Chen Zhang
CSC 242 Artificial Intelligence
Project 2
Due Date: 03/15/16
No partners

Java Implementation on Basic Model Checking and Resolution Inference

Introduction

The purpose of this project was to implement two different algorithms for A.I to solve some simple logic inference. An inference program is very important to A.I development due to its ability to extend a knowledge base (KB) automatically. Recall that, KB is a set of propositions that represent what the A.I knows about the real world. In the problems of Modus Ponens and Wumpus World, basic model checking was used to solve those two problems. Modus Ponens is a straightforward implication elimination. It can be asserted as “P implies Q, and P is true; then Q can be concluded as true”; formal equation is discussed in basic concepts session. For Horn Clauses problems, resolution inference was applied to solve the problem due to limitation of basic model checking.

Basic Concepts for Model Checking

Model checking is basically enumerating all the possibilities of propositional logic by a given model; then exclusively check this model meets a specification. The formal representation of Modus Ponenes:

$$\{P, P \Rightarrow Q\} \vDash Q \quad \textit{Equation 1}$$

In this case, if we want to enumerate all the possible combinations of P and Q, we need $2^2 = 4$ rows to represent the model checking:

P	Q
True	True
True	False
False	True
False	True

Table 1

Similarly, in Wumpus World, we need 2^7=128 models to enumerate all the possibilities:

P1,1	P1,2	P2,1	P2,2	P3,1	B1,1	B2,1
T	T	T	T	T	T	T
T	T	T	T	T	T	F
T	T	T	T	T	F	F

Table 2, only three combinations are shown in the table. The entire model was printed out in the java program(java WPSolver)

The general states needed for n propositional symbols model checking are 2^n, which is usually unacceptable in computer science. That's one of the important reasons why we need theorem proving, applying rules of inference directly to the sentences in KB to construct a proof in Horn Clauses problems.

Implementation for Model Checking

Since Prof. Ferguson provided his schema for how to implement the model checking, I would like to emphasize my own codes and how I used his codes. Also, some minor revision was applied in original Prof. Ferguson codes for my design. Symbol is basically the class we used to create new propositional symbol. The important method here is isSatisfedBy, which returns the boolean value of the symbol. UnaryCompoundSentence (UCS) and BinaryCompoundSentence (BCS) divide the propositional logics into two categories. For UCS, the only applicable logic symbol is Negation, which extends UCS. In this class, isSatisfedBy returns true if the argument is not satisfied by the model. For BCS, the applicable logic symbols are Conjunction, Disjunction, Implication and Biconditional (if and only if). Those classes are similar but we need pay close attention to their conditions on isSatisfedBy methods. Conjunction is satisfied when two sentences are true, Disjunction is satisfied by a given model when either of the sentence is true, Implication is satisfied when either a sentence is not true or the other sentence is true. Biconditional is satisfied when the two sentences share the same boolean values.

In ModusPonenesKB, symbol "P" and "Q" are the only two symbols needed in this case. Recall that our knowledge base for this: P and $P \Rightarrow Q$, so I added those two sentences in our sentenceList. In the original interface of model, Prof. Ferguson has two methods that check a model satisfies a given model and sentence. The method declaration parameter is KB and sentence respectively. For future convenience, I added additional parameter (model) in the methods declaration.

SolverModel is the class I wrote to implement model. One of the functions of this class is to wire the symbol with a boolean value. I applied HashMap<Symbol, Boolean> in the set method, so the run time of get method is also O(1). Then, I overrode check a KB/sentence is satisfied by a given model. In *satisfies (KB kb, Model model)* method, it loops through the entire sentenceList and check if each sentences is satisfied. If all sentences in sentenceList satisfies the model, then KB is satisfied.

MPProver is the class that calls all the previous written classes together and tries to prove the entailment. The essences of this program is generateTable(), which generates the entire truth table with the corresponding symbol size. The truth table is represented as a 2D array. The row size is 2^n (NN), where n is the size of symbol. Instead of using Math.power class, I used shift 1 left logically by N times. After getting the truth table, a model array was created for contain all the models. Then I assigned each symbol to the corresponding boolean value in each model object (generateAllModels()). The printModels() method was created for generating the table like Table.1 and Table.2. Last, one sentence is proven to be true if the model satisfies the KB and given sentence. To be user-friendly, I also printed out the correct model for the satisfying model.

WPSolver has similar concepts as MPProver, it implements Solver instead of Prover, which requires override *solve(KB kb, Sentence s)* method. This model returns the model that satisfies the entire KB. In my implementation, the second parameter (sentence s) is useless here. The idea behind this method is looping through the entire model array, if one model satisfies the entire KB, return the model. Similarly, I also printed out the correct model for TA-friendly.

Basic Concepts for Resolution Inference

In some practical cases, it's inefficient to enumerate models if the size of propositional logic increases to a large number. Sometimes, searching for proofs can be very efficient because we can ignore the irrelevant propositions. In the horn clauses problem, I applied resolution inference to solve the problem. The procedure of proof by resolution are shown as below (Resolution Refutation):

1. Convert all sentences to CNF
    - Eliminate ⇔, converting $\alpha \Leftrightarrow \beta$ to $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
    - Eliminate ⇒, converting $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:
    - Apply De Morgan law to move negation inwards, $\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$
    - Apply Distribute Law: distributing $\vee$ over $\wedge$ wherever possible
2. Negate the desired conclusion (converted to CNF)
3. Apply resolution rule until either
    - Derive a contradiction
    - Cannot apply any more

Resolution refutation for propositional logic is a complete proof procedure. The above steps means that if we have applied the resolution rule and we can't apply it anymore, then our desired conclusion can’t be proved. It’s guaranteed that we will always either prove false, or run out of possible steps. It’s complete, because it always generates an answer. Furthermore, the process is sound: the answer is always correct.

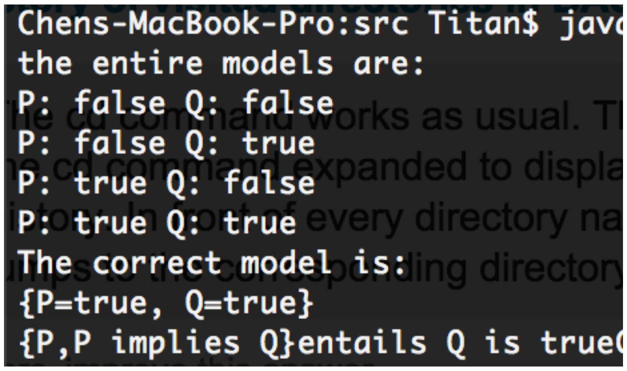
Implementation for Resolution Inference

The two core parts for implementation for resolution inference are converting the clauses into CNF, and applying resolution to CNF. Prof. Ferguson has provided source code for CNF converter. Another significant class in the implementation is Clause class, which implements ArraySet. The Clause class has methods to convert literals (sentences) to a set of clauses. ArraySet is also a class that is provided by Prof. Ferguson, and it not only gives a function of Set but also enables us to add and retrieve elements instead of using iterators.

HornClauseSolver is the class actually solves the problem, assuming CNF converter functions well. First, I defined Symbol: mythical, mortal, mammal, magical and horned as global variables. generateKBSet() method then codes arguments “If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.” into four sentences. After that, CNFConverter is called to convert the KB sentences into clauses. An arraySet<Clause> was used to add all the clauses. *resolution(sentence s)* is the method that determines if the sentence is true or false. First, it calls CNFconverter to convert the negation of the sentence (desired conclusion) and adds it to the KB array set (step 2 in resolution refutation procedure above). Then inside the nested for loop, it first checks two clauses are complementary to each other. If so, perform the resolution methods on the pair of clauses. Once a resolvent contains the empty clause, then a contradiction is said to be found, which return true. Otherwise keep looping though the KB arrayset until no new resolvents are generated. This algorithm was adopted from the book Figure 7.12. Several helper methods were created for resolution(sentence s) method. isComplementary decides whether a pair of clauses has complementary literals among them by checking if their contents are the same and the polarity is opposite. resolve(Clause c1, Clause c2) basically eliminates the complementary literals and return the result of the resolvent clause.

Result:

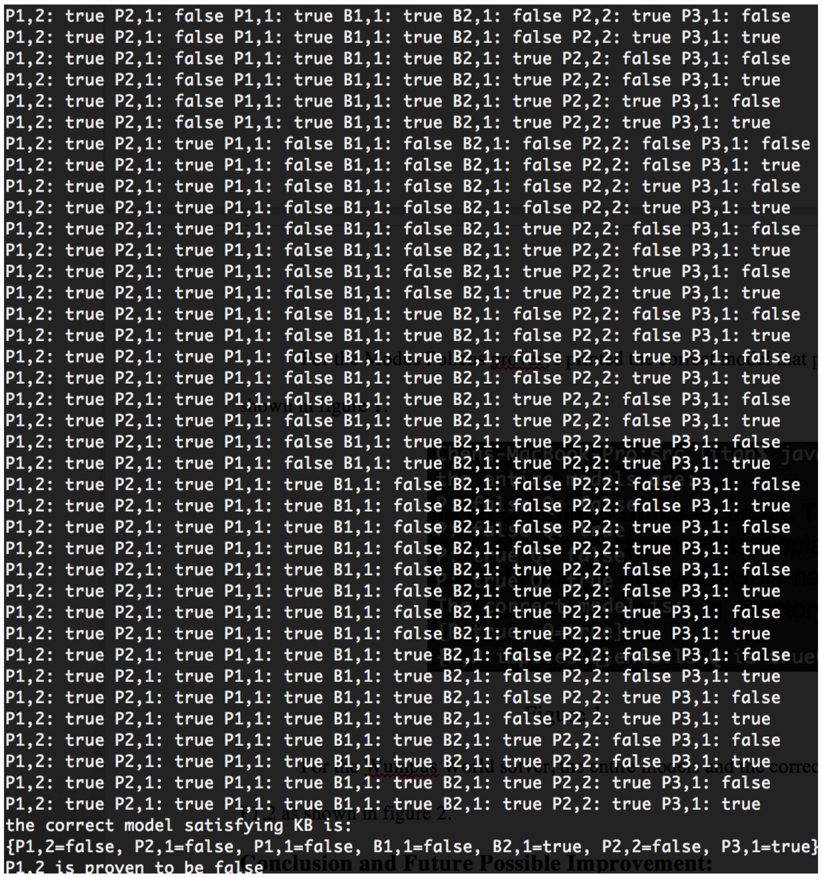
For the Modus Pollens prover, I printed the correct model that proves equation 1 as shown in figure 1.



```
Chens-MacBook-Pro:src Titan$ jav
the entire models are:
P: false Q: false
P: false Q: true
P: true Q: false
P: true Q: true
The correct model is:
{P=true, Q=true}
{P,P implies Q}entails Q is true
```

Figure 1

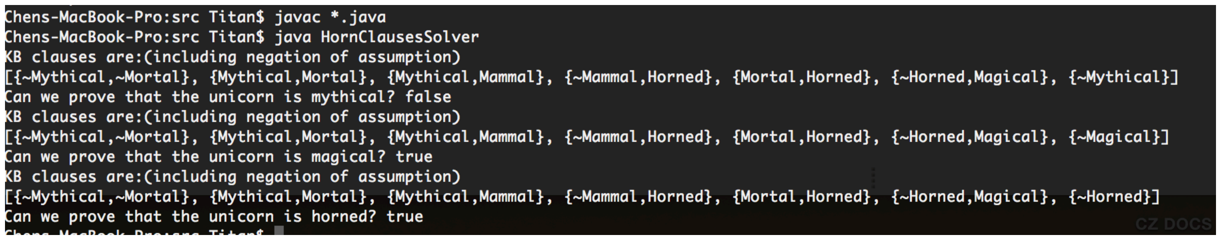
For the Wumpus World solver, the partial of models and the correct model that solves the P1,2 as shown in figure 2. (128 models can be shown when running java WPSolver)



```
P1,2: true P2,1: false P1,1: true B1,1: true B2,1: false P2,2: true P3,1: false
P1,2: true P2,1: false P1,1: true B1,1: true B2,1: false P2,2: true P3,1: true
P1,2: true P2,1: false P1,1: true B1,1: true B2,1: true P2,2: false P3,1: false
P1,2: true P2,1: false P1,1: true B1,1: true B2,1: true P2,2: false P3,1: true
P1,2: true P2,1: false P1,1: true B1,1: true B2,1: true P2,2: true P3,1: false
P1,2: true P2,1: false P1,1: true B1,1: true B2,1: true P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: false P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: false P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: false P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: false P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: true P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: true P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: true P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: false B2,1: true P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: false P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: false P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: false P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: false P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: true P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: true P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: true P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: false B1,1: true B2,1: true P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: false P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: false P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: false P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: false P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: true P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: true P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: true P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: false B2,1: true P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: false P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: false P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: false P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: false P2,2: true P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: true P2,2: false P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: true P2,2: false P3,1: true
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: true P2,2: true P3,1: false
P1,2: true P2,1: true P1,1: true B1,1: true B2,1: true P2,2: true P3,1: true
the correct model satisfying KB is:
{P1,2=false, P2,1=false, P1,1=false, B1,1=false, B2,1=true, P2,2=false, P3,1=true}
P1,2 is proven to be false
```

Figure 2

For horn clauses problems, I printed out the KB clauses and the result of the if the unicorn is mythical/magical/horned. ***If you want to see the entire proving steps, please open HornClausesSolver.java under src folder. Remove the comments in lines 65-67 and line 72. The compiler should print out the new clauses arraySet and resolvents for each step.***



```
Chens-MacBook-Pro:src Titan$ javac *.java
Chens-MacBook-Pro:src Titan$ java HornClausesSolver
KB clauses are:(including negation of assumption)
[{~Mythical,~Mortal}, {Mythical,Mortal}, {Mythical,Mammal}, {~Mammal,Horned}, {Mortal,Horned}, {~Horned,Magical}, {~Mythical}]
Can we prove that the unicorn is mythical? false
KB clauses are:(including negation of assumption)
[{~Mythical,~Mortal}, {Mythical,Mortal}, {Mythical,Mammal}, {~Mammal,Horned}, {Mortal,Horned}, {~Horned,Magical}, {~Magical}]
Can we prove that the unicorn is magical? true
KB clauses are:(including negation of assumption)
[{~Mythical,~Mortal}, {Mythical,Mortal}, {Mythical,Mammal}, {~Mammal,Horned}, {Mortal,Horned}, {~Horned,Magical}, {~Horned}]
Can we prove that the unicorn is horned? true
```

Figure 3

Conclusion:

This project allows me to practice two different algorithms to solve the logic inference by model checking and resolution inference. Still, those problems are very simple and does not involve any first logic order. The basic design schema Prof. Ferguson provided to us is very elegant, especially the array set class, which merges List function and Set function.

Theoretically speaking, resolution inference should be more efficient than model checking because it ignores irrelevant propositional logics. But I found the algorithm in Figure 7.12 from the textbook is not that efficient. The set of clauses in the CNF representation will grow every time if a new resolvent is resolved by a pair of clause. I tried to solve some online propositional logic problem that has a size of 20 arguments by using same algorithm. However, it takes more than 5 mins to solve it. Future improvement should involve how to dynamically remove irrelevant clauses from the set of clauses.