Automated Reasoning

Thomas Kaizer

March 6, 2018

Introduction

For this project I worked with two other students so when I refer to we it is in reference to myself and both of them. The purpose of this project was to implement two different inference methods for propositional logic and demonstrate their use on a series of examples. In our case, these methods were entailment and resolution denoted by the methods `entails` and `resolution`. We demonstrate the use of these methods on the following knowledge bases *Modus Ponens, Wumpus World, Horn Clause, Liars and Truth Tellers, and Advanced Liars and Truth tellers.* Propositional logic works by taking a series of rules or aspects of a given world and assuming them to be factual. This is our **knowledge base**. With this knowledge we can take a query in the form of a proposed sentence, and derive a conclusion about the truthfulness of that query by using **inference**. Two methods of inference, which are implemented in this project are **entailment** and **resolution**.

Entailment and the Entails Method

The idea behind entailment is that for any given knowledge base, for all models of the world in which all aspects of our knowledge are true, our assertion α must also be true. In simpler terms, a proposition entails a conclusion if the conclusion is true every time the proposition is true.

I began implementing the truth table enumeration method by first copying over all the files given to us by Professor Ferguson. It took a little while to look through everything and get a sense of what I was looking at but I

eventually came to the incorrect conclusion that I should build my `entails` method inside the `model_imp` class that I created, which basically just filled out the methods from the `model` interface.
Next, my parters and I began work on filling out the `entails` method in class `Prover1`. This came from figure 7.10 in the textbook and was a program for the `TT-Entails` method. In doing this we also had to make the `Check` method which was used recursively to basically generate a truth table based on the given knowledge base and sentence.

```
@Override
public boolean entails(KB kb, Sentence alpha) {

        //Collection<Symbol> sym = kb.symbols();
        Model_imp m = new Model_imp();
        List<Symbol> sym = new ArrayList<Symbol>();
        //sym.add((Symbol) alpha);
        sym.addAll(kb.symbols());
        return Check(kb, alpha, sym, m);
}

public boolean Check(KB kb, Sentence s, List<Symbol> sym, Model_imp m){

    if(sym.isEmpty()) {
        m.dump();
        if(m.satisfies(kb)) {
            return m.satisfies(s);
        }
        else {

            return true;
        }
    }
    else {
        Symbol P = sym.get(0);
        List<Symbol> rest =  sym.subList(1, sym.size());
        //System.out.println("check again");
        return  Check(kb, s, rest, m.set(P, false))&&(Check(kb, s, rest, m.set(P, true)) );
    }
}
```

Figure 1: Our entails and checkall methods

Our implementation `entails` takes in a knowledge base and a query α. More or less the rest of `entails` just instantiated the variables we need to call `check`, which include an empty instance of `Model_imp` and a list *sym* containing all the symbols in the knowledge base, as well as in the query.

Our implementation of `check` was where all the real action happened. It more or less followed the textbook's pseudo code from **Figure 7.10**, but to put it in maybe a slightly more understandable language it did the following:

1. Check if *sym* (the list of all symbols in the knowledge base and the sentence) was empty

2. If *sym* was indeed empty, it checked if the model in that instance satisfied the knowledge base

3. if it did it returned the boolean value associated with checking the satisfiability of the query given the model

4. otherwise it returned true

If *sym* was not empty however, it would do the following:

1. take the first symbol in *sym* and save it as viable P

2. create a new list *rest*, which is a copy of *sym* but without P

3. return `Check` two times, with one replacing the empty model with the assignment of P to `true`, and the other return assigning P to `false`. Both of these recursive calls to `check` would also replace *sym* with *rest*

At first we had some trouble with these methods because they would only print three rows of the `ModusPonensKB` truth table when there should be four, and it always return true whether or not that was correct or not. This ended up being because we were trying to code up `entails` and `check` in the method `Model_imp` because that just logically seemed like a place where the methods would go. It wasn't until later that we discovered the `Prover` interface which was a give away that we should put our methods in something that implements that.

HornClause

There were also issues because we were trying to create the *symbol* parameter in `check` as a collection rather than an ArrayList. Upon fixing this our entails method began to work correctly and generate the entire truth table, as well as a correct boolean value for the arguments it was given.
For our third example we created a knowledge base for the Hornclause problem. To do this, we modeled after the `Modus PonensKB` and the `WumpusWorldKB` classes to create the `HornClauseKB` class. This knowledge base took in the following statements.

$$\text{mammal} \Leftrightarrow mortal$$
$$\text{mythical} \Rightarrow \neg mortal$$
$$\neg mythical \Rightarrow mammal$$
$$(\neg mortal \bigvee mammal) \Rightarrow horned$$
$$\text{horned} \Rightarrow magical$$

It functioned correctly so we decided to move on to part two, instantiating a new method for logical inference. The one we chose to do was resolution because it seemed most relevant to what we had learned in class.

```
public boolean PL_Resolution(KB kb, Sentence alpha){
  Set <Clause> notAlpha = CNFConverter.convert(new Negation(alpha));
  Set <Clause> clauses = CNFConverter.convert(kb);
  clauses.addAll(notAlpha);
  Set <Clause> newClauses = new HashSet<Clause>();
  while (true){
    for(Clause Ci : clauses) {
      for(Clause Cj : clauses) {
        if(!Ci.equals(Cj)) {
          Set <Clause> resolvents;
          resolvents = PL_Resolve(Ci, Cj);
          for(Clause Ck: resolvents){
            if (Ck.isEmpty()){
              return true;
            }
          }
          newClauses.addAll(resolvents);
        }
      }
    }
    if (clauses.containsAll(newClauses)){
      return false;
    }
    clauses.addAll(newClauses);
  }
}
```

Figure 2: Our Resolution method

```java
public Set<Clause> PL_Resolve(Clause A, Clause B){
  Set<Clause> ret = new HashSet<Clause>();
  Clause localA = A.clone();
  Clause localB = B.clone();
  Iterator<Literal> iteratorA = A.iterator();
  Iterator<Literal> iteratorB = B.iterator();
  while(iteratorA.hasNext()){
    Literal a = iteratorA.next();
    while(iteratorB.hasNext()){
      Literal b = iteratorB.next();
      if (a.getContent().equals(b.getContent()) && a.getPolarity() != b.getPolarity()){
        localA.remove(a);
        localB.remove(b);
        Clause localC = localA.clone();
        localC.addAll(localB);
        ret.add(localC);
      }
    }
  }
  return ret;
}
}
```

Figure 3: Our Resolve method

Resolution

Writing up the Resolution and `Resolve` methods proved to be quite a bit more difficult than `entails` and `check`. The biggest issue, for me at least, was figuring out a structure to hold the set of clauses being compared that allowed the algorithm to compare each combination of clauses. We did this using nested `for each` loops. At the time of my writing this however, `resolution` seems to work correctly on `HornClauseKB` and `ModusPonensKB` but not on `WumpusWorldKB`. I am confused as to why it would work on some examples, but not others. My assumption is that using a nested `for each` loop causes duplication of some clauses to occur and this might be what is messing up our program, but that is just speculation.

The problem, defined more specifically is that `Resolution` is printing out its steps but it never gets to a point where it returns an actually boolean `true` or `false` value.

UPDATE We seem to have gotten `resolution` to work! The problem seemed to be where we in `resolve`. We added the clauses we generated at the end of our code to our returned set. We changed this addition to the

middle of the function and it worked! Sometimes you don't know why things work. They just do. Such is the life of a programmer.

Eventually I have decided to add another example, `LandTTKB` as well as `LandTTbKB`, these represent the two parts of the *Liars and Truth Tellers* problem. Upon completion it appears these work with resolution as well, which added a bit more to the confusion as to why `WumpusWorldKB` was not working.

Later on I added *Advanced Liars and Truth Tellers* as well. I did this in the method `BigLandTTKB`. Adding to my confusion, when running entails on this knowledge base, the program doesn't seem to generate more than seven of the eight lines of the truth table that it should, given that there are three constant symbols, and a truth table should have 2^n row when n is the number of constant symbols.

Driver

It is mentioned in my `README.txt` file but I think it is worth talking about how we made the project run all in one program named `Driver`. We also gave it a nice pretty interface where you can see how each example is evaluated by `entails` and `resolution` one at a time. To do this we just took the `main` method from each knowledge base class and turned it into a renamed driver method. We then created instances of these knowledge bases and ran these methods on them in `Driver`, in an orderly fashion.

Conclusion

This project at first appeared fairly straight forward, I had done work programming with boolean algebras before and I assumed this would be fairly similar. In actuality this problem ended up giving me and my group a series of very delicate debugging issues that we had never seen before. I appreciate however the elegant way that `entails` and `resolution` can solve boolean algebra problems. By coding up some more complex examples, I could have a riddle solver on my computer now!