

Vorlesung “Logik”

Wintersemester 2006/07

Barbara König
Übungsleitung: Tobias Heindel

Resolution (Motivation)

Komplexitätstheoretische Einordnung

Das Erfüllbarkeitsproblem für aussagenlogische Formeln in konjunktiver Normalform ist **NP-vollständig**.

Das bedeutet insbesondere, dass es keine bekannten Algorithmen mit polynomialer Laufzeit gibt, die für eine Formel bestimmen können, ob sie erfüllbar ist oder nicht.

Das Konzept der NP-Vollständigkeit wird in der Vorlesung "Informatik B2" (4. Semester) behandelt.

Resolution (Motivation)

Daher: man benötigt Algorithmen für Erfüllbarkeitstests, die zumindest in vielen Fällen gutartiges Verhalten zeigen.

Dennoch ist im worst case immer eine exponentielle Laufzeit zu erwarten.

Resolution (Motivation)

Wir betrachten nun eine Formel, bei der viele Erfüllbarkeitstester (SAT-Solver) ein schlechtes Verhalten zeigen.

Pigeonhole-Prinzip/Dirichletsches Schubfachprinzip

Wenn $N + 1$ Tauben in N Schlägen sitzen, dann gibt es in mindestens einem Schlag zwei (oder mehr) Tauben.

Wir verwenden zur Formalisierung folgende atomare Formeln: $T_{i,j}$ mit $i \in \{1, \dots, N + 1\}$, $j \in \{1, \dots, N\}$.

- $\mathcal{A}(T_{i,j}) = 1$ bedeutet dabei, dass die i -te Taube im j -ten Schlag sitzt.

Resolution (Motivation)

Wir formalisieren zunächst: “ $N + 1$ Tauben sitzen in N Schlägen”.

- Jede Taube sitzt in mindestens einem Schlag:

$$F_1 = \bigwedge_{i \in \{1, \dots, N+1\}} \bigvee_{j \in \{1, \dots, N\}} T_{i,j}$$

- Keine Taube sitzt in zwei Schlägen:

$$F_2 = \bigwedge_{i \in \{1, \dots, N+1\}} \bigwedge_{j_1, j_2 \in \{1, \dots, N\}, j_1 \neq j_2} (\neg T_{i,j_1} \vee \neg T_{i,j_2})$$

- In einem Schlag sitzen (mindestens) zwei Tauben:

$$G = \bigvee_{i_1, i_2 \in \{1, \dots, N+1\}, i_1 \neq i_2} \bigvee_{j \in \{1, \dots, N\}} (T_{i_1,j} \wedge T_{i_2,j})$$

Resolution (Motivation)

Der gesamte aussagenlogische Ausdruck lautet daher

$$F_1 \wedge F_2 \rightarrow G.$$

Er ist gültig genau dann, wenn $F_1 \wedge F_2 \wedge \neg G$ unerfüllbar ist.

Dabei ist $F_1 \wedge F_2 \wedge \neg G$ schon “beinahe” in konjunktiver Normalform.

Mit Hilfe eines Programms kann man Formeln für jedes N erzeugen und mit Hilfe von `limboole` überprüfen. Für diese Art von Formeln zeigt `limboole` ein relativ schlechtes Laufzeitverhalten, bereits ab $N = 12$ dauert der Test sehr lange.

Resolution (Idee)

$$(F \vee A) \wedge (F' \vee \neg A) \equiv (F \vee A) \wedge (F' \vee \neg A) \wedge (F \vee F')$$

Aus der Herleitung der leeren Disjunktion (= leere Klausel) folgt Unerfüllbarkeit.

Zwei Fragen:

- Kann man aus einer unerfüllbaren Formel immer die leere Klausel herleiten? (**Vollständigkeit**)
- Gibt es eine Möglichkeit, die Herleitung kompakter aufzuschreiben?

Mengendarstellung

- **Klausel:** Menge von Literalen (Disjunktion).
 $\{A, B\}$ stellt $(A \vee B)$ dar.
- **Formel:** Menge von Klauseln (Konjunktion).
 $\{\{A, B\}, \{\neg A, B\}\}$ stellt $((A \vee B) \wedge (\neg A \vee B))$ dar.

Die leere Klausel (= leere Disjunktion) ist äquivalent zu einer unerfüllbaren Formel. Diese wird auch mit \square bezeichnet.

Die leere Formel (= leere Konjunktion) ist äquivalent zu einer gültigen Formel.

Vorteile der Mengendarstellung

Man erhält automatisch:

- **Kommutativität:**

$$(A \vee B) \equiv (B \vee A),$$

beide dargestellt durch $\{A, B\}$

- **Assoziativität:**

$$((A \vee B) \vee C) \equiv (A \vee (B \vee C)),$$

beide dargestellt durch $\{A, B, C\}$

- **Idempotenz:**

$$(A \vee A) \equiv A,$$

beide dargestellt durch $\{A\}$

Resolvent (I)

Definition (Resolvent)

Seien K_1 , K_2 und R Klauseln. Dann heißt R *Resolvent* von K_1 und K_2 , falls es ein Literal L gibt mit $L \in K_1$ und $\bar{L} \in K_2$ und R die Form hat:

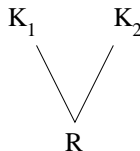
$$R = (K_1 - \{L\}) \cup (K_2 - \{\bar{L}\}).$$

Hierbei ist \bar{L} definiert als

$$\bar{L} = \begin{cases} \neg A_i & \text{falls } L = A_i, \\ A_i & \text{falls } L = \neg A_i \end{cases}$$

Resolvent (II)

Wir stellen diesen Sachverhalt durch folgendes Diagramm dar
(Sprechweise: R wird aus K_1 , K_2 nach L resolviert).



Ferner: falls $K_1 = \{L\}$ und $K_2 = \{\bar{L}\}$, so entsteht die leere Menge als Resolvent. Diese wird mit dem speziellen Symbol \square bezeichnet, das eine unerfüllbare Formel darstellt.

Resolutions-Lemma

Resolutions-Lemma

Sei F eine Formel in **KNF**, dargestellt als Klauselmenge. Ferner sei R ein Resolvent zweier Klauseln K_1 und K_2 in F . Dann sind F und $F \cup \{R\}$ äquivalent.

Beweis: Folgt direkt aus

$$\underbrace{(F_1 \vee A)}_{K_1} \wedge \underbrace{(F_2 \vee \neg A)}_{K_2} \equiv \underbrace{(F_1 \vee A)}_{K_1} \wedge \underbrace{(F_2 \vee \neg A)}_{K_2} \wedge \underbrace{(F_1 \vee F_2)}_R$$

Definition von $Res(F)$

Definition

Sei F eine Klauselmenge. Dann ist $Res(F)$ definiert als

$$Res(F) = F \cup \{R \mid R \text{ ist Resolvent zweier Klauseln in } F\}.$$

Außerdem setzen wir:

$$\begin{aligned} Res^0(F) &= F \\ Res^{n+1}(F) &= Res(Res^n(F)) \quad \text{für } n \geq 0 \end{aligned}$$

und schließlich sei

$$Res^*(F) = \bigcup_{n \geq 0} Res^n(F).$$

Aufgabe

Angenommen, die Formel F enthält n atomare Formeln. Dann gilt welche Abschätzung für $Res^*(F)$?

A $|Res^*(F)| \leq 2^n$ **B** $|Res^*(F)| \leq 4^n$

C $|Res^*(F)|$ kann beliebig groß werden

Dabei bezeichnet $|Res^*(F)|$ die Anzahl der Elemente in $Res^*(F)$.

Aufgabe

Was passiert, wenn alle **Klauseln maximal zweielementig** sind?
Durch die Resolution von zweielementigen Klauseln können nur wieder zweielementige Klauseln entstehen. (Das ist bei drei- und mehrelementigen Klauseln anders.)

Da es bei n verschiedenen atomaren Formeln nur $\binom{2n}{2} = \frac{2n(2n-1)}{2}$ viele zweielementige und $2n$ viele einelementige Klauseln gibt, werden nach spätestens dieser **polynomialen Anzahl** von Schritten keine neuen Klauseln mehr abgeleitet.

Resolutionsatz

Wir zeigen nun die **Vollständigkeit der Resolution**:

Resolutionsatz (der Aussagenlogik)

Eine Klauselmengemenge F ist unerfüllbar genau dann, wenn
 $\square \in Res^*(F)$.

Induktionsprinzip

Um die Aussage

Für jedes $n \in \{0, 1, 2, 3, \dots\}$ gilt $P(n)$.

zu zeigen, gehen wir im allgemeinen folgendermaßen vor:

- Wir zeigen, daß $P(0)$ gilt. (Induktionsanfang)
- Wir zeigen, daß für jedes n gilt:
Wenn $P(n)$ gilt, dann gilt auch $P(n+1)$. (Induktionsschritt)

Dann kann man schließen, daß $P(n)$ für jedes beliebige n gilt.

Beweisidee (I)

Induktion über die Anzahl der atomaren Formeln.

Hier: **Induktionsschritt** mit $n + 1 = 4$

$$F = \{ \{A_1\}, \{ \neg A_2, A_4 \}, \{ \neg A_1, A_2, A_4 \}, \{ A_3, \neg A_4 \}, \{ \neg A_1, \neg A_3, \neg A_4 \} \}$$

Beweisidee (I)

Induktion über die Anzahl der atomaren Formeln.

Hier: **Induktionsschritt** mit $n + 1 = 4$

$$F = \{ \{A_1\}, \{ \neg A_2, A_4 \}, \{ \neg A_1, A_2, A_4 \}, \{ A_3, \neg A_4 \}, \{ \neg A_1, \neg A_3, \neg A_4 \} \}$$

$$F_0 = \{ \{A_1\}, \{ \neg A_2 \}, \{ \neg A_1, A_2 \} \}$$

Beweisidee (I)

Induktion über die Anzahl der atomaren Formeln.

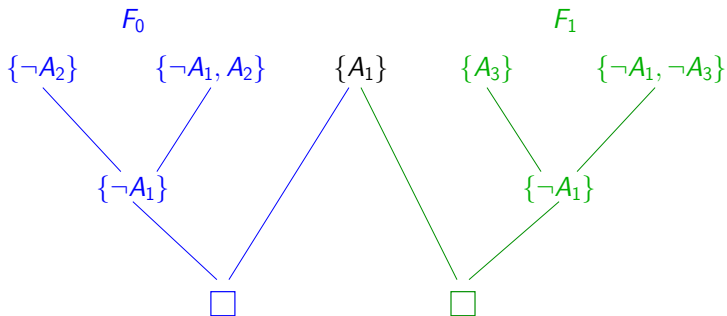
Hier: **Induktionsschritt** mit $n + 1 = 4$

$$F = \{\{A_1\}, \{\neg A_2, A_4\}, \{\neg A_1, A_2, A_4\}, \{A_3, \neg A_4\}, \{\neg A_1, \neg A_3, \neg A_4\}\}$$

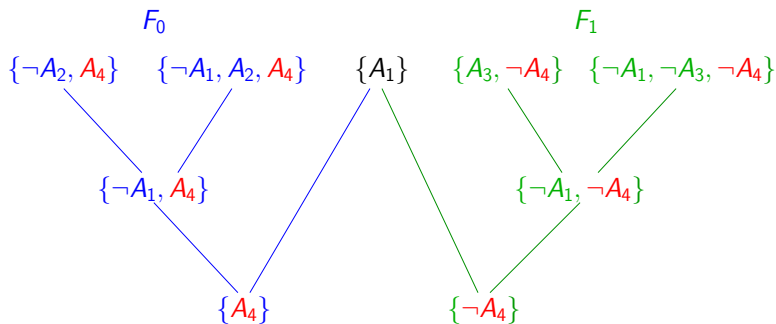
$$F_0 = \{\{A_1\}, \{\neg A_2\}, \{\neg A_1, A_2\}\}$$

$$F_1 = \{\{A_1\}, \{A_3\}, \{\neg A_1, \neg A_3\}\}$$

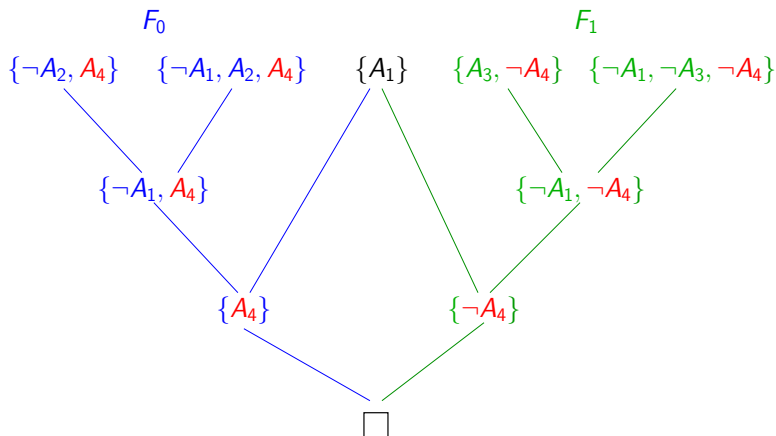
Beweisidee (II)



Beweisidee (II)



Beweisidee (II)



Deduktion

Definition (Deduktion)

Eine *Deduktion* (oder *Herleitung* oder *Beweis*) der leeren Klausel aus einer Klauselmenge F ist eine Folge von K_1, K_2, \dots, K_m von Klauseln mit folgenden Eigenschaften:

K_m ist die leere Klausel und für jedes $i = 1, \dots, m$ gilt, daß K_i entweder Element von F ist oder aus gewissen Klauseln K_a, K_b mit $a, b < i$ resolviert werden kann.

Eine Klauselmenge ist unerfüllbar genau dann, wenn eine Deduktion der leeren Klausel existiert.

Es ist also nicht notwendig, ganz $Res^*(F)$ zu berechnen, sondern es können geeignete Such-Heuristiken verwendet werden.

Resolutionskalkül

Mit dem Begriff *Kalkül* bezeichnet man eine Menge von *syntaktischen* Umformungsregeln, mit denen man *semantische* Eigenschaften herleiten kann.

- *Syntaktische* Umformungsregeln: Resolution, Stopp bei Erreichen der leeren Klausel
- *Semantische* Eigenschaft: Unerfüllbarkeit

Wünschenswerte Eigenschaften eines Kalküls:

- *Korrektheit*: Wenn die leere Klausel aus F abgeleitet werden kann, dann ist F unerfüllbar.
- *Vollständigkeit*: Wenn F unerfüllbar ist, dann ist die leere Klausel aus F ableitbar.

Beispiel mit otter

Wir wollen zeigen, daß

$$((AK \vee BK) \wedge (AK \rightarrow BK) \wedge (BK \wedge RL \rightarrow \neg AK) \wedge RL) \rightarrow (\neg AK \wedge BK)$$

gültig ist. Das ist genau dann der Fall, wenn

$$(AK \vee BK) \wedge (\neg AK \vee BK) \wedge (\neg BK \vee \neg RL \vee \neg AK) \wedge RL \wedge (AK \vee \neg BK)$$

unerfüllbar ist. (Wegen: $F \rightarrow G$ gültig gdw. $F \wedge \neg G$ unerfüllbar.)

Beispiel mit otter

Wir verwenden `otter` – einen Theorembeweiser basierend auf Resolution.

`otter` – <http://www.cs.unm.edu/~mccune/otter/>

- Mit Hilfe von `otter` kann man Resolutionsbeweise durchführen.
- `otter` ist eigentlich für die Prädikatenlogik gedacht, für die Aussagenlogik ist im allgemeinen ein aussagenlogischer SAT-Solver (wie `limboole`) angebracht.
- `otter` ist inzwischen zu `prover9` weiterentwickelt worden, wir verwenden in der Vorlesung jedoch trotzdem `otter`, weil man in der Ausgabe sehr schön die Resolutionsbeweise nachvollziehen kann.

Beispiel mit otter

Eingabesyntax (Beispielformel)

```
set(prolog_style_variables).  
set(binary_res).  
clear(unit_deletion).  
clear(factor).  
list(sos).  
ak | bk.           % (ak | bk) in limboole-Syntax  
-ak | bk.          % (!ak | bk)  
-bk | -rl | -ak.  % (!bk | !rl | !ak)  
rl.                % rl  
ak | -bk.          % (ak | !bk)  
end_of_list.
```

Die Optionen zu Beginn stellen sicher, dass die in der Vorlesung eingeführte Art der Resolution verwendet wird.

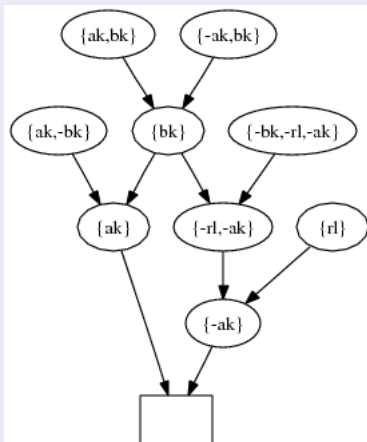
Beispiel mit otter

Textuelle Ausgabe von otter

```
----- PROOF -----  
1 [] ak|bk.  
2 [] -ak|bk.  
3 [] -bk| -r1| -ak.  
4 [] r1.  
5 [] ak| -bk.  
6 [binary,2.1,1.1] bk.  
7 [binary,5.2,6.1] ak.  
8 [binary,3.1,6.1] -r1| -ak.  
11 [binary,8.1,4.1] -ak.  
12 [binary,11.1,7.1] $F.  
----- end of proof -----
```

Beispiel mit otter

Ausgabe von otter – visuell mit GraphViz aufbereitet

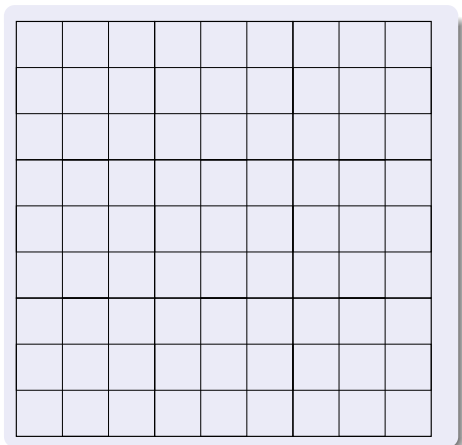


Anwendung: Sudoku

Beispiel: ein Sudoku-Solver, basierend auf Erfüllbarkeitstests

Sudoku-Regeln

Ein Sudoku-Feld ist ein Schachbrett mit 9 Zeilen und 9 Spalten. Es ist außerdem unterteilt in 9 Regionen, bestehend aus 3×3 Feldern.



Anwendung: Sudoku

Beispiel: ein Sudoku-Solver, basierend auf Erfüllbarkeitstests

Sudoku-Regeln

Zu Beginn sind einige dieser Felder mit Zahlen aus der Menge $\{1, \dots, 9\}$ gefüllt.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Anwendung: Sudoku

Beispiel: ein Sudoku-Solver, basierend auf Erfüllbarkeitstests

Sudoku-Regeln

Die Aufgabe besteht nun darin, das Feld vollständig auszufüllen, so dass in jeder Zeile, in jeder Spalte und in jeder Region jede Zahl zwischen 1 und 9 genau einmal vorkommt.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Anwendung: Sudoku

Wir verwenden folgende atomare Formeln, um ein Sudoku zu beschreiben: $S_{i,j,k}$ mit $i, j, k \in \{1, \dots, 9\}$.

- $\mathcal{A}(S_{i,j,k}) = 1$ bedeutet dabei, dass sich auf Feld (i, j) die Zahl k befindet.

Mit Hilfe dieser $9 \cdot 9 \cdot 9 = 729$ atomaren Formeln lassen sich nun die Sudoku-Spielregeln mit Hilfe der Aussagenlogik formalisieren.

Anwendung: Sudoku

- Auf jedem Feld befindet sich eine Zahl:

$$\bigwedge_{i,j \in \{1, \dots, 9\}} \bigvee_{k \in \{1, \dots, 9\}} S_{i,j,k}$$

- Auf keinem Feld befinden sich zwei Zahlen:

$$\bigwedge_{i,j \in \{1, \dots, 9\}} \bigwedge_{k_1, k_2 \in \{1, \dots, 9\}, k_1 \neq k_2} (\neg S_{i,j,k_1} \vee \neg S_{i,j,k_2})$$

- In keiner Zeile, Spalte, Region kommt eine Zahl doppelt vor:
für Zeile 1 lautet die Formel wie folgt

$$\bigwedge_{j_1, j_2 \in \{1, \dots, 9\}, j_1 \neq j_2} \bigwedge_{k \in \{1, \dots, 9\}} (\neg S_{1,j_1,k} \vee \neg S_{1,j_2,k})$$

(Für die restlichen Zeilen und für die Spalten und Regionen ergeben sich analoge Formeln).

Anwendung: Sudoku

Für die Erzeugung dieser Formel als `limboole`-Formel bietet es sich an, eine Programm zu schreiben, das die (sehr große) Formel in eine Datei schreibt. Die Datei besteht aus ca. 220.000 Zeichen. Anschließend muss nur noch die Situation in einem bestimmten Sudoku-Rätsel dargestellt werden, wie z.B. für das obige Beispiel:

```
S115 & S123 & S157 & S216 & S241 & S259 & S265 &  
S329 & S338 & S386 & S418 & S456 & S493 & S514 &  
S548 & S563 & S591 & S617 & S652 & S696 & S726 &  
S772 & S788 & S844 & S851 & S869 & S895 & S958 &  
S987 & S999
```

Die entstehende Formel ist erfüllbar und `limboole` gibt (trotz der Größe der Formel) ohne erkennbare Zeitverzögerung die Lösung aus.