Algorithms, Measures and Upper Bounds for Satisfiability and Related Problems

by

Magnus Wahlström
Algorithms, Measures, and Upper Bounds for Satisfiability and Related Problems

Magnus Wahlström
Abstract

The topic of exact, exponential-time algorithms for NP-hard problems has received a lot of attention, particularly with the focus of producing algorithms with stronger theoretical guarantees, e.g. upper bounds on the running time on the form $O(c^n)$ for some $c$. Better methods of analysis may have an impact not only on these bounds, but on the nature of the algorithms as well.

The most classic method of analysis of the running time of dpll-style (“branching” or “backtracking”) recursive algorithms consists of counting the number of variables that the algorithm removes at every step. Notable improvements include Kullmann’s work on complexity measures, and Eppstein’s work on solving multivariate recurrences through quasiconvex analysis. Still, one limitation that remains in Eppstein’s framework is that it is difficult to introduce (non-trivial) restrictions on the applicability of a possible recursion.

We introduce two new kinds of complexity measures, representing two ways to add such restrictions on applicability to the analysis. In the first measure, the execution of the algorithm is viewed as moving between a finite set of states (such as the presence or absence of certain structures or properties), where the current state decides which branchings are applicable, and each branch of a branching contains information about the resultant state. In the second measure, it is instead the relative sizes of the modelled attributes (such as the average degree or other concepts of density) that controls the applicability of branchings.

We adapt both measures to Eppstein’s framework, and use these tools to provide algorithms with stronger bounds for a number of problems. The problems we treat are satisfiability for sparse formulae, exact 3-satisfiability, 3-hitting set, and counting models for 2- and 3-satisfiability formulae, and in every case the bound we prove is stronger than previously known bounds.
Acknowledgements

There are many people I want to thank, without whom I never would have made it through the graduation process. First of all, of course, there is my supervisor Peter Jonsson, who has taught me invaluable lessons about how our brand of science is performed and about the state of the field.

I want to thank Vilhelm Dahllöf, for cooperation on our common papers, as well as my other colleagues at TCSlab, for many interesting and entertaining discussions.

To Fedor Fomin, Alexey Stepanov, and the others at the University of Bergen, I thank you for the rewarding visit; I will keep in touch.

Finally, I am greatly thankful to my friends outside of the university, who have not let my work absorb literally all of my life.

This research work was funded in part by CUGS (the National Graduate School in Computer Science, Sweden).

Linköping, Sweden, March 2007

Magnus Wahlström
List of papers

Parts of this thesis are based on the following refereed papers:


Contents

I Introduction and General Topics  1

1 Introduction  3
  1.1 Approaches ........................................ 4
    1.1.1 Polynomial-time Cases ....................... 5
    1.1.2 Approximation Algorithms .................... 8
    1.1.3 Exact Algorithms ............................ 9
  1.2 Our Approach ........................................ 12
  1.3 The Problems ....................................... 16
    1.3.1 Satisfiability ................................. 16
    1.3.2 Exact Satisfiability .......................... 19
    1.3.3 Hitting Sets .................................... 20
    1.3.4 Counting Models for Satisfiability .......... 22
  1.4 Outline of the Thesis ................................. 24

2 Preliminaries  27
  2.1 Boolean Formulae ..................................... 27
  2.2 Graphs and Hypergraphs .............................. 30
  2.3 Problem Definitions ................................. 31
  2.4 Algorithm and Branching Concepts .................. 33

3 Measures of Complexity  37
  3.1 Introductory Example ............................... 37
  3.2 Non-classical Measures .............................. 42
  3.3 Standard Weight-based Measures .................... 47
  3.4 Finite Global States Modelling .................... 49
Part I

Introduction and General Topics
Chapter 1

Introduction

Suppose you have a problem. Suppose, to be more specific, that you have a problem that seems fit to be solved by a computer, and that after isolating the important features of the problem (translating a problem phrased in terms of, say, people and seating, or trucks, roads and cities, into a problem in terms of variables or graphs), you are left with a problem description on a binary domain: variables that may be assigned true or false, along with some restrictions that are to be fulfilled, possibly fulfilled in the best way; or a graph, or set of sets (hypergraph), where the goal is to find some set of vertices (or edges) under similar conditions (in which case we view “assigned true” as “included in the solution set”). How you go about this process—identifying the best way to view a problem, and translating it into appropriate terms of computer science—is not dealt with in this thesis; let us just assume that you have done so.

The topic of this thesis is the creation of algorithms for such problems. Moreover, the provided algorithms are exact and may require exponential time to finish—meaning, in the general case, that given a large problem they may chew on it for a very long time, but when they do finish, the answer they give will be guaranteed to be correct. Depending on your needs, and the nature of your problem, this may or may not be what you are looking for (well, you are probably not looking explicitly for an algorithm that will take a very long time,
1.1. Approaches

The problems in this thesis are divided into three categories: decision problems (where the question is whether any solution exists or not), optimisation problems (where the task is to find some best solution with respect to some measure, e.g. a cheapest solution), and counting problems (where the question is how many solutions a problem has). Satisfiability (or boolean satisfiability, abbreviated SAT) is quite general among the binary decision problems: an instance, called a formula, consists of a set of restrictions called clauses, where each clause eliminates exactly one set of assignments (e.g. a clause \((v_1 \lor v_2 \lor v_3)\), interpreted as “\(v_1\) is true or \(v_2\) is true or \(v_3\) is true”, eliminates all assignments where \(v_1 = v_2 = v_3 = 0\), and \((\overline{v}_1)\), where \(\overline{v}_1\) is a negated occurrence of \(v_1\), eliminates all assignments where \(v_1 = 1\). A formula is satisfiable if there is an assignment that satisfies all clauses (i.e. is not eliminated by any clause). Literally every restriction on a set of boolean variables can be implemented in this way, though for some restrictions the number of clauses required is very high (e.g. the condition that at most \(i\) out of \(k\) variables may be false requires \(\binom{k}{i+1}\) clauses in a straight-forward encoding). An example of an optimisation problem could be to try to satisfy a formula while setting as few variables to true as possible. This also covers a number of more common problems as special cases, for instance independent set, where we want to pick as many vertices as possible in a graph without selecting both end points of any edge. For a counting problem, of course counting the number of solutions to a SAT formula works (written as the problem \(#SAT\); the counting versions of problems are often written as \(#\) prefixed to the problem name), but again more specific cases are more common. For instance, in linear algebra there is a prob-
1. Introduction

Problem known as computing the permanent of a matrix, and for matrices where all entries are 0 or 1 the problem is equivalent to counting the number of perfect matchings to a bipartite graph. A matching is a set of edges of a graph that have no common endpoints, and a perfect matching in a graph with \( n \) vertices is a matching of \( n/2 \) edges—as such a matching will contain all \( n \) different vertices as endpoints, no larger matching is possible. Counting perfect matchings, in turn, is expressible as a \#sat problem.

Note that we do not deal with problems with non-binary domains in this thesis. Such problems are probably better modelled by using constraint satisfaction problems \[73\] than by sat.

Either way, once you have an appropriate problem encoding, what are your options for solving it? We will examine this question from a number of perspectives over the following pages.

1.1.1 Polynomial-time Cases

If you are very lucky, then it may turn out that the problem is equivalent to one of the problems for which there exist efficient and exact algorithms (algorithms that finish in time \( O(p(n)) \) for some polynomial \( p(n) \) and \( n \) variables; we say that the problem is in the class P). There are indeed some cases where this can be done when it is not obvious that this is the case, of which one of the more famous is the problem of maximum matching. In this problem, the instance is a graph, and the goal is to find a matching of as many edges as possible (as mentioned, a matching is a set of edges where no two edges intersect). A polynomial-time algorithm has been known since 1965, when Edmonds \[26\] constructed one. In general, however, it seems that most interesting problems are unlikely to have such algorithms—for instance, the problem of counting the number of perfect matchings, previously mentioned, is likely difficult even though finding a matching is easy. Our strongest reasons for believing that these problems are indeed impossible to solve in polynomial time (and that it is not just a lack of imagination or insight on our part that has caused us to fail to find any way to do so) involves the concept of complexity classes, from the field of computational complexity (for a more
technical and precise treatment of these matters, see e.g. Kozen’s book [55]).

As we mentioned, P is the class of all problems for which an exact algorithm exists that can solve a problem in polynomial time (i.e. in an amount of time that grows polynomially with the instance size). The class NP contains all problems for which we can recognise the solution to a problem in polynomial time. It is easy to see that the class NP contains many interesting problems that we would wish to be in P (including, for instance, many of the problems mentioned so far). However, a surprisingly large majority of naturally occurring problems that fit the NP categorisation (in fact, a large majority of all “natural” problems in NP, whether the word “natural” implies that a problem occurs naturally or that it is judged to be mathematically interesting) are either already known to be polynomial, or are NP-complete: if any one of them has a polynomial-time algorithm, then this algorithm can be used to solve every other problem in NP as well in polynomial time. By consequence, such an algorithm would provide us with a polynomial-time solution for any problem for which we can write a program that takes as input a problem instance on n variables and a proposed solution, and verifies (in time $O(p(n))$) whether the proposed solution works for the problem instance. (With a bit of poetic license, this has been described as “automating creativity”.) This possibility is referred to as P=NP (as it would imply that the classes contain exactly the same problems), and while deciding whether this is the case is an important and famous open problem, P≠NP is overwhelmingly seen as the likely outcome; see the poll by Gasarch [42]. In this case, as mentioned, a large number of interesting problems will never have efficient and exact algorithms.

As a sidenote, it is common in theoretical computer science to use “efficient algorithm” as a synonym for “polynomial time algorithm”. While it is possible for an algorithm to finish in polynomial time without being efficiently usable from a real-world perspective (the definition of polynomial time includes both $O(n^{100})$ and $O(n)$ where the constant is $10^{1000}$), it fortunately seems common that polynomial-time algorithms which give an exact solution to a natural problem
also have reasonable behaviour, or can be reimplemented with reasonable behaviour after some further research. The caveat about exact solutions is mainly due to the area of approximation algorithms (see below). Another reason for this consensus is that the concept of polynomial-time versus non-polynomial time algorithms\(^1\) is an easy one to work with, and a reasonable theoretical concept.

So far, we have covered the possibility of an efficient and exact algorithm, that will give a solution (or a best solution) for every instance of the target problem. Another possibility is that the particular problem in question has some extra property that makes efficient solutions possible. For instance, a horn clause is a clause where at most one literal is positive (e.g. \((\overline{v}_1 \lor \overline{v}_2 \lor \ldots \lor \overline{v}_{k-1} \lor v_k))\), and if one literal is positive, then such a clause is equivalent to an implication containing no negations (the given example is equivalent to \(v_1 \land \ldots \land v_{k-1} \rightarrow v_k\)). If every clause in a formula is a horn clause, then there exist polynomial-time methods to decide whether the formula has any solution. This fact is central to the field of logic programming [2, 65] (though not immediately important to the subject of this thesis). Another example (which is important to the subject of this thesis) is when the shortest clause of a formula has at least as many members as the maximum number of occurrences of any variable (for instance, when a formula contains no clauses with only one literal, and no variable occurs more than twice). In such a case (or indeed in any case where a set of \(k\) clauses always includes at least \(k\) different variables), a solution always exists, and can be found through an application of an algorithm for maximum matching [80]. Still, such cases are not necessarily any more common than when the problem is polynomial to begin with. The consensus seems to be that cases when we can get any kind of a guarantee of a polynomial-time exact algorithm for interesting problems are rare.

\(^1\)Note that NP is not a synonym for non-polynomial.
1.1.2 Approximation Algorithms

For optimisation problems, another path to polynomial time remains open. If we relax our condition that we want the absolutely best solution to say that we will settle for some decent approximation, then we can for some problems and levels of approximation find polynomial-time algorithms that achieve this. For instance, consider the problem of vertex cover. The instance is a graph, and the task is to find a set of as few vertices as possible that includes at least one endpoint of every edge. This problem can be approximated to within a factor of 2 [54], meaning that there exists a polynomial-time algorithm that finds a vertex cover for a graph that is at most twice as big as the smallest possible vertex cover. However, if the conjecture referred to as the unique games conjecture [53] is true, then we cannot give a better guarantee than this unless P=NP, and even if the unique games conjecture is false, our possibilities are still limited: if P\(\neq\)NP, then we can never approximate within a factor of 1.36 or better [22]. However, the problem of independent set (which we previously described) cannot be approximated to within \(n^{1-\epsilon}\) on \(n\) variables for any \(\epsilon > 0\), again unless P=NP [88]. Of course, when the maximum possible solution is \(n\), knowing that we can approximate to within a factor of \(n\) is not very useful.

For other problems, there are polynomial-time approximation schemes (PTAS), where for every \(\epsilon > 0\) there exists a polynomial-time algorithm that guarantees that the returned solution is at most \(1 + \epsilon\) times as big as the smallest possible solution (assuming that the problem is a minimisation problem), but at the cost that the amount of time the algorithm requires depends on the \(\epsilon\) that is chosen. However, even then there are drawbacks. Besides the obvious point that many problems do not have a PTAS unless P=NP, the dependence of the running time on the parameter \(\epsilon\) for those problems that to have a PTAS can be quite bad. Independent set does have a PTAS when the instances are restricted to unit disk graphs (essentially, the instances can be viewed as a number of overlapping coins lying on a table, with the problem of finding a maximum number of coins that do not touch each other, even though they may touch other coins that
were not chosen), but the best known time is $O\left(n^{1/\epsilon}\right)$ [50], meaning we get $n^{10}$ for the largest error of ten percent, $n^{100}$ for a largest error or one per cent, and so on. Also, it is unlikely that the degree of the polynomial will ever be free of $\epsilon$ (if some PTAS with a running time of $O\left(f(\epsilon) \cdot p(n)\right)$ exists for any $f$ and a polynomial $p(n)$, then FPT=W[1] [60]; while not as universally disbelieved as P=NP, this is still believed to be false). Note that the behaviour of $O\left(n^{1/\epsilon}\right)$ is not extraordinary; Downey [23] lists a number of problems for which the then-best PTASs would require running times ranging from $O\left(n^{15000}\right)$ to $O\left(n^{10^{60}}\right)$ for a relative error of twenty per cent.

1.1.3 Exact Algorithms

If we have eliminated all of these options, then there seem to be essentially two remaining options: to apply an algorithm that always finishes quickly and hopefully gives good results, though no guarantees can be given for the quality of its solutions—a heuristic—or to apply an exact algorithm that needs super-polynomial time to finish, in the hope that it will still finish in reasonable time for the instances we need to solve. Reasonable success has been reported for both approaches. We will not focus on the topic of heuristics here, except to say that there may be situations when we do want some kind of a guarantee—in particular, if we have a decision problem, then the information that no solution exists may be more valuable than the information that our particular heuristic was unable to identify a solution in the time given. For instance, our problem could be one of verifying certain properties of a proposed hardware design, in which case a solution to our formula may mean the presence of a fault. In this area, exact algorithms for SAT are frequently used, and do indeed generally finish in reasonable time, even after having verified that no solutions to the formula existed [5, 61, 63, 77]. Another aspect is that research on improved upper bounds can lead to or inspire improved heuristics as well, as there will usually be some improvements in the algorithm to which the improvement in the bound can be traced.

This, then, provides one (long-winded) explanation of why the
topic of exact algorithms for NP-complete problems is an interesting one to study. (In very brief summary, because NP-complete problems capture significant properties of relevant problems, and because attacking them with exact algorithms is sometimes our best choice, and will often work in practice.) Other reasons have more to do with the conditions for performing useful theory work (and the elusive concept of mathematical intuition): while heuristics and “things that seem to work” may be useful in industrial situations, when we are performing research on theoretical computer science we would rather focus on something that is more concrete than this, and easier to formalise; not because doing so is easier, but because it is expected to bear more fruit. At least, that is this author’s interpretation. It is also noted that the topic has received an increasing amount of attention in later years. For other people’s views on these reasons, see for instance the survey from 2003 by Woeginger [85]. Consider also the survey by Fomin, Grandoni, and Kratsch [37] for some aspects of the field that are omitted in Woeginger’s paper, and Schöning’s briefer general introduction to the field of exponential-time algorithms [75].

Note that an exact, exponential-time algorithm is not the same thing as an algorithm that solves a problem by exhaustive search (or, depending on how the terms are used, that an exhaustive search does not need to visit every potential solution to a problem). As mentioned, the problems in this thesis are all on a binary domain, meaning that such an exhaustive search (i.e. cycling through all possible assignments and see if one of them works) would require a time of $\theta^*(2^n)$, while the upper bounds given in this thesis range from $O^*(1.0984^n)$ for X3SAT to $O^*(1.6671^n)$ for #3SAT (see Section 1.3). Comparing $2^n$ to 1.0984, the latter behaviour allows more than seven times as large instances before the same time consumption is reached. By contrast, the exponential growth of computer power (while this trend

\footnotesize
\[ \text{The notation } O^*(\cdot), \theta^*(\cdot), \text{ etc, means that polynomial factors have been ignored.} \]
\[ \text{If our bounds are } \theta^*(2^n) \text{ to } O^*(1.0984^n), \text{ then there are also the questions of how large the polynomial and constant factors that have been ignored are, but as long as they both are reasonable, the exponential parts will dominate the comparison.} \]
1. Introduction

continues) increases the feasible instance size by a constant amount for every time the power is doubled (in the example of \( \theta(2^n) \), even if Moore’s law will continue to hold, it would predict that we can add one single variable to the instance size every eighteen months). As we see, improvements in the exponential behaviour can translate into significant improvements in the feasible instance sizes.

However, there is a secondary effect as well: all the upper bounds that we have are imprecise, and it seems that the actual behaviour is rarely (if ever) as bad as the predicted upper bound (the exception might be algorithms that actually do explicitly enumerate over a set of assignments of known size); we have a kind of “theory gap” in that the empirical results are far better than the theoretical guarantees. This gap can be divided into two parts: the tendency for so-called real world instances to be easier than the worst possible cases (a gap between the behaviour of real-world cases and the actual worst-case behaviour), and non-tightness in our upper bounds (a gap between actual worst-case behaviour and the upper bounds).

The first part of the gap has seen some examination. One way related to it is to identify some property or parameter of an instance that limits the exponential behaviour, for instance the treewidth and similar decomposition properties [46]. Instances with low treewidth have a sort of locality property, where vertices in “one end” of the instance do not occur together with vertices in the “other end”. In general, the study of algorithms where the super-polynomial behaviour is confined to a parameter (e.g. times on the form \( O(p(n) \cdot f(k)) \) for a parameter \( k \), where \( p(n) \) is polynomial and \( f(k) \) is not) occurs in the field of parameterised complexity [35]. While these improved bounds are only guaranteed to hold if dedicated algorithms are used, these parameters can still be seen as ways in which an instance can be easy, hopefully in ways that influence the behaviour of the “popular” (i.e. commonly used) algorithms as well.

The second part, that of how good our bounds are, seems to have been less studied, but there do exist cases where algorithms have been re-analysed and given better bounds, with either no improvements or very simple improvements to the actual algorithm. The algorithm for
1.2. Our Approach

###2SAT given in Chapter 7 is one such example, where the original publication gave a bound of \( O^*(1.2561^n) \) [15], which has later been improved to \( O^*(1.2461^n) \) [40] and is now given as \( O^*(1.2377^n) \) (see below). The problem of dominating set provides a more dramatic example: an algorithm by Grandoni which was originally given a bound of \( O^*(1.8021^n) \) [45] has a currently best bound of \( O^*(1.5137^n) \), by Fomin, Grandoni, and Kratsch [36]. It is possible that better methods of analysis may improve bounds for other algorithms as well.

A secondary effect of the method of analysis is that sometimes, even though the worst-case bound is not affected, a weaker analysis may require the algorithm to be very complicated in order to guarantee the bound, while in a stronger analysis we may be able to prove that the same bound holds for a more natural phrasing of the algorithm. For an example, admittedly imperfect, let us compare two algorithms for independent set: the currently best bound is for an algorithm found in a technical report by Robson [72], which has a running time in \( O^*(1.2025^n) \) if polynomial space is used but consists of a list of cases and subcases that requires several pages to be described, while another algorithm, again by Fomin, Grandoni, and Kratsch [38], that has a running time in \( O^*(1.2210^n) \) can be stated in ten lines of pseudocode.

###1.2 Our Approach

Different strategies exist for solving NP-hard problems in better than the trivial time. Among them, we can name uses of dynamic programming (which often achieves an exponential speedup at the cost of having to remember an exponential amount of partial results), variations on local search in an exact or probabilistic setting, and other uses of randomisation and probabilistic algorithms. Again, we refer to the surveys of Woeginger [85], Fomin et al. [37], and Schöning [75] for an overview. However, the approach that is used in this thesis is that variously known as dpll (Davis-Putnam-Logemann-Loveland, from an early paper where the method is used [19]), branching (or branch-and-bound), or backtracking. We will describe it in better de-
1. Introduction

tail in following chapters, but essentially it is a recursive search for a solution: when working with a boolean domain, you pick one variable, and make recursive calls in turn to find a solution where this variable is set to true or false (called branching on the variable). Since both instances for which the recursive calls are made have fewer unassigned variables, some progress is made, and eventually the process terminates (either because a solution has been found, or because the current line of search has left an unsatisfiable constraint, in which case the search backtracks to another path). This can then be strengthened in various ways, by giving better ways to branch, or by giving simplifications (known as reductions) that can be applied to the algorithm before the branching is performed.

Branching algorithms seem in a way to provide the backbone of exact algorithms for NP-hard problems. Some kind of branching algorithm is practically always possible, if only the trivial $O^*(2^n)$ behaviour, and very often it is possible to make observations that improve this. They also lend themselves immediately to simple analysis of upper bounds on the behaviour. This, too, will be expanded on in later chapters, but let us just say that by counting the number of variables that are removed in every step, we can find an upper bound; for instance, if an algorithm creates two subproblems when branching and both problems contain two fewer variables, then the running time will be in $O^*(2^{n/2}) \subset O^*(1.4143^n)$ (since there will be zero variables left after exactly $n/2$ steps), and if one subproblem contains one variable less while the other contains three variables less, then the bound $O^*(1.4656^n)$ can be found through standard methods [43].

Sometimes, this kind of analysis (counting variables) is all that is performed, but it is not all that can be done. More advanced variants are possible with the concept of complexity measures. A complexity measure is a numerical measure of the judged difficulty of an instance—$n$ is a complexity measure, albeit a very simple one—which can be used in the analysis of the running time of an algorithm in much the same way as $n$ was used in the previous example; instead of counting the number of removed variables in a branch from an instance $F$ to a subinstance $F'$, with a complexity measure of $f(F)$ we
1.2. Our Approach

will use $\Delta f = f(F) - f(F')$ to measure the amount of progress that has been made from $F$ to $F'$ (i.e. by how much the judged difficulty of the instance has decreased). If $\Delta f \geq 2$ in every step of the algorithm, then the bound $O^* \left( 1.4143^{f(F)} \right)$ on the running time will be valid [56] (and if $f(F) \leq n$, then $O^* \left( 1.4143^n \right)$ will be valid as well). Kullmann did early work that put a focus on this concept [56–58]; much of our terminology is taken from his work. The advantage of using more advanced complexity measures is that we can pick any quantifiable and representative measure of difficulty we need (including measures that involve several parameters, one of which may be $n$), according to what best characterises the behaviour of a particular algorithm or problem. The corresponding disadvantage of only using $n$ would be that it may be a non-representative, slightly artificial way of measuring the difficulty, which can lead to lower-quality upper bounds, and also to algorithms that are themselves somewhat artificial in their construction; more on this is said below.

As mentioned, we may in an analysis using complexity measures consider several aspects at once; a common extension is to consider the number of variables of each degree, or to consider the number of clauses or edges in addition to $n$. In this way, we can include more information about the behaviour of the algorithm in the analysis; if we for instance use the number of variables of each degree, then changes in variable degrees can be used in the analysis as well. Having modelled the possible branchings of the algorithm in terms of how each branching will change the values of the considered attributes, we can then (thanks to the work of Eppstein on quasiconvex analysis [31]) automatically produce a complexity measure $f(F)$ from this data. The measure will be a weighted sum of the considered attributes, and the resulting upper bound $O^* \left( c^{f(F)} \right)$ will be tight with respect to the analysis (meaning that the bound is the actual asymptotic growth of the model, while the model may itself be too pessimistic, due to information that was not considered in the analysis).

One thing to note is that the analysis requires every branching that the algorithm uses—every step of the algorithm—to make good “progress”, as measured by our complexity measure, in order for a
good bound to be possible. If our analysis is performed solely in terms of $n$, then as a consequence we have to design an algorithm that removes a high number of variables in every step. Doing so may, as previously implied, require treating a large number of special cases, with specific instructions for how to treat each case. With a multi-parameter complexity measure, any change in any of the parameters may constitute progress; for instance, a change in the degree of a non-removed variable, as mentioned above. There seems to be a tendency that when we consider a larger number of such effects, cases that previously seemed to require special treatment turn out to not be so special after all, if a small number of removed variables is accompanied by stronger changes in other aspects. (Another approach for performing an analysis where not every branching needs to be strong is used by Kullmann for 3SAT [56], where, by considering the whole execution of the algorithm, he can compensate for some weak branchings under the condition that enough strong branchings are used.)

The contributions to the process of analysis made in this thesis are two new variants of complexity measures that are introduced in Chapter 3. The first, referred to as analysis by finite global states, can be seen as an attempt to soften up the condition of “progress in every step” mentioned above, by introducing an explicit notion of state to the analysis. This state will encode some global property of the instance, particularly the presence or absence of certain structures or properties; in the usage in the thesis, the encoded property is the number of short clauses in a formula, but this is only one example. If this property is correctly chosen, then certain states (the presence of short clauses) will admit better branchings while other states (the absence of short clauses) will be limited to poorer branchings but will result in the state being changed to something better. Through an automatic step of analysis, these effects are then evened out, so that the bound $O^*(c^n)$ that is achieved lies somewhere between the bounds for the best and the worst branchings.

The second measure, referred to as a compound measure, is similar in that it deals with a state that controls the behaviour of the
applicable branchings, but the meaning of the state is quite different. Where in the previous method the point is to analyse the effects of the algorithm moving back and forth between the states, here the state will be a function of the relative values of the attributes, and the important effect which is modelled is its gradual change. For example, with attributes \( n(F) \) and \( l(F) \), where \( l(F) \) represents the total length of the formula (i.e. the total number of occurrences for all variables), the value of \( l(F)/n(F) \), representing the average number of occurrences for a variable in the formula, could determine the state. The analysis will study the effect of this state on the behaviour of the algorithm: with a low value of \( l(F)/n(F) \) we could have a low total running time, even though a small number of variables is removed (because polynomial cases would apply when \( l(F)/n(F) \) is low enough, and the algorithm may pick a strategy designed to reduce \( l(F)/n(F) \) as quickly as possible), while a higher value of \( l(F)/n(F) \) could imply that we get branchings immediately removing a larger number of variables. Other “density-like” states are also possible, such as the relative number of variables with a certain property, as is compound measures using several attributes (e.g. replacing \( n(F) \) by a number of attributes \( n_i(F) \) representing the number of variables of each degree). The final step of analysis is again automatic, producing better bounds than what is possible in an analysis without considering such a state.

1.3 The Problems

Now, we will present the problems treated in this thesis, and give some background on previous work on them.

1.3.1 Satisfiability

The boolean satisfiability problem, and its restricted variants, is one of the most well studied classes of NP-complete problems. The problem instance is a boolean formula on conjunctive normal form (CNF), which means that the formula is a conjunction of constraints known as *clauses*, where each clause is on a disjunctive form (i.e. \((x \lor y \lor z)\)
or \((\bar{a} \lor b \lor \bar{c} \lor d)\), and the question is whether there exists some assignment to the variables that satisfies every constraint (a more technical definition is given in the next section).

For the general case (that is, for general formulae on CNF), no algorithm that solves the problem in \(O^*(c^n)\) time with \(c < 2\) for \(n\) variables is known, and is sometimes believed not to exist, though algorithms that beat \(O(2^n)\) do exist. The strongest current result, for \(n\) variables and \(m\) clauses, is \(O(2^n(1-1/\alpha))\) where \(\alpha = \log(m/n) + O(\log \log m)\) from a paper by Dantsin, Hirsch, and Wolpert [17] (which is a deterministic result; the same expected bound is achieved by a probabilistic algorithm in an older result of Dantsin and Wolpert [18]). See Dantsin, Hirsch, and Wolpert [17] for an overview of earlier related results.

Better studied, however, are the various restricted variants for which a bound of \(O^*(c^n)\) with \(c < 2\) are possible, and the most notable of these restricted problems are \(k\)-SAT where each clause may have at most \(k\) literals. For \(k = 2\) this is polynomial [41], and for \(k > 2\) it is NP-complete [41]. For \(k = 3\) the best results are a probabilistic algorithm which runs in time \(O^*(1.3238^n)\) [51] and a deterministic one which runs in time \(O^*(1.473^n)\) [8], while for general \(k\)-SAT there is a probabilistic algorithm with running time in \(O^*((2 - 2/k)^n)\) [74] (and a somewhat stronger bound by Paturi, Pudlák, Saks and Zane which is difficult to state succinctly [66, 67]) and a deterministic algorithm running in time \(O^*((2 - 2/(k + 1))^n)\) [16]. \(3\)SAT in particular has a long history of improvements.

Another type of restriction, which is less well-studied, is when every variable is limited to at most \(d\) occurrences in a formula. This restriction is most closely related to the one considered in this thesis (we consider the more general variant where the average number of occurrences per variable is at most \(d\)). When \(d = 2\), this is solvable in linear time, while for \(d \geq 3\) it is in the general case NP-complete [80]. If there is also the restriction that each clause has exactly (or at least) \(k\) literals, then the problem is trivial (a solution always exists) if \(d \leq k\) and NP-complete otherwise [80]. (Note that these complexity results hold for when the restriction is on the maximum degree, not
necessarily when the restriction is on the average degree.) Previous results for this kind of restriction include $O^*(3^{n/9}) \subset O^*(1.1299^n)$ for the case with at most 3 occurrences per variable by Kullmann [57,58], and algorithms by Hirsch [49] that run in time $O^*(1.2389^m)$ and $O^*(1.0740^l)$ (where $m$ is the number of clauses of a formula, and $l$ is the total length, i.e. the total number of occurrences for all variables), which (since $l \leq dn$) results in bounds of $O^*(1.3305^n)$ for $d = 4$, $O^*(1.4290^n)$ for $d = 5$, and $O^*(1.5348^n)$ for $d = 6$. Also, Szeider has given an algorithm whose time is bounded in terms of a parameter known as the maximum deficiency of a formula [78]: if $m(F)$ is the number of clauses and $n(F)$ the number of variables of $F$, then the maximum deficiency is $D = \max_{F' \subseteq F} (m(F') - n(F'))$ and the algorithm runs in time $O^*(2^D)$. When there exists a lower bound on the clause length of $k$, we have $m(F) - n(F) \leq (dn(F)/k) - n(F) = (d/k - 1)n(F)$, which guarantees a polynomial algorithm when $d \leq k$ and provides a bound of $O^*(2^{n/3}) \subset O^*(1.2600^n)$ when $k = 3$ and $d = 4$. However, when 2-clauses are allowed, the result is not as strong.

The results in this thesis are a reworking of the contents of two conference papers [83, 84]. For $d \leq 4$, we have bounds of $O^*(1.1279^n)$ for $d = 3$ and $O^*(1.2721^n)$ for $d = 4$, when $d$ is the maximum number of occurrences, or $O^*(1.1279^{l-2n+s})$, where $s$ is the number of single-occurring variables, when $d$ is the average number of occurrences (note that adding a single-occurring variable increases both $l$, $n$, and $s$ by one each, so that the net difference is 0). For $d \geq 5$, we achieve the same bounds for when $d$ is the maximum number of occurrences as for when $d$ is the average number of occurrences, these bounds being $O^*(1.3783^n)$ for $d = 5$, $O^*(1.4548^n)$ for $d = 6$, and for general $d$ the bound approaches $2^n$ at a rate of $O^*\left(2^{(1-c/(d+1)+O(1/k^2))n}\right)$ for some constant $c$. A bound in terms of $l(F)$ of $O^*(1.0663^{l(F)})$ for any formula $F$ is also achieved.
1.3.2 Exact Satisfiability

Exact Satisfiability (Xsat) is the problem of deciding whether, given a boolean formula on CNF, there exists an assignment to all variables such that every clause is satisfied by exactly one member. Exact 3-Satisfiability (X3sat) is the restriction of this problem to instances with maximum clause length 3, and is also known under the name 1-in-3 SAT. When no negations are present in the formula, it is known as exact cover or exact hitting set (as it becomes the exact version of the Hitting Set problem described below).

Because of the much stronger structure imposed by such constraints, Xsat and X3sat can be solved much more efficiently than the sat and 3sat problems: the best results so far are $O^*(1.1730^n)$ for Xsat by Dahllöf [13], improving on the $O^*(1.1739^n)$ result of Byskov et al. [9], and $O^*(1.1003^n)$ for X3sat by Byskov et al. [9]. In terms of the number of clauses $m$, the bound $O^*(2^m)$ was recently achieved by Björklund and Husfeldt [6].

Dahllöf’s thesis [13] contains results for a large number of variations on this problem, such as counting the number of solutions (#Xsat), deciding whether there is a solution that satisfies exactly $i > 1$ members of each clause (XiSAT), or finding a solution that satisfies as many clauses as possible (MAX XSAT). Guruswami and Trevisan [48] consider the approximation properties of MAX XSAT and variations (e.g. whether you can find, in polynomial time, an assignment that satisfies at least half as many clauses as an optimal assignment). The result varies with the variant of definition, but in general for every problem variant there is some factor $c$ such that you cannot approximate within better than $c$ (i.e. you cannot guarantee satisfying more than $1/c$ of the optimum number of clauses in polynomial time).

We provide an algorithm for the X3SAT problem that runs in $O^*(1.0984^n)$ time, improving on the $O^*(1.1003^n)$ result by Byskov et al. The chief differences in the algorithm are that we allow longer clauses (at an extra cost to the formula complexity measure; the full bound is $O^*(1.0984^n+l−3m)$ where $l−3m$ is the part due to long clauses), and a more intricate analysis of the case when the formula
is sparse.

### 1.3.3 Hitting Sets

The Hitting Set problem is probably easiest to describe in terms of hypergraphs. A hypergraph is a generalisation of a graph where the edges, sometimes called hyperedges, can be arbitrary sets containing several vertices, and the hitting set problem is the Vertex Cover problem for hypergraphs: given a hypergraph \( H \) on vertices \( V \), find a smallest set of vertices \( T \) such that \( T \cap E \neq \emptyset \) for every hyperedge \( E \) in \( H \) (the set must “hit” every hyperedge). In the \( k \)-Hitting Set (\( k \)-HS) problem, \( |E| \leq k \) for every hyperedge \( E \).

This problem exists under many names: sometimes it is referred to just as the Vertex Cover problem for hypergraphs, or as the Hitting Set problem, and since a set that hits every hyperedge is also known as a transversal, another name is the Minimum Transversal problem. Also, we can observe that there exists a sort of duality operation: for every vertex \( v \) in a hypergraph \( H \), form the set \( \{ E \in H \mid x \in E \} \); the collection of these sets (or edges) is also a hypergraph. Thus, the problem of finding a smallest hitting set in \( H \) is equivalent to the problem of finding a smallest set of hyperedges that include all vertices, a problem known as Minimum Set Cover.

The problem is of course NP-complete (as is Vertex Cover, which is equivalent to 2HS) [41], and hard to approximate within better than a factor of \( k-1 \), if \( k \) is the maximum edge size [21]; a relatively simple algorithm achieves a \( k \)-approximation (i.e. finds, in polynomial time, a hitting set that is at most \( k \) times bigger than the smallest hitting set), and it is conjectured that this is tight (it follows from the unique games conjecture) [21].

The connection to Satisfiability lies in a similarity of structure: for any CNF formula \( F \), we can form a hypergraph by creating, for each clause in \( F \), a hyperedge containing exactly the variables that occur in the clause. Thus, the hypergraph represents the structure of the formula, once negations are ignored (and we actually get yet another formulation of our Hitting Set problem: considering a formula \( F \) where there are no negations, the hitting set problem is equivalent
1. Introduction

Applications of hypergraphs in general, and of the hypergraph transversal problem in particular (see next paragraph), appear in various areas, including database theory and artificial intelligence [28, 47, 64]. For more on hypergraphs, see the book by Berge [4].

Regarding the term transversal, one important problem for hypergraphs is to generate all minimal transversals of a hypergraph \( \mathcal{H} \) (i.e. all hitting sets that do not contain any other hitting set as a subset). The result is another hypergraph \( Tr(\mathcal{H}) \) known as the transversal hypergraph of \( \mathcal{H} \), and the problem of generating this hypergraph (or, in a decision problem setting, deciding whether \( Tr(\mathcal{H}) = \mathcal{G} \) for given hypergraphs \( \mathcal{G}, \mathcal{H} \)) is known as the Transversal Hypergraph problem.\(^4\) Note that this is not identical to the Hitting Set problem; while we could certainly find a minimum hitting set by looking through all minimal hitting sets and comparing their sizes, generating the whole transversal hypergraph could require much more work than we require, since the number of minimal hitting sets can be exponential in the number of vertices, and it is unknown exactly how big this number can be when the possible size of an edge is limited (without such a limit, a hypergraph can have \( \Theta^*(2^n) \) minimal hitting sets). However, a number of algorithms exist for the problem of generating \( Tr(\mathcal{H}) \), including a classic algorithm by Berge [4], an adaption and improvement on this by Kavvadias and Stavropoulos [52], and various algorithms that (under varying restrictions) have a running time that is polynomial\(^5\) in \( |\mathcal{H}| + |Tr(\mathcal{H})| \) [27, 29, 62]. Note that in addition to having an unknown bound on the running time in terms of \( \mathcal{O}^*(c^n) \), some of these algorithms require an exponential amount of space (as they need to remember the whole of \( Tr(\mathcal{H}) \) in the process of generating it).

We treat the problem in the form of 3-Hitting Set. No exact

\(^4\)Sometimes, Transversal Hypergraph is taken to only refer to the decision problem, while the problem of generating \( Tr(\mathcal{H}) \) given \( \mathcal{H} \) gets a name such as Transversal Computation.

\(^5\)Different levels of such polynomiality exist, e.g. \textit{polynomial total time}, \textit{incremental polynomial time}, and \textit{polynomial delay}; see [29] for an overview of references.
1.3. The Problems

Algorithm with a bound of \( O^*(c^n) \) had been published for this version of the problem prior to our article [82], but there have been attacks on the problem within the parameterised setting: given a 3HS instance \( \mathcal{H} \), and a parameter \( k \), one wants to find a hitting set of at most size \( k \) in time on the form \( O(p(n) \cdot c^k) \). A dependence on \( k \) of \( 3^k \) is easily achieved (given an edge \( \{a, b, c\} \) one can try in turn to set \( a = 1 \), \( b = 1 \), and \( c = 1 \)); Niedermeier and Rossmanith [64] give an algorithm where this dependence is \( 2.27^k \), and Fernau [33] later improved this to \( 2.179^k \). Fernau has also given an algorithm for the weighted version of \( k \)-HS [34], which for weighted 3HS runs in \( O^*(2.2470^k) \) time.

The results in this thesis on 3-Hitting Sets are an extension and improvement on our previous work [82], where we gave an algorithm running in time \( O^*(1.6538^n) \), with an exponential memory speedup running in \( O^*(1.6316^n) \), by using Niedermeier and Rossmanith’s parameterised algorithm [64] in a subcase. In this thesis, we give an algorithm that simultaneously improves both the parameterised and classical bounds: it runs in time \( O(p(n) \cdot 2.0755^k) \) given a parameter, and in time \( O^*(1.6359^n) \) in general. There is also a speedup for using exponential space, similar to the previous one [82], that runs in time \( O^*(1.6278^n) \). The main differences to our earlier work are some general improvements to the algorithm, and an improved analysis on how the parameter \( k \) can be bounded for low-degree instances.

1.3.4 Counting Models for Satisfiability

From a computational complexity point of view, the problem class \( \#P \) of problems where you want to know the number of solutions to some problem in NP is a very difficult one. The class was proposed by Valiant in the 1970’s [81], and it was later proved that the so-called polynomial hierarchy is contained in \( P^{\#P} \) [79] (i.e. that a polynomial-time algorithm for any \( \#P \)-complete problem would allow us to solve any problem in the polynomial hierarchy in polynomial time; in fact, a single query to the algorithm would suffice). \( \#P \)-complete problems include the counting counterparts of both NP-complete problems such as \( 3\text{sat} \) (counting counterpart \( \#3\text{sat} \)) and problems that are in P. For instance, while finding a perfect matching in a graph can be done
in polynomial time, counting the number of perfect matchings is \#P-complete. The same story holds for 2SAT: solving 2SAT is polynomial, while \#2SAT is \#P-complete.

From an algorithmic point of view, however, the difference between looking for a solution (any solution), looking for an optimal solution (in some sense), and looking for the number of solutions seems in many cases to be one of which useful “tricks” there are that can be applied when solving the problem. The actual upper bound also seems to be more strongly affected by what the base problem is, with the concrete question (i.e. the complexity class) having a smaller impact (apart from the cases where the decision counterpart is polynomial); this is particularly true when comparing the optimisation and the counting variants. One example is XSAT: as stated above, there is an algorithm that can solve an XSAT instance in time $O^*(1.1730^n)$ [13]; the counting counterpart can be solved in time $O^*(1.2190^n)$ [13]. Another example is provided by the \#2SAT and \#3SAT problems considered in this thesis. For an optimisation version of 2SAT, one can imagine looking for a solution that sets as few variables to true as possible (or some weighted extension of this definition). This problem includes the independent set problem, for which the best bound using polynomial space is $O^*(1.2025^n)$, while the problem \#2SAT$_w$ (where the subindex $w$ signifies that we are counting max-weight solutions) in this thesis receives an algorithm with a running time bounded by $O^*(1.2377^n)$. For 3SAT, the optimisation version includes 3-Hitting Set in the same way as Independent Set is included in the optimisation version of 2SAT. While the best results for the decision variant are based on local search (the best upper bound for an exact algorithm is $O^*(1.473^n)$ [8]), we have a gap between the optimisation and counting variants of 3SAT of $O^*(1.6359^n)$ for 3-Hitting Set with polynomial space and $O^*(1.6671^n)$ for \#3SAT$_w$. (Note however that the MAX SAT-type problems, where the goal is to satisfy a maximum number of clauses instead of all clauses, seem far harder in terms of upper bounds, and behave quite differently.)

Earlier work on the \#2SAT and \#3SAT problems appears in Dubois [25], Zhang [87], Littman, Pitassi, and Impagliazzo [59], and Dahllöf,
Jonsson, and Wahlström [14, 15]. The work in this thesis for both problems is based on that by Dahllof, Jonsson, and Wahlström [15], where the bounds $O^*(1.2561^n)$ for $\#2SAT_w$ and $O^*(1.6737^n)$ for $\#3SAT_w$ are given. In the case of $\#2SAT_w$, Furer and Kasiviswanathan [40] have performed a more detailed analysis of the same algorithm, and achieved the bound of $O^*(1.2461^n)$. In this thesis, we improve the method of analysis and prove the bound $O^*(1.2377^n)$ for the same algorithm. In the case of $\#3SAT_w$, we perform a more careful analysis of essentially the same algorithm and prove the bound of $O^*(1.6671^n)$.

Note that the improvement of the bound for $\#2SAT_w$ translates into improvements for some problems whose current best solutions involve reductions to this problem, e.g. counting solutions for Constraint Satisfaction problems on non-binary domains [1]. It is possible that attacking some of these problems (e.g. counting solutions for problems on 3-valued domains) directly with the methods of this thesis, instead of reducing them to $\#2SAT_w$, can yield still better bounds. However, a general improvement on $d$-valued domains for all $d$ would likely require advancements in algorithm analysis.

1.4 Outline of the Thesis

Two more preliminary chapters follow, making up Part I of the thesis. In Chapter 2, we give definitions of notation, common terms, and other preliminaries, then in Chapter 3 we give a closer presentation of the method of estimating upper bounds by using complexity measures, and introduce the different categories of complexity measures that are used. After this, there are five chapters on the individual problems, divided into three parts. Part II deals with decision problems, consisting of Chapter 4 which treats the problem of satisfiability in sparse formulae and Chapter 5 which treats the problem of exact 3-satisfiability; Part III deals with optimisation problems, consisting of Chapter 6, which treats the problem of 3-hitting set; and Part IV deals with counting problems and consists of Chapters 7 and 8 which treat the problems of counting max-weight models for $2SAT$ and $3SAT$ formulae, respectively. Finally, Chapter 9 contains conclusions and
1. Introduction

directions for future work.

Unfortunately, we must admit that many of the proofs of this thesis are somewhat lengthy (consisting mostly of case enumeration). This is due to an attempt to make them more complete than what is usually done, when the length of a publication is an issue. We hope in this way to avoid hidden traps, or unpleasant surprises in the proofs, which may otherwise have a tendency to sneak in.
1.4. Outline of the Thesis
Chapter 2

Preliminaries

Here, we will give the definitions and technical background for the material that will appear in the rest of the thesis, and describe our notation.

2.1 Boolean Formulae

For the most part of this thesis, the formulae considered will be standard satisfiability formulae in conjunctive normal form (CNF). Such a formula $F = (a \lor b \lor c) \land (\bar{a} \lor d) \land \ldots$ is a conjunction of distinct clauses $C_i$, where each clause is a disjunction of literals, and each literal either $v$ or $\bar{v}$ for some boolean variable $v$ (referred to as positive and negative occurrences of $v$, respectively). A clause may contain both $v$ and $\bar{v}$ for a variable $v$, in which case it is a trivial clause. Whether a clause is allowed to contain multiple copies of a literal or not varies depending on the application: in Chapter 4, where such clauses can be created during the execution of the algorithm, we remove duplicate copies of a literal explicitly for clarity, while in Chapters 7 and 8, this is not necessary and no duplicate literals are allowed in any clause. (Note that the problems of Chapters 5 and 6 do not use disjunctive clauses.) A boolean variable $v$ can take values $v = 1$, in which case the literal $v$ is true, or $v = 0$, in which case the literal $\bar{v}$ is true; a clause is satisfied by an assignment if any of its literals is true, and a
formula is satisfied if all of its clauses are satisfied. A model $M$ for a formula $F$ is an assignment to all its variables such that $F$ is satisfied.

Note that we do not intend to say anything about the internal representation of a formula in a computer program by this; these definitions only serve to define how a formula is written in the text. The details of the internal representation do not matter for the exponential behaviour of the running time, but only affect the polynomial factors which we ignore. Also, in a pragmatic manner, we can treat a formula as a set of clauses, and a clause as a set or multiset of literals, since the order of clauses in a formula and literals in a clause does not affect the meaning.

Sometimes, we refer to a literal by name, e.g. $l$, without specifying whether it is a positive or negative occurrence of a variable. In such a case, $\bar{l}$ would refer to $\bar{a}$ if $l = a$ and to $a$ if $l = \bar{a}$. We use the convention that any literal referred to by the letter $l$ (such as $l'$ or $l_j$) can be either a positive or a negative occurrence, while any literal referred to by another lowercase letter is exactly as written (i.e. $v$ will be a positive occurrence, and $\bar{v}$ will be negative). A literal $\bar{v}$ is either $v$ or $\bar{v}$.

Sometimes, a clause is given a description such as $(l \lor C)$ or $(l \lor C \lor D)$. This is understood to be a clause containing the literal $l$ and every literal occurring in $C$ (or in $C$ or $D$). Unless explicitly stated otherwise, such a $C$ is assumed to be non-empty. In the second form, the same goes for $D$, and in addition, $C$ and $D$ are assumed to have an empty intersection, and in either case, neither contains $l$. For two clauses $C$ and $D$, if every literal of $C$ appears in $D$, then $C$ subsumes $D$, and an assignment that satisfies $C$ will necessarily satisfy $D$ as well.

A clause which contains exactly $k$ literals is a $k$-clause; $F$ is a $k$-SAT-formula if each of its clauses contains at most $k$ literals. If every clause in $F$ has exactly $k$ literals, then $F$ is $k$-uniform. For a clause $C$, $|C|$ denotes the number of literals in $C$. A variable $v$ which occurs exactly $k$ times in total in $F$ (counting both $v$ and $\bar{v}$) is a $k$-variable, and if it occurs exactly $k_1$ times unnegated and exactly $k_2$ times negated, then it is also a $(k_1, k_2)$-variable. We say that
the degree of \( v \) in \( F \) is \( d(v, F) = k \), i.e. the number of occurrences of \( v \) in \( F \); usually we just use \( d(v) = k \), where the formula \( F \) is understood from the context. We also say that the positive degree of \( v \) is \( d^+(v) = k_1 \), and that the negative degree of \( v \) is \( d^-(v) = k_2 \). We use \( d(F) \) for the maximum degree of any variable occurring in \( F \). If every variable in \( F \) occurs exactly \( k \) times, then \( F \) is said to be \( k \)-regular. Any \((k,0)\)- or \((0,k)\)-variable is called a pure variable (and a literal \( l \) of such a variable occurring in a formula is a pure literal). A singleton is a variable \( v \) with \( d(v) = 1 \). A heavy variable is a variable \( v \) with \( d(v) > 2 \).

\( \text{Vars}(F) \) is the set of all variables that occur in some clause of \( F \). We use \( n(F) \) for the number of variables in \( F \) \((n(F) = |\text{Vars}(F)|)\) and \( m(F) \) for the number of clauses in \( F \). We use \( l(F) \) for the total length of \( F \), i.e. \( l(F) = \sum_{C \in F} |C| = \sum_{v \in \text{Vars}(F)} d(v, F) \), \( n_i(F) \) for the number of \( i \)-variables in \( F \), and \( m_i(F) \) for the number of \( i \)-clauses in \( F \).

For a literal \( l \), the neighbours of \( l \) are all literals \( l' \neq l \) such that some clause \( C \) in \( F \) contains both \( l \) and \( l' \). If a clause \( C \) contains a literal of both (distinct) variables \( a \) and \( b \), then \( a \) and \( b \) co-occur in \( C \).

For a formula \( F \), containing a variable \( a \), let \( F[a = 1] \) be the result of deleting every clause \((a \lor C)\) in \( F \) and shortening every clause \((\bar{a} \lor C)\) in \( F \) to \((C)\), where \( C \) is allowed to be empty (i.e. a clause \((a)\) is deleted and a clause \((\bar{a})\) is shortened to a contradiction ()). We define \( F[a = 0] \) conversely, and \( F[l = 1] \) for a literal \( l \) has the natural definition. This process of shortening and deleting is the propagation of the assignment \( a = 1 \).

We also need the classic concept of resolution\(^1\) \([20]\). For clauses \( C = (a \lor l_1 \lor \cdots \lor l_d) \) and \( D = (\bar{a} \lor l'_1 \lor \cdots \lor l'_{d'}) \), the resolvent of \( C \) and \( D \) by \( a \) is the clause \((l_1 \lor \cdots \lor l_d \lor l'_1 \lor \cdots \lor l'_{d'}) \), shortened to remove duplicate literals. If this new clause contains both \( v \) and \( \bar{v} \) for some variable \( v \), then it is said to be a trivial resolvent. For a formula \( F \) and a variable \( v \) occurring in \( F \), \( DP_v(F) \) is the formula where all non-

\(^1\)More general types of resolution exist in the literature, but in this thesis we only use this variant, which is also known as DP-resolution.
trivial resolvents by \( v \) have been added to \( F \) and all clauses containing the variable \( v \) have been removed from \( F \). Resolution is the process of creating \( DP_v(F) \) from \( F \).

### 2.2 Graphs and Hypergraphs

A graph \( G = (V, E) \) consists of a set of vertices \( V \) and a set of edges \( E \), where an edge is an unordered pair of distinct vertices \( (u, v) \) (i.e. a set \( \{u, v\} \) where \( u \neq v \), though we will rather use the notation \( (u, v) \)). We use \( n(G) \) for the number of vertices, and \( m(G) \) for the number of edges of a graph. In many cases, \( n \) and \( m \) will be used when \( G \) is clear from the context.

We use \( d(v, G) \) or \( d(v) \) for the degree of \( v \) in \( G \): the number of edges in \( G \) that contain \( v \). Much of the degree-related concepts that we introduced for formulae apply to graphs: \( d(G) \) is the maximum degree of any vertex in \( G \); a graph is \( k \)-regular if \( d(v, G) = k \) for every vertex \( v \) in \( G \); a vertex of degree \( k \) is referred to as a \( k \)-vertex, or a singleton if \( k = 1 \); and \( n_i(G) \) is the number of vertices of \( G \) that have degree \( i \).

For a vertex \( v \) in a graph \( G = (V, E) \), the (open) neighbourhood \( N(v) \) of \( v \) is the set of all vertices \( w \) such that there exists an edge \( (v, w) \) in \( E \). The closed neighbourhood \( N[v] \) is defined as \( N(v) \cup \{v\} \).

A set \( S \) of vertices is independent if no edge \( (u, v) \) exists such that \( u, v \in S \). A vertex cover is a set that includes some vertex from each edge. We see that these are dual concepts: if \( S \) is independent, then \( \overline{S} = V - S \) must be a vertex cover, and vice versa.

A hypergraph \( \mathcal{H} = \{E_1, E_2, \ldots, E_m\} \) is a generalisation of a graph, where the edges \( E_i \) are arbitrary sets called hyperedges. The vertices of \( \mathcal{H} \) are \( V(\mathcal{H}) = \bigcup E_i \). Sometimes \( \mathcal{H} \) is given as a pair \( (V, \mathcal{E}) \) where \( V \) are the vertices and \( \mathcal{E} \) are the hyperedges, but for our purposes, the definition we use is simpler.

We use \( n(\mathcal{H}) \) for the number of vertices, and \( m(\mathcal{H}) = |\mathcal{H}| \) for the number of hyperedges of \( \mathcal{H} \). The degree \( d(v, \mathcal{H}) = |\{E_i \in \mathcal{H} \mid v \in E_i\}| \) of a vertex \( v \) in \( \mathcal{H} \) is the number of edges that contain \( v \). As with formulae and graphs, \( d(\mathcal{H}) \) is the maximum degree of any vertex.
2. Preliminaries

in $\mathcal{H}$ (or 0, if $\mathcal{H}$ contains no vertices); a hypergraph is \( k \)-regular if \( d(v, \mathcal{H}) = k \) for every vertex that appears in $\mathcal{H}$; a vertex of degree $k$ is referred to as a \( k \)-vertex, or a singleton if $k = 1$; and $n_i(\mathcal{H})$ is the number of vertices of $\mathcal{H}$ that have degree $i$. In addition, we define $d_k(v, \mathcal{H})$ as the number of hyperedges of cardinality $k$ that contain $v$.

The rank of a hypergraph is the maximum cardinality of any hyperedge in it (or 0, if the hypergraph is empty), and a hypergraph $\mathcal{H}$ is \( r \)-uniform if $|E| = r$ for every $E \in \mathcal{H}$. An edge of cardinality $k$ is a \( k \)-edge, and an edge of cardinality 1 is a loop.

A transversal is the hypergraph equivalent of a vertex cover: a transversal of $\mathcal{H}$ is a set $T$ such that $T \cap E \neq \emptyset$ for every $E \in \mathcal{H}$. A transversal is also called a hitting set, particularly in the context of the problem $k$-Hitting Set (as defined in the next section).

We will sometimes use graph theoretic concepts, e.g. connected, in the context of formulae. In these cases, we refer implicitly to the graph (or hypergraph) of a formula: for a 2sat formula $F$, the graph $G = (V, E)$ is the graph where $V = Vars(F)$ and where there is an edge $(u, v)$ for every clause $(\bar{u} \lor \bar{v})$ in $F$. For a 3sat or sat formula $F$, the graph of $F$ is the hypergraph with an edge $\{v_1, \ldots, v_d\}$ for every clause $(\bar{v}_1 \lor \ldots \lor \bar{v}_d)$ in $F$. The formula $F$ is connected if the graph of $F$ is connected.

2.3 Problem Definitions

We will now give the precise definitions of the problems considered in this thesis.

- **Satisfiability (sat)**
  
  *Instance*: A cnf formula $F$.

  *Question*: Does there exist a satisfying assignment to $F$?

- **$k$-Satisfiability ($k$-sat)**

  *Instance*: A cnf formula $F$ where each clause has at most $k$ literals.

  *Question*: Does there exist a satisfying assignment to $F$?
2.3. Problem Definitions

- **Exact Satisfiability (Xsat)**
  
  *Instance:* A cnf formula $F$.

  *Question:* Does there exist an assignment to all variables in $F$ such that each clause is satisfied by exactly one literal?

- **Exact $k$-Satisfiability ($X_{k}sat$)**

  *Instance:* A cnf formula $F$ where each clause has at most $k$ literals.
  
  *Question:* Does there exist an assignment to all variables in $F$ such that each clause is satisfied by exactly one literal?

- **$k$-Hitting Set ($k$-HS)**

  *Instance:* A hypergraph $H$ of rank $k$.

  *Question:* What is the cardinality of the smallest hitting set of $H$?

  *Comment:* In some papers from the field of parameterised complexity, this problem is defined to also include a parameter $p$ defining the largest hitting set we would be interested in (i.e. the question becomes “does there exist a hitting set of cardinality at most $p$”). Such a parameter is not expected in this thesis.

- **Counting Weighted $k$-Satisfiability ($\#k$-sat$_w$)**

  *Instance:* A cnf formula $F$ where each clause has at most $k$ literals, along with a real-valued vector $w$ defining the weight of each literal.

  *Question:* If the weight of a model $M$ for $F$ is

  $$W(M) = \sum_{l \text{ is true in } M} w_l$$

  how many max-weight models does $F$ have (i.e. how many models $M$ have a weight that is identical to the maximum weight for any model of $F$)?
2.4 Algorithm and Branching Concepts

The common method used by our algorithms in most cases is the branching. In its most basic variant, we select a branching variable \( a \), and branch on it, i.e. make one recursive call with \( a = 1 \) and another with \( a = 0 \), in each call propagating the effects of the assignment, and then we calculate the solution from these results. The recursive calls made by the algorithm are often visualised as a tree, which explains the terminology (as each call, leading to further sub-calls, results in a branch of the tree).

The method is sometimes extended to more complicated branchings, with more than one assignment made in each call, and possibly more than two recursive calls made. In either case, we have to make sure that all the branches of the branchings, collectively, cover all relevant possibilities so that the right answer can be guaranteed.

We identify branches by the assignments made in them, e.g. the branch \( a=1 \) and the branch \( a=0 \) in the previous example. Alternatively, we may sometimes talk of the assignment \( a \) (resp. \( \bar{a} \)) to refer to an assignment \( a = 1 \) (resp. \( a = 0 \)), and the branch \( a \) (resp. \( \bar{a} \)) to refer to the branch \( a = 1 \) (resp. \( a = 0 \)).

The recursion process terminates in either trivial cases (such as when a formula is empty or contains a direct contradiction), or in cases where another algorithm can solve the remaining part of the problem fast enough (e.g. in polynomial time). These are the base cases of the algorithm. In addition to these, there are usually a handful of cases making only one recursive call, involving either some kind of clean-up work (such as removing a subsumed clause) or making some safe or forced assignment (such as assigning \( l = 1 \) when a formula contains a clause (l)). These cases are referred to as reductions as they reduce the current instance to some smaller or otherwise easier instance. When no reduction or base case applies, we say that the instance is fully reduced.

The approach we use for analysing the running time of such an algorithm is based on a measure of complexity \( f(F) \). We say that \( f(F) \) is a well-behaved measure for a certain algorithm if the following hold:
2.4. Algorithm and Branching Concepts

1. $f(F) \geq 0$ for all possible $F$;

2. $f(F) = 0$ only when $F$ is solved in polynomial time by the algorithm;

3. $f(F') \leq f(F)$ if the algorithm, when applied to $F$, applies a reduction replacing $F$ by $F'$; and

4. $f(F') < f(F)$ if the algorithm, when applied to $F$, performs a branching where $F'$ is one of the branches.

Assume that $f(F)$ is a well-behaved measure of complexity for some algorithm, which has a single branching rule that, for a formula $F$, creates recursive calls for subproblems $F_1, \ldots, F_d$. If we can guarantee that $f(F_i) \leq f(F) - \delta_i$ for each $i$, then we say that $(\delta_1, \ldots, \delta_d)$ is the branching tuple of the branching, and we can calculate a numerical value $c$ from this branching tuple, known as the branching number, such that the running time of the algorithm is in $O^*(c^{f(F)})$ (the notation $O^*(f)$ means that polynomial factors have been suppressed). The branching number is the unique positive solution to

$$
\sum_i x^{-\delta_i} = 1
$$

For a proof of this, we refer to Kullmann’s paper on 3SAT [56]. The values $\delta_i = f(F) - f(F_i)$ are referred to as the reduction of $f$ in branch $i$ (not to be confused with the previous usage of reduction, which refers to certain rules in an algorithm); we will use $\Delta_i f := f(F) - f(F_i)$ to denote this value. We use $\tau(\ldots)$ as a name for the function that returns this branching number, e.g. $\tau(\delta_1, \ldots, \delta_d) = c$.

If several different branchings are possible in the algorithm, then the running time is still in $O^*(c^{f(F)})$, if $c$ is the maximum branching number over all possible branchings. The branchings with the highest branching number will be referred to as the hard cases of the algorithm. A few observations can be made about the $\tau$ function; to begin

---

2This is sometimes modified to say that $f(F') \leq f(F)$ must hold over the whole chain of reductions, i.e. when $F'$ is the fully reduced version of $F$. 
with, it is invariant under reordering of the terms, increasing any term \( \delta_i \) will decrease the branching number, and it can be shown that for any \( 0 < a < \delta_1 \), \( \tau(\delta_1, \delta_2, \delta_3, \ldots, \delta_d) \leq \tau(\delta_1 - a, \delta_2 + a, \delta_3, \ldots, \delta_d) \) if and only if \( \delta_1 - a \leq \delta_2 \). In particular, it is true when \( \delta_1 \leq \delta_2 \), leading to the observation that the branching number is the highest when the reductions in the branching are as unbalanced as possible. This also has the consequence that changing the reduction in a branch with a small \( \delta_i \) has a bigger impact on the branching number than a corresponding change in a branch with a greater reduction.

For two branching tuples \( B_a = (a_1, \ldots, a_d) \) and \( B_b = (b_1, \ldots, b_d) \), we say that \( B_a \) dominates \( B_b \) if \( a_i \leq b_i \) for each \( i \), ensuring that \( \tau(B_a) \geq \tau(B_b) \). For binary branchings \( B_a = (a_1, a_2) \) and \( B_b = (b_1, b_2) \), with \( a_1 \leq a_2 \) and \( b_1 \leq b_2 \), we say that \( B_a \) is a more balanced version of \( B_b \) if \( a_1 + a_2 = b_1 + b_2 \) and \( a_1 > b_1 \).

Regarding the quality of the method of analysis by branching numbers, it is well known that if a branching tree has a branching number of exactly \( c \) in every node, and a measure of \( n \) in the root instance, then the number of leaves of the tree will be \( c^n \). See for instance Lemma 14.2 of Kullmann’s original paper [56], or the Lemmas and the Theorem of Section 6 of Eppstein’s paper on quasiconvex analysis [31]; the statement can also easily be proved inductively.
2.4. Algorithm and Branching Concepts
Chapter 3

Measures of Complexity

In this chapter, we examine the process of analysis closer, and define the different kinds of measures used in the analysis of the running time of the algorithms. We also show how part of the analysis can be performed automatically by a computer program for each kind of measure, and for some of the measures we provide tightness results for this process. For Eppstein’s weight-based measures, these results are known [31]; for the other measures, the results are new.

We begin in Section 3.1 with an example of a simple analysis, then we give a general overview in Section 3.2 of the different kinds of measures used. After that, we give the actual descriptions: in Section 3.3 we describe Eppstein’s method of quasi-convex analysis of multivariate recurrences, and the weight-based measure used therein [31]; in Section 3.4 we describe our state-based measure for analysis based on finite global states, and how to automate this analysis; then in Section 3.5 we present our compound measure, and a way to automate this analysis.

3.1 Introductory Example

We will now look closer at how a measure of complexity can be constructed, but first, a (hopefully) clarifying example. As we have seen, the word “branching” can be used with different meanings. To illus-
Algorithm SimpleSAT$(F)$

0. If $F$ is empty, then return 1. If $F$ contains an empty clause, then return 0.

1. If $(l) \in F$, then return SimpleSAT($F[l = 1]$).

2. If there is a pure literal $l$ in $F$, then return SimpleSAT($F[l = 1]$).

3. Pick any variable $v$ and return SimpleSAT($F[v = 1]$) $\lor$ SimpleSAT($F[v = 0]$).

Figure 3.1: A simple algorithm for deciding satisfiability

trate, consider the Satisfiability algorithm shown in Figure 3.1. Case 0 of the algorithm contains the base cases. Cases 1 and 2 contain reductions (where the first one is a forced assignment and the second is a safe assumption, as there is no reason to assign $l = 0$ if $l$ is a pure literal), and case 3 contains a branching rule. If we only look at the immediate effect of applying this case, then we could perhaps say that the algorithm uses only a single branching (as the immediate assignments in the two branches are always the same, even though the variable changes). However, when calculating $\Delta f = f(F) - f(F')$, we will often let $F'$ be the fully reduced result of applying the reductions as well as the prescribed assignment (for instance, if there is a 2-clause $(v \lor w) \in F$ and we branch on $v$, then we will include the effects of assigning $w = 1$ in the $v = 0$ branch). With this view, our single branching rule can cause several different branchings to occur; on the one hand, the basic $v = 0 / v = 1$; on the other hand, any of several improved versions such as $v = 0, w = 1 / v = 1$.

Now, the most classical measure of all is probably to just use $n(F)$. In this case, we see clearly that $n(F)$ is a well-behaved measure for SimpleSAT, and SimpleSAT$(F)$ is most definitely contained in

---

1This is of course not a competitive algorithm for the problem, but only meant as an example.
3. Measures of Complexity

$O^\ast(2^n(F))$, as the worst-case branching number for the branching is $\tau(1,1) = 2$, but this seems to be the best we can say, and indeed, in the general case it seems impossible to solve SAT in a time $O^\ast(c^n)$ for $c < 2$. On the other hand, we could be analysing the behaviour of SimpleSAT in terms of the length of the formula, $l(F)$. This is also a well-behaved measure, but now the case analysis becomes slightly more involved. Just for the sake of the example, let us go through the case analysis in this simplified form to see what it would look like.

We have a worst case when $d(v) = 2$ and $v$ is not pure, since more occurrences would mean that more literals are removed and at least two occurrences must exist when case 2 does not apply, so let the clause containing $v$ be $C$ and the clause containing $\bar{v}$ be $D$. On the one hand, it would be possible that $|C| = |D| = 3$, in which case the propagation would decrease $l(F)$ by 4 in both branches, for a branching number of $\tau(4,4) = 2^{1/4} < 1.1892$. On the other hand, if $|C| = 2$, say $C = (v \lor a)$, then we could have a lower reduction of $l(F)$ in the $v = 1$ branch, but in the $v = 0$ branch case 1 of the algorithm sets $a = 1$, and $l(F)$ decreases in this branch by at least $|D| + |C|$ (more if $a$ has an occurrence outside of $C$ and $D$). $|C| = |D| = 2$ implies that this applies to both branches, for a branching number of $\tau(4,4) < 1.1892$, and if $|C| = 2$ and $|D| = 3$ then we could have $\tau(3,5) < 1.1939$. Every other possible case means adding more clauses to be removed or just making $C$ or $D$ longer, which definitely does not make the branching number worse. The worst-case branching number for this algorithm would be 1.1939 and since $l(F)$ is a well-behaved measure, this algorithm runs in time $O^\ast(1.1939^{l(F)})$.

Of course, $O^\ast(1.1939^F)$ is a far way from $O^\ast(1.0663^F)$, the bound which is given in Chapter 4, so there is a lot of room for improvement. If one wanted to improve this, then there are a number of things one could try. First of all, the analysis is not tight even for this simple algorithm, but let us overlook that. The first thing we would probably want is to do as many things in reductions or in polynomial time as possible. For instance, the algorithm does not use resolution. Applying resolution to a (1,1)-variable $v$ does not in-

---

All branching numbers in this thesis are rounded upwards.
crease \( l(F) \), so we could add a case 2.5: “If there is a \((1,1)\)-variable \( v \) in \( F \), then return \( \text{SimpleSAT}(DP_v(F)) \).” This reduction would eliminate the hard cases described in the previous paragraph. As for polynomial cases, we could for instance add a case that checks if \( F \) is a \( 2\text{sat} \)-instance and applies a polynomial algorithm if it is. We could also add similar checks for algorithms relating to matching or similar properties. However, before adding these checks, we should probably have a reason to believe that these extra cases will actually improve the worst-case running time. If we start adding checks for such cases simply because we can, even if these cases do not affect the hard cases of the algorithm and are not likely to occur very often, then we end up with a cluttered algorithm that becomes hard to implement.

Another natural step is to modify or extend the branching cases of the algorithm, to either avoid the hard cases if possible or to add new, possibly more complicated branchings that deal with the hard cases in a different way. In \( \text{SimpleSAT} \), case 3 should probably be modified to specify how \( v \) is chosen, for instance by saying that \( d(v) \) or \( \min(d^+(v), d^-(v)) \) should be maximised. When such simple modifications can no longer be made to improve or avoid the hard cases, a common step is to look at each case that is judged by the analysis to be hard and try to find a new rule to add to the algorithm that will deal with this case in a more efficient way. In doing so, though, we again run a risk of creating a cluttered, hard-to-implement algorithm.

In addition to these concerns, it is often stated (on the level of “folklore” or common knowledge) that adding too many cases to an algorithm does not improve the actual observed efficiency, and may even somewhat increase the running time. Sometimes this effect is explained by the “hidden constants” of the \( O(\cdot) \) notation, implying that the extra cases would start to make an observable difference if we were able to apply our algorithms to large enough problems, but this is a premature conclusion. Other possible sources of the effect are the possibility that hard problem instances do exist for real-world sizes, but that they are so rare and sparsely distributed that they are hard to find or create in experiments, or (probably most importantly, as far as the need for many cases in an algorithm goes) the possibility
that the effect on the theoretical upper bound of these cases is mostly an artifact of the application of the theory.

Using the right kind of complexity measures ties into this last point. When analysing upper bounds for branching algorithms, methods based around calculating branching numbers are very common (although the terminology and the notation are not always the same as what we use), and with such a method, as noted, to prove a good time bound essentially requires proving that every single branching has a good branching number. If the analysis is only performed in terms of the value of $n$, then this means that the algorithm essentially has to guarantee that each possible branching removes a large number of variables. In contrast, if the analysis is performed through a measure that includes effects such as the number of short clauses and the degrees of the variables of a formula, or some other property that has an influence on the possible branchings, then the analysis only needs to show that each branching is good enough in one of these features, or rather that it is good enough in the combination of these features. It is often the case that with a measure that assigns a value to these effects, we can keep the algorithms natural and still prove strong upper bounds.

Another effect, which has already been implied, is the connection between choice of measure and choice of reductions: we can generally speaking only allow ourselves to use reductions which decrease the measure of the instance. Thus, we can use reductions to get rid of troublesome cases if we use a measure $f$ in which these reductions are proven to reduce $f(F)$, and for which a good upper bound on the running time is possible. For an example, consider the case when two clauses $C$ and $D$ overlap on exactly two literals, say $C = (C' \lor E')$ and $D = (D' \lor E')$. Replacing $C$ and $D$ by $(C' \lor x), (D' \lor x), (\overline{x} \lor E')$ for a new variable $x$ results in an equivalent formula, but since this reduction increases both $l(F), m(F)$ and $n(F)$, it would not normally be used. However, using this reduction means that the variables of $E'$ decrease their degrees, and in some situations, this would count as progress. If we use a measure of complexity in which the degree of each variable is an important effect (so that the introduction of one
3.2. Non-classical Measures

3-variable hurts the measure less than the reduction of the degrees of two variables improves it), then this replacement would be a good idea to perform. The sat algorithm which is presented in Chapter 4 is analysed in terms of a measure where this is true in some cases, and in these cases, the algorithm does perform such replacements.

One could perhaps distinguish between, on the one hand, performing an analysis in terms of many or few properties of the same kind, and on the other hand introducing qualitatively new properties into the analysis. In the first case, we can compare performing an analysis in terms of the number \( n_i(F) \) of variables of degree \( i \) for each \( i \), to performing an analysis either in terms of \( l(F) \) and \( n(F) \), where these degrees are somewhat implicit, or in the single property

\[
\sum_{v \in \text{Vars}(F)} \max(0, d(v) - 2) = l(F) - 2n(F) + s(F)
\]

where \( s(F) \) is the number of singletons in \( F \), as is done in part of Chapter 4; in the second case, we could consider adding components relating to the number and lengths of clauses to said analysis. It seems, perhaps, that the former is more related to the naturalness of the algorithm (the need to add complicated cases to the algorithm may be countered by such a more fine-grained analysis of the effects of the existing cases), while the latter is more related to the kinds of reductions and branching strategies that can be used (as you can design the algorithm to make use of entirely different effects).

The rest of this chapter is devoted to introducing and describing the different kinds of measures that are used in the analyses in this thesis.

3.2 Non-classical Measures

Our example analysis in Section 3.1 was entirely classical, with a single attribute \( l(F) \) being counted. To present the extensions of this method, let us point out that the process can be viewed as performed in two phases: first, the behaviour of the algorithm is modelled in
3. Measures of Complexity

\[ T(l(F)) = \max \begin{cases} T(l(F) - 4) + T(l(F) - 4) \\ T(l(F) - 3) + T(l(F) - 5) \end{cases} \]

Figure 3.2: The recurrence constructed in Section 3.1 (base cases omitted)

terms of the attribute(s) considered, in the form of a recurrence; then this model is analysed and a bound on its asymptotic growth is given.

For instance, the recurrence which is constructed in Section 3.1 is given in Figure 3.2. Of course, in such a classical case (where “classic” refers to the use of only a single parameter \( l(F) \)) the second phase is easy enough to be invisible: for every branching which would add a line to the recurrence, the branching number is calculated by the \( \tau(\cdot) \) function, and we immediately find out whether this case is better or worse than the worst of the cases so far. In such a case, there is also no question about the tightness of this second phase of the analysis; the bound \( O^*(\tau(3,5)^{l(F)}) \) is indeed the tightest possible for the recurrence in Figure 3.2. (This can even be seen by a direct inductive proof, the inductive step being: if \( T(x') = kc^{x'} \) for all \( x' < x \), then \( T(x) = kc^{x-3} + kc^{x-5} = kc^x \) for \( c = \tau(3,5) \), by the definition of \( \tau(\cdot) \).)

However, since we do take the step via a model, any information that is not included in the model (i.e. any information not inherent in the change in the value of \( l(F) \)) is “lost”, with no impact on the final bound. By replacing our model (i.e. the type of recurrence that is allowed) by something more advanced, more such information can influence the final bound (as previously observed).

An abstract example of the type of recurrence used with Eppstein’s method, which is introduced in Section 3.3, is given in Figure 3.3. The change is that we now allow any number of parameters to the recurrence, which significantly increases both the amount of information that can be included in the recurrence, and the apparent difficulties of the second phase, that of producing a tight upper bound on the growth of this system. The way this second phase is handled
3.2. Non-classical Measures

\[
T(a, b) = \max \begin{cases} 
T(a - \delta_{1,1,1}, b - \delta_{1,1,2}) + T(a - \delta_{1,2,1}, b - \delta_{1,2,2}) \\
T(a - \delta_{2,1,1}, b - \delta_{2,1,2}) + T(a - \delta_{2,2,1}, b - \delta_{2,2,2}) \\
\ldots
\end{cases}
\]

**Figure 3.3:** An example multi-variate recurrence

In Eppstein’s method is to reduce the recurrence of Figure 3.3 to a recurrence expressed in a single measure \( f(F) = w_a a(F) + w_b b(F) \), with appropriate values of \( w_a \) and \( w_b \), and then handle the analysis in terms of \( f(F) \) in the same way as the analysis of the model in Figure 3.2 is performed. As long as the resulting measure \( f(F) \) is well-behaved, such an analysis will certainly produce a valid bound, and Eppstein has both shown that such an approach will produce a tight upper bound for some values of the weights \( w \), and provided an algorithm for finding the best such values for a given recurrence [31]. We will provide more details on this in Section 3.3.

One limitation that does remain in a model such as that in Figure 3.3 is that there is no easy way to express conditions on the applicability of a branching (i.e. limits on under what circumstances a line of the recurrence can be used). The two further measures that we use introduce different ways to get around this.

The first extension, referred to as state-based analysis or analysis by finite global states, uses an explicit concept of state in the model; an example recurrence is in Figure 3.4. In this example, there are three states (“1”–“3”); there can of course be an arbitrary number. Note that for every state, there is a separate list of possible cases, and in every branch of every possible case, the state of the resulting instance is explicitly provided. This type of model is presented in more detail in Section 3.4, where we also describe the associated complexity measure and show how to convert such a state-based model into the form required by Eppstein’s method, so that the second phase of the analysis can be performed automatically. For certain types of recurrences we also show that the bound produced by the analysis is
3. Measures of Complexity

\[
\begin{align*}
T_1(a, b) &= \max \left\{ T_1(a - \delta_{1,1,1}, b - \delta_{1,1,2}) + T_2(a - \delta_{1,2,1}, b - \delta_{1,2,2}) \right. \\
& \quad \left. \ldots \right. \\
T_2(a, b) &= \max \left\{ T_1(a - \delta_{2,1,1}, b - \delta_{2,1,2}) + T_3(a - \delta_{2,2,1}, b - \delta_{2,2,2}) \right. \\
& \quad \left. \ldots \right. \\
T_3(a, b) &= \ldots
\end{align*}
\]

**Figure 3.4:** An example recurrence for state-based analysis

tight with respect to the model.

If preferred, of course such a model can be visualised as a state diagram rather than connected recurrences. Figure 3.5 contains an illustration from Chapter 6 of the hard cases under a certain analysis of the algorithm MinTr defined there for the 3-Hitting Set problem.

In the second method we introduce, referred to as analysis by compound measure, the applicable branchings depend on the relative values of the modelled attributes. In Figure 3.6 there is an example for two parameters \(a\) and \(b\), but the method covers cases with more parameters and other patterns of state division as well. Note the difference compared to the model with explicit states: here, the individual branchings contain no information about the state of the resulting instances, but the recurrence of \(T_2\) is considered applicable until \(a < 2b\), when the “state” is changed and \(T_1\) becomes applicable instead. In Section 3.5, we give more details on this kind of model, describe the associated complexity measures (referred to as compound measures), and again describe how to reduce to Eppstein’s model so that the second phase of analysis can be performed automatically. Unfortunately, no tightness results are known for the bounds produced by this method.
3.2. Non-classical Measures

Two 2−edges

No 2−edges

One 2−edge

Three 2−edges

Two 2−edges

Figure 3.5: Example state-based model in graphical form (from Chapter 6)

\[ T(a, b) = \begin{cases} 
0 & \text{if } a < b \\
T_1(a, b) & \text{if } b \leq a < 2b \\
T_2(a, b) & \text{if } 2b \leq a < 3b \\
\ldots & 
\end{cases} \]

\[ T_{1}(a, b) = \max \begin{cases} 
T(a - \delta_{1,1,1}, b - \delta_{1,1,2}) + T(a - \delta_{1,2,1}, b - \delta_{1,2,2}) \\
T(a - \delta_{2,1,1}, b - \delta_{2,1,2}) + T(a - \delta_{2,2,1}, b - \delta_{2,2,2}) \\
\ldots & 
\end{cases} \]

\[ T_{2}(a, b) = \max \begin{cases} 
T(a - \delta_{3,1,1}, b - \delta_{3,1,2}) + T(a - \delta_{3,2,1}, b - \delta_{3,2,2}) \\
T(a - \delta_{4,1,1}, b - \delta_{4,1,2}) + T(a - \delta_{4,2,1}, b - \delta_{4,2,2}) \\
\ldots & 
\end{cases} \]

Figure 3.6: An example recurrence for analysis by compound measure
3.3 Standard Weight-based Measures

The most straightforward, and most fundamental, of the non-classical measures is a measure \( f(F) \) which is a linear function of a number of attributes. For instance, if the degrees of the variables of \( F \) are believed to be important to the running time of a certain algorithm, then one could analyse this algorithm in terms of a measure

\[
f(F) = \sum_{i<d} w_i n_i(F) + n_{\geq d}(F)
\]

for some max degree \( d \), where \( n_i \) is the number of \( i \)-variables, and \( w_i \) is the weight of an \( i \)-variable. As long as a few basic properties hold—say \( w_1 = 0 < w_2 < \ldots < w_{d-1} < 1 \)—these weights can be set to any combination of values, according to the nature of the algorithm. With this kind of a measure, assigning \( a = 1 \) for some variable \( a \) would decrease \( f(F) \) by \( w_d(x) \) for every removed variable \( x \) plus \( w_d(y) - w_{d(y) - 1} \) for every variable \( y \) that shares a clause with the literal \( a \), where the latter part is a kind of gain that is not visible in the measure \( n(F) \). Now, every branching number depends on the specific weights \( w_i \), and since \( w_i \leq 1 \) the running time will be in \( O^* \left( c_w^f(f(F)) \right) \subseteq O^* \left( c_w^n \right) \)

where \( c_w \) is the highest branching number that occurs in the algorithm when analysed using the weights vector \( w \).

As previously mentioned, every legal \( w \) gives us some limit \( O^* \left( c_w^n \right) \), and through the work of Eppstein [31], we can find the best possible vector \( w \) for a recurrence such as that in Figure 3.3 in a reasonable amount of time, and we know that the resulting bound is tight with respect to the model. Let us give the notation and framework used.

A problem, in this context, has an integer dimension \( d \), and the recurrence is defined as

\[
F(x) = \max_i \sum_j F(x - \delta_{i,j})
\]

where \( x \) and \( \delta_{i,j} \) are in \( \mathbb{Z}^d \), \( i \) ranges over the different possible branchings, and \( j \) ranges over the branches of a branching (so that \( \delta_{i,j} \) is the reduction in the problem instance in branch \( j \) of branching \( i \)). Base
3.3. Standard Weight-based Measures

cases \( F(0) = 1, \) and \( F(y) = 0 \) if no sequence of branches can reach the state 0 from \( y, \) are assumed. There is also a target vector \( t \in \mathbb{Z}^d \) which is used in the optimisation: it is the growth of \( f(n) = F(nt) \) that is estimated. In this context, Eppstein gives the following results.

**Lemma 1.** [Lm. 3.3 of [31]] Let \( w \in \mathbb{R}^d \) be such that, for each summand \( F(x - \delta_{i,j}) \) of the input recurrence, \( w \cdot \delta_{i,j} \) is positive, and let \( w \cdot t = 1. \) Then, \( f(n) \leq \max_{w \cdot x \leq n} F(x) \in \mathcal{O}(c_w). \)

In our terms, we would perhaps say that if \( f(x) = w \cdot x \) is a well-behaved measure for the recurrences, and if the maximum branching number using this measure is \( c_w, \) then the running time is in \( \mathcal{O}^*(\frac{f(x)}{c_w}). \)

**Theorem 2.** [Th. 6.1 of [31]] \( f(n) = F(nt) \in \Omega(c^n n^{(1-d)/2}) \) where \( c = \min_w c_w. \)

Eppstein’s paper also provides a local search procedure to find an optimal set of weights \( w. \) Tests indicate that this search procedure converges quickly, and that the problem is rather in the number of cases. For instance, if a measure like \( f(F) = \sum_{i<d} w_i n_i(F) + n_d(F) \) above is used, and the algorithm contains a case where we branch on a variable \( x, \) then we most likely have to enumerate one case in the recurrence for each combination of \( d(x) \) and \( d(y) \) for \( y \in N(x). \) With this measure, we usually get on the order of hundreds of cases or less, but if we want the weight of \( x \) to depend on more properties of \( x, \) then the number of cases explodes. A weight based on both \( d^+(x) \) and \( d^-(x) \) could possibly be managed, but if we want the weight of \( x \) to depend in some manner on the entire neighbourhood \( N(x), \) then the number of cases is likely to be unmanageable. The number of cases can be reduced somewhat by making analytical observations based on the behaviour of branching numbers, and further by making correct assumptions on the optimal weights \( w_i \) (such as the assumption that \( \Delta w_i = w_i - w_{i-1} \) decreases by increasing \( i; \) one could also limit the search to enforce this property, at the risk of producing a lower-quality bound), but it is hard to get away from the fundamental explosive behaviour. On the other hand, it is perhaps likely that the
improvement one would get from introducing more kinds of weights is gradually smaller.

Direct applications of Eppstein’s method seem mostly limited to the work done by Fomin et al. under the name of “measure and conquer”, see e.g. [36–39]. Other uses of weighted multi-parameter complexity measures have occurred, usually with two parameters and thus a single weight; examples include $3\text{SAT}$ papers by Zhang [87] and Kullmann [56], and the $(3,2)$-CSP (constraint programming) result of Beigel and Eppstein [3].

One restriction of this method in its pure form is that it is not possible to introduce restrictions on when branchings can be used. Any line of the recurrence definition can be used for any point $x$. The next two sections present two ways to use such restrictions in an analysis.

### 3.4 Finite Global States Modelling

One variation of weight-based measure that is used in this thesis uses the concept of a finite number, say $s$, of global states that affect which branchings that are possible. Assume that any instance $F$ is in exactly one of the states $S_1$ through $S_s$, and say that every branching is possible for only one of these states (an assumption that is made without loss of generality, naturally), and that for each branch of this branching, the formula ends up in a known new state. Let $S(F)$ be the state that $F$ is in (i.e., if $F$ is in state $S_k$, then $S(F) = k$). One way to model this in a measure is

$$f(F) = n(F) - \Psi(S(F)),$$

where $\Psi(k) \geq 0$ is a constant-sized perturbation that is applied to $n(F)$ depending on the state of $F$. Note that the numbers assigned to the states are arbitrary; in this thesis, the states will have a natural numbering that we will follow, but this is not necessary. The function $\Psi(k)$ is essentially a set of $s$ constants, one for each state.

A branch from state $S_1$ to state $S_3$, removing two variables, will now correspond to a reduction of $\Delta f = 2 + \Psi(3) - \Psi(1)$. The idea
is that if the branchings of state $S_3$ are better than those of state $S_1$, then $\Psi(3) > \Psi(1)$ and the state transition is counted as an extra bonus in $\Delta f$. Conversely, the reverse transition would be counted as a penalty, but by assumption, the base branching in terms of $\Delta n$ will be better when starting from state $S_3$. For some set of values $\Psi(k)$, these effects will balance out and we will have an upper bound of $O^*(c^n)$ for some $c$ depending on $\Psi(k)$.

We can clearly not use $\Psi(k)$ as a single parameter in Eppstein’s framework, as this would enforce $\Psi(k) = w_k$ which is an undesired property (and even nonsensical if the states are unordered), but we can model it if we unroll $\Psi(k)$ into $s$ weights $w_1$ through $w_s$. Under this model, we essentially let each branching that is valid for state $S_i$ have an entry of $-1$ in column $i$ of each branch, and each branch has an entry of $+1$ in some column $j$, indicating that that branch represents a transition $S_i \rightarrow S_j$. In addition to this, each branch has some entry in column zero representing the actual loss of variables (or whatever else is used as a main measure). At least this is the idea; we have to make a minor adjustment to be able to formulate a target vector.

More precisely, single out some state to be the starting state $S_0$. A transition $S_0 \rightarrow S_i$ for $i \neq 0$ is modelled as $+1$ in column $i$; a transition $S_i \rightarrow S_0$ for $i \neq 0$ is modelled as $-1$ in column $i$; a transition $S_i \rightarrow S_j$ is otherwise modelled as before, and any branch that leaves the state unchanged has zero in each column $1, \ldots, s-1$. Figure 3.7 illustrates these changes, starting from Figure 3.4 with the state 1 as starting state. In this setting, a target vector $t$ with $t_0 = 1$ and $t_i = 0$ for $i > 0$ can be used. We also no longer require that $w_i > 0$, as the state weights $w_1, \ldots, w_{s-1}$ are now relative to $w_0 = 0$. However, any constant change in all weights leaves the $O^*(c^n)$ bound unchanged.

The setup does not enforce the exact rules of the state transitions, since transitions can be taken regardless of the value of the parameters $x_1, \ldots, x_{s-1}$ (where in the original model, transitions from state $S_i$ can only be taken if $x_i = 1$ and $x_j = 0$ for $j > 0, j \neq i$), but since we use a target vector with all state-related variables set to 0, it still holds that the total sum of state changes, for every path down the branching
3. Measures of Complexity

\[ T(a, b, s_2, s_3) = \max \begin{cases} 
T(a - \delta_{1,1,1}, b - \delta_{1,1,2}, s_2, s_3) + \\
+ T(a - \delta_{1,2,1}, b - \delta_{1,2,2}, s_2 + 1, s_3) \\
T(a - \delta_{2,1,1}, b - \delta_{2,1,2}, s_2 + 1, s_3) + \\
+ T(a - \delta_{2,2,1}, b - \delta_{2,2,2}, s_2, s_3 + 1) \\
T(a - \delta_{3,1,1}, b - \delta_{3,1,2}, s_2 - 1, s_3) + \\
+ T(a - \delta_{3,2,1}, b - \delta_{3,2,2}, s_2, s_3) \\
T(a - \delta_{4,1,1}, b - \delta_{4,1,2}, s_2 - 1, s_3 + 1) + \\
+ T(a - \delta_{4,2,1}, b - \delta_{4,2,2}s_2 - 1, s_3) \\
\ldots 
\end{cases} \]

Figure 3.7: The branchings of Figure 3.4 with unrolled state tree, is zero (e.g. if a state is entered twice, it must also be left twice). We see in the next lemma that the likeness is strong enough to give an upper bound that is tight within a polynomial factor for the original model, in the case when a single non-state attribute is used.

Lemma 3. For a set of state-based recurrences, which are described in terms of a single main measure \( n(F) \) and have a connected state space, if \( c = \min_w c_w \), then \( O^*(c^{n(F)}) \) is both valid and tight as a bound on the size of the recurrence.

Proof. The bound is valid since every branching has the same branching number in both models. To prove tightness, we will show that a branching tree of the appropriate size can be constructed using only applicable worst-case branchings. Let \( f(F) \) be the measure that is constructed.

Consider the set of branchings with branching number \( c \) for some optimal weights \( w \). These branchings will divide the states into states that are both entered and left by some branches, and optionally states that are either never entered, never left, or not used at all by these worst-case branchings.

Only those branchings that move between states of the first kind
need to be considered. If some worst-case branching has a branch that enters some state that no worst-case branching leaves, then the weight of this state can be increased, and the branching will no longer be worst-case. Likewise, if some worst-case branching leaves a state that is not entered by any branch of a worst-case branching, then the weight of this state can be decreased, and the branching will no longer be worst-case.

Therefore, as the weights are assumed to be optimal, there must remain some set of worst-case branchings that move only between states of the first kind, so that for any state we reach through a branch of one of these worst-case branchings, one of these branchings will be applicable. Call these the active branchings, and the states involved the active states.

Whatever state we start from, it is possible by assumption to reach some active state through a constant number of branches, and therefore the size of the instance will now differ from the size of the input instance only by a constant. Once this state has been reached, apply any applicable active branching, then recursively apply any applicable active branching for every created subproblem, as long as the instance size is higher than some constant. This will create a subtree $T'$ which acts as a branching tree where every node has a branching number of exactly $c$, when measured by the measure $f$, and where the difference in measure from the root of $T'$ to any leaf of $T'$ is within a constant of the size of the input measure. As mentioned earlier, this is a guarantee that this subtree has $c^{f(F)-k}$ leaves, for a constant $k$, which brings the total size of the tree to within a polynomial factor of $c^{f(F)}$.

In this thesis, when the model is used in Chapters 6 and 8, the state is the number of 2-clauses in a 3SAT formula, which is clearly a numerical state, and the obvious starting state is when a formula is 3-uniform (which admits worse branchings in terms of $\Delta n$ than when 2-clauses exist). Still, the same observations as above hold true; when modelling through different weights $w_i$ rather than using a uniform 2-clause-cost $w$, then transitions leaving a state with 3
short clauses must be used as many times as the state is entered (rather than, say, entering it twice and then using a branching that removes 6 short clauses). The top state $S_s$ represents “at least $s$ short clauses”, which allows us to construct branchings without having to make assumptions about maximum degree: removing $s+1$ or $2s$ short clauses with a single assignment is no worse than removing $s$ short clauses.

This approach of analysis has been used in fixed-parameter tractable algorithms for 3-Hitting Set \[33, 64\] and in an algorithm for \#3SAT \[59\], though none of these papers have used the approach of state weights; Niedermeier and Rossmanith \[64\] performed an analysis with what is essentially the 2-state version of this approach (that is, “2-clauses exist” and “2-clauses do not exist”), while Fernau \[33\] and Littman, Pitassi, and Impagliazzo \[59\] did model the behaviour of their algorithms in terms of the different numbers of short clauses as well, but performed the calculation of a worst-case branching number by different methods.

The idea of perturbing the measure depending on some state of the instance also appears in a paper on parameterised Vertex Cover by Chen, Kanj, and Xia (which exists as a conference publication with omitted proof \[12\], and as a technical report with full proof \[11\]) where the proof is given through a single proof by induction. In their paper, the method is given the name of “local amortised analysis”. However, the first occurrence that we are aware of was in our article on 3-Hitting Set \[82\].

3.5 Compound Measures

In this section, we present another way to introduce restrictions on the applicability of branchings. Instead of letting the state which determines our applicable set of branchings be a direct attribute of our measure, we consider states that are implicit in the combination of values that the modelled attributes have, such as when the average degree of an instance determines which possible branchings exist. (At the very least, when the average degree is higher than $d$, a variable of
degree at least $d + 1$ exists, though we can sometimes find stronger connections than that; Lemma 87 shows such a connection). By using a compound measure we can model the effect of such an implicit state on the total running time of the algorithm, by letting the exact parameters of the measure vary along with the behaviour of the algorithm.

In particular, suppose that the algorithm is fast for different reasons depending on the state—that is, that the strongest bound on the running time varies depending on the state. For example, it is often the case that a maximum degree (or maximum clause lengths) of 2 implies the instance can be solved in polynomial time. Examples from this thesis include 2SAT (e.g. SAT with a maximum clause length of 2), and SAT, XSAT, and 2#SAT_w for a maximum degree of 2. When using the average degree as a parameter, this might mean that cases with a low average degree are fast because they reduce to this polynomial base case quickly, even though the branchings in terms of $n(F)$ are poor, while cases with a high average degree are fast because many variables are removed in each branching. An immediate way to use this, not using compound measures, is to refer to a separate analysis for those cases where $n(F)$ is not the best measure, and ignore those cases in the $n(F)$-based analysis. For instance, this is done in the conference version of the #2SAT paper of Dahllöf, Jonsson and Wahlström [14]. Essentially, in the terms of this thesis, an algorithm is analysed in that paper in terms of $l(F)$, giving a bound in the form of $O^*\left(\frac{l(F)}{c_l}\right)$, which is then used as a bound of $O^*\left(\frac{d_n(F)}{c_l}\right)$ for the case of $d(F) \leq d$, while the guarantee $d(F) > d$ is used to get a better branching number $c_n$ for the remaining cases (the final bound becomes $O^*\left(c^p\right)$ where $c = \max(c_d, c_n)$). Yet, this bound may not be tight. The bound of $O^*\left(\frac{d_n(F)}{c_l}\right)$ is hardly tight if $d$ grows bigger (since higher degrees of the branching variable admits better branching numbers), while $c_n$ for degree 4 may be unnecessarily

---

3Of course, comparing different bounds requires a common base of comparison; we will assume that the base of comparison is $n(F)$, so that a bound of $c^f(F)$ is converted into a bound of $c^f_n(F)$ before comparison.
high. Suppose that we have a worst-case branching number $c_3$ when $d(F) = 3$, and another branching number $c_4 < c_3$ when $d(F) = 4$, both numbers analysed in terms of $l(F)$, and we want a good bound when $d(F) = 4$. The bound $O^*(c_3^{4n(F)})$ is not tight, because when $l = 4n$ the branching tree contains a large number of $c_4$ or better local branching numbers, while $O^*(c_4^{4n(F)})$ is too optimistic, since the tree may contain branching numbers of $c_3$ as well. With a compound measures-based approach we divide the problem space into sections, but before we give the technical details on this, let us see how our example problem can be managed, to show how the principles work.

We will illustrate two such principles: using the distance to an easier case as a parameter of the analysis, and performing a smooth transition between bounds of different kinds. In our case, we would want to perform a transition between $l(F)$ for low degrees, and $n(F)$ for high degrees. Let $r(F) = l(F) - 3n(F)$; this is the distance to the case $d(F) = 3$. Since we are making a transition towards $n(F)$, the two components of our analysis will be $n(F)$ and $r(F)$. When $r(F) = 0$, we will be forced to use the bound $O^*(c_3^{3n(F)})$, but in the general case, we can derive a bound of $O^*(c_3^{3n(F)+wr(F)})$ for some $w$; let $f(F) = 3n(F) + wr(F)$. If the branching number analysed in terms of $n$ for degree 4 is already at most $c_3^3$, then we do not need the second component and can set $w = 0$ for a bound entirely in terms of $n$; on the other hand, with $w = 1$ the measure reduces to $f(F) = l(F)$ and no progress at all has been made towards $n(F)$. For intermediate values, we get a mixture. We set $w$ to the lowest number such that the branching number for degree 4, in terms of $f(F) = 3n(F) + wr(F)$, is $c_3$, and our final bound for $d(F) = 4$ will be $O^*(c_3^{(3+w)n(F)})$, which can be used as the starting point for another iteration of this process.

More fine-grained divisions of this sort are of course also possible. Assume that we are analysing our instances $F$ in terms of a set of attributes $h_1(F), \ldots, h_d(F)$. We model the applicability of a case as depending on the relative values of these $h_i$ (as in the relative values of $l(F)$ and $n(F)$, or of $n_i(F)$, in the previous example). For instance,
there may be a bad case that occurs when, say, \( d(F) = 3 \) and every neighbour of every 3-variable is a 2-variable, which is only possible when \( 3n_3(F) \leq 2n_2(F) \), and when this is not true, we may be guaranteed that an easier case will appear. If \( h_i \) measures the number of variables of each degree, then this condition on applicability can be directly encoded as a ratio of some \( h_i \). Let the space of possible parameters\(^4\) be \( \mathbb{Z}^d \) and let \( S: \mathbb{Z}^d \rightarrow \mathbb{N} \) be a function that divides the space into sections, according to the applicability constraints on our branching cases (for instance, the region where \( n_3 \leq 2n_2/3 \) may be one section). A compound measure is a piecewise linear function on \( \mathbb{Z}^d \), where each section (each linear piece, as it were) corresponds to a weight-based measure, optimised for those branching cases that can apply in this particular section of the space. In order to be able to easily find the worst-case behaviour, we require that two constraints apply to the linear functions: the compound function must be continuous and concave. Note that these sections are qualitatively different from the states of the previous section; the model of finite global states does not help in the cases given in this section, since we can not look at a single branching and get any information about e.g. the relative density of \( n_2 \) in the resulting instances.

Let us give the definitions for the generic case. Assume as stated that the parameters we are using in some analysis are \( h_1(F), \ldots, h_d(F) \) for each instance \( F \). Let \( S(F) = S(h_1(F), \ldots, h_d(F)) \in [0, t] \) decide the section of an instance \( F \). A section of 0 for an instance \( F \) (i.e. \( S(F) = 0 \)) is only allowed if \( F \) is not a fully reduced instance. Otherwise, we have the following conditions:

\[
\begin{align*}
    f(F) &= f(h_1(F), \ldots, h_d(F)) \\
    f(x_1, \ldots, x_d) &= f_i(x_1, \ldots, x_d) \text{ if } S(x_1, \ldots, x_d) = i \\
    f_i(x_1, \ldots, x_d) &= w_{i,1}x_1 + \ldots + w_{i,d}x_d \\
    f_i(x_1, \ldots, x_d) &\geq f(x_1, \ldots, x_d) \text{ if } S(x_1, \ldots, x_d) > 0
\end{align*}
\]

\(^4\)Non-integer attributes \( h_i \) might be possible, but most natural attributes seem to be integers.
Due to (3.4), if \( S(F) = i \), then we have

\[
\Delta f = f(F) - f(F') \geq \Delta f_i = f_i(F) - f_i(F')
\]

regardless of \( S(F') \), which allows us to find the worst-case branchings within each section without worrying about what section the subinstances \( F' \) will be in. As before, if the worst-case branching number (taken over all sections) is \( c \), then the running time of our algorithm will be in \( O^*(c^{f(F)}) \).

The constraint that \( f \) is continuous follows from (3.3) and (3.4), but it is worth some attention in its own right, as it can be quite restrictive. In the general case, if two sections \( i \) and \( j \) touch along a border, then the continuity requirement translates into the requirement that for every point \( X \) on this border, \( f_i(X) = f_j(X) \). If \( f_i \) has been fixed, then this may mean that we have only one degree of freedom in choosing \( f_j \). On the other hand, deviating from the requirement that \( f_j \) be linear may introduce difficulties in estimating \( \Delta f_j \) for our branchings. We have no general tightness results for this method, but we shall see in Chapter 7 that the method does give better results for some cases than what is attainable through only standard weight-based modelling.

### 3.5.1 Analysis by Average Degree

In the work in this thesis, the division into sections of the parameter space follows the average degree of the instance. More precisely, there is a number \( k_0 \) such that \( d(v) < k_0 \) implies that \( v \) can be removed by some reduction and such that \( d(F) = k_0 \) implies that \( F \) can be solved in polynomial time — in other words, if the average degree of \( F \) is at most \( k_0 \), then \( S(F) = 0 \) and \( f(F) = f_0(F) = 0 \) — and for every branching case there may be a maximum average degree above which a better case is known to apply. A simple example of such an effect is when we branch on a variable of maximum degree — average degree \( 3.01 \) would guarantee that \( d(F) > 3 \) — but more detailed observations can be made for other strategies for picking branching variables. We divide into sections according to the worst of these
3.5. Compound Measures

cases: we get a sequence of numbers \( k_i \), such that the hardest case that can appear when the average degree is higher than \( k_{i-1} \) will only appear when the average degree is at most \( k_i \), and let \( S(F) = i \) when \( k_{i-1} < l(F)/n(F) \leq k_i \). There are two different ways to define \( f(F) \) from this: we can either let the attributes be \( l(F) \) and \( n(F) \), or we can use \( n_i(F) \) up to some maximum degree \( i \) for attributes. Let us first consider \( f(F) = f(l(F), n(F)) \) with linear functions \( f_i(l, n) = w_i,0 \cdot l + w_i,1 \cdot n \). The continuity constraint translates into \( f_i(k_i n, n) = f_{i+1}(k_i n, n) \) for every \( k_i, \ i > 0 \), i.e.
\[
w_{i+1,1} = w_{i,1} + k_i (w_{i,0} - w_{i+1,0}).
\]
Then, condition (3.4) follows from \( f_i(k_i n + x, n) \geq f_{i+1}(k_i n + x, n) \) when \( x > 0 \), i.e.
\[
w_{i+1,0} \leq w_{i,0}.
\]
Note the pattern of dependence: \( w_{i+1,0} \) can be freely set between 0 and \( w_{i,0} \), while \( w_{i+1,1} \) can be calculated from the values \( w_{1,0} \) through \( w_{i+1,0} \) without using any value of \( w_{j,1} \). Let us give the explicit expansion of the definition of \( w_{i,1} \):
\[
\chi_i = \sum_{j=1}^{i} (k_j - k_{j-1}) w_{j,0} \quad \quad (3.5)
\]
\[
w_{i,1} = \chi_{i-1} - w_{i,0} k_{i-1} \quad \quad (3.6)
\]
\[
f_i(l, n) = (l - k_{i-1} n) w_{i,0} + \chi_{i-1} n \quad \quad (3.7)
\]
Note that \( f(k_i n, n) = \chi_i n \), so that the running time of a reduced formula with average degree no more than \( k_i \) is in \( \mathcal{O}^* (c^{\chi_i n}) \) where \( c \) is the maximum branching number of the algorithm (taken over all sections). Eventually, for some \( i \), we may have \( w_{i+1,0} = 0 \) and the worst-case running time for any fully reduced formula will be in \( \mathcal{O}^* (c^{\chi_i n}) \) as a higher degree no longer makes the problem more difficult.

Consider again the case of a worst-case branching appearing when \( d(F) = 3 \) and \( n_3 \leq 2n_2/3 \), and suppose we have \( k_0 = 2 \). Since
3. Measures of Complexity

\[ f_0(F) = 0, \text{ we get } w_{1,1} = -2w_{1,0}, \text{ and } w_{1,0} \text{ is unrestricted since (3.4) does not apply when } S(F) = 0; \text{ each variable } v \text{ increases } f_1(F) \text{ by } (d(v) - 2)w_{1,0}, \text{ which is non-negative if } F \text{ is fully reduced. When } d(F) = 3, \text{ this would mean that } f_1(F) = w_{1,0}n_3(F), \text{ and that 2-variables are ignored; when } d(F) > 3 \text{ we get } f_1(F) = w_{1,0}(n_3(F) + 2n_4(F) + \ldots). \text{ Assuming that no harder cases appear when } d(F) > 3, \text{ this part would behave just like a case analysis in terms of } n_3(F), \text{ producing a bound along the lines of } O\left(p(n) \cdot c^{n_3 + 2n_4 + \ldots}\right). \text{ Under the assumption that all cases only have a highest associated average degree, so that all cases could occur at an average degree close to } k_0, \text{ this bound would be valid on its own, but it would likely be of low quality.}

\[
\]

We can set \( w_{1,0} \) to any value we wish, and the base \( c \) of the running time will scale so that the running time \( O^* \left(c^{(k_1-2)w_{1,0}n}\right) \) for a reduced formula with average degree at most \( k_1 \) is invariant, given a value for \( k_1 \); we may choose to pick either \( c = 2 \) or \( w_{1,0} = 1 \), for convenience. However, when the average degree exceeds 2.4, the situation \( d(F) = 3 \) and \( 3n_3 = 2n_2 \) can no longer occur, which means that the hardest branching case is no longer applicable. We set \( k_1 = 2.4 \), and can now set \( w_{2,0} < w_{1,0} \), since all cases that apply in section 2 are easier than the worst case of section 1. We pick the value of \( w_{2,0} \) so that the branching number for the hardest case that appears in section 2 is equal to \( c \). At some average degree \( k_2 \), the worst cases of section 2 no longer apply, and we start section 3 with \( w_{3,0} \) still lower, and still with a worst-case branching number of \( c \). We see that we are performing a kind of progression by average degree \( k_0 < k_1 < k_2 < \ldots \) (until we reach some hard case which does not have an associated maximum average degree). The changes of section \( k_i \) are useful only when placed at the points at which some hard case stops being applicable; putting another change of sections between 2 and 2.4 would have no effect, since we would keep the same worst case and be forced to set the same weights. Note that in every step of this process, there is only ever one variable that can vary (namely \( w_{i,0} \)) to adjust the branching number to \( c \), so the “optimisation” is trivial.
3.5.2 Multiple Attributes Analysis

If we want to perform the analysis in terms of a larger number of attributes, say using \(n_i(F)\) as considered attributes instead of \(l(F)\) and \(n(F)\), then the optimisation is no longer trivial, so we need some way to automate it. We will show how to do this, but first, we show how to implement the constraints on the weights with these attributes. Each component measure is

\[
f_i(F) = \sum_j w_{i,j} n_j(F)
\]

where \(w_{i,j}\) is the weight of a \(j\)-variable in the component measure used in section \(i\). By the linearity of all \(f_i\), the continuity of \(f\), and condition (3.4), we can write

\[
f_{i+1}(F) = f_i(F) - \alpha_i (l(F) - p_i n(F))
\]

for \(i \geq 1\), where \(p_i\) is the average degree at which the switch from section \(i\) to section \(i + 1\) occurs, and \(\alpha \geq 0\) is a weight to optimise. Note again the occurrence of a “distance to the next easier case” in \(l(F) - p_i n(F)\). Expanding, we get

\[
\sum_j w_{i+1,j} n_j(F) = \sum_j (w_{i,j} - \alpha_i + \alpha_i p_i) n_j(F)
\]

as a definition of \(w_{i+1,j}\) from \(w_{i,j}\), \(p_i\), and \(\alpha_i\). On the other hand, \(f_1(F)\) can be given any combination of weights as long as it is a well-behaved measure. It is easy to see that this obeys the definitions and conditions given. Note that \(f_{i+1}\) only has one degree of freedom since the boundary between \(f_i\) and \(f_{i+1}\) has \(d - 1\) dimensions (where \(d\) is the number of weights in the component measures). As a result, after the weights of \(f_1\) have been determined, the best values of \(\alpha\) are easily found as well (each \(\alpha\) is set so that the worst-case branching of each section has the same branching number).

For this reason, there is an issue with the optimisation of weights: we are mostly interested in the worst-case bound given for the highest considered average degree — most likely, the value of \(w_{t,d}\) — but the
component measure that can be optimised freely is the lowest-degree measure. A direct local search optimisation for $f_1$ does not optimise for the correct goal. Instead, we can add the values of $\alpha_i$ as weights to be optimised, as follows.

Let the weights of the optimisation be $w_{t,j}$ and $\alpha_i$, with $w_{t,d}$ being the only non-zero component of the target vector. By reordering (3.8), we can calculate the value of any $w_{i,j}$ from this data:

$$w_{t-k,j} = w_{t,j} + \sum_{i=t-k}^{t-1} (j - p_i)\alpha_i.$$ 

Then, for every section of the problem, add one line for every branching, using this definition of $w_{i,j}$ when doing so. Performing the local search optimisation on this will optimise for the correct target. Though this is not known to guarantee a bound which is tight for the average degree-based model, it will guarantee that the combination of component measures achieves the best bound possible within the method.
3.5. Compound Measures
Part II

Decision Problems
Chapter 4

Satisfiability for Sparse Formulae

The problem we attack in this chapter is the SAT problem, but from a different perspective than the most common one. Instead of restricting the lengths of the clauses, we ask the following question: given that we have a formula $F$ where the average degree $l(F)/n(F)$ of a variable is limited, but there are otherwise no restrictions on the lengths or disposition of the clauses, how can we decide the satisfiability of $F$ as fast as possible?

Of the previous research devoted to satisfiability problems, two results in particular are applicable. Oliver Kullmann gave an algorithm with a running time in $O(3^{n/9}) \subset O(1.1299^n)$ for the specific case that $d(F) \leq 3$ [58], and Edward Hirsch gave an algorithm where the running time is bounded in terms of $l(F)$ by $O(1.0740^{l(F)})$, which translates into a bound of $O(1.0740^{k-n(F)})$ when the average degree is $k$, which beats $O(2^n)$ for $k \leq 9.7$. In this chapter, we give an algorithm called SparseSAT for this problem, and use an analysis by average degree to give a bound that approaches $O^*(2^n)$ but never exceeds it. A summary of the bounds for $k \leq 10$ is given in Table 4.1, and a comparison of the new and old bounds is in Figure 4.1 on page 87. We also show an upper bound of $O^*(2^{0.0926l(F)}) \subset O^*(1.0663^{l(F)})$ for the algorithm.
Running time $\leq 2$ Reductions apply
3 $O^*(1.1279^n)$
4 $O^*(1.2721^n)$
5 $O^*(1.3783^n)$
6 $O^*(1.4548^n)$
7 $O^*(1.5152^n)$
8 $O^*(1.5641^n)$
9 $O^*(1.6043^n)$
10 $O^*(1.6381^n)$

Table 4.1: Bounds on the running time of SparseSAT depending on average degree

Regarding the asymptotics of the upper bound itself as $k$ increases, Dantsin, Hirsch and Wolpert [17] have a deterministic algorithm with a bound of $O\left(2^n(1-1/\alpha)\right)$ where $\alpha = \ln(m(F)/n(F)) + O\left(\ln \ln m(F)\right)$, which does not provide concrete limits for any value of $k$, but which (disregarding the $O\left(\ln \ln m(F)\right)$ factor) gives a stronger bound with respect to an increasing $k$ than ours.

In the algorithm, we will use both standard resolution and an inverse to it that we call backward resolution. If a formula $F$ contains two clauses $C_1 = (C \lor D)$, $C_2 = (C \lor E)$ where $D$ and $E$ share no literals, then $DP_{C_1,C_2}$ is the formula where $C_1$ and $C_2$ have been replaced by clauses $(\overline{a} \lor C), (a \lor D), (a \lor E)$ for a fresh variable $a$. Backward resolution is the inverse to resolution in that applying resolution to $a$ in this new formula recreates the original $F$.

Recall that in this chapter, we allow clauses to contain multiple copies of a literal; since the algorithm uses both resolution and replacement, such clauses can be created if one is not careful, so we felt it most clear if they are handled explicitly.

This chapter is divided into Section 4.1 covering the algorithm that will be used, Section 4.2 giving an upper bound on the running
time when $l(F)/n(F) \leq 4$, and Section 4.3 giving an upper bound in the general case.

### 4.1 The Algorithm

The algorithm that we will deal with is shown below as Algorithm 6. When discussing it, we will refer to cases 1–5 as *simple reductions*, since the effect of these cases is only to remove literals or variables from $F$, without adding any new literals or variables. Cases 6 and 7 are referred to as non-simple reductions. We will say that a formula $F'$ is the *step $k$-reduced version* of $F$ if $F'$ is the result of applying the algorithm until none of the cases 0–$k$ applies. Fully reduced is in this case a synonym to step 7-reduced.

*Standardising* a CNF formula $F$ refers to applying the following reductions as far as possible:

1. **Subsumption**: if there are two clauses $C, D$ in $F$, and if every literal in $C$ also occurs in $D$, then $D$ is *subsumed* by $C$. Remove $D$ from $F$.

2. **Trivial or duplicate clauses**: if $F$ contains several copies of some clause $C$, then $C$ is a duplicate clause. If there is a clause $C$ in $F$ such that both literals $v$ and $\bar{v}$ occur in $C$ for some variable $v$, then $C$ is a trivial clause. In both cases, remove $C$ from $F$.

3. **Multi-occurring literals**: if there is a clause $C$ in $F$ where some literal $l$ occurs more than once, then remove all but one of the occurrences of $l$ from $C$.

A formula $F$ where none of these reductions apply is said to be in *standard form*.

Essentially, the simple reductions can always be applied, while resolution and backward resolution must be limited to cases when applying these reductions makes progress (i.e., leads to a simpler formula). Applying resolution will lead to a formula with fewer variables, while in the general case, the remaining variables will have more occurrences, possibly implying that the resulting formula is longer than
the original one. Since sparse formulae is the topic of the chapter, we see that the former probably constitutes positive progress, while the latter probably constitutes negative progress (as it makes the problem less sparse), and we have to decide how to balance these effects. The answer we use in this work is given in the next definition. The definitions for \( k \leq 4 \) follows from considering \( d(v) - 2 \) to be the fundamental difficulty (or weight) of a variable \( v \) for such sparse cases; the rest follows from the analysis, as variables are gradually given more similar weights.

**Definition 4.** Let \( F \) be a step 5-reduced CNF formula, and let \( F' \) be the step 5-reduced version of \( DP_x(F) \), for some variable \( x \) in \( F \). Let \( k = \lceil l(F)/n(F) \rceil \), \( \Delta l = l(F) - l(F') \) and \( \Delta n = n(F) - n(F') \). We say that resolution on \( x \) in \( F \) is *admissible* if

- \( k \leq 4 \) and \( \Delta l \geq 2\Delta n \), or
- if \( k = 5 \) and \( \Delta l \geq \Delta n \), or
- if \( k > 5 \) and \( \Delta l \geq 0 \).

For backward resolution, if there are two clauses \( C_1 = (C \lor D), C_2 = (C \lor E) \) in \( F \), then let \( F' \) be the step 5-reduced version of \( DP^{-1}_{C_1,C_2}(F) \). Backward resolution on \( C_1, C_2 \) is *admissible* if

- \( k \leq 4 \) and \( \Delta l > 2\Delta n \), or
- if \( k = 5 \) and \( \Delta l > \Delta n \).

Once the measure \( f(F) \) that is used for the analysis is defined, it will be clear that this definition guarantees that \( f(F) \geq f(F') \) when resolution is admissible, and that \( f(F) > f(F') \) when backward resolution is admissible.

The final case of the algorithm uses an algorithm for the constraint satisfaction problem known as \((3,2)\)-csp to solve the remaining problems (see [73] for a general description of constraint satisfaction problems). This is the constraint satisfaction problem where each variable can take 3 different values and the constraints are arbitrary binary constraints. For this purpose, we use an algorithm by Eppstein [30] (see also journal version in [3]), with the following bound:
Theorem 5. [Th. 3.1 of [30]] Eppstein’s algorithm can solve any (3, 2)-csp instance in time $O((4, 4, 5, 5)) \subset O(1.36443^n)$.

Now, we present the algorithm SparseSAT.

Algorithm 6. SparseSAT(F):

0. If $F = \emptyset$, then return 1. If $\emptyset \in F$, then return 0.

1. If $F$ is not in standard form, then standardise it and return SparseSAT($F$).

2. If there is some 1-clause $(l) \in F$, then return SparseSAT($F[l]$).

3. If there is a pure literal $l$ in $F$, then return SparseSAT($F[l]$).

4. A pair of variables co-occurs twice:
   a) If there is a 2-clause $(l_1 \lor l_2)$ and a clause $D = (l_1 \lor \overline{l_2} \lor C)$ in $F$ for some possibly empty $C$, then construct $F'$ from $F$ by deleting $\overline{l_2}$ from $D$.
   b) If there are 2-clauses $C_1 = (l_1 \lor l_2)$ and $C_2 = (\overline{l_1} \lor \overline{l_2})$, then create $F'$ from $F$ by replacing all occurrences of $l_2$ by $\overline{l_1}$ and all occurrences of $\overline{l_2}$ by $l_1$, and removing $C_1$ and $C_2$.

   Return SparseSAT($F'$).

5. If there is a variable $x$ in $F$ with at most one non-trivial resolvent, then return SparseSAT($\text{DP}_x(F)$).

6. If there is a variable $x$ in $F$ with $d(x) = 3$ such that resolution on $x$ is admissible then return SparseSAT($\text{DP}_x(F)$).

7. If there are two clauses $C_1 = (C \lor D), C_2 = (C \lor E)$ such that backward resolution on $C_1, C_2$ is admissible then return SparseSAT($\text{DP}_{C_1, C_2}^{-1}(F)$).

8. If $d(F) \geq 4$, then pick a variable $x$ of maximum degree. If some literal of $x$, assume $\overline{x}$, occurs only in a single clause $(\overline{x} \lor l_1 \lor \ldots \lor l_k)$, then return
    
    \[
    \text{SparseSAT}(F[x]) \lor \text{SparseSAT}(F[\{\overline{x}, \overline{l_1}, \ldots, \overline{l_k}\}])
    \]
4.1. The Algorithm

If both $x$ and $\bar{x}$ occur in at least two clauses, then return

$$\text{SparseSAT}(F[x]) \lor \text{SparseSAT}(F[\bar{x}])$$

9. If there is a 2-literal $l$ such that the step 5-reduced version of $F[l]$ has at most $n(F) - 6$ variables, then assume that $l$ occurs in a clause $C$ along with literals $l_1, \ldots, l_k$ and return

$$\text{SparseSAT}(F[l]) \lor \text{SparseSAT}(F[\{\bar{l}, \bar{l}_1, \ldots, \bar{l}_k\}])$$

10. If there is a clause $C = (\bar{v}_1 \lor \ldots \lor \bar{v}_k)$ that contains only 1-literals and $|C| \geq 4$, then return

$$\text{SparseSAT}(F - C + (\bar{v}_1 \lor \ldots \lor \bar{v}_{\lfloor k/2 \rfloor})) \lor$$

$$\text{SparseSAT}(F - C + (\bar{v}_{\lfloor k/2 \rfloor + 1} \lor \ldots \lor \bar{v}_k))$$

11. Let $a$ be a 2-literal (assumed to be positive) with a maximum number of neighbours. Let the clause that contains $\bar{a}$ be $(\bar{a} \lor \bar{b} \lor \bar{c})$. If the literal $a$ has at least three neighbours, then return

$$\text{SparseSAT}(F[a]) \lor \text{SparseSAT}(F[\{\bar{a}, b, c\}])$$

12. If no previous case applied, then the formula can be converted into a $(3, 2)$-csp instance with $n(F)/3$ variables, as described in Lemma 8. Perform this conversion, and apply Eppstein’s algorithm from [30] (see Theorem 5).

Algorithm ends.

We use two measures of complexity for this algorithm. In Section 4.2, where $l(F) \leq 4n(F)$ is guaranteed, we use $f_A(F) = l(F) - 2n(F) + s(F)$, where $s(F)$ is the number of singletons in $F$, which is equivalent to $f_A(F) = \sum_{v \in \text{Vars}(F)} \max(0, d(v) - 2)$. In Section 4.3, we use a compound measure $f_B(F)$ with component measures $f_i(l(F), n(F)) = a_i n(F) + b_i l(F)$ applying for an average degree of $i - 1$ to $i$. The values
of \( a_i \) are calculated as in Section 3.5.1, and the values of \( b_i \) are defined as follows:

\[ b_1 = b_2 = 0 \quad (4.1) \]
\[ \tau(4b_3, 8b_3) = 2 \quad (4.2) \]
\[ \tau(4b_4, 8b_4) = 2 \quad (4.3) \]
\[ \tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2 \quad (4.4) \]
\[ \tau(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k-3)b_k) = 2 \text{ for } k \geq 6 \quad (4.5) \]

While this does give us three separate trivial measures \( f_0 = f_1 = f_2 = 0 \), it is more mnemonic to have component \( i \) apply to the case of a guaranteed maximum degree of \( i \).

**Lemma 7.** The following hold for the parameters \( a_k, b_k \) and \( \chi_k \).

- \( b_3 = b_4 = (\log_2(\sqrt{5} + 1) - 1)/4 \) and \( a_3 = a_4 = -2b_3 \)
- \( b_5 = 2b_3/3 \) and \( a_5 = -b_5 \)
- For \( k \geq 4 \), \( a_k < a_{k+1} < 1 \), \( b_k > b_{k+1} > 0 \), \( \chi_k < \chi_{k+1} < 1 \), and \( a_k > 0 \) for \( k \geq 6 \).

**Proof.** The results for \( k \leq 5 \) can be derived directly from equations (4.2)–(4.4).

Let \( b_k, k \geq 6 \) be defined according to equation (4.5) and assume that \( b_k > 0 \). By the balance property of \( \tau \), as \( \tau(1, 1) = 2 \), we have \( \chi_{k-1} + 5b_k = \chi_k + 4b_k \leq 1 \), so \( \chi_k < 1 \), and thus \( b_{k+1} > 0 \). As \( b_k > 0 \) for \( 3 \leq k \leq 6 \), we find \( b_k > 0 \) for \( k \geq 3 \). The very same argument shows that \( \chi_k < 1 \) for all \( k \). It follows immediately that \( \chi_k = \sum b_k \) must be increasing.

Consider (4.5) for \( k = k' \) and \( k = k' + 1 \). If \( b_{k'+1} \geq b_{k'} \), then both parts of the \( \tau \) function of (4.5) would increase from \( k = k' \) to \( k = k' + 1 \), contradicting (4.5) for \( k = k' + 1 \). It holds for all \( k \geq 4 \) that \( b_{k+1} < b_k \).

The properties of \( a_k \) remain to be shown. As noted, we have \( a_k = \chi_{k-1} - (k-1)b_k \), proving immediately that \( a_k < 1 \). With \( a_{k+1} = a_k + (k-1)(b_k - b_{k+1}) \), we also see that \( a_k \) is increasing, and \( a_6 > 0 \) can be easily verified. \( \square \)
Note that $f_3$ and $f_4$ are scalings of $f_A$ with the $s(F)$ term omitted, that $f_3$ is a scaling of $l(F) - n(F)$, and that $a_i, b_i > 0$ for $i \geq 6$. Due to this, the application of an admissible resolution keeps $f_B$ non-increasing while the application of an admissible backward resolution will strictly reduce $f_B$ (the same holds for $f_A$, when $l(F) \leq 4n(F)$). The reason for using different measures $f_A$ and $f_B$ is that when $l(F) \leq 4n(F)$, a single variable can have a negative contribution to the total weight in $f_B$ but not in $f_A$, due to the $s(F)$ term, and having this property simplifies some things. We will now give lemmas, in turn, for the correctness of the conversion to a $(3,2)$-csp instance (Lemma 8), the correctness of one type of branching used in the algorithm (Lemma 9), and finally the correctness of the algorithm (Lemma 10).

**Lemma 8.** Given a 3-regular sat formula $F$ without pure variables, where all 2-literals occur only in 2-clauses and all 1-literals occur only in 3-clauses, there is a corresponding $(3,2)$-csp instance $I$, constructible in polynomial time and with $n(F)/3$ variables, that is satisfiable if and only if $F$ is satisfiable.

**Proof.** By Lemma 14.6 of [56], a formula $F$ with $c$ 3-clauses and otherwise only 2-clauses can be converted into an instance $I$ of $(3,2)$-csp with $c$ variables (by first creating one variable in $I$ for each clause in $F$, and then performing a reduction used by Eppstein in [30] to remove every variable with only two values). Since every variable of $F$ occurs in only one 3-clause, the resulting instance $I$ has $n(F)/3$ variables. \[\square\]

**Lemma 9.** Let $\bar{x}$ be a 1-literal in a formula $F$, and let the clause where $\bar{x}$ occurs be $C = (\bar{x} \lor l_1 \lor \ldots \lor l_d)$. Then either $F[x]$ is satisfiable, or $F[\bar{x}]$ and $F[\{\bar{x}, l_1, \ldots, l_d\}]$ are equi-satisfiable (i.e. either both are satisfiable, or neither).

**Proof.** Assume that $F[x]$ is unsatisfiable. If there is a satisfying assignment $A$ to $F$, then it must set $\bar{x}$ to true and changing the value of $x$ in $A$ must create an unsatisfied clause. The only possible such clause is $C$, which means that all other literals of $C$ must be false in $A$. \[\square\]
Lemma 10. The algorithm SparseSAT applied to a CNF formula $F$ correctly calculates the satisfiability of $F$.

Proof. Case 0 is correct by the definition of the problem, and cases 1–4 are easily checked. Cases 5–7 use resolution, and the correctness of this operation is proven in e.g. [20]. Furthermore, the reduction process will terminate, which we will prove using the measure $f_B(F)$. Assume that $F$ is step 3-reduced (this can be assumed, since cases 1–3 clearly terminate). The reduction in case 4a clearly keeps $f_B(F)$ non-increasing and reduces $l(F)$, and the reduction in case 4b either removes both $a$ and $b$, or produces a new variable with $d(a) + d(b) - 4$ occurrences, and $\max(0, d(a) + d(b) - 6) \leq d(a) - 2 + d(b) - 2$; the reduction in case 4b also reduces $l(F)$. Resolution keeps $f_B(F)$ non-increasing as noted, while either decreasing $l(F)$ (when $l(F) \leq 5n(F)$) or keeping $l(F)$ non-increasing while decreasing $n(F)$ (see Definition 4). Backward resolution decreases $f_B(F)$ strictly. This shows that no infinite chain of reductions is possible. Cases 8,9 and 11 either use a branching with two assignments $x$ and $\bar{x}$, which is obviously correct, or branchings that are correct by Lemma 9. Case 10 is correct, as any assignment that satisfies $C$ must satisfy at least one of the new clauses. In case 11, the length of the clause containing the 1-literal must be 3, as a 2-clause with a 1-literal $\bar{x}$ implies that resolution on $x$ is admissible (see Lemma 13). The correctness and completeness of case 12 given that cases 0–11 do not apply is proven in Lemma 22 in the next section, as this proof uses a number of other lemmas, that are best shown in the context of the algorithm analysis.

4.2 Average Degree up to Four

In this section we give the first part of the analysis of an upper bound on the running time of the algorithm SparseSAT, proving a time bound for the cases with $l(F) \leq 4n(F)$. Therefore, we assume throughout the section that $l(F) \leq 4n(F)$ holds. The main reason for splitting the analysis into two main parts is that, as stated, the
measure

\[ f_A(F) = l(F) - 2n(F) + s(F) = \sum_{v \in \text{Vars}(F)} \max(0, d(v) - 2) \]

which is used in this section, unlike \( f_3(F) \) or \( f_4(F) \) that are the corresponding components of the compound measure used in the next section, has the property that every variable in \( F \) contributes some non-negative amount to the total weight of \( F \).

Note that a variable \( v \) with \( d(v) \leq 2 \) will be removed in one of the simple reductions, meaning that \( f_A(F) \) assigns a weight of zero to a variable that can easily be removed, and other variables get a weight according to how far they are from being removable. We will see that this measure is correctly balanced for our purposes, in that it gives us one hard case for \( d(F) = 3 \) and one for \( d(F) = 4 \), with the same branching number.

In the next section, we perform an analysis by average degree of the algorithm, using a compound measure as outlined in the previous chapter, with the result from this section as a starting point. In this section, we are allowed to ignore all such concerns and only focus on our one linear measure \( f_A(F) \).

We will begin by proving that \( f_A(F) \) has the technical properties required of a measure, and then we will proceed with the case analysis, essentially progressing according to the cases of the algorithm.

**Lemma 11.** Let \( F \) be a \( \text{cnf} \) formula with \( l(F) \leq 4n(F) \), and let \( F' \) be the fully reduced version of \( F \). Then, \( f_A(F) \geq 0 \), \( f_A(F) = 0 \) implies that \( F' \) is trivial, and \( f_A(F') \leq f_A(F) \).

**Proof.** That \( f_A(F) \geq 0 \) is obvious from the previous presentation of \( f_A(F) \), as every variable contributes a non-negative amount to \( f_A(F) \). Likewise, \( f_A(F) = 0 \) if and only if every variable in \( F \) appears at most two times, in which case every variable in \( F \) will be removed. Finally, no reduction increases the value of \( f_A(F) \): the simple reductions add no new occurrences of variables, and when \( l(F) \leq 4n(F) \) applies, cases 6 and 7 are defined so that they never increase \( f_A(F) \). \( \square \)
4. Satisfiability for Sparse Formulae

We give one more technical result: the next lemma will allow us to more easily predict differences $f_A(F) - f_A(F')$ over a branch.

**Lemma 12.** Let $F$ be a fully reduced formula, $A$ an assignment to variables of $F$, and $F'$ the reduced version of $F[A]$. Further, let $F_0$ be the result of a sequence of applications of the reductions in cases 1–3 in any order to $F[A]$. If $F'$ contains no empty clause, then we have $f_A(F') \leq f_A(F_0)$.

**Proof.** The result can be shown by induction on the number of reductions applied. Remember that cases 1–3 only remove clauses and literals from $F$, and note that the only case of these that will ever remove the last occurrence of a literal from a clause without also removing the entire clause is case 2.

First, if some reduction is applicable on $F[A]$, then every clause and literal that would be removed by the application of this reduction will be removed by any sequence of applications of cases 1–3 ending in a step 3-reduced formula. This can be verified without any great difficulty (using the above observations and the fact that $F'$ contains no empty clause).

Secondly, assume that the induction hypothesis is true for every sequence of $k$ of these reductions acting on $F[A]$; that is, for any sequence of $k$ applications of cases 1–3 acting on $F[A]$, removing a set of clauses $C^*$ and a set of literals $L$, every possible sequence of such reductions ending in a step 3-reduced formula will remove at least these clauses and literals. It can likewise be verified that any extra clauses and literals that would be removed by the application of one further reduction will also be missing in any resulting step 3-reduced formula (again using that $F'$ contains no empty clause).

Thus, if $F_1$ is the true step 3-reduced version of $F[A]$, then for every variable $v$ that occurs in both $F_0$ and $F_1$, $d(v, F_0) \geq d(v, F_1)$, which gives us $f_A(F_0) \geq f_A(F_1)$, and clearly $f_A(F_1) \geq f_A(F')$. 

The rest of this section is divided into subsections as follows: Section 4.2.1 deals with the effects of cases 0–7, Section 4.2.2 analyses case 8, Section 4.2.3 covers the analysis of case 9, and finally Section 4.2.4 concludes the analysis.
4.2.1 Basic Structural Properties

This section contains some results regarding the basic structural properties that exist in a fully reduced formula. First we give a lemma that shows a sufficient condition for when resolution on a variable \(x\) is admissible.

**Lemma 13.** Let \(F\) be a step 5-reduced \(\text{CNF}\) formula, and \(x, d(x) = 3\), be a variable occurring in \(F\). If the following hold, then resolution on \(x\) is admissible:

1. Applying resolution to \(x\) increases the degree of at most \(c\) variables; and
2. applying resolution to \(x\), plus applying the reductions in cases 1–5 to the result, decreases the degree of at least \(c\) variables, including \(x\).

**Proof.** Since \(d(v, F) > 2\) for every variable in a step 5-reduced formula, every reduced or removed variable reduces \(f_A(F)\) by one point. Since \(d(x, F) = 3\), no variable can increase its degree or its contribution to \(f_A(F)\) by more than one in the resolution process. \(\square\)

Next, Lemmas 14 and 15 show the mentioned structural properties.

**Lemma 14.** If \(F\) is a 3-regular, fully reduced formula, and if \(C, D\) are two clauses in \(F\), then \(|\text{Vars}(C) \cap \text{Vars}(D)| \leq 2\). If in addition \(|C| = 2\), then \(|\text{Vars}(C) \cap \text{Vars}(D)| \leq 1\) and \(\text{Vars}(C) \not\subseteq \text{Vars}(D)\).

**Proof.** For the first part, note that some reduction applies both if \(l_1, l_2 \in C\) and \(l_1, l_2 \in D\), and if \(l_1, l_2 \in C\), \(\bar{l}_1, \bar{l}_2 \in D\). There is no way for \(C\) and \(D\) to share three variables without one of these cases occurring. For the second part, if \(C = (l_1 \lor l_2)\) and \(l_1, \bar{l}_2 \in D\), then case 4 applies and \(D\) is shortened. \(\square\)

**Lemma 15.** Let \(F\) be a step 5-reduced formula, and let \(a, b\) be \((2, 1)\)-variables in \(F\). If any of the following structures is present, then there exists an admissible resolution:
4. Satisfiability for Sparse Formulae

1. a 2-clause $C$ with $\bar{a} \in C$;
2. a 3-clause $C$ with $\bar{a}, b \in C$ and a clause $D$ with $a, b \in D$; or
3. a 3-clause $C$ with $\bar{a}, l \in C$, a clause $D$ with $a, b \in D$ and a 2-clause $(\bar{l} \lor b)$ for some literal $l$.

Proof. In the first two cases, we see immediately by Lemma 13 that resolution on $a$ is admissible. In the third case, we see that one resolvent is either a copy of an existing clause or will be shortened or removed in case 4 at the latest. In either case, $f_A(F)$ has increased by at most 1 in the resolution process, and at least one simple reduction which strictly decreases $f_A(F)$ applies, guaranteeing that resolution on $a$ is admissible.

With these tools, we can now prove that cases 8–12 get a branching number of $\tau(4, 8)$ or better.

4.2.2 Case 8: Variables of Higher Degree

Here, we prove that the branching number is sufficiently good when branching on any variable $x$ with $d(x) > 3$.

Lemma 16. If $F$ is a fully reduced formula with $d(F) > 3$, then applying case 8 of the algorithm results in a branching dominated by $(4, 8)$.

Proof. We show first that $\Delta f$ in both branches is at least $d(x) - 2$ plus the number of 2-clauses containing the variable $x$, and then we derive the effects of the long clauses. We will see that when no literal of $x$ occurs in more than two long clauses, it holds that $\Delta_1 f_A + \Delta_2 f_A \geq 12$ and $\Delta_i f_A \geq 4$ for $i = 1, 2$, and then we will prove that the branching is achieved when a literal of $x$ occurs in more long clauses as well.

The removal of $x$ increases $\Delta f_A$ by at least $d(x) - 2$ in each branch, and for each variable $y$, literals of $x$ and $y$ co-occur in at most one 2-clause, meaning that $\Delta f_A$ increases in both branches by the number of 2-clauses containing a literal of $x$, and the first claim is proven.
4.2. Average Degree up to Four

We will show that in addition to these reductions of $\Delta f_A$, as long as no literal of $x$ occurs in more than two long clauses, each long clause with $x$ increases $\Delta f_A$ by at least two in the $F[x]$ branch (and symmetrically for $\bar{x}$).

To see this, look closer at the list of possible cases. For any 2-clause $(x \lor y)$ with $d(y) = 3$, no further co-occurrence of $x$ and $y$ is possible. Also, $x$ and $y$ do not co-occur in two or more 2-clauses. The only case when there exists a 2-clause $(x \lor y)$ and $x$ and $y$ can co-occur in a long clause is if $d(y) > 3$ and there exists some (but only one) clause $(\bar{x} \lor \bar{y} \lor C)$ for some $C$. Similarly, if variables $x$ and $y$ co-occur more than once but never in a 2-clause, then either $d(y) > 3$ and $x$ and $y$ can co-occur several times as long as the same pair of literals never occurs in more than one clause (i.e. the variable $y$ occurs at most twice with the literal $x$), or $d(y) = 3$ and $x$ and $y$ co-occur only in clauses $(x \lor y \lor C)$, $(x \lor \bar{y} \lor D)$ (or similarly with $\bar{x}$), where $C$ and $D$ are both non-empty, do not share variables, and supposing that $\bar{y}$ is the 1-literal, $|D| > 1$.

From all of this, we can infer the following: if the variables $x$ and $y$ co-occur in both short and long clauses, then $d(y) > 3$ and the variables co-occur in exactly one short and one long clause, in which case $y$ is worth two points and we can count one point for each occurrence, and if the variables $x$ and $y$ co-occur in several long clauses, then either $d(y) > 3$ or we have the last case of the previous paragraph. Let $k$ be 2 plus the number of 2-clauses containing any literal of $x$. If the literal $x$ occurs in only one long clause $(x \lor C)$, then $\Delta f_A \geq k + |C|$ in the $F[x]$ branch. If the literal $x$ occurs in only two long clauses $(x \lor C)$ and $(x \lor D)$ not matching the last case of the previous paragraph, then $\Delta f_A \geq k + |C| + |D|$ in the $F[x]$ branch. If the last case of the previous paragraph does occur, and there are only two long clauses with the literal $x$, then $\Delta f \geq k + 1 + |C| + |D| \geq k + 4$ in the $F[x]$ branch. Clearly, as long as no literal of $x$ occurs with more than two long clauses, $\Delta_1 f_A + \Delta_2 f_A \geq 12$, and we need to show $\Delta f_A \geq 4$. Assume without loss of generality that $d^+(x) \geq d^-(x)$. If $\bar{x}$ is at least a 2-literal, or a 1-literal present in a 3-clause or longer clause, then the result is immediate. If $\bar{x}$ is a 1-literal present in a 2-
clause, say \((x \lor y)\), then the extra assignment \(\bar{y}\) removing some clause will ensure \(\Delta f_A \geq 4\) and prove the result.

Otherwise, if there are three long clauses with the literal \(x\), then \(\Delta f_A \geq k + 5\); the case \(k + 5\) can occur in a situation such as when clauses \((x \lor y \lor a), (x \lor z \lor b), (x \lor \bar{y} \lor \bar{z} \lor c)\) exist, with \(d(v) = 3\) for every involved variable \(v \neq x\). If \(\bar{x}\) is at least a 2-literal, then \(d(x) \geq 5\) and we have a branching dominated by \((5, 8)\). If \(\bar{x}\) is a 1-literal occurring in a 3-clause or longer, then we get a branching dominated by \((5, 7)\). If it occurs in a 2-clause \((\bar{x} \lor w)\), then the assignment \(\bar{w}\) when \(x = 0\) ensures that we get \(\Delta f_A \geq 4\) in this branch and a branching dominated by \((4, 8)\). This concludes the proof.

4.2.3 Case 9: Imposing More Structure

In every case from here on, \(F\) is 3-regular. We give some conditions under which case 9 of the algorithm applies, and show that the branching number will be at most \(\tau(6, 6)\). For the sake of convenience, assume without loss of generality that for any variable \(v\), the literal \(\bar{v}\) occurs only once in \(F\).

Lemma 17. If \(F\) is a 3-regular, fully reduced formula, then the following statements are true:

1. Any branch \(F[a]\) for a variable \(a\) reduces \(f_A(F)\) by at least 6.

2. Any branch \(F[a]\) for a variable \(a\) where the literal \(a\) occurs in some clause \(C\) with \(|C| \geq 5\) reduces \(f_A(F)\) by at least 6.

3. If literals \(a, b\) occur together in one clause, and \(a, \bar{b}\) occur together in another, then a branch \(F[a]\) reduces \(f_A(F)\) by at least 6.

Proof. For the first part, let \(S\) be the set of literals that occur in a clause together with \(a\) in \(F\). For every literal \(l \in S\), \(l\) is assigned 0 in the branch. We know that if \(l_1, l_2 \in S\), then any clause containing \(l_1\) does not contain \(l_2\) or \(a\); and for a clause \(C\) with \(l_1, l_2 \in C\), we have \(|C| > 2\) and \(|S| > 2\) if \(l_1\) is a negated literal, and \(|C| > 3\) if \(l_1\) is an
unnegated literal. Either way, each assignment \( \bar{l}_i \) affects at least two
literals not from the variables in \( S \).

- If \(|S| \geq 3\), then at least four variables are assigned in the branch,
and at least six literals beyond these are removed from \( F \). By
a simple counting argument, this requires at least six variables
to be affected.

- If \(|S| = 2\), then let \( S = \{l_1, l_2\} \) where \( l_1, l_2 \) are some literals
for variables \( b \) and \( c \), respectively. If some clause \( C \) contains
both literal \( \bar{l}_i \) and variable \( c \), then by necessity \( l_1 = b, l_2 = c \)
and \( C = (\bar{b} \lor c \lor C') \) where \(|C'| \geq 2\) and \( C' \) contains no literals
of variables \( a, b, c \). In this case, no clause containing \( \bar{c} \) can be
formed without using a sixth variable, by Lemma 14.

- Otherwise, \(|S| = 2\) and any clause containing \( \bar{l}_i \) for \( i = 1, 2 \) has
no other variable in common with the clause containing \( \bar{a} \). We
have three further cases, depending on the negations in \( S \).

1. If \( S = \{b, c\} \), then there must exist clauses \((\bar{b} \lor C), (\bar{c} \lor D)\)
with \(|C|, |D| \geq 2\). If less than six variables are affected,
then \( Vars(C) = Vars(D) \) and \(|C| = |D| = 2\), but then,
either resolution or backward resolution is admissible on a
variable in \( C \). Otherwise, at least six variables are removed
in the branch.

2. If \( S = \{b, \bar{c}\} \), then there exist clauses \((\bar{b} \lor C)\) with \(|C| \geq 2\)
and \((c \lor D), (c \lor E)\) with \(|D|, |E| \geq 1\). If less than six
variables are affected, then \(|D| = |E| = 1\) and \( Vars(C) =\)
\( Vars(D) \cup Vars(E) \), and by Lemma 15, we must have
clauses \((\bar{b} \lor \bar{u} \lor \bar{v}), (c \lor u), (c \lor v)\) for variables \( u, v \). Now,
the second appearances of literals \( u \) and \( v \) must occur in
different clauses, where no other literal of the variables
\( a, b, c, u \) or \( v \) can occur. Counting these clauses, at least
six variables are removed in the branch.

3. If \( S = \{\bar{b}, \bar{c}\} \), then we have clauses \((b \lor A), (b \lor B), (c \lor C), (c \lor D)\), where no case uses only six variables. By
Lemma 15 and since case 7 does not apply, we have $A, B \neq C, D$, and by Lemma 14, we have $\text{Vars}(A) \neq \text{Vars}(B)$ and $\text{Vars}(C) \neq \text{Vars}(D)$, so either $A, \ldots, D$ are all of length one with distinct variables (for a reduction of at least 7 in the branch) or at least one, say $C$, has $|C| > 1$. In the latter case, $D$ still introduces a variable not in $C$, for a total reduction of at least 6.

This concludes the proof of the first part of the lemma.

For the second part, unless the statement is trivial, assume without loss of generality that $C = (a \lor l_1 \lor l_2 \lor l_3 \lor l_4)$, where $l_1, \ldots, l_4$ are literals of variables $b, \ldots, e$, respectively. By assumption, there is one more clause $D$ containing literal $a$, and by Lemma 14, $D$ contains at least one variable other than $a, \ldots, e$. At least six variables are affected by the assignment $a$.

For the third part, by Lemma 15, unless the statement is trivial the clauses can without loss of generality be assumed to be $(a \lor b \lor l_1)$, $(a \lor b \lor l_2 \lor l_3)$ where $l_1, \ldots, l_3$ are literals of variables $c, \ldots, e$. If the reduction in $f_A(F)$ is less than 6, then the second occurrence of literal $b$ must occur in a clause using only these variables. No such clause can exist.

Note that we have now covered all cases where two variables $a$ and $b$ have more than one co-occurrence in $F$:

- If literals $a, b$ co-occur in one clause, and $\bar{a}, \bar{b}$ co-occur in another, then resolution on $a$ or $b$ produces only one non-trivial resolvent. The same holds if the co-occurring pairs of literals are $a, \bar{b}$ and $\bar{a}, b$.

- If literals $a, b$ co-occur in one clause, and $a, \bar{b}$ co-occur in another, then either a reduction applies (e.g. resolution on $b$ is admissible) or a branching dominated by $\tau(6, 6)$ is performed, by Lemma 17. Co-occurring pairs of literals $\bar{a}, b$ and $\bar{a}, b$ is not possible, under the assumption that any literal $\bar{v}$ for any variable $v$ has only one occurrence.
4.2. Average Degree up to Four

- If literals $a, b$ co-occur twice, then backward resolution is admissible. We would add one new variable of weight 1, but we would reduce the weights of both variables $a$ and $b$. Other co-occurring pairs of literals are not possible, under the assumption that any literal $\bar{v}$ for any variable $v$ has only one occurrence.

This will be useful in the next lemma, where we show that, in addition, any clause containing variables of mixed signs causes case 9 to apply, with a branching dominated by $\tau(6, 6)$.

**Lemma 18.** Let $F$ be a 3-regular, fully reduced formula where no condition from Lemma 17 applies. Assume without loss of generality that for every variable $v$, literal $\bar{v}$ is a 1-literal. Then the following statements hold:

1. If there is a clause $C$ with literals $a, \bar{b}, c$ for some variables $a, b, c$, then a branch $F[a]$ reduces $f_A(F)$ by at least 6.

2. If there is no such clause, but there is a clause $C$ with literals $a$ and $\bar{b}$ for some variables $a, b$, then a branch $F[a]$ reduces $f_A(F)$ by at least 6.

**Proof.** We begin by proving the first part. Assignments $b = 1$ and $c = 1$ will be made, so that five further clauses are satisfied: one with $a$, and two each with $b$ and $c$. If $|C| = 4$, then a fourth variable is assigned, and no single variable occurs in more than 3 clauses. If the literal $b$ (or $c$) has at least three neighbours, then these must all be from different variables other than $a, b$ or $c$, and at least six variables are removed. Otherwise, clauses $(b \lor d)$ and $(b \lor e)$ occur, and by Lemma 15, $c$ cannot occur with the literal $d$ (and not with the literal $\bar{d}$ either, since the clause must be short). We get $\Delta f_A \geq 6$.

Now we prove the second part. Assignment $b$ will be made, and $|C| \geq 3$. As before, if the literal $b$ has at least three neighbours, then $\Delta f_A \geq 6$, otherwise clauses $(b \lor d), (b \lor e)$ must occur, plus a clause containing only the literal $a$ and literals of $d$ and $e$. If this latter clause is a 2-clause, say $(a \lor d)$, then resolution on $b$ leaves at most one surviving resolvent. If this latter clause contains a negated
variable, say the clause \((a \lor d \lor \bar{e})\) or \((a \lor \bar{d} \lor \bar{e})\), then resolution on \(e\) leaves at most one surviving resolvent. Otherwise, both occurrences of literals \(d\) and \(e\) have been accounted for, so that \(d = e = 0\) are assigned, and the literals \(\bar{d}\) and \(\bar{e}\) occur in separate clauses which are not 2-clauses. Each of these must contain a sixth variable.

We see that for any \(F\) where none of cases 0–9 apply, we have a specific structure where every clause \(C\) contains either only 2-literals, in which case \(2 \leq |C| \leq 4\), or only 1-literals, in which case \(|C| \geq 3\). Additionally, every pair of variables co-occurs in at most one clause.

### 4.2.4 The Final Cases

Given the structure imposed by case 9, showing the rest of the results is relatively easy. Case 10 imposes a stricter limit on the length of a clause with 1-literals, with a branching dominated by \(\tau(6, 6)\) as shown in Lemma 19; case 11 gives us stronger guarantees on the neighbourhood of a 2-literal, with a branching dominated by \(\tau(4, 8)\) as shown in Lemma 21; and finally, if all other cases fail to apply, then case 12 can be applied to convert the formula to an instance of \((3, 2)\)-csp, as shown in Lemma 22. We begin by giving the bound for case 10.

**Lemma 19.** Let \(F\) be a sat formula where case 10 is the earliest case of the algorithm SparseSAT that applies. The branching for this case is dominated by \(\tau(6, 6)\).

**Proof.** Let \(C\) be the clause that is being split. For any literal \(l_i \in C\) that is not included in the new clause, \(\bar{l_i}\) becomes a pure literal, and an assignment \(l_i = 0\) is made. For each such assignment, two literals for other variables are affected. If there are at least three such assignments, then at least six additional literals occur in satisfied clauses, since no pair of variables from \(C\) can not co-occur under any negations, and by a counting argument these six literals will cause at least three further variables to get their degrees decreased, since no clauses with mixed signs for the members exist. We will refer to a variable that either gets its degree decreased or is assigned as reduced.
If only two literals become pure, say $a, b$, then let $S_i$ for $i = 1, 2$ be the set of literals $v$ such that $v$ occurs in $i$ clauses together with literal $a$ or $b$. Assume without loss of generality that $S_1 = \{u_1, \ldots, u_d\}$ and $S_2 = \{v_1, \ldots, v_e\}$. We have $|S_1| + 2|S_2| \geq 4$, and for every literal $l \in S_2$ an additional assignment $\bar{l}$ is made. We trace these assignments:

1. If $S_2 = \emptyset$, then the reduction in $f_A(F)$ is at least $2 + |S_1| \geq 6$.

2. If $|S_2| = 1$ and $|S_1| \geq 2$, then let $D$ be the clause where $\bar{v}_1$ occurs. Five variables contribute to $\Delta f$ already; the only way to form the clause $D$ using only these five variables is $D = (\bar{u}_1 \lor \bar{u}_2 \lor \bar{v}_1)$, but then $u_1$ and $u_2$ are assigned and must lie in different clauses, which requires extra variables that are reduced. Otherwise, a sixth variable is reduced when $D$ is satisfied.

3. If $|S_2| = 2$, then some literal $\bar{w}$ shares a clause with some $\bar{v}_i$, and assignment $w$ is made. At least one occurrence of $w$ is in a clause with some new variable, for a reduction of at least 6.

4. Finally, $|S_2| \geq 3$. If $|S_2| + |S_1| > 3$, then the reduction is at least 6. Otherwise, some extra variable is required to form a clause with $\bar{v}_i$, and in any case, at least six variables in total are reduced.

Our next lemma simplifies the analysis of case 11 in Lemma 21.

**Lemma 20.** Let $F$ be a CNF formula such that no case before case 11 of SparseSAT applies. Let $a$ be a variable and, without loss of generality, assume that literal $\bar{a}$ occurs once in $F$. If $a$ is a member of $k$ 2-clauses, then the branch $F[\bar{a}]$ reduces $f_A(F)$ by at least $7 + k$.

**Proof.** Let the clause that contains $\bar{a}$ be $(\bar{a} \lor \bar{b} \lor \bar{c})$, so that assignments $b$ and $c$ are made. Each 2-clause containing $a, b$ or $c$ contributes one variable which does not occur among the other clauses, and each longer clause contains at least 2 literals of further variables. In total, since no mixed clauses exist, at least $7 + k$ variables are assigned or get their degrees reduced. 

\qed
4. Satisfiability for Sparse Formulae

Now, we give the bound for case 11.

**Lemma 21.** If $F$ is a CNF formula such that case 11 is the earliest case of SparseSAT that applies, then the branching is dominated by $\tau(4, 8)$. If case 11 does not apply either, then every 2-literal $l$ is involved in exactly two 2-clauses.

*Proof.* If $a$ is part of no 2-clauses, then the number of variables affected by assignment $a$ is at least 5, which by Lemma 20 leads to a branching with a branching number of at most $\tau(5, 7) < \tau(4, 8)$. If literal $a$ neighbours only three other variables, then $a$ must be involved in one 2-clause, and by the same lemma, we have a branching with a branching number of at most $\tau(4, 8)$. The remaining case, with only two other variables, can only be achieved by two 2-clauses. \hfill \square

Finally, we bound the total time used by case 12.

**Lemma 22.** If $F$ is a CNF formula such that no case among cases 0–11 of SparseSAT applies to $F$, then the construction in Lemma 8 is applicable, and the total time for SparseSAT($F$) is $O \left( \frac{1.3645 n(F)}{3} \right) \subset O \left( \frac{1.1902 f_A(F)}{f_A(F)} \right)$.

*Proof.* In addition to the structural properties noted previously, we have by case 10 that $|C| = 3$ for every clause $C$ with 1-literals and by case 11, as noted in Lemma 21, $|C| = 2$ for every clause $C$ with 2-literals, which proves the applicability of the construction. Eppstein’s algorithm [30] runs in time $O \left( 1.3645^n \right)$, and the resulting CSP instance has $n(F)/3$ variables. With $f_A(F) = n(F)$ at this point in the algorithm, we get the described running time. \hfill \square

This concludes our sequence of lemmas. We will wrap up with the main theorem of this section.

**Theorem 23.** If $F$ is a CNF formula with $s(F)$ singletons and $l(F) \leq 4n(F)$, then SparseSAT($F$) decides the satisfiability of $F$ in time $O^* \left( \tau(4, 8)^{l(F)} - 2n(F) + s(F) \right) \subset O^* \left( 1.1279^{l(F)} - 2n(F) + s(F) \right)$. 

4.3 Average Degree More than Four

In the previous section, we gave an upper bound on the running time of SparseSAT($F$) when the average degree of $F$ is at most four. We can now use the method of analysis by average degree, using the compound measure $f_B(F)$ defined in Section 4.1, to get better bounds for higher degrees. In the process, we will derive a bound of $O\left(1.0663^{f_B(F)}\right)$ for any CNF formula $F$.

We repeat the definitions of $b_k$ for convenience:

\begin{align*}
b_1 &= b_2 = 0 & (4.6) \\
\tau(4b_3, 8b_3) &= 2 & (4.7) \\
\tau(4b_4, 8b_4) &= 2 & (4.8) \\
\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) &= 2 & (4.9) \\
\tau(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k - 3)b_k) &= 2 \text{ for } k \geq 6 & (4.10)
\end{align*}

Recall from Lemma 7 that there are analytical solutions for $b_3$ to $b_5$, while for $b_k$ with $k \geq 6$, we will need numerical approximations. The upper bound is given by $\chi_k = \sum_{i=1}^{k} b_k$. Numerical values for $k \leq 10$ are found in Table 4.2, and the asymptotic growth of the bound for higher $k$ is derived in the following lemma.

**Lemma 24.** $\chi_k = 1 - c/(k + 1) + O\left(1/k^2\right)$ for some $c$.

**Proof.** Revisit the formula $\tau(1 - x, 1 + y) = 2$, i.e.

\[2^{-1+x} + 2^{-1-y} = 1.\]
Figure 4.1: Worst-case running time expressed as $O(2^{cn})$ depending on average degree $l/n$ (c on vertical axis, $l/n$ on horizontal axis) for Hirsch’s algorithm (top line) and SparseSAT (bottom line)

<table>
<thead>
<tr>
<th>$k$</th>
<th>$a_k$</th>
<th>$b_k$</th>
<th>$\chi_k$</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>-0.347121</td>
<td>0.173560</td>
<td>0.173560</td>
<td>$O(2^{0.117666n}) \subset O(1.1279^n)$</td>
</tr>
<tr>
<td>4</td>
<td>-0.347121</td>
<td>0.173560</td>
<td>0.347121</td>
<td>$O(2^{0.3472n}) \subset O(1.2721^n)$</td>
</tr>
<tr>
<td>5</td>
<td>-0.115707</td>
<td>0.115707</td>
<td>0.462828</td>
<td>$O(2^{0.4629n}) \subset O(1.3783^n)$</td>
</tr>
<tr>
<td>6</td>
<td>0.073130</td>
<td>0.077940</td>
<td>0.540768</td>
<td>$O(2^{0.5408n}) \subset O(1.4548^n)$</td>
</tr>
<tr>
<td>7</td>
<td>0.188505</td>
<td>0.058710</td>
<td>0.599478</td>
<td>$O(2^{0.5995n}) \subset O(1.5152^n)$</td>
</tr>
<tr>
<td>8</td>
<td>0.278738</td>
<td>0.045820</td>
<td>0.645298</td>
<td>$O(2^{0.6453n}) \subset O(1.5641^n)$</td>
</tr>
<tr>
<td>9</td>
<td>0.352328</td>
<td>0.036621</td>
<td>0.681920</td>
<td>$O(2^{0.6820n}) \subset O(1.6043^n)$</td>
</tr>
<tr>
<td>10</td>
<td>0.411685</td>
<td>0.030026</td>
<td>0.711946</td>
<td>$O(2^{0.7120n}) \subset O(1.6381^n)$</td>
</tr>
</tbody>
</table>

Table 4.2: Approximate values for the parameters in $f_k(l, n) = a_k n + b_k l$ and $\chi_k = \sum_{i=1}^{k} b_i$, and worst-case running time in each section
We have $y = -1 - \log_2(1 - 2^{-1+x})$. By calculating the first term of the Taylor power series of this, we get $y = x + O\left(x^2\right)$. Now, $x = 1 - \chi_k - 4b_k$ and $y = (2k - 4)b_k - (1 - \chi_k)$. We have

$$b_k = (1 - \chi_k)/k + O\left(x^2/k\right),$$

or, as $\chi_k = \chi_{k-1} + b_k$,

$$b_k = (1 - \chi_{k-1})/(k + 1) + O\left(x^2/k\right).$$

Regarding the value of $x = 1 - \chi_k - 4b_k$, consider equation (4.10) again. We may assume that $k \geq 6$, so that this equation is valid. By the balance property of $\tau$, the average of the two parts is greater than $1$, i.e.

$$(2\chi_{k-1} + (2k + 2)b_k)/2 > 1.$$  

We get $(1 - \chi_{k-1})/(k + 1) < b_k$. Now, let $r_k = 1 - \chi_k$. We have

$$r_k = r_{k-1} - b_k < r_{k-1} - r_{k-1}/(k + 1) = r_{k-1} \cdot k/(k + 1).$$

If $r_{k-1} < c/k$ for some $c$, then $r_k < c/(k + 1)$. Fix $c$ so that $r_5 < c/6$. Now, by induction, $1 - \chi_k < c/(k + 1)$ for $k \geq 6$.

We have $x < 1 - \chi_k < c/(k + 1)$, so

$$b_k = (1 - \chi_{k-1})/(k + 1) + O\left(1/k^3\right).$$

Expressed in $r_k$, we have

$$r_k = r_{k-1} - b_k = r_{k-1} \cdot k/(k + 1) - O\left(1/k^3\right).$$

From this, we get that $\chi_k = 1 - r_k$ is $1 - c/(k + 1) + O\left(1/k^2\right)$ for some $c$.

Finally, we show the relation between the bound $O\left(2^{\chi_k n}\right)$ and a bound of the form $O\left(2^{al}\right)$.

**Lemma 25.** For all values of $l$ and $n$, $f(l, n) \leq 0.0926l$, with a maximum $f(l, n)/l$ value occurring when $l = 5n$. 


Figure 4.2: Worst-case running time expressed as $O(2^{cl})$ depending on $l/n$ (c on vertical axis, $l/n$ on horizontal axis)
4.3. Average Degree More than Four

Proof. With \( f_k(l, n) = a_k n + b_k l \), let \( l = (k - 1 + \alpha) n \) (where \( 0 \leq \alpha \leq 1 \)). We have \( f_k(l, n) = (b_k + a_k / (k - 1 + \alpha)) \cdot l \). We see that the highest value of \( f_k(l, n)/l \) occurs when \( \alpha = 0 \) if \( a_k > 0 \) and when \( \alpha = 1 \) otherwise. By Lemma 7, \( a_k > 0 \) for \( k \geq 6 \), so the globally highest value of \( f_k(l, n)/l \) is \( \chi_5 / 5 < 0.0926 \).

The running times are illustrated graphically in Figure 4.1, along with the previously best bound for a comparable algorithm, Hirsch’s [49] algorithm with a running time in \( O(2^{0.10297 l}) \), included for comparison. In the rest of this section, we prove that \( \text{SparseSAT}(F) \) has a running time in \( O^*(2^{f(F)}) \), divided into Section 4.3.1 for the case \( 4n < l \leq 5n \), and Section 4.3.2 for \( l > 5n \). But first, we state a simple result that connects the result from the previous section with the measure used here.

Lemma 26. If \( F \) is a fully reduced \( \text{CNF} \) formula with \( l(F) \leq 3n(F) \), then \( \text{SparseSAT}(F) \) decides the satisfiability of \( F \) in time \( O^*(2^{\chi_3 n}) \). If \( l(F) \leq 4n(F) \), then the time is \( O^*(2^{\chi_4 n}) \).

4.3.1 Five Occurrences per Variable

In this section, \( f_5(l, n) = a_5 n + b_5 l \) is used, with \( a_5 = -b_5 \) and \( b_5 = 2/3 \cdot b_3 \approx 0.115707 \). We will prove that \( \tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2 \) is the worst-case branching.

Lemma 27. If \( F \) is a step 5-reduced formula with \( \lceil l(F)/n(F) \rceil = 5 \), then the following hold:

1. If there is a clause \((l \lor C)\) in \( F \), such that \( l \) is a 1-literal, the variable of \( l \) is of degree 3, and \(|C| \leq 2\), then resolution on \( l \) is admissible.

2. If there are two clauses \((l_1 \lor l_2 \lor C)\) and \((l_1 \lor l_2 \lor D)\) in \( F \), such that \( l_1 \) or \( l_2 \) is a literal of a variable of degree 3, then backward resolution on these clauses is admissible.

Proof. For the first part, resolution is admissible if \( \Delta l(F) \geq \Delta n(F) \). If resolution on \( l \) is performed, then \( \Delta n(F) = 1 \) and \( \Delta l(F) \geq 3 - |C| \geq \)
4. Satisfiability for Sparse Formulae

1. Since no reduction increases $\Delta n(F)$ without also increasing $\Delta l(F)$, we find that resolution on $l$ is admissible.

For the second part, backward resolution requires $\Delta l(F) > \Delta n(F)$. Assume that $l_1$ is a literal of a 3-variable. If backward resolution is applied on the clauses, then we immediately have $\Delta n(F) = -1$ and $\Delta l(F) = -1$, and we know that the reduction in case 5 applies to $l_1$. Since no singleton has been created, we find that some reduction applies (possibly case 5, but not necessarily) and that after the application of this reduction, $\Delta l(F) > \Delta n(F)$. We find that backward resolution on these clauses is admissible.

Since $a_5 + b_5 = 0$, the only pitfall when evaluating branchings in this section is a variable that has all its occurrences within the clauses removed by an assignment $x$ or $\bar{x}$ (or assignments $\bar{x}, \bar{l}_1, \ldots, \bar{l}_d$ if $\bar{x}$ is a 1-literal). This limits the number of cases that we have to consider in the following result.

Lemma 28. If $F$ is a fully reduced CNF formula and $[l/n] = 5$, then the worst-case branching number for SparseSAT when applied to $F$ is $\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2$.

Proof. Let $x$ be the variable we branch on; $d(x) \geq 5$. If there is a 2-clause $(x \lor l)$, then there can be no co-occurrences of literals $\bar{x}$ and $l$. Also, at least two resolvents by $l$ must be non-trivial, so if $l$ is a 1-literal, then at least two occurrences of $\bar{l}$ are in clauses not containing the literal $\bar{x}$, and if $l$ is at least a 2-literal, then all occurrences of the literal $l$ are in clauses not containing the literal $\bar{x}$. We find that at least two literals of $l$ occur in clauses without $\bar{x}$. That is, as $a_5$ and $b_5$ cancel each other out, a 2-clause contributes at least $b_5$ to both branches, while a 3-clause contributes $2b_5$ to one branch. Therefore, a 2-clause only occurs in a worst case if it causes a higher imbalance in $\Delta f$.

Regarding the complications arising from having $a_5 < 0$, the only case where $\Delta f$ could decrease due to an unexpectedly high $\Delta n$ is when a variable has all its occurrences among literals already accounted for, and thus disappears from $F$ without an assignment. For most of this proof, this can only occur if some variable $v$, $d(v) \geq 4$, has all its
occurrences with the literal $x$ (or $\bar{x}$), which requires that $x$ occurs in at least four 3-clauses, with a minimum $\Delta f$ of $(d(x) + 8)b_5 + 3a_5$ in the branch $x$, while the minimum $\Delta f$ when $x$ is a 3-literal occurring in no 2-clauses is $(d(x) + 6)b_5 + a_5$. No new worst-case branchings are introduced by this case. When there are other cases where a variable can disappear, these are addressed specifically.

To start with, assume that $\bar{x}$ is a 1-literal. In the branch with assignment $\bar{x}$, $x$ contributes at least $5b_5 + a_5$, and each neighbour of $\bar{x}$ with degree $d$ contributes at least $db_5 + a_5$. Let $C$ be the clause containing $\bar{x}$. If $|C| \geq 3$, then $\Delta f$ in this branch is at least $8b_5$. Otherwise, assume that $C = (\bar{x} \lor y)$ and let $D$ be a clause where $\bar{y}$ appears. The clause $D$ contains at least one literal not of variable $x$ or $y$. If $d(y) = 3$, then $|D| \geq 4$; otherwise, $D$ may contain only one further literal. In either case, the minimum reduction in $f(F)$ is again $8b_5$, and no further variables can disappear without also increasing $\Delta l$. The literal $x$ is at least a 4-literal, and as noted above, the minimum reduction is $10b_5$. We have a branching $\tau(8b_5, 10b_5) = \tau((5 + 1/3)b_3, (6 + 2/3)b_3) < 2$.

Next, assume that $\bar{x}$ is a 2-literal. If $\bar{x}$ is involved in two 2-clauses, then the branch $\bar{x}$ will reduce $f(F)$ by at least $6b_5$, and the branch $x$ by at least $12b_5$, counting the contributions from the variables of the 2-clauses and the neighbours of $x$ that do not appear in these 2-clauses. Our branching number is $\tau(6b_5, 12b_5) = \tau(4b_3, 8b_3) = 2$. If $\bar{x}$ is involved in one 2-clause, then the branching number is $\tau(7b_5, 11b_5) < 2$, and with no 2-clauses, $\tau(8b_5, 10b_5) < 2$. Having $d(x) > 5$ will not yield any harder cases. Expressing the worst case in $\chi_4$ and $b_5$, we have $\tau(\chi_4 + 3b_5, 3\chi_4 + 3b_5) = 2$.

### 4.3.2 Six or More Occurrences per Variable

In all further component measures, $a_k$ and $b_k$ are both positive, and the worst-case branchings all appear when there are no 2-clauses. For $l \leq 10n$, we need to prove this section by section. When $l > 10n$, we can give a general proof.

**Lemma 29.** For a fully reduced CNF formula $F$, let $k = \lceil l(F)/n(F) \rceil$. If $k > 5$, then the worst-case branching number for SparseSAT when
applied to $F$ is $\tau(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k - 3)b_k) = 2$.

**Proof.** Assume that we are branching on variable $x$, in section $k$, and $d(x) = k$. First, let $\bar{x}$ be a 2-literal, involved in zero, one or two 2-clauses. With no 2-clauses, we get a reduction of $(k + 4)b_k + a_k$ in branch $\bar{x}$, and $(3k - 4)b_k + a_k$ in branch $x$. With $a_k = \chi_{k-1} - (k - 1)b_k$, we get a branching number of $\tau(\chi_{k-1} + 5b_k, \chi_{k-1} + (2k - 3)b_k)$, which is 2 by definition of $b_k$.

For a 2-clause $(\bar{x} \lor y)$, the statements in Lemma 28 still hold, and assignment $y$ in branch $x$ adds a further $2b_k + a_k$, independently of reductions due to other assignments and clauses containing $x$. We find that with $\bar{x}$ being involved in one 2-clause the branching is $\tau(\chi_{k-1} + 4b_k, 2\chi_{k-1} + kb_k)$ and with two 2-clauses, $\tau(\chi_{k-1} + 3b_k, 3\chi_{k-1} + 3b_k)$. We will show that both of these branching numbers are lower than 2 for every value of $k$. Note that $\tau(\chi_7, 3\chi_7) < 2$ already, so when $l > 7n$, the case with two 2-clauses can be excluded as a worst case. Similarly, $\tau(\chi_{10}, 2\chi_{10}) < 2$, so when $l > 10n$, we can see immediately that the case where $\bar{x}$ is a 2-literal and not involved in any 2-clauses is the worst case. For the sections up to those points, we need to evaluate the branching number case by case. Doing so, we find that $\tau(\chi_{k-1} + 4b_k, 2\chi_{k-1} + kb_k) < 2$ and $\tau(\chi_{k-1} + 3b_k, 3\chi_{k-1} + 3b_k) < 2$ for every $k \geq 6$.

If $\bar{x}$ is instead a 1-literal, then assume that the clause containing $\bar{x}$ is $(\bar{x} \lor l \lor C)$ for some possibly empty $C$, where $l$ is a literal of the variable $y$. The reduction in the branch $x$ will be no lower than when $\bar{x}$ is a 2-literal occurring in no 2-clauses, since $x$ here appears in at least one more clause; we will show that the reduction in the branch $\bar{x}$ is also no lower than when $\bar{x}$ is a 2-literal occurring in no 2-clauses.

- If $d(y) \geq 4$, then the reduction from variables $x$ and $y$ alone in the $\bar{x}$ branch is at least $(k + 4)b_k + 2a_k$.
- If $d(y) = 3$ and $l$ is a 2-literal, then the clause $D$ where $\bar{l}$ appears is at least a 5-clause, with at least three literals not of variables $x$ and $y$, for a reduction of at least $(k + 6)b_k + 2a_k$.
- If $d(y) = 3$ and $l$ is a 1-literal, then $|C| \geq 3$ and counting only the assignments we have a reduction of at least $(k + 12)b_k + 5a_k$. 
We see that the case when $\bar{x}$ is a 1-literal can add no more difficult case. This concludes the proof.

Now that we know that the branching number is at most 2 for every section of $l/n$, we can give the general theorem.

**Theorem 30.** If $F$ is a CNF formula where either $l(F) \geq 4n(F)$ or $F$ is free of singletons, then the running time of SparseSAT$(F)$ is in $O\left(2^{f_B(F)}\right)$, where $f_B$ is the function defined earlier in this chapter.

**Proof.** For a fully reduced $F$, it follows from the lemmas in this section. If $l(F) < 4n(F)$ and $F$ contains no singletons, then it follows from Theorem 23. If $l(F) \geq 4n(F)$, then we see that the process of applying the reductions never increases $f_k(F)$ (where $k$ is the section that $F$ belongs to).

**Corollary 31.** The running time of SparseSAT$(F)$, without any restrictions on $F$, is in $O\left(2^{0.0926l(F)}\right) \subset O\left(1.0663^{l(F)}\right)$ regardless of the value for $n(F)$.

**Proof.** By Lemma 25, $f(l, n) \leq 0.0926l$ for all $l$ and $n$. If $F'$ is the step 3-reduced version of $F$, then $l(F') \leq l(F)$, and by Theorem 30, the running time of SparseSAT$(F')$ will be in $O\left(2^{f(F')}\right)$, and thus also in $O\left(2^{0.0926l(F')}\right)$.
Chapter 5

One-in-three Satisfiability

In this chapter we consider the problem of one-in-three satisfiability (X3sat), and give a new algorithm XShort that solves X3sat instances in time $O^*(1.0984^n)$. The algorithm works partly by applying resolution to certain variables, creating longer clauses; the algorithm thus applies to instances with longer clauses as well, with a running time of $O^*(1.0984^{n+\lambda})$ where $\lambda$ is a “length bonus” putting extra cost on longer clauses, defined as follows.

Definition 32. Let $F$ be an xsat instance and let $m_i$ be the number of $i$-clauses in $F$. The length bonus of $F$ is $\lambda(F) = \sum_{i \geq 4} (i - 3)m_i$. The word is also used in the context of a branching; there, a length bonus in a branch refers to a reduction in the length bonus of the instance (i.e. $\Delta f(F)$ increases due to the length bonus).

The algorithm is partially based on Byskov’s algorithm for X3sat with a running time of $O^*(1.1004^n)$ [9]. The main differences are the use of resolution to eliminate $(1,1)$-variables and a more intricate analysis of the case when the instance is sparse.

5.1 Algorithm Preliminaries

Clauses in this problem must be satisfied by exactly one literal (rather than by at least one literal). To distinguish such clauses from stan-
standard disjunctive clauses, we write them as \((l_1, l_2, \ldots, l_k)\) (and they are allowed to contain multiple copies of the same literal). We also get a different definition of propagation for the exact satisfiability problems:

- \(F[l_1 = l_2]\) for literals \(l_1\) and \(l_2\) is constructed by replacing every occurrence of \(l_2\) in \(F\) by \(l_1\), and every occurrence of \(\overline{l}_2\) by \(\overline{l}_1\).

- \(F[l_1 \neq l_2]\) is defined as \(F[l_1 = \overline{l}_2]\).

- \(F[l = 1]\) for a literal \(l\) is constructed by replacing every occurrence of \(l\) by 1, removing any occurrence of \(\overline{l}\) (shortening any clause \((\overline{l}, C)\) into just \((C)\), even if \(C\) is empty), replacing any clause containing more than one 1 by the empty clause, and for any clause \((1, l_1, \ldots, l_i)\) containing exactly one 1 also performing the assignments \(l_1 = 0\) up to \(l_i = 0\) in the same manner.

- \(F[l = 0]\) is defined as \(F[\overline{l} = 1]\).

The process is generalised to apply to several variables, e.g. \(F[A = 0]\) for a set \(A\) is interpreted to mean that every member of \(A\) is set to 0. That this process terminates in a satisfiable formula if and only if there is a model for \(F\) with the appropriate assignment(s) and replacement(s) is considered evident.

We need a number of definitions before we present the algorithm: Section 5.1.1 defines concepts and reductions relating to cycles in the formula; Section 5.1.2 defines concepts and reductions relating to a connectivity concept we refer to as interfaces; and Section 5.1.3 presents the actual algorithm.

### 5.1.1 Cycles

Next, we give some terminology for cycles and define a reduction we refer to as cycle replacement, but before we give the technical definition, let us give an example.

Consider a set of clauses \((a, t, u), (b, u, v), (c, v, w), (d, t, w)\). Because of the pattern of occurrences for the variables \(t, \ldots, w\), these clauses will be referred to as a cycle. The variables \(t, \ldots, w\) are the
5. One-in-three Satisfiability

<table>
<thead>
<tr>
<th>Label sequence</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>a,b,c,d</td>
<td>a = d, b = c</td>
</tr>
<tr>
<td>a,b,c,e</td>
<td>a = d = e, b = c = f</td>
</tr>
<tr>
<td>a,c,b,d</td>
<td>Remove any one clause of the cycle</td>
</tr>
<tr>
<td>a,c,b,e</td>
<td>c = f, d = e</td>
</tr>
<tr>
<td>a,b,c,e,c,f</td>
<td>a = d, b = c</td>
</tr>
<tr>
<td>a,c,a,d,b,e</td>
<td>a = e, b = f</td>
</tr>
<tr>
<td>a,c,a,e,b,f</td>
<td>a = c, b = d</td>
</tr>
<tr>
<td>a,c,e,a,d,f</td>
<td>c = e, d = f</td>
</tr>
<tr>
<td>a,c,e,a,f,d</td>
<td>Remove any one clause of the cycle</td>
</tr>
</tbody>
</table>

Table 5.1: Cases for cycle replacement

core of the cycle (since they are the variables that occur in a circular pattern), and a, . . . , d are the labels.

Definition 33. A k-cycle is a set of k clauses \((l_1, c_1, c_2), (l_2, c_2, c_3), \ldots, (l_k, c_k, c_1)\), where \(c_i\) are non-negated literals of \(k\) different variables called the core of the cycle, and \(l_i\) are literals of variables that are different from the core variables, but not necessarily from one another. The sequence \(l_1, \ldots, l_k\) are the labels of the cycle. A cycle replacement is said to apply to \(F\) if there is a \((3, 0)\)-variable \(x\) occurring only in short clauses \((x, a, b), (x, c, d), (x, e, f)\) and a cycle where all labels are positive literals of \(a–f\), in the following manner:

- Case 1: The cycle is a 4-cycle. In this case, match the labels of the cycle against Table 5.1 (see below on how to match); the corresponding reduction is given in the table.

- Case 2: The cycle is a 6-cycle, and exactly four variables are represented among the labels, with two variables being represented twice. In this case, let \(p\) and \(q\) be the variables that are represented once; the corresponding reduction is \(p = q\).

- Case 3: The cycle is a 6-cycle, and exactly five variables are represented among the labels. In this case, match the labels...
of the cycle against Table 5.1 (see below); the corresponding reduction is given in the table.

– **Case 4:** The cycle is a 6-cycle, and all six variables are represented among the labels. In this case, the corresponding reduction is $x = 1$.

Matching the labels against the table is to be done under structure-preserving transformations: rotation of the cycle, reading the cycle backwards, reordering of the clauses containing $x$, and reordering of the literals within the clauses.

We restrict ourselves to cycles with non-negated core literals mainly because the main application of the cycle terminology is in the context of cycle replacements, where we do not need to handle cycles with negated core literals.

As an example of a cycle replacement, assume that there exist clauses $(x, a, b), (x, c, d), (x, e, f)$ for a $(3, 0)$-variable $x$. If the clauses $(a, t, u), (b, u, v), (c, v, w), (d, t, w)$ from the previous example exist, then a cycle replacement applies: the four clauses of the cycle match exactly the first line of Table 5.1, and we can set $a = d$ and $b = c$ while preserving the satisfiability of the instance. The same case also applies to other sets of clauses, under the structure-preserving transformations:

– The clauses $(b, t, u), (c, u, v), (d, v, w), (a, t, w)$ match under rotation of the cycle (if we start reading from the label $a$, we get the sequence $a, b, c, d$). The same reduction applies.

– The clauses $(b, t, u), (a, u, v), (c, v, w), (d, t, w)$ match under reordering of the literals of the clause $(x, a, b)$ (i.e. swapping the positions of $a$ and $b$). In this case, the reduction is changed to $b = d, a = c$.

– The clauses $(a, t, u), (b, u, v), (e, v, w), (f, t, w)$ match under reordering of the clauses containing $x$ (so that $e, f$ take on the role of $c, d$ in the definition). In this case, the reduction is changed to $a = f, b = e$. 
Definition 34. A cycle of replacements is a set of 2-clauses that form a cycle (e.g. \((a, b), (a, c), \) and \((\bar{b}, c)\)). It is consistent if performing a replacement \(l_1 \neq l_2\) for each 2-clause \((l_1, l_2)\) leads to a consistent set of replacements (e.g. the mentioned 3-cycle of replacements is consistent, but replacing \(b\) by \(\bar{b}\) would result in a cycle of replacements that is not consistent).

Cycles of replacements are only indirectly related to cycle replacements: cycle replacements apply to cases where setting \(x = 1\) might otherwise leave a consistent, even cycle of replacements. For instance, consider the cycle \((a, t, u), (b, u, v), (c, v, w), (d, t, w)\), with the variable \(x\) appearing in clauses \((x, a, b), (x, c, d), (x, e, f)\). When \(x = 1\), we get \(a = \ldots = d = 0\) and the clauses of the cycle reduce to a consistent 4-cycle of replacements \(t \neq u \neq v \neq w\). In this instance, we get four replacements which, when applied, reduce the number of variables by three. On the other hand, when \(x = 0\) there are for each clause \((a, t, u)\) two options:

- If \(a = 1\), then \(t = u = 0\).
- If \(a = 0\), then \(t \neq u\).

For each combination of values to the labels of a cycle, we get some cycle of equalities and inequalities on the core variables of the cycle. In particular, if an odd number of the labels are set to 0 (which is impossible for the particular example given, but possible for e.g. the label sequence \(a, b, c, e\)), then we get an odd number of inequalities in the cycle, which is a contradiction.

The cycle replacements do not cover all cases where a cycle of replacements can occur after an assignment \(x = 1\), but only enough that remaining cycles have no negative impact on the worst-case running time. We will now prove the correctness of these replacements. Unfortunately, there is no particular uniting theme of the proof; we essentially just progress and prove correctness case by case.

Lemma 35. The cycle replacements preserve the satisfiability of an Xsat instance \(F\).
Proof. As we pointed out, an even number of the labels have to be set to 0, since a label of 0 means that the values of the associated core variables are different. In the following, when we talk of parity (e.g. odd parity of a label subsequence) we mean the number of inversions on the core variables, i.e. the number of zeros among the labels, not the number of ones. We also generally assume that \( x = 0 \) when proving equalities (and consequently \( a \neq b, c \neq d, e \neq f \)), since \( x = 1 \) implies that all labels are 0, and therefore equal.

If case 1 applies (the cycle is a 4-cycle), then the restriction that two neighbouring core variables may not be set to 1 implies that when a label is set to 1, some neighbouring label is also set to 1. There are four reductions specified in Table 5.1.

1. For a label sequence \( a, b, c, d \), we get that \( a = 1 \) implies \( d = 1 \) (since \( b = 0 \)). Since the case is symmetric, the reduction follows.

2. For a label sequence \( a, b, c, e \), \( a = 1 \) implies \( e = 1 \) since \( b = 0 \), and \( c = 0 \) by parity; \( d = 1 \) and \( f = 0 \) follow. The case \( b = 1 \) is symmetric, and \( a = b = 0 \) implies \( x = 1 \).

3. For a label sequence \( a, c, b, d \), by symmetry it suffices to show that one clause, say \( (d, c_1, c_4) \), is implied by the others. If \( x = 1 \) then \( c_1 \neq c_4 \) by the three remaining inequalities, which is equivalent to the effect of the removed clause, so assume \( x = 0 \). Then, the sequence \( a, c, b \) has a parity different to that of \( c \), so \( d = 1 \) implies \( c_1 = c_4 \) and \( d = 0 \) implies \( (since \( x = 0 \)) \( c_1 \neq c_4 \). We see that \( c_1 = c_4 \) if and only if \( d = 1 \), and if we also eliminate the possibility \( c_1 = c_4 = 1 \), then we are done, and this is easy: \( c_1 = c_4 = 1 \) implies \( a = b = 0 \), which has already been dealt with.

4. For a label sequence \( a, c, b, e \), the assignment \( a = 1 \) implies \( c \neq e \) (by parity and \( b = 0 \)), and \( c \neq d, e \neq f \) imply \( c = f, d = e \). The case \( b = 1 \) is symmetric, and \( a = b = 0 \) implies \( x = 1 \).

If case 2 applies (6-cycle, two labels repeated twice plus \( p,q \) once each), then \( p = q \) by the parity argument (a label that is repeated twice cannot change the parity).
If case 3 applies (6-cycle, one label repeated twice), then we will rely more heavily upon parity arguments.

1. For a label sequence $a, b, c, e, c, f$, the sequence $a, b$ inverts parity and $e \neq f$. If $c_1 = 1$, then $a = 0, b = 1, f = 0, e = 1$, and $c = 1$ since its core variables are 0. If $c_3 = 1$, then $b = c = 0, a = 1$ and $d = 1$. We see that $a = d$ and $b = c$ hold in all cases.

2. For a label sequence $a, c, a, d, b, e$, the label sequence $a, c, a, d$ has odd parity and so must the sequence $b, e$, which implies that $b \neq e$. Since $a \neq b$ and $e \neq f$, we get $a = e$ and $b = f$.

3. For a label sequence $a, c, a, e, b, f$, note that $e$ and $f$ have odd parity together, and so must $b$ and $c$ (since $a$ does not affect the parity). We get $a \neq b \neq c \neq d$.

4. For a label sequence $a, c, a, e, d, f$, the assignment $c = 1$ implies $a = 1$ (since $c = 1, a = 0$ implies $e = f = 0$), and $c = 0$ implies $a = 0$ (since one core variable being neighbour to $a$ is set to 1). It then follows by $a \neq b$ and $c \neq d$ that $b = d$.

5. For a label sequence $a, c, e, a, d, f$, if $a = 1$, then $c = e$ and $d = f$ (since the sequences $c, e$ and $d, f$ must have even parity), and if $a = 0$, then $c = 0$ or $f = 0$, and $d = 0$ or $e = 0$, being equivalent to $c \neq f$ when $x = 0$, so we have $d \neq c \neq f \neq e$.

6. For a label sequence $a, c, e, a, f, d$, we have to verify that either of the clauses $(a, c_1, c_2)$ and $(c, c_2, c_3)$ can be removed. First, if $x = 1$ then the effect of the cycle is six inequalities $c_1 \neq c_2 \neq \ldots \neq c_6 \neq c_1$, from which any one inequality can be removed without changing the result, so we can assume that $x = 0$. If we remove the clause that contains $a$, then the sequence $c, e, a, f, d$ will still have the same parity as $a$, so $c_1 = c_2$ if and only if $a = 1$; we just have to verify that $c_1 = c_2 = 1$ is impossible. Setting $c_1 = c_2 = 1$, we get $c = d = 0$ so that $x = 1$, which has been dealt with.

Likewise, if we remove the clause that contains $c$, then the parity of $e, a, f, d, a$ is different from the parity of $d$. If $c = 1$, then we
get \( d = 0 \) and \( c_2 = c_3 \) is forced, and if \( c = 0 \) then (assuming \( x = 0 \)) \( c_2 \neq c_3 \) is forced; \( c_2 = c_3 \) if and only if \( c = 1 \). Setting \( c_2 = c_3 = 1 \), we get \( a = e = 0 \), and \( f = 0 \), but then \( x = 1 \).

If case 4 applies, then \( x = 0 \) means that exactly 3 labels are set to 0, which is a contradiction.

5.1.2 Interfaces

Next, we give the concepts of interfaces and interface replacements. In short, an interface is what connects one part of the formula, for instance a neighbourhood, to the rest. Thus, the only influence the neighbourhood has on the satisfiability of the formula as a whole can be summarised in its influence on the allowed values of the variables in the interface. When the number of variables in the interface is small, the neighbourhood can be replaced by an equivalent formula containing fewer variables, connecting to the same interface. (The concept is related to the multiplier reduction we will use in Chapters 7 and 8.)

For example, consider the clauses \((a, b, c), (a, d, s), (b, d, f), (c, g, h)\). If these represent all occurrences of variables \(a, \ldots, d\) and \(s\), then the interface of the clauses \((a, b, c), (a, d, s)\) is the clauses \((b, d, f), (c, g, h)\), and the external parts of these clauses are the occurrence of \(f\), and the subclause \((g, h)\). In this case, the first two clauses can be dropped entirely: for every value of \(f\), and whether \(g = h = 0\) or \(g \neq h\), there exists some assignment to variables \(a, \ldots, d\) and \(s\) that satisfies the four clauses.

**Definition 36.** The interface of a set of clauses \(S \subseteq F\) is every clause \(C \in F\) (\(C \notin S\)) that contains some variable that appears somewhere in \(S\), and some variable that does not. The external part of such a clause \(C\) is the subclause whose variables do not appear anywhere in \(S\). We say that a set of variables and subclauses \(I\) forms the external interface of a set of variables \(V\) (\(|V| < n/2\)) if, for every clause \(C\) where some, but not all, variables in \(C\) occur in \(V\), the external part of \(C\) is included in \(I\).
Definition 37. *Interface replacement* applies when some $I$ forms the external interface of a set of variables $V$, and one of the following holds:

- $I$ consists of a single subclause or variable and $|V| > 1$, or
- $I$ consists of two subclauses or variables and $|V| > 3$, or
- $I$ consists of three subclauses consisting of pairs on three variables $p, q, r$ (e.g. $(p, q)$, $(\bar{p}, r)$, $(\bar{q}, \bar{r})$ where $\bar{x}$ is $x$ or $\bar{x}$) and $|V| > 3$.

If such a case applies, then let $F_0$ be the neighbourhood of $V$ (i.e. all clauses containing any occurrence of one or more variables of $V$). Find the restrictions that $F_0$ impose on $I$ and replace $F_0$ by an equivalent part with fewer variables, in the following manner:

- **Case 1:** If $I$ consists of a single variable $v$, then check the satisfiability of $F_A = F_0[v = 1]$ and $F_B = F_0[v = 0]$. If $I$ consists of a single subclause $C$, then check the satisfiability of $F_0$ with $C$ replaced by 1, referred to as $F_A$, and of $F_0$ with $C$ replaced by 0 (i.e. $F_0$ with $C$ shortened away), referred to as $F_B$. As $C$ can contain only one or zero true literals, we refer to these cases as $C = 1$ and $C = 0$, and talk of assigning values to $C$, for the extent of this definition. Restrictions to $v$ or $C$ equivalent to those of $F_0$ can be implemented as follows:

  - If $F_A$ is satisfiable but $F_B$ is not, then assign $v = 1$ if $I$ is a variable, or include the clause $(C)$ if $I$ consists of the subclause $C$. The latter is referred to as an implementation of $C = 1$.
  
  - If $F_B$ is satisfiable but $F_A$ is not, then assign $v = 0$ if $I$ is a variable, or assign $C = 0$, i.e. assign $l = 0$ to every $l \in C$, if $I$ consists of the subclause $C$.
  
  - If both are satisfiable, then do nothing if $I$ is a variable, and include a clause $(C, s)$ for a fresh singleton variable $s$ if $I$ consists of the subclause $C$. The latter is referred to as an implementation of an unrestricted subclause $C$.
– If neither is satisfiable, then we have a contradiction. Include an empty clause in \( F \) to signal this.

– Case 2: If \( I \) consists of two variables \( v_1, v_2 \) or subclauses \( C, D \), then check the satisfiability of the four possible ways to replace them by 1 or 0. Restrictions to the members of the interface equivalent to those of \( I \) and \( F_0 \) can be implemented as follows:

– Any restriction expressible as a combination of assignments and unrestricted variables or subclauses can be implemented as in the previous case.

– The restriction that \( v_1 = v_2 \) can be implemented by direct replacement. The restriction that \( C = D \) can be implemented by clauses \((C, v), (D, v)\) for a fresh variable \( v \).

– The restriction that \( v_1 \neq v_2 \) can be implemented by direct replacement. The restriction that \( C \neq D \) can be implemented as a clause \((C, D)\).

– A single excluded combination of values to \( v_1, v_2 \) can be implemented by a clause \((\bar{v}_1, \bar{v}_2, s)\) for a fresh singleton variable \( s \), under the appropriate pattern of negations. A single excluded combination of values to \( C, D \) can be implemented by clauses \((C, c), (D, d), (\bar{c}, \bar{d}, s)\) for fresh variables \( c, d, s \), under the appropriate pattern of negations. (Note that in the presence of negations, resolution will apply to \( c \) or \( d \).)

– Case 3: If \( I \) consists of the pairs \((p, q), (p, r), (q, r)\), then there are four options: exactly one of \( p, q, r \) is true, or \( p = q = r = 0 \). If \( F_0 \) is satisfiable under all options, then replace it by \((p, q, s_1), (p, r, s_2), (q, r, s_3)\) where \( s_i, 1 \leq i \leq 3 \), are fresh singletons. If only \( p = q = r = 0 \) is impossible, then replace \( F_0 \) by \((p, q, r)\). Otherwise, the constraints on \( p, q, r \) are implementable through assignments (e.g. if \( p = 0 \) is forced, then assign \( p = 0 \)), equalities or inequalities (e.g. if \( p = q \) is forced, then replace \( p = q \)), or one excluded combination for a pair of variables (e.g. \( p = q = 0 \) is excluded by a clause \((\bar{p}, \bar{q}, s)\) for a fresh singleton \( s \)).
- Case 4: If \( I \) consists of the pairs \((p, q), (p, r), (q, \bar{r})\), then there are four options: \( p = 1 \) and \( q = r = 0 \), or \( p = 0 \) and \( q = r = 1 \), or \( p = q = 0 \) and \( r = 1 \), or \( p = q = r = 0 \). If \( F_0 \) is satisfiable under all options, then replace it by \((p, q, s_1), (p, r, s_2), (q, \bar{r}, s_3)\) where \( s_i \), \( 1 \leq i \leq 3 \) are fresh singleton variables. All other constraints are implementable through assignments, equalities or inequalities, or one excluded combination for a pair of variables.

We do not need to consider the options with two or three negations among the pairs, because these admit other, direct reductions, e.g. the clauses in the interface immediately imply an assignment or replacement among \( p, q, \) and \( r \).

Now we will prove that the interface replacements are correct, and possible to perform as defined.

**Lemma 38.** The interface replacements preserve satisfiability, and the result of performing one is a formula with fewer variables, and as used in the algorithm XShort, it is possible to detect whether interface replacements applies in polynomial time (in the sum of \( n \) and \( m \)).

**Proof.** In cases 1 and 2, where the interface may contain arbitrarily long subclauses, we get a few technical details to deal with.

Any time a subclause \( C \) occurs in the interface, the restriction that \( C \) may not be oversatisfied is kept as \( C \) always occurs inside some clause in the replacement (unless \( C = 0 \) is assigned). Under this constraint, every subclause can only contain either one or zero true literals, represented as \( C = 1 \) and \( C = 0 \), and any further restrictions result in restrictions on the values of subclauses \( C \) and \( D \).

No increased length bonus occurs, since each replacement is equivalent to some replacement using only short clauses plus clauses such as \((C, v), (D, w)\) containing \( C \) or \( D \) plus exactly one variable, where each of \( C \) and \( D \) occurs exactly once. Clauses of these lengths must certainly exist in \( F \).

Finally, regarding the “building blocks” of the replacements in these cases, remember that \((C, s)\) for a singleton \( s \) and any \( C \) is equivalent to at most one literal of \( C \) being true. The replacements for a
free subclause and an assignment are \((C, s), (C)\) and \(C = 0\); all easy to see. The replacements for equality and inequality are \((C, D)\) (equivalent to \((C, v), (D, \bar{v})\)), and \((C, v), (D, v)\), which are correct since \((C, v)\) is equivalent to \(C\) not being oversatisfied (i.e. one or zero literals of \(C\) are true) plus \(C \neq v\) (i.e. \(v\) is true if and only if no literal of \(C\) is true). As for the implementation of one excluded combination, any clause \((l_1, l_2, s)\) for a singleton \(s\) is equivalent to excluding \(l_1 = l_2 = 1\).

None of these complications apply to cases 3 and 4, since we deal here with direct restrictions on the possible values of variables. The rest of this proof progresses according to the cases of the definition.

If case 1 applies, i.e. \(I\) is a single entity (variable or subclause), then it is obvious that the only options are an unconstrained entity, an assignment, or a contradiction. At most one new variable is used.

If case 2 applies, i.e. \(I\) is two entities (call the entities \(C\) and \(D\), whether subclauses or variables), then consider the number of possibilities that are allowed.

- If all four assignments are allowed, then there are no extra constraints, and two fresh variables are used.
- When three ways are allowed, only one is excluded, and it is clear that the definition covers all such cases, using up to three fresh variables.
- If two ways are allowed, then either one entity has its value decided (assigned) and the other is unconstrained, using one variable, or the entities must have equal or different values, which is implemented using at most one fresh variable.
- When one way is allowed, there are two assignments requiring no fresh variables, and a contradiction halts the execution.

In cases three and four, when no extra constraint occurs, we use three fresh variables \(s_1, s_2, s_3\), and otherwise we use at most one extra variable:

- In case three, the possibility that \(p = q = r = 0\) is the only excluded assignment is handled, and any other case results in some
variable being assigned false, leaving only two possibly unassigned variables. The possible restrictions on these two variables are the same as those already mentioned for two generic entities: no restrictions, one excluded combination (requiring one variable), replacement (equal or unequal), further assignment, or contradiction.

- In case four, a restriction equivalent to a clause $C = (\bar{p}, \bar{q}, r)$ is not possible: the possible assignments are $p = 1$ and $q = r = 0$; $p = 0$ and $q = r = 1$; $p = q = 0$ and $r = 1$; and $p = q = r = 0$. Since $p = 1$ and $q = 1$ each only occur once among these options, the clause would have to be $C = (p, q, r)$, but then no negation for $r$ works.

- Other options in case four for up to three assignments to three variables include assignment or replacement, followed by the possible restrictions on two variables, or a contradiction, and nothing else. Let one assignment be $l_p = l_q = l_r = 0$, where $l_p, l_q, l_r$ are literals of $p, q, r$, respectively, and consider a second assignment, assuming that no assignment or replacement occurs:

  - If another assignment assigns two zeros to $l_p, l_q, l_r$, say $l_p = l_q = 0, l_r = 1$ (without loss of generality), then for any choice of third assignment either $p = q$ or one of $l_p$ and $l_q$ is always 0.

  - Otherwise, let another assignment be $l_p = 0, l_q = l_r = 1$; the third assignment must contain $l_p = 1$ and $q \neq r$, leaving us with $l_p = l_q = l_r = 0$; $l_p = 0, l_q = l_r = 1$; and $l_p = 1, l_q = 0, l_r = 1$; this is equivalent to $(l_p, l_q, \bar{l}_r)$.

As for two or three negations not being possible, if the pairs are $(p, q), (p, r), (\bar{q}, \bar{r})$, then $p = 1$ oversatisfies the last pair; if the pairs are $(p, q), (\bar{p}, r), (q, \bar{r})$, then $q = 1$ oversatisfies the middle pair; and if the pairs are $(p, q), (\bar{p}, r), (\bar{q}, \bar{r})$, then $p = q$ implies $p = q = 0$ and there is a contradiction regarding the value of $r$. 

Finally, we show the polynomiality. Checking for cases when the interface contains only single variables is obviously possible, and an interface which is a subclause can be detected by checking for each clause $C$ whether $F$ with the clause $C$ removed is connected (and finding the appropriate part of the connecting clause after that is easy). Also, when the applicability of an interface replacement is checked in the algorithm, two clauses share at most one literal, so if there is an interface of two subclauses, then this can be detected by removing two clauses $C$ and $D$.

5.1.3 The Algorithm

One more definition is needed before we present the algorithm. This is a definition which will be used to reason about the sparseness of an instance.

**Definition 39.** A dense clause is a clause $(a, b, c)$ where $d(a) = d(b) = d(c) = 2$ and $a$, $b$, and $c$ all appear with some heavy variable.

The first algorithm we present, XMatch, is a helper routine used for deciding satisfiability for an instance where no variable occurs more than twice. It uses a reduction to matching, first presented by Porschen et al. [68]; our version is from Dahllöf (where it is called MatchDecide) [13].

**Algorithm 40.** XMatch($F$):

1. If there are any non-pure variables, then apply resolution to them.

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 $S$ (i.e. zero, one, or two.)
4. Find a maximum weighted matching in $G$. If that weight is equal to $|S|$, then return 1, otherwise 0.

Algorithm ends.

Note that the empty formula will produce the answer 1.

**Lemma 41.** [Lm. 3 of [13]] For an instance $F$ of XSAT such that all variables have at most degree 2, XMatch($F$) will in polynomial time return ‘Yes’ iff $F$ is satisfiable and ‘No’ otherwise.

Now, we can finally present our main algorithm XShort.

**Algorithm 42.** XShort($F$):

0. If $F = \emptyset$, then return 1. If $\emptyset \in F$, then return 0.

1. If $F$ consists of two separate subformulae $F_1$ and $F_2$ with no variables in common, then return XShort($F_1$) $\land$ XShort($F_2$).

2. If $(l) \in F$, then return XShort($F[l = 1]$).

3. If $(l, l) \in F$, then return 0. If $(l, \overline{l}) \in F$, then drop this 2-clause and return XShort($F$). If otherwise $(l_1, l_2) \in F$, then return XShort($F[l_1 \neq l_2]$).

4. If $C \subseteq D$ for $C, D \in F$, then let $F' = F - D$ and return XShort($F'[D - C] = 0$).

5. If there is a clause $(l, l, C)$ in $F$, then return XShort($F[l = 0]$).

6. If there is a pure variable $a$ such that every clause containing $a$ also contains a singleton, then return XShort($F[a = 0]$).

7. If there are variables $a$ and $b$ such that $a$ is a $(k, 1)$-variable, $b$ is a pure variable, $(\overline{a}, b, C) \in F$ is the only co-occurrence of the variables $a$ and $b$, and every other occurrence of $a$ or $b$ is in a clause with a singleton, then return XShort($F[b = 0]$).
8. If there are clauses \((l_1, l_2, C), (\bar{\bar{l}}_1, \bar{\bar{l}}_2, D)\) in \(F\), then let \(F' = F[l_1 = 0]\). If there are clauses \((l_1, l_2, C), (\bar{l}_1, \bar{l}_2, D)\) in \(F\), then let \(F' = F[C = D = 0]\). In both cases, return \(\text{XShort}(F')\).

9. If there are clauses \((a, C), (\bar{a}, D)\) where \(a\) is a \((1, 1)\)-variable, then let \(F'\) be \(F\) with these clauses replaced by \((C, D)\) and return \(\text{XShort}(F')\).

10. If there are clauses \((A, C), (B, C)\) where \(A \cap B = \emptyset\), \(|C| > 1\), then a replacement applies. If \(|A| = |B| = 1\), then let \(A = l_1\) and \(B = l_2\), and return \(\text{XShort}(F[l_1 = l_2])\); otherwise, let \(x\) be a fresh variable, and let \(F'\) be \(F\) with these clauses replaced by \((A, x), (B, x), (\bar{x}, C)\). Return \(\text{XShort}(F')\).

11. If there is any variable \(v\) in \(F\) such that \(v = b\) leads to a contradiction (after the application of cases 0–10), then return \(\text{XShort}(F[v = (1 - b)])\). If there are any variables \(v, w\) in \(F\) that either appear together in a clause, or have a common neighbour, and if \(v = w\) leads to a contradiction (after the application of cases 0–10), then return \(\text{XShort}(F[v \neq w])\), and if \(v \neq w\) leads to a contradiction, then return \(\text{XShort}(F[v = w])\).

12. Replacement cases:
   a) If any cycle replacement is applicable (see Definition 33), then apply the corresponding reduction.
   b) If clauses \((x, a, b), (\bar{x}, c, d), (a, c, e), (b, d, \bar{e})\) exist, then remove any one of the two latter clauses.
   c) If any interface replacement is applicable (see Definition 37), then perform it.

In either case, let the result be \(F'\) and return \(\text{XShort}(F')\).

13. If \(d(F) > 3\), then pick a variable \(x\) of maximum degree and return \(\text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0])\).

14. If \(F\) contains some \((2, 1)\)-variable, then let \(x\) be a \((2, 1)\)-variable with a maximum number of neighbours. Return \(\text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0])\).
5. One-in-three Satisfiability

15. If $F$ contains some heavy variable with more than six neighbours, then let $x$ be a heavy variable with a maximum number of neighbours, which as a secondary criterion avoids situations where several clauses in the interface of the neighbourhood of $x$ have identical external parts. Return $\text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0])$.

16. If there is a heavy variable $x$ occurring in clauses $(x, a, b), (x, c, d), (a, c, E)$ for some $E$ where $d(b) \geq d(d)$, then we know that $d(b) > 1$. Return $\text{XShort}(F[b = 1]) \lor \text{XShort}(F[b = 0])$.

17. Let $x$ be a heavy variable that maximises the sum of the degrees of the neighbours of $x$. If this sum is at least 12, then return $\text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0])$.

18. If there is a clause $(x, a, s)$ where $x$ is heavy, $d(a) = 2$, $s$ is a singleton, and $a$ occurs with a variable $b$ such that every other occurrence of $b$ is in a clause with a singleton, then return $\text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0])$.

19. If there is a clause $(x, a, b)$ where $x$ is heavy, $d(a) = d(b) = 2$, the sum of degrees of all neighbours of $x$ is 11, and $a$ is a neighbour of another heavy variable $y$, then return $\text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0])$.

20. If there are clauses $(x, a, p)$ and $(a, q, s)$ where $x$ is heavy, $d(a) = 2$, $d(p) > 1$, $x$ occurs with two singletons, and $s$ is a singleton, then return $\text{XShort}(F[q = 1]) \lor \text{XShort}(F[q = 0])$.

21. If there are clauses $(x, a, p), (a, q, A)$ where $x$ is heavy and occurs with two singletons, $d(a) = 2$, $d(p) > 1$, and any other occurrence of $q$ is with a singleton, then return $\text{XShort}(F[p = 1]) \lor \text{XShort}(F[p = 0])$.

22. If there is a heavy variable $x$ such that the sum of degrees of all neighbours of $x$ is 11, and either at most one light neighbour of $x$ appears in a non-dense clause (see Definition 39), or $x$ has only one neighbour that is a singleton and at most three
5.1. Algorithm Preliminaries

light neighbours of \( x \) appear in non-dense clauses, then return
\[ \text{XShort}(F[x = 1]) \lor \text{XShort}(F[x = 0]). \]

23. Explicitly enumerate all assignments to the heavy variables,
avoiding the combination \( x = y = 1 \) when \( x \) and \( y \) are neigh-
bours, and check using XMatch whether any of the resulting
light instances is satisfiable.

Algorithm ends.

The trend as the algorithm progresses through the cases is one of
relative sparsification: we eliminate in turn cases where the degree is
another than (1, 0), (2, 0) or (3, 0), cases where heavy variables have
a common neighbour, and certain cases of co-occurrences of variables
\( v, w \) with a heavy neighbour each, to finally reach a case where enu-
meration on the heavy variables yields few enough light instances.

As mentioned, the algorithm is designed to be used primarily for
\( X3\text{sat} \) instances, but through the resolution step in case 9 it creates
longer clauses. It balances this by a cost in the complexity measure:
the measure is \( f(F) = n(F) + \lambda(F) \), where \( \lambda(F) \) is the length bonus,
as previously defined. Each long clause increases the weight of the
instance by +1 for each member after the third. By this uniform
cost, some of the analysis is simplified while instances with a higher
number of long clauses (and particularly with clauses longer than 4)
get a too high weight.

Now we prove the correctness of the algorithm.

**Lemma 43.** XShort(\( F \)) decides the satisfiability of an Exact Satisfi-
ability instance \( F \).

**Proof.** Cases 0–5 and 8 contain obviously required assignments or
replacements. Cases 6 and 7 can be seen to preserve satisfiability.
Cases 9 and 10 are variants of resolution and backwards resolution,
as used in Chapter 4, and can easily be seen to be correct. Case 11
is correct by the correctness of previous cases.

In case 12, cycle replacement and interface replacement are correct
by Lemmas 35 and 38. The reduction in subcase b is also correct:
5. One-in-three Satisfiability

the two first clauses imply the clause \((a, b, c, d)\) which with the third clause implies \(e = (b, d)\), i.e. a clause \((b, d, \bar{e})\), and likewise with the last clause implies \(e \neq (a, c)\), i.e. a clause \((a, c, e)\).

The actual branchings used in cases 13–22 are all trivially correct. The statement in case 16 that \(b\) is not a singleton follows from the non-applicability of interface replacement. Finally, case 23 is obviously correct and complete, given the applicability of the matching method.

5.2 Many Neighbours and Non-pure Cases

This section deals with the cases up to case 16, which includes all cases where any heavy variable has more than six neighbours, and all cases when \(F\) contains some non-pure variable. First, we will prove a base branching that gives reasonable bounds in many cases.

Lemma 44. Let \(F\) be a fully reduced XSAT formula, and \(x\) be a heavy \((k_1, k_2)\)-variable in \(F\) where the literals \(x\) and \(\bar{x}\) have \(a\) respectively \(b\) neighbours. Branching on \(x\) admits a branching of

\[
\tau(2a - 2k_1 + k_2 + 1 + e_1, 2b - 2k_2 + k_1 + 1 + e_2),
\]

where:

1. If \(k_2 = 0\), then \(e_1 \geq 3\)
2. If \(k_2 = 1\), then \(e_1 \geq 2\), and if \(\bar{x}\) appears in \(C = (\bar{x}, D)\), then \(e_2 \geq 2\) if there are at least two occurrences of a variable or variables of \(D\) not in the clause \(C\), and \(e_2 \geq 1\) otherwise
3. Otherwise, \(e_1, e_2 \geq 2\)

Proof. The parts that do not include \(e_1\) and \(e_2\) are easy: the clauses of the variable \(x\) lead to assignments, shortenings, and replacements due to 2-clauses, and since no pair of variables co-occur twice, all 2-clauses contain variables different from those counted as assigned. For the values of \(e_1\) and \(e_2\), we progress according to the value of \(k_2\).
5.2. Many Neighbours and Non-pure Cases

When \( k_2 = 0 \), \( x \) is pure and every neighbour of \( x \) is assigned when \( x = 1 \). If \( e_1 = 2 \), then either the neighbourhood of \( x \) has an interface of only two clauses, or of three clauses where the external parts are pairs on the same three variables, both of which are covered by an interface replacement.

When \( k_2 = 1 \) and \( x = 1 \), we first prove \( e_1 \geq 2 \). There are at least two clauses that contain the literal \( x \), and at least two occurrences of variables from these clauses in another clause.

- If all further occurrences of these variables are in the same clause, then the external part of this clause and the variable \( x \) form the interface for at least four variables (note that “external” in this part of the proof is relative to the clauses containing the literal \( x \)).

- Likewise, if the external part of each clause containing some further occurrence of these variables consists of the same variable \( v \), then \( v \) and \( x \) form the interface for at least four variables.

Otherwise, we can assume that there are at least two such clauses with different external parts, and we have to prove that \( f(F) \) reduces by at least three points due to these clauses and the clause containing \( \bar{x} \). If the clause containing \( \bar{x} \) is long, then the result is immediate. Otherwise, we have one replacement due to this clause, say \( a \neq b \) (without making any assumptions about the signs of the occurrences of \( a \) and \( b \)), and two shortened clauses leading to either assignments or replacements. We divide into cases according to the effects of such shortened clauses.

1. If there exists an assignment to another variable than \( a \) or \( b \), or a replacement not involving \( a \) or \( b \), or if one of the clauses is long, then the result is also immediate.

2. If we have only assignments to both \( a \) and \( b \), then these variables form the interface of every neighbour of the literal \( x \) plus the variable \( x \).
3. If we have a 3-cycle of replacements, say \( a \neq c \) and \( b = c \), then the variables \( a, b \) and \( c \), occurring in three pairs, form the interface for \( x \) plus the \( 2k_1 \geq 4 \) neighbours of the literal \( x \) which admits an interface replacement (if not an earlier reduction).

4. In all other cases, \( e_1 \geq 2 \).

As for \( e_2 \), any occurrence of a variable \( a \) of \( D \) is in a clause \((a, E)\) or \((\bar{a}, E)\) where \( E \) shares no variable with \( D \). There must exist at least one such clause, and since the clauses containing the variable \( x \) do not overlap, if \((E)\) is a 2-clause then it is a new replacement, and otherwise we get a length bonus, ensuring \( e_2 \geq 1 \) in all cases. To prove the \( e_2 \geq 2 \) part, we need to consider the possible clauses closer.

1. If \( a \) occurs under different negations in \( C \) and \( E \), then at least two assignments are made to variables in \( E \), and these variables have not been the target of a replacement.

2. If some literal in \( E \) occurs under the opposite sign with the literal \( x \), then \( a = 1 \) is detected as impossible by case 11: setting \( a = 1 \) forces \( x = 1 \), which forces the mentioned literal to become true in \( E \), causing an oversatisfied clause.

3. If \( E \) contains only literals that occur under the same negation with the literal \( x \), then let \( b \) be another variable that occurs in \( D \) (assume without loss of generality that \( b \) is a positive occurrence). Case 11 detects \( b = 1 \) as impossible: both \( a = 0 \) and \( x = 1 \) are implied, setting every literal in \( E \) to false, leaving an empty clause.

4. If there is a second clause \((a, E')\), then \( E \) and \( E' \) share no variables and do not only contain neighbours of \( x \), and \( e_2 \geq 2 \) is obvious.

5. Otherwise, let there be a second clause \((b, E')\) (where \( b \) occurs in \( D \)). The remaining potentially problematic case is when both \( E \) and \( E' \) are short, and form a 3-cycle of replacements together with one pair due to the \( x = 0 \) assignment, but then, assuming
that the latter pair is \((c, d)\), we can assume that \(E = (c, e)\), 
\(E' = (d, \bar{e})\) for a variable \(e\), in which case one of \((a, E)\) and 
\((b, E')\) is dropped by case 12b.

We see that \(e_2 \geq 2\) when the two clauses exist and no reduction applies.

When \(k_1, k_2 \geq 2\), then as before there exist at least two clauses 
with separate external parts (where “external” is relative to the neighbour-
bours of the literal \(x\) (or \(\bar{x}\)), not the variable \(x\)). It is clear that 
\(e_1, e_2 \geq 1\); we will show that \(e_1 \geq 2\).

1. If there exists a short clause that contains one neighbour of \(x\) 
and two neighbours of \(\bar{x}\), then let the clause contain variables 
\(a, c\) and \(e\), where the literals \(a\) and \(c\) occur with \(\bar{x}\) and \(e\) with \(x\).

(a) If the clause contains \(\bar{e}\), then both assignments to \(a\) and \(c\) 
contribute towards \(e_1\).

(b) If the clause contains \(\bar{a}\), then \(e = 1\) implies \(a = 0\), oversat-
isfying the clause, which is caught by case 11.

(c) Finally, if the clause is \((a, c, e)\) then let \(e'\) be a neighbour 
of literals \(e\) and \(x\) (e.g. through a clause \((x, e, e')\)). An 
assignment \(e' = 1\) implies \(a = c = e = 0\), leaving an empty 
clause, and is also caught by case 11.

2. If no such short clause occurs, then there are two options for 
\(e_1 = 1\): the clauses may imply assignments such as \(a = 1\) and 
\(b = 0\) where a clause \((\bar{x}, a, b)\) occurs, or the clauses may result 
in a 3-cycle of replacements involving \(a\), \(b\), and some variable \(c\), 
where a clause \((\bar{x}, a, b)\) occurs. In the first case, \(b = 1\) implies 
\(x = 1\) which implies \(b = 0\), and this is caught by case 11.

3. In the second case, when the clauses result in a 3-cycle of re-
placements involving \(a\), \(b\) and some variable \(c\), a clause \((\bar{x}, a, b)\) 
occurs and there is (without loss of generality) a replacement 
\(a \neq c\). In addition, either a pair \((b, c)\) or a pair \((b, \bar{c})\) occurs, 
and there is no other consequence of \(x = 1\).
(a) If the former pair occurs, then assume without loss of generality that the clause that contains the pair \((a, c)\) is \((e, a, c)\) and that \(e\) occurs in a clause \((x, e, e')\). An assignment \(e' = 1\) implies \(e = a = b = c = 0\), leaving an empty clause, so case 11 applies.

(b) If the latter pair occurs, then assume without loss of generality that the clauses that contain the pairs \((a, c)\) and \((b, \overline{c})\) are \((a, c, e)\) and \((b, \overline{c}, f)\). Since only two clauses exist with neighbours of \(x\), also let clauses \((x, e, e')\) and \((x, f, f')\) occur. Now, this is also caught by case 11: \(e \neq f'\) implies \(x = 0, a = b = 0, e = f\), leaving pairs \((e, e)\) and \((c, \overline{e})\), and triggering a contradiction.

Otherwise, \(e_1 \geq 2\) (and since our case is symmetrical, \(e_2 \geq 2\) as well).

The next lemma provides the branching numbers for cases 13 and 14 of the algorithm.

**Lemma 45.** If \(d(x) \geq 4\), then branching on \(x\) yields a branching dominated by \(\tau(12, 5) < 1.0908\) or a more balanced version thereof. If \(x\) is a \((2, 1)\)-variable which appears in some long clause, then branching on \(x\) yields a branching dominated by \(\tau(10, 6) < 1.0927\) or a more balanced version thereof. If \(x\) is a \((2, 1)\)-variable appearing only in short clauses, then branching on \(x\) yields a branching dominated by \(\tau(9, 6) < 1.0983\) or a more balanced version thereof.

**Proof.** All results for \(d(x) > 3\) and for \(d(x) = 3\) with \(x\) in some long clause are achieved by plugging values into Lemma 44. When \(x\) is a \((2, 1)\)-variable appearing in only short clauses, let the clauses be \((x, a, b), (x, c, d), (\overline{x}, e, f)\) (without making any assumptions about whether the positive or negative literals of variables \(a, \ldots, f\) have the most occurrences). If \(e\) and \(f\) have at least two further occurrences, then by Lemma 44 we have a branching dominated by \(\tau(8, 7) < 1.0970\). Otherwise, we have the difficult part of the lemma; assume without loss of generality that \(a\) and \(c\) are non-singletons, that \(e\) is a \((2, 0)\)-variable, and that \(f\) is a singleton. In the branch...
5.2. Many Neighbours and Non-pure Cases

\(x = 0, \Delta f \geq 6\) by Lemma 44, and when \(x = 1\) we count assignments to \(a, \ldots, d\), and \(f\) is removed (since it then only appears in the pair \((e, f)\)). We show that \(f(F)\) must be reduced by at least three more points, showing that a branching dominated by \(\tau(8, 7)\) or \(\tau(9, 6)\) will occur.

1. No clause exists that only contains neighbours of \(x\): if a clause 
\((a, c, e)\) existed, then case 11 would assign \(f = 0\); if a clause contained \(e\) and \(\bar{a}\) (or \(\bar{c}\)), then case 11 would assign \(e = 0\); and if a clause contained \(a\) (or \(c\)) and \(\bar{e}\), then case 11 would assign \(a = 0\) (resp. \(c = 0\)). If a clause contains \(\bar{a}\) and \(\bar{c}\), then \(x = 0\) is assigned.

2. If a clause that contains \(e\) and one other neighbour of \(x\) appears, say \((a, e, P)\), then \(e\) has no more occurrences. The clause reduces to \((e, P)\).

   (a) If \(|P| = 1\) (say \(P = p\)), then this is a replacement that cannot be part of a cycle of replacements (since \(f\) is a singleton). At least two other clauses exist that contain neighbours of \(x\). If one of these clauses does not lead to an assignment, then \(\Delta f \geq 9\) in this branch, and if only assignments are made (and not replacements or shortenings of long clauses), then we know that there are at least two variables assigned by these clauses which are not \(e\).

   (b) If \(|P| > 1\), then \(a = 0\) leads to a shortening of a long clause, and \(\Delta f\) increases by at least two points due to the two other clauses (no cycle of replacements is possible when \(f\) is a singleton, and if only assignments are made, then they must be to different variables, unless the neighbourhood of \(x\) is to have a small interface).

In either case, we get a branching dominated by \(\tau(9, 6)\) if such a clause occurs.

3. If any variable of \(a, \ldots, d\) occurs in any long clause, then the result also holds.
5. One-in-three Satisfiability

(a) If setting $a = \ldots = d = 0$ reduces $f(F)$ by at least three points due to shortenings of long clauses, then $\Delta f \geq 9$.

(b) If $f(F)$ is decreased by two points due to shortenings of long clauses, then since $a, \ldots, d$ have at least three occurrences, there must be one replacement or assignment, leading to $\Delta f \geq 9$.

(c) If $f(F)$ is decreased by only one point due to a shortened long clause and only one clause is shortened below length 3, then this clause must be made into a 1-clause, and the external variable of this clause cannot be a singleton (by connectivity) and must have some neighbour not among $a, \ldots, d$. We get another reduction of $f(F)$ due to a shortened long clause, replacement, or another assignment. Again, $\Delta f \geq 9$.

(d) If $f(F)$ is decreased by only one point due to a shortened long clause and at least two clauses are shortened below length 3, then we have two replacements which are not for the same pair of variables, or a replacement plus an assignment, and neither involves $e$ or $f$.

In all cases, if any variable of $a, \ldots, d$ occurs in any long clause, then we get a branching dominated by $\tau(9, 6)$.

4. If there is a clause $(a, c, p)$ or $(\overline{a}, c, p)$, then $p$ is assigned some value. Consider the other 2- or 1-clauses created due to $a = \ldots = d = 0$.

(a) If these lead to two replacements, then we get $\Delta f \geq 9$.

(b) If they lead either to an assignment to a variable $q$ plus a replacement for a pair other than $p$ and $q$, or to assignments to two variables $q$ and $r$, then we get $\Delta f \geq 9$.

(c) If they lead to one replacement only, then $p$ must have an external neighbour which, if the replacement involves $p$, is not involved in the replacement, by connectivity.
5.2. Many Neighbours and Non-pure Cases

(d) If they lead to an assignment to a variable $q$, plus optionally a replacement to the pair $p$ and $q$, then $p$ (or $q$) must have an external neighbour.

In all cases, $\Delta f \geq 9$ unless $x$ and $p$ form an interface for $a, \ldots, d$, in which interface replacement applies.

5. If there is a clause such as $(\bar{a}, p, q)$ then $p$ and $q$ are assigned, and in addition $a$ occurs in another clause where $p$ and $q$ do not occur, leading to $\Delta f \geq 9$.

6. Otherwise, we have only replacements.

(a) If there are at least four replacements, or three replacements not in a cycle, then at least three variables must disappear and $\Delta f \geq 9$.

(b) If there is a cycle of three replacements, then assume that $a$, $b$ and $c$ are non-singletons (all must be light). There must be a variable $p$ such that both $a$ and $b$ are neighbours of $p$, and in the $x = 0$ branch this leads to a clause where the variable $p$ occurs twice, yielding a branching dominated by $\tau(8,7)$.

In the next lemma, our instance is pure but we are branching on a heavy variable occurring in a long clause.

**Lemma 46.** Case 15 of XShort results in a branching dominated by $\tau(13,4) < 1.0955$.

**Proof.** If $x$ has at least 8 neighbours, then the base branching gives us a branching dominated by $\tau(14,4)$. If $x$ has exactly 7 neighbours, then assume that $x$ appears in clauses $(x, a, b, c)$, $(x, d, e)$, and $(x, f, g)$, where at least five variables (assumed to be $a$, $b$, $d$, $f$ plus one more) are not singletons. Setting $x = 1$ removes 8 variables and shortens the long clause, so we need to show that the further effects of setting $a = \ldots = g = 0$ reduce $f$ by at least 4 points. Remember that
no variable in \( F \) occurs negated and that no clause can contain only neighbours of \( x \). Let \( C, D \) and \( E \) be three clauses in the interface of the neighbourhood of \( x \), with external parts \( P, Q \) and \( R \). Pick \( C, D \) and \( E \) so that the external parts are all different (which is possible unless interface replacement applies). We first deal with cases where some additional length bonus occurs.

1. If setting \( a = \ldots = g = 0 \) reduces \( f \) by at least three points due to shortened clauses (other than \((x, a, b, c)\)), then either \( \Delta f \geq 13 \) due to length bonuses alone, or some external part has length at most 2 and reduces \( f \) further.

2. If the reduction due to extra length bonuses is two, then we still have at least three occurrences of the variables \( a, \ldots, g \) to account for. If the external parts cause at least two replacements, one replacement and some assignment, or two different assignments, then \( \Delta f \geq 13 \).

3. If two clauses turn into identical 1-clauses when \( a = \ldots = g = 0 \), say the 1-clause \((r)\), then either \( r \) occurs with another external variable or only two subclauses, say \( P \) and \( Q \), connect the neighbourhood of \( x \) with the rest of the formula.

4. If the reduction due to extra length bonuses is one, then we have at least four occurrences of the variables \( a, \ldots, g \) to account for. We divide into cases according to the number of replacements.

   (a) If at least three replacements occur, then all three must count (unless we hit a contradiction).

   (b) If two replacements occur, then there must also be some assignment, and these three effects all count.

   (c) If one replacement occurs, and an assignment to a variable which is either not involved in the replacement or which has an external neighbour not involved in the replacement, then let the assignment be \( r = 1 \). Again either only two clause-parts connect the neighbourhood of \( x \) to the rest of the formula, or \( r \) has some external neighbour, which is
assigned 0 in an assignment that counts as another point of reduction.

(d) If one replacement \( q \neq r \) occurs, and an assignment to \( r \) whose only external neighbour is \( q \), then either the assignment \( q = 0 \) causes a second replacement involving another external variable, or only the clause-part \( P \) connects the neighbourhood of \( x \) to the rest of the formula.

(e) If no replacement occurs, then by the same connectivity arguments there must be two different assignments \( q = 1 \) and \( r = 1 \) to variables that both have some neighbour not among \( a, \ldots, g \). As no replacement was created, either we hit a contradiction or some further variable is assigned 0.

In all cases, as long as there exists any long clause other than \((x, a, b, c)\) containing one of \( a, \ldots, g \), our branching is dominated by \( \tau(13, 4) \).

Assume, then, that no extra length bonus is given, and we must account for at least five occurrences of the variables \( a, \ldots, g \). We divide into cases according to the number of replacements.

1. If only replacements occur, then at least four variables are removed due to these, as five inequalities on four variables is not possible.

2. If otherwise at least three replacements among the external variables occur, then either at least four variables are removed due to these, or three variables are removed due to these and there is some assignment to some external variable.

3. If two replacements are made among the external variables, then at least three occurrences of \( a, \ldots, g \) are unaccounted for.

(a) If the replacements are disjoint, say \( p_1 \neq p_2 \) and \( q_1 \neq q_2 \), and assignments are made to two different external variables, then whether the assignments are \( p_1 = q_1 = 1 \), \( p_1 = r = 1 \), or \( r_1 = r_2 = 1 \) (or a combination of assignments equivalent to one of these), we remove four external variables in total.
(b) If the replacements are disjoint and only one external variable, which occurs in the replacements, is assigned, then by connectivity we must remove at least five variables. If only one external variable is assigned which does not occur in the replacements, then let this variable be a fifth external variable $r$. By connectivity, $r$ must have some external neighbour which counts as a fourth removed external variable, and if this variable is, say, $q_1$, then either $q_1$ or $q_2$ has some other external neighbour, by connectivity. Otherwise, this removed variable counts immediately.

(c) If the replacements are not disjoint, say $p \neq q_1$ and $p \neq q_2$, then any assignment to one of these must have an effect on some other external variable by connectivity, and an assignment to another external variable $r$ must in any case lead to the removal of some fourth external variable, by connectivity.

In all cases, $\Delta f \geq 13$, and we achieve a branching dominated by $\tau(13, 4)$.

4. If only one replacement is made, say $p_1 \neq p_2$, then immediate assignments must be made to at least two variables, and one of these must be different from $p_1$ and $p_2$ (call this variable $q$). Whether one has assigned values to $p_1$, $p_2$ and $q$, or to $q$ and $r$ for some $r$ (and in addition replaced $p_1 \neq p_2$), some further variable is assigned by connectivity, and $\Delta f \geq 13$.

5. If no replacement is made, then assignments to a least three different external variables must occur due to connectivity, and some further assignment to some external variable must be made. Barring a contradiction, at least four external variables are removed.

This finishes the case enumeration, and we see that a branching dominated by $\tau(13, 4)$ is guaranteed in all cases.

Our final lemma of this section deals with case 16: all variables are pure and all heavy variables occur in exactly three short clauses, but
some heavy variable $x$ occurs in a 3-cycle (or has two of its neighbours occurring in a long clause). Unfortunately, the proof is lengthy and consists mostly of case enumeration.

**Lemma 47.** Case 16 of XShort (with clauses $(x, a, b)$, $(x, c, d)$, $(a, c, E)$, where we are branching on $b$) results in a branching dominated by $\tau(9, 6) < 1.0983$ or a more balanced version thereof.

**Proof.** Assume without loss of generality that the clauses are $(x, a, b)$, $(x, c, d)$, $(x, e, f)$, $(a, c, E)$, and let the second occurrence of $b$ be in $(b, B)$. The variables $a$, $b$, $c$, $d$ or $x$ do not occur in $E$. The variables $e$ or $f$ can occur in $E$, but if $E = e$ then $x = 0$ (since $x = 1$ leaves an empty clause). In summary we find that $E$ may or may not contain $e$ or $f$, and in addition contains $g_1, \ldots, g_i$ for $i \geq 1$. Also note that the alternatives for $B$ are limited: $b$ is already a neighbour of $x$ and $a$, cannot be a neighbour of $c$ (since $c = 1$ would leave an empty clause, and be caught by case 11), and cannot be a neighbour of $d$ (since $a \neq d$ would trigger a contradiction, with $x = b = c = 0$ leaving $(a)$ and $(\overline{a})$). This leaves the following possibilities for $E$ and $B$.

- $E$ will contain $g_1, \ldots, g_i$ for $i \geq 1$
- $E$ may contain one of $e$ and $f$
- $B$ can contain variables from the set $\{e, f, g_1\}$, and can also contain some further variables $h_j$.

Keeping these possibilities in mind, we divide into cases in the following manner: first if $b$ occurs in a long clause, then cases for $|E| = 1$, and finally cases where $|E| > 1$.

1. If $b$ occurs in a long clause $(b, B)$, then setting $b = 1$ assigns $x = a = B = 0$, replaces $c \neq d$ and $e \neq f$, shortens the clause $(a, c, E)$ (resulting in either a replacement including $g_1$, a shortened long clause, or $c = 1$), and removes the long clause $(b, B)$, for a total reduction in $f(F)$ of at least 10. Setting $b = 0$, we shorten the clause $(b, B)$ and perform the replacement $x \neq a$, leaving the clauses $(x, c, d)$ and $(\overline{x}, c, E)$ so that $c = 0$ by case
8, with \(d \neq x\) and leaving the clause \((\bar{x}, E)\) resulting in either a replacement involving \(g_1\) or a shortened long clause. In total, we can guarantee \(\tau(10, 6)\) for this case.

2. If \(|E| = 1\) (say, \(E = g\)), then in addition to the previous restrictions, \(b\) cannot be a neighbour of \(g\) since case 11 would apply \((a \neq d\) leads to \(x = c = 0\) by case 8, and \(d = 1\) causes \(b = g = 1\)), so \(b\) must be a neighbour of some variable \(h_1\), and at least one of \(h_2, e\) or \(f\). By symmetry, we can ignore \(f\), which leaves two cases to examine.

(a) If the clause \((b, h_1, h_2)\) occurs, then \(b = 1\) causes \(x = a = h_1 = h_2 = 0\) and replacements \(c \neq d, e \neq f, c \neq g\), for a reduction of \(f\) by at least 8 points. Consider then the second occurrence of \(h_1\): since no negated variables remain, either \(h_1\) appears in a long clause or setting \(h_1 = 0\) creates some new inequality replacement. In the former case, we have \(\Delta f \geq 9\) immediately. In the latter case, note that the inequality cannot be between two variables that have both been assigned values (as only \(b\) has been counted as being assigned 1, and \(b\) and \(h_1\) co-occur in another clause), and the three replacements we have counted do not imply any fourth inequality. We find again \(\Delta f \geq 9\) in the \(b = 1\) branch.

In the \(b = 0\) branch, \(a \neq x\) and \(h_1 \neq h_2\). The replacement \(a \neq x\) leads to \(x = 0\) (by case 8), with \(x \neq d\) and \(a \neq g\). In total, we have assigned values to \(b\) and \(c\), and removed \(a, d, g\) and \(h_2\) by replacement. The branching is dominated by \(\tau(9, 6)\).

(b) If the clause \((b, e, h)\) occurs instead, then consider the second occurrence of \(h\), if it has any. We will show that \(\Delta f \geq 9\) in the \(b = 1\) branch.

i. If \(h\) is a singleton, then setting \(b = 1\) implies \(x = a = e = h = 0, f = 1, c \neq d\) and \(c \neq g\). Since \(h\) is a singleton, \(f\) cannot be one and the literal \(f\) must occur in another clause, where all neighbours are assigned.
0. This clause must contain some unassigned variable (since \(h\) does not appear), which brings \(\Delta f\) to at least 9.

ii. If \(h\) appears in a long clause, then the length reduction brings \(\Delta f\) to at least 9.

iii. If \(h\) appears in a clause that is already satisfied by the assignments counted so far, then the only potential such clause is \((a, f, h)\), but with this clause the replacement \(x \neq h\) is detected as impossible by case 11 (the replacement would have forced assignments \(a = b = e = 0\) by case 8, leaving 1-edges \((x)\) and \((\bar{x})\)).

iv. Otherwise, due to the occurrence of \(h\), a new inequality replacement or assignment is added. Since only two inequalities have been counted so far, this inequality must contribute to \(\Delta f\), bringing it to at least 9.

Setting \(b = 0\) causes \(a \neq x\) and \(e \neq h\), and case 8 leads to \(c = 0\), with \(x \neq d\) and \(a \neq g\). In total, we get a branching dominated by \(\tau(9, 6)\).

3. Finally, we have \(|E| > 1\). Either \(E = (e, g_1, \ldots, g_i)\) with \(i \geq 1\), or \(E = (g_1, \ldots, g_i)\) with \(i \geq 2\), and in either case, \(d(a) = d(e) = 2\) and \(d(v) \leq 2\) for every \(v\) in \(E\). A clause \((b, e, g_1)\) or \((b, f, g_1)\) cannot occur, since then case 11 would apply (in the first case, \(a \neq e\) is a contradiction: by case 8, \(x = b = g_1 = 0\) is required, which leaves two 1-edges \((a)\), \((\bar{a})\); the second case is symmetrical). We see that \(b\) must have a new variable \(h\) as neighbour.

(a) If \(E = (e, g_1, \ldots, g_i)\), then the clause \((b, B)\) where \(b\) occurs includes either two new variables, or one new variable plus either \(g_1\) or \(f\). Setting \(b = 1\) forces \(x = a = B = 0\), \(c \neq d\), \(e \neq f\), and a shortening of the clause \((a, c, E)\) for eight guaranteed points of reduction (as the variables of \(B\) cannot be \(x\), \(a\) or any pair of variables that are equal by replacements). Setting \(b = 0\), we get a replacement from
5. One-in-three Satisfiability (B), and \( x \neq a \), which leads to both \( c = 0 \) and \( e = 0 \) by case 8, implying \( x \neq d \), \( x \neq f \), and a shortening of the clause \((a, c, E)\). We have assigned values to \( b \), \( c \) and \( e \), removed \( a \), \( d \), \( f \) and a variable \( h_1 \) of \( B \) by replacement, and shortened a long clause, for a branching dominated by \( \tau(8, 8) \).

(b) If \( E = (g_1, g_2, \ldots, g_i) \) with \( i \geq 2 \), then \( b \) can occur in \((b, e, h_1)\) or \((b, f, h_1)\) (which are symmetrical cases in this subcase), in \((b, g_1, h_1)\), or in \((b, h_1, h_2)\). We split according to these possibilities.

i. If \((b, e, h_1)\) occurs, then setting \( b = 1 \) causes \( x = a = e = h_1 = 0 \), \( c \neq d \), \( f = 1 \), and the clause \((a, c, E)\) is shortened to \((c, E)\). If \( f \) is a singleton, then \( h_1 \) cannot be a singleton and setting \( h_1 = 0 \) causes either an extra point of reduction due to a shortened clause, or creates an inequality. This inequality cannot contain two variables that have already been counted as being assigned different values, and the replacements that have been counted do not form an implicit inequality, bringing the total reduction in \( f \) to 9. If \( f \) is not a singleton, then every neighbour of \( f \) is assigned 0. Since \( d(a) = 2 \), this clause must contain some unassigned variable. Setting \( b = 0 \) gives us \( x \neq a \), \( e \neq h_1 \), \( c = 0 \) by case 8, \( x \neq d \), and the clause \((a, c, E)\) is shortened to \((\bar{x}, E)\), for a branching number dominated by \( \tau(9, 6) \).

ii. If \((b, g_1, h_1)\) occurs, then setting \( b = 1 \) causes \( x = a = g_1 = h_1 = 0 \), \( c \neq d \), \( e \neq f \), and the clause \((a, c, g_1, \ldots, g_i)\) is shortened to \((c, g_2, \ldots, g_i)\). If \( i \geq 3 \), then this accounts for a reduction of at least 9 due to two points of length bonus, and if \( i = 2 \), then \( c \neq g_2 \) is an additional replacement. Setting \( b = 0 \), we get \( x \neq a \), \( g_1 \neq h_1 \), \( c = 0 \) by case 8, \( x \neq d \), and the clause \((a, c, g_1, \ldots, g_i)\) is shortened, for a total branching dominated by \( \tau(9, 6) \).
iii. If \((b, h_1, h_2)\) occurs, then one of \(h_1\) and \(h_2\) will not be a singleton (say \(h_1\)). When \(b = 1\), in addition to \(x = a = h_1 = h_2 = 0\), \(c \neq d\), \(e \neq f\), and the clause \((a, c, E)\) which is shortened to \((c, E)\), \(h_1 = 0\) implies a replacement, and as before no implicit inequalities are possible among the replacements accounted for, and \(h_1\) cannot be a neighbour variable assigned 1. The total reduction in \(f(F)\) is at least 9. Setting \(b = 0\) we again get \(x \neq a\), \(h_1 \neq h_2\), \(c = 0\) by case 8, and \(x \neq d\), and the clause \((a, c, E)\) is shortened to \((\bar{x}, E)\), and the branching is dominated by \(\tau(9, 6)\).

\[\square\]

5.3 Sparsification Cases

When the algorithm reaches past case 16 of XShort, we have a situation where we can predict the reduction in \(f(F)\) due to setting \(x = 1\) for some heavy variable. After this point, the overarching goal of the algorithm is to reach a situation where the heavy variables contribute a sufficiently small part to \(f(F)\) that explicit enumeration on them fits within our time bound; more details on this later. First, we give a lemma showing that we can indeed predict the effects of setting \(x = 1\).

Lemma 48. Let \(x\) be a heavy variable, and let \(S(x)\) be the sum of the degrees of all neighbours of \(x\). In every case after case 16 of XShort, if \(S(x) \leq 12\) then setting \(x = 1\) reduces \(f(F)\) by \(S(x) + 1\).

Proof. Let the clauses containing \(x\) be \((x, a, b)\), \((x, c, d)\), \((x, e, f)\). Since no pair of variables among \(a, \ldots, f\) co-occur in any other clause, the effects of setting \(x = 1\) (in addition to removing seven variables) are shortenings of long clauses, and inequality replacements. We will show limits on the kinds of cycles of replacements that can occur; the proof is closely tied to the applicability of cycle replacements from Definition 33.
5. One-in-three Satisfiability

First, no odd cycle of replacements can occur, since every variable in \( F \) is pure (and such a cycle would be non-consistent, and caught by case 11). To reach a reduction of up to 13 points, we need to consider 4-cycles and 6-cycles.

A 4-cycle of replacements with one variable occurring twice among the labels would be caught by case 11. Neighbouring clauses of the cycle cannot have the same label, so assume that labels 1 and 3 of the 4-cycle are \( a \). Also, the variable \( b \) cannot occur as a label of this cycle, as \( d(a) = 3 \) and \( a \) will not appear in a 3-cycle, which allows us to assume that label 2 is \( c \). The clauses of the cycle are then \((a,t,u),(c,u,v),(a,v,w),(l_4,t,w)\) for some \( l_4 \), which leads to a contradiction if \( a \neq c \): \( u = v = 0 \) is required by case 8, but this propagates to the contradiction \( t = w = 1 \).

All 4-cycles of replacements with four variables among the labels are covered by some cycle replacement. All statements that “label \( i \) is \( v \)” are to be taken without loss of generality (i.e. the first variable mentioned, which is \( a \), represents any variable among \( a, \ldots, f \); once \( a \) has been used, then \( c \) represents variables \( c, \ldots, f \); and so on). Without loss of generality, label 1 is \( a \), and label 2 is \( b \) or \( c \).

- If label 2 is \( b \), then label 3 is \( c \) and label 4 is \( d \) or \( e \), both leading to cases occurring in the table. This covers all cases where two neighbouring labels occur in the same clause with \( x \).

- If label 2 is \( c \) and label 3 is \( b \), then label 4 is \( d \) or \( e \), and both cases occur in the table. We have now covered all cases where any two labels occur in the same clause with \( x \).

- Otherwise, we need to pick four labels from three clauses without choosing two from the same clause, which is impossible.

We see that no cycle of four replacements can occur.

If a 6-cycle of replacements using three variables as labels occurs, then there is a fourth non-singleton variable among \( a, \ldots, f \) whose occurrence has not been accounted for. If this occurrence shortens a long clause or leads to a replacement involving some variable not in the 6-cycle, then we reduce \( f(F) \) by in total 13 points. Otherwise,
5.3. Sparsification Cases

this occurrence leads to a replacement involving two variables in the 6-cycle, but this replacement cannot be identical to an existing replacement, which means it either creates odd cycles or creates two 4-cycles, neither of which can exist.

Clearly, no 6-cycle of replacements using four or six variables as labels can occur, as cycle replacement would apply, but also, any 6-cycle using five variables as labels will trigger some reduction. Assume first that labels 1 and 4 are both \( a \) (covering all cases where two labels on distance three in the cycle are identical). Since \( d(a) = 3 \), \( b \) cannot occur as a neighbouring label of \( a \) in the cycle, which requires that the other labels use all of \( c, d, e \) and \( f \).

- If a sequence like \( a, c, d, a \) occurs, i.e. by clauses \((a, i, j), (c, j, k), (d, k, l), (a, l, m)\), then case 11 is triggered: \( a = 1 \) forces both \( c \neq d \) and \( j = l = 0 \), so that \( c \neq k \) and \( d \neq k \), leading to a detected contradiction.

- Otherwise, the start of the sequence of labels is without loss of generality \( a, c, e, a \), and the last two are either \( d, f \) or \( f, d \), where both are cases that appear in Table 5.1.

Next, assume that labels 1 and 2 are \( a \) and \( b \) (covering all new cases where two neighbouring labels have variables occurring in the same clause with \( x \)). In this case, there is a 3-cycle with \( a \) and \( b \), meaning that neither variable is heavy, so we can assume that label 3 is \( c \), ignoring the cases where label 6 is \( c \).

- If label 4 is \( d \), then labels 5 and 6 can use only \( e \) and \( f \), which triggers some reduction in every case.

- Otherwise, label 4 is \( e \), and label 5 is \( c, d \), or \( f \). If label 5 is \( c \), then label 6 is \( f \), and the sequence is \( a, b, c, e, c, f \) which occurs in Table 5.1.

- If label 4 is \( e \) and label 5 is \( d \), then the sequence is assumed to be \( a, b, c, e, d, e \) which occurs in the table.
5. One-in-three Satisfiability

- If label 4 is e and label 5 is f, then no option remains: both a, b, e and f must be degree 2; label 6 is not c by assumption; and label 6 is not d since this uses six different variables.

Finally, assume that labels 1 and 3 are both a, in which case (ignoring the already visited cases) label 2 is c, and label 4 is d or e.

- If label 4 is d, then label 5 is b (since otherwise there is no option for label 6) and the only possible sequence is a, c, a, b, e which appears in the table.

- If label 4 is e, then label 5 is b or d. If b, then the sequence is a, c, a, e, b followed by d or f; the latter option is in the table, while the former option is in the table if read in reverse.

- If label 4 is e and label 5 is d, then label 6 is f and the sequence appears in the table.

It is perhaps interesting to note that among the cases we consider, there is only exactly one kind of label configuration for a 4-cycle of replacements, and one kind of label configuration for a 6-cycle of replacements using at least four variables, that does not imply an assignment or replacement. In either case, we apply the result of Lemma 48 to case 17 of the algorithm.

Lemma 49. Case 17 of XShort (when the sum of degrees of neighbours of x is at least 12) results in a branching dominated by \( \tau(13, 4) < 1.0955 \).

Proof. The result follows immediately from Lemma 48.

There are now four different types of heavy variable remaining, if one considers the neighbourhood. They are given and named in Figure 5.1, and will be used in the future discussion. Variables a, \ldots, e are light non-singletons, each s is a different singleton, and x and y are heavy (obviously x and y are of the same type when both occur). We also give names to the types of occurrences of light variables in
clauses containing a heavy variable. An occurrence of type e.g. 1a is an occurrence equivalent to that of the variable $a$ in the description of a heavy variable of type 1; we get the six types of occurrences 1a, 1c, 2a, 2e, 3a, 3b, 4a, and 4c.

The rest of the algorithm, as mentioned, mainly has the goal of controlling the occurrences of the neighbours of a heavy variable $x$; first to remove cases where the neighbour of a neighbour of a heavy variable $x$ is heavy, then to put limits on how neighbours of heavy variables co-occur. For a “preview” of the significance of the various cases, see Tables 5.2 and 5.3 on pages 136 and 137.

Our next lemma gives the branching for case 18, which deals with certain variables with singleton neighbours.

**Lemma 50.** Case 18 of XShort (with clauses $(x, a, s), (a, b, C)$ where all other occurrences of $b$ are in singleton-clauses) results in a branching dominated by $\tau(11, 5)$.

**Proof.** In the $x = 1$ branch, $\Delta f \geq 11$ by Lemma 48. In the $x = 0$ branch, first $f$ reduces by 4 due to the clauses where $x$ appears, then the variable $a$ becomes a singleton so that every occurrence of $b$ is in a clause with a singleton; $b = 0$ is assigned, for a branching dominated by $\tau(11, 5)$. \qed

Case 19, which we treat next, can be summarised as handling most cases where a heavy variable not of type 1 has a light neighbour that occurs with any other heavy variable.

**Lemma 51.** Case 19 of XShort (with sum of degrees of neighbours 11, light neighbours $a$ and $b$, and a heavy neighbour $y$ of $a$) results

<table>
<thead>
<tr>
<th>$(x, a, b)$</th>
<th>$(x, a, b)$</th>
<th>$(x, y, a)$</th>
<th>$(x, y, s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x, c, s)$</td>
<td>$(x, c, d)$</td>
<td>$(x, b, s)$</td>
<td>$(x, a, b)$</td>
</tr>
<tr>
<td>$(x, d, s)$</td>
<td>$(x, e, s)$</td>
<td>$(x, c, s)$</td>
<td>$(x, c, s)$</td>
</tr>
</tbody>
</table>

Type 1 | Type 2 | Type 3 | Type 4

**Figure 5.1:** The types of heavy variables after case 17
5. One-in-three Satisfiability

in either a branching dominated by \( \tau(12, 5) < 1.0908 \), or a branching dominated by \( \tau(12, 17, 8) < 1.0984 \).

Proof. Without loss of generality, let the clauses containing \( x \) be \((x, a, b), (x, e, s)\), with \( s \) a singleton and the other variables 2-variables, plus either \((x, c, d)\) or \((x, z, s')\) where \( d(c) = d(d) = 2 \) or \( d(z) = 3 \) and \( s' \) is a singleton (the heavy variable \( z \) cannot be identical to \( y \), since \( a \) and \( y \) are neighbours). We have \( \Delta f \geq 12 \) in the \( x = 1 \) branch, by Lemma 48. In the \( x = 0 \) branch, three replacements are performed, and if any further reduction which reduces \( f(F) \) occurs, then the result is a branching dominated by \( \tau(12, 5) \). Otherwise, we will show that a branching dominated by \( \tau(13, 4) \) is guaranteed immediately in this branch, making the total 3-way branching \( \tau(12, 17, 8) \).

The reductions that can occur that do not decrease \( f(F) \) are case 3 for the special case of a 2-clause \((p, \bar{p})\), case 4 when a duplicate clause occurs, case 9, and case 12 for certain particular cycle replacements. Cases 3 and 4 will not occur in this manner for this branch, since only three disjoint replacements have been made of which any other existing clause contains at most one variable, and case 9 applies only to the one or two variables that are the results of the replacements \( a \neq b \) and possibly \( c \neq d \); call the resulting \((1, 1)\)-variables \( a \) and, if it exists, \( c \). The application of case 9 does not change the number of occurrences of the remaining variables, so unless \( \Delta f \geq 5 \), we know that case 9 will apply exactly to these variables. Let the clause in \( F \) that contains \( y \) and \( a \) be \((y, a, p)\) (where \( p \) can have any degree up to 3, but is not a neighbour of \( x \)), and the second clause containing \( b \) be \((b, B)\); we know that \(|B| \geq 2\), and that \( B \) does not contain \( x \), \( y \), or any neighbour of \( x \) or \( y \). After the replacements and the applications of case 9, the clause \((y, p, B)\) occurs, \( y \) is still heavy, and there are no negative occurrences of any variable. Cases 3 and 4 are still not relevant, and case 9 will no longer apply. We will show that cycle replacement in case 12 also does not apply, so that either \( \Delta f \geq 5 \) or case 15 is applied.

We know that cycle replacement does not apply to \( F \). Since the applicability of cycle replacement only depends on the configuration of clauses, the lengths of the clauses, and the existence of a central
heavy variable to which the labels are neighbours, no replaceable cycle can have been created: the only new clauses are either long \(((y, p, B))\), and the clause created by applying case 9 to \(c\), if performed), or existed in the same configuration before with the only difference that their members have changed their degrees (any occurrence of \(e\) and \(z\)). Thus, case 12 does not apply.

Cases 13 and 14 also do not apply, since no variable of the corresponding kind exists. Thus, case 15 applies, which by Lemma 46 results in a branching dominated by \(\tau(13, 4)\). Extending the second branch of our \(\tau(12, 4)\)-branching by this, we get the promised branching \(\tau(12, 17, 8) < 1.0984\).

The next lemma, for case 20 of the algorithm, does not appear in any of the tables, but is referred to in the proof of Lemma 53.

**Lemma 52.** Case 20 of XShort (with clauses \((x, a, p)\), \((a, q, s)\) where \(x\) has two singletons as neighbours, and we branch on \(q\)) results in a branching dominated by \(\tau(8, 7) < 1.0970\).

**Proof.** If \(q\) is heavy, then case 18 applies.

Otherwise, \(d(q) = 2\) and \(q\) appears in a clause \((q, Q)\) where \(Q\) does not contain \(x\) or a singleton.

- If \(|Q| > 2\), then in the \(q = 1\) branch we remove at least 6 variables by assignment, shorten a long clause, and perform a replacement \(x \neq s\), getting \(\Delta f \geq 8\).
- If \(Q\) contains \(p\), then setting \(q = 1\), we get \(a = p = 0\) and \(x = 1\), causing assignments to more than eight variables.
- Otherwise, setting \(q = 1\) causes \(Q = s = a = 0\), \(x \neq p\), and two further clauses are shortened. We cannot create a cycle of only three replacements without causing a contradiction, and the variable \(q\) does not appear in any further clause, so we must get \(\Delta f \geq 8\).

In the branch \(q = 0\), \(f\) is immediately reduced by 3, and \(a\) becomes a singleton, leading to \(x = 0\) and three further replacements against
singleton (one of which is a); since only two of the replacements do not involve a variable that is a singleton in F, we get $\Delta f \geq 7$. Thus, the branching when $q$ is light is dominated by $\tau(8, 7)$.

Now we reach case 21, the final case represented in the tables and the last part of this section. After this case, only one more case, dealing with dense and non-dense clauses, is needed before the enumeration process starts.

**Lemma 53.** Case 21 of XShort (with clauses $(x, a, p), (a, q, A)$ where every other occurrence of $x$ or $q$ is with a singleton, and we branch on $p$) results in a branching dominated by $\tau(9, 6) < 1.0983$ or $\tau(12, 5) < 1.0908$.

**Proof.** If $p$ is heavy, then the other two occurrences of $p$ must be in clauses with a light variable and a singleton. By setting $p = 1$, we get $\Delta f \geq 12$, and when $p = 0$, we get the replacements $x \neq a$ as well as two replacements involving singletons, which is equivalent in effect to dropping these clauses, turning the two light variables into singletons. The variable $q$ cannot be one of these light variables, or case 16 would apply (there would be a 3-cycle with $p$). We get a fifth point of reduction: either $x$ or $q$ occurs in a clause with two singletons, or $q$ has a singleton as neighbour in each clause where it appears, or we have a situation where the literal $x$ appears in two clauses with singletons, the literal $\bar{x}$ appears in a clause with $q$, and the variable $q$ otherwise only appears in clauses with singletons, triggering case 7, setting $q = 0$. We get $\Delta f \geq 5$.

If $p$ is light, then let the second occurrence of $p$ be in a clause $(p, P)$, where $P$ contains no singleton by case 20 and cannot contain $x$ or any neighbour of $x$ (but may share a variable with $A$). Setting $p = 1$, we get $x = a = P = 0$ and we have two further occurrences of $x$, one occurrence of $a$, and at least two occurrences of variables in $P$. The occurrences of $x$ are in clauses containing a light variable and a singleton. None of the other three occurrences can be in these clauses, and no consistent cycle of replacements is possible. Furthermore, at least one variable of $P$ does not occur in $A$. We get $\Delta f \geq 9$ in the $p = 1$ branch. In the $p = 0$ branch, we get one shortening or
replacement due to the clause \((P)\), and we get \(x \neq a\), creating the clause \((\bar{x}, q, A)\) (or possibly \((\bar{x}, q, A')\) where \(A'\) differs from \(A\) by the replacement due to the clause \((P)\)), which leads to \(q = 0\) by case 7. Now, in total \(p = q = 0\) has been assigned, and four clauses have been shortened, one of which includes a singleton so that a consistent cycle of replacements is not possible. We have \(\Delta f \geq 6\) and get a branching dominated by \(\tau(9,6)\).

Table 5.2 illustrates which cases that apply when a light variable \(a\) has both occurrences in clauses that contain some heavy neighbour. The labels (1a, 1c, etc) of the rows and columns are references to Figure 5.1, as previously described. For instance, if there is a light variable with one occurrence of type 1c and one of type 2e, then case 6 applies, as the variable has a singleton as neighbour in both occurrences.

**Lemma 54.** After case 21 of XShort has been passed, no light variable \(v\) occurs in two clauses with some heavy variable, as explained in Table 5.2.

**Proof.** Case 6 applies when the variable \(v\) has two singleton neighbours, which is the case in the entries of the table containing “6”.

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>1c</th>
<th>2a</th>
<th>2e</th>
<th>3a</th>
<th>3b</th>
<th>4a</th>
<th>4c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>18</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>1c</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>2a</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>2e</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>3a</td>
<td>21</td>
<td>18</td>
<td>18</td>
<td>21</td>
<td>18</td>
<td>19</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>3b</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>4a</td>
<td>19</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>18</td>
<td>19</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>4c</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>6</td>
<td>18</td>
<td>6</td>
</tr>
</tbody>
</table>
Table 5.3: Cases that apply when two light variables, each with an occurrence with some heavy variable, co-occur in a clause without a heavy variable

<table>
<thead>
<tr>
<th></th>
<th>1a</th>
<th>1c</th>
<th>2a</th>
<th>2e</th>
<th>3a</th>
<th>3b</th>
<th>4a</th>
<th>4c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
</tr>
<tr>
<td>1c</td>
<td>21</td>
<td>18</td>
<td>18</td>
<td>21</td>
<td>18</td>
<td>21</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>2a</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2e</td>
<td>21</td>
<td>18</td>
<td>18</td>
<td>21</td>
<td>21</td>
<td>18</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>3a</td>
<td>-</td>
<td>21</td>
<td>21</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
<td>-</td>
</tr>
<tr>
<td>3b</td>
<td>21</td>
<td>18</td>
<td>-</td>
<td>21</td>
<td>18</td>
<td>21</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>4a</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
<td>-</td>
<td>21</td>
</tr>
<tr>
<td>4c</td>
<td>21</td>
<td>18</td>
<td>-</td>
<td>21</td>
<td>18</td>
<td>21</td>
<td>18</td>
<td>18</td>
</tr>
</tbody>
</table>

Case 18 applies if one occurrence is of type 1c, 2e, 3b, or 4c (the clause \((x, a, s)\) in the case description) and the other of type 1a, 3a, or 4a (the co-occurrence of \(a\) and \(b\) in the case description), which agrees with the table.

Case 19 applies if one occurrence is of type 2a or 4a, with no restrictions on the other occurrence (except that it is heavy). This agrees with the table (except of course for the cases already handled by case 18).

Case 20 covers no extra case in this context, and case 21 covers the remaining cases: both occurrences are of type 1a or 3a.

The corresponding table for how light neighbours of heavy variables can occur in the same clause is Table 5.3. It is read the same way—for instance, if two variables \(u, v\) co-occur, when \(u\) has another occurrence of type 1a and \(v\) another occurrence of type 3b, then case 21 applies, while if \(u\) instead has an occurrence of type 2a, then no case necessarily applies. Also note that an occurrence of a light neighbour of a heavy variable that is not in a dense clause leads to a decreased density of heavy variables in the measure \(f(F)\), as this occurrence must be in a clause which is either long (increasing the length bonus part of \(f(F)\)) or has a light member without heavy neighbours (decreasing the \(n_3(F)/n(F)\) density of \(F\)). Note the pat-
tern: a co-occurrence of variables \( u \) and \( v \), when these are neighbours of different heavy variables, is possible if either both variables have their occurrence in a clause without a singleton (type 1a, 2a, 3a, and 4a), or one of the variables has an occurrence of type 2a. Note also that we do not yet guarantee that any occurrence of a light neighbour of a heavy variable is in anything but a dense clause; the significance of this table is more indirect.

**Lemma 55.** Table 5.3 correctly shows the possible co-occurrences of light variables which otherwise occur with some heavy neighbour.

**Proof.** Call the variables that co-occur \( u \) and \( v \). When both \( u \) and \( v \) have their other occurrence in clauses \((x, u, s), (y, v, s')\) (i.e. types 1c, 2e, 3b, and 4c), case 18 applies (with \( u, v \) as \( a, b \) in the case description), which agrees with the table. Case 21 applies when \( u \) occurs in type 1a, 3a, or 4a (as \( a \) in the case description) and \( v \) occurs in a clause \((y, v, s)\), where \( v \) is \( q \) in the case description. This also agrees with the table.

### 5.4 Final Cases

When we reach this point, as no light neighbour of a heavy variable has another heavy variable as neighbour in another clause, there must exist six light variables for each variable of type 1 or 2, and nine light variables in total for each pair of variables of type 3 or 4. This is close to what we need, but not quite there yet; by these numbers alone, we would reach \( 2^{n/7} \approx 1.1041^n \) light instances if all heavy variables are of type 1 or 2, and \( 3^{n/11} \approx 1.1051^n \) light instances if all heavy variables are of type 3 or 4 (and something in between when there is a mixture). It is at this point that the significance of the dense clauses enters the picture: case 22 of the algorithm applies when a variable of type 2, 3, or 4 has light neighbours that practically only occur in dense clauses, and as we have seen from Table 5.3 a variable of type 1 whose neighbours only occur in dense clauses has a lot of secondary neighbours (neighbours of neighbours) that have occurrences of type 2a. In this way, we can prove that the weight of the heavy variables...
comprises a sufficiently small part of $f(F)$ that the number of light instances created by enumeration is $O^*(1.0984^{f(F)})$.

But first, the proof of the branching time for case 22. Unfortunately, it is one of the longest case enumerations of the chapter.

**Lemma 56.** Case 22 of XShort results in either a branching dominated by $\tau(13, 4) < 1.0955$, or a branching dominated by $\tau(26, 16, 4) < 1.0981$.

*Proof.* Branching on $x$ reduces $f(F)$ by 12 in the $x = 1$ branch and 4 in the $x = 0$ branch. In the $x = 1$ branch, every occurrence of a neighbour of $x$ in a dense clause results after replacement in a $(1,1)$-variable having a heavy variable as neighbour in each occurrence. Unless some reduction applies which brings the branching to $\tau(13, 4)$, case 9 is applied to each such variable resulting in long clauses that contain more than one heavy variable.

If some variable $y$ in the branch $x = 1$ occurs in two long clauses, or in some clause of length at least five, then the base branching (Lemma 44) alone proves that we get a branching dominated by $\tau(14, 4)$ in this branch, and a total branching dominated by $\tau(26, 16, 4)$. Otherwise, we have to look a little closer at the possible cases to prove that a branching dominated by $\tau(14, 4)$ is guaranteed.

First, let $x$ be of type 2 or 3, and let less than two occurrences of light neighbours of $x$ be in non-dense clauses. We shall make a few observations.

1. The replacements that have been performed amount to only inequalities (i.e. there is no implied replacement $u = v$). When $x$ is of type 2, this is immediate; if $x$ is of type 3, then note that the neighbour $y$ of $x$ otherwise only occurs with singletons, which amount to dropped clauses when $y = 0$ is set.

2. The only new or changed clauses are long. This is because no variable in a 2-clause was heavy, so that every 2-clause lead to either a light variable turning singleton, or a $(1,1)$-variable to which case 9 is applied.
3. No non-$f(F)$-reducing reduction (case 3, 4, 9, or 12) can apply, once case 9 has done away with the initial $(1, 1)$-variables, because of the previous observations.

Thus, some heavy variable occurs in a long clause with another heavy variable, and case 15 is used; call the variable that is branched on in case 15 $y$, and let the long clause where $y$ appears be $(y, z, p, q)$ where $y$ and $z$ are heavy, $1 \leq d(p), d(q) \leq 3$, and $y$ and $p$, and $z$ and $q$, occurred together in $F$. When $y = 1$, an immediate reduction of $f$ by 9 occurs due to 8 assigned variables and one removed long clause, and there are at least six further occurrences of neighbours of $y$. The important question is how many clauses contain more than one neighbour of $y$.

1. If all six further occurrences result in shortened long clauses or newly created 2-clauses, then we certainly get $\Delta f \geq 14$.

2. If there is one clause containing exactly one variable that is not a neighbour of $y$, then call this variable $t$. We have $t = 1$, and four further occurrences of neighbours of $y$. If these occurrences form anything other than a 4-cycle of replacements, then we get $\Delta f \geq 14$. For $\Delta f < 14$ to hold, there would need to be no length bonuses, four replacements in a 4-cycle, and two occurrences in a short clause with $t$ as the third member, but $z$ cannot be involved in the latter clause (or a neighbour of $z$ would be a neighbour of $y$ as well in $F$), and at most once in the 4-cycle (or neighbours of $z$ would be neighbours in $F$).

3. Finally, any clause containing only one member that is not a neighbour of $y$ must contain $q$ (even if it is a long clause formed by case 9, at least two of the non-external members must have been neighbours in $F$), which means that if more than one such clause occurs, then the external variables must all be different, and each such clause either accounts for only one non-$q$ occurrence or accounts for two non-$q$ occurrences but reduces $f$ by two.
We get $\Delta f \geq 14$ in every case, and $\Delta f \geq 4$ in the other branch, forming the branching of $\tau(26, 16, 4)$.

If $x$ is of type 2 and two or three occurrences of neighbours of $x$ are in non-dense clauses, or if $x$ is of type 4 and less than two occurrences of neighbours of $x$ are in non-dense clauses, then let us again make some observations.

1. One replacement $u = v$ can occur, but only one. In the first case, this is immediate. In the second case, the neighbour $y$ of $x$ occurs in one clause with two light non-singleton neighbours, and this along with the non-dense occurrence of a light neighbour of $x$ could form an equality.

2. In both cases, at least two occurrences of neighbours of $x$ are still in dense clauses, leading to applications of case 9 forming long clauses with multiple heavy variables.

Case 12 could apply, if a short clause containing $v$ occurs in a cycle when $v$ is replaced by $u$, but if case 12 applies in a way that does not reduce $f$, then a whole block of the formula is connected through only the heavy variable of the cycle (referring to the variable to which the labels of the cycle are neighbours; call this variable $w$):

1. No 6-cycle-replacement can occur, since no heavy variable in short clauses has neighbours with six further occurrences (the $u = v$ replacement does not change this, since both $u$ and $v$ have two occurrences in $F$). Therefore, we need only look at a 4-cycle-replacement with label sequence $a, c, b, d$ (as per Definition 33).

2. If a 4-cycle with label sequence $a, c, b, d$ occurs, then three of the clauses exist in $F$, and therefore all four variables in the cycle are light. Also, no label can be heavy, since each heavy variable with a heavy neighbour occurs in only one clause without a singleton, and the pattern requires clauses $(w, a, b)$ and $(w, c, d)$ where $a, \ldots, d$ are not singletons (and, once again, no variable has a higher degree after the replacements than in $F$). Thus,
the clauses \((w, a, b)\) and \((w, c, d)\), along with the four clauses of the cycle, represent every occurrence of eight variables, and connect only through the variable \(w\).

We see that unless the branching \(\tau(13, 4)\) when branching on \(x\) applies, case 12 has not been used, and case 15 will be used. Now, let us go through the possibilities for this case.

Call the variable that is branched on in case 15 \(y\), and let the clause be \((y, z, p, q)\), for heavy variables \(y, z\), where \(y\) and \(p\), and \(z\) and \(q\), are neighbours in \(F\). As before, setting \(y = 1\) removes 8 variables, shortens the long clause containing \(y\), and shortens at least six further clauses. Unless an \(f(F)\)-reducing reduction occurs after \(x = 1\), case 9 is applied twice, leaving four different heavy variables that occur in long clauses. Next, we will show that no two clauses with a single external variables can have the same external part, for the chosen variable \(y\).

- Before the \(u = v\) replacement, no two clauses with a single external variable can have the same external part at all, since all such clauses must go through one variable \((q\) in the case of \((y, z, p, q)\) as given above).

- Refer to a clause that contains exactly one external member but does not contain an occurrence of \(q\) as problematic. The effect of the replacement \(u = v\) can be viewed as a single occurrence of \(v\) being replaced by an occurrence of \(u\); if this replacement occurs in a clause that ends up as a long clause, then this long clause cannot be problematic for both \(y\) and \(z\) (as it would need to have length four and have three members each that are neighbours of \(y\) and \(z\)), otherwise the replacement occurs in a short clause that can be problematic for \(y\) and \(z\), but not at the same time for the second pair of heavy variables \(w\) and \(w'\) (at least, not unless some heavy variable occurs in two long clauses or in an extra-long clause, causing a secondary branching of at least \(\tau(14, 4)\) by the base branching).

- Therefore, we can assume that either \(y\) occurs in two long clauses or one clause which is at least a 5-clause, or by the
secondary criterion of case 15, \( y \) was chosen so that no two clauses in the interface of the neighbourhood of \( y \) have identical external parts.

We can conclude that if there is a problematic clause for the chosen branching variable \( y \), then either \( y \) gets a branching of \( \tau(14, 4) \) by the base branching or all external parts of such clauses are different. As noted, at least six clause shortenings occur due to the neighbours of \( y \) in the \( y = 1 \) branch; we trace their effects.

1. If there are two clauses with a single member as their external parts, then as mentioned these external parts must be different; say that they are \( r \) and \( t \). Because of connectivity, the number of external neighbours to \( r \) and \( t \) plus the number of further, separate shortened clauses equals at least three, and we reduce \( f(F) \) by at least five further points.

2. If there is exactly one clause with a single external part \( t \), then consider the replacements that occur.

   (a) If \( t \) is a singleton, then \( y \) has at most one neighbour that is a singleton, and there are at least five replacements or shortenings of long clauses, which must involve at least five variables. The same holds in any situation where there are at least five replacements.

   (b) If there are exactly four replacements that form a cycle that contains \( t \), then four variables are assigned and some external neighbour of these variables exists.

   (c) If there are exactly four replacements that form a cycle that does not contain \( t \), then as noted, \( t \) is not a singleton and must have some external neighbour, which is assigned 0.

   (d) If there are exactly four replacements that do not form a cycle, then we get five immediate points of reduction.

3. If at least six replacements or shortenings of long clauses occur, then this must bring at least five points of reduction:
(a) If any of these is a shortening of a long clause, then the result is immediate.

(b) Otherwise, we create six inequalities. Either at least one of these includes a singleton (by \( z \)), or all four neighbours of \( z \) are light, and in both cases one cannot create six consistent inequalities on five variables.

We get a branching on \( y \) dominated by \( \tau(14, 4) \) and a total branching dominated by \( \tau(26, 16, 4) \).

Now, we can finally prove the main result. For this, we need to prove that the enumeration that occurs in the final case of the algorithm will produce \( O^* \left( 1.0984^{f(F)} \right) \) light instances.

**Theorem 57.** The algorithm XShort decides the satisfiability of an XSat instance \( F \) in time \( O^* \left( 1.0984^{n+\lambda} \right) \) where \( n \) is the number of variables in \( F \) and \( \lambda \) is the length bonus. For an X3Sat instance, XShort has a running time in \( O^* \left( 1.0984^n \right) \).

**Proof.** Every case up to case 22 has a branching number of at most \( \tau(17, 12, 8) < 1.0984 \), as proved in the previous lemmas. As for case 23, once this case is reached, every heavy variable has at most one heavy neighbour, so if there are \( n_a \) heavy variables that do not have any heavy neighbour, and \( n_b \) heavy variables that do, then the time taken for the enumeration is \( O^* \left( 2^{n_a} \cdot 3^{n_b/2} \right) \), and we need to calculate the maximum density of heavy variables once the case is reached.

We will prove that the density is low enough using an argument based on marking variables: for each heavy variable we mark a number of light variables (and a part of the length bonus, if applicable); if we can avoid ever marking a light variable or a point in the length bonus more than once, and if we can mark light variables and length bonus points to a worth of at least \( k_1 \) points for each variable in \( n_a \) and \( k_2 \) points for each variable in \( n_b \), then we have shown that \( f(F) \geq (k_1 + 1)n_a + (k_2 + 1)n_b \). We will talk of fractional markings as well: if a light variable is marked by \( 1/2 \) from at most two sources, then each time we mark it, we can count \( 1/2 \) point to \( k_1 \) or \( k_2 \). An alternative way to see it is through association: when we mark a light
variable while considering a heavy variable $x$, then this is equivalent to associating the light variable to $x$.

By Lemma 54, all light neighbours of any heavy variable can be marked, leading to six marked light variables for each heavy variable of type 1 or 2, and nine marked light variables for each pair of heavy variables of type 3 or 4 (i.e. 4.5 marked light variables for each such variable). In addition, by case 22 a variable of type 2 has at least four occurrences of neighbours in clauses other than dense clauses. For each such occurrence, if the clause contains a light variable which has no heavy neighbour, then $1/4$ of this light variable can be marked (two occurrences of the variable and two neighbours in each occurrence means that there are at most four sources that wish to mark the variable), and if the clause is long, then the length bonus divided to the members is at least $1/4$. We find that this type of bonus provides at least +1 for each variable of type 2.

For each variable of type 3 or 4, at least two occurrences of light neighbours are in non-dense clauses. Let $x$ and $y$ be heavy neighbours. If they are of type 4, or if they have at least four occurrences of light neighbours in non-dense clauses in total, then they contribute at least 12 towards $f(F)$, and $3^{1/12} < 1.0959$. If they are of type 3 and have three light neighbours occurring in non-dense clauses in total (if the common variable $a$ is one of these occurrences), then they contribute at least 11.75 towards $f(F)$, and $3^{1/11.75} < 1.0981$. We see that the $3^{n_b/2}$ part does not bring the remaining running time above 1.0981.

For each variable of type 1, consider the occurrences of the variables of type 1c (represented by $c$ and $d$). Note that the only neighbours with heavy neighbours that these variables can have are of type 2a. For each variable of type 1 whose associated type 1c-variables have in total at least two neighbours not of type 2a, we mark at least 6.5 points (so that the total contribution towards $f(F)$ associated to such a variable is at least 7.5). If a variable of type 1 has exactly one neighbour of its associated type 1c-variables not of type 2a, then it immediately contributes 7.25 towards $f(F)$ and “consumes” three occurrences of type 2a, and otherwise it contributes 7 towards $f(F)$ and consumes four occurrences of type 2a. Each variable of type 2 “provides” four
occurrences of type 2a. If there are $\alpha$ variables of type 1 with an immediate associated contribution of 7.25, and $\beta$ variables with an immediate associated contribution of 7, then there are at least $0.75\alpha + \beta$ variables of type 2. If one takes these variables and average out the contribution among them, then we get $(7.25 + 0.75 \cdot 8)/(1.75) \approx 7.57$ from the $\alpha$ variables and $(7 + 8)/2 = 7.5$ from the $\beta$ variables. We see that the combined contributions of variables of types 1 and 2 is no lower than 7.5 per variable, and $2^{1/7.5} < 1.0969$. These variables also do not bring the running time too high.
Part III

Optimisation
Chapter 6

3-Hitting Set

In this chapter, we will look at the 3-Hitting Set problem, also known as the problem of finding a minimum transversal for a rank-3 hypergraph. We construct an algorithm for solving it, and analyse its running time in two parts: $O^*(1.6359^n)$ for any instance, and $O(|H| + p(k) \cdot 2.0755^k)$, for a polynomial $p$, when a hitting set of at most $k$ vertices is requested. We also provide a speedup that uses exponential space and runs in $O^*(1.6278^n)$ time. This chapter is based on the work in [82], with the main changes being a new section with a parameterised analysis (see below), and an improved handling of vertices of degree 2.

The bound $O(|H| + p(k) \cdot 2.0755^k)$ is an example of a parameterised bound, and such bounds are studied in the topic of parameterised complexity [24, 35]. In this field, one is interested in limiting the exponential (or otherwise super-polynomial) behaviour of an algorithm to a parameter: one typically gets a running time in $O(p_1(n) + p_2(k) \cdot c^k)$ for some provided parameter $k$ (where $p_1$ and $p_2$ are polynomials). In the case of 3-Hitting Set, a natural parameter is the size of the hitting set that is returned, and this is the parameter for which we analyse our algorithm. The previous best bounds in classic and parameterised contexts are $O^*(1.6538^n)$ when using polynomial space and $O^*(1.6316^n)$ with exponential space [82], both from one of our previous papers (which was the first to analyse this par-
ticular problem in a classical context), and a parameterised bound in $O(|H| + p(k) \cdot 2.179^k)$ by Fernau [33], improving on a bound of $O(|H| + p(k) \cdot 2.270^k)$ by Niedermeier and Rossmanith [64].

Note that the algorithm is identical when analysed in the parameterised and classical context: only the focus of the analysis differs.

Section 6.1 contains some further definitions and results that we will need in the rest of this chapter. After that, Section 6.2 presents the algorithm, Section 6.3 analyses the running time in terms of the parameter, and Section 6.4 presents the analysis for the classical case. Finally, Section 6.5 provides the way to speed up the algorithm at the cost of using exponential memory, and analyses the running time for this case.

### 6.1 More on Hypergraphs and Hitting Sets

Recall that a hypergraph $H$ is a collection of sets $\{E_1, \ldots, E_m\}$ referred to as hyperedges, and that a transversal or hitting set is a set $T$ such that $E \cap T \neq \emptyset$ for every $E \in H$.

A hypergraph is *simple* if, for all edges $E_i, E_j \in H$, $E_i \not\subset E_j$. $Min(H)$ is the hypergraph with edges $\{E \in H \mid \forall F \in H : F \not\subset E\}$ (and can be obviously be calculated in polynomial time). In other words, it is the hypergraph of all minimal hyperedges in $H$. Clearly, it is simple.

The *transversal hypergraph* $Tr(H)$ is the hypergraph where the hyperedges are all minimal transversals $T_i$ of $H$ (i.e. all transversals $T_i$ such that for every transversal $T$, $T \not\subset T_i$). In the general hypergraph literature, the problem of calculating $Tr(H)$ given $H$ has been given much more attention than the problem of finding a minimum transversal (see e.g. Berge’s book [4] or the papers by Eiter and Gottlob [27, 28]). Therefore, we use the less ambiguous phrase $k$-Hitting Set for the optimisation problem. A set $T$ is a minimal hitting set of $H$ if and only if it is a minimal hitting set of $Min(H)$ [4].

For a vertex $x$, $H[x = 1]$ is the hypergraph $\{E \in H \mid x \notin E\}$ and $H[x = 0]$ is the hypergraph $\{E - \{x\} \mid E \in H\}$, just as if the hypergraph were a CNF formula with only positive literals. A vertex $x$
6. 3-Hitting Set

is dominated by another vertex $y$ if $x \in E \Rightarrow y \in E$ for every $E \in \mathcal{H}$. Note that if $d(x) = 1$ then either $x$ is in a loop or $x$ is dominated by some other vertex.

For a hypergraph $\mathcal{H}$ and a hitting set $T$, the edges where $v$ is unique for some vertex $v \in T$ are $U_{\mathcal{H}}(v, T) = \{E \in \mathcal{H} \mid E \cap T = \{v\}\}$.

We will now show a result on how the minimum size of a hitting set depends on the maximum degree of $\mathcal{H}$.

**Lemma 58.** Let $\mathcal{H}$ be a 3-uniform hypergraph with $d(\mathcal{H}) = d$ and $d(x) \geq 2$ for every vertex $x$. Let $T$ be a minimum hitting set for $\mathcal{H}$. For $i \leq d$, define $T_i = \{v \in T \mid |U(v, T)| = i\}$ and let $k_i = |T_i|$. Then, the following holds:

$$\frac{n(\mathcal{H})}{d+1} \leq |T| \leq n(\mathcal{H}) \cdot \frac{d+1}{d+5} \quad (6.1)$$

**Proof.** We first prove two intermediate results. First, note that if any vertex $v \not\in T$ appears in the unique set for two vertices $x, y \in T_1$, then $T \cup \{v\} \setminus \{x, y\}$ is a smaller hitting set for $\mathcal{H}$ than $T$. Thus, we get:

$$n(\mathcal{H}) - |T| \geq 2k_1. \quad (6.2)$$

Second, since there are two occurrences of vertices not in $T$ for every edge in $U(v, T)$ for every $v \in T$, and at most $d$ occurrences per vertex, we get:

$$(n(\mathcal{H}) - |T|) \cdot d \geq \sum_i (2i k_i). \quad (6.3)$$

The upper bound of (6.1) follows from (6.2) and (6.3). First, assume $k_1 \geq 2|T|/(d + 1)$. Then, (6.2) gives:

$$n(\mathcal{H}) - |T| \geq 4|T|/(d + 1) \Rightarrow \quad (6.4)$$

$$n(\mathcal{H}) \geq |T| \cdot \left(1 + \frac{4}{d+1}\right) \Rightarrow \quad (6.5)$$

$$|T| \leq n(\mathcal{H}) \cdot \frac{d+1}{d+5} \quad (6.6)$$
Otherwise, we get $|T| - k_1 \geq |T| \cdot (d-1)/(d+1)$. Now, $\sum_i (2i k_i) \geq 4|T| - 2k_1$, so (6.3) gives:

$$n(H) - |T| \geq 2|T| + 2(|T| - k_1) \geq |T| \cdot (2 + 2 \frac{d-1}{d+1}) \Rightarrow$$

$$n(H) - |T| \geq |T| \cdot \frac{4}{d+1} \Rightarrow$$

$$|T| \leq n(H) \cdot \frac{d+1}{d+5} \tag{6.9}$$

In both cases, the result holds.

The lower bound of (6.1) follows from a separate line of reasoning: there are at most $2 \cdot |H| \leq 2d \cdot |T|$ occurrences of vertices not in $T$, and with every vertex having degree at least two, there can be at most $d \cdot |T|$ vertices not in $T$. In total, there can be at most $(d+1) \cdot |T|$ vertices.

### 6.2 An Algorithm for 3-Hitting Set

The algorithm that we use is given below as Algorithm 59. It takes a parameter $k$, with the semantics that if no hitting set of size at most $k$ is found, then the minimality of the returned hitting set is not guaranteed. Two “wrapper” functions are given: MinTrClassic, as Algorithm 60, for returning the smallest hitting set without providing a parameter, and MinTrParam, Algorithm 62, for solving the parameterised version of 3-Hitting Set; see Section 6.3 for details on the latter.

**Algorithm 59. MinTr($H$, $k$):**

0. If $H$ is empty, then return $\emptyset$. If $k = 0$, then return $V(H)$.

1. If $H$ is not simple, then return MinTr($Min(H)$, $k$).

2. If $H$ consists of connected components $C_1, \ldots, C_t$, then return $\bigcup_i$ MinTr($C_i$, $k_i$)
where \( k_1 = k - (t - 1) \) and \( k_{i+1} = k_1 - |\text{MinTr}(C_i, k_i)| \).

3. If there exists a loop \( \{x\} \), then return \( \{x\} \cup \text{MinTr}(\mathcal{H}[x = 1], k - 1) \).

4. If some vertex \( x \) is dominated by some other vertex, then return \( \text{MinTr}(\mathcal{H}[x = 0], k) \).

5. If \( \mathcal{H} \) is 3-uniform, then let \( d = d(\mathcal{H}) \). If \( k > n(\mathcal{H}) \cdot (d+1)/(d+5) \), then return \( \text{MinTr}(\mathcal{H}, \lfloor n(\mathcal{H}) \cdot (d+1)/(d+5) \rfloor) \). If \( k < n(\mathcal{H})/(d + 1) \), then return \( V(\mathcal{H}) \).

6. If there exists some 2-vertex \( x \) involved in edges \( E_1 \) and \( E_2 \), and there exists some edge \( E \subseteq ((E_1 \cup E_2) - \{x\}) \), then return \( \text{MinTr}(\mathcal{H}[x = 0], k) \).

7. If there exists some vertex \( v \) with \( d_2(v) > 0 \) and \( d(v) \geq 3 \), then let \( x \) be a vertex with maximum \( d(x) \) among all vertices with maximum \( d_2(x) \). If \( d_2(x) \geq 1 \) and \( d(x) \geq 3 \), then return
   \[
   \min(\{x\} \cup \text{MinTr}(\mathcal{H}[x = 1], k - 1), \text{MinTr}(\mathcal{H}[x = 0], k)).
   \]

8. If there exists some 2-vertex \( v \) with \( d_2(v) \geq 1 \), then let \( x \) be a vertex that maximises \( d_2(x) \) and let \( E_1, E_2 \) be the edges containing \( x \). Assuming \( |E_1| \leq |E_2| \), let \( E_1 = \{x, y\} \). If \( |E_2| = 2 \), then let \( E_2 = \{x, z\} \), and return
   \[
   \begin{align*}
   \min(\{x\} \cup \text{MinTr}(\mathcal{H}[x = 1, y = z = 0], k - 1), \\
   \{y, z\} \cup \text{MinTr}(\mathcal{H}[x = 0, y = z = 1], k - 2)).
   \end{align*}
   \]
   Otherwise, let \( E_2 = \{x, z, w\} \). Return
   \[
   \begin{align*}
   \min(\{x\} \cup \text{MinTr}(\mathcal{H}[x = 1, y = z = w = 0], k - 1), \\
   \{y, z\} \cup \text{MinTr}(\mathcal{H}[x = 0, y = 1], k - 1)).
   \end{align*}
   \]

9. If \( d(\mathcal{H}) \leq 3 \) and \( d(v) = 2 \) for some vertex \( v \), then assume that the edges containing \( v \) are \{\( v, w, x \}, \{v, y, z \}. \) Return
   \[
   \begin{align*}
   \min(\{x\} \cup \text{MinTr}(\mathcal{H}[v = 1, w = x = y = z = 0], k - 1), \\
   \text{MinTr}(\mathcal{H}[x = 0], k)).
   \end{align*}
   \]
10. Finally, pick a vertex $x$ with maximum $d(x)$ and return
\[
\min(\{x\} \cup \text{MinTr}(H|x = 1|, k - 1), \text{MinTr}(H|x = 0|, k)).
\]
Algorithm ends.

Our next algorithm is the simple wrapper for when one does not wish to provide a parameter.

**Algorithm 60.** MinTrClassic($H$):

1. Return MinTr($H, n(H)$).

Algorithm ends.

For the parameterised version, we apply a *reduction to problem kernel*, as given by Niedermeier and Rossmanith [64]. This is a common tool in parameterised complexity, used to reduce an input of arbitrary size to an instance with a size polynomial in the parameter $k$. This has the effect of reducing the dependence on $n$ in the upper bound on the running time.

**Lemma 61.** [Prop. 1 of [64]] There is a problem kernel of size $O(k^3)$ for 3HS, and it can be found in linear time.

The following algorithm is the parameterised wrapper, which uses the reduction to problem kernel result.

**Algorithm 62.** MinTrParam($H, k$):

1. Reduce the problem to its kernel, see Lemma 61, leaving an instance of size $O(k^3)$.

2. Let $T = \text{MinTr}(H, k)$. If $|T| \leq k$, then return $T$; otherwise return failure.

Algorithm ends.
6. 3-Hitting Set

The result of Lemma 58 is used to restrict the parameter $k$ further when $\mathcal{H}$ has a low maximum degree; essentially, since we know that some small solution does exist, we make sure that our algorithm does not waste time optimising solutions in those parts of the branching tree where the final solution is guaranteed to be too big.

We will now commence with the correctness proofs.

**Lemma 63.** For any rank-3 hypergraph $\mathcal{H}$ and parameter $k$, if there exists some hitting set of $\mathcal{H}$ of size at most $k$, then $\text{MinTr}(\mathcal{H}, k)$ returns the smallest hitting set of $\mathcal{H}$. Otherwise, $\text{MinTr}(\mathcal{H}, k)$ returns some hitting set of $\mathcal{H}$.

**Proof.** First, we show that the algorithm never creates an empty edge: no loops remain after case 3 has been passed, and the only cases that set more than one vertex to 0 are cases 8 and 9, which cannot create empty edges since case 6 has been passed.

Secondly, we show that the algorithm always returns some hitting set: the two base cases in case 0 both return a hitting set, case 1 returns a hitting set if it receives a hitting set for $\text{Min}(\mathcal{H})$, and in the other cases, no edge is removed without being hit by some included vertex (in case 2 each edge is included in some branch).

Finally, we show that unless $k$ is too small, the algorithm returns a smallest hitting set. For case 0, it is true by assumption. Cases 1–3 do not eliminate any smallest hitting sets (in case 2 each subcall returns a smallest hitting set for the component, unless $k$ is too small).

Case 4 never eliminates all smallest hitting sets: for any hitting set $T$ containing $x$, there exists some hitting set of size at most $|T|$ containing the dominating neighbour of $x$. The restrictions on $k$ in case 5 are safe by Lemma 58.

In case 6, it cannot be the case that all neighbours of $x$ are set to 0, and therefore for any hitting set $T$, $|U(x, T)| \leq 1$ and the branch $\mathcal{H}[x = 0]$ contains some hitting set of size at most $|T|$. In case 8, roughly the same reasoning applies: for any hitting set $T$, either no neighbour of $x$ is included in $T$ or $|U(x, T)| \leq 1$, and in the latter case, a hitting set of size at most $|T|$ exists in the branch $\mathcal{H}[x = 0]$. Since case 6 does not apply if this case is reached, the created subproblems
do not contain any empty edges. The same reasoning applies to case 9.

The remaining cases all use obviously safe branchings.

All analysis is performed through the model of finite global states, as defined in Section 3.4; we use the number of 2-edges in the hypergraph for our state, dividing into states “no 2-edges”, “one 2-edge”, and so on, up to the state “at least \( t \) 2-edges” for some maximum number of considered 2-edges \( t \), and the number of 2-edges is allowed to influence the available branchings. The measures we use, again according to the discussion in Section 3.4, are \( f_c(\mathcal{H}) = n(\mathcal{H}) - \Psi_i(m_2) \) and \( f_p(\mathcal{H}, k) = k - \Psi_i(m_2) \) for various \( i \); each \( \Psi_i \) represents one way to encode the influence of the state into the complexity measure, with \( \Psi_i(0) = 0 \) for the hardest case of having no 2-edges, and \( \Psi_i(m_2) > 0 \) for each \( m_2 > 0 \) being a constant weight assigned to state \( m_2 \). (Technically, each \( \Psi_i \) has its own associated maximum number of considered 2-edges \( t_i \), and \( \Psi_i(m_2) = \Psi_i(t_i) \) when \( m_2 > t_i \).) We use \( \Psi \) as a generic name when it does not matter which particular \( \Psi_i \) we use.

In this way, we get an easily calculated value for \( \Delta f \) along a branch: in a branch from a graph \( \mathcal{H} \) to a graph \( \mathcal{H}' \), we get \( \Delta f_c = \Delta n + \Delta \Psi \) and \( \Delta f_p = \Delta k + \Delta \Psi \), where \( \Delta \Psi = \Psi(m_2(\mathcal{H}')) - \Psi(m_2(\mathcal{H})) \): the influence of the states on \( \Delta f \) is the difference in the weight of the states of the instances \( \mathcal{H} \) and \( \mathcal{H}' \). Creating more 2-edges will cause \( \Psi \) to grow so that we get an increase in \( \Delta f \), while removing 2-edges causes \( \Psi \) to shrink so that \( \Delta f \) decreases (although in such a case, we would often have a better branching to begin with). We use \( \Delta \Psi(i) = \Psi(i) - \Psi(i - 1) \) to represent the incremental weight of the \( i \):th 2-edge (i.e. the gain of adding one 2-edge when \( i - 1 \) 2-edges exist, or the loss of removing one 2-edge when \( i \) 2-edges exist).

### 6.3 A Parameterised Analysis

Parameterised complexity theory \([24, 35]\) is a relatively recent field that roughly speaking studies the complexity of problems in terms of parameters other than the size of the problem. In particular, one is
interested in showing that a computationally hard problem (e.g. an NP-complete problem, such as 3HS) for which only super-polynomial exact algorithms are known can be solved in a time where the super-polynomial behaviour is restricted to the parameter— for an instance of size \( n \) and with a provided parameter of \( k \), one wants to show that a bound \( \mathcal{O}(p(n) \cdot f(k)) \) is possible, for a polynomial \( p(n) \) and any function \( f(k) \), e.g. vertex cover for ordinary graphs and a maximum size of the returned solution of \( k \) can be solved in time \( \mathcal{O}(1.2738^k + kn) \) [12]. If this can be done, then the problem is \textit{fixed-parameter tractable} (i.e. if the value of the parameter is fixed, then the running time is polynomial in the instance size). For other problems, it is believed that no such results are possible; see e.g. the book of Flum and Grohe [35] for an overview of the field.

The 3HS problem, as mentioned, belongs to the problems that are fixed-parameter tractable. In this section, we analyse \textsc{MinTr} in terms of the parameter \( k \) (which represents the maximum size of the solution), and show the bound \( \mathcal{O}(|\mathcal{H}| + p(k) \cdot 2.0755^k) \) on the running time.

Through our use of Lemma 58 to limit the size of the parameter, for instances with a small maximum degree the parameter limits the running time more strongly than the number of variables does. For this reason, the parameterised bound is used as a part of the classical, non-parameterised analysis in the next section, but it is also a new result in its own right.

To get a better bound, we use three different measures of complexity (i.e. three different assignments of weights to the states): \( f_{p,3}(\mathcal{H}, k) = k - \Psi_3(m_2) \) which is used when \( d(\mathcal{H}) \leq 3 \), \( f_{p,4}(\mathcal{H}, k) = k - \Psi_4(m_2) \) which is used when \( d(\mathcal{H}) = 4 \), and \( f_{p,5}(\mathcal{H}, k) = k - \Psi_{\geq 5}(m_2) \) which is used when \( d(\mathcal{H}) \geq 5 \). We use the name \( f_p \) to refer to the collection of these measures. The values are given in Tables 6.1 and 6.2. Using separate sets of weights for \( d(\mathcal{H}) < 5 \) is possible since the algorithm never increases the degree of any vertex. The sets of values \( \Psi \) use only as many weights (and distinguished states) as necessary to get a bound of 2.0755 or lower. We will see that the worst case is \( d(\mathcal{H}) = 3 \), with the worst-case branching number of 2.0755, and that
6.3. A Parameterised Analysis

\[ \Delta \Psi_3(\Delta m^2) = \Delta \Psi_4(\Delta m^2) \]

Table 6.1: Weights for states in the parameterised analysis \((d(H) \leq 3, d(H) = 4)\)

\[
\begin{array}{|c|c|c|c|c|}
\hline
\Delta m^2 & \Psi_3(\Delta m^2) & \Delta \Psi_3(\Delta m^2) & \Psi_4(\Delta m^2) & \Delta \Psi_4(\Delta m^2) \\
\hline
0 & 0 & - & 0 & - \\
1 & 0.293719 & 0.293719 & 0.287965 & 0.287965 \\
2 & 0.538580 & 0.244862 & 0.531472 & 0.243508 \\
3 & 0.714107 & 0.175526 & 0.712035 & 0.180563 \\
4 & 0.800684 & 0.086577 & 0.853398 & 0.141363 \\
5 & 0.800684 & 0 & 1.021995 & 0.168597 \\
\geq 6 & 0.800684 & 0 & 1.106711 & 0.084716 \\
\hline
\end{array}
\]

Table 6.2: Weights for states in the parameterised analysis \((d(H) \geq 5)\)

\[
\begin{array}{|c|c|c|}
\hline
\Delta m^2 & \Psi_{\geq 5}(\Delta m^2) & \Delta \Psi_{\geq 5}(\Delta m^2) \\
\hline
0 & 0 & - \\
1 & 0.268835 & 0.268835 \\
2 & 0.508065 & 0.239230 \\
3 & 0.705382 & 0.197317 \\
4 & 0.848886 & 0.143504 \\
\geq 5 & 0.924443 & 0.075557 \\
\hline
\end{array}
\]
the distribution of hard cases is such that adding more states to the
analysis will not improve it; we also see that all cases with \( d(\mathcal{H}) \geq 4 \)
get better branching numbers than when \( d(\mathcal{H}) = 3 \) (although this
does not influence our final bound).

We will show that the bound holds, by proving for each case of
the algorithm that all three measures result in branching numbers of
at most 2.0755. We begin by showing that the measures are well-
behaved.

**Lemma 64.** The measures \( f_{p,d} \) for \( d \in \{3,4\} \) are well-behaved for
\( \text{MinTr} \) when \( d(\mathcal{H}) \leq d \), and \( f_{p,5} \) is well-behaved for \( \text{MinTr} \) in all cases.

*Proof.* We take a look at the conditions for \( f_p(\mathcal{H},k) = k - \Psi(m_2) \)
being a well-behaved measure for \( \text{MinTr} \). It is clear that \( k = 0 \) implies
that \( \text{MinTr} \) finishes in polynomial time (strictly speaking, \( k = 0 \) and
\( m_2 > 0 \) can appear with \( f_p < 0 \) as a result, but since \( f_p \) will be
bounded from below by the maximum value of \( \Psi(m_2) \), this causes no
problem), so it remains to prove that any reduction in the algorithm
causes a non-negative reduction in \( f(\mathcal{H}) \), and that any branching step
in the algorithm causes a positive reduction in \( f(\mathcal{H}) \).

- Case 1 may remove some 2-edge, but if so, then there exists a
loop which will cause \( k \) to decrease. We need to look at the
whole process of reduction: removing \( i \) loops, along with all 2-
edges that intersect these loops, will reduce \( k \) by \( i \) and remove
up to \( i \cdot d(\mathcal{H}) \) 2-edges.

- Case 2 with \( t \) components requires at most \( t \) times the time
for solving a component with parameter \( k - (t - 1) \), which is
not higher than the time for solving a single component with
parameter \( k \).

- Case 4 will either increase the number of 2-edges, in case \( d_3(x) > 0 \),
or we have the case where \( x \) only appears in one edge \( \{x,y\} \);
setting \( x = 0 \), and subsequently \( y = 1 \), we reduce \( k \) and remove
up to \( d(\mathcal{H}) \) 2-edges.
6.3. A Parameterised Analysis

– Case 6 will only decrease \( m_2 \) through the immediate assignment if \( |E_1| = |E_2| = 2 \), in which case we have created two loops. Satisfying these decreases \( k \) by 2 and removes in total up to \( 2d(H) \) 2-edges.

– All other reductions reduce the value of \( k \).

We find that if we can ensure \( \Psi(t) - \Psi(t - d(H)) \leq 1 \) for any \( t \), then \( f_p \) is a well-behaved measure with respect to the reductions. We can also note as a general statement that setting \( v = 0 \) for any vertex \( v \) never increases \( f_p \), by the same reasoning as above, since any decrease in \( m_2 \) is always overtaken by the subsequent application of case 3. The branches of branchings where variables are assigned 0 without \( k \) being decreased all result in 2-edges being created so that the weight of the state increases.

The measures \( f_{p,3} \) and \( f_{p,5} \) are well-behaved since \( \Psi_3(m_2) \) and \( \Psi_\geq5(m_2) \) are both smaller than 1, and \( f_{p,4} \) is well-behaved since \( \Psi_4(m_2) - \Psi_4(m_2 - 4) < 1 \), for every value of \( m_2 \). \( \square \)

Next, we show the bounds for case 7.

**Lemma 65.** Case 7 results in a branching number of at most 2.0755 using the measure \( f_p \).

**Proof.** In the branch \( H[x = 1] \), we simply reduce \( k \) by 1 and remove \( d_2(x) \) 2-edges, while in the branch \( H[x = 0] \), things get a little more complicated. Remember that \( d_2(x) \) is maximum and \( d(x) \geq 3 \). In this branch, \( d_3(x) \) 2-edges and \( d_2(x) \) loops are created, and the 2-edges are all new, different, and do not intersect the loops. Assigning \( x = 0 \) and satisfying the loops, we reduce \( k \) by a total of \( d_2(x) \) and remove up to \( d_2(x)^2 - d_3(x) \) 2-edges, but at least \( d_3(x) \) 2-edges remain. Numbering the branches so that \( H_1 = H[x = 1] \) and \( H_2 = H[x = 0] \), we get \( \Delta_1 f_p \geq 1 - (\Psi(m) - \Psi(m - d_2(x))) \), and different expressions for \( \Delta_2 f_p \) depending on \( d_2(x) \):

- If \( d_2(x) = 1 \), then \( \Delta_2 f_p \geq 1 + \Delta\Psi(m_2 + 1) \);
- if \( d_2(x) = 2 \), then \( \Delta_2 f_p \geq 2 - (\Psi(m_2) - \Psi(max(m_2 - 3, 1))) \);


\[ \text{and} \]
### 6. 3-Hitting Set

<table>
<thead>
<tr>
<th>$d_2(x)$</th>
<th>$m_2$</th>
<th>$d(\mathcal{H}) \leq 3$ time</th>
<th>$d(\mathcal{H}) = 4$ time</th>
<th>$d(\mathcal{H}) \geq 5$ time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2.0755</td>
<td>2.0708</td>
<td>2.0554</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2.0755</td>
<td>2.0708</td>
<td>2.0556</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2.0755</td>
<td>2.0416</td>
<td>2.0545</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2.0648</td>
<td>1.9927</td>
<td>2.0560</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2.0708</td>
<td>2.0560</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
<td>2.0633</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2.0543</td>
<td>2.0446</td>
<td>2.0143</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2.0082</td>
<td>2.0139</td>
<td>2.0336</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1.9117</td>
<td>1.9895</td>
<td>2.0149</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1.7187</td>
<td>1.9416</td>
<td>1.8432</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1.6374</td>
<td>1.8573</td>
<td>1.7014</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2.0755</td>
<td>2.0708</td>
<td>2.0560</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1.8438</td>
<td>1.9193</td>
<td>1.9342</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1.6821</td>
<td>1.9042</td>
<td>1.8086</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1.6078</td>
<td>1.8521</td>
<td>1.6905</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>-</td>
<td>2.0708</td>
<td>2.0560</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>-</td>
<td>1.8746</td>
<td>1.7576</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>-</td>
<td>1.7246</td>
<td>1.5942</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>-</td>
<td>-</td>
<td>2.0560</td>
</tr>
</tbody>
</table>

Table 6.3: Branching numbers for case 7
6.3. A Parameterised Analysis

- if $d_2(x) > 2$, then we just use $\Delta_2 f_p \geq d_2(x) - \Psi(m_2)$.

Table 6.3 lists the branching number for the different combinations of $d_2(x)$ and $m_2$, using $\Psi_3$, $\Psi_4$, and $\Psi_{\geq 5}$. The case $m_2 > 6$ does not introduce any harder cases when six or fewer weights are used, and $d_2(x) \leq d(\mathcal{H})$ naturally holds.

Case 8 is dealt with next; we will show that it is easy.

**Lemma 66.** Case 8 results in a branching number no higher than 2 using the measure $f_p$.

**Proof.** We have a number of possibilities in this case, but they are all easy. Remember that for every vertex $v$, if $d_2(v)$ is maximal, then $d(v) = 2$.

1. If the first subcase is used, then either $d_2(y) = d_2(z) = 1$, or some assignment $w = 1$ is made in the branch $\mathcal{H}[x = 1, y = z = 0]$ (possibly from an edge $\{y, z, w\}$). In the former case, a net of at most one 2-edge is removed in the branch $\mathcal{H}[x = 1, y = z = 0]$ branch and only two 2-edges are removed in the $\mathcal{H}[x = 0, y = z = 1]$ branch, for a branching of $\tau(1 - \Delta \Psi(m_2), 2 - (\Psi(m_2) - \Psi(m_2 - 2))) \leq \tau(1 - \Psi_3(1), 2 - \Psi_3(2)) < 1.9516$. In the latter case, $k$ decreases by 2 in both branches and $m_2$ decreases by no more than 4, for a branching of at most $\tau(1, 1) = 2$.

2. If the second subcase is used and $d_2(y) = 1$, then in both the immediate branches $\mathcal{H}[x = 1, y = z = w = 0]$ and $\mathcal{H}[x = 0, y = 1]$, the value of $m_2$ is at least as high as in $\mathcal{H}$, or higher in the first branch if $d(\mathcal{H}) = 3$ since no pair of vertices occur three times in edges together, or we have an extra assignment $t = 1$ removing a limited number of 2-edges, and we get a branching of $\tau(1, 1)$ or better.

In all of these cases, we get a branching number no higher than 2. 

So far, the branchings have been largely independent of $d(\mathcal{H})$, since the primary condition for selecting a branching variable has been the existence of 2-edges. Now, the instance is 3-uniform and $d(\mathcal{H})$ becomes important.
Lemma 67. Cases 9 and 10 result in branching numbers of at most 2.0755.

Proof. When \( d(\mathcal{H}) \leq 3 \), either \( \mathcal{H} \) is 3-regular or case 9 is used. Since the algorithm never creates new edges, we can only get a 3-regular instance at most once in every path of the branching tree, so the contribution of this case to the overall running time can be ignored. If case 9 is used, then we get at worst a branching number of \( \tau(1 + \Psi_3(2), \Psi_3(2)) < 2.0755 \).

If case 10 is used, then the base branching is \( \tau(1, \Psi(d(\mathcal{H}))) \), which is good enough with \( d(\mathcal{H}) \geq 5 \) at \( \tau(1, \Psi_{\geq 5}(5)) < 2.0560 \). With \( d(\mathcal{H}) = 4 \), we also consider that the second branch in this branching will hit one further interesting case.

1. If we hit a reduction which decreases \( k \), then we get a branching number of at most \( \tau(1, 1) = 2 \) counting \( k \) alone.

2. If we hit a reduction which increases the number of 2-edges, then we get a branching of \( \tau(1, \Psi_4(5)) < 1.9851 \). Every reduction ends up in one of these two cases.

3. If case 7 is reached, branching on a vertex \( y \) with \( d_2(y) < d_2(x) \) (since \( x \) does not dominate \( y \) in \( \mathcal{H} \) and no new 2-edges are assumed to have been created), then using the same branches as previously for these cases (see the proof of Lemma 65), we get:

   - \( \tau(1, 1 + \Psi_4(3), 1 + \Psi_4(5)) < 2.0708 \) when \( d_2(y) = 1 \),
   - \( \tau(1, 1 + \Psi_4(2), 2 + \Psi_4(1)) < 2.0708 \) when \( d_2(y) = 2 \), and
   - \( \tau(1, 1 + \Psi_4(1), 3) < 2.0509 \) when \( d_2(y) = 3 \).

4. If case 8 is reached immediately in the \( \mathcal{H}[x = 0] \) branch, then consider which subcase we hit, and use the branchings given in the proof of Lemma 66. If we hit the first subcase, then we either get a branching of \( \tau(1, 1 + \Psi_4(3), 2 + \Psi_4(2)) < 1.9708 \), or a branching \( \tau(1, 2, 2) = 2 \) in terms of \( k \). If we hit the second subcase, then we get \( \tau(1, 1 + \Psi_4(4), 1 + \Psi_4(4)) < 2.0735 \), or \( \tau(1, 2, 2) = 2 \), or \( \tau(1, 1 + \Psi_4(4), 2) < 2.0362 \).
6.3. A Parameterised Analysis

This finishes the case enumeration, and we see that every case has a branching number of at most 2.0755. Figure 6.1 illustrates the state diagram corresponding to the hard cases of this analysis. The arrows represent branches, and are labelled with the reduction in $k$ along the branch. For instance, the state “one 2-edge” has one arrow, label 1, leading to state “no 2-edge” and one arrow, label 1, leading to state “two 2-edges”, representing the 2-way branching of case 7 with $d_2(x) = 1$, $m_2 = 1$.

Finally, we state the main results.

**Theorem 68.** MinTr($\mathcal{H}, k$) runs in time $O\left(p(n) \cdot 2.0755^k\right)$, where $p(n)$ is polynomial in $n$.

**Proof.** By Lemmas 64–67, the measures are well-behaved and every case has a branching number of at most 2.0755. □

**Corollary 69.** The algorithm MinTrParam with parameter $k$ solves 3-Hitting Set in time $O\left(\left|\mathcal{H}\right| + p(k) \cdot 2.0755^k\right)$ if a hitting set of size at most $k$ exists.

**Proof.** The “reduction to problem kernel” step described in [64] runs in linear time and leaves an instance of size $O\left(k^3\right)$. The result follows from this and Theorem 68. □
6.4 A Non-Parameterised Analysis

In this section, we give an upper bound on the running time of MinTr-Classic. For low-degree cases, the parameterised analysis in the previous section in combination with Lemma 58 (or rather, the application of Lemma 58 in case 5 of the algorithm) provides a good bound; for the rest of the cases, we analyse the running time by the same method, using new values of \(\Psi(m_2)\) and the guarantee that high-degree vertices exist. This analysis is again performed using two measures: one for \(d(\mathcal{H}) \leq 8\) and one for the case of unbounded degree.

First, we use Theorem 68 to give a classic bound for the running time of MinTr when \(d(\mathcal{H}) \leq 7\).

**Lemma 70.** For a 3-uniform hypergraph \(\mathcal{H}\) with \(d(\mathcal{H}) \leq 7\), algorithm MinTr runs in time \(O^*(1.6272^n)\) (regardless of the value of \(k\)).

**Proof.** Case 5 guarantees that \(k \leq 2n/3\), and by Theorem 68, the running time of MinTr will thus be in \(O^*(2.0755^{2n/3}) \subset O^*(1.6272^n)\).

Now that the groundwork of the case analysis has been done in Section 6.3 (albeit for another measure), bounding the running time of MinTr in a classic context is easier, so we perform the proof without dividing into lemmas.

The analysis uses the measures \(f_8(\mathcal{H}) = n - \Psi_8(m_2)\) for the case \(d(\mathcal{H}) = 8\) and \(f_{\geq 9}(\mathcal{H}) = n - \Psi_{\geq 9}(m_2)\) for the case \(d(\mathcal{H}) \geq 9\), with all weights given in Table 6.4.

**Theorem 71.** For a hypergraph \(\mathcal{H}\), MinTr runs in time \(O^*(1.6359^n)\).

**Proof.** First, the measures are well-behaved: \(\Psi_8(m_2) - \Psi_8(m_2 - 8) < 1\), and \(\Psi_{\geq 9}(m_2) < 1\) for every value of \(m_2\), and one assignment \(v = 1\), or the existence of a loop \(\{v\}\), can remove no more than \(d(\mathcal{H})\) 2-edges.

In case 7, the case enumeration is the same as in Theorem 68, except that \(\Delta n\) is often bigger than \(\Delta k\) (and easy to find: \(\Delta_1 n \geq 1\) and \(\Delta_2 n \geq d_2(x) + 1\) when branching on a vertex \(x\)). Since \(\Delta \Psi(m_2)\) decreases with increasing \(m_2\) for both \(\Psi_8\) and \(\Psi_{\geq 9}\), we can calculate some of the branching numbers in summarised form, rather than list
### 6.4. A Non-Parameterised Analysis

#### Table 6.4: Weights for states in non-parameterised analysis ($d(\mathcal{H}) = 8$, $d(\mathcal{H}) \geq 9$).

<table>
<thead>
<tr>
<th>$m_2$</th>
<th>$\Psi_8(m_2)$</th>
<th>$\Delta \Psi_8(m_2)$</th>
<th>$\Psi_{\geq 9}(m_2)$</th>
<th>$\Delta \Psi_{\geq 9}(m_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0.115054</td>
<td>0.115054</td>
<td>0.111074</td>
<td>0.111074</td>
</tr>
<tr>
<td>2</td>
<td>0.230104</td>
<td>0.115050</td>
<td>0.221901</td>
<td>0.110827</td>
</tr>
<tr>
<td>3</td>
<td>0.345147</td>
<td>0.115043</td>
<td>0.332277</td>
<td>0.110375</td>
</tr>
<tr>
<td>4</td>
<td>0.460178</td>
<td>0.115031</td>
<td>0.441829</td>
<td>0.109553</td>
</tr>
<tr>
<td>5</td>
<td>0.575187</td>
<td>0.115009</td>
<td>0.549883</td>
<td>0.108054</td>
</tr>
<tr>
<td>6</td>
<td>0.690156</td>
<td>0.114969</td>
<td>0.655212</td>
<td>0.105329</td>
</tr>
<tr>
<td>7</td>
<td>0.805051</td>
<td>0.114895</td>
<td>0.755603</td>
<td>0.100390</td>
</tr>
<tr>
<td>8</td>
<td>0.919809</td>
<td>0.114759</td>
<td>0.847087</td>
<td>0.091484</td>
</tr>
<tr>
<td>9</td>
<td>1.034320</td>
<td>0.114511</td>
<td>0.922664</td>
<td>0.075577</td>
</tr>
<tr>
<td>10</td>
<td>1.148376</td>
<td>0.114056</td>
<td>0.970305</td>
<td>0.047641</td>
</tr>
<tr>
<td>11</td>
<td>1.261599</td>
<td>0.113223</td>
<td>0.970305</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>1.373300</td>
<td>0.111701</td>
<td>0.970305</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>1.482223</td>
<td>0.108923</td>
<td>0.970305</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1.586089</td>
<td>0.103865</td>
<td>0.970305</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1.680800</td>
<td>0.094711</td>
<td>0.970305</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>1.759100</td>
<td>0.078301</td>
<td>0.970305</td>
<td>0</td>
</tr>
<tr>
<td>$\geq 17$</td>
<td>1.808495</td>
<td>0.049394</td>
<td>0.970305</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 6.5: Branching numbers for case 7

<table>
<thead>
<tr>
<th>$d_2(x)$</th>
<th>$m_2$</th>
<th>$d(H) = 8$ time</th>
<th>$d(H) \geq 9$ time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1–10</td>
<td>1.6359 (for all $m_2$)</td>
<td>1.6353 (for all $m_2$)</td>
</tr>
<tr>
<td>1</td>
<td>11–17</td>
<td>1.6359 (for all $m_2$)</td>
<td>1.6182 (for all $m_2$)</td>
</tr>
<tr>
<td>2</td>
<td>Any</td>
<td>1.5801</td>
<td>1.5749</td>
</tr>
<tr>
<td>3</td>
<td>Any</td>
<td>1.5784</td>
<td>1.5571</td>
</tr>
<tr>
<td>4</td>
<td>Any</td>
<td>1.5915</td>
<td>1.4912</td>
</tr>
<tr>
<td>5</td>
<td>Any</td>
<td>1.5343</td>
<td>1.4504</td>
</tr>
<tr>
<td>6</td>
<td>Any</td>
<td>1.5064</td>
<td>1.4296</td>
</tr>
<tr>
<td>7</td>
<td>Any</td>
<td>1.5118</td>
<td>1.4251</td>
</tr>
<tr>
<td>8</td>
<td>Any</td>
<td>1.5859</td>
<td>1.4382</td>
</tr>
<tr>
<td>9</td>
<td>Any</td>
<td>-</td>
<td>1.4763</td>
</tr>
<tr>
<td>10</td>
<td>Any</td>
<td>-</td>
<td>1.5441</td>
</tr>
</tbody>
</table>

every single combination of $d_2(x)$ and $m_2$. For $d_2(x) = 1$ and any value of $m_2$, the branching is $\tau(1 - \Delta \Psi(m_2), 2 + \Delta \Psi(m_2 + 1))$, while for $d_2(x) > 1$ we use $\tau(1 - \Psi(d_2(x)), (d_2(x) + 1) - \Psi(t))$ where $t = 3$ for $d_2(x) = 2$, $t = 9$ for $d_2(x) = 3$, $t = 16$ for $d_2(x) = 4$, and $t = 17$ for any $d_2(x) \geq 5$. The branching numbers are given in Table 6.5. We see that every case with $d_2(x) = 1$ and $d(H) = 8$, and only those cases, are the hard cases for this part of the algorithm.

In case 8 of the algorithm, note that less than 8 2-edges will have been removed in each branch. Thus, the branching number in the first subcase is dominated by $\tau(2, 2) < 1.4143$ and in the second subcase by $\tau(1, 3) < 1.4656$.

In case 9, there are no 2-edges and the branching in terms of $n$ is $\tau(1, 5) < 1.3248$.

Finally, in case 10, if $d(H) \leq 7$, then Lemma 70 applies; with $d(H) = 8$, we get a branching $\tau(1, 1 + \Psi_8(8)) < 1.6359$; and with $d(H) \geq 9$, we get a branching $\tau(1, 1 + \Psi_{\geq 9}(9)) < 1.6353$. Note that since the branching number calculation of case 7 with $d_2(x) = 1$, $d(H) = 8$, and $m_2 = 17$ uses $\Psi_8(18)$, the number calculated is not completely tight. However, the time for a branching tree where only case 7 with $d_2(x) = 1$ and case 10 with $d(x) = 8$ can be calculated
by balancing the branchings $\tau(1, 1 + 8w)$ and $\tau(1 - w, 2 + w)$, for the
result $w \approx 0.115$ yielding a branching number which rounds to 1.6359,
so the non-tightness is limited.

No illustration of the loop of hard cases is given for this lemma,
since the number of cases is so high, and since the diagram would be
so regular: for $m_2 = 0$, case 10 with $d(x) = 8$ would be used, leading
to states $m_2 = 0$ and $m_2 = 8$; for $1 \leq m_2 \leq 16$, case 7 with $d_2(x) = 1$
would be used, leading to states $m_2 - 1$ and $m_2 + 1$; and for the state
$m_2 \geq 17$, case 7 with $d_2(x) = 1$ would again be used, but in this case
the exiting arrows would lead to states 16 and $\geq 17$ (since no state
18 exists).

**Corollary 72.** $\text{MinTrClassic}(\mathcal{H})$ runs in time $O^*(1.6359^n)$.

## 6.5 An Exponential-Space Speedup

In this section, we show how to modify the algorithm to run in
$O^*(1.6278^n)$ time by modifying it to use $O^*(1.6278^n)$ memory.

The modification uses an idea by Robson [71], where one uses dy-
namic programming to store all instances with up to $\alpha n$ vertices (if
the input instance has $n$ vertices). For the Independent Set problem
on ordinary graphs, which is the context of Robson’s paper, the basic
speedup is straightforward: given that every instance which appears
in the branching tree (reached by some combination of branches and
reductions) is an induced subgraph of the input graph, there are only
$\binom{n}{\alpha n}$ possible unique instances of size $\alpha n$. When the instances are hy-
pergraphs, this cannot be guaranteed, but if we cache every instance
which is known to be an induced subgraph of the input graph, then
it turns out that the remaining instances are easier to solve.

**Lemma 73.** Starting from a simple input instance $\mathcal{H}$, every 3-uniform
instance $\mathcal{H}'$ that appears in the branching tree of $\text{MinTrClassic}(\mathcal{H})$
equals $\{E \mid E \in \mathcal{H}, |E| = 3, E \subseteq V\}$ for some vertex set $V \subseteq V(\mathcal{H})$.

**Proof.** Let $V$ be the vertices of $\mathcal{H}'$. Since the algorithm never creates
a 3-edge, every edge in $\mathcal{H}'$ must exist in $\mathcal{H}$. Assume that for some
3-edge $E \in \mathcal{H}$ with $E \subseteq V$, $E$ does not exist in $\mathcal{H}'$. The only possible case that could have removed the edge $E$ is the minimisation step, but this would have required an edge $E' \subset E$ to exist at some point, which does not exist in $\mathcal{H}'$ (as $|E'| < 3$). As the algorithm only removes such edges in connection with an assignment $v = 1$ for some vertex $v$ in the edge, this cannot be.

Now, we present the algorithm that is used when $\mathcal{H}$ has few enough vertices that a 3-uniform instance would fit in the cache. In this case, we want to perform the search in a manner that minimises the number of created 2-edges (since the search ends when no 2-edges remain).

**Algorithm 74.** MTCacheSearch($\mathcal{H}$):

0. If $\mathcal{H}$ is empty, then return $\emptyset$.

1. If there is a loop $\{x\} \in \mathcal{H}$, then return $\{x\} \cup \text{MTCacheSearch}(\mathcal{H}[x = 1])$.

2. If $d_2(x) > 1$ for some vertex $x$, then return $\min(\{x\} \cup \text{MTCacheSearch}(\mathcal{H}[x = 1]), \text{MTCacheSearch}(\mathcal{H}[x = 0]))$.

3. If $d_2(x) = 1$ for some vertex $x$, then let the 2-edge containing $x$ be $\{x, y\}$ and return

   $\min(\{x\} \cup \text{MTCacheSearch}(\mathcal{H}[x = 1]), \{y\} \cup \text{MTCacheSearch}(\mathcal{H}[y = 1]))$.

4. If the solution has been previously calculated, then return it from the cache.

5. Otherwise, pick some 3-edge $\{x, y, z\} \in \mathcal{H}$ and return

   $\min(\{v\} \cup \text{MinTrExp}(\mathcal{H}[v = 1]) \mid v \in \{x, y, z\})$

   and remember the result.
Algorithm ends.

For simplicity, let us describe the overall process of calculating a smallest hitting set for an input instance $\mathcal{H}_0$, having $n_0$ vertices, by the following steps:

1. A cache is filled with the smallest hitting set for every hypergraph $\{E \mid E \in \mathcal{H}, |E| = 3, E \subset V\}$ for every $V \subset V(\mathcal{H}_0)$ with $|V| \leq \alpha n_0$, taking polynomial time for each entry in the cache.

2. Modify MinTr to call MTCacheSearch for each instance $\mathcal{H}$ with $n(\mathcal{H}) \leq \alpha n_0$. Call the new version MinTrExp. MTCacheSearch is given as Algorithm 74; since the cache has been filled in advance, MTCacheSearch does only polynomial work on a 3-uniform instance.

For the right values of $\alpha$, MinTrExp will run faster than MinTr, as we will prove in the rest of this section. Unfortunately, we are not able to use the parameterised perspective in the analysis of the speedup, which is the reason that the gap between the classical bounds for polynomial and exponential space is relatively small—$O^*(1.6359^n)$ versus $O^*(1.6278^n)$. For this reason, we also need to provide a single complexity measure $f_\ast = n - \Psi_\ast(m_2)$ that is valid in all cases.

The rest of the section is laid out as follows: Lemma 75 bounds the time used for “filling the cache”, Lemma 76 bounds the time MTCacheSearch requires for a non-3-uniform instance, Lemma 77 contains the analysis of MinTr in terms of the measure $f_\ast$, and finally Theorem 78 puts the pieces together into a single bound.

**Lemma 75.** The total time for all calls to MTCacheSearch($\mathcal{H}$) with 3-uniform instances $\mathcal{H}$, $n(\mathcal{H}) \leq \alpha n_0$, that are not contained in the cache at the time the call is made is in $O^*\left(\binom{n_0}{\alpha n_0}\right)$.

**Proof.** In the final branching of MinTr, each subinstance is a 3-uniform instance with fewer variables. The time required to calculate and fill in an entry on $\mathcal{H}$, therefore, is polynomial plus the time required to calculate and fill in entries for any of these subinstances that had not
Table 6.6: Weights for states in the purely parameter-free analysis.

<table>
<thead>
<tr>
<th>$m_2$</th>
<th>$\Psi_\ast(m_2)$</th>
<th>$\Delta\Psi_\ast(m_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0.233568</td>
<td>0.233568</td>
</tr>
<tr>
<td>2</td>
<td>0.449745</td>
<td>0.216177</td>
</tr>
<tr>
<td>3</td>
<td>0.619111</td>
<td>0.169366</td>
</tr>
<tr>
<td>4</td>
<td>0.758052</td>
<td>0.138941</td>
</tr>
<tr>
<td>$\geq 5$</td>
<td>0.854487</td>
<td>0.096435</td>
</tr>
</tbody>
</table>

been previously calculated. Every time an instance which has an entry in the tree is reached (possibly in another branch of the same call to MTCacheSearch), the recursion stops in polynomial time. Therefore, the total work can be distributed into a polynomial amount of local work for each entry in the cache, showing that the total work is a polynomial times the number of entries.

**Lemma 76.** If the cache contains the smallest hitting set for every hypergraph $\{E \mid E \in \mathcal{H}, |E| = 3, E \subset V\}$ with $V \subseteq V(\mathcal{H})$, then MTCacheSearch($\mathcal{H}$) returns the smallest hitting set of $\mathcal{H}$ in time $O^\ast(1.4656^n(\mathcal{H}))$.

**Proof.** Either the total time is in $O^\ast\left(2^{n/2}\right) \subset O^\ast\left(1.4143^n\right)$ or we have a branching of $\tau(1,3) < 1.4656$ in terms of $n$, counting the assignments due to the loops.

The measure $f_\ast(\mathcal{H}) = n(\mathcal{H}) - \Psi_\ast(m_2)$ uses the values for $\Psi_\ast$ given in Table 6.6.

**Lemma 77.** Using the measure $f_\ast$, every branching number of MinTrExp when $n(\mathcal{H}) > \alpha n_0$ is at most 1.6685.

**Proof.** Since $\Psi_\ast(m_2) < 1$ for every $m_2$, the measure is well-behaved, as every reduction that removes 2-edges also removes some variable.

In case 7, the case enumeration is once again the same as in Theorems 68 and 71. The branchings and the corresponding branching
6.5. An Exponential-Space Speedup

<table>
<thead>
<tr>
<th>$d_2(x)$</th>
<th>$m_2$</th>
<th>Branching</th>
<th>Br. number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$\tau(1 - \Psi_<em>(1), 2 + \Delta \Psi_</em>(2))$</td>
<td>1.6646</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$\tau(1 - \Delta \Psi_<em>(2), 2 + \Delta \Psi_</em>(3))$</td>
<td>1.6670</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>$\tau(1 - \Delta \Psi_<em>(3), 2 + \Delta \Psi_</em>(4))$</td>
<td>1.6531</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>$\tau(1 - \Delta \Psi_<em>(4), 2 + \Delta \Psi_</em>(5))$</td>
<td>1.6497</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>$\tau(1 - \Delta \Psi_*(5), 2)$</td>
<td>1.6543</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$\tau(1 - \Psi_<em>(2), 3 - \Delta \Psi_</em>(2))$</td>
<td>1.6609</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$\tau(1 - (\Psi_<em>(3) - \Psi_</em>(1)), 3 - (\Psi_<em>(3) - \Psi_</em>(1)))$</td>
<td>1.6574</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$\tau(1 - (\Psi_<em>(4) - \Psi_</em>(2)), 3 - (\Psi_<em>(4) - \Psi_</em>(1)))$</td>
<td>1.6456</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>$\tau(1 - (\Psi_<em>(5) - \Psi_</em>(3)), 3 - (\Psi_<em>(5) - \Psi_</em>(2)))$</td>
<td>1.5920</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$\tau(1 - \Psi_<em>(3), 4 - \Psi_</em>(3))$</td>
<td>1.6685</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>$\tau(1 - (\Psi_<em>(4) - \Psi_</em>(1)), 4 - \Psi_*(4))$</td>
<td>1.6267</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>$\tau(1 - (\Psi_<em>(5) - \Psi_</em>(2)), 4 - \Psi_*(5))$</td>
<td>1.5785</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$\tau(1 - \Psi_<em>(4), 5 - \Psi_</em>(4))$</td>
<td>1.6626</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>$\tau(1 - (\Psi_<em>(5) - \Psi_</em>(1)), 5 - \Psi_*(5))$</td>
<td>1.5651</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$\tau(1 - \Psi_<em>(5), 6 - \Psi_</em>(5))$</td>
<td>1.6685</td>
</tr>
</tbody>
</table>

Table 6.7: Branching numbers for case 7 using $\Psi_*$.
numbers are given in Table 6.7. We see that the worst-case branching number of 1.6685 appears when \( d_2(x) = m_2 = 3 \) and \( d_2(x) = m_2 = 5 \).

In case 8 of the algorithm, we once again get a branching dominated by either \( \tau(2, 2) \) or \( \tau(1, 3) \), and in case 9 we get a branching of \( \tau(1, 5) < 1.3248 \).

Finally, in case 10, we can again ignore the 3-regular case since it only appears at most once in every path of the tree. If \( d(H) \geq 5 \), then we get a branching of \( \tau(1, 1 + \Psi_*(5)) < 1.6515 \). If \( d(H) = 4 \), then the immediate branching is not good enough, and we have to consider what happens in the second branch.

- If we hit a reduction that either increases \( m_2 \) or removes some variable, then we get a branching dominated by \( \tau(1, 2) < 1.6181 \) or \( \tau(1, 1 + \Psi_*(5)) < 1.6515 \).

- If we hit case 7, then assume that we are branching on a variable \( y \). If \( d_2(y) = 1 \), then we get \( \tau(1, 2 + \Psi_*(3), 3 + \Psi_*(5)) < 1.6685 \); if \( d_2(y) = 2 \), then we get \( \tau(1, 2 + \Psi_*(2), 4 + \Psi_*(1)) < 1.6678 \); if \( d_2(y) = 3 \), then we get \( \tau(1, 2 + \Psi_*(1), 5) < 1.6641 \); and no higher value of \( d_2(y) \) is possible since \( d(H) = 4 \) and \( x \) does not dominate \( y \) in \( H \). This covers case 7.

- If we hit case 8, then we get branchings of \( \tau(1, 4, 4) < 1.5437 \) or \( \tau(1, 3, 5) < 1.5702 \).

This covers every possibility. \( \square \)

Now, we have all the tools we need to pick a value of \( \alpha \) and give the time and space requirements for the algorithm.

**Theorem 78.** The running time for MinTrExp for the input instance \( \mathcal{H}_0 \), not counting the time needed for filling the cache, is in \( O^*(1.6685^{(1-\alpha)n_0} \cdot 1.4656^{\alpha n_0}) \). The time needed for filling the cache is in \( O^*(\alpha^{-\alpha n_0} \cdot (1 - \alpha)^{-\alpha n_0}) \), which is also the memory requirements for the cache. With \( \alpha \approx 0.190675 \), they balance at \( O^*(1.6278^{n_0}) \) time and space requirement.
Proof. Consider the whole branching tree starting at the instance $H_0$. Every node with a subinstance $H$ with $n(H) > \alpha n_0$ has a branching number, as analysed by $f_*$, of at most 1.6685, by Lemma 77. This implies that the tree formed by only these nodes has $O\left(1.6685^{(1-\alpha)n_0}\right)$ “leaves”, each with a subtree of size $O\left(1.4656^{\alpha n_0}\right)$. The total size of the branching tree is the product of these.

It is well known that $n!$ is within a polynomial factor of $(n/e)^n$ (by Stirling’s approximation, see [43]). Through standard algebraic manipulation, starting from $\binom{n}{k} = n!/(k!\cdot(n-k)!)$, we get the desired form.

With $\alpha \approx 0.190675$, both parts are in $O^*\left(1.6278^{n_0}\right)$. \qed
Part IV

Counting Problems
Chapter 7

Counting 2SAT

This chapter gives an algorithm for \#2sat\_w and an upper bound on its running time of \(O^*(1.2377^n)\), and lays down some common groundwork for the \#2sat\_w and \#3sat\_w algorithms. The work in this chapter is based on our previous publications with Dahlöf and Jonsson [14, 15], where the bound \(O^*(1.2561^n)\) was given; Fürer and Kasiviswanathan [40] improved this bound to \(O^*(1.2461^n)\) through a refined analysis. The improvement in the bound in this chapter is due to improvements in the analysis, which is now performed as a compound analysis in multiple attributes (where it previously was performed as a compound analysis in two attributes only).

For technical reason (because we need the cardinality vector in our algorithm), we consider an extended variant of the problem. In addition to a \(k\)-CNF formula \(F\) and the weight vector \(w\) we use a cardinality vector \(c\) associating a cardinality with each literal, and define the problem as follows: Let the weight of a model \(M\) for \(F\) be

\[
\mathcal{W}(M) = \sum_{l \text{ is true in } M} w_l
\]

and the cardinality be

\[
\mathcal{C}(M) = \prod_{l \text{ is true in } M} c_l.
\]

What is the sum of \(\mathcal{C}(M)\) for all max-weight models \(M\)?
Recall that in this chapter, as well as Chapter 8, we treat clauses essentially as sets of literals, i.e. a clause contains no more than one copy of a literal.

Section 7.1 provides some concepts that will be used by our algorithms for \#2SAT\(_w\) and \#3SAT\(_w\), and gives our algorithm for the \#2SAT\(_w\) problem. Section 7.2 gives an upper bound on its running time for the case that \(d(F) \leq 4\), using the method of analysis by average degree, and Section 7.3 gives an upper bound on the running time in the general case, using a standard weight-based measure approach.

7.1 Algorithm Preliminaries

The algorithm for the \#2SAT\(_w\) problem is not complicated, but there is a fair bit of book-keeping involved. To begin with, the standard operation \(F[x = 1]\) cannot be directly used in recursion: let \(F = (x \lor y)\); \(F\) has 3 models. However, \(\#2\text{SAT}(F[x = 1]) + \#2\text{SAT}(F[x = 0]) = 1 + 1\) since \(F[x = 1]\) is the empty formula which has one solution by definition. The variable \(y\) is “lost” in \(F[x = 1]\), and to get a correct answer, we have to keep track of such variables. We define a function Prop for managing the propagation of effects that \(F[l = b]\) would perform, and in addition the managing of the reductions associated with 1-clauses and subsumed clauses. Since the same procedure is needed in both the \#2SAT and \#3SAT algorithms, we define a Prop procedure that handles both 2- and 3-clauses. For the extent of this chapter, let \(F\{x = 1\}\) be the result of replacing every occurrence of the literal \(x\) in \(F\) by 1, and every occurrence of \(\bar{x}\) by 0, without otherwise modifying the formula.

When we talk of the graph of a 2SAT formula \(F\), this is mainly an analogous term: we consider a graph as defined in Section 2.2, where we have one vertex for every variable in \(F\), and one edge \((a, b)\) for every 2-clause \((\bar{a}, b)\) in \(F\), where \(\bar{v}\) is \(v\) or \(\bar{v}\), and let terms such as connected component and subgraph hold the meaning they would have in this graph.

Our main algorithm \(C(F, c, w)\), taking a formula \(F\), a cardinality vector \(c\), and a weight vector \(w\), is defined as Algorithm 83 later in
Prop($F, c, w$) is defined as Algorithm 79. It returns a tuple ($F, c, w$) where $F$ is the resulting formula, $c$ is a number to multiply the number of models by, and $w$ is a number to add to the weight of each model (both numbers derived from variables that have been lost in the propagation process). The function Prop is defined for 3SAT formulae since it is used in the #3SAT algorithm as well; naturally, only the 2SAT parts will be used in the algorithms of this chapter.

**Algorithm 79.** Prop($F, c, w$):

Initialise: $c := 1, w := 0$

While any of the following applies, apply the first applicable rule.

1. If $F$ contains an empty clause, then return ($F, 0, 0$).

2. If there is a clause ($1 \lor \ldots$), then remove this clause from $F$. For each variable $a$ that hereby gets removed, do as follows:
   
   (a) If $w(a) = w(\bar{a})$, then $c := c \cdot (c(a) + c(\bar{a}))$ and $w := w + w(a)$.
   
   (b) If $w(a) > w(\bar{a})$, then $c := c \cdot c(a)$ and $w := w + w(a)$.
   
   (c) If $w(a) < w(\bar{a})$, then $c := c \cdot c(\bar{a})$ and $w := w + w(\bar{a})$.

3. If there is a clause ($0 \lor \ldots$), then remove 0 from this clause.

4. If there is a clause ($a$), then remove this clause and let $c := c \cdot c(a)$ and $w := w + w(a)$. If $a$ still appears in $F$, then let $F := F\{a = 1\}$.

5. If there are two clauses ($x \lor y \lor z$), ($x \lor y$), then remove the first clause. If the variable $z$ hereby gets removed, do as in rule 2.

When no rule applies, return ($F, c, w$).

Algorithm ends.

In addition, let the phrase (recursively) branch on $x$ for a variable $x$ refer to the following steps:
1. Let \((F_t, c_t, w_t) = \text{Prop}(F\{x = 1\}, c, w)\) and \((F_f, c_f, w_f) = \text{Prop}(F\{x = 0\}, c, w)\).

2. Let \((c'_t, w'_t) = \text{C}(F_t, c, w)\) and \((c'_f, w'_f) = \text{C}(F_f, c, w)\).

3. Let \(W_{true} = w(x) + w_t + w'_t, W_{false} = w(\overline{x}) + w_f + w'_f, C_{true} = c(x) \cdot c_t \cdot c'_t,\) and \(C_{false} = c(\overline{x}) \cdot c_f \cdot c'_f.\) There are three cases.

   (a) If \(W_{true} = W_{false},\) then return \((C_{true} + C_{false}, W_{true}).\)

   (b) If \(W_{true} > W_{false},\) then return \((C_{true}, W_{true}).\)

   (c) Otherwise, return \((C_{false}, W_{false}).\)

The function Prop and the process of recursive branching hide the details of book-keeping in the algorithms.

**Lemma 80.** Let \((F', c, w) = \text{Prop}(F, c, w)\). If \(F'\) has \(c'\) max-weight models of weight \(w'\), then \(F\) has \(c \cdot c'\) max-weight models of weight \(w + w'\). Furthermore, the process of recursively branching on a variable produces correct results, assuming that the algorithm called is correct.

**Proof.** The correctness of Prop is quite natural, and the correctness of the branching process follows from this. \(\square\)

There is also a reduction that is used in both the \#2SAT and \#3SAT algorithms. Consider a formula \(F = (\overline{a} \lor \overline{b}) \land (\overline{a} \lor \overline{c}) \land (b \lor \overline{c}) \land F_1,\) where \(b\) and \(c\) do not appear in \(F_1\) (though \(a\) does). Assume that all positive literals have weight 1, and all negative literals weight 0. Then, we can observe the following:

- For any model for \(F_1\) of weight \(W\) with \(a = 1\), there is one model for \(F\) with weight \(W\) (since \(b = c = 0\) must be assigned).

- For any model for \(F_1\) of weight \(W\) with \(a = 0\), there are two models for \(F\) with weight \(W + 1\) (since \(b = 1\) or \(c = 1\) is possible). There is also one model for \(F\) with weight \(W\) (where \(b = c = 0\)), but since we are looking for max-weight models, we can ignore this model.
With the help of the cardinality vector, we can encode exactly this information into the data for a: add $+1$ to $w(\bar{a})$ and multiply $c(\bar{a})$ by 2, and an algorithm which counts max-weight models for $F_1$ will correctly count the number and weight of max-weight models for all of $F$; we can drop the three first clauses and only keep $F_1$. In the terminology of graph theory, $a$ is a cut vertex in the graph of the formula.

In precise terms, suppose that $F$ is a formula which can be partitioned into two formulae $F_1$ and $F_2$, each with more than one variable, such that $|\text{Var}(F_1) \cap \text{Var}(F_2)| = 1$, say $\text{Var}(F_1) \cap \text{Var}(F_2) = \{v\}$. Assume that every clause in $F$ belongs to either $F_1$ or $F_2$. We then say that multiplier reduction applies, and we can calculate $\#\text{2SAT}_w$ for $F$ as follows:

1. Calculate the number of max-weight models and the maximum weight of a model for $F_1$ when $v = 1$ and $v = 0$, as previously. Assume that there are $c_t$ max-weight models for $F_1$ of weight $w_t$ when $v = 1$ (not counting the weight or cardinality of $v$) and $c_f$ max-weight models for $F_1$ of weight $w_f$ when $v = 0$ (again not counting the weight or cardinality of $v$).

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

3. Return $C(F_2, c, w)$ with the modified vectors $c$ and $w$.

Note the similarity to the interface replacements of Chapter 5. The differences are that on the one hand, we do not need to create any new variables in a multiplier reduction, while on the other hand, multiplier reduction covers fewer cases than the interface replacements. The process described above is referred to as removing $F_1$ by multiplier reduction. In the algorithm, when multiplier reduction applies and we have two parts $F_1$ and $F_2$, we want to remove the lightest part (since we calculate the number of solutions for the removed part twice). In most cases, this choice is either obvious or not important, but for technical reasons we need to define it precisely: if $d(F) = 3$ and $l(F) \leq 2.4n(F)$, then the lightest part is the part which minimises $n_3(F_i)$, otherwise it is the part with a minimum number of variables.
**Lemma 81.** The process of applying multiplier reduction produces correct results, assuming that the algorithm called is correct.

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

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

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

We need one more definition, related to the condition for selecting a branching variable in the algorithm.

**Definition 82.** In a formula $F$ with average degree $l(F)/n(F) = k$, the associated average degree of a variable $x$ in $F$ is $\alpha(x)/\beta(x)$, where:

$$\alpha(x) = d(x) + |\{y \in N(x) \mid d(y) < k\}| \quad (7.1)$$

$$\beta(x) = 1 + \sum_{\{y \in N(x) \mid d(y) < k\}} \frac{1}{d(y)} \quad (7.2)$$

We will see in Lemma 87 that there always exists some variable with both degree and associated average degree at least $k$.

Now, we can provide the algorithm. Note that though the analysis is split into several parts, using different measures, these parts are only different ways of analysing this same algorithm.
Algorithm 83. \( C(F, c, w) \):

1. If \( F \) contains no clauses, then return \((1, 0)\). If \( F \) contains an empty clause, then return \((0, 0)\).

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 \((c_i, w_i) = C(F_i, c, w)\) for the connected components \( F_0, \ldots, F_j \).

3. If multiplier reduction applies, then apply it, removing the lightest part (as previously defined).

4. If \( d(F) \in \{3, 4\} \), then let \( x \) be a variable of maximum degree, secondarily maximising the associated average degree \( \alpha(x)/\beta(x) \).
   
   (a) If there exists a set of two heavy variables \( \{y, z\} \), \( y, z \not\in N[x] \), whose removal leaves \( F \) disconnected and leaves \( N(x) \) in a non-heaviest component, then recursively branch on \( y \).
   
   (b) Otherwise, recursively branch on \( x \).

5. Let \( x \) be a variable of maximum degree, which if possible does not have only neighbours of degree \( d(x) \), and recursively branch on \( x \).

Algorithm ends.

Lemma 84. \( C(F, c, w) = \#2\text{SAT}_w(F, c, w) \).

Proof. The correctness of each step follows from previous lemmas, and the completeness of the algorithm is obvious. \( \square \)

### 7.2 Maximum Degree 4

In this section, we will give upper bounds for the running time of the algorithm in cases where \( d(F) \leq 4 \). The bounds of this section are given using the method of analysis by average degree, as defined in Section 3.5.1. We begin with an observation for the case \( d(F) = 2 \).
Lemma 85. The algorithm C applied to a formula $F$ with $d(F) \leq 2$ runs in polynomial time.

Proof. Let $F'$ be the maximally reduced version of $F$. Any variable in $F$ with only one occurrence is taken care of by case 2 or 3, without increasing the degree of any other variable, so if $F'$ is non-empty then it will be 2-regular. Hence, the graph of $F'$ is a cycle. Removing any one variable from $F'$ leaves a formula whose graph is a path, which will be entirely cleaned up using multiplier reduction, leaving a formula of constant size.

Before we give the rest of the bounds, perhaps a few words on the reasons for the choice of method are in order. While it may seem that including information about the number of variables of each degree should provide enough information to analyse the behaviour of the algorithm, the following lemma shows that such an analysis would produce an inferior result, compared to that which we give.

Lemma 86. A (non-compound) weight-based analysis of the 3-regular case of C, with weights based on the degree of a variable, can give no better bound than $O^*(1.1892^n)$.

Proof. Let $x$ be a 3-variable with all variables in $N(x)$ light, and assume that case 4b of C is used. In both branches, all of $N[x]$ is removed (either by Prop, or by multiplier reduction). Now, there can be no internal edges in $N(x)$, since then multiplier reduction would apply, so in addition three edges leaving $N(x)$ are removed. Assume that these edges hit different 3-variables. If $f(F) = \sum_i w_i n_i$, then the reduction in $f$ is as follows:

$$\Delta f = w_3 + 3w_2 + 3(w_3 - w_2) = 4w_3$$

In other words, this case leaves a branching with a reduction of exactly $4w_3$ in both branches, and the branching number for such a branching will always be $2^{1/4w_3}$. If $n = n_3$, then the bound can be no better than $O^* \left(2^{f(F)/4w_3} \right) = O^* \left(2^{n/4} \right)$. 

\qed
7. Counting 2SAT

The branching case appearing when branching on a 3-variable with only light neighbours turns out to make standard weight-based analysis inappropriate. Indeed, no better result than $O^*(2^{n_3/4})$ seems possible through any method when this case applies. On the other hand, such a case will only be used by the algorithm when $d(F) = 3$ and every heavy variable has only light neighbours, implying that $2n_2 \geq 3n_3$, so $n \geq 2.5n_3$ and $O^*(2^{n_3/4}) \subseteq O^*(2^{n/10})$, using Lemma 85 when $n_3 = 0$ and assuming that no harder case appears under these conditions (which turns out to be true), while better branchings are guaranteed when $n_2 < 1.5n_3$ (implying a switch in behaviour at average degree 2.4). These circumstances—polynomial time for degree 2, a relatively poor-quality branching for average degree 2.4, and progressively better branchings for higher average degrees—make a compound measure switching behaviour based on the average degree a good tool. As Lemma 88 will show, using this method we can prove a stronger bound of $O^*(1.1499^n)$ for the case $d(F) = 3$. When $d(F) > 4$, however, we use a non-compound weight-based measure as described in the analysis, since no useful progression of easier cases is apparent—the worst cases for both degree 5 and 6 include cases that can always appear, no matter what the average degree.

We will soon commence with proving the bounds, but first we need a lemma making a connection between the average degree and the guaranteed branching cases.

**Lemma 87.** Let $F$ be a non-empty formula such that $l(F)/n(F) = k$, and recall that the associated average degree of a variable $x$ is $\alpha(x)/\beta(x)$ where:

\[
\begin{align*}
\alpha(x) &= d(x) + |\{y \in N(x) \mid d(y) < k\}| \quad (7.3) \\
\beta(x) &= 1 + \sum_{\{y \in N(x) \mid d(y) < k\}} 1/d(y) \quad (7.4)
\end{align*}
\]

Then, there exists some variable $x \in Var(F)$ with $d(x) \geq k$ with associated average degree at least $k$. 
7.2. Maximum Degree 4

Proof. Consider the following sums:

\[
A = \sum_{\{x \in \text{Var}(F) \mid d(x) \geq k\}} \alpha(x) \quad (7.5)
\]

\[
B = \sum_{\{x \in \text{Var}(F) \mid d(x) \geq k\}} \beta(x) \quad (7.6)
\]

We may view every variable \(x\) with \(d(x) \geq k\) as contributing exactly \(d(x)\) to \(A\) and 1 to \(B\), and each variable \(y\) with \(d(y) < k\), \(i\) to \(A\) and \(i/d(y)\) to \(B\), for some integer \(i \leq d(y)\) (which can be viewed as contributing a fraction \(i/d(y)\) of the full contributions of a variable of degree \(d(y)\)). 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:

\[
A = \sum_i in'_i(F) = m(F) - \sum_{i<k} i(n_i(F) - n'_i(F)) \quad (7.7)
\]

\[
B = \sum_i n'_i(F) = n(F) - \sum_{i<k} (n_i(F) - n'_i(F)) \quad (7.8)
\]

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

\[
A \geq k \cdot B \quad (7.9)
\]

The set \(\{x \in \text{Var}(F) \mid d(x) \geq k\}\) is clearly not empty. Hence, if we had \(\alpha(x) < k\beta(x)\) for all \(x\) with \(d(x) \geq k\), then inequality (7.9) could not hold. Therefore there is an \(x\) with \(d(x) \geq k\) such that \(\alpha(x) \geq k\beta(x)\). \(\square\)

We will now give the bound for the case \(d(F) = 3\). For reference, the possible neighbourhoods of a heavy variable, with their respective average degree guarantees, are given in Table 7.1. The measure is based on the attributes \(l(F)\) and \(n(F)\), rather than \(n_2(F)\) and \(n_3(F)\), since they are equivalent when there are only two attributes, and the former is somewhat easier to work with.

Lemma 88. For a formula \(F\) with \(d(F) \leq 3\), algorithm C runs in \(O^*(1.1499^n)\) time.
Table 7.1: Possible neighbourhoods with associated average degree when branching on a 3-variable.

<table>
<thead>
<tr>
<th>Neighbours</th>
<th>Highest average degree</th>
<th>Branching (case 4b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 2, 2)</td>
<td>6/2.5 = 2.4</td>
<td>(\tau(12w_l + 4w_n, 12w_l + 4w_n))</td>
</tr>
<tr>
<td>(2, 2, 3)</td>
<td>5/2 = 2.5</td>
<td>(\tau(10w_l + 3w_n, 18w_l + 6w_n))</td>
</tr>
<tr>
<td>(2, 3, 3)</td>
<td>4/1.5 ≈ 2.67</td>
<td>(\tau(8w_l + 2w_n, 16w_l + 5w_n))</td>
</tr>
<tr>
<td>(3, 3, 3)</td>
<td>3/1 = 3</td>
<td>(\tau(6w_l + w_n, 16w_l + 4w_n))</td>
</tr>
</tbody>
</table>

Table 7.2: Component measures \(w_l(F) + w_n(F)\) for maximum degree 3

<table>
<thead>
<tr>
<th>Section</th>
<th>(w_l)</th>
<th>(w_n)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–2.4</td>
<td>0.25</td>
<td>−0.5</td>
<td>(O^* (2^{0.1495n}) \subset O^* (1.1092^n))</td>
</tr>
<tr>
<td>2.4–2 + 2/3</td>
<td>0.185373</td>
<td>−0.344895</td>
<td>(O^* (2^{0.1495n}) \subset O^* (1.1092^n))</td>
</tr>
<tr>
<td>2 + 2/3–3</td>
<td>0.155985</td>
<td>−0.266527</td>
<td>(O^* (2^{0.2015n}) \subset O^* (1.1499^n))</td>
</tr>
</tbody>
</table>

Proof. The components of the compound measure for this case are on the form \(f_a(l, n) = w_l l + w_n n\), with the parameters of the measures given in Table 7.2. It may seem strange that \(w_n < 0\) for these components, but this can be translated into the form \(\sum_i w_i n_i\) with \(w_i = iw_l + w_n\), in which case \(w_i \geq 0\) for every \(i \geq 2\). Since every reduction step in Prop either removes variables or decreases the degrees of variables, we see that every reduction leaves \(f_a(l, n)\) non-increasing. Also, let \(F'\) be the maximally reduced version of \(F\); we have \(f_a(F) \geq f_a(F')\) by this observation, and \(f_a(F') \geq f(F')\) by condition (3.4) from Section 3.5. For cases 2 and 3 of the algorithm, note that since \(f_a\) is linear, \(f(F) = f_a(F) = \sum_i f_a(F_i) \geq \sum_i f(F_i)\). For both cases, the time spent on all but the heaviest component is dominated by the time spent on the heaviest component. We see that \(f(F)\) is a well-behaved measure for the algorithm. Also, when estimating \(\Delta f\), this means that our underestimations are safe unless the formula we compare against contains a singleton (since \(w_1 < 0\) for some sections of \(f\)). Specifically, \(\Delta f\) can be described as \(w_2\) for every
removed 2-variable, \( w_3 \) for every removed 3-variable, and \( w_l \) for every variable that has had its degree reduced from 3 to 2.

Next, consider case 4a. We can see that in both branches we will have removed all of \( N[x] \) plus the variable \( y \), and at least two heavy variables will have been reduced to light variables (the easiest way to see this is that the variables we are sure to remove form a connected subgraph of the formula, which must therefore connect to at least two variables in the rest of \( F \)). Both branches get a reduction of at least \((S(x) + 5)w_l + 5w_n\), which will compare favourably to the results of using case 4b, and will never result in a worse branching.

Now, for the worst cases of the algorithm. The branchings are given in terms of generic weights \( w_l \) and \( w_n \), since they do not depend upon the particular measure associated with the current section. For reference, these branchings are listed in Table 7.1 as well.

1. If \( x \) has no heavy neighbours, then all members of \( N(x) \) are removed in both branches (since they are reduced to singletons when not assigned), and unless case 4a applies, at least three further edges are removed, in a way so that at least three heavy variables have their degrees decreased. In total, \( \Delta f \geq 12w_l + 4w_n \).

2. If \( x \) has exactly one heavy neighbour \( y \), then we use that \( y \) was not chosen as a branching variable to derive that \( y \) has no other heavy neighbour. The light neighbours of \( x \) disappear in both branches, and in one branch, \( y \) is assigned as well. If there is a path of only light variables from \( x \) to \( y \) (for instance, if \( x \) and \( y \) have a common neighbour), then case 4a applies. Otherwise, in the branch where \( y \) is not removed, \( \Delta_1f \geq w_3 + 2w_2 + 3w_l = 10w_l + 3w_n \), and in the branch where \( y \) is removed, in total at least four light variables are removed, so \( \Delta_2f \geq 2w_3 + 4w_2 + 4w_l = 18w_l + 6w_n \) (by the parity of \( l(F) \); i.e. \( l(F) \) must be even, as all clauses have length 2).

3. If \( x \) has exactly two heavy neighbours \( y, z \), then \( \Delta_1f \geq w_3 + w_2 + 3w_l = 8w_l + 2w_n \), and \( \Delta_2f \geq 3w_3 + w_2 + 3w_l = 14w_l + 4w_n \). 

7. Counting 2SAT

Table 7.3: Branching cases for $d(F) = 3$ analysis

<table>
<thead>
<tr>
<th>Section</th>
<th>Case</th>
<th>Time (4a)</th>
<th>Time (4b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–2.4</td>
<td>(2, 2, 2)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2–2.4</td>
<td>(2, 2, 3)</td>
<td>1.741102</td>
<td>1.754878</td>
</tr>
<tr>
<td>2–2.4</td>
<td>(2, 3, 3)</td>
<td>1.587402</td>
<td>1.754878</td>
</tr>
<tr>
<td>2–2.4</td>
<td>(3, 3, 3)</td>
<td>1.485995</td>
<td>1.618034</td>
</tr>
<tr>
<td>2.4–2 + 2/3</td>
<td>(2, 2, 3)</td>
<td>1.927676</td>
<td>1.964799</td>
</tr>
<tr>
<td>2.4–2 + 2/3</td>
<td>(2, 3, 3)</td>
<td>1.747730</td>
<td>2</td>
</tr>
<tr>
<td>2.4–2 + 2/3</td>
<td>(3, 3, 3)</td>
<td>1.625448</td>
<td>1.850966</td>
</tr>
<tr>
<td>2 + 2/3–3</td>
<td>(3, 3, 3)</td>
<td>1.691247</td>
<td>2</td>
</tr>
</tbody>
</table>

4. Finally, if $x$ has only heavy neighbours, then $\Delta_1 f \geq w_3 + 3w_l = 6w_l + w_n$ and $\Delta_2 f \geq 4w_3 + 4w_l = 16w_l + 4w_n$.

Now that both the measures and the branchings have been given, all that remains is to verify the claim that every branching number is at most 2; see Table 7.3 for this. (The “case” column describes the degrees of the neighbours, as in Table 7.1.) When $l(F)/n(F) \leq 2.4$, every case is possible, and the worst case is when all neighbours are light, with an associated maximum average degree of 2.4. When $l(F)/n(F) > 2.4$, $w_l$ is decreased and $w_n$ is increased until some new case gets a branching number of 2; in this case, the case where one neighbour is light is the most difficult of the remaining cases, with an associated maximum average degree of $2 + 2/3$. When $l(F)/n(F) > 2 + 2/3$, finally, only the last case is possible. We see that the time is indeed in $O^*(2^{f_3(F)})$ for the $f_3(F)$ given in Table 7.2, and the total worst time is $O^*(1.1499^n)$, as given in the table.

Now, we present the analysis of the case when $d(F) = 4$. For this case, the multiple attributes-version of analysis by average degree (see Section 3.5.2) is used, with component measures $\sum_i w_in_i(F)$, and correspondingly one weight for each variable degree for each section of the compound analysis. We use $\Delta w_i = w_i - w_{i-1}$ to simplify expressions of branchings. However, as explained in Section 3.5, because the
### 7.2. Maximum Degree 4

Overall compound measure must be continuous, our freedom in choosing these weights is quite limited. Once the bottom-most component measure has been decided, each following component measure can be chosen with exactly one degree of freedom (which can be described as the amount of pivoting that is performed around the boundary point), and if we add that the worst-case branching number in each section should be the same, then each following component measure is uniquely determined by the first. In this case, we have the added restriction that for any combination of values for \( n_2 \) and \( n_3 \) when \( n_4 = 0 \), the time bound we get from our component measure must be no higher than that which we get from our previous analysis of the \( d(F) = 3 \) case. Luckily, according to the conditions on the compound measure (specifically, condition (3.4), as given in Section 3.5), it is enough that the bound from our \( d(F) = 4 \) compound measure is at least as big as the bound from one component measure of the \( d(F) = 3 \) compound measure (we chose the top-most component measure, since this has the best bound).

The weights of the measure are given in Table 7.4. These weights were calculated automatically according to the approach described in Section 3.5.1, with resulting pivot points at average degrees 3, 3.2, 3.5, and 3.75 (the amount of pivot at the other potential pivot points was found to be zero in an optimal solution). The component measure for section 2–3 coincides with the top-most component measure for \( d(F) = 3 \): \( 0.155985 l(F) - 0.266527 n(F) \) results in \( w_2 = 2w_l - w_n = 0.045443 \) and \( w_3 = 3w_l - w_n = 0.201428 \). The automatic weight

<table>
<thead>
<tr>
<th>Section</th>
<th>( w_2 )</th>
<th>( w_3 )</th>
<th>( w_4 )</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2–3</td>
<td>0.045443</td>
<td>0.201428</td>
<td>0.324788</td>
<td>( O^* (1.1499^n) )</td>
</tr>
<tr>
<td>3–3.2</td>
<td>0.084777</td>
<td>0.201428</td>
<td>0.285454</td>
<td>( O^* (1.1634^n) )</td>
</tr>
<tr>
<td>3.2–3.5</td>
<td>0.092882</td>
<td>0.202779</td>
<td>0.280051</td>
<td>( O^* (1.1822^n) )</td>
</tr>
<tr>
<td>3.5–3.75</td>
<td>0.097593</td>
<td>0.204349</td>
<td>0.278481</td>
<td>( O^* (1.1975^n) )</td>
</tr>
<tr>
<td>3.75–4</td>
<td>0.107950</td>
<td>0.208788</td>
<td>0.277001</td>
<td>( O^* (1.2117^n) )</td>
</tr>
</tbody>
</table>

Table 7.4: Component measures \( \sum_i w_i n_i(F) \) for maximum degree 4
calculation also guarantees that the choice of weights and pivoting strategy is optimal. The bound achieved for $d(F) = 4$ is $O^* (1.2117^n)$.

First, we show that the weights agree with the definition of a compound measure.

**Lemma 89.** The weights of Table 7.4 form a correct compound measure.

*Proof.* We will verify that the two neighbouring components at every pivot point meet. The direction of pivoting is correct, as $w_4$ decreases.

- At average degree $3.0$, we can split the contribution to the weight into that from 3-variables, which is identical for $f_1$ and $f_2$ since $w_3$ does not change, and the contribution $0.5w_2 + 0.5w_4 = 0.185115$ from a mixture of 2- and 4-variables, which is also identical for $f_1$ and $f_2$.

- At average degree $3.2$, we can split the contribution to the weight into $0.4w_2 + 0.6w_4 = 0.205183$, identical for $f_2$ and $f_3$, and $0.8w_3 + 0.2w_4 = 0.218233$, also identical for $f_2$ and $f_3$.

- At average degree $3.5$, we can split the contribution to the weight into $0.25w_2 + 0.75w_4 = 0.233259$ for both $f_3$ and $f_4$, and $0.5w_3 + 0.5w_4 = 0.241415$ for both $f_3$ and $f_4$.

- At average degree $3.75$, we can split the contribution to the weight into $0.125w_2 + 0.875w_4 = 0.255870$ for both $f_4$ and $f_5$, and $0.25w_3 + 0.75w_4 = 0.259948$ for both $f_4$ and $f_5$, and this is the final pivoting point.

Now that this is established, we provide the proof of the upper bound.

**Lemma 90.** For a formula $F$ with $d(F) = 4$, $C(F)$ runs in time $O^* (1.2117^n)$. 

### 7.2. Maximum Degree 4

<table>
<thead>
<tr>
<th>Degrees of neighbours</th>
<th>Max average degree</th>
<th>Branching (case 4b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 2, 2, 2)</td>
<td>3</td>
<td>$\tau(5w_4 - 4w_3 + 4w_2, 5w_4 - 4w_3 + 4w_2)$</td>
</tr>
<tr>
<td>(2, 2, 2, 3)</td>
<td>3</td>
<td>$\tau(4w_4 - 2w_3 + 2w_2, 4w_4 - 2w_3 + 3w_2)$</td>
</tr>
<tr>
<td>(2, 2, 2, 4)</td>
<td>3</td>
<td>$\tau(5w_4 - 4w_3 + 3w_2, 6w_4 - 4w_3 + 3w_2)$</td>
</tr>
<tr>
<td>(2, 2, 3, 3)</td>
<td>3</td>
<td>$\tau(3w_4, 5w_4 - 2w_3 + 2w_2)$</td>
</tr>
<tr>
<td>(2, 2, 3, 4)</td>
<td>3</td>
<td>$\tau(4w_4 - 2w_3 + w_2, 5w_4 - 2w_3 + 2w_2)$</td>
</tr>
<tr>
<td>(2, 2, 4, 4)</td>
<td>3</td>
<td>$\tau(5w_4 - 4w_3 + 2w_2, 7w_4 - 4w_3 + 2w_2)$</td>
</tr>
<tr>
<td>(2, 3, 3, 3)</td>
<td>$16/5 = 3.2$</td>
<td>$\tau(2w_4 + 2w_3 - 2w_2, 4w_4 + w_2)$</td>
</tr>
<tr>
<td>(2, 3, 3, 4)</td>
<td>$42/13 \approx 3.23$</td>
<td>$\tau(3w_4 - w_2, 6w_4 - 2w_3 + w_2)$</td>
</tr>
<tr>
<td>(2, 3, 4, 4)</td>
<td>$36/11 \approx 3.27$</td>
<td>$\tau(4w_4 - 2w_3, 6w_4 - 2w_3 + w_2)$</td>
</tr>
<tr>
<td>(2, 4, 4, 4)</td>
<td>$10/3 \approx 3.33$</td>
<td>$\tau(5w_4 - 4w_3 + w_2, 8w_4 - 4w_3 + 2w_2)$</td>
</tr>
<tr>
<td>(3, 3, 3, 3)</td>
<td>$24/7 \approx 3.43$</td>
<td>$\tau(w_4 + 4w_3 - 4w_2, 5w_4)$</td>
</tr>
<tr>
<td>(3, 3, 3, 4)</td>
<td>$7/2 = 3.5$</td>
<td>$\tau(2w_4 + 2w_3 - 3w_2, 5w_4)$</td>
</tr>
<tr>
<td>(3, 3, 4, 4)</td>
<td>$18/5 = 3.6$</td>
<td>$\tau(3w_4 - 2w_2, 7w_4 - 2w_3)$</td>
</tr>
<tr>
<td>(3, 4, 4, 4)</td>
<td>$15/4 = 3.75$</td>
<td>$\tau(4w_4 - 2w_3 - w_2, 7w_4 - 2w_3)$</td>
</tr>
<tr>
<td>(4, 4, 4, 4)</td>
<td>4</td>
<td>$\tau(5w_4 - 4w_3, 9w_4 - 4w_3)$</td>
</tr>
</tbody>
</table>

Table 7.5: Possible neighbourhoods with associated average degree when branching on a 4-variable.
Proof. We refer again to Table 7.4 for a definition of the weights in the compound measure. The measure is clearly well-behaved. The branching depends on the neighbourhood of the variable that is chosen; see Table 7.5 for a list of neighbourhoods, with corresponding highest average degree and worst-case branching. We will prove that these are the worst-case branchings shortly, but first we consider case 4a: if case 4a is used, then $N[x]$ and $y$ are removed in both branches, and at least two variables decrease their degree, which can be adjusted for the parity of $l(F)$. In the heavy branch of the maximally unbalanced branching, $N[x]$ is removed and at least three variables get their degrees decreased, likewise adjusted for the parity of $l(F)$.

We see that in case 4a, both branches will be at least as heavy as the heaviest possible branch of case 4b. Thus, we only consider case 4b in the following.

When removing only the variable $x$ and repeatedly applying cases 1 and 2 of the algorithm, if any variable gets its degree reduced to 0\(^1\) or ends up in a non-biggest connected component (even if this happens after subsequent applications of case 2), then $x$ is a cut-vertex and multiplier reduction applies to $F$. Also, obviously, if any variable gets its degree reduced to 1, then multiplier reduction applies and one more reduction of the degree of some variable occurs. Thus, when $x$ has $k$ light neighbours and only $x$ is removed in one branch, the total reduction is at least $w_4 + kw_2 + k\Delta w_4$ plus the reductions in degree of the other neighbours of $x$, for the light branch of the maximally unbalanced branching. When some other variable is assigned, this does not apply, though we do know that in total, there are at least three variables outside of $N[x]$ that have links to variables in $N(x)$.

The issue we need to handle is when the neighbours of $x$ are not all removed in the same branch. When these neighbours are heavy, we get easier branchings by the balancing property of $\tau$, but when some neighbours are light (and thus already removed in both branches), then the matter can get more complicated. In such a case, there will be some branch where only one heavy neighbour $y$ is assigned, and

\(^1\)By “reduced to 0” we mean that all neighbours of the variable are removed, and we do not include when a variable is removed by multiplier reduction.
in the other branch all variables of $N[x]$ except $y$ are removed. If in the second branch $y$ is also removed, then the case cannot be harder than the maximally unbalanced one, so there must be at least two neighbours of $y$ that are not removed in the second branch, and these neighbours get their degrees decreased in the first branch.

- When there are at least 3 light variables in $N(x)$, the only possible distribution of signs is the maximally unbalanced one.

- If there are 2 light variables in $N(x)$, then the observations about $y$ hold for both heavy variables, and as a result (if the degrees of the heavy variables are $a$ and $b$) the reductions in the branches will be at least $w_1 + 2w_2 + w_a + \Delta w_b + 3\Delta w_4$ respectively $w_4 + 2w_2 + w_b + \Delta w_a + 3\Delta w_4$ (where the third reduction of $\Delta w_4$ is due to one of the light variables having a neighbour that has not been assigned, by $a, b \leq 4$). Compared to the unbalanced branching, the first branch reduces $f(F)$ by $w_{a-1} + \Delta w_4$ more than the light branch of the unbalanced branching, and the second branch reduces $f(F)$ by up to $w_{a-1} + \Delta w_4$ less. Clearly, the balanced branching will simply not be harder than the unbalanced branching.

- If there is exactly one light variable in $N(x)$, then let $d(y) = a$.

  - If $a = 3$, then $a$ is not affected by the light variable, and on the light side of the branching, we reduce the degrees of all other heavy neighbours of $x$, plus three more reductions: two because of neighbours of $y$, and a third because of the light variable. In this case, the reduction in the light branch is at least $w_2 + 2\Delta w_4$ higher than in the light branch of the unbalanced branching. The reduction in the heavy branch is at most $w_2 + 2\Delta w_4$ lighter than in the heavy branch of the unbalanced branching: $x$ plus three neighbours of $x$ are removed; if $S(x)$ is even, then the sum of degrees in this part is odd, and in addition to the connection between $x$ and $y$ there exist at least two further
connections away from the assigned variables. We see that this does not introduce any harder cases either.

– If \( a = 4 \), then \( a \) can be a neighbour of the light variable, and we may at worst reduce \( f(F) \) by \( w_3 + \Delta w_4 \) more on the light branch, and by \( w_3 + 2 \Delta w_4 \) less on the heavy branch, compared to the unbalanced branching (if \( S(x) \) is even and only one link is removed in the heavy branch, then three links will be removed in the light branch, which is easier). While we will not show as a general result that this is easier than the unbalanced branchings, we will show for each component measure that the cases created in this way are not the hardest cases.

For the section 2–3, a quick look at the numbers suffices to show this. The “light” side of the more balanced branching when \( d(y) = 3 \)
reduces $f(F)$ by at least $w_4 + w_2 + w_3 + 5\Delta w_4 \approx 1.1885$ and the “heavy” side by at least $w_4 + w_2 + 2w_3 + \Delta w_3 + \Delta w_4 \approx 1.0524$. When $d(y) = 4$, the reductions are at least $w_4 + w_2 + w_4 + 4\Delta w_4 \approx 1.1885$ and $w_4 + w_2 + 2w_3 + 2\Delta w_4 \approx 1.0198$. For the sections 3–3.2 and 3.2–3.5, we need to handle the cases one by one.

- Case (2,3,3,3): Only $d(y) = 3$ possible.
- Case (2,3,3,4): When $d(y) = 4$, we split into cases: if the light variable is a neighbour of $y$, then we get $\tau(w_4 + w_2 + w_4 + 2\Delta w_3 + 2\Delta w_4, w_4 + w_2 + 2w_3 + (w_4 - w_2) + 2\Delta w_4) < 1.8791$ for section 3–3.2 and $1.9078$ for section 3.2–3.5; otherwise, $\tau(w_4 + w_2 + w_4 + 2\Delta w_3 + 3\Delta w_4, w_4 + w_2 + 2w_3 + 2\Delta w_4) < 1.9502$ for section 3–3.2 and $1.9779$ for section 3.2–3.5.
- Case (2,3,4,4): $\tau(w_4 + w_2 + w_4 + \Delta w_3 + 3\Delta w_4, w_4 + w_2 + w_3 + w_4 + 2\Delta w_4) < 1.9668$ for section 3–3.2 and $1.9966$ for section 3.2–3.5, using $d(y) = 4$.
- Case (2,4,4,4): $\tau(w_4 + w_2 + w_4 + 4\Delta w_4, w_4 + w_2 + 2w_4 + 2\Delta w_4) < 1.9359$ for section 3–3.2 and $1.9685$ for section 3.2–3.5.

When the average degree is more than 3.5, no case with a light neighbour can occur, which leaves only the unbalanced cases as potentially difficult.

Table 7.6 contains the branching numbers for applying the maximally unbalanced version of case 4b in each combination of section and neighbourhood. As can be seen from the table, each branching number is at most 2. The total worst-case time for the $d(F) = 4$ case, as stated, is $O^*(2^{w_4n})$ for the final value of $w_4 = 0.277001$, or $O^*(1.2117^n)$. □

7.3 General Case

With $d(F) > 4$, the effects of a changing average degree seem to be less important than the number of variables removed. The analysis is performed in terms of a standard weight-based measure $f(F) = $
Table 7.7: Weights for $d(F) > 4$ analysis

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
$w_2$ & $w_3$ & $w_4$ & $w_5$ & $w_6$ \\
0.115507 & 0.208788 & 0.277001 & 0.301245 & 0.307612 \\
\hline
\end{tabular}
\caption{Weights for $d(F) > 4$ analysis}
\end{table}

$\sum_i w_i n_i(F)$, whose weights are given in Table 7.7. Note that while the values of $w_3$ and $w_4$ are the same as in the topmost measure for the $d(F) = 4$ analysis, the value of $w_2$ is increased to get a better worst-case branching number. This inequality is no problem, since the degree of a variable never increases by the application of a reduction: once $d(F) < 5$, the case $d(F) > 4$ does not appear in any subinstance.

We will see that the hard cases are one case with a smallest-possible neighbourhood ($d(x) = 5$ and $N(x)$ is 2-regular), and the two cases with biggest-possible neighbourhoods (for $d(x) = 5$ and $d(x) = 6$). Since we have hard cases with a maximum value of the average degree, this suggests that a compound measure is not the right tool for this analysis.

**Lemma 91.** Using $f(F) = \sum_i w_i n_i(F)$ with the weights given in Table 7.7, the running time of C for a formula $F$ with $d(F) \leq 6$ is in $O^*(2^{f(F)})$.

**Proof.** If $d(F) < 5$, then see Section 7.2 (note that the weights $w_2, \ldots, w_4$ give a bound that is consistent with that for section 3.75–4, which is in turn a valid bound for all cases with $d(F) \leq 4$). As before, the application of case 4a guarantees a reduction in both branches that is at least as high as the reduction in the heavy branch of the maximally unbalanced branching, and all cases with a branching number of 2 appear in case 4b with the maximally unbalanced branching. The branching numbers for case 4b with the maximally unbalanced branching are given in Table 7.8 for branchings with $d(x) = 5$ and Tables 7.9–7.12 for branchings with $d(x) = 6$; the cases with a branching number of 2 are when $d(x) = 2$ and the neighbourhood is 2-regular, when $d(x) = 5$ and the neighbourhood is 5-regular except for one neighbour (which has degree 4), and when $d(x) = 6$ and the neighbourhood is 6-regular except for one neighbour (which has degree 5).
The cases of $k$-regular neighbourhood with $d(x) = k$ are avoided as far as possible in case 5 of the algorithm, and as a result these cases happen at most once each in every path through the branching tree: they only apply if the $k$-variables form a regular connected component, and since no reduction creates a new occurrence of any variable in the formula, any $k$-regular connected component that appears in some subsequent subcase of some $k$-regular formula $F$ must occur as a subformula in $F$, which is impossible. Since these cases occur at most once in every path of the tree, they contribute only to the polynomial part of the running time.

It remains to show that no case with a more balanced branching has a higher branching number than 2. As in the proof of Lemma 90, the issues are the neighbourhoods with light variables, and as before, if there is any variable $y$ which is the only assigned heavy neighbour of $x$ in some branch, then $y$ has at least two neighbours external to $N[x]$, unless $y$ is removed in both branches (which leads to an easier case). Let $d(y) = a$ and $d(x) = b$.

- If $y$ has one neighbour among the light variables, then we increase the reduction on the light side by at least $w_{a-1} + \Delta w_b$, while the reduction on the heavy side decreases by no more than $w_{a-2} + 3\Delta w_b$. Now, with $a \leq b$, $\Delta w_{a-1} > 2\Delta w_b$, so we add at least as much on the light side as we remove on the heavy side, and do not create a harder case.

- If $y$ has at least two neighbours among the light variables, then we may increase the reduction on the light side by no more than $w_{a-1}$, but we decrease the reduction on the heavy side by at most $w_{a-3} + 3\Delta w_b$, and $w_{a-1} - w_{a-3} > 3\Delta w_b$, so the same reasoning applies.

- If there is some variable that gets its degree reduced to 0, for instance if $y$ and some light variables are its only neighbours, then we do not get a harder case: $w_i/i > \Delta w_b$ for every $i$, so this case is easier than when every link goes to a unique $b$-variable. Otherwise, we will not get a harder case by moving $y$ from the heavy to the light side.
Finally, we have to deal with the cases where at least two heavy neighbours are assigned in each branch.

If \( d(x) = 5 \), then two heavy variables are assigned in each branch. If the light variable does not have a neighbour among these, then the case cannot be harder than the unbalanced case. Let the heavy neighbours of \( x \) have degrees \( a, b, c, \) and \( d \), and assume that \( a \) and \( b \) are assigned in the same branch, \( a \leq b, c \leq d \), and that the neighbour of the light variable is \( a \) or \( b \). Then the reduction in the branch where \( a \) and \( b \) are assigned is at least

\[
w_5 + w_2 + w_a + w_b + \Delta w_c + \Delta w_d + (a + b - 5)\Delta w_5 \geq w_5 + w_2 + w_a + w_b + (a + b - 3)\Delta w_5, \tag{7.10}
\]

and the reduction in the other branch is at least

\[
w_5 + w_2 + w_c + w_d + \Delta w_a + (w_b - w_{b-2}) + (c + d - 4)\Delta w_5 \geq w_5 + w_2 + 2w_3 + \Delta w_a + (w_b - w_{b-2}) + 2\Delta w_5. \tag{7.11}
\]

This reduces in different ways depending on \( a \) and \( b \).

1. If \( b = 5 \), then (7.10) is no lower than \( w_5 + w_2 + w_3 + w_5 + 5\Delta w_5 > 1.0480 \), while (7.11) is no lower than \( w_5 + w_2 + 2w_3 + 4\Delta w_5 + \Delta w_4 > 0.9995 \), and \( \tau(1.0480,0.9995) < 1.9684 \).

2. If \( b = 4 \), then (7.10) is no lower than \( w_5 + w_2 + w_3 + w_4 + 4\Delta w_5 > 0.9995 \), while (7.11) is no lower than \( w_5 + w_2 + 2w_3 + 3\Delta w_5 + (w_4 - w_2) > 1.0685 \), and \( \tau(0.9995,1.0685) < 1.9555 \).

3. If \( a = b = 3 \), then (7.10) is at least \( w_5 + w_2 + 2w_3 + 3\Delta w_5 > 0.9070 \), and (7.11) is at least \( w_5 + w_2 + 3w_3 + \Delta w_3 + 2\Delta w_5 > 1.1848 \), and \( \tau(0.9070,1.1848) < 1.9481 \).

This covers all cases when \( d(x) = 5 \). When \( d(x) = 6 \), we again divide into cases.

1. If there are two light neighbours, then the reduction in each branch is at least \( w_6 + 2w_2 + 2w_3 + 3\Delta w_6 > 0.9753 \).
7.3. General Case

- If there is any neighbour of degree at least 4, then the reduction in one branch is at least \( w_6 + 2w_2 + w_3 + w_4 + 3\Delta w_6 > 1.0435 \), and \( \tau(0.9753, 1.0435) < 1.9877 \).
- Otherwise, we get a reduction of at least \( w_6 + 2w_2 + 2w_3 + 2\Delta w_3 + \Delta w_6 > 1.1491 \).

2. If there is one light neighbour, then the heavy variables will be divided so that two are assigned in one branch and three in the other.

- If one of the two variables has degree at least 4, then our branching is dominated by \( \tau(w_6 + w_2 + w_3 + w_4 + 5\Delta w_6, w_6 + w_2 + 3w_3 + 3\Delta w_6) < 1.9956 \).
- Otherwise, it is dominated by \( \tau(w_6 + w_2 + 2w_3 + 4\Delta w_6, w_6 + w_2 + 3w_3 + 2\Delta w_3 + \Delta w_6) < 1.9443 \).

We see that all cases have a branching number of at most 2. \( \square \)

**Theorem 92.** The algorithm C counts the number of max-weight models for a formula \( F \) in time \( O^*(1.2377^n) \).

*Proof.* The correctness has been shown in Lemma 84. As for the time bound, if \( d(F) \leq 6 \), then this follows from Lemma 91. Otherwise, we can perform an quick analysis in terms of \( n(F) \): the measure \( n(F) \) is a well-behaved measure for the algorithm and since \( d(F) \geq 7 \), the branching number for case 5 is at worst \( \tau(1, 8) < 1.2321 \). \( \square \)
### Table 7.8: $d(F) = 5$ cases (neighbourhood and branching number)

<table>
<thead>
<tr>
<th>Case</th>
<th>Time</th>
<th>Case</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 2, 2, 2)</td>
<td>2</td>
<td>(2, 3, 4, 5, 5)</td>
<td>1.9311</td>
</tr>
<tr>
<td>(2, 2, 2, 3)</td>
<td>1.9862</td>
<td>(2, 3, 5, 5, 5)</td>
<td>1.9703</td>
</tr>
<tr>
<td>(2, 2, 2, 4)</td>
<td>1.9759</td>
<td>(2, 4, 4, 4, 4)</td>
<td>1.8837</td>
</tr>
<tr>
<td>(2, 2, 2, 5)</td>
<td>1.9777</td>
<td>(2, 4, 4, 4, 5)</td>
<td>1.8967</td>
</tr>
<tr>
<td>(2, 2, 3, 3)</td>
<td>1.9759</td>
<td>(2, 4, 4, 5, 5)</td>
<td>1.9329</td>
</tr>
<tr>
<td>(2, 2, 3, 4)</td>
<td>1.9397</td>
<td>(2, 4, 5, 5, 5)</td>
<td>1.9511</td>
</tr>
<tr>
<td>(2, 2, 3, 5)</td>
<td>1.9714</td>
<td>(2, 5, 5, 5, 5)</td>
<td>1.9949</td>
</tr>
<tr>
<td>(2, 2, 4, 4)</td>
<td>1.9334</td>
<td>(3, 3, 3, 3, 3)</td>
<td>1.9134</td>
</tr>
<tr>
<td>(2, 2, 4, 5)</td>
<td>1.9393</td>
<td>(3, 3, 3, 3, 4)</td>
<td>1.9114</td>
</tr>
<tr>
<td>(2, 2, 5, 5)</td>
<td>1.9730</td>
<td>(3, 3, 3, 5, 5)</td>
<td>1.9229</td>
</tr>
<tr>
<td>(2, 2, 3, 3, 3)</td>
<td>1.9410</td>
<td>(3, 3, 3, 4, 4)</td>
<td>1.8890</td>
</tr>
<tr>
<td>(2, 2, 3, 3, 4)</td>
<td>1.9353</td>
<td>(3, 3, 3, 4, 5)</td>
<td>1.9230</td>
</tr>
<tr>
<td>(2, 2, 3, 3, 5)</td>
<td>1.9422</td>
<td>(3, 3, 3, 5, 5)</td>
<td>1.9382</td>
</tr>
<tr>
<td>(2, 2, 3, 4, 4)</td>
<td>1.9066</td>
<td>(3, 3, 4, 4, 4)</td>
<td>1.8891</td>
</tr>
<tr>
<td>(2, 2, 3, 4, 5)</td>
<td>1.9386</td>
<td>(3, 3, 4, 4, 5)</td>
<td>1.9032</td>
</tr>
<tr>
<td>(2, 2, 3, 5, 5)</td>
<td>1.9488</td>
<td>(3, 3, 4, 5, 5)</td>
<td>1.9407</td>
</tr>
<tr>
<td>(2, 2, 4, 4, 4)</td>
<td>1.9035</td>
<td>(3, 3, 5, 5, 5)</td>
<td>1.9604</td>
</tr>
<tr>
<td>(2, 2, 4, 4, 5)</td>
<td>1.9130</td>
<td>(3, 4, 4, 4, 4)</td>
<td>1.8707</td>
</tr>
<tr>
<td>(2, 2, 4, 5, 5)</td>
<td>1.9475</td>
<td>(3, 4, 4, 4, 5)</td>
<td>1.9055</td>
</tr>
<tr>
<td>(2, 2, 5, 5, 5)</td>
<td>1.9613</td>
<td>(3, 4, 4, 5, 5)</td>
<td>1.9236</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 3)</td>
<td>1.9376</td>
<td>(3, 4, 5, 5, 5)</td>
<td>1.9658</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 4)</td>
<td>1.9098</td>
<td>(3, 5, 5, 5, 5)</td>
<td>1.9910</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 5)</td>
<td>1.9426</td>
<td>(4, 4, 4, 4, 4)</td>
<td>1.8728</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 4)</td>
<td>1.9072</td>
<td>(4, 4, 4, 4, 5)</td>
<td>1.8895</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 5)</td>
<td>1.9177</td>
<td>(4, 4, 4, 5, 5)</td>
<td>1.9285</td>
</tr>
<tr>
<td>(2, 3, 3, 5, 5)</td>
<td>1.9532</td>
<td>(4, 4, 5, 5, 5)</td>
<td>1.9516</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 4)</td>
<td>1.8841</td>
<td>(4, 5, 5, 5, 5)</td>
<td>2</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 5)</td>
<td>1.9172</td>
<td>(5, 5, 5, 5, 5)</td>
<td>Avoided</td>
</tr>
</tbody>
</table>
### 7.3. General Case

<table>
<thead>
<tr>
<th>Case</th>
<th>Time</th>
<th>Case</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>((2, 2, 2, 2, 2))</td>
<td>1.9489</td>
<td>((2, 2, 2, 5, 5, 6))</td>
<td>1.8766</td>
</tr>
<tr>
<td>((2, 2, 2, 2, 3))</td>
<td>1.9222</td>
<td>((2, 2, 2, 5, 6, 6))</td>
<td>1.8897</td>
</tr>
<tr>
<td>((2, 2, 2, 2, 4))</td>
<td>1.8964</td>
<td>((2, 2, 2, 6, 6, 6))</td>
<td>1.8977</td>
</tr>
<tr>
<td>((2, 2, 2, 2, 5))</td>
<td>1.9141</td>
<td>((2, 2, 3, 3, 3, 3))</td>
<td>1.8321</td>
</tr>
<tr>
<td>((2, 2, 2, 2, 6))</td>
<td>1.9187</td>
<td>((2, 2, 3, 3, 3, 4))</td>
<td>1.8208</td>
</tr>
<tr>
<td>((2, 2, 2, 3, 3))</td>
<td>1.8856</td>
<td>((2, 2, 3, 3, 3, 5))</td>
<td>1.8338</td>
</tr>
<tr>
<td>((2, 2, 2, 3, 4))</td>
<td>1.8697</td>
<td>((2, 2, 3, 3, 3, 6))</td>
<td>1.8447</td>
</tr>
<tr>
<td>((2, 2, 2, 3, 5))</td>
<td>1.8814</td>
<td>((2, 2, 3, 3, 4, 4))</td>
<td>1.8058</td>
</tr>
<tr>
<td>((2, 2, 2, 3, 6))</td>
<td>1.8926</td>
<td>((2, 2, 3, 3, 4, 5))</td>
<td>1.8243</td>
</tr>
<tr>
<td>((2, 2, 2, 4, 4))</td>
<td>1.8498</td>
<td>((2, 2, 3, 3, 4, 6))</td>
<td>1.8302</td>
</tr>
<tr>
<td>((2, 2, 2, 4, 5))</td>
<td>1.8679</td>
<td>((2, 2, 3, 3, 5, 5))</td>
<td>1.8395</td>
</tr>
<tr>
<td>((2, 2, 2, 4, 6))</td>
<td>1.8732</td>
<td>((2, 2, 3, 3, 5, 6))</td>
<td>1.8511</td>
</tr>
<tr>
<td>((2, 2, 2, 5, 5))</td>
<td>1.8817</td>
<td>((2, 2, 3, 3, 6, 6))</td>
<td>1.8580</td>
</tr>
<tr>
<td>((2, 2, 2, 5, 6))</td>
<td>1.8936</td>
<td>((2, 2, 3, 4, 4, 4))</td>
<td>1.7970</td>
</tr>
<tr>
<td>((2, 2, 2, 6, 6))</td>
<td>1.8998</td>
<td>((2, 2, 3, 4, 4, 5))</td>
<td>1.8112</td>
</tr>
<tr>
<td>((2, 2, 3, 3, 3))</td>
<td>1.8599</td>
<td>((2, 2, 3, 4, 4, 6))</td>
<td>1.8221</td>
</tr>
<tr>
<td>((2, 2, 3, 3, 4))</td>
<td>1.8408</td>
<td>((2, 2, 3, 4, 5, 5))</td>
<td>1.8318</td>
</tr>
<tr>
<td>((2, 2, 3, 3, 5))</td>
<td>1.8588</td>
<td>((2, 2, 3, 4, 5, 6))</td>
<td>1.8387</td>
</tr>
<tr>
<td>((2, 2, 3, 3, 6))</td>
<td>1.8641</td>
<td>((2, 2, 3, 4, 6, 6))</td>
<td>1.8507</td>
</tr>
<tr>
<td>((2, 2, 3, 4, 4))</td>
<td>1.8290</td>
<td>((2, 2, 3, 5, 5, 5))</td>
<td>1.8495</td>
</tr>
<tr>
<td>((2, 2, 3, 4, 5))</td>
<td>1.8420</td>
<td>((2, 2, 3, 5, 5, 6))</td>
<td>1.8621</td>
</tr>
<tr>
<td>((2, 2, 3, 4, 6))</td>
<td>1.8529</td>
<td>((2, 2, 3, 5, 6, 6))</td>
<td>1.8701</td>
</tr>
<tr>
<td>((2, 2, 3, 5, 5))</td>
<td>1.8619</td>
<td>((2, 2, 3, 6, 6, 6))</td>
<td>1.8836</td>
</tr>
<tr>
<td>((2, 2, 3, 5, 6))</td>
<td>1.8681</td>
<td>((2, 2, 4, 4, 4, 4))</td>
<td>1.7848</td>
</tr>
<tr>
<td>((2, 2, 3, 6, 6))</td>
<td>1.8802</td>
<td>((2, 2, 4, 4, 4, 5))</td>
<td>1.8041</td>
</tr>
<tr>
<td>((2, 2, 4, 4, 4))</td>
<td>1.8134</td>
<td>((2, 2, 4, 4, 4, 6))</td>
<td>1.8105</td>
</tr>
<tr>
<td>((2, 2, 4, 4, 5))</td>
<td>1.8320</td>
<td>((2, 2, 4, 4, 5, 5))</td>
<td>1.8206</td>
</tr>
<tr>
<td>((2, 2, 4, 4, 6))</td>
<td>1.8379</td>
<td>((2, 2, 4, 4, 5, 6))</td>
<td>1.8324</td>
</tr>
<tr>
<td>((2, 2, 4, 5, 5))</td>
<td>1.8472</td>
<td>((2, 2, 4, 4, 6, 6))</td>
<td>1.8399</td>
</tr>
<tr>
<td>((2, 2, 4, 5, 6))</td>
<td>1.8589</td>
<td>((2, 2, 4, 5, 5, 5))</td>
<td>1.8437</td>
</tr>
<tr>
<td>((2, 2, 4, 6, 6))</td>
<td>1.8658</td>
<td>((2, 2, 4, 5, 5, 6))</td>
<td>1.8517</td>
</tr>
<tr>
<td>((2, 2, 5, 5, 5))</td>
<td>1.8693</td>
<td>((2, 2, 4, 5, 6, 6))</td>
<td>1.8649</td>
</tr>
</tbody>
</table>

**Table 7.9:** \(d(F) = 6\) cases, part 1
### 7. Counting 2SAT

<table>
<thead>
<tr>
<th>Case</th>
<th>Time</th>
<th>Case</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 2, 4, 6, 6, 6)</td>
<td>1.8737</td>
<td>(2, 3, 4, 5, 5, 5)</td>
<td>1.8281</td>
</tr>
<tr>
<td>(2, 2, 5, 5, 5, 5)</td>
<td>1.8643</td>
<td>(2, 3, 4, 5, 5, 6)</td>
<td>1.8409</td>
</tr>
<tr>
<td>(2, 2, 5, 5, 5, 6)</td>
<td>1.8782</td>
<td>(2, 3, 4, 5, 6, 6)</td>
<td>1.8497</td>
</tr>
<tr>
<td>(2, 2, 5, 5, 6, 6)</td>
<td>1.8877</td>
<td>(2, 3, 4, 6, 6, 6)</td>
<td>1.8636</td>
</tr>
<tr>
<td>(2, 2, 6, 6, 6, 6)</td>
<td>1.9027</td>
<td>(2, 3, 5, 5, 5, 5)</td>
<td>1.8542</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 3, 3)</td>
<td>1.9132</td>
<td>(2, 3, 5, 5, 5, 6)</td>
<td>1.8636</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 3, 4)</td>
<td>1.8128</td>
<td>(2, 3, 5, 5, 6, 6)</td>
<td>1.8782</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 3, 5)</td>
<td>1.7984</td>
<td>(2, 3, 5, 6, 6, 6)</td>
<td>1.8887</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 3, 6)</td>
<td>1.8168</td>
<td>(2, 3, 6, 6, 6, 6)</td>
<td>1.9046</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 4, 4)</td>
<td>1.8227</td>
<td>(2, 4, 4, 4, 4, 4)</td>
<td>1.7624</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 4, 5)</td>
<td>1.7900</td>
<td>(2, 4, 4, 4, 4, 5)</td>
<td>1.7826</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 4, 6)</td>
<td>1.8043</td>
<td>(2, 4, 4, 4, 4, 6)</td>
<td>1.7895</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 4, 5)</td>
<td>1.8150</td>
<td>(2, 4, 4, 4, 5, 5)</td>
<td>1.8005</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 5, 5)</td>
<td>1.8248</td>
<td>(2, 4, 4, 4, 5, 6)</td>
<td>1.8125</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 5, 6)</td>
<td>1.8316</td>
<td>(2, 4, 4, 4, 6, 6)</td>
<td>1.8207</td>
</tr>
<tr>
<td>(2, 3, 3, 3, 6, 6)</td>
<td>1.8436</td>
<td>(2, 4, 4, 4, 5, 5)</td>
<td>1.8248</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 4, 4)</td>
<td>1.7782</td>
<td>(2, 4, 4, 5, 5, 6)</td>
<td>1.8336</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 4, 5)</td>
<td>1.7975</td>
<td>(2, 4, 4, 5, 6, 6)</td>
<td>1.8472</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 4, 6)</td>
<td>1.8039</td>
<td>(2, 4, 4, 6, 6, 6)</td>
<td>1.8569</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 5, 5)</td>
<td>1.8140</td>
<td>(2, 4, 5, 5, 5, 5)</td>
<td>1.8474</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 5, 6)</td>
<td>1.8257</td>
<td>(2, 4, 5, 5, 5, 6)</td>
<td>1.8617</td>
</tr>
<tr>
<td>(2, 3, 3, 4, 6, 6)</td>
<td>1.8332</td>
<td>(2, 4, 5, 5, 6, 6)</td>
<td>1.8722</td>
</tr>
<tr>
<td>(2, 3, 3, 5, 5, 5)</td>
<td>1.8370</td>
<td>(2, 4, 5, 6, 6, 6)</td>
<td>1.8878</td>
</tr>
<tr>
<td>(2, 3, 3, 5, 5, 6)</td>
<td>1.8450</td>
<td>(2, 4, 6, 6, 6, 6)</td>
<td>1.8995</td>
</tr>
<tr>
<td>(2, 3, 3, 5, 6, 6)</td>
<td>1.8581</td>
<td>(2, 5, 5, 5, 5, 5)</td>
<td>1.8774</td>
</tr>
<tr>
<td>(2, 3, 3, 6, 6, 6)</td>
<td>1.8669</td>
<td>(2, 5, 5, 5, 5, 6)</td>
<td>1.8887</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 4, 4)</td>
<td>1.7719</td>
<td>(2, 5, 5, 5, 6, 6)</td>
<td>1.9053</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 4, 5)</td>
<td>1.7874</td>
<td>(2, 5, 5, 6, 6, 6)</td>
<td>1.9179</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 4, 6)</td>
<td>1.7983</td>
<td>(2, 5, 6, 6, 6, 6)</td>
<td>1.9362</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 5, 5)</td>
<td>1.8089</td>
<td>(2, 6, 6, 6, 6, 6)</td>
<td>1.9505</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 5, 6)</td>
<td>1.8163</td>
<td>(3, 3, 3, 3, 3, 3)</td>
<td>1.7912</td>
</tr>
<tr>
<td>(2, 3, 4, 4, 6, 6)</td>
<td>1.8286</td>
<td>(3, 3, 3, 3, 3, 4)</td>
<td>1.7832</td>
</tr>
</tbody>
</table>

**Table 7.10:** \( d(F) = 6 \) cases, part 2
### 7.3. General Case

<table>
<thead>
<tr>
<th>Case</th>
<th>Time</th>
<th>Case</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3, 3, 3, 3, 5)</td>
<td>1.7975</td>
<td>(3, 3, 6, 6, 6, 6)</td>
<td>1.8935</td>
</tr>
<tr>
<td>(3, 3, 3, 3, 6)</td>
<td>1.8081</td>
<td>(3, 4, 4, 4, 4, 4)</td>
<td>1.7525</td>
</tr>
<tr>
<td>(3, 3, 3, 4, 4)</td>
<td>1.7719</td>
<td>(3, 4, 4, 4, 4, 5)</td>
<td>1.7693</td>
</tr>
<tr>
<td>(3, 3, 3, 4, 5)</td>
<td>1.7910</td>
<td>(3, 4, 4, 4, 4, 6)</td>
<td>1.7804</td>
</tr>
<tr>
<td>(3, 3, 3, 4, 6)</td>
<td>1.7974</td>
<td>(3, 4, 4, 4, 5, 5)</td>
<td>1.7919</td>
</tr>
<tr>
<td>(3, 3, 3, 5, 5)</td>
<td>1.8076</td>
<td>(3, 4, 4, 4, 5, 6)</td>
<td>1.8001</td>
</tr>
<tr>
<td>(3, 3, 3, 5, 6)</td>
<td>1.8191</td>
<td>(3, 4, 4, 4, 6, 6)</td>
<td>1.8127</td>
</tr>
<tr>
<td>(3, 3, 3, 6, 6)</td>
<td>1.8266</td>
<td>(3, 4, 4, 5, 5, 5)</td>
<td>1.8129</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 4)</td>
<td>1.7658</td>
<td>(3, 4, 5, 5, 5, 6)</td>
<td>1.8262</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 5)</td>
<td>1.7814</td>
<td>(3, 4, 5, 5, 6, 6)</td>
<td>1.8358</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 6)</td>
<td>1.7922</td>
<td>(3, 4, 6, 6, 6, 6)</td>
<td>1.8503</td>
</tr>
<tr>
<td>(3, 3, 4, 5, 5)</td>
<td>1.8028</td>
<td>(3, 4, 5, 5, 6, 6)</td>
<td>1.8407</td>
</tr>
<tr>
<td>(3, 3, 4, 5, 6)</td>
<td>1.8102</td>
<td>(3, 4, 5, 5, 5, 6)</td>
<td>1.8511</td>
</tr>
<tr>
<td>(3, 3, 4, 6, 6)</td>
<td>1.8224</td>
<td>(3, 4, 5, 6, 6, 6)</td>
<td>1.8664</td>
</tr>
<tr>
<td>(3, 3, 5, 5, 5)</td>
<td>1.8220</td>
<td>(3, 4, 6, 6, 6, 6)</td>
<td>1.8780</td>
</tr>
<tr>
<td>(3, 3, 5, 6, 6)</td>
<td>1.8347</td>
<td>(3, 5, 5, 5, 6, 6)</td>
<td>1.8948</td>
</tr>
<tr>
<td>(3, 3, 6, 6, 6)</td>
<td>1.8435</td>
<td>(3, 5, 5, 5, 6, 6)</td>
<td>1.8675</td>
</tr>
<tr>
<td>(3, 3, 6, 6, 6, 6)</td>
<td>1.8573</td>
<td>(3, 5, 5, 5, 6, 6)</td>
<td>1.8838</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 4, 4)</td>
<td>1.7568</td>
<td>(3, 5, 5, 5, 6, 6)</td>
<td>1.8964</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 4, 5)</td>
<td>1.7768</td>
<td>(3, 5, 5, 6, 6, 6)</td>
<td>1.9144</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 4, 6)</td>
<td>1.7838</td>
<td>(3, 5, 6, 6, 6, 6)</td>
<td>1.9286</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 5, 5)</td>
<td>1.7948</td>
<td>(3, 6, 6, 6, 6, 6)</td>
<td>1.9487</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 5, 6)</td>
<td>1.8067</td>
<td>(4, 4, 4, 4, 4, 4)</td>
<td>1.7454</td>
</tr>
<tr>
<td>(3, 3, 4, 4, 6, 6)</td>
<td>1.8149</td>
<td>(4, 4, 4, 4, 4, 5)</td>
<td>1.7665</td>
</tr>
<tr>
<td>(3, 3, 4, 5, 5, 5)</td>
<td>1.8190</td>
<td>(4, 4, 4, 4, 4, 6)</td>
<td>1.7741</td>
</tr>
<tr>
<td>(3, 3, 4, 5, 5, 6)</td>
<td>1.8278</td>
<td>(4, 4, 4, 4, 5, 5)</td>
<td>1.7861</td>
</tr>
<tr>
<td>(3, 3, 4, 5, 6, 6)</td>
<td>1.8413</td>
<td>(4, 4, 4, 4, 5, 6)</td>
<td>1.7985</td>
</tr>
<tr>
<td>(3, 3, 4, 6, 6, 6)</td>
<td>1.8510</td>
<td>(4, 4, 4, 4, 6, 6)</td>
<td>1.8074</td>
</tr>
<tr>
<td>(3, 3, 5, 5, 5, 5)</td>
<td>1.8416</td>
<td>(4, 4, 4, 5, 5, 5)</td>
<td>1.8119</td>
</tr>
<tr>
<td>(3, 3, 5, 5, 5, 6)</td>
<td>1.8558</td>
<td>(4, 4, 4, 5, 5, 6)</td>
<td>1.8216</td>
</tr>
<tr>
<td>(3, 3, 5, 5, 6, 6)</td>
<td>1.8663</td>
<td>(4, 4, 4, 5, 6, 6)</td>
<td>1.8358</td>
</tr>
<tr>
<td>(3, 3, 5, 6, 6, 6)</td>
<td>1.8818</td>
<td>(4, 4, 4, 6, 6, 6)</td>
<td>1.8465</td>
</tr>
</tbody>
</table>

**Table 7.11:** $d(F) = 6$ cases, part 3
### Table 7.12: $d(F) = 6$ cases, part 4

<table>
<thead>
<tr>
<th>Case</th>
<th>Time</th>
<th>Case</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4, 4, 5, 5, 5, 5)</td>
<td>1.8367</td>
<td>(4, 5, 6, 6, 6, 6)</td>
<td>1.9336</td>
</tr>
<tr>
<td>(4, 4, 5, 5, 6)</td>
<td>1.8518</td>
<td>(4, 6, 6, 6, 6, 6)</td>
<td>1.9498</td>
</tr>
<tr>
<td>(4, 4, 5, 6, 6)</td>
<td>1.8634</td>
<td>(5, 5, 5, 5, 5)</td>
<td>1.9017</td>
</tr>
<tr>
<td>(4, 4, 5, 6, 6, 6)</td>
<td>1.8800</td>
<td>(5, 5, 5, 5, 5, 6)</td>
<td>1.9208</td>
</tr>
<tr>
<td>(4, 4, 6, 6, 6, 6)</td>
<td>1.8930</td>
<td>(5, 5, 5, 5, 6, 6)</td>
<td>1.9364</td>
</tr>
<tr>
<td>(4, 5, 5, 5, 5, 5)</td>
<td>1.8692</td>
<td>(5, 5, 5, 6, 6, 6)</td>
<td>1.9579</td>
</tr>
<tr>
<td>(4, 5, 5, 5, 6)</td>
<td>1.8818</td>
<td>(5, 5, 6, 6, 6, 6)</td>
<td>1.9757</td>
</tr>
<tr>
<td>(4, 5, 5, 6, 6, 6)</td>
<td>1.8995</td>
<td>(5, 6, 6, 6, 6, 6)</td>
<td>2</td>
</tr>
<tr>
<td>(4, 5, 6, 6, 6, 6)</td>
<td>1.9138</td>
<td>(6, 6, 6, 6, 6, 6)</td>
<td>Avoided</td>
</tr>
<tr>
<td>(4, 5, 6, 6, 6, 6)</td>
<td>1.9138</td>
<td>(6, 6, 6, 6, 6, 6)</td>
<td>Avoided</td>
</tr>
</tbody>
</table>

7. Counting 2SAT
7.3. General Case
Chapter 8

Counting 3SAT

In this chapter, we give the algorithm D for solving the \#3SAT\_w problem in time $O^* (1.6671^n)$, making a slight improvement on our previous bound $O^* (1.6737^n)$ [15]. The approach we use here is quite similar to that used for 3HS (and indeed the problems are quite similar; this algorithm can be used to count the number of min-weight hitting sets for a 3HS instance). The analysis is based on finite states, which represent the number of short clauses.

Recall that in this chapter, as in Chapter 7, we treat clauses essentially as sets of literals, i.e. a clause contains no more than one copy of a literal.

8.1 The Algorithm

In this section we will present the algorithm D for \#3SAT\_w. We refer to Chapter 7 for definitions of the propagation procedure Prop, of the multiplier reduction, and of the meaning of \textit{recursively branch on}.

The algorithm is given below. When starting, assume that $F$ cannot be further simplified by Prop.

\textbf{Algorithm 93.} D($F, c, w$):

1. If $F = \emptyset$, then return 1. If $\emptyset \in F$, then return 0.
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 $d(v) = d_3(v) = 1$, then let $a$ be a neighbour of maximum degree and recursively branch on $a$.

5. If there exists a variable $v$ such that $d(v) = 2$ and $d_2(v) > 0$, then let $a$ be a neighbour that shares a 3-clause with $v$, if possible, or else a neighbour of maximum $d_2(a)$, and recursively branch on $a$.

6. If there exists at least one 2-clause in $F$, then let $v$ be a variable with maximum $d(v)$ among all variables with maximum $d_2(v)$, and recursively branch on $v$.

7. If there exists a variable $v$ such that $d(v) = d_3(v) = 2$ then, assuming that one 3-clause containing $v$ is $(v \lor a \lor b)$, recursively branch on $b = 1$, $b = 0 \land a = 1$, and $b = 0 \land a = 0 \land v = 1$. Similarly for other 3-clauses containing $v$ or $\overline{v}$.

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

Algorithm ends.

**Lemma 94.** $D(F, c, w) = \#3\text{SAT}_w(F, c, w)$.

**Proof.** The correctness of each step follows from the correctness of Prop, multiplier reduction, and of the process of recursively branching (see Chapter 7), and the completeness of the algorithm is obvious. $\blacksquare$
### 8.2 The Analysis

For analysing the running time of $D$, we use the “finite global states modelling” approach also used in Chapter 6: the global state that is modelled is the number of 2-clauses (in the categories of $m_2 = 0, m_2 = 1, \ldots, m_2 \geq 4$), and our measure of complexity is $f(F) = n - \Psi(m_2(F))$, where $\Psi(k)$ is the amount by which we modify the measure when there are $k$ 2-clauses in the formula. The values of $\Psi(k)$ are optimised in the usual manner. We use $\Delta \Psi(k)$ for the incremental cost $\Psi(k) - \Psi(k - 1)$ of the $k$:th 2-edge.

We give two sets of values: $\Psi_4(k)$, for modelling the hard case that $d(F) \leq 4$, and $\Psi_{\geq 5}(k)$, for modelling the easier case $d(F) \geq 5$. The weights are given in Table 8.1. We will prove that using the measures $f_4(F) = n(F) - \Psi_4(m_2(F))$ when $d(F) \leq 4$ and $f_{\geq 5}(F) = n(F) - \Psi_{\geq 5}(F)$ when $d(F) \geq 5$, each case in $D(F)$ gets a branching number of at most 1.6671.

We use $k$ to denote the number of 2-clauses in $F$. First, we see directly that both measures are well-behaved for the algorithm, since all values of $\Psi(k)$ are less than one. Note also that multiplier reduction applies to the case that $d(v) = d_2(v) = 1$, and to the case that $d(v) = d(w) = 1$ when $v$ and $w$ appear in the same clause. We will present our proof as a sequence of lemmas for the cases of the algorithm, beginning with case 4.

**Lemma 95.** Case 4 results in a branching number of at most 1.6181.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$\Psi_4(k)$</th>
<th>$\Delta \Psi_4(k)$</th>
<th>$\Psi_{\geq 5}(k)$</th>
<th>$\Delta \Psi_{\geq 5}(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>N/A</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>1</td>
<td>0.238220</td>
<td>0.238220</td>
<td>0.206606</td>
<td>0.206606</td>
</tr>
<tr>
<td>2</td>
<td>0.452546</td>
<td>0.214326</td>
<td>0.404816</td>
<td>0.198211</td>
</tr>
<tr>
<td>3</td>
<td>0.617605</td>
<td>0.165058</td>
<td>0.586082</td>
<td>0.181266</td>
</tr>
<tr>
<td>4</td>
<td>0.761780</td>
<td>0.144175</td>
<td>0.733793</td>
<td>0.147711</td>
</tr>
<tr>
<td>$\geq 5$</td>
<td>0.879032</td>
<td>0.117252</td>
<td>0.835032</td>
<td>0.101239</td>
</tr>
</tbody>
</table>

Table 8.1: Weights for the states when $d(F) \leq 4$ and $d(F) \geq 5$
Proof. Assume without loss of generality that the variable $v$ appears in the clause $(v \lor a \lor b)$. The variables $a$ and $v$ will both disappear in both branches (as multiplier reduction applies on $b$ in the $a = 0$ branch, unless some other reduction applies which removes both $b$ and $v$). If no more variables are removed, then no 2-clause can disappear in the $a = 1$ branch and the branching is dominated by $\tau(2 - \Psi(k), 2) < \tau(1, 2) < 1.6181$; otherwise, the branching is dominated by $\tau(2 - \Psi(k), 3 - \Psi(k)) < \tau(1, 2) < 1.6181$.

Next, we prove the bound for case 5.

**Lemma 96.** Case 5 results in a branching number of at most 1.6671.

**Proof.** If $d_2(v) = 1$, then assume without loss of generality that $v$ appears in the clause $(v \lor a \lor b)$. In the branch $a = 1$, $v$ is removed (by multiplier reduction if nothing else), along with the 2-clause containing $v$, while in the branch $a = 0$, the 2-clause $(v \lor b)$ is created. In addition, $a$ appears in at least one more clause, which results in either an extra 2-clause in one branch, or fewer 2-clauses but one more assignment in some branch.

1. If $v$ appears in some 2-clause containing a literal of $a$ (or $b$), then $v$ will be removed in the $a = 0$ branch as well and the branching is dominated by either $\tau(2 - \Psi(1), 2 - \Psi(1)) < \tau(1.75, 1.75) < 1.4860$ if no further variable is removed, or $\tau(2 - \Psi(k), 3 - \Psi(k)) < \tau(1, 2) < 1.6181$ otherwise.

2. If no further variable is removed in any branch, then one more 2-clause is created in some branch and the branching is dominated by $\tau(1 + \Delta \Psi(k + 1), 2) < \tau(1, 2) < 1.6181$ or by $\tau(1 + \Psi(k + 2) - \Psi(k), 2 - \Delta \Psi(k))$, which when using $\Psi_4$ is at most $\tau(1, 2 - \Delta \Psi_4(5)) < 1.6408$, and when using $\Psi_5$ is no higher than $\tau(1, 2 - \Psi_{\geq 5}(1)) < 1.6668$.

3. Otherwise, $a$ appears in one or several 2-clauses.

(a) If the literal $a$ appears in at least two 2-clauses, or if both literals $a$ and $\bar{a}$ appear in 2-clauses, then the branching is dominated by $\tau(2 - \Psi(k), 3 - \Psi(k)) < \tau(1, 2) < 1.6181$. 


(b) If the literal \( a \) appears in exactly one 2-clause, then the branching is dominated by \( \tau(2 - (\Psi(k) - \Psi(k - 2)), 2 - \Psi(k)) < \tau(2 - \Psi(2), 1) \) which is less than 1.6563 using \( \Psi_4 \) and 1.6603 using \( \Psi_{\geq 5} \).

(c) If the literal \( \bar{a} \) appears in \( i \) 2-clauses, then the branching is dominated by \( \tau(2 + i - \Psi(k), 1 - (\Psi(k) - \Psi(k + 1 - i))) < \tau(1 + i, 1 - \Psi(i - 1)) \), which for every value of \( i \) is dominated by some case appearing in case 6 (see Table 8.2).

Hence, the claim holds for all cases with \( d_2(v) = 1 \).

If \( d_2(v) = 2 \), then assume without loss of generality that there exists a 2-clause \( (v \lor a) \) in \( F \). In both branches, at least the variables \( v \) and \( a \) are removed, as well as at least two 2-clauses. If \( a \) is involved in some 2-clause not containing \( v \) or \( \bar{v} \), then at least one more variable is removed in some branch, leading to a branching dominated by \( \tau(2 - \Psi(k), 3 - \Psi(k)) < \tau(1, 2) < 1.6181 \), otherwise \( d_3(a) > 0 \) and we have a worst-case branching dominated by \( \tau(2 - \Psi(2), 2 - \Psi(1)) \), which is less than 1.5213 with \( \Psi_4 \) and 1.5063 with \( \Psi_{\geq 5} \).

Now for case 6, the first case that has a worst-case branching number matching that of the algorithm as a whole (i.e. the first hard case).

**Lemma 97.** Case 6 results in a branching number of at most 1.6671.

**Proof.** We split the analysis by \( d_2(v) \). Note that the worst case branching for a particular value of \( d_2(v) \) will always have a minimum \( d_3(v) \): if \( v \) is a literal in a 3-clause, then this 3-clause contributes nothing when \( v = 1 \) and increases \( k \) by 1 when \( v = 0 \). The branchings and branching numbers are given in Table 8.2. Every branching number is at most 1.6671 when \( \Psi_4 \) is used, and 1.6562 when \( \Psi_{\geq 5} \) is used. We will now show that the branchings used are the worst possible branchings.

- If \( d_2(v) = 1 \), then the worst case is when \( d(v) = 3 \), and supposing that the 2-clause is \( (v \lor a) \), we know that \( d_3(a) = 1 \), so only one 2-clause is removed in both branches. Also, \( d_3(v) = 2 \),
Table 8.2: Branching tuples and branching numbers for case 6

<table>
<thead>
<tr>
<th>$d_2(v)$</th>
<th>$k$</th>
<th>Branching</th>
<th>$d(F) \leq 4$</th>
<th>$d(F) \geq 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>$\tau(1 - \Psi(1), 2 + \Delta \Psi(2))$</td>
<td>1.6671</td>
<td>1.6562</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>$\tau(1 - \Delta \Psi(2), 2 + \Delta \Psi(3))$</td>
<td>1.6671</td>
<td>1.6562</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>$\tau(1 - \Delta \Psi(3), 2 + \Delta \Psi(4))$</td>
<td>1.6501</td>
<td>1.6562</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>$\tau(1 - \Delta \Psi(4), 2 + \Delta \Psi(5))$</td>
<td>1.6472</td>
<td>1.6523</td>
</tr>
<tr>
<td>1 $\geq$ 5</td>
<td></td>
<td>$\tau(1 - \Delta \Psi(5), 2)$</td>
<td>1.6629</td>
<td>1.6562</td>
</tr>
<tr>
<td>2</td>
<td>Any</td>
<td>$\tau(2 - \Psi(3), 2 - \Psi(3))$</td>
<td>1.6511</td>
<td>1.6327</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$\tau(1 - \Psi(2), 3 - \Delta \Psi(2))$</td>
<td>1.6622</td>
<td>1.6330</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$\tau(1 - \Delta \Psi(2), 3 - \Psi(2))$</td>
<td>1.5919</td>
<td>1.5778</td>
</tr>
<tr>
<td>2 $\geq$ 3</td>
<td>3</td>
<td>$\tau(1 - (\Psi(3) - \Psi(1)), 3 - (\Psi(3) - \Psi(1)))$</td>
<td>1.6530</td>
<td>1.6531</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>$\tau(1 - \Delta \Psi(3), 3 - \Psi(3))$</td>
<td>1.6022</td>
<td>1.6026</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>$\tau(1 - (\Psi(4) - \Psi(2)), 3 - (\Psi(4) - \Psi(1)))$</td>
<td>1.6458</td>
<td>1.6562</td>
</tr>
<tr>
<td>2 $\geq$ 4</td>
<td>4</td>
<td>$\tau(1 - \Delta \Psi(4), 3 - \Psi(4))$</td>
<td>1.6218</td>
<td>1.6176</td>
</tr>
<tr>
<td>2 $\geq$ 5</td>
<td></td>
<td>$\tau(1 - (\Psi(5) - \Psi(3)), 3 - (\Psi(5) - \Psi(2)))$</td>
<td>1.6063</td>
<td>1.6018</td>
</tr>
<tr>
<td>2 $\geq$ 5</td>
<td></td>
<td>$\tau(1 - \Delta \Psi(5), 3 - (\Psi(5) - \Psi(1)))$</td>
<td>1.5888</td>
<td>1.5810</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$\tau(1 - \Psi(3), 4 - \Psi(3))$</td>
<td>1.6671</td>
<td>1.6393</td>
</tr>
<tr>
<td>3 $\geq$ 4</td>
<td>4</td>
<td>$\tau(1 - (\Psi(4) - \Psi(1)), 4 - \Psi(4))$</td>
<td>1.6267</td>
<td>1.6245</td>
</tr>
<tr>
<td>3 $\geq$ 5</td>
<td></td>
<td>$\tau(1 - (\Psi(5) - \Psi(2)), 4 - \Psi(5))$</td>
<td>1.5923</td>
<td>1.5877</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$\tau(1 - \Psi(4), 5 - \Psi(4))$</td>
<td>1.6671</td>
<td>1.6353</td>
</tr>
<tr>
<td>4 $\geq$ 5</td>
<td></td>
<td>$\tau(1 - (\Psi(5) - \Psi(1)), 5 - \Psi(5))$</td>
<td>1.5806</td>
<td>1.5675</td>
</tr>
<tr>
<td>$\geq$ 5</td>
<td>5</td>
<td>$\tau(1 - \Psi(5), 6 - \Psi(5))$</td>
<td>N/A</td>
<td>1.6380</td>
</tr>
</tbody>
</table>

Table 8.2: Branching tuples and branching numbers for case 6
resulting in two newly created 2-clauses. The worst case is then the case where both 3-clauses use the literal \( v \), so that the branching is \( \tau(1 - \Delta\Psi(k), 2 + \Delta\Psi(k + 1)) \). The branching tuples and branching numbers for these cases are lines 1–5 of Table 8.2. Cases with \( k > 5 \) result in \( \tau(1, 2) < 1.6181 \).

- If \( d_2(v) = 2 \), then we similarly have \( d(v) = 3 \) and, if the neighbours of \( v \) in the 2-clauses are \( a \) and \( b \), then \( d_2(a) \leq 2 \) and \( d_2(b) \leq 2 \). For the non-pure case \( (v \lor a), (\bar{v} \lor b) \) we have, disregarding the 3-clause, two variables and two or three 2-clauses removed both when \( v = 1 \) and \( v = 0 \); the worst case is the case in line 6 of Table 8.2. For the pure case \( (v \lor a), (v \lor b) \), we have one variable and two 2-clauses removed when \( v = 1 \), and three variables and up to four 2-clauses removed when \( v = 0 \). One 2-clause will be created in one of the branches, guaranteeing that \( k \geq 1 \) in that branch. We get two cases for each value of \( k \), given in lines 7–14 of Table 8.2. No new worst cases will appear when \( k > 5 \), as \( \Psi(k) \) flattens out and \( \Psi(k) - \Psi(k') \) decreases.

- If \( d_2(v) \geq 3 \), then the worst case is when \( d_2(v) = 0 \). If both literals \( v \) and \( \bar{v} \) appear in 2-clauses, then the branching is dominated by \( \tau(2 - \Psi(k), 3 - \Psi(k)) < \tau(1, 2) < 1.6181 \). Otherwise, let \( d_2(v) = i \); we have cases with branchings dominated by \( \tau(1 - (\Psi(k) - \Psi(k - i)), 1 + i - \Psi(k)) \), given in lines 15–20 of Table 8.2.

Now, we give the bound for case 7, which is fairly easy.

Lemma 98. Case 7 results in a branching number of at most 1.6181.

Proof. When case 7 is reached, we know that \( k = 0 \), so we only need to count the number of variables removed. In the first two branches, either \( v \) is removed by a reduction or case 4 is met. If some extra variable is removed in the first branch, then the branching \( \tau(2, 2, 3) < 1.6181 \) is good enough. If some extra variable is removed in the second branch, then we get a branching of \( \tau(3, 3, 3, 3) < 1.5875 \)
due to case 4 being reached in the first branch. Otherwise, we get a branching of \( \tau(3, 3, 4, 4, 3) < 1.6181 \).

Now we reach the final case, which requires a bit more effort to prove.

**Lemma 99.** Case 8 results in a branching number of at most 1.6671.

*Proof.* By the balancing property of \( \tau \) and the non-increasing values of \( \Delta \Psi(k) \), the worst cases are when \( v \) is pure, so assume that only the literal \( v \) appears. If \( d(F) \geq 5 \), then the branching \( \tau(1, 1 + \Psi_{\leq 5}(5)) < 1.6562 \) is good enough. The case when \( d(F) = 3 \) and case 8 is reached occurs only when \( F \) is 3-regular; as every modification of the formula that our algorithm performs is either a deletion of a variable of clause, or a shortening of a clause, this situation only occurs once along each path down the branching tree, and does not change the asymptotic running time. If \( d(F) = 4 \), finally, then we have to look at the \( v = 0 \) branch closer.

Note that if some variable \( w \) occurs in every 2-clause created in the \( v = 0 \) branch, then in the \( v = 1 \) branch the variable \( w \) has no occurrences, and we have a branching dominated by \( \tau(1, 2) < 1.6181 \), and the same holds for any other reduction that removes some extra variable. Otherwise, in the branch \( v = 0 \), all variables occur in less than \( d(v) \) 2-clauses, and one of cases 4–6 is reached.

- If the first case that is used in the \( v = 0 \) branch is case 4, then either the branching is dominated by \( \tau(1, 3, 3 + \Psi_4(4)) < 1.6331 \) or by \( \tau(1, 3, 4) < 1.6181 \).

- If the first case used is case 5, then there are many possibilities, but every one is dominated by one of \( \tau(1, 3, 3 + \Psi_4(2)) < 1.6555 \), \( \tau(1, 2 + \Psi_4(5), 3 + \Psi_4(1)) < 1.6553 \), or one of the cases that occur if case 6 is the first case reached.

- If case 6 is the first case reached, then let \( w \) be the variable that is branched upon. If \( d_2(w) = 1 \), then the branching is dominated by \( \tau(1, 2 + \Psi_4(3), 3 + \Psi_4(5)) < 1.6671 \). If \( d_2(w) = 2 \), then the possible dominating cases are \( \tau(1, 3 + \Psi_4(1), 3 + \Psi_4(1)) < 1.6511 \),
Finally, we state our main result.

**Theorem 100.** \( D(F) \) runs in time \( O^*(1.6671^n) \).

*Proof.* This follows from Lemmas 95–99.

Figure 8.1 illustrates the state diagram corresponding to the loop of hard cases of this analysis. Though there are hard cases not represented in the diagram, these cases are not part of any loop of only hard cases (e.g. the case where case 8 of the algorithm is followed by case 6 with \( d_2(w) = 1 \) is a hard case, but has a child where \( k = 5 \), and there are no hard cases for \( k \geq 5 \)). For the same reason, although the states \( k = 4 \) and \( k \geq 5 \) are not represented in the diagram (except implicitly through case 8, in the \( k = 4 \) case), these extra cases are nonetheless needed in the analysis to keep other branchings from getting a too big branching number.
8.2. The Analysis
Chapter 9

Future Work

Here, we review some open questions and possible directions of future research.

9.1 Algorithm Analysis and Complexity Measures

We have specified complexity measures based on finite global states and compound measures, and shown how to convert models of these types to models fitting in Eppstein’s framework, but we have not shown to satisfaction whether this conversion introduces non-tightness in the final bound or not.

The tightness result that Eppstein has given for his framework is quite strong; the only question which may perhaps be examined more closely is the influence of the target vector. On the other hand, the tightness result that we have given for the model of finite global states only applies to models with two parameters, one of which is the state, and for compound measures we have given no tightness result at all.

For the state-based approach at least, we would conjecture that the end result is still tight (relative to a given target vector) when more parameters are involved in the analysis.

For the generic case of an analysis by compound measure, this seems a bit too much to ask, but if so, then a tightness result may
still be possible for restricted cases; an obvious candidate for such a restricted case is when the division into sections depends on only one quotient of the modelled attributes (e.g. the value of $l(F)/n(F)$), possibly with the additional restriction that the applicability for every branching is on the form of an upper bound on this quotient (e.g. a case may be applicable when and only when $l(F)/n(F) \leq k$).

There are also other questions about the approach of compound measures in general. For instance, is it always enough to only use those sections immediately given by the applicability constraints, or are there situations where we would get better bounds by introducing further subdivisions among the sections (perhaps even an infinite amount of division)? Also, we note that each application so far (both published and tentative work) has been using some kind of single density parameter for the division into sections. There may be extra complications that arise when this is not the case, that would need to be examined. (One observation to make is that for any division into sections, with a section boundary expressed as a linear function $\alpha_1 h_1 + \alpha_2 h_2 + \ldots = \beta$ of the attributes $h_i$, if $\alpha_i = 0$ for some $i$ then weight $w_i$ of the attribute $h_i$ must be equal on both sides of this boundary. For instance, when switching sections depending on whether the average degree is at most or at least 3, the value of $w_3$ cannot change, since adding or removing 3-variables will not cause us to leave the boundary.)

There are similarities between this method of analysis and Markov decision processes [69]; further study would be needed to determine how deep these go and whether the extensive Markov theory can be of any application here.

Leaving the questions of tightness behind, some method to reduce the number of cases in an analysis in Eppstein’s framework would be greatly welcome. When the weight of a variable depends on a single property of the variable, such as the degree, then the number of cases is usually not a problem, but if we additionally want to add that the weight of a variable depends on both its degree and the lengths of the clauses where it appears, then the number of possible weights grows quadratically, and the number of possible neighbourhoods that we
9. Future Work

would need to enumerate explodes. For instance, we may be interested in performing an analysis of an algorithm similar to that for \#3Sat\_w in Chapter 8 with a weight for a variable that depends on both its degree and the number of 2-clauses it appears in.

One observation (that we will state here, but not prove) is that at least for binary branchings, the balancing property of the branching numbers (i.e., the property that \(\tau(a - \delta, b + \delta) > \tau(a, b)\) when \(a < b\) and \(\delta > 0\)) can be extended: if \(\tau(a, b) = c\), \(a \leq b\), and \(\delta > 0\), then \(\tau(a - \delta, b + \delta \cdot k) \geq \tau(a, b)\) for \(k \leq 1/(c^a - 1)\) (express \(b\) as a function of \(a\) and \(c\), then take the derivative with respect to \(a\)). By similar reasoning to this, it may be possible to identify in advance that certain groups of cases are the only possible worst cases.

Any further extensions of the complexity measures approach, or of analysis of exponential upper bounds in general, would of course also be welcome. One suggested path that has been used e.g. by Kullmann [56], and by Chen, Kanj, and Xia [10] (in the latter case under the name of amortised analysis) is to generalise the concept of a complexity measure to a form of distance function giving the labels of a branching tree, in a way so that this branching tree as a whole can be considered in the analysis. Interestingly, in the case of Chen, Kanj, and Xia, in a later publication on the same problem [11] they have replaced their method of analysis by an inductive analysis they refer to as local amortised analysis, which seems to be equivalent to what we refer to as state-based analysis. In this case they have changed the algorithm as well so that the object of analysis is different, but still, it does perhaps raise the question of under what conditions such an analysis yields a better bound.

This question can also be asked generically: how do we know which approach of analysis will give the best results? To make it more concrete: under what conditions will one method of analysis give better bounds than another, and when will they be equivalent?

Linking into this, it would be good to have more examples of lower bounds for our algorithms, e.g. classes of instances for which we can prove that a certain exponential behaviour can occur. Note that unlike e.g. the work on proof systems for unsatisfiability, these
bounds do not have to be valid for any wider classes of algorithms; for our purposes it is enough to prove that a particular algorithm will in the worst case have $\Omega(c^n)$ behaviour, so that we can judge how big the gap between the algorithm’s actual worst-case behaviour and our upper bound is. While this would mean that the analysis has to be repeated a larger number of times, it seems likely that certain patterns of hard instances, and tools of analysis, will repeat themselves. Some work in this vein has been performed in the context of the “measure and conquer” approach [37–39].

It may be possible that some adversarial argument can be adapted for this purpose: given a set of branches of hard branchings that have been taken by an algorithm, we may ask whether there always exists some input instance for which such a sequence of branches is a possible behaviour for our algorithm, and where a further hard branching step could be taken. However, this idea is admittedly not very concrete.

### 9.2 Connections Between Parameterised and Classic Approaches

Connections between classical and parameterised analysis of upper bounds can probably be examined closer. In Chapter 6, we use connections in two directions: we use more advanced complexity measures in a parameterised analysis to improve the parameterised bound, and we use the parameterised analysis to improve the non-parameterised bound, by bounding the possible values of the parameter in terms of the classical parameter $n(F)$ for certain cases. It certainly seems possible that the same general thing can be done in other cases; the problem of Independent Set/Vertex Cover seems to some extent to be developing in this direction already. As we have already mentioned a handful of times, the best parameterised bound $O\left(1.2738^k + kn\right)$ for Vertex Cover is by an analysis that seems equivalent to the state-based analysis of this thesis [11]; and bounds on the running time for solving Independent Set for graphs with max-degree 3 have been deduced from bounds on Vertex Cover for such graphs [10], though the current best bound of $O^*(1.1034^n)$ [70] is
produced through non-parameterised methods\textsuperscript{1}.

However, although in principle it should be possible to use any measure that is bounded by the parameter $k$ in the analysis of a purely parameterised algorithm, in practice it is not trivial to produce such measures in a useful manner. The state-based measure can be used, since it only introduces constant-sized perturbations in the measure of an instance, but while e.g. splitting the parameter $n(F)$ into several parameters $n_i(F)$ is straightforward enough in a classic context, it is not clear what a corresponding split of a parameter $k$ measuring the maximum size of the solution into several parameters $k_i$ which somehow depend on variable degree would signify.

Tighter couplings than this may be possible as well, e.g. hybrid approaches that simultaneously consider parameterised and classic attributes. One could view such a hypothetical approach as a unified field of analysis of upper bounds.

### 9.3 Automated Analysis

Given the nature of a typical case analysis for a branching algorithm, it is tempting to suggest that the step of finding the possible branchings may be automated as well. To some extent this is already done—see for instance the lists of degrees of neighbours in Chapter 7, and similar enumerations in the analysis of e.g. Minimum Set Cover of Fomin et al. [36]—but such lists may cover only the “regular” cases, or omit other aspects of the neighbourhood so that cases appear harder than they are. Such an analysis is also harder to construct for formulae than for graphs.

The case analysis is often equivalent to enumerating all local neighbourhoods up to some maximum degree, and for each neighbourhood either considering the effects of the branching rule or concluding that the branching rule cannot be used on that particular neighbourhood, and in addition, it is often a limited, regular, and somewhat

\textsuperscript{1}I found a reference through Google Scholar to a paper in a Chinese journal giving $O(1.1030^n)$ [86], that from the summary seems to be based on a parameterised process, but I have been unable to locate the actual paper.
predictable set of cases (though not always so) that turn out to be
the hard cases (while all other cases present the large number of “spe-
cial cases”, where a large amount of effort and paper space is used
to show that they are not hard). It would not seem to be much of a
loss of potential “theoretic insight” to be able to refer to a computer
evaluation rather than a manual enumeration in a case such as this.
However, we are not aware of any work that performs or simplifies an
automatic analysis of a provided algorithm. (It is of course quite pos-
sible that many “helper applications” exist as personal, unpublished
programs that individual researchers use to verify and produce proofs
that are then provided as explicit proofs in the ordinary style. For
instance, it has been stated [44] that Robson used a similar program
in his work on Independent Set [71, 72].)

It seems that most approaches in this direction have instead in-
volved trying to automate the algorithm generation as well (e.g. pro-
ducing large numbers of case-specific branching rules); see for instance
the work by Gramm, Guo, Hüffner, and Niedermeier [44] on a frame-
work for automated generation of algorithms for graph modification
problems. Alternatively, in work such as that by Fedin and Ku-
likov [32], the algorithm which is analysed contains instructions to
try each of a number of branchings for each variable of the instance,
and to use the one that produces the best branching number, with
respect to a given measure. (Admittedly, the algorithm SparseSAT
of Chapter 4 of this thesis contains a case similar to this, and the al-
gorithm of Hirsch [49] for sat uses similar cases as well.) It seems to
be “common folklore” (e.g. Fernau [33] cites “personal communication
by P. Shaw” to this effect) that implementing large case enumerations
in algorithms often does not improve the actual performance of the
algorithm, and it would seem that attempting a quadratic number
of possible branchings for every branching that is actually performed
also runs the risk of being unrealistic in reality. To this end, it seems
to us that a better approach might be something similar to what
Fernau [33] describes as a “top-down approach”: to provide what he
calls a “heuristic”, i.e. general instructions on how to pick a branch-
ing variable (along the lines of “maximise the variable degree”), and
9. Future Work

to automate the analysis of such an algorithm. Besides the issue of implementing the algorithm, we consider it useful to know the underlying principle which causes an algorithm to fare well, rather than to (essentially) only know that a certain number $c$ (e.g. $\mathcal{O}^* (c^n)$) is achievable.

Finally, the question of proof validity may be raised. Even though the process will generate a list of cases that may in principle be checked, once the list is too large to be manually verified it either has to be taken on faith that any error in the program which generates the cases would be apparent in a large number of places, or otherwise always easy to detect by partial inspection of the proof; or correctness must be guaranteed through some other method. One option is to create the proof in the file format of some standardised proof system, but this is not likely to be easy, as it seems that such proof systems are far from trivial to use (and it still has to be verified that the result which is proven corresponds to the lemma in the paper, for which the reviewer must be familiar with the proof system, but perhaps this is acceptable). Another option is to present the actual program that has generated the case enumeration, along with further proofs that this program works correctly (or at least that the fundamental process used is correct). Perhaps time will tell which option is to be preferred.

9.4 Further Problems and Relations between Problems

All the problems considered in this thesis are boolean problems, with binary domains. It might be possible, and interesting, to extend or apply some of the techniques to work on other classes of constraint satisfaction problems.

In particular, as mentioned in Section 1.3.4, for the problem of counting solutions for $(d,2)$-CSP (i.e. problems with binary constraints but a $d$-ary domain), the best result is due to a reduction to $\#2\text{SAT}_w$, for which the best current bound is the one given in Chapter 7 in this thesis. At least for certain values of $d$, attacking
the problem directly in its native form may yield a better bound than this indirect approach.

Another type of problems, that are considered from the viewpoint of complexity theory, have to do with the type of relations that can be used (i.e. the type of clauses) [7]. While Post’s lattice is probably a division that is too coarse from an upper bounds point of view, the same principle can be used. For instance, standard 3SAT can be deterministically solved in time $O^*(1.473^n)$, while X3SAT gets a bound of $O^*(1.0984^n)$ in this thesis. It is also possible (joint unpublished work with Gustav Nordh) to solve Not-All-Equal 3SAT in time $O^*(1.455^n)$ (through a re-analysis of the local search routine under NAE constraints). While each individual such satisfiability problem may not hold independent interest, a wider study of several such problems and their relations to one another may prove interesting.

On the other side of the coin, we note that for this kind of problems, it is sometimes possible to create reductions between problems that only increase the number of variables by a constant amount. For instance, in addition to the trivial reductions from any constraint of arity $k$ to $k$-SAT, there exist reductions from $k$-SAT to NAE-$k + 1$-SAT that use only one additional variable. Between which further SAT variants do reductions that perform a sub-linear blowup in the number of variables exist? (Section 2.4 of Skjernaa’s thesis [76] also contains some work in this direction.)

This question about reductions can also be asked generally (though we then have fewer tools to attack it): between which problems do there exist reductions that do not significantly increase the number of variables (e.g. at most by a linear factor)? Is there any kind of evidence that such reductions are in some cases impossible in polynomial time? In either case, it seems fundamental that gadget-style reductions (that convert each piece of the input instance into an equivalent chunk of the output instance) cannot achieve this in general, as the number of new variables introduced by such a method will normally be linear in the length of the input instance.
Bibliography

tion and Related Problems: Methods and Applications*. Linköping
Studies in Science and Technology, PhD Dissertation no 947,
2005.

1997.

[3] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *Jour-


dolic model checking using SAT procedures instead of BDDs. In *Pro-
cedings of the 36th ACM/IEEE conference on Design Au-

[6] A. Björklund and T. Husfeldt. Exact algorithms for exact sat-

sifiability and number of perfect matchings. In *Proceedings of the 33rd
International Colloquium on Automata, Languages and


boolean blocks, part I: Post’s lattice with applications to com-


<table>
<thead>
<tr>
<th>No</th>
<th>Author</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>214</td>
<td>Tony Larsson</td>
<td>A Formal Hardware Description and Verification Method, 1989</td>
<td>ISBN 91-7870-517-7</td>
<td></td>
</tr>
<tr>
<td>252</td>
<td>Peter Eklund</td>
<td>An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991</td>
<td>ISBN 91-7870-784-6</td>
<td></td>
</tr>
<tr>
<td>281</td>
<td>Christer Bäckström</td>
<td>Computational Complexity</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


No 582 Vanja Josifovski: Design, Implementation and


No 1009 Iakov Nahkimovski: Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.


<table>
<thead>
<tr>
<th>No</th>
<th>Författare</th>
<th>Titel</th>
<th>År</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1035</td>
<td>Vaida Jakoniene</td>
<td>Integration of Biological Data, 2006</td>
<td>ISBN 91-85523-28-3</td>
<td></td>
</tr>
</tbody>
</table>

**Linköping Studies in Information Science**

<table>
<thead>
<tr>
<th>No</th>
<th>Författare</th>
<th>Titel</th>
<th>År</th>
<th>ISBN</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>No</th>
<th>Författare</th>
<th>Titel</th>
<th>År</th>
<th>ISBN</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Karin Hedström</td>
<td>Spår av datoriseringens värden - Effekter av IT i äldreomsorg, 2004</td>
<td>ISBN 91-7373-963-4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Ewa Braf</td>
<td>Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004</td>
<td>ISBN 91-85295-47-7</td>
<td></td>
</tr>
</tbody>
</table>