Artificial Intelligence and Decision Systems
Propositional Logic Reasoner
Assignment II

Pedro Ferreira - 75263 Rúben Tadeia - 75268

Group 14

1 Formulation of the problem

The goal of this project is to build a propositional logic reasoner, based on the resolution principle. Its key components are a program to convert sentences in propositional logic to clausal normal form (CNF) and a resolution-based theorem prover for a CNF knowledge base.

2 CNF Converter

The goal of the CNF converter is to convert sentences in propositional logic to clausal normal form (conjunctions of disjunctions). The first step to tackle this problem is to parse the input file and store each sentence as an element of a list. This is done with function **parse_file()**. Then, for each sentence of that list:

1. Create a binary tree of the sentence (as may be seen in figure 1 for an entry of ('<=>', 'A', ('and', 'B', 'C')));
2. Recursively eliminate equivalences;
3. Recursively eliminate implications;
4. Recursively move negations inwards in order to appear in literals only;
5. Recursively apply the distributive law, in order to obtain conjunctions of disjunctions.

Creating the tree of the sentence is done when instantiating the object, and each of the recursive steps is done with function **apply_recursion()** by passing the pretended step as input. For instance, removing the equivalences will be done by passing as inputs *(tree, '<=>')*. At the end of this steps, function **get_clauses()** is called. This function prints all the clauses for a given sentence and takes into consideration two simplifications. First, if it is detected that a clause has a tautology, the clause is not printed to the output of the system. Second, literals that appear more than once in the same clause are removed (factoring). It's worth noting that two other common simplifications were not, unfortunately, implemented. It's also worth noting that the there are issues with the implementation of the distributive part. Since it is not prepared to deal with cases of several nested ANDs and ORs, some illegal trees might be built, and those correspondent clauses will not be printed. All of the code regarding this part may be seen in *convert.py*, with the class *Tree* implemented in *classes.py*.

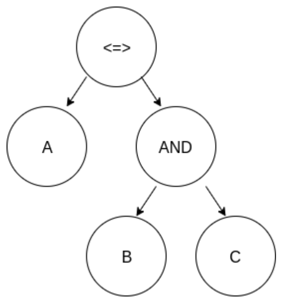

Figure 1: Tree for ('<=>', 'A', ('and', 'B', 'C'))

3 Theorem Prover

In this section a sentence α has to be proved given a knowledge base, KB, of facts. Depending on whether α could be proved given KB, $KB \vdash \alpha$. Since the algorithm for the Resolution Inference System is a proof by contradiction of $KB \models \alpha$, this means that if any set of sentence is unsatisfiable (they are false, meaning that they aren't satisfied by any model), then the resolution method will output True.

This problem was solved taking into account the **resolution inference method**, the **factoring rules** and the **unit preference strategy**. To prove the model we used the generic resolution inference system algorithm. So, in order to begin, lets first assume $C = KB \cup \{\neg\alpha\}$.

Algorithm 1 Resolution Inference System

```
1:  function RESOLUTION($KB, \neg\alpha$)
2:      while no new clauses result from any pair of clauses in $C$ do
3:          while $(C_i \; and \; C_j) \; in \; C$ do
4:              $result = Resolution(C_i, C_j)$.
5:              if result $\supset \{\}$ then
6:                  return $True$
7:              else
8:                  return $False$
9:              end if
10:             $C \leftarrow C \cup result$.
11:         end while
12:     end while
13: end function
```

For this part of the project, the code is structured as it follows:

1. **prover.py**
2. **resolution.py**

The first file deals with the input file. The second file includes the class *Resolution* and is responsible for the resolution algorithm and every comparison between literals.

After the development of the algorithm, which pseudocode may be seen in 1, a script was created to solve every input file and obtain the results. These results may be seen in table 1. It is worth stressing that these results were obtained by using just *prover.py*, and not the pipe with *converter.py*. Using the pipe not all the results obtained are the same, since some outputs of the converter to cnf are not correct.

Table 1: Results of the given input files

File	*trivial.txt*	*sentences.txt*	*p1.txt*	*p2.txt*	*p3.txt*	*p4.txt*
Output	True	False	False	False	False	True

Some of these problems could become complex and take more time to solve. Another approach for those cases would be to use a SAT solver like DPLL (Davis-Putnam-Logemann-Loveland) algorithm, to check whether it is satisfiable, *e.g.*, i.e., if there is a model that makes the sentence true. This algorithm is very often used as a SAT problem solver since it has a worst-case exponential run time.