Artificial Intelligence

Practice 2: Reversi

Jorge de Miguel Pires
Laura de Paz Carbajo
Group 2351
Team 07

Alpha-Beta pruning

Code

```
class MinimaxAlphaBetaStrategy(Strategy):
    def __init__(
        self,
        heuristic: Heuristic,
        max_depth_minimax: int,
        verbose: int = 0,
    ) -> None:
        super().__init__(verbose)
        self.heuristic = heuristic
        self.max_depth_minimax = max_depth_minimax

    def next_move(
        self,
        state: TwoPlayerGameState,
        gui: bool = False,
    ) -> TwoPlayerGameState:
        successors = self.generate_successors(state)
        minimax_value = -np.inf
        for successor in successors:
            successor_minimax_value = self._min_value(
                successor,
                self.max_depth_minimax,
                -np.inf,
                np.inf,
            )
            if (successor_minimax_value > minimax_value):
                minimax_value = successor_minimax_value
                next_state = successor
        return next_state

    def _min_value(
        self,
        state: TwoPlayerGameState,
        depth: int,
        alpha: int,
        beta: int,
    ) -> float:
        if state.end_of_game or depth == 0:
            minimax_value = self.heuristic.evaluate(state)
        else:
            minimax_value = np.inf
            successors = self.generate_successors(state)
```

```
            for successor in successors:
                successor_minimax_value = self._max_value(
                    successor, depth - 1, alpha, beta,
                )
                if (successor_minimax_value < minimax_value):
                    minimax_value = successor_minimax_value
                if (minimax_value < alpha):
                    return minimax_value
                if (minimax_value < beta):
                    beta = minimax_value
        return minimax_value

    def _max_value(
        self,
        state: TwoPlayerGameState,
        depth: int,
        alpha: int,
        beta: int,
    ) -> float:
        if state.end_of_game or depth == 0:
            minimax_value = self.heuristic.evaluate(state)
        else:
            minimax_value = -np.inf
            successors = self.generate_successors(state)
            for successor in successors:
                successor_minimax_value = self._min_value(
                    successor, depth - 1, alpha, beta,
                )
                if (successor_minimax_value > minimax_value):
                    minimax_value = successor_minimax_value
                if (minimax_value > beta):
                    return minimax_value
                if (minimax_value > alpha):
                    alpha = minimax_value
        return minimax_value
```

Implementation

The Alpha Beta pruning and the Minimax algorithm have a similar structure. Both have a main method called `next_move` which returns the best move for the current player to make and two auxiliary methods called `_min_value` and `_max_value` that represent the selection of the path to take depending on the type of node (min or max). In `_min_value` the beta value is updated with the minimum between the current beta value and the alpha values of the children. Meanwhile, in `_max_value` the alpha value is updated with the maximum between the current alpha value and the beta values of the children.

The `minimax_value` variable in both auxiliary methods lets us save the value obtained from the children we've just explored to compare it with the current alpha or beta value and update it if necessary.

Tests

To check that our implementation of alpha-beta pruning was correct we created some demo Reversi games with players that use this new strategy that has as arguments a heuristic and the maximum depth the algorithm is allowed to explore.

We ran several games between an ordinary minimax player and an alpha-beta minimax player, both using the same heuristic and maximum depth, and we could observe that their behavior was quite similar. They both took their time when calculating their next move but the difference was not that significant so as to claim that one algorithm is faster than the other one.

We tried varying the players' parameters (both heuristic and maximum depth) but we always reached the same result.

With these outcomes, we realized that the algorithm was not doing anything wrong but we wanted to prove that it was pruning correctly and therefore, it was more efficient than MiniMax.

We have just mentioned the time difference between 2 players that are the same except for the algorithm used to select their next move. By measuring this time we could check that the difference was not independent of the strategy used and that the player using Alpha-Beta pruning was faster than the one using MiniMax. So we can say that the algorithm is well implemented, in other words, it is pruning some branches of the search tree. (Some of these time measurements can be checked in the next section).

Efficiency

Computer dependent measures

To determine if the implementation of the Alpha Beta Pruning was correct we used the library timeit that was suggested in the practice. Using the command below, we run 5 different Reversi games in a 6 by 6 default board counting the time it took every couple of players to play it, and then we got the average time per game.
We created the players timeMiniMax1, timeMiniMax2, timeAlphaBeta1 and timeAlphaBeta2 that played with the strategies Heuristic1, EdgesAndCorners and WeightAndTimes (the last two are two of the ones that we submitted to the class tournaments) at levels 3 and 4 of depth.
Here is the piece of code we used to measure the time it took the players to play the 5 games with each strategy:

```
timeit.timeit("match.play_match()",
              setup="from __main__ import match",
              number=5)
```

With this method we got the following average times per game for the different heuristics and algorithms:

	Minimax	**Alpha Beta Pruning**
Heuristic1	41.7884	40.6027
EdgesAndCorners	62.4501	27.2405
WeightAndTimes	58.3773	24.2681

We can extract from the table above that the Alpha Beta pruning improves as the heuristic improves, going from around 40 seconds per game to a little more than 20. This improvement is obtained due to the Alpha Beta algorithm discarding more branches of the graph as the heuristic improves. This makes the player save some time by not checking every possible child.
Also we see that the Minimax algorithm, since it doesn't take advantage of the heuristic as much as the Alpha Beta does, it goes slower. That's because as the heuristic improves, its complexity is also bigger, and it takes more time to process and get the values of each node.

As a conclusion, we saw that using a good heuristic makes the Alpha Beta algorithm work much better compared to Minimax. But as the heuristic gets better, it also makes the Minimax algorithm work slower due to the computational complexity.

Computer independent measures

All the tests above were run on the same computer thus the comparison between both algorithms should not depend on the computer itself. Even so, we thought about some ways of measuring the algorithm independent of the computer you execute it in.
We came up with the idea of counting the number of times each algorithm with each strategy called the methods `_min_values` and `_max_values`. The number of calls is independent from the computer and constant to the strategy, that's why it's a good measure of the time that will consume each algorithm.
Here are the results we obtained:

	Minimax	**Alpha Beta Pruning**
Heuristic1	74938	74938
EdgesAndCorners	114822	28453
WeightAndTimes	102653	32860

We can see that the number of times each algorithm calls these functions is somewhat related to the time shown in the previous section. As the pruning doesn't take place in the Minimax algorithm, it can't discard some branches and it ends up calling approx 3 times more than the Alpha Beta to the min and max functions.

Heuristics

References on Reversi strategies

We searched for some information on the Internet to get to know the game better and the different strategies people had already studied to make the most out of every movement. We found some interesting researches such as the ones detailed below:

Sannidhanam, V., Annamalai, M. (2015). An Analysis of Heuristics in Othello.
https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf

Greene, W.A. (1991). Machine Learning Of Othello Heuristics.
http://cs.uno.edu/people/faculty/bill/ML-Othello-Heuristics-ACM-SE-Regl-1991.pdf

Festa, J., Davino, S. (2013). "IAgo Vs Othello": An artificial intelligence agent playing Reversi.
http://ceur-ws.org/Vol-1107/paper2.pdf

Design process

Our objective in this task was to design the best possible heuristic, one that could win any opponent. In order to get there we knew we had to test several heuristics.

First we began by implementing the most simple heuristics we could think of and then designing more complex ones by coming up with new ideas, improving others already implemented or even combining them.

We first design the heuristic called **PieceDifference**, whose evaluation function calculates, as its name suggests, the difference between the number of black and white pieces in the board and assigns a positive or negative sign whether the state which is being analized is from a max or min player, respectively.

After simulating a considerable amount of games, we realized that both the corners and the edges were key positions in the game. Once a player captures a corner, it will never change its color no matter the moves of the opponent. In the case of the edges, they can still be stolen by the opponent but it is much less likely than a position in the middle of the board since an edge has fewer adjacent positions from which it can be captured, 2 to be precise. That's why we implemented **Corners** and **Edges** heuristics and tested them against each other and against the previous heuristic as we explain in detail below.

Later we considered joining the heuristics mentioned above to make a global heuristic that would take advantage of all these parameters. Following this thought, we created the heuristic **PiecesEdgesCorners** which calculates the value of each move by adding up all the parameters mentioned in the last paragraph. After simulating quite a few tournaments between our heuristics, we realized that this new one could easily beat the other 3.

To improve **PiecesEdgesCorners**, we made a heuristic called **Weights** in which we weighted each of the parameters (pieces, edges and corners) with a constant value, depending on what we considered more important to win the game. This heuristic didn't improve that much compared to the last one mentioned, but we were starting the games

from an advanced stage of the game because from the initial board the executions lasted much longer. Thus, we thought this heuristic could improve in the actual tournaments.

Later on, we researched a little bit about general strategies for the Reversi game, and we found out on the Lazard website[1] that, depending on the stage of the game, the weights mentioned above had to change in order to make the strategy the best one possible. That's why we implemented two more heuristics. First we created a private function called **turn_number** that calculated the turn depending on the amount of pieces that were laid down on the board, and then we used that function to create the heuristics **WeightsAndTimes** and **WeightsAndTimes2**. Both follow a similar strategy, in the early stages of the game, it tries to catch as few pieces as possible and go for the edges and corners. As the game advances, it starts to capture as many pieces as possible and gives less weight to the corners and edges. The main difference about these two heuristics is that the second one also takes into account the number of available moves that the player has. In this case, as the game advances, it gives more weight to capturing more pieces and having more possible moves.

Tests

To test the heuristics we created a global file called ourHeuristics.py in which we kept all the heuristics we design. Then we duplicated the file demo_tournament.py and renamed it to demoTournHeuristics.py in which we tested our heuristics by executing many tournaments and having them play against each other. First we started playing tournaments from an advanced stage of the game in which every heuristic participated, and as we were creating more heuristics, we decided to include only the better ones and have them play from an initial 8 by 8 board.

We also decided which heuristics to improve according to their position in the class tournament posted in Moodle. The ones that obtained the greatest amount of points were the ones we wanted to keep or even improved, if possible.

Final heuristic

After executing many tournaments and trying to get the weights of the parameters as tuned as possible, we decided to select the heuristics **PiecesEdgesAndCorners**, **Weights** and **WeightsAndTimes2**. These three were the ones that gave us the best results in our tournaments and every time we did one, the three of them would tie with the same amount of wins.

[1] Lazard, E. (1993, March). *Othello Strategy guide.* http://radagast.se/othello/Help/strategy.html