Assignment 3

Programming Assignment - Game Playing Algorithms / Propositional Logic

Total : 100 points

Task 1 (50 points):

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.

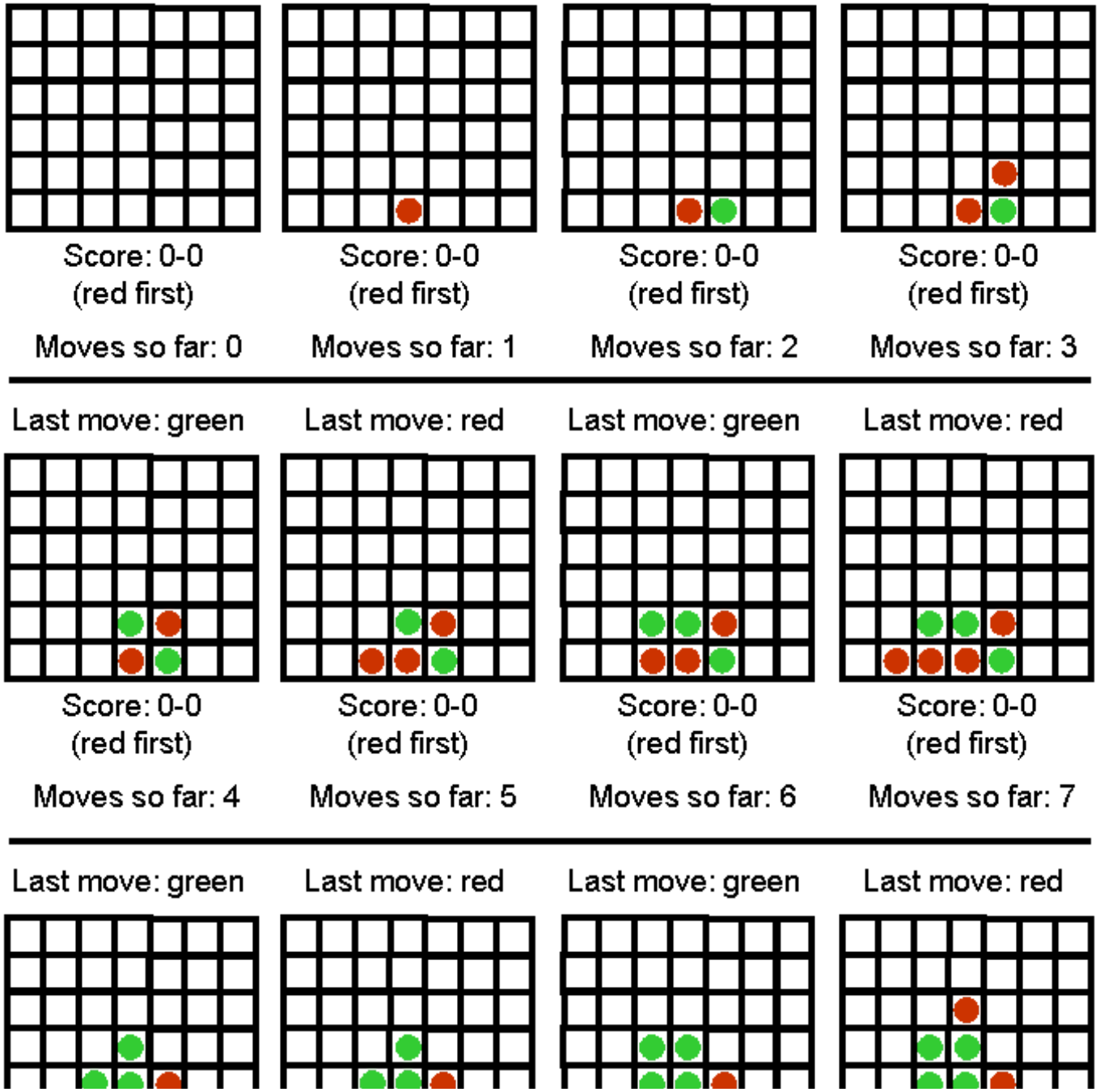
It is best to think of the board as standing upright. We will assign a number to every row and column, as follows: columns are numbered from left to right, with numbers 1, 2, ..., 7. Rows are numbered from bottom to top, with numbers 1, 2, ..., 6. When a player makes a move, the move is completely determined 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.

Last move: none Last move: red Last move: green Last move: red



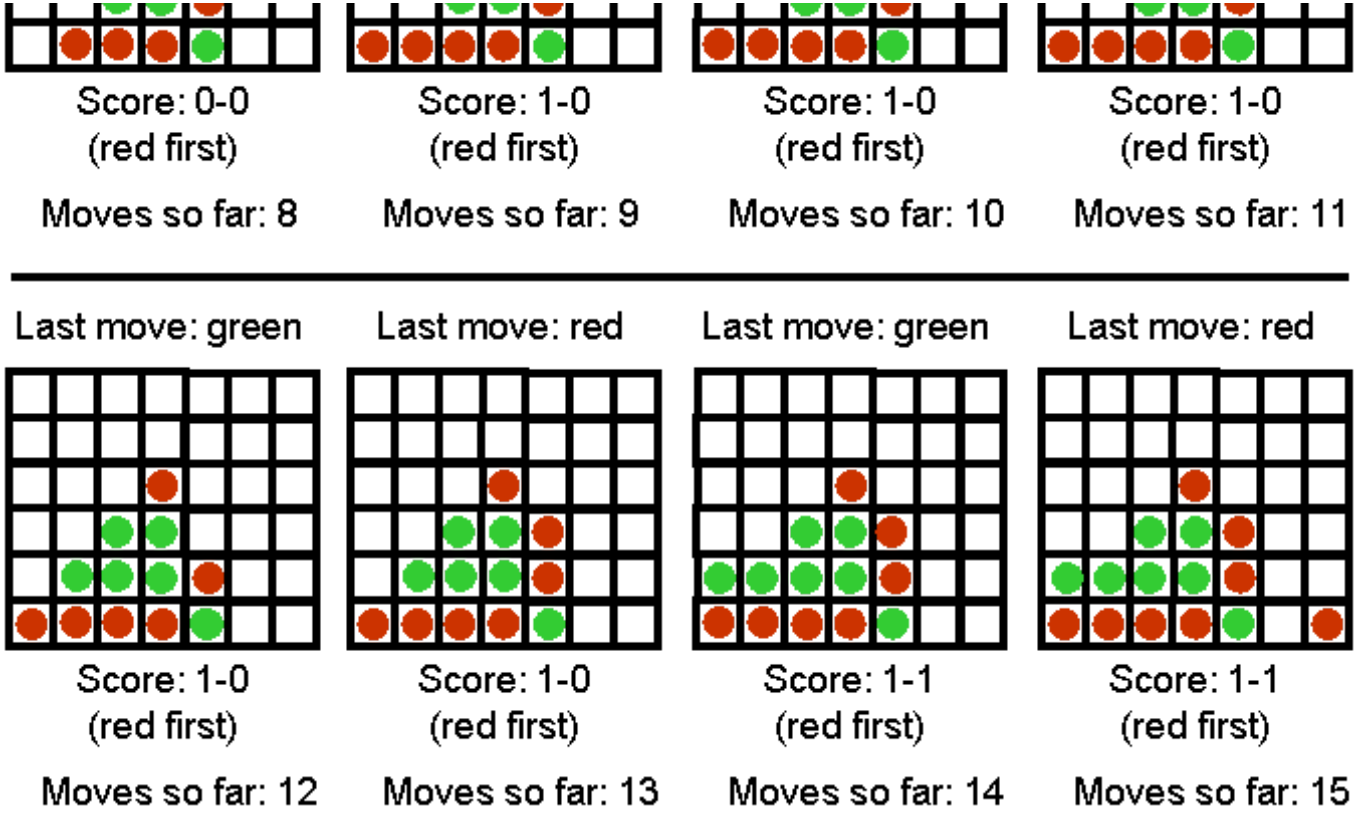


Figure 1: Sample Max-Connect Game (15 moves in)

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 and 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, GameBoard.java, and 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. 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 and MaxConnect4Game.py. (Alt link to zipped version of files here).

  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 task will be graded out of 50 points.

- 20 points: Implementing plain minimax.
- 15 points: Implementing alpha-beta pruning (if correctly implemented, will also get the 20 points for plain minimax, **you don't need to have separate implementation for it**)
- 10 points: Implementing the depth-limited version of minimax (if correctly implemented, and includes alpha-beta pruning, you also get the 20 points for plain minimax and 15 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.
- 3 points: Include a file, **eval_explanation.txt (can also use .pdf, .doc or .docx)**, that explains the evaluation function used for depth-limited search.
- 2 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.

Task 2 (50 Points):

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.
- 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 and LogicalExpression.java

- C++: files check_true_false.cpp and check_true_false.h
- Python (ver 2.4): check_true_false.py and logical_expression.py. (Zipped version of files here).

You can test this code, by compiling on omega, 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

Brute-force enumeration of the 2^{64} 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 2^{16} rows in the truth table, which is quite doable in a reasonable amount of time (a few seconds).

Grading

The task will be graded out of 50 points.

- 25 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.
- 5 points: satisfying the efficiency requirement, terminating in less than roughly 2 minutes when the additional knowledge file specifies values for at least 48 of the 64 symbols.
- 20 points: correctness of results. In particular, 5 points will be allocated for each of the four output cases, and you get all 20 points if your program always produces the correct output for each of those four cases