# CMPT 310 Final Assignment Report

## By: Denzel Tjokroardi / 301375072

The project I for this assignment to create a Reversi game with an added AI component where it is done through two ways:

1. Pure Monte-Carlo Tree Search Algorithm
2. Monte-Carlo Tree Search Algorithm with added Heuristics

The game is coded in Java 8 using the Intellij IDEA IDE. If played by the user, the game will be played through the command-line where the user will input 2 coordinates by help of the labeled gameboard. The game itself is quite straight forward, it contains a “checker” to check if the inputted move is valid. Then, if the move is valid, it will call another function to “flip” all of the other pieces between the inputted move and the target.

**Pure Monte-Carlo Tree Search Algorithm**

The Monte-Carlo Tree Search Algorithm (MCTS) was implemented by means of a separate class. The class takes in the gameboard class with stores the 2d array. Then, it calls a helper method which will store into a list, all of the possible moves from the 2d array. The list will then be passed onto another method called “getBestMove” which will return the coordinate of the best “move”. This method will first create a Hash-map with the possible moves as keys and an array with 3 slots as its value. This array will store the number of Wins, Draws, and Losses of the random testing. Once the Hash-map is created this method will call another method called “simulate” which basically simulates a playthrough of the game by creating a copy of the game and actually running through the full game. This simulation is done 1000 times, for each possible move. Once all simulations are done, we go back to the previous method where it will process the Win, Loss, and Draw data by finding the maximum difference between the Win + Draw – Loss. The move that corresponds to this maximum is then returned by this function to the real board and is set as the input of the AI.

**Monte-Carlo Tree Search Algorithm with added Heuristics**

The MCTS algorithm with heuristics is essentially the same except that the “getBestMove” method contains a few added functionalities. The method used to create a bias towards a certain move is both by returning a coordinate straight away if it is deemed valuable enough, or by putting a “weight” on it by adding/ subtracting the Win + Draw – Loss value by a certain amount if the coordinate is encountered within the list of possible moves. The heuristics used are as follows:

1. Corner Coordinates
   1. If a coordinate in the corner is available to be filled, it will be filled straight away without any data manipulation.
2. Coordinates close to the corners
   1. Coordinates close to the corners are dangerous due to the fact that they may provide a way for the opponent to fill the corner. Therefore, in the early-mid game, these coordinates are weighted to decrease the Win + Draw – Loss value.
3. Coordinates at the edge
   1. Same as the above, these coordinates can provide a gateway for the opponent to secure the corner, to a lesser extent, therefore it is also weighted to decrease the Win + Draw - Loss value up until the mid-game.
4. Mobility Heuristic
   1. This heuristic puts a positive weight (addition) to the Win + Draw – Loss value if the coordinate in question minimizes the opponents possible move in the next round which increases the possibility of securing corners.

When pitted against each other, the 2 algorithms curiously are pretty even. The game was run 10 times with the results tabulated in the graphs below:

*Table 1. Average time each algorithm took to finish*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| MCTS | 381 | 389 | 357 | 358 | 410 | 401 | 318 | 414 | 443 | 339 |
| MCTSH | 325 | 355 | 242 | 347 | 261 | 353 | 287 | 376 | 349 | 246 |

A screenshot of a social media post

Description automatically generated

*Graph 1. Average time each algorithm took to finish*

*Table 2. Number of chips each algorithm had at the end of the game*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| MCTS | 42 | 32 | 48 | 18 | 29 | 36 | 19 | 38 | 16 | 35 |
| MCTSH | 22 | 32 | 16 | 46 | 35 | 28 | 45 | 26 | 48 | 29 |
| Winner | MCTS | Draw | MCTS | MCTSH | MCTSH | MCTS | MCTSH | MCTS | MCTSH | MCTS |

A screenshot of a social media post

Description automatically generated

*Graph 2. Number of chips each algorithm had at the end of the game*

As seen from the data above, Table and Graph 1 demonstrates that the MCTS implementation with added Heuristics is generally faster which makes sense since heuristics are essentially short cuts which allow for the algorithm to cut off some of its processing time whenever it knows that an answer is right. However, the data in Table and Graph 2 makes less sense. In theory, the MCTS with Heuristics should be winning more than half of the games however, here it is actually winning less. After analysis of my implementation, I concluded that it is possible that my weights as well as the timing of when to put the weights are not finetuned enough, hence while providing allowing for the algorithm to have an easier time making a decision, I end up causing the algorithm to possible take the less than ideal solution, which is worse off than just random answers.