Jonsson, without whom this thesis would never have been accomplished. I thank Ola Angelsmark and Magnus Wahlström for co-operation in writing papers and proofreading. Ulf Nilsson has also been helpful in proofreading the manuscript. My co-supervisors have been Sten F. Andler and Jörgen Hansson. This research work was funded in part by CUGS (the National Graduate School in Computer Science, Sweden), and by the Swedish Research Council (VR) under grant 621-2002-4126.

Linköping, Sweden, April 2006

Vilhelm Dahllöf

List of papers

Parts of this thesis have been previously published in the following refereed papers:

- Vilhelm Dahllöf and Peter Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2002)*, pages 292–298, 2002. [17]
- Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting satisfying assignments in 2SAT and 3SAT. In *Proceedings of Computing and Combinatorics, 8th Annual International Conference, (COCOON-2002)*, pages 535–543. 2002. [19]
- Vilhelm Dahllöf, Peter Jonsson, and Richard Beigel. Algorithms for four variants of the exact satisfiability problem. *Theoretical Computer Science*, 320(2–3):373–394, 2004. [18]
- Vilhelm Dahllöf. Applications of general exact satisfiability in propositional logic modelling. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-2004)*, pages 95–109, 2004. [15]
- Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1–3):265–291, 2005. [20]
- Vilhelm Dahllöf. Algorithms for max Hamming exact satisfiability. In *Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC-2005)*, pages 829–838, 2005. [16]

Contents

I Introduction and Background **1**

1 Introduction **3**
1.1 General Background 6
1.2 Objectives and Problem Descriptions 9
1.2.1 Exact Satisfiability 10
1.2.2 General Exact Satisfiability 13
1.2.3 Counting Exact Satisfiability 16
1.2.4 Counting 2SAT and Counting 3SAT 20
1.2.5 Maximum Exact Satisfiability 22
1.2.6 Max Hamming Exact Satisfiability 24
1.3 Summary of Results 25
1.4 Outline of the Thesis 26

2 Preliminaries **29**
2.1 Formula Related Concepts 29
2.2 Graph Related Concepts 31
2.3 Problem Definitions 32
2.4 Branching Algorithms 34
2.5 Canonisation 36
2.6 Reduction to Matching 40

II Deciding **43**

3 Exact Satisfiability **45**

3.1 A Polynomial Space Exact Algorithm 45

4 General Exact Satisfiability 55

4.1 Properties of the X_iSAT Problem 55
4.2 A Polynomial Space Exact Algorithm 58
4.3 An Algorithm for X_2SAT 63
4.4 Solving X_iSAT in Exponential Space 71

III Counting 75

5 Counting Exact Satisfiability 77

5.1 Preliminaries 77
5.2 Algorithm for #X3SAT$_w$ 83
5.3 Algorithm for #XSAT$_w$ 87

6 Counting 2SAT and Counting 3SAT 91

6.1 Algorithm analysis 91
6.2 Propagation 92
6.2.1 Multiplier Reduction 95
6.3 Algorithm for #2SAT$_w$ 97
6.3.1 The function C_3 97
6.3.2 The function C_5 100
6.3.3 The main function C 115
6.4 Algorithm for #3SAT$_w$ 116
6.5 Algorithm C_{sep} 124
6.6 Applications 129

IV Optimising 133

7 Maximum Exact Satisfiability 135

7.1 On the Hardness of MAX XSAT 135
7.2 Solving RESTRICTED MAX XSAT 137

8 Max Hamming Exact Satisfiability 141
8.1 Using an External XSAT Solver 141
8.2 Using Branching 145
8.2.1 Extra Preliminaries 145
8.2.2 The Algorithm Q' 148
8.2.3 Improving and Analysing the Running Time . 155

9 Conclusions and Future Work 159
9.1 Decision Problems 159
9.2 Counting Problems 161
9.3 Optimisation Problems 163
9.4 Other Problems of the XSAT family 165
9.5 Final Remarks 166

Bibliography 169

Part I

Introduction and Background

Chapter 1

Introduction

Even the ancient Greeks concerned themselves with efficient algorithms. One celebrated example is the "sieve of Eratosthenes" [24] which enumerates all prime numbers up to n in time $\mathcal{O}(n \log \log n)$[1]. Despite the often speculative nature of Greek thinking, the Greeks apparently never asked the question of which problems allow efficient algorithms and which do not. At least we have no records left of any such lines of thought.

Many centuries later, in 1736, Leonhard Euler wrote his famous paper "The Seven Bridges of Königsberg" [32]. It treats a graph problem known as *Euler walk*: Given a number of islands connected by bridges, is it possible to visit every island, crossing each bridge exactly once? Euler proved that this problem is efficiently solvable: If there are more than two islands having an odd number of bridges, then the Euler walk is impossible, otherwise there must be such a walk! In Figure 1.1 an instance of this problem is given. Note that using Euler's algorithm, we do not actually need to find a solution in order to give a *yes* or *no* answer. Now, what about this almost identical problem: Given a number of islands connected by bridges, is it possible to visit every island exactly once in a walk? This problem is called

[1] The cover of the thesis uses two pages from Ἀριθμητικῆς εἰσαγωγή (Introduction to arithmetic) [44] by Nicomachus of Gerasa (flourished circa A.D. 100), where the sieve of Eratosthenes is first described.

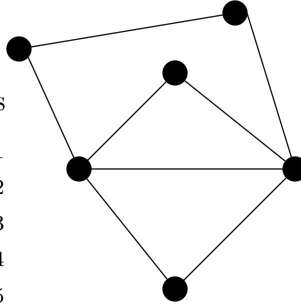

Figure 1.1: *Euler walk: Visit all vertices, using each edge exactly once. For this graph, there is an Euler walk*

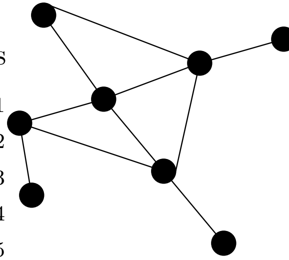

Figure 1.2: *Hamilton walk: Visit every vertex once. This graph has no Hamilton path*

Hamilton walk and we know no efficient algorithm for it. That is, all known real-world algorithms run in time that grows exponentially in the number of islands. Figure 1.2 presents an example of a graph without a Hamilton path.

The last century saw the foundation of a science capable of classifying computational problems in a more satisfactory way. In 1936 Alan Turing [72] presented imaginary machines, that can be seen as a form of computers, able to run algorithms. It is a widely held belief that these machines can compute everything that is computable, because all other known models of computation have been shown to be

equivalent to Turing machines with respect to *what* can be computed.

When it comes to the question of *how* a solution to a problem can be computed, refined models of the Turing machines have been proposed. The original Turing machine works in a deterministic, non-parallel way. Though somewhat exotic in design, it is still reasonable in the sense that it can be mimicked by a real-world computer without too much loss in time and memory. At the end of the 1950's, non-deterministic Turing machines were considered [70]. These models are unreasonable in the sense that during a computation, the right choices are always made (if the problem is solvable.) Using the Hamilton path as an example, such a machine would look at all the edges, and then pick the first one in a Hamilton path, then the second one in the path and so on until a full Hamilton path is obtained. Thus, for this model of computation, we can solve the problem in polynomial time. With these machines (and other variants) as a tool, it has become possible to classify computational problems. For instance, Euler walk belongs to the class of problems which can by solved by a deterministic machine in polynomial time, while the Hamilton walk, as we believe, would require a non-deterministic machine if we wanted to solve it in polynomial time.

The set of problems that can be solved in polynomial time on a deterministic Turing machine is called P and the set of problems that can be solved in polynomial time on a non-deterministic machine is called NP. It is clear that $P \subseteq NP$, and it is a widely held belief that $P \subset NP$, see [39]. This is known as the $P \neq NP$ conjecture and, if true, it implies, among other things, that Hamilton path cannot be solved in polynomial time on an ordinary computer. The formulation of the $P \neq NP$ conjecture and the definition of the class NP by Karp [48] in 1972 must be seen as a breakthrough in the process of classifying computational problems. We are certainly better off than the ancient Greeks.

1.1 General Background

Boolean formulae play an important rôle in this thesis. We will therefore start with a small toy example from propositional logic. Let p be the statement "It rains" and q the statement "The sun is shining." The two statements are either true or false, and therefore p and q can be seen as (Boolean) variables that are assigned either true or false. Now the compound statement "If it rains the sun is not shining" can be written as $p \to \bar{q}$ (where $\bar{q}$ is the negation of q.) Let us add the assumption that q is true so that we get the formula $q \wedge (p \to \bar{q})$. Given this formula, one can ask if there is an assignment to p and q such that the entire formula becomes true, i.e. satisfied. Such an assignment is known as a *model* and there is clearly only one for this formula, namely $q = true, p = false$. The problem of deciding whether a propositional logic formula has a model is called SATISFIABILITY, abbreviated SAT.

There are many logical connectives such as $\wedge, \vee, \to, \leftrightarrow$ etc, and it is a well known fact that not all are needed. As a convention, the instances of SAT are written in conjunctive normal form (CNF.) This means that the variables (possibly negated) appear in disjunctions (i.e. they are glued together using logical or ($\vee$)) called clauses. The clauses in turn appear in conjunctions, (i.e. they are glued together using logical and ($\wedge$).) Our example given in CNF:

$$(\bar{p} \vee \bar{q}) \wedge (q)$$

In the rest of the thesis we will omit the AND-symbols, sometimes replacing them with commas, which is easier to read for larger formulae. We will return to SAT several times.

Hamilton path, Euler walk and SAT are decision problems, i.e. we expect to get a 'yes' or 'no' answer. This thesis will treat not only decision problems but also counting problems and optimisation problems. As an example of a counting problem, we have the problem of *how many* Hamilton paths a graph contains. For a related optimisation problem, assign weights to the edges of a graph and search for a Hamilton path minimising the overall weight of the edges. Note that as soon we know the solution to either the optimisation prob-

lem or the counting problem, then we also know the solution to the decision problem. When we have this situation — that the answer to problem A with only polynomial computational costs can be transformed to an answer to problem B — then we say that A is *at least as hard as* B and that there is a (polynomial) reduction from A to B. It turns out that Hamilton path is at least as hard as every other problem in *NP* — it is *NP*-hard (as are the derived optimisation and counting problems.) SAT is interesting not only because it can be used to model real-world phenomena; it also plays a special rôle in the history of complexity theory. In 1971, SAT was the first problem to be proved *NP*-complete by Cook [12] (*NP*-complete means that it is both in *NP* and *NP*-hard.)

Many of the *NP*-hard problems, such as SAT, have important practical applications, and in order to solve them, different approaches have been tried. One can try to use *heuristic methods.* Such methods make no guarantee that we will obtain a solution, but practise has shown that they often work. They perform a relatively limited exploration of the search space and produce solutions within modest computing times. One of the most common heuristic methods is *simulated annealing* [50]. It is based on an analogy from the annealing process of steel or some other solid. First the steel is heated to high temperatures and then it is gradually cooled. Thus, at first, the atoms have high energy and move at high speed, but at later stages their movements are more constrained. In terms of solving SAT the analogy works as follows:

Every assignment to the variables satisfies a certain number of clauses, the more the better. During the process, we flip the assignments to single variables to obtain better solutions. In the beginning, with a certain high probability (the analogue of high temperature) we may make assignments that satisfy fewer clauses, but gradually the probability decreases and we converge towards a solution. The idea behind this heuristic is that the initial high temperature (probability) will help us avoid getting stuck at local optima. Note that "solution" here does not mean model, but rather an assignment satisfying an unspecified (hopefully high) number of clauses.

Apart from the heuristic algorithms, there are polynomial time algorithms for many hard problems that can make guarantees on the output produced. *Randomised algorithms* return an acceptable solution with a certain (guaranteed) probability. *Approximation algorithms* return solutions within a bounded distance from an optimal solution. Returning to SAT, we have the following, trivial, approximation algorithm: Choose an assignment at random, and if less than half of the clauses are satisfied, assign each variable the opposite value. This algorithm guarantees that at least half of the clauses are satisfied. Of course, there are non-trivial algorithms also. Yannakakis [78] for instance, shows how 3/4 of the clauses can be satisfied.

This thesis is concerned with *exact algorithms* for *NP*-hard problems. Although such algorithms all run in super-polynomial time in the worst case (and always will be, assuming that $P \neq NP$), they must be taken into account. In many cases, approximation algorithms cannot return acceptable results and then we have to use algorithms with exponential time complexity. For instance, an assignment $q = false, p = false$ in our toy SAT example $(\bar{p} \vee \bar{q}) \wedge (q)$ may be theoretically satisfying (because it is an approximation satisfying half of the clauses) but the solution we really want is a model. The price to pay for solving SAT exactly is that an $\mathcal{O}\left(2^{n-2\sqrt{n/\log n}}\right)$ time algorithm must be used, where n is the number of variables. This algorithm was presented by Dantsin et al. [21] in 2004 and is so far the best when measuring formula size in the number of variables.

When constructing exponential time algorithms, which have running times in $\mathcal{O}(c^{\mu})$ for some constant c and some measure μ of the instance size, it is crucial to reduce c as much as possible — reducing c to its square root doubles the size of the largest instance possible to solve. Resorting to the use of exponential space algorithms, the time complexity can sometimes be considerably improved. One example of this are the best known algorithms for MAXIMUM INDEPENDENT SET (MIS) problem; the problem is depicted in Figure 1.3. MIS is *NP*-complete (see [38]) and the so far fastest algorithm for this problem by Robson [64] runs in $\mathcal{O}(1.1889^{n})$ and uses an exponential amount of memory. This is to be compared with the fastest poly-

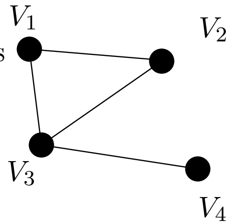

Figure 1.3: MIS*: A maximum independent set is a largest possible subset of the vertices of a graph such that no pair in the subset is joined by an edge. In this graph, the set* $\{V_2, V_4\}$ *constitutes a* MIS

nomial space algorithm, also by Robson, which runs in $\mathcal{O}(1.2025^n)$ time. We will encounter this phenomenon later in this thesis, in the context of X$_i$SAT, which is the problem treated in Chapter 4. For practical use, exponential space requirements of an algorithm are of course highly undesirable.

1.2 Objectives and Problem Descriptions

In this thesis we will investigate a family of *NP*-hard Boolean problems. Their common property is that the problems are concerned with the question of *how* a clause is satisfied. As opposed to the archetype Boolean problem, SAT, which considers a clause satisfied if it contains *at least* one true literal (a literal is variable or a negated variable), EXACT SATISFIABILITY (XSAT) requires *exactly* one true literal. XSAT can be seen as the starting point for deriving a number of other interesting problems. A generalised decision problem of XSAT is X$_i$SAT where we require a clause to contain exactly i true literals. One can also generalise XSAT to a optimisation problem by searching for an assignment such that a maximum number of clauses are satisfied. Other members of the family are the problem of counting the number of solutions, and the problem of finding two solutions that are as different as possible. Treating all these different problems together has proved fruitful, since the same techniques can often be

reused. This is foremost seen in the so-called canonical rules, which we will treat in Chapter 2.

As will be seen in this thesis, this family of problems is interesting not only for theoretic reasons. X_iSAT (and its special case XSAT) can enhance SAT modelling tools. Counting models for certain XSAT instances is equivalent to a problem called *computing the permanent of a 0/1 matrix* which has many applications, see e.g. [47]. The academic study of the problems concerned with finding solutions that are as different as possible has just begun, but the practical applications are manifold. For instance, when scheduling a collection of activities, one typically wants to present substantially different alternatives to choose between. Other examples are given by Hebrard et al. [42].

1.2.1 Exact Satisfiability

XSAT is a well studied problem. This Boolean problem, even when restricted to formulae with maximum clause length 3 (a problem called $X_1$3SAT), was proved *NP*-complete by Schaefer [67] in 1978. We will in general refer to the solutions as models, sometimes x-models, when there is a need to distinguish them from SAT solutions. Note that unlike SAT, there are no partial assignments that can be easily verified to guarantee a model. Just because all clauses are satisfied we are not necessarily done — we must also make sure no clause is over-satisfied.

Exact algorithms for XSAT have been presented by several authors [10, 18, 26, 28, 43, 63, 68]. The algorithm presented in this thesis is the currently fastest when measuring in the number of variables. XSAT can of course be solved within the trivial time bound of $\mathcal{O}(2^n)$, where n is the number of variables. This is done by exhaustively trying all 2^n assignments to the variables. The first algorithm to beat this was presented in 1981 by Schroeppel and Shamir [68]. Their algorithm runs in time $\mathcal{O}(1.4143^n)$ and space $\mathcal{O}(1.1893^n)$. (In Chapter 4 we will give a more thorough description of this algorithm.) Also in 1981 an algorithm running in time $\mathcal{O}(1.1843^n)$ was presented by Monien et al. [60]. This, and all the following algorithms, have polynomial space requirements and are built on two ideas: Canonical rules and good branching variables.

$x = (a \vee b \vee c \vee d)$	$x' = (b \vee c \vee d)$
$y = (a \vee \bar{b} \vee f \vee g)$	$y' = (\bar{b} \vee f \vee g)$
$z = (h \vee i)$	

Figure 1.4: *Two formulae: For $F = \{x, y, z\}$ canonical rules are applicable; $F' = \{x', y'\}$ is the result of applying two canonical rules on F*

To illustrate the first idea: Consider the XSAT instance F of Figure 1.4. Given the fact that only one literal must be true in each clause, a must be false. Otherwise, if a is true, then either b or $\bar{b}$ will make a clause over-satisfied. Hence, we can assign a false and remove it. This is an example of a canonical rule, i.e. a polynomial time reduction of an XSAT instance. The same figure may also illustrate another rule: If there is a clause of length 2, such as z, we may reduce the formula. As only one literal is allowed to be true in z, h and i must be assigned opposite values. This means that we can substitute every occurrence of h by $\bar{i}$ and every occurrence of $\bar{h}$ by i and so we have removed the variable h from the formula. The resulting formula is shown as F' of 1.4.

As for the second idea, one systematic approach to solve an instance of XSAT is built on the simple observation that a variable a is either true or false. This gives us two smaller XSAT instances to consider, one where a is true, and one where a is set to false and then canonical rules are applied. The two smaller instances can be solved by repeating the procedure. Of course, in practice we do not have the two instances at the same time, but rather first explore one instance, and then, if no solution is found, we explore the other. For an example, consider this instance:

$$\begin{aligned} x &= (a \vee b \vee c) \\ y &= (a \vee d \vee e \vee f) \\ z &= (d \vee g \vee h \vee i) \end{aligned}$$

Let us choose e to branch on (i.e. tentatively assign it true or false.) We start by assigning e true. By the nature of the XSAT

problem, we deduce that a, d and f must be assigned false and that b can be replaced by $\bar{c}$ (because x shrunk to length 2.) The only remaining clause in the instance is $(g \vee h \vee i)$. We now choose h to branch on and assign it true. The last clause is satisfied and g and i are set to false. Now we have found a model. This procedure was first presented by Davis et al. [22] for solving SAT and is known as DPLL *branching*. In the example, we were lucky and found a model without having to try other assignments to the branching variables, but this is of course not always the case. In general, an exponential running time can be expected. Note that when a variable occurs in more than two clauses, it will have many neighbours and so is often a good choice for branching. Such variables are called *heavy*.

In 1999, Drori and Peleg presented a dedicated $X_1$3SAT algorithm running in time $\mathcal{O}(1.1545^n)$. This far, the improved upper bounds on the running time was due only to invention of canonical rules and clever choices of branching variables. The next major leap forward came with Porschen et al. [63] who in 2002 presented an $\mathcal{O}(1.1382^n)$ time $X_1$3SAT algorithm. The improved running time is due to the discovery that polynomial time matching techniques can be used when the instance contains no heavy variables, thereby avoiding the necessity to deal with non-heavy variables using branching. The next improvement was done by Byskov et al. [10] and Dahllöf et al. [18], who, independently of each other, developed the concept of "sparsity." The idea is that when the ratio heavy variables to non-heavy variables is small enough (the instance is sparse), one can afford to examine all possible assignments to the heavy variables. When all heavy variables are assigned values, the rest of the instance can be solved by the matching techniques. The previously best algorithm is the Byskovian with a running time in $\mathcal{O}(1.1749^n)$. Our new algorithm has a running time in $\mathcal{O}(1.1730^n)$.

Our main focus in this thesis will be on exact algorithms that measure their running times in the number of variables. However, it is interesting to relate other known results. Measuring the formula size in the number of clauses, m, Madsen [58] in 2006 showed that XSAT can be solved in exponential space and time $\mathcal{O}(2^m)$ or polynomial

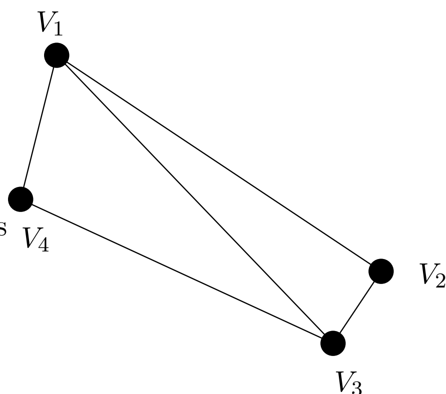

Figure 1.5: *A 3-colourable graph*

space and time $\mathcal{O}(m!)$. In a still unpublished paper by Björklund and Husfeldt [8], this has been improved to polynomial space and time $\mathcal{O}(2^m)$.

1.2.2 General Exact Satisfiability

SAT is often a convenient modelling tool. However, there are cases when large formulae are needed to express quite simple things. Consider for instance the graph in Figure 1.5. Assume that we have three colours and want to assign them to the vertices V_1, V_2, V_3 and V_4 of the graph with the following two constraints: 1) each vertex gets exactly one colour and 2) two vertices that share an edge do not have the same colour. This problem is known as the 3-COLOURABILITY PROBLEM.

In Figure 1.6 we see how the 3-colourability problem for the graph of Figure 1.5 can be expressed in terms of SAT. Twelve Boolean variables are created, each with an intended meaning such as "Vertex V_i has colour Q." For instance, c_R^2 can be interpreted as "Vertex V_2 has colour RED." The first line (i.e. the first four clauses) ensures that each vertex is assigned *at least* one colour. The following three lines make sure that each vertex is assigned *at most* one colour. Lines

1	$(c_R^1 \vee c_G^1 \vee c_B^1)$	$(c_R^2 \vee c_G^2 \vee c_B^2)$	$(c_R^3 \vee c_G^3 \vee c_B^3)$	$(c_R^4 \vee c_G^4 \vee c_B^4)$
2	$(\bar{c}_R^1 \vee \bar{c}_G^1)$	$(\bar{c}_R^2 \vee \bar{c}_G^2)$	$(\bar{c}_R^3 \vee \bar{c}_G^3)$	$(\bar{c}_R^4 \vee \bar{c}_G^4)$
3	$(\bar{c}_G^1 \vee \bar{c}_B^1)$	$(\bar{c}_G^2 \vee \bar{c}_B^2)$	$(\bar{c}_G^3 \vee \bar{c}_B^3)$	$(\bar{c}_G^4 \vee \bar{c}_B^4)$
4	$(\bar{c}_B^1 \vee \bar{c}_R^1)$	$(\bar{c}_B^2 \vee \bar{c}_R^2)$	$(\bar{c}_B^3 \vee \bar{c}_R^3)$	$(\bar{c}_B^4 \vee \bar{c}_R^4)$
5		$(\bar{c}_R^1 \vee \bar{c}_R^2)$	$(\bar{c}_G^1 \vee \bar{c}_G^2)$	$(\bar{c}_B^1 \vee \bar{c}_B^2)$
6		$(\bar{c}_R^1 \vee \bar{c}_R^3)$	$(\bar{c}_G^1 \vee \bar{c}_G^3)$	$(\bar{c}_B^1 \vee \bar{c}_B^3)$
7		$(\bar{c}_R^1 \vee \bar{c}_R^4)$	$(\bar{c}_G^1 \vee \bar{c}_G^4)$	$(\bar{c}_B^1 \vee \bar{c}_B^4)$
8		$(\bar{c}_R^2 \vee \bar{c}_R^3)$	$(\bar{c}_G^2 \vee \bar{c}_G^3)$	$(\bar{c}_B^2 \vee \bar{c}_B^3)$
9		$(\bar{c}_R^3 \vee \bar{c}_R^4)$	$(\bar{c}_G^3 \vee \bar{c}_G^4)$	$(\bar{c}_B^3 \vee \bar{c}_B^4)$

Figure 1.6: *An instance of 3-colourability expressed in* SAT

5 – 9 express the fact that two vertices joined by an edge cannot have the same colour. The reader may verify for herself that assigning c_R^1, c_B^2, c_G^3 and c_B^4 true and the remaining variables false, we have a model.

Several extensions to the basic SAT language have been proposed with names such as equality, pseudo-boolean constraints and cardinality atoms (c-atoms), see e.g. [3, 25, 54]. Of interest to us are the c-atoms, which have been studied by several authors, e.g. [6, 25, 57, 69] (although Simons [69] does not use the name, he does use the concept.) A c-atom is an expression $k\{a_1, \ldots, a_n\}m$, where $a_1, \ldots, a_n$ are boolean variables and the expression is *true* iff at least k and no more than m of the a_i's are true. Revisiting the 3-colourability example using c-atoms, it may be formulated as in Figure 1.7. We see that the c-atoms come in handy. We also see that there is a close connection between these c-atoms and XSAT. As a matter of fact, each of these c-atoms can be seen as a clause in an XSAT instance — exactly one literal is true.

It is now time to present the next member of our family of problems. A natural extension of XSAT is the problem X$_i$SAT, asking if a formula allows an assignment to the variables such that exactly i literals are true in each clause. Note that an X$_i$SAT clause $(a_1 \vee a_2 \vee \ldots \vee a_n)$ is not just a special case of the c-atom $h\{a_1, a_2, \ldots, a_n\}i$, for some

1	$1\{c_R^1, c_G^1, c_B^1\}1$	$1\{c_R^2, c_G^2, c_B^2\}1$	$1\{c_R^3, c_G^3, c_B^3\}1$	$1\{c_R^4, c_G^4, c_B^4\}1$
5		$(\bar{c}_R^1 \vee \bar{c}_R^2)$	$(\bar{c}_G^1 \vee \bar{c}_G^2)$	$(\bar{c}_B^1 \vee \bar{c}_B^2)$
6		$(\bar{c}_R^1 \vee \bar{c}_R^3)$	$(\bar{c}_G^1 \vee \bar{c}_G^3)$	$(\bar{c}_B^1 \vee \bar{c}_B^3)$
7		$(\bar{c}_R^1 \vee \bar{c}_R^4)$	$(\bar{c}_G^1 \vee \bar{c}_G^4)$	$(\bar{c}_B^1 \vee \bar{c}_B^4)$
8		$(\bar{c}_R^2 \vee \bar{c}_R^3)$	$(\bar{c}_G^2 \vee \bar{c}_G^3)$	$(\bar{c}_B^2 \vee \bar{c}_B^3)$
9		$(\bar{c}_R^3 \vee \bar{c}_R^4)$	$(\bar{c}_G^3 \vee \bar{c}_G^4)$	$(\bar{c}_B^3 \vee \bar{c}_B^4)$

Figure 1.7: *The same 3-colourability instance expressed in* SAT *extended with c-atoms*

$h < i$. If we add the new variable b into the X$_i$SAT clause, then finding a satisfying assignment for the modified clause $(b \vee a_1 \vee a_2 \vee \ldots \vee a_n)$ is tantamount to finding a satisfying assignment for the c-atom $(i-1)\{a_1, a_2, \ldots, a_n\}i$. Repeating the procedure, we obtain a transformation that introduces $i - h$ new variables. Additionally, we see that X$_i$SAT well captures interesting properties of c-atoms. To the best of our knowledge, no one has tried to exactly solve instances of this extended SAT variant, without transforming the c-atoms into SAT clauses.

In [57] Liu and Truszczyński describe three possible ways to solve the extended SAT language; two are reductions to the basic SAT language. First a method that does not introduce any new variables but increases the number of clauses exponentially. They dismiss this technique:

> "This approach ... is practical only if k and m are small (do not exceed, say 2.) Otherwise the size ... quickly gets too large for SAT solvers to be effective."

They also present a second method that does not have this problem. However, the reduction more than doubles the number of variables. Rather than compiling away the c-atoms, Liu and Truszczyński investigate the possibility of keeping them and then applying an incomplete (i.e. randomised) search method. In effect, they deal with a formula having different kinds of clauses (ordinary SAT clauses as well as different kinds of c-atoms.)

We propose a fourth way which combines treats of the second and third approach: Compile the c-atoms into X_iSAT clauses (as described above) and then solve the instance exactly. As a motivating example, let us again consider the formula in Figure 1.7. As XSAT is solvable in time $\mathcal{O}(1.1730^n)$ (shown in Chapter 3) and SAT in $\mathcal{O}\left(2^{n-2\sqrt{n/\log n}}\right)$ time (see [21]), it seems like a waste to compile the c-atoms into SAT clauses. As will be seen below, also the X_iSAT problem seems to be nicer than SAT in terms of running times for known algorithms. To the best of our knowledge, there is no previous exact algorithm proposed to take advantage of the structure of the c-atoms, which, naïvely speaking, seems to be nicer than the structure of the SAT problem. We will not provide algorithms for solving formulae with both SAT and X_iSAT clauses as this is beyond the scope of this thesis, but our algorithms provide a foundation for such research.

Chapter 4 shows that X_iSAT is solvable in time $\mathcal{O}(1.4143^n)$, where n is the number of variables, if a $\mathcal{O}(1.1893^n)$ space consumption is allowed. This method is an application of a general algorithm for a special class of *NP*-complete problems described by Schroeppel and Shamir [68]. We previously mentioned this algorithm in section 1.2.1, since in the original paper XSAT was shown to belong to this class. Showing that the running time does not necessarily depend upon i i of theoretical interest, however, the use of exponential space is of course highly undesirable. For practical use, in the context of c-atoms, we present a branching algorithm with polynomial space requirements that obtains a running time in $\mathcal{O}\left(2^{(1-\varepsilon)n}\right)$, where the value of ε depends on i. For $i=2$, $i=3$ and $i=4$ we obtain upper time bounds in $\mathcal{O}(1.5157^n)$, $\mathcal{O}(1.6202^n)$ and $\mathcal{O}(1.6844^n)$ respectively. We also present a dedicated X_2SAT algorithm running in polynomial space and time $\mathcal{O}(1.4511^n)$.

1.2.3 Counting Exact Satisfiability

Most of the efforts in algorithm construction have been dedicated to algorithms for decision problems, i.e. finding *a* solution to the problem instance. This can involve finding an Euler walk in a graph or a

PSfrag replacements

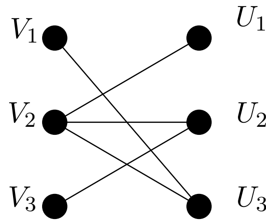

Figure 1.8: *A bipartite graph*

satisfying assignment to a Boolean formula. As a natural extension we have the counting problems, where one wants to not merely decide the existence of a solution, but to find the *number* of solutions. In the 1970's Valiant [74] proposed the counting complexity class $\#P$ for such problems. It is interesting to note that both NP-complete problems as well as some decision problems, known to be in P, can have a counting counterpart which is $\#P$-complete. For instance, 2SAT (SAT restricted to maximum clause length 2) is in P (see [23]), 3SAT is NP-complete (see [38]), but both #2SAT and #3SAT (the derived counting problems) are $\#P$-complete [52, 75]. The reader may note that, as a convention, counting problems have names starting with a '#'. For instance, the problem of counting models for SAT is denoted #SAT.

While algorithms for many NP-complete decision problems are fairly well-studied, this is not the case for their $\#P$-complete counterparts. One of the first algorithms for a counting problem came in the early 1960's with Ryser's [66] $\mathcal{O}(2^n)$ time algorithm for counting the number of perfect matchings in a bipartite graph, a problem which is equivalent to a matrix problem known as computing the *permanent* of a 0/1 matrix. This problem was proved complete for $\#P$ in 1979 by Valiant [74]. Some definitions are needed here: A *perfect matching* for a graph is a subset of the edges such that all vertices are covered ("touched" by an edge) exactly once. A graph is *bipartite* if one can partition its vertices into two sets such that within each set, no two vertices are joined by an edge. In Figure 1.8 a bipartite graph

is given. The set $\{(V_1, U_3), (V_2, U_1), (V_3, U_2)\}$ constitutes a perfect matching for this graph. We return to the problem of counting perfect matchings later in this section. For a long time Ryser's algorithm was the only exact algorithm for a $\#P$-complete problem. However, the interest in $\#P$-complete problems has increased recently. There have been a number of papers on counting problems and the class $\#P$, for instance Greenhill [41], Zhang [79], Dahllöf and Jonsson [17], Vadhan [73], Goldberg and Jerrum [40], and Valiant [75].

Counting models for Boolean formulae, mainly #SAT, has been studied by several authors, e.g. [7, 29, 46]. The problem of model counting is not only mathematically interesting, it has important applications within, for instance, artificial intelligence. Many AI reasoning tasks require counting the number of models or are reducible to this problem [7, 65].

When it comes to the problem of counting models for XSAT, #XSAT, it is interesting to note that the exactness property makes the problem similar to the problem of counting the number of perfect matchings in a graph. Consider the following reduction: Given a graph $G = (V, E)$, let each edge in E form a variable e_i. For each vertex in V make a clause v_j consisting of all e_i incident to it. The conjunction of these clauses gives a formula F_G. Any XSAT model of F_G corresponds to a perfect matching in G, since all vertices are covered, but only once, and so the number of models equals the number of perfect matchings. If we let $a = (V_1, U_3), b = (V_2, U_2), c = (V_2, U_2), d = (V_2, U_3)$ and $e = (V_3, U_2)$ then the problem of counting perfect matchings for the instance given in Figure 1.8 can be expressed as the #XSAT instance $(a), (b \vee c \vee d), (e), (b), (c \vee e), (a \vee d)$. We see that the answer is 1, as there is only one model, namely

$$a = true, b = true, c = false, d = false, e = true$$

The best algorithm so far for counting matchings in a bipartite graph is still the Ryser algorithm running in $\mathcal{O}(2^n)$ [66], where n is the number of *vertices*. Using the above reduction and our algorithm for #XSAT, we get a running time in $\mathcal{O}\left(1.2190^{|E|}\right)$, where $|E|$ is the

Formula:	$(a \vee b \vee c)$
Weights:	$w(a) = 1$ $w(\bar{a}) = 5$
	$w(b) = 2$ $w(\bar{b}) = 7$
	$w(c) = 2$ $w(\bar{c}) = 3$

Figure 1.9: *For this formula, $a = false, b = false, c = true$ is a maximum weighted x-model of weight 14*

number of *edges*. As the number of edges may be quadratic in the number of vertices, our algorithm is inferior in the general case. However, for graphs with a maximum degree of up to 7 (i.e. no vertex has more than seven edges) we have a better running time. As far as we know, this is the first algorithm for sparse graphs (a graph is sparse when its degree is bounded by some (small) constant.)

The first non-trivial upper time bound for solving #XSAT was presented in 2002 by Dahllöf and Jonsson [17]. By reduction to the #MAXIMUM INDEPENDENT SET problem an upper time bound of $\mathcal{O}(1.7548^n)$ was achieved. In 2004 Dahllöf and Jonsson [18] presented an exact algorithm for #XSAT with a running time in $\mathcal{O}(1.2190^n)$. In 2005 Porschen [62] presented a $\mathcal{O}(1.3247^n)$ time algorithm for #XSAT (or rather a variant where weights are associated to the variables and the number of minimum weighted models is wanted). The algorithm $\#D$ presented in Chapter 5 is essentially the same as in [18], however, we extend the algorithm to counting maximum weighted models. That means that there are non-negative weights associated to each *literal*, see Figure 1.9 for an example. For decision problems, the adding of weights often makes a problem harder, 2SAT, for instance is in P (see [23]), while the weighted version is *NP*-complete as we shall see. It is therefore interesting to see that for #XSAT, the adding of weights can be done without major modifications of the algorithm.

Our algorithm $\#D$ for #XSAT is not as fast as our XSAT algorithm, which is mainly due to the fact that XSAT can be solved in polynomial time when there are no heavy variables. This is unlikely to be the case for #XSAT, since it would allow us to solve #PERFECT MATCHING in polynomial time (because of the reduction shown

$$\begin{array}{rl} \text{Formula:} & (a \vee b), (c \vee \bar{b}) \\ \text{Weights:} & w(a) = 1 \; w(\bar{a}) = 1 \\ & w(b) = 1 \; w(\bar{b}) = 7 \\ & w(c) = 2 \; w(\bar{c}) = 3 \end{array}$$

Figure 1.10: *For this* 2SAT *formula,* $a = true, b = false, c = true$ *is a maximum weighted model of weight 10*

above — no heavy variables are introduced.) Furthermore, there are canonical rules that do not extend to counting, to the best of our knowledge. One such example is *resolution* which, in the context of XSAT, allows us to remove any variable a appearing both as a and $\bar{a}$ in an XSAT instance. Using clever choices of branching variables and the canonical rules we have at our disposal, we arrive at an $\mathcal{O}(1.2190^n)$ time bound for $\#D$. We also present a dedicated algorithm for #X3SAT running in $\mathcal{O}(1.1461^n)$ time, built along the same lines.

1.2.4 Counting 2SAT and Counting 3SAT

In this thesis we will also consider #2SAT and #3SAT, i.e. the problems of counting models for 2SAT and 3SAT formulae. At a first glance, they do not seem to belong to the XSAT family. However, it turns out that #2SAT is closely related to MAX X2SAT (to be presented in the next section) and thus well defends its place here; #3SAT is included as a generalisation of #2SAT.

Algorithms for #2SAT and #3SAT with better time bounds than the trivial $\mathcal{O}(2^n)$ bound have been presented by Dubois [29], Zhang [79], Littman et al. [56] and Dahllöf, Jonsson and Wahlström [19]. In Chapter 6, which extends the work in [17] and [19], and reproduces [20], we improve on the previously best running times for both these problems, with algorithms that count maximum weighted models. For an example see Figure 1.10. Considering weights of solutions is not only of theoretical interest in this context, but also opens the field for more applications, as many reductions from other problems become

possible, as seen for instance in [4] and later in Chapter 6.

For the problem of counting the number of maximum weighted models for a 2SAT formula, here referred to as $\#2\text{SAT}_w$, we present an algorithm with a running time in $\mathcal{O}(1.2561^n)$, significantly improving on the previously best bound for #2SAT of $\mathcal{O}(1.3247^n)$, achieved in [19]. There are several factors behind this improvement. One is a trick that among other things provides a way to remove variables which occur only once in a formula (i.e. in only one clause) in polynomial time. Another factor is our method of analysis, where we use a special measure of formula complexity combining the number of variables and the number of clauses into a single value which is more representative for formula complexity than the standard measure n, where n is the number of variables. In a technical report by Fomin et al. [34] about recent trends in the area of development of exponential algorithms, this kind of approach seen as a new promising technique. In the report they call it “Measure and Conquer” and they give examples of a number of *NP*-hard problems that benefit from this technique, among them the 3-COLOURABILITY PROBLEM and other graph problems.

In an unpublished paper by Fürer and Kasiviswanathan [35], our algorithm is re-analysed and the authors claim that they can prove an $\mathcal{O}(1.2461^n)$ time bound. In this paper, no new techniques are introduced, the analysis is just refined.

To say something about the decision problem corresponding to $\#2\text{SAT}_w$, we see that it is not 2SAT, but rather a weighted variant, 2SAT_w. We are not aware of any dedicated algorithms for this problem, but to get some idea of its hardness, one can note that it contains MAXIMUM INDEPENDENT SET as a special case. It is thus *NP*-hard.

For $\#3\text{SAT}_w$, our algorithm has a running time in $\mathcal{O}(1.6737^n)$, and the previously best result for #3SAT is $\mathcal{O}(1.6894^n)$, achieved in [19]. This improvement is mainly due to a more precise complexity analysis, where we use another measure of formula complexity to better capture the effects of having 2-clauses in the formula. This measure is then retranslated back to the useful measure n, the number of variables. As for the corresponding decision problem, we are not aware of any non-trivial worst-case time bounds for 3SAT_w or any related optimisation

problems, but one can note that the so far best exact polynomial space algorithm for 3SAT runs in $\mathcal{O}(1.4802^n)$ time. One can also mention that Khanna et al. [49] have obtained approximability results for a number of problems, two of which are similar to $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$, namely weighted MAX ONES 2SAT and weighted MAX ONES 3SAT. They are found to be poly-APX-hard, i.e. provided $NP \neq P$ there is no polynomial time algorithm that can return an answer guaranteed to be within a constant factor from the correct answer.

Note that there is a close connection between graphs and 2SAT formulae, since disregarding any negations, a clause of length two can be viewed as an edge in a graph, which is called the *constraint graph* of a 2SAT formula. In Chapter 6 we also present an algorithm that counts weighted models for 2SAT formulae having *separable constraint graphs*. Simply speaking, a graph is separable if one relatively easy can remove a subset of the vertices such that the graph falls apart in two or more components of about the same size and these components are not joined by any edges. While this class of formulae may sound exotic, we will present interesting graph applications. The separable graphs form a broad class, including many well-studied subclasses such as geometric graphs, graphs embeddable on surfaces of bounded genus, planar graphs, forests, grids, graphs with an excluded minor and graphs with bounded tree width. Counting in many of those classes still remains $\#P$-complete as shown by Vadhan [73]. Yet the restrictions can be used to obtain fast running times as we will show.

1.2.5 Maximum Exact Satisfiability

We have indicated the usefulness of X_iSAT and XSAT in the context of propositional logic modelling. This suggests that the problem of deciding the maximum number of simultaneously exactly satisfied clauses is of interest. For that problem, MAX XSAT, some results are already known. Madsen and Rossmanith [59] present results for two restricted variants of the problem but no algorithm for the MAX XSAT problem itself.

MAX XkSAT is MAX XSAT restricted to formulae of maximum clause length k. For $k = 2$ Madsen and Rossmanith present an

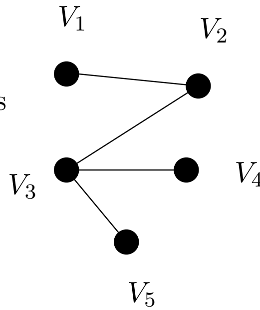

Figure 1.11: MAX CUT *is the problem of finding and removing a set of the vertices such that a maximum number of edges get "cut off," i.e. get exactly one endpoint removed. In this example,* $\{V_1, V_3\}$ *is such a set*

$\mathcal{O}\left(2^{m/4}\right)$ time algorithm, where m is the number of clauses. MAX X2SAT without unit clauses (clauses of length one) and variables that appear both negated and unnegated, is equivalent to the MAX CUT problem, see Figure 1.11 (the clauses correspond to edges.) There is an algorithm for MAX CUT by Fedin and Kulikov [33] with a running time in $\mathcal{O}\left(2^{|E|/4}\right)$ (where $|E|$ is the number of edges.) The algorithm of Madsen and Rossmanith is almost identical to the one of Fedin and Kulikov. It seems that the addition of unit clauses and negation contributes little or nothing to the complexity of the problem. This reminds us of the situation for XSAT, where unit clauses and negations can be dealt with using canonical rules. The currently best algorithm for MAX CUT, MAX 2SAT and MAX X2SAT by Kneis and Rossmanith [51] runs in time $\mathcal{O}\left(1.1421^K\right)$, where K is m or $|E|$ depending on the problem.

The second problem for which Madsen and Rossmanith present an algorithm is RESTRICTED MAX XSAT, which is MAX XSAT with the additional restriction that no clause may contain more than one true literal (i.e. clauses must be exactly satisfied or otherwise not satisfied at all.) Their algorithm runs in $\mathcal{O}(1.3248^n)$ time, which we improve by reduction to $\#2\text{SAT}_w$ to $\mathcal{O}(1.2561^n)$ time.

MAX XSAT can of course be solved within $\mathcal{O}(2^n)$ time, testing all

assignments. No better upper time bound is known and in Chapter 7 we show that for every fixed k, Max x$k$sat is at least as hard as Max $k$sat, i.e. the corresponding Sat problem. Given the lack of results for Max sat, we get an indication of the hardness of Max xsat in terms of upper time bounds. We will make more comparisons between Max xsat and Max sat in Chapter 7.

1.2.6 Max Hamming Exact Satisfiability

Most previous algorithms for optimisation problems have contented themselves with producing *one* best or good-enough solution. However, there is often an actual need for *several solutions* that are as different as possible. The notion of "as different as possible" can be captured using the concept of Hamming distance. The Hamming distance between two assignments to a set of variables is the number of variables on which they disagree. Somewhat surprisingly, the maximum Hamming distance problems have only recently become an area of academic research. The first paper (to the best of our knowledge) by Crescenzi and Rossi [14] came in 2002. In their paper they present some results on the hardness of approximating solutions to various Boolean problems. Angelsmark and Thapper [5] present exact and randomised algorithms for the general finite domain problem (i.e. not only Boolean problems) as well as dedicated algorithms for Max hamming sat. Hebrard et al. [42] consider a broader range of problems, including finding solutions that are similar. They also test some heuristic methods. The so far best exact algorithm for Max hamming sat by Angelsmark and Thapper [5] runs in $\mathcal{O}(4^n)$ time (where n is the number of variables) and polynomial space.

When it comes to other Boolean problems of interest, one can note that Max hamming nae sat (where a model is an assignment such that every clause is satisfied, but there is at least one false literal in each clause) is trivial if one model is known. Due to the nature of the problem, every model has a symmetrical twin where the opposite assignments are made. Hence, if the instance is satisfiable, there are always two models at Hamming distance n and so the problem is solvable in time $\mathcal{O}(2^n)$.

Exact algorithms for Max hamming xsat have not been considered before, and as the problem is not efficiently approximable (see [14]), exact algorithms are not only of theoretical interest.

We will present two polynomial space algorithms P and Q for Max hamming xsat and prove that they run in $\mathcal{O}(2^n)$ and $\mathcal{O}(1.8348^n)$ time respectively. Previous algorithms for maximum Hamming problems have relied on an external solver for the base problem. P is also such an algorithm. However, there is a novelty: By using a polynomial time test, many unnecessary calls to the solver can be avoided. Thereby the running time is improved substantially. Q represents something totally new in this area, because it works directly on the inherent structure of the Max hamming xsat problem. More precisely, a new kind of dpll branching is introduced. Though the running time of P is slightly inferior to the running time of Q, there are good reasons to present both algorithms: P resembles previous algorithms and gives a hint on how they can be improved, and it is easy to implement given an external Xsat solver. Furthermore, if one is content with getting two models that have at least the Hamming distance d, for some constant d, then P can be easily modified to have a provably better upper time bound than Q. Apart from the immediate interest of the Max hamming xsat problem itself, we hope that the ideas presented here will also be applicable for other problems such as Max hamming scheduling, Max hamming mis and the like.

1.3 Summary of Results

We here summarise the main result of the thesis:

- For Xsat a polynomial space algorithm running in $\mathcal{O}(1.1730^n)$ time is presented.
- We show that $X_i$sat is solvable in time $\mathcal{O}(1.4143^n)$ (where n is the number of variables) regardless of i if exponential space is allowed. For polynomial space, we present an algorithm which solves $X_i$sat for all i strictly better than the trivial $\mathcal{O}(2^n)$

bound. For $i = 2, 3$ and 4 we obtain upper time bounds in $\mathcal{O}(1.5157^n)$, $\mathcal{O}(1.6202^n)$ and $\mathcal{O}(1.6844^n)$ respectively. We also present a dedicated X_2SAT algorithm running in polynomial space and time $\mathcal{O}(1.4511^n)$.

– For #XSAT we present an $\mathcal{O}(1.2190^n)$ time algorithm. We also present an $\mathcal{O}(1.1487^n)$ time algorithm for #X3SAT .

– We present algorithms for counting maximum weighted models for 2SAT and 3SAT formulae. They use polynomial space and run in time $\mathcal{O}(1.2561^n)$ and $\mathcal{O}(1.6737^n)$ respectively, where n is the number of variables. We also provide an algorithm for the restricted case of separable 2SAT formulae, with fast running times for well-studied input classes.

– Regarding MAX XSAT, we show that for every fixed k there is a polynomial reduction from MAX kSAT to MAX XkSAT that does not increase the instance in terms of variables. It is thus reasonable to argue that MAX XSAT is at least as hard as MAX SAT. By reduction to #2SAT$_w$ we also show that RESTRICTED MAX XSAT is solvable in time $\mathcal{O}(1.2561^n)$.

– We present two exact algorithms for MAX HAMMING XSAT, i.e. the problem of finding two XSAT models at maximum Hamming distance. Using the XSAT solver of chapter 3 as an auxiliary function, an $\mathcal{O}(2^n)$ time algorithm can be constructed, where n is the number of variables. This upper time bound can be further improved to $\mathcal{O}(1.8348^n)$ by introducing a new kind of branching, more directly suited for finding models at maximum Hamming distance.

1.4 Outline of the Thesis

After this first introductory chapter, Chapter 2 deals with preliminaries, notation and problem definitions. We also relate a well known method for estimating the worst case running time of branching algorithms. The special requirements for how a clause gets satisfied yield

a great deal of possibilities to reduce a formula in polynomial time, when the formula contains certain structures. This is also elaborated upon in this chapter. The special case where no variable occurs more than twice in a formula is also known to be polynomial time decidable. For the sake of completeness a matching algorithm to that end is given.

After the preliminaries and polynomial time procedures we treat the various members of our family of problems. The thesis is divided into three major parts given by the complexity classes of the problems.

The first part deals with the *NP*-complete decision problems XSAT and X_iSAT. In the second part counting problems are considered. The third part is devoted to optimisation problems. In each chapter we present exact algorithms and/or reductions to obtain interesting upper time bounds and hardness results. Finally we discuss the results and give possible future research directions for our family of problems and *NP*-hard problems in general.

Chapter 2

Preliminaries

In this chapter we give the notation and definitions used in the thesis. We also relate known results for estimating worst case running times for branching algorithms; furthermore, canonical rules are given and a matching technique for solving XSAT instances when there are no heavy variables is presented.

2.1 Formula Related Concepts

A *Boolean variable* (or *variable* for short) has either the value $true$ or $false$. A *literal* b is either a variable a or the negation of a variable $\bar{a}$; we say that the literal b is *derived* from the variable a. For a literal b, $Var(b)$ is the variable from which it is derived.

The literal a is $true$ iff $Var(a)$ is $true$ and the literal $\bar{a}$ is $true$ iff $Var(a)$ is $false$. When *flipping* a literal a ($\bar{a}$) one gets $\bar{a}$ (a.) A *clause* is a multiset of literals and the constants $true$ and $false$. The *length* of a clause x, denoted $|x|$, is its cardinality. The clause with length 0 is called the *empty clause*. An n-clause has length n. As a convention, a clause is presented with its members separated by logical or ($\vee$.) We sometimes need a subclause notation in this way: $(a \vee b \vee C)$, such that C is not a literal or constant, but rather summarises several members of x. For instance, if $C = c \vee true$ then $(a \vee b \vee C) = (a \vee b \vee c \vee true)$. As a convention, literals are indicated

by lower-case letters and subclauses by upper-case letters.

For a clause x, $Var(x)$ is the set of variables from which the literals of x are derived. Two clauses x and y are said to *connect* if $|Var(x) \cap Var(y)| > 0$; if $|Var(x) \cap Var(y)| > 1$ we say that x and y *overlap*. We say that a variable a occurs in a clause x if $a \in Var(x)$.

A *formula* is a set of clauses. As a convention, a formula is presented with its members separated by logical and ($\wedge$.) The formula F such that $|F| = 0$ is called the *empty formula*. For a formula F, *Var(F)* is $\cup_{x \in F} Var(x)$.

Let b and $\bar{b}$ be the two literals derived from the variable a; a is said to be *constant* (in the formula F) if only b or only $\bar{b}$ occurs in clauses of F. The *degree* of the variable c in a formula F, denoted $\delta_F(c)$, is the number of clauses that contain at least one literal derived from c. We omit the subscript F from $\delta_F(c)$ when there is no risk of confusion. If $\delta(c) = 1$ we call c a *singleton*. If $\delta(c) \geq 3$ we say that c is *heavy*. $\delta_k(c)$ indicates the number of clauses of length k that contains literals derived from c. The maximum degree of any variable in F is denoted $\delta(F)$.

To measure the complexity of a formula, we introduce the following: $n(F) = |Var(F)|$ and $n_d(F)$ denotes the number of variables of degree d; $m(F)$ is the number of clauses in F; $l(F)$ denotes the length of F, i.e. $l(F) = \sum_{x \in F} |x|$.

A *satisfying assignment* or *model* is an assignment to every variable of a formula such that there is at least one true member in each clause. Note that the constant *true* is considered true under any assignment; the constant $false$ is false under every assignment. An *extendible assignment* is an assignment to a subset of the variables, such that no clause becomes false. Note that an empty formula has one model and a formula containing the empty clause has no model.

An *x-model* is an assignment to the variables of a formula F such that there is exactly one true member in every clause. The member that exactly satisfies a clause is called a *satisfactor*. Given a formula F, an assignment to the variables is *inconsistent* if any clause has no true literal (it is *unsatisfied*) or more than one true literals (it is *over-satisfied*.)

The *Hamming distance* between two assignments is the number of assignments to the individual variables that disagree.

Let a be a literal and α a literal or one of the constants $true$ or $false$, then $F(a/\alpha)$ denotes *substitution* of every occurrence of a by α in the formula F. It is convenient to drop the restriction that both a and α are a single element, and thus we also have the following two variants:

1. Let a be a literal and C a multiset of literals, then $F(a/C)$ denotes substitution of every occurrence of a by C in the formula F.

2. Let C be a multiset of literals and β one of the constants $true$ or $false$, then $F(C/\beta)$ denotes substitution of every occurrence of a member of C by β in the formula F.

The notation $F(a/\delta; b/\gamma)$ indicates repeated substitution: $F(a/\delta)(b/\gamma)$ (first a is replaced and then b.)

For a given multiset of literals $B = (a \vee b \vee \bar{c} \vee \ldots)$, the multiset $\bar{B}$ is the result of flipping all literals of B, i.e. $\bar{B} = (\bar{a} \vee \bar{b} \vee c \vee \ldots)$.

2.2 Graph Related Concepts

A *graph* $G = (V, E)$ is an ordered pair consisting of a finite (possibly empty) set V of *vertices* and a set E of unordered pairs (u, v) of distinct vertices, called *edges*. The size of a graph G, denoted $|G|$, is its number of vertices. A set S of vertices is *independent* if $(u, v) \notin E$ for all $u, v \in S$.

We now define the *constraint graph* of a formula F as the graph where the vertex set is the variables of F and the set of edges is

$$\{(a, b) \mid \text{the set } (a, b) \text{ is a subset of } Var(x) \text{ for some } x \in F\}.$$

This graph is also known as a *Gaifman graph*, see [37].

The *neighbourhood* of a vertex x in a graph G, denoted $N_G(x)$, is the set of vertices having an edge in common with x. The neighbourhood of a vertex set X is the union of the neighbourhoods of the vertices of X. The neighbourhood of a variable in a formula F is defined to be the corresponding neighbourhood in the constraint graph. The *extended size* of the neighbourhood of x, $S(x)$, we will measure as $S(x) = \delta(x) + \sum_{y \in N(x)} \delta(y)$.

A *path* $x_0, \ldots, x_k$ is a sequence of vertices such that each x_i has an edge to x_{i+1}. We say that a formula is *connected* iff in the corresponding constraint graph, there is a path from each variable to every other. Otherwise, the formula consists of *connected components* and within each connected component this path-condition holds. The components can be found in polynomial time by breadth-first search.

A graph is said to be *n-regular* if every vertex has degree n.

2.3 Problem Definitions

We are now able to give more precise problem definitions. Most of the problems defined here belong to the XSAT family, while the others are of interest due to kinship, either by reduction or from similarities in structure.

- MAXIMUM INDEPENDENT SET (MIS):

  Instance: A graph G.

  Question: What is the size of the largest subset I of the vertices of G such that no pair of the vertices of I is joined by an edge?

- SATISFIABILITY (SAT):

  Instance: A formula F.

  Question: Is it possible to assign values to the variables of F such that each clause contains at least one true literal?

- k-SATISFIABILITY (kSAT):

  Instance: A formula F such that no clause contains more than k literals.

Question: Is it possible to assign values to the variables of F such that each clause contains at least one true literal?

– EXACT SATISFIABILITY (XSAT):

Instance: A formula F.

Question: Is it possible to assign values to the variables of F such that each clause contains exactly one true literal?

– GENERAL EXACT SATISFIABILITY (X_iSAT):

Instance: A formula F.

Question: Is it possible to assign values to the variables of F such that each clause contains exactly i true literals?

The languages XSAT, X_2SAT, X_3SAT *etc.* are referred to as *sublanguages* of X_iSAT.

– COUNTING EXACT SATISFIABILITY (#XSAT):

Instance: A formula F.

Question: What is the number of x-models for F?

– COUNTING WEIGHTED EXACT SATISFIABILITY ($\#\text{XSAT}_w$):

Instance: A formula F with weights associated to every literal.

Question: What is the number of maximum weight x-models for F?

– COUNTING SATISFIABILITY (#SAT):

Instance: A formula F.

Question: What is the number of models for F?

– COUNTING WEIGHTED SATISFIABILITY ($\#\text{SAT}_w$):

Instance: A formula F with weights associated to every literal.

Question: What is the number of maximum weight models for F?

- MAXIMUM SATISFIABILITY (MAX SAT):

  Instance: A formula F.

  Question: What is the maximum number of simultaneously satisfiable clauses of F?

- MAXIMUM EXACT SATISFIABILITY (MAX XSAT):

  Instance: A formula F.

  Question: What is the maximum number of simultaneously x-satisfiable clauses of F?

- RESTRICTED MAXIMUM EXACT SATISFIABILITY (RESTRICTED MAX XSAT):

  Instance: A formula F.

  Question: Under the additional restriction that no clause must be over-satisfied, what is the maximum number of simultaneously x-satisfiable clauses of F?

- MAXIMUM HAMMING EXACT SATISFIABILITY (MAX HAMMING XSAT):

  Instance: A formula F.

  Question: What is the maximum Hamming distance between any two x-models of F?

2.4 Branching Algorithms and Estimations of their Running Times

A common approach to exactly solve Boolean problems is to use so called DPLL style branching [22]. In its basic form, one variable a is selected from the formula F and the problem of finding a model is reduced to the problem of finding a model for either $F_1 = F(a/true)$ or $F_2 = F(a/false)$; we say that we *branch* on a. A note here: strictly speaking, we need to construct F_1 and F_2 as $F_1 = F(a/true; \bar{a}/false)$ and $F_2 = F(a/false; \bar{a}/true)$, but for convenience we use the shorter form.

The basic DPLL style branching can be generalised so that, given an instance F, we branch on one or more variables, i.e. we assign values to the variable(s) such that the problem for F is reduced to the problem for two or more formulae $F_1, \ldots, F_k$ with fewer variables. Most of our algorithms to be presented are based on this principle. By the nature of this approach, the running times will, in the worst case, be exponential in the number of variables. In order to obtain non-trivial worst case running times we want each F_i to shrink as much as possible.

In the complexity analysis of our algorithms, we will often dismiss some cases as 'easy' and not candidates for being the worst case. One such easy case is when the instance is not connected: Assume that we are using an exponential time algorithm $\beta(F)$ with a running time in $\mathcal{O}\left(c^{|F|}\right)$ for some measure $|F|$ of the size of the formula F. Furthermore, assume that the constraint graph of F falls apart into two components F_1 and F_2 such that solving F_1 and F_2 separately, their solutions can in polynomial time be combined to a solution for F. We then get a running time in $\mathcal{O}\left(c^{|F_1|} + c^{|F_2|}\right)$ which is in $\mathcal{O}\left(c^{|F|}\right)$, i.e. the running time will never be worse if we have such a situation. This of course also holds if we have more than two components.

In what follows, when presenting upper time bounds for exponential time algorithms, any polynomial factor will be suppressed. Furthermore, when giving time bounds in terms of the number of variables, polynomial factors in the length of the formula will be suppressed.

For the analysis of most of our algorithms, a method by Kullmann will be used [53]. Consider the branching tree that the algorithm (implicitly) constructs when applied to a problem instance. If a node v in the tree has d branches, which are labelled with real, positive numbers $t_1, \ldots, t_d$ (think of these labels as measures of the reduction of complexity in the respective branch), then the *branching tuple* for v is $(t_1, \ldots, t_d)$ and the *branching number* is the positive real-valued solution of

$$\sum_{i=1}^{d} x^{-t_i} = 1.$$

The branching number of a branching tuple B is denoted by $\tau(B)$. A branching tuple $(t_1, \ldots, t_d)$ is said to *dominate* another branching tuple $(u_1, \ldots, u_d)$ if $t_i \le u_i$ for all $1 \le i \le d$, ensuring that $\tau(t_1, \ldots, t_d) \ge \tau(u_1, \ldots, u_d)$.

In most of our analyses, the labels of the branches will be the decrease in the number of variables, i.e. for a branch from the formula F to the formula F', the label will be $t_i = n(F) - n(F')$. In this case, we will have a running time in $\mathcal{O}(\alpha^n)$ such that α is the largest branching number found in the branching tree. In Chapter 6 we encounter other ways to measure formula complexity.

Let $R = \sum_{i=1}^{k} r_i$ and note that due to the nature of the function $f(x) = 1 - \sum_{i=1}^{k} x^{-r_i}$, the smallest possible real-valued root will appear when each r_i is as close to R/k as possible, i.e. when the decrease of size of the instance is balanced through the branches. Say for instance that $R = 4, k = 2$. Then $\tau(1,3) = \tau(3,1) \approx 1.4656$ and $\tau(2,2) \approx 1.4142$. We will refer to this as *the balanced branching effect.* We will use the shorthand notation $\tau(r^k, \ldots)$ for $\tau(\underbrace{r, r \ldots r}_{k}, \ldots)$, e.g. $\tau(5^2, 3^3)$ for $\tau(5,5,3,3,3)$.

2.5 Canonisation

In [27] Drori and Peleg introduced the name *canonical instance* for an instance of XSAT that cannot be simplified by a given set of polynomial time pruning rules. The concept comes in handy when reasoning about the time complexity — assuming that the instance is canonical we can limit the number of possible cases to analyse. Foremost, however, it is used in the algorithms where a major goal is to create non-canonical instances that can be pruned in polynomial time.

Canonical instances enjoy certain properties that we will list. Along with the properties we describe transformations (rules) that guarantee they are true. Although some rules may create violations of other properties, we can continue transforming until all properties hold. This process is guaranteed to terminate because each transformation shrinks the formula F.

Many of these transformations can be applied to other members of our family of problems if extra structures are used, as we shall see. Let F be a formula, F is said to be canonical if:

1. **For every constant variable, only the positive literal appears in the clauses.**

   If there is a variable a that appears only negatively, then let $F \leftarrow F(a/\bar{a})$.

2. **No clause contains the constants true or false.**

   If F contains a clause $x = (true \vee C)$ (where C is a multiset), then remove x and let $F \leftarrow F(C/false)$

   If F contains a clause $x = (false \vee C)$, then let $x \leftarrow C$.

3. **There are no 1-clauses.**

   If F contains the clause (a), then let $F \leftarrow F(a/true)$.

4. **There are no 2-clauses.**

   If F contains the clause $(a \vee b)$, then let $F \leftarrow F(a/\bar{b})$.

5. **No clause contains the same variable more than once.**

   If F contains the clause $(a \vee a \vee C)$, then let $F \leftarrow F(a/false)$.

   If F contains the clause $(a \vee \bar{a} \vee C)$, then let $F \leftarrow F(C/false)$ and remove the clause.

6. **There is no constant variable a such that**

   (a) a **occurs only in clauses with singletons or**

   (b) a **occurs always with another constant variable b and singletons, such that b always appears in clauses with a.**

   If F contains such a variable a, then let $F \leftarrow F(a/false)$.

7. **If two variables a and b both appear in two clauses x and y, then a and b have the same sign in x and in y.**

   If $x = (a \vee b \vee C)$ and $y = (\bar{a} \vee b \vee D)$, then let $F \leftarrow F(b/false)$

   If $x = (a \vee b \vee C)$ and $y = (\bar{a} \vee \bar{b} \vee D)$, then let $F \leftarrow F(a/\bar{b})$

8. **All variables are constant.**

   If there is a variable a such that there are two clauses $x = (a \vee C_1)$ and $y = (\bar{a} \vee C_2)$, then

   (a) Replace every clause $(a \vee C')$ by $(C' \vee C_2)$
   (b) Replace every clause $(\bar{a} \vee C'')$ by $(C'' \vee C_1)$
   (c) For every literal $b \in C_1 \cap C_2$ let $F \leftarrow F(b/false)$

   This rule is called *resolution*.

9. **Any r-clause and any s-clause have at most $r - 2$ variables in common.**

   If there are two clauses $x = (A \vee b)$ and $y = (A \vee B)$, then let $x \leftarrow (A \vee b)$ and $y \leftarrow (\bar{b} \vee B)$.

10. **If there are clauses $(a \vee b_1 \vee C_1)$, $(a \vee b_2 \vee C_2)$, ..., $(a \vee b_k \vee C_k)$ (not necessarily distinct), then $x = (b_1 \vee \cdots \vee b_k)$ is not a clause in F.**

    If such clauses exist, then let $F \leftarrow F(a/false)$.

11. **There is no pair of clauses x, y such that $x \subset y$.**

    If such clauses exist, then let $F \leftarrow F(a/false)$ for each $a \in y, a \notin x$ and remove x.

12. **There are no three clauses $x = (a \vee b \vee c \vee e)$, $y = (a \vee b \vee C_1)$ and $z = (a \vee c \vee C_2)$ such that $\delta(a) = 3$, e is a singleton and $\delta(c) = \delta(b) = 2$**

    If the above configuration exists, then replace x, y and z by $(b \vee C_1)$ and $(c \vee C_2)$

Lemma 1. The process of applying the above transformations on a formula F terminates in polynomial time, measuring in $n(F)$.

Proof. All transformations but the first two run in polynomial time and are guaranteed to remove variables. When transformation 2 does not remove any variables it takes constant time (remember that when measuring in n we disregard any polynomial factors in the length of the formula) and the same holds for transformation 1. Then note that the first transformation can be applied at most n times after any transformation that removes variables, and that the non-removing variant of the second transformation can be applied at most once. Hence they contribute only a polynomial factor to the total running time.

Note that resolution as formulated here may increase the length of the formula, but only with a constant factor. □

Lemma 2. Every transformation above preserves the x-satisfiability of a formula F.

Proof. For most of the transformations the correctness is easily seen, however, some transformations may need further clarification:

- Transformation 6: In a model where a is true, b and the singletons will be false. This implies that removing a, the formula is still x-satisfiable (by b and singletons.) Similarly, a model where a is false will still be a model for the formula with a removed.
- Transformation 9: In a model where b is true all of A must be false, and one in B true. This is still the case after the transformation. Similarly for the case $b = false$.
- Transformation 10: If a is true, x cannot be satisfied.
- Transformation 12: The configuration in the boldface allows $C_1 = C_2$ =false, $C_1 = C_2 =$ true, $C_1 =$ false when $C_2 =$ true and $C_1 =$ true when $C_2 =$ false. This is still the case after the transformation.

□

Algorithm $MatchDecide(F)$

1. If there are any non-constant variables, apply resolution.
2. Let each clause form a vertex and add an edge between every two clauses having a variable in common. This forms the graph $G_F = (V, E)$.
3. Let $S \subseteq V$ contain the clauses having no singleton variable. Let the weight of an edge e be the number of endpoints it has that belong in S (zero, one, or two.)
4. Find a maximum weighted matching in G. If that weight is equal to $|S|$, then return 'Yes' otherwise 'No'. Note that the empty graph will produce 'Yes'.

Figure 2.1: *An algorithm for deciding* XSAT *when there are no heavy variables*

2.6 Reduction to Matching

In this section we will present a polynomial time algorithm for deciding the existence of models for instances of $X_1$3SAT and XSAT where all variables have degree at most 2, i.e. there are no heavy variables. Such an instance, F, will be transformed into a graph G_F, and a *maximum weighted matching* of a certain weight found in G_F corresponds to a model of the instance. Some more graph definitions are necessary here: For an edge $e = (u, v)$, u and v are called the *endpoints* of e. A matching for a graph $G = (V, E)$ is a subset of edges without common endpoints, not necessarily covering all vertices. Assuming each edge has a non-negative weight, a maximum weighted matching is a matching such that no other matching has higher weight. A maximum weighted matching is computable in polynomial time [36]. This transformation technique was first presented by Porschen et al. in [63] for use in their algorithm for $X_1$3SAT. We here give our own, slightly different, proof for the sake of completeness.

Lemma 3. For an instance F of XSAT such that all variables have at most degree 2, $MatchDecide(F)$ will in polynomial time return 'Yes' iff F is satisfiable and 'No' otherwise.

Proof. The first line ensures that all the variables of the instance are constant.

To see that the maximum weighted matching found corresponds to a model: For the clauses in S, make *true* the literals pointed out by the edges of the perfect matching and all other literals $false$. For the clauses of $V - S$, each clause covered by an edge assigns *true* to the literal pointed out and $false$ to the other literals of that clause. For the uncovered clauses: Assign *true* to the singleton and $false$ to the other variables. This is a model since:

1. Each clause is satisfied.

2. No clause is over-satisfied. Each vertex/clause is covered by at most one edge in the matching. Further, since each variable of the instance is constant, making a literal $false$ in any clause will never cause over-satisfaction.

The algorithm runs in polynomial time since the number of vertices is a polynomial in n.

Note that an empty clause makes the algorithm produce 'No', and that the empty set of edges is a maximum weighted matching of weight $|S|$ for a graph without vertices, i.e. an empty formula has one model. □

Part II

Deciding

Chapter 3

Exact Satisfiability

In this chapter we encounter the archetype member of our family of problems. We will here present an exact, polynomial space algorithm, prove it correct and give an upper bound on its running time.

3.1 A Polynomial Space Exact Algorithm

A basic idea in our algorithm for XSAT is that when the ratio heavy variables to the total number of variables is small enough (i.e. the formula is sparse) we need only test all assignments to the heavy variables, because for each such partial assignment it is possible to use Lemma 3 to decide the x-satisfiability of the whole instance. The procedure of testing all possible assignments to the heavy variables and then deciding the instance using matching techniques, is referred to as *test-and-match*.

The algorithm to be presented here improves upon the previous ones by exploiting more thoroughly the concept of sparsity. The previous algorithms by Byskov et al. [10] and Dahllöf et al. [18] consist of a series of cases. In the last case there is just one possible configuration for heavy variables left and then test-and-match is used. In Figure 3.1 the configuration left by Dahllöf et al. is shown. We see that if all heavy variables occur in this way, the ratio heavy variables to all variables is at most 2 to 11. This is because in the worst case, all the

$$(a \vee b \vee c \vee d)$$
$$(a \vee e \vee f \vee g)$$
$$(a \vee h \vee i \vee j)$$

Figure 3.1: *Part of an* XSAT *instance.* $\delta(a) = 3$ *and the degree of the other variables is at most 2*

non-heavy variables have degree two, and every non-heavy variable occurs in clauses with heavy variables. This gives a running time in $\mathcal{O}\left(2^{2n/11}\right) \subseteq \mathcal{O}\left(1.1344^n\right)$, using test-and-match. Naïvely branching on a we would have a running time in $\mathcal{O}\left(\tau(10,1)^n\right) \subseteq \mathcal{O}\left(1.1975^n\right)$ (in the branch where a is true all variables are removed, in the other branch only a.) We thus see that the test-and-match technique allows us to avoid this branching.

The algorithm presented here leaves several configurations of heavy variables and so many disadvantageous branchings can be avoided. In the complexity analysis it is shown that the ratio $n_3(F)/n(F)$ (the only heavy variables left at the end are variables of degree three) is still small enough with respect to the upper time bound we want to establish.

The overall strategy of the algorithm DX for XSAT is to ensure a certain sparsity so that the test-and-match technique can be efficiently applied, at the same time avoiding disadvantageous branchings. In particular, by DPLL style branching we remove various kinds of overlapping clauses and other configurations that would allow too high a ratio $n_3(F)/n(F)$. In so doing we make a branching that in the worst case removes one variable in one branch and 12 variables in the other. That will decide the overall worst case running time of $\mathcal{O}\left(1.1730^n\right)$.

The algorithm DX can be seen as a switch statement: Given the instance F, apply the first line that is applicable. As for the canonisation, we will assume that there is a line 0 that applies all the rules of Chapter 2. This ensures that each of the following lines is applied to a canonical instance.

For the sake of convenience we will denote sets of variables by their extremal members, for instance, "$b - e$" is short for b, c, d, e.

Algorithm $DX(F)$

1. Pick a variable a such that $|N(a)| \geq 11$; **return** $DX(F(a/true))$ **or** $DX(F(a/false))$

2. For two clauses $x = (C_1 \vee C)$ and $y = (C \vee C_2)$ such that $|C| \geq 3$ or such that $|C| = 2$, in the latter case with these restrictions: If $|x| = 5$, then $|y|$ cannot be 5 or 6 and if $x = (a \vee b \vee c \vee d \vee e), y = (a \vee b \vee f \vee g)$ and $\delta(b) = 2$, then y must contain no singleton; **return** $DX(F \wedge C)$ **or** $DX(F(C/false))$.

3. For two clauses $x = (a \vee b \vee C_1)$ and $y = (a \vee b \vee C_2)$ such that a occurs in a 3-clause; **return** $DX(F(a/true))$ **or** $DX(F(a/false))$

4. For three clauses $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee C_1)$ and $z =(a \vee c \vee C_2)$ such that d and e are not singletons; **return** $DX(F \wedge (d \vee e))$ **or** $DX(F(d/false; e/false))$

5. For three clauses $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee C_1)$ and $z = (a \vee c \vee C_2)$ such that e is a singleton and $\delta(b) = \delta(c) = 2$; **return** $DX(F(d/true))$ **or** $DX(F(d/false))$

6. For three clauses $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g \vee h \vee i)$ and $z = (a \vee c \vee f \vee j \vee k)$; **return** $DX(F \wedge (a \vee b))$ **or** $DX(F(a/false; b/false))$

7. Pick a heavy variable a such that a does not occur in three 4-clauses, two 4-clauses and a 5-clause, or two overlapping clauses; **return** $DX(F(a/true))$ **or** $DX(F(a/false))$

8. (a) For a clause $x = (a \vee b \vee c \vee d)$ such that no member of x is a singleton and a and b are heavy variables.

   (b) or a clause $y = (a \vee b \vee c \vee d)$ such that $\delta(a) = 3$ and $|N(b)| = 10$

   (c) or a clause $z = (a \vee b \vee c \vee d)$ such that no member of z is a singleton, $|N(a)| = 10$ and $|N(b)| \geq 7$

(d) or a clause $w = (a \vee b \vee c \vee d)$ such that no member of w is a singleton, $\delta(a) = \delta(b) = 2$ and a and b both appear in a clause with a singleton;

return $DX(F(a/true))$ **or** $DX(F(b/true))$ **or** $DX(F(a/false; b/false))$

9. For four clauses $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g \vee h)$, $z = (a \vee C_1)$ and $w = (b \vee C_2)$ such that some of $c - h$ are heavy or appears in a 4-clause $w' = (\alpha \vee C_3)$ such that α is heavy; **return** $DX(F \wedge (a \vee b))$ **or** $DX(F(a/false; b/false))$

10. For two clauses $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g)$ such that some of $c - e$ are heavy or appears in a 4-clause $w' = (\alpha \vee C_3)$ such that α is heavy; **return** $DX(F \wedge (a \vee b)))$ **or** $DX(F(a/false; b/false))$

11. For two clauses $x = (a \vee b \vee c \vee d)$ and $y = (b \vee e \vee f \vee g)$ such that a is heavy and no member of x or y is a singleton; **return** $DX(F \wedge (b \vee e))$ **or** $DX(F(b/false; e/false))$

12. For two clauses $x = (a \vee b \vee c \vee d)$ and $y = (c \vee e \vee f \vee g)$ such that a and e are heavy and such that no member of x is a singleton; **return** $DX(F(a/true))$ **or** $DX(F(b/true))$ **or** $DX(F(a/false; b/false))$

13. Test all possible assignments to the remaining heavy variables. For each such partial assignment to the variables, apply the canonical rule 2 and then use Lemma 3 to decide whether there is a model. Return 'True' if such a model is found and 'False' otherwise.

Theorem 4. $DX(F)$ decides XSAT.

Proof. We proceed by inspecting the lines of DX. The invisible line 0 with all the canonical rules is correct by Lemma 2. The correctness of lines 1 – 12 is obvious. Line 13 is correct by Lemma 3. □

We now consider the worst case time complexity of DX.

Theorem 5. $DX(F)$ runs in time $\mathcal{O}(1.1730^n)$ where $n = n(F)$.

Proof. The time complexity analysis consists of a number of case and subcase analyses. Typically the analysis of a case m will establish an upper time bound U_α "for this case" which should be interpreted: If throughout the whole execution of the algorithm, α is the only case applicable, then U_α is an upper bound of the execution time. Hence one can easily see that an overall upper time bound for the algorithm is the maximum U_j established for all cases j.

1. This case clearly runs in time $\mathcal{O}(\tau(1,12)^n) \subseteq \mathcal{O}(1.1730^n)$.

2. By the canonical rules 9 and 8, $|C_1|$, $|C_2| \geq 2$. If $|x| = |y| = 4$ we get a running time in $\mathcal{O}(\tau(5,4)^n) \subseteq \mathcal{O}(1.1674^n)$; if $|x| = 5$ and $|y| = 4$ we get a running time in $\mathcal{O}(\tau(7,3)^n) \subseteq \mathcal{O}(1.1586^n)$ (if a and b occur together in another clause at least one more variable will be removed in the first call; if they occur in different clauses both will be removed by resolution) or $\mathcal{O}(\tau(6,4)^n) \subseteq \mathcal{O}(1.1510^n)$ (if no member of y contains a singleton, then the second call will remove at least 4 variables); if $|x| = 5, |y| = 5$ and x and y overlap in 3 variables, we get a running time in $\mathcal{O}(\tau(5,4)^n)$; if $|x| = |y| = 6$ and $|C| = 2$, we get a running time in $\mathcal{O}(\tau(9,2)^n) \subseteq \mathcal{O}(1.1618^n)$. These cases dominate all other, and so we get a running time in $\mathcal{O}(\tau(5,4)^n) \subseteq \mathcal{O}(1.1674^n)$.

3. As $|C_1|+|C_2|$ is 5, 6 or 7 the first call removes at least 9 variables. In the second call, the 3-clause shrinks to a 2-clause which will be removed by canonical rule 4. Thus this case runs in time $\mathcal{O}(\tau(9,2)^n) \subseteq \mathcal{O}(1.1618^n)$.

4. In the first call the variables $c - e$ are removed. If d and e do not occur together in another clause both will be removed; else they must appear together in a clause of length 4, 5 or 6 and then at least 2 more variables are removed. In any case, the first call removes at least 5 variables. The second call and canonisation creates the clauses $x = (a \vee b \vee c)$, $y = (\bar{c} \vee b \vee C_1)$ $z = (a \vee \bar{b} \vee C_2)$. By resolution, b and c will be removed. Hence this case runs in time $\mathcal{O}(\tau(5,4)^n) \subseteq \mathcal{O}(1.1673^n)$.

Remark 1 *By now no variable a occurs more than thrice. After line 1 it is still possible for a variable a to occur four times. However, after lines 2 and 3 all such instances consist of overlapping 5-clauses (and possibly 4-clauses.) By lines 2 and 4 all of these will contain a singleton and so a will be removed by canonisation (rule 6.)*

5. Since e is a singleton, d must occur in at least another clause and so, when d is true, 7 variables are removed. When d is false rule 12 is applicable and so this case runs in time $\mathcal{O}(\tau(7,3)^n) \subseteq \mathcal{O}(1.1586^n)$.

Remark 2 *Note that by now, the only remaining cases for a 5-clause to overlap with a 4-clause are these two: $x = (a \vee b \vee c \vee d \vee e), y = (a \vee b \vee f \vee g), z = (a \vee h \vee i \vee j)$ or $x = (a \vee b \vee c \vee d \vee e), y = (a \vee b \vee f \vee g), z = (a \vee h \vee i \vee j \vee k)$ such that both x and y contain a singleton and $\delta(b) = 2$.*

6. This is the only remaining case for a 5-clause $x = (a \vee b \vee c \vee d \vee e)$ and a 6-clause $y = (a \vee b \vee f \vee g \vee h \vee i)$ sharing two variables. Because, by canonicity a must appear in at least one other clause z. By line 1, z must contain no more than two variables not seen in x and y. The case $|z| = 3$ is taken care of by line 3. That leaves the only possibilities that $z = (a \vee c \vee f \vee j \vee k)$ or $z = (a \vee c \vee j \vee k)$. By lines 4 and 2 e and j must be singletons. If b occurs elsewhere we trivially end up with a running time in $\mathcal{O}(\tau(9,2)^n) \subseteq \mathcal{O}(1.1618^n)$. Else, after the first call z will contain two singletons and so we get a running time in $\mathcal{O}(\tau(9,2)^n) \subseteq \mathcal{O}(1.1618^n)$ here too.

Remark 3 *By now, no 5-clause overlaps with a 6-clause.*

7. If a occurs in two 3-clauses we will have a running time in $\mathcal{O}(\tau(3,7)^n) \subseteq \mathcal{O}(1.1586^n)$, if a occurs in two 4-clauses and one 3-clause a running time in $\mathcal{O}(\tau(9,2)^n) \subseteq \mathcal{O}(1.1618^n)$ and if a occurs in two 4-clauses and one 6-clause a running time in $\mathcal{O}(\tau(13,1)^n) \subseteq \mathcal{O}(1.1632^n)$. These cases dominate all other.

8. We look at the subcases:

   (a) The case with x gives a running time in $\mathcal{O}\left(\tau(10,10,4)^n\right) \subseteq \mathcal{O}\left(1.1610^n\right)$,

   (b) the case with y gives a running time in $\mathcal{O}\left(\tau(10,11,3)^n\right) \subseteq \mathcal{O}\left(1.1721^n\right)$,

   (c) the case with z gives a running time in $\mathcal{O}\left(\tau(11,8,4)^n\right) \subseteq \mathcal{O}\left(1.1694^n\right)$,

   (d) and as for the case with w, in both calls all members of w will be removed and in the first call a clause with two singletons will be created. Thus this subcase runs in time $\mathcal{O}\left(\tau(5,4)^n\right) \subseteq \mathcal{O}\left(1.1674^n\right)$.

   For the entire case with all subcases we get a running time in $\mathcal{O}\left(\tau(10,11,3)^n\right) \subseteq \mathcal{O}\left(1.1721^n\right)$.

9. As a and b occur without the other, in the first call at least 8 variables are removed and in the second at least 2 variables. Now, if in either of these two calls a 2-clause is created we will have a running time in $\mathcal{O}\left(\tau(9,2)^n\right)$ or $\mathcal{O}\left(\tau(8,3)^n\right)$. Otherwise no new variables of degree higher than 2 are created and we will have a limited number of possibilities as for what case of DX will become applicable next. If one of $c - h$ appears in w', then after the first call line 3 or 7 will be applicable and in the worst case we get a running time in $\mathcal{O}\left(\tau(17,10,2)^n\right) \subseteq \mathcal{O}\left(1.1720^n\right)$.

   Before we look at the other cases — that one of $c - h$ is heavy — we need to enumerate the remaining configurations of heavy variables. In the following enumeration some variables are obviously heavy, such as a. The status of the other is unspecified, unless otherwise stated.

   (a) 5-5-5a: $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g \vee h)$, $z = (a \vee c \vee i \vee j \vee k)$ and $w = (b \vee f \vee l \vee m \vee n)$.

   (b) 5-5-5b: $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g \vee h)$, $z = (a \vee c \vee i \vee j \vee k)$ and $w = (b \vee l \vee m \vee n)$.

(c) 5-5-4a: $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g \vee h)$, $z = (a \vee i \vee j \vee k)$ and $w = (b \vee l \vee m \vee n)$.

(d) 5-5-4b: $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee b \vee f \vee g \vee h)$, $z = (a \vee i \vee j \vee k)$ such that $\delta(b) = 2$.

(e) 5-5-4c: $x = (a \vee b \vee c \vee d \vee e), y = (a \vee b \vee f \vee g), z = (a \vee h \vee i \vee j \vee k)$

(f) 5-4-4: $x = (a \vee b \vee c \vee d \vee e)$, $y = (a \vee f \vee g \vee h)$, $z = (a \vee i \vee j \vee k)$

(g) 4-4-4: $x = (a \vee b \vee c \vee d)$, $y = (a \vee e \vee f \vee g)$, $z = (a \vee h \vee i \vee j)$

Clearly, after the second call line 3 or 7 will be applicable and we get a running time in $\mathcal{O}(\tau(8, 11, 4)^n) \subseteq \mathcal{O}(1.1694^n)$. Hence, for the entire case we get a running time in $\mathcal{O}(\tau(17, 10, 2)^n) \subseteq \mathcal{O}(1.1720^n)$.

10. In the first call 6 variables are removed and in the second at least 2 variables. Reasoning as in the previous case we see that this case runs in time $\mathcal{O}(\tau(15, 8, 3)^n) \subseteq \mathcal{O}(1.1709^n)$.

11. In both calls all members of y will be removed and after the second call line 7 will be applicable. Hence, this case runs in time $\mathcal{O}(\tau(13, 7, 4)^n) \subseteq \mathcal{O}(1.1694^n)$.

    Remark 4 *Consider a 4-clause ω that contains at least one heavy variable and has no singleton. By line 8(a) it contains only one heavy variable. By line 11, if it shares a variable with another 4-clause ω', then ω' must contain a singleton. By line 8(d), ω' is the only 4-clause ω shares variables with.*

12. In the first call at least 10 variables are removed. And in the third at least 4. As for the second call, by Remark 4, b must appear in another clause of length at least 5 and so at least 8 variables are removed. Hence, this case runs in time $\mathcal{O}(\tau(10, 8, 4)^n)$. However, in the last call the heavy variable e occurs in a 3-clause which in the next step of DX will give a running time in at worst $\mathcal{O}(\tau(9, 2)^n)$, and so this case runs in $\mathcal{O}(\tau(10, 8, 6, 13)^n) \subseteq \mathcal{O}(1.1718^n)$ time.

Remark 5 *By Remark 4 and line 12 a 4-clause ω containing a heavy variable and no singletons shares no variables with another 4-clause containing heavy variables.*

On the other hand, consider ω' which is a 4-clause containing a singleton and at least one heavy variable. By canonicity it shares no non-heavy variables with another similar 4-clause.

13. We will again look at the above enumeration of configurations to decide the ratio of heavy variables to all variables. For some configurations we "claim" a number of non-heavy variables to each heavy variable. We thus need to ensure that a claimed variable is claimed only once. Consider the clause

$$w = (a \vee b \vee c \vee d).$$

Say that a is heavy and $b - d$ have degree two and are guaranteed not to appear in another clause containing a heavy variable. As a is the only neighbouring heavy variable, a may safely claim all of $b - d$. However, sometimes the situation is more complicated. For some configurations we cannot ensure that a claimed variable is not claimed by other heavy variables. We then claim fractions of non-heavy variables. For an example, consider again w, but now say that a is heavy, and so are possibly also b and c. However, d is guaranteed to be a singleton. Provided that b and c do not claim more than one third each of d, a may also claim one third.

(a) 5-5-5a: By line 4 both x and y must contain a singleton and so by canonicity both c and f must be heavy. That means that line 9 has already removed all appearances of this configuration and we do not need to consider it anymore.

(b) 5-5-5b: To a and b we claim all of c, e, l, m, n and half of d, f, g and h.

None of the wholly claimed can be claimed by another pair of heavy variables in a 5-5-5b configuration because in x

only a and b are heavy, c has degree 2 and by line 9 z cannot play the role of x or y in another 5-5-5b configuration; e is a singleton by line 7 and x cannot contain another heavy variable by line 9. As for $l - n$, by line 9 none of them can participate in x or y of a 5-5-5b configuration, by line 8 none of them are heavy and by Remark 5 none of them can appear in another 4-clause having a heavy variable.

For this configuration we have a ratio of 2/9.

(c) 5-5-4a: To a and b we claim all of $i - n$ and half of $c - h$. The claiming of $i - n$ is defended as the variables of w in 5-5-5b. By line 9 none of $c - h$ can play the role of e or c in a 5-5-5b configuration.

For this configuration we have a ratio of 2/11.

(d) 5-5-4b: To a we claim all of b, $i - k$.

For this configuration we have a ratio of 1/5.

(e) 5-5-4c: To a we claim all of b, f, g and half of $c - e$.

For this configuration we have a ratio of 2/11.

(f) 5-4-4: To a we claim all of $f - k$.

For this configuration we have a ratio of 1/7.

(g) 4-4-4: At least one clause, say x has no singleton and we may safely claim $b - d$. In the worst case e, f, h and j are heavy. We then claim one third of g and one third of j.

For this configuration we have a ratio of 3/14.

As $1/7 < 2/11 < 3/14 < 2/9$ we see that this case runs in $\mathcal{O}\left(2^{\frac{2}{9}n}\right) \subseteq \mathcal{O}\left(1.1666^n\right)$ time.

Examining all of the above cases we conclude that the running time of DX is in $\mathcal{O}\left(1.1730^n\right)$. □

Chapter 4

General Exact Satisfiability

In this chapter we study a natural generalisation of XSAT. We first present some features of the language X_iSAT, including canonical rules. Then follow Sections 4.2 and 4.3, which present and analyse two exact polynomial space algorithms: One for X_iSAT and one dedicated algorithm for X_2SAT. Section 4.4 shows how to deal with X_iSAT using exponential space.

4.1 Properties of the X_iSAT Problem

The X_iSAT problem is obviously *NP*-complete since XSAT $\leq_m^p$ X_iSAT: Any occurrence of the literal a is replaced by i occurrences. (The reader may recall that the notation $p_1 \leq_m^p p_2$ means that there exists a polynomial reduction from the problem p_1 to the problem p_2.) As one could imagine, for higher i our polynomial space algorithms have worse running times. Surprisingly, this does not hold for the exponential space algorithm to be presented.

We will use Roman numerals to indicate the number of true literals a clause requires to be satisfied. For instance, $x = (a \vee b \vee c \vee d)^{\text{II}}$ is a clause in an X_2SAT instance.

One property of X$_i$SAT that will prove useful in the branching algorithms is given in the following lemma:

Lemma 6. A formula F, where each variable occurs at most twice, can be reduced in polynomial time to a formula F', such that $F' \in$ XSAT iff $F \in$ X$_i$SAT. Furthermore, the number of variables is increased only by a polynomial amount and each variable in F' occurs at most twice.

Proof. For a clause $(a_1 \vee a_2 \vee a_3 \vee \ldots \vee a_k)^i$ we make the following XSAT clauses, where the b's are new variables:

$$
\begin{array}{ll}
x_1 = (\bar{a}_1 \vee b_1^1 \vee b_1^2 \vee \ldots \vee b_1^i)^I & y_1 = (b_1^1 \vee b_2^1 \vee b_3^1 \vee \ldots \vee b_k^1)^I \\
x_2 = (\bar{a}_2 \vee b_2^1 \vee b_2^2 \vee \ldots \vee b_2^i)^I & y_2 = (b_1^2 \vee b_2^2 \vee b_3^2 \vee \ldots \vee b_k^2)^I \\
x_3 = (\bar{a}_3 \vee b_3^1 \vee b_3^2 \vee \ldots \vee b_3^i)^I & y_3 = (b_1^3 \vee b_2^3 \vee b_3^3 \vee \ldots \vee b_k^3)^I \\
\vdots & \vdots \\
\vdots & y_i = (b_1^i \vee b_2^i \vee b_3^i \vee \ldots \vee b_k^i)^I \\
x_k = (\bar{a}_k \vee b_k^1 \vee b_k^2 \vee \ldots \vee b_k^i)^I &
\end{array}
$$

First note that an assignment satisfying $(a_1 \vee a_2 \vee a_3 \vee \ldots \vee a_k)^i$ also allows an assignment satisfying the new XSAT clauses: Assume w.l.o.g. that $a_1 - a_i$ are true and $a_{i+1} - a_k$ false. One solution is to assign all b_l^l (i.e. $a_1^1, a_2^2, \ldots, a_i^i$) true and the other b's false. For the other direction: If we consider the clauses $x_1 - x_k$ as a matrix X (as depicted, i.e. the first column is $\bar{a}_1 - \bar{a}_k$), then we see that the clauses $y_1 - y_i$ force exactly one true literal in each column of X, except the first column. As the number of y's is i, that means that i of the clauses in X get satisfied and in these i of the a's will become true (because the $\bar{a}_x$'s will be false.) In the other clauses of X, the a's will be false.

In the reduction we need to add $i \cdot k$ new variables for each X$_i$SAT clause. As each variable occurs at most twice in F, the number of clauses of F is a polynomial in n, and so $n(F')$ is a polynomial in $n(F)$. □

To illustrate this reduction, consider the clause $x = (a \vee b \vee c \vee d)^{\mathrm{II}}$. The resulting XSAT clauses are shown in Figure 4.1.

$$\begin{array}{ll} (\bar{a} \vee k \vee l)^{\text{I}} & (k \vee m \vee o \vee q)^{\text{I}} \\ (\bar{b} \vee m \vee n)^{\text{I}} & (l \vee n \vee p \vee r)^{\text{I}} \\ (\bar{c} \vee o \vee p)^{\text{I}} & \\ (\bar{d} \vee q \vee r)^{\text{I}} & \end{array}$$

Figure 4.1: *An instance of* XSAT *equivalent to the clause* $(a \vee b \vee c \vee d)^{\text{II}}$

We now conclude the following:

Corollary 7. For a formula F where each variable occurs at most twice, it is polynomial time decidable whether $F \in \text{X}_i\text{SAT}$.

The corollary follows from Lemma 3 and Lemma 6.

In the branching algorithm for X_iSAT to be presented later, the recursive decomposition will create various kinds of constraints represented as clauses, that is, when setting the variables of a clause, other clauses will be affected. For instance, the clause $(a \vee b \vee c \vee d)^{\text{II}}$, which requires two true literals to be satisfied, will become $(a \vee b \vee c)^{\text{I}}$ if d is set to true. When there are different types of clauses (that require a different number of true literals) in a formula, we say that it is *mixed*.

Some of the canonical rules of Chapter 2 extend to X_iSAT, others do not. For clarity of presentation we here present the ones used in this chapter. It is straightforward to see that they can be enforced in polynomial time and that they do not change the X_iSAT satisfiability when applied to a formula F. For our general X_iSAT solver we use only the first rule, but the dedicated algorithm for X_2SAT uses all of them.

1. Pick an X_iSAT clause with $i+k$ singletons. Remove k singletons.
2. Pick a clause $(a \vee b)^{\text{I}}$, remove it and let $F \leftarrow F(a/\bar{b})$
3. Pick an X_iSAT clause A such that $|A| = i$. Remove it and let $F \leftarrow F(a_j/true)$ for all literals a_j of A.
4. Pick two clauses $(a \vee b \vee A)^{\text{I}}$ and $(\bar{a} \vee b \vee B)^{\text{I}}$ and let $F \leftarrow F(b/false)$

5. For two X_iSAT clauses (A) and $(A \vee B)$ let $F \leftarrow F(B/false)$
6. If there are two clauses $x = (A \vee B)^{\text{II}}$ and $y = (A \vee \bar{B})^{\text{II}}$, such that $|B| \pmod 2 = 1$ or $|B| > 4$, then let $F \leftarrow \{()\}$
7. If there are two clauses $x = (A \vee b \vee c)^{\text{II}}$ and $y = (A \vee \bar{b} \vee \bar{c})^{\text{II}}$, then let $F \leftarrow F(b/\bar{c})$
8. For two X_iSAT clauses $(a \vee A)$ and $(A \vee b)$, let $F \leftarrow F(a/b)$
9. If there are two clauses $(a \vee A \vee b)^{\text{II}}$ and $(\bar{a} \vee A \vee c)^{\text{II}}$, then let $F \leftarrow F(b/\bar{c})$
10. If there are two clauses $(a \vee b \vee A \vee c)^{\text{II}}$ and $(\bar{a} \vee \bar{b} \vee A \vee d)^{\text{II}}$, then let $F \leftarrow F(c/\bar{d})$
11. If there are two clauses $(a \vee b \vee c \vee d \vee e \vee A)^{\text{II}}$ and $(\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{d} \vee \bar{e} \vee B)^{\text{II}}$, then let $F \leftarrow \{()\}$
12. If there are two clauses $(a \vee b \vee c \vee d \vee A)^{\text{II}}$ and $(\bar{a} \vee \bar{b} \vee \bar{c} \vee \bar{d} \vee B)^{\text{II}}$, then let $F \leftarrow F(A/false; B/false)$
13. If there are two clauses $(a \vee b \vee c \vee A)^{\text{II}}$ and $(\bar{a} \vee \bar{b} \vee \bar{c} \vee B)^{\text{II}}$, then let $F \leftarrow F(A/false; B/false)$

4.2 A Polynomial Space Exact Algorithm

The basic idea behind all known polynomial space algorithms for XSAT has been DPLL style branching. One way to deal with X_iSAT is to generalise the approach, so that for a certain clause y, we test all $\binom{|y|}{i}$ assignments to the variables of y making i literals true. That makes $\binom{|y|}{i}$ recursive calls, in each of which $|y|$ variables are removed — when i literals are true, the rest have to be false and so $|y|$ variables are set. Of course the length of y is crucial for the running time. Short clauses (in comparison with i) are good w.r.t. the running time. However, long clauses are also good. In Table 4.1 is an overview of the first sublanguages of X_iSAT and the first clause lengths. Each entry is calculated as $\binom{|y|}{i}^{n/|y|}$. Using this table, we can easily find

L	XSAT	X$_2$SAT	X$_3$SAT	X$_4$SAT
1	*	*	*	*
2	*	*	*	*
3	$3^{n/3} < 1.45^n$	$3^{n/3} < 1.45^n$	*	*
4	$4^{n/4} < 1.42^n$	$6^{n/4} < 1.57^n$	$4^{n/4} < 1.42^n$	*
5	$5^{n/5} < 1.38^n$	$10^{n/5} < 1.59^n$	$10^{n/5} < 1.59^n$	$5^{n/5} < 1.38^n$
6	$6^{n/6} < 1.35^n$	$15^{n/6} < 1.58^n$	$20^{n/6} < 1.65^n$	$15^{n/6} < 1.58^n$
7	$7^{n/7} < 1.33^n$	$21^{n/7} < 1.55^n$	$35^{n/7} < 1.67^n$	$35^{n/7} < 1.67^n$
8	$8^{n/8} < 1.30^n$	$28^{n/8} < 1.52^n$	$56^{n/8} < 1.66^n$	$70^{n/8} < 1.71^n$
9	$9^{n/9} < 1.28^n$	$36^{n/9} < 1.49^n$	$84^{n/9} < 1.64^n$	$126^{n/9} < 1.72^n$
10	$10^{n/10} < 1.26^n$	$45^{n/10} < 1.47^n$	$120^{n/10} < 1.62^n$	$210^{n/10} < 1.71^n$

Table 4.1: *Running times for D_i should it always encounter the same kind of clause; the first column indicates clause length and '*' indicates polynomial time*

a clause which is the best choice, i.e. gives the fastest running time. We call such a clause *preferable*. We need a lemma showing that it is polynomial time computable to find a preferable clause.

Lemma 8. For any instance of X$_i$SAT, where i is fixed, it is polynomial time computable to find a preferable clause.

Proof. Looking at Table 4.1, we see that it consists of real numbers, which we cannot represent. And obviously, for long clauses the float number representation will not do. However, note that we actually only need to compare all the clauses pairwise in order to find a preferable clause and due to the fact that $\binom{|y_1|}{i}^{n/|y_1|} < \binom{|y_2|}{i}^{n/|y_2|}$ iff $\binom{|y_1|}{i}^{n \cdot |y_2|} < \binom{|y_2|}{i}^{n \cdot |y_1|}$, this obstacle can be overcome.

We also need to show that the longest clause length is a polynomial in n. We will assume that no clause contains the same literal a more than i times (in this trivial case a can be directly set to false.) Thus, the longest clause has length at most $2in$ (then it contains all possible literals.) □

As the formula changes during the recursive decomposition of generalised DPLL branching, different clauses will become preferable. Our

Algorithm $D_i(F)$

1. If there is an $\mathrm{X}_i\textsc{sat}$ clause with $i+k$ singletons, then remove k of those singletons;

2. If 3/5 or less of the variables are heavy, then test all possible assignments to these variables. For each such partial assignment to the variables, use Corollary 7 to decide if there is a model.

3. Pick a preferable clause y with as few singletons as possible and make $\binom{|y|}{i}$ recursive calls, where each call is on the form $D_i(F(a_1/true; a_2/true, \ldots, \vee a_i/true, a_{i+1}/false, \ldots))$.

Figure 4.2: *Algorithm D_i for deciding* $\mathrm{X}_i\textsc{sat}$

algorithm D_i, as shown in Figure 4.2, elaborates on this idea. For clarity of presentation we assume that a variable occurs in each clause at most once. (Loosening of this restriction would not introduce any new worst cases, but it would introduce some uninteresting technicalities.) We also assume that in the substitution, if a (partial) assignment is made such that any clause becomes over-satisfied or too short to ever be satisfied, then the unsatisfiable formula $F = \{()\}$ is the result. The purpose of line 2 is to limit the number of singletons in the following lines. It works by forcing a certain percentage of the variables to be heavy. The choice of 3/5 is made so that it works well for the first sublanguages. A higher ratio would give a worse upper time bound for $\mathrm{X}_2\textsc{sat}$, while a lower would not limit the number of singletons sufficiently. As will become clear in the time complexity analysis, it is reasonable to believe that for higher i a larger constant will give a better trade-off. Note that when F is small enough, line 2 is applicable and so the recursion ends. Line 1 will limit the number of singletons in a preferable clause when the clause is long. The benefit of this will become clear in the time complexity analysis.

Theorem 9. $D_i(F)$ decides whether $F \in \mathrm{X}_i\textsc{sat}$.

Proof. Line 1 is a canonical rule. Line 2 is correct by Corollary 7.

Line 3 is correct since all models for F must have i literals true in y. □

We now examine D_i w.r.t. time complexity. Let T_{D_i} indicate the running time of $D_i(F)$.

Theorem 10. For every fixed i, T_{D_i} is in

$$\mathcal{O}\left(\max\left\{\max_{\substack{m \leq n \\ i \leq m}} \binom{m}{i}^{1/m}, 1.5157\right\}^n\right) \subseteq \mathcal{O}\left(2^{(1-\varepsilon)n}\right)$$

for some ε such that $0 < \varepsilon < 1$.

Proof. Line 1 takes polynomial time to execute. As for line 2, we can safely disregard the polynomial time work spent on matching. Hence the interesting thing is the size of the recursion tree, which is $2^{3n/5} \approx 1.5157^n$. Similarly for line 3, we disregard the polynomial work done. The recursion tree of $D_i(F)$ has size at most $\binom{m}{i}^{n/m}$ and so the first big-Oh expression is justified.

To justify the $\mathcal{O}\left(2^{(1-\varepsilon)n}\right)$ inclusion, first note that $\binom{m}{i}^{n/m} = 2^{\log_2 \binom{m}{i}^{n/m}} = 2^{\frac{n}{m}\log_2 \binom{m}{i}}$. Then, remember that $\binom{m}{i}$ is the number of subsets of size i whose elements are picked from a set of size m. As the power set has size 2^m, $\binom{m}{i}$ is always smaller than that. Hence it follows that $\log_2 \binom{m}{i} < m$ and so $2^{\frac{n}{m}\log_2 \binom{m}{i}} = 2^{(1-\varepsilon)n}$ for some ε $0 < \varepsilon < 1$. □

We now try to refine the analysis to achieve a tighter upper time bound. Unfortunately, in order to do that we need to know the worst clause length for every sublanguage. Looking again at Table 4.1 one could think that the worst clause length is $2i+1$. However, that is not always the case. Extending the table, one can see that the pattern is changed for X$_{12}$SAT where the worst clause length is 26. It is still an open problem where to find the worst clause length for a given sublanguage. However, for the first sublanguages we can perform a better analysis:

Theorem 11. For X$_2$SAT, X$_3$SAT and X$_4$SAT T_{D_i} is in $\mathcal{O}(1.5157^n)$, $\mathcal{O}(1.6214^n)$ and $\mathcal{O}(1.6848^n)$ respectively.

Proof. Starting with X$_2$SAT, we need to take a closer look at line 3.

Once line 3 has been applied, the formula is likely to have become mixed, and so in the general case, y might be a clause requiring one or two true literals. If y requires one true literal we have a worst clause length of 3, where 3 branches are made, in each of which 3 variables are removed. If the algorithm always did this branching, we would have a branching tree of size $\mathcal{O}(1.4423^n)$. If y requires two true literals, we will have a worst case when $|y| = 5$. If D_i always had to branch upon such a y, we would have a running time in $\mathcal{O}\left(\binom{5}{2}^{n/5}\right) \subseteq \mathcal{O}(1.5849^n)$. However, note that due to the previous lines and the fact that y has the smallest possible number of singletons, at most one variable of y is a singleton. (Line 1 is not strong enough to impose this, but line 2 ensures that there are clauses with at least 4 heavy variables.) That means that in each of the 10 calls, other clauses will be affected. As y was most preferable, all clauses must be X$_2$SAT clauses of length 5, and of the 10 calls at most one call will not set a literal true in another clause (even in the worst case, only one combination of the non-singletons will not set a literal true, see Figure 4.3.) Hence, for the worst case, in 9 of the recursive calls the algorithm will in the immediately following step encounter an XSAT clause of length 4 and in one recursive call encounter an X$_2$SAT clause of length 4. This means that we will have an upper time bound $\mathcal{O}(c^n)$ where $c = \tau(9^6, 9^{9\cdot 4}) \approx 1.5149$. In this case, line 2 will decide the overall running time of the algorithm.

Looking at Table 4.1, let us examine the other clause lengths that are possible worst-case candidates.

1. Clause length 4: In this case there will be at most one singleton in the clause picked and so we get $c = \tau(7^{5\cdot 3}, 7^3) \approx 1.5113$.

2. Clause length 6: As $\frac{2\cdot 6}{5} = 2.4$ there may be two singletons in the clause we picked. Hence we get $c = \tau(11^{14\cdot 5}, 11^{2\cdot 10}) \approx 1.5055$.

$$
\begin{array}{ll}
(a \vee b \vee c \vee d \vee e)^{\mathrm{II}} & (\bar{a} \vee a_1 \vee a_2 \vee a_3 \vee a_4)^{\mathrm{II}} \\
 & (b \vee b_1 \vee b_2 \vee b_3 \vee b_4)^{\mathrm{II}} \\
 & (c \vee c_1 \vee c_2 \vee c_3 \vee c_4)^{\mathrm{II}} \\
 & (d \vee d_1 \vee d_2 \vee d_3 \vee d_4)^{\mathrm{II}}
\end{array}
$$

Figure 4.3: *For this configuration and the assignment $a = e = true$, no* XSAT *clause will be created, instead the next step of D_i will branch on some* X_2SAT *clause of length 4*

3. Clause length 7: As $\frac{2 \cdot 7}{5} = 2.8$ there may be two singletons in the clause we picked. Hence we get $c = \tau(13^{19 \cdot 6}, 13^{2 \cdot 15}) \approx 1.4657$.

4. Clause length 8: $\frac{2 \cdot 8}{5} = 3.2$ but line 2 prevents the possibility of three singletons. Hence we get $c = \tau(15^{26 \cdot 7}, 15^{2 \cdot 21}) \approx 1.4345$.

When it comes to X_3SAT and X_4SAT the analysis is almost identical. Here c will be $\tau(13^{33 \cdot 15}, 13^{2 \cdot 20})$ and $\tau(17^{123 \cdot 56}, 17^{3 \cdot 70})$ respectively. □

4.3 An Algorithm for X_2SAT

The use of canonisation has proved fruitful in the construction of algorithms for XSAT, and so one could hope that the use of more canonical rules would improve D_i further (in terms of proved upper time bounds.) However, the problem is that while canonisation helps improve many cases such as overlaps between clauses, many singletons and few occurrences of heavy variables, yet the worst case of the algorithm still remains, namely: All clauses have the worst possible length, no pair of clauses share more than one variable and there are many heavy variables. For X_2SAT we have constructed an algorithm that obtains a better upper time than D_i. The algorithm $D_2(F)$, as shown in Figure 4.4, carefully chooses variables to branch on, uses canonisation, and arrives at the bad case described. Then the algorithm picks a clause that has two heavy variables a and b. It makes three recursive calls, $D_2(F(a/true;\ b/true))$, $D_2(F(a/\bar{b}))$ and

$D_2(F(a/false; b/false))$. By a careful case analysis of how D_2 behaves in the three calls, an interesting time bound can be established. When this case is no longer applicable, it can be shown that there are sufficiently few heavy variables left and the test and match technique can be used.
The following theorem establishes the correctness of D_2:

Theorem 12. $D_2(F)$ will correctly decide whether F has an X_2SAT model.

Proof. We look at the non-trivial cases of D_2:

1. Correct, as a model must be a model for all components.

2. If the clause is an XSAT clause, then both a and b cannot be $true$, so the two cases 1) one of a and b is $true$; 2) both are $false$, cover all possibilities. If the clause is $(\bar{a} \vee \bar{b} \vee \bar{c})^{\text{II}}$, it is in effect identical to $(a \vee b \vee c)^{\text{I}}$ and so this is also correct.

3. The two cases cover all possibilities: Either it holds that one of a and b and one of c and d are $true$, or it holds that both a and b are $true$ and the other two $false$ or vice versa.

4. When $c = true$ it holds that $a \neq b$.

5. Both of b and c cannot be $true$, and so all possible cases are covered.

6. One of $b = true$ and $b = false$ holds. In the first case one of $a = true$ and $a = false$ holds.

7. Zero, one or two literals of A are $true$.

8. The second branch covers the two possibilities $a = true, b = false$ and $a = false, b = true$.

9. Correct by Lemma 3 and Lemma 6.

□

Algorithm $D_2(F)$

0. Canonise F and if $n(F) < 10$, then perform an exhaustive search to find a model
1. If F is not connected, then, for the connected components $F_1, \ldots, F_j$ return 'Yes' if all calls $D_2(F_i)$ produces 'Yes' and return 'No' otherwise
2. Pick a clause $(a \vee b \vee A)^{\mathrm{I}}$ or a clause $(\bar{a} \vee \bar{b} \vee \bar{c})^{\mathrm{II}}$; return $D_2(F(a/\bar{b}))$ OR $D_2(F(a/false; b/false))$
3. Pick a clause $(a \vee b \vee c \vee d)^{\mathrm{II}}$; return $D_2(F(a/\bar{b}; c/\bar{d}))$ OR $D_2(F(a/b;\ c/d; b/\bar{d}))$
4. Pick two clauses $(a \vee b \vee c \vee A)^{\mathrm{II}}$ and $(\bar{a} \vee \bar{b} \vee c \vee B)^{\mathrm{II}}$ such that the intersection between $Var(A)$ and $Var(B)$ is empty; return $D_2(F(c/true; a/\bar{b}))$ OR $D_2(F(c/false))$
5. Pick two clauses $(a \vee b \vee c \vee A)^{\mathrm{II}}$ and $(\bar{a} \vee b \vee c \vee B)^{\mathrm{II}}$; return $D_2(F(b/\bar{c}; a/true))$ OR $D_2(F(b/\bar{c}; a/false))$ OR $D_2(F(b/false; c/false))$
6. Pick two clauses $(a \vee b \vee A)^{\mathrm{II}}$ and $(\bar{a} \vee b \vee B)^{\mathrm{II}}$; return $D_2(F(a/true; b/true))$ OR $D_2(F(a/false; b/true))$ OR $D_2(F(b/false))$
7. Pick two clauses $x = (A \vee B)^{\mathrm{II}}$ and $y = (A \vee C)^{\mathrm{II}}$ such that $|A| \geq 2$; return $D_2(F \cup (A)^{\mathrm{II}})$ OR $D_2(F \cup (A)^{\mathrm{I}})$ OR $D_2(F(A/false))$.
8. Pick a clause $x = (a \vee b \vee A)^{\mathrm{II}}$ where a and b are heavy variables; return $D_2(F(a/true;\ b/true))$ OR $D_2(F(a/\bar{b})$ OR $D_2(F(a/false; b/false)$.
9. Cycle through all possible assignments to the heavy variables. For each such partial assignment to the variables, transform the instance to an XSAT instance, using the reduction of Lemma 6, then use the matching techniques by Lemma 3.

Figure 4.4: *Algorithm D_2 for deciding* $\mathrm{X_2SAT}$

Theorem 13. Algorithm D_2 runs in time $\mathcal{O}(1.4511^n)$.

Proof. We examine each of the cases:

0. Runs in polynomial time.
1. This case will not increase the running time.
2. We look at the possible subcases:
   (a) We picked a clause $(a \vee b \vee c \vee d)^{\mathrm{I}}$: In the first branch, in the call $D_2(F(a/\bar{b}))$ we will have the following steps: The clause will become $(\bar{b} \vee b \vee c \vee d)^{\mathrm{I}}$ which will become $(true \vee c \vee d)^{\mathrm{I}}$. Then the clause is removed and c and d replaced by $false$ and hence 3 variables are removed. The second branch will result in the following steps: $(false \vee false \vee c \vee d)^{\mathrm{I}}$, $(c \vee d)^{\mathrm{I}}$. The clause $(c \vee d)$ will then be immediately taken care of by the canonisation step following. Hence, this case runs in $\mathcal{O}(\tau(3,3)^n) \subseteq \mathcal{O}(1.2600^n)$ time.

   (b) We picked a clause $(a \vee b \vee c)^{\mathrm{I}}$: In the first branch, when a is replaced by $\bar{b}$, c will be put to $false$, so two variables are removed. In the other branch, when $a = b = false$, the clause $(c)^{\mathrm{I}}$ is created and three variables are removed in this branch. Hence, this case runs in time $\mathcal{O}(\tau(3,2)^n) \subseteq \mathcal{O}(1.3248^n)$.

   (c) We picked an XSAT clause longer than 4. The worst case is when the clause has length 5. This case runs in $\mathcal{O}(\tau(4,2)^n) \subseteq \mathcal{O}(1.2721^n)$ time.

   (d) We picked a clause $(\bar{a} \vee \bar{b} \vee \bar{c})^{\mathrm{II}}$. This case runs in time $\mathcal{O}(\tau(3,2)^n) \subseteq \mathcal{O}(1.2721^n)$.
3. This case runs in $\mathcal{O}(\tau(2,3)^n) \subseteq \mathcal{O}(1.2600^n)$ time.
4. As the formula has been canonised, $|A \cup B| \geq 3$ and so in the first branch at least 5 variables are removed (c is $true$ and $b \vee \bar{b}$ equals $true$ so the literals of A and B are set to $false$.) Hence, we have a running time in $\mathcal{O}(\tau(5,1)^n) \subseteq \mathcal{O}(1.3248^n)$ time.

5. This case runs in $\mathcal{O}(\tau(4,4,2)^n) \subseteq \mathcal{O}(1.4143^n)$ time.

6. By case 2, both clauses are longer than 4 and so this case runs in $\mathcal{O}(\tau(5,5,1)^n) \subseteq \mathcal{O}(1.4511^n)$ time.

7. Doing a naïve analysis like in previous cases, looking only at the direct effects, we would obtain very bad figures. For example, assuming $|x| = |y| = 5$ and $|A| = 2$, we would reason that in the first branch 2+3+3 variables are removed, in the second 1 variable and in the third 2 variables, giving an upper bound in $\mathcal{O}(\tau(8,1,2)^n) \subseteq \mathcal{O}(1.6408^n)$. However, doing the same reasoning as in D_i, broadening the perspective to the branchings that will be done immediately afterwards, we end up with better time bounds. In our example, the first branch we cannot say more about, and so we stay with 8 variables removed. In the second branch, however, we will have the two clauses $(B)^{\mathrm{I}} = (a \vee b \vee c)$ and $(C)^{\mathrm{I}} = (d \vee e \vee f)$, and case 1 will be applicable twice. Following their way downward the recursion tree, see Figure 4.5, we see that effectively, there will be four branches and the number of variables removed are 6, 5, 6 and 7 respectively (the one variable removed by the explicit creation of $(A)^{\mathrm{I}}$ included.) We may continue and reason similarly about the third branch, however, the figures we obtained are good enough: $\mathcal{O}(\tau(8;6,5,6,7;2)^n) \subseteq \mathcal{O}(1.4401^n)$. Note that the sign ';' is used to help the reader see how the expansion is done. We now look at the remaining cases:

   (a) For $|A| = 2$, we have already described the worst case, because if any of B and C are longer than 3 we will be able to remove more variables.

   (b) For $|A| = 3$; if $|B| = |C| = 2$ we note that the second branch can be expanded to two branches due to the explicit creation of $(A)^{\mathrm{I}}$ and so we get a running time in $\mathcal{O}(\tau(4;5,4;3)^n) \subseteq \mathcal{O}(1.4253^n)$. If $|B| = 2, |C| = 3$ we note that the second branch can be expanded to four branches and we get a $\mathcal{O}(\tau(5;5,6,6,7;3)^n) \subseteq \mathcal{O}(1.4276^n)$ running

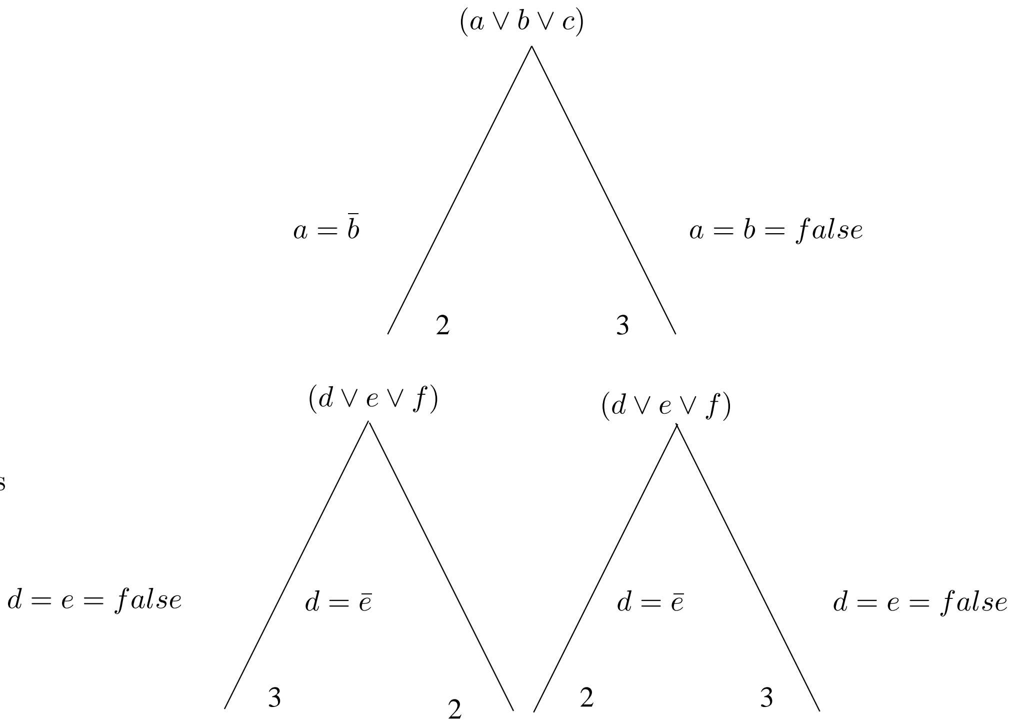

Figure 4.5: *Two steps of D_2 applying the first case on* XSAT *clauses of length 3; the numbers indicate how many variables are removed*

time. If $|B| = |C| = 3$ we may expand each of the three branches, the first to two branches, the second to eight branches and the third to two, thereby obtaining a running time in $\mathcal{O}(\tau(9,8;8,8,7,7,7,6,9,8;8,7)^n) \subseteq \mathcal{O}(1.3993^n)$. When we look back in the complexity analysis we see that the worst case possible for any XSAT clause or any X$_2$SAT clause shorter than 5 is that in one branch 2 variables are removed and in the other 3. As this was the case for all extra branchings when $|B| = |C| = 3$ we are now done with the subcase of $|A| = 3$.

(c) For $|A| = 4$, we note that due to the canonical rules it holds that $|x| + |y| > 10$ so the first subcase to consider is $|B| = 1, |C| = 2$. As a matter of fact, we will have only two branches, because in the third branch the formula will immediately be found unsatisfiable by the canonisation. Hence this case runs in $\mathcal{O}(\tau(3,2)^n) \subseteq \mathcal{O}(1.3248^n)$ time. If $|B| = |C| = 2$ a naïve analysis show that we have a running time in $\mathcal{O}(\tau(4,2,4)^n) \subseteq \mathcal{O}(1.4143^n)$. For $|B| = |C| = 2$ we also have the bound $\mathcal{O}(\tau(4,2,4)^n) \subseteq \mathcal{O}(1.4143^n)$. For $|B| = 2, |C| = 3$ we get the bound $\mathcal{O}(\tau(5;4,4;4)^n) \subseteq \mathcal{O}(1.3888^n)$. If $|B| = |C| = 2$, then we get a running time in $\mathcal{O}(\tau(6;3,3;4)^n) \subseteq \mathcal{O}(1.4459^n)$ and this is clearly the worst case for $|A| = 4$.

(d) For $|A| = 5$, when $|B| = 1, |C| = 2$, we get a running time in $\mathcal{O}(\tau(3,2,5)^n) \subseteq \mathcal{O}(1.4300^n)$. The case $|B| = |C| = 2$ gives a bound $\mathcal{O}(\tau(4,2,5)^n) \subseteq \mathcal{O}(1.3803^n)$. If $|B| = 2$ and $|C| = 3$, then we get a running time in $\mathcal{O}(\tau(5,1,5)^n) \subseteq \mathcal{O}(1.4511^n)$. For $|B| > 2$ and $|C| = 3$ we have a bound $\mathcal{O}(\tau(6;4,2;5)^n) \subseteq \mathcal{O}(1.4352^n)$ (the second branch is expanded by making use of the clause $(A)^{\mathrm{I}}$.) The other cases are all better than this last one. We also see that for $|A| > 5$ we will have no case worse than the ones already analysed.

8. We know that there are X$_2$SAT clauses y, y' and z such that

$a \in y$, $a \in y'$, $b \in z$ and $b \in z'$ which, by the earlier cases, are all different from x, do not share any other variables than a and b and are at least 5 in length. That means that there is a subset of the formula looking like this ($\dot{a}$ indicates a or $\bar{a}$):

$$y = (\dot{a} \vee c \vee c' \vee C)^{\text{II}}$$

$$y' = (\dot{a} \vee d \vee d' \vee D)^{\text{II}}$$

$$x = (a \vee b \vee a' \vee a'' \vee A)^{\text{II}}$$

$$z = (\dot{b} \vee e \vee e' \vee E)^{\text{II}}$$

$$z' = (\dot{b} \vee f \vee f' \vee F)^{\text{II}}$$

We will examine the cases depending on $|x|$, using the same expanded view as in the previous case:

(a) If $|x| = 5$, then there are five variants depending on the actual look of $\dot{a}$ and $\dot{b}$ — none of the dotted variables is negated, one is negated, etc.

None of the dotted is negated: This case runs in $\mathcal{O}\left(\tau(21, 19^4, 17^6, 15^4, 13; 3, 4; 4, 5)^n\right) \subseteq \mathcal{O}\left(1.4413^n\right)$ time. The first branch can be extended into 16 branches, taking care of the four XSAT clauses created by $a = b = true$. The first figure, 21, is 5 (obtained from x) $+4+4+4+4$ (obtained from the other four clauses.) Note that due to the balanced branching effect, the worst case will be when $|y| = |y'| = |z| = |z'| = 6$.

One of the dotted is negated: This case runs in $\mathcal{O}\left(\tau(17, 15^3, 13^3, 11; 3, 4; 8, 6, 9, 7)^n\right) \subseteq \mathcal{O}\left(1.4138^n\right)$ time.

Two of the dotted are negated: This case runs in $\mathcal{O}\left(\tau(13, 11, 11, 9; 3, 4; 12, 10, 10, 8, 13, 11, 11, 9)^n\right) \subseteq \mathcal{O}\left(1.4001^n\right)$ time.

Three of the dotted are negated: This case runs in $\mathcal{O}\left(\tau(9, 7; 3, 4; 16, 14^3, 12^3, 10, 17, 15^3, 13^3, 11)^n\right) \subseteq \mathcal{O}\left(1.3934^n\right)$ time.

All of the dotted are negated: For this case we will have a running time in $\mathcal{O}\left(\tau(5; 3, 4; 20, 18^4, 16^6, 14^4, 12, 21, 19^4, 17^6, 15^4, 13)^n\right) \subseteq \mathcal{O}\left(1.3920^n\right)$.

(b) For $|x| = 6$ we can of course make the same subcase analysis as above, however, by now we have seen that due to the balanced branching effect, we only need to look at the case when no dotted is negated. This case has a running time in $\mathcal{O}\left(\tau(22, 20, 20^3, 18^6, 16^4, 14; 4, 4; 4, 5)^n\right) \subseteq \mathcal{O}\left(1.4396^n\right)$.

(c) For $|x| = 7$, in the third branch, when $a = b = false$, there will be a $\mathrm{X_2SAT}$ clause of length 5 created. We will not expand that branch and so we get a running time in $\mathcal{O}\left(\tau(23, 21^4, 19^6, 17^4, 15; 5, 3; 2)^n\right) \subseteq \mathcal{O}\left(1.4400^n\right)$ time. Clearly, there is no need to examine the cases when $|x| > 7$ as they will all be better than this one.

9. As each clause has length at least 5 and contains at most one heavy variable, the ratio heavy variables to the total number of variables is at most 2/6. Thus, this case runs in time $\mathcal{O}\left(2^{1/3}\right) \subseteq \mathcal{O}\left(1.12600^n\right)$.

Inspecting all the above cases, we conclude that the overall running time is in $\mathcal{O}\left(1.4511^n\right)$.

□

4.4 Solving X_iSAT in Exponential Space

From the above algorithms and properties presented, it seems reasonable that the running time of an algorithm for deciding X_iSAT should always depend heavily upon the actual i. However, that is not the case. In the early 1980's, Shroeppel and Shamir found a way to solve a class of *NP*-complete problems in time $\mathcal{O}\left(2^{n/2}\right) \subseteq \mathcal{O}\left(1.4143^n\right)$ and space $\mathcal{O}\left(2^{n/4}\right) \subseteq \mathcal{O}\left(1.1893^n\right)$. There has been a recent interest in this kind of algorithms, for instance, Williams' [77] exponential space algorithm for Max 2sat and related problems.

One member of this class is Xsat, and in order to illustrate the algorithm, the reader may have a look at Figure 4.6. The formula shown contains 5 variables. We now divide these into four groups. Within each group we test all possible assignments to the variables and for each such assignment we make a multiset of the clauses that

$$x = (a \vee b \vee c \vee e), y = (\bar{a} \vee d \vee b), z = (c \vee b)$$

a	e	$S(a,e)$	b	$S(b)$	c	$S(c)$	d	$S(d)$
0	0	$\{y\}$	0	$\{\}$	0	$\{\}$	0	$\{\}$
0	1	$\{x, y\}$	1	$\{x, y\}$	1	$\{x, z\}$	1	$\{y\}$
1	0	$\{x\}$						
1	1	$\{x, x\}$						

Figure 4.6: *Solving* XSAT *in exponential space*

become exactly satisfied. We then proceed by picking one multiset from each group such that adding all these multisets, we get a multiset $\{x, y, z\}$, i.e. all clauses are satisfied exactly once. In the example, we pick the first multiset from $S(a, e)$, the first from $S(b)$, the second from $S(c)$ and the first from $S(d)$. We see that this corresponds to the model $a =$ false, $e =$ false, $b =$ false, $c =$ true, $d =$ false.

There are two conditions a problem must satisfy in order for the algorithm of Shroeppel and Shamir to be applicable: First, that given a solution (or rather an assignment in the solution space), the problem instances satisfied by the solution must be enumerable in polynomial time and space, and second, that a problem instance can be split in a way such that the split operation enjoys certain algebraic properties.

One could think that the enumerability requirement makes the algorithm inapplicable to problems involving Boolean formulae (given an assignment one can construct an infinite set of formulae for which that is a model.) However, as explained by Shroeppel and Shamir, we may consider the formula fixed (namely the input formula) and so the requirement boils down to simple evaluation.

In the context of X$_i$SAT, a possible implementation of their algorithm is just extending the XSAT algorithm above: If we want two literals true in y, we want the resulting overall multiset to contain two occurrences of y etc. The multiset can be represented by a list indicating the number of true literals in each clause.

To make this a little bit more formal: The X$_i$SAT instance is described by a list of variables and a list of numbers indicating for

each clause how many true literals it needs to be satisfied, i.e. i. Now split the variable list in four parts and for each part, tabulate all possible assignments to the variables, and for each assignment make a list indicating for each clause how many literals become true. We now have four tables and want to scan them to see if there are four lists which can be piece-wise added so that the list $(i, i, i, \ldots)$ is obtained. If these four lists are found the formula has an X_iSAT model. We will not go into details on how the search is done. Instead, we will restate the main theorem of Shroeppel and Shamir and then prove that the split-operation described for X_iSAT makes the theorem applicable. First however, we need to define the following:

A binary operator $\oplus$ is a *monotonic composition operator* iff

1. for all problem instances P' and P'', $|P' \oplus P''| = |P'| + |P''|$
2. for any two solutions x' to P' and x'' to P'' there is a simple concatenation $x'x''$ which is a solution to $P' \oplus P''$.
3. for every solution x to P any any representation of x as $x'x''$, there are problems P' and P'' such that x' solves P', x'' solves P'' and $P = P' \oplus P''$.
4. $P' \oplus P''$ can be computed in polynomial time (in the lengths of P' and P''.)
5. There is a total ordering $<$ such that $|P'| = |P''|$ and $P' < P''$ imply that $P' \oplus P < P'' \oplus P$ and $P \oplus P' < P \oplus P''$.

Theorem 14. (Shamir and Shroeppel) If a set of problems is polynomially enumerable and has a monotonic composition operator, then its instances of size n can be solved in time $T = \mathcal{O}\left(2^{n/2}\right)$ and space $S = \mathcal{O}\left(2^{n/4}\right)$.

In the above described representation of X_iSAT the requirement of polynomially enumerability is satisfied as, for a fixed formula, simple evaluation reveals whether an assignment satisfies the formula. Furthermore, in our case, $\oplus$ is the concatenation of the variable lists of P' and P'' and the piece-wise addition of the list of numbers. It is

easy to see that it is monotonic (for property 5 use lexicographical ordering.)

Two final remarks here: As pointed out already by Shamir and Shroeppel, SAT cannot be solved by this method, because there is no monotonic composition operator (property 5 fails.) Furthermore, the method requires that the size of the formula is a polynomial in the number of variables.

Part III

Counting

Chapter 5

Counting Exact Satisfiability

This chapter deals with the counting problems corresponding to the decision problems XSAT and $X_1$3SAT. We first give some extra preliminaries needed in this and the following chapter. We then present the algorithms, prove them correct and analyse them with respect to worst case time complexity.

5.1 Preliminaries

Recall that two clauses are said to *connect* if there is at least one variable p occurring in both clauses. If p occurs as the same literal, the connection *is in the same sign*. We say that the clauses x and y n-connect iff they share n variables. This is a generalisation of the concept of overlapping clauses (the reader may recall that two clauses overlap if they share at least two variables.)

The canonical rule 6 can be used to remove superfluous singletons. Say for instance that there is a clause $(a \vee b \vee c)$ such that a and b are singletons. The choice of variable to remove — a or b — is rather arbitrary. For every model where $a = true$ and $b = false$ there is a model where $a = false$ and $b = true$. This means that in order to use the canonical rules in a *counting* algorithm we must introduce

$$x = (a \vee b \vee c), y = (a \vee d)$$

$M_1 : \{a = true, b = false, c = false, d = false\}$
$M_2 : \{a = false, b = true, c = false, d = true\}$
$M_3 : \{a = false, b = true, c = false, d = true\}$

Figure 5.1: *In this* XSAT *instance, b and c are singletons. As can be seen, there are three models*

additional structures to keep track of removed variables. For this we have the following:

To each literal a_i a variable $c(a_i) \in N$ is associated; the vector c containing these weights is called a *cardinality vector*. Its purpose is to keep track of the *number* of maximum weighted models.

Similarly to the cardinality vector, we introduce a *weight vector* for keeping track of the contribution to the *weight* of the models arising from eliminated variables.

Let F be a formula, and let L be the set of all literals for all variables occurring in F. Given a weight vector w and a model M for F, we define the weight of M as

$$\mathcal{W}(M) = \sum_{\{l \in L \mid l \text{ is true under } M\}} w(l)$$

We use the acronym MWM for *maximum weighted model.*

Given a cardinality vector c and a model M for F, we define the cardinality of M as

$$\mathcal{C}(M) = \prod_{\{l \in L \mid l \text{ is true under } M\}} c(l)$$

This intuitively means that in a DPLL style canonising algorithm, "remaining" models carry the information about symmetrical models that have been removed using canonisation.

To give an example of the use of the cardinality vector, we have the formula of Figure 5.1. For clarity of presentation, we will not consider weights here. Initially, each entry of the cardinality vector holds 1.

Now, assume that we want to remove the superfluous singleton c. Then the remaining singleton b must represent both. In a model where b is true, c is false. However, we know that there is a model where b is false and c is true. When b is false, c must also be false. Therefore, when removing c, we update the cardinality vector in this way: $c(\bar{b}) \leftarrow c(\bar{b}) \cdot c(\bar{c}) = 1$ and $c(b) \leftarrow c(b) \cdot c(\bar{c}) + c(\bar{b}) \cdot c(c) = 2$. The formula now consists of the clauses $x = (a \vee b), y = (a \vee d)$. Assigning a true, we get a model with cardinality $c(a) \cdot c(\bar{b}) \cdot c(\bar{d}) = 1$; assigning a false we get a model with cardinality $c(\bar{a}) \cdot c(b) \cdot c(\bar{d}) = 2$. Thus, we see that the number of models has not changed by the removal of c.

We will now give the following definition of #XSAT$_w$, which somewhat modifies our previous definition. Let M' be an arbitrary MWM for F if any exist and $S(F)$ denote the set of x-models for F:

– COUNTING WEIGHTED EXACT SATISFIABILITY (#XSAT$_w$):

  Instance: A formula F, a cardinality vector c and a weight vector w.

  Question: What is the tuple $\left(\sum_{M \in S(F)} \mathcal{C}(M), \mathcal{W}(M')\right)$?

If F has no x-models, then the tuple is $(0, 0)$.

We will assume that canonisation is performed in the function *Prop*, which takes a formula F and applies canonical rules. When a variable a is removed its weight (or rather the weight of the literals a and $\bar{a}$) must be preserved and similarly for the cardinality. We have seen one example of how the cardinality is preserved when removing singletons. However, as we will see, there are other situations that require other solutions. *Prop* returns a tuple (F', C, W, c', w') such that F' is the canonical formula obtained from F, C is a multiplicative contribution to the number of MWM's coming from removed variables, W is a similar (additive) weight contribution to the maximum weight, and c' and w' are modified cardinality and weight vectors.

The following rules of Chapter 2 will be used: 2 – 7, 9 and 11. Rule 6, however, is weakened to the following:

No clause contains more than one singleton.

For a clause $(a \vee b \vee \ldots)$ such that a and b are singletons we remove a.

Rule 9 is weakened to the following:

There is no pair of clauses $x = (A \vee a)$ and $y = (A \vee b)$.

If there are two clauses $x = (A \vee a)$ and $y = (A \vee b)$, then let $F \leftarrow F(a/\bar{b})$.

In case an inconsistency is discovered, the tuple $(\{()\}, 0, 0, c, w)$ is returned. When we substitute a constant or expression for a literal in F, we need to update the structures accordingly. Below let a and b denote any literals. The first thing that happens in *Prop* is that W and C are initialised to 1. We then apply the canonical rules and the following modifications:

1. When we set $a \leftarrow$ false ($a \leftarrow$ true), we set $C \leftarrow C \cdot c(\bar{a})$ ($C \leftarrow C \cdot c(a)$) and $W \leftarrow W + w(\bar{a})$ ($W \leftarrow W + w(a)$.)

2. If a variable a is removed without being assigned a definite value (such as might be the case in rule 5, second case, where a clause $w = (a \vee \bar{a} \vee \ldots)$ is removed without assigning a a value) we have three cases:

   (a) if $w(a) = w(\bar{a})$, then let $W \leftarrow W + w(a)$ and $C \leftarrow C(c(a) + c(\bar{a}))$

   (b) if $w(a) < w(\bar{a})$, then let $W \leftarrow W + w(\bar{a})$ and $C \leftarrow C \cdot c(\bar{a})$

   (c) if $w(a) > w(\bar{a})$, then let $W \leftarrow W + w(a)$ and $C \leftarrow C \cdot c(a)$

3. When applying rule 4 (for 2-clauses), removing a we set $w(b) \leftarrow w(b) + w(\bar{a})$, $c(b) \leftarrow c(b) \cdot c(\bar{a})$, $w(\bar{b}) \leftarrow w(\bar{b}) + w(a)$ and $c(\bar{b}) \leftarrow c(\bar{b}) \cdot c(a)$.

4. When applying the weakened rule 6 removing a, we set $c(\bar{b}) \leftarrow c(\bar{b}) \cdot c(\bar{a}), w(\bar{b}) \leftarrow w(\bar{b}) + w(\bar{a})$ and then we have three cases:

   (a) if $w(b) + w(\bar{a}) = w(\bar{b}) + w(a)$, then let $w(b) \leftarrow w(b) + w(\bar{a})$ and $c(b) \leftarrow c(b) \cdot c(\bar{a}) + c(\bar{b}) \cdot c(a)$.

(b) if $w(b) + w(\bar{a}) < w(\bar{b}) + w(a)$, then let $w(b) \leftarrow w(\bar{b}) + w(a)$ and $c(b) \leftarrow c(\bar{b}) \cdot c(a)$.

(c) if $w(b) + w(\bar{a}) > w(\bar{b}) + w(a)$, then let $w(b) \leftarrow w(b) + w(\bar{a})$ and $c(b) \leftarrow c(b) \cdot c(\bar{a})$.

Lemma 15. Let $(F', C, W, c', w') \leftarrow Prop(F, c, w)$ and $(C', W') \leftarrow \#\textsc{xsat}_w(F', c', w')$. Then, $(C \cdot C', W + W') = \#\textsc{xsat}_w(F, c, w)$.

Proof. By Lemma 2 we know that the canonical rules are correct for deciding the existence of a model. Let us now inspect the modifications above:

1. Obviously correct.

2. We choose the value of a such that a MWM is obtained.

3. If a is true b must be false and vice versa.

4. The remaining singleton b represents both singletons. If b is false, then a must also be false. We then have the three cases when a singleton is true: If both possibilities have equal weight, we add their cardinalities. Otherwise, we keep only the maximum weighted possibility.

□

Because of the bookkeeping involved in using the $Prop(F, c, w)$ auxiliary function and the c and w vectors, the actual process of branching on a variable or performing an assignment and making a number of recursive calls in the algorithms is somewhat lengthy, and will not be written explicitly in the algorithms. Instead we will use the phrase *(recursively) branch on* $F_1, \ldots, F_k$ as a shorthand for the procedure below. Before presenting it we need some more preliminaries.

We say that given a formula F and a set $\Gamma = \{F_1, \ldots, F_k\}$ of formulae, such that $Var(F_i) \subseteq Var(F)$ for the members F_i of Γ, Γ *covers* F if the following holds:

1. For different $F_i, F_j \in \Gamma$, F_i and F_j have no models in common.

2. Every model for F is a model for some F_i.

For an example, consider any formula F and $F_1 = F(a/true)$, $F_2 = F(a/false)$. Clearly, F_1 and F_2 have no models in common (because they assign different values to a), and every model for F is a model for either F_1 or F_2. For XSAT we have more complicated situations. For instance, if F contains the clause $(a \vee b \vee c)$, then $F_1 = F(a/true), F_2 = F(b/true), F_3 = F(c/true)$ cover F.

We assume that our algorithm, here denoted β, is given a formula F and that $F_1, \dots, F_k$ cover F. Let $w_A(F_i)$ denote the weight of the literals that are explicitly created when F_i is derived from F, for instance, $w_A(F(a/false)) = w(\bar{a})$; $c_A(F_i)$ is defined accordingly, i.e. $c_A(F(a/false)) = c(\bar{a})$. Now, *branching on* $F_1, \dots, F_k$ refers to the following:

1. Let $(F_1, C_1, W_1, c_1, w_1) \leftarrow Prop(F_1, c, w), \dots, (F_k, C_k, W_k, c_k, w_k) \leftarrow Prop(F_k, c, w)$.

2. Let $(C'_1, W'_1) \leftarrow \beta(F_1, c_1, w_1), \dots, (C'_k, W'_k) \leftarrow \beta(F_k, c_k, w_k)$.

3. Let $W_1 \leftarrow w_A(F_1) + W_1 + W'_1, W_2 \leftarrow w_A(F_2) + W_2 + W'_2, \dots, W_k \leftarrow w_A(F_k) + W_k + W'_k$ and $C_1 \leftarrow c_A(F_1) \cdot c_1 \cdot c'_1, C_2 \leftarrow c_A(F_2) \cdot c_2 \cdot c'_2, \dots, C_k \leftarrow c_A(F_k) \cdot c_k \cdot c'_k$.

4. Let $W_{heavy} \leftarrow \{W_I, W_{II}, \dots, W_j\}$ be the (possibly singleton) set of maximum valued W's and return $(C_I + C_{II} + \dots + C_j, W_I)$.

Intuitively, the first line will canonise all the formulae to branch on; the second line performs all the recursive calls on these now canonical formulae; the third line "glues together" the result of canonising formula F_i with the result of recursively solving the canonical F_i; in the fourth line the MWM's are collected. We need a lemma for the correctness of this construct:

Lemma 16. The result of recursively branching on $F_1, \dots, F_k$ given the formula F equals $\#\text{XSAT}_w(F, c, w)$.

Algorithm $\#D_3(F, c, w)$

1. If $n(F) \leq 9$, then return $\#D_E(F, c, w)$
2. If F is not connected, then assuming the connected components are $F_1, \ldots, F_k$, let $(C_i, W_i) \leftarrow \#D_3(Prop(F_i, w))$ for each of the connected components and return $(\prod C_i, \sum W_i)$
3. Pick any c such that $\delta(c) \geq 3$ and branch on $F(c/true)$ and $F(c/false)$
4. Pick any non-constant c and branch on $F(c/true)$ and $F(c/false)$
5. For two clauses $x = (a \vee b \vee c)$ and $y = (c \vee d \vee e)$ such that x contains no singleton, branch on $F(c/true)$ and $F(c/false)$
6. Pick any non-singleton variable c and branch on $F(c/true)$ and $F(c/false)$

Figure 5.2: *The algorithm $\#D_3$ for solving #X3SAT*

Proof. Lines 1 and 2 are straightforward. Line 3 is correct by Lemma 15. Line 4 is correct by the fact that $F_1, \ldots, F_k$ cover F, and the observation that we choose the MWM's only. □

5.2 Algorithm for $\#\text{X3SAT}_w$

In this section we will present the algorithm $\#D_3$ for $\#\text{XSAT}_w$ when the clause length is at most 3. We assume there will be an auxiliary algorithm $\#D_E$ that performs an exhaustive search on smaller instances of bounded size in order to solve the problem. The algorithm is shown in Figure 5.2.

For the correctness of $\#D_3$ we have the following theorem:

Theorem 17. $\#\text{XSAT}_w(F, c, w) = \#D_3(F, c, w)$.

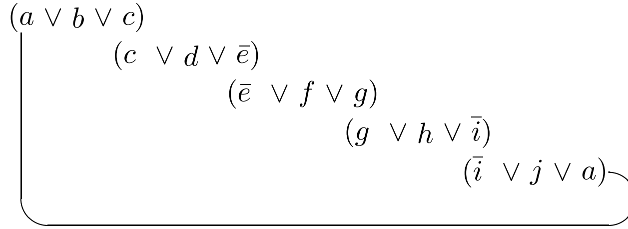

Figure 5.3: *A canonical formula where each clause connects with two other, and each connection is in the same sign*

Proof. We look at the cases of $\#D_3$:

1. This case is correct by assumption.

2. Having connected components $F_1, \dots, F_k$, any MWM of F_i can be combined with any MWM of F_j $(i \neq j)$.

3. This and the remaining cases are correct by Lemma 16.

□

As usual, we want as many clauses as possible to be removed by an assignment of a variable. The following lemma deals with the clauses surrounding any pair of clauses, and it will come in handy in the complexity analysis:

Lemma 18. For any two connected 3-clauses x and y in a canonical formula, there are clauses x' and y' such that x connects with x' and y connects with y'.

Proof. For a 3-clause z to connect with only one other clause z', the connection must be in two variables since there are no clauses having two singletons, by the weakened rule 6. However, this cannot be, as rule 9 prohibits two 3-clauses to share two variables. This means that every clause connects with at least two different clauses and so the lemma follows. □

$$x = (a \vee b \vee c)$$
$$y = (c \vee d \vee e)$$
$$x' = y' = (a \vee f \vee e)$$

Figure 5.4: *A canonical formula where each clause connects with two other*

Note that it might be that $x' = y'$, as in Figure 5.4. This will create some extra cases to consider in the analysis.

Theorem 19. $\#D_3$ runs in $\mathcal{O}(\tau(5,5)^n) \subseteq \mathcal{O}(1.1487^n)$ time.

Proof. We look into the cases of $\#D_3$ to find the one which gives the upper time bound. Let $T(n)$ denote the running time of $\#D_3$. Note that as all clauses have length at most three, a shrinking clause means that at least one variable is removed. For clarity we will often show how a 3-clause shrinks to a 2-clause. We say that an assignment *yields* a certain number of removed variables and shrunken clauses.

1. This case takes constant time, as the exhaustive search will be applied only to instances of fixed size.

2. Connected components will not increase the running time.

3. When c participates in three clauses, x, y and z, we know that there is no other connection among these clauses, since the instance is canonical. There are two cases:

   (a) c has the same sign in all clauses: $x = (a \vee b \vee c), y = (c \vee d \vee e), z = (c \vee f \vee g)$. The assignment $c = true$ yields $\{\bar{a}, \bar{b}, \bar{d}, \bar{e}, \bar{f}, \bar{g}\}$. Furthermore, since the three clauses are not the whole of the instance (by case 1) and are connected (by case 2) there will be at least one more clause v connecting with x, y or z. In the worst case v will not be satisfied, but it will anyhow shrink. The second branch, $c = false$, yields $\{(a \vee b), (d \vee e), (f \vee g)\}$. Thus, $T(n) \in \mathcal{O}(\tau(8,4)^n)$.

(b) c has not the same sign in all clauses: $x = (a \vee b \vee c), y = (c \vee d \vee e), z = (\bar{c} \vee f \vee g)$. The assignment $c = true$ yields $\{\bar{a}, \bar{b}, \bar{d}, \bar{e}, (f \vee g)\}$ and at least two more shrunken clauses while $c = false$ yields $\{(a \vee b), (d \vee e), \bar{f}, \bar{g}\}$ and at least one more shrunken clause; $T(n) \in \mathcal{O}(\tau(7,5)^n)$.

It is obvious that c occurring in three clauses is the worst case for c participating in more than two clauses.

4. Let x and y be the clauses containing c and $\bar{c}$ and x' and y' the clauses in contact with x, y, which we know exist by Lemma 18:

   (a) $x' = y'$. If x and y connect with x' in one variable each, say a and e, then the worst case will have $x' = (a \vee e \vee f)$ and then $c = true$ yields $\{\bar{a}, \bar{b}, (d \vee e), (e \vee f)\}$ and $c = false$ yields $\{\bar{d}, \bar{e}, (a \vee b), (a \vee f)\}$; $T(n) \in \mathcal{O}(\tau(5,5)^n)$.

   (b) $x' \neq y'$. The clause satisfied by c will yield $\{\bar{a}, \bar{b}\}$ and since at least one of these participates in another clause, that clause will at least shrink. The clause containing $\bar{c}$ will shrink as well. That removes at least five variables and the branch for $\bar{c}$ is symmetric; $T(n) \in \mathcal{O}(\tau(5,5)^n)$.

5. We know that there are two other clauses x' and x'' connecting with x and by Lemma 18 there is one clause y' (and possibly another clause y'') connecting with y:

   (a) $x' = y' = (a \vee d \vee f), x'' = y'' = (b \vee e \vee g)$. We know that $f \neq g$ holds as the instance is connected, the instance contains more than three clauses and no variable occurs more than twice. The assignment $c = true$ yields $\{\bar{a}, \bar{b}, \bar{d}, \bar{e}, f, g\}$ and since either f or g (or both) occur in another clause z, the two other variables of z will be removed and in turn cause other variables to be removed or clauses to shrink. The assignment $c = false$ yields $\{(a \vee b), (d \vee e), \bar{f}, \bar{g}\}$ and at least one shrunken clause; $T(n) \in \mathcal{O}(\tau(11,6)^n)$.

(b) $x' = y' = (a \vee d \vee f)$, $x'' = (b \vee g \vee h)$. The assignment $c = true$ yields $\{\bar{a}, \bar{b}, \bar{d}, \bar{e}, f, (g \vee h)\}$ and since either d or f connects with another clause z, the two other variables of z are removed which causes at least two more clauses to shrink. The assignment $c = false$ yields $\{(a \vee b), (d \vee e)\}$; $T(n) \in \mathcal{O}(\tau(10,3)^n)$.

(c) $x' = (a \vee f \vee g) \neq y' = (e \vee h \vee i)$, $x'' = (b \vee j \vee k)$. The assignment $c = true$ yields $\{\bar{a}, \bar{b}, \bar{d}, \bar{e}, (f \vee g), (j \vee k), (h \vee i)\}$ and $c = false$ yields $\{(a \vee b), (d \vee e)\}$; $T(n) \in \mathcal{O}(\tau(8,3)^n)$.

6. As for the last case, each clause connects with exactly two other clauses, and the connection is in the same sign — a circle-like form, see Figure 5.3. The substitution $F(c/true)$ will open the circle and the ending clauses will contain two singleton variables each. *Prop* will then consume the entire formula. The other branch works similarly. By Lemma 1 this case runs in polynomial time.

Straightforward calculations give that $\mathcal{O}(\tau(8,3)^n) \subseteq \mathcal{O}(1.1461^n)$, $\mathcal{O}(\tau(5,5)^n) \subseteq \mathcal{O}(1.1487^n)$ and the other cases are dominated by these two cases. Thus, $\#D_3$ runs in $\mathcal{O}(1.1487^n)$ time. ☐

5.3 Algorithm for $\#\text{XSAT}_w$

In this section we will present $\#D$ which counts models for instances of arbitrary clause length. Its correctness and a bound for its running time will also be proved. The algorithm can be seen in Figure 5.5.

Theorem 20. $\#\text{XSAT}_w(F, c, w) = \#D(F, c, w)$.

Proof. The correctness proof of $\#D(F, w)$ is similar to the one for Theorem 17. The only thing that needs justification is that the cases of the algorithm are actually exhaustive, i.e. all possibilities are covered.

Let us study the possible structure of the instance after each case: After case 3 all variables must be constant; after case 4 all clauses

Algorithm $\#D(F, c, w)$

1. If $|F| \leq 9$, then return $\#D_E(F, c, w)$
2. If F is not connected, then assuming the connected components are $F_1, \ldots, F_k$, apply $(C_i, W_i)\#D(Prop(F_i, w))$ to each of the connected components and return $(\prod C_i, \sum W_i)$
3. Pick any non-constant variable c and branch on $F(c/true)$ and $F(c/false)$
4. Pick a variable c and a variable d such that there are two clauses x and y where $c, d \in x$ and $c, d \in y$; branch on $F \cup (c \vee d)$ and $F(c/false; d/false)$
5. Pick any non-singleton variable c in a clause x of length 3 and branch on $F(c/true)$ and $F(c/false)$
6. Pick any non-singleton c occurring in no clause of length 4 and branch on $F(c/true)$ and $F(c/false)$
7. Pick a clause $(a \vee b \vee c \vee d)$ where all variables are non-singletons and branch on $F(a/true), F(b/true), F(c/true)$ and $F(d/true)$
8. The instance contains this structure: $(a \vee b \vee c \vee d), (a \vee C_1), (b \vee C_2), (c \vee e \vee f \vee g)$, where d and e are singletons and $|C_1|$ and $|C_2|$ are at least 3; branch on $F(a/true), F(b/true)$ and $F \cup (c \vee d)$.

Figure 5.5: *The algorithm $\#D$ for $\#\text{XSAT}$*

connect in one variable only; after case 5 all clauses have length at least 4; after case 6 each clause either has length 4 or has length ≥ 5 and connects only with clauses of length 4; after case 7 each clause of length 4 has one singleton and each such clause must connect with at least one other clause of length 4 (since the longer clauses do not connect.) Thus, for the instances remaining, case 8 will be applicable and the algorithm is correct. □

Theorem 21. $\#D$ runs in $\mathcal{O}(\tau(7,7,7,7)^n) \subseteq \mathcal{O}(1.2190^n)$ time.

Proof. We look at the non-trivial cases:

3. The worst case here is obviously when c appears in only two clauses $c \in x$ and $\bar{c} \in y$.

   (a) $|x| = |y| = 3$: $c = true$ removes the two other variables in x and makes y shrink. The other branch is symmetric and so this case runs in $\mathcal{O}(\tau(4,4)^n)$.

   (b) $|x| = 3, |y| = 4$: $c = true$ removes the two other variables in x. The other branch removes the three other variables in y and makes x shrink. Thus, this case runs in $\mathcal{O}(\tau(3,5)^n)$.

   (c) $|x| \geq 4, |y| \geq 4$: $c = true$ removes the three other variables in x. The other branch is symmetric. Hence this case runs in $\mathcal{O}(\tau(4,4)^n)$. We also see that any increase of length in either x or y will make the running time even better.

4. The worst case here is $x = (a \vee b \vee c \vee d \vee e), y = (c \vee d \vee e \vee f \vee g)$ which runs in $\mathcal{O}(\tau(6,2)^n)$ time.

5. Due to the previous cases of $\#D$, we know that the connection is in one variable only and that the variables are constant. Let $x = (a \vee b \vee c)$. The case $|y| = 3$ runs in $\mathcal{O}(\tau(4,4)^n)$ time and the remaining cases run in $\mathcal{O}(\tau(|y|+1,3)^n)$. Hence the worst running time is $\mathcal{O}(\tau(5,3)^n)$.

6. Obviously, the worst case involves two clauses of length 5, which gives $\mathcal{O}(\tau(9,1)^n)$ time.

7. The worst case is when each of the non-singletons appears in one other 4-clause. This gives $\mathcal{O}(\tau(7,7,7,7)^n)$ time.

8. This case gives $\mathcal{O}(\tau(7,7,4)^n)$ time: For the 4 in $\tau(7,7,4)$, one removed variable comes from the substitution $F(c/\bar{d})$, two comes from $\bar{a}, \bar{b}$ and the last removed variable comes from $(\bar{d} \vee e \vee f \vee g)$ since c and e both are singletons and one of them will be removed by *Prop*.

Straightforward calculations give that $\mathcal{O}(\tau(4,4)^n) \subseteq \mathcal{O}(1.1892^n)$, $\mathcal{O}(\tau(7,7,4)^n) \subseteq \mathcal{O}(1.2085^n)$, $\mathcal{O}(\tau(3,5)^n) \subseteq \mathcal{O}(1.1939^n)$, $\mathcal{O}(\tau(6,2)^n) \subseteq \mathcal{O}(1.2106^n)$, $\mathcal{O}(\tau(9,1)^n) \subseteq \mathcal{O}(1.2131^n)$ and $\mathcal{O}(\tau(7,7,7,7)^n) \subseteq \mathcal{O}(1.2190^n)$. Thus the overall running time is in $\mathcal{O}(1.2190^n)$. $\square$

Chapter 6

Counting 2SAT and Counting 3SAT

This chapter treats $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$. We start with some additional preliminaries, definitions and technical tools in Section 6.1. Section 6.2 deals with a procedure *Prop* to simplify a formula while keeping track of certain information, similar to *Prop* in the previous chapter. Then follow Sections 6.3 and 6.4 on the algorithms for $\#2\text{SAT}_w$ and $\#3\text{SAT}_w$. In Section 6.5 we present an algorithm for counting in separable 2SAT formulae. Section 6.6 deals with applications for the algorithms.

6.1 Algorithm analysis

So far, when estimating the size of a branching tree in order to get an upper bound of the running time, we have used a simple variant of the method described by Kullmann [53]: As a measure we have simply used the number of variables. We have then analysed all possible worst cases and obtained a global worst case with a branching number α, and finally concluded that our algorithm runs in time $\mathcal{O}(\alpha^n)$.

In this chapter, we will use more complicated measures of formula complexity which enable us to obtain tighter worst case time bounds. For Kullmann's method of analysis to work, our measure $f(F)$ must

be non-negative. Fortunately, this (natural) restriction will be easy to enforce. In order to make Δf behave nicely we want f to be continuous. For the method to be useful, we want to be able to retranslate $f(F)$ to the standard measure $n(F)$. Defining $f_{\max}(n) = \max_{n(F)=n} f(F)$, this ensures a running time of $\mathcal{O}\left(\alpha^{f_{\max}(n)}\right)$, where α is the highest branching number of any branching tuple that can occur in the tree.

The ordinary measure of $n(F)$ contains little information in itself regarding the possible branchings — when $n(F)$ is sufficiently large any case can occur. Here we will use measures richer in such information. In our algorithm C for #2SAT$_w$, $f(F)$ will be a function $f(n, l)$ whose behaviour changes with the quotient l/n. This enables us to divide our analysis into cases depending on the average degree of a formula, and capture and quantify the beneficial effects of having an algorithm which ensures that the average degree of a formula will be gradually decreasing. In our algorithm D for #3SAT$_w$, $f(F)$ is a function $f(n, k)$, where k is the number of 2-clauses. As 2-clauses offer other possibilities for branching than 3-clauses, the number of 2-clauses is an important factor in the estimation of the size of the recursion tree.

In what follows, for any measure h, Δh (e.g. Δn or Δl) is understood to mean $h(F) - h(F')$ in the context of a branch from some formula F to some formula F'. If there are more than one possible F', say F_1 and F_2, then $\Delta_i h = h(F) - h(F_i)$.

6.2 Propagation

Similarly to the previous chapter, we will have a function *Prop* that performs polynomial time reductions of our formula F. We see that these reductions are in effect canonical rules. However, since that name is not used in the context of SAT, we will follow the conventions and just speak of reductions. Instead of saying that a formula is canonical we will here say that it is *maximally reduced.* The reductions are the following (we will later add one more):

1. **No clause contains the constants true or false.**

If F contains a clause $x = (true \vee C)$, then remove x.

If F contains a clause $x = (false \vee C)$, then let $x \leftarrow C$.

2. **There are no 1-clauses.**

   If F contains the clause (a), then let $F \leftarrow F(a/true)$.

3. **There is no pair of clauses x, y such that $x \subset y$.**

   If such clauses exist, then remove y.

4. **There is no pair of variables a, b such that a and b occur together in more than one 2-clause.**

   If there are two clauses $(a \vee b)$ and $(a \vee \bar{b})$, then let $F \leftarrow F(a/true)$. If there are two clauses $(a \vee b)$ and $(\bar{a} \vee \bar{b})$, then let $F \leftarrow F(a/\bar{b})$.

Just like in the previous chapter, when dealing with $\#\text{XSAT}_w$, we cannot use the reductions without extra structures. This is illustrated by the following: Let $F\{a = true\}$ denote substitution of *true* for a and reduction to simplify F. Then consider this example:

$$F = \{(a \vee b \vee c)\}$$

Note that $\#3\text{SAT}(F) = 7$. It would seem that $\#3\text{SAT}(F\{b = true\})+ \#3\text{SAT}(F\{b = false\}) = \#3\text{SAT}(F)$ but this is not the case. We see that the expression $(a \vee b \vee c)\{b = true\}$ is simplified to the empty formula (which has 1 model by definition) and $\#3\text{SAT}(F\{b = true\})+\#3\text{SAT}(F\{b = false\}) = 4$. The problem is that the variables a and c (which can be given arbitrary values) are *eliminated* in the simplification process.

To keep track of eliminated variables — the impact they have on the number of solutions as well as their weights — we will use the four structures of the previous chapter:

1. An integer variable $C \geq 0$ for keeping track of the contribution to the number of models arising from the eliminated variables.

2. A cardinality vector c.

3. An integer variable $W \geq 0$ for keeping track of the contribution to the weight of the models arising from the eliminated variables.

4. A weight vector w.

We will now give the following definition of #SAT$_w$, which somewhat modifies our previous definition. Let M' be an arbitrary MWM for F if any exist and $S(F)$ denote the set of maximum weighted models for F:

– COUNTING WEIGHTED SATISFIABILITY (#SAT$_w$):

  Instance: A formula F, a cardinality vector c and a weight vector w.

  Question: What is the tuple $\left(\sum_{M \in S(F)} \mathcal{C}(M), \mathcal{W}(M')\right)$?

If F has no models, then the tuple is $(0, 0)$.

The function *Prop* returns a tuple (F', C, W, c', w') such that F' is the maximally reduced formula obtained from F, C is a multiplicative contribution to the number of MWM's coming from removed variables, W is a similar (additive) weight contribution to the maximum weight, and c' and w' are modified cardinality and weight vectors. The four steps of the algorithm are performed until not applicable. Initially, let $W \leftarrow 0$ and $C \leftarrow 1$. We see that this is a procedure similar to *Prop* of the previous chapter. The modifications of the structures are also most similar:

1. When we set $a \leftarrow$ false ($a \leftarrow$ true), we set $C \leftarrow C \cdot c(\bar{a})$ ($C \leftarrow C \cdot c(a)$) and $W \leftarrow W + w(\bar{a})$ ($W \leftarrow W + w(a)$.)

2. If F contains an empty clause, then return $(\{()\}, 0, 0, c, w)$

3. If a variable a is removed without being assigned a definite value, then there are three cases:

   (a) if $w(a) = w(\bar{a})$, then $C \leftarrow C \cdot (c(a) + c(\bar{a}))$; $W \leftarrow W + w(a)$

   (b) if $w(a) < w(\bar{a})$, then $C \leftarrow C \cdot c(\bar{a})$; $W \leftarrow W + w(\bar{a})$

(c) if $w(a) > w(\bar{a})$, then $C \leftarrow C \cdot c(a)$; $W \leftarrow W + w(a)$

4. When we have two clauses $(a \vee b)$ and $(\bar{a} \vee \bar{b})$, removing a we set $w(b) \leftarrow w(b) + w(\bar{a})$, $c(b) \leftarrow c(b) \cdot c(\bar{a})$, $w(\bar{b}) \leftarrow w(\bar{b}) + w(a)$ and $c(\bar{b}) \leftarrow c(\bar{b}) \cdot c(a)$.

The following lemma extends of course to $\#2\text{SAT}_w$ as well. Its proof is almost identical to the proof of Lemma 15 and is therefore omitted.

Lemma 22. Let $(F', C, W, c', w') \leftarrow \mathit{Prop}(F, c, w)$ and $(C', W') \leftarrow \#3\text{SAT}_w(F', c',' w)$, then $\#3\text{SAT}_w(F, c, w) = (C \cdot C', W + W')$.

We will also use the phrase *recursively branch on* as in the previous chapter.

6.2.1 Multiplier Reduction

This subsection deals with an additional reduction rule. The reason for presenting it separately is that it is more complicated and requires more space for justification.

F cannot be partitioned into two formulae F_1 and F_2 such that $|Var(F_1) \cap Var(F_2)| = 1$, and such that that every clause in F belongs to either F_1 or F_2.

If this rule is violated, then assume γ is an algorithm for $\#2\text{SAT}_w$ or$\#3\text{SAT}_w$, $Var(F_1) \cap Var(F_2) = \{a\}$ and do the following:

1. $(c_t, w_t) \leftarrow \gamma(F_1(a/true), c, w)$, $(c_f, w_f) \leftarrow \gamma(F_1(a/false), c, w)$

2. Let $c(a) \leftarrow c_t \cdot c(a)$, $c(\bar{a}) \leftarrow c_f \cdot c(\bar{a})$, $w(a) \leftarrow w_t + w(a)$ and $w(\bar{a}) \leftarrow w_f + w(\bar{a})$

This procedure is referred to as removing F_1 by *multiplier reduction*, and if it is possible to partition F into F_1 and F_2 in this way, with $n(F_1), n(F_2) > 1$, then we say that *multiplier reduction applies.* In Figure 6.1 we see a formula where multiplier reduction applies.

$$
\begin{array}{ll}
(c \vee a) & \\
(d \vee a) & (a \vee b) \\
(\bar{c} \vee d) &
\end{array}
$$

Figure 6.1: *For this formula multiplier reduction applies*

Assuming that c' and w' are the vectors c and w as modified in the second step of the reduction, we have the following lemma (which of course also extends to $\#2\textsc{sat}_w$):

Lemma 23. $\#3\textsc{sat}_w(F, c, w) = \#3\textsc{sat}_w(F_2, c', w')$.

Proof. Suppose that F is partitioned into F_1 and F_2 with a as the common variable, and that F_1 is removed by multiplier reduction.

Every model M for F, with an assignment $a = \beta$, consists of a model M_1 for F_1 and a model M_2 for F_2, with both M_1 and M_2 assigning $a = \beta$. In other words, M consists of a model M_2 for F_2, assigning $a = \beta$, and a model $M_{1,\beta}$ for $F_1(a/\beta)$. Conversely, every model M_2 for F_2, assigning $a = \beta$, can be combined with a model $M_{1,\beta}$ for $F_1(a/\beta)$ into a model M for F. As $F_1(a/\beta)$ and F_2 have disjoint variable sets, $\mathcal{C}(M) = \mathcal{C}(M_{1,\beta}) \cdot \mathcal{C}(M_2)$ and $\mathcal{W}(M) = \mathcal{W}(M_{1,\beta}) + \mathcal{W}(M_2)$. The maximum $\mathcal{W}(M)$ that can be achieved by extending some particular M_2 assigning $a = \beta$ is $\mathcal{W}(M_2) + w_\beta$, and the weighted model count for the models for $M_{1,\beta}$ that achieve weight W_b is C_β, for a combined weighted model count for M of $\mathcal{C}(M_2) \cdot c_\beta$.

After the modifications to c and w have been made by multiplier reduction, $\mathcal{C}(M_2)$ and $\mathcal{W}(M_2)$ produce exactly these numbers for each model M_2 for F_2, which means that the final return value will be the same. □

Consider again the formula of Figure 6.1 and note that $F_1 = \{(a \vee b)\}$. This means that singletons are removed in polynomial time. We will not consider a formula maximally reduced if multiplier reduction applies.

6.3 Algorithm for #2SAT$_w$

Here we present the algorithm C for #2SAT$_w$, prove it correct and prove an upper time bound. We have three subsections for various degrees of F: the case $\delta(F) \leq 3$, the case $\delta(F) \leq 5$ and the general case.

6.3.1 The function C_3

The algorithm C is split into three functions depending on $\delta(F)$ of the input formula F. The main function C, given in Section 6.3.3, is used whenever $\delta(F) > 5$, and it has three auxiliary functions: C_5, given in Figure 6.4, which is used when $4 \leq \delta(F) \leq 5$, C_3, given in Figure 6.2, which is used when $\delta(F) \leq 3$. The auxiliary function C_E performs an exhaustive search on small instances in order to solve the problem. This subsection deals with the auxiliary function C_3. We first give a correctness lemma for it and then prove an upper limit on its running time.

Lemma 24. $C_3(F, c, w) = \#2\text{SAT}_w(F, c, w)$ when $\delta(F) \leq 3$.

Proof. We examine the cases of C_3:

1. Correct by assumption.

2. As every MWM of a component can be combined with every MWM of the other components this case is correct.

3. Correct by Lemma 23.

4. Correct by Lemma 16.

□

Lemma 25. For a maximally reduced formula F with $l \leq 2n$, we have a polynomial running time for $C_3(F, c, w)$.

Algorithm $C_3(F, c, w)$

1. If $n(F) < 10$, then return $C_E(F)$
2. If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = C(F_i, c, w)$ for the connected components $F_0, \ldots, F_j$.
3. If multiplier reduction applies, then apply it, removing the part with the lowest $n_3(F)$ value.
4. If $\delta(F) = 3$, pick a variable a, $\delta(a) = 3$, with as many neighbours of degree 3 as possible, and recursively branch on it. Otherwise, recursively branch on any variable.

Figure 6.2: *Auxiliary function for computing $\#2\textsc{sat}_w$ when $\delta(F) \leq 3$*

$$
\begin{array}{ccc}
 & (a \vee b) & \\
(a \vee d) & & (b \vee c) \\
 & (d \vee \bar{c}) &
\end{array}
$$

Figure 6.3: *A polynomial time solvable instance of $\#2\textsc{sat}$*

Proof. The existence of a singleton variable a in F implies that $n(F) = 2$, and this is surely polynomial time solvable by C_E.

Otherwise, every variable occurs exactly twice. This means that the constraint graph is either a cycle or a number of cycles. If there are several cycles they will each be solved separately, as they form disconnected components. Assume therefore that we have one cycle as in Figure 6.3. Assigning any literal true, we open the chain and singletons arise that will be taken care of, in turn creating new singletons. Assigning any literal false will also create singletons in a similar fashion. □

Lemma 26. $C_3(F, c, w)$ runs in $\mathcal{O}\left(\tau(4,4)^n\right) \subseteq \mathcal{O}\left(\tau(4,4)^{n_3(F)}\right) \subseteq \mathcal{O}\left(1.1892^{n_3(F)}\right)$ time.

Proof. We will derive the result by using the number of variables of degree 3 in a formula F, $n_3(F)$, as a measure: Clearly $\mathcal{O}\left(\tau(4,4)^n\right) \subseteq \mathcal{O}\left(\tau(4,4)^{n_3(F)}\right)$. We assume that F is maximally reduced, connected, and that there are no variables of higher degree than 3. By Lemma 25, if $n_3(F) = 0$, then we have a polynomial running time, so we will assume this is not the case. As F is maximally reduced, there are no singletons, and so, $n_3(F) = l(F) - 2n(F)$. We want to prove that $\Delta n_3 \geq 4$ along any branch F' and this is equivalent to proving that $\Delta l \geq 2\Delta n + 4$:

$$\begin{aligned}
\Delta n_3 &\geq 4 \\
n_3(F) - n_3(F') &\geq 4 \\
l(F) - 2n(F) - (l(F') + 2n(F')) &\geq 4 \\
l(F) - l(F') &\geq 2(n(F) - n(F')) + 4
\end{aligned}$$

We will assume that when branching on a variable a, we eventually end up with two maximally reduced formulae F_1 and F_2. It is of course possible that the branching results in more than two maximally reduced formulae, if case 2 or case 3 applies, but as we know, solving components separately will not give a worse upper time bound.

Let V be the variables of F, and V_1 the variables of F_1, let $V' = V - V_1$ and C be the clauses of F. Note that a clause $x = (a \vee b)$ in F_1 means that the same clause exists in F and that *both* $a, b \in V_1$, otherwise x would have been removed during the branching. Therefore, the reduction Δl in l is

$$\sum_{b \in V'} \delta(b) + |\{C' \in C \mid C' \text{ contains variables from both } V' \text{ and } V_1\}|$$

Since $\delta(a) = 3$ and since there are no singletons in F, the first term is at least $2\Delta n + 1$. As for the second term, note that F is maximally reduced, and since multiplier reduction does not apply, at least two variables from V_1 occur in clauses with variables from V', and so the second term is at least 2. We now know that we have at least $2\Delta n + 3$. Using that $l(F)$ is twice the number of clauses in F and therefore even, we have $\Delta l \geq 2\Delta n + 4$, so $n_3(F) - n_3(F_1) \geq 4$. The same argument applies to F_2. $\square$

Algorithm $C_5(F, c, w)$

1. If $n(F) < 10$, then return $C_E(F)$
2. If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = C(F_i, c, w)$ for the connected components $F_0, \ldots, F_j$.
3. If $\delta(F) < 4$, return $C_3(F, c, w)$.
4. If multiplier reduction applies, apply it, removing the part with lowest $f(F)$ value.
5. Pick a variable a of maximum degree such that $S(a)$ is maximised.
   (a) If $N(a)$ is connected to the rest of the graph through only 2 external vertices b and c such that $\delta(b) \geq \delta(c)$, then branch on b.
   (b) Otherwise, branch on a.

Figure 6.4: *Auxiliary function for computing* $\#2\text{SAT}_w$ *when* $4 \leq \delta(F) \leq 5$. *The function* f *is defined in the text; recall that* $S(a)$*, measuring the size of a neighbourhood* $N(a)$*, is defined by* $S(a) = \delta(a) + \sum_{b \in N(a)} \delta(b)$

6.3.2 The function C_5

This subsection treats the function C_5, given in Figure 6.4. The correctness comes from an argument very similar to that concerning the correctness of $C_3(F, c, w)$ and therefore we give no correctness theorem.

When it comes to the running time of C_5, we know that if $\delta(F) \leq 3$, then we have a running time in $\mathcal{O}(\tau(4,4)^n) \subseteq \mathcal{O}(1.1893^n)$. Doing a simple analysis, using only the number of variables as a measure, we could reason as follows:

We say that a branching is *maximally unbalanced* if the branching variable a appears only as a or $\bar{a}$. The worst case for a in C_5 must

be when $\delta(a) = 4$ and the branching is maximally unbalanced. This gives a running time in $\mathcal{O}(\tau(1,5)^n) \subseteq \mathcal{O}(1.3248^n)$: When a is true one variable, *viz.* a, is removed; when a is false all neighbours must be true.

This analysis, while correct, is quite unfair, because our choice of branching variable guarantees that in every worst case situation, the quotient $l(F)/n(F)$ will decrease. This means that we come closer and closer to C_3 being applicable. We will try to remedy the unfairness by using a better measure of formula complexity. The time bound we want to prove is $\mathcal{O}(1.2561^n)$. For $\delta(F) \geq 6$, when C is applicable, this is accomplished already by a trivial analysis (because $\tau(1,7) < 1.2555$) and so, we need the better formula measurement only for the analysis of C_5.

Thus, for the analysis of C_5, we use a piecewise linear function $f(n,l)$ as a measure of complexity, with a behaviour determined by the quotient $l(F)/n(F)$. When $l > kn$ we can always find a variable with degree at least $k+1$. Since C_5 picks a variable of maximum degree we get a sequence of worst cases, depending on the average degree of F. If we define $f(n,l)$ such that all these worst cases have the same branching number α, the bound $\mathcal{O}\left(\alpha^{f_{\max}(n)}\right)$ gets closer to the actual worst-case running time.

In our analysis, we will find a sequence of worst cases as we consider higher l/n quotients, and with each worst case we associate a linear function $f_i(n,l) = a_i n + b_i l$, a lower limit k_i for the l/n quotient below which worse cases appear, and an upper limit k_{i+1} for the l/n quotient above which the case does not appear. Each function $f_i(n,l)$ has its parameters chosen so that the worst-case branching number in the range $k_i < l/n \leq k_{i+1}$ will be equal to $\tau(4,4)$ for every i. The range k_i–k_{i+1} is referred to as *section* i, for each i, and $f(n,l)$ is partitioned into functions $f_i(n,l)$ in the same way as the l/n quotient axis is partitioned into sections.

The bottom-most non-zero function $f_1(n,l)$ corresponds to the algorithm C_3, and is applicable for l/n values from $k_1 = 2$ to $k_2 = 3$, as this is the range of l/n where C_3 is the worst case, as we shall see. The rest of the functions $f_i(n,l)$ corresponds to worst cases for

i	k_i	χ_i	Running time
0	0	0	$\mathcal{O}(1)$
1	2	0	$\mathcal{O}(poly(n))$
2	3	1	$\mathcal{O}(1.1892^n)$
3	3.5	1.1340	$\mathcal{O}(1.2172^n)$
4	3.75	1.1914	$\mathcal{O}(1.2294^n)$
5	4	1.2410	$\mathcal{O}(1.2400^n)$
6	4+4/29	1.2536	$\mathcal{O}(1.2427^n)$
7	4+4/9	1.2788	$\mathcal{O}(1.2481^n)$
8	4+4/7	1.2881	$\mathcal{O}(1.2501^n)$
9	4.8	1.3033	$\mathcal{O}(1.2534^n)$
10	5	1.3154	$\mathcal{O}(1.2561^n)$

Table 6.1: *k_i, χ_i and running times*

C_5. Our measure $f(n,l)$ is undefined for $l > 5n$, because as already indicated, we do not need it for those cases.

To simplify the presentation of the time complexity proof, we give the definitions of $f(n,l)$ and related terms here, then proceed to state some of its properties. Definitions:

$$f(n,l) = f_i(n,l) \text{ if } k_i < l/n \leq k_{i+1},\ 0 \leq i \leq 9 \quad (6.1)$$

$$f_i(n,l) = \chi_i n + (l - k_i n) b_i,\ 0 \leq i \leq 9 \quad (6.2)$$

$$\chi_0 = 0 \quad (6.3)$$

$$\chi_i = \chi_{i-1} + (k_i - k_{i-1}) b_{i-1},\ 1 \leq i \leq 10 \quad (6.4)$$

$$a_i = \chi_i - k_i b_i \quad (6.5)$$

Note that we now have two equivalent ways of expressing $f_i(n,l)$: $\chi_i n + (l - k_i n) b_i$ and $a_i n + b_i l$. The exact values of k_i can be found in Table 6.1, along with rounded-off values for χ_i and $\tau(4,4)^{\chi_i}$, the latter being c in the $\mathcal{O}(c^n)$ upper limit on the running time for a formula F with $l(F)/n(F) \leq k_i$. Hence, our $f_{max}(n) = \max f(n,l)$, $k_i < l/n \leq k_{i+1}$ for all $l \in \mathbb{N}$ is $\chi_i n$.

i	b_i, definitions	b_i	a_i
0	0	0	0
1	1	1	-2
2	$\tau(1+5b_2, 5+5b_2) = \tau(4,4)$	0.2680	0.1961
3	$\tau(\chi_3+4.5b_3, 5\chi_3+4.5b_3) = \tau(4,4)$	0.2295	0.3308
4	$\tau(\chi_4+4.25b_4, 5\chi_4+5.25b_4) = \tau(4,4)$	0.1987	0.4461
5	$\tau(\chi_5+6b_5, 6\chi_5+2b_5) = \tau(4,4)$	0.0914	0.8755
6	$\tau(\chi_6+(5+25/29)b_6, 6\chi_6+(3+5/29)b_6) = \tau(4,4)$	0.0821	0.9139
7	$\tau(\chi_7+(5+5/9)b_7, 6\chi_7+(3+1/3)b_7) = \tau(4,4)$	0.0736	0.9517
8	$\tau(\chi_8+(5+3/7)b_8, 6\chi_8+(4+4/7)b_8) = \tau(4,4)$	0.0665	0.9841
9	$\tau(\chi_9+5.2b_9, 6\chi_9+5.2b_9) = \tau(4,4)$	0.0602	1.0143

Table 6.2: *b_i and a_i parameters*

The expressions defining the values of b_i can be found in Table 6.2, along with rounded-off numerical values for b_i and a_i. These expressions come from the branching numbers for the worst case in each section i. We see that $f(n,l)$ satisfies the requirements for our method of analysis:

1. The function $f(n,l)$ is continuous, as $f_{i-1}(n, k_i n) = \chi_{i-1}n + (k_i n - k_{i-1}n)b_{i-1}) = (\chi_{i-1} + (k_i - k_{i-1})b_{i-1})n = \chi_i n = f_i(n, k_i n)$ for all i.

2. For a formula without singletons, $f(n,l)$ is obviously never negative.

The properties of $f(n,l)$ that will be used in the analysis are next presented as four lemmas. The first two follow easily from Table 6.2.

Lemma 27. $f(n,l) > f(n-1,l)$ if $l > 3n$.

Lemma 28. $f(n,l) > f(n,l-1)$ if $l > 2n$.

Intuitively, the following lemma says that when doing a branching from F to F_1 such that F_1 belongs in another section than F, then this will not be a worst case.

Lemma 29. $f(n,l) - f(n_1,l_1) \geq f_i(n,l) - f_i(n_1,l_1)$ when $k_i n \leq l \leq k_{i+1}n$.

Proof. We first prove that the lemma holds when $l_1/n_1 < l/n$. The equality certainly holds if $k_i n_1 \leq l_1 \leq k_{i+1} n_1$ so assume this is not the case. We will focus on the transition of a single barrier k_i, i.e. that l/n belongs in section i while l_1/n_1 belongs in section $i-1$. If the property holds for all such barriers, then it holds globally.

Assume that $k_i n \leq l \leq k_{i+1} n$, $k_{i-1} n_1 \leq l_1 \leq k_i n_1$, $n_1 < n$ and $l_1 < l$. We want to check that $f_i(n_1,l_1) \geq f_{i-1}(n_1,l_1)$, i.e.

$$f_i(n_1,l_1) - f_{i-1}(n_1,l_1) = (a_i - a_{i-1})n_1 + (b_i - b_{i-1})l_1 \geq 0$$

We have that

$$a_i - a_{i-1} = (\chi_i - k_i b_i) - (\chi_{i-1} - k_{i-1} b_{i-1}) = (\chi_{i-1} + (k_i - k_{i-1})b_{i-1} - k_i b_i) - (\chi_{i-1} - k_{i-1} b_{i-1}) = k_i(b_{i-1} - b_i)$$

Inserting this into the previous inequality, we get $(b_{i-1} - b_i)(k_i n_1 - l_1) \geq 0$, and as $k_i n_1 \geq l_1$ by assumption, we see that the first part of the lemma follows from the observation that b_i is decreasing with increasing i.

As for the case when $l_1/n_1 > l/n$, we reason similarly: We want to check that $f_i(n_1,l_1) \geq f_{i-1}(n_1,l_1)$, i.e.

$$(a_i - a_{i+1})n_1 + (b_i - b_{i+1})l_1 \geq 0$$

We have that $a_i - a_{i+1} = k_{i+1}(b_{i-1} - b_i)$ and inserting this into the inequality, we get $(b_{i+1} - b_i)(k_{i+1} n_1 - l_1) \geq 0$. We see that both factors are less than or equal to 0 and we have proved the lemma. ☐

The following lemma can be used to justify that solving components separately will not increase the running time when measuring in $f(n,l)$.

Lemma 30. $f(n,l) \geq f(n_1,l_1) + f(n-n_1, l-l_1)$ if $0 \leq n_1 \leq n$ and $0 \leq l_1 \leq l$.

Proof. Let a and c be constants so that $n_1 = an$ and $l_1 = al + c$. Furthermore, let $k_i \leq k = l/n \leq k_{i+1}$ and $\chi = f(n,l)/n$. Now

consider the constant c. If $c \geq 0$, then $l_1/n_1 \geq l/n$; we will here w.l.o.g. assume that this is the case. We now derive some intermediate results to show the lemma. Let α be any number.

$$\begin{array}{rcll} f(\alpha n, k\alpha n) & = & \chi_i \alpha n + (k\alpha n - k_i \alpha n) b_i & = \\ \alpha(\chi_i n + (kn - k_i n) b_i) & = & \alpha(\chi_i n + (l - k_i n) b_i) & = \\ \alpha f(n, l) & = & \chi \alpha n & \end{array}$$

The next intermediate step:

$$f(an, kan + c) = \chi an + cb_{up} \text{ for some } b_{up}$$

To explain the part $b_{up}c$: Assume that l/n belongs in section i and that l_1/n_1 belongs in section $i+j$. Then $b_{up}c$ is shorthand for

$$\begin{array}{l}
f_{i+j}(an, kan + c) - f_i(an, kan) \\
= \\
(f_{i+j}(an, kan + c) - f_{i+j}(an, k_{i+j}an))+ \\
(f_{i+j}(an, k_{i+j}an) - f_{i+j-1}(an, k_{i+j-1}an))+ \\
\ldots + \\
(f_{i+2}(an, k_{i+2}an) - f_{i+1}(an, k_{i+1}an))+ \\
(f_{i+1}(an, k_{i+1}an) - f_i(an, kan)) \\
= \\
(kan + c - k_{i+j}an)b_{i+j}+ \\
(\chi_{i+j-1} + (k_{i+j} - k_{i+j-1}))b_{i+j-1}an - \chi_{i+j-1}an \\
\ldots + \\
(\chi_{i+1} + (k_{i+2} - k_{i+1})b_{i+1})an - \chi_{i+1}an+ \\
(\chi_i + (k_{i+1} - k_i)b_i)an - (\chi_i + (k - k_i)b_i)an \\
= \\
(kan + c - k_{i+j}an)b_{i+j}+ \\
(k_{i+j}an - k_{i+j-1}an)b_{i+j-1}+ \\
\ldots + \\
(k_{i+2}an - k_{i+1}an)b_{i+1}+ \\
(k_{i+1}an - k_i an)b_i - (kan - k_i an)b_i
\end{array}$$

Note that cb_{up} can be expressed as $c_1 b_i + c_2 b_{i+1} + \ldots + c_j b_{i+j}$ with $c_1 + c_2 + \ldots + c_j = c$ so that $b_{up}c \leq b_i c$ (because b_i is decreasing with higher i.) The next intermediate step:

$$f((1-a)n, k(1-a)n-c) = \chi(1-a)n - cb_{down} \text{ for some } b_{down}$$

As for the part cb_{down}, it is a summation similar to cb_{up}. Doing the same formula manipulation as for cb_{up}, one sees that $cb_{down} \geq b_i c$. Thus,

$$\begin{aligned} f(n_1, l_1) + f(n - n_1, l - l_1) &= \\ f(an, kan + c) + f((1-a)n, k(1-a)n - c) &= \\ (\chi an + cb_{up}) + (\chi(1-a)n - cb_{down}) &= \\ \chi n + (b_{up} - b_{down})c &\leq \\ \chi n &= f(n, l) \end{aligned}$$

□

We need a lemma that allows us to make a connection between the value of $l(F)/n(F)$ and worst-case branchings. The following lemma gives a means to deduce a minimum value for $S(a)$ for a given value of $l(F)/n(F)$. We will soon give an example to illustrate its use.

Lemma 31. Let F be a non-empty formula such that $l(F)/n(F) = k$, and define $\alpha(a)$ and $\beta(a)$ such that

$$\begin{aligned} \alpha(a) &= \delta(a) + |\{b \in N(a) \mid \delta(b) < k\}| \\ \beta(a) &= 1 + \sum_{\{b \in N(a) \mid \delta(b) < k\}} 1/\delta(b) \end{aligned}$$

Then there exists some variable $a \in Var(F)$ with $\delta(a) \geq k$ such that $\alpha(a)/\beta(a) \geq k$.

For an example of the use of the lemma (before the proof): Say that the formula F is in Section 2, i.e. $l/n \in [3, 3.5]$ and that $\delta(F) = 4$. We then know that there is a variable a such that $\delta(a) = 4$ and such that $\alpha(a)/\beta(a) \geq 3$. If a has one neighbour b of degree 2 and three other neighbours, we have that $\alpha(a)/\beta(a) = (4+1)/(1+1/2) =$

$10/3 > 3.33$. However, it is not possible that two neighbours have degree 2, because then $\alpha(a)/\beta(a) = (4+2)/(1+1) = 3$. This means that the maximum $S(a)$ must be at least 15 (for the case that the neighbours have degrees 2, 3, 3 and 3.) Note that this only tells us that $S(a) \geq 15$; we have no guarantees that $\alpha(a)/\beta(a) > 3$ for the chosen variable a. We now proceed to the proof of the lemma:

Proof. Consider the following sums:

$$A = \sum_{\{a \in Var(F) \mid \delta(a) \geq k\}} \alpha(a)$$
$$B = \sum_{\{a \in Var(F) \mid \delta(a) \geq k\}} \beta(a)$$

We may view every variable a with $\delta(a) \geq k$ as contributing exactly $\delta(a)$ to A and 1 to B, and each variable b with $\delta(b) < k$, i to A and $i/\delta(b)$ to B, for some integer $i \leq \delta(b)$ (i is not guaranteed to equal $\delta(b)$ because b may occur with other variables of degree less than k.) We find that there are numbers $n'_i(F)$ with $n'_i(F) \leq n_i$ for $i < k$ and $n'_i(F) = n_i(F)$ for $i \geq k$ such that the following holds:

$$A = \sum_i i n'_i(F) = l(F) - \sum_{i<k} i(n_i(F) - n'_i(F))$$
$$B = \sum_i n'_i(F) = n(F) - \sum_{i<k} (n_i(F) - n'_i(F))$$

Here, we used $\sum_i i \cdot n_i(F) = l(F)$ and $\sum_i n_i(F) = n(F)$. As $l(F) = k \cdot n(F)$, we have:

$$A \geq k \cdot B \tag{6.6}$$

The set $\{a \in Var(F) \mid \delta(a) \geq k\}$ is clearly not empty. Hence, if we had $\alpha(a) < k\beta(a)$ for all a with $\delta(a) \geq k$, inequality (6.6) could not hold. Therefore there is an a with $\delta(a) \geq k$ such that $\alpha(a) \geq k\beta(a)$. □

Now, we can proceed by proving an upper bound for the running time of C_5. The proof will be divided into lemmas according to the

value of $l(F)/n(F)$. We will need to prove that the worst-case branching number in each section is $\tau(4,4)$, i.e. that we have chosen the right values for the parameters of $f(n,l)$.

As a final thing before the proof, a note on case 5a of the algorithm. If case 5a of the algorithm is applied, we remove b by assignment and $N(a)$ by multiplier reduction. In total, we reduce n by at least $\delta(a)+2$ and l by at least $S(a)+4$ in both branches. We will see that when using case 5b, the largest worst-case reduction is $\Delta n = \delta(a)+1$ and $\Delta l = S(a)+4$. When $l > 3n$, by Lemma 27, this is clearly a harder case than case 5a.

Lemma 32. *$C_5(F,c,w)$ runs in time $\mathcal{O}\left(\tau(4,4)^{f(F)}\right)$ for a maximally reduced formula F with $l \leq 3n$.*

Proof. In this lemma, we will use $f_1(n,l)$ as a measure, with $b_1 = 1$ and $a_1 = -2$. We inspect the cases:

1. Polynomial time solvable by assumption.
2. By Lemma 30 this will not be a worst case.
3. Recall that in the analysis of C_3 we measured the number of variables of degree 3 as $l-2n$ and this is the same as $b_1 = 1$ and $a_1 = -2$.
4. By Lemma 30 this will not be a worst case.
5. For a branching from a maximally reduced F to a maximally reduced F_1 or F_2, f is reduced by at least 6: We know that $\delta(a)$ is at least 4. In any branch, at least a is removed, along with r other variables, $0 \leq r \leq \delta(a)$. This means that $\Delta l = 2\delta(a)+2r$ and $\Delta n = 1+r$. With the current parameters of f, we get $\Delta f = 2(\delta(a)+r) - 2(1+r)$.

□

Lemma 33. *$C_5(F,c,w)$ runs in time $\mathcal{O}\left(\tau(4,4)^{f(F)}\right)$ for a maximally reduced formula F with $l \leq 4n$.*

$$\begin{array}{ll} (a \vee b) & (b \vee d) \\ (a \vee c) & (c \vee e) \\ (a \vee d) & (d \vee f) \\ (a \vee e) & (e \vee b) \\ (a \vee f) & (f \vee g) \\ & (c \vee h) \end{array}$$

Figure 6.5: *For this part of a formula, assigning a true results in* $\Delta n_1 = 1$ *and* $\Delta l_1 = 10$*; assigning a false results in* $\Delta n_2 = 6$ *and* $\Delta l_2 = 22$

Proof. This lemma uses $f_2(n, l)$ through $f_4(n, l)$ as measures, with parameters as previously defined.

The main algorithm C takes care of variables of degree 6 and higher and non-constant variables of degree 5. This means that the first case to consider is $\delta(F) = 5$ such that all variables of degree 5 are constant. Note that the first priority of the algorithm when choosing a branching variable is that it have maximum degree. This means that we have no guarantees concerning $S(a)$. For instance, a might be the only variable of degree 5 and the variable guaranteed by Lemma 31 has degree 4. To find the possible worst cases to examine one can argue like this: Trying to minimise Δl by finding scenarios with many neighbours of degree 2 we increase Δn because those neighbours will get removed either by multiplier reduction or the assignment of a. Let us therefore examine all possible cases of the number of neighbours of degree 2. The reader should keep in mind that we are examining case 5b. This means that there must be at least three neighbours in contact with external variables. For instance, we can disregard scenarios such as the one depicted in Figure 6.5.

1. 5 neighbours of degree 2: This scenario is depicted in Figure 6.6. We have that $\Delta_1 n = \Delta_2 n = 6$, $\Delta_1 l = \Delta_2 l = 20$ and

$$\tau(f_4(6, 20), f_4(6, 20)) < \tau(6.6506, 6.6506) < \tau(4, 4)$$
$$\tau(f_3(6, 20), f_3(6, 20)) < \tau(6.5748, 6.5748) < \tau(4, 4)$$
$$\tau(f_2(6, 20), f_2(6, 20)) < \tau(6.5366, 6.5366) < \tau(4, 4)$$

$$
\begin{array}{ll}
(a \vee b) & (b \vee g) \\
(a \vee c) & (c \vee h) \\
(a \vee d) & (d \vee i) \\
(a \vee e) & (e \vee j) \\
(a \vee f) & (f \vee k)
\end{array}
$$

Figure 6.6: *For this part of a formula, assigning a either value results in* $\Delta n = 6$ *and* $\Delta l = 20$

$$
\begin{array}{ll}
(a \vee b) & (b \vee g) \\
(a \vee c) & (b \vee h) \\
(a \vee d) & (c \vee d) \\
(a \vee e) & (e \vee i) \\
(a \vee f) & (f \vee j)
\end{array}
$$

Figure 6.7: *For this part of a formula, assigning a true results in* $\Delta n_1 = 5$ *and* $\Delta l_1 = 16$*; assigning a false results in* $\Delta n_2 = 6$ *and* $\Delta l_2 = 20$

2. 4 neighbours of degree 2: This scenario is depicted in Figure 6.7. We have that $\Delta_1 n = 5, \Delta_1 l = 16$, $\Delta_2 n = 6, \Delta_2 l = 20$ and

$$
\begin{aligned}
\tau(f_4(5,16), f_4(6,20)) &< \tau(5.4097, 6.6506) < \tau(4,4) \\
\tau(f_3(5,16), f_3(6,20)) &< \tau(5.326, 6.5748) < \tau(4,4) \\
\tau(f_2(5,16), f_2(6,20)) &< \tau(5.2685, 6.5366) < \tau(4,4)
\end{aligned}
$$

3. 3 neighbours of degree 2: This scenario is depicted in Figure 6.8. We have that $\Delta_1 n = 4, \Delta_1 l = 14$, $\Delta_2 n = 6, \Delta_2 l = 20$ and

$$
\begin{aligned}
\tau(f_4(4,14), f_4(6,20)) &< \tau(4.5662, 6.6506) < \tau(4,4) \\
\tau(f_3(4,14), f_3(6,20)) &< \tau(4.5362, 6.5748) < \tau(4,4) \\
\tau(f_2(4,14), f_2(6,20)) &< \tau(4.5364, 6.5366) < \tau(4,4)
\end{aligned}
$$

4. 2 neighbours of degree 2: This scenario is depicted in Figure 6.9. We have that $\Delta_1 n = 3, \Delta_1 l = 12$, $\Delta_2, n = 6\Delta_2 l = 22$. The following branching numbers are not dominated by $\tau(4,4)$,

$$\begin{array}{ll} (a \vee b) & (b \vee c) \\ (a \vee c) & (d \vee g) \\ (a \vee d) & (e \vee f) \\ (a \vee e) & (e \vee h) \\ (a \vee f) & (f \vee i) \end{array}$$

Figure 6.8: *For this part of a formula, assigning a true results in* $\Delta n_1 = 4$ *and* $\Delta l_1 = 14$*; assigning a false results in* $\Delta n_2 = 6$ *and* $\Delta l_2 = 20$

$$\begin{array}{ll} (a \vee b) & (b \vee c) \\ (a \vee c) & (d \vee g) \\ (a \vee d) & (e \vee h) \\ (a \vee e) & (f \vee i) \\ (a \vee f) & (d \vee e) \\ & (f \vee j) \end{array}$$

Figure 6.9: *For this part of a formula, assigning a true results in* $\Delta n_1 = 3$ *and* $\Delta l_1 = 12$*; assigning a false results in* $\Delta n_2 = 6$ *and* $\Delta l_2 = 22$

but we can easily show that they are smaller since $1.1892 < \tau(4,4) < 1.1893$.

$$\begin{array}{l} \tau(f_4(3,12), f_4(6,22)) < \tau(3.7227, 7.048) < 1.1426 < \tau(4,4) \\ \tau(f_3(3,12), f_3(6,22)) < \tau(3.7464, 7.0338) < 1.1425 < \tau(4,4) \\ \tau(f_2(3,12), f_2(6,22)) < \tau(3.8043, 7.0726) < 1.1408 < \tau(4,4) \end{array}$$

5. 1 neighbour of degree 2: This scenario is depicted in Figure 6.10. We have that $\Delta_1 n = 2, \Delta_1 l = 12$, $\Delta_2 n = 6, \Delta_2 l = 22$ and

$$\begin{array}{l} \tau(f_4(2,12), f_4(6,22)) < \tau(3.2766, 7.048) < 1.1515 < \tau(4,4) \\ \tau(f_3(2,12), f_3(6,22)) < \tau(3.4156, 7.0338) < 1.1487 < \tau(4,4) \\ \tau(f_2(2,12), f_2(6,22)) < \tau(3.6082, 7.0726) < 1.1444 < \tau(4,4) \end{array}$$

6. 0 neighbours of degree 2: This scenario is depicted in Figure 6.11. We have that $\Delta_1 n = 5, \Delta_1 l = 10$, $\Delta_2 n = 6, \Delta_2 l = 24$ and

$$\begin{array}{ll} (a \vee b) & (b \vee i) \\ (a \vee c) & (c \vee d) \\ (a \vee d) & (d \vee e) \\ (a \vee e) & (e \vee g) \\ (a \vee f) & (f \vee h) \\ & (f \vee c) \end{array}$$

Figure 6.10: *For this part of a formula, assigning a true results in $\Delta n_1 = 2$ and $\Delta l_1 = 12$; assigning a false results in $\Delta n_2 = 6$ and $\Delta l_2 = 22$*

$$\begin{array}{ll} (a \vee b) & (b \vee d) \\ (a \vee c) & (c \vee e) \\ (a \vee d) & (d \vee f) \\ (a \vee e) & (e \vee g) \\ (a \vee f) & (f \vee h) \\ & (b \vee i) \\ & (c \vee j) \end{array}$$

Figure 6.11: *For this part of a formula, assigning a true results in $\Delta n_1 = 1$ and $\Delta l_1 = 10$; assigning a false results in $\Delta n_2 = 6$ and $\Delta l_2 = 24$*

$$\begin{array}{l} \tau(f_4(1,10), f_4(6,24)) < \tau(2.4331, 7.4454) < 1.1681 < \tau(4,4) \\ \tau(f_3(1,10), f_3(6,24)) < \tau(2.6258, 7.4928) < 1.1617 < \tau(4,4) \\ \tau(f_2(1,10), f_2(6,24)) < \tau(2.8761, 7.6086) < 1.1537 < \tau(4,4) \end{array}$$

Next, the actual worst cases, beginning with section 2. In order to find the worst cases, we reason similarly as above. However, we now have guarantees on the values of $S(a)$. Given those minimum values of $S(a)$, we try to minimise the number of clauses where neighbours participate and yet make sure that there are three external variables in these clauses. Due to the balanced branching effect, the worst cases are maximally unbalanced.

Section 2: $\boldsymbol{l/n \in [3, 3.5]}$, $\boldsymbol{\delta(F) = 4}$ Lemma 31 guarantees that there will be some variable a with $\alpha(a)/\beta(a) > 3$, and the minimum

$$\begin{array}{ll} (a \vee b) & (b \vee f) \\ (a \vee c) & (c \vee d) \\ (a \vee d) & (c \vee g) \\ (a \vee e) & (d \vee e) \\ & (e \vee h) \end{array}$$

Figure 6.12: *For this part of a formula, assigning a true results in $\Delta n_1 = 2$ and $\Delta l_1 = 10$; assigning a false results in $\Delta n_2 = 5$ and $\Delta l_2 = 18$*

$$\begin{array}{ll|ll} (a \vee b) & (b \vee f) & (a \vee b) & (b \vee f) \\ (a \vee c) & (c \vee d) & (a \vee c) & (c \vee d) \\ (a \vee d) & (c \vee g) & (a \vee d) & (c \vee b) \\ (a \vee e) & (d \vee e) & (a \vee e) & (d \vee e) \\ & (e \vee h) & & (e \vee g) \\ & (b \vee i) & & (b \vee h) \end{array}$$

Figure 6.13: *For these two configurations, assigning a true results in $\Delta n_1 = 1$ and $\Delta l_1 = 8$; assigning a false results in $\Delta n_2 = 5$ and $\Delta l_2 = 20$*

$S(a)$ for variables with this property is 15, occurring if the degrees of the neighbours are 2, 3, 3 and 3.

There are two candidates for worst case recursion to examine in this case.

1. With $S(a) = 15$, at least one neighbour has degree 2. The worst variant of this case is shown in Figure 6.12, with a branching number of $\tau(2a_2 + 10b_2, 5a_2 + 18b_2) = \tau(2(\chi_2 - k_2b_2) + 10b_2, 5(\chi_2 - k_2b_2) + 18b_2) = \tau(2(1 - 3b_2) + 10b_2, 5(1 - 3b_2) + 18b_2) = \tau(2 + 4b_2, 5 + 3b_2) < \tau(4, 4)$.

2. The worst possible case without neighbours of degree 2 is when every neighbour has degree 3, with $S(a) = 16$, as shown in Figure 6.13. We have $\tau(a_2 + 8b_2, 5a_2 + 20b_2) = \tau(1 + 5b_2, 5 + 5b_2) = \tau(4, 4)$ by definition of b_2. This case also occurs with $S(a) = 17$, with neighbours of degrees 3, 3, 3, and 4.

We find that the second case is harder than the first, and that both

are at most $\tau(4,4)$. As given in the tables, we have $b_2 \approx 0.2680$, $a_2 \approx 0.1961$, and $\chi_3 = 1 + 0.5b_2 \approx 1.1340$.

Cases with $S(a) = 17$ can occur when $l(F)/n(F) \leq 3.5$, so the next section begins when $l(F)/n(F) > 3.5$.

Section 3: $\boldsymbol{l/n \in [3.5, 3.75]}$, $\boldsymbol{\delta(F) = 4}$ The minimum value for $S(a)$ for variables with $\alpha(a)/\beta(a) > 3.5$ is 18, and $S(a) = 18$ and $S(a) = 19$ both provide worst-case branchings with branching number $\tau(a_3 + 8b_3, 5a_3 + 22b_3) = \tau(\chi_3 + 4.5b_3, 5\chi_3 + 4.5b_3) = \tau(4,4)$, by definition of b_3. We have $b_3 \approx 0.2295$, $a_3 \approx 0.3308$ and $\chi_4 = \chi_3 + 0.25b_3 \approx 1.1914$.

Cases with $S(a) = 19$ can occur up to $l(F)/n(F) = 3.75$, so the next section begins at 3.75.

Section 4: $\boldsymbol{l/n \in [3.75, 4]}$, $\boldsymbol{\delta(F) = 4}$ The only possible value for $S(a)$ is 20, resulting in a worst case branching number of $\tau(a_4 + 8b_4, 5a_4 + 24b_4) = \tau(\chi_4 + 4.25b_4, 5\chi_4 + 5.25b_4) = \tau(4,4)$ by definition of b_4. We have $b_4 \approx 0.1987$, $a_4 \approx 0.4461$ and $\chi_5 = \chi_4 + 0.25b_4 \approx 1.2410$.

As we see, the worst-case branching number is $\tau(4,4)$ for sections 2, 3 and 4. □

Lemma 34. $C_5(F, c, w)$ runs in time $\mathcal{O}\left(\tau(4,4)^{f(F)}\right)$ for a maximally reduced formula F with $l \leq 5n$.

Proof. This lemma uses $f_5(n,l)$ through $f_9(n,l)$ as measures, with parameters as previously defined. We know that $\delta(F) = 5$, so we proceed immediately with section 5.

Section 5: $\boldsymbol{l/n \in [4, 4 + 4/29]}$ The minimum value for $S(a)$ for variables with $\alpha(a)/\beta(a) > 4$ is 23, with a worst-case branching with branching number $\tau(a_5 + 10b_5, 6a_5 + 26b_5) = \tau(\chi_5 + 6b_5, 6\chi_5 + 2b_5) = \tau(4,4)$ by definition of b_5. We have $b_5 \approx 0.0914$, $a_5 \approx 0.8755$ and $\chi_6 = \chi_5 + (4/29)b_5 \approx 1.2536$.

$S(a) = 23$ can occur up to $l(F)/n(F) = 4 + 4/29 \approx 4.1379$, so the next section begins at that point.

Section 6: $\boldsymbol{l/n \in [4 + 4/29, 4 + 4/9]}$ The minimum value for $S(a)$ for variables with $\alpha(a)/\beta(a) > 4 + 4/29$ is 24, and $S(a) = 24$ and $S(a) = 25$ both have worst-case branchings with a branching number of $\tau(a_6 + 10b_6, 6a_6 + 28b_6) = \tau(\chi_6 + (5 + 25/29)b_6, 6\chi_6 + (3 + 5/29)b_6) =$

$\tau(4,4)$ by definition of b_6. We have $b_6 \approx 0.0821$, $a_6 \approx 0.9139$ and $\chi_7 = \chi_6 + (4/9 - 4/29)b_6 \approx 1.2788$.

$S(a) = 25$ can occur up to $l(F)/n(F) = 4 + 4/9 \approx 4.4444$, so the next section begins at that point.

Section 7: $l/n \in [4 + 4/9, 4 + 4/7]$ The minimum value for $S(a)$ for variables with $\alpha(a)/\beta(a) > 4 + 4/9$ is 26, and $S(a) = 26$ and $S(a) = 27$ both have worst-case branchings with a branching number of $\tau(a_7 + 10b_7, 6a_7 + 30b_7) = \tau(\chi_7 + (5 + 5/9)b_7, 6\chi_7 + (3 + 1/3)b_7) = \tau(4,4)$ by definition of b_7. We have $b_7 \approx 0.0736$, $a_7 \approx 0.9517$ and $\chi_8 = \chi_7 + (4/7 - 4/9)b_7 \approx 1.2881$.

$S(a) = 27$ can occur up to $l(F)/n(F) = 4 + 4/7 \approx 4.5714$, so the next section begins at that point.

Section 8: $l/n \in [4 + 4/7, 4.8]$ The minimum value for $S(a)$ for variables with $\alpha(a)/\beta(a) > 4 + 4/7$ is 28, and $S(a) = 28$ and $S(a) = 29$ both have worst-case branchings with a branching number of $\tau(a_8 + 10b_8, 6a_8 + 32b_8) = \tau(\chi_8 + (5 + 3/7)b_8, 6\chi_8 + (4 + 4/7)b_8) = \tau(4,4)$ by definition of b_8. We have $b_8 \approx 0.0665$, $a_8 \approx 0.9841$ and $\chi_9 = \chi_8 + (0.8 - 4/7)b_8 \approx 1.3033$.

$S(a) = 29$ can occur up to $l(F)/n(F) = 4.8$, so the last section begins at that point.

Section 9: $l/n \in [4.8, 5]$ The only possible value for $S(a)$ is 30, with a worst case branching number of $\tau(a_9 + 10b_9, 6a_9 + 34b_9) = \tau(\chi_9 + 5.2b_9, 6\chi_9 + 5.2b_9) = \tau(4,4)$ by definition of b_9. We have $b_9 \approx 0.0602$, $a_9 \approx 1.0143$ and $\chi_{10} = \chi_9 + 0.2b_9 \approx 1.3154$.

We have shown that C_5 runs in $\mathcal{O}\left(\tau(4,4)^{f(F)}\right)$ time. □

Corollary 35. *C_5 runs in $\mathcal{O}\left(\tau(4,4)^{\chi_{10} n}\right) \subseteq \mathcal{O}\left(1.2561^n\right)$ time.*

6.3.3 The main function C

Now, we can finally give the last part of our algorithm: The function C, applicable to a general formula F is shown in Figure 6.14.

The following theorem will establish an upper time bound for C:

Theorem 36. *$C(F, c, w)$ runs in $\mathcal{O}\left(1.2561^n\right)$ time.*

Algorithm $C(F, c, w)$

1. If $n(F) < 10$, then return $C_E(F)$
2. If F is not connected, return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = C(F_i, c, w)$ for the connected components $F_0, \ldots, F_j$.
3. If there exists a non-constant variable a with $\delta(a) \geq 5$, branch on a.
4. If $\delta(F) < 6$, return $C_5(F, c, w)$.
5. Pick a variable a of maximum degree and branch on it.

Figure 6.14: *The algorithm C for $\#2\text{SAT}_w$*

Proof. Cases 1 and 2 are reductions, or take only polynomial time. For case 3, the worst running time is $T(n) = T(n-2) + T(n-5)$ with solutions in $\mathcal{O}(\tau(2,5)^n) \subseteq \mathcal{O}(1.2366^n)$. Case 4 takes $\mathcal{O}(1.2561^n)$ time by Corollary 35. For case 5, the worst case is $T(n) = T(n-1) + T(n-7)$ with solutions in $\mathcal{O}(\tau(1,7)^n) \subseteq \mathcal{O}(1.2555^n)$. All these cases are contained in $\mathcal{O}(1.2561^n)$. $\square$

6.4 Algorithm for $\#3\text{SAT}_w$

In this section we present the algorithm D for $\#3\text{SAT}_w$ and provide an upper bound on its running time. The complexity analysis is somewhat delicate and requires numerical calculations to obtain a solution to an optimisation problem.

There is a auxiliary function $D_E(F, c, w)$ that will use exhaustive search to calculate $\#3\text{SAT}_w(F, c, w)$. We will only apply it to instances of constant size, and thus we consider its running time to be in $\mathcal{O}(1)$.

The main algorithm is given in Figure 6.15. When starting, assume that F is maximally reduced.

Theorem 37. $D(F, c, w) = \#3\text{SAT}_w(F, c, w)$

Algorithm $D(F, c, w)$

1. If $n(F) < 10$, then return $D_E(F, c, w)$.

2. If F is not connected, then return (c, w) where $c = \prod_{i=0}^{j} c_i$, $w = \sum_{i=0}^{j} w_i$ and $(c_i, w_i) = D(F_i, c, w)$ for the connected components $F_0, \ldots, F_j$.

3. If multiplier reduction applies, then apply it, removing the part with lowest $n(F)$ value.

4. If there exists a variable v such that $\delta(v) = \delta_3(v) = 1$, then let a be a neighbour of maximum degree and recursively branch on a.

5. If there exists a variable v such that $\delta(v) = 2$ and $\delta_2(v) > 0$, then let a be a neighbour that shares a 3-clause with v, if possible, or else a neighbour of maximum $\delta_2(a)$, and recursively branch on a.

6. If there exists at least one 2-clause in F, then let v be a variable with maximum $\delta(v)$ among all variables with maximum $\delta_2(v)$, and recursively branch on v.

7. If there exists a variable v such that $\delta(v) = \delta_3(v) = 2$, then, assuming that one 3-clause containing v is $(v \vee a \vee b)$, recursively branch on $b = true$, $b = false \wedge a = true$ and $b = false \wedge a = false \wedge v = true$.

8. Pick a variable v of maximum degree and recursively branch on it.

Figure 6.15: *The algorithm D for #3SAT$_w$*

Proof. For cases 1 – 6 and 8 the correctness can be proved using the same arguments as for Lemma 24. The correctness of case 7 follows from Lemma 16 and the observation that $F(b/false; a/false; v/true)$, $F(b/true)$ and $F(b/false; a/true)$ cover F. □

In the time complexity analysis, we will measure the formula complexity using the function $f(F) = n - \Psi(k)$, where k is the number of 2-clauses in F. $\Psi(k)$ is a real-valued function with the following two properties:

1. $0 = \Psi(0) < \Psi(1) < \ldots < \Psi(4) < 1$, $\Psi(k) = \Psi(4)$ for all $k > 4$.

2. $\Psi(k+1) - \Psi(k) \geq \Psi(k+2) - \Psi(k+1)$ for all k.

In other words, we use a more fine-grained measure than just $n(F)$, with four subdivisions between n and $n+1$. $\Psi(1)$ through $\Psi(4)$ will be given exact values, using numerical optimisation. Note that since $\Psi(k) < 1$, we have $n-1 < f(F) \leq n$ and $f(F) \leq 0$ only if $n = 0$, and since $k = 0$ whenever $n = 0$, we have $f(F) \geq 0$, as required by our method of analysis. Note also that $f_{\max}(n) = n$. Property 2 helps simplify the complexity analysis by making it easier to find the worst cases:

Lemma 38. $\Psi(1) \geq \Psi(k) - \Psi(k-1)$, $\Psi(2) \geq \Psi(k) - \Psi(k-2)$ and $\Psi(3) \geq \Psi(k) - \Psi(k-3)$ for all k.

Proof. The first part of the lemma is trivial. As for the second part, $\Psi(2) \geq \Psi(2) - \Psi(0)$ obviously holds. To show $\Psi(2) \geq \Psi(3) - \Psi(1)$:

$$\begin{aligned}
\Psi(2) - \Psi(1) &\geq \Psi(3) - \Psi(2) \rightarrow \\
\Psi(2) &\geq \Psi(3) - (\Psi(2) - \Psi(1)) \rightarrow \\
\Psi(2) &\geq \Psi(3) - \Psi(1)
\end{aligned}$$

The inequalities $\Psi(2) \geq \Psi(4) - \Psi(2)$ and $\Psi(3) \geq \Psi(k) - \Psi(k-3)$ are shown similarly. □

k	$\Psi(k)$	$\Psi(k) - \Psi(k-1)$
1	0.24478	0.24478
2	0.45956	0.21478
3	0.62457	0.16501
4	0.76707	0.14250

Table 6.3: $\Psi(k)$ *values*

The optimised values of $\Psi(k)$ are given in Table 6.3. Having them available when the proof is presented makes it easier to verify the claims.

A note: In the previous section we found the branching numbers by applying our measuring function to Δn and Δl. Here the situation is somewhat more complicated. For an example, let $k(F)$ be the number of 2-clauses in F and say that $n(F) = 100$, $k(F) = 1$ and $n(F_1) = 95$, $k(F_1) = 2$. We have that $\Delta n = 5$ and $\Delta k = -1$. However, $\Delta f = (100 - \Psi(1)) - (95 - \Psi(2)) = 5 + (\Psi(2) - \Psi(1))$ which does not necessarily equal $5 + \Psi(1)$.

We now proceed to prove the time complexity of $D(F)$.

Theorem 39. $D(F)$ runs in $\mathcal{O}(1.6737^n)$ time.

Proof. We will give the branching numbers for the various cases of the algorithm, using the measure $f(F) = n - \Psi(k)$ with the values for $\Psi(k)$ given above.

Case 1 takes $\mathcal{O}(1)$ time. Cases 2 and 3 do not increase the time complexity.

Case 4: In both branches, at least the variables v and a are removed, as v will be removed by multiplier reduction when $\delta(v) = \delta_2(v) = 1$. If $\delta_2(a) > 0$, then at least one more variable is removed in some branch and we will have a branching number smaller than $\tau(2 - \Psi(4), 3 - \Psi(4)) < 1.5101$. Otherwise, if $\delta_2(a) = 0$, we have $\Delta k \leq 0$ in both branches and the branching number is at most $\tau(2, 2) < 1.4143$.

Case 5: If $\delta_2(v) = 2$, assume w.l.o.g. that we have the 2-clauses $x = (v \vee a), y = (v \vee b)$. In both branches, at least the variables v

and a are removed, as well as at least the two 2-clauses x and y. If a is involved in some other 2-clause, then at least one more variable is removed in some branch, leading to a branching number smaller than $\tau(2 - \Psi(4), 3 - \Psi(4)) < 1.5101$. Otherwise, when $\delta_3(a) > 0$, by Lemma 38 we have a worst-case branching number smaller than $\tau(2 - \Psi(2), 2 - \Psi(1)) < 1.5239$.

If $\delta_2(v) = 1$, assume w.l.o.g. that there exists a 3-clause $x = (v \vee a \vee b)$ in F and that we are branching on a. In one branch, v is removed, and in the other branch either v is removed by some reduction or a new 2-clause $y = (v \vee b)$ is created. Let us enumerate the possible cases:

1. If there is a 2-clause $(\bar{v} \vee \bar{b})$, $(\bar{v} \vee b)$ or $(v \vee \bar{b})$, then we will have a reduction, and by Lemma 38 the branching number is at most $\tau(2 - \Psi(2), 2 - \Psi(2)) < 1.5683$.

2. If $\delta_2(a) = 0$ and v is not removed by some reduction, then there are two variants depending on the sign of a in x and the other 3-clause y (which must exist by case 4.) If a has the same sign in both x and y we have branching numbers $\tau(2-(\Psi(k)-\Psi(k-1)), 1 - (\Psi(k) - \Psi(k+2))) \leq \tau(2 - (\Psi(4) - \Psi(3)), 1 - (\Psi(4) - \Psi(6))) < 1.6507$, else, if a has different signs, we have branching numbers $\tau(2, 1 - (\Psi(k) - \Psi(k+1))) < \tau(2, 1) < 1.6181$.

3. Otherwise, if $\delta_2(a) > 0$ and v is not removed by some reduction, at least one more variable is removed in some branch and we have branching numbers smaller than $\tau(2 - \Psi(2), 2 - \Psi(1)) < 1.5239$ and $\tau(3 - \Psi(2), 1 - \Psi(1)) < 1.6053$ (the two cases for $\delta_2(a) = 1$); $\tau(2-\Psi(3), 3-\Psi(2)) < 1.4413$, $\tau(3-\Psi(3), 2-\Psi(1)) < 1.4330$ and $\tau(4 - \Psi(3), 1 - \Psi(1)) < 1.4903$ (the three cases for $\delta_2(a) = 2$); $\tau(5-\Psi(4), 1-\Psi(3)) < 1.5576$ (the most unbalanced case for $\delta_2(a) = 3$); $\tau(6 - \Psi(5), 1 - \Psi(4)) < 1.5583$ (the most unbalanced case for $\delta_2(a) = 4$.)

Case 6: We will give the possible worst case branchings for each value of $\delta_2(v)$. Note that the worst case branching for a particular value of $\delta_2(v)$ will always have a minimum $\delta_3(v)$: If v is a literal in

Ref.	Tuple	Number
1	$(1-\Psi(1), 2+(\Psi(2)-\Psi(1)))$	1.6701
2	$(1-(\Psi(2)-\Psi(1)), 2+(\Psi(3)-\Psi(2)))$	1.6674
3	$(1-(\Psi(3)-\Psi(2)), 2+(\Psi(4)-\Psi(3)))$	1.6504
4	$(1-(\Psi(4)-\Psi(3)), 2+(\Psi(5)-\Psi(4)))$	1.6737
5	$(1-\Psi(2), 3-(\Psi(2)-\Psi(1)))$	1.6664
6	$(1-(\Psi(2)-\Psi(1)), 3-\Psi(2))$	1.5933
7	$(2-(\Psi(2)-\Psi(1)), 2-\Psi(2))$	1.5184
8	$(1-(\Psi(3)-\Psi(1)), 3-(\Psi(3)-\Psi(1)))$	1.6533
9	$(1-(\Psi(3)-\Psi(2)), 3-\Psi(3))$	1.6034
10	$(2-(\Psi(3)-\Psi(1)), 2-\Psi(3))$	1.5902
11	$(1-(\Psi(4)-\Psi(2)), 3-(\Psi(4)-\Psi(1)))$	1.6448
12	$(1-(\Psi(4)-\Psi(3)), 3-\Psi(4))$	1.6222
13	$(2-(\Psi(4)-\Psi(2)), 2-(\Psi(4)-\Psi(1)))$	1.5496
14	$(1-\Psi(3), 4-\Psi(3))$	1.6737
15	$(1-(\Psi(4)-\Psi(1)), 4-\Psi(4))$	1.6268
16	$(1-\Psi(4), 5-\Psi(4))$	1.6737

Table 6.4: *Branching tuples and branching numbers for case 6*

a 3-clause, then this 3-clause contributes nothing when $v =$ true and increases k by 1 when $v =$ false.

Since there are so many hard worst case branchings for this case, the branchings and the branching numbers are given in Table 6.4 for overview. The branching numbers are all at most 1.6737.

If $\delta_2(v) = 1$, the worst case is when $\delta(v) = 3$ (by case 5, $\delta(v) \neq 2$), and supposing that the 2-clause is $(v \vee a)$, we know that $\delta_2(a) = 1$, so only one 2-clause is removed in both branches. Also, $\delta_3(v) = 2$, resulting in two newly created 2-clauses. The worst case, because of the balanced branching effect, is the case where both 3-clauses include the literal v, so that we have $\tau(1-(\Psi(k)-\Psi(k-1)), 2+(\Psi(k+1)-\Psi(k)))$. The branching tuples and branching numbers for these cases are lines $1-4$ of Table 6.4. Cases with $k > 4$ result in $\tau(1, 2) < 1.6181$.

If $\delta_2(v) = 2$, we similarly have $\delta(v) = 3$ and, if the neighbours of v in the 2-clauses are a and b, $\delta_2(a) \leq 2$ and $\delta_2(b) \leq 2$. For the non-constant case $(v \vee a), (\bar{v} \vee b)$ we have, disregarding the 3-clause, two variables and two or three 2-clauses removed both when $v =$ true and $v =$ false. For the constant case $(v \vee a), (v \vee b)$, we have one variable and two 2-clauses removed when $v =$ true, and three variables and two to four 2-clauses removed when $v =$ false. In both cases, one 2-clause will be created in one of the branches, guaranteeing that $k \geq 1$ in that branch. Because of this guarantee, the results are different depending on the value of k. Lines $5-7$ of Table 6.4 contain the cases for $k = 2$, beginning with the constant case. Lines $8-10$ contain the cases for $k = 3$ and lines $11-13$ contain the cases for $k = 4$, both beginning with the constant case. Having $k > 4$ will not bring about any new worst cases, as $\Psi(k)$ flattens out and $\Psi(k) - \Psi(k_1)$ decreases.

If $\delta_2(v) \geq 3$, the worst case is when $\delta_3(v) = 0$. When $k = 3$, we have $\delta_2(v) = 3$ resulting in $\tau(1-\Psi(3), 4-\Psi(3))$ or $\tau(2-\Psi(3), 3-\Psi(3))$, where the former is clearly the worst case. When $k = 4$, we have $\delta_2(v) = 3$ resulting in $\tau(1-(\Psi(4)-\Psi(1)), 4-\Psi(4))$, which is a candidate for the worst case, or $\tau(2-\Psi(4), 3-\Psi(4)) < \tau(1, 2) < 1.6181$, and $\delta_2(v) = 4$ resulting in $\tau(1-\Psi(4), 5-\Psi(4))$, $\tau(2-\Psi(4), 4-\Psi(4))$ and $\tau(3-\Psi(4), 3-\Psi(4))$, where the first case is clearly the worst case. The worst of these cases are lines $14-16$ of Table 6.4.

Case 7: Note that $k = 0$ when this case is met. In the first two branches, either v is removed by some reduction, and/or case 4 is met.

1. If v is removed in the first branch, then the branching number is at most $\tau(2, 2, 3) < 1.6181$. If v is removed in the second branch, then case 4 is met in the first branch. As the worst branching number of case 4 is smaller than $\tau(2-\Psi(4), 3-\Psi(4)))$ we here have a branching number smaller than $\tau(3 - \Psi(4), 4 - \Psi(4)), 3, 3) < 1.6322$.

2. Otherwise, in both of the two first branches case 4 is met. We need to refine the analysis of case 4 for this case. Note that here, in case 7, at most one 2-clause is created that can be destroyed in the subsequent step. As $\tau(2-\Psi(1), 3-\Psi(1))) < \tau(2, 2)$ we here get a worst case branching number of $\tau(3, 3, 4, 4, 3) < 1.6181$.

Case 8: Since $\delta(v) \geq 3$ for all variables in F, the only situation where $\delta(v) = 3$ for the chosen variable v is when F is 3-regular. However, in the general case we may assume that $\delta(v) \geq 4$, because the 3-regular situation only occurs once along each path down the branching tree. To see this, note that with one exception, every modification of the formula that our algorithm performs is either a deletion of a variable or clause, or a shortening of a clause. The exception is when *Prop* performs the following:

> If there are two clauses $(a \vee b)$ and $(\bar{a} \vee \bar{b})$, then let $F \leftarrow F(a/\bar{b})$.

Now, note that if the degrees of a and b are at most 3 (as is the case when there has been a 3-regular situation), then the remaining variable b has degree at most two. Hence, we will not encounter this situation more than once.

When $\delta(v) \geq 4$, if there are i 3-clauses that contain v, and j 3-clauses that contain $\bar{v}$, then the branching number for this case is $\tau(1 + \Psi(i), 1 + \Psi(j))$. Because of the balanced branching effect, the worst case branching number is $\tau(1, 1 + \Psi(4)) < 1.6737$. □

We will now give a description of how the values for $\Psi(k)$ were found. The branchings that can not be proved to have a branching number of at most 1.6737 when $\Psi(k)$ is unknown are two cases from case 5, fourteen cases from case 6 and one case from case 8. Let $B_1(x), \ldots, B_{17}(x)$ be the value of each of these branching numbers given a vector $x = (\Psi(1), \ldots, \Psi(4))$. Then the running time of $D(F)$ is $\mathcal{O}(m(x)^n)$ where $m(x) = \max_{1 \leq i \leq 17} B_i(x)$, and the optimisation problem we have to solve is to find the x which minimises $m(x)$ under the conditions that $0 < \Psi(1) < \ldots < \Psi(4) < 1$. Note that since each $B_i(x)$ is continuous as long as these conditions hold, $m(x)$ is also continuous, which is a sufficient condition for standard tools to find at least a local optimum. The result of this optimisation is the values given in Table 6.3.

6.5 Algorithm C_{sep}

In this section we present an algorithm for #2SAT$_w$ for a special class of formulae, namely those with a separable constraint graph. Due to the kinship between 2SAT formulae and graphs, this class of formulae enjoys interesting properties as we shall see. Before that, however, we need some additional preliminaries.

A graph is *complete* if every pair of distinct vertices is joined by an edge. The complete graph with n vertices is denoted K_n. $H = (V_H, E_H)$ is a *subgraph* of $G = (V_G, E_G)$ iff $V_H \subseteq V_G$ and $E_H \subseteq E_G$. For an edge $e = (u, v)$, u and v are called its *endpoints*. If S is a set of edges of G, the operation of deleting S from G and identifying the endpoints is called *contracting* S. A *minor* of a graph G is any graph H obtained from G by a series of vertex deletions, edge deletions and edge contractions.

Let S be a class of graphs closed under the subgraph relation and $f(n)$ be a non-negative function. In [55], Lipton and Tarjan define an $f(n)$-*separator theorem* for S as follows:

Definition 40 (Separator theorem). There exist constants $\alpha < 1$ and $\beta > 0$ such that if G is any n-vertex graph in S, then the vertices

of G can be divided into three sets A, B and C such that no edge joins a vertex in A with a vertex in B, $max(|A|, |B|) \leq \alpha n$ and $|C| \leq \beta f(n)$.

A *separator algorithm* is a polynomial time algorithm Sep that takes a graph G as input and returns the tuple (A, B, C) of Definition 40. The constants α, β and f differ between different separator algorithms, but we require that $f(n) \in o(n)$. If there is an $f(n)$-separator theorem for S we say that S is a *separable* graph class and that $G \in S$ is a separable graph. The class of separable 2SAT formulae we define as the class of formulae having separable constraint graphs.

Let F be a separable 2SAT formula with the constraint graph G_F, such that $G_F = (V, E)$, where $|V| = n$, and let Sep be a separator algorithm for the class that G_F belongs to. The algorithm $C_{sep}(F, Sep)$ will return a tuple (m, w), such that m is the number of maximum weighted models for F and w is the weight of any such model. The algorithm recursively breaks down its input until a constant size, say b, of the input is reached. The constant sized subproblems are solved by calling C_E which will exhaustively search for the number and weight of the MWM's. To break down the input, Sep will be used to obtain A, B and C. If A and B were both disjoint and their union would equal G_F we would need just two recursive calls, but that is not possible in general. Instead, we are required to do a recursive call for every assignment of the variables of C, or rather every assignment of C that is extendible. The idea is that we check every possible configuration of C, deciding which variables should be set to 1, and which should be set to 0. Let F_A (F_B) denote the formula obtained from collecting just the clauses where variables in A (B) participate. The weight of a partial assignment κ_C is the sum of the weights of the literals which are true under κ_C.

C_{sep} uses a function $(c', w') = comb(a, (c_1, w_1), (c_2, w_2), (c, w))$ with the following definition:

$$(c', w') = \begin{cases} (c, w) & \text{if } a + w_1 + w_2 < w \\ (c + (c_1 \cdot c_2), w) & \text{if } a + w_1 + w_2 = w \\ (c_1 \cdot c_2, a + w_1 + w_2) & \text{if } a + w_1 + w_2 > w \end{cases}$$

Algorithm $C_{sep}(F, Sep)$

1. If $n(F) < b$, return $C_E(F)$
2. $c \leftarrow 0; w \leftarrow 0$
3. Let $(A, B, C) = Sep(G_F)$
4. For each extendible assignment κ_C to the variables of C:
   $(c, w) = comb(w(\kappa_C), C_{sep}(F_A, Sep), C_{sep}(F_B, Sep), (c, w))$
5. Return (c, w)

Figure 6.16: *The algorithm C_{sep} for #2SAT$_w$*

We will use *comb* to combine the results so far with the current recursive calls. Thus the algorithm reads as in Figure 6.16. We have the following theorem for the correctness of $C_{sep}(G_F, Sep)$:

Theorem 41. $C_{sep}(G_F, Sep)$ returns the tuple (c, w), where c is the number of MWM's in F and w the weight of any such MWM.

Proof. We note that if $n(F) < b$, then the output is correct by assumption. Else, we calculate A, B and C, and since every extendible assignment of C is checked, the algorithm is sound and complete. Note that if no assignment is extendible, $(0, 0)$ will be returned. □

We now give an upper time bound on the running time of C_{sep}:

Theorem 42. The running time $T(n)$ of C_{sep} is in $\mathcal{O}\left(poly(n) \cdot 2^{\beta f(n) z \log n}\right)$, where $z = -\frac{1}{\log \alpha}$.

Proof. Any tree of depth k having maximum degree t has at most $\frac{t^{k+1}-1}{t-1}$ nodes. Let us first establish an upper bound on k:

Looking at a particular node y in the recursion tree, it processes the subgraph $G_F^y \subseteq G_F$. We know that $|G_F| = n$, and that in each child y' of y it holds that $|G_F^{y'}| \leq \alpha |G_F^y|$. Following one path downwards the recursion tree, at the last level the node processes a sub-

graph of constant size, say 1. Hence, $\alpha^k n \leq 1 \Leftrightarrow \log \alpha^k n \leq \log 1 \Leftrightarrow k \log \alpha + \log n \leq 0 \Leftrightarrow k \leq -\frac{\log n}{\log \alpha}$

As for the degree, we know that $t = 2^{\beta f(n)}$, and hence the number of nodes is

$$\frac{(2^{\beta f(n)})^{k+1} - 1}{2^{\beta f(n)} - 1}$$

In each node the work is $\mathcal{O}\left(poly(n)\right)$ and so we have

$$T(n) \in O\left(poly(n) \cdot \frac{(2^{\beta f(n)})^{(z \log n)+1}}{2^{\beta f(n)} - 1}\right) = \mathcal{O}\left(poly(n) \cdot 2^{\beta f(n) z \log n}\right)$$

□

We next look at some particular instance classes to see how restrictions on the constraint graph can be exploited.

Graphs with an excluded minor H: Alon et al. [2] have presented a separator algorithm that guarantees $|C| \leq h^{3/2} n^{1/2}$ and $\alpha \leq 2/3$, where $h = |H|$. Hence, for this class, $\#2\text{SAT}_w$ can be solved in time $\mathcal{O}\left(2^{h^{3/2}\sqrt{n} z' \log_2 n}\right)$, where $z' \approx 1.7$. For two special subclasses we have even faster running times:

1. **Graphs of bounded genus:** Aleksandrov and Djidjev [1] have presented a separator algorithm for splitting a graph $G = (V, E)$ that is embeddable on a surface of bounded genus g. It runs in $\mathcal{O}\left(n + g\right)$ time and guarantees that $|C| \leq 4\sqrt{gn + n/\varepsilon}, \forall \varepsilon \in (0, 1)$ and $|F| \leq \varepsilon n$ for every connected component F obtained from $G - C$. We choose $\varepsilon = 1/2$ and get $\alpha \leq 2/3$ and so we have that $T(n) \in \mathcal{O}\left(2^{4\sqrt{gn+2n} z' \log_2 n}\right)$.

2. **Planar graphs:** The classical separator theorem for planar graphs by Lipton and Tarjan [55] has $\alpha \leq 2/3$ and $|C| \leq 2\sqrt{2n}$. This gives a running time in $\mathcal{O}\left(2^{2\sqrt{2n} z' \log_2 n}\right)$.

A complexity theoretical note is appropriate here: Vadhan [73] has proved that #MAXIMUM INDEPENDENT SET remains #P-complete even for planar bipartite graphs of degree ≤ 3. As we shall see, #2SAT$_w$ is at least as hard as #MAXIMUM INDEPENDENT SET, and so one can say the following about the complexity of the graphs in the previous paragraph: #2SAT$_w$ for the last class is obviously #P-complete, and the same holds for the second class. As for the first class, there are subclasses for which #2SAT$_w$ is polynomial time solvable (e.g. an excluded K_2 minor.) However, for many interesting subclasses e.g. graphs with no K_h minor (and $h > 2$), the problem is #P-complete.

We now look at a polynomial time solvable case:

Graphs with bounded tree width: For a graph G with a bounded tree width w, one can in $\mathcal{O}(n^w)$ time separate G such that $|C| \leq w + 1$ and $\alpha = 2/3$ (see for instance [9].) Using Theorem 42 we would get an upper time bound in $\mathcal{O}\left(n^{2w}\right)$, but we can do better. The running time $T(n)$ can be described by the recurrence

$$T(n) \leq 2^{w+1} T(2n/3) + n^w.$$

Applying the Master Theorem for divide and conquer recurrences (see for instance [13]), we get a running time in $\mathcal{O}\left(n^{\log_{3/2} 2^{w+1}}\right) \subseteq \mathcal{O}\left(n^{1.71w+1.71}\right)$. This is better than $\mathcal{O}\left(n^{2w}\right)$ for $w \geq 6$.

The matter of counting in this class has been previously dealt with by Díaz et al. [30]. Their main result is an algorithm that counts homomorphisms in linear time for fixed w. This algorithm can be used for counting, among other things, the number of independent sets in a bounded tree width graph in linear time. Their results are however not fully comparable with ours: Using their algorithm for counting *maximum* independent sets would take time exponential in the number of vertices of the graph (see Corollary 5.10 in [30]), whereas our reduction (to be shown later in this chapter) implies an $\mathcal{O}\left(n^{1.71w+1.71}\right)$ time algorithm. On the other hand, modifying our reduction for maximum independent sets to allow all independent sets still gives an $\mathcal{O}\left(n^{1.71w+1.71}\right)$ time algorithm.

6.6 Applications

We here give some interesting applications for the counting of MWM's for 2SAT and 3SAT formulae. We start with an application for D. This reduction first appeared in an article by Valiant [75].

– #CIRCUIT SATISFIABILITY can be solved in $\mathcal{O}\left(1.6737^{n+m}\right)$ time, where n is the number of inputs and m the number of gates.

  Instance: A one-output boolean combinatorial circuit consisting of AND, OR and NOT gates.

  Question: How many (different) inputs make the output 1?

  Reduction: Each gate has (at most) two inputs and one output. Hence it can be mimicked using (at most) three variables. So, at the first level of the circuit each input translates to a variable and then each gate gives a variable.

There is a kinship between #2SAT and many graph problems which we will exploit in the following. We first look at the #MAXIMUM INDEPENDENT SET and #INDEPENDENT SET problems.

– #MAXIMUM INDEPENDENT SET and #INDEPENDENT SET for general graphs is solvable in $\mathcal{O}\left(1.2561^{n}\right)$ time; for graphs with an excluded H minor in time $\mathcal{O}\left(2^{|H|^{3/2}\sqrt{n}1.7\log_2 n}\right)$, for graphs of bounded genus g in $\mathcal{O}\left(2^{4\sqrt{gn+2n}1.7\log_2 n}\right)$ time, for planar graphs within time $\mathcal{O}\left(2^{2\sqrt{2n}1.7\log_2 n}\right)$ and for graphs with a bounded tree width of w in $\mathcal{O}\left(n^{1.71w+1.71}\right)$ time.

  Instance: A graph $G = (V, E)$, with a weight $w(x)$ for each vertex $x \in V$.

  Question: What is the number of independent sets/maximum independent sets?

  Reduction: For the #MAXIMUM INDEPENDENT SET problem, let each vertex x_i of G form a boolean variable x_i, each edge (x_j, x_k) give rise to a clause $(\bar{x}_j \vee \bar{x}_k)$, and each $z \in V$ that

has no edge the clause (z) (since z must be in every maximum independent set.) The weights associated with each vertex x_i of G is transferred to the positive literal x_i and $w(\bar{x}_i) = 0$. Clearly, the number of MWM's equals #MAXIMUM INDEPENDENT SET. For the #INDEPENDENT SET problem one simply assigns $w(x) = w(\bar{x}_i) = 1$ and each $z \in V$ that has no edge gives rise to the clause $(z \vee \bar{z})$.

As we now have algorithms for the #MAXIMUM INDEPENDENT SET problem, we can employ the following reductions:

- #EXACT COVER for general graphs can be solved in $\mathcal{O}(1.2561^n)$ time (and for separable graphs time bounds are given above.)

  Instance: A finite set U and a collection C of subsets $c_1, \ldots, c_n$ of U.

  Question: If C contains an exact cover for U, i.e. a subcollection $C' \subseteq C$ such that every element of U occurs in exactly one member of C', what is the number of exact covers?

  Reduction: Let (U, C) be an arbitrary instance of the #EXACT COVER problem. Construct a weighted graph $W = (V, E)$ as follows: Let each $c_i \in C$ give rise to a vertex $v_i \in V$ whose weight is $|c_i|$. Add an edge between v_i and v_j if and only if $c_i \cap c_j \neq \emptyset$. Clearly, no independent set in W can have a weight greater than $|U|$. Furthermore, the independent sets of weight $|U|$ in W corresponds to solutions of (U, C).

- #EXACT HITTING SET can be solved in $\mathcal{O}(1.2561^n)$ time.

  Instance: A finite set U and a collection C of subsets $c_1, \ldots, c_n \subseteq U$ such that $\bigcup_{c_i} = U$

  Question: A solution is a minimum size subset $H \subseteq U$ hitting each c_i exactly once, i.e. $|c_i \cap H| = 1$. What is the number of solutions?

  Reduction/Modification: To ensure the minimum size property we will have to add a slight modification to the algorithms such that whenever a heavier solution is preferred over lighter ones,

there is also a preferment of having as few true positive literals as possible. Then, let (U, C) be an arbitrary instance of the #EXACT HITTING SET problem. Construct a weighted graph $W = (V, E)$ as follows: Let each element $x_j \in U$ form a vertex. Add an edge between each pair of elements that are contained within the same c_i. The weight $w(x_j)$ is the number of subsets x_j appears in. Obviously, a maximum independent set mis such that the weight of mis equals $|C|$ covers all subsets.

– #WEIGHTED SET PACKING for general graphs can be solved in time $\mathcal{O}(1.2561^n)$.

  Instance: A finite set U and a collection C of subsets $c_1, \ldots, c_n \in U$ and for each c_i there is an associated weight $w(c_i)$.

  Question: A solution is a collection $C' \subseteq C$ of disjoint sets of maximum weight. What is the number of such solutions?

  Reduction: Let (U, C) be an instance of the #WEIGHTED SET PACKING problem. Construct a weighted graph $W = (V, E)$ as follows: Introduce one vertex v_i for each $c_i \in C$ and assign it weight $w(c_i)$. Add an edge between v_i and v_j if and only if $c_i \cap c_j \neq \emptyset$. Obviously, a maximum weighted, independent set found in W constitutes a solution.

Part IV

Optimising

Chapter 7

Maximum Exact Satisfiability

In this chapter we study MAX XSAT and its variant RESTRICTED MAX XSAT. Unlike previous chapters, the main results are obtained by reductions to other problems. For RESTRICTED MAX XSAT this means a non-trivial upper time bound for solving it, while for MAX XSAT we get yet an indication about its hardness.

7.1 On the Hardness of MAX XSAT

To the best of our knowledge, measuring in the number of variables, there is no algorithm for MAX XSAT with a better running time than the trivial $\mathcal{O}(2^n)$ bound. As we will show in this section, such results would be surprising, given the lack of results for MAX SAT. In the concluding Chapter 9 we will also compare the results for algorithms measuring the running time in $m = m(F)$. For now, we concentrate on measuring in $n(F)$ and note the following for MAX XkSAT and MAX kSAT:

1. Using polynomial space there are no non-trivial results even for $k = 2$.

2. Resorting to the use of exponential space, Williams [77] has shown that MAX X2SAT and MAX 2SAT are solvable in time $\mathcal{O}(1.7314^n)$.

3. For $k > 2$ no results are known.

The following theorem shows that a $\mathcal{O}(c^n)$ time algorithm for MAX XkSAT implies a $\mathcal{O}(c^n)$ time algorithm for MAX kSAT.

Theorem 43. Every instance F of MAX kSAT is polynomial time reducible to an instance G of MAX XkSAT such that $n(F) = n(G)$.

Proof. Let F be an arbitrary instance of MAX kSAT and consider the following reduction from MAX kSAT to MAX XkSAT: for every clause $C = (a_1 \vee a_2 \vee \ldots \vee a_{j-1} \vee a_j)$ in F we mimic the allowed assignments by creating a number of new XSAT clauses. The first class of allowed assignments for C is the ones making exactly one of the literals true. This is captured by the XSAT clause $C_1^1 = (a_1 \vee a_2 \vee \ldots \vee a_{j-1} \vee a_j)$. The second class of allowed assignments is the ones making two of the literals true. This can be mimicked by creating the XSAT clauses

$$
\begin{aligned}
&C_2^1 = (\bar{a}_1 \vee \bar{a}_1 \vee a_2 \vee \ldots \vee a_{j-1} \vee a_j) \\
&C_2^2 = (a_1 \vee a_1 \vee \bar{a}_2 \vee \bar{a}_2 \vee \ldots \vee a_{j-1} \vee a_j) \\
&\vdots \\
&C_2^{j-1} = (a_1 \vee a_1 \vee a_2 \vee a_2 \vee \ldots \vee \bar{a}_{j-1} \vee \bar{a}_{j-1} \vee a_j)
\end{aligned}
$$

These clauses can be understood as "a_1 true and yet another unspecified literal true; a_1 false, a_2 true and yet another unspecified literal true" etc. The other classes are modelled accordingly, ending with $C_i^1 = (a_1 \vee \bar{a}_2 \vee \bar{a}_2 \vee \ldots \vee \bar{a}_{j-1} \vee \bar{a}_{j-1} \vee \bar{a}_j \vee \bar{a}_j)$ for all literals true. We say that the SAT clause C *yields* the XSAT clauses C_h^k. Note that we here view G as a multiset, allowing multiple occurrences of a clause. Another possibility would be to assign weights to the clauses.

In order to prove the theorem we first need to prove that every assignment that satisfies μ clauses in F satisfies *at least* μ clauses in G. It is easy to see that this holds: By construction of G, for every assignment satisfying any clause $C \in F$ there is at least one clause yielded by C that is satisfied.

$$x = (\bar{a}) \quad y = (a \vee b \vee c) \quad z = (a \vee \bar{b}) \quad w = (a \vee \bar{c})$$

Figure 7.1: *An instance of* MAX SAT *with answer 3*

$$x_1^1 = (\bar{a})$$

$$\begin{array}{lll} y_1^1 = (a \vee b \vee c) & z_1^1 = (a \vee \bar{b}) & w_1^1 = (a \vee \bar{c}) \\ y_2^1 = (\bar{a} \vee \bar{a} \vee b \vee c) & z_2^1 = (a \vee b \vee b) & w_2^1 = (a \vee c \vee c) \\ y_2^2 = (a \vee a \vee \bar{b} \vee \bar{b} \vee c) & & \\ y_3^1 = (a \vee \bar{b} \vee \bar{b} \vee \bar{c} \vee \bar{c}) & & \end{array}$$

Figure 7.2: *An instance of* MAX XSAT *with answer 3*

Secondly, we need to prove that an assignment A that satisfies μ clauses in F satisfies *at most* μ clauses in G. We note that due to the construction of G, no clause $C \in F$ that is unsatisfied under A yields a clause that is satisfied under A. Further, for a set Γ of clauses yielded by $C \in F$, there is no assignment that satisfies two clauses of Γ simultaneously. $\square$

We will illustrate how the reduction works by an example. In Figure 7.1 an instance of MAX SAT is shown. We pick each clause and apply the rules. Starting with $x = (\bar{a})$, it simply translates to $x_1^1 = (\bar{a})$. For $y = (a \vee b \vee c)$ we get $y_1^1 = (a \vee b \vee c)$ which captures the case when only one literal is true. For two literals true we get $y_2^1 = (\bar{a} \vee \bar{a} \vee b \vee c)$ and $y_2^2 = (a \vee a \vee \bar{b} \vee \bar{b} \vee c)$ and all three true is modelled by $y_3^1 = (a \vee \bar{b} \vee \bar{b} \vee \bar{c} \vee \bar{c})$. The translation of the two remaining clauses is trivial and the overall result is shown in Figure 7.2.

7.2 Solving RESTRICTED MAX XSAT

As we have already pointed out, even when restricting MAX XkSAT to $k = 2$, for polynomial space the best known upper time bound is still

the trivial $\mathcal{O}(2^n)$ bound. It is therefore somewhat surprising that another, seemingly minor restriction can allow much faster algorithms. As already mentioned, Madsen and Rossmanith [59] have shown how to solve Restricted max xsat in time $\mathcal{O}(1.3248^n)$. We will improve this to $\mathcal{O}(1.2561^n)$ by means of a reduction to 2sat$_w$. We recall that Restricted max xsat is Max xsat with the restriction that no clause may be over-satisfied. Note that the restriction makes at least one canonical rule available: given the clauses $(a \vee b)$ and $(a \vee \bar{b})$, a must be false, because otherwise one clause gets over-satisfied. Thus, this problem seems to have a nicer structure compared to Max xsat which, as far as we know, allows no canonical rules at all. The known algorithms also give a strong indication of this.

Lemma 44. An instance F of Restricted max xsat is polynomial time reducible to an instance G of 2sat$_w$ such that $n(F) = n(G)$.

Proof. Construct the 2sat$_w$ instance G as follows:

For every pair of literals a and b occurring in a clause of F, make the clause $(\bar{a} \vee \bar{b})$. Each variable a in G carries two integer weights: $w(a)$ and $w(\bar{a})$. Let $w(a)$ be number of clauses containing a and $w(\bar{a})$ be the number of clauses containing $\bar{a}$.

First note that by the construction of the clauses of G, for G to have a model at all, no clause of F must be over-satisfied. Second, any literal a that is true under a model for G satisfies $w(a)$ number of clauses in F, and correspondingly for a false literal. Therefore, a max-weighted model of G correspond to an assignment to F such that a maximum number of clauses are x-satisfied.

The number of clauses in G is at most $(2n)^2$ (assuming that clauses are not duplicated) and so the reduction is polynomial. No new variables are introduced. □

For an example of the reduction, consider the following instance of Restricted max xsat, with answer 1:

$$(a), (a \vee b), (a \vee \bar{b})$$

Applying the reduction we get the weighted 2sat formula of Figure 7.3. Note that every model must have $a = false$. We now arrive

Formula:	$(\bar{a} \vee \bar{b}), (\bar{a} \vee b)$
Weights:	$w(a) = 3$ $w(\bar{a}) = 0$
	$w(b) = 1$ $w(\bar{b}) = 1$

Figure 7.3: *For this weighted* 2SAT *formula,* $a = false, b = false$ *is a maximum weighted model of weight 1*

at the following theorem which proves a new upper time bound for RESTRICTED MAX XSAT:

Theorem 45. RESTRICTED MAX XSAT for a formula F is solvable in polynomial space and time $\mathcal{O}(1.2561^n)$.

Proof. Follows from Lemma 44 and Theorem 36. □

Chapter 8

Max Hamming Exact Satisfiability

In this chapter we present two algorithms for solving MAX HAMMING XSAT, prove them correct and provide upper bounds on their running time. For the sake of conciseness, we phrase the algorithms in such a way that they answer the question "what is the maximum Hamming distance between any two models?" However, it is trivial to see how they can be modified to actually produce two such models.

8.1 Using an External XSAT Solver

Instead of directly presenting an algorithm for MAX HAMMING XSAT, let us first discuss another, similar problem. Consider the following XSAT instance, $\mathcal{I}$:

$$x = (a \vee b), y = (\bar{a} \vee c).$$

The question is whether there are two models M and M' that differ in their assignment to the variable a. One systematic approach to this problem is to build a new instance $\mathcal{I}'$ such that $\mathcal{I}'$ is x-satisfiable iff there are two such models. One such instance $\mathcal{I}'$ is this:

$x = (a \vee b), y = (\bar{a} \vee c), x' = (a' \vee b), y' = (\bar{a}' \vee c), z = (a \vee a').$

We see that the clauses containing a are copied and that a new variable a' is introduced, replacing a in the copied clauses. The addition of the clause z now forces $a \neq a'$ and we see that $\mathcal{I}'$ is x-satisfiable iff $\mathcal{I}$ allows two models differing on a. We can use this principle to produce an algorithm for MAX HAMMING XSAT: For every subset X (in our example $X = \{a\}$) of the variables, build a new instance $\mathcal{I}'$ by copying the clauses where variables from X occur, replacing the variables from X with new variables X' and finally adding clauses such that $a \neq a'$ for every variable $a \in X$ and its copy $a' \in X'$. The time for this would be in $\mathcal{O}\left(2^n 1.1730^{2n}\right) \subseteq \mathcal{O}\left(2.7519^n\right)$: There are 2^n possible subsets X and for each such subset we need to check whether the corresponding $\mathcal{I}'$ is x-satisfiable, and $\mathcal{I}'$ might be twice the size of the original instance. However, we can do better.

The first step towards a faster algorithm is an observation made by Angelsmark and Thapper [5]: We do not need to introduce any new variables. Since the constraint is $a \neq a'$ and we have a Boolean domain, in the copied clauses, we flip every occurrence of a literal from X. For our example above, $\mathcal{I}'$ is constructed thus:

$x = (a \vee b), y = (\bar{a} \vee c), x' = (\bar{a} \vee b), y' = (a \vee c)$

By the observation that $\mathcal{I}'$ does not have to contain new variables (and this works for SAT also), Angelsmark and Thapper [5] were able to solve MAX HAMMING SAT in time $\mathcal{O}\left(2^{2n-2\sqrt{n/\log n}}\right)$: 2^n subsets to consider and SAT is solvable in time $\mathcal{O}\left(2^{n-2\sqrt{n/\log n}}\right)$ (see [21].) Using the same technique, MAX HAMMING XSAT can be solved in time $\mathcal{O}\left(2.3460^n\right)$. However, we can do even better, because we do not have to consider every instance $\mathcal{I}'$ and thus can skip a number of calls to the XSAT solver. This is shown in the following lemma:

Lemma 46. Assume that M and M' are x-models for F and that X is the subset of variables assigned different values. Then each clause of F contains either zero or two literals derived from X.

Proof. For the sake of contradiction, assume there is a clause containing exactly one literal a from X. The clause cannot be (a) because then it would be unsatisfied under one model. Therefore the clause must be $(a \vee A)$ where all members of A have the same value under

Algorithm $P(F)$

```
1  ans ← ⊥
2  for k ← 0 to n  do
3     for every subset X ⊆ Var(F) of size k  do
4        Let C be the set of clauses containing any literal derived
          from X
5        Let C′ be a copy of C where every literal derived from X is
          flipped
6        if all clauses of C contain either 0 or 2 literals from X then
7           if F ∪ C′ is x-satisfiable then ans ← k
8  return ans
```

Figure 8.1: *The algorithm P for* MAX HAMMING XSAT

both models. If one literal of A is true, then a must be false under both models (to avoid over-satisfaction), clearly a contradiction. If all literals of A are false, a must be false so this also is a contradiction.

Again for the sake of contradiction, assume there is a clause containing three literals a, b and c, from X. In M one, say a is true, and the other consequently false. However, $a =$ false, $b = c =$ true can never hold in M'. We see that this also holds for the case of more than three literals from X in any clause. □

We will return to the question of how many of the $\mathcal{I}'$s that can comply to Lemma 46. First, however, we will present our algorithm, which can be seen in Figure 8.1. If the formula is x-unsatisfiable, then $\perp$ is returned. The answer 0 of course indicates that there is only one model.

Theorem 47. $P(F)$ decides MAX HAMMING XSAT for F.

Proof. For completeness: Assume there are two models M and M' at maximum hamming distance k and that the differing variables are collected in X. The clauses containing zero literals from X remain the same under both models, the interesting case is a clause $(a \vee b \vee C)$ where a and b are from X (by Lemma 46 this is the only possible case.) Assume w.l.o.g. that a is true and b is false under M and the

l	1	2	3	4	5	6	7	8	9	10
$g(l)$	1	1.4143	1.5875	1.6266	1.6154	1.5875	1.5552	1.5234	1.4937	1.4665

Table 8.1: *The first ten values of g*

opposite holds for M'. Then the clause $(\bar{a} \vee \bar{b} \vee C)$ is x-satisfied under both models.

For soundness: Assume we have a model M for $F \cup C'$. Consider the clauses in the set $C \cup C'$. We see that each clause have two members from X such that one is true and the other false. Thus, it is possible to form another model M' by assigning all variables of X the opposite values. □

We can now start examining the running time of P. Let an *allowed subset* S of variables in a formula F be a subset such that each clause of F contains either zero or two members of S. The following lemma establishes an upper bound for the number of allowed subsets.

Lemma 48. For any formula F the number N of allowed subsets is in $\mathcal{O}(\lambda^n) = \mathcal{O}\left(7^{n/4}\right) \subseteq \mathcal{O}(1.6266^n)$.

Proof. Consider an arbitrary variable a. When calculating the number N of allowed subsets a can participate in, it is clear that the higher the degree of a, the lower the N. Hence, a formula consisting only of singletons has maximum N.

Which clause length l maximises N? We see that $N = (\binom{l}{2} + 1)^{n/l} = (l^2/2 - l/2 + 1)^{n/l}$. Hence, we now need to maximise $g(l) = (l^2/2 - l/2 + 1)^{1/l}$ for integer values of l. In Table 8.1 are given the first 10 values for g. They can be interpreted as "Clause length 2 makes $N \in \mathcal{O}\left(2^{n/2}\right) \subseteq \mathcal{O}(1.4143^n)$" etc. We see that for these 10 values, g has a maximum at clause length 4, making $N \in \mathcal{O}\left(7^{n/4}\right) \subseteq \mathcal{O}(1.6266^n)$.

We need to establish that this is a global maximum, i.e. that this series is decreasing for all values $l > 10$. We do that by overestimating

g. Note that $f(l) = (l^2)^{1/l}$ is greater than g for $l > 2$. The derivative of f is $f'(l) = (l^2)^{1/l} \cdot (2/l^2 - 2\ln(l)/l^2)$ and we see that for $l > e$, where $e = 2.712\ldots$, f' is always negative, which means that f is always decreasing. As $f(10) < 1.5849$, we know that we have found a global maximum. □

Theorem 49. *$P(F)$ runs in polynomial space and time $\mathcal{O}(2^n)$.*

Proof. Clearly P uses polynomial space. Furthermore, the running time is $\mathcal{O}(2^n + \lambda^n C^n)$, where λ is the constant of Lemma 48 and C is a constant such that XSAT is solvable in polynomial space and time $\mathcal{O}(C^n)$. The currently best value for C is 1.1730 and so the upper time bound is $\mathcal{O}(2^n + 1.6266^n \cdot 1.1730^n) \subseteq \mathcal{O}(2^n + 1.9079^n) = \mathcal{O}(2^n)$. □

Note that if we are not searching for the *maximum* Hamming distance, but are content with a constant distance d, then P can be easily modified for this, and we obtain a running time in $\mathcal{O}(1.1730^n)$.

8.2 Using Branching

We will now move on to another polynomial space algorithm Q with a provably better running time than P. Its description and preliminaries are quite lengthy, and so we will divide this section into subsections. The first subsection will introduce the principles and extra structures used in the algorithm. In subsection 8.2.2 a working, but somewhat inefficient algorithm Q' is presented. This is done because adding the extra lines of code to speed up the algorithm will obscure the structure of it. In this section we also prove the correctness of Q' and its auxiliary algorithms. Finally, in subsection 8.2.3, we add the extra lines to speed up the algorithm and then prove an upper time bound.

8.2.1 Extra Preliminaries

The algorithm Q is a DPLL style algorithm relying on the following observation: In two models M and M' at a maximum Hamming distance, we have a limited number of possibilities for a variable a:

1. a is true under both
2. a is false under both
3. a has different values under the two models

For an example, branching on a, the clause $w = (a \vee b \vee c \vee d)$ gives 5 recursive calls:

1. $Q(F(a/true))$
2. $Q(F(a/false))$
3. $Q(F \wedge (a \vee b))$
4. $Q(F \wedge (a \vee c))$
5. $Q(F \wedge (a \vee d))$

In the first call, when a is true under both models, we know that the other literals will be false and so they can be removed in the usual way. The second call is also straightforward, we assign a false, but cannot in general remove any other variables. The last three calls cover the case when a has different values under the two models. By Lemma 46 there is exactly one more variable a' in w that has different values and we need to examine all possible cases of a'. Of course, in the last three calls we may also assign false to the literals that are not a or a'.

In Q, during the branching, canonical rules are used. Two of these will force the use of some additional structures, similar to the situation when counting x-models and models in Chapters 5 and 6. Therefore the following is needed: For every clause y in the instance F, we will have a clause y_σ that initially is empty. The formula F_σ consists of all the y_σ's along with a number of 2-clauses which we will return to soon. To give a first hint about the use of the y_σ's, consider a clause $(a \vee b \vee c)$, where b and c are singletons. Assume that we remove c, and that later, in a leaf of the recursive tree of the algorithm, a model M is found such that b is true. Then we know that

there is another model M' assigning $b = false$ and $c = true$, i.e. the Hamming distance between M and M' is at least 2. Hence, we have to keep track of removed singletons. As for the 2-clauses of F_σ, let us for now say the following: Q will never directly assign a singleton true. Instead there will arise 2-clauses like $w = (a \vee b)$, where a is a non-singleton and b is a singleton. In this situation, w will be moved to F_σ, renamed to $(a \vee b)_\rho$. Note that a still occurs in F and that its future value will decide the value of b, and thus it is important to know that the relation between a and b is $(a \vee b)_\rho$. We will use the subscripts σ and ρ to distinguish between the two types of clauses of F_σ.

We will tacitly assume that there is a line 0 of the algorithm applying the following canonical rules: 2, 3, 5 and 11. These are the rules that remove the constants true and false, 1-clauses, variables that occur more than once in a clause, and make sure that no clause is a strict subset of the other. For these, no extra structures are needed.

As a consequence of the use of the canonical rules, in the leaves of the recursion tree a kind of generalised models are found, that summarise several models, and this is represented by F_σ. We will return to this later. For now, we hide the details in the auxiliary algorithm U which we will come back to after the presentation of the main algorithm Q. The reason for doing so, is that we first need to see how the canonical rules and the branchings work in detail.

Some more technicalities: Like the algorithm P, Q may return $\perp$ if F is unsatisfiable. Therefore we define $\perp < 0$ and $\perp + j = \perp$, for all j; furthermore, $\max_\perp(\perp, Z)$ returns Z, even if $Z = \perp$. For both types of clauses of F_σ, the order of the literals is of importance. That means for instance that $(a \vee b)_{rho} \neq (b \vee a)_{rho}$. To motivate this, consider again the case of a clause $(a \vee b)$ in F, such that b is a singleton and a is not. As we shall see, the algorithm will later need to know which variable was singleton and which was not. As for the order within σ clauses, the first literal in such a clause will be the last one to have been removed from F, and this is also an important piece of information. We assume furthermore that no variable occurs twice

in any clause. If the literal a is added to $y_\sigma = (a \vee b \vee c)_\sigma$, the result is still $(a \vee b \vee c)_\sigma$.

8.2.2 The Algorithm Q'

For clarity of presentation we first give a simplified algorithm Q', which is shown in Figure 8.2. We will later add an optimisation to improve the running time. When applying the algorithm, the original instance F is assumed to contain no clause where all variables are singletons. To see that that is no restriction, consider the following lemma:

Lemma 50. For a formula F with a clause x ($|x| > 1$) such that all members of x are singletons, two x-models M and M' at a maximum Hamming distance will assign different members of x true.

The proof is trivial, and we see that one consequence is that all clauses consisting only of singletons can be removed, adding 2 to the maximum Hamming distance.

In order to illustrate the use of F_σ, the reader may have a look at Figure 8.3. To get an idea about the possible construction of this particular F_σ, consider this series of steps: In F there is a clause $x = (\alpha \vee h \vee a \vee b \vee c \vee d \vee e)$ such that $a - e$ are singletons. In a first canonising step, the clause x_σ becomes $(a \vee b \vee c \vee d \vee e)$ and x becomes $x = (\alpha \vee h \vee a)$. Next, Q assigns α false and x becomes $(h \vee a)$, which will be removed by canonisation; to F_σ is now added the clause $(h \vee a)_\rho$. The variable h also occurs in the clause $z = (\delta \vee h \vee r)$, such that r is a singleton and $z_\sigma = (r \vee s \vee t \vee u)$. Q then assigns δ false and the clause $(h \vee r)_\rho$ is added to F_σ. The variable h is now a singleton occurring in the clause $y = (\beta \vee \gamma \vee l \vee h)$. Since previously, F_σ contains the clauses $y_\sigma = (l \vee m \vee n)_\sigma$, $(m \vee o)_\rho$ and $(o \vee p \vee q)_\sigma$ and canonisation now adds h into y_σ, removing it from y. Finally, γ is assigned true, and consequently l is set to false.

Now, how to compute the maximum Hamming distance that this particular F_σ contributes? As $l =$ false, m, n and h must also be false (remember that l was the singleton left in F to represent all singletons.) That means that a, o and r are true, and each of the

Algorithm $Q'(F)$

1 For a clause $y = (a_1 \vee a_2 \vee \ldots \vee a_j \vee b \vee \ldots)$ such that $a_1 - a_j$ are singletons, add all the singletons to y_σ and remove $a_2 - a_j$ from y. The copying is done so that a_1 will be the first literal of y_σ.
2 For a 2-clause $w = (a \vee b)$ such that neither a nor b is a singleton, let $F \leftarrow (a/\bar{b})$ and remove w. If one, say b, is a singleton, then remove w and create the copy $(a \vee b)_\rho$ in F_σ.
If a singleton was created, by any of these two operations, goto the previous line.
3 **if** $F = \{()\}$ **then return** $\perp$
4 **elsif** $F = \{\}$ **then return** $U(F_\sigma)$
5 **elsif** F is not connected **then** assume the components are $F_1, \ldots, F_k$ and return $\sum_{i=1}^{k} Q'(F_i)$
6 **else**
7 Pick a longest clause $w = (a_1 \vee a_2 \vee \ldots \vee a_k)$ and assume w.l.o.g. that a_1 is a non-singleton. Now do the following:
8 $ans_{true} \leftarrow Q'(F(a_1/true))$
9 $ans_{false} = Q'(F(a_1/false))$
10 **if** $ans_{true} = \perp$ **or** $ans_{false} = \perp$ **then**
11 **return** $\max_\perp(ans_{true}, ans_{false})$
12 **else**
13 **for** $i = 2$ to k **do**
14 Let $ans_i \leftarrow Q'(F \wedge (a_1 \vee a_i))$
15 **return** $\max_\perp(ans_{true}, ans_{false}, (ans_2 + 1), \ldots, (ans_k + 1))$

Figure 8.2: *The algorithm Q' for* MAX HAMMING XSAT

$$
\begin{array}{ll}
 & y_\sigma = (l \vee m \vee n \vee h)_\sigma \\
(m \vee o)_\rho & w_\sigma = (o \vee p \vee q)_\sigma \\
(h \vee a)_\rho & x_\sigma = (a \vee b \vee c \vee d \vee e)_\sigma \\
(h \vee r)_\rho & z_\sigma = (r \vee s \vee t \vee u)_\sigma
\end{array}
$$

Figure 8.3: *A possible F_σ*

clauses w_σ, x_σ and z_σ will contribute 2 to the maximum Hamming distance. In toto, the maximum Hamming distance in this case is 6.

We will next present a more complicated example. Before that, however, we need some more preliminaries. In the example of Figure 8.3, the constraint graph of F_σ was connected. However, as is easily seen, that is not the case in general. Note that each component G can be seen as a tree T_G in the following way: Each clause y_σ is a vertex and each w_ρ is an edge, and due to the construction, there are no cycles. We will consider the edges directed towards the leaves. When z_ρ points at a clause y_σ (such as $(m \vee o)_\rho$ does at $w_\sigma = (o \vee p \vee q)_\sigma$) we will say that y_σ is the *goal* of z_ρ. Furthermore, let $dual(a)$ denote the set of ρ clauses having a as the first element. Let $Link(a)$ denote the set $dual(a) \cup \{b \in C \mid C \in goal(a') \text{ for all members } a' \in dual(a)\}$. We say that a variable b is *transitively linked* to a if $b \in Link(a)$ or b is transitively linked to a member in $Link(a)$. For instance, in the example of Figure 8.3, $\{o, p, q\}$ is the set of variables transitively linked to m. Finally, the last variable to remain in F is called the *entry-point* of the component.

In the example of Figure 8.3, the last variable to remain in F, l, was set to false. As Q never directly assigns any singleton true, if the last remaining variable is to be set true, we must have a situation similar to the one in Figure 8.4. Here we see that the last remaining variable might have been assigned false. Assuming that is the case, we see that l is true and that the situation is more complicated than in the previous figure.

In order to compute the maximum Hamming distance, we first note that all models must have one of the members of y_σ true. It might be that two models M and M' at maximum distance assigns the same member true. We also have the possibility that they assign two different members true. In the first case we need a function Fix that assigns a variable a a fixed value and then proceeds downwards in the tree T_G to find the maximum number of variables that can be assigned different values under two different models. We can then tentatively assign each member of y_σ true and use Fix. For $y_\sigma = (l \vee m \vee n \vee h)_\sigma$ that would mean that we assign $l = $ true and $m = n = h = $ false and

$$
\begin{array}{ll}
(v \vee \bar{l})_\rho & y_\sigma = (l \vee m \vee n \vee h)_\sigma \\
(m \vee \bar{o})_\rho & w_\sigma = (o \vee p \vee q)_\sigma \\
(h \vee \bar{a})_\rho & x_\sigma = (a \vee b \vee c \vee d \vee e)_\sigma \\
(h \vee r)_\rho & z_\sigma = (r \vee s \vee t \vee u)_\sigma
\end{array}
$$

Figure 8.4: *A possible* F_σ

then calculate $Fix(l) + Fix(m) + Fix(n) + Fix(h)$, next we try $m =$ true and $l = n = h =$ false and so on. The maximum value found is kept as k_1. The function Fix can be seen in Figure 8.5.

The function H_f used by Fix, is given a clause w_σ such that no member of it must be true (i.e. a situation we saw for y_σ in Figure 8.3), and it returns the maximum number of variables, transitively linked to any member of w_σ that can be assigned different values under any two models. H_t is actually the function we are in the middle of describing. It is similar to H_f, but its argument is a clause w_σ such that one member of w_σ must be true. We have so far discussed the first case of H_t — that both M and M' assigns the same member of w_σ true.

We now discuss the other case — that M and M' assigns different variables of w_σ, a and a', true. We will have to test all possible choices of a and a', and for each choice we will assign all the other members of w_σ false and calculate their contribution, κ_i. Then, for both a and a', we will use the function Di as shown in Figure 8.8 and add $Di(a)$ and $Di(a')$ to κ_i. $Di(a)$ calculates the maximum number of transitively linked variables that can assume one value in a model where a is true and the other value in a model where a is false. The maximum κ_i found is compared to k_1 and the maximum is returned.

Before presenting $U(F_\sigma)$, we need one more definition: $Goals(a)$ for an entry point a is the union of the goals of the members of $dual(a)$. Now, $U(F_\sigma)$ is shown in Figure 8.9. Although F is an empty formula, it is assumed that from it, every variable assigned a value during the execution of Q can be reached.

Lemma 51. Using $U(F_\sigma)$, the variables removed by lines 1 and 2 of

```
1  Algorithm Fix(a)
2  k ← 0
3  for each x ∈ dual(a)  do
4     Let C = (ḃ ∨ ...) = goal(x)
5     if ḃ = true then
6        k ← k + H_t(C)
7     else
8        k ← k + H_f(C)
9  return k
```

Figure 8.5: *The auxiliary function Fix*

```
1  Algorithm H_f(C)
2  k ← 0
3  for each member a_j ∈ C  do
4     Let k ← k + Fix(a_j)
5  return k
```

Figure 8.6: *The auxiliary function* H_f

```
1  Algorithm H_t(C)
2  for each member a_j ∈ C  do
3     Assign a_j true and the other members of C false.
4     κ_i = Fix(a_1) + Fix(a_2) + ... + Fix(a_|C|)
5  k_1 ← max(κ_1, ..., κ_|C|
6  Try all choices of picking two members a′ and a″ from C = (a_2 ∨
   ...∨a_m). For each choice calculate κ_j ← Di(a)+Di(a′)+∑ di(a_i)
   such that a_i ∈ {a_1, ..., a_m} \ {a′ ∪ a″}.
7  k_2 ← max(κ_1, ..., κ_(|C| choose 2))
8  return max(k_1, k_2)
```

Figure 8.7: *The auxiliary function* H_t

```
1   Algorithm Di(a)
2   k ← 1
3   for each x ∈ dual(a)  do
4      Let C = goal(x)
5      for each c_j ∈ C  do
6         κ_j = Di(c_j)
7         k ← k + max(κ_1, ..., κ_|C|)
8   return k
```

Figure 8.8: *The auxiliary function Di*

```
1   Algorithm U(F_σ)
2   k ← 0
3   for each component T_i  do
4      Let a be the entry-point of T_i
5      if a is the first member of a clause y_σ then
6         k ← k + U_x(y_σ), where x ∈ {t, f} depending on the value
           of a.
7      else
8         for each member y_σ^j ∈ Goals(a)  do
9            k ← k + U_x(y_σ^j), where x ∈ {t, f} depending on the value
              of the first member of y_σ^j.
10  return k
```

Figure 8.9: *The auxiliary function U*

Q' will give the correct contribution to the overall maximum Hamming distance between any two x-models. Furthermore, $U(F_\sigma)$ runs in polynomial time.

Proof. Starting with the topmost function U, note that the entry-point a may appear in several clauses $(a \vee b)_\rho, (a \vee \bar{c})_\rho$ etc, and that this is correctly handled.

As for $H_f(C)$, note how Fix is used to examine the transitively linked variables to see if there are any possibilities for different assignments under the models that must have all members of C false.

When it comes to $H_f(C)$, note that in $Di(a)$, k is initialised to 1, because, as $Di(a)$ is the maximum number of variables that have one value when a is true and the other when a is false, then a itself must contribute 1.

Finally, as there are no cycles in any component, both Fix and Di run in polynomial time. Also, neither H_f nor H_t calls these functions more than a polynomial number. □

Theorem 52. $Q'(F)$ decides MAX HAMMING XSAT for F.

Proof. We inspect the lines of Q':

1. Correct by Lemma 51.

2. Correct by Lemma 51.

3. The formula $\{()\}$ is unsatisfiable and thus $\perp$ is returned.

4. Correct by Lemma 51.

5. If F is not connected every model for one component can be combined with any model for another component in order to form a model for F.

6. For this and the remaining lines: Assume there are two models M and M' at maximum Hamming distance k. If a_1 is true under both models then the formula where all other literals of w are set to false is x-satisfiable and the recursive call will return k (assuming that the algorithm is correct for smaller

```
1   if |W| ≠ 4 then ans_false = Q(F(a_1/false))
2   else
3     let W = (a_1 ∨ a_2 ∨ a_3 ∨ a_4) and assume that a_2 is a non-singleton
4     ans_f^1 ← Q(F(a_1/false; a_2/true));
5     ans_f^2 ← Q(F(a_1/false; a_2/false)
6     if ans_f^1 = ⊥ or ans_f^2 = ⊥ then ans_false ← max_⊥(ans_f^1, ans_f^2)
7     else
8       ans_f^3 ← Q(F(a_1/false; a_2/ā_3);
9       ans_f^4 ← Q(F(a_1/false; a_2/ā_4)
10      ans_false ← max_⊥(ans_f^1, ans_f^2, (ans_f^3 + 1), (ans_f^4 + 1))
```

Figure 8.10: *Some lines of code for speeding up Q'*

input.) Similarly for line 9. If both lines 8 and 9 returned an integer, we know that there are models under which a_1 is false and models under which a_1 is true. Thus M and M' may assign different values to a_1. Assume this is the case. By Lemma 46 we know that M and M' differ in exactly one more variable in w. Assume w.l.o.g. that a_2 is that literal. Then we know that a_1 and a_2 have different values and that the other literals of w are false.

□

8.2.3 Improving and Analysing the Running Time

As for the running time of Q', the handling of clauses of length 4 will cause an unnecessarily bad upper time bound. The problem is that in line 10 only one variable is removed. However, a clause of length 3 is created which can be exploited. Hence, we replace line 10 in Q' by the lines shown in Figure 8.10, thereby obtaining the algorithm Q. The correctness is easily seen, because it is the same kind of branching we have already justified.

Theorem 53. $Q(F)$ runs in polynomial space and time $\mathcal{O}(1.8348^n)$.

Proof. Let $T(n)$ be the running time for $Q(F)$. The analysis will proceed by examining what the running time would be if Q always encountered the same case. It is clear that the worst case will decide an overall upper time bound for Q. We inspect the lines of Q:
Line 1–4: All these lines are polynomial time computable.
Line 5: This line does not increase the running time as clearly, $\sum_{i=1}^{k} T(n_i) \leq T(n)$ when $n = \sum_{i=1}^{k} n_i$.
Lines 6–: It is clear that the worst clause length will decide an overall upper time bound for Q. Note that if there are variables left in F, then there will be at least two clauses left and one of the cases below must be applicable.

1. $|w| \geq 5$. Already a rough analysis suffices here: In the call $Q(F(a_1/true))$ a_1 as well as all the other variables in w get a fixed value and hence $|w|$ variables are removed. The next call only removes one variable, namely a_1. In every of the other $|w|-1$ calls $|w|-1$ variables are removed. Hence, the running time will be in $\mathcal{O}\left(\tau(|w|, 1, (|w|-1)^{|w|-1})^n\right)$ and the worst case is $\mathcal{O}\left(\tau(5, 1, 4^4)^n\right) \subseteq \mathcal{O}\left(1.7921^n\right)$.

2. $|w| = 4$. For a better readability, assume $w = (a \vee b \vee c \vee d)$. As a and b are not singletons there are clauses $a \in y$ and $b \in z$. There are several possibilities for y and z, but due to the balanced branching effect, we may disregard cases where $a \in w$ but $\bar{a} \in y$ etc.

   (a) $y = (a \vee e \vee f \vee g)$, $z = (b \vee h \vee i \vee j)$. The call $Q(F(a/true))$ removes 7 variables — all variables of w and y. The call $Q(F(a/false; b/true))$ removes 7 variables — all variables of w and z. The call $Q(F(a/false; b/false))$ removes 3 variables, because the clause $w = (c \vee d)$ will in the next recursive step be simplified. The call $Q(F(a/false; b/\bar{c}))$ removes 3 variables, because the clause $w = (c \vee \bar{c} \vee d)$ implies $d =$ false, which will be effectuated by the substitution operation. The call $Q(F(a/false; b/\bar{d}))$ removes 3 variables for the same reasons. The call $Q(F(a/\bar{b}))$ removes 3 variables — c and d must be false. Similarly for

the remaining two calls. Hence, the running time is in $\mathcal{O}\left(\tau(7^2, 3^6)^n\right) \subseteq \mathcal{O}\left(1.8348^n\right)$.

Remark 1 *If $|z| = 3$, then regardless of y we get cases better than the above case:*

(b) $z = (b \vee e \vee f)$. Counting removed variables as previously we get that this case runs in time $\mathcal{O}\left(\tau(6, 4^4, 3^3)^n\right) \subseteq \mathcal{O}\left(1.7605^n\right)$.

(c) $z = (a \vee b \vee e)$ or $z = (b \vee c \vee e)$ or $z = (b \vee d \vee e)$. All these cases run in time $\mathcal{O}\left(\tau(5^2, 4^6)^n\right) \subseteq \mathcal{O}\left(1.6393^n\right)$.

Remark 2 *If $|y| = 3$, then regardless of y we get cases better than the so far worst:*

(d) $y = (a \vee e \vee f)$. This case runs in time $\mathcal{O}\left(\tau(6, 5, 4^3, 3^3)^n\right) \subseteq \mathcal{O}\left(1.7888^n\right)$.

(e) $y = (a \vee b \vee e)$. Already examined.

(f) $y = (a \vee c \vee e)$ or $y = (a \vee d \vee e)$. These cases run in time $\mathcal{O}\left(\tau(5^2, 4^5, 3)^n\right) \subseteq \mathcal{O}\left(1.6749^n\right)$.

Remark 3 *If y shares more than one variable with w, then regardless of z we get cases better than the so far worst:*

(g) $y = (a \vee b \vee c \vee e)$ or $y = (a \vee c \vee d \vee e)$ or $y = (a \vee b \vee e \vee f)$. These cases run in time $\mathcal{O}\left(\tau(5^2, 4^6)^n\right) \subseteq \mathcal{O}\left(1.6393^n\right)$, $\mathcal{O}\left(\tau(6^2, 5, 4, 3^4)^n\right) \subseteq \mathcal{O}\left(1.7416^n\right)$ and $\mathcal{O}\left(\tau(5^4, 4^4)^n\right) \subseteq \mathcal{O}\left(1.5971^n\right)$ respectively.

(h) $y = (a \vee c \vee e \vee f)$ or $y = (a \vee d \vee e \vee f)$. These cases run in time $\mathcal{O}\left(\tau(6, 5^2, 4, 3^4)^n\right) \subseteq \mathcal{O}\left(1.7549^n\right)$.

Remark 4 *If z shares more than one variable with w, then regardless of y we get cases better than the so far worst:*

(i) $z = (a \vee b \vee \ldots)$. Already examined.

(j) $z = (b \vee c \vee d \vee e)$. This case runs in $\mathcal{O}\left(\tau(5^2, 4^6)^n\right) \subseteq \mathcal{O}\left(1.6393^n\right)$.

(k) $z = (b \vee c \vee e \vee f)$ or $z = (b \vee d \vee e \vee f)$. These cases run in time $\mathcal{O}\left(\tau(6^2, 5, 4, 3^4)^n\right) \subseteq \mathcal{O}\left(1.7416^n\right)$.

3. $|w| = 3$. We know that there is another clause y such that $|y| = 3$ and $a \in y$ and $y \neq w$. Hence we have a running time in $\mathcal{O}\left(\tau(4, 3, 2^2)^n\right) \subseteq \mathcal{O}\left(1.7107^n\right)$.

□

Chapter 9

Conclusions and Future Work

In this final chapter we will summarise and discuss the results in order to derive conclusions and point out possible directions for future research.

The three following sections treat *NP*-complete decision problems, counting problems and optimisation problems. In section 9.4 we discuss other members of our family not treated in this thesis. The last section gives some concluding remarks.

9.1 Decision Problems

For XSAT we have presented a new faster exact polynomial space algorithm. The algorithm builds on previous techniques but achieves a better upper time bound by a more thorough exploitation of the sparsity concept.

When it comes to the question of how to improve the algorithm even further, one can note that the canonical properties presented so far are not the only ones; we have found several others. These other properties look somewhat exotic and were not useful in the algorithms presented here, but they might find their future use. As an example:

There is no constant variable d with neighbours $c_1, c_2, \ldots, c_n$ such that there is a clause $x = (c_1 \vee c_2 \vee \cdots \vee c_n \vee \gamma)$ where each c_i has the same sign in the clauses of c and in x.

The following XSAT instance violates the rule and we see that, obviously, γ must have the same value as d:

$$(a \vee b \vee c \vee d \vee e)$$
$$(g \vee f \vee c \vee d \vee e)$$
$$(g \vee h \vee a \vee d \vee e)$$
$$(h \vee b \vee f \vee c \vee e \vee \gamma)$$

Historically, algorithms for XSAT have been improved by the invention of new canonical rules and clever new choices of variables to branch on. It is not a bold statement to suggest that a faster algorithm will be developed along these lines, too. The current upper time bound of $\mathcal{O}\left(\tau(1,12)^n\right)$ points out the need to find a means to further reduce formula complexity when a literal is set to false. Perhaps a new measure of formula complexity will be helpful, as was the case for the $\#2\text{SAT}_w$ algorithm.

XSAT can be seen as a restricted variant of SAT and one could hope that the increased knowledge for other restricted SAT problems would give some hints on how to improve the XSAT algorithms. For instance, the 3SAT problem is well studied. Unfortunately, although the two problems have properties in common — both being *NP*-complete Boolean decision problems — there is a fundamental difference in their structure. Due to the exactness property of XSAT, each assignment of a variable effectively constrains the allowed values of a number of other variables. This is reflected in the considerably lower upper time bounds for XSAT compared to 3SAT.

The currently best algorithm for SAT, by Dantsin et al. [21], achieves its time bound of $\mathcal{O}\left(2^{n-2\sqrt{n/\log n}}\right)$ by local search in 'small' neighbourhoods. To be more precise: A number of assignments are generated and a model is searched for in the Hamming ball of a certain radius around each assignment. For XSAT it is hard to see how the use of Hamming spheres can be used to improve algorithms as the

structure is very different. In terms of n, the running times are much better for XSAT algorithms.

Rephrasing XSAT into a more general decision problem, X_iSAT, we have shown its practical usefulness. When it comes to structural insights gained, we note that our DPLL style algorithm could still use matching techniques to find a solution when the instance contains no heavy variables. However, powerful canonical rules such as resolution are lost. In the future, one can hope that other canonical rules are found that can to some degree replace resolution.

Due to the low upper time bounds for XSAT ("low" for an *NP*-complete problem), no randomised algorithms have been proposed, to the best of our knowledge. However, such algorithms could be of interest for the general X_iSAT problem if deployed in incomplete model checkers. As an example of one simple algorithm: Choose an assignment to all variables at random. Then choose a clause x that has not been satisfied (here we count over-satisfied as not satisfied) and contains heavy variables such that these heavy variables can be used to satisfy the clause. Say for instance that x lacks two true literals, then at random choose two of the heavy variables that are false and assign them so that x is satisfied. Now consider all the heavy variables of x fixed and move on to another similar clause. This is performed until all heavy variables are instantiated and then the matching technique is deployed.

9.2 Counting Problems

Just as for the decision problems our two algorithms for #XSAT$_w$ and #X3SAT are DPLL style algorithms. The algorithm for #XSAT improves the previous result through careful case analysis and exploitation of possible neighbourhood configurations. Our algorithm for #X3SAT is built along the same lines and is the first one to this end. Unfortunately, when we move from deciding XSAT to counting XSAT, we lose the possibility to find a solution in polynomial time using the matching technique. Resolution also seems lost. This accounts for the worse upper time bounds for XSAT (in $\mathcal{O}(1.1730^n)$)

compared to #XSAT (in $\mathcal{O}(1.2190^n)$.)

One question that naturally arises is how to construct counting algorithms in a more systematic fashion. We note that many algorithms for *NP*-complete problems search the entire search tree with the exception of certain branches that can be easily pruned (i.e. if a formula contains a unit clause (p), then we do not need to consider branches where p is set to false.) Given such an algorithm, one could assume that the algorithm could be modified to also count the solutions along the way. Unfortunately, the modifications needed for keeping track of the number of solutions are not always obvious. This is illustrated by the introduction of extra structures in our algorithm for #XSAT$_w$; the pruning rule is quite obvious but the method for counting solutions is less so. Then there is also the seemingly insurmountable obstacle of canonical rules that do not extend to counting. Thus, it is not likely that a general pattern can be found that will take any algorithm for a *NP*-complete problem and transform it to an algorithm for the corresponding counting problem.

For #2SAT$_w$ we have presented two algorithms C and C_{sep}. C improves the running time and introduces several novelties. The algorithm C_{sep} exploits the possibilities of separation to count fast in 2SAT formulae. The algorithm for #3SAT$_w$ improves the running time and has an interesting time complexity analysis.

The method of analysis used in chapter 6, combined with the appropriate measures, provides a convenient way to capture and quantify the effects of properties that might otherwise be difficult to analyse, such as the decreasing average degree in C and the existence of 2-clauses in some branches in D. In algorithms with similar properties, it is likely that these measures can be reused.

When it comes to ways of even further improving algorithms for #2SAT and #3SAT, one alternative might be to perform a more careful analysis of the neighbourhood configurations, especially for #2SAT. Another option, if large memory usage is acceptable, might be to employ various dynamic programming or caching techniques, trading a decreased run-time for, possibly, an exponential memory usage.

The ideas in D might well be extended into an algorithm for #kSAT for any fix k, with a running time dependent on the parameter k but better than $\mathcal{O}(2^n)$. We have examined this issue briefly, and while the results seem promising, performing an analysis for the general case is challenging.

In the context of decision problems, adding weights seems to increase the complexity considerably. For instance, 2SAT is polynomial time solvable whereas the 2SAT_w problem is *NP*-hard (it contains the MAXIMUM INDEPENDENT SET problem.) It is thus interesting to note that our algorithms for #2SAT and #3SAT were easily extended into algorithms for #2SAT_w and #3SAT_w. To the best of our knowledge, there are no dedicated algorithms for solving the 2SAT_w and 3SAT_w problems in the literature.

The results using C_{sep} can be seen as an extension to the work by Díaz et al. [30]. There the authors present an algorithm for counting homomorphisms in graphs with a bounded tree width w. Among the applications they show how to count independent sets in linear time, while an algorithm for counting *maximum* independent sets would take exponential time using their approach. Here we show how our algorithm for counting in separable graphs, for the special case of graphs with fixed tree width, yields algorithms for counting maximum independent sets as well as independent sets, both with running times in $\mathcal{O}\left(n^{1.71w+1.71}\right)$.

9.3 Optimisation Problems

We have studied two very different optimisation problems. Chapter 7 provides some new results for MAX XSAT. On the positive side, a new upper time bound of $\mathcal{O}(1.2561^n)$ for RESTRICTED MAX XSAT was obtained. The restriction that no clause must be over-satisfied has proved powerful. The main result, however, is a negative one. By Theorem 43 it is clear that for any fixed k, a non-trivial upper time bound for solving MAX XkSAT implies the same time bound for MAX kSAT. Let us now compare the known results for exact algorithms for MAX XSAT and MAX SAT. Measuring in n we know that Williams'

algorithm [77] can be used to solve Max 2sat and Max x2sat in exponential space and time $\mathcal{O}(1.7314^n)$. Apart from that no non-trivial results are known. As for Restricted max xsat, it has no corresponding Sat problem. Turning to algorithms that measure their running times in $m(F)$ we have the following: Björklund and Husfeldt [8] have given an Max xsat algorithm running in $\mathcal{O}(4^m)$ time. This is to be compared with Max sat which can be solved in time $\mathcal{O}(1.3247^m)$ as shown by Chen and Kanj [11]. Max x2sat and Max 2sat can be solved in time $\mathcal{O}(1.1421^m)$ as shown by Kneis and Rossmanith [51].

There is something surprising about these results. For every other type of problem — deciding, counting and maximum Hamming distance — Xsat seems to quite easily allow the construction of faster algorithms than does Sat, at least when measuring in n. But in the context of maximising the number of satisfied clauses this does not hold. Let us also mention that we have tried without results to find an inverted version of Theorem 43, i.e. a theorem that says that: If we can solve Max sat in time $\mathcal{O}(c^n)$, then we can solve Max xsat within that same time.

Finally, we have presented two non-trivial, exact, polynomial space algorithms for Max hamming xsat and provided non-trivial upper bounds on their running time. Both algorithms point out new interesting research directions and indicate that problems such as Max hamming sat might be solvable in time better than $\mathcal{O}(4^n)$. Using P as a template when constructing an algorithm for a maximum Hamming distance problem, the goal is to analyse the instance at hand to see which calls to the external solver are superfluous. The algorithm Q indicates that it is possible to take direct advantage of the inherent structure of the problem itself; the algorithm can use two canonical rules, namely rule 4 and the weakened rule 6 that can be used when counting Xsat models.

9.4 Other Problems of the XSAT family

Of course there are other problems of the XSAT family that deserve interest, at least out of theoretical curiosity.

At the beginning of the 1980's a new complexity class, D^p, was introduced by Papadimitrious and Yannakakis [61]. It is the class of languages that are the intersection of a language in *NP* and a language in *coNP*, and it contains problems such as deciding the set of Boolean formulae that has one unique model. It also contains so-called *critical problems*, which ask if a structure lacks a property, but removing some part of the structure, the property holds. One such critical problem is critical satisfiability, SAT^{crit}. A formula F is in SAT^{crit} if F has no model, but the removal of any clause makes F satisfiable. This problem, which is also known as the *minimal unsatisfiability problem*, has many industrial applications (e.g. see [45].) Unfortunately, for the general problem, only the two obvious algorithms are known:

1. Test if the formula is unsatisfiable and then, for each clause, if the removal makes the formula satisfiable.

2. Keep a list of all the clauses. Cycle through all possible assignments and for every assignment that is a model except for one clause C mark C as visited. The formula is in SAT^{crit} iff no assignment was a model and every clause is marked visited.

The first approach yields an polynomial space and $O(2^{2n})$ time algorithm (the number of clauses may be 2^n), the latter gives an exponential space and $O(2^n)$ time algorithm.

SAT^{crit} is fixed parameter tractable as shown by Szeider [71] who proved an $O(2^k)$ running time, where $k = m(F) - n(F)$ (no formula with $n(F) \geq m(F)$ is unsatisfiable.)

The corresponding problem within the XSAT family would naturally be defined like this:

– CRITICAL EXACT SATISFIABILITY (XSAT^{crit}):

Instance: A formula F.

Question: Is it the case that F is not x-satisfiable, but removing any clause, the resulting formula is x-satisfiable?

Finding non-trivial algorithms for this problem would be of theoretical interest as well of practical importance. Perhaps we have a situation similar to the MAX HAMMING XSAT case, where the structure of the problem can be directly taken advantage of. It would also be interesting to know whether XSAT^{crit} is fixed parameter tractable.

While the XSAT^{crit} problem seems more complicated to solve than XSAT, the problem of deciding whether F has one unique XSAT model adds only a polynomial amount of work for any XSAT solver. The algorithm to show this is trivial: choose any variable a and test $F \wedge (a)$ and $F \wedge (\bar{a})$. If both return 'Yes' we know that there are at least two models. Otherwise, assume $F \wedge (\bar{a})$ has an x-model, next try $F \wedge (\bar{a}) \wedge (b)$ and $F \wedge (\bar{a}) \wedge (\bar{b})$ and so on, until all variables have been assigned a value and one x-model is found.

One can also imagine XSAT problems in the polynomial hierarchy worth studying. Consider for instance this quantified formula:

$$\forall \alpha, \beta : (a \vee b \vee \alpha), (a \vee b \vee \beta)$$

The question is: for every value of α and β, is there an x-model? For our example the answer is 'No', because $\alpha = true, \beta = false$ allows no x-model. Formulae similar to this one, in the context of SAT, has been studied for instance by Williams [76].

9.5 Final Remarks

When considering Table 9.1 it seems that there is a close connection between the running time and the canonical rules available. Though it is possible to claim that new strong rules will be invented so that there will be no differences in rules available to the various algorithms, it seems more reasonable to say the lack of rules is a part of the structural differences between different computational problems. Here, like in so many other parts in the area of computer science, we are waiting for results that separate various time complexity classes.

Problem	Time	#Can. Rules	Matching
XSAT	$\mathcal{O}(1.1730^n)$	12	Yes
X_2SAT	$\mathcal{O}(1.4511^n)$	10	Yes
X_3SAT	$\mathcal{O}(1.6214^n)$	2	Yes
X_4SAT	$\mathcal{O}(1.6848^n)$	2	Yes
#XSAT	$\mathcal{O}(1.2190^n)$	10	No
RESTRICTED MAX XSAT	$\mathcal{O}(1.2561^n)$	5	No
MAX HAMMING XSAT	$\mathcal{O}(1.8348^n)$	2	No
MAX XSAT	$\mathcal{O}(2^n)$	None	No

Table 9.1: *Summarising results*

Under the heading 'Final Remarks', one might also expect some personal reflections on insights gained from the doctoral work. If so, let us say that the remaining impression from working with canonical formulae is a sense of interconnectedness in these structures. Or, to put it differently:

No clause is an island.

Bibliography

[1] Lyudmil Aleksandrov and Hristo Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal on Discrete Mathematics*, 9(1):129–150, 1996.

[2] Noga Alon, Paul Seymour, and Robin Thomas. A separator theorem for graphs with an excluded minor and its applications. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC-1990)*, pages 293–299, 1990.

[3] Fadi Aloul, Arathi Ramani, Igor Markov, and Karem Sakallah. PBS: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability (SAT-2002)*, pages 346–353, 2002.

[4] Ola Angelsmark and Peter Jonsson. Improved algorithms for counting solutions in constraint satisfaction problems. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-2003)*, pages 81–95, 2003.

[5] Ola Angelsmark and Johan Thapper. Algorithms for the maximum Hamming distance problem. In Boi Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *Recent Advances in Constraints, Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming*, LNCS 3419, pages 128–141, 2004.

[6] Belaid Benhamou, Lakhdar Sais, and Pierre Siegel. Two proof procedures for a cardinality based language in propositional calculus. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS-1994)*, pages 71–82, 1994.

[7] Elazar Birnbaum and Eliezer Lozinskii. The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 10:457–477, 1999.

[8] Andreas Björklund and Thore Husfeldt. Polynomial space algorithms for exact satisfiability and chromatic number. http://www.cs.lu.se/~thore/papers/exact.pdf.

[9] Hans Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1–2):1–45, 1998.

[10] Jesper Makholm Byskov, Bolette Ammitzbøll Madsen, and Bjarke Skjernaa. New algorithms for exact satisfiability. *Theoretical Computer Science*, 332(1–3):515–541, 2005.

[11] Jianer Chen and Iyad Kanj. Improved exact algorithms for MAX-SAT. In *Proceedings of the 5th Latin American Theoretical Informatics (LATIN-2002)*, pages 341–355, 2002.

[12] Stephen Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing (STOC-1971)*, pages 151–158, 1971.

[13] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms.* The MIT press, Cambridge, Massachusetts, second edition, 2001.

[14] Pierluigi Crescenzi and Gianluca Rossi. On the Hamming distance of constraint satisfaction problems. *Theoretical Computer Science*, 288(1):85–100, 2002.

[15] Vilhelm Dahllöf. Applications of general exact satisfiability in propositional logic modelling. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-2004)*, pages 95–109, 2004.

[16] Vilhelm Dahllöf. Algorithms for max Hamming exact satisfiability. In *Proceedings of the 16th International Symposium on Algorithms and Computation (ISAAC-2005)*, pages 829–838, 2005.

[17] Vilhelm Dahllöf and Peter Jonsson. An algorithm for counting maximum weighted independent sets and its applications. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2002)*, pages 292–298, 2002.

[18] Vilhelm Dahllöf, Peter Jonsson, and Richard Beigel. Algorithms for four variants of the exact satisfiability problem. *Theoretical Computer Science*, 320(2–3):373–394, 2004.

[19] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting satisfying assignments in 2SAT and 3SAT. In *Proceedings of Computing and Combinatorics, 8th Annual International Conference, (COCOON-2002)*, pages 535–543, 2002.

[20] Vilhelm Dahllöf, Peter Jonsson, and Magnus Wahlström. Counting models for 2SAT and 3SAT formulae. *Theoretical Computer Science*, 332(1–3):265–291, 2005.

[21] Evgeny Dantsin, Edward Hirsch, and Alexander Wolpert. Algorithms for SAT based on search in Hamming balls. In *Proceedings of the 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS 2004)*, pages 141–151, 2004.

[22] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[23] Alvaro del Val. On 2-sat and renamable horn. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th*

Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI-2000), pages 279–284, 2000.

[24] Martin Luther D'Ooge, Frank Edleston Robbins, and Louis Charles Karpinski, editors. *Nicomachus of Gerasa: Introduction to Arithmetic*, volume 16. Macmillan, 1926.

[25] Michael Dransfield, Lengning Liu, Victor Marek, and Mirosław Truszczyński. Satisfiability and computing van der Waerden numbers. *The Electronic Journal of Combinatorics*, 11(1), 2004.

[26] Limor Drori and David Peleg. Faster exact solutions for some NP-hard problems. In *Proceedings of the 7th Annual European Symposium on Algorithms (ESA-1999)*, pages 450–461, 1999.

[27] Limor Drori and David Peleg. Faster solutions for exact hitting set and exact SAT. Technical Report MCS99-15, Belfer Institute of Mathematics and Computer Science, 1999.

[28] Limor Drori and David Peleg. Faster solutions for some NP-hard problems. *Theoretical Computer Science*, 287(2):473–499, 2002.

[29] Olivier Dubois. Counting the number of solutions for instances of satisfiability. *Theoretical Computer Science*, 81(1):49–64, 1991.

[30] Josep Díaz, Maria Serna, and Dimitrios Thilikos. Counting H-colorings of partial k-trees. *Theoretical Computer Science*, 281(1–2):291–309, 2002.

[31] Nils Olof Engfeldt. *Beiträge zur Kenntnis der Biochemie der Acetonkörper*. PhD thesis, Matematiska–naturvetenskapliga fakulteten vid Stockholms högskola, 1920.

[32] Leonhard Euler. The seven bridges of Königsberg. In James Newman, editor, *The World of Mathematics*, pages 573–580. Simon and Schuster, 1956.

[33] Sergey Fedin and Alexander Kulikov. A $2^{|E|/4}$-time algorithm for max-cut. *Zapiski nauchnyh seminarov POMI*, 293:129–238, 2002.

[34] Fedor Fomin, Fabrizio Grandoni, and Dieter Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. Technical Report 307, Department of informatics, University of Bergen, 2005.

[35] Martin Fürer and Shiva Kasiviswanathan. Algorithms for counting 2-sat solutions and colorings with applications. Technical Report 033, Electronic Colloquium on Computational Complexity, 2005.

[36] Harold Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-1990)*, pages 434–443, 1990.

[37] Haim Gaifman. On local and nonlocal properties. In *Proceedings of the Herbrand Symposium, Logic Colloquium '81*, pages 105–135, 1982.

[38] Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[39] William Gasarch. Guest column: The $P =?NP$ poll. *SIGACT News Complexity Theory Column*, 36, 2002.

[40] Leslie Ann Goldberg and Mark Jerrum. Counting unlabelled subtrees of a tree is #P-complete. *LMS Journal of Computation and Mathematics*, 3:117–124, 2000.

[41] Catherine Greenhill. The complexity of counting colourings and independent sets in sparse graphs and hypergraphs. *Computational Complexity*, 9(1):52–72, 2000.

[42] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan, and Toby Walsh. Finding diverse and similar solutions in constraint programming. In *Proceedings of the 20th International Conference on AI (AAAI-2005)*, pages 372–377, 2005.

[43] Edward Hirsch and Alexander Kulikov. An upper bound $\mathcal{O}\left(2^{0.16254n}\right)$ for exact 3-satisfiability: A simpler proof. *Zapiski nauchnyh seminarov POMI*, 293:118–128, 2002.

[44] Richard Hoche, editor. Νικομάχου Γερασηνοῦ Πυθαγορικοῦ ἀριθμητικῆς ἐισαγωγή. Teubner, Leipzig, 1866.

[45] Jinbo Huang. MUP: A minimal unsatisfiability prover. In *Proceedings of the 10th Asia and South Pacific Design Automation Conference (ASP-DAC-2005)*, pages 432–437, 2005.

[46] Harry Hunt III, Madhav Marathe, Venkatesh Radhakrishnan, and Richard Stearns. The complexity of planar counting problems. *SIAM Journal on Computing*, 27(4):1142–1167, 1998.

[47] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM*, 51(4):671 – 697, 2004.

[48] Richard Karp. Reducibility among combinatorial problems. In Raymond Miller and James Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[49] Sanjeev Khanna, Madhu Sudan, Luca Trevisan, and David Williamson. The approximability of constraint satisfaction problems. *SIAM Journal on Computing*, 30(6):1863–1920, 2001.

[50] Scott Kirkpatrick, Charles Gelatt, and Mario Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[51] Joachim Kneis and Peter Rossmanith. A new satisfiability algorithm with applications to Max-Cut. Technical Report AIB-2005-08, RWTH Aachen, 2005.

[52] Dexter Kozen. *The design and analysis of algorithms*. Springer-Verlag, 1992.

[53] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.

[54] Chu Min Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on on Innovative Applications of Artificial Intelligence (AAAI/IAAI-2000)*, pages 291–296, 2000.

[55] Richard Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36(2):177–189, 1979.

[56] Michael Littman, Toniann Pitassi, and Russell Impagliazzo. On the complexity of counting satisfying assignments. In the working notes of the LICS 2001 workshop on Satisfiability.

[57] Lengning Liu and Mirosław Truszczyński. Local-search techniques for propositional logic extended with cardinality constraints. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-2003)*, pages 495–509, 2003.

[58] Bolette Ammitzbøll Madsen. An algorithm for exact satisfiability analysed with the number of clauses as parameter. *Information Processing Letters*, 97(1):28–30, 2006.

[59] Bolette Ammitzbøll Madsen and Peter Rossmanith. Maximum exact satisfiability: NP-completeness proofs and exact algorithms. Technical Report RS-04-19, Basic research in Computer Science (BRICS), 2004.

[60] Burkhard Monien, Ewald Speckenmeyer, and Oliver Vornberger. Upper bounds for covering problems. *Methods of Operations Research*, 43:419–431, 1981.

[61] Christos Papadimitriou and Mihalis Yannakakis. The complexity of facets (and some facets of complexity). In *Proceedings of*

the 12th Annual ACM Symposium on the Theory of Computing (STOC-1982), pages 255–260, 1982.

[62] Stefan Porschen. On some weighted satisfiability and graph problems. In *Proceedings of the 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM-2005)*, pages 278–287, 2005.

[63] Stefan Porschen, Bert Randerath, and Ewald Speckenmeyer. X3SAT is decidable in time $O(2^{n/5})$. In *Proceedings of the 5th International Symposium on the Theory and Applications of SAT (SAT-2002)*, pages 231–235, 2002.

[64] Mike Robson. Finding a maximum independent set in time $O(2^{n/4})$. Technical Report 1251-01, LaBRI, Université Bordeaux I, 2001.

[65] Dan Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1–2):273–302, 1996.

[66] Herbert John Ryser. *Combinatorial Mathematics.* The Mathematical Association of America, Washington, 1963.

[67] Thomas Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on the Theory of Computing (STOC-1978)*, pages 216–226, 1978.

[68] Richard Schroeppel and Adi Shamir. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.

[69] Patrik Simons. *Extending and Implementing the Stable Model Semantics.* PhD thesis, Helsinki University of Technology, 2000.

[70] Edith Spaan, Leen Torenvliet, and Peter van Emde Boas. Nondeterminism, fairness and a fundamental analogy. *Bulletin of the EATCS*, 37:186–193, 1989.

[71] Stefan Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Electronic Colloquium on Computational Complexity*, 10(2), 2003.

[72] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of London Mathematical Society ser. 2*, 43:230–265, 1936.

[73] Salil Vadhan. The complexity of counting in sparse, regular, and planar graphs. *SIAM Journal on Computing*, 31(2):398–427, 2001.

[74] Leslie Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[75] Leslie Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[76] Ryan Williams. Algorithms for quantified Boolean formulas. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2002)*, pages 299–307, 2002.

[77] Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theoretical Computer Science*, 348(2-3):357–365, 2005.

[78] Mihalis Yannakakis. On the approximation of maximum satisfiability. In *Proceedings of the 3rd annual ACM-SIAM symposium on Discrete algorithms (SODA-1992)*, pages 1–9, 1992.

[79] Wenhui Zhang. Number of models and satisfiability of sets of clauses. *Theoretical Computer Science*, 155(1):277–288, 1996.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall:** Semantisk analys av processbeskrivningar i naturligt språk, 1977, ISBN 91-7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91-7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91-7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation,1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372-805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies,1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity

of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2.

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani**: Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms,1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund**: Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld**: Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén**: Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Langugaes from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis,1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN

91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi**: Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer**: Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-suppor-ted Inter-organisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationsystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing - An Emperical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004. ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linquistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modellling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005. ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tesanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with

Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informa-tionssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998. ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000. ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration - method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.