Umeå University
Department of Computing Science

Artificial Intelligence - Methods and Applications
5DV181

Othello

Valentin Lecompte
ens18vle

supervised by
Ola Ringdahl
Juan Carlos Nieves Sánchez

November 30, 2018

Contents

Chapter 1

General information

1.1 How to run the program

The assignment has been developed in Java using the helper code. The sources are in the *src* folder. A script *./othello.sh* is in the root folder in order to execute and compile the Java code.

The script *./othello.sh* is based on the given script with the same name. It runs the same way : *./othello.sh a b c*. Where

- *a* is a string of 65 character representing the current state of the othello grid.
- *b* is the maximum execution time of the AI in seconds.
- *c* is a boolean set to *true* if the script should compile the Java code. If this parameter is set to true, the script will only compile the Java code, it will not execute the AI.

1.2 List of files

Following, the list of all the files that are in the Java source folder.

- **Othello.java**: Input file of the AI. The main function is in this file.
- **OthelloAlgorithm.java**: interface for the Alpha-Beta algorithm.
- **OthelloEvaluator.java**: interface for the evaluators of the Alpha-Beta algorithm.
- **OthelloPosition.java**: class representing the board.
- **OthelloAction.java**: class representing an action on the board. The action is either a coordinate (i, j) or is "pass".

- **AlphaBetaAlgorithm.java**: class implementing the Alpha-Beta algorithm.
- **MiniMaxEvaluator.java**: class representing the first heuristic implementation.
- **NaiveWashingtonEvaluator.java**: class representing the second heuristic implementation.

Chapter 2

Implementation of the algorithm

2.1 Global description

The main function (in **Othello.java**) create an instance of *OthelloPosition* parsing the input string. This instance of *OthelloPosition* represent the current state of the board.

Then it create an instance of *AlphaBetaAlgorithm* and set the *stopAt* timestamp. Then it loop through the depths and call the function *evaluate* of the *AlphaBetaAlgorithm* instance.

The function *evaluate* call $evaluateMax$ if the current player is the white one, otherwise it calls the function $evaluateMin$. The function *evaluate* return an instance of *OthelloAction* representing the best action to do for the AI.

2.2 The makeMove() algorithm

The following pseudo-code describe how I implemented the $makeMove()$ function. The first thing that the function does is to check whether the action is to pass or not. If the action is to pass, the function just return the current board changing the player to move.

Then for each direction of the action (at most the directions are East, North-East, North, North-West, West, South-West, South, South-East), the function will check which ones are eligible. A direction is eligible if there is a square of the current player in this direction. Moreover, the square closest to the action must belong to the opponent.

Algorithm 1 makeMove()

```
1: procedure MAKEMOVE(action)
2:     nextBoard ← clone the current board
3:     cPlayer ← 'O' if the white player is playing, 'X' otherwise
4:     nextBoard[action.row, action.column] ← cPlayer
5:     for each direction d do
6:         if the direction d is eligible then
7:             (i, j) ← position of the first opponent square in d direction
8:             while nextBoard[i, j] is not a square of the current player do
9:                 nextBoard[i, j] ← cPlayer
10:                (i, j) ← position of the next square in d direction
11:            end while
12:        end if
13:    end for
14: end procedure
```

2.3 The heuristic implementation

I tried 3 different heuristics. The first one is based on a naive interpretation of the board. The 2 others heuristics implementations are based on the paper "An Analysis of Heuristics in Othello" [1].
The first heuristic implementation increase the score of the player depending on the position of his squares in the board. Here are the specification depending on the position:

- The corners cost 5 points. Once a corner is taken, the opponent can't take it back.
- Each cells of the edges costs 2 points.
- All the other cells cost 1 point.

Then, it will return the difference between the score of the white player and the block player. If the difference if positive, the white player have a better score, otherwise the black player have a better score.

The second heuristic implementation is based on a static weight for each cell of the board. The weight is based on the importance of the cell to change opponent's cells. For instance, the corners have a high value but the cells next to the corner have a negative value: they can easily changed with the corner.

10	-3	6	3	3	6	-3	10
-3	-5	-1	-1	-1	-1	-5	-3
6	-1	1	0	0	1	-1	6
3	-1	0	1	1	0	-1	3
3	-1	0	1	1	0	-1	3
6	-1	1	0	0	1	-1	6
-3	-5	-1	-1	-1	-1	-5	-3
10	-3	6	3	3	6	-3	10

Table 2.1: Static weight of each cell of the board

The third heuristic implementation is more complex. The score of a player is based on 4 properties: the parity, the mobility, the corner captured and the stability [1].

My implementation of those properties is bugged and this Evaluator did not give relevant score.

2.4 Time to depth translation

In order to stop the AI when it reach the max running time, I choose to throw **TimeOutException** an exception. To do so, the main function parse the input time limit into the timestamp defining the end of the Alpha-Beta search. I choose to subtract 20 milliseconds to the timestamp to ensure that the program does not go over the time limit.

The main function has a loop incrementing the depth search for the Alpha-Beta search. The loop goes from a depth of 7 (the running time is always strictly bellow 1 second) to a depth of 20 (the running time is always greater than 10 seconds).

A try-catch in the loop will catch the Exception if it is thrown.

Algorithm 2 Time to depth translation

```
1: bestAction ← new OthelloAction()
2: for depth ← 7...20 do
3:     alphaBeta.setDepth(depth)
4:     try {
5:     bestAction ← alphaBeta.evaluate(position);
6:     } catch (TimeOutException e) {
7:     break;
8:     }
9: end for
10: print bestAction
```

The exception is thrown in $evaluateMin()$ or $evaluateMax()$ thanks to the function $checkTime()$.

Chapter 3

Experimentation and analysis

The following table represent the results of the AI with the first heuristic implementation. For each time limit, 5 runs has been made: then table contains the average of the runs.

We can see that the score as white is slightly better than the score as black.

Time limit	1	2	3	4	5	6	7	8	9	10
Score as white	61	62	60	48	62	61	60	61	62	62
Score as black	50	62	34	32	34	60	62	62	62	62

Table 3.1: Result of the AI with the first heuristic

List of Algorithms

Bibliography

[1] Vaishnavi Sannidhanam and Muthukaruppan Annamalai. An analysis of heuristics in othello. Master's thesis, University of Washington, 2003. https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf.