Softwarepraktikum SS 2025

Assignment 1

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

Task 1

1.1 Build-tool

Since we chose Java as the programming language for our project, Gradle was the obvious choice to use in need of a build tool. Furthermore, the Gradle Java Plugin came in handy, offering a lot of predefined tasks for working with Java projects, especially the "jar" task, able to construct a jar file of any given Java project. For the jar file to work in an offline environment, we also have to ensure that the file has no external dependencies. The shadow plugin provides a solution to this problem, the task "shadowJar" creates a so-called Fat JAR, which includes all external dependencies needed to run the program. Now we need to make sure that the "shadowJar" task is run after issuing the Gradle assemble command, and we need a cleanUp task deleting all compiled files created through the "jar" task. This is achieved using the dependsOn definition in Gradle, declaring that the assemble task depends on the cleanUp task, which again depends on the shadowJar task, assuring the tasks are called in the correct sequence. Manually setting the source and destination addresses allows us to store the complete jar file in the bin folder.

1.2 gitignore

The gitignore file provides useful mechanisms to prevent certain folders and files from being pushed onto the repositories. This is especially useful considering the gradle assemble task creates .class files, a .gradle directory and a build directory, which should not be on the git repository. By naming *.class, .gradle and build in the .gitignore, git excludes these files when pushing.

Task 2

2.1 introduction

In the binary repository we found 4 executables, which can be run in the console. I will make this part short as the learnings were quite trivial.

2.2 server

this command hosts a server on localhost 7777. mandatory Flags are -t x, where x is the time in seconds the "players" have to send their move. The other mandatory flag is -s/mapname specifying which map to run -s being the standard map. Other arguments are rather irrelevant to the current state of the project. This binary always has to be started first.

2.3 ai

This command runs an ai agent, which connects to the server as a "player" sending moves, when its turn starts. There are no relevant flags at this point in time

2.4 client

This command can be used to manually connect to the server and play reversi. This can be used to play against your own ai or the ai provided.

2.5 mapinfo

this command can be used to validate .map files, as well as give some interesting statistics about the map.

Task 3

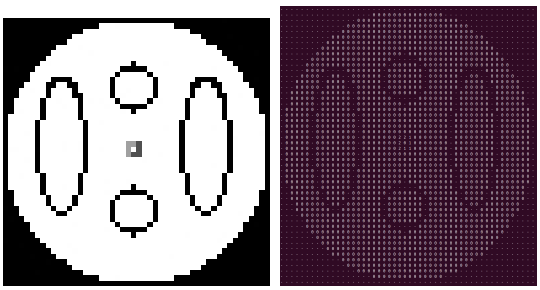
3.1 The Idea

After exploring the .map format, we learned that creating large and complex maps by typing in individual characters would be tedious and make quality control hard. Our solution was to automate the process of map creation and limit the human input to a minimum, This was achieved through creating a Python script that leverages image processing techniques to translate any image of appropriate size into a map, this not only makes for a more streamlined and simple workflow when creating (now drawing) maps but also enables for intricate pixel art to be turned into stunning maps with ease.

3.2 How we did it

For this script, we chose to differ from Java, which we chose for our codebase, and instead develop the map Generator as its separate entity. This design decision was made in part because Python is just better suited for image processing than Java, but also because the independence of the Generator enables more widespread usage independent from this specific project to increase ease of use, not just for us but also for future map creators. We came across the second fork in the road when having to decide on how to convert color values to characters; When choosing to map a subset of all possible intensity values to the needed characters, we enable a straightforward map creation through image manipulation software, but we lose the ability to convert pre-existing images into maps. The option we chose was to create a many-to-one mapping by just rounding all values that are not in the subset of 1 to 1 correspondence to their nearest neighbour that is in the subset.

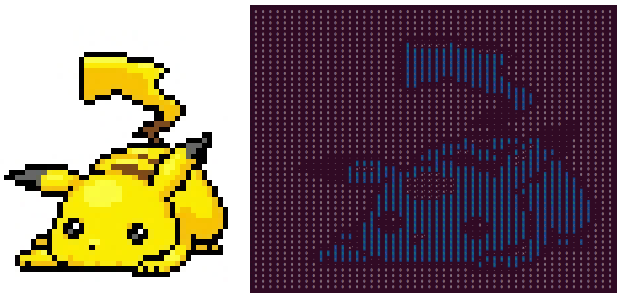
3.3 Circles



This map (input and output picture displayed above), as the name suggests, was all about circles; the arena is one big circle, and within that circle, we find more circles and ellipses made of holes. This map does a great job of showing of

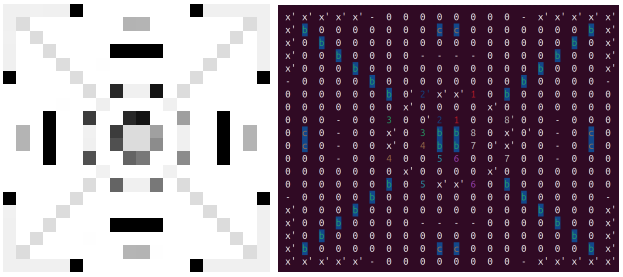
the improvements the map Generator made to the map creation workflow; Whilest previously the creation of perfect circles was very tedious, if not impossible, now all one has to do is go into some image editor and use its tools to create whatever geometric shapes they desire. Other than that, this map aims to challenge the basic reversi ability of the clients while at the same time eliminating corners, which in the original reversi seem like an overpowered position to be in since they can not be captured. Additionally the Map is kept simple by renouncing the usage of extra tiles and only implementing added transitions that are directionally straight.

3.4 Pikatchu



This map (input and output picture displayed above) goes all out and embodies pure chaos. It was generated with a picture from the web, and even though there were no manual adjustments apart from giving input numbers to the algorithm (number of players, bombs, etc.), it remains perfectly playable and makes for a unique challenge with unique visuals.

3.5 Competetive



This map (input and output picture displayed above), in contrast to previous endeavors, leverages the Map Generator to create a competitive and fair map through image creation by hand. This Map is aimed at catering towards a tournament setting and therefore enables 8 players at a time to go head to head whilst also implementing multiple "portals", a lot of extra tiles, and interesting dilemmas that challenge the competitor's decision-making ability on more than just search depth. For example the first move from player one already makes for three different options: firstly you can capture in the

center ring and fight for dominance there, secondly, you can capture on the second ring, and last but not least, you can also opt to take one of the corners. Ironically I think that most AIs will end up capturing on the first or second ring, whilst I think that claiming the corners is the best move.

Task 4

4.1 simple Model

When we started thinking about the data structure our first concern was simplicity to implement and use and modularity. While Efficiency took a secondary role. This idea was based on the modular nature of this data structure and any Algorithms implemented using this "visual structure" can later be used on a less visual but more efficient implementation.

In the later parts of the projects, we will have to take a closer look at parameters such as memory efficiency, computational efficiency and synergy with our algorithms to ascertain the optimal structure.

Our Data types will be split in two categories: 1. the model, which purpose is to save the data transmitted to us by the server in an abstract form. 2. The crutches for our Algorithms. They do not contain information, which cannot be ascertained from the model, but save recomputational effort for our algorithms.

4.2 Data Model

Our first implementation contains two classes used in modeling: Tile and mapModell.

mapModell represents the abstract model containing all the Information provided to us by the server. We save the basic numeral Data (player number, bombs, etc.) in simple integer attributes of the mapModell object. For the number of bombs and overrides, that are relevant to track relative to each player, we will in a later implementation roll out different data types, probably integer arrays, when/if our algorithm would benefit from the information about other players bombs and overrides. Currently, we will only be tracking our counts in the integers mentioned.

The abstract Data Structure of the board is currently a Tile matrix [height][width], in which we save an object from our Tile class for each "Tile" on the board.

Tile currently contains an integer array [] storing the "special" transitions concerning this tile. The array index of a transition is relative to the direction -> number encoding also used in game rules. The other attributes are integer owner (0 is no Owner) and boolean isTile(false if "-"). This is still subject to discussion at this point in the project. Our first upgrade will be eliminating isTile and changing owner to an integer status. Status would encode owner, if there is an owner, ex. (1...8), 0 if no owner and not special, (10,11,12,13) for (c, i,b, x) and 15 for ("-"). As after an owner is assigned to the special tiles where it is possible, the significance of the tile being special is lost. Unless an Override would still trigger the effect, but after consulting the rules, we found this not to be the case.

4.3 Data types as crutches

Our first idea was to use an integer list of myTiles, to collect the xy value pairs (encoded with integer code=100*x+y) for the mapModel matrix, as the only significant tiles for moves are connected to at least one of these Tiles (x is seen as one of those).

We also added an integer list for opponentTiles, for the inverted approach as myTiles, but this avenue of looking up possible moves seems to be less efficient at this point in time. As for more than two players the cardinality of myTiles would be shorter than opponentTiles, so we would have fewer iterations -> more efficiency. This will probably lead to a discontinuation of this string of thought in favor of the myTiles approach.

Last, we combined the Ideas into a "relevant" list, which contains only opposing tiles that neighbor myTiles.Whenever a move is made, we would check the implications of that move and change the list accordingly. This will decrease the number of iterations, especially in the later part of the game. This is, at this point in time, the most promising idea.

4.4 Vision for the future of Data types in this project

As mentioned in simple Model, the idea behind our current implementation mainly focuses on simplicity, opposed to efficiency. Our future prospects for the project will probably need improvements in this area. Because of this we already started thinking of possible impactful improvements to our Data structure. Three of those ideas are mentioned here.

4.5 increase memory efficiency

Our idea here is to eliminate the need for the Tile class by encoding the information from the tile object into an integer.

We encode the board in an integer Matrix[height][width], where we would encode all relevant info contained in a Tile Object into an integer. We would use 4bit for type and the other 28bit to encode the special transitions, starting from direction 0 to direction 8 encoding adjacent Tiles xy coordinates. Problem: max sizes of x and y using a 50x50 Matrix would be 50 -> 6 bit integers x+y=12bit *8 directions 96+4bit= 100bit would be needed. This means we would need another data type, bigger than integer.

advantage: save memory disadvantage: complexity, integer does not work Also, the next idea, if implemented, would make the impact here rather small.

4.6 increase efficiency of move finding Algorithm

In our current implementation our move finding Algorithm always has to check every Tile, if a move there is possible. This has been to a certain extent already adressed in simple model crutches. But we have a possibly more promising approach, which I will present here.

Saving possible moves in a List of moves, changing the list whenever a move is made. Would reduce the overhead for finding all possible moves for the next move, as changing the list for the new stones placed is on average less computational effort than iterating either through relevantStones or MyStones. This is still a working theory, so we still need to test whether the proposed improvement to efficiency would manifest itself in reality.

4.7 experimental Ideas for later deep search memory efficiency

In the later part of the project, we will need to perform deep searches for the optimal move. This would need the whole Matrix to be saved for every iteration. Our proposition would be to save the move made not in a new mapModel, but rather as some form of delta between the last state and the state now. This would probably decrease efficiency by a little as computing possible moves etc. would be more efficient with the whole matrix. But we think the efficiency/memory tradeoff will be worth it. However this idea is still in its infancy, so there is still a lot to be discussed. For example between how many move delta saved moves we would need a checkpoint move saved as a real matrix.

4.8 conclusion

So in conclusion there is still a lot of work to be done on the Data structure, but as we can change it in a modular way we should first find out the constraints of our Algorithms and alter our Data structure accordingly.

Task 5

5.1 The challenge

Task 5 presented a unique challenge. First of all, it required the previously developed data structure to work well for it to function. This not only presented planning challenges with the individual Tasks' deadlines but also required us to collaborate closely on making everything work as a unit. This didn't always work, and we ran into issues when not all of us were on the same page as to how exactly the data structure works. Thankfully, we managed to work together and figure out a solution in a true group effort.

5.2 The solution

The basic idea of our solution is to first get rid of all the "simple" cases, which rule out a move as invalid and do not require checking for a capture. After that, all we have to check for is whether a capture is happening, which is the same for overwrite stones and normal stones, since we are going from the newly placed stone to an old stone. A capture is happening when the placed stone starts a path involving more than two tiles with another stone of the same player marking the end of the path, apart from the two stones of the player, all other stones should be of different players or expansion stones. By this logic we looped through every single direction and followed the path in that direction until either we arrived at a valid path leading to capture and therefore a valid move, or the path became invalid (for example, due to an empty tile before the second player-tile) for every direction which would mean that there is no capture so the move is invalid. For the Bombing phase, our solution is rather straightforward: when we have a positive amount of bombs and the bomb isn't placed on a hole, then the move is valid. This solution is algorithmically the most effective since we base the search on the known end of the path. In contrast, if we were to search from the other side (so start with an old tile), there would be significantly more potential paths instead of the 8 that we have to evaluate in the worst case with the algorithm we chose.

Task 6

6.1 Evaluating the Parameters

Upon execution our jar needs to respond to the -h, -p and -s parameters, therefore the args array needs to be read. If -h is the only parameter, a short manual is printed. Else if the jar is executed using the -p and -s parameters, followed by their respective value (here, the order does not matter), the jar starts connecting to the server. If any other parameters are set, an error message is displayed.

6.2 Connecting to the server

Java offers two useful libraries for creating a server-client network, the java.net library and the java.io library. First, we need to create an InetSocket using the given parameters. As the constructor works for both, an IP-Address or a hostname, no further processing of the input is needed. By defining a timeout value (1 second), we can now connect our Socket to the server specified in the InetSocket. The default protocol used by this Socket is TCP. If this connection attempt fails, an error message is displayed. To allow for simple sending and receiving of messages, we construct instances of a DataOutputStream and a DataInputStream for our Socket.

6.3 Implementing the Protocol

In order to send our group number to the server, we can make use of the DataOutputStreams write method. Analogously, we can use the DataInputStreams read method to receive our player number. To obtain the map specification, the ai needs to read n bytes, where n is the length of the string describing the map, and convert these bytes again back to a string for further processing. After that, the server starts the game, during which our ai uses the read method to stay up to date. It is working via interrupts, so it does not busy-wait until a byte is ready to read. Receiving the first byte of a message, its value is checked and the program forwards the content to a specified method.

6.4 Sending the first move

The function responsible for delivering the first move, relies on the function checking the validity of a move. It calls this function with every possible combination of x,y coordinates and special fields indicator, upon a positive return value the move is send to the server. This part of the task is performed by the sendMove()

function, which uses the DataOutputStream object to transfer the move, following the required network protocol.