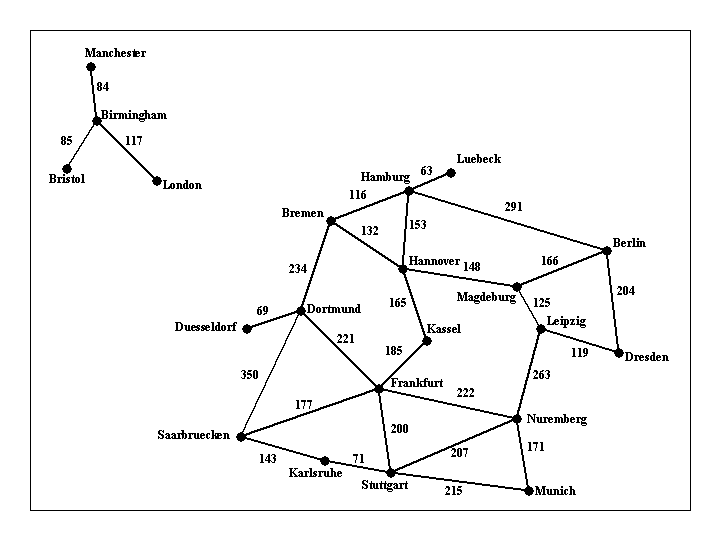
**Assignment 1**

**Programming Assignment - Uninformed Search**

  
Figure 1: Visual representation of [input1.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt1_files/input1.txt)

Implement a search algorithm that can find a route between any two cities. Your program will be called find\_route, and will take exactly three commandline arguments, as follows:  
  
***find\_route input\_filename origin\_city destination\_city***  
  
An example command line is:  
  
find\_route input1.txt Munich Berlin  
  
Argument input\_filename is the name of a text file such as [input1.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt1_files/input1.txt), that describes road connections between cities in some part of the world. For example, the road system described by file input1.txt can be visualized in Figure 1 shown above. You can assume that the input file is formatted in the same way as [input1.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt1_files/input1.txt): each line contains three items. The last line contains the items "END OF INPUT", and that is how the program can detect that it has reached the end of the file. The other lines of the file contain, in this order, a source city, a destination city, and the length in kilometers of the road connecting directly those two cities. Each city name will be a single word (for example, we will use New\_York instead of New York), consisting of upper and lowercase letters and possibly underscores.  
  
**IMPORTANT NOTE**: MULTIPLE INPUT FILES WILL BE USED TO GRADE THE ASSIGNMENT, FILE [input1.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt1_files/input1.txt) IS JUST AN EXAMPLE. YOUR CODE SHOULD WORK WITH ANY INPUT FILE FORMATTED AS SPECIFIED ABOVE.  
  
The program will compute a route between the origin city and the destination city, and will print out both the length of the route and the list of all cities that lie on that route. For example,  
  
find\_route input1.txt Bremen Frankfurt  
  
should have the following:  
  
distance: 455 km  
route:   
Bremen to Dortmund, 234 km   
Dortmund to Frankfurt, 221 km   
  
and  
  
find\_route input1.txt London Frankfurt  
  
should have the following output:  
  
distance: infinity  
route:   
none  
  
For full credit, you should produce outputs identical in format to the above two examples.

**Suggestions**

The code needs to run on omega. If you have not even tried logging in on omega until the last day, there is a high probability that something will go wrong. You may find it convenient to do the code development and testing on your own laptop or home machine, but it is highly recommended that you log in to omega and compile a toy program ASAP, and that you compile and run an intermediate version of your code well before the deadline. Notify the instructor for any problems you may have.   
  
Pay close attention to all specifications on this page, including specifications about output format, submission format. Even in cases where the program works correctly, points will be taken off for non-compliance with the instructions given on this page (such as a different format for the program output, wrong compression format for the submitted code, and so on). The reason is that non-compliance with the instructions makes the grading process significantly (and unnecessarily) more time consuming.

**Grading**

The assignments will be graded out of 100 points.

* 40 points: The program always finds a route between the origin and the destination, as long as such a route exists.
* 30 points: The program terminates and reports that no route can be found when indeed no route exists that connects source and destination (e.g., if source is London and destination is Berlin, in the above example).
* 30 points: In addition to the above requirements, the program always returns optimal routes. In other words, no shorter route exists than the one reported by the program.
* Negative points: penalty points will be awarded by the instructor and TA generously and at will, for issues such as: code not running on omega, submission not including precise and accurate instructions for how to run the code, wrong compression format for the submission, or other failures to comply with the instructions given for this assignment. Partial credit for incorrect solutions will be given ONLY for code that is well designed and well documented. Code that is badly designed and badly documented can still get full credit as long as it accomplishes the required tasks.

**How to submit**

Implementations in C, C++, Java, and Python will be accepted. If you would like to use another language, make sure it will compile on omega and clear it with the instructor beforehand. Points will be taken off for failure to comply with this requirement.  
The assignment should be submitted via [Blackboard](http://elearn.uta.edu/). Submit a ZIPPED directory called assignment1\_<net-id>.zip (no other forms of compression accepted, contact the instructor or TA if you do not know how to produce .zip files). The directory should contain source code. Including binaries that work on omega (for Java and C++) is optional. The submission should also contain a file called readme.txt, which should specify precisely:

* Name and UTA ID of the student.
* What programming language is used.
* How the code is structured.
* How to run the code, including very specific compilation instructions, if compilation is needed. Instructions such as "compile using g++" are NOT considered specific.
* Insufficient or unclear instructions will be penalized by up to 10 points.
* **Code that does not run on omega machines gets AT MOST 75 points**.

**Submission checklist**

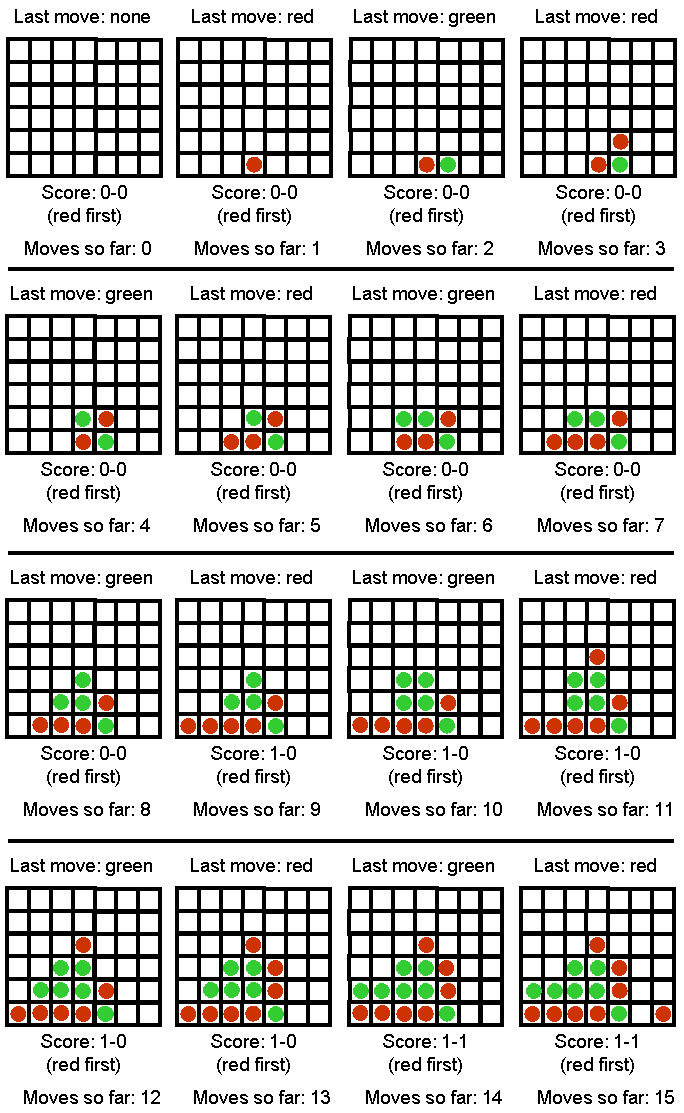
Is the code running on omega?  
Does the submission include a readme.txt file, as specified?

**Assignment 4**

**Programming Assignment - Game Playing Algorithms**

**Task**

The task in this programming assignment is to implement an agent that plays the Max-Connect4 game using search. Figure 1 shows the first few moves of a game. The game is played on a 6x7 grid, with six rows and seven columns. There are two players, player A (red) and player B (green). The two players take turns placing pieces on the board: the red player can only place red pieces, and the green player can only place green pieces.  
  
It is best to think of the board as standing upright. We will assign a number to every row and column, as follows: columns are numbered from left to right, with numbers 1, 2, ..., 7. Rows are numbered from bottom to top, with numbers 1, 2, ..., 6. When a player makes a move, the move is completely determined by specifying the COLUMN where the piece will be placed. If all six positions in that column are occupied, then the move is invalid, and the program should reject it and force the player to make a valid move. In a valid move, once the column is specified, the piece is placed on that column and "falls down", until it reaches the lowest unoccupied position in that column.  
  
The game is over when all positions are occupied. Obviously, every complete game consists of 42 moves, and each player makes 21 moves. The score, at the end of the game is determined as follows: consider each quadruple of four consecutive positions on board, either in the horizontal, vertical, or each of the two diagonal directions (from bottom left to top right and from bottom right to top left). The red player gets a point for each such quadruple where all four positions are occupied by red pieces. Similarly, the green player gets a point for each such quadruple where all four positions are occupied by green pieces. The player with the most points wins the game.  
  
Your program will run in two modes: an interactive mode, that is best suited for the program playing against a human player, and a one-move mode, where the program reads the current state of the game from an input file, makes a single move, and writes the resulting state to an output file. The one-move mode can be used to make programs play against each other. Note that THE PROGRAM MAY BE EITHER THE RED OR THE GREEN PLAYER, THAT WILL BE SPECIFIED BY THE STATE, AS SAVED IN THE INPUT FILE.  
  
As part of this assignment, you will also need to measure and report the time that your program takes, as a function of the number of moves it explores. All time measurements should report CPU time, not total time elapsed. CPU time does not depend on other users of the system, and thus is a meaningful measurement of the efficiency of the implementation.

  
Figure 1: Sample Max-Connect Game (15 moves in)

**Interactive Mode**

In the interactive mode, the game should run from the command line with the following arguments (assuming a Java implementation, with obvious changes for C++ or other implementations):  
  
***java maxconnect4 interactive [input\_file] [computer-next/human-next] [depth]***  
  
For example:  
  
*java maxconnect4 interactive input1.txt computer-next 7*

* Argument interactive specifies that the program runs in interactive mode.
* Argument [input\_file] specifies an input file that contains an initial board state. This way we can start the program from a non-empty board state. If the input file does not exist, the program should just create an empty board state and start again from there.
* Argument [computer-first/human-first] specifies whether the computer should make the next move or the human.
* Argument [depth] specifies the number of moves in advance that the computer should consider while searching for its next move. In other words, this argument specifies the depth of the search tree. Essentially, this argument will control the time takes for the computer to make a move.

After reading the input file, the program gets into the following loop:

1. If computer-next, goto 2, else goto 5.
2. Print the current board state and score. If the board is full, exit.
3. Choose and make the next move.
4. Save the current board state in a file called computer.txt (in same format as input file).
5. Print the current board state and score. If the board is full, exit.
6. Ask the human user to make a move (make sure that the move is valid, otherwise repeat request to the user).
7. Save the current board state in a file called human.txt (in same format as input file).
8. Goto 2.

**One-Move Mode**

The purpose of the one-move mode is to make it easy for programs to compete against each other, and communicate their moves to each other using text files. The one-move mode is invoked as follows:  
  
***java maxconnect4 one-move [input\_file] [output\_file] [depth]***  
  
For example:  
  
*java maxconnect4 one-move red\_next.txt green\_next.txt 5*  
  
In this case, the program simply makes a single move and terminates. In particular, the program should:

* Read the input file and initialize the board state and current score, as in interactive mode.
* Print the current board state and score. If the board is full, exit.
* Choose and make the next move.
* Print the current board state and score.
* Save the current board state to the output file **IN EXACTLY THE SAME FORMAT THAT IS USED FOR INPUT FILES**.
* Exit

**Sample code**

The sample code needs an input file to run. Sample input files that you can download are [input1.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/input1.txt) and [input2.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/input2.txt). You are free to make other input files to experiment with, as long as they follow the same format. In the input files, a 0 stands for an empty spot, a 1 stands for a piece played by the first player, and a 2 stands for a piece played by the second player. The last number in the input file indicates which player plays NEXT (and NOT which player played last). Sample code is available in:

* Java: download files [maxconnect4.java](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/maxconnect4.java), [GameBoard.java](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/GameBoard.java), and [AiPlayer.java](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/AiPlayer.java). Compile on omega using:  
    
  *javac maxconnect4.java GameBoard.java AiPlayer.java*  
    
  An example command line that runs the program (assuming that you have input1.txt saved in the same directory) is:  
    
  *java maxconnect4 one-move input1.txt output1.txt 10*
* C++: download file [maxconnect4.cpp](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/maxconnect4.cpp). Compile on omega using:  
    
  *g++ -o maxconnect4 maxconnect.cpp*  
    
  An example command line that runs the program (assuming that you have input1.txt saved in the same directory) is:  
    
  *maxconnect4 one-move input1.txt output1.txt 10*
* Python (Version 2.4): download file [maxconnect4.py](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/maxconnect4.py) and [MaxConnect4Game.py](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt4_files/MaxConnect4Game.py).   
    
  An example command line that runs the program (assuming that you have input1.txt saved in the same directory) is:  
    
  *./maxconnect4.py one-move input1.txt output1.txt 10*

The sample code implements a system playing max-connect4 (in one-move mode only) by making random moves. While the AI part of the sample code leaves much to be desired (your assignment is to fix that), the code can get you started by showing you how to represent and generate board states, how to save/load the game state to and from files in the desired format, and how to count the score (though faster score-counting methods are possible).

**Measuring Execution Time**

You can measure the execution time for your program on omega by inserting the word "time" in the beginning of your command line. For example, if you want to measure how much time it takes for your system to make one move with the depth parameter set to 10, try this:  
  
time java maxconnect4 one-move red\_next.txt green\_next.txt 10  
  
Your output will look something like:  
*real    0m0.003s  
    user    0m0.002s  
    sys     0m0.001s*  
  
Out of the above three lines, the **user** line is what you should report.

**Grading**

The assignments will be graded out of 100 points. There is also upto 15 possible extra credit points.

* 40 points: Implementing plain minimax.
* 25 points: Implementing alpha-beta pruning (if correctly implemented, will algo get 40 points for plain minimax, **you don't need to have separate implementation for it**)
* 20 points: Implementing the depth-limited version of minimax (if correctly implemented, and includes alpha-beta pruning, you also get the 40 points for plain minimax and 25 points for alpha-beta search, **you don't need to have separate implementations for those**).
  + For full credit, you obviously need to come up with a reasonable evaluation function to be used in the context of depth-limited search.
  + A "reasonable" evaluation function is defined to be an evaluation function that allows your program to consistently beat a player who just plays randomly.
* 5 points: Include a file, **eval\_explanation.txt (can also use .pdf, .doc or .docx)**, that explains the evaluation function used for depth-limited search.
* 10 points: Include in your submission an accurate table of depth limit vs CPU runtime (for making a single move using one move mode) when the board is empty. Document the number of measurements for each entry on the table. All measurements should be performed on omega. Your table should include every single depth, until (and including) the first depth for which the time exceeds one minute.

**How to submit**

Implementations in C, C++, Java, and Python will be accepted. If you would like to use another language, make sure it will compile on omega and clear it with the instructor beforehand. Points will be taken off for failure to comply with this requirement.  
The assignment should be submitted via [Blackboard](http://elearn.uta.edu/). Submit a ZIPPED directory called assignment4\_<net-id>.zip (no other forms of compression accepted, contact the instructor or TA if you do not know how to produce .zip files). The directory should contain source code. Including binaries that work on omega (for Java and C++) is optional. The submission should also contain a file called readme.txt, which should specify precisely:

* Name and UTA ID of the student.
* What programming language is used.
* How the code is structured.
* How to run the code, including very specific compilation instructions, if needed. Instructions such as "compile using g++" are NOT considered specific.
* Do you want to participate in the tournament (Make sure your code is eleigible. **Invalid code will be Penalized!!**).
* Insufficient or unclear instructions will be penalized by up to 10 points.
* **Code that does not run on omega machines gets MAX of 75 points**.

**Submission checklist**

Is the code running on omega?  
Does the submission include eval\_explanation.txt?  
Does the submission include the table of depth limit vs runtime?  
Does the submission include a readme.txt file, as specified?

**Assignment 6**

**Programming Assignment - Propositional Logic**

**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 commandline 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](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/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**):

* If there is a monster at square (i,j), there is stench at all adjacent squares.
* If there is stench at square (i,j), there is a monster at one of the adjacent squares.
* If there is a pit at square (i,j), there is breeze at all adjacent squares.
* If there is breeze at square (i,j), there is a pit at one or more of the adjacent squares.
* There is one and only one monster (no more, no fewer).
* Squares (1,1), (1,2), (2,1), (2,2) have no monsters and no pits.
* The number of pits can be between 1 and 11.
* 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](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/CheckTrueFalse.java) and [LogicalExpression.java](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/LogicalExpression.java)
* C++: files [check\_true\_false.cpp](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/check_true_false.cpp) and [check\_true\_false.h](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/check_true_false.h)
* Python (ver 2.4): [check\_true\_false.py](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/check_true_false.py) and [logical\_expression.py](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/logical_expression.py)

You can test this code, by compiling on omega, and running on input files [a.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/a.txt), [b.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/b.txt), and [c.txt](http://omega.uta.edu/~gopikrishnav/classes/2018/spring/4308_5360/assmts/assmt6_files/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**

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. Correct handling of any of the eight rules 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, make sure it will compile on omega and clear it with the instructor beforehand. Points will be taken off for failure to comply with this requirement.  
The assignment should be submitted via [Blackboard](http://elearn.uta.edu/). Submit a ZIPPED directory called assignment6\_<net-id>.zip (no other forms of compression accepted, contact the instructor or TA if you do not know how to produce .zip files). The directory should contain source code and the wumpus\_rules.txt file. Including binaries that work on omega (for Java and C++) is optional. The submission should also contain a file called readme.txt, which should specify precisely:

* Name and UTA ID of the student.
* What programming language is used.
* How the code is structured.
* How to run the code, including very specific compilation instructions, if compilation is needed. Instructions such as "compile using g++" are NOT considered specific.
* Insufficient or unclear instructions will be penalized by up to 10 points.
* **Code that does not run on omega machines gets AT MOST 75 points**.

**Submission checklist**

* **DID YOU INCLUDE THE wumpus\_rules.txt file**?
* Is the code running on omega?
* Does the attachment include a readme.txt file, as specified?