CS312 Lab 7 Report

Soumya Srividhya, 170010038
Anudeep Tubati, 170010039

I. OVERVIEW

This report elaborates on the Minimax algorithm and the Alpha-Beta Pruning algorithm and compares the two in terms of complexity and winning criteria to deduce the better one between them.

II. DESCRIPTION OF ALGORITHMS

A. MINIMAX

In the Minimax algorithm, there is a Max player and a Min player. Max player tries to maximise the score (evaluated based on some heuristic) and Min player tries to minimise the same. Max player plays by recursively exploring a tree of moves which is turn based. First level of the tree consists of the possible moves that can be played by the Max player. After playing these moves (each move being a node at level 1), each node is then played by the Min player, with their own set of possible moves (giving rise to level 2 of min nodes). This continues till a max-depth is reached and then the leaf nodes return the score of their final configuration. As this score flows back to the root, the Min nodes choose the move with the least score out of the possible moves. Similarly, Max nodes choose the move with the highest score out of the possible moves.

This way, when the scores of each move is back-propagated to the root, it chooses the move which gives the highest possible score (with a fixed max-depth) assuming that the opponent is rational.

In a way, this algorithm works by choosing the best out of the worst possible scenarios.

minimax(*board, depth, max_player, current_player*)

if *depth* **is** 0:
    return *eval(board, max_player)*

if *current_player is max_player*:
    maxEval = "-infinity"
    for each *move* **in** *possible_moves*:

```
            eval = minimax(execute(board, move), depth - 1, max_player, other_player)
            maxEval = max(eval, maxEval)
        return maxEval

else:
        minEval = "+infinity"
        for each move in possible_moves:
            eval = minimax(execute(board, move), depth - 1, max_player, other_player)
            minEval = min(eval, minEval)
        return minEval
```

B. ALPHA-BETA PRUNING

Alpha-Beta pruning optimizes the Minimax algorithm by not exploring moves which are certain to give a worse result than what currently exists. This way, it is faster than Minimax.

A node initially gets alpha/beta from the parent node. The nodes pass Alpha/Beta to their children and update Alpha/Beta based on the children's evaluations (alpha = max{alpha, eval(child)} and beta = min{beta, eval(child)}).

Note - Eval refers to an arbitrary heuristic function or the value returned by such a function. They are used interchangeably.

For a Max node, Beta signifies the least *eval* seen by its parent. This parent, being a Min node, will pick at most Beta-valued move. Since Max node updates Alpha after evaluating each of its child, Alpha signifies the highest seen *eval* by it. Max node will pick a move with at least Alpha value. When Alpha is greater than Beta, it means that Max node will definitely be valued more than the pre-existing least *eval* for its parent. Hence it is guaranteed that the parent Min node won't pick this Max node. This allows the algorithm to stop exploring this Max node altogether.

Similarly, for a Min node, the Alpha signifies the *eval* its parent will pick at least. If the updated Beta turns out to be less than the parent Max nodes Alpha value, it means that the Min node will be a value lesser than the minimum/threshold value its parent is going to pick. Hence, this Min node can be discarded from exploring as its parent Max node certainly won't pick it.

alpha_beta_pruning(*board, depth, max_player, current_player, alpha, beta*)

```
if depth is 0:
        return eval(board, max_player)

if current_player is max_player:
        maxEval = "-infinity"
        for each move in possible_moves:
```

```
            eval = minimax(execute(board, move), depth - 1, max_player, other_player, alpha, beta)
            maxEval = max(eval, maxEval)
            alpha = max(eval, alpha)

            if beta <= alpha:
                break
        return maxEval

    else:
        minEval = "+infinity"
        for each move in possible_moves:
            eval = minimax(execute(board, move), depth - 1, max_player, other_player, alpha, beta)
            minEval = min(eval, minEval)
            beta = min(eval, beta)

            if beta <= alpha:
                break
        return minEval
```

III. HEURISTIC FUNCTIONS

A. netPlayerCoins

This heuristic returns (maxPlayerCoins - minPlayerCoins).

Though the main goal of the game is to get the highest number of coins on the board, using this heuristic does not give very encouraging results. This is due to the greediness of the heuristic. It may trade in a lot of mobility (explained in the next subsection), which is crucial in the beginning, for gaining a few more coins.

B. complexHeuristic

This heuristic considers various parameters of the game to reduce the greediness aspect as much as possible and build a strategy to maximise coins in the end.

1. **cornerOccupancy** - Calculates the number of corners captured by the player. Corners are a major part of the game as they cannot be outflanked in any way. This grants immense stability to the player who captures this position.
2. **mobility** - Calculates total valid moves that can be played by the player. This ensures that the player has a good number of moves, as there is a higher probability to find a good move in a higher number of moves.
3. **positionalScore** - A score is attached to every square on the board. For example, corners have a high score whereas the tiles adjacent to corners have extremely high scores. A weighted sum of all coins is calculated with this score. This helps the player filter better moves more efficiently.

4. **pieceDifference** - (maxPlayerCoins - minPlayerCoins). This is required as the final goal of the game is to maximise the Max player's coins.

With all of this, the *total number of coins on the board* is an additional parameter, as the game demands different strategies in the beginning, middle and ending.

During the first 20 turns, positioning and mobility play a crucial role. As mid-game commences (20 - 56 turns), piece difference starts to play a role as well. As the game approaches an end (57 - 64 turns), piece difference is of highest importance and nothing else really matters.

Note that corners are important throughout the game (1 - 64 turns).

C. Comparison of Heuristics

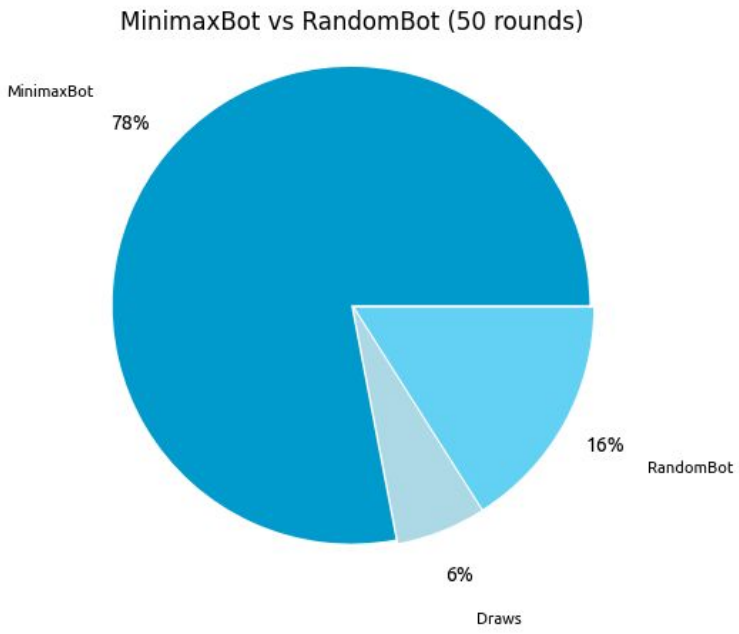

Fig. MinimaxBot vs RandomBot using netPlayerCoins heuristic

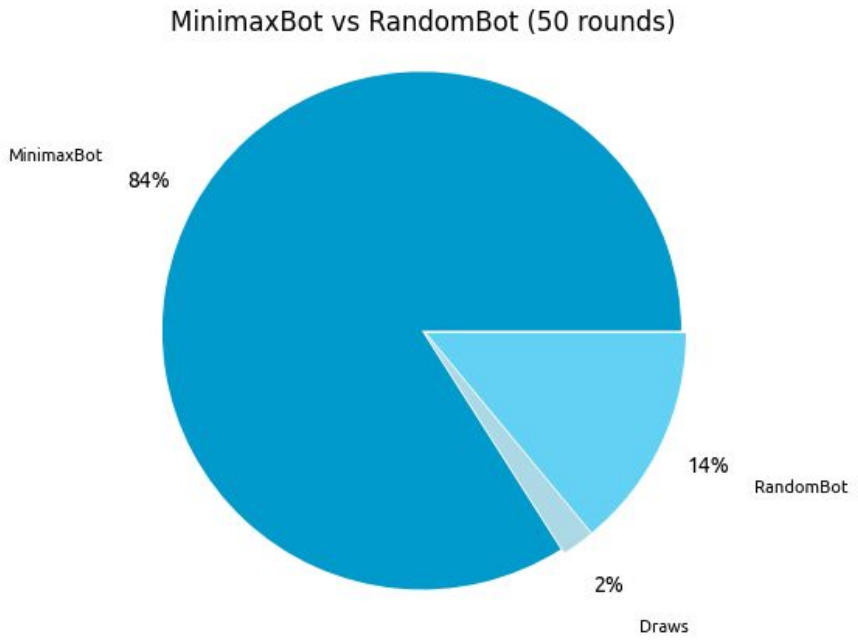

Fig. MinimaxBot vs RandomBot using complexHeuristic

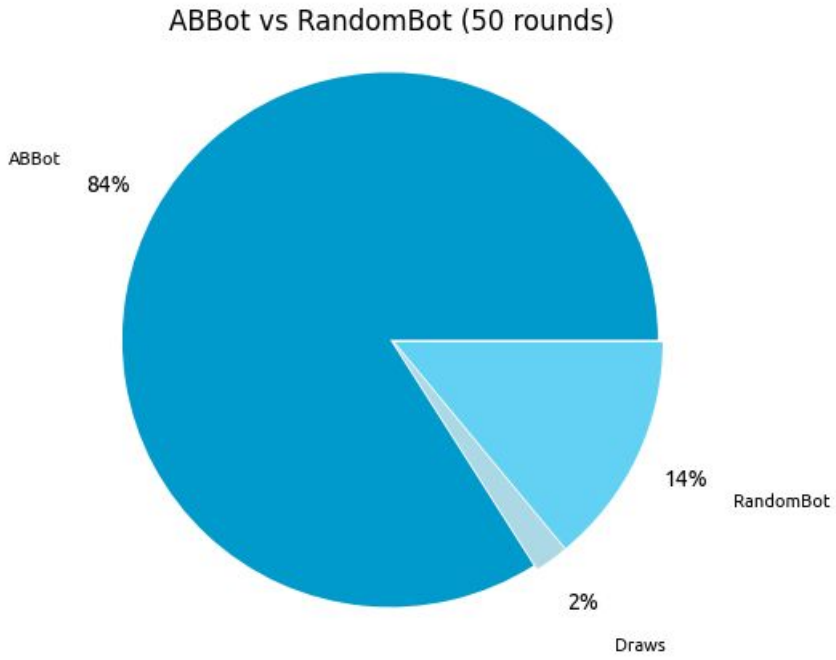

Fig. ABPruningBot vs RandomBot using netPlayerCoins heuristic

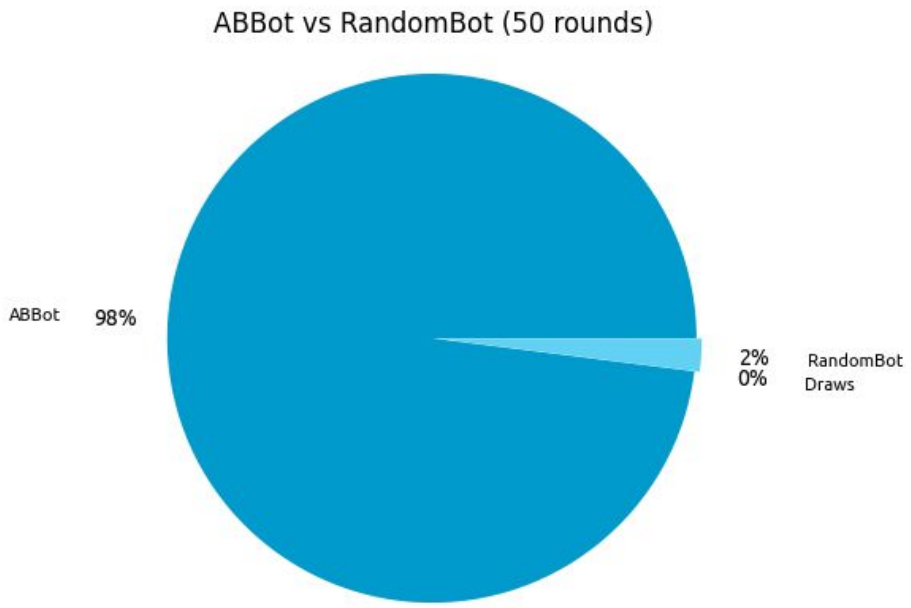

Fig. ABPruningBot vs RandomBot using complexHeuristic

As expected, the *complexHeuristic* performs really well. Almost 10% improvement over the netPlayerCoins heuristic for Minimax algorithm and 20% improvement for ABPruning algorithm.

IV. TREES VISUALISATION

Note - Other than few of the starting moves (which are trivial), all other moves explore more than 150 moves (nodes) on average (with depth just 3). As making a tree with so many nodes is infeasible, we are attaching a link to txt files which contain all the details for 3 moves for each algorithm.

The terminology of text files is the following (to better understand the move logs)

1. Each file represents a move where BLACK is the Max player and it's their chance to play.
2. The file starts with a Root (which is the current config of the board). Root then starts exploring the valid moves which can be made from that particular configuration of the board. Each possible move is a node.
3. Each of root's children have their own possible moves and hence their own child nodes. This goes on till a depth of 3.

4. Each node has a unique ID. When a node is initialised, it prints its depth from root, its ID, its parent's ID. This way, it is easy to understand the txt file in a tree format.
5. For each node, a string is printed at the start of processing its children nodes.
   For eg. ***[START Explore children of MOVE (1, 7) by BLACK]***
6. At the end of exploring each move, the calculated *eval* and *bestEval* are printed.
   For eg. ***[END Explore children of MOVE (1, 7) by BLACK]. Eval = 3 and bestEval = 11***
7. In case of ABPruning, the number of moves that have been pruned is printed.
   For eg. ***[PRUNED] Falling Back to ParentID: 5984 as remaining 8 moves got pruned***

Links to Moves [Heuristic Used - *netPlayerCoins()*]

https://drive.google.com/drive/folders/1agOSLk6JmdeKedpJGS6K3-2FClr8poaF?usp=sharing

V. COMPARISON

A. Space and Time Complexity

Minimax explores all nodes in k-depth. Whereas, ABPruning explores at most all nodes in the same depth (worst case). As explained in Section II, ABPruning does not explore the nodes which are guaranteed to give a worse result than the existing one (or better result, in case of Min node). Hence, the (worst case) time complexity and space complexity of both the algorithms are $O(b^d)$ and $O(bd)$ respectively, where *b* represents the average branching factor for the game and *d* represents the depth allowed to explore.

However, ABPruning usually works faster than Minimax as its constants are smaller compared to those of Minimax. This was verified in the experiment by measuring the average depth explored by each Bot. For a game with a time constraint of 2 seconds for each move, Minimax explored an average depth of **5.3** whereas ABPruning explored an average depth of **6.7**.

B. Winning Criteria

When tested against a random Bot for 50 games, ABPruning Bot achieved better results as compared to the Minimax Bot. This owes to the "pruning" aspect of the ABPruning Bot as discussed earlier. Since it makes the Bot explore more than Minimax Bot in 2 seconds, there's more chance of it coming up with a better move.

When tested against each other, ABPruningBot won over MinimaxBot irrespective of who went first. It is interesting to know that they play the same moves in each game (when playing against each other) as there is no randomness involved in either of them.

As a conclusion, **ABPruning Bot performs better than the Minimax Bot** when individually tested against a Random Bot and when tested against the Minimax Bot. This proves that AlphaBeta Pruning optimizes the Minimax algorithm well, as the winning rate by almost 20% over traditional Minimax.

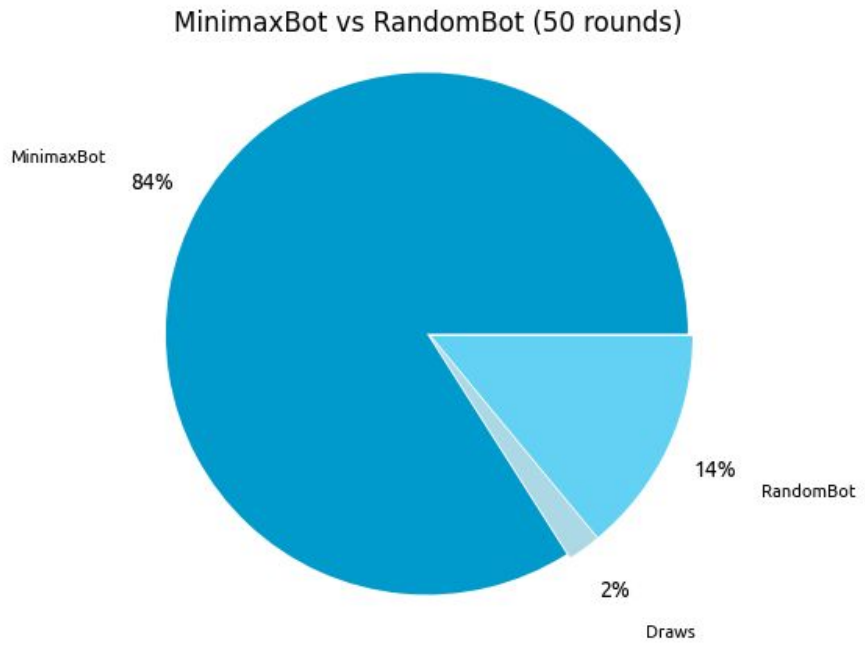

Fig. MinimaxBot vs RandomBot using complexHeuristic

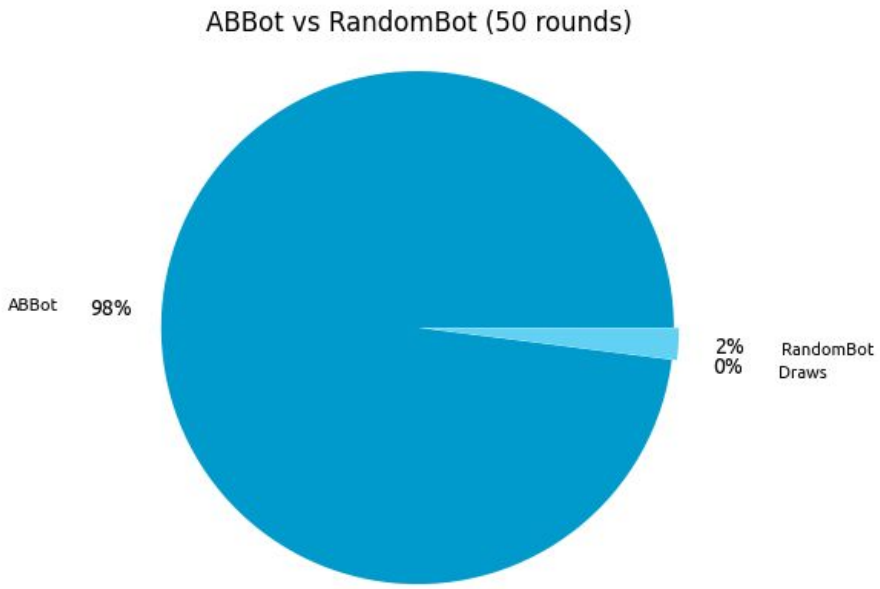

Fig. ABPruningBot vs RandomBot using complexHeuristic