IT UNIVERSITY IN COPENHAGEN

INTRODUCTION TO ARTIFICIAL INTELLIGENCE 2023

Othello

Nikolaj Sørensen (nikso@itu.dk) Daniel Kjellberg (dakj@itu.dk)
Jeppe Lindhard (jepli@itu.dk) Johan Flensmark (jokf@itu.dk)

Course Code: KSINARI1KU
Course Manager: Rune Møller Jensen

24. March 2023

1 Introduction

This short paper is made to accommodate the implementation that is made for IAI Project 1 - Othello. The goal of this paper is to briefly describe the solution and argue why it works. The entire source code can be found on Github at github.com/nikso-itu/IAI.

2 Implementation details

Our implementation of the Minimax algorithm can be seen in `Minimax.java`. This implementation is based on the Minimax implementation from Russell [2010] in figure 5.3, and it follows the recurrence relation also included in the lecture slides, see equation 1 for the recurrence relation for the Minimax search algorithm. The algorithm searches through the game tree to find the best move for the current player, assuming that the other player will also make optimal moves.

H-MINIMAX(s,d) =

$$\begin{cases} EVAL(s, MAX), & \text{if Is-CUTOFF}(s,d). \\ max_{a \in ACTIONS(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1), & \text{if TO-MOVE}(s) = MAX. \\ min_{a \in ACTIONS(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1), & \text{if TO-MOVE}(s) = MIN. \end{cases} \quad (1)$$

The algorithm consists of two main methods, `maxValue` and `minValue`, which alternate between trying to maximize and minimize the utility function. The max-value method tries to find the move that leads to the highest utility score for the current player, while the min-value method tries to find the move that leads to the lowest utility score for the other player.

The implementation depends on the `GameState` object, which contains the current state of the game board and the player whose turn it is. The algorithm uses the `legalMoves` method of the `GameState` object to get a list of all legal moves that can be made from the current state.

The algorithm uses alpha-beta pruning, based on the implementation in figure 5.7 from Russell [2010]. Alpha-beta pruning is a technique that eliminates branches of the game tree that are guaranteed to lead to a worse outcome than the best outcome found so far, which should speed up the overall search time. Our implementation of alpha-beta pruning required only a minor modification to the plain Minimax search algorithm. We maintain two values, alpha and beta, which represent the best score found so far for MAX and MIN, respectively. During execution, the algorithm compares the current utility of a recursion step to alpha or beta (depending on whose turn it is) to eliminate any nodes that are guaranteed to lead to a worse outcome, which are thereby meaningless to explore.

The algorithm uses a depth-limited search, which means that it only considers a fixed number of moves ahead of the current state. To do this the implementation uses two helper meth-

ods, `isCutOff` and `isTerminal`, to determine when the search should be terminated. `isCutOff` checks whether the search has reached a maximum depth, while `isTerminal` checks whether the current state is a final state (i.e., whether the game is over). We use a cutoff to reduce the computation time as the branching factor for this game is around 10, as stated in Norvig [1992]. The use of a cutoff is also why we need an evaluation/utility function that can capture the desirability of an intermediate state of the game and not just the terminal states.

The algorithm also requires an evaluation/utility function which assigns a score to each possible game state. The evaluation function used in this implementation is a simple heuristic that assigns a score to each position on the game board based on its desirability for the current player. The score is higher for positions such as corners and edges of the board, and lower for positions that could potentially benefit the other player, such as the positions next to the corners. These positions are called the danger zones and placing a token on in this zone can act as bridges to the desired locations. The exact weight allocations for a 8x8 board size are shown in figure 1. For other board sizes, we have made a dynamic weight allocation that tries to replicate this pattern. Our heuristic is inspired by the approach of Sannidhanam and Annamalai [2015]. The advantage of using this approach, is that we get a simple heuristic that captures the general/average importance of each position. However, its weakness is that it is static and cannot reflect a position's desirability as a function of the game state, which could vary a lot.

4	-3	2	2	2	2	-3	4
-3	-4	-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

Table 1: Static weights assigned to individual board positions that reflect their desirability. Weight allocations are taken from Sannidhanam and Annamalai [2015].

3 Performance Evaluation

Our AI can be tweaked by changing the depth that the Minimax search goes to, before it cuts off. When the depth value is increased, more moves will be considered which in turn increases the difficulty of the AI but also drastically increases the search time. To see the time it took to run our algorithm using different depth levels, we performed some experiments, from which the results can be seen in table 2.

Depth level	Average time in milliseconds
8	219.96
9	256.51
10	3327.74
11	27660.48

Table 2: Average runtime of our algorithm calculated over a full game, using the specified depth levels.

These results show the average time in milliseconds it takes for our AI to make a move, given the different depth levels. The experiments were performed on a system running Windows 11, and using a CPU with statistics given in table 3. All experiments where done against DumAI on a board-size of 8, and our AI won all the games.

CPU statistics	
Model name	11th Gen Intel(R) Core(TM) i9-11900K
Core(s) per socket	8
CPU speed	3.5 GHz
Max boost speed	5.3 GHz

Table 3: CPU specification

As it can be seen in table 2, the computation time of our algorithm increase drastically as the depth level increases. The higher the depth level, the better performance we see against the DumAI, yet we need to limit the depth level given the maximum time requirements specified in the project specification. Therefore the optimal depth level will be 10[1].

4 Conclusion

In this project we implemented an Othello AI using the adversarial search algorithm Minimax search. Given the time-to-move limit of 10 seconds we decided to have a depth limit for the cutoff at 10, based on the evaluations made in section 3. With a cutoff at depth 10, we managed to consistently beat the DumAI opponent.

Points of improvement For our implementation, the evaluation-function that associates a score to any given game state uses a static weight table, which gives an, on average, good score for the desirability of a position. However, this does not reflect the changing state of the game and might vary a lot in the accuracy of desirability. To improve the AI in the future, it would be beneficial to research popular tips and strategies and implement them into the evaluation function, to make the AI better adapt it's way of evaluating throughout the game.

[1] Using a depth level of 10 gives us an average time per move which is lower than the 10 seconds specified in the project description. This might, however, not be true if the AI is run on a weaker system

References

Peter Norvig. Chapter 18 - search and the game of othello. In Peter Norvig, editor, *Paradigms of Artificial Intelligence Programming*, pages 596–654. Morgan Kaufmann, San Francisco (CA), 1992. ISBN 978-0-08-057115-7. doi: https://doi.org/10.1016/B978-0-08-057115-7.50018-2. URL https://www.sciencedirect.com/science/article/pii/B9780080571157500182.

Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 3 edition, 2010.

Vaishnavi Sannidhanam and Muthukaruppan Annamalai. An analysis of heuristics in othello, 2015.