Othello AI
Introduction to Artificial Intelligence - Project 1
Group 12 (Boolean Hooligans)

Simon Titkó
siti@itu.dk

Tobias Lysdal Hansen
tolh@itu.dk

Julian Kristoffer Valbjerg Pedersen
jkrp@itu.dk

March 22, 2024

1 Introduction

This document reports on the Othello AI project that we developed during the Introduction to Artificial Intelligence course at the IT University of Copenhagen. The source code of our is available in the following GitHub repository:
https://github.com/simontitk/Othello-AI.git

2 Evaluation and cut-off

2.1 Evaluation

Our approach to evaluating a game-state is based on our experience we gained by playing a handful of games of Othello. We could observe the following rule-of-thumbs:

1. A corner position is the most valuable one, since your opponent cannot overtake a piece here.
2. Edge positions are the second most valuable, since your opponent can only overtake these if they also have a piece along the same edge, or cannot at all, if the edge pieces continuously build off of a corner piece.
3. "Inner edge" positions (one row/column towards the center of the board from the edge) are to be avoided, since your opponent overtaking a piece here guarantees them an edge position.

We represent these with a 2D array (programatically generated depending on the board size), containing values 1000000, 100, 1, and 10, inserted into corners, edges, inner edges and "default" (anything not falling into the above) positions, respectively. We chose these numbers so that there is a sufficient proportional difference between a single edge piece and several inner pieces, as well as to ensure that going for a corner position outweighs all other options.

After the value array is generated, a given game-state can be evaluated by extracting its board as a 2D array, which contains 1's (representing pieces of player 1), 2's (representing pieces of player 2) and 0's (representing empty positions). To determine one single numeric value, the utility of the game-state, the following steps are performed:

- With two for-loops, every two corresponding positions in the value array and the board array are accessed,
- the two numbers are multiplied, with 2's in the board array replaced with -1,
- the product is added to a running sum.

This way, a single integer is determined, with high positive values representing a state favorable to player 1, and large negative numbers for player 2. While the calculation of the sum takes quadratic time, it should be sufficiently fast for not extraordinarily large board-sizes.
The functionality described above is implemented in the class `BoardEvaluator`. The class has two public methods, `generateValues(int size)` for generating the value array and storing it in a class field (so it is not recalculated for further evaluations) and `evaluate(Gamestate s)`, which returns a single integer representing the value of a board state.

2.2 Cut-off

To prevent excessive computation times, we cut off our search after a certain depth in the game tree. These depths were empirically determined for different board sizes, so that the running time of the algorithm is on the scale of a few seconds at most. During our testing against `DumAI`, we used `System.currentTimeMillis()` before and after the search method, and logged the difference of these into `.csv` files, to get a rough estimate of the search's running time.

The results for different depths on an 8x8 board are shown in Fig 1. It can be seen that with a depth of 10, the computation is too slow even during the first few moves of the game (and timed out later on), so we settled on a depth of 8 in this case. A similar procedure was carried out for the other board sizes, too. It must be noted that the numbers here are not representative, as they heavily depend on the environment where the program is run, as well as the actual game-state and available moves.

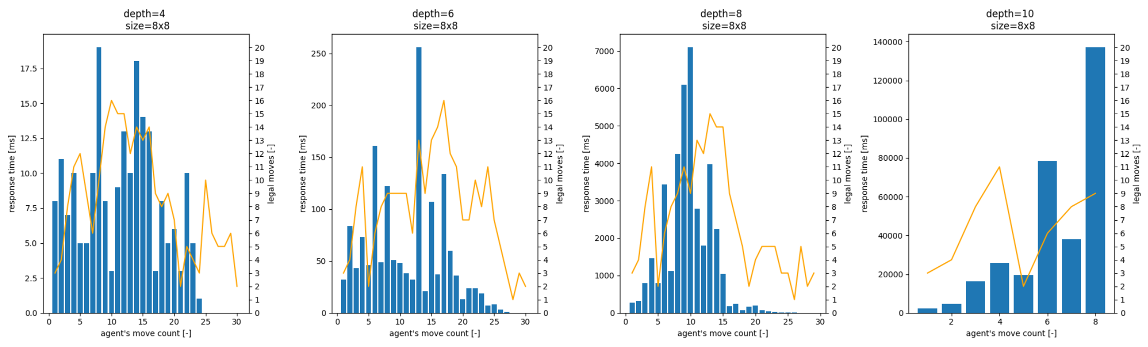

Figure 1: Run time of search method and number of available moves for different depths

3 Search algorithm

We implemented the `minimax` search algorithm with alpha-beta pruning to efficiently find the optimal move for our agent, represented by the `HooliganAI class`. The entry point for the search is the `decideMove(Gamestate s)` method. Here, the initial values for alpha, beta, and the current depth are initialized as `Integer.MIN_VALUE` ($-\infty$), `Integer.MAX_VALUE` ($+\infty$), and 0. The max depth is also dynamically determined here and stored in a class field, based on the size of the board (by calling the `getMaxDepth(int boardSize)` private method). Then, the search is performed by subsequent calls to the `MaxValue` and `MinValue` functions.
In these, if the search has reached its maximal depth, or if we have reached a terminal move, the game-state is evaluated by the `BoardEvaluator` class and the function call exits with an early return.
Otherwise, the agent iterates through the list of available moves, where a new, hypothetical game-state resulting from each is created. This, alongside with the alpha-beta values and the current depth is then passed to the opposing `MinValue/MaxValue` function, which, after a number of recursive calls, eventually returns a (move, value) tuple. Throughout the iterations, the best value and the corresponding move is kept track of, alongside with the alpha (Max's highest) and beta (Min's lowest) values. These are used for the pruning, because if the opposing function returns a value which is worse for us than what's currently stored in alpha/beta, the function exits by returning the currently investigated move and the corresponding value, and that sub-tree is not checked any further.

After resolving every call to `MaxValue` and `MinValue`, the final return from the `decideMove` method is the `Position` chosen by the agent. The absence of available moves is represented by the position (-1, -1).

The (move, value) 2-tuple used throughout the function calls is represented by the private class `MoveValue`, which simply includes a `Position` object and an integer as its fields.

We introduced another way of pruning by checking the list of legal moves in a given gamestate. If it contains any moves resulting in a corner piece, all other moves are ignored, and only the subtrees stemming from these are evaluated. This not only reduces the computation time, but forces our agent to pick the best possible move. This functionality is achieved by the `getMovesToCheck(GameState s)` and `isCorner(Position position, int boardSize)` private methods.

4 Conclusion

To have a more representative testing process, we implemented a modified version of `DumAI`, called `RandomAI`, that randomly picks one of the available moves. Our agent has been able to consistently beat it on different board sizes, with response times on the scale of a few seconds. This asserts that we picked an appropriate evaluation heuristic and implemented an efficient pruning strategy.