Algorithmen für das SAT-Problem

Skript zur Vorlesung im WS 02/03
gehalten von Dr. Jan Johannsen

Version vom 5. Februar 2003

1 Grundlagen

1.1 Syntax der Aussagenlogik

Formeln der Aussagenlogik sind zusammengesetzt aus Variablen, von denen es eine abzählbar unendliche Menge Var gibt, mittels der logischen Zeichen $\neg, \vee, \wedge$, sie sind induktiv definiert wie folgt :

1. Die Konstanten 0 und 1 sind Formeln.
2. Jede Variable ist eine Formel.
3. Ist F eine Formel, so ist auch $\neg F$ eine Formel.
4. Sind F und G Formeln, so sind auch $F \vee G$ sowie $F \wedge G$ Formeln.

Variablen bezeichnen wir im Allgemeinen mit x, y, z und Formeln mit F, G, H, beide evtl. mit Indizes. Die Menge der in F vorkommenden Variablen wird mit $V(F)$ bezeichnet und kann induktiv definiert werden durch:

$$V(x) = \{x\} \qquad V(F \wedge G) = V(F) \cup V(G)$$
$$V(\neg F) = V(F) \qquad V(F \vee G) = V(F) \cup V(G)$$

Die Größe $|F|$ einer Formel F ist die Anzahl der in F vorkommenden Zeichen, und kann induktiv definiert werden durch:

$$|x| = 1 \qquad |F \wedge G| = |F| + |G| + 1$$
$$|\neg F| = |F| + 1 \qquad |F \vee G| = |F| + |G| + 1$$

1.2 Semantik der Aussagenlogik

Eine *Bewertung* ist eine endliche partielle Abbildung α, die Variablen $x \in \mathrm{Var}$ einen Wahrheitswert $\alpha(x) \in \{0,1\}$ zuordnet. Wir beschreiben eine Bewertung durch eine Liste von Wertzuweisungen

$$[x_1 \leftarrow \epsilon_1, \ldots, x_k \leftarrow \epsilon_k]$$

für paarweise verschiedene Variable $x_1, \ldots, x_k$ und Werte $\epsilon_1, \ldots, \epsilon_k \in \{0,1\}$. Bewertungen bezeichnen wir im Allgemeinen mit α, β, γ, evtl. mit Indizes.

Der Definitionsbereich von α wird mit $\mathrm{dom}\,\alpha$ bezeichnet. Eine Bewertung α heißt *total* für F, falls sie jede in F vorkommende Variable bewertet, also $V(F) \subseteq \mathrm{dom}\,\alpha$ ist, andernfalls heißt sie *partiell*.

Der Wert $F\alpha$ einer Formel F unter der Bewertung α wird errechnet, indem jede Variable $x \in \mathrm{dom}\,\alpha$ in F durch die Konstante $\alpha(x)$ ersetzt werden, und dann gemäß der folgenden Regeln vereinfacht wird:

$$\begin{array}{lll} \neg 0 \rightsquigarrow 1 & F \wedge 0\,,\ 0 \wedge F \rightsquigarrow 0 & F \vee 0\,,\ 0 \vee F \rightsquigarrow F \\ \neg 1 \rightsquigarrow 0 & F \wedge 1\,,\ 1 \wedge F \rightsquigarrow F & F \vee 1\,,\ 1 \vee F \rightsquigarrow 1 \end{array}$$

Falls α nicht total für F ist, ist $F\alpha$ eine Formel mit $V(F\alpha) \subseteq V(F) \setminus \mathrm{dom}\,\alpha$, kann aber auch ein Wert $F\alpha \in \{0,1\}$ sein, z.B. $(x \vee y)[x \leftarrow 1] = 1$.

Eine Bewertung β *erweitert* α, in Zeichen $\beta \supseteq \alpha$, falls $\mathrm{dom}\,\alpha \subseteq \mathrm{dom}\,\beta$ ist, und für alle $x \in \mathrm{dom}\,\alpha$ gilt $\alpha(x) = \beta(x)$.

Falls für eine Formel F und Bewertung α gilt $F\alpha = 1$, so sagen wir auch "α erfüllt F" und schreiben dafür $\alpha \models F$.

Eine Formel F heißt

- *erfüllbar*, wenn es eine Bewertung α gibt mit $\alpha \models F$.
- *gültig* oder *Tautologie*, wenn $\alpha \models F$ für jede für F totale Bewertung α gilt.

Zwei Formeln F und G heißen *äquivalent*, in Zeichen $F \equiv G$, falls für jede Bewertung α, die für F und G total ist, gilt $F\alpha = G\alpha$. Dies läßt sich auch ausdrücken wie folgt: für jede totale Bewertung α gilt $\alpha \models F$ genau dann, wenn $\alpha \models G$.

Zwei Formeln F und G heißen erfüllbarkeits-äquivalent, falls gilt: F ist erfüllbar genau dann wenn G ist erfüllbar.

1.3 Normalformen

Ein *Literal* ist eine Variable x oder eine negierte Variable $\neg x$. Eine Variable x heißt auch *positives* Literal und eine negierte Variable $\neg x$ *negatives* Literal.

Die Literale x und $\neg x$ sind *komplementär* zueinander. Literale bezeichnen wir im Allgemeinen mit $a, b, c, \ldots$, evtl. mit Indizes.

Die Notation für Bewertungen wird wie folgt auf Literale erweitert: Ist a die Variable x, so bedeutet $[a \leftarrow \epsilon]$ dasselbe wie $[x \leftarrow \epsilon]$, und ist a die negierte Variable $\neg x$, so bedeutet $[a \leftarrow \epsilon]$ dassselbe wie $[x \leftarrow (1 - \epsilon)]$.

Negations-Normalform

Eine Formel F ist in *Negations-Normalform* (NNF), falls Negationszeichen in F nur vor Variablen vorkommen, also falls F der folgenden induktiven Definition genügt:

- Jedes Literal ist eine Formel in NNF.
- Sind F und G Formeln in NNF, so sind auch $F \vee G$ sowie $F \wedge G$ Formeln in NNF.

Für eine Formel F in NNF definieren wir die negierte Formel $\bar{F}$ in NNF durch:

- Ist $F = x$, so ist $\bar{F} = \neg x$.
- Ist $F = \neg x$, so ist $\bar{F} = x$.
- Ist $F = G_1 \vee G_2$, so ist $\bar{F} = \bar{G}_1 \wedge \bar{G}_2$.
- Ist $F = G_1 \wedge G_2$, so ist $\bar{F} = \bar{G}_1 \vee \bar{G}_2$.

Proposition 1. *Für jede Formel F in NNF ist $\bar{F}$ äquivalent zu $\neg F$.*

Beweis durch Induktion nach der obigen induktiven Defintion der NNF.

- Ist $F = x$, so ist $\bar{F} = \neg x = \neg x$.
- Ist $F = \neg x$, so ist $\bar{F} = x \equiv \neg\neg x = \neg F$.
- Ist $F = G \vee H$, so ist $\bar{F} = \bar{G} \wedge \bar{H}$. Dies ist nach Induktionshypothese äquivalent zu $\neg G \wedge \neg H$, und nach dem De Morgan'schen Gesetz ist dies äquivalent zu $\neg(G \vee H) = \neg F$.
- Ist $F = G \wedge H$, so ist $\bar{F} = \bar{G} \vee \bar{H} \equiv \neg G \vee \neg H$ nach Induktionshypothese und dies ist äquivalent zu $\neg(G \wedge H) = \neg F$.

Proposition 2. *Für jede Formel F gibt es eine äquivalente Formel $\tilde{F}$ in NNF.*

Beweis durch Induktion nach dem Formelaufbau:

- Ist $F = x$, so ist F bereits in NNF, also setze $\tilde{F} = x$.
- Ist $F = G \circ H$ für $\circ \in \{\wedge, \vee\}$, so gibt es nach Induktionshypothese $\tilde{G} \equiv G$ und $\tilde{H} \equiv H$ in NNF, und wir setzen $\tilde{F} = \tilde{G} \circ \tilde{H}$ und erhalten offensichtlich $\tilde{F} \equiv G \circ H = F$.
- Ist $F = \neg G$ so gibt es nach Induktionshypothese $\tilde{G} \equiv G$ in NNF. Definiere $\tilde{F} = \bar{\tilde{G}}$, nach Prop. 1 gilt dann $\tilde{F} \equiv \neg G = F$.

Disjunktive und Konjunktive Normalform

Definition. *Wir definieren folgende spezielle Klassen von Formeln:*

1. *Ein* Term *ist eine Konjunktion* $a_1 \wedge \ldots \wedge a_k$ *von Literalen.*
2. *Eine* Klausel *ist eine Disjunktion* $a_1 \vee \ldots \vee a_k$ *von Literalen.*
3. *Eine Formel in* disjunktiver Normalform *(DNF) ist eine Disjunktion* $T_1 \vee \ldots \vee T_m$ *von Termen.*
4. *Eine Formel in* konjunktiver Normalform *(KNF) ist eine Konjunktion* $C_1 \wedge \ldots \wedge C_m$ *von Klauseln.*

Man beachte, dass jede Formel in DNF oder KNF auch in NNF ist.

Satz 3. *Für jede Formel* F *gibt es äquivalente Formeln* $\widehat{F}$ *in DNF und* $\check{F}$ *in KNF.*

Beweis. Für eine Variable x seien die Literale $x^1 = x$ und $x^0 = \neg x$ definiert. Für $\epsilon = 0, 1$ gilt also, dass $\alpha \models x^\epsilon$ genau dann wenn $\alpha(x) = \epsilon$ ist.
Wir definieren nun die Formel $\widehat{F}$. Sei $V(F) = \{x_1, \ldots, x_n\}$. Für eine Bewertung $\alpha = [x_1 \leftarrow \epsilon_1, \ldots, x_n \leftarrow \epsilon_n]$ betrachte den Term

$$T(\alpha) = x_1^{\epsilon_1} \wedge \ldots \wedge x_n^{\epsilon_n} \ .$$

Für jede Bewertung β gilt dann: $\beta \models T(\alpha)$ genau dann, wenn $\beta \supseteq \alpha$.
Es seien nun $\alpha_1, \ldots, \alpha_m$ genau diejenigen Bewertungen der Variablen $x_1, \ldots, x_n$ mit $\alpha_j \models F$ für $j = 1, \ldots, m$. Wir definieren dann

$$\widehat{F} = T(\alpha_1) \vee \ldots \vee T(\alpha_m)$$

und müssen zeigen, dass $\widehat{F} \equiv F$ ist.
Sei also β eine für F totale Bewertung mit $\beta \models F$, dann gibt es ein $j = 1, \ldots, m$ so dass $\beta \supseteq \alpha_j$, also gilt $\beta \models T(\alpha_j)$, und somit $\beta \models \widehat{F}$.

Umgekehrt gelte $\beta \models \widehat{F}$, dann muss gelten $\beta \models T(\alpha_j)$ für ein $j = 1, \ldots, m$, und somit $\beta \supseteq \alpha_j$, also nach Definition $\beta \models F$.

Die Formel $\check{F}$ in KNF erhält man nun wie folgt: Bilde zunächst die zu $\neg F$ äquivalente Formel G in DNF, $G = \widehat{\neg F}$. Insbesondere ist G in NNF, also definieren wir $\check{F} = \bar{G}$, und nach Prop. 1 ist $\check{F} \equiv \neg G \equiv F$. □

Im folgenden werden vor allem Formeln in KNF für uns von Interesse sein. Für eine solche Formel F legen wir die folgenden Bezeichnungen fest:

n	die Anzahl der Variablen in F,
m	die Anzahl der Klauseln in F,
k	die maximale Anzahl von Literalen in einer Klausel in F.

Ferner identifizieren wir zur Vereinfachung der Notation Formeln in KNF mit Mengen von Klauseln. Eine Menge M von Klauseln stehe also für die Formel $F_M := \bigwedge_{C \in M} C$ und umgekehrt. Außerdem bedeute $F \setminus C$ für eine Klausel C dasselbe wie $F \setminus \{C\}$.

Definition. *Eine Formel $F = C_1 \wedge \ldots \wedge C_m$ in KNF ist in k-KNF für $k \in \mathbb{N}$, falls jede Klausel C_j höchstens k Literale enthält.*

Die k-KNF ist im Gegensatz zur vollen KNF keine echte Normalform, da nicht jede Formel in diese Form gebracht werden kann:

Proposition 4. *Für jedes k gibt es eine Formel F in $(k+1)$-KNF, für die es keine äquivalente Formel in k-KNF gibt.*

Beweis. Sei $F = x_1 \vee \ldots \vee x_{k+1}$. Angenommen, G sei eine zu F äquivalente Formel in k-KNF, also $G = C_1 \wedge \ldots \wedge C_m$. Die Klausel C_1 enthält höchstens k Variablen, also o.B.d.A. die Variable x_{k+1} nicht.

Wähle also eine Bewertung α so, dass $\alpha(C_1) = 0$ ist, aber $\alpha(x_{k+1}) = 1$. Dann gilt $\alpha \models F$, aber $\alpha \not\models G$, im Widerspruch zur Annahme. □

1.4 Das Problem SAT und seine Komplexität

Das Entscheidungsproblem SAT ist definiert wie folgt:

Gegeben:	Eine Formel F in KNF.
Frage:	Ist F erfüllbar?

Ohne Beweis zitieren wir das folgende zentrale Resultat:

Satz 5. *SAT ist* **NP***-vollständig.*

Im Gegensatz dazu ist es für Formeln in DNF sehr einfach zu entscheiden, ob sie erfüllbar sind. Eine solche Formel F ist nämlich genau dann erfüllbar, wenn es einen Term in F gibt, der keine komplementären Literale a und $\bar{a}$ enthält. Daraus erhält man aber keinen effizienten Algorithmus für SAT, da die Umwandlung von Formeln in DNF aufwändig sein kann.

Proposition 6. *Es gibt Formeln F_n in KNF mit $|F_n| = O(n)$, derart dass jede zu F_n äquivalente Formel G_n in DNF die Größe $|G_n| \geq n2^n$ hat.*

Beweis. Die Formeln F_n sind definiert durch

$$F_n = \bigwedge_{i=1}^{n} (x_{i,0} \vee x_{i,1}) \wedge (\neg x_{i,0} \vee \neg x_{i,1}) \; .$$

Eine Bewertung erfüllt die Formel F_n genau dann, wenn sie für jedes $i = 1, \ldots, n$ *genau eine* der Variablen $x_{i,0}$ und $x_{i,1}$ mit 1 bewertet.

Sei G_n eine zu F_n äquivalente Formel in DNF. Wir zeigen zunächst, dass jeder Term in G_n für jedes $i = 1, \ldots, n$ genau eine der Variablen $x_{i,0}$ oder $x_{i,1}$ positiv enthalten muss.

Ist nämlich T ein Term, der weder $x_{i,0}$ noch $x_{i,1}$ positiv enthält, dann wähle eine Bewertung $\alpha \models T$ mit $\alpha(x_{i,0}) = \alpha(x_{i,1}) = 0$. Ist andererseits T ein Term, der sowohl $x_{i,0}$ als auch $x_{i,1}$ positiv enthält, dann wähle eine Bewertung $\alpha \models T$ mit $\alpha(x_{i,0}) = \alpha(x_{i,1}) = 1$. In beiden Fällen gilt $\alpha \models G_n$, aber $\alpha \not\models F_n$.

Ferner zeigen wir, dass für jedes $\epsilon = (\epsilon_1, \ldots, \epsilon_n) \in \{0,1\}^n$ ein Term in G_n existiert, der gerade die Variablen $x_{1,\epsilon_1}, \ldots, x_{n,\epsilon_n}$ positiv enthält. Ist dies für ein $\epsilon \in \{0,1\}^n$ nicht der Fall, dann muss jeder Term in G_n eine der Variablen $x_{1,(1-\epsilon_1)}, \ldots, x_{n,(1-\epsilon_n)}$ positiv enthalten. Dann wähle eine Bewertung α mit $\alpha(x_{i,\epsilon_i}) = 1$ und $\alpha(x_{i,(1-\epsilon_i)}) = 0$ für alle $i = 1, \ldots, n$. Für diese gilt $\alpha \models F_n$, aber $\alpha \not\models G_n$.

Die Formel G_n enthält also mindestens 2^n Terme, von denen jeder mindestens n Literale hat, also ist $|G_n| \geq n2^n$. □

3-SAT

Das Entscheidungsproblem k-SAT ist der folgende Spezialfall von SAT:

Gegeben:	Eine Formel F in k-KNF.
Frage:	Ist F erfüllbar?

Die Probleme k-SAT für $k \geq 3$ sind ebenso schwierig wie das allgemeine Problem, wie die folgende Reduktion zeigt.

Proposition 7. *Für jede Formel* F *in KNF gibt es eine erfüllbarkeits-äquivalente Formel* $\check{F}$ *in 3-KNF . Weiter ist* $\check{F}$ *aus* F *in polynomieller Zeit konstruierbar.*

Beweis. Für jede Klausel $C = a_1 \vee \ldots \vee a_{k+1}$ wird eine Formel $\check{C}$ in 3-KNF wie folgt konstruiert: Seien $y_1, \ldots, y_k$ neue Variablen, dann ist $\check{C}$ die Formel

$$(a_1 \vee y_1) \wedge (\bar{y}_1 \vee a_2 \vee y_2) \wedge \ldots \wedge (\bar{y}_{k-1} \vee a_k \vee y_k) \wedge (\bar{y}_k \vee a_{k+1})$$

Nun gilt: Falls $\alpha \models C$, so gibt es ein $\alpha' \supseteq \alpha$ mit $\alpha' \models \check{C}$. Ist $\alpha(a_i) = 1$, so definiere $\alpha'(y_j) = 1$ für alle $j < i$ und $\alpha'(y_j) = 0$ für alle $j \geq i$, und man prüft leicht nach, dass dann $\alpha \models \check{C}$ gilt.

Andererseits: ist $\beta \models \check{C}$, so auch $\beta \models C$. Denn angenommen $\beta(a_i) = 0$ für alle $i = 1, \ldots, k+1$, dann muss $\beta(y_1) = 1$ sein, damit die erste Klausel in C' erfüllt ist. Dann muss aber auch $\beta(y_2) = 1$ sein, damit die zweite Klausel erfüllt ist.

Analog schließt man für alle i, dass $\beta(y_{i+1}) = 1$ sein muss, falls $\beta(y_i) = 1$ ist. Es muss aber $\beta(y_k) = 0$ sein, damit die letzte Klausel in C' erfüllt ist. Durch Induktion ergibt sich ein Widerspruch, also muss $\beta(a_i) = 1$ für mindestens ein $i = 1, \ldots, k+1$ sein, und somit $\beta \models C$.

Für eine Formel $F = C_1 \wedge \ldots \wedge C_m$ in KNF wird nun die Formel $\check{F}$ definiert als $\check{C}_1 \wedge \ldots \wedge \check{C}_m$. Nun läßt sich nach dem Obigen jede Bewertung $\alpha \models F$ zu einer Bewertung $\alpha' \models \check{F}$ erweitern, und jede Bewertung $\beta \models \check{F}$ erfüllt auch F, also sind F und $\check{F}$ erfüllbarkeits-äquivalent. □

Da die Konstruktion eine polynomielle Reduktion von SAT auf 3-SAT ist, folgt somit:

Korollar 8. *3-SAT ist* **NP***-vollständig.*

1.5 Einfache Spezialfälle

Wir betrachten zwei Spezialfälle von Formeln, für die es effiziente Algorithmen für das Erfüllbarkeitsproblem gibt.

2-SAT

Im Kontrast zu Korollar 8 ist das Problem 2-SAT einfach:

Satz 9. *2-SAT ist in linearer Zeit* $O(m)$ *entscheidbar.*

Beweis. Für jede Formel F in 2-KNF wird ein gerichteter Graph $G(F)$ definiert wie folgt:

- Die $2n$ Knoten von $G(F)$ sind die Literale x und $\neg x$ für jede Variable $x \in V(F)$.
- In $G(F)$ gibt es eine Kante $a \to b$, wenn die Klausel $\bar{a} \vee b$ in F vorhanden ist.
- Außerdem gibt es die Kanten $\bar{a} \to a$ für jede Einerklausel a.

Jede Zweierklausel $a \vee b$ entspricht also zwei Kanten $\bar{a} \to b$ und $\bar{b} \to a$.
Wir zeigen die folgende Behauptung:

> Ist b von a in $G(F)$ erreichbar, und ist $\alpha \models F$ mit $\alpha(a) = 1$, dann ist auch $\alpha(b) = 1$.

Es genügt, dies für den Fall zu zeigen, wo eine Kante $a \to b$ vorhanden ist, die allgemeine Behauptung folgt induktiv. Für jede soche Kante ist aber eine Klausel $\bar{a} \vee b$ in F vorhanden, die nicht erfüllt ist, wenn $\alpha(a) = 1$ und $\alpha(b) = 0$ ist.
Bezeichne nun $[a]$ für jedes Literal a die starke Zusammenhangskomponente von $G(F)$, in der der Knoten a liegt. Es gilt also: ist $\alpha \models F$, und $[a] = [b]$, so ist $\alpha(a) = \alpha(b)$. Daher ist F unerfüllbar, falls $[x] = [\bar{x}]$ für eine Variable x ist. Umgekehrt gilt: ist $[x] \neq [\bar{x}]$ für jede Variable x, dann ist F erfüllbar. Eine Bewertung $\alpha \models F$ wird wie folgt konstruiert:
Seien $[a_1], \ldots, [a_r]$ die starken Zusammenhangskomponenten in umgekehrter topologischer Ordnung (des induzierten gerichteten azyklischen Graphen).

```
for j := 1 to r do
    falls die Literale in [a_j] noch unbewertet sind
        α(b) := 1 für b ∈ [a_j]
        α(b) := 0 für b ∈ [ā_j]
```

Nach der Voraussetzung ist diese Bewertung wohldefiniert. Offenbar erfüllt α alle Klauseln, die Kanten innerhalb einer Komponente entsprechen.
Sei $a \to b$ also eine Kante zwischen zwei Komponenten, und nehmen wir $\alpha \not\models \bar{a} \vee b$ an. Dann müssen die Literale in $[a]$ mit 1 und die in $[b]$ mit 0 bewertet worden sein. Deshalb liegt in der topologischen Ordnung $[\bar{a}]$ vor $[a]$ und $[\bar{b}]$ hinter $[b]$, und daher auch $[\bar{b}]$ hinter $[\bar{a}]$. Dies ist ein Widerspruch, da $G(F)$ auch die Kante $\bar{b} \to \bar{a}$ enthält.
Da die starken Zusammenhangskomponenten eines Graphen in linearer Zeit bestimmt und topologisch angeordnet werden können, ist diese Konstruktion und der Test auf Erfüllbarkeit in linearer Zeit möglich. □

Der Algorithmus zeigt sogar, dass das Problem 2-SAT in der Komplexitätsklasse **NL** liegt, also von einem nichtdeterministischen Algorithmus mit logarithmischem Speicherplatzbedarf gelöst werden kann. Tatsächlich ist 2-SAT auch vollständig für **NL** (unter einem geeigneten schwachen Reduktionsbegriff).

Horn-Formeln

Definition. *Eine Klausel heißt*

- positiv *(*negativ*), wenn sie nur positive (negative) Literale enthält,*
- Einerklausel, *wenn sie aus genau einem Literal besteht,*
- Horn-Klausel, *wenn sie höchstens ein positives Literal enthält.*
- definit, *wenn sie genau ein positives Literal enthält.*

Jede Horn-Klausel ist also entweder definit oder negativ.
Eine Horn-Formel *ist eine Konjunktion von Horn-Klauseln, also eine Formel in KNF, in der jede Klausel eine Horn-Klausel ist.*

Horn-SAT ist der folgende Spezialfall von SAT:

Gegeben:	Eine Horn-Formel F.
Frage:	Ist F erfüllbar?

Satz 10. *Horn-SAT ist polynomialer Zeit* $O(nm)$ *entscheidbar.*

Beweis. Der folgende Algorithmus entscheidet, ob eine eingegebene Horn-Formel F erfüllbar ist, und gibt gegebenenfalls eine erfüllende Bewertung aus.

Horn-Sat(F)
  $\alpha := []$
  while $F\alpha$ enthält eine positive Einerklausel
    wähle eine solche x
    $\alpha := \alpha \cup [x \leftarrow 1]$
  if $\alpha' \models F$
    then return α'
    else return unerfüllbar

wobei α' definiert ist durch

$$\alpha'(x) = \begin{cases} 1 & \alpha(x) = 1 \\ 0 & x \notin \operatorname{dom} \alpha \end{cases} .$$

Wir zeigen zuerst, dass für jede definite Klausel $C = x_1 \vee \bar{x}_2 \vee \ldots \vee \bar{x}_k$ in F gilt $\alpha' \models C$. Ist $\alpha'(x_i) = 0$ für ein $i = 2, \ldots, k$, dann gilt $\alpha' \models C$. Andernfalls kommt zu dem Zeitpunkt, wenn alle $x_2, \ldots, x_k$ in $\operatorname{dom} \alpha$ sind, die Einerklausel x_1 in $F\alpha$ vor, und somit setzt der Algorithmus $\alpha(x_1) = 1$. Dann gilt aber auch $\alpha' \models C$.

Wir zeigen nun, jede Bewertung β, die alle definiten Klauseln in F erfüllt, eine Erweiterung $\beta(x) \supseteq \alpha$ ist. Andernfalls sei x die erste Variable, für die der Algorithmus $\alpha(x) = 1$ setzt, und für die $\beta(x) = 0$ ist. Zum Zeitpunkt, wo $\alpha(x) = 1$ gesetzt wird, kommt die Einerklausel x in $F\alpha$ vor, also gibt es eine definite Klausel $C = x \vee \bar{y}_1 \vee \ldots \vee \bar{y}_k$ in F, so dass die Variablen $y_1, \ldots, y_k$ bereits in $\operatorname{dom} \alpha$ sind. Nach Wahl von x ist $\beta(y_1) = \ldots = \beta(y_k) = 1$, also ist $\beta \not\models C$.

Schließlich zeigen wir, dass F unerfüllbar ist, falls $\alpha' \not\models F$, woraus sofort die Korrektheit des Algorithmus folgt. Falls $\alpha' \not\models F$, gibt es eine negative Klausel $C = \bar{y}_1 \vee \ldots \vee \bar{y}_k$ in F mit $\alpha' \not\models C$. Also sind $y_1, \ldots, y_k \in \operatorname{dom} \alpha$, und daher ist für jede Bewertung β, die alle definiten Klauseln in F erfüllt, $\beta(C) = 0$. Damit ist F unerfüllbar.

Offenbar ist die Zahl der Schleifendurchläufe durch n beschränkt, und in jedem Schleifendurchlauf wird nach einer positiven Einerklausel gesucht. Durch Verwendung geeigneter Datenstrukturen (s.u.) ist dies in Zeit $O(m)$ möglich, der gesamte Zeitaufwand ist also $O(nm)$. □

Insbesondere liegt also Horn-SAT in der Klasse **P**, und ist sogar vollständig für diese Klasse.

2 DLL-Algorithmen

Offensichtlich ist SAT trivialerweise in Zeit $O(2^n)$ durch Probieren aller 2^n möglichen Bewertungen lösbar. Dieses Ausprobieren läßt sich am besten durch ein Backtracking-Verfahren, in dem die vorkommenden Variablen sukzessive bewertet werden, organisieren. Solche Backtracking-Verfahren und ihre Verfeinerungen nennen wir DLL-Algorithmen, nach den Autoren Davis, Logemann und Loveland, die das erste Verfahren dieser Art 1962 beschrieben haben.

Die grundlegende Struktur eines DLL-Algorithmus kann wie folgt beschrieben werden. Der Aufruf DLL($F, []$) liefert entweder eine Bewertung zurück, die F erfüllt, oder NIL falls F unerfüllbar ist.

```
DLL(F, α)
  if Fα = 0
      then return NIL
  if Fα = 1
      then return α
  wähle x ∈ V(Fα)
  β := DLL(F, α ∪ [x ← 0])
  if β ≠ NIL
      then return β
      else  return DLL(F, α ∪ [x ← 1])
```

Die Backtracking-Strategie ist unter Umständen effizienter als ein einfaches brute-force durchprobieren aller 2^n möglichen Bewertungen: wenn für eine partielle Bewertung α bereits $C\alpha = 0$ ist für eine Klausel C in F, so ist auch $C\beta = 0$ für alle Erweiterungen $\beta \supseteq \alpha$. Diese Erweiterungen brauchen also nicht mehr untersucht werden, der Suchbaum wird an dieser Stelle gekappt und somit kleiner.

Da jeder rekursive Aufruf bei DLL nur polynomiale Zeit benötigt, ist die Laufzeit beschränkt durch $T(n) \cdot n^{O(1)}$, wobei $T(n)$ die Größe des Rekursionsbaumes bezeichnet. Der polynomiale Faktor wird bei der Angabe der Komplexität meist unterschlagen, man gibt als Komplexität nur die Baumgröße $T(n)$ an. Ferner wird die Größe des Baumes durch die Anzahl seiner Blätter abgeschätzt, da für einen ect verzweigenden Baum die Größe durch ein konstantes Vielfaches dieser Anzahl beschränkt ist.

Man erhält so bereits eine bessere Komplexitätsschranke für k-SAT, wenn bei DLL die folgende Strategie verfolgt wird: Wähle immer eine Klausel C, und verzweige nacheinander nach den höchstens k Variablen, die in C vorkommen.

Von den 2^k möglichen Bewertungen dieser Variablen ist immer eine α mit $C\alpha = 0$, also wird bei diesem α nicht weiter verzweigt. Der Algorithmus erzeugt also einen $(2^k - 1)$-verzweigenden Rekursionsbaum der Tiefe n/k, dieser hat $(2^k - 1)^{n/k}$ Blätter.

Die Laufzeit für k-SAT ist also beschränkt durch $O((2^k - 1)^{n/k}) = O({b_k}^n)$, wobei $b_k = \sqrt[k]{2^k - 1} < 2$ ist. Allerdings strebt b_k für große k recht schnell gegen 2, die Werte b_k für kleine k sind ungefähr:

k	3	4	5	6	7	8
b_k	1.91294	1.96799	1.98735	1.99477	1.99777	1.99903

2.1 Reduktionen

Einerklauseln

Eine besondere Rolle bei DLL-Algorithmen spielen Einerklauseln: wird nach einer Variablen x verzweigt, die in einer Einerklausel a vorkommt, so wird der rekursive Aufruf, der $[a \leftarrow 0]$ setzt, sofort im nächsten Schritt abbrechen. Es findet also keine echte Verzweigung statt, man kann dies so auffassen, dass einfach gleich $[a \leftarrow 1]$ gesetzt wird.

Da dies für die Effizienz günstig ist, werden die Einerklauseln in den meisten DLL-Algorithmen bevorzugt zuerst behandelt, dies wird als *unit propagation* bezeichnet.

UP(F, α)
  while in F gibt es eine Einerklausel a
    $F := F[a \leftarrow 1]$
    $\alpha := \alpha \cup [a \leftarrow 1]$

Dieser Schritt wird gesondert als Vereinfachungsschritt, eine sog. *Reduktion*, behandelt, der unmittelbar nach dem (rekursiven) Aufruf der Prozedur ausgeführt wird. Allgemein bezeichnet man als Reduktionen effizient (etwa in polynomieller Zeit) zu berechnende Abbildungen der Formeln in KNF in sich, die die Erfüllbarkeits-äquivalenz erhalten.

Neben der unit propagation gibt es weitere solche Reduktionen, darunter auch solche, die die Formel F auf andere Weise als durch Variablenbewertung modifizieren. Das allgemeine Schema eines DLL-Algorithmus läßt sich also treffender wie folgt beschreiben:

DLL(F, α)
  reduziere F
  if $F = 0$ then return NIL

if $F = 1$ then return α
wähle $x \in V(F)$ und $\epsilon \in \{0, 1\}$
$\beta := \mathrm{DLL}(F[x \leftarrow \epsilon], \alpha \cup [x \leftarrow \epsilon])$
if $\beta \neq \mathrm{NIL}$
    then return β
    else return $\mathrm{DLL}(F[x \leftarrow (1 - \epsilon)], \alpha \cup [x \leftarrow (1 - \epsilon)])$

Ein DLL-Algorithmus hängt also von zwei Parametern ab: der Regel, nach der die Verzweigungsvariable ausgewählt wird, und den verwendeten Reduktionen. Dabei werden Reduktionen, die die Formel verkleinern, meist solange iteriert, bis keine mehr anwendbar sind.

Reine Literale

Eine fast immer verwendete Reduktion ist die Elimination von reinen Literalen (engl. *pure literals*):

Definition. *Ein Literal* a *heißt* rein *in* F*, wenn* $\bar{a}$ *nicht in* F *vorkommt.*

Ist ein Literal a rein in F, so braucht die Bewertung $[a \leftarrow 0]$ nicht betrachtet werden: Ist α eine Bewertung, die F erfüllt, mit $\alpha(a) = 0$, so wird F auch von der Bewertung α' mit $\alpha'(a) = 1$ und $\alpha'(x) = \alpha(x)$ für $x \in \operatorname{dom} \alpha \setminus V(a)$ erfüllt. Man kann also reine Literale gleich mit 1 bewerten, was die folgende Reduktion rechtfertigt:

$\mathrm{PL}(F, \alpha)$
  while in F gibt es ein reines Literal a
    $F := F[a \leftarrow 1]$
    $\alpha := \alpha \cup [a \leftarrow 1]$

Subsumption

Eine Reduktion, die nicht durch Bewertung einer Variablen entsteht, ist die Subsumptionsregel.

Definition. *Eine Klausel* C subsumiert D*, in Zeichen* $C \leq D$*, wenn jedes Literal in* C *auch in* D *vorkommt.*

Offensichtlich erfüllt dann jede Bewertung $\alpha \models C$ auch D, also gilt die folgende Beobachtung.

Proposition 11. *Ist* F *eine Formel, in der Klauseln* C *und* D *mit* $C \leq D$ *vorkommen, so sind* F *und* $F \setminus D$ *erfüllbarkeits-äquivalent.*

Dies rechtfertigt die *Subsumptionsregel*, die aus einer Formel alle Klauseln enfernt, die von einer anderen subsumiert werden. Diese Regel gehörte zum ursprünglichen DLL-Algorithmus, wird aber heute wegen des hohen Aufwandes meist nicht mehr verwendet.

Resolution

Eine Reduktion, die die Formel vergrößert, ist die Resolutionsregel: Kommen in F zwei Klauseln vor, die komplementäre Literale enthalten, also etwa $a \vee C$ und $\bar{a} \vee D$, so kann die *Resolvente* $C \vee D$ zu F hinzugefügt werden.

Erfüllt α die beiden Klauseln $a \vee C$ und $\bar{a} \vee D$, so gilt: ist $\alpha(a) = 0$, dann ist $\alpha \models C$, und ist $\alpha(a) = 1$, dann ist $\alpha \models D$, also in jedem Fall $\alpha \models C \vee D$. Daher erhält die Hinzufügung der Resolvente die Erfüllbarkeit, ist also eine Reduktion.

Die Resolutionsregel rechtfertigt auch die folgenden verkürzenden Reduktionen, deren Korrektheit sofort aus der Korrektheit der Resolution und der obigen Reduktionen folgt.

Einerklausel nach Resolution Enthält eine Formel F zwei Klauseln $a \vee b$ und $\bar{a} \vee b$, so ersetze F durch $F[b \leftarrow 1]$.

Subsumption nach Resolution Enthält eine Formel F zwei Klauseln $a \vee C$ und $\bar{a} \vee C'$, sowie D mit $C \vee C' \leq D$, so ersetze F durch $F \setminus D$.

Resolution mit Subsumption einer Elternklausel Enthält eine Formel F zwei Klauseln $a \vee C$ und $\bar{a} \vee D$ mit $D \leq C$, so ersetze F durch $(F \setminus (a \vee C)) \wedge C$.

2.2 Algorithmus von Monien-Speckenmeyer

Eine Verzweigungsstrategie, die bei k-SAT die oben benutzte Eigenschaft hat, ist die folgende: wähle immer die erste Variable in einer kürzesten Klausel. Im nächsten Schritt ist dann in einem der Zweige diese Klausel nicht mehr vorhanden, im Anderen ist der Rest dieser Klausel eine kürzeste Klausel. Daher kann man den DLL-Algorithmus mit dieser Verzweigungsstrategie auch äquivalenterweise wie folgt schreiben:

simple-MS(F, α)
  if $F = 0$ then return NIL
  if $F = 1$ then return α

wähle kürzeste Klausel $C = a_1 \vee \ldots \vee a_r$ in F
for $i := 1$ to r
    $\gamma_i := [a_1 \leftarrow 0, \ldots, a_{i-1} \leftarrow 0, a_i \leftarrow 1]$
    $\beta :=$ simple-MS$(F\gamma_i, \alpha \cup \gamma_i)$
    if $\beta \neq$ NIL then return β
return NIL

Es wird also bei Auswahl einer kürzesten Klausel r-fach verzweigt, wobei im i-ten Zweig i Variablen bewertet sind, also höchstens $n - i$ Variablen übrig bleiben.

Da stets $r \leq k$ ist, erhält man folgende Rekursionsgleichung für die Anzahl $T(n)$ der Blätter im Verzweigungsbaum

$$T(n) = T(n-1) + \ldots + T(n-k)$$

und der Ansatz $T(n) = b^n$ liefert die Gleichung $b^k = b^{k-1} + \ldots + b + 1$ für die Basis $b = b_k$. Die Lösungen dieser Gleichung für kleine Werte von k sind ungefähr:

k	3	4	5	6	7	8
b_k	1.83929	1.92757	1.96595	1.98359	1.99197	1.99603

Autarkien

Eine Verbesserung des Algorithmus läßt sich durch Verwendung des Begriffs der Autarkie erreichen.

Definition. *Eine Bewertung α heißt* autark *für F, wenn jede Klausel in $F\alpha$ auch schon in F vorkommt.*

Anders ausgedrückt ist α autark für F, wenn jede Klausel in F, die eine Variable in $\operatorname{dom}\alpha$ enthält, schon von α erfüllt wird. Autarke Bewertungen haben die folgende Eigenschaft:

Proposition 12. *Ist α autark für F, dann ist F erfüllbar genau dann, wenn es $\beta \models F$ gibt mit $\beta \supseteq \alpha$.*

Beweis. Ist $F\alpha$ erfüllbar, so ist offensichtlich F erfüllbar. Umgekehrt sei $\beta \models F$, dann definiere $\beta' \supseteq \alpha$ durch

$$\beta'(x) = \begin{cases} \alpha(x) & x \in \operatorname{dom}\alpha \\ \beta(x) & x \in \operatorname{dom}\beta \setminus \operatorname{dom}\alpha \end{cases}$$

Jede Klausel in F, die durch α erfüllt wird, wird auch durch β' erfüllt. Die übrigen Klauseln in F, also diejenigen in $F\alpha$, enthalten die Variablen in

$\operatorname{dom}\alpha$ nicht, müssen also von β dadurch erfüllt werden, dass Literale aus Variablen in $\operatorname{dom}\beta \setminus \operatorname{dom}\alpha$ mit 1 bewertet werden. Diese Klauseln werden also ebenfallso durch β' erüllt, also gilt $\beta' \models F$. □

Die Verbesserung des Algorithmus besteht darin, dass nach Auswahl einer kürzesten Klausel erst überprüft wird, ob eine der möglichen erfüllenden Bewertungen dieser Klausel autark ist. Ist dies der Fall, wird nur der Zweig für diese Bewertung weiterverfolgt, andernfalls wird wiederum über alle möglichen Bewertungen verzweigt.

```
MS(F, α)
  if F = 0 then return NIL
  if F = 1 then return α
  wähle kürzeste Klausel C = a_1 ∨ ... ∨ a_r in F
  for i := 1 to r
      γ_i := [a_1 ← 0, ..., a_{i-1} ← 0, a_i ← 1]
      if γ_i autark für F then return MS(Fγ_i, α ∪ γ_i)
  for i := 1 to r
      β := MS(Fγ_i, α ∪ γ_i)
      if β ≠ NIL then return β
  return NIL
```

Ist nun keine der Bewertungen $\gamma_1, \ldots, \gamma_r$, autark, so muss es für jede dieser Bewertungen γ_i eine Klausel C' geben mit $C'\gamma_i \neq 1$, aber $V(C') \cap \operatorname{dom}\gamma_i \neq \emptyset$. Es wird also ein Literal in C' mit 0 bewertet, also hat die Klausel $C'\gamma_i$ höchstens $k-1$ Literale. Für jedes $i = 1, \ldots, r$ enthält also $F\gamma_i$ eine Klausel mit höchstens $k-1$ Literalen, daher wird im nächsten Schritt höchstens $(k-1)$-fach verzweigt.

Bezeichnen wir wiederum mit $T(n)$ die Größe des Verzweigungsbaumes für Formeln in n Variablen, und mit $T'(n)$ die Größe des Verzweigungsbaumes für Formeln in n Variablen, die mindestens eine Klausel der Breite höchstens $k-1$ enthalten. Dann gelten die folgenden Rekursionsgleichungen:

$$T(n) = \max(T(n-1), T'(n-1) + \ldots + T'(n-k))$$
$$T'(n) = \max(T(n-1), T'(n-1) + \ldots + T'(n-k+1))$$

Durch Induktion nach n zeigt man leicht, dass für alle n gilt

$$T(n-1) \leq T'(n-1) + \ldots + T'(n-k+1) \ ,$$

also vereinfacht sich die Rekursion zu

$$T(n) = T'(n-1) + \ldots + T'(n-k)$$
$$T'(n) = T'(n-1) + \ldots + T'(n-k+1)$$

und der Ansatz $T'(n) = b^n$ ergibt die Gleichung $b^{k-1} = b^{k-2} + \ldots + b + 1$ für die Basis $b = b_k$. Die erste Gleichung liefert auch $T(n) = O({b_k}^n)$, somit ist die Komplexität des Algorithmus im wesentlichen $O({b_k}^n)$, wobei die Werte b_k für kleine Werte von k die folgenden sind:

k	3	4	5	6	7	8
b_k	1.61804	1.83929	1.92757	1.96595	1.98359	1.99197

Eine Verfeinerung dieser Methode für 3-SAT wurde von Schiermeyer angegeben, die eine Komplexität von b^n mit $b = 1.5782$ erreicht.

2.3 Analysemethode von Kullmann

Für einen Vektor $(d_1, \ldots, d_m)$ mit $m \geq 1$ und $d_i > 0$ für $i = 1, \ldots, m$ sei $\tau(d_1, \ldots, d_m)$ definiert als die eindeutig bestimmte positive Lösung der Gleichung

$$x^{-d_1} + \ldots + x^{-d_m} = 1 \ .$$

Sei T ein Baum, und d eine Kantenbewertung: jedem Knoten v mit Sohn w wird eine Distanz $d(v,w) > 0$ zugeordnet. Für einen Knoten v mit den Söhnen $w_1, \ldots, w_m$ sei $d(v) = (d_1, \ldots, d_m)$ mit $d_i := d(v, w_i)$für $i = 1, \ldots, m$. Dann ist die *Verzweigungszahl* $\tau(v)$ von v definiert als $\tau(d(v))$, und $\tau(T) := \max(\tau(v))$ über alle inneren Knoten v in T.

Für einen Pfad $p = v_1, \ldots, v_m$ in T sei ferner

$$d(p) = \sum_{i=1}^{m-1} d(v_i, v_{i+1})$$

und $d(T) := \max(d(p))$ über alle Pfade in T von der Wurzel zu einem Blatt. Dann gilt die folgende obere Schranke an die Größe des Baumes:

Satz 13. *Die Anzahl der Blätter in* T *ist höchstens* $\tau(T)^{d(T)}$.

Beweis. Für jeden inneren Knoten gilt $\tau(v)^{d_1} + \ldots + \tau(v)^{d_m} = 1$, außerdem ist $\tau(v) \geq 1$, und und $\tau(v) = 1$ genau dann, wenn $m = 1$ ist, und somit ist $0 < \tau(v)^{-d_i} \leq 1$ für alle i. Daher bilden die Werte $p_i := \tau(v)^{-d_i}$ eine Wahrscheinlichkeitsverteilung auf den Kanten, die wir so interpretieren, dass von v mit Wahrscheinlichkeit p_i zum Sohn w_i übergegangen wird.

Dadurch wird ein zufälliger Pfad in T von der Wurzel zu einem Blatt erzeugt. Sei w ein Blatt, und $p = v_1, \ldots, v_m = w$ der eindeutig bestimmte Pfad von der Wurzel zu w. Die Wahrscheinlichkeit, dass der zufällige Pfad

w erreicht, ist genau die Wahrscheinlichkeit, dass er gerade p ist. Diese Wahrscheinlichkeit ist beschränkt durch

$$\prod_{i=1}^{m-1} \tau(v_i)^{-d(v_i,v_{i+1})} \geq \prod_{i=1}^{m-1} \tau(T)^{-d(v_i,v_{i+1})}$$

$$\geq \tau(T)^{-\sum_{i=1}^{m-1} d(v_i,v_{i+1})} = \tau(T)^{-d(p)} \geq \tau(T)^{-d(T)}$$

also wird jedes Blatt mit Wahrscheinlichkeit $\tau(T)^{-d(T)}$ erreicht. Daher kann die Anzahl der Blätter höchstens $\tau(T)^{d(T)}$ sein. □

Um eine möglichst gute Analyse der Laufzeit eines DLL-Verfahrens in Abhängigkeit von n zu erhalten, gilt es also, eine Distanzfunktion d für den Rekursionsbaum T zu finden, derart dass $d(T) \leq n$ ist, und $\tau(T)$ möglichst klein wird.

Um zu sehen, wie diese Methode die bisher verwendete verallgemeinert, präsentieren wir hier noch einmal die Analyse von simple-MS für 3-SAT in diesem Rahmen:

Definiere die Distanz $d(v, w)$ als die Anzahl der Variablen, die beim Übergang von v nach w bewertet werden. Auf jedem Pfad können höchstens alle Variablen bewertet werden, also ist für jeden Pfad $d(p) \leq n$, also $d(T) \leq n$.

Jeder innere Knoten im Rekursionsbaum T hat drei Söhne, je einen mit den Distanzen 3, 2 und 1. Also ist $\tau(v) = \tau(3, 2, 1)$ die Lösung der Gleichung

$$x^{-3} + x^{-2} + x^{-1} = 1$$

und da wir nur an positiven Lösungen interessiert sind, können wir diese mit x^3 multiplizieren. Wir erhalten dieselbe Gleichung

$$x^3 = x^2 + x + 1$$

die wir bei der obigen Analyse von gelöst haben, also ist $\tau(T)$ ungefähr 1.83929, und wir erhalten das gleiche Ergebnis wie oben.

Zur Analyse des Algorithmus von Monien-Speckenmeyer für 3-SAT wird dieselbe Distanzfunktion wie oben benutzt. Hier gibt es drei verschiedene Typen von inneren Knoten:

- Hat ein Knoten v genau einen Sohn (im Fall, dass eine autarke Bewertung gefunden wurde), so ist $\tau(v) = \tau(1) = 1$.
- Für Knoten v mit zwei Söhnen hat einer die Distanz 2, der andere Distanz 1, also ist $\tau(v) = \tau(2, 1)$ die positive Lösung von

  $$x^{-2} + x^{-1} = 1 \quad \text{bzw.} \quad x^2 = x + 1\,,$$

  also der goldene Schnitt $\phi \approx 1.61804$.

- Daneben gibt es Knoten v mit drei Söhnen, je einen mit den Distanzen 3, 2 und 1. Für diese ist wie oben $\tau(v) = \tau(3,2,1)$.

Es scheint also gegenüber dem vereinfachten Algorithmus zunächst nichts gewonnen zu sein.

Es gilt jedoch, dass ein Knoten v — mit Ausnahme der Wurzel — nur dann drei Söhne haben kann, wenn er keine 2-Klausel enthält, also durch eine autarke Bewertung erzeugt wurde. Dann ist v der einzige Sohn seines Vaters v', und hat von diesem mindestens die Distanz 1.

Um dies bei der Analyse auszunutzen, können v und v' zu einem neuen Knoten v^* verschmolzen werden. Dieser hat nun drei Söhne, jeweils mit den Distanzen 4, 3 und 2, also ist $\tau(v^*) = \tau(4,3,2)$ bestimmt durch Gleichung

$$x^{-4} + x^{-3} + x^{-2} = 1 \quad \text{bzw.} \quad x^4 = x^2 + x + 1$$

deren positive Lösung ist ungefähr $\tau(4,3,2) < 1.46558 < \tau(2,1)$. Damit ist $\tau(T) = \tau(2,1) < 1.61804$, und wir erhalten also obere Schranke für die Größe des Baumes $O(1.61804^n)$.

Eine alternative Analyse, bei der die Struktur des Baumes erhalten bleibt, besteht darin, die Distanz zwischen v und v' um einen Wert $\epsilon < 1$ zu vermindern, und die Distanzen zwischen v und seinen drei Söhnen um den gleichen Betrag zu erhöhen. Dadurch ändern sich die Pfadlängen $d(p)$ im Baum nicht, und für v erhalten wir die neue Verzweigungszahl $\tau(v) = \tau(3+\epsilon, 2+\epsilon, 1+\epsilon)$. Für $\epsilon = 1/2$ ist dies $\tau(7/2, 5/2, 3/2) < 1.59704 < \tau(2,1)$, also erhalten wir wiederum $\tau(T) = \tau(2,1)$.

Offensichtlich spielt die Reihenfolge der Argumente für den Wert der τ-Funktion keine Rolle. Weiterhin hat sie die folgenden Eigenschaften, von denen wir einige oben schon ausgenutzt haben:

Proposition 14. *1. Ist* $d_1' > d_1$, *so ist* $\tau(d_1', \ldots, d_m) < \tau(d_1, \ldots d_m)$.

2. Ist $d_1' + d_2' = d_1 + d_2$, *und* $\min(d_1', d_2') \geq \min(d_1, d_2)$, *dann ist*

$$\tau(d_1', d_2', d_3, \ldots, d_m) \leq \tau(d_1, d_2, d_3, \ldots, d_m)$$

wobei Gleichheit nur dann gilt, wenn sie in der Voraussetzung gilt.

3. Sei $d = (d_1, \ldots, d_m)$ *und* $d' := (d_1', \ldots, d_n')$, *und sei* $d^* := (d_1 + d_1', \ldots, d_1 + d_n', d_2, \ldots, d_m)$. *Dann gilt:*

- *Ist* $\tau(d) \leq \tau(d')$, *so ist* $\tau(d) \leq \tau(d^*) \leq \tau(d')$.
- *Ist* $\tau(d) \geq \tau(d')$, *so ist* $\tau(d) \geq \tau(d^*) \geq \tau(d')$.

Dabei sind beide Ungleichungen echt, falls die Ungleichung in der Voraussetzung echt ist.

Anschaulich haben diese Eigenschaften die folgende Bedeutung, die man bei Entwurf und Analyse von DLL-Verfahren ausnutzen kann:

1. Vergrößern der Distanz zu einem Sohn verbessert die Verzweigungszahl eines Knotens. Beispielsweise ist $\tau(3,1) < 1.46558 < \tau(2,1)$.

2. Von zwei Verzweigungen mit gleicher Summe der Distanzen liefert die symmetrischere die bessere Verzweigungszahl. Beispielsweise ist $\tau(2,2) = \sqrt{2} < \tau(3,1)$.

3. Werden zwei Knoten zu einem verschmolzen, so liegt die Verzweigungszahl des neuen Knotens zwischen denen der alten Knoten. Beispielsweise ist $\tau(4,3,2) = \tau(3,1+3,1+1) = \tau(3,1)$, und

$$\tau(3,1) < \tau(3,3,2) = \tau(3,1+2,1+1) < 1.52138 < \tau(2,1)\ .$$

2.4 Der Algorithmus von Zhang

Lemma 15. *Falls* a *und* b *in* F *nicht vorkommen, dann ist* $F \wedge (a \vee b \vee c)$ *genau dann erfüllbar, wenn eine der folgenden Formeln erfüllbar ist.*

$$F[c \leftarrow 1, a \leftarrow 0, b \leftarrow 0]$$
$$F[c \leftarrow 0, a \leftarrow 1, b \leftarrow 0]$$
$$F[c \leftarrow 0, a \leftarrow 0, b \leftarrow 1]$$

Dabei können die komplementären Literale $\bar{a}$ und $\bar{b}$ durchaus in F vorkommen.

Beweis. Da a und b in F nicht vorkommen, kann jedes $\alpha \models F$ so abgeändert werden, dass a oder b mit 0 bewertet werden.
Ist $\alpha(c) = 1$, so gilt $\alpha \models (a \vee b \vee c)$, und daher können a und b mit 0 bewertet werden. Ist andererseits $\alpha(c) = 1$, so muss eines von a und b mit 1 bewertet werden, es genügt aber auch eines. □

Ist F eine Formel, a ein Literal in F und C eine Klausel, dann bezeichnet $F\langle a := C\rangle$ die Formel, die aus F ensteht, indem jedes Vorkommen von a durch die Klausel C ersetzt wird. Die Vorkommen von $\bar{a}$ bleiben dabei unangetastet.

Lemma 16. *Falls* a *in* F *nicht vorkommt, dann sind* $F \wedge (a \vee C)$ *und* $F\langle \bar{a} := C\rangle$ *erfüllbarkeits-äquivalent.*

Beweis. Sei $\alpha \models F \wedge (a \vee C)$. Ist $\alpha(a) = 0$, so muss $\alpha(\bar{a}) = \alpha(C) = 1$ sein, also ist $F\langle \bar{a} := C\rangle\alpha = F\alpha = 1$.

Ist andererseits $\alpha(a) = 1$, so ist ist in jeder Klausel D in F, die $\bar{a}$ enthält, ein anderes Literal erfüllt, also ist $D\langle \bar{a} := C\rangle\alpha = D\alpha = 1$, und somit $F\langle \bar{a} := C\rangle\alpha = 1$. Also gilt in jedem Fall $\alpha \models F\langle \bar{a} := C\rangle$.

Sei umgekehrt $\alpha \models F\langle \bar{a} := C\rangle$, und o.E. a von α unbewertet. Ist $\alpha \models C$, dann setze $\alpha' := \alpha \cup [a \leftarrow 0]$, dann folgt $\alpha \models F \wedge (a \vee C)$.

Ist andererseits $C\alpha = 0$, dann muss in jeder Klausel D in $F\langle \bar{a} := C\rangle$ ein Literal, dass nicht in C ist, erfüllt sein, also gilt auch $\alpha \models F$. Also gilt $\alpha' \models F \wedge (a \vee C)$, wobei $\alpha' := \alpha \cup [a \leftarrow 1]$. □

Korollar 17. *Falls a in F nur in der Klausel $(a \vee b \vee C)$ vorkommt, dann ist F genau dann erfüllbar, wenn eine der folgenden Formeln erfüllbar ist.*

$$F[b \leftarrow 1, a \leftarrow 0]$$
$$F\langle \bar{a} := C\rangle[b \leftarrow 0, a \leftarrow 1]$$

Beweis. In $F[b \leftarrow 1]$ kommt a nicht mehr vor, also ist es erfüllbarkeits-äquivalent zu $F[b \leftarrow 1, a \leftarrow 0]$.

In $F[b \leftarrow 0]$ kommt a nur in der Klausel $a \vee c$ vor also ist es nach Lemma 16 erfüllbarkeits-äquivalent zu $F\langle \bar{a} := c\rangle[b \leftarrow 0]$. Da aber nun $\bar{a}$ nicht mehr vorkommt, ist diese erfüllbarkeits-äquivalent zu $F\langle \bar{a} := c\rangle[b \leftarrow 0, a \leftarrow 1]$. □

Für eine Formel F in KNF definieren wir die folgenden Mengen von Klauseln:

U	die Menge der Einerklauseln in F
D	eine maximale Menge von 2-Klauseln aus F, derart dass jede Variable höchstens einmal in $U \cup D$ vorkommt
T	die Menge der 2-Klauseln in F, die nicht in D sind.

und damit die Werte $u := |U|$, $d := |D|$ und $t := \min(|T|, 2)$.

Der Algorithmus von Zhang nimmt o.E. an, dass die Eingabeklausel F mindestens eine Einerklausel oder 2-Klausel enthält. Eine der Ideen des Algorithmus ist, sicherzustellen dass diese Eigenschaft erhalten bleibt.

Daher wird auf Reduktionen, die unkontrolliert kurze Klauseln eliminieren (wie z.B. das Beseitigen reiner Literale), verzichtet. Die einzigen verwendeten Reduktionen sind:

- Teste, ob komplementäre Einerklauseln a und $\bar{a}$ vorhanden sind, in diesem Fall gib NIL zurück.

- Lösche subsumierte 3-Klauseln.
- Falls $u \geq 2$ oder $u = 1$ und $d > 0$ ist, wähle eine Einerklausel a und setze $[a \leftarrow 1]$.

Die andere wesentliche Idee ist, neben autarken Bewertungen, durch die keine neuen Klauseln entstehen, auch solche Bewertungen gesondert zu behandeln, bei denen nur eine Variable bewertet wird so dass nur eine neue Klausel entsteht.

Definition. *Eine Bewertung* α *mit* $|\operatorname{dom}\alpha| = 1$ *heißt* quasi-autark *für* F, *wenn* $|F\alpha \setminus F| = 1$ *ist.*

Der Algorithmus von Zhang wird beschrieben durch die folgende rekursive Prozedur.

```
Zh(F, α)
  (F, α) := reduziere(F, α)
  if F = 0 then return NIL
  if F = 1 then return α
  if u = 1 then return unit(F, α)
  (γ1, γ2) := verzweige(F, α)
  if γi autark für F für ein i = 1, 2
       then return aut(F, γi, α)
  if γi quasi-autark für F für ein i = 1, 2
       then return qu-aut(F, γi, α)
  β := Zh(Fγ1, α ∪ γ1)
  if β ≠ NIL
       then return β
       else return Zh(Fγ2, α ∪ γ2)
```

Nach Reduktion ist entweder $u = 0$, oder es ist $u = 1$ und $d = 0$, d.h. es ist noch eine Einerklausel a vorhanden, und jede 2-Klausel enthält entweder a oder $\bar{a}$. In diesem Fall wird wie folgt vorgegangen:

```
unit(F, α)
  wähle eine 3-Klausel (b ∨ c ∨ d)
  γ := [a ← 1, b ← 0, c ← 0, d ← 1]
  if γ autark für F
       then return aut(F, γ, α)
  β := Zh(F[a ← 1] ∧ (b ∨ c), α ∪ [a ← 1])
  if β ≠ NIL
       then return β
       else return Zh(Fγ, α ∪ γ)
```

Sind keine Einerklauseln mehr vorhanden ($u = 0$), werden Bewertungen γ_1, γ_2 gemäss der folgenden Fallunterscheidung berechnet und nach diesen verzweigt.

Fall 1: $t \geq 1$, d.h. es gibt 2-Klauseln mit gemeinsamen Variablen.

Fall 1.1: es gibt 2-Klauseln $(a \vee b)$ und $(\bar{a} \vee c)$.

$\gamma_1 := [a \leftarrow 1, c \leftarrow 1]$

$\gamma_2 := [a \leftarrow 0, b \leftarrow 1]$

Fall 1.2: andernfalls wähle 2-Klauseln $(a \vee b)$ und $(a \vee c)$.

$\gamma_1 := [a \leftarrow 1]$

$\gamma_2 := [a \leftarrow 0, b \leftarrow 1, c \leftarrow 1]$

Fall 2: $t = 0$, d.h. alle 2-Klauseln sind variablen-disjunkt.

$\gamma_1 := [a \leftarrow 1]$

$\gamma_2 := [a \leftarrow 0, b \leftarrow 1]$

Beim Auftreten einer autarken Bewertung γ wird nicht verzweigt, sondern nur γ weiter verfolgt, dabei wird aber dafür gesorgt, dass eine Einerklausel in der Formel entsteht.

aut(F, γ, α)
  sei $\gamma = \gamma' \cup [a \leftarrow 1]$
  return $Zh(F\gamma' \wedge a, \alpha \cup \gamma')$

Beim Auftreten einer quasi-autarken Bewertung wird die Verzweigung verworfen und wie folgt eine neue berechnet. Ist $[a \leftarrow 1]$ quasi-autark, so kommt $\bar{a}$ in F nur in einer Klausel C vor. Diese kann nur eine 3-Klausel $C = (\bar{a} \vee b \vee c)$ sein, da quasi-autarkie nur im Fall 1.2 oder 2 aufgerufen wird. Wäre eine 2-Klausel $(\bar{a} \vee b)$ in F vorhanden, so wäre Fall 1.1 eingetreten.

Nach Korollar 17 ist also F genau dann erfüllbar, wenn eine der Formeln $F[b \leftarrow 1, a \leftarrow 1]$ oder $F\langle a := c\rangle[b \leftarrow 0, a \leftarrow 0]$ erfüllbar ist. Es wird dann unterschieden, on eine dieser Formeln keine neuen Klauseln enthält.

Enthält $F\langle a := c\rangle[b \leftarrow 0, a \leftarrow 0]$ keine neuen Klauseln, so kommt b in F nur in der Klausel C vor. Also ist nach Lemma 15 die in der folgenden Prozedur gewählte Verzweigung korrekt.

qu-aut(F, γ, α)
  sei $\gamma = [a \leftarrow 1]$
  sei $F\gamma \setminus F = \{(b \vee c)\}$
  if $[b \leftarrow 1, a \leftarrow 1]$ ist autark für F
    then return aut$(F, [b \leftarrow 1, a \leftarrow 1], \alpha)$
  if $F\langle a := c\rangle[b \leftarrow 0, a \leftarrow 0] \subseteq F$

```
        then γ1 := [c ← 1, a ← 1, b ← 0]
             γ2 := [c ← 0, a ← 0, b ← 0]                          (A)
             γ3 := [c ← 0, a ← 1, b ← 1]
             if es gibt 2-Klausel (b̄ ∨ d) mit d ∉ {a, ā, c, c̄}
                  then γ3 := γ3 ∪ [d ← 1]                         (B)
             if γi autark für F für ein i = 1, ..., 3
                  then return aut(F, γi, α)
             for i := 1 to 3
                  β := Zh(Fγi, α ∪ γi)
                  if β ≠ NIL then return β
             return NIL
    β := Zh(F[b ← 1, a ← 1], α ∪ [b ← 1, a ← 1])                  (C)
    if β ≠ NIL
         then return β
         else return Zh(F⟨a := c⟩[b ← 0, a ← 0], α ∪ [b ← 0, a ← 0])
```

Verwendet man zur Analyse des Algorithmus als Distanzfunktion wie oben die Anzahl der bewerteten Variablen, so erhält man keine bessere Schranke als die für den Algorithmus von Monien-Speckenmeyer, da im Fall 2 bei der Verzweigung Knoten auftreten, die zwei Söhne mit Distanzen 2 und 1 haben. Wir verwenden daher eine feinere Distanzfunktion, die die Anzahl der kurzen Klauseln mit berücksichtigt.

Sei ϵ ein Parameter, dessen Wert wir später festlegen werden. Als Maß für die Komplexität einer Formel F definieren wir

$$\mu := n - \epsilon(u + d + t) \; .$$

Ist v ein Knoten im Rekursionsbaum, und F die bei v betrachtete Formel mit Maß μ, sowie w ein Sohn von v und F' die bei w betrachtete Formel mit Maß μ', so definieren wir

$$d(v, w) = \mu - \mu'$$

Die Distanz zweier Knoten ist also die Anzahl der bewerteten Variablen plus der (gewichteten) Differenz der Werte u, d und t, die Auskunft über die kurzen Klauseln in der Formel geben.

An den folgenden Stellen im Algorithmus treten Verzweigungen auf:

- Prozedur unit()

  Im ersten Zweig verschwindet eine Einerklausel, alle evtl. vorhandenen 2-Klauseln werden zu Einerklauseln, und es wird eine 2-Klausel hinzugefügt. Daher bleibt $(u + d + t)$ konstant, die Distanz ist 1.

Im zweiten Zweig verschwindet eine Einerklausel, schlimmstenfalls alle 2-Klauseln, aber es wird eine neue 2-Klausel erzeugt, da γ nicht autark ist wenn echt verzweigt wird. Die Distanz ist also mindestens $4 - 2\epsilon$.

- Fall 1.1

  Im schlimmsten Fall verschwinden in beiden Zweigen zwei Klauseln aus D und alle aus T, da aber (im Fall einer echten Verzweigung) keine Autarkie vorliegt, ensteht mindestens eine neue kurze Klausel. Daher nimmt $(u + d + t)$ höchstens um 3 ab, die Distanz ist also mindestens $2 - 3\epsilon$.

- Fall 1.2

  Im ersten Zweig verschwindet nur eine Klausel aus D, aber möglicherweise alle aus T. Da aber γ_1 nicht quasi-autark ist, kommen auch zwei neue kurze Klauseln hinzu. Die Distanz ist also mindestens $1 - \epsilon$.

  Im zweiten Zweig verschwinden höchstens zwei Klauseln aus D und möglicherweise alle aus T, aber es kommt eine kurze Klausel hinzu, da γ_2 nicht autark ist. Somit ist die Distanz mindetens $3 - 3\epsilon$.

- Fall 2

  In jedem der Zweige verschwindet nur eine Klausel aus D. Im ersten Zweig kommen zwei neue kurze Klauseln, da γ_1 nicht quasi-autark ist, also ist die Distanz mindestens $1 + \epsilon$ Im zweiten kommt mindestens eine kurze Klausel hinzu, also ist die Distanz mindestens 2.

- Prozedur qu-aut(), Fall (A)

  Falls $\bar{b}$ nicht, oder nur gemeinsam mit $a, \bar{a}, c, \bar{c}$ in 2-Klauseln vorkommt, so verschwinden höchstens zwei Klauseln aus D und schlimmstenfalls alle aus T, es kommt aber eine neue 2-Klausel hinzu. Daher ist die Distanz in jedem Zweig mindestens $3 - 3\epsilon$.

- Prozedur qu-aut(), Fall (B)

  Kommt dagegen $\bar{b}$ gemeinsam mit einem anderen Literal d in einer 2-Klausel vor, so können drei Klauseln aus D verschwinden. Daher wird in einem Zweig eine vierte Variable eliminiert, so dass die Distanz in zwei Zweigen mindestens $3 - 4\epsilon$ und im dritten mindestens $4 - 4\epsilon$ beträgt.

- Prozedur qu-aut(), Fall (C)

  Dieser Fall ist analog zum Fall 1.1.

Der Wert $\tau(T)$ ist dementsprechend das Maximum aus den folgenden Werten, die Knoten entsprechen, wo gemäß der genannten Fälle verzweigt wird.

$\tau(1, 4-2\epsilon)$	Prozedur unit()
$\tau(2-3\epsilon, 2-3\epsilon)$	Fall 1.1 und Prozedur qu-aut(), Fall (C)
$\tau(1-\epsilon, 3-3\epsilon)$	Fall 1.2
$\tau(1+\epsilon, 2)$	Fall 2
$\tau(3-3\epsilon, 3-3\epsilon, 3-3\epsilon)$	Prozedur qu-aut(), Fall (A)
$\tau(3-4\epsilon, 3-4\epsilon, 4-4\epsilon)$	Prozedur qu-aut(), Fall (B)

Der Wert ϵ, bei dem das Maximum dieser Werte minimal wird, liegt ungefähr bei $\epsilon = 0,1528477$, und für diesen ist $\tau(T) = \tau(1+\epsilon, 2) < 1.570214$. Daher ist die Größe des Rekursionsbaumes, und somit die Komplexität des Algorithmus ungefähr $O(1.570214^n)$.

3 Effiziente Implementierung von DLL

3.1 Datenstrukturen

Grundsätzlich wird unterschieden zwischen destruktiven und nicht-destruktiven Datenstrukturen. Bei den ersteren werden die Änderungen der Formel bei Bewertung einer Variablen (Löschen erfüllter Klauseln und mit 0 bewerteter Literale) tatsächlich ausgeführt, bei den letzteren werden diese nur durch Markierungen angezeigt.

Bei nicht-destruktiven Datenstrukturen kann daher eine Bewertung rückgängig gemacht werden, was ein explizites Backtracking, und damit eine iterative Implementierung (s.u.) ermöglicht. Bei destruktiven Datenstrukturen ist dagegen backtracking nur durch Rekursion möglich, wobei der vorherige Zustand auf dem Stack abgelegt wird.

Die meisten Implementierungen verwenden eine Datenstruktur zur internen Darstellung von Formeln in KNF, die aus verzeigerten Listen aufgebaut ist. Eine solche nicht-destruktive Datenstruktur wird unten exemplarisch dargestellt.

Eine grundlegend andere Datenstruktur, die sogenannten *tries*, wird z.B. beim SAT-Solver SATO (H. Zhang, University of Iowa) verwendet. Sie ist ähnlich der Baumstruktur zur Speicherung von Wörterbüchern.

Verzeigerte Listen

Die Struktur zur Darstellung einer Klausel hat die Einträge:

- zwei Zeiger `pos-lit` und `neg-lit` auf Listen von Variablen,
- zwei Zähler `act-pos` und `act-neg`,
- ein Zeiger `sat` auf eine Variable.

Die Liste `pos-lit` enthält Zeiger auf die Variablen, die in der Klausel positiv vorkommen, und `neg-lit` entsprechend diejenigen, die negativ vorkommen.

Der Zähler `act-pos` gibt an, wieviele der Variablen in `pos-lit` noch unbewertet sind, und entsprechend für `act-neg` und `neg-lit`. Diese Zähler werden gebraucht, um Einerklauseln und leere Klauseln zu erkennen, und werden beim Bewerten von Variablen dekrementiert.

Der Zeiger sat ist der Nullzeiger NULL, solange die Klausel nicht erfüllt ist. Ist die Klausel erfüllt, so zeigt er auf die Variable, durch deren Bewertung die Klausel zuerst erfüllt wurde. Dadurch kann beim backtracking erkannt

werden, ob beim Rücksetzen des Wertes einer Variablen die Klausel wieder als unerfüllt zu markieren ist.
Die Struktur zur Darstellung einer Variable hat die Einträge:

- `wert`, der die Werte 0, 1 und NIL annimmt,
- zwei Zeiger `pos` und `neg` auf Listen von Klauseln.

Der `wert` ist NIL, solange die Variable unbewertet ist, und gibt sonst ihren Wert an. Die Listen `pos` und `neg` enthalten Zeiger auf diejenigen Klauseln, in denen die Variable positiv bzw. negativ vorkommt.
Eine Implementierung dieser Struktur in C ist in Tabelle 1 angegeben.

```
struct clause {
  struct variable *satisfied;
  struct var_list *pos_lit,*neg_lit;
  int act_pos,act_neg;
};

struct clause_list {
  struct clause *entry;
  struct clause_list *next;
};

struct variable {
  short value;
  struct clause_list *pos,*neg;
};

struct var_list {
  struct variable *entry;
  struct var_list *next;
};
```

Tabelle 1: Datenstruktur für KNF-Formeln in C

Bei Zuweisung des Wertes 1 an eine Variable x wird in dieser zunächst `wert` auf 1 gesetzt. Anschließend wird für jede Klausel in der Liste `pos-occ` der Zeiger `sat` auf x gesetzt, falls er NULL ist, also die Klausel noch nicht erfüllt ist. Danach wird für jede unerfüllte Klausel in `neg-occ` der Zähler `act-neg` dekrementiert. Eine Implementierung dieser Prozedur `set-true` in C ist in Tabelle 2 angegeben. Dual dazu arbeitet die Prozedur set-false, die eine Variable mit 0 bewertet.

```
void set_true(x)
     struct variable *x ;
{
  struct clause_list *c ;

  x->value = 1 ;
  c = x->pos ;
  while ( c ) {
    if ( c->entry->satisfied == NULL )
      c->entry->satisfied = x ;
    c = c->next ;
  }
  c = x->neg ;
  while ( c ) {
    if ( c->entry->satisfied == NULL )
      c->entry->act_neg-- ;
    c = c->next ;
  }
}
```

Tabelle 2: Prozedur `set-true` in C

Um beim backtracking die Bewertung einer Variablen x mit 1 rückgängig zu machen, wird zuerst in den unerfüllten Klauseln in `neg-occ` der Zähler `act-neg` wieder inkrementiert. Anschließend werden die Klauseln in `pos-occ` darauf untersucht, ob `sat` auf x zeigt, und in diesem Fall wird `sat` wieder auf NULL gesetzt. Schließlich wird textttwert wieder auf NIL gesetzt. Eine Implementierung dieser Prozedur `set-true` in C ist in Tabelle 3 angegeben. Dual dazu arbeitet die Prozedur unset-false, die die Bewertung einer Variablen mit 0 rückgängig macht.

Einerklauseln werden durch das Zählen der unbewerteten Literale einfach erkannt: eine Klausel ist zur Einerklausel geworden, wenn `sat` = NULL und `act-pos` + `act-neg` = 1 ist.

Sollen auch reine Literale erkannt werden, oder eine Verzweigungsheuristik die Anzahl der Vorkommen von Literalen benutzen, so kann diese auf ähnliche Weise gehalten werden. Die Struktur für Variablen wird dazu um zwei Zähler num-pos und num-pos ergänzt, die bei unbewerteten Variablen die Anzahl der positiven bzw. negativen Vorkommen in unerfüllten Klauseln angibt.

Diese Zähler müssen für alle unbewerteten Variablen in `pos-lit` und `neg-lit` dekrementiert werden, wenn eine Klausel in `set-true` oder `set-false` als erfüllt markiert wird. In der C-Prozedur `set_true(x)` sind also die Zeilen

```
void unset_true(x)
     struct variable *x ;
{
  struct clause_list *c ;

  c = x->neg;
  while ( c ) {
    if ( c->entry->satisfied == NULL )
      c->entry->act_neg++ ;
    c = c->next ;
  }
  c = x->pos ;
  while ( c ) {
    if ( c->entry->satisfied == x )
      c->entry->satisfied = NULL ;
    c = c->next ;
  x->value = NIL ;
  }
}
```

Tabelle 3: Prozedur `unset-true` in C

9-10 zu ersetzen durch das folgende Codesegment:

```
    if ( c->entry->satisfied == NULL ) {
      c->entry->satisfied = x;
      v = c->entry->pos_lit;
      while ( v ) {
        if ( v->entry->value == NIL )
          v->entry->num_pos--;
        v = v->next;
      }
      v = c->entry->neg_lit;
      while ( v ) {
        if ( v->entry->value == NIL )
          v->entry->num_neg--;
        v = v->next;
      }
    }
```

Analog sind die Zähler bei `unset-true` wieder zu inkrementieren, wenn eine Klausel als unerfüllt markiert wird.

3.2 Verzweigungsheuristiken

Die meisten in der Praxis benutzten Verzweigungsstrategien orientieren sich an der Zahl der Vorkommen on Literalen. Dabei gibt es einerseits einfache Strategien, die nur die absolute Anzahl von Vorkommen betrachten, andererseits komplexere, die die Vorkommen nach der Länge der Klauseln differenzieren. Grundidee aller dieser Heuristken ist, solche Bewertungen zu bevorzugen, bei denen möglichst viele kurze Klauseln, idealerweise Einerklauseln, entstehen.

Zur Formulierung dieser Strategien definieren wir zunächst die Werte

$$h_k(a) := \text{Anzahl der Vorkommen von } a \text{ in Klauseln der Größe } k$$

$$h(a) := \sum_{k=2}^{n} h_k(a) \ .$$

Neben Auswahlstrategien, die von der aktuellen Anzahl der Vorkommen in der bewerteten Formel abhängen, gibt es auch solche, wie etwa die unten beschriebene Auswahlstrategie von CHAFF, die von der aktuellen Variablenbewertung unabhängig sind. Dazu zählt auch die einfachste Auswahlstrategie, die einfach immer eine Variable aus den noch unbewerteten und einen Wert zufällig wählt.

Einfache Anzahlheuristiken

Wir betrachten zunächst solche Strategien, die nur die absolute Anzahl der Vorkommen von Literalen berücksichtigen:

DLIS (*dynamic largest individual sum*) : Wähle ein Literal a mit $h(a)$ maximal, und setze $[a \leftarrow 1]$. Es wird also die nächste Bewertung so ausgewählt, dass möglichst viele Klauseln erfüllt werden. Im anderen Zweig $[a \leftarrow 0]$ weerden dann möglichst viele Klauseln kleiner.

DLCS (*dynamic largest combined sum*) : Wähle eine Variable x mit $h(x) + h(\neg x)$ maximal, und setze $[x \leftarrow 1]$, falls $h(x) \geq h(\neg x)$, und $[x \leftarrow 0]$ sonst. Hier wird also die nächste Bewertung so ausgewählt, dass in beiden Zweigen möglichst viele Klauseln erfüllt bzw. kleiner werden.

Varianten Beide Auswahlstrategien gibt es auch in probabilistischen Varianten, bei der zwar die Variable nach der Strategie a, aber der zugewiesene Wert zufällig ausgewählt wird:

Die probabilistische Variante **RDLCS** (*randomized* DLCS) wählt also eine Variable x wie in DLCS, aber dann einen zufälligen Wert $\epsilon \in \{0, 1\}$ und setzt $[x \leftarrow \epsilon]$.

Bei der probabilistischen Variante RDLIS (*randomized* DLIS)wird eine Variable x so gewählt, dass $h(x)$ oder $h(\neg x)$ maximal wird, und dann dieser ein zufälliger Wert zugewiesen.

Die MOM-Heuristik

Die Abkürzung MOM steht für *maximum occurrences in minimal clauses*, die Strategie ist wie im folgenden beschrieben:

Sei $\ell \geq 2$ die minimale Länge der aktuell vorkommenden Klauseln. Dann wird eine Variable x gewählt, für die der Wert

$$(h_\ell(x) + h_\ell(\neg x)) \cdot 2^\alpha + h_\ell(x) \cdot h_\ell(\neg x)$$

maximal ist. Dabei ist α ein Parameter, der hinreichend gross gewählt sein sollte, so dass der erste Term den zweiten dominiert. Eigentlich soll also das Paar $(h_\ell(x) + h_\ell(\neg x), h_\ell(x) \cdot h_\ell(\neg x))$ in der lexikographischen Ordnung maximiert werden.

Die Idee ist, dass eine Variable gewählt wird, die am häufigsten in den kürzesten Klauseln auftritt, bei der also $h(x) + h(\neg x)$ maximal ist. Unter diesen wird eine gewählt, bei der das Produkt maximal wird, bei dem also die Anzahlen der Vorkommen von x und $\neg x$ sich möglichst wenig unterscheiden[1].

Bohm's Heuristik

Für eine Variable x wird der folgende Vektor $H(x)$ definiert:

$$\begin{aligned} H(x) &:= (H_2(x), \ldots, H_n(x)) \\ H_i(x) &:= \alpha \max(h_i(x), h_i(\neg x)) + \beta \min(h_i(x), h_i(\neg x)) \end{aligned}$$

wobei α und β frei wählbare Parameter sind.

Die Strategie wählt eine Variable x, für die der Vektor $H(x)$ in der lexikographischen Ordnung maximal ist.

Auch hier werden also bevorzugt Variablen ausgewählt, die in möglichst vielen kürzesten Klauseln vorkommen, bei Gleichheit wird nach den zweitkürzesten entschieden und so weiter. Dabei werden stets die Anzahl von Vorkommen

[1] Unter den Produkten ab mit gleicher Summe $a+b$ ist dasjenige mit $a = b$ das größte, d.h. von allen Rechtecken gleichen Umfangs hat das Quadrat den größten Flächeninhalt.

von x und $\neg x$ berücksichtigt, die entscheidende Größe ist eine gewichtete Summe aus diesen beiden Werten, mit unterschiedlichen Gewichten für den größeren und kleineren der zwei Werte.

Bei der ersten SAT Competition 1992 war der Algorithmus mit der besten Performanz eine Implementierung von DLL mit Bohm's Heuristik als Auswahlstrategie.

Die Jeroslaw-Wang Heuristik

Für ein Literal a wird der Wert $J(a)$ definiert als

$$J(a) := \sum_{i=1}^{n} h_i(a) 2^{-i} = \sum_{a \in C} 2^{-|C|} .$$

Bei der *einseitigen* Jeroslaw-Wang Heuristik wird ein Literal a, für das $J(a)$ maximal wird, gewählt und $[a \leftarrow 1]$ gesetzt.

Die *zweiseitige* Jeroslaw-Wang Heuristik wählt eine Variable x, für die $J(x)+J(\neg x)$ maximal wird, und setzt $[x \leftarrow 1]$ falls $J(x) \geq J(\neg x)$ und $[x \leftarrow 0]$ sonst.

Auch bei dieser Heuristik werden also, durch den in der Klauselläge invers exponentiellen Faktor, Variablen bevorzugt gewählt, die in vielen kurzen Klauseln vorkommen.

3.3 Iterative Implementierung

Bei einer iterativen Implementierung muss das Backtracking vom Programm selbst organisiert werden, während bei einer rekursiven Implementierung der Rekursionsstack diese Aufgabe übernimmt. Daher muss eine iterative Implementierung Operationen für das Setzen und das Rücksetzen einer Variablenbewertung enthalten, und sämtliche ausgeführte Bewertungen in einem Stack speichern, um sie später rückgängig machen zu können.

Jeder Eintrag auf dem Wertzuweisungsstack enthält folgende Informationen:

- `var` gibt an, welche Variable zugewiesen wurde.
- `value` gibt an, welcher Wert zugewiesen wurde.
- ein flag `branch` zeigt an, ob die Zuweisung durch Verzweigung oder durch Reduktion zustande gekommen ist.

Die iterative Prozedur DLL sieht also in Pseudocode wie folgt aus:

```
while true do
```

```
reduziere
if leere Klausel vorhanden
   then backtrack
if Formel erfüllt
   then output(stack)
wähle Variable x und Wert ϵ
push (x, ϵ, true)
set(x, ϵ)
```

Beim Backtracking müssen dann solange die Wertzuweisungen vom Stack gepopt und rückgängig gemacht werden, bis auf eine gestoßen wird, bei der das Flag branch gesetzt ist, die also durch Verzweigen zustande kam. Für diese Variable wird dann der andere Wert gesetzt und auf den Stack gepusht, wobei nun das flag branch gelöscht wird. Beim Rückgängigmachen dieser Setzung wird dann also weiter bis zur vorherigen Verzweigung zurückgegangen. Unerfüllbarkeit der Formel wird nun erkannt, wenn der Stack beim Backtracking leer wird, also keine Verzweigung mehr geändert werden kann.
In Pseudocode sieht die Backtracking-Prozedur aus wie folgt:

```
while not top.branch
   unset(top.var,top.value)
   pop
if stack-empty
   then output(unsatisfiable)
   else unset(top.var,top.value)
        top.value := 1−top.value
        top.branch := false
        set(top.var,top.value)
        if leere Klausel vorhanden
           then backtrack
```

3.4 Konfliktanalyse und Lernen

Der Implikationsgraph

Zur Vereinfachung nehmen wir an, dass Variablen nur durch Verzweigung und unit propagation bewertet werden.
Mit jeder Wertzuweisung wird eine Zahl assoziiert, die Entscheidungstiefe. Diese ist anfänglich 0 und wird bei jeder Wertzuweisung durch Verzweigen inkrementiert, und beim Backtracking wieder dekrementiert. Die nach einer Verzweigung folgenden Wertzuweisungen durch unit propagation erhalten die gleiche Entscheidungstiefe.

Der Implikationsgraph ist ein gerichteter azyklischer Graph, dessen Knoten mit jeweils einer Variable, ihrem Wert und der Enscheidungstiefe, bei der der Wert zugewiesen wurde, beschriftet sind. Wird bei Tiefe t der Variablen x der Wert ϵ zugewiesen, wird ein neuer Knoten in den Graphen eingefügt, diesen bezeichnen wir mit $\nu(x) = (x, \epsilon(x), t(x))$.

Ist diese Werzuweisung durch unit propagation entstanden, muss in der Formel eine Klausel

$$x^{\epsilon} \vee y_1^{\delta_1} \vee \ldots \vee y_k^{\delta_k}$$

vorkommen, so dass die Werte $\epsilon(y_i) = 1 - \delta_i$ bereits zugewiesen sind. In diesem Fall werden Kanten von den Knoten $\nu(y_i)$ zu $\nu(x)$ eingefügt. Diese sagen aus, dass die Wertzuweisungen an die y_i der Grund für die Wertzuweisung an x sind.

Mit $A(x)$ wird die Menge der Vorgänger von $\nu(x)$ im Implikationsgraphen bezeichnet, also im obigen Fall $A(x) = \{\nu(y_1), \ldots, \nu(y_k)\}$. Ist die Wertzuweisung an x durch Verzweigen entstanden, so ist $A(x) = \emptyset$.

Wird eine leere Klausel gefunden, so muss in der Formel eine Klausel

$$y_1^{\delta_1} \vee \ldots \vee y_k^{\delta_k}$$

vorkommen, so dass die Werte $\epsilon(z_i) = 1 - \delta_i$ zugewiesen sind. In diesem Fall wird ein spezieller Knoten $\square$ in den Graphen eingefügt, mit $A(\square) = \{\nu(y_1), \ldots, \nu(y_k)\}$.

Beispiel: es seien unter Anderen die folgenden Klauseln in der Formel vorhanden:

$$(\bar{x}_1 \vee x_2) \quad (\bar{x}_1 \vee x_3 \vee x_9) \quad (\bar{x}_3 \vee \bar{x}_3 \vee x_4) \quad (\bar{x}_4 \vee x_5 \vee x_{10})$$
$$(\bar{x}_4 \vee x_6 \vee x_{11}) \quad (\bar{x}_5 \vee \bar{x}_6) \quad (x_1 \vee x_7 \vee \bar{x}_{12}) \quad (x_1 \vee x_8) \quad (\bar{x}_7 \vee \bar{x}_8 \vee \bar{x}_{13})$$

und die folgenden Knoten im Implikationsgraphen:

$$(x_9, 0, 1) \quad (x_{10}, 0, 3) \quad (x_{11}, 0, 3) \quad (x_{12}, 1, 2) \quad (x_{13}, 1, 2)$$

und es werde nun die Zuweisung $[x_1 \leftarrow 1]$ als Verzweigung bei Entscheidungstiefe 6 ausgeführt. Dann wird dadurch der in Abbildung 1 gezeigte Ausschnitt des Implikationsgraphen angelegt.

Konfliktanalyse und nichtchronologisches Backtracking

Die Menge $A(x)$ wird partitioniert in $A(x) = A_<(x) \cup A_=(x)$, die definiert sind wie folgt:

$$A_<(x) := \{\, \nu(y) \in A(x) \,;\, t(y) < t(x) \,\}$$
$$A_=(x) := \{\, \nu(y) \in A(x) \,;\, t(y) = t(x) \,\}$$

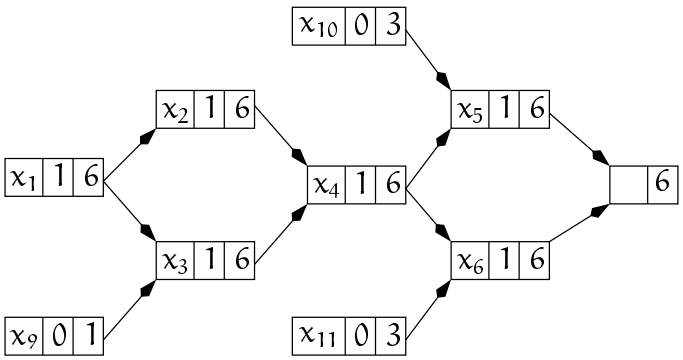

Abbildung 1: Implikationsgraph

Damit wird die Menge $G(x)$ der Gründe für die Zuweisung $[x \leftarrow \epsilon(x)]$ wird rekursiv definiert als

$$G(x) := \begin{cases} \{\nu(x)\} & \text{falls } A(x) = \emptyset \\ A_<(x) \cup \bigcup_{\nu(y) \in A_=(x)} G(y) & \text{sonst} \end{cases}$$

Die Konfliktmenge, also die Ursache für den durch die leere Klausel manifestierten Widerspruch, ist $G(\square)$. Im obigen Beispiel ist

$$G(\square) = \{ (x_{10}, 0, 3) , (x_{11}, 0, 3) , (x_9, 0, 1) , (x_1, 1, 6) \} .$$

Es sei $b := \max\{ t(x) ; \nu(x) \in G(\square) \}$

Falls $b = t(\square)$, dann handelt es sich bei dem eindeutigen Knoten $\nu(x)$ der Tiefe b in $G(\square)$ um die letzte Verzweigung.

Es werden dann alle Knoten der Tiefe b aus dem Graphen entfernt, und die entsprechenden Wertzuweisungen rückgängig gemacht. Dann wird für x die andere Zuweisung $[x \leftarrow 1 - \epsilon(x)]$ ausgeführt, also ein neuer Knoten $\nu'(x) = (x, 1 - \epsilon(x), t(x))$ in den Implikationsgraphen eingefügt, und dann die daraus folgenden unit propagations berechnet.

Dieser neue Knoten wird nun allerdings nicht als Verzweigungsknoten mit $A(x) = \emptyset$ behandelt. Da die Zuweisung $[x \leftarrow \epsilon(x)]$ zusammen mit den übrigen Zuweisungen in $G(\square)$ zwangsläufig zu einem Widerspruch führt, kann man es so auffassen, dass die übrigen Zuweisungen in $G(\square)$ die Zuweisung $[x \leftarrow 1 - \epsilon(x)]$ erzwingen. Daher erhält der neue Knoten $\nu'(x)$ diese als Vorgänger, es werden also Kanten von den Knoten in $G(\square) \setminus \{\nu(x)\}$ zu $\nu'(x)$ eingefügt, es wird also nun $A(x) = G(\square) \setminus \{\nu(x)\}$ gesetzt.

Ist dagegen $b < t(\square)$, so werden zunächst alle Knoten der Tiefe größer als b aus dem Graphen entfernt, und die entsprechenden Wertzuweisungen

rückgängig gemacht. Anschließend wird ein neuer Knoten $\square$ der Tiefe $t(\square) = b$ eingefügt und für diesen die Konfliktanalyse wiederholt, also wiederum die Konfliktmenge $G(\square)$ und daraus eine neue Backtrack-Tiefe b berechnet.

Im obigen Beispiel entsteht also nach Entfernen der Knoten der Tiefe 6 und Einfügen des neuen Knoten $(x_1, 0, 6)$ der in Abbildung 2 gezeigte neue Ausschnitt des Implikationsgraphen. Dann ist jetzt

$$G(\square) = \{\nu(x_9), \nu(x_{10}), \nu(x_{10}), \nu(x_{10}), \nu(x_{10})\}$$

und somit $b = 3$. Es werden also alle Zuweisungen der Tiefe mindestens 4 rückgängig gemacht, und dann ein neuer Knoten $\square$ mit $t(\square) = 3$ und $A(\square) = \{\nu(x_9), \nu(x_{10}), \nu(x_{10}), \nu(x_{10}), \nu(x_{10})\}$ eingefügt. Für diesen wird die neue Konfliktmenge $G(\square)$ berechnet.

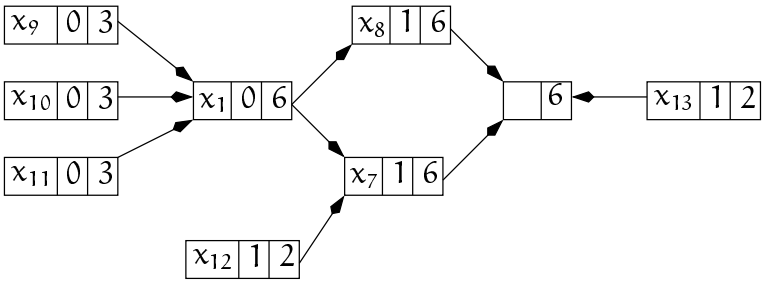

Abbildung 2: Der neue Implikationsgraph

Lernen von Klauseln

Wurde bei der Konfliktanalyse die Menge $G(\square)$ berechnet, so ist bekannt, dass die Wertzuweisungen in dieser Menge zu einem Widerspruch führen. Diese Feststellung kann aber unter Umständen erst lange nach dem Setzen dieser Wertzuweisungen erfolgt sein, d.h. es war viel Rechenarbeit nötig, um den Widerspruch aus diesen Wertzuweisungen zu finden.

Tritt in Zukunft an anderer Stelle im Verzweigungsbaum die selbe Kombination von Werten wieder auf, so möchte man sich diese Arbeit ersparen und den Widerspruch sofort erkennen, also die Tatsache, dass diese Wertzuweisung widersprüchlich ist, geeignet speichern Dies geschieht dadurch, dass eine Klausel zur Formel hinzugfügt wird, die unter dieser Wertzuweisung den Wert 0 erhält, nämlich die Klausel

$$C_{G(\square)} := \bigvee_{\nu(x) \in G(\square)} x^{1-\epsilon(x)} .$$

Um Einzusehen, dass die Hinzunahme dieser Klausel die Erfüllbarkeit der Formel nicht ändert, definieren wir allgemein für eine Menge M von Knoten im Implikationsgraphen die Klausel $C_M := \bigvee_{\nu(x)\in M} x^{1-\epsilon(x)}$, und betrachten noch einmal die Berechnung der Menge $G(\Box)$.

Diese wird berechnet durch das folgende Verfahren. Es sei t die aktuelle Verzweigungstiefe. Dann wird zunächst $G := A(\Box)$ gesetzt, und dann in einer Schleife jeder Knoten $\nu(x)$ der Tiefe $t(x) = t$ mit $A(x) \neq \emptyset$ durch die Knoten in $A(x)$ ersetzt. Die Schleife bricht ab, wenn kein solcher Knoten gefunden wird.

Betrachtet man die Klauseln C_G, die den Mengen G im Verlauf des obigen Algorithmus entsprechen, so ist zuerst $C_{A(\Box)}$ diejenige Klausel, die gerade leer geworden ist. Bei jeder Iteration der Schleife wird aus der Klausel C_G ein Literal $x^{1-\epsilon(x)}$ entfernt, und die Literale in $C_{A(x)}$ hinzugefügt. Nun ist aber $C_{A(x)} \vee x^{\epsilon(x)}$ gerade die Klausel, die bei der Zuweisung $[x \leftarrow \epsilon(x)]$ Einerklausel war. Die neue Klausel G ist also eine Resolvente aus dem alten G und dieser Klausel. Also entsteht jede der Klauseln C_G, und insbesondere die Klausel $C_{G(\Box)}$ am Ende der Schleife, durch Resolution aus Klauseln der Formel. Daher ist die um die Klausel $C_{G(\Box)}$ erweiterte Formel äquivalent zur ursprünglichen.

Zur Implementierung

Bei der Implementierung wird der Implikationsgraph dadurch repräsentiert, dass einer bewerteten Variabl ‚ein Zeiger auf die Klausel, die bei ihrer Bewertung Einerklausel war, beigegeben wird. Aus dieser Information läßt sich der gesamte Graph rekonstruieren.

Ist die Wertzuweisung durch Verzweigen zustande gekommen, so ist der Zeiger NULL. Bei einer beim Backtracking erzwungenen Wertzuweisung zeigt der Zeiger auf die gelernte Klausel $C_{G(\Box)}$. Der Zeiger ist also genau dann NULL, wenn es sich um eine Verzweigung handelt, für die der zweite Wert noch zu testen ist, er ersetzt also das Flag `branch` auf dem Wertzuweisungsstack.

Man sieht ier, dass zur Konfliktanayse manche der Klauseln $C_{G(\Box)}$ in jedem Fall gespeichert werden müssen. Soll nur nichtchronologisches Backtracking implementiert werden, so können diese gelöscht werden, wenn die erzwungene Wertzuweisung rückgängig gemacht wird. Beim Lernen hingegen werden diese Klauseln länger im Speicher behalten. Für die Entscheidung, wann diese wieder gelöscht werden, gibt es verschiedene Strategien, auf die hier aus Platzgründen nicht eingegangen wird.

Verzweigen anhand gelernter Klauseln

Bei SAT-Solvern, die das Lernen von Klauseln implementieren, kann auch die Verzweigungsstrategie von den gelernten Klauseln abhängig gemacht werden. So verwendet CHAFF die Strategie VSIDS (*variable state independent decaying sum*), die nur von den gelernten Klauseln, nicht aber von der aktuellen Bewertung abhägt.

Dazu erhält jedes Literal a einen Zähler, der am Anfang mit 0 oder mit der Zahl der Vorkommen von a in der Eingabeformel initialisiert wird. Wird eine gelernte Klausel C hinzugefügt, wird bei jedem Literal, das in C vorkommt, der Zähler inkrementiert.

Zusätzlich werden alle Zähler in regelmäßigen Abständen durch eine Konstante, etwa 2, dividiert.

Es wird beim Verzweigen stets eine Literal ausgewählt, bei dem der Zähler maximal ist. Da die Zähler relativ selten aktualisiert werden (im Gegensatz zu den Zählern für die DLIS oder DLCS Strategie), kann man die Literale in einer Prioritätsqueue, etwa einem Heap, halten.

Die Idee hinter dieser Strategie ist, dass immer soche Literale zur Verzweigung ausgewählt werden, die an vielen Konflikten beteiligt waren, und dabei (durch das regelmäßige Dividieren) solche bevorzugt werden, für die dies erst kürzlich der Fall war.

4 Lokale Suche und Zufallsirrfahrt

In diesem Kapitel werden Algorithmen betrachtet, die statt der Backtracking-Strategie eine lokale Verbesserungsstrategie verwenden. Alle diese Algorithmen arbeiten nach dem folgenden Schema:

```
wähle eine totale Bewertung α
repeat
    if α ⊨ F then return α
    wähle Variable x
    α := α with [x ← 1 − α(x)]
```

Dies wird solange iteriert, bis entweder eine erfüllende Bewertung $\alpha \models F$ gefunden wird, oder bis hinreichend lange gesucht wurde, dass man mit einiger Sicherheit auf die Unerfüllbarkeit von F geschlossen werden kann.

Verschiedene Algorithmen unterscheiden sich darin, wie einerseits die Anfangsbewertungen α gewählt werden, und wie die Variable x, an der die Bewertung bei der Suche geändert wird, ausgewählt wird. Beide Auswahlen können deterministisch oder zufällig getroffen werden.

Werden die Stellen x, an der die Bewertung geändert wird, systematisch ausgewählt, aber die Anfangsbewertungen zufällig, so führt dies zu einem einfachen probabilistischen Algorithmus, dem *Hammingkugel*-Algorithmus. Eine derandomisierte Version davon, bei der auch die Anfangsbewertungen deterministisch ausgewählt werden, ist der derzeit asymptotisch beste deterministische Algorithmus für k-SAT; er erreicht bei 3-SAT die Komplexität $O(1.481^n)$.

Werden dagegen beide Auswahlen zufällig getroffen, führt dies zum Irrfahrt-Algorithmus (*random walk*). Dieser ist mit einer Komplexität von $O((4/3)^n)$ der derzeit asymptotisch beste probabilistische Algorithmus für 3-SAT. Eine noch leicht verbesserte Version erreicht die Laufzeit $O(1.3302^n)$.

4.1 Hammingkugel-Algorithmus

Für zwei Bewertungen α, β ist die Hamming-Distanz

$$d(\alpha, \beta) := \left|\{x; \alpha(x) \neq \beta(x)\}\right| ,$$

die Hamming-Kugel vom Radius r um α ist

$$H(\alpha, r) := \{\, \beta \,;\, d(\alpha, \beta) \leq r \,\} .$$

Offenbar ist $|H(\alpha, r)| =: V(r) = \sum_{i=0}^{r} \binom{n}{r}$ unabhängig von α. Sei $\rho := r/n$ der normierte Radius. Es ist eine aus der Informationstheorie bekannte Tatsache,

dass für $0 < \rho \leq 1/2$ gilt

$$\frac{1}{\sqrt{8n\rho(1-\rho)}} 2^{h(\rho)n} \leq V(r) \leq 2^{h(\rho)n} ,$$

wobei $h(\rho) := -\rho \log \rho - (1-\rho)\log(1-\rho)$ die Entropiefunktion ist. Ist also ρ konstant, so ist bis auf einen polynomiellen Faktor $V(r) \approx 2^{h(\rho)n}$.

Der folgende Algorithmus durchsucht die Hammingkugel $H(\alpha, r)$ nach einer erfüllenden Bewertung.

```
suche(α, r)
  if α ⊨ F
      then return α
  if r = 0
      then return NIL
  wähle C = a_1 ∨ ... ∨ a_k mit Cα = 0
  for i := 1 to k do
      α_i := α with [a_i ← 1]
      β := suche(α_i, r − 1)
      if β ≠ NIL
          then return β
  return NIL
```

Der Zeitaufwand ist offenbar k^r, da ein k-fach verzweigender Suchbaum der Tiefe r aufgespannt wird. Dies ist unter Umständen weniger als das Volumen $V(r)$. So ist z.B. für $k = 3$ und $r = n/2$ das Volumen $V(r) \geq 2^{n-1}$, aber $k^r = \sqrt{3}^n \geq 1.7321^n$.

Damit ergibt sich ein einfacher deterministischer Algorithmus für 3-SAT, der für die Wertzuweisungen α_0, die alle Variablen mit 0 bewertet, und α_1, die alle mit 1 bewertet, nacheinander suche$(\alpha_0, n/2)$ und suche$(\alpha_0, n/2)$ aufruft. Die Komplexität ist $O(\sqrt{3}^n) = O(1.7321^n)$.

Eine bessere Komplexität erreicht man, wenn man einen kleineren Radius $r = \rho n$ für ein $\rho < \frac{1}{2}$ wählt, und dafür mehrere Startpunkte zufällig wählt. Damit erhält man die folgende probabilistische Variante des Hammingkugel-Algorithmus:

```
for i := 1 to w do
      wähle α_i zufällig
      β := suche(α_i, ρn)
      if β ≠ NIL
          then return β
  return NIL
```

Dabei werden die Anfangsbewertungen α_i gleichverteilt und unabhängig aus den 2^n möglichen Bewertungen der n Variablen gezogen.

Die Komplexität ist offenbar $w \cdot k^{\rho n}$. Es gilt nun, die Parameter w und ρ so zu wählen, dass die Fehlerwahrscheinlichkeit vernachlässigbar klein wird, und dabei die Laufzeit minimiert wird.

Der einzige mögliche Fehler ist, dass die Eingabeformel erfüllbar ist, aber der Algorithmus keine erfüllende Bewertung findet.

Sei also $\alpha \models F$. Ein Aufruf von suche$(\alpha_i, \rho n)$ findet α, falls $\alpha_i \in H(\alpha, r)$ ist, was mit Wahrscheinlichkeit $V(\rho n)2^{-n}$ geschieht. Diese Wahrscheinlichkeit ist ungefähr $2^{-(1-h(\rho))n}$.

Die Wahrscheinlichkeit, dass α bei w unabhängigen Wiederholungen nicht gefunden wird, ist also höchstens

$$\left(1 - 2^{-(1-h(\rho))n}\right)^w \leq e^{-w \cdot 2^{-(1-h(\rho))n}} .$$

Also werden $w = c \cdot 2^{-(1-h(\rho))n}$ Wiederholungen gebraucht, um die Fehlerwahrscheinlichkeit unter e^{-c} zu drücken. Daher ist die Laufzeit, bis auf einen polynomiellen Faktor

$$\left(2^{-(1-h(\rho))n} \cdot k^{\rho}\right)^n = \left(2\rho^{\rho}(1-\rho)^{(1-\rho)}k^{\rho}\right)^n .$$

Indem man die Basis nach ρ differenziert und das Ergebnis nach ρ auflöst, erhält man, dass der Ausdruck für $\rho = \frac{1}{k+1}$ minimal wird. Durch einsetzen und elementare Vereinfachungen erhält man dann die Basis $\frac{2k}{k+1}$. Die Komplexität des probabilistischen Algorithmus ist also $\left(\frac{2k}{k+1}\right)^n$. Diese Basis ist für kleine Werte von k:

k	3	4	5	6	7	8
$\frac{2k}{k+1}$	1.5	1.6	1.66667	1.71429	1.75	1.77778

Derandomisierung mit Überdeckungscodes

Bei der probabilistischen Version des Hammingkugel-Algorithmus sind die einzigen zufälligen Entscheidungen, die getroffen werden, die Wahl der Anfangsbewertungen α_i. Um einen deterministischen Algorithmus zu gewinnen, wird der Begriff des Überdeckungscodes verwendet.

Definition. *Eine Menge* C *von Bewertungen ist ein* Überdeckungscode *vom Radius* r*, falls für jede Bewertung* α *gilt* $\alpha \in H(\gamma, r)$ *für ein* $\gamma \in C$.

So ist zum Beispiel die oben betrachtete Menge $\{\alpha_0, \alpha_1\}$ aus den beiden konstanten Bewertungen ein Überdeckungscode vom Radius $n/2$.

Hat man einen Überdeckungscode C vom Radius r zur Verfügung, so hat man den folgenden deterministischen Algorithmus für k-SAT, mit Zeitkomplexität $|C| \cdot k^r$:

```
for each γ ∈ C do
    β := suche(γ, r)
    if β ≠ NIL
        then return β
return NIL
```

Um also einen deterministischen Algorithmus zu erhalten, brauchen wir ein Verfahren, um deterministisch einen Überdeckungscode zu konstruieren.

Das Problem, einen möglichst kleinen solchen Code zu konstruieren, ist ein Spezialfall des bekannten NP-schweren Optimierungsproblems SET COVER:

Gegeben:	Eine Menge S, und
	eine Familie $F \subseteq 2^S$ von Teilmengen von S
Gesucht:	Eine Teilfamilie $C \subseteq F$, die S überdeckt,
	d.h. $S = \bigcup C$.
Ziel:	Minimiere $\|C\|$.

In unserem Fall ist S die Menge 2^n möglichen Bewertungen, und F die Menge der Hammingkugeln vom Radius r.

Der folgende Greedy-Algorithmus löst das Problem Set Cover: Ein Element $s \in S$ heißt unüberdeckt, falls $s \notin \bigcup C$ ist.

Am Anfang wird $C = \emptyset$ gesetzt. Solange es noch unüberdeckte Elemente in S gibt, wähle ein $c \in F$, dass maximal viele unüberdeckte Elemente enthält, und füge es zi C hinzu.

Falls am Ende der Schleife $\bigcup C = S$ ist, ist C eine Lösung, andernfalls gibt es keine, d.h. $S \setminus \bigcup F \neq \emptyset$.

Es ist eine bekannte Tatsache, dass dieser Greedy-Algorithmus ein $\log|S|$-Approximationsverfahren für Set Cover ist, d.h. er liefert stets eine Lösung C mit $|C| \leq (\log|S|) \cdot |C^*|$, wobei C^* die optimale Lösung, also eine kleinstmögliche Überdeckung, ist.

In unserem Fall ist $\log|S| = O(n)$, also liefert der Greedy-Algorithmus einen Überdeckungscode, der nur um einen Faktor n größer ist als der minimale. Um die Größe, und damit die Komplexität des obigen deterministischen Verfahrens, abschätzen zu können, brauchen wir noch eine obere Schranke an die Größe eines minimalen Überdeckungscodes.

Diese liefert das folgende Lemma. Aus Kardinalitätsgründen muss jeder Überdeckungscode C die Größe $|C| \geq 2^n/V(r)$ haben. Man kann beweisen, dass immer ein Code existiert, der nur um einen Faktor n größer ist.

Lemma 18. *Für alle n und $r = \rho n$ mit $0 < \rho \leq 1/2$ gibt es einen Überdeckungscode vom Radius r der Größe*

$$s := n \cdot \sqrt{8n\rho(1-\rho)} \cdot 2^{(1-h(\rho))n} .$$

Beweis. Die Aussage wird mit der probabilistischen Methode bewiesen: es wird gezeigt, dass eine zufällig gewählte Menge von s Bewertungen mit nicht verschwindender Wahrscheinlichkeit ein Überdeckungscode ist. Daher muss es insbesondere einen solchen geben.

Sei also β eine feste Bewertung, und α eine zufällig gewählte. Dann ist $\beta \in H(\alpha, r)$ mit Wahrscheinlichkeit $V(r)/2^n$.

Ist also C eine Menge von s unabhängig zufällig gewählten Bewertungen, so ist die Wahrscheinlichkeit, dass β von C nicht überdeckt wird, höchstens

$$\left(1 - \frac{V(r)}{2^n}\right)^s \leq e^{-s\frac{V(r)}{2^n}} \leq e^{-s\frac{2^{(h(\rho)-1)n}}{\sqrt{8n\rho(1-\rho)}}} \leq e^{-n} .$$

Somit ist die Wahrscheinlichkeit, dass irgendeine Bewertung nicht überdeckt wird, höchstens $2^n \cdot e^{-n} < 1$, also ist C mit positiver Wahrscheinlichkeit ein Überdeckungscode. □

Da es stets einen Überdeckungscode dieser Größe s gibt, ist insbesondere der optimale Code C^* von der Größe $|C^*| \leq s$, und damit liefert der Greedy-Algorithmus einen Überdeckungscode der Größe

$$O(n) \cdot s = O(n^{5/2}) \cdot 2^{(1-h(\rho))n} .$$

Allerdings ist die Laufzeit des Greedy-Algorithmus für unsere Zwecke zu groß: selbst im günstigsten Fall braucht er mindestens $2^{(1-h(\rho))n}$ Auswahlen, und bei jeder Auswahl muss die Zahl der unüberdeckten Bewertungen für alle 2^n Hammingkugeln aktualisiert werden, die Laufzeit ist also selbst bei günstigster Implementierung größer als 2^n. Andererseits läuft der Greedy-Algorithmus aber auch, selbst in der einfachsten möpglichen Implementierung, in Zeit 2^{3n}.

Die Lösung besteht in der folgenden einfachen Beobachtung: ist C ein Überdeckungscode vom Radius r für n Variablen, so gewinnt man daraus einen ebensolchen für dn Variablen, indem man die Variablen in d Blöcke mit je n Variablen partitioniert, und alle Bewertungen betrachtet, in denen die Variablen in den Blöcken unabhängig mit allen Bewertungen aus C bewertet werden. Dieser Code C^d hat die Größe $|C|^d$ und den Radius $r \cdot d$.

Es wird also der Greedy-Algorithmus benutzt, um einen Überdeckungscode C für $n/6$ Variablen von Radius $\rho n/6$ zu finden. Dies braucht die Zeit $2^{3n/6} = \sqrt{2}^n$. Dann benutzt man den Code C^6, der die Größe

$$\left[O\left(\left(\frac{n}{6}\right)^{5/2}\right) \cdot 2^{(1-h(\rho))\frac{n}{6}}\right]^6 = O(n^{15}) \cdot 2^{(1-h(\rho))n}$$

hat. Die Gesamtlaufzeit des deterministischen Algorithmus setzt sich also zusammen aus der Zeit $\sqrt{2}^n$ für den Greedy-Algorithmus zum Auffinden

des Überdeckungscodes C, plus der Zeit $\left(2^{1-h(\rho)}k^{\rho}\right)^n$ zum Durchsuchen der Hammingkugeln vom Radius ρn um die Bewertungen im Code C[6]. Der deterministische Algorithmus hat also im wesentlichen die gleiche Zeitkomplexität wie der probabilistische Hammingkugel-Algorithmus, nämlich $\left(\frac{2k}{k+1}\right)^n$ für k-SAT.

Der Nachteil dieser Methode ist, dass der Greedy-Algorithmus zur Konstruktion des Überdeckungscodes exponentiell viel Speicherplatz braucht. Es gibt eine zweite Methode, die ohne zusätzlichen Speicherbedarf auskommt, aber dafür eine etwas schlechtere Zeitkomplexität liefert.

Für eine gegebenes $\delta > 0$ wähle ein kleinstmögliches b so dass folgende Ungleichung gilt:

$$b\sqrt{8b\rho(1-\rho)}2^{(1-h(\rho))b} \leq 2^{(1-h(\rho)+\delta)b}$$

Dann existiert nach Lemma 18 ein Überdeckungscode C_b für b Variablen vom Radius ρb der Größe $2^{(1-h(\rho)+\delta)b}$. Dieser endliche Code, den man etwa durch einmalige exhaustive Suche findet, wird im Algorithmus festverdrahtet. Man gewinnt daraus den Code $C_b^{n/b}$ für n Variablen vom Radius ρn, dieser hat die Größe $2^{(1-h(\rho)+\delta)n}$.

Es sei nun ein $\epsilon > 0$ vorgegeben. Dann setzt man $\delta := \log\left(\frac{k+1}{2k}\epsilon + 1\right)$, und benutzt den Code C_b für das zu diesem δ gehörige b wie oben. Die Laufzeit des deterministischen Algorithmus berechnet sich nun wie oben als $\left(2^{1-h(\rho)+\delta} \cdot k^{\rho}\right)^n$, und mit der Setzung $\rho := 1/(k+1)$ erhält man, dass sie Basis sich errechnet zu

$$\begin{aligned}
\left(2^{1-h(\rho)+\delta} \cdot k^{\rho}\right) &= \left(2^{1+\frac{1}{k+1}\log(\frac{1}{k+1})+\frac{k}{k+1}\log(\frac{k}{k+1})+\delta} \cdot k^{\frac{1}{k+1}}\right) \\
&= \left(2^{1-\frac{1}{k+1}\log(k+1)+\frac{k}{k+1}\log k-\frac{k}{k+1}\log(k+1)+\delta} \cdot 2^{\frac{1}{k+1}\log k}\right) \\
&= 2^{1+\log k-\log(k+1)+\delta} \;=\; 2^{1+\log(\frac{k}{k+1})+\delta} \\
&= 2\frac{k}{k+1}\left(\frac{k+1}{2k}\epsilon + 1\right) \;=\; \frac{2k}{k+1} + \epsilon
\end{aligned}$$

Also erhalten wir als Komplexität den Wert $\left(\frac{2k}{k+1} + \epsilon\right)^n$, für beliebig kleines $\epsilon > 0$.

Für k-SAT mit $k \geq 4$ ist dies der beste bekannte deterministische Algorithmus. Für 3-SAT ist die Komplexität $(1.5 + \epsilon)^n$, die von dem oben erwähnten Algorithmus von Schiermeyer unterboten wird. Es ist aber möglich, die Prozedur suche(α, r) im Falle von 3-SAT geschickter zu implementieren, so dass sie statt 3^r nur eine Laufzeit von 2.848^r benötigt. Durch Verwendung dieser verbesserten Prozedur erreicht der deterministische Hammingkugel-Algorithmus die Komplexität 1.481^n, und ist damit auch der beste bekannte deterministische Algorithmus für 3-SAT.

4.2 Zufallsirrfahrt

Ein anderer Ansatz besteht darin, nicht nur die Anfangsbewertungen zufälig zu wählen, sondern auch die Verbesserungsschritte. Die innere Schleife führt also eine zufällige Irrfahrt (random walk) im Raum der möglichen Bewertungen durch. Der folgende Algorithmus setzt diese Idee um:

```
repeat w times
    wähle α zufällig
    repeat 3n times
        if α ⊨ F
            then return α
        wähle C = a_1 ∨ ... ∨ a_k mit Cα = 0
        wähle i zufällig aus {1, ..., k}
        α := α with [a_i ← 1]
return NIL
```

Dabei werden die Anfangsbewertungen α wiededrum gleichverteilt und unabhängig gewählt.

Nun ist die Anzahl der Wiederholungen w so zu wählen, dass die Fehlerwahrscheinlichkeit klein wird. Da der Algorithmus nur dann einen Fehler machen, wenn F erfüllbar ist, sei wieder α^* eine fest gewählte Bewertung mit $\alpha^* \models F$.

Wir berechnen die Wahrscheinlichkeit $p(n)$ dafür, dass ein Durchlauf der inneren Schleife α^* findet. Dann ergibt sich die Fehlerwahrscheinlichkeit als $(1-p(n))^w \leq e^{-wp(n)}$, und somit reichen $w = c\frac{1}{p(n)}$ Wiederholungen aus, damit die Fehlerwahrscheinlichkeit höchstens e^{-c} beträgt.

Da $\alpha^* \models F$, gibt es in jeder Klausel $C = (a_1 \vee \ldots \vee a_k)$ in F mindestens ein Literal a_i mit $\alpha^*(a_i) = 1$. Für jede Klausel C wählen wir nun ein festes Literal a_C mit $\alpha^*(a_C) = 1$.

Zur Analyse der Erfolgswahrscheinlichkeit $p(n)$ eines Schleifendurchlaufs wird nun eine Zufallsvariable Z definiert. Bei der Wahl der Anfangsbewertung α ist Z die Hamming-Distanz zwischen α und α^*, also

$$Z := d(\alpha, \alpha^*) \ .$$

Wird in einem Durchlauf der Schleife das Literal a_i aus der Klausel C gewählt, so wird gesetzt

$$Z := \begin{cases} Z-1 & \text{falls } a_i = a_C \\ Z+1 & \text{sonst.} \end{cases}$$

Wird $a_i = a_C$ gewählt, so verringert sich die Distanz $d(\alpha, \alpha^*)$ um 1, da $\alpha^*(a_C) = 1$ ist. Wird dagegen ein Literal $a_i \neq a_C$ gewählt, so wird die Distanz $d(\alpha, \alpha^*)$ entweder um 1 kleiner (falls $\alpha^*(a_i) = 1$ ist) oder um 1 größer (falls $\alpha^*(a_i) = 0$ ist). Es gilt also stets $Z \geq d(\alpha, \alpha^*)$.

Daher ist die Wahrscheinlichkeit, dass α^* gefunden wird, mindestens so gross wie die Wahrscheinlichkeit, dass $Z = 0$ erreicht wird. Diese Wahrscheinlichkeit ist aber wesentlich leichter auszurechnen, als direkt die Wahrscheinlichkeit zu berechnen, dass $\alpha = \alpha^*$ wird, da sich die Zufallsvariable Z sehr einfach verhält.

Man kann sich Z als den Zustand in einem probabilistischen Automaten (einer sog. Markov-Kette), wie in Abbildung 3 dargestellt, vorstellen. Der Automat startet in einem zufällig zwischen $Z = 0$ und $Z = n$ gewählten Anfangszustand, und geht dann vom einem Zustand Z stets mit Wahrscheinlichkeit $\frac{1}{k}$ in den Zustand $Z - 1$ und mit Wahrscheinlichkeit $1 - \frac{1}{k}$ in den Zustand $Z + 1$ über.

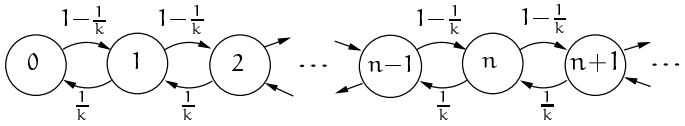

Abbildung 3: Die Markov-Kette zum Irrfahrt-Algorithmus

Der Wahrscheinlichkeitsverteilung des Anfangszustand lässt sich nun leicht ausrechnen, da α gleichverteilt gewählt wird, und am Anfang $Z = d(\alpha, \alpha^*)$ ist. Es gibt $\binom{n}{j}$ Bewertungen, die von α^* die Distanz j haben, und jede davon wird mit Wahrscheinlichkeit 2^{-n} gewählt. Der Automat startet daher mit Wahrscheinlichkeit $2^{-n}\binom{n}{j}$ im Zustand $Z = j$.

Es bezeichne nun p_j die Wahrscheinlichkeit, dass der Zustand $Z = 0$ ausgehend von Anfangszustand $Z = j$ erreicht wird. Mit Hilfe der Theorie der Markov-Ketten errechnet man den Wert $p_j = \left(\frac{1}{k-1}\right)^j$. Daher errechnet sich die Wahrscheinlichkeit, dass $Z = 0$ erreicht wird, zu

$$\begin{aligned}
&\sum_{j=0}^{n} 2^{-n}\binom{n}{j} p_j \\
&= 2^{-n} \sum_{j=0}^{n} \binom{n}{j} \left(\tfrac{1}{k-1}\right)^j \\
&= 2^{-n}\left(1 + \tfrac{1}{k-1}\right)^n \\
&= \left(\frac{k}{2(k-1)}\right)^n
\end{aligned}$$

Damit ist auch die Erfolgswahrscheinlichkeit eines Schleifendurchlaufs mindestens $\left(\frac{k}{2(k-1)}\right)^n$, und wir erhalten als obere Schranke an die nötige Zahl von Wiederholungen, und damit die Laufzeit des Algorithmus $O\left(\left(\frac{2(k-1)}{k}\right)^n\right)$. Für $k = 3$ ist dies $(4/3)^n$, und für größere Werte von k erhalten wir die folgenden Basen:

k	3	4	5	6	7	8
$\frac{2(k-1)}{k}$	1.33334	1.5	1.6	1.66667	1.71429	1.75

Verbesserung durch unabhängige Klauseln

Für 3-SAT läßt sich die Komplexität des Irrfahrt-Algorithmus noch leicht verbessern.

Definition. *Eine Menge* $G \subseteq F$ *von Klauseln in* F *heißt* unabhängig, *wenn je zwei Klauseln* C, D *in* F *variablendisjunkt sind, also* $V(C) \cap V(D) = \emptyset$ *ist.*

Ist G unabhängig, so ist $|V(G)| = 3|G|$.

Der Algorithmus berechnet zunächst in Greedy-Manier eine maximale unabhängige Teilmenge G in F wie folgt:

$G := \emptyset$
while es gibt C in F mit $V(C) \cap V(G) = \emptyset$
    $G := G \cup \{C\}$

Nun wird je nach der Größe von G unterschiedlich verfahren:

Da G maximal unabhängig ist, enthält jede der übrigen Klauseln in $F \setminus G$ mindestens eine der Variablen in $V(G)$ enthalten. Wäre nämlich C eine Klausel, für die dies nicht der Fall ist, so könnte G um diese Klausel erweitert werden.

Ist nun α eine Bewertung der Variablen in $V(G)$, so ist $F\alpha$ eine 2-KNF-Formel, für die das Erfüllbarkeitsproblem in polynomieller Zeit gelöst werden kann. Es gibt insgesamt $7^{|G|}$ Bewertungen, die die Klauseln in G erfüllen.

Ist nun G klein, nämlich $|G| \leq \delta n$ für einen noch zu bestimmenden Parameter $0 < \delta \leq \frac{1}{3}$, so wird für jede der $7^{|G|}$ Bewertungen $\alpha \models G$ das Erfüllbarkeitsproblem für die 2-SAT Instanz $F\alpha$ gelöst. Die Zeitkomplexität ist damit im wesentlichen $7^{\delta n}$.

Ist dagegen $|G| > \delta n$, so wird der Irrfahrt-Algorithmus durchgeführt, wobei allerdings die Anfangsbewertungen nicht gleichverteilt, sondern gemäß der Verteilung $A_G(p_1, p_2, p_3)$ gewählt werden.

Für Wahrscheinlichkeiten $p_1, p_2, p_3 > 0$ mit $3p_1 + 3p_2 + p_3 = 1$ ist die Verteilung $V_G(p_1, p_2, p_3)$ von Bewertungen definiert wie folgt:

Zunächst werden die Variablen in $V(G)$ bewertert, indem für jede Klausel $C = (a_1 \vee a_2 \vee a_3)$ in G Werte $\epsilon_1, \epsilon_2, \epsilon_3$ mit Wahrscheinlichkeit nach der Tabelle

ϵ_1	ϵ_2	ϵ_3		ϵ_1	ϵ_2	ϵ_3	
0	0	0	0	0	1	1	p_2
0	0	1	p_1	1	0	1	p_2
0	1	0	p_1	1	1	0	p_2
1	0	0	p_1	1	1	1	p_3

gewählt werden, und dann die Bewertung

$$[a_1 \leftarrow \epsilon_1, a_1 \leftarrow \epsilon_2, a_1 \leftarrow \epsilon_3]$$

gesetzt wird. Anschliessend wird für jede Variable x in $V(F) \setminus V(G)$ ein Wert $\epsilon \in \{0, 1\}$ zufällig gewählt und $[x \leftarrow \epsilon]$ gesetzt.

Zur Abschätzung der Erfolgswahrscheinlichkeit erinnern wir und, dass die Wahrscheinlichkeit, von einer Anfangsbewertung α aus die erfüllende Bewertung α^* zu erreichen, mindestens

$$\left(\frac{1}{k-1}\right)^{d(\alpha,\alpha^*)}$$

ist. Für $k = 3$ ist dies $\left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)}$.

Es bezeichne $p(\alpha)$ die Wahrscheinlichkeit, dass α als Anfangsbewertung gewählt wird. Dann ist die Erfolgswahrscheinlichkeit

$$p(n) \geq \sum_{\alpha} p(\alpha)\left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)} ,$$

und der rechte Term ist gerade der Erwartungswert $E[\left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)}]$ der Zufallsvariable $\left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)}$.

Wir betrachten nun folgende Zufallsvariablen: X_i sei die Indikatorvariable des Ereignisses, dass $\alpha(x_i) \neq \alpha^*(x_i)$ ist, also

$$X_i = \begin{cases} 1 & \text{falls } \alpha(x_i) \neq \alpha^*(x_i) \\ 0 & \text{sonst.} \end{cases}$$

Dann ist $d(\alpha, \alpha^*) = X_1 + \ldots + X_n$, also ist

$$E\left[\left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)}\right] = E\left[\left(\frac{1}{2}\right)^{X_1+\ldots+X_n}\right] .$$

Bei gleichverteilter Wahl von α sind nun die Variablen X_i unabhängig, daher können wir ausrechnen

$$E\left[\left(\frac{1}{2}\right)^{X_1+\ldots+X_n}\right] = \prod_{i=1}^{n} E\left[\left(\frac{1}{2}\right)^{X_i}\right],$$

und der Erwartungswert $E[(\frac{1}{2})^{X_i}]$ ist $(\frac{1}{2})^0 \cdot \frac{1}{2} + (\frac{1}{2})^1 \cdot \frac{1}{2} = \frac{3}{4}$, woraus wir wie oben $p(n) \geq (\frac{3}{4})^n$ erhalten.

Im Fall der Wahl von α gemäß $V_G(p_1, p_2, p_3)$ sind die Zufallsvariable X_i nicht mehr unabhängig, aber Abhängigkeit besteht nur zwischen den Variablen innerhalb einer Klausel in G.

Es sei also $G = \{C_1, \ldots, C_{\delta n}\}$ mit $V(C_{i+1}) = \{x_{3i+1}, x_{3i+1}, x_{3i+1}\}$ für $i = 0, \ldots, \delta n - 1$. Wir definieren die Zufallsvariable $S_i = X_{3i+1} + X_{3i+1} + X_{3i+1}$, dann sind diese Variablen S_i wiederum unabhängig, und wir erhalten

$$E\left[\left(\frac{1}{2}\right)^{X_1+\ldots+X_n}\right] = \prod_{i=1}^{\delta n} E\left[\left(\frac{1}{2}\right)^{S_i}\right] \cdot \prod_{j=3\delta n+1}^{n} E\left[\left(\frac{1}{2}\right)^{X_j}\right].$$

Der rechte Faktor berechnet sich wie oben zu $(\frac{3}{4})^{1-3dn}$.

Zur Berechnung des linken Faktors brauchen wir die Erwartungswerte der Variablen $(\frac{1}{2})^{S_i}$. Diese hängen aber davon ab, wieviele der Literale in C_i von α^* erfüllt werden.

Falls α^* genau ein Literal a in C_i mit 1 bewertet, so hat S_j den Wert 0, wenn α auf C_i mit α^* übereinstimmt, also nur a mit 1 und die anderen Literale in C_i mit 0 bewertet. Dies passiert mit Wahrscheinlichkeit p_1.

Weiter hat S_i den Wert 1, wenn α das Literal a und ein weiteres mit 1 bewertet, dafür gibt es 2 Möglichkeiten, die jeweils mit Wahrscheinlichkeit p_2 eintreten.

S_i hat den Wert 2, wenn α alle drei Literale in C_i mit 1 bewertet, was mit Wahrscheinlichkeit p_3 eintritt, oder wenn α das Literal a mit 0 und ein anderes mit 1 bewertet, was mit Wahrscheinlichkeit $2p_1$ eintritt.

Schließlich hat S_i den Wert 3, wenn a von α mit 0 und die zwei anderen Literale mit 1 bewertet werden, was mit Wahrcheinlichkeit p_2 eintritt.

Insgesamt errechnet sich der Erwartungswert in diesem Fall also als

$$E\left[\left(\tfrac{1}{2}\right)^{S_i}\right] = \left(\tfrac{1}{2}\right)^0 p_1 + \left(\tfrac{1}{2}\right)^1 2p_2 + \left(\tfrac{1}{2}\right)^2 (2p_1 + p_3) + \left(\tfrac{1}{2}\right)^3 p_2 = \tfrac{3}{2}p_1 + \tfrac{9}{8}p_2 + \tfrac{1}{4}p_3 .$$

Im Fall, dass α^* genau zwei Literale in C_i mit 1 bewertet, errechnet man auf ähnliche Weise

$$E\left[\left(\tfrac{1}{2}\right)^{S_i}\right] = \left(\tfrac{1}{2}\right)^0 p_2 + \left(\tfrac{1}{2}\right)^1 (2p_1 + p_3) + \left(\tfrac{1}{2}\right)^2 2p_2 + \left(\tfrac{1}{2}\right)^3 p_1 = \tfrac{9}{8}p_1 + \tfrac{3}{2}p_2 + \tfrac{1}{2}p_3 .$$

Im letzten Fall, wo α^* alle drei Literale in C_i mit 1 bewertet, errechnet man schließlich

$$E\left[\left(\tfrac{1}{2}\right)^{S_i}\right] = \left(\tfrac{1}{2}\right)^0 p_3 + \left(\tfrac{1}{2}\right)^1 3p_2 + \left(\tfrac{1}{2}\right)^2 3p_1 = \tfrac{3}{4}p_1 + \tfrac{3}{2}p_2 + p_3 \ .$$

Für $j = 1, 2, 3$ sei also m_j die Anzahl der Klauseln in G, in denen α^* genau j Literale erfüllt. Dann ist also der linke Faktor des obigen Produktes gleich

$$\left(\tfrac{3}{2}p_1 + \tfrac{9}{8}p_2 + \tfrac{1}{4}p_3\right)^{m_1} \cdot \left(\tfrac{9}{8}p_1 + \tfrac{3}{2}p_2 + \tfrac{1}{2}p_3\right)^{m_2} \cdot \left(\tfrac{3}{4}p_1 + \tfrac{3}{2}p_2 + p_3\right)^{m_3} \ .$$

und durch Lösen des linearen Gleichungssystems aus der Gleichheit der drei Basen zusammen mit der Einschränkung $3p_1 + 3p_2 + p_3 = 1$ erhält man die Werte

$$p_1 = \tfrac{4}{21}, \qquad p_2 = \tfrac{2}{21}, \qquad p_3 = \tfrac{1}{7} \quad ,$$

für die sich die Basis als $frac37$ ergibt. Damit ist die Erfolgswahrscheinlichkeit mindestens

$$p(n) \geq \left(\tfrac{3}{7}\right)^{m_1+m_2+m_3} \cdot \left(\tfrac{3}{4}\right)^{1-3\delta n} = \left(\tfrac{3}{7}\right)^{\delta n} \cdot \left(\tfrac{3}{4}\right)^{1-3\delta n} \ ,$$

und damit die nötige Anzahl von Wiederholungen und somit auch die Laufzeit des Algorithmus

$$\left(\left(\tfrac{4}{3}\right)^{1-3\delta} \cdot \left(\tfrac{7}{3}\right)^{\delta}\right)^n \ .$$

Man beobachtet nun, dass dieser Wert in δ monoton fallend ist, während die Laufzeit $7^{\delta n}$ im anderen Zweig monoton in δ ansteigt. Der minimale Wert der Laufzeit ergibt sich also an der Schnitstelle, wo

$$7^{\delta} = \left(\tfrac{4}{3}\right)^{1-3\delta} \cdot \left(\tfrac{7}{3}\right)^{\delta}$$

ist. Dieser Wert von δ ist ungefähr 0.14666, und der Wert 7^{δ} an dieser Stelle ist ungefähr 1.33026, also erhalten wir für den Algorithmus insgesamt eine Laufzeit von $O(1.33026^n)$.

Es gibt noch eine verbesserte Version des Algorithmus, die eine Laufzeit von $O(1.330193^n)$ erreicht, diese wird aber aus Platzgründen hier nicht behandelt.

5 Algorithmus von Paturi, Pudlák und Zane

In diesem Kapitel betrachten wir einen probabilistischen Algorithmus, der der Vorgehensweise bei DLL ähnlicher ist also die im vorherigen Kapitel.

Wie bei einem DLL-Verfahren werden die Variablen der Reihe nach betrachtet und ihnen Werte zugewiesen. Dabei werden Variablen in Einerklauseln natürlich so bewertet, dass diese erfüllt sind, die übrigen Variablen werden zufällig bewertet. Die Reihenfolge, in der die Variablen betrachtet werden, ist durch eine Permutation $\pi \in S_n$ der Indizes $1, \ldots, n$ vorgegeben, die ebenfalls zufällig gewählt wird.

Ist am Ende die Formel erfüllt, wird die so gefundene Bewertung audsgegeben, andernfalls wird sie verworfen und mit einer neuen Permutation von vorn begonnen.

Die Prozedur suche(π) betrachtet die Variablen in der durch π vorgegebenen Reihenfolge und weist ihnen einen zufälligen Wert zu, außer wenn ein Wert bereits durch Einerklauseln erzwungen ist.

suche(π)
  for $i := 1$ to n do
    if Einerklausel $x^{\epsilon}_{\pi(i)}$ in F
      then $\epsilon_i := \epsilon$
      else wähle ϵ_i zufällig aus $\{0, 1\}$
    $F := F[x_{\pi(i)} \leftarrow \epsilon_i]$
  if $\alpha \models F$
    then return α
    else return NIL

Diese Prozedur wird in einer Schleife eine bestimmte Anzahl w mal wiederholt.

  repeat w times
    wähle $\pi \in S_n$ zufällig
    $\beta :=$ suche(π)
    if $\beta \neq$ NIL
      then return β
  return NIL

Wie bei den probablilistischen Algorithmen im vorhergehenden Kapitel berechnen wir die Wahrscheinlichkeit $p(n)$ dafür, dass ein Schleifendurchlauf erfolgreich ist, also eine erfüllende Bewertung findet, im Fall dass F erfüllbar ist. Daraus ergibt sich, dass $w = O(1/p(n))$ Wiederholungen genügen, um die Fehlerwahrscheinlichkeit beliebig klein zu halten, und damit eine Laufzeit von $O(1/p(n))$.

Definition. *Für eine Menge A von Bewertungen und $\alpha \in A$ ist*

$$N_A(\alpha) := |H(\alpha, 1) \cap A| - 1$$

die Anzahl der Nachbarn von α in A.

Lemma 19. *Ist $A \neq \emptyset$, so gilt*

$$\sum_{\alpha \in A} 2^{-N_A(\alpha)} \geq 1 \ .$$

Beweis. Die Aussage wird durch Induktion nach n bewiesen. Der Induktionsanfang für $n = 0$ ist trivial.

Sei also $n > 0$, und gelte die Aussage für Mengen von Bewertungen von $n-1$ Variablen. Für eine Bewertung α der Variablen $x_1, \ldots, x_{n-1}$ und $i = 0, 1$ sei $\alpha i := \alpha \cup [x_n \leftarrow i]$.

Sei $A \neq \emptyset$ eine Menge von Bewertungen aller n Variablen. Für $i = 0, 1$ sei $A_i := \{\, \alpha \,;\, \alpha i \in A \,\}$. Für mindestens ein i muss $A_i \neq \emptyset$ sein.

Ist $A_0 = \emptyset$, so muss $A_1 \neq \emptyset$ sein. Also ist

$$\sum_{\alpha \in A} 2^{-N_A(\alpha)} \ = \ \sum_{\alpha \in A_1} 2^{-N_A(\alpha 1)} \ = \ \sum_{\alpha \in A_1} 2^{-N_{A_1}(\alpha)} \ \geq \ 1 \ ,$$

wobei die letzte Ungleichung nach der Induktionshypothese gilt. Die letzte Gleichheit gilt, da für $\alpha \in A_1$ der einzige Nachbar von $\alpha 1$, der nicht von der Form $\beta 1$ ist, gerade $\alpha 0$ ist. $A_0 = \emptyset$ ist aber $\alpha 0 \notin A$.

Der Fall $A_1 = \emptyset$ ist symmetrisch. Seien nun A_0 und A_1 beide nichtleer. Dann ist

$$\begin{aligned} \sum_{\alpha \in A} 2^{-N_A(\alpha)} \ &= \ \sum_{\alpha \in A_0} 2^{-N_A(\alpha 0)} + \sum_{\alpha \in A_1} 2^{-N_A(\alpha 1)} \\ &\geq \ \sum_{\alpha \in A_0} 2^{-N_{A_0}(\alpha)-1} + \sum_{\alpha \in A_1} 2^{-N_{A_1}(\alpha)-1} \\ &\geq \ \frac{1}{2} + \frac{1}{2} \ = \ 1 \ . \end{aligned}$$

Dabei gilt die vorletzte Ungleichung wiederum, da es für $i = 0, 1$ höchstens einen Nachbarn von αi gibt, der nicht von der Form β_i ist, nämlich $\alpha 0$ □

Zur Berechnung der Erfolgswahrscheinlichkeit $p(n)$ sei $A := \{\, \alpha \,;\, \alpha \models F \,\}$ die Menge der F erfüllenden Bewertungen, und sei $\alpha \in A$. Definiere

$$j(\alpha) := n - N_A(\alpha)$$

und für $i = 1, \ldots, n$ sei $\alpha_i := \alpha$ with $[x_i \leftarrow 1 - \alpha(x)]$. Das heißt, es gibt $j(\alpha)$ Variablen x_i mit $\alpha_i \not\models F$, diese heißen die *kritischen* Variablen.

Für jede kritische Variable x_i gibt es eine *kritische* Klausel C_i in F, derart dass das einzige Literal a in C_i mit $\alpha(a) = 1$ entweder x_i oder $\neg x_i$ ist.
Für eine Permutation $\pi \in S_n$ sei

$$R(\pi, \alpha) = \left\{ x_i \text{ kritisch} ; \pi^{-1}(i) \geq \pi^{-1}(j) \text{ für alle } x_j \in V(C_i) \right\}$$

die Menge derjenigen kritischen Variablen x_i, die unter den Variablen in ihrer kritischen Klausel C_i in der Permutation π die letzte sind, und $r(\pi, \alpha) = |R(\pi, \alpha)|$.
Da die Permutation π gleichverteilt gewählt wird, ist jede der k Variablen in C_i mit gleicher Wahrscheinlichkeit $1/k$ die letzte in π. Jede der $j(\alpha)$ kritischen Variablen ist also mit Wahrscheinlichkeit $1/k$ in $R(\pi, \alpha)$, daher ist der erwartete Wert $E[r(\pi, \alpha)] = j(\alpha)/k$.
Mit jeder Variablen $x_i \in R(\pi, \alpha)$ erhöht sich die Wahrscheinlichkeit dafür, dass α bei suche(π) gefunden wird: wenn die anderen Variablen y in C_i zufällig den richtigen Wert $\alpha(y)$ erhalten, dann erhält auch x_i den Wert $\alpha(x_i)$, da zu dem Zeitpunkt, wo der Algorithmus x_i betrachtet, die Klausel C_i Einerklausel ist.
Daher ist die Wahrscheinlichkeit $p_{\alpha,\pi}$ dafür dass α gefunden wird, falls die Permutation π gewählt wurde, mindestens $p_{\alpha,\pi} \geq 2^{-n+r(\pi,\alpha)}$.
Also ist die Wahrscheinlichkeit p_α dafür, dass α gefunden wird, insgesamt

$$\begin{aligned} p_\alpha &= \sum_{\pi \in S_n} \frac{1}{n!} p_{\alpha,\pi} = \sum_{\pi \in S_n} \frac{1}{n!} 2^{-n+r(\pi,\alpha)} \\ &= 2^{-n} \sum_{\pi \in S_n} \frac{1}{n!} 2^{r(\pi,\alpha)} = 2^{-n} \cdot E[2^{r(\pi,\alpha)}] . \end{aligned}$$

Der Erwartungswert $E[2^{r(\pi,\alpha)}]$ ist aber nach der Jensen'schen Ungleichung mindesten $2^{E[r(\pi,\alpha)]}$, also ist

$$p_\alpha \geq 2^{-n+E[r(\pi,\alpha)]} \geq 2^{-n+j(\alpha)/k} .$$

Die Jensen'sche Ungleichung besagt, dass für eine Zufallsvariable X, die jeden der Werte $a_1, \ldots, a_n$ mit Wahrscheinlichkeit $1/n$ annimmt, gilt $E[2^X] \cdot 2^{E[x]}$. Dies rechnet man nach, indem man den Logarithmus zur Basis 2 betrachtet, also zeigt $\log E[2^X] \geq E[X]$. Es ist nämlich

$$\begin{aligned} \log E[2^X] &= \log\Big(\frac{1}{n} \sum_{i=1}^{n} 2^{a_i}\Big) \geq \log\Big(\Big(\prod_{i=1}^{n} 2^{a_i}\Big)^{1/n}\Big) \\ &= \frac{1}{n} \log \prod_{i=1}^{n} 2^{a_i} = \frac{1}{n} \sum_{i=1}^{n} \log 2_i^a = \frac{1}{n} \sum_{i=1}^{n} a_i = E[X] \end{aligned}$$

wobei die erste Ungleichung nach dem Satz vom arithmetischen und geometrischen Mittel gilt.

Die gesamte Erfolgswahrscheinlichkeit lässt sich nun beschränken durch

$$\begin{aligned} p(n) &= \sum_{\alpha\in A} p_\alpha \;\geq\; \sum_{\alpha\in A} 2^{-n+j(\alpha)/k} \\ &= \sum_{\alpha\in A} 2^{-n+n/k+(j(\alpha)-n)/k} \;=\; 2^{-n+n/k} \sum_{\alpha\in A} 2^{-N(\alpha)/k} \\ &\geq\; 2^{-n+n/k} \sum_{\alpha\in A} 2^{-N(\alpha)} \;\geq\; 2^{-n+n/k}\,. \end{aligned}$$

wobei die letzte Ungleichung nach Lemma 19 gilt.

Also ist $p(n) \geq 2^{-n+n/k}$, und somit erhalten wir als Schranke an die Laufzeit $O\left((2^{1-1/k})^n\right)$. Die Basis errechnet sich für kleine Werte von k wie folgt:

k	3	4	5	6	7	8
$2^{1-1/k}$	1.58741	1.68180	1.74111	1.78180	1.81145	1.83401

Dieser Algorithmus war zum Zeitpunkt seiner Veröffentlichung 1997 der erste, der eine bessere Komplexität für k-SAT mit $k \geq 4$ erreicht als der von Monien und Speckenmeyer. Ausserdem war dies der erste theoretisch analysierte probabilistische Algorithmus für k-SAT.

Eine verbesserte Form dieses Algorithmus wurde von Paturi, Pudlák, Saks und Zane etwas später angegeben. Der Algorithmus arbeitet genau wie der hier vorgestellte, aber es wird ein Präprozessor vorgeschaltet, der die Formel um alle Klauseln ergänzt, die aus den Eingabeklauseln durch Resolution erzeugt werden können, und deren Größe durch einen (von k und n abhängigen) Parameter s beschränkt ist. Die Analyse ist zu komplex, um hier präsentiert zu werden.

Dieser verbesserte Algorithmus ist derzeit der beste probabilistische Algorithmus für k-SAT mit $k \geq 4$, er erreicht die Komplexität $O(b_k^n)$ für die folgenden Werte der Basis b_k:

k	3	4	5	6	7	8
b_k	1.36406	1.49579	1.56943	1.63788	1.68753	1.72521