Knowledge Representation & Reasoning

Project 1 - Resolution & Davis-Putnam SAT Solver

Radu-Tudor Vrînceanu, 407 (Artificial Intelligence)

December 9, 2024

University of Bucharest
Faculty of Mathematics and Computer Science
2024-2025

1 Introduction

In the beginning, this project was built using the SWI-Prolog language and Python for the web application using StreamLit as the foundational library that empowers it. In order to start the project and get it up and running please assure that on the machine there is Python v3.10+ installed and also the StreamLit utility is installed using `pip/pip3 install streamlit`. Right after that run the command `streamlit run app.py` and the application will start without any problems on your localhost.

The **Knowledge Base** that needed to be defined is the following one which was inspired from *professor Gordon S. Novak Jr.* - Resolution Algorithm:

1. Every adult loves to play with every lego set.
2. Anyone who plays with some lego set is not a technology fanatic.
3. Anyone who uses a smartphone is a technology fanatic.
4. Anyone who buys a smartphone either craves it or uses it.
5. Paul buys a smartphone.
6. Darth Vader's Scene is one of the lego sets.
7. **Question**: If Paul is an adult, then Paul craves to have a smartphone.

We will use the following symbols to denote the relationships:

1. $Adult(X)$ - denotes if X is an adult;
2. $LEGOSet(Y)$ - denotes if Y is a lego set;
3. $UsesSmartphone(X)$ - denotes if X uses a smartphone;
4. $TechFanatic(X)$ - denotes if X is a technology fanatic;
5. $CravesSmartphone(X)$ - denotes if X craves or not for a smartphone;
6. $BuysSmartphone(X)$ - denotes if X buys a smartphone;
7. $PlaysWith(X, Y)$ - denotes the relationship between X and Y;
8. DVS - denotes the constant for the Darth Vader's Scene inside the **Knowledge Base**;

9. $Paul$ - denotes the constant for Paul inside the **Knowledge Base**.

We will convert the **Knowledge Base** in **First Order Logic** based on the symbols denoted previously:

1.

$$\forall x \forall y\, (Adult(x) \land LEGOSet(y) \Rightarrow PlaysWith(x, y))$$

2.

$$\forall x\, (\exists y\, (LEGOSet(y) \land PlaysWith(x, y)) \Rightarrow \neg TechFanatic(x))$$

3.

$$\forall x\, (UsesSmartphone(x) \Rightarrow TechFanatic(x))$$

4.

$$\forall x\, (BuysSmartphone(x) \Rightarrow (CravesSmartphone(x) \lor UsesSmartphone(x)))$$

5.

$$BuysSmartphone(Paul)$$

6.

$$LEGOSet(DVS)$$

7. **Question**:

$$Adult(Paul) \Rightarrow CravesSmartphone(Paul)$$

We will convert the **First Order Logic** to **CNF**:

1. $\forall x \forall y(\neg Adult(x) \lor \neg LEGOSet(y) \lor PlaysWith(x, y))$

2. $\forall x \forall y(\neg LEGOSet(y) \lor \neg PlaysWith(x, y) \lor \neg TechFanatic(x))$

3. $\forall x(\neg UsesSmartphone(x) \lor TechFanatic(x))$

4. $\forall x(\neg BuysSmartphone(x) \lor CravesSmartphone(x) \lor UsesSmartphone(x))$

5. $BuysSmartphone(Paul)$

6. $LEGOSet(DVS)$

7. $\neg Adult(Paul) \lor CravesSmartphone(Paul)$

To ensure the question is entitled for this **Knowledge Base** we will use the negation of the question stated i.e. $Adult(Paul) \land \neg CravesSmartphone(Paul)$. And check if the KB with the negation of the question is UNSATISFIABLE. This means our CNFs will be the following ones:

1. $\forall x \forall y(\neg Adult(x) \lor \neg LEGOSet(y) \lor PlaysWith(x, y))$

2. $\forall x \forall y(\neg LEGOSet(y) \lor \neg PlaysWith(x, y) \lor \neg TechFanatic(x))$

3. $\forall x(\neg UsesSmartphone(x) \lor TechFanatic(x))$

4. $\forall x(\neg BuysSmartphone(x) \lor CravesSmartphone(x) \lor UsesSmartphone(x))$

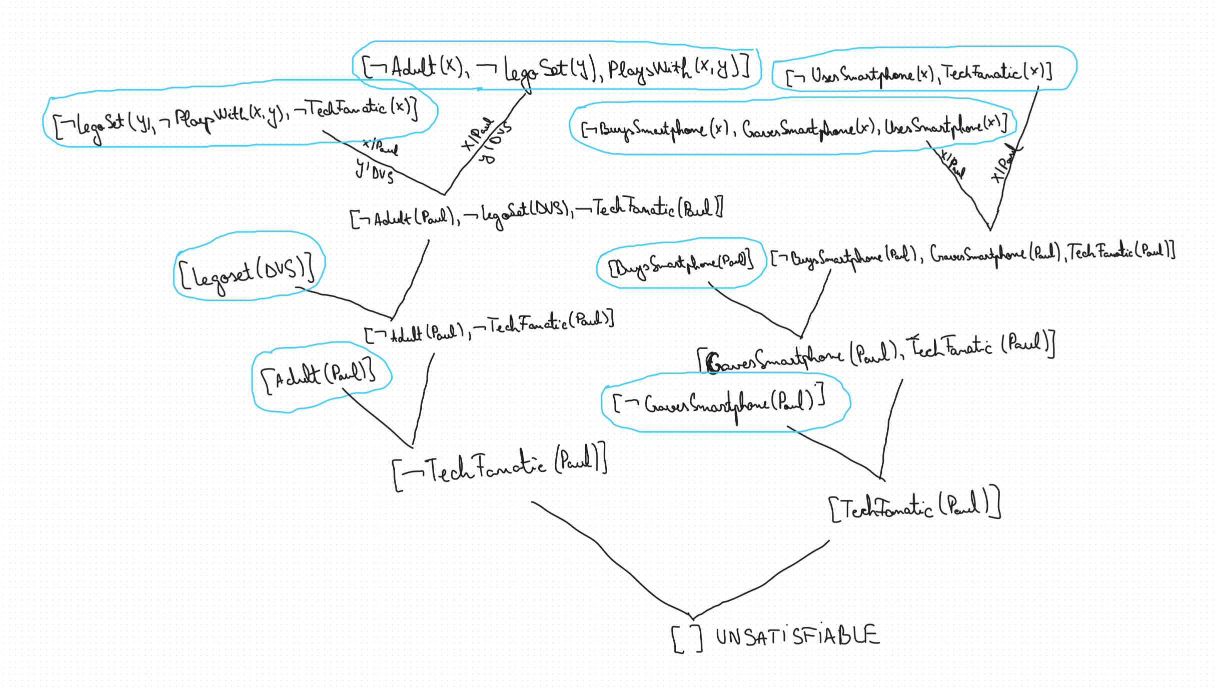
5. $BuysSmartphone(Paul)$

6. $LEGOSet(DVS)$

7. $Adult(Paul)$

8. $\neg CravesSmartphone(Paul)$

The manual demonstration of this Knowledge base is the following one:



[¬Adult(X), ¬LegoSet(Y), PlaysWith(X,Y)]
[¬UsesSmartphone(x), TechFanatic(x)]
[¬LegoSet(y), ¬PlaysWith(x,y), ¬TechFanatic(x)]
[¬BuysSmartphone(x), CravesSmartphone(x), UsesSmartphone(x)]
x/Paul
y/DVS
x/Paul
y/DVS
x/Paul
x/Paul
[¬Adult(Paul), ¬LegoSet(DVS), ¬TechFanatic(Paul)]
[LegoSet(DVS)]
[BuysSmartphone(Paul)]
[¬BuysSmartphone(Paul), CravesSmartphone(Paul), TechFanatic(Paul)]
[¬Adult(Paul), ¬TechFanatic(Paul)]
[CravesSmartphone(Paul), TechFanatic(Paul)]
[Adult(Paul)]
[¬CravesSmartphone(Paul)]
[¬TechFanatic(Paul)]
[TechFanatic(Paul)]
[ ] UNSATISFIABLE

2 Resolution

The resolution algorithm is a refutation-based proof technique used to derive a contradiction from a set of logical clauses, thereby proving unsatisfiability. It operates on clauses in Conjunctive Normal Form (CNF) and relies on the refutation strategy, meaning that the algorithm assumes the negation of the statement to be proved and tries to derive a contradiction. The code for the resolution procedure is the following one:

```
read_input(FilePath, X) :-
    see(FilePath), read(X), seen.

retrieve_all_literals_selection(KB, Lit) :-
    member(Clause, KB),
    member(Lit, Clause).
retreive_negation_literals_selection(AllLiterals, NegLit) :-
    member(Lit, AllLiterals),
    negate(Lit, NegLit),
    \+ member(NegLit, AllLiterals).
exclude_pure_clauses_from_kb(KB, NewKB) :-
    findall(Lit, retrieve_all_literals_selection(KB, Lit), AllLiterals),
    findall(NegLit, retreive_negation_literals_selection(AllLiterals, NegLit), NotFoundLiterals),
    exclude(is_pure_clause(NotFoundLiterals), KB, NewKB).
is_pure_clause(NotFoundLiterals, Clause) :-
    member(Lit, Clause), negate(Lit, NegLit), member(NegLit, NotFoundLiterals).

exclude_tautologies_from_kb(KB, NewKB) :- exclude(is_tautology, KB, NewKB).
```

```prolog
is_tautology(Clause) :- member(Lit, Clause), negate(Lit, NegLit), member(NegLit, Clause).

exclude_subsumed_clauses_from_kb(KB, NewKB) :- exclude(is_subsummed(KB), KB, NewKB).
is_subsummed(KB, Clause) :- member(NewClause, KB), NewClause \= Clause, subset(NewClause, Clause).

negate(n(X), X) :- !.
negate(X, n(X)).

solve(FirstClause, SecondClause, Resultant) :-
    member(Lit, FirstClause),
    negate(Lit, NegLit),
    member(NegLit, SecondClause),
    select(Lit, FirstClause, RestOfFirstClause),
    select(NegLit, SecondClause, RestOfSecondClause),
    append(RestOfFirstClause, RestOfSecondClause, IntermediateResultant),
    sort(IntermediateResultant, Resultant).

resolve_kb(KB, NewKB) :-
    findall(Resultant,
        (member(FirstClause, KB),
         member(SecondClause, KB),
         FirstClause @< SecondClause,
         solve(FirstClause, SecondClause, Resultant),
         \+ member(Resultant, KB)),
    NewClauses),
    append(KB, NewClauses, TempKB),
    sort(TempKB, NewKB).

apply_solve_resolution(KB) :-
    (member([], KB) -> write('UNSATISFIABLE'), nl, ! ;
     resolve_kb(KB, NewKB),
     (KB == NewKB -> write('SATISFIABLE'), nl ; apply_solve_resolution(NewKB))).

resolution_fol(FilePath) :-
    read_input(FilePath, KB),
    write('Initial KB: '), write(KB), nl,
    apply_solve_resolution(KB).

resolution_propositional(FilePath) :-
    read_input(FilePath, KB),
    write('Initial KB: '), write(KB), nl,
    exclude_pure_clauses_from_kb(KB, FilteredPureClausesKB),
    write('Our KB after pure classes removal: '), write(FilteredPureClausesKB), nl,
    exclude_tautologies_from_kb(FilteredPureClausesKB, FilteredTautologiesClausesKB),
    write('Our KB after tautologies removal: '), write(FilteredTautologiesClausesKB), nl,
    exclude_subsumed_clauses_from_kb(FilteredTautologiesClausesKB, NewKB),
    write('Our KB after subsumed clauses removal: '), write(NewKB), nl,
    apply_solve_resolution(KB).
```

3 Davis-Putnam SAT Solver

The Davis-Putnam (DP) algorithm is a classic method for solving the Boolean satisfiability problem (SAT), which involves determining if there exists an assignment of truth values to variables that makes a given Boolean formula true. The algorithm, introduced by Martin Davis and Hugh Putnam in 1960, is a backtracking-based approach that recursively eliminates variables and their associated clauses.

Strategies to select the atom for the dot operation that were implmented for this project were:

1. the **most frequent** atom present in all the clauses;

2. the atom in the **shortest clause**.

```
retreive_p_by_value(P, ValueOfP, NewP) :-
(
    ValueOfP = true -> NewP = P
    ;
    ValueOfP = false -> negate(P, NegP), NewP = NegP
).

dot_operation([], _, _, []) :- !.
dot_operation([CurrentClause | RestClauses], P, ValueOfP, NewClauses) :-
    retreive_p_by_value(P, ValueOfP, NewP),
    (member(NewP, CurrentClause) ->
        dot_operation(RestClauses, P, ValueOfP, NewClauses)
        ;
        negate(NewP, NegP), (member(NegP, CurrentClause) ->
            delete(CurrentClause, NegP, NewCurrentClause),
            (NewCurrentClause == [] -> fail ; true),
            dot_operation(RestClauses, P, ValueOfP, RecursiveNewClauses),

            NewClauses = [NewCurrentClause | RecursiveNewClauses]
            ;
            dot_operation(RestClauses, P, ValueOfP, RecursiveNewClauses),

            NewClauses = [CurrentClause | RecursiveNewClauses]
            )
        ).

choose_most_frequent_atom(KB, P) :-
    findall(Lit, retrieve_all_literals_selection(KB, Lit), AllLiterals),
    construct_frequency_list(AllLiterals, P).
% https://stackoverflow.com/questions/50437617/prolog-function-that-returns-the-most-frequent-element-in-a-list
construct_frequency_list(AllLiterals, P) :-
    sort(AllLiterals, UniqueLiterals),
    findall([Freq, X], (member(X, UniqueLiterals), include(=(X), AllLiterals, XX), length(XX, Freq)), Freqs),
    sort(Freqs, SFreqs),
    last(SFreqs, [Freq, P]).
apply_solve_davis_putnam_most_frequent([], []) :- !.
apply_solve_davis_putnam_most_frequent(KB, _) :- member([], KB), !, fail.
apply_solve_davis_putnam_most_frequent(KB, Solution) :-
    choose_most_frequent_atom(KB, P),
    (
        dot_operation(KB, P, true, NewKBTrue),
        write('Choosen␣Atom:␣'), write(P), write('␣with␣value␣TRUE'), nl,
        write('Clauses␣after␣dot␣operation:␣'), write(NewKBTrue), nl,
        apply_solve_davis_putnam_most_frequent(NewKBTrue, RecursiveSolutions),

        Solution = [atom(P, true) | RecursiveSolutions]
        ;
        dot_operation(KB, P, false, NewKBFalse),
        write('Choosen␣Atom:␣'), write(P), write('␣with␣value␣FALSE'), nl,
        write('Clauses␣after␣dot␣operation:␣'), write(NewKBFalse), nl,
        apply_solve_davis_putnam_most_frequent(NewKBFalse, RecursiveSolutions),

        Solution = [atom(P, false) | RecursiveSolutions]
        ).
davis_putnam_most_frequent(FilePath) :-
    read_input(FilePath, KB),
    write('Initial␣KB:␣'), write(KB), nl,
    (apply_solve_davis_putnam_most_frequent(KB, Solution) ->
        write('YES'), nl, write('Solution:␣'), write(Solution)
        ;
        write('NO')
        ).

choose_shortest_clause_atom(KB, P) :-
```

```
    findall([Length, Clause], (member(Clause, KB), length(Clause, Length)), AllClauseLengths),
    sort(AllClauseLengths, SortedAllClauseLengths),
    SortedAllClauseLengths = [[_, ShortestClause] | _],
    member(P, ShortestClause).
apply_solve_davis_putnam_shortest_clause([], []) :- !.
apply_solve_davis_putnam_shortest_clause(KB, _) :- member([], KB), !, fail.
apply_solve_davis_putnam_shortest_clause(KB, Solution) :-
    choose_shortest_clause_atom(KB, P),
    (
        dot_operation(KB, P, true, NewKBTrue),
        write('Choosen Atom: '), write(P), write(' with value TRUE'), nl,
        write('Clauses after dot operation: '), write(NewKBTrue), nl,
        apply_solve_davis_putnam_most_frequent(NewKBTrue, RecursiveSolutions),

        Solution = [atom(P, true) | RecursiveSolutions]
        ;
        dot_operation(KB, P, false, NewKBFalse),
        write('Choosen Atom: '), write(P), write(' with value FALSE'), nl,
        write('Clauses after dot operation: '), write(NewKBFalse), nl,
        apply_solve_davis_putnam_most_frequent(NewKBFalse, RecursiveSolutions),

        Solution = [atom(P, false) | RecursiveSolutions]
        ).

davis_putnam_shortest_clause(FilePath) :-
    read_input(FilePath, KB),
    write('Initial KB: '), write(KB), nl,
    (apply_solve_davis_putnam_shortest_clause(KB, Solution) ->
        write('YES'), nl, write('Solution: '), write(Solution)
        ;
        write('NO')
        ).
```