Contents

1 Introduction

Machine learning is a subfield of artificial intelligence that focuses on engineering algorithms that are able to learn, make predictions, and eventually make informed decisions. These algorithms are rooted in statistical analysis performed on some set of data and have a vast array of different applications. Some examples of the diversity of machine learning applications include bioinformatics, adaptive web application design, cyber-fraud detection, stock-market analysis, and robotics [12].

1.1 Categorizing Machine Learning Algorithm

Machine learning algorithms are typically separated into two main categories based on the different learning styles used in the approach: supervised and unsupervised. Supervised learning involves having the computer learn from a labeled training data set. Each element of the data set would therefore consist of two parts: an input object, often represented as a vector, and the output value, often referred to as a supervisory signal. On the other hand, unsupervised learning algorithms attempt to organized data points which have no associated labels in such a way that the algorithm is able to make some meaningful conclusions. A third type of machine learning approach, reinforcement learning, is in some ways a combination of supervised and unsupervised learning approaches. Reinforcement learning, often used in robotics and other agile applications, relies of reward (or punishment) signals to serve as an indication of the success of an action made in response to a particular data point. With these reward signals, the algorithm can then updated its decision making policy such that given a particular data point it will choose the action that will maximize the reward, and overall success of the algorithm. This project explores how different types of machine learning algorithms, namely reinforcement and supervised learning techniques, can be applied to perfect information zero-sum board games, and attempts to draw some basic comparisons about these approaches and their results of their implementations [3].

1.2 Game Theory

In game theory perfect information game is a deterministic game in which the players all have access to all of the information about the current state of the game. Therefore, things like tic-tac-toe, connect four, and chess would all be perfect games but a game like poker, in which the opponents cards remain unknown to the player, is not. In a zero-sum game, the success of each player is directly counterbalanced by the failure of the opposing player [1].

2 Reinforcement Learning

Reinforcement learning algorithms are a variation of traditional unsupervised learning techniques, where the algorithm is able to learn from an interactive environment by aiming to maximize long term rewards [4]. Reinforcement learning algorithms involve mapping certain game states to decisions which would be most advantageous to the player. So rather than explicitly telling the learner which actions to take, as is done with most other forms of supervised learning, reinforcement learning involves the learner discovering which actions lead to the greatest reward for it [3]. Reinforcement algorithms traditionally consist of four main components: the state set, s, the action set, a, the rewards reaped from the environment, R, and state-action pairs, V. With board game applications, different "pre-state" actions can lead to the same "after-state". The traditional reinforcement paradigm then can simply be altered to prevent the redundancy of many state-action pairs inherent in deterministic board game applications by slightly altering the traditional parameters. Instead of having a distinct state set and action set, we can simply have a combined "after-state" set, S [10]. The algorithm described in this section is based on a variation of the temporal-difference method, in which state values are approximated by maintaining updated values of the states visited in each training game. This is represented as

$$V(s) \leftarrow V(s) + a[V(s') - V(s)$$

where s represents the current state, s' represents the next state, V(s) represents the value of state s and is the learning rate, which varies within $(0, 1]$ [5]. Thus if one of the following conditions are met, we would say that the state value converges [5].

$$\lim_{T\to\infty} \sum_{t=0}^{T} a_t = \infty \qquad \lim_{T\to\infty} \sum_{t=0}^{T} a_t^2 < \infty$$

Strategy selection during this training phase is typically based on the Boltzman's distribution. [5]

$$\Pi(s) = Pr(s) = \frac{e^{v(s)/\tau}}{\sum_{i=1}^{n_s} e^{v(s_i)/\tau}}$$

In the above expression, n represents the number of after states and τ represents the temperature. A high temperature, indicates that the probability destitution is nearly uniform. The lower the temperature however, the more greedy the selection strategy is. The final strategy π^* is therefore found using the following expression [4]

$$\pi^* = argmax\ v(s_i)$$

2.1 Tic-Tac-Toe

This process can be illustrated with an implementation of a simple game of tic-tac-toe. A standard 3x3 tic-tac-toe game board contains nine locations, each of which may either contain an 'X', contain an 'O', or be empty at a point given in the game.

2.1.1 Implementation Details

The approach taken in this implementation is a slight variation on the classic reinforcement learning model described earlier. For the purposes of this implementation the current state of the game is represented with a multidimensional vector consisting of 'X's 'O's and spaces corresponding to the contents of each location at that point in the game. So for example, the board in Fig. 1 would be represented as $s = [X, O, O, O, X, \ , \ , X, \ ]^T$. This reinforcement learning framework, as with many

X	O	O
O	X	
	X	

Figure 1: **Tic-Tac-Toe Board State.**

other machine learning approaches, consists of two primary phases: the training phase and game-play. In the learning phase the algorithm progressively gains new information about the best decision making strategies. With each iteration, the algorithm notes the current state, s and relies on the distribution of available state-values of the potential next moves, or $V(s')$, in order to make a decision. With this information, the algorithm adds s' to a sequence vector. This process repeats until the game has ended at which point the algorithm updates the values of all states visited in the sequence and is rewarded or punished based on the outcome of the game [5]. After the learning phase, comes actual gameplay. At this point, the algorithm no longer learns new information and instead relies on information gathered in the previous phase. So, the algorithm simply observes s, the current state of the game, and makes a move based on distribution of $V(s')$.

The proposed implementation of this algorithm maintains two parallel vectors. The first contain the states of the board, which are continually added as the computer encounters an un-tracked state. There is a second vector contains the strategy groups corresponding to particular states that enable the learner to make an informed decision about the best move to make. Initially the learning player has an equal chance of choosing any of the available locations for a particular state. As the algorithm progresses, it keeps track of the success or failures associated with a

particular decision and is either rewarded or punished by altering the make-up of the strategy group vector such that the probability of making the same decision if the computer encounters that state again will either be increased or decreased. After repeated iterations of this game, allowing the computer to play hundreds thousands of games it is able to make near-perfect decisions in the future [5].

2.1.2 Results

We performed a series of experiments using the implementation of the algorithm we developed. These experiments demonstrated a relatively large margin of success.During the training phase of the algorithm, we allowed the learning algorithm to play 1,000,000 games against a randomized opponent, in which time the algorithm collected data about the different board states, the potential moves, and the outcomes from previous moves. In total this process took 388.869 seconds, or about $6\frac{1}{2}$ minutes. Each game took an average of 0.388869 milliseconds to complete. Throughout this phase we also keep track of the number of games won. With this information we were able to determine that the algorithm completed the learning phase, where it initially started off with zero knowledge of the game and eventually "learned" how to make smart decisions, with a 92.69% success rate. We were able to write all information obtained by the algorithm during the learning phase to a separate file that we could then import into the game-play phase of the algorithm. At this point, we allowed the, now intelligent algorithm, to again play 1,000,000 games against a randomized opponent. Again each game took an average of about 0.342055 milliseconds and after playing 1,000,000 games we found that our algorithm had a 93.14% success rate [11].

2.1.3 General Remarks

This method relies on accumulating a list of encountered states and eventually enumerating all of the possible states because there is no explicitly defined and quantifiable reward function. Although it is quite effective in this particular application, it was quickly evident that it is incredibly memory expensive and may not be the most efficient approach to tackle more complex problems. While the previously discussed method was quite effective for a game as simple as tic-tac-toe, the same approach of documenting decisions made at each potential board state is not scalable to any reasonably complex problem.

3 Supervised Learning

Supervised learning algorithms often attempt to foresee the possible consequences of a particular action and make decisions that would be most advantageous. Of particular interest in this particular project is a flavor of supervised learning algorithms which utilize structures called decision trees, through which potential decisions are mapped

out in a tree-structure. This tree then serves as the algorithm's predictive model, mapping observations made about some data to some conclusions about the data's target value. In general, tree structures are a widely used data structure in computer science and are an integral component of many different types of algorithms and applications. The figure below is an illustration of a generic tree structure. Fig. 2 each node of the tree contains some value and relationships between each of the nodes are hierarchical in nature.

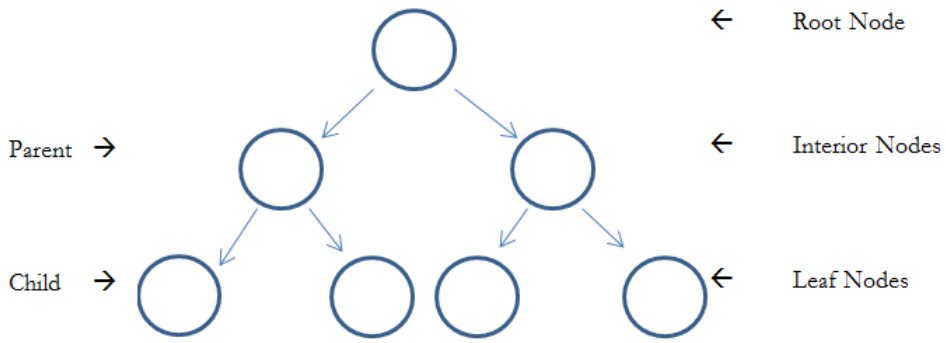

Figure 2: **General Tree Diagram.**

3.1 Game Trees

This expansion of into a tree structure, in game applications, is often represented as a game tree. Game trees are the standard representation of all of the possible moves in-perfect information games. Fig. 3 illustrates a general game tree. With any reasonably complex game, the full game tree is much too large to feasibly generate in its entirety. The computational cost even for a game as simplistic as tic-tac-toe is enormous. With the nine possible positions of a standard tic-tac-toe we can calculate that there are a total of 986410 unique states. [13]

$$\sum_{0}^{n=9} \frac{9!}{(n+1)!} = 986410$$

While not impossible for a computer to store and process this much information, it is alarming just how computationally expensive such a simple game is. There needs to be a way to utilize the game tree without such great cost. One possible approach is to simply enumerate all the possible moves. This is not enough however, as simply looking toward the next move may not always result in ending the game. So in order

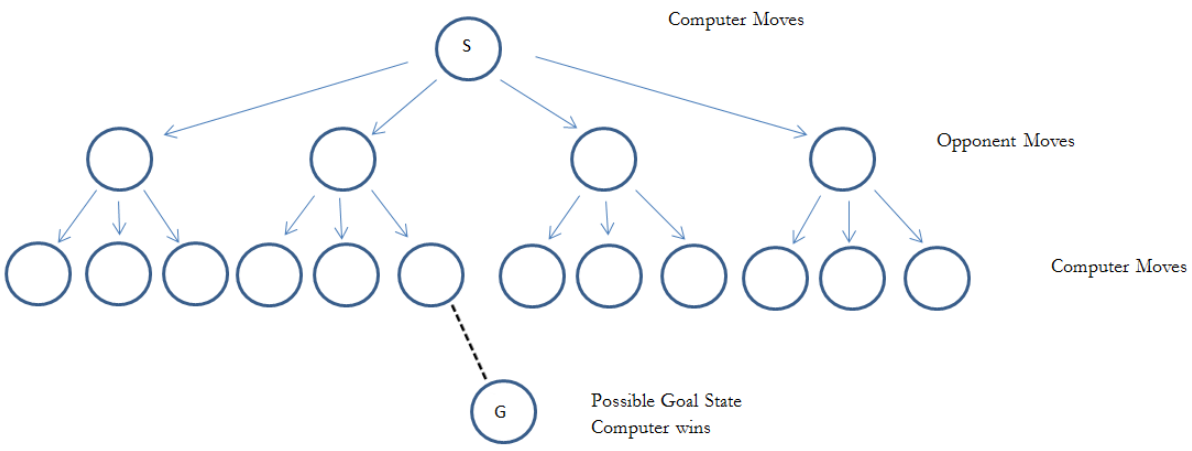

Figure 3: **General Game Tree Diagram.**

to really make use of the game tree in this case one would have to recursively generate all the potential moves at each turn prior to the game's termination. Making use of an algorithm known as the minimax algorithm enables the learning player to utilize the concept of a game tree to make relatively intelligent decisions regarding game play [13].

3.2 The Minimax Algorithm

One possible approach to implementing this algorithm is to simply enumerate all the possible moves. This is not enough however, as simply looking toward the next move may not always result in ending the game. So in order to really make use of the game tree in this case one would have to recursively generate all the potential moves at each turn prior to the game's termination and based off this somehow determine the best possible move the machine could make. This is accomplished through the use of an additional heuristic function, which is used to evaluate the desirability of a certain game state. A generic evaluation function would look like

$$evaluate\,(position)\ = \sum_i w_i\ *\ f_i(position)$$

At the end of each game, there are then three possible outcomes: the machine will win, the machine will lose, or the game will end in a draw. Say for example that when we calculate the score we assign weights of 1, -1, and 0 respectively to each of these outcomes, the machine will then be trying to maximize the score, while the opponent would hope to minimize the score. It is then simple to assign a score

to terminal states. The challenge becomes how to handle the intermediate states. Operating under the assumption that our opponent is a perfect player, we assume each player selects the action that results in the highest gain and can calculate the score accordingly by recursively looking ahead to the terminal state and propagating their values upwards throughout the tree as illustrated in the figure below. The minimax algorithm follows this process, one subtree at a time. It therefore is able to recursively score each subtree and return the best of them, indicating the most advantageous action that the algorithm should take given the current state [13].

3.2.1 Depth of Game Tree in Minimax Applications

With applications of the minimax algorithm, the tree grows exponentially with respect to the depth to tree depth, so the algorithm will run on a tree of depth d in $O(2^d)$ time. It is therefore too computationally expensive to run the minimax algorithm on the full tree, even when considering very simple games. It is therefore standard to only expand the tree up to some maximum depth. When dealing with non-terminal states, we have to assign some value based on our heuristic function [7].

Algorithm 1 *The Minimax Algorithm [13]*

```
1. Expand the game tree as far as possible
2. Assign state evaluations at each open node
3. Propagate upwards the minimax choices
     If the parent is a Min node (opponent)
         Propagate up the minimum value of the children
     If the parent is a Max node (computer)
         Propagate up the maximum value of the children
```

3.3 Connect Four

Connect four is a two player game in which players take turns placing different colored disks into a 6x7 game board by selecting a column and allowing the disk to fall to the lowest open position in that column. Players continue with the end goal of getting four of their pieces in a row either horizontally, vertically of diagonally [6]. Connect four, like tic-tac-toe, is a solved game, which means that at any given point the outcome of the game can be perfectly predicted under the assumption that each player always makes the move which is most advantageous to themselves. There is some dispute among game theory experts and scholars as to how many possible positions. Kissman and Tromp calculated a total of 4,531985,219,092 possible configurations of the board [8] [14]. While Allis estimated that there was more like 70,728,639,995,483 possible states. In his master's thesis he also proved that with two perfectly playing players,

the player who makes the first move will always win [2]. Regardless, it is plainly evident that any type of brute force is not an option for such a game

3.3.1 Implementation Details

In this implementation the current state of the board is stored in a two-dimensional array of characters. Each element of thee array represents the possible values at each position of the board: it can contain a red piece ('R'), a yellow piece ('Y') or be empty. Each move the computer makes is based off of the minimax algorithm described in the previous section. At each step the algorithms constructs a game state tree. As it is not computationally efficient to expand the game tree in its entirety, the depth of the tree is constrained by an additional parameter. The next step is to evaluate each state by assigning them values which indicate the best moves. First the leaves of the tree are evaluated. The higher the score assigned to a state by the evaluation function, the more advantageous it is for the computer. If the state leads to the computer getting four consecutive pieces, then the state is assigned the score of ∞, represented by the maximum integer value. On the contrary, if a state leads to the opponent getting four consecutive pieces then it is assigned the score $-\infty$, represented by the minimum integer value. Based on this information, the algorithm evaluates the parent nodes. With all of this the computer is then able to act in such a way that it minimizes the opponents score and maximizes its own score. The value of each of the parent states can then easily be determined. If it is currently the computer's turn at a given state, it uses the maximum value from the list of children as the current state's value. On the other hand, if it is the opponents turn at a given state, it will use the minimum value from the list of children.

3.3.2 Results

We tested the effectiveness of our implementation of the minimax algorithm [11] at with different maximum game tree depth values. Using varying maximum depth values, we ran the algorithm on 10,000 games and calculated the success rate. The results of the experiment are included in Table. 1

3.3.3 Discussion

As expected, the deeper we expand the tree for each iteration the longer it takes to run each game. The performance gains from increasing the depth are minimal for this application as a game tree with depth 1, looking ahead one move and making a decision based on the score assigned to that given state, is sufficient enough to yield more than a 98% success rate when playing against a randomized opponent. However as the depth is increased the success rate increases with one notable exception, when the tree is expanded to depth 2, the algorithm's success rate dips to approximately 96.6%. The most likely explanation for such an anomalous decrease in an otherwise

Max Tree Depth	Total Experiment Runtime	Average Time Per Game	Success Rate
1	25.628s	2.5628ms	98.07%
2	40.564s	4.0564ms	96.60%
3	45.502s	4.5502ms	98.71%
4	158.887s	15.8887ms	99.44%
5	1057.155s	105.7155ms	99.82%
6	5673.049s	567.3049ms	99.76%
7	10.789hr	388.924ms	99.98%
8	75.254hr	27091.504ms	100.00%

Table 1: Results after 10,000 games

increasing curve, would be that when considering two moves in advanced we are not given enough information about the future of the game. This leads to a slight increase in the instances of unsuccessful actions.

4 Conclusions

The aim of this project was to survey some different machine learning approaches to apply to deterministic board games. This provides a tangible and scalable application of complex algorithms. With the approaches described we were able to dissect the reasonably complex problem of "teaching" the computer to use reasoning to make intelligent decisions while playing games designed for humans, wnd were able to develop successful algorithms to do so.

4.1 Future Work

While this work developed a strong foundation in classical machine learning algorithms and demonstrated the diversity of approaches that can be taken to achieve the goal of intelligent gameplay without explicit programing, there is still much more that can be explored on the topic. Though we saw great success in implementing these algorithms for Tic-Tac-Toe and Connect-Four, it would be prudent to expand the scope of the project further, applying these techniques to other types of games, and potentially to non-game applications. Also, while we addressed some of the fundamental approaches taken in classical machine learning, it would interesting to explore different, more contemporary and experimental techniques. Finally, as previously discussed, many of these algorithms present a computational bottleneck particularly in the training phase, which limits the efficiency and overall performance of these algorithms. Another interesting area of exploration would be the application of high performance computing techniques, such as parallel programming, to these problems in an attempt to mitigate the effects of the inherent computation complexity

A
Tic Tac Toe Source Code

```
/**
 * Colleen Rogers
 * Fall 2015
 * Machine Learning Applications: Intelligent Connect Four
 * About Connect4:
 *
 *
 *  Implementing Machine Learning Algorithm for Connect4:
 *
 *
 *  Uses the Minimax Algorithm in conjunction with a basic heuristic
     function I developed
 *  Can change depth of the algorithm; Will affect performance and speed.
 *
 *  https://github.com/crogers719/Machine-Learning
 *
 */
import java.util.Scanner;



/**
 *BOARD CLASS:
 *Contains board (a 2-dimensional character array that holds the board in
    its current state)
 *Performs basic functions pertaining to alteration of the board itself
 *
 *
 */
class Board{

   //Standard Dimensions of a Connect four board (6x7)
   public static final int NUM_ROWS = 6;
   public static final int NUM_COLS = 7;

   //Player1 ('Red') is the one using the AI algorithm
   public static final char COMPUTER='R';
   //Player2 ('Yellow') is a naive opponent; Can either have (random)
      automated moves, or can be controlled by a user
   public static final char OPPONENT='Y';
   //Represents an empty cell
```

```
public static final char EMPTY=' ';


char[][] board = new char[NUM_ROWS][NUM_COLS];

/*
 * Clear board sets all positions in board equal to default value
    (EMPTY=' ')
 */

 public void clearBoard(){
   for (int r=0; r<NUM_ROWS; r++){
      for (int c=0; c<NUM_COLS; c++){
         board[r][c]=EMPTY;
      }
   }
 }

 /*
  * Will display the contents of the game board in its current state
  */
    public void displayBoard(){
        System.out.print("\n");
        for(int r=0;r<NUM_ROWS;r++){
            for(int c=0;c<NUM_COLS;c++){
                System.out.print(board[r][c]+" |");
            }
            System.out.print("\n-----------------------\n");
        }
        System.out.println();
    }

 /*
  * Given a column number, will place corresponding character (player)
  * in the highest (farthest down) empty row in that column
  */
 public boolean placePiece(int column, char player){
     if(!validMove(column)) {
      System.out.println("Illegal move!");
      return false;
     }
     for(int r=NUM_ROWS-1;r>=0;r--){
         if(board[r][column] == EMPTY) {
             board[r][column] = player;
             return true;
```

```
            }
        }
        return false;
    }


    /*
     * Valid Move determines whether a move (choice of column) is valid
     *   Checks that the column number is between 1-7 and that the chosen
         column is not full
     */
    public boolean validMove(int column){
      if(column>=0 && column<NUM_COLS && board[0][column]==' '){
         return true;
        }
        else{
            return false;
        }
    }



    /*
     * Given the column number will remove the last piece inserted into
         the column
     * Useful for undoing moves made as part of the Minimax algorithm
     */

    public void undo(int column){
        for(int r=0;r<NUM_ROWS;++r){
            if(board[r][column] != EMPTY) {
                board[r][column] = EMPTY;
                break;
            }
        }
    }

}
/**
 * CONNECTFOUR CLASS
 * Contains:
 *         Constants for: board dimensions, characters, number of games,
    and autoplay settings
 *         An object of the board class to hold the game board,the next
    move and, the maximum depth to be used in Minimax
```

```
 * Performs:
 *         Machine learning algorithm that enables the intelligent player
    (COMPUTER) to make advantageous
 *         decisions using the Minimax Algorithm
 */

public class ConnectFour {
   //Standard Dimensions of a Connect four board (6x7)
   public static final int NUM_ROWS = 6;
   public static final int NUM_COLS = 7;

   //Player1 ('Red') is the one using the AI algorithm
   public static final char COMPUTER='R';
   //Player2 ('Yellow') is a naive opponent; Can either have (random)
      automated moves, or can be controlled by a user
   public static final char OPPONENT='Y';
   //Represents an empty cell
   public static final char EMPTY=' ';

   //Number of consecutive games to be played
   public static final int NUM_OF_GAMES=250;
   //If true, will randomize moves; Otherwise a user can play against the
      machine
   public static final boolean AUTOPLAY=true;


    private Board gameBoard;
    private Scanner kbd;
    private int nextMove=-1;

    /*The maximum depth used by the min-max algorithm--need cut off point
    * Tested for values between 1-8(depth =1 --> fast with about 97%
       success rate
    * depth =8 -->slower but with a near 100% success rate
    */
    private int maxDepth =7;

    /*
     * ConnectFour constructor
     *         Takes a Board object, b, as a parameter and assigns
        gameBoard=b
     *         Sets up a Scanner object (kbd) to be used in class functions
     */

    public ConnectFour(Board b){
```

```
    gameBoard = b;
    kbd = new Scanner(System.in);
}



/*
 * Allows the opponent to make a move. If AUTOPLAY is on will simply
    generate a number between 1-7(NUM_COLS)
 * Otherwise it will take input from the user for a column number (1-7)
 * Then calls the place piece method (in Board class) to place the
    piece at the position corresponding to the
 * column number obtained.
 *
 */
public void opponentMove(){
  int move=-1;
  if (AUTOPLAY){
     do{
        move= (int)(Math.random()*NUM_COLS) + 1;
     }while (!gameBoard.validMove(move-1));
  }
  else{
     System.out.println("Enter a Column (1-7): ");
     move = kbd.nextInt();
     while(!gameBoard.validMove(move-1)){
        System.out.println("Invalid move.\n\nEnter a Column (1-7): ");
        move = kbd.nextInt();
     }
  }
    //call to place piece (in Board class); this assumes that player 2
       is the opponent
    gameBoard.placePiece(move-1, OPPONENT);
}



/*
 * Finds the results of the game by checking the gameBoard for a winner
 *     Four ways to win:
 *          Horizontally
 *          Vertically
 *          Diagonal#1 (lower left->upper right)
 *          Diagonal#2 (lower right->upper left)
 *     If finds a winner, returns the character associated with
    winning player.
```

```
 */
public char findWinner(Board gameBoard){
    int countComputer = 0;
    int countOpponent = 0;
    for(int r=NUM_ROWS-1;r>=0;r--){
        for(int c=0;c<NUM_COLS;c++){
           if(gameBoard.board[r][c]==EMPTY){
              continue;
           }

            //Check For Horizontal Win
            if(c<=3){
                for(int incr=0;incr<4;incr++){
                        if(gameBoard.board[r][c+incr]==COMPUTER) {
                          countComputer++;
                        }
                        else if(gameBoard.board[r][c+incr]==OPPONENT) {
                          countOpponent++;
                        }
                        else{
                          break;
                        }
                }
                if(countComputer==4){
                 return COMPUTER;
                }
                else if (countOpponent==4){
                 return OPPONENT;
                }
                //No winner-reset counters
                countComputer = 0;
                countOpponent = 0;
            }

            //Check for vertical win
            if(r>=3){
                for(int incr=0;incr<4;incr++){
                        if(gameBoard.board[r-incr][c]==COMPUTER) {
                          countComputer++;
                        }
                        else if(gameBoard.board[r-incr][c]==OPPONENT) {
                          countOpponent++;
                        }
                        else{
                          break;
```

```
            }
        }
        if(countComputer==4){
         return COMPUTER;
        }
        else if (countOpponent==4){
         return OPPONENT;
        }
        countComputer = 0;
        countOpponent = 0;
    }

    //Check for diagonal (up-right) win
    if(c<=3 && r>= 3){
        for(int incr=0;incr<4;incr++){
            if(gameBoard.board[r-incr][c+incr]==COMPUTER){
               countComputer++;
            }
            else if(gameBoard.board[r-incr][c+incr]==OPPONENT) {
               countOpponent++;
            }
            else{
               break;
            }
        }
        if(countComputer==4){
         return COMPUTER;
        }
        else if (countOpponent==4){
         return OPPONENT;
        }
        countComputer = 0;
        countOpponent = 0;
    }

    //Check for diagonal (up-left) win
    if(c>=3 && r>=3){
        for(int incr=0;incr<4;incr++){
            if(gameBoard.board[r-incr][c-incr]==COMPUTER){
               countComputer++;
            }
            else if(gameBoard.board[r-incr][c-incr]==OPPONENT){
               countOpponent++;
            }
            else{
```

```
                        break;
                    }
                }
                if(countComputer==4){
                 return COMPUTER;
                }
                else if (countOpponent==4){
                 return OPPONENT;
                }
                countComputer = 0;
                countOpponent = 0;
            }
        }
    }

    for(int c=0;c<NUM_COLS;c++){
        //There is still an empty location-Game has not ended yet
        if(gameBoard.board[0][c]==EMPTY){
           return '-';
        }
    }
    //No winner and no empty locations mean that the game is a DRAW
    return 'D';
}

/*
 * Calls Minimax, a recursive function, which will eventually change
    the value of next move
 * Then returns nextMove
 */

public int informedDecision(){
    nextMove = -1;
    int start =0;

    Minimax(start, COMPUTER);
    return nextMove;
}

/*
 * MINIMAX ALGORITHM
 * Used when expanding the full game tree is not possible
 * Aims to minimize the possible loss for worst case scenario.
 * Minimizes the loss against a perfectly playing opponent.
```

```
* Uses a a basic heuristic function to evaluate the desirability of given
    board states
*/
   public int Minimax(int depth, char turn){
       //check for winner will return COMPUTER or OPPONENT if win
     char findWinner = findWinner(gameBoard);

       if(findWinner==COMPUTER){ //computer won
        return Integer.MAX_VALUE;
       }
       else if(findWinner==OPPONENT){ //oponent won
        return Integer.MIN_VALUE;
       }
       else if(findWinner==EMPTY){
        return 0;
       }

       if(depth==maxDepth){
        return determineFitness(gameBoard);
       }

       int maxScore=Integer.MIN_VALUE;
       int minScore = Integer.MAX_VALUE;

       for(int c=0;c<NUM_COLS;c++){
           if(!gameBoard.validMove(c)){
              continue;
           }

           if(turn==COMPUTER){
              gameBoard.placePiece(c, COMPUTER);
               int currentScore = Minimax(depth+1, OPPONENT);
               maxScore = Math.max(currentScore, maxScore);
               if(depth==0){
                 System.out.println("Score for location "+c+" =
                     "+currentScore);
                  if(maxScore==currentScore) {
                   nextMove = c;
                  }
               }
           }

           else if(turn==OPPONENT){
                  gameBoard.placePiece(c, OPPONENT);
                  int currentScore = Minimax(depth+1, COMPUTER);
```

```
                minScore = Math.min(currentScore, minScore);
            }
            gameBoard.undo(c);
        }
        if (turn==COMPUTER){

         return maxScore;
        }

        else{          //turn==OPPONENT

         return minScore;
        }
    }
    /*
     * A basic heuristic function to evaluate the desirability of board
        states
     * Totals up the amount of consecutive pieces and calls calculateScore
        to
     * weigh the score based on how close player is to victory
     *
     */
     public int determineFitness(Board gameBoard){

        int computerPoints=1;
        int score=0;
        int blanks = 0;
        int offset=0;
        int countMoves=0;
        for(int r=NUM_ROWS-1;r>=0;r--){
            for(int c=0;c<NUM_COLS;c++){

                if(gameBoard.board[r][c]==EMPTY ||
                    gameBoard.board[r][c]==OPPONENT){
                 continue;
                }
                //potential horizontal
                if(c<=3){
                    for(offset=1;offset<4;offset++){
                        if(gameBoard.board[r][c+offset]==COMPUTER){
                          computerPoints++;
                        }
                        else if(gameBoard.board[r][c+offset]==OPPONENT){
                          computerPoints=0;
                          blanks = 0;
```

```
            break;
        }
        else{
            blanks++;
        }
    }

    countMoves = 0;
    if(blanks>0) {
        for(int col=1;col<4;col++){
            int column = c+col;
            for(int rw=r; rw<= 5;rw++){
             if(gameBoard.board[rw][column]==EMPTY){
              countMoves++;
             }
                else{
                   break;
                }
            }
        }
    }

    if(countMoves!=0){
       score = score+ calculateScore(computerPoints,
           countMoves);
    }
    computerPoints=1;
    blanks = 0;
}

//potential vertical
if(r>=3){
    for(offset=1;offset<4;++offset){
        if(gameBoard.board[r-offset][c]==COMPUTER){
            computerPoints++;
        }
        else if(gameBoard.board[r-offset][c]==OPPONENT){
            computerPoints=0;
            break;
        }
    }
    countMoves = 0;

    if(computerPoints>0){
        int column = c;
```

```
        for(int rw=r-offset+1; rw<=r-1;rw++){
         if(gameBoard.board[rw][column]==EMPTY){
           countMoves++;
         }
            else{
             break;
            }
        }
    }
    if(countMoves!=0) {
       score = score+ calculateScore(computerPoints,
           countMoves);
    }
    computerPoints=1;
    blanks = 0;
}

if(c>=3){
    for(offset=1;offset<4;offset++){
        if(gameBoard.board[r][c-offset]==COMPUTER){
          computerPoints++;
        }
        else if(gameBoard.board[r][c-offset]==OPPONENT){
          computerPoints=0;
          blanks=0;
          break;
        }
        else{
          blanks++;
        }
    }
    countMoves=0;
    if(blanks>0) {
        for(int col=1;col<4;col++){
            int column = c- col;
            for(int rw=r; rw<= 5;rw++){
             if(gameBoard.board[rw][column]==EMPTY){
              countMoves++;
             }
                else{
                   break;
                }
            }
        }
    }
```

```
    if(countMoves!=0){
       score = score+ calculateScore(computerPoints,
           countMoves);
    }
    computerPoints=1;
    blanks = 0;
}
//Potential diagonal1
if(c<=3 && r>=3){
    for(offset=1;offset<4;offset++){
        if(gameBoard.board[r-offset][c+offset]==COMPUTER){
          computerPoints++;
        }
        else
            if(gameBoard.board[r-offset][c+offset]==OPPONENT){
          computerPoints=0;
          blanks=0;
          break;
        }
        else{
          blanks++;
        }
    }
    countMoves=0;
    if(blanks>0){
        for(int col=1;col<4;col++){
            int column = c+col;
            int row = r-col;
            for(int rw=row;rw<=5;++rw){
                if(gameBoard.board[rw][column]==EMPTY){
                   countMoves++;
                }
                else
                    if(gameBoard.board[rw][column]==COMPUTER);
                else{
                   break;
                }
            }
        }
        if(countMoves!=0) {
          score = score+ calculateScore(computerPoints,
              countMoves);
        }
        computerPoints=1;
```

```
                    blanks = 0;
                }
            }
             //Potential diagonal2
            if(r>=3 && c>=3){
                for(offset=1;offset<4;++offset){
                    if(gameBoard.board[r-offset][c-offset]==COMPUTER){
                      computerPoints++;
                    }
                    else
                        if(gameBoard.board[r-offset][c-offset]==OPPONENT){
                      computerPoints=0;
                      blanks=0;
                      break;
                    }
                    else blanks++;
                }
                countMoves=0;
                if(blanks>0){
                    for(int col=1;col<4;col++){
                        int column = c-col;
                        int row = r-col;
                        for(int rw=row;rw<=5;rw++){
                            if(gameBoard.board[rw][column]==EMPTY){
                               countMoves++;
                            }
                            else
                                if(gameBoard.board[rw][column]==COMPUTER);
                            else{
                               break;
                            }
                        }
                    }
                    if(countMoves!=0){
                      score = score+ calculateScore(computerPoints,
                          countMoves);
                    }
                    computerPoints=1;
                    blanks = 0;
                }
            }
        }
    }
    return score;
}
```

```
 /*
  * Calculates some score to evaluate the desirability of a given
     decision for the computer
  * Based on how many points the computer already has, will weight
     decisions differently
  */

 int calculateScore(int computerPoints, int countMoves){

   //how many moves away from win
   int moveScore = 4 - countMoves;

     if(computerPoints==0){
      return 0;
     }
     else if(computerPoints==1){
      return 1*moveScore;
     }
     else if(computerPoints==2){
      return 10*moveScore;
     }
     else if(computerPoints==3){
      return 100*moveScore;
     }
     else{
      return 1000;
     }
 }

/*
 * playGames controls actual game play
 * Will loop through so multiple games can be played at once.
 * If playing multiple games toggle which player moves first
 *     Keeps track of computer wins, opponent wins, and draws. Calculates
    success rate
 */



 public char playGame(char goesFirst){
```

```
char winner;
   gameBoard.clearBoard();
   if(goesFirst==OPPONENT){   //opponent goes first else computer
       goes first
      opponentMove();
          gameBoard.displayBoard();
   }


      gameBoard.placePiece(3, COMPUTER);
      gameBoard.displayBoard();

      while(true){
         opponentMove();
         gameBoard.displayBoard();

         winner = findWinner(gameBoard);
         if(winner==COMPUTER){
            System.out.println("Computer Wins!");
            break;
         }
         else if(winner==OPPONENT){
            System.out.println("You Win!");
            break;
         }
         else if(winner=='D'){
            System.out.println("Draw!");
            break;
         }

         gameBoard.placePiece(informedDecision(), COMPUTER);
         gameBoard.displayBoard();
         winner = findWinner(gameBoard);
         if(winner==COMPUTER){
            System.out.println("AI Wins!");
            //computerWins++;
            break;
         }
         else if(winner==OPPONENT){
            System.out.println("You Win!");
            //opponentWins++;
            break;
         }
         else if(winner=='D'){
            System.out.println("Draw!");
```

```
            //draws++;
            break;
          }

       }
       return winner;
}
/*
 * Main Method
 * creates a game board and a learner and calls the play game function
 */

public static void main(String[] args) {
    Board gameBoard = new Board();
    ConnectFour learner = new ConnectFour(gameBoard);
  Scanner kbd = new Scanner(System.in);
    int computerWins=0;
    int opponentWins=0;
    int draws=0;
    char winner;
  long startTime = System.currentTimeMillis();

    System.out.println("*******CONNECT FOUR**********");
    for (int gameCount=1; gameCount<=NUM_OF_GAMES;gameCount++){
     System.out.println("GAME #"+ gameCount);
     System.out.println("============");
     if(gameCount%2==0){
        winner = learner.playGame(OPPONENT);
     }
     else{
        winner = learner.playGame(COMPUTER);
     }
     if(winner==COMPUTER){
        computerWins++;

     }
     else if(winner==OPPONENT){
        opponentWins++;

     }
     else if(winner=='D'){
        draws++;

     }
```

```
    }

    System.out.println("************STATISTICS***********");
    System.out.println("Games Played: "+ NUM_OF_GAMES);
    System.out.println("Learning Player Won: "+ computerWins);
    System.out.println("Opponent Won: "+ opponentWins);
    System.out.println("Endend in a draw: "+ draws);
    double rate=(double) computerWins/NUM_OF_GAMES;
    System.out.printf("\nSUCCESS RATE: %.2f%% " , (rate*100));

    //Max Depth
    System.out.println("\nMax Depth" + learner.maxDepth);

    // Get the Java runtime
    Runtime runtime = Runtime.getRuntime();

    // Run the garbage collector
    runtime.gc();
    // Calculate the used memory
    long memory = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("\nUsed memory is bytes: " + memory);

   System.out.println("Used memory is megabytes: " + memory / (1024L *
       1024L));
    System.out.println("Free memory: " + runtime.freeMemory() + " bytes.");

    //Calculate Run time
    long endTime  = System.currentTimeMillis();
    long totalTime = endTime - startTime;
    System.out.println("Total Time(in miliseconds): "+totalTime);
    System.out.println("Total Time(in seconds): "+ (double)totalTime/1000);

    System.out.println("Average time per game(in miliseconds): "+
        (double)totalTime/NUM_OF_GAMES);


}

}
```

B
Connect Four Source Code

```
/**
 * Colleen Rogers
 * Fall 2015
 * Machine Learning Applications: Intelligent Connect Four
 * About Connect4:
 *  Implementing Machine Learning Algorithm for Connect4:
 *
 *
 *  Uses the Minimax Algorithm in conjunction with a basic heuristic
    function I developed
 *  Can change depth of the algorithm; Will affect performance and speed.
 https://github.com/crogers719/Machine-Learning
 *
 */
import java.util.Scanner;



/**
 *BOARD CLASS:
 *Contains board (a 2-dimensional character array that holds the board in
    its current state)
 *Performs basic functions pertaining to alteration of the board itself
 */
class Board{

   //Standard Dimensions of a Connect four board (6x7)
   public static final int NUM_ROWS = 6;
   public static final int NUM_COLS = 7;

   //Player1 ('Red') is the one using the AI algorithm
   public static final char COMPUTER='R';
   //Player2 ('Yellow') is a naive opponent; Can either have (random)
      automated moves, or can be controlled by a user
   public static final char OPPONENT='Y';
   //Represents an empty cell
   public static final char EMPTY=' ';


   char[][] board = new char[NUM_ROWS][NUM_COLS];
```

```
/*
 * Clear board sets all positions in board equal to default value
    (EMPTY=' ')
 */

 public void clearBoard(){
   for (int r=0; r<NUM_ROWS; r++){
      for (int c=0; c<NUM_COLS; c++){
         board[r][c]=EMPTY;
      }
   }
 }

 /*
  * Will display the contents of the game board in its current state
  */
    public void displayBoard(){
        System.out.print("\n");
        for(int r=0;r<NUM_ROWS;r++){
            for(int c=0;c<NUM_COLS;c++){
                System.out.print(board[r][c]+" |");
            }
            System.out.print("\n-----------------------\n");
        }
        System.out.println();
    }

 /*
  * Given a column number, will place corresponding character (player)
  * in the highest (farthest down) empty row in that column
  */
 public boolean placePiece(int column, char player){
     if(!validMove(column)) {
      System.out.println("Illegal move!");
      return false;
     }
     for(int r=NUM_ROWS-1;r>=0;r--){
         if(board[r][column] == EMPTY) {
             board[r][column] = player;
             return true;
         }
     }
     return false;
 }
```

```
    /*
     * Valid Move determines whether a move (choice of column) is valid
     *   Checks that the column number is between 1-7 and that the chosen
         column is not full
     */
    public boolean validMove(int column){
      if(column>=0 && column<NUM_COLS && board[0][column]==' '){
         return true;
        }
        else{
            return false;
        }
    }



    /*
     * Given the column number will remove the last piece inserted into
         the column
     * Useful for undoing moves made as part of the Minimax algorithm
     */

    public void undo(int column){
        for(int r=0;r<NUM_ROWS;++r){
            if(board[r][column] != EMPTY) {
                board[r][column] = EMPTY;
                break;
            }
        }
    }

}
/**
 * CONNECTFOUR CLASS
 * Contains:
 *         Constants for: board dimensions, characters, number of games,
    and autoplay settings
 *         An object of the board class to hold the game board,the next
    move and, the maximum depth to be used in Minimax
 * Performs:
 *         Machine learning algorithm that enables the intelligent player
    (COMPUTER) to make advantageous
 *         decisions using the Minimax Algorithm
```

```
 */

public class ConnectFour {
   //Standard Dimensions of a Connect four board (6x7)
   public static final int NUM_ROWS = 6;
   public static final int NUM_COLS = 7;

   //Player1 ('Red') is the one using the AI algorithm
   public static final char COMPUTER='R';
   //Player2 ('Yellow') is a naive opponent; Can either have (random)
      automated moves, or can be controlled by a user
   public static final char OPPONENT='Y';
   //Represents an empty cell
   public static final char EMPTY=' ';

   //Number of consecutive games to be played
   public static final int NUM_OF_GAMES=250;
   //If true, will randomize moves; Otherwise a user can play against the
      machine
   public static final boolean AUTOPLAY=true;


    private Board gameBoard;
    private Scanner kbd;
    private int nextMove=-1;

    /*The maximum depth used by the min-max algorithm--need cut off point
    * Tested for values between 1-8(depth =1 --> fast with about 97%
       success rate
    * depth =8 -->slower but with a near 100% success rate
    */
    private int maxDepth =7;

    /*
     * ConnectFour constructor
     *        Takes a Board object, b, as a parameter and assigns
        gameBoard=b
     *        Sets up a Scanner object (kbd) to be used in class functions
     */

    public ConnectFour(Board b){
        gameBoard = b;
        kbd = new Scanner(System.in);
    }
```

```
/*
 * Allows the opponent to make a move. If AUTOPLAY is on will simply
    generate a number between 1-7(NUM_COLS)
 * Otherwise it will take input from the user for a column number (1-7)
 * Then calls the place piece method (in Board class) to place the
    piece at the position corresponding to the
 * column number obtained.
 *
 */
public void opponentMove(){
  int move=-1;
  if (AUTOPLAY){
     do{
        move= (int)(Math.random()*NUM_COLS) + 1;
     }while (!gameBoard.validMove(move-1));
  }
  else{
     System.out.println("Enter a Column (1-7): ");
     move = kbd.nextInt();
     while(!gameBoard.validMove(move-1)){
        System.out.println("Invalid move.\n\nEnter a Column (1-7): ");
        move = kbd.nextInt();
     }
  }
    //call to place piece (in Board class); this assumes that player 2
       is the opponent
    gameBoard.placePiece(move-1, OPPONENT);
}


/*
 * Finds the results of the game by checking the gameBoard for a winner
 *     Four ways to win:
 *          Horizontally
 *          Vertically
 *          Diagonal#1 (lower left->upper right)
 *          Diagonal#2 (lower right->upper left)
 *     If finds a winner, returns the character associated with
    winning player.
 */
public char findWinner(Board gameBoard){
    int countComputer = 0;
    int countOpponent = 0;
```

```
for(int r=NUM_ROWS-1;r>=0;r--){
    for(int c=0;c<NUM_COLS;c++){
       if(gameBoard.board[r][c]==EMPTY){
          continue;
       }

        //Check For Horizontal Win
        if(c<=3){
            for(int incr=0;incr<4;incr++){
                    if(gameBoard.board[r][c+incr]==COMPUTER) {
                      countComputer++;
                    }
                    else if(gameBoard.board[r][c+incr]==OPPONENT) {
                      countOpponent++;
                    }
                    else{
                      break;
                    }
            }
            if(countComputer==4){
             return COMPUTER;
            }
            else if (countOpponent==4){
             return OPPONENT;
            }
            //No winner-reset counters
            countComputer = 0;
            countOpponent = 0;
        }

        //Check for vertical win
        if(r>=3){
            for(int incr=0;incr<4;incr++){
                    if(gameBoard.board[r-incr][c]==COMPUTER) {
                      countComputer++;
                    }
                    else if(gameBoard.board[r-incr][c]==OPPONENT) {
                      countOpponent++;
                    }
                    else{
                      break;
                    }
            }
            if(countComputer==4){
             return COMPUTER;
```

```
    }
    else if (countOpponent==4){
     return OPPONENT;
    }
    countComputer = 0;
    countOpponent = 0;
}

//Check for diagonal (up-right) win
if(c<=3 && r>= 3){
    for(int incr=0;incr<4;incr++){
        if(gameBoard.board[r-incr][c+incr]==COMPUTER){
           countComputer++;
        }
        else if(gameBoard.board[r-incr][c+incr]==OPPONENT) {
           countOpponent++;
        }
        else{
           break;
        }
    }
    if(countComputer==4){
     return COMPUTER;
    }
    else if (countOpponent==4){
     return OPPONENT;
    }
    countComputer = 0;
    countOpponent = 0;
}

//Check for diagonal (up-left) win
if(c>=3 && r>=3){
    for(int incr=0;incr<4;incr++){
        if(gameBoard.board[r-incr][c-incr]==COMPUTER){
           countComputer++;
        }
        else if(gameBoard.board[r-incr][c-incr]==OPPONENT){
           countOpponent++;
        }
        else{
           break;
        }
    }
    if(countComputer==4){
```

```
                    return COMPUTER;
                }
                else if (countOpponent==4){
                    return OPPONENT;
                }
                countComputer = 0;
                countOpponent = 0;
            }
        }
    }

    for(int c=0;c<NUM_COLS;c++){
        //There is still an empty location-Game has not ended yet
        if(gameBoard.board[0][c]==EMPTY){
            return '-';
        }
    }
    //No winner and no empty locations mean that the game is a DRAW
    return 'D';
}

/*
 * Calls Minimax, a recursive function, which will eventually change
     the value of next move
 * Then returns nextMove
 */

public int informedDecision(){
    nextMove = -1;
    int start =0;

    Minimax(start, COMPUTER);
    return nextMove;
}

/*
 * MINIMAX ALGORITHM
 * Used when expanding the full game tree is not possible
 * Aims to minimize the possible loss for worst case scenario.
 * Minimizes the loss against a perfectly playing opponent.
 * Uses a a basic heuristic function to evaluate the desirability of given
    board states
 */
public int Minimax(int depth, char turn){
    //check for winner will return COMPUTER or OPPONENT if win
```

```
char findWinner = findWinner(gameBoard);

  if(findWinner==COMPUTER){ //computer won
   return Integer.MAX_VALUE;
  }
  else if(findWinner==OPPONENT){ //oponent won
   return Integer.MIN_VALUE;
  }
  else if(findWinner==EMPTY){
   return 0;
  }

  if(depth==maxDepth){
   return determineFitness(gameBoard);
  }

  int maxScore=Integer.MIN_VALUE;
  int minScore = Integer.MAX_VALUE;

  for(int c=0;c<NUM_COLS;c++){
      if(!gameBoard.validMove(c)){
         continue;
      }

      if(turn==COMPUTER){
         gameBoard.placePiece(c, COMPUTER);
          int currentScore = Minimax(depth+1, OPPONENT);
          maxScore = Math.max(currentScore, maxScore);
          if(depth==0){
            System.out.println("Score for location "+c+" =
                "+currentScore);
             if(maxScore==currentScore) {
              nextMove = c;
             }
          }
      }

      else if(turn==OPPONENT){
             gameBoard.placePiece(c, OPPONENT);
             int currentScore = Minimax(depth+1, COMPUTER);
             minScore = Math.min(currentScore, minScore);
      }
      gameBoard.undo(c);
  }
  if (turn==COMPUTER){
```

```
     return maxScore;
    }

    else{          //turn==OPPONENT

     return minScore;
    }
}
/*
 * A basic heuristic function to evaluate the desirability of board
    states
 * Totals up the amount of consecutive pieces and calls calculateScore
    to
 * weigh the score based on how close player is to victory
 *
 */
 public int determineFitness(Board gameBoard){

    int computerPoints=1;
    int score=0;
    int blanks = 0;
    int offset=0;
    int countMoves=0;
    for(int r=NUM_ROWS-1;r>=0;r--){
        for(int c=0;c<NUM_COLS;c++){

            if(gameBoard.board[r][c]==EMPTY ||
                gameBoard.board[r][c]==OPPONENT){
             continue;
            }
            //potential horizontal
            if(c<=3){
                for(offset=1;offset<4;offset++){
                    if(gameBoard.board[r][c+offset]==COMPUTER){
                      computerPoints++;
                    }
                    else if(gameBoard.board[r][c+offset]==OPPONENT){
                      computerPoints=0;
                      blanks = 0;
                      break;
                    }
                    else{
                      blanks++;
                    }
```

```
    }

    countMoves = 0;
    if(blanks>0) {
        for(int col=1;col<4;col++){
            int column = c+col;
            for(int rw=r; rw<= 5;rw++){
             if(gameBoard.board[rw][column]==EMPTY){
              countMoves++;
             }
                else{
                   break;
                }
            }
        }
    }

    if(countMoves!=0){
       score = score+ calculateScore(computerPoints,
           countMoves);
    }
    computerPoints=1;
    blanks = 0;
}

//potential vertical
if(r>=3){
    for(offset=1;offset<4;++offset){
        if(gameBoard.board[r-offset][c]==COMPUTER){
          computerPoints++;
        }
        else if(gameBoard.board[r-offset][c]==OPPONENT){
          computerPoints=0;
          break;
        }
    }
    countMoves = 0;

    if(computerPoints>0){
        int column = c;
        for(int rw=r-offset+1; rw<=r-1;rw++){
         if(gameBoard.board[rw][column]==EMPTY){
           countMoves++;
         }
            else{
```

```
            break;
           }
        }
    }
    if(countMoves!=0) {
       score = score+ calculateScore(computerPoints,
          countMoves);
    }
    computerPoints=1;
    blanks = 0;
}

if(c>=3){
    for(offset=1;offset<4;offset++){
        if(gameBoard.board[r][c-offset]==COMPUTER){
          computerPoints++;
        }
        else if(gameBoard.board[r][c-offset]==OPPONENT){
          computerPoints=0;
          blanks=0;
          break;
        }
        else{
          blanks++;
        }
    }
    countMoves=0;
    if(blanks>0) {
        for(int col=1;col<4;col++){
            int column = c- col;
            for(int rw=r; rw<= 5;rw++){
             if(gameBoard.board[rw][column]==EMPTY){
              countMoves++;
             }
                else{
                   break;
                }
            }
        }
    }

    if(countMoves!=0){
       score = score+ calculateScore(computerPoints,
          countMoves);
    }
```

```
    computerPoints=1;
    blanks = 0;
}
//Potential diagonal1
if(c<=3 && r>=3){
    for(offset=1;offset<4;offset++){
        if(gameBoard.board[r-offset][c+offset]==COMPUTER){
          computerPoints++;
        }
        else
            if(gameBoard.board[r-offset][c+offset]==OPPONENT){
          computerPoints=0;
          blanks=0;
          break;
        }
        else{
          blanks++;
        }
    }
    countMoves=0;
    if(blanks>0){
        for(int col=1;col<4;col++){
            int column = c+col;
            int row = r-col;
            for(int rw=row;rw<=5;++rw){
                if(gameBoard.board[rw][column]==EMPTY){
                   countMoves++;
                }
                else
                    if(gameBoard.board[rw][column]==COMPUTER);
                else{
                   break;
                }
            }
        }
        if(countMoves!=0) {
          score = score+ calculateScore(computerPoints,
              countMoves);
        }
        computerPoints=1;
        blanks = 0;
    }
}
 //Potential diagonal2
if(r>=3 && c>=3){
```

```
                for(offset=1;offset<4;++offset){
                    if(gameBoard.board[r-offset][c-offset]==COMPUTER){
                      computerPoints++;
                    }
                    else
                        if(gameBoard.board[r-offset][c-offset]==OPPONENT){
                      computerPoints=0;
                      blanks=0;
                      break;
                    }
                    else blanks++;
                }
                countMoves=0;
                if(blanks>0){
                    for(int col=1;col<4;col++){
                        int column = c-col;
                        int row = r-col;
                        for(int rw=row;rw<=5;rw++){
                            if(gameBoard.board[rw][column]==EMPTY){
                               countMoves++;
                            }
                            else
                                if(gameBoard.board[rw][column]==COMPUTER);
                            else{
                               break;
                            }
                        }
                    }
                    if(countMoves!=0){
                      score = score+ calculateScore(computerPoints,
                          countMoves);
                    }
                    computerPoints=1;
                    blanks = 0;
                }
            }
        }
    }
    return score;
}



/*
```

```
  * Calculates some score to evaluate the desirability of a given
     decision for the computer
  * Based on how many points the computer already has, will weight
     decisions differently
  */

 int calculateScore(int computerPoints, int countMoves){

   //how many moves away from win
   int moveScore = 4 - countMoves;

     if(computerPoints==0){
      return 0;
     }
     else if(computerPoints==1){
      return 1*moveScore;
     }
     else if(computerPoints==2){
      return 10*moveScore;
     }
     else if(computerPoints==3){
      return 100*moveScore;
     }
     else{
      return 1000;
     }
 }

/*
 * playGames controls actual game play
 * Will loop through so multiple games can be played at once.
 * If playing multiple games toggle which player moves first
 *    Keeps track of computer wins, opponent wins, and draws. Calculates
    success rate
 */



 public char playGame(char goesFirst){

   char winner;
      gameBoard.clearBoard();
      if(goesFirst==OPPONENT){   //opponent goes first else computer
         goes first
        opponentMove();
```

```
        gameBoard.displayBoard();
}


   gameBoard.placePiece(3, COMPUTER);
   gameBoard.displayBoard();

   while(true){
      opponentMove();
      gameBoard.displayBoard();

      winner = findWinner(gameBoard);
      if(winner==COMPUTER){
         System.out.println("Computer Wins!");
         break;
      }
      else if(winner==OPPONENT){
         System.out.println("You Win!");
         break;
      }
      else if(winner=='D'){
         System.out.println("Draw!");
         break;
      }

      gameBoard.placePiece(informedDecision(), COMPUTER);
      gameBoard.displayBoard();
      winner = findWinner(gameBoard);
      if(winner==COMPUTER){
         System.out.println("AI Wins!");
         //computerWins++;
         break;
      }
      else if(winner==OPPONENT){
         System.out.println("You Win!");
         //opponentWins++;
         break;
      }
      else if(winner=='D'){
         System.out.println("Draw!");
         //draws++;
         break;
      }

   }
```

```
        return winner;
}
/*
 * Main Method
 * creates a game board and a learner and calls the play game function
 */

public static void main(String[] args) {
    Board gameBoard = new Board();
    ConnectFour learner = new ConnectFour(gameBoard);
  Scanner kbd = new Scanner(System.in);
    int computerWins=0;
    int opponentWins=0;
    int draws=0;
    char winner;
  long startTime = System.currentTimeMillis();

    System.out.println("*******CONNECT FOUR**********");
    for (int gameCount=1; gameCount<=NUM_OF_GAMES;gameCount++){
     System.out.println("GAME #"+ gameCount);
     System.out.println("============");
     if(gameCount%2==0){
        winner = learner.playGame(OPPONENT);
     }
     else{
        winner = learner.playGame(COMPUTER);
     }
     if(winner==COMPUTER){
        computerWins++;

     }
     else if(winner==OPPONENT){
        opponentWins++;

     }
     else if(winner=='D'){
        draws++;

     }

}

System.out.println("************STATISTICS***********");
System.out.println("Games Played: "+ NUM_OF_GAMES);
System.out.println("Learning Player Won: "+ computerWins);
```

```
    System.out.println("Opponent Won: "+ opponentWins);
    System.out.println("Endend in a draw: "+ draws);
    double rate=(double) computerWins/NUM_OF_GAMES;
    System.out.printf("\nSUCCESS RATE: %.2f%% " , (rate*100));

    //Max Depth
    System.out.println("\nMax Depth" + learner.maxDepth);
    // Get the Java runtime
    Runtime runtime = Runtime.getRuntime();
    // Run the garbage collector
    runtime.gc();
    // Calculate the used memory
    long memory = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("\nUsed memory is bytes: " + memory);

   System.out.println("Used memory is megabytes: " + memory / (1024L *
       1024L));
    System.out.println("Free memory: " + runtime.freeMemory() + " bytes.");
    //Calculate Run time
    long endTime  = System.currentTimeMillis();
    long totalTime = endTime - startTime;
    System.out.println("Total Time(in miliseconds): "+totalTime);
    System.out.println("Total Time(in seconds): "+ (double)totalTime/1000);
    System.out.println("Average time per game(in miliseconds): "+
        (double)totalTime/NUM_OF_GAMES);


}

}
```

References

[1] Valeri Alexiev. A knowledge-based approach of connect-four. Master's thesis, VU University Amsterdam, Amsterdam, The Netherlands, 1988.

[2] Victor Allis. A knowledge-based approach of connect-four. Master's thesis, VU University Amsterdam, Amsterdam, The Netherlands, October 1988.

[3] Taiwo Oladipupo Ayodele. *New Advances in Machine Learning*, chapter 3. InTech, February 2010. http://www.intechopen.com/books/new-advances-in-machine-learning/types-of-machine-learning-algorithms.

[4] Andrew Barto and Richard S. Sutton. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[5] Peng Ding and Tao Mao. Reinforcement Learning in Tic-Tac-Toe Game and Its Similar Variations. Master's thesis, Thayer School of Engineering at Dartmouth College.

[6] Colin Foster. *Resources for Teaching Mathematics 14-16*, chapter 14, pages 54–58. Bloomsbury Academic, October 2010. Chapter on "Connect Four".

[7] Amir Kamil. Topics: Minimax. http://web.eecs.umich.edu/ akamil/teaching/sp03/minimax.pdf.

[8] Peter Kissman. *Symbollic search in planning and general game playing*. PhD thesis, University of Bremen, Bremen, Germany, May 2012. Doctoral Dissertation.

[9] Michael L. Litman Leslie Pack Kaelbling and Andrew W. Moore. Reinforcement learning: A survey. *Journal of AI Research*, 4, 1996.

[10] Kevin Murphy. A Brief Introduction to Reinforcement Learning. 1998.

[11] Colleen Rogers. Machine learning project repository. https://github.com/crogers719/Machine-Learning. Git Hub Repository Containing Project Source Code Files.

[12] SAS. "Machine Learning: What it is and why it matters". http://www.sas.com/en$_u$s/*insights/analytics/machine learning.html*. *PublishedbySAS*.

[13] Glenn Strong. The minimax algorithm. https://www.cs.tcd.ie/Glenn.Strong/3d5/minimax-notes.pdf.

[14] John Tromp. "John's connect four playground". https://tromp.github.io/c4/c4.html.