Knowledge Representation and Reasoning Project 2

Olteanu Fabian Cristian

FMI, AI Master, Year 1

1. Implementing the Back/Forward-Chaining Procedures

1.1. Formulation of the Rules

For this task the following rules were used:

KB
1. In Romania it is a crime for civilians to own guns.
2. Criminals break laws.
3. Lawbreakers get sent to jail.

This knowledge base can be converted into the conjunctive normal form like so:

CNF
1. $\neg Romanian(x) \vee \neg Civilian(x) \vee \neg GunOwner(x) \vee Criminal(x)$
2. $\neg Criminal(x) \vee LawBreaker(x)$
3. $\neg LawBreaker(x) \vee SentToJail(x)$

1.2. The Back-Chaining procedure

The back-chaining procedure is an algorithm that can be used to decided whether a full set of sentences can be entailed or not from a given KB. It is expressed in pseudocode below [1].

Algorithm 1 SOLVE($q_1, ..., q_n$)

```
Input: a finite list of atomic sentences $q_1, ..., q_n$
Output: "yes" or "no" depending on whether a given KB entails all of the $q_i$
  if $n = 0$ then
      return yes
  end if
  for each clause $c \in KB$ do
      if $c = [q_1, \neg p_1, ..., \neg p_m] and SOLVE [p_1, ..., p_m, q_2, ..., q_n]$ then
          return yes
      end if
  end for
  return no
```

This algorithm was implemented in Prolog following this way of thinking:

- The knowledge base (the rules from the previous subsection) is read from an external file and the following three questions are asked by the program: "Is the person Romanian?", "Is the person a civilian?" and "Does the person own a gun?" (typing "stop." in the console will interrupt this and proceed with the answers gathered up to that point).
- The back-chaining procedure is implemented as a predicate called backchaining/2, which takes the KB and a list of atomic clauses (Q) that are based on the answers given by the user. For example, should the user answer "yes", "yes" and "no", the program would assign the values a, b and not(c) to Q.
- If Q is empty, the predicate returns true
- If Q is not empty, the predicate foreach/2 is used to iterate through KB, in conjuncture with a custom predicate, do_loop/3, which takes a clause from the KB, the KB itself and Q.
- The do_loop predicate simulates the for-block from Algorithm 1. It first sorts the KB Clause from the respective iteration, since doing that to a positive horn clause arranges the elements in such a way that the positive atomic sentence is placed first $(q_1, \neg p_1, ..., \neg p_m)$.
- Finally, another custom predicate called if_condition/4, is used to compare the heads of the sorted KB Clause and of the Q list and to call the backchaining predicate recursively with the respective parameters from the algorithm presented above. If the condition is met, the procedure returns false (in order to stop further execution), and the message "yes" is displayed.

References

[1] Ronald Brachman, Hector Levesque, “Knowledge Representation and Reasoning,” *Morgan Kaufmann*, p. 92, 2004

```
n(not(A), A).
n(A, not(A)).

head([H|_], H).
tail([_|T], T).

neg_list([], []).
neg_list([Head|Tail], [NegHead|NegTail]) :-
    n(Head, NegHead),
    neg_list(Tail, NegTail), !.

if_condition(KB, Q, QClause, KBClause_Head) :-
    QClause == KBClause_Head,
    backchaining(KB, Q).

backchaining(_, Q) :-member([], Q).
backchaining(KB, Q) :-
    foreach(
        member(KBClause, KB),
        do_loop(KBClause, KB, Q)
)    .

do_loop(KBClause, KB, Q) :-
    sort(KBClause, KBClause_Sorted),
    head(KBClause_Sorted, KBClause_Sorted_Head),
    head(Q, QClause),
    tail(KBClause_Sorted, KBClause_Sorted_Tail),
    neg_list(KBClause_Sorted_Tail, KBClause_Sorted_Tail_Pos),
    tail(Q, QTail),
    append(KBClause_Sorted_Tail_Pos, QTail, NewQ),
    if_condition(KB, NewQ, QClause, KBClause_Sorted_Head) -> fail; !.

read_KB_and_output_answer_bc(Str, _, []) :-
    at_end_of_stream(Str), !.

read_KB_and_output_answer_bc(Str, Q, [_|T]) :-
    not(at_end_of_stream(Str)),
    read(Str, X),
    backchaining(X, Q) ->
        write(no), nl,
        read_KB_and_output_answer_bc(Str, Q, T)
    ;
        write(yes), nl,
        read_KB_and_output_answer_bc(Str, Q, T).

main :-
    open('data.in', read, Str),
    Questions = [
        'Is the person Romanian?',
        'Is the person a civilian?',
        'Does the person own a gun?'
    ],
    In = [a, b, c],
    read_answers(Questions, In, Q),
    read_KB_and_output_answer_bc(Str, Q, _).
```

```
read_answers(_, [], _).
read_answers([H|T], [In_H|In_T], Out) :-
    writeln(H),
    read(Input),
(
        Input == 'stop'
    ->
        writeln('Program stopped')
    ;
        Input == 'yes'
    ->
        append([In_H], [], Out),
        read_answers(T, In_T, Out)
    ;
        Input == 'no'
    ->
        n(In_H, In_H_Neg),
        append([In_H_Neg], [], Out),
        read_answers(T, In_T, Out)
)    .
```

Listing 1: Resolution Implementation in Prolog