COS 570 AI HW4: Resolution Theorem Prover

Nicholas Soucy

December 2, 2021

Contents

1 Introduction

This document discusses the resolution theorem prover implementation created by Nicholas Soucy. This resolution theorem prover is influenced by the resolution theorem prover created by Dr. Roy Turner in the COS 570 AI github.

All axiom sets are explained in depth along with each function in ns_rtp.lisp. Directions on how to run the code along with experiments are explained in the Running RTP section.

2 Methods

This section explains the axiom sets used in this implementation along with the resolution theorem prover.

2.1 Axiom Sets

2.1.1 Garden World

The Garden World axiom set is as follows:

1. John likes carrots.
2. Mary likes carrots.
3. John grows vegetables that he likes.
4. Carrots are vegetables.
5. When you like a vegetable, you grow it.
6. To eat something, you have to own it.
7. When you grow something, you own it.
8. In order to grow something, you must own a garden.

We can then convert this to first order logic:

1. $Likes(John, Carrots)$
2. $Likes(Mary, Carrots)$
3. $\forall x Likes(John, x) \wedge Vegetable(x) \rightarrow Grows(John, x)$
4. $Vegetable(Carrots)$
5. $\forall x \forall y Likes(x, y) \wedge Vegetable(y) \rightarrow Grows(x, y)$
6. $\forall x \forall y Eats(x, y) \rightarrow Owns(x, y)$
7. $\forall x \forall y Grows(x, y) \rightarrow Owns(x, y)$
8. $\forall x \forall y \exists g Grows(x, y) \rightarrow Owns(x, g) \wedge Garden(g)$

Now for our Resolution Theorem Prover (RTP), we need to covert our first order logic statements into conjunctive normal form (CNF). The CNF of the garden world axiom set is listed below:

1. $Likes(John, Carrots)$
2. $Likes(Mary, Carrots)$
3. $\neg Likes(John, x_1) \vee \neg Vegetable(x_1) \vee Grows(John, x_1)$
4. $Vegetable(Carrots)$
5. $\neg Likes(x_2, y_1) \vee \neg Vegetable(y_1) \vee Grows(x_2, y_1)$
6. $\neg Eats(x_3, y_2) \vee Owns(x_3, y_2)$
7. $\neg Grows(x_4, y_3) \vee Owns(x_4, y_3)$
8. $\neg Grows(x_5, y_4) \vee Owns(x_5, sk1(x_5, y_4))$
9. $\neg Grows(x_6, y_5) \vee Garden(sk2(x_6, y_5))$

2.1.2 Custom Axiom Set: Mean Geese

1. All geese are mean.
2. Tim is a goose
3. When something is mean, it will bite you.
4. When something bites you, you run from it.
5. Greg is running.
6. Greg is nice.
7. When you are nice, you aren't mean.

Now we convert this to first order logic:

1. $\forall x Goose(x) \rightarrow Mean(x)$
2. $Goose(Tim)$
3. $\forall x \forall y Mean(x) \rightarrow Bites(x, y)$
4. $\forall x \forall y Bites(x, y) \rightarrow Running(y)$
5. $Running(Greg)$
6. $Nice(Greg)$
7. $\forall x Nice(x) \rightarrow \neg Mean(x)$

Now for our Resolution Theorem Prover (RTP), we need to covert our first order logic statements into conjunctive normal form (CNF). The CNF of the mean geese axiom set is listed below:

1. $\neg Goose(x_1) \vee Mean(x_1)$
2. $Goose(Tim)$
3. $\neg Mean(x_2) \vee Bites(x_2, y_1)$
4. $\neg Bites(x_3, y_2) \vee Running(y_2)$
5. $Running(Greg)$
6. $Nice(Greg)$
7. $\neg Nice(x_4) \vee \neg Mean(x_4)$

2.2 Resolution Theorem Prover

To begin the discussion of our Resolution Theorem Prover (RTP) implementation, lets look at the psudocode:

Listing 1: Resolution Theorem Prover

```
input: Clause, Axioms, Bindings
output: Result, Bindings
begin
    if Clause is nil
        return nil, bindings
    else
        for literal in clause
            if literal is a computable predicate and not a true predicate
                (rtp (remove literal from clause) axioms bindings)
            else
                for axiom in axioms
                    for axiom-literal in axiom
                        if resolves(literal axiom-literal)
                            (rtp resolved-literal axiom bindings)
                        else
                            return nil, bindings
end
```

As you can see, RTP is quite straight forward and short when looking at the logic.

We start with an input clause, axioms, and bindings:

1. Clause: The inputted clause is the negated clause that is inputted from the user that we wish to find a contradiction with. For example, 'John grows Carrots → (grows John Carrots)' would be negated into 'John does not grow Carrots → (not (grows John Carrots))'. This negated clause would be the starting point in finding a contradiction.

2. Axioms: The user also specifies the axiom set used in the proof. In this implementation we use the garden world and the custom geese world as described in the previous section. In our axiom set, we integrated a simple heuristic: Axiom clause sorting. By sorting our axiom set list in increasing character size, we always look at the shortest axioms first for resolution. This gives us a larger percent chance in resolving into a contradiction faster because we will have less literals in our clause to resolve next.

3. Bindings: Our bindings is a list of bounded variables. Initially it is empty, but if we have a literal (Likes John Carrots) and Unify.lisp determines it matches with our axiom clause (not (Likes ?x2 ?y1), the bindings list would keep the information ((?x2 John), (?y1 Carrots). The bindings list is reported to the user after if a contradiction is found.

With our inputs explained, we can now explain the algorithm in more detail. In the beginning, we do a simple check to see if the given clause to prove is nil. If it is, this means we found our contradiction and we return.

If there is a given clause to contradict, we iterate through each literal in the clause. If the literal is a computable predicate (like less than or greater than), and is not a true predicate, we remove that literal from the clause, as it is something we do not determine through resolution, and recursively call RTP with the updated clause. NOTE: the garden world nor the geese world have computable predicates, so this is not really a concern for us. At this stage we would have to compute the predicate directly to determine its truth value. Once we do not use computable predicates, it is left as an exercise to the reader if they decide to use an axiom set with them.

If we have a standard literal, we then search through each literal in each axiom in the axiom set starting with the smallest axioms first as our search heuristic. We then check if the current literal in our clause resolves with our axiom literal, if it does, then we recursively call RTP with the resolved result and current bindings. However, if the current clause literal resolves to nil, then we return our values because we reached our contradiction.

To aid in the simplicity of our RTP, we employ helper functions.

2.2.1 User Wrapper Function: Prove

Our wrapper function Prove is shown in psudocode below:

Listing 2: Prove

```
input: theorem, axioms, bindings
output: text to user, bindings, axiom set
begin
    results, bindings ← RTP(negate(theorem), axioms, bindings)
    if not results
        print("Theorem can be proven, contradiction found. Bindings = ~s." bindings)
        print("Axiom Set Used = ~s" axioms)
    else
        print("Theorem cannot be proven.")
end
```

This wrapper function prove is what the actual user calls. It automatically calls RTP with the negated theorem. RTP will return nil if we get a contradiction, so we print the theorem can be proven along with the bindings and axiom set. If we did not find a contradiction we report failure to the user.

2.2.2 Negation Functions

The negation function psudocodes are given below:

Listing 3: Negate

```
input: clause
output: negated clause
begin
   if clause is nil
       return nil
   else
       for literal in clause
           negated clause ← negated clause.append(negate literal(literal))
       return negated clause
end
```

Listing 4: Negate Literal

```
input: Literal
output: Negated Literal
begin
   if not in literal
       return literal without not
   else
       return not literal
end
```

Once we are working with CNF, we do not have any and's in our expressions. When A and B is true, A is true and B is true, so we can treat A and B as separate statements. Therefore, we do not have to worry about and's in our negate function, only or's.

Once we store our clauses in a list, if we have multiple literals in the clause it is an implied or. Therefore, we use de Morgan's law on the statement and return a list of list with the multiple negated literals. To negate a literal, we simply remove a not if it is present, or add one if it is not.

2.2.3 Resolving Functions

The resolving function psudocodes are given below:

Listing 5: Resolves

```
input: Literal 1, Literal 2, bindings
output: True or False
   Unify(negate-literal(Literal 1), Literal 2, Bindings)
end
```

Listing 6: Resolvent

```
input: Clause-literal, Clause, Axiom, Axiom-literal, Bindings
output: Resolved Literal
begin
   return append(remove(Clause-literal, Clause), remove(Axiom-literal, Axiom))
end
```

Most of the resolving work happens in the Unify function that is explained in Unify.pdf given by Dr. Roy Turner. Our Resolve function first determines if two literals resolve by calling unify, if they do, return true, else, false.

The Resolvent function, implements the resolution theorem rule in logic given here:

$$\frac{\alpha \vee \beta,\ \cancel{\beta} \vee \gamma}{\alpha \vee \gamma} \tag{1}$$

We therefore remove the clause-literal from clause and axiom-literal from axiom, then append the results together.

2.2.4 Unbound Variable Function

This function determines if a given literal has a variable in it that is unbound. Psudocode for the algorithm is given below.

Listing 7: Unbound Variable in Literal

```
input: Literal
output: True or False
begin
```

```
    if literal is nil
        return nil
    else if it is a list
        for element in list
            Unbound Variable in Literal(element)
    else if variable?(literal)
        return t
    else
        return nil
end
```

The function starts by determining if the literal is nil, if it is, return nil. If it is a list, recursively call the function with each element in the list, if any of them are unbound, then the entire literal is unbound. If the literal has a variable (implemented via Dr. Roy Turner's variable? function located in unify.lisp), return true, else, return nil.

2.2.5 Computable Predicate and True Predicate Functions

The psudocode for the computable predicate and true predicate functions are given below:

Listing 8: computable predicate

```
input: Literal
output: True or False
begin
    if literal has a not
        computable predicate(negate-literal(literal))
    else
        assoc(literal, *computable-predicates*)
end
```

The computable predicate function determines if a given literal is a computable predicate by searching in the global variable list names *computable-predicates*, if it is in that list, return true, else return nothing. It also removes not's if they are contained within the literal.

Listing 9: true predicate

```
input: Literal, bindings
output: True or False
begin
    if unbound variable in literal(literal)
        return T
    else if there is a not in literal
        true predicate(not (negate(literal)))
    else computable predicate(literal)
        return evalutated literal
end
```

The true predicate function determines if a given literal is true by seeing if it has an unbound variable, or by evaluating the computable predicate via the association defined in the global variable *computable-predicates* list. NOTE: as both of the axiom sets used in this example have no computable predicates, the evaluate is left as a future work to anybody who needs them. These functions are mostly the same from Dr. Roy Turner's rtp.lisp code as an addition to flesh out the future possibilities for this code.

3 Running RTP

3.1 Directions

To run this program, the user has to know two things: what they want to prove, and what axiom set they want to use. In this example, we only have the garden world and geese world axiom sets, though more can be added in the future.

You call the prove function like this: (prove clause axiom-set) where clause is the conjunctive normal form of a logical clause and axiom-set is the name of the axiom-set you wish to use.

After that, it will display the current progress of the resolution, then if the theorem can be proven or not. If it can, it will produce the bindings and axiom set.

3.2 Experiments

For the experiments, six experiments were ran.

1. John Grows Carrots - Garden world - (grows John Carrots)
2. Mary Owns Carrots - Garden world - (owns Mary Carrots)
3. Nicholas grows Money - Garden world - (grows Nicholas Money)
4. Tim bit Greg - Geese world - (bites Tim Greg)
5. Greg is Mean - Geese world - (mean Greg)
6. Greg bit Tim - Geese world - (bites Greg Tim)

The results text results are shown below in their respective section.

3.2.1 John Grows Carrots - Garden World Experiment

Input - (prove '((grows John carrots)) garden_world)
Output -

```
    Attempting to find contradiction with clause ((NOT (GROWS JOHN CARROTS))).
Resolved: ((NOT (GROWS JOHN CARROTS)))
    with: ((NOT (LIKES JOHN ?X1)) (NOT (VEGETABLE ?X1)) (GROWS JOHN ?X1))
Attempting to find contradiction with clause ((NOT (LIKES JOHN CARROTS))
                                              (NOT (VEGETABLE CARROTS))).
Resolved: ((NOT (LIKES JOHN CARROTS)) (NOT (VEGETABLE CARROTS)))
    with: ((LIKES JOHN CARROTS))
Attempting to find contradiction with clause ((NOT (VEGETABLE CARROTS))).
Resolved: ((NOT (VEGETABLE CARROTS)))
    with: ((VEGETABLE CARROTS))
Attempting to find contradiction with clause NIL.
Theorem can be proven, contradiction found.  Bindings = ((?X1 CARROTS)).
Axiom Set Used = (((VEGETABLE CARROTS)) ((LIKES JOHN CARROTS))
                  ((LIKES MARY CARROTS))
                  ((NOT (GROWS ?X6 ?Y5)) (GARDEN (SK2 ?X6 ?Y5)))
                  ((NOT (EATS ?X3 ?Y2)) (OWNS ?X3 ?Y2))
                  ((NOT (GROWS ?X4 ?Y3)) (OWNS ?X4 ?Y3))
                  ((NOT (GROWS ?X5 ?Y4)) (OWNS ?X5 (SK1 ?X5 ?Y4)))
                  ((NOT (LIKES JOHN ?X1)) (NOT (VEGETABLE ?X1))
                   (GROWS JOHN ?X1))
                  ((NOT (LIKES ?X2 ?Y1)) (NOT (VEGETABLE ?Y1)) (GROWS ?X2 ?Y1)))
```

3.2.2 Mary Owns Carrots - Garden World Experiment

Input - (prove '((owns Mary carrots)) garden_world)
Output -

```
    Attempting to find contradiction with clause ((NOT (OWNS MARY CARROTS))).
Resolved: ((NOT (OWNS MARY CARROTS)))
    with: ((NOT (EATS ?X3 ?Y2)) (OWNS ?X3 ?Y2))
Attempting to find contradiction with clause ((NOT (EATS MARY CARROTS))).
Resolved: ((NOT (OWNS MARY CARROTS)))
    with: ((NOT (GROWS ?X4 ?Y3)) (OWNS ?X4 ?Y3))
Attempting to find contradiction with clause ((NOT (GROWS MARY CARROTS))).
Resolved: ((NOT (GROWS MARY CARROTS)))
    with: ((NOT (LIKES ?X2 ?Y1)) (NOT (VEGETABLE ?Y1)) (GROWS ?X2 ?Y1))
Attempting to find contradiction with clause ((NOT (LIKES MARY CARROTS))
                                              (NOT (VEGETABLE CARROTS))).
```

```
Resolved: ((NOT (LIKES MARY CARROTS)) (NOT (VEGETABLE CARROTS)))
    with: ((LIKES MARY CARROTS))
Attempting to find contradiction with clause ((NOT (VEGETABLE CARROTS))).
Resolved: ((NOT (VEGETABLE CARROTS)))
    with: ((VEGETABLE CARROTS))
Attempting to find contradiction with clause NIL.
Theorem can be proven, contradiction found.  Bindings = ((?Y1 CARROTS)
                                                         (?X2 MARY)
                                                         (?Y3 CARROTS)
                                                         (?X4 MARY)).
Axiom Set Used = (((VEGETABLE CARROTS)) ((LIKES JOHN CARROTS))
                  ((LIKES MARY CARROTS))
                  ((NOT (GROWS ?X6 ?Y5)) (GARDEN (SK2 ?X6 ?Y5)))
                  ((NOT (EATS ?X3 ?Y2)) (OWNS ?X3 ?Y2))
                  ((NOT (GROWS ?X4 ?Y3)) (OWNS ?X4 ?Y3))
                  ((NOT (GROWS ?X5 ?Y4)) (OWNS ?X5 (SK1 ?X5 ?Y4)))
                  ((NOT (LIKES JOHN ?X1)) (NOT (VEGETABLE ?X1))
                   (GROWS JOHN ?X1))
                  ((NOT (LIKES ?X2 ?Y1)) (NOT (VEGETABLE ?Y1)) (GROWS ?X2 ?Y1)))
```

3.2.3 Nicholas Grows Money - Garden World Experiment

Input - (prove '((grows Nicholas money)) garden_world)
Output -

```
    Attempting to find contradiction with clause ((NOT (GROWS NICHOLAS MONEY))).
Resolved: ((NOT (GROWS NICHOLAS MONEY)))
    with: ((NOT (LIKES ?X2 ?Y1)) (NOT (VEGETABLE ?Y1)) (GROWS ?X2 ?Y1))
Attempting to find contradiction with clause ((NOT (LIKES NICHOLAS MONEY))
                                              (NOT (VEGETABLE MONEY))).
Cannot prove the theorem.
```

3.2.4 Tim Bit Greg - Geese World Experiment

Input - (prove '((bites Tim Greg)) geese_world)
Output -

```
    Attempting to find contradiction with clause ((NOT (BITES TIM GREG))).
Resolved: ((NOT (BITES TIM GREG)))
    with: ((NOT (MEAN ?X2)) (BITES ?X2 ?Y1))
Attempting to find contradiction with clause ((NOT (MEAN TIM))).
Resolved: ((NOT (MEAN TIM)))
    with: ((NOT (GOOSE ?X1)) (MEAN ?X1))
Attempting to find contradiction with clause ((NOT (GOOSE TIM))).
Resolved: ((NOT (GOOSE TIM)))
    with: ((GOOSE TIM))
Attempting to find contradiction with clause NIL.
Theorem can be proven, contradiction found.  Bindings = ((?X1 TIM) (?Y1 GREG)
                                                         (?X2 TIM)).
Axiom Set Used = (((NICE GREG)) ((GOOSE TIM)) ((RUNNING GREG))
                  ((NOT (GOOSE ?X1)) (MEAN ?X1))
                  ((NOT (MEAN ?X2)) (BITES ?X2 ?Y1))
                  ((NOT (NICE ?X4)) (NOT (MEAN ?X4)))
                  ((NOT (BITES ?X3 ?Y2)) (RUNNING ?Y2)))
```

3.2.5 Greg is Mean - Geese World Experiment

Input - (prove '((mean Greg)) geese_world)
Output -

```
    Attempting to find contradiction with clause ((NOT (MEAN GREG))).
Resolved: ((NOT (MEAN GREG)))
    with: ((NOT (GOOSE ?X1)) (MEAN ?X1))
Attempting to find contradiction with clause ((NOT (GOOSE GREG))).
Cannot prove the theorem.
```

3.2.6 Greg Bit Tim - Geese World Experiment

Input - (prove '((bites Greg Tim)) geese_world)
Output -

```
    Attempting to find contradiction with clause ((NOT (BITES GREG TIM))).
Resolved: ((NOT (BITES GREG TIM)))
    with: ((NOT (MEAN ?X2)) (BITES ?X2 ?Y1))
Attempting to find contradiction with clause ((NOT (MEAN GREG))).
Resolved: ((NOT (MEAN GREG)))
    with: ((NOT (GOOSE ?X1)) (MEAN ?X1))
Attempting to find contradiction with clause ((NOT (GOOSE GREG))).
Cannot prove the theorem.
```

4 Conclusion

In conclusion we have successfully implemented a resolution theorem prover. As long as our axioms and inputted clause are in Conjunctive Normal Form, then our code will determine if a contradiction is reached within the given axiom set.