Bachelorarbeit

SAT-Algorithmen

Eine Laufzeitanalyse von deterministischen und randomisierten Exponentialzeit-Algorithmen

Arne Meier

29. August 2005

Universität Hannover
Institut für Informationssysteme
Fachgebiet Theoretische Informatik

Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Inhaltsverzeichnis

1 Einleitung

Als der typische Vertreter der Klasse $\mathcal{NP}$ kommt SAT (für Satisfiability), das Erfüllbarkeitsproblem für aussagenlogische Formeln, eine besondere Rolle in der Komplexitätstheorie zu. Die $\mathcal{NP}$-Vollständigkeit von SAT wurde 1971 von Stephen Cook in [Coo71] gezeigt. Es gilt $\mathcal{P} \neq \mathcal{NP}$ genau dann, wenn es keine deterministischen, in Polynomialzeit laufenden Algorithmen gibt, die bei Eingabe einer aussagenlogischen Formel ausgeben, ob diese erfüllbar ist oder nicht. Um den Aufwand zur Problemlösung zu reduzieren, sind „gute“ Exponentialzeitalgorithmen erforderlich.

Man kann ohne Bedenken die Eingabe auf eine Formel in konjunktiver Normalform mit maximal drei Literalen pro Klausel reduzieren, da jede Formel mit polynomiellen Aufwand in einen äquivalenten 3-KNF-Ausdruck umgeformt werden kann (siehe hierzu [HMU02]).

Bei der Suche nach einer Lösung wird ein bestimmtes Schema verfolgt. Der wohl einfachste, aber auch am längsten dauernde Weg ist das Ausprobieren aller Belegungen mit sofortigem Abbruch beim Finden einer erfüllenden Belegung der eingegebenen Formel. Dies ist ein sogenannter Brute-Force-Algorithmus. Im schlechtesten Fall gibt es keine erfüllende Belegung und folglich werden alle Möglichkeiten durchgetestet. Hier beträgt die Laufzeit bei n Variablen und damit 2^n verschiedenen Belegungen $O(2^n)$.

Es gibt jedoch bessere Strategien als das Ausprobieren aller Möglichkeiten. Der deterministische Algorithmus BSAT (Algorithmus 3.2) aus [MS85] arbeitet beispielsweise mit einer Laufzeit von $O(1{,}6181^n)$. Dieser „verkleinert“ in rekursiven Schritten die Formel bis ein sogenannter Trivialfall eintritt. Dieses ist zum einen der Fall, wenn die leere Klausel in der Formel vorkommt (dann ist die Formel nicht erfüllbar). Zum anderen ist es der Fall, wenn die Formel die leere Menge ist (die Formel ist dann erfüllbar). In den rekursiven Schritten wird überprüft, ob eine speziell gewählte Belegung erfüllend ist. Anschließend werden Veränderungen an Klauseln vorgenommen und teilweise Klauseln gestrichen, die aus dem logischen Zusammenhang nicht mehr relevant für das Ergebnis sind.

Randomisierte Algorithmen hingegen können im Mittel bessere Laufzeiten als deterministische Algorithmen erreichen; momentan in Bereichen von unter $O(1{,}4^n)$. In [Sch03] werden drei typische Schritte eines randomisierten Algorithmus' beschrieben:

(i) Man konstruiert eine Lösung, indem man eine zufällige Belegung für die zu testende Formel wählt.

(ii) Man modifiziert die Lösung an den Stellen, die eine Klausel zu „nicht erfüllbar“ evaluiert hat. Hierbei verändert man die Belegung einer der Klausel-Variablen damit die Klausel nun erfüllt ist.

(iii) Schließlich versucht man diese veränderte Lösung auf einen effizient lösbaren Spezialfall zu reduzieren.

Ein durch die randomisierten Algorithmen neu entstandenes Problem ist das Erzeugen von Zufallszahlen. Algorithmisch, das heißt deterministisch, also ohne eine Quelle „echten“ Zufalls, gibt es keine Möglichkeit, Zufallszahlen zu erzeugen wie sie in der Stochastik als Zufallsvariablen betrachtet werden. Bekommt der Algorithmus die selben Initialwerte, so ist der Ablauf immer der gleiche. Als „echte“ Zufallszahlen sind solche zu verstehen, die nicht durch einen Algorithmus sondern durch einen stochastischen, physikalischen Prozess erzeugt wurden. Man nimmt an, solche Prozesse sind zum Beispiel der radioaktive Zerfall von Atomkernen oder elektrisches Rauschen.

Die Resultate der randomisierten Algorithmen hängen von der Qualität der Zufallszahlen-Generatoren ab. Je weniger „zufällig“ die erzeugten Zahlen sind, desto größer ist in der Regel die Abweichung von der theoretischen Laufzeit.

In der vorliegenden Arbeit werden die in [MS85] und [DGHS00] vorgestellten deterministischen Algorithmen, die in [PPSZ98] und [HSSW02] vorgestellten randomisierten Algorithmen, sowie ein Brute-Force Algorithmus vorgestellt, implementiert und anschließend Laufzeitanalysen unterzogen. Abschließend wird ein Vergleich mit den theoretisch bestimmten Werten, sowie untereinander vorgenommen. Die Beschreibung von Methoden zur Erzeugung von Zufallszahlen wird kein Bestandteil dieser Arbeit sein, siehe hierfür etwa [Knu98].

2 Allgemeine Begrifflichkeiten

Zu Beginn dieser Arbeit werden wir einige notwendige Notationen und Begriffe einführen, die sich an den Standardwerken [EFT98] und [Rau02] orientieren. Wir verwenden den Begriff *Literal* für eine aussagenlogische Variable oder ihre Negation, wobei eine Negation mittels $\overline{x}$ ausgedrückt wird. In der folgenden Tabelle werden einige gängige Junktoren vorgestellt.

Symbol	**Bedeutung**
$\vee$	„oder“, Disjunktion
$\wedge$	„und“, Konjunktion
$\overline{x}$	„nicht“, Negation
$\rightarrow$	Implikation

Tabelle 2.1: Junktoren

Wir sprechen von einer *Formel* (engl. *formula*), wenn die Verknüpfung von Literalen mit diesen Junktoren gemeint ist. Ohne Beschränkung der Allgemeinheit setzen wir voraus, dass Formeln immer nur die Variablen $x_1, \ldots, x_n$ enthalten (für ein geeignetes $n \in \mathbb{N}$), sonst wird eine Umbenennung durchgeführt. Es sei $V_n := \{x_i \mid 1 \leq i \leq n\}$ die Menge der enthaltenen Variablen. Sei L_n die Menge aller Literale über V_n, also $L_n := \{\overline{x} \mid x_i \in V_n\} \cup V_n$.

Eine *Klausel* (engl. *clause*) ist die Disjunktion von Literalen (statt $C = L_1 \vee \cdots \vee L_m$ verwenden wir die Mengen-Schreibweise $C = \{L_1, \ldots, L_m\}$. Dabei gehen wir davon aus, daß kein Literal in einer Klausel mehrfach auftritt. Analog zur Mengen-Schreibweise bezeichne die Länge einer Klausel C die Anzahl ihrer Literale (hierzu schreiben wir $|C|$). Für die *leere Klausel* schreiben wir $\square$. Eine *Einheitsklausel* (engl. *unit clause*) ist eine Klausel mit genau einem Literal. Eine Formel ist in *konjunktiver Normalform* (kurz KNF), wenn sie eine Konjunktion von Klauseln ist, also von der Form $\bigwedge_{i=1}^{n} \bigvee_{k=1}^{m_i} L_{i,k}$. Auch hier verwenden wir analog die oben eingeführte Mengen-Schreibweise, eine Formel in KNF ist also eine Menge von Literalmengen. Eine Formel in KNF mit einer Klauselläng von maximal k Literalen wird als k-KNF-Formel bezeichnet.

Gemäß [HMU02, Abschnitte 10.3.2–10.3.4] kann jede Formel mit polynomiellen Zeitaufwand in einen äquivalenten 3-KNF-Ausdruck umgeformt werden. Daher gehen wir im weiteren davon aus, dass uns alle Formeln in 3-KNF vorliegen.

Mit Wahrheitswerten sind im folgenden die Bezeichner *falsch* und *wahr* für die Elemente der Menge $\{0, 1\}$ gemeint. Sei $\mathbb{N} = \{0, 1, \ldots\}$ die Menge der natürlichen Zahlen. Für

$n \in \mathbb{N}$ ist eine *Belegung* (engl. *assignment*) θ eine (totale) Abbildung der Form

$$\theta \colon V_n \to \{0, 1\}.$$

Die Menge aller Belegungen in n Variablen sei

$$\Theta_n := \{\theta \mid \theta \colon V_n \to \{0, 1\}\}.$$

Ist n aus dem Kontext klar, so schreiben wir nur Θ statt Θ_n.

Definition 1 (Evaluierung) *Es sei $n \in \mathbb{N}$ und $\theta \in \Theta_n$. Es sei $F_n^* \supseteq V_n$ die Menge aller Formeln, die mit Variablen aus V_n konstruiert werden können. $\hat{\theta} \colon F_n^* \to \{0, 1\}$ sei definiert durch:*

1. *Für $x \in V_n$ ist $\hat{\theta}(x) = \theta(x)$.*
2. *Sind $F, G \in F_n^*$ Formeln, dann ist*

$$\hat{\Theta}(F \wedge G) = \begin{cases} 1 & \text{für } \hat{\Theta}(F) = 1 \text{ und } \hat{\Theta}(G) = 1, \\ 0 & \text{sonst,} \end{cases}$$

$$\hat{\Theta}(F \vee G) = \begin{cases} 1 & \text{für } \hat{\Theta}(F) = 1 \text{ oder } \hat{\Theta}(G) = 1, \\ 0 & \text{sonst,} \end{cases}$$

$$\hat{\Theta}(\neg F) = \begin{cases} 1 & \text{für } \hat{\Theta}(F) = 0, \\ 0 & \text{sonst.} \end{cases}$$

Ist F eine Formel und $\theta \in \Theta_n$, dann nennen wir $\hat{\theta}(F)$ also den Wahrheitswert von F unter θ. Eine Formel F ist genau dann erfüllbar, wenn eine Belegung existiert, die die Formel zu wahr evaluiert (engl. *satisfies*) oder $F = \emptyset$ gilt.

Definition 2 (Modell) *Es sei $n \in \mathbb{N}$, $F \in F_n^*$ und $\theta \in \Theta_n$. Gilt $\hat{\Theta}(F) = 1$, so nennen wir θ ein* Modell *von F, geschrieben $\theta \models F$. Ist $\hat{\theta}(F) = 0$, so schreiben wir $\theta \not\models F$.*

Bemerkung 1 (Trivialfälle) *Wir betrachten zwei zur Definition 1 gehörende Spezialfälle. Ist die leere Klausel in einer Formel enthalten, also $\square \in F$, so ist diese unerfüllbar. Ist F die leere Menge, also $F = \emptyset$, so ist diese erfüllbar.*

Sei SAT die Menge aller aussagenlogischen Formeln in KNF die erfüllbar sind, also alle die Formeln F, für die es eine Belegung θ mit $\theta \models F$ gibt:

$$\text{SAT} := \{F \mid F \in F_n^*, n \in \mathbb{N}, F \text{ ist erfüllbar.}\}$$

Ebenso sei k-SAT die Menge aller aussagenlogischen Formeln, die erfüllbar und in k-KNF sind.

$\theta(x_1)$	$\theta(x_2)$	$\theta(x_3)$	$\hat{\theta}(B_1)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

Tabelle 2.2: Alle möglichen Belegungen zu B_1.

Beispiele

Im folgenden betrachten wir spezielle erfüllbare Formeln und demonstrieren wie es auf dem Weg der vollständigen Suche möglich ist, zu einer Lösung – nämlich einer erfüllenden Belegung – zu kommen. Wir greifen in Abschnitt 3 auf diese Beispiele zurück.

Beispiel 1 Es sei $B_1 = \underbrace{(x_1 \vee \overline{x_2})}_{c_1} \wedge \underbrace{(\overline{x_1} \vee \overline{x_3})}_{c_2}$. In Tabelle 2.2 werden alle möglichen Belegungen $\theta \in \Theta$ mit ihrer Evaluation $\hat{\theta}(B_1)$ dargestellt. Da es mindestens eine, in diesem Fall sogar vier erfüllende Belegungen gibt, ist $B_1 \in \text{SAT}$. Ein Entscheidungsbaum zum obigen Beispiel ist in Abbildung 2.1 dargestellt. Die Blätter des Baumes geben in diesem

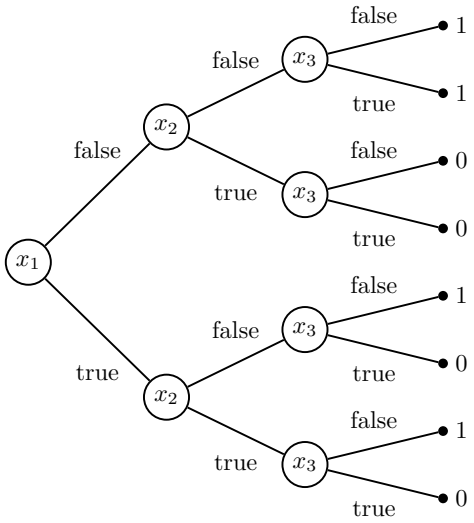

Abbildung 2.1: Entscheidungsbaum

Fall an, ob die Belegung erfüllend ist.

Beispiel 2 Sei $B_2 = (\overline{x_1}) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2})$.

Beispiel 3 Sei $B_3 = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_3} \vee \overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_1)$.

Beispiel 4 Sei $B_4 = (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_1 \vee x_2 \vee x_3)$.

3 Algorithmen

Im nun folgenden Kapitel stellen wir die hier behandelten Algorithmen vor. Wir unterscheiden deterministische und randomisierte Algorithmen. Der Unterschied zwischen beiden Algorithmen-Typen ist der „Zufall“ im randomisierten Algorithmus. In deterministischen Algorithmen werden bei jedem Programmaufruf immer die selben Schritte durchlaufen, wobei beim randomisierten Algorithmus abhängig von der Erzeugung der Pseudozufallszahlen verschiedene Pfade im Algorithmus gewählt werden.

Jeden Abschnitt beginnen wir mit einem leicht verständlichen Algorithmus, der typisch für die jeweilige Verfahrenskategorie ist.

3.1 Deterministische Algorithmen

3.1.1 Der Brute-force-Algorithmus

Beschreibung

Algorithmus 3.1 Brute-Force

Eingabe: CNFFormula F with n variables

1: Create a new assignment θ with size n;
2: Start with $\theta = \{(x_1, 0), \ldots, (x_n, 0)\}$;
3: **for** each $\theta \in \Theta_n$ **do**
4: **if** F is satisfied by θ **then**
5: **return** true;
6: **end if**
7: **end for**
8: **return** false;

Der Brute-Force-Algorithmus (3.1) ist ein typischer deterministischer Algorithmus. Er bekommt als Eingabe eine Formel und probiert Schritt für Schritt alle möglichen Belegungen durch. Bei n Variablen sind dies 2^n mögliche Belegungen. In Beispiel 1 sind es $n = 3$ und damit die 8 Belegungen aus Tabelle 2.2. Im schlimmsten Fall durchläuft der Algorithmus alle Belegungen und gibt `false` zurück falls die Formel unerfüllbar ist. Dies ist hier jedoch nicht der Fall, da es vier erfüllende Belegungen gibt.

Man erkennt, dass abhängig von der Reihenfolge, in der der BRUTE-FORCE-Algorithmus alle Belegungen testet, es möglich wäre eine erfüllende Belegung im ersten Schritt oder auch erst im letzten zu erhalten. Diese „Wegsuche“ zu einer erfüllenden Belegung wird in Abbildung 2.1 veranschaulicht.

Der Algorithmus geht hierbei die möglichen Belegungen nach einem bestimmten Schema durch. Wir interpretieren θ als Binärzahl und zählen im Algorithmus mittels der Umkehrfunktion f^{-1} beginnend bei $(\underbrace{0\dots0}_{n})_2$ aufwärts bis $(\underbrace{1\dots1}_{n})_2$. So erhalten wir alle 2^n möglichen Belegungen der gegebenen Formel. $\{0,1\}^n$ bijeektiv auf Θ_n abbilden.

Sei F eine Formel mit $n \in \mathbb{N}$ Variablen, w_i Wahrheitswerte und Θ_n die Menge aller Belegungen. Dann sei $f\colon \Theta_n \to \mathbb{N} \cap [0 : 2^{n-1}]$ für alle $\theta\colon \{w_1, \dots, w_n\} \in \Theta_n$ definiert durch

$$\begin{aligned} f(\theta) &= w_1 2^0 + w_2 2^1 + \dots + w_n 2^{n-1} \\ &= \sum_{i=1}^{n} w_i 2^{i-1}. \end{aligned}$$

Beispiel

Getestet mit Formel B_1 erzeugt der Algorithmus nun schrittweise die in Tabelle 2.2 gezeigten Belegungen. Da die mit der Binärzahl 000 assoziierte Belegung eine erfüllende Belegung ist, wird sofort mit „erfüllbar“ abgebrochen.

3.1.2 Der Algorithmus von Monien und Speckenmeyer

Zuerst führen wir einige Begriffe ein. Sei $n \in \mathbb{N}$, F eine Formel und $L' \subseteq L_n$. Dann definieren wir

$$\operatorname{lit}(L') := \{x \mid x \in L'\} \cup \{\overline{x} \mid \overline{x} \in L'\}$$

als Menge von gegensätzlichen Literalen.

Sei F eine Formel in k-KNF über V_n mit r Klauseln. Sei $K = \{L_1, \dots, L_l\} \in F$ eine Klausel mit $l \leq k$ Literalen. Offensichtlich gilt: Gibt es eine für F erfüllende Belegung θ, so erfüllt diese insbesondere die Klausel K. $\hat{\theta}_i$ sei für $i, j = 1, \dots, l$ und $j \leq i$ so gewählt, dass

$$\hat{\theta}_i(L_j) := \begin{cases} 1 & \text{für } i = j \\ 0 & \text{sonst} \end{cases} \tag{3.1}$$

Überprüfen wir F auf Erfüllbarkeit über die l Belegungen, die über die Belegungen θ_i erhalten werden, dann ist F genau dann erfüllbar, wenn eine der l Belegungen F erfüllt.

Sei $L' \subseteq L_n$ und θ eine Belegung. θ wird genau dann *autark* genannt, wenn

$$\forall c \in F : c \cap \text{lit}(L') \neq \emptyset \rightarrow \exists x \in c \cap \text{lit}(L') : \theta(x) = 1. \tag{3.2}$$

Ist eine Belegung autark, so werden alle Klauseln K mit $K \cap \text{lit}(L')$ durch θ erfüllt. Ist die Belegung nicht autark, so werden alle Literale im Schnitt durch θ auf 0 gesetzt.

Beschreibung

Der Algorithmus 3.2 erhält als Eingabe eine KNF-Formel F. Ausgegeben wird entweder „erfüllbar“ oder „nicht erfüllbar“. Durch die rekursiven Aufrufe wird die Formel auf einen der beiden Trivialfälle reduziert. Sollte keiner der Fälle bei Programmaufruf eintreten, so wird eine kürzeste Klausel $c = \{x_1, \ldots, x_l\}$ bestimmt. Der Programm-Abschnitt Zeile 10 – 18 wird maximal l mal ausgeführt oder bis eine autarke Belegung gefunden wurde.

Es wird in Zeile 11 überprüft ob θ_i eine autarke Belegung für die in der kürzesten Klausel vorkommenden Literale ist. Ist dies nicht der Fall, so wird i um eins erhöht. Andererseits wird eine neue Formel F' gebildet und mit der Prozedur BSat aufgerufen. F' beinhaltet nur genau die Klauseln, die mit Literalen $\text{lit}(\{x_1, \ldots, x_i\})$ der kürzesten Klausel disjunkt sind, da auf Grund Gleichung 4.2 alle Formeln die im Schnitt liegen, durch θ_i erfüllt werden.

Der Programm-Abschnitt ab Zeile 19 wird nur ausgeführt, falls keine autarke Belegung gefunden wurde. Es wird eine Formel F' in Zeile 23 nach dem obigen Schema gebildet. Ist eine der l Formeln, die über die Belegung 4.1 erhalten werden erfüllbar, so auch F.

Beispiel

Um nun ein besseres Verständnis für den Algorithmus zu bekommen, wird dieser mit der Formel B_1 aus Beispiel 1 demonstriert. Beim folgenden Ablauf wurden teilweise Abschnitte im Programm weggelassen, sowie abkürzende Schreibweisen verwendet damit die Übersicht erhalten bleibt.

1. Der Algorithmus bestimmt als kürzeste Klausel $(x_1 \vee \overline{x_2})$. Daraus folgt der Parameter $l \leftarrow 2$, die Variable `notaut` wird auf `true` gesetzt.
2. In der folgenden Schleife wird die Autarkie der jeweiligen Belegungen überprüft.
   a) Für $i = 1$ folgt die Menge $\text{lit} = \{x_1, \overline{x_1}\}$ und $\hat{\theta}_1(x_1) = 1$. Nun folgt die Autarkie-Prüfung:

   $$c_1 \cap \text{lit} = \{x_1\}, \hat{\theta}(x_1) = 1$$
   $$c_2 \cap \text{lit} = \{\overline{x_1}\}, \hat{\theta}(\overline{x_1}) = 0$$

   Also ist θ_1 nicht autark, i wird um eins inkrementiert.

b) Für $i = 2$ folgt die Menge $\text{lit} = \{x_1, \overline{x_1}, x_2, \overline{x_2}\}$ und $\hat{\theta}_2(x_1) = 0, \hat{\theta}_2(x_2) = 0$. Nun folgt die Autarkie-Prüfung:

$$c_1 \cap \text{lit} = \{x_1, \overline{x_2}\}, \hat{\theta}_2(\overline{x_2}) = 1$$
$$c_2 \cap \text{lit} = \{\overline{x_1}\}, \hat{\theta}_2(\overline{x_1}) = 1$$

Also ist θ_2 autark, `notaut` wird auf `false` gesetzt. Im nächsten Schritt wird

$$F' = \{c \in F \mid c \cap \text{lit} = \emptyset\} = \emptyset$$

erzeugt und BSAT(F') aufgerufen, welches `true` zurückgibt, weshalb „F satisfiable“ ausgegeben wird.

Implementierung

Die Autarkie-Prüfung ist der aufwändigste Abschnitt im Algorithmus. Hier müssen zunächst die Variablen der kürzesten Klausel zurücksubstituiert werden, damit die jeweilige Belegung θ angepasst werden kann. Das Substituieren der Literalnummern der jeweils kürzesten Klausel wurde über eine neue Liste gelöst, in die die Literale kopiert und so mittels ihrer Platznummer in der Liste assoziiert wurden. Anschließend wird für jede Klausel $K \in F$ bei nicht-leerem Schnitt überprüft, ob $\theta \models K$.

Algorithmus 3.2 BSat

Eingabe: KNF-Formel F

Ausgabe: **if** F is satisfiable **then** satisfiable **else** unsatisfiable

1: **if** $F = \emptyset$ **then**
2: **return** F satisfiable;
3: **end if**
4: **if** $\square \in F$ **then**
5: **return** F unsatisfiable;
6: **end if**
7: Determine a shortest clause $\{x_1, \ldots, x_l\} \in F$ and its length l;
8: notaut $\leftarrow$ `true`;
9: $i \leftarrow 1$;
10: **while** $i \leq l$ **and** notaut **do**
11: **if** $x_1 = \ldots = x_{i-1} =$ `false`, $x_i =$ `true` induces an autark truth assignment on $\mathrm{lit}(\{x_1, \ldots, x_i\})$ **then**
12: $F' \leftarrow \{c \in F \mid c \cap \mathrm{lit}(\{x_1, \ldots, x_i\}) = \emptyset\}$;
13: notaut $\leftarrow$ `false`;
14: **if** BSat(F') is satisfiable **then**
15: **return** F satisfiable;
16: **else**
17: **return** F unsatisfiable;
18: **end if**
19: **else**
20: $i \leftarrow i + 1$;
21: **end if**
22: **end while**
23: **if** notaut **then**
24: unsat $\leftarrow$ `true`;
25: i $\leftarrow 1$;
26: **while** $i \leq l$ **and** unsat **do**
27: $F_i \leftarrow \{c \backslash \{x_1, \ldots, x_{i-1}, \overline{x_i}\} \mid c \in F, c \cap \{\overline{x_1}, \ldots, \overline{x_{i-1}}, x_i\} = \emptyset\}$;
28: **if** BSat(F_i) is satisfiable **then**
29: unsat $\leftarrow$ `false`;
30: **return** F satisfiable;
31: **else**
32: $i \leftarrow i + 1$;
33: **end if**
34: **end while**
35: **if** $i = l + 1$ **then**
36: **return** F satisfiable;
37: **end if**
38: **end if**

3.1.3 Der Algorithmus von Dantsin, Goerdt, Hirsch und Schöning

Wir führen einige Begriffe aus der Kodierungstheorie ein. Sei $\mathcal{P}(A)$ die Potenzmenge einer Menge A. Es beschreibe das Symbol $\oplus$ die Antivalenz (den XOR-Junktor).

Definition 3 (Hamming-Distanz) *Sei $n \in \mathbb{N}$. Die Abbildung*

$$\mathrm{d}\colon \Theta_n \times \Theta_n \to \mathbb{N}$$

sei für alle $\theta_1, \theta_2 \in \Theta_n$ definiert durch

$$\mathrm{d}_n(\theta_1, \theta_2) := \sum_{i=1}^{n} \big(\theta(x_i)_1 \oplus \theta_2(x_i)\big)$$

und wird Hamming-Distanz *genannt.*

Die Hamming-Distanz gibt an in wie vielen Variablen sich zwei Belegungen unterscheiden. Analog wird zum Bestimmen der minimalen Hamming-Distanz zwischen zwei Belegungsmengen Σ und Γ die Abbildung $\mathrm{d}_{\min}$ definiert.

Definition 4 (Minimale Hamming Distanz) *Sei $n \in \mathbb{N}$. Die Abbildung*

$$\mathrm{d}_{\min}\colon \mathcal{P}(\Theta_n) \times \mathcal{P}(\Theta_n) \to \mathbb{N}$$

sei für alle $\Sigma, \Gamma \in \mathcal{P}(\Theta_n)$ definiert durch

$$\min \bigcup_{\sigma \in \Sigma, \gamma \in \Gamma} \mathrm{d}(\sigma, \gamma).$$

Wir nennen $\mathrm{d}_{\min}(\Sigma_\Gamma)$ die minimale Hamming-Distanz *zwischen Σ und Γ.*

Definition 5 (Code) *Ein Code der Länge n ist eine Untermenge von $\{0,1\}^n$.*

Definition 6 (Direkte Summe) *Seien $\mathcal{C}_1, \mathcal{C}_2$ Codes der Länge $n_1, n_2 \in \mathbb{N}$. Die direkte Summe (direct sum) von $\mathcal{C}_1$ und $\mathcal{C}_2$ ist der Code $\mathcal{C}$ der Länge $n_1 + n_2$, der alle möglichen Konkatenationen $w_1 \circ w_2$ enthält, wobei $w_1 \in \mathcal{C}_1$ und $w_2 \in \mathcal{C}_2$.*

Ist F eine Formel, so bezeichnet $F_{|L=1}$ die Formel, in der alle Klauseln, die das Literal L beinhalten gelöscht wurden und das Literal $\overline{L}$ aus allen Klauseln gelöscht wurde. Die jeweilige Belegung θ wurde so verändert, dass das Literal L zu „wahr“ evaluiert wird.

Beschreibung

Für diesen Algorithmus wird ein spezieller abdeckender Code $\mathcal{C}$ als Parameter benötigt, der im Algorithmus eine bestimmte Menge an Belegungen enthält (jedes $\theta \in \mathcal{C}$ ist eine Belegung). In [DGHS00, 3] wird beschrieben, welche Eigenschaften $\mathcal{C}$ haben muss, um ein sog. gut abdeckender Code zu sein. Wir beschränken uns auf einen später beschriebenen Spezialfall. Im folgenden bezeichne $\widetilde{\mathcal{C}}$ diesen speziellen Code und $\varepsilon > \delta > 0$ und $0 < \varrho < \frac{1}{2}$ Parameter, die bei der Code-Erzeugung notwendig waren.

Ist $r = \lfloor n(\varrho + \varepsilon) \rfloor$ berechnet, so wird der Algorithmus $\textsc{Search}(F, \theta, r)$ für jedes $\theta \in \widetilde{\mathcal{C}}$ aufgerufen. Gibt einer dieser $|\widetilde{\mathcal{C}}|$ Aufrufe „wahr“ zurück, so ist die Formel erfüllbar, sonst nicht.

Search Es folgen drei Abfragen und ein rekursiver Aufruf.

1. Ist $\theta \models F$ eine erfüllende Belegung, so wird „wahr“ zurückgegeben.
2. Ist $r \leq 0$, so wird „falsch“ zurückgegeben.
3. Wenn $\square \in F$, so wird „falsch“ zurückgegeben.
4. Wähle eine durch θ nicht erfüllte Klausel K. Für jedes Literal $L \in K$, rufe $\textsc{Search}(F_{|L=1}, \theta, r-1)$ auf und gebe „wahr“ aus, wenn mindestens einer dieser $|L|$ rekursiven Aufrufe „wahr“ ausgibt, sonst „falsch“.

Algorithmus 3.3 CoveringSearch

Eingabe: $\varepsilon > \delta > 0, 0 < \varrho < \frac{1}{2}$

```
1: Generate a covering C for {0,1}^n by using GenerateGoodCoverings();
2: r = ⌊n(ϱ + ε)⌋;
3: for each code word θ in C do
4:    Search(F, θ, r);
5:    if at least one procedure call returns true then
6:       return true;
7:    else
8:       return false;
9:    end if
10: end for
```

Algorithmus 3.4 GenerateGoodCoverings

Eingabe: Code Γ of length ν

1: Find the smallest integer q such that $n \leq q\nu$.
2: Generate the direct sum Γ^q. If n is not divisble by ν, we restrict the code words of Γ^q to the first n positions.

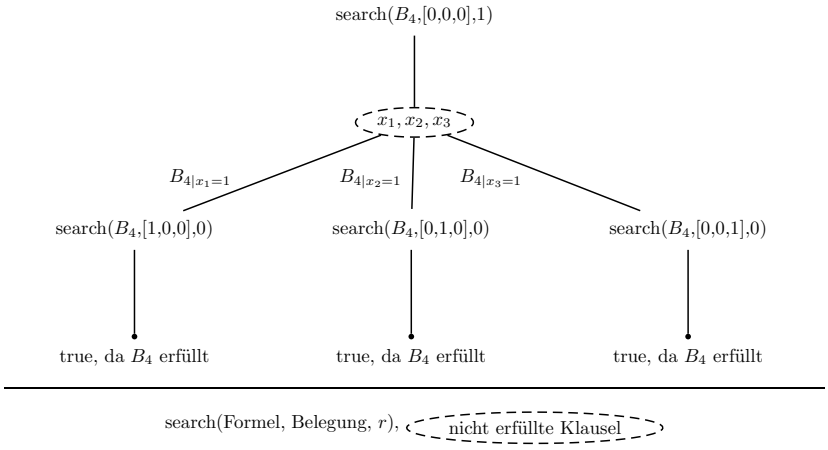

Abbildung 3.1: Rekursion von SEARCH in COVERINGSEARCH

Beispiel

Wir testen den Algorithmus mit der Formel $B_4 =: F$.

1. $\hat{\theta}_1(x_1) = 0, \hat{\theta}(x_2) = 0, \hat{\theta}(x_3) = 0$. Es wird $\textsc{Search}(F, \theta_1, 1)$ aufgerufen. Die Belegung θ_1 erfüllt nicht F, $r > 0$ und $\square \notin F$. Der Algorithmus bestimmt die nicht erfüllte Klausel $(x_1 \vee x_2 \vee x_3)$. Es folgen drei rekursive Aufrufe.

   a) $\textsc{Search}(F_{|x_1=1}, \theta_1, 0) \rightarrow$ gibt „wahr“ zurück, da erfüllende Belegung.

   b) $\textsc{Search}(F_{|x_2=1}, \theta_1, 0) \rightarrow$ gibt „wahr“ zurück, da erfüllende Belegung.

   c) $\textsc{Search}(F_{|x_3=1}, \theta_1, 0) \rightarrow$ gibt „wahr“ zurück, da erfüllende Belegung.

   Da mindestens eine Prozedur „wahr“ ausgegeben hat, ist die Formel erfüllbar. In Abbildung 3.1 erkennt man die rekursive Struktur des Algorithmus. Sobald ein „Blatt“ des Baumes wahr ist, ist die Formel erfüllbar.

Implementierung

In dieser Arbeit wurde der in [DGHS00, Kap. 2, S. 239] angesprochene Spezialfall implementiert, da die Bestimmung von $\widetilde{\mathcal{C}}$ zur Generierung guter Abdeckungen im Algorithmus „fest verdrahtet“ ist, da sonst ein exponentieller Laufzeitfaktor mit einfließen würde. Hierzu wählten wir ein $\widetilde{\mathcal{C}}$ der Form

$$\widetilde{\mathcal{C}} = \{\{\underbrace{0, \ldots, 0}_{n}\}, \{\underbrace{1, \ldots, 1}_{n}\}\},$$

und einen Abdeckungsradius von $r = \frac{n}{2}$. Der Algorithmus mit diesem speziellen, abdeckenden Code hat eine theoretische Laufzeit mit der oberen Schranke von $1{,}733^n$.

3.2 Randomisierte Algorithmen

Algorithmus 3.5 RandomSAT

Eingabe: CNFFormula F with n variables

```
1: Create an assignment θ with size n;
2: for i = 0 to 100 do
3:    Create a new randomly choosen assignment θ;
4:    for each clause C in F do
5:       if C is not satisfied by θ then
6:          Change the assignment θ in such a way that it satisfies C;
7:       end if
8:    end for
9:    if F is satisfied by θ then
10:      return true;
11:   end if
12:   i ← i + 1;
13: end for
14: return false;
```

In den folgenden Abschnitten wird der Begriff „zufällig“ desöfteren benutzt. Wir meinen damit eine Auswahl eines Elementes auf Basis einer gleichverteilten Menge und schreiben stattdessen auch *pseudozufällig*.

Der randomisierte Algorithmus wählt zunächst eine neue Belegung folgender Art. Für jede Variable der übergebenen Formel wird er diese auf entweder „wahr“ oder „falsch“ setzen. Dieses geschieht pseudozufällig unter dem Prinzip einer Gleichverteilung beider Wahrheitswerte. Sei $n \in \mathbb{N}$ die Anzahl der Variablen und $m \in \mathbb{N}$ die Anzahl der erfüllenden Belegungen. Demnach testet der Algorithmus eine erfüllende Belegung beim ersten Versuch mit einer Wahrscheinlichkeit von $\frac{m}{2^n}$. In Beispiel 1 würde dies $\frac{4}{2^3} = \frac{1}{2}$ bedeuten.

Man sieht recht schnell für Beispiel 1, sollte er eine der vier falschen Belegungen „erraten“, so muss nicht zwingenderweise sie im Verbesserungsschritt in eine erfüllende umgewandelt werden. In Tabelle 3.1 sind die Hamming-Distanzen von den erfüllenden Belegungen Θ_T (links, vergl. Tab. 2.2) zu den nicht erfüllenden Belegungen Θ_F (oben) dargestellt.

	010	011	101	111
000	1	2	2	3
001	2	1	1	2
100	2	3	1	2
110	1	2	2	1

Tabelle 3.1: Hamming-Distanzen zu Belegungen von Beispiel 1

Man erkennt, dass

$$\forall \theta \in \Theta_T : \mathrm{d}_{\min}(\Theta_F, \theta) = 1.$$

Das bedeutet, es ist möglich durch die Änderung einer Variablenzuweisung eine nicht erfüllende Belegung in eine erfüllende zu ändern. Sollte eine Belegung nicht erfüllbar sein, so wird in der `for`-Schleife (Zeile 4) eine Klausel gefunden (Zeile 5), die durch θ nicht erfüllt wird. Anschließend wird θ durch das Wechseln der Belegung einer in der Klausel vorhandenen Variable derart geändert, dass die betroffene Klausel zu „wahr“ evaluiert wird. Man sieht, je öfter die `for`-Schleifen (Zeile 2 – 8) durchlaufen werden, desto wahrscheinlicher werden mehrere verschiedene Belegungen getestet. Dadurch steigt die Wahrscheinlichkeit, dass eine erfüllende Belegung θ gefunden wird.

Es ist möglich, dass der Algorithmus eine Belegung θ zu Beginn wählt, für die $\theta \not\models B_1$ gilt. Eine der vier möglichen wäre zum Beispiel die Belegung

$$\hat{\theta}(x_1) = 0, \hat{\theta}(x_2) = 1, \hat{\theta}(x_3) = 1.$$

Der Algorithmus erkennt, dass die erste Klausel $(x_1 \vee \overline{x_2})$ nicht erfüllt wird und verändert die Belegung z. B. zu

$$\hat{\theta}(x_1) = 1, \hat{\theta}(x_2) = 1, \hat{\theta}(x_3) = 1.$$

Nun gilt zwar $\theta \models (x_1 \vee \overline{x_2})$, jedoch nicht $\theta \models B_1$. Der Algorithmus ändert diese Belegung wieder wegen der zweiten Klausel. An diesem Beispiel erkennt man, weshalb alle randomisierten Algorithmen zusätzlich eine sog. Erfolgswahrscheinlichkeit (success probability) haben. Es besteht immer eine bestimmte Möglichkeit, dass eine erfüllbare Formel als „nicht erfüllbar“ ausgewiesen wird. Dies gilt trivialerweise im Umkehrschluss nicht.

3.2.1 Der Algorithmus von Paturi, Pudlák, Saks und Zane

Für den in [PPSZ98] beschriebenen Algorithmus definieren wir die starke Resolution.

Definition 7 (Starke Resolution) *Seien C_1 und C_2 Klauseln, die genau eine Variable x gemeinsam haben, wobei $x \in C_1$ und $\overline{x} \in C_2$ (sollte dies nicht gewährleistet sein, so führen wir Umbenennungen durch). Für dieses Klauselpaar sei ihr Resolvent $R(C_1, C_2)$ definiert durch die Klausel $C = D_1 \vee D_2$, wobei D_1, D_2 durch das Löschen der Literale $x, \overline{x}$ aus den Klauseln C_1 und C_2 erhalten wird.*

Der Begriff „s-beschränkt“ gibt an, dass ein s-beschränkter Resolvent maximal s Literale beinhalten darf.

Beschreibung

Die Algorithmen 3.6 bis 3.9 arbeiten mit der logischen Resolution, sowie mit randomisierten Wahrheitsbelegungen. Als Eingabe erhält er eine KNF-Formel F mit $n > 1$ Variablen, sowie zwei Zahlen $s, i \in \mathbb{N}_+$, die sich wie folgt berechnen:

$$i = \left\lfloor n \cdot 2^{\frac{2}{3} \cdot n} \right\rfloor$$
$$s = \left\lfloor \frac{n}{\log n} \right\rfloor$$

Anschließend durchläuft der Algorithmus zwei Phasen:

Resolve Gesucht wird ein Resolvent zweier Klauseln C_1, C_2 in der Formel mit der Eigenschaft $|R(C_1, C_2)| \leq s$. Der vorherige Schritt wird so oft wiederholt, bis kein neuer Resolvent gefunden wird.

Search Die folgenden Schritte werden i-mal wiederholt: Es werden eine zufällige Permutation π von den Zahlen 1 bis n und ein zufällige Belegung θ erstellt. Nun wird nach der Reihenfolge von π überprüft, ob die jeweilige Variable x_i als Einheitsklausel in der Formel vorkommt. Ist dies der Fall, so wird $\hat{\theta}(x_i) = 1$ gesetzt, wenn es eine positive Einheitsklausel ist bzw. $\hat{\theta}(x_i) = 0$ gesetzt, wenn es eine negative Einheitsklausel ist. Gibt es keine Einheitsklauseln innerhalb der übergebenen Formel, so übernimmt man die zufällig bestimmte Belegung.

Erfüllt nun die danach erhaltene Belegung die Formel, so gibt man die gefundene Belegung aus und beendet das Programm.

Die Fehlerrate vom Algorithmus beträgt $\mathrm{O}(e^{-n})$.

Beispiel

Nun testen wir den Algorithmus mit der Eingabe B_2. Beim folgenden Ablauf wurden teilweise Abschnitte im Programm weggelassen, sowie abkürzende Schreibweisen verwendet damit die Übersicht erhalten bleibt. Wir wählen die Parameter $s = \left\lfloor \frac{4}{\log 4} \right\rfloor = 2$, und für die Anzahl i der Wiederholungen erhalten wir $i = 25$. Aufgrund der Übersichtlichkeit schreiben wir die Belegung θ in der Mengenschreibweise. Sei $n \in \mathbb{N}$. Die Abbildung $\cap_V \colon L_n \times L_n \to V_n$, sei für alle $C_1, C_2 \in L_n$ definiert durch

$$C_1 \cap_V C_2 := \{x \mid x \in C_1 \wedge \overline{x} \in C_2\} \cup \{x \mid x \in C_2 \wedge \overline{x} \in C_1\}.$$

1. Zuerst wird ResolveSat$(B_2, 2, 25)$ aufgerufen.
2. In Resolve Zeile 1–3 (Algorithmus [3.7]) werden folgende vier Schritte durchlaufen:

a) $F_s = \underbrace{(\overline{x_1})}_{c_1} \wedge \underbrace{(x_2 \vee \overline{x_3} \vee x_4)}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_2})}_{c_3}$

$$c_1 \cap_V c_2 = \emptyset$$
$$c_1 \cap_V c_3 = \{x_1\} \qquad \Rightarrow R(c_1, c_2) = (\overline{x_2})$$

$$F_s = \underbrace{(\overline{x_1})}_{c_1} \wedge \underbrace{(x_2 \vee \overline{x_3} \vee x_4)}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_2})}_{c_3} \wedge \underbrace{(\overline{x_2})}_{c_4}$$

b)

$$c_1 \cap_V c_4 = \emptyset$$
$$c_2 \cap_V c_3 = \{x_2\} \qquad \Rightarrow R(c_2, c_3) = (\overline{x_3} \vee x_4 \vee x_1)$$

$$F_s = \underbrace{(\overline{x_1})}_{c_1} \wedge \underbrace{(x_2 \vee \overline{x_3} \vee x_4)}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_2})}_{c_3} \wedge \underbrace{(\overline{x_2})}_{c_4} \wedge \underbrace{(\overline{x_3} \vee x_4 \vee x_1)}_{c_5}$$

c)

$$c_1 \cap_V c_5 = \{x_1\} \qquad \Rightarrow R(c_1, c_5) = (\overline{x_3} \vee x_4)$$

$$F_s = \underbrace{(\overline{x_1})}_{c_1} \wedge \underbrace{(x_2 \vee \overline{x_3} \vee x_4)}_{c_2} \wedge \underbrace{(x_1 \vee \overline{x_2})}_{c_3} \wedge \underbrace{(\overline{x_2})}_{c_4} \wedge \underbrace{(\overline{x_3} \vee x_4 \vee x_1)}_{c_5} \wedge \underbrace{(\overline{x_3} \vee x_4)}_{c_6}$$

d) Im folgenden Durchlauf werden keine neuen, noch nicht in der Formel vorhandenen Resolventen gefunden.

Damit erhalten wir

$$F_s = (\overline{x_1}) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_2}) \wedge (\overline{x_3} \vee x_4 \vee x_1)(\overline{x_3} \vee x_4).$$

3. SEARCH$(F_s, 25)$ läuft wie folgt ab:

$\pi_1 \leftarrow \{1, 3, 2, 4\}$
$\theta_1 \leftarrow \{0, 1, 1, 1\}$
$\gamma_1 \leftarrow$ MODIFY(F_s, π_1, θ_1)

a) MODIFY erkennt die beiden Einheitsklauseln $(\overline{x_1})$ und $(\overline{x_2})$, ändert die Belegung θ_1 an den entsprechenden Stellen und setzt $\gamma_1 = \{0, 0, 1, 1\}$

Die Erfüllbarkeitsprüfung in Zeile 5 in Algorithmus 3.8 ist erfolgreich und gibt γ_1 aus.

Die Prozedur MODIFY nimmt nur bei vorhandenen Einheitsklauseln Veränderungen an der pseudozufälligen Belegung θ vor. Dies sind für den Betrachter „offensichtliche Anpassungen“, da Klauseln mit nur einem Literal trivialerweise eine Belegung benötigen, die das eine Literal erfüllen.

Algorithmus 3.6 ResolveSat

Eingabe: CNFFormula F, integer $s, i \in \mathbb{N}^+$

1: $F_s \leftarrow$ **Resolve**(F, s)
2: **Search**(F_s, i)

Algorithmus 3.7 Resolve

Eingabe: CNFFormula F, integer s $F_s \leftarrow F$;

1: **while** F_s has a s-bounded resolvable pair C_1, C_2 with $R(C_1, C_2) \notin F_s$ **do**
2: $\quad F_s \leftarrow F_s \wedge R(C_1, C_2)$.
3: **end while**
4: **return** F_s

Algorithmus 3.8 Search

Eingabe: CNFFormula F, integer i

1: **repeat**
2: $\quad \pi \leftarrow$ uniformly random permutation of $1, \ldots, n$
3: $\quad \theta \leftarrow$ uniformly random assignment $\in \{0,1\}^n$
4: $\quad \gamma \leftarrow$ **Modify**(F, π, θ)
5: $\quad$ **if** γ satisfies F **then**
6: $\quad\quad$ **return** γ;
7: $\quad$ **end if**
8: **until** i times repeated
9: **return** false;

Algorithmus 3.9 Modify

Eingabe: CNFFormula $G(x_1, \ldots, x_n)$, permutation π of $\{1, \ldots, n\}$, assignment θ
Ausgabe: assignment γ

1: $G_0 \leftarrow G$
2: **for** $i \leftarrow 1$ to n **do**
3: $\quad$ **if** G_{i-1} contains the unit clause $x_{\pi(i)}$ **then**
4: $\quad\quad z_{\pi(i)} \leftarrow 1$
5: $\quad$ **else if** G_{i-1} contains the unit clause $\overline{x_{\pi(i)}}$ **then**
6: $\quad\quad z_{\pi(i)} \leftarrow 0$
7: $\quad$ **else**
8: $\quad\quad z_{\pi(i)} \leftarrow \gamma_{\pi(i)}$
9: $\quad$ **end if**
10: $\quad G_{i+1} \leftarrow G_i$ mit $x_{\pi(i)} \leftarrow z_{\pi(i)}$
11: **end for**

Implementierung

Die Überprüfung nach der Existenz eines Resolventen mit nur $s \in \mathbb{N}_+$ Elementen wurde in zwei Schritten vollzogen (Algorithmus 3.7, Zeile 1). Seien hierzu F eine Formel mit $k \in \mathbb{N}$ Klauseln und $i \neq j \leq k \in \mathbb{N}$.

1. Wir wählen Klausel K_i und testen ob der Schnitt mit Klausel K_j maximal s Elemente hat und nicht leer ist.
2. Nun überprüfen wir, ob für ein Literal $L \in K_i \cap K_j$ eine erfüllende Belegung $\hat{\theta}(L) = 1$ existiert.

Zur Erzeugung der zufälligen Permutation π verwendeten wir den in [Grü05, S. 109, Satz 4.9] beschriebenen Algorithmus 3.10, der ausgehend von $n-1$ gleichverteilten Zufallsvariablen $U_1, \ldots, U_{n-1}$ die Permutation $\Pi = (\Pi_1, \ldots, \Pi_n)$ konstruiert. Diese ist auf der Menge der Permutationen von $(1, \ldots, n)$ gleichverteilt.

Algorithmus 3.10 RandomPermute

Eingabe: $U_1, \ldots, U_{n-1}, n$

1: $i \leftarrow 1, \Pi \leftarrow (1, \ldots, n)$;
2: **while** $i < n$ **do**
3: $\quad K_i := i + \lfloor (n+1-i)U_i \rfloor$;
4: $\quad$ Change the position of Π_i and Π_{K_i};
5: $\quad i \leftarrow i + 1$;
6: **end while**

3.2.2 Der Algorithmus von Hofmeister, Schöning, Schuler und Watanabe

Die hier vorgestellten Algorithmen RW und RED2 benötigen als Eingabe eine spezielle Klauseluntermenge von der eingegebenen 3-KNF-Formel. Im folgenden definieren wir die Eigenschaft dieser Menge.

Definition 8 (Unabhängigkeit) *Zwei Klauseln C und C' werden* unabhängig *(independent) genannt, wenn sie keine Variable gemeinsam haben. Sei F eine Formel mit m Klauseln und $\mathcal{C}$ eine Teilmenge der in F enthaltenen Klauseln. Wir nennen $\mathcal{C} \subseteq F$ eine* maximal unabhängige Klauselmenge*, wenn alle in $\mathcal{C}$ enthaltenen Klauseln paarweise unabhängig sind und keine Klausel von $F \backslash \mathcal{C}$ hinzugefügt werden kann ohne dieses zu zerstören.*

Beschreibung

Um die von dem in [HSSW02] beschriebenen Algorithmus erzielte Laufzeit zu erreichen, werden zwei leicht verschiedene Algorithmen RED2 und RW nacheinander ausgeführt (siehe Algorithmen 3.11 und 3.13).

Beide Prozeduren sind in den ersten Aufrufen identisch (bis auf die Startparameter p_i und q_i). Ihnen liegt jeweils eine 3-KNF-Formel als Eingabe vor. Sie setzen voraus, dass zunächst eine maximal unabhängige Menge von Klauseln $\mathcal{C} = \{C_1, \ldots, C_m\}$ erzeugt wird, die o.B.d.A. die Variablen $x_1, \ldots, x_{3m}$ beinhalten. Die „Rollen“ der Variablen x_i und $\overline{x_i}$ werden so vertauscht, dass keine Negationen vorhanden sind. Schließlich wählt man eine von den Parametern abhängige „zufällige“ Belegung θ in den Zeilen 4–6 des Algorithmus 3.12. Damit sind $3m$ Variablen Wahrheitswerte zugewiesen.

Von nun an unterscheidet sich die Vorgehensweise beider Algorithmen.

Red2 Die nun über θ erhaltene 2-SAT Formel wird mit dem Polynomialzeit-Algorithmus BTOSAT gelöst.

RW Die restlichen Variablen $x_{3m+1}, \ldots, x_n$ werden „zufällig“ mit 0 oder 1 belegt. Anschließend wird die Prozedur RANDOMWALK(θ) aufgerufen.

RandomWalk Solange die Belegung θ nicht erfüllend ist, wird eine neue Belegung θ' erzeugt, indem ein Literal einer Klausel, das nicht durch θ erfüllt wurde, pseudozufällig gewählt und die zugehörige Belegung getauscht wird. Dies wird $3n$-mal wiederholt.

Es ist möglich, dass auf Grund der Wahl der unabhängigen Klauselmenge $\mathcal{C}$ und der anschließenden pseudozufälligen Belegung eine 2-Sat-Instanz entsteht, die nicht erfüllbar ist. Auch über die Wahl des nicht erfüllten Literals in RANDOMWALK wird ersichtlich, weshalb der Algorithmus bei einer erfüllbaren Formel auch „nicht erfüllbar“ ausgeben kann. Die Erfolgswahrscheinlichkeit des Algorithmus beträgt $\Omega(1{,}330258^{-n})$.

Algorithmus 3.11 Red2

Eingabe: $q_1, q_2, q_3 \in \mathbb{R}^+ \ \{ \ 3q_1 + 3q_2 + q_3 = 1\}$

1: **Ind-Clauses-Assign**(q_1, q_2, q_3);
2: 2CNFFormula $F' \leftarrow$ simplify the given formula with the assignment obtained by **Ind-Clauses-Assign**;
3: Call BTOSat(F');

Algorithmus 3.12 Ind-Clauses-Assign

Eingabe: $p_1, p_2, p_3 \in \mathbb{R}^+ \ \{3p_1 + 3p_2 + p_3 = 1\}$
Ausgabe: An assignment that satisfies all clauses in $\mathcal{C}$

1: **for** $C \in \mathcal{C}$ **do**
2: Assume $C = x_i \vee x_j \vee x_k$
3: Set the variables x_i, x_j, x_k randomly in such a way that for $\theta = (x_i, x_j, x_k)$ the following holds:
4: $\Pr\{\theta = (0,0,1)\} = \Pr\{\theta = (0,1,0)\} = \Pr\{\theta = (1,0,0)\} = p_1$
5: $\Pr\{\theta = (0,1,1)\} = \Pr\{\theta = (1,0,1)\} = \Pr\{\theta = (1,1,0)\} = p_2$
6: $\Pr\{\theta = (1,1,1)\} = p_3$
7: **end for**

Algorithmus 3.13 RW

Eingabe: $p_1, p_2, p_3 \in \mathbb{R}^+ \ \{3p_1 + 3p_2 + p_3 = 1\}$

1: **Ind-Clauses-Assign**(p_1, p_2, p_3);
2: Set the variables $x_{3m+1}, \ldots, x_n$ independently of each other to 0 or 1, each with a probability $\frac{1}{2}$.
3: To the assignment θ obtained in this way, apply **RandomWalk**(θ).

Algorithmus 3.14 RandomWalk

Eingabe: Assignment θ

1: $\theta' \leftarrow \theta$
2: **for** $3n$ times **do**
3: **if** θ' satisfies F **then**
4: **return** true;
5: **end if**
6: $C \leftarrow$ a clause of F that is not satisfied by θ';
7: Modify θ' as follows: Select a literal of C uniformly at random and flip the assignment to this literal;
8: **end for**

Algorithmus 3.15 BTOSat

Eingabe: 2CNFFormula F

1: $F \leftarrow$ **PropUnit**(F);
2: **while** $\square \notin F$ **and** F is not empty **do**
3: Choose an unassigned variable x;
4: $P \leftarrow$ **PropUnit**$(F \wedge (x))$;
5: **if** $\square \in P$ **then**
6: $F \leftarrow$ **PropUnit**$(F \wedge (\overline{x}))$;
7: **else**
8: $F \leftarrow P$;
9: **end if**
10: **if** $\square \in F$ **then**
11: **return** false;
12: **else**
13: **return** true;
14: **end if**
15: **end while**

Algorithmus 3.16 PropUnit

Eingabe: CNFFormula F

1: **while** F contains an unit clause U **do**
2: **if** F contains $\overline{U}$ **then**
3: $F \leftarrow \big(F \backslash \{U, \overline{U}\}\big) \cup \square$;
4: **else**
5: CNFFormula temp = new CNFFormula();
6: **for** each clause C in F **do**
7: **if** C contains U **or** C contains $\overline{U}$ **then**
8: **if** C contains $\overline{U}$ **then**
9: Remove $\overline{U}$ from C;
10: **end if**
11: Add C to temp;
12: **end if**
13: **end for**
14: F $\leftarrow$ temp;
15: $F \leftarrow F \backslash (U)$;
16: **end if**
17: **end while**
18: **return** F;

Beispiel

Wir demonstrieren die Funktionsweise des Algorithmus' an der Formel B_3. Der erste Algorithmusabschnitt wird mit RED2($\frac{1}{7}, \frac{1}{7}, \frac{1}{7}$) gestartet.

1. Die unabhängige Klauselmenge wird zu $\mathcal{C} = \{\{x_1, \overline{x_2}, x_3\}\}$ bestimmt.
2. Da die erzeugte Pseudozufallszahl in der Prozedur IND-CLAUSES-ASSIGN $3 \cdot p_1 + 3 \cdot p_2 < 0{,}993 < 1$ ist, wird die Belegung $\theta = \{1, 0, 1\}$ gewählt, die alle Literale in der Formel zu wahr evaluiert.
3. Im nächsten Schritt wird die Formel abhängig von θ zu $B_1 = \{\square\}$ vereinfacht und der 2-SAT-Algorithmus BTOSAT gestartet. Dieser gibt „nicht erfüllbar" zurück, da die leere Klausel enthalten ist.
4. Nun wird der zweite Algorithmus mit RW($\frac{4}{21}, \frac{2}{21}, \frac{3}{21}$) gestartet. Dieser erzeugt die selbe unabhängige Klauselmenge wie oben und erzeugt in der Prozedur IND-CLAUSES-ASSIGN die Pseudozufallszahl „0,436". Da $2q_1 < 0{,}436 < 3q_1$ ist, wird $\theta = \{1, 1, 0\}$ gewählt (dies ist die Belegung, bei der nur das erste Literal erfüllt wird).
5. RANDOMWALK verändert nichts, da allen Variablen ein Wahrheitswert zugeordnet ist. Die Erfüllbarkeitsprüfung wird mit „erfüllbar" abgebrochen.

Implementierung

Wir erzeugen die unabhängige Klauselmenge $\mathcal{C}$, indem wir beginnend bei der ersten Klausel nacheinander die Klauseln hinzufügen, die die Eigenschaft der Unabhängigkeit nicht zerstören. Für den in RED2 benötigten Polynomialzeit-Algorithmus zur Lösung eines 2-SAT-Problems verwenden wir den in [Cha04, S. 15] beschriebenen Algorithmus 3.15. In diesem Algorithmus wird für jede Variable eine Einheitsklausel hinzugefügt und anschließend überprüft, ob eine für diese Klausel erfüllende Belegung auch die Formel erfüllt. Man nennt diese Vorgehensweise Einheitsklausel-Resolution (unit resolution). Seien $m, n \in \mathbb{N}$. Bei n Variablen und m Klauseln beträgt die Laufzeit des Algorithmus $O(m \cdot n)$.

4 Implementierung

4.1 DIMACS-Format

Um nun ein gängiges Dateiformat für das Einlesen der Formeln zu verwenden, wurde das vom Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) in [D93a] beschriebene Format verwendet. Dies ist ein De-facto-Standard, welcher in mehreren Wettbewerben benutzt wurde, in denen es unter anderem um das Lösen von SAT-Problemen ging. Diese Wettstreite heißen „DIMACS-Challenges“ [DC92]. Aus diesem Grund hat das Institut eine große Menge von Formeln in diesem Format online zugängig gemacht. Ein Teil dieser Formeln wurde beim Testen der in dieser Arbeit behandelten Algorithmen verwendet.

Eine nach diesem Standard formatierte Datei gliedert sich in drei aufeinander folgende Bereiche. Die Struktur ist zeilenorientiert.

Kommentare Dieser Bereich wird dazu genutzt um Informationen über die Datei vermitteln zu können und wird vom Parser übersprungen. Jede Kommentar-Zeile beginnt mit einem klein geschriebenem `c`.

```
c Dies ist ein Beispiel-Kommentar.
```

Problem-Zeile Es gibt genau eine Problem-Zeile pro Datei. Sie muss immer die letzte über den Klausel-Zeilen stehende Zeile sein und wird mit einem klein geschriebenem `p` eingeleitet.

```
p FORMAT VARIABLES CLAUSES
```

Das Feld `FORMAT` steht für das folgende Format der Klauseln, in diesem Fall Konjunktive Normalform und damit `CNF`. Eine andere, hier nicht genutzte Möglichkeit wäre `SAT`. Genaueres hierzu siehe [D93a, 2.2]. Das Feld `VARIABLES` ist gleich der Anzahl der Variablen der Formel. In das Feld `CLAUSES` wird die Anzahl der Klauseln der Formel geschrieben.

Klauseln Die Klauseln der Formel erscheinen sofort nach der Problem-Zeile. Es wird angenommen, dass die Variablen von 1 bis n durchnummeriert sind. Jede Klausel wird als Nummernsequenz repräsentiert, die durch Leerzeichen getrennt sind. Die nicht negierte Variable wird als i, die negierte als $-i$ geschrieben.

Laut Standard wird jede Klausel durch eine 0 beendet. Hierdurch sind Klauseln über mehrere Zeilen möglich. Das Einlesen solcher zeilenübergreifender Klauseln wurde jedoch nicht implementiert, da die Testdaten solche Sonderfälle nicht beinhalten.

Möchten wir die aussagenlogische Formel aus Beispiel 1 im DIMACS-Format darstellen, so würde man eine Datei mit folgendem Inhalt erhalten.

```
c Beispiel 1
p CNF 3 2
1 -2 0
-1 -3 0
```

Abbildung 4.1: CNF-Format für die Beispiel-Formel B_1.

4.2 Programmdesign und Rahmenwerk

Alle Algorithmen wurden in Java (Java 2 Platform Standard Edition 5.0) implementiert. Um eine möglichst effiziente Implementierung der Formeln zu finden, wurde zunächst analysiert, welche Art von Operationen hier in den Algorithmen durchgeführt werden. Da wir den Zugriff auf einzelne Elemente, sowie das Iterieren durch die Elemente benötigen, fiel die Wahl auf das Interface `List` aus dem Standardpaket `java.util`. Sie ist die Java-Repräsentation einer linearen Liste. `ArrayList` erlaubt das Einfügen von Elementen an beliebiger Stelle und bietet sowohl sequentiellen als auch wahlfreien Zugriff auf die Elemente. Das JDK realisiert `ArrayList` als Array von Elementen des Typs `Object`. Daher sind Zugriffe auf vorhandene Elemente und das Durchlaufen der Liste schnelle Operationen.

Da alle Algorithmen mit Formeln in KNF, sowie Belegungen arbeiten, wurde eine Struktur mit vier Klassen entwickelt und diese mit JUnit Tests getestet:

Assignment ist die Repräsentation einer Liste mit n booleschen Elementen. Sie entspricht einer Implementierung einer Belegung θ.

Literal beinhaltet die Variablennummer und ein zusätzliches boolesches Feld für eine mögliche Negation.

Clause ist die Repräsentation einer Liste von Literal-Objekten.

CNFFormula ist die Repräsentation einer Liste von Clause-Objekten.

Für die Klassen `Literal`, `Clause` und `CNFFormula` wurden notwendige Methoden zur Erfüllbarkeitsprüfung und Elementverwaltung implementiert. Hierdurch wurde der eigentliche Programmcode übersichtlich und semantisch einheitlich. In Abbildung 4.2 ist ein UML-Klassendiagramm für diese Klassen dargestellt.

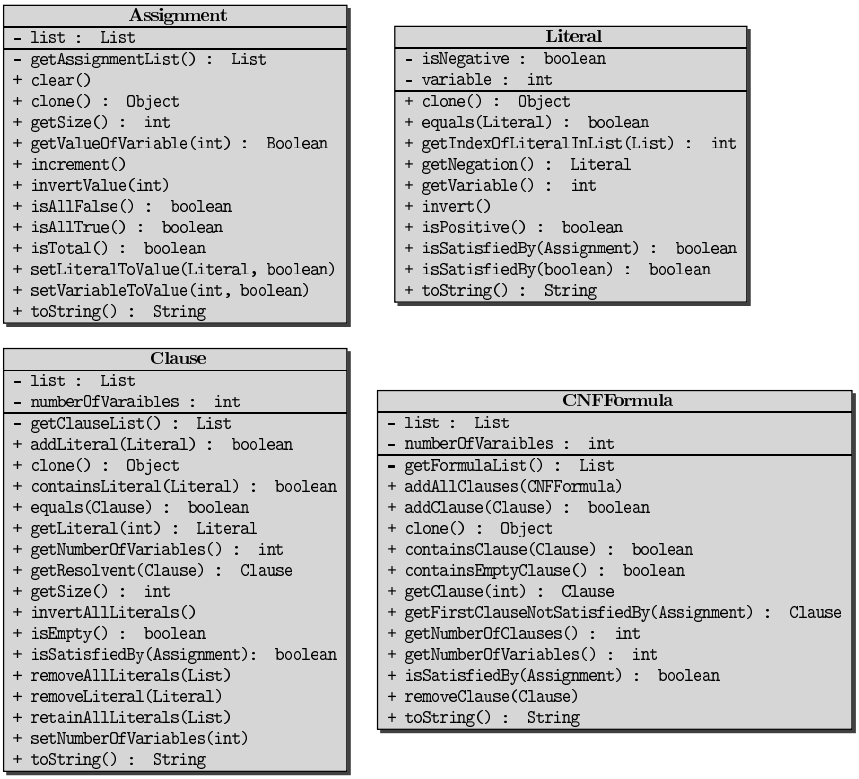

Abbildung 4.2: UML-Klassendiagramme für die Klassen `Assignment`, `Literal`, `Clause`, `CNFFormula`

Wenn wir im folgenden über „zufällig“ sprechen, legen wir eine Gleichverteilung auf $\{0, 1\}$ zu Grunde. In den randomisierten Algorithmen kommen teilweise „zufällige“ Belegungen vor. Hierfür verwenden wir aus dem Standard-Paket `java.util` die Klasse `Random` zur Erzeugung von Pseudozufallszahlen. Die Klasse implementiert einen linearen Kongruenzgenerator mit einem 48-bit Startwert (vgl. [Knu98, 3.2.1]). Sollte eine „zufällige“ Entscheidung bezüglich der Belegung einer Variable getroffen werden, so verwenden wir die Methode `nextBoolean()`. Der in [HSSW02] vorgestellte Algorithmus legt verschiedene Wahrscheinlichkeiten für die Erstellung einer Belegung zu Grunde. Hierzu nutzen wir die Methode `nextDouble()`, welche eine Zahl im Intervall $[0, 1)$ liefert.

5 Laufzeitanalyse

Im folgenden Abschnitt wird die Konfiguration des verwendeten Testsystems vorgestellt. Anschließend beschreiben wir die Testdaten und wie diese erzeugt wurden. Wir erklären, wie wir die Laufzeitmessungen unternommen haben und wie diese Testergebnisse verwendet wurden, um daraus eine mit den theoretisch bestimmten Werten vergleichbare Funktion zu interpolieren. Abschließend stellen wir die bestimmten Laufzeiten der fünf Algorithmen Brute-Force, BSat, CoveringSearch, ResolveSat und Red2 / RW vor.

5.1 Testsystem

CPU: 2 × AMD Opteron (246) 2 GHz

RAM: 2 GB (DDR400)

Board: Tyan Thunder K8W

OS: Fedora Core 3, angepasste Installation, Kernel 2.6.11-1.35_FC3smp

Java: J2SE Development Kit 5.0 Update 4 (Linux AMD64 Platform)

Test Umgebung: Konsole, neben den per Default vorhandenen Anwendungen liefen noch ein LDAP- sowie ein DHCP-Daemon (slapd bzw. dhcpd). Beide sind keiner nennenswerten Last ausgesetzt. Auf Grund der Implementierung beanspruchen die Algorithmen nur eine der beiden CPUs.

5.2 Testablauf

Wir verwenden „zufällig“ erzeugte 3-SAT-Formeln, die wie folgt erstellt wurden: Für $n, k \in \mathbb{N}$ werden in jede der k Klauseln drei der $2n$ möglichen Literale pseudozufällig eingefügt, sodass jedes Literal mit der selben Wahrscheinlichkeit von $\frac{1}{2}n$ gewählt wird. Klauseln mit mehren gleichen Literalen, sowie Klauseln mit einem Literal und seiner Negation werden nicht übernommen. Jedes n und k ergibt eine bestimmte Verteilung von Random-3-Sat-Instanzen. „Uniform Random-3-SAT“ ist die Vereinigung aller dieser Verteilungen über alle n und k.

```
1 # include <sys/time.h>
2 jlong os::javaTimeNanos() {
3   timeval time;
4   jlong usecs = jlong(time.tv_sec) * (1000 * 1000) + jlong(time.tv_usec);
5   return 1000 * usecs;
6 }
```

Abbildung 5.1: Methode `javaTimeNanos`() (Auszug)

Bei „erzwungenen“ (engl. *forced*) Instanzen wird eine Belegung θ zu Grunde gelegt, auf deren Basis die Formel erstellt wird, sodass diese Belegung erfüllend ist. Dadurch ist gewährleistet, dass mindestens eine erfüllende Belegung vorhanden ist und die Formel damit in SAT liegt. Nicht erzwungene (engl. *unforced*) Instanzen haben keine vorgegebene Belegung und werden nach ihrer Erzeugung auf Erfüllbarkeit geprüft (vergl. [D93b, Uniform Random-3-SAT Description]). Die hier verwendeten „Uniform Random-3-SAT“-Formeldaten der DIMACS-Challenge [D93b] sind unforced.

In Tabelle 5.1 sind die getesteten Formeln aufgelistet. Nicht alle Algorithmen können mit der selben Anzahl von Formeln getestet werden. Der Brute-Force-Algorithmus benötigt bereits bei 50 Variablen mehr als einen Tag um ausgeben zu können, ob eine Formel erfüllbar ist oder nicht. Der randomisierte Algorithmus ResolveSat braucht bei einer Variablenanzahl von 20 schon einen Wiederholungswert i von 100.000 um zuverlässig eine erfüllbare Formel erkennen zu können. Der deterministische Algorithmus BSat hat hingegen keine vergleichbaren „Schwierigkeiten“ mit Formeln mit 100–120 Variablen und dementsprechend hohen Klauselanzahlen. Aus diesem Grund benötigen wir ein entsprechendes Spektrum an Testdaten.

Für Formeln mit einer Variablenanzahl von größer 20 verwenden wir die oben beschriebenen „Uniform Random-3-SAT“-Formeln. Damit Formeln mit 5, 10 und 15 Variablen vorhanden sind, implementieren wir eine Klasse, mit der pseudozufällige und unforced 3-KNF-Instanzen erzeugt werden können. Wir erzeugen hiermit jeweils 1.000 Instanzen und überprüfen diese dann anschließend auf ihre Erfüllbarkeit. Danach sortieren wir die vorhandenen Formeldaten und haben geeignete Testdaten vorliegen. Im Testablauf über die Algorithmen testen wir erfüllbare und unerfüllbare Formeln jeweils getrennt voneinander.

Zur Laufzeitberechnung benutzen wir die Methode `nanoTime()` aus der Standardklasse `java.lang.System`. Diese wird von der Java-VM von Sun unter Linux über die in Abbildung 5.1 dargestellte Methode bestimmt, welche eine Genauigkeit von einer Mikrosekunde hat (siehe auch Linux Kernel 2.6.11, Klasse `time.c` in der x86_64 Architektur). Für die randomisierten Algorithmen summieren wir über die Laufzeiten, die notwendig sind um eine erfüllbare Formel als „erfüllbar“ zu identifizieren. Wir verwenden eine selbst implementierte Klasse `StopTime`, welche die nötigen Methoden zum Zeitmessen und Ausgeben der Werte bereitstellt.

Algorithmus	**#Variablen**	**#Klauseln**	**#Formeln**
Brute-Force	5	28	309
	10	50	479
	15	70	431
	20	91	1000
BSat	5	28	309
	10	50	479
	15	70	431
	20	91	1000
	50	218	1000
	75	325	100
	100	430	1000
	125	538	100
CoveringSearch	5	28	309
	10	50	479
	15	70	431
	20	91	1000
ResolveSat	5	28	309
	10	50	479
	15	70	431
	20	91	1000
Red2 / RW	5	28	309
	10	50	479
	15	70	431
	20	91	1000
	50	218	1000

Tabelle 5.1: Testfälle

Mit den erhaltenen Laufzeiten berechnen wir das arithmetische Mittel und erhalten eine Punktemenge (x, y), wobei $x \in \{5, 10, 15, 20, 50, 75, 100, 125\}$ die Variablenanzahl und $y \in \mathbb{R}^+$ die Laufzeit in Millisekunden repräsentiert.

Da die zu den untersuchten Algorithmen gehörenden theoretischen Laufzeiten exponentiell sind, suchen wir jeweils eine Exponentialfunktion g der Form

$$g(x) = b \cdot \mathrm{e}^{c \cdot x}$$

für geeignete $b, c \in \mathbb{R}^+$, die die Laufzeiten beschreibt. Hierzu logarithmieren wir die Werte $f(x) := \ln(g(x))$ sowie $d := \ln(b)$ und erhalten damit eine linearisierte Gleichung

$$f(x) = \ln(g(x)) = \ln\left(b \cdot e^{c \cdot x}\right) = d + c \cdot x.$$

Wir interpolieren über die Methode kleinster Quadrate (siehe hierzu etwa [Bjö96]) eine Ausgleichsgerade und formen die erhaltenen Parameter in die Exponentialwerte um. Die so bestimmte Exponentialfunktion g kann direkt mit der in O-Notation angegebenen Laufzeitfunktion f verglichen werden, um den polynomiellen Faktor p in

$$g(x) = p(x) \cdot f(x)$$

zu bestimmen. Dieses Polynom approximieren wir mittels Taylorpolynomen.

5.3 Der Brute-Force-Algorithmus

		mittlere Laufzeit in ms	
#Variablen	**#Klauseln**	**erfüllbare Formeln**	**unerfüllbare Formeln**
5	28	0,162	0,417
10	50	3,782	5,664
15	70	81,364	267,367
20	91	2673,723	-

Tabelle 5.2: Laufzeiten von Brute-Force

Theoretische Laufzeit: $\mathrm{O}(2^n)$

Interpolierte Laufzeit für

erfüllbare Formeln: $0{,}00609 \cdot 1{,}90430^n \in \mathrm{O}(1{,}90430^n)$

unerfüllbare Formeln: $0{,}01340 \cdot 1{,}90830^n \in \mathrm{O}(1{,}90830^n)$

Polynomfaktor (erfüllbare Formeln):

$$\begin{aligned}
&0{,}1343156340 \cdot 10^{-16} \cdot x^9 \\
-&0{,}6305277359 \cdot 10^{-16} \cdot x^8 \\
-&0{,}5153301363 \cdot 10^{-15} \cdot x^7 \\
+&0{,}1162442392 \cdot 10^{-12} \cdot x^6 \\
-&0{,}1437859597 \cdot 10^{-10} \cdot x^5 \\
+&0{,}1466712555 \cdot 10^{-8} \cdot x^4 \\
-&0{,}1196529066 \cdot 10^{-6} \cdot x^3 \\
+&0{,}7320804408 \cdot 10^{-5} \cdot x^2 \\
-&0{,}000298609104 \cdot x \\
+&0{,}00609
\end{aligned}$$

5.4 Der Algorithmus von Monien und Speckenmeyer

		mittlere Laufzeit in ms	
#Variablen	**#Klauseln**	**erfüllbare Formeln**	**unerfüllbare Formeln**
5	28	0,968	1,544
10	50	2,472	3,056
15	70	4,681	8,429
20	91	14,779	-
50	218	455,495	1339,363
75	325	5947,930	13124,270
100	430	36784,663	99789,025
125	538	328622,620	853956,050

Tabelle 5.3: Laufzeiten von BSat

Theoretische Laufzeit: $\mathrm{O}(1{,}6181^n)$

Interpolierte Laufzeit für

erfüllbare Formeln: $1{,}13877 \cdot 1{,}11098^n \in \mathrm{O}(1{,}11098^n)$

unerfüllbare Formeln: $1{,}56587 \cdot 1{,}11803^n \in \mathrm{O}(1{,}11803^n)$

Polynomfaktor (erfüllbare Formeln):

$$\begin{aligned}
&-0{,}4714885689 \cdot 10^{-9} \cdot x^9 \\
&+0{,}1128527777 \cdot 10^{-7} \cdot x^8 \\
&-0{,}2401055891 \cdot 10^{-6} \cdot x^7 \\
&+0{,}4469929760 \cdot 10^{-5} \cdot x^6 \\
&-0{,}7132674507 \cdot 10^{-4} \cdot x^5 \\
&+0{,}9484684280 \cdot 10^{-3} \cdot x^4 \\
&-0{,}0100898182 \cdot x^3 \\
&+0{,}0805017038 \cdot x^2 \\
&-0{,}4281890360 \cdot x \\
&+1{,}138770000
\end{aligned}$$

5.5 Der Algorithmus von Dantsin, Goerdt, Hirsch und Schöning

		mittlere Laufzeit in ms	
#Variablen	**#Klauseln**	**erfüllbare Formeln**	**unerfüllbare Formeln**
5	28	0,231	1,188
10	50	5,186	18,777
15	70	48,137	268,406
20	91	1475,654	-

Tabelle 5.4: Laufzeiten von CoveringSearch

Theoretische Laufzeit $\mathrm{O}(1{,}733^n)$

Interpolierte Laufzeit für

erfüllbare Formeln: $0{,}01371 \cdot 1{,}76869^n \in \mathrm{O}(1{,}76869^n)$

unerfüllbare Formeln: $0{,}08034 \cdot 1{,}71952^n \in \mathrm{O}(1{,}71952^n)$

Polynomfaktor (erfüllbare Formeln):

$$\begin{aligned}
&-0{,}2283171762 \cdot 10^{-17} \cdot x^9 \\
&+0{,}3001813658 \cdot 10^{-16} \cdot x^8 \\
&-0{,}2500583832 \cdot 10^{-15} \cdot x^7 \\
&+0{,}2796814774 \cdot 10^{-14} \cdot x^6 \\
&+0{,}3977984606 \cdot 10^{-12} \cdot x^5 \\
&+0{,}9864080671 \cdot 10^{-10} \cdot x^4 \\
&+0{,}1935667189 \cdot 10^{-7} \cdot x^3 \\
&+0{,}2848624315 \cdot 10^{-5} \cdot x^2 \\
&+0{,}2794803914 \cdot 10^{-3} \cdot x \\
&+0{,}01371
\end{aligned}$$

5.6 Der Algorithmus von Paturi, Pudlák, Saks und Zane

		mittlere Laufzeit in ms	
#Variablen	**#Klauseln**	**erfüllbare Formeln**	**unerfüllbare Formeln**
5	28	5,582	0,343
10	50	589,953	0,633
15	70	23470,845	0,763
20	91	852075,155	-
50	218	-	10,482
75	325	-	24,430

Tabelle 5.5: Laufzeiten von ResolveSat

Theoretische Laufzeit: $\mathrm{O}(1{,}362^n)$

Interpolierte Laufzeit für

erfüllbare Formeln: $0{,}14689 \cdot 2{,}203^n \in \mathrm{O}(2{,}203^n)$

unerfüllbare Formeln: $0{,}55913 \cdot 1{,}04014^n \in \mathrm{O}(1{,}04014^n)$

Polynomfaktor (erfüllbare Formeln):

$$\begin{aligned}
&0{,}5564738294 \cdot 10^{-9} \cdot x^9 \\
+&0{,}1041509549 \cdot 10^{-7} \cdot x^8 \\
+&0{,}1732723644 \cdot 10^{-6} \cdot x^7 \\
+&0{,}2522338730 \cdot 10^{-5} \cdot x^6 \\
+&0{,}3147246112 \cdot 10^{-4} \cdot x^5 \\
+&0{,}3272478246 \cdot 10^{-3} \cdot x^4 \\
+&0{,}0027221548 \cdot x^3 \\
+&0{,}0169828325 \cdot x^2 \\
+&0{,}0706343862 \cdot x \\
+&0{,}14689
\end{aligned}$$

5.7 Der Algorithmus von Hofmeister, Schöning, Schuler und Watanabe

		mittlere Laufzeit in ms	
#Variablen	**#Klauseln**	**erfüllbare Formeln**	**unerfüllbare Formeln**
5	28	0,373	0,447
10	50	0,682	0,587
15	70	3,530	0,963
20	91	18,789	-
50	218	10116,073	7,971
75	325	-	26,320
100	430	-	30,546
125	538	-	50,560

Tabelle 5.6: Laufzeiten von RED2 / RW

Theoretische Laufzeit $\mathrm{O}(1{,}3302^n)$

Interpolierte Laufzeit für

erfüllbare Formeln: $0{,}10927 \cdot 1{,}26008^n \in \mathrm{O}(1{,}26008^n)$

unerfüllbare Formeln: $0{,}64584 \cdot 1{,}03637^n \in \mathrm{O}(1{,}03637^n)$

Polynomfaktor (erfüllbare Formeln):

$$\begin{aligned}
&-0{,}1294360418 \cdot 10^{-17} \cdot x^9 \\
&+0{,}2016594962 \cdot 10^{-15} \cdot x^8 \\
&-0{,}2963007476 \cdot 10^{-13} \cdot x^7 \\
&+0{,}3828090549 \cdot 10^{-11} \cdot x^6 \\
&-0{,}4241102629 \cdot 10^{-9} \cdot x^5 \\
&+0{,}3915751434 \cdot 10^{-7} \cdot x^4 \\
&-0{,}2892301919 \cdot 10^{-5} \cdot x^3 \\
&+0{,}1602262124 \cdot 10^{-3} \cdot x^2 \\
&-0{,}5917418059 \cdot 10^{-2} \cdot x \\
&+0{,}10927
\end{aligned}$$

5.8 Schlussbemerkung

Algorithmus	theoretische Laufzeit	ermittelte Laufzeit für erfüllbare Formeln	ermittelte Laufzeit für unerfüllbare Formeln
Brute-Force	$O(2^n)$	$O(1{,}9043^n)$	$O(1{,}9083^n)$
BSat	$O(1{,}6181^n)$	$O(1{,}11098^n)$	$O(1{,}11803^n)$
CoveringSearch	$O(1{,}733^n)$	$O(1{,}76869^n)$	$O(1{,}71952^n)$
ResolveSat	$O(1{,}362^n)$	$O(2{,}203^n)$	$O(1{,}04014^n)$
Red2 / RW	$O(1{,}3302^n)$	$O(1{,}26008^n)$	$O(1{,}03637^n)$

Tabelle 5.7: Zusammenfassung der Laufzeitanalyse, obere Hälfte deterministische Algorithmen, untere Hälfte randomisierte Algorithmen

Auffällig sind bei den randomisierten Algorithmen die stark unterschiedlichen Laufzeiten für erfüllbare und unerfüllbare Formeln. Wir nehmen an, dass auf Grund der vorhandenen Erfolgs- bzw. Misserfolgsrate, die bei randomisierten Algorithmen immer vorhanden ist, zwangsweise unerfüllbare Formeln mit größerer Wahrscheinlichkeit korrekt klassifiziert werden als erfüllbare Formeln. Genau deswegen beinhalten diese Algorithmen immer eine gewisse Wiederholungszahl (ResolveSat in Algorithmus 3.8 Zeile 1–8, Red2 / RW in Algorithmus 3.14 Zeile 2–8) damit ein korrektes Ergebnis wahrscheinlicher wird. Aus diesem Grund betrachten wir im folgenden nur die ermittelten Laufzeitergebnisse für erfüllbare Formeln.

Insgesamt erkennt man, dass der deterministische Algorithmus BSat der effizienteste von den hier getesteten Algorithmen in dem Bereich von 20–125 Variablen ist.

Der Algorithmus ResolveSat besitzt eine viel schlechtere Laufzeit als die theoretisch ermittelte. Dies kann an der Qualität der hier verwendeten Pseudozufallszahlen liegen, wird in dieser Arbeit jedoch nicht weiter untersucht. Die hier interpolierte Laufzeit vom Algorithmus CoveringSearch übersteigt die theoretisch bestimmte Laufzeit in der zweiten Nachkomma-Stelle. Alle anderen Algorithmen erreichen Ergebnisse die unter der theoretisch bestimmten Schranke liegen. Wir vermuten, dass für eine asymptotische Betrachtung der Variablenanzahlen, die theoretische Laufzeit erreicht bzw. überschritten wird.

Literaturverzeichnis

[Bjö96] Åke Björk. *Numerical methods for least squares problems. SIAM, Philadelphia*, 1996.

[Cha04] Tom Chang. *Horn Formula Minimization.* Diplomarbeit, Rochester Institute of Technology – Department of Computer Science, http://www.cs.rit.edu:8080/ms/static/eh/2003/3/tyc0429/report.ps, Mai 2004.

[Coo71] Stephen C. Cook. *The complexity of theorem-proving procedures. Third ACM Symposium on Theory of Computing, ACM, New York*, Seiten 151–158, 1971.

[D93a] *Satisfiability suggested format.* http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/satformat.ps.

[D93b] *SATLIB – Benchmark Problems, Uniform Random-3-SAT.* http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html.

[DC92] *The second DIMACS challenge.* http://mat.gsia.cmu.edu/challenge.html, September 1992.

[DGHS00] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch und Uwe Schöning. *Deterministic algorithms for k-SAT based on covering codes and local search. Proc. 27th International Colloquium on Automata, Languages and Programming, ICALP'00, Lecture Notes in Comp. Sci. 1853*, Seiten 236–247, 2000.

[EFT98] Heinz-Dieter Ebbinghaus, Jörg Flum und Wolfgang Thomas. *Einführung in die mathematische Logik.* Spektrum Akademischer Verlag, 4., überarbeite Aufl., 1998.

[Grü05] Rudolf Grübel. *Stochastische Algorithmen, Vorlesungsskript mit Aufgaben und Lösungen.* Universität Hannover, Institut für Mathematische Stochastik, http://www.stochastik.uni-hannover.de/SALGO_WS04/stochalg2.pdf, WS 2004/2005.

[HMU02] John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und*

Komplexitätstheorie. Addison Wesley Longman, Pearson Education Deutschland GmbH, 2. Aufl., 2002.

[HSSW02] Thomas Hofmeister, Uwe Schöning, Rainer Schuler und Osamu Watanabe. *A probabilistic 3-SAT algorithm further improved. H. Alt and A. Ferreira (Eds.): STACS 2002, LNCS 2285*, Seiten 192–202, 2002.

[Knu98] Donald E. Knuth. *Seminumerical Algorithms*, Bd. 2 von *The Art of Computer Programming*. Addison Wesley, 3. Aufl., 1998.

[MS85] Burkhard Monien und Ewald Speckenmeyer. *Solving satisfiability in less than* 2^n *steps. Discrete Applied Mathematics*, *Bd. 10*:287–295, 1985.

[PPSZ98] Ramamohan Paturi, Pavel Pudlák, Michael E. Saks und Francis Zane. *An improved exponential-time algorithm for k-SAT. Proc. of the 39th Ann. IEEE Sympos. on Foundations of Comp. Sci. (FOCS'98), IEEE*, Seiten 628–637, 1998.

[Rau02] Wolfgang Rautenberg. *Einführung in die mathematische Logik*. Vieweg Verlag, 2., überarbeitete Aufl., 2002.

[Sch03] Uwe Schöning. *Stochastische Algorithmen für das Erfüllbarkeitsproblem*. Vortrag am Institut für Informationssysteme, Fachgebiet Theoretische Informatik, Universität Hannover, Januar 2003.

Tabellenverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis