Checkers-AI Project Performance Evaluation

Alpha-Beta Agent Performance

Alpha-Beta agent's performance using search depths of 1, 3, 5, 8, and 10 are compared. Each search depth is tested on the alpha-beta agent with a total of three games each vs a human player, which is me trying my best to beat the agent. The winner for each game and the average time the agent computes one move (in milliseconds) is recorded. The results are tabulated in the table below:

Search Depth	Game #	Agent win/ lose?	Average time per move (ms)	Average time per move over all the games played (ms)
1	1	lose	1	1.33
	2	lose	1	
	3	lose	2	
3	1	win	7	6.33
	2	win	7	
	3	win	5	
5	1	win	31	27.67
	2	win	27	
	3	win	25	
8	1	win	211	203.67
	2	win	142	
	3	win	258	
10	1	win	1679	2168
	2	win	1341	
	3	win	1316	

- As we can see, as search depth increases, the average time needed for the agent to decide a move increases exponentially. This is because, as search depth increases, the number of leaf nodes at the bottom of the tree increases exponentially, which means the agent needs to go through exponentially more leaf nodes to decide the move with maximum utility and that increases computational time exponentially.
- Also, as we can see, as search depth increases, the harder it is to win against the agent. Although we can't really see that from my table of results since I have been dominated by the agent starting from search depth 3 and beyond due to my status as a beginner in checkers. Still, if the results are generated with an expert player, the difficulty level increase will become more obvious. The reason for the increase in difficulty is due to the feature that the search depth represents how many steps in the future the agent can see. The more steps into the future the agent can see, the more confidence the agent will know which move to choose next that will result in a more favorable winning condition.

The above results are generated using an evaluation function (which will be my **baseline** evaluation function) that gives the utility of the state based on how many red normal pieces, red king pieces, black normal pieces, and black king pieces there are on the state's board configuration. In my evaluation function, a normal piece (whether red or black) has a value of 1, whereas a king piece (whether red or black) has a value of 3. The score is initially 0, and it:

- Increases by the normal piece value for each agent's (BLACK) piece.
- Decreases by the normal piece value for each human (RED) player's piece

- Increases by the king piece value for each agent's (BLACK) king piece.
- Decreases by the king piece value for each human player's (RED) king piece.

Lastly, the score is divided by 36 since that is the maximum possible score that the agent can have, which is having all 12 black checkers pieces as kings and having no red checkers pieces on the board (12 king piece = 12 * 3 = 36). This normalization is necessary to restrict the output utility value to be between -1 and 1, since they are the agent's utility values for losing and winning, respectively.

However, one observation from using this evaluation function is that when the agent's (BLACK) normal piece turns into a king piece, the king pieces keep hovering around the last few rows, with little intention to attack the lone red piece staying on the top few rows. This can result in a draw even though there is clearly an advantage in numbers for the agent. Therefore, to clear out of this situation, we need to add more scoring mechanism to our previous evaluation function. For instance, we want the agent's (BLACK) king pieces to close in on their opponent (RED) pieces only when there are more agent (BLACK) pieces on the board than the human (RED) pieces. Thus, we can score based on how close the agent's (BLACK) king pieces are to their closest enemy (RED) pieces when the agent has more pieces on the board by using Manhattan distance as my heuristic function.

In my **improved** evaluation function, which is built upon my previous evaluation function, if there are more agent (BLACK) pieces than enemy (RED) pieces, the agent's (BLACK) king pieces sort of go on an offensive mode, where agent's (BLACK) king pieces are forced to close up onto their nearest enemy (RED) pieces in an attempt to eliminate those pieces to prevent a long-drawn battle. Here, I added the scoring mechanism that only applies when there are more agent (BLACK) pieces than human (RED) pieces:

- For every agent's (BLACK) king pieces, find its Manhattan distance *d* with the closest enemy (RED) piece (whether normal or king), and increase the score by (1 / *d*). This way, the score is higher when the agent's (BLACK) king piece is closer to its nearest enemy (RED) piece, which gives the agent a better utility value when it chooses a move that closes up the distance between its king piece and the opponent's pieces.
- Despite the change in the scoring system, we are still normalizing the final score over 36 (12 king piece = 12 * 3 = 36) as the score increase in score will never go beyond 36. This is because the increase in score (1 / *d*) is always at most 0.5 as the minimum Manhattan distance that a piece can achieve with one another (diagonal with each other) is 2 (1 / 2 = 0.5). For there to actually have a increase in score based on this scoring system, there has to exist a king piece for the agent (BLACK) and a piece (normal or king) for the human (RED). Since we are decreasing the score by 1 for every normal human (RED) piece and 3 for every king human (RED) piece, we will be decreasing the overall score by at least 0.5 for every normal human (RED) piece and at least 2.5 for every king human (RED) piece, which means that the overall score will not be exceeding 36.

If the number of agent (BLACK) pieces is less than or equal to the number of human (RED) pieces, this evaluation function basically reverts back to the previous evaluation function where we are only scoring based on the counts of agent (BLACK) and human (RED) pieces. To compare the performance between the previous evaluation function and the improved evaluation function, I tested the alpha-beta search agent with both evaluation functions at a search depth of 8 (since it has plays very optimally while taking less than a second on average to make a move), with total of three games for each evaluation function. The human player in this case is still me and I will compare the average number of moves per game needed for the agent to beat me while I play as optimally as I can (sub-optimally). The results are:

Evaluation Function	Game #	Number of moves taken to beat me	Average number of moves taken to beat me in three games.
Baseline	1	50	47.333
	2	37	
	3	55	
Improved	1	25	25.333
	2	29	
	3	22	

From my table of results, we can see that the improved evaluation function helps the agent to devise a plan to achieve a win against me in much less steps since the agent now smartly uses its advantage in numbers to close into the enemy pieces that it wants to eliminate instead of aimlessly hover its king pieces at the bottom row, waiting for the human player to move its pieces slowly towards them.

MCTS Agent Performance

I will compare the performance of the MCTS agent using the theoretical optimal value C = $\sqrt{2}$ and C = 0 for the upper confidence bound formula by playing against the agent for a total of three games for each value of proposed C. The results are:

C = 0

```
Number of childrens: 7

Node 1:
        wins: 1.0
        playouts: 3.0
        Win ratio: 0.3333333333333333
        Move taken: (2,0) -> (3,1)
Node 2:
        wins: 0.0
        playouts: 1.0
        Win ratio: 0.0
Node 3:
        wins: 0.0
        playouts: 1.0
        Win ratio: 0.0
Node 4:
        wins: 0.0
        playouts: 1.0
        Win ratio: 0.0
Node 5:
        wins: 566.0
        playouts: 857.0
        Win ratio: 0.6604434072345391
        Move taken: (2,4) -> (3,5)
Node 6:
        wins: 2.5
        playouts: 6.0
        Win ratio: 0.4166666666666667
Node 7:
        wins: 62.5
        playouts: 131.0
        Win ratio: 0.4770992366412214

------> Chosen move at node 5
```

C = $\sqrt{2}$

```
Number of childrens: 7

Node 1:
        wins: 37.0
        playouts: 90.0
        Win ratio: 0.4111111111111111
        Move taken: (2,0) -> (3,1)
Node 2:
        wins: 98.0
        playouts: 184.0
        Win ratio: 0.532608695652174
        Move taken: (2,2) -> (3,1)
Node 3:
        wins: 109.5
        playouts: 202.0
        Win ratio: 0.5420792079207921
        Move taken: (2,2) -> (3,3)
Node 4:
        wins: 27.0
        playouts: 73.0
        Win ratio: 0.3698630136986301
Node 5:
        wins: 68.5
        playouts: 140.0
        Win ratio: 0.48928571428571427
Node 6:
        wins: 103.5
        playouts: 193.0
        Win ratio: 0.5362694300518135
Node 7:
        wins: 54.5
        playouts: 118.0
        Win ratio: 0.461864406779661

------> Chosen move at node 3
```

C	Game #	Agent win/ lose?
0	1	Lose
	2	Win
	3	Lose
$\sqrt{2}$	1	Win
	2	Win
	3	Win

Here, we can see that:

- When C = 0, the agent will select the node that gives the highest winning ratio (# wins/ # playouts) most of the time to expand and simulate, which is why there is one transition node that has a very large amount of playout while the other transition nodes have very few playouts. This is because the exploration term is multiplied by 0, which essentially means that the agent is not doing any exploration and is only exploiting the move that it thinks that has the highest value. Without exploration of some previously unexplored path, the agent might miss the true optimal path, which causes it to play sub-optimally. When I play against the agent for this case, there are some cases when the agent chooses a move that gives its opponent (me) more advantages, which is why I was able to win 2 out of 3 games against this agent.
- When C = $\sqrt{2}$, the agent will balance out between exploitation and exploration when selecting a node to expand and simulate, which is why all possible transition nodes have almost the same number of playouts, meaning each possible moves are experienced with almost the same number of times. Since each possible moves are experienced sufficiently, the agent has a general idea of which move to choose that leads to a greater chance of winning in order to play optimally. Therefore, I wasn't able to win against this agent at all.

As a result, the MCTS agent with the theoretical optimal C = $\sqrt{2}$ for the upper confidence bound performs way better than C = 0, which signifies the importance of exploring paths that have not been explored as there might be a winning path among those paths.

Performance Comparison Between Different Agents

Lastly, I will compare the performances by the three different agents: alpha-beta search, MCTS, and hybrid. The configuration for each agent is as follows:

- **Alpha-Beta-Search Agent**:
  - Search depth = 8
  - Normal piece value for evaluation function = 1
  - King piece value for evaluation function = 3
  - Evaluation function used: **Improved** version
  - The game is considered a draw if there are no piece being capture for 40 moves straight (20 moves from both players). However, that will require a search depth of at least 40 to be implemented, which is not computationally efficient. Therefore, there is no draw implemented for this agent.
- **MCTS Agent**:
  - C = $\sqrt{2}$
  - Number of simulations = 1000
  - The game is considered a draw if there are no piece being captured for 40 moves straight (20 moves from both players).

- **Hybrid Agent**:
    - Uses the configuration of Alpha-Beta-Search Agent and MCTS Agent proposed above.
    - Randomly select which agent to decide a move with a probability of 0.5 for each.

Each agent will be played against a human player (me) for a total of five games each. The winner of each game will be recorded and the average time needed for the agent to make a move is also recorded. The results are:

Agent	Game #	Agent win/lose?	Number of moves taken to beat me	Average time per move (ms)	Average number of moves taken to beat me in five games.	Average time per move over all the games played (ms)
Alpha-Beta Search	1	win	26	184	24.6	186.2
	2	win	23	135		
	3	win	24	217		
	4	win	29	223		
	5	win	21	172		
MCTS	1	win	33	526	31.6	478.6
	2	win	20	476		
	3	win	50	421		
	4	win	32	496		
	5	win	23	474		
Hybrid	1	win	23	225	23.2	262
	2	win	19	272		
	3	win	29	237		
	4	win	24	296		
	5	win	21	280		

Overall, the Alpha-Beta Search performed best in terms of computational time to decide a move, whereas the hybrid agent performed best in terms of the fastest to beat me over 5 games. In this case, it seems like the Alpha-Beta search agent performs best overall since it is the fastest at generating a move and the number of moves it takes to beat me is not much greater than the Hybrid agent. Since the branching factor of the checkers game is not that high, Alpha-beta search shines more than MCTS in this case since it can search for the optimal move to execute in a few milliseconds (due to the small branching factor). On the other hand, MCTS has to perform *N* (1000 in my case) simulations of playouts, which is my it takes more time to generate a move. Also, since MCTS randomly selects a move for both players during simulation, there's a chance for the agent to select a bad move that has high win rate during simulation but not during the actual game (where the opponent is playing optimally). A higher number of simulations *N* will eventually converge and the agent will most likely always choose the move that is actually the optimal one, but it does come with the cost of more computational time. Lastly, since the hybrid agent is a mix between the Alpha-Beta search agent and the MCTS agent, its performance is expected to be somewhere between the two agents, which is shown by the results table.

However, since these data are generated using me, a beginner, as the opponent of the agent, they might not be that reliable since I might sometimes play sub-optimally and make a bad move that gives the agent more advantage (which is why I lost all the games played). Thus, having an expert human player in checkers might help evaluate the agent's performance more accurately.