Project 1 KRR – Resolution & Davis Putnam

Author: Marinescu Alexandru

Group: 407

Table of Contents:

Resolution

Satisfiability

Given a Knowledge Base, we desire to discover new facts given the existing ones. This is the purpose of reasoning. This needs to also be done automatically, meaning to find a set of patterns to use in order to derive the new facts. Given a new fact (clause), how do we know if we can add it our KB?

Let's say that KB is the following set of clauses {α1, …, αn } and α is the new clause. We can add it to KB if:

1) 1 KB ⊨ α
2) ⊨ [(α1^ … ^αn) ⊃ α]
3) KB U {¬α} is not satisfiable
4) KB U {¬α} ⊨ ¬TRUE

Resolution Derivations

We first need to convert the sentences in the Knowledge Base, into clauses, in their CNF form.

CNF = conjunctive normal form = ***conjunction*** of ***disjunction*** of ***literals*** (could be in negative form as well).
Ex: *If I eat something, I am full*.
We already have an implication (R → G), which is equivalent with (¬R ∨ G), which is already in CNF.

To check if we can derive the clause α it is sufficient to do the following:

1) Convert the sentences in KB and ¬α into CNF;
2) Determine the satisfiability of the resulting set;

The basis of this is with the fact that if 2 elements of a conjunction (disjunctions) contain the clause and the other contains its negation, those can be excluded from the overall evaluation of the conjunction and can obtain a new set of clauses, more simplified (can apply for more than 2 elements).

If we have:

- Clause C1 ∪ {ρ}
- Clause C2 ∪ {¬ρ}

We can infer a **resolvent** (a new set of clauses): C1 ∪ C2 , which is added to the initial set.

We stop, when:

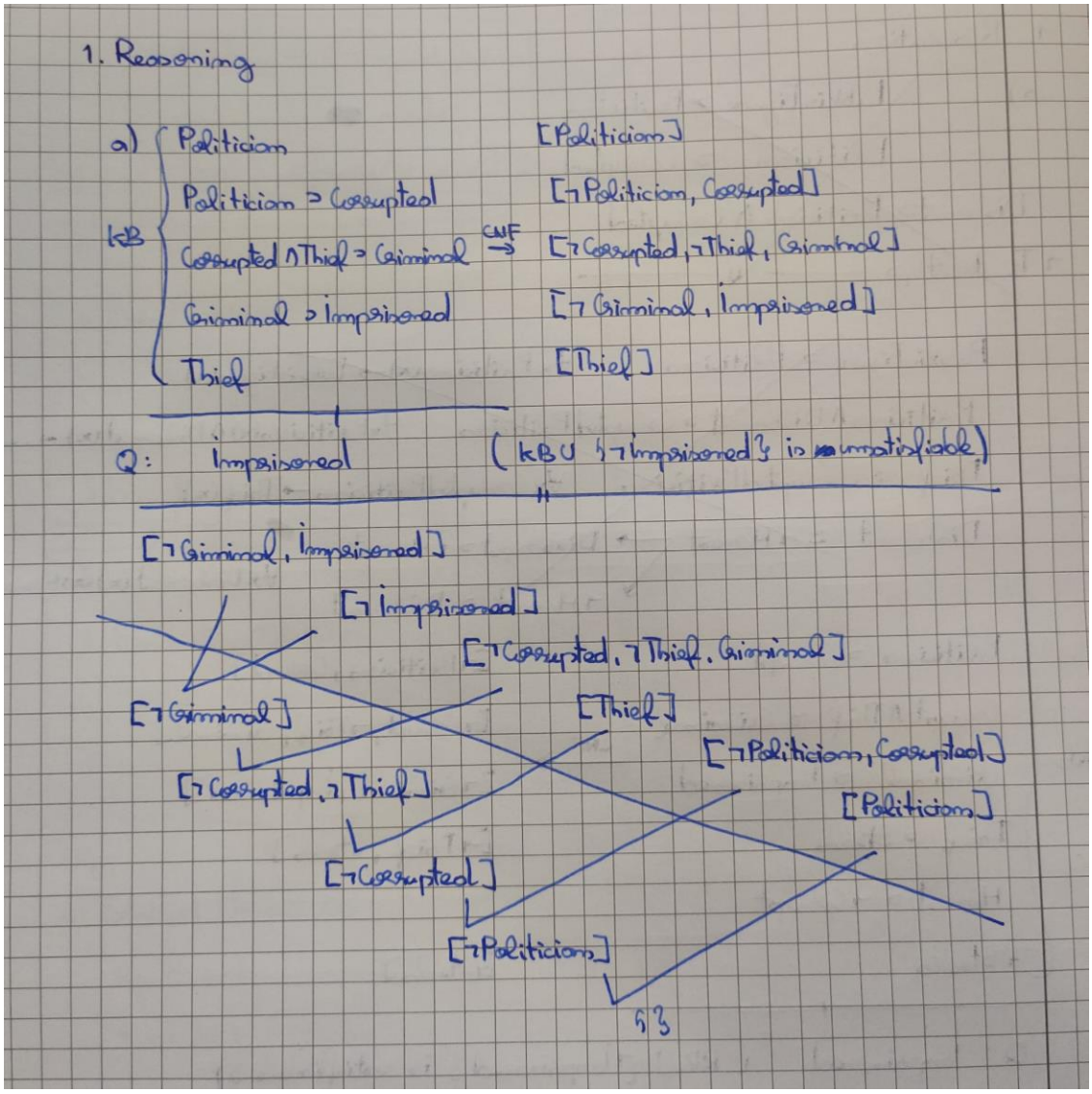
- We derive [] clause (disjunction of nothingness) → KB is **unsatisfiable** → KB ⊨ α
- We derived all possible clauses and couldn't obtain [] clause → KB is **satisfiable**

Personal Example & Given Cases

I have the following sentences as an example:

1) Every person is a politician [Politician]
2) Every person who is a politician is corrupted [¬ Politician, Corrupted]
3) Every person who is corrupted and a thief is a criminal [¬Corrupted, ¬Thief, Criminal]
4) Every person who is a criminal is imprisoned [¬Criminal, ¬Imprisoned]
5) Every person is a thief [Thief]

I want to derive if every person is imprisoned [Imprisoned]?



For the given cases, I have found out:

I. [(¬a or b) and (c or d) and (¬d or b) and ¬b and (¬c or b) and e and (f or a or b or ¬f)] - UNSATISFIABLE
II. [(¬b or a) and (¬a or b or e) and (a or ¬e) and ¬a and e] - UNSATISFIABLE
III. [(¬a or b) and (c or f) and ¬c and (¬f or b) and (¬c or b)] - SATISFIABLE
IV. [(a or b) and (¬a or ¬b) and c] - SATISFIABLE

Applications and Implementation

Implementation was done using the steps mentioned above and described in the course as well. The file content was formatted similar to the exercise in the 3rd laboratory (ex: [a, b, n(b)]. [d, e, n(f)].).
Thus, excluding the input rules to extract the KB, there are 3 main groups of rules:

- Rules for choosing the complementary
- Rules for resolution cases (recursive and end cases)
- Actual resolve rule

Reasoning in FOL is more of a backbone for other inference techniques, so specific applications for this are sparse. What could be rather noted is the way to choose the next clause for resolution and there are a couple of techniques here:

1) **Basic resolution:** repeatedly selects two clauses from the knowledge base (KB), resolves them to produce a resolvent, and adds the resolvent to the KB if it's not already present. This straightforward approach aligns with the foundational resolution principle but does not explicitly apply strategies to prune or limit the search space.
2) **Linear resolution:** resolves the most recently derived resolvent against another clause, creating a straightforward, linear deduction structure**.**
3) **Pruning Unnecessary Paths:** reduce the search space in resolution-based theorem by removing duplicate clauses, or more specific clauses (least count), etc.

Davis-Putnam Sat Solver

Satisfiability

Unlike the previously mentioned resolution, the approach is to search for an interpretation (a set of truth/false assignments for literals, or variables) that makes the given set of clauses of a KB true. It's like a simplification for KB when we want to know if it's satisfiable , or not. This is how SaT Solvers generally work, including Davis-Putnam.

Thus, a KB is:

- **satisfiable** if after the simplification, the set of clauses in the KB can be simplified to {}
- **unsatisfiable** if after the simplification, the set of clauses in the KB can be simplified to []

The difference relies in how the graph of clauses is constructed and parsed.

Interpretation DP Procedure

C•m is defined as following: C•m = {c|c∈C, m∉c, ¬m∉c} U {(c- ¬m| c∈C, m∉c, ¬m∈c}, meaning:

- First set is the set of clauses in C that don't contain m, and thus we don't need m in their evaluation
- Second set is the set of clauses in C which contain the negation of m, and are simplified without it

Following the above mentioned logic, DP Procedure attempts at each step to choose an atom m, compute C•m, check the **satisfiable/unsatisfiable** end cases, and if not, try with the negation of that atom C• ¬m.

At the next step, it will choose another atom from the remaining ones. There are multiple methods of choice here:

- p appears in the most clauses in C;
- p appears in the fewest clauses in C;
- p appears in the shortest clause in C;
- p is the most balanced atom in C (i.e. the number of positive occurrences in C is closest to the number of negative occurrences);
- p is the least balanced atom;

Given Cases

For the given cases, I have found out:

I. [toddler and (¬toddler or child) and (¬child or ¬male or boy) and (¬infant or child) and (¬child or ¬female or girl) and female and girl] – SATISFIABLE with interpretation, [child/true,female/true,girl/true,boy/true,toddler/true]
II. [toddler and (¬toddler or child) and (¬child or ¬male or boy) and (¬infant or child) and (¬child or ¬female or girl) and female and ¬girl] – UNSATISFIABLE
III. [(¬a or b) and (c or d) and (¬d or b) and (¬c or b) and ¬b and e and (f or a or b or ¬f)] – UNSATISFIABLE

IV. [(¬b or a) and (¬a or b or e) and e and (a or ¬e) and ¬a] – UNSATISFIABLE
V. [(¬a or ¬e or b) and (¬d or e or ¬b) and (¬e or f or ¬b) and (f or ¬a or e) and (e or f or ¬b)] – SATISFIABLE with interpretation [e/true,b/true,f/true]
VI. [(a or b) and (¬a or ¬b) and (¬a or b) and (a or ¬b)] – UNSATISFIABLE

Please, bear in mind that for the SATISFIABLE cases, there could be another interpretation as well, meaning it's not unique.

Applications and Implementation

Started with 2 rules for the end cases of the recursive call. Then, I attempted to implement 2 strategies:

- atoms from the shortest clause
- and the most frequent atoms

For these I used 2 atoms *most_frequent* and *shortest_clause* which are decided based on numeric inputs in the entry solve rule (either 1, or 0, exception case treated).

Finally, there are 2 rules, for simplification of a clause (when that atom is true, when it's not true). And last one is the rule that performs these operations to build the new set of clauses at each step.

Note: there are separate rules for input of the KB from a file. The file content was formatted similar to the exercise in the 4th laboratory (ex: [[n(a), b], [c, d]].)

Most applications for this seem to involve satisfiability modulo theories (SMT), which is a SAT problem in which propositional variables are replaced with formulas of a mathematical theory, making interpretation a bit more known.

A personal notable case is Zelkova from AWS which uses automated reasoning to analyze the current policies of an AWS product and future consequences of those policies (what other accesses will that product receive, or what does it need).

Bibliography

1) **How AWS uses automated reasoning to help you achieve security at scale (** https://aws.amazon.com/blogs/security/protect-sensitive-data-in-the-cloud-with-automated-reasoning-zelkova/ **)**
2) **Automated Reasoning - Stanford Encyclopedia of Philosophy – First published Wed Jul 18, 2001; substantive revision Wed Oct 5, 2005 (** https://plato.stanford.edu/ARCHIVES/WIN2009/entries/reasoning-automated/ **)**
3) **3rd and 4th courses of Knowledge Representation and Reasoning, 2024-2025, University of Bucharest, Faculty of Mathematics and Informatics, prof. M. Cidota**

Code Implementations

Resolution

```
load_kb(File, KB):- see(File),
   read_clauses(KB),
   seen.

read_clauses([]):-
   at_end_of_stream,
   !.
read_clauses([Clause|Rest]):-
   \+ at_end_of_stream,
   read(Clause),
   (Clause \= end_of_file ->
      read_clauses(Rest)
   ;  read_clauses(Rest)
   ).

solve_resolution(File):-
   load_kb(File, KB),
   res(KB).

complementary(Literal1, Literal2) :-
   Literal1 = n(Literal2).
complementary(n(Literal1), Literal2) :-
   Literal1 = Literal2.

choose_clauses(KB, Clause1, Clause2) :-
   member(Clause1, KB),
   member(Clause2, KB),
   Clause1 \= Clause2.

resolve(Clause1, Clause2, Resolvent) :-
   select(Literal, Clause1, Rest1),
   complementary(Literal, OppositeLiteral),
   select(OppositeLiteral, Clause2, Rest2),
   append(Rest1, Rest2, ResolventUnsorted),
   sort(ResolventUnsorted, Resolvent).

res(KB) :-
   member([], KB),
   !,
   write('UNSATISFIABLE'), nl.

res(KB) :-
   choose_clauses(KB, Clause1, Clause2),
   resolve(Clause1, Clause2, Resolvent),
   \+ member(Resolvent, KB),
   res([Resolvent | KB]).

res(_) :-
   write('SATISFIABLE'), nl.
```

Davis-Putnam Sat Solver

```
davis_putnam([], [], _) :- !.
davis_putnam(L, _, _) :-
  member([], L),
  !, fail.
davis_putnam(L, [C/true | Solution], Strategy) :-
  choose_atom(L, C, Strategy),
  perform_operation(L, C, true, L1),
  davis_putnam(L1, Solution, Strategy).
davis_putnam(L, [C/false | Solution], Strategy) :-
  choose_atom(L, C, Strategy),
  perform_operation(L, C, false, L2),
  davis_putnam(L2, Solution, Strategy).

perform_operation([], _, _, []) :- !.
perform_operation([Clause | Rest], Atom, Value, NewClauses) :-
  (member(Atom, Clause), Value = true ->
    perform_operation(Rest, Atom, Value, NewClauses);
   member(n(Atom), Clause), Value = false ->
    perform_operation(Rest, Atom, Value, NewClauses);
   \+ member(Atom, Clause), \+ member(n(Atom), Clause) ->
    perform_operation(Rest, Atom, Value, NewRest),
    NewClauses = [Clause | NewRest];
   simplify_clause(Clause, Atom, Value, SimplifiedClause),
   perform_operation(Rest, Atom, Value, NewRest),
   NewClauses = [SimplifiedClause | NewRest]
  ).

simplify_clause(Clause, Atom, true, Simplified) :-
  delete(Clause, n(Atom), Simplified).
simplify_clause(Clause, Atom, false, Simplified) :-
  delete(Clause, Atom, Simplified).

choose_atom(L, Atom, most_frequent) :-
  choose_most_frequent(L, Atom).
choose_atom(L, Atom, shortest_clause) :-
  choose_shortest_clause_atom(L, Atom).

choose_most_frequent(L, Atom) :-
  findall(A, (member(Clause, L), member(A, Clause), \+ A = n(_)), Positives),
  findall(A, (member(Clause, L), member(n(A), Clause)), Negatives),
  append(Positives, Negatives, AllAtoms),
  msort(AllAtoms, SortedAtoms),
  clumped(SortedAtoms, CountedAtoms),
  sort(2, @>=, CountedAtoms, [Atom-_|_]).

clumped([], []).
clumped([H | T], [H-N | Clumped]) :-
  clump(H, T, N, Rest),
  clumped(Rest, Clumped).

clump(_, [], 1, []).
clump(X, [X | T], N, Rest) :-
  clump(X, T, N1, Rest),
  N is N1 + 1.
clump(X, [Y | T], 1, [Y | T]) :-
  X \= Y.

choose_shortest_clause_atom(L, Atom) :-
  ( member([], L) ->
    fail
  ; sort(2, @=<, L, [ShortestClause | _]),
    member(Atom, ShortestClause)
  ).

read_clauses(File, Clauses) :-
  open(File, read, Stream),
  read(Stream, Clauses),
  close(Stream).

solve(File, 1) :-
  Strategy = most_frequent,
  read_clauses(File, Clauses),
  (davis_putnam(Clauses, Solution, Strategy) ->
    writeln('YES'),
    writeln('Solution:'),
    writeln(Solution);
   writeln('NO')).

solve(File, 0) :-
  Strategy = shortest_clause,
  read_clauses(File, Clauses),
  (davis_putnam(Clauses, Solution, Strategy) ->
    writeln('YES'),
    writeln('Solution:'),
    writeln(Solution);
   writeln('NO')).

solve(_, _) :-
  writeln('Invalid strategy. Use 1 for most frequent or 0 for shortest clause.'),
  fail.
```