Knowledge Representation and Reasoning Project 1

Olteanu Fabian Cristian

FMI, AI Master, Year 1

---

1. Resolution

Let us consider the following knowledge base:

KB $\begin{cases} \end{cases}$

1. Every DOTA2 player is a gamer.
2. There are DOTA2 players who are professional.
3. Some professional DOTA2 players who purchase Divine Rapier lose games.
4. Anyone who loses games gets angry.

We want to prove that the following Question is logically entailed from our KB by applying the Resolution algorithm:

5. Do some gamers who purchase Divine Rapier get angry?

1.1. Representing the KB in FOL (a)

The above written KB can be expressed in FOL in the following way:

KB

1. $\forall x.DOTA2\_Player(x) \supset Gamer(x)$
2. $\exists x.DOTA2\_Player(x) \wedge Pro(x)$
3. $\exists x.PRO(x) \wedge Buys\_Rapier(x) \wedge Loses\_Games(x)$
4. $\forall x.Loses\_Games(x) \supset Gets\_Angry(x)$
5. $\exists x.Gamer(x) \wedge Buys\_Rapier(x) \wedge Gets\_Angry(x)$.

For the sake of simplicity, we can write it in the following way:

KB

1. $\forall x.P_1(x) \supset P_2(x)$
2. $\exists x.P_1(x) \wedge P_3(x)$
3. $\exists x.P_3(x) \wedge P_4(x) \wedge P_5(x)$
4. $\forall x.P_5(x) \supset P_6(x)$
5. $\exists x.P_2(x) \wedge P_4(x) \wedge P_6(x)$.

1.2. Proving manually that the Question is logically entailed from the KB (b)

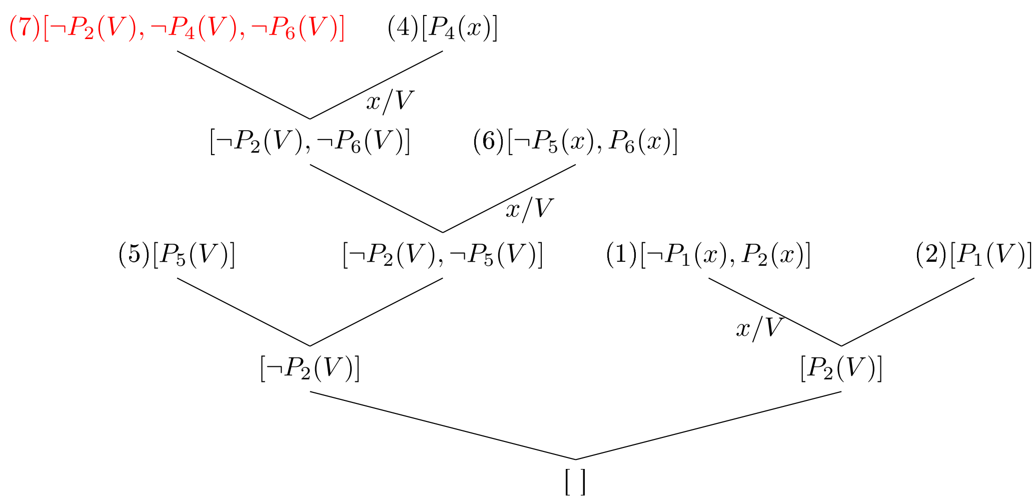
In order to apply the resolution algorithm, we must first transform the KB written in FOL in the conjunctive normal form (CNF), after which the following transformation happens:

$$\text{CNF}\begin{cases} 1.\ \neg P_1(x) \vee P_2(x) \\ 2.\ P_1(V) \\ 3.\ P_3(V) \\ 4.\ P_4(V) \\ 5.\ P_5(V) \\ 6.\ \neg P_5(x) \vee P_6(x) \\ 7.\ P_2(V) \wedge P_4(V) \wedge P_6(V), \end{cases}$$

where V is a Skolem constant (we now have seven members in the list because we can "divide" into individual items logical sequences like number 2 from the last page: $P_1(x) \wedge P_3(x)$ becomes 2. $P_1(x)$ and 3. $P_3(x)$).

Lastly, we need to prove that the negated form of the question that we want to prove is logically entailed from our KB is not satisfiable. After negating, our converted KB looks like this:

$$\text{CNF}\begin{cases} 1.\ \neg P_1(x) \vee P_2(x) \\ 2.\ P_1(V) \\ 3.\ P_3(V) \\ 4.\ P_4(V) \\ 5.\ P_5(V) \\ 6.\ \neg P_5(x) \vee P_6(x) \\ 7.\ \neg P_2(V) \vee \neg P_4(V) \vee \neg P_6(V). \end{cases}$$



Thus, we have proven that the negated form of sentence 5 is unsatisfiable given our KB. In conclusion, the sentence: "Some gamers who purchase Divine Rapier get angry" is logically entailed from our KB.

1.3. Implementing the Resolution Algorithm in Prolog (c)

The Resolution algorithm can be expressed in the following way, using pseudocode [1]:

Algorithm 1 Res(S)

Input: S, a finite set of propositional clauses (of form [[w, s, n(p)], [a, n(w), r, t],[[q]])
Output: S is satisfiable or unsatisfiable
  if $[] \in S$ **then**
    return unsat
  else if There are two clauses in S such that they resolve to produce another clause not already in S **then**
    Add the new resolvent clause to S and remove the clauses used to obtain it
    Res(S)
  else
    return sat
  end if

To implement this algorithm in Prolog, based on the pseudocode presented above, I followed this way of thinking:

- The implemented algorithm returns false if the given set of clauses (KB) is satisfiable and true if it is unsatisfiable,
- For the resolution procedure I implemented a predicated called res/1 which takes KB as a parameter. The stop condition for the predicate is finding an empty list as a member.
- If there's no empty list in the KB, it does the following:
  1. The KB is sorted using the built-in predicate sort/2[2] to remove tautologies,
  2. It takes a member clause from KB (Clause1) using the built-in predicate member/2[3],
  3. It takes a different member clause (Clause2) from KB,
  4. It selects a Subclause from Clause1, using the built-in predicate select/3[4],
  5. It checks if the negated Subclause is a member of Clause2; if it is, it selects the clauses from which the two subclauses originate and appends them to a Resolvent (using a built-in predicate called append/3[5]),
  6. The Resolvent is sorted (for the same effect as in step 1),
  7. The two clauses that generated the Resolvent are delete from the KB using a built in predicate delete/3[6],
  8. If the Resolvent is empty, return true (unsatisfiable),
  9. Otherwise, append the Resolvent to the new KB and continue recursively with the new KB.

For the sake of reading input from files, I also implemented a predicate called read_clauses_from_file/2, which applies the res predicate to each line of input and outputs whether the clause set is satisfiable or unsatisfiable in the terminal.

To run the Prolog script on the set of clauses presented in the first page, it needs to be rewritten in the following way:

$$KB = [[not(a), b], [a], [c], [d], [e], [not(e), f], [not(b), not(d), not(f)]].$$

The letters a through f are direct mappings of the clauses P_1 through P_6 from page 2. Querying

$$KB = [[not(a), b], [a], [c], [d], [e], [not(e), f], [not(b), not(d), not(f)]], res(KB).$$

yields the output "unsat", as expected.

1.4. Running the Resolution Algorithm on Other Sets of Propositional Clauses (d)

Implementing the read_clauses_from_file/2 predicate and placing it inside of a main/0 predicate, along with creating a data.in file allows the Prolog script to swiftly compute the output for each of the following sets of clauses:

1. $[[not(a), b], [c, d], [not(d), b], [not(b)], [not(c), b], [e], [a, b, not(f), f]]$: unsat
2. $[[not(b), a], [not(a), b, e], [a, not(e)], [not(a)], [e]]$: unsat
3. $[[not(a), b], [c, f], [not(c)], [not(f), b], [not(c), b]]$: sat
4. $[[a, b], [not(a), not(b)], [c]]$: sat.

2. Implementing the Davis-Putnam Procedure in Prolog

Like the Resolution algorithm, the Davis-Putnam procedure also checks whether a finite set of clauses is satisfiable, but is generally faster. It also returns a list of truth values assigned to clauses from the Knowledge Base when it is satisfiable.

The algorithm, presented in pseudocode, is the following[1]:

Algorithm 2 DP(KB)

Input: KB, a finite set of propositional clauses (of form [[w, s, n(p)], [a, n(w), r, t],[[q]])
Output: are the clauses from KB satisfiable or unsatisfiable, YES or NO?

```
if KB is empty then
    return YES
end if
if KB contains [] then
    return NO
end if
p ← some atom from KB
if DP(KB · p)=YES then
    return YES
else
    return DP(C · ¬p)
end if
```

The mathematical definition for the dot operation can be expressed as such:

$$C \cdot m = \{c | c \in C, m \notin c\} \cup \{(c - \neg m) | c \in C, m \notin c, \neg m \in c\}.$$

The operation can be computed using the following algorithm: For each L from C:

1. if L contains m, remove L from C,
2. if L contains $\neg m$, remove $\neg m$ from L.

Another challenge in implementing the algorithm is finding optimal strategies of choosing p. Two of those that were used were

- p appears in the shortest clause(s) in KB,
- p is the most balanced atom.

Taking all of these facts into account, the outline of the implementation of the algorithm was the following:

1. A custom predicate called kb_dot_p/3 was implemented, which performs the dot operation. It is based on the two steps from above and uses two custom predicates called negate/2 and delete_parameter/3 to perform the second step. The first step is achieved through the built-in findall/3[7] predicate which filters KB to exclude all clauses that include p. For the second step, the predicate delete_parameter/3 uses maplist/3[8] to filter the whole KB using the custom delete_occurences_of_p/3 predicate (which filters only one clause from KB to remove ocurences of p).

2. For the selection of p, two predicates were created corresponding to the two different strategies used. The first one is called select_shortest_p/2 (finds the list of minimum length, selects a member from that list and assigns it to P). The second one is select_most_balanced_p

3. There are two versions of the Davis-Putnam procedures implemented as two different predicates. The first one is dp_shortest_clauses_p/2 and uses the select_shortest_p/2 predicate. Its implementation is very close to the pseudocode, the only addition being the recursive addition of the truth values for the literals in a list. The second predicate is called dp_most_balanced_p/2 and apart from using the other selection predicate is exactly the same.

4. Similar to the Resolution algorithm implementation, the prolog script in which the Davis-Putnam procedure was implemented also reads input from a file, but two different predicates had to be implemented evaluate the two custom predicates on the clause sets

   (read_clauses_from_file1/2 and read_clauses_from_file2/2).

References

[1] Ronald Brachman, Hector Levesque, "Knowledge Representation and Reasoning," *Morgan Kaufmann*, p. 54-78, 2004

[2] `https://www.swi-prolog.org/pldoc/man?predicate=sort/2`

[3] `https://www.swi-prolog.org/pldoc/doc_for?object=member/2`

[4] `https://www.swi-prolog.org/pldoc/doc_for?object=select/3`

[5] `https://www.swi-prolog.org/pldoc/doc_for?object=append/3`

[6] `https://www.swi-prolog.org/pldoc/doc_for?object=delete/3`

[7] `https://www.swi-prolog.org/pldoc/man?predicate=findall/3`

[8] `https://www.swi-prolog.org/pldoc/doc_for?object=maplist/3`

Listing 1: Resolution Implementation in Prolog

```
%choose two clauses from KB, apply Resolution, add the Resolvent (if new to KB) =>
%new KB..., res(newKB)

%1. RES(KB) :- member(X, KB), member(Y, KB)
%2. select 2 lists from the KB (the second one should contain a negated subclause from
%(the two lists (clauses) need to be two different elements)
%3. check whether they have the connective element that allows us to apply resolution
%(like "not x or x")
%4. add the new element to the KB and delete the clauses that were used to obtain it
%(if it's an empty list ommit it)
%5. continue recursively

%returns false if satisfiable
%returns true if unsatisfiable

% my example KB = [[not(a), b], [a], [c], [d], [e], [not(e), f], [not(b), not(d), not

%I. KB = [[not(a), b], [c,d], [not(d), b], [not(b)], [not(c), b], [e], [a, b, not(f),
%II. KB = [[not(b),a], [not(a),b,e], [a, not(e)], [not(a)], [e]]
%III. KB = [[not(a),b], [c,f], [not(c)], [not(f),b], [not(c),b]]
%IV. KB = [[a,b], [not(a), not(b)], [c]]

res(KB) :- member([], KB), !.
res(KB) :-
    sort(KB, KB_Sorted),                               %remove repeating clauses
    member(Clause1, KB_Sorted),
    delete(KB_Sorted, Clause1, KB_Without_Clause1),    %delete the first member into
    member(Clause2, KB_Without_Clause1),               %different from the first memb
    select(Subclause, Clause1, Prop1),
    (
        memberchk(not(Subclause), Clause2), true ->
            select(not(Subclause), Clause2, Prop2)     %if the connective element exi
        ;
            false                                      %otherwise return false (satis
    ),
    append(Prop1, Prop2, Resolvent0),                  %Resolvent element
    sort(Resolvent0, Resolvent),
    delete(KB_Sorted, Clause1, KB_New1),
    delete(KB_New1, Clause2, KB_New2),                 %delete the elements used to o
    (
        not(member(_, Resolvent)), true ->             %if the resolvent is empty, re
            true, !
        ;
            append(KB_New2, [Resolvent], KB_New3),     %otherwise append the resolven
            res(KB_New3), !
    ).

read_clauses_from_file(Str, []) :-
    at_end_of_stream(Str), !.

read_clauses_from_file(Str, [_|T]) :-
    not(at_end_of_stream(Str)),
    read(Str, X),
    res(X) ->
        write(unsat), nl,
```

```
        read_clauses_from_file(Str, T)
    ;
        write(sat), nl,
    read_clauses_from_file(Str, T).


main :-
    open('/Users/chocogo/Desktop/Master/Projects/KRR_Project_1/data.in', read, Str),
    read_clauses_from_file(Str, _),
    close(Str).
```