UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR THEORETISCHE INFORMATIK

Lokale Suchstrategien für Erfüllbarkeitsprobleme

Local search strategies for satisfiability problems

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Tobias Schomann

ausgegeben und betreut von
Prof. Dr. math. Rüdiger Reischuk

Lübeck, den 13. Dezember 2012

Kurzzusammenfassung

Lokale Suchstrategien für Erfüllbarkeitsprobleme

Schöning [6] lieferte einen randomisierten Algorithmus, der das Problem k-SAT in $O(\text{poly}(n) \cdot (2 - \frac{2}{k})^n)$ löst, wobei $\text{poly}(n)$ ein Polynom und n die Anzahl der Variablen einer aussagenlogischen Formel ist. Dantsin u.a. [2] lieferte einen deterministischen Algorithmus mit Zeitkomplexität in $O(\text{poly}(n) \cdot (2 - \frac{2}{k+1})^n)$, der als teilsweise Derandomisierung von Schönings Algorithmus betrachtet werden kann. Eine vollständige Derandomisierung kommt von Moser und Scheder [4]. Beide deterministischen Algorithmen verwenden eine lokale Suchstrategie, die auf Covering Codes aufbauen. In dieser Arbeit werden neben einer theoretischen Analyse der genannten deterministischen Algorithmen auch Methoden zum Konstruieren von Covering Codes vorgestellt. Weiterhin wurden alle drei genannten Algorithmen implementiert und deren Zeitkomplexität experimentell bestimmt. Die Ergebnisse der Experimente werden untereinander verglichen.

Abstract

Local search strategies for satisfiability problems

Schöning [6] gave a randomized algorithm that solves the problem k-SAT in time $O(\text{poly}(n) \cdot (2 - \frac{2}{k})^n)$, where $\text{poly}(n)$ is a polynomial and n the number of variables in a propositional formula. Dantsin et al. [2] gave an deterministic algorithm with running time $O(\text{poly}(n) \cdot (2 - \frac{2}{k+1})^n)$, which can be viewed as an partially derandomisation of Schönings algorithm. A complete derandomisation comes from Moser and Scheder [4]. Both deterministic algorithms use a local search strategy, based on covering codes. In this work, a theoretical analysis of said deterministic algorithms and methods to construct covering codes are presented. Furthermore, all three algorithms have been implemented and the running time has been determined experimentally. The results of the experiments are compared with each other.

Inhaltsverzeichnis

1 Einleitung **5**
1.1 Aufgabenstellung 6
1.2 Aufbau der Arbeit 6

2 Covering Codes **7**
2.1 Einführung und Notation 7
2.2 Schranken von Covering Codes 8
2.3 Konstruktion durch Greedy-Algorithmus 8
2.4 Konstruktion durch direkte Summe 10
2.5 Ergebnisse 11

3 k-SAT Algorithmen **14**
3.1 Begriffe und Notation 14
3.2 Formeln reduzieren 15
3.3 Lokale Suche von Dantsin u.a. 16
3.4 Lokale Suche von Moser und Scheder 17
3.5 k-SAT Algorithmus von Moser und Scheder 20
3.6 k-SAT Algorithmus von Dantsin 22
3.7 k-SAT Algorithmus von Schöning 22

4 Experimente **24**
4.1 Vorgehensweise 24
4.2 Formeln erzeugen 24
4.3 Testumgebung 25
4.4 Lokale Suche bei Formeln mit genau einer erfüllenden Belegung 26
4.5 Laufzeitanalyse von Schönings k-SAT Algorithmus bei Formeln mit genau einer erfüllenden Belegung 29
4.6 Laufzeitanalyse der k-SAT Algorithmen bei zufälligen Formeln 31
4.7 Zusammenfassung der Ergebnisse 39

5 Zusammenfassung **40**

A Quelltexte **41**
A.1 Bibliothek: code.py 41
A.2 Bibliothek: formula.py 44
A.3 Bibliothek: localsearch.py 46

A.4 Bibliothek: sat.py 47

B Formel in 3-KNF **49**

C Literaturverzeichnis **50**

Kapitel 1

Einleitung

Das Erfüllbarkeitsproblem der Aussagenlogik (kurz: SAT) ist ein Entscheidungsproblem, das die Frage stellt, ob für eine aussagenlogische Formel eine Belegung der Variablen mit den Wahrheitswerten `wahr` und `falsch` (oder auch 1 und 0) existiert, so dass die Formel erfüllt ist. In der Komplexitätstheorie ist das Erfüllbarkeitsproblem nicht nur sehr bekannt, es ist auch von größter Bedeutung. Der 1971 durch Cook (und unabhängig davon auch Levin) erstmalig erbrachte Nachweis der NP-Vollständigkeit von SAT war Ausgangspunkt für viele weitere NP-Vollständigkeitsnachweise. Die meisten Beweise dieser Art machen sich zunutze, dass sich das entsprechende Problem in Polynomialzeit auf SAT reduzieren lässt. Eine Effizienzverbesserung der SAT-Algorithmen kann daher auch für andere NP-vollständige Probleme von Bedeutung sein.

Eine wesentliche Eigenschaft NP-vollständiger Probleme ist, dass sie sich *vermutlich*[1] nicht effizient lösen lassen, also dass die Laufzeit in jedem Fall exponentiell ist. Auch eine deutliche Verbesserung der Computergeschwindigkeit hätte kaum Auswirkung auf die effektive Geschwindigkeit, mit der sich solche Probleme lösen ließen. Betrachtet man zum Beispiel den trivialen Algorithmus, der SAT löst. Für eine Formel über n Variablen testet er alle 2^n möglichen Belegungen der Variablen aus und kann somit SAT mit einer Laufzeit in $O(p \cdot 2^n)$ entscheiden, wobei p ein polynomieller Faktor ist. Würden nun Computer um den Faktor 1000 schneller, hätte das zur Folge das Formeln mit gerade mal $\log_2 1000 \approx 10$ Variablen mehr in der gleichen Zeit entschieden werden könnten. Wenn es allerdings gelingt, den Algorithmus so zu verbessern, dass die Laufzeit zum Beispiel in $O(2^{n/2})$ liegt, dann ließen sich bei gleicher Computergeschwindigkeit und in gleicher Zeit Formeln mit doppelt so vielen Variablen entscheiden. Das Finden von effizienteren Algorithmen ist daher ein viel beachtetes Thema, dem sich viele Informatiker zuwenden.

Eine bekannte Strategie, die den trivialen Algorithmus hinsichtlich der Zeitkomplexität schlägt, ist die lokale Suche nach erfüllenden Belegungen für eine Formel. Einen Algorithmus, der diese Strategie verfolgt, hat Schöning 1999 umfassend analysiert[6]. Es ist ein randomisierter WalkSAT Algorithmus, der nach folgendem Grundprinzip funktioniert: Für die Variablen einer aussagenlogischen Formel in konjunktiver Normalform wird eine Belegung zufällig gewählt. Wenn die Belegung die Formel erfüllt, wird der Algorithmus beendet, sonst wird durch gezielte Negierung der Belegung einzelner Variablen, die aus nicht erfüllten Klauseln gewählt werden, versucht, sich einer erfüllenden Belegung anzunähern. Da es aber passieren kann, dass der Algorithmus in einem lokalen Minimum stecken bleibt, wird nach einer bestimmten Anzahl von Schritten mit einer

[1] Um behaupten zu können, das NP-vollständige Probleme nicht in Polynomialzeit lösbar sind, ist $\mathsf{P} \neq \mathsf{NP}$ notwendig, wobei P die Klasse der in Polynomialzeit lösbaren Probleme ist. Der Beweis dafür steht allerdings (noch) aus.

neuen, ebenfalls zufällig gewählten, Belegung das Vorgehen wiederholt. Für eine Formel über n Variablen und einer Klauselgröße von maximal k hat der Algorithmus eine erwartete Laufzeit von $O\left(p \cdot (2-2/k)^n\right)$, wobei p ein polynomieller Faktor ist. Für $k = 3$ ist die Laufzeit folglich $O\left(p \cdot 1{,}34^n\right)$.
Als ein Versuch der Derandomisierung von Schönings Algorithmus kann der deterministische Algorithmus, der 2002 von Dantsin u.a. geliefert[2] wurde, gesehen werden. Man spricht hier von einem Versuch, weil die Laufzeit in $O(1{,}5^n)$ bei $k = 3$ liegt und somit langsamer ist als Schönings. Darauf aufbauend haben Moser und Scheder 2009 einen deterministischen Algorithmus geliefert, der die gleiche Laufzeitkomplexität erreicht wie Schönings und daher als eine vollständige Derandomisierung verstanden werden kann. Der wesentlicher Ansatz bei der deterministischen lokalen Suche ist, dass statt der zufällig erzeugten Belegungen, die aus der Codierungstheorie bereits bekannten Covering Codes zum deterministischen Wählen von Startbelegungen genutzt werden.

1.1 Aufgabenstellung

In dieser Arbeit wird, vor dem Hintergrund der Algorithmen von Schöning und Dantsin u.a., eine theoretische Untersuchung des k-SAT Algorithmus von Moser und Scheder in Bezug auf Zeitkomplexität und Korrektheit vorgestellt. Weiterhin werden die k-SAT Algorithmen von Schöning, Dantsin u.a. und Moser/Scheder implementiert, experimentell untersucht und die Ergebnisse werden diskutiert.

1.2 Aufbau der Arbeit

Die beiden deterministischen k-SAT Algorithmen, die in dieser Arbeit untersucht werden, verwenden die aus der Codierungstheorie bekannten Covering Codes. Das nachfolgende Kapitel 2 führt einige Begriffe aus der Codierungstheorie ein und thematisiert weiterhin untere Schranken und die Konstruktion von Covering Codes. In Kapitel 3 wird das Erfüllbarkeitsproblem und die lokale Suche für Erfüllbarkeitsprobleme eingeführt. Entsprechende Algorithmen werden theoretisch untersucht bzw. vorgestellt. Im vierten Kapitel werden die Experimente, die mit den implementierten Algorithmen durchgeführt wurden, beschrieben, und die Resultate werden vorgestellt und gedeutet. Im letzten Kapitel werden die Ergebnisse noch einmal zusammengefasst und diskutiert. Die Quelltexte der implementierten Algorithmen sind im Anhang zu finden.

Kapitel 2

Covering Codes

Der deterministische k-SAT Algorithmus von Dantsin kann die Startbelegungen für die lokale Suche nicht zufällig wählen, sondern benötigt eine ebenfalls deterministische Methode um diese Belegungen zu finden. Dabei ist einerseits zu beachten, das die Startbelegungen nicht zu „ähnlich“ sind, damit sich die Suchbereiche nicht überschneiden. Denn das hätte zur Folge, dass einige Belegungen mehrfach geprüft werden. Andererseits dürfen sich die Belegungen aber auch nicht zu *unähnlich* sein, da sonst Lücken zwischen Bereichen, die von der lokalen Suche erreicht werden, entstehen und der Algorithmus eventuell nicht korrekt entscheidet. Die aus der Codierungstheorie bekannten Covering Codes können genutzt werden, um Startbelegungen mit den beschriebenen Eigenschaften zu finden.

2.1 Einführung und Notation

Ein Wort $w = (w_1, ..., w_n)$ der Länge n bezeichnet ein n-Tupel über einem Alphabet A, wobei ein Alphabet eine endliche Menge von Symbolen ist. Ein Code $\mathcal{C}$ der Länge n ist eine endliche Menge von Worten der Länge n. Wenn ein Wort Element eines Codes ist, wird es auch als Codewort bezeichnet. Die Menge aller Worte der Länge n über einem Alphabet A ist ein Raum und wird durch $H_A(n)$ bezeichnet. Die Hamming-Distanz zweier Worte $u = (u_1, ..., u_n)$ und $v = (v_1, ..., v_n)$ ist $d(u, v) := \sum_{u_i \neq v_i} 1$. Eine Kugel ist eine Teilmenge eines Raumes und enthält alle Worte, die eine durch einen Radius bestimmte Hamming-Distanz zu einem Mittelpunkt-Wort nicht überschreiten. Sei A ein Alphabet mit $|A| = q$, $r \in \mathbb{N}$ und $w_M \in H_A(n)$. Dann ist eine Kugel definiert durch

$$K_A(w_M, r) := \{w \in H_A(n) \mid d(w, w_M) \leq r\} .$$

Die Anzahl der Wörter, die in einer Kugel enthalten sind, wird als Volumen der Kugel bezeichnet und ist unabhängig vom Mittelpunkt. Das Kugelvolumen wird bestimmt durch

$$V_q(n, r) := \sum_{i=0}^{r} \binom{n}{i} (q-1)^i .$$

Im binären Fall, also $q = 2$, können folgende Aussagen getroffen werden, die aus [2] entnommen sind. Die Funktion $h(x) = -x \cdot \log_2 x - (1-x) \cdot \log_2(1-x)$ ist dabei die binäre Entropie. Sei $1 \leq r \leq n/2$, dann beschränkt

$$\frac{2^{h(r/n)}}{n} \leq \frac{2^{h(r/n)}}{\sqrt{8r(1-r/n)}} \leq V_2(n, r) \leq 2^{h(r/n)n} \tag{2.1}$$

das Kugelvolumen. Ein Code $\mathcal{C}$ ist ein Covering Code mit Radius r für einen Raum $H_A(n)$ genau dann, wenn

$$H_A(n) = \bigcup_{c \in \mathcal{C}} K_A(c, r) \; .$$

2.2 Schranken von Covering Codes

Sei ein Alphabet A mit $|A| = q$ und $n, r \in \mathbb{N}$. Eine bekannte untere Schranke für die Mächtigkeit eines Covering Code $\mathcal{C}$ der Länge n und mit Radius r ist die Kugelüberdeckungsschranke, die aus dem Kugelvolumen folgt. Weil offensichtlich $|\mathcal{C}| \cdot V_A(n, r) \geq q^n$ ist auch

$$\frac{q^n}{V_q(n, r)} \leq |\mathcal{C}|$$

Nicht jeder Covering Code erreicht diese Schranke, auch wenn seine Mächtigkeit minimal ist. Oft ist es unvermeidbar, dass die Kugeln eines Covering Codes sich überschneiden, also dass es in einem Raum Worte gibt, die in mehreren Kugeln eines Covering Codes liegen. Moser und Scheder beweisen in ihrer Arbeit [4, Lemma 9], dass für einen Raum $H_A(n)$ ein Covering Code mit Radius r existiert, der

$$|\mathcal{C}| \leq \left\lceil \frac{n \, \ln(q) \, q^n}{\binom{n}{r}(q-1)^r} \right\rceil \tag{2.2}$$

erfüllt. In dem Beweis wurde probabilistisch argumentiert. Wenn eine durch (2.2) berechnete Anzahl von Codewörten zufällig gewählt wird, dann ist die Wahrscheinlichkeit p, dass diese Codewörter einen Covering Code ergeben $p > 0$ und somit ist die Existenz erwiesen. Weiterhin gibt es obere Schranken die unter anderem aus [2, Lemma 4] hervorgehen und nur im binären Fall gelten.

$$|\mathcal{C}| \overset{[2]}{\leq} \left\lceil \frac{n \, 2^n}{V_2(n, r)} \right\rceil \overset{(2.1)}{\leq} \left\lceil \frac{n^2 \, 2^n}{2^{h(r/n)n}} \right\rceil = n^2 \, 2^{(1-h(r/n))n} \; . \tag{2.3}$$

Die Wahrscheinlichkeit, dass ein zufällig gewählter Code der in (2.3) angegebenen Mächtigkeit ein Covering Code ist, konvergiert im binären Fall mit wachsendem n gegen 1. Eine Möglichkeit binäre Covering Codes zu konstruieren, ist folglich, die Codeworte zu raten.

2.3 Konstruktion durch Greedy-Algorithmus

Zunächst betrachten wir das als Optimierungsproblem formulierte Mengenüberdeckungsproblem oder auch SET-COVER-Problem. Für eine Menge U und eine Familie S von Teilmengen von U wird eine Teilfamilie $C \subseteq S$ von Mengen gesucht, dessen Vereinigung gleich U ist. Dabei soll $|C|$ einen möglichst kleinen Wert annehmen.
Für dieses Problem ist bereits ein deterministischer Greedy Algorithmus bekannt. Er löst das SET-COVER-Problem nach der folgenden Vorgehensweise: In jedem Schritt wähle eine Teilmenge, die eine maximale Anzahl von bisher nicht überdeckten Elementen überdeckt. Wiederhole den Vorgang bis alle Elemente überdeckt sind. Die Anzahl so ausgewählter Teilmengen ist maximal um den Faktor $G(s)$ größer als die minimale Anzahl, wie bereits an anderer Stelle gezeigt wurde[1, Seite 233]. Wobei s die Mächtigkeit der zu überdeckenden Menge U ist, und

$$G(s) = \sum_{k=1}^{s} \frac{1}{k} \leq \ln s + 1$$

die harmonische Reihe. Das Problem, einen Covering Code mit möglichst geringer Mächtigkeit zu finden, kann in ein SET-COVER-Problem überführt werden. Sei A mit $|A| = q$ ein Alphabet und $n, r \in \mathbb{N}$. Es gilt die Menge $H_A(n)$ durch l Kugeln $K_A(w_i, r) \subseteq H_A(n)$ mit $w_1, ..., w_l \in H_A(n)$ zu überdecken. Zusammengefasst bedeutet das, ein Covering Code der den Raum $H_A(n)$ überdeckt, kann von einem Greedy Algorithmus so konstruiert werden, dass seine Mächtigkeit maximal um den Faktor

$$G(H_A(n)) = \ln |H_A(n)| + 1 = \ln q^n + 1 = n \ln q + 1 \in O(n) \tag{2.4}$$

größer ist, als die minimale Mächtigkeit, sofern q konstant ist.
Nachfolgend wird ein deterministischer Greedy-Algorithmus zum Konstruieren eines Covering Code über einem Alphabet A, der Länge n und mit Radius r beschrieben. Es wird mit der Menge $\mathcal{C} := \emptyset$ und der Liste $L := H_A(n)$ gestartet. Jedem Wort $w \in L$ wird eine Menge $D(w)$ zugeordnet, die zunächst mit $D(w) = \emptyset$ initialisiert wird. Diese Mengen sollen später die Worte enthalten, die in der Kugel des entsprechenden Wortes liegen, aber bereits durch ein bereits ausgewähltes Codewort überdeckt werden. Für jedes Wort $w \in H_A(n)$ kann die Anzahl der noch nicht überdeckten Worte in seiner Kugel durch das unabhängige Kugelvolumen $V_A(n, r)$, abzüglich der Größe $|D(w)|$ der zugehörigen Menge, berechnet werden. In jeder nun folgenden Iteration wird die Kugel ermittelt, die die meisten unüberdeckten Worte enthält und der Mittelpunkt wird in $\mathcal{C}$ eingefügt. Weiterhin werden alle Worte der Kugel aus der Liste L entfernt, und für jedes Wort $w \in L$ die zugeordnete Menge $D(w)$ aktualisiert, dadurch dass alle aus L entfernten Worte mit denen in L verbleibenden bezüglich der Hamming-Distanz verglichen werden. Sobald $L = \emptyset$ wird mit $\mathcal{C}$ beendet.

Algorithmus 1 `cov-greedy` (Alphabet A, Länge n, Radius r)

1. $\mathcal{C} \leftarrow \emptyset$; $L \leftarrow H_A(n)$
2. Für jedes $w \in L$ sei $D(w) = \emptyset$
3. **while** $L \neq \emptyset$ **do**
4. $\quad$ $c \leftarrow u$ mit $V_A(n, r) - |D(u)| = \max_{v \in R} (V_A(n, r) - |D(v)|)$
5. $\quad$ $L \leftarrow L \setminus K_A(c, r)$
6. $\quad$ **for all** $u \in L$ **do**
7. $\quad\quad$ **for all** $v \in K_A(c, r)$ **do**
8. $\quad\quad\quad$ **if** $d(u, v) \leq r$ **then** $D(u) \leftarrow D(u) \cup \{v\}$ **end if**
9. $\quad\quad$ **end for**
10. $\quad$ **end for**
11. $\quad$ $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$
12. **end while**
13. **return** $\mathcal{C}$

Der Platzbedarf des Algorithmus beträgt $\text{poly}(n) \cdot q^n \cdot V_A(n, r) \in O(\text{poly}(n) \cdot q^{2n})$ und die Zeitkomplexität kann wie folgt bestimmt werden: Es müssen $|\mathcal{C}|$ Iterationen durchgeführt werden, die jeweils maximal q^n Schritte benötigt um c zu ermitteln und maximal $q^n \cdot V_A(n, r)$ um die verbleibenden Kugeln zu aktualisieren. Alle übrigen Berechnungen lassen sich in polynomieller Zeit durchführen. Für den Fall $q > 2$ ergibt das

$$\text{poly}(n) \cdot |\mathcal{C}| \cdot (q^n + q^n V_q(n, r)) \in O(\text{poly}(n) \cdot q^{3n}) .$$

Im Fall $q = 2$ kann mit der oberen Schranke für den binären Fall aus der Ungleichung (2.3) ein besserer Wert erreicht werden.

$$\begin{aligned} & \text{poly}_1(n) \cdot |\mathcal{C}| \cdot (2^n + 2^n\, V_2(n,r)) \\ \le\ & \text{poly}_1(n) \cdot |\mathcal{C}| \cdot 2 \cdot 2^n\, V_2(n,r) \\ =\ & \text{poly}_1(n) \cdot \frac{n \cdot 2^n}{V_2(n,r)} \cdot 2 \cdot 2^n\, V_2(n,r) \\ =\ & \text{poly}_1(n) \cdot 2n\, 2^{2n} \\ =\ & \text{poly}_1(n) \cdot \text{poly}_2(n) \cdot 2^{2n} \\ =\ & \text{poly}(n) \cdot 2^{2n} \end{aligned}$$

2.4 Konstruktion durch direkte Summe

Wenn bereits ein oder mehrere Covering Codes mit *kleiner* Länge zur Verfügung stehen, ist es möglich diese zu einem Covering Code mit *großer* Länge zusammenzusetzen. Auf diese Weise kann die Laufzeit der Konstruktion, zum Beispiel gegenüber dem Greedy Algorithmus, verbessert werden. Der Nachteil ist allerdings, dass sich die Mächtigkeit des Covering Codes dabei weiter von dem optimalen Wert entfernt. Die nachfolgend verwendete Konkatenation von Mengen (direkte Summe) ist definiert durch $A \circ B = \{a \circ b \mid a \in A \wedge b \in B\}$, wobei A und B endliche Mengen sind.

Lemma 2.1. *Sei $m, n \in \mathbb{N}$. A ein Alphabet. $\mathcal{C}_1 \subseteq H_A(m)$ ein Covering Code mit Radius r_1 und $\mathcal{C}_2 \subseteq H_A(n)$ ein Covering Code mit Radius r_2. Dann ist $\mathcal{C}_1 \circ \mathcal{C}_2 \subseteq H_A(m+n)$ ein Covering Code mit Radius $r_1 + r_2$.*

Beweis. Sei $w_1 \in H_A(m)$ und $w_2 \in H_A(n)$. Dann existiert $c_1 \in \mathcal{C}_1$ und $c_2 \in \mathcal{C}_2$ mit $d(w_1, c_1) \le r_1$ und $d(w_2, c_2) \le r_2$. Folglich ist $d(w_1 \circ w_2, c_1 \circ c_2) \le r_1 + r_2$. Da $c_1 \circ c_2 \in \mathcal{C}_1 \circ \mathcal{C}_2$, ist $\mathcal{C}_1 \circ \mathcal{C}_2$ ein Covering Code mit Radius $r_1 + r_2$. □

Sei ein Alphabet A mit $|A| = 2$ und ein Covering Code $\mathcal{D}$ mit Länge $\hat{n}$, Radius $\hat{r}$ und minimaler Mächtigkeit. Mit der d-fachen direkten Summe, wobei $d \ge 2$, wird daraus ein Covering Code $\mathcal{C} = \mathcal{D}^d$ der Länge $n = \hat{n} \cdot d$ mit Radius $r = \hat{r} \cdot d$ konstruiert.

$$\begin{aligned} |\mathcal{C}| = |\mathcal{D}|^d &= \left(\hat{n}^2\, 2^{(1-h(\hat{r}/\hat{n}))\hat{n}}\right)^d \\ &= \left((n/d)^2\, 2^{(1-h(r/n))n/d}\right)^d \\ &= (n/d)^{2d}\, 2^{(1-h(r/n))n} \ . \end{aligned}$$

Die Mächtigkeit von $\mathcal{C}$ ist um einen polynomiellen Faktor vom Grad $g = g(d)$ größer als der entsprechende Covering Code mit minimaler Mächtigkeit. Wenn wir nun weiterhin annehmen, dass $\mathcal{D}$ konstant ist, dann wächst $d = d(n)$ folglich, wenn n wächst. In diesem Fall ist $|\mathcal{C}|$ um einen exponentiellen Faktor größer als der minimale Wert.
Eine weitere Möglichkeit ist, dass man den Greedy Algorithmus mit der d-fachen direkten Summe kombiniert. Wobei d jetzt konstant ist. Wir lassen den Greedy Algorithmus einen Covering Code $\mathcal{G}$ der Länge n/d mit Radius r/d konstruieren. Mit $\mathcal{G}^d$ erhalten wir nun einen Covering Code der Länge n mit Radius r, dessen Mächtigkeit um einen polynomiellen Faktor von konstanten Grad

$g = g(d)$ größer ist als das entsprechende Minimum. Unter der Berücksichtigung der Laufzeit des Greedy Algorithmus und der Zeit die notwendig ist um die direkte Summe zu bilden, erhalten wir für diese Vorgehensweise eine Gesamtzeitkomplexität von $O(\text{poly}(n) \cdot 2^{2n/d} + 2^{(1-h(r/n))n})$. Wenn zum Beispiel $d = 5$ gewählt wird, kann ein binärer Covering Code der Länge n mit Radius r in der Zeit $O(\text{poly}(n) \cdot 1{,}32^n + 2^{(1-h(r/n))n})$ konstruiert werden.

2.5 Ergebnisse

In diesem Kapitel wurden Algorithmen zur Konstruktion von Covering Codes vorgestellt und analysiert. Die bei der Implementation erzielten Ergebnisse sollen hier vorgestellt werden. Die Mächtigkeiten der Covering Codes, die mit dem Greedy-Algorithmus konstruiert werden konnten, sind in Tabelle 2.1, Tabelle 2.2 und Tabelle 2.3 angegeben. Um vergleichen zu können, wurde für jedes Paar n, r eine untere Schranke aus [3] und die obere Schranke angegeben, die aus dem Existenzbeweis resultiert, der in diesem Kapitel vorgestellt wurde. Weiterhin sind die Covering Codes mit Länge n und Radius $r = n/4$, die bei den deterministische k-SAT Algorithmen noch zum Einsatz kommen werden, in einem Diagramm dargestellt, was in Abbildung 2.1 zu sehen ist. Die Codes mit Länge $n \leq 17$ wurden mit dem Greedy Algorithmus konstruiert, die anderen mittels direkter Summe aus den vorhandenen Covering Codes. Es ist zu sehen, dass die durch die Direkte Summe konstruierten Covering Codes die durch den Existenzbeweis gegebene Schranke überschreiten können.
Bei der Implementation der Greedy-Algorithmen wurden die Elemente des Hamming-Raums nach Zahlenwert sortiert. Alternativ wurde eine lexikographische Sortierung, die Sortierung nach Hamming-Gewicht (Hamming-Distanz zu dem 0-Wort) und eine Sortierung, die Worte mit einem Hamming-Gewicht von $2r+1$ bevorzugt getestet. Keine dieser Alternativen führten zu „besseren“ Covering Codes.

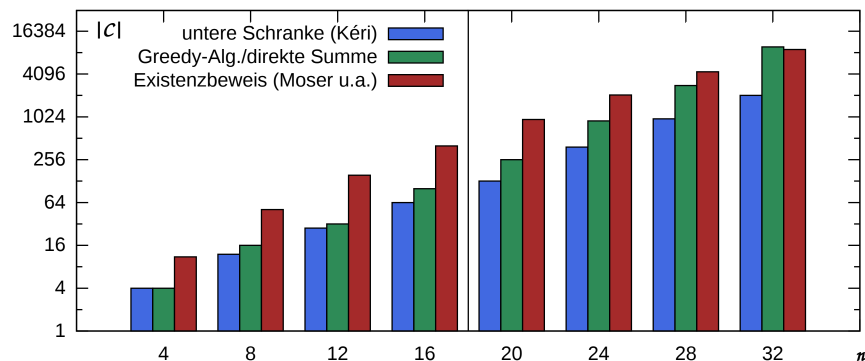

Abbildung 2.1: Mächtigkeiten der Covering Codes mit Länge n und Radius $r = n/4$, die von dem 3-SAT Algorithmus von Dantsin u.a. verwendet werden. Codes mit $n \leq 16$ wurden durch den Greedy-Algorithmus konstruiert, die Übrigen durch direkte Summe.

n	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$	$r = 6$	$r = 7$	$r = 8$
3	2 2 6							
4	4 4 13							
5	7 8 27	2 2 10						
6	12 16 55	4 4 18						
7	16 16 112	7 8 31	2 2 14					
8	32 32 228	12 16 56	4 4 23					
9	57-62 72 461	16 32 101	7 8 36	2 2 18				
10	105-120 128 931	24-30 32 183	9-12 16 59	4 4 27				
11	180-192 256 1878	37-44 80 337	12-16 16 95	7 8 47	2 2 34			
12	342-380 512 3781	62-78 160 623	18-28 32 165	8-12 16 62	4 4 31			
13	598-736 1024 7607	97-128 192 1158	28-42 64 282	11-16 16 98	7 8 45	2 2 26		
14	1172-1408 2048 15292	157-256 256 2164	44-64 116 489	16-28 48 156	8-12 16 67	4 4 36		
15	2048 2048 30720	310-384 512 4063	70-112 216 854	22-32 56 254	9-16 24 100	7 8 50	2 4 30	
16	4096 4096 61681	512-768 1024 7654	115-192 356 1505	33-64 100 417	13-28 38 153	8-12 16 71	4 4 40	
17	7414-8192 12166 123791	859-1536 2568 14469	187-320 512 2672	52-112 176 694	19-32 56 237	8-16 28 103	7 - 55	2 - 34

Tabelle 2.1: Mächtigkeiten für binäre Covering Codes mit unterschiedlicher Länge n und Radius r. Es sind jeweils drei Werte angegeben: untere Schranke, Konstruktion durch Greedy Algorithmus und Konstruktion durch zufällige Auswahl der Codeworte.

n	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$	$r = 6$	$r = 7$
2	3 3 5						
3	5 9 15	3 3 7					
4	9 9 44	3 3 15	3 3 11				
5	27 36 133	8 12 33	3 3 17	3 3 17			
6	71-73 99 400	15-17 23 80	6 9 30	3 3 20	3 3 25		
7	156-186 298 1201	26-34 53 200	11-12 19 60	3 3 30	3 3 25	3 3 38	
8	402-486 808 3604	54-81 139 515	14-27 40 129	9 14 51	3 3 32	3 3 32	3 3 56
9	1060-1269 - 10812	130-219 - 1352	27-54 91 290	11-18 - 97	6 - 49	3 - 37	3 - 43

Tabelle 2.2: Mächtigkeiten für Covering Codes mit Alphabetgröße $q = 3$, Länge n und Radius r. Es sind jeweils drei Werte angegeben: untere Schranke, Konstruktion durch Greedy Algorithmus und Konstruktion durch zufällige Auswahl der Codeworte.

n	$r = 1$	$r = 2$	$r = 3$	$r = 4$
2	4 4 7			
3	8 10 30	4 4 10		
4	24 64 118	7 8 26	4 4 13	
5	64 64 473	16 16 79	4 4 26	4 4 18

Tabelle 2.3: Mächtigkeiten für Covering Codes mit Alphabetgröße $q = 4$, Länge n und Radius r. Es sind jeweils drei Werte angegeben: untere Schranke, Konstruktion durch Greedy Algorithmus und Konstruktion durch zufällige Auswahl der Codeworte.

Kapitel 3

k-SAT Algorithmen

3.1 Begriffe und Notation

Das Erfüllbarkeitsproblem der Aussagenlogik ist ein Entscheidungsproblem. Die Frage ist, ob für die Variablen einer aussagenlogischen Formel eine Belegung existiert, die die Formel erfüllt.
Ein Literal u ist eine aussagenlogische Variable x oder ihre Negation $\overline{x}$. Eine endliche Menge $K = \{u_1, ..., u_l\}$ von Literalen ist eine Klausel, wobei $l \leq k$. Eine endliche Menge $F = \{K_1, ..., K_m\}$ von Klauseln bezeichnet eine Formel in k-KNF (konjunktive Normalform). Die Menge var(K) enthält die Variablen die den Literalen in einer Klausel K zugrunde liegen. Für eine Formel F ist var$(F) := \bigcup_{K \in F}$ var(K).
Eine Belegung β für eine Formel F ist eine Abbildung $\texttt{var}(F) \to \{0, 1\}$. Sie ordnet jeder in einer Formel vorkommenden Variable den Wahrheitswert `0` oder `1` zu. Ein nicht negiertes Literal besitzt den Wahrheitswert der zugrunde liegenden Variable. Ein negiertes Literal mit einer zugrunde liegenden Variable v und durch β belegt, hat den Wahrheitswert $1 - \beta(v)$. Eine Klausel ist durch β erfüllt, wenn sie mindestens ein Literal enthält, das den Wahrheitswert 1 besitzt. Eine Formel in k-KNF ist erfüllt, wenn alle darin enthaltenen Klauseln erfüllt sind. Die Hamming-Distanz zweier Belegungen β, γ über einer Variablenmenge V ist $d_V(\beta, \gamma) := |\,\{x \in V \mid \beta(x) \neq \gamma(x)\}\,|$. Die Kugel einer Belegung β über einer Variablenmenge V und mit Radius r ist $K_V(\beta, r) := \{\gamma \mid d_V(\beta, \gamma) \leq r\}$.
Das Erfüllbarkeitsproblem für Formeln in k-KNF wird abgekürzt durch k-SAT. Das Suchproblem PROMISE-BALL-k-SAT ist wie folgt definiert: Ausgehend von einer Formel F in k-KNF über der Variablenmenge $V = \texttt{var}(F)$, einer Belegung β für V, einem Radius $r \in \mathbb{N}$ und der Annahme, dass die Kugel $K_V(\beta, r)$ zumindest eine Belegung enthält, die F erfüllt, gilt es irgendeine erfüllende Belegung für F zu finden. Nicht notwendig ist allerdings, dass die gefundene Belegung innerhalb des Suchradius liegt. Es ist möglich k-SAT in mehrere Probleme der Form PROMISE-BALL-k-SAT zu zerlegen. Dazu betrachten wir eine Formel F in k-KNF über einer Variablenmenge $V = \texttt{var}(F)$ und eine Menge B_V die alle möglichen Belegungen für V enthält. Weiterhin sei C_V eine Teilmenge von B_V und $r \in \mathbb{N}$. Wenn für jedes $\beta \in C_V$ das Suchproblem PROMISE-BALL-k-SAT gelöst wird, wird dadurch k-SAT entschieden, wenn

$$\bigcup_{\beta \in C_V} K_V(\beta, r) = B_V \; .$$

Denn ist F erfüllbar, liegt zumindest eine erfüllende Belegung in B_V und somit existiert auch eine Kugel, die diese Belegung enthält. Also wird zumindest ein Aufruf von PROMISE-BALL-k-SAT

sie finden. Andererseits, wenn F unerfüllbar ist, dann existiert keine erfüllende Belegung, und sie kann daher auch nicht gefunden werden.

3.2 Formeln reduzieren

Wenn angenommen wird, dass eine Variable einer aussaugenlogischen Formel mit einem festen Wahrheitswert permanent belegt ist, dann kann die Formel zu einer äquivalenten Formel reduziert werden, indem alle Klauseln, die aufgrund der permanenten Variablenbelegung erfüllt sind, aus der Formel entfernt werden und ebenso alle Literale, die aus demselben Grund einen Wahrheitswert von `0` haben, aus den Klauseln entfernt werden.

Algorithmus 2 `reduce` (Formel F in k-KNF, Variable v, Wahrheitswert b)

1. $F' \leftarrow \emptyset$
2. **for all** $K \in F$ **do**
3.   **if** $b = 0$ **then**
4.   **if** $\bar{v} \notin K$ **then** $F' \leftarrow F' \cup (K \setminus \{v\})$ **end if**
5.   **else**
6.   **if** $v \notin K$ **then** $F' \leftarrow F' \cup (K \setminus \{\bar{v}\})$ **end if**
7.   **end if**
8. **end for**
9. **return** F'

Lemma 3.1. *Sei F eine KNF-Formel, v eine in F vorkommende Variable, b ein Wahrheitswert und $F' = \texttt{reduce}(F, v, b)$ dann gilt:*

1. *Für jede Belegung β mit $\beta(v) = b$ ist F durch β erfüllt genau dann wenn F' durch β erfüllt ist.*
2. *$v \notin \texttt{var}(F')$*

Beweis.

1. Fall $b = 0$: Jede Klausel die das Literal $u = \bar{v}$ enthält, ist für jede mögliche Belegung β mit $\beta(v) = b$ erfüllt, und zwar unabhängig davon ob die anderen Literale in der Klausel erfüllt sind. Die Klausel ist also ein neutrales Element bezüglich der Konjunktion und kann aus der Formel entfernt werden. Jede Klausel die das Literal $u = v$ enthält, ist für keine Belegung β mit $\beta(v) = b$ durch u erfüllt. u ist also ein neutrales Element bezüglich der Disjunktion und kann aus der Klausel entfernt werden.

   Fall $b = 1$: Analog zu Fall $b = 0$, nur das zuerst $u = v$ und dann $u = \bar{v}$ betrachtet wird.

2. Der Algorithmus $\texttt{reduce}(F, v, b)$ iteriert durch alle Klauseln aus F. Bei jeder Iteration gilt: Fall $b = 0$: Entweder ist $\bar{v}$ in der Klausel und die Klausel wird daher nicht in die reduzierte Formel übernommen oder die Klausel wird ohne v übernommen. Fall $b = 1$: Entweder ist v in der Klausel und die Klausel wird daher nicht in die reduzierte Formel übernommen oder die Klausel wird ohne $\bar{v}$ übernommen. Folglich ist $v \notin \texttt{var}(\texttt{reduce}(F, v, w))$.

□

3.3 Lokale Suche von Dantsin u.a.

Die theoretische Untersuchung der lokale Suchstrategie, die bei dem k-SAT Algorithmus von Dantsin u.a. zum Einsatz kommt, wird in diesem Abschnitt vorgestellt.

Algorithmus 3 `sb` (Formel F in k-KNF, Belegung β, Radius r)

1. **if** β erfüllt F **then**
2. **return** `true`
3. **else if** $r = 0$ **then**
4. **return** `false`
5. **else**
6. $K \leftarrow$ eine durch β nicht erfüllte Klausel in F
7. **for all** $u \in K$ **do**
8. $v \leftarrow$ die u zugrunde liegende Variable
9. $\beta' \leftarrow \beta$
10. $\beta'(v) \leftarrow 1 - \beta(v)$
11. $F' \leftarrow \texttt{reduce}(F, \beta', v)$
12. **if** $\texttt{sb}(F', \beta', r-1)$ **then** **return** `true` **end if**
13. **end for**
14. **return** `false`
15. **end if**

Lemma 3.2. *Sei eine Formel F in k-KNF, eine Belegung β für die Variablen von F und ein Radius r. Der Algorithmus $\texttt{sb}(F, \beta, r)$ liefert `true`, wenn* PROMISE-BALL-k-SAT *für F, β, r lösbar ist. Wenn F unerfüllbar ist, dann liefert er `false`.*

Beweis. Betrachten wir zunächst den Fall, dass F erfüllbar und PROMISE-BALL-k-SAT für F, β, r lösbar ist. Aufgrund der Problemdefinition von PROMISE-BALL-k-SAT gibt es $l \geq 1$ Belegungen $\beta_1^*, ..., \beta_l^*$ die F erfüllen mit $d_H(\beta_i^*, \beta) \leq r$. Weil eine Klausel K gewählt wird, die durch β nicht erfüllt ist, muss für jede erfüllende Belegung β^* ein Literal $u \in K$ existieren, so dass für $v = \texttt{var}(u)$ gilt $\beta(v) \neq \beta^*(v)$. Weiterhin wird für jedes Literal $u \in K$ mit $v = \texttt{var}(u)$ eine Belegung β' aus β erzeugt, in der $\beta'(v) = 1 - \beta(v)$ ist. Also existiert für jedes β^* ein $u \in K$ mit $v = \texttt{var}(u)$, so dass $\beta(v) \neq \beta^*(v)$ und $\beta'(v) = \beta^*(v)$ und daher wird für jedes β^* mindestens ein rekursiver Aufruf des Algorithmus gestartet, bei dem gilt $d_H(\beta', \beta^*) \leq r - 1$. Da `sb` sich die Variablen, dessen Belegung geändert wird, anhand der nicht erfüllten Klauseln sucht, zeigt Lemma 3.1, dass eine einmal durchgeführte Veränderung der Belegung einer Variable wegen `reduce` im weiteren Verlauf des Algorithmus nicht erneut verändert wird. Folglich gibt es für jedes β^* einen Pfad im Rekursionbaum mit der maximalen Rekursionstiefe r, der zu einer Belegung β' führt, für die gilt $d_H(\beta', \beta^*) = 0$. Der Algorithmus liefert `true`, sobald das zum erstem mal der Fall ist. Wenn F nicht erfüllbar ist, existiert keine Belegung die F erfüllt und der Algorithmus terminiert mit `false`, wenn die maximale Rekursionstiefe erreicht ist. □

Lemma 3.3. *Sei eine Formel in k-KNF über n Variablen und r ein Radius. Dann hat der Algorithmus `sb` eine Zeitkomplexität in $O(poly(n) \cdot k^r)$.*

Beweis. Betrachten wir die rekursiven Aufrufe des Algorithmus als Baum, dann beträgt die Anzahl der rekursiven Aufrufe in einem Knoten k und die Höhe des Baums maximal r. Der Rekur-

sionsbaum enthält also maximal k^r Blätter. Weiterhin ist das Testen der Belegung und die Reduzierung von F in jedem Knoten mit polynomieller Laufzeit in Abhängigkeit von der Anzahl der Variablen n zu berücksichtigen. Es resultiert somit eine Zeitkomplexität von $O(\text{poly}(n) \cdot k^r)$. □

Lemma 3.4. *Sei F eine Formel in k-KNF über n Variablen, β eine Belegung für diese und r ein Radius. Wenn alle durch β unerfüllten Klauseln in F maximal $k-1$ Literale enthalten, dann hat* $\texttt{sb}(F, \beta, r)$ *hat eine Zeitkomplexität in $O(poly(n) \cdot (k-1)^r)$.*

Beweis. Der Algorithmus wählt ausschließlich unerfüllte Klauseln, durch die iteriert wird. Diese haben alle $k-1$ Literale und daher finden in jedem Knoten des Rekursionsbaumes ebenfalls maximal $k-1$ rekusive Aufrufe statt. Eine erfüllte Klausel kann nur zu einer Unerfüllten werden, wenn die Belegung einer Variable permanent geändert wird, also die Formel reduziert wird. Aus Lemma 3.1 geht hervor, dass eine Klausel die nach der Reduzierung unerfüllt ist ebenfalls maximal $k-1$ Literale enthalten kann. □

3.4 Lokale Suche von Moser und Scheder

Für den Algorithmus von Moser und Scheder ist es notwendig, dass die Literale einer Klausel eine beliebige aber feste Position besitzen. Es wird zunächst der Hilfsalgorithmus `flip` betrachtret, der die Belegung einzelner Variablen gezielt negiert.

Algorithmus 4 `flip` (Menge $\{K_1, ..., K_m\}$ von Klauseln, Belegung β, Vektor $(w_1, ..., w_m) \in \{1, ..., k\}^m$)

Require: $\forall i \in \{1, ..., m\} : |K_i| = k$

1. $\beta' \leftarrow \beta$
2. **for** $i \in \{1, ..., m\}$ **do**
3. $\quad u \leftarrow$ das w_i-te Literal in K_i
4. $\quad \beta'(\texttt{var}(u)) \leftarrow 1 - \beta'(\texttt{var}(u))$
5. **end for**
6. **return** β'

Beispiel 3.1. *Sei eine Formel $(x_1 \vee x_2 \vee x_3) \wedge (y_1 \vee y_2 \vee y_3) \wedge (z_1 \vee z_2 \vee z_3)$ mit der entsprechenden Mengendarstellung $K = \{\{x_1, x_2, x_3\}, \{y_1, y_2, y_3\}, \{z_1, z_2, z_3\}\}$, eine Belegung β, die allen Variablen den Wahrheitswert 0 zuordnet und ein Vektor $w = (1, 2, 3)$. Dann ist $\beta' = \texttt{flip}(K, \beta, w)$ eine Belegung in der die Wahrheitswerte für x_1, y_2 und z_3 negiert wurden. Das bedeutet, β' belegt die drei genannten Variablen mit 1 und die restlichen weiterhin mit 0.*

Nachfolgend wird der eigentliche Algorithmus der lokalen Suche von Moser und Scheder betrachtet. Eine Vorrausbedingung für den Algorithmus ist $t = t(r) = \log_2 \log_2 r$.

Lemma 3.5. *Sei eine Formel F in k-KNF, eine Belegung β und ein Radius r. Der Algorithmus* `sb-fast` *liefert* `true`*, wenn* PROMISE-BALL-k-SAT *für F,β,r lösbar ist. Wenn F nicht erfüllbar ist, dann liefert* `sb-fast` *`false`.*

Beweis. Betrachten wir zunächst den Fall, dass F erfüllbar und PROMISE-BALL-k-SAT für F, β, r lösbar ist. Aufgrund der Problemdefinition von PROMISE-BALL-k-SAT gibt es $l \geq 1$ Belegungen

Algorithmus 5 `sb-fast` ($k \in \mathbb{N}$, Formel F in k-KNF, Belegung β, Radius r, Code $\mathcal{C} \subseteq \{1, ..., k\}^t$ mit Radius t/k)

1. **if** β erfüllt F **then**
2. **return** `true`
3. **else if** $r = 0$ **then**
4. **return** `false`
5. **else**
6. $G \leftarrow$ maximale Menge paarweise disjunkter Klauseln aus F, die jeweils genau k Literalen enthalten und durch β nicht erfüllt sind
7. **if** $|G| < t$ **then**
8. $\{v_1, ..., v_m\} \leftarrow \mathtt{var}(G)$ mit $m = |\mathtt{var}(G)|$
9. **for all** $(w_1, ..., w_m) \in \{0, 1\}^m$ **do**
10. $F' \leftarrow F$
11. **for all** $i \in \{1, ..., m\}$ **do**
12. $\beta(v_i) \leftarrow w_i$
13. $F' \leftarrow \mathtt{reduce}(F', v_i, w_i)$
14. **end for**
15. **if** $\mathtt{sb}(F', \beta, r)$ **then** **return** `true` **end if**
16. **end for**
17. **return** `false`
18. **else**
19. $H \leftarrow \{K_1, ..., K_t\} \subseteq G$ und mit beliebiger aber fest gewählter Reihenfolge
20. **for all** $w = (w_1, ..., w_t) \in \mathcal{C}$ **do**
21. $\beta' \leftarrow \mathtt{flip}(H, \beta, w)$
22. $r' \leftarrow r - (t - 2t/k)$
23. **if** $\mathtt{sb\text{-}fast}(k, F, \beta', r', \mathcal{C})$ **then** **return** `true` **end if**
24. **end for**
25. **return** `false`
26. **end if**
27. **end if**

$B = \{\beta_1^*, ..., \beta_l^*\}$ die F erfüllen und für jedes $\beta^* \in B$ gilt $\beta^* \in K(\beta, r)$ also $d_{\text{var}(F)}(\beta^*, \beta) \leq r$. Wenn bereits feststeht, dass F durch β nicht erfüllt ist und der maximale Radius noch nicht erreicht wurde, dann wird zwischen zwei Fällen unterschieden.
Im Fall $|G| < t$ werden basierend auf einem Vektor $(w_1, ..., w_m) \in \{0,1\}^m$ die Variablen $\{v_1, ..., v_m\} \in \texttt{var}(G)$ so belegt, dass $\bigwedge_{j=1}^m \beta(v_j) = w_j$. Wenn nun für ein β_i^* gilt

$$\bigwedge_{j=1}^{m} \beta(v_j) = \beta_i^*(v_j) \,, \tag{3.1}$$

dann liefert $\texttt{sb}(F', \beta, r)$ aufgrund von Lemma 3.2 `true` und somit terminiert der dieser Algorithmus mit ebenfalls mit `true`. Weil dieser Vorgang für jedes mögliche $(w_1, ..., w_m) \in \{0,1\}^m$ erfolgt, muss (3.1) für jedes $\beta^* \in B$ einmal gelten.
Im Fall $|G| \geq t$ wird eine Menge $H = \{K_1, ..., K_t\}$ nicht erfüllter Klauseln gewählt. Also existiert für jedes $\beta^* \in B$ in jeder Klausel aus H ein Literal mit einer zugrunde liegenden Variable v, so dass $\beta(v) \neq \beta^*(v)$. Die jeweiligen Positionen dieser Literale in den Klauseln ist beschrieben durch einen Vektor $w^* = (w_1, ..., w_t)$. Zusammengefasst bedeutet das: Für jedes $\beta^* \in B$ existiert mindestens ein $w^* \in \{1, ..., k\}^t$ mit $\beta' = \texttt{flip}(H, \beta, w^*)$, so dass $d_{\text{var}(F)}(\beta, \beta^*) = d_{\text{var}(F)}(\beta', \beta^*) - t$. Weiterhin ist jeder Vektor aus dem Raum $\{1, ..., k\}^t$ durch $\mathcal{C}$ abgedeckt. Beim Durchlaufen aller $w \in \mathcal{C}$ existiert also für jedes w^* mit dem entsprechenden β^* ein $w' \in \mathcal{C}$, so dass $w^* \in K_{\{1,..,k\}}(w', t/k)$ und somit auch $d_{\text{var}(F)}(w', w^*) \leq t/k$. Wenn also eine Belegung $\beta' = \texttt{flip}(H, \beta, w')$ erzeugt wird, dann sind auf der einen Seite im schlechtesten Fall t/k Variablen in β' unterschiedlich zu β^* belegt, die zuvor (in β) gleich belegt waren. Auf der anderen Seite, sind folglich $t - t/k$ Variablen gleich zu β^* belegt, die zuvor unterschiedlich waren. Der resultierende Nettogewinn ist $\Delta = (t - t/k) - (t/k) = t - 2t/k$.
Demzufolge wird für jedes $\beta^* \in B$ in einer Iteration ein β' erzeugt, so dass $d_{\text{var}(F)}(\beta', \beta^*) = d_{\text{var}(F)}(\beta, \beta^*) - \Delta$. Da bei jedem rekursiven Aufruf $r - \Delta$ als Radius übergeben wird und der Algorithmus initial mit $d_{\text{var}(F)}(\beta, \beta^*) \leq r$ gestartet ist, gilt bei einer Rekursiontiefe von r/Δ wenn $r = 0$ auch $d_{\text{var}(F)}(\beta, \beta^*) = 0$. Also jede Belegung β^* würde gefunden werden, wenn der Algorithmus nicht schon bei der ersten gefundenen erfüllenden Belegung terminieren würde. □

Lemma 3.6. *Sei eine Formel in k-KNF über n Variablen und ein Radius r. Der Algorithmus* `sb-fast` *hat eine Laufzeit von $O(poly(n) \cdot (k-1)^r)$.*

Beweis. G ist eine Menge aus paarweise disjunkten k-Klauseln die, nicht erfüllt sind. Also enthalten alle nicht erfüllten Klauseln, die nicht in G sind, mindestens eine Variable, die in einer der Klauseln in G ebenfalls enthalten ist, oder es sind Klauseln die weniger als k Literale enthalten. Diese Beobachtung zeigt, dass wenn F um die Variablen $\texttt{var}(G)$ reduziert wird, alle unerfüllten Klauseln in F maximal $k-1$ Literale enthalten.
Im Fall $|G| < t$ werden für die $k \cdot |G|$ Variablen in G alle möglichen $2^{k \cdot |G|}$ Belegungen erzeugt und jeweils mit einem um $\texttt{var}(G)$ reduzierten F' der Algorithmus `sb` aufgerufen. Aus der vorangegangenen Beobachtung und Lemma 4 ergibt sich für diesen Fall zunächst eine Laufzeit von $O(2^{k \cdot |G|} \cdot \text{poly}(n) \cdot (k-1)^r)$. Weil $|G| < t < \log_2 r$ und offensichtlich $r \leq n$, gilt auch $2^{k \cdot |G|} < 2^{k \cdot t} < 2^{k \cdot \log_2 r} \leq 2^{k \cdot \log_2 n} = 2^k \cdot n$. Folglich gilt in diesem Fall eine Laufzeit von $O(\text{poly}(n) \cdot (k-1)^r)$.
In dem Fall $|G| \geq t$ werden $|\mathcal{C}|$ rekursive Aufrufe gestartet. Als Radius wird bei jedem rekursiven Aufruf $r - (t - 2t/k)$ übergeben. Das ergibt eine Rekursiontiefe von $r/(t - 2t/k)$. Der Rekursi-

onbaum hat also $|\mathcal{C}|^{r/(t-2t/k)}$ Blätter. Moser und Scheder haben in ihrer Arbeit [4] gezeigt:

$$|\mathcal{C}|^{r/(t-2t/k)} \leq (t^2(k-1)^{t-2t/k})^{r/(t-2t/k)} = (t^{2/(t-2t/k)}(k-1))^r$$

Da $t = t(r_0) = \log\log r_0$ eine Funktion mit $\lim_{t\to\infty} t^{2/(t-2t/k)} = 1$ ist und unter Berücksichtigung, dass die Berechnung von H in polynomieller Laufzeit in Abhängigkeit von n möglich ist, ergibt sich eine Laufzeit für den Fall $|G| \geq t$ von $O(\text{poly}(n) \cdot (k-1)^r)$.
Da die Verifizierung von F in polynomieller Zeit also $O(\text{poly}(n))$ durchführbar ist, das Prüfen von $r = 0$ in $O(1)$ und die verbleibenden Fälle $|G| < t$ und $|G| \geq t$ in $O(\text{poly}(n) \cdot (k-1)^r)$, hat `sb-fast` eine Laufzeit von $O(\text{poly}(n) \cdot (k-1)^r)$. □

Zwar erreicht der Algorithmus `sb-fast` theoretische Laufzeit, die in $O((k-1)^r)$ liegt, bei Formeln über n Variablen in k-KNF und einem Kugelradius r, allerdings ist dazu notwendig, dass $t = t(r)$ gegen ∞ geht. In der Praxis könnte das bedeuten, r und letztendlich auch n müssten sehr große Werte annehmen, damit sich `sb-fast` an die genannte Zeitkomplexität annähert. Welche Laufzeiten bei *realistischeren* Werten erreicht werden, soll an dieser Stelle kurz untersucht werden. Wie schon im Beweis zu Lemma 3.6 festgestellt, hat `sb-fast` im Fall $|G| \geq t$ eine Laufzeit, die sich aus den $|\mathcal{C}|$ rekursiven Aufrufen bei einer Rekursionstiefe von $r/(t-2t/k)$ ergibt. Wenn man Formeln in 3-KNF betrachtet, resultiert daraus die Laufzeit $(f(t))^r$ wobei $f(t) = 2 \cdot t^{6/t}$. Zwar konvergiert die Funktion $f(t)$ gegen 2, allerdings nimmt sie auf dem Weg dahin auch höhere Werte an, wie aus Abbildung 3.1 entnommen werden kann. Als Beispiel betrachten wir eine Formel in 3-KNF über $n = 1024$ Variablen. Der Radius für die zu durchsuchenden Kugeln werde durch $r = 1024/(3+1) = 256$ bestimmt. Daraus ergibt sich $t = t(256) = \log_2\log_2 256 = 3$. Für 3-KNF Formeln über $n \leq 1024$ Variablen hätte der Algorithmus also eine Laufzeit in $O(f(3)^r) = O(18^r)$.

3.5 k-SAT Algorithmus von Moser und Scheder

Was der k-SAT Algorithmus prinzipiell zu tun hat, ist, für jede Belegung eines Covering Codes den Algorithmus für die lokale Suche zu starten. Mit den in Kapitel 2 vorgestellten Algorithmen lassen sich Covering Codes, dessen Mächtigkeit nur um einen polynomiellen Faktor größer ist als der entsprechende optimale Covering Code, in ausreichender Zeit konstruieren. Es wird hier daher davon ausgegangen, dass ein passender Covering Code bereits zur Verfügung steht. Es bleibt die Frage, wie der Suchradius optimalerweise gewählt werden sollte. In Dantsins Algorithmus, auf dem dieser aufbaut, wird für Formeln in k-KNF über n Variablen ein Radius von $n/(k+1)$ verwendet. Allerdings hat die lokale Suche dort eine Laufzeit von $O(k^r)$. Wie der Radius bei der

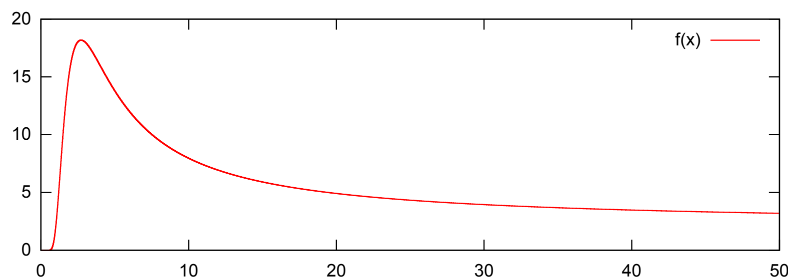

Abbildung 3.1: $f(t) = 2 \cdot t^{6/t}$

verbesserten lokalen Suche mit der Laufzeit $O(k-1)^r$ am besten zu wählen ist, soll zunächst ermittelt werden. Wenn wir polynomielle Faktoren erst einmal außen vor lassen, dann hat der Algorithmus eine Laufzeit $L(n,k)$ die sich aus dem Produkt der Mächtigkeit des Covering Code und der Laufzeit der lokalen Suche ergibt.

$$\begin{aligned} L(n,k) &= 2^{(1-h(r/n))n} \cdot (k-1)^r \\ &= 2^{((1-h(r/n))+\log_2(k-1)r/n)n} \\ &= 2^{l(r)n} \end{aligned}$$

Es wird ein r gesucht, so dass $l(r) = 1-h(r/n)+\log_2(k-1)r/n$ minimal ist. Um den Extremwert zu bestimmen, wird die Ableitung gebildet.

$$\begin{aligned} l'(r) &= 1 - h(r/n) + \log_2(k-1)r/n \\ &= \log_2(k-1) - \log_2(1-r/n) + \log_2 r/n \\ &= \log_2 \frac{(k-1)r/n}{1-r/n} \end{aligned}$$

Da $l'(n/k) = 0$, ist n/k der gesuchte Extremwert und damit der zu verwendende Suchradius. Nun wird der Radius in $l(r)$ eingesetzt.

$$\begin{aligned} l(n/k) &= 1 - h\left(\frac{1}{k}\right) + \log_2(k-1)\frac{1}{k} \\ &= 1 + \frac{1}{k} \cdot \log_2 \frac{1}{k} + \frac{k-1}{k} \cdot \log_2 \frac{k-1}{k} + \log_2(k-1)\frac{1}{k} \\ &= 1 + \log_2 \frac{1}{k}\left(\frac{1}{k} + \frac{k-1}{k}\right) + \log_2(k-1)\left(\frac{k-1}{k} + \frac{1}{k}\right) \\ &= 1 + \log_2 \frac{k-1}{k} \end{aligned}$$

Dieses Ergebnis wird nun in die Ausgangsgleichung eingesetzt.

$$L(n,k) = 2^{l(n/k)n} = 2^n 2^{n \log_2 \frac{k-1}{k}} = \left(2\,\frac{k-1}{k}\right)^n$$

Algorithmus 6 `sat-moser` (k-KNF Formel F über n Variablen)

1. Es sei $\{v_1, ..., v_n\} = \text{var}(F)$
2. $r \leftarrow n/k$
3. $t \leftarrow \log_2 \log_2 r$
4. Sei $\mathcal{C}_1$ ein Covering Code über $\{0,1\}$ der Länge n mit Radius r
5. Sei $\mathcal{C}_2$ ein Covering Code über $\{1, ..., k\}$ der Länge t mit Radius t/k
6. **for all** $\{w_1, ..., w_n\} \in \mathcal{C}_1$ **do**
7. Es sei β eine Belegung mit $\beta(v_i) = w_i$
8. **if** `sb-fast`$(F, \beta, r, \mathcal{C}_2)$ **then return** `true`
9. **end for**
10. **return** `false`

Satz 3.1. *Sei eine Formel F in k-KNF über n Variablen. Dann entscheidet der Algorithmus* `sat-moser` *das Problem k-SAT für F deterministisch in der Zeit $O(poly(n) \cdot (2(k-1)/k)^n)$.*

Beweis. Zunächst wird gezeigt, dass der Algorithmus k-SAT korrekt entscheidet. Sei F eine k-KNF Formel über der Variablenmenge $V = \texttt{var}(F)$ und B die Menge der erfüllenden Belegungen für diese Variablen. Dann existiert für jedes $\beta^* \in B$ ein $\beta \in \mathcal{C}$, so dass $d_V(\beta, \beta^*) \leq r$. Wegen Lemma 3.5 wird für jede erfüllende Belegung zumindest ein Aufruf von `sb-fast` den Wert `true` liefern (Lemma 3.5) und somit liefert `sat-moser` ebenfalls `true`. Wenn F unerfüllbar ist, liefert jeder Aufruf von `sb-fast` den Wert `false` (Lemma 3.5) und auch `sat-moser` liefert `false`. `sat-moser` ist deterministisch, weil `sb-fast` deterministisch ist. Die Laufzeit folgt aus der obigen Berechnung. □

3.6 k-SAT Algorithmus von Dantsin

Zur Vollständigkeit wird hier der k-SAT Algorithmus von Dantsin u.a. kurz vorgestellt. Er unterscheidet sich nur in wenigen Punkten von `sat-moser`. Der Radius für die lokale Suche beträgt $k + 1$, für die lokale Suche wird der Algorithmus `sb` verwendet und demzufolge ist der Covering Code $\mathcal{C}_2$ hier nicht notwendig. Die Korrektheit und die Zeitkomplexität kann nach der gleichen Vorgehensweise bestimmt werden wie bei `sat-moser`. Die Zeitkomplexität beträgt $O(\text{poly}(n) \cdot (2 - 2/(k+1))^n)$.

k	3	4	5	6	7	...	∞
$2 - 2/(k+1)$	1,5	1,6	1,667	1,714	1,75	...	2

Algorithmus 7 `sat-dantsin` (Formel F in k-KNF über n Variablen)

1. Es sei $\{v_1, ..., v_n\} = \text{var}(F)$
2. $r \leftarrow n/(k+1)$
3. Sei $\mathcal{C}_1$ ein Covering Code über $\{0, 1\}$ der Länge n mit Radius r
4. **for all** $\{w_1, ..., w_n\} \in \mathcal{C}_1$ **do**
5. Es sei β eine Belegung mit $\beta(v_i) = w_i$
6. **if** $\texttt{sb}(F, \beta, r)$ **then return** `true`
7. **end for**
8. **return** `false`

3.7 k-SAT Algorithmus von Schöning

Auch der besagte Algorithmus von Schöning soll hier kurz vorgestellt werden.

Algorithmus 8 `sat-schoening` (Formel F in k-KNF über n Variablen)

1. $t \leftarrow (2 - 2/k)^n$
2. **for** $i = 1 \rightarrow t$ **do**
3. $\quad\beta \leftarrow$ Eine zufällig gewählte Belegung für die Variablen von F
4. $\quad$**for** $j = 1 \rightarrow 3n$ **do**
5. $\quad\quad$**if** F durch β erfüllt **then return** `true`
6. $\quad\quad C \leftarrow$ beliebige Klausel in F, die durch β nicht erfüllt ist
7. $\quad\quad u \leftarrow$ zufällig gewähltes Literal aus C
8. $\quad\quad \beta(\text{var}(u)) \leftarrow 1 - \beta(\text{var}(u))$
9. $\quad$**end for**
10. **end for**
11. **return** `false`

Es ist einfach feststellbar, dass dieser randomisierte Algorithmus eine Zeitkomplexität in $O(\text{poly}(n) \cdot (2 - 2/k)^n)$ hat. Schöning hat gezeigt[6], dass dieser Algorithmus k-SAT mit einer Wahrscheinlichkeit von $1 - \mathbf{e}^{-20}$ korrekt entscheidet.

k	3	4	5	6	7	...	∞
$2 - 2/k$	1,333	1,5	1,6	1,667	1,714	...	2

Kapitel 4

Experimente

4.1 Vorgehensweise

Für die folgenden Experimente mit den k-SAT Algorithmen wurden entweder Formeln verwendet, von denen bekannt ist, dass für sie genau eine erfüllende Belegung existiert, oder die Formeln wurden zufällig erzeugt. Da unter anderem eine Abschätzung der Laufzeiten in Abhängikeit der Anzahl der Variablen n der Formeln vollzogen werden sollte, wurden stets Formeln mit unterschiedlichen n betrachtet. Bei zufällig erzeugten Formeln ist die Anzahl der Klauseln m zumeist so gewählt worden, dass die Wahrscheinlichkeit, dass sie erfüllbar ist, bei etwa 0,5 liegt. So können jeweils genügend Messdaten für erfüllbare und unerfüllbare Formeln gesammelt werden. Weiterhin wird angenommen, dass der Schwierigkeitsgrad, das Entscheidungsproblem zu lösen, auf diese Weise vergleichbar ist für verschiedene n. (Die Unsicherheit ist am größten, wenn beide Ereignisse gleichwahrscheinlich sind.)
Für jeden Lauf von einem der Algorithmen wurde die Laufzeit in Millisekunden und die Anzahl der getesteten Variablenbelegungen ermittelt, die benötigt wurden um k-SAT bzw. PROMISE-BALL-k-SAT lösen. Aus den so gesammelten Messwerten sind getrennt nach n, Laufzeit und Anzahl der getesteten Belegungen und erfüllbaren und unerfüllbaren Formeln die arithmetischen Mittelwerte gebildet worden.
In einigen Experimenten ist aus den Mittelwerten das Wachstum der Laufzeit in Abhängigkeit von n approximiert worden. Dies geschah mit Hilfe der Methode der kleinsten Quadrate, ausgehend von der Modellfunktion

$$f(n) = a \cdot 2^{bn} \quad \Leftrightarrow \quad \log_2 f(n) = \log_2(a) + bn \; .$$

Wobei $a, b \in \mathbb{R}$ die zu bestimmenden Konstanten sind. Weiterhin sind die Fehler nicht absolut, sondern prozentual zum jeweiligen Messwert betrachtet worden, indem f und ebenso die Messwerte logarithmiert wurden.

4.2 Formeln erzeugen

Es steht eine Formel F_{12} in 3-KNF über 12 Variablen und mit 40 Klauseln zur Verfügung, von der bekannt ist, dass für sie genau eine erfüllende Belegung der Variablen existiert. Diese Formel ist

n	m	3-KNF-Formeln pro (n, m)
4	$8, 9, 10, ..., 40$	200
8	$16, 18, 20, ..., 80$	200
12	$24, 27, 30, ..., 120$	200
16	$32, 36, 40, ..., 160$	200
20	$40, 45, 50, ..., 200$	50

Tabelle 4.1: Testfälle zum Ermitteln der Wahrscheinlichkeit p der Erfüllbarkeit von zufällig erzeugten Formeln in 3-KNF. Besonders von Interesse sind die Werte von $\alpha = m/n$, an denen $p = 1/2$.

im Anhang B zu finden. Aus F_{12} lassen sich Formeln

$$F_{12\ell} = \bigwedge_{i=1}^{\ell} F_{12}\left[x_j \to x_{12i+j}\right]$$

über $n = 12\ell$ Variablen und mit $m = 40\ell$ Klauseln konstruieren, wobei durch die Bezeichnung $F_{12}\left[x_j \to x_{12i+j}\right]$ eine Formel dargestellt wird, die F_{12} entspricht, nur das alle Variablen sinngemäß umbenannt wurden. Für eine solche Formel existiert ebenfalls genau eine erfüllende Belegung. Bei den Experimenten wurde darauf geachtet, dass der Algorithmus die Klauseln in regelmäßigen Abständen gleichverteilt durcheinander würfelt, wenn eine solche Formel getestet wird. So wird vermieden, dass die Formeln Teil für Teil gelöst werden. Was zu einer drastischen Verbesserung der Laufzeit führen könnte, aber nichts über die tatsächliche Laufzeit des Algorithmus aussagen würde.
Um deterministische k-SAT-Algorithmen in der Praxis zu untersuchen, ist es notwendig, dass viele verschiedene Formeln zu Verfügung stehen. Formeln in k-KNF über n Variablen mit m Klauseln lassen sich zufällig erzeugen, indem aus den k-elementigen Teilmengen der Variablenmenge $\{x_1, ..., x_n\}$ gleich-verteilt m Klauseln ausgewählt werden. Weiterhin wird jedes Literal in den Klauseln mit einer unabhängigen Wahrscheinlichkeit von $1/2$ negiert. Das Verhältnis zwischen der Anzahl der Variablen und der Anzahl der Klauseln $m = \alpha n$ ist dabei von Bedeutung für die Frage, ob eine so erzeugte Formel erfüllbar ist. Ein Experiment (siehe Tabelle 4.1) mit 3-KNF-Formeln hat gezeigt, dass Formeln mit *kleinem* α eine gegen $p = 1$ gehende Wahrscheinlichkeit haben erfüllbar zu sein und Formeln mit *großem* α eine gegen $p = 0$ gehende Wahrscheinlichkeit. Der Übergangsbereich, indem p deutlich zwischen 0 und 1 liegt, ist umso kleiner, desto größer n ist. Die genauen Ergebnisse des Experiments sind im Abbildung 4.1 zu sehen. Die Wertepaare (n, m) für die gilt $p = 1/2$ werden bei den Laufzeittests der SAT-Algorithmen noch von Bedeutung sein.

4.3 Testumgebung

Zur Implementierung der Algorithemn wurde die Programmiersprache Python in der Version 2.7 eingesetzt. Als Testsystem wurde ein PC mit 4 GB Arbeitsspeicher, einem 64-bit CPU (AMD Athlon64 X2 5000+) mit zwei Kernen und 2,7 GHz Taktfrequenz verwendet. Es wurde keine Form der Parallelverarbeitung implementiert, die Algorithmen hatten, abgesehen von den schlafenden Hintergrundprozessen des Betriebssystems, in einem Durchlauf stets genau einen Kern fast exklusiv zur Verfügung. Zeitmessungen sind mit Hilfe der Python-Bibliothek `time` durchgeführt

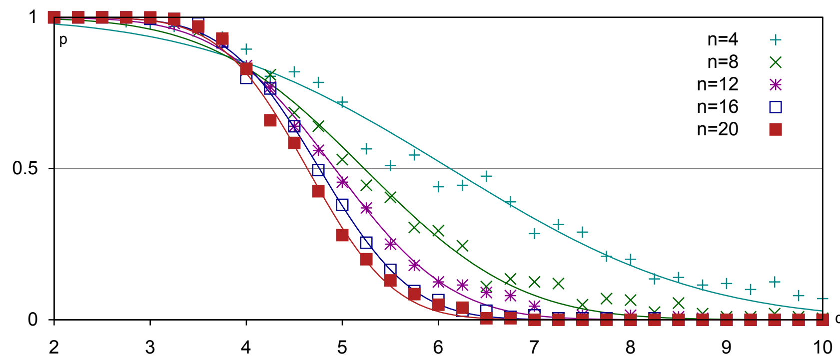

Abbildung 4.1: Testergebnisse für Formeln in 3-KNF über n Variablen. Eine zufällig erzeugte Formel mit $m = \alpha n$ Klauseln ist mit einer Wahrscheinlichkeit p erfüllbar. Für jeden Punkt wurden 200 Formeln gestestet. Die Punkte, an denen $p = 1/2$ sind näherungsweise $\alpha_{n=4} = 6{,}10$, $\alpha_{n=8} = 5{,}20$, $\alpha_{n=12} = 4{,}90$, $\alpha_{n=16} = 4{,}74$ und $\alpha_{n=20} = 4{,}63$.

worden und als Quelle für Pseudo-Zufallszahlen diente die Python-Bibliothek `random`. Zum Approximieren der Laufzeiten wurde das Programm Gnuplot eingesetzt. Es verwendet den auf der Methode der kleinsten Quadrate aufbauenden Levenberg-Marquardt-Algorithmus (NLLS) zur Bestimmung von Ausgleichsgeraden.

4.4 Lokale Suche bei Formeln mit genau einer erfüllenden Belegung

Beschreibung des Experiments

Es steht eine Formel in 3-KNF über $n_1 = 12$ Variablen und mit $m_1 = 40$ Klauseln zur Verfügung, von der bekannt ist, dass für die Variablen genau eine Belegung existiert, so dass die Formel erfüllt ist. Nach dem in Abschnitt 4.2 beschriebenen Prinzip wurde aufbauend auf dieser Formel eine Weitere über $n_2 = 24$ Variablen und mit $m_2 = 80$ Klauseln konstruiert, für die ebenfalls genau eine erfüllende Belegung existiert. Zum Einsatz kamen die PROMISE-BALL-k-SAT Algorithmen `sb` von Dantsin u.a. und `sb-fast` von Moser und Scheder. Wobei `sb-fast` mit verschieden langen Covering Codes zum „flippen“ der Variablenbelegungen als Eingabe verwendet wurde. Es wurden Covering Codes der Länge $t = 3, 6, 9$ und jeweils mit Radius $t/3$ eingesetzt. Für jede Formel wurden die Algorithmen jeweils mit 1500 gleichverteilt zufällig erzeugten Startbelegungen bei einem Kugelradius von $n/4$ ausgeführt.
Für jeden Algorithmus und jede Formel wurde, differenziert nach Sucherfolg, die Anzahl der in einem Lauf getesteten Variablenbelgungen und die benötigte Zeit in Millisekunden gemessen und arithmetisch gemittelt. Weiterhin ist aus den Messwerten ermittelt worden, wie groß die Wahrscheinlichkeit ist, dass der entsprechende Algorithmus die erfüllende Belegung der jeweiligen Formel findet.

Ergebnisse

Die gemessenen Werte sind in der folgenden Tabelle 4.2 zu finden. Besonders auffällig ist, dass die Wahrscheinlichkeit für das Finden der erfüllenden Belegung bei `sb-fast` weit höher ist, als die theoretische Wahrscheinlichkeit dafür, dass die erfüllende Belegung in der entsprechenden Kugel liegt, die gerade $V_2(n,r)/2^n$ beträgt. Offenbar findet `sb-fast` in den meisten Fällen eine

Belegung, die außerhalb der zu durchsuchenden Kugel liegt.
Der Algorithmus `sb` findet die erfüllende Belegung dagegen mit einer Wahrscheinlichkeit, die in etwa der theoretischen entspricht. Auf der anderen Seite hat `sb-fast` eine deutlich höhere Laufzeit als `sb`. In wie weit sich diese Effekte ausgleichen, zeigen die noch folgenden Experimente der eigentlichen k-SAT Algorithmen.
Weiterhin lässt sich beobachten, dass die Varianten von `sb-fast` sich ebenfalls unterschiedlich verhalten. Zum Einen scheint ab einem gewissen n_0 die Wahrscheinlich für das Finden der erfüllenden Belegung größer zu sein, wenn t größer ist. Man beachte, dass $\texttt{sb-fast}_6$ und $\texttt{sb-fast}_9$ bei $n = 12$ gleiche Wahrscheinlichkeit haben, aber bei $n = 24$ die von $\texttt{sb-fast}_9$ größer ist. Zum Anderen unterscheiden sich die Laufzeiten und die Anzahl der getesteten Variablenbelegungen, allerdings ist es an dieser Stelle schwierig eine Regelmäßigkeit zu erkennen.

Lokale Suche

Formel			Algorithmus	Belegung gefunden			keine Belegung gefunden			exp. WS.	theo. WS.
n	m	r		$\#_1$	∅ Prüf.	∅ Zeit	$\#_2$	∅ Prüf.	∅ Zeit	$\#_1/1500$	$V_2(n,r)/2^n$
12	40	3	`sb`	101	12,6	2,6ms	1399	28,6	6,1ms	0,067	0,073
			`sb-fast`$_3$	773	88,7	29,1ms	727	207,1	68,3ms	0,515	
			`sb-fast`$_6$	934	49,9	54,4ms	566	97,5	29,4ms	0,623	
			`sb-fast`$_9$	934	49,9	54,4ms	566	97,5	29,4ms	0,623	
24	80	6	`sb`	21	105,8	48,6ms	1479	164,2	84,4ms	0,014	0,011
			`sb-fast`$_3$	475	9770,6	3805,9ms	1025	46311,5	17433,4ms	0,317	
			`sb-fast`$_6$	883	2159,6	8664,6ms	617	14757,3	33490,0ms	0,589	
			`sb-fast`$_9$	947	1941,6	19268,0ms	553	2066,9	6225,7ms	0,631	

Tabelle 4.2: Ergebnisse der deterministischen Algorithmen für die lokalen Suche. Getestet wurden zwei Formeln über n Variablen und mit m Klauseln bei jeweils 1500 zufälligen Startbelegungen. Angegeben ist neben der im Mittel getesteten Anzahl von Variablenbelegungen und der benötigten Zeit auch die experimentell bestimmte und die theoretische Wahrscheinlichkeit, dass eine Belegung gefunden wird.

4.5 Laufzeitanalyse von Schönings k-SAT Algorithmus bei Formeln mit genau einer erfüllenden Belegung

Beschreibung des Experiments

Als Grundlage dienten sechs Formeln in 3-KNF über $n = 12, 24, 36, 48, 60, 72$ Variablen und mit $m = 40, 80, 120, 160, 200, 240$ Klauseln, für die jeweils genau eine bekannte erfüllende Variablenbelegung existiert. Sie wurden ausgehend von einer Formel über 12 Variablen und mit 40 Klauseln (siehe dazu Anhang B) nach dem in Abschnitt 4.2 beschriebenen Prinzip konstruiert. Diese Formeln dienten als Eingabe für den k-SAT Algorithmus von Schöning mit einer theoretischen Zeitkomplexität von $O(\text{poly}(n) \cdot 1{,}34^n)$ für 3-KNF Formeln. Für jede Formel ist der Algorithmus 200 mal ausgeführt worden und sowohl die Laufzeit in Millisekunden als auch die Anzahl der getesteten Variablenbelegungen arithmetisch gemittelt worden. Anhand der Mittelwerte wurde Zeitkomplexität des Algorithmus approximiert. Da es sich um einen randomisierten Algorithmus handelt, der k-SAT mit einer Wahrscheinlichkeit von $\mathbf{e}^{-20}$ falsch entscheidet, wurde auch die Anzahl der Fehlentscheidungen gemessen.

Ergebnisse

Die Messwerte sind der nachfolgenden Tabelle 4.3 zu entnehmen. Der Algorithmus hat in den insgesamt 1200 Durchläufen, wie anhand der Fehlerwahrscheinlichkeit zu erwarten war, keine Fehlentscheidung getroffen. Auffällig ist, dass die approximierte Zeitkomplexität recht deutlich unter der theoretischen Komplexität liegt, wobei das Wachstum der Anzahl der getesteten Variablenbelegungen etwas unter dem Wachstum der Laufzeit liegt. Das könnte dadurch erklärt werden, dass bei der Zeitmessung das Testen der Variablenbelegung, welches linear von m abhängt, mit berücksichtig wird. Weiterhin zeigen die Ergebnisse deutlich, dass der Algorithmus nur einen kleinen Bruchteil der möglichen Belegungen betrachten muss, um die erfüllende Belegung zu finden.

k-SAT Algorithmus von Schöning

n	m	2^n	#	Fehler	∅ Prüf.	∅ Zeit	Verhältnis
12	40	4.096	200	0	332	7 ms	12
24	80	16.777.216	200	0	3.720	125 ms	4.510
36	120	68.719.476.736	200	0	24.025	1.101 ms	2.860.332
48	160	$2{,}8 \cdot 10^{14}$	200	0	163.087	9.548 ms	1.725.919.152
60	200	$1{,}2 \cdot 10^{18}$	200	0	939.461	67.731 ms	$4{,}7 \cdot 10^{12}$
72	240	$4{,}7 \cdot 10^{21}$	200	0	5.257.722	436.686 ms	$9{,}0 \cdot 10^{14}$
					$O(1{,}17^n)$	$O(1{,}20^n)$	

Tabelle 4.3: Ergebnisse des randomisierten k-SAT Algorithmus von Schöning mit einer theoretischen Laufzeit $O(\text{poly}(n) \cdot 1{,}34^n)$. Getestet wurden Formeln in 3-KNF über n Variablen und mit m Klauseln, für die genau eine erfüllende Belegung existiert. Zu sehen ist die Anzahl Ausführungen des Algorithmus, die Anzahl der im Mittel geprüften Variablenbelegungen pro Ausführung, die im Mittel benötigte Zeit pro Ausführung, die Anzahl der vom Algorithmus getroffenen Fehlentscheidungen und die approximierte Zeitkomplexität. Außerdem ist das Verhältnis zwischen den im Mittel geprüften Variablenbelegungen und der Anzahl der existierenden Variablenbelegungen dargestellt.

4.6 Laufzeitanalyse der k-SAT Algorithmen bei zufälligen Formeln

Beschreibung des Experiments

Es wurden drei k-SAT Algorithmen getestet: die beiden deterministischen Algorithmen von Dantsin u.a. und Moser/Scheder und der randomisierte Algorithmus von Schöning. Die deterministischen Algorithmen verwenden einen Covering Code $\mathcal{C}_1$ für die Startbelegungen der lokalen Suche. Die Mächtigkeit $|\mathcal{C}_1|$ und der Radius r_1 des Codes sind in den entsprechenden Tabellen mit angegeben. Es wurden Formeln über n Variablen mit m Klauseln zufällig erzeugt, wobei m so gewählt wurde, dass die Wahrscheinlichkeit für die Erfüllbarkeit der Formel bei etwa $0{,}5$ liegt. Um die Zeitkomplexität der Algorithmen in Abhängigkeit von n bestimmen zu können, wurden für verschiedene n jeweils 400 (für größere n nur noch 200) Formeln erzeugt. Allerdings wurde bei den deterministischen Algorithmen n so gewählt, dass es durch den vom Algorithmus vorgegebenen Radius der lokalen Suche teilbar ist. Der Algorithmus von Dantsin u.a. verwendet bei 3-SAT einen Radius von 4 und der von Moser und Schder einen von 3. Wie bei den vorherigen Experimenten wurden auch hier die Messwerte arithmetisch gemittelt und zwischen erfüllbaren und unerfüllbaren Formeln und zwischen Laufzeit und Anzahl der getesteten Variablenbelegungen differenziert. Auf Grundlage der Mittelwerte sind dann Zeitkomplexitäten approximiert worden. Bei den Messwerten von Schönings Algorithmus ist weiterhin die Anzahl der Fehlentscheidungen angegeben, soweit die Formeln auch von deterministischen Algorithmen entschieden werden konnten. Der Algorithmus von Moser und Scheder wurde für $t = 3, 6, 9$ getestet.

Ergebnisse von Schönings Algorithmus

Die Messwerte sind der nachfolgenden Tabelle 4.4 zu entnehmen. Wie schon bei dem vorherigen Experiment mit Schönings Algorithmus, wächst die Laufzeit etwas stärker als die Anzahl der getesteten Variablenbelegungen. Auch hier könnte das darauf zurückzuführen sein, dass das Testen einer Variablenbelegung einen von m abhängigen linearen Zeitaufwand benötigt, aber nur in der Zeitmessung berücksichtigt wird. Weiterhin liegen die absoluten Werte der erfüllbaren Formeln deutlich unter denen der unerfüllbaren, wie auch deutlich an dem angegebenen Verhältnis der Anzahl der im Mittel getesteten Variablenbelegungen zwischen erfüllbaren und unerfüllbaren Formeln zu sehen ist. Dieses Verhältnis wächst mit steigendem n. Die theoretische Zeitkomplexität von $O(\text{poly}(n) \cdot 1{,}34^n)$ wird in den getesteten Bereichen bei den erfüllbaren Formeln noch unterschritten und bei den unerfüllbaren liegt sie etwas darüber.

Ergebnisse von u.a. Dantsins Algorithmus

Die Messwerte sind der nachfolgenden Tabelle 4.5 zu entnehmen. Die theoretische Zeitkomplexität von $O(\text{poly}(n) \cdot 1{,}5^n)$ wird nur bei erfüllbaren Formeln erreicht, bei den unerfüllbaren liegt sie darüber. Eine Erklärung dafür könnte sein, dass die Covering Codes für $n > 17$ nicht direkt vom dem aus Abschnitt 2.3 beschriebenen Greedy-Algorithmus konstruiert wurden, sondern, aufgrund der ungünstigen Zeitkomplexität des Greedy-Algorithmus, aus der direkten Summe zweier „kleinerer“ Covering Codes. Das verschlechtert allerdings die „Güte“ des Codes bezüglich der Mächtigkeit und daher sind die in den Fällen $n \leq 16$ näher am Minimum als die übrigen, wie auch in Abbildung 2.1 erkennbar ist. Es könnte also sein, dass Dantsins Algorithmus im Bereich $n \leq 16$ hier eine bessere Laufzeit aufgrund besserer Covering Codes hat, wodurch die Approximation der Zeitkomplexität verfälscht wird.

Ergebnisse von Mosers und Scheders Algorithmus

Die Messwerte sind der nachfolgenden Tabelle 4.6, 4.7 und 4.8 zu entnehmen. Auf den ersten Blick fällt auf, dass die approximierte Zeitkomplexität eher besser ist, wenn t groß ist, insbesondere wenn man die Anzahl der getesten Variablenbelegungen betrachtet. Auch die Differenz zwischen den approximierten Zeitkomplexitäten der gemessenen Zeit und der Anzahl der getesteten Belegungen scheint größer zu sein, wenn t groß ist. Weiterhin ist zu beachten, dass der Algorithmus im Fall $t = 9$ und bei erfüllbaren Formeln, die Anzahl der getesteten Belegungen am geringsten wächst und auch in absoluten Zahlen niedriger ist als bei den anderen beiden Varianten, aber was die gemessene Zeit betrifft, ist das Wachstum nicht wesentlich geringer und die absoluten Zahlen sind sogar deutlich höher.
Es ist zu beobachten, dass immer wenn vermehrt der Fall eintritt, dass $\geq t$ unerfüllte Klauseln in einer Formel vorhanden sind, die Laufzeit und insbesondere die Anzahl der getesteten Belegungen verstärkt ansteigt. Das ist auf den Abbildungen 4.2 und 4.3 jeweils an den Stellen $n = 9$ und $n = 24$ gut zu sehen. Bei der Variante mit $t = 9$ tritt der besagte Fall in verschwindent geringer Anzahl ein.
Nicht ausser Acht gelassen werden darf, dass der Algorithmus einen exponentiellen Faktor besitzt, der sich nicht zwangsläufig in gleichem Maße auf die Entwicklung der Anzahl der getesteten Belegungen, wie auf die Entwicklung der tatsächlich gemessenen Laufzeit auswirkt. Wenn in einer Formel weniger als t paarweise disjunkte, nicht erfüllte Klauseln vorhanden sind, werden für die Variablen dieser Klauseln alle möglichen Belegungen einmal angenommen und die Formel um diese reduziert. Dieser Vorgang wird häufig durch das Auftreten von z.B. leeren Klauseln vorzeitig abgebrochen, ohne dass die entsprechende Belegung auch getestet wurde.

k-SAT Algorithmus von Schöning

$k = 3$				erfüllbare Formeln			unerfüllbare Formeln			Verhältnis
n	m	2^n	Fehler	#	∅ Prüf.	∅ Zeit	#	∅ Prüf.	∅ Zeit	∅ Prüf.
4	24	64	5	183	10,7	0,2 ms	217	48	0,8ms	0,2229
8	42	256	2	194	45,0	1,0 ms	206	240	5,2ms	0,1875
12	57	4096	0	196	109,7	2,8 ms	204	1.152	28,1ms	0,0952
16	76	65.536	0	179	343,7	11,5 ms	221	4.800	155,3ms	0,0716
20	93	1.048.576	0	201	786,3	30,4 ms	199	18.960	708,9ms	0,0415
24	107	16.777.216	0	231	1.520,7	57,6 ms	169	71.784	2.671,2ms	0,0212
28	125	268.435.456	-	196	2.298,5	102,0 ms	204	264.600	11.341.9ms	0,0140
32	142	4.294.967.296	-	95	5.393,0	255,1 ms	105	955.680	42.132.2ms	0,0056
36	160	68.719.476.736	-	86	9.926,1	509,1 ms	114	3.398.004	170.324,5ms	0,0029
40	178	1.099.511.627.776	-	103	25.333,8	1.341,4 ms	97	11.932.560	645.615,6ms	0,0021
					$O(1{,}22^n)$	$O(1{,}25^n)$		$O(1{,}41^n)$	$O(1{,}45^n)$	

Tabelle 4.4: Ergebnisse des randomisierten k-SAT Algorithmus von Schöning mit einer theoretischen Laufzeit $O(\text{poly}(n) \cdot 1{,}34^n)$ und zufällig erzeugten Formeln in 3-KNF über n Variablen, mit m Klauseln. Zu sehen ist die Anzahl der getesteten Formeln, die Anzahl der im Mittel geprüften Variablenbelegungen pro Formel, die im Mittel benötigte Zeit pro Formel und die Anzahl der vom Algorithmus getroffenen Fehlentscheidungen. Außerdem ist das Verhältnis zwischen den im Mittel geprüften Variablenbelegungen für erfüllbare und unerfüllbare Formeln angegeben.

k-SAT Algorithmus von Dantsin u.a.

Formeln			Code		erfüllbare Formeln			unerfüllbare Formeln		
n	m	2^n	$\|\mathcal{C}_1\|$	r_1	#	∅ Prüf.	∅ Zeit	#	∅ Prüf.	∅ Zeit
4	24	64	4	1	188	6,5	0,9 ms	212	16,0	2,2 ms
8	42	256	16	2	196	36,2	8,6 ms	204	181,5	43,6 ms
12	57	4.096	32	3	196	159,3	46,2 ms	204	922,1	265,8 ms
16	76	65.536	100	4	179	703,3	290,2 ms	221	7.028,6	2.903,4 ms
20	93	1.048.576	256	5	201	6.405,5	3.290,7 ms	199	42.927,5	22.235,3 ms
24	107	16.777.216	896	6	231	25.924,8	15.662,2 ms	169	361.233,1	220.744,6 ms
28	125	268.435.456	2816	7	196	79.106,5	55.076.6 ms	204	-	-
32	142	4.294.967.296	9856	8	95	398.517,2	316.718.4 ms	105	-	-
						$O(1{,}49^n)$	$O(1{,}58^n)$		$O(1{,}63^n)$	$O(1{,}75^n)$

Tabelle 4.5: Ergebnisse des deterministischen k-SAT Algorithmus von Dantsin u.a. mit einer theoretischen Laufzeit $O(\text{poly}(n) \cdot 1{,}5^n)$.

k-SAT Algorithmus von Moser und Scheder

mit $\mathcal{C}_2$ der Länge $t = 3$ und $r_2 = 1$

Formeln			Code		erfüllbare Formeln			unerfüllbare Formeln		
n	m	2^n	$\|\mathcal{C}_1\|$	r_1	Anz.	∅ Prüf.	∅ Zeit	Anz.	∅ Prüf.	∅ Zeit
6	30	64	4	2	199	7,9	2,8 ms	201	40,9	20,4 ms
9	44	512	8	3	201	12,0	10,0 ms	199	195,4	177,1 ms
12	57	4.096	16	4	224	52,1	32,0 ms	176	4419,5	2.804,8 ms
15	70	32.786	24	5	199	446,3	251,1 ms	201	89.937,6	53.764,3 ms
$18'$	82	262.144	32	6	209	2514,0	1.436,8 ms	191	1.080.555,8	632.238,1 ms
21	96	2.097.152	64	7	215	8.754,8	6.116,2 ms	185	-	-
24	107	16.777.216	112	8	230	28.346,7	20.175,3 ms	170	-	-
$27'$	121	134.217.728	200	9	207	91.958,6	72.130,3 ms	193	-	-
$30'$	133	1.073.741.824	352	10	218	192.988,1	138.665,2 ms	182	-	-
						$O(1{,}58^n)$	$O(1{,}61^n)$		$O(2{,}42^n)$	$O(2{,}41^n)$

Tabelle 4.6: Ergebnisse des deterministischen k-SAT Algorithmus von Moser/Scheder mit einer theoretischen Laufzeit $O(\text{poly}(n) \cdot 1{,}34^n)$.

k-SAT Algorithmus von Moser und Scheder

mit $\mathcal{C}_2$ der Länge $t = 6$ und $r_2 = 2$

Formeln			Code		erfüllbare Formeln			unerfüllbare Formeln		
n	m	2^n	$\|\mathcal{C}_1\|$	r_1	Anz.	∅ Prüf.	∅ Zeit	Anz.	∅ Prüf.	∅ Zeit
6	30	64	4	2	195	6,6	2,8 ms	205	41,1	23,0 ms
9	44	512	8	3	195	13,3	14,3 ms	205	174,1	209,8 ms
12	57	4.096	16	4	215	28,7	57,2 ms	185	909,0	1.962,6 ms
15	70	32.786	24	5	211	67,7	218,8 ms	189	3.191,9	12.367,5 ms
18	82	262.144	32	6	217	140,7	751,1 ms	182	10.405,8	64.116,3 ms
21	96	2.097.152	64	7	97	302,9	2.773,0 ms	103	57.014,3	529.716,8 ms
24	107	16.777.216	112	8	212	730,1	7.061,6 ms	188	-	-
27	121	134.217.728	200	9	186	4.800,1	31.474,1 ms	214	-	-
30	133	1.073.741.824	352	10	87	15.081,9	75.801,5 ms	113	-	-
						$O(1{,}36^n)$	$O(1{,}53^n)$		$O(1{,}61^n)$	$O(1{,}93^n)$

Tabelle 4.7: Ergebnisse des deterministischen k-SAT Algorithmus von Moser/Scheder mit einer theoretischen Laufzeit $O(\text{poly}(n) \cdot 1{,}34^n)$.

k-SAT Algorithmus von Moser und Scheder

mit $\mathcal{C}_2$ der Länge $t = 9$ und $r_2 = 3$

Formeln			Code		erfüllbare Formeln			unerfüllbare Formeln		
n	m	2^n	$\|\mathcal{C}\|$	r	Anz.	∅ Prüf.	∅ Zeit	Anz.	∅ Prüf.	∅ Zeit
6	30	64	4	2	202	7,3	2,7 ms	198	41,4	19,1 ms
9	44	512	8	3	213	12,9	11,3 ms	187	176,3	184,3 ms
12	57	4.096	16	4	196	28,2	62,0 ms	204	905,6	1.761,1 ms
15	70	32.786	24	5	181	60,8	213,4 ms	219	3.198,1	11.324,5 ms
18	82	262.144	32	6	209	133,5	950,2 ms	191	10.377,6	68.045,1 ms
21	96	2.097.152	64	7	204	298,2	3.166,6 ms	196	48.653,7	527.047,3 ms
24	107	16.777.216	112	8	231	685,2	13.623,7 ms	169	-	-
27	121	134.217.728	200	9	199	1.473,9	49.062,8 ms	201	-	-
30	133	1.073.741.824	352	10	106	2.935,8	296.784,2 ms	94	-	-
						$O(1{,}29^n)$	$O(1{,}60^n)$		$O(1{,}59^n)$	$O(1{,}96^n)$

Tabelle 4.8: Ergebnisse des deterministischen k-SAT Algorithmus von Moser/Scheder mit einer theoretischen Laufzeit $O(\text{poly}(n) \cdot 1{,}34^n)$.

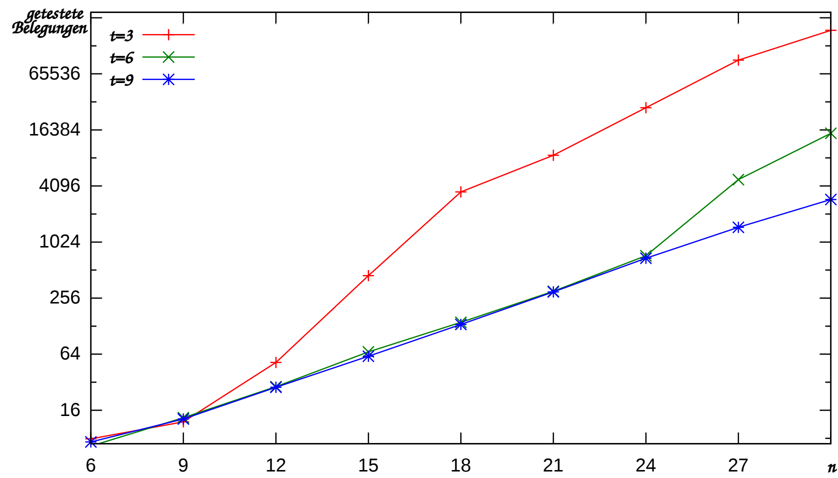

Abbildung 4.2: Laufzeitentwicklung der Varianten von Moser und Scheders k-SAT Algorithmus bei erfüllbaren Formeln.

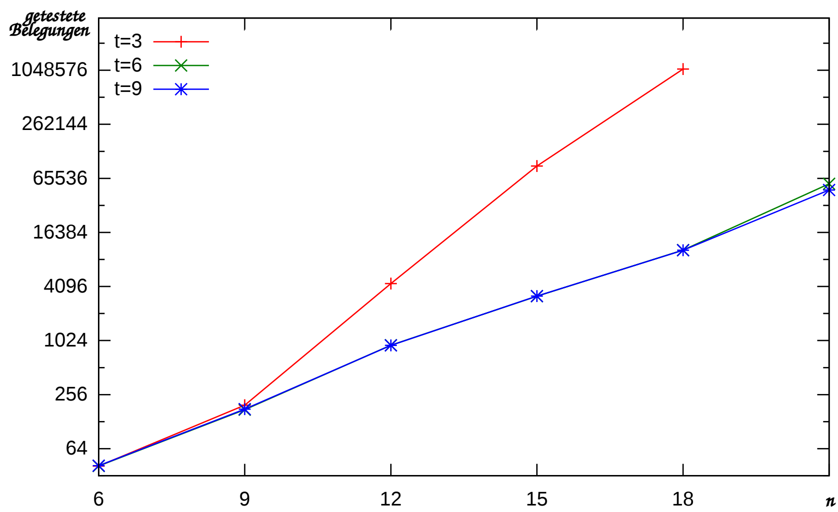

Abbildung 4.3: Laufzeitentwicklung der Varianten von Moser und Scheders k-SAT Algorithmus bei unerfüllbaren Formeln.

4.7 Zusammenfassung der Ergebnisse

Betrachten wir die Ergebnisse für unerfüllbare Formeln, dann liefert der Algorithmus von Schöning mit deutlichen Abstand die besten Resultate (siehe Abbildung 4.4). Das gilt aber nicht nur für die Werte der jeweiligen Anzahl der getesteten Variablenbelegungen, sondern auch für tatsächlichen Laufzeiten, wie den Tabellen zu entnehmen ist. Wenn wir allerdings die Werte bei den erfüllbaren Formeln betrachten, ist es nicht mehr so deutlich (siehe Abbildung 4.5). Zwar liefert, was das Wachstum der Anzahl der getesteten Variablenbelegungen und auch die tatsächliche Laufzeit betrifft, der Algorithmus von Schöning auch hier die besten Ergebnisse, aber der Algorithmus von Moser und Scheder kommt je nach Variante recht nah heran. Das gilt allerdings nicht für die tatsächlich gemessenen Zeiten, vermutlich fallen polynomiellen Berechnungungen, die bei Mosers und Scheders Algorithmus deutlich höher sein dürften, noch stark ins Gewicht.

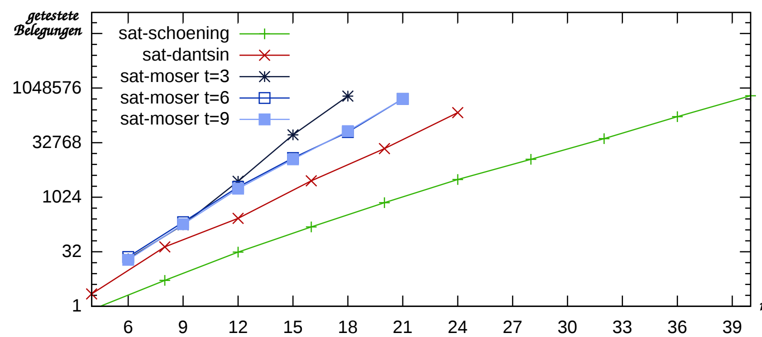

Abbildung 4.4: Laufzeitentwicklung der k-SAT Algorithmen bei unerfüllbaren Formeln.

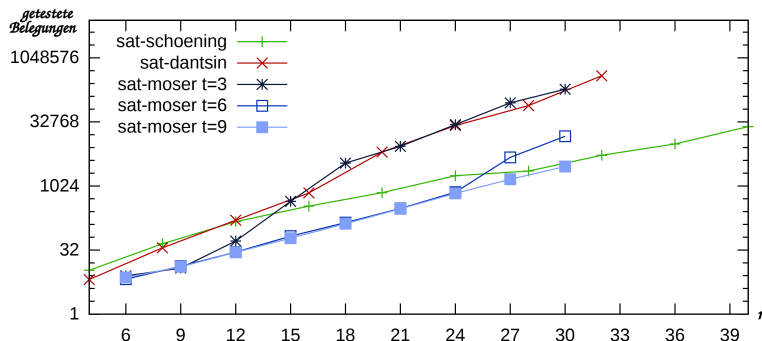

Abbildung 4.5: Laufzeitentwicklung der k-SAT Algorithmen bei erfüllbaren Formeln.

Kapitel 5

Zusammenfassung

Der k-SAT Algorithmus von Schöning ist nicht nur sehr einfach implementierbar, er hat in den Experimenten auch die besten Ergebnisse erzielt, was die approximierte Zeitkomplexität und absoluten Zahlen der Zeitmessungen und bei unerfüllbaren Formeln auch, was die Anzahl der getesteten Belegungen betrifft. Nachteilig ist allerdings, das der Algorithmus randomisiert ist und mit einer zwar geringen aber dennoch vorhandenen Wahrscheinlichkeit k-SAT falsch entscheidet.
Die determinstischen Algorithmen von Dantsin u.a. und Moser und Scheder sind dagegen relativ schwierig zu implementieren, auch weil sich um die Konstruktion von Covering Codes gekümmert werden muss. Die Anzahl der getesteten Variablenbelegungen sind zum Teil besser, aber liegen in einigen Fällen auch deutlich über Schönings Algorithmus. Die Differenz des approximierten Wachstums bei der gemessenen Zeit und der Anzahl der getesteten Belegungen bei den jeweiligen Algorithmen lässt vermuten, dass das polynomielle „Beiwerk“ der deterministischen Algorithmen (insbesondere bei dem von Moser und Scheder) noch einen wesentlichen Faktor darstellt. Die Optimierung dieser polynomiellen Berechnungen wurde bei der Implementation nicht gesondert beachtet und man muss daher davon ausgehen, das hier noch Potenzial vorhanden ist.
Die Untersuchungen zeigen, dass das Wachstum der Laufzeit des Algorithmus von Moser und Scheder an einer bestimmten Stelle, die offenbar von t abhängt, einen Sprung macht. In weiteren Experimenten könnte man diese Stellen mit einer kleineren Schrittweite für n genauer untersuchen und so möglicherweise eine Regelmäßigkeit in Abhängigkeit von t formulieren.

Anhang A

Quelltexte

A.1 Bibliothek: code.py

Hier sind Funktionen zu finden, die das Konstruieren von Covering Codes betreffen. Die Funktion `gen` entspricht dem in dieser Arbeit vorgestellten Greedy-Algorithmus und `directsum` der direkten Summe.

```
import random
import math

d = [0, 1, 1, 2, 1, 2, 2, ..., 7, 8, 8, 9, 8, 9, 9, 10]

def distance(x,y):
    return sum(b == '1' for b in bin(x^y)[2:])

def distance_fast(x,y):
    v = x^y
    return d[v&1023] + d[(v>>10)&1023]

def distance_slow(v,w):
    d = 0
    for i in range(len(v)):
        d += 1 if v[i] != w[i] else 0
    return d

def space(k,n,L=[[]]):
    A = range(k)
    if n == 0:
        return L
    else:
        R = []
        for w in L:
            for a in A:
                R.append([a] + w)
        return space(k,n-1,R)

def factorial(n):
    result = 1
    while n:
        result = result * n
        n = n - 1
    return result

def binom(n,k):
    return float(factorial(n)) / float(float(factorial(k)) * float(factorial(n - k)))

```

```
40 def maxvol(q,n,r):
41     return sum([binom(n,i) for i in range(r+1)]) * (q-1)**r
42
43 def cov_exist_size(q,n,r):
44     if q < 2:
45         return None
46     if q == 2:
47         return int(math.ceil((n * 2**n) / maxvol(2,n,r)))
48     else:
49         return int(math.ceil((n * math.log(q) * q**n) / (binom(n,r) * (q-1)**r)))
50
51 def bin_seq(S):
52     for e in S: print bin(e)
53
54 def cov_test(q,C,r):
55     if len(C) == 0:
56         return None
57     n = len(C[0])
58     for w in space(q,n):
59         for c in C:
60             cov = False
61             if distance_slow(w,c) <= r:
62                 cov = True
63                 break
64         if not cov:
65             return False
66     return True
67
68 def cov_rand(q,n,r):
69     C = list()
70     s = cov_exist_size(q,n,r)
71     for _ in range(s/4):
72         C.append([random.randint(0,q-1) for _ in range(n)])
73     return C
74
75 def gen(n,r):
76     C = []
77     vol = maxvol(2,n,r)
78     H = [[vol,set()] for _ in range(2**n)]
79     R = set(range(2**n))
80     S = list(R)
81     S.sort()
82     D = set()
83     while (True):
84         max_vol = 0
85         max_w = 0
86         for i in S:
87             if H[i][0] > max_vol:
88                 max_vol = H[i][0]
89                 max_w = i
90                 if max_vol == vol: break
91         if max_vol == 0:
92             return C
93         else:
94             C.append(max_w)
95         D.clear()
96         for i in S:
97             if distance_fast(i,max_w) <= r:
98                 D.add(i)
99                 H[i][0] = 0
100                H[i][1].clear()
101        R = R - D
102        S = list(R)
103        S.sort()
```

```
104         for i in S:
105             for j in D:
106                 if H[i][0] > 0 and distance_fast(i,j) <= r and j not in H[i][1]:
107                     if H[i][0] == 1:
108                         H[i][0] = 0
109                         H[i][1].clear()
110                         D.remove(i)
111                     else:
112                         H[i][0] -= 1
113                         H[i][1].add(j)
114 
115 
116 def fromstring(s):
117     S = s.splitlines()
118     C = []
119     k,n,r,l = S[0].split()
120     del S[0]
121     for line in S:
122         C.append([int(x) for x in line.split()])
123     return int(k),int(n),int(r),int(l),C
124 
125 def readfile(filename):
126     s = open(filename,'r').read()
127     s = s.split("#\n")
128     del s[0]
129     codes = {}
130     for x in s:
131         k,n,r,l,C = fromstring(x)
132         codes[(k,n,r)] = C
133     return codes
134 
135 def directsum(C1,C2):
136     C = list()
137     for c1 in C1:
138         for c2 in C2:
139             C.append(c1+c2)
140     return C
141 
142 def getcode(k,n,r):
143     if r == 0:
144         return space(k,n)
145     if float(n)/r <= 2:
146         return [[0]*n,[1]*n]
147     if k == 2 and n <= 17:
148         return codes[(2,n,r)]
149     if k == 3 and n <= 9:
150         return codes[(3,n,r)]
151     if k == 2 and n == 18 and r == 6:
152         return directsum(codes[(2,11,5)],codes[(2,7,1)])
153     if k == 2 and n == 20 and r == 5:
154         return directsum(codes[(2,7,1)],codes[(2,13,4)])
155     if k == 2 and n == 20 and r == 7:
156         return directsum(codes[(2,7,3)],codes[(2,13,4)])
157     if k == 2 and n == 21 and r == 7:
158         return directsum(codes[(2,13,4)],codes[(2,8,3)])
159     if k == 2 and n == 24 and r == 6:
160         return directsum(codes[(2,7,1)],codes[(2,17,5)])
161     if k == 2 and n == 24 and r == 8:
162         return directsum(codes[(2,9,4)],codes[(2,15,4)])
163     if k == 2 and n == 27 and r == 9:
164         return directsum(codes[(2,11,5)],codes[(2,16,4)])
165     if k == 2 and n == 28 and r == 7:
166         return directsum(codes[(2,17,4)],codes[(2,11,3)])
167     if k == 2 and n == 28 and r == 9:
```

```
        return directsum(codes[(2,11,5)],codes[(2,17,4)])
    if k == 2 and n == 30 and r == 10:
        return directsum(codes[(2,17,4)],codes[(2,13,6)])
    if k == 2 and n == 32 and r == 8:
        return directsum(codes[(2,17,4)],codes[(2,15,4)])
    if k == 2 and n == 32 and r == 11:
        return directsum(codes[(2,16,7)],codes[(2,16,4)])

codes = readfile("coveringcodes.dat")
```

A.2 Bibliothek: formula.py

Es wird die aussagenlogische Klausel und Formel als Klasse implementiert. Die in der Arbeit vorgestellten Algorithmen `flip` und `reduce` werden hier ebenfalls implementiert.

```
from random import randint,sample,shuffle,choice
from copy import copy
from string import zfill

class Clause:
    def __init__(self,literals):
        self.literals = list(literals)
        self.variables = set([abs(u) for u in literals])

    def satisfied(self,b):
        for u in self.literals:
            v = abs(u)
            if (u < 0) == (b[v-1] == 0): return True
        return False

    def lit(self):
        return self.literals

    def var(self):
        return self.variables

    def remove(self,u):
        if len(self.literals) > 1:
            self.literals.remove(u)
            self.variables.remove(abs(u))
            return True
        else:
            return False

    def copy(self):
        return Clause(copy(self.literals))

    def k(self):
        return len(self.literals)


class Formula:
    def __init__(self):
        self.variables = set()
        self.clauses = list()

    def add_clause(self,C):
        self.clauses.append(C)
        self.variables.update(C.var())
```

```
45
46      def fill_randomly(self,k,n,m):
47          self.variables = set()
48          self.clauses = list()
49          pop = range(1,n+1)
50          while len(self.clauses) < m:
51              C = set([u*choice([1,-1]) for u in sample(pop,k)])
52              if C not in self.clauses:
53                  self.clauses.append(C)
54          for C in self.clauses:
55              self.variables.update([abs(u) for u in C])
56          self.clauses = [Clause(C) for C in self.clauses]
57
58      def load_from_file(self,filename):
59          self.variables = set()
60          self.clauses = list()
61          f = open(filename, "r")
62          while True:
63              line = f.readline()
64              if len(line) > 0 and line[0] == "p":
65                  S = line.split()
66                  m = int(S[3])
67                  break
68          for _ in range(m):
69              line = f.readline()
70              C = [int(s) for s in line.split()]
71              self.variables.update([abs(c) for c in C])
72              self.clauses.append(Clause(C))
73
74      def shuffle(self): shuffle(self.clauses)
75
76      def get_unsat_clause(self,b):
77          for C in self.clauses:
78              if not C.satisfied(b): return C
79          return None
80
81      def get_unsat_clauses(self,b,k,t):
82          G = list()
83          Gvar = set()
84          for C in self.clauses:
85              if not C.satisfied(b):
86                  Cvar = C.var()
87                  if (len(Cvar) == k) and (len(Gvar&Cvar) == 0):
88                      G.append(C)
89                      Gvar.update(C.var())
90              #if (len(G) >= t): break
91          return G
92
93      def reduce(self,var,val):
94          clauses = list()
95          for C in self.clauses:
96              if var in C.var():
97                  u = var if var in C.lit() else -var
98                  if (u<0) == (val==1):
99                      if not C.remove(u): return False
100                     clauses.append(C)
101             else:
102                 clauses.append(C)
103         self.clauses = clauses
104         self.variables.remove(var)
105         return True
106
107     def copy(self):
108         F = Formula()
```

```
109         for C in self.clauses:
110             F.add_clause(C.copy())
111         return F
112 
113     def n(self):
114         return max(self.variables)
115 
116     def k(self):
117         return max([C.k() for C in self.clauses])
118 
119 def random_assignment(n):
120     return [randint(0,1) for _ in range(n)]
121 
122 def flip(H,b,w):
123     c = copy(b)
124     i = 0
125     for C in H:
126         lits = C.lit()
127         u = lits[w[i]]
128         v = abs(u)-1
129         c[v] = 1-c[v]
130     return c
131 
132 def i2a(i,n):
133     l = bin(i)
134     l = l[2:]
135     l = zfill(l,n)
136     l = [int(i) for i in l]
137     return l
```

A.3 Bibliothek: localsearch.py

Die Implementation der beiden deterministischen Algorithmen für die lokale Suche.

```
1  from copy import copy
2  from formula import i2a, flip
3  from math import ceil
4  
5  def sb(F,b,r,counter=0):
6      counter += 1
7      C = F.get_unsat_clause(b)
8      if C == None: return b,counter
9      if r <= 0: return None,counter
10     for u in C.lit():
11         v = abs(u)
12         c = copy(b)
13         c[v-1] = 1 - c[v-1]
14         G = F.copy()
15         if G.reduce(v,c[v-1]):
16             d,counter = sb(G,c,r-1,counter)
17             if d != None: return d,counter
18     return None,counter
19 
20 def sb_fast(k,F,b,r,C,counter=0):
21     counter += 1
22     t = len(C[0])
23     H = F.get_unsat_clauses(b,k,t)
24     if len(H) == 0: return b,counter
25     if r <= 0: return None,counter
26     if len(H) < t:
```

```
27          Hvar = set()
28          for K in H: Hvar.update(set(K.var()))
29          Hvar = list(Hvar)
30          for i in xrange(2**len(Hvar)):
31              G = F.copy()
32              a = i2a(i,len(Hvar))
33              c = copy(b)
34              for j in xrange(len(Hvar)):
35                  if not G.reduce(Hvar[j],a[j]):
36                      done = False
37                      break
38                  c[Hvar[j]-1] = a[j]
39                  done = True
40              if done:
41                  d,counter = sb(G,c,int(ceil(r)),counter)
42                  if d != None: return d,counter
43          return None,counter
44      else:
45          for w in C:
46              c = flip(H,b,w)
47              s = r - (float(t) - ((2.0*float(t)) / float(k)))
48              d,counter = sb_fast(k,F,c,s,C,counter)
49              if d != None: return d,counter
50          return None,counter
```

A.4 Bibliothek: sat.py

Die Implementation der vorgestellten k-SAT Algorithmen und zusätzlich der Bruteforce Algorithmus.

```
1  from localsearch import sb,sb_fast
2  from code import getcode
3  from math import ceil
4  from formula import random_assignment,i2a
5  from random import randint
6
7  def dantsin(F):
8      r = int(round(float(F.n())/float(F.k()+1)))
9      C = getcode(2,F.n(),r)
10     counter = 0
11     for b in C:
12         e,c = sb(F,b,r)
13         counter += c
14         if e != None: return e,counter
15     return None,counter
16
17 def moser(F,t=3):
18     k = F.k()
19     r = int(round(float(F.n())/float(k)))
20     C1 = getcode(2,F.n(),r)
21     C2 = getcode(k,t,t/k)
22     counter = 0
23     for b in C1:
24         a,c = sb_fast(k,F,b,float(r),C2)
25         counter += c
26         if a != None: return a,counter
27     return None,counter
28
29 def schoening(F):
```

```
30      n = F.n()
31      k = F.k()
32      t = int(ceil((2.0*(1.0 - 1.0/k))**n))
33      counter = 0
34      for _ in xrange(t):
35          b = random_assignment(n)
36          F.shuffle()
37          for _ in range(3*n):
38              counter += 1
39              C = F.get_unsat_clause(b)
40              if C == None: return b,counter
41              i = randint(0,C.k()-1)
42              v = abs(C.lit()[i])-1
43              b[v] = 1-b[v]
44      return None,counter
45
46  def bruteforce(F):
47      for i in range(2**F.n()):
48          b = i2a(i,F.n())
49          s,_ = F.satisfied1(b)
50          if s: return True,b,i
51      return False,None,2**F.n()
```

Anhang B

Formel in 3-KNF

Nachfolgend eine aussagenlogische Formel in konjunktiver Normalform mit 12 Variablen, 40 Klauseln mit je drei Literalen und genau einer erfüllenden Belegung. Sie stammt aus den Vorlesungsunterlagen von Prof. Reischuk.

$$
\begin{aligned}
&(x_1 \lor x_2 \lor x_9) \land (x_5 \lor x_6 \lor x_7) \land (x_3 \lor x_4 \lor x_{10}) \land (x_3 \lor x_9 \lor x_{12}) \land \\
&(\overline{x}_3 \lor \overline{x}_8 \lor \overline{x}_{11}) \land (\overline{x}_1 \lor \overline{x}_5 \lor \overline{x}_{10}) \land (x_1 \lor x_6 \lor \overline{x}_{11}) \land (\overline{x}_1 \lor x_7 \lor x_{11}) \land \\
&(\overline{x}_2 \lor \overline{x}_5 \lor x_9) \land (x_2 \lor x_8 \lor \overline{x}_9) \land (x_1 \lor \overline{x}_3 \lor \overline{x}_5) \land (\overline{x}_8 \lor \overline{x}_{10} \lor x_{11}) \land \\
&(\overline{x}_2 \lor \overline{x}_8 \lor x_{10}) \land (x_3 \lor x_5 \lor \overline{x}_9) \land (\overline{x}_1 \lor \overline{x}_9 \lor x_{12}) \land (\overline{x}_2 \lor \overline{x}_3 \lor \overline{x}_6) \land \\
&(x_2 \lor x_{10} \lor x_{11}) \land (\overline{x}_2 \lor \overline{x}_4 \lor \overline{x}_5) \land (\overline{x}_8 \lor \overline{x}_9 \lor \overline{x}_{12}) \land (x_4 \lor \overline{x}_7 \lor x_9) \land \\
&(x_3 \lor x_5 \lor \overline{x}_8) \land (\overline{x}_5 \lor x_8 \lor x_{10}) \land (\overline{x}_4 \lor \overline{x}_6 \lor x_7) \land (x_6 \lor \overline{x}_7 \lor \overline{x}_{12}) \land \\
&(\overline{x}_4 \lor x_5 \lor \overline{x}_7) \land (\overline{x}_6 \lor x_8 \lor \overline{x}_{12}) \land (x_2 \lor \overline{x}_6 \lor x_{12}) \land (x_8 \lor x_{11} \lor x_{12}) \land \\
&(\overline{x}_7 \lor \overline{x}_9 \lor \overline{x}_{10}) \land (x_6 \lor x_7 \lor x_8) \land (\overline{x}_4 \lor \overline{x}_6 \lor \overline{x}_{11}) \land (\overline{x}_1 \lor \overline{x}_7 \lor \overline{x}_{12}) \land \\
&(x_7 \lor \overline{x}_{10} \lor x_{11}) \land (x_4 \lor x_9 \lor \overline{x}_{12}) \land (x_2 \lor \overline{x}_4 \lor x_{12}) \land (\overline{x}_3 \lor x_{10} \lor \overline{x}_{11}) \land \\
&(\overline{x}_1 \lor x_3 \lor \overline{x}_{10}) \land (\overline{x}_2 \lor x_4 \lor \overline{x}_{11}) \land (x_1 \lor x_4 \lor x_5) \land (x_1 \lor \overline{x}_3 \lor x_6)
\end{aligned}
$$

Die erfüllende Belegung dieser Formel ist:

$$
\begin{array}{lll}
x_1 = 1 & x_5 = 1 & x_9 = 0 \\
x_2 = 0 & x_6 = 0 & x_{10} = 0 \\
x_3 = 0 & x_7 = 0 & x_{11} = 1 \\
x_4 = 1 & x_8 = 1 & x_{12} = 1
\end{array}
$$

Anhang C

Literaturverzeichnis

[1] V. Chvatal. A greedy heuristic for set-covering problem. In *Mathmatics of Operations Research Vol. 4 No. 3*, page 233. 1979.

[2] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan J., Kleinberg C., Papadimitriou O., Raghavan, and U. Schoening. A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. In *Theoretical Computer Science 289*, pages 69–83. 2002.

[3] Gerzson Kéri. Tables for bounds on covering codes, 2011. `http://www.sztaki.hu/~keri/codes/index.htm`.

[4] R. A. Moser and D. Scheder. A Full Derandomization of Schöning's k-SAT Algorithm. In *Proceedings of the 43rd annual ACM symposium on Theory of computing*, pages 245–252. 2011.

[5] R. Reischuk. Codierung und Sicherheit (Vorlesungsskript), 2011/2012.

[6] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 410. 1999.

[7] U. Schöning. *Algorithmik*. Spektrum Akademischer Verlag, 2001.

[8] E. Welzl. Boolean satisfiability – combinatorics and algorithms (lecture notes), 2005. `http://www.inf.ethz.ch/~emo/SmallPieces/SAT.ps`.