université de BORDEAUX

Master CSI

# Reversi: Advanced Computer Player

Nicolas Neudeck

22. November 2019

Bordeaux

# Contents

# 1 Introduction

Game playing, as one of the most challenging fields of artificial intelligence has received a lot of attention. Games like Reversi, which have proven to fit in well with computer game playing strategies, have spawned a lot of research in this direction.
Othello, how Reversi is called in most parts of the world, is a relatively modern game. Its origins lie in England at the end of the nineteenth century but the game as it is known now was set down in Japan as recently as 1971.[3] It hails from the Go family of board games where the primary objective is to capture territory on the board by surrounding the opponent's pieces.
The motto of Reversi is "Easy to learn, difficult to master" and as such it is ideal for computer implementation. A rather simple Reversi Game can be implemented in Java or python, using generic data structure such as bidimensional arrays. But for this project the performance, code quality (robustness, efficiency, bug free, ease of use...) and memory handling had first priority. So, throughout a five-week course at the University of Bordeaux, the game mechanics and rules were implemented in native C. Starting with the build-system and option parser, a board module as well as a file parser were implemented. A bitboard data structure [2] for the game board and player module followed. To finish the fully human playable reverse game, the main loop and a random computer player were tackled.

```
   A B C D E F G H
1  _ _ _ _ _ _ _ _
2  _ _ _ _ _ _ _ _
3  _ _ _ _ _ _ _ _
4  _ _ _ O X _ _ _
5  _ _ _ X O _ _ _
6  _ _ _ _ _ _ _ _
7  _ _ _ _ _ _ _ _
8  _ _ _ _ _ _ _ _

Score: 'X' = 2, 'O' = 2

'X' player's turn.
Give your move (e.g. 'A5' or 'a5'), press 'q' or 'Q' to quit:
```

Leading up to this report, which focuses on the further implementation of a game playing artificial intelligence. Artificial intelligence applications in board games have been around as early as the 1950's, and computer programs have been developed for games including Checkers, Chess, and Go with varying results. Even though numerous computer Reversi players have been designed, and have beaten human world champions, it is not very clear as to how the various Othello heuristics interact [9]. The over 50 years of trying to build the best Reversi AI gave us a lot of research material to work with. Here some of the most useful examples:[7] [5] [1]
As most game playing programs, the AI's main components are its search algorithm and its evaluation function.[4] For the search algorithm, a minimax tree search algorithm with alpha-beta pruning up to a given depth was chosen. The evaluation function combines different heuristics with weights depending on the game stage. All in greater detail in chapter 2 Algorithms. The chapter 3 Implementation deals with how to implement those algorithms and still have a decent performance. And lastly the Champter 4 Extra Efforts is about assuring code quality and robustness.

# 2 Algorithms

The minimax tree search algorithm is used to minimizing the possible loss for a worst case scenario. It's a recursive algorithm for choosing the next move in an two-player game.
A value is associated with each position or state of the game. This value is computed by means of a position evaluation function and it indicates how good it would be for a player to reach that position. The player then makes the move that maximizes the minimum value of the position resulting from the opponent's possible following moves. This can be extended if we can supply a heuristic evaluation function which gives values to non-final game states without considering all possible following complete sequences. We can then limit the minimax algorithm to look only at a certain number of moves ahead. In Reversi, the game often takes more than 60 turns to produce a winner, we use a heuristic function to evaluate the current game stage value by only looking ahead a couple of moves. This heuristic function is actually a collection of several heuristics and calculates the utility value of a board position by assigning different weights to those heuristics.

## 2.1 Heuristics

When looking at a Reversi board, its not clear at first sight which player has the advantage. Its easier to just focus on separated aspects of the playing board. Firstly we look at the coin parity, the most obvious one.

### 2.1.1 Score Heuristic

The `int score_heuristic(board_t *board, disc_t player)` function returns the difference in coins between the max player and the min player.

### 2.1.2 Mobility Heuristic

The `int board_mobility(board_t *board, disc_t player)` function attempts to capture the relative difference between the number of possible moves for the max and the min players, with the intent of restricting the opponent's mobility and increasing

one's own mobility.
To compute the possible moves for a player, the paper "Bitboard Methodes for Games" [2] serves us with a fast function using bitwise operators. This function, which returns a bitboard, is called for both players. For the resulting bitboards the possible moves (bits set to 1) are counted. Finally the amount of possible moves the opponent has is subtracted from the possible moves the player has.

```
size_t player_moves = bitboard_popcount(
  compute_moves(board->size, player_bitboard, opponent_bitboard));
size_t opponent_moves = bitboard_popcount(
  compute_moves(board->size, opponent_bitboard, player_bitboard));
return player_moves - opponent_moves;
```

### 2.1.3 Stability Heuristic

Since the aim is to have maximum discs at the end of the game, the number of your discs that are guaranteed to be there till the end is a good measure of the value of a position. Because these discs, called stable discs, cannot be flipped, they will stay till the very end and necessarily count towards your score.
The `int board_stable(board_t *board, disc_t player)` function calculates an estimated amount of stable discs a player has minus the estimated stable pieces the opponent has to offer. A key observation to make here is that discs placed on corner squares are always stable. Also, any disc on one of the edges of the board can only be flipped by a move along that edge. An edge disc therefore, is also stable if its next to a corner disc or another stable edge disc of the same color. Spinning this theory further, an algorithm came up to calculate the stable pieces.
For that algorithm, all 4 direction (north/south, west/east, nw/se, ne/sw) must be evaluated if a disc had either a border or a already stable disc is either of the two adjacent discs in that direction. Here is an example for the direction up/down.

```
candidates_up = player_bitboard >> shift_up;
candidates_up &=  ~(direction_1 & alread_stable_discs);
candidates_up = candidates_up >> shift down;
candidates_up ^= player;

candidates_down = player_bitboard >> shift_down;
candidates_down &= ~(candidates_down & alread_stable_discs);
candidates_down = candidates_down >> shift_up;
candidates_down ^= player;

stable_up_down = candidates_up | candidates_down;
```

This proces needs to be repeated for all 4 directions. And in the end the stable disc bitboard can be updated.

```
already_stable_discs |=
  (stable_up_down & stable_west_east & stable_nw_se & stable_ne_sw);
```

This algorithm requires the board to remember its stable pieces for each player. This was done by adding two new bitboards to the board_t struct. Annother sideeffect of this implmentation is, that it may need to be run several times to get all stable discs. Here the example for the black player.

```
bitboard_t check = board->stable_black + 1;
while(check != board->stable_black){
  check = board->stable_black;
  /* the void compute_stable_pieces(board_t *board, size_t size,
   * bool is_black) function updated the bitboard board->stable_black */
  compute_stable_pieces(board, board->size, true);
}
```

It's important to notice, that this algorithm is only a negative estimation. It only calculates the stable pieces which are in direct or indirect contact to a corner square, but there are more discs, which can be stable. Stability occurs when a piece is in rows that are completely filled in all four flipping directions. It also occurs when a piece is next to a row of stable squares of its own color in each of the four directions [10]. Here an example.

```
   A B C D E F G H
1  O O X O _ _ _ _
2  O X O _ _ _ _ _
3  O ? O _ _ _ _ _
4  X O O _ _ _ _ _
5  O X _ _ _ _ _ _
6  _ X _ _ _ _ _ _
7  _ X _ _ _ _ _ _
8  O O _ _ _ _ _ _
```

Disc ? in b3 is a white disc and is stable, because in all four directions a row is either full or next to stable piece. But the algorithm above would not detect disc A as stable, hence the negative estimation which only returns a minimum of stable pieces.

### 2.1.4 Disc Evaluation Heuristic

A common practice in Reversi heuristics is, to give certain fields importance. For example there is the A-B-C-X method [8]. The B-squares are in the center of the edge, the C-squares are on the edge next to the corner, and the A-squares lie between the B-squares and C squares. The X-squares are diagonally adjacent to the corners, with the 'X' indicating danger. Given the rules of the game, the only way for your opponent to take a corner is if you play in one of the squares next to a corner, i.e., the C-squares or X-squares. The X-squares are particularly dangerous, and a move to an X-square early in the game is almost certain to give up the adjacent corner. So we want moves, where the player places a disc on a "dangerous" square, to have a negative impact on the heuristic value. Therefor a grid with values for each square was created [6].

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

The `int board_evaluat_discs(board_t *board, disc_t player)` function calculates the value of a current board, by adding the square values of the players positions and subtracting those of the opponent. Done is that trough bitboards. Here an example for the corner squares, each worth 20.

```
bitboard_t corners = 1;
corners = corners << 7;
corners += 1;
corners = corners << 49;
corners += 1;
corners = corners << 7;
corners += 1;
result += (bitboard_popcount(player_bitboard & corners) * 20);
result -= (bitboard_popcount(opponent_bitboard & corners) * 20);
```

This heuristic, contrary to all the other ones, only works on a 8x8 board.

### 2.1.5 Frontier Heuristic

In the Disc Evaluation Heuristic, we learnt about the value of X and B squares. In games between players that are both aware of the strategy, neither player will voluntarily make the sort of bad X-square and C-squares moves that give up corners for no reason. If you want your opponent to make these moves, then you will have to force him to do so. That is, you want to create a situation where the only moves available to your opponent are bad moves.
Frontier discs are defined as discs that border one or more empty squares. Although technically discs on the edge squares could fit this definition, they are not included when speaking of frontier discs. A move which creates many new frontier discs is called a loud move, while a quiet move creates relatively few frontier discs. A wall is a connected group of frontier discs of the same color.
Building a long wall leaves you with nothing to flip, cutting off your access to the squares on the other side of the wall. Meanwhile, the same wall gives your opponent a wide range of choices. Building walls and running out of moves usually go hand-in-hand. In general, this means that quiet moves, which avoid creating a lot of frontier discs, are better than loud moves [8].

The Algorithm checks for every disc, if it has an empty square in one of the eight directions (north, south, west, east, nw, ne, sw, se). Therefore, the player bitboard is shifted in one direction, ANDed with the empty squares and shifted back in the opposite direction(Opposite of north is south...).

```
bitboard_t empty = ~(player_bitboard | opponent_bitboard);
bitboard_t frontiers = 0;
bitboard_t matched;
for each direction d
  matched = empty & (player_bitboard >> shift_d);
  frontiers |= (matched >> shift_opposite_d);
```

In the Reversi game, A lot of frontier discs and walls will lower the probability of a player to win the game. So it would make sense to create a heuristic for walls as well, but this was not done in this implementation.
The `int board_frontiers(board_t *board, disc_t player)` function returns:

```
-(my_frontiers - opponent_frontiers);
```

## 2.2 Weights

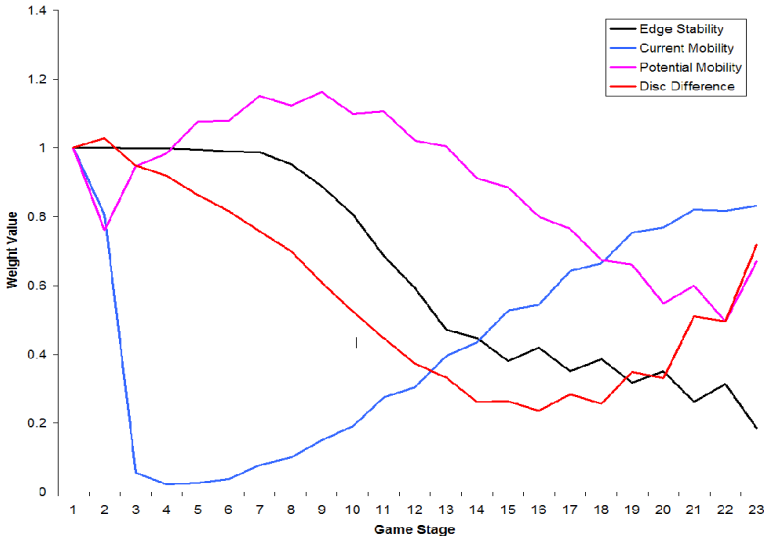
The five different heuristics which are calculated for a specific board need to be weighted relative to their importance in the game. This is the crucial step for the Reversi AI. And also the hardest. It's the hardest, because of several reasons.

1. There is not a lot of research material. From reading through over 50 papers on Reversi AIs, only 2-3 gave an estimation of weights. On the other hand a lot of Reversi guides for human players tell the player on which objective he should focus in which stage of the game. In the AI it was a mixture of Brian Rose's Othello a minute to learn.. A lifetime to master [8], and an Othello Program called Caesar build in 2003 by students [4].

2. In this implementation, the heuristics are not normed, which means, the return values for each heuristic have different maxima's, minimums. For example the score heuristic returns a value between $-BoardSize^2$ and $BoardSize^2$ and the disc evaluation heuristic a value between -360 and 360.

3. To test the different weights an optimal case would be to have a better external Reversi AI to play against, which is still beatable and not deterministic. This way the current weights can play 100 games against the external AI and check how many games were won. After tweaking the weights the process is repeated until the maximum of wins is acquired. This was not possible in this case, because the three opponents to test the AI against (explained in Implementation 3.3) served not as a good test opponent.

Since the Caesar AI split the game up in 23 different stages and has a different weight for each of their 4 heuristic for every Stage as shown in this figure.



This Reversi AI does a similar thing. The game is split up in 4 stages: Early game, Mid game, End game and End end game. Early game taking up the first third of the game, mid game the second third and end game the last third. End end game is only for the last 5 steps, because in those, certain heuristics such as disc evaluation or stability have less importance in comparison to score heuristic.
Here are the values for the different stages.

| | Score | Mobility | Stability | Disc Evaluation | Frontier |
|---|---|---|---|---|---|
| Early Game | 1 | 8 | 20 | 7 | 3 |
| Mid Game | 3 | 2 | 10 | 4 | 3 |
| End Game | 7 | 10 | 2 | 1 | 2 |
| End end Game | 10 | 2 | 0 | 0 | 0 |

## 2.3 End Game Search

In the ending stages of a game, the branching factor gets bounded by the number of empty squares left on the board. As it turns out only a few of these are legal moves for either player - reducing the branching factor even further. Hence, it is possible to search much deeper in the end game than in the opening or the middle game. The goal was to make the Reversi AI a perfect end game player. The result is that it should plays a perfect game 9 suaqres from the end of the game. This was never really tested, it the game is perfect then, just assumed.
Done is that by counting the empty squares on the board, and if the are less or equal to 9, they are givin as the depth parameter to the minimax tree search function.

# 3 Implementation

In this section a hand full of function and extra code will be explained, which helped building the final product.

## 3.1 Bitboard Approach and Performance

A tradeoff must be made between playing fast and playing well. Since the only factor that the computer player uses to choose a move is the backed-up score generated after searching a given number of moves, it can make a better move if it searches further in the future of the game and has more information on which to base a decision. The faster score evaluation is at any level, the deeper a search will be possible. That's why the fast bitboard were chosen to store the game boards. Bitwise operations on a bitboard to get the next possible moves is undeniable faster than going trough a couple of while loops and evaluating bidimensional arrays. More on the topic in "Bitboard methods for games" [2]. Functions like board_stable or board_frontiers, which calculate a certain type of disc on the board rely only on bitwise operations making them fast and efficient.

## 3.2 Time

Time plays a major role when implementing a Reversi AI. Human players don't want to wait minutes until a move is calculated and when playing against ther AIs there is often a time constraint. In this project a Time constraint for 30 seconds per move calculation was given. To stay within the bounds of the time constraint the recursive minimax tree search needed an extra exit criterium.
This was done by defining a MAX_TIME value in the header and checking if the time has come to exit. The variable timer was initialized with `time_t timer = time(NULL)` at the first call of the minimax function and given as a parameter to the minimax_help function.

```
minimax_help(board_t *board, size_t depth,..., time_t timer){
  int value = final_heuristic(board, player);
  if(depth == 0 || (time(NULL) - timer) >= MAX_TIME){
    return value;
  }
  ...
}
```

It is not intended, that this happens, which is why an appropriate depth is needed. Here is a Mid Game board which has 13 possible moves for black. One of the most time-consuming moves to calculate in the whole game.

```
   A B C D E F G H
1  O _ _ _ _ _ _ _
2  _ O _ _ * _ _ _
3  _ * O X O * _ _
4  _ * O O X * O _
5  _ * O X X O * _
6  _ * O X O X * _
7  _ * O O O * _ _
8  _ * _ * * _ _ _

'X' players turn
```

Here are the seconds it takes with different depths to calculate the best move.

| Depth | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Time (in sec.) | 0 | 0 | 0 | 1 | 8 | 94 |

A standard depth of 5 was chosen.

## 3.3 Opponent AIs

To test the progressively getting better AI, especially in regard to weights, a worthy opponent was needed. A key factor of this opponent was that it's play style was not deterministic, because then every game looks exactly the same. And testing of how many wins you get over 100 games was either 0 or 100. So, throughout implementation 3 different opponents played against the final product.

### 3.3.1 Random

The random_player function implemented in homework 4 returns a random move among the possible ones. It was the first contrahend and actually performed better than the simple score heuristic. But because it's to random nature, it wasn't quite fit for meaningful testing. You can never be sure if one AI is really better than another one, just because it player better against the completely random player. And still, the final product loses 1 out of every 70 games to the random player.

### 3.3.2 Better Random

The so called better random player evaluates the score, and any other given heuristic for every possible move. It then choses randomly one of the two with the best values. It basically performs a minimax tree search with depth 0 and a random element at the end. This served as a great adversary until the stable and disc evaluation heuristic were implemented. Then it would lose 100 out of 100 times, and it was impossible to emend the weights, because the "better random" player was just too bad. The reason why the better random player performed worse against the AI is, because the minimax tree search expects the opponent to pick a good move. And while the better random player doesn't always pick the absolute best move, it doesn't pick the worst one either like the total random player might.

### 3.3.3 Random Start

The random start player was the last try to get a working opponent which would not be completely random or to bad to play against. Its concept is to start the game with a couple random moves and the after a specific time turn into a good AI. With this method weight calibrations were tested. If a heuristic weight was changed, the AI with new weights would play against the random start one with the old weight values, to check if it would win significantly more games. Unfortunately, this did not turn out the way it was intended, and no massive improvement was made trying to calibrate the weights.

# 4 Extra Efforts

The priority of this project lied admittedly on the performance and getting the algorithms to work. With more time, more effort could be made towards code quality or modularity.

## 4.1 Code Quality

That doesn't mean no efforts were made in view of code quality. After all the code has to run bug free, produce no memory leak or corruption and stand up to the code styling guidelines.

#### GDB

GDB (Gnu Debugger) is a powerful debugging tool for both C and C++. A debugger is a program that helps you find bugs by letting you examine the state of the program as it's running. Its way more powerful than used in this project. Its sole use case was to check where the segmentation fault takes place in the code, if one occurred. In hindsight a couple of time a debugger would have been more powerful than primitive debugging print statements, and in future GDB will be a bigger priority. But in this project a lot of the development time was spent on paper coming up or understanding the algorithms. Once they were understood the implementation ran rather smoothly.

#### Valgrind

Valgrind is a multipurpose code profiling and memory debugging tool for the programing language C in Linux. In this project it was used to seek out any memory leaks or corruptions.
The command `valgrind -leak-check=full -track-origins_yes ./reversi` and trying out every thinkable game scenario, the memory handling should be on point.

#### clang-tidy

clang-tidy is a clang-based C/C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations,

interface misuse, or bugs that can be deduced via static analysis. It was supposed to fix all style guideline violations in this project. For example, adding a space in front of else statements if there is none between the "}" and the else. Unfortunately, the documentation was too complicated/ reader unfriendly to make out what to change in the style section to apply it to the right guidelines that were given for this project. With more time to get the installation and set up right this would have saved time which was now used to search through the code manually. It really sounds like a good tool and in future projects this will for sure find its use, just for this project it was too late.

## 4.2 Performance

Several small performance improvements have been made. The performance was tested one and the same mid game board with the <time.h> header as a measurement. A `time_t timer = time(NULL)` was initialized before the minimax tree search call and `(timer - time(NULL))` was printed out after the move was calculated to check how long (in seconds) it took.
One example of a performance improvement, which saved several seconds on a depth=6 tree search was to check if stable discs are possible in this stage of the game. Before calculating how many stable discs a player has in comparison to his opponent, a check is stable discs are possible is executed. The way the stability heuristic 2.1.3 is implemented, no stable piece can be found before a corner square is occupied. So the `bool check_stable_is_possible(board_t *board)` function checks if either player has a disc in at least one of the corners.

```
bool check_stable_is_possible(board_t *board){
  return (bitboard_popcount((board->white | board->black) &
    stable_check) > 0);
}
```

The bitboard stable_check is a global variable and gets initialized once at the beginning of the game.

```
bitboard stable_check = 1;
stable_check = stable_check << (board->size - 1);
stable_check += 1;
stable_check = stable_check << (((board->size - 2) * board->size) + 1);
stable_check += 1;
stable_check = stable_check << (board->size - 1);
stable_check += 1;
```

# 5 Conclusion

Reversi is an amazing game to learn game playing artificial. The primary reason being the small branching factor, allowing the computer to look ahead in abundance, thrashing human intuition and reasoning. This leads to an extending gap in the Othello-playing ability of humans and computers. After implementing only a few basic heuristics, me as the developer stood no chance against my own creation. That was an amazing feeling.
It was a lot of fun implementing the game in a C - a new language for me. Bitboards as a data structure and being concerned about memory leaks were also new to me. My favorit part was adding personal code to improve performance. There is a lot which I would like to add the project with more time. For example, a better calculation of the heuristic weights. That point hunts me the most. I know that the AI could have probably been even better with slight adjustments to the heuristic weights, but no opponent I implemented could help improve the values.
Also to this report I would have liked to add more segments, but it is already too long. Just to name a few:

1. The additional data structures like game_stage enumerations or global and internal bitboards.

2. The tweak to the minimax algorithm to give winning a higher value, but dependent on how much depth is left an even higher value.

3. Adding graphical visualization to the heuristic algorithm explanation chapter.

All in all I had a really good time on this project and learned a lot for my future as a developer.

# Bibliography


[1] K. J. Binkley, K. Seehart, and M. Hagiwara. "A Study of Artificial Neural Network Architectures for Othello Evaluation Functions." In: *Information and Media Technologies* 2.4 (2007), pp. 1129–1139. DOI: 10.11185/imt.2.1129.

[2] C. B. Browne. "Bitboard methods for games." In: *ICGA Journal* 37.2 (2014), pp. 67–84.

[3] D. Currie. *Othello Game Player*. 1997. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.103.494 (visited on 11/20/2019).

[4] P. S. Deshwal. "Caesar-An Othello Program." In: 2003.

[5] C. Frankland and N. Pillay. "Evolving Game Playing Strategies for Othello Incorporating Reinforcement Learning and Mobility." In: *Proceedings of the 2015 Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*. SAICSIT '15. Stellenbosch, South Africa: ACM, 2015, 16:1–16:9. ISBN: 978-1-4503-3683-3. DOI: 10.1145/2815782.2815809.

[6] K. Kukreja. *Heuristic/Evaluation Function for Reversi/Othello*. 2013. URL: https://kartikkukreja.wordpress.com/2013/03/30/heuristic-function-for-reversiothello/ (visited on 11/20/2019).

[7] K.-F. Lee and S. Mahajan. "The development of a world class Othello program." In: *Artificial Intelligence* 43.1 (1990), pp. 21–36. ISSN: 0004-3702.

[8] B. Rose. *Othello - A Minute to Learn... A Lifetime to Master*. URL: http://www.ffothello.org/livres/othello-book-Brian-Rose.pdf (visited on 11/20/2019).

[9] V. Sannidhanam and M. Annamalai. *An Analysis of Heuristics in Othello*. URL: https://de.scribd.com/document/353439817/An-Analysis-of-Heuristics-in-Othello (visited on 11/20/2019).

[10] A. Wierman. *Othello implementation*. URL: http://pressibus.org/ataxx/autre/minimax/node3.html (visited on 11/20/2019).