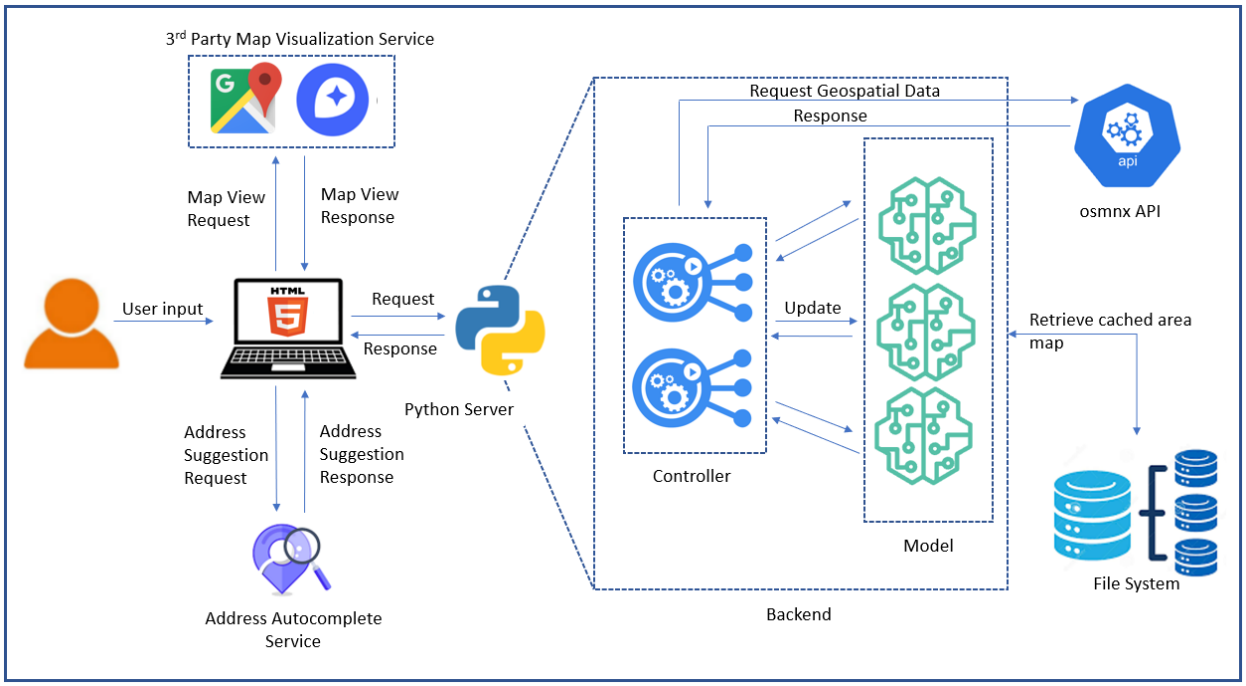
CS520 Developer Document
SEleNa
Simplified Elevation Navigation

Chirag Kamath | Divya Maiya | Neha Prakash
Philip George | Rahul Tandon

Table of Contents

Application Architecture



The architecture diagram above depicts the data flow and interactions between the various components of the system when given user input.

Manual Setup

There is some manual setup done prior to deploying the web application which is outlined below:

1. Since our application supports both Boston and Amherst addresses, we used the OSMNX library to download the maps as local pickle files.
2. These files act as the base map to which we add elevation data based on our needs via another OSMNX API.
3. Once files are set up, we deploy our application with these files being persisted as part of the deployed file system.

Flow of Control

The flow of control for the web application is as follows:

1. The user interacts with the front-end UI of the web application and provides the various inputs required to calculate and visualize the path between source and destination as per the user's preference.

2. At the time of taking user input, the source and destination fields make API calls to the **Google Places Autocomplete API** to provide address autocomplete functionality to the

user, thereby allowing for precise source and destination addresses which facilitate more accurate results.

3. Once the user input has been entered, the user is allowed to make 2 types of requests:
    a. View elevation statistics and graphs based on the user input
        i. The different user selections are packaged into an API request body. A POST API call is made to the Flask backend server and passing the request body to retrieve the list of nodes that represent the path that has been found by the algorithm and **OSMNX**, along with other elevation information relating to this path.
        ii. Once the API response is received by the front end, elevation statistics are calculated based on the elevation information provided in the API response. The following statistical measures are calculated at the client side:
            1. Total distance from source to destination
            2. Steepest incline
            3. Total Elevation Gain/Loss
            4. Average inclination
        iii. Finally, the elevation gain at each node is mapped to a line chart that is visualized via the **Google Chart Library**, which allows the user to understand the change in elevation throughout the path.

    b. Visualize the path in real time between source and destination
        i. The different user selections are packaged into URL query parameters and passed to the view_route.html page as part of the URL. This new page unpacks and decodes these parameters and makes a POST API call to the Flask backend server to retrieve the list of nodes that represent the path that has been found by the algorithm and OSMNX, along with other elevation information relating to this path.
        ii. This information is now mapped to a real-time map using the **MapBoxGL** visualization library that allows for the simple rendering of real-time maps with customizable paths and markers.

4. As can be seen from the above data paths, a single common API endpoint is exposed at the backend but is used in different contexts as per the requirement. This promotes code reusability and removes the need to create two different endpoints when the base information used by both use cases is the same. Furthermore, since the client-side calculation is minimalistic and lightweight, this is an optimal approach.

5. At the backend when the API endpoint is called, the user input present in the API request body is extracted and fed to the appropriate controller functions. We support various combinations of user data to calculate the elevation-based path from source to destination, namely:

a. City – Amherst / Boston
b. Source and Destination
c. Algorithm – Dijkstra / A-Star

d. Deviation Limit – 0 to 100%
e. Elevation Optimization – Maximize / Minimize / Shortest Path

6. The backend code structuring and implementation follow the **MVC architecture**. The server file exposes the endpoint that is called by the front end. As per the user input, OSMNX is used to add elevation-based geospatial data to our base map pickle files and the appropriate algorithm is used to find the shortest path between source and destination and then we maximize or minimize elevation gain.

7. The user also has options to clear and reset their requests to try different combinations of input.

Tech Stack

Front End

1. **HTML / CSS / JavaScript** - Used to develop the different component pages in our web application and integrate them with the backend server.
2. **MapboxGL JS** - Mapbox GL JS is a JavaScript library for vector maps on the Web. Its performance, real-time styling, and interactivity features set the bar for anyone building fast, immersive maps on the web.
3. **Bootstrap 5** - Bootstrap is a free and open-source collection of CSS and JavaScript/jQuery code used for creating dynamic layout websites and web applications.
4. **Google Places API and Maps JavaScript API** - The Places API is a service that returns information about places using HTTP requests. Places are defined within this API as establishments, geographic locations, or prominent points of interest. The Maps JavaScript API lets you customize maps with your own content and imagery for display on web pages and mobile devices. The Maps JavaScript API features four basic map types (roadmap, satellite, hybrid, and terrain) which you can modify using layers and styles, controls and events, and various services and libraries.
5. **Google Charts Library** - Google Charts is an interactive Web service that creates graphical charts from user-supplied information. The user supplies data and a formatting specification expressed in JavaScript embedded in a Web page; in response the service sends an image of the chart.

Backend

1. **Python** – Used as the base language for the entire server-side application in conjunction with the Flask Framework.
2. **Flask Framework** - Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.
3. **OSMNX** - OSMNX is a Python package that lets you download geospatial data from OpenStreetMap and model, project, visualize, and analyze real-world street networks and any other geospatial geometries.
4. **NetworkX** - NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.
5. **PyTest** - PyTest is a mature full-featured Python testing tool that helps you write better programs. The PyTest framework makes it easy to write small tests yet scales to support complex functional testing for applications and libraries is a mature full-featured Python testing tool that helps you write better programs.
6. **Python unittest** - The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.
7. **Python Mockito** - A port of Java's testing framework with the same name, it's safe by default unlike mock from the standard library, and has a nice, easy-to-use API. It also helps with the maintenance of tests by being very strict about unexpected behaviors.

8. **Python Coverage** - Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.


Software Architecture and Design Patterns

SEleNa largely follows a hybrid of the three-tier and model-view-controller software architectures.

Three-Tier Architecture

1) Presentation Tier

The presentation tier is the user interface. In our application, that consists of the following two files written in HTML, CSS, and Javascript

a) **Selena.html**, which is the main application interface
b) **View_route.html**, which is used to display the rendered route

2) Application Tier

The application tier contains SEleNa's functional logic. Briefly, these are the modules

a) **Server:** Initializes the Flask server and links the '/' and '/route' API endpoints to their respective functions. It's the point of entry into the application from the UI.
b) **Controller:** The functions in server.py that are linked to API endpoints delegate the actual implementation of the logic to the controller which handles the specifics of the request - source, destination, deviation, mode, and algorithm to

be used. It manipulates the model whenever a change to the application state is triggered.

c) **Model:** Contains the current state of the application - the algorithm being used, the mode (minimum elevation, maximum elevation), the permitted deviation from the shortest path, etc
d) **Algorithms**: Implements the template design pattern with a single abstract Algorithms class with get_maximum_elevation() and get_minimum_elevation() functions. The concrete implementations of A Star, Dijkstra, and BFS are contained in their respective classes and are only concerned with the get_shortest_path() functionality.
e) **Utils:** map_utils and graph_utils contain helper functions, for e.g. retrieving the closest node from an OSMNX graph given a string representation of an address.

3) Data Tier

The data tier consists of the .pkl files that allow us to persist the NetworkX graphs corresponding to Boston and Amherst.

MVC Architecture

SEleNa also **loosely** follows the MVC architecture.

1) View

Logically, the view corresponds to the presentation layer as defined above. While most of the updates at the UI level (text box validation, display graph, etc) are handled by the frontend JS code, the core logic of the program (get route) involves the python controller updating the view in response to an API call. Therefore, this is a partial implementation of a view.

2) Model

The model has been explained above in the Application Tier. Since it relies on the pickle files (from the data tier) to persist its state, it's a partial implementation of a true model.

3) Controller

The controller has been explained above in the Application Tier. The functionality corresponding to the actual algorithms used (A*, Dijkstra, BFS) has been abstracted in a separate module; this is therefore a partial implementation of the Controller.

Template Design Pattern

We also implemented the Template Design Pattern in the Algorithms module.

1) Abstract Class (Algorithms)

Algorithms.py contains the abstract Algorithms class. This class defines a constructor and methods for minimum and maximum elevation that are agnostic to the specific algorithm required in that iteration of the program. The minimum and maximum elevation methods take the shortest path between two points as input and attempt to modify the path to maximize elevation gain, given the constraints of the maximum permitted deviation. They are indifferent to the algorithm used to calculate the shortest path.

2) Concrete Classes (AStar and Dijkstra)

These classes implement the get_shortest_path() method which takes two map nodes as input and calculates the shortest path between them. Once this is done, the calculation of the maximum and minimum elevation variant of that path is left to the abstract class.

Experimental Evaluation

The experimental evaluation for the web application mainly revolved around the comparison of the different algorithms and measuring the quality of our code.

Metric	Dijkstra	A-Star
Time Complexity	O((|V| + |E|) log V)) where : V is the number of vertices E is the total number of edges	O(|V|+|E|) where : V is the number of vertices E is the total number of edges
Space Complexity	O((|V| + |E|) log V) where : V is the number of vertices E is the total number of	O(|V|) where V is the number of vertices
Run Time	Using the *time* library in Python we measured the run time of both algorithms under the same source-destination pairs and same optimization criteria. The A-Star algorithm consistently had a run time of 12-15% less than Dijkstra's algorithm, which makes it faster. This was consistent with the expected results.	

Statistics

Apart from providing the elevation path between source and destination on the map, our web application also provides the user with certain elevation-based statistics which provides useful insights about the optimal path traced by the selected algorithm. The calculation of the statistical metrics was done as follows:

1. **Total Distance (in kilometers)**: $\Sigma(\text{distance}_i)/1000$, where distance_i is the distance from node x_{i-1} to x_i.
2. **Steepest Incline (in vertical meters per 10 meters)**: $\Sigma(\text{Max}(0, \text{grade}(x_i))) * 10$, where $\text{grade}(x_i)$ is the elevation grade at node x_i.
3. **Total Elevation Gain (in meters)**: $\Sigma(\text{elevation}(x_i): \text{elevation}(x_{i-1}))$, where $\text{elevation}(x_i)$ denotes the elevation gain at node x_i with reference to the source node.
4. **Average Incline (in vertical meters per 10 meters)**: $(\Sigma(\text{grade}(x_i)) / N) * 10$, where $\text{grade}(x_i)$ is the elevation grade at node x_i and N is the number of nodes between source and destination.

We tested different Amherst and Boston addresses with Minimum and Maximum elevation paths using both the supported algorithms. To verify our results relating to the elevation data we calculated the above statistics and found that they were consistent with the expected results from the internet.
The results and sample screenshots can be found in the README.md file present in the repository.

Code Quality using SonarQube

We used Python's version of SonarLint to make sure our code quality adhered to certain standards and best practices. Overall our code was evaluated based on the following criteria:

1. **Cognitive Complexity** - We maintained our method complexity under 10-12 as far as possible.
2. **Duplications** - In order to eliminate duplicate code and promote code reusability, we made sure our code had no duplicate blocks.
3. **Maintainability** - No code was pushed to the remote repository which had any minor/major code smells. This resulted in low technical debt and our overall code has a maintainability rating of A, which maps to the best rating available.
4. **Reliability -** To prevent accidental breakage of the data pipeline, our code was monitored for possible logical or syntactical bugs. Our final code had a Bug rating of A which maps to 0 major or minor bugs in the code.
5. **Coverage** - We made sure our final code had a line and condition coverage of at least 85% to ensure that we tested all code before pushing it to the repository.

Testing

One of the key principles of Software Design is testing. Having a test suite helps easily

detect bugs, unwanted behavior, and any red herrings. Additionally, it helps reproduce and fix any new bugs. We used several different testing approaches and frameworks to ensure that the code we wrote was up to industry standards. We used the *unittest* and *pytest* Python framework for our test suite. Below we summarize the different types of testing we undertook.

Unit Testing

Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. Unit testing is an important step in the development process because if done correctly, it can help detect early flaws in code which may be more difficult to find in later testing stages.

Our implementation includes a vast test suite to cover several different scenarios. These tests independently test the different modules in the source code. In order to write efficient unit test cases that do not rely on external services, we use the technique of mocking. Mocking is primarily used in unit testing because an object under test may have dependencies on other (complex) objects. To isolate the behavior of the object we replace the other objects with mocks that simulate the behavior of the real objects. In our test suite, we use the *mockito* library to create mocks of several modules that are dependencies of the module under test.

Implementation modules covered in testing:

- Algorithm
- Cache
- Controller
- Model
- Utils
- Server

For each of the above modules, we tried to cover several different testing scenarios, in order to capture all intended and unintended system behavior. For example, for the AStar algorithm, our test suite covers the following scenarios -

- Path between start and end points exists
- No path exists between start and end points
- Path with min elevation
- Path with max elevation

Integration testing

Integration testing is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements. Since we use external libraries like *osmnx* and *networkx*, in order to ensure that these third-party applications and our internal

modules work seamlessly together we performed a series of automated and manual integration tests to test the combination and evaluate the results.

For instance, we have several tests for the controller and server modules, which test the end-to-end system performance. These tests cover both our internal modules and external APIs to see how well they perform together.

User Acceptance Testing

User Acceptance Testing (UAT) is the final stage of any software development life cycle. This is when actual users test the software to see if it is able to carry out the required tasks it was designed to address in real-world situations. In this phase, we performed Alpha and Beta testing.

- ### Alpha Testing

  Alpha Testing is a type of acceptance testing performed to identify all possible issues and bugs before releasing the final product to the end-users. For our project, we each individually alpha tested the components and summarized our findings. We then used this to improve a few things in our approach.

- ### Beta Testing

  Beta Testing is the testing performed by users (not developers) of the software application. In order to Beta test our application, we asked a few of our friends and acquaintances (potential end-users of the product) to use our application and provide feedback on the product quality using a **Usability Survey** we created. We then used this feedback to update some of the UI features to improve intuitiveness and overall user experience.

Code Coverage

Code coverage is the percentage of code that is covered by automated tests and allows for a feedback loop in development. We use the python *coverage* library to evaluate the effectiveness of our automated tests. We received an overall coverage of 84% on our implementation.

The below table summarizes the per module code coverage we obtained:

Coverage report: 84%

Module	statements	missing	excluded	coverage
src/__init__.py	0	0	0	100%
src/backend/__init__.py	0	0	0	100%
src/backend/algorithm/__init__.py	0	0	0	100%
src/backend/algorithm/algorithm.py	58	14	0	76%
src/backend/algorithm/astar.py	52	0	0	100%
src/backend/algorithm/bfs.py	30	2	0	93%
src/backend/algorithm/dijkstra.py	49	1	0	98%
src/backend/controller/__init__.py	0	0	0	100%
src/backend/controller/controller.py	31	6	0	81%
src/backend/data/__init__.py	0	0	0	100%
src/backend/data/download_data.py	0	0	0	100%
src/backend/model/__init__.py	0	0	0	100%
src/backend/model/model.py	27	5	0	81%
src/backend/sample.py	0	0	0	100%
src/backend/server.py	37	19	0	49%
src/backend/utils/__init__.py	2	0	0	100%
src/backend/utils/graph_utils.py	36	10	0	72%
src/backend/utils/map_utils.py	56	5	0	91%
Total	**378**	**62**	**0**	**84%**

coverage.py v6.2, created at 2021-12-13 17:45 -0500

Project Management

Collaboration Tools and IDEs

1) **Slack** for all team communication.
2) **PyCharm** as python IDE, **VSCode** for HTML, and chrome developer tools
3) **Git** for Version Control

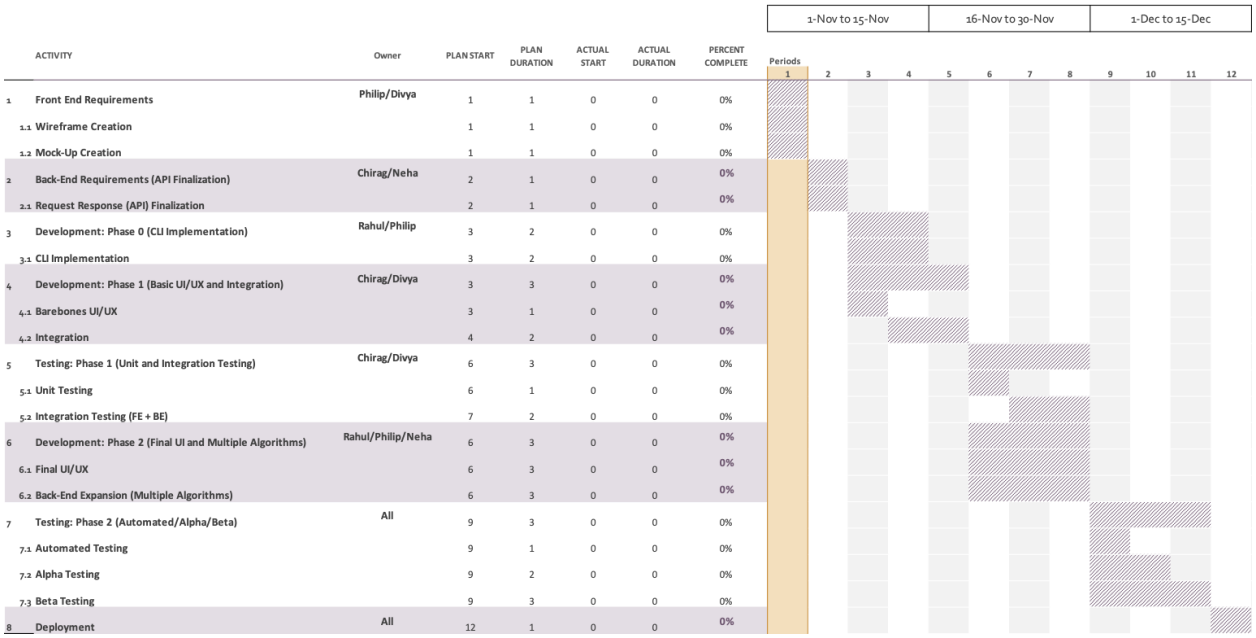
Agile (Scrum) Practices

Due to the small size of the team, and the relatively short timeline, we only loosely followed agile development practices.

Timelines

1) The total timeline for the project was 12 weeks
2) Each sprint was 1 week long
3) We followed an MVP approach, so every sprint was an internal release sprint, i.e. our entire team tested the latest MVP every week.

Detailed timelines are given below:



	ACTIVITY	Owner	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
							1-Nov to 15-Nov / 16-Nov to 30-Nov / 1-Dec to 15-Dec
							Periods 1 2 3 4 5 6 7 8 9 10 11 12
1	Front End Requirements	Philip/Divya	1	1	0	0	0%
1.1	Wireframe Creation		1	1	0	0	0%
1.2	Mock-Up Creation		1	1	0	0	0%
2	Back-End Requirements (API Finalization)	Chirag/Neha	2	1	0	0	0%
2.1	Request Response (API) Finalization		2	1	0	0	0%
3	Development: Phase 0 (CLI Implementation)	Rahul/Philip	3	2	0	0	0%
3.1	CLI Implementation		3	2	0	0	0%
4	Development: Phase 1 (Basic UI/UX and Integration)	Chirag/Divya	3	3	0	0	0%
4.1	Barebones UI/UX		3	1	0	0	0%
4.2	Integration		4	2	0	0	0%
5	Testing: Phase 1 (Unit and Integration Testing)	Chirag/Divya	6	3	0	0	0%
5.1	Unit Testing		6	1	0	0	0%
5.2	Integration Testing (FE + BE)		7	2	0	0	0%
6	Development: Phase 2 (Final UI and Multiple Algorithms)	Rahul/Philip/Neha	6	3	0	0	0%
6.1	Final UI/UX		6	3	0	0	0%
6.2	Back-End Expansion (Multiple Algorithms)		6	3	0	0	0%
7	Testing: Phase 2 (Automated/Alpha/Beta)	All	9	3	0	0	0%
7.1	Automated Testing		9	1	0	0	0%
7.2	Alpha Testing		9	2	0	0	0%
7.3	Beta Testing		9	3	0	0	0%
8	Deployment	All	12	1	0	0	0%

Ceremonies

1) We had a Daily Standup meeting for 15 minutes.
2) We had weekly Sprint Planning and Review meetings of an hour each.
3) We had ad-hoc retrospective meetings depending on specific challenges in the release cycle.