Softwarepraktikum SS 2025

Assignment 2

Group 6

Henri Schaudinn	456738	henri.schaudinn@rwth-aachen.de
Malte Ewald	456139	Malte.Ewald@rwth-aachen.de
Frithjof Alms	456247	frithjof.alms@rwth-aachen.de

Contents

Work Distribution

It is difficult to make exact distinctions on which task was completed by what group member, as there were always several group members working on different small parts of the whole task. Frithjof focused mainly on task 1 and 2, implementing and testing the map updates and the full gameplay code. Malte especially helped with the processing of special fields and with the creation and integration of several classical heuristics. Henri worked on the loop detection, implemented the other heuristics and analyzed the coin experiment of task 4. The corresponding parts of the presentation and reports are written by the group member who worked on the parts of the assignment.

Task 1

2.1 The maps new data structure

To efficiently save and calculate the successor states, we split the array, which contained a tile object for each tile on the map, into two separate objects. The static component of this new implementation is the "Neighbors" array, which stores an int array of length 8 for every tile. The entries of these arrays are of the form dxxyy, with x and y representing the corresponding coordinates and d being the direction the tile has to be left when entering from direction i, which is the index of the array. A value of -1 indicates that no valid tile is neighboring the selected tile in direction i. These transitions remain identical throughout the game, so there is no need to safe them more than once. The current position is tracked using a hashmap, which stores the owner number for every tile using the key xxyy. Keys from -1 to -16 are used to store all players' bombs and overwrites.

2.2 Move execution Build Phase

To execute a received move, it has to be split into parts, the x- and y-coordinates and the special number. We used a switch case to cover all possible types of tiles a player can place his stone. The first step is to change the owner number in the hashmap for given x- and y- coordinates. Secondly, all captured stones are transferred to the capturing player. Lastly, all special tiles have to be considered, if the player uses an overwrite stone, the corresponding hashmap entry is decreased, analogously, if the player captures a choice tile, the entry is increased. In addition to that, the invert and swap tiles need to be processed, this is achieved by iterating all hashmap entries using two for-loops and switching the owner numbers.

2.3 The replaceStones function

This proved to be the most difficult and time consuming part of task 1, as there are many special cases to be considered and debugging appeared to be quite challenging. This function tries to find connecting paths (meaning a path following only one direction and only crossing capturable stones) starting from the x- and y-coordinates of the placed stone, to another stone of the same player. For that, it calls the tileWalk function, using the coordinates and each direction as parameters. In order to find the correct neighbors, this function iterates through the tiles corresponding neighbor array entry and, in case of a special transition, changes to the stored direction. For every tile the function passes, it checks if the tile can be captured, and if the first tile, which cannot be captured, is owned by the player,

the length of the path is returned. The tileWalkSwap function then replaces all tiles reachable in length steps for each computed length in each direction.

2.4 Loop detection

We came up with several ideas to prevent the tileWalk function from being stuck in a loop. The first Idea, the simplest approach, just increased a counter for every visited tile. If the counter exceeded $8 * t$, where t is the total amount of tiles, the tileWalk function must be stuck in a loop, because each tile was already visited from every direction, and the next tile the function entered was already visited from the same direction. In case of a loop, this approach performs very poorly, as a small loop may be run through many times before its detection. Nonetheless, this approach offers a good performance on maps where few to none loops occur. The second idea involves saving all combinations of visited tiles and entering directions in an ArrayList, while checking for each step if the tile has been entered from the same direction before. In contrast to the first approach a loop is hereby immediately detected, but checking all list entries for matches takes more effort than incrementing a counter. Assuming a loop always enters the starting tile twice from the same direction, the third approach is based on testing if such a recurrence happened in the path traveled so far. We settled for this approach, as it detects a loop as fast as approach 2, and needs as many comparisons as approach 1.

2.5 Move execution Elimination Phase

To find all tiles reachable in n steps, where n is the bomb strength, a counter variable can be used. It starts at 0 and runs to 10^n. For each value the counter takes, the function traverses n steps from the starting point according to the numbers digits. The digits 0 and 9 indicate that the tile is not switched, a number between 1 and 8 indicates the direction the next tile neighbors the current tile. The owner number of the last reached tile is set to 10, encoding a hole. This function is quite ineffective, as it takes exponential steps (compared to the bomb strength) to fill a quadratic area with holes.

Task 2

3.1 Basic implementation

The implementation of this task was very simple, as we already had the necessary functions after implementing task 1. The servers move request triggers the makeMove function (from assignment 1), which finds a valid move for a given game state, and the found move is returned to the server. The move announcement triggers the moveMade function, which calls the map updating function from task 1. The updated map is stored in the hashmap object in the MapModell class.

3.2 Further ideas

As one can notice, most of the MapModells and ReversiGames functions take two hashmaps as parameters; one hashmap represents the current position of the map, and the other is edited during the execution of the function. Furthermore, the functions calculateMask and calulateNextMask provide a way to calculate mask hashmaps for a given map position, meaning the mask only contains (value, key) pairs for given tiles. This could be of use evaluating different positions using heuristics, as for each move only a small part of the whole map has to be stored. After "laying" the mask onto the current map position, further calculations can be made.

Task 3

4.1 "Classical" heuristics

4.1.1 Basic Ideas

When we started thinking about which heuristics to use for grading our moves, we first ran into the question "Which factors predict a victory in Reversi?". On first thought, this is an easy question, as the biggest predictor of a game which is won on points is usually the amount of points relative to your opponent. This is the case in sports like football or tennis, for example. In Reversi this is not the case as the strength of the moves your opponent can play scales with the number of stones/points you have.

This causes the importance of having more stones than your opponent to wane, especially in the early part of the game. Of course, this does not mean it is irrelevant. But if the number of stones is not a good heuristic, what is it?

Our next thought was to think about how to increase our number of stones without increasing the strength of our opponent's moves. This can be achieved through the corners of the board, as stones on the corners of the board can not be captured. As it turns out, the stones on the corners of the board are a strong predictor of winning or losing a game of Reversi, as can be seen in the success of our heuristic using this.

As the corners have the advantage of being impossible to capture and predict victory, we concluded that the borders of the board are also more important than the center of the board. Pieces in the middle of the board can be attacked from 8 sides, while border stones can only be attacked from two sides. This should result in a result similar, to but much weaker than the cornerstones.

4.1.2 Our Implementations

We implemented three heuristics and their delta applications for multiplayer games. 1. biggest stone count 2. smallest stones count 3. corners worth 5, border worth 1

4.1.3 Results standard map

time + us	**most stones**	**least stones**	**first**	**Corners+Borders**
5s P1	32 / 32	38 / 26	28 / 36	38 / 26
5s P2	38 / 26	50 / 14	9 / 55	32 / 32
2s P1	36 / 28	38 / 26	28 / 36	38 / 26
2s P2	32 / 32	13 / 51	50 / 14	5 / 59

4.1.4 Interpretation

As I will elaborate on in 3.4 the significance of the results leaves a lot to be desired. But we can see the trend that "corners +borders" gets the most stable result. As it does not lose, whereas the other heuristics encounter losses against the rather weak AI. However, during testing, we realized that getting border tiles is not always optimal. As border tiles can open up the opportunity for your opponent to capture the corners, if a stone is placed directly adjacent to a cornerstone. This idea will be revisited when we have changed our heuristic accordingly.

4.1.5 Problems with our results

There are two big problems concerning the results of testing our heuristics against the standard AI provided.

Problem number 1: as it turns out, the outcome of using a not random heuristic against the standard AI results in the same game. So tests are deterministic if the time limit is similar. If the time limit is changed, the result also usually changes. This causes the scope of testing to be very small(in this case four tests per heuristic). This renders the outcome statistically irrelevant. But this can be amended by using Matchpoint to test our Algorithms. This will be done in the future when the impact of special spaces on our heuristics is explored and implemented. Problem number 2: The strength of the standard AI is too low to generate good results, as it sometimes loses against the implementation choosing the first move found Algorithm. This causes the variance in AI play sequences to interfere with our testing as well.

In conclusion, our preliminary tests on the standard map can be used to see trends, but not in any way shape, or form to make conclusions.

4.1.6 Problems with special rules

Currently, our heuristics are very vulnerable to swap and inversion stones, as they invert the significance of "having a good position". So in some cases, it can be better to make "worse" moves for your position if inversions/swaps are in the game. This is a subject still up for discussion in our group.

Another problem is bombs, as owning cornerstones, cause your stones to cluster around them as they "strengthen" stones close to them. This makes us more vulnerable to bombs later.

4.2 Other heuristics

4.2.1 Item heuristic

This heuristic is simple in nature but tackles a complex issue in our version of reversi. Bonus tiles present the player with an important choice: Receive a bomb or receive an overwrite stone. A bomb has a lot of "absolute" impact due to its radius, for example with a bomb strength of 2, 25 individual tiles are "bombed". When using an overwrite stone, even when including the captured tiles on a potential capture, the absolute impact is usually lower. Instead overwrite stones provide "compounding" impact by enabling new moves that would not have been possible before; therefore, when there usually would not have been a possible move, they keep the player in the game and therefore play a part in all subsequent moves. Due to this, it is important to always have overwrite stones to keep you alive, but always chose bombs over overwrite stones that are not needed. We achieved this by rating overwrite stones more highly when we have fewer of them, ergo need more to be safe. In addition to that we value the bomb depending on its strength since the bomb strength directly influences the bomb's absolute impact.

4.2.2 Neighbour heuristic

This heuristic aims to deny the opposition's resources by making it harder to capture the tiles choice, inversion and bonus. To capture such a tile, there needs to be a "capturale" tile next to it, meaning that there needs to be a player- or expansion-tile around it. We can prevent this by not placing a stone next to a "special" tile. The heuristic implements this strategy by accumulating a rating by iterating through all neighbours and scoring them based on whether we want them to be our neighbours or not. Therefore we return high scores if the neighbours are not "special" tiles and low scores if they are.

Task 4

5.1 Possible pitfalls

While it may seem simple at first, the question of what happens when a fair coin is thrown many times can lead to wrong conclusions: Intuitively, one might think that the more we toss the coin, the closer we will get to the expected value, but this is not the truth.

5.2 Theoretical approach and Hypothesis

Before diving into the plots, it is important to form a hypothesis as well as get an understanding of the different parameters. The expected value of a fair coin toss with 1 for heads and -1 for tails is zero.

$$\mathbb{E}[X] = \sum_x x \cdot P(X = x) = 0.5 \cdot 1 + 0.5 \cdot (-1) = 0$$

And via linearity of the mean, it follows that the Sum over n means is just the mean of the sum:

$$\sum_i^n \mathbb{E}[X_i] = n \cdot 0 = \mathbb{E}[\sum_i^n X_i]$$

So the mean of the random experiment that tosses n coins is 0.

$$Var(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2 = \mathbb{E}[X^2] = 0.5 \cdot 1^2 + 0.5 \cdot (-1)^2 = 1$$

Therefore, the variance of 1 coin toss is 1. Analogous to the expected value, we can now derive the variance of the n independent coin tosses using linearity:

$$\sum_i^n Var(X_i) = n \cdot 1 = Var(\sum_i^n X_i)$$

So the Variance of n coin tosses is n; this means that it is to be expected that the higher our n becomes, the further we will be away from the mean (on average).

5.3 Modeling the Experiment

Now that we know what to expect, let us see if our hypothesis is correct: To model the n coin tosses, we implemented a plot displaying the cumulative value derived by the random increments or decrements caused by heads or tails. The corresponding figure with n = 10000 can be seen above. We can already observe that the hypothesis seems to hold, but 3 graphs don't tell the whole story, there is to much randomness for this depiction to appropriately verify our hypothesis,

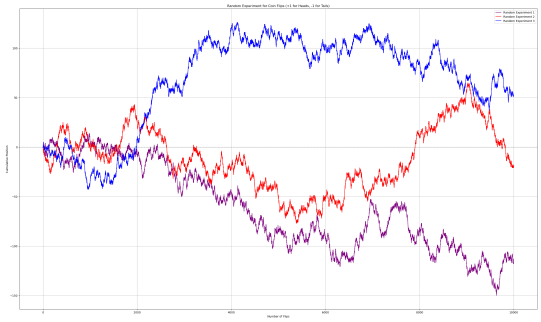

Figure 1: This Figure models 3 different instances of the same random experiment

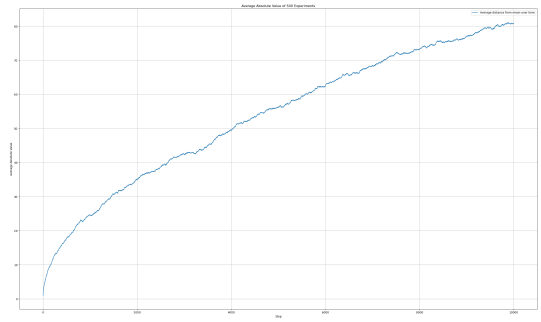

Figure 2: This Figure displays the average Absolute distance from the mean over 500 random experiments

so we need a different graph; Since more plots would hurt visibility we decided to plot the average distance over the mean over 500 plots keeping the n at 10000, this way we can see what is happening more clearly. As expected, the average distance from the mean increases with the number of coin tosses. But what does this mean? While maybe if we were betting money on the coin toss, we would want to continue after a streak of bad luck because the odds are even, there is no reason for us to expect that with more tosses we will get our money back, since the cumulative value does not converge to the expected value over time.