CS 4365 Artificial Intelligence
Assignment 3: Knowledge Representation & Reasoning

Part I: Programming (100 points)

In this problem you will be implementing a theorem prover for a clause logic using the resolution principle. Well-formed sentences in this logic are clauses. As mentioned in class, instead of using the implicative form, we will be using the disjunctive form, since this form is more suitable for automatic manipulation. The syntax of sentences in the clause logic is thus:

$$\begin{array}{rcl} Clause & \rightarrow & Literal \vee \ldots \vee Literal \\ Literal & \rightarrow & \neg Atom \mid Atom \\ Atom & \rightarrow & \textbf{True} \mid \textbf{False} \mid P \mid Q \mid \ldots \end{array}$$

We will regard two clauses as identical if they have the same literals. For example, $q \vee \neg p \vee q$, $q \vee \neg p$, and $\neg p \vee q$ are equivalent for our purposes. For this reason, we adopt a standardized representation of clauses, with duplicated literals always eliminated.

When modeling real domains, clauses are often written in the form:

$$Literal \wedge \ldots \wedge Literal \Rightarrow Literal$$

In this case, we need to transform the clauses such that they conform to the syntax of the clause logic. This can always be done using the following simple rules:

1. $(p \Rightarrow q)$ is equivalent to $(\neg p \vee q)$
2. $(\neg(p \vee q))$ is equivalent to $(\neg p \wedge \neg q)$
3. $(\neg(p \wedge q))$ is equivalent to $(\neg p \vee \neg q)$
4. $((p \wedge q) \wedge \ldots)$ is equivalent to $(p \wedge q \wedge \ldots)$
5. $((p \vee q) \vee \ldots)$ is equivalent to $(p \vee q \vee \ldots)$
6. $(\neg(\neg p))$ is equivalent to p

The proof theory of the clause logic contains only the resolution rule:

$$\frac{\begin{array}{c}\neg a \vee l_1 \vee \ldots \vee l_n, \\ a \vee L_1 \vee \ldots \vee L_m\end{array}}{l_1 \vee \ldots \vee l_n \vee L_1 \vee \ldots \vee L_m}$$

If there are no literals $l_1, \ldots l_n$ and $L_1, \ldots, L_m$, the resolution rule has the form:

$$\frac{\neg a, a}{\textbf{False}}$$

Remember that inference rules are used to generate new valid sentences, given that a set of old sentences are valid. For the clause logic this means that we can use the resolution rule to generate new valid clauses given a set of valid clauses. Consider a simple example where $p \Rightarrow q$, $z \Rightarrow y$ and p are valid clauses. To prove that q is a valid clause we first need to rewrite the rules to disjunctive form: $\neg p \vee q$, $\neg z \vee y$ and p. Resolution is then applied to the first and last clause, and we get:

$$\frac{\neg p \vee q, p}{q}$$

If **False** can be deduced by resolution, the original set of clauses is inconsistent. When making proofs by contradiction this is exactly what we want to do. The approach is illustrated by the resolution principle explained below.

The Resolution Principle

To prove that a clause is valid using the resolution method, we attempt to show that the negation of the clause is *unsatisfiable*, meaning it cannot be true under any truth assignment. This is done using the following algorithm:

1. Negate the clause and add each literal in the resulting conjunction of literals to the set of clauses already known to be valid.
2. Find two clauses for which the resolution rule can be applied. Change the form of the produced clause to the standard form and add it to the set of valid clauses.
3. Repeat 2 until **False** is produced, or until no new clauses can be produced. If no new clauses can be produced, report failure; the original clause is not valid. If **False** is produced, report success; the original clause is valid.

Consider again our example. Assume we now want to prove that $\neg z \vee y$ is valid. First, we negate the clause and get $z \wedge \neg y$. Then each literal is added to the set of valid clauses (see 4. and 5.). The resulting set of clauses is:

1. $\neg p \vee q$
2. $\neg z \vee y$
3. p
4. z
5. $\neg y$

Resolution on 2. and 5. gives:

1. $\neg p \vee q$
2. $\neg z \vee y$

3. p

4. z

5. $\neg y$

6. $\neg z$

Finally, we apply the resolution rule on 4. and 6. which produces **False**. Thus, the original clause $\neg z \vee y$ is valid.

(A) The Program

Your program should read a text file containing the initial set of valid clauses and a clause for each literal in the negated clause that we want to test validity of. Each line in the file defines a single clause. The literals of each clause are separated by a blank space and $\sim$ is used to represent negation.

Your program should implement the resolution algorithm as explained in the previous section. The output is either "Failure" if the clause cannot be shown to be valid, or the list of clauses in the proof tree for deducing *False*. In either case you should also return the size of the final set of valid clauses.

Let us consider a correct solution for testing the validity of $\neg z \vee y$ for our example. The input file would be:

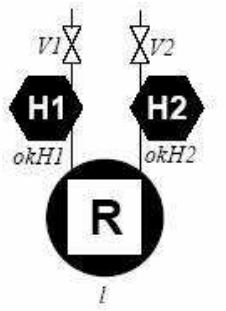
```
~p q
~z y
p
z
~y
```

A possible final set of valid sentences could be:

```
1.  ~p q   {}
2.  ~z y   {}
3.  p      {}
4.  z      {}
5.  ~y     {}
6.  ~z     {2,5}
7.  False  {4,6}
```

Note how the program keeps track of the parents of new clauses. This is used for extracting the clauses in the proof tree. The solution you return consists of these clauses and the size of the final set of valid clauses:

```
2.  ~z y   {}
4.  z      {}
5.  ~y     {}
6.  ~z     {2,5}
7.  False  {4,6}
Size of final clause set:  7
```



(B) Power Plant Diagnosis

In the last part of this assignment you will be using your resolution prover to verify the safety requirements of a reactor unit in a nuclear power plant. The reactor unit is shown in the figure on the next page and consists of a reactor R, two heat exchangers $H1$ and $H2$, two steam valves $V1$ and $V2$, and a control stick l for changing the level of energy production. The state of the reactor unit is given by 5 propositional variables l, $okH1$, $okH2$, $V1$ and $V2$. If l has the value **True** the production level is 2 energy units. Otherwise, the production level is 1 energy unit. At least one working heat exchanger is necessary for each energy unit produced to keep the reactor from overheating. Unfortunately a heat exchanger i can start leaking reactive water from the internal cooling system to the surroundings. $okHi$ is **False** if heat exchanger Hi is leaking. Otherwise, $okHi$ is **True**. When a heat exchanger i is leaking, it must be shut off by closing its valve Vi. The state variable Vi indicates whether the valve Vi is closed (**False**) or open (**True**). Formally, the safety requirements are described by the following clauses:

$$
\begin{aligned}
NoLeak \wedge LowTemp &\Rightarrow ReactorUnitSafe \\
NoLeakH1 \wedge NoLeakH2 &\Rightarrow NoLeak \\
okH1 &\Rightarrow NoLeakH1 \\
\neg okH1 \wedge \neg V1 &\Rightarrow NoLeakH1 \\
okH2 &\Rightarrow NoLeakH2 \\
\neg okH2 \wedge \neg V2 &\Rightarrow NoLeakH2 \\
l \wedge V1 \wedge V2 &\Rightarrow LowTemp \\
\neg l \wedge V1 &\Rightarrow LowTemp \\
\neg l \wedge V2 &\Rightarrow LowTemp
\end{aligned}
$$

Assume that the current state of the reactor unit is given by the clauses:

$$
\begin{aligned}
&\neg l \\
&\neg okH2 \\
&okH1 \\
&V1 \\
&\neg V2
\end{aligned}
$$

1. Rewrite the safely rules from their implicative form to the disjunctive form used by your resolution prover. The initial set of valid clauses is the union of the rule clauses and the clauses defining the current state. Write the clauses in a file called `facts.txt`.

2. Use your resolution prover to test whether $LowTemp$ is a valid clause:

   (a) Save the input in a file called `task1.in`.

   (b) Test the result of your prover.

3. Now test the validity of $ReactorUnitSafe$ in a similar way:

   (a) Save the input in a file called `task2.in`.

   (b) Test the result of your prover.

4. Consider a simpler set of safety rules:

$$\begin{aligned} NoLeakH1 \wedge NoLeakH2 &\Rightarrow NoLeak \\ okH1 &\Rightarrow NoLeakH1 \\ \neg okH1 \wedge \neg V1 &\Rightarrow NoLeakH1 \\ okH2 &\Rightarrow NoLeakH2 \\ \neg okH2 \wedge \neg V2 &\Rightarrow NoLeakH2 \end{aligned}$$

   and a reduced current state description:

$$\begin{aligned} &\neg okH2 \\ &okH1 \\ &\neg V2 \end{aligned}$$

   Test the validity of $\neg NoLeak$:

   (a) Save the input in a file `task3.in`.

   (b) Test the result of your prover.

Implementation and Submission Requirements

The program must be written in C/C++, Java (JDK 8), or Python, and needs to be run/compile on the UTD Linux boxes without installing extra software.

You may submit as many source files as needed, but you must make sure your provide a main code entry that follows the following naming convention:

- Java: Main.java
- Python: main.py
- C: main.c
- C++: main.cpp

Your code should not use any external libraries.

Submission

You must directly submit all your source files, task 1, 2, and 3 .in files, and the facts.txt file to eLearning (do not put them in a folder or zip file). Any program that does not conform to this specification will receive no credit.

Grading

Make sure you follow the formatting from the example test files: be careful not to insert extra lines, tabs instead of spaces, etc ... When you submit, your code will be **graded using hidden test cases**, so we encourage you to test your code thoroughly.

Important: Be mindful of the efficiency of your implementation; as the test cases we will use are quite long, poorly written code might time out (and receive no credit!). We will provide you with example input files and approximate timings for each, so you can get an idea of how fast your code is.