For this assignment you will be implementing a specific type of bounding backtracker: a minimax algorithm with alpha/beta pruning. This algorithm will be used for two player game playing. The specific game you will be implementing is TicTacToe. There is a specific design you must follow for this assignment, outlined below. Note that you may receive up to 3 extra credit points for additionally implementing a connect four game (note the restrictions given below).

Order of coding and testing

Note that this is a complex project with many classes to write. It is important to code it in an order that will make it easy to debug (or easier at least!). My suggestion is the following:

Start with the TwoPlayerGameBoard interface. This is trivial since I basically give you the entire interface.

Next code the TicTacToeBoard class. Specifically, code just the constructors, hasMoreChildren, nextChild, staticEvaluation, and toString methods. Now, before you continue onward, test these methods to make sure they work. That is, create a TestCase of some sort that creates a new TicTacToe board and then ask it for children or evaluations, etc. Make sure it works before you continue onto to the next part. You will probably have a hard time using asserts, but at a minimum you can print out the resulting children to make sure they are correct. The advantage of a TestCase is that you can save the tests to be able to rerun them later. However, the TestCase is not required for this assignment – just a suggestion.

Next, create the MiniMax class. This class uses the functions you have already coded in the TicTacToe class in order to select the child that is the best move. My suggestion is to forget about alpha/beta pruning at the start and simply write code that finds the best child by expanding all states without bound pruning. Once you are convinced you are getting the correct child you should add the alpha/beta pruning into the code and ensure you are getting the same answer. Note that in order to make your testing job easier you might want to temporarily change the default constructor of your TicTacToeBoard to construct a board near the bottom of the state space rather than a blank board. That is, a construct a board with most of the positions already filled in. This will make it easier to see if your MiniMax is working.

Up to this point, I have suggested that you create no graphics. Graphics are nice visually, but debugging is often easier with plain text output where you can examine the intermediate steps rather the just see the final output. But, now it is time to add the GUI. Create the GamePlayer and the BoardPanel. Note that the BoardPanel will need to call the draw method of your TicTacToe class. Thus, you also need to fill in this method. And, at the same time, you should fill in the placeUserMove method because it needs to be consistent with the draw method. Attach the methods to the menu items so you can create your initial TicTacToeBoard. Attach the mouse method to the board so you can get input from the user. It is in this method that you would update the board, create a MiniMax, and update the board again with the computer move. I would not add the Thread stuff until after you have the basic back-and-forth game play working. The Thread simply makes it visually more appealing and can be easily added up after you have the rest working.

Then, once you have everything else working, create a ConnectFourBoard.
That is my suggested path though the project. Following it should maximize your point total. Below you will find more detailed description of the various classes.

**TwoPlayerGameBoard Interface**

The following is the TwoPlayerGameBoard Interface:

```java
// Methods used in MiniMax
public boolean hasMoreChildren(); // returns true if the board has more children
public TwoPlayerGameBoard nextChild(); // returns the next feasible child board (note that we have
  // combined nextChild and isFeasible into this method)
public double staticEvaluation(); // returns a static evaluation of the current board (-1 loss, +1 win, 0 draw)

// Methods used for drawing
public void draw(Graphics g); // draws the game board on the given graphics object

// Methods used for game play
public boolean isComputerWinner(); // returns true if the computer is a winner
public boolean isDraw(); // returns true if the game is a draw
public boolean isUserWinner(); // returns true if the user is a winner

// Methods used for handling user input
public void placeUserMove(Point2D mouseLocation) throws Exception; // modifies the current problem to
  // place a user’s piece on a particular location on the board indicated by the user clicking with their
  // mouse. If the user’s click was an invalid position, the method will throw an Exception (this
  // should probably be a specific type of exception, but a generic Exception will do for now).
  // Note that this method will also need to change any instance variables in the class to indicate that
  // it is now the computer’s turn.
```

The above interface is logically broken up into 4 sections (although it is still one interface). The first 3 methods are used for the MiniMax algorithm. The next method is used to display a view of the board. The next 3 are used to control the game player. And the final method is used to handle user input.

You will, of course, need a concrete implementation of this interface. TicTacToeBoard and ConnectFourBoard provide these implementations.

**GamePlayer Class**

The GamePlayer class is what displays the GUI and controls the back and forth game play between the human and the computer. The basic GUI should look like the pictures below.

First the GUI needs a couple of menus. There should be a menu for starting a game (TicTacToe or ConnectFour) and a menu for selecting the difficulty level (easy, medium, hard).
There should also be a large area where the game is to be displayed. Note that the game will actually be displayed using a boardPanel class that overrides the paintComponent method (see below). For my examples, I picked the size of this panel to be about 700x600 – however, you can pick whatever size you wish.

The GamePlayer will need to keep track of the current TwoPlayerGameBoard. This TwoPlayerGameBoard is first created when the user selects one of the menu options from the Game menu. Thus, you will need to attach actionPerformed methods to these menu items. It is in these actionPerformed methods where the starting TicTacToe and ConnectFour boards are initially created.

The GamePlayer should also contain code that alternately allows the user and computer to move, displaying the updated board after each step. So we need code to process user moves, code to compute the computer moves and code to redisplay the new state of the board.

In order to process user moves, the GamePlayer should attach a mouseReleased listener to the BoardPanel it has created. This will allow the GamePlayer to watch for the user releasing the mouse. The attached method is called by the system when the user selects a new position for their piece. It should try to place the user’s piece at the location specified by the mouse location. This is done by calling the TwoPlayerGameBoard’s placeUserMove method. The TwoPlayerGameBoard does this work instead of
the GamePlayer, because the GamePlayer doesn’t know about the specifics of the actual board being used (i.e. it could be a TicTacToe board or it could be a ConnectFour board). If the placement was successful (i.e. no exception was thrown), it should update the board on the screen by telling the boardPanel about the modified board, and then let the computer move by processing the current board with the minimax algorithm. This, of course, involves creating a new MiniMax class and making a call to its generateNextMove method (see below for details of the MiniMax class). After the computer has finished its move, it should then display the results on the screen so that the user can move again. Note that there are issues here involving Threads (see below). That is, if the MiniMax processing takes a long time, the user’s move won’t show up on the screen until after the computer has moved. This is probably not what we want, so we use a Thread to fix the situation.

Note that when the MiniMax code is called, you can pass the maximum level to which the state space tree can be expanded. This controls how smart the computer is (it also control how long the computer takes to pick its move). The number you pass to MiniMax should be set by the Difficulty menu options. Easy → 2, Medium → 5, Hard → 8. You should setup actionPerformed method on these menu items to set the maxLevel variable.

**BoardPanel Class**

A boardPanel is just like the other JPanels we have used in this class. It should extend JPanel and override the paintComponent method. Additionally, it should have a method called:

```java
public void setBoard(TwoPlayerGameBoard newBoard)
```

that tells it to display a new gameboard. The GamePlayer is in charge of telling the BoardPanel when to display a new board. The BoardPanel is just in charge of redrawing the current board on the screen.

Note that the work of paintComponent is quite simple – it simply needs to tell the TwoPlayerGameBoard to draw itself. Additionally, it should check the status of the TwoPlayerGameBoard to see if the game is over. If the game is over, it should display an appropriate message.

**MiniMax Class**

The MiniMax class should have one public method, besides the constructor:

```java
public TwoPlayerGameBoard generateNextMove(TwoPlayerGameBoard currentProblem)
```

This method will take in a TwoPlayerGameBoard and return one of its children. The child it returns is the one where the computer’s move is maximized. If there is a tie, it can return any of the methods that tied for the best. For debugging purposes it is best if you always return the first (or last) of any of the tied states. From a gaming standpoint picking a random state from a set of tied states makes the game more fun.

Note that you will also need a recursive method:

```java
private double recursiveMiniMaxAlphaBeta(TwoPlayerGameBoard currentProblem, int currentLevel, double alpha, double beta)
```

For certain games (e.g. connectFour, checkers, chess) you will be unable to advance all the way to the solutions because of the explosion of the number of states that you would need to look at. Because of this, it is a good idea to be able to specify a maximum level (stored as an instance variable in my code). If, in your processing of the game tree, you reach the max level before you reach a solution then you just run the static evaluation on this level and backtrack. For TicTacToe our static evaluation for boards that are not
done is fairly bad (it always returns a 0), but for a game like chess you would provide a more useful evaluator based on your expert knowledge of chess.

When you are debugging, it is useful to not start with a blank board as the tree is just too deep and creates too many states to work through. I would suggest starting with a fixed board closer to the bottom of the tree. The following starting board is a good example as it has many different options:

```
X X O
- - -
X O O
```

The alpha/beta pruning works like I described in class. That is, alpha and beta are the two bounds on the best costs we can obtain. Alpha is a lower bound and Beta is an upper bound. Thus, alpha can start at negative infinity and beta can start at positive infinity. As the state space is examined alpha will increase and beta will decrease. When they cross (i.e. alpha > beta) that means pruning can occur. The basic algorithm pseudocode is as follows:

If level is a minimizing level:
   As long as we have more children and alpha is still smaller than beta
      Evaluate the next child, passing it the current alpha and beta
      If the value returned from the child is smaller than beta, replace beta
   Return beta as our cost
If level is a maximizing level:
   As long as we have more children and alpha is still smaller than beta
      Evaluate the next child, passing it the current alpha and beta
      If the value returned from the child is larger than alpha, replace alpha
   Return alpha as our cost

**TicTacToeBoard Class**

TicTacToeBoard is to provide a concrete implementation of the TwoPlayerGameBoard interface. You will certainly need to keep track of the current board as an instance variable. And, just like with your GraphColoringProblem, you will need to keep some information around that keeps track of which children you have already given out, such as a nextOpenPosition variable.

My suggestion is that you represent the board as a single array (i.e. [] ) of size 9. The board is never going to grow or shrink in size so arrays are an acceptable representation. nextOpenPosition is then nothing more than an int index into that array. Additionally, you may wish to have a boolean that keeps track of the next player to move (human/computer).

You will certainly want to have a default constructor that creates the initial board. You might also want to have a copy constructor that can be used by the nextChild method to create a copy of itself to use as the starting point for the child creation. Note that we didn’t do this in the lab. In the lab, we just created a new default child and did the coping in nextChild. The copy constructor is probably a better way to proceed. The nextChild, hasMoreChildren, and staticEvaluation methods are used in the MiniMax algorithm to pick the computer’s next move.

The isUserWinner, isComputerWinner, and isDraw method are used by the GamePlayer and BoardPanel to control the game play. Note that there are some common things that these three methods and the staticEvaluation method need to do. Feel free to add private helper methods into this class to make your coding easier.

Note that the placeUserMove and draw methods need to be consistent with each other so that the board that the board coordinates that are drawn on the screen are the same coordinates used in deciding which position of the board the user clicked into.
ConnectFourBoard Class

This is exactly like the TicTacToeBoard class, except that it implements the game of connect four as opposed to TicTacToe. It must also implement the TwoPlayerGameBoard interface. Recall that connect four has a board that is 7 horizontal by 6 vertical. Someone wins if they connect 4 of their pieces together in a row, column, or diagonal.

This class is completely optional. If you implement it correctly you will receive 3 extra credit points. HOWEVER, you will not receive any points for this class unless the rest of your code is working for TicTacToe. That is, this class is for extra credit if you already have finished the entire TicTacToe part of the assignment. It cannot be used to try and get 3 points to offset something not working in your TicTacToe code.

ComputerThinkingThread Class

The ComputerThinkingThread is a class that implements the Runnable interface. This class is used to put the computation of the computer’s move (i.e. the minimax algorithm) into its own thread. In this way, the event handler can return immediately after processing the user’s move (so that the user’s move can be drawn on the screen). Eventually, the computer will generate a move for itself. Once this is done, it needs to tell the GamePlayer about the new move. Thus, GamePlayer should have an additional method called:

public void computerDone(TwoPlayerGameBoard computerMove)

and ComputerThinkingThread should have access to the particular GamePlayer that created it in the first place. Thus, when the computer has finished its move, it can tell the GamePlayer about the new board via the computerDone method.

You should also prevent the user from making additional moves until the computer is done with its move.

Submission:

- A document describing any problems with your code
- A jar file containing your code so I can run it

You may work with a partner.