DPLL und SLD Resolution

Einführung in die logische Programmierung

2018

Michael Leuschel

Wiederholung: Resolution

- Aus Klauseln: $p \vee \alpha$ und $\neg p \vee \beta$ leiten wir neue Klausel ab: $\alpha \vee \beta$
  - p atomare Ausage
  - α, β beliebige Formel, kann leer sein
  - Anmerkung: Disjunktion ist assoziativ und kommutativ:
    - $s \vee (\neg \mathbf{t} \vee r) \equiv (s \vee \neg \mathbf{t}) \vee r \equiv \neg \mathbf{t} \vee (s \vee r)$
    - Eine Klausel kann also als Menge an Literalen angesehen werden: {s,¬t,r}
- Um Resolution gut anwenden zu können wandeln wir unsere Formeln in KNF um

KNF Umwandlung: Ritter-Puzzle

1: A sagt: “B ist ein Schurke oder C ist ein Schurke”
2: B sagt“A ist ein Ritter”

- Theorie:
- $(A \Leftrightarrow \neg B \vee \neg C) \wedge (B \Leftrightarrow A)$
- $(A \rightarrow \neg B \vee \neg C) \wedge(\neg B \vee \neg C \rightarrow A) \wedge (B \rightarrow A) \wedge (A \rightarrow B)$
- $(\neg A \vee \neg B \vee \neg C) \wedge(\neg(\neg B \vee \neg C) \vee A) \wedge (\neg B \vee A) \wedge (\neg A \vee B)$
- $(\neg A \vee \neg B \vee \neg C) \wedge((B \wedge C) \vee A) \wedge (\neg B \vee A) \wedge (\neg A \vee B)$
- $(\neg A \vee \neg B \vee \neg C) \wedge(B \vee A) \wedge (C \vee A) \wedge(\neg B \vee A) \wedge (\neg A \vee B)$
- **$\{\neg A \vee \neg B \vee \neg C, B \vee A, C \vee A, \neg B \vee A, \neg A \vee B\}$**

- Query (negiert):
- $\neg(A \wedge B \wedge \neg C)$
- **$\{\neg A \vee \neg B \vee C\}$**

```
c CanonicalCNFFormat
p cnf 3 6
-1 -2 -3 0
1 2 0
1 3 0
1 -2 0
-1 2 0
-1 -2 3 0
```

KNF in Mengen-Notation

- Einzelne Formel in KNF:
  - $(\neg A \vee \neg B \vee \neg C) \wedge (B \vee A) \wedge (C \vee A) \wedge (\neg B \vee A) \wedge (\neg A \vee B)$
- Schreiben wir als Menge von Klauseln:
  - $\{\neg A \vee \neg B \vee \neg C, B \vee A, C \vee A, \neg B \vee A, \neg A \vee B\}$
- Jede Klausel selber kann als Menge gesehen werden:
  - $\{ \{\neg A,\neg B,\neg C\}, \{B,A\}, \{C,A\}, \{\neg B,A\}, \{\neg A,B\} \}$
- Resolution in dieser Notation:
  - Falls $p \in C$, $\neg p \in C'$ kann man Resolution anwenden:
    neue Klausel: $(C \setminus \{p\}) \cup (C' \setminus \{\neg p\})$

DPLL

- Gegegeben einer Aussagenlogikformel in KNF, bestimmen
  - Ob die Formel erfüllbar ist (d.h.: gibt es mindestens ein Modell?)
  - Und falls ja, ein Modell ausgeben

DPLL – partielle Interpretationen

- DPLL baut inkrementell Interpretationen auf in der Hoffnung Modelle zu finden

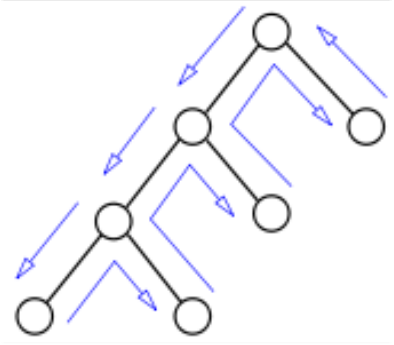
- Beispiel:
  - **{¬A ∨ ¬B ∨ ¬C, B ∨ A, C ∨ A, ¬B ∨ A, ¬A ∨ B, ¬A ∨ ¬B ∨ C}**
- **Interpretation: {A,B,C}** ⟶ {true,false}
- **Modell:**
  - Interpretation die alle Klauseln wahr macht (Konjunktion an Klauseln)
  - Modell macht eine Klausel wahr wenn mindestens ein Literal wahr ist (Disjunktion an Literalen)

- **Partielle Interpretation:**eine Abbildung von einer **Teilmenge** der Aussagen nach {true,false}

DPLL (Davis-Putnam-Logemann-Loveland) Algorithmus

- Resolutions-basiertes Verfahren
- Findet auch Modelle
- Grundlage der SAT- und SMT-Solver

- Zwei Rückgabewerte:
  - SAT: ein Modell wurde generiert
  - UNSAT: kein Modell existiert



Literal auf wahr setzen: $\mathbf{KNF}|_{\{Lit\}}$

- Was muss man mit Klauseln (KNF) machen wenn man ein Literal wahr macht?
- Beispiel: wir machen B wahr in folgenden Klauseln:
- KNF={ {¬A,¬B,¬C}, {**B**,A}, {C,A}, {¬B,A}, {¬A,**B**} }
  - Alle Klauseln in denen **B** positiv aufkommt können gelöscht werden; diese sind nun erfüllt
  - { {¬A,¬**B**,¬C}, {C,A}, {¬**B**,A} }
  - Nun wenden wir Resolution von **B** mit allen Klauseln an in denen ¬**B** auftaucht:
  - { {¬A, ¬C}, {C,A}, { A} } = $KNF|_{\{B\}}$
  - Die Klausel { A} ist eine “Unit-Clause”: A ist nun gezwungermaßen wahr

Basic DPLL (‘60, ‘62)

- Baum-basierte Suche nach einem Modell, guided by the clauses of Φ.

```
DPLL-recursive(formula F, partial assignment p)
  (F,p) = Unit-Propagate(F, p);
  If F contains clause {} then
        return (UNSAT, null);
  If F = {} then
        return (SAT, p);
  x = literal such that x and ¬x are not in p;
  (status, p’) = DPLL-recursive(F|{x}, pU{x});
  If status == SAT then
        return (SAT, p’);
  Else return
        DPLL-recursive(F|{¬x}, pU{¬x});
```

If a clause tells you the value of a var, set it appropriately.

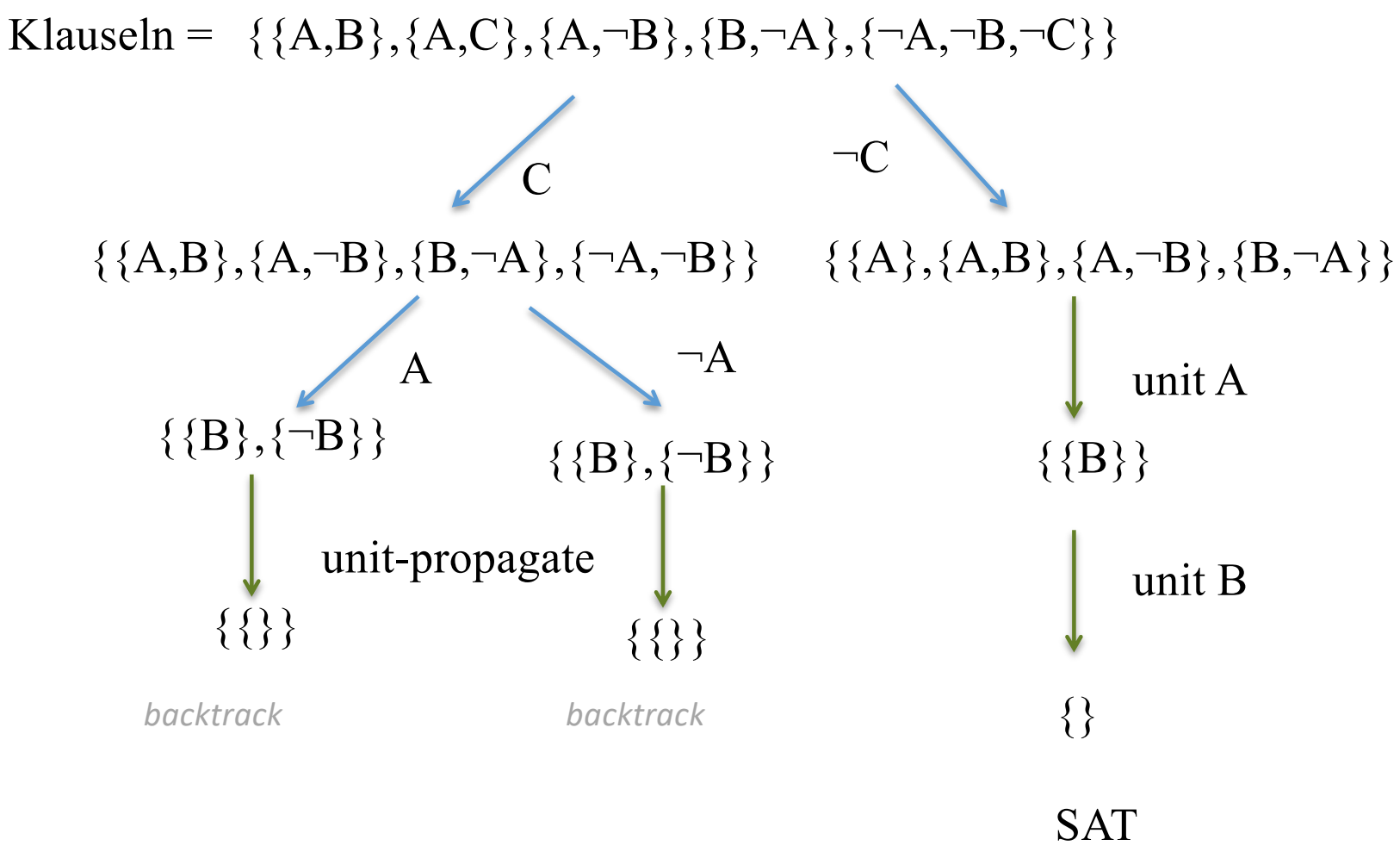
Choose a branch. Many heuristics to choose from.

Basic DPLL

If a clause tells you the value of a variable, set it appropriately.

```
Unit-Propagate(formula F, partial assignment p)
        If F has no empty clause then
                While F has a unit clause {x}
                        F = F|{x};
                        p = p U {x};
        return (F,p)
```

Ausführungsbeispiel DPLL



Prolog Version von DPLL - 1

- ```
  problem(1,'Knights & Knaves',
      [ [neg(a),neg(b),neg(c)],
        [pos(b),pos(a)],
        [pos(c),pos(a)],
        [neg(b),pos(a)],
        [neg(a),pos(b)] ]).
  ```
- `negate(pos(A),neg(A)).`
- `negate(neg(A),pos(A)).`
  ```

Prolog Version von DPLL – 2 – KNF|{Lit}

- ```
  becomes_true(TrueLit,Clause) :-
        member(TrueLit,Clause).
  ```
- ```
  simplify(FalseLit,Clause,SimplifedClause) :-
      delete(Clause,FalseLit,SimplifedClause).
      % Resolution
  ```

- ```
  set_literal(Lit,Clauses,NewClauses) :-
      exclude(becomes_true(Lit),Clauses,Clauses2),
      negate(Lit,NegLit),
      maplist(simplify(NegLit),Clauses2,
                               NewClauses).
  ```
  ```

Prolog Version von DPLL – 3 – Algorithm

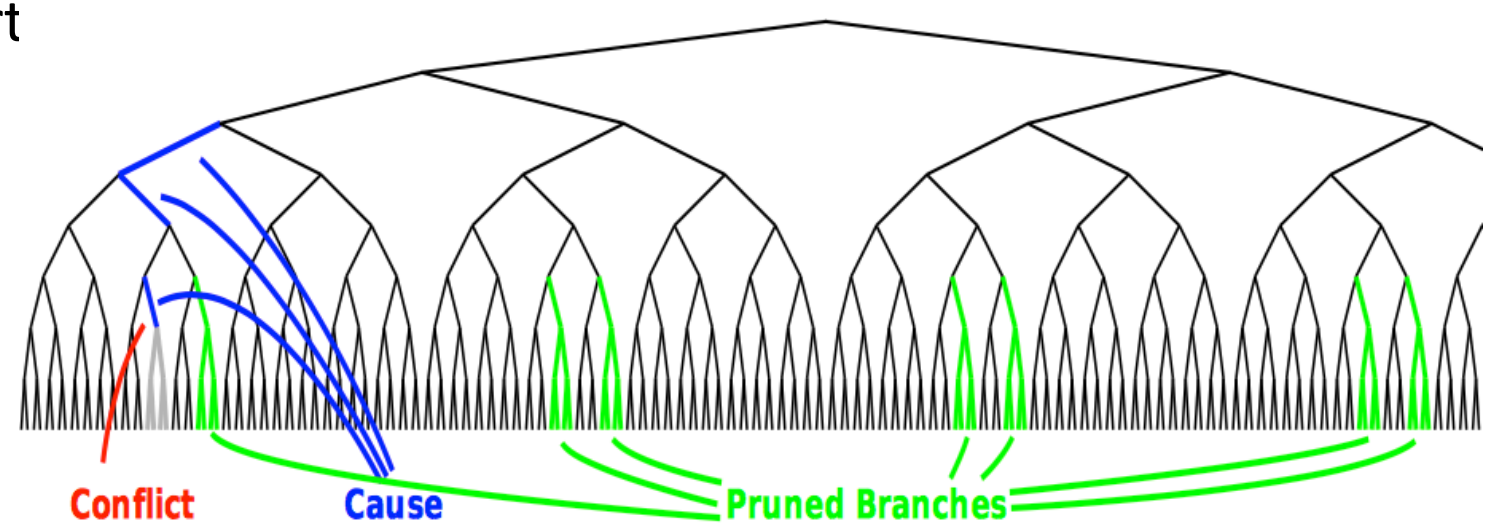
```
• dpll(Clauses,[unit(Lit)|Stack]) :-
      select([Lit],Clauses,Clauses2), % unit clause found
      set_literal(Lit,Clauses2,Clauses3),
      dpll(Clauses3,Stack).
• dpll([],Stack) :- !, Stack=[]. % SAT
• dpll(Clauses,[branch(Lit)|Stack]) :-
      nonmember([],Clauses), % no inconsistency
      choose_literal(Clauses,Lit),
      set_literal(Lit,Clauses,Clauses2),
      dpll(Clauses2,Stack).

• choose_literal([ [Lit|_] | _], Lit).
• choose_literal([ [Lit|_] | _], NegLit) :-
      negate(Lit,NegLit).
```

Optimierungen

- Pure Literals: wenn ein Literal nur positiv oder nur negativ auftaucht kann man es setzen und die betroffenen Klauseln löschen
  - Wird aus Performance Gründen oft nicht gemacht
- Watched Literals und Unit-Propagation:
  - Für jede Klausel beobachtet man 2 Literale (wenn es nur 1 Literal gibt hat man eine Unit-Klausel)
    - Damit aus der Klausel eine Unit-Klausel wird muss mindestens eines der beiden Literale gesetzt werden
    - Beispiel: {¬A,¬B,¬C,D} : wir beobachten nur A und B
      - {**¬A,¬B**,¬C,D} ⇝ C= true ⇝ wir beobachten immer noch A und B
      - {**¬A,¬B**,¬C,D} ⇝ B=true ⇝ wir beobachten nun A und D
      - {**¬A**,¬B,¬C,**D**} ⇝ D=false ⇝ nun haben wir eine Unit-Klausel: setze A auf false
  - (es gibt einen eleganten Prolog SAT-Solver von Howe&King der “watched literals” mit Koroutinen implementiert)

Andere Optimierungen

- Nicht-chronologisches Backtracking
- Konflikt-Klauseln lernen
- Variable-Ordering
- Randomised Restart
- …



SAT Solver

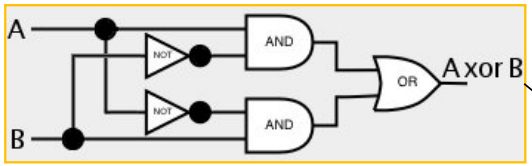
- Sat4J (http://www.sat4j.org)
- MiniSat
- Glucose
- Lingeling, Plingeling
- …

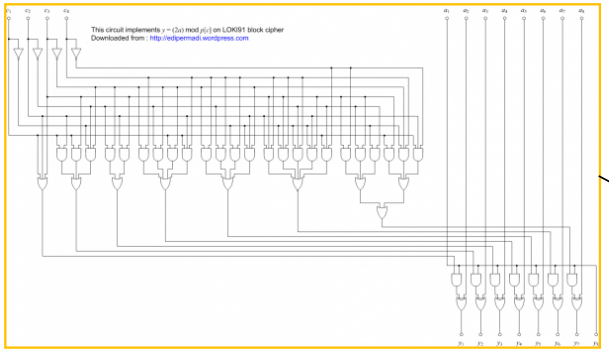
- Wettbewerbe: http://www.satcompetition.org

	Gold	Silver	Bronze	Gold	Silver	Bronze	Gold	Silver	Bronze
		Agile Track			Main Track			Random Track	
SAT+UNSAT	Riss	TB_Glucose	CHBR_Glucose	MapleCOMSPS	Riss	Lingeling	Dimetheus	CSCCSat	DCCAlm
		Parallel Track			No-Limit Track			Incremental Library Track	
SAT+UNSAT	Treengeling	Plingeling	CryptoMiniSat	BreakIDCOMiniSatPS	Lingeling	abcdSAT	CryptoMiniSat	Glucose	Riss
	Best Application Benchmark Solver in the Main Track			Best Crafted Benchmark Solver in the Main Track			Best Glucose Hack in the Main Track		
SAT+UNSAT	MapleCOMSPS			TC Glucose			Kiel		

```
$ java -jar /Applications/Development/SAT/sat4j-core-
v20130525/org.sat4j.core.jar KnightKnaves.cnf
c SAT4J: a SATisfiability library for Java (c) 2004-2013 Artois
University and CNRS
c This is free software under the dual EPL/GNU LGPL licenses.
c See www.sat4j.org for details.
c version 2.3.5.v20130525
c java.runtime.name Java(TM) SE Runtime Environment
c java.vm.name                Java HotSpot(TM) 64-Bit Server VM
c java.vm.version   25.73-b02
c java.vm.vendor    Oracle Corporation
c sun.arch.data.model         64
c java.version                1.8.0_73
c os.name           Mac OS X
c os.version                  10.12.1
c os.arch           x86_64
c Free memory                 125533992
c Max memory                  1908932608
c Total memory                128974848
c Number of processors        4
c --- Begin Solver configuration ---
c
org.sat4j.minisat.constraints.MixedDataStructureDanielWL@65b3120
a
c Learn all clauses as in MiniSAT
c claDecay=0.999 varDecay=0.95 conflictBoundIncFactor=1.5
initConflictBound=100
c VSIDS like heuristics from MiniSAT using a heap lightweight
component caching from RSAT
c Expensive reason simplification
c Glucose 2.1 dynamic restart strategy
c Glucose 2 learned constraints deletion strategy
c timeout=2147483647s
c DB Simplification allowed=true
c Listener: org.sat4j.minisat.core.VoidTracing@79fc0f2f
c --- End Solver configuration ---
c solving KnightKnaves.cnf
c reading problem ...
c ... done. Wall clock time 0.005s.
c declared #vars     3
c #constraints  5
c constraints type
c org.sat4j.minisat.constraints.cnf.OriginalBinaryClause => 4
c org.sat4j.minisat.constraints.cnf.OriginalWLClause => 1
c 5 constraints processed.
c starts            : 1
c conflicts                   : 1
c decisions                   : 1
c propagations                : 4
c inspects                    : 8
c shortcuts                   : 0
c learnt literals   : 1
c learnt binary clauses       : 0
c learnt ternary clauses      : 0
c learnt constraints          : 0
c ignored constraints         : 0
c root simplifications        : 2
c removed literals (reason simplification)        : 0
c reason swapping (by a shorter reason) : 0
c Calls to reduceDB : 0
c Number of update (reduction) of LBD   : 0
c Imported unit clauses       : 0
c speed (assignments/second)  : 2000.0
c non guided choices          1
c learnt constraints type
c constraints type
c org.sat4j.minisat.constraints.cnf.OriginalBinaryClause => 4
c org.sat4j.minisat.constraints.cnf.OriginalWLClause => 1
c 5 constraints processed.
s SATISFIABLE
v 1 2 -3 0
c Total wall clock time (in seconds) : 0.01
```

Praktische Anwendungen





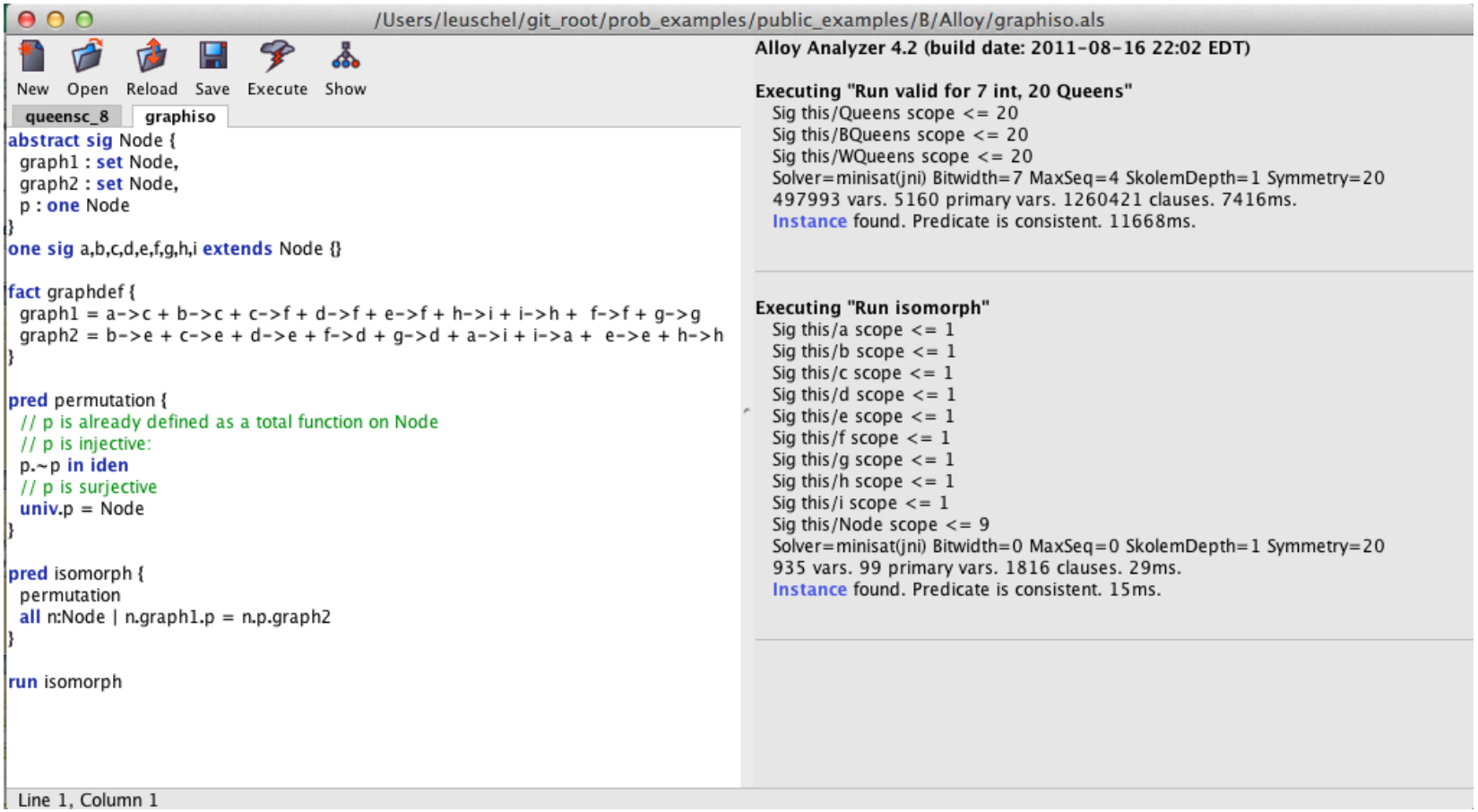
Problem

```
if(x==null) {
  search(x.value)
…
}
```

Formel
In Aussagenlogik

SAT Solver

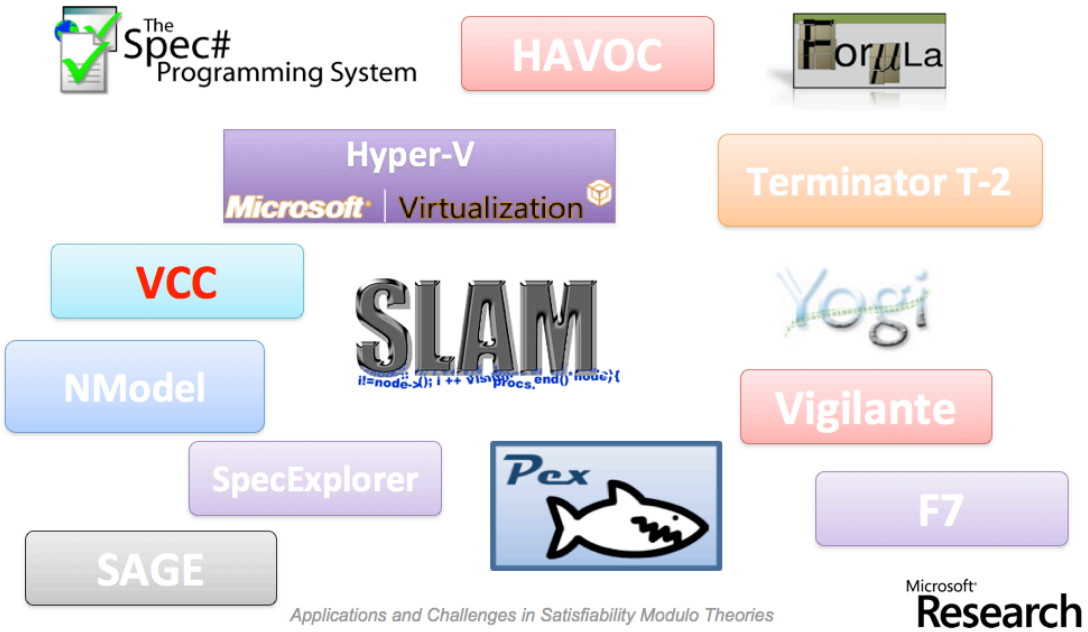
```
Solver=minisat(jni) Bitwidth=5 MaxSeq=3 SkolemDepth=1 Symmetry=20
   118805 vars. 2368 primary vars. 293152 clauses. 1462ms.
   No instance found. 44ms.
```


/Users/leuschel/git_root/prob_examples/public_examples/B/Alloy/graphiso.als
New
Open
Reload
Save
Execute
Show
queensc_8
graphiso
abstract sig Node {
graph1 : set Node,
graph2 : set Node,
p : one Node
}
one sig a,b,c,d,e,f,g,h,i extends Node {}
fact graphdef {
graph1 = a->c + b->c + c->f + d->f + e->f + h->i + i->h + f->f + g->g
graph2 = b->e + c->e + d->e + f->d + g->d + a->i + i->a + e->e + h->h
}
pred permutation {
// p is already defined as a total function on Node
// p is injective:
p.~p in iden
// p is surjective
univ.p = Node
}
pred isomorph {
permutation
all n:Node | n.graph1.p = n.p.graph2
}
run isomorph
Line 1, Column 1
Alloy Analyzer 4.2 (build date: 2011-08-16 22:02 EDT)
Executing "Run valid for 7 int, 20 Queens"
Sig this/Queens scope <= 20
Sig this/BQueens scope <= 20
Sig this/WQueens scope <= 20
Solver=minisat(jni) Bitwidth=7 MaxSeq=4 SkolemDepth=1 Symmetry=20
497993 vars. 5160 primary vars. 1260421 clauses. 7416ms.
Instance found. Predicate is consistent. 11668ms.
Executing "Run isomorph"
Sig this/a scope <= 1
Sig this/b scope <= 1
Sig this/c scope <= 1
Sig this/d scope <= 1
Sig this/e scope <= 1
Sig this/f scope <= 1
Sig this/g scope <= 1
Sig this/h scope <= 1
Sig this/i scope <= 1
Sig this/Node scope <= 9
Solver=minisat(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
935 vars. 99 primary vars. 1816 clauses. 29ms.
Instance found. Predicate is consistent. 15ms.

SMT (SATisfiability Modulo Theories)

- Erweiterung von SAT:
  - Aussagen sind Prädikate in einer Theorie
  - aus {¬A,¬B,¬C,D} wird {¬(x>10),¬(x>20),¬(x<y),(y<10)}
  - Es gibt also Abhängikeiten zwischen den Aussagen
    - Wenn B wahr ist, dann ist auch A wahr, wenn A falsch ist dann ist auch B falsch,…
  - Es gibt eine verschiedene Theorien: lineare Arithmetik, Bitvektoren,…
- Bekanntester Solver: Z3 von Microsoft

Some Applications @ Microsoft



Microsoft Research

Lineare Resolution in Prolog

Horn Klauseln

- Literal = p oder $\neg$p wo p eine atomare Aussage ist
  (p: positives, $\neg$p: negatives Literal)
- Klausel= Disjunktion von Literalen
  - Bsp: $\neg p \vee q$, wet $\vee \neg$rains $\vee \neg$outside

- **Horn** Klausel wenn maximal 1 positives Literal
- Anfrage (Denial) wenn kein positives Literal

Prolog Programme und KNF (Wdh.)

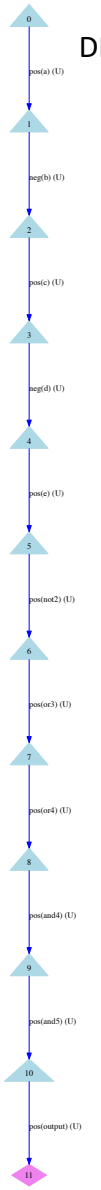
- Anmerkung: $\neg p \vee q \equiv p \rightarrow q$
- Prolog
  - `wet :- rains, outside.`
  - Dies steht für die logische Formel:

    wet $\leftarrow$ rains $\wedge$ outside

  - Diese ist logisch äquivalent zu

    wet $\vee \neg$rains $\vee \neg$outside

  - Jede Prolog Regel steht für eine Horn Klausel !



Übung: Schaltung per Resolution
Was macht Prolog ?

```
% Die Belegung von der Aufgabe:
a. not_b. c. not_d. e.

% Schaltungen der ersten Ebene
and11 :- a,b.

or11 :- b.
or11 :- c.

and12 :- c,d.

not1 :- not_e.

% Schaltungen der zweiten Ebene
or21 :- and11.
or21 :- not1.

and2 :- or11, not1.

or22 :- and12.
or22 :- not1.

not2 :- e.  % \+ not1.
```

```
% Schaltungen der dritten Ebene
and3 :- or21, and2.

or3 :- or22.
or3 :- not2.

% Schaltungen der vierten Ebene
or4 :- and3.
or4 :- or3.

and4 :- or3, not2.

% Letzte Ebene
and5 :- and4, or4.

output :- and5.
```

Programm $\Rightarrow_?$ output

Lineare Resolution

T:
$p \vee \neg q$
q

D: $\neg p$

D': $\neg q$

- 1. Nehme eine Theorie T in KNF mit Horn Klauseln, aber ohne Anfragen
- 2. Nehme eine Anfrage D
- 3. Führe Resolution von D mit einer Klausel aus T aus
  - wir erhalten eine neue Anfrage **D'**
- 4. Falls **D'**=□ sind wir fertig
- 5. Sonst setze D :=**D'** (vergesse die alte Anfrage) and gehe zu Schritt 3 zurück
- Beobachtungen:
  - Negative Literale kommen immer aus D,
  - Positive Literale kommen immer aus T, T bleibt unverändert
  - In Schritt 3 können mehrere Klauseln aus T anwendbar sein

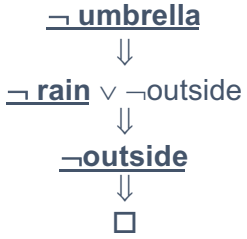
Lineare Resolution und Prolog

```
umbrella :- rain, outside.
rain.
outside.

?- umbrella.

yes
```

$$\text{umbrella} \leftarrow \text{rain} \wedge \text{outside}$$

$$\text{umbrella} \vee \neg\, \text{rain} \vee \neg \text{outside}$$

rain
outside



Linear =
Man kann die vorherigen Anfragen "vergessen"

Selektions-Funktion und SLD

- "Auswahlregel": welches negative Literal in der aktuellen Anfrage soll benutzt werden:
  - $\neg$ **rain** $\vee$ $\neg$outside
  - $\neg$ **rain** $\vee$ $\neg$outside
- SLD Resolution: lineare Resolution parametrisiert durch eine Selektions-Funktion

SLD-Resolution

- S: Selektions-Funktion
- L: Lineare Resolution
- D: Definite Klauseln (Horn Klauseln, keine Negation im Rumpf von Klauseln)

- definiert die Ausführung von logischen Programmen

Terminologie

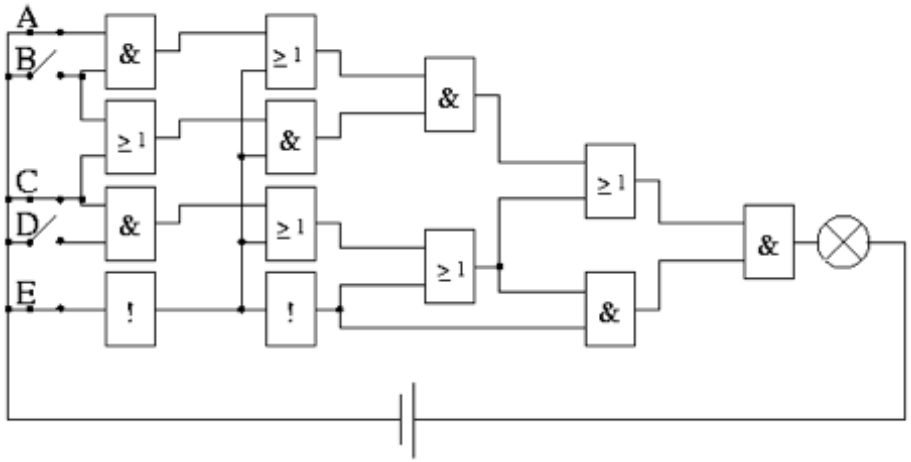
- **SLD-Ableitung** für $P \cup \{G\}$
  - Folge von Anfragen G_i :
    - Anwendung der Auswahlregel auf G_i
    - **Resolutionsschritt** mit ausgewähltem Literal von G_i und einer Programmklausel um die nächste Anfrage G_{i+1} abzuleiten

- **SLD-Refutation** für $P \cup \{G\}$
  - SLD-Ableitung die mit dem Widerspruch $\square$ endet

```
% Die Belegung von der Aufgabe:
a. not_b. c. not_d. e.
% Schaltungen der ersten Ebene
and11 :- a,b.
or11 :- b.
or11 :- c.
and12 :- c,d.
not1 :- not_e.
% Schaltungen der zweiten Ebene
or21 :- and11.
or21 :- not1.
and2 :- or11, not1.
or22 :- and12.
or22 :- not1.
not2 :- e.  % \+ not1.
% Schaltungen der dritten Ebene
and3 :- or21, and2.
or3 :- or22.
or3 :- not2.
% Schaltungen der vierten Ebene
or4 :- and3.
or4 :- or3.
and4 :- or3, not2.
% Letzte Ebene
and5 :- and4, or4.
output :- and5.
```

eine erfolgreiche SLD-Ableitung ?

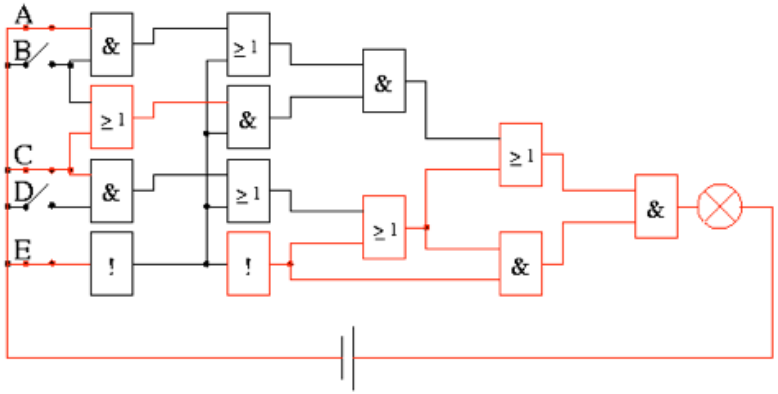
```
?- output.
...
```



```
% Die Belegung von der Aufgabe:
a. not_b. c. not_d. e.
% Schaltungen der ersten Ebene
and11 :- a,b.
or11 :- b.
or11 :- c.
and12 :- c,d.
not1 :- not_e.
% Schaltungen der zweiten Ebene
or21 :- and11.
or21 :- not1.
and2 :- or11, not1.
or22 :- and12.
or22 :- not1.
not2 :- e.  % \+ not1.
% Schaltungen der dritten Ebene
and3 :- or21, and2.
or3 :- or22.
or3 :- not2.
% Schaltungen der vierten Ebene
or4 :- and3.
or4 :- or3.
and4 :- or3, not2.
% Letzte Ebene
and5 :- and4, or4.
output :- and5.
```

eine erfolgreiche SLD-Ableitung

```
?- output.
?- and5.
?- and4,or4.
?- or3,not2,or4.
?- not2,not2,or4.
?- e,not2,or4.
?- not2,or4.
?- e,or4.
?- or4.
?- or3.
?- not2.
?- e.
[]
```

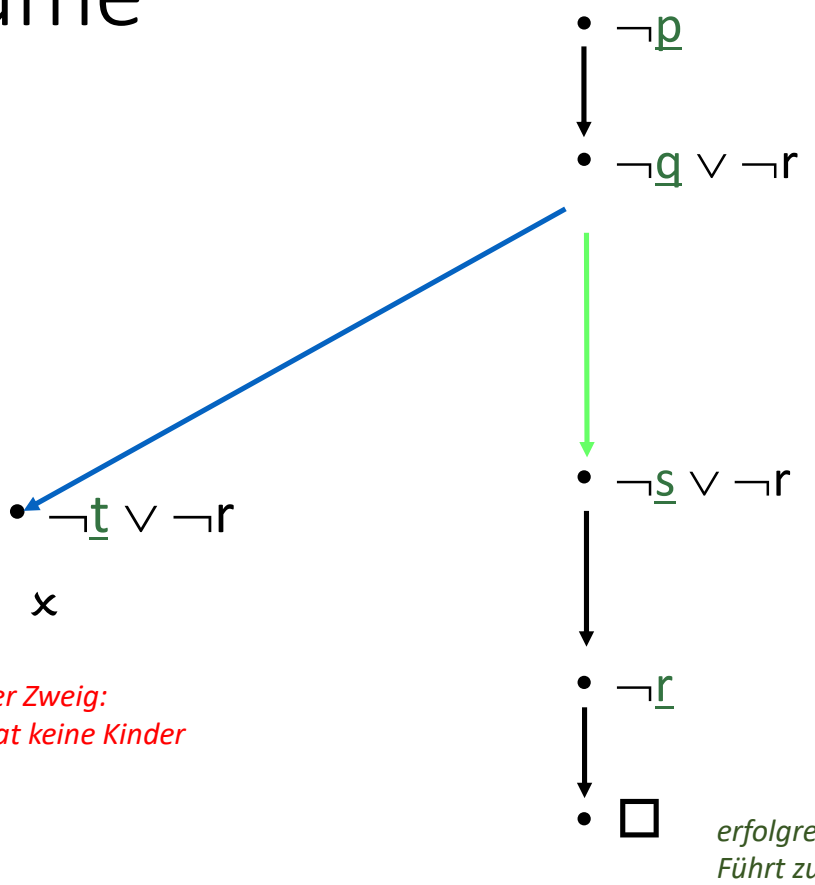


SLD-Bäume: Definition

- **SLD-Baum** für $P \cup \{\neg Q\}$:
- **Wurzel** des Baumes:
  - Anfrage: $\neg Q$
- **Blätter** sind entweder:
  - Erfolg $\square$ oder
  - Blätter wo keine Klausel auf das ausgewählte Literal passt

- Für jeden Knoten (ausser $\square$):
  - Ausgewähltes Literal L ist <u>unterstrichen</u>
  - Die **Kinder** sind alle möglichen Resolventen (Resolution von L mit einer Klausel aus P)

SLD-Bäume




```
p :- q,r.
q :- t.
q :- s.
r.
s.
```

Fehlgeschlagener Zweig:
Dieser Knoten hat keine Kinder

erfolgreicher Zweig:
Führt zu einem Widerspruch

Prolog: Tiefensuche (Depth-First), Selection Rule: erstes Literal links

SLD-Bäume und Prolog

- Auswahlregel von Prolog wählt immer das erste Literal in der Anfrage aus (left-to-right)
  - (man kann dies aber durch Block/When Annotationen in modernen Prologsystemen ändern)
- Prolog durchläuft den SLD-Baum per Tiefensuche mit Backtracking bei einem Fehlschlag
  - Die Reihenfolge der Klauseln im Programm ist für die Reihenfolge der Abarbeitung relevant

Arten von SLD Bäumen

- Fehlgeschlagen (Failed) vs Ergolgreich (Successful)
  - Erfolgreich: es gibt mindestens ein Blatt mit □
  - Fehlgeschlagen: alle anderen Fälle
- Endlich vs Unendlich

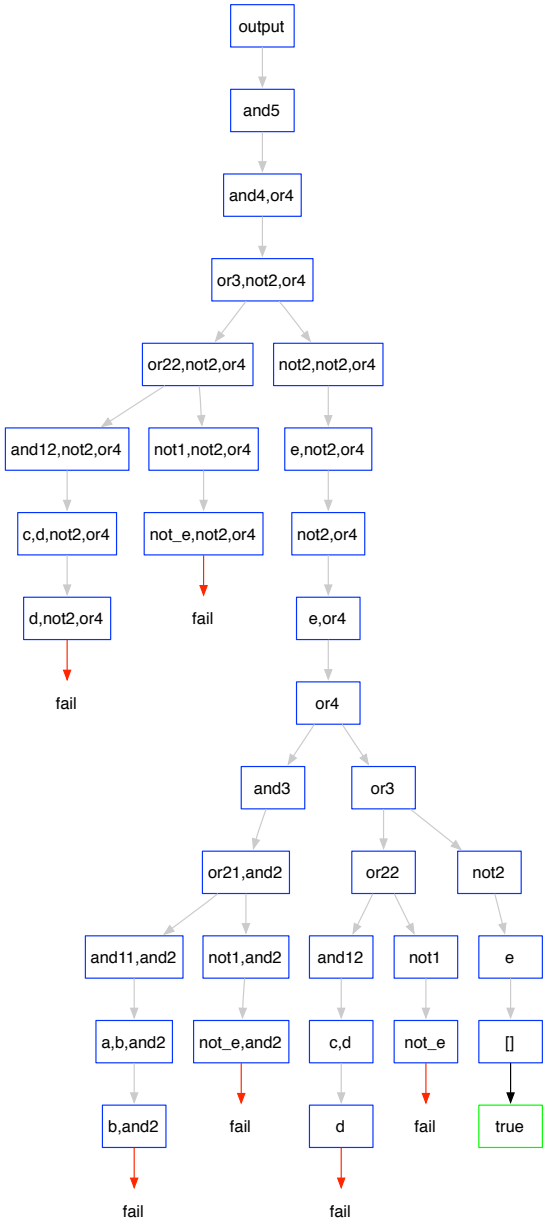
Theorem

- $P \cup \{\neg Q\}$ ist unerfüllbar
  - gdw
- der SLD-Baum für $P \cup \{\neg Q\}$ erfolgreich ist

- Gilt für jede Auswahlregel
- Achtung: der Baum kann unendlich sein
- Achtung: je nach Auswahlregel kann der Baum für die gleiche Anfrage und das gleiche Programm endlich oder unendlich sein

Beispiel für einen SLD-Baum

```
% Die Belegung von der Aufgabe:
a. not_b. c. not_d. e.
% Schaltungen der ersten Ebene
and11 :- a,b.
or11 :- b.
or11 :- c.
and12 :- c,d.
not1 :- not_e.
% Schaltungen der zweiten Ebene
or21 :- and11.
or21 :- not1.
and2 :- or11, not1.
or22 :- and12.
or22 :- not1.
not2 :- e.   % \+ not1.
% Schaltungen der dritten Ebene
and3 :- or21, and2.
or3 :- or22.
or3 :- not2.
% Schaltungen der vierten Ebene
or4 :- and3.
or4 :- or3.
and4 :- or3, not2.
% Letzte Ebene
and5 :- and4, or4.
output :- and5.
```



Übung: Finden Sie $P \cup \{\neg Q\}$ für:

- einen endlichen fehlgeschlagenen SLD-Baum (finitely-failed SLD-tree)
- einen unendlichen fehlgeschlagenen SLD-Baum
- einen unendlichen erfolgreichen SLD-Baum

- einen unendlichen erfolgreichen SLD-Baum in dem Prolog keine Lösung findet

Nicht Lineare Resolution

- One day Tokusan told his student Gantõ,
  "*I have two monks who have been here for many years. Go and examine them.*"
- Gantõ picked up an ax, saying,
  "*If you say a word I will cut off your heads;*
  and if you do not say a word, I will also cut off your heads."

1) cut $\vee \neg$ say
2) cut $\vee$ say

Nicht-lineare Resolution nötig um zu zeigen, dass "cut" eine logische Konsequenz ist.

Beobachte: 2) ist keine Horn Klausel

Linear =
Man kann die vorherigen Anfragen "vergessen"

Lineare Resolution mit Prädikatenlogik

```
umbrella(City) :- rain(City), outside(City).
rain(glasgow).
rain(wuppertal).
rain(berlin).
outside(berlin).
?- umbrella(X).
X=berlin
```

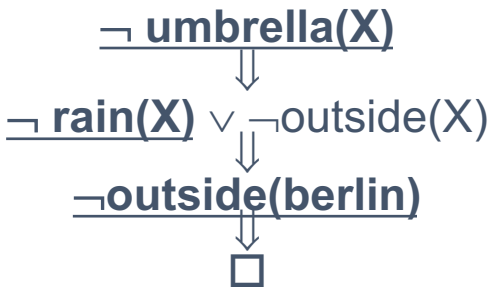
$$\forall \text{ City} . \text{umbrella(City)} \leftarrow \text{rain(City)} \wedge \text{outside(City)}$$

umbrella(City) $\vee \neg$ rain(City) $\vee \neg$outside(City)

…

rain(berlin)

outside



Resolution wird um Variablen erweitert

Beim Resolutionsschritt werden die Argumente “unifiziert”

Programmklauseln bekommen jedes Mal frische Variablen

Zusammenfassung

- SAT Solver
  - DPLL Algorithmus um Erfüllbarkeit zu prüfen und Modelle zu finden
- Lineare Resolution für Prolog
  - Verlangt KNF mit Horn Klauseln
  - Beweis durch Widerspruch
  - Ein Denial wird mit einer Horn-Klausel aus dem Programm kombiniert
  - SLD-Ableitung, SLD-Refutation, SLD-Baum