CSE 440: Artificial Intelligence (Section 1)

**Summer 2016**

**Assignment 1**

**Due date: Tuesday 28th, June, 2016**

### Task

The task in this programming assignment is to implement a knowledge base and an inference engine for the wumpus world. First of all, you have to create a knowledge base (stored as a text file) storing the rules of the wumpus world, i.e., what we know about pits, monsters, breeze, and stench. Second, you have to create an inference engine, that given a knowledge base and a statement determines if, based on the knowledge base, the statement is definitely true, definitely false, or of unknown truth value.

### Command-line Arguments

The program should be invoked from the command line as follows:

check\_true\_false wumpus\_rules.txt [additional\_knowledge\_file] [statement\_file]

For example:

check\_true\_false wumpus\_rules.txt kb1.txt statement1.txt

* Argument wumpus\_rules.txt specifies the location of a text file containing the wumpus rules, i.e., the rules that are true in any possible wumpus world, as specified above (once again, note that the specifications above are not identical to the ones in the book).
* Argument [additional\_knowledge\_file] specifies an input file that contains additional information, presumably collected by the agent as it moves from square to square. For example, see kb3.txt.
* Argument [statement\_file] specifies an input file that contains a single logical statement. The program should check if, given the information in wumpus\_rules.txt and [additional\_knowledge\_file], the statement in [statement\_file] is definitely true, definitely false, or none of the above.

### Output

Your program should create a text file called "result.txt". Depending on what your inference algorithm determined about the statement being true or false, the output file should contain one of the following four outputs:

* **definitely true**. This should be the output if the knowledge base entails the statement, and the knowledge base does not entail the negation of the statement.
* **definitely false**. This should be the output if the knowledge base entails the negation of the statement, and the knowledge base does not entail the statement.
* **possibly true, possibly false**. This should be the output if the knowledge base entails neither the statement nor the negation of the statement.
* **both true and false**. This should be the output if the knowledge base entails both the statement and the the negation of the statement. This happens when the knowledge base is always false (i.e., when the knowledge base is false for every single row of the truth table).

Note that by "knowledge base" we are referring to the conjunction of all statements contained in wumpus\_rules.txt AND in the additional knowledge file.

Also note that the sample code provided below stores the words "result unknown" to the result.txt file. Also, the "both true and false" output should be given when the knowledge base (i.e., the info stored in wumpus\_rules.txt AND in the additional knowledge file) entails both the statement from statement\_file AND the negation of that statement.

### Syntax

The wumpus rules file and the additional knowledge file contain multiple lines. Each line contains a logical statement. The knowledge base constructed by the program should be a conjunction of all the statements contained in the two files. The sample code (as described later) already does that. The statement file contains a single line, with a single logical statement.

Statements are given in prefix notation. Some examples of prefix notation are:

(or M\_1\_1 B\_1\_2)

(and M\_1\_2 S\_1\_1 (not (or M\_1\_3 M\_1\_4)))

(if M\_1\_1 (and S\_1\_2 S\_1\_3))

(iff M\_1\_2 (and S\_1\_1 S\_1\_3 S\_2\_2))

(xor B\_2\_2 P\_1\_2)

P\_1\_1

B\_3\_4

(not P\_1\_1)

Statements can be nested, as shown in the above examples.

Note that:

* Any open parenthesis that is not the first character of a text line must be preceded by white space.
* Any open parenthesis must be immediately followed by a connective (without any white space in between).
* Any close parenthesis that is not the last character of a text line must be followed by white space.
* If the logical expression contains just a symbol (and no connectives), the symbol should NOT be enclosed in parentheses. For example, (P\_1\_1) is not legal, whereas (not P\_1\_1) is legal. See also the example statements given above.
* Each logical expression should be contained in a single line.
* The wumpus rules file and the additional knowledge file contain a set of logical expressions. The statement file should contain a single logical expression. If it contains more than one logical expression, only the first one is read.
* Lines starting with # are treated as comment lines, and ignored.
* You can have empty lines, but they must be totally empty. If a line has a single space on it (and nothing more) the program will complain and not read the file successfully.

There are six connectives: and, or, xor, not, if, iff. No other connectives are allowed to be used in the input files. Here is some additional information:

* A statement can consist of either a single symbol, or a connective connecting multiple (sub)statements. Notice that this is a recursive definition. In other words, statements are symbols or more complicated statements that we can make by connecting simpler statements with one of the six connectives.
* Connectives "and", "or", and "xor" can connect any number of statements, including 0 statements. It is legal for a statement consisting of an "and", "or", or "xor" connective to have no substatements, e.g., (and). An "and" statement with zero substatements is true. An "or" or "xor" statement with zero substatements is false. An "xor" statement is true if exactly 1 substatement is true (no more, no fewer).
* Connectives "if" and "iff" require exactly two substatements.
* Connective "not" requires exactly one substatement.

The only symbols that are allowed to be used are:

* M\_i\_j (standing for "there is a monster at square (i, j)).
* S\_i\_j (standing for "there is a stench at square (i, j)).
* P\_i\_j (standing for "there is a pit at square (i, j)).
* B\_i\_j (standing for "there is a breeze at square (i, j)).

NO OTHER SYMBOLS ARE ALLOWED. Also, note that i and j can take values 1, 2, 3, and 4. In other words, there will be 16 unique symbols of the form M\_i\_j, 16 unique symbols of the form S\_i\_j, 16 unique symbols of the form P\_i\_j, and 16 unique symbols of the form B\_i\_j, for a total of 64 unique symbols.

### The Wumpus Rules

Here is what we know to be true in any wumpus world, for the purposes of this assignment (NOTE THAT THESE RULES ARE NOT IDENTICAL TO THE ONES IN THE TEXTBOOK):

1. If there is a monster at square (i,j), there is stench at all adjacent squares.
2. If there is stench at square (i,j), there is a monster at one of the adjacent squares.
3. If there is a pit at square (i,j), there is breeze at all adjacent squares.
4. If there is breeze at square (i,j), there is a pit at one or more of the adjacent squares.
5. There is one and only one monster (no more, no fewer).
6. Squares (1,1), (1,2), (2,1), (2,2) have no monsters and no pits.
7. The number of pits can be between 1 and 11.
8. We don't care about gold, glitter, and arrows, there will be no information about them in the knowledge base, and no reference to them in the statement.

### Sample code

The following code implements, in Java and C++, a system that reads text files containing information for the knowledge base and the statement whose truth we want to check. Feel free to use that code and build on top of it. Also feel free to ignore that code and start from scratch.

* Java: files CheckTrueFalse.java and LogicalExpression.java
* Python, version 2.4 (to make sure it runs: check\_true\_false.py and logical\_expression.py
* C++: files check\_true\_false.cpp and [check\_true\_false.h](http://vlm1.uta.edu/~athitsos/courses/old/cse4308_fall2015/assignments/assignment4a/check_true_false.h)

You can test this code, by compiling and running on input files a.txt, b.txt, and c.txt. For example, for the Java code you can run it as:

javac \*.java

java CheckTrueFalse a.txt b.txt c.txt

and for C++, you can do:

g++ -o check\_true\_false check\_true\_false.cpp

./check\_true\_false a.txt b.txt c.txt

### Efficiency Requirement

Brute-force enumeration of the 264 possible assignments to the 64 Boolean variables will be too inefficient to produce answers in a reasonable amount of time. Because of that, we will only be testing your solutions with cases where the additional knowledge file contains specific information about at least 48 of the symbols.

For example, suppose that the agent has already been at square (2,3). Then, the agent knows for that square that:

* There is no monster (otherwise the agent would have died).
* There is no pit (otherwise the agent would have died).

Furthermore, the agent knows whether or not there is stench and/or breeze at that square. Suppose that, in our example, there is breeze and no stench.

Then, the additional knowledge file would contain these lines for square 2,3:

(not M\_2\_3)

(not P\_2\_3)

B\_2\_3

(not S\_2\_3)

You can assume that, in all our test cases, there will be at least 48 lines like these four lines shown above, specifying for at least 48 symbols whether they are true or false. Assuming that you implement the TT-Entails algorithm, your program should identify those symbols and their values right at the beginning. You can identify those symbols using these guidelines:

* Note that the sample code stores the knowledge base as a LogicalExpression object, whose connective at the root is an AND. Let's call this LogicalExpression object knowledge\_base.
* Suppose that you have a line such as "B\_2\_3" in the additional knowledge file. Such a line generates a child of knowledge\_base that is a leaf, and has its "symbol" variable set to "B\_2\_3". You can write code that explicitly looks for such children of knowledge\_base.
* Suppose that you have a line such as "(not M\_2\_3") in the additional knowledge file. Such a line generates a child of knowledge\_base whose connective is NOT, and whose only child is a leaf with its "symbol" variable set to "M\_2\_3". You can write code that explicitly looks for such children of knowledge\_base.

This way, your program will be able to initialize the model that TT-Entails passes to TT-Check-All with boolean assignments for at least 48 symbols, as opposed to passing an empty model. The list of symbols passed from TT-Entails to TT-Check-All should obviously NOT include the symbols that have been assigned values in the initial model. This way, at most 16 symbols will have unspecified values, and TT-Check-All will need to check at most 216 rows in the truth table, which is quite doable in a reasonable amount of time (a few seconds).

### Grading

The assignment will be graded out of 100 points.

* 40 points: submitting an appropriate wumpus\_rules.txt file that can be used as the first command-line input to the program, according to the propositional logic syntax and symbols defined above. The file should contain logical statements corresponding to the wumpus rules stated above. For each of the 8 rules, you need to determine if you need to add any statements to wumpus\_rules.txt because of that rule, and if so, what statements to add. For each of the eight rules, correct handling of that rule is worth 5 points.
* 12 points: satisfying the efficiency requirement, terminating in less than 120 seconds when the additional knowledge file specifies values for at least 48 of the 64 symbols.
* 48 points: correctness of results. In particular, 12 points will be allocated for each of the four output cases, and you get those 12 points if your program always produces the correct output for each of those four cases.

### How to submit

Implementations in C, C++, Java, and Python will be accepted. If you would like to use another language, please first check with the instructor via e-mail. Points will be taken off for failure to comply with this requirement.

The assignment should be submitted via gmail (cse440nsu@gmail.com). Submit a ZIPPED directory called assignment5.zip (no other forms of compression accepted, contact the instructor if you do not know how to produce .zip files). The directory should contain source code. The submission should also contain a file called readme.txt, which should specify precisely:

* Name and NSU ID of the student.
* What programming language is used.
* How to run the code, including very specific compilation instructions, if compilation is needed. Instructions such as "compile using g++" are NOT considered specific.

Submit the assignment as an attachment (zipped file) to the email: [cse440nsu@gmail.com](mailto:cse440nsu@gmail.com). The subject of the email must be “Assignment 5, NSU\_ID”.

Insufficient or unclear instructions will be penalized by up to 20 points.