CS312 - Artificial Intelligence Lab

Assignment 5

Arun R Bhat (190010007), Mohammad Sameer (190010024)

1 Introduction

We were assigned to design a bot that plays the game of Othello in an optimal manner. We used ***minmax*** algorithm in order to do so. It's time and space complexity were boosted by using a technique called ***alpha-beta pruning***. So we'll discuss both these algorithms and various evaluation functions used in designing the bot.

2 Algorithms

2.1 MINMAX

Minmax uses the idea from Game Theory of Economics. Players of the game will play in such a way that their chances of winning increase and the opponent's chances of winning decrease. Players take this thought into consideration in each turn. Player 1 is called MAX and player 2 is called MIN. MAX player always picks the node in the given level which has the highest score whereas MIN player always picks the node in the given level which has the lowest score.

Each turn gets the game one level deeper and this continues until the leaf node is reached - that is - players can't make any moves. Hence, we can say that MAX tries to choose the move which is expected to take him towards the victory whereas MIN chooses the move from which MAX benefits the least.

2.2 ALPHA-BETA PRUNING

It is same as Minmax algorithm. It just reduces unnecessary steps by not exploring nodes that don't matter for playing the game optimally. A node's alpha and beta are obtained from it's parent node. Then they are updated and passed back to the parent.

alpha = **max***(alpha, f(child))*
beta = **min***(beta, f(child))*
Here, ***f(child)*** returns the heuristic value of the child node.

Beta is used by the MAX node and alpha is used by the MIN node. MAX's parent is a MIN node and hence it chooses the minimum valued node from it's children. This min-value is updated while traversing the children (which are MAX nodes) as beta. Similarly, MIN's parent is a MAX node and it chooses the maximum valued node from it's children. This max-value is updated while traversing the children (which are MIN nodes) as alpha.

Nodes valued higher than beta are certainly rejected by MIN node and nodes valued lower than alpha are definitely rejected by MAX node. So, those are discarded immediately and are not further explored. This saves huge amount of time and space as compared to applying MINIMAX directly.

3 Pseudocodes

Algorithm 1 Minimax (board, depth, maxPlayer)

```
1: if depth is 0 then
2:     return eval(board, turn)
3: end if
4:
5: if currentPlayer is maxPlayer then
6:     maxEval = "-infinity"
7:     for each move in possible moves do
8:         eval = minimax(execute(board, move), depth - 1, maxPlayer)
9:         maxEval = max(eval, maxEval)
10:    end for
11:    return maxEval
12: else
13:    minEval = "+infinity"
14:    for each move in possible moves do
15:        eval = minimax(execute(board, move), depth - 1, maxPlayer)
16:        minEval = min(eval, minEval)
17:    end for
18:    return minEval
19: end if
```

Algorithm 2 ABPruning (board, depth, maxPlayer, alpha, beta)

```
1: if depth is 0 then
2:     return eval(board, turn)
3: end if
4:
5: if currentPlayer is maxPlayer then
6:     maxEval = "-infinity"
7:     for each move in possible moves do
8:         eval = minimax(execute(board, move), depth - 1, maxPlayer,
9:           alpha, beta)
10:        maxEval = max(eval, maxEval)
11:        alpha = max(eval, alpha)
12:    end for
13:    if beta is not greater than alpha then
14:        break
15:    end if
16:    return maxEval
17: else
18:    minEval = "+infinity"
19:    for each move in possible moves do
20:        eval = minimax(execute(board, move), depth - 1, maxPlayer,
21:          alpha, beta)
22:        minEval = min(eval, minEval)
23:        beta = min(eval, beta)
24:    end for
25:    if beta is not greater than alpha then
26:        break
27:    end if
28:    return minEval
29: end if
```

4 Heuristics

4.1 Composite Eval A

This heuristic takes into account a variety of game elements in order to minimise greediness and design a strategy that maximises coins in the end. It was assumed that this would represent a well designed agent playing the game.

4.2 Composite Eval B

This heuristic was chosen as it was assumed to mimic a human player, who keeps track of mobility and coin difference. These particular terms are explained below. This heuristic does not produce particularly promising outcomes. This is owing to the heuristic's greediness.

5 Parts of Heuristic

5.1 Number of corners occupied

This parameter counts the number of corners captured by the player. Because corners cannot be outmanoeuvred in any way, they are an important component of the game. This gives the player who takes this position a lot of solidity.

5.2 Mobility

This parameter counts the total number of valid moves that can be played at the moment by the player. This is another greedy but logical approach that thinks the probability of finding better move is directly proportional to the number of moves possible.

5.3 Score based on Importance of Positions

Every square on the board is assigned a score to it. Corners, as mentioned above, provide a lot of stability and hence have a higher score. Similarly, it's nearby squares get exceptionally high values. Sum of these scores will give a very good indication of the game's state and hence this parameter is very useful for the player in finding better moves.

5.4 Coin Difference

This parameter returns (no. of coins(MAX) - no. of coins(MIN)). This parameter takes care of the game's final state by widening the difference between MAX and MIN players as much as possible.

5.5 Phase of the game

We also take total no. of coins placed as a parameter. This gives us an idea about the game's phase, i.e., whether the game is in initial, middle or end phase. We use this to take the different weighted average of other parameters at different phases. Mobility and Positional score are very important in the initial phase. Coin difference becomes more and more crucial as the game goes on.

6 Complexity

Minmax does a k-depth exploration of all nodes. ABPruning, on the other hand, investigates all nodes at the same depth (worst case). As stated earlier, ABPruning does not investigate nodes that are certain to provide a worse (or better, in the case of the Min node) outcome than the present one. As a result, the (worst case) time and space complexity of both methods are $O(b^d)$ and $O(bd)$, respectively, where b represents the game's average branching factor and d indicates the depth permitted to explore. ABPruning, on the other hand, is generally quicker than Minmax since its constants are smaller.

7 Results and Conclusion

7.1 Stats

We played 30 games in order to test bots. In the figures below, Black stands for our player (uses MiniMax and Alpha Beta Pruning bots) and Red stands for opponent (Random bot).

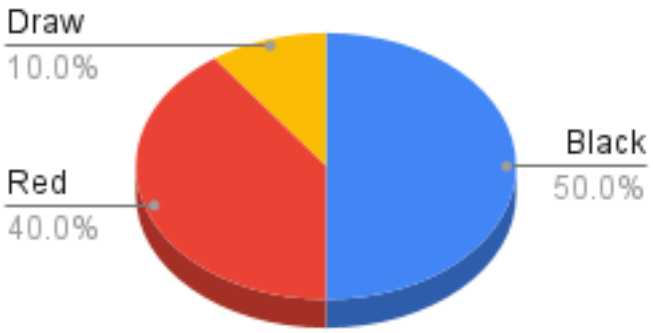

Figure 1: MiniMax bot using Composite Eval B heuristic (Black) vs Random bot (Red)

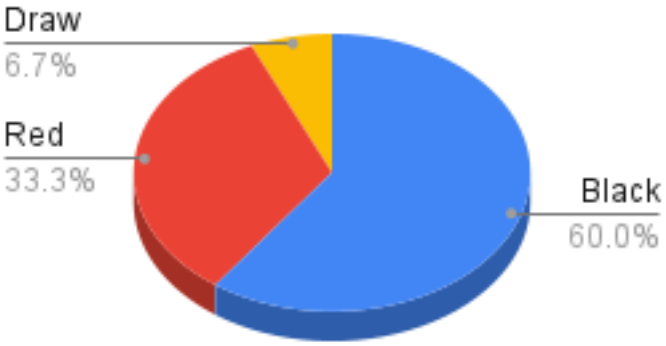

Figure 2: MiniMax bot using Composite Eval A heuristic (Black) vs Random bot (Red)

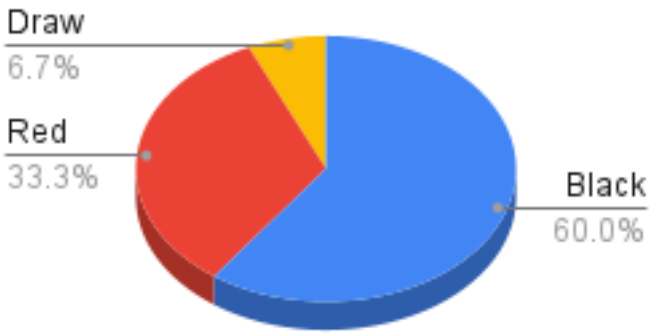

Figure 3: ABPruninig bot using Composite Eval B heuristic (Black) vs Random bot (Red)

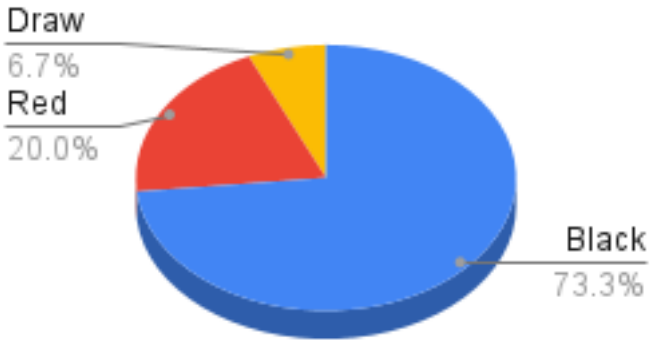

Figure 4: ABPruning bot using Composite Eval A heuristic (Black) vs Random bot (Red)

7.2 Inference

Though Minmax-bot performs much better than Random-bot, using alpha-beta pruning on top of it improves the results further. Similarly, using composite eval A lead to larger Space and Time requirements. So, composite eval A increases minimax-bot's performances only a little in comparison to composite eval B. When played against MinmaxBot, ABPruningBot came out on top, regardless of who ran first. It's noteworthy to note that they make the identical actions in each game (when playing against each other), as their steps aren't random.