Checkers: Minimax Algorithm with Alpha-Beta Pruning

Abstract

There are many articles on the Internet about the development of algorithms and AI for playing Chess, however, similar articles about algorithms and their sequential development for playing Checkers are not so common. The idea of this project is to study and implement the board game of Checkers using the Minimax algorithm, optimized by Alpha-Beta pruning, and effectively compare the performance of Minimax alone with the introduction of Alpha-Beta pruning. The Minimax algorithm is a recursive decision-making algorithm that utilizes a tree structure and is commonly used in two-player games. It assumes that both players are playing optimally and tries to minimize the maximum loss that the player can face. Alpha-Beta pruning is an optimization technique that significantly reduces the number of paths or nodes evaluated by the Minimax algorithm. The practical part of the work will be done in Python, including visualization of the game board.

What are Checkers?

Checkers is a logical board game for two players. There exist a lot of different variations of Checkers rules but here are the ones we chose while designing the game and the algorithm:

The composition of the game:

1. The game board has 64 (8x8) cells. The cells are of two contrasting colors, usually white and black, arranged diagonally.
2. Checkers of two different colors (also white and black) of 12 pieces each.

The purpose of the game:

A win is considered when the opponent has no checkers left, the opponent's checkers are blocked or the opponent prematurely admits defeat.
If it is impossible for any of the participants to win the game, the game is considered finished in a draw.

Rules:

- 2 players take part in the game. The players are positioned on opposite sides of the playing field (board).
- The board is positioned in such a way that the corner dark cell is located at the left side of the player.
- The choice of color by the players is determined by lot or by agreement. Checkers are placed in three rows on dark squares closest to the player. The right of the first move usually belongs to the player who plays with white checkers. The moves are made by the opponents in turn.
- At the beginning of the game, all the opponents' checkers are simple. Simple checkers can only be moved forward diagonally to the adjacent free square.
- If a simple checker has reached the last row of the opposite side of the board, it becomes a “king” and is indicated by turning over (in our case, the crown icon). The king can walk one field diagonally forward or backward.
- The opponent's checker is captured by moving the player's own checker over it. It is only possible in case it is located on a diagonal square adjacent to the opponent’s checker and there is an empty field behind it. Capturing an opponent's checker with a simple checker can only be done forward (the king checker can capture the opponent's checker in any direction). Capturing the opponent's checkers is not mandatory.

Based on these rules, we created a game board with the possibility of a game mode (Figure 1).

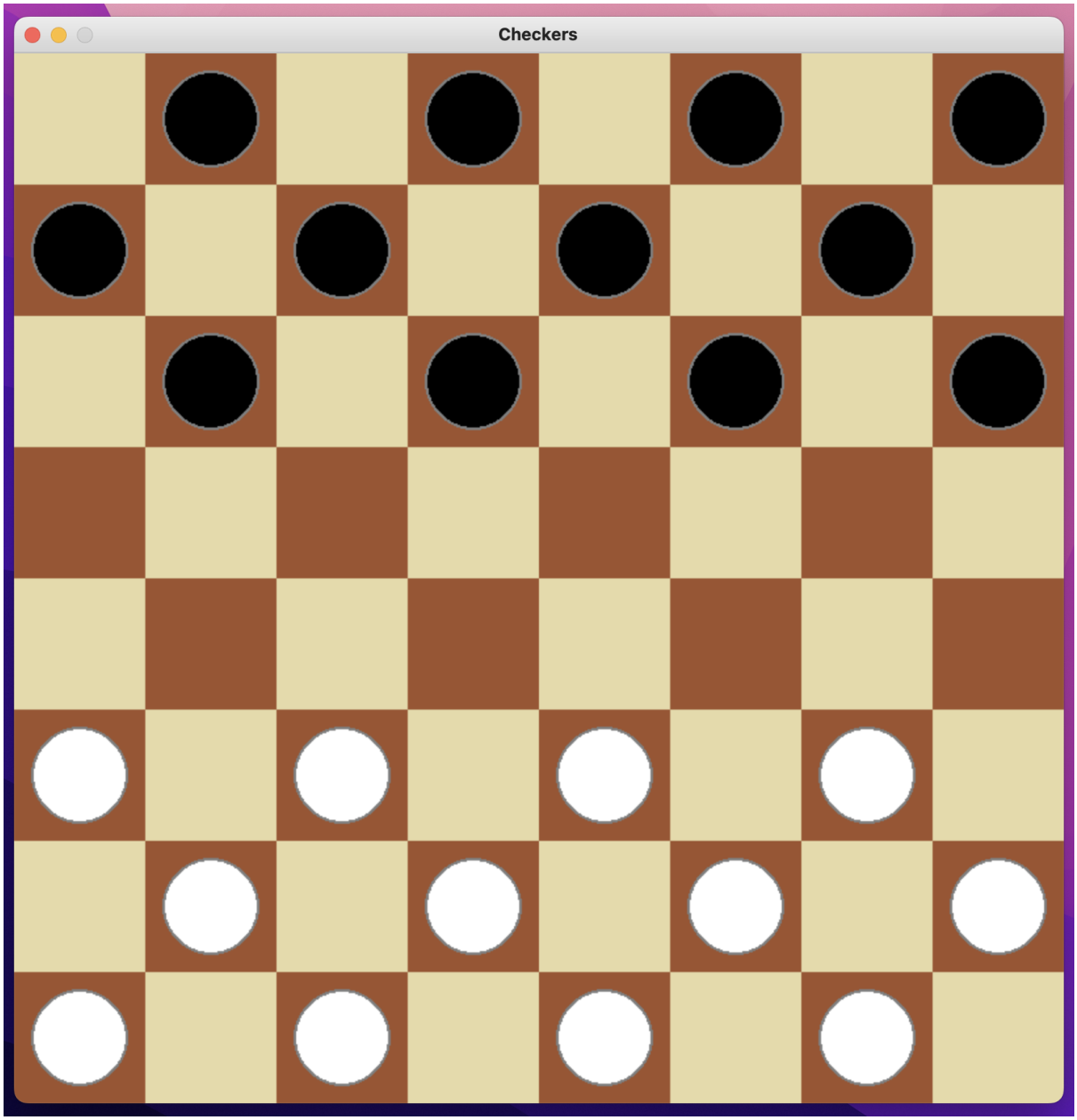

Figure 1 - Game Mode

When the player clicks on any checker, the blue dots demonstrate possible options for moves with that checker (Figure 2). In this case, a person plays for white checkers, while AI plays for black ones, based on our Minimax algorithm with Alpha-Beta pruning.

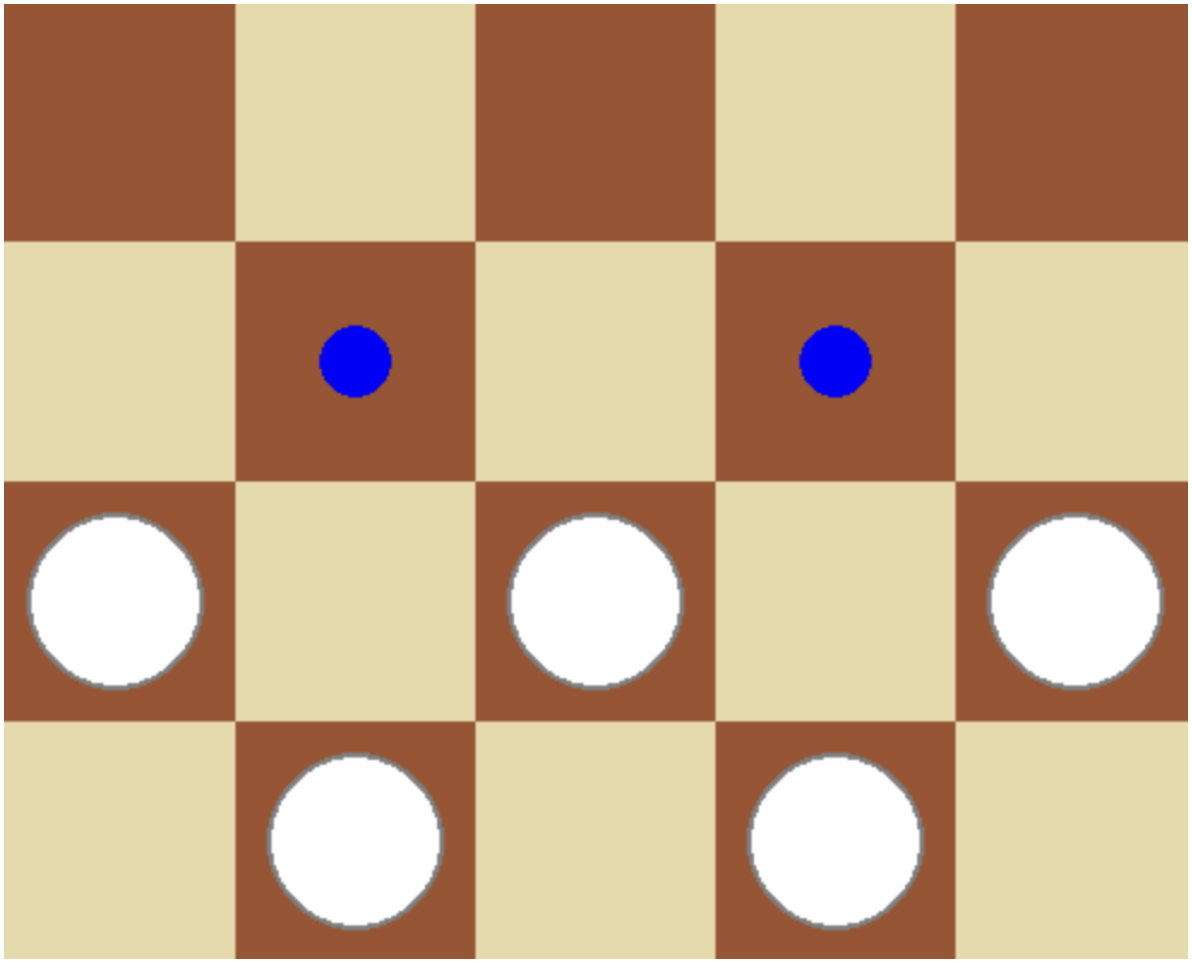

Figure 2 - Possible Moves

In this project, our aim is to deploy the Minimax algorithm alongside Alpha-Beta pruning to develop an AI adversary. We will then assess the algorithms' efficacy by contrasting its performance with Minimax alone, and with the inclusion of Alpha-Beta pruning.

What is Minimax?

Minimax is a decision-making algorithm based on the fact that a player chooses the best move for themself, that is, minimizes the chances of another player to win. Minimax can be thought of as a recursive method that plays moves on behalf of both players until a certain depth is reached, i.e., the number of subsequent steps, and then calculates how good the current position is (the weight of the position). This is done for all possible moves at the selected depth. The algorithm calculates the weights only at the edge of the branch (nodes). Then one can find the weight for the entire branch of the move, based on the fact that the player

wants to win, that is, tries to minimize the score, while AI tries to maximize the score. Therefore, if each minimax method call can be divided into minimization (player) and maximization (AI), it is easier to represent it as a tree consisting of a set of move options.

So, the way that the Minimax algorithm works for the game of Checkers is it assumes that the opponent is going to make the best possible move that it can. What we are going to do is consider every single possible move that we can potentially make, and then for each one of those moves, we are going to consider every single possible move that the other player can potentially make against us. Essentially, we want to minimize the score, and the other player wants to maximize it. So, what is the score? We can give a score for every single possible position on the game board. The easiest way to determine the score is to simply count the number of white checkers and the number of black checkers left on the board. So, we can say that:

score = #black_left - #white_left

The white player wants to make it as low as possible to minimize the score and has the greatest number of pieces, while black player (AI) wants to maximize that score. Thus, the algorithm assumes that when we make our move, we are going to minimize, that is to pick the move that has the best potential score for us, and black will pick its move that has the maximal potential score. Below is the algorithm illustration (Figure 3). We can go to infinite depth if we extend the tree even further but that also means that it gets exponentially larger, increasing the computations. Therefore, we are probably going to experiment with some small numbers of moves ahead (depth).

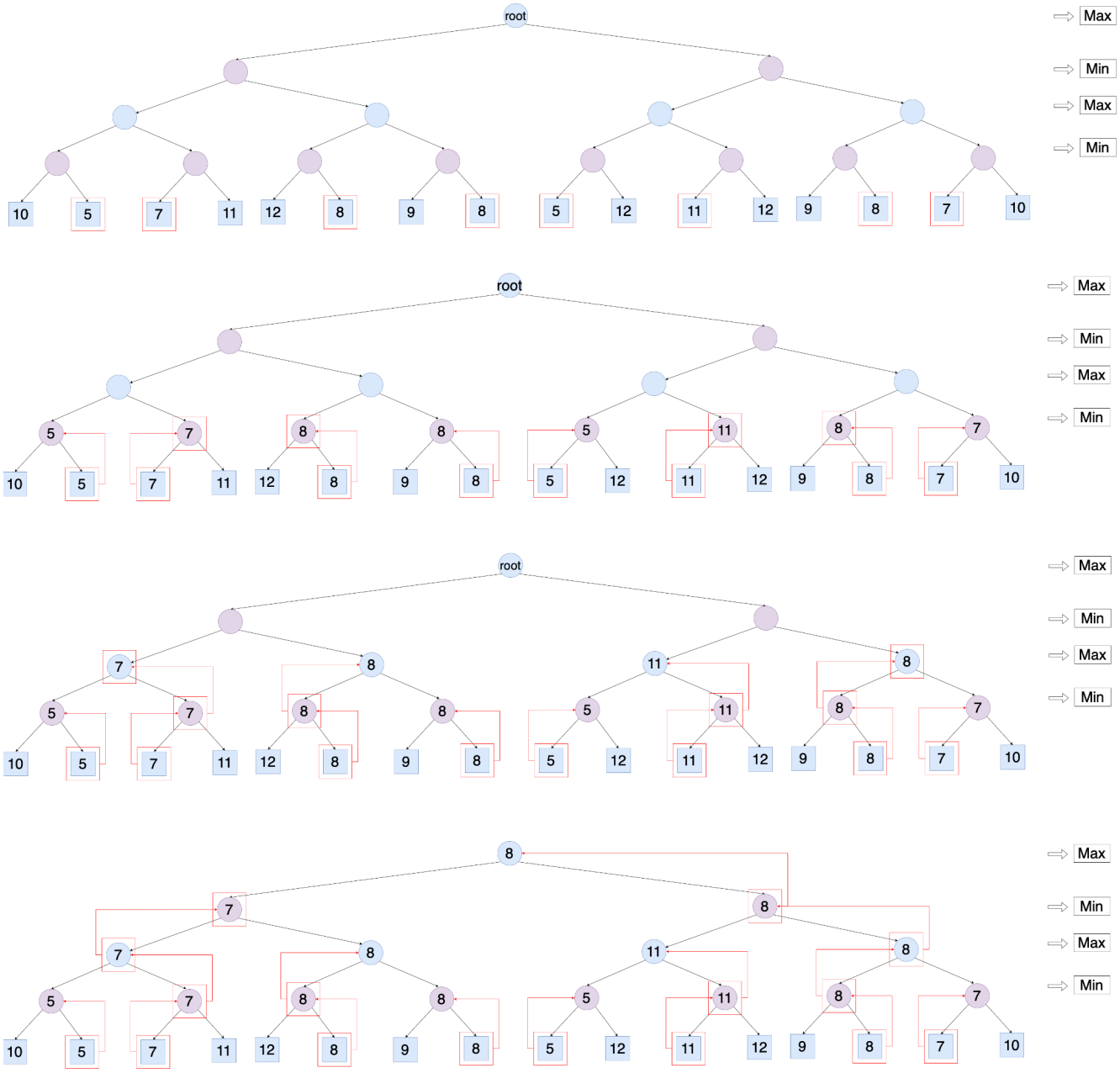

Figure 3 - Minimax Tree

Since the king is considered a more valuable checker (thanks to its maneuverability), we want to give it some additional weight in the score, and thus make the algorithm prioritize it. Therefore, the final score for our board has the following form:

score = #black_left - #white_left + (#black_kings * 0.5 - #white_kings * 0.5)

Note that the chosen coefficients for the kings can be any number, but we decided to go with 0.5 for several reasons. First of all, if the coefficient for kings was too high (e.g., 1 or greater), the algorithm could overly prioritize kings, leading to strategies overly focused on obtaining kings at the expense of other strategic considerations. On the other hand, if the coefficient was too low (e.g., 0.1 or less), kings might not be valued enough relative to regular pieces, diminishing their strategic importance. Secondly, in many games, including Checkers, the value of a king is often considered to be roughly equivalent to that of two regular pieces. By assigning a coefficient of 0.5 to kings in the scoring function, we are effectively giving them half the value of a regular piece. This maintains a consistent valuation framework where kings are valued proportionally higher but not excessively so.

The code for the Minimax algorithm can be found in CS566_Project/code/minimax/algorithm.py file.

1. minimax function: This is the core of the algorithm. It recursively evaluates the game state tree to determine the best move for the current player.

- It takes parameters:
  - position: the current state of the game;
  - depth: the depth of the search tree to explore;
  - max_player: a flag indicating whether it is the maximizing player's turn;
  - game: the game object representing the current state of the game.

- Base cases:
  - if the depth is 0 or there's a winner, it returns the evaluation of the position and the position itself.

- If it is the maximizing player's turn (max_player == True), it iterates through all possible moves, calculates their evaluations recursively, and selects the move with the maximum evaluation.

- If it is the minimizing player's turn (max_player == False), it does the same but selects the move with the minimum evaluation.

2. simulate_move function: This function applies a move to the board and updates it accordingly.

3. get_all_moves function: This function generates all possible moves for a given player on the current board state.

4. draw_moves function: This function visualizes the valid moves for a piece on the board. We have this function commented, however, if we want to see the AI logic process during the game, we can uncomment it.

Time Complexity: O(b^m)

where b - the number of legal moves at each point
m - the maximum depth of the tree

Alpha Beta Pruning

Alpha Beta Pruning is an optimization of the minimax algorithm. We know that minimax explores all possible states to find the best move. However, alpha-beta pruning significantly reduces the number of nodes evaluated by the minimax algorithm by pruning the branches that are guaranteed to be worse than previously explored branches. It eliminates the moves that the player would least likely make, in favor of the player. Hence by pruning these nodes, it makes the algorithm fast. Figure 4 illustrates the algorithm.

This is how alpha beta pruning works in minimax:

- Like in regular minimax, the algorithm explores the game tree to a certain depth, alternating between maximizing and minimizing player moves. At each level, it maintains 2 values:
    - Alpha, which represents the best value that the maximizing player can guarantee. The initial value is set to **−∞.**
    - Beta, which represents the best value that the minimizing player can guarantee. the initial value is set to **+∞.**
- The main condition followed throughout this algorithm is **alpha>=beta**
- Some key points to note about alpha-beta pruning:
    - Max player will only update the value of alpha
    - Min player will only update the value of beta

- While backtracking in the tree, the node values will be passed to upper nodes instead of those of alpha/beta

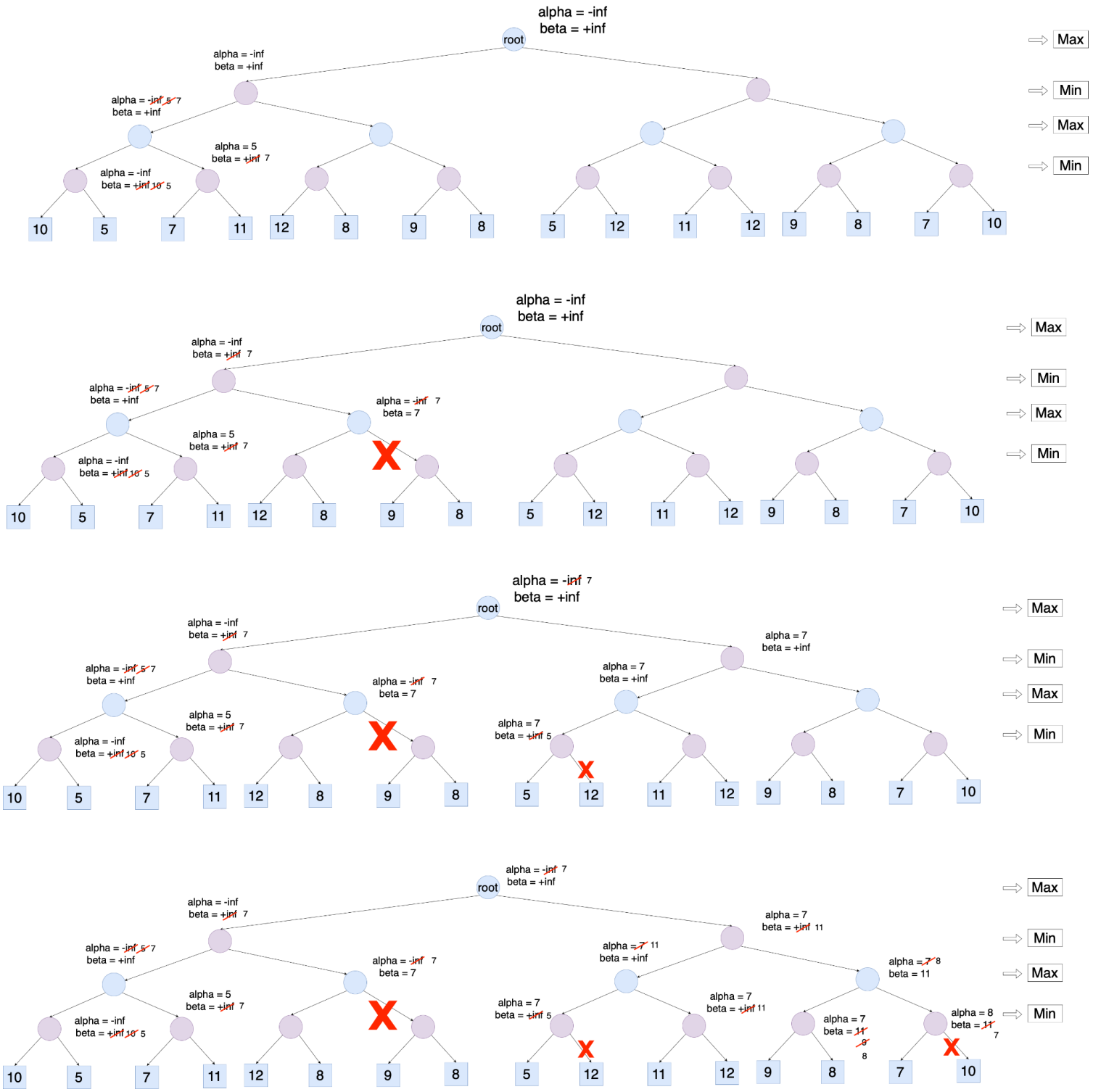

Figure 4 - Alpha-Beta Tree

The efficiency of alpha-beta pruning heavily relies on the sequence in which nodes are inspected. The arrangement of moves plays a crucial role in this pruning technique.

Two main types of move order can be distinguished:

1. Worst Ordering: In certain scenarios, the alpha-beta pruning algorithm fails to trim any branches of the tree, essentially mimicking the behavior of the minimax algorithm. This situation prolongs execution due to the factors associated with alpha-beta, termed worst ordering. Typically, the best move is located towards the right side of the tree. The time complexity in such cases is O(b^m).

2. Ideal Ordering: The optimal move order for alpha-beta pruning occurs when extensive trimming of branches takes place in the tree, with the best moves predominantly situated on the left side. By employing Depth-First Search (DFS), the algorithm prioritizes exploring the left side of the tree, effectively delving deeper twice as fast as the minimax algorithm within the same time frame. The time complexity in the case of ideal ordering is O($b^{m/2}$).

where b - the number of legal moves at each point
m - the maximum depth of the tree

Below are the comparison tables for the Minimax and Alpha-Beta Pruning algorithms. For each move, we present the number of nodes per each depth and the time taken for the AI to make a move in seconds. For the comparison, we decided to take depths 3, 4, 5 and 6.

Depth: 3

Minimax				**Alpha-Beta Pruning**			
White Move	**Black Move**	**Nodes Per Depth**	**Time Taken, (seconds)**	**White Move**	**Black Move**	**Nodes Per Depth**	**Time Taken, seconds**

5,2	4,7	{2: 54, 1: 7, 0: 1}	0.148	5,2	4,7	{2:13, 1: 7, 0: 1}	0.069
5,6	5,8	{2: 59, 1: 7, 0: 1}	0.162	5,6	5,8	{2:20, 1: 7, 0: 1}	0.080
5,4	4,5	{2: 90, 1: 8, 0: 1}	0.242	5,4	4,5	{2:48, 1: 8, 0: 1}	0.156
4,7	6,7	{2: 98, 1: 9, 0: 1}	0.270	4,7	6,7	{2:30, 1: 7, 0: 1}	0.107
5,8	5,6	{2: 86, 1: 8, 0: 1}	0.218	5,8	5,6	{2:32, 1: 6, 0: 1}	0.099
3,8	4,5	{2: 86, 1: 8, 0: 1}	0.244	3,8	4,5	{2:15, 1: 6, 0: 1}	0.063
AVERAGE		2: 79 1: 8 0: 1	0.214	**AVERAGE**		2: 26 1: 7 0: 1	0.096

Depth: 4

Minimax				**Alpha-Beta Pruning**			
White Move	**Black Move**	**Nodes Per Depth**	**Time Taken, (seconds)**	**White Move**	**Black Move**	**Nodes Per Depth**	**Time Taken, seconds**
5,2	4,7	{3: 406, 2: 54, 1: 7, 0: 1}	1.022	5,2	4,7	{3:57, 2: 13, 1: 7, 0: 1}	0.196
5,6	5,8	{3: 457,	1.178	5,6	5,8	{3:59,	0.196

		2: 59, 1: 7, 0: 1}				2: 14, 1: 7, 0: 1}	
5,4	4,5	{3: 731, 2: 90, 1: 8, 0: 1}	2.096	5,4	4,5	{3:76, 2: 19, 1: 8, 0: 1}	0.304
4,7	6,3	{3: 835, 2: 98, 1: 9, 0: 1}	2.386	4,7	6,3	{3:109, 2: 16, 1: 7, 0: 1}	0.338
5,4	4,5	{3: 820, 2: 99, 1: 8, 0: 1}	2.659	5,4	4,5	{3: 111, 2: 23, 1: 8, 0: 1}	0.378
3,6	4,1	{3: 582, 2: 71, 1: 7, 0: 1}	1.541	3,6	4,1	{3: 115, 2: 26, 1: 7, 0: 1}	0.396
AVERAGE		3: 639 2: 79 1: 8 0: 1	1.813	**AVERAGE**		3: 88 2: 19 1: 7 0: 1	0.301

Depth: 5

Minimax				**Alpha-Beta Pruning**			
White Move	**Black Move**	**Nodes Per Depth**	**Time Taken, (seconds)**	**White Move**	**Black Move**	**Nodes Per Depth**	**Time Taken, seconds**
5,2	4,7	{4: 3323, 3: 406, 2: 54, 1: 7,	8.178	5,2	4,7	{4: 197, 3: 64, 2: 13, 1: 7,	0.62

		0: 1}				0: 1}	
5,6	**5,8**	{4: 3924, 3: 457, 2: 59, 1: 7, 0: 1}	9.758	5,6	**5,8**	{4: 200, 3: 67, 2: 16, 1: 7, 0: 1}	0.612
5,4	6,7	{4: 7112, 3: 731, 2: 90, 1: 8, 0: 1}	17.7	5,4	6,7	{4: 646, 3: 144, 2: 35, 1: 8, 0: 1}	1.89
4,7	**5,8**	{4: 5028, 3: 535, 2: 72, 1: 7, 0: 1}	11.852	4,7	**5,8**	{4: 525, 3: 91, 2: 25, 1: 7, 0: 1}	1.348
5,6	6,7	{4: 8291, 3: 799, 2: 93, 1: 8, 0: 1}	20.234	**5,6**	6,7	{4: 639, 3: 155, 2: 36, 1: 8, 0: 1}	1.868
5,8	4,5	{4: 7731, 3: 687, 2: 88, 1: 7, 0: 1}	16.844	**5,8**	4,5	{4: 734, 3: 157, 2: 40, 1: 7, 0: 1}	1.835
AVERAGE		4: 5902 3: 603 2: 76 1: 7 0: 1	14.094	**AVERAGE**		4: 490 3: 113 2: 28 1: 7 0: 1	1.362

Depth: 6

Minimax	**Alpha-Beta Pruning**

White Move	Black Move	Nodes Per Depth	Time Taken, (seconds)	White Move	Black Move	Nodes Per Depth	Time Taken, seconds
5,2	4,7	{5:26776, 4: 3323, 3: 406, 2: 54, 1: 7, 0: 1}	67.586	5,2	4,7	{5: 705, 4: 145, 3: 57, 2: 13, 1: 7, 0: 1}	1.952
5,6	5,8	{5:32269, 4: 3924, 3: 457, 2: 59, 1: 7, 0: 1}	82.787	5,6	5,8	{5: 668, 4: 145, 3: 59, 2: 14, 1: 7, 0: 1}	1.88
5,4	4,5	{5:59447, 4: 7112, 3: 731, 2: 90, 1: 8, 0: 1}	159.302	5,4	4,5	{5: 990, 4: 233, 3: 76, 2: 19, 1: 8, 0: 1}	3.100
4,7	5,6	{5:69206, 4: 8119, 3: 835, 2: 98, 1: 9, 0: 1}	186.365	4,7	5,6	{5: 852, 4: 154, 3: 56, 2: 16, 1: 7, 0: 1}	2.44
6,7	**7,8**	{5:72934, 4: 8506, 3: 872, 2: 101, 1: 9, 0: 1}	192.93	6,7	**7,8**	{5:410, 4: 758, 3: 204, 2: 37, 1: 7, 0: 1}	10.701
6,7	**7,6**	{5:63301, 4: 7368, 3: 743, 2: 90,	164.411	6,7	**7,6**	{5:241, 4: 466, 3: 143, 2: 29,	5.791

		1: 8, 0: 1}				1: 8, 0: 1}	
AVERAGE		5: 53989 4: 6390 3: 674 2: 82 1: 8 0: 1	142.230	**AVERAGE**		5: 644 4: 315 3: 99 2: 21 1: 7 0: 1	4.310

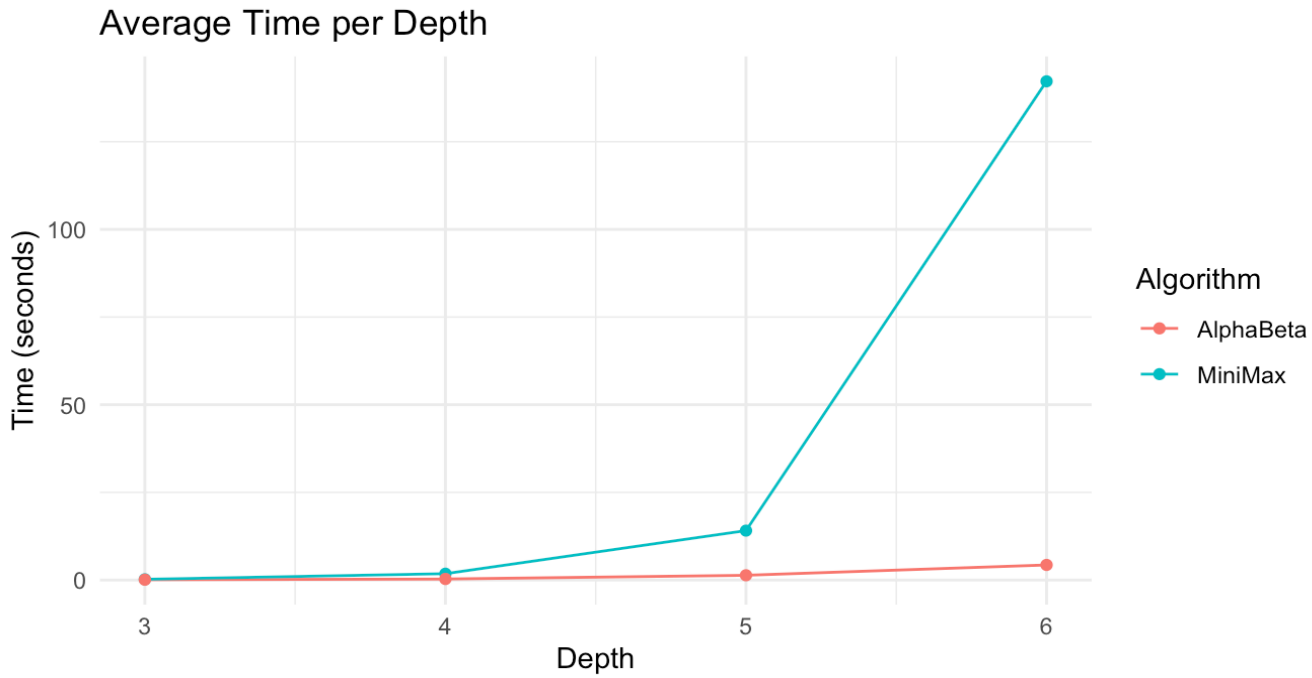

Figure 5 - Average Time per Depth

Figure 5 graph shows the average time taken by two algorithms at different depths of the tree.

The graph shows that both Alpha-Beta and Minimax take longer to reach a decision as the depth of the decision tree increases. Alpha-Beta appears to take consistently less time than Minimax at all depths shown in the graph.

For the explored depths, the time taken by Alpha-Beta is lower than the time taken by Minimax by:

- Depth 3: 2.2 times
- Depth 4: 6.0 times

- Depth 5: 10.3 times
- Depth 6: 33.0 times

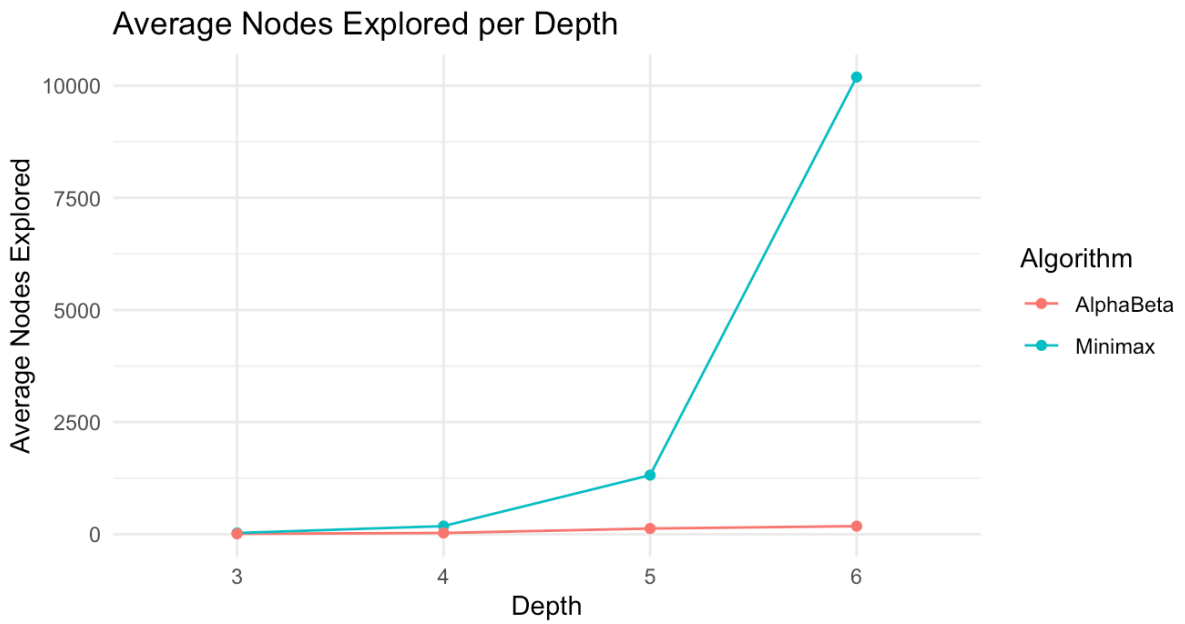

Figure 6 - Average Nodes Explored per Depth

Figure 6 graph shows the average nodes explored by two algorithms at different depths of the tree.

The line for Alpha-Beta is consistently lower than the line for Minimax, which means that the Alpha-Beta algorithm explores fewer nodes on average than the Minimax algorithm at all depths shown in the graph.

For the explored depths, the number of nodes explored by Alpha-Beta is lower than the number of nodes explored by Minimax by:

- Depth 3: 2.6 nodes
- Depth 4: 6.3 nodes
- Depth 5: 10.2 nodes
- Depth 6: 56.3 nodes

Conclusion

Our results indicate that both Minimax and Alpha-Beta Pruning were effective in finding good moves for the checkers game. However, we observed that Alpha-Beta Pruning consistently outperformed Minimax in terms of computational efficiency, exploring fewer nodes while still finding the same optimal move. This efficiency improvement becomes more significant as the search depth increases.

In conclusion, our project demonstrates the effectiveness of both Minimax and Alpha-Beta Pruning in solving the checkers game. While Minimax provides a baseline for understanding the search process, Alpha-Beta Pruning offers a significant optimization that improves computational efficiency, allowing for deeper searches within a reasonable time frame. This optimization is crucial for real-time applications or scenarios where deeper analysis is necessary.