CISC 481 Programming Assignment 2

October 29, 2019

In this assignment you are asked to write a simplified version of a resolution theorem-prover. The theorem prover will be simplified in that it will take a reduced set of logic (Horn Clauses) as its axioms, and work with a conjunction of positive literals as its goal. In addition, it should implement the set of support heuristic strategy in proving the theorem.

1 Resolution Theorem Proving

Resolution is a form of proof by refutation. A resolution theorem prover takes as input a number of axioms in clausal form along with a negated theorem, and attempts to derive the $\emptyset$ clause through the process of *resolution*. (Note that if the goal actually follows from the axioms then the negated goal causes an inconsistency which allows the $\emptyset$ clause to be derived.) Resolution operates by taking two clauses that each contain a unifiable literal – in positive form in one clause and in negative form in the other (these are called *complementary literals*). The resolvent is obtained by combining all of the other literals in the two clauses (after making the appropriate substitutions), dropping the complementary literals (which cancel each other). If you continue to do this and there is a contradiction, then the $\emptyset$ clause will be reached.

For simplicity we will restrict our theorem prover so that the axioms are all *Horn Clauses* and the goal will be restricted to a conjunction of positive literals.

A Horn clause has the logical form:

$$P \Longleftarrow (P_1 \bigwedge P_2 \bigwedge ... \bigwedge P_n),$$

where $P_1, P_2 ... P_n$ are all positive literals (i.e., positive atomic formulas). This should be read, if $P_1, P_2 ... P_n$ are all true, then P is true. Notice, that this formula has the equivalent clausal form:

$$P \bigvee \neg P_1 \bigvee \neg P_2 \bigvee \neg ... \bigvee \neg P_n.$$

Thus each horn clause (and thus each axiom to your theorem prover) will be the disjunction of one positive atomic formula and 0 to n negative atomic formulas. The way these should be represented will be discussed below.

The one goal clause will have the logical form:

$$Q_1 \bigwedge Q_2 \bigwedge ... \bigwedge Q_m,$$

Where $Q_1, Q_2, ... Q_m$ are all positive literals. This formula, when negated, becomes the clause:

$$\neg Q_1 \bigvee \neg Q_2 \bigvee ... \bigvee \neg Q_m.$$

The advantage of restricting the input axioms to Horn clauses is that if we resolve an axiom clause (which contains 1 positive literal) against a clause that has both the form of the goal (all negative literals) and contains a literal which unifies with the positive literal in the axiom clause, the result will have the same form as the goal clause. Also note that since only the first literal in an axiom clause is in positive form (everything else is negative) it is the only literal eligible to form a complementary literal with a non-axiom clause literal.

2 Your Theorem-Prover

For this Lisp implementation, each axiom will be a Horn Clause. A Horn clause will be coded as a list of predicates, where the first predicate will be assumed non-negated and the rest of the predicates will be assumed negated (so we do not need to explicitly represent negation in the system). The goal clause will be coded as a list of predicates all of which will be assumed negated.

A predicate (literal) will also be represented as a list. Its first element will be the predicate name and the remaining elements will be zero or more constant and/or variable names. Variable names will begin with '?' while constant and predicate names will not. Functions, such a `(father ?x)`, will not be part of the implementation.

For instance, consider the following axioms:

```
(grandparent ?x ?y) ⟸ (parent ?x ?z) ⋀ (parent ?z ?y)
(parent Brian Doug)
(parent Doug Mary)
(parent Mary Sue)
```

These would be represented in your system as:

```
(defvar *premises*
        '(((grandparent ?x ?y) (parent ?x ?z) (parent ?z ?y))
          ((parent Brian Doug))
          ((parent Doug Mary))
          ((parent Mary Sue))))
```

Where, if you wanted to find out who was the grandparent of whom, the (negated) goal would be represented:

```
(defvar *goals*
        '((grandparent ?x ?y)))
```

The job of the theorem prover is to try to prove that the axioms and the negated theorem together are inconsistent using resolution to derive the $\emptyset$ clause. When you yourself try to prove a set of clauses inconsistent, you can choose which clauses to resolve against which, and often, because you know what they *mean*, you come up with a quick proof. But since a theorem-prover is working without your insights, it can make useless resolutions. In fact, it could go down a useless path of resolutions for an infinite amount of time. To get around this problem, we invent strategies for deciding which clauses to resolve when.

The *set of support* strategy dictates that we resolve clauses only if at least one of them *came from* a goal clause (i.e., a goal clause or a derived clause whose predecessors include a goal clause). This strategy is *complete*, i.e., it guarantees a proof if one exists – eventually. As we noted before, a derived clause worth deriving will always have the same form as the initial goal clause. Further note that due to our restriction on the form of the axiom and goal clauses, the required resolutions will always involve a clause from the initial set of axioms against a clause from the set of support (i.e., the goal or a derived clause).

In writing your theorem-prover, notice that what it is doing can be viewed as a kind of graph search. Design your theorem prover so that it can do the following:

- it must perform a depth-first search that will generate at least one proof if no infinite recursion occurs;
- it must be designed so that you can modify it relatively easily to perform a breadth-first search;
- if there are variables in the goal clause, it must report their bindings for the proof(s) it generates, as in the example below:

  ```
  --> (prove *premises* *goals*)
  ?x = Brian, ?y = Mary
  ?x = Doug, ?y = Sue
  t
  ```

- and it must abort after some maximum number of resolutions, say 300, to catch infinite recursions.

3 A Simplified Example: Seeing it as Graph Search

Take your axioms to be your KB – these will be used in ”generating successors”.

AXIOMS:

```
((grandparent ?x ?y) (parent ?x ?z) (parent ?z ?y))
((parent Brian Doug))
((parent Doug Mary))
((parent Mary Sue))
```

NOTE: the first predicate in each of these lists is in positive form, while the remainder are in negative form. E.g., the first one really means:
`(grandparent ?x ?y) OR NOT(parent ?x ?z) OR NOT(parent ?z ?y)`

Your open list is going to contain the items that can be resolved against the KB – this will be your growing ”set of support”. Remember each element in your set of support will contain only negative clauses so they can’t be resolved against each other (only against things in your KB). Initially your open list will only contain the negated goal clause which is `((grandparent ?x1 ?y1))` in this case.

NOTE: this clause really stands for `NOT(grandparent ?x1 ?y1)`.

Start by taking the first element off of open and generating all of the results of resolving it against the KB (generating all of its successors). Note that you should generate a "successor" clause if the first predicate in the `expl` unifies with the first predicate of a KB clause. The result should be the append of the remainder of the KB clause involved and the remainder of `expl`. (NOTE: this ordering is important so that you get the infinite loop in the test program!!) By convention, these will all be negative clauses (like the original goal) and belong on the open list (using open like a stack for DFS and as a queue for BFS). In this case, there is only one complementary literal and thus we have only one successor and that is:

```
((parent ?x ?z) (parent ?z ?y)) -- with the binding ?x1/?x, ?y1/?y
```

Next time through the graph search the above node is made expl – notice that it will resolve with several elements in the KB generating 3 successor states:

```
((parent Doug ?y)) -- with the binding ?x1/?x, ?y1/?y and ?x/Brian, ?z/Doug
((parent Mary ?y)) -- with the binding ?x1/?x, ?y1/?y and ?x/Doug, ?z/Mary
((parent Sue ?y)) -- with the binding ?x1/?x, ?y1/?y and ?x/Mary, ?z/Sue
```

Now take the first of these against the KB. It unifies with the first "parent" clause in the KB generating:

```
null -- with the binding ?x1/?x, ?y1/?y and ?x/Brian, ?z/Doug and ?y/Mary
```

Thus we get the output: `?x1 = Brian, ?y1 = Mary` (this we got by chasing through the binding list finding the values applied to the variables in our original goal – I used `?x1` and `?y1` here because the initial KB contains and `?x` and `?y` and I thought this would be less confusing – you want to report the variables the user put in the initial goal).

Since there are still nodes on the open list, we continue trying to find a complementary literal to `((parent Mary ?y))`. Again we reach the null clause with the successor:

```
null -- with the binding ?x1/?x, ?y1/?y and ?x/Doug, ?z/Mary and ?y/Sue
```

Thus we get the output: `?x1 = Doug, ?y1 = Sue`

We keep going, but find that the only element left on the open list does not generate any successors, and thus we fail.

Please note the intention here was to get you to see how this could be viewed as a graph search. Details have been glossed (e.g., I have not included uniquify described below).

4 Some Functions You Should Write

You may find the following functions useful to write:

(unify pred1 pred2 bindings) – Given two predicates and a *list of list* of previous bindings, it returns a new (improved) list of a list of bindings, or `nil` if the unification failed. It returns a list of a list in order to distinguish between a successful resolution that required no bindings `(nil)` from a failed resolution `nil`. Bindings are lists each giving a variable and its binding. `unify` returns any previous bindings along with the new ones if it succeeds. Note that it is probably easiest to write the unifier so that it may return a list of transitive bindings like `(((?x ?y) (?y constant)))`. Examples:

```
--> (unify '(p ?x) '(p ?y))
(((?x ?y)))

--> (unify '(p ?x ?y) '(p ?z ?w) '(((?x ?w))))
(((?y ?z) (?w ?z) (?x ?w)))
```

(binding var bindings) – Chases var through transitive bindings. This is important given the way it was suggested `unify` be implemented. For instance:

```
--> (binding '?x '(((?x ?y) (?y ?z) (?w ?q) (?z ?w))))
?q
```

(uniquify clause) – returns a copy of `clause` with variables renamed uniquely. You will need to use this function before each unification to guarantee a "fresh" instance of the premise you are unifying. This is because you may be resolving a premise clause with another clause derived from that same premise, and the variables from the two instances must not be the same. You may find the (gensym) function helpful here.

```
--> (uniquify '((p ?x ?y) (a ?x ?z) (a ?z ?y)))
((p ?x4 ?y7) (a ?x4 ?z3) (a ?z3 ?y7))
```

5 Testing Your Theorem Prover

In order to test your functions you should load the file `test-prover.lisp` available on Canvas. It will contain three sets of test data.

The test data `*premises-1*` and `*goals-1*` should illustrate the difference between depth-first and breadth-first search; `*premises-2*` and `*goals-2*` comprise a typical Prolog theorem and should exercise your theorem-prover well; and `*premises-3*` and `*goals-3*` contains a left-recursion to illustrate that Prolog can go into an infinite loop.

First, run your theorem-prover on the three test data using depth-first search. Then modify the program to do breadth-first search, and run it a second time on `*premises-1*` and `*goals-1*`. Hand in a typescript of these runs, along with a listing of your program. You should also show test runs of your major helping functions (e.g., the ones suggested here) just in case the test cases we give do not show their full power.