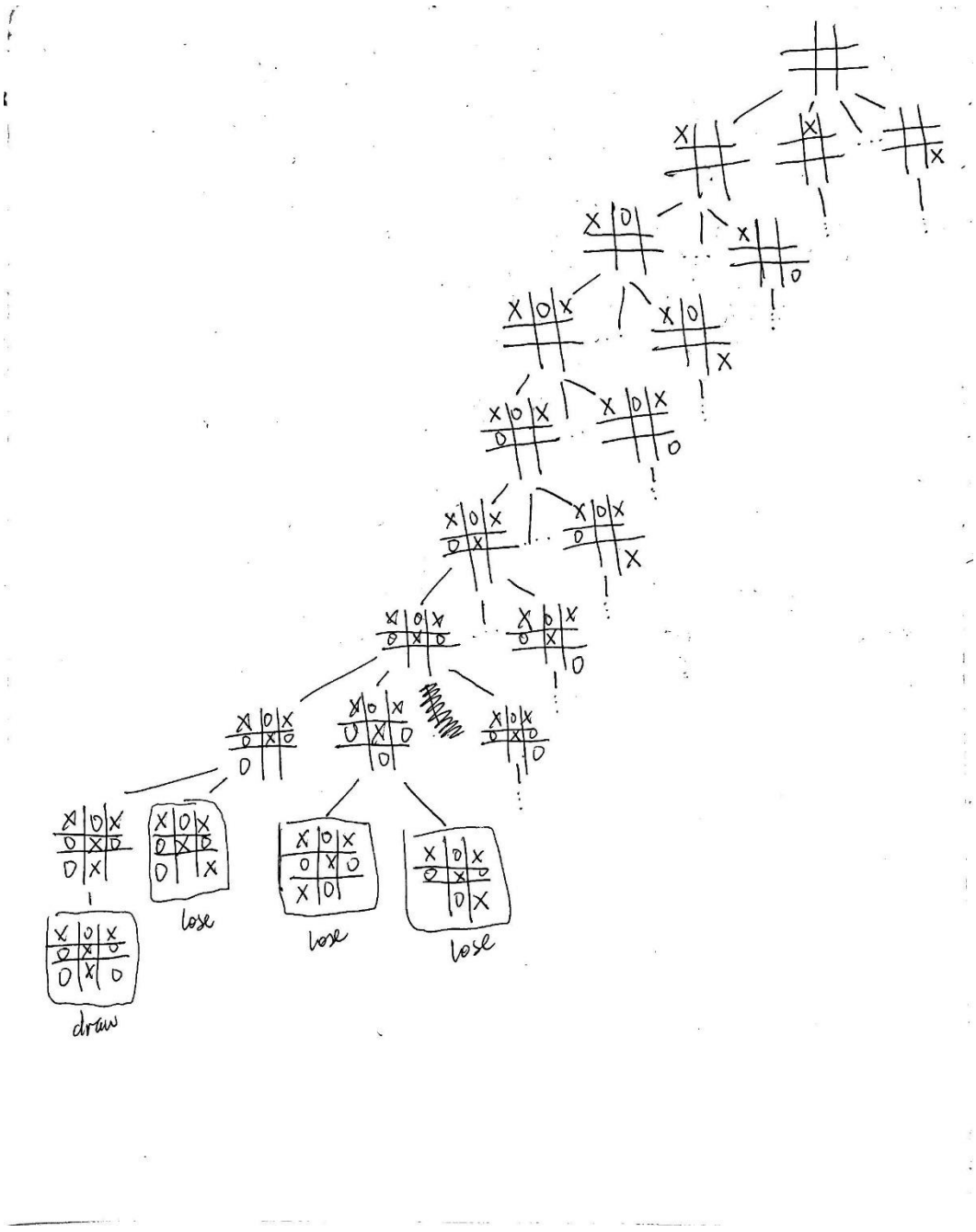
Jared Yu

Homework 5

Algorithms for Data Science

Part 1)

Part a) The following is an example of the first four leaf nodes according to a depth-first search approach to analyzing the simulated possible results for a game of tic-tac-toe between a user (X) and the AI (O). The result is a tie and three losses.



Part b)

The way that the following pseudo code works is that it treats the current game state as if on the first turn, the user has already placed an X in a place on the tic-tac-toe board. For instance, on turn one, the user has placed an X on the top left corner. Then on turn two, for the AI, it must determine a score for each of the remaining eight empty positions. The AI will choose the position with the highest value. So, on every even turn, the AI must determine a score for each of the remaining empty positions. The recursive method used is MiniMax, where the goal is to minimize the maximize loss possible for the AI. In the levels of Maximizing and Minimizing, the AI is the Maximizing step and the user is the Minimizing step.

Although the methodology used is MiniMax, the helper functions can be included anywhere in the .py file, the pseudocode has been made such that the MiniMax would be placed in checkWinPos. The final checkWinPos is at the end of the document.

Given that b is the maximum branching factor and m is the maximum depth of space, the algorithm is $O(b^m)$ since it is based on the MiniMax algorithm. Normally, b is 9 since from the empty root node there are 9 possible positions. However, since the first board state given to the AI is after the user has already placed an X, the actual value of b is 8. Similarly, given that the empty board root node is at depth 0, then the maximum depth would be 9. The practical root node would instead be the root node where the user has already placed an X, so a sensible value for m instead would be 8. The running time of the algorithm then is $O(b^m) = O(8^8) = O(16{,}777{,}216)$.

References:

https://github.com/Cledersonbc/tic-tac-toe-minimax/blob/master/README.md

https://www.ntu.edu.sg/home/ehchua/programming/java/JavaGame_TicTacToe_AI.html#zz-1.5

https://www.cs.jhu.edu/~jorgev/cs106/ttt.pdf

http://www.r-5.org/files/books/computers/algo-list/common/Heineman_Pollice_Selkow-Algorithms_in_a_Nutshell-EN.pdf

Helper functions for the MiniMax algorithm:

// The purpose of LEAF-NODE-SCORE() is to determine the score at the leaf node of a tree during the exiting stage of the recursive MiniMax algorithm. It utilizes the checkwin2() function included in the assignment. It checks if the winner is the AI, the user, or if it is a tie. If the AI wins, a score of 1 is returned, if the user wins, -1 is returned, if it is a tie, 0 is returned. At this point, the simulated game must have already ended since it is in the leaf node.

// Arguments:

board – The ‘current’ board state to be scored using MiniMax

```
LEAF-NODE-SCORE(place)
// Check if AI won
If (checkWin2(place) == 2):
  Return 1
// Check if user won
Else if (checkWins2(place) == 1):
  Return -1
// Case if tie
Else:
  Return 0
```

// The END-OF-GAME() function checks during the recursive process if a simulated game has ended either through an AI win, user win, or a tie. The win for the AI or user is determined if either has a 3-in-a-row. The tie is determined if all there are no more empty positions and neither opponents have won. It utilizes the checkWin2() function included in the assignment to check if either of these states have occurred.

```
// Arguments:
board – The ‘current’ board state to be scored using MiniMax
END-OF-GAME(place):
// Return 1 if game over, -1 otherwise
If (checkWin2(place) == 2):
  Return 1
Else if (checkWins2(place) == 1):
  Return 1
Else if (checkWins2(place) == 0):
  Return 1
Else:
  Return -1
```

MiniMax function:

// The MINI-MAX() function recursively searches through the remaining empty positions in the given board state to determine a score and position for all of the possible remaining outcomes. It uses the MiniMax algorithm to simulate moves by either the AI or the user to end up at different leaf nodes, from which a final score of +1, -1, or 0 can be determined in the case of an AI win, user win, or tie, respectively. At the last stage of recursion, the simulated are undone on the board state to return to the original given board state. In each of the remaining empty positions, a score has been derived, and the highest score is given as an output for the AI to use.

// Arguments:

board – The 'current' board state to be scored using MiniMax

number_of_remaining_positions – The number of empty positions left on the board.

human_or_ai – A value of either +1 or -1 to show that the next move to be placed is either the human or AI via simulation. The AI (+1) is to be maximized and the user (-1) is to be minimized.

```
MINI-MAX(board, number_remaining_positions, human_or_ai):
// Determine if MiniMax simulation is for user or AI
If (human_or_ai == 1):
  Default_best = [-1, -inf]
  Next_move = 1
Else:
  Default_best = [-1, inf]
  Next_move = 2

// Exit recursion once leaf node is reached
If (number_remaining_positions == 0 or END-OF-GAME(board) == 1):
  Simulated_score = LEAF-NODE-SCORE(board)
  Return [-1, Simulated_score]

// Create a list (Python style list comprehension) of the indices of the remaining empty positions
Empty_indices = [indices for indices, value in enumerate(board) if value == 0]
// Determine a score for each of the remaining empty cells
For Empty_position in Empty_indices:
  // Simulate move in empty board position
```

```
  board[Empty_position] = Next_move
  // Determine score of simulated move
  Simulated_score = MINI-MAX(board, number_remaining_positions, (-1) * human_or_ai)
  // Undo simulated move
  board[Empty_position] = 0
  // Update the score to show the position that led to the resulting score
  Simulated_score[0] = Empty_position
  // Maximize at the level of AI
  If human_or_ai == 1:
    If Simulated_score[1] > Default_best[1]:
      Default_best = Simulated_score
  // Minimize at the level of user
  Else:
  If Simulated_score[1] < Default_best[1]:
      Default_best = Simulated_score
// Return a list of the resulting position and score combination
Return Default_best
```

// The checkWinPos() function receives a board state after the user has placed an X. The function determines the score according to the MiniMax algorithm for each of the remaining positions. It returns the position that returns the highest possible score.

```
Def checkWinPos(place):
// Create a list (Python style list comprehension) of the indices of the remaining empty positions
Empty_indices = [indices for indices, value in enumerate(board) if value == 0]
// Number of remaining positions
Number_remaining_positions = len(Empty_indices)
// Exit recursion
If (Number_remaining_positions == 0 or END-OF-GAME(place)):
  Return
```

```
// Determine best move according to MINI-MAX() algorithm

Mini_max_result = MINI-MAX(board=place, number_remaining_positions=
Number_remaining_positions, human_or_ai=1)

// Return best position to select

Return Mini_max_result[1]
```