SAT Solving

Skript zur Vorlesung im WS 2018/19
gehalten von Dr. Jan Johannsen

Version vom 3. Dezember 2018

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 $x \in \text{Var}$ 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:

$$\begin{aligned} V(0) &= \emptyset & V(1) &= \emptyset \\ V(x) &= \{x\} & V(\neg F) &= V(F) \\ V(F \wedge G) &= V(F) \cup V(G) & V(F \vee G) &= V(F) \cup V(G) \end{aligned}$$

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

$$\begin{aligned} |0| &= 1 & |1| &= 1 \\ |x| &= 1 & |\neg F| &= |F| + 1 \\ |F \wedge G| &= |F| + |G| + 1 & |F \vee G| &= |F| + |G| + 1 \end{aligned}$$

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, in Zeichen $F \approx G$, 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 unmittelbar vor Variablen vorkommen, also falls F der folgenden induktiven Definition genügt:

- Jedes Literal und jede Konstante 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 $F = 0$, so ist $\bar{F} = 1$. Ist $F = 1$, so ist $\bar{F} = 0$.
- 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 $\bar{F} = \bar{G}_1 \vee \bar{G}_2$.

Anschaulich entsteht $\bar{F}$ aus F daurch, dass jedes Literal durch das entsprechende komplementäre Literal ersetzt wird, und die Symbole 0 und 1 sowie $\wedge$ und $\vee$ vertauscht werden.

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

Beweis durch strukturelle Induktion nach der obigen induktiven Defintion der NNF.

- Ist $F = 0$, so ist $\bar{F} = 1 \equiv \neg 0$. Ist $F = 1$, so ist $\bar{F} = 0 \equiv \neg 1$.
- 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* $n(F)$ *in NNF.*

Beweis durch Induktion nach dem Formelaufbau:

- Ist $F = x$, $F = 0$ oder $F = 1$, so ist F bereits in NNF, also setze $n(F) = F$.
- Ist $F = G \circ H$ für $\circ \in \{\wedge, \vee\}$, so gibt es nach Induktionshypothese $n(G) \equiv G$ und $n(H) \equiv H$ in NNF, und wir setzen $n(F) = n(G) \circ n(H)$ und erhalten offensichtlich $n(F) = n(G) \circ n(H) \equiv G \circ H = F$.
- Ist $F = \neg G$ so gibt es nach Induktionshypothese $n(G) \equiv G$ in NNF. Definiere $n(F) = n(\bar{G})$, nach Prop. 1 gilt dann $n(F) = n(\bar{G}) \equiv \neg n(G) \equiv \neg G = F$.

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 KNF und* $\check{F}$ *in DNF.*

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 $\check{F}$. 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 $\alpha_1, \ldots, \alpha_m$ alle Bewertungen mit $\operatorname{dom} \alpha_j = \{x_1, \ldots, x_n\}$ und $\alpha_j \models F$ für $j = 1, \ldots, m$. Wir definieren dann

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

und müssen zeigen, dass $\check{F} \equiv F$ 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 $\beta \models \check{F}$.

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

Die im obigen Beweis konstruierte Formel $\check{F}$ hat die Größe $O(n2^n)$. Dass diese Konstruktion kann nicht wesentlich verbessert werden kann, zeigt die folgende untere Schranke:

Proposition 4. *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 $G_n = T_1 \vee \ldots \vee T_m$ eine zu F_n äquivalente Formel in DNF. O.b.d.A. können wir annehmen, dass keiner der Terme T_j unerfüllbar ist, ansonsten könnte man diesen einfach weglassen und erhielte eine kleinere 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_j ein Term in G_n, 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_j 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 nämlich 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 $m \geq 2^n$ Terme, von denen jeder mindestens n Literale hat, also ist $|G_n| \geq n2^n$. □

Korollar 5. *Es gibt Formeln F'_n in DNF mit $|F'_n| = O(n)$, derart dass jede zu F_n äquivalente Formel G_n in KNF die Größe $|G_n| \geq n2^n$ hat.*

Beweis. Sei $F'_n := \bar{F}_n$ für die Formeln F_n aus Proposition 4. Diese Formeln sind in DNF, und wenn es Formeln $G_n \equiv F'_n$ in KNF mit $G_n < n2^n$ gäbe, dann wäre auch $\bar{G}_n \equiv F_n$ eine Formel in DNF mit $|\bar{G}_n| = |G_n| < n2^n$, im Widerspruch zu Proposition 4. □

Die *Breite* $w(C)$ einer Klausel $C = a_1 \vee \ldots \vee a_k$ ist k, die Anzahl der Literale in C. Eine Klausel der Breite $w(C) = k$ wird auch als k-Klausel bezeichnet.

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 die Breite $w(C_j) \leq k$ hat.*

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

Proposition 6. *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 $G = C_1 \wedge \ldots \wedge C_m$. O.B.d.A. ist C_1 nicht tautologisch, und 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. □

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 Breite 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 $F \setminus \{C\}$.

1.4 Das Problem SAT und seine Komplexität

Das Entscheidungsproblem FSAT ist definiert wie folgt:

Gegeben:	Eine Formel F der Aussagenlogik.
Frage:	Ist F erfüllbar?

Ohne Beweis zitieren wir das folgende zentrale Resultat:

Satz 7 (Cook 1971)**.** *FSAT ist NP-vollständig.*

Satz 8. *Für jede Formel F gibt es eine erfüllbarkeits-äquivalente Formel $F' \approx F$ in 3-KNF mit $|F'| = O(|F|)$. Weiter ist F' aus F in polynomieller Zeit berechenbar.*

Beweis. Wir definieren für jede Teilformel G von F eine Variable x_G und eine Menge F_G von Klauseln wie folgt:

- Ist G eine Variable x, so ist $x_G = x$ und $F_G = \emptyset$.
- Ist $G = H \vee H'$, so ist x_G eine neue Variable, und F_G enthält Klauseln, die ausdrücken, dass $x_G \leftrightarrow (x_H \vee x_{H'})$ ist, nämlich $(\neg x_G \vee x_H \vee x_{H'})$, $(\neg x_H \vee x_G)$ und $(\neg x_{H'} \vee x_G)$).
- Ist $G = H \wedge H'$, so ist x_G eine neue Variable, und F_G enthält die Klauseln $(\neg x_H \vee \neg x_{H'} \vee x_G)$, $(\neg x_G \vee x_H)$ und $(\neg x_G \vee x_{H'})$).
- Ist $G = \neg H$, so ist x_G eine neue Variable, und F_G enthält die Klauseln $(x_G \vee x_H)$ und $(\neg x_G \vee \neg x_H)$.

Die Formel F' ist nun definiert als $\bigwedge_G F_G \wedge x_F$. Es bleibt zu zeigen, dass $F' \approx F$ ist.

Sei also $\alpha \models F'$. Durch Induktion nach dem Aufbau von F zeigen wir, dass für jede Teilformel G von F gilt $\alpha(x_G) = G\alpha$. Insbesondere ist also $F\alpha = \alpha(x_F)$, und wegen des letzten Konjunktionsglieds muss $\alpha(x_F) = 1$ sein, somit gilt $\alpha \models F$.

Der Induktionsanfang $G = x$ ist trivial. Sei also z.B. $G = H \vee H'$. Nach Induktionshypothese ist $\alpha(x_H) = H\alpha$ und $\alpha(x_{H'}) = H'\alpha$. Die Klauseln F_G erzwingen nun, dass $\alpha(x_G) = \alpha(x_H) \vee \alpha(x_{H'})$ ist, also folgt $\alpha(x_G) = H\alpha \vee H'\alpha = G\alpha$. Die übrigen Fälle sind analog.

Sei umgekehrt $\alpha \models F$. Dann wird eine Bewertung $\alpha' \supseteq \alpha$ definiert durch $\alpha'(x_G) = G\alpha$ für jede Teilformel G von F. Dann rechnet man leicht nach, dass $\alpha' \models F_G$ für alle Formeln F_G gilt. Da $F\alpha = 1$ ist, gilt auch $\alpha'(x_F) = 1$, also ist insgesamt $F'\alpha' = 1$, also $\alpha' \models F'$. □

Das Entscheidungsproblem SAT ist der folgende Spezialfall von FSAT:

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

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

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

Aus Satz 8 folgt unmittelbar, dass die Erfüllbarkeit von Formeln in 3-KNF ebenso schwierig ist wie für allgemeine Formeln.

Korollar 9. *3-SAT ist NP-vollständig. Daher sind auch SAT und* k*-SAT für jedes* $k \geq 3$ *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, wie Prop. 4 zeigt.

1.5 Beschränkte Vorkommen

Sei $\mathcal{F}$ eine Klasse von Formeln in KNF. Eine Formel $F \in \mathcal{F}$ ist in $\mathcal{F}(l)$ für $l \in \mathbb{N}$, wenn jede Variable in F höchstens l mal vorkommt. Das Problem SAT(l) ist das Erfüllbarkeitsproblem SAT für Formeln in KNF(l).

Satz 10. *Für jede Formel* F *in KNF gibt es eine erfüllbarkeits-äquivalente Formel* $D(F)$ *in KNF(3) mit* $w(D(F)) = w(F)$.

Beweis. Wir geben eine polynomielle Reduktion von SAT auf SAT(3) an. Sei F eine Formel in KNF. Für jede Variable x, die in F ℓ mal vorkommt, wählen wir ℓ neue Variablen $x_1, \ldots, x_\ell$, und ersetzen das i-te Vorkommen von x in F durch x_i, für $1 \leq i \leq \ell$. Das Ergebnis dieser Ersetzungen sei F'. Damit F' zu F erfüllbarkeits-äquivalent wird, müssen wir erzwingen, dass die Variablen $x_1, \ldots, x_\ell$ immer gleich bewertet werden. Dazu fügen wir für jede Variable $x \in V(F)$ die Klauseln E_x hinzu:

$$E_x := (\bar{x}_1 \vee x_2) \wedge \ldots \wedge (\bar{x}_{ell-1} \vee x_\ell) \wedge (\bar{x}_\ell \vee x_1)$$

Definiere nun die Formel $D(F) := F' \wedge \bigwedge_{x \in V(F) E_x}$. Offenbar ist $D(F)$ in KNF(3), denn jede Variable x_i kommt in $D(F)$ dreimal vor: einmal in F', und zweimal in E_x.

Jede Bewertung $\alpha \models F$ lässt sich nun zu einer $\alpha' \models D(F)$ umbauen, indem man für jede Variable x setzt $\alpha'(x_i) = \alpha(x)$ für $1 \leq i \leq \ell$.
Umgekehrt ist $\beta \models D(F)$, so müssen die Variabeln x_i für ein $x \in V(F)$ alle den gleichen Wert bekommen, da $\beta \models E_x$. Also definiere $\beta'(x) := \beta(x_1)$, dann folgt $\beta' \models F$, da ja $\beta \models F'$. □

Die Konstruktion ist eine polynomielle Reduktion, also folgt:

Korollar 11. *3-SAT(3) ist NP-vollständig.*

Die Reduktion im obigen Beweis erzeugt Klauseln der Breite 2. Die folgenden Überlegungen zeigen, dass dies notwendig ist.
Eine Formel in KNF ist in E3-KNF, wenn jede Klausel genau die Breite k hat. Ek-SAT bezeichnet das Erfüllbarkeitsproblem für solche Formeln.

Satz 12. *E3-SAT(3) ist trivial, d.h., jede Formel in E3-KNF(3) ist erfüllbar.*

Sei $G = (U, V, E)$ ein bipartiter Graph mit $E \subseteq U \times V$. Eine Menge $M = \{(u_1, v_1), \ldots, (u_m, v_m)\}$ ist ein *Matching* in G, wenn die Kanten in M keine gemeinsamen Endpunkte haben, also wenn sowohl die u_i als auch die v_i paarweise verschieden sind. Ein Matching in G ist perfekt, wenn jeder Knoten in U mit einer Kante in M inzidiert, also wenn $\{u_1, \ldots, u_m\} = U$ ist. Ein perfektes Matching ordnet also jedem Knoten in U eindeutig einen Nachbarn in V zu.
Halls Theorem, auch bekannt als *Heiratssatz*, ist ein notwediges und hinreichendes Kriterium fr die Existenz eines perfekten Matchings in G.
Für eine Teilmenge $S \subseteq U$ ist die Nachbarschaft $N(S) \subseteq V$ die Menge derjenigen $v \in V$, für die es eine Kante (u, v) zu einem $u \in S$ gibt.

Satz 13 (Halls Theorem). *Für einen bipartiten Graphen* $G = (U, V, E)$ *sind äquivalent:*

- *Es gibt ein perfektes Matching in* G,
- *Für jede Teilmenge* $S \subseteq U$ *ist* $|N(S)| \geq |S|$.

Beweis von Theorem 12. Für eine Formel $F = C_1 \wedge \ldots \wedge C_m$ ist der *Inzidenzgraph* der folgende bipartite Graph $G(F) = (U, V, E)$, wobei $U = \{C_1, \ldots, C_m\}$ die Klauseln in F sind, un $dV = V(F)$ die Variablen in F sind. Zwischen Klausel C und einer Variablen x gibt es eine Kante $(C, x) \in E$, falls x in C vorkommt.
Sei nun $L \subseteq U$ eine Menge von Klauseln der Gröe $s = |L|$, und $N(L) = V(L) \subseteq V$ die Menge ihrer Nachbarn, mit $t = |N(L)|$. Es bezeichne e die

Anzahl der Kanten zwischen L und $N(L)$. Da F in E3-KNF ist, gilt $e = 3s$, und da F in KNF(3) ist, gilt $e \leq 3t$. Also ist $3t \geq 3s$ und somit $t \geq s$.

Nach Halls Theorem gibt es also ein perfektes Matching in $G(F)$, d.h., jeder Klausel C können wir eindeutig eine Variable $x_C \in V(C)$ zuordnen, deren Wert so gesetzt werden kann, dass C erfüllt ist. □

Auf der anderen Seite gilt:

Satz 14. E3-*SAT(4) ist NP-vollständig.*

Beweis. Für jede Formel F in E3-KNF definieren wir eine Formel $D'(F)$ in E3-KNF(4), die genau dann erfüllbar ist, wenn F erfüllbar ist. Die Konstruktion basiert auf der im Beweis von Satz 10. Die Formel F' ist genau so definiert wie dort, aber statt der Formeln E_x, die die Äquivalenz der Kopien $x_1, \ldots, x_\ell$ einer Variable x erzwingen und aus Klauseln der Breite 2 bestehen, konstruieren wir Formeln E'_x mit dem selben Zweck.

Für jede Variable x werden ℓ neue Variablen $y_1, \ldots, y_\ell$ eingeführt, und die Klauseln in E_x ersetzt durch

$$E_x := (\bar{x}_1 \vee x_2 \vee y_1) \wedge \ldots \wedge (\bar{x}_{\ell-1} \vee x_\ell \vee y_{\ell-1}) \wedge (\bar{x}_\ell \vee x_1 \vee y_\ell)$$

Für jedes $i \leq \ell$ werden nun weitere Klauseln hinzugefügt, die erzwingen, dass die Variable y_i in jeder erfüllenden Bewertung mit 0 bewertet wird. Dann erfüllen die Klauseln E'_x denselben Zweck wie E_x im Beweis von Satz 10.

Dafür werden für jedes $i \leq \ell$ noch vier neue Variablen a_i, b_i, c_i, d_i eingeführt, und dazu die sechs Klauseln

$$\begin{array}{lll} \bar{y}_i \vee a_i \vee \bar{b}_i & a_i \vee b_i \vee \bar{c}_i & b_i \vee c_i \vee \bar{d}_i \\ b_i \vee c_i \vee \bar{d}_i & \bar{y}_i \vee \bar{a}_i \vee \bar{d}_i & \bar{y}_i \vee \bar{a}_i \vee d_i \end{array}$$

Man überprüft leicht, dass diese sechs Klauseln unerfüllbar sind, wenn y_i mit 1 bewretet wird. Ausserdem hat jede Klausel die Breite 3, und es kommt jede Variable höchstens 4 mal vor, also ist die konstruierte Formel $D'(F)$ in E3-KNF(4).

Offenbar ist $D'(F)$ aus F in polynomieller Zeit konstruierbar, also ist dies eine polynomielle Reduktion von E3-SAT auf E3-SAT(4). □

Der Beweis von Theorem 12 ist leicht für beliebige Breiten k zu verallgemeinern: für jedes $k \geq 2$ ist Ek-SAT(k) trivial. Dagegen verallgemeinert sich Theorem 14 nicht direkt, Ek-SAT($k+1$) ist nicht für jedes k schwierig. Allerdings gilt, dass der abrupte Sprung von trivial zu NP-schwer für jedes k bei einer gewissen Anzahl s von Vorkommen auftritt:

Satz 15. *Für jedes* $k \geq 3$ *gibt es ein* $s = s(k) \geq k$ *mit:*

- Ek-*SAT(s) ist trivial,*
- Ek-*SAT(s + 1) ist NP-vollständig.*

Der genaue Wert von $s(k)$ ist nur für kleine Werte von k bekannt. Für hinreichend große k gelten die Abschätzungen $2^k/ek \leq s(k) < 4 \cdot 2^k/k$. Die besten bekannten konkreten unteren und oberen Schranken für kleine Werte von k sind in der folgenden Tabelle zusammengefasst:

k	$s(k) \geq$	$s(k) <$
3	3	4
4	4	5
5	5	8
6	7	12
7	13	18
8	24	30
9	41	52

Wegen des abrupten Sprunges von trivial zu NP-schwer reicht es z.B. zur Verbesserung der oberen Schranke in der dritten Zeile, eine einzige unerfüllbare Formel in (E5)-KNF mit höchstens 7 Vorkommen jeder Variablen zu finden.

2 Einfache Spezialflle

Wir betrachten einige Spezialfälle von Formeln, für die es effiziente Algorithmen für das Erfüllbarkeitsproblem gibt. Eine Klasse $\mathcal{F}$ von Formeln ist ein einfacher Spezialfall von SAT, wenn die folgenden Bedingungen gelten:

- Es ist in polynomieller Zeit entscheidbar, ob eine Formel F in $\mathcal{F}$ liegt.
- Für Formeln $F \in \mathcal{F}$ ist die Erfüllbarkeit in polynomieller Zeit entscheidbar.

Wir betrachten im Folgenden einige klassische einfaceh Spezialfälle.

Horn-Formeln

Definition. *Eine Klausel heißt*

- positiv *(*negativ*), wenn sie nur positive (negative) Literale enthält,*
- 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 16. *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)
  α := []
  while Fα enthält eine positive 1-Klausel
      wähle eine solche x
      α := α ∪ [x ← 1]
  if α' ⊨ 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 1-Klausel x_1 in $F\alpha$ vor, und somit setzt der Algorithmus $\alpha(x_1) = 1$. Dann gilt aber auch $\alpha' \models C$. Es bezeichne nun F_D die Menge der definiten Klauseln in F.

Wir zeigen nun, dass jede Bewertung $\beta \models F_D$ eine Erweiterung $\beta \supseteq \alpha$ 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 1-Klausel 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 $\beta \models F_D$ auch $\beta(C) = 0$. Damit ist F unerfüllbar.

Offenbar ist die Zahl der Schleifendurchläufe durch n beschränkt, und in jedem Schleifendurchlauf wird nach einer positiven 1-Klausel 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-SAT

Im Kontrast zu Korollar 9 ist das Problem 2-SAT einfach. Für die Konstruktion benötigen wir einige Begriffe und Algorithmen für gerichtete Graphen.

Sei $G = (V, E)$ ein gerichteter Graph mit $E \subseteq V \times V$. Ein Knoten v ist von u erreichbar, wenn es einen gerichteten Weg von u nach v gibt, also eine Folge von Knoten $v_1, \ldots, v_\ell$ mit $u = v_1$ und $v = v_\ell$ und $(v_i, v_{i+1}) \in E$ für $1 \leq i < \ell$.

Zwischen zwei Knoten $u, v \in V$ wird die Relation $u \sim v$ definiert durch: v ist von u erreichbar, und u ist von v erreichbar, d.h., u und v liegen auf einem Kreis. Diese Relation ist offenbar eine Äquivalenzrelation, die Äquivalenzklassen dieser Relation heißen *starke Zusammenhangskomponenten*. Die Äquivalenzklasse, die den Knoten $v \in V$ enthält, wird mit $[v]$ bezeichnet.

Es gibt einen Algorithmus, der in linearer Zeit (in der Anzahl der Knoten und Kanten) einen gerichteten Graphen in seine starken Zusammenhangskomponenten zerlegt.

Der Faktorgraph $G/\sim$ ist der Graph, dessen Knoten die Zusammenhangskomponenten $\{[v]; v \in V\}$ von G sind, mit einer Kante zwischen $[u]$ und $[v]$, wenn es $u' \in [u]$ und $v' \in [v]$ gibt mit $(u', v') \in E$. Dieser Graph ist also ein gerichteter, azyklischer Graph.

Für einen solchen gerichten, azyklischen Graphen gibt es eine *topologische Ordnung*, d.h. eine Anordnung $v_1, \ldots, v_n$ der Knoten so dass alle Kanten in der Ordnung nach rechts gehen: ist also $(v_i, v_j) \in E$, so ist $i < j$. Eine solche topologische Ordnung kann in linearer Zeit berechnet werden.

Satz 17. *2-SAT ist in linearer Zeit* $O(n + m)$ *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 1-Klausel 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
        $\alpha(b) := 1$ für $b \in [a_j]$
        $\alpha(b) := 0$ für $b \in [\bar{a}_j]$

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 $O(n + m)$ bestimmt und topologisch angeordnet werden können, ist diese Konstruktion und der Test auf Erfüllbarkeit in linearer Zeit $O(m)$ 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).

SAT(2)

Analog zum vorherigem Abschnitt steht das folgende Ergebnis im Kontrast zu Korollar 11:

Satz 18. *SAT(2) ist in linearer Zeit* $O(n)$ *entscheidbar.*

Beweis. Der folgende sehr einfache Algorithmus entscheidet SAT(2):

while in F gibt es eine 1-Klausel a
    $F := F[a \leftarrow 1]$
if $F = 0$
    then return UNSAT
    else return SAT

Nach Verlassen der Schleife enthält F keine 1-Klausel mehr. Enthält F nun eine leere Klausel, ist also $F = 0$, so ist sie offenichtlich unerfüllbar. Andernfalls hat jede Klausel C in F die Breite $w(C) \geq 2$, und jede Variable

kommt höchstens zweimal vor. Daher ist F erfüllbar, wie dasselbe Matching-Argument wie für die Erfüllbarkeit von Formeln in E3-KNF(3), Satz 12, zeigt.

Ein anderer Ansatz zur Entscheidung von SAT(2) ist in gewissem Sinne dual zu dem Algorithmus für 2-SAT.

Ein *markierter Graph* $G = (V, E, M)$ ist ein ungerichteter Multigraph (V, E) (es kann also mehrere Kanten zwischen zwei Knoten geben) mit einer ausgezeichneten Teilmenge $M \subseteq V$ vom *markierten* Knoten. Eine Zusammenhangskomponente von G heiße *markiert*, falls sie mindestens einen markierten Knoten enthält.

Für eine Formel F in KNF(2) definiere einen markierten Graphen $G(F)$ wie folgt:

- $G(F)$ hat m Knoten, einen v_C für jede Klausel C in F.
- Falls die Klauseln C und D komplementäre Literale x und $\bar{x}$ enthalten, so gibt es eine Kante e_x zwischen v_C und v_D.
- Enthält C ein Literal a, derart dass $\bar{a}$ in F nicht vorkommt (ein *reines* Literal), so ist v_C markiert.

Man beachte, dass es zwischen v_C und v_D mehrere Kanten geben kann, falls C und D mehrere Paare komplementärer Literale enthalten, z.B. $C = (x \vee \bar{y})$ und $D = (\bar{x} \vee y \vee z)$.

Eine Bewertung der Variablen in F kann man nun so auffassen, dass sie den Kanten in $G(F)$ eine Richtung gibt: die Kante e_x wird so gerichtet, dass sie von derjenigen Klausel, die das mit 1 bewertete Literal aus $x, \bar{x}$ enthält, zu der Klausel, die das mit 0 bewertete Literal enthält, zeigt. Eine Klausel C ist dann erfüllt, wenn in dieser Ausrichtung v_C eine ausgehende Kante hat. Da Klauseln mit reinen Literalen stets erfüllt werden können, gilt die folgende offensichtliche Behauptung:

Proposition 19. *Eine Formel F in KNF(2) ist genau dann erfüllbar, genau dann wenn die Kanten in $G(F)$ so gerichtet werden können, dass es keine unmarkierte Senke gibt.*

Dies führt uns zum folgenden Lemma:

Lemma 20. *Eine Formel F in KNF(2) ist genau dann erfüllbar, wenn jede Zusammenhangskomponente in $G(F)$ markiert ist oder einen Kreis enthält.*

Beweis. Es reicht zu zeigen, dass die Bedingung des Lemmas äquivalent zu der Bedingung in Proposition 19 ist. Sie ist offensichtlich notwendig, da in

einem endlichen gerichteten Graphen jeder Weg in einer Senke oder einem Kreis enden muss.
Ferner ist die Bedingung auch hinreichend: wir konstruieren eine passende Ausrichtung der Kanten für jede Zusammenhangskomponente, die entweder markiert ist oder einen Kreis enthält.
In einer markierten Komponente wird eine Ausrichtung wie folgt vorgenommen:

- Betrachte einen Spannbaum T der Komponente.
- Jede Kante in T wird so gerichtet, dass sie von dem von v weiter entfernten zum näher an v liegenden Knoten führt.

Die übrigen Kanten können dann beliebig gerichtet werden, da jeder Knoten außer der Wurzel v bereits eine ausgehende Kante in T hat.
In einer Komponente K , die einen Kreis $v_1, \ldots, v_k, v_{k+1} = v_1$ enthält, wird eine Ausrichtung wie folgt vorgenommen:

- Zuerst werden die Kanten in diesem Kreis um diesen herum gerichtet, also von v_1 zu v_2, $\ldots$, von v_{k-1} zu v_k und von v_k zu v_1.
- Sei E' die Menge der Kanten in K ohne diejenigen, die die Knoten $v_1, \ldots, v_k$ verbinden (also ohne den Kreis und seine Sehnen).
- Für jedes $1 \leq i \leq k$ sei K_i die Menge der Knoten, die von v_i, aber keinem v_j für $j < i$ aus in E' erreichbar sind.
- Für jedes i sei T_i ein Spannbaum des induzierten Graphen auf K_i. Die Kanten in T_i werden wie oben auf die Wurzel v_i hin gerichtet wie oben.

Auf diese Weise erhält jeder Knoten in K eine ausgehende Kante. Alle übrigen Kanten können dann beliebig gerichtet werden. □

Die Bedingung in Lemma 20 läßt sich leicht mittels Tiefensuche überprüfen, etwa mit dem folgenden Algorithmus:

```
solange noch unarkierte Knoten in G
    wähle den kleinsten unmarkierten Knoten v
    führe Tiefensuche ab v durch
    falls keinen markierten Knoten und keine rückläufige Kante gefunden
        return FALSE
return TRUE
```

Daher kann man die Erfüllbarkeit in Zeit $O(n + m)$ testen, dies ist von der Größenordnung $O(n)$, da $m \leq 2n$ ist. □

Tatsächlich liegt das Problem SAT(2) sogar in der Komplexitätsklasse L, kann also von einem deterministischen Algorithmus mit logarithmischem Speicherplatzbedarf gelöst werden, und ist auch vollständig für L (wiederum unter einem geeigneten schwachen Reduktionsbegriff).

Horn-umbenennbare Formeln

Eine *Umbenennung* von F ist eine Permutation r der Menge der Literale von F mit $r(a) \in \{a, \bar{a}\}$ für jedes Literal a. Eine Umbenennung ändert also nur die Vorzeichen mancher Literale.

Eine Formel F ist Horn-umbenennbar, wenn es eine Umbenennung r von F gibt, so dass $r(F)$ eine Horn-Formel ist.

Satz 21. *Es gibt einen Algorithmus, der in Zeit* $O(n^2 m)$ *entscheidet, ob eine Formel* F *Horn-umbenennbar ist und ggf. ein Umbenennung* r *berechnet, für die* $r(F)$ *eine Horn-Formel ist.*

Beweis. Definiere für F die folgende Formel F^* in 2-KNF:

$$F^* := \{\, a \vee b \;;\; \text{es gibt } C \in F \text{ mit } a \in C \text{ und } b \in C \,\}$$

Es gilt, dass F^* genau dann erüllbar ist, wenn F Horn-umbenennbar ist, und aus einer Bewertung, die F^* erfüllt, gewinnt man eine Umbenennung die dies bezeugt.

Sei also $\alpha \models F^*$, daraus definieren wir die Umbenennung r_α mit

$$r_\alpha(x) = \begin{cases} \bar{x} & \text{falls } \alpha(x) = 1 \\ x & \text{sonst.} \end{cases}$$

Nun gilt: Ist $r_\alpha(a)$ positiv, so ist $\alpha(a) = 0$. Denn ist a positiv, so ist $r_\alpha(a) = a$, also $\alpha(a) = 0$. Ist dagegen $a = \bar{x}$, so ist $r_\alpha(a) = x = \bar{a}$, also ist $\alpha(x) = 1$ und somit $\alpha(a) = 0$.

Es folgt dass $r_\alpha(F)$ eine Horn-Formel ist. Denn andernfalls sei $C = a_1 \vee \ldots \vee a_k$ eine Klausel, für die $r_\alpha(C)$ keine Horn-Klausel ist. Dann gibt es Literale a_i und a_j in C, so dass $r_\alpha(a_i)$ und $r_\alpha(a_j)$ positiv sind, also ist $\alpha(a_i) = \alpha(a_j) = 0$, und somit ist die Klausel $a_i \vee a_j$ in F^* von α nicht erfüllt, im Widerspruch zur Voraussetzung.

Sei umgekehrt r eine Umbenennung, für die $r(F)$ eine Horn-Formel ist. Daraus definieren wir die Bewertung α_r mit

$$\alpha_r(x) = \begin{cases} 1 & \text{falls } r(x) = \bar{x} \\ 0 & \text{sonst.} \end{cases}$$

Nun gilt für jedes Literal a: ist $\alpha_r(a) = 0$, so ist $r(a)$ positiv. Denn ist $a = x$, so ist $\alpha_r(a) = \alpha_r(x) = 0$, also ist $r(x) = x$. Ist dagegen $a = \bar{x}$, so ist $\alpha(x) = 1$, also ist $r(a) = \bar{a} = x$.

Es folgt dass $\alpha_r \models F^*$. Denn andernfalls sei $a \vee b$ eine unerfüllte Klausel in F^*, dann ist $\alpha_r(a) = \alpha_r(b) = 0$. Dann gibt es aber eine Klausel $C \in F$ mit $(a \vee b) \subseteq C$, und $r(a)$ und $r(b)$ sind beide positiv, also ist $r(C)$ keine Horn-Klausel, im Widerspruch zur Voraussetzung.

Der Algorithmus konstruiert nun aus F die Formel F^*, die hat die Größe $O(n^2m)$. Dann löst er diese mit dem Linearzeit-Algorithmus für 2-SAT und gewinnt im Falle der Erfüllbarkeit die Umbenennung nach der obigen Definition. □

Ein Algorithmus für Horn-umbenennbare Formeln folgt daraus wie folgt:

- Berechne eine Umbenennung r so dass $r(F)$ eine Horn-Formel ist.
- Löse $r(F)$ mit dem Algorithmus für Horn-Formeln.
- Berechne ggf. aus der erfüllenden Bewertung für $r(F)$ eine für F durch Umkehrung der Umbenennung.

Die Laufzeit ist die zur Berechnung der Umbenennung $O(n^2m)$, da dies der aufwändigste Schritt ist.

Cluster-Formeln

Eine Formel F in KNF ist eine *Hitting*-Formel, wenn je zwei Klauseln in F komplementäre Literale enthalten, d.h., für je zwei $C, D \in F$ gibt es ein Literal a so dass $a \in C$ und $\bar{a} \in D$ ist.

Eine *Cluster*-Formel ist eine Konjunktion $F = H_1 \wedge \ldots \wedge H_\ell$ von Hitting-Formeln mit disjunkten Variablen, d.h. für $i \neq j$ ist $V(H_i) \cap V(H_j) = \emptyset$.

Die Erfüllbarkeit von Hitting-Formeln lässt sich leicht arithmetisch testen:

Satz 22. *Sei $F = C_1 \wedge \ldots \wedge C_m$ eine Hitting-Formel. Dann ist F genau dann erfüllbar, wenn gilt:*

$$\sum_{i=1}^{m} 2^{-w(C_i)} < 1 \,.$$

Beweis. Für eine Klausel $C_i \in F$ sei A_i die Menge der totalen Bewertungen α mit $C_i\alpha = 0$.

Behauptung: Für $i \neq j$ ist $A_i \cap A_j = \emptyset$.

Sei $\alpha \in A_i$. Ist $C_i\alpha = 0$, so ist $\alpha(a) = 0$ für alle $a \in C_i$. Nun ist aber $\bar{a} \in C_j$ für ein $a \in C_i$, also ist $C_j\alpha = 1$ und somit $\alpha \notin C_j$.

Sei ferner A die Menge der totalen Bewertungen α mit $\alpha \not\models F$. Also ist F genau dann erfüllbar, wenn $|A| < 2^n$.

Wegen der Disjunktheit ist $|A| = \sum_{i=1}^{m} |A_i|$, also ist F genau dann erfüllbar, wenn $\sum_{i=1}^{m} |A_i| < 2^n$ ist. Da $|A_i| = 2^{n-w(C_i)}$ ist, ist dies genau dann der Fall, wenn $\sum_{i=1}^{m} 2^{n-w(C_i)} < 2^n$ ist, also wenn $\sum_{i=1}^{m} 2^{-w(C_i)} < 1$ ist. □

Da eine Cluster-Formel $F = H_1 \wedge \ldots \wedge H_\ell$ erfüllbar ist, wenn alle H_i erfüllbar sind, ist diese arithmetische Bedingung für alle H_i zu überprüfen.

3 DPLL-Algorithmen

Offensichtlich ist SAT trivialerweise in Zeit $2^n \cdot O(|F|)$ 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 DPLL-Algorithmen, nach den Autoren Davis, Putnam, Logemann und Loveland, die das erste Verfahren dieser Art 1960-1962 beschrieben haben.

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

```
DPLL(F, α)
  if Fα = 0
      then return UNSAT
  if Fα = 1
      then return α
  wähle x ∈ V(Fα)
  β := DPLL(F, α ∪ [x ← 0])
  if β ≠ UNSAT
      then return β
      else  return DPLL(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 $C\alpha = 0$ für eine Klausel C in F ist, 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 DPLL nur polynomiale Zeit benötigt, ist die Laufzeit beschränkt durch $T(n) \cdot n^{O(1)}$, wobei $T(n)$ die Größe des Rekursionsbaumes bezeichnet. Der polynomielle 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 echt verzweigenden Baum die Größe durch ein konstantes Vielfaches dieser Anzahl beschränkt ist.

Man erhält so bereits eine bessere Komplexitätsschranke für k-SAT, wenn bei DPLL 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

3.1 Reduktionen

1-Klauseln

Eine besondere Rolle bei DPLL-Algorithmen spielen 1-Klauseln (engl. *unit clause*): wird nach einer Variablen x verzweigt, die in einer 1-Klausel 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 1-Klauseln in den meisten DPLL-Algorithmen bevorzugt zuerst behandelt, dies wird als *unit propagation* bezeichnet.

```
UnitProp(F, α)
  while in F gibt es eine 1-Klausel a
    F := F[a ← 1]
    α := α ∪ [a ← 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 DPLL-Algorithmus läßt sich also treffender wie folgt beschreiben:

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

Ein DPLL-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 $x \in \text{dom}\,\alpha \setminus V(a)$ erfüllt. Man kann also reine Literale gleich mit 1 bewerten, was die folgende Reduktion rechtfertigt:

PureLit(F, α)
  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 23. *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 DPLL-Algorithmus, wird aber heute wegen des hohen Aufwandes meist nicht mehr verwendet.

3.2 Algorithmus von Monien-Speckenmeyer

Eine mögliche Verzweigungsstrategie 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 DPLL-Algorithmus mit dieser Verzweigungsstrategie auch äquivalenterweise wie folgt schreiben:

simple-MS(F, α)
  if $F = 0$ then return UNSAT
  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 $\beta \neq$ UNSAT then return β
  return UNSAT

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 24. *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 UNSAT
  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 ≠ UNSAT then return β
  return UNSAT
```

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:

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

Wir zeigen nun, dass für alle n gilt

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

also wird in den obigen Gleichungen das Maximum stets im zweiten Term angenommen. Wir setzen für $T(n)$ und $T'(n)$ die Definition ein, und definieren $t_1 := T'(n-1) + \ldots + T'(n-k)$ und $t_2 := T'(n-1) + \ldots + T'(n-k+1)$. Dann ist zu zeigen

$$\max(T(n-1), t_1) \leq \max(T(n-1), t_2) + T'(n-1) \qquad (*)$$

Da $t_1 = t_2 + T'(n-k)$ ist, gilt offenbar $t_2 \leq t_1$.

Falls nun $T(n-1) \geq t_1$ ist, gilt auch $T(n-1) \geq t_2$, also wird auf beiden Seiten das Maximum im ersten Term angenommen, daher gilt $(*)$, da $T(n-1) \leq T(n-1) + T'(n-1)$ ist.

Andernfalls ist $T(n) = \max(T(n-1), t_1) = t_1$, und es gilt

$$\begin{aligned} t_1 &= t_2 + T'(n-k) \\ &\leq \max(T(n-1), t_2) + T'(n-k) \\ &\leq T'(n) + T'(n-2) + \ldots + T'(n-k) \\ &\leq T'(n) + \max(T(n-2), T'(n-2) + \ldots + T'(n-k)) \\ &= T'(n) + T'(n-1) \end{aligned}$$

also gilt $(*)$ auch in diesem Fall.

Die Rekursionsgleichungen vereinfachen sich somit zu

$$\begin{aligned} T(n) &= T'(n-1) + \ldots + T'(n-k) \\ T'(n) &= T'(n-1) + \ldots + T'(n-k+1) \end{aligned}$$

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 $T(n) \leq b_k^{n-1} + \ldots + b_k^{n-k} = O({b_k}^n)$, 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.

3.3 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 25. *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 DPLL-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 26. *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 DPLL-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

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

3.4 Der Algorithmus von Zhang

Bei der Analyse des Algorithmus von Monien-Speckenmeyer für 3-SAT hat sich gezeigt, dass es für die Komplexität vorteilhaft ist, wenn in der Formel 1-Klauseln oder 2-Klauseln vorkommen.
Zur Beschreibung und Analyse des Algorithmus wird die Formel F stets partitioniert in $F = U \cup D \cup T \cup F'$ wie folgt:

- U ist die Menge der 1-Klauseln in F
- D ist eine maximale Menge von 2-Klauseln, die mit den Klauseln in U und untereinander keine gemeinsamen Variablen haben.
- T ist die Menge der übrigen 2-Klauseln.

Wir definieren daraus die Parameter $u := |U|$, $d := |D|$ und $t := \max(|T|, 2)$.
Der Algorithmus von Zhang nutzt den Vorteil durch die Existenz kurzer Klauseln aus, indem er sicherstellt, dass stets kurze Klauseln, also 1- oder 2-Klauseln vorhanden sind. Dabei nimmt man o.E. an, dass die Eingabeklausel F bereits mindestens eine kurze Klausel enthält.
Um dies zu ermöglichen, wird auf Reduktionen, die unkontrolliert kurze Klauseln eliminieren (wie z.B. das Beseitigen reiner Literale), verzichtet. Die einzigen verwendeten Reduktionen sind:

- Teste, ob komplementäre 1-Klauseln a und $\bar{a}$ vorhanden sind, in diesem Fall gib UNSAT zurück.

- Lösche subsumierte 3-Klauseln.
- Falls $u \geq 2$ oder $u = 1$ und $d > 0$ ist, wähle eine 1-Klausel 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, und durch die 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.*

Das heißt in anderen Worten, die Bewertung $[a \leftarrow 1]$ ist quasi-autark, wenn a ein fast reines Literal ist, also $\bar{a}$ nur einmal in F vorkommt.

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

```
Zh(F, α)
  (F, α) := reduziere(F, α)
  if F = 0 then return UNSAT
  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 ≠ UNSAT
       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 1-Klausel 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 $\beta \neq$ UNSAT
  then return β
  else return $Zh(F\gamma, \alpha \cup \gamma)$

Sind keine 1-Klauseln 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 1-Klausel in der Formel entsteht.

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

Zur Behandlung quasi-autarker Bewertungen benötigen wir die folgenden zwei Lemmas. 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 27. *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 28. *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 27 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 $F\langle \bar{a} := c\rangle[b \leftarrow 0, a \leftarrow 1]$. □

Lemma 29. *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. □

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 in den Fällen 1.2 oder 2 auftreten kann. Wäre eine 2-Klausel $(\bar{a} \vee b)$ in F vorhanden, so wäre aber Fall 1.1 eingetreten.

Nach Korollar 28 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 29 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 $\gamma_1 := [c \leftarrow 1, a \leftarrow 1, b \leftarrow 0]$
      $\gamma_2 := [c \leftarrow 0, a \leftarrow 0, b \leftarrow 0]$ (A)
      $\gamma_3 := [c \leftarrow 0, a \leftarrow 1, b \leftarrow 1]$
      if es gibt 2-Klausel $(\bar{b} \vee d)$ mit $d \notin \{a, \bar{a}, c, \bar{c}\}$
        then $\gamma_3 := \gamma_3 \cup [d \leftarrow 1]$ (B)
      if γ_i autark für F für ein $i = 1, \ldots, 3$
        then return aut(F, γ_i, α)
      for $i := 1$ to 3
        $\beta := Zh(F\gamma_i, \alpha \cup \gamma_i)$
        if $\beta \neq$ UNSAT then return β
      return UNSAT
  $\beta := Zh(F[b \leftarrow 1, a \leftarrow 1], \alpha \cup [b \leftarrow 1, a \leftarrow 1])$ (C)
  if $\beta \neq$ UNSAT
    then return β
    else return $Zh(F\langle a := c\rangle[b \leftarrow 0, a \leftarrow 0], \alpha \cup [b \leftarrow 0, a \leftarrow 0])$

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

  Im zweiten Zweig verschwindet eine 1-Klausel, 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 $O(1.570214^n)$.

Satz 30. *Zhangs Algorithmus löst 3-SAT in Zeit* $O(1.570214^n)$.

Ähnliche, noch komplexere Algorithmen wurden angegeben, von Kullmann mit einer Laufzeit $O(1.5045.)$ und Schiermeyer mit einer Laufzeit von $O(1.4963^n)$.