Resolution and SAT Solvers

Vlad-Mihai Olaeriu

December 2024

1 Resolution

Task: create your knowledge base and a question, that is logically entailed from the knowledge base. The sentences represented in FOL must contain variables. The recommended size of the knowledge base is 4-6 sentences.

1. Anybody who plays as a midfielder is smart and agile.
2. Anybody who scores goals is skillful or lucky.
3. Every player who is agile and skillful can play as a forward.
4. Every player who is strong and smart can play as a defender.
5. Every player with a big mouth (towards the referee) is strong and not lucky.
6. Anybody who can play as a defender, midfielder, and forward is an all-round player.

Besides these general sentences, I have also added a few facts about some players from the Manchester United team, facts that I will use in the interface for the user to play with. The most important facts are related to Bruno Fernandes:

7. Bruno Fernandes plays as a midfielder.
8. Bruno Fernandes scores goals.
9. Bruno Fernandes has a big mouth.

Question: is Bruno Fernandes an all-rounded player?

1.1 Sentences to FOL

1. $\forall X.Midfielder(X) \rightarrow Smart(X) \land Agile(X)$
2. $\forall X.ScoreGoals(X) \rightarrow Skillful(X) \lor Lucky(X)$
3. $\forall X.(Agile(X) \land Skillful(X)) \rightarrow Forward(X)$
4. $\forall X.(Strong(X) \land Smart(X)) \rightarrow Defender(X)$
5. $\forall X.BigMouth(X) \rightarrow \neg Lucky(X) \land Strong(X)$
6. $\forall X.(Defender(X) \land Midfielder(X) \land Forward(X)) \rightarrow AllRoundPlayer(X)$

1.2 FOL to CNF

My knowledge base would be represented by a big conjunction ($\wedge$) between all sentences, from 1 to 6, that are described below.

1. $\forall X.(\neg Midfielder(X) \vee Smart(X)) \wedge (\neg Midfielder(X) \vee Agile(X))$
   (a) $\forall X.\neg Midfielder(X) \vee Smart(X)$
   (b) $\forall X.\neg Midfielder(X) \vee Agile(X)$
2. $\forall X.\neg ScoreGoals(X) \vee Skillful(X) \vee Lucky(X)$
3. $\forall X.\neg Agile(X) \vee \neg Skillful(X) \vee Forward(X)$
4. $\forall X.\neg Strong(X) \vee \neg Smart(X) \vee Defender(X)$
5. $\forall X.(\neg BigMouth(X) \vee \neg Lucky(X)) \wedge (\neg BigMouth(X) \vee Strong(X))$
   (a) $\forall X.\neg BigMouth(X) \vee \neg Lucky(X)$
   (b) $\forall X.\neg BigMouth(X) \vee Strong(X)$
6. $\forall X.\neg Defender(X) \vee \neg Midfielder(X) \vee \neg Forward(X) \vee AllRoundPlayer(X)$

Now, adding the facts about Bruno Fernandes:

7. $Midfielder(BrunoFernandes)$
8. $ScoreGoals(BrunoFernandes)$
9. $BigMouth(BrunoFernandes)$

And, to prove that Bruno Fernandes is an all-round player, we need to add the negated question to the knowledge base as well:

10. $\neg AllRoundPlayer(BrunoFernandes)$

1.3 Proof of the question

In the following subsection, I will prove that the question is logically entailed from the knowledge base. For this, I will manually apply resolution, and then the resolvent of the two clauses I chose at the current step will be added to my knowledge base.

Resolve clause (1a) and clause (7):
$\neg Midfielder(X) \vee Smart(X)$ and $Midfielder(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
11. $Smart(BrunoFernandes)$

Resolve clause (1b) and clause (7):
$\neg Midfielder(X) \vee Agile(X)$ and $Midfielder(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
12. $Agile(BrunoFernandes)$

Resolve clause (2) and clause (8):
$\neg ScoreGoals(X) \vee Skillful(X) \vee Lucky(X)$ and $ScoreGoals(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
13. $Skillful(BrunoFernandes) \vee Lucky(BrunoFernandes)$

Resolve clause (5a) and clause (9):
$\neg BigMouth(X) \vee \neg Lucky(X)$ and $BigMouth(BrunoFernandes)$
Substitute: $X = BrunoFernandes$

14. $\neg Lucky(BrunoFernandes)$

Resolve clause (13) and clause (14):
$Skillful(BrunoFernandes) \vee Lucky(BrunoFernandes)$ and $\neg Lucky(BrunoFernandes)$
15. $Skillful(BrunoFernandes)$

Resolve clause (5b) and clause (9):
$\neg BigMouth(X) \vee Strong(X)$ and $BigMouth(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
16. $Strong(BrunoFernandes)$

Resolve clause (3) and clause (12):
$\neg Agile(X) \vee \neg Skillful(X) \vee Forward(X)$ and $Agile(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
17. $\neg Skillful(BrunoFernandes) \vee Forward(BrunoFernandes)$

Resolve clause (17) and clause (15):
$\neg Skillful(BrunoFernandes) \vee Forward(BrunoFernandes)$ and $Skillful(BrunoFernandes)$
18. Forward(BrunoFernandes)

Resolve clause (4) and clause (11):
$\neg Strong(X) \vee \neg Smart(X) \vee Defender(X)$ and $Smart(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
19. $\neg Strong(BrunoFernandes) \vee Defender(BrunoFernandes)$

Resolve clause (16) and clause (19):
$Strong(BrunoFernandes)$ and $\neg Strong(BrunoFernandes) \vee Defender(BrunoFernandes)$
20. Defender(BrunoFernandes)

Resolve clause (6) and clause (7):
$\neg Defender(X) \vee \neg Midfielder(X) \vee \neg Forward(X) \vee AllRoundPlayer(X)$ and
$Midfielder(BrunoFernandes)$
Substitute: $X = BrunoFernandes$
21. $\neg Defender(BrunoFernandes) \vee \neg Forward(BrunoFernandes) \vee AllRoundPlayer(BrunoFernandes)$

Resolve clause (18) and clause (21):
$\neg Defender(BrunoFernandes) \vee \neg Forward(BrunoFernandes) \vee AllRoundPlayer(BrunoFernandes)$ and
$Forward(BrunoFernandes)$
22. $\neg Defender(BrunoFernandes) \vee AllRoundPlayer(BrunoFernandes)$

Resolve clause (20) and clause (22):
$\neg Defender(BrunoFernandes) \vee AllRoundPlayer(BrunoFernandes)$ and $Defender(BrunoFernandes)$
23. AllRoundPlayer(BrunoFernandes)

Resolve clause (10) and clause (23):
$AllRoundPlayer(BrunoFernandes)$ and $\neg AllRoundPlayer(BrunoFernandes)$
24. [] empty clause

After deriving the 24th clause, it can be stated that the knowledge base with the negated question is unsatisfiable; thus, the question is entailed from the knowledge base.

1.4 Implementation

I have chosen to use Prolog for the implementation of the resolution algorithm. First, I needed to find a way to represent my knowledge base as a data structure, so I chose a list of lists. Let's consider that I have the following knowledge base:

$$(a \vee b \vee \neg c) \wedge (\neg d \vee e)$$

which contains two clauses. This was represented in Prolog as:

$$[[a, b, n(c)], [d, e]],$$
$$\text{where } n(c) = \neg c.$$

The inner lists represented my clauses in CNF form, and the *wrapper list* represents the conjunction between them.

The first step in the algorithm is to search for two clauses that can create a resolvent together; more precisely, to find two clauses, one that contains predicate p and the other one contains $n(p)$. With the two clauses in hand, merge the two clauses $C_1 \cup p$ and $C_2 \cup n(p)$ into the resolvent $C_1 \cup C_2 \setminus \{p, n(p)\}$, then add the resolvent to the knowledge base. With the updated knowledge base, repeat this process until no resolvent can be created anymore.

To be optimal, I have added two extra requirements:

- *step 1:* verify if the two clauses were used previously to create a resolvent
- *step 2:* verify if the resolvent is already part of the knowledge base

With these extra verifications, we ensure that we do not use the same clauses multiple times, respectively we do not add an existing resolvent to the knowledge base (which would be redundant).

Besides these verifications, to optimize the search for the shortest derivation path in resolution, the *removal of tautologies* was added as a pre-processing step. Before starting to search for clauses that could resolve together, the knowledge base is *cleaned of tautologies*.

NOTE: For the resolution with variables, another extra step of pre-processing was added: the replacing of variables. For humans, it is easier to annotate each variable with X when we want to express clauses such as $\forall X.p(X)$. If this is done multiple times, for different clauses, Prolog would interpret that the same variable is used across multiple clauses. To make use of the Prolog's built-in substitution, the variables needed to be replaced for each clause:

$$\text{from } [[p(X), p(Y)], [q(X), q(Y)]]$$
$$\text{to } [[p(X_1, Y_1)], [q(X_2), q(Y_2)]]$$

because resolving a clause that contains variable X from $p(X)$ would also influence the value of $q(X)$, and the structure of the knowledge base would be compromised.

For the clauses written in CNF in exercise *1d*, I obtained the following results:

1. $(\neg a \vee b) \wedge (c \vee d) \wedge (\neg d \vee b) \wedge (\neg b) \wedge (\neg c \vee b) \wedge (e) \wedge (f \vee a \vee b \vee \neg f)$: unsatisfiable
2. $(\neg b \vee a) \wedge (\neg a \vee b \vee e) \wedge (a \vee \neg e) \wedge (\neg a) \wedge (e)$: unsatisfiable
3. $(\neg a \vee b) \wedge (c \vee f) \wedge (\neg c) \wedge (\neg f \vee b) \wedge (\neg c \vee b)$: satisfiable
4. $(a \vee b) \wedge (\neg a \vee \neg b) \wedge (c)$: satisfiable

2 SAT Solver

Task: implement the Davis-Putnam SAT procedure. For S, a set of clauses in written in CNF, the procedure will display YES, respectively NOT, as S is satisfiable or not. In the case of YES, the procedure will also display the truth values assigned to the literals (e.g. w/true; s/false; p/false ...). Choose two strategies (but not in pairs like least/most) of selection of the atom to perform the • operation and discuss/compare the results.

```
Algorithm: SAT Solver Davis-Putnam
Input: KB = list of clauses
Output: satisfability of the clauses

davis_putnam(KB):
1. if (C is empty) return "YES"
2. if ([] in C) return "NO"
3. Choose atom p from a clause in C
4. if (davis_putnam(KB • p) is "YES") return "YES"
5. else return davis_putnam(KB • n(p))
```

The knowledge base was represented in the same way as it was for the resolution algorithm, a list of lists (a list of disjunctions).

Regarding this algorithm, an interesting choice that needs to be made is the way in which you choose the atom from the knowledge base. I used the following strategies for choosing the atom p:

1. atom p that appears in most clauses;
2. atom p from the shortest clause.

For the examples provided in the task, I have obtained the following results:

Examples	**Strategy 1**	**Strategy 2**
1	125ms	0ms
2	266ms	0ms
3	234ms	0ms
4	0ms	0ms
5	63ms	0ms
6	0ms	0ms

Table 1: Runtime statistics with different strategies

Strategy 1:

1. **YES:** girl/true, child/true, female/true, boy/true, toddler/true
2. **NO**
3. **NO**
4. **NO**
5. **YES:** f/true, b/false, e/false
6. **NO**

Strategy 2:

1. **YES:** girl/true, female/true, toddler/true, child/true, male/false

2. **NO**
3. **NO**
4. **NO**
5. **YES:** e/true, f/true, a/false
6. **NO**

3 Code

utils.pl

```
neg(n(X), X) :- !.
neg(X, n(X)).

is_member(_, []) :- !, false.
is_member(X, [H|_]) :-
    permutation(X, H), !.
is_member(X, [_|T]) :-
    is_member(X, T).
is_not_member(X, List) :- \+ is_member(X, List).

eliminate(Target, [Target|T], T).
eliminate(Target, [H|T], [H|Result]) :- eliminate(Target, T, Result).

merge([], L, L).
merge([H|T], L, Result) :- member(H, L), !, merge(T, L, Result).
merge([H|T], L, [H|Result]) :- merge(T, L, Result).

unpack_kb([KB], KB).
```

read.pl

```
read_file(Filename, Lines) :-
    open(Filename, read, Stream),
    read_lines(Stream, Lines),
    close(Stream).

line_to_atom(Line, Term) :-
    atom_string(Atom, Line),
    atom_to_term(Atom, Term, _).

read_lines(Stream, []) :-
    at_end_of_stream(Stream).
read_lines(Stream, [Atom|Rest]) :-
    \+ at_end_of_stream(Stream),
    read_line_to_string(Stream, Line),
    line_to_atom(Line, Atom),
    read_lines(Stream, Rest).
```

parse.pl

```
process_sentence(Sentece, Result) :- translate(Result, Sentece, []).

process_sentences([], []).
process_sentences([Sentece|Sentences], [Result|Results]) :-
    process_sentence(Sentece, Result),
    process_sentences(Sentences, Results).
```

```
translate([A|B]) --> disjunction(A), [and], translate(B).
translate([A])   --> disjunction(A).

disjunction(A) --> ['('], list_of_options(A).

list_of_options([Element])   --> [Element, ')'].
list_of_options([Element|T]) --> [Element], [or], list_of_options(T).
```

resolution-variables.pl

```
:- ["./utils/read.pl", "./utils/parse.pl", "./utils/utils.pl"].
get_variables_from_clause(Clause, Variables) :-
    maplist(term_variables, Clause, VariablesList),
    flatten(VariablesList, FlatVariables),
 sort(FlatVariables, Variables).

replace_vars_from_clause([], []).
replace_vars_from_clause([Clause|KB], [NewClause|NewKB]) :-
    get_variables_from_clause(Clause, Variables),
    copy_term(Variables, Clause, _, NewClause),
    replace_vars_from_clause(KB, NewKB).

make_resolvent(Clause1, Clause2, Literal, Resolvent) :-
    eliminate(Literal, Clause1, Clause1New),
    neg(Literal, LiteralNeg),
    eliminate(LiteralNeg, Clause2, Clause2New),
    merge(Clause1New, Clause2New, Resolvent).

do_resolve([], _, []).
do_resolve([H1|T1], Clause, Literal) :-
    neg(H1, H2), member(H2, Clause), Literal = H1, !;
    do_resolve(T1, Clause, Literal).

search_matching_clause([], [], [], [], [], []).
search_matching_clause([Clause|KB], KBOriginal, Resoluted, TargetClause, Matching, Resolvent) :-
    copy_term(TargetClause, TargetClauseSubst),
    copy_term(Clause, ClauseSubst),
    is_not_member([TargetClause, Clause], Resoluted),
    do_resolve(TargetClauseSubst, ClauseSubst, Literal), Literal \= [],
    make_resolvent(TargetClauseSubst, ClauseSubst, Literal, Resolvent),
    is_not_member(Resolvent, KBOriginal),
    Matching = Clause, !;
    search_matching_clause(KB, KBOriginal, Resoluted, TargetClause, Matching, Resolvent).

search_clauses([], [], [], [], [], []).
search_clauses([Clause|KB], KBOriginal, Resoluted, Clause, Matching, Resolvent) :-
    search_matching_clause(KB, KBOriginal, Resoluted, Clause, Matching, Resolvent), Matching \= [], !.
search_clauses([_|KB], KBOriginal, Resoluted, Clause, Matching, Resolvent) :-
    search_clauses(KB, KBOriginal, Resoluted, Clause, Matching, Resolvent).

resolution_helper(KB, _, Result) :-
    member([], KB), Result = "UNSATISFIABLE", write(Result), nl, !.
resolution_helper(KB, Resoluted, Result) :-
    search_clauses(KB, KB, Resoluted, Clause, Matching, Resolvent),
    Clause \= [], Matching \= [],
    resolution_helper([Resolvent|KB], [[Clause, Matching]|Resoluted], Result), !.
resolution_helper(_, _, Result) :-
```

```
    Result = "SATISFIABLE", write(Result), nl.

resolution(KB, Result) :- replace_vars_from_clause(KB, KBNew), resolution_helper(KBNew, [], Result).

resolution_on_list([]).
resolution_on_list([KB|KBs]) :-
    write("Resolution for:"), nl,
    write(KB), nl,
    resolution(KB, _), nl, nl,
    resolution_on_list(KBs).

solve :-
    read_file("./inputs/resolution-variables.txt", KBs),
    process_sentences(KBs, KBParsed),
    resolution_on_list(KBParsed).

solve_football :-
    read_file("./inputs/football.txt", KBs),
    unpack_kb(KBs, KB),
    process_sentence(KB, KBParsed),
    write(KBParsed), nl,
    resolution(KBParsed, _).
```

resolution-propositional.pl

```
:- ["./utils/read.pl", "./utils/parse.pl", "./utils/utils.pl"].
is_tautology([]) :- false.
is_tautology([H|T]) :- neg(H, Hneg), member(Hneg, T), !.
is_tautology([_|T]) :- is_tautology(T).

remove_tautologies([], []).
remove_tautologies([Clause|KB], KBNew) :- is_tautology(Clause), remove_tautologies(KB, KBNew).
remove_tautologies([Clause|KB], [Clause|KBNew]) :- remove_tautologies(KB, KBNew).

make_resolvent(Clause1, Clause2, Literal, Resolvent) :-
    eliminate(Literal, Clause1, Clause1New),
    neg(Literal, LiteralNeg),
    eliminate(LiteralNeg, Clause2, Clause2New),
    merge(Clause1New, Clause2New, Resolvent).

do_resolve([], _, []).
do_resolve([H1|T1], Clause, Literal) :-
    neg(H1, H2), member(H2, Clause), Literal = H1, !;
    do_resolve(T1, Clause, Literal).

search_matching_clause([], [], [], [], [], []).
search_matching_clause([Clause|KB], KBOriginal, Resoluted, TargetClause, Matching, Resolvent) :-
    is_not_member([TargetClause, Clause], Resoluted),
    do_resolve(TargetClause, Clause, Literal), Literal \= [],
    make_resolvent(TargetClause, Clause, Literal, Resolvent),
    is_not_member(Resolvent, KBOriginal),
    Matching = Clause, !;
    search_matching_clause(KB, KBOriginal, Resoluted, TargetClause, Matching, Resolvent).

search_clauses([], [], [], [], [], []).
search_clauses([Clause|KB], KBOriginal, Resoluted, Clause, Matching, Resolvent) :-
    search_matching_clause(KB, KBOriginal, Resoluted, Clause, Matching, Resolvent), Matching \= [], !.
```

```
search_clauses([_|KB], KBOriginal, Resoluted, Clause, Matching, Resolvent) :-
    search_clauses(KB, KBOriginal, Resoluted, Clause, Matching, Resolvent).

resolution_helper(KB, _) :-
    member([], KB), write("UNSATISFIABLE"), !.
resolution_helper(KB, Resoluted) :-
    remove_tautologies(KB, KBOpt),
    search_clauses(KBOpt, KBOpt, Resoluted, Clause, Matching, Resolvent),
    Clause \= [], Matching \= [],
    resolution_helper([Resolvent|KBOpt], [[Clause, Matching]|Resoluted]), !.
resolution_helper(_, _) :- write("SATISFIABLE").
resolution(KB) :- resolution_helper(KB, []).

resolution_on_list([]).
resolution_on_list([KB|KBs]) :-
    write("Resolution for:"), nl,
    write(KB), nl,
    resolution(KB), nl, nl,
    resolution_on_list(KBs).

solve :-
    read_file("./inputs/resolution-propositional.txt", KBs),
    process_sentences(KBs, KBParsed),
    resolution_on_list(KBParsed).
```

sat-solver.pl

```
 :- ["./utils/read.pl", "./utils/parse.pl", "./utils/utils.pl"].
eliminate_clauses_with_literal([], _, []).
eliminate_clauses_with_literal([C|KB], Literal, KBNew) :-
    member(Literal, C),
    eliminate_clauses_with_literal(KB, Literal, KBNew), !.
eliminate_clauses_with_literal([C|KB], Literal, [CNew|KBNew]) :-
    neg(Literal, NegLiteral), member(NegLiteral, C), eliminate(NegLiteral, C, CNew),
    eliminate_clauses_with_literal(KB, Literal, KBNew), !.
eliminate_clauses_with_literal([C|KB], Literal, [C|KBNew]) :-
    eliminate_clauses_with_literal(KB, Literal, KBNew).

literal_in_clause(Literal, Clause, 1) :- member(Literal, Clause).
literal_in_clause(_, _, 0).

count_literal(_, [], 0).
count_literal(Literal, [Clause|KB], F) :-
    literal_in_clause(Literal, Clause, LC),
    count_literal(Literal, KB, FR), F is LC + FR.

extract_truth_literal(n(X), [X, false]).
extract_truth_literal(X, [X, true]).

extract_literals_from_KB(KB, Literals) :-
    append(KB, KBLiterals),
    list_to_set(KBLiterals, Literals).


extract_first_literal_from_clause([], _).
extract_first_literal_from_clause([Literal|_], Literal).
```

```
choose_most_frequent_literal_helper(_, [], [_, 0]).
choose_most_frequent_literal_helper(KB, [Literal|Literals], [Literal, Frequency]) :-
    count_literal(Literal, KB, Frequency),
    choose_most_frequent_literal_helper(KB, Literals, [_, CurrentMaxFrequency]),
    Frequency > CurrentMaxFrequency, !.
choose_most_frequent_literal_helper(KB, [_|Literals], [MaxLiteral, MaxFrequency]) :-
    choose_most_frequent_literal_helper(KB, Literals, [MaxLiteral, MaxFrequency]).

choose_most_frequent_literal(KB, Literal) :-
    extract_literals_from_KB(KB, Literals),
    choose_most_frequent_literal_helper(KB, Literals, [Literal, _]).

choose_shortest_clause([], [_, 2**63 - 1]).
choose_shortest_clause([Clause|KB], [Clause, ClauseLength]) :-
    length(Clause, ClauseLength),
    choose_shortest_clause(KB, [_, CurrentMinClauseLength]),
 ClauseLength < CurrentMinClauseLength, !.
choose_shortest_clause([_|KB], [MinLiteral, MinClauseLength]) :-
 choose_shortest_clause(KB, [MinLiteral, MinClauseLength]).

choose_literal_from_shortest_clause(KB, Literal) :-
    choose_shortest_clause(KB, [Clause, _]),
    extract_first_literal_from_clause(Clause, Literal).

davis_putnam([], []).
davis_putnam(KB, _) :- member([], KB), !, fail.
davis_putnam(KB, [LiteralSol|S]) :-
    choose_literal_from_shortest_clause(KB, Literal),
    extract_truth_literal(Literal, LiteralSol),
    eliminate_clauses_with_literal(KB, Literal, KBNew),
    davis_putnam(KBNew, S), !.
davis_putnam(KB, [LiteralSol|S]) :-
    choose_literal_from_shortest_clause(KB, Literal), neg(Literal, NegLiteral),
    extract_truth_literal(NegLiteral, LiteralSol),
    eliminate_clauses_with_literal(KB, NegLiteral, KBNew),
    davis_putnam(KBNew, S).

print_solution_helper([]).
print_solution_helper([[Literal, Truth]]) :- write(Literal), write("/"), write(Truth).
print_solution_helper([[Literal, Truth]|S]) :-
    write(Literal), write("/"), write(Truth), write(";"), print_solution_helper(S).
print_solution(S) :- write("{"), print_solution_helper(S), write("}").

sat_solve(KB) :- davis_putnam(KB, Solution), write("YES"), nl, print_solution(Solution), nl, !.
sat_solve(_) :- write("NO"), nl.

sat_solve_on_list([]).
sat_solve_on_list([KB|KBs]) :-
    write("SAT Solver for: "), write(KB), nl,
    sat_solve(KB), nl,
    sat_solve_on_list(KBs).

solve :-
    read_file("./inputs/sat-solver.txt", KBs),
    process_sentences(KBs, KBParsed),
    sat_solve_on_list(KBParsed).
```