Algorithms for the Satisfiability Problem

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herrn Dipl.-Inf. Daniel Rolf
geboren am 5.1.1979 in Neuruppin

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Wolfgang Coy

Gutachter:

1. Prof. Dr. Martin Grohe
2. Prof. Dr. Stephan Kreutzer
3. Prof. Dr. Walter Kern

eingereicht am:	30. Mai 2006
Tag der mündlichen Prüfung:	17. November 2006

Abstract

This work deals with worst-case algorithms for the satisfiability problem regarding boolean formulas in conjunctive normal form. The main part of this work consists of the analysis of the running time of three different algorithms, two for 3-SAT and one for Unique-k-SAT.

Research on the satisfiability problem has made reasonable progress during the last years. After the introduction in Chapter 1, we will study some interesting algorithms and their running time bounds in Chapter 2.

In Chapter 3, we establish a randomized algorithm that finds a satisfying assignment for a satisfiable 3-CNF formula G on n variables in $\mathcal{O}(1.32793^n)$ expected running time. The algorithm is based on the analysis of so-called *strings*, which are sequences of clauses of size three, whereby non-succeeding clauses do not share a variable, and succeeding clauses share one or two variables. If there are not many strings, it is likely that we already encounter a solution of G while searching for strings. In 1999, Schöning proved a bound of $\mathcal{O}(1.3334^n)$ for 3-SAT. If there are many strings, we use them to improve the running time of this algorithm.

Furthermore, in Chapter 4, we derandomize the PPSZ algorithm for Unique-k-SAT. The PPSZ algorithm presented by Paturi, Pudlak, Saks, and Zane in 1998 has the feature that the solution of a uniquely satisfiable 3-CNF formula can be found in expected running time at most $\mathcal{O}(1.3071^n)$. In general, we will obtain a derandomized version of the algorithm for Unique-k-SAT that has essentially the same bound as the randomized version. This settles the currently best known deterministic worst-case bound for the Unique-k-SAT problem. We apply the 'Method of Small Sample Spaces' in order to derandomize the algorithm.

Finally, in Chapter 5, we improve the bound for the algorithm of Iwama and Tamaki to get the currently best known randomized worst-case bound for the 3-SAT problem of $\mathcal{O}(1.32216^n)$. In 2003 Iwama and Tamaki combined Schöning's and the PPSZ algorithm to yield an $\mathcal{O}(1.3238^n)$ bound. We tweak the bound for the PPSZ algorithm to get a slightly better contribution to the combined algorithm.

Keywords:
Satisfiability problem, k-SAT, algorithms, worst case bounds

Zusammenfassung

Diese Arbeit befasst sich mit Worst-Case-Algorithmen für das Erfüllbarkeitsproblem boolescher Ausdrücke in konjunktiver Normalform. Im Wesentlichen betrachten wir Laufzeitschranken drei verschiedener Algorithmen, zwei für 3-SAT und einen für Unique-k-SAT.

In der Forschung zum Erfüllbarkeitsproblem gab es in den letzten Jahren eine Reihe von Fortschritten. Nach der Einleitung in Kapitel 1 werden in Kapitel 2 einige interessante Algorithmen für das k-SAT-Problem und ihre Laufzeitschranken behandelt.

In Kapitel 3 entwickeln wir einen randomisierten Algorithmus, der eine Lösung eines erfüllbaren 3-CNF-Ausdrucks G mit n Variablen mit einer erwarteten Laufzeit von $\mathcal{O}(1.32793^n)$ findet. Der Algorithmus basiert auf der Analyse sogenannter *Strings*, welche Sequenzen von Klauseln der Länge drei sind. Dabei dürfen aufeinander folgende Klauseln keine Variablen und nicht aufeinander folgende Klauseln ein oder zwei Variablen gemeinsam haben. Gibt es wenige Strings, so treffen wir wahrscheinlich bereits während der String-Suche auf eine Lösung von G. 1999 entwarf Schöning einen Algorithmus mit einer Schranke von $\mathcal{O}(1.3334^n)$ für 3-SAT. Viele Strings erlauben es, die die Laufzeit dieses Algorithmusses zu verbessern.

Weiterhin werden wir in Kapitel 4 den PPSZ-Algorithmus für Unique-k-SAT derandomisieren. Der 1998 von Paturi, Pudlak, Saks und Zane vorgestellte PPSZ-Algorithmus hat die besondere Eigenschaft, dass die Lösung eines eindeutig erfüllbaren 3-CNF-Ausdrucks in höchstens $\mathcal{O}(1.3071^n)$ erwarteter Laufzeit gefunden wird. Die derandomisierte Variante des Algorithmusses für Unique-k-SAT hat im Wesentlichen die gleiche Laufzeitschranke wie die randomisierte Variante. Wir erreichen damit die momentan beste deterministische Worst-Case-Schranke für Unique-k-SAT. Um den Algorithmus zu derandomisieren, wenden wir die „Methode der kleinen Zufallsräume“ an.

Schließlich verbessern wir in Kapitel 5 die Schranke für den Algorithmus von Iwama und Tamaki. 2003 kombinierten Iwama und Tamaki den PPSZ-Algorithmus mit Schönigs Algorithmus und konnten eine Schranke von $\mathcal{O}(1.3238^n)$ bewiesen. Um seinen Beitrag zum kombinierten Algorithmus zu steigern, justieren wir die Schranke des PPSZ-Algorithmusses. Damit erhalten wir die momentan beste randomisierte Worst-Case-Schranke für das 3-SAT-Problem von $\mathcal{O}(1.32216^n)$.

Schlagwörter:
Erfüllbarkeitsproblem, k-SAT, Algorithmen, obere Schranken

Acknowledgments

Firstly, I would like to thank my supervisor, Martin Grohe, for listening to, reading, and verifying my ideas. I enjoyed the warm environment he and his/my colleagues created at the chair of Logic in Computer Science.

Moreover, I thank Deryk Osthus for introducing the SAT-topic to me about four years ago and for supervising my Diploma Thesis and the first part of my doctoral research.

To my dear pals, my colleagues at the university and at work, my family, and my great parents-in-law: Thank you guys, thank you!

But most of all, I am in great debt of gratitude to my beloved wife Linda, who has always supported me with her love and friendship, and who has given birth to our little daughter, Isabell Maja, in this May.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Preliminaries

Firstly, we make some common definitions. An assignment β to a set of variables V maps each variable in V to 0 or 1. For two assignments β_1, β_2 to V, let $dist(\beta_1, \beta_2)$ denote the hamming distance of β_1 and β_2, i.e. the number of variables in V for which β_1 and β_2 take different values. A *literal* is a variable or its negation. Such a literal l is satisfied by β when it is true that $\beta(l) = 1$ if l is not negated resp. $\beta(\bar{l}) = 0$ if l is negated. A *clause* is a set of literals based on different variables. A clause is satisfied by some assignment β if at least one literal is satisfied by β. Moreover, a *formula in conjunctive normal form (CNF)* is a set of clauses, which is satisfied by β if each clause is satisfied by β. A *k-clause* is a clause of size k, and a *k-CNF formula* is a set of clauses where each has size k. Also, an 1-clause is known as *unit clause*, whereas the empty clause is denoted by $\perp$, which is not satisfiable. Two clauses are called *independent* if they do not have any variables in common. For a set of clauses G, let $vars(G)$ be the set of variables occurring in G. Finally, let n_G stand for $|vars(G)|$. Note that an empty k-CNF formula is satisfiable, but a k-CNF formula containing $\perp$ is not satisfiable.

We will not consider polynomial factors in complexity calculations because we always expect an exponential expression which outweighs all polynomials for large problems. Because the number of clauses is $\mathcal{O}\left(n_G^k\right)$, polynomials that depend on the number of clauses can also be replaced by some polynomial in n_G.

For a CNF formula G and a literal l, we denote with $G|_l$ the formula obtained by making l true in G, i.e. we remove all clauses that contain l and remove $\bar{l}$ from all clauses that contain it. Let $L = \{l_1, .., l_s\}$ be a finite set of literals of distinct variables. Then $G|_L$ is defined as $G|_{l_1}|...|_{l_s}$. Clearly, a satisfying assignment for $G|_L$ can be extended to a satisfying assignment for G by assigning 1 to all literals in L.

A clause pair (C_1, C_2) is a *resolvent pair* if they have only one variable v in common whereby $v \in C_1$ and $\overline{v} \in C_2$. Their *resolvent* $R(C_1, C_2)$ is the clause $(C_1 - v) \cup (C_2 - \overline{v})$. Because any satisfying assignment for C_1 and C_2 must also satisfy $R(C_1, C_2)$, adding $R(C_1, C_2)$ to a CNF formula does not change its set of satisfying assignments.

s-bounded resolution means to add to G all resolvent pairs of clauses in G where the size of the resolvent is at most s, over and over again, until there is nothing more to do. Note that if s is a constant, this has polynomial time and space complexity in n_G.

1.2 The Satisfiability Problem

In general, the *satisfiability problem (SAT)* asks whether a boolean expression is satisfiable or not. A *boolean expression* is made of variables, negations, conjunctions, disjunctions, and parentheses. The problem of deciding satisfiability of a boolean expression in conjunctive normal form is called the *CNF-SAT* problem. CNF-SAT was the first problem that had been shown to be NP-complete. This was done by Cook in 1971, cf. [5].

To decide whether a k-CNF formula G has a satisfying assignment is commonly known as the *k-SAT* problem, which is NP-complete for $k > 2$ because every CNF formula can be reduced to a 3-CNF formula in polynomial time. Hence if $NP \neq P$ holds (which is widely assumed), there is no hope to find a polynomial time algorithm for the k-SAT problem for $k > 2$. While this may be true, 2-SAT can surely be solved in polynomial time.

A closely related problem of particular interest is the Unique-k-SAT problem – decide whether a k-CNF formula G with at most one satisfying assignment is satisfiable –, which is also NP-complete for $k > 2$.

1.3 How Hard Is k-SAT?

Let us define the following sequence:

$$s_k = \inf \left\{ c \middle| \begin{array}{l} \text{there is an } \mathcal{O}\left(2^{c \cdot n}\right) \text{ randomized algorithm that solves} \\ k\text{-SAT on } n \text{ variables} \end{array} \right\}$$

Moreover, let s_∞ be the limit of s_k for k tending to infinity. Little is known about s_k; it is not even known whether $s_\infty < 1$ holds or whether there is some $k > 2$ with $s_k > 0$. $s_\infty = 1$ would mean that for every $c < 1$, there is some k so that the best algorithm for k-SAT has worst case running time $\Theta\left(2^{c \cdot n}\right)$, i.e. the algorithms are doomed to be only 'slightly' faster than the $\mathcal{O}\left(2^n\right)$ naive enumeration approach.

The *Exponential-Time Hypothesis (ETH)* for k-SAT asserts that $s_k > 0$ is true for every $k > 2$, meaning that k-SAT cannot be solved in sub-exponential time in the worst case for $k > 2$.

In [7], Impagliazzo and Paturi could prove the following theorem:

Theorem 1.1. *There is some constant $d > 0$ so that $s_k \leq s_\infty(1 - d/k)$ is true.*

Hence s_k is an infinitely often increasing sequence. Furthermore, the authors also conjectured that s_k is strictly monotone increasing.

Analogously to s_k, we define σ_k for Unique-k-SAT and σ_∞ to be the limit of σ_k for k tending to infinity. Because every instance of the Unique-k-SAT problem is also an instance of the k-SAT problem, $\sigma_k \leq s_k$ holds. But, what is really harder: To find a satisfying assignment for a k-CNF with a single solution or for a k-CNF that has many solutions? A partial answer was given in [4], where Calabro, Impagliazzo, Kabanets, and Paturi proved the following relationship:

Theorem 1.2. *It is true that $s_k \leq \sigma_k + \mathcal{O}\left((\ln^2 k)/k\right)$. Thus $s_\infty = \sigma_\infty$ also holds.*

Roughly speaking, Unique-k-SAT can be only a little bit easier than k-SAT for large k. In addition, the authors conjectured that $s_k = \sigma_k$ holds for all $k \geq 3$ and even expressed their 'hope' that it could be possible to prove that finding a solution of a formula with many satisfying assignments is easier than for a formula with few satisfying assignments. In contrast, the currently best known bound for 3-SAT depends on the number of solutions, being worst when there are about $2^{0.028 \cdot n_G}$ satisfying assignments for a 3-CNF formula G, cf. Figure 5.3 in Section 5.2.5.

1.4 Record Breaking

c	Type	Section	Year	Reference
1.861	det.	Section 2.2.2	1997	[12]
1.849	det.		1998	[16]
1.618	det.	Section 2.2.1	1985	[10]
1.588	rand.	Section 2.2.2	1997	[12]
1.579	det.	Section 2.2.1	1993	[21]
1.505	det.	Section 2.2.1	1999	[9]
1.5	rand.		1999	[25]
1.497	det.	Section 2.2.1	1996	[22]
1.481	det.	Section 2.3.3	2002	[6]
1.473	det.	Section 2.3.3	2004	[3]
1.363	rand.	Section 2.2.3	1998	[13]
1.334	rand.	Section 2.3.2	1999	[24]
1.3302	rand.	Section 2.4.1	2002	[23]
1.32971	**rand.**	**Section 2.4.1**	**2003**	**[17]**
1.329	rand.	Section 2.4.1	2003	[2]
1.32793	**rand.**	**Chapter 3**	**2003**	**[18]**
1.3238	rand.	Section 2.4.2	2003	[8]
1.32216	**rand.**	**Chapter 5**	**2005**	**[20]**

Table 1.1: Running Time Bounds $\mathcal{O}(c^n)$ for 3-SAT on n Variables

Recall that already 3-SAT is NP-complete, and since k-SAT is 'getting harder' with increasing k, the best bounds are expected for $k = 3$. The evolution of running time bounds for algorithms for 3-SAT on n variables in terms of $\mathcal{O}(c^n)$ is given in Table 1.1. Algorithms that are mentioned in more detail in this work are found where column 'Section' states. Bounds in bold lines are due to the author of this work.

The proofs of the $\mathcal{O}(1.32793^n)$ and $\mathcal{O}(1.32216^n)$ bounds are results of this work, presented in Chapter 3 resp. Chapter 5.

The graphs in Figure 1.1 show c^n for every c in Table 1.1, i.e. they give a rough estimate of these bounds with respect to the number of variables.

Assuming that the bounds were tight and sub-exponential factors in $\mathcal{O}(c^n)$ neglectable, we can compare the performance of the trivial $\mathcal{O}(2^n)$ algorithm and the best known, $\mathcal{O}(1.32216^n)$ algorithm for 3-SAT. Then the $\mathcal{O}(1.32216^n)$ algo-

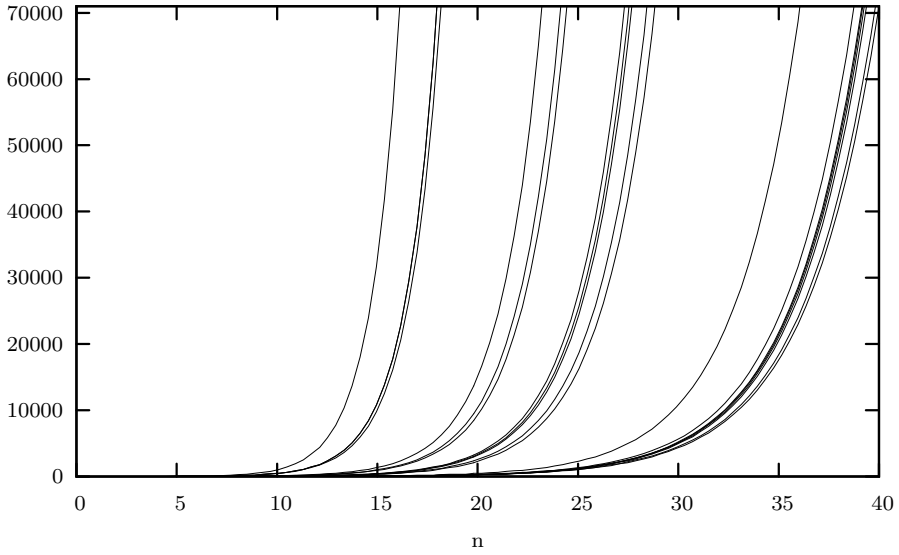

Figure 1.1: Running Time Bounds for n variables

rithm would solve 3-SAT instances with 2.5 times the number of variables in the same time than the trivial algorithm could. Note that doubling the speed of computers would yield only a summand to the number of variables, i.e. if the trivial algorithm could process n variables, a computer two times faster would process $n+1$ variables in the same time. In contrast, the gain by providing exponentially faster algorithms is much bigger than what which we gain by Moore's Law from year-to-year!

A slightly different situation occurs for Unique-k-SAT. In [13], Paturi, Pudlak, Saks, and Zane proved that for a uniquely satisfiable 3-CNF formula G, the solution can be found in $\mathcal{O}\left(1.3071^{n_G}\right)$ expected running time at most. This is the best randomized bound known for Unique-3-SAT. But paradoxically, the bound gets worse when the number of solutions increases. Alas, for the general 3-SAT and 4-SAT case on n variables, this algorithm achieves expected running time bounds of $\mathcal{O}\left(1.362^n\right)$ resp. $\mathcal{O}\left(1.476^n\right)$ only. However, in [19], the author showed that this algorithm can be derandomized yielding $\mathcal{O}\left(1.3071^n\right)$ deterministic running time for Unique-3-SAT on n variables. Essentially, the bounds for Unique-k-SAT obtained by the derandomization are (almost) the same bound as for the randomized version. Thus making them the best known deterministic bound for Unique-k-SAT. The proof is a result of this work and presented in Chapter 4.

1.5 Further Research

At present, we have two different approaches rivaling for k-SAT. The deterministic algorithms are based entirely on local search, whereas the randomized ones use a combination of local search and unit clause propagation after bounded resolution. The currently known best deterministic algorithm for Unique-k-SAT relies solely on unit clause propagation after bounded resolution.

The author believes that a better understanding of the algorithm presented by Paturi, Pudlak, Saks, and Zane in [13] could yield better bounds on its running time, beating the current best bound for general k-SAT.

For 3-SAT and 4-SAT, roughly speaking, the bound for this algorithm provided by Paturi et al is getting worse when the number of satisfying assignments increases. The author conjectures that this can be avoided, i.e. that it is possible to prove that the bounds hold for 3-SAT resp. 4-SAT in general.

Chapter 2

Algorithms for k-SAT

2.1 Introduction

Algorithms for k-SAT can more or less be divided into two classes: Davis-Putnam-style and local search algorithms. In this chapter, we will discuss several interesting algorithms of both classes and, finally, two algorithms which combine both classes.

2.2 Davis-Putnam Algorithms

Davis-Putnam-style algorithms try to iteratively guess the assignment for the formula. At some point where the algorithm has to decide from a set of possible choices, the algorithm branches and recursively calls itself for each branch. When it encounters a contradiction, backtracking is performed.

```
Algorithm DP(CNF G)
1    if ⊥ ∈ G then return false
2    if G = ∅ then return true
3    Choose a variable v
4    if DP(G|_v̄) or DP(G|_v) then return true
5    return false
```

Figure 2.1: Trivial Davis-Putnam Algorithm

In Figure 2.1, a trivial Davis-Putnam Algorithm is shown. Essentially, this algorithm tries all assignments and stops only if the formula is empty or there is a contradiction. If the algorithm encounters an empty formula, the initial formula is

satisfiable. However, if an empty clause is found, the algorithm does not have to further examine assignments to that formula since the empty clause will remain in the formula. In this case, the algorithm backtracks. This behavior is well known as search-tree pruning because the algorithm can be seen as evaluating the search tree, and whenever a contradiction is encountered, the search-tree rooted at the current node is pruned.

However, we cannot guarantee that empty clauses occur, so the worst case running time of that trivial algorithm is $\mathcal{O}\left(poly(|G|) \cdot 2^{n_G}\right)$ for a CNF formula G.

2.2.1 Monien-Speckenmeyer Algorithm

A simple observation is that any k-clause in a CNF formula G has only $2^k - 1$ possible satisfying assignments. Hence we can develop an algorithm that chooses the shortest clause and selects one of the possible assignments. Since we can pick at most $\lceil n_G/k \rceil$ times a clause with at most k literals, it is rather straightforward to prove that this algorithm solves k-SAT in running time at most

$$\mathcal{O}\left(poly(|G|) \cdot \left(2^k - 1\right)^{\lceil n_G/k \rceil}\right),$$

e.g. achieving $\mathcal{O}\left(1.913^{n_G}\right)$ for 3-SAT.

As a different approach, we can select the first literal in the shortest clauses and branch on its assignment instead of reducing all variables in the shortest clause in one step. Assume that we have the shortest clause abc. On the one hand, we assign 1 to a, which makes the clause true and will remove it from the formula. On the other hand, if we assign 0 to a, bc becomes the shortest clause in the new formula. Therefore, define $T_k(n)$ to be the number of steps to solve a k-CNF formula on n variables. By recursively applying the preceding argument, we have $T_k(n) \leq T_k(n-1) + ... + T_k(n-k) + poly(n)$ for $n > k$. k-CNF formulas with at most k variables can be solved in $\mathcal{O}\left(poly(n)\right)$ running time. We have that $T_k(n) \leq \alpha^n \cdot poly(n)$ where α is the largest zero of $1 - x^{-1} - ... - x^{-k}$. For $k = 3$, we obtain a running time of at most $\mathcal{O}\left(1.840^n\right)$.

In [10], Monien and Speckenmeyer extended this approach using autark partial assignments to eliminate variables. A partial assignment to a non-empty subset U of the variables is called *autark* if every clause in G that contains a variable in U is

satisfied (and thus removed) when applying the partial assignment to the formula. If there is some autark partial assignment, it can be found in $\mathcal{O}(poly(n_G))$. Monien and Speckenmeyer observed that when assigning a variable in a CNF formula, an autark partial assignment must exist in the new formula or a clause was shortened. To obtain the Monien-Speckenmeyer algorithm, we try to select and apply an autark partial assignment when the shortest clause has size k, cf. Figure 2.2

Algorithm $MS(k$**-CNF** $G)$
1 **while** the shortest clause in G has size k and there is some autark partial assigment **do**
2 Select and apply some autark partial assignment to G
3 **if** $\bot \in G$ **then return** $false$
4 **if** $G = \emptyset$ **then return** $true$
5 $v :=$ first literal in the shortest clause in G
6 **if** $MS(G|_{\overline{v}})$ or $MS(G|_v)$ **then return** $true$
7 **return** $false$

Figure 2.2: Monien-Speckenmeyer Algorithm

Always, except in the first step, when the shortest clause is a k-clause, there must be an autark partial assignment. Hence, as long as the shortest clause has size k, we can eliminate some variables. Afterwards, the formula contains a clause with size less than k or is empty. Thus the recurrence for this algorithm is $T_k(n) \leq T_k(n-1) + ... + T_k(n-k+1) + poly(n)$ for $n > k$, so $T_k(n) \leq \alpha^n \cdot poly(n)$ where α is the largest zero of $1 - x^{-1} - ... - x^{-k+1}$. We have:

Theorem 2.1. *The algorithm of Monien and Speckenmeyer finds a satisfying assignment for a satisfiable* 3*-CNF formula in running time at most* $\mathcal{O}(1.618^{n_G})$.

By adding more branching-rules for the decision of which variables to assign what values, Schiermeyer claimed bounds of $\mathcal{O}(1.579^{n_G})$ ([21]) and $\mathcal{O}(1.497^{n_G})$ ([22]) for 3-SAT, and Kullmann claimed a bound of $\mathcal{O}(1.5045^{n_G})$ ([9]).

2.2.2 The Algorithm of Paturi, Pudlak, and Zane

In 1997 in [12], Paturi, Pudlak, and Zane published the algorithm shown in Figure 2.3.

Algorithm PPZ(**k-CNF** G)
1 $\beta :=$ assignment to G drawn uniformly at random
2 $\pi :=$ permutation of $vars(G)$ uniformly at random
3 **for each** variable $v \in vars(G)$ ordered by π **do** {
4 **if** G' contains a unit clause v or $\overline{v}$
5 **then** Set the value of v in β to satisfy that unit clause
6 Choose $G := G'|_v$ or $G := G'|_{\overline{v}}$ depending on $\beta(v) = 1$ or $\beta(v) = 0$.
7 }
8 **return** β

Figure 2.3: Algorithm of Paturi, Pudlak, and Zane

For Unique-k-SAT, the analysis of this algorithm is quite simple. We call a clause $C \in G$ a *critical clause* for v if the only true literal in C with respect to β is the one corresponding to v, i.e. flipping the value assigned to v in β would make C instantly false. Let β^* be the satisfying assignment for a uniquely satisfiable k-CNF formula G.

Since for every variable $v \in vars(G)$, $\beta^* \oplus v$ does not satisfy C, there must exist a clause C that is not satisfied by $\beta^* \oplus v$. Thus C is a critical clause for v with respect to β^*. Assume that Algorithm PPZ chooses β^* for β. The crucial observation is that if the variables in C appear before v in π, C is reduced to a unit clause that forces the assignment to v so that C is satisfied. The probability for that to happen is at least $1/k$.

By linearity of expectation, we expect to have at least n_G/k variables forced by unit clauses on the average. For these variables, Algorithm PPZ does not have to guess the right assignment in β because their assignment is fixed up when the algorithm reaches the unit clause. Hence Algorithm PPZ has to guess the right assignment for at most $n_G - n_G/k$ variables on the average.

For this reason, the probability that Algorithm PPZ succeeds is at least $2^{-n_G+n_G/k}$ on the average. So we can conclude that the expected running time of Algorithm PPZ is at most $\mathcal{O}\left(poly(n_G) \cdot 2^{n_G-n_G/k}\right)$. For Unique-3-SAT, this is $\mathcal{O}\left(1.588^{n_G}\right)$.

For k-CNF formulas with more than one solution, we cannot guarantee the existence of critical clauses for all variables. However, if there are many solutions, possibly, there may be more good choices for β. Indeed, Paturi et al proved that

the expected running time of Algorithm PPZ is at most $\mathcal{O}\left(poly(n_G) \cdot 2^{n_G - n_G/k}\right)$ in general for k-SAT.

They also derandomized the algorithm and achieved a deterministic running time of $\mathcal{O}\left(1.861^{n_G}\right)$ for 3-SAT.

2.2.3 The Algorithm of Paturi, Pudlak, Saks, and Zane

In 1998 in [13], Paturi, Pudlak, Saks, and Zane proposed to add s-bounded resolution as a preprocessing step for some large s, hoping that the resolution step produces some more critical clauses. Thus their new algorithm, shown in Figure 2.4, is a simple extension of Algorithm PPZ.

```
Algorithm PPSZ(k-CNF G, integer d, assignment β)
1    G := do k^d-bounded resolution on G
2    π := permutation of vars(G) uniformly at random
3    for each variable v ∈ vars(G) ordered by π do {
4       if G' contains a unit clause v or v̄
5       then Set the value of v in β to satisfy that unit clause
6       Choose G := G'|_v or G := G'|_v̄ depending on β(v) = 1 or β(v) = 0.
7    }
8    return β
```

Figure 2.4: Algorithm of Paturi, Pudlak, Saks, and Zane

Paturi et al proved that this algorithm solves instances of 3-SAT and 4-SAT in $\mathcal{O}\left(1.363^{n_G}\right)$ resp. $\mathcal{O}\left(1.477^{n_G}\right)$ expected running time for large d when an assignment β drawn uniformly at random is passed to the algorithm. For $k \geq 5$, the expected running time can be bounded by

$$\mathcal{O}\left(2^{\left(1-\frac{\mu_k}{k-1}\right)n_G + \epsilon n_G}\right) \text{ with}$$

$$\mu_k = \sum_{j=1}^{\infty} \frac{1}{j\left(j + \frac{1}{k-1}\right)}$$

and some arbitrary small positive ϵ.

Research on this algorithm is a main part of this work in Chapters 4 and 5.

2.3 Local-Search Algorithms

In contrast to Davis-Putnam Algorithms, Local-Search Algorithms start with some assignment and iteratively try to improve it by choosing a new assignment within some small hamming distance.

2.3.1 Papadimitriou's Algorithm

In 1991 in [11], Papadimitriou presented a nice algorithm that runs in polynomial time and solves 2-SAT with probability at least 1/2, cf. Figure 2.5.

```
Algorithm PAPA(2-CNF G)
1    Let β be some arbitrary assignment to G
2    repeat 2n_G^2 times {
3       if β satisfies G then break
4       Select an arbitrary clause C ∈ G that is not satisfied by β
5       Choose a variable in C uniformly at random and flip its value in β
6    }
7    return β
```

Figure 2.5: Papadimitriou's Algorithm

Theorem 2.2. *For a satisfiable 2-CNF formula G, Algorithm $PAPA(G)$ runs in polynomial time and returns a satisfying assignment with probability at least 1/2.*

Proof. At first, we analyze a random walk on the line. Assume that we have a token walking on the integer line, but its position is limited to the integers 0,...,n for some integer n. At every step, the token decides uniformly at random whether to move one position smaller or greater, but if it is at position n, it moves always to $n-1$. For $0 \leq i \leq n$, let $s(i)$ denote the expected number of steps the tokens walks until it hits position 0. We have:

$$\begin{aligned} s(0) &= 0 \\ s(n) &= s(n-1) + 1 \\ s(i) &= \frac{s(i-1) + s(i+1)}{2} + 1 \text{ for } 1 < \text{i} < \text{n} \end{aligned}$$

This recursion can be solved to $s(i) = i(2n - i) \leq n^2$.

Now, let β^* be a satisfying assignment for G. We track the hamming distance of β and β^*. In every repetition, we choose a wrong clause C. So at least one of the two variables in C is assigned a wrong value by β. Since we choose a variable from C uniformly at random, we decrease the hamming distance by one with probability at least $1/2$ in every repetition. The expected number of repetitions needed to reach hamming distance 0 depends on the probability that we decrease the hamming distance in every step. Clearly, the maximum is reached when every repetition does this with probability exactly $1/2$. Hence $s(i)$ is an upper bound for the number of repetitions necessary to reach hamming distance 0 when i denotes the hamming distance of β^* and the initial β. Because $s(i)$ increases with i, $s(n_G)$ is an upper bound for the number of repetitions considering an arbitrary initial β.

By Markov's Inequality, with probability $1/2$, we need at most twice the number of expected repetitions. Thus when we repeat the process $2n^2$ times, we find β^* with probability at least $1/2$. □

2.3.2 Schöning's Algorithm

In 1999 in [24], Schöning established the beautiful randomized algorithm in Figure 2.6.

```
Algorithm SCH(k-CNF G, assignment β)
1    repeat 3n_G times {
2       if β satisfies G then break
3       Select an arbitrary clause C ∈ G that is not satisfied by β
4       Choose a variable in C uniformly at random and flip its value in β
5    }
6    return β
```

Figure 2.6: Schöning's Algorithm

Indeed, it is only a slight variation of Papadimitriou's 2-SAT Algorithm. However, to bound the running time of this algorithm, Schöning proved the following theorem, which bounds the success probability of this algorithm in terms of the hamming distance $dist(\beta, \beta^*)$ of some initial assignment β and some satisfying assignment β^*:

Theorem 2.3. *Let G be a satisfiable k-CNF formula and β^* be a satisfying assignment for G. For each initial assignment β, the probability that Algorithm $SCH(G, \beta)$ finds a satisfying assignment is at least*

$$(k-1)^{-dist(\beta,\beta^*)-o(n_G)}.$$

Proof. Here we only sketch the proof. Similar to the proof of Theorem 2.2, a random walk on the line is analyzed. This time, the token moves to the position one smaller with probability $1/k$ and to the one greater with probability $(k-1)/k$. If the token starts a position i, Schöning proved that the probability that the token hits position 0 is at least $(k-1)^{-i-o(n_G)}$.

Now, let β^* be a satisfying assignment for G. We track the hamming distance of β and β^*. In every repetition, we choose a wrong clause C. So at least one of the k variables in C is assigned a wrong value by β. Since we choose a variable from C uniformly at random, we decrease the hamming distance by one with probability at least $1/k$ in every repetition. Hence the probability to reach hamming distance 0 is at least the probability that a token starting at position $dist(\beta, \beta^*)$ hits position 0. The claim follows. □

Immediately from Theorem 2.3, we have the following corollary:

Corollary 2.4. *Let G be a satisfiable k-CNF formula and β^* be a satisfying assignment for G. Let P be a probability distribution that maps each assignment β to some probability. The probability that Algorithm $SCH(G, \beta)$, where β is an assignment selected at random according to P, finds a satisfying assignment is at least*

$$\mathbb{E}\left[(k-1)^{-dist(\beta,\beta^*)-o(n_G)}\right]$$

where the expectation is computed with respect to $P(\beta)$.

Schöning used this to show that if we draw some assignment uniformly at random and call Algorithm SCH with this assignment, we find a satisfying assignment with probability at least $(2-2/k+\epsilon)^{-n_G}$, which immediately yields the $\mathcal{O}\left((2-2/k+\epsilon)^{n_G}\right)$ expected running time bound for some arbitrary small positive ϵ.

Indeed, the approach can be transferred to more general constraint satisfaction problems (CSP). Having a CSP with n variables, where each can take at most d values and each constraint involves at most l variables, the random-walk approach needs $\mathcal{O}\left(d \cdot (1 - 1/l) + \epsilon\right)^n)$ expected running time at most, cf. [24].

2.3.3 Deterministic Local Search

When derandomizing Schöning's Algorithm, we have two major problems: how to go and where to start. Dantsin, Goerdt, Hirsch, Kannan, Kleinberg, Papadimitriou, Raghavan, and Schöning presented in [6] in 2002 a way to address both problems.

Let us first focus on the first problem. For some assignment β and some integer $r \geq 0$, define $B(\beta, r)$ to be the *hamming ball* of radius r around β, i.e. the set of all assignments β' with $dist(\beta, \beta') \leq r$.

Lemma 2.5. *Given a k-CNF formula G, an assignment β that does not satisfy G, and let $B(\beta, r)$ contain a satisfying assignment β^* of G. Let C be a clause that is not satisfied by G. Then there is a literal l in C so that $B(\beta, r-1)$ contains a satisfying assignment for $G|_{l=1}$.*

Proof. β^* must satisfy C, so for at least one of the literals l in C, $G|_{l=1}$ must be satisfiable. $G|_{l=1}$ does not care anymore for the value assigned to l by β^*, thus $\beta^* \oplus l$ is a satisfying assignment for $G|_{l=1}$. $\beta^* \oplus l$ is contained in $B(\beta, r-1)$. □

From this lemma, it is quite straightforward to devise the algorithm shown in Figure 2.7, which searches a hamming ball of radius r around some assignment β.

Algorithm *Search*(k**-CNF** G**, assignment** β**, integer** r**)**
1 **if** β satisfies G **then return** *true*
2 **if** $r = 0$ or $\perp \in G$ **then return** *false*
3 Select an arbitrary clause $C \in G$ that is not satisfied by β
4 **for each** literal l in C **do if** $Search(G|_{l=1}, \beta, r-1)$ **then return** *true*
5 **return** *false*

Figure 2.7: Search a Hamming Ball

The algorithm evaluates a search tree of maximum depth r and branching factor k. Since this can be done in $\mathcal{O}\left(poly(n) \cdot k^r\right)$, we have:

Lemma 2.6. *Given a k-CNF formula G, an assignment β of G, and an integer r. If $B(\beta, r)$ contains a satisfying assignment for G, then Algorithm $Search(G, \beta, r)$ will find it in $\mathcal{O}\left(poly(n_G) \cdot k^r\right)$ running time.*

To address the second problem, we have to cover the search space with hamming balls of size r. A (binary) *code* of length n is a set of bit strings with length n. We use such a code of length n_G to produce the requested covering. To generate an assignment from a codeword, we simply let the ith bit in the codeword decide whether 0 or 1 is assigned to the ith variable in an arbitrary but fixed ordering of $vars(G)$. In this way, a code generates a set of assignments. The distance of an assignment β to the code is just the minimum that can be achieved by checking the hamming distance of β to every assignment generated by the code. The *covering radius* of a code is defined to be the maximum of the distances of all possible assignments of G to the code. Having a code that has covering radius r, we have to call Algorithm *Search* with every assignment generated by the code and r for the search radius.

Dantsin et al proved that a code with covering radius $0 < r < n_G/2$ can be enumerated in $\mathcal{O}\left(2^{(1-h(r/n_G)+\epsilon)n_G}\right)$ where $h(p)$ is the binary entropy of p and ϵ is some arbitrary small positive real. Hence enumerating the code and calling Algorithm *Search* with each generated assignment takes time at most

$$\mathcal{O}\left(2^{(1-h(r/n_G)+\epsilon)n_G} \cdot poly(n_G) \cdot k^{rn_G}\right)$$

By choosing $r = n/(k+1)$, the running time is

$$\mathcal{O}\left(\left(2 - \frac{2}{k+1} + \epsilon\right)^{n_G}\right)$$

for some arbitrary small positive ϵ.

For 3-SAT, Dantsin et al provided a better variant of Algorithm *Search*, one that exploits some more structural properties in order to lower the branching factor so that the running time can be bounded by $\mathcal{O}\left(1.481^{n_G}\right)$. In 2004 in [3], Brueggemann and Kern were able to improve the branching technique, again lowering the branching factor. They achieved the currently best known deterministic bound of $\mathcal{O}\left(1.473^{n_G}\right)$ for 3-SAT.

2.4 Davis-Putnam and Local Search

2.4.1 Schöning's Algorithm and Reduction to 2-SAT

In 2002 in [23], Hofmeister, Schöning, Schuler, and Watanabe came up with the algorithm for 3-SAT shown in Figure 2.8.

Algorithm $HSSW$(3-**CNF** G)
1 $M :=$ maximum independent set of 3-clauses in G
2 $G' := G$
3 **for each** $C \in M$ **do**
4 In G', fix the variables in C to one of the seven satisfying assignments of C and simplify
5 **if** the 2-CNF formula G' is satisfiable **then return** *true*
6 Set up an assignment β for G in the following way: {
7 **for each** $C = (a, b, c) \in M$ **do**
8 **select** at random {
9 **w.p.** 4/21: Initialize (a, b, c) with $(0, 0, 1)$
10 **w.p.** 4/21: Initialize (a, b, c) with $(0, 1, 0)$
11 **w.p.** 4/21: Initialize (a, b, c) with $(1, 0, 0)$
12 **w.p.** 2/21: Initialize (a, b, c) with $(0, 1, 1)$
13 **w.p.** 2/21: Initialize (a, b, c) with $(1, 0, 1)$
14 **w.p.** 2/21: Initialize (a, b, c) with $(1, 1, 0)$
15 **w.p.** 3/21: Initialize (a, b, c) with $(1, 1, 1)$
16 }
17 Each variable not initialized yet is assigned 0 or 1 uniformly at random
18 }
19 **return** $SCH(G, \beta)$

Figure 2.8: Algorithm of Hofmeister, Schöning, Schuler, and Watanabe

In the first stage, the algorithm tries to reduce the given 3-CNF formula to a 2-CNF formula by fixing all variables in a maximum independent set M of 3-clauses. Since all clauses that are not contained in M share a variable with some clause in M, they lose at least one variable. Hence G' is a 2-CNF formula, so we can decide satisfiability of G' in polynomial time. The probability that G' is satisfiable when G is satisfiable is at least $(1/7)^{|M|}$, i.e. then we choose the right of the seven possible assignments for all clauses in M. Observe that this part of the algorithm already finds a satisfying assignment with high probability when $|M|$ is small.

In the second stage, an assignment β is set up by assigning variables in blocks. As noted in Corollary 2.4, the success probability of Algorithm SCH is subject to the probability space used to set up the assignment. Hofmeister et al proved that the success probability of Algorithm $SCH(G, \beta)$ is at least $(3/4)^{n_G - 3|M|} \cdot (3/7)^{|M|}$ when β is selected using the random scheme in the algorithm. In contrast to the first part, here we have a high success probability when $|M|$ is large.

Combining both bounds, we have a worst case value for $|M|$ at approximately $0.1466525 \cdot n_G$. So Algorithm $HSSW$ needs $\mathcal{O}(1.3303^{n_G})$ repetitions on the average until a satisfying assignment for a satisfiable 3-CNF formula G is found.

Hofmeister et al approached the problem by reducing a 3-CNF to a 2-CNF formula on the one hand, and on the other hand, information gathered during the reduction process is used to improve the initial assignment passed to Schöning's Algorithm. This approach has been shown to provide a series of improvements of the running time with bounds of $\mathcal{O}(1.32971^{n_G})$ ([17]), $\mathcal{O}(1.329^{n_G})$ ([2]), and finally, $\mathcal{O}(1.32793^{n_G})$ ([18]). The proof of the last bound is given in Chapter 3 in this work.

2.4.2 The Algorithm of Iwama and Tamaki

In 2004 in [8], Iwama and Tamaki proposed to combine Schöning's Algorithm with the PPSZ Algorithm, cf. Figure 2.9.

Algorithm $COMB$(**k-CNF** G,**integer** d)
1 $\beta :=$ assignment to G drawn uniformly at random
2 $\beta' := PPSZ(G, d, \beta)$
3 **if** β' satisfies G **then return** β'
4 $\beta' := SCH(G, \beta)$
5 **if** β' satisfies G **then return** β'
6 **return** *null*

Figure 2.9: Algorithm of Iwama and Tamaki

Roughly speaking, the idea behind this approach is based on the observation that the bound for the PPSZ Algorithm gets worse when the number of solutions increases, whereas the bound for Schöning's Algorithm improves in this case.

Iwama and Tamaki proved that repeating Algorithm $COMB$ until a solution for a satisfiable 3-CNF formula G is found has expected running time $\mathcal{O}(1.3238^{n_G})$ (for some large d). The result is improved in Chapter 5 to yield the currently best randomized running time bound of $\mathcal{O}(1.32216^{n_G})$ for 3-SAT.

Chapter 3

Improving Randomized Local Search by Initializing Strings of 3-Clauses

3.1 Introduction

In 1999 in [24], Schöning established a beautiful randomized algorithm, which we already discussed in Section 2.3.2. Using this algorithm, Schöning proved that a satisfying assignment for a satisfiable 3-CNF G can be found in $\mathcal{O}\left((4/3+\epsilon)^{n_G}\right)$ expected running time.

In Section 3.2, we show how to combine a randomized solver with Schöning's algorithm in a general way by exploiting information extracted during the solving process. This is based on the initial idea given in [23], which describes an $\mathcal{O}\left(1.3302^{n_G}\right)$ time randomized algorithm for 3-SAT. We already discussed their algorithm in Section 2.4.1. In Section 3.4, we apply this idea to Algorithm *Strings* and establish an $\mathcal{O}\left(1.32793^{n_G}\right)$ expected running time bound. Although it was independently invented, the approach we use in Section 3.2 is similar to the one found in [2].

Given a probability distribution P for assignments to G, recall that Corollary 2.4 states that the success probability of Algorithm SCH is at least

$$\mathbb{E}\left[2^{-dist(\beta,\beta^*)-o(n_G)}\right]$$

where the expectation is calculated with respect to $P(\beta)$.

In Section 3.2, we will see how we can achieve better bounds using an optimized probability distribution, which biases for assignments that *could* be closer to the satisfying assignment.

3.2 Combining Algorithm SCH with a Randomized Solver

A *local pattern* $\mathcal{P}$ is a tuple $(G_\mathcal{P}, n_\mathcal{P}, \mu_\mathcal{P}, P_\mathcal{P}, \lambda_\mathcal{P})$ where $G_\mathcal{P}$ is a CNF formula on $n_\mathcal{P} > 0$ variables, $0 < \mu_\mathcal{P} < 1$ and $(3/4)^{n_\mathcal{P}} < \lambda_\mathcal{P} < 1$ are arbitrary reals, and $P_\mathcal{P}$ is a probability distribution that maps each assignment for $G_\mathcal{P}$ to some probability so that

$$\lambda_\mathcal{P} \leq \mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right]$$

holds for all satisfying assignments β^* of $G_\mathcal{P}$, where the expectation is computed with respect to $P_\mathcal{P}(\beta)$. We define

$$o_\mathcal{P} = \frac{\ln \mu_\mathcal{P}}{\ln \mu_\mathcal{P} - \ln \lambda_\mathcal{P} \ + n_\mathcal{P} \ln(3/4)}$$

for a local pattern $\mathcal{P}$. How to compute such a probability distribution is a somewhat technical issue, which we will deal with in Section 3.5. Moreover, the meaning of $\mu_\mathcal{P}$ will become clear throughout the rest of this section. For now, let it be some arbitrary real in $(0, 1)$.

Let G and H be two arbitrary CNF formulas. We call G and H *isomorphic* if there exists an one-to-one mapping ϕ from $vars(G)$ to $vars(H)$ and a set of variables X so that transforming G into H can be done by renaming variables using ϕ and then flipping all signs of all variables in X. For example, we can transform $ab\overline{c}$ into fgh with $\phi = \{(a, f), (b, g), (c, h)\}$ and $X = \{h\}$. Consequently, we transform an assignment for G into the corresponding assignment β' of H: For $v \in vars(G)$, set $\beta'(\phi(v)) = \beta(v)$ if $v \notin X$ and $\beta'(\phi(v)) = 1 - \beta(v)$ if $v \in X$. Of course, there may be many isomorphisms from G to H, but let $map_{G,H}$ be a function that maps each assignment for G to an assignment for H using some arbitrary but fixed isomorphism.

Let Ψ be a non-empty, finite set of local patterns, then Ψ is called a *local scheme*. We define

$$o_\Psi = (3/4)^{\max_{\mathcal{P}\in\Psi} o_\mathcal{P}},$$

which is, as we will see later, the success probability of forthcoming Algorithm *Combine*.

Let I be a mapping which maps each local pattern $\mathcal{P} \in \Psi$ to a set of formulas $I(\mathcal{P})$, where each formula in $I(\mathcal{P})$ is isomorphic to $G_\mathcal{P}$. Furthermore, do not let each two different formulas from $\bigcup_{\mathcal{P}\in\Psi} I(\mathcal{P})$ share variables, i.e. they are mutually independent. Then I is a called an *instance* of Ψ. With n_I, we denote the total number of variables involved in the instance I. We define:

$$\mu(I) = \prod_{\mathcal{P}\in\Psi} \mu_\mathcal{P}^{|I(\mathcal{P})|} \quad \text{and}$$
$$\lambda(I) = \prod_{\mathcal{P}\in\Psi} \lambda_\mathcal{P}^{|I(\mathcal{P})|} \cdot (3/4)^{n_G - n_I}$$

We will prove the following invariant:

Lemma 3.1. *For every instance I of Ψ, it is true that*

$$\max\{\mu(I), \lambda(I)\} \geq o_\Psi^{n_G}.$$

Proof. $\max\{\lambda(I), \mu(I)\}$ depends on an actual instance I. Therefore, to obtain what claimed, we have to establish a lower bound on $\max\{\lambda, \mu\}$ which does not care for I, i.e. we have to minimize $\max\{\lambda(I), \mu(I)\}$ with respect to all possible instances I of Ψ.

For convenience, we write $i(\mathcal{P})$ to stand for $|I(\mathcal{P})|$. We take the logarithm of both $\mu(I)$ and $\lambda(I)$ to obtain

$$l_\mu = \ln \mu = \sum_{\mathcal{P}\in\Psi} (\ln \mu_\mathcal{P} \cdot i(\mathcal{P})) \quad \text{and}$$
$$l_\lambda = \ln \lambda = \sum_{\mathcal{P}\in\Psi} ((\ln \lambda_\mathcal{P} - n_\mathcal{P} \ln(3/4)) \cdot i(\mathcal{P})) + \ln(3/4) \cdot n_G.$$

Since $\lambda_\mathcal{P} > (3/4)^{n_\mathcal{P}}$, $0 < \lambda_\mathcal{P} < 1$, and $0 < \mu_\mathcal{P} < 1$ hold for all $\mathcal{P} \in \Psi$, we observe that all coefficients of $i(\mathcal{P})$ in l_μ and l_λ are negative resp. positive.

$l_\mu = l_\lambda$ can be considered as a plane equation. So, fix some arbitrary point on that plane and vary $i(\mathcal{P})$ away from the plane in some direction. Because of the signs of the coefficients in l_μ and l_λ, depending on the direction, l_μ resp. l_λ will not decrease. Thus the minimum of l_μ constrained to $l_\mu = l_\lambda$ is the global minimum of $\ln \max\{\lambda(I), \mu(I)\}$ with respect to all possible instances I. Hence we need to consider the following linear program:

$$\begin{aligned} &\text{Minimize } l_\mu \\ &\text{with respect to } i(\mathcal{P}) \geq 0 \text{ for all } \mathcal{P} \in \Psi \\ &\text{constrained to } l_\mu = l_\lambda. \end{aligned} \tag{3.1}$$

We know from the theory of linear programming (cf. [26]) that the minimum will be attained on some intersection of the border planes of the solution space. So, let $\mathcal{R} \in \Psi$ be a local pattern that maximizes $o_\mathcal{R}$, and set $i(\mathcal{P})$ to 0 for all $\mathcal{P} \in \Psi - \mathcal{R}$. Moreover, we solve constraint (3.1) and set

$$i(\mathcal{R}) = \frac{\ln(3/4)}{\ln \mu_\mathcal{R} - \ln \lambda_\mathcal{R} + n_\mathcal{R} \ln(3/4)} n_G = o_\mathcal{R} \frac{\ln(3/4) \cdot n_G}{\ln \mu_\mathcal{R}}.$$

Observe that this is a feasible basic solution to the linear program, i.e. one that satisfies all constraints. We will rewrite the objective function l_μ using the null (non-basic) variables (by replacing $i(\mathcal{R})$) and verify that all coefficients of non-basic variables are at least 0. Furthermore, we know that l_μ is a minimal solution if all non-basic variables in this rewritten form have coefficients at least 0. So, with $d_\mathcal{P}$, we denote the coefficient of $i(\mathcal{P})$ in the rewritten form, and let $c^\mu_\mathcal{P}$ and $c^\lambda_\mathcal{P}$ denote the coefficient of $i(\mathcal{P})$ in the original form of l_μ resp. l_λ. Then we obtain for all $\mathcal{P} \in \Psi - \mathcal{R}$ that

$$d_\mathcal{P} = c^\mu_\mathcal{P} - c^\mu_\mathcal{R} \frac{c^\lambda_\mathcal{P} - c^\mu_\mathcal{P}}{c^\lambda_\mathcal{R} - c^\mu_\mathcal{R}}.$$

To show that $d_\mathcal{P} \geq 0$ holds, we have to prove that

$$c^\mu_\mathcal{P} \geq c^\mu_\mathcal{R} \frac{c^\lambda_\mathcal{P} - c^\mu_\mathcal{P}}{c^\lambda_\mathcal{R} - c^\mu_\mathcal{R}}.$$

So, we insert the actual values to obtain equivalently

$$\ln \mu_\mathcal{P} \geq \ln \mu_\mathcal{R} \frac{\ln \lambda_\mathcal{P} - \ln \mu_\mathcal{P} - n_\mathcal{P} \ln(3/4)}{\ln \lambda_\mathcal{R} - \ln \mu_\mathcal{R} - n_\mathcal{R} \ln(3/4)}.$$

Observe that the numerator of the fraction is greater than 0 because of the definition of local patterns and the precondition for $\lambda_{\mathcal{P}}$. Thus we can divide by the numerator and by -1 to equivalently obtain

$$\frac{\ln \mu_{\mathcal{P}}}{\ln \mu_{\mathcal{P}} - \ln \lambda_{\mathcal{P}} + n_{\mathcal{P}} \ln(3/4)} \leq \frac{\ln \mu_{\mathcal{R}}}{\ln \mu_{\mathcal{R}} - \ln \lambda_{\mathcal{R}} + n_{\mathcal{R}} \ln(3/4)}, \text{ i.e.}$$
$$o_{\mathcal{P}} \leq o_{\mathcal{R}},$$

which is true due to the maximality of $o_{\mathcal{R}}$, so $d_{\mathcal{P}} \geq 0$ holds.

We apply the minimal solution to the equation for l_μ and obtain

$$\begin{aligned}
\ln \max\{\lambda(I), \mu(I)\} &\geq \ln \mu_{\mathcal{R}} \cdot i(\mathcal{R}) \\
&= \ln \mu_{\mathcal{R}} \cdot o_{\mathcal{R}} \frac{\ln(3/4) \cdot n_G}{\ln \mu_{\mathcal{R}}} \\
&= o_{\mathcal{R}} \cdot \ln(3/4) \cdot n_G \\
\max\{\lambda(I), \mu(I)\} &\geq (3/4)^{o_{\mathcal{R}} \cdot n_G} \\
&= o_\Psi^{n_G}.
\end{aligned}$$

This finishes the proof. □

Algorithm $IRWSolve$**(CNF** G**, local scheme** Ψ**, instance** I**)**
1 $\beta :=$ uninitialized assignment for G
2 **for each** $\mathcal{P}$ in Ψ and each $H \in I(\mathcal{P})$ **do**
3 In β, randomly assign the variables of H where each partial assignment β' of H has probability $P_{\mathcal{P}}(map_{H,G_{\mathcal{P}}}(\beta'))$
4 **for each** variable v of G that is not initialized yet **do**
5 Uniformly at random, assign 0 or 1 to v in β
6 **return** $SCH(G, \beta)$

Figure 3.1: Running Schöning's Algorithm with Better Initial Assignments

The algorithm shown in Figure 3.1 uses an instance of a local scheme to run Algorithm SCH with better initial assignments. The relevant properties are stated in the following lemma:

Lemma 3.2. *Let G be a satisfiable 3-CNF formula, Ψ a local schema, and I an instance of Ψ so that each formula in $I(\mathcal{P})$ is a sub formula of G. Then with probability at least $\lambda(I) \cdot 2^{-o(n_G)}$, Algorithm $IRWSolve(G, \Psi, I)$ returns a satisfying assignment for G.*

Proof. Fix some satisfying assignment β^*. From Corollary 2.4, we know that the success probability is at least $\mathbb{E}\left[2^{-dist(\beta,\beta^*)-o(n_G)}\right]$ computed with respect to to the assignment probability distribution that has been used to set up the assignment. Algorithm $IRWSolve$ initializes the assignment in blocks which are mutually independent from each other. Let l denote the number of independent blocks, and let V_i with $i \in [l]$ denote the set of variables that are set up in block i. Then

$$\mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right] = \prod_{i\in[l]} \mathbb{E}\left[2^{-dist(\beta_i,\beta_i^*)}\right]$$

is true where β_i and β_i^* denotes the restriction of β resp. β^* to V_i. For some block i, the expectation depends only on the assignment to the variables in V_i. Hence we only need to compute the expectation with respect to the probability space P_i that is used for the assignment to V_i, i.e.

$$\mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right] = \prod_{i\in[l]} \mathbb{E}_i\left[2^{-dist(\beta_i,\beta_i^*)}\right]$$

where $\mathbb{E}_i$ denotes the expectation with respect to a random assignment β_i to V_i having probability $P_i(\beta_i)$.

For a block i that is set up in step 5, we simply have $\mathbb{E}_i\left[2^{-dist(\beta_i,\beta_i^*)}\right] = \frac{1}{2}2^0 + \frac{1}{2}2^{-1} = 3/4$ since the one and only variable in V_i is set to 0 or 1 with probability $1/2$ each, so we may guess the right assignment with probability $1/2$.

Hence, we consider a block i in step 3, i.e. a CNF formula H with local pattern $\mathcal{P}$. P_i is just obtained by applying $map_{H,G_\mathcal{P}}$ to $P_\mathcal{P}$. So we have

$$\begin{aligned}\mathbb{E}_i\left[2^{-dist(\beta_i,\beta_i^*)}\right] &= \mathbb{E}_\mathcal{P}\left[2^{-dist(map_{G_\mathcal{P},H}(\beta'),\beta_i^*)}\right] \\ &= \mathbb{E}_\mathcal{P}\left[2^{-dist(\beta',map_{H,G_\mathcal{P}}(\beta_i^*))}\right] \\ &\geq \lambda_\mathcal{P}\end{aligned}$$

where $\mathbb{E}_\mathcal{P}$ denotes the expectation with respect to a random assignment β' of $G_\mathcal{P}$ having probability $P_\mathcal{P}(\beta')$. The first equality follows since $map_{G_\mathcal{P},H}$ does not change hamming distances. The inequality follows since $map_{H,G_\mathcal{P}}(\beta_i^*)$ is a satisfying assignment for $G_\mathcal{P}$, and thus, we can apply the definition of $\lambda_\mathcal{P}$.

The claim follows because for each $\mathcal{P}$, we have $|I(\mathcal{P})|$ times factor $\lambda_\mathcal{P}$, and finally, we have $n_G - n_I$ times factor $3/4$. □

A randomized Ψ-solver $Solver(G)$ that returns a pair (β, I) is a polynomial time algorithm that makes a number of random decisions using fixed positive probabilities to solve a 3-CNF formula and that returns an instance I of Ψ and an assignment β. This algorithm can be viewed as descending a top-down decision tree until it hits a leaf. Every leaf is labeled by an assignment β and an instance I. If β is *null*, the explored leaf does not yield a satisfying assignment for G, otherwise β is a satisfying assignment for G. However, every leaf has a certain probability of being explored, and we require that a leaf labeled with an instance I, has probability at least $\mu(I)$ of being explored. Furthermore, if G is satisfiable, then at least one possible leaf reachable by *Solver* must be labeled by a satisfying assignment for G.

```
Algorithm Combine(CNF G, local scheme Ψ, Ψ-solver Solve)
1    I := ∅
2    repeat {
3       (β, I') := Solve(G)
4       if β ≠ null then return β
5       if λ(I') > λ(I) then I := I'
6       β := IRWSolve(G, Ψ, I)
7       if β satisfies G then return β
8    }
```

Figure 3.2: Combine Ψ-Solver and Schöning's Algorithm

The algorithmic idea is that if the decision tree is not *deep*, then a satisfying leaf can be found quickly, yet if the decision tree is *deep*, then a *long* path can be found quickly which yields a good instance for Algorithm $IRWSolve$. The polynomial time algorithm shown in Figure 3.2 is a template that combines Algorithm $IRWSolve$ and a randomized Ψ-solver.

Now, we settle an important result for this algorithm:

Proposition 3.3. *Let Ψ be a local scheme and Solve a randomized Ψ-solver. Then Algorithm Combine$(G, \Psi, Solve)$ finds a satisfying assignment for a satisfiable 3-CNF formula G in expected running time at most*

$$o_{\Psi}^{-n_G} \cdot 2^{o(n_G)}.$$

Proof. Because G is satisfiable, there is at least one satisfiable leaf. Every node in the tree explored by the solver *Solve* is visited with a certain probability, which is the sum of the probabilities of the leaves contained in the subtree rooted at that node. Here, we assign to every node the probability that it is being visited by a run of Algorithm *Combine*.

At first, we consider the case that there is some leaf x that has probability less than $o_\Psi^{n_G}$. Beginning at this leaf, we climb up the tree until we hit the first node y that has probability at least $o_\Psi^{n_G}$, which must exist since the root has probability 1 of being visited. Let z be the child of y that lies on the path from y to x. Let p be the lowest probability that is used in any random decision of the algorithm, note that p is a fixed positive constant. Then z has probability less than $o_\Psi^{n_G}$, but at least $o_\Psi^{n_G} \cdot p$. Thus we expect at most $o_\Psi^{-n_G}/p$ iterations until the algorithm explores z and then descends to a leaf below z. Because z has probability less than $o_\Psi^{-n_G}$, every leaf in the subtree rooted at z has probability less than $o_\Psi^{-n_G}$. Hence every instance I' that is returned by the solver when reaching some leaf under z has $\mu(I') < o_\Psi^{n_G}$ since $\mu(I')$ is a lower bound for the probability that a leaf with instance I' is reached.

By Lemma 3.1, we know that $\lambda(I') \geq o_\Psi^{n_G}$ holds. Since the algorithm replaces I only by I' if $\lambda(I') > \lambda(I)$ is true, $\lambda(I) \geq o_\Psi^{n_G}$ holds all the time after such an instance I' was found.

As soon as $\lambda(I) \geq o_\Psi^{n_G}$ holds, we expect another at most $o_\Psi^{-n_G} \cdot 2^{o(n_G)}$ iterations until a satisfying assignment is found by Algorithm $IRWSolve$, cf. Lemma 3.2. Hence we expect at most

$$o_\Psi^{-n_G}/p + o_\Psi^{-n_G} \cdot 2^{o(n_G)} = o_\Psi^{-n_G} \cdot 2^{o(n_G)}$$

iterations until we find a satisfying assignment for G.

We still have to consider the case that there is not any leaf that has probability less than $o_\Psi^{n_G}$. In this case, every leaf has probability at least $o_\Psi^{n_G}$ of being visited, and thus also a leaf containing a satisfying assignment is expected to be found within $o_\Psi^{-n_G}$ iterations. The claim follows. □

In Section 3.4, we will devise a randomized solver called *Strings*, which is to be used with Algorithm *Combine*. But before, in Section 3.3, we have to prove some interesting properties of unit clause propagation.

3.3 Unit Clause Propagation

In this section, we focus on the elimination of unit clauses. The crucial observation is that a unit clause can only be satisfied by assigning 1 to its lonely literal. Despite that this idea is quite simple, the elimination of unit clauses is quite powerful. We already discussed two algorithms, Algorithm PPZ and $PPSZ$, which use elimination of unit clauses as lonely rule to infer partial assignments.

Let a CNF formula that does not contain a unit clause be called *unit-free*. For two CNF formulas G and H, we define $\chi(G, H)$ to be the set of pairs $(C, D) \in G \times H$ with $D \subset C$ and $|D| = 2$, i.e. a clause C which is a 3-clause in G and that has been reduced to a 2-clause D in H. The following proposition shapes the kernel of the algorithms in this chapter:

Proposition 3.4. *Let G be a 3-CNF formula and L a set of literals so that $G|_L$ is unit-free. Then at least one of the following holds:*

(1) If G is satisfiable, then $G|_L$ is satisfiable.

(2) $G|_L$ contains $\perp$ and thus is not satisfiable.

(3) $\chi(G, G|_L)$ is not empty.

Proof. If G is not satisfiable, then obviously (1) holds. Therefore, let G be satisfiable. Assume that neither (2) nor (3) hold. We will show that (1) holds in this case.

Let β^* be a satisfying assignment for G. Then there must be a clause $D \in G|_L$ that is not satisfied by β^* since $G|_L$ is not satisfiable by our assumption. But, observe that if $D \in G$ held D would be satisfied by β^*, thus D cannot be in G. However, $D \in G|_L$ holds, and thus there is a clause $C \in G$ with $D \subset C$, i.e. D is obtained by removing at least one literal $\bar{l}$ with $l \in L$ from some clause $C \in G$.

Firstly, D cannot be $\perp$ since (2) is assumed to be wrong, but secondly, D cannot be a unit clause since $G|_L$ is unit-free, and finally, D cannot be a 3-clause since then C would have to be a 4-clause, but G is a 3-CNF formula. We conclude that D has to be a 2-clause, yet this violates our assumption that (3) is wrong since (C, D) would be in $\chi(G, G|_L)$.

Hence such a clause $D \in G|_L$ cannot exist, and finally, we conclude that $G|_L$ is also satisfied by β^*, which completes the proof. □

Algorithm $Simplify$**(CNF** G**)**
1 **while** there exists a unit clause $l \in G$ **do**
2 $G := G|_l$
3 **return** G

Figure 3.3: Unit-Clause Propagation

We can apply this proposition to any case where a unit-free CNF formula G is obtained from a 3-CNF formula G by fixing arbitrary literals in any order. Proposition 3.4 requires a unit-free CNF formula $G|_L$, and now, we discuss how to handle unit clauses. Since a unit clause forces its literal to be true, it is quite easy to remove all unit clauses from some CNF formula G. This can be done using the trivial polynomial time algorithm shown in Figure 3.3.

As stated before, a unit clause allows only one value for its literal, so we see that the result of $Simplify(G)$ is satisfiable if and only if G is satisfiable. Since the formula returned as result of this algorithm is unit-free, we can conclude:

Lemma 3.5. *Let G be a unit-free CNF formula, L a set of literals, and $H = Simplify(G|_L)$. Let L' denote the set of all literals fixed in the loop in Algorithm $Simplify$. Then the 3-CNF formula G and the set of literals $L \cup L'$ satisfy the precondition of Proposition 3.4.*

In Section 3.4, we will use the preceding lemma in combination with Proposition 3.4 in order to optimize the reduction process.

Let us deal with the following simple yet powerful lemma:

Lemma 3.6. *Let G be a 3-CNF formula and ab a 2-clause in G. Set H to $Simplify(G|_a)$. Let (C, D) be an arbitrary pair in $\chi(G, H)$. If $b \in D$ holds, then G is equivalent to $G' = G - C + D$.*

Proof. Assuming that b is contained in D, let d be the literal of C that is missing in D. Observe that d can even be $\overline{a}$. We show that any assignment for G satisfies G' and vice versa.

So, let β^* be a satisfying assignment for G. We need to show that β^* satisfies D. Observe that assigning 1 to a in G results in fixing d to 0 by Algorithm $Simplify$. Hence if β^* assigns 1 to a, it must also assign 0 to d. Because $d \in C$, some literal $l \in C$ with $l \neq d$ must be true under β^*. Moreover, l must be contained in D since d is the literal removed from C to obtain D. Thus D is true under β^*. On the other hand, if β^* assigns 0 to a, then β^* must assign 1 to b in order to satisfy ab. Then D is satisfied by β^* since $b \in D$ holds. We conclude that β^* satisfies G'.

Now, let β^* be a satisfying assignment for G'. Then β^* satisfies D and thus also satisfies C since $D \subset C$. Hence G is also satisfied by β^*. □

A 3-CNF formula G is *clean* if Lemma 3.6 is not applicable to any 2-clause in G. To *clean* a 3-CNF formula means to apply Lemma 3.6 as long as possible. For example, $\{ab, \overline{a}bc\}$ or $\{ab, bc, a\overline{c}d\}$ cannot occur in a clean 3-CNF formula.

The following lemma shows that cleaning preserves the properties of Proposition 3.4:

Lemma 3.7. *Let G be a clean 3-CNF formula and L a set of literals. Set $H = Simplify(G|_L)$, and let H' be the cleaned version of H. Then the following holds:*

(1) H contains $\bot$ if and only if H' contains $\bot$.

(2) If $\chi(G, H)$ is empty, then $\chi(G, H')$ is empty.

(3) If $\chi(G, H)$ is not empty, then $\chi(G, H')$ is not empty.

Proof. Case (1) is obvious since cleaning does neither add nor remove $\bot$.

Consider case (2). Assuming that $\chi(G, H)$ is empty, there is not any 2-clause in H that has not been already a 2-clause in G. Thus, Lemma 3.6 is not applicable to H because G is already clean and new 3-clauses cannot emerge, showing $H = H'$.

Finally, we prove case (3). Cleaning does not decrease the number of 2-clauses in a 3-CNF formula. So all 2-clauses in H are also 2-clauses in H'. That means $\chi(G, H) \subseteq \chi(G, H')$. □

We will exploit these facts in our randomized solver presented in Section 3.4. Figure 3.4 shows an extension of $Simplify$ that also cleans the formula.

Algorithm $CleanSimplifyClean$(**CNF** G)
1 Clean G
2 $G := Simplify(G)$
3 Clean G
4 **return** G

Figure 3.4: Clean, Simplify, and Clean

3.4 Randomized Solver Using Strings

Let $(C_1, .., C_l)$ be a sequence of 3-clauses so that successive clauses share no more than two, but at least one variable, and that non-successive clauses are independent. Then we call $(C_1, .., C_l)$ a *string* of length l. For a string S and a clause C, let $S \odot C$ denote the string obtained by appending C to S.

The *type of a string* S, denoted with $type(S)$, is built as follows. $type(S)$ is a sequence having one item less than S. Each item in $type(S)$ describes how the corresponding succeeding clauses in S are 'connected'. p means both clauses share exactly one variable and that with the same sign, whereas n means both clauses share exactly one variable and that with different signs. Finally, nn tells that both clauses share exactly two variables and both with different signs. These types are sufficient for our intentions. For example, the string $(abc, \overline{b}\overline{c}d, \overline{d}ef, fgh)$ has type (nn, n, p). The types of strings are interesting because all strings of the same type are isomorphic to each other. Hence one representative string of some type can be used as $G_{\mathcal{P}}$ in a local pattern $\mathcal{P}$.

We will establish a randomized solver which outputs a couple of strings as byproduct. These will serve as an instance passed to Algorithm $IRWSolve$. The solver extracts strings in the formula one-by-one by 'growing' them. Growing means that a string is started with a single clause and extended step-by-step with appropriate clauses.

The randomized solver is split into a set of algorithms, which we will discuss in this section. Despite the fact that the algorithms call each other, they do not branch, i.e. they will finish in polynomial time. However, the algorithms make a lot of random decisions, where some may fail to produce a satisfying assignment. But, the algorithms guarantee completeness, i.e. if the formula is satisfiable before some decision is done, there is at least one choice at that step that preserves

satisfiability. Thus if the input formula is satisfiable, there is a positive probability to find a satisfying assignment, i.e. there exists a satisfying leaf.

Since a string may become very long, the algorithms request a set of string types $\mathcal{T}$ which has the following meaning. $\mathcal{T}$ is a set of forced stop types, i.e. if a string has type in $\mathcal{T}$, it will not be extended any longer. Beside these, there are cases where the algorithms automatically decide to stop the string, these are called the automatic stop types. At the end, we will see that we only need to consider strings of length at most five, i.e. that looking for longer strings does not improve the running time.

All algorithms make us of some global variables: G is the current formula, I contains the current instance of strings, S is the string currently being grown, and $\mathcal{T}$ contains a set of stop string types.

```
Algorithm Finish()
1    I(type(S)) := I(type(S)) + S
2    S := ()
```

Figure 3.5: Finish the Current String

The algorithm shown in Figure 3.5 finishes the current string, i.e. it adds string S to the corresponding set of the instance I and resets S.

```
Algorithm Extend(3-clause C, set of clauses D)
1    if S ≠ () and C is independent to S then Finish
2    S := S ⊙ C
3    if type(S) ∈ T then {
4        Finish
5        Uniformly at random, select β from the set of satisfying assignments for
             D
6        In D, toggle the sign of all literals which are wrong under β, i.e. build a
             set of literals reflecting β
7        G := G|_D
8        return false
9    }
10   return true
```

Figure 3.6: Extend the Current String

The algorithm shown in Figure 3.6 tries to extend the current string S by clause C. If C and S are independent, S is finished, i.e. $type(S)$ is an automatic stop type. However, C is added to S, and the algorithm checks if the type of the result string is in $\mathcal{T}$, i.e. $type(S)$ is a forced stop type. In this case, S is finished and a random assignment β for D is chosen uniformly from the set of assignments satisfying D, i.e. each satisfying assignment for D has probability $1/|sat(D)|$. In G, the variables of D are fixed according to β. Finally, Algorithm *Extend* returns whether the type S was not finished because its type is not in $\mathcal{T}$, i.e. if *true* is returned, the string is extensible.

Algorithm *Strings*(**CNF** G)
1 For all $\mathcal{P} \in \Psi$, set $I(\mathcal{P}) := \emptyset$
2 $S := ()$
3 **while** exists literal c in G **do**
4 $Choose2(c, \overline{c})$
5 **if** $\perp \in G$ **then** $\beta := null$
6 **else** $\beta :=$ final assignment (how all the literals were fixed)
7 **return** (β, I)

Figure 3.7: Randomized Solver Using Strings

The algorithm shown in Figure 3.7 is the algorithm we are going to use in Algorithm *Combine*. It does some initialization and essentially loops as long as it finds a literal that is not fixed and calls Algorithm *Choose*2 to deal with the literal. Finally, it returns an assignment and an instance containing all the strings found.

The algorithm shown in Figure 3.8 gets two sets of literals C_1 and C_2 and decides whether it should use $G|_{C_1}$ or $G|_{C_2}$ to continue with. Note that the choice of C_1 and C_2 must ensure that at least one of $G|_{C_1}$ or $G|_{C_2}$ is satisfiable if G is satisfiable.

At first, the algorithm checks whether one of the two sets can be excluded using Lemma 3.7 and Lemma 3.5. This is done by checking for $\perp$ and empty χ, which would reveal that one of the two choices preserves satisfiability. If a check is true, then the string is finished and the function returns.

If all checks fail, then we choose whether to take $G|_{C_1}$ or $G|_{C_2}$ at random and get a reduced 3-clause to extend the current string. This way, the new clause

```
Algorithm Choose2(clause C_1, C_2 )
1    G_1 := CleanSimplifyClean(G|_{C_1})
2    G_2 := CleanSimplifyClean(G|_{C_2})
3    if ⊥ ∈ G_1 then G := G_2, Finish, return
4    if ⊥ ∈ G_2 then G := G_1, Finish, return
5    if χ(G, G_1) = ∅ then G := G_1, Finish, return
6    if χ(G, G_2) = ∅ then G := G_2, Finish, return
7    select at random {
8       w.p. 1/2: {
9          (C, D) := arbitrary pair in χ(G, G_1)
10         G := G_1
11         if Extend(C, {D}) then Choose3(C, D)
12      }
13      w.p. 1/2: same as previous case, but use G_2 instead of G_1
14   }
```

Figure 3.8: Choose from Two Sets of Assignments

will extend S using an n- or p-connection or starts a new string. If the string is extensible, control is passed to Algorithm *Choose*3 in order to look for more extensions.

The algorithm shown in Figure 3.9 takes a 2-clause ab that has to be in G and decides whether to continue with $G|_{ab}$, $G|_{a\bar{b}}$, or $G|_{\bar{a}b}$.

At first, we check whether we can derive a contradiction for $a = 1$ or $b = 1$. If a or b can be deduced to be 0, we can safely fix the other variable to 1 since ab must be satisfied. In this case, we finish the string and return.

Secondly, we look for a contradiction in every of the three possible assignments that satisfy ab. If we find some, we can exclude the contradictious assignment and pass control to Algorithm *Choose*2 to select from the two remaining choices.

Thirdly, we check for empty χ since in absence of a contradiction, an empty χ tells us that satisfiability of G implies that the respective new formula is satisfiable too, cf. Lemma 3.7 and Proposition 3.4. If we can eliminate both variables, we finish the string and return. If we can deduce only the value of one variable, we pass control to Algorithm *Choose*2 to select from the remaining two choices.

Algorithm $Choose3$**(3-clause** C**, 2-clause** ab**)**
1 $G_a := CleanSimplifyClean(G|_a)$
2 $G_b := CleanSimplifyClean(G|_b)$
3 $G_{ab} := CleanSimplifyClean(G|_{ab})$
4 $G_{a\bar{b}} := CleanSimplifyClean(G|_{a\bar{b}})$
5 $G_{\bar{a}b} := CleanSimplifyClean(G|_{\bar{a}b})$
6 **if** $\perp \in G_a$ **then** $G := G_{\bar{a}b}$, $Finish$, **return**
7 **if** $\perp \in G_b$ **then** $G := G_{\bar{a}b}$, $Finish$, **return**
8 **if** $\perp \in G_{a\bar{b}}$ **then** $Choose2(ab, \bar{a}b)$, **return**
9 **if** $\perp \in G_{b\bar{a}}$ **then** $Choose2(ab, a\bar{b})$, **return**
10 **if** $\perp \in G_{ab}$ **then** $Choose2(a\bar{b}, \bar{a}b)$, **return**
11 **if** $\chi(G, G_{ab}) = \emptyset$ **then** $G := G_{ab}$, $Finish$, **return**
12 **if** $\chi(G, G_{a\bar{b}}) = \emptyset$ **then** $G := G_{a\bar{b}}$, $Finish$, **return**
13 **if** $\chi(G, G_{\bar{a}b}) = \emptyset$ **then** $G := G_{\bar{a}b}$, $Finish$, **return**
14 **if** $\chi(G, G_a) = \emptyset$ **then** $Choose2(a\bar{b}, ab)$, **return**
15 **if** $\chi(G, G_b) = \emptyset$ **then** $Choose2(ab, \bar{a}b)$, **return**
16 **if** $\exists c : \bar{a}\bar{b}c \in G$ **then** {
17 **if** not $Extend(\bar{a}\bar{b}c, \{\bar{a}\bar{b}c, ab\})$ **then return**
18 $Choose5(\bar{a}\bar{b}c, ab)$
19 **return**
20 }
21 **select** at random {
22 **w.p.** 1/3: {
23 $(C, D) :=$ arbitrary pair in $\chi(G, G_a)$
24 $G := G_{a\bar{b}}$
25 **if** $Extend(C, \{D\})$ **then** $Choose3(C, D)$
26 }
27 **w.p.** 1/3: same as previous cause, but swap a and b
28 **w.p.** 1/3: {
29 $(C, D) :=$ arbitrary pair in $\chi(G, G_{ab})$
30 $G := G_{ab}$
31 **if** $Extend(C, \{D\})$ **then** $Choose3(C, D)$
32 }
33 }

Figure 3.9: Choose from Three Sets of Assignments

Fourthly, we check if there is some clause $\overline{a}\overline{b}c$ for some c in G. Then we extend the string by $\overline{a}\overline{b}c$, and if still extensible, we pass control to Algorithm $Choose5$ to solve this special case. This way, an nn-connection is generated.

Hence, we can assume that there is not such a clause. However, because of the checks, we know that we can find reduced clauses for all three possible assignments to ab and thus extend the string by C or start a new string for some pair (C, D) in χ. If the string is still extensible, we pass control to Algorithm $Choose3$ for further extensions. We show that C neither contains a, b, nor $\overline{a}\overline{b}$.

If $\overline{a}\overline{b}$ was in C then we would have passed control to Algorithm $Choose5$. Assume that b is in C. We do not need to consider $G|_b$ or $G|_{ab}$ since then b would be fixed to 1 and C would be removed, but not shortened to D. Hence we have to deal only with $G|_a$.

At first, assume that $b \in D$. Then Lemma 3.6 would be applicable, meaning that G_a would not be clean. However, this cannot be true since G_a was cleaned by Algorithm $CleanSimplifyClean$. So b must be the literal that was removed from C to obtain D. This means that setting a to 1 induces b to be 0, yielding $\perp$ in $G|_{ab}$. Since the algorithm passed that check, we know that this cannot happen, thus b cannot be the removed literal. Hence b cannot be in C.

The proof for $a \notin C$ follows by swapping a and b. So, we proved that C cannot share a variable with the last clause in the string with same sign (p-connection), and moreover, C cannot share two variables with different signs (nn-connection). Hence clause C will extend S using an n-connection or starts a new string.

The algorithm shown in Figure 3.10 needs a 2-clause $ab \in G$ and a 3-clause $\overline{a}\overline{b}c \in G$. We have five possible assignments to that clause pair and the algorithm decides which assignment to use. It uses some probability constants p_0, p_1, q_0, and q_1 with $2p_0 + p_1 = 1$ and $q_0 + q_1 = 1$, which are subject to optimization and given in Section 3.5.

In the outer **select** statement, we select one of the three satisfying assignments for ab, so this decision propagates satisfiability in at least one of the choices.

The first case corresponds to settings $a = 1$ and $b = 1$, which imply $c = 1$. We finish the string and return, i.e. $type(S)$ is an automatic stop type. This choice has probability p_1.

```
Algorithm Choose5(3-clause ā b̄ c, 2-clause ab)
1    select at random {
2       w.p. p_1 : G := G|_abc, Finish
3       w.p. p_0 : {
4          G := G|_ab̄
5          G_c := CleanSimplifyClean(G|_c)
6          G_c̄ := CleanSimplifyClean(G|_c̄)
7          if ⊥ ∈ G_c then G := G_c̄, Finish, return
8          if χ(G, G_c) = ∅ then G := G_c, Finish, return
9          select at random {
10            w.p. q_0 : G := G_c̄, Finish, return
11            w.p. q_1 : {
12               (C, D) = arbitrary pair in χ(G, G_c)
13               G := G_c
14               if Extend(C, {D}) then Choose3(C, D)
15            }
16         }
17      }
18      w.p. p_0 : same as previous case, but swap a and b
19   }
```

Figure 3.10: Choose from Five Sets of Assignments

The second (and analogously third) case means setting $a = 1$ and $b = 0$. We still have to consider the value for c. At first, we check if we could apply Proposition 3.4 for $c = 1$. If we find $\perp$ in G_c, we are safe to go with $c = 0$, whereas an empty $\chi(G, G_c)$ would suggest $c = 1$. In both case, we finish the string and return. Here, we have probability p_0.

Otherwise we choose whether to continue with $c = 0$ or $c = 1$. On the one hand, if we decide to follow $c = 0$, we finish the string and return. On the other hand, we select a reduced clause pair (C, D) from $\chi(G, G_c)$, extend the string by C or start a new string, and we go on with Algorithm $Choose3$ if the string is extensible. The first choice has total probability p_0q_0, while the second has p_0q_1.

To obtain our final algorithm, we use Algorithm *Strings* in the call to Algorithm $Combine(G, \Psi, Strings)$. In Section 3.5, we explain how Ψ and $\mathcal{T}$ are set up. These are rather technical issues and omitted here. Yet, they show that the worst case is determined by the strings of type () (i.e. single clauses), and in that

case, $\lambda_{()} = 3/7$ and $\mu_{()} = 1/3$ hold. By inserting these values in the equation in Proposition 3.3, we obtain the following proposition, which is the main result of this chapter:

Proposition 3.8. *Let G be a satisfiable 3-CNF formula. Then Algorithm $Combine(G, \Psi, Strings)$ finds a satisfying assignment for G in expected running time at most $\mathcal{O}(1.32793^{n_G})$.*

Interestingly, the bound does not decrease if we try to seek for longer strings because the case () may always arise and thus will always be the limit, at least using our approach.

3.5 Local Scheme for Algorithm *Strings*

In this section, we deal with the local schema Ψ that has to be used in our $\mathcal{O}(1.32793^{n_G})$ algorithm presented in the preceding section. For each string type T, we form a local pattern $\mathcal{P}_T$ to be included in Ψ. For a string type T, we set $G_{\mathcal{P}_T}$ to $\{C_1, ..., C_l\}$ where the string $S_{\mathcal{P}_T} = (C_1, ..., C_l)$ is some arbitrary string that has type T. Note that we can use any such string $(C_1, ..., C_l)$ because all formulas $\{C_1, ..., C_l\}$ where $(C_1, ..., C_l)$ has type T are isomorphic to each other.

We split the types into two cases: the forced stop types in $\mathcal{T}$ and the automatic stop types not in $\mathcal{T}$. The types in $\mathcal{T}$ bound the strings that can be found by the algorithms in Section 3.4. The forced stop types occur if a string is finished in Algorithm *Extend* because its type is found in $\mathcal{T}$. The automatic stop types may occur if a string is finished because Algorithm *Extend* recognizes an independent clause in S starting a new string or because it was stopped by calling Algorithm *Finish* in Algorithms *Choose*2, *Choose*3, or *Choose*5.

Since we are interested in a lower bound of the probability that a certain leaf containing an instance I is reached, we need to multiply all the probabilities of the random choices the algorithm has taken to reach the leaf. Thus we disassemble $\mu(I)$ into a product of factors $\mu_{\mathcal{P}}$ for an instance I:

$$\mu(I) := \prod_{\mathcal{P} \in \Psi} \mu_{\mathcal{P}}^{|I(\mathcal{P})|}$$

$\mu_{\mathcal{P}}$ can be calculated as follows. Assume that we have a string S of type $\mathcal{P}$ that is being flushed in Algorithm *Finish*. Then S has been found by making some choices in the algorithms. The probability of each taken choice is a weight to be included in $\mu_{\mathcal{P}}$.

The first clause of the first string is always generated with weight 1/2 in Algorithm *Choose*2 called by Algorithm *Strings*. We postpone this weight to the last string of all strings found during one loop in Algorithm *Strings*. So, at first, we analyze all but the last string found in one loop.

Let S be a non-last string. Consider an n-connection. If it is preceded by an nn-connection, both together have weight p_0q_1, else it has weight 1/3 or 1/2. In contrast, a p-connection always has weight 1/2. When Algorithm *Extend* fails to extend the string because of independence, the weight to choose the assignment for the last clause is always at least 1/3. Let r denote the number of nn, n-connections, s the number of n-connections not preceded by an nn-connection, and t the number of p-connections. Then the weight of a non-last string S is at least

$$(p_0q_1)^r \cdot (1/3)^s \cdot (1/2)^t \cdot 1/3$$

where the last factor 1/3 is for finishing the last clause.

Now, we consider the last string S. At first, assume that the last string is also an automatic stop type. If it ends with an nn-connection, this has weight $\min\{p_1, p_0q_0\}$. In other cases, the last clause was fixed because the right assignment could be derived from the formula. All other connections are similar to the previous non-last string case. But, we have to include the initial 1/2. We conclude that if the string ends with an nn-connection, the weight of S is at least

$$1/2 \cdot (p_0q_1)^r \cdot (1/3)^s \cdot (1/2)^t \min\{p_1, p_0q_0\}$$

where the first factor 1/2 is the weight postponed from the first string in the loop. Similarly, if S does not end with an nn-connection, the weight of S is at least

$$1/2 \cdot (p_0q_1)^r \cdot (1/3)^s \cdot (1/2)^t.$$

Finally, assume that the last string S is a forced stop type. The analysis is similar to the previous case. On the on hand, if the last connection is an nn-connection, we have five assignments to choose from when Algorithm *Choose*3

calls Algorithm $Extend(\overline{a}\overline{b}c, \{\overline{a}\overline{b}c, ab\})$. Hence we have that the weight of S is at least

$$1/2 \cdot (p_0 q_1)^r \cdot (1/3)^s \cdot (1/2)^t \cdot 1/5$$

where the first factor $1/2$ is the weight postponed from the first string in the loop. On the other hand, if the last connection is not an nn-connection, Algorithm $Extend$ has only to choose from 3 assignments. Then the weight of S is at lest

$$1/2 \cdot (p_0 q_1)^r \cdot (1/3)^s \cdot (1/2)^t \cdot 1/3.$$

Having calculated the $\mu_{\mathcal{P}}$ values, we can now focus on the $\lambda_{\mathcal{P}}$ values. We need to determine a probability distribution $P_{\mathcal{P}}$ over the assignments of $G_{\mathcal{P}}$. From the definition of $\lambda_{\mathcal{P}}$, we have that $\lambda_{\mathcal{P}}$ must be a lower bound for $\mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right]$ for every satisfying assignment β^* of $G_{\mathcal{P}}$ where the expectation is calculated with respect to $P_{\mathcal{P}}(\beta)$. Because we would like $\lambda_{\mathcal{P}}$ to be as large as possible, we need to find the distribution $P_{\mathcal{P}}$ that maximizes

$$\min\left\{\mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right] | \beta^* \text{ satisfies } G_{\mathcal{P}}\right\} \tag{3.2}$$

where the expectation is computed with respect to $P_{\mathcal{P}}(\beta)$. Using the maximizing $P_{\mathcal{P}}$, we assign to $\lambda_{\mathcal{P}}$ the minimum of $\mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right]$ with respect to all possible satisfying assignments for $G_{\mathcal{P}}$. Then $\lambda_{\mathcal{P}}$ is a lower bound for the expectation for all $\beta^* \in sat(G_{\mathcal{P}})$.

Thus we have a classical max-min optimization problem and could form a linear program and solve it using the Simplex-Method constraining $P_{\mathcal{P}}$ to be a probability distribution, i.e. summing to one and being at least 0 for all assignments. However, we use a different approach (cf. [17]): Instead of computing max-min, we compute max-equal, i.e. find a probability distribution so that $\mathbb{E}\left[2^{-dist(\beta,\beta^*)}\right]$ is equal (and thus maximal) for all β^* satisfying $G_{\mathcal{P}}$. This can be done using any linear equation solver. But, beware that this may yield an invalid $P_{\mathcal{P}}$, i.e. with some negative values, so we have to check that the computed $P_{\mathcal{P}}$ is a valid probability space. Fortunately, this is true for our string types.

We have built the stop types in a way that a string is stopped as soon as it is yielding a bound that is below the worst case bound, as mentioned, already

determined by type (). The probability constants p_0, p_1, q_0, and q_1, which affect the weights of strings involving nn-connections, are set to the following values:

$$p_1 = 61083/250000$$
$$p_0 = 188917/500000$$
$$q_1 = 44167/125000$$
$$q_0 = 80833/125000$$

We set them this way to get those strings below the worst case bound.

For each string, we can compute $c_{\mathcal{P}} := (4/3)^{o_{\mathcal{P}}}$. Due to Proposition 3.3, the largest of these determines the constant c for the expected running time $c^{n_G} \cdot 2^{\mathcal{O}(n_G)}$ of our algorithm. Instead of writing formulas for $G_{\mathcal{P}}$, we write only the types, but each string of such a type can stand for $G_{\mathcal{P}}$.

The string types are separated into two table. Table 3.1 and Table 3.2 contain the forced stop types, which will be in $\mathcal{T}$. Table 3.3 shows the automatic stop types. Both tables together can be used to form a local scheme Ψ together with Algorithm *Strings* to establish the bound.

$type(S_{\mathcal{P}})$	$n_{\mathcal{P}}$	$\mu_{\mathcal{P}}$	$\lambda_{\mathcal{P}}$	$c_{\mathcal{P}} \leq$
(p,p)	7	1/24	243/1739	1.32790
(p,n,p)	9	1/72	2187/27334	1.32773
(p,n,n,p)	11	1/216	729/15904	1.32760
(p,n,n,n)	11	1/324	729/15848	1.32777
(p,n,n,nn)	10	1/180	1215/19894	1.32745
(p,n,nn)	8	1/60	405/3799	1.32755
(p,nn)	6	1/20	27/145	1.32765
(n,p,p)	9	1/72	729/9110	1.32772
(n,p,n,p)	11	1/216	2187/47732	1.32763
(n,p,n,n)	11	1/324	729/15856	1.32780
(n,p,n,nn)	10	1/180	405/6634	1.32748
(n,p,nn)	8	1/60	45/422	1.32753
(n,n,p,p)	11	1/216	2187/47704	1.32759
(n,n,p,n)	11	1/324	729/15856	1.32780
(n,n,p,nn)	10	1/180	1215/19888	1.32743
(n,n,n,p)	11	1/324	729/15848	1.32777
(n,n,n,n)	11	1/486	243/5264	1.32791
(n,n,n,nn)	10	1/270	405/6608	1.32764
(n,n,nn)	8	1/90	135/1262	1.32778
(n,nn,n,p)	10	$\frac{8343897139}{2250000000000}$	1215/19904	1.32790
(n,nn,n,n,p)	12	$\frac{8343897139}{6750000000000}$	10935/312692	1.32776
(n,nn,n,n,n)	12	$\frac{8343897139}{10125000000000}$	3645/103864	1.32789
(n,nn,n,n,nn)	11	$\frac{8343897139}{5625000000000}$	405/8692	1.32765
(n,nn,n,nn)	9	$\frac{8343897139}{1875000000000}$	675/8299	1.32777
(nn,n,p,p)	10	$\frac{8343897139}{1500000000000}$	405/6653	1.32768
(nn,n,p,n)	10	$\frac{8343897139}{2250000000000}$	405/6634	1.32789
(nn,n,p,nn)	9	$\frac{8343897139}{1250000000000}$	675/8321	1.32752
(nn,n,n,p)	10	$\frac{8343897139}{2250000000000}$	1215/19894	1.32787
(nn,n,n,n,p)	12	$\frac{8343897139}{6750000000000}$	3645/104168	1.32773
(nn,n,n,n,n)	12	$\frac{8343897139}{10125000000000}$	243/6920	1.32786
(n,nn,n,n,nn)	11	$\frac{8343897139}{5625000000000}$	405/8692	1.32765
(n,nn,n,nn)	9	$\frac{8343897139}{1875000000000}$	675/8299	1.32777
(nn,n,p,p)	10	$\frac{8343897139}{1500000000000}$	405/6653	1.32768
(nn,n,p,n)	10	$\frac{8343897139}{2250000000000}$	405/6634	1.32789

Table 3.1: Forced Stop Types

$type(S_{\mathcal{P}})$	$n_{\mathcal{P}}$	$\mu_{\mathcal{P}}$	$\lambda_{\mathcal{P}}$	$c_{\mathcal{P}} \leq$
(nn, n, p, nn)	9	$\frac{8343897139}{1250000000000}$	675/8321	1.32752
(nn, n, n, p)	10	$\frac{8343897139}{2250000000000}$	1215/19894	1.32787
(nn, n, n, n, p)	12	$\frac{8343897139}{6750000000000}$	3645/104168	1.32773
(nn, n, n, n, n)	12	$\frac{8343897139}{10125000000000}$	243/6920	1.32786
(nn, n, n, n, nn)	11	$\frac{8343897139}{5625000000000}$	225/4826	1.32762
(nn, n, n, nn)	9	$\frac{8343897139}{1875000000000}$	45/553	1.32774
(nn, n, nn)	7	$\frac{8343897139}{625000000000}$	25/176	1.32790

Table 3.2: Forced Stop Types, continued

$type(S_{\mathcal{P}})$	$n_{\mathcal{P}}$	$\mu_{\mathcal{P}}$	$\lambda_{\mathcal{P}}$	$c \leq$
$()$	3	1/3	3/7	1.32793
(p)	5	1/6	81/331	1.32688
(p, n)	7	1/18	81/578	1.32700
(p, n, n)	9	1/54	729/9080	1.32702
(n)	5	1/9	27/110	1.32755
(n, p)	7	1/18	81/578	1.32700
(n, p, n)	9	1/54	243/3028	1.32706
(n, n)	7	1/27	9/64	1.32738
(n, n, p)	9	1/54	729/9080	1.32702
(n, n, n)	9	1/81	243/3016	1.32729
(n, nn, n, n)	10	$\frac{8343897139}{1687500000000}$	135/2204	1.32737
(n, nn, n)	8	$\frac{8343897139}{562500000000}$	405/3788	1.32745
(n, nn)	6	$\frac{15270727861}{375000000000}$	45/241	1.32769
(nn, n, p)	8	$\frac{8343897139}{375000000000}$	405/3799	1.32712
(nn, n, n, n)	10	$\frac{8343897139}{1687500000000}$	405/6608	1.32733
(nn, n, n)	8	$\frac{8343897139}{562500000000}$	135/1262	1.32741
(nn, n)	6	$\frac{8343897139}{187500000000}$	45/241	1.32753
(nn)	4	$\frac{15270727861}{125000000000}$	15/46	1.32793

Table 3.3: Automatic Stop Types

Chapter 4

Derandomization of PPSZ for Unique-k-SAT

4.1 Introduction

In [13], Paturi, Pudlak, Saks, and Zane proved that for a uniquely satisfiable 3-CNF formula G, the solution can be found in $\mathcal{O}(1.3071^{n_G})$ expected running time at most, cf. Section 2.2.3. This is the best randomized bound known for Unique-3-SAT. We refer to their algorithm as the PPSZ algorithm. But paradoxically, the bound gets worse when the number of solutions increases. Alas, for the general 3-SAT and 4-SAT case, this algorithm achieves expected running time bounds of $\mathcal{O}(1.362^{n_G})$ resp. $\mathcal{O}(1.476^{n_G})$ only, which is worse than the best known randomized bounds of $\mathcal{O}(1.3238^{n_G})$ (cf. [20] or Chapter 5) resp. $\mathcal{O}(1.474^{n_G})$ (cf. [8] or Section 2.4.2).

The best bounds for k-SAT on n variables make excessive usage of random bits so that enumerating the entire probability space would yield useless bounds, i.e. much more than $\mathcal{O}(2^n)$. But, do random bounds really compete with deterministic bounds when the existence of true randomness is not provable? At least, randomized algorithms often supply a good starting point to develop fast deterministic algorithms. For example, the algorithm of Schöning in [24] (cf. Section 2.3.2), based on randomized local search and restart, yields a bound of $\mathcal{O}((2-2/k+\epsilon)^n)$ expected running time at most, which has been derandomized in [6] and improved for $k=3$ in [3] to the best known deterministic bounds of $\mathcal{O}((2-2/(k+1)+\epsilon)^n)$ for $k>3$ resp. $\mathcal{O}(1.473^n)$ for $k=3$, based on limited local search and cover-

ing codes (cf. Section 2.3.3). Alas, like so often, the deterministic bound is much worse than the original randomized one. However, in this chapter, we derandomize the PPSZ algorithm for the uniquely satisfiable case yielding (almost) the same bound as the randomized version making it the best known deterministic bound for Unique-k-SAT.

We use the Method of Small Sample Spaces (cf. [1]) to prove that the algorithm can be adapted to enumerate some small probability space yielding a deterministic running time which equals to the former expected running time up to a subexponential factor. Moreover, this means that the best bound for Unique-3-SAT is not only a deterministic one, but also better than the best known randomized bound for 3-CNF formulas with 'many' solutions, cf. Chapter 5.

4.2 Method of Small Sample Spaces

Before we discuss the derandomization of the PPSZ algorithm, we deal with the Method of Small Sample Spaces.

To prove the existence of a combinatorial structure with some desired properties, one may construct a finite probability space and prove that a random element from this probability space has all the desired properties with positive probability. Since the probability that we may select such an element is positive, there must exist such an element. This is well known as the Probabilistic Method. The first application is due to Szele in 1943, e.g. see [1, Chapter 2]. Starting in 1947, Erdös applied this method to a number of problems and he can be called the Pioneer of the Probabilistic Method.

Furthermore, we can similarly apply this method to another setting. Given a random variable X on some finite probability space, and let X have expectation $\overline{X}$ on this probability space. Then there must exist some outcome of X that is at least $\overline{X}$ and some outcome that is at most $\overline{X}$.

But, how can we select such an element in a deterministic way? We need to enumerate the probability space. Since the probability space is finite, we can go through all elements in the probability space, check the properties, and eventually, we will find an element satisfying the desired properties. In the worst case, we have to enumerate the entire probability space. Hence if the probability space is

vast (e.g. exponential size), this can take a while. But, sometimes, we can reduce the probability space to another one which is a great deal smaller (e.g. polynomial size), and thus, we can save a lot of time on the enumeration. This is called the Method of Small Sample Spaces.

As a simple example, we study the MAX-k-SAT-Problem in short. Given a k-CNF formula G, let $m(G)$ be the maximum number of clauses that can be simultaneously satisfied by some assignment. The problem of selecting an assignment that satisfies at least $m(G)$ clauses is known as the MAX-k-SAT problem, which is NP-complete for $k \geq 2$.

Assume that every clause in G is made of (exactly) k distinct literals. We will show that then there is a deterministic polynomial-time algorithm that finds an assignment that always satisfies at least $\lfloor (1 - 2^{-k})|G| \rfloor$ clauses, i.e. at least $\lfloor (1 - 2^{-k}) \rfloor m(G)$ since $m(G) \leq |G|$. At first, consider the randomized algorithm shown in Figure 4.1

Algorithm $Rand - MkS$(**CNF** G)
1 **return** an assignment to G uniformly at random

Figure 4.1: Trivial Algorithm for Max-k-SAT

The algorithm looks trivial, but we prove:

Proposition 4.1. *For a CNF formula G on n variables, where every clause contains k distinct literals, Algorithm $Rand-MkS$ returns an assignment that satisfies at least $\lfloor (1 - 2^{-k})|G| \rfloor$ clauses on the average in expected polynomial time.*

Proof. Select an assignment β uniformly at random. For every clause C, let X_C denote the binary random variable indicating whether clause C is satisfied. Moreover, let X denote the random variable that counts the number of clauses satisfied by β. By linearity of expectation, we have:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{C \in G} X_C\right] = \sum_{C \in G} \mathbb{E}[X_C]$$

Fix a clause $C \in G$. Because no variable appears twice in C, every literal in C satisfies C with probability $1/2$ independently from each other literal in C. C is wrong if every literal in C is wrong under β, which happens with probability 2^{-k}.

Hence we have $\mathbb{E}[X_C] = 1 - 2^{-k}$. We conclude that

$$\mathbb{E}[X] = |G| \cdot \left(1 - 2^{-k}\right)$$

is true, which finishes the proof. □

Now, we will try to derandomize this trivial algorithm in order to select an assignment that satisfies at least $\lfloor (1 - 2^{-k})|G| \rfloor$ clauses. Since we expect at least $\lfloor (1 - 2^{-k})|G| \rfloor$ to be satisfied, we could just try all assignments until we find a sufficient assignment. Alas, this has running time 2^{n_G}. But, can we build a probability space for assignments which is smaller while still having the same number of satisfied clauses expected for a random assignment?

Looking at the proof of the preceding proposition, we see that each random variable X_C depends only on the assignment of k variables. Thus Theorem 2.1 from [1, Chapter 15] is just what we need:

Theorem 4.2. *For every $n \geq w \geq 1$, there exists a probability space $\Omega(n, w)$ of size $\mathcal{O}\left(n^{w/2}\right)$ and w-wise independent random variables y_1,...,y_n over $\Omega(n, w)$ each of which takes 0 or 1 each with probability 1/2. $\Omega(n, w)$ can be constructed in polynomial time.*

We define a probability space $\Omega(G, w)$ that produces random assignments for G in the following way. Let v_i with $i \in [n_G]$ denote an arbitrary, but fixed ordering of $vars(G)$. Instantiate some w-wise independent random binary variables y_1,...,y_{n_G} over some fixed $\Omega(n, w)$ as denoted in the preceding theorem. The corresponding random assignment β is obtained by setting $\beta(v_i)$ to y_i for every $i \in [n_G]$. By this construction, the following is true:

Corollary 4.3. *For $|vars(G)| \geq w \geq 1$, there exists a probability space $\Omega(G, w)$ for random assignments β with the following properties. $\Omega(G, w)$ has size $\mathcal{O}\left(n_G^{w/2}\right)$, and the assignments in $\Omega(G, w)$ can be enumerated in polynomial time. Moreover, for $v \in vars(G)$, $\beta(v)$ takes 0 or 1 each with probability 1/2. Finally, for every set $V \subseteq vars(G)$ with $|V| \leq w$, all values $\beta(v)$ for $v \in V$ are independent from each other, i.e. the values of β are w-wise independent.*

Using the 'small' probability space $\Omega(G, w)$, we can derandomize Algorithm $Rand - MkS$ and obtain the algorithm shown in Figure 4.2.

```
Algorithm MkS(k-CNF G)
1    for each assignment β ∈ Ω(n_G, k) do
2        if β satisfies at least ⌊(1 − 2^{−k})|G|⌋ clauses then return β
```

Figure 4.2: Deterministic Algorithm for Max-k-SAT

Proposition 4.4. *For a k-CNF formula G on n variables, where every clause contains k distinct literals, Algorithm MkS returns an assignment that satisfies at least $\lfloor(1-2^{-k})|G|\rfloor$ clauses in deterministic polynomial time.*

Proof. Recall the definition of X and X_C in the proof of Proposition 4.1. Since the values of an assignment in $\Omega(G,k)$ are k-wise independent, the analysis for $\mathbb{E}[X]$ with respect to $\Omega(G,k)$ is the same as in the proof of Proposition 4.1. Hence there exists an assignment in $\Omega(G,k)$ that satisfies at least $\lfloor(1-2^{-k})|G|\rfloor$ clauses and that assignment (or some other with the same properties) will be found in polynomial time by the algorithm. □

4.3 Algorithm PPSZ Derandomized

```
Algorithm dPPSZ(k-CNF G, integer d, integer L, integer t)
1    G := do k^d-bounded resolution on G
2    for each π = π(α) with α ∈ Ω(G, ((k − 1)^{d+1} − 1)/(k − 2), L) do
3        for each bit string b of size t do {
4            G' := G
5            repeat as long as there is an unused bit in b {
6                v := next unused variable in π
7                if G' contains a unit clause v or v̄
8                then G' := G'|_v resp. G' := G'|_{v̄}
9                else Choose G' := G'|_v or G' := G'|_{v̄} depending on the next unused
                     bit of b being 1 or 0
10           }
11           if G' is the empty formula then return true
12       }
13   return false
```

Figure 4.3: Derandomized Algorithm of Paturi, Pudlak, Saks, and Zane

In Figure 4.3, we have a derandomized form of the PPSZ algorithm. Note that π denotes a permutation of the variables of G computed using a polynomial time function $\pi(\alpha)$ where α is a member of some set $\Omega(G, w, L)$. Both objects will be introduced during the analysis.

The rest of this chapter will focus on the analysis of this algorithm. Roughly speaking, we will adapt the original analysis of the PPSZ algorithm where necessary in order to prove the bound for the derandomized version.

The only difference between the PPSZ algorithm and this one is that the PPSZ algorithm chooses a permutation π of $vars(G)$ and a bit string b of length n_G uniformly at random.

4.4 Analysis of Algorithm $dPPSZ$

4.4.1 Deterministic Bounds for Unique-k-SAT

Fix some uniquely satisfiable k-CNF formula G. In the algorithm, we use a set $\Omega(G, w, L)$ with $w = w_{k,d} := ((k-1)^{d+1} - 1)/(k-2)$, the set will be defined in Section 4.4.2. For now, let us use it as a black box probability space that can be used to draw permutations π of $vars(G)$ at random so that the following lemma is satisfied, which is proved in Section 4.4.4:

Lemma 4.5. *Let d and L be integers and let G be a uniquely satisfiable k-CNF formula with more than d variables. Fix some variable v of G. Assume that Algorithm $dPPSZ$ reaches variable v and all variables before v in π were set according to β. At this step, there will be a unit clause for v with probability at least $\lambda_{k,d,L}$ with*

$$\lambda_{k,d,L} = \frac{\mu_k}{k-1} - \epsilon_{k,d,L}$$

with

$$\mu_k = \sum_{j=1}^{\infty} \frac{1}{j\left(j + \frac{1}{k-1}\right)}$$

where $\epsilon_{k,d,L}$ can be made arbitrary small positive by choosing L and d large enough.

With respect to $\pi = \pi(\alpha)$ for random α from $\Omega(G, w, L)$, let the random set F contain all variables v for which there is a unit clause for v when the algorithm processes v. For each variable v, let F_v denote the binary random variable indicating membership of v in F. By the preceding lemma, the probability that F_v is 1 is at least $\lambda_{k,d,L}$. By linearity of expectatation, we have:

$$\begin{aligned}
\mathbb{E}[F] &= \mathbb{E}\left[\sum_{v \in vars(G)} F_v\right] \\
&= \sum_{v \in vars(G)} \mathbb{E}\left[F_v\right] \\
&= \sum_{v \in vars(G)} \mathbb{P}\left[F_v = 1\right] \\
&\geq \sum_{v \in vars(G)} \lambda_{k,d,L} \\
&= \lambda_{k,d,L} n_G
\end{aligned}$$

Because we try all elements of $\Omega(G, w, L)$, we must encounter at least on permutation π where the number of variables in F is at least $\lambda_{k,d,L} n_G$. Now, assume that the bit string b is chosen so that all bits used for variables agree with β. But, because at least $\lfloor \lambda_{k,d,L} n_G \rfloor$ variables are determined using unit clauses, we only need at most $n_G - \lfloor \lambda_{k,d,L} n_G \rfloor$ bits from b. We conclude that if we set $t = \lceil n_G - \lambda_{k,d,L} n_G \rceil$, we will face that *good* bit string.

Enumerating all bit strings of length t takes time at most $\mathcal{O}\left(2^t\right)$. In Section 4.4.2, we will prove that $\Omega(G, w, L)$ can be constructed and enumerated in polynomial time in $\mathcal{O}\left(n_G^{Lw/2}\right)$ which is a polynomial in n_G for constant k, d, and L. CNF formulas which do not satisfy the precondition of Lemma 4.5, i.e. which have at most d variables, can be solved in polynomial time since d is a constant. Finally, we can state the main result of this chapter:

Proposition 4.6. *For a uniquely satisfiable k-CNF formula G, integers $d > 0$, $L > 0$, and $t = \lceil n_G - \lambda_{k,d,L} n_G \rceil$, Algorithm dPPSZ finds the satisfying assignment in deterministic running time at most*

$$\mathcal{O}\left(2^{\left(1 - \frac{\mu_k}{k-1}\right) n_G + \epsilon_{k,d,L} n_G}\right)$$

where $\epsilon_{k,d,L}$ can be made arbitrary small positive by choosing L and d large enough.

Corollary 4.7. *For uniquely satisfiable 3-CNF and 4-CNF formulas on n variables, the satisfying assignment can be found in deterministic running time at most $\mathcal{O}(1.3071^n)$ resp. $\mathcal{O}(1.4699^n)$.*

4.4.2 Small Probability Space for Variable Ordering

The (original) PPSZ algorithm chooses a permutation π uniformly at random, but in Section 4.4.3, we will see that we need only randomness with respect to a subset of the variables, which has size bounded by a constant w for the Unique-k-SAT case. Hence we could just draw n_G integers from a finite pool (to have a finite probability space) of w-wise independent integers and order π according to the rank of these. But, what do we do if we draw the same integer for two variables? Fortunately, the bigger the pool is, the less likely it is for two variables to clash. Guided by this idea, we will discuss a handy construction of π and show some useful properties. For that, our basic tool is Corollary 4.3.

Let us start with a mapping α which maps each variable in G to a value in $[0, 1)$. Given α, we construct a permutation $\pi = \pi(\alpha)$ of $vars(G)$ so that $\alpha(u) < \alpha(v)$ implies that u occurs before v in π. Such a permutation can clearly be constructed in a deterministic way by ordering $vars(G)$ due to the values α takes on them with some arbitrary deterministic rule if two take the same value.

In order to have a random permutation π that is distributed uniformly on the set of all possible permutations of $vars(G)$, we need to draw a value $\alpha(v)$ uniformly at random from $[0, 1]$ (with infinite precision) independently for each $v \in vars(G)$. For two variables $u, v \in vars(G)$, the probability that the binary encodings of $\alpha(u)$ and $\alpha(v)$ differ at the ith bit, but not before, is 2^{-i}. Note that this happens with positive probability for arbitraily large i. Hence such a probability space is not derandomizable in finite time. To solve this problem, we approximate $[0, 1]$ with a large discrete subset, i.e. for fixed $L > 0$, every $\alpha(v)$ must be encoded using L bits. Then we have only 2^L possible values for each $\alpha(v)$. However, this is still too much since for n variables, this probability space still has size 2^{nL} if each $\alpha(v)$ is chosen independent from each other. As noted at the beginning of this subsection, we only need independence with respect to a *small* subset of variables, so Corollary 4.3 provides all we need.

We construct a random α in the following way. Define integers $w > 0$, $L > 0$. Now, let Ω_1,...,Ω_L be independent instances of the probability space $\Omega(G, w)$ from Corollary 4.3.

For each $l \in [L]$, let β_l denote a random assignment from Ω_l. Define

$$\alpha(v) = \sum_{l \in [L]} 2^{-l} \beta_l(v)$$

for variable $v \in vars(G)$, i.e. $\beta_l(v)$ is seen as a binary encoding for $\alpha(v)$ with length L. Let $A^{(L)}$ be the set of all possible rational values $\alpha(.)$ can take. On the one hand, for fixed $v \in V$, the values of $\beta_1(v)$,...,$\beta_L(v)$ are fully independent since they are drawn from independent probability spaces. Hence each value in $A^{(L)}$ has equal probability to be chosen for $\alpha(v)$. On the other hand, for fixed l, the values of β_l are w-wise independent since they are drawn using an instance of $\Omega(G, w)$. Because this holds for every l independently, the values of α are w-wise independent.

We conclude that the construction above yields w-wise independent values for α, where each of them is uniformly distributed on $A^{(L)}$. So, let $\Omega(G, w, L)$ denote the probability space for random α as constructed above. We have:

Lemma 4.8. *$\Omega(G, w, L)$ can be constructed with size $\mathcal{O}\left(n_G^{Lw/2}\right)$ and in polynomial time in its size.*

Proof. By Corollary 4.3, each Ω_l has size $\mathcal{O}\left(n_G^{w/2}\right)$ and can be constructed in polynomial time in its size. $\Omega(G, w, L)$ is a product of the probability spaces Ω_1,...,Ω_L. Thus $\Omega(G, w, L)$ has size $\mathcal{O}\left(n_G^{Lw/2}\right)$ and can be enumerated in polynomial time by enumerating the product space. □

Fix some arbitrary $v \in vars(G)$ and fix some arbitrary $V \subseteq vars(G) - v$ with $|V| < w$. We want to have a lower bound for the probability that a variable u in V occurs before v in π. The fact that $\alpha(u) = \alpha(v)$ could hold makes the analysis a little bit complicated because whether u occurs before v in π is arbitrary then. Fortunately, this is not very likely and the probability for that to happen decreases with increasing L. Therefore, we call v *unique* with respect to V if $\alpha(u) \neq \alpha(v)$ holds for all $u \in V$. Clearly, the probability that v is unique with respect to V is $(1 - 2^{-L})^{|V|}$ since all values $\alpha(u)$ for $u \in V + v$ are independent from each other.

Now, assume that we already know that v is unique with respect to V. Then all $\alpha(u)$ for $u \in V$ can still be seen as being drawn independently at random from $A^{(L)} - \alpha(v)$. Again, fix a variable u in V. Under the condition that v is unique with respect to V, the probability that $\alpha(u) < \alpha(v)$, i.e. that u occurs before v in π, is equal to $\alpha(v) \cdot 2^L/(2^L - 1)$. This comes from the fact that we have $\alpha(v) \cdot 2^L$ elements in $A^{(L)}$ which are strict less than $\alpha(v)$ and because the condition allows all $2^L - 1$ elements of $A^{(L)} - \alpha(v)$ to be chosen for $\alpha(u)$ uniformly at random. Let us sum up:

Lemma 4.9. *Let $v \in vars(G)$ be a variable and $V \subseteq vars(G)$ be a set of variables with $|V| < w$ and $v \notin V$. Then the following are true:*

1. *The probability that v is unique with respect to V is $(1 - 2^{-L})^{|V|}$.*
2. *Given that v is unique with respect to V, all $\alpha(u)$ with $u \in V$ are independent, and for each $u \in V$, the probability that $\alpha(u) < \alpha(v)$ holds is equal to $\alpha(v) \cdot 2^L/(2^L - 1)$.*

4.4.3 Admissible Trees

Before we can go back to Unique-k-SAT, we need the notion of an admissible tree and have to prove some important properties.

Fix some variable $v \in vars(G)$. Let T be a tree where the root is labeled by v. Each node of the tree can have a label in $vars(G)$ or it is unlabeled. Moreover, for each path from a leaf to the root, no integer occurs more than once as a label. Then T is called an *admissible tree*. The depth of T is the maximum distance from any leaf to the root, e.g. a tree containing only one node has depth 0. We limit the depth of an admissible tree to d and we limit the number of children of each node to $k-1$. Then T has at most $w_{k,d}$ nodes. A *cut* A is a set of nodes that does not include the root, and every path from the root to a leaf includes a node in A. Figure 4.4 shows an example of an admissible tree, and moreover, encircled nodes form a cut.

Let $\pi = \pi(\alpha)$ where α is drawn from $\Omega(n, w_{k,d}, L)$ at random. We say a cut A *happens* if all variables corresponding to labeled nodes of A occur before v in π. We like to calculate the probability that at least one cut happens with respect to

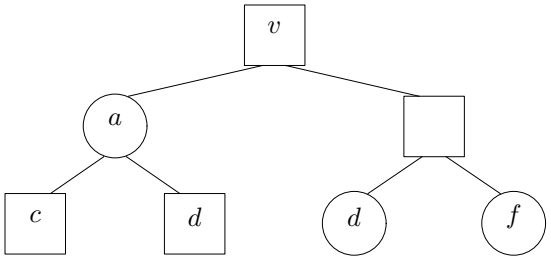

Figure 4.4: Admissible Tree with a Cut

a random permutation. In the example above, the cut happens if c and d appear before v in π. But, also the cuts $\{a\}$, $\{c, d, f\}$, or $\{a, d, f\}$ could happen. If some node appears after v, then we have to look in the subtree for a cut, e.g. a appears before v, otherwise c and d have to appear before v. Roughly speaking, taking a subtree rooted at some node x, a cut in that subtree happens if x appears before v in π or each of the subtrees rooted at the children of x have a cut happening. We will formalize this idea in the following.

We say that v is *unique with respect to* T, if v is unique with respect to the set of variables occurring as labels in T excluding v. So, given that v is unique with respect to T and some subtree T_0 of T, we denote with $Q_{T_0}(r)$ the probability that at least one of the possible cuts of T_0 happens and, conveniently, where we use r to stand for $\alpha(v) \cdot 2^L/(2^L - 1)$. We will establish a lower bound for $Q_{T_0}(r)$:

Lemma 4.10. *Given an admissible tree T with root labeled by v, let T_0 be some subtree of T with more than one node, and let $T_1, ..., T_t$ be the subtrees rooted at the labeled children of the root of T_0. Let $u_1, ..., u_t$ be the labels of their roots. Then it is true that*

$$Q_{T_0}(r) \geq \prod_{i=1}^{t} (r + (1 - r)Q_{T_i}(r))$$

holds where the empty product is interpreted as 1.

Proof. [1] Consider the case that $t = 0$. Since T_0 has at least one child, there is a cut in the tree. Because no child is labeled, the cut is empty, which corresponds to an empty event, which occurs with probability 1.

[1] Note that this proof is almost the same like the one for Lemma 4 in [13], which can be found in [15].

Hence, we assume $t \geq 1$. Let U be the set of variables occurring as labels in T. Since $|U| \leq w_{k,d}$ and since we have chosen α from $\Omega(n, w_{k,d}, L)$, we can apply Lemma 4.9 using $U - v$ for V. Thus we have that the probability of $\alpha(u_i) < \alpha(v)$ is equal to $r = \alpha(v) \cdot 2^L/(2^L - 1)$ since v is unique with respect to T.

Consider the event that $\alpha(u_i) < \alpha(v)$ holds and the event that a cut in T_i happens. Because a subtree of an admissible tree is also admissible, u_i does not occur anywhere else in T_i. Hence both events are independent, causing their union, denoted with K_i, to have probability $r + (1 - r)Q_{T_i}(r)$.

To finish the proof, we have to show that $\mathbb{P}[\bigcap_{i=1}^{t} K_i] \geq \prod_{i=1}^{t} \mathbb{P}[K_i]$. At first, let us recall some standard correlation inequality, which is a special case of the FKG-inequality (cf. Theorem 3.2 in [1, Chapter 6]):

Lemma 4.11. *Let N be a finite set and let $\mathcal{A}$ and $\mathcal{B}$ be two monotone increasing families of subsets of N, i.e. each super-set of a set in $\mathcal{A}$ resp. $\mathcal{B}$ is also contained in $\mathcal{A}$ resp. $\mathcal{B}$. Draw a random set $M \subseteq N$ by choosing each u in N independently with probability p. Then it is true that*

$$\mathbb{P}[M \in \mathcal{A} \cap \mathcal{B}] \geq \mathbb{P}[M \in \mathcal{A}]\mathbb{P}[M \in \mathcal{B}].$$

We set N to be the set of all variables occurring as labels in T_0, but we exclude v. Moreover, we determine M as follows. For all $u \in N$, we include u in M if it occurs before v in π. These events occur independently each with probability r. Let $\mathcal{W}_i$ denote the family of all subsets of N that imply K_i, i.e. all sets of variables $W \subseteq vars(G)$ for which holds that K_i happens when all $u \in W$ occur before v in π. Because K_i only depends on variables in N, M is a member of $\mathcal{W}_i$ if and only if the event K_i happens. Clearly, $\mathcal{W}_i$ is monotone increasing since all supersets of a set that implies K_i also imply K_i, i.e. more variables than necessary before v in π is not bad. The set $\mathcal{V}_i = \bigcap_{j=1}^{i-1} \mathcal{W}_i$ is also monotone increasing. We plug $\mathcal{V}_i$ and $\mathcal{W}_i$ as $\mathcal{A}$ and $\mathcal{B}$ in the lemma and obtain

$$\begin{aligned}\mathbb{P}[M \in \mathcal{V}_{i+1}] &\geq \mathbb{P}[M \in \mathcal{V}_i]\mathbb{P}[M \in \mathcal{W}_i] \\ &\geq \prod_{j \leq i} \mathbb{P}[M \in \mathcal{W}_j].\end{aligned}$$

Because $M \in \mathcal{V}_{t+1}$ means that the event $\bigcap_{i=1}^{t} K_i$ happens, we can conclude that

$$\mathbb{P}\left[\bigcap_{i=1}^{t} K_i\right] \geq \prod_{i=1}^{t} \mathbb{P}[K_i]$$

is true, which completes the proof. □

Thus we only need to compute the recursion in the lemma for a lower bound on $Q_T(r)$. Paturi et al established in Section 3.4 in [13] that

$$\lim_{d\to\infty} \int_0^1 Q_{T(d)}(r')dr' \geq \frac{\mu_k}{k-1}$$

holds where $T(d)$ is an admissible tree of depth d. However, the integral in the left hand side may be less than the right hand side of the inequality. But, the limes tells us that for any given $\epsilon > 0$, we may choose some large d_ϵ so that for all $d > d_\epsilon$, the integral is at least $\frac{\mu_k}{k-1} - \epsilon$.

Hence we need to express the probability that a cut happens in a way that we can apply the preceding inequality. In the following, we show that for L tending to infinity, the uniqueness of v is almost 'neglectable'. So, let $Q'_T(\alpha(v))$ denote the probability that at least one of the possible cut of T happens given that only the value of $\alpha(v)$ is known with respect to random $\alpha \in \Omega(n, w_{k,d}, L)$. Comparing to $Q_T(r)$, we dropped uniqueness.

We have that

$$Q'_T(\alpha(v)) = Q_T\left(\alpha(v) \cdot 2^L / \left(2^L - 1\right)\right) \cdot \left(1 - 2^{-L}\right)^{w_{k,d}}$$

holds since the probability that v is unique with respect to T is equal to $(1 - 2^{-L})^{w_{k,d}}$. For L tending to infinity, both factors involving L in the preceding equation tend to 1. Moreover, $Q_T(.)$ is a continuous function on $[0, 1]$ (cf. [13]).

So we can conclude that

$$\lim_{L\to\infty} Q'_T(\alpha(v)) = Q_T(\alpha(v))$$

holds.

Again, for any $\epsilon > 0$, we may choose L_ϵ so that for all $L > L_\epsilon$, $Q'_T(\alpha(v))$ is at least $Q_T(\alpha(v)) - \epsilon$ for all $\alpha(v)$ in $A^{(L)}$. So, let Q'_T denote the probability that at

least one of the possible cuts of T happens for random $\alpha \in \Omega(n, w_{k,d}, L)$, i.e. $\alpha(v)$ is also random now.

For $L > L_\epsilon$, we have:

$$\begin{aligned} Q'_T &= \sum_{r' \in A^{(L)}} Q'_T(r') \cdot \mathbb{P}[\alpha(v) = r'] \\ &\geq \sum_{r' \in A^{(L)}} (Q_T(r') - \epsilon) \cdot 2^{-L} \\ &\geq \sum_{r' \in A^{(L)}} Q_T(r') \cdot 2^{-L} - \epsilon \\ &\geq 2^{-L} \sum_{l=0}^{2^L - 1} Q_T\left(l/2^L\right) - \epsilon \end{aligned}$$

For L tending to infinity, we have:

$$\begin{aligned} \lim_{L \to \infty} Q'_T &\geq \lim_{L \to \infty} \sum_{l=0}^{2^L - 1} Q_T\left(l/2^L\right) - \epsilon \\ &\geq \int_0^1 Q_T(r') dr' - \epsilon \end{aligned}$$

Clearly, by choosing a large L, we can get Q'_T arbitrary close to the right hand side of the inequality. Thus we have proved:

Lemma 4.12. *Given an admissible tree T of depth d, the probability that at least one of the possible cuts of T happens is at least $\lambda_{k,d,L}$ with*

$$\lambda_{k,d,L} = \frac{\mu_k}{k-1} - \epsilon_{k,d,L}$$

where $\epsilon_{k,d,L}$ can be made arbitrary small positive by choosing L and d large enough.

4.4.4 Critical Clause Trees

Now, let us draw the connection between Unique-k-SAT and our abstract admissible trees.

We call a clause $C \in G$ a *critical clause* for v if the only true literal in C with respect to β is the one corresponding to v, i.e. flipping the value assigned to v in β would make C instantly false.

Algorithm $dPPSZ$ applies k^d-bounded resolution to G and then steps through the variables ordered by a permutation π. Assume that the bit string b is chosen so that all bits used for variables agree with β. When the algorithm reaches a variable v and there is a critical clause C for v so that the variables $vars(C) - v$ occur before v in π, C has been reduced to a unit clause for v so that the algorithm can immediately determine the right assignment to v. But, when is there a clause C meeting this condition? We need the notion of a critical clause tree.

We call an admissible tree T with root labeled by v a *critical clause tree* for v if for each cut A of T, there exists a critical clause for v in G where $vars(C) - v$ contains only variables which occur as labels in A.

An example of a critical clause tree follows. Consider a 3-CNF formula G that is satisfied if and only if 0 is assigned to all variables. Let $\overline{v}ab$, $\overline{a}cd$, and $\overline{b}de$ be clauses in G. Then the tree shown in Figure 4.5 is a critical clauses tree for v with depth two.

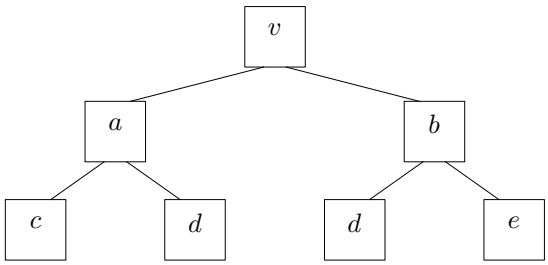

Figure 4.5: Critical Clause Tree for v

Clearly, every cut in the tree corresponds to a critical clause in G after we applied 9-bounded resolution. For example, the cut $\{c, d, b\}$ corresponds to clause $\overline{a}cdb$ which can be derived by resolving $\overline{v}ab$ with $\overline{a}cd$. Indeed, the existence of such a critical clause tree for a sufficient large formula is guaranteed by Lemma 2 in [13]:

Lemma 4.13. *Let G be a uniquely satisfiable k-CNF formula with more than d variables. Apply k^d-bounded resolution to G. For each $v \in vars(G)$, there exists a critical clause tree for v with depth d.*

We conclude that if a cut of a critical clause tree of v happens with respect to π, there must be a critical clause C corresponding to that cut meeting the condition.

By Lemma 4.12, this has probability at least $\lambda_{k,d,L}$, and finally, we have proved what was claimed in Lemma 4.5.

4.5 Conclusion

We derandomized the uniquely satisfiability case of the PPSZ algorithm using an approximation of the uniform distribution on $[0, 1]$ using a discrete subset of $[0, 1]$ and showed that we can come arbitrary close to the randomized bound by making the discrete subset large enough.

We can also conclude that a sufficient pseudo-random number generator can be used for the PPSZ algorithm instead of true randomness for the unique satisfiability case.

Chapter 5

Improved Bound for the PPSZ/Schöning-Algorithm for 3-SAT

5.1 Introduction

Using an elegant-simple random-walk algorithm, Schöning showed in 1999 that a satisfying assignment for a satisfiable 3-CNF formula G can be found in $\mathcal{O}\left((4/3+\epsilon)^{n_G}\right)$ expected running time, cf. Section 2.3.2 or [24].

In [13], Paturi, Pudlak, Saks, and Zane proved that for a uniquely satisfiable 3-CNF formula G, the solution can be found in $\mathcal{O}\left(1.3071^{n_G}\right)$ expected running time at most. We refer to their algorithm as the PPSZ algorithm, cf. Section 2.2.3. This is the best randomized bound known for Unique-3-SAT and it is possible to derandomize it, essentially yielding the same bound deterministically, cf. Chapter [19]. But paradoxically, the bound gets worse when the number of solutions increases.

In 2004 in [8], Iwama and Tamaki combined both algorithms and were able to prove a randomized bound of $\mathcal{O}\left(1.3238^{n_G}\right)$ for 3-SAT for the combined algorithm. The algorithm was already presented in Section 2.4.2. Up to now, this was best known randomized bound for 3-SAT. Their bound automatically improves to $\mathcal{O}\left(1.32266^{n_G}\right)$ by modifying their analysis to use the latest bound for the PPSZ algorithm that was presented in Corollary 14 in [15]. However, we tune the bound to improve their result to $\mathcal{O}\left(1.32216^{n_G}\right)$.

5.2 The Analysis

5.2.1 Main Result

Iwama and Tamaki proved in [8] that the expected number of repetitions of $COMB(G, d)$ is $\mathcal{O}(1.3238^{n_G})$ (resp. $\mathcal{O}(1.32266^{n_G})$ as noted in the introduction) for a satisfiable 3-CNF formula G and some large but fixed d. We improve that result to:

Proposition 5.1. *For a satisfiable 3-CNF formula G and some large but fixed d, the expected number of repetitions of $COMB(G, d)$ is $\mathcal{O}(1.32216^{n_G})$.*

In Section 5.2.2, we show how to disassemble the analysis for the combined algorithm into two separate ones. We provide bounds for both algorithms in Section 5.2.3 resp. Section 5.2.4. After that, we combine the bounds for both algorithm to prove the main result in 5.2.5. Finally, we consider some technical details in Section 5.2.6 and 5.2.7 that were left out in Section 5.2.4.

5.2.2 Disassembling $COMB$

For a set of variables $D \subseteq vars(G)$ and some assignment β of G, we define the set $B(D, \beta)$ to be the set of all assignments that agree with β on at least the variables in D, i.e. the subcube of the solution space where the variables in D are fixed to their values according to β and the others take all possible combinations.

For example, assume $vars(G) = \{a, b, c, d\}$, $D = \{a, b\}$ and let β assign 0 to all variables in $vars(G)$. Then $B(D, \beta)$ contains the following assignments:

a	b	c	d
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1

From [15], we know:

Lemma 5.2. *For a satisfiable k-CNF formula G, there exists a family of sets of variables $(D_\beta : \beta \in sat(G))$ so that the family of the corresponding subcubes*

$(B(D_\beta, \beta) : \beta \in sat(G))$ partitions the solution space (i.e. covering completely while being pairwise distinct). Moreover, it is true that

$$\sum_{\beta \in sat(G)} 2^{-|D_\beta|} = 1.$$

Proof. We show how to construct such a family of sets of variables. At first, we construct a sequence of binary trees T_i. Each node in the tree is labeled with a set of assignments, and each edge in the tree is labeled by a variable.

Start with a tree T_0 consisting of one node labeled by $sat(G)$. Note that we keep the invariants that every node is labeled by a *non-empty* set of assignments and that on every path from the root to a leaf no variable occurs twice as an edge label.

Having constructed T_{i-1}, if all leaves are labeled by a set of assignments of cardinality one, then stop. Otherwise, let b be a leaf with label A where $|A| \neq 1$. By the invariant $A \neq \emptyset$, $|A| \geq 2$ holds. Then there must be a variable $v \in vars(G)$ and two assignments $\beta_0, \beta_1 \in A$ so that $\beta_0(v) = 0$ and $\beta_1(v) = 1$. Let v be such a pivot variable. We partition A in two sets A_0 and A_1 where A_0 gets all assignments $\beta \in A$ with $\beta(v) = 0$ and A_1 all with $\beta(v) = 1$. Because $\beta_0 \in A_0$ and $\beta_1 \in A_1$ holds, both sets are not empty. To obtain T_i from T_{i-1}, give b one child labeled with A_0 and one labeled with A_1. Moreover, label both new edges with variable v. Clearly, all assignments in A_0 assign the same value to v, so v will never be chosen again as a pivot variable, i.e. the first invariant is kept. Since the same holds for A_1, the second invariant is sustained, i.e. no variables occurs twice as edge label on a path from the root to a leaf.

In the final tree T, every leaf is labeled by a set containing exactly one satisfying assignment for G. Moreover, every two leaves are labeled by distinct sets. For every $\beta \in sat(G)$, we define D_β to be the set of variables occurring as labels on the edges from the leaf b_β containing $\{\beta\}$ as label to the root. Since no variable occurs twice on such a path, $|D_\beta|$ is equal to the depth of b_β (root has depth 0). Thus $\sum_{\beta \in sat(G)} 2^{-|D_\beta|} = 1$ can be proved by reverse induction since it is the sum of 2^{-d_b} over all leaves b where d_b is the depth of b.

Hence, it is left to prove that the family of subcubes $B(D_\beta, \beta)$ partitions the solution space. By induction, we prove that for every T_i, the leaves induce a family

of partitioning subcubes in the following way. For every leaf b in T, we set D_b to be the set of all variables occurring as edge labels from the path from b to the root. Furthermore, let β_b be an arbitrary assignment in the set used as label in b. Then, we assign subcube $B(D_b, \beta_b)$ to b.

For T_0, we have one leaf b and $D_b = \emptyset$. Thus the claim holds for T_0. So, let the result hold for non-final T_{i-1} and let b be the node used in the tree-construction process. Let b_1 and b_2 be the children added to b in order to obtain T_i from T_{i-1}. Clearly, the subcubes $B(D_{b_1}, \beta_{b_1})$ and $B(D_{b_2}, \beta_{b_2})$ are disjoint and the union of both equals to $B(D_b, \beta_b)$. Thus the claim holds for T_i. This finishes the proof since the family $B(D_\beta, \beta)$ in the final tree is the same as $B(D_b, \beta_b)$ with $b = b_\beta$. $\square$

So, throughout the rest of this chapter, fix (D_β) to be some arbitrary such family, and let (B_β) be the corresponding subcubes.

For some $\beta^* \in sat(G)$, let β be drawn uniformly at random from $\beta \in B_{\beta^*}$. Then the success probability of Algorithm $COMB$ is at least

$$\max\{\mathbb{P}[PPSZ : \beta \in B_{\beta^*}], \mathbb{P}[SCH : \beta \in B_{\beta^*}]\}$$

where $PPSZ$ and SCH denote the events that Algorithm $PPSZ(G, d, \beta)$ resp. $SCH(G, \beta)$ return some satisfying assignment. For a random β, the probability that $\beta \in B_{\beta^*}$ holds is equal to $2^{-|D_{\beta^*}|}$. Observe that β is still distributed uniformly on B_{β^*}. To get the success probability, we just sum up the success probabilities over all subcubes. Hence Algorithm $COMB$ succeeds with probability at least

$$\begin{aligned} &\sum_{\beta^* \in sat(G)} 2^{-|D_{\beta^*}|} \cdot \max\{\mathbb{P}[PPSZ : \beta \in B_{\beta^*}], \mathbb{P}[SCH : \beta \in B_{\beta^*}]\} \\ \geq &\min_{\beta^* \in sat(G)} \max\{\mathbb{P}[PPSZ : \beta \in B_{\beta^*}], \mathbb{P}[SCH : \beta \in B_{\beta^*}]\}. \end{aligned}$$

The inequality follows because we know that $\sum_{\beta^* \in sat(G)} 2^{-|D_{\beta^*}|} = 1$.

Therefore, to have a lower bound on the success probability, we can focus on computing a lower bound for the success probability given a single satisfying assignment β^* and its subcube. Hence, fix some $\beta^* \in sat(G)$, $B = B_{\beta^*}$, $D = D_{\beta^*}$, and $N = vars(G) \setminus D$ to the end of this chapter.

5.2.3 Bound for SCH

Schöning's Algorithm was already discussed in Section 5.2.3. Given a probability distriuition P for assignments and a satisfying assignment β^* of a satisfiable k-CNF formula G, Corollary 2.4 states that the success probability of Algorithm SCH is at least

$$\mathbb{E}\left[(k-1)^{-dist(\beta,\beta^*)-o(n_G)}\right]$$

where the expectation is calculated with respect to $P(\beta)$.

Conditioning on $\beta \in B_{\beta^*}$, we know that β agrees with β^* on D, whereby the assignment to N is uniformly distributed. So we have that

$$\begin{aligned}
&\mathbb{P}\left[SCH : \beta \in B_{\beta^*}\right] \\
&\geq \mathbb{E}\left[(k-1)^{-dist(\beta,\beta^*)-o(n_G)} : \beta \in B_{\beta^*}\right] \\
&= 2^{-o(n_G)} \prod_{v \in N} (\mathbb{P}[\beta(v) = \beta^*(v)] \cdot (k-1)^0 + \mathbb{P}[\beta(v) \neq \beta^*(v)] \cdot (k-1)^{-1}) \\
&= 2^{-o(n_G)} \prod_{v \in N} (1/2 \cdot (k-1)^0 + 1/2 \cdot (k-1)^{-1}) \\
&= (2-2/k)^{-|N|-o(n_G)} \\
&= (2-2/k)^{-n_G+|D|-o(n_G)} \\
&= 2^{-\sigma_k(n_G-|D|)-o(n_G)}
\end{aligned}$$

where $\sigma_k = \log_2(2-2/k)$.

Obviously, the success probability of Algorithm SCH increases with increasing $|D|$ as shown by the graph in Figure 5.1 for $k = 3$.

5.2.4 Bound for $PPSZ$

Let us define a *nice distribution* H. H is a nondecreasing, continuous mapping from $[0,1]$ to $[0,1]$ with $H(0) = 0$ and $H(1) = 1$. Moreover, it must be differentiable in all but at most a finite number of points. Finally, its derivate h must be uniformly bounded on $[0,1]$. We set

$$\begin{aligned}
\beta_H &= \int_0^1 h(r) \log_2(h(r))\, dr \\
\gamma_H &= \int_0^1 \min\{H(r)^{k-1}, R_k(r)\}\, dr
\end{aligned}$$

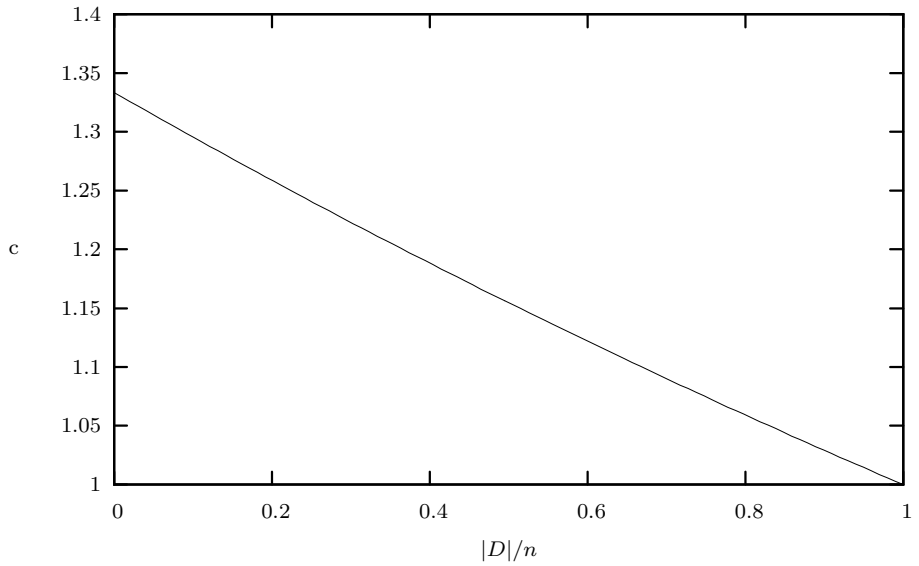

Figure 5.1: Basis c for Running Time $\mathcal{O}(c^n)$ of SCH

where $R_k(r)$ is the smallest non-negative x that satisfies $f_k(x,r) = x$ with $f_k(x,r) = (r + (1-r)x)^{k-1}$.

We will prove:

Lemma 5.3. *The probability that Algorithm $PPSZ$ finds a satisfying assignment given $\beta \in B_{\beta^*}$ is at least*

$$2^{-\beta_H|D| - (1-\gamma_H)(n_G - |D|) - \epsilon n_G - o(n_G)}$$

where ϵ can be made arbitrary small positive by choosing d large enough.

Of course, H is only a parameter in the analysis, it actually does not change the success probability of Algorithm $PPSZ$. However, β_H and γ_H are subject to H. Hence choosing H affects the upper bound on the number of repetitions needed for Algorithm $PPSZ$ in terms of $|D|/n_G$ as shown in Figure 5.2.

The graphs were computed using $H_\theta(r) = \min\{1, r/\theta\}$ and using $k = 3$. These nice distributions are analyzed in more detail in Section 5.2.7.

For $k = 3$, we are not able to find H in such a way that the success probability does not decrease for small $|D|$. But, we will see that we can tweak H so that the bound does not decrease too much until Schöning's can take over.

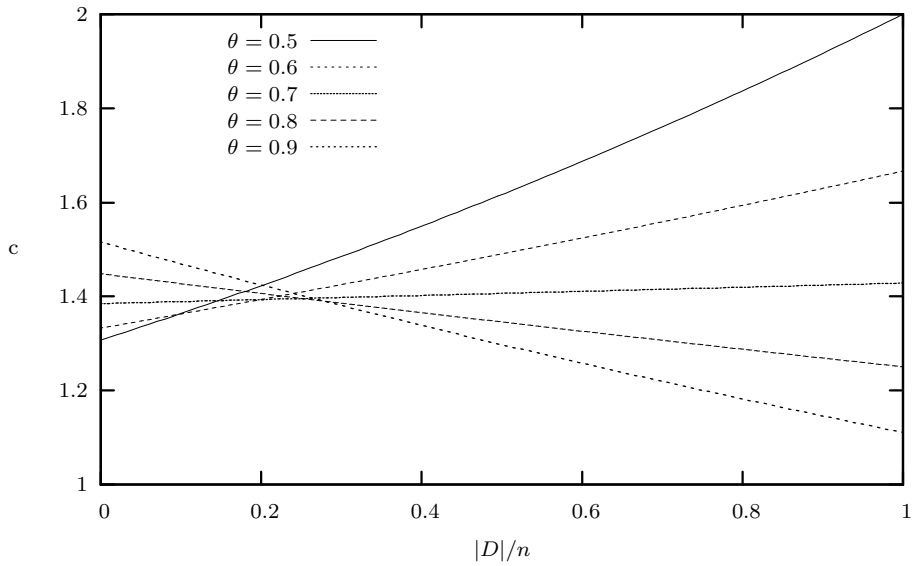

Figure 5.2: Basis c for Running Time $\mathcal{O}(c^n)$ of $PPSZ$

5.2.5 Reassembling $COMB$

We saw that the bound for SCH and the bound for $PPSZ$ depend on $|D|$, where the first increases with increasing $|D|$ and the second decreases with increasing $|D|$ if H is chosen appropriately. Hence we have to find the 'worst' $|D|$. Clearly, the worst case $|D|$ is attained when $\max\{\mathbb{P}[PPSZ : \beta \in B_{\beta^*}], \mathbb{P}[SCH : \beta \in B_{\beta^*}]\}$ is minimized.

Assuming that we have a distribution H so that the success probability of $PPSZ$ decreases with increasing $|D|$, we can compute the worst $|D|$ since the success probability of SCH increases with increasing $|D|$. Thus

$$\max\{\mathbb{P}[PPSZ : \beta \in B_{\beta^*}],\mathbb{P}[SCH : \beta \in B_{\beta^*}]\}$$

is minimized if

$$\begin{aligned}\sigma_k(n_G - |D|) + o(n_G) &= \beta_H|D| + (1-\gamma_H)(n_G - |D|) + o(n_G)\\ |D| &= n_G\frac{\sigma_k - 1 + \gamma_H}{\sigma_k - 1 + \gamma_H + \beta_H} + o(n_G)\end{aligned}$$

holds.

We have proved:

Proposition 5.4. *Let H be a nice distribution so that the bound for $PPSZ$ decreases with $|D|$, and let*

$$\delta = \frac{\sigma_k - 1 + \gamma_H}{\sigma_k - 1 + \gamma_H + \beta_H}$$

be well defined with $0 \leq \delta \leq 1$. For a satisfiable k-CNF formula G, the success probability of Algorithm $COMB(G, d)$ is at least

$$2^{-\sigma_k(1-\delta)n_G - \epsilon n_G - o(n_G)}$$

where ϵ can be made arbitrary small positive by choosing d large enough.

In Section 5.2.7, we will provide some H_3 with $\beta_{H_3} \leq 0.90925$, $\gamma_{H_3} \geq 0.61229$, and thus $\delta_3 \geq 0.02927$. Therefore, we have a lower bound of $\Omega\left(1.32216^{-n_G}\right)$ for the success probability of $COMB$ for a satisfiable 3-CNF formula G. This finishes the proof of the main result, Proposition 5.1.

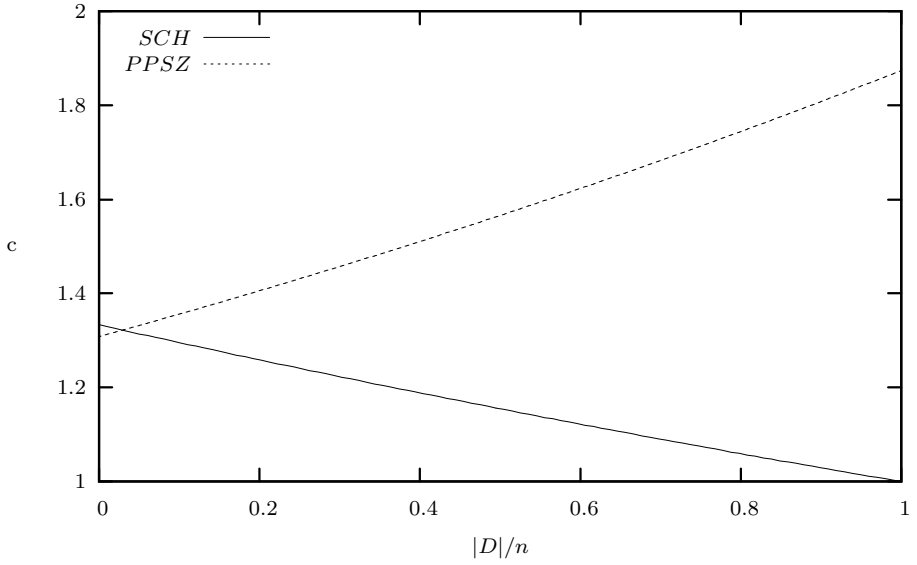

Figure 5.3: Basis c for Running Times $\mathcal{O}\left(c^n\right)$ of $PPSZ$ and SCH

The graph in Figure 5.3 shows the running time of the individual algorithms in terms of $|D|/n_G$. Note that the running time of the combined algorithms is the minimum of both.

As shown in the picture, the bound is worst when $|D|$ is about $\delta_3 \cdot n$.

5.2.6 Proof of the $PPSZ$ Bound

At first, we have to recapitulate some technical features around the PPSZ algorithm, some of which have been dismissed from the latest version of [15] because they are not necessary anymore by their analysis, but we need them for this one.[1]

For some permutation π, let $F(\pi)$ denote the set of variables in N that have been reduced to unit clauses during a run of Algorithm $PPSZ$. When β agrees with β^* on the variables in $vars(G) \setminus F(\pi)$, the algorithm will find β^*. Given that $\beta \in B_{\beta*}$, we know that β and β^* already agree on D. Thus we have:

$$\mathbb{P}[PPSZ : \beta \in B_{\beta^*}] \geq 2^{-N} \cdot \mathbb{E}\left[2^{F(\pi)}\right]$$

In order to have a good bound on the expectation, we will choose some subset Γ of the permutation space and compute instead:

$$\mathbb{P}[PPSZ : \beta \in B_{\beta^*}] \geq 2^{-N} \cdot \mathbb{P}[\pi \in \Gamma] \cdot \mathbb{E}\left[2^{|F(\pi)|} : \pi \in \Gamma\right]$$

A placement α is a function that maps each variable to a real value in $[0, 1]$. With $\pi(\alpha)$, we denote the permutation obtained by ranking the variables of G due to the values α takes on them with some arbitrary rule for breaking ties. Hence a uniform distribution of $\alpha(.)$ yields a uniform distribution of $\pi(\alpha(.))$.

Let v be a variable in N. For a set of placements Γ, we define $Q_\Gamma(r)$ to be the probability that v is in $F(\pi(\alpha))$ where α is a random placement from Γ having $\alpha(v) = r$. Then we have:

$$\mathbb{P}[v \in F(\pi(\alpha)) : \alpha \in \Gamma] \geq Q_\Gamma = \int_0^1 Q_\Gamma(r)\, dr$$

For every $\lambda \in [0, 1)$, we consider the set of placements $\Gamma_{H,\lambda,D}$ to be the set of all placements where for each $r \in [\lambda, 1]$, at least $H(r)|D|$ variables $v \in D$ have $\alpha(v) < r$.

From Lemma 26 in the old version of [15], we know that:

Lemma 5.5. *Define the recursive function $Q_k^d(r)$ by $Q_k^0(r) = 0$ and $Q_k^d(r) = f_k(Q_k^{d-1}(r), r)$ for $d > 0$. For $\Gamma = \Gamma_{H,\lambda,D}$ and $r \in [\lambda, 1]$, it is true that*

$$Q_\Gamma(r) \geq \min\{H(r)^{k-1}, Q_k^d(r)\} - \rho(H(r))$$

[1] An older version of [15] is still available in the citeseer-cache at http://citeseer.ist.psu.edu/paturi98improved.html.

where $\rho(x) = 0$ *for* $x \in \{0,1\}$ *and* $\rho(x) = \min\left\{\frac{k^{2d}}{|A|}\left(\frac{1}{x(1-x)}\right), 1\right\}$ *for* $x \in (0,1)$.

We compute Q_Γ for $\Gamma = \Gamma_{H,\lambda,D}$:

$$\begin{aligned} Q_\Gamma &= \int_0^1 Q_\Gamma(r)\,dr \\ &\geq \int_\lambda^1 Q_\Gamma(r)\,dr \\ &\geq \int_0^1 (\min\{H(r)^{k-1}, Q_k^d(r)\} - \rho(H(r)))\,dr - \lambda \\ &\geq \int_0^1 \min\{H(r)^{k-1}, Q_k^d(r)\}\,dr - \int_0^1 \rho(H(r))\,dr - \lambda \end{aligned}$$

Paturi et al evaluated $\int_0^1 \rho(H(r))\,dr$ to be $o(1)$ when $|D| \geq \sqrt{n}_G$ as n_G tends to infinity. We omit the analysis for $|D| \leq \sqrt{n}_G$ here since it is very likely for less than $\sqrt{n}_G$ variables to appear at the very beginning of the permutation π before all variable in N. For those, Paturi et al showed that $Q_\Gamma \geq \int_0^1 Q_k^d(r)$. We conclude:

$$Q_\Gamma \geq \int_0^1 \min\{H(r)^{k-1}, Q_k^d(r)\}\,dr - o(1) - \lambda$$

In Proposition 3 in [13], they also show that $Q_k^d(r)$ converges to $R_k(r)$ for every $r \in [0,1]$. Hence for every small positive ϵ, there exists a large d_ϵ so that for every $d \geq d_\epsilon$, $Q_k^d(r) \geq R_k(r) - \epsilon$ is true for all $r \in [0,1]$. We conclude that

$$\begin{aligned} Q_\Gamma &\geq \int_0^1 \min\{H(r)^{k-1}, R_k(r) - \epsilon\}\,dr - \lambda - o(1) \\ &\geq \int_0^1 \min\{H(r)^{k-1}, R_k(r)\}\,dr - \epsilon - \lambda - o(1) \end{aligned}$$

is true.

For the reader familiar with the details of [15], it is noticable that we have just proved a generalized version of Lemma 24 in the old version of [15]. In that lemma, they restricted $H(r)$ to be at most $R_k(r)^{1/(k-1)}$. The corresponding lemma in the latest version, Lemma 13 in [15], does not make use of any function H at all. Nevertheless, comparing the details, the new lemma looks like using $H(r) = \min\{r \cdot (k-1)/(k-2), 1\}$ in the old one. But, that H violates the (unnecessary) restriction $H(r) \leq R_k(r)^{1/(k-1)}$. Therefore, in the proof above, we only unified both approaches.

Since we have computed Q_Γ, we can consider the expected number of variables that will be in $F(\pi(\alpha))$:

$$\begin{aligned}&\mathbb{E}\left[2^{|F(\pi(\alpha))|} : \alpha \in \Gamma_{H,\lambda,D}\right]\\ &\geq 2^{\mathbb{E}[|F(\pi(\alpha))|:\alpha\in\Gamma_{H,\lambda,D}]}\\ &\geq 2^{\gamma_H|N|-\epsilon|N|-\lambda|N|-o(|N|)}\end{aligned}$$

Thus we conlude:

$$\mathbb{P}[PPSZ : \beta \in B_{\beta^*}] \geq \mathbb{P}[\alpha \in \Gamma_{H,\lambda,D}] \cdot 2^{-(1-\gamma_H)|N|-\epsilon|N|-\lambda|N|-o(|N|)}$$

For $\mathbb{P}[\alpha \in \Gamma_{H,\lambda,D}]$, Paturi et al proved a nice lower bound, cf. Lemma 23 in the old version of [15]:

Lemma 5.6. *For $\lambda > 0$, it is true that*

$$\mathbb{P}[\alpha \in \Gamma_{H,\lambda,D}] \geq 2^{-\beta_H|D|-o(|D|)}.$$

Because ϵ and λ are both arbitrary small positive values, we have

$$\mathbb{P}[PPSZ : \beta \in B_{\beta^*}] \geq 2^{-\beta_H|D|-(1-\gamma_H)(n_G-|D|)-\epsilon' n_G-o(n_G)},$$

where ϵ' is some arbitrary small positive real. This finishes the proof of Lemma 5.3.

5.2.7 Optimized Nice Distributions for 3-SAT

By Proposition 5.4, the running time bound depends on the choice of some H which produces a large δ. Experiments showed that we should consider functions H where there is some $r_0 \leq 1/2$ with $H(r)^2 \geq R_3(r)$ for $r \leq r_0$ and $H(r)^2 \leq R_3(r)$ for $r \geq r_0$. In this case, we have:

$$\gamma_H = \int_0^{r_0} R_3(r)\,dr + \int_{r_0}^1 H(r)^2\,dr$$

For $r \in [0, 1/2]$, we have:

$$R_3(r) = \frac{r^2}{(1-r)^2}$$

$$\int_0^r R_3(r')\,dr' = 2\ln(1-r) - 1 + \frac{r^2-r-1}{r-1}$$

As a simple example, we consider the function $H_\theta(r) = \min\{1, r/\theta\}$ for some $\theta \in [1/2, 1]$. Firstly, for $r \in [0, 1-\theta]$, we have $H_\theta(r)^2 \geq R_3(r)$. Secondly, for $r \in [1-\theta, \theta]$, we have $H_\theta(r)^2 \leq R_3(r)$, and finally, for $r \in [\theta, 1]$, we have $H_\theta(r)^2 = R_3(r) = 1$. Hence the following holds:

$$\begin{aligned}
\gamma_{H_\theta} &= \int_0^{1-\theta} R_3(r)\,dr + \int_{1-\theta}^{\theta} H_\theta(r)^2\,dr + 1 - \theta \\
&= \frac{6\,\ln(\theta)\theta^2 + 6\,\theta - 4\,\theta^3 - 1}{3\,\theta^2} \\
\beta_{H_\theta} &= \int_0^{\theta} \frac{1}{\theta} \log_2\left(\frac{1}{\theta}\right)\,dr \\
&= -\log_2(\theta)
\end{aligned}$$

We insert this into the formula for δ in Proposition 5.4 and compute the root of the derivate with respect to θ to get the optimal $\theta = 0.5109968782$. For this θ, we get $\beta_{H_\theta} \leq 0.9686136176$, $\gamma_{H_\theta} \geq 0.613242472$, and thus $\delta \geq 0.028368$. This yields an upper bound of $\mathcal{O}\left(1.3225^{n_G}\right)$ for the expected number of repetitions of Algorithm $COMB$.

But, we can do better. In order to find an optimal H, we can set up a continuous function H consisting of linear pieces and try to optimize it until we hit the best result. Experiments showed that the resulting curve is perfectly resembled by the following function, with some appropriate parameters a and b:

$$H(r) = \begin{cases} r/\theta & \text{if } r \in [0, 1-\theta) \\ 1 - (-a\,\ln(r))^b & \text{if } r \in [1-\theta, 1] \end{cases}$$

$$h(r) = \frac{dH}{dr} = \begin{cases} 1/\theta & \text{if } r \in [0, 1-\theta) \\ -b\,\frac{(-a\,\ln(r))^b}{r\,\ln(r)} & \text{if } r \in [1-\theta, 1) \end{cases}$$

$H(r)$ must be continuous, and naturally, it should also be differentiable completely. Moreover, we propose that $H(r)$ should hit $R_3(r)^{1/2}$ exactly when the linear part finishes, i.e. at $1-\theta$ since $R_3(r)^{1/2} = r/(1-r)$ for $r \in [0, 1/2]$. Using these constraints, i.e.

$$\begin{aligned}
H(1-\theta) &= R_3(1-\theta)^{1/2} \quad \text{and} \\
h(1-\theta) &= 1/\theta,
\end{aligned}$$

we can eliminate a and b:

$$a = -\left(\frac{2\,\theta - 1}{\theta}\right)^{\frac{2\,\theta-1}{\ln(1-\theta)(\theta-1)}} (\ln(1-\theta))^{-1}$$
$$b = \frac{\ln(1-\theta)\,(\theta-1)}{2\,\theta - 1}$$

For the antiderivative of $h(r)\log h(r)$, we have

$$\beta_1(r) = -\frac{r\log\theta}{\theta} + C$$

for $r \in [0, 1-\theta)$ and

$$\beta_2(r) = \frac{-(-a\ln r)^b\left(b\ln r - b^2 + 1 + (b+b^2)\ln\left(-\frac{(-a\ln r)^b b}{r\ln r}\right)\right)}{(b+b^2)\ln 2} + C$$

for $r \in [1-\theta, 1)$. Observe that $\beta_2(r)$ is not defined for $r = 1$. However, when r approaches 1^-, then $\beta_2(r)$ tends to C. We have:

$$\begin{aligned}\beta_H &= \int_0^1 h(r)\log h(r)\,dr \\ &= \beta_1(1-\theta) - \beta_1(0) + \lim_{r\to 1^-}\beta_2(r) - \beta_2(1-\theta) \\ &= \beta_1(1-\theta) - \beta_1(0) - \beta_2(1-\theta)\end{aligned}$$

For the antiderivative of $H(r)^2$, we have

$$\gamma_2(r) = r - 2\Gamma(1+b, -\ln r)\cdot a^b + \Gamma(1+2b, -\ln r)\cdot a^{2b}$$

for $r \in [1-\theta, 1]$ where $\Gamma(a,x)$ is the (upper) incomplete gamma function. For $r \in [0, 1-\theta)$, we need the antiderivative of $R_3(r)$, which is

$$\gamma_1(r) = 2\ln(1-r) + \frac{r^2 - r - 1}{r-1} + C.$$

Since $H(r)^2 \geq R_3(r)$ for $r \in [0, 1-\theta]$ and $H(r)^2 \leq R_3(r)$ for $r \in [\theta, 1]$, we conclude:

$$\begin{aligned}\gamma_H &= \int_0^{1-\theta} R_3(r)\,dr + \int_{1-\theta}^1 H(r)^2\,dr \\ &= \gamma_1(1-\theta) - \gamma_1(0) + \gamma_2(1) - \gamma_2(1-\theta)\end{aligned}$$

To find θ so that δ in Proposition 5.4 is maximized, we just insert the terms for γ_H and β_H in the formula for δ and find the optimum with respect to θ. Numerical optimization yields that δ is maximized using:

$$\begin{aligned}
\theta &= 0.5111885981... \\
a &= 1.1437170697... \\
b &= 15.635592073... \\
\beta_H &\leq 0.9062404894 \\
\gamma_H &\geq 0.6122939734 \\
\delta_H &\geq 0.0292762355 \\
2^{\sigma_3(1-\delta_H)} &\leq 1.3221508262
\end{aligned}$$

This yields an upper bound of $\mathcal{O}\left(1.32216^{n_G}\right)$ for the expected number of repetitions of Algorithm $COMB$. The graphs in Figure 5.4 and Figure 5.5 show $H(r)^2$ and $R_3(r)$ resp. $H(r)$ and $R_3(r)^{1/2}$.

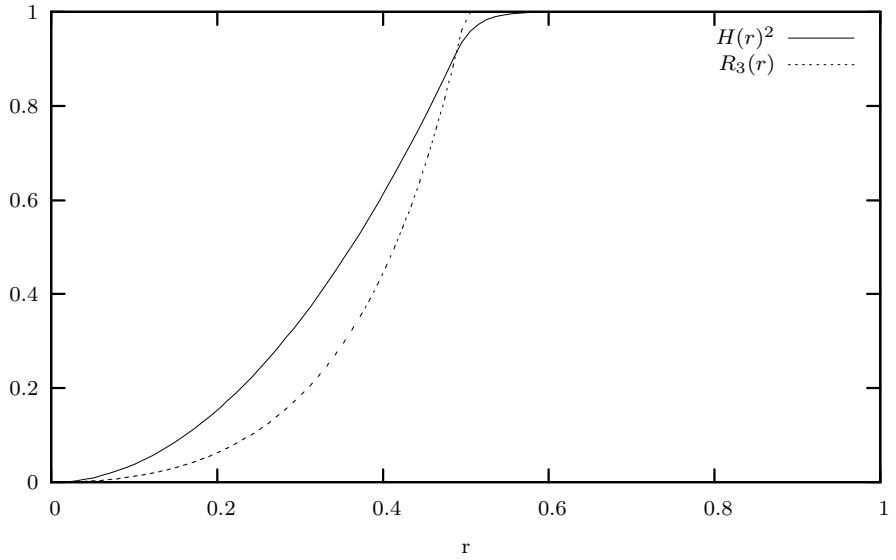

Figure 5.4: $H(r)^2$ and $R_3(r)$

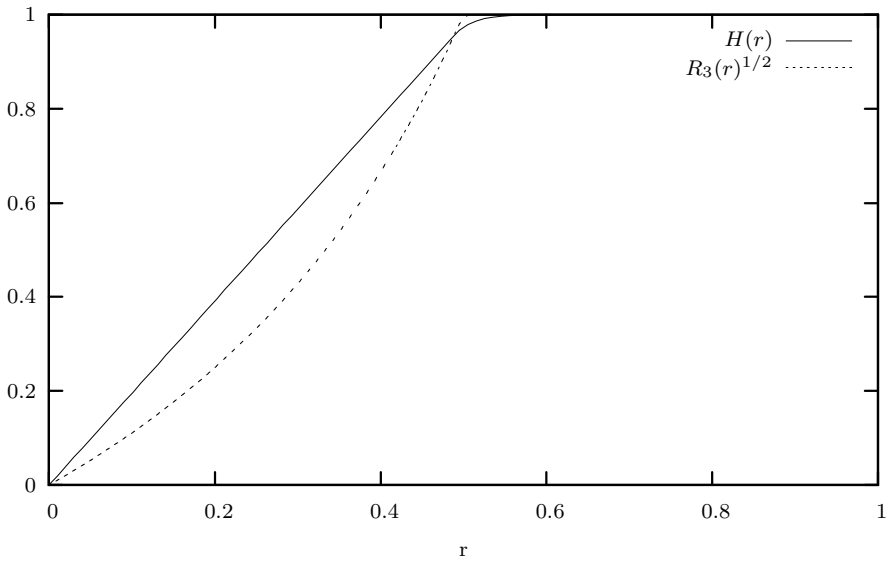

Figure 5.5: $H(r)$ and $R_3(r)^{1/2}$

Bibliography

[1] Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 1992.

[2] Sven Baumer and Rainer Schuler. Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 150–161, 2003.

[3] Tobias Brueggemann and Walter Kern. An improved deterministic local search algorithm for 3-SAT. *Theoretical Computer Science (TCS)*, 329:303–313, 2004.

[4] Chris Calabro, Russell Impagliazzo, Valentine Kabanets, and Ramamohan Paturi. The complexity of unique k-SAT: An isolation lemma for k-CNFs. In *Proceedings of the 18th Annual IEEE Conference on Computational Complexity (CCC)*, pages 135–141, 2003.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158, 1971.

[6] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science (TCS)*, 289:69–83, 2002.

[7] Russell Impagliazzo and Ramamohan Paturi. Complexity of k-SAT. In *Proceedings of the 14th Annual IEEE Conference on Computational Complexity (CCC)*, pages 237–240, 1999.

[8] Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 328–328, 2004.

[9] Oliver Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science (TCS)*, 223:1–72, 1999.

[10] Burkhard Monien and Ewald Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[11] Christos H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 163–169, 1991.

[12] Ramamohan Paturi, Pavel Pudlak, and Francis Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 566–574, 1997.

[13] Ramamohan Paturi, Pavel Pudlak, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 628–637, 1998.

[14] Ramamohan Paturi, Pavel Pudlak, and Francis Zane. Satisfiability coding lemma. *Chicago Journal of Theoretical Computer Science*, 1999.

[15] Ramamohan Paturi, Pavel Pudlak, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-SAT. *Journal of the Association for Computing Machinery (JACM)*, to appear.

[16] Pavel Pudlak. Satisfiability – algorithms and logic. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 129–141, 1998.

[17] Daniel Rolf. 3-SAT $\in RTIME(1.32971^n)$. Diploma thesis, Department Of Computer Science, Humboldt University Berlin, Germany, 2003.

[18] Daniel Rolf. 3-SAT $\in RTIME(O(1.32793^n))$ - improving randomized local search by initializing strings of 3-clauses. *Electronic Colloquium on Computational Complexity (ECCC)*, (54), 2003.

[19] Daniel Rolf. Derandomization of PPSZ for Unique-k-SAT. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 216–225, 2005.

[20] Daniel Rolf. Improved bound for the PPSZ/Schöning-algorithm for 3-SAT. *Electronic Colloquium on Computational Complexity (ECCC)*, (159), 2005.

[21] Ingo Schiermeyer. Solving 3-satisfiability in less than 1.579^n steps. In *Selected Papers from the 6th Workshop on Computer Science Logic (CSL)*, pages 379–394, 1993.

[22] Ingo Schiermeyer. Pure literal look ahead: an $\mathcal{O}(1.497^n)$ 3-satisfiability algorithm. In *Workshop on Satisfiability*, pages 63–72, 1996.

[23] Rainer Schuler, Uwe Schöning, and Osamu Watanabe. A probabilistic 3-SAT algorithm further improved. In *Proceedings of the 19th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 192–202, 2002.

[24] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 410–414, 1999.

[25] Uwe Schöning. On the complexity of constraint satisfaction problems. *Ulmer Informatik Berichte Nr. 99-03, Universität Ulm*, 1999.

[26] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer Academic Publishers, 1996.

Selbständigkeitserklärung

Hiermit erkläre ich, dass

- ich die vorliegende Dissertationsschrift selbständig und ohne unerlaubte Hilfe verfasst habe;
- ich mich nicht bereits anderwärtig um einen Doktorgrad beworben habe oder einen solchen besitze;
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Daniel Rolf Berlin, den 30. Mai 2006