CS 4365 Artificial Intelligence

Assignment 3: Knowledge Representation & Reasoning

Due: Friday, Nov 11^{th}

Instructions:

1. Your solution to this assignment must be submitted via eLearning.
2. For the written problems, submit your solution as a **single PDF** file.
   - Only use **blue or black pen** (black is preferred). Scan your PDF using a scanner and upload it. Make sure your final PDF is **legible**. Regrades due to non-compliance will receive a 30% score penalty.
   - Verify that both your answers and procedure are **correct, ordered, clean, and self-explanatory** before writing. Please ask yourself the following questions before submitting:
     - Are my answers and procedure legible?
     - Are my answers and procedure in the same order as they were presented in the assignment? Do they follow the specified notation?
     - Are there any corrections or scratched out parts that reflect negatively on my work?
     - Can my work be easily understood by someone else? Did I properly define variables or functions that I am using? Can the different steps of my development of a problem be easily identified, followed, and understood by someone else? Are there any gaps in my development of the problem that need any sort of justification (be it calculations or a written explanation)? Is it clear how I arrived to each and every result in my procedure and final answers? Could someone describe my submission as messy?
3. **You may work individually or in a group of two**. Only one submission should be made per group. If you work in a group, **make sure to indicate both group members when submitting through eLearning**.
4. **IMPORTANT**: As long as you follow these guidelines, your submission should be in good shape; if not, we reserve the right to penalize answers and/or submissions as we see fit.

Part I: Written Problems (100 points)

0.1 Representing Sentences in First-Order Logic (32 points)

Assume that you are given the following predicates, where the universe of discourse consists of all the students in your class

- $I(x)$: x has an Internet connection
- $C(x, y)$: x and y have chatted over the Internet

Write the following statements using these predicates and any needed quantifiers.

1. Exactly one student in your class has an Internet connection.
2. Everyone except one student in your class has an Internet connection.
3. Everyone in your class with an Internet connection has chatted over the Internet with at least one other student in your class.
4. Someone in your class has an Internet connection but has not chatted with anyone else in your class.
5. There are two students in your class who have not chatted with each other over the Internet.
6. There is a student in your class who has chatted with everyone in your class over the Internet.
7. There are at least two students in your class who have not chatted with the same person in your class.
8. There are two students in the class who have chatted with everyone else in the class.

0.2 Validity and Satisfiability (8 points)

For each of the sentences below, decide whether it is valid, satisfiable, or neither. Verify your decisions using truth tables or the Boolean equivalence rules.

1. $Big \vee Dumb \vee (Big \Rightarrow Dumb)$
2. $(Smoke \Rightarrow Fire) \Rightarrow ((Smoke \wedge Heat) \Rightarrow Fire)$

0.3 Models (12 points)

Consider a world in which there are only four proposition, A, B, C, and D. How many models are there for the following sentences? Justify your answer.

1. $(A \wedge B) \vee (B \wedge C)$
2. $A \vee B$
3. $A \Leftrightarrow B \Leftrightarrow C$

0.4 Unification (12 points)

Compute the **most general unifier** (mgu) for each of the following pairs of atomic sentences, or explain why no mgu exists. (Recall that lowercase letters denote variables, and uppercase letters denote constants.)

1. $Q(y, Gee(A, B)), Q(Gee(x, x), y)$.

2. $Older(Father(y), y), Older(Father(x), John)$.
3. $Knows(Father(y), y), Knows(x, x)$.

0.5 Inference in First-Order Logic (36 points)

Consider the following statements in English.

1. Every child loves every candy.
2. Anyone who loves some candy is not a nutrition fanatic.
3. Anyone who eats any pumpkin is a nutrition fanatic.
4. Anyone who buys any pumpkin either carves it or eats it.
5. John buys a pumpkin.
6. Lifesavers is a candy.

Assume that you are given the following predicates:

1. $Child(x)$ — x is a child.
2. $Fanatic(x)$ — x is a nutrition fanatic.
3. $Candy(y)$ — y is a candy.
4. $Pumpkin(y)$ — y is a pumpkin.
5. $Carves(x, y)$ — x carves y.
6. $Eats(x, y)$ — x eats y.
7. $Buys(x, y)$ — x buys y.
8. $Loves(x, y)$ — x loves y.

(a) **(12 pts)** Translate the six statements above into first-order logic using these predicates.

(b) **(12 pts)** Convert the six sentences in part (a) into CNF.

(c) **(12 pts)** Prove using resolution by refutation that *If John is a child, then John carves some pumpkin.* Show any unifiers required for the resolution. Be sure to provide a clear numbering of the sentences in the knowledge base and indicate which sentences are involved in each step of the proof.

Part II: 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:

$$Clause \rightarrow Literal \vee ... \vee Literal \tag{1}$$
$$Literal \rightarrow \neg Atom | Atom \tag{2}$$
$$Atom \rightarrow \mathbf{True} | \mathbf{False} | P | Q | ... \tag{3}$$

We will regard two clauses as identical if they have the same literals. For example, $q \vee \neg p \vee q$, $q \vee \neg p$, $\neg p \vee q$ 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:

$$Literal \wedge ... \wedge Literal \Rightarrow Literal \tag{4}$$

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. $((p \wedge q) \wedge ...)$ is equivalent to $(p \wedge q \wedge ...)$
5. $((p \vee q) \vee ...)$ is equivalent to $(p \vee q \vee ...)$
6. $(\neg(\neg p))$ is equivalent to p

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

$$\frac{\neg a \vee l_1 \vee ... \vee l_n, a \vee L_1 \vee ... \vee L_m}{l_1 \vee ... \vee l_n \vee L_1 \vee ... \vee L_m} \tag{5}$$

If there are no literals $l_1, l_2, ..., l_n$ and $L_1, L_2, ..., L_m$, the resolution rule has the form:

$$\frac{\neg a, a}{\mathbf{False}} \tag{6}$$

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:

$$\frac{\neg p \vee q, p}{q} \tag{7}$$

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.

0.6 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.

0.7 The Program

0.7.1 Files and Task Description

Your program should take exactly one argument from the command line:

1. A `.kb` file that contains the initial KB and the clause whose validity we want to test. The input file contains n lines organized as follows: the first $n-1$ lines describe the initial KB, while line n contains the (original) clause to test. Note that the KB is written in CNF, so each clause represents a disjunction of literals. The literals of each clause are separated by a blank space, while negated variables are prefixed by $\sim$.

Your program should adhere to the following policy:

- If the negated version of the clause to validate has ANDs, your program should split it into separate clauses. These clauses should be added to the KB from left to right order.
- Resolution should proceed as follows: For each clause $i[1, n]$ (where n is the last clause in the KB), attempt to resolve clause i with every previous clause $j[1, i)$ (in order). If a new clause is generated, it is added to the end of the KB (therefore the value of n changes). Your system should continue trying to resolve the next clause $(i+1)$ with all previous clauses until 1) a contradiction is found (in which case 'Contradiction' should be added to the KB) or 2) all possible resolutions have been performed.
- Redundant generated clauses should not be added to the KB. A clause is redundant if the KB contains another clause which is logically equivalent to it.
- Clauses that evaluate to True should not be added to the KB.
- Generated clauses should not have redundant (repeated) literals.

0.7.2 Requirements: Output

Your program should implement the resolution algorithm as explained in the previous section. Your program should output a line for every clause in the final KB (in the order they were added), each line should be single-space-separated and contain: 1) the clause number followed by a period (starting from 1), 2) the clause in DNF, and 3) the parent clauses (if this clause was generated through resolution) written as $\{i, j\}$. Finally, your program should print a final line containing the word $Valid$ or $Fail$ depending on whether the proof by contradiction succeeded or not.

Let us consider a correct solution for testing the validity of $\neg z \vee y$, given the input:

$\sim p\ q$

$\sim z\ y$

p

$\sim z\ y$

Your program's output should be:

1. $\sim p\ q$ {}
2. $\sim z\ y$ {}
3. p {}
4. z {}
5. $\sim y$ {}
6. q {3,1}
7. y {4,2}
8. $\sim z$ {5,2}
9. Contradiction {5,7}

Valid

0.7.3 Implementation

To implement the program, you should use Python 3. You **may not** use external libraries. Your code will be run on a Linux distribution, so you should make sure that it runs/compiles on the UTD Linux Server (see `https://cs.utdallas.edu/about/computing-facilities/`). If you have any questions, please send an email to the staff.

0.7.4 Submission Requirements

Directly submit all your source files as a zip file to eLearning. Note that **you must provide an entry point main.py** for your code to run (these are the only files that we will directly run when grading). If you worked in a group, make sure to add your partner when you submit! Any program that does not conform to this specification will receive no credit.

0.7.5 Example Files

We will provide the corresponding *example_in.txt* and *example_out.txt* for one test case.

0.7.6 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.

0.7.7 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!).