**Project 3**

**Artificial Intelligence**

**CSCE 5210 – Fall 2021**

**Distributed: Wednesday 27 October**

**Due: Wednesday, November 24**

The game : Tic-Tac-Toe

Board size : 3 x 3 blocks

The initial stage : Blank board

The player :

1. Max player : AI player which is represented by ‘X’. The actions are performed by the program.
2. Min player : human player which is represented by ‘O’. The actions are performed via the keyboard input, from 1-9.
   1. There are two types of input in this game.
      1. Invert numpad - like an ordinary numpad pattern in desktop keyboard.

7 | 8 | 9

4 | 5 | 6

1 | 2 | 3

* + 1. Sequence numpad - this type of input is suitable for the laptop keyboard.

1 | 2 | 3

4 | 5 | 6

7 | 8 | 9

Three condition in the game :

1. ‘X’ won - AI won the game.
2. ‘O’ won - the player won the game.
3. Tie - no one wins the game, and there is no space(‘-’) left to play.

Note : The game will be conducted for ten rounds.

**Check win condition :**

The winner must have got all three :

1. Horizontal : (0, 1, 2), (3, 4, 5), (6, 7, 8)
2. Vertical : (0, 3, 6),(1, 4, 7), (2, 5, 8)
3. Two diagonal : (0, 4, 8), (2, 4, 6)

**How is this game conducted?**

There will be ten rounds in the Tic-Tac-Toe. The first player that gets a chance to play has the advantage to win more than the second player. According to our experiments, AI who plays first almost always wins.

In this project, we let player A, computer, be the first player to start Tic-Tac-Toe, and we figure that we can’t beat this AI. To be convenient, we add ‘check\_keyboard’ to check the numeric pattern as an input, such as numpad pattern(reversed) or sequence number pattern.

**Apply Minimax function in the game.**

For the board size 3 x 3, there are (3 \* 3)! = 9! or 362,880 possible situations for the first move, 8! for the second move, 7! third move, and so on. However, we don’t need to go through all 362,880 states.

In this algorithm, we will use minimax which will be conducted by the time complexity, O(b^m), and the space complexity, O(bm). b is the number of moves at each point, and m is the maximum depth of the tree.

The MIN, human player’s, goal is to try to get the score to minimize in every move. On the other hand, MAX player, AI, will try to get the maximum score in every turn as well.

Alpha-beta pruning will be the method to tell the minimax algorithm without searching on every possible node in this game tree.

* Alpha value will be a positive value, like +1.
* Beta value will be a negative value, like -1.
* Tie will be zero.

To start with, the initial value of Alpha started with the negative infinity, and the initial value of Beta started with the positive infinity. Then we will update the alpha or beta at every human turn’s end by comparing between best and the current value, and prune the branch of the tree whenever the alpha value is greater than or equal to the beta value(alpha >= beta). In this method, we can eliminate a ton of branches and reduce processing time for the MAX player, AI.

**A case in which MAX wins (include the alpha and beta values for the game nodes)**

Here is one example of alpha beta value in the game.

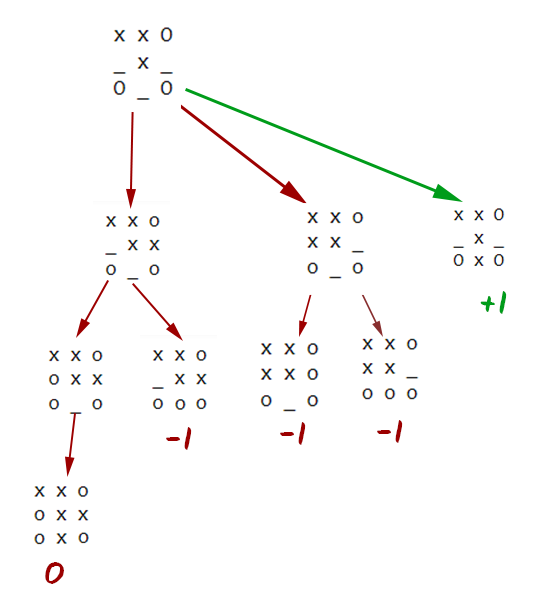
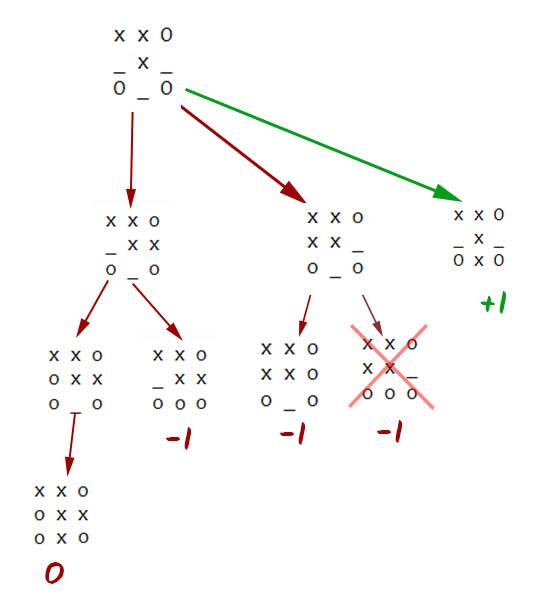


Figure one : Tic tac toe with two moves left for ‘X’(robot), and one move left for ‘O’(human).

According to figure one, we can see three possible scenarios for ‘O’ win, and one possible scenario for ‘X’ win as well as ‘tie’. However we can prune one of the branches in the center to reduce the processing time.



Hence, we also use ‘depth’ as the score by initially giving zero value to it. Depth is useful in situations when there is a tie in terms of score among different situations, the algorithm will choose the one with the least depth(Right one), which is a way of pruning.

**The result :**  
 After applying the minimax algorithm to the AI and letting the ‘X’ to be the first player, we learn that there is no way to win during these ten rounds.