3 Effiziente Implementierung von DLL

3.1 Datenstrukturen

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

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

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

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

Verzeigerte Listen

Die Struktur zur Darstellung einer Klausel hat die Einträge:

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

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

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

Der Zeiger sat ist der Nullzeiger NULL, solange die Klausel nicht erfüllt ist. Ist die Klausel erfüllt, so zeigt er auf die Variable, durch deren Bewertung die Klausel zuerst erfüllt wurde. Dadurch kann beim backtracking erkannt werden, ob beim Rücksetzen des Wertes einer Variablen die Klausel wieder als unerfüllt zu markieren ist.

Die Struktur zur Darstellung einer Variable hat die Einträge:

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

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

Eine Implementierung dieser Struktur in C ist in Tabelle 1 angegeben.

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

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

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

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

Tabelle 1: Datenstruktur für KNF-Formeln in C

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

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

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

Tabelle 2: Prozedur `set-true` in C

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

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

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

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

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

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

Tabelle 3: Prozedur `unset-true` in C

9-10 zu ersetzen durch das folgende Codesegment:

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

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

3.2 Verzweigungsheuristiken

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

Zur Formulierung dieser Strategien definieren wir zunächst die Werte

$$h_k(a) := \text{Anzahl der Vorkommen von } a \text{ in Klauseln der Größe } k$$
$$h(a) := \sum_{k=2}^{n} h_k(a) \; .$$

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

Einfache Anzahlheuristiken

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

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

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

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

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

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

Die MOM-Heuristik

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

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

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

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

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

Bohm's Heuristik

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

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

wobei α und β frei wählbare Parameter sind.

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

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

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

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

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

Die Jeroslaw-Wang Heuristik

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

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

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

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

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

3.3 Iterative Implementierung

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

Jeder Eintrag auf dem Wertzuweisungsstack enthält folgende Informationen:

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

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

```
while true do
```

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

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

In Pseudocode sieht die Backtracking-Prozedur aus wie folgt:

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

3.4 Konfliktanalyse und Lernen

Der Implikationsgraph

Zur Vereinfachung nehmen wir an, dass Variablen nur durch Verzweigung und unit propagation bewertet werden.

Mit jeder Wertzuweisung wird eine Zahl assoziiert, die Entscheidungstiefe. Diese ist anfänglich 0 und wird bei jeder Wertzuweisung durch Verzweigen inkrementiert, und beim Backtracking wieder dekrementiert. Die nach einer Verzweigung folgenden Wertzuweisungen durch unit propagation erhalten die gleiche Entscheidungstiefe.

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

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

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

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

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

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

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

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

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

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

und die folgenden Knoten im Implikationsgraphen:

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

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

Konfliktanalyse und nichtchronologisches Backtracking

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

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

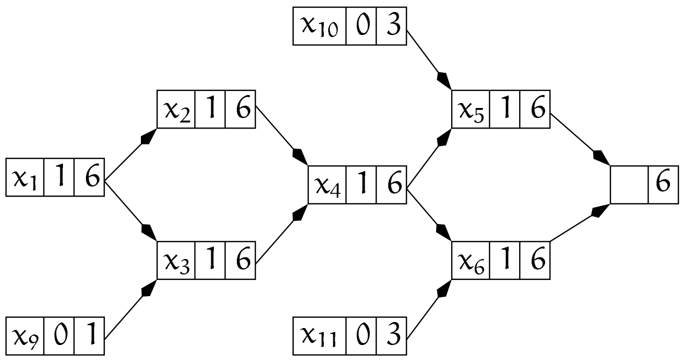

Abbildung 1: Implikationsgraph

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

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

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

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

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

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

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

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

Ist dagegen $b < t(\square)$, so werden zunächst alle Knoten der Tiefe größer als b aus dem Graphen entfernt, und die entsprechenden Wertzuweisungen rückgängig gemacht. Anschließend wird ein neuer Knoten $\square$ der Tiefe $t(\square) = b$ eingefügt und für diesen die Konfliktanalyse wiederholt, also wiederum die Konfliktmenge $G(\square)$ und daraus eine neue Backtrack-Tiefe b berechnet.

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

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

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

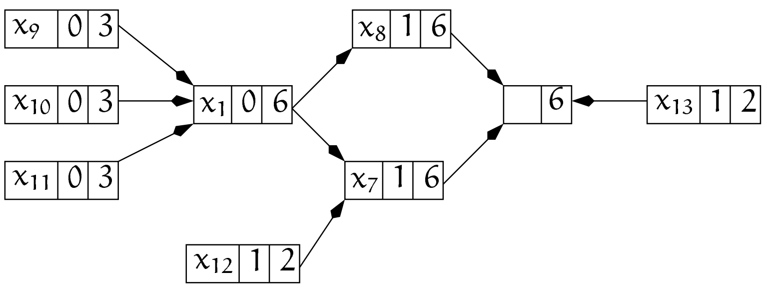

Abbildung 2: Der neue Implikationsgraph

Lernen von Klauseln

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

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

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

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

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

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

Zur Implementierung

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

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

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

Verzweigen anhand gelernter Klauseln

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

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

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

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

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