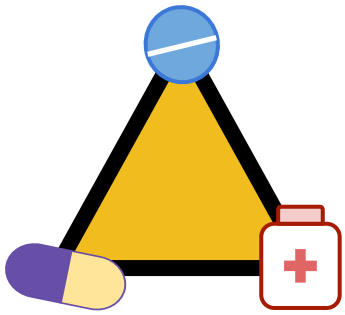
CSCI-561 - Fall 2019 - Foundations of Artificial Intelligence
Homework 3

Due November 25, 2019 23:59:59



Guidelines

This is a programming assignment. You will be provided with sample inputs and outputs (see below). Please understand that the goal of the samples is to check that you can correctly parse the problem definition and generate a correctly formatted output. The samples are very simple and it should not be assumed that if your program works on the samples it will work on all test cases. There will be more complex test cases and it is your task to make sure that your program will work correctly on any valid input. You are encouraged to try your own test cases to check how your program would behave in some complex special case that you might think of. Since **each homework is checked via an automated A.I. script**, your output should match the example format *exactly*. Failure to do so will most certainly cost some points. The output format is simple and examples are provided. You should upload and test your code on vocareum.com, and you will submit it there. You may use any of the programming languages provided by vocareum.com.

Grading

Your code will be tested as follows: Your program should take no command-line arguments. It should read a text file called "input.txt" in the current directory that contains a problem definition. It should write a file "output.txt" with your solution. Format for files input.txt and output.txt is specified below. End-of-line convention is Unix (since vocareum is a Unix system).

The grading A.I. script will, 50 times:

- Create an input.txt file, delete any old output.txt file.
- Run your code.
- Compare output.txt created by your program with the correct one.
- If your outputs for all 50 test cases are correct, you get 100 points.
- If one or more test case fails, you **lose 2 points for each failed test case**. (note that one test case involves several answers in this HW; if any answer on a given test case is wrong, then the whole case is counted as wrong).

Note that if your code does not compile, or somehow fails to load and parse input.txt, or writes an incorrectly formatted output.txt, or no output.txt at all, or OuTpUt.TxT, **you will get zero points**. Please test your program with the provided sample files to avoid this. You can submit code as many times as you wish on vocareum, and the last submitted version will be used for grading.

Academic Honesty and Integrity

All homework material is checked vigorously for dishonesty using several methods. All detected violations of academic honesty are forwarded to the Office of Student Judicial Affairs. To be safe you are urged to err on the side of caution. Do not copy work from another student or off the web. Sanctions for dishonesty are reflected in *your permanent record* and can negatively impact your future success. As a general guide:

> **Do not copy** code or written material from another student. Even single lines of code should not be copied.
> **Do not collaborate** on this assignment. The assignment is to be solved individually.
> **Do not copy** code off the web. This is easier to detect than you may think.
> **Do not share** any custom test cases you may create to check your program's behavior in more complex scenarios than the simplistic ones considered below.
> **Do not copy** code from past students. We keep copies of past work to check for this. Even though this problem differs from those of previous years, do not try to copy from homeworks of previous years.
> **Do not ask on piazza** how to implement some function for this homework, or how to calculate something needed for this homework.
> **Do not post code on piazza** asking whether or not it is correct. This is a violation of academic integrity because it biases other students who may read your post.
> **Do not post test cases on piazza** asking for what the correct solution should be.
>
> **Do** ask the professor or TAs if you are unsure about whether certain actions constitute dishonesty. It is better to be safe than sorry.

Project Description

RipeApe pharmacy is developing a self-service automated system to alert customers about potential drug interactions for both prescription and over-the-counter drugs. Many medications should not be taken together or should not be taken if a patient has certain symptoms and allergies. Evaluating all the criteria for whether a patient can take a particular medication requires the patient's medical history and expert knowledge from a health care provider. The system, however, can provide an instant guideline to keep patients informed and minimize the risks.

You are assigned by RipeApe to develop a beta version of the system using the first order logic inference. Patient history and drug compatibility data will be encoded as first order logic clauses in the knowledge base. The program takes a query of new drug list and provide a logical conclusion whether to issue a warning.

You will use **first-order logic resolution** to solve this problem.

Format for input.txt

```
<N = NUMBER OF QUERIES>
<QUERY 1>
…
<QUERY N>
<K = NUMBER OF GIVEN SENTENCES IN THE KNOWLEDGE BASE>
<SENTENCE 1>
…
<SENTENCE K>
```

The first line contains an integer *N* specifying the number of queries. The following *N* lines contain one query per line. The line after the last query contains an integer *K* specifying the number of sentences in the knowledge base. The remaining *K* lines contain the sentences in the knowledge base, one sentence per line.

Query format: Each query will be a single literal of the form `Predicate(Constant_Arguments)` or `~Predicate(Constant_Arguments)` and will not contain any variables. Each predicate will have between 1 and 25 constant arguments. Two or more arguments will be separated by commas.

KB format: Each sentence in the knowledge base is written in one of the following forms:

1) An *implication* of the form $p_1 \wedge p_2 \wedge \ldots \wedge p_m \Rightarrow q$, where its premise is a conjunction of literals and its conclusion is a single literal. Remember that a literal is an atomic sentence or a negated atomic sentence.
2) A single literal: *q* or ~*q*

Note:

1. `&` denotes the *conjunction* operator.
2. `|` denotes the *disjunction* operator. It will not appear in the queries nor in the KB given as input. But you will likely need it to create your proofs.
3. `=>` denotes the *implication* operator.
4. `~` denotes the *negation* operator.
5. No other operators besides `&`, `=>`, and `~` are used in the knowledge base.
6. There will be no parentheses in the KB except as used to denote arguments of predicates.

7. Variables are denoted by a single lowercase letter.
8. All predicates (such as `HighBP`) and constants (such as `Alice`) are case sensitive alphabetical strings that begin with uppercase letters.
9. Each predicate takes at least one argument. Predicates will take at most 25 arguments. A given predicate name will not appear with different number of arguments.
10. There will be at most 10 queries and 100 sentences in the knowledge base.
11. See the sample input below for spacing patterns.
12. You can assume that the input format is exactly as it is described.
13. There will be no syntax errors in the given input.
14. The KB will be true (i.e., will not contain contradictions).

Format for output.txt:

For each query, determine if that query can be inferred from the knowledge base or not, one query per line:

```
<ANSWER 1>
…
<ANSWER N>
```

Each answer should be either TRUE if you can prove that the corresponding query sentence is true given the knowledge base, or FALSE if you cannot.

Notes and hints:

- Please name your program "**homework.xxx**" where 'xxx' is the extension for the programming language you choose. ("**py**" for python, "**cpp**" for C++, and "**java**" for Java). If you are using C++11, then the name of your file should be "homework11.cpp" and if you are using python3 then the name of your file should be "homework3.py".
- If you decide that the given statement can be inferred from the knowledge base, every variable in each sentence used in the proving process should be unified with a Constant (i.e., unify variables to constants before you trigger a step of resolution).
- All variables are assumed to be universally quantified. There is no existential quantifier in this homework. There is no need for Skolem functions or Skolem constants.
- Operator priorities apply (negation has higher priority than conjunction). There will be no parentheses in the sentences, other than around arguments of predicates.
- The knowledge base is consistent.
- If you run into a loop and there is no alternative path you can try, report FALSE. An example for this would be having two rules ***(1)*** A(x) => B(x) and ***(2)*** B(x) => A(x) and wanting to prove A(John). In this case your program should report FALSE.

- Note that the KB is not in Horn form because we allow more than one positive literal. So you indeed must use resolution and cannot use generalized Modus Ponens.

Example 1:

For this input.txt:

```
1
Take(Alice,NSAIDs)
2
Take(x,Warfarin) => ~Take(x,NSAIDs)
Take(Alice,Warfarin)
```

your output.txt should be:

```
FALSE
```

Example 2:

For this input.txt:

```
2
Alert(Bob,NSAIDs)
Alert(Bob,VitC)
5
Take(x,Warfarin) => ~Take(x,NSAIDs)
HighBP(x) => Alert(x,NSAIDs)
Take(Bob,Antacids)
Take(Bob,VitA)
HighBP(Bob)
```

your output.txt should be:

```
TRUE
FALSE
```

Example 3:

For this input.txt:

```
3
Alert(Alice,VitE)
Alert(Bob,VitE)
Alert(John,VitE)
9
Migraine(x) & HighBP(x) => Take(x,Timolol)
Take(x,Warfarin) & Take(x,Timolol) => Alert(x,VitE)
Migraine(Alice)
Migraine(Bob)
HighBP(Bob)
OldAge(John)
HighBP(John)
Take(John,Timolol)
Take(Bob,Warfarin)
```

your output.txt should be:

```
FALSE
TRUE
FALSE
```

Disclaimer: Examples and test cases are for the purpose of testing logic inference in this homework only. They do NOT represent factual information on drug interactions.