# 4_DPLL

November 15, 2022

## 1 DPLL SAT Solving as a Prolog program

We take again our Knights and Knaves puzzle from Raymond Smullyan.

- A says: "B is a knave or C is a knave"
- B says: "A is a knight"

We can convert our Knights and Knaves puzzle to conjunctive normal form (CNF):

```
(A   ¬B   ¬C)    (B   A)
(A → ¬B   ¬C)    (¬B   ¬C → A)    (B → A)  (A → B)
(¬A   ¬B   ¬C)    (¬(¬B   ¬C)   A)    (¬B   A)  (¬A   B)
(¬A   ¬B   ¬C)    ((B   C)   A)    (¬B   A)  (¬A   B)
(¬A   ¬B   ¬C)    (B   A)    (C   A)   (¬B   A)  (¬A   B)
{¬A   ¬B   ¬C,  B   A,  C   A,  ¬B   A,  ¬A   B}
```

One can encode a clause as a set of literals, and one can encode the CNF as a set of clauses. Implicitly, the literals in clause are disjoined, while all the clauses in a CNF are conjoined. In Prolog, we can represent sets using lists. Below is an encoding of the CNF as a list of lists. For example, the clause `¬A   ¬B   ¬C` is represented as the list `[neg(a),neg(b),neg(c)]`. Every literal is of either the form `neg(p)` or `pos(p)` with `p` being a proposition.

[1]:
```
:- discontiguous problem/3.
:- dynamic problem/3.
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)).
```

We can negate literals as follows:

[2]:
```
?- negate(pos(a),NA), negate(neg(b),NB).
```

```
NA = neg(a),

NB = pos(b)
```

We now show to perform resolution of two clauses. First we can pick two clauses simply using `member`:

```
[3]: ?- problem(1,_,Clauses),
        member(C1,Clauses), member(Lit1,C1).
```

```
Clauses =␣
 ↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],

C1 = [neg(a),neg(b),neg(c)],

Lit1 = neg(a)
```

We can now pick matching literals with opposing polarity as follows:

```
[4]: ?- problem(1,_,Clauses),
        member(C1,Clauses), member(Lit1,C1), negate(Lit1,Lit2),
        member(C2,Clauses), member(Lit2,C2).
     %comment.
```

```
Clauses =␣
 ↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],

C1 = [neg(a),neg(b),neg(c)],

Lit1 = neg(a),

Lit2 = pos(a),

C2 = [pos(b),pos(a)]
```

We can now implement resolution of two particular clauses as follows:

```
[5]: :- use_module(library(lists)).
     resolve(Clause1,Clause2,Resolvent) :-
        select(Lit1,Clause1,R1), negate(Lit1,NLit1),
        select(NLit1,Clause2,R2),
        append(R1,R2,Resolvent).
```

The above predicate uses the predicate `select/3` from `library(lists)`, which can be used to select an element from a list and returns a modified list with the element removed:

```
[6]: ?- select(2,[1,2,3],Res1).
```

```
Res1 = [1,3]
```

The predicate `resolve` works as follows:

```
[7]: ?- resolve([neg(a),pos(b)],[neg(b),pos(c)],Resolvent).
```

```
Resolvent = [neg(a),pos(c)]
```

```
[8]: ?- resolve([pos(b)],[neg(b)],Resolvent).
```

```
Resolvent = []
```

```
[9]: ?- resolve([neg(a),neg(b)],[neg(b),pos(c)],Resolvent).
```

```
false
```

Let us now resolve two clauses from our puzzle:

```
[10]: ?- problem(1,_,Clauses),
         member(C1,Clauses), member(C2,Clauses),
         resolve(C1,C2,NewClause).
```

```
Clauses =␣
 ↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],

C1 = [neg(a),neg(b),neg(c)],

C2 = [pos(b),pos(a)],

NewClause = [neg(b),neg(c),pos(b)]
```

Let us compute all direct resolvents:

```
[11]: ?- problem(1,_,Clauses),
         findall(NewClause,(member(C1,Clauses), member(C2,Clauses),
                            resolve(C1,C2,NewClause)
                            ), NewClauses),
         length(NewClauses,NrNew),
         sort(NewClauses,SortedC), length(SortedC,NrNewUnique).
```

```
Clauses =␣
 ↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],

NewClauses =␣
 ↪[[neg(b),neg(c),pos(b)],[neg(a),neg(c),pos(a)],[neg(b),neg(c),pos(c)],[neg(a),neg(b),pos(a)]

NrNew = 22,

SortedC =␣
 ↪[[neg(a),neg(a),neg(c)],[neg(a),neg(b),pos(a)],[neg(a),neg(c),neg(a)],[neg(a),neg(c),pos(a)]

NrNewUnique = 20
```

As you can see, the number of new clauses can grow considerably when applying resolution. Below we study the DPLL algorithm to check satisfiability of a formula in CNF, using a simple form of resolution (where one clause is a single literal and the other clauses comes from the CNF of the problem).

### 1.0.1 DPLL Algorithm

We now present bit by bit the DPLL algorithm as a Prolog program manipulating the above clause database.

The algorithm works by selecting a literal and then making it true and checking whether this leads to a solution. If not, the algorithm backtracks and makes the negated version of the literal true and tries to find a solution.

```
[12]: becomes_true(TrueLit,Clause) :-
          member(TrueLit,Clause).
```

The above checks if making a given literal true makes the second argument, a clause, true (i.e., the clause is satisfied):

```
[13]: ?- becomes_true(pos(b),[neg(a),pos(b)]).
```

```
true
```

```
[14]: ?- becomes_true(neg(a),[neg(a),pos(b)]).
```

```
true
```

The code below simplifies a clause given a literal `TrueLit` that has just become true. This is done using resolution:

```
[15]: simplify(TrueLit,Clause,SimplifedClause) :-
         negate(TrueLit,FalseLit),
         delete(Clause,FalseLit,SimplifedClause).    % Resolution with TrueLit if␣
       ↪possible
```

The code uses the `delete/3` predicate from `library(lists)`. Unlike select it also succeeds when the element is not in the list:

```
[16]: ?- delete([a,b,c],d,R).
```

```
R = [a,b,c]
```

```
[17]: ?- delete([a,b,c],a,R).
```

```
R = [b,c]
```

The above code simplifies a clause by resolution:

```
[18]: ?- simplify(pos(c),[pos(a),neg(c)],SC).
```

```
SC = [pos(a)]
```

If no resolution is possible the clause is returned unchanged:

```
[19]: ?- simplify(pos(b),[pos(a),neg(c)],SC).
```

```
SC = [pos(a),neg(c)]
```

We can now define the whole procedure to set a literal `Lit` to true (`Clauses|{Lit}`) by - removing all clauses which have become true (and no longer need to be checked) - simplifying the remaining clauses with resolution if possible.

```
[20]: set_literal(Lit,Clauses,NewClauses) :-
         exclude(becomes_true(Lit),Clauses,Clauses2),
         maplist(simplify(Lit),Clauses2,NewClauses).
```

The predicate `exclude/3` from `library(lists)` is a higher-order predicate. It maps its first argument (a predicate) over the second argument (a list) and excludes all elements where the predicate succeeds.

```
[21]: ?- exclude(becomes_true(pos(a)), [[neg(b)], [pos(a),pos(c)]],R).
```

```
R = [[neg(b)]]
```

`exclude` will perform these two calls and hence remove the second clause and keep the first one:

```
[22]: ?- becomes_true(pos(a),[neg(b)]).
      ?- becomes_true(pos(a),[pos(a),pos(c)]).
```

```
false
```

```
true
```

The predicate `maplist/3` from `library(lists)` is another higher-order predicate. It maps its first argument (a predicate) over the elements in the second and third arguments (both lists).

```
[23]: ?- maplist(simplify(neg(c)), [[neg(b),pos(c)], [pos(a),pos(c)]],R).
```

```
R = [[neg(b)],[pos(a)]]
```

`maplist` will perform these two calls and put `R1` and `R2` into the result list `R` above:

```
[24]: ?- simplify(neg(c),[neg(b),pos(c)],R1),
         simplify(neg(c),[pos(a),pos(c)],R2).
```

```
R1 = [neg(b)],
```

```
R2 = [pos(a)]
```

The predicate `set_literal` can now be called as follows:

```
[25]: ?- set_literal(pos(a),[[neg(a),neg(b),neg(c)]],Res).
```

```
Res = [[neg(b),neg(c)]]
```

```
[26]: ?- set_literal(neg(a),[[neg(a),neg(b),neg(c)]],Res).
```

```
Res = []
```

In the call above we have found a model: all clauses have been satisfied. In the next call we have found a contradiction, as the resulting list contains the empty clause (aka the obvious contradiction).

```
[27]: ?- set_literal(neg(a),[[neg(a),neg(b),neg(c)],[pos(a)]],Res).
```

```
Res = [[]]
```

Let us now use this code on our puzzle:

[28]:
```
?- problem(1,T,C1), set_literal(pos(a),C1,C2).
```

```
T = Knights & Knaves,
C1 =␣
↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],
C2 = [[neg(b),neg(c)],[pos(b)]]
```

[29]:
```
?- problem(1,_,C1), set_literal(pos(a),C1,C2), set_literal(pos(b),C2,C3).
```

```
C1 =␣
↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],
C2 = [[neg(b),neg(c)],[pos(b)]],
C3 = [[neg(c)]]
```

This is now the top-level of the DPLL algorithm. It first tries to find unit clauses, to deterministically find forced assignments. It then checks if all clauses have now been satisfied. If not, it checks for inconsistency. If there is no inconsistency, then a literal is chosen and forced to one and then if required the other value.

[30]:
```
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]) :-
    \+ member([],Clauses), % no inconsistency
    choose_literal(Clauses,Lit), % this selects one literal and returns it first␣
 ↪in original then in negated form
    set_literal(Lit,Clauses,Clauses2),
    dpll(Clauses2,Stack).

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

[31]:
```
?- problem(1,_,C1), dpll(C1,Stack).
```

```
C1 =␣
↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],
Stack = [branch(pos(a)),unit(pos(b)),unit(neg(c))]
```

[32]:
```
test(Nr) :- problem(Nr,Str,Clauses),
            format('Solving problem ~w : ~w~n',[Nr,Str]),
             (dpll(Clauses,Stack)
               -> format('SAT: Model found: ~w~n',[Stack])
               ; format('UNSAT: no model exists',[])).
```

[33]:
```
?- test(1).
```

```
Solving problem 1 : Knights & Knaves
SAT: Model found: [branch(pos(a)),unit(pos(b)),unit(neg(c))]
```

**true**

[34]:
```
problem(2,'Knights & Knaves (Proof by contradiction)',
    [ [neg(a),neg(b),neg(c)],
      [pos(b),pos(a)],
      [pos(c),pos(a)],
      [neg(b),pos(a)],
      [neg(a),pos(b)],
      [neg(a),neg(b),pos(c)] ]).  % <--- negated Query
problem(3,'Princess & Tiger',
      [[pos(t1),pos(z2),neg(t1)],
       [pos(t1),neg(z2)],
       [neg(t1),neg(z2)],
       [pos(t1),pos(z1),neg(t2)],
       [pos(t2),neg(z1)],
       [neg(t1),neg(z1)],
       [pos(z1),pos(z2)],
       [neg(z1),neg(z2)]]).
```

[35]:
```
?- test(2).
```

```
Solving problem 2 : Knights & Knaves (Proof by contradiction)
UNSAT: no model exists
```

**true**

[36]:
```
?- test(3).
```

```
Solving problem 3 : Princess & Tiger
SAT: Model found: [branch(neg(t1)),unit(neg(z2)),unit(pos(z1)),unit(pos(t2))]
```

**true**

[37]:
```
problem(4,'uf-20-02',
  [[ neg(x10) , neg(x16) , pos(x5) ] ,
   [ pos(x16) , neg(x6) , pos(x5) ] ,
   [ neg(x17) , neg(x14) , neg(x18) ] ,
   [ neg(x10) , neg(x15) , pos(x19) ] ,
   [ neg(x1) , neg(x9) , neg(x18) ] ,
   [ pos(x3) , pos(x7) , neg(x6) ] ,
   [ neg(x13) , pos(x1) , pos(x6) ] ,
   [ neg(x2) , neg(x16) , neg(x20) ] ,
   [ pos(x7) , pos(x8) , pos(x18) ] ,
   [ neg(x7) , pos(x10) , neg(x20) ] ,
   [ pos(x2) , neg(x14) , neg(x17) ] ,
   [ pos(x2) , pos(x1) , pos(x19) ] ,
   [ pos(x7) , neg(x20) , neg(x1) ] ,
   [ neg(x11) , pos(x1) , neg(x17) ] ,
   [ pos(x3) , neg(x12) , pos(x19) ] ,
   [ neg(x3) , neg(x13) , pos(x6) ] ,
   [ neg(x13) , pos(x3) , neg(x12) ] ,
   [ pos(x5) , neg(x7) , neg(x12) ] ,
   [ pos(x20) , pos(x8) , neg(x16) ] ,
   [ neg(x13) , neg(x6) , pos(x19) ] ,
   [ neg(x5) , pos(x1) , pos(x14) ] ,
   [ pos(x9) , neg(x5) , pos(x18) ] ,
   [ neg(x12) , neg(x17) , neg(x1) ] ,
   [ neg(x20) , neg(x16) , pos(x19) ] ,
   [ pos(x12) , pos(x10) , neg(x11) ] ,
   [ pos(x6) , neg(x7) , neg(x2) ] ,
   [ pos(x13) , neg(x10) , pos(x17) ] ,
   [ neg(x20) , pos(x8) , neg(x16) ] ,
   [ neg(x10) , neg(x1) , neg(x8) ] ,
   [ neg(x7) , neg(x3) , pos(x19) ] ,
   [ pos(x19) , neg(x1) , neg(x6) ] ,
   [ pos(x19) , neg(x2) , pos(x13) ] ,
   [ neg(x2) , pos(x20) , neg(x9) ] ,
   [ neg(x8) , neg(x20) , pos(x16) ] ,
   [ neg(x13) , neg(x1) , pos(x11) ] ,
   [ pos(x15) , neg(x12) , neg(x6) ] ,
   [ neg(x17) , neg(x19) , pos(x9) ] ,
   [ pos(x19) , neg(x18) , pos(x16) ] ,
   [ pos(x7) , neg(x8) , neg(x19) ] ,
   [ neg(x3) , neg(x7) , neg(x1) ] ,
   [ pos(x7) , neg(x17) , neg(x16) ] ,
   [ neg(x2) , neg(x14) , pos(x1) ] ,
   [ neg(x18) , neg(x10) , neg(x8) ] ,
   [ neg(x16) , pos(x5) , pos(x8) ] ,
   [ pos(x4) , pos(x8) , pos(x10) ] ,
   [ neg(x20) , neg(x11) , neg(x19) ] ,
   [ pos(x8) , neg(x16) , neg(x6) ] ,
   [ pos(x18) , pos(x12) , pos(x8) ] ,
   [ neg(x5) , neg(x20) , neg(x10) ] ,
   [ pos(x16) , pos(x17) , pos(x3) ] ,
   [ pos(x7) , neg(x1) , neg(x17) ] ,
   [ pos(x17) , neg(x4) , pos(x7) ] ,
   [ pos(x20) , neg(x9) , neg(x13) ] ,
   [ pos(x13) , pos(x18) , pos(x16) ] ,
   [ neg(x16) , neg(x6) , pos(x5) ] ,
   [ pos(x5) , pos(x17) , pos(x7) ] ,
   [ neg(x12) , neg(x17) , neg(x6) ] ,
   [ neg(x20) , pos(x19) , neg(x5) ] ,
   [ pos(x9) , neg(x19) , pos(x16) ] ,
   [ neg(x13) , neg(x16) , pos(x11) ] ,
   [ neg(x4) , neg(x19) , neg(x18) ] ,
   [ neg(x13) , pos(x10) , neg(x15) ] ,
   [ pos(x16) , neg(x7) , neg(x14) ] ,
   [ neg(x19) , neg(x7) , neg(x18) ] ,
   [ neg(x20) , pos(x5) , pos(x13) ] ,
   [ pos(x12) , neg(x6) , pos(x4) ] ,
   [ pos(x7) , pos(x9) , neg(x13) ] ,
   [ pos(x16) , pos(x3) , pos(x7) ] ,
   [ pos(x9) , neg(x1) , pos(x12) ] ,
   [ neg(x3) , pos(x14) , pos(x7) ] ,
   [ pos(x1) , pos(x15) , pos(x14) ] ,
   [ neg(x8) , neg(x11) , pos(x18) ] ,
   [ pos(x19) , neg(x9) , pos(x7) ] ,
   [ neg(x10) , pos(x6) , pos(x2) ] ,
   [ pos(x14) , pos(x18) , neg(x11) ] ,
   [ neg(x9) , neg(x16) , pos(x14) ] ,
   [ pos(x1) , pos(x11) , neg(x20) ] ,
   [ pos(x11) , pos(x12) , neg(x4) ] ,
   [ pos(x13) , neg(x11) , neg(x14) ] ,
   [ pos(x17) , neg(x12) , pos(x9) ] ,
   [ pos(x14) , pos(x9) , pos(x1) ] ,
   [ pos(x8) , pos(x19) , pos(x4) ] ,
   [ pos(x6) , neg(x13) , neg(x20) ] ,
   [ neg(x2) , neg(x13) , pos(x11) ] ,
   [ pos(x14) , neg(x13) , pos(x17) ] ,
   [ pos(x9) , neg(x11) , pos(x18) ] ,
   [ neg(x13) , neg(x6) , pos(x5) ] ,
   [ pos(x5) , pos(x19) , neg(x18) ] ,
   [ neg(x4) , pos(x10) , pos(x11) ] ,
   [ neg(x18) , neg(x19) , neg(x20) ] ,
   [ pos(x3) , neg(x9) , pos(x8) ]
   ]).
problem(5,'schaltung_a1.pl',
 [
%a. not_b. c. not_d. e.
   [pos(a)], [neg(b)], [pos(c)], [neg(d)], [pos(e)],
% Schaltungen der ersten Ebene
%and11 :- a,b.
    [pos(and11),neg(a),neg(b)],
%or11 :- b.
```

```
     [pos(or11),neg(b)],
%or11 :- c.
     [pos(or11),neg(b)],
%and12 :- c,d.
     [pos(and12),neg(c),neg(d)],
%not1 :- not_e.
     [pos(not1),pos(e)],
% Schaltungen der zweiten Ebene
%or21 :- and11.
     [pos(or21),neg(and11)],
%or21 :- not1.
     [pos(or21),neg(not1)],
%and2 :- or11, not1.
     [pos(and2),neg(or11),neg(not1)],
%or22 :- and12.
     [pos(or22),neg(and12)],
%or22 :- not1.
     [pos(or22),neg(not1)],
%not2 :- e.  % \+ not1.
     [pos(not2),neg(e)],
% Schaltungen der dritten Ebene
%and3 :- or21, and2.
     [pos(and3),neg(or21),neg(and2)],
%or3 :- or22.
     [pos(or3),neg(or22)],
%or3 :- not2.
     [pos(or3),neg(not2)],
% Schaltungen der vierten Ebene
%or4 :- and3.
     [pos(or4),neg(and3)],
%or4 :- or3.
     [pos(or4),neg(or3)],
%and4 :- or3, not2.
     [pos(and4),neg(or3),neg(not2)],
% Letzte Ebene
%and5 :- and4, or4.
     [pos(and5),neg(and4),neg(or4)],
%output :- and5.
     [pos(output),neg(and5)],
% ?- output.
     [neg(output)]
   ]).

problem(6, 'doodle-wo-max1-constraint',
 [
   [pos(x1),pos(x3)],
   [neg(x3)],
   [neg(x5)],
   [pos(x4),pos(x5)]
  ]).
problem(7, 'doodle-with-max1-constraint',
 [
   [pos(x1),pos(x3)],
   [neg(x3)],
   [neg(x5)],
   [pos(x4),pos(x5)],

   [neg(x1),neg(x2)], [neg(x1),neg(x3)], [neg(x1),neg(x4)],
   [neg(x1),neg(x5)],
   [neg(x2),neg(x3)], [neg(x2),neg(x4)],[neg(x2),neg(x5)],
   [neg(x3),neg(x4)],[neg(x3),neg(x5)],
   [neg(x4),neg(x5)]
  ]).
```

[38]:
```
?- test(4).
```

```
Solving problem 4 : uf-20-02
SAT: Model found:␣
 ↪[branch(neg(x10)),branch(pos(x16)),branch(neg(x17)),branch(neg(x1)),branch(pos(x3)),branch(n
```

```
true
```

[39]:
```
?-test(5).
```

```
Solving problem 5 : schaltung_a1.pl
UNSAT: no model exists
```

```
true
```

Below we show another version of the DPLL algorithm which returns the full decision tree. It is for education purposes, e.g., to visualise the choices made by the algorithm:

[40]:
```
dpll_tree(Clauses,unit(Lit,Tree),Res) :-
    select([Lit],Clauses,Clauses2), % unit clause found
    !,
    set_literal(Lit,Clauses2,Clauses3),
    dpll_tree(Clauses3,Tree,Res).
dpll_tree([],Tree,Res) :- !, Tree=[], Res=sat. % SAT
dpll_tree(Clauses,Tree,Res) :- member([],Clauses),!, Tree=unsat,Res=unsat.
dpll_tree(Clauses,branch(Lit,TreePos,TreeNeg),Res) :-
    choose_literal(Clauses,Lit),
    !,
    set_literal(Lit,Clauses,Clauses1),
    dpll_tree(Clauses1,TreePos,Res1),
    (Res1=sat
      -> TreeNeg=unexplored, Res=sat
      ; negate(Lit,NegLit),
        set_literal(NegLit,Clauses,Clauses2),
        dpll_tree(Clauses2,TreeNeg,Res)
    ).
```

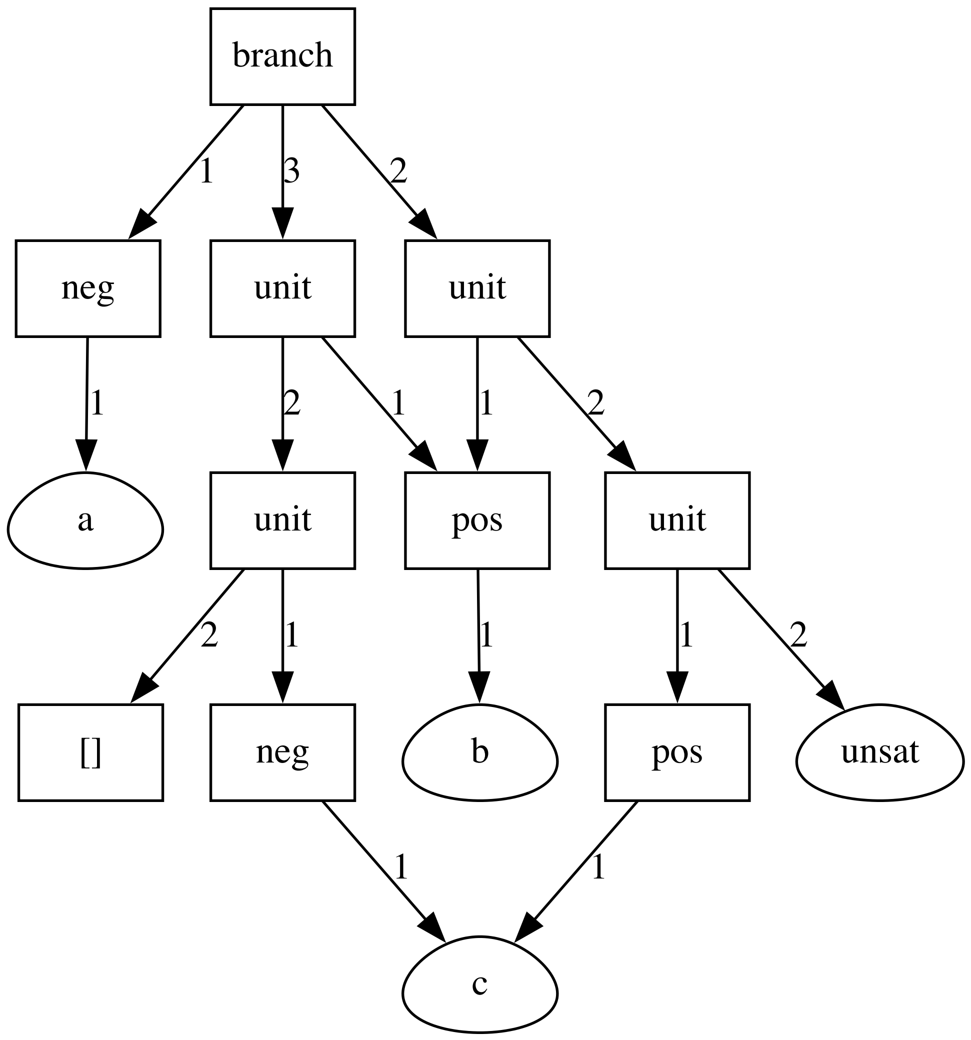
[41]:
```
?- problem(1,Str,Clauses), dpll_tree(Clauses,Tree,Res).
```

```
Str = Knights & Knaves,

Clauses =␣
 ↪[[neg(a),neg(b),neg(c)],[pos(b),pos(a)],[pos(c),pos(a)],[neg(b),pos(a)],[neg(a),pos(b)]],

Tree =␣
 ↪branch(neg(a),unit(pos(b),unit(pos(c),unsat)),unit(pos(b),unit(neg(c),[]))),

Res = sat
```

[42]:
```
?- print($Tree),nl.
```

```
branch(neg(a),unit(pos(b),unit(pos(c),unsat)),unit(pos(b),unit(neg(c),[])))

Tree =␣
 ↪branch(neg(a),unit(pos(b),unit(pos(c),unsat)),unit(pos(b),unit(neg(c),[])))
```

[43]:
```
show_term($Tree)
```


branch
1
3
2
neg
unit
unit
1
2
1
1
2
a
unit
pos
unit
2
1
1
1
2
[]
neg
b
pos
unsat
1
1
c

```
true
```

[ ]: