2 DLL-Algorithmen

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

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

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

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

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

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

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

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

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

2.1 Reduktionen

Einerklauseln

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

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

```
UP(F, α)
  while in F gibt es eine Einerklausel a
      F := F[a ← 1]
      α := α ∪ [a ← 1]
```

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

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

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

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

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

Reine Literale

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

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

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

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

Subsumption

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

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

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

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

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

Resolution

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

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

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

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

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

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

2.2 Algorithmus von Monien-Speckenmeyer

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

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

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

Es wird also bei Auswahl einer kürzesten Klausel r-fach verzweigt, wobei im i-ten Zweig i Variablen bewertet sind, also höchstens $n-i$ Variablen übrig bleiben.
Da stets $r \leq k$ ist, erhält man folgende Rekursionsgleichung für die Anzahl $T(n)$ der Blätter im Verzweigungsbaum

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

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

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

Autarkien

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

also vereinfacht sich die Rekursion zu

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

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

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

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

2.3 Analysemethode von Kullmann

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Proposition 14. *1. Ist $d'_1 > d_1$, so ist $\tau(d'_1, \dots, d_m) < \tau(d_1, \dots d_m)$.*

2. Ist $d'_1 + d'_2 = d_1 + d_2$, und $\min(d'_1, d'_2) \geq \min(d_1, d_2)$, dann ist

$$\tau(d'_1, d'_2, d_3, \dots, d_m) \leq \tau(d_1, d_2, d_3, \dots, d_m)$$

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

3. Sei $d = (d_1, \dots, d_m)$ und $d' := (d'_1, \dots, d'_n)$, und sei $d^* := (d_1 + d'_1, \dots, d_1 + d'_n, d_2, \dots, d_m)$. Dann gilt:

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

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

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

1. Vergrößern der Distanz zu einem Sohn verbessert die Verzweigungszahl eines Knotens. Beispielsweise ist $\tau(3,1) < 1.46558 < \tau(2,1)$.
2. Von zwei Verzweigungen mit gleicher Summe der Distanzen liefert die symmetrischere die bessere Verzweigungszahl. Beispielsweise ist $\tau(2,2) = \sqrt{2} < \tau(3,1)$.
3. Werden zwei Knoten zu einem verschmolzen, so liegt die Verzweigungszahl des neuen Knotens zwischen denen der alten Knoten. Beispielsweise ist $\tau(4,3,2) = \tau(3,1+3,1+1) = \tau(3,1)$, und
$$\tau(3,1) < \tau(3,3,2) = \tau(3,1+2,1+1) < 1.52138 < \tau(2,1)\ .$$

2.4 Der Algorithmus von Zhang

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

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

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

Beweis. Da a und b in F nicht vorkommen, kann jedes $\alpha \models F$ so abgeändert werden, dass a oder b mit 0 bewertet werden.

Ist $\alpha(c) = 1$, so gilt $\alpha \models (a \vee b \vee c)$, und daher können a und b mit 0 bewertet werden. Ist andererseits $\alpha(c) = 1$, so muss eines von a und b mit 1 bewertet werden, es genügt aber auch eines. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Lösche subsumierte 3-Klauseln.

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

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

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

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

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

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

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

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

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

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

$\gamma_1 := [a \leftarrow 1, c \leftarrow 1]$
$\gamma_2 := [a \leftarrow 0, b \leftarrow 1]$

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

$\gamma_1 := [a \leftarrow 1]$
$\gamma_2 := [a \leftarrow 0, b \leftarrow 1, c \leftarrow 1]$

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

$\gamma_1 := [a \leftarrow 1]$
$\gamma_2 := [a \leftarrow 0, b \leftarrow 1]$

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

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

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

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

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

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

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

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

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

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

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

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

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

An den folgenden Stellen im Algorithmus treten Verzweigungen auf:

- Prozedur unit()

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

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

- Fall 1.1

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

- Fall 1.2

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

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

- Fall 2

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

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

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

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

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

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

  Dieser Fall ist analog zum Fall 1.1.

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

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

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