CS 440 Programming Assignment 3

Chis-Emio Raymond
Ivan Mata and Zhan Hao Xu (Tiger)

11/25/19

Problem Definition: Our problem is that we have to construct an AI that is able to win a 4x4x4 tic-tac-toe game. This result is useful because we can use this knowledge and apply it to any other game. An assumption we make is that a runTicTacToe and a positionTicTacToe file are included so that our code may work. Some of the anticipated difficulties we thought we would have were figuring out how to calculate the heuristic value, properly incorporating the minimax function, and determining which location would be best to place our mark. We defined our nodes as being states of the board. The children of that node are any possible states of the board stemming from that original one. So for example, say we are playing on a 3x3 board. If our original state of the board is one where there is an X on the top left corner, then the children nodes would be any board where there's an X on the top left. Our terminal node is where our game ends and the state of the board at that instance.

Method and Implementation:

myAIAlgorithm() - We loop through every position in the 4x4x4 tic-tac-toe board and try to find the position with the largest value using our minimax function. We then change our x,y,z coordinates to that respective spot and place our mark there.

miniMaxABP() - We first determine the heuristic value of the current state of the board and then proceed to check if depth is equal to 0 because if so we return the heuristic value we calculated. If depth does not equal to 0 and maximizingPlayer is equal to true, then we used a for loop through all 64 spots to see if they are occupied and if they're not then we recursively call minimax again on it but with depth - 1. The same process goes for if maximizingPlayer is equal to false.

evalHeuristicValue() - We determine the heuristic value by instantiating a list of lists containing all the possible winning combinations. We then used a for loop through each pair of winning combinations and set 4 points to each of the 4 spots required to win for that specific winning combination and name them p0, p1, p2, and p3. We then find the state of each of those 4 positions and add it to a list called states. We then check to see if that spot/state is occupied by us or by the opponent. If we occupy that spot then we increment maxScore by 1, but if the opponent occupies that spot then we increment minscore by 1. In the end we then add a specific value depending on how we want to weigh the spots.

Experiments: For our experiments we ran our code against the random AI algorithm 100 times and calculated the ratio of wins to losses. We then ran our algorithm against itself 25 times and again calculated the ratio of wins to losses.

Results:

Us vs. Random AI

```
(3,3,0)state: 1
(3,3,1)state: 1
(3,3,2)state: 1
(3,3,3)state: 1
Player1 Wins
level(z) 0
[____]
[____]
[_O__]
[___X]

level(z) 1
[____]
[____]
[____]
[___X]

level(z) 2
[____]
[____]
[____]
[___X]

level(z) 3
[__O_]
[____]
[___O]
[___X]
```

```
(3,2,0)state: 1
(3,2,1)state: 1
(3,2,2)state: 1
(3,2,3)state: 1
Player1 Wins
level(z) 0
[_O_O]
[____]
[____]
[O_XO]

level(z) 1
[_O__]
[__O_]
[____]
[__XX]

level(z) 2
[____]
[____]
[____]
[__XX]

level(z) 3
[____]
[____]
[____]
[_OXX]
```

```
(3,2,0)state: 1
(3,2,1)state: 1
(3,2,2)state: 1
(3,2,3)state: 1
Player1 Wins
level(z) 0
[____]
[_OO_]
[OO__]
[__XX]

level(z) 1
[____]
[_O__]
[____]
[__XO]

level(z) 2
[____]
[____]
[____]
[__XX]

level(z) 3
[____]
[____]
[____]
[__XX]

Player 1 score 100
Player 2 score 0
Draw score 0
```

Us vs. Ourselves

```
(3,0,3)state: 1
(3,1,3)state: 1
(3,2,3)state: 1
(3,3,3)state: 1
Player1 Wins
level(z) 0
[____]
[____]
[____]
[_OOO]

level(z) 1
[____]
[____]
[____]
[_XXX]

level(z) 2
[____]
[____]
[____]
[_OOO]

level(z) 3
[____]
[____]
[____]
[XXXX]
```

```
(3,0,3)state: 1
(3,1,3)state: 1
(3,2,3)state: 1
(3,3,3)state: 1
Player1 Wins
level(z) 0
[____]
[____]
[____]
[_OOO]

level(z) 1
[____]
[____]
[____]
[_XXX]

level(z) 2
[____]
[____]
[____]
[_OOO]

level(z) 3
[____]
[____]
[____]
[XXXX]
```

```
(3,0,3)state: 2
(3,1,3)state: 2
(3,2,3)state: 2
(3,3,3)state: 2
Player2 Wins
level(z) 0
[____]
[____]
[____]
[_XXX]

level(z) 1
[____]
[____]
[____]
[_OOO]

level(z) 2
[____]
[____]
[____]
[_XXX]

level(z) 3
[____]
[____]
[____]
[OOOO]

Player 1 score 51
Player 2 score 49
Draw score 0
```

Average time used for AI “thinking”:

```
3449
 avg 3449
4677
 avg 2338
95695
 avg 31898
96422
 avg 24105
196167
 avg 39233
353378
 avg 58896
(3,3,0)state: 1
(3,3,1)state: 1
(3,3,2)state: 1
(3,3,3)state: 1
Player1 Wins
level(z) 0
[____]
[____]
[____]
[___X]

level(z) 1
[__O_]
[___O]
[____]
[___X]

level(z) 2
[____]
[____]
[O___]
[___X]

level(z) 3
[____]
[___O]
[____]
[___X]

Player 1 score 1
Player 2 score 0
Draw score 0
```

Discussion: Our AI applies more weight to spots that are in line with it vertically. If it's blocked then it will continue onto the next vertical spot in an attempt to continue on from there. The reason our program takes the time that it does is because we are running it with a depth of 3. If it were lower, then our program would run faster than it already is.

Conclusions: We concluded that our algorithm for the most part works but if it were to play against a more advanced opponent then we'd lose more than we'd win. Our main message is that our algorithm could use some work and could be improved upon but as of right now it still correctly calculates the heuristic values of board positions. One way we could improve upon it is by speeding up the time it takes to calculate the minimax at bigger depths.

Credits and Bibliography:

https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/ - Date of Access: 11/22/19
https://en.wikipedia.org/wiki/Minimax - Date of Access: 11/22/19
https://en.wikipedia.org/wiki/Alpha–beta_pruning - Date of Access: 11/22/19