Othello - Group 30

Kim Ida Schild, Jowita Julia Podolak, Philine Zeinert

March 15, 2019

1 Introduction

The given Othello game implementation was extended by an additional AI-Player next to the `DumAI`. The AI is implemented by the class `AI`, which extends the interface `IOthelloAI`. It makes use of the minimax-algorithm including pruning operations as well as an evaluation function, that allows reasonable response time and higher winning chances for the AI-Player. The following sections provide a more detailed description of the implementation and the logic behind the `AI`-algorithm. Furthermore, we created a `RandomAI` for testing the winning chances of our AI, which will be described in the last section.

2 Minimax Algorithm with Alpha-Beta pruning

The game characteristics of Othello can be identified as a deterministic, two-player, turn-taking game with a zero-sum utility and observable, discrete states. This environment allows to make use of an algorithm, that is able to make strategic, informed decisions of the next move: the minimax-algorithm and its further developed version with alpha-beta pruning. The implementation is separated over three methods within the `AI` class, that call each other recursively along the game tree. The `decideMove` method is the entry point of the `AI` class and represents the minimax-method of the minimax-algorithm approach by returning an action, that serves as the AI's next move in the game. In Othello actions are represented by positions on the board game and symbolized by two integers identifying the indices in the two-dimensional array that keeps track of the tokens on the board.
`DecideMove` receives a `GameState` as a parameter to create a list of possible legal `Positions` for the AI. If the list is empty an illegal position is returned. If it is not empty, a loop over the `Positions` is executed. In each iteration, the current `Gamestate` is newly copied to a temporary `GameState`. This operation makes sure that modifications of the `Gamestate` along the leaves in the game tree do not effect the actual `GameState` but still enables calculations of possible utility values by counting the tokens on the board. For every `Position` the `MinValue` method is invoked. At each iteration a temporary `Gamestate` with the inserted token at the current `Position` is passed as a parameter to the

method. The returned utility value is compared to the current highest utility value stored in a local variable outside the loop. If the returned value is higher, then the local variable for the maxValue as well as a variable, that remembers the most favourable `Position` are updated, since the `AI` is aiming for the maximal possible utility value among all possible utility values of the opponent player (`MIN`). The `Position` will be returned at the end of the method `decideMove`. `MinValue` - method and `MaxValue` - method: `Min`- and `MaxValue` both receive a `Gamestate` as a parameter to create a list of legal moves. Before a list is create, it is checked if the current `Gamestate` represents a terminal state. If this is the case, the utility value will be returned. In the basic implementation of Othello, this meant that the end of a leave in a game tree was reached. The state was explored using depth-first search and a utility value was calculated. So far, the value was identified by counting the tokens on the board of the `AI`-player. If a `Gamestate` does not fulfill a terminal role, as in `decideMove`, legal `Positions` are evaluated by recursively calling the opposite method of `Min`- and `MaxValue` and passing the updated `Gamestate` with the token inserted to the corresponding method. `MaxValue` will return the highest utility value among all returned `MIN` - values at the end of the method, whereas `MinValue` returns the lowest utility value among all returned `MAX` - Values.

Integration of Alpha-Beta-Pruning: In addition to the described implementation, our algorithm is able to ignore irrelevant leaves of the game tree, which leads to improved response time by applying pruning. The `decideMove` method makes use of two variables, alpha and beta, that keep track of the best choice for the `MAX`-player (alpha) and `MIN`-player (beta). The variables are initialized within the `decideMove` method and set to the values `MIN VALUE` for alpha, as `MAX` always tries to reach the highest utility value, and correspondingly `MAX VALUE` for beta from the java class Integer. The variables are passed afterwards to the `Min`- and `MaxValue`-method as parameters along with the `Gamestate`. In the `MaxValue`-method the current highest utility value is compared to `MIN`'s beta value in every iteration. If the utility value is bigger or equal to beta the loop is terminated and the utility value returned. The algorithm does not continue to loop through the other possible `Positions`. The other states at this leave can be ignored, because `MAX` already prefers choosing this path to maximize its win instead of taking the former path where the expected utility value is lower.

Is this not the case the alpha value will be updated by comparing the current alpha value and the current highest utility value for `MAX` and choosing the maximal value of both. This ensures that the next time `MinValue` is invoked the new alpha value is forwarded. The `MinValue` method contains the same logic except that a leave/ subtree in the game tree is pruned in case the current lowest utility value for `MIN` is smaller or equal to alpha.
A possible performance improvement to this program could be, that we ensure that alpha values are passed from one leave over the root to the next leave. Hence, the alpha value is also updated within the iterations through legal moves

in the `decideMove` method and pruning can be applied earlier.

3 Cut-Off function and Evaluation function

Until this point, we decreased the size of the game's search space with alpha-beta pruning. However, our program still searches all the way to terminal states for a big portion of space, disallowing us to make a move in a reasonable time. In order to change that, we cut off the search earlier, by specifying a depth at which the search stops. This is done by replacing the terminal test by a cutoff test which returns utility not only after we reached the final state (`s.finished()`) but also when we reach the depth chosen by us.

After we improved the speed of making a move, we focused on improving the AI's chances to win. The URL supplied in the project description discusses some of the basic strategies of the Othello. Some positions are more valuable to capture than others. Therefore, instead of simply counting a number of white tokens at the terminal states, weighted values are applied for each position on the game board. We chose and applied three strategies:

- Corner positions are most valuable (discs in the corners cannot be outflanked)
- Avoid playing discs in the spaces immediately next to the extreme corners (X positions)
- Remaining edges' positions are good because they are harder to outflank

This is achieved by implementing a new method `Evaluation` which returns an integer `finalValue`. It iterates over all positions on the board and whenever a token of player 2 (white) is encountered, instead of simply increasing `finalValue` by one, the method distinguishes positions between: corner positions (adds up 5 to the `finalValue`), x positions (adds up 1 to the `finalValue`), remaining edges' positions (adds up 4 to the `finalValue`) and other positions(adds up 2 to the `finalValue`). The only exception occurs if the board is of size 4 - there we only apply the first point, because the board is too small to take other edge positions into consideration.

The resultant `finalValue` heuristic evaluation is returned in place of the utility function `s.countTokens` in the terminal conditions (either when no valid positions remain on the board or when we reach the defined depth, which is set to four) in `MaxValue` method and `MinValue` method.

After applying the evaluation function we could experiment a bit more with the running time for different depths. In order to be more sure about the AI performance, we created a class `RandomDumAI` which chooses the next move randomly from the list of legal moves. For the depth 4, game response is immediate for both classes. For depth 6, `DumAI` vs. `AI` deciding a move takes a few seconds on average (a bit faster for `DumAI`), and for 8 it is far too long. In the table below, running time results for depth 4 and 6 for both `DumAI` and `RandomDumAI` are displayed. Both depths lead to a win over `DumAI` and `RandomDumAI` in our tests. They can be chosen, depending on whether a very quick response or an even higher probability of winning are prioritized.

Opponent	RandomDumAI		DumAI	
Depth	4	6	4	6
	0,014	0,092	0,014	0,075
	0,021	0,085	0,044	0,095
	0,012	0,406	0,009	0,274
	0,012	0,284	0,003	0,071
	0,028	1,761	0,009	0,215
	0,022	2,282	0,042	0,421
	0,089	2,874	0,055	0,696
	0,066	13,308	0,1	0,719
	0,061	2,975	0,035	1
	0,115	1,963	0,133	4,381
	0,352	3,368	0,071	5,921
	0,075	5,839	0,03	0,886
	0,169	7,14	0,027	2,894
	0,091	3,192	0,02	0,864
	0,272	4,297	0,022	3,11
	0,037	16,31	0,021	0,86
	0,059	1,999	0,015	0,577
	0,064	0,831	0,01	0,252
	0,043	0,668	0,014	0,153
	0,034	0,542	0,008	0,223
	0,029	0,306	0,009	0,066
	0,014	0,138	0,016	0,044
	0,008	0,114	0,011	0,253
	0,009	0,12	0,019	0,102
	0,005	0,008	0,02	0,509
	0,003	0	0,005	0,097
	0,002	0,001	0,003	0,016
	0,001	0,002	0	0,004
	0	0,003	0	0
	0	0,001	0	0
	0	0		
	0	0		
Average	**0,05**	**2,22**	**0,03**	**0,83**
*results in seconds				

References

Russell S.J.,Norvig P. 2010., *Artificial Intelligence A Modern Approach Third Edition*, Person