Project 1 - Othello

alja antni asmt

March 24, 2023

1 MiniMax

MiniMax is a decision-making algorithm used to determine the best possible move for a player in a two-player game. It assumes that the opponent will always make the move that is most harmful to the player, and therefore, the player must choose the move that minimizes the maximum possible loss. The algorithm works by constructing a game tree representing all possible moves and outcomes of the game and assigning a score to each node. The scores are then propagated up the tree, with the player choosing the move with the highest score at the root of the tree. In our implementation we initially took the alpha beta pruning MiniMax pseudo code from the slides and translated it to Java. This initially only had a single turn look ahead. Which closesly resembles a greedy search. Running our naive MiniMax against DumAI, resulted in a win ratio of $\sim 65\%$, adjusting it to look four moves ahead gives $\sim 85\%$ win rate. This is okay, but we can do better! Let's come up with some efficient heuristics to guide our AI. If we study the game of Othello, we come across some interesting trends.

The aim is to have as many of your pieces on the board as possible. When neither player can make a move, the player with the most pieces wins the game. Despite this, we can observe that having many pieces on the board early in the game is not necessarily a good thing. Any new piece needs to be placed on tiles bordering the opponents pieces. This actually means having fewer pieces on the board initially can make it significantly harder for your opponent. This also works well when applying our second strategy. Specific tile placement! Not all tiles on the board are created equal, and looking at the last few moves of a game, reveals that having the corners of the board constitutes a particularly attractive advantage. With all these possible heuristics in mind, it is important to note that our MiniMax algorithm should be prioritized when we can predict a winning branch in our tree. Meaning we should give many points for either an early termination or close to the end of the game with fewer remaining moves.

Below is our final implementation which utilizes the heuristics mentioned above.

```
public Position decideMove(GameState s){
    if (s.legalMoves().isEmpty())
        return new Position(-1,-1);
    return ABSearch(clone(s));

}

private int cme;
private Position ABSearch(GameState s){
    int size = s.getBoard().length;
    int me = s.getPlayerInTurn();
    if (pv == null || cme != me || pv[0].length != size ){
        cme = me;
        int[][] pv_plus = generatePosValue(size);
        int[][] pv_minus = new int[size][size];
        int[][] zeroes = new int[size][size];
        for (int i = 0; i<size; i++){
            for (int j = 0; j<size; j++){
                pv_minus[i][j] = -pv_plus[i][j];
            }
        }
        pv = new int[3][][];
        pv[0] = zeroes;
        pv[me] = pv_plus;
        pv[3-me] = pv_minus;
    }
    return firstMaxValue(s, Integer.MIN_VALUE, Integer.MAX_VALUE, 6, me);
}

private Position firstMaxValue(GameState s, int alpha, int beta, int count, int me){
    boolean fin = s.isFinished();
    if (fin || count <= 0)
        return null;
    int v = Integer.MIN_VALUE;
    Position move = null;
    var moves = s.legalMoves();
    if (moves.isEmpty())
        moves.add(new Position(-1, -1));
    int[] vs = new int[moves.size()];
    IntStream.range(0, moves.size()).parallel().forEach(i->{
        Position a = moves.get(i);
        GameState sPrime = clone(s);
        sPrime.insertToken(a);
        vs[i] = minValue(sPrime,alpha,beta, count-1, me);
```

```
        });
        for(int i = 0; i < vs.length; i++){
            if (vs[i]>v){
                v = vs[i];
                move = moves.get(i);
            }
        }
        return move;
    }
    private int maxValue(GameState s, int alpha, int beta, int count, int me){
        boolean fin = s.isFinished();
        if (fin || count <= 0)
            return utility(s,me, fin);
        int v = Integer.MIN_VALUE;
        var moves = s.legalMoves();
        if (moves.isEmpty())
            moves.add(new Position(-1, -1));
        for(Position a : moves){
            GameState sPrime = clone(s);
            sPrime.insertToken(a);
            int v2 = minValue(sPrime,alpha,beta, count-1, me);
            if (v2>v){
               v = v2;
               alpha = Math.max(alpha, v);
            }
            if (v >= beta)
                return v;
        }
        return v;
    }

    private int minValue(GameState s, int alpha, int beta, int count, int me){
        boolean fin = s.isFinished();
        if (fin || count <= 0)
            return utility(s,me, fin);
        int v = Integer.MAX_VALUE;
        var moves = s.legalMoves();
        if (moves.isEmpty())
            moves.add(new Position(-1, -1));
        for(Position a : moves){
            GameState sPrime = clone(s);
            sPrime.insertToken(a);
            int v2 = maxValue(sPrime,alpha,beta, count-1, me);
            if (v2<v){
               v = v2;
               beta = Math.min(beta, v);
```

```
        }
        if (v <= alpha)
            return v;
    }
    return v;
```

Our alpha-beta search function ABSearch initializes the gameboard weights for each player before it hands over the execution to the firstMaxValue function. The firstMaxValue function is almost identical to maxValue, but was created to enable the algorithm to make use of multi-threading. It is only called once before handing over execution to the maxValue and minValue functions, in order to avoid creating new threads in the recursive loop. The gameboard weights are positive for the max player and negative for the min player, since they target the two extremes respectively.

The algorithm starts at the firstMaxValue function, which tries to maximize the points the player can get, while also calling minValue to avoid considering plays that the opponent will not let go through. Both functions start off by checking whether we have reached our cut-off depth or if the game is finished, so that we can evaluate the game tree branch with our utility function before attempting to do otherwise useless work. The functions look for legal moves that the player can perform, but if there are no legal moves, position (-1, -1) is added instead to signal that the player passes their turn. If we have some legal moves, the functions will loop through them and insert a token at the given position on a copy of the gameboard, before handing it over to the minValue or maxValue function depending on the caller. The return value is then compared to previous moves before deciding whether it will replace the current alpha value for maxValue or the beta value for minValue. These values are used for the early termination at the end of the for loop. This ensures that we give up on branches in the game tree, that we will never be allowed to perform by the opponent.

1.1 Evaluation Function

The current iteration of our utility/evaluation function looks like this:

```
private int getPostitionValues(GameState s, int me){
    int[][] board = s.getBoard();
    int res = 0;
    int size = board.length;
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            res += pv[board[i][j]][i][j];
        }
    }
    return res;
```

```
private int utility(GameState s, int me, boolean fin){
    int[] counts = s.countTokens();
    int diff = counts[me - 1] - counts[2-me];
    int placedTileCount = counts[0]+counts[1];
    if (fin) {
        if (diff > 0) return 1000 - placedTileCount;
        if (diff < 0) return -1000 + placedTileCount;
        return 0;
    }
    return -diff + getPostitionValues(s,me);
}
```

it consists of a couple of steps:

1. Calculate the difference in the current players tokens and the other players tokens.
2. If we have reached an end state we do not care by how much we win or lose and instead return a large fixed value moderated by the number of tiles placed to incentivize quick wins over longer games with more possible points.
3. Return the inverted difference in tokens plus the weighted sum of tokens on the board. Each position on the board has a weigth associated with it. Many points at the edges to incentivize taking corners and negative points for tiles adjecent to the edges to desentivise making it easy for the opponent to get the corner. Coupling this with having few tokens on the board to further restrict the opponents possible moves and avoid getting strategically important squares.

This gives us a very high win rate while having an average time to move of under a second spiking far into the game (average time is the worst on move 29 at around .8 seconds) to at most 4-5 seconds (8x8 board, search depth of 6, Random AI as opponent).

1.2 Cut-off function

We do not have a dedicated cut-off function, instead we simply pass the depth we want to explore as an argument to the recursive function, which decreases by one for every iteration until it reaches zero at which it will return the function call. The argument is present in the maxValue and minValue functions and is decreased when the functions call eachother.

```
private int minValue(GameState s, int alpha, int beta, int count, int me);
```