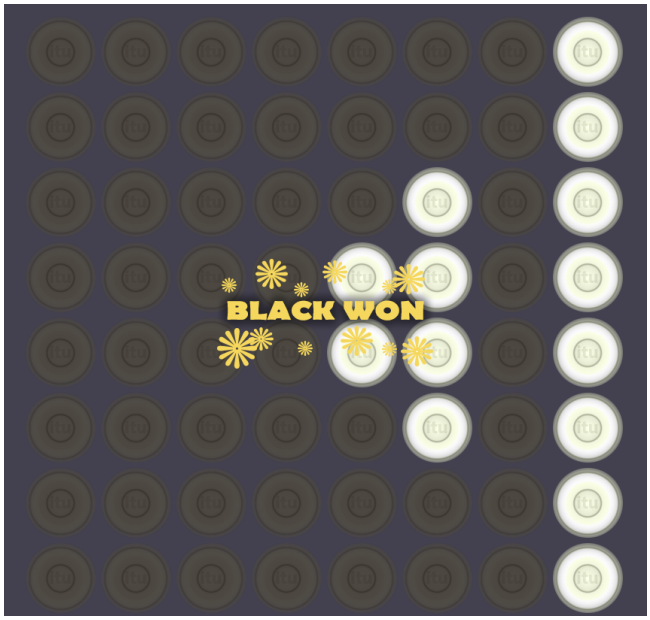
AI Othello Project 1



Group 01

Sverri Sølbæk Trond Himmelstrup Malte Helin Johnsen
Lasse Overgaard Simon Olafsson

{SVHA,MHEJ,LOVE,SIMOL}@itu.dk

A game project using a weighted Alpha Beta Minimax search algorithm to create an AI opponent in Othello
For ITU course "Introduction To Artificial Intelligence"

Software Design
IT University of Copenhagen
Denmark
22/03/2023

IT UNIVERSITY OF CPH

1 MiniMax Search

Before assessing different viable strategies in creating an Othello AI, we started out by implementing the basic MiniMax algorithm. The method that starts out the MiniMax search, is the decideMove() function given by the OthelloAI-interface. In our implementation, the decideMove() function starts the MiniMax search, given that there is at least one legal move to be made:

```
1    //From decideMove implementation in MiniMax implementation
2    ...
3        if (!moves.isEmpty()){
4            Node move = maxValue(s);
5            return move.getPosition();}
6        else
7          return new Position(-1,-1);
```

Whenever a certain GameState is given to the maximizing player, our AI will play as MAX, choosing the move associated with the highest utility (number of captured coins) of all legal moves in that current state, assuming that the adversary, represented by MIN, will choose the lowest possible utility for MAX in the subsequent ply.

The second necessary element of our AI implementation that we identified, was a Node-class. This is implemented to associate a utility-value (an integer) with a Position-object ensuring that utility and position can be stored as a tuple:

```
1      public Node(Integer utility, Position position) {
2        this.utility = utility;
3        this.position = position;
4    }
```

The heart of the basic implementation of the MiniMax algorithm, is the two methods max-Value() and minValue(), that recursively call each other.

The recursive steps taken in the algorithm happens when traversing down the game tree through the possible moves or actions for MAX (maximizing player) and MIN(minimizing player), each subsequent method call representing a ply of the game. The algorithm searches every possible game state reachable from the given state:

```
1    //From MaxValue implementation in MiniMax
2        ...
3        if(TerminalState){
4            return new Node(utility(s), new Position(-1, -1));}
5        ...
6        int v = Integer.MIN_VALUE;
7        for (Position position : moves) {
```

```
            ...
            GameState s2 = new GameState(s.getBoard(),s.getPlayerInTurn());
            move = position
            ...
            Node p2 = minValue(s2);
            if(p2.getUtility() > v){
                v = p2.getUtility();
            }
        }
        return new Node(v, move);
```

A call to minValue() will have the MIN player choose the move associated with the lowest utility. It does this by calling maxValue() on each available position:

```
    //From MinValue implementation in MiniMax
        ...
        if(TerminalState){
            return new Node(utility(s), new Position(-1, -1));}
        ...
        int v = Integer.MAX_VALUE;
        for (Position position : moves) {
            ...
            GameState s2 = new GameState(s.getBoard(),s.getPlayerInTurn());
            move = position
            ...
            Node p2 = maxValue(s2);
            if(p2.getUtility() < v){
                v = p2.getUtility();

            }
        }
        return new Node(v, move);
```

Every time a terminal state is reached, the algorithm uses a utility-function. Our utility function takes a particular GameState as an argument and returns an integer, that signifies the number of captured coins for MAX in that gamestate:

```
    private Integer utility(GameState s) {
            int[] utility = s.countTokens();
            return utility[playerToken == 1 ? 0 : 1];
        }
```

Since the algorithm searches all the way to the bottom of the search tree, we were only able to run the algorithm for a board size of 4 x 4 tiles while achieving acceptable running times. Therefore, some optimizations had to be made to increase performance.

2 Heuristic Alpha Beta Search

The branching factor will not be uniform (i.e. there will be a different number of child nodes / number of possible positions) as the search algorithm traverses down the tree. The time complexity of the MiniMax algorithm is $O(b^m)$, where b is the number of available positions on the board in each state and m is the max depth of the game tree. As the number of possible moves increases exponentially with increasing plies, the time for the AI to calculate each move is becomes inefficient.

This is where a modification of the MiniMax search called Alpha-Beta pruning was be considered as an alternative solution for the time complexity problem of MiniMax. In the best case, the time complexity of Alpha-Beta pruning is $O(b^{m/2})$ and $O(b^m)$ at worst, where the latter is the time complexity from MiniMax. This means that large parts of the search tree that do not affect the final result are pruned which affects the search time in the game tree significantly.

The main difference from the former implementation of the MiniMax algorithm is the use of these Alpha and Beta values. These values are designed to enable MAX and MIN to keep track of their individually best value along the tree. Alpha is the best choice for MAX, since it's the highest value, and Beta is the best choice for MIN, since it's the lowest value. In our implementation, both the MaxValue and MinValue methods take Alpha and Beta (integers) as parameters, to dynamically change the values traversing the game tree. The properties of our pruning implementation:

MaxValue method: If the alpha value is equal or higher than the beta value → alpha cut appears.

```
1    //From MaxValue method in AlphaBeta implementation
2
3         for (Position position : moves) {
4            GameState s2 = new GameState(s.getBoard(),s.getPlayerInTurn());
5            s2.insertToken(position);
6            Node p2 = minValue(s2, alpha, beta, depth+1);
7            if(p2.getUtility() > v){
8                v = p2.getUtility();
9                move = position;
10           }
11           if(v >= beta){return new Node(v, move);}
12           if(v > alpha){alpha = v;}
13
14       }
15       return new Node(v, move);
```

MinValue method: if the beta value is equal or lower than the alpha value → beta cut appears.

```
1    //From MinValue method in AlphaBeta implementation
2
3        for (Position position : moves) {
4            GameState s2 = new GameState(s.getBoard(),s.getPlayerInTurn());
5            s2.insertToken(position);
6            Node p2 = maxValue(s2, alpha, beta, depth+1);
7            if(p2.getUtility() < v){
8                v = p2.getUtility();
9                move = position;
10           }
11           if(v <= alpha){return new Node(v, move);}
12           if(v < beta){beta = v;}
13       }
14       return new Node(v, move);
```

To make the AI able to run within a realistic time-frame, we implemented a function to cut off our seach early. Our cutOff method is implemented implicitly throughout the minValue and maxValue functions. These methods are changed to include a depth parameter, which is incremented in each method call. In the beginning of each method call, the depth parameter is then evaluated against a maxDepth value, which is assigned in the field of the AI. For our implementation and running on our machines we ran the algorithm with a depth of 8, for a balance between computing time and accuracy, but this can be altered for the specific machine running the algorithm.

This condition essentially replaces our previous isTerminal method in our basic implementation. The condition for which to cut off is therefore an OR boolean evaluation of either the state being terminal or the max depth for the search being reached:

```
1    private Node maxValue(GameState s, int alpha, int beta, int depth) {
2    ...
3    if(moves.size() == 0 || depth > maxDepth){
4            return new Node(CoinParityUtility(s), new Position(-1, -1));}
```

For obvious reasons, when we reach a max depth and have to return a utility value associated with a move, we need to return a value that indicates how likely we are to win when the game continues from the cut-off depth. For this, we need an evaluation-function.

In exploring different alternatives for an evaluation function, we consulted a paper that delved into the implementation of different heuristics for an AI playing Othello (Sannidhanam Annamalai, s.d.: 5). Overall, four different heuristics were found to have an impact on the gameplay. First and foremost was the coin parity (ibid.) that provides an insight into how the coins on the board are balanced between the maximum and the minimum player.

To optimize our AI to evaluate the desirability of different positions on the board in different game states, we changed our initial utility function to evaluate coin parity instead for each player in a certain gamestate.

```
1  private int CoinParityUtility(int[] utility){
2          int max = utility[playerToken == 1 ? 0 : 1];
3          int min = utility[playerToken == 1 ? 1 : 0];
4
5          int diff = 100*(max-min);
6          // to avoid zero division and negative division
7          int total = (max+min)<1 ? 1 : (max+min);
8          return diff/total;
9      }
```

When creating an additional viable evaluation function to further utilize our evaluation function implementation, we made a method that returns an integer with an estimate of desirability of that position by calculating a defined board with static weights given to each board position. This method calls the coin parity function and returns both the coin parity evaluation with added static weights based on a certain game state.

```

    private int eval(GameState s) {
        int[][] board = s.getBoard();
        int size = board[0].length;
        int palyer1Heuristic = 0;
        int palyer2Heuristic = 0;
        for (int y = 0; y < size; y++){
            for (int x = 0; x < size; x++){
                if ( board[y][x] == 1 ){
                    palyer1Heuristic += getHeuristic(x,y);
                }
                else if ( board[y][x] == 2 ){
                    palyer2Heuristic += getHeuristic(x,y);
                }
            }
        }
        int[] utility = {palyer1Heuristic, palyer2Heuristic};
        return CoinParityUtility(utility);
    }
```

```
    //Field with static weights from our AlphaBeta implementaion

    private int[][] staticWeights =  {  {4, -3, 2, 2, 2, 2, -3, 4},
                                        {-3, -3, -1, -1, -1, -1, -4, -3},
                                        {2, -1, 1, 0, 0, 1, -1, 2},
                                        {2, -1, 0, 1, 1, 0, -1, 2},
                                        {2, -1, 0, 1, 1, 0, -1, 2},
                                        {2, -1, 1, 0, 0, 1, -1, 2},
                                        {-3, -4, -1, -1, -1, -1, -4, -3},
                                        {4, -3, 2, 2, 2, 2, -3, 4}
                                        };


    //Method to retrieve board weights
   private int getHeuristic(int x, int y){
        return staticWeights[y][x];
    }
```

The second heuristic evaluated in the paper was the mobility of the players, namely maximizing one's own mobility all the while minimizing the mobility of the opponent (ibid.). This heuristic is split into actual and potential mobility. Potential mobility consists of the potential moves that any move might lead to whereas the actual mobility consists of the legal moves in a given game state.

The third evaluated heuristic was corners captured (ibid.: 6). In Othello, the corners themselves have the greatest value while the squares directly adjacent to the corners are generally not favored; this is due to the number of squares/coins that the opponent can capture

when capturing a corner. This was a heuristic that we sought to implement, as it proved the most useful. In the paper, a dynamic approach was taken, but in our implementation, we weighted the board statically so as to opt our AI to always seek the squares that were represented by the highest value.

The fourth and last heuristic that was evaluated was that of stability (ibid.: 6). Stable coins cannot be flanked and as such can be evaluated as safer as opposed to unstable coins that suffer from moves being able to flank - and capture - them.

We tried to utilize a corner heuristic to make a more strict strategic move pattern for our AI class. When comparing each version / implementation of the different heuristics in a head to head challenge, there was a clear difference in win ratio when using a combination of calculated coin parity and static weights of position.

References

[1] Sannidhanam, V., Annamalai, M., (s.d.), "An Analysis of Heuristics in Ohtello", Department of Computer Science and Engineering, University of Washington