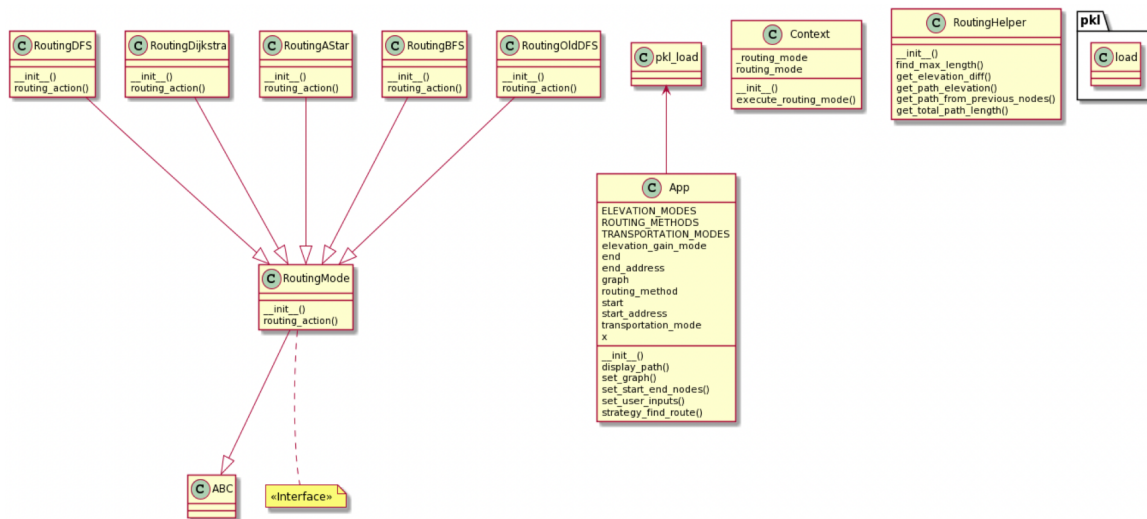


Group name: Heatwave 2.0

Group Members: Kashish Arora, Srisuma Movva, Ji Ye, Tarang Mittal

EleNa Project Document

Architecture



MVC

We followed the MVC pattern for this project. The **model** updates the view and is manipulated by the controller. So in our case the model includes our app.py and it's fields, self.start, self.end, self.routing_method etc. It also consists of the code corresponding to the routing algorithms which is part of our strategy pattern using RoutingMode. The **view** updates the user, this is our OSM plot that displays the path on a map and tkinter GUI that provides statistics and information specific to their route. The **controller** is the initial prompt our app provides to the user via the command line. The user uses the controller to input the following information: the start and end location, routing algorithm of choice, maximize/minimize elevation, x% of shortest path, and preferred mode of transportation.

Techstack

- Python
 - pickle
 - networkx
 - pytest
 - abc (abstract base class)

- heapq
- OpenStreetMap ([OSM](#)) API
 - Graph data
 - Default shortest path
 - Nearest nodes for start and end locations
 - Plotting path on map
- [Open Elevation API](#)
 - Used to get elevation data for our map
- Tkinter
 - Displaying path statistics in window

Caching (map.py)

Since downloading map graphical data takes a few minutes, it is not efficient to download a copy of the map every time a user uses our app. Instead, we decided to download and store a copy of the Boulder, CO graph inside of a Python pickle file. This way, we do not waste time downloading a map every time resulting in a poor user experience.

In order to obtain a graph of the Boulder, CO map, we use OSM's `graph_from_place` API. We download the maps for driving, walking, and biking giving users options for transportation modes. `graph_from_place` returns a networkx multidigraph which contains nodes with latitude and longitude data and edges with "length" data (the distance between two nodes in meters). Since the nodes returned from OSM do not contain elevation data, we use the Open Elevation API to add elevation information to each node.

Once the graph is complete with elevation data, we store the graph object inside of a Python pickle file. Everytime a user runs the app, we "unpickle" the graph object and use it to run our algorithms.

Algorithms (routing_actions.py)

We have implemented three different algorithms for our app: Dijkstra, A*, and DFS.

When a user enters start and end addresses, we use OSM's `nearest_nodes` API to get the nearest start and end nodes for those addresses. Our algorithms will find a path between these two nodes.

Note: When start and end locations are too close, the API will return the same nodes for both addresses

Before jumping into our algorithm, we calculate the `max_length` our path can have using the variance the user inputs and the length of the default shortest path (returned from OSM API) → see `find_max_length` in `RoutingHelper`. If there is no default shortest path, that means there is no valid path between start and end and we return None.

Dijkstra

- Queue priority is total elevation from start node to current node unless the user chooses not to maximize or minimize elevation, then it is total distance from start to current node
 - Queue element = (total elevation from start node to current node, total distance from start to current node, current node, previous node)

A*

- Queue priority is elevation from start to current node + elevation difference between current node and end node unless the user chooses not to maximize or minimize elevation. In that case, A* will perform like Dijkstra (no heuristic)
 - Queue element = (f_elevation, g_elevation, total distance from start to current node, current node, previous node)
 - f_elevation = elevation from start to current node + elevation difference between current node and end node
 - g_elevation = elevation from start to current node

More Dijkstra/A*

- More or less follows basic Dijkstra/A* algorithm except
 - We don't visit a node if the total distance from start to that node is > max_length
 - We may visit a node more than once if we find a shorter path to it from a different node → need to re-process/re-calculate total elevation from this node
 - This is useful for "backtracking" as we may get stuck at nodes whose neighbors don't have a total distance <= max_length
 - We update the previous nodes table, elevations, and distances when we process the node, not when we add the node to the queue
 - We only update the node if we have not processed it before or if the total distance to this node from the new previous node is shorter than the one we have already seen

DFS

- Since basic DFS will not always find the best path in a weighted graph, we decided to use DFS to calculate every possible path between start and end that is within `max_length`
- Once we have all of these paths, we find the total elevation of each path
 - If user wants to maximize, we return the path with the highest elevation

- If user wants to minimize, we return the path with the lowest elevation
- If user wants shortest path, we simply return the shortest path and disregard elevation

Design Pattern

As we have to implement several different routing algorithms that are essentially trying to achieve the same behavior i.e., finding the shortest path between two locations with the possibility of either minimizing or maximizing elevation, we thought it would be appropriate to refactor our codebase to follow the strategy design pattern. We chose this behavioral pattern as the class behavior can change at runtime. So, we created objects which represent the different routing methods as strategies and a context object that will change according to the routing object. Thus, each routing object represents a strategy object which will implement the executing algorithm wrapped in the context object.

We have 4 files where the strategy design pattern is reflected, namely `context.py`, `routing_mode.py`, `routing_actions.py`, and `routing_helper.py`. Given the strategy design pattern, `routing_mode.py` instantiates the `RoutingMode` object which represents the strategy object, which in our case are `RoutingDijkstra`, `RoutingAStar`, `RoutingDFS`, and `RoutingBFS`. These strategy objects are then wrapped in the `Context` class, so depending on the context, the specific routing algorithm (routing action) will be executed. The actual routing algorithms are implemented within separate strategy objects in `routing_actions.py`. The class `routing_helper.py` contains all the helper functions needed to execute the routing algorithms.

Testing (test_routing.py)

In order to make sure our program is accurate, we wrote test suites for each of our algorithms and a test suite for our helper methods (getting total path elevation, path length, etc.). We used `'pytest'` to create our test suites.

To test, we generated 3 different graphs using `networkx` (`create_test_graphs.py`):

- Small uniform graph (5 nodes)
- Medium uniform graph (15 nodes)
- Small non-uniform graph (5 nodes)

We decided to write some basic test cases before implementing the algorithms but ended up adding more later on as we came up with more scenarios.

Each of the algorithm test suites has the same test cases for both small and medium graphs:

- Minimizing elevation

- Minimizing elevation with a variance in path length
 - Should return a path that has an elevation $<$ shortest path and has length less than or equal to $x\%$ of shortest path
- Minimizing elevation with no variance
 - Should return the shortest path
- Minimizing elevation when there is only one path from start to end
 - Should return the shortest path
- Maximizing elevation
 - Maximizing elevation with a variance in path length
 - Should return a path that has an elevation $>$ shortest path and has length less than or equal to $x\%$ of shortest path
 - Maximizing elevation with no variance
 - Should return the shortest path
 - Maximizing elevation when there is only one path from start to end
 - Should return the shortest path
- Shortest path
 - When user chooses to not maximize or minimize elevation, by default we return shortest path
- When no path exists
 - This test is done on the non-uniform small (5 node) graph
 - Since the other two graphs are uniform, there were never 2 nodes that did not have a valid path between them
- When start and end nodes are the same
 - Should return a path with just one node
 - Path length should be 0

Helper method tests:

- `get_path_elevation`
 - Tests if the total path elevation returned is correct
- `get_total_path_length`
 - Tests if the total path length returned is correct
- `get_path_from_previous_nodes`
 - Tests if the complete path returned from a dictionary that maps each node to its predecessor is correct (used for Dijkstra and A*)
- `find_max_length`
 - Tests if length returned is $x\%$ of the shortest path length
 - For example, if x is 50 and shortest path length is 10 \rightarrow return $1.5 * 10 = 15$

The reason we decided to test with two different-sized graphs is to test against a variety of inputs. Although our actual Boulder graph has thousands of nodes, we would not be able to write

effective test cases for such a big graph. With a smaller-sized graph, we were able to manually calculate the output path (knowing that is the correct output) and use that for our test cases.

We think these test cases sufficiently test many of the valid scenarios that could occur when a user uses our app.

Debugging

While developing our algorithms, we ran into many bugs such as the algorithm not finding a path even though it existed or calculating elevation wrong. In order to debug, we found it helpful to add print statements to the codebase to see values of specific variables throughout the program. Additionally, it was much more efficient to test the algorithms on our smaller graphs using the tests we had written. Using the smaller graphs, we were able to visualize how the algorithm was working and it became much easier to see where the bug was occurring.

Experimental evaluation

- The search algorithm experimental evaluation would summarize the results in terms of the performance of the algorithms (in terms of computation time).
- Experimental Evaluation: For performance analysis, we timed the runtime of each routing algorithm to see which algorithm returns the path faster. For user inputs we will use the same inputs to maintain controls for comparing the computation time on our algorithms: start address = 2505 Agate Rd, Boulder, CO 80304, end address = 47th St, Boulder, CO 80301, maximize elevation gain, $x \% = 30$. The only difference between each table will be the graph that we are pulling the data from based on the transportation mode.

Drive

Name of Algorithm	Computation Time
Dijkstra	0.0021719932556152344
A*	0.0022459030151367188
DFS	Timeout

Walk

Name of Algorithm	Computation Time
-------------------	------------------

Dijkstra	2.430320978164673
A*	4.805061101913452
DFS	Timeout

Bike

Name of Algorithm	Computation Time
Dijkstra	9.316324949264526
A*	3.7839720249176025
DFS	Timeout

Based on these values, we currently implemented Dijkstra's, A*, and DFS with a depth limit and understand how these three algorithms fit our use case. It is worth noting that if a better heuristic is used for A*, it could significantly bring down its runtime and potentially have it outperform Dijkstra in all cases. In the results, A* seems to work better against Dijkstra only for the Bike graph. This is likely due to the heuristic we chose, which is the difference of elevations between two nodes. We chose to add a depth limit to DFS so that the algorithm does not timeout by trying to explore every possible depth. However, we also implemented BFS, but don't think this algorithm fits our use cases in terms of it being valid on a weighted graph and finding the shortest route.

- User Feedback: For the user feedback, we asked two of our friends to run our application end-to-end. So, they would input 2 addresses from Boulder, Colorado, and add in the additional parameters that they'd want to include, such as the x percentile, maximize/minimize elevation gain, transportation mode, and preferred routing algorithm. We modified our application accordingly:

- Person A Comments: For inputting the preferred routing algorithm, it is a bit annoying to ensure that the inputs have to be lowercase. It is also not clear what "x %" means.

Response: We made sure to lowercase the strings inputted for the preferred routing algorithm, so users don't have to factor that in while doing inputs. We also changed the prompt for inputting x% to be "Enter what (x) percentage of the shortest path you're able to add travel: " instead of simply "Please enter x%: to make it more clear on what the user is actually inputting.

- Person B Comments: It's not completely clear that what you're showing on the GUI is meant to be the path returned since there's no other background behind the path.

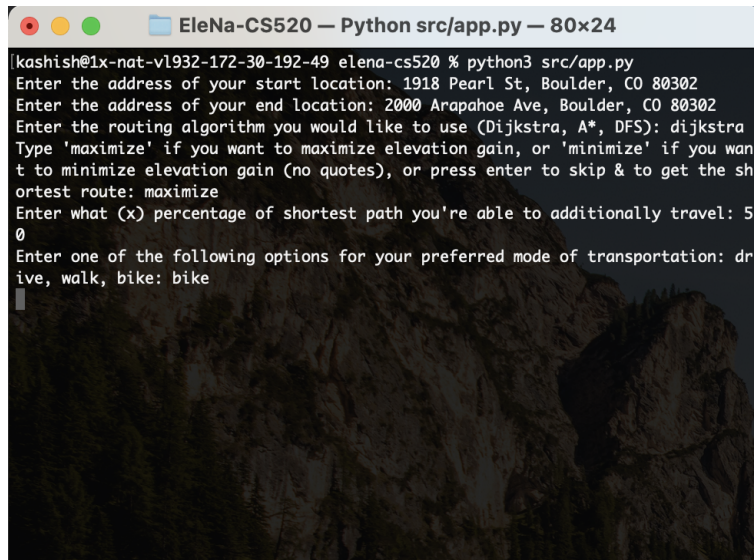
Response: Instead of simply returning the final path on the GUI, we chose to display the highlighted path over the subgraph of the area. This is more representative of traditional maps, so it is easier for users to interpret how the path would be over the area they have to travel.

GUI

Our implementation of the GUI features a graph of the Boulder map and a summary that includes data about the discovered path that was calculated by the algorithm.

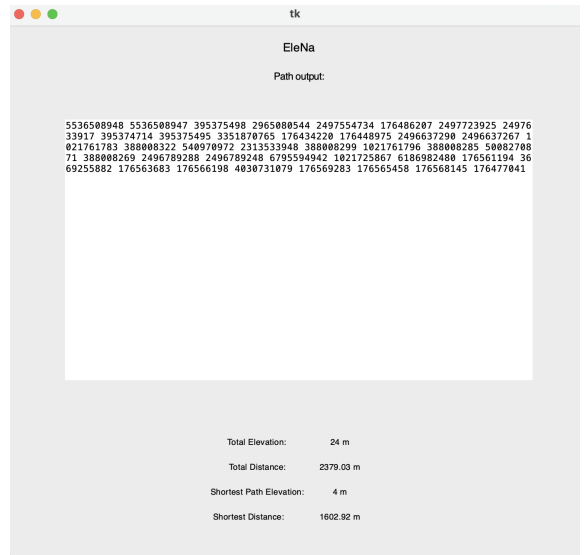
After a path has been generated by our algorithm, the GUI will generate the calculated graph using OSM's `plot_graph_routes` API. The path that was created by our algorithm will be highlighted in red while the default shortest path (from OSM API) will be highlighted in blue. The graph will also include the surrounding edges and nodes to allow the user to get a richer perspective of how the path fits on the graph. Additionally, we will have a Tkinter window displaying helpful information such as total distance and total elevation of our path vs. shortest path.

Example Run



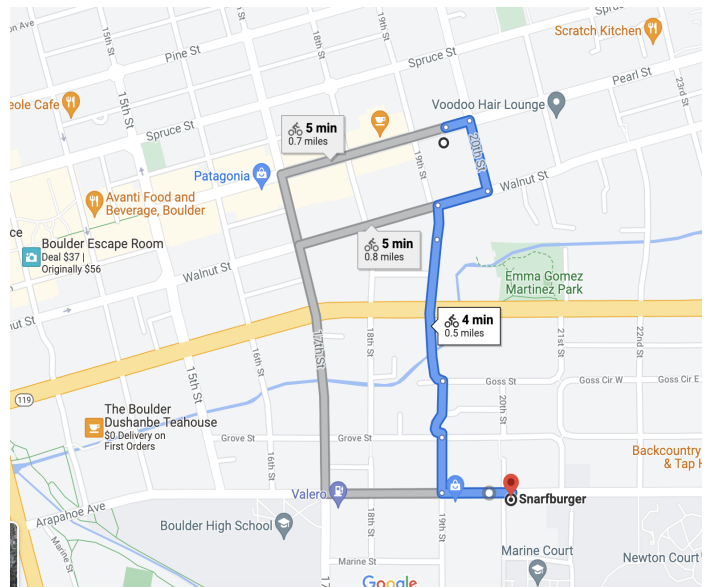
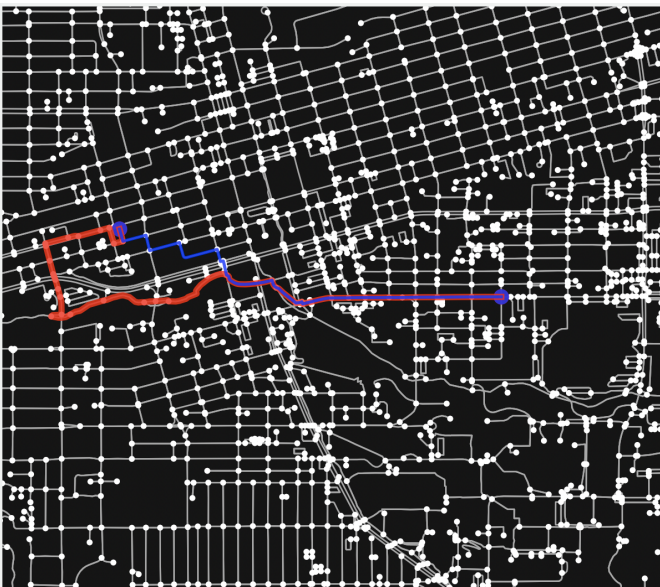
```
EleNa-CS520 — Python src/app.py — 80x24
[kashish@1x-nat-v1932-172-30-192-49 elena-cs520 % python3 src/app.py
Enter the address of your start location: 1918 Pearl St, Boulder, CO 80302
Enter the address of your end location: 2000 Arapahoe Ave, Boulder, CO 80302
Enter the routing algorithm you would like to use (Dijkstra, A*, DFS): dijkstra
Type 'maximize' if you want to maximize elevation gain, or 'minimize' if you want to minimize elevation gain (no quotes), or press enter to skip & to get the shortest route: maximize
Enter what (x) percentage of shortest path you're able to additionally travel: 5
0
Enter one of the following options for your preferred mode of transportation: drive, walk, bike: bike
```

User input



Path summary compared with shortest path summary

Plot returned by OSM API (red is our Dijkstra algorithm, blue is shortest path returned by OSM)



Zoomed in plot vs. path in Google Maps

Satisfied Requirements

- Agile Development
 - We decided to follow an agile methodology to wrap up this project as we decided that the project had a big overall goal but small, discernible tasks to reach this goal. So, we split up the main focuses of our project's timeline into multiple smaller phases. This method made it more effective to work and communicate remotely with different members of the team. It also allowed us to ask timely questions to the Professor as needed. Following this step-by-step methodology also allowed us to quickly adapt to any necessary changes that we realized we had to make along the way.
- Readability
 - We have created a README which explains the project, how to use it, and how to test it. Additionally, throughout our codebase we have added documentation for each of our functions/methods and single-line comments to explain more complex code. We also focused on proper variable and function naming and making sure to cite other developers' code if we found it helpful when writing our own code.
- Extensibility
 - We adopted the strategy design pattern by instantiating our routing algorithms as strategies (see Design section for more details). This makes it easier for us or other developers to add another strategy to the `routing_actions.py` file where we are defining our strategy objects. For eg., if a developer wants to add Bellman-Ford as a routing algorithm, then to declare and define the strategy, they will simply have to create a `RoutingBellmanFord` class, which implements the abstract strategy class `RoutingMode`, in the `routing_actions.py` file and instantiate the context where we run the algorithm (after reading user inputs).
- Error-handling
 - We made sure to handle errors when a user inputs something incorrect. For example, we ask the user to type "Dijkstra", "A*", or "DFS" for the algorithm type. If a user does not enter one of the three, then we prompt the user to enter an algorithm again. We do this for all of our user inputs. Additionally, we use the OSM API to get the latitude and longitude of the start and end addresses the user inputs. If the API throws an error, we make sure to catch and relay to the user that the addresses they entered were invalid.
- Usability
 - We decided to go with a very simple UI for users to quickly read a few prompts and be able to get their desired routing outputs effectively. We also display a subgraph of where the found route occurs, so that users have a more visual representation of the path that they are recommended to take based on their

requirements. This makes it more user-friendly vs. simply displaying a path of addresses.

- Testability
 - We have separate functions for each of our algorithms and have separated much of our reusable code into helper methods. This allows us to individually test parts of our program. We were also able to add print statements throughout our program (now removed) to understand why/how tests failed.
- Our stakeholders
 - Developers (Us)
 - Professor Conboy & TAs
 - App Users (hikers, bikers, walkers, drivers, etc.)

Challenges

- Elevation data
 - Since the nodes returned from OSM do not contain elevation data and the `add_node_elevations` OSM API is deprecated, we needed to figure out a way to add node elevation data to the graph. OSM has an `add_node_elevations_google` API which uses the Google Maps Elevation API to add node elevation, but since the Google API is not free, we decided against using it. Fortunately, we found an alternative open-source elevation API called Open Elevation. OSM is open-source, so we used their code for `add_node_elevations_google` to help us implement the same thing with the Open Elevation API.
- Gui
 - We faced an issue where we were unable to see the graph in a wider perspective and were able to use the OSM API to configure the graph such that it shows the surrounding edges as well as nodes. With this, we can see exactly which routes the algorithms picked.
- Algorithms:
 - We initially implemented the normal DFS and BFS algorithms to calculate the shortest path. However, we realized that BFS can't be used on weighted graphs and DFS will not always find the best path in a weighted graph. So we implemented a new DFS algorithm that finds all the paths from the start to end node within a max length, and then finds the path which maximizes/minimizes elevation otherwise retrieves the shortest path.
 - While testing out Dijkstra and A* we found that our algorithm would sometimes not return the complete path from start to end and instead just return a path from start to some node in the middle. We realized this was happening due to the max_length constraint. Once we reach that node in the middle, we have already reached our max_length and cannot explore nodes further. After a few days of

debugging and trying out various conditions, we concluded that it was necessary to re-visit a node if a shorter path could be found to it from a different node. This way, we would be able to “backtrack” in a way in case we get stuck in the middle due to the constraint.

- Merge conflicts
 - We ran into merge conflicts a couple of times as we had multiple people making edits on the same file. To resolve these conflicts, we communicated and evaluated what changes needed priority over others and what needed to be kept/removed.