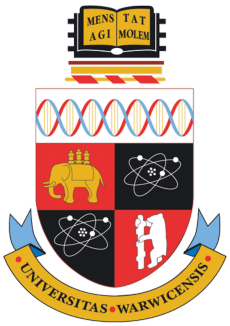
Scaling Behaviour of SAT: Theory and Practice

An exposition on the gap between theoretical conditional lower bounds for SAT and the practical performance of SAT solvers



Eavan Thomas Pattie

Supervisor: Dmitry Chistikov

Department of Computer Science
University of Warwick

Year of Study: 3rd

October 2019

Acknowledgements

I would like acknowledge my supervisor Dmitry Chistikov for our valuable conversations and guidance, without which the project would not have been completed. I would also like to thank Gabrielė Stravinskaitė for her continued support throughout the project. I'd like to thank my family for keeping my progress in check, and finally I would like to thank Ava, Aaron, Nicola and Michal for their help with typesetting and general support.

Abstract

The Boolean Satisfiability Problem (SAT) is a well studied NP-COMPLETE problem for which there exists no known algorithm faster than $\mathcal{O}^*(2^n)$ where n is the number of variables. However, fast "in practice" SAT solvers have been developed that are used in a wide range of domains. We give an introduction into how these solvers function and contrast their performance with the conditional lower bounds that can be shown via hypotheses about the complexity of SAT. We detail the sparsification lemma, a critical technique used to show these lower bounds. Finally, we cover observations about the structure of SAT that modern solvers exploit to achieve speeds faster than what worst case complexity would suggest.

Keywords: SAT, SAT Solvers, Conditional Lower Bounds, Subexponential Time, Phase Transition, Backdoor sets, Sparsification Lemma

Table of contents

List of figures

List of tables

Nomenclature

Other Symbols

F Boolean Formula

F $\approx$ G Boolean formulas F and G are equisatisfiable

$\mathcal{O}^*(\cdot)$ Big-O ignoring polynomial factors

Acronyms / Abbreviations

CDCL Conflict Driven CLause Learning

CNF Conjunctive Normal Form

DNF Disjunctive Normal Form

DP Davis Putnam procedure

DPLL Davis Putnam Logemann Loveland

ETH Exponential Time Hypothesis

FPT Fixted Parameter Tractable

k-SAT Boolean Satisfiability Problem with each clause restricted to have size k or less

LBD Literal Block Distance

SAT Boolean Satisfiability Problem

SETH Strong Exponential Time Hypothesis

UIP Unit Implication Point

Chapter 1

Introduction

1.1 Boolean Satisfiability Problem

The boolean satisfiability problem, abbreviated as SAT, is the problem of finding a satisfying assignment to the variables to make a boolean formula evaluate to true. CNF-SAT is the same problem, but with formulas restricted to Conjunctive Normal Form (CNF). A boolean formula in CNF is composed of a conjunction of clauses where each clause is a disjunction of literals. For example consider the following boolean formula in CNF:

$$(\neg x \vee y \vee \neg z) \wedge (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \tag{1.1}$$

This boolean formula has a satisfying assignment $x = \text{false}, y = \text{true}, z = \text{false}$. Therefore the boolean formula is satisfiable. However, if we consider the boolean formula

$$\begin{aligned} &(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \\ \wedge &(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \\ &\wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \end{aligned} \tag{1.2}$$

It's clear to see that this boolean formula can have no satisfying assignment, since each assignment to the variables will at least leave one clause unsatisfied.

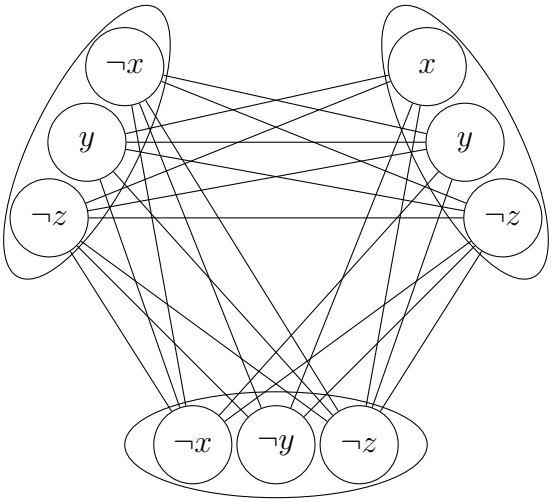


Fig. 1.1 Illustration of the 3SAT in Equation 1.1. Vertices are grouped by their clauses and the edges represent literals that do not contradict. This establishes a reduction from 3SAT to clique

1.2 Uses of Boolean Satisfiability

1.2.1 Theoretical Significance

SAT is a widely studied problem and has had a significant impact on our understanding of computational complexity. For instance, SAT was the first problem shown to be NP-HARD. This then forms the basis of the NP-COMPLETE complexity class [17, 41]. This then means that a large class of problems all can be reduced to each other [26]. SAT has then in some sense become the defacto NP-COMPLETE problem and our understanding of NP-COMPLETEness is driven by our understanding of SAT.

Furthermore, study of SAT has lead to new conjectures about the complexity of SAT that are stronger than the $P \neq NP$ conjecture, roughly stating that SAT requires exponential time [39]. These conjectures allow for new conditional lower bounds on a wide variety of other problems, not strictly limited to NP-COMPLETE problems [75, 52]. This gives us an understanding of the relative difficulty of finding improved algorithms for many computational problems. For instance, we know that finding a $\mathcal{O}(n^{2-\epsilon})$ time algorithm for string edit distance would violate some of these conjectures [75, 48]. Hence making an improvement over an $\mathcal{O}(n^2)$ algorithm should be seen as being as hard as discovering an improved SAT algorithm.

1.2.2 Practical Applications of Boolean Satisfiability

Although, even with these conjectures about the hardness of SAT, as SAT solvers have become more efficient, SAT solving has found numerous applications [55, 46]. One of these applications is model checking, where the use SAT allows for better memory usage than traditional methods and hence larger models [8]. Others include, but are not limited to, crosstalk noise prediction in integrated circuits [15], termination analysis [25], design debugging [71], AI planning [43], haplotype inference in bioinformatics [53], knowledge-compilation [19], software testing [44], package management in software distributions [73], checking of pedigree consistency [54], Test-pattern generation in digital systems [47] and circuit delay computation [58].

Naturally, as the computational power available to SAT solvers grows and as the solvers become more efficient, SAT solving will find more practical applications.

1.3 Organisation and Motivation For This Report

There is a juxtaposition of theory and practice where conjectures about the computational complexity of SAT suggest that we should not expect it to be solvable in subexponential time [39], and yet there are highly optimised SAT solvers that are "fast" in practice. Some of these solvers are able to handle SAT instances having on the order of 10^5 variables and 10^6 clauses [6, 35]. This report will attempt to provide an exposition on this gap between theory and practice.

First this will be done by providing an overview of how modern SAT solvers work, by both covering the algorithms that these solvers are based on and also some of the implemented solvers that use these algorithms.

To contrast this, the next two chapters will cover the complexity conjectures that are based on the hardness of SAT and will go over some conditional lower bounds that can be proven using these conjectures. This report also decides to dedicate a chapter to a commonly used lemma in these proofs called the sparsification lemma. This lemma is instrumental in many conditional lower bounds that apply to polynomial time algorithms such as string edit distance, hence this chapter will explain and go over the proof of the lemma, providing some new intuitions along the way.

Finally, the last chapter will cover some research into structures in SAT instances that can be used to either explain why some SAT instances are easy to solve compared to what the worst case complexity of SAT solving would suggest, or provide motivation for currently employed heuristics in modern SAT solvers.

Taken as a whole this report will provide the reader with a foundational understanding of both how practical SAT solvers work and how they are able to solve SAT instances that are significantly larger than what we might expect were possible given what worst case complexity suggests. This report attempts to be accessible to anyone with an undergraduate level understanding of computational complexity and algorithms who is interesting in both the practical and theoretical side of SAT.

1.4 Notation

A brief mention of notation before moving on, unless otherwise specified a SAT instance will be in Conjunctive Normal Form (CNF), hence we use the term SAT informally to refer to CNF-SAT. This means that a formula is a conjunction of clauses where each clause is a disjunction of literals. In terms of representation, it will often be convenient to represent a clause C as a set of literals and a formula F as a set of clauses. Therefore a formula

$$F = (a \vee b) \wedge (\neg a \vee b \vee \neg c) \wedge (b \vee c)$$

will be represented as

$$\{\{a, b\}, \{\neg a, b, \neg c\}, \{b, c\}\}$$

Thus we can denote the number of clauses as $|F|$ and we could express the set of literals as $\bigcup_{C \in F} C$.

Throughout this report, the letters n and m will always denote the number of variables and the number of clauses respectively. These parameters of the formula have notable significance on the running time of algorithms for SAT and variants of SAT. Hence it is often natural to express the complexity of these algorithms as a parameterized complexity with often n as the parameter (Although this is not always the case, see [36]). Often this will give expressions of the form $2^n \cdot (n+m)^{\mathcal{O}(1)}$. However, the polynomial factor is not of much interest and hence we will use the notation $\mathcal{O}^*(\cdot)$ to suppress any polynomial factors. k-SAT is CNF-SAT with the additional restriction that $\forall C \in F : |C| \leq k$. For boolean formulas F and G, we

use the notation $F \approx G$ to mean that F and G are equisatisfiable. Let $vars()$ be a function that returns the set of variables that are used in its argument. So for a formula F, then $vars(F)$ is the set of n variables, for a clause C, then $vars(C)$ is the set of variables that appear in the clause and for a literal l, then $vars(l)$ is simply the variable that corresponds to that literal.

Chapter 2

SAT Solvers

2.1 Backtracking Based Algorithms

Currently there are predominantly two classes of SAT solvers: those based on backtracking search and those based on stochastic local search [35]. This chapter will go over the two different classes, giving the algorithms that they are based on followed by an overview of the details put into implementations of a solver.

2.1.1 Davis–Putnam–Logemann–Loveland

One of the first SAT solving algorithms came from a desire to show the validity of statements made in first order predicate logic. Here, we shall only concern ourselves with the portion that is applicable to SAT formulas. Before introducing Davis Putnam Logemann Loveland (DPLL), we shall consider its predecessor, the Davis-Putnam procedure (DP)[21]. This procedure uses a resolution based approach and is based on 3 rules:

1. (One Literal Clause Rule) If a formula has a clause containing a single unassigned literal, eliminate all clauses that contain this literal, and remove any occurrence of the negated literal from the remaining clauses. This is also referred to as unit propagation or boolean constraint propagation. A clause with only one unassigned literal, is sometimes referred to as a unit clause.

$$(a) \wedge (a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \approx (\neg b \vee c)$$

2. (Pure Literal Rule) If a variable in a formula occurs either only as a literal in its positive form or only as a literal in its negative form. Then eliminate all

clauses containing this literal.

$$(a \vee b) \wedge (a \vee \neg b) \wedge (b \vee c) \approx (b \vee c)$$

3. (Resolution Rule) If there exist two clauses which share a variable where in one clause the literal is negated and in the other it is not. The two clauses can be replaced by a single clause containing the union of their literals excluding the shared variable.

$$(a \vee x) \wedge (b \vee \neg x) \approx (a \vee b)$$

The simplified formulas that result from the application of these rules are not necessarily equivalent to the original formulas, but they are equisatisfiable. Therefore if a formula is simplified to the point it contains no clauses then it is trivially satisfiable and the original formula is also satisfiable. Observe that it is always possible to reduce a satisfiable formula to the empty formula, since if the resolution rule cannot be applied then the pure literal rule can remove all clauses.

To see this procedure in action consider checking whether the following formula in CNF is satisfiable.

$$(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee \neg b) \tag{2.1}$$

As we see in Equation 2.1, we can apply the resolution rule to the first two clauses. This yield the following simplification.

$$(b) \wedge (a \vee \neg b) \tag{2.2}$$

From here we see that we can apply the one literal clause rule to eliminate the first clause. Note that the last clause will be simplified to just (a). Hence we can apply the one literal clause rule again to get the empty formula. At this point the procedure finishes. Since we obtained the empty formula we know that the original formula was satisfiable.

If a formula is unsatisfiable then the input formula will eventually simplify to a formula containing the empty clause, which cannot be satisfied, at this point the procedure would halt [21].

Consider Equation 2.1, but with the added clause $(\neg a \vee \neg b)$. This modified formula is now unsatisfiable and if we apply the same rules to this new formula we will, instead of ending up with (a), we will have $(a) \wedge (\neg a)$. At this point, applying the pure literal rule to either a or $\neg a$ will leave one clause empty.

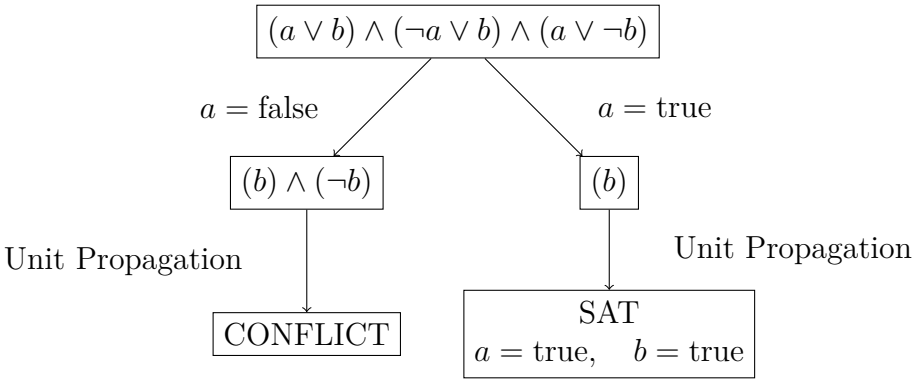


Fig. 2.1 A run of DPLL on Equation 2.1

Moving to DPLL

An issue was that the resolution rule could cause the size of clauses to increase rapidly. This was an issue that prevented the procedure to implemented due to memory constraints. The solution was to replace the resolution rule by the splitting rule which can be stated as the following. Let x be some variable in a formula, the formula is inconsistent if and only if the formula is inconsistent when $x = \text{true}$ and when $x = \text{false}$ [20]. This is theoretically equivalent to the resolution rule, however this rule allowed for better memory usage in practice.

The substitution of the resolution rule for the splitting rule forms the basis for DPLL with most improvements focusing on finding efficient data structures for implementing DPLL and on developing good heuristics for which variable to branch on.

An important observation is that the splitting rule defines a tree with the original formula as the root and the two children of the root are the two formulas obtained from the splitting rule. Hence a formula is satisfiable if and only if one of the leaves in the tree is satisfiable. DPLL moves through the tree in a pre-order like fashion, this is called chronological backtracking [9]. However, note that the order in which the tree is traversed and how the algorithm backtracks does not affect the correctness of the algorithm as long as we can guarantee that every node in the tree will be visited eventually. A run of DPLL on Equation 2.1 can be seen in Figure 2.1.

2.1.2 Conflict Driven Clause Learning

Recent competitive solvers are based on Conflict Driven Clause Learning (CDCL) [6, 35], such as Glucose. CDCL builds on DPLL in a few ways, the key methods are as follows:

- Derive new clauses when search yields a conflict [57].
- Use of efficient lazy data structures [60, 65].
- Non-chronological backtracking [57].
- Restarting search [34, 33]

The structure of a CDCL solver without restarts can be seen in Algorithm 1.

Algorithm 1: Organisation of CDCL [9, 56]

Input: CNF-SAT Formula F
Output: SAT or UNSAT

```
1  α ← ∅
2  if unit_propagation(F, α) == CONFLICT then
3      return UNSAT
4  decision_level ← 0
5  while ¬ all_variables_assigned(F, α) do
6      (x, v) ← pick_branching_variable(F, α)
7      decision_level ← decision_level +1
8      α ← α ∪ {(x, v)}
9      if unit_propagation(F, α) == CONFLICT then
10         β ← conflict_analysis(F, α)
11         if β < 0 then
12             return UNSAT
13         else
14             backtrack(F, α, β)
15             decision_level ← β
16 return SAT
```

In Algorithm 1 α refers to the set of assignments, where each assignment is represented as a tuple (x, v) where x is a variable and $v \in \{\text{false}, \text{true}\}$ is the value. The unit propagation function call executes the one literal clause rule from Section 2.1.1. Pick branching variable chooses which variable to branch on and gives it

a value. Choice of heuristics for which variable to select is motivated mostly by which features can be computed efficiently [49, 60].

Backtrack takes in a parameter β, which indicates the decision level to backtrack to. This means that search happens in a non-chronological fashion. The level to backtrack to is decided by the conflict analysis function.

Conflict Analysis

The most characteristic part of CDCL is the clause learning that results from the clause analysis process. There are slight variants of this procedure, but we will cover the most common variations popularised in GRASP [57] and Chaff [60].

For a particular partial assignment α, we use decision level to mean the order that a given variable was assigned by the `pick_branching_variable` routine. For variables that were assigned via the `unit_propagation` routine, they inherit the highest decision level of the other variables in the clause. We consider a variable that is assigned via `unit_propagation` to be implied by the other variables in the unit clause. Since `unit_propagation` may lead to more clauses becoming unit, we can construct a directed acyclic graph of implications. We call this graph the implication graph.

Consider the following boolean formula:

$$\begin{aligned}&(\neg x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_5) \wedge (\neg x_4 \vee x_5 \vee x_6) \\ &\wedge(\neg x_6 \vee x_7) \wedge (\neg x_3 \vee \neg x_6 \vee \neg x_8) \wedge (\neg x_7 \vee x_8)\end{aligned} \tag{2.3}$$

For example, say that `pick_branching_variable` first assigns $x_1 = \text{true}$ at decision level 1, then $x_3 = \text{true}$ at decision level 2 and then $x_2 = \text{false}$ at decision level 3. At this point, the first two clauses in Equation 2.3 are unit clauses and applying `unit_propagation` will lead to a conflict. The resulting implication graph can be seen in Figure 2.2. Each vertex represents an assignment or a conflict and there is a directed edge from one vertex to another if that assignment implies the other via unit propagation. For example, in Figure 2.2 there is an edge from $x_1 = \text{true}$ and $x_2 = \text{false}$ to $x_4 = \text{true}$ because the first two assignments make the clause $(\neg x_1 \vee x_2 \vee x_4)$ a unit clause which implies $x_4 = \text{true}$. There are also edges from $x_7 = \text{true}$ and $x_8 = \text{false}$ to a conflict vertex since these two assignments leave the clause $(\neg x_7 \vee x_8)$ unsatisfied. Each edge is labelled with the clause that caused the implication.

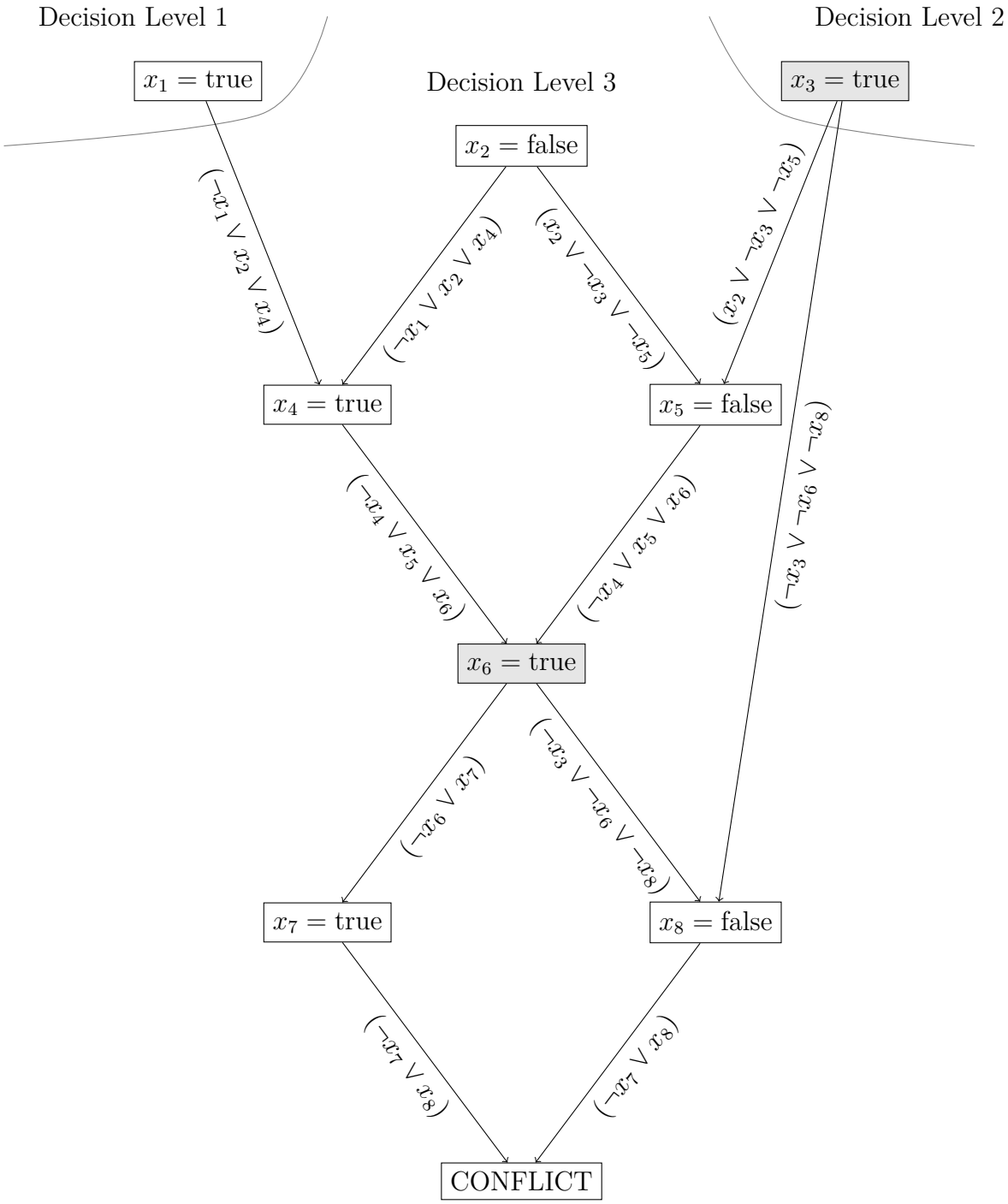


Fig. 2.2 Implication graph for Equation 2.3 for a partial assignment to x_1, x_2, x_3. First UIP is shaded grey

Step	Resolution
1	$(\neg x_7 \vee x_8) \odot (\neg x_6 \vee x_7) = (\neg x_6 \vee x_8)$
2	$(\neg x_6 \vee x_8) \odot (\neg x_3 \vee \neg x_6 \vee \neg x_8) = (\neg x_3 \vee \neg x_6)$
3	$(\neg x_3 \vee \neg x_6) \odot (\neg x_4 \vee x_5 \vee x_6) = (\neg x_3 \vee \neg x_4 \vee x_5)$
4	$(\neg x_3 \vee \neg x_4 \vee x_5) \odot (x_2 \vee \neg x_3 \vee \neg x_5) = (x_2 \vee \neg x_3 \vee \neg x_4)$
5	$(x_2 \vee \neg x_3 \vee \neg x_4) \odot (\neg x_1 \vee x_2 \vee x_4) = (\neg x_1 \vee x_2 \vee \neg x_3)$

Table 2.1 Resolution steps for Equation 2.3 with partial assignments to x_1, x_2, x_3

The `conflict_analysis` routine will construct this implication graph and then apply a number of resolution steps to obtain the learned clause. Define a resolution operator $\odot$ for two clauses C_1 and C_2 with a unique variable x_r such that $\neg x_r \in C_1$ and $x_r \in C_2$ or vice versa, then $C_1 \odot C_2 = (C_1 \cup C_2) \setminus \{x_r, \neg x_r\}$ i.e. a clause containing all literals in C_1 and C_2 excluding x_r and $\neg x_r$. `conflict_analysis` then begins by taking the clause that caused the conflict (in this example $(\neg x_7 \vee x_8)$) and applying the resolution operator with this clause and a clause that implied one of the literals in the first clause. In our example the first resolution step could be either $(\neg x_7 \vee x_8) \odot (\neg x_6 \vee x_7) = (\neg x_6 \vee x_8)$ or $(\neg x_7 \vee x_8) \odot (\neg x_3 \vee \neg x_6 \neg x_8) = (\neg x_3 \vee \neg x_6 \vee \neg x_7)$. A full set of resolution steps using the example from Equation 2.3 can be seen in Table 2.1. Since $\forall C_1, C_2 \in F : C_1 \odot C_2 \approx C_1 \wedge C_2$ then $F \approx F \cup \{C_1 \odot C_2\}$. This means that we can take any clause from Table 2.1 and add it to Equation 2.3 without changing its satisfiability.

There are different options for how many resolution steps to perform, however, a guiding principle is that the learned clause should be as short as possible and should remain a unit clause after backtracking. The most common approach is to stop at what is called the first Unit Implication Point (UIP) [9], which is the first point at which there is only one literal in a learned clause that has the highest decision level. In Table 2.1 this occurs at step 2, since in the clause $(\neg x_3 \vee \neg x_6)$ only x_6 has decision level 3 while x_3 has decision level 2. Note that if we look at the literals in a clause obtained by any number of resolution steps, the corresponding vertices in the implication graph form a vertex separator, separating the side of the graph containing the conflict vertex and the side containing the variables in the assignments made by `pick_branching_variable`. Therefore, we can interpret a UIP as a clause where the variable with the highest decision level is a dominator for the variable most recently assigned by `pick_branching_variable` with respect to the conflict vertex. This can be seen in Figure 2.2 as all paths from $x_2 = \text{false}$ to the conflict vertex must pass through $x_6 = \text{true}$.

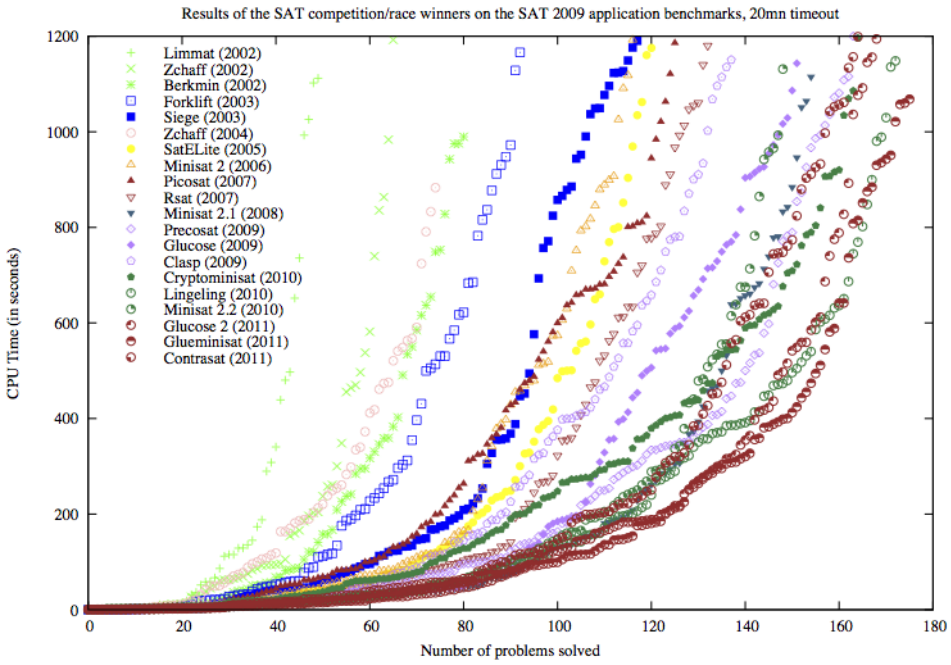


Fig. 2.3 Number of sat problems taken from the 2009 SAT competition that were solved by various solvers with a 20 minute timeout [46]

The amount to backtrack by (given by β in Algorithm 1) also varies from solver to solver and most often depends on how inexpensive backtracking is given the datastructures used by the solver. However, most modern solvers now use lazy datastructures which make it cheap to backtrack. For these solvers a common approach is to backtrack until just before the learned clause no longer is a unit clause. In the example given in Equation 2.3, the learned clause was $(\neg x_3 \vee \neg x_6)$. As long as x_3 remains assigned then the learned clause will be a unit clause, and x_3 was assigned at decision level 2, so we can backtrack to decision level 2. In general we can backtrack to the second highest decision level in the learned clause. This is the backtracking method used by Chaff [60]. We can guarantee completeness because, since the learned clauses prevent the solver from exploring the same assignment twice.

Performance

In figure 2.3 we can see the number of solved instances of practical sat benchmarks submitted to the 2009 SAT competition by various different solvers [46]. Notably more recent solvers, whilst still demonstrating exponential scaling behaviour are able to solve significantly more problems than prior solvers.

For the 2018 SAT competition the benchmarks were taken from various different domains: [35]

- Proving the existence of a nonce for blockchain mining algorithms.
- Searching for a unit-distance graph with chromatic number 6 and k-colourability of graphs.
- Verifying simple floating-point programs.
- Cryptanalysis.
- Time-Slot allocation problems with optimal preference satisfaction

2.1.3 Glucose

Glucose is an efficient implementation of CDCL that has featured prominently in many SAT competitions [2] building on top of MiniSAT [23]. Glucose implements a few additional features on top of CDCL, most prominently it implements new clause deletion policies which help control the memory usage of the learned clauses generated by the conflict analysis. These deletion policies expand on those used in the BerkMin solver [32], going beyond analysing clauses just by size and how close they are to being a unit clause [22, 7].

These policies were established in work by Audemard et al. where it is observed that the decision level at which conflicts are encountered tends to decrease as the solver progresses [3], although this effect was only observed for "industrial" SAT instances. This means that the solver is able to find conflicts faster over time, i.e. it is assigning many variables via unit propagation. Additionally, this allows for a crude estimate of when the solver will terminate. Audemard et al. then devise a metric called the "Literal Block Distance" (LBD) which is the number of distinct decision levels in a clause for a given assignment and show that when keeping only clauses with low LBD the observation that decision levels decrease still holds. This suggests that these clauses are the most important for allowing the solver to spend significant time executing unit propagation. Applying this deletion policy, around 93% of learned clauses were deleted [2]. However, since the learned clauses are needed for guaranteeing completeness, Glucose is not complete.

Of practical note is that the Glucose solver is particularly amenable to paralellisation and remains one of the most competitive parallel solvers [4].

2.2 Stochastic Local Search Algorithms

A different family of algorithms to backtracking algorithms is local search algorithms which are based on the idea of randomly choosing an assignment and then making small adjustments to assignment in order to try and find a satisfying assignment. We will cover an important algorithm, namely Schöning's algorithm [67], and then consider WalkSat, a solver based on similar principles.

2.2.1 Schöning's Algorithm

An interesting k-SAT algorithm that is able to beat the $\mathcal{O}^*(2^n)$ complexity for bounded k is Schöning's algorithm seen in Algorithm 2. Given that a satisfying assignment exists it will be found in expectation after $(2 - \frac{2}{k})^n$ runs of Schöning's algorithm [67]. Hence the expected running time is $\mathcal{O}^*((2 - \frac{2}{k})^n)$.

Algorithm 2: Schöning's Algorithm for k-SAT

Input: k-SAT instance, F
Output: Instance is satisfiable or not

```
1 Guess initial assignment
2 for i ∈ {1, ..., 3n} do
3     if F is Satisfied then
4         return Accept
5     else
6         flip a random variable in an unsatisfied clause
```

Analysis of Schöning's Algorithm

For the sake of brevity, we will only consider the case where $k = 3$. Since we require that there is at least one satisfying assignment let α^* be one such satisfying assignment. Let α be the assignment guessed in the first line and let d be the Hamming distance between α and α^*, i.e. $H(\alpha, \alpha^*) = d$ the number of variables in which they differ. The probability of guessing an assignment α which differs by exactly d variables from α^* is given by the following expression, where n is the number of variables.

$$2^{-n}\binom{n}{d} \tag{2.4}$$

If the algorithm flips a variable in which α and α^* differ, then we call this a step towards the solution, i.e. $H(\alpha, \alpha^*)$ decreases by 1. Conversely, if they do not differ then we call this a step away from the solution, i.e. $H(\alpha, \alpha^*)$ increases by 1.

Consider the probability of making $2d$ steps towards the solution and d steps away in the first $3d$ steps. It is clear that this will reach α^* and since $d \leq n$, then the total number of steps will not be greater than $3n$. Since, in an unsatisfied clause, at least one variable is different between α and α^*, then when the algorithm flips a random variable in that clause the probability of it being one where the assignments are different is $\frac{1}{3}$. Hence, the probability of making $2d$ toward the solution and d steps away in the first $3d$ steps is given by the following expression.

$$\binom{3d}{d}\left(\frac{1}{3}\right)^{2d}\left(\frac{2}{3}\right)^{d} \tag{2.5}$$

Hence to find the probability that the algorithm finds a satisfying solution we simply have combine this with the probability of guessing a first assignment that has Hamming distance of exactly d, and sum over all possible values of d.

$$\Pr[\text{Find satisfying assignment}] = \sum_{d=0}^{n} 2^{-n}\binom{n}{d}\binom{3d}{d}\left(\frac{1}{3}\right)^{2d}\left(\frac{2}{3}\right)^{d} \tag{2.6}$$

To simplify Equation 2.6 it will be helpful to consider a lower bound of $\binom{3d}{d}$. To do this we will use Stirling's approximation of the factorial:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^{n} \tag{2.7}$$

Using Equation 2.7 we can derive an approximation as follows.

$$\begin{aligned}\binom{3d}{d} &= \frac{(3d)!}{(2d)! \cdot d!} \\ &\sim \frac{\sqrt{6\pi d}\left(\frac{3d}{e}\right)^{3d}}{\sqrt{4\pi d}\left(\frac{2d}{e}\right)^{2d}\sqrt{2\pi d}\left(\frac{d}{e}\right)^{d}} \\ &\sim \frac{\sqrt{3}}{2\sqrt{\pi d}} \cdot \frac{3^{3d}}{2^{2d}}\end{aligned}$$

Since $n \geq d$ we replace $\sqrt{d}$ with $\sqrt{n}$ to simplify the analysis. Then since the binomial coefficient is always strictly positive then what remains is to choose a sufficiently large constant δ such that we have for $d \geq 0$:

$$\binom{3d}{d} \geq \frac{1}{\delta\sqrt{n}} \cdot \frac{3^{3d}}{2^{2d}} \tag{2.8}$$

Combining Equations 2.6 and 2.8 we get a lower bound on the probability of finding a satisfying assignment.

$$\begin{aligned}
\Pr[\text{Find satisfying assignment}] &\geq \sum_{d=0}^{n} 2^{-n} \binom{n}{d} \frac{1}{\delta\sqrt{n}} \cdot \frac{3^{3d}}{2^{2d}} \left(\frac{1}{3}\right)^{2d} \left(\frac{2}{3}\right)^{d} \\
&\geq \frac{2^{-n}}{\delta\sqrt{n}} \sum_{d=0}^{n} \binom{n}{d} \frac{1}{2^d} \\
&\geq \frac{2^{-n}}{\delta\sqrt{n}} \left(1 + \frac{1}{2}\right)^{n} \\
&\geq \frac{1}{\delta\sqrt{n}} \left(\frac{3}{4}\right)^{n}
\end{aligned}$$

Repeatedly running Algorithm 2 defines a geometric sequence. Therefore, the expectation of the number of runs required to find the satisfying assignment is given by

$$\mathbb{E}[\#\text{runs}] = \frac{1}{\Pr[\text{Find satisfying assignment}]} \leq \delta\sqrt{n}\left(\frac{4}{3}\right)^{n} \tag{2.9}$$

Hence the time complexity can be written as $\mathcal{O}^*(1.334^n)$.

2.2.2 WalkSat

WalkSat is an solver similar to Schöning's algorithm that is inspired by GSAT [70, 68, 69] and by ideas laid forth by Papadimitriou [62]. The main difference between WalkSat and Schöning's algorithm is that it employs a semi-greedy strategy when deciding which variable to flip in an unsatisfied clause.

With some probability p WalkSat will flip a random variable, but with probability $1 - p$ WalkSat will flip the variable that minimises the number of clauses that become unsatisfied. The parameter p is often called the noise parameter and optimal choice of p depends on the structure of the formula being tested. Therefore, modern improvements have attempted to adapt p automatically to each instance [38].

Formula #Vars	#Clauses	DP Time	GSAT Time	WalkSat Time
708	1702	*	0.081	0.013
649	1562	*	0.058	0.014
8704	32316	*	94.1	1.0
8432	31310	*	456.6	0.7
300	730	23096	0.009	0.002
125	310	1.4	0.009	0.001

Table 2.2 Comparing solution times of an implementation of the Davis Putnam procedure (DP) with GSAT and WalkSat (time in seconds) [69]

WalkSat is an efficient solver that is able to tackle SAT instances with a large number of variables and clauses, and is able to find solutions to instances where an efficient implementation of the Davis Putnam procedure was unable to a find a solution. An overview of solution times for a few SAT benchmark problems from 1993 is seen in Table 2.2.

Chapter 3

Complexity Conjectures

3.1 Exponential Time Hypothesis

Of all the algorithms covered in the previous chapter, they all had exponential computational time complexity, with all of the backtracking based algorithms requiring $\mathcal{O}^*(2^n)$ in the worst case. Slightly better was Schöning's Algorithm 2, which had a complexity of $\mathcal{O}^*((2-\frac{2}{k})^n)$. However, for a general formula in CNF k could be arbitrarily large and as $k \to \infty$ then we recover the same complexity as the other algorithms. So even when considering the case where we have a 3SAT instance, we are unable to improve over the exponential complexity.

We know that k-SAT is NP-COMPLETE for $k \geq 3$[66], so unless $P = NP$ we should not expect to find a polynomial time algorithm. However, it is still theoretically possible, under the assumption that $P \neq NP$, for there to exist an algorithm which is superpolynomial but subexponential. Here we say that $f(x)$ is subexponential if it is $\mathcal{O}^*(2^{\epsilon n})$ for all $\epsilon > 0, \quad \epsilon \in \mathbb{R}$ with $\mathcal{O}^*(g(x))$ meaning $\mathcal{O}(poly(n) \cdot g(x))$, i.e. ignoring polynomial factors. Such an algorithm could have running time $T(n) = \mathcal{O}^*(2^{\frac{n}{\log n}})$ which is $\mathcal{O}^*(2^{\epsilon n}), \quad \epsilon > 0$. However, no such algorithm for SAT has been found.

Due to the seeming difficulty of finding significantly faster k-SAT algorithms there have been two conjectures put forward by Impagliazzo and Paturi, which relate to the complexity of k-SAT. These are the "Exponential Time Hypothesis" and the "Strong Exponential Time Hypothesis", abbreviated to ETH and SETH respectively [39].

Informally ETH hypothesises that there does not exist a subexponential time algorithm solving SAT. More formally, consider the set of all k-SAT algorithms and express their time complexities in the form $\mathcal{O}^*(2^{\delta n})$ for some constant $\delta \in \mathbb{R}^+ \cup \{0\}$.

Then for $k \in \mathbb{N}, \quad k \geq 3$ let s_k be the infimum of δs taken from this set

$$s_k = \inf\{\delta : k\text{-SAT is solvable in time } \mathcal{O}^*(2^{\delta n})\} \tag{3.1}$$

The ETH then states that $s_3 > 0$ [39]. Intuitively this means that there must exist some non-zero constant s_3 such that solving 3-SAT takes time $\Omega^*(2^{s_3 n})$ which rules out a subexponential time algorithm for 3-SAT. The statement that $s_3 > 0$ is equivalent to the statement that for all $k \in \mathbb{N}$ if $k \geq 3$ then $s_k > 0$, i.e. there then cannot exist a subexponential time algorithm for k-SAT with $k \geq 3$. This is because a 3-CNF can be easily reduced to a k-CNF for all $k \geq 3$.

If we, instead of considering all k-SAT algorithms, consider just Schöning's algorithm, then we get a new sequence of s'_k where $\forall k \in \mathbb{N} : s'_k \geq s_k$. Rearranging the complexity of Schöning's algorithm we get that

$$s'_k = \log_2(2 - \frac{2}{k}), \quad k \geq 3 \tag{3.2}$$

which yields the sequence $\sim 0.415, \sim 0.585, \sim 0.678, \ldots$ for $k = 3, 4, 5, \ldots$. It is clear from Equation 3.2 that as $k \to \infty$ then $s'_k \to 1$. There are known algorithms that produce smaller constants [37], however these algorithms still generate a sequence that tends to 1 as $k \to \infty$.

The "Strong Exponential Time Hypothesis" (SETH) considers this limit as k tends to infinity. SETH states that

$$\lim_{k \to \infty} s_k = 1 \tag{3.3}$$

Intuitively, if SETH were proven true, this would mean that for general SAT there is no algorithm that is asymptotically faster than brute force search and algorithms such as DP, DPLL and CDCL would be optimal up to subexponential factors.

3.1.1 Conditional Lower Bounds

The ETH and SETH both imply that $P \neq NP$, so we should not hope to prove either conjecture just yet. However, we can consider the implications that these conjectures would have on other problems if they were proven true. This allows for us to show that many problems have conditional lower bounds on their complexity and for many problems that current algorithms are optimal assuming one of these conjectures.

If we consider attempting to show lower bounds from an assumption of the ETH, then since the ETH deals with the non-existence of subexponential time algorithms for k-SAT, we could hope to reduce k-SAT to a number of different problems in such a way that preserves subexponential time.

Since we are dealing with parameterized complexities for k-SAT with parameters n and m, when considering general problems we will use the mapping $\kappa : \Sigma^* \mapsto \mathbb{N}$ to denote the parameter an instance of the problem $x \in \Sigma^*$. Where Σ^* is the set of problem instances.

Definition A Turing reduction from a problem (A_1, κ_1) to a problem (A_2, κ_2) is considered a SERF-T reduction if

1. The reduction on an instance x of A_1 runs in time $\mathcal{O}(2^{\epsilon\kappa_1(x)}|x|^{\mathcal{O}(1)})$ for a choice of $\epsilon > 0$.

2. For a query to A_2 with input x':

   (a) $|x'| \leq |x|^{\mathcal{O}(1)}$

   (b) $\kappa_2(x') \leq \alpha\kappa_1(x)$

   Where the constants hidden in the $\mathcal{O}(\cdot)$ do not depend on the choice of ϵ. The constant α may depend on ϵ

To see how this works, consider two parameterized problems (A_1, κ_1) and (A_2, κ_2), where the second problem has a parameterized subexponential time algorithm. We wish to show that if (A_1, κ_1) is SERF-T reducible to (A_2, κ_2) then there also exists a parameterized subexponential time algorithm for (A_1, κ_1), i.e. for a choice of $\epsilon > 0$ we need to show that (A_1, κ_1) runs in time $\mathcal{O}(2^{\epsilon\kappa_1(x)}|x|^{\mathcal{O}(1)})$.

Do show this, choose an $\epsilon > 0$. Let $\epsilon' = \frac{\epsilon}{2}$ and run the SERF-T reduction with parameter ϵ'. Since this reduction runs in time $\mathcal{O}(2^{\epsilon'\kappa_1(x)}|x|^{\mathcal{O}(1)})$ then this bounds the number of calls to (A_2, κ_2) by the same amount. Each call to (A_2, κ_2) has an instance $|x'| \leq |x|^{\mathcal{O}(1)}$ and $\kappa_2(x') \leq \alpha\kappa_1(x)$. Since, from our assumption that (A_2, κ_2) runs in parameterized subexponential time we can choose $\epsilon'' = \frac{\epsilon'}{\alpha}$ and then each call to (A_2, κ_2) can be made to run in time

$$\mathcal{O}(2^{\epsilon'\kappa_1(x)}|x|^{\mathcal{O}(1)}) \tag{3.4}$$

Therefore, the total time for solving (A_1, κ_1) is given by

$$\mathcal{O}(2^{\epsilon'\kappa_1(x)}|x|^{\mathcal{O}(1)}) \cdot \mathcal{O}(2^{\epsilon'\kappa_1(x)}|x|^{\mathcal{O}(1)}) = \mathcal{O}(2^{\epsilon\kappa_1(x)}|x|^{\mathcal{O}(1)}) \tag{3.5}$$

which is what we wanted to show. [52]

Lower Bounds for 3-colouring

To show that ETH implies that there is no subexponential time algorithm for 3-colourability we must show that the standard reduction from 3-SAT to 3-colourability is also a SERF-T reduction. The first requirement that the reduction runs in time $\mathcal{O}(2^{\epsilon\kappa_1(x)}|x|^{\mathcal{O}(1)})$ for all $\epsilon > 0$ is trivially satisfied since the reduction runs in polynomial time. Hence it also follows that the reduced instance $|x'| \leq |x|^{\mathcal{O}(1)}$.

Thus, the only thing left to show is that $\kappa_2(x') \leq \alpha\kappa_1(x)$ for some constant α. In this case $\kappa_2(x')$ would be the number of vertices in the reduced instance and $\kappa_1(x) = n$.

To do this, recall the the standard reduction reduction from 3-SAT to 3-colourability (see Figure 3.1). We have a triangle and label the vertices as True, False and Base. For each variable x_i we create vertices labelled x_i and $\neg x_i$ and edges $(x_i, \neg x_i), (x_i, \text{Base}), (\neg x_i, \text{Base})$. Then for each clause we have two "or" gadgets that consist of 3 vertices. Thus, the total number of vertices in the instance x' is given by

$$\kappa_2(x') = 2n + 3m + 3 \tag{3.6}$$

This appears to be an issue, since we are unable to make $\kappa_2(x')$ linear in n since m could be a superlinear function of n for an arbitrary instance of 3-SAT x. However, we can make use of the sparsification lemma [40] to reduce any k-SAT instance into a subexponential number of k-SAT instances where m is linear in n and this takes subexponential time. We will detail the sparsification lemma in Chapter 4.

Hence, to get our SERF-T reduction we first apply the sparsification lemma to reduce our original instance x into subexponentially many new k-SAT instances and apply our standard 3-SAT to 3-colourability reduction to each new instance. Since we can now guarantee that m is $\mathcal{O}(n)$ then $2n + 3m + 3$ is also $\mathcal{O}(n)$. We let α be the constant hidden in the $\mathcal{O}(\cdot)$. Since the sparsification lemma produces a subexponential number of instances that are no longer than the original instance and since it runs in subexponential time, then we do not violate the time constraints of the SERF-T reduction.

Hence we have shown that this reduction satisfies all the requirements of a SERF-T requirement. We can therefore say that if 3-colourability on $|V|$ vertices can be solved in time $\mathcal{O}^*(2^{\epsilon\cdot|V|})$ for all $\epsilon > 0$ then 3-SAT could be solved in time $\mathcal{O}^*(2^{\epsilon\cdot n})$ for all $\epsilon > 0$, which would violate the ETH.

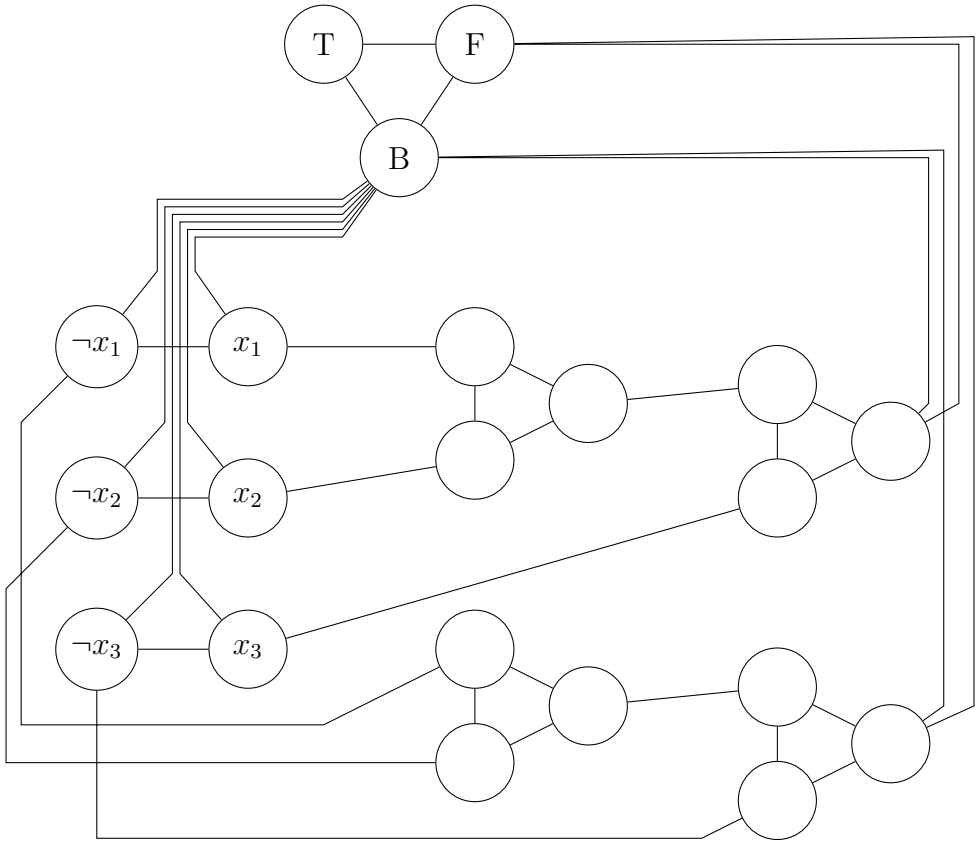


Fig. 3.1 Standard reduction of $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$ to 3-colourability

Using similar techniques we can also establish that the ETH implies that there are no subexponential time algorithms for the following [52].

- Independent Set
- Dominating Set
- Vertex Cover
- Hamiltonian Path

Lower Bounds for Planar Graph Problems

Note that the SERF-T reduction from 3-SAT to 3-colourability depended on the fact that the number of vertices in the 3-colourability instance was linear in the in the number of variables in the 3-SAT instance. Typically for planar graph problems, this property cannot be obtained.

For instance the reduction from 3-SAT to planar hamiltonian cycle creates a graph G where the number of vertices is $\mathcal{O}(m^2)$ [27]. As such we cannot establish a SERF-T reduction from 3-SAT to planar hamiltonian cycle parameterized by the number of vertices. However, if we let the parameter be $\kappa_2(x') = \sqrt{|V|}$, then it is obvious that this parameter is $\mathcal{O}(m)$. Then we can use the same techniques as before to establish that the ETH implies that there is no $2^{o(\sqrt{|V|})}$ algorithm for planar hamiltonian cycle.

The same argument can be made for many planar graph problems. Taken in conjunction with algorithms using the planar separator theorem [51] this then means that many planar graph algorithms are optimal [50].

3.1.2 Further Conditional Lower Bounds for Algorithms in P

Proofs of conditional lower bounds are not limited to problems that have super-polynomial complexities. Here we will give an example of a lower bound for the orthogonal vectors problem based on the assumption of the SETH [76].

Definition The orthogonal vectors problem is, given two sets U and V of bit vectors of length d where $|U| = |V| = n$ and d is $\omega(\log n)$, does there exist $u \in U$ and $v \in V$ such that $\sum_{i=1}^{d} u_i \cdot v_i = 0$

Best known algorithms for orthogonal vectors take time $\mathcal{O}(n^{2-o(1)})$ and it is unknown whether there is any "strongly subquadratic" algorithm, i.e. an algorithm taking time $\mathcal{O}(n^{2-\epsilon})$ for some $\epsilon > 0$. However, Williams et al. show that SETH implies that no such algorithm exists [76].

To show this we will need the concept a fine grained reduction. Such a reduction has the property that for two problems A and B with best known running times $a(n)$ and $b(n)$ respectively, an improvement on B to $\mathcal{O}(b(n)^{1-\epsilon})$ leads to an improvement on A to $\mathcal{O}(a(n)^{1-\delta})$ for all $\epsilon > 0$ and some $\delta > 0$. Such a reduction is often denoted as a $(a(n), b(n))$-reduction [75]

Definition A fine grained reduction M from problem A to problem B with running times $a(n)$ and $b(n)$ respectively obeys the following: For all $\epsilon > 0$, there is some $\delta > 0$ and some constant d such that for all $n \geq 1$ there is some constant k_n and

- M runs in time $d \cdot (a(n))^{1-\delta}$,
- M produces at most k_n instances of B adaptively,
- $\sum_{i=1}^{k_n}(b(n'_i)^{1-\epsilon}) \leq d \cdot (a(n))^{1-\delta}$, where n'_i is the i_{th} instance of B.

Instance sizes n'_i may depend on n and ϵ, but d can depend only on ϵ and not on n.

The $(2^n, n^2)$-reduction from k-SAT to orthogonal vectors is then as follows. Apply the sparsification lemma to the k-SAT instance, then partition the variables arbitrarily into two sets D_1, D_2 such that each set contains $\frac{n}{2}$ variables. Construct two sets of vectors U_1, U_2 as follows: For all $i \in \{1, 2\}$ and for all assignments α to the variables in D_i, $v_{i,\alpha} \in U_i$ where for all clauses C, $v_{i,\alpha}[C] = 0$ if and only if the clause C is satisfied by the assignment α to the variables in D_i, 1 otherwise. Therefore, if two vectors are orthogonal then there exists an assignment to the variables in D_1 and D_2 such that for every clause it is either satisfied by the assignment to D_1 or the assignment to D_2 [76].

Hence, any $\mathcal{O}(n^{2-\epsilon})$ time algorithm for orthogonal vectors implies that there is a $\mathcal{O}(2^{(1-\delta)n})$ algorithm for k-SAT for all k. Therefore, the SETH implies that there cannot be such an algorithm for orthogonal vectors.

Applying similar techniques, SETH implies

- Fréchet distance cannot be computed in $\mathcal{O}(n^{2-\epsilon})$ [11]
- Edit distance cannot be computed in $\mathcal{O}(n^{2-\epsilon})$ [5]
- Dynamic Time Warping distance cannot be computed in $\mathcal{O}(n^{2-\epsilon})$ [12]
- Longest Common Subsequence cannot be computed in $\mathcal{O}(n^{2-\epsilon})$ [13, 63]

Chapter 4

Sparsification Lemma

Introduction

In this chapter we will explain the sparsification lemma that allowed us to show lower bounds on problems in P conditioned on the SETH in Chapter 3. To do this we will first consider an example problem which again motivates the need for the sparsification lemma. The example is the following. Recall the definition of s_k from Equation 3.1.

Theorem 4.0.1. *If there exists a $k \geq 3$ such that $s_k > 0$, then $s_3 > 0$ [39].*

To show this, we would have to prove that if there are no subexponential time algorithms for k-SAT for some $k > 3$ then there is no subexponential algorithm for 3-SAT. To make things simpler, we will show the contrapositive, that if there is a subexponential time algorithm for 3-SAT then there is a subexponential time algorithm for k-SAT for all $k \geq 3$.

Assume then that we have an algorithm A that solves 3-SAT in subexponential time i.e. it takes $\mathcal{O}^*(2^{\epsilon n})$ time and we want to solve an instance of k-SAT where $k > 3$. We can attempt to solve this instance via a reduction to 3-SAT. To reduce k-SAT to 3-SAT consider a k-CNF formula F_k and consider a specific clause $C = (x_1 \vee x_2 \vee \ldots \vee x_k)$ from F_k. We will then introduce new variables $y_1, y_2, \ldots, y_{k-3}$ and define $k-2$ new clauses as follows:

$$
\begin{aligned}
C'_1 &= (x_1 \vee x_2 \vee y_1) \\
C'_2 &= (\overline{y_1} \vee x_3 \vee y_2) \\
C'_3 &= (\overline{y_2} \vee x_4 \vee y_3) \\
&\vdots \\
C'_{k-2} &= (\overline{y_{k-3}} \vee x_{k-1} \vee x_k)
\end{aligned}
$$

Observe that the conjunction of the clauses $C'_1, C'_2, \ldots, C'_{k-2}$ is satisfiable if and only if the original clause C is satisfiable. This procedure can then be repeated for all clauses in F_k to convert the k-SAT formula with n variables and m clauses to a 3-SAT formula with $n + (k-3)m$ variables and $(k-2)m$ clauses. If we now use algorithm A, then we solve the instance in time $\mathcal{O}^*(2^{\epsilon \cdot (n+(k-3)m)})$. This is an issue, if we consider the fact that for an arbitrary k-SAT formula the number of clauses could be significantly larger than the number of variables. Consider the case where $m = n^2$. This then means that A would run in time $2^{\mathcal{O}(n^2)}$ which is certainly not subexponential.

Fundamentally, the issue is that the reduction introduced too many new variables and these new variables came from the number of clauses in the original formula. If we could guarantee that the number of clauses in original formula was linear in the number of variables, i.e. $m = \mathcal{O}(n)$, then the reduction would work.

4.1 Statement of the lemma

The sparsification lemma as stated by Impagliazzo, Paturi and Zane originally relates to k-Set Cover, from which they derive the k-SAT version as a corollary [40]. The original formulation is stated here for completeness. Note that although this problem is called k-Set Cover by Impagliazzo et al. [40], it is typically refered to as k-Hitting Set.

An instance of k-Set Cover has a universe of n elements $x_1, x_2, \ldots, x_n$ and a collection $\mathcal{S}$ of subsets $S \subseteq \{x_1, x_2, \ldots, x_n\}$. However, $\forall S \in \mathcal{S} : |S| \leq k$. A k-set cover is a set $C \subseteq \{x_1, x_2, \ldots, x_n\}$ such that $\forall S \in \mathcal{S} : S \cap C \neq \emptyset$. Let $\sigma(\mathcal{S})$ be the collection of all such sets that cover $\mathcal{S}$. The goal is to find a minimal size k-set cover. Let $\mathcal{T}$ be a restriction on $\mathcal{S}$ if for each $S \in \mathcal{S}$ there is a $T \in \mathcal{T}$ such that $T \subseteq S$.

Theorem 4.1.1. *(Sparsification Lemma for k-Set Cover) For all $\epsilon > 0$ and $k \in \mathbb{N}^+$ there is a constant C and an algorithm that given an instance $\mathcal{S}$ of k-Set Cover on a universe of size n, produced a list of $t \leq 2^{\epsilon n}$ restrictions $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_t$ of $\mathcal{S}$ so that $\sigma(\mathcal{S}) = \bigcup_{i=1}^{t} \sigma(\mathcal{T}_i)$ and so that $\forall i : |\mathcal{T}_i| \leq Cn$. Additionally the algorithm runs in time $\mathcal{O}^*(2^{\epsilon n})$ [40].*

This essentially states that for each k and ϵ there exists an algorithm that can compute a subexponential number of new collections such that each new collection contains a number of subsets linear in n. The corollary for k-SAT can be obtained by considering the universe of elements in k-Set Cover to be the $2n$ literals in the formula. $\mathcal{S}$ is then the formula with each $S \in \mathcal{S}$ being a clause. We also require an additional set for each variable, $\{x_i, \overline{x_i}\}$, thus every cover of size exactly n covers exactly one literal for each variable. Each restriction $\mathcal{T}_i$ can be thought of as a subformula of $\mathcal{S}$.

Corollary 4.1.2. *(Sparsification Lemma for k-SAT) For all $k \in \mathbb{N}^+$ and $\epsilon > 0$ there exists an algorithm that takes in a k-CNF formula F with n variables and m clauses and returns a collection of k-CNF formulas $F'_1, F'_2, \ldots, F'_t$, with the following properties.*

1. *F is satisfiable if and only if $\bigvee_{i=1}^{t} F'_i$ is satisfiable.*
2. *F'_i has no more variables than F*
3. *Number of clauses in F'_i is less than $c(k, \epsilon)n$, where $c(k, \epsilon)$ is a constant that does not depend on n.*
4. *The algorithm takes time $\mathcal{O}^*(2^{\epsilon n})$ and $t \leq 2^{\epsilon n}$*

To see how this could be used, consider the issue from earlier. Recall that we had an instance of k-SAT F with $m = n^2$. After a reduction to 3-SAT this instance could be solved in time $2^{\mathcal{O}(n^2)}$. Consider first applying the lemma $F'_1, F'_2, \ldots, F'_t = \texttt{Sparsify}(F, \epsilon_2)$ for some $\epsilon_2 > 0$. Apply the reduction to each F' and solve them with the subexponential algorithm for 3-SAT. Since, by the lemma, the number of clauses in each F'_i is linear in n we get that the time to solve a single instance after the reduction to 3-SAT is $\mathcal{O}^*(2^{\epsilon_1 \cdot (n + (k-3)c(k,\epsilon_2)n)})$, where ϵ_1 is the ϵ that we choose for our subexponential time algorithm for 3-SAT. Choosing $\epsilon_1 > 0$ small enough we can eliminate the other constants to show that a single F'_i can be solved in time $\mathcal{O}^*(2^{\epsilon_3 n})$ for any $\epsilon_3 > 0$. Sparsification can be done in time $\mathcal{O}^*(2^{\epsilon_2 n})$,

the reduction to 3-SAT takes polynomial time. Hence the total time is

$$\mathcal{O}^*(2^{\epsilon_2 n}) + 2^{\epsilon_2 n} \cdot poly(n) \cdot \mathcal{O}^*(2^{\epsilon_3 n}) \tag{4.1}$$

By choosing constants ϵ_2 and ϵ_3 this can be simplified to $\mathcal{O}^*(2^{\epsilon n})$ for any $\epsilon > 0$. So we can solve k-SAT in subexponential time which is what we wanted to show. Hence, what remains is to detail the sparsification algorithm and to prove the sparsification lemma.

4.2 The Sparsification Algorithm

4.2.1 Flowers

Before moving forward, it makes sense to define some new notation which will be helpful later. This notation borrows from the k-Set Cover formulation of the Sparsification Lemma. We will consider a Formula F with n variables and m clauses as a set of clauses $\{C_1, C_2, \ldots, C_m\}$ with each C being a set of literals $\{l_1, l_2, \ldots, l_s\}$ with a literal being either a variable or its negation and $|C| \leq k$. Let

$$vars(F) = \{x \ : x \in C \vee \neg x \in C, C \in F\}$$

and let $lits(F) = \bigcup_{C \in F} C$. It is then easy to see that $|F| = m$ and $|vars(F)| = n$.

We can now being to describe the workings of the algorithm. Let the constants k and $\epsilon > 0$ be given. Define a sequence of $\theta_1, \theta_2, \ldots, \theta_k$. Their values will be decided at a later point and will depend *only* on k and our choice of ϵ but for now consider θ_i to be much larger than θ_{i-1}.

Definition (Flower) A flower is a set of clauses $G = \{C_1, C_2, \ldots, C_t\}$ with the following properties:

1. $\exists l \in \mathbb{N} : \forall C \in G : |C| = l$. All clauses have the same size l.
2. $H = \bigcap_{C \in G} C \neq \emptyset$. The intersection of the clauses is non-empty.

Denote $H = \bigcap_{C \in G} C$ as the heart and $P_i = C_i \setminus H$ as the petals. A flower is considered good if it has many short petals. More formally a flower is good if and only if

$$|G| \geq \theta_{l-|H|} \tag{4.2}$$

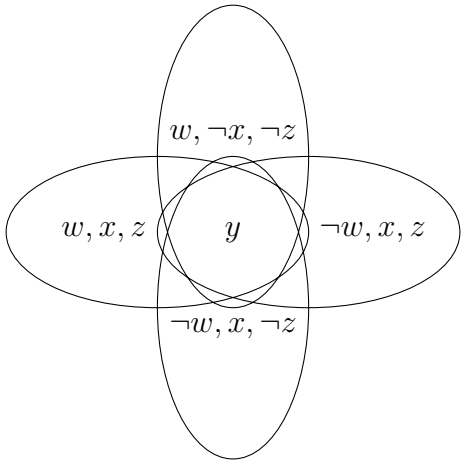


Fig. 4.1 Visualisation of a flower with 4 petals and $H = \{y\}$.

Of good flowers G_1 and G_2, we say that G_1 is better than G_2 if it has shorter clauses $l_1 < l_2$ or if $l_1 = l_2$ but $|H_1| > |H_2|$. It is worth noting that all of these intersections are over literals, so $\{x\} \cap \{\neg x\} = \emptyset$.

Example We can consider a simple formula

$$\begin{aligned} F =& \{C_1, C_2, C_3, C_4, C_5\} \\ F =& \{\{w, x, y, z\}, \{w, \neg x, y, \neg z\}, \{\neg w, \neg x, \neg y, \neg z\} \\ & \{\neg w, x, y, z\}, \{\neg w, x, y, \neg z\}\} \end{aligned}$$

We can form a flower with $G = \{C_1, C_2, C_4, C_5\}$, such that the heart $H = \{y\}$ and the petals $P = \{\{w, x, z\}, \{w\neg x, \neg z\}, \{\neg w, x, z\}, \{\neg w, x, \neg z\}\}$. Note that in this case $l = 4$ and for the flower to be good we would need to satisfy equation 4.2, so we must have $|G| \geq \theta_{4-|H|}$ or $5 \geq \theta_3$. This flower can be seen in Figure 4.2.

4.2.2 Algorithm

The Sparsification Algorithm at each step finds the best flower and then adds the petals to one copy of the formula and the heart to the other copy of the formula, removing redundant clauses. The algorithm then applies itself recursively to the two new formulas. If there are no good flowers, then we are done. This algorithm can be seen in Algorithm 3.

A clause $D \in F$ is considered redundant if there is a clause $C \in F$ such that $C \subseteq D$. Clause D can be removed from F without changing the satisfiability of F. This is because satisfying C automatically satisfies D, hence D has no effect on

Algorithm 3: Sparsification Algorithm (`sparsify`)

Input: k-CNF Formula F, $k \in \mathbb{N}$, $\epsilon > 0$
Output: $F'_1, F'_2, \ldots, F'_t$

$G = \{C_1, C_2, \ldots C_s\} \leftarrow$ `best_flower`(F, ϵ, k)
if *F contains a good flower* **then**
  $H \leftarrow \bigcap_{C \in G} C$
  $P \leftarrow \{C \setminus H : C \in G\}$
  $F_{heart} \leftarrow$ `reduce`$(F \cup \{H\})$
  $F_{petals} \leftarrow$ `reduce`$(F \cup P)$
  return `sparsify`$(F_{heart}, k, \epsilon) \cup$
    `sparsify`$(F_{petals}, k, \epsilon)$
else
  return F

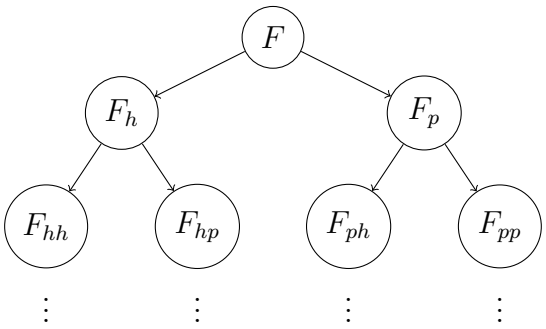


Fig. 4.2 Visualisation of the first two recursive levels of Algorithm 3

the satisfiabilty of F. The algorithm for removal of redundant clauses can be seen in Algorithm 4.

Algorithm 4: Formula Reduction (`reduce`)

Input: F a formula
Output: F' a formula equivalent to F with redundant clauses removed

```
for C, D ∈ F² do
    if C ⊆ D then
        F ← F \ D
return F
```

To find the best flower we can take advantage of the fact that the largest possible heart will at most be of size k. So we don't need to consider every subset of the clauses in F, but just all all possible hearts which is polynomial in n. This algorithm can be seen in Algorithm 5, and it runs in time $\mathcal{O}(k^3(2n)^k m \log k)$. Since k is a constant then this is $\mathcal{O}(n^k m)$, which even when $m = poly(n)$ is still polynomial in n. Correctness of Algorithm 5 follows from the application of the definition of a good flower from Equation 4.2, and from the fact that flowers are considered in order from best to worst, so the best flower will always be returned if one exists.

Algorithm 5: Finding the best flower (`best_flower`)

Input: k-CNF formula F, $k \in \mathbb{N}, \epsilon > 0$
Output: $G = \{C_1, C_2, \ldots, C_s\}$, the best flower good flower in F if any

```
for l ∈ {1, 2, ..., k} do
    for heart_size ∈ {k, k − 1, ..., 1} do
        F* ← {C ∈ F : |C| = l}
        for H ∈ {H ⊆ F* : |H| = heart_size} do
            G ← ∅
            for C ∈ F* do
                if H ⊆ C then
                    G ← G ∪ C
            if |G| ≥ θ_{l−heart_size} then
                return G
return F contains no good flower
```

4.3 Execution of Algorithms

To better understand Algorithm 3 it can be helpful to go through a simple example. We will start with a small CNF formula that we want to sparsify. We let $k = 3$, $n = 3$, we let ϵ be sufficiently large so that the sequence of θs become small enough for any flower with at last 2 petals to be good. The example CNF formula is given as follows.

$$\begin{aligned} F =&(\neg x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_2) \wedge \\ &(\neg x_1 \vee x_3 \vee x_1) \wedge (\neg x_1 \vee \neg x_2 \vee x_1) \wedge \\ &(\neg x_3 \vee x_2 \vee x_1) \end{aligned}$$

Notice that the literals $\neg x_1$ and $\neg x_2$ appears in 2 clauses in F, this becomes the heart of the first flower. The two subformulas obtained after applying Algorithm 4 are given as follows where h denotes the subformula resulting from adding the heart of the first flower and p denotes the subformula resulting from adding the petals.

$$\begin{aligned} h :&(\neg x_1 \vee \neg x_3 \vee x_2) \wedge (\neg x_3 \vee x_2 \vee x_1) \wedge \\ &(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_1) \\ p :&(x_2) \wedge (x_1) \end{aligned}$$

The subformula resulting from adding the heart still has a good flower with $(\neg x_3 \vee x_2)$ as the heart, so the algorithm does not terminate.

$$\begin{aligned} hh :&(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_1) \wedge (\neg x_3 \vee x_2) \\ hp :&(x_1) \wedge (\neg x_1) \end{aligned}$$

No more flowers can be formed at this point, so the algorithm terminates. Note that not every subformula is satisfiable, namely hp, but as long as F is satisfiable the disjunction of all the subformulas is satisfiable. In this case the subformula formed at the first petal branch p is satisfiable and yields a satisfying assignment to F.

4.4 Proof

4.4.1 Satisfiability of F

To prove point 1 from Corollary 4.1.2 we shall aim to show that F is satisfiable if and only if $\bigvee_i^t F_i'$ is satisfiable.

Proof. Consider a non-leaf node in the tree of recursive calls, pictured in Figure 4.2, let the formula at this point be F^*. since this is an non-leaf node, then it has a good flower G and two children F_h^* and F_p^* for the heart branch and petal branch respectively. If either F_h^* or F_p^* is satisfiable then either the heart or all of the petals of G are satisfiable. This means that all clauses in G are satisfiable and any clauses that were made redundant in F_h^* or F_p^* will also be satisfiable in F^*. All other clauses remain the same between F_h^* or F_p^* and F^*. Hence these clauses must also be satisfiable. Thus we have that F^* is satisfiable if F_h^* or F_p^* is satisfiable. Since our choice of F^* was arbitrary, this holds of all non-leaf nodes. Therefore, via induction, we have that if $\bigvee_i^t F_i'$ is satisfiable then F is satisfiable.

To see the converse, let F be satisfiable and let α be the satisfying assignment. If F is not also a leaf node then it has a good flower G. Notice that for a flower $G \subseteq F$, α must either satisfy the heart of the flower or all of the petals. So either F_h or F_p must be satisfied. By repeated application of this argument on F_h and F_p we find that at least one of the leaves must be satisfiable. Which is what we require. □

The proof of point 2 that the algorithm does not add any new variables is obvious from looking at Algorithm 3.

4.4.2 Formulas are linear in n when the algorithm terminates

Definition Given a CNF formula F, a literal u and an integer $l \in \mathbb{N}$, let

$$d_l(u, F) = |\{C \in F : |C| = l \wedge u \in C\}| \tag{4.3}$$

$$d_l(F) = \max_u \{d_l(u, F)\} \tag{4.4}$$

We can interpret the first function as the size of the largest flower with clauses of size l and with u in the heart. Note that the flowers do not need to be good. The second function can therefore be interpreted as the largest flower with clauses of size l. This leads to the first observation.

Lemma 4.4.1. *Let F be a $(\leq k)$-CNF formula that contains no good flowers with clauses of size l, then $d_l(F) \leq \theta_{l-1} - 1$.*

Proof. Suppose that $d_l(F) \geq \theta_{l-1}$. From the definition of $d_l(F)$ in Equation 4.4, this means that

$$\exists u : |\{C \in F : |C| = l \wedge u \in C\}| \geq \theta_{l-1} \tag{4.5}$$

Consider the flower G of clauses with length l with $u \in H$. We know that from Equation 4.5 this flower has θ_{l-1} petals, however, since the heart is at least of size 1 then the length of the petals is $\leq l - 1$. Therefore, from equation 4.2 the flower G is good. Thus

$$d_l(F) \geq \theta_{l-1} \implies F \text{ has a good flower of size } l \tag{4.6}$$

Therefore, if we take the contrapositive of (4.6) we obtain what we want to show.

$$F \text{ does not have a good flower of size } l \implies d_l(F) \leq \theta_{l-1} - 1 \tag{4.7}$$

□

Corollary 4.4.2. *Let F be a $(\leq k)$-CNF formula without any good flowers. Then $|F| \leq 2k\theta_{k-1} \cdot |vars(F)|$.*

Proof. We can apply lemma 4.4.1 for all clause lengths. Thus,

$$\forall l \in \{1, 2, \ldots, k\} : \quad d_l(F) \leq \theta_{l-1} - 1 \tag{4.8}$$

$$\forall l \in \{1, 2, \ldots, k\} : \quad \forall u \in lits(F) : d_l(u, F) \leq \theta_{l-1} - 1 \tag{4.9}$$

Since there are at most $2 \cdot |vars(F)|$ literals in F, summing over u yields

$$\forall l \in \{1, 2, \ldots, k\} : \sum_{u \in lits(F)} d_l(u, F) \leq 2\theta_{l-1} \cdot |vars(F)| \tag{4.10}$$

Additionally, since every clause contains at least one literal then $|\{C \in F : |C| = l\}| \leq \sum_{u \in lits(F)} d_l(u, F)$ since if $u \in C$ then C contributes to $d_l(u, F)$. Therefore

$$\forall l \in \{1, 2, \ldots, k\} : |\{C \in F : |C| = l\}| \leq 2(\theta_{l-1} - 1) \cdot |vars(F)| \tag{4.11}$$

Then we can sum up across all l to get

$$|F| \leq 2 \sum_{l=1}^{k} (\theta_{l-1} - 1) \cdot |vars(F)| \tag{4.12}$$

Recall that $\theta_i > \theta_{i-1}$, so we can upper bound all θ_{l-1} by θ_{k-1}.

$$|F| \leq 2\sum_{l=1}^{k}(\theta_{k-1} - 1) \cdot |vars(F)| \tag{4.13}$$

$$|F| \leq 2k\theta_{k-1} \cdot |vars(F)| \tag{4.14}$$

□

Observe that $2k\theta_{k-1}$ is a constant that is independent of n, hence we satisfy point 3) from Corollary 4.1.2 of the Sparsification Lemma by setting $c(k, \epsilon) = 2k\theta_{k-1}$.

4.4.3 There are subexponentially many leaves

The final point is to show that $t \leq 2^{\epsilon n}$. This is the most involved section of the proof. We will show two things, which will uppert bound the number of leaves in the recursion tree. Firstly, we will claim that the longest path from the root to any leaf is at most $\alpha \cdot n$, for some constant $\alpha \in \mathbb{N}$ Secondly, we will show that the number of branches that correspond to adding petals to the formula is low. Specifically, we will show that the number of petal branches is at most $\frac{kn}{\delta}$. The constants α and δ will depend only on the choices of of k and ϵ.

To get a sense for the number of leaves in the recursion tree, consider that each path from the root to a leaf can be written as a sequence of p's and h's, with p's corresponding to when the recursion tree branched to add petals and h's corresponding to the when the recursion tree branched to add the heart of the flower. The start of this recursion tree can be seen in Figure 4.2. The four nodes at the bottom show the start of this sequence of p's and h's. To then count the number of leaves in the recursion tree, we need to just count the number of sequences. This can be done by summing the number of permutations of p's and h's for each number of petal branches. The number of paths from the root to a leaf with exactly P petal branches would be given by $\binom{\alpha \cdot n}{P}$. So therefore, if we know that there are at most $\frac{kn}{\delta}$ petal branches then the total number of leaves is given by

$$\sum_{i}^{kn/\delta} \binom{\alpha \cdot n}{i} \leq 2^{H(\frac{k}{\alpha \cdot \delta})\alpha n} \tag{4.15}$$

where $H(x) = -x\log(x) - (1-x)\log(1-x)$ is the binary entropy function. This bound can be obtained from Stirling's approximation of $n!$ (for more details see [24]). We will show that $H(k/\alpha\delta)\alpha \leq \epsilon$ which would complete the proof.

Lemma 4.4.3. *Let F be the current formula at a non-leaf node occurring in the recursion tree as exemplified in Figure 4.2. Let $G = \{C_1, C_2, \ldots, C_s\}$ be the flower identified as the best flower at this node. Denote F' as a child of F in the recursion tree. If $|C_i| = l$, then F' would be formed by adding clauses of size $1 \leq j < l$ to F. We claim that*

$$d_j(F') \leq 2 \cdot \theta_{j-1} \tag{4.16}$$

Proof. There are two cases that need to be considered. The first is when $F' = \texttt{reduce}(F \cup \bigcap_{C \in G} C)$, i.e. by adding the heart of the flower to F and removing redundant clauses. The second is when $F' = \texttt{reduce}(F \cup \{C \setminus H : C \in G\})$ where H is the heart of the flower, i.e. adding all the petals to F and removing redundant clauses.

Case: Adding the heart to F

Recall that a flower is considered better than another if has shorter clauses. Therefore, since the best flower is picked at every step, there cannot have been a good flower in F with clauses all of size j, since $j < l$. Then from Lemma 4.4.1, we can say that

$$d_j(F) \leq \theta_{j-1} - 1 \tag{4.17}$$

Then, since only one clauses is added to F', namely the heart and this is a clause of size j so then we can conclude that

$$d_j(F') \leq \theta_{j-1} \leq 2 \cdot \theta_{j-1} \tag{4.18}$$

which is what we require.

Case: Adding petals to F

For the sake of contradiction, we will assume that $d_j(F') \geq 2 \cdot \theta_{j-1} + 1$. Recall, from the definition of $d_j(F)$ in Equation 4.4, this means that there exists some literal u that appears in $\geq 2 \cdot \theta_{j-1} + 1$ clauses of size j in F'. Since we know that there was no good flower with clauses of size j in F then we know from Lemma 4.4.1, that $d_j(u, F) \leq \theta_{j-1} - 1$. So then, since there are $\geq 2 \cdot \theta_{j-1} + 1$ clauses of size j containing a literal u in F' but only $\leq \theta_{j-1} - 1$ in F, then these clauses must have been added to F' from the petals of the flower, i.e. $d_j(u, \{C_1 \setminus H, C_2 \setminus H, \ldots, C_t \setminus H\}) \geq \theta_{j-1}$ where H is the heart of the flower. However, we can form a better flower in F with $\{u\} \cup H$ as the heart. This is a good flower because the new petals have size $j - 1$

and we know that there are more than θ_{j-1} of them, so it satisfies Equation 4.2 and is a good flower. This is a contradiction since the heart is larger and hence it is a better flower than the one picket in F, but Algorithm 5 always picks the best flower. We then have that our assumption must be false and

$$d_j(F') \leq 2 \cdot \theta_{j-1} \tag{4.19}$$

as required. This concludes the proof. □

The interpretation of this proof is that once a clause of size j is added to a formula in the recursion tree, then the number of clauses of size j will be linear in n. This is because adding clauses of a different size cannot increase the number of clauses of size j, and adding clauses of size j invokes the lemma to show that this level is still linear in n.

We introduce the notion of *Native* and *Immigrant* clauses, meaning clauses that were in the original formula F at the root of the recursion tree and clauses that were added as hearts and petals respectively. Note that only *Immigrants* can cause a clause to be removed from a formula in a call to `reduce`.

We will now bound the number of *Immigrant* clauses that can be removed by a single clause in a call to `reduce`.

Lemma 4.4.4. *For a formula F, in a call to `reduce(F)`, a single clause C can remove at most $2 \cdot \theta_{j-1}$ Immigrant clauses of size j in F*

Proof. If no *Immigrant* clauses of size j exist in F then none can be removed and we are done. Otherwise, since a clause of size j has at some point been added to F we can apply Lemma 4.4.3 to say that $d_j(u, F) \leq 2 \cdot \theta_{j-1}$, where u is a literal contained in the clause C. It then follows that C can be a subset of at most $2 \cdot \theta_{j-1}$ clauses of size j. Since these are the clauses that are removed by C, then we are done. □

We will now attempt to bound the number of *Immigrant* clauses added to the formula along a path from the root to the leaf. To do this, we define a sequence of numbers $\gamma_1 = 2$ and $\gamma_l = 4\theta_{l-1}\gamma_{l-1}$.

Lemma 4.4.5. *For a path from the root to a leaf in the recursion tree and for $1 \leq l \leq k-1$, let $I_{\leq l}$ denote the number of Immigrant clauses of size $\leq l$ that are added along the path. Then*

$$I_{\leq l} \leq \gamma_l \cdot n \tag{4.20}$$

Proof. We will show this by induction on l. Let F denote the formula at the root of the recursion tree and let F' denote the formula at a leaf. Let R_l be the number of *Immigrant* clauses of size exactly l that are removed along the path from F to F'

The base case where $l = 1$ follows from the fact that there are only $2n$ clauses of size 1, so $I_{\leq l} \leq 2n = \gamma_1 n$. We assume that our inductive hypothesis holds for $l - 1$.

$$I_{\leq l-1} \leq \gamma_{l-1} n \tag{4.21}$$

We now show that this implies that the lemma holds for l.

Denote I_l as the number of *Immigrant* clauses of size exactly l that are added along the path from F to F'. Each such added clause will either be removed at some point in the path or remain in F'. Hence we can bound I_l from above.

$$I_l \leq R_l + |\{C \in F' : |C| = l\}| \tag{4.22}$$

From Equation 4.11 in Corollary 4.4.2 we know that $|\{C \in F' : |C| = l\}| \leq 2\theta_{l-1} n$.

We will now bound R_l. If $R_l \neq 0$ then there is some clause D of size l that was removed by some *Immigrant* clause $C \subset D$ where $|C| \leq l - 1$. By Lemma 4.4.4 we know that this clause could have removed at most $2\theta_{l-1}$ clauses of size l. From the inductive hypothesis we know that there are at most $\gamma_{l-1} n$ such clauses. Hence we obtain the following bound on R_l.

$$R_l \leq 2\theta_{l-1}\gamma_{l-1} n \tag{4.23}$$

Additionally, we observe that

$$I_{\leq l} = I_{\leq l-1} + I_l \tag{4.24}$$

Combining these inequalities we obtain the following:

$$\begin{aligned} I_{\leq l} &= I_{\leq l-1} + I_l \\ &\leq \gamma_{l-1} n + R_l + |\{C \in F' : |C| = l\}| \\ &\leq \gamma_{l-1} n + 2\theta_{l-1}\gamma_{l-1} n + 2\theta_{l-1} n \\ &\leq 4\theta_{l-1}\gamma_{l-1} n \\ &\leq \gamma_l n \end{aligned}$$

By the principle of induction we know that this holds for all $l \geq 1$, which concludes the proof. □

Additionally, since at every step in the recursion, at least one *Immigrant* clause is added to the formula and this clause must have size $\leq k-1$ then it immediately follows that

Corollary 4.4.6. *Any path from root to leaf has at most $\gamma_{k-1}n$ edges.*

Since the length of a path from the root to a leaf is at most $\gamma_{k-1}n$, if we set $\alpha = \gamma_k \geq \gamma_{k-1}$ then we achieve our first goal of bounding the length of the longest path by $\alpha \cdot n$. What remains is to show that the number of petal branches in the recursion tree is at most $\frac{kn}{\delta}$.

Lemma 4.4.7. *For any path from the root to a leaf in the recursion tree, at most $\frac{kn}{\delta}$ branches are from adding petals.*

Proof. For a fixed path from the root to a leaf in the recursion tree, let P_l be the number of times that petals are added to the formula with petals of size l. Recall the definition of a good flower, Equation 4.2 tells us that such a flower must have had at least θ_l petals. Therefore, we can conclude that along this path $P_l \cdot \theta_l$ individual petal clauses of size l would have been added. Recall from Lemma 4.4.5 that $I_{\leq l}$ is the number of immigrant clauses of size $\leq l$ added along a path. Since every petal clause is an immigrant we obtain

$$I_{\leq l} \geq P_l \cdot \theta_l \tag{4.25}$$

However, from Lemma 4.4.5 we also have that $I_{\leq l} \leq \gamma_l n$, so

$$\begin{aligned} P_l \cdot \theta_l \leq & I_{\leq l} \leq \gamma_l n \\ P_l \cdot \theta_l \leq & \gamma_l n \\ P_l \leq & \frac{\gamma_l}{\theta_l} n \end{aligned}$$

Thus, if we were to have that $\delta = \frac{\theta_l}{\gamma_l}$ then we obtain that $P_l \leq \frac{n}{\delta}$. If we sum over all l to get the total number of petal branches then we obtain what we require.

$$\begin{aligned} \#\text{petal branches} &= \sum_{l=1}^{k} P_l \\ \#\text{petal branches} &\leq \sum_{l=1}^{k} \frac{n}{\delta} \\ \#\text{petal branches} &\leq \frac{kn}{\delta} \end{aligned}$$

This concludes the proof. □

We are now ready to set the constants θ_i. We require that $\frac{\theta_l}{\gamma_l} = \delta$ from Lemma 4.4.7 and that $\gamma_l = 4\gamma_{l-1}\theta_{l-1}$ from Lemma 4.4.5. Consider $\gamma_l = (4\delta)^{2^{l-1}-1}$ and $\theta_l = \delta\gamma_l$. Check that we have $\frac{\theta_l}{\gamma_l} = \delta$ as required. We also check that

$$\begin{aligned}\gamma_l &= 4\gamma_{l-1}\theta_{l-1} \\ &= 4\delta\gamma_{l-1}^2 \\ &= 4\delta((4\delta)^{2^{l-2}-1})^2 \\ &= (4\delta)^{2^{l-1}-1}\end{aligned}$$

as required.

We have now bounded the length of the longest path in the recursion tree by $\gamma_k n$ and bounded the number of petal branches by kn/δ. We therefore have that the number of leaves in the recursion tree is bounded by

$$\sum_{i=0}^{kn/\delta} \binom{\gamma_k n}{i} \leq 2^{H(\frac{k}{\delta\gamma_k})\gamma_k n}$$

What remains is to show that $H(\frac{k}{\delta\gamma_k})\gamma_k \leq \epsilon$. If we require that $\delta \geq k$ then we get that $\frac{k}{\delta\gamma_k} \leq \frac{1}{2}$. Consider the binary entropy function

$$H(x) = -x\log(x) - (1-x)\log(1-x) \tag{4.26}$$

for small x we have that $-x\log(x) \geq -(1-x)\log(1-x)$, so we can bound

$$H(x) \leq -2x\log(x) \tag{4.27}$$

Plugging in $x = k/(\delta\gamma_k)$, we obtain

$$\begin{aligned} H(\frac{k}{\delta\gamma_k}) &\leq -2\frac{k}{\delta\gamma_k}\log(\frac{k}{\delta\gamma_k}) \\ &\leq -2\frac{k}{\delta\gamma_k}(\log(k) - \log(\delta) - \log(\gamma_k)) \\ &\leq -2\frac{k}{\delta\gamma_k}(\log(k) - \log(\delta) - (2^{k-1}-1)\log(4\delta)) \\ &\leq -2\frac{k}{\delta\gamma_k}(-2^k\log(\delta)) \\ &\leq 2\frac{k2^k\log(\delta)}{\delta\gamma_k} \\ H(\frac{k}{\delta\gamma_k})\gamma_k &\leq 2\frac{k2^k\log(\delta)}{\delta} \end{aligned}$$

Therefore, given k and ϵ all that remains is to choose our δ such that

$$2\frac{k2^k\log(\delta)}{\delta} \leq \epsilon \tag{4.28}$$

Which is what we require for our claim that there are at most $2^{\epsilon n}$ leaves. It immediately follows that the run time of the algorithm is $\mathcal{O}^*(2^{\epsilon n})$, since there can be at most $2^{\epsilon n}$ recursive calls and each takes polynomial time.

This concludes the proof. So we can say that Equation 4.1 holds and if 3-SAT can be solved in subexponential time, then so can k-SAT for all $k \geq 3$.

4.5 Exploration of the Lemma

To gain a better understanding of the relationship between the constants $k, \epsilon, \delta, \gamma_l$ and θ_l consider the following.

$$\begin{aligned} 2\frac{k2^k\log(\delta)}{\delta} &\leq \epsilon \\ \frac{\log\delta}{\delta} &\leq \frac{\epsilon}{k\cdot 2^{k+1}} \end{aligned}$$

l	θ_l	γ_l
1	23 039 999	1
2	$\sim 2.123 \cdot 10^{15}$	92159996
3	$\sim 1.803 \cdot 10^{30}$	$\sim 7.828 \cdot 10^{23}$

Table 4.1 Growth of Coefficients

Using the following upper bound $\frac{\log x}{x} \leq \frac{\log(x+1)}{x} \leq \frac{1}{\sqrt{1+x}}$ [72]. We can therefore choose δ such that

$$\frac{1}{\sqrt{1+\delta}} \leq \frac{\epsilon}{k \cdot 2^{k+1}}$$
$$\delta \geq \frac{k^2 \cdot 2^{2k+2}}{\epsilon^2} - 1$$

We can then note that δ grows exponentially with k and polynomially with $\frac{1}{\epsilon}$. Thus from the definition of γ_l and θ_l, we can see that they grow roughly as δ^{2^l}, so therefore we can justify our original claim that $\theta_l \gg \theta_{l-1}$.

Example We can consider an example where we have a 3-CNF formula that we wish to sparsify, $k = 3$ and we choose $\epsilon = 10^{-2}$. We can see that this leads to

$$\delta \geq \frac{3^2 \cdot 2^{2\cdot 3+2}}{(10^{-2})^2} - 1 = 23\,039\,999$$

We can the calculate the terms of γ_l and θ_l. The growth of these coefficients for this example can be seen in Table 4.1. Notice, that even for small examples these constants exhibit rapid growth. Therefore, even though we know from Corollary 4.4.2 that at the end of Algorithm 3, $|F| \leq 2k \cdot \theta_{k-1} n$ which is $\mathcal{O}(n)$, the constant hidden in the Big-$\mathcal{O}$ is typically significantly larger than most practical instances of SAT which have typically at most on the order of 10^6 clauses [6].

Therefore, due to large constants, the sparsification lemma is not easy to apply to k-SAT solvers in the hopes of achieving better practical performance as measured by wall time. However, it is sufficient to show that Equation 4.1 holds.

However, one by notice that there exists many upper and lower bounds in this proof which may have better bounds, leading to smaller constants. For example, in Corollary 4.4.2 we bounded the sequence $(\theta_1 + \theta_2 + \cdots + \theta_{k-1})$ by $k\theta_{k-1}$. Although, due to the rapid growth of θ_l, this is almost k times greater. Hence it is possible that there exists tighter bounds than ones shown in this chapter.

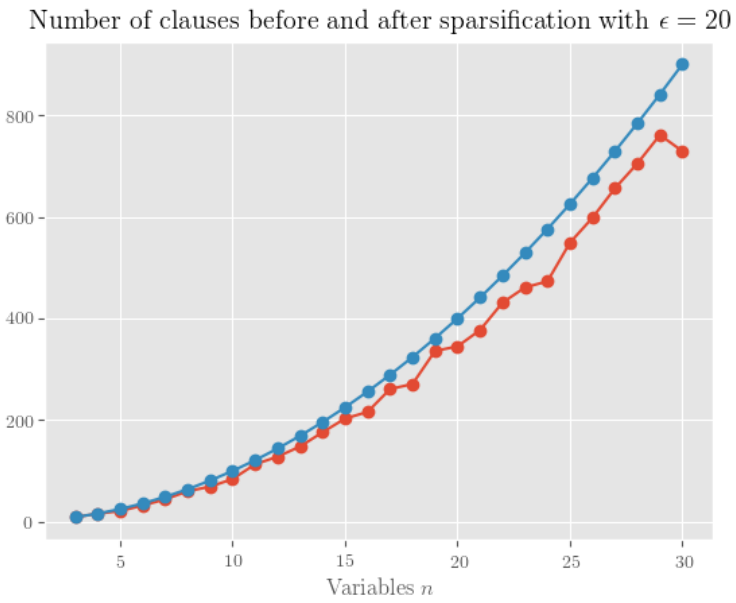


Fig. 4.3 The size of the largest subformula compared with the size of the original formula

Numerical Tests

To gain a better understanding of how the sparsification lemma behaves we implemented the algorithms presented in this chapter and ran them on randomly CNF formulas. Taking inspiration from the problem presented in the introduction, we generate CNF formulas where $m = n^2$. Each clause picks 3 literals at random with uniform probability with the additional requirement that all clauses must be unique. We do not concern ourselves with whether the formulas are satisfiable or not. Some results from these tests can be seen in Figures 4.3, 4.4, 4.5.

As we can see in Figure 4.3, for small n the largest subformula is only slightly smaller than the size of the original formula. However, we note that the bound for this instance was approximately $600n$, so for small n the algorithm could have done nothing whilst satisfying the bound. This suggests that often Algorithm 3 will return CNF formulas sparser than required.

Due to computational restrictions, we were unable to test the behaviour of the algorithm as the length of the original formula approaches that of the bound. We also note that due to the large hidden constant in the bound on sparsity for small n there are often not enough unique clauses for the original formula to be above the bound on sparsity.

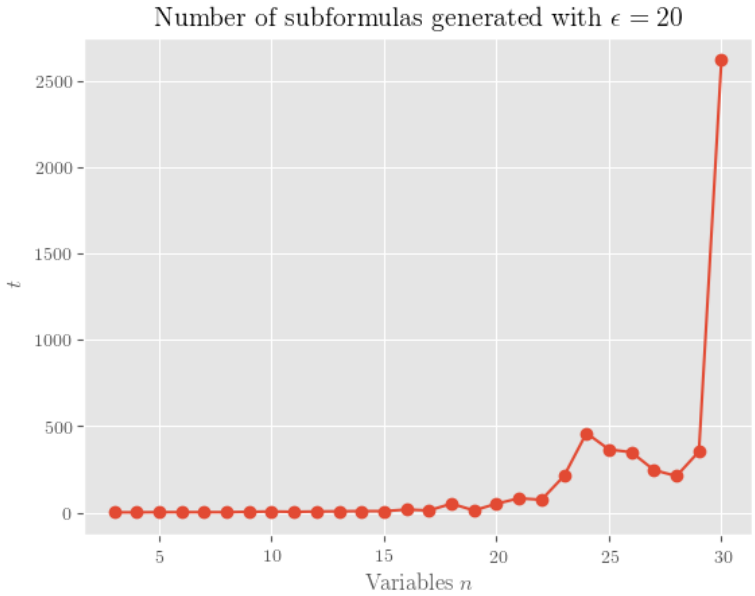


Fig. 4.4 Number of subformulas returned

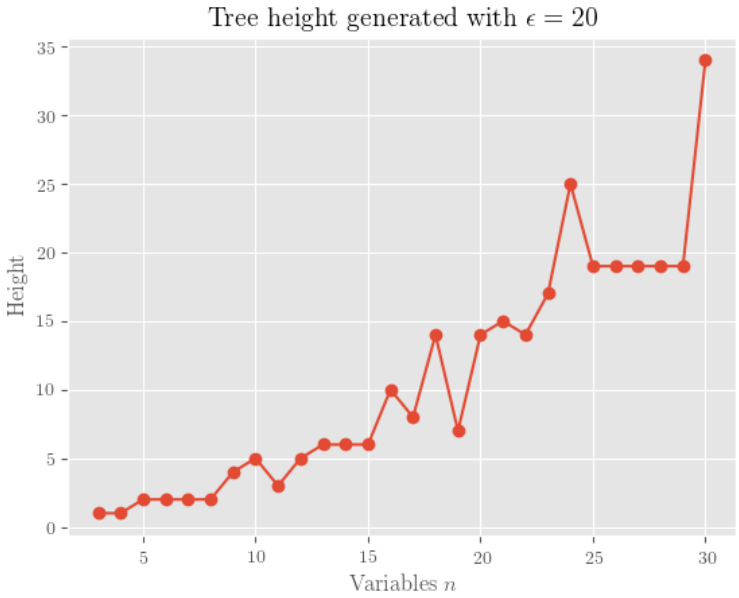


Fig. 4.5 The height of the recursion tree for varying n

To get a sense for computational cost see Figures 4.4 and 4.5, where we see rapid growth of the number of subformulas returned by Algorithm 3 and steady growth in the height of the recursion tree. However, considering our choice of $\epsilon = 20$ we can only guarantee that the number of subformulas returned is less than 2^{20n} which is a function growing much faster than what we observe in Figure 4.4. This again suggests our theoretical bounds can be improved by some constant factor.

Furthermore, even though Algorithm 5 runs in polynomial time it quickly becomes infeasible to the best flower of even the first formula. Considering that we let $m = n^2$ and $k = 3$ the computational complexity of Algorithm 5 becomes $\mathcal{O}(n^5)$ where the hidden constant is > 100. While this is certainly polynomial time it makes it infeasible to test formulas large enough to be above the bound on sparsity.

Intuitions and Ideas for Further Work

The sparsification lemma seems to capture an idea that dense formulas are somehow easy since they can be made sparse in subexponential time potentially cutting away many clauses. This is an idea that will be covered from another perspective in Chapter 5. We can imagine that the hardest CNF formulas are those where m is $\mathcal{O}(n)$, since this can be enforced by the sparsification lemma, incurring only subexponential cost.

Furthermore, consider that the branching performed in Algorithm 3 is similar to the branching performed by DPLL seen in Chapter 2. However, in the worst case DPLL runs in $\mathcal{O}(2^n)$. Impagliazzo et al. [40] comment that branching on a disjunction of variables rather than single variables ends up requiring less branching in total. This then brings up the question of whether the idea of branching on disjunctions of variables could applied to SAT solvers. If we relax the requirement that the clause branched on had to be the best flower and instead picked disjunctions of variables based on sensible heuristics that are efficiently computable, would this lead to smaller recursion trees in algorithms based on DPLL? We leave this question for further work.

Chapter 5

Structures in SAT

Introduction

In this chapter we attempt to provide an explanation of why it is simultaneously possible to have fast "in practice" SAT solving algorithms, as seen in Chapter 2, and also have standing conjectures about the complexity of SAT that suggest that we should not expect to solve SAT in subexponential time, as seen in Chapter 3. From this it seems that many SAT instances are significantly easier than their worst-case counterparts.

This discrepancy between the hardness of the worst-case instances and the typical hardness has been known for NP-COMPLETE problems such as k-colouring [74], where it was shown by Turner et al. that "almost all" k-colourable graphs could be coloured by a simple $\mathcal{O}(|V| + |E| \log k)$ time algorithm, where "almost all" means that Pr[Algorithm finds a colouring] $\to 1$ as $|V| \to \infty$. A similar argument could then be made for SAT via reduction to k-colourability (see Figure 3.1).

We will cover exactly where these hard worst-case instances lie by considering the clause ratio of an instance and also considering a notion distance of an instance to a polynomial time tractable sub-classes of SAT. We will also touch on the issue of attempting to determine whether a single SAT instance is hard or not.

5.1 Heavy-Tailed Distributions of Difficulty

5.1.1 Issues Measuring Expected Running Time of SAT Solvers

A natural question when attempting to measure the difficulty of certain SAT instances is whether in instance is inherently hard, or if the specific solver used was simply unlucky with this instance.

To shed light on this issue Gomes et al. attempt to characterise the distribution of run times for SAT solvers (with some random element) on fixed SAT instances [33]. To do this they empirically measure the number of backtrack steps required to solve a SAT instance for a solver based on the Davis-Putnam-Logemann-Loveland procedure (DPLL) [20]. Gomes at al. demonstrate that the sample mean does not stabilise with increasing the size of the sample. This indicates that the distribution of backtracks required has undefined moments, i.e has non-negligible tail probabilities.

This has the unfortunate consequence that a single SAT instance may appear trivial to solve at one point and very challenging at another and there is no well defined mean difficulty. This can be seen in Figure 5.1, which shows a failure of the sample mean to converge to any value. Gomes et al. note that the sample median, however, does converge quickly to a value of 1.

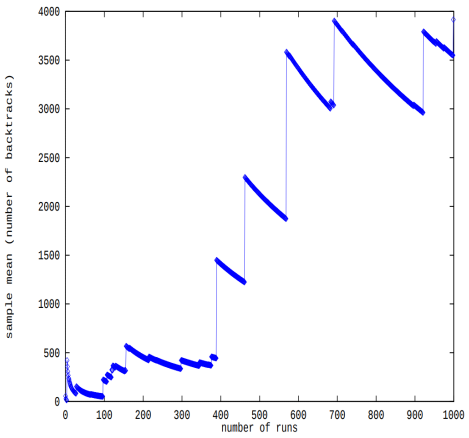


Fig. 5.1 Sample mean of the number of backtracks required to solve a SAT encoding of the quasigroup completion problem.

5.1.2 Randomising SAT Solvers

In order to permit a distribution over backtracks required, a random element has to be added to the SAT solvers being analysed. Gomes et al. consider modifications

to Satz and Relsat which are both based on DPLL. These solvers use a collection of heuristics to determine which variable to branch on. Gomes et al. define a constant H, such that all variables that are in the top H-percent, as evaluated by the heuristics, are chosen at random with uniform probability.

5.1.3 Pareto-Lévy Distributions

Distributions with infinite moments

Gomes et al. consider a class of distributions called Pareto-Lévy distributions which have the following form.

$$\begin{aligned} \mathrm{P}(X > x) &\sim Cx^{-\alpha}, \;\; x > 0^{1} \\ C &> 0 \\ 0 < &\alpha < 2 \end{aligned}$$

This means that the distribution decays polynomially in x as opposed to exponentially, as would be the case for a Gaussian distribution. An example of a probability distribution with this form is the Cauchy distribution which is defined as follows.

$$\mathrm{Cauchy}(\gamma, \delta) = \frac{1}{\pi} \cdot \frac{\gamma}{\gamma^2 + (x - \delta)^2}$$

For the Cauchy distribution $\alpha = 1$. For the Pareto-Lévy distribution $\alpha = 0.5$.

Distribution of Backtracks required by randomised SAT Solvers

Gomes et al. empirically sample the number of backtracks required to solve a SAT instance for a few different instances taken from different domains. These domains are the Quasigroup completion problem, Scheduling, logistics planning and register allocation. For brevity only the scheduling example is shown here in Figure 5.2. This plot shows the probability $\mathrm{P}(X > x)$, where x is the number of backtracks required to solve the instance on the randomised solver. The scale on the plot is logarithmic so a straight line implies polynomial decay.

$$\begin{aligned} m \cdot \log(x) + b &= \log(\mathrm{P}(X > x)) \\ x^m e^b &= \mathrm{P}(X > x) \\ \mathrm{P}(X > x) &\sim Cx^m \end{aligned}$$

[1] Here $f(x) \sim g(x)$ is used to mean $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1$

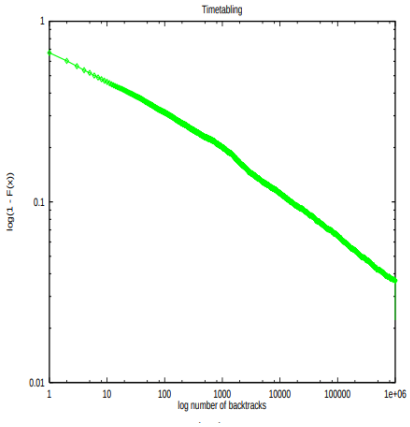


Fig. 5.2 Cumulative log plot of the number of instances solved with a given number of backtracks. Gomes et al. estimates a value of $\alpha = 0.219$ for this problem

Hence we can estimate α from the slope of the line in the plot.

5.1.4 Motivation for rapid restarts in SAT solvers

Gomes et al. go on to state that since there is non-negligible probability to the left side of the distribution then a sequence of many short runs would be more effective than a single long run. Gomes et al. then empirically validate their claim by modifying Satz and Relsat to include rapid restarts when a certain number of backtracks is reached. This results in an order of magnitude reducing in the time to solve SAT instances, although the point at which restarting is most effective depended on the domain of the problem.

5.1.5 Points of Caution

From this we should be cautious to label any given SAT instance as difficult since the underlying distribution of the number of backtracks required to find a satisfying assignment has undefined mean and variance. As such we also cannot immediately trust a sample of runs giving a mean runtime, since the mean does not converge to anything. Ideally, we should consider the median runtime, which is defined for heavy-tailed distributions and is therefor more trustworthy in empirical tests.

5.2 Phase Transition

Consider a SAT instance F with few clauses compared to the number of variables, intuitively this formula has many degrees of freedom and few constraints so it is

highly likely that F will be satisfiable. Furthermore, if we consider how a run of a backtracking solver would behave on such an instance, since it only backtracks when it finds a conflict and few conflicts exist then it would find an assignment without requiring many backtracking steps. If we instead consider a SAT instance F' with many clauses compared to the number of variables we can see that a backtracking solver would be spending most of its time applying unit propagation. Hence in this case also the solver would be able to decide that F' is not satisfiable quickly.

We call these instances underconstrained and overconstrained respectively. Work by Cheeseman et al. suggests that all NP-COMPLETE problems have some order parameter which exhibits a phase transition between underconstrained and overconstrained. Furthermore, the hard instances of an NP-COMPLETE problem occur exactly at the point of the phase transition [14]. Here, hardness is measured by the number of backtracking steps that are required to solve an given instance of an NP-COMPLETE problem. Taking k-SAT as an example, the order parameter is the ratio of the number of clauses to the number of variables $\frac{m}{n}$ and the hardest instances of k-SAT are found at some critical value c_k. This explains how Turner et al. were able to achieve their algorithm, most random instances are far from the critical value [74].

5.2.1 Transition Sharpness

The phase transition is known to be "sharp" [29] in the following sense. Let $U_k(n, m)$ be the uniform distribution over all k-CNF formulas with n variables and m clauses and c_k be the critical value, then

$$\Pr_{F \sim U_k(n,m)}[F \text{ is satisfiable}] = \begin{cases} 0 & \frac{m}{n} > c_k \\ 1 & \frac{m}{n} < c_k \end{cases} \quad \text{as } n \to \infty \tag{5.1}$$

For 2-SAT this critical value is known to be $c_2 = 1$, [16, 31]. However, for all $k \geq 3$, no exact value of c_k is known. Although it is known that $3.003 < c_3 < 4.81$ and $c_3 \approx 4.24$ [18] (see Figure 5.4).

For large n only instances close to critical values will be hard (See Figure 5.3). This fact has been used by Mitchell et al. [59] to define probability distributions over k-SAT instances that are hard for the purpose of creating new challenging benchmarks.

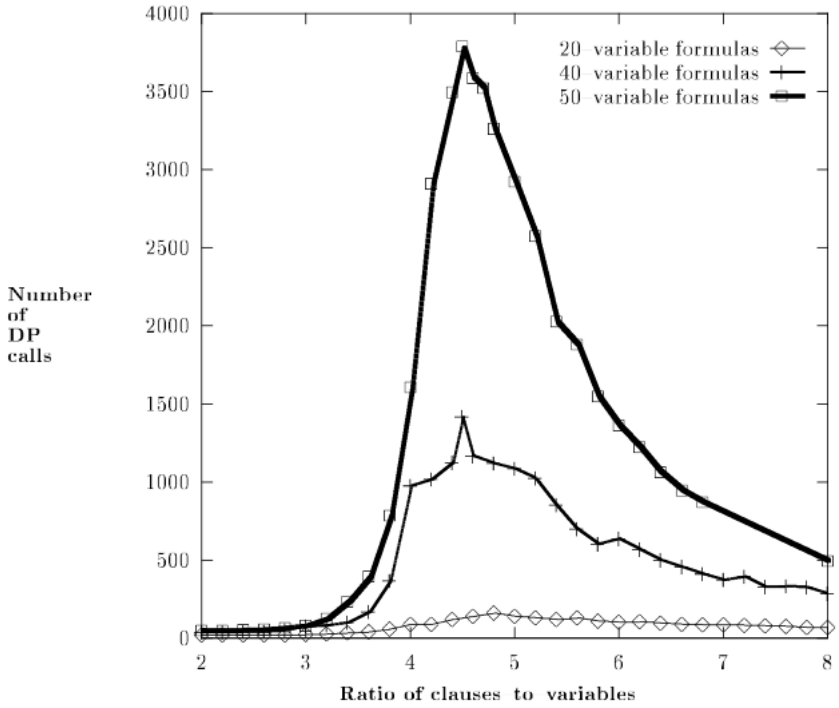


Fig. 5.3 Median number of backtracks required to solve a 3-SAT instance for varying clause ratio [59]

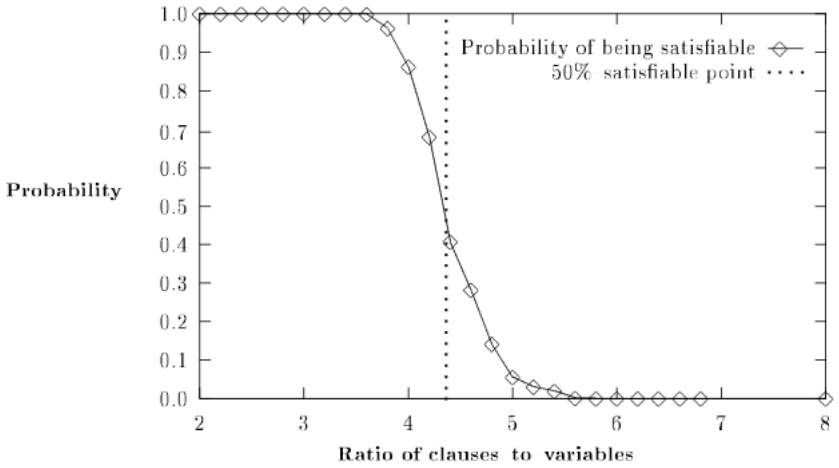


Fig. 5.4 Probability that a random 3-SAT instance is satisfiable for varying clause ratio [59]

Analogies to Spin Glasses and Statistical Mechanics

Work by Kirkpatrick uses elements of statistical mechanics and an analogy of random k-SAT to spin glasses to propose a form for phase transition. Their work suggest that for some fundamental function f and constants c_k and v then

$$\Pr_{F \sim U_k(n,m)}[F \text{ is satisfiable}] = f\left(\left(\frac{m}{n} - c_k\right) \cdot \frac{n^{1/v}}{c_k}\right) \tag{5.2}$$

Using empirical methods f and the constants c_k and v can be estimated. Kirkpatrick et al. estimate $c_3 \approx 4.17$ and $v \approx 1.5$ [45]. We note that this form for the satisfiability probability illustrates that for any $\frac{m}{n} - c_k \neq 0$ then as $n \to \infty$ the probability tends to $f(-\infty)$ or $f(\infty)$. If we have that $\lim_{x \to -\infty} f(x) = 1$ and $\lim_{x \to \infty} f(x) = 0$ then we can justify Equation 5.1.

5.2.2 Generalisations

However, most "in practice" SAT instances do not have fixed clause length and instead have many short and long clauses. Work by Gent et al. generalises the SAT phase transition to mixed-clause instances [29]. Consider a probability distribution over the integers $\phi(k)$, let a ϕ-SAT formula F be such that for a randomly selected clause $C \in F$

$$\Pr[|C| = k] = \phi(k) \tag{5.3}$$

i.e. clauses of size k are picked to be in F with probability $\phi(k)$.

Denote the critical value for a particular distribution c_ϕ. Gent et al. give a simple upper bound for c_ϕ. Let α be an assignment chosen uniformly at random and C_k be an arbitrary clause of length k. For a given distribution of clause lengths ϕ we define d_ϕ as follows:

$$d_\phi = \Pr_{k \sim \phi}[\alpha \text{ satisfies } C_k] = \sum_{k=1}^{\infty} \phi(k)\left(1 - \left(\frac{1}{2}\right)^k\right) \tag{5.4}$$

If we consider a ϕ-SAT instance $F = \{C_1, C_2, \ldots, C_m\}$ with m clauses then

$$\begin{aligned} \Pr[\alpha \text{ satisfies } F] &= \prod_{i=1}^{m} \Pr[\alpha \text{ satisfies } C_i] \\ &= d_\phi^m \end{aligned}$$

Thus, if the number of variables is n, then the expected number of satisfying assignments is $2^n d_\phi^m$. For a formulas to be unsatisfiable as $n \to \infty$ we must have

that $2^n d_\phi^m < 1$ or equivalently $2(d_\phi)^{\frac{m}{n}} < 1$. Rearranging we get $\frac{m}{n} > \frac{-1}{\log_2(d_\phi)}$. Since as $n \to \infty$ all formulas with a clause ratio greater than this are unsatisfiable then we know that

$$c_\phi \leq \frac{-1}{\log_2(d_\phi)} \tag{5.5}$$

This gives us our desired upper bound.

5.3 Backdoors

5.3.1 Formalising Intuitions

One intuition for why some SAT instances are easy come from the idea that not all variables are equal. One can consider a dichotomy of variables, dependent variables and independent variables, where dependent variables are variables that are needed to encode a specific problem but are simple consequences of the assignment to the independent variables. As an example, consider a software verification task, it is likely that there many variables included in the SAT encoding that are encoding straightforward implications or auxiliary information. The assignments to these variables can be quickly deduced after the heart of the combinatorial problem is solved, namely finding an assignment to the independent variables. As such, dependent variables do not contribute to the combinatorial hardness of the instance, since it is sufficient to find an assignment to the independent variables and the dependent variables can be determined efficiently from this assignment.[30]

We can attempt to explain the in practice performance of many SAT solvers by understanding the effect of having many dependent variables on the complexity of SAT and by attempting to verify that having many dependent variables is common. To do this we need a formalism that captures the idea of there being a set of independent variables and a set of dependent variables that can be derived quickly from the independent variables. Work by Williams et al. does exactly this [77].

A brief mention of notation, for a SAT instance F and a partial assignment α_S to a subset of the variables S, let $F[\alpha]$ denote the simplified version of F under the assignment α_S. A backdoor set can then be defined as follows:

Definition A Backdoor set is a subset of variables S such that there exists an assignment α_S of the variables in S and $F[\alpha_S]$ is solvable by a "sub solver". [77]

Definition A sub solver A is an algorithm which has the following properties [77]:

- Given a SAT instances, A either refuses the instance or correctly determines that it is satisfiable or unsatisfiable.
- A runs in polynomial time.
- If for a SAT instance F, A does not refuse F, then for all assignments to the variables α, A does not refuse $F[\alpha]$

An example sub solver could be a solver for 2-SAT [1], q-HORN [10] or a modified unit propagation procedure [21].

We can see that the backdoor set captures our notion of the independent variables that once assigned allow the dependent variables to be deduced via some sub solver that is efficient.

5.3.2 Complexity for Instances with Small Backdoors

For a given instance F with n variables, if we knew that this instance had a backdoor set S, we could simply try all the partial assignments to S, running the sub solver at each step. Since the sub solver runs in polynomial time, then this would take time $\mathcal{O}^*(2^{|S|})$. So for small S this would be asymptotically faster than brute force search or backtracking search. Ideally, if we had that $|S|$ was $\mathcal{O}(\log n)$ then we could solve satisfiability in polynomial time.

However, this depends on us knowing what the backdoor set S is, which is rarely the case. Therefore, we need to account for the additional complexity for finding the backdoor set. Luckily Williams et al. define a deterministic algorithm that for sufficiently small backdoors finds a backdoor and solves the SAT instance asymptotically faster than $\mathcal{O}(2^n)$

The deterministic algorithm can be seen in Algorithm 6. It is analogous to iterative deepening search: checking all possible subsets in increasing order of size and then all assignments to the variables in those subsets. If any assignment simplifies F to the point where $F[\alpha_S]$ is not refused by A then we can solve the rest of the instance using A. If A finds that $F[\alpha_S]$ is satisfiable, then so is F.

To analyse the complexity of Algorithm 6 first let the size of the smallest backdoor be upper bounded by the function $B(n)$. We assume that $B(n) \leq \frac{n}{2}$. Since A runs in $poly(n)$ by definition we can bound the total running time $T(n)$ by the following.

$$T(n) \leq poly(n) \sum_{i=1}^{B(n)} \binom{n}{i} 2^i \tag{5.6}$$

Algorithm 6: Deterministic Solver

Input: CNF Formula F with $|V| = n$ variables and m clauses, sub solver A
Output: SAT or UNSAT

```
for i ∈ 1, 2, ..., n do
    for S ∈ {S ⊆ V : |S| = i} do
        for α_S ∈ {True, False}^|S| do
            if F[α_S] is not refused by A then
                if A(F[α_S]) = SAT then
                    return SAT
return UNSAT
```

Following from our assumption that $B(n) \leq \frac{n}{2}$ then we recognise that

$$\sum_{i=1}^{B(n)} \binom{n}{i} \leq n \cdot \binom{n}{B(n)}$$

and

$$\sum_{i=1}^{B(n)} 2^i \leq 2 \cdot 2^{B(n)}$$

Therefore by combining this with Equation 5.6 we obtain the following upper bound

$$\begin{aligned}
poly(n) \sum_{i=1}^{B(n)} \binom{n}{i} 2^i &\leq poly(n) \Big(\sum_{i=1}^{B(n)} \binom{n}{i} \Big) \Big(\sum_{i=1}^{B(n)} 2^i \Big) \\
&\leq poly(n) \cdot n \cdot \binom{n}{B(n)} \cdot 2 \cdot 2^{B(n)} \\
&\leq poly(n) \binom{n}{B(n)} 2^{B(n)} \\
&\leq poly(n) \frac{n^{B(n)}}{B(n)^{\frac{B(n)}{2}}} 2^{B(n)} \qquad (5.7) \\
&\leq poly(n) \left(\frac{2n}{\sqrt{B(n)}} \right)^{B(n)} \qquad (5.8)
\end{aligned}$$

We get arrive at Equation 5.7 by realising that for all $n \in \mathbb{N}$ we have that $B(n)! \geq B(n)^{B(n)/2}$ and that $n!/(n - B(n))! \leq n^{B(n)}$

If we consider the case where the size of the smallest backdoor is logarithmic in n (i.e. $B(n) = \log n$), then Williams et al. derive the complexity of Algorithm 6 to

be the following [77]:

$$\left(\frac{n}{\sqrt{\mathcal{O}(\log n)}}\right)^{\mathcal{O}(\log n)} \tag{5.9}$$

Which is asymptotically faster than $\mathcal{O}(2^n)$.

5.3.3 Improved Upper Bounds

However, we note that this can be improved slightly. In Equation 5.7, Williams et al. use the fact that $\forall n \in \mathbb{N} : n! \geq n^{n/2}$. However we state the following lower bound for the factorial:

$$\forall k \in (1, \infty) : \exists \delta \in \mathbb{R} : \forall n \in \mathbb{N} : \delta n! \geq n^{n/k} \tag{5.10}$$

Proof. Applying Stirling's approximation of $n!$ we can restate the bound as

$$n^{n/k} \leq \delta\sqrt{2\pi}n^{n+\frac{1}{2}}e^{-n}$$

Rearranging we derive the following:

$$n^{n/k-n-\frac{1}{2}} \leq \delta\sqrt{2\pi}e^{-n}$$

Taking logs on both sides, we see that

$$(n/k - n - \frac{1}{2})\log n \leq \log(\delta\sqrt{2\pi}) - n$$

Rearranging again

$$\left(\frac{1}{k} - 1\right)n\log n + n - \frac{1}{2}\log n \leq \log(\delta\sqrt{2\pi}) \tag{5.11}$$

It is clear that if $k > 1$ then $(\frac{1}{k} - 1) < 0$, which means that the $n\log n$ term dominates as $n \to \infty$. Hence the LHS of Equation 5.11 has a finite maximum. To complete the proof we simply choose δ to be

$$\delta = \max_{n\in\mathbb{N}} \left\{ \frac{1}{\sqrt{2\pi}} \exp\left\{ \left(\frac{1}{k} - 1\right)n\log n + n - \frac{1}{2}\log n \right\} \right\} \tag{5.12}$$

□

We can then use the Inequality 5.10 and 5.7 to derive a new upper bound on the running time of Algorithm 6 for all $k > 1$.

$$T(n) \leq poly(n)\left(\frac{2\delta n}{B(n)^{1/k}}\right)^{B(n)} \tag{5.13}$$

So if we again consider the case where $B(n) = \mathcal{O}(\log n)$, then for all $k > 1$ the time complexity is

$$\left(\frac{n}{\mathcal{O}(\log n)^{1/k}}\right)^{\mathcal{O}(\log n)} \tag{5.14}$$

Which, for $1 < k < 2$ is an improvement over the bound achieved by Williams et al.

5.3.4 Empirical Results

Williams et al. also state that for $B(n) = \frac{n}{4.404}$ Algorithm 6 runs in time $\mathcal{O}(c^n)$ for some $c < 2$. Hence, SETH implies that there must exist some infinite set of SAT instances where the backdoor size is larger than $\frac{n}{4.404}$. Granted, this tells us nothing about the proportion of SAT instances that do have small backdoor sets, although we should perhaps question whether it is reasonable to expect to find small backdoors reliably.

However, Williams et al. use a modified version of Satz-rand [42] to effectively implement Algorithm 6 and experimentally verify the existence of small backdoors for a few benchmark instances. The results of applying this to a few SAT benchmarks can be seen in Table 5.1. We can see that these practical instances have backdoor sets that are much smaller than their total number of variables. This provides support to the idea that the performance of modern SAT solvers comes in part from their ability to exploit small backdoor sets in practical instances.

However, it is not clear from Table 5.1 that we can expect to find small backdoors in all SAT instances. In particular if we consider the instances with the smallest fractional backdoor set sizes we see that they either have largest clause ratios or smallest clause ratios. This means that, although we do not know the exact phase transition for these instances, they likely lie far from the phase transition and therefore we already expect them to be easier (see Figure 5.3).

More work is needed to examine if there are small backdoor sets at the point of the phase transition and to examine potential links between the distance from the phase transition and sizes of the smallest backdoor sets.

instance name	#vars	#clauses	backdoor size	fraction	clause ratio
logistics.d	6783	437431	12	0.0018	64.5
3bitadd_32	8704	32316	53	0.0061	3.71
pipe_01	7736	26087	23	0.0030	3.37
qg_30_1	1235	8523	14	0.0113	6.90
qg_35_1	1597	10658	15	0.0094	6.67

Table 5.1 Sizes of Backdoors in SAT benchmark instances [77]

5.3.5 Parameterization by Backdoor Size

Another approach to attempting to quantify the effect of backdoors is to consider the complexity of SAT, parameterized by the size of the backdoor. This can potentially be a little confusing since we already usually express the complexity in terms of the parameter n and will now be introducing a different parameter k.

Ideally what we want is an algorithm that is in FPT, so if k is the size of the backdoor and x is the SAT instance then the algorithm runs in $\mathcal{O}(f(k)|x|^c)$ for some constant $c \in \mathbb{R}^+$ that does not depend on k and some computable function f. If we consider the complexity of Algorithm 6 and its complexity in Inequality 5.13, if we try to reformulate this complexity into the FPT form we will see that we are unable to get the constant c to be independent of k (which is referred to as $B(n)$ in Inequality 5.13).

Work by Nishimura et al. [61] and Gaspers et al. [28] show that detecting backdoors is $W[2]$-HARD for a range of popular sub solvers. Therefore, since Algorithm 6 is general for all sub solvers it is also at least $W[2]$-HARD. So unless FPT= $W[2]$ then we should not expect to find FPT algorithms for backdoor sets.

However, Nishimura et al. also show that with sub solvers for 2-SAT and Horn formulas, strong backdoor sets can be found in FPT time and that with sub solvers for q-horn [10] finding strong backdoor sets is $W[2]$-HARD [61]. However, work by Ramanujan et al. show that for a q-Horn sub solver deletion backdoors can also be found in FPT time [64].

Definition A strong backdoor, for a sub solver A and a SAT instance F with a variable set V, is a subset $S \subseteq V$ such that for all assignments α_S to the variables in S $F[\alpha_S]$ is not refused by the sub solver A.

Definition A deletion backdoor, for a sub solver A and a SAT instance F with a variable set V, is a subset $S \subseteq V$ such that $F_S = \bigcup_{C \in F}\{\{l \in C : vars(l) \notin S\}\}$ is not refused by A.

Sub solver	weak backdoor	strong backdoor	deletion backdoor
Horn	$W[2]$-HARD	FPT $\mathcal{O}(2^k\|x\|)$	FPT $\mathcal{O}(2^k\|x\|)$
2-CNF	$W[2]$-HARD	FPT $\mathcal{O}(3^k\|x\|)$	FPT $\mathcal{O}(3^k\|x\|)$
q-Horn	$W[2]$-HARD	$W[2]$-HARD	FPT $\mathcal{O}(12^k k^5\|x\|)$

Table 5.2 Summary of FPT results for different backdoors and sub solvers [61, 64, 28]

An important observation is that for the same sub solver A, every deletion backdoor is also a strong backdoor. This follows from the fact that for any assignment to a variable $v \in S$, either all clauses including a positive literal of v are removed and all instances of $\neg v$ are removed or all clauses including $\neg v$ are remove and all instances of the positive literal v are removed. So for either assignment to v all positive and negative occurrences of v are removed from F. So therefore for any assignment α_S to a deletion backdoor S, $F[\alpha_S] \subseteq F_S$. So any sub solver not refusing F_S must also not refuse $F[\alpha_S]$ for all α_S. Hence, any algorithm that detects strong backdoors can also be easily modified to detect deletion backdoors.

However, a drawback of deletion backdoors is that the smallest deletion backdoor can be arbitrarily larger than the smallest strong backdoor. The following example from Gaspers et al. illustrates this.

$$F = \bigcup_{i=1}^{n} \{\{x_{i,1}, x_{i,2}, x_{i,3}, a\}, \{\neg x_{i,1}, \neg x_{i,2}, \neg x_{i,3}, \neg a\}\} \tag{5.15}$$

If we consider the sub solver A to the application of the pure literal rule from DP [21], then we can see that $\{a\}$ is a strong backdoor, but there is no deletion backdoor set smaller than n.

If we find a deletion backdoor set of size k, we can solve the SAT instance in time $\mathcal{O}(2^k|x|^c)$ by checking all assignments to the deletion backdoor. What remains is to find a deletion backdoor in FPT time. A summary of of the results by Nishimure et al., Ramanujan et al. and Gaspers et al. can be seen in Table 5.2. We see that as the "power" of the sub solver increases, the difficulty of finding a backdoor increases.

5.4 Conclusions and Further Work

In Chapter 2 we covered a few of the most prominent algorithms and their corresponding solvers. However, in the worst case these solvers could take exponential time in the number of variables in the input. This gives the impression that SAT

is infeasible to solve in practice and results covered in Chapter 3 show that the exact worst case complexity of SAT is related to the exact complexity of other NP-COMPLETE problems and also related to the exact complexity of some problems in P. This leads us to believe that improving the worst case complexity of SAT is a challenging problem, albeit, perhaps less challenging than showing that the complexity cannot be improved.

However, in this Chapter we see that the worst case behaviour can be confined to a specific region of the input space, namely where the clause ratio is close to phase transition. Due to the sharpness of the phase transition, large SAT instances that are not close to the phase transition will with high probability either be satisfiable or unsatisfiable, depending on the side of the phase transition that the instance is on.

Furthermore, we see that the complexity can be improved if we can guarantee the existence of small backdoor sets and finding certain types of backdoor sets is fixed parameter tractable. Additionally, empirical studies find that small backdoor sets are often prevalent in industrial SAT instances, providing another explanation for why they can be solved quickly. It is not known whether the prevalence of small backdoor sets and the distance from the phase transition point are related and this could be explored in further work.

Therefore, taking these two aspects into account it becomes clearer why many industrial SAT instances can be solved significantly faster than their worst case complexities would suggest. Since all NP-COMPLETE problems can be reduced to SAT, this also suggests that many other NP-COMPLETE problems exhibit the same behaviour. Thus, one practical technique for dealing with NP-COMPLETE problems in practice would be to reduce the problem to SAT and then use a SAT solver similar to one discussed in Chapter 2.

References

[1] Aspvall, B., Plass, M. F., and Tarjan, R. E. (1979). A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123.

[2] Audemard, G. and Simon, L. (2009a). Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8.

[3] Audemard, G. and Simon, L. (2009b). Predicting learnt clauses quality in modern sat solvers. In *Twenty-first International Joint Conference on Artificial Intelligence*.

[4] Audemard, G. and Simon, L. (2018). On the glucose sat solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001.

[5] Backurs, A. and Indyk, P. (2015). Edit distance cannot be computed in strongly subquadratic time (unless seth is false). In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 51–58. ACM.

[6] Balyo, T., Heule, M. J., and Järvisalo, M. (2017). Proceedings of sat competition 2017: Solver and benchmark descriptions.

[7] Bayardo Jr, R. J. and Schrag, R. (1997). Using csp look-back techniques to solve real-world sat instances.

[8] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer.

[9] Biere, A., Heule, M., van Maaren, H., and Walsh, T. (2009). Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153.

[10] Boros, E., Hammer, P. L., and Sun, X. (1994). Recognition of q-horn formulae in linear time. *Discrete Applied Mathematics*, 55(1):1–13.

[11] Bringmann, K. (2014). Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless seth fails. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 661–670. IEEE.

[12] Bringmann, K. and Künnemann, M. (2015). Quadratic conditional lower bounds for string problems and dynamic time warping. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*, pages 79–97. IEEE.

[13] Bringmann, K. and Künnemann, M. (2018). Multivariate fine-grained complexity of longest common subsequence. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1216–1235. Society for Industrial and Applied Mathematics.

[14] Cheeseman, P. C., Kanefsky, B., and Taylor, W. M. (1991). Where the really hard problems are. In *IJCAI*, volume 91, pages 331–337.

[15] Chen, P. and Keutzer, K. (1999). Towards true crosstalk noise analysis. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 132–138. IEEE Press.

[16] Chvátal, V. and Reed, B. (1992). Mick gets some (the odds are on his side)(satisfiability). In *Proceedings., 33rd Annual Symposium on Foundations of Computer Science*, pages 620–627. IEEE.

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

[18] Crawford, J. M. and Auton, L. D. (1993). Experimental results on the crossover point in satisfiability problems. In *AAAI*, volume 93, pages 21–27. Citeseer.

[19] Darwiche, A. (2004). New advances in compiling cnf into decomposable negation normal form. In *Proc. ECAI-04: 16th European Conf. on Artificial Intelligence*, pages 328–332.

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

[21] Davis, M. and Putnam, H. (1960). A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215.

[22] Dechter, R. (1990). Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312.

[23] Eén, N. and Sörensson, N. (2003). An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer.

[24] Flum, J. and Grohe, M. (2006). Parameterized complexity theory. page 427.

[25] Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., and Zankl, H. (2007). Sat solving for termination analysis with polynomial interpretations. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 340–354. Springer.

[26] Garey, M. R., Johnson, D. S., and Stockmeyer, L. (1974). Some simplified np-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM.

[27] Garey, M. R., Johnson, D. S., and Tarjan, R. E. (1976). The planar hamiltonian circuit problem is np-complete. *SIAM Journal on Computing*, 5(4):704–714.

[28] Gaspers, S., Ordyniak, S., Ramanujan, M., Saurabh, S., and Szeider, S. (2016). Backdoors to q-horn. *Algorithmica*, 74(1):540–557.

[29] Gent, I. P. and Walsh, T. (1994). The sat phase transition. In *ECAI*, volume 94, pages 105–109. PITMAN.

[30] Gerevini, A. and Serina, I. (2003). Planning as propositional csp: from walksat to local search techniques for action graphs. *Constraints*, 8(4):389–413.

[31] Goerdt, A. (1996). A threshold for unsatisfiability. *Journal of Computer and System Sciences*, 53(3):469–486.

[32] Goldberg, E. and Novikov, Y. (2007). Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561.

[33] Gomes, C. P., Selman, B., Crato, N., and Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100.

[34] Gomes, C. P., Selman, B., Kautz, H., et al. (1998). Boosting combinatorial search through randomization.

[35] Heule, M. J., Järvisalo, M. J., Suda, M., et al. (2018). Proceedings of sat competition 2018.

[36] Hirsch, E. A. (2000). New worst-case upper bounds for sat. *Journal of Automated Reasoning*, 24(4):397–420.

[37] Hofmeister, T., Schöning, U., Schuler, R., and Watanabe, O. (2002). A probabilistic 3—sat algorithm further improved. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 192–202. Springer.

[38] Hoos, H. H. et al. (2002). An adaptive noise mechanism for walksat. In *AAAI/IAAI*, pages 655–660.

[39] Impagliazzo, R. and Paturi, R. (2001). On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367–375.

[40] Impagliazzo, R., Paturi, R., and Zane, F. (2001). Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530.

[41] Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

[42] Kautz, H. and Selman, B. (1999). Unifying sat-based and graph-based planning. In *IJCAI*, volume 99, pages 318–325.

[43] Kautz, H. A., Selman, B., et al. (1992). Planning as satisfiability. In *ECAI*, volume 92, pages 359–363. Citeseer.

[44] Khurshid, S. and Marinov, D. (2004). Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, 11(4):403–434.

[45] Kirkpatrick, S., Györgyi, G., Tishby, N., and Troyansky, L. (1994). The statistical mechanics of k-satisfaction. In *Advances in Neural Information Processing Systems*, pages 439–446.

[46] Kullmann, O. (2009). *Theory and Applications of Satisfiability Testing-SAT 2009: 12th International Conference, SAT 2009, Swansea, UK, June 30-July 3, 2009. Proceedings*, volume 5584. Springer.

[47] Larrabee, T. (1992). Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15.

[48] Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.

[49] Lewis, M. D., Schubert, T., and Becker, B. W. (2005). Speedup techniques utilized in modern sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 437–443. Springer.

[50] Lipton, R. J. and Tarjan, R. E. (1977). Applications of a planar separator theorem. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 162–170. IEEE.

[51] Lipton, R. J. and Tarjan, R. E. (1979). A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189.

[52] Lokshtanov, D., Marx, D., Saurabh, S., et al. (2013). Lower bounds based on the exponential time hypothesis. *Bulletin of EATCS*, 3(105).

[53] Lynce, I. and Marques-Silva, J. (2006). Efficient haplotype inference with boolean satisfiability. In *National Conference on Artificial Intelligence (AAAI)*. AAAI Press.

[54] Manolios, P., Oms, M. G., and Valls, S. O. (2007). Checking pedigree consistency with pcs. In Grumberg, O. and Huth, M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 339–342, Berlin, Heidelberg. Springer Berlin Heidelberg.

[55] Marques-Silva, J. (2008). Practical applications of boolean satisfiability. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pages 74–80. IEEE.

[56] Marques-Silva, J., Lynce, I., and Malik, S. (2009). Conflict-driven clause learning sat solvers. In *Volume 185: Handbook of Satisfiability*, pages 131–153.

[57] Marques-Silva, J. P. and Sakallah, K. A. (1999). Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521.

[58] McGeer, P. C., Saldanha, A., Stephan, P. R., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1991). Timing analysis and delay-fault test generation using path-recursive functions. In *IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, pages 180–183. IEEE.

[59] Mitchell, D., Selman, B., and Levesque, H. (1992). Hard and easy distributions of sat problems. In *AAAI*, volume 92, pages 459–465.

[60] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM.

[61] Nishimura, N., Ragde, P., and Szeider, S. (2004). Detecting backdoor sets with respect to horn and binary clauses. *SAT*, 4:96–103.

[62] Papadimitriou, C. H. (1991). On selecting a satisfying truth assignment. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 163–169. IEEE.

[63] Polak, A. (2018). Why is it hard to beat o (n2) for longest common weakly increasing subsequence? *Information Processing Letters*, 132:1–5.

[64] Ramanujan, M. and Saurabh, S. (2017). Linear-time parameterized algorithms via skew-symmetric multicuts. *ACM Transactions on Algorithms (TALG)*, 13(4):46.

[65] Ryan, L. (2004). *Efficient algorithms for clause-learning SAT solvers*. PhD thesis, Theses (School of Computing Science)/Simon Fraser University.

[66] Schaefer, T. J. (1978). The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226. ACM.

[67] Schoning, T. (1999). A probabilistic algorithm for k-sat and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 410–414. IEEE.

[68] Selman, B., Kautz, H. A., and Cohen, B. (1994). Noise strategies for improving local search. In *AAAI*, volume 94, pages 337–343.

[69] Selman, B., Kautz, H. A., Cohen, B., et al. (1993). Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability*, 26:521–532.

[70] Selman, B., Levesque, H. J., Mitchell, D. G., et al. (1992). A new method for solving hard satisfiability problems. In *AAAI*, volume 92, pages 440–446. Citeseer.

[71] Smith, A., Veneris, A., Ali, M. F., and Viglas, A. (2005). Fault diagnosis and logic debugging using boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(10):1606–1621.

[72] Topsok, F. (2006). Some bounds for the logarithmic function. *Inequality theory and applications*, 4:137.

[73] Tucker, C., Shuffelton, D., Jhala, R., and Lerner, S. (2007). Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th international conference on Software Engineering*, pages 178–188. IEEE Computer Society.

[74] Turner, J. S. (1988). Almost all k-colorable graphs are easy to color. *Journal of algorithms*, 9(1):63–82.

[75] Vassilevska Williams, V. (2015). Hardness of easy problems: Basing hardness on popular conjectures such as the strong exponential time hypothesis (invited talk). In *10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[76] Williams, R. (2004). A new algorithm for optimal constraint satisfaction and its implications. In *International Colloquium on Automata, Languages, and Programming*, pages 1227–1237. Springer.

[77] Williams, R., Gomes, C. P., and Selman, B. (2003). Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, pages 1173–1178, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.