# Compsci 520 Final Project Design Document

Team Nomad
Zihao Zhang, Shenghao Guan, Eric Hsu, Haoru Song

## Introduction

The Elevation based navigation(Elena) is a program that determines a route between two locations based on the desired optimization of elevation gain. It allows users to minimize or maximize elevation gain while also limiting the total distance to a specified percentage of the shortest path. This is particularly useful for hikers and bikers who want to plan their routes based on their desired level of intensity and elevation gain. The system considers the topography of the area and calculates a route that meets the specified optimization criteria.

## Requirement

1. The system must be able to determine a route between two locations based on the desired optimization of elevation gain.
2. The system must allow the user to specify whether they want to minimize or maximize elevation gain.
3. The system must be able to calculate the total distance between the two locations and allow the user to specify a maximum distance as a percentage of the shortest path.
4. The system must be able to take into account the topography of the area and calculate a route that meets the optimization criteria.
5. The system must provide a visual representation of the route.
6. The system must be able to handle multiple modes of transportation, such as walking, biking, or driving.
7. The system must be user-friendly and easy to use.

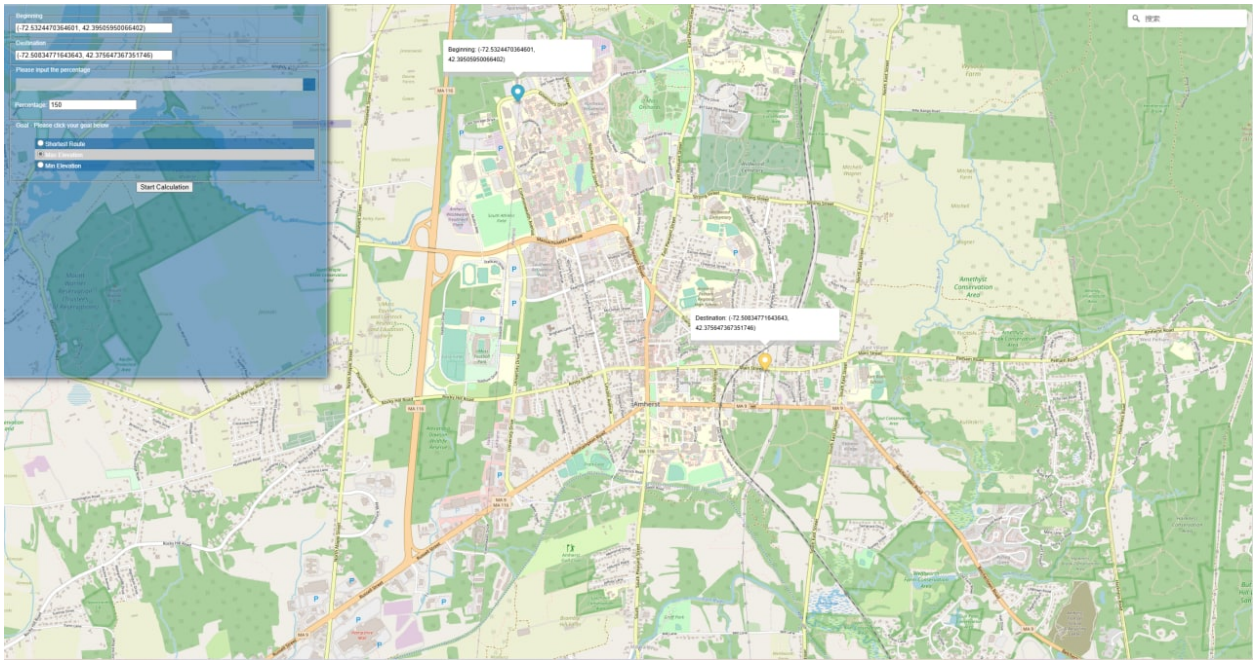
## Architecture

We are using the MVC (Model-View-Controller) architecture for this program. It separates our program into three components: the model, the view, and the controller.

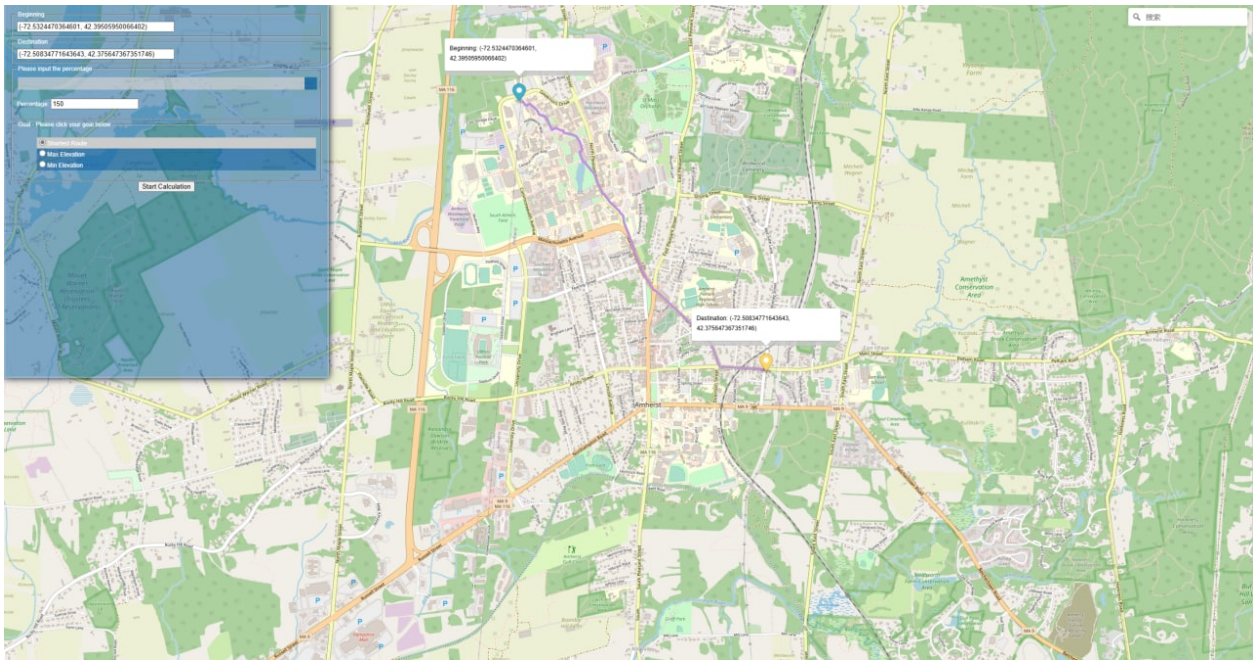
### View

The front-end of the application, known as the view, is responsible for displaying the user interface and handling user input. On the right side of the view is a map that allows users to select the start and destination points. The left side of the view is the controller panel, which displays the coordinates of the start and end points. Users can also specify their search requirements in this panel. When the user clicks the search button, the results are displayed on the map.

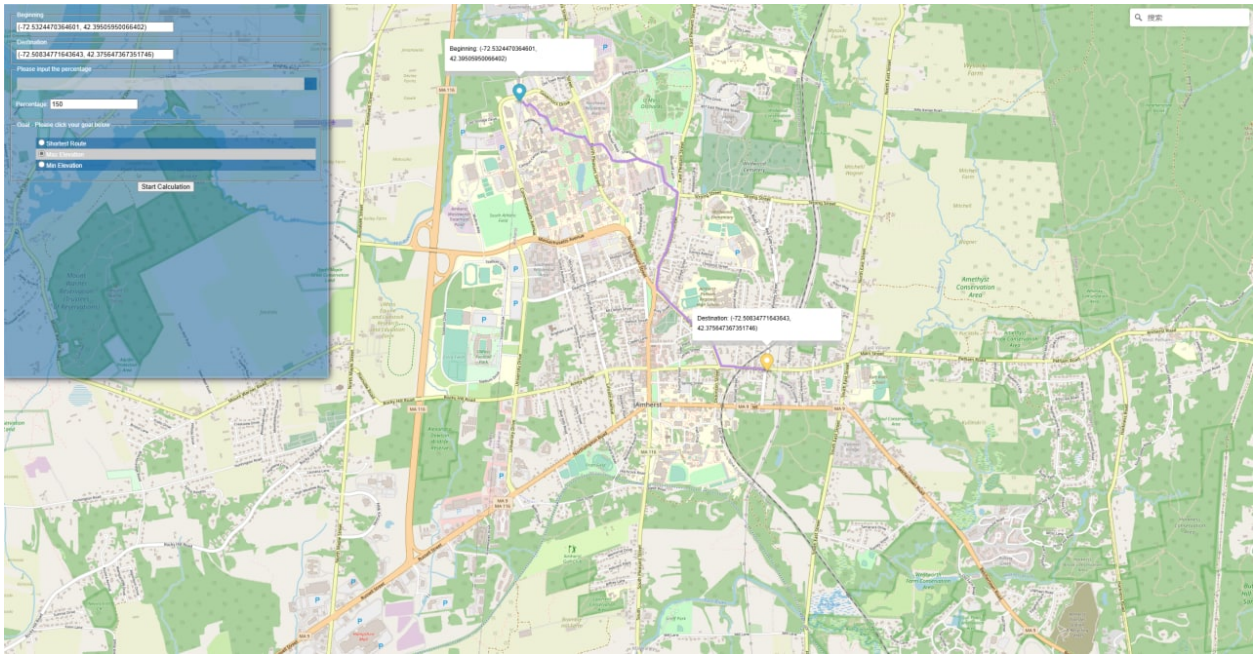
***The initial view for the page:***



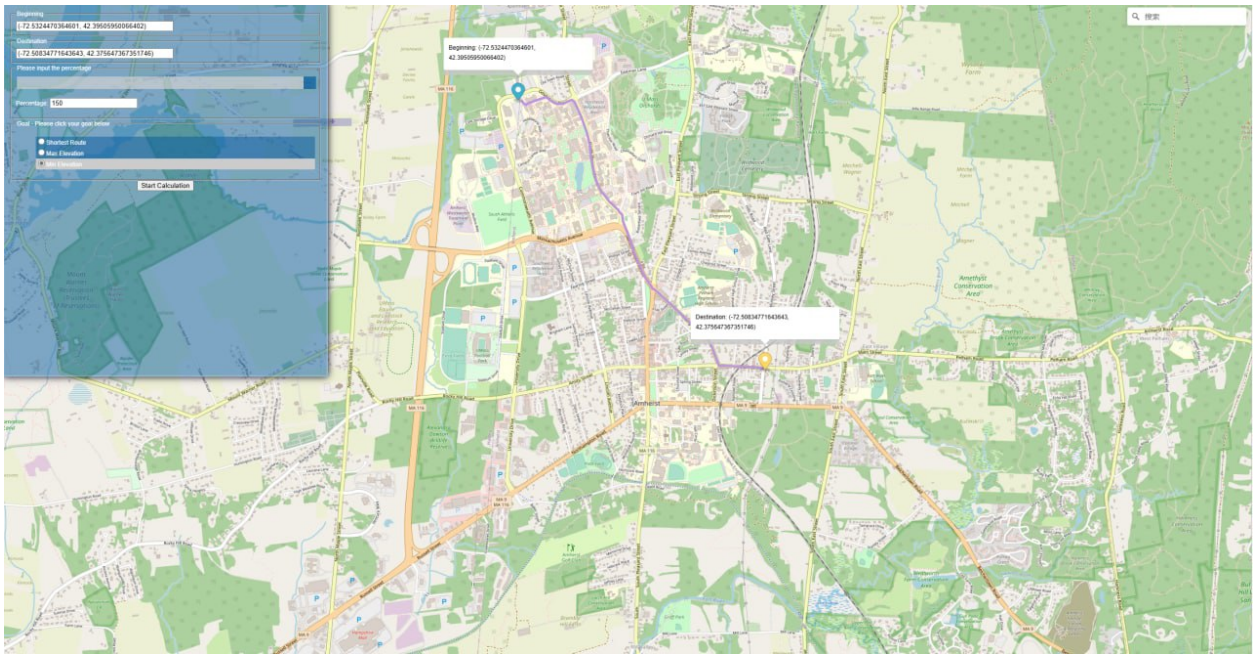

***Route with shortest distance:***




***Route with maximum elevation gain:***



***Route with minimum elevation gain***



## Model

The model contains the data and logic of the application, including the map data, the return object, and the api key, and searching algorithms. It is responsible for manipulating the data and providing it to the view as needed.

***GeoDataModel***

The GeoData class is used to request and store map data for a given search area. It takes the starting and destination coordinates and an API key as arguments and uses them to access the map data. The data is stored in a networkx.MultiDiGraph object that can be used to search for routes within the specified area.
Args:

- source ((float, float)) : the coordinate of the starting point
- dest((float, float)) : the coordinate of the destination point
- key (String) : the API key used to access the map data.
- G (networkx.MultiDiGraph) : MutliDiGraph object that contains the
- map data in the searching area. Defaults: None
- test( boolean): flag for unit test. Default: False

***Keys***

File that stores all api keys that will be used.

- google_elevation_api_key: used for getting elevation data of each node in the networkx.MultiDiGraph object.

***RouteModel***

The RouteData class stores information about a particular route. It takes a list of coordinates representing the path, the total length of the path, and the total elevation gain as arguments. The coordinates correspond to the nodes in the node list in the same order.
Args:

- path (list of coordinates) : list contains coordinates, which each
  coordinate corresponds to a node in the node_list in the same order
- length (float) : the total length of the path
- elevation (float) : the total elevation gain of the path

***Route_Algorithm***

Elena navigation contains four search methods.
The function will be invoked by the server, once the server receives the request from the client. It will first determine the shortest route. Based on the length of this route, the algorithm runs the A*, Dijkstra and genetic algorithms to find a route that meets the specified search criteria as closely as possible. Finally, it will wrap the path, length, and elevation gain into the RouteModel class, and return the RouteModel class to the server.

***Shortest Route Algorithm***

We used a method in osmnx, ox.shortest_path, to find the shortest route between start and destination point.

***A* Algorithm***

We modified original A* algorithm to make elevation gain as a factor of routing.
To do this, we modified the cost function of A* to curr_distance + 15 * elevation gain - heuristic_distance

***Dijkstra Algorithm***

We modified the original Dijkstra algorithm to make elevation gain as a factor of routing.
To do this, we also consider the elevation when calculating the distance to update each node. For example, when calculating the maximum elevation gain, we subtract the altitude from the distance to make the algorithm prefer routes with greater elevation gain.

***Genetic Algorithm***

A genetic algorithm is a heuristic search algorithm that is inspired by the process of natural evolution. The basic idea behind a genetic algorithm is to use a population of candidate solutions, called chromosomes, and to apply genetic operators such as crossover and mutation to evolve these chromosomes towards better solutions. The goal is to find the fittest chromosome in the population, which is the chromosome that represents the best solution to the problem at hand.
To implement a genetic algorithm for finding max/min elevation gain, we designed a fitness function, a crossover function and a mutation function. The fitness function calculates the fitness of a given route in a graph by the elevation gain of the route and the distance of the route. For example, when calculating maximum elevation gain, the fitness function returns the elevation gain of the route if the route length is less than the distance limit and returns a very small value if the route length exceeds the distance. The crossover function combines two routes together to create a new route. If no intersection is found between input routes, the function randomly selects two nodes from both routes and finds the shortest route between them, which is then inserted between the two selected nodes in the original routes to create a new route. If an intersection is found, the function randomly samples two nodes from the intersection, and finds their indices in the two routes. The new route is then constructed by combining route1 and route2 together using these indexes. The mutation function randomly applies mutation operation to input routes, nodes in the input route are randomly replaced by another node in the geodata to create a new route which is slightly different from the input route.

## Controller

The controller is the intermediary between the model and the view. It receives user input from the view and communicates with the model in the backend to update the data and return it to the view.

***Client side***

Once the user hits the start calculation button, the client will wrap the following data to the JSON and send a POST request (/get_rout) to the server.

- Source (float, float): latitude and longitude of starting point. The first element is latitude, the second element is longitude.
- Destination (float, float): latitude and longitude of destination point. The first element is latitude, the second element is longitude.
- Is_max (String): Specify the searching criteria to determine whether to return the route with maximum elevation gain, minimum elevation gain, or shortest distance.
- percentage(Integer):  a percentage value that is used in the calculation of the distance limit for the elevation gain routing methods

***Server side***
There is a function, called get_route, which will handle the route request from the client.

Here are the steps that the server takes after receiving a request:

1. When a request is received by the server, the source and destination points remain unchanged. The server then divides the percentage value by 100. If the is_max value equals 'Shortest_Route', the server initializes a new variable called method and sets it equal to ‘S'. If is_max equals 'Max elevation', the server sets is_max to true, otherwise it sets is_max to false.

2. The server calls find_route function to get the route closest to the requirement specified by the user.

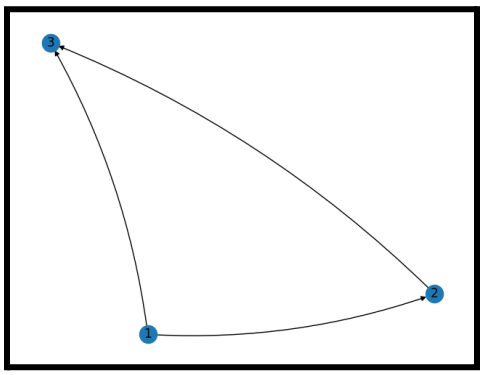
3. The server creates a JSON file, containing the path of the route and sends it back to the client.

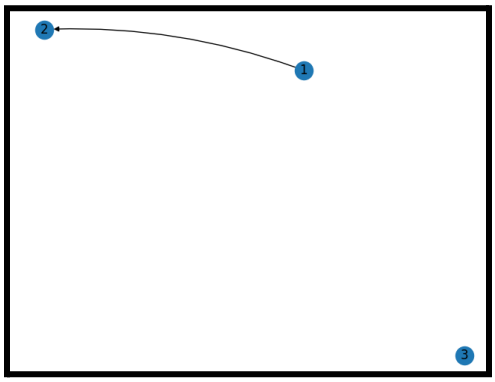
## Testing

Testing included tests for the routing algorithms, utility functions, and API connection.

Due to the large number of extraneous factors in the OSmnx data (elevation, length, placement of nodes), testing needed to be conducted in a closed environment of easily interpretable lengths.

For our testing of the utility and routing functions within *Route_Algorithm_test.py* and *util_test.py*, we used the following 2 networks of nodes as our testing environments:




The fully connected environment gives an environment to see the expected behavior of the algorithm with no exception handling and to find standard bugs from edge cases. The disconnected environment brings about the possibility of impossible situations (i.e., length of

inexistant edge between nodes, find shortest path between two disconnected nodes, etc.) and the throwing of exceptions.

As for the API connection testing within *googleAPI_test.py*, the test ensures that the API key that we used is accepted by the Google Maps API for finding elevation.

## Experiment

Percentage = 1.5

We conducted experiments on our algorithm with 40 randomly selected starting and ending groups in Amherst. The table below shows the average running times, average elevation gain and average route length of these three algorithms we used.

| Algorithm | Goal: Maxium elevation gain | | | |
|---|---|---|---|---|
| | Shortest route | Astar | Dijkstra | Genetic |
| Average run time(s) | / | 0.03 | 4.01 | 269.44 |
| Average elevation gain(%) | 100 | 147 | 111 | 284 |
| Average length(%) | 100 | 125 | 116 | 145 |

| Algorithm | Goal: Minimium elevation gain | | | |
|---|---|---|---|---|
| | Shortest route | Astar | Dijkstra | Genetic |
| Average run time(s) | / | 0.16 | 4.89 | 291.42 |
| Average elevation gain(%) | 100 | 97 | 99 | 80 |
| Average length(%) | 100 | 106 | 115 | 104 |

It turns out that the genetic algorithm always has the best performance, but it can take a long time to run this algorithm. Astar algorithm performs well when finding maximum elevation gain, it's very fast and get acceptable elevation gain, but this algorithm sometimes cannot find a route and returns nothing and performs not very good when finding minimum elevation gain.