GAME-TREE BASED CONNECT 4 GAME

ASSIGNMENT 2 REPORT

Vidhi Kabra

Student ID: 2020B3A70568G

Assignment 2 Report

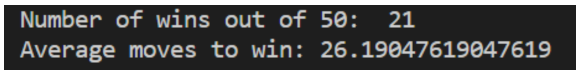
CS F407 Artificial Intelligence

November 2023

Contents

1 Comparing various evaluation functions

evaluation_function_one is a basic evaluation heuristic that checks if any winner has been declared, if not difference between the number of pieces each player has on the board is returned. If GameTreePlayer has won, a score of 5000 is returned to suggest that moves leading to victory are preferred above everything else. When MyopicPlayer wins, a score of -1000 is returned indicating that losing is a significantly unfavourable scenario and such a move is highly discouraged.

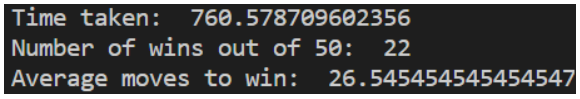


```
Number of wins out of 50:  21
Average moves to win: 26.19047619047619
```

Figure 1: Output for *evaluation_function_one*

evaluation_function_two is a simple scoring function to evaluate the game state in the terminal conditions: depth limit is reached or a game-ending move has been played. Here is how it works:

- If the GameTreePlayer is the winner in the current game state, a score of 1000 is returned. This is a highly favorable outcome for it, indicating that the move was a game-winning move.
- Conversly, if MyopicPlayer is the winner, a score of -1000 is returned. The large negative score indicates that the move was very unfavourable and GameTreePlayer should avoid such outcomes.
- When the game is not finished or is drawn and neither player has won yet, the function returns a score of 0. This indicates that the game state does not clearly help or hurt the GameTreePlayer's chances of winning.



```
Time taken:  760.578709602356
Number of wins out of 50:  22
Average moves to win:  26.545454545454547
```

Figure 2: Output for *evaluation_function_two*

evaluation_function _three is a more complex heuristic to evaluate the game state from the perspective of the GameTreePlayer. Here is how it works:

- **Center Column Dominance** Given that its central location permits the greatest number of four-in-a-row combinations, the center column is given priority. To account for the strategic benefit of having this column occupied, the method counts the number of coins that the player has in the center column and assigns a score that is three times the count.

- **Horizontal Scores** It iterates each row for every possible "slice" of consecutive four cells and uses the *individual_score_function* to assign a score based on the number of player's coins, opponent's coins, and empty spaces within these windows.
- **Vertical Scores** It iterates each column for every possible set of consecutive four cells and uses the *individual_score_function* to assign scores.
- **Diagonal Scores** Both positive and negative diagonal are checked for possibilities of connect-four cells and given a score using *individual_score _function*.
- **Individual_Score_Function** This function is used to score a given "window" of four consecutive board positions using the following scoring logic: if a window contains 4 coins belonging to GameTreePlayer i.e. a guaranteed win, a score of 100 is given. If the given "window" has three coins and an empty space, a score of 5 is given. If the given "window" has two coins and two blank spaces, a score of 2 is given. If the given "window" contains 3 opponent coins and an empty space, a score of -4 is given to indicate that a blocking move needs to be played in the next turn and dismiss this immediate threat.

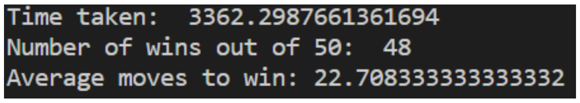

```
Time taken:  3362.2987661361694
Number of wins out of 50:  48
Average moves to win: 22.708333333333332
```

Figure 3: Output for *evaluation_function _three*

evaluation_function _four is a composite evaluation function that takes the weighted average of the above three evaluation functions. This is done to balance the basic count of pieces on the board, more complex strategies that consider the quality of piece placement (such as center control and potential linkages), and immediate game results (like winning or losing).

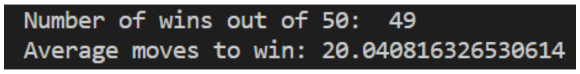
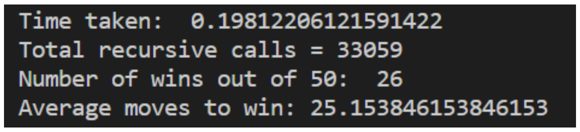

```
Number of wins out of 50:  49
Average moves to win: 20.040816326530614
```

Figure 4: Output for *evaluation_function _four*

The first evaluation function performs decently by winning 21 games which suggests it is moderately effective considering only the quantity and not the placement of pieces. The second evaluation function, which is a basic evaluation function assessing win, loss, or draw, achieves a slightly better outcome with 22 wins and takes marginally more moves on average (26.54) to secure a win. Based on its success, it can be concluded that although identifying winning states is useful, it remains a basic strategy lacking in strategic depth. The third function exhibits a significant leap in performance. The strategic value and patterns in the game board's center column are considered when evaluating it. Its performance indicates that it recognizes the value of board control and anticipates the opponent's move, effectively balancing offensive and defensive strategies. The effectiveness of the fourth evaluation function can be attributed to its weighted decision-making process, which adopts an all-encompassing perspective on the game state and produces more intelligent and practical decisions.

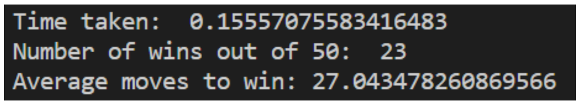
2 Implementing alpha-beta pruning.

The effectiveness of the game tree search is greatly increased by using alpha-beta pruning, which removes branches that do not require further investigation. Each of the four evaluation functions takes a different amount of time to run 50 games at a depth of three; evaluation functions one and two require less time (0.19 and 0.155 seconds, respectively) than functions three and four (0.78 and 0.75 seconds, respectively). Each evaluation function's calculation difficulty and intensity level accounts for this time discrepancy. Function three requires more time than function one's basic count or function two's win/loss/draw check since it has more strategic depth. Even at this level, function four, which is the most complicated, does not take noticeably longer than Function Three, indicating that its more complexity does not correspondingly increase calculation time.



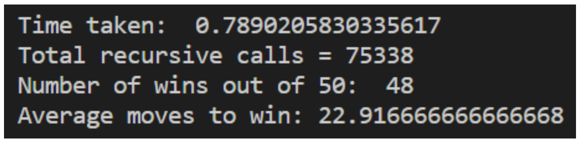
```
Time taken:  0.19812206121591422
Total recursive calls = 33059
Number of wins out of 50:  26
Average moves to win: 25.153846153846153
```

Figure 5: Output for*evaluation_function_one* and depth = 3



```
Time taken:  0.15557075583416483
Number of wins out of 50:  23
Average moves to win: 27.043478260869566
```

Figure 6: Output for *evaluation_function_two* and depth = 3

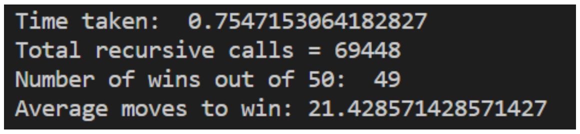


```
Time taken:  0.7890205830335617
Total recursive calls = 75338
Number of wins out of 50:  48
Average moves to win: 22.916666666666668
```

Figure 7: Output for *evaluation_function_three* and depth = 3

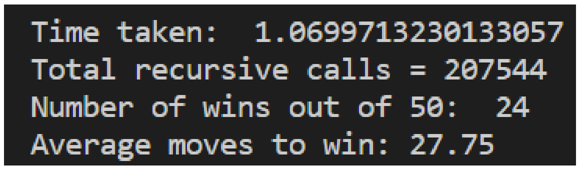
However, the performance patterns diverge more pronouncedly when the depth rises to 5. Evaluation function four's time jumps sharply to 9.03 seconds, suggesting a sharp rise in computation as the depth of search increases, while function three's efficiency stays at 0.422 seconds. This shows that exponentially more computing is needed for the increasingly complex analyses carried out by Function Four as depth increases.

Level 5's depth of search yields more victories for evaluation function four—a perfect record of 50 wins—than for function one, function two, and function three—which have fewer wins (24, 29, and 48). This is true even if the search takes longer. This illustrates



```
Time taken:  0.7547153064182827
Total recursive calls = 69448
Number of wins out of 50:  49
Average moves to win: 21.428571428571427
```

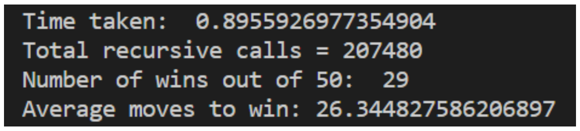
Figure 8: Output for *evaluation_function_four* and depth = 3



```
Time taken:  1.0699713230133057
Total recursive calls = 207544
Number of wins out of 50:  24
Average moves to win: 27.75
```

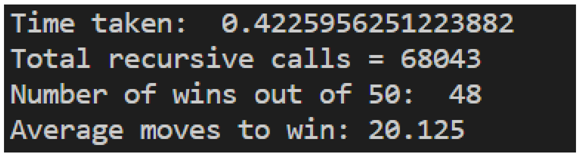
Figure 9: Output for *evaluation_function_one* and depth = 5

how search depth, time complexity, and winning results are traded off. Function four takes longer to complete tasks at a deeper level because it analyzes the game state more thoroughly. This results in a better procedure for making strategic decisions, which leads to more consistent wins.



```
Time taken:  0.8955926977354904
Total recursive calls = 207480
Number of wins out of 50:  29
Average moves to win: 26.344827586206897
```

Figure 10: Output for *evaluation_function_two* and depth = 5



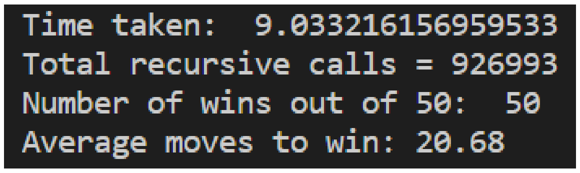
```
Time taken:  0.4225956251223882
Total recursive calls = 68043
Number of wins out of 50:  48
Average moves to win: 20.125
```

Figure 11: Output for *evaluation_function_three* and depth = 5

3 Implementing move ordering heuristic.

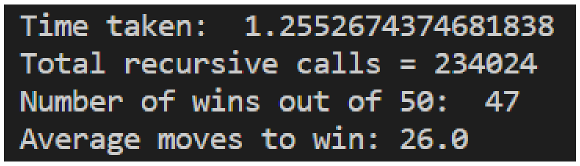
Figure 9 and 13 show the results for *evaluation _function _one* when depth is five and move ordering is not applied and applied respectively. We can observe that the move ordering heuristic has significantly impacted the performance of the first evaluation function by increasing the number of wins from 24 to 47. Because there is a higher chance of connections in the center column and positions near the center, it directs the search to prioritize these areas. This implies that the algorithm is already biased toward more advantageous moves, improving the likelihood of seeing winning opportunities early on and avoiding moves that could result in a loss. Similar observations can be made for *evaluation_function_two*.

Figures 11 and 13 show the results for *evaluation _function _three* when depth is five and move ordering is not applied and applied respectively. The application of move ordering along with *evaluation _function _three* has resulted in a significant reduction in the number of recursive calls made by the minimax function, as well as a decrease in the average duration to beat the MyopicPlayer. We know move ordering prioritizes the central column, ensuring that alpha-beta pruning is more effective and has earlier cutoffs in the search tree. The reduced number of recursive calls is evidence of this increased efficiency; since the algorithm prunes more branches earlier, it needs to make fewer calls overall. Because of this efficiency, decisions are made more quickly, as shown by the shorter time to victory. Similar analysis can be done for *evaluation _function _four* as we can observe significant decrease in time to victory and recursive calls.



```
Time taken:  9.033216156959533
Total recursive calls = 926993
Number of wins out of 50:  50
Average moves to win: 20.68
```

Figure 12: Output for *evaluation_function_four* and depth = 5



```
Time taken:  1.2552674374681838
Total recursive calls = 234024
Number of wins out of 50:  47
Average moves to win: 26.0
```

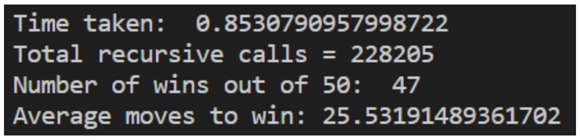
Figure 13: Output for *evaluation _function_one* with move ordering heuristic

4 Depth cut-off analysis (Depth = 3 vs Depth = 5)

The figure 17 shows the average moves to win and number of wins out of 50 games for the *evaluation _function_three* where the depth is three and five along with the application of move ordering heuristic. The program is run 10 times, in each run 50 games were played. It can be observed that with an increase in depth, the frequency of winning has increased along with a slight decrease in the number of moves to win. The higher frequency can be due to the following reasons:

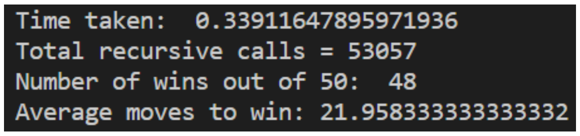
- The algorithm can see further into the future and consider more potential states that could result from present activities when it has greater depth level. This implies that the GameTreePlayer is making more intelligent decisions, which could result in more strategic play and, ultimately, more victories.
- The system can more accurately predict and plan for the opponent's moves with a deeper search, which could result in a more robust defensive and aggressive attack.
- Deeper search may also enable the algorithm to find and take advantage of long-term benefits that aren't visible at shallower depths, even at the expense of short-term rewards for advantageous placements.

The GameTreePlayer may select more cautious plays that prolong the game if the search space is more complicated at deeper levels, due to which the number of moves to win may remain almost the same or even increase.



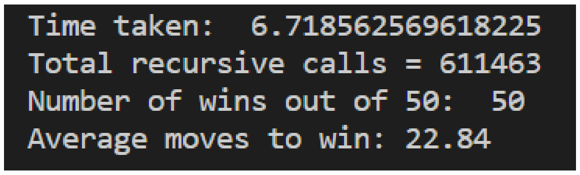
```
Time taken:  0.8530790957998722
Total recursive calls = 228205
Number of wins out of 50:  47
Average moves to win: 25.53191489361702
```

Figure 14: Output for *evaluation _function_two* with move ordering heuristic



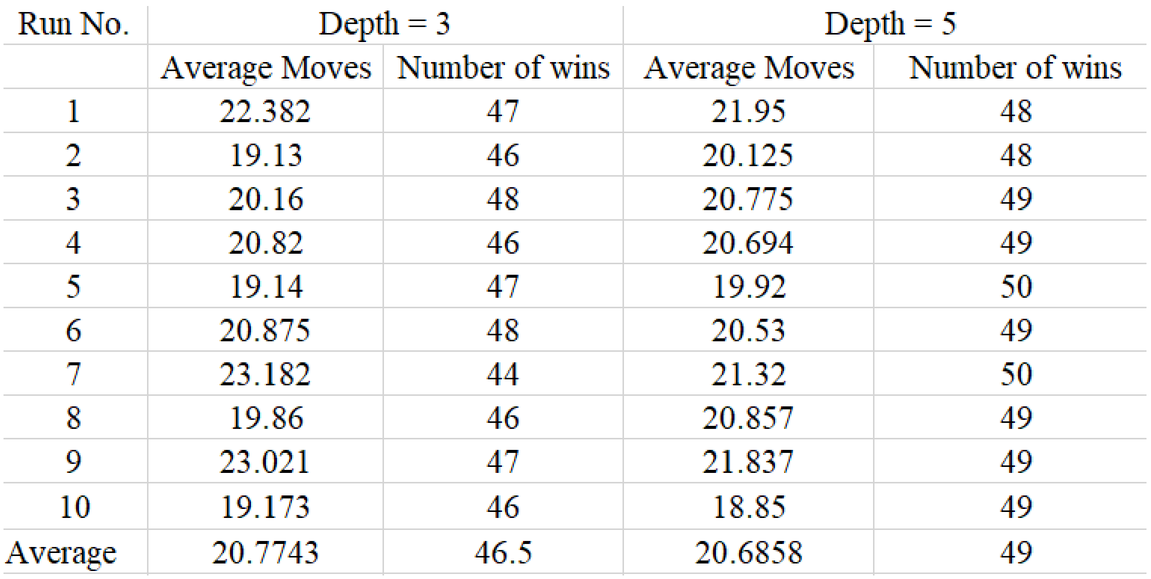
```
Time taken:  0.33911647895971936
Total recursive calls = 53057
Number of wins out of 50:  48
Average moves to win: 21.958333333333332
```

Figure 15: Output for *evaluation _function_three* with move ordering heuristic



```
Time taken:  6.718562569618225
Total recursive calls = 611463
Number of wins out of 50:  50
Average moves to win: 22.84
```

Figure 16: Output for *evaluation _function_four* with move ordering heuristic



Run No.	Depth = 3		Depth = 5	
	Average Moves	Number of wins	Average Moves	Number of wins
1	22.382	47	21.95	48
2	19.13	46	20.125	48
3	20.16	48	20.775	49
4	20.82	46	20.694	49
5	19.14	47	19.92	50
6	20.875	48	20.53	49
7	23.182	44	21.32	50
8	19.86	46	20.857	49
9	23.021	47	21.837	49
10	19.173	46	18.85	49
Average	20.7743	46.5	20.6858	49

Figure 17: Table comparing depths 3 and 5, *evaluation _function_three is used*