Viet Duong

CSC 242

March 17th, 2017

Project 02 Write-up
Automated Reasoning

A. Program Design

My design consists of 5 packages: core, cnf, util, solver and test.

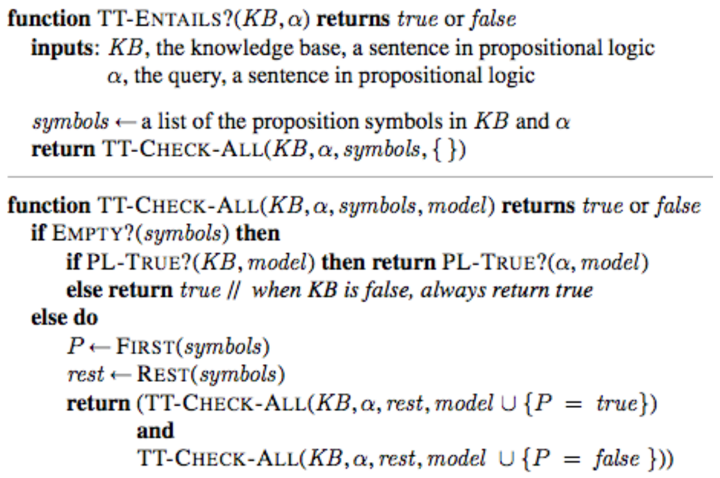
- 'core' package: representation of Propositional Logic sentences in the program using Professor Ferguson's sample code as a foundation, which includes:
  - Symbols: a symbol stands for proposition that is true or false
  - Connectives: enumerations denoting unary connective (NOT) and binary connectives (AND, OR, IMPLIES, IFF) that are used to connect two or more sentences to form Compound Sentences.
  - Compound Sentence: constructed by simpler sentences using connectives, represented by an abstract class that has two children: Unary Compound Sentence and Binary Sentence. Unary Compound Sentence corresponds to Negation, while Binary Compound Sentences consist of Conjunction, Disjunction, Implication and Biconditional.
  - Model: assigns the truth value – true or false – for every propositional symbol. The class Model Implemented includes the implemented methods of the Model interface such as satisfies() for checking if the model satisfies a given Knowledge Base or Sentence and clone() for creating duplicate Models to avoid pointer error, which is necessary for recursive Model Checking algorithm.
  - Knowledge Base: a set of Sentences and a Symbol Table holding the propositional symbols used in those Sentences, that is premises about the world in which we are doing model checking.
- 'cnf' package: paradigm for converting sentences to Conjunctive Normal Form, which is a set of clauses, where a clause is a disjunction of literals, which is used for Theorem Proving by Resolution in the advanced part of the project.

- Clause: includes a method to clone a clause to iterate through its literals when resolving two propositional clauses in Theorem Proving by Resolution.
- Literals: either a Symbol or the negation of a Symbol.
- CNFConvertor: provides a static method for converting a Sentence of propositional logic to conjunctive normal form (implemented by Professor Ferguson).

- ‘util’ package: (provided by Professor Ferguson) a set implementation backed be an ArrayList, which is used in CNFConverter.
- ‘solver’ package: implementations of simple Model Checking and Theorem Proving by Resolution.
- ‘test’ package: test the implementations in ‘solver’ by solving the following problems: Modus Ponens, Simple Wumpus World, Horn Clauses, Liars and Truth-tellers, More Liars and Truth-tellers.

B. Simple Model Checking

For this part of the assignment, I implemented directly the “truth-table enumeration method” described in AIMA Figure 7.10, as shown below.



```
function TT-ENTAILS?($KB, \alpha$) returns true or false
  inputs: $KB$, the knowledge base, a sentence in propositional logic
          $\alpha$, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in $KB$ and $\alpha$
  return TT-CHECK-ALL($KB, \alpha$, symbols, { })

function TT-CHECK-ALL($KB, \alpha$, symbols, model) returns true or false
  if EMPTY?(symbols) then
    if PL-TRUE?($KB$, model) then return PL-TRUE?($\alpha$, model)
    else return true //  when KB is false, always return true
  else do
    $P$ ← FIRST(symbols)
    rest ← REST(symbols)
    return (TT-CHECK-ALL($KB, \alpha$, rest, model $\cup$ {$P$ = true})
            and
            TT-CHECK-ALL($KB, \alpha$, rest, model $\cup$ {$P$ = false }))
```

The implementation is very straight forward. The FIRST/REST method for traversing a list is accomplished by using ArrayList. Obtaining the FIRST(symbols) from ArrayList is simply removing the element with index 0 (symbols.remove(0)) and the resulting ArrayList is REST(symbols). Also, my representation of a Model described in **Program Design** is appropriate for the algorithm, because it includes the satisfies() method to check whether α or KB is true given the values from the model.

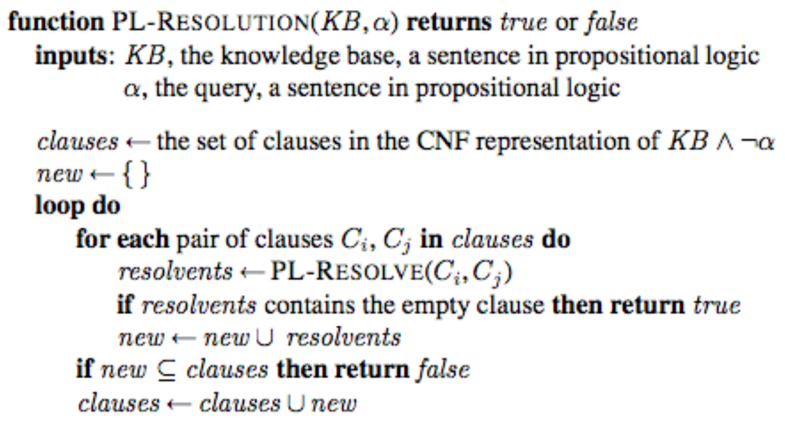
It is important to analyze the soundness and completeness of the algorithm. The algorithm derives directly from the definition of entailment: enumerate the models, in every model in which KB is true, α is also true:

if PL-TRUE?(KB, model) return PL-TRUE?(α,model).

Hence the algorithm is sound. The algorithm is also complete because it works for any KB and α and always terminate because there are finitely many models to check. The time complexity is $O(2^n)$ (α and KB contain n symbols) and the space complexity is $O(n)$.

C. Advanced Propositional Inference

I implemented the resolution algorithm using the pseudocode from AIMA Figure 7.12.



```
function PL-RESOLUTION($KB, \alpha$) returns true or false
   inputs: $KB$, the knowledge base, a sentence in propositional logic
           $\alpha$, the query, a sentence in propositional logic

   clauses ← the set of clauses in the CNF representation of $KB \land \neg\alpha$
   new ← { }
   loop do
      for each pair of clauses $C_i, C_j$ in clauses do
         resolvents ← PL-RESOLVE($C_i, C_j$)
         if resolvents contains the empty clause then return true
         new ← new ∪ resolvents
      if new ⊆ clauses then return false
      clauses ← clauses ∪ new
```

In order to obtain the set of clauses in the CNF representation of KB ∧ ~α, I used the static method CNFConvertor.convert() to convert the positional logic sentences in KB ∧ ~α, which implements the following steps:

- Eliminate Biconditionals
- Eliminate Implications
- Move Negations inwards
- Distribute ORs over AND

The algorithm basically iterates through every pair of clause to compute the list of *resolvents* using PL-RESOLVE(C_i, C_j), that is the set of all possible clauses obtained by resolving C_i, C_j for $i \neq j$. I implement my own PL-RESOLVE(), which follows the pseudocode:

```
function PL_RESOLVE(. , .) returns resolvents
    inputs: a pair of clauses $C_i$, $C_j$
    resolvents ← {}
    for each pair of literals $L_i$ in $C_i$, $L_j$ in $C_j$ do
        if symbol($L_i$) = symbol($L_j$) and polarity($L_i$) $\neq$ polarity($L_j$) then
            $C_i$.remove($L_i$)
            $C_j$.remove($L_j$)
            resolvents.add($C_i$)
            resolvents.add($C_j$)
            return resolvents
    resolvents.add($C_i$)
    resolvents.add($C_j$)
    return resolvents
```

What this function does is to check for any pair of literals from two clauses that have the same symbol but opposite polarity, then remove that from both clauses and add the resulting clauses into the list of *resolvents*. If the clauses could not be resolved, add them to the *resolvents* into the list.
The soundness of the inference rule of resolution algorithm can be shown by deducing truth-table for the clauses in CNF. It is also complete as stated by the ground resolution theorem: If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause (AIMA Section 7.5).

D. Testing

I tested my implementations by solving Modus Ponens, Simple Wumpus World, Horn Clauses, Liars and Truth-tellers, More Liars and Truth-tellers.

As required by the assignment, I solved for Modus Ponens, Simple Wumpus World and Horn Clauses using both algorithms and the results are produced rather quickly and accurate. For Theorem Proving using Resolution, I added print commands in order to print out step-by-step proof by resolution (the sets of clauses and resolvents). Because printing all the steps requires a lot of space, I had to comment those out and include the proofs in 3 text files for better illustration of the algorithm.

I also solved Liars and Truth-tellers and More Liars and Truth-tellers for extra-credit. The resolution algorithm works fine for the Liars and Truth-Tellers, but could not produce any result for the More Liars and Truth-tellers. This shows the depreciation in efficiency of the resolution algorithm when more complex sentences are added to the model.

E. Conclusions

This project has allowed me to learn a great deal about the propositional logic “language”. Although my program has shown some difficulty in tackling extremely complex problems, I believe it is certainly adequate for solving basic and intermediate logical problems accurately in reasonable runtime.

Instructions to compile my code are included in a README.txt and there are also 3 step-by-step proofs by resolution for Modus Ponens, Simple Wumpus World and Horn Clauses in .txt files.