UMEÅ UNIVERSITY
Department of Computing Science
Report

Using alpha-beta pruning to make an Othello agent

OU1
Artificial Intelligence - Methods and Applications
57219HT18

Name	Frans-Lukas Lövenvald
CAS	frlo0015@umu.se
CS	c16fld@cs.umu.se
Date	2018-11-20

Teachers: Juan Carlos Nieves Sanchez, Ola Ringdahl

Contents

1 Introduction

Othello is a two player deterministic fully observable game which means it can be exploited by searching for the best move. One searching algorithm that takes into account how both players move is the min-maxi algorithm. To further expand upon the min-maxi algorithm this paper will present how the alpha-beta pruning algorithm can be used to make an effective Othello bot.

2 System Description

The Othello program includes a position, an action, an evaluator and the algorithm. Each of which has a part in helping making a good move. Each part including compilation instructions will be described in this chapter.

2.1 Compilation

To run and compile the program a shell script is included. The shell script is in turn given to a executable java file which takes three arguments, the white player, the black player and a time limit for each move the players can make. To test the program against a player; type the following in the test_code folder:

./othellostart whiteplayer blackplayer timelimit

The white and black player can be replaced by the Othello shell script which starts the bot described in this report or your own implementation of an Othello bot. To start the program with the described bot against a naive implementation; type the following:

./othellostart ../othello.sh ./othello_naive 1

This will start the naive bot as the black player with a time limit of 1 second against the bot described in this report.

2.2 Othello Position

The position class handles the board state and what player is to move. It also changes the board state when a move is done. It includes some helper methods for the other classes to use, such as counting of empty spaces, the number of points to make from a move and a method that returns all available moves.

2.3 Othello Action

This class includes a value and a coordinate for a move to be made. It is given as argument to the Othello Position makeMove method when a move should be applied to a position.

2.4 Othello Evaluator

The Evaluator class implements the evaluator interface to decide the "wellness" of a move, i.e. how good a move is in the form of an integer. A higher value is better and a lower value is worse. A evaluation can be negative. The implementation done for the bot described in this report is a naive implementation with a weight on the different board coordiantes. Corners are worth 20 more points than non corners.

2.5 Othello Algorithm

The algorithm class is the class the calculates what move to do given an Othello Position. It uses the alpha-beta pruning[2] algorithm with account for the time limit given as the argument for the

program. The algorithm is described more in depth below.

2.6 Heuristic

The heuristic used for evaluating a board state is a simple modification of the naive herusitic. The naive heurisitic counts the number of disks for each player and gives a positive score if the bots color have more disks than the enemy color and a negative value if the opposite is true. The difference done from the naive heuristic is that corners counts as 20 points instead of one. This motivates the bot to make moves which leads to it taking corners.

2.7 Time into depth

Since the bot has a time limit for each move it needs to take into consideration what the timing is and interrupt its search when the time limit is reached. This is solved by incrementally increase the depth of the alpha-beta search each time the search completes within the time limit. When a node in the search is reached when the time limit has been passed, the search is interrupted and the value obtained from the search so far is invalidated and not set on the current searched action. The algorithm starts with a depth of 5 as it will always reach at least depth 5 within the lowest time limit of 1 second.

2.8 Alpha-beta

The alpha beta search is parallelized for each action available at the start of the search and the evaluation function will evaluate for the color of the player color given to the alpha-beta method, i.e. $isWhite = trueorisWhite = false$. Other than that the min-maxi is doen as explained in [2].

2.9 makeMove() in othelloPosition

The makeMove method in the othelloPosition class applies the action given to the method and flips all the pieces, changes to the opponents turn and returns a itself. The method first makes sure the move is valid by counting the score gained for the move, if the score is zero; the move is invalid. Otherwise it flips the disks that are flippable. It does this by looking at the neighbours of the disk to be placed, if a neighbour is of the opponents color, it will trace the direction until either the trace leads to an empty state, or out of bounds or until it reaches a friendly disk. Only if it reaches a friendly disk will it flip all the opponents disks in between the friendly disk and the disk to be placed.

3 Testing

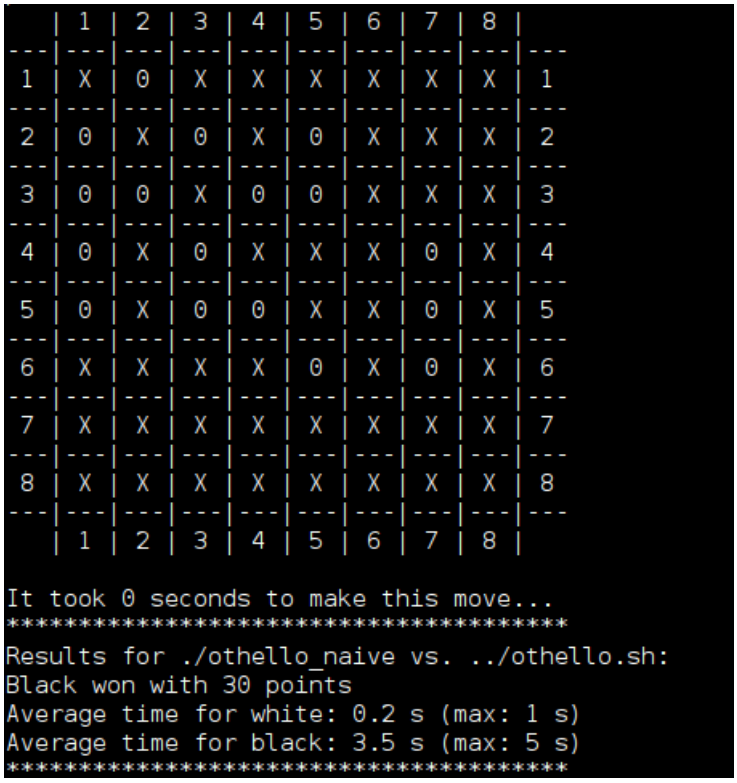

Figure 1: The result when the bot played against a naive implementation with a time limit of 4 seconds as black.

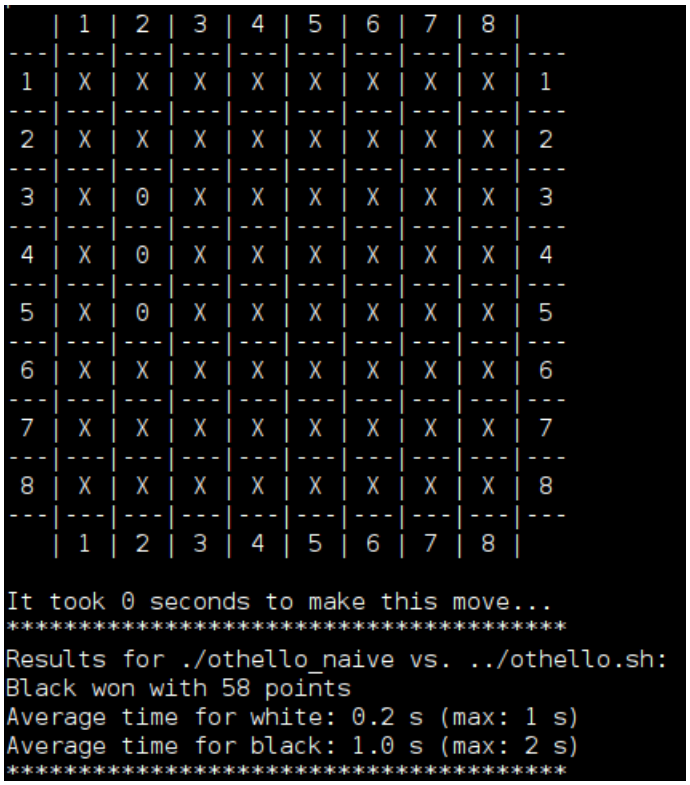

Figure 2: The result when the bot played against a naive implementation with a time limit of 1 seconds as black.

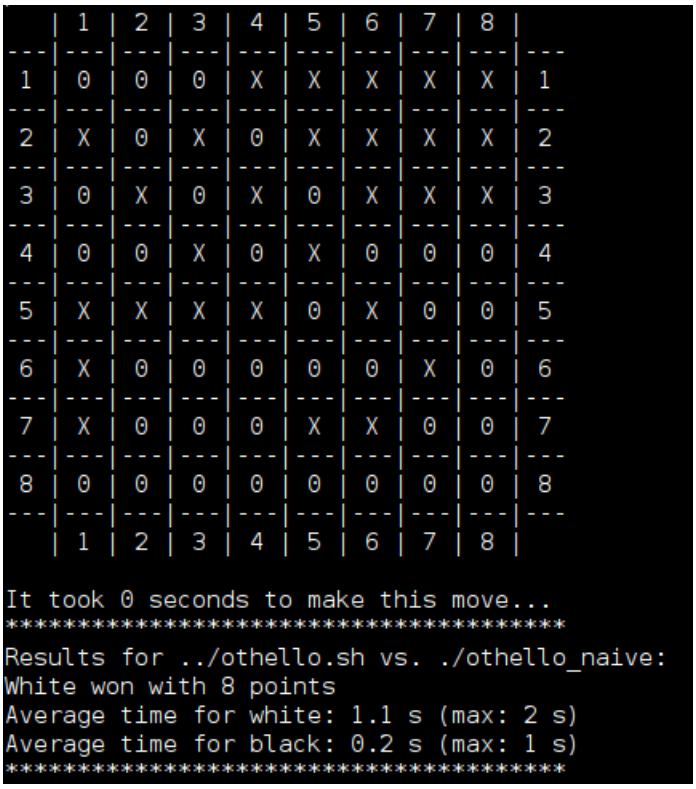

Figure 3: The result when the bot played against a naive implementation with a time limit of 1 seconds as white.

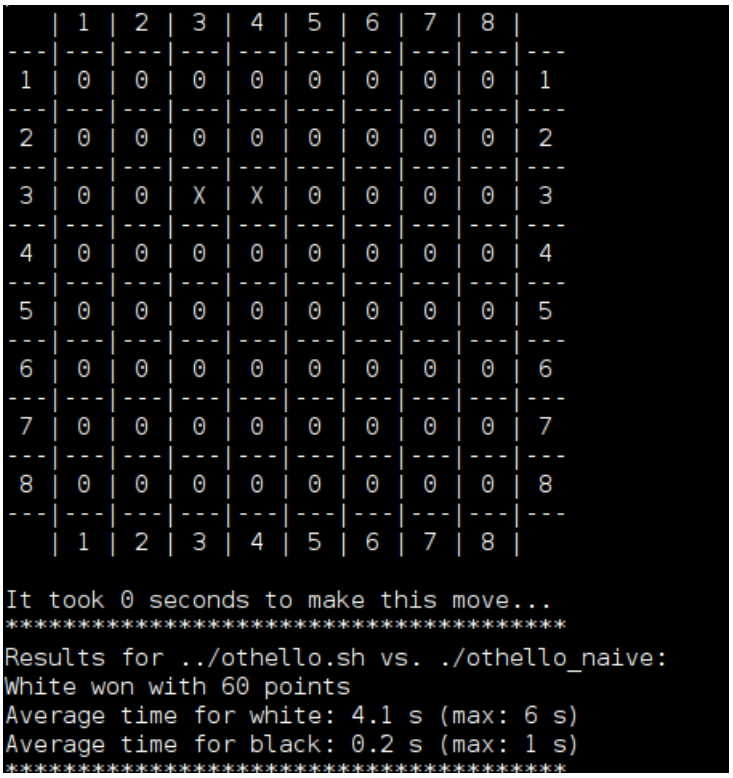

Figure 4: The result when the bot played against a naive implementation with a time limit of 5 seconds as white.

As can be seen in figure 1-4 the bot handles both different time limits and winning as both black and white against the naive implementation.

4 Further work

The alpha beta algorithm can be updated to not be called for each possible move from the starting position but to only return the best action possible while also pruning the start moves if necessary. The makeMove method should also be made to return a clone of itself as to simplify the other methods in the project. The evaluation method can also definitely be improved, one such suggestion is to count mobility, potential mobility and stability instead of only counting the disk parity [1]. For the bot to be able to reach deeper into the search the board representation and actions could be optimized. One such optimization could be to represent the board as two 64-bit integers, one for the positions of the white player and one for the positions of the black player. This needs further investigation.

5 Reflection & Workflow

The work went fast at first, the implementation of the missing parts of the position class went smoothly with unit tests written along the development. The first implementation of the evaluation class and the algorithm class went smoothly as well. However, the implementation performed poorly at first because of many small errors in how the alpha-beta was implemented. An example of an error was that the alpha-beta started with the max method when it should have started with the min function as I was choosing the max value after having called alpha-beta on every action possible. Another example of an error was that the evaluation method was always evaluating for maximizing the opponents score and not the bots score. Finding these errors was time consuming because no clear way of effectively traversing the search tree was found.

6 References

[1] Inside Reversi/Othello. http://home.datacomm.ch/t$_w$*olf/tw/misc/reversi/html/index.html.Accessed* : $2018 - 11 - 21$.

[2] Min-maxi and alpha-beta pruning. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/. Accessed: 2018-11-20.