Umeå University
Department of Computing Science

Artificial intelligence
Methods and applications

5DV181

Othello artificial intelligence

Thomas Ranvier

supervised by
Ola Ringdahl
Juan Carlos Nieves Sanchez

November 30, 2018

Abstract

In this report I present my implementation of the Alpha Beta algorithm for the Othello game.

This implementation uses Java, in the first part it is explained how to run the program. The program uses an iterative deepening Alpha Beta algorithm, the implementation details are presented, the details of the heuristics function are then also presented. The program has been tested in both configurations using time limits between 1 and 10 seconds, my program beats the naive program every time.

Contents

Chapter 1

How to run my code

To run my code you have to use the 'othellostart' script and put in parameters the two scripts that you want to use, the first one is the white player, the second one is the black player, the variable 't' is the time limit.

```
./othellostart .script_1 .script_2 t
```

Chapter 2

My implementation

I implemented my program in Java, I used the given helper code as reference during my work but I implemented everything in my own way.

2.1 The makeMove function

This is the description of the makeMove function.

I choose to realise the clone of the current position using another function which is directly called from the alphaBeta function, it is then not included in the makeMove function.

The first thing done in the function is to check if the asked move is in the grid boundaries and if it is not a pass move. Then it sets the cell selected by the move to the owned player. After that for all eight directions it calls the function checkDirection which returns true if the enemy cells on that direction are being captured. If it returns true the function fillDirection is called, this will capture the enemy cells in between by replacing them by the owned player. Once the process is done the player is changed for its opponent.

Here is the pseudocode of that function.

Algorithm 1 make move

```
1: procedure makeMove(x, y)
2:     if x and y are in boundaries then
3:         this.grid[x][y] ← this.player
4:         for every direction around the cell as dir_x and dir_y do
5:             if this.checkDirection(x, y, dir_x, dir_y) then
6:                 this.fillDirection(x, y, dir_x, dir_y)
7:             end if
8:         end for
9:         this.changePlayer()
10:    end if
11: end procedure
```

The variables 'dir_x' and 'dir_y' corresponds to the values that one has to add respectivelly to 'x' and 'y' in order to progress along the corresponding direction. To access to the eight possible directions we just have to make 'dir_x' and 'dir_y' loop through the tree following values: [-1, 0, 1] and not consider the direction where both 'dir_x' and 'dir_y' are 0.

2.2 Alpha Beta implementation

I implemented the Alpha Beta algorithm following the wikipedia pseudocode. My code can be found in the AlphaBeta.java file that contains the class which is used to determine the best move to do.

Here is the pseudocode of my implementation adapted to the Othello game.

Algorithm 2 Alpha Beta

```
1: procedure alphaBeta(initial_position, depth, alpha, beta)
2:     if depth == 0∨ no more valid moves then
3:         return a move with value set to the evaluation
4:     end if
5:     maximizing ← initial_position.getPlayer() == this.player
6:     best_move ← a default move
7:     value ← −∞ if maximizing, +∞ otherwise
8:     for all move from valid moves do
9:         new_pos ← a clone of initial_position
10:        new_pos.makeMove(move)
11:        res_move ← alphaBeta(new_pos, depth − 1, alpha, beta)
12:        if maximizing then
13:            if res_move.value > value then
14:                value ← res_move
15:                move.value ← value
16:                best_move ← move
17:            end if
18:            if value ≥ beta then
19:                move.value ← value
20:                return move
21:            end if
22:            alpha ← max(value, alpha)
23:        else
24:            if res_move.value < value then
25:                value ← res_move
26:                move.value ← value
27:                best_move ← move
28:            end if
29:            if value ≤ alpha then
30:                move.value ← value
31:                return move
32:            end if
33:            beta ← min(value, beta)
34:        end if
35:    end for
36: end procedure
```

The recursive alphaBeta function returns an Othello move.

When the maximal depth is reached or if no other moves can be performed the function returns a move with a value set to the evaluation value. The evaluation value is determined by the evaluator which analyses the board of the game and returns a value that is high if the evaluation for the actual player is 'good' and lower otherwise. The evaluator uses a heuristic function that will be described later in this report.

The recursive function calls itself back and since when the 'makeMove' function is called the player is changed, the function can automatically know if it must maximize or minimize its selection. Indeed when the function is selected a move for its own player the higher the evaluation score the better, this is why the function maximizes its selection in that case. On the other hand when it is selecting a move for the opponent if the evaluation score is low it means that it is a good move for the opponent so the function minimizes its selection in that case.

2.3 Iterative deepening

To handle the time limit the solution is to use iterative deepening for the alphaBeta function. There is an internal timer in the AlphaBeta class, this timer is checked at each iteration through the valid moves in the alphaBeta function and as soon as the time is reached the function throws an exception.

The alphaBeta function is initially called with a depth of 5 (since it is always way under 1 second) and then the depth is incremented by 1 and the function is called again with the new depth. That process repeats itself until it catches an exception, then the whole process stops and the move saved from the previous depth is returned as best move. The maximal depth is set at 20, indeed there is no point to try to analyse the game deeper because it would just be a waste of time when the game is near to end.

Here is the pseudocode of the bestMove function which implements the iterative deepening described above and also directly returns a pass move if the game is over.

Algorithm 3 best move using Alpha Beta

```
1: procedure BESTMOVE(initial_position)
2:     if number of valid moves == 0 then
3:         return a pass move
4:     end if
5:     depth ← 5
6:     best_move ← a default move
7:     while depth < 20 ∧ current time < time limit do
8:         try {
9:         best_move ← this.alphaBeta(initial_position, depth, −∞, ∞)
10:        depth ← depth + 1
11:        } catch {}
12:    end while
13: end procedure
```

2.4 Used heuristics

The heuristics function is what evaluates a board in order to determine if doing a given move is a good idea or not. It is the part that differenciates the artificial intelligences from one another, indeed a well balanced and adapted heuristics function will perform way better than a naive heuristics function.

However an evaluation function that is too costly can make the Alpha Beta algorithm go less deep in the recursion and can also be hard to balance.

2.4.1 Static weights matrix

To build my evaluation function I first created a grid that contains weights, each weight corresponds to a cell of the grid. Then to evaluate the board I sum the weights values of the cells owned by each player and return the difference between the total of my player and the total of the other one. By doing so my program could already beat the naive implementation without any problem.

This is the final weights matrix that is used in my program.

20	-3	11	8	8	11	-3	20
-3	-7	-4	-1	-1	-4	-7	-3
11	-4	4	2	2	4	-4	11
8	-1	2	1	1	2	-1	8
8	-1	2	1	1	2	-1	8
11	-4	4	2	2	4	-4	11
-3	-7	-4	-1	-1	-4	-7	-3
20	-3	11	8	8	11	-3	20

We can see that the corners have a high value, that is because when a player captures a corner that piece cannot be captured back by the other player. The pieces on the sides also have a high value because once they are captured by

a player they are harder to be captured back by the other player since they can only be captured from 2 directions instead of 8 for the interior pieces. The pieces around the corners have negative weights, indeed to capture a corner it is needed that the opponent capture one of those pieces, so having negative weights in those places make the program avoid capturing them unless it is really worth it. The negative weights that are not around the corners have the same purpose that is to facilitate the capture of the pieces on the sides.

2.4.2 Actual mobility evaluation

To improve my evaluation function an idea was to consider the possibility of my player future mobility versus the opponent future mobility. Indeed the more moves are possible to do the more good potential choices will appear for us in the future. Another point is that the less moves the opponent can do the easier it will be to beat it.

To evaluate the mobility of a player it is very simple, we just have to call the 'getValidMoves' function of the position and the size of the returned vector is the number of possible moves that the player will access to. Then to get the mobility of the opponent we just have to change the position player, call again 'getValidMoves' and then switch the players again to return in the initial state of position.

The returned value of the function is just the difference between the mobility of the player and the mobility of its opponent.

2.4.3 Pieces stability

The last heuristic that I added to my complete evaluation function is the stability evaluation.

To check if a given cell of the board is totally stable we would need to build a complex and especially very not efficient function. We are then in the case where we need to simplify the function to a point where the function is efficient enough not to slow the Alpha Beta algorithm too much but also give good evaluation results.

I decided to consider a piece as stable if it cannot be captured by the opponent by placing a piece adjacent to the cell the very next move and unstable otherwise. That kind of stability is called edge stability and it is not very costly to compute it. Indeed it would be too costly to determine if a piece is definitely stable in the game and the results would not be much better anyway.

The final function that determines if a piece is stable follows that pseudocode.

Algorithm 4 is a stable cell

```
1: procedure isStableCell(grid, size, x, y)
2:     for every direction around the cell do
3:         if the first neighbour of the cell in that direction is empty then
4:             if there is an enemy or empty cell in the opposite direction then
5:                 return false
6:             end if
7:         end if
8:     end for
9:     return true
10: end procedure
```

I added the stability evaluation to the function that uses the static weights to determine the value of each cell in the grid. Now when a cell is owned by a player the value of that cell is equal to the static weight plus a bonus if the cell is stable for the next move. The bonus added to the cell value has a value of 8, which is the value that I found gave the best results.

Chapter 3

Obtained results

To evaluate my program performances against the naive program I made them fight against each other using both possible compositions and time limits between 1 and 10 seconds, those are the results obtained using the final heuristic function described in this report.

Time limit	1	2	3	4	5	6	7	8	9	10
Score as white	36	56	54	54	52	58	54	54	54	58
Score as black	54	60	54	62	62	42	44	48	46	46

As we can see with those results, no matter the time limit my program beats the naive program very easily.

I also made my program compete against the one of an other student to try to make it better against more advanced programs. I improved the heuristic to be able to beat more easily my opponent and I then ran the tests with the naive program again, those are the obtained results.

Time limit	1	2	3	4	5	6	7	8	9	10
Score as white	58	56	58	52	56	52	58	52	62	61
Score as black	60	60	60	42	42	36	36	36	44	44

It was interesting to notice a drop of performances with time limits greater than 3 seconds against the naive program when my program plays the black player. The performances against the other student were however way better playing both players.

Chapter 4

Encountered issues

I initially implemented the Alpha Beta algorithm using Python 3. My implementation worked but for a reason that I was not able to find, even with the help of another student the program was incredibly slow, in mid game it was not able to reach a depth of 7 under almost 20 seconds.

After a while I decided to simply reimplement the exact same program using Java and the program was then able to reach a depth of 7 under half a second at any point in the game.

This was the main issue that I encountered, after switching to Java I had no other difficulties.

Conclusion

To make that program better the main improvements would be done on the heuristics function.

An idea could be to adapt the heuristics function to make it evolve depending on the number of moves left. Indeed it is interesting to minimise the number of owned pieces in early game because by doing so the opponent will have a very low mobility where our player will have a lot of possibilities. In the end game however it is the opposite, since most of the board is filled both players don't have a lot of mobility left and at that moment the best thing is to maximize the number of pieces that we own to win the game.

As explained in the stability part of the heuristics function the stability evaluation is not complete at all in my implementation, it should be possible to improve it without decreasing the speed of the whole process too much.

Of course it could also be possible to improve the efficiency by improving some functions in the Position class that go through all the cells of the board where it is not always necessary.

List of Algorithms