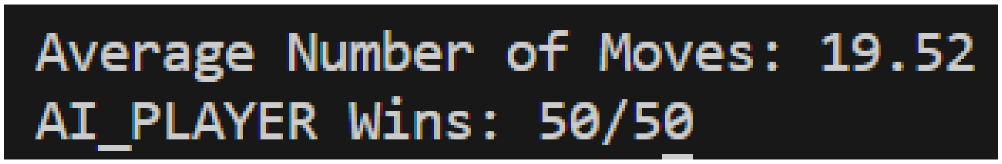
Assignment 2 - Connect 4

Evaluation Functions and Analysis: (Part a and b)

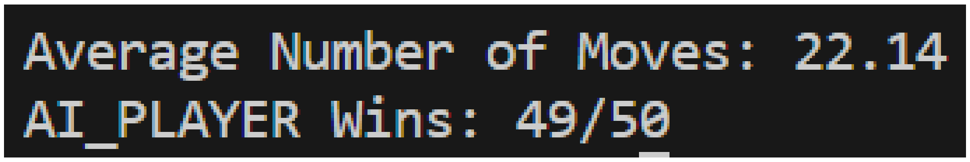
I considered multiple different evaluation metrics, whose description and results have been mentioned below is descending order of performance (Lower number of moves and higher number of wins is considered better):

1. We can consider all windows of 4 (in all directions i.e. horizontally, vertically, diagonally on both negative and positive sides) for all the positions in the game such that the window has pieces of only one of the players (can have empty cells). Count the number of cells with the player piece and calculate the weighted sum of windows with 4, 3 or 2 pieces. Only the condition of the opponent winning is considered here and is assigned a very large negative value, no other opponent score is subtracted.
   Parameters that can be tuned: weights assigned to count
   Weights assigned to case where the window is fully composed of one player is a termination state as one of the players have won - so we assign this as a very high weight to bias the score.



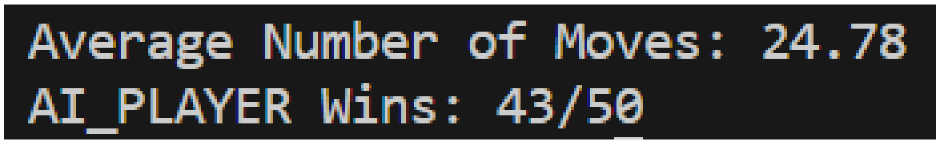
```
Average Number of Moves: 19.52
AI_PLAYER Wins: 50/50
```

2. Count the number of possible four-in-a-rows. The computer assesses the possible winning alignments available on the current game board, determining the count of ways it could win, and then subtracting the number of ways the computer's opponent could win. This can be done by considering all windows of 4 as done above but scoring is simply a sum of such windows (except when in winning state) subtracted with the opponent's score.



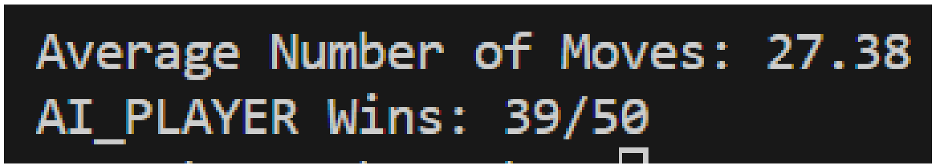
```
Average Number of Moves: 22.14
AI_PLAYER Wins: 49/50
```

3. Count the number of sequences of twos, threes and fours of the players piece and compute a weighted score after summation (weight of four >>>> weight of three > weight of 2). The final score is calculated by subtracting the opponent's score obtained using the same heuristic.
   Parameters that can be tuned: weights assigned to count



```
Average Number of Moves: 24.78
AI_PLAYER Wins: 43/50
```

4. A simple heuristic function that assigns positive weight to winning, negative weight to losing and 0 otherwise.



```
Average Number of Moves: 27.38
AI_PLAYER Wins: 39/50
```

(The above results are for depth = 5 with alpha-beta pruning)

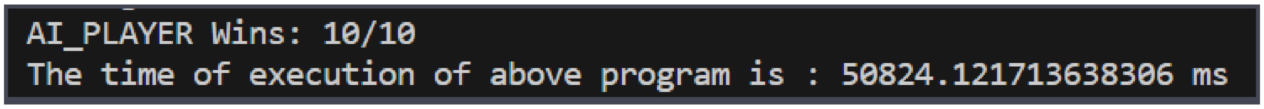
As visible, the first 2 evaluation functions perform well. Method 3 doesn't perform as well as it considers only consecutive sequences of pieces, which is a false representation of the chances of winning as oftentimes fours are made by connecting disconnected sequences. The fourth evaluation function is too reductive and should be used only to improve time performance.

Minimax function without alpha-beta pruning v/s with: The performance in terms of accuracy doesn't change by introducing the pruning mechanism, but it significantly improves the time taken to produce results. With alpha-beta pruning, we can eliminate evaluating redundant branches thus leading to faster execution. Without pruning, running minimax up to a depth of 5 was extremely time-consuming and not practical, but with pruning, it was much faster.

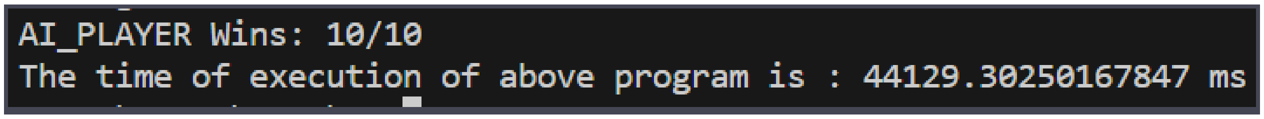
Move Ordering: (Part c)

Move ordering heuristic is used to prioritize and explore certain moves before others during the minimax search. The goal is to improve the efficiency of the search by considering more promising moves early on. In my code, the heuristic is based on the evaluation of winning positions after making each move. It counts the number of potential winning positions for the player after making the move. The count is based on potential winning sequences in the horizontal, vertical, and diagonal directions. The minimax search explores moves with higher heuristic values first. Thus branches that are unlikely to lead to a better outcome are pruned earlier on.

Here's a time comparison between Minimax without move ordering (up) and with move ordering (down) for 10 games. Here, the time calculated is the total time to complete 10 games.



```
AI_PLAYER Wins: 10/10
The time of execution of above program is : 50824.121713638306 ms
```

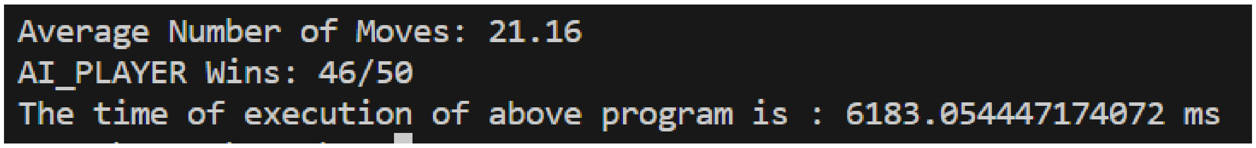


```
AI_PLAYER Wins: 10/10
The time of execution of above program is : 44129.30250167847 ms
```

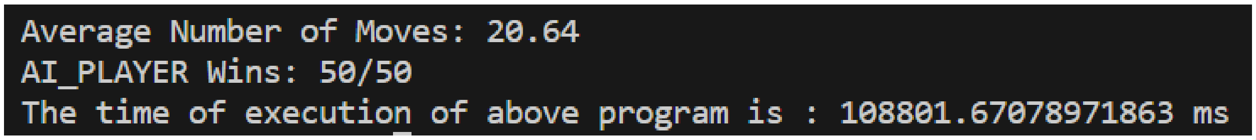
Comparing cut-off depth : (Part d)

Below are the performance metrics attached for depth = 3 (up) and depth = 5 (down)

1. Using evaluation function 1 (good performance)

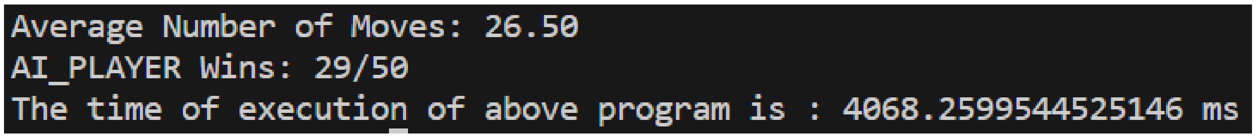


```
Average Number of Moves: 21.16
AI_PLAYER Wins: 46/50
The time of execution of above program is : 6183.054447174072 ms
```

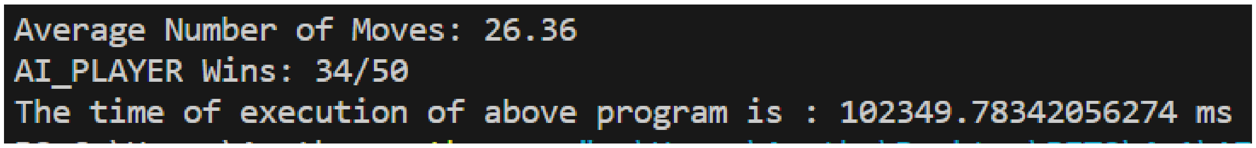


```
Average Number of Moves: 20.64
AI_PLAYER Wins: 50/50
The time of execution of above program is : 108801.67078971863 ms
```

2. Using evaluation function 2



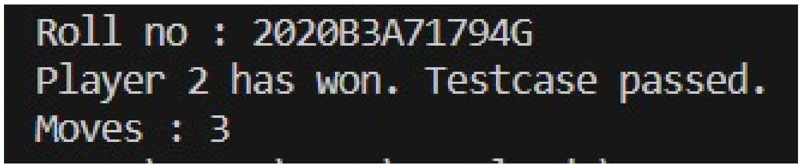
```
Average Number of Moves: 26.50
AI_PLAYER Wins: 29/50
The time of execution of above program is : 4068.2599544525146 ms
```



```
Average Number of Moves: 26.36
AI_PLAYER Wins: 34/50
The time of execution of above program is : 102349.78342056274 ms
```

Increasing the cut-off depth improved both the average number of moves and the number of wins the AI could achieve. The average number of moves remains around the same but there’s a significant difference in the number of wins. This came at a significant cost of time as the time taken by the code to run is more than 17x the time it took for depth = 3 to run.

Screenshot of test case:



```
Roll no : 2020B3A71794G
Player 2 has won. Testcase passed.
Moves : 3
```