CSCI 561: Foundations of Artificial Intelligence
Spring 2016, Homework #2
Due on Mar 9, 2016 at 11:59 PM PST

Introduction

Thanks to your battle planner, the wise Master Yoda made some crucial strategic decisions, and led his brave squirrel warriors to win a series of decisive battles against their opponent. The Viterbi Fluffy Hackers family controls all territories in the USC campus, and has eradicated the treacherous Leavey Ninja Squirrels family for good.

Uniting the whole USC campus, Master Yoda now set his sight on other campuses in the Los Angeles galaxy, rumored to have fruitful production of tree nuts. As an ancient prophecy reveals, some squirrel families from other campuses (e.g. The squirrel family from UCLA campus led by family patriarch Darth Sidious) have fallen into the dark side, and a galaxy war is imminent. Unfortunately, Master Yoda has found out some squirrels from his own family are also tempted by the dark side, constantly revealing strategic secrets to their enemy via SquirrelChat. Using the power of the force (a.k.a. the Great Firewall), Master Yoda can take control of all SquirrelChat message traffic. However, due to the large volume of messages, he again requested you, the Chosen One, to design a logical system to find out the family traitors.

To help you develop your system, Master Yoda has provided you with some knowledge bases he already learned. The knowledge bases contain **first-order definite clauses** with the following defined operators:

NOT X	~X
X AND Y	X && Y
X IMPLIES Y	X => Y

The wise Master Yoda advises you to use **backward-chaining** to develop the system.

Assignment

You will be given a knowledge base and a query sentence. You need to determine if the query can be inferred from the information given in the knowledge base. You are required to use **backward-chaining** (AIMA Figure 9.6) to solve this problem.

Pseudocode

A simple backward-chaining algorithm for first-order knowledge bases: AIMA Figure 9.6

Input

You will be given a knowledge base and the query in a text file ending with a **.txt** extension. The first line of the input file contains the query. The query can have three forms:

1) as a *fact* with a **single** atomic sentence:
   e.g. `Traitor(Anakin)`
2) as several *facts* with **multiple** atomic sentences, separated by ' && ':
   e.g. `Knows(Sidious, Pine) && Traitor(Anakin)`
3) as a **single** predicate with **one** unknown variable:
   e.g. `Traitor(x)`

The second line contains an integer *n* specifying the number of clauses in the knowledge base.

The remaining lines contain the clauses in the knowledge base, **one per line**. Each clause is written in one of the following forms:

1) as an *implication* of the form $p1 \wedge p2 \wedge ... \wedge pn \Rightarrow q$, whose premise is a conjunction of atomic sentences and whose conclusion is a **single** atomic sentence.
2) as a *fact* with a **single** atomic sentence: q. Each atomic sentence is a predicate applied to a certain number of arguments.

The example below illustrates the format of the input file:

(sample01.txt)

```
Traitor(Anakin)
8
ViterbiSquirrel(x) && Secret(y) && Tells(x, y, z) && ~Ally(z) =>
Traitor(x)
Knows(Sidious, Pine)
Resource(Pine)
Resource(x) && Knows(Sidious, x) => Tells(Anakin, x, Sidious)
Resource(x) => Secret(x)
~Enemy(x, USC) => Ally(x)
ViterbiSquirrel(Anakin)
Enemy(Sidious, USC)
```

NOTES:

- && denotes the AND operator. ~ denotes the negation operator. => denotes the implication operator. No other operators besides &&, ~ and => are used.
- Conjunctions will be separated by ' && ' (whitespace on the both sides). Multiple arguments in a predicate will be separated by ', ' (whitespace on the right side). Implication operator (=>) is surrounded by whitespace on the both sides.
- Variables are denoted by a **single lowercase** letter.
- All predicates (such as Secret) and constants (such as Anakin) begin with **uppercase** letters. You can assume that all predicate names or constant names have at most 20 letters.
- A query will have at most **one** unknown variable, but it may contain other constants as arguments. There won't be a query with multiple atomic sentences containing unknown variables. In other words, if && is in the query, there won't be an unknown variable.
- You can assume that all implications have at most five predicates in the premise (on the left side of =>).
- You can assume that all predicates have at least one and at most three arguments.
- Each input file is independent. There is no need to save knowledge base for other test cases.
- The input file given to your program will not contain any formatting errors, so it is not necessary to check for those.

Output

The process of backward-chaining should be printed to a file called **output.txt**. Given the sample input above, the output content should be as follows:

(sample01.output.txt)

```
Ask: Traitor(Anakin)
Ask: ViterbiSquirrel(Anakin)
True: ViterbiSquirrel(Anakin)
Ask: Secret(_)
Ask: Resource(_)
True: Resource(Pine)
True: Secret(Pine)
Ask: Tells(Anakin, Pine, _)
Ask: Resource(Pine)
True: Resource(Pine)
Ask: Knows(Sidious, Pine)
True: Knows(Sidious, Pine)
True: Tells(Anakin, Pine, Sidious)
Ask: ~Ally(Sidious)
Ask: Ally(Sidious)
Ask: ~Enemy(Sidious, USC)
Ask: Enemy(Sidious, USC)
```

```
True: Enemy(Sidious, USC)
False: ~Enemy(Sidious, USC)
False: Ally(Sidious)
True: ~Ally(Sidious)
True: Traitor(Anakin)
True
```

In each line, a colon (with a whitespace on its right side) separates the action/reply (left side) and the query (right side). There are only 3 possible action/reply: Ask, True, and False. Taking the example above, we first 'Ask' the knowledge base (KB) whether Anakin is a traitor (line 1 in output), which triggered a rule (line 3 in input). Based on the rule, whose first literal is USCFamily(x) , the KB 'Ask' whether Anakin is from USCFamily (line 2 in output). The KB found Anakin is indeed from USCFamily (line 9 in input), so it replied 'True' (line 3 in output). Next, the KB checked the 2nd literal, Secret(y), of the rule. Because variable y was unknown, it replaced y with an underscore, and 'Ask' what is the resource. The process continued until the initial query was proved (True), disproved (False), or unprovable (False). The last line of the output file should be the conclusion (either True or False).

If the query is changed to `Traitor(Frodo)`, then the output should be as follows. (Note: with False meaning that you either disprove the query, or the query is unprovable.)

```
Ask: Traitor(Frodo)
Ask: ViterbiSquirrel(Frodo)
False: ViterbiSquirrel(Frodo)
False: Traitor(Frodo)
False
```

For the 3 different types of query:

1) If the query is a *fact* with a **single** atomic sentence, then print the log, ending with a conclusion True / False.
   e.g. The example above.

2) If the query is several *facts* with **multiple** atomic sentences, then ask the knowledge base about each fact one-by-one **from left to right**. If any fact is disproved or unprovable, end the query and print False. If all facts are proved, print True. Only print True / False in the end for the whole query, not individual facts.
   e.g. If the query for the input example is changed to:
   `Knows(Sidious, Pine) && Traitor(Anakin)`
   Then output example will have two more lines in the beginning:
   `Ask: Knows(Sidious, Pine)`
   `True: Knows(Sidious, Pine)`
   Please also refer to *sample04.txt* for this kind of query.

3) If the query is a **single** predicate with **one** unknown variable, ask the knowledge base until a variable substitution is found so that the predicate is proved, and print True, or if no such substitution can be found, print False. There are possibly more than one valid substitution, but you only need to find one, and you should search **in the order of the input file (first line to the last line)**, and **from left to right for each sentence** to find the same one as in the solution. As the other types of queries, you only need to print True / False in the end. **Don't print any extra line for variable substitution.**
   e.g. If the query for the input example is changed to:
   `Traitor(x)`
   Then the first two lines of the output example will be changed to:
   `Ask: Traitor(_)`
   `Ask: ViterbiSquirrel(_)`
   Please also refer to *sample02.txt* and *sample03.txt* for this kind of query.

NOTES:

- Process backward-chaining **in the order of the input file (first line to the last line)**, and **from left to right for each sentence** so that your output has the same order as the solution.
- Action and reply should be separated by ': ' (whitespace on the right side). Multiple arguments in a predicate should be separated by ', ' (whitespace on the right side).
- **Replace all unknown variables as an underscore** in the output file. **Don't** output any unknown variable as lowercase letter.
- The conclusion should be False if you either disprove the query, or the query is unprovable.
- Please follow the format of sample output carefully, including the location and number of spaces.
- You will get zero grade if you don't follow the output format exactly. Any regrading request about this will be ignored. E.g. if you print "Correct" instead of "True", you will get zero grade for all affected test cases, even if your program is actually correct.
- With the given description, we don't believe that multiple outputs are possible for any test case. If you are able to think of any such case, please let us know and we will make the necessary changes in the grading guidelines.
- The final test cases will be different from the sample test cases provided. Your assignment will be graded based on the performance on the final test cases only.

Additional Rules for Outputs

- These rules are defined to make the output file more reasonable and readable, and also to ensure it's unique. Please read and follow them very carefully.
- To begin with, here are the input and output of *sample05.txt* as example, please pay attention to the bolded lines:

(sample05.txt)

```
Aunt(Jane, Shelly)
14
Man(Peter)
Man(Gary)
Man(Kevin)
~Man(May)
~Man(Jane)
Woman(Shelly)
Parent(Kevin, Peter)
Parent(Kevin, Gary)
Parent(Kevin, Jane)
Parent(May, Shelly)
Parent(p, a) && Parent(p, b) => Siblings(a, b)
Man(u) && Siblings(u, p) && Parent(p, n) => Uncle(u, n)
~Man(a) && Siblings(a, p) && Parent(p, n) => Aunt(a, n)
~Woman(a) => Man(a)
```

(sample05.output.txt)

```
Ask: Aunt(Jane, Shelly)
Ask: ~Man(Jane)
True: ~Man(Jane)
Ask: Siblings(Jane, _)
Ask: Parent(_, Jane)
True: Parent(Kevin, Jane)
Ask: Parent(Kevin, _)
True: Parent(Kevin, Peter)
True: Siblings(Jane, Peter)
Ask: Parent(Peter, Shelly)
False: Parent(Peter, Shelly)
Ask: Parent(Kevin, _)
True: Parent(Kevin, Gary)
True: Siblings(Jane, Gary)
Ask: Parent(Gary, Shelly)
False: Parent(Gary, Shelly)
Ask: Parent(Kevin, _)
True: Parent(Kevin, Jane)
True: Siblings(Jane, Jane)
Ask: Parent(Jane, Shelly)
False: Parent(Jane, Shelly)
False: Aunt(Jane, Shelly)
False
```

- **For "Ask"**: If there are multiple valid unification for an "Ask", you should **print one "Ask" every time for each unification** (e.g. "**`Ask: Parent(Kevin, _)`**" in the above sample). But also remember that the latter ones are only visited if the previous ones do not work out in the end (in the following recursive levels), and you should visit them in the order of the knowledge base. If none of the unification work out in the end, you don't need to print another "Ask" (e.g. there's no additional "**`Ask: Parent(Kevin, _)`**" before the conclusion "False").
- **For "False"** (in the log, not conclusion): According to the pesudocode, the query is solved in a recursive process. However, if an "Ask"ed predicate is ever proved to be "True", even in the followed recursive calls other predicate cannot be proved and returned, this **proved predicate should never print a "False"**. For example, in the above example, there's an implication knowledge:

  `~Man(a) && Siblings(a, p) && Parent(p, n) => Aunt(a, n)`

  In the end of the logs, after "**`False: Parent(Jane, Shelly)`**", the next log is "**`False: Aunt(Jane, Shelly)`**", but no "False" for "~Man()" or "Siblings()", because they are already proved in early process.

  Similarly, for queries with multiple predicates, if the first predicate is proved but the second one is disproved, only print "False" log for the second predicate, then the "False" conclusion. Please refer to *sample04.txt* for an example.
- For negation operator **~**: For simplicity, **negation operator will always appear in knowledge base either in a fact, or in one predicate on the left side of an implication**. *sample01.txt* and *sample05.txt* show examples for each of these cases. It will never appear in the conclusion (right side) of an implication, or in the query.

  In addition, you may assume (if you follow the correct processing order) whenever a predicate with negation operator is the next query, it will never contain an unresolved variable (any variable should already have a substitution from earlier predicate, e.g. "**`Ask: ~Ally(Sidious)`**" in *sample01.txt*).

  The correct order to process a negated predicate is first to check if there is a negated fact in KB as a direct match (e.g. "**`Ask: ~Man(Jane)`**" in *sample05.txt*), otherwise, ask the reversed question (remove negation, as in *sample01.txt*).

  You may also assume there won't be negation implications forming a circle (e.g., if there is "**`~Woman(a) => Man(a)`**", there **won't** be "**`~Man(a) => Woman(a)`**"), so keep asking the un-negated questions won't lead to a dead cycle.
- For simplicity, in an implication sentence with variables, **different variables can be substituted with the same constant**, e.g. "**`True: Siblings(Jane, Jane)`**" is also checked in the above example (*sample05.output.txt*).
- **Don't save any proved new knowledge to the KB**. If the same knowledge is asked again during the same test case, you should treat it as an unknown knowledge and enter the recursive process as normal. E.g. if the query of *sample01.txt* is changed to:

  `Traitor(Anakin) && Tells(Anakin, Pine, Sidious)`

Even though "**`Tells(Anakin, Pine, Sidious)`**" is already proved when querying "**`Traitor(Anakin)`**", the output should still have these extra logs in the end:

```
True: Traitor(Anakin)       (the same as in sample01.output.txt)
Ask: Tells(Anakin, Pine, Sidious)
Ask: Resource(Pine)
True: Resource(Pine)
Ask: Knows(Sidious, Pine)
True: Knows(Sidious, Pine)
True: Tells(Anakin, Pine, Sidious)
True                        (the same as in sample01.output.txt)
```

Grading Notice

- **Please follow the instructions carefully. Any deviations from the instructions will lead your grade to be zero for the assignment**. If you have any doubts, please use the discussion board. Do not assume anything that is not explicitly stated.
- You must use **PYTHON** (Python 2.7) to implement your code. You are allowed to use standard libraries only. You have to implement any other functions or methods by yourself.
- You need to create a file named "hw2cs561s16.py". The command to run your program would be as follows: (When you submit the homework on labs.vocareum.com, the following commands will be executed.)

  python hw2cs561s16.py –i *inputFile*

- You will use labs.vocareum.com to submit your code. Please refer to http://help.vocareum.com/article/30-getting-started-students to get started with the system. Please only upload your code to the "/work" directory. Don't create any subfolder or upload any other files.
- If we are unable to execute your code successfully, you will not receive any credits.
- **When you press "Submit" on Vocareum, your code will be run against the grading script and the sample input/output files. You will receive feedback on where your code is making errors, if any. You can click "Submit" to test new versions of your code as many times as you like up until the deadline. Your last submission will be graded. The sample input/output files are designed to cover many basic situations of the problem and rules of the output log, so please utilize them as much as you can.**

- Some details about grades:
  - For each test case, your output file ("output.txt") should contain two parts: **conclusion** (True / False) in the final line, and **log** in all remaining lines.
  - **Please make sure you print a conclusion in the final line of your output, and don't print any extra line break after the conclusion).**

    - If you end up print more than one conclusion in one output file, only the final one is counted, and the others will be considered as part of the log (and thus you won't get any points for the log).

- Your program should handle all test cases within a reasonable time (not more than a few seconds for each sample test case). The complexity of test cases is similar to, but not necessarily the same as, the ones provided in the homework.
- The deadline for this assignment is Mar 9, 2016 at 11:59 PM PST. **No late homework will be accepted.** Any late submission will not be graded. Any email for late submission will be ignored.