RouteGuesser
Group 9

0. Project Members

1. CJ Reitter
2. Anna Marini
3. Nico Ramos-Fernandez
4. Laith Agbaria
5. Mateusz Wilk
6. Alex Lopez-Ruiz

1. Introduction

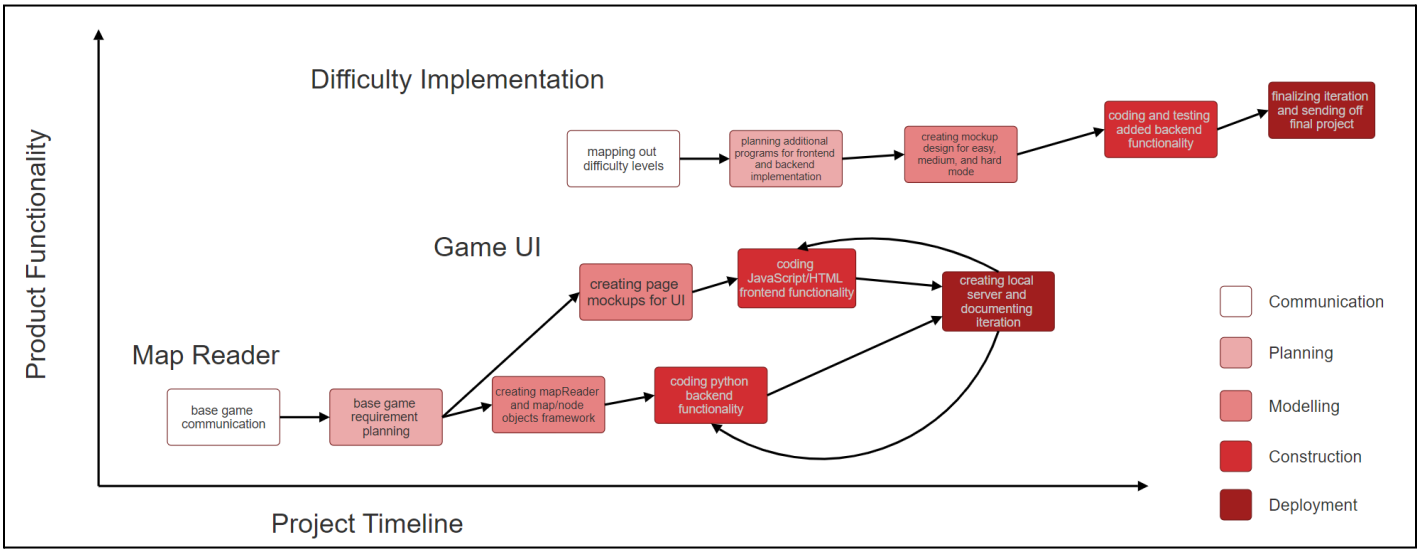
For our Software Development project, we designed a web-based game hosted on a local server called RouteGuesser. The objective of the game is to find the most optimal route from two intersections in a map of Leiden, where you receive a score based on how close your route is to an ideal route between the two points. There is no real endpoint or winning/lose system, since you receive your score and the game either continues or returns to the start page (depending on how many routes you selected to the play). You can change the difficulty which adjusts the number of blocked roads on the map (nodes that you can't select for your path).

We always had a pretty good idea of the framework of the game without a strong sense of the details, so we used a Incremental Process Model to guide our process, with our backend and frontend teams working on reading the dataset and programming the pathing mechanic of the game, and building the framework of the UI respectively. Throughout the project, the other team members worked on testing and debugging the programs, along with the documentation and keeping the directory clean.

2. Software Requirement Specification

Group 9 - RouteGuesser SRS.docx

3. Software Development Process



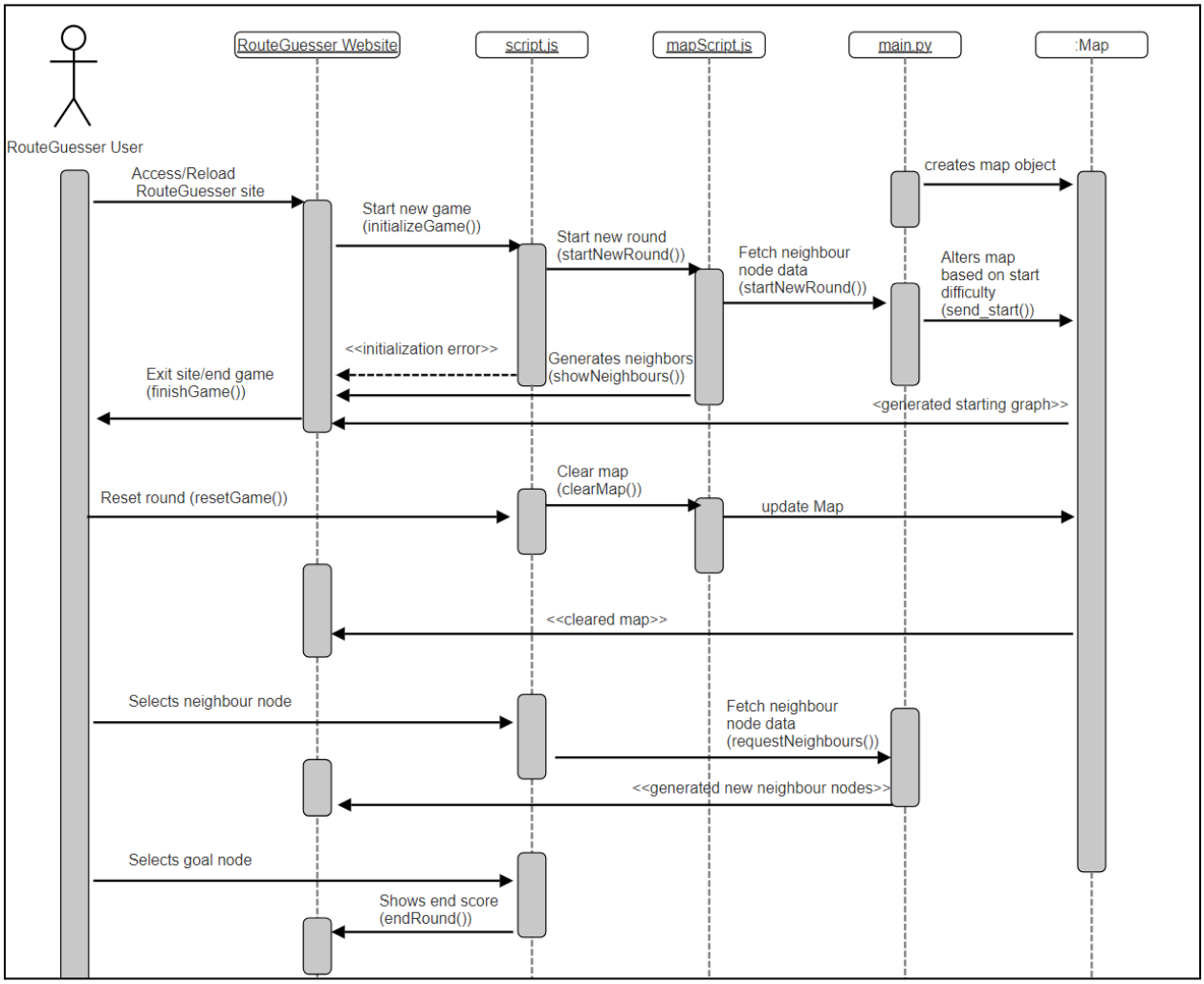
As stated in the introduction, we followed a timeline based around the Incremental Process Model. We started together as a group working on the base planning of the game, figuring out what we would need to simply run a round. From there, the Map Reader and Game UI were designed and constructed in parallel, with testing done concurrently as the frontend needed the backend to work and vice versa. Since we were ahead of schedule, we planned the difficulty implementation as we were working on the base game, implementing it into the base game once we were done with debugging of the base game. This allowed us to debug knowing that any base game problems ultimately came from the addition of the difficulty feature.

4. Software Architecture

4.1. Dataset

The dataset used in the game map dataset was obtained from overpass-turbo.eu in a geojson format using a query designed to reduce as many unnecessary details as possible while maintaining data clarity. Later we created a python module with the purpose of parsing the resulting geojson into a json file containing only the “clean” and useful data that was used in the program files, the clean data contained all the essential information to create a graph representing the game world such as the coordinates of all possible player positions and the roads connecting all the playable coordinates. By first cleaning the data and saving it in a more efficient file format, we reduced the total runtime and made the data more readable reducing the difficulty of debugging. The data was saved in a json file as dictionaries and later parsed into a more easily useable format during the game runtime, by reading and saving it to a custom class object we called Map which handles all the backend game operations and is connected to the main module to communicate with the frontend.

4.2. Methodology



The application under the following methodology, the dataset cleaning module runs to ensure the presence of a clean and usable dataset before applying the startup main program. In the main program a connection is created with the webpage to collect the user input data, it then passes the data to the module responsible for the backend computations by creating an object instance which generates a NetworkX graph representing the current game world followed by a random generation of blocked roads depending on the user chosen difficulty as well as random start and goal nodes. The object then can be used to find the optimal path between the starting and goal locations which will be used to compare the player path later in the game, and after the completion of a round the object can be used to reset the game board without wiping all data to offer the possibility of playing another time. The data calculated and obtained inside the object is finally forwarded back to the frontend through the main channel, this has the benefit of simplicity and compartmentalization.

From a user perspective, the game begins with a small introduction followed by a starting screen to choose a difficulty. After choosing a difficulty the game is initiated and the player is informed of their current location and goal location visually on the interactive map, and by selecting the reachable roads the player can change their location at the cost of increasing their

score. The player is expected to try reaching the goal location with the least score increases by utilizing their understanding of the map and guesses, the player is informed of their score and the game repeats multiple rounds until completion.

4.3. APIs Used

The application we developed employs the NetworkX API due to the plethora of efficient and easy to use functions relating to graphs, which we employed and utilized in the creation of our custom game methods such as those for path finding and blocked roads generation.
The application also employs Flask to communicate effectively and dynamically with the frontend through the use of servers to send and receive JSON requests, by connecting the backend to the server we can dynamically run the methods we developed to update the generated game instance. The frontend uses the Leaflet web API to load a map of Leiden and draw lines representing the player routes and blocked roads, it's also used for the clickable markers and to display the information present on the server.
Since the project uses a lot of dependencies like numpy and Flask, we built two Docker containers, one for the backend and the other for frontend, that communicate with each to make the application run on an isolated process in the user's operating system and to automatically manage the dependencies which makes the installation straightforward.

5. Software Testing

The software was both manually and automatically tested. The manual testing was completed by us developers post-production by first playing multiple rounds of the final game version as a normal player, and then a few rounds attempting to find possible faults in the game logic by doing irrational actions.

As for the automated testing, we deployed a combination of unit tests, functional tests, acceptance tests, and smoke tests depending on which test or combination was most optimal for the given functionality. For each module we programmed, a test file was created to ensure the complete testing of each essential functionality, however due to the smaller scale of the project we did not develop any integration tests. The tests were developed in python using the unittest framework with each test being split into multiple subtests to ensure that results are accurate and as expected while maintaining coding efficiency, with each subtest being responsible for only checking a small part of the overall functionality such as return type or return values. Finally after each module had its own testing file complete and the project was nearing its end, an integrated testing file was created the runs all the separate testing files together to ensure no bugs appear in the last stages of the code.

6. Reflection and Work Distribution

Matuesz Wilk & Anna Marini - Backend Programming (Python/Data acquisition)
Nico Ramos Fernandez & Alex Lopez Ruiz - Frontend Programming (HTML/CSS/JavaScript)
Laith Agbaria - Testing & Programming Cleanup

CJ Reitter - GitLab Management & Documentation

7. Conclusion

Overall, the entire team is very proud of the work we accomplished with this project. We had some problems with inter team fighting during the early planning and communication stages but because all of those were aired out quickly and everyone participated in the project, there was very little miscommunication throughout the project. We had a clear design strategy and subteam assignments, so we were able to streamline the development of the game efficiently. The main problems throughout the project were simple programming problems that are bound to happen during the development of any software.

One of the main limitations was the stagnate nature of our backend code. Everytime our JavaScript program needs information while changing the UI image, the python code is run again. In the future, we would like to implement a more dynamic backend where most of the program, such as the a-star program, only needs to be run once per round and is stored in the server. Another limitation was the inherent time constraint and lack of external testing, meaning there could be bugs in the program that we weren't able to catch during our testing as people much more acquainted with the code. In the future, we would have asked classmates to be beta testers.

We're all happy with the end result of this project and learned a lot about professionalism during these projects and teamwork during the project design.