Vorlesungsskript

SAT-Algorithmen

Sommersemester 2014

Dr. Arne Meier

Institut für Theoretische Informatik
Leibniz Universität Hannover

Leibniz Universität Hannover

Version vom 2. Juni 2014

Inhaltsverzeichnis

Vorwort **v**

1 Codierungen und Kombinatorik **1**
1.1 Negationsnormalform . . . 1
1.2 Tseitin-Codierung . . . 2
1.3 Serien-Parallele Graphen . . . 4
1.4 Autarke Belegungen . . . 8
1.5 Kombinatorische Beobachtungen . . . 10

2 Spezialfälle **15**
2.1 Krom-SAT . . . 15
2.2 Horn-Formeln . . . 18

3 Ein Meta-DPLL-Algorithmus **21**
3.1 Probabilistische Varianten . . . 22
3.2 Klauselllernen . . . 25
3.3 Nicht-chronologisches Backtracking . . . 26

4 Baumartige Resolution und das Pigeonhole-Prinzip **29**

5 Lokale Suche **35**
5.1 Deterministische lokale Suche . . . 36
5.2 Random Walk . . . 37
5.3 Greedy Algorithmen und Scores . . . 39

6 Ein Divide and Conquer Ansatz **41**

7 Enumeration **45**

Vorwort

Das Erfüllbarkeitsproblem in der Aussagenlogik (SAT) ist das wichtigste Problem in der theoretischen Informatik. Cook (1971) wies die NP-Vollständigkeit von SAT nach; unabhängig hierzu erreichte Levin (1973) ein analoges Resultat. Diese Arbeiten waren das erste Stück in einem großen und bisher noch unvollständigen Puzzle um die Frage wie die Komplexitätsklassen P und NP zueinander stehen: sind beide Klassen gleich oder nicht?

Diese Frage sucht nach einer Antwort danach, ob komplexe und große Suchräume in polynomieller Laufzeit durchforstet werden können oder nicht. Ist es für ein Transportunternehmen möglich optimale Routen in kurzer Zeit zu bestimmen? Können Mobilfunkmasten optimal aufgestellt werden ohne, dass sich Frequenzen ungünstig überlagern? Kann ein Versandhandel seine Pakete für eine Bestellung mehrerer Gegenstände kostenoptimal verpacken? Alle diese Fragen sind mit 'Ja' zu beantworten, wenn es einen Polynomialzeit-Algorithmus gibt, der die Erfüllbarkeit von aussagenlogischen Formeln entscheidet.

Aktuell ist nicht bekannt, ob ein solcher Algorithmus existiert. Es sind lediglich Exponentialzeit-Algorithmen für SATvorhanden. Ein großer Teil der Wissenschaftler, die in diesem Bereich der theoretischen Informatik arbeiten – mich eingeschlossen – vermuten, dass die Klassen eher nicht gleich sind. Jedoch ist dies scheinbar kein wirkliches Problem für die Gemeinschaft, da die Forschung aktuell sehr gute SAT-Solver geliefert hat. Es geht sogar so weit, dass oft Probleme aus anderen Bereichen, in denen es keine wirklich guten Algorithmen gibt, diese in Aussagenlogik formulieren um in den Genuss der SAT-Solver zu kommen, da diese so „performant“ laufen – performant bedeutet hier natürlich nicht polynomiell, aber dennoch in annehmbarer Zeit für einen Exponentialzeitalgorithmus. Es geht sogar so weit, dass man schon häufig in Vorträgen hört, dass SAT industriell schon effizient gelöst wird; Instanzen mit mehreren Tausend an Variablen stellen heutzutage kein wirkliches Problem mehr dar. Dies zeigt, dass die Wissenschaft hier ein enormes algorithmisches Know-How entwickelt hat. Allerdings kann das immer fortgeführte Optimieren vorhandener Algorithmen auch zu einer Art Einbahnstraße werden, sodass keine neuen Ideen entstehen, wie Moshe Vardi vor einigen Jahren auf der SAT in Swansea angesprochen hat.

Moshe Vardi, *1954,

In diesem Forschungsbereich wurde viel Arbeit in die Entwicklung und Optimierung der SAT-Algorithmen gesteckt. Es gibt jährlich auf der SAT-Konferenz Wettbewerbe bei denen die aktuellsten Stände der Algorithmen gegeneinander antreten und immer wieder neue Fortschritte erzielen.

In dieser Vorlesung wollen wir uns intensiv mit diesen Algorithmen beschäftigen. Wir möchten die verschiedenen Ansätze und ihre angewendeten Tricks kennenlernen. Hierzu werden wir uns im großen an das neue Buch von Uwe Schöning und Jacobo Torán *Das Erfüllbarkeitsproblem SAT – Algorithmen und Analysen* halten.

Arne Meier, 15. November 2013

Kapitel 1

Codierungen und Kombinatorik

Wir benutzen die üblichen Grundlagen, die aus Logik-Vorlesungen bekannt sind. Variablen oder Propositionen werden über Kleinbuchstaben u, v, w, x, y, z gekennzeichnet. Für die beiden Konstanten wahr und falsch verwenden wir 1 und 0. Literale sind Variablen oder Negationen von ihnen. Klauseln sind Mengen von Literalen bzw. eine Disjunktionen von ihnen. Eine Formel in k-KNF für ein $k \in \mathbb{N}$ ist eine Konjunktion von Klauseln mit je k Literalen. Die üblichen binären Konnektoren sind $\wedge, \vee, \rightarrow, \leftrightarrow$. Der einzige unäre Konnektor ist die Negation $\neg$ oder auch $\overline{\cdot}$. Belegungen sind Funktionen, die Variablen auf Wahrheitswerte abbilden und werden üblicherweise mit θ, α, β bezeichnet. Für Formeln verwenden wir griechische Klein- und manchmal auch Großbuchstaben $\phi, \varphi, \psi, \chi, \Phi$. Die Menge aller aussagenlogischen Formeln wird mit $\mathcal{L}$ abgekürzt. Für eine Belegung θ bezeichnet $\mathrm{Vars}(\theta)$ die Menge der Variablen, denen θ Werte zuweist. Für eine Formel ϕ bezeichnet $\mathrm{Vars}(\phi)$ die Menge der Variablen von ϕ. Für $\phi \in \mathcal{L}$ und eine Belegung θ schreiben wir $\theta \models \phi$ wenn die Ersetzung der Variablen in ϕ durch die von θ bestimmten Wahrheitswerte zu wahr evaluiert. Diese Ersetzung schreiben wir kurz mit $\phi[\theta]$. Ist $\mathrm{Vars}(\theta) \subset \mathrm{Vars}(\phi)$, so sei $\phi[\theta]$ die Vereinfachung von ϕ anhand von θ, d. h. die Formel wird so weit vereinfacht, bis keine Konstanten mehr enthalten sind. SAT bezeichnet die Menge aller erfüllbaren Formeln in $\mathcal{L}$. Seien $f(n), g(n)$ Funktionen. Dann gilt $f \in O^*(g)$, wenn es $c, n_0 \in \mathbb{N}$ und ein Polynom p gibt, sodass für alle $n \geq n_0$ gilt $f(n) \leq p(n) \cdot g(n)$.

1.1 Negationsnormalform

Wir folgen Johannsen (2005).

Negationsnormalform

Definition 1. Jedes Literal und jede Konstante ist eine Formel in NNF. Sind ϕ und ψ Formeln in NNF, so sind auch $\phi \vee \psi$, sowie $\phi \wedge \psi$ Formeln in NNF.

Für eine Formel ϕ in NNF definieren wir die negierte Formel $\overline{\phi}$ in NNF via:

- Ist $\phi = 0$, so ist $\overline{\phi} = 1$. Ist $\phi = 1$, so ist $\overline{\phi} = 0$.
- Ist $\phi = x$, so ist $\overline{\phi} = \neg x$.
- Ist $\phi = \neg x$, so ist $\overline{\phi} = x$.
- Ist $\phi = \psi_1 \vee \psi_2$, so ist $\overline{\phi} = \overline{\psi_1} \wedge \overline{\psi_2}$.
- Ist $\phi = \psi_1 \wedge \psi_2$, so ist $\overline{\phi} = \overline{\psi_1} \vee \overline{\psi_2}$. ◀

Lemma 1. Für jede Formel ϕ in NNF ist $\overline{\phi}$ äquivalent zu $\neg\phi$. ◀

Beweis Beweis durch Induktion über den Formelaufbau.

Induktionsanfang: Ist $\phi = 0$, so ist $\overline{\phi} = 1 \equiv \neg 0$. Ist $\phi = 1$, so ist $\overline{\phi} = 0 \equiv \neg 1$. Ist $\phi = x$, so ist $\overline{\phi} = \neg x = \neg x$. Ist $\phi = \neg x$, so ist $\overline{\phi} = x \equiv \neg\neg x = \neg\phi$.

Induktionsschritt: Ist $\phi = \psi \vee \chi$, so ist $\overline{\phi} = \overline{\psi} \wedge \overline{\chi}$. Dies ist nach IH äquivalent zu $\neg\psi \wedge \neg\chi$ und nach De Morgan wieder äquivalent zu $\neg(\psi \vee \chi) = \neg\phi$. Ist $\phi = \psi \wedge \chi$, so ist $\overline{\phi} = \overline{\psi} \vee \overline{\chi} \equiv \neg\psi \vee \neg\chi$ nach IH und dies ist äquivalent zu $\neg(\psi \wedge \chi) = \neg\phi$. ■

Lemma 2. Für jede Formel ϕ gibt es eine äquivalente Formel $n(\phi)$ in NNF. ◀

Beweis Induktion über Formelaufbau.

Induktionsanfang: Ist $\phi = x, \phi = 0$, oder $\phi = 1$, so ist ϕ bereits in NNF, also setze $n(\phi) = \phi$.

Induktionsschritt: Ist $\phi = \psi \circ \chi$ und $\circ \in \{\wedge, \vee\}$, so gibt es nach IH $n(\psi) \equiv \psi$ und $n(\chi) \equiv \chi$ in NNF und setze $n(\phi) = n(\psi) \circ n(\chi)$. Damit erhalten wir $n(\phi) = n(\psi) \circ n(\chi) \equiv \psi \circ \chi = \phi$.

Ist nun $\phi = \neg\psi$, so gibt es nach IH $n(\psi) \equiv \psi$ in NNF. Definiere $n(\phi) = n(\overline{\psi})$. Aus Lemma 1 folgt $n(\phi) = n(\overline{\psi}) \equiv \neg n(\psi) \equiv \neg\psi = \phi$. ■

1.2 Tseitin-Codierung

Es ist bekannt, dass die Umformung einer beliebigen aussagenlogischen Formel in eine konjunktive Normalform die Formellänge um einen exponentiellen Faktor vergrößern kann. Häufig arbeiten Algorithmen mit Formeln in dieser Normalform. Jedoch werden wir im folgenden ein Verfahren sehen, mit dem eine Formel in eine erfüllbarkeitsäquivalente Formel umgeformt wird.

Viele Algorithmen arbeiten mit KNF-Formeln, in welchen Klauseln der Größe 1 auftauchen können. Solche nennen wir *Unit-Klauseln* (oder auch *Einheitsklauseln*). Tauchen Unit-Klauseln in Formeln auf so können wir *unit propagation* anwenden: ist $\{u\}$ die Unit-Klausel in einer Formel φ, dann muss u zwingend auf wahr gesetzt werden. Folglich kann man $\varphi[u = 1]$ weiter untersuchen, da φ und $\varphi[u = 1]$ erfüllbarkeitsäquivalent sind. Hierdurch kann natürlich die leere Formel $\emptyset$ bzw. die leere Klausel $\square$ auftreten. $\emptyset$ ist äquivalent zu wahr und $\square$ ist äquivalent zu falsch. Ist θ eine Belegung, dann sei $\phi[\theta]$ die Formel, die entsteht, wenn man θ auf ϕ anwendet.

Definition 2. Eine *Basis* B ist eine endliche Menge von Boole'schen Funktionen und Familien Boole'scher Funktionen. ◀

Definition 3. Sei B eine Basis. Ein *Boole'scher Schaltkreis* über B mit n Eingängen und m Ausgängen ist ein 5-Tupel $C = (V, E, \alpha, \beta, \omega)$, wobei

- (V, E) ein endlicher, gerichteter, azyklischer Graph ist,
- $\alpha\colon E \to \mathbb{N}$ eine injektive Funktion ist, die den Kanten eine Reihenfolge gibt,

- $\beta\colon V \to B_2 \cup \{x_1, \ldots, x_n\}$ jedem Knoten einen Funktion zuordnet und
- $\omega\colon V \to \{y_1, \ldots, y_m\}$ eine partielle Funktion ist, die festlegt welches die Ausgänge sind. ◀

Satz 1. Zu jedem Schaltkreis C mit genau einem Ausgang gibt es eine erfüllbarkeitsäquivalente Formel ψ in 3KNF. Die Umformung von C nach ψ ist in Polynomialzeit möglich. ◀ Tseitin

Beweis Gegeben ist ein Boole'scher Schaltkreis $C = (V, E, \alpha, \beta, \omega)$ über Basis B mit n Eingangsgattern und einem Ausgang. Ohne Beschränkung der Allgemeinheit können wir von ausschließlich zweistelligen Gattern in B ausgehen.

Wir konstruieren nun in polynomieller Zeit eine Formel $\psi(x_1, \ldots, x_n, y)$ in 3KNF, sodass für alle $c_1, \ldots, c_n \in \{0, 1\}$ gilt:

$$C(c_1, \ldots, c_n) = 1 \quad \text{gdw.} \quad \psi(c_1, \ldots, c_n, y_1, \ldots, y_m) \text{ ist erfüllbar,}$$

wobei m die Anzahl der Gatter von C ist. Hierbei wird jedem Gatterausgang ein y_i zugeordnet; y_1 entspricht dem Ausgangsgatter.

Ist der Gatterausgang zu y_i gleichzeitig ein Gattereingang für ein anderes Gatter, so ist dieses Gatter auch mit y_i assoziiert. Gatter, in die Eingangsleitungen fließen sind mit jeweiligem x_j assoziiert. Nun definieren wir ψ wie folgt

$$\psi = (y_1) \wedge \bigwedge_{i=1}^{m} \big(z_{i,3} \leftrightarrow f_i(z_{i,1}, z_{i,2})\big),$$

wobei $z_{i,1}, z_{i,2} \in \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$ die Eingänge des i-ten Gatters sind und $z_{i,3} \in \{y_1, \ldots, y_m\}$ der Ausgang des i-ten Gatters ist.

Das bedeutet, die Formel ist erfüllbar, wenn den y-Gattern den Funktionswerten entsprechende Werte zugewiesen werden können und das Ausgangsgatter y_1 den Wert 1 hat.

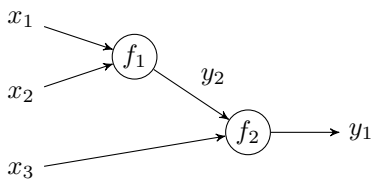
Die 3KNF bekommt man durch Ersetzen des Ausdrucks $f_i(z_{i,1}, z_{i2})$ mit der zugehörigen DNF über die Min-Terme und anschließender Ausmultiplikation. Dieser Ausdruck erzeugt maximal vier Klauseln (mit $\leq$ 3 Literalen) pro Konjunktionsglied. ■

Korollar 1. Zu jeder ausagenlogischen Formel φ gibt es eine erfüllbarkeitsäquivalente Formel ψ in 3KNF. Die Umformung von φ nach ψ ist in Polynomialzeit möglich. ◀

Beispiel 1. Gegeben sei die folgende Formel:

$$\varphi = f_2(f_1(x_1, x_2), x_3).$$

Hieraus ergibt sich der Schaltkreis



und die Boole'sche Funktion $\psi = (y_1) \wedge (y_2 \leftrightarrow f_1(x_1, x_2)) \wedge (y_1 \leftrightarrow f_2(y_2, x_3))$. ◀

Natürlich kann nach Anwendung des Verfahrens die immer auftretende Unit-Klausel nach dem üblichen Schema entfernt werden. Hieraus können wiederum weitere Unit-Klauseln entstehen, die eventuell die Formel weiter schrumpfen lassen.

Liegt φ in NNF vor dann existiert ein *monotoner* Schaltkreis existiert, d. h. nur $\wedge$- und $\vee$-Gatter Verwendung finden. Vor den Eingängen stehen dann ausschließlich Literale. Der Nutzen hiervon sind einfachere Umformungen der Formel $(z_{i,3} \leftrightarrow f_i(z_{i,1}, z_{i,2}))$ in KNF Ausdrücke wie folgt:

$$(z \leftrightarrow f(x,y)) = \begin{cases} (\overline{z} \vee x) \wedge (\overline{z} \vee y) \wedge (z \vee \overline{x} \vee \overline{y}) & \text{, wenn } f = \wedge \\ (\overline{z} \vee x \vee y) \wedge (z \vee \overline{x}) \wedge (z \vee \overline{y}) & \text{, wenn } f = \vee \end{cases}.$$

1.3 Serien-Parallele Graphen

Das im folgenden Abschnitt betrachtete Verfahren beruht auf der Arbeit von Bubeck u. Kleine Büning (2009). Hierzu werden sogenannte serien-parallele Graphen definiert.

Definition 4. Ist $V = \{x, y\}$ eine Knotenmenge, α ein Label und $E = \{(x \to y : \alpha)\}$, dann ist $G = (V, E)$ ein *serien-paralleler Graph* (SP-Graph). Wir nennen x *Quelle* und y *Senke*.

Sind $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ serien-parallele Graphen, die die gleiche Quelle x und Senke y teilen und außerdem $(V_1 \setminus \{x, y\}) \cap (V_2 \setminus \{x, y\}) = \emptyset$ gilt, dann ist $G = (V_1 \cup V_2, E_1 \cup E_2)$ ein serien-paralleler Graph.

Sind $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ serien-parallele Graphen, bei denen z die Senke von G_1 und gleichzeitig die Quelle von G_2 ist und außerdem $(V_1 \setminus \{z\}) \cap (V_2 \setminus \{z\}) = \emptyset$ gilt, dann ist $G = (V_1 \cup V_2, E_1 \cup E_2)$ ein serien-paralleler Graph. ◀

Bemerkung. Serien-parallele Graphen sind gerichtete, azyklische Multigraphen mit genau einer Quelle und einer Senke, deren Kanten sich nie kreuzen. ◀

Semantisch verstehen wir einen SP-Graphen G durch die Verbindung mit einer aussagenlogischen Formel Φ_G. Die Kanten-Label $(u \to w : \alpha)$ sind Klauseln $(u \to w) \vee \alpha$ und es gibt zusätzlich genau zwei Einheitsklauseln, die durch Quelle und Senke entstehen.

Definition 5. Sei $G = (V, E)$ ein SP-Graph mit Quelle x, Senke y und inneren Knoten $z_1, \ldots, z_t$. Die zu G zugehörige Formel ist dann definiert als

$$\Phi_G = \exists x \exists y \exists z_1 \ldots \exists z_t x \wedge \neg y \wedge \bigwedge_{(u \to w:\alpha) \in E} \big((u \to w) \vee \alpha\big).$$

Manchmal schreiben wir auch kurz $\exists \vec{z}$ statt alle z_i Variablen zu nennen. ◀

Nun können wir einen interessanten Zusammenhang beweisen.

Pfad Semantik

Satz 2. Sei $G = (V, E)$ ein SP-Graph mit Quelle x, Senke y, und sei Φ_G die zugehörige Formel. Dann gilt

$$\Phi_G \equiv \bigwedge_{\text{Pfade } \pi \text{ von } x \text{ nach } y} \left(\bigvee_{(u \to w:\alpha) \in \pi} \alpha \right).$$

Welche Größe hat Φ_G allgemein?

Beweis Sei

$$\pi = (x \to z_1 : \alpha_1), (z_1 \to z_2 : \alpha_2), \ldots, (z_{t-1} \to z_t : \alpha_t), (z_t \to y : \alpha_{t+1})$$

ein beliebiger Pfad von der Quelle zur Senke. Dann muss Φ_G die Klauseln

$$x, \neg x \vee z_1 \vee \alpha_1, \neg z_1 \vee z_2 \vee \alpha_2, \ldots, \neg z_{t-1} \vee z_t \vee \alpha_t, \neg z_t \vee y \vee \alpha_{t+1}, \neg y$$

enthalten. Wenn die Label α_i entfernt werden führt dies zu einer unerfüllbaren Formel. Also impliziert die gesamte Klausel-Menge die Klausel $\alpha_1 \vee \cdots \vee \alpha_{t+1}$. Dies zeigt die Implikation von links nach rechts bei der Erfüllbarkeitsäquivalenz.

Für die Rückrichtung nehme die erfüllende Belegung θ für den rechten Teil. Diese Belegung wenden wir auf die freien Variablen von Φ_G an und vereinfachen die resultierende Formel, genannt Φ_G^*. Angenommen Φ_G^* ist unerfüllbar. Dann muss es eine Kette

$$x \to z_1, z_1 \to z_2, \ldots, z_{t-1} \to z_t, z_t \to y$$

in Φ_G^* geben, da ausschließlich $\neg y$ die einzige negative Klausel ist. Jedoch ist diese Kette ein Pfad von Quelle x zur Senke y, was jedoch der Annahme, dass die rechte Formel wahr ist, widerspricht. ■

AND-OR Semantik

Satz 3. Sei $G = (V_1 \cup V_2, E_1 \cup E_2)$ ein SP-Graph mit Quelle x und Senke y, der aus zwei kleineren SP-Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ mit zugehörigen Formeln $\Phi_{G_1} = \exists \alpha_1 \phi_{G_1}$ und $\Phi_{G_2} = \exists \alpha_2 \phi_{G_2}$ entstanden ist, wobei α_i der jeweilige Quantoren-Präfix ist $(i = 1, 2)$.

(i) Sind G_1 und G_2 an einer parallelen Verbindung angeordnet, bei der beide die gleiche Quelle x und Senke y teilen und außerdem $(V_1 \setminus \{x, y\}) \cap (V_2 \setminus \{x, y\}) = \emptyset$ gilt, dann ist $\Phi_G \equiv \exists \alpha \phi_{G_1} \wedge \phi_{G_2}$.

(ii) Sind G_1 und G_2 in einer seriellen Verbindung angeordnet, bei der z sowohl Senke von G_1 als auch Quelle von G_2 ist und außerdem $(V_1 \setminus \{z\}) \cap (V_2 \setminus \{z\}) = \emptyset$, dann ist $\Phi_G \equiv \exists \alpha \phi_{G_1} \vee \phi_{G_2}$. ◀

Beweis

(i) In Φ_G duplizieren wir die Einheitsklauseln für Quelle und Senke, sowie partitionieren die Kanten in zwei disjunkte Mengen:

$$\begin{aligned}
\Phi_G &= \exists x \exists y \exists \vec{z} x \wedge \neg y \wedge \bigwedge_{(u \to w:\ \beta) \in E} ((u \to w) \vee \beta) \\
&\equiv \exists x \exists y \exists \vec{z} x \wedge \neg y \wedge \bigwedge_{(u \to w:\ \beta) \in E_1} ((u \to w) \vee \beta) \wedge \\
&\qquad x \wedge \neg y \wedge \bigwedge_{(u \to w:\ \beta) \in E_2} ((u \to w) \vee \beta) \\
&\equiv \exists x \exists y \exists \vec{z} \phi_{G_1} \wedge \phi_{G_2}
\end{aligned}$$

(ii) Aus Satz 2 wissen wir, dass $\Phi_G \equiv \bigwedge_{\pi \text{ Pfad von } x \text{ nach } y} (\bigvee_{(u \to v:\ \alpha) \in \pi} \alpha)$ gilt. Hieraus folgt, dass jeder Pfad von x nach y durch z gehen muss. Betrachte einen einzelnen Pfad von x nach z. Ist Φ_G war, so ist auf diesem Pfad ein Label erfüllt oder

jeder Pfad von z nach y hat ein erfülltes Label. Da dies für jeden x-z-Pfad gilt, impliziert Φ_G die Formel

$$\left(\bigwedge_{\pi \text{ Pfad von } x \text{ nach } z}\left(\bigvee_{(u\to v\colon \alpha)\in\pi}\alpha\right)\right)\vee\left(\bigwedge_{\pi \text{ Pfad von } z \text{ nach } y}\left(\bigvee_{(u\to v\colon \alpha)\in\pi}\alpha\right)\right).$$

Die Implikation von rechts nach links ist klar: wenn alle Pfade in einer Hälfte es Graphen erfüllt sind, dann sind alle Pfade im gesamten Graphen erfüllbar. Also folgt, dass beide Formel erfüllbarkeitsäquivalent sind. Eine Anwendung von Satz 2 führt dazu, dass die rechte Formel äquivalent zu $\Phi_{G_1} \vee \Phi_{G_2}$ ist. Ziehen wir nun die Quantoren an den Anfang, folgt der Satz, da $(\exists v\phi_{G_1}) \vee (\exists v\phi_{G_2}) \equiv \exists v\phi_{G_1} \vee \phi_{G_2}$ gilt.

■

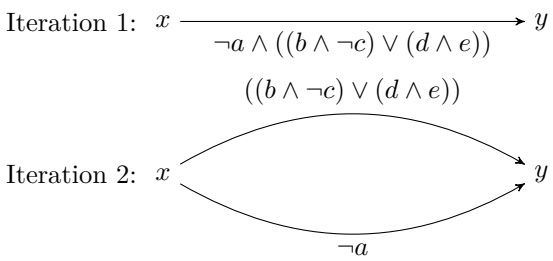
Nun können wir die folgende Idee verfolgen: ausgehend von einer Formel ψ in NNF konstruiere den SP-Graph G. Hieraus können wir nun die Formel $\Phi_G = \exists\alpha\phi$ erhalten, für die ϕ in 3KNF ist und $\Phi_G \equiv \psi$ gilt. Damit die Kantenlabel Klauseln darstellen, müssen die Label α Literale sein. Daher benötigen wir einen Weg eine Formel ψ auf einen äquivalenten SP-Graph $G = \mathrm{sp}(\psi)$ welcher nur mit Literalen gelabelt ist, abzubilden.

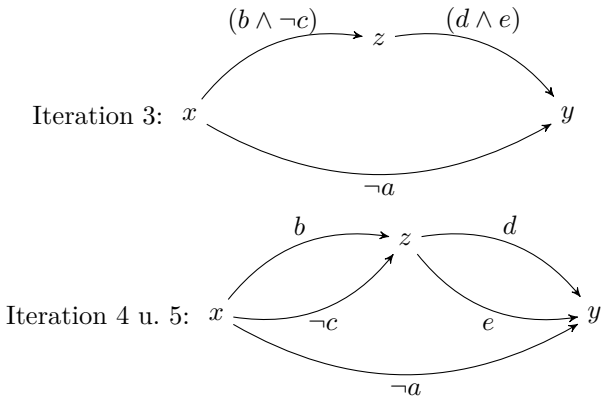
Dies führt nun zu dem folgenden top-down Verfahren: Wähle eine beliebige Kante, die mit ψ gelabelt ist. Abhängig davon ob ψ eine Konjunktion oder Disjunktion von Teilformeln α_1, α_2 ist, teilen wir diese Kante in zwei parallele oder serielle Kanten gelabelt mit α_1, α_2. Dies wird nun weitergeführt auf den neu entstandenen Kanten bis alle Label Literale sind.

Eingabe: Aussagenlogische Formel ψ in NNF
1: Setze $G := (V, E)$ mit $V = \{x, y\}$ und $E = \{(x \to y\colon \psi)\}$
2: **while** in G gibt es eine Kante $(u \to w\colon \alpha)$ und α ist kein Literal **do**
3: **if** $\alpha = \alpha_1 \wedge \alpha_2$ **then**
4: $E := (E \setminus \{(u \to w\colon \alpha)\}) \cup \{u \to w\colon \alpha_1, u \to w\colon \alpha_2\}$
5: **else if** $\alpha = \alpha_1 \vee \alpha_2$ **then**
6: $E := (E \setminus \{(u \to w\colon \alpha)\}) \cup \{u \to z_i\colon \alpha_1, z_i \to w\colon \alpha_2\}$
7: $V := V \cup \{z_i\}$ für neue Variable z_i
8: $\Phi_G := \exists x\exists y\exists z_1 \cdots \exists z_t x \wedge \neg y \wedge \bigwedge_{(u\to w\colon \alpha)\in E}(\neg u \vee w \vee \alpha)$
Ausgabe: SP-graph G mit zugehöriger 3KNF Formel $\Phi_G \equiv \psi$

Die Laufzeit vom Algorithmus beträgt $O(|\psi|)$.

Beispiel 2. Sei $\psi = \neg a \wedge ((b \wedge \neg c) \vee (d \wedge e))$. Nun wenden wir den Algorithmus an:





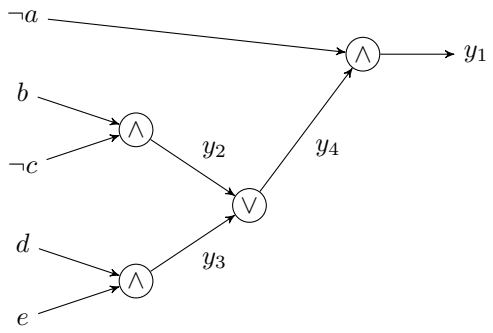
Die zugehörige Formel heißt dann

$$\Phi_G = \exists x \exists y \exists z x \wedge \neg y \wedge (\neg x \vee y \vee \neg a) \wedge (\neg x \vee z \vee b) \wedge (\neg x \vee z \vee \neg c) \wedge (\neg z \vee y \vee d) \wedge (\neg z \vee y \vee e).$$

Nun können wir die Einheitsklauseln für Quelle und Senke, wie immer, propagieren:

$$\Phi_G = \exists z \neg a \wedge (z \vee b) \wedge (z \vee \neg c) \wedge (\neg z \vee d) \wedge (\neg z \vee e).$$

Über die Tseitin-Konstruktion betrachten wir den zu ψ gehörenden Schaltkreis Hieraus ergibt sich der Schaltkreis



und erhalten die zu ψ äquivalente Formel mit zunächst 4 neuen Variablen

$$y_1 \wedge (y_1 \leftrightarrow (\neg a \wedge y_4)) \wedge (y_4 \leftrightarrow (y_2 \vee y_3)) \wedge (y_2 \leftrightarrow (b \wedge \neg c)) \wedge (y_3 \leftrightarrow (d \wedge e))$$

die mit den Übersetzungsregeln einfach in KNF gebracht werden kann:

$$\begin{aligned} y_1 \wedge &(\overline{y}_1 \vee \neg a) \wedge (\overline{y}_1 \vee y_4) \wedge (y_1 \vee \overline{\neg a} \vee \overline{y}_4) \wedge \\ &(\overline{y}_4 \vee y_2 \vee y_3) \wedge (y_4 \vee \overline{y}_2) \wedge (y_4 \vee \overline{y}_3) \wedge \\ &(\overline{y}_2 \vee b) \wedge (\overline{y}_2 \vee \neg c) \wedge (y_2 \vee \overline{b} \vee \overline{\neg c}) \wedge \\ &(\overline{y}_3 \vee d) \wedge (\overline{y}_3 \vee e) \wedge (y_3 \vee \overline{d} \vee \overline{e}) \end{aligned}$$

Nun noch vereinfachen und Einheitsklausel y_1 (bzw. resultierend $\neg a$ und y_4) propagieren:

$$\begin{aligned} &(\overline{y}_2 \vee b) \wedge (\overline{y}_2 \vee \neg c) \wedge (y_2 \vee \overline{b} \vee \overline{c}) \wedge \\ &(\overline{y}_3 \vee d) \wedge (\overline{y}_3 \vee e) \wedge (y_3 \vee \overline{d} \vee \overline{e}), \end{aligned}$$

welche zwei neue Variablen benötigt. ◀

1.4 Autarke Belegungen

Definition 6. Eine Belegung θ heißt *autark für eine Formel* ϕ in KNF, falls für alle Klauseln $C \in \phi$ gilt: Aus $\mathrm{Vars}(\theta) \cap \mathrm{Vars}(C) \neq \emptyset$ folgt $\theta \models C$. ◀

Ist die leere Belegung $\theta = \emptyset$ autark?

Intuitiv gesprochen ist die Belegung autark, wenn sie alle Klauseln erfüllt, für die sie Variablen mit Wahrheitswerten belegt.

Definition 7. Eine Literal ℓ kommt in einer Formel ϕ *pur* vor, wenn $\neg\ell \notin \phi$ gilt. ◀

Beobachtung. Für jedes pure Literal ℓ in einer Formel ϕ ist die Belegung θ mit $\theta(\ell) = 1$ autark. ◀

Beispiel 3. Die Belegung $\theta(x_1) = 1, \theta(x_2) = 0$ ist autark für $\phi = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee x_4)$. ◀

Satz 4. Sei ϕ eine Klausel in KNF und θ eine autarke Belegung für ϕ. Dann sind ϕ und $\phi[\theta]$ erfüllbarkeitsäquivalent. ◀

Beweis Ist α eine erfüllende Belegung für $\phi[\theta]$ (mit $\mathrm{Vars}(\alpha) \cap \mathrm{Vars}(\theta) = \emptyset$), dann erfüllt $\theta \cup \alpha$ die Formel ϕ.

Sei umgekehrt ϕ erfüllt durch Belegung α. Sei α' die Belegung α eingeschränkt auf die Variablenmenge $\mathrm{Vars}(\phi) \setminus \mathrm{Vars}(\theta)$. Dann erfüllt α' alle Klauseln in ϕ, die keine Variablen aus $\mathrm{Vars}(\theta)$ enthalten. Also wird $\phi[\theta]$ durch α' erfüllt. ■

Burkhard Monien, *1943,
(c) Uni Paderborn

Das Spannende an autarken Belegungen ist das Folgende: eine backtracking-Prozedur wendet die Belegung θ auf die Formel ϕ an, um einen rekursiven Aufruf mit $\phi[\theta]$ zu starten. Es ist leicht zu prüfen, ob θ autark ist. Ist dies so, dann kann man einfach mit $\phi[\theta]$ weiterarbeiten.

Tritt jedoch der wahrscheinlichere Fall auf, dass jedoch θ nicht autark ist, kann man hier einen kleinen Vorteil erzielen: angenommen wir haben eine k-KNF vorliegen. Wenn θ nicht autark ist, wird also mindestens eine Klausel C existieren in der ein Literal ℓ auf falsch gesetzt wurde. Also hat C in $\phi[\theta]$ nur noch Länge $\leq k-1$. Im nächsten Aufruf kann man diese Klausel eventuell zur weiteren Verarbeitung heranziehen (Idealfall wäre C ist Einheitsklausel).

Der Algorithmus von Monien und Speckenmeyer

Dieser Algorithmus von 1985 war ein erster Schritt in die Verbesserung der bekannten Exponentialzeit-Algorithmen und bewies eine untere Schranke von $1{,}618^n$. Wir folgen in diesem Abschnitt Schöning (2009).

Der Algorithmus wählt das erste Literal ℓ in der ersten kürzesten Klausel der gegebenen Formel und verzweigt sich darauf genau einmal mit einer Belegung, welche ℓ wahr werden lässt. Wird nun keine erfüllende Belegung gefunden, wissen wir, dass wir ℓ aus der Klausel herauswerfen können und arbeiten an dieser Klausel (die die kürzestes bleibt) weiter. Diesen Ansatz kann man nutzen um mehrere dieser Verzweigungsvorgänge zusammen zu fassen.

Algorithmus MS
Eingabe: Aussagenlogische Formel ϕ in k-KNF
1: **if** ϕ ist die leere Formel **then return** 1;
2: **if** ϕ enthält die leere Klausel **then return** 0;
3: Sei $K = \{\ell_1, \ldots, \ell_r\}$ die erste kürzeste Klausel in ϕ;
4: **for** $i = 1, \ldots, r$ **do**
5: $\quad \theta_i = \{\ell_1 = 0, \ldots, \ell_{i-1} = 0, \ell_i = 1\}$;
6: $\quad$ **if** MS$(\phi[\theta_i])$ **then Return** 1 **end if**
7: **return** 0;

Maßgeblich für den Algorithmus sind die maximal k verschiedenen Verzweigungen. Dies liefert für $n = |\text{Vars}(\phi)|$ eine Rekursionsgleichung $T(n) = T(n-1)+\cdots+T(n-k)$. Wir vermuten nun eine Exponentiallaufzeit, also $T(n) = \beta^n$. Damit erhalten wir nun

$$\begin{aligned} T(n) \leq T(n-1) + \cdots + T(n-k) &\Leftrightarrow \beta^n = \beta^{n-1} + \cdots + \beta^{n-k} \\ &\Leftrightarrow \beta^k = \beta^{k-1} + \cdots + \beta^0. \end{aligned}$$

Aus $\beta^{k-1} + \cdots + \beta^1 + 1 = \frac{1-\beta^k}{1-\beta}$ für $\beta \neq 1$ erhalten wir $\beta^{k+1} + 1 = 2 \cdot \beta^k$. Im folgenden ist abgebildet, welche Werte β für verschiedene k annehmen kann:

$k = 3$	4	5	6	7	8
$\beta = 1{,}839$	1,927	1,966	1,984	1,992	1,996

Im folgenden werden wir die im vorherigen Abschnitt angesprochenen Autarkien benutzen, um den Algorithmus signifikant zu verbessern.

Ewald Speckenmeyer„
(c) Uni Köln

Algorithmus MSa
Eingabe: Aussagenlogische Formel ϕ in k-KNF
1: **if** ϕ ist die leere Formel **then return** 1;
2: **if** ϕ enthält die leere Klausel **then return** 0;
3: Sei $K = \{\ell_1, \ldots, \ell_r\}$ die erste kürzeste Klausel in ϕ;
4: **for** $i = 1, \ldots, r$ **do**
5: $\quad \theta_i = \{\ell_1 = 0, \ldots, \ell_{i-1} = 0, \ell_i = 1\}$;
6: $\quad$ **if** θ_i ist autark für ϕ **then return** MSa$(\phi[\theta_i])$;
7: **for** $i = 1, \ldots, r$ **do**
8: $\quad \theta_i = \{\ell_1 = 0, \ldots, \ell_{i-1} = 0, \ell_i = 1\}$;
9: $\quad$ **if** MSa$(\phi[\theta_i])$ **then return** 1;
10: **return** 0;

Wieso erzielen wir jetzt eine bessere Laufzeit? Wir wissen, dass in der zweiten Schleife keine autarken Belegungen mehr auftreten können. Also folgt, dass $\phi[\theta_i]$ mindestens eine Klausel der Länge $k-1$ enthalten muss. Daher ändert sich die Rekursionsgleichung zu insgesamt zwei Gleichungen (für jede Schleife eine) wie folgt

$$\begin{aligned} T(n) &\leq \max\{T(n-1), \cdots, T(n-k)\} \\ T'(n) &\leq \max\{T'(n-1), \cdots, T'(n-k+1)\} \end{aligned}$$

Der schlimmsten Fall ist rein rechnerisch der zweite Fall, wenn keine Autarkien auftreten. Also genügt es diesen Fall zu betrachten. Für $T'(n) = T'(n-1) + \cdots +$

$T'(n-k+1)$, sowie für $T(n) = O(T'(n))$ und aus der vorherigen Betrachtungsweise der exponentiellen Laufzeit erhalten wir

$$\begin{aligned} \beta^n &= \beta^{n-1} + \cdots + \beta^{n-k+1} \\ \Leftrightarrow \beta^{k-1} &= \beta^{k-2} + \cdots + 1. \end{aligned}$$

Wir vorher benutzen wir die charakteristische Gleichung $\beta^{k-1} + \cdots + \beta^1 + 1 = \frac{1-\beta^k}{1-\beta}$ für $\beta \neq 1$ und bekommen $\beta^k + 1 = 2\beta^{k-1}$. Damit ergibt sich für $k = 3$ ein Wert für $\beta = \frac{\sqrt{5}+1}{2} \approx 1{,}618$ die Zahl des goldenen Schnitts. Die anderen Werte sind

$k = 3$	4	5	6	7	8
$\beta = 1{,}618$	1,839	1,928	1,966	1,984	1,992

Satz 5. Der deterministische Monien-Speckenmeyer-Algorithmus für k-SAT (unter Berücksichtigung der autarken Belegungen) erreicht eine Laufzeit von $O^*(a^n)$, wobei $a > 1$ die Lösung der Gleichung $a^k + 1 = 2a^{k-1}$ ist. ◀

1.5 Kombinatorische Beobachtungen

Es gibt bestimmte Eigenschaften von Formeln, die sofort zur Beantwortung der Erfüllbarkeitsfrage genutzt werden können. Solche wollen wir in diesem Abschnitt genauer betrachten.

Lemma 3. Ist ϕ eine Konjunktion von Klauseln und es gibt keine Klausel mit nur positiven (bzw. nur negativen) Literalen, dann ist ϕ erfüllbar. ◀

Beweis Entweder nehmen wir die Belegung $\theta(x) = 0$ oder $\theta(x) = 1$ für alle Variablen $x \in \mathrm{Vars}(\phi)$. ■

Lemma 4. Ist ϕ eine Konjunktion von Klauseln, die alle genau k Literale enthalten und besteht ϕ aus weniger als 2^k Klauseln, dann ist ϕ erfüllbar. ◀

Beweis Sei $\mathrm{Vars}(\phi) = n \geq k$. Da jede Klausel C genau k Literale enthält, gibt es exakt 2^{n-k} Belegungen θ mit $\mathrm{Vars}(\theta) = n$, die C falsch werden lassen. Insgesamt gibt es somit weniger als $2^k \cdot 2^{n-k} = 2^n$ Belegungen, die ϕ falsifizieren. ϕ muss also erfüllbar sein.

Das heißt, jede Klausel C schließt einen Anteil von 2^{-k} bzw. $2^{-|C|}$ der potenziell erfüllbaren Belegungen aus. ■

Satz 6. Sei $\phi = \bigwedge_{i=1}^{m} C_i$ eine Konjunktion von Klauseln. Ist

$$\sum_{j=1}^{m} 2^{-|C_j|} < 1,$$

so ist ϕ erfüllbar. ◀

Wir sehen, dass eine nicht zu große Anzahl von Klauseln immer die Erfüllbarkeit der Formel sichert. Darüber hinaus gibt es noch eine andere interessante Eigenschaft mit einem ähnlichen Nutzen. Wenn es wenige Klauseln gibt, die verschränkt sind, also Variablen gemein haben, dann können wir die Formel auch immer erfüllen. Dies wird im folgenden Satz gezeigt.

Lovász Local Lemma

Satz 7. Sei ϕ eine Formel in k-KNF. Gibt es für jede Klausel C in ϕ weniger als 2^{k-2} Klauseln C' mit $\text{Vars}(C) \cap \text{Vars}(C') \neq \emptyset$, dann ist ϕ erfüllbar. ◀

Beweis Wir konstruieren einen randomisierten Algorithmus, welcher mit hoher Wahrscheinlichkeit unter der Voraussetzung vom Satz terminiert und eine erfüllende Belegung findet. Sei $\phi = \bigwedge_{i=1}^{m} C_i$ und $\text{Vars}(\phi) = \{x_1, \ldots, x_n\}$. Der Algorithmus ist nun wie folgt aufgebaut und verwendet einen Unteraufruf an eine Methode Repair:

Eingabe: Aussagenlogische Formel ϕ in k-KNF mit m Klauseln
1: Wähle zufällige Belegung θ;
2: **for** $i = 1, \ldots, m$ **do**
3: **if** $\theta \not\models C_i$ **then Repair**(C_i, θ, ϕ) **end if**
4: **return** θ;

In der Methode müssen wir die Belegung dahingehend modifizieren, dass sie C_i erfüllt und außerdem keine Klausel, die von C_i vorher erfüllt wurde nun falsifiziert wird. Daraus ergibt sich, dass wir eine erfüllende Belegung am Ende konstruiert haben. Nun formulieren wir diese Methode:

Methode Repair
Eingabe: k-Klausel C, Belegung θ, Formel ϕ mit m Klauseln
1: Belege die k Variablen in C mit zufälligen Werten;
2: **for** $j = 1, \ldots, m$ **do**
3: **if** $\text{Vars}(C_j) \cap \text{Vars}(C) \neq \emptyset$ **then**
4: **if** $\theta \not\models \phi$ **then Repair**(C_j, θ, ϕ) **end if**

Unklar ist die Frage nach der Termination der Methode Repair. Betrachten wir die Zufallswerte im Berechnungsprozess:

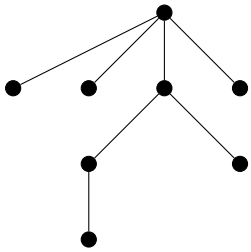
- Anfangsbelegung benötigt n Zufallswerte und
- nach s Aufrufen von Repair benötigen wir $s \cdot k$ Zufallswerte.

Im folgenden betrachten wir eine kompaktere Repräsentation der $n + s \cdot k$ vielen Zufallswerte, sobald s eine bestimmte Schwelle übersteigt.

Zur Rekonstruktion dieser Werte benötigen wir die folgenden Informationen:

- n Bits von θ nach s Repair-Aufrufen und
- m Bits, die die Repair-Aufrufe kodieren, also das j-te Bit ist 1, wenn ein Aufrufe für die j-te Klausel vorgenommen wurde im Hauptprogramm.

Ein *rekursiver* Aufruf an Repair findet nur dann statt, wenn es eine Variablen-verschränkte Klausel C_j gibt. Es gibt maximal $2^{k-2-\varepsilon}$ solcher Klauseln (für ein $\varepsilon > 0$). Daher können wir die Folge der durch einen Repair-Aufruf betroffenen Klauseln durch $(k - 2 - \varepsilon) \cdot s$ viele Bits beschreiben (relative Indizierung). Nun müssen wir noch die rekursive Baumstruktur der Aufrufe kennen. Diese kann durch $2s$ viele Bits beschrieben werden, wobei eine 1 dann einen tiefer liegenden Aufruf und eine 0 einen Rücksprung bedeutet. Beispiel:



Der linke Baum wird beschrieben durch den String 10 10 11 10 01 00 10.

Insgesamt benötigt diese Beschreibung also $n + m + s \cdot (k - \varepsilon)$ viele Bits.

Mit dieser Informationen können wir also alle Repair-Aufrufe reversibel machen. Eine Klausel C_j kann nur auf genau eine Art auf 0 gesetzt werden. Die k betroffenen Bits dieser Klausel können wieder so gesetzt werden, dass C_j zu falsch evaluiert. Dies ist dann der Status vor dem Repair-Aufruf. Nun können wir alle $n + s \cdot k$ Bits rekonstruieren. Mit weniger ist dies nicht möglich (*Incompressibility*-Argument, Kolmogorov-Komplexität). Daraus resultiert die Ungleichung

$$n + s \cdot k \leq n + m + s \cdot (k - \varepsilon) \Leftrightarrow s \leq \frac{m}{\varepsilon}$$

für *fast alle* Zufallbitsfolgen. Also wird der Prozess mit hoher Wahrscheinlichkeit enden und eine erfüllende Belegung ausgeben sobald s größer wird als $\frac{m}{\varepsilon}$. ■

Nach Moser und Tardos (2009) kann man den Wert 2^{k-2} etwas vergrößern zu $\frac{2^k}{e}$.

Lemma 5. Sei ϕ eine Formel in KNF. Gilt für jede Teilmenge K der Klauseln von ϕ, dass $|K| \leq |\text{Vars}(K)|$, dann ist ϕ erfüllbar. ◀

Beweis Sei $\phi = \bigwedge_{i=1}^{m} C_i$ eine KNF-Formel und $\text{Vars}(\phi) = \{x_1, \ldots, x_n\}$. Konstruiere einen bipartiten Graphen wie folgt: $G = (U, V, E)$, wobei

$$U := \{v_{C_i} \mid 1 \leq i \leq m\},$$
$$V := \{v_{x_i} \mid 1 \leq i \leq n\} \text{ und}$$
$$E := \{(u, v) \mid u \in U, v \in V, v \text{ kommt als positives oder negatives Literal in } u \text{ vor.}\}$$

Aus dem Heiratssatz von Hall wissen wir, dass ein Matching mit $|\phi|$ Kanten in G existiert, wenn für alle $K \subseteq \phi$ gilt, dass $|K| \leq |\text{Vars}(K)|$. Hierdurch weisen wir jeder Klausel eine andere Variable zu. Diese kan dann so belegt werden, dass K erfüllt wird, woraus die Erfüllbarkeit von F folgt. ■

Wir sagen eine KNF-Formel ϕ ist *minimal unerfüllbar*, wenn ϕ unerfüllbar ist, aber für jede Klausel $C \in \phi$ die Formel $\phi \setminus \{C\}$ ist erfüllbar.

Tarsi's Lemma, Aharoni u. Linial (1986)

Satz 8. Ist ϕ minimal unerfüllbar, dann gilt $|\phi| > |\text{Vars}(\phi)|$. ◀

Beweis Aus Lemma 5 wissen wir, dass es mindestens eine Teilmenge $K \subseteq \phi$ geben muss, sodass $|K| > |\text{Vars}(K)|$ gilt. Wähle eine solche maximale Teilmenge K. Ist $K = \phi$ folgt der Satz. Angenommen es gilt $K \subsetneq \phi$ an. Sei $K' \subseteq \phi \setminus K$ beliebig. Nun gilt $|K'| < |\text{Vars}(K') \setminus \text{Vars}(K)|$ (andernfalls folgt $|K \cup K'| > |\text{Vars}(K \cup K')|$ was der Maximalität von K widerspricht).

Also erfüllt die Klauselmenge $\phi \setminus K$ die Voraussetzung von Lemma 5 und ist erfüllbar (und zwar durch Belegung θ mit $\text{Vars}(\theta) = \text{Vars}(\phi) \setminus \text{Vars}(K)$). Da ϕ minimal unerfüllbar ist, muss K erfüllbar sein (durch Belegung θ' mit $\text{Vars}(\theta') = \text{Vars}(K)$). Also ist $\theta \cup \theta'$ eine ϕ erfüllende Formel. Widerspruch. Also gilt $K = \phi$. ■

Satz 9. Sei $k \geq 1$ und ϕ eine KNF Formel mit $\geq k$ Literalen pro Klausel, sodass jede Variable in $\leq k$ Klauseln (positiv oder negativ) vorkommt. Dann ist ϕ erfüllbar. ◀ Tovey (1984)

Beweis Sei K eine beliebige Klauselteilmenge von ϕ. Nun gibt es $\geq k \cdot |K|$ viele Literale (Voraussetzung vom Satz). Jede Variable kann an höchstens k dieser Literalen beteiligt sein. Daher gibt es in K mindestens $|K|$ viele verschiedene Variablen, also $|K| \leq |\text{Vars}(K)|$. Also folgt Lemma 5 und ϕ ist erfüllbar. ■

Korollar 2. Sei ϕ eine KNF-Formel. Kommt jede Variable ≤ 2 mal als Literal vor, ist die Erfüllbarkeit von ϕ in P entscheidbar. Kommt eine Variable ≥ 3 mal als Literal vor, so ist das Erfüllbarkeitsproblem NP-vollständig. ◀

Beweis Ist $k = 1$, so ist die Formel trivial erfüllbar. Für den Fall $k = 2$ skizzieren wir einen P-Algorithmus: Eliminiere mit Unit Propagation die Einheitsklauseln. Entsteht eine leere Klausel, ist ϕ unerfüllbar. Andernfalls ist ϕ erfüllbar, da Satz 9 für $k = 2$ folgt.

Anschließend zeigen wir, dass für $k = 3$ das Problem bereits NP-vollständig ist, was den Satz beweist. Hierzu konstruieren wir zu jeder Formel eine erfüllbarkeitsäquivalente Formel in der jede Variable höchstens dreimal auftritt. Sei x eine Variable, die $s > 3$ mal auftritt. Nun ersetze jedes Vorkommen von x durch eine neue Variable $x_1, \ldots, x_s$. Außerdem füge die folgenden neuen Klauseln hinzu:

$$(\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), \ldots, (\overline{x_{s-1}} \vee x_s), (\overline{x_s} \vee x_1).$$

Diese Klauseln sind genau dann erfüllbar, wenn alle x_i für $1 \leq i \leq s$ den selben Wahrheitswert zugewiesen bekommen. Daraus ergibt sich die Erfüllbarkeitsäquivalenz zur vorherigen Formel. ■

Kapitel 2

Spezialfälle

Im folgenden betrachten wir Einschränkungen von aussagenlogischen Formeln, die in der Regel auf bestimmten Normalformen beruhen und für die es effiziente Polynomialzeit-Algorithmen gibt.

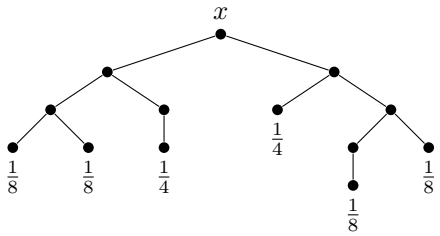
2.1 Krom-SAT

Bevor wir zu diesen P-Resultaten kommen, möchten wir die probabilistische Klasse RP betrachten.

Probabilistische Turingmaschine (PTM)

Definition 8. Sei $M = (Z, \Sigma, \delta, z_0)$ eine NTM mit einem weiteren ausgezeichneten Zustand $z_{\text{neut}} \in Z$. Für ein $z \in Z$ und ein $a \in \Sigma$ gelte $|\delta(z,a)| \leq 2$. Ist $|\delta(z,a)| = 2$, so wird jede der beiden möglichen Nachfolgekonfigurationen mit Wahrscheinlichkeit $1/2$ gewählt.

Also werden Pfade in $T_M(x)$ mit einer bestimmten Wahrscheinlichkeit gewählt:



Sei π ein Pfad in $T_M(x)$ für eine PTM M. Wir definieren

$$\varphi_M(x|\pi) := \begin{cases} 1 & \text{, falls } M \text{ auf } p \text{ in } z_{\text{akz}} \text{ hält,} \\ 0 & \text{, falls } M \text{ auf } p \text{ in } z_{\text{verw}} \text{ hält,} \\ ? & \text{, falls } M \text{ auf p in } z_{\text{neut}} \text{ hält.} \end{cases}$$

$\wp[M(x) = a]$ ist die Wahrscheinlichkeit, dass bei einem zufällig gewählten Pfad π $\varphi_M(x|\pi) = a$ gilt.

Die Klasse RP ist besteht aus allen Sprachen A, für die es eine polynomiell zeitbeschränkte PTM M gibt, sodass für alle x gilt:

$$x \in A \Rightarrow \wp[M(x) = 1] > \frac{1}{2}, \text{ und}$$
$$x \notin A \Rightarrow \wp[M(x) = 1] = 0.$$

Melven R. Krom, *1932,
Copyright ucdavis

◀

Satz 10. Das Problem

$$\text{Krom-SAT} := \{\langle\phi\rangle \mid \phi \text{ ist eine erfüllbare aussagenlogische Krom-Formel }\}$$

ist in RP. ◀

Beweis Hierbei verfolgen wir den folgenden Ansatz:

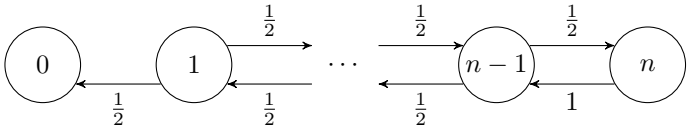
Eingabe: Krom-Formel ϕ
1: Wähle eine beliebige zufällige Anfangsbelegung θ über $\text{Vars}(\phi)$.
2: **if** $\theta \models \phi$ **then**
3: **akzeptiere**
4: **else**
5: Sei $K = (u, v)$ eine Klausel in ϕ mit $\theta \not\models K$.
6: Mit W'keit jeweils $\frac{1}{2}$ ändere θ, sodass $\theta \models u$ oder $\theta \models v$.

Frage: Wie lange sollten wir diesen Algorithmus wiederholen?

Richard Hamming, 1915-1998,
Copyright Louis Fabian Bachrach

Angenommen ϕ ist erfüllbar durch θ_0. Die Belegung θ aus Zeile 1 unterscheidet sich in einigen Bits von θ_0. Sei $d(\theta, \theta_0)$ die Hamming-Abstand dieser Belegungen.

Ändern wir die Belegung in Schritt 6, so erhalten wir mit W'keit von $\geq \frac{1}{2}$ eine Abnahme des Wertes $d(\theta, \theta_0)$ um eins, da $\theta_0 \models K$. Eine Erhöhung des Wertes um 1 tritt mit W'keit von $\leq \frac{1}{2}$ ein. Der Mehrfachausführung des Algorithmus ist also ein sogenannter *Random Walk* auf einer Markov-Kette mit den Zuständen $0, 1, \ldots, n$, wobei der Zustand immer den Hamming-Abstand angibt. Zustand n ist reflektierend und Zustand 0 ist absorbierend:



Sei $S(j)$ mit $0 \leq j \leq n$ die erwartete Schrittzahl bis Zustand 0 besucht wird. Dies entspricht der gefundenen Belegung θ_0, wenn wir bei einer Belegung θ mit $d(\theta, \theta_0) = j$ beginnen. Nun gilt:

$$\begin{aligned} S(0) &= 0, \\ S(j) &= 1 + \frac{1}{2} \cdot S(j-1) + \frac{1}{2} \cdot S(j+1) \text{ für } j = 1, 2, \ldots, n-1 \text{ und} \\ S(n) &= 1 + S(n-1). \end{aligned}$$

Eine Induktion zeigt $S(j) = 2 \cdot j \cdot n - j^2$. Also ist die erwartete Schrittzahl beschränkt durch n^2. Aus der Markov-Ungleichung

$$P(X \geq 2 \cdot n^2) \leq \frac{1}{2}$$

ergibt sich, dass die Wahrscheinlichkeit, dass wir innerhalb von $2n^2$ Schritten eine erfüllende Belegung *nicht* bekommen höchstens $\frac{1}{2}$ ist, also $\wp[M(x) = 1] > \frac{1}{2}$ gilt. Also ist Krom-SAT $\in$ RP. ■

Satz 11. Krom-SAT $\in$ P. ◀

Beweis Hierzu sei ϕ eine aussagenlogische Krom-Formel über den Variablen $x_1, \ldots, x_n$. Die Formel hat also die Form $\phi = \bigwedge_{i=1}^{m}(l_{i1} \vee l_{i2})$, wobei $m \in \mathbb{N}$ und l_{ij} für $1 \leq i \leq m$ und $j = 1, 2$ Literale sind, d.h., $l_{ij} = x_k$ oder $l_{ij} = \neg x_k$ für ein x_k und $1 \leq k \leq n$.

Die Idee bei dem Algorithmus für Krom-SAT ist, dass man jede Klausel in ϕ in zwei verschiedene Implikationen umformen kann, da die Klausel $u \vee v$ äquivalent zu $\neg u \to v$ bzw. $\neg v \to u$ sind. Diese Implikationen drücken wir durch Kanten in einem Graphen aus. Die Kantenrelation entspricht dann der logischen Implikation von den Belegungen der Wahrheitswerte der Literale (und damit der Variablen) in ϕ. Folglich wählen wir als Knotenmenge in dem Graphen dann alle Variablen und ihre Negation (also alle Literale).

Gibt es nun eine Variable x in ϕ, sodass ein Pfad von x zu $\overline{x}$ und ein Pfad von $\overline{x}$ zu x in dem Graphen existiert, dann ist die Formel ϕ nicht erfüllbar.

Nun definieren wir den gerichteten Graphen $G = (V, E)$ formal wie folgt:

$$V = \{x_i, \neg x_i \mid 1 \leq i \leq n\},$$
$$E = \{(\sim l_{i1}, l_{i2}), (\sim l_{i2}, l_{i1}) \mid 1 \leq i \leq m\},$$

wobei $\sim \ell_{ij} = x$ falls $\ell_{ij} = \neg x$ und $\sim \ell_{ij} = \neg x$ falls $\ell_{ij} = x$ sonst. Der Algorithmus für Krom-SAT lautet nun

Eingabe: Krom-Formel ϕ
1: Konstruiere den gerichteten Graphen $G = (V, E)$ wie oben beschrieben.
2: **for** jede Variable x in ϕ **do**
3: **if** es gibt einen Zyklus auf dem x und $\neg x$ liegen **then**
4: **verwerfe**
5: **akzeptiere**

Die Laufzeit von dem Algorithmus ist insgesamt polynomiell, da die **for**-Schleife n-mal durchlaufen wird und in der **if**-Abfrage ein Zyklus gefunden werden muss. Die Zyklussuche ist im Endeffekt nichts anderes als eine Breitensuche von x nach $\neg x$ und umgekehrt.

Nun zur Korrektheit des Algorithmus: gilt $\phi \notin$ Krom-SAT, dann verwirft der Algorithmus. Die Korrektheit hiervon folgt sofort aus den obigen Überlegungen und der sich hieraus ergebenden logischen Äquivalenz von einer Variable x und ihrer Negation.

Nun bleibt zu zeigen, wie wir für den akzeptieren-Fall eine valide Belegung θ konstruieren können, sodass $\theta \models \phi$. Hierzu wählen wir ein beliebiges Literal ℓ in dem Graphen, dessen zugehörige Variable noch nicht auf wahr oder falsch gesetzt worden ist und für die es keinen Pfad von ℓ nach $\sim \ell$ gibt. Setze nun $\theta(\ell) := 1$ und für alle Literale ℓ_{ij}, die über Kanten von ℓ aus erreicht werden können auch $\theta(\ell_{ij}) := 1$. Hieraus ergibt sich, dass für alle Knoten $(\sim \ell_{ij}) = 0$ gilt. Dieser Schritt ist auch wohldefiniert, da wenn es Pfade von ℓ zu sowohl ℓ' als auch $\sim \ell'$ gäbe (für ein anderen Knoten ℓ'), es auch Pfade zu $\sim \ell$ von beiden gäbe und wir damit fälschlicherweise einen Pfad von ℓ zu ℓ' hätten. Dies folgt aus der Symmetrie der Implikation. Wir belegen nun Variablen mit Wahrheitswerten nach dem beschriebenen Schema und erreichen irgendwann den Punkt, dass alle Variablen mit Werten belegt sind in θ. Da es keine widersprüchlichen Variablen gibt, ist dies eine gültige Belegung für ϕ. ■

2.2 Horn-Formeln

Eine *Horn*-Formel ist eine aussagenlogische Formel in KNF, sodass jede Klausel höchstens ein positives Literal enthält.

Satz 12. Das Problem

$$\text{HORN-SAT} := \{\langle\phi\rangle \mid \phi \text{ ist eine erfüllbare Horn-Formel}\}$$

ist in P. ◀

Beweis Aus Lemma 3 wissen wir, dass eine Horn-Formel, die keine positive Einheitsklausel enthält oder die keine negative Klausel enthält erfüllbar ist. Nun ist jede Horn-Formel, die keine positive Klausel enthält, erfülltbar durch Setzen aller Variablen auf 0.

Hat die Formel eine positive Einheitsklausel, dann müssen wir diese auf 1 setzen und wenden Unit Propagation an. Hieraus erhalten wir wieder eine Horn-Formel und wir fahren fort.

Eingabe: Horn-Formel ϕ
1: Belegung $\theta := \emptyset$;
2: **while** $\{x\} \in \phi$ für eine Variable x **do**
3: $\quad\theta := \theta \cup \{x = 1\}$;
4: $\quad\phi := \phi[\theta]$;
5: **if** $\square \in \phi$ **then verwerfe end if**
6: **akzeptiere**

Der Algorithmus läuft in Zeit $O(|\phi|^2)$. θ ist außerdem ein sogenanntes *Minimalmodell* für ϕ: nur die Variablen, die wirklich auf wahr gesetzt werden müssen, werden dies. Alle anderen werden mit 0 belegt. ■

Renamable Horn

Definition 9. Eine Formel ϕ in KNF heißt *renamable Horn* (*hidden Horn*), wenn es eine Teilmenge $U \subseteq \text{Vars}(\phi)$ gibt, sodass $\phi[x = \neg x \mid x \in U]$ eine Horn-Formel ist. U nennen wir dann *Umbenennung*. ◀

Satz 13. Die Formel ϕ ist renamable Horn genau dann, wenn die Krom-Formel

$$\phi^\star := \{(u, v) \mid u, v \in K \text{ für eine Klausel } K \in F\}$$

erfüllbar ist. Existiert eine erfüllende Belegung θ von $\phi^\star$, dann entspricht diese der Umbenennung U, sodass $x \in U \Leftrightarrow \theta \models x$. ◀

Beweis Sei ϕ eine renamable Horn-Formel und $U \subseteq \text{Vars}(\phi)$ die entsprechende Umbenennung. Betrachte die Belegung

$$\theta = \{x = 1 \mid x \in U\} \cup \{x = 0 \mid x \notin U\}.$$

Sei (u, v) eine beliebige Klausel in $\phi^\star$. Dann sind die Literale u, v aus einer Klausel K in ϕ. Nach der Umbenennung ist in $(u \vee v)[x = \neg x \mid x \in U]$ mindestens ein Literal negativ; es wird mindestens ein Literal in $(u \vee v)$ durch θ auf 1 gesetzt. Daher ist $\phi^\star$ erfüllbar.

Sei θ eine $\phi^\star$ erfüllende Belegung. Also werden in ϕ keine zwei Literale in einer Klausel durch θ auf 0 gesetzt. Setze $U = \{x \in \text{Vars}(\phi) \mid \theta \models x\}$. Hiermit erhalten wir eine Umbenennung, die keine zwei positiven Literale in einer Klausel generiert. Es entsteht also die gewünschte Horn-Formel. ■

Da Krom-SAT $\in P$ gilt, folgt folgendes Korollar.

Korollar 3. Umbenennungen können effizient gefunden werden. ◀

Kapitel 3

Ein Meta-DPLL-Algorithmus

Dieses Kapitel entspricht (Schöning u. Torán, 2012, Kapitel 4).

Im folgenden betrachten wir einige Varianten der wichtigsten Grundlage vieler SAT-Algorithmen. Hierbei geht es um den DPLL-Algorithmus (benannt nach Davis, Putnam, Logemann und Loveland). Zunächst fangen wir mit einer *Meta*-Version des Algorithmus an.

Algorithmus Meta-DPLL
Eingabe: Klauselmenge ϕ
Ausgabe: Akzeptiert gdw. ϕ erfüllbar ist.
1: **if** $\square \in \phi$ **then verwerfe end if**
2: **if** $\phi = \emptyset$ **then akzeptiere end if**
3: **if** ϕ enthält Einheitsklausel $\{u\}$ **then return Meta-DPLL**$(\phi[u = 1])$ **end if**
4: **if** ϕ enthält pures Literal u **then return Meta-DPLL**$(\phi[u = 1])$ **end if**
5: Wähle eine Variable $x \in \text{Vars}(\phi)$;
6: **if Meta-DPLL**$(\phi[x = 0])$ **then akzeptiere end if**
7: **return Meta-DPLL**$(\phi[x = 1])$;

In Zeile 5 wird nicht weiter spezifiziert wie genau man die Variable x auswählt. Hierzu betrachten wir einige Möglichkeiten und Heuristiken. Die Unit Propagation in Zeile 3 ist leicht zu implementieren und kommt in vielen SAT-Solvern vor. Pure Literale zu bestimmen, wie in Zeile 4 getan ist zeitlich aufwändiger und wird daher häufig weggelassen.

Definition 10. Ist ϕ eine KNF-Formel, dann ist

Häufigkeits-Funktion

$$f_k(u) = |\{C \mid \text{Klausel } C \in \phi, |C| = k, u \in C\}|,$$
$$f(u) = \sum_{k=2}^{n} f_k(u) \quad \text{(Anzahl der Vorkommen von } u \text{ in } \phi)$$

◀

Die wichtigsten heuristischen Auswahlfunktionen sind

Dynamic Largest Individual Sum (DLIS): Wähle ein Literal u mit $f(u)$ maximal und setze erst $u = 1$.

Dynamic Largest Clause Sum (DLCS): Wähle eine Variable x mit $f(x)+f(\neg x)$ maximal und setze erst $x = 1$, falls $f(x) \geq f(\neg x)$; sonst setze $x = 0$.

Maximum Occurence in Minimal Size Clauses (MOM): Sei k die Länge einer kürzesten Klausel. Wähle eine Variable x mit $(f_k(x) + f_k(\neg x)) \cdot p + f_k(x) \cdot f_k(\neg x)$ maximal, wobei p ein hinreichend großer Parameter ist, sodass der erste Term den zweiten dominiert. Im Prinzip wählt der Algorithmus dann die Variable, die am häufigsten und in den kürzesten Klauseln vorkommt. Gibt es mehrere solche, dann wird diejenige ausgewählt, die gleich verteilt für die pos./neg. Vorkommen sind.

Böhm-Heuristik: Sei $H(x) = (H_2(x), \ldots, H_n(x))$, wobei

$$H_i(x) = p_1 \cdot \max\{f_i(x), f_i(\neg x) + p_2 \cdot \min\{f_i(x), f_i(\neg x)\}\}.$$

p_1, p_2 sind frei wählbare Parameter. Wähle nun die Variable x, sodass $H(x)$ (lexikographisch) maximal ist.

Jeroslaw-Wang: Definieren wir für ein Literal u

$$jw(u) = \sum_{i=2}^{n} f_i(u) \cdot 2^{-i} = \sum_{u \in K} 2^{-|K|}.$$

Wähle nun ein Literal u mit maximalem $jw(u)$ und setze $u = 1$.

Hier gibt es auch eine *zweiseitige Variante*: Wähle x mit maximalem $jw(x) + jw(\neg x)$ und setze $x = 1$, falls $jw(x) \geq jw(\neg x)$; sonst setze $x = 0$.

Kürzeste Klausel: Wähle ein beliebiges Literal aus einer kürzesten Klausel. Dieses Prinzip wurde bereits schon im Algorithmus MS bzw. MSa von Seite 9ff angewendet.

3.1 Probabilistische Varianten

Im Gegensatz zum gewöhnlichen Verzweigen durch Backtracking müssen probabilistische Verfahren entsprechend oft wiederholt werden. Sei S_n die Menge der Permutationen über n Elementen. Solche Algorithmen wählen in der Regel eine zufällige Startpermutation $\pi \in S_n$ aus und nimmt dies als vorgegebene Reihenfolge über den Variablen. Kommt die aktuell betrachtete Variable in einer Klausel alleine vor, wenden wir wie üblich Unit Propagation an. Können wir kein Unit Propagation durchführen belegen wir die Variable *pseudo-zufällig* mit 0 oder 1 (anstelle des Backtrackings).

Satz 14. Es gibt einen Algorithmus für Formeln in k-KNF mit n Variablen, der mit Wahrscheinlichkeit $\geq 2^{-n(1-\frac{1}{k})}$ eine erfüllende Belegung findet. Die Komplexität dieses Verfahrens ist daher $O^*(2^{n(1-\frac{1}{k})})$. ◀

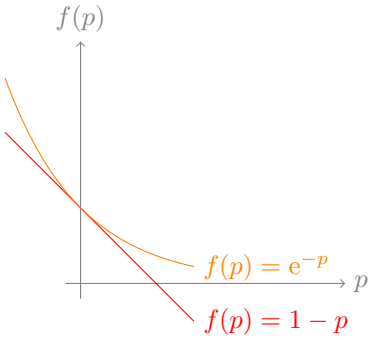
Beweis Betrachten wir den folgenden Algorithmus

Algorithmus PPZ (Paturi, Pudlák, Zane, 1997)
Eingabe: KNF-Formel ϕ mit $n = \text{Vars}(\phi)$

```
1: θ := ∅;
2: Wähle zufällige Permutation π ∈ S_n;
3: for i := 1, ..., n do
4:     if {x_π(i)} ∈ φ[θ] then
5:         θ := θ ∪ {x_π(i) = 1};
```

```
6:    else if {¬x_π(i)} ∈ φ[θ] then
7:        θ := θ ∪ {x_π(i) = 0};
8:    else
9:        Wähle zufällig a ∈ {0,1};
10:       θ := θ ∪ {x_π(i) = a};
11: return θ;
```

Sei p die W'keit, dass PPZ eine erfüllende Belegung findet. Wenn der Algorithmus t-mal mit unabhängigen Zufallszahlen wiederholt wird, ist die W'keit, dass in keiner dieser Versuche eine erfüllende Belegung gefunden wird $(1-p)^t \leq e^{-pt}$. Um eine konstante Fehlerw'keit von e^{-c} zu erzielen, reicht es, den Algorithmus $\frac{c}{p}$ oft zu wiederholen (also für t den Wert $\frac{c}{p}$ einzusetzen).



Betrachten wir die Basiszahl $a = 2^{1-\frac{1}{k}}$ für einige k-Werte, sowie die Zahl $b = 1-\frac{1}{k}$. b gibt Aufschluss darüber, welcher Anteil der Variablen im Durchschnitt erraten werden muss (bei deterministischen Algorithmen wäre dies der Backtracking-Anteil).

k	3	4	5	6	7	8
a	1.587	1.682	1.741	1.782	1.811	1.834
b	0.667	0.75	0.8	0.833	0.857	0.857

Man sieht, für steigendes k konvergieren die Basiszahlen nicht ganz so schnell gegen 2, wie bei anderen k-SAT-Algorithmen. Nun zum Beweis des Satzes.

Sei ϕ erfüllbar via θ mit $\theta(x_i) = a_i$ für $1 \leq i \leq n$ und $a_i \in \{0,1\}, \mathrm{Vars}(\phi) = \{x_1, \ldots, x_n$. Wir nennen das Bit a_i *kritisch*, falls das Flippen von a_i eine nicht-erfüllende Belegung produziert, also $\theta[x_i = \overline{a_i}] \not\models \phi$. Ist Bit i in θ kritisch, dann gibt es eine Klausel $K \in \phi$, so dass genau ein Literal ℓ in K den Wert 1 unter θ erhält. Sei x_i die Variable zu ℓ. Dann nennen wir auch ℓ und x_i kritisch. Für eine erfüllende Belegung θ kann eine kritische Klausel nur eine kritische Variable enthalten, weil sie nur ein Literal mit Wert 1 besitzt. Sei $j(\theta)$ die Anzahl der kritischen Bits von θ (also $0 \leq j(\theta) \leq n$) und sei $s(\theta) = n - j(\theta)$. Also ist $s(\theta)$ die Anzahl der Bits deren Flippen von θ aus wieder auf eine erfüllende Belegung von ϕ führt.

Behauptung. Sei ϕ eine erfüllbare Formel und sei $\emptyset \subsetneq S \subseteq \{0,1\}^n$ die Menge der erfüllenden Belegungen von ϕ. Dann gilt $\sum_{\theta \in S} 2^{-s(\theta)} \geq 1$.

Beweis (Behauptung) Induktion über n. Induktionsanfang $n = 0$ ist trivial.

Induktionsschritt: Sei $n > 0$ und definiere Mengen

$$S_0 = \{s \in \{0,1\}^{n-1} \mid s0 \in S\}, \quad S_1 = \{s \in \{0,1\}^{n-1} \mid s1 \in S\}.$$

Jede dieser Menge habe eine Funktion s_0, s_1, die sich wie s in Bezug auf S verhalten. Nach IV gilt $\sum_{\theta \in S_i} 2^{-s_i(\theta)} \geq 1$. Da $1 \leq |S| = |S_0| + |S_1|$ gilt, können nicht beide Mengen leer sein. Sind beide Mengen nichtleer, dann folgt nach IV

$$\begin{aligned}\sum_{\theta \in S} 2^{-s(\theta)} &= \sum_{\chi \in S_0} 2^{-s(\chi 0)} + \sum_{\chi \in S_1} 2^{-s(\chi 1)} \\ &\geq \sum_{\chi \in S_0} 2^{-s_0(\chi)-1} + \sum_{\chi \in S_1} 2^{-s_1(\chi)-1} \\ &\overset{\text{IV}}{\geq} \frac{1}{2} + \frac{1}{2} = 1.\end{aligned}$$

Sei o.B.d.A. S_0 leer. Dann folgt

$$\sum_{\theta \in S} 2^{-s(\theta)} = \sum_{\chi \in S_1} 2^{-s(\chi 1)} = \sum_{\chi \in S_1} 2^{-s_1(\chi)} \geq 1.$$

■

Nun schätzen wir die W'keit ab, dass PPZ ein bestimmtes θ findet. Folgen wir dem Schema aus dem Algorithmus und belegen die Variablen mit Werten und ist eine kritische Variable x_i die letzte, die in ihrer kritischen Klausel aufgezählt wird, dann wird vom Algorithmus der richtige Wert für x_i auf deterministische Weise mittels Unit Propagation belegt. Das erhöht die W'keit, dass die Belegung θ gefunden wird. Für eine Permutation σ bezeichnen wir mit $r(\theta, \sigma)$ die Anzahl der kritischen Variablen unter θ, die in ihrer kritischen Klausel die letzte sind, in derjenigen Anordnung, die durch $\sigma \in S_n$ definiert wird. Es gilt $0 \leq r(\theta, \sigma) \leq j(\theta)$. Wenn die Programmvariable π den Wert σ hat, so ist $r(\theta, \sigma)$ die Anzahl der eingesparten Zufallsbits in der Suche nach θ. Also gilt

$$P(\text{Algorithmus findet } \theta \mid \pi = \sigma) = 2^{-n+r(\theta,\sigma)}.$$

Da wir die Permutation unter Gleichverteilung wählen (d. h. $\pi \sim S_n$), ist jede der k Variablen in einer Klausel mit W'keit $\frac{1}{k}$ die letzte in π. Das gilt für jede der $j(\theta)$ kritischen Variablen, un der Erwartungswert für $r(\alpha, \pi)$ ist daher $\geq \frac{j(\theta)}{k}$:

$$\mathbb{E}[r(\theta, \pi)] = \frac{1}{n!} \cdot \sum_{\sigma \in S_n} r(\theta, \sigma) \geq \frac{j(\theta)}{k}. \tag{3.1}$$

Nun schätzen wir die W'keit ab ein erfüllendes θ zu finden:

$$\begin{aligned}
P(\text{Algorithmus findet } \theta) &= \sum_{\sigma \in S_n} P(\text{Algorithmus findet } \theta \mid \pi = \sigma) \cdot P(\pi = \sigma) \\
&= \sum_{\sigma \in S_n} 2^{-n+r(\theta,\sigma)} \cdot \frac{1}{n!} \\
&= 2^{-n} \cdot \left(\frac{1}{n!} \cdot \sum_{\sigma \in S_n} 2^{r(\theta,\sigma)} \right) = 2^{-n} \mathbb{E}[2^{r(\theta,\pi)}] \\
&\geq 2^{-n} 2^{\mathbb{E}[r(\theta,\pi)]} && \text{(Jensen'sche Ungleichung)} \\
&= 2^{-n} 2^{\frac{1}{n!} \sum_{\sigma \in S_n} r(\theta,\sigma)} \\
&\geq 2^{-n} 2^{\frac{j(\theta)}{k}} && \text{(wegen (3.1))}
\end{aligned}$$

Sei S nun die Menge der erfüllenden Belegungen von ϕ. Die W'keit, dass der Algorithmus *irgendeine* Belegung aus S findet, ist dann

$$\begin{aligned}
\sum_{\theta \in S} P(\text{Algorithmus findet } \theta) &\geq \sum_{\theta \in S} 2^{-n} 2^{\frac{j(\theta)}{k}} \\
&=^1 2^{-(n-\frac{n}{k})} \cdot \sum_{\theta \in S} 2^{\frac{-(n-j(\theta))}{k}} \\
&\geq^2 2^{-n(1-\frac{1}{k})} \cdot \underbrace{\sum_{\theta \in S} 2^{-s(\theta)}}_{\geq 1} \geq 2^{-n(1-\frac{1}{k})}.
\end{aligned}$$

(^{1}) gilt da: $-n + \frac{j(\theta)}{k} = \frac{-n+j(\theta)}{k} - n + \frac{n}{k} = \frac{-(n-j(\theta))}{k} - (n - \frac{n}{k})$.
(^{2}) gilt da: $s(\theta) = n - j(\theta)$ und k im Nenner wegfällt.
Die letzte Ungleichung folgt aus der obigen Behauptung. ■

Der Algorithmus wurde noch in Zusammenarbeit mit Saks 1998 und 2005 verbessert und erreicht eine Laufzeit von $O(1.36^n)$ für 3-SAT. Durch eine detailliertere Analyse zeigt Hertli 2011 die Schranke $O(1.308^n)$, was die zur Zeit beste obere Schranke für 3-SAT und probabilistische Algorithmen ist.

3.2 Klausellernen

Ein großes Problem beim Backtracking kann die wiederholte Berechnung bestimmter Informationen sein. Im ursprünglichen Algorithmus gehen also Information verloren, wenn einer solcher Rücksprung geschieht. Eine Möglichkeit um diesem Problem Herr zu werden ist zusätzliche Klauseln an die Formel zu fügen, die jene Informationen kodieren. Wird nun im weiteren Prozess eine dieser Klausel falsifiziert, dann müssen jede Belegungen nicht mehr betrachtet werden, die nun durch diese Klausel ausgeschlossen werden. Dies kann einiges an Rechenzeit einsparen. Dieses Prinzip nennt man *Klausellernen* oder auch *CDCL* für *conflict-driven clause-learning*. Dieses Verfahren geht zurück auf die Arbeit *J. P. Marques-Silva and Karem A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability, IEEE Trans. Computers, C-48, 5:506-521, 1999.*

Joao Marques-Silva
(c) UCD Dublin

Entsteht bei der Konstruktion einer partiellen Belegung θ im DPLL-Algorithmus ein Konflikt (eine Klausel evaluiert zu falsch), dann wird eine *Konfliktklausel* zu der Originalformel ϕ hinzugefügt, die eine logische Konsequenz von ϕ ist. Hierbei unterteilen wir die belegten Variablen in zwei Klassen:

Entscheidungsvariablen wurden in einem rekursiven Aufruf belegt,

Propagationsvariablen wurden durch Unit Propagation belegt.

Die *Stufe* einer Variable x ist die Anzahl der Entscheidungsvariablen, die vor x belegt wurden. Alle zu Beginn durch Unit Propagation belegte Variablen sind daher Stufe 0. Die *Konfliktstufe* ist die Stufe, in der ein Konflikt stattfindet, weil eine schon belegte Variable mit einem anderen Wert belegt werden soll. Auch für jede Propagationsvariable wird ihr Grund gespeichert: der Index derjenigen Klausel, welche durch die Belegung zu einer Unit-Klausel wurde und den Wert der Propagationsvariablen danach herbeiführte.

Implikationsgraph

Definition 11. Sei ϕ eine aussagenlogische Formel und θ eine Belegung. Dann ist der Implikationsgraph $G_I(\phi, \theta) = (V, E)$ ein gerichteter Graph, wobei $V = \{\ell \mid \theta \models \ell\}$, und

$$E = \left\{ (u, v) \,\middle|\, \begin{array}{l}\text{die Variable zu } v \text{ ist eine Propagationsvariable und} \\ \overline{u} \text{ ist in der Grundklausel für diese Variable } v\end{array} \right\}.$$

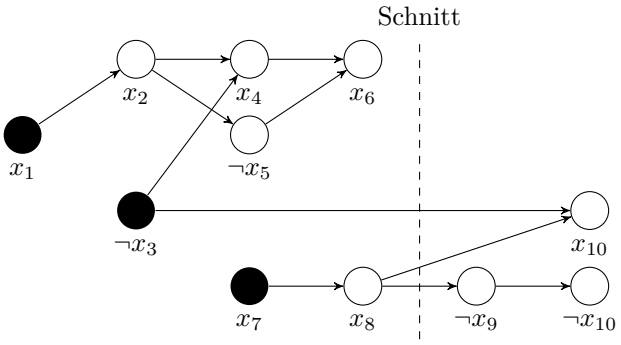
Bemerkung: Statt der Literale können genauso direkte Wertzuweisungen der Form $v = a$ in den Knoten stehen. ◀

Die Knoten der Literale der Entscheidungsvariablen haben Eingangsgrad 0. Im Konfliktfall bleibt ein Literal w, das schon mit dem Wert 0 belegt wurde, als letztes unbelegtes Literal in einer Klausel K übrig. In diesme Fall fügen wir entsprechende Knoten für w und $\overline{w}$ ein, und K ist in diesem Fall der Grund für w. Die Literale w und $\overline{w}$. Wir nennen w und $\overline{w}$ *Konfliktliterale*.

Beispiel 4. Sei

$$\phi = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee x_5 \vee x_6) \wedge (\neg x_7 \vee x_8) \wedge (\neg x_8 \vee \neg x_9) \wedge (x_9 \vee \neg x_{10}) \wedge (x_3 \vee \neg x_8 \vee x_{10}).$$

Angenommen im Algorithmus wird $x_1 = 1$ gewählt. Aus Unit Propagation folgt $x_2 = 1$ und $x_5 = 0$. Wird nun x_3 auf 0 gesetzt, folgen $x_4 = 1$ und $x_6 = 1$. Wird x_7 mit 1 belegt, folgt sofort $x_8 = 1, x_9 = 0, x_{10} = 1$ und auch $x_{10} = 0$. Es tritt also ein Konflikt für x_{10} auf. Der Implikationsgraph hierfür lautet (Entscheidungsvariablen in schwarz):



Karem A. Sakallah
(c) Uni Michigan

Ist ℓ das zugehörige Literal einer Entscheidungsvariable, die in der Konfliktstufe belegt wurde, dann heißt ein Knoten v *Unique Implication Point* (UIP), wenn alle Pfade von ℓ zu den Konfliktliteralen durch v verlaufen. Auch ℓ selbst ist ein UIP. Derjenige UIP, der am nächsten bei den Konfliktliteralen liegt, nennt man den *First UIP*. In unserem Beispiel gibt es zwei UIPs, die mit den Literalen x_7 und x_8 markiert sind. Hierbei ist x_8 der First UIP.

Betrachte einen Schnitt im Graphen, der Konfliktliterale von den übrigen Entscheidungsliteralen trennt. Wir nennen die eine Seite *Konflikt-* und die andere *Entscheidungsseite*. Alle Knoten auf der Konfliktseite können ein Konfliktliteral erreichen. Der Schnitt läuft über die Kanten zwischen der Entscheidungs- und der Konfliktseite. Diese Kanten sind für den Konflikt verantwortlich, da eine Belegung der Startknoten den Widerspruch erzeugt. In unserem Fall ist dies $\neg x_3 \wedge x_8$. Die Negation hiervon, also $x_3 \vee \neg x_8$ ist deswegen eine logische Konsequenz der Anfangsformel ϕ. Diese Klausel fügen wir deshalb zur Formel ϕ hinzu. Im Prinzip ist $x_3 \vee \neg x_8$ redundant, da sie von ϕ impliziert wird. Jedoch wenn später diese Klausel von einer partiellen Belegung falsifiziert wird, kann man daraus schließen, dass keine Erweiterung der aktuellen Belegung die Formel erfüllen kann. Diese Klausel bezeichnen wir daher als *gelernte Klausel*. ◀

3.3 Nicht-chronologisches Backtracking

Für einen Implikationsgraphen kann es mehrere mögliche Schnitte geben. Jeder dieser Schnitte produziert verschiedene Konfliktklauseln. Die Wahl dieser Konfliktklauseln kann das Backtracking beeinflussen. Manche SAT-Solver wählen die First UIP Konfliktklausel, welche von einem Schnitt produziert wird, der den First UIP und alle Knoten von Literalen, die vor der Entscheidungsstufe belegt wurden, auf der betreffenden Entscheidungsseite hat, und diejenigen Literale, die nach dem UIP belegt

wurden, auf der Konfliktseite hat. Dies ist auch unsere Klausel $x_3 \vee \neg x_8$ aus dem vorherigen Beispiel.

Sei K die Konfliktklausel unter einer partiellen Belegung θ und x die einzige Variable in K, die in der Konfliktstufe belegt wurde. Sei y die letzte Variable in K, die vor x belegt wurde und s die Stufe von y. Weil K gelernt wird, kann man im Backtracking-Prozess θ aktualisieren, indem man alle Belegungen, die in Stufen größer als s gemacht wurden, entfernt. Der Algorithmus geht also zurück zur Stufe s. Die neue partielle Belegung θ' falsifiziert alle Literale in K außer dem von x. Der Algorithmus belegt dann x mittels Unit Propagation, um K zu erfüllen.

Da x nicht die letzte Variable, die belegt wurde, sein muss, kann man mit dieser Idee einige Backtracking-Stufen überspringen und wird daher auch *nicht-chronologisches Backtracking* genannt. In unserem Beispiel gibt es drei Belegungsstufen. In der Konfliktklausel $x_3 \vee \neg x_8$ ist x_8 die Variable, die in der Konfliktstufe belegt wurde und der Algorithmus würde dann zu der Stufe der Variable x_3 zurückspringen. Mittels Unit Propagation wird dann erzwungen, die Variable x_8 mit dem Wert 0 zu belegen.

Algorithmus CDCL
Eingabe: Klauselmenge ϕ

```
1:  Implikationsgraph G_I = (∅, ∅), d = 0, θ = ∅;
2:  if UnitPropagation(φ, θ, G_I, d) == CONFLICT then return Unerfüllbar;
3:  else
4:      while φ[θ] ≠ ∅ do
5:          d++;
6:          Belege nächste Variable v mit einem Wert a und aktualisiere θ;
7:          Füge Knoten (v = a) zu G_I hinzu und setze mark(v = a) = d;
8:          while UnitPropagation(φ, θ, G_I, d) == CONFLICT do
9:              Bestimme Schnitt in G_I und Konfliktklausel K = (ℓ_1 ∨ ··· ∨ ℓ_k);
10:             Seien u_i die Knoten, die ℓ_i falsifizieren, 1 ≤ i ≤ k;
11:             b = max{mark(u_i) | 1 ≤ i ≤ k};
12:             if b == 0 then return Unerfüllbar;
13:             else
14:                 Entferne alle u aus V und θ mit mark(u) ≥ b;
15:                 Setze φ := φ ∪ {K}, d − −;
16:         d++;
17:     return Erfüllbar;
```

Algorithmus UnitPropagation
Eingabe: Klauselmenge ϕ, Belegung θ, Implikationsgraph G, Level d

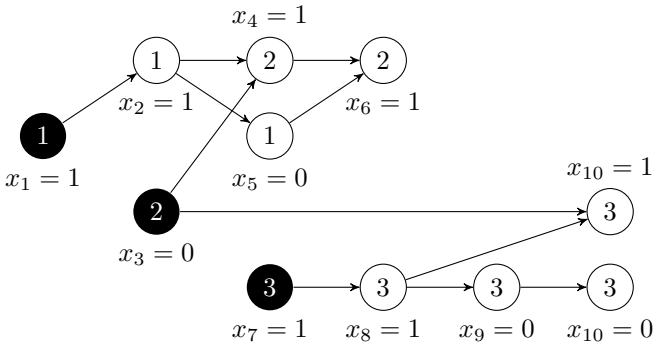
```
1:  while es gibt noch Einheitsklauseln in φ[θ] do
2:      Sei K = (ℓ) die erste Einheitsklausel in φ[θ];
3:      Ist ℓ = ¬x, dann setze a := 0, sonst a := 1;
4:      θ := θ ∪ {x = a};
5:      Sei K = (ℓ ∨ ℓ_1 ∨ ··· ∨ ℓ_k) die Klausel in φ (ohne angewendete Belegung).
6:      Füge Knoten x = a zu G hinzu und setze mark(x = a) = d;
7:      Füge Kanten (u, (x = a)) zu G hinzu, wobei u ℓ_i falsifiziert, 1 ≤ i ≤ k.
8:  if θ ⊭ φ then return CONFLICT end if
```

Beispiel 5. Spielen wir den Algorithmus CDCL an der schon bekannten Formel

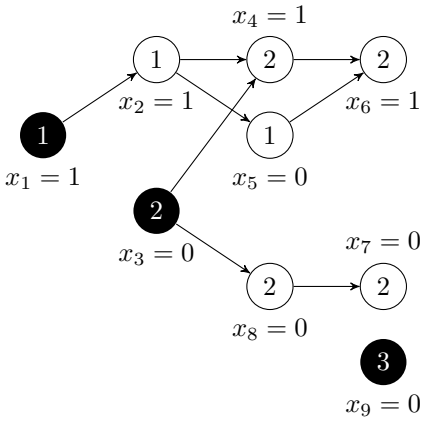
$$\phi = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_4 \vee x_5 \vee x_6) \wedge (\neg x_7 \vee x_8) \wedge (\neg x_8 \vee \neg x_9) \wedge (x_9 \vee \neg x_{10}) \wedge (x_3 \vee \neg x_8 \vee x_{10})$$

durch.

Wir erhalten zunächst folgende Situation (analog zu vorher), wobei das Entscheidungslevel im Knoten steht:



An dieser Stelle gibt UnitPropagation "CONFLICT" zurück. Nun wird der Schnitt bestimmt (wie vorher) und liefert die Konfliktklausel $K = (x_3 \vee \neg x_8)$. Der Wert von b ist dann 3. K wird zu ϕ hinzugefügt und der Implikationsgraph geändert (alle Knoten mit 2 werden gelöscht). Wir erhalten nun $x_8 = 0$, $x_7 = 0$ von UnitPropagation. Dann wählen wir $x_9 = 0$ und sind fertig.



Anschließend gibt CDCL "Erfüllbar" zurück. ◀

Kapitel 4

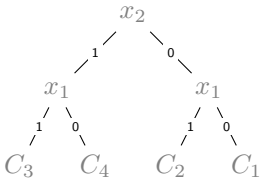
Baumartige Resolution und das Pigeonhole-Prinzip

In diesem Kapitel betrachten wir die Konsequenzen und Zusammenhänge von Resolution in Bezug auf DPLL Algorithmen und die Klasse NP. Wir folgen in großen Teilen der Argumentation in Beyersdorff u. a. (2010). Wie wir wissen produziert die DPLL Prozedur baumartige[1] Resolutionswiderlegungen auf KNF-Formeln. Hierbei gibt es nur eine Regel

$$\frac{C \vee x \qquad D \vee \neg x}{C \cup D}.$$

Beweise führen dann zu einer leeren Klausel für unerfüllbare Formeln und sind Bäume, in deren Wurzel die leere Klausel steht. Hier ein Beispiel für die Klauselmenge $\{x_1, x_2\}, \{\neg x_1, x_2\}, \{\neg x_1, \neg x_2\}, \{x_1, \neg x_2\}$ (wir nummerieren die Klauseln von links nach rechts durch):

Entscheidungsbaum



$$\dfrac{\dfrac{\{x_1,x_2\} \qquad \{\neg x_1,x_2\}}{\{x_2\}} \qquad \dfrac{\{\neg x_1,\neg x_2\} \qquad \{x_1,\neg x_2\}}{\{\neg x_2\}}}{\square}$$

Im Prinzip zielt alles auf den folgenden Satz hinaus, den wir ohne Beweis angeben.

Satz 15. Wenn es einen korrekten und widerlegungsvollständigen Kalkül $\mathcal{K}$, sowie ein Polynom p gibt, so dass für jede unerfüllbare Formel (oder Klauselmenge) ϕ der Nachweis von $\phi \vdash_{\mathcal{K}} \square$ durch einen Beweis möglich ist, der höchstens $p(n)$ Schritte lang ist, wobei n die Länge von ϕ ist, dann folgt NP = coNP. ◀

Wir betrachten im folgenden ein Spiel, mit dem wir anschließend zeigen, dass das diesem Satz geforderte Kalkül nicht die baumartige Resolution sein kann, da hier kein solches Polynom existiert. Wir werden sehen, dass die Argumentation über das Spiel deutliche Vorteile hat.

Delayer-Prover-Spiel

Definition 12. Sei ϕ unerfüllbar und θ eine partielle Belegung. Der *Delayer* behauptet er kennt eine erfüllende Belegung. Der *Prover* möchte einen Widerspruch finden. Das Spiel ist rundenbasiert:

[1] In baumartigen Ableitungen wird jede abgeleitete Klausel höchstens einmal als Vorbedingung der Resolutionsregel verwendet. Baumartige Resolutionswiderlegungen können als Boole'scher Entscheidungsbaum dargestellt werden. Dies ist ein Binärbaum, dessen innere Knoten mit Variablen der betrachteten Formel und die Blätter mit falsifizierten Klauseln beschriftet sind.

- Der Prover fragt nach einer Variable x.
- Der Delayer antwortet 0/1 oder überlässt dem Prover die Wahl. Wählt dieser, so bekommt er Punkte wie folgt:
  - Der Delayer wählt zwei Funktionen $c_0(x,\theta)$ und $c_1(x,\theta)$ mit $\frac{1}{c_0(x,\theta)} + \frac{1}{c_1(x,\theta)} = 1$ $(\star)$.
  - Der Delayer bekommt $\log c_b(x,\theta)$ Punkte wenn der Prover $b \in \{0,1\}$ wählt.
  - Der Delayer bekommt 0 Punkte wenn er selber wählt.
- Der Prover gewinnt, wenn die partielle Belegung eine Klausel von ϕ falsifiziert.

Diese Art von Spiel nennen wir (c_0, c_1)-Spiel auf ϕ. ◀

Die *Größe* einer Resolutionswiderlegung ist die Anzahl der verwendeten Klauseln.

Beyersdorff, Galesi, Lauria, 2010

Satz 16. Sei ϕ eine unerfüllbare Formel in KNF und seien c_0, c_1 zwei Funktionen im obigen Sinne für alle partiellen Belegungen θ für die Variablen von ϕ. Wenn ϕ eine baumartige Resolutionswiderlegung der Größe $\leq S$ hat, dann bekommt der Delayer $\leq \log S$ Punkte in jedem (c_0, c_1)-Spiel auf ϕ. ◀

Beweis Sei ϕ eine unerfüllbare KNF Formel mit $\mathrm{Vars}(\phi) = \{x_1, \ldots, x_n\}$ und Π eine baumartige Resolutionswiderlegung von ϕ. Angenommen Delayer und Prover spielen ein Spiel auf ϕ bei dem sie schrittweise die Belegung θ bauen. Sei θ_i die partielle Belegung, die nach i Runden des Spiels entstanden ist, d. h. θ_i belegt i Variablen mit 0/1. Mit p_i bezeichnen wir die Punktzahl des Delayers nach i Runden und Π_{θ_i} sei der Teilbaum des Entscheidungsbaumes von Π dessen Wurzel durch den Pfad von θ_i spezifiziert wird.

Wir beweisen nun mittels Induktion über die Anzahl der Runden des Spiels um die folgende Behauptung zu beweisen:

$$|\Pi_{\theta_i}| \leq \frac{|\Pi|}{2^{p_i}} \text{ für jede beliebige Runde } i.$$

Aus dieser Behauptung folgt der Satz aufgrund der folgenden Überlegungen. Sei θ eine im Spiel konstruierte Belegung, welche p_θ Delayer-Punkte erzeugt. Da ein Widerspruch gefunden wurde, ist die Größe von Π_θ genau 1 und aus der Behauptung folgt $1 \leq \frac{|\Pi|}{2^{p_\theta}} \Leftrightarrow p_\theta \leq \log|\Pi|$ wie gewünscht.

Zu Beginn des Spiels ist Π_{θ_0} der vollständige Baum und der Delayer hat 0 Punkte. Also gilt die Behauptung.

Für den Induktionsschritt nehmen wir an, die Behauptung gilt nach i Runden und der Prover fragt nach dem Wert der Variable x in Runde $i+1$. Wenn der Delayer den Wert selber wählt, dann gilt $p_{i+1} = p_i$ und damit

$$|\Pi_{\theta_{i+1}}| \leq |\Pi_{\theta_i}| \leq \frac{|\Pi|}{2^{p_i}} = \frac{|\Pi|}{2^{p_{i+1}}}.$$

Weigert sich der Delayer, dann befolgt der Prover die folgende Strategie um den Wert von x zu setzen. Der Prover setzt $x = 0$ wenn

$$|\Pi_{\theta_i \cup \{x=0\}}| \leq \frac{|\Pi_{\theta_i}|}{c_0(x,\theta_i)},$$

und sonst $x = 1$. Aus $\frac{1}{c_0(x,\theta)} + \frac{1}{c_1(x,\theta)} = 1$ folgt wenn $x = 1$ gesetzt wird, dass

$$|\Pi_{\theta_i \cup \{x=1\}}| \leq \frac{|\Pi_{\theta_i}|}{c_1(x, \theta_i)}.$$

Also, wenn die Wahl des Provers $x = j$ ist für $j \in \{0, 1\}$ dann erhalten wir

$$|\Pi_{\theta_{i+1}}| = |\Pi_{\theta_i \cup \{x=j\}}| \leq \frac{|\Pi_{\theta_i}|}{c_j(x, \theta_i)} \overset{\text{IV}}{\leq} \frac{|\Pi|}{c_j(x, \theta_i) \cdot 2^{p_i}} \overset{2^{\log}}{=} \frac{|\Pi|}{2^{p_i + \log(c_j(x,\theta_i))}} = \frac{|\Pi|}{2^{p_{i+1}}}.$$

■

Pigeonhole Prinzip

Definition 13. PHP_n^m mit $m > n$ ist eine Menge von Klauseln über den Variablen $x_{i,j}$ mit $1 \leq i \leq m$ und $1 \leq j \leq n$. Die Variable $x_{i,j}$ besagt, dass Taube i in Loch j sitzt. Wir setzen

$$\mathsf{PHP}_n^m := \left\{ \bigvee_{1 \leq j \leq n} x_{i,j} \;\middle|\; 1 \leq i \leq m \right\} \cup \{\neg x_{i_1,j} \vee \neg x_{i_2,j} \mid 1 \leq i_1 \neq i_2 \leq m, 1 \leq j \leq n\}.$$

Satz 17. Jede baumartige Resolutionswiderlegung von PHP_n^m hat die Größe $2^{\Omega(n \log n)}$. ◀

Beweis Sei θ eine partielle Belegung. Sei

$$E_i(\theta) = |\{j \in [n] \mid \theta(x_{i,j}) = 0 \text{ und } \theta(x_{i',j}) \neq 1 \text{ für alle } i' \in [m]\}|.$$

Intuitiv gesprochen ist $E_i(\theta)$ die Anzahl der noch freien Löcher, die aber explizit für Taube i durch θ ausgenommen sind (wir zählen keine Löcher die ausgenommen sind, weil eine andere Taube drin sitzt). Wir definieren nun

$$c_0(x_{i,j}, \theta) = \frac{\frac{n}{2} + 1 - E_i(\theta)}{\frac{n}{2} - E_i(\theta)} \quad \text{und} \quad c_1(x_{i,j}, \theta) = \frac{n}{2} + 1 - E_i(\theta).$$

Aus Einfachheit nehmen wir an n ist gerade. Während des Spiels wird nie der Fall eintreten, dass der Prover eine Wahl bekommt wenn $E_i(\theta) \geq \frac{n}{2}$. Deshalb sind die Funktionswerte von c_0, c_1 immer größer als 0 wenn der Delayer Punkte erhält, wonach die Punktefunktion immer wohldefiniert ist. Beachte, dass $(\star)$ für die c_i gilt.

Wir beschreiben nun die Delayer-Strategie in einem (c_0, c_1)-Spiel auf PHP_n^m. Fragt der Prover nach dem Wert von $x_{i,j}$, dann entscheidet sich der Delayer wie folgt

setzte $\theta(x_{i,j}) = 0$ falls es ein $i' \in [m] \setminus \{i\}$ gibt mit $\theta(x_{i',j}) = 1$, oder
falls es ein $j' \in [n] \setminus \{j\}$ gibt mit $\theta(x_{i,j'}) = 1$;

setzte $\theta(x_{i,j}) = 1$ falls $E_i(\theta) \geq \dfrac{n}{2}$ und es kein $i' \in [m]$ gibt mit $\theta(x_{i',j}) = 1$, und

Prover wählt sonst.

Intuitiv gesprochen, überlässt der Delayer dem Prover die Wahl so lange wie Taube i nicht in einem Loch sitzt, das Loch j immer noch frei ist und höchstens $\frac{n}{2}$ ausgenommene freie Löcher für Taube i existieren.

Wir wollen kurz erläutern wieso die Punkte so definiert werden. Die erste Beobachtungen umfasst, dass der Delayer immer mehr Punkte erhält, wenn der Prover eine Variable auf 1 statt auf 0 setzt. Dies ist korrekt, da die Menge an Freiheit für den Delayer um das Spiel fortzuführen viel stärker abnimmt, wenn man eine Taube i in ein Loch j setzt, als wenn man lediglich Loch j für Taube i ausschließt. Tatsächlich können unsere Punkte vollständig informationstheoretisch interpretiert werden: Wenn der Prover eine Taube in ein Loch setzt, sollte der Delayer immer ungefähr $\log n$ Punkte erhalten für diese Taube. Für unsere Delayer-Strategie bedeutet das Setzen einer Taube i in ein Loch, dass der Prover $\frac{n}{2}$ Löcher für Taube i ausgeschlossen hat oder, dass der Prover Taube i direkt in ein Loch gesetzt hat. Wenn wir das Spiel spielen, sollte der Delayer in jeder Runde eine Punktzahl proportional zu dem Fortschritt des Provers eine Taube i in ein Loch zu setzen, bekommen. Zum Beispiel, wenn der Prover gleich zu Beginn Taube i in ein Loch setzt, indem er $x_{i,j} = 1$ wählt, sollte der Delayer sofort $\log n$ Punkte erhalten.

Von der anderen Extremseite betrachtet, wenn der Prover bereits $\frac{n}{2} - 1$ Löcher für Taube i ausgeschlossen hat, dann spielt es keine Rolle, ob der Prover $x_{i,j}$ auf 0 oder 1 setzt, da nach beiden Antworten die Taube i in ein Loch gezwungen wird. Folglich erhält der Delayer in letzterem Fall nur 1 Punkt, egal ob der Prover antwortet mit 0 oder 1.

Verfolgt der Delayer unsere Strategie, dann werden die kleinen Krom-Klauseln $\neg x_{i_1,j} \vee \neg x_{i_2,j}$ in PHP_n^m nie falsifiziert werden. Deshalb wird ein Widerspruch immer durch eine der großen Klauseln $\bigvee_{j\in[n]} x_{i,j}$ auftreten. Angenommen das Spiel endet durch Falsifizieren von $\bigvee_{j\in[n]} x_{i,j}$, d. h. für Taube i sind alle Variablen $x_{i,j}$ mit $j \in [n]$ auf 0 gesetzt. Sobald die Anzahl $E_i(\theta)$ der ausgeschlossenen freien Löcher für Taube i den Schwellwert $\frac{n}{2}$ erreicht, wird der Delayer die Wahl nicht mehr dem Prover überlassen. Anstelle dessen wird der Delayer versuchen Taube i in ein Loch zu setzen. Anwortet der Delayer immer noch mit 0 für $x_{i,j}$ sogar wenn $E_i(\theta) > \frac{n}{2}$, dann muss der Fall eingetreten sein, dass bereits eine andere Taube in Loch j sitzt, d. h. für ein $i' \neq i$ gilt $\theta(x_{i',j}) = 1$. O.B.d.A. nehmen wir an, dass dies die Variablen x_{i,j_i} sind für $i = 1, \ldots, \frac{n}{2}$.

Wenden wir uns der Gesamtpunktzahl des Delayers zu. Wir berechnen diese separat für jede Taube $i = 1, \ldots, \frac{n}{2}$ und unterscheiden ob x_{i,j_i} auf 1 vom Delayer oder Prover gesetzt wurde. Angenommen der Delayer hat dies getan. Dann wurde Taube i noch keinem Loch zugeordnet und darüber hinaus muss es $\frac{n}{2}$ freie ausgeschlossene Löcher für Taube i durch θ geben, d. h. es gibt ein $J \subseteq [n]$ mit $|J| = \frac{n}{2}$, $\theta(x_{i',j'}) \neq 1$ für $i' \in [m], j' \in J$, da der Delayer eine 0 für $x_{i,j'}$ gewählt hat, als eine andere Taube bereits in Loch j' gesessen hat und dies ist nicht der Fall für Löcher in J (in dem Moment als der Delayer x_{i,j_i} auf 1 setzt). Folglich hat der Delayer bereits Punkte für alle $\frac{n}{2}$ Variablen $x_{i,j'}, j' \in J$ erhalten bevor er $\theta(x_{i,j_i}) = 1$ setzt. Hieraus folgen

$$\sum_{p=0}^{\frac{n}{2}-1} \log \frac{\frac{n}{2}+1-p}{\frac{n}{2}-p} = \log \prod_{p=0}^{\frac{n}{2}-1} \frac{\frac{n}{2}+1-p}{\frac{n}{2}-p} = \log\left(\frac{n}{2}+1\right)$$

Punkte für den Delayer. Beachte, dass der Delayer wirklich diese Punktzahl für *jede* der Variablen, die er auf 1 setzt bekommt, da er niemals zulässt, dass eine Taube in mehr als ein Loch kommt.

Wenn umgekehrt der Prover $x_{i,j_i} = 1$ setzt, dann bekommt der Delayer $\log(\frac{n}{2} + 1 - E_i(\theta))$ Punkte dafür, aber er erhält außerdem Punkte für die $E_i(\theta)$ Variablen, die zuvor auf 0 durch den Prover gesetzt wurden. Also, in diesem Fall erhält der Delayer für Taube i

$$\begin{aligned}\log\left(\frac{n}{2} + 1 - E_i(\theta)\right) + \sum_{p=0}^{E_i(\theta)-1} \log \frac{\frac{n}{2} + 1 - p}{\frac{n}{2} - p} \\ = \log\left(\frac{n}{2} + 1 - E_i(\theta)\right) + \log \frac{\frac{n}{2} + 1}{\frac{n}{2} - E_i(\theta) + 1} \\ = \log\left(\frac{n}{2} + 1\right)\end{aligned}$$

Punkte. Insgesamt erhält der Delayer mindestens

$$\frac{n}{2} \cdot \log\left(\frac{n}{2} + 1\right)$$

Punkte im Spiel. Aus Satz 16 und Kontraposition erhalten wir $2^{\frac{n}{2} \cdot \log(\frac{n}{2}+1)}$ als untere Schranke für die Größe jeder baumartigen Resolutionswiderlegung von PHP_n^m. ■

Kapitel 5

Lokale Suche

Wir folgen einerseits (Schöning u. Torán, 2012, Kapitel 5) und andererseits (Saß, 2013, Abschnitt 3.2). Ausgehend von einer initialen Belegung, die üblich nicht erfüllend ist, wollen wir über die nicht erfüllten Klauseln die Belegung verändern, sodass diese (hoffentlich) wieder erfüllt werden. Hierbei können wir zum Beispiel das Ziel haben, den Hamming-Abstand zur nächsten erfüllenden Belegung zu verringern oder die Anzahl der erfüllten Klauseln zu maximieren. Man muss beachten, dass beide Punkte nicht miteinander einhergehen wie folgendes Beispiel zeigt.

Beispiel 6. Die betrachtete Formel ist

$$\phi = (\neg x_1 \vee x_2) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_4) \wedge (x_1 \vee x_3)$$

und die Wahrheitstabelle der Formel ist (im Exponent steht eine nicht erfüllte Klausel, sofern $\theta \not\models \phi$)

ϕ	x_4	x_3	x_2	x_1
0^4	0	0	0	0
0^4	0	0	0	1
0^5	0	0	1	0
0^3	0	0	1	1
0^4	0	1	0	0
0^4	0	1	0	1
1	0	1	1	0
0^3	0	1	1	1

ϕ	x_4	x_3	x_2	x_1
0^5	1	0	0	0
0^1	1	0	0	1
0^5	1	0	1	0
0^3	1	0	1	1
0^2	1	1	0	0
0^3	1	1	0	1
1	1	1	1	0
0^3	1	1	1	1

Ist die Startbelegung $\theta(x_i) = 0$, dann ist die von der Hamming Distanz betrachtet nächste erfüllende Belegung $\theta_n(x_1) = \theta_n(x_4) = 0, \theta_n(x_2) = \theta_n(x_3) = 1$. Nun falsifiziert θ die letzten beiden Klauseln, d. h. alle Variablen stehen für eine mögliche Änderung des Wahrheitswertes zur Auswahl.

Setzen wir $\theta(x_1) = \theta(x_4) = 1$, dann erfüllen wir alle bis auf die erste Klausel, also haben wir die Anzahl der Klauseln erhöht. Der Hamming-Abstand zur nächsten erfüllenden Belegung ist jedoch gleich geblieben (wenn auch nun die nächste erfüllende Belegung $\theta'_n(x_i) = 1$ ist und der Abstand zu θ_n auf 4 gewachsen ist).

Sethen wir $\theta(x_2) = 1$, dann verringert sich der Hamming-Abstand zu θ_n auf 1 und die Anzahl der erfüllenden Klauseln steigt auf 4 an. ◀

Der Erwartungswert vom Hamming-Abstand zu einer fixen erfüllenden Belegung bei einer zufälligen Startbelegung ist $\frac{n}{2}$ (vergleiche Satz 10). Wählen wir eine zufällige Variable und ändern ihren Wert, so verkleinert sich der Hamming-Abstand mit W'keit $\frac{1}{2}$.

Sind wir jedoch (im Laufe eines Verfahrens) zum Beispiel nur $\frac{n}{20}$ von einer erfüllenden Belegung entfernt, dann besteht die W'keit durch das Ändern einer zufälligen Variable noch näher zu kommen nur bei $5\% = \frac{1}{20}$.

5.1 Deterministische lokale Suche

Diese Algorithmen sind *vollständig*, d. h. ihre Ausgabe *erfüllbar* bzw. *unerfüllbar* ist korrekt.

Algorithmus LocalSearch
Eingabe: Klauselmenge ϕ, Belegung θ, Radius p
Ausgabe: 1, falls es ein $\theta' \models \phi$ gibt mit $d(\theta, \theta') \leq p$
1: **if** $\theta \models \phi$ **then return** θ; **end if**
2: **if** $p = 0$ **then return** 0; **end if**
3: Sei $K = (\ell_1 \vee \ell_2 \vee \ell_3)$ in ϕ eine Klausel mit $\theta \not\models K$;
4: **for** $i = 1, \ldots, 3$ **do**
5: **if LocalSearch**$(\phi, \theta[\ell_i = 1], p - 1)$ **then return** 1; **end if**
6: **return** 0;

Üblicherweise hat p die Form $\delta \cdot n$ für $\delta \leq \frac{1}{2}$ und $n = |\text{Vars}(\phi)|$. Der Algorithmus deckt also den folgenden Bereich von Belegungen ab:

$$H_P(\theta) := \{\theta' \mid d(\theta, \theta') \leq p\}.$$

Wir nennen diesen Bereich *Hamming-Kugel um Belegung θ mit Radius p*. Die Laufzeit von LocalSearch ist gegeben durch die Rekursion $T(p) = 3 \cdot T(p-1)$, also eine Laufzeit von $O^*(3^p)$. Für k-KNF Formeln ergibt dies $O^*(k^p)$.

Satz 18. Es gibt einen 3SAT Algorithmus, der in Zeit $O(1.733^n)$ läuft. ◀

Wieso ist der Algorithmus bereits für 4-KNF Formeln unnötig?

Beweis

Algorithmus LS-SAT
Eingabe: Klauselmenge ϕ
1: **if not LocalSearch**$(\varphi, \{x_1 = 1, \ldots, x_n = 1\}, \lceil \frac{|\text{Vars}(\phi)|}{2} \rceil)$ **then**
2: **return LocalSearch**$(\varphi, \{x_1 = 0, \ldots, x_n = 0\}, \lceil \frac{|\text{Vars}(\phi)|}{2} \rceil)$

Aus den vorherigen Beobachtungen läuft der Algorithmus in Zeit

$$O^*(3^{\frac{n}{2}}) = O(1.733^n) = O(2^{0.793n}).$$

■

Optimierungen

- Verhindere Back-Flipps durch Änderung in Zeile 5 zu

  "**if LocalSearch**$(\phi[\ell_i = 1], \theta[\ell_i = 1], p - 1)$ **then return** 1; **end if**"

- Bei der Wahl der Klausel K, sollten eher eine kürzeste Klausel dieser Art gewählt werden.

5.2 Random Walk

Satz 19. Es gibt einen probabilistischen Algorithmus für k-SAT mit einer Laufzeit von $O^*((2 \cdot (1 - \frac{1}{k}))^n)$, d. h. für 3SAT eine Laufzeit von $O^*((\frac{4}{3})^n)$. ◀

Beweis Betrachten wir die folgende Änderung des bekannten Algorithmus:

Algorithmus RandomWalk
Eingabe: Klauselmenge ϕ
1: **for** $i = 1, \ldots, t$ **do**
2: Wähle zufällige Anfangsbelegung θ;
3: **for** $j = 1, \ldots, |\text{Vars}(\phi)|$ **do**
4: **if** $\theta \models \phi$ **then return** "erfüllbar"; **end if**
5: Wähle eine Klausel $K = (\ell_1 \vee \ell_2 \vee \ell_3)$ mit $\theta \not\models K$;
6: Wähle zufällig $k \in \{1, 2, 3\}$;
7: $\theta := \theta[\ell_k = 1]$;
8: **return** "unerfüllbar";

LS-SAT hat bisher eine Hamming-Kugel mit Radius p in 3^p Schritten durchsucht. Gibt es genau eine erfüllende Belegung in diesem Hamming-Abstand, so würde RandomWalk über p Schritte diese Belegung mit W'keit von 3^{-p} finden. Jede Wahl von einem ℓ_k müsste dabei jedes mal genau "treffen".

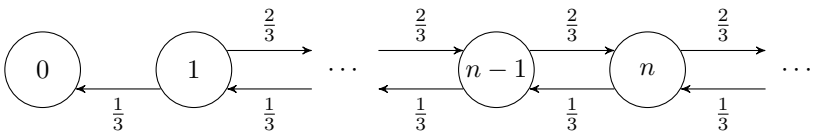
Ein größer Vorteil vom Algorithmus ist jedoch die folgende Beobachtung. Erhöhen wir die Suchdauer auf $3p$ Schritt, dann vergrößert dies die Gesamtlaufzeit nur um eine Konstante, wohingegen die W'keit die einzige erfüllende Belegung zu finden immens ansteigt: Innerhalb von $3p$ Schritten dürfen p Literal-Auswahlen falsch sein, wenn $2p$ mal richtig gewählt wird. Die Erfolgsw'keit ist damit

$$\binom{3p}{2p} \cdot \left(\frac{1}{3}\right)^{2p} \cdot \left(\frac{2}{3}\right)^{p} \approx 2^{-p}.$$

Die optimale Wahl von p ist $\frac{n}{3}$ (also wird die innere for-Schleife $3p$ mal ausgeführt).

Wir nennen einen Durchlauf der äußeren for-Schleife einen *Lauf*. Wir müssen nun die W'keit $q(n)$ bestimmen innerhalb eines solchen Laufs eine erfüllende Belegung zu finden. Die Wiederholungszahl t kann dann mit $O(\frac{1}{q(n)})$ festgelegt werden – dann haben wir ausreichende Sicherheit.

Sei ϕ erfüllende Formel und θ' eine erfüllende Belegung. In jeder Klausel $K = (\ell_1 \vee \ell_2 \vee \ell_3)$ gibt es mindestens ein Literal ℓ_k, das unter θ' den Wert 1 erhält. Fixiere genau ein solches Literal in jeder Klausel K. Die W'keit, dass der Algorithmus genau dies findet, ist $\frac{1}{3}$. Bei dieser Auswahl verringert sich der Hamming-Abstand zu θ' um 1. Dies ergibt die folgende Markov-Kette:



Durch die Wahl der Anfangsbelegung legen wir einen bestimmten Hamming-Abstand zu θ' fest, der zwischen 0 und n liegt (gemäß symmetrischer Binomialverteilung). Jeder Zustand wird also mit W'keit $\binom{n}{j} \cdot 2^{-n}$ eingenommen. Hiernach finden n zufällige

Schritte statt. Das richtige Literal wird mit W'keit $\frac{1}{3}$ gewählt, welches uns von Zustand j nach $j-1$ bringt; ein falsches führt von j zu $j+1$. Hierbei können wir auch Werte größer als n annehmen, da wir Variablen wieder zurückflippen müssen, die fälschlicherweise geflippt wurden.

Es können zwei Ereignisse eintreten:

E_1: Wir haben einen initialen Hamming-Abstand von genau $\frac{n}{3}$.

E_2: Wir starten im Zustand $\frac{n}{3}$ und sind nach n zufälligen Schritten im Zustand 0, wobei innerhalb der Schritt genau $\frac{n}{3}$ mal falsch und $\frac{2n}{3}$ mal richtig gewählt wurde.

Der erhaltene Hamming-Abstand ist binomialverteilt (also $d(\theta,\theta') \sim Bin(n,\frac{1}{2})$), woraus sich folgende W'keiten ergeben:

$$P(E_1) = \binom{n}{\frac{n}{3}} \cdot 2^{-n} \quad \text{und} \quad P(E_2) = \binom{n}{\frac{n}{3}} \left(\frac{1}{3}\right)^{\frac{2n}{3}} \cdot \left(\frac{2}{3}\right)^{\frac{n}{3}}.$$

Daraus erhalten wir:

$$\begin{aligned} P(\text{Erfolg bei einem Lauf}) &\geq P(E_1 \wedge E_2) = P(E_1) \cdot P(E_2) \\ &= \binom{n}{\frac{n}{3}} \cdot 2^{-n} \cdot \binom{n}{\frac{n}{3}} \left(\frac{1}{3}\right)^{\frac{2n}{3}} \cdot \left(\frac{2}{3}\right)^{\frac{n}{3}} \end{aligned}$$

Nun gilt $\binom{n}{\frac{n}{3}} \overset{poly}{=} 2^{h(\frac{1}{3})\cdot n}$ für $h(x) = -x\log_2(x) - (1-x)\log_2(1-x)$ nach Shannon und damit

$$\begin{aligned} &\overset{poly}{=} 2^{h(\frac{1}{3})n} \cdot 2^{-n} \cdot 2^{h(\frac{1}{3})n} \left(\frac{1}{3}\right)^{\frac{2n}{3}} \cdot \left(\frac{2}{3}\right)^{\frac{n}{3}} \\ &= \left(\frac{3}{4}\right)^n \qquad \text{(Übungsaufgabe)} \end{aligned}$$

Aus der Binomialverteilung $Bin(n,p) = \begin{pmatrix} 0 & 1 & \cdots & n \\ q_0 & q_1 & \cdots & q_n \end{pmatrix}$ wissen wir, dass

$$q_j = \binom{n}{j} \cdot p^j (1-p)^{n-j}$$

gilt. Demnach ist die W'keit nach t Wiederholungen *keine* erfüllende Belegung zu finden:

$$\begin{aligned} P(X=0) &= \binom{t}{0} \left(\left(\frac{3}{4}\right)^n\right)^0 \cdot \left(1 - \left(\frac{3}{4}\right)^n\right)^{t-0} \\ &= \left(1 - \left(\frac{3}{4}\right)^n\right)^t \leq e^{-\left(\frac{3}{4}\right)^n \cdot t} \end{aligned}$$

Wir wollen, dass die W'keit nicht größer als e^{-c} für ein $c \in \mathbb{N}$ wird. Also müssen wir die Wiederholungszahl

$$t \geq \frac{c}{\left(\frac{3}{4}\right)^n} = c \cdot \left(\frac{4}{3}\right)^n$$

wählen. Wählen wir nun $t = 20 \cdot \frac{4}{3}^n$, dann beträgt die W'keit, bei erfüllbarer Eingabeformel keine erfüllende Belegung zu finden, höchstens e^{-20}. ■

5.3 Greedy Algorithmen und Scores

Selman u. a. (1996) und (Schöning u. Torán, 2012, Kapitel 5.6) sind die Quellen diesen Abschnitts. Im folgenden betrachten wir Strategien, welche gierig verfahren und hierbei sogenannte *Scores*, wie z. B. Anzahl von Vorkommen der betreffenden Variablen in erfüllten bzw. unerfüllten Klauseln oder auch Länge der Klauseln einsetzen. Hierzu läuft man zu Beginn, nach Wahl der Startbelegung θ, über die Formel um eine Variable x zu finden, für die der Wert

$$\text{score}_\theta(x) = |\{K \mid K \text{ ist eine durch } \theta[x] \text{ unerfüllte Klausel}\}|$$

minimal ist, wobei $\theta[x]$ die Belegung ist, bei der für die Belegung θ der zugewiesene Wahrheitswert von x gedreht wird, d. h. ist $\theta(x) = 1$, dann ist $\theta[x](x) = 0$ und umgekehrt. Für diese Variable wird der Wert dann geflippt.

Bram Cohen
Creative Commons Attribution-Share Alike 2.0 Generic

Algorithmus GSAT
Eingabe: Klauselmenge ϕ, Integer maxFlips, Integer maxTries

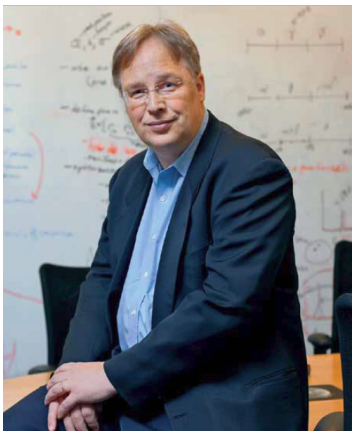
```
1: for i = 1,...,maxTries do
2:     Sei θ eine zufällige Startbelegung
3:     for j = 1,...,maxFlips do
4:         if θ ⊨ φ then return θ; end if
5:         Wähle zufällige Variable x ∈ min_{x∈Vars(φ)} |score_θ(x)|
6:         θ ← θ[x];
7: return Keine erfüllende Belegung gefunden;
```

Selman u. a. beschreiben, dass dieser Algorithmus an lokalen Minima hängen bleibt. Der folgende Algorithmus zeigt eine deutliche Effizienzsteigerung.

Henry Kautz
(c) Uni Rochester

Algorithmus WalkSAT
Eingabe: Klauselmenge ϕ, Integer maxFlips, Integer maxTries

```
1: for i = 1,...,maxTries do
2:     Sei θ eine zufällige Startbelegung
3:     for j = 1,...,maxFlips do
4:         if θ ⊨ φ then return θ; end if
5:         Wähle unter den Klauseln K mit θ ⊭ K eine zufällige aus.
6:         if ∃x ∈ Vars(K), sodass es keine Klausel K' gibt mit θ ⊨ K' und
   θ[x] ⊭ K' then
7:             θ ← θ[x];
8:         else
9:             Entscheide zufällig:
10:            p: wähle unter Gleichverteilung eine Variable x in K;
11:            1 − p: Wähle zufällige Variable x ∈ min_{x∈Vars(φ)} |score_θ(x)|;
12:            θ ← θ[x];
13: return Keine erfüllende Belegung gefunden;
```



Bart Selman
(c) Cornell University

Es hat sich herausgestellt, dass die W'keit $p = 0.57$ ein guter Wert ist.

Kapitel 6

Ein Divide and Conquer Ansatz

Wir betrachten nun einen Algorithmus von Pudlák aus (Pudlák, 1998, Abschnitt 2.3), der auch in (Schöning u. Torán, 2012, Abschnitt 6.1) erläutert wird.

Pavel Pudlák
(c) Academy of Science Czech Republic

Algorithmus DivideConquer-SAT
Eingabe: ϕ in 3KNF, $\text{Vars}(\phi) = \{x_1, \ldots, x_n\}$ mit n gerade, reelle Zahl $0 < \delta < 1$

Sei $V_1 = \{x_1, \ldots, x_{\frac{n}{2}}\}$, und $V_2 = \{x_{\frac{n}{2}+1}, \ldots, x_n\}$.
$K_1 := \{K \in \phi \mid \text{Vars}(K) \subseteq V_1\}$, $K_2 := \{K \in \phi \mid \text{Vars}(K) \subseteq V_2\}$
$K_3 := \{K = (\ell_1 \vee \ell_2 \vee \ell_3) \in \phi \mid \text{Vars}(\{\ell_1, \ell_2\}) \subseteq V_1, \text{Vars}(\{\ell_3\}) \subseteq V_2\}$.
$K_4 := \{K = (\ell_1 \vee \ell_2 \vee \ell_3) \in \phi \mid \text{Vars}(\{\ell_1, \ell_2\}) \subseteq V_2, \text{Vars}(\{\ell_3\}) \subseteq V_1\}$.
$A := \{\theta \mid \theta \models K_1\}$, $B := \{\theta \mid \theta \models K_2\}$.
for $\theta \in A$ **do**
    Wende auf θ Unit Propagation an.
    if $\theta \not\models K_3$ **then** $A := A \setminus \{\theta\}$. **end if**
for $\theta \in B$ **do**
    Wende auf θ Unit Propagation an.
    if $\theta \not\models K_4$ **then** $B := B \setminus \{\theta\}$. **end if**
FindConsistentPair1(ϕ, A, B, δ)
FindConsistentPair1(ϕ, B, A, δ)
FindConsistentPair2(ϕ, A, B, δ)

Die Ausführung der letzten beiden **FindConsistentPair**-Aufrufe als naiven Vergleich der Elemente aus A mit denen aus B führt nicht zu einem besseren Algorithmus als der BruteForce-Ansatz, da wir $O^*(2^{\frac{n}{2}} \cdot 2^{\frac{n}{2}}) = O^*(2^n)$ Vergleiche durchführen müssten. Im folgenden definieren wir diese Algorithmen genauer und berechnen anschließend einen Wert für δ fest

Algorithmus FindConsistentPair1
Eingabe: ϕ in 3KNF, Belegungsmengen A, B, reelle Zahl $0 < \delta \leq 12$
Ausgabe: Konsistentes Paar in (A, B) sofern vorhanden

for $\theta \in A$ mit $n - |\text{Vars}(\theta)| \leq (1 - \delta) \cdot \frac{n}{2}$ **do**
    for jede vollständige Erweiterung von θ **do**
        if es gibt ein passendes $\theta' \in B$ **then return** (θ, θ') **end if**

Behauptung. **FindConsistentPair1** benötigt $O(2^{(1-\frac{\delta}{2})n})$ Schritte.

Beweis Für jedes θ gibt es höchstens $2^{(1-\delta)\cdot\frac{n}{2}}$ mögliche Erweiterungen. Damit der Algorithmus im Sinne der Behauptung läuft, verwenden wir einen binären Suchbaum für die Belegungen in B. Der Baum hat Tiefe $\frac{n}{2}$ und in jeder Stufe wird gemäß einer Variablen in V_2 verzweigt. Jede Belegung $\theta' \in B$ definiert einen Pfad von der Wurzel zu einem Blatt im Baum. In den Blättern sind die Werte der Variablen in V_1 gespeichert, d. h. haben Werte der Form $0, 1$ oder frei. Für jede Vervollständigung von θ kann über die Suche im Baum in n Schritten getestet werden, ob es für sie ein konsistentes Paar in B gibt. Für alle partiellen Belegungen θ aus A und alle möglichen Vervollständigungen braucht der Algorithmus damit $O^*(2^{\frac{n}{2}} \cdot 2^{(1-\delta)\cdot\frac{n}{2}}) = O(2^{(1-\delta)\cdot n})$ Schritte. ■

Der folgende Algorithmus erläutert die Handhabung der übrigen Belegungen.

Algorithmus FindConsistentPair2
Eingabe: ϕ in 3KNF, Belegungsmengen A, B, reelle Zahl $0 < \delta < 1$
Ausgabe: Erfüllbar gdw. ein konsistentes Paar in (A, B) gibt

1: $A', B' \leftarrow \emptyset$.
2: **for** $\theta \in A$ mit $n - |\mathrm{Vars}(\theta)| > (1-\delta)\cdot\frac{n}{2}$ **do**
3: **for** alle $V' \subseteq V_1$ mit $|V'| \leq \frac{\delta n}{2}$ **do**
4: Sei θ' die Belegung in der alle Variablen aus $V_1 \setminus V'$ gelöscht werden.
5: $A' \leftarrow A' \cup \{\theta'\}$.
6: **for** $\theta \in B$ mit $n - |\mathrm{Vars}(\theta)| > (1-\delta)\cdot\frac{n}{2}$ **do**
7: **for** alle $V' \subseteq V_2$ mit $|V'| \leq \frac{\delta n}{2}$ **do**
8: Sei θ' die Belegung in der alle Variablen aus $V_2 \setminus V'$ gelöscht werden.
9: $B' \leftarrow B' \cup \{\theta'\}$.
10: **if** $A' \cap B' \neq \emptyset$ **then return** erfüllbar **end if**

Behauptung. **FindConsistentPair2** benötigt $O^*(2^{\frac{n}{2}+h(\delta)\cdot\frac{n}{2}})$ Schritte.

Beweis Insgesamt gibt es

$$\sum_{i\leq\frac{\delta n}{2}} \binom{\frac{n}{2}}{i}$$

viele Teilmengen der Form $V' \subseteq V_j$ für $j = 1, 2$ mit $|V'| \leq \frac{\delta n}{2}$. Da wir für jede Belegung in A (bzw. B) dies tun und nach Aufruf $|A| = 2^{\frac{n}{2}}$ (analog B) gilt, müssen $2^{\frac{n}{2}}$ viele solche Teilmenge konstruieren (analog für B). Für die Schnittbetrachtung in der letzten Zeile betrachten wir die effiziente Sortierung von A' und B' und erhalten dadurch lediglich einen polynomiellen zusätzlichen Faktor in der Gesamtlaufzeit, also

$$O^*\left(2^{\frac{n}{2}} \cdot \sum_{i\leq\frac{\delta n}{2}} \binom{\frac{n}{2}}{i}\right) = O^*(2^{\frac{n}{2}+h(\delta)\cdot\frac{n}{2}}),$$

da $\sum_{i=0}^{\delta n} \binom{n}{i} \overset{poly}{=} 2^{h(\delta)n}$ gilt.

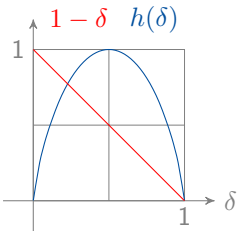
■

Wir wollen nun δ so bestimmen, dass die Komplexität von **DivideConqueor-SAT** minimal wird. Dies ist der Fall, wenn die Laufzeiten von **FindConsistentPair1** und

FindConsistentPair2 gleich sind, d. h.

$$
\begin{aligned}
& & 2^{(1-\frac{\delta}{2})\cdot n} &= 2^{\frac{n}{2}+h(\delta)\cdot\frac{n}{2}} \\
\Leftrightarrow & & \left(1-\frac{\delta}{2}\right)\cdot n &= \frac{n}{2}+h(\delta)\cdot\frac{n}{2} \\
\Leftrightarrow & & 1-\frac{\delta}{2} &= \frac{1}{2}+h(\delta)\cdot\frac{1}{2} \\
\Leftrightarrow & & 2-\delta &= 1+h(\delta) \\
\Leftrightarrow & & h(\delta) &= 1-\delta
\end{aligned}
$$

Also gilt



$$
\begin{aligned}
& & h(\delta) &= 1-\delta \\
\Leftrightarrow & & -\delta\cdot\log_2(\delta)-(1-\delta)\log_2(1-\delta) &= 1-\delta \\
\Leftrightarrow & & \delta &\approx 0.22709
\end{aligned}
$$

Daraus ergibt sich eine Laufzeitkomplexität von $O(1.8486^n)$.

Kapitel 7

Enumeration

In diesem Kapitel beschäftigen wir uns mit einer neuen Art von Problemen, sogenannten *Enumerationsproblemen*. Wir sind daran interessiert *alle* Lösungen zu einem Entscheidungsproblem auszugeben. Die entscheidende Frage hierbei ist: *Wie viel Zeit vergeht zwischen zwei Lösungen?* Dies ist die Frage nach dem *Delay*. Wir werden zunächst ein formales Framework definieren und anschließend betrachten wie das zu SAT zugehörige Enumerationsproblem sich verhält.

Enumerationsproblem

Definition 14. Ein *Enumerationsproblem* ist ein Tupel $E = (I, \text{Sol})$, wobei

- I die Menge der Instanzen und
- Sol eine Funktion ist, sodass für alle $x \in I$ $\text{Sol}(x)$ eine endliche Menge der Lösungen von x ist. ◀

Nun betrachten wir Enumerationsalgorithmen.

Enumerationsalgorithmus

Definition 15. Sei $E = (I, \text{Sol})$ ein Enumerationsproblem. Ein Algorithmus $\mathcal{A}$ ist ein *Enumerationsalgorithmus für* E, wenn das folgende gilt für jedes $x \in I$ gilt:

- $\mathcal{A}(x)$ terminiert nach einer endlichen Anzahl von Schritten.
- $\mathcal{A}(x)$ gibt genau die Elemente von $\text{Sol}(x)$ ohne Duplikate aus. ◀

Bevor Komplexitätsklassen definiert werden können, benötigen wir den Begriff des Delays.

Definition 16. Sei $E = (I, \text{Sol})$ ein Enumerationsproblem und $\mathcal{A}$ ein Enumerationsalgorithmus für E. Sei $x \in I$ eine Instanz. Dann definieren wir

- den i-ten Delay von $\mathcal{A}$ als die Zeit zwischen der Ausgabe der i-ten und $(i+1)$-ten Lösung in $\text{Sol}(x)$,
- den 0-ten Delay als die *Vorberechnungszeit*, was die Zeit ist bevor die erste Ausgabe produziert wird und
- den n-ten Delay als die *Nachberechnungszeit*, was die Zeit ist nachdem die letzte Ausgabe produziert wurde bis der Algorithmus terminiert. ◀

Definition 17. Sei $E = (I, \text{Sol})$ ein Enumerationsproblem und $\mathcal{A}$ ein Enumerationsalgorithmus für E.

(i) E gehört zur Klasse TotalP wenn es ein Polynom p gibt, sodass $\mathcal{A}$ für jedes $x \in I$ alle Lösungen in $\mathrm{Sol}(x)$ in Zeit $O(p(|x|))$ ausgibt.

(ii) E gehört zur Klasse DelayP wenn es ein Polynom p gibt, sodass $\mathcal{A}$ für jedes $x \in I$ alle Lösungen in $\mathrm{Sol}(x)$ mit einem Delay von $O(p(|x|))$ ausgibt. ◀

Sei EnumKrom-SAT das zu Krom-SAT zugehörige Enumerationsproblem, alle erfüllenden Belegungen einer Krom-Formel auszugeben.

Satz 20. EnumKrom-SAT $\in$ DelayP. ◀

Beweis Im folgenden konstruieren wir einen Enumerationsalgorithmus, der das Konzept der *Selbst-Reduzierung* nutzt. Außerdem verwenden wir das Resultat aus Satz 11, in dem gezeigt wurde, dass Krom-SAT $\in$ P.

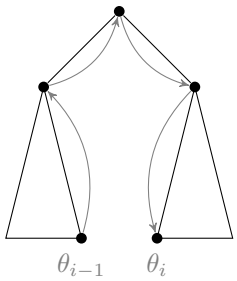
Algorithmus EnumKrom
Eingabe: Krom-Formel ϕ, Partielle Belegung θ
1: **if** $\phi[\theta] \equiv 1$ und $\mathrm{Vars}(\theta) = \mathrm{Vars}(\phi)$ **then print** θ. **end if**
2: **if** $\phi[\theta]$ ist erfüllbar und $\mathrm{Vars}(\theta) \neq \mathrm{Vars}(\phi)$ **then**
3: Wähle Variable $x \in \mathrm{Vars}(\phi[\theta])$.
4: **EnumKrom**$(\phi, \theta \cup \{x = 0\})$.
5: **EnumKrom**$(\phi, \theta \cup \{x = 1\})$.

Der erste Aufruf findet mit **EnumKrom**$(\phi, \emptyset)$ statt.

Behauptung. Der i-te Delay vom Algorithmus **EnumKrom** auf Eingabe $(\phi, \emptyset)$ ist polynomiell.

Beweis (Behauptung) Seien p, q die Polynome, die zeigen, dass die Evaluation einer Formel in P ist bzw. dass Krom-SAT $\in$ P gilt. Weiterhin sei $T = (V, E)$ der Binärbaum, der die rekursiven Aufrufe von **EnumKrom** beschreibt. Die Tiefe von T ist $|\mathrm{Vars}(\phi)|$. O.B.d.A. sei $x_1, \ldots, x_n$ die Reihenfolge, in der die Variablen gewählt werden. D. h. wenn wir die rekursiven Aufrufe im $i$ten Level von T betrachten, wählen alle rekursiven Aufrufe in dieser Tiefe immer die Variable x_i. Betrachten wir die ausgegebenen erfüllenden Belegungen während der Berechnung, dann fällt auf, dass diese in lexikographischer Reihenfolge ausgegeben werden (da zuerst für eine Variable $x = 0$ geprüft wird).

Fall 1



Seien nun θ_{i-1}, θ_i die Belegungen, die vor bzw. nach dem i-ten Delay ausgegeben werden und seien $y = y_1 \cdots y_n$ und $z = z_1 \cdots z_n$ mit $y_i, z_i \in \{0, 1\}$ die Bitvektoren, die die Belegungen θ_{i-1} und θ_i beschreiben. Der größtmögliche lexikographische Abstand zwischen y und z kann $2^n - 1$ sein, nämlich genau dann wenn $y = 0 \cdots 0$ und $z = 1 \cdots 1$ gilt. Allgemeiner, ist z die lexikographisch nächste Belegung nach y, so ist der i-te Delay im schlimmsten Fall $O(n \cdot (p(|\phi|) + q(|\phi|)))$, wenn $y = 01 \cdots 1$ und $z = 10 \cdots 0$ und wir den gesamten Bewegungsraum von der Wurzel bis zum Blatt durchgehen müssen (Fall 1).

Andererseits gibt es Belegungen $\chi_1, \ldots, \chi_k$, die genau die $k \in \{1, 2^n - 2\}$ lexikographischen Belegungen zwischen θ_{i-1} und θ_i sind, sodass für alle $1 \leq \ell \leq k$ gilt $\chi_\ell \not\models \phi$ und diese daher nicht ausgegeben werden. Im folgenden wollen wir betrachten wie groß der Delay nun sein kann (und zeigen, dass dieser nicht exponentiell ist).

Gilt in einem rekursiven Aufruf, dass es keine erfüllende Belegung für $x_i = c$ gibt, dann schließt dies 2^{n-i} Belegungen aus, die der Algorithmus nie betrachtet. Also

können wir nach dem folgenden Schema die Belegungen zusammenfassen: Sei $p_1 \cdots p_r$ der gemeinsame Präfix von y und z mit $r \in \{0, \ldots, n-2\}$, wobei $r = 0$ bedeutet, dass es keinen gemeinsamen Präfix gibt (d. h. $y_1 = 0$ und $z_1 = 1$).

(i) Nun wiederholen wir die folgenden Schritte für $i = n, \ldots, r+2$:
   a) Ist $y_i = 0$, dann gruppiere 2^{n-i} Aufrufe der Form $y_1 \cdots y_{i-1} 1 e_1 \cdots e_{n-i}$ mit $e_j \in \{0, 1\}$.
   b) Ist $y_i = 1$, dann tue nichts.

(ii) Nun wiederholen wir die folgenden Schritte für $i = r+2, \ldots, n$:
   a) Ist $z_i = 0$, dann tue nichts.
   b) Ist $z_i = 1$, dann gruppiere 2^{n-i} Aufrufe der Form $z_1 \cdots z_{i-1} 0 e_1 \cdots e_{n-i}$ mit $e_j \in \{0, 1\}$.

Beispiel für Gruppierung: $y = 0010, z = 1001, r = 0$.

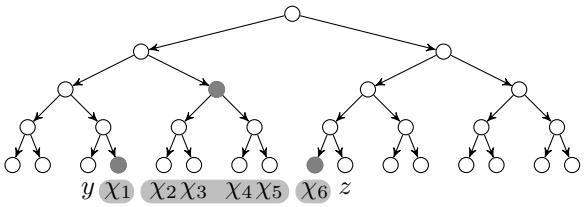
i	#	Muster
4	1	0011
3	–	wegen $y_3 = 1$
2	4	$01e_1e_2$
2	–	wegen $z_2 = 0$
3	–	wegen $z_3 = 0$
4	1	1000

Für jede Gruppe dieser Belegungen ist immer ein abgebrochener rekursiver Aufruf „zuständig“. Das heißt, die zweite **if**-Bedingung gibt falsch zurück und schließt damit alle Belegungen in dieser Gruppe aus.

Da die Vereinigung der Belegungen all dieser Gruppen $\chi_1, \ldots, \chi_k$ ergibt, müssen wir nur noch betrachten wie viele Gruppen wir haben und erhalten damit die Laufzeit des i-ten Delays. Die meisten Gruppen bekommen wir, wenn für (i) nie b) und für (ii) nie a) eintrifft. Im schlimmsten Fall haben wir also $2 \cdot (n - r - 2)$ Gruppen mit einer Laufzeit von je $O(p(|\phi|) + q(|\phi|))$. ■

■

Beispiel 7. Hier wird noch mal die Gruppierung im vollständigen Belegungsbaum direkt dargestellt für $y = 0010, z = 1001$. Die grauen Kreise markieren den Abbruch des rekursiven Aufrufs im Algorithmus.



◀

Literaturverzeichnis

[Aharoni u. Linial 1986] Aharoni, R. ; Linial, N.: Minimal non-two-colorable hypergraphs and minimal unsatisfiable formulas. In: *Journal of Combinatorial Theory* 43 (1986), Nr. 2, S. 196–204

[Beyersdorff u. a. 2010] Beyersdorff, O. ; Galesi, N. ; Lauria, M.: A lower bound for the pigeonhole principle in tree-like Resolution by asymmetric Prover–Delayer games. In: *Information Processing Letters* 110 (2010), Nr. 23, S. 1074–1077

[Bubeck u. Kleine Büning 2009] Bubeck, U. ; Kleine Büning, H.: A New 3-CNF Transformation by Parallel-serial Graphs. In: *Inf. Process. Lett.* 109 (2009), März, Nr. 7, S. 376–379

[Cook 1971] Cook, S. A.: The complexity of theorem proving procedures. In: *Proceedings 3rd Symposium on Theory of Computing*, ACM Press, 1971, S. 151–158

[Johannsen 2005] Johannsen, J.: *Algorithmen für das SAT-Problem, Skript zur Vorlesung.* 2005

[Levin 1973] Levin, L. A.: Universal sorting problems. In: *Problems of Information Transmission* 9 (1973), S. 265–266

[Pudlák 1998] Pudlák, P.: Satisfiabilityalgorithms and logic. In: *Mathematical Foundations of Computer Science 1998*, Springer, 1998, S. 129–141

[Saß 2013] Sass, B.: *SAT-Algorithmen.* Bachelor-Arbeit, Institut für Theoretische Informatik, Leibniz Universität Hannover, 2013

[Schöning 2009] Schöning, U.: *SAT Solving, Vorlesungsmitschrift von O. Gableske.* 2009

[Schöning u. Torán 2012] Schöning, U. ; Torán, J.: *Das Erfüllbarkeitsproblem SAT – Algorithmen und Analysen.* Lehmanns media, 2012

[Selman u. a. 1996] Selman, B. ; Kautz, H. ; Cohen, B.: Local search strategies for satisfiability testing. In: *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge* 26 (1996), S. 521–532

[Tovey 1984] Tovey, C. A.: A simplified NP-complete satisfiability problem. In: *Discrete Applied Mathematics* 8 (1984), S. 85–89