Tic-tac-toe extra credit, Spring 2012

1 TTT.java (extra credit)

This is about writing a program that implements a perfect player at tic-tac-toe. A perfect player is one who always makes a move that minimizes the best possible outcome of the opponent.

Look at the methods in TTTInterface.java. In addition to these methods, you need to implement a default constructor. You must also write a constructor that has a String parameter for passing a filename for a CharMatrix file that holds an initial tic-tac-toe configuration.

An example of a tic-tac-toe CharMatrix file can be found in TTT.dat. The marks made by first player who moved are W's, for "white" and the marks made by the second player who moved are B's, for "black". Empty squares are represented by the character "-". The configuration represented by `TTT.dat` is one where white and black have each made one move. It must be 3-by-3.

Being able to specify an input file allows you to start the opponent (your program) with a board state from which it can force a win, or a board state that allows you to force a win.

You will need to write a separate class that allows you to alternate moves with your TTT object, but you don't need to turn it in. We will use our own.

The *state* of the board consists of the current locations of marks and a boolean value `whiteMove`, which is true if it's white's move, and false if it's black's move.

At the heart of the strategy of the automated tic-tac-toe player is the `stateScore` method, which assigns a score to any possible state of the board. This score can be -1, 0, or 1. A score of 1 means that White has won or can force a win, a score of -1 means that Black has won or can force a win, and a score of 0 means that the game has ended in a draw, or neither player can force a win.

Such a method can act as a "crystal ball"; a player can try placing a piece in a square and consult with the crystal ball about whether a wind can be forced from the new configuration. If it can't, the player can remove the piece and try placing it in a different square and reconsulting the crystal ball. In the end, she can leave the piece in the empty square that gives her the best score.

If we can define this score by induction, then this definition can be turned into the basis of a recursive algorithm to compute it. Let us define it by induction on the number of open spaces left on the board:

Base case: Suppose all spaces have been filled.

1. If both White and Black have three in a row, the state score is undefined, as this can't happen in a game. (The method can return a score of either 1 or -1 in this case, but the score is meaningless.)
2. If only White has three in a row, the score is 1.
3. If only Black has three in a row, the score is -1.

4. If neither has three in a row, the score is 0.

Induction step: Suppose there are $n > 0$ empty spaces.

1. If both White and Black have three in a row, the state score is undefined, as this can't happen in a game. (The method might return 1 or -1, but this is meaningless.)
2. If only White has three in a row, the score is 1.
3. If only Black has three in a row, the score is -1.
4. If neither has three in a row, then:
   (a) If it's White's move, then the state score is the maximum of the n state scores that result when white tries out each of his n possible moves. (Each of these will have at most $n - 1$ open spaces, and be a Black move.) By induction, each of these state scores can be found with a recursive call each of on these n board configurations.
   (b) If it's Black's move, then the state score is the minimum of the n state scores that result when black tries out a move in each open square. It can be found out with a similar use of n recursive calls.

1.1 Debugging

Once again, if your program makes a mistake, then there is some smallest recursive call where it makes a mistake. Look at the configuration where the program makes its first dumb move. Then try to exonerate the n recursive calls it makes to figure out its move by checking that their preconditions are met, stepping over them, and seeing whether their postconditions are met.

If you find a culprit call, then you've found a smaller case where your program messes up. Adjust your input board to turn it into this case, by adding a move, and repeat this process with the color of the player whose move is next reversed. Eventually, you will get down to a base case that messes up, or an inductive case where all recursive calls have worked correctly, but the code in the induction step messes up. Now it will be easy to find the problem using the debugger or by studying the code.