SAT Solving

Prof. Dr. Uwe Schöning

Mitschrift von
Dipl.-Inf. Oliver Gableske

Institut für Theoretische Informatik
Universität Ulm

11. August 2009

Inhaltsverzeichnis

Kapitel 1

Vorlesungsinhalte

Die Vorlesung "SAT Solving", gehalten im Sommersemester 2009 von Uwe Schöning, umfasst folgende Themen:

- Grundlagen (Aussagenlogik, Normalformen, kombinatorische Eigenschaften von (k-)SAT, Tseitin-Codierung, NP-Vollständigkeit von SAT)
- Kalkülbasierte Algorithmen (Resolution, DP
- DPLL-Algorithmen (DPLL, Heuristiken wie z.B. Monien-Speckenmeyer, nicht-chronologisches Backtracking, Klausellernen, reale SAT Solver)
- Lokale-Suche-Algorithmen(deterministische lokale Suche mit Überdeckungs-codes, stochastische lokale Suche mit random walk, sowie Heuristiken und reale SAT Solver)
- Spezialfälle von SAT $\in P$ (2-KNF, Hornformeln, Mixed Horn, Renamable Horn)
- Spezialthemen (SAT-Schwellenwert, Lovasz Local Lemma)

Kapitel 2

Aussagenlogik und SAT

2.1 Grundlegende Begriffe

2.1.1 Syntax

Definition 1 *Gegeben sei eine abzählbar unendliche Menge von Variablen* $V = \{x_1, x_2, \ldots\}$. *Formeln (über der Variablenmenge* V*) werden induktiv wie folgt definiert:*

- *Konstanten* $0, 1$ *sind für sich genommen Formeln.*
- *Jede Variable* $x \in V$ *ist eine Formel.*
- *Wenn* F *eine Formel ist, dann auch* $\neg F$ *(manchmal auch* $\sim F$ *oder* $\overline{F}$ *geschrieben).*
- *Wenn* F *und* G *Formeln sind, dann auch* $(F \wedge G), (F \vee G)$. *Weitere Möglichkeiten:* $(F \rightarrow G), (F \leftrightarrow G), (F \oplus G)$.

Definition 2 *Wir bezeichnen die Menge der Variablen in einer Formel mit* $Var(F)$.

Formaler 1 *Es ist* $Var(0) = Var(1) = \emptyset$, *sowie* $Var(x) = \{x\}, Var(\neg F) = Var(F)$ *und* $Var(F \circ G) = Var(F) \cup Var(G)$.

Definition 3 *Die Größe einer Formel* F *bezeichnen wir mit* $|F|$.

Formaler 2 *Die Größen verschiedener Formeln:* $|0| = |1| = |x| = 1$. *Desweiteren ist* $|\neg F| = |F| + 1$, *sowie* $|F \circ G| = |F| + |G| + 1$.

Definition 4 *Ein Literal ist eine Variables* x_i *oder eine negierte Variable* $\neg x_i$.

Definition 5 *Eine Disjunktion von Literalen* $(y_1 \vee \ldots \vee y_j)$ *wird als Klausel bezeichnet.*

2.1.2 Semantik

Definition 6 *Die Belegung/Bewertung ist eine endliche, partielle Abbildung* $\alpha : V \rightarrow \{0, 1\}$.

Hinweis: Bei einer Abbildung, die für die Variable x_i keine Zuweisung macht, schreibt man anstatt $\alpha(x_i)$ undef. auch $\alpha(x_i) = \perp$ bzw. $\alpha(x_i) = *$. Das Angeben von Belegungen wird durch eine Mengenschreibweise vorgenommen. Beispiel: $\alpha = \{x_1 \leftarrow 1, x_2 \leftarrow 0\}$ (weist Variablen x_1 eine 1 zu und Variablen x_2 eine 0).

Definition 7 *Sei α eine Belegung für eine Formel F, dann heißt α total, wenn* $\underbrace{D(\alpha)}_{\text{Definitionsbereich von } \alpha} \supseteq Var(F)$.

Definition 8 *Eine Auswertung einer Formel F mit Hilfe einer Belegung α, geschrieben $F\alpha$, erhalten wir, indem wir für jede Variable in der Formel $x_i \in Var(F) \cap D(\alpha)$ ihr Vorkommen mit ihrem Wert $\alpha(x_i)$ ersetzen. Sodann führt man folgende Vereinfachungen auf der entstandenen Formel durch:*

- $\neg 0 \rightsquigarrow 1, \neg 1 \rightsquigarrow 0$.
- $0 \wedge F \rightsquigarrow 0, F \wedge 0 \rightsquigarrow 0$.
- $0 \vee F \rightsquigarrow F, F \vee 0 \rightsquigarrow F$.
- $1 \wedge F \rightsquigarrow F, F \wedge 1 \rightsquigarrow F$.
- $1 \vee F \rightsquigarrow 1, F \vee 1 \rightsquigarrow 1$.
- $\neg\neg F \rightsquigarrow F$

Definition 9 *Falls für eine Formel F und eine Belegung α gilt $F\alpha = 1$, so sagen wir α erfüllt F (oft auch α ist ein Modell von F).*

Definition 10 *Eine Formel F heißt erfüllbar, falls es ein α gibt mit $F\alpha = 1$.*

Definition 11 *Eine Formel heißt gültig (oder Tautologie), falls für alle totalen Belegungen α für F gilt $F\alpha = 1$.*

Definition 12 *Die Formeln F und G heißen äquivalent (geschrieben $F \equiv G$), falls für alle totalen Belegungen für F und G gilt $F\alpha = G\alpha$.*

Definition 13 *Zwei Formeln F und G heißen erfüllbarkeitsäquivalent (geschrieben: $F \approx G$), gdw. gilt F erfüllbar $\Leftrightarrow$ G erfüllbar.*

2.1.3 Notation

Es werden hiermit folgende Notationen vereinbart:
F, G, H sind Formeln.
α, β, γ sind Belegungen (sowohl partiell als auch total).
a, b, c sind Wahrheitswerte.
x, y, z (ggf. mit Index) sind Variablen.
n Anzahl der Variablen in einer Formel.
m Anzahl der Klauseln in einer Formel.
Wir fassen Klauseln als Mengen von Literalen auf. Formeln werden als Mengen von Klauseln aufgefasst.

2.1.4 Äquivalenzumformungen

Es gelten folgende Äquivalenzumformungen:
Distributivität: $F \wedge (G \vee H) = (F \wedge G) \vee (F \wedge H)$ und $F \vee (G \wedge H) = (F \vee G) \wedge (F \vee H)$.
Absorptionsgesetze: $F \wedge (F \vee G) = F$ und $F \vee (F \wedge G) = F$.
De-Morgan: $\neg(F \vee G) = \neg F \wedge \neg G$ und $\neg(F \wedge G) = \neg F \vee \neg G$, sowie $\neg\neg F = F$, $\neg 0 = 1$ und $\neg 1 = 0$.

2.1.5 Normalformen

Definition 14 *F liegt in Negationsnormalform (NNF) vor, falls in F als logische Operationen nur* $\wedge, \vee, \neg$ *verwendet werden und alle Negationen direkt vor den Variablen stehen.*

Bemerkung 1 *Die Umformung einer Formel in eine äquivalente NNF ist mit linearem Aufwand möglich.*

Definition 15 *Wir definieren* $pos(F, \alpha)$ *als die Menge der Literale in F, die unter* α *den Wert* 1 *annehmen.*

Satz 1 *Sei F in NNF. Seien* α, α' *Belegungen mit* $pos(F, \alpha) \subseteq pos(F, \alpha')$*. Dann gilt:* $\alpha \models F \Rightarrow \alpha' \models F$*.*

Beweis 1 *Klar.*

Lemma 1 ***(Koinzidenzlemma)*** *Sei* α *eine Belegung und F eine Formel. Sei* α' *eine eingeschränkte Belegung von* α *auf nur die Variablen, die in F vorkommen. Dann ist* $F\alpha' = F\alpha$*.*

Das Koinzidenzlemma begründet die Verwendung von Wahrheitstafeln. Sei $Var(F) = \{x_1, \ldots, x_n\}$.

	$x_1, x_2, \ldots, x_n$	$F\alpha_i$
α_1	$0, 0, \ldots, 0$	$F\alpha_1$
α_2	$0, 0, \ldots, 1$	$F\alpha_2$
$\vdots$	$\vdots$	$\vdots$
α_{2^n}	$1, 1, \ldots, 1$	$F\alpha_{2^n}$

Tabelle 2.1: Beispiel einer Wahrheitstafel. Das Tupel der Wahrheitswerte der Formel F unter allen möglichen Belegungen α_i nennt man den *Wahrheitswerteverlauf* (von F).

Bemerkung 2 *F ist erfüllbar* $(F \in SAT) \Leftrightarrow$ *Der Wahrheitswerteverlauf von F enthält mindestens eine* 1*.*
F ist ist Tautologie $(F \in TAUT) \Leftrightarrow$ *Der Wahrheitswerteverlauf von F enthält keine* 0*.*

Definition 16 *Eine Formel F liegt in KNF vor, falls* $F = K_1 \wedge \ldots \wedge K_m$*, wobei* $K_i = (l_{i_1} \vee \ldots \vee l_{i_{n_i}})$ *und* $l_j \in \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$ *(Literale).*

Definition 17 *Eine Formel F liegt in DNF vor, falls* $F = K_1 \vee \ldots \vee K_m$ *(mit* K_i *Klauseln), wobei* $K_i = (l_{i_1} \wedge \ldots \wedge l_{i_{n_i}})$ *und* $l_j \in \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$ *(Literale).*

Bemerkung 3 *KNF's und DNF's sind auch immer NNF's.*

Satz 2 *Für jede Formel F gibt es eine äquivalente Formel* F' *in DNF, sowie eine äquivalente Formel* F'' *in KNF.*

Beweis 2 *Assoziiere mit jeder Belegung* $\alpha : Var(F) \to \{0, 1\}$ *einen Term*

$$T(\alpha) = x_1^{e_1} \wedge \ldots \wedge x_n^{e_n}, \text{ wobei}$$

$$x^e = \begin{cases} x, & \text{wenn } \alpha(x) = 1 \\ \neg x, & \text{wenn } \alpha(x) = 0 \end{cases}$$

Es gilt für alle Belegungen β folgende Aussage:

$$\beta \models T(\alpha) \Leftrightarrow \beta \supseteq \alpha.$$

Seien $\alpha_1, \ldots, \alpha_m$ diejenigen Belegungen, mit

$$\alpha_j \models F (j \in \{1, \ldots, m)\}).$$

Konstruktion F':
Setze $F' = T(\alpha_1) \vee T(\alpha_m)$. Dies ist eine DNF. Es gilt $F' \equiv F$, denn:

$$\beta \models F \Leftrightarrow \exists j \in \{1, \ldots, m\} : \alpha_j \models F, \beta \supseteq \alpha_j \tag{2.1}$$

$$\Leftrightarrow \exists j : \beta \models T(\alpha_j) \Rightarrow \beta \models F'. \tag{2.2}$$

Konstruktion F'':
Bilde $\neg F$ und konstruiere zu dieser Formel eine DNF F''' gemäß des ersten Teils dieses Beweises. Bilde nun $\neg F'''$. Dann ist $F \equiv \neg F''' \underbrace{\equiv \ldots \equiv}_{\text{de Morgan}} F''$.

□

Bemerkung 4 *Das Erzeugen einer DNF (bzw. KNF) aus einer gegebenen Formel hat im Allgemeinen exponentiellen Aufwand. Ein Beispiel für solch einen Fall ist die Paritätsfunktion.*

Definition 18 *Die Formel F ist in k-KNF (bzw. k-DNF), falls F in KNF (bzw. DNF) vorliegt und für alle Klauseln K_j der Formel gilt: $|K_j| \leq k$.*

Sowohl k-KNF, als auch k-DNF sind keine Normalformen. Es lässt sich für alle k eine Formel F in $(k+1)$-KNF angeben, so dass diese zu keiner k-KNF Formel äquivalent ist, die genau die selben Variablen enthält.[1]

Wähle hierzu bspw. $F = (x_1 \vee \ldots \vee x_{k+1})$. F ist in $(k+1)$-KNF. Angenommen, es gibt eine Formel G in k-KNF mit $G \equiv K_1 \wedge \ldots \wedge K_m$, $|K_i| \leq k$.

Nun kann keine der Klauseln K_j in G alle $k+1$ Variablen enthalten. Sei o. B. d. A. $x_{k+1} \notin K_1$.

Wähle nun eine Belegung α, so dass $K_1\alpha = 0$ und $\alpha(x_{k+1}) = 1$. Daraus folgt dann aber, dass $F\alpha = 1$ und $G\alpha = 0$. Somit können F und G nicht äquivalent sein. Dies ist also nur eine erfüllbarkeitsäquivalente Umformung.

Eine weitere Menge von Formeln sind die sogenannten Horn-Formeln.

Definition 19 *Eine Formel heißt Horn-Formel, wenn sie in KNF ist und für alle Klauseln der Formel gilt, dass sie höchstens ein positives Literal enthalten.*

Wir unterscheiden die in Horn-Formeln enthaltenen Klauseln wie folgt:

- *Zielklauseln: Klauseln, die nur aus negativen Literalen bestehen.*
- *Faktenklauseln: Klauseln, die genau ein positives Literal enthalten.*
- *Regelklauseln: Klauseln, die genau ein positives und mindestens ein negatives Literal enthalten.*

Faktenklauseln und Regelklauseln werden zusammen auch als Menge der definiten Horn-Klauseln bezeichnet.

Wir werden später spezielle Algorithmen zur Lösung von Horn-Formeln betrachten.

Tseitin-Codierung

Die Tseitin-Codierung wird verwendet, um eine Boolesche Formel in eine erfüllbarkeitsäquivalente 3-KNF Formel zu überführen. Siehe Abbildung 2.1 für ein Beispiel

[1] Dies darf nicht damit verwechselt werden, dass man jede $(k+1)$-KNF in eine k-KNF (oder sogar 3-KNF) umwandeln kann. Hierzu müssen aber ggf. neue Variablen und neue Klauseln eingeführt werden.

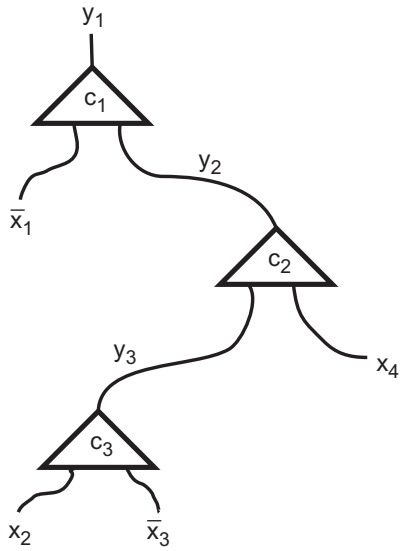

Abbildung 2.1: Eine Formel F gegeben als Schaltkreis mit mit "Unterschaltkreisen' c_i (die c_i haben jeweils genau zwei Eingänge.)

einer solchen Formel F gegeben als Schaltkreis. Führe für jedes c_i eine neue Variable y_i ein und setze

$$F' = (y_1) \wedge (y_1 \leftrightarrow c_1(\neg x_1, y_2)) \tag{2.3}$$
$$\wedge\, (y_2 \leftrightarrow c_2(\neg y_3, x_4)) \tag{2.4}$$
$$\wedge \quad \underbrace{(y_3 \leftrightarrow c_3(\neg x_2, \neg x_3))}_{\text{Boolesche Funktion in 3 Variablen}} \tag{2.5}$$

Damit ist $F \approx F'$. Definitionsgemäß enthält eine 3-KNF Formel lediglich die logischen Operationen $\wedge, \vee, \neg$. Es müssen zur Herstellung einer 3-KNF also noch die Äquivalenzoperationen aufgelöst werden. Die Terme, die die Äquivalenzoperationen enthalten, werden dazu durch (maximal 8) 3-KNF Klauseln ersetzt (in Abh. der enthaltenen c_i). Als Ergebnis erhalten wir F'' mit

$$F'' = (y_1) \wedge (\ldots) \wedge \ldots \wedge (\ldots) \tag{2.6}$$
$$\wedge\, (\ldots) \wedge \ldots \wedge (\ldots) \tag{2.7}$$
$$\wedge \quad \underbrace{(\ldots) \wedge \ldots \wedge (\ldots)}_{\text{äquivalent zu maximal 8 3-KNF Klauseln}} \tag{2.8}$$

Bemerkung 5 *Falls die c_i monoton sind (also NNF's darstellen), dann sind die Äquivalenzoperationen "$\leftrightarrow$" in den Termen nach der ersten Umformung in der Tseitin-Codierung lediglich Implikationen "$\rightarrow$". Dies erzeugt im zweiten Schritt der Tseitin-Codierung evtl. weniger bzw. einfachere 3-KNF Klauseln und somit eine einfachere Formel F''.*

Die Tseitin-Codierung begründet eine Reduktion $F \mapsto F'$ von SAT nach 3-SAT, so dass $F \in SAT \Leftrightarrow F' \in SAT \cap 3-KNF$.

2.2 Kombinatorische Eigenschaften von (k-)SAT

Satz 3 *Jede k-KNF Formel ist erfüllbar, falls sie weniger als 2^k Klauseln hat, wobei jede Klausel genau k verschiedene Literale haben muss.*

Beweis 3 ***(1. probabilistisch)***
Sei $F \in k-KNF$, sowie $n = |Var(F)|$ und $m = |F|$. Wähle Belegung $\alpha \in_R \{0,1\}^n$ zufällig. Sei C eine Klausel in F.
Es gilt: $Pr(\alpha$ erfüllt C nicht$) = 2^{-k}$. Definiere:

$$X_C := [\alpha \text{ erfüllt } C \text{ nicht}] = \begin{cases} 1, C\alpha = 0 \\ 0, \text{ sonst} \end{cases}$$

Damit wird

$$E(\#\text{ unerfüllte Klauseln}) = E(\textstyle\sum_{C\in F} X_C) = \sum_{C\in F} E(X_C) = m \cdot 2^{-k}.$$

Dies ist < 1 falls $m < 2^k$. Dann: $E(\#$ unerfüllte Klauseln$) < 1$.
$\Longrightarrow \exists\alpha : \#$unerfüllte Klauseln unter α ist 0; d.h. α ist Modell von F.

□

Beweis 4 ***(2. Zählen)***
Für $C \in F$ sei $\overline{sat}(C)$ die Menge der Belegungen, die C nicht erfüllen. Dann gilt:

$$\overline{sat}(C) = 2^{n-k}$$

Die k Variablen in C müssen auf eine bestimmte Art belegt werden; die restlichen $n-k$ Variablenbelegungen können frei gewählt werden.

$$\Longrightarrow |\textstyle\bigcup_{C\in F} \overline{sat}(C) \leq \sum_{C\in F} |\overline{sat}(C)| = m \cdot 2^{n-k}$$

Sofern $m < 2^k$ folgt $m \cdot 2^{n-k} < 2^n = \{0,1\}^n$. Daraus folgt $\exists\alpha : F\alpha = 1$.

□

Verallgemeinerung und Verschärfung von Satz 3:

Satz 4 *Sei F eine KNF-Formel mit $\sum_{C\in F} 2^{-|C|} < 1$. Dann ist F erfüllbar. Ferner kann die erfüllende Belegung effizient (in $|F|$ und n) berechnet werden.*

Beweis 5 ***(Existenz)***
Wähle $\alpha \in_R \{0,1\}^n$. Für $C \in F$ sei $X_C := [\alpha$ erfüllt C C nicht$]$.
Dann ist $E(X_C) = Pr(\alpha$ erfüllt C nicht$) = 2^{-|C|}$.
Sei $Y_F \in \{0,\ldots,m\}$ eine Zufallsvariable, wobei $Y_F = \#$ Klauseln, die unter α nicht erfüllt werden. Dann wird:
$E(Y_F) = E(\sum_{C\in F} X_C) = \sum_{C\in F} E(X_C) = \sum_{C\in F} 2^{-|C|} < 1$, nach Voraussetzung.
$\Longrightarrow Pr(Y_F = 0) > 0 \Longrightarrow \exists\alpha : F\alpha = 1$.

(Effizienz, Derandomisierung - Methode der bedingten Wahrscheinlichkeit)
Es gilt:

$$\begin{aligned} E(Y_F) &= E(Y_F|x_1 \leftarrow 0) \cdot Pr(x_1 \leftarrow 0) + E(Y_F|x_1 \leftarrow 1) \cdot Pr(x_1 \leftarrow 1) \\ &= \tfrac{1}{2} \cdot (E(Y_F|x_1 \leftarrow 0) + E(Y_F|x_1 \leftarrow 1)). \end{aligned}$$

Es folgt, dass es eine Auswahl $a_1 \in \{0,1\}$ gibt, so dass $E(Y_F|x_1 \leftarrow a_1) \leq E(Y_F)$.
Es gilt $E(Y_F|x_1 \leftarrow 0) = \sum_{C\in F} \underbrace{E(X_{C\{x_1\leftarrow 0\}})}_{=2^{-|C\{x_1\leftarrow 0\}|}}$.
Somit kann der Wert für a_1 (die Belegung für x_1) effizient berechnet werden. Fahre so fort und ermittle die Werte $a_2,\ldots,a_n$.
Für die totale Belegung $\beta = \{x_1 \leftarrow a_1,\ldots,x_n \leftarrow a_n\}$ gilt $E(Y_F|\beta) \leq E(Y_F) < 1$.
Also ist der Erwartungswert für die Anzahl der Klauseln, die durch β nicht erfüllt werden, kleiner 1. Somit $F\beta = 1$.

(Die Methode der bedingten Wahrscheinlichkeiten geht auf Erdös zurück.)

□

Beweis 6 ***Alternativer Beweis mit Scoring-Funktionen***
Gegeben eine partielle Belegung α, definiere Score einer Klausel C durch:

$$sc(C, alpha) = \begin{cases} 0, C\alpha = 1 \\ 2^{-(C\alpha)}, \ sonst \end{cases}.$$

Für Variablen x, die in α nicht belegt werden gilt:

$$sc(C, \alpha) = \tfrac{1}{2} \cdot (sc(C, \alpha \cup \{x = 0\}) + sc(C, \alpha \cup \{x = 1\}))$$

Erweitere die Score-Funktion von Klauseln auf Formeln:

$$sc(F, \alpha) = \textstyle\sum_{C \in F} sc(C, \alpha)$$

Dann gilt:

- $\alpha = \emptyset \Rightarrow sc(F, \alpha) = \sum_{C \in F} 2^{-|C|}$
- α *total* $\Rightarrow sc(F, \alpha) = \#$ *unerfüllte Klauseln unter* α
- α *partiell* x *in* α *nicht belegt*
  $\Rightarrow sc(F, \alpha) = \frac{1}{2} \cdot (sc(C, \alpha \cup \{x = 0\}) + sc(C, \alpha \cup \{x = 1\}))$

Finde sukzessive Werte a_i mit $sc(F, \alpha \cup \{x_i \leftarrow a_i\}) \leq sc(F, \alpha)$. Dies ergibt eine erfüllende Belegung.

□

Aufgabe: Zeigen Sie, dass folgende Aussagen äquivalent sind:

- $\sum_{C \in F} 2^{-|C|} = 1$
- Alle totalen Belegungen α erfüllen F bis auf eine Klausel.
- Für alle Paare $C, D \in F, C \neq D$ gilt: $C \cup D$ enthält komplementäres Paar $x, \neg x$ von Literalen.

2.3 Lovász Local Lemma und SAT

Im Folgenden betrachten wir Formeln $F = K_1 \wedge \cdots \wedge K_m$ in k-KNF, bestehend aus m Klauseln, $|K_i| = k$.

Definition 20 *Die* Nachbarschaft *einer Klausel $K \in F$ ist definiert als*

$$N(K) \ = \ \{K' \in F \mid Var(K) \cap Var(K') \neq \emptyset\}$$

Man beachte, dass K selbst als Nachbar von K betrachtet wird.

Satz 5 *Sei $F = K_1 \wedge \cdots \wedge K_m$ eine Formel in k-KNF. Es gelte $|N(K)| < 2^{k-2}$ für alle Klauseln K in F. Dann ist F erfüllbar, und in diesem Falle kann eine erfüllende Belegung mit einem stochastischen Algorithmus (mit hoher Wahrscheinlichkeit) effizient (in der Länge der Formel) bestimmt werden.*

Beweis 7 *Sei n die Zahl der Variablen in F. Betrachte folgenden stochastischen Algorithmus:*

1. *Wähle eine zufällige Anfangsbelegung $\alpha \in_R \{0,1\}^n$.*
2. *FOR $i := 1$ TO m DO IF $K_i\alpha = 0$ THEN Fix(K_i)*
3. *Gib die gefundene, erfüllende Belegung α aus.*

Der Prozeduraufruf Fix(K) modifiziert die Belegung α so, dass am Ende - sofern die Prozedur terminiert - die Klausel K erfüllt wird, und alle Klauseln, die vor Aufruf von Fix bereits erfüllt waren, immer noch erfüllt bleiben.

1. *PROC Fix(K: Klausel)*
2. *Ändere α ab, indem für die Variablen in K neue k Zufallsbits $\in_R \{0,1\}^k$ zugewiesen werden.*
3. *WHILE (es gibt $K' \in N(K)$ mit $K'\alpha = 0$) DO Fix(K')*

Wir müssen zeigen, dass die Prozedur Fix immer terminiert (und das sogar innerhalb polynomialer Zeit). Betrachten wir einen Zeitpunkt, wenn genau s mal die Prozedur Fix (rekursiv) aufgerufen wurde. Bis zu diesem Zeitpunkt sind insgesamt $n+sk$ viele Zufallsbits in die Rechnung eingeflossen (für die Anfangsbelegung n Bits, und für jeden Fix-Aufruf k Bits). Nehmen wir an, der Algorithmus bezieht seine Zufallsbits aus einem Kolmogorov-random String x, genauer, sei $C(x|F) \geq n + sk$.

Wir beschreiben den String x nun auf eine besondere Weise, und schätzen damit $C(x|F)$ nach oben ab. Mit n Bits lässt sich diejenige Belegung angeben, die α am Ende nach den s Fix-Aufrufen erhalten hat. Im Hauptprogramm kann ein Aufruf von Fix(K) für jede Klausel K nur höchstens einmal erfolgen. Also lassen sich diejenigen Klauseln K, für die im Hauptprogramm ein Fix-Aufruf erfolgte, mittels m vielen Bits spezifizieren (das i-te Bit ist =1, genau dann, wenn ein Aufruf von $Fix(K_i)$ erfolgt).

Jeder rekursive Fix-Aufruf geschah, da eine der Klauseln in der Nachbarschaft der zuvor betrachteten Klausel K auf Null gesetzt wurde. Diese Nachbarschaft besteht aus maximal $r < 2^{k-2}$ Klauseln. Daher lässt sich die anschließend rekursiv aufgerufene Klausel durch eine Zahl $\in \{1, \ldots, r\}$ beschreiben. Für die Spezifikation einer solchen Folge $(k_1, \ldots, k_s)$, $k_i \in \{1, \ldots, r\}$, benötigt man $s \log_2(r)$ Bits.

Sofern für K jedoch gilt, dass kein Nachbar den Wert 0 hat, so kann ein Rücksprung in die aufrufende Inkarnation von Fix erfolgen. Um die genaue Aufrufstruktur nachvollziehen zu können notieren wir dies durch eine entsprechende Bitfolge (1 entspricht einem (neuen) rekursiven Aufruf von Fix; 0 entspricht einem Rücksprung in die aufrufende Inkarnation von Fix). In einem solchen Bitstring kann es höchstens so viele Nullen wie Einsen geben, daher ist dessen Länge höchstens 2s. Mit Hilfe dieser $s \log_2(r) + 2s$ vielen Bits an Informationen lässt sich der ganze Rekursionsbaum, einschließlich der bei den Aufrufen verwendeten Klauseln rekonstruieren.

Aus diesen verschiedenen Informationen lässt sich der gesamte Rechenvorgang rückwärts nachvollziehen, so dass man alle Bits in x rekonstruieren kann. Entscheidend ist dabei die Beobachtung, dass es zum Rückgängigmachen eines Fix(K)-Aufrufs genügt, die k Variablen-Bits in α aufzusuchen, die zu den in K vorkommenden Variablen gehören, und diese auf die einzig mögliche Belegung zurückzusetzen, die die Klausel K auf 0 setzen kann.

Somit erhalten wir:

$$n + sk \leq C(x|F) \leq n + m + s \log_2(r) + 2s$$

Unter Verwendung der Voraussetzung $r < 2^{k-2}$ bzw. $\log_2(r) < k - 2$ erhalten wir dann $s \leq cm$ für eine Konstante c. Der Algorithmus terminiert also, und das auch noch effizient.

Da die "meistenSStrings x Kolmogorov-random sind, wird obiger stochastischer Algorithmus mit hoher Wahrscheinlichkeit in $O(m)$ Schritten eine erfüllende Belegung finden.

□

Möglichkeiten der Verbesserung/Modifikation:

- Mit 2^{2s} ist die Anzahl der 0-1-Strings mit n Einsen und der Eigenschaft, dass in keinem Präfix die Anzahl der Nullen überwiegt, nur grob abgeschätzt. Das geht vielleicht noch besser (Prüfer-Codes?)

- Der Algorithmus wäre auch dann korrekt, wenn man nur dann Fix(K_j) rekursiv aufruft, wenn $j \leq i$. Vielleicht bringt das was (vor allem wenn man zuvor die Reihenfolge der Klauseln zufällig permutiert).

- Man könnte den Algorithmus so modifizieren, dass er anstelle gleich k neue Zufallsbits für eine auf Null gesetzte Klausel auszuwählen, nur ein Literal in der Klausel auswählt und den Wert dieses Literals flippt. Dann hat man einerseits nur $C(x|F) \geq n + s\log_2(k)$, man kann aber beim Begriff der Nachbarschaft spezifischer sein: eine Nachbarklausel ist von dieser Belegungsänderung höchstens dann betroffen, wenn sie das betreffende komplementäre Literal enthält, man muss also nicht ganz $N(K)$ berücksichtigen.

Eine interessante Variante: Sei l ein Literal in einer Klausel K. Definiere die Menge der *l-Resolutionspartner* von K als $Res(K,l) = \{K' \in F \mid \bar{l} \in K'\}$.

Satz 6 *Sei $F = K_1 \wedge \cdots \wedge K_m$ eine Formel in k-KNF. Es gelte $|Res(K,l)| < k/4$ für alle Klauseln K in F und $l \in K$. Dann ist F erfüllbar, und in diesem Falle kann eine erfüllende Belegung mit einem stochastischen Algorithmus (mit hoher Wahrscheinlichkeit) effizient (in der Länge der Formel) bestimmt werden.*

Beweis 8 *Analog zum oben angegebenen, mit einigen kleinen Änderungen. Bei einem Aufruf von Fix(K) wähle man per Zufall nur eines der k Literale von K aus und flippe dessen Wert in α. Diese Auswahlentscheidung erfordert $\log_2(k)$ zufällige Bits. Die einzigen Klauseln, die von dieser Maßnahme betroffen sein könnten (und ihren Wert unter α von 1 nach 0 verändern könnten) sind die Klauseln in $Res(K,l)$ wobei l das zum Flippen ausgewählte Literal in K ist. Wir erhalten die entsprechende Abschätzung:*

$$n + s\log_2(k) \leq C(x|F) \leq n + m + s\log_2(r) + 2s,\ r < k/4$$

und erhalten daraus $s \leq cm$ für eine Konstante c.

□

2.4 Zufallsformeln für k-SAT

2.4.1 Random k-SAT

Im folgenden betrachten wir ein Modell $\mathcal{F}_{k,n,p}$ zum Erzeugen von k-KNF Zufallsformeln (auch bekannt als RANDOM instances). Der dem Modell zu Grunde liegende Algorithmus zum Erzeugen der Formeln bekommt drei Parameter als Eingabe:

- n, die Anzahl der Variablen, die in der Formel enthalten sein sollen,

- k, die Länge der Klauseln in der k-KNF, und

- p, mit $p \cdot n = m$ die Anzahl der Klauseln, in der erzeugten Formel. Parameter p wird oft auch als "ratio" bezeichnet.

Der Algorithmus führt nun folgende Schritte durch, um eine Zufallsformel $F \in_R \mathcal{F}_{k,n,p}$ zu erzeugen:

1. Setze zu Beginn $F := \emptyset$.

2. Wiederhole $p \cdot n$ mal:

   (a) Wähle k verschiedene Variablen aus den n zur Verfügung stehenden aus. Hierfür gibt es $\binom{n}{k}$ Möglichkeiten.

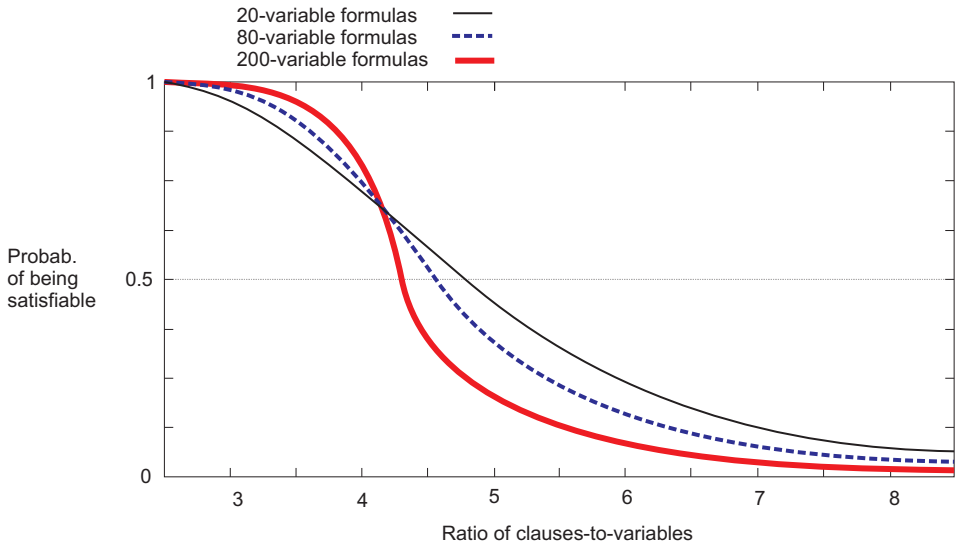

Abbildung 2.2: Die Abbildung zeigt die Wahrscheinlichkeit für Formeln $F \in_R \mathcal{F}_{3,n,p}$ erfüllbar zu sein (für festes n und variables p). Eine wichtige Beobachtung ist hierbei, dass der Schwellwert scheinbar gegen 4.26 konvergiert. Dies ist jedoch nicht bewiesen.

(b) Wähle für jede der gewählten Variablen zufällig (Wahrscheinlichkeit 0.5) ihr Vorzeichen. Hierfür gibt es 2^k Möglichkeiten.

(c) Fasse die k gewählten Literale in einer Klausel K zusammen.

(d) Füge K zu F hinzu.

Bemerkung 6 *Das durch $\mathcal{F}_{k,n,p}$ beschriebene Modell zum Erzeugen von Zufallsformeln entspricht dem "Ziehen mit Zurücklegen" von $p \cdot n$ Klauseln aus der Menge aller k-stelligen Klauseln auf n Variablen.*

Bemerkung 7 ***Empirische Beobachtungen***
Falls p "klein" gewählt ist, so ist die Formel F fast sicher erfüllbar (under-constrained). Falls p "groß" gewählt ist, so ist die Formel F fast sicher unerfüllbar (over-constrained). Die Bezeichnungen "klein" und "groß" hängen dabei sehr stark von k ab.

Bisher ist für 2-SAT folgendes bekannt:

- Für $F \in_R \mathcal{F}_{2,n,p}, p < 1$ ist $\lim_{n\to\infty} Pr(F \in SAT) = 1$.
- Für $F \in_R \mathcal{F}_{2,n,p}, p > 1$ ist $\lim_{n\to\infty} Pr(F \in SAT) = 0$.

D.h. dass der Schwellenwert für random 2-SAT Formeln genau bei 1 liegt. Für 3-SAT wurde solch ein Schwellenwert bisher nur empirisch beobachtet (bei $p = 4,26$, siehe Abbildung 2.2).

Die **theoretischen Ergebnisse** zum Schwellenwert bei 3-SAT Formeln sind wie folgt:

1. Für $F \in_R \mathcal{F}_{3,n,p}, p < 3,9$ ist $\lim_{n\to\infty} Pr(F \in SAT) = 1$.
2. Für $F \in_R \mathcal{F}_{3,n,p}, p > 4,8$ ist $\lim_{n\to\infty} Pr(F \in SAT) = 0$.
3. (Nach Friedgut) Je größer n wird, desto steiler fällt die Kurve am Schwellenwert ab. Jedoch ist nicht bekannt, wo genau der Schwellenwert liegt.

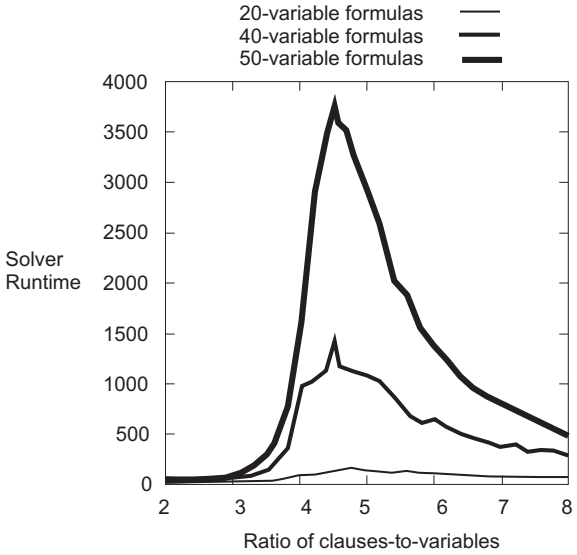

Abbildung 2.3: Die Abbildung zeigt die Laufzeit eines vollständigen SAT Solvers für Formeln $F \in_R \mathcal{F}_{3,n,p}$ (für festes n und variables p). Eine wichtige Beobachtung ist hierbei, dass die Laufzeit am längsten ist, wenn die Formel gerade mit 50% Wahrscheinlichkeit erfüllbar ist (vgl. hierzu Abbildung 2.2).

Bei SAT Solvern, die sowohl die Erfüllbarkeit als auch die Unerfüllbarkeit von Formeln feststellen können (complete SAT solver), ergibt sich ein Laufzeitverhalten wie in Abbildung 2.3 dargestellt. Man muss hierbei verstehen, dass das n, also die Anzahl der Variablen in der Formel, fest ist. Lediglich p und damit $m = p \cdot n$ (die Anzahl der Klauseln) wird variiert.

Zu beobachten ist, dass vollständige SAT Solver bei steigender ratio zunächst länger braucht, um die (Un)erfüllbarkeit der Formel festzustellen, da er beim Suchen einer erfüllenden Belegung mehr Klauseln beachten muss. Nach dem Schwellenwert hingegen sinkt die Laufzeit wieder ab, da es dem vollständigen SAT Solver nun (aufgrund der vielen Klauseln) möglich ist, einfacher einen Widerspruch in der Formel zu finden und damit ihre Unerfüllbarkeit festzustellen.

Betrachten wir Punkt 2 der theoretischen Ergebnisse etwas genauer. Ein Beweis für eine solche Schranke von p (wenn auch etwas schlechter als in Punkt 2 angegeben), lautet wie folgt:
Sei α eine Belegung (fest). Sei K eine k-KNF Zufallsklausel für eine Formel $F \in_R \mathcal{F}_{k,n,p}$. Dann ist:

$$Pr(K\alpha = 1) = \tfrac{7}{8}.$$

Für die Formel F mit $p \cdot n$ Klauseln gilt dann:

$$Pr(F\alpha = 1) = \left(\tfrac{7}{8}\right)^{n \cdot p}.$$

Damit:

$$Pr(\exists \alpha : F\alpha = 1) \leq \sum_{\alpha} Pr(F\alpha = 1) \tag{2.9}$$

$$= 2^n \left(\frac{7}{8}\right)^{n \cdot p} \tag{2.10}$$

$$\stackrel{!}{=} \frac{1}{2} \tag{2.11}$$

Und dies ergibt eine obere Schranke für den Schwellenwert:

$$2^n \left(\frac{7}{8}\right)^{n \cdot p} = \frac{1}{2} \Leftrightarrow n + n \cdot p \cdot \log_2 \frac{7}{8} = -1 \tag{2.12}$$

$$\Leftrightarrow n \cdot p \cdot \log_2 \frac{8}{7} = 1 + n \tag{2.13}$$

$$\Leftrightarrow n \cdot p = \frac{1+n}{\log_2 \frac{8}{7}} \tag{2.14}$$

$$\Leftrightarrow p = \frac{1}{\log_2 \frac{8}{7}} \cdot \left(\frac{1}{n} + 1\right) \tag{2.15}$$

$$\Rightarrow p \to 5,1909 (n \to \infty) \tag{2.16}$$

Im folgenden betrachten wir noch ein Modell, mit dessen Hilfe Zufallsformeln mit mindestens einer garantierten Lösung erzeugt werden können.

2.4.2 Planted random k-SAT

Um eine Zufallsformel F mit n Variablen, die mindestens eine garantierten Lösung hat, zu erzeugen, geht man wie folgt vor:

1. Wähle $\alpha \in_R \{0,1\}^n$.
2. Wiederhole $p \cdot n$ mal:
   (a) Wähle k verschiedene Variablen aus den n zur Verfügung stehenden aus. Hierfür gibt es $\binom{n}{k}$ Möglichkeiten.
   (b) Wähle aus den $2^k - 1$ möglichen Vorzeichenwahlen für die k Variablen, welche unter α den Wert 1 für die Klausel ergeben, unter Gleichverteilung eine aus.
   (c) Fasse die k gewählten Literale in einer Klausel K zusammen.
   (d) Füge K zu F hinzu.

Das Problem bei so erzeugten Formeln ist, dass eine Analyse der Variablenvorkommen es gestattet zu erkennen, ob es sich um eine planted random oder um eine random Formel handelt. Mithin kann (effizient mit hoher Wahrscheinlichkeit) entschieden werden, ob die Formel erfüllbar ist oder nicht. Man gehe dazu wie folgt vor:
Sei o.B.d.A. $\alpha = 1^n$ (die Ausgangsbelegung für das erzeugen der planted random Formel). Alle rein negativen Klauseln dürfen in der Formel nicht vorkommen, da diese von α nicht erfüllt werden. Wir betrachten nun die Erwartungswerte der Literalvorkommen zu den Variablen, und stellen fest, dass diese nicht gleich sind.

$$E(\#x_i \in F) = \sum_{j=1}^{p \cdot n} Pr(x_i \in K_j) \tag{2.17}$$

$$= \sum_{j=1}^{p \cdot n} Pr(i \text{ wird ausgewählt}) \cdot \frac{4}{7} \tag{2.18}$$

$$\neq \sum_{j=1}^{p \cdot n} Pr(i \text{ wird ausgewählt}) \cdot \frac{3}{7} \tag{2.19}$$

$$= \sum_{j=1}^{p \cdot n} Pr(\neg x_i \in K_j) \tag{2.20}$$

$$= E(\#\neg x_i \in F) \tag{2.21}$$

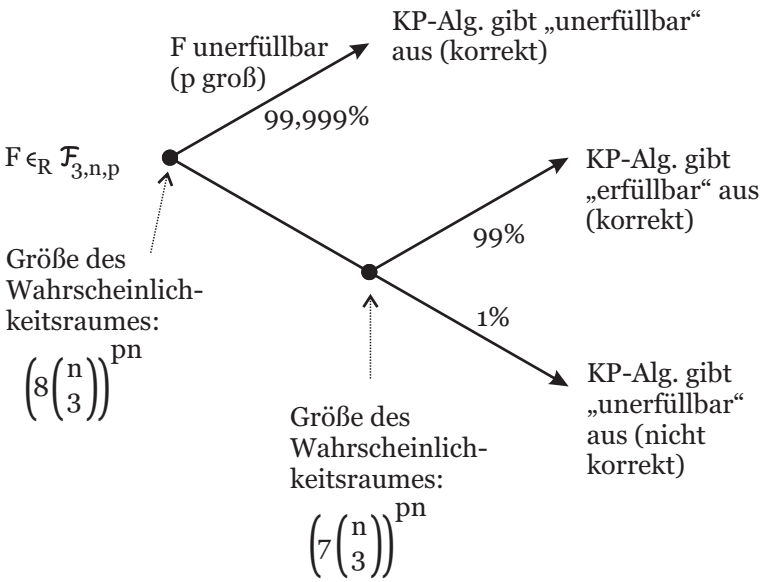

Abbildung 2.4: Die Abbildung zeigt die möglichen Antworten des KP-Algorithmus und ob diese korrekt sind. Dabei ist der Fall einer falschen Antwort von Interesse, und es soll gezeigt werden, wie die Wahrscheinlichkeit, eine falsche Antwort zu erhalten, vom Klausel-zu-Variablen Verhältnis (p) der Formel F abhängt.

Hierbei ist

$$Pr(i \text{ wird ausgewählt}) = \left(\tfrac{n-1}{n}\right)^3 = \left(1 - \tfrac{1}{n}\right)^3 \approx e^{-\frac{3}{n}}.$$

2.4.3 Algorithmus von Koutsoupias und Papadimitriou

Es wird im Folgenden der Algorithmus von Koutsoupias und Papadimitriou untersucht.

Listing 2.1: KP-Algorithmus

```
KP(F: k−KNF Formel, α: Ausgangsbelegung){
    Starte mit Ausgangsbelegung α;
    Setze β = 0^n;
    for (i = 1; i <=n; i++){
        if ((# erf. Klauseln in Fα|_{x_i geflippt}) > (# unerf. Klauseln in Fα|_{x_i geflippt})){
            β[i] = 1; //markiere x_i für einen Flip
        }
    }
    if (F(β ⊕ α) = 1) then {
        output(“ erfüllbar ’’);
    } else {
        output(“ unerfüllbar ’’);
    }
}
```

Sei $F \in_R \mathcal{F}_{3,n,p}$. Sei α^* eine erfüllende Belegung für F. Sei α eine beliebige für F totale Belegung mit $d(\alpha, \alpha^*) \leq \frac{n}{2}$. Wir analysieren im Folgenden, mit welcher Wahrscheinlichkeit der KP-Algorithmus eine richtige Antwort in Bezug auf die Erfüllbarkeit von F ausgibt, startend von α.

Die für diese Analyse zu betrachtenden Fälle sind in Abbildung 2.4 zusammengefasst. Wir wollen herausfinden, ab welchem p der Algorithmus bei seiner Ausgabe nur mit sehr kleiner Wahrscheinlichkeit einen Fehler macht (z.B. mit Wahrscheinlichkeit e^{-20}).

Definition 21 *Gegeben eine Belegung α und ein Index $i \in \{1, \ldots, n\}$ der Variablenindizes. Eine Klausel $K \in F$ heißt* signalisierend, *falls $K\alpha \neq K\alpha|_{x_i \leftarrow 1-x_i}$.*

Sei ferner α^* mit $F\alpha^* = 1$ gegeben. Ist K signalisierend, so gibt K ein günstiges *Signal (in Bezug auf α^*), genau dann wenn*

- $(K\alpha = 0) \wedge (\alpha_i \neq \alpha_i^*)$, *oder*
- $(K\alpha = 1) \wedge (\alpha_i = \alpha_i^*)$.

Ist K signalisierend, so gibt K ein ungünstiges *Signal (in Bezug auf α^*), genau dann wenn*

- $(K\alpha = 0) \wedge (\alpha_i = \alpha_i^*)$, *oder*
- $(K\alpha = 1) \wedge (\alpha_i \neq \alpha_i^*)$.

Die Anzahl der Klauseln, die in F ein günstiges Signal (in Bezug auf α^*) liefern ist $\binom{n-1}{2} \approx \frac{1}{2}n^2$.

Die Anzahl der Klauseln, die in F ein ungünstiges Signal (in Bezug auf α^*) liefern ist $\binom{n-1}{2} - \binom{n/2}{2} \approx \frac{3}{8}n^2$. Man beachte hier, dass von den möglichen Klauseln für diesen Fall noch diejenigen abgezogen werden können, bei denen bereits zwei Literale durch α richtige Werte bekommen. Eine solche Klausel kann kein ungünstiges Signal in Bezug auf α^* geben.

Sei X_i^+ die Anzahl der günstigen Klauseln in Bezug auf α und die Variable mit Index i. Es wird:

$$E(X_i^+) \approx pn \cdot \frac{\frac{1}{2}n^2}{7\binom{n}{3}} \approx \frac{pn\frac{1}{2}n^2}{7\frac{n^3}{6}} = \frac{3}{7}p \tag{2.22}$$

Sei X_i^- die Anzahl der ungünstigen Klauseln in Bezug auf α und die Variable mit Index i. Es wird:

$$E(X_i^-) \approx pn \cdot \frac{\frac{3}{8}n^2}{7\binom{n}{3}} \approx \frac{9}{28}p \tag{2.23}$$

Man fasse die Auswertung der Klauseln, in Bezug auf ihre Eigenschaft günstig oder ungünstig signalisierend zu sein, als ein Bernoulli Experiment auf (siehe Abbildung 2.5). Dabei wird eine Klausel erst in dem Moment erzeugt (gezogen), wenn sie in Bezug auf α als günstig oder ungünstig einschätzt wird. Man ist nun an dem für `KP` schlechten Fall interessiert, bei dem die Anzahl der günstigen Klauseln kleiner oder gleich ist wie die Anzahl der ungünstigen Klauseln (immer in Bezug auf α und genau einen Variablenindex i).

Zunächst ist

$$Pr(X_i^+ \leq X_i^-) \leq Pr(X_i^+ \leq \lambda) + Pr(X_i^- \geq \lambda) \tag{2.24}$$

mit $\lambda = \sqrt{\frac{9}{28}p \cdot \frac{3}{7}p} \approx 0,3712 \cdot p$ (dem geometrischen Mittel über die Erwartungswerte der Anzahl Klauseln für den günstigen und ungünstigen Fall in Bezug auf α und genau eine Variable mit Index i).

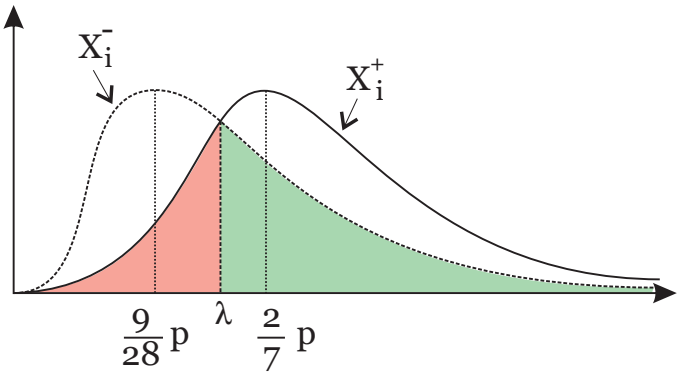

Abbildung 2.5: Die Wahrscheinlichkeitsverteilungen von X_i^- und X_i^+. Das geometrische Mittel von $E(X_i^-)$ und $E(X_i^+)$ bezeichnen wir hier mit λ.

Um mit der Ungleichung in 2.24 weiterarbeiten zu können, verwendet man die *Chernoff-Schranken*: Sei X eine n, w-binomialverteilte Zufallsvariable. Es ist $E(X) = nw$. Für die Chernoff-Schranken gilt nun

$$Pr(X > (1+\delta)nw) \leq e^{-\delta^2 nw/3} \tag{2.25}$$
$$Pr(X \leq (1-\delta)nw) \leq e^{-\delta^2 nw/4} \tag{2.26}$$

Damit erhalten wir aus 2.24 die Wahrscheinlichkeit für einen Fehler mit Variable i:

$$Pr(X_i^+ \leq X_i^-) \leq Pr(X_i^+ \leq \lambda) + Pr(X_i^- \geq \lambda) \tag{2.27}$$
$$\leq 0,9963^\lambda + 0,9963^\lambda \tag{2.28}$$
$$= 2 \cdot 0,9963^\lambda \tag{2.29}$$

Betrachten wir alle Variablen, so erhalten wir:

$$Pr(\exists i : X_i^+ \leq X_i^-) \leq \sum_{i=1}^{n} Pr(X_i^+ \leq X_i^-) \tag{2.30}$$
$$\leq n \cdot 2 \cdot 0,9963^\lambda \tag{2.31}$$

Soll dieser Fehler nun kleiner sein als bspw. e^{-20}, so ergibt sich:

$$n \cdot 2 \cdot 0,9963^{\lambda} = e^{-20} \Leftrightarrow \ln n + \ln 2 + \lambda \ln 0,9963 = -20 \tag{2.32}$$

$$\Leftrightarrow \lambda = \frac{-20 - \ln 2 - \ln n}{\ln 0,9963} \tag{2.33}$$

$$\Leftrightarrow \lambda \approx \frac{-(20 + \ln 2 + \ln n)}{-0,0037068} \tag{2.34}$$

$$\Leftrightarrow \lambda \approx \frac{20,69314 + \ln n}{0,0037068} \tag{2.35}$$

$$\Leftrightarrow \sqrt{\frac{9}{28}p \cdot \frac{3}{7}p} \approx \frac{20,69314 + \ln n}{0,0037068} \tag{2.36}$$

$$\Leftrightarrow \frac{27}{196}p^2 \approx \frac{20,69314 + \ln n}{0,0037068} \tag{2.37}$$

$$\Leftrightarrow p^2 \approx \frac{4055,85544 + \ln n}{0,1} \tag{2.38}$$

$$\Leftrightarrow p \approx \sqrt{\frac{4055,85544 + \ln n}{0,1}} \tag{2.39}$$

Die Wahrscheinlichkeit, dass der `KP`-Algorithmus einen Fehler macht, ist also kleiner als e^{-20}, falls p für das Erzeugen von $F \in_R \mathcal{F}_{3,n,p}$ wie oben gewählt ist. Für eine Formel mit 100 Variablen ist dies $p \approx 202$ ($\lambda \approx 5597$).

Kapitel 3

DPLL Algorithmen

Betrachten wir zunächst den naiv implementierten Test, ob eine Formel $F \in \text{SAT}$.

Listing 3.1: NAIV-SAT

```
NAIV−SAT(F:KNF Formel){
    for (all α : Var(F) → {0,1}){
        if (Fα == 1){
            return 1;
        } else {
            return 0;
        }
    }
}
```

Die worst-case Komplexität von NAIV-SAT liegt bei $O^*(2^n)$. Hierbei bedeutet $O^*(2^n)$ eine Laufzeit von $p(n) \cdot 2^n$ für irgendein Polynom p.

Definition 22 *Eine Klausel K heißt tautologisch, falls $x_i, \neg x_i \in K$ für ein festes i.*

Bemerkung 8 *Falls K eine tautologische Klausel ist, und F eine Formel mit $K \in F$, dann gilt: F erfüllbar $\Leftrightarrow F \backslash \{K\}$ erfüllbar.*

Definition 23 *Sei K eine Klausel und α eine Belegung. Dann heißt K unter α:*

- *erfüllt, falls für ein Literal $l \in K$ gilt, $\alpha(l) = 1$.*
- *widersprüchlich (Konfliktklausel), falls $\forall l \in K : \alpha(l) = 0$.*
- *unit-clause, falls es genau ein Literal $l \in K$ gibt mit $\alpha(l)$ undefiniert und $\forall l' \in K, l' \neq l : \alpha(l') = 0$.*
- *offen/unentschieden, andernfalls.*

Bemerkung 9 *Sei K eine unit-clause mit Literal l unter α. Sei F eine Formel und sei $K \in F$. Dann gilt $F\alpha$ erfüllbar $\Leftrightarrow F\alpha \cup \{l \leftarrow 1\}$ erfüllbar.*

Definition 24 *Ein Literal l heißt pur in einer Formel F, falls nur l aber nicht $\neg l$ in F vorkommt.*

Bemerkung 10 *Sei l pur in F, dann gilt: F ist erfüllbar $\Leftrightarrow F\{l \leftarrow 1\}$ ist erfüllbar.*

Wenn man, ausgehend von einer leeren Belegung, nach und nach einzelne Variablen belegt und die Formel unter diesen sukzessiven Belegungen immer weiter vereinfacht, so resultiert dies in einigen Fällen in vereinfachten Formeln, die unit-clauses und pure Literale enthalten.

Des weiteren können durch den Einsetzungsprozess (und die damit verbundenen Vereinfachungen) entweder die leere Klausel ($K = \square$), oder die leere Formel $F = \emptyset$ entstehen. Im ersten Fall kann die Belegung, unter der die Vereinfachungen vorgenommen wurden, sicher nicht zu einer Lösung von F erweitert werden. Im zweiten Fall wurden aus der Formel alle Klauseln entfernt (da alle erfüllt wurden), und somit ist die verwendete Belegung eine Lösung für F. Es gelten folgende Zusammenhänge:

Bemerkung 11 *$F \subseteq G, G$ erfüllbar $\Rightarrow$ F erfüllbar. $F \subseteq G, F$ unerfüllbar $\Rightarrow$ G unerfüllbar.*

Ist eine Formel erfüllbar, so muss es für jede der in der Formel enthaltenen Variablen eine Belegung geben, unter der die Formel wahr wird. In anderen Worten: Sei $x_i \in Var(F)$, dann gilt

$$F \text{ erfüllbar} \Leftrightarrow F\{x_i \leftarrow 1\} \text{ erfüllbar oder } F\{x_i \leftarrow 0\} \text{ erfüllbar.}$$

Umgekehrt gilt auch:

$$F \text{ unerfüllbar} \Leftrightarrow F\{x_i \leftarrow 1\} \text{ unerfüllbar und } F\{x_i \leftarrow 0\} \text{ unerfüllbar.}$$

3.1 DPLL

Unter Zuhilfenahme der oben genannten Fakten, lässt sich der DPLL-SAT Algorithmus angeben.

Listing 3.2: DPLL-SAT

```
DPLL−SAT(F:KNF Formel){
    while (∃ unit−clauses K = {l} ∈ F){
        F = F{l ← 1};
    }
    //eventuell kann man hier auch pure Literale belegen
    if (F == 0) return 0;
    if (F == 1) return 1;
    x_i = WähleAus(Var(F)); // gemäß Heuristik oder randomisiert
    if (DPLL−SAT(F{x_i ← 1}) == 1){
        return 1;
    } else {
        return DPLL−SAT(F{x_i ← 0});
    }
}
```

Bemerkung 12 *Der DPLL-SAT Algorithmus hat eine exponentielle worst-case Komplexität. Des weiteren kann dieser Algorithmus leicht so erweitert werden, dass er im Falle von F erfüllbar eine erfüllende Belegung ausgibt.*

3.2 Resolution

Definition 25 *Seien K_1, K_2 nicht-tautologische Klauseln. Seien ferner die Literale $x_i \in K_1$ und $\neg x_i \in K_2$. Man sagt dann, K_1, K_2 sind resolvierbar auf x_i. Es heißt*

$$R := (K_1 \backslash \{x_1\}) \cup (K_2 \backslash \{\neg x_1\})$$

der Resolvent von K_1, K_2 nach x_i.

Lemma 2 ***(Resolutionslemma)***
F enthalte Klauseln K_1, K_2, die resolvierbar sind und den Resolvent R erzeugen. Dann gillt:

$$F \text{ erfüllbar} \Leftrightarrow F \cup \{R\} \text{ erfüllbar.}$$

Beweis 9 *"$\Leftarrow$": klar.*
"$\Rightarrow$": Sei F erfüllbar. Dann existiert α mit $\alpha \models F$, und insbesondere auch $\alpha \models K_1, \alpha \models K_2$.
Daher existieren Literale $l_1 \in K_1$ und $l_2 \in K_2$ mit $\alpha \models l_1$ und $\alpha \models l_2$.
Nun ist $l_1 \neq x_i$ oder $l_2 \neq x_i$, da sonst entweder K_1 oder K_2 nicht von α erfüllt würde.
Nach Resolution ist nun $l_1 \in R$ oder $l_2 \in R$, da mindestens eines dieser Literale durch die Resolution unberührt bleibt.
Daraus folgt $\alpha \models R$ und mithin $\alpha \models F \cup \{R\}$.

□

Das Resolutionskalkül ist *korrekt*, d.h. dass wenn □ aus F ableitbar ist, so muss F unerfüllbar sein. Das Wort "ableitbar" bedeutet hier, dass es eine Folge von Klauseln $K_1, \ldots, K_t = \square$ gibt, so dass für $i = 1, \ldots, t$ gilt: K_i ist aus F, oder K_i ist Resolvent von K_{j_1}, K_{j_2} mit $j_1, j_2 < i$. In anderen Worten: Man darf nur mit Klauseln resolvieren, die entweder in der Formel selbst vorkommen, oder die bereits durch einen vorhergehenden Resolutionsschritt erzeugt wurden.

Das Resolutionskalkül ist *widerlegungsvollständig*, d. h. wenn F unerfüllbar, dann lässt sich □ aus F ableiten. Dies kann man durch Induktion über $n = |Var(F)|$ zeigen:

Induktionsanfang:
Klar für $n = 0$, da hier F die leere Klausel bereits enthalten muss um unerfüllbar zu sein. Ebenso ist bei $n = 1$ die Unerfüllbarkeit genau dann gegeben, wenn es zwei unit-clauses gibt die die Variable einmal als positives Literal in der einen unit-clause und einmal als negatives Literal in der anderen unit-caluse enthalten. Man benötigt hier genau einen Resolutionsschritt (zwischen diesen beiden Klauseln), um die leere Klausel zu erzeugen.
Induktionschritt:
Sei F also unerfüllbar und $|Var(F)| > 0$. Dann gibt es $x_i \in Var(F)$. Dann gilt: $F\{x_i \leftarrow 1\}$ unerfüllbar und $F\{x_i \leftarrow 0\}$ unerfüllbar. Nun lässt sich nach Induktionsvoraussetzung □ einmal aus $F\{x_i \leftarrow 1\}$ und einmal aus $F\{x_i \leftarrow 0\}$ ableiten.

Die selbe Beweisstruktur lässt sich auch auf F ohne Belegung anwenden und liefert dann $\{x_i\}$ und $\{\neg x_i\}$. Dies kann dann zu □ resolviert werden.

Im worst-case ist die Beweislänge $B(n) \geq 2^n$. Hierbei ist $B(0) = 0, B(1) = 1$ und $B(n) \leq 2B(n-1) + 1$.

Bemerkung 13 *Sind zwei Klauseln nach mehr als einer Variablen resolvierbar, so muss der Resolvent zwangsläufig eine Tautologie werden. Daher ist die Resolution zweier solcher Klauseln zwecklos, denn mit einer tautologischen Klausel wird man nie in der Lage sein, die leere Klausel abzuleiten.*

3.2.1 Resolutionsstrategien

Eine Strategie legt fest, wie ein nichtdeterministischer Prozess (hier Resolution) deterministisch abgearbeitet wird. Durch das Anwenden von Strategien verliert man nicht die Widerlegungsvollständigkeit der Resolution, da das Resolutionskalkül selbst nicht verändert wird. Beispiele für Resolutionsstrategien:

- **Level-saturation (Breitensuche):** Hierbei erzeugt man zuerst alle Resolventen, mit den initial zur Verfügung stehenden Klauseln (Level 1). Danach

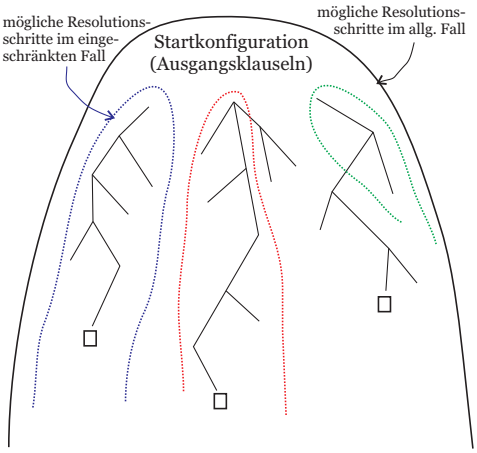

Abbildung 3.1: Führt man Restriktionen für die Resolution ein, so kann man damit den Resolutionsbeweis in eine bestimmte Form bringen. Der Verzweigungsgrad aller möglichen Resolutionsbeweise für eine Formel wird dann kleiner. Unter Umständen werden aber die Resolutionsbeweise länger oder das Kalkül verliert seine Widerlegungsvollständigkeit.

erzeugt man alle Resolvencten, mit den initialen Klauseln und denen aus Level 1 etc.

- **Unit-preference:** Hierbei werden immer zuerst die Resolventen gebildet, die sich mit unit Klauseln bilden lassen.

3.2.2 Resolutionsrestriktionen

Bei Restriktionen wird ein Kalkül abgeändert, so dass ursprünglich erlaubte Ableitungsschritte nun verboten sind. Man bringt damit den Resolutionsbeweis in eine bestimmte Form. Durch solche Verbote kann das Resolutionskalkül seine Widerlegungsvollständigkeit verlieren. Der positive Effekt von solchen Verboten liegt darin, dass der "branching Faktor" der möglichen Beweise kleiner wird. Ein negativer Effekt von solchen Verboten liegt darin, dass ein Beweis evtl. länger wird (siehe Abbildung 3.1). Beispiele für Resolutionsrestriktionen:

- **P-Resolution:** Eine der beiden Elternklauseln für die Resolution darf nur aus positiven Literalen bestehen.
- **N-Resolution:** Eine der beiden Elternklauseln für die Resolution darf nur aus negativen Literalen bestehen.
- **Lineare Resolution:** Zu Beginn wählt man sich eine Basisklausel aus der Menge der Ausgangsklauseln aus und resolviert diese dann mit einer weiteren Klausel aus der Menge der Ausgangsklauseln. Danach müssen die Resolventen einer Resolution wieder als Elternklausel für die nächste Resolution (mit einer Seitenklausel) verwendet werden. In anderen Worten: ein gebildeter Resolvent muss im nächsten Resolutionsschritt wieder verwendet werden (siehe Abbildung 3.2).
- **Unit Resolution:** Mindestens eine Elternklausel muss Unit sein. Unit-Resolution ist im Allgemeinen nicht widerlegungsvollständig.

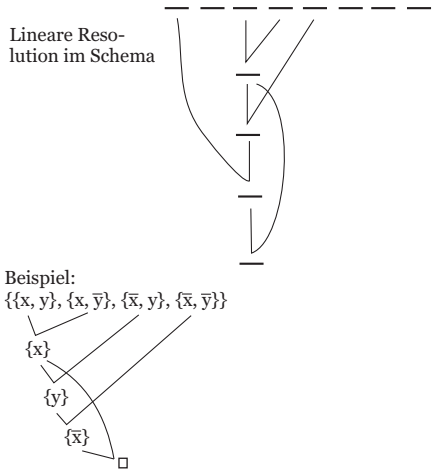

Abbildung 3.2: Lineare Resolution (oben) und ein Beispiel für lineare Resolution (unten).

- **SLD Resolution:** Bei dieser Resolution wird mit einer rein negativen Klausel und einer für diese Klausel resolvierbaren Horn-Klausel resolviert. Die dann durch die Resolution zur negativen Klausel hinzugefügten Literale müssen durch nachfolgende Resolutionsschritte mit weiteren Horn-Klauseln wieder entfernt werden. Die Resolutionsschritte sind demnach Stack-artig angelegt. Der Prologinterpreter arbeitet mit diesem Verfahren.

Satz 7 *P-Resolution ist widerlegungsvollständig, d.h. es gilt:*

$$F \text{ unerfüllbar} \Rightarrow F \vdash_{P-Res} \square.$$

Beweis 10 *Induktion über $n = |Var(F)|$. Die Induktionsvoraussetzung ist, dass es einen Resolutionsbeweis für die Unerfüllbarkeit von F mittels P-Resolution gibt.*
Induktionsanfang: *$n = 0 : F = \{\square\}$.*
Induktionsschritt: *$n \to n+1 : x \in Var(F)$. Betrachte $F_0 = F\{x \leftarrow 0\}$ und $F_1 = F\{x \leftarrow 1\}$. Nach Induktionsvoraussetzung existieren P-Resolutionen, so dass $F_0 \vdash_{P-Res} \square$ und $F_1 \vdash_{P-Res} \square$.*
Stelle die in F_0 gestrichenen Vorkommen von x wieder her. Dies liefert einen P-Resolutionsbeweis der Form

$$F \vdash_{P-Res} \{x\}.$$

Behandle alle noch vorhandenen Klauseln, die $\neg x$ enthalten, durch P-Resolution mit $\{x\}$. Damit stehen jetzt alle KLauseln aus F_1 zur Verfügung. Es kann nun der nach IV vorhandene Resolutionsbeweis $F_1 \vdash_{P-Res} \square$ verwendet werden, um die leere Klausel abzuleiten.
Die Anzahl der Resolutionsschritte ist $A(n) = 2 \cdot A(n-1) + m$.

Bemerkung 14 *Der obige Satz und sein Beweis gelten sinngemäß auch für N-Resolution.*

Satz 8 *Sei F unerfüllbar. Sei $F' \subseteq F$ minimal unerfüllbar. Sei $K \in F'$.*
Dann gibt es einen linearen Resolutionsbeweis, basierend auf K.

Beweis 11 *Wir zeigen den obigen Satz durch Induktion nach $n = |Var(F')|$.*
Induktionsanfang: *$(n = 0)$ $F' = \{\square\}; K = \square$.*
Induktionsschritt: *$(n = 0)$ Wir führen eine Fallunterscheidung durch.*

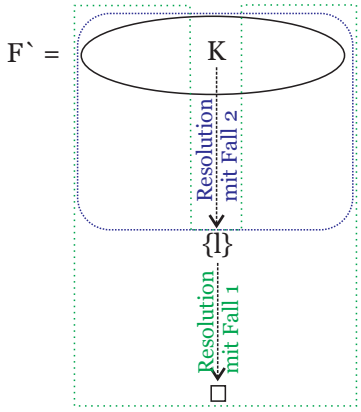

Abbildung 3.3: Die im Beweis hintereinander durchgeführten Resolutionsbeweise ergeben zusammen einen Resolutionsbeweis für die Unerfüllbarkeit von F'.

Fall 1: $|K| = 1$ *mit* $K = \{l\}$.
Dann ist $F'\{l \leftarrow 1\}$ *auch unerfüllbar, wobei aber nun* $|Var(F')| < n$. *Sein nun* $F'' \subseteq F'\{l \leftarrow 1\}$ *minimal unerfüllbar.*
In F'' *muss eine Klausel* K' *enthalten sein, so dass* $K' \cup \{\neg l\} \in F'$ *(ansonsten wäre* $F'' \subseteq F \backslash \{K\}$ *und damit wäre* F'' *erfüllbar, was nicht sein kann).*
Nach Induktionsvoraussetzung existiert ein linearer Resolutionsbeweis von F'', *basierend auf* K'. *Wir konstruieren nun, aufbauend auf dem linearen Resolutionsbeweis für* F'', *einen linearen Resolutionsbeweis für* F', *der wiederum auf* K *basiert.*
Mit $K = \{l\}, K' \cup \{\neg l\} \in F'$ *resolvieren wir linear und erhalten* K'.
Danach folgt der nach Induktionsvoraussetzung existierende lineare Resolutionsbeweis für F'' *und* K' *mit wieder eingefügtem Literal* l. *Als Ergebnis erhalten wir nun* $\{\neg l\}$, *was wir mit* K *zur leeren Klausel resolvieren können.*

Fall 2: $|K| > 1$.
Wähle ein beliebiges Literal $l \in K$. *Definiere* $K' = K \backslash \{l\}$.
Es gilt: $F'\{l \leftarrow 0\}$ *ist unerfüllbar, und* $K' \in F'\{l \leftarrow 0\}$.
Behauptung: $F'\{l \leftarrow 0\} \backslash \{K'\}$ *ist erfüllbar. Sei hierzu* α *eine erfüllende Belegung. Dann ist* $F' \backslash \{K\}\alpha = 1$. *Nun gilt* $K\alpha = 0$, *denn* $F'\alpha = 0$. *Daraus folgt* $\alpha(l) = 0$, *da* $l \in K$. *Daraus folgt wiederum* $F'\{l \leftarrow 0\} \backslash \{K\}\alpha = 1$. *Sei nun* F'' *minimal unerfüllbare Teilmenge von* $F'\{l \leftarrow 0\}$. *Aufgrund der obigen Behauptung muss* K *in* F'' *enthalten sein.*
Es gilt für F'': $|Var(F'') < n$.
Nach Induktionsvoraussetzung gibt es einen linearen Resolutionsbeweis von F'', *der auf* K' *basiert. Fügt man in diesen Beweis das Literal* l *wieder ein, so erhält man durch diesen Beweis die Klausel* $\{l\}$. *Dies liefert einen linearen Resolutionsbeweis für* K *und* F:
Beobachtung: $(F' \backslash \{K\}) \cup \{\{l\}\}$ *ist unerfüllbar. Aber* $F' \backslash \{K\}$ *ist erfüllbar (klar, denn* F' *ist minimal unerfüllbar).*
Mittels Fall 1 existiert nun ein linearer Resolutionsbeweis für $(F' \backslash \{K\}) \cup \{\{l\}\}$ *basierend auf* $\{l\}$. *Das zusammenfügen beider Resolutionsbeweise ergibt einen linearen Resolutionsbeweis für* F' *(siehe Abbildung 3.3).*

□

Korollar 1 *Inputresolution ist widerlegungsvollständig für Horn-Formeln.*

Beweis 12 *Sei* F *eine Horn-Formel. Man erinnere sich zunächst an die Definition von Horn-Formeln auf Seite 8. Eine Horn-Formel* F *ohne Zielklauseln ist stets*

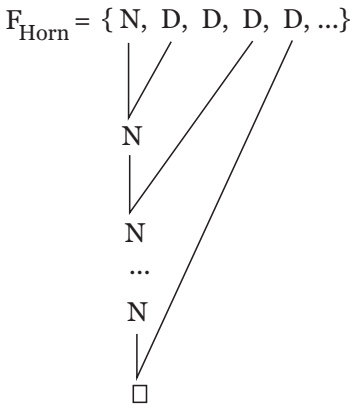

Abbildung 3.4: Die Resolution in einer Horn-Formel, die als Basisklausel eine negative Klausel verwendet, folgt der Restriktion der Input-Resolution, da mit jedem Resolutionsschritt nur wieder negative Klauseln entstehen können. Daher muss jede weitere verwendete Klausel aus der Menge der Anfangsklauseln sein.

erfüllbar (mit Belegung α, so dass $\alpha(x) = 1 \forall x \in Var(F)$). In einer minimal unerfüllbaren Teilformel F' von F muss mindestens eine Zielklausel enthalten sein. Wegen Satz 11 existiert ein linearer Resolutionsbeweis für F', basierend auf einer Zielklausel (siehe Abbildung 3.4). Man muss hierbei für jeden Resolutionsschritt eine input-Klausel verwenden, da man die negativen Klauseln nicht miteinander resolvieren kann. Daraus folgt, dass der Resolutionsbeweis eine input-Resolution ist. Input-Resolution ist widerlegungsvollständig für Horn-Formeln.

Satz 9 *Unit Resolution ist widerlegungsvollständig für (renamable) Horn-Formeln.*

Beweis 13 *Rein positive Horn-Klauseln sind die positiven unit-Klauseln. Eine unerfüllbare Horn-Formel muss mindestens eine positive unit-Klausel enthalten. Sonst konstruieren wir eine erfüllende Belegung mit α, so dass $\alpha(x) = 0 \forall x \in Var(F)$. Wegen Widerlegungsvollständigkeit der P-Resolution gibt es einen Resolutionsbeweis, wobei eine der Elternklausel immer eine Unit-Klausel ist.*

Wenn wir es mit einer renamable Horn-Formel zu tun haben, argumentieren wir wie folgt: Zuerst wenden wir ein Renaming an, was eine Horn-Formel ergibt. Dann führen wir unit-Resolution durch, wie oben beschrieben. Zum Schluss wird im gesamten Beweis das Renaming rückgängig gemacht, was uns einen unit-Resolutionsbeweis für die ursprüngliche Formel gibt.

□

Bemerkung 15 *Folgende Aussagen sind äquivalent:*

- *F besitzt unit-Resolutionsbeweis.*
- *F besitzt input-Resolutionsbeweis.*
- *F ist unerfüllbare (renamable) Horn-Formel.*

3.3 DP als Vorgänger von DPLL

Listing 3.3: DP

```
DP(F:KNF Formel){
    for (K ∈ F){
        if (K ist Tautologie) F := F\{K};
    }
    while (K ≠ ∅ und □ ∉ F){
        wähle x_i ∈ Var(F); //gemäß Heuristik
        //Bilde alle Resolventen nach x_i
        F' = F ∪ {K ∪ K' | K ∪ {x_i} ∈ F, K' ∪ {¬x_i} ∈ F}
        //Entferne die Elternklauseln der Resolventen
        F'' = {K ∈ F' | x_i ∉ Var(K)}
        //Entferne alle Tautologien
        F = {K ∈ F' | K nicht tautologisch}
    }
    if (F == ∅) return 1;
    if (□ ∈ F) return 0;
}
```

DP ist im Allgemeinen nicht so effizient wie DPLL, denn durch die Resolution entsteht ein exponentieller Blow-Up von $|F|$. Sei hierzu eine Formel $F = m_0 + m_1 + m_2$ gegeben, wobei m_1 die Anzahl der Klauseln ist, die x_i enthalten, m_2 ist die Anzahl der Klauseln, die $\neg x_i$ enthalten, und m_0 ist die Anzahl der dann noch verbleibenden Klauseln. Die größe der Formel F ist nach allen möglichen Resolutionen nach x_i: $\leq m_0 + m_1 \cdot m_2 - (m_1 + m_2)$.

Satz 10 *Es gilt F erfüllbar $\Leftrightarrow$ F' erfüllbar $\Leftrightarrow$ F'' erfüllbar.*

Beweis 14 *Es müssen vier Implikationen gezeigt werden.*
*"F' **erfüllbar** $\Rightarrow$ F **erfüllbar**": Klar, da $F \subseteq F'$.*
*"F' **erfüllbar** $\Leftarrow$ F **erfüllbar**": Sei F erfüllbar mit $\alpha \models F$.*
Betrachte $K \cup K' \in F'$, wobei $K \cup \{x_i\} \in F, K' \cup \{\neg x_i\} \in F$.
Falls $\alpha(x_i) = 0$, so gilt $\alpha \models K$, und mithin auch $\alpha \models K \cup K'$.
Falls $\alpha(x_i) = 1$, so gilt $\alpha \models K'$, und mithin auch $\alpha \models K \cup K'$.
*"F' **erfüllbar** $\Rightarrow$ F'' **erfüllbar**": Klar, da $F'' \subseteq F'$.*
*"F' **erfüllbar** $\Leftarrow$ F **erfüllbar**": Seien hierzu*
$F_+ := \{K | K \cup \{x_i\} \in F\}$ und $F_- := \{K | K \cup \{\neg x_i\} \in F\}$
Wir unterscheiden nun drei Fälle.
Fall 1: $\alpha \models K \forall K \in F_+$. Setze $\beta = \alpha \cup \{x_i \leftarrow 0\}$. Dann folgt $\beta \models F'$.
Fall 2: $\alpha \models K \forall K \in F_-$. Setze $\beta = \alpha \cup \{x_i \leftarrow 1\}$. Dann folgt $\beta \models F'$.
Fall 3: Wenn weder Fall 1 noch Fall 2 auftreten gibt es $K \in F_+$ mit $\alpha \not\models K$ und $K' \in F_-$ mit $\alpha \not\models K'$. Man erinnere sich, dass $K \cup \{x_i\} \in F$ und $K' \cup \{\neg x_i\} \in F$. Für den Resolventen $K \cup K' \in F''$ gilt nun $\alpha \not\models K \cup K'$. Widerspruch. Fall 3 kann es nicht geben.

□

Bemerkung 16 *DP hat im Allgemeinen exponentiellen Aufwand. Allerdings ist für 2-KNF die Anzahl der in der DP Methode maximal erzeugten Klauseln polynomiell beschränkt mit $1 + 2n + (2n)^2$. Daraus folgt, dass 2-KNF-SAT $(= SAT \cap$ 2-KNF$) \in P$.*

3.4 Zusammenfassung der möglichen Reduktionen für DPLL basierte Algorithmen

- **Tautologie:** $K \in F$, mit K tautologisch. Vereinfache mit $F := F \backslash \{K\}$.
- **Unit:** $K \in F, K = \{l\}$. Vereinfache mit $F := F\{l \leftarrow 1\}$.

- **Pures Literal:** l kommt in F pur vor. Vereinfache mit $F := F\{l \leftarrow 1\}$.
- **Subsumption:** $K_1, K_2 \in F, K_1 \subseteq K_2$. Vereinfache mit $F := F \backslash \{K_2\}$.
- **Resolution+Unit:** $\{x, y\}, \{\neg x, y\} \in F$. Vereinfache mit $F := F\{y \leftarrow 1\}$.
- **Resolution+Subsumption:** $K_1 \cup \{x\}, K_2 \cup \{\neg x\} \in F, K_1 \subseteq K_2$. Vereinfache mit $F := (F \backslash \{K_2 \cup \{\neg x\}\}) \cup K_2$.
- **DP-Reduktion:** $F' = F \cup$ alle Resolventen nach x. $F'' = F \backslash$ Elternklauseln der Resolutionen. Vereinfache $F = F'' \backslash$ tautologische Klauseln.
- **Klausel-Lernen:** Sei α so dass $F\alpha = 0$. Füge zu F eine Klausel K_α hinzu mit $Var(K_\alpha) = Def(\alpha)$, sowie K_α enthält x wenn $\alpha(x) = 0$, und K_α enthält $\neg x$ wenn $\alpha(x) = 1$.

Die folgende Reduktionsregel verwendet den Begriff der Autarkie.

Definition 26 *Eine Belegung α heißt eine Autarkie von F, wenn $\forall K \in F\alpha : K \in F$. Mit anderen Worten, α ist eine Autarkie von F, wenn $\forall K \in F : (Var(K) \cap Def(\alpha) \neq \emptyset \Rightarrow \alpha \models K)$.*

- **Autarkie:** Sei α eine Autarkie von F. Vereinfache mit $F := F\alpha$.

Satz 11 *Sei α eine Autarkie für F. Dann gilt:*
$$F \text{ erfüllbar} \Leftrightarrow \exists \beta, \alpha \subseteq \beta : \beta \models F \Leftrightarrow F\alpha \text{ erfüllbar.}$$

Beweis 15 *Zunächst: "$\Leftarrow$" klar.*
"$\Rightarrow$": Sei F erfüllbar mit $\gamma \models F$.
$$\text{Sei } \gamma' = \begin{cases} \alpha(x), \text{falls } x \in Def(\alpha) \\ \gamma(x), \text{falls } x \in Def(\gamma) \backslash Def(\alpha) \end{cases}$$
Dann gilt $\gamma' \models F$.

3.5 Heuristiken für die Auswahl und Belegung von Variablen in DPLL-Algorithmen

Wir definieren zunächst l als ein Literal, sowie

- $h_k(l) :=\#$ Vorkommen von l in Klauseln der größe k, und
- $h(l) := \sum_{k=2}^{n} h_k(l)$.

Mit Hilfe von h und h_k lassen sich folgende Heuristiken formulieren:

- **DLIS (Dynamic Largest Individual Sum):** Wähle ein Literal l mit $h(l)$ maximal und setze $l \leftarrow 1$.
- **DLCS (Dynamic Largest Clause Sum):** Wähle eine Variable x mit $h(x) + h(\neg x)$ maximal und setze zuerst
$$x \leftarrow \begin{cases} 1, h(x) \geq h(\neg x) \\ 0, h(x) < h(\neg x) \end{cases}.$$
- **RDLCS (Random Dynamic Largest Clause Sum):** Variablenwahl wie bei DLCS, jedoch wird die Variable zufällig belegt.
- **MOM (Maximum Occurance in Minimal Clause):** Sei w die kürzeste vorkommende Klausellänge. Wähle x so dass
$$(h_w(x) + h_w(\neg x)) \cdot 2^p + h_w(x) \cdot h_w(\neg x)$$
maximal ist. Hierbei ist p ein frei wählbarer Parameter.

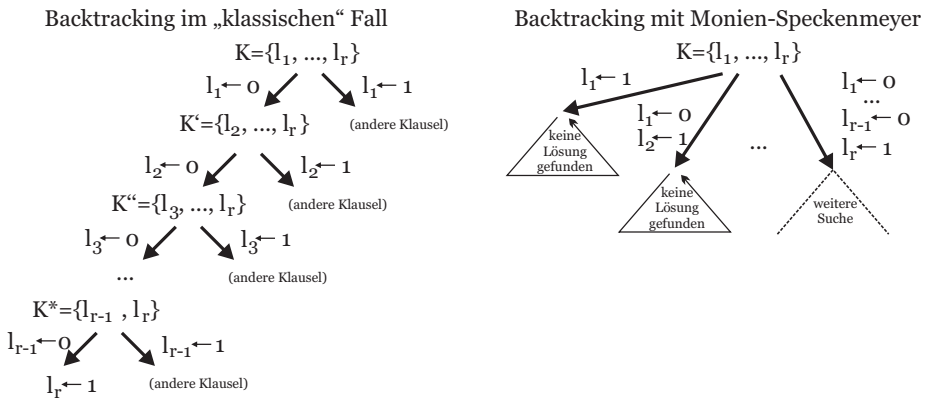

Abbildung 3.5: Im klassischen Backtracking wird stets nur eine Zuweisung vorgenommen, um die Suche voranzutreiben (links). Im Falle von Monien-Speckenmeyer werden mehrere solcher Zuweisungen zusammengefasst (rechts).

- **Böhm-Heuristik:** Sei x eine Variable. Definiere einen Vektor
$$H(x) = (H_2(x), \ldots, H_n(x)), \text{ mit}$$
$$H_i(x) := p_1 \cdot max(h_i(x), h_i(\neg x)) + p_2 \cdot min(h_i(x), h_i(\neg x)).$$
Hierbei sind p_1, p_2 frei wählbare Parameter. Man wähle nun diejenige Variable x aus, die gemäß einer lexikographischen Sortierung ein maximales $H(x)$ hat (beginnend mit $H_2(x)$).

- **Jeroslow-Wang-Heuristik:** Sei zunächst
$$J(l) := \sum_{i=2}^{n} h_i(l) \cdot 2^{-i} = \sum_{K:l \in K} 2^{-|K|}$$
  - **Einseitige Variante:** Wähle l mit $J(l)$ maximal und setze $l \leftarrow 1$.
  - **Zweiseitige Variante:** Wähle x mit $J(x) + J(\neg x)$ maximal und setze $x \leftarrow 1$, falls $J(x) \geq J(\neg x)$ und $x \leftarrow 0$, sonst.

- **Monien-Speckenmeyer-Heuristik:** Wähle erstes (beliebiges) Literal in einer (der ersten) kürzesten Klausel (siehe Kapitel 3.5.1).

3.5.1 Die Monien-Speckenmeyer-Heuristik

Sei $K = \{l_1, \ldots, l_r\}, r \leq k$ die erste kürzeste Klausel in der betrachteten Formel. Die Monien-Speckenmeyer-Heuristik wählt nun das erste Literal in dieser Klausel l_1, und verzweigt darauf einmal mit $l_1 \leftarrow 1$. Für den Fall, dass keine erfüllende Belegung gefunden wird, ist $K' = K \backslash \{l_1\}$ automatisch wieder die kürzeste Klausel und wird daher bei der nächsten Verzweigung gleich wieder gewählt (in K' wird man nun mit l_2 fortfahren). Man kann nun mehrere solcher Verzweigungsvorgänge zusammenfassen (siehe Abbildung 3.5). Als Prozedur richtig notiert löst die Monien-Speckenmeyer-Heuristik das k-SAT-Problem (siehe Listing 3.4).

Listing 3.4: MS

```
MS(F:k−KNF Formel){
    if (F == ∅) return 1;
    if (□ ∈ F) return 0;

    K = wähle erste kürzeste Klausel in F; //Sei K = {l_1, ..., l_r}
```

```
for (i = 1; i ≤ r; i++){
    α_i = {l_1 ← 0, ..., l_{i-1} ← 0, l_i ← 1};
    if (MS(Fα_i) == 1) return 1;
}
return 0;
```

Sei nun $n = Var(F)$. Die Laufzeit von `MS` ergibt sich über eine Rekursionsgleichung $T(n)$, mit

$$T(n) \leq T(n-1) + \ldots + T(n-k),$$

welche die maximal k verschiedenen Verzweigungen in `MS` für den worst-case modelliert. Unter der Annahme, dass $T(n) = \beta^n$ (also dass `MS` eine exponentielle worst-case Komplexität hat und dieser Fall auch eintritt), ergibt sich nun für T folgende Äquivalenzumformung:

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

Berechnet man β für verschiedene k, so ergeben sich die Werte aus Tabelle 3.1.

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

Tabelle 3.1: Die β-Werte für verschiedene k in der Rekursionsgleichung für die Berechnung der Laufzeit von `MS`.

Eine Modifikation von `MS` zur Verbesserung der Laufzeit kann mit Hilfe von Autarkien geschehen (siehe Listing 3.5).

Listing 3.5: MS2

```
MS2(F:k−KNF Formel){
    if (F == ∅) return 1;
    if (□ ∈ F) return 0;

    K = wähle erste kürzeste Klausel in F; //Sei K = {l_1, ..., l_r}
    for (i = 1; i ≤ r; i++){ //Autarkietest
        α_i = {l_1 ← 0, ..., l_{i-1} ← 0, l_i ← 1};
        if (α_i ist Autarkie für F) return MS2(Fα_i);
    }
    for (i = 1; i ≤ r; i++){
        α_i = {l_1 ← 0, ..., l_{i-1} ← 0, l_i ← 1};
        if (MS(Fα_i) == 1) return 1;
    }
    return 0;
```

Die bessere Laufzeit von `MS2` gegenüber `MS` ergibt sich nun wie folgt. Wir wissen, dass nach der ersten `for`-Schleife die α_i keine Autarkien mehr für F sein können. Daraus folgt, dass $F\alpha_i$ mindestens eine Klausel der Länge $k-1$ enthalten muss. Daher kann die Rekursionsgleichung umformuliert werden:

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

Der Worst-Case ist rein rechnerisch jeweils der zweite Fall, also der Fall, in dem keine Autarkien auftreten. Daher genügt es, den zweiten Fall zu analysieren. Wir erhalten für

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

sowie für

$$T(n) = O(T'(n))$$

und dem vorherigen Ansatz, dass die Rekursionsgleichung exponentiell ist ($T'(n) = \beta^n$), nun die charakteristische Gleichung

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

Daraus ergibt sich nun mit $k = 3$: $\beta = \frac{\sqrt{5}+1}{2} \approx 1,618$ (die Zahl des goldenen Schnitt).

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

Tabelle 3.2: Die β-Werte für verschiedene k in der Rekursionsgleichung für die Berechnung der Laufzeit von `MS2`.

3.6 Entwicklung der Laufzeitschranken

Im Laufe der Zeit wurden immer bessere Schranken für die Laufzeit von verschiedenen deterministischen 3-SAT Algorithmen entdeckt.

Erfinder	**Jahr**	**Gezeigte untere Schranke**
Monien, Speckenmeyer	1985	$1,618^n$
Kullmann	1990	$1,505^n$
Rodosek, Schiermeyer[1]	1992/1993	$1,497^n, 1,493^n$
Goerdt, Schöning	2000	$1,481^n$
Brüggemann, Kern	2002	$1,476^n$
Scheideler	?	$1,469^n$

Tabelle 3.3: Die β-Werte für verschiedene k in der Rekursionsgleichung für die Berechnung der Laufzeit von `MS2`.

3.7 Das Pigeon-Hole Problem (PH_n)

Trotz der Errungenschaften der letzten Jahrzehnte gibt es dennoch Formeln, die für SAT Solver inhärent schwierig bleiben. Ein Beispiel für solche Formeln ist das *Pigeon-Hole* Problem (auch als Schubfachschlussprinzip bekannt).

Wir zeigen im Folgenden, dass eine Resolutionswiderlegung für PH_n eine exponentielle Länge haben muss. Dazu beweisen wir zunächst ein Lemma welches besagt, dass eine Resolutionswiderlegung für eine Formel F in eine Resolutionswiderlegung der Formel $F\{y \leftarrow 1\}$ überführt werden kann. Danach führen wir den

[1] Die Ergebnisse von Rodosek und Schiermeyer sind bis heute umstritten.

Begriff des Pigeon-Hole-Problems formal ein und beweisen den Satz über die exponentielle Mindestlänge von Resolutionswiderlegungen von PH_n. Hierzu wird noch ein Lemma benötigt, was im Anschluss gezeigt wird.

Lemma 3 *Sei $\mathcal{R} = (K_1, K_2, \ldots, K_t = \emptyset)$ eine Resolutionswiderlegung einer Klauselmenge F. (Das heißt, für $i = 1, \ldots, t$ gilt, dass K_i entweder aus F stammt, oder dass K_i Resolvent zweier Klauseln K_j, K_l ist mit $j, l < i$.) Sei y ein Literal in F. Dann gibt es eine Resolutionswiderlegung $\mathcal{R}'$ für die Klauselmenge $F\{y \leftarrow 1\}$. Dieser neue Resolutionsbeweis enthält keine Klausel, die y enthält. Ferner ist die Länge von $\mathcal{R}'$ höchstens die Länge von $\mathcal{R}$ minus die Anzahl der Klauseln in $\mathcal{R}$, die y enthalten.*

Beweis 16 *Wir konstruieren den gesuchten Resolutionsbeweis $\mathcal{R}'$ schrittweise (also für $i = 1, \ldots, t$) aus $\mathcal{R}$. Und zwar bestehen für die Klauseln in $\mathcal{R}'$ jeweils drei Möglichkeiten: entweder wird die betreffende Klausel gestrichen und kommt in $\mathcal{R}'$ nicht vor, oder zweitens, die betreffende Klausel K'_i in $\mathcal{R}'$ hat die Eigenschaft $K'_i \subseteq K_i$ und ist ein Resolvent zweier Klauseln $K'_j \subseteq K_j$ und $K'_l \subseteq K_l$, die zuvor in $\mathcal{R}'$ aufgelistet wurden, oder drittens, K'_i ist ein Duplikat einer Klausel K'_j mit $j < i$. Auch in diesem dritten Fall gilt $K'_i \subseteq K_i$.*

- *Falls K_i aus F stammt und weder y noch $\overline{y}$ enthält, so enthält $\mathcal{R}'$ an dieser Stelle ebenfalls K_i.*
- *Falls K_i aus F stammt und $\overline{y}$ enthält, so enthält $\mathcal{R}'$ an dieser Stelle $K_i \setminus \{\overline{y}\}$.*
- *Falls K_i das Literal y enthält, so entsteht an dieser Stelle keine Klausel (also eine Streichung).*
- *Falls K_i Resolvent zweier Klauseln K_j, K_l ist, und die Klauseln $K'_j \subseteq K_j$ und $K'_l \subseteq K_l$ enthalten beide die Variable, nach der resolviert wird, so ist K_i in gleicher Weise Resolvent von K'_j, K'_l.*
- *Falls K_i Resolvent zweier Klauseln K_j, K_l ist, und die Klausel $K'_j \subseteq K_j$ enthält die Resolutionsvariable nicht, so wird $K'_i = K'_j$ gesetzt. (Dann gilt nämlich $K'_i \subseteq K_i$.)*
- *Falls K_i Resolvent zweier Klauseln K_j, K_m ist, und die Klausel K_j (oder K_m) wurde in $\mathcal{R}'$ bereits gestrichen, so enthielt diese Klausel das Literal l, während dieses im Resolventen nicht mehr vorkommt. Das heißt, es wurde nach l resolviert. Wir setzen $K'_i = K'_m \setminus \{\overline{l}\}$ (bzw. $K'_i = K'_j \setminus \{\overline{l}\}$).*

In dem man nun die Nummerierung der Klauseln neu vornimmt, wobei man gestrichene Klauseln bei der Nummerierung übergeht, und duplizierte Klauseln nur einmal auflistet, erhält man eine Resolutionswiderlegung $\mathcal{R}' = (K'_1, K'_2, \ldots, K'_r = \emptyset)$ mit $r \leq t$ für $F\{y \leftarrow 1\}$.

□

Damit ist das erste Lemma gezeigt und ein bestehender Resolutionsbeweis $\mathcal{R}$ für F kann in leicht abgeänderter Form $\mathcal{R}'$ auch für $F\{y \leftarrow 1\}$ verwendet werden. Wir führen nun das Pigeon-Hole-Problem formal ein.

Definition 27 *Die Pidgeonhole-Klauselmenge PH_n definiert man wie folgt: Dies sind zum einen für jedes $i = 0, 1, \ldots, n$ die Klausel*

$$(x_{i,1} \vee x_{i,2} \vee \cdots \vee x_{i,n})$$

Also: "Taube i muss in Taubenschlag 1 oder 2 oder ... oder n untergebracht werden".

Desweiteren haben wir für jedes $j = 1, 2, \ldots, n$ die Menge der $\binom{n+1}{2}$ vielen Klauseln

$$(\overline{x_{0,j}} \vee \overline{x_{1,j}}),\ (\overline{x_{0,j}} \vee \overline{x_{2,j}}), \ldots, (\overline{x_{n-1,j}} \vee \overline{x_{n,j}})$$

Also: "In Taubenschlag j darf nur eine Taube untergebracht werden".

Es ist klar, dass man $n+1$ viele Tauben nicht in n Taubenschläge auf diese Weise unterbringen kann, daher ist die Klauselmenge PH_n unerfüllbar.

Wir vermerken, dass PH_n insgesamt $n(n+1)$ Boolesche Variablen enthält und aus $(n+1) + n \cdot \binom{n+1}{2} = O(n^3)$ Klauseln besteht.

Es folgt nun der zentrale Satz dieses Kapitels der zeigt, dass jede Resolutionswiderlegung von PH_n exponentielle Länge in n haben muss.

Satz 12 *Jede Resolutionswiderlegung von PH_n muss mindestens die Länge $2^{n/20}$ haben.*

Beweis 17 *Nehmen wir an, es gibt einen Resolutionsbeweis $\mathcal{R} = (K_1, K_2, \ldots, K_s)$, $K_s = \emptyset$, von PH_n mit $s < 2^{n/20}$. Mit jeder Klausel K in $\mathcal{R}$ assoziieren wir eine Klausel $\widehat{K}$, die aus K hervorgeht, indem man jedes vorkommende negative Literal $\overline{x_{i,j}}$ streicht und stattdessen die positiven Literale $x_{i,k}$ für $k \neq j$ und $x_{l,j}$ für $l \neq i$ einfügt. Wir wollen im Folgenden eine Klausel K aus $\mathcal{R}$* groß *nennen, wenn die assoziierte Klausel $\widehat{K}$ mindestens $n(n+1)/10$ viele (positive) Literale enthält. Der Resolutionsbeweis $\mathcal{R}$ enthalte r viele große Klauseln. Natürlich gilt $r \leq s$. In diesen r vielen großen Klauseln kommen insgesamt mindestens $rn(n+1)/10$ viele Literale vor. Also muss es ein Literal $x_{i,j}$ geben, das in mindestens $r/10$ vielen großen Klauseln vorkommt. Indem wir nun der Reihe nach die folgenden partiellen Belegungen durchführen*

$$x_{i,j} \leftarrow 1,\ \textit{und}\ x_{i,k} \leftarrow 0\ \textit{für}\ k \neq j,\ \textit{und}\ x_{l,j} \leftarrow 0\ \textit{für}\ l \neq i,$$

und gemäß des vorigen Lemmas den Resolutionsbeweis $\mathcal{R}$ mit Hilfe dieser partiellen Einsetzungen zurechtstutzen, erhalten wir einen Resolutionsbeweis für PH_{n-1}, denn die oben angegebenen Belegungen "frieren"die Zuordnung von Taube i zu Taubenschlag j ein, und es verbleibt noch eine PH_{n-1}-Formel per Resolution zu widerlegen. Außerdem sind durch diese Maßnahme - wie man dem Beweis des obigen Lemmas entnimmt - mindestens $r/10$ viele große Klauseln K im neuen Resolutionsbeweis für PH_{n-1} weggefallen, da durch diese Belegungen entweder das positive Literal $x_{i,j}$ in K auf 1 gesetzt wurde, oder ein negatives Literal der Form $x_{i,k}$, $k \neq j$, oder $x_{l,j}$, $l \neq i$, auf 0 gesetzt wurde. Nach einem solchen Restriktionsschritt enthält der sich ergebende Resolutionsbeweis für PH_{n-1} nur noch höchstens $9r/10 \leq 9s/10$ viele große Klauseln. So fortfahrend können wir immer einen Anteil von mindestens einem Zehntel der verbleibenden großen Klauseln eliminieren, so dass nach spätestens $\log_{10/9} s$ vielen Restriktionsschritten keine große Klausel mehr vorhanden ist. Zu diesem Zeitpunkt haben wir es mit einem Resolutionsbeweis für $PH_{n'}$ zu tun, wobei

$$n' \geq n - \log_{10/9} s > n - \log_{10/9} 2^{n/10} > 0.671n$$

Das heißt, alle (assoziierten) Klauseln $\widehat{K}$ in dem sich ergebenden Resolutionsbeweis für $PH_{n'}$ haben weniger als $n(n+1)/10$, also auch weniger als $2n'(n'+1)/9$ viele Literale. Das folgende Lemma zeigt, dass dies ein Widerspruch ist. Damit war unsere Eingangsannahme $s < 2^{n/10}$ falsch, und der Satz ist bewiesen.

□

Lemma 4 *In jedem Resolutionsbeweis für PH_m muss es eine Klausel C geben, so dass die assoziierte Klausel $\widehat{C}$ mindestens $2m(m+1)/9$ viele Literale enthält.*

Beweis 18 *Sei $\mathcal{R}$ ein Resolutionsbeweis von PH_m.*

*Eine Belegung α heiße k-*kritisch*, wenn sie jeder Taube bis auf Taube k genau einen (jeweils verschiedenen) Taubenschlag zuordnet. Formaler: α ist k-kritisch, falls es eine bijektive Funktion f von $\{0,1,\ldots,m\}\setminus\{k\}$ nach $\{1,2,\ldots,m\}$ gibt, so dass gilt*

$$\alpha(x_{i,j}) \;=\; \begin{cases} 1, & f(i)=j \\ 0, & sonst \end{cases}$$

Wir definieren die Signifikanz *einer Klausel C in $\mathcal{R}$, abgekürzt $sfz(C)$, wie folgt:*

$$sfz(C) \;=\; \sum_{k=0}^{m} [\,\textit{es gibt eine } k\textit{-kritische Belegung, die } C \textit{ falsifiziert}\,]$$

Der eckige Klammerausdruck stellt die Indikatorfunktion dar: wenn die Aussage zwischen den Klammern zutrifft, so ist der Wert = 1, sonst = 0. Es ist klar, dass die Klauseln in PH_m Signifikanz 1 bzw. 0 haben. Die leere Klausel, die am Ende des Resolutionsbeweises $\mathcal{R}$ steht, hat Signifikanz $m+1$. Sofern K ein Resolvent zweier Klauseln K_1, K_2 ist, welche Signifikanz s_1 und s_2 haben, so ist die Signifikanz von K höchstens s_1+s_2. Daraus folgt, dass es eine Klausel C in $\mathcal{R}$ geben muss mit einer Signifikanz $s \in [\frac{m+1}{3}, \frac{2(m+1)}{3}]$. (Man wähle die erste Klausel in $\mathcal{R}$ mit Signifikanz $\geq \frac{m+1}{3}$.) Wähle eine k-kritische Belegung α, welche C falsifiziert. (Eine solche kann man, nach Definition von $sfz(C)$, auf s verschiedene Weisen auswählen.) Sei j solcherart, dass alle *j-kritischen Belegungen die Klausel C erfüllen. (Ein solches j kann man auf $m+1-s$ verschiedene Weisen auswählen; also gibt es $s\cdot(m+1-s)$ viele (k,j)-Kombinationen.) Wir modifizieren α an einer Stelle ab, so dass diese zu einer j-kritschen Belegung α' mutiert: anstelle der Taube k keinen Taubenschlag zuzuordnen und der Taube j einen Taubenschlag, sagen wir l, zuzuordnen, wird in α' der Taube k der Taubenschlag l und der Taube j nichts zugeordnet; sonst ändert sich in α' gegenüber α nichts. Diese Modifikation hat zur Folge, dass α' die Klausel C erfüllt. Also muss in $\widehat{C}$ das Literal $x_{k,l}$ existieren. Auf diese Weise kann man die Existenz von $s\cdot(m+1-s)$ vielen Literalen in $\widehat{C}$ nachweisen. Aus $s \in [\frac{m+1}{3}, \frac{2(m+1)}{3}]$ folgt, dass $s\cdot(m+1-s) \geq \frac{2m(m+1)}{9}$, und damit ist das Lemma bewiesen.*

□

Insgesamt haben wir damit gezeigt, dass es Formeln gibt, bei denen eine Resolutionswiderlegung exponentiellen Aufwand hat. Insbesondere bedeutet dies, dass resolutionsbasierte SAT-Solver die Unerfüllbarkeit solcher Formeln nicht effizient zeigen können.

3.8 Reale SAT Solver

Wir haben bereits zwei SAT Solver Implementierungen (`DPLL-SAT` auf Seite 22, `DP` auf Seite 27) vorgestellt, die auf Rekursion zurückgreifen. Im folgenden betrachten wir noch eine Implementierung, die auf Rekursion verzichtet (siehe Listing 3.6).

3.8.1 Iterativer DPLL-SAT

Listing 3.6: Iterativer DPLL-SAT

```
LS(F:k-KNF Formel){
    if (BCP() == CONFLICT){
        return UNSAT;
    }
    while (true){
```

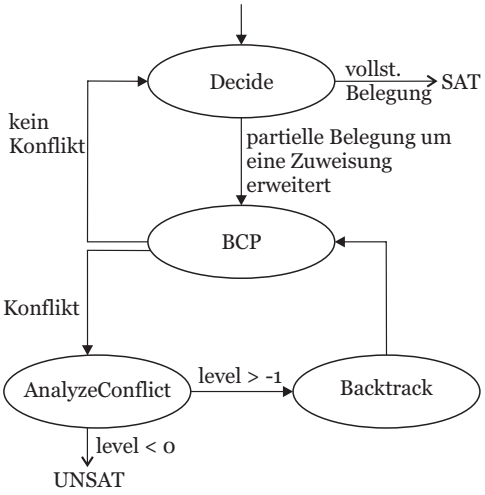

Abbildung 3.6: Die Arbeitsweise von DPLL-SAT ohne Rekursion.

```
    if (Decide() == noDecision){
        return SAT;
    } else {
        while (BCP() == CONFLICT){
            backtrack−level = AnalyzeConflict();
            if (backtrack−level < 0){
                return UNSAT;
            } else {
                Backtrack(backtrack−level);
            }
        }
    }
}
```

Abbildung 3.6 veranschaulicht die Arbeitsweise von `DPLL-SAT` ohne Rekursion. Die von diesem Algorithmus verwendeten Methoden sollen im einzelnen kurz vorgestellt werden.

- `Decide()`: `Decide()` ist eine Methode, die zunächst prüft, ob überhaupt noch eine Variable in der Formel belegt werden muss. Dies ist genau dann der Fall, wenn es noch unbelegte Variablen in unerfüllten Klauseln gibt. Falls `Decide()` keine solche Variable finden kann, muss die Formel durch die aktuell konstruierte Belegung bereits erfüllt sein. Eine Klausel, in der alle Literale auf 0 evaluieren, kann es hier nicht geben, da vor jeder Entscheidung sichergestellt wurde, dass die aktuelle Belegung widerspruchsfrei ist (und also die Formel nicht die leere Klausel enthalten kann). `Decide()` wird ebenfalls festlegen, welcher Wert der gewählten Variable zugewiesen wird und merkt sich, in welchem backtrack-level diese Entscheidung getroffen wurde (bzw. die wievielte Entscheidung dies war).
- `BCP()`: `BCP()` wendet Vereinfachungsregeln wie unit-propagation an, um die durch die aktuelle Belegung entstehenden direkten Folgerungen durch die Formel zu propagieren. Das backtrack-level für die abgeleiteten Belegungen entspricht dem Level der vorangegangenen Entscheidung. Es kann hierbei durchaus zu einem widerspruch (der leeren Klausel) kommen. Falls dem so ist, muss der aktuelle Konflikt analysiert werden. Falls kein Widerspruch auftritt, kann

mit einer neuen Entscheidung fortgefahren werden. Insbesondere enthält die Formel nicht die leere Klausel.

- `AnalyzeConflict()`: `AnalyzeConflict()` wird ausgeführt, wenn die `BCP()` Methode nach einer Entscheidung einen Widerspruch (die leere Klausel) feststellt. AnalyzeConflict() wird diesen Widerspruch mit Hilfe eines Konfliktgraphen analysieren und zwei Dinge entscheiden. Erstens, welche Entscheidungen zu diesem Konflikt geführt haben und in Zukunft zu vermeiden sind und zweitens, bis zu welchem backtrack-level die Entscheidungen rückgängig zu machen sind. Ferner wird `AnalyzeConflict()` eine Klausel lernen, die in Zukunft verhindert, dass wieder die selben widersprüchlichen Entscheidungen getroffen werden. Diese gelernte Klausel wird der Klauselmenge in der Formel hinzugefügt.

  Falls `AnalyzeConflict()` feststellt, dass bis zum -1 backtrack-level zurückgegangen werden muss, um den Konflikt aufzulösen, muss die Formel unerfüllbar sein da dies bedeutet, dass ein Widerspruch auftritt bevor überhaupt irgendeine Variable belegt wurde.

- `Backtrack(backtrack-level)`: `Backtrack()` wird gerufen, wenn eine neue Klausel gelernt wurde und der analysierte Widerspruch aufgelöst werden muss. Die geschieht dadurch, dass alle Entscheidungen und deren Ableitungen bis (ausschließlich) dem backtrack-level zurückgenommen werden. Das entsprechende backtrack-level ergibt sich aus der gelernten Klausel.

Im folgenden wird erklärt, wie Konfliktgraphen dazu verwendet werden, einen Konflikt zu analysieren, eine Klausel daraus zu lernen und mit Hilfe dieser Klausel zu entscheiden, bis zu welchem backtrack-level zurückgegangen werden muss.

3.8.2 Konfliktgraphen

Ein Konfliktgraph ist ein gerichteter Graph, der Knoten für die durch `Decide()` oder `BCP()` belegten Variablen enthält. Ein Knoten $v = (l, t(l))$ enthält dabei zwei Informationen: welche Variable hier wie belegt wurde (in Form eines Literals l, wobei $l = \neg x$ bedeutet, dass $\alpha(x) = 0$) und in welchem backtrack-level ($t(l)$) dies geschehen ist. Kanten in diesem Graph entsprechen Implikationen für Variablenbelegungen (siehe Abbildung 3.7). Hat ein Knoten mehrere direkte Vorgänger, so bedeutet dies, dass die Belegungen in allen diesen Vorgängern zusammen die Belegung für die Variable im Knoten bedingen.

Haben wir beispielsweise die Entscheidungen $\neg x_3, \alpha(x_3) = 0$ auf Level $t(x_3)$, $x_5, \alpha(x_5) = 1$ auf Level $t(x_5)$ und $x_8, \alpha(x_8) = 1$ auf Level $t(x_8)$ gefällt, die letztendlich zu einem Widerspruch führen, so wissen wir, dass

$$(\neg x_3 \wedge x_5 \wedge x_8) \text{ nicht gelten darf.}$$

Damit wissen wir, dass die Bedingung (Klausel)

$$(x_3 \vee \neg x_5 \vee \neg x_8)$$

erfüllt werden muss um den Konflikt zu vermeiden. Wir fügen diese Klausel also der Klauselmenge hinzu und lernen damit diese Klausel. Bleibt nun zu ermitteln, in welches backtrack-level wir zurückgehen müssen, um den aktuellen Konflikt aufzulösen. Wir wählen

$$\text{backtrack-level} = \max\{t(\neg x_3), t(x_5), t(x_8)\} - 1,$$

das Level genau vor der letzten zum Konflik führenden Entscheidung, und stellen damit sicher, dass wir mindestens eine der Bedingungen für den Konflikt zerstören. Weiterhin nehmen wir so wenig Entscheidungen wie möglich zurück, um den Suchfortschritt soweit es geht zu erhalten.

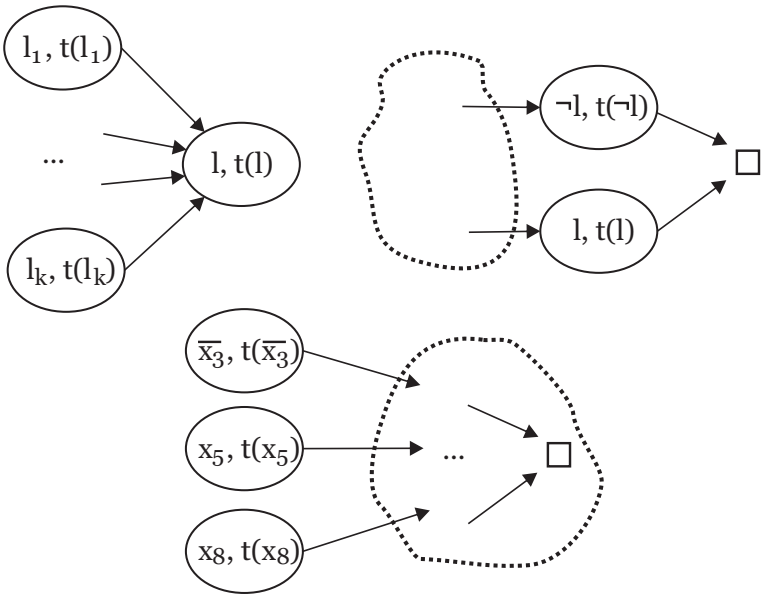

Abbildung 3.7: Ein Implikationsgraph (links oben) wird zu einem Konfliktgraph (rechts oben), sobald er einen Knoten für die leere Klausel enthält. Das Beispiel (unten) wird verwendet, um zu zeigen wie aus einem Konfliktgraph eine Klausel gelernt werden kann.

3.8.3 Beispiel für das Lernen einer Klausel

Angenommen, der iterative DPLL Solver sucht eine Lösung für eine Formel, die unter anderem folgende Klauseln enthält:

$F = \ldots \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3 \vee x_9) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_2 \vee x_5 \vee x_{10}) \wedge (x_1 \vee x_8) \wedge$
$(\neg x_4 \vee x_6 \vee x_{11}) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_1 \vee x_7 \vee x_{12}) \wedge (\neg x_7 \vee \neg x_8 \vee \neg x_{13}) \wedge \ldots$

Seien ferner bereits einige Entscheidungen (decisions) gefällt:

- Level 1: $\neg x_9$
- Level 2: x_{12} mit Implikation x_{13}
- Level 3: $\neg x_{10}$ mit Implikation $\neg x_{11}$
- Level 4,5: weitere (hier nicht relevante) Entscheidungen

In Abbildung 3.8 (oben) ist der Implikationsgraph bis ausschließlich Level 6 gezeigt. Falls nun in Level 6 die Entscheidung

- Level 6: x_1

gefällt wird, so hat dies aufgrund der Klauseln in F weitere Konsequenzen (siehe Abbildung 3.8 (unten)). Insbesondere tritt hier ein Konflikt auf. Es ist nun zu klären, was aus diesem Konflikt gelernt werden kann, so dass dieser Konflikt im weiteren Suchverlauf nicht wieder auftritt.

Die einfachste Möglichkeit etwas zu lernen besteht darin, sich die Entscheidungsknoten anzusehen, die zusammen den Konflikt implizieren. Im obigen Beispiel wären dies $\neg x_9, \neg x_{10}$ und x_6. Daraus ließe sich die Klausel $(x_9 \vee x_{10} \vee \neg x_6)$ lernen, allerdings kann es bei diesem einfachen Lernschema dazu kommen, dass die gelernten Klauseln sehr groß werden, und somit die Einschränkung, die sie für die Suche erbringen, sehr klein wird.

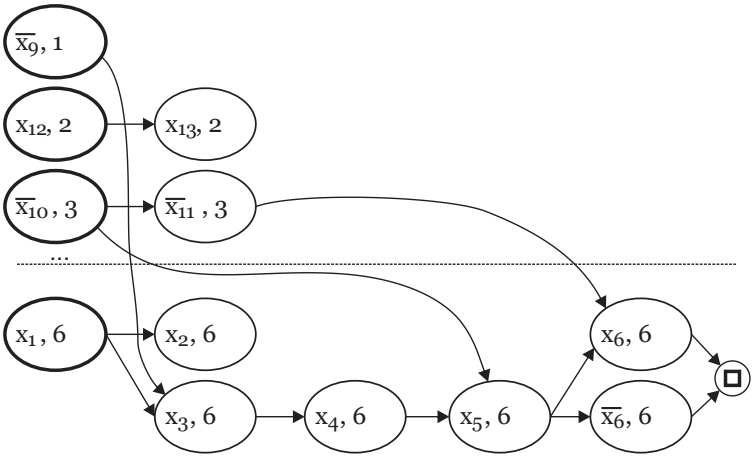

Abbildung 3.8: Ein Konfliktgraph.

Ein weiteres Lernschema (genannt FirstUIP) sucht ausgehend vom Knoten, der die leere Klausel repräsentiert, eine minimale Menge von im Graphen vorhandene Voraussetzungen für den Konflikt. Dies können sowohl Knoten mit getroffenen Entscheidungen sein (dicke Umrandung in Abbildung 3.8), als auch Unique Implication Points (UIPs). Ein UIP ist ein Knoten im Konfliktgraph, durch den alle Pfade einer bestimmten Voraussetzung hin zum Konflikt führen. In Abbildung 3.8 (unten) ist x_4 ein solcher UIP (für die Voraussetzungen $\neg x_9$ und x_1). Alle Vorgänger von x_4 können für die Zusammenstellung der minimalen Menge an Konfliktvoraussetzungen ignoriert werden. Zusammen mit x_{10} als Konfliktvoraussetzung ergibt sich die gelernte Klausel $(x_{10} \vee \neg x_4)$. Die durch das FirstUIP Schema gelernten Klauseln heißen auch *asserting clauses*. Diese asserting clauses haben bestimmte Vorteile. Zum einen sind sie möglichst klein, weswegen sie die Einschränkungen für die Suche maximieren. Zum anderen werden asserting clauses zu unit clauses kurz bevor der zu ihnen gehörende Konflikt auftritt. Dadurch leiten sie die Suche in eine andere Richtung, in der dieser Konflikt nicht auftritt.

Nehmen wir an, dass $(x_{10} \vee \neg x_4)$ als Klausel gelernt wurde. Nehmen wir weiterhin an, dass die ersten Entscheidungen aus Abbildung 3.8 (oben) wieder vorliegen. dann ist insbesondere die gelernte Konfliktklausel eine unit clause: $(\neg x_4)$. Diese impliziert für x_4 als Belegung eine 0. Durch die Klauseln in der Formel F folgt nun, dass x_1 ebenfalls auf 0 zu setzen ist, wodurch eine Voraussetzung für den gezeigten Konflikt zerstört wurde.

Bleibt zu klären, zu welcher Entscheidung man zurückkehren muss (Backtracking), nachdem eine Klausel mit FirstUIP gelernt wurde. Um dies zu beantworten muss man sich die gelernte Klausel $(x_{10} \vee \neg x_4)$ ansehen. Man ignoriert das Literal, welches aus dem UIP Knoten im Graph abgeleitet wurde (hier $\neg x_4$). Sodann betrachte man die in der Klausel enthaltenen Entscheidungen und geht zum größten Entscheidungslevel der in der Klausel enthaltenen Entscheidungen (hier hat x_{10} als einzige verbliebene Entscheidung das Entscheidungslevel 3). Wenn auf Level 3 zurückgesetzt wurde (wohl gemerkt ohne dieses zu verändern), ergibt sich durch die asserting clause, die hier immer eine unit clause sein muss, sofort eine Belegung, die den Konflikt vermeidet.

3.9 Ein randomisierter DPLL-Algorithmus (nach Paturi, Pudlak und Zane)

Sei im folgenden $\mathcal{S}_n$ die symmetrische Gruppe über n Elementen (die Menge aller Permutationen, die mit n Elementen möglich sind). Sei $\pi \in \mathcal{S}_n$ eine Permutation.

Der folgende SAT Solver wählt eine solche Permutation und nimmt diese als Reihenfolge, um die Variablen nacheinander zu belegen. Dabei wird die unit propagation angewendet, wenn eine zu belegende Variable allein in einer Klausel vorkommt ($(x_{\pi(i)})$ bzw. $(\neg x_{\pi(i)})$). Falls die unit propagation Regel nicht angewendet werden kann, wird die gewählte Variable $x_{\pi(i)}$ zufällig mit 0 oder 1 belegt.

Listing 3.7: Randomized-DPLL

```
Randomized−DPLL(F:k−KNF Formel){
    Wähle π ∈_R S_n;
    α = ∅;

    for (i = 1; i < n; i++){
        if (α erfüllt F){
            return SAT;
        }
        if (Fα enthält die unit Klausel (x_π(i))){
            α = α ∪ {x_π(i) ← 1};
        } else if (Fα enthält die unit Klausel (¬x_π(i))){
            α = α ∪ {x_π(i) ← 0};
        } else {
            α = α ∪ {x_π(i) ←_R {0,1}};
        }
    }
}
```

Der oben beschriebene Algorithmus kann wie folgt analysiert werden.

Definition 28 *Sei α^* eine erfüllende Belegung für F. Variable x_i heißt kritisch (in Bezug auf α^*), falls $F|_{\alpha^* \leftarrow \neg\alpha^*} = 0$. Wenn x_i für α^* kritisch ist, so gibt es eine Klausel $K \in F$ mit $x_i \in K$ oder $\neg x_i \in K$, so dass das entsprechende Literal allein die Klausel wahr macht. Wir nennen ein solches K eine kritische Klausel.*

Sei S die Menge aller erfüllenden Belegungen für F. Sei $\alpha \in S$. Wir nennen x_i kritisch für S, wenn $\alpha|_{x_i \leftarrow \neg x_i} \notin S$.

Sei nun $j(\alpha^*)$ die Anzahl der kritischen Variablen in α^* in Bezug auf S (d.h. das ändern der Belegung einer solchen Variable führt aus S heraus). Es ist $0 \leq j(\alpha^*) \leq n$. Sei zusätzlich $s(\alpha^*) = n - j(\alpha^*)$ (die Anzahl der nicht-kritischen Variablen in α^* in Bezug auf S).

Lemma 5 *Sei $S \subseteq \{0,1\}^*, |S| > 0$, die Menge der erfüllenden Belegungen von F. Sei $\alpha \in S$. Dann gilt:*

$$\sum_{\alpha \in S} 2^{-s(\alpha)} \geq 1.$$

Beweis 19 *Beweis durch Induktion nach n.*
Induktionsanfang: *Im Fall $n = 1$, klar.*
Induktionshypothese: *Es gelte für $n-1$: $\sum_{\alpha \in S} 2^{-s(\alpha)} \geq 1$.*
Induktionsschritt:
Seien $S_0 = \{y \in \{0,1\}^{n-1} | y0 \in S\}$ und $S_1 = \{y \in \{0,1\}^{n-1} | y1 \in S\}$.
Den Mengen S_i seine entsprechende Funktionen s_i zugeordnet, die sich wie s in Bezug auf S verhalten.
Nach Induktionsvoraussetzung gilt: $\sum_{y \in S_i} 2^{-s_i(y)} \geq 1$.

Wegen $0 < |S| = S_0 + S_1$ können nicht beide Mengen S_1, S_2 leer sein. Wir unterscheiden für den Induktionsschritt nun die beiden Fälle, dass beide Mengen nicht leer sind und dass genau eine der beiden Mengen leer ist.

Falls beide Mengen nicht leer sind:

$$\sum_{\alpha \in S} 2^{-s(\alpha)} = \sum_{y \in S_0} 2^{-s_0(y0)} + \sum_{y \in S_1} 2^{-s_1(y1)} \tag{3.1}$$

$$\geq \sum_{y \in S_0} 2^{-s_0(y)-1} + \sum_{y \in S_1} 2^{-s_1(y)-1} \tag{3.2}$$

$$\overset{IV}{\geq} \frac{1}{2} + \frac{1}{2} = 1 \tag{3.3}$$

Falls o.B.d.A. nur S_0 nicht leer ist:

$$\sum_{\alpha \in S} 2^{-s(\alpha)} = \sum_{y \in S_0} 2^{-s(y0)} = \sum_{y \in S_0} 2^{-s_0(y)} \geq 1 \tag{3.4}$$

□

Bemerkung 17 *Sei $\pi \in \mathcal{S}_n$. Sei $\alpha \in S$. Die Wahrscheinlichkeit, dass π die kritische Variable einer kritischen Klausel als letztes aufzählt, ist bei k-SAT $\frac{1}{k}$.*

Beispiel 1 *Sei*

$$F = \ldots \wedge (x_1 \vee x_5 \vee \neg x_7) \wedge (x_1 \vee \neg x_8 \vee \neg x_4) \wedge \ldots$$

mit x_5 und x_4 als kritische Variablen in den gezeigten Klauseln. Sei ferner

$$\sigma = (7, 8, 4, 1, 5, \ldots)$$

eine zufällig gezogene Permutation aus $\mathcal{S}_n$. Die Variablen x_1 und x_7 werden unter dieser Permutation zuerst belegt. Für den Fall, dass die entsprechenden Literale auf 0 gesetzt werden (was in diesem Fall richtig sein muss, denn sonst wäre x_5 nicht kritisch), führt dies zum anwenden der unit-propagation-Regel auf die kritische Variable x_5. Diese wird dann zwangsweise auch korrekt belegt. In anderen Worten: sind die nicht-kritischen Literale eine Klausel richtig belegt, so wird auch das kritische Literal dieser Klausel richtig belegt.

Die Permutation σ zählt jedoch die kritische Variable x_4 vor x_1 auf. Für die zweite Klausel bedeutet dies, dass die unit-propagation-Regel hier nicht dazu verwendet werden kann, die kritische Variable korrekt zu belegen. Entweder wird die Klausel erfüllt, weil die kritische Variable x_4 zufällig korrekt belegt wird, oder die konstruierte Belegung wird auf jeden Fall falsch sein, da dass setzen der kritischen Variable auf einen falschen Wert die Belegung insgesamt aus der Menge der erfüllenden Belegungen herausführt.

Definition 29 *Für $\sigma \in \mathcal{S}_n$ sei $r(\alpha, \sigma)$ die Anzahl der kritischen Variablen von $\alpha \in S$, die als letzte in ihrer kritischen Klausel durch σ aufgezählt werden. Es ist $0 \leq r(\alpha, \sigma) \leq j(\alpha) \leq n$.*

Die strategie für die Analyse von `Randomized-DPLL` ist wie folgt.

- Wir berechnen zuerst die Wahrscheinlichkeit, dass der Algorithmus ein gegebenes α unter gegebenem σ findet (dies wird von $r(\alpha, \sigma)$ abhängen).
- Dann errechnen wir den Erwartungswert von $r(\alpha, \sigma)$ für $\mathcal{S}_n$
- Danach wird errechnet, wie groß die Wahrscheinlichkeit ist, dass der Algorithmus ein bestimmtes $\alpha \in S$ findet, ohne Einschränkung auf ein bestimmtes σ.

- Zuletzt errechnen wir die Wahrscheinlichkeit, wie groß die Wahrscheinlichkeit ist, dass der Algorithmus irgendein $\alpha \in S$ findet.

$$P(\text{Alg. findet } \alpha | \pi = \sigma) = 2^{-(n-r(\alpha,\sigma))} \tag{3.5}$$
$$= 2^{-n+r(\alpha,\sigma)} \tag{3.6}$$

Der Erwartungswert für r kann unter Verwendung der Jensen-Ungleichung, $E(2^X) \geq 2^{E(X)}$, wie folgt abgeschätzt werden:

$$\frac{1}{n!} \sum_{\alpha \in \mathcal{S}_n} r(\alpha, \sigma) \geq \frac{j(\alpha)}{k} \tag{3.7}$$

Daraus folgt die Wahrscheinlichkeit, dass der Algorithmus ein festes α für irgendeine Permutation σ findet.

$$P(\text{Alg. findet festes } \alpha \in S) = \sum_{\sigma \in \mathcal{S}_n} P(\text{Alg. findet } \alpha | \pi = \sigma) \cdot P(\pi = \sigma) \tag{3.8}$$
$$= \sum_{\sigma \in \mathcal{S}_n} 2^{-n+r(\alpha,\sigma)} \cdot \frac{1}{n!} \tag{3.9}$$
$$= 2^{-n} \cdot \left(\frac{1}{n!} \sum_{\sigma \in \mathcal{S}_n} 2^{r(\alpha,\sigma)} \right) \tag{3.10}$$
$$\overset{\text{Jensen}}{\geq} 2^{-n} \cdot 2^{\frac{1}{n!} \sum_{\sigma \in \mathcal{S}_n} 2^{r(\alpha,\sigma)}} \tag{3.11}$$
$$\overset{3.7}{\geq} 2^{-n} \cdot 2^{\frac{j(\alpha)}{k}} \tag{3.12}$$

Daraus folgt die Wahrscheinlichkeit, dass der Algorithmus irgendeine erfüllende Belegung findet.

$$P(\text{Alg. findet irgendeine erf. Belegung}) = \sum_{\alpha \in S} Pr(\text{Alg. findet } \alpha) \tag{3.13}$$
$$\overset{3.12}{\geq} \sum_{\alpha \in S} 2^{-n} \cdot 2^{\frac{j(\alpha)}{k}} \tag{3.14}$$
$$= 2^{-n \cdot (\frac{n}{k})} \sum_{\alpha \in S} 2^{-\frac{n-j(\alpha)}{k}} \tag{3.15}$$
$$\geq 2^{-n(1-\frac{1}{k})} \sum_{\alpha \in S} 2^{-\overbrace{(n-j(\alpha))}^{=s(\alpha)}} \tag{3.16}$$
$$\overset{\text{Lemma 3}}{\geq} 2^{-n(1-\frac{1}{k})} \tag{3.17}$$

Setzt man nun in Gleichung 3.17 entsprechende k-Werte ein, ergibt sich eine Laufzeit wie in Tabelle 3.4 gezeigt.

$k = 3$	4	5	6	7	8
$2^{-n(1-\frac{1}{k})}$ $= 1,587^n$	$1,681^n$	$1,741^n$	$1,781^n$	$1,811^n$	$1,830^n$

Tabelle 3.4: Laufzeit von `Randomized-DPLL` für verschiedene k.

Eine Verbesserung des oben beschriebenen `Randomized-DPLL` Algorithmus ist durch einen Vorverarbeitungsschritt gegeben, den Paturi, Pudlak, Sachs und Zane (1998) vorgestellt haben. Man erzeugt vor aufrufen des `Randomized-DPLL` zunächst alle Resolventen, deren Größe $c \cdot \log n$ nicht überschreitet, und fügt diese der Formel hinzu. Danach wird `Randomized-DPLL` gerufen. Da die Analyse jedoch sehr aufwändig ist, werden wir sie hier nicht weiter erörtern.

3.10 Einschub: 3-SAT Algorithmus von Pudlak (divide+conquer)

Der 3-SAT Algorithmus von Pudlak teilt zunächst die Variablenmenge $\{x_1, \ldots, x_n\}$ willkürlich in zwei gleichgroße Mengen auf. Sei dies hier gegeben mit $V_1 = \{x_1, \ldots, x_{\frac{n}{2}}\}$ und $V_2 = \{x_{\frac{n}{2}+1}, \ldots, x_n\}$.

Sodann gruppiert man die Klauseln in drei verschiedene und disjunkte Mengen:

- $\mathcal{K}_1$: Alle Klauseln, die nur Variablen aus V_1 enthalten.
- $\mathcal{K}_2$: Alle Klauseln, die nur Variablen aus V_2 enthalten.
- $\mathcal{K}_3$: Alle Klauseln, die sowohl Variablen aus V_1 als auch V_2 enthalten. Wir unterscheiden hier $\mathcal{K}_3 = \mathcal{K}_3^{12} \cup \mathcal{K}_3^{21}$, wobei $K \in \mathcal{K}_3^{ij}$ bedeutet, dass K i Literale aus V_1 und j Literale aus V_2 enthält.

Es ist nun $F = \mathcal{K}_1 \cup \mathcal{K}_2 \cup \mathcal{K}_3$.

Man sucht nun alle erfüllenden Belegungen für die Klauseln in $\mathcal{K}_1$ über V_1, und speichere diese in die A-Liste. Des weiteren sucht man alle erfüllenden Belegungen für $\mathcal{K}_2$ über V_2 und speichere diese in die B-Liste. Der Aufwand hierfür liegt bei $O^*(2^{\frac{n}{2}})$.

Man prüft nun alle $\alpha \in A$-Liste, indem man die Klauseln $K \in \mathcal{K}_3^{21}$ inspiziert. Ist ein solches K unter α unit, so erweitert man das α mit der unit-propagation Regel. Finden sich keine weiteren unit Klauseln für das α (kann dieses also nicht mehr erweitert werden), prüft man, ob eventuell eine leere Klausel entstanden ist. Ist dies nicht der Fall speichert man das erweiterte α in die A^*-Liste. Ansonsten wird das erweiterte α verworfen und nicht in die A^*-Liste übernommen.

Analog konstruiert man die B^*-Liste mit Klauseln aus $\mathcal{K}_3^{12}$.

Kann man nun ein Matching zwischen der A^*-Liste und der B^*-Liste identifizieren, hat man eine erfüllende Belegung für ganz F gefunden. Es gilt:

$$F \text{ erfüllbar} \Leftrightarrow \exists \text{ match } \alpha \in A^*, \beta \in B^*.$$

Beispiel 2 *Sei F eine erfüllbare Formel mit $n = 8$. Sei ferner $V_1 = \{x_1, \ldots, x_4\}$ und $V_2 = \{x_5, \ldots, x_8\}$.*
Sei $A^* = \{0010 * 01*, 1101 * *0*\}, B^* = \{*10 * 1100, 0 * * * 0100\}$.
Es findet sich ein Matching zwischen $1101 * *0* \in A^*$ und $*10*1100 \in B^*$: 11011100.
Dies ist für dieses Beispiel eine erfüllende Belegung.

Das finden eines Matchings benötigt, bei naiver Implementierung, $O^*(2^{\frac{n}{2}} + 2^{\frac{n}{2}}) = O^*(2^n)$. Verwendet man jedoch zum finden eines Matchings optimierte Algorithmen, so lässt sich die Laufzeit verbessern. Genauer: In Abhängigkeit der nicht belegten Variablen in α, β (bezeichnet mit $*$), kommen zwei verschiedene Algorithmen zum tragen.

Der erste Algorithmus ist schneller, wenn α oder β wenige unbelegte ($*$) Variablen enthält. Wenig bedeutet hier höchstens $(1 - \epsilon)\frac{n}{2}, \epsilon \approx 0.22$ viele.

Der zweite Algorithmus ist schneller, wenn α und β viele (nicht wenige) unbelegte Variablen enthalten. Im folgenden sind die beiden Algorithmen genauer beschrieben.

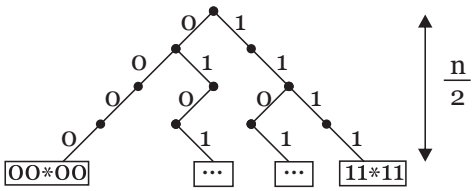

Abbildung 3.9: Eine Liste als Baum gespeichert fasst bestimmte Elemente in einem Knoten zusammen.

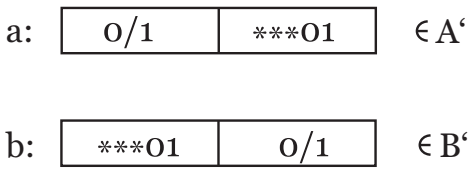

Abbildung 3.10: Die Elemente aus den A'- und B'-Listen.

- **Algorithmus 1:** Sei $a \in A^*$ mit $\#*$ in a höchstens $(1-\epsilon)\frac{n}{2}$. Man ersetzt nun systematisch alle $*$'e durch 0/1 (insgesamt entstehen so $2^{(1-\epsilon)\frac{n}{2}}$ viele Ausprägungen von a). Ist B^* als Baumstruktur gegeben, kann man nun durch ablaufen eines Pfades von der Wurzel bis zu einem Blatt dieses Baumes feststellen, ob es ein Matching zwischen der Ausprägung des a und einem $b \in B^*$ gibt (siehe Abbildung 3.9).

  Der Aufwand für ein $a \in A^*$ (mit wenigen unbelegten Variablen) ist $O^*(2^{(1-\epsilon)\frac{n}{2}})$. Mit maximal $2^{\frac{n}{2}}$ vielen Elementen in der A^*-Liste ist der Aufwand für das finden eines Matchings insgesamt $O^*(2^{\frac{n}{2}} \cdot 2^{(1-\epsilon)\frac{n}{2}}) = O^*(2^{n(1-\epsilon)})$.

  Analog lässt sich die Suche für ein $b \in B^*$ (mit wenig $*$ Einträgen) gestalten.

- **Algorithmus 2:** Seien $A' \subseteq A^*$ und $B' \subseteq B^*$ diejenigen a's bzw. b's, mit vielen $*$ Einträgen (siehe Abbildung 3.10).

  Wähle aus jedem $a \in A'$, in der vorderen Hälfte systematisch bis zu $\frac{n}{2}\epsilon$ viele Positionen aus. Belasse diese Positionen (als 0/1). Die anderen, nicht ausgewählten Positionen werden auf $*$ gesetzt. Die so abgeänderten a werden in die $\hat{A}$-Liste eingefügt.

  Es gilt: $|\hat{A}| = |A'| = \sum_{i=0}^{\frac{n}{2}\epsilon} \binom{\frac{n}{2}}{i} \leq |A'| \cdot 2^{h(\epsilon)\frac{n}{2}} \leq 2^{\frac{n}{2}(1+h(\epsilon))}$.

  Ferner gilt: F erfüllbar $\Leftrightarrow \exists a \in \hat{A}, b \in \hat{B} : a = b \Leftrightarrow \hat{A} \cap \hat{B} \neq \emptyset$.

  Um herauszufinden, ob es solch ein a bzw. b gibt, sortiert man $\hat{A} \cup \hat{B}$ (bestehend aus m Elementen) und überprüft, ob Duplikate vorhanden sind. Die Laufzeit für das sortieren ist $O(m \underbrace{\log n}_{\text{polynomial in } n}) = O^*(2^{\frac{n}{2}(1+h(\epsilon))})$.

  Der beste "Kompromiss" für ϵ ist dann:

$$2^{\frac{n}{2}(1+h(\epsilon))} \stackrel{!}{=} 2^{n(1-\frac{\epsilon}{2})} \tag{3.18}$$

$$\Leftrightarrow \frac{1}{2}(1+h(\epsilon)) = 1 - \frac{\epsilon}{2} \tag{3.19}$$

$$\Leftrightarrow h(\epsilon) = 1 - \epsilon \tag{3.20}$$

  Numerisch ist somit $\epsilon = 0.22709$ (siehe Abbildung 3.11). Mit $\epsilon = 0.22709$ ist die Laufzeit insgesamt $O^*(1,8487^n)$.

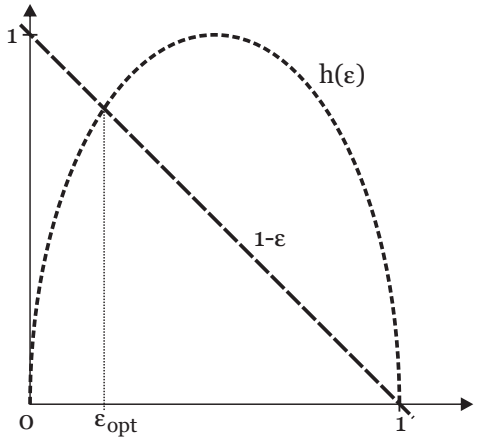

Abbildung 3.11: Über den Schnitt der beiden Funktionen lässt sich ein optimaler Wert für ϵ bestimmen.

Kapitel 4

Lokale Suche

Während bei DPLL SAT Solvern versucht wird, konstruktiv ein Modell zu finden (ausgehen von der leeren Belegung), versuchen Lokale Suche Algorithmen, ausgehend von einer vollständigen Belegung, ein Modell in der Nachbarschaft dieser vollständigen Belegung zu finden. Die "Nachbarschaftsbeziehung" ist gegeben über die Hamming-Distanz zweier Belegungen. Genauer: Seien α und β vollständige Belegungen, dann ist die Hamming-Distanz zwischen diesen Belegungen $d(\alpha, \beta)$ gerade die Anzahl an Variablen, die durch α und β *nicht* den selben Wert zugewiesen bekommen.

Der folgende Algorithmus vollführt eine lokale Suche um eine vollständige Belegung α, bei der für die durchsuchten Nachbarn von α, genannt $\alpha*$, gilt $d(\alpha, \alpha*) \leq w$.

Listing 4.1: Lokale Suche

```
LS(F:k−KNF Formel, α vollst. Belegung, w ∈ ℕ){
    // Liefert 1, gdw. ∃α*, d(α, α*) ≤ w : Fα* = 1.
    if (α ⊨ F) return 1;
    if (w = 0) return 0;

    Sei K = {l_1, ..., l_r}, r ≤ k, eine kürzeste Klausel in F, mit Kα = 0.

    for (i = 1; i ≤ r; i++){
        if (LS(F, α |_{l_i ← 1}, w − 1) == 1) return 1;
    }

    return 0;
```

Hierbei bezeichnet $\alpha \mid_{l_i \leftarrow 1}$ das Abändern der Belegung α derart, dass Literal l_i erfüllt wird (in der ursprünglich gewählten Klausel K galt ja $K\alpha = 0$, und somit gilt für alle Literale $l_i \in K$ gerade auf $\alpha \not\models l_i$).

Die Laufzeit von `LS` ist zunächst gegeben durch $O^*(k^w)$ (z.B. mit $k = 3$). Durch verschiedene Anwendungen von `LS` kann dies jedoch verbessert werden. Diese Anwendungen greifen auf Überdeckungscodes zurück, die wir im folgenden erläutern wollen.

4.1 Überdeckungscodes und ihre Konstruktion

4.1.1 Überdeckungscodes

Wir geben zunächst folgende Definition:

Definition 30 *$C = \{\alpha_1, \ldots, \alpha_t\}$ heißt Überdeckungscode mit Radius r, falls*
$$\{0,1\}^n = \bigcup_{i=1}^{t} H(\alpha_i, r),$$

wobei $H(\alpha_i, r) = \{\alpha' | \alpha' \in \{0,1\}^n, d(\alpha_i, \alpha') \leq r\}$ *und mit* $a_i, b_i \in \{0,1\}$ $d(\alpha, \beta) = d(a_1, \ldots, a_n, b_1, b_n) = \sum_{i=1}^{n} (a_i - b_i)^2$. $H(\alpha_i, r)$ *wird auch als Hamming-kugel bezeichnet.*

Zusätzlich wird im folgenden die Shannon-Entropie-Funktion h verwendet. Es ist

$$h(x) = -x \log_2(x) - (1-x) \log_2(1-x).$$

Wir geben zunächst eine *obere Schranke* für die Größe von Hammingkugeln an. Es gilt: $|H(\alpha_i, \lambda n)| = \sum_{k=0}^{\lambda \cdot n} \binom{n}{k} \leq 2^{h(\lambda) \cdot n}$ (für $\lambda < 0.5$), denn

$$\begin{aligned}
1 = (\lambda + (1-\lambda))^n &= \sum_{k=0}^{n} \binom{n}{k} \lambda^k (1-\lambda)^{nk} && (4.1)\\
&\geq \sum_{k=0}^{\lambda n} \binom{n}{k} \lambda^k (1-\lambda)^{nk} && (4.2)\\
&= \left(\sum_{k=0}^{\lambda n} \binom{n}{k} \left(\frac{\lambda}{1-\lambda} \right)^k \right) \cdot (1-\lambda)^n && (4.3)\\
&\geq (1-\lambda)^n \cdot \left(\frac{\lambda}{1-\lambda} \right)^{\lambda n} \cdot \sum_{k=0}^{\lambda n} \binom{n}{k} && (4.4)\\
& && (4.5)
\end{aligned}$$

$$\begin{aligned}
\Longrightarrow \sum_{k=0}^{\lambda n} \binom{n}{k} &\leq \left(\frac{1}{\lambda} \right)^{\lambda n} \cdot \left(\frac{1}{1-\lambda} \right)^{(1-\lambda)n} && (4.6)\\
&= 2^{-\lambda n \log_2(n) - (1-\lambda) n \log_2(1-\lambda)} && (4.7)\\
&= 2^{h(\lambda) n} && (4.8)
\end{aligned}$$

Ferner gilt:

$$2^{h(\lambda)n} = \left(\left(\frac{1}{\lambda} \right)^{\lambda} \cdot \left(\frac{1}{1-\lambda} \right)^{1-\lambda} \right)^n \tag{4.9}$$

Für die *untere Schranke* für die Größe von Hammingkugeln gilt: $|H(\alpha_i, \lambda n)| = \sum_{k=0}^{\lambda \cdot n} \binom{n}{k} \geq \frac{1}{n+1} \cdot 2^{h(\lambda) n}$, denn

$$\begin{aligned}
1 = (\lambda + (1-\lambda))^n &= \sum_{k=0}^{n} \underbrace{\binom{n}{k} \lambda^k \cdot (1-\lambda)^{n-k}}_{\Pr(X=k) \text{ für } (n,\lambda)\text{-binomialverteilte Zufallsvariable}} && (4.10)\\
&\leq (n+1) \cdot \max_k \left\{ \binom{n}{k} \cdot \lambda^k \cdot (1-\lambda)^{n-k} \right\} && (4.11)\\
&= (n+1) \cdot \binom{n}{\lambda n} \cdot \lambda^{\lambda n} \cdot (1-\lambda n)^{(1-\lambda)n} && (4.12)\\
&\overset{4.9}{=} (n+1) \cdot \binom{n}{\lambda n} \cdot 2^{-h(\lambda) \cdot n} && (4.13)
\end{aligned}$$

$$\Rightarrow \sum_{k=0}^{\lambda n} \binom{n}{k} \geq \binom{n}{\lambda n} \geq \frac{1}{n+1} \cdot 2^{h(\lambda) n} \tag{4.14}$$

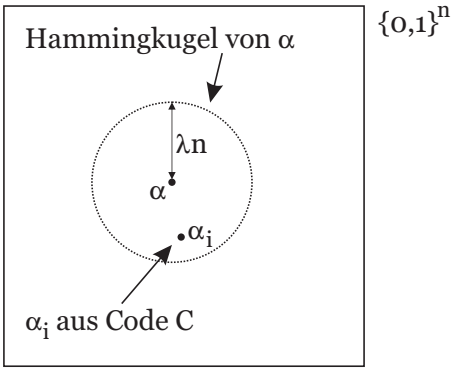

Abbildung 4.1: Ein zufällig gezogenes α liegt in einer Hammingkugel $H(\alpha_i, \lambda n)$ von α_i, wenn umgekehrt dieses α_i auch in der Hammingkugel von α liegt.

D. h. wenn polynomielle Faktoren ignoriert werden, gilt

$$|H(\alpha_i, \lambda n)| = \sum_{k=0}^{\lambda n} \binom{n}{k} \approx \binom{n}{\lambda n} \approx 2^{h(\lambda)n}, 0 \leq \lambda \leq 0,5 \tag{4.15}$$

Dies wird später bei der Laufzeitanalyse von `LS` verwendet. Hierbei bedeutet $f(n) \approx g(n)$ gdw. für Polynom p gilt:

$$\frac{1}{p(n)} g(n) \leq f(n) \leq p(n) g(n).$$

also: $\binom{n}{\lambda n} \approx 2^{(h(\lambda)n}$ für $0 \leq \lambda \leq 1$.

4.1.2 Die Konstruktion von Überdeckungscodes

Im folgenden wird gezeigt, wie man einen Überdeckungscode $\{\alpha_1, \ldots, \alpha_t\} = C \subseteq \{0,1\}^n$ mit Radius $r = \lambda \cdot n$ konstruieren kann. Durch die Hamming-Schranke ist bekannt:

$$|C| \geq \frac{2^n}{|H(\alpha_i, \lambda n)|} \tag{4.16}$$

$$= \frac{2^n}{\sum_{i=0}^{\lambda n} \binom{n}{i}} \tag{4.17}$$

$$\overset{4.15}{\approx} 2^{n \cdot (1 - h(\lambda))} \tag{4.18}$$

Wähle nun einen zufälligen Code $C \subseteq \{0,1\}^n$, so dass

$$|C| = \frac{n \cdot 2^n}{\sum_{i=0}^{\lambda n} \binom{n}{i}} \tag{4.19}$$

Wir wollen nun zeigen, dass dieser zufällig gewählte Code C mit hoher Wahrscheinlichkeit ein Überdeckungscode ist, wenn n groß genug gewählt ist. In anderen Worten: wir wollen zeigen, dass $Pr(\text{C kein Überdeckungscode}) = 0 (n \to \infty)$. Sei dazu $\alpha \in \{0,1\}^n$ beliebig gewählt. Wir errechnen nun, mit welcher Wahrscheinlichkeit ein zufällig gezogenes (Codewort) $\alpha_i \in C$ in der Hammingkugel von α liegt. Umgekehrt liegt dann auch α in der Hammingkugel von α_i (siehe Abbildung 4.1).

$$Pr(\alpha_i \text{ liegt in Hammingkugel von } \alpha) = \tag{4.20}$$

$$Pr(\alpha_i \in H(\alpha, \lambda n)) = Pr(\alpha \in H(\alpha_i, \lambda n)) \tag{4.21}$$

$$= \frac{\sum_{i=0}^{\lambda n} \binom{n}{i}}{2^n} \tag{4.22}$$

Daraus folgt unmittelbar die Wahrscheinlichkeit, dass α nicht von C überdeckt wird:

$$Pr(\alpha \text{ wird von keinem } \alpha_i \in C \text{ überdeckt}) = \left(1 - \frac{\sum_{i=0}^{\lambda n} \binom{n}{i}}{2^n}\right)^{|C|} \tag{4.23}$$

$$\overset{1-x \leq e^{-x}}{\leq} e^{-\frac{\sum_{i=0}^{\lambda n} \binom{n}{i}}{2^n} \cdot \frac{n \cdot 2^n}{\sum_{i=0}^{\lambda n} \binom{n}{i}}} \tag{4.24}$$

$$= e^{-n} \tag{4.25}$$

Wir errechnen daraus die Wahrscheinlichkeit, dass C kein Überdeckungscode ist:

$$Pr(C \text{ kein Überdeckungscode}) = Pr(\exists \alpha : \alpha \notin H(\alpha_i, \lambda n) \forall \alpha_i \in C) \tag{4.26}$$

$$\leq \sum_{\alpha \in \{0,1\}^n} Pr(\alpha \text{ wird von keinem } \alpha_i \in C \text{ überdeckt}) \tag{4.27}$$

$$\leq 2^n \cdot e^{-n} = \left(\frac{2}{e}\right)^n \tag{4.28}$$

$$\to 0 (n \to \infty) \tag{4.29}$$

Die probabilistische Konstruktion der Überdeckungscodes, die wir oben beschrieben haben, erweckt den Anschein, dass jeder Algorithmus, der solche Codes verwendet, ein probabilistischer Algorithmus ist. Wir wollen im folgenden noch zeigen, wie Überdeckungscodes auf deterministische Weise konstruiert werden können.

Lemma 6 *Sei C_1 ein Überdeckungscode der Länge n_1 mit Radius r_1. Sei C_2 ein Überdeckungscode der Länge n_2 mit Radius r_2.*
Dann ist $C_1 \times C_2 = \{\alpha_1 \alpha_2 : \alpha_1 \in C_1, \alpha_2 \in C_2\}$ ein Überdeckungscode der Länge $n_1 + n_2$ mit Radius $r_1 + r_2$.

Beweis 20 *Klar.*

Finde nun einen Überdeckungscode C_1 der Länge n_0 (n_0 ist im folgenden fest). Dieser Code muss nur einmal erzeugt werden. Ist nun ein n gegeben (durch die Anzahl der Variablen in einer Formel, die man mit Hilfe von Überdeckungscodes lösen will), konstruiert man einen passenden Überdeckungscode wie folgt:

$$C = \underbrace{C_1 \times \ldots \times C_1}_{\frac{n}{n_0} \text{ mal}} \tag{4.30}$$

Die oben gezeigte Konstruktion ist deterministisch und führt in einen Algorithmus, der diese Codes verwendet, keinen Probabilismus ein. Das Problem hierbei ist Folgendes: **weiterer Text, hier fehlt noch was**.

4.2 Die Verwendung von Überdeckungscodes zu Verbesserung der Lokalen Suche

Wir zeigen im Folgenden die Anwendung von zwei verschiedenen Codes zur Verbesserung der lokalen Suche.

- **Anwendung 1.** Konstruiere den Code C der Länge n und Radius $\frac{n}{2}$, mit
  $$C = \{\{x_1 \leftarrow 0, \ldots, x_n \leftarrow 0\}, \{x_1 \leftarrow 1, \ldots, x_n \leftarrow 1\}\}.$$
  Dies ist ein Überdeckungscode. Vollführe für die Suche zwei Aufrufe:
  $$\text{LS}(F, \{x_1 \leftarrow 0, \ldots, x_n \leftarrow 0\}, \tfrac{n}{2}) \text{ und } \text{LS}(F, \{x_1 \leftarrow 1, \ldots, x_n \leftarrow 1\}, \tfrac{n}{2})$$

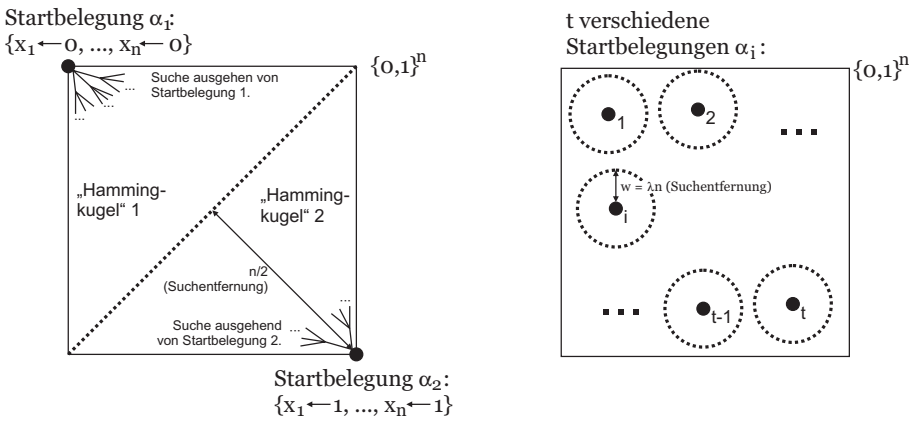

Abbildung 4.2: (Abbildung links) Die Überdeckung des Suchraums $\{0,1\}^n$ durch zwei Teile, die separat durch LS betrachtet werden. Die Suche startet dann einmal von $\alpha_1 = \{x_1 \leftarrow 0, \ldots, x_n \leftarrow 0\}$ und einmal von $\alpha_2 = \{x_1 \leftarrow 1, \ldots, x_n \leftarrow 1\}$. In beiden Fällen darf sich die lokale Suche maximal $n/2$ Schritte vom Ausgangspunkt entfernen.(Abbildung rechts) Durch eine geschicktere Überdeckung des Suchraums durch t verschiedene Teile, kann die Suchzeit verringert werden. Die Suche darf sich dann maximal $w = \lambda \cdot n$ Schritte weit von der jeweiligen Startbelegung α_i entfernen.

(siehe Abbildung 4.2, links). Damit deckt die Suche den gesamten Suchraum ab, die Laufzeit beträgt nun aber (im Falle $k = 3$) $O^*(2 \cdot k^{\frac{n}{2}}) = O^*(3^{\frac{n}{2}}) = O^*(1,76^n)$, was besser ist als die bisher angegebene Laufzeit $O^*(2^n)$.

- **Anwendung 2.** Durch die Verwendung eines besseren Überdeckungscodes kann die Suche erneut beschleunigt werden. Bestimme einen Überdeckungscode
$$C = \{\alpha_1, \ldots, \alpha_t\}, \text{ so dass } \{0,1\}^n = \bigcup_{i=1}^{n} H(\alpha_i, w)$$
für ein geeignetes w mit $w = \lambda \cdot n$. Der Parameter t (die Anzahl der verschiedenen Hammingkugeln, mit denen wir den Suchraum überdecken) ist hierbei noch offen.

  Idealerweise: $H(\alpha_i, w) \cap H(\alpha_j, w) = \emptyset \forall i \neq j$ (perfekter Code und damit perfekte Überdeckung des Suchraums).

  Es gilt $\sum_{i=0}^{\lambda \cdot n} \binom{n}{i} \approx 2^{h(\lambda) \cdot n}$ (für $\lambda < 0.5$, siehe Seite 47). Und damit:

$$t = \frac{2^n}{|H(\alpha_i, w)|} = \frac{2^n}{|H(\alpha_i, \lambda \cdot n)|} \tag{4.31}$$
$$= \frac{2^n}{\sum_{i=0}^{\lambda \cdot n} \binom{n}{i}} \tag{4.32}$$
$$\approx \frac{2^n}{2^{h(\lambda) \cdot n}} \tag{4.33}$$
$$= 2^{n(1-h(\lambda))} \tag{4.34}$$

  Der Suchraum ist nun von t Hammingkugeln überdeckt, wobei t noch vom bisher unbestimmten Parameter λ abhängt. Die Lokale Suche wird dann für 3-SAT jede der t Hammingkugeln mit Aufwand $3^{\lambda \cdot n}$ durchsuchen (siehe Abbildung 4.2, rechts).

Laufzeit insgesamt (für 3 SAT):

$$O^*(t \cdot k^{\lambda \cdot n}) = O^*(2^{n(1-h(\lambda))} \cdot 3^{\lambda \cdot n}) \tag{4.35}$$

$$= O^*([\underbrace{2^{1-h(\lambda)} \cdot 3^{\lambda}}_{f(\lambda)}]^n) \tag{4.36}$$

$$= O^*(1.5^n) \tag{4.37}$$

Man kann in 4.36 das Minimum von f bestimmen, um ein geeignetes λ für die Minimierung der Basis zu erhalten, indem man die Funktion $f(\lambda)$ nach λ ableitet, die Ableitung auf 0 setzt und dann nach λ auflöst. Das Minimum liegt bei $\lambda = 0.25$. Die optimierte Laufzeit von LS liegt bei $O^*(1.5^n)$.

Insgesamt erhalten wir den `Search`-Algorithmus, der mit Hilfe von verschiedenen Überdeckungscodes auf LS zurückgreift.

Listing 4.2: Lokale Suche 2

```
Search(F:k−KNF Formel){
    Setze λ = 0.25;
    Errechne t; //mit Hilfe von λ wie oben gezeigt

    //die optimale Zerlegung des Suchraums in Hammingkugeln
    Konstruiere C = {α_1, ..., α_t};

    for (α_i : C){ //durchsuche jede Hammingkugel
        if (LS(F, α_i, λ · n) == 1) return 1 ;
    }

    return 0;
}
```

4.3 Verbesserung der Backtracking-Strategie $\left(3^{\lambda n} \rightarrow (3-\epsilon)^{\lambda n}\right)$

Zur Berechnung der Laufzeit in 4.35 haben wir einen Verzweigungsgrad β [1] von 3 für das Backtracking von `LS` gewählt, da dies bei 3-SAT der worst-case ist. Im folgenden werden wir zeigen, dass der maximale Verzweigungsgrad verringert werden kann, um so die Laufzeit von `LS` und damit auch `Search` insgesamt zu verbessern.

Gegeben sei zunächst eine Belegung α_i mit $\alpha_i \not\models F$. Die unter α_i in F enthaltenen Klauseln K werden nun in vier Mengen aufgeteilt: K ist vom TYP_j ($j \in \{0, \ldots, 3\}$), wenn genau j Literale in K unter α_i wahr sind. Es gilt nun mit Sicherheit $|TYP_0| > 0$ (d.h. es gibt unerfüllte Klauseln), da sonst bereits $\alpha_i \models F$.

Wir betrachten im Folgenden fünf verschiedene Fälle von Klauselkonfigurationen, und geben für jeden dieser Fälle eine Backtracking-Strategie an, die den theoretischen worst-case vom Verzweigungsgrad 3 unterbietet. Diese fünf Fälle sind bereits vollständig was die Menge aller Klauselkonfigurationen betrifft, und mithin kann der Verzweigungsgrad von LS damit auf den nun übrig gebliebenen schlechtesten Fall verringert werden, der jedoch besser ist als der bisher verwendete.

- **Fall 0:** $\exists K \in TYP_0 : |K| \leq 2$. Falls es eine unerfüllte Klausel gibt, die höchstens zwei Literale enthält, so kann der Verzweigungsgrad auf dieser Klausel höchstens zwei sein. Daraus folgt $\beta \leq 2$.

[1] auch Branching-Faktor genannt

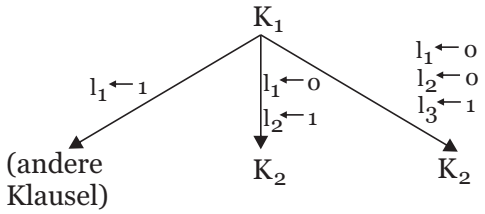

Abbildung 4.3: Optimiertes Backtracking im Fall 1.

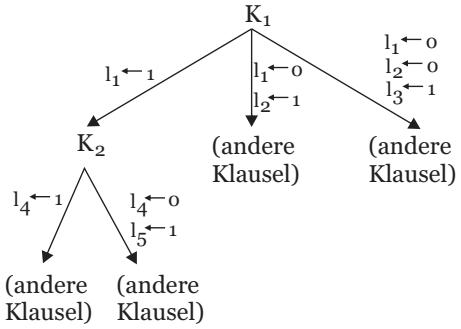

Abbildung 4.4: Optimiertes Backtracking im Fall 2.

- **Fall 1:** $\exists K_1, K_2 \in TYP_0 : K_1 \cap K_2 \neq \emptyset, K_1 \cap K_2 = \{l_1\}$. Wenn es zwei unerfüllte Klauseln gibt, die genau ein Literal gemeinsam haben, so folgt daraus, dass wir beide Klauseln wahr machen können, indem wir das gemeinsame Literal wahr machen. Für den Fall, dass dies zu keiner Lösung führt, betrachten wir die Restklauseln von K_1, K_2 ähnlich wie bei der Monien-Speckenmeyer-Heuristik (siehe Abbildung 4.3). Für den Verzweigungsgrad gilt hier wegen der charakteristischen Gleichung für diesen Fall:

$$T(w) = T(w-1) + 4 \cdot T(w-2) \tag{4.38}$$

$$\beta^w = \beta^{w-1} + 4 \cdot \beta^{w-2} \tag{4.39}$$

$$\beta^2 - \beta - 4 = 0 \tag{4.40}$$

$$\Rightarrow \beta = \frac{1+\sqrt{17}}{2} \approx 2,56 \tag{4.41}$$

- **Fall 2:** $\exists K_1 \in TYP_0, K_2 \in TYP_1$:
$$K_1 = \{l_1, l_2, l_3\}, K_2 = \{\neg l_1, l_4, l_5\}, \alpha_i \models \neg l_1.$$
Falls es eine unerfüllte Klausel K_1 und eine durch genau ein Literal erfüllte Klausel K_2 gibt, und das K_2 erfüllende Literal auch in K_1 vorkommt, so müssen wir entweder nur zwei Literale in K_1 oder nur zwei Literale in K_2 betrachten, auch hier wieder ähnlich Monien-Speckenmeyer (siehe Abbildung 4.4). Für den Verzweigungsgrad gilt hier wegen der charakteristischen Gleichung:

$$T(w) = 2 \cdot T(w-1) + 2 \cdot T(w-2) \tag{4.42}$$

$$\beta^2 = 2\beta + 2 \tag{4.43}$$

$$\Rightarrow \beta = \frac{2+\sqrt{12}}{2} \approx 2.76 \tag{4.44}$$

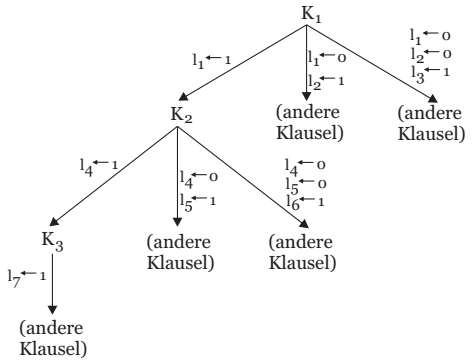

Abbildung 4.5: Optimiertes Backtracking im Fall 3.

- **Fall 3:** $\exists K_1, K_2 \in TYP_0, K_3 \in TYP_2$:
  $K_1 = \{l_1, l_2, l_3\}, K_2 = \{l_4, l_5, l_6\}, K_3 = \{\neg l_1, \neg l_4, l_7\}$ mit $\alpha_i \models \neg l_1, \alpha_i \models \neg l_4$
  Gegeben sind hier zwei unerfüllte und disjunkte Klauseln K_1 und K_2. Ferner haben wir eine Klausel K_3, die durch zwei Literale $(\neg l_1, \neg l_4)$ wahr wird. Diese zwei Literale sind jedoch negiert einmal in K_1 und einmal in K_2 enthalten. Für den Fall, dass wir sowohl die Zuweisung von l_1 als auch die Zuweisung von l_4 invertieren, müssen wir, um Klausel K_3 noch zu erfüllen, zwangsweise die Zuweisung von l_7 invertieren. Für den Fall, dass wir die Zuweisung von l_1 invertieren, die von l_4 aber belassen, müssen wir nur K_2 erfüllen, was lediglich mit l_5 oder l_6 gelingen muss. Für den Fall, dass wir die Zuweisung von l_1 nicht ändern, müssen wir K_1 mit Hilfe von l_2 oder l_3 erfüllen (K_2 wird hierbei nicht beachtet, siehe Abbildung 4.5). Insgesamt ergibt sich für den Verzweigungsgrad wegen der charakteristischen Gleichung:

$$T(w) = 2 \cdot T(w-1) + 2 \cdot T(w-2) + T(w-3) \tag{4.45}$$
$$\beta^3 = 2\beta^2 + 2\beta + 1 \tag{4.46}$$
$$\Rightarrow \beta \approx 2,83 \tag{4.47}$$

- **Fall 4:** Für die Betrachtung dieses Falls führen wir folgende Bezeichner ein:

  Definition 31 *Variablen, die in TYP_0 Klauseln enthalten sind, nennen wir interne Variablen. Alle sonstigen Variablen nennen wir externe Variablen.*

  Wir streichen nun zunächst einige Klauseln aus der Betrachtung für Fall 4, da sie irrelevant sind:

  - Klauseln, die aufgrund einer externen Variablen erfüllt werden, da die Belegung der externen Variablen nicht geändert wird.
  - Klauseln, die zwei Literale l_1, l_2 enthalten, so dass $\neg l_1, \neg l_2$ in einer TYP_0-Klausel vorkommt, da in jeder TYP_0-Klausel nur ein Literal geflippt werden soll, bleiben diese Klauseln immer erfüllt.

  Die jetzt noch übrigen Klauseln (bezeichnet als relevante Klauseln), werden betrachtet.

  $\exists K_0 \in TYP_0, K_7, K_8, K_9 \in TYP_3$:
  $K_0 = \{l_1, l_2, l_3\}, K_7 = \{\neg l_1, \neg l_1', \neg l_1''\}, K_8 = \{\neg l_2, \neg l_2', \neg l_2''\}, K_9\{\neg l_3, \neg l_3', \neg l_3''\}$
  Abbildung 4.6 zeigt die hier möglichen Modifikationen von α_i. Es ist wichtig zu

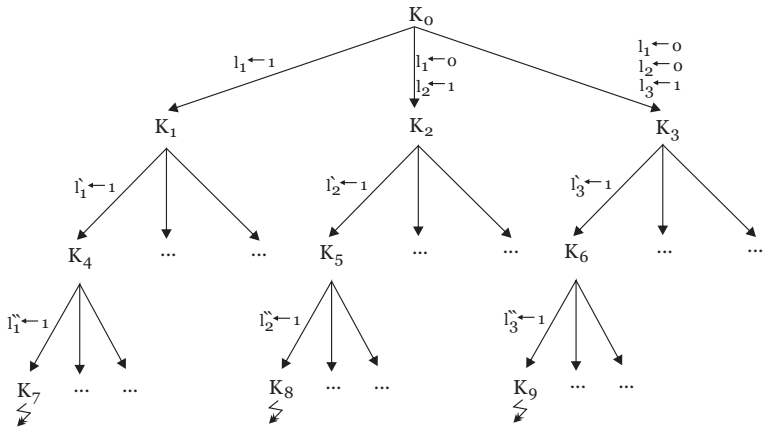

Abbildung 4.6: Optimiertes Backtracking im Fall 4.

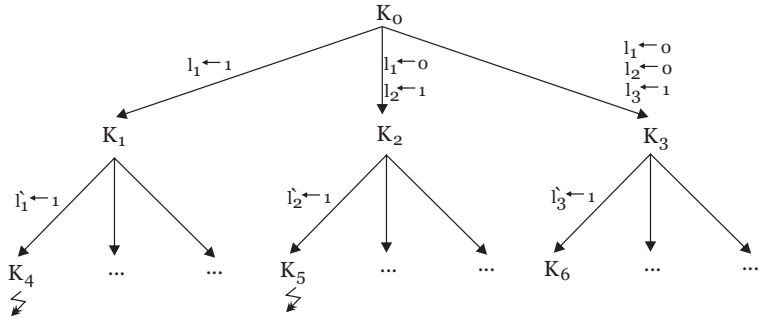

Abbildung 4.7: Optimiertes Backtracking, falls die Fälle 1 bis 4 nicht auftreten.

verstehen, dass dieser Fall in polynomieller Zeit festgestellt werden kann. Für den Verzweigungsgrad ergibt sich nun aus der charakteristischen Gleichung:

$$T(w) = 6 \cdot T(w-2) + 6 \cdot T(w-3) \tag{4.48}$$

$$\beta^3 = 6\beta + 6 \tag{4.49}$$

$$\Rightarrow \beta = 2,82 \tag{4.50}$$

- **Restfälle :** Falls keiner der Fälle 0 bis 4 auftritt, ergibt sich für jede Wahl von $K_0 \in TYP_0$ als Wurzel das Bild in Abbildung 4.7. Hier kann in K_0 die Zuweisung von l_3 invertiert werden. Es kann keine relevante Klausel geben, die $\neg l_3$ enthält, da sonst bereits Fall 4 vorliegen würde.

Wir betrachten nun erneut die Gleichung aus 4.35, verwenden aber anstelle von $k = 3$ (dem vormaligen worst-case für den Verzweigungsfaktor), $k = 2,83$ (dem nun vorliegenden worst-case aus Fall 3). Des weiteren verwenden wir nun anstelle von λ zur Festsetzung der Suchtiefe $w = \lambda \cdot n$ eine neue Variable γ, da nun die Funktion f ein anderes Minimum haben wird (nicht mehr $\lambda = 0.25$, sondern $\gamma = 0.26$). Es

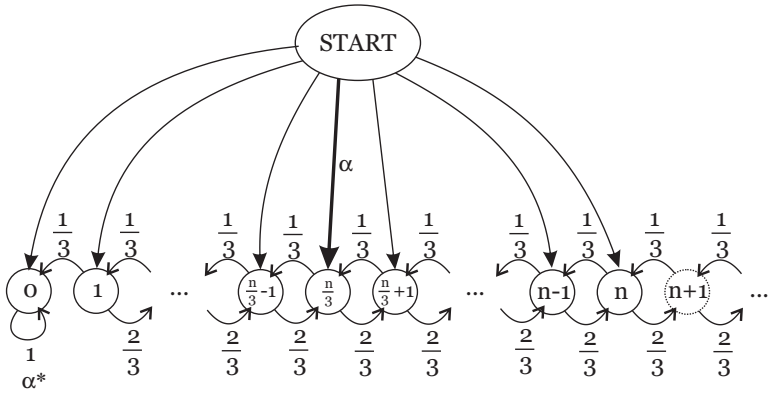

Abbildung 4.8: Der Vorgang des Lösens einer erfüllbaren Formel F durch den `WalkSAT-Schöning`-Algorithmus als Markov-Kette. Die Belegung α^* erfüllt F. α wird zu Beginn eines "try" zufällig gezogen. Wir betrachten den Fall, dass der Hamming-Abstand von α zur Lösung α^* bei $\frac{n}{3}$ liegt.

wird:

$$O^*(t \cdot k^{\gamma \cdot n}) = O^*(2^{n(1-h(\gamma))} \cdot 2.83^{\gamma \cdot n}) \tag{4.51}$$

$$= O^*([\underbrace{2^{1-h(\gamma)} \cdot 2.83^{\gamma}}_{f(\gamma)}]^n) \tag{4.52}$$

$$= O^*(1.481^n) \tag{4.53}$$

Hierbei hat f nun bei $\gamma = 0.26$ das Minimum. Die Laufzeit wurde durch die eingeführten Fallunterscheidungen von $O^*(1.5^n)$ auf $O^*(1.481^n)$ verbessert.

4.4 Ein random-walk Algorithmus für k-SAT

Betrachtet wird im folgenden ein WalkSAT-Algorithmus. Die folgenden Ergebnisse lassen sich für allgemeines k angeben, wir beschränken uns jedoch bei der Analyse auf $k = 3$.

Listing 4.3: WalkSAT-Schöning

```
WalkSAT−Schöning(F:k−KNF Formel, erfüllbar){
    for(i = 1; i < MAX−TRIES; i++){
        α ∈R {0,1}^n; //Zufällige Startbelegung
        for (j = 1; j < MAX−FLIPS; j++){
            if (Fα = 1) return "satisfiable";
            //Sei K = {l1,l2,l3} ∈ F, Kα = 0
            z ∈R {1,2,3};
            α = α|lz←1−lz; //Flip für lz.
        }
        //keine erfüllende Belegung gefunden
        return "unknown";
    }
```

Wir sind nun daran interessiert, mit welcher Wahrscheinlichkeit der Algorithmus "satisfiable" ausgibt. Dazu verwenden wir das bereits bekannte Modell der Markov-Kette (siehe 4.8). Wir gehen davon aus, dass die F erfüllende Belegung in α^* gegeben

ist. Die Zustandsnummern der Markov-Kette entsprechen dem Hamming-Abstand des gerade betrachteten α zu α^*, also $d(\alpha, \alpha^*)$. Die Übergangswahrscheinlichkeiten werden mit $\frac{1}{3}$ für einen Schritt in Richtung α^* und $\frac{2}{3}$ für einen Schritt von α^* weg angegeben. Tatsächlich ist $Pr(j \to j-1) \geq \frac{1}{3}$, aber $p^j(1-p)^{n-j} \geq \frac{1}{3}^j \left(\frac{2}{3}\right)^{n-j}$. Wir nehmen hier also den worst-case an, vereinfachen uns aber die Analyse damit erheblich. Zunächst betrachten wir die Wahrscheinlichkeit für einen Erfolg in einem "try", und errechnen damit als Kehrwert die Gesamtlaufzeit des Algorithmus.

Bezogen auf einen "try" analysieren wir die Wahrscheinlichkeit für einen Erfolg für genau eine Starkonstellation: $j = d(\alpha, \alpha^*) = \frac{n}{3}$ (siehe 4.8).

$$
\begin{aligned}
Pr(\text{Erfolg für einen TRY}) \geq Pr(j = \frac{n}{3}) \cdot & && (4.54)\\
& Pr(\text{Random Walk erreicht Zustand 0 nach} && (4.55)\\
& \text{genau } n \text{ Schritten; statend bei Zustand } n/3) && (4.56)\\
& \geq \binom{n}{n/3} \cdot \left(\frac{1}{2}\right)^n \cdot \binom{n}{n/3} \cdot \left(\frac{2}{3}\right)^{\frac{n}{3}} \cdot \left(\frac{1}{3}\right)^{\frac{2n}{3}} && (4.57)\\
& \gtrsim \left(3^{\frac{2}{3}} \cdot \left(\frac{3}{2}\right)^{\frac{4}{3}} \cdot \left(\frac{1}{2}\right) \cdot \left(\frac{2}{3}\right)^{\frac{1}{3}} \cdot \left(\frac{1}{3}\right)^{\frac{2}{3}}\right)^n && (4.58)\\
& = \left(\frac{3}{4}\right)^n && (4.59)
\end{aligned}
$$

Dabei zeigt in 4.58 das Symbol "$\gtrsim$" an, dass polynomiale Faktoren nicht beachtet werden. Der Umformung liegt zu Grunde, dass folgendes gilt:

$$
\binom{n}{n \cdot n} \approx 2^{h(w) \cdot n} = 2^{(w \log \frac{1}{w} + (1-w) \cdot \log \frac{1}{1-w}) \cdot n} = \left(\left(\frac{1}{w}\right)^w \cdot \left(\frac{1}{1-w}\right)^{1-w}\right)^n \tag{4.60}
$$

Vergleiche hierzu auch 4.15 auf Seite 49.

Daraus ergibt sich eine Laufzeit von $O(\frac{4}{3}^n) = O(1,3333\ldots^n))$. Im Folgenden werden noch Verbesserungen für den random-walk betrachtet, welche die Laufzeit noch weiter verringern.

4.4.1 Verbesserung der Laufzeit

Ursprünglich betrachten wir

$$
Pr(Erfolg) \gtrsim \sum_{j=0}^{n} \binom{n}{j} 2^{-n} \cdot \left(\frac{1}{2}\right)^j \overset{\text{Binomial}}{=} \left(\frac{3}{4}\right)^n \tag{4.61}
$$

Definiere:

$$
X_i = \begin{cases} 1, \alpha_i \neq \alpha_i^* \\ 0, \alpha_i = \alpha_i^* \end{cases} \tag{4.62}
$$

$$
\left(\frac{1}{2}\right)^{X_i} = \begin{cases} \frac{1}{2}, X_i = 1 \\ 1, X_i = 0 \end{cases} \quad \Rightarrow E\left(\left(\frac{1}{2}\right)^{X_i}\right) = \frac{3}{4} \tag{4.63}
$$

Legt man das Augenmerk auf die zufällig gezogene Belegung α, so kann man die Gleichung aus 4.61 umschreiben:

$$Pr(Erfolg) \gtrsim \sum_{\alpha \in \{0,1\}^n} Pr(\alpha \text{ wurde gewählt}) \cdot \left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)} \tag{4.64}$$

$$= E\left(\left(\frac{1}{2}\right)^{d(\alpha,\alpha^*)}\right) \tag{4.65}$$

$$\overset{4.62}{=} E\left(\left(\frac{1}{2}\right)^{X_1+X_2+\ldots+X_n}\right) \tag{4.66}$$

$$= E\left(\prod_{i=1}^{n}\left(\frac{1}{2}\right)^{X_i}\right) \tag{4.67}$$

$$\overset{X_i \text{ unabh.}}{=} \prod_{i=0}^{n} E\left(\left(\frac{1}{2}\right)^{X_i}\right) \tag{4.68}$$

$$\overset{4.63}{=} \left(\frac{3}{4}\right)^n \tag{4.69}$$

Das Ergebnis aus 4.69 war bereits bekannt, verwendet werden kann jetzt jedoch 4.66, um zu analysieren, wie sich das gleichzeitige Flippen mehrerer Variablen in α auf die Erfolgswahrscheinlichkeit in einem "try" auswirkt. Sei hierzu $K \in F, K = \{l_1, l_2, l_3\}$, mit $K\alpha = 0$ (d.h. $(l_1, l_2, l_3) = (0, 0, 0)$). Je nachdem, welche der drei zu Grunde liegenden Variablen hier einen neuen Wert zugewiesen bekommen, gehört die (neue) Kombination (l_1, l_2, l_3)‘ in eine der Folgenden Fälle:

1. (l_1, l_2, l_3)‘ $\in \{(0, 0, 1), (0, 1, 0), (1, 0, 0)\}$ (genau ein Flip), mit Wahrscheinlichkeit p_1
2. (l_1, l_2, l_3)‘ $\in \{(0, 1, 1), (1, 0, 1), (1, 1, 0)\}$ (genau zwei Flips), mit Wahrscheinlichkeit p_2
3. (l_1, l_2, l_3)‘ $\in \{(1, 1, 1)\}$ (alle drei Variablen geflippt), mit Wahrscheinlichkeit p_3

Insgesamt muss gelten: $3p_1 + 3p_2 + p_3 = 1$. Wir können nun die Gleichung 2.66 umschreiben, und gruppieren die Variablen, die zusammen geflippt werden, neu (seien dies o.B.d.A. die Variablen x_1, x_2, x_3):

$$E\left(\left(\frac{1}{2}\right)^{X_1+\ldots+X_n}\right) = E\left(\left(\frac{1}{2}\right)^{X_1+X_2+X_3} \cdot \left(\frac{1}{2}\right)^{X_4+\ldots+X_n}\right) \tag{4.70}$$

Bezogen auf α^* (über das wir keine Kontrolle haben), ergeben sich für die Variablen aus K die selben drei Fälle von oben.

1. $\alpha^* \in \{(0, 0, 1), (0, 1, 0), (1, 0, 0)\}$
2. $\alpha^* \in \{(0, 1, 1), (1, 0, 1), (1, 1, 0)\}$
3. $\alpha^* \in \{(1, 1, 1)\}$

Man ist nun daran interessiert, die drei zusammen durchgeführten Flips in α so zu gestalten, dass das dadurch modifizierte α im selben Fall einsortiert wird, wie α^*. Dies hängt zum einen von α^* ab, das jedoch unbekannt ist und sich der Kontrolle

des Algorithmus entzieht. Zum Anderen hängt dies aber von den Wahrscheinlichkeiten p_1, p_2, p_3 ab, die bestimmen, für welchen Fall wir α modifizieren. Unter der Annahme, dass α^* fest zu einem Fall gehört, und α modifiziert werden muss, führen wir nun eine Fallunterscheidung durch. Für diese drei Fälle lassen sich die p_i bestimmen. Wir zählen hierzu einfach ab, um wie viele Bit sich α und α^* nach dem Flippen der drei Variablen unterscheiden können, wenn α^* sich in einem fest vorgegebenen Fall befindet.

1. α^* wie in Fall 1. $E\left(\left(\frac{1}{2}\right)^{X_1+X_2+X_3}\right) = p_1 \cdot \left(\frac{1}{2}\right)^0 + 2p_2 \cdot \left(\frac{1}{2}\right)^1 + (2p_1+p_3) \cdot \left(\frac{1}{2}\right)^2 + p_2 \cdot \left(\frac{1}{2}\right)^3 = \frac{p_3}{4} + \frac{9p_2}{8} + \frac{3p_1}{2}$
2. α^* wie in Fall 2. $\ldots = \frac{p_3}{2} + \frac{3p_2}{2} + \frac{9p_1}{8}$
3. α^* wie in Fall 3. $\ldots = p_3 + \frac{3p_2}{2} + \frac{3p_1}{4}$

Diese Gleichungen ergeben eine lineare Abbildung in den Variablen p_1, p_2, p_3, die unter der Nebenbedingung $3p_1+3p_2+p_3 = 1$ minimiert werden kann. Das Minimum für der Abbildung in p_1, p_2, p_3 ergibt dann einen neuen (maximalen) Erwartungswert $E((1/2)^{X_1+X_2+X_3})$ für den erfolgreichen Flip der drei Variablen. Wird der Erwartungswert maximiert, so erhöht sich die Wahrscheinlichkeit, dass das modifizierte α in den geänderten Positionen mit α^* übereinstimmt.

Wir erhalten für die Wahrscheinlichkeiten: $p_1 = \frac{4}{21}, p_2 = \frac{2}{21}, p_3 = \frac{3}{21}$, und es wird

$$\underbrace{E\left(\left(\frac{1}{2}\right)^{X_+X_2+X_3}\right)}_{\text{Drei Flips mit Berücks. der } p_i} = \frac{3}{7} < \left(\frac{3}{4}\right)^3 = \underbrace{E\left(\left(\frac{1}{2}\right)^{X_1} \cdot \left(\frac{1}{2}\right)^{X_1} \cdot \left(\frac{1}{2}\right)^{X_1}\right)}_{\text{Einzelne Flips ohne Berücks. der } p_i} \quad (4.71)$$

Wenn man für den i-ten Flip eine Menge $K_{i_{\widehat{m}}}$ von Klauseln findet, die Variablendisjunkt sind, so kann man gleich mehrere "kombinierte" Flips durchführen. Des Weiteren kann man, abhängig davon wie viele solcher unabhängigen Klauseln man findet, eine Fallunterscheidung durchführen.

Falls die Anzahl $\widehat{m}$ der variablendisjunkten Klauseln klein ist ($< 0,1466525 \cdot n$), so bringt die oben beschriebene Verbesserung wenig. Allerdings kann man, über die in den Klauseln enthaltenen Variablen eine Menge von partiellen Belegungen $\beta_1, \ldots, \beta_{7^{\widehat{m}}}$ erstellen, die, angewendet auf die Formel F, nur noch 2-KNF Formeln übrig lassen. Diese Unterformeln $F\beta_l$ können dann effizient auf Erfüllbarkeit getestet werden. Dieser Ansatz funktioniert umso besser, je weniger partielle Belegungen β_l betrachtet werden müssen. Für eine Klausel K_{i_j} mit ihren drei Variablen müssen auch nicht alle 8 0-1-Kombinationen an Belegungen betrachtet werden (dies wären 8 Stück). Eine dieser Belegungen falsifiziert die Klausel, und kann daher ignoriert werden. Pro Klausel bleiben also 7 Belegungen zu prüfen. Insgesamt sind dies bei $\widehat{m}$ Klauseln $7^{\widehat{m}}$ viele Belegungen. In anderen Worten, je kleiner die Anzahl der variablendisjunkten Klauseln ist, desto weniger Variablen stehen für das Bilden der β_l zur Verfügung. Daher ist die Anzahl der zu überprüfenden 2-KNF Formeln klein und der Aufwand sinkt.

Falls die Anzahl der variablendisjunkten Klauseln groß ist werden $3 \cdot \widehat{m}$-viele der Variablen unter den oben beschrieben wahrscheinlichkeiten Belegt. Der Erwartungswert für einen "Teilerfolg" für diese Variablen liegt hier bei $\frac{3}{7}$. Die restlichen Variablen werden nach dem Zufallsprinzip einzeln belegt und die Erwartungswerte liegen hier bei $\frac{3}{4}$. Insgesamt liegt der Erwartungswert für den Erfolg für einen solchen "try" bei $\left(\frac{3}{4}\right)^{n-3\widehat{m}} \cdot \left(\frac{3}{7}\right)^{\widehat{m}}$. Unter der Annahme, dass $\widehat{m} < 0,1466525 \cdot n$ als "klein" definiert ist, erhalten wir eine Laufzeit von $O(1,330258^n)$.

Die Arbeitsweise des modifizierten `WalkSAT-Schöning`-Algorithmus ist nun wie folgt:

Listing 4.4: WalkSAT-Schöning-Mod

```
WalkSAT−Schöning−Mod(F:k−KNF Formel, erfüllbar){
    for(i = 1; i < MAX−TRIES; i++){
        α ∈_R {0,1}^n; //Zufällige Startbelegung
        for (j = 1; j < MAX−FLIPS; j++){
            if (Fα = 1) return "satisfiable";
            //Finde maximale Menge von unerfüllten Klauseln U = {K_{i_1}, ..., K_{i_m̂}},
            //mit |K_{i_j}| = 3 und Var(K_{i_j}) ∩ Var(K_{i_j'}) = ∅, j ≠ j'.
            if (m̂ = |U| < 0,1466525){
                for (∀β ∈ {β_1, ..., β_{7^m̂}:partielle Belegung, die alle Klauseln K_{i_j} erfüllt,
                      aber nur die Variablen aus den K_{i_j} belegen }){
                    if (Fβ = 1){ //Test effizient, da Fβ ∈ 2−KNF
                        return " satisfiable ";
                    }
                }
            } else {
                Flippe in α die Variablen in den K_{i_j} ∈ U gemäß
                den Wahrscheinlichkeiten p_1, p_2, p_3;
                //Die unerfüllten Klauseln, die nicht in U liegen,
                //werden behandelt wie im originalen Algorithmus
                //Sei K = {l_1, l_2, l_3} ∉ U, Kα = 0
              z ∈_R {1,2,3};
              α = α|_{l_z ← 1−l_z}; //Flip für l_z.
        }
        //keine erfüllende Belegung gefunden
        return "unknown";
    }
```

Kapitel 5

Polynomial lösbare Fragmente von SAT

5.1 2-KNF-SAT $\in P$ und 2-KNF-QBF $\in P$ (graphentheoretischer Ansatz)

Sei $F \in 2-KNF$. Definiere Graph G_F. Knotenmenge $K = \{x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n\}$.

Falls $\{x_i, x_j\} \in F$, so fasse dies als Implikation $\neg x_i \to x_j$ (sowie $\neg x_j \to x_i$) auf, und trage diese Implikationen als Kanten $(\neg x_i, x_j), (\neg x_j, x_i)$ in die Kantenmenge des Graphen G_F ein.

Falls $\{x_i\} \in F$, so trage genau eine Kante $(\neg x_i, x_i)$ in die Kantenmenge des Graphen G_F ein. Ein Beispiel findet sich in Abbildung 5.1.

Satz 13 *(Aspvall et. al., 1980).*
F unerfüllbar $\Leftrightarrow$ G_F enthält einen Zyklus der Form

$$x_i \to \ldots \to \neg x_i \to \ldots \to x_i.$$

Bemerkung 18 *Dieses Kriterium lässt sich sogar in linearer Zeit prüfen und es zeigt auch, dass 2-KNF-QBF*[1] *in P liegt.*

[1] QBF steht für Quantified Boolean Formulas und bezeichnet eine Verallgemeinerung von SAT bei der neben (den sonst implizit gegebenen) Existenzquantoren auch Allquantoren erlaubt sind.

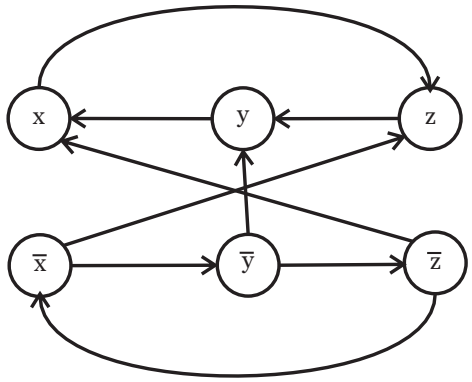

Abbildung 5.1: Der Graph zur Formel $F = (x \vee \neg y) \wedge (x \vee z) \wedge (y \vee \neg z) \wedge (\neg x \vee z) \wedge (y)$.

Beweis 21 *"⇐": Klar.*
"⇒": G_F habe keinen solchen Zyklus. Konstruiere eine erfüllende Belegung für die Formel F wie folgt. Sei x eine beliebige Variable der Formel.

Falls der Pfad

$$\neg x \rightarrow \ldots \rightarrow x$$

im konstruierten Graphen enthalten ist, so setze $x = 1$, sonst $x = 0$. Startend von x (bzw. $\neg x$) setzt man alle erreichbaren Knoten (bzw. deren Literale) entsprechend.

Der Fall, dass x sowohl y als auch $\neg y$ impliziert, kann nicht auftreten, da damit auch umgekehrt die Implikationen von y nach $\neg x$ und $\neg y$ nach $\neg x$ enthalten sein müssten, was einen Zyklus hervorruft (es ist ja $\neg x$ impliziert x bereits gegeben). In anderen Worten: die gemachten Zuweisungen rufen keine Inkonsistenzen hervor.

Falls nach diesen Zuweisungen immer noch unbelegte Variablen im Graphen enthalten sind, so fahre mit diesen fort.

5.2 2-KNF-SAT $\in RP$ (probabilistischer Ansatz)

Gegeben sei folgender Algorithmus von Papadimitriou:

Listing 5.1: Randomisierte Suche

```
RandomSearch(F:2−KNF Formel){
  Wähle α ∈_R {0,1}^n;
  while (true){
    if (Fα = 1){
        return 1;
    } else {
        //Sei C ∈ F so, dass Cα = 0, C = {x, y}.
        Setze:α = { α|_{x←¬x}, mit Wahrscheinlichkeit 1/2
                  { α|_{y←¬y}, mit Wahrscheinlichkeit 1/2
    }
  }
}
```

Wir analysieren nun, wie viele erwartete Schritte (Änderungen von α) der Algorithmus durchführen muss, bis er ein Modell der Formel gefunden hat. Zu dieser Analyse verwenden wir eine Markov-Kette wie in Abbildung 5.2 gezeigt.

Sei $t(j)$ die erwartete Anzahl an Schritten, bis Zustand 0 erreicht ist. Es gilt:

$$j = 0 : t(j) = 0 \tag{5.1}$$

$$0 < j < n : t(j) \leq 1 + \frac{1}{2}t(j-1) + \frac{1}{2}t(j+1) \tag{5.2}$$

$$j = n : t(j) = 1 + t(n-1) \tag{5.3}$$

Falls hierbei Gleichheit gilt ($0 < j < n$), so gilt:

$$t(j) = 2jn - j^2.$$

Gezeigt wird dies mit Induktion.

Der Worst-case ist mit $j = n$ gegeben mithin ist $t(n) = n^2$. Es gilt die Markov-Ungleichung:

$$\text{Sei } X \text{ eine Zufallsvariable, } X \geq 0.\ Pr(X \geq a \cdot E(X)) \leq \tfrac{1}{a}.$$

Für $a = 2$ erhalten wir

$$Pr(\#\text{Schritte bis 0-Zustand} \geq 2n^2) \leq \tfrac{1}{2}.$$

Der 0-Zustand kann auf mehreren Wegen durch die Markov-Kette erreicht werden, wenn die Anzahl der erlaubten Schritte nur groß genug gewählt ist (siehe Abbildung 5.3). In anderen Worten: Wird die Anzahl der erlaubten Schritte größer, so steigt auch die Anzahl der Pfade, die zum 0-Zustand führen. Mithin steigt auch die Wahrscheinlichkeit den 0-Zustand zu erreichen und ein Modell für F zu finden.

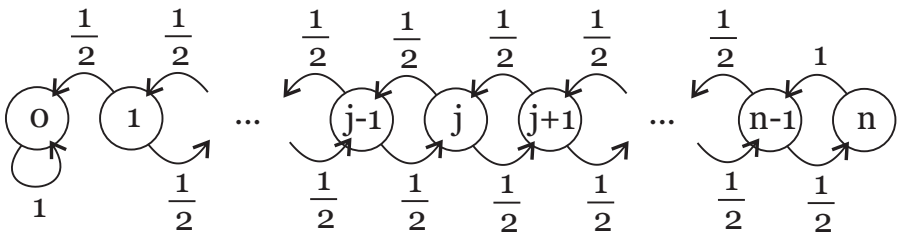

Abbildung 5.2: Ein Knoten j der Markovkette bezeichnet hierbei einen Zustand, in dem der Algorithmus in α eine Belegung hat, die zu einem Modell den Hammingabstand j hat. Die Übergangswahrscheinlichkeiten sind, bis auf die Randknoten, mit 0.5 gegeben. Der linke Randknoten ist ein Zustand, in dem in α ein Modell vorliegt. Aus diesem Knoten entfernt sich der Algorithmus nicht mehr. Der rechte Randknoten ist ein Zustand, bei dem der Algorithmus in α eine Belegung konstruiert hat, die im Hammingabstand zu einem Modell der Formel maximal weit weg liegt ($j = n$ ist also der worst-case). Aus dem rechten Randknoten muss sich der Algorithmus zwangsweise wieder entfernen.

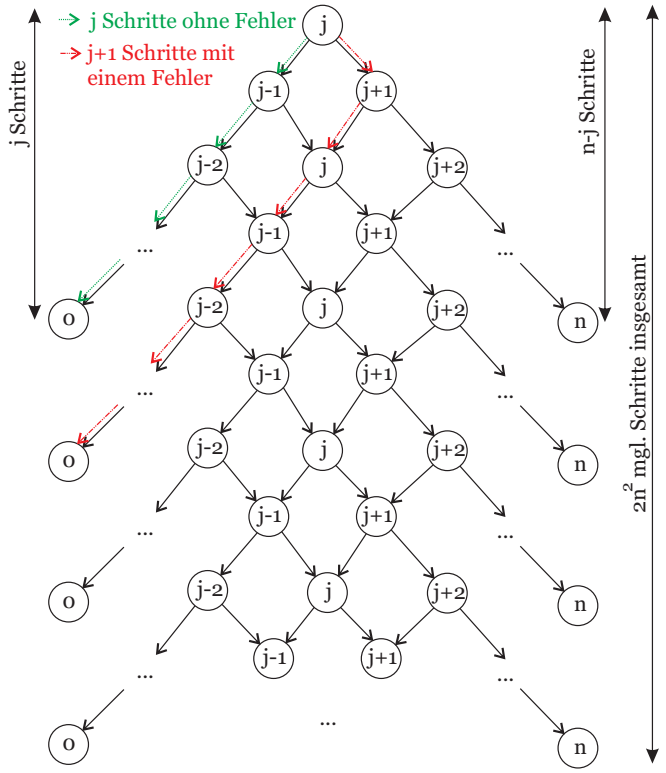

Abbildung 5.3: Um von Knoten j aus zum Knoten 0 zu gelangen sind mehrere Möglichkeiten vorhanden, je nachdem, wie viele Schritte ein Pfad lang werden darf. Es existiert hier genau ein Pfad der länge j bis zum 0-Zustand, $j+1$ Pfade der Länge $j+1$ (mit einem "Fehlschritt"), etc. Darf der Pfad nur lang genug werden, so steigt die Wahrscheinlichkeit, dass man mit einer bestimmten Anzahl von "Fehlschritten" dennoch den 0-Zustand erreicht.

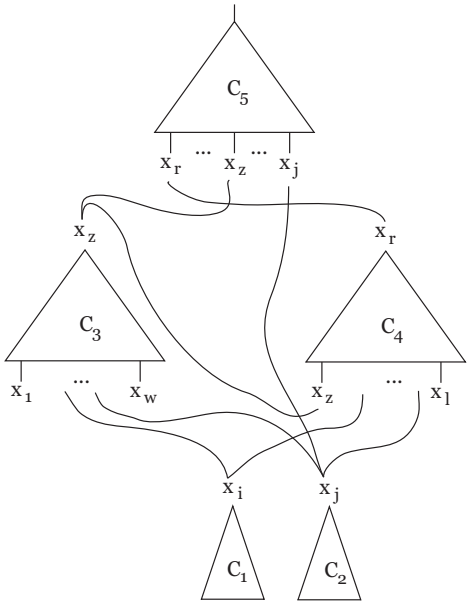

Abbildung 5.4: Eine Horn-Formel als Schaltkreis aufgefasst (die C_i werden dabei als Und-Verknüpfungen der Eingänge aufgefasst). Eine Schaltung ohne Eingänge repräsentiert hierbei eine unit-Klausel (bspw. C_1, C_2). Eine Schaltung ohne Ausgabevariable ist eine Zielklausel (bspw. C_5). Falls eine solche Schaltung mit 1 durchschaltet ist die Horn-Formel unerfüllbar.

Bemerkung 19 *Dieser probabilistische Ansatz lässt sich zu einem k-SAT Algorithmus mit gemäßigt exponentieller Laufzeit ausweiten.*

5.3 Horn-Formeln und Renamable Horn

5.3.1 Horn-Formeln

Zur Erinnerung noch einmal die Definition für Horn-Formeln.

Definition 32 *Eine Formel heißt Horn-Formel, wenn sie in KNF ist und für alle Klauseln der Formel gilt, dass sie höchstens ein positives Literal enthalten. KNF-Klauseln mit keinem positiven Literal heißen Zielklauseln (der Horn-Formel). Klauseln mit genau einem positiven Literal heißen definite Hornklauseln (der Horn-Formel).*

Beispiel 3 *Die Klausel* $(\neg x_1 \vee \ldots \vee \neg x_n \vee y)$ *ist eine definite Hornklausel. Man kann diese auch als Implikation auffassen:* $(x_1 \wedge \ldots \wedge x_n) \rightarrow y$. *Die Zielklausel*

$$(\neg x_1 \vee \ldots \vee \neg x_n) = (\neg x_1 \vee \ldots \vee \neg x_n \vee 0)$$

kann ebenso als Implikation aufgefasst werden: $(x_1 \wedge \ldots \wedge x_n) \rightarrow 0$.

Eine unit-Klausel ist entweder eine Zielklausel oder eine definite Hornklausel. Beispiel für eine definite Hornklausel mit nur einem Literal: $(x) = (x \vee 0) \equiv x \rightarrow 0$.

Eine Horn-Formel kann auch als Schaltkreis aufgefasst werden (siehe Abbildung 5.4). Ausgehend von den unit-Klauseln einer Horn-Formel werden nun Belegungen für die Variablen durchgeführt. Dabei kann es sein, dass alle Voraussetzungen für eine definite Hornklausel belegt werden, welche dann einen weiteren Wert impliziert (siehe Abbildung 5.4). Wird dabei auch nur eine Zielklausel nicht erfüllt, muss

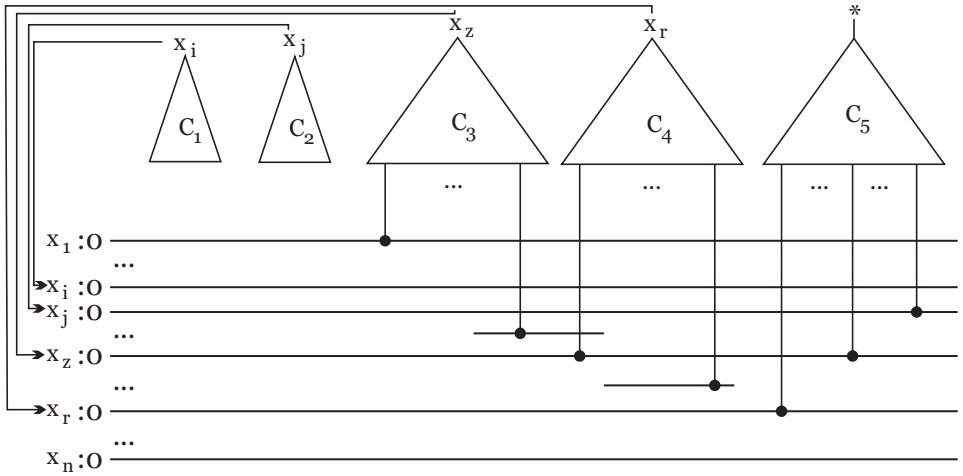

Abbildung 5.5: Eine Horn-Formel kann auch als Rückgekoppeltes Schaltnetz aufgefasst werden (vgl. hierzu auch Abbildung 5.4).

die Horn-Formel unerfüllbar sein. Man kann eine Horn-Formel auch als Schaltkreis mit Rückkopplung auffassen (siehe Abbildung 5.5). Ausgehend von allen Variablen belegt mit 0 werden rückgekoppelt die Belegungen über die einzelnen Schaltungen verändert. Dabei beginnt man mit den Schaltungen der unit-Klauseln und arbeitet sich dann über die Schaltungen der definiten Hornklauseln bis zu den Zielklauseln vor. Auch hier gilt: falls eine der Zielklauseln mit 1 durchschaltet, ist die Horn-Formel unerfüllbar.

5.3.2 Renamable Horn

Definition 33 *F heißt renamable (oder hidden) Horn-Formel, falls*

$$\exists I \subseteq \{1, n\}, \text{ so dass } F|_{x_i \leftrightarrows \neg x_i}, i \in I$$

eine Horn-Formel ist (F ist dann erfüllbarkeitsäquivalent zu $F|_{x_i \leftrightarrows \neg x_i}$).

Die Schreibweise $x_i \leftrightarrows \neg x_i$ bedeutet hierbei, dass x_i mit $\neg x_i$ ersetzt wird (und umgekehrt).

Um zu prüfen, ob eine gegebene Formel F, vorliegend als k-KNF, eine renamable Horn-Formel ist, konstruiert man zunächst eine Hilfsformel F^* aus F wie folgt:

$$F^* = \bigwedge_{C \in F} \bigwedge_{u,v \in C, u \neq v} (u \vee v).$$

Es gilt dann folgender Satz:

Satz 14 *F ist renamable Horn (mittels I) $\Leftrightarrow$ $F^* \in 2-KNF$ ist erfüllbar, wobei die erfüllende Belegung α für F^* nun die Indexmenge I bestimmt: $i \in I \Leftrightarrow \alpha(x_i) = 1$.*

Beweis 22 *"$\Rightarrow$": Sei α eine Belegung mit $F^*\alpha = 1$.*
Setze $I = \{x \in Var(F) | \alpha(x) = 1\}$. Wende die Indexmenge I an, um aus F die Formel F_{Horn} herzustellen (mittels Renaming).
Dann ist zu zeigen: F_{Horn} ist Horn-Formel.
Annahme: F_{Horn} ist keine Horn-Formel. Dann enthält F_{Horn} eine Klausel mit mehr als einem positiven Literal.
In jedem Fall ergibt sich, dass α diese Klausel in F^* nicht erfüllt. Widerspruch.
"$\Leftarrow$": Sei I eine Indexmenge von F zu F_{Horn}, so dass F_{Horn} eine Horn-Formel ist.

Setze: $\alpha = \begin{cases} 1, x \in I \\ 0, x \notin I \end{cases}$.
Nun verwenden wir das selbe Argument wie zuvor. Dies zeigt, dass F^* *durch* α *erfüllt wird.*

□

Beispiel 4 *Sei folgende Formel gegeben:*

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3).$$

Die daraus resultierende Formel für den Test, ob F *hidden Horn ist, ist dann wie folgt:*

$$F^* = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$$
$$\wedge(\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_3).$$

Die Belegung $\alpha = \{x_1 \leftarrow 1, x_2 \leftarrow 1, x_3 \leftarrow 0\}$ *ist ein Modell für* F^*. *Damit ist die Indexmenge für das Ersetzen der Variablen in* F *gegeben mit* $I = \{1, 2\}$.

Damit ist:

$$F_{Horn} = (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3).$$

die aus F *abgeleitete Horn-Formel (also ist* F *renamable Horn).*

Bemerkung 20 *Es ist wichtig zu verstehen, dass* F^* *erfüllbar nur bedeutet, dass es eine Variablenersetzung für* F *(mittels* I*) gibt, mit der* F *in eine Horn-Formel umgewandelt werden kann. Die Erfüllbarkeit von* F^* *sagt nichts darüber aus, ob auch* F *bzw.* F_{Horn} *erfüllbar sind. Jedoch ist* F *zu* F_{Horn} *erfüllbarkeitsquivalent. Ein Modell* α *für* F_{Horn} *wird zu einem Modell von* F*, wenn alle Variablenzuweisungen in* α*, deren Index in* I *zu finden ist, invertiert werden.*

Bemerkung 21 *Es gelten folgende Aussagen:*

- *SAT ist NP-vollständig.*
- *QBF ist PSPACE-vollständig.*
- *HORN-SAT ist P-vollständig.*
- *2-KNF-SAT ist NL-vollständig.*

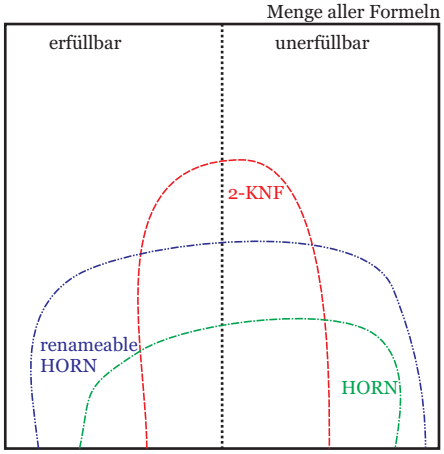

Abbildung 5.6: Die Abbildung gibt einen Überblick über die betrachteten Formelmengen. Eine besondere Menge, genannt MIXED-HORN, sind diejenigen Formeln, deren Klauseln sowohl Horn-Klauseln als auch 2-KNF Klauseln sein können. MIXED-HORN ist nicht die Vereinigung von 2-KNF-SAT und HORN, sondern eine echto Obermenge davon. Bisher ist kein effizienter Algorithmus für MIXED-HORN-SAT bekannt.