George Rossney
Project 2: Automated Reasoning
CSC 242
3/15/16

Automated Reasoning: Propositional Logic Problems

The goal of this project is to design a basic and an advanced model for implementing propositional logic problems. My goal was to design a program that parses in user input that contains propositional logic in postfix sentences. For part one, my program provides the user with the input format, prompts them for input, then evaluates their input using string comparisons and searching for certain sub-strings. The output should tell the user whether their input is true or false, rather satisfiable or not. For basic model checking, my design includes hard coding in a knowledge base. The knowledge base includes truth table variables (p,q,r) and truth table symbols(v,^,~,->,<->,:-),which stand for *or, and, not, implies, double implies (iff),and entails* respectively. If the knowledge base is correct and includes most possible cases involving logic sentences with the variables and symbols given, then the program should be able to evaluate sentence rules, such as modus ponens. A set of correct basic truth table enumerations should result in more complicated enumerations being correct also. For input, I implemented a basic scanner that accepts sentences in propositional logic. Parsing is difficult and does not exhaust all cases, so I declared certain cases as strings. At the beginning I declare the alphabet for the propositional logic sentences in an *ArrayList* structure as suggested, but I modified my code to use simple string structures. After the user input is accepted, it is evaluated for equivalence against the possible cases to

determine satisfiability. If the user's input is incorrect form or the sentence cannot be evaluated, an error printed statement will be outputted to the console.

For part two, the advanced propositional inference, my goal was to implement the DPLL algorithm for checking satisfiability. As a base, I referred to the algorithm pseudo code provided in the *AIMA* textbook, but I struggled completing an effective Java implementation. I did not attempt to implement the forward-chaining algorithm. Overall, my goal was to use the DPLL algorithm to create a more complicated, but efficient version of the basic logic sentence evaluator from part one to evaluate any of the given sample problems. I looked at the given code for CNF conversion so that I could understand how it works, but I did not modify or include any of the given code. For the DPLL algorithm, I attempted to use the suggested *ArrayList* again to set the negated versions of the *p, q,* and *r*, variables as additional variables themselves. Having negated versions should simplify the evaluation process by narrowing the number of possible comparisons. In order for the DPLL algorithm to work, the input must be in CNF. Once in the correct form, satisfiability should be determined as a result of checking the possible conditions. As backtracking is used thru the logic sentence by evaluating it in a tree-like structure, a true or false result should be determined at each step. My DPLL algorithm is incomplete, but my goal was to create a vocabulary of literals that are pre-evaluated so that the input sentence can be determined as satisfiable or satisfiable in less steps. Hard-coding the common cases, or truth table portions, should lead to increased efficiency and a faster run-time. The worst-case time performance of the DPLL algorithm is $O(2^n)$, which would not be increased by building the program to handle the common cases easier.

The modus ponens rule is a satisfiable sentence in propositional logic than can demonstrated as valid by proving that its three sub-sentences are valid and satisfiable. If my DPLL algorithm was complete, it would be able to parse out the three sub-sentence of modus ponens, evaluate them, then combine the results to prove the rule satisfiable.

For the Wumpus World problem, the *1,2, or 3* coordinates of the *P* and *B* variables can be enumerated as separated variables. There are seven different variables in the given problem, which I declared as variables in my file. I struggled with implementing or solving this problem, but I think that the DPLL algorithm can be used to solve it. At pitfall should be detected when an unsatisfiable phrase is detected in the backtracking algorithm.

The horn clauses problem is represented in English sentences, but can be easily translated or simplified into CNF. I declared each word as a variable and attempted to solve the problem by testing the premises with if-else statements that evaluated conditions in a Boolean method. Using combinations of modus tollens and DeMorgan's law, it can be shown that all three given cases are provable. The unicorn can be mythical, magical, or horned.

Sources Used

1. *Artificial Intelligence: A Modern Approach*, third edition, by Stuart Russell and Peter Norvig.
2. Prof. Ferguson's lecture slides
3. I did not use any of the code from Prof. Ferguson's sample PL folder, but I reviewed each Java source file to help me understand the problems.