Project and Design Report

Project: ELeNa - Elevation Based Navigation Application
Team: 0-Day-Exploiters
Members: Aditi Mittal, Chaitanya Rajesh Banala, Stuti Ajmera, Venkata Bramara Parthiv Dupakuntla

This document provides an extensive overview of our project ELeNa including the details of architecture, requirement specifications, and functionalities.

- *Functional Requirements:*
  - Determine a route between user defined start and end locations with either maximum elevation gain or minimum.
  - Also, constrain the route to be at most twice of the actual shortest path.
  - User is given a choice between choosing either Astar or Dijkstra's algorithm to calculate the desired route.
  - The route is displayed using Google Maps to enhance user experience and important route statistics such as distance of the route and the elevation gained are also displayed.

- *Application Flow:*
  The flow of the application can be depicted by segregating the flow of elements running on the backend and on the frontend. We proposed an MVC architecture in the proposal and during the development phase, we realized that a Client-Server Architecture is a suitable framework for the application. Hence, we followed the Client-Server Architectural Model to ensure the application is robust. Figure 1 depicts the architecture that is explained below:
  - *Backend:*
    - We have designed 2 algorithms Astar and Dijkstra's to calculate the shortest path possible while limiting it to x% of the actual shortest path.

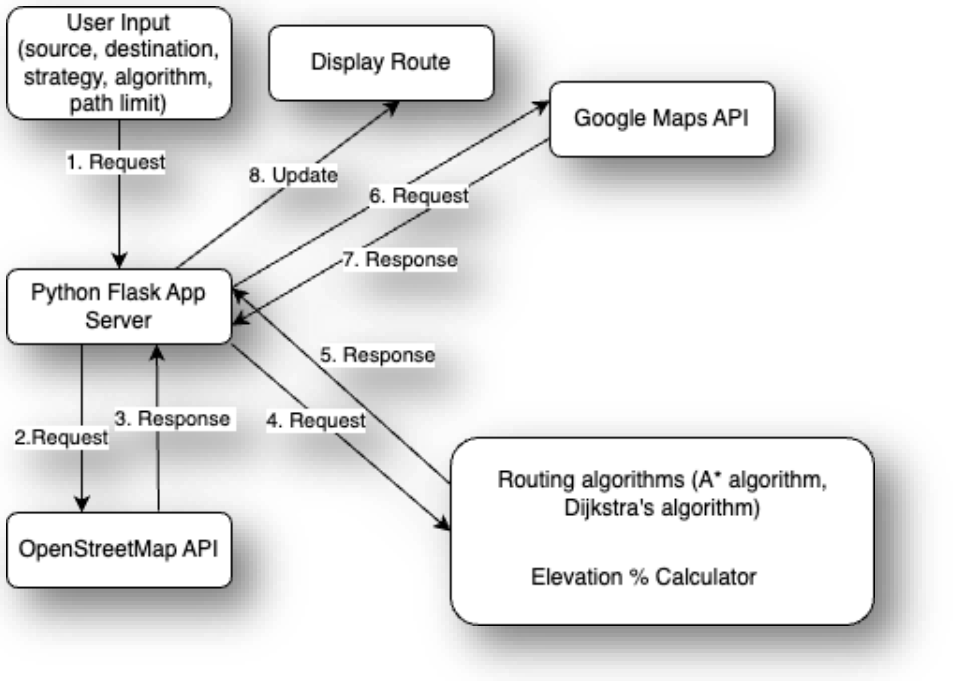

Figure 1: Architecture

- To enhance performance, we have stored the map data readily available for the user to be queried at the back end using the Open Street Maps API.
- This map data pertains to 30,000 meters geographically around Amherst region. Amherst region gives us the ability to perform white box testing as well during the testing phase.
- The map data is parsed through to get the necessary attributes of each node in the graph such as its coordinates, the elevation (measured in meters) at the point.
- We also track the distance of each node from the destination node to have a heuristic setup, useful for the A Star algorithm to run.

- For this to be possible, we run a micro web framework called Flask, which makes appropriate request, response and update calls depicted in Figure 1.
- All the backend files including webapp, algorithms can be found in the '*backend'* folder.

- *Frontend*
  - We developed the frontend of this application keeping in mind that the user experience is mainly dependent on how fluidic the response on the web client is.
  - We used HTML, CSS, and JavaScript to develop the web application.
  - The following functionalities are provided to the user:
    - Source, destination entries
    - Algorithm Selection
    - Elevation Mode Selection
    - Path limit Selection
    - Reset
    - Route Display
    - Route Statistics Display
    - Help
  - Figure 2 depicts the web application and the UI Functionalities
  - We have used Google Maps API to display the final route. This design decision can be attributed to the smooth user experience Google Maps API provides over other map APIs.
  - The default algorithm selected is 'Astar' and the default mode of elevation is 'minimum'.
  - Path limit can be specified up to 200% of the actual shortest path using a slider.

- Keeping in mind the scalability of the application, the user can widen the area of the graph or can change the area according to their preferences in the '*graph.py'* model.
- Help feature gives the user critical information on using the application. This is often an important but overlooked feature from a UI perspective.

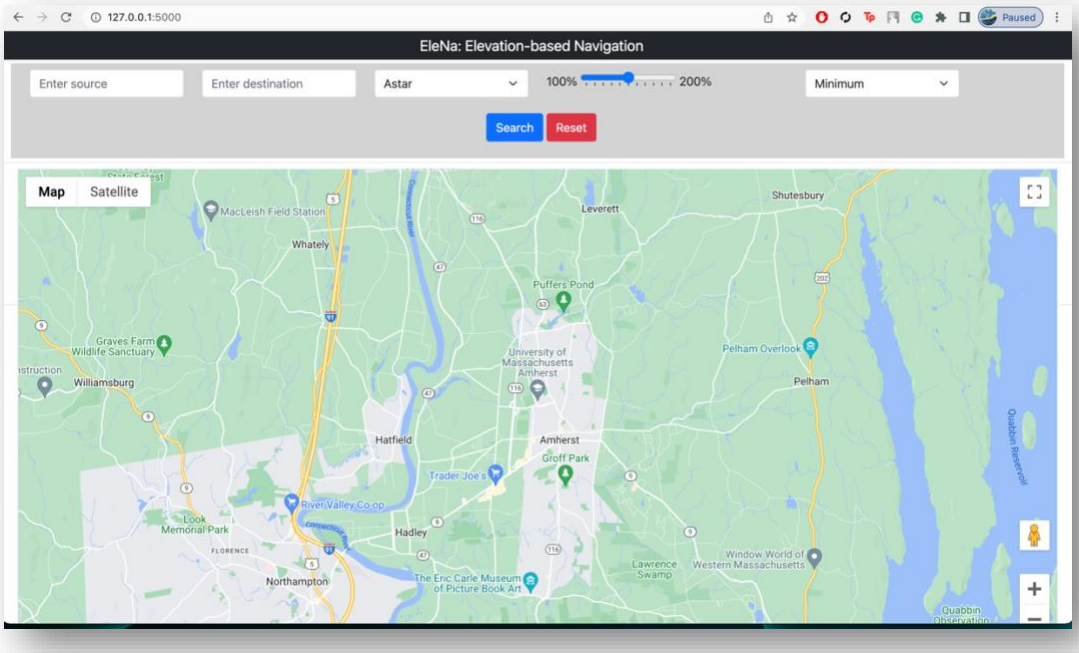

Figure 2: The Application Interface

- *Performance Analysis:*
  - A star algorithm is an extension of Dijkstra's where the pathfinding process is guided by the heuristic function. In a nutshell, the next node's cost is calculated as follows:

$$F(x) = G(x) + H(x)$$

  - G(x) is the g score calculated using the costs between two nodes. These costs are set using the cost function depending on which mode of elevation we are using.
  - H(x) is the heuristic function which stores costs of all nodes from the end node.
  - F(x) is the total cost of the current node.

- Using Dijkstra's, the algorithm searches in all directions without choosing the next 'best' node. This should not be a problem theoretically considering the time complexities of both the algorithms is asymptotically same, i.e, if we assume the heuristic function to be consistent, the worst-case time complexity of both the algorithms would be $O(|E|+|V|\log|V|)$.
- However, considering a real-time graph with possibly complex node structure, A star performs slightly better than Dijkstra's, which is the reason for choosing A star to be the default algorithm selected. It is worth noting that Dijkstra's will always give the correct result, which can be considered as a tradeoff with performance.

- *Testability:*
  - Testability of an application is an important aspect that keeps the software smooth and bug free.
  - We have implemented several tests on the application, testing various features such as its UI, server, algorithmic correctness, etc.
  - We have performed White box testing on the output as well by verifying if the route displayed is actually the desired route.

  - One such test instance is explained below:
    - We checked for a route with maximum elevation between two of our houses. Since we are familiar with the shortest route between these two points, we were certain that the route being displayed is not the shortest. We traversed the route and found out that the deviation from the actual route is because of 2 evident slopes being present, very near to the nodes on the shortest path. Hence, there was a detour to include those slopes so that the maximum gain increased, proving the algorithmic correctness as well as UI correctness.
    - We performed several such traversals in our proximity to further reinforce the correctness of the application.

- Apart from that, we have written several test cases covering frontend elements, etc., and web server elements such as requests and algorithmic tests are included as well.
- This Test suite, comprising of extensive test coverage helped us enhance debuggability during the process of development as well.
- We have written an extensive Test Suite of 21 test cases covering the backend as explained above. Please refer to Figure 3 for test session summary.
- Please find attached the UI Testing document in the documentation section of the repository

Figure 3: Test Session Summary

- Design Strategy:
  - We have used the template design pattern as evident from the UI, we are giving the user options to choose for the algorithms, elevation etc.
  - We have also used a responsive UI strategy as mentioned above to enhance UI/UX capabilities of the application.

- *File Structure/ Source Code framework:*
  - Frontend contains the following files:
    - Webapp.py: Source code for setting up the flask app and running the server with appropriate requests
    - Static and Templates: Collection of frontend stack built using html, JavaScript <map.js>, and CSS files.
  - Backend contains the source code for the following files:
    - Utils: Sub directory for all the utility functions –
      - Algo_utils.py: All the necessary helper functions for algorithms

        - astar.py: Design and development of A* algorithm
        - Dijkstra.py: Design and development of Dijkstra's algorithm
    - Algorithms.py: Final shortest route design using all the algorithms and utilities
    - Graph_utils.py: Development of the graph, fetching all the parameters required to implement algorithms

Hence, there is a lot of scope for extensibility, usability, and scalability of the application which sums up the goal of a development project and we believe that application of the learning outcomes of this course are evident from this project report.