Project 2 Report

Yanhao Ding

October 23, 2018

Contents

The project is developed using Java. The attached README file provides the instructions for running the project. This report outlines the procedure to develop the project and decribes the results of the program.

1 Basic Model Checking

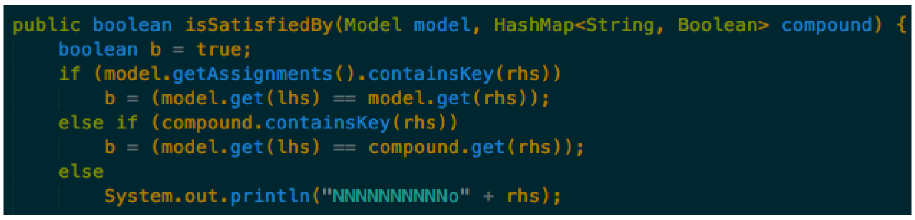
1.1 Representation of Propositional Logic sentence

Class **Symbool** was developed to represent the basic buliding block of propositional logic sentence. The **Board** coded in a separated file named Symbol.java. Since it is the basic unit of propositional logic sentence, it can be extended by another classes. And all the classes that extended the **Symbool** could be add to **ArrayList<Symbol>** , which can work as both knowledge base and all the sentences we need to test. The knowledge base and tested sentences set can also be setted up as classes, however I think it is not necessary to write a class, the arraylist is good enough for this project.

There are three fields of Class **Symbool** including **name**, **lhs** and **rhs**. The field **name** is considering the case like negation which only one symbol is needed. The fields **lhs** and **rhs** is developed to satisfy the need for compound symbols for example the conjuction and implication. The constuctor for this class is overloaded to includes both cases.

The first proposition logic I implement is PureSymbol extend **Symbool**. The constructor needs the input String s to represent the symbol. The method **isSatisfiedBy** is included to give the boolean value of the sysmbol.

Moreover, **Iff** represents the proposition logic if and only if =>. The constructor needs two inputs string **lhs** and **rhs**. method **isSatisfiedBy** is included to give the boolean value of the proposition logic after the calculation. The sample code is shown as Fig 1.



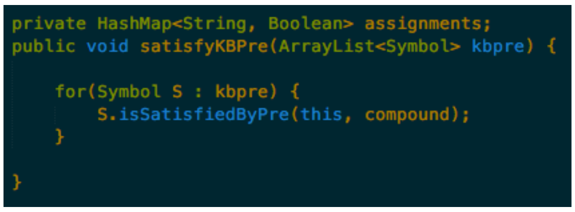
```
public boolean isSatisfiedBy(Model model, HashMap<String, Boolean> compound) {
    boolean b = true;
    if (model.getAssignments().containsKey(rhs))
        b = (model.get(lhs) == model.get(rhs));
    else if (compound.containsKey(rhs))
        b = (model.get(lhs) == compound.get(rhs));
    else
        System.out.println("NNNNNNNNNo" + rhs);
```

Figure 1: Sample Code for proposition logic IF and ONLY IF

In the same way, I implement all the required propositional logic by extending the **Symbool**, including **And**, **Implication**, **Negation**, **Or**. Each of which use the same way as **Iff** and PureSymbol case. So I won't introduce them separately.

1.2 Representation of Models

The basic ideal for model check algorithm inference is enumerating all the models and check that α is true in every model in which the KB in true. Models are assignments of $true$ or $false$ to every propositional symbol. **Model** class was developed to represent the model. HashMap was used to map the symbol to boolean value $true$ or $false$. The method **satisfyKB** and **satisfy**α is implemented separataly in this class to test whether the enumerated model is satisfy the knowledge base and the α, which is shown as Fig 2. The method will interate over all the symbols in the KB or α to check the model. It will call the method **isSatisfiedBy** of the class symbols and pass the intance of model to check. This process can also be seen from Fig 1.



```
private HashMap<String, Boolean> assignments;
public void satisfyKBPre(ArrayList<Symbol> kbpre) {

    for(Symbol S : kbpre) {
        S.isSatisfiedByPre(this, compound);
    }

}
```

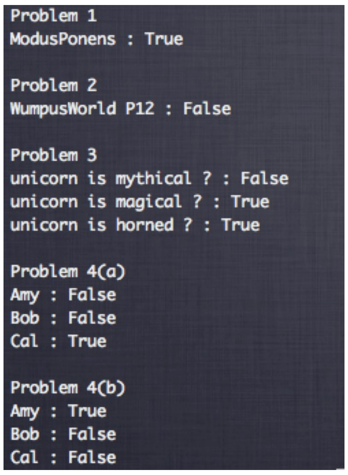
Figure 2: Sample Code for the Model

1.3 Model Check Algorithm

Class **ModelCheck** implements the truth-table enumeration for deciding propositional entialment. Two method was implemented in this class, **TTEntails** and **TTCheckAll**, which is also shown as Figure 7.10 in the AIMA textbook. **TTCheckAll** calls itself twice and the end of the method to recursively get the true table working as model. If the symbol table is empty, call the the satisfy function to check the model. The check returns true if a sentence holds within a model. The variable model represents a partial model-an assignment to some of the symbols. The two resursive calls are connected using logical operation of and.

1.4 Test of Implementation

The overall results are shown as Fig 3. The detail describtion is following the Fig 3.

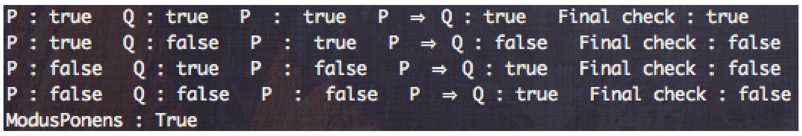


```
Problem 1
ModusPonens : True

Problem 2
WumpusWorld P12 : False

Problem 3
unicorn is mythical ? : False
unicorn is magical ? : True
unicorn is horned ? : True

Problem 4(a)
Amy : False
Bob : False
Cal : True

Problem 4(b)
Amy : True
Bob : False
Cal : False
```

Figure 3: Model Check for all four tests

1.4.1 Modus Ponens

The first problem I test the algorithm on is Modus Ponens {P, P⇒Q } $\models$ Q. The problem is pretty simple, only two variabls P, Q are involved. With two symbols, there are $2^2 = 4$ possible models; The possible all the combination of symbols and corresponding propositional logic value is shown as Fig 4. The figure shows us the value of P, Q enumerate over true or false, the value of P⇒Q is also shown. We can easily see from it that the model satisfy the knowledge requirement is only the first one. So we only care about the first one, in which we also can simply chech the value of Q is also true, satisfying the entailment condition. So we can safely conclude that Modus Ponens is true. So the final output should be true. I will shown the final output of four test examples together since it is more convenient for people to test the implementation all together.



```
P : true    Q : true    P  :  true    P  ⇒ Q : true    Final check : true
P : true    Q : false    P  :  true    P  ⇒ Q : false    Final check : false
P : false    Q : true    P  :  false    P  ⇒ Q : true    Final check : false
P : false    Q : false    P  :  false    P  ⇒ Q : true    Final check : false
ModusPonens : True
```

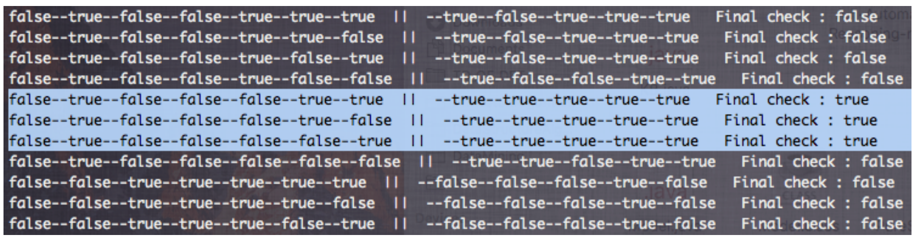
Figure 4: Model Check for Modus Ponens

1.4.2 Wumpus World(Simple)

The simple Wumpus World gives us the knowledge base :

$$\begin{aligned} &\neg P_{1,1} \\ &\neg B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \\ &\neg B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}) \\ &\neg B_{1,1} \\ &\neg B_{2,1} \end{aligned}$$

We need to prove that $P_{1,2}$ which works as α is true or not. This test involves seven symbols $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$ and $P_{2,3}$. With seven symbols, there are $2^7 = 128$ possible models; Part of the truth table is shown as Fig 5. The figure shows us the value of all symbols enumerate over true or false which is shown before || sign. After || sign, the figure shows us the boolean value of five knowledge base sentences after the propositional logical calculation. Three of these models satisfying all the five five knowledge base, which is also highlighted in the figure. Among all three models, we could check wether *alpha* is true or not. From Fig 5, we could see that *alpha* which is $P_{1,2}$ the fourth column is all false. From this we can conclude that this Wumpus World is false. The final results will be shown at the end of four tests.



```
false--true--false--false--true--true--true  ||  --true--false--true--true--true   Final check : false
false--true--false--false--true--true--false  ||  --true--false--true--true--true   Final check : false
false--true--false--false--true--false--true  ||  --true--false--true--true--true   Final check : false
false--true--false--false--true--false--false  ||  --true--false--false--true--true   Final check : false
false--true--false--false--false--true--true  ||  --true--true--true--true--true   Final check : true
false--true--false--false--false--true--false  ||  --true--true--true--true--true   Final check : true
false--true--false--false--false--false--true  ||  --true--true--true--true--true   Final check : true
false--true--false--false--false--false--false  ||  --true--true--false--true--true   Final check : false
false--false--true--true--true--true--true  ||  --false--false--false--true--false   Final check : false
false--false--true--true--true--true--false  ||  --false--false--false--true--false   Final check : false
false--false--true--true--true--false--true  ||  --false--false--false--true--false   Final check : false
```

Figure 5: Model Check for Wumpus World(Simple)

1.4.3 Horn Clauses

First we need to translate the English sentences into the propositional logic symbol language, the knowledge base is shown below.

$$\begin{aligned}
&\text{mythical} \Rightarrow \text{immortal}\\
&\neg\text{mythical} \Rightarrow \text{mammal}\\
&\text{immortal} \vee \text{mammal} \Rightarrow \text{horned}\\
&\text{horned} \Rightarrow \text{magical}
\end{aligned}$$

The α is mythical, magical, horned. We can simply add all the sentences from knowledge and *alpha* to the program and let it determine whether the inferences in correct or not. Since the precedure is almost same as previous two test examples, Instead of attaching the true table to this problem, I just show the final results. "Unicorn is mythica" is false, "Unicorn is magical" is true and "Unicorn is horned" is also true.

1.4.4 Liars and Truth-tellers

For this problem 4(a), the translated knowledge is shown as following :

$$\begin{aligned}
&\text{Amy} \Leftrightarrow \text{Amy} \wedge \text{Cal}\\
&\text{Bob} \Leftrightarrow \neg\text{Cal}\\
&\text{Cal} \Leftrightarrow \text{Bob}\neg\text{Amy}
\end{aligned}$$

The α is Amy, Bob, Cal. Following the above described routine to add the knowledge base to the program, and the outputs : "Amy tell truth" is false, "Bob tell truth" is false, "Cal tell truth" is true.

For this problem 4(b), the translated knowledge is shown as following :

$$\begin{aligned}
&\text{Amy} \Leftrightarrow \neg\text{Cal}\\
&\text{Bob} \Leftrightarrow \text{Amy} \wedge \text{Cal}\\
&\text{Cal} \Leftrightarrow \text{Bob}
\end{aligned}$$

The α is Amy, Bob, Cal. Following the above described routine to add the knowledge base to the program, and the outputs : "Amy tell truth" is true, "Bob tell truth" is false, "Cal tell truth" is false.

2 Resolution-based theorem proven

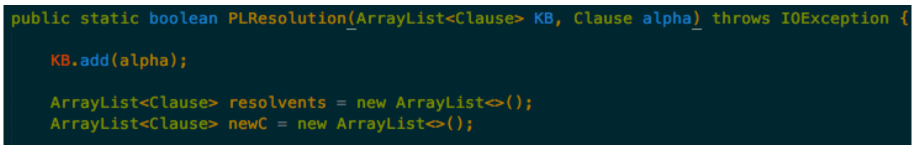
Resolution is a single inference rule that yields a complete inference algorithm when coupled with any complete search algorithm. When we resolve two literal it means for example if there is pit in L11 or L12 or L13, if it not in L11, then it should in L12, L13.

2.1 Representation of Propositional Logic sentence(different from the first one)

For part2, it is possible to use the same representation of Propositional Logic sentence as the first part. However, I found my implememtation is quite complicated so I try to simplify it in the second part. In addition, resolution rules applied only to clauses(that is, disjunction of literals), so the knowledge base and queries should only consisting of clauses. This addition requirement leads to both difficulties and simplicity for the inference program. The arising difficulty is that we must express all the sentences in **conjunctive norm form** or **CNF**. I do not have enough time to write a program to convert the sentence into **CNF** form. So I arrange the sentences I need expressed as clauses by myself(allowed in instrunction). The merits that the clauses bring us is that we only have negation and disjunction now, it is much easier compared with the first part, So I re-write the representation of propositional logic.

Class **Term** was write to represent the baisc literals including negation or not. The class contains two fields one string for the single symbol, one int sign -1 or 1 to represent negation or not. If two instance of Term have the same String and opposite sign, then we can use the resolution to resolve them. Once we set up the building block, we could use ArrayList to add all the Term(Actually I realized that Set is a better choice considering the later on operation).

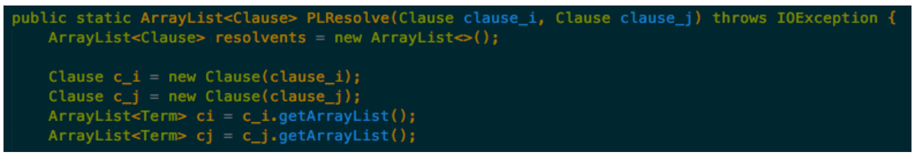
2.2 Resolution Algorithm



```
public static boolean PLResolution(ArrayList<Clause> KB, Clause alpha) throws IOException {

    KB.add(alpha);

    ArrayList<Clause> resolvents = new ArrayList<>();
    ArrayList<Clause> newC = new ArrayList<>();
```

Figure 6: Small sample of PLResolution

The main part of resolution algorithm for proposition logic is implemented as **PLResolution**, which is shown as Fig 6. The input of this method is the a list (Set) of knowledge base clause and query clause. We also need **PLResolve** to resovle two clauses and returns all possible results after it. The implmentation of **PLResolve** is shown as Fig 7.



```
public static ArrayList<Clause> PLResolve(Clause clause_i, Clause clause_j) throws IOException {
    ArrayList<Clause> resolvents = new ArrayList<>();

    Clause c_i = new Clause(clause_i);
    Clause c_j = new Clause(clause_j);
    ArrayList<Term> ci = c_i.getArrayList();
    ArrayList<Term> cj = c_j.getArrayList();
```

Figure 7: Small sample of PLResolve

2.3 Convert to CNF

2.3.1 Modus Ponens

Before we test the implementation of program, we need translate the sentences into the CNF form. The first one pretty simple. The converted CNF is shown as following. The negated query α is also included.

$$
\begin{gathered}
\mathrm{P} \\
\neg \mathrm{P} \vee \mathrm{Q} \\
\neg \mathrm{Q}
\end{gathered}
$$

2.3.2 Wumpus World(Simple)

For Wumpus World, it is a lit bit complicated. The CNF form include the negated query α is shown as following. The KB also hard coded into the program.

$\neg$P11

$\neg$B11 $\vee$ P12 $\vee$ P21

$\neg$P12 $\vee$ B11

$\neg$P21 $\vee$ B11

$\neg$B21 $\vee$ P11 $\vee$ P22 $\vee$ P31

$\neg$P11 $\vee$ B21

$\neg$P22 $\vee$ B21

$\neg$P31 $\vee$ B21

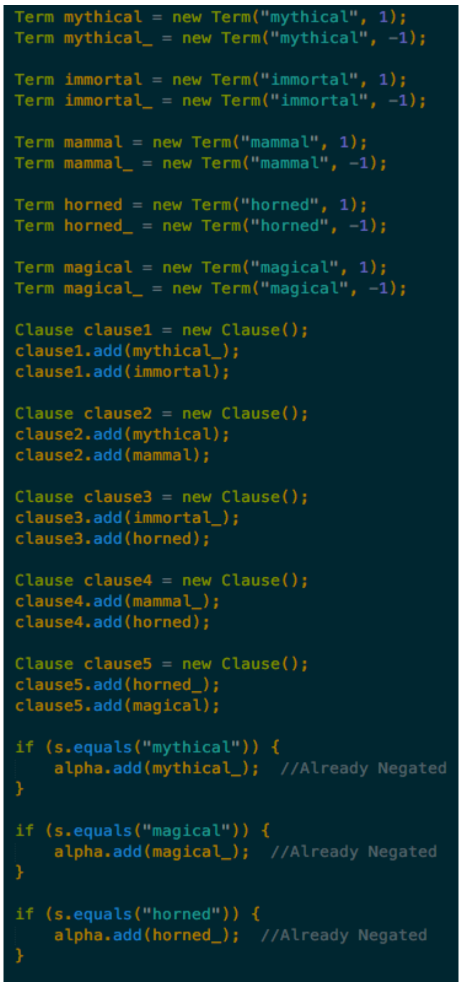
$\neg$B11

B21

$\neg$P12

2.3.3 Horn Clauses

Type the CNF into the LATEX is pretty tedious since I already implemented them in the code. I will use screenshot to represent them, which is shown as Fig 8. As I stated before, in the class **Term**, String contains the symbol and int represent negation of not. Different clauses are label by label1, label2 etc.

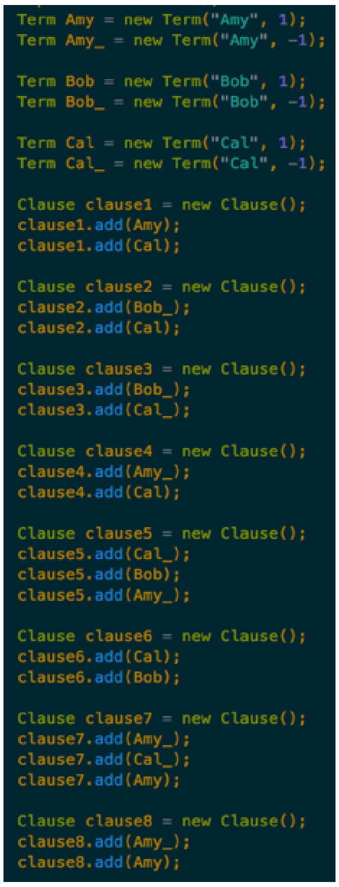


```
Term mythical = new Term("mythical", 1);
Term mythical_ = new Term("mythical", -1);

Term immortal = new Term("immortal", 1);
Term immortal_ = new Term("immortal", -1);

Term mammal = new Term("mammal", 1);
Term mammal_ = new Term("mammal", -1);

Term horned = new Term("horned", 1);
Term horned_ = new Term("horned", -1);

Term magical = new Term("magical", 1);
Term magical_ = new Term("magical", -1);

Clause clause1 = new Clause();
clause1.add(mythical_);
clause1.add(immortal);

Clause clause2 = new Clause();
clause2.add(mythical);
clause2.add(mammal);

Clause clause3 = new Clause();
clause3.add(immortal_);
clause3.add(horned);

Clause clause4 = new Clause();
clause4.add(mammal_);
clause4.add(horned);

Clause clause5 = new Clause();
clause5.add(horned_);
clause5.add(magical);

if (s.equals("mythical")) {
    alpha.add(mythical_);  //Already Negated
}

if (s.equals("magical")) {
    alpha.add(magical_);  //Already Negated
}

if (s.equals("horned")) {
    alpha.add(horned_);  //Already Negated
}
```

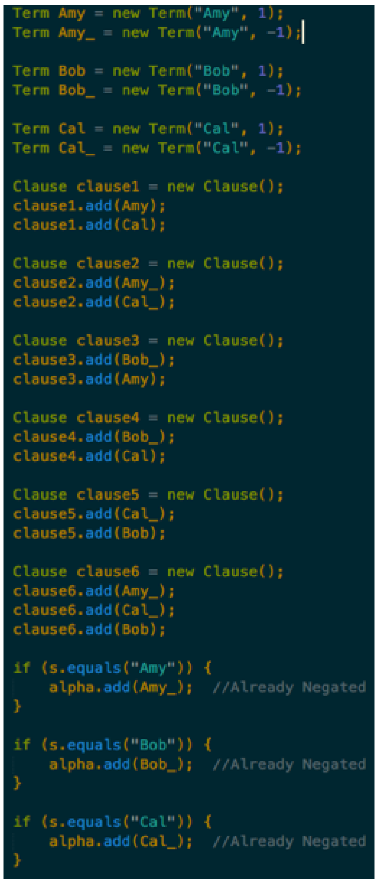
Figure 8: CNF of Horn Clauses

2.3.4 Liars and Truth-tellers

The CNF of Liars and Truth-tellers is shown as Fig 9.



```java
Term Amy = new Term("Amy", 1);
Term Amy_ = new Term("Amy", -1);

Term Bob = new Term("Bob", 1);
Term Bob_ = new Term("Bob", -1);

Term Cal = new Term("Cal", 1);
Term Cal_ = new Term("Cal", -1);

Clause clause1 = new Clause();
clause1.add(Amy);
clause1.add(Cal);

Clause clause2 = new Clause();
clause2.add(Bob_);
clause2.add(Cal);

Clause clause3 = new Clause();
clause3.add(Bob_);
clause3.add(Cal_);

Clause clause4 = new Clause();
clause4.add(Amy_);
clause4.add(Cal);

Clause clause5 = new Clause();
clause5.add(Cal_);
clause5.add(Bob);
clause5.add(Amy_);

Clause clause6 = new Clause();
clause6.add(Cal);
clause6.add(Bob);

Clause clause7 = new Clause();
clause7.add(Amy_);
clause7.add(Cal_);
clause7.add(Amy);

Clause clause8 = new Clause();
clause8.add(Amy_);
clause8.add(Amy);
```

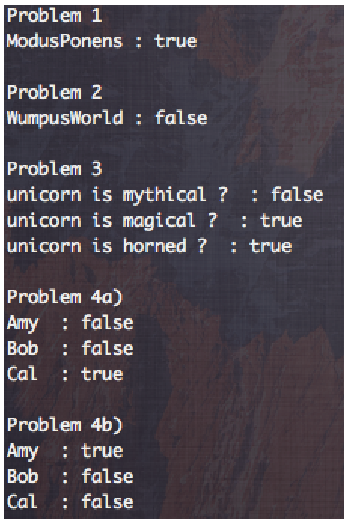


```java
Term Amy = new Term("Amy", 1);
Term Amy_ = new Term("Amy", -1);

Term Bob = new Term("Bob", 1);
Term Bob_ = new Term("Bob", -1);

Term Cal = new Term("Cal", 1);
Term Cal_ = new Term("Cal", -1);

Clause clause1 = new Clause();
clause1.add(Amy);
clause1.add(Cal);

Clause clause2 = new Clause();
clause2.add(Amy_);
clause2.add(Cal_);

Clause clause3 = new Clause();
clause3.add(Bob_);
clause3.add(Amy);

Clause clause4 = new Clause();
clause4.add(Bob_);
clause4.add(Cal);

Clause clause5 = new Clause();
clause5.add(Cal_);
clause5.add(Bob);

Clause clause6 = new Clause();
clause6.add(Amy_);
clause6.add(Cal_);
clause6.add(Bob);

if (s.equals("Amy")) {
    alpha.add(Amy_);  //Already Negated
}

if (s.equals("Bob")) {
    alpha.add(Bob_);  //Already Negated
}

if (s.equals("Cal")) {
    alpha.add(Cal_);  //Already Negated
}
```

Figure 9: CNF of Liars and Truth-tellers

2.4 Results of Implementation

After add all the CNF to the program, run the code the results are shown as Fig 10.

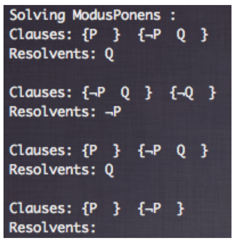


```
Problem 1
ModusPonens : true

Problem 2
WumpusWorld : false

Problem 3
unicorn is mythical ?  : false
unicorn is magical ?  : true
unicorn is horned ?  : true

Problem 4a)
Amy  : false
Bob  : false
Cal  : true

Problem 4b)
Amy  : true
Bob  : false
Cal  : false
```

Figure 10: Resolution for all four tests

2.5 Example of resolution detailed process

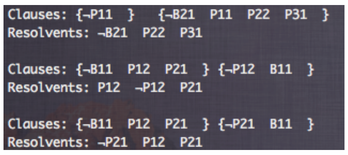
Moreover, I also output the entire resolution process in attached file “CalculationDetails.txt”. Let us analysis several example to see whether the resolve precess is correct or not. The following Fig 11 shows the resolution process of ModusPonens. We can see that in the first step P and $\neg P$ is resolved, left resolvent Q. Next, $\neg Q$ and Q is resolved, left resolvent $\neg P$. In the final step, $\neg P$ resolve with P, the empty set happen, we prove the contradiction. So the query is true. The resolution process is correct.



```
Solving ModusPonens :
Clauses: {P  }  {¬P  Q  }
Resolvents: Q

Clauses: {¬P  Q  }  {¬Q  }
Resolvents: ¬P

Clauses: {P  }  {¬P  Q  }
Resolvents: Q

Clauses: {P  }  {¬P  }
Resolvents:
```

Figure 11: Resolution process of Modus Ponens

Another example I want to show is WumpusWorld, Since It is a long resolution process. I cannot show all the steps. Here only a sample Fig 12 is shown to illustrate the resolution process is correct. In the first step $\neg P11$ was resovled with $P11$, left the resolvent $\neg B21, P22, P21$. In the following two examples, the precedure is the same. The final results is false since we cannot found empty set even if we reach the end state in which no new clause was generated by resolution. We could see the implementation is correct.



```
Clauses: {¬P11  }   {¬B21  P11  P22  P31  }
Resolvents: ¬B21  P22  P31

Clauses: {¬B11  P12  P21  } {¬P12  B11  }
Resolvents: P12  ¬P12  P21

Clauses: {¬B11  P12  P21  } {¬P21  B11  }
Resolvents: ¬P21  P12  P21
```

Figure 12: Resolution process of Wumpus World

As I stated before, the whole resolution process is attached as “CalculationDetails.txt”, re run the program would also generate the file. Readers who interested in the detailed process could take a look at it.

3 Future Improvement

HashSet could improve the program part 2 since the resolvent cannot contain duplicated objects. I didn’t realized it when I start writting the program. I used List and determine whether the object is already inside the list before add them. It is complicated and easily lead to bugs.

Another point is that there should be more elegent way to represent the propositional logial sentence. My implementation is so complicated. I think using Stack to store symbols and Queue to store logical operators might be a way out.