Master's Degree in Electrical and Computer Engineering

Propositional Logic Reasoner

Artificial Intelligence and Decision Systems - Assignment #2

Authors: Luís Rei (78486), João Girão (78761)

December 21, 2017

...

1 Problem Formulation

One of the most crucial aspects of an intelligent agent is the ability to represent the world and reasoning about it when determining the actions to be executed. In this report we will guide you through our solution for this problem obtained using a knowledge base with information on the world and an inference engine. To build the knowledge base a logical approach (namely propositional logic), composed of both an epistemic (semantics) and a logical (syntax) levels, was chosen, as it allows the user to draw irrefutable conclusions from a set of premises. For the inference engine we used the resolution inference system on sentences in Conjunctive Normal Form (CNF). The assignment consisted on the development of two programs that together formed the basis for a propositional logic reasoner.

2 CNF Converter

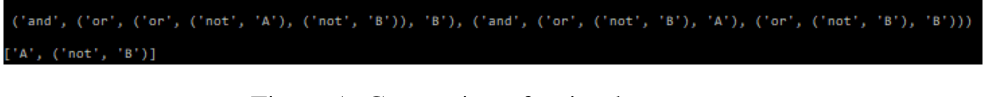
This first program, "convert.py", has as its purpose the proper conversion of a sentence written in propositional logic to its equivalent in CNF. When calling it without input files the user is able to insert whatever sentences he/she wishes as long as it is one of the acceptable formats. All sentences not in these formats will be ignored.

Accepted formats for the input are:

- Propositional logic - as defined in the assignment sheet;
- CNF - a set of conjunctions of disjunctions (both also as defined in the assignment sheet);
- Simplified CNF - clauses composed of one or more literals.

In the first two cases, sentences are received one by line, with a sentence containing a literal (string delimited by apostrophes or quote marks or negation of a string) or a closed proposition (in accordance to the guide). In the last case, each line should have a clause (disjunction of literals) in the following form: [literal 1, literal 2, ... , literal n]. To finish building the Knowledge Base just press the "Enter" key without anything before.

The program goes through the syntactic tree, converting each branch iteratively into CNF. After this is done we are left with a sentence that is a set of conjunctions of disjunctions, which is then simplified by removing tautologies and factoring it. The simplified knowledge base is then outputted, one clause at a time, as shown in the example below.



```
('<=>', ('and', 'A', 'B'), 'B')
('and', ('or', ('or', ('not', 'A'), ('not', 'B')), 'B'), ('and', ('or', ('not', 'B'), 'A'), ('or', ('not', 'B'), 'B')))
['A', ('not', 'B')]
```

Figure 1: Conversion of a simple sentence

In the example above we can see the different stages of the algorithm. The input is given by the first line and corresponds to the sentence $(A \wedge B) \Leftrightarrow B$, which can be seen in CNF in the second line: $(\neg A \vee \neg B \vee B) \wedge (\neg B \vee A) \wedge (\neg B \vee B)$. After applying the simplifications mentioned above and eliminating redundancies such as the disjunctions of two complementary literals, the remaining clauses are inserted in the knowledge base: $(A \vee \neg B)$.

3 Theorem Prover

The second program, ”prove.py”, receives a knowledge base KB and the negation of the sentence α to be proven, all in simplified CNF, and outputs either ’True’ or ’False’ if α can or can’t be inferred given KB, respectively.

For optimization purposes, further simplifications to the knowledge base are made, as well as refactoring and removing tautologies from it, since the user can input directly to the program a non simplified version of KB. The new simplifications are the removal of clauses containing literals that aren’t complemented by at least one other in the rest of the KB and the elimination of supersets of clauses in the KB (if c1 is a subset of c2 then c2 is removed from KB).

As in the previous program, the end condition of a call for ”prove.py” without files is the pressing of the ”Enter” key without anything before.

After simplifying the knowledge base and ordering it both alphabetically and by size in ascending order, the program takes the first clause in the KB and applies the resolution method with the next clause in line. After every application, the new clauses are added, if they haven’t been already, and the KB is ordered again to facilitate future applications of the method. If an empty clause is originated as a result of the present iteration, the algorithm will stop and return ”True”. To optimize the algorithm, a list containing the pairs of clauses on which the method had already been used was created, so that only new pairs of clauses are considered when choosing the inputs for the resolution algorithm.

If after every pair of clauses has been considered there is no empty clause, the program will output ”False” and come to an end.

4 Full Reasoner Implementation

For a more intuitive use of the reasoner it is advised to call both programs in tandem through the execution of the following commands:

- **python3 convert.py | python3 prove.py** - for inputing sentences directly from the console;
- **python3 convert.py < file.txt | python3 prove.py** - for building the knowledge base with the content in *file.txt*

This way, the user can define the sentences in propositional logic, CNF or simplified CNF and understand easily the end result ’True’/’False’.