Report: Artificial Intelligence / Assignment 2 Connect 4

Group members: Jia Jun Yong (s3688090) / Vincent Hong Wei Heng (s3643760)
Equal Contribution (50/50)
Team Ayyy I: https://github.com/yongjiajun/ConnectFour-AI-Bot

Agent Implementation:

First of all, we have added Alpha-beta pruning functionality into the provided *dfMiniMax()* function to speed up the process of selecting the best values for the Minimax algorithm. In that function, we have also added code that calls *evalauteBoardState()* instead of using MiniMax to provide the best move when the search depth is bigger than current depth that the player can move to prevent the agent from crashing when it is being depth limited.

In the *evalauteBoardState()* function, we use the provided built-in functions from *board.py* such as *board.update_scores()* to update each player's score using *board.score_array[id][]* array through reading each value in the *board.board[x][y]* array. Since there are 69 winning possibilities in Connect 4 with the default height and width board with all vertical, diagonal and horizontal winning possibilities taken into account, in *board.update_scores()* function, it evaluates the board by obtaining a *win_index* which can be from 0 to 68 from *winning_zones[x][y][]* array each loop iteration using specified coordinates, then updating *board.score_array[id][]* array of the current player with the help of *win_index* to help locate the value in the array. We're using this method because it provides the most accurate evaluation of the board, no matter the discs are placed vertically, diagonally or horizontally in the entire board without missing out any winning combinations.

Then, we have added a function called *scoreCalculate()*, it calculates the total scores of available winning steps of each player by looping through the score array of each player.

```
def scoreCaculate(self, board, id):
    p = 0
    for i in range(len(board.score_array[id-1])):
        if (board.score_array[id-1][i] == 4):
            p += 12500
        elif (board.score_array[id-1][i] == 3 and board.score_array[id%2][i] == 0):
            p += 198
        elif (board.score_array[id-1][i] == 2 and board.score_array[id%2][i] == 0):
            p += 72
        elif (board.score_array[id-1][i] == 1 and board.score_array[id%2][i] == 0):
            p += 9
        elif (board.score_array[id-1][i] == 0 and board.score_array[id%2][i] == 0):
            p += 1
    return p
```

Figure 1: code for *scoreCalculate()* function

As you can see in figure 1, a fixed score distribution that has been optimised is used in this matter to provide an accurate calculation of the probability score. In each iteration of the loop, it adds a higher score to a total score variable if the player has a higher chance of winning and a lower score if the player has a lower chance of winning. This function is used for calculating the scores for both players so that we can calculate the **winning probability score for each move** of the current player for winning against the opponent by using the formula ***p/(p+pf)*** where ***p*** is the current player's score and ***pf*** is the opponent's score. The probability score calculated can only be in the range of 0 to 1. If both ***p*** and ***pf*** are 0, a probability score of **0.5** will be used as both players have equal chances of winning the game. If the probability score calculated is closer to 1, meaning the current player has a higher chance of winning than the opponent, but if the probability score calculated is closer to 0, it means that current player has a lower chance of winning than the opponent. This probability score value is returned back to its caller function so that the move with the **highest winning possibility** is chosen.

Finally, we have the *maxDepth* value set to 4 to optimise performance and time taken. Initially it was set to 5, however, we still find that *maxDepth* 4 provides the best agent performance and does not run over the time limit. We will explain more about it in the *Performance Analysis* section.

Performance Analysis:

Let's talk about our agent performs best when *maxDepth* is set to 4. We've set up a test to have 2 of our same copy of agent but with different depths to fight against each other, and these are the results.

	Player 1 won	Player 2 won
maxDepth 5 vs *maxDepth* 4		✓
maxDepth 3 vs *maxDepth* 4		✓
maxDepth 2 vs *maxDepth* 4		✓
maxDepth 6 vs *maxDepth* 4		✓

Table 1: Battles of Agent with Different *maxDepth* Value

As we can see in Table 1, *maxDepth* 4 produces the best agent. But you may be wondering why we have been preferring the use of our agent as player 2 in table 1. Take a look at the following table:

	Player 1 won	Player 2 won
maxDepth 3 vs *maxDepth* 4		✓
maxDepth 4 vs *maxDepth* 3	✓	
maxDepth 4 vs *maxDepth* 4		✓
maxDepth 4 vs *maxDepth* 6		✓
maxDepth 4 vs Friend's agent		✓
Friend's agent vs *maxDepth* 4		✓

Table 2: Battles of Agents

As seen in table 2, with *maxDepth* 4 being the strongest, our agent got beaten as player 1 for some unexplainable issues in rare situations such as the ones in table 2 even though we understand that in Connect 4, player 1 is given the winning advantage. One of my friend's agent also behaves weakly as player 1 but not as player 2, we are unsure why and still looking for a solution to that. This is one of the weaknesses and it can definitely be improved.

Another reason why we chose *maxDepth* 4 is because of time constraint. In contest 3, we were using *maxDepth* 5 and it performed poorly in terms of terrible play style and the long time taken to process for the next move, we're talking an average time of 24 seconds when played against staff bots and didn't get high enough scores when playing against other student agents. However, in contest 4 and 5, we've used *maxDepth* 5, which has shorten the average time taken (6.3 seconds) and performed well against other student agents.

After all the experiments and testings, our optimised agent now has a balanced play style for both defending and aggressing.

Without alpha-beta pruning, our agent with *maxDepth* 4 takes approximately 1 second to generate the best move. However, once alpha-beta pruning is implemented, the time taken **gets cut down to half**, taking only around 0.5 seconds to generate the best move.

That being said, our agent is not perfect. As discussed earlier, we should do more research to look for better heuristics to improve its play style so that our agent can utilise its full power to provide the absolute best answer for each move, hence destroying all of its opponents. Furthermore, we should also improve the score distribution as seen figure 1 that is used for calculating **winning probability scores** by using a different score set that is closest to its optimal state for providing a more accurate solution. Finally, a more advanced evaluation technique can be used for evaluating the board to produce a more accurate winning probability. For example, here are two Connect 4 blocks filled twice:

O	O		

	O	O	

Our current implementation will produce the same winning probability for both blocks, but a more advanced evaluation may produce different winning probabilities for both blocks to increase accuracy.