MapLab r177a

Vedang Patankar
Benjamin Chang
Group #8
E177 Spring 2017

Abstract:
Our project seeks to create a path finding program for Berkeley, tailored to cyclists and walkers alike in traversing the maze that is our campus.

Introduction

Project Description:

The idea for MapLab r177a was originally inspired by our desire to explore and find optimal bicycling routes throughout the university campus. From our personal experience, despite our familiarity with campus, we still do not know of every nook and cranny on campus, and sometimes there exists a more optimal route that we are unaware of. Given the large number of possible routes and numerous biking restrictions such as dismount zones, finding the most optimal path from point A to point B on campus clearly isn't always as easy as it seems. This is especially true for new students or visitors who are still unfamiliar with the campus layout.

We decided to run with this idea of mapping out all possible biking paths throughout campus, and expand the functionality of the program to include all paths throughout campus. Of course, a robust route finding program already exists in the form of Google Maps' navigation features. Although the same fundamental graph traversal algorithm is at the core of both programs, Google's map data is unfortunately inaccurate when it comes to pedestrian paths, especially on college campuses. Information about small footpaths on campus is often absent from Google Maps, despite being vital to the objective of path finding. Thus, our goal of MapLab r177a is to overcome the shortcomings of Google Maps and build a lightweight, specifically tailored program to facilitate path finding on the Berkeley campus.

Purpose:

Our design goals:

- **Improve upon Google's map data**, and possibly recreate it from scratch
- Provides a solution for students to **optimize their day to day route** between classes
- Provides a fast solution for users to **calculate shortest path** between two locations on campus
- Allows users to choose between the two most common ways of getting around: **walking vs biking**
- Provides **useful information regarding the calculated route**
- Create an easy to use, aesthetically pleasing **GUI interface**
- Create a **modular, well documented program system design** that is easy to maintain and expand upon

Our target demographic includes:

- Freshman/transfer students
- Existing members of campus community
- Visitors

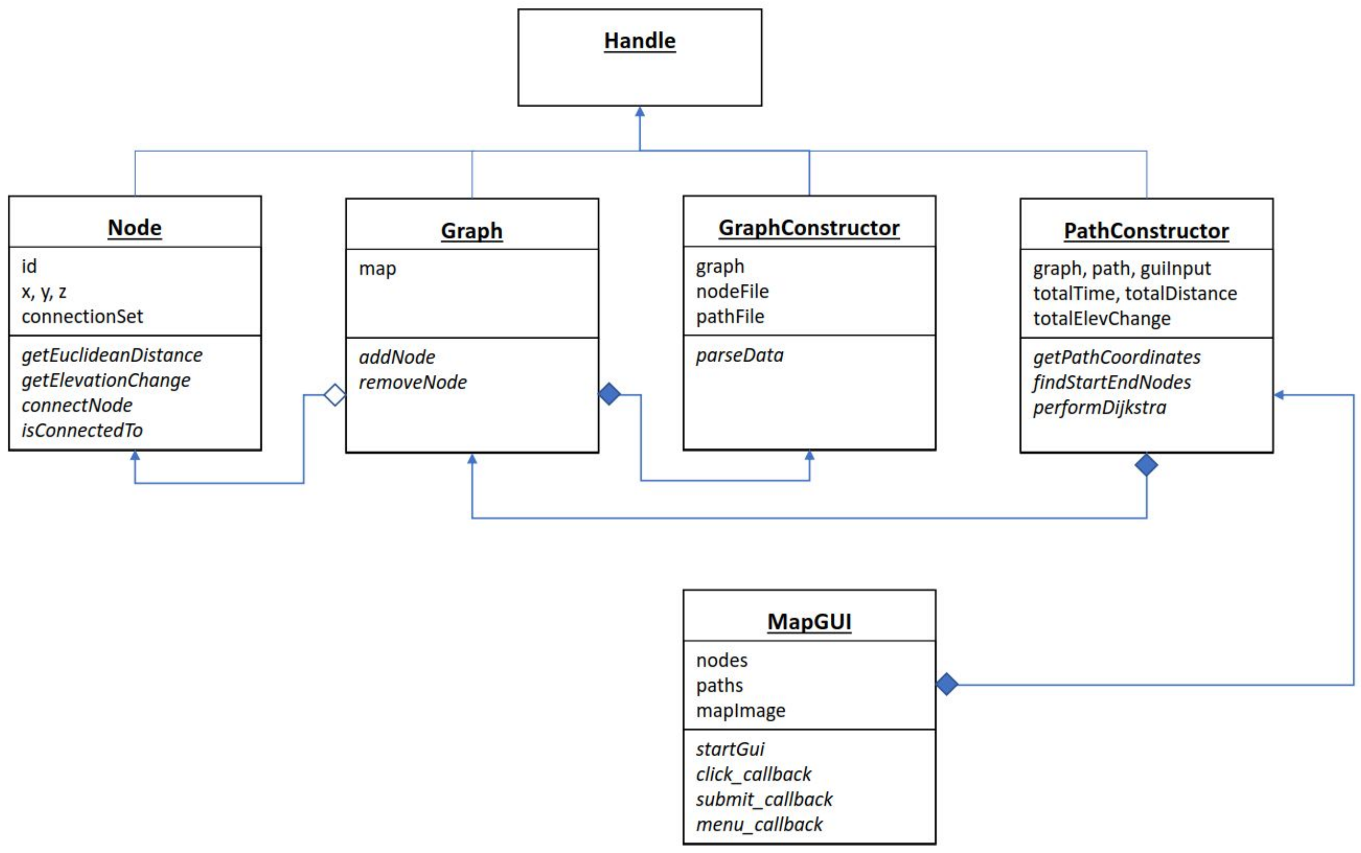
Functionality:

Features and functionality implemented by our program include:

- **Multiple GUI input options** to select the desired start and end locations - **mouse click input, textbox search, and drop down menu**
- Users can **favorite select locations**, which appends these locations to the top of the drop down menu
- Users can specify whether they are **walking or biking**
- GUI displays **minimalistic and informative map** that labels buildings/features on campus, supports zoom, and displays common start locations off campus.
- GUI **interpolates calculated path** onto the map in easy to understand way
- GUI outputs useful information about the path, including **total distance, total time, total elevation change, and an elevation change plot**
- Potential for expandability with **separate GUI class** that provides convenient interface to **create custom graph objects**

Program Structure

UML Diagram:



Class Documentation:

1. Nodes Class (BEN) - Represents a map node object, which holds information about its location and neighbors

< handle	Name	Description	Inputs	Outputs
Properties:	id x, y, z coordinates connectionSet isConnected	Name/ID of location long/lat/elevation set of all connected neighbors boolean		
Methods: (public)	Constructor getEuclideanDistance getElevationChange connectNode isConnectedTo	constructor method returns distance between nodes returns difference in elevation connects two nodes together checks if nodes are connected	id, x, y, z node1, node2 node1, node2 node1, node2 node1, node2	obj double double ~ boolean

2. Graph Class (BEN) - Represents graph object, which manages all the nodes.

< handle	Name	Description	Inputs	Outputs
Properties:	map biking	maps ID's to nodes boolean		
Methods: (public)	Constructor addNode removeNode	constructor method adds <ID,node> to map removes <ID, node> from map	node/path file node1, node2 node1, node2	obj ~ ~

3. GraphConstructor (BEN) - parses through node and path dataset to construct graph object

< handle	Name	Description	Inputs	Outputs
Properties:	graph nodeFile pathFile biking	Graph object node data file path data file boolean		
Methods: (public)	Constructor parseData	constructor method parse data files, populates graph	graph, node/path file	obj ~
Methods: (private)	extractNodeData extractPathData validPathType connectNodesInPath removeIsolatedNodes	extracts node data extracts path data returns if path is valid connects all nodes in array Purges disconnected nodes	nodeDataRow pathDataRow bikeable, biking node array ~	id,x,y,z id,arr,bikeable boolean ~ ~

4. PathConstructor (BEN) - instantiates graph, performs underlying calculations, and interfaces with GUi

< handle	Name	Description	Inputs	Outputs
Properties: (public)	graph path guiInput totalTime totalDistance totalElevChange	Graph of system from input files Array of nodes in calculated path User input data Total time estimate of path Total length of path Total elevation change of path		
Properties: (Constant)	SCALING_FACTOR AVG_WALKING_SPEED	Unit convert from pixels to ft Avg walking speed in ft/min		
Methods: (public)	getPathCoordinates	returns x,y,z coord of nodes in path	~	[vector, vector]
Methods: (private)	getElevationChange getTotalTime findStartEndNodes performDijkstra formatString getDistanceToNode convertPixelsToDistanc	Calc total elevation change Estimates total time of path Finds nearest start/end nodes performs Dijkstra’s Algm. remove spaces, lowercase calc distance from coord to node converts pixels to ft	~ ~ ~ start+end Node string node, x, y double	double double [node, node] pathNodes string double double

	e			

5. MapGUI (VEDANG) - Implements and controls the entire front end graphical user interface

	Name	Description	Inputs	Outputs
Properties: (public)	Nodes Paths MapImage	nodeFile string name pathFile string name Map image file		
Properties: (private)	Default List	default list of locations for dropdown menu		
Methods: (public)	StartGui	Starts and controls the GUI	MapGUI object	~
	Click_callback	Handles event changes for the mapclick option.	img,MapGUI object	~
	submit_callback	Constructs the PathConstructor and retrieves the path values, then passes them to pathPlotter.	obj	~
	menu_callback	Handles event changes for the menu option.	obj	~
	pathPlotter	Plots the paths from start to destination, as well as a mini elevation plot.	coordsCell, mapAxes, start, final, miniplotaxes	~

Implementation Details

Back End Theory and Implementation: Ben

The back end is composed of four distance classes: Node, Graph, GraphConstructor, and PathConstructor. The back end's purpose is to overlay a graph of nodes on top of the map image and provide a functions to traverse through the nodes and find the shortest path between two given nodes.

Each Node object represents a physical point on the map. It has an ID, which represents its building name (if the node represents a building) or tag number (if the node represents a path). It also has properties: x, y, and z properties, which represent the node's physical latitude, longitude, and elevation, respectively. Lastly, the node contains a list of all the other node objects that it is connected to. Every node, in combination with every node's connection list, effectively represents the entire graph structure.

The Graph object serves to physically keep track of all the nodes in the graph structure. It manages the nodes in a containers.Map data structure, which maps the node ID's to the node objects themselves. The constructor for this class instantiates a separate GraphConstructor object, which helps construct the Graph object.

The GraphConstructor object is a helper class that helps construct the Graph object. It parses through the nodes.mat and paths.mat GUI input data (which we populated from scratch using GraphConstructerGUI) to create every Node object, add them into Graph's map data structure, and connect all the nodes together.

The PathConstructor object performs all the underlying graph traversal algorithms and directly interfaces with the GUI. It constructs a Graph object from the GUI input data and has functions that locates the correct start and end nodes in Graph, perform the graph traversal algorithm, and returns the relevant data and results to the GUI. The graph traversal algorithm employed was Dijkstra's Algorithm, which traverses breadth first from a start node until the end node is found. In order to accurate estimate the total time of the path, we employ two strategies: Naismith's Rule and VAM. Naismith's Rule states that for every 2000 feet of elevation gain, we would append 1 hour of hiking time to our total. VAM is the equivalent of Naismith's Rule for biking.

Front End Theory and Implementation: Vedang
The Front End was largely comprised of the central GUI itself, which features entirely custom graphics. One of the fundamental goals in designing the GUI was to create an interface that had a significantly sleeker and modern look than the default Matlab UI controls. A challenge in the GUI design was constructing the nodes and paths, which was an arduous process that involved the creation of two separate GUI scripts to ease the process.

Our intent was to design a detailed map that displayed as many landmarks and paths on campus as possible, and thus the map image itself was constructed almost entirely from scratch in Adobe Illustrator - we were displeased by the quality of most of the available campus maps, and didn't want to copy the Google Maps API as it does not display each path on campus.

Two scripts were created as an intermediate step to provide the final GUI with data. The first script obtained the necessary elevation data, plotted the graph, and displayed a UI that permitted us to enter the pixel location on the map axes and a name in order to generate nodes. The second script displayed the map and the generated nodes by ID, and permitted the user to determine paths by entering a series of nodes. The script then assembled paths and performed interpolations on their to display the paths to a desired accuracy.

The MapGUI class contains three public properties, Nodes, Paths, and mapImage, which contain the .mat filename for the nodes, the .mat filename for the paths, and the name of the map file. It additionally holds a private property, Default List, which contains the list of locations displayed in its dropdown menu.

Its method StartGUI assembles the GUI itself, by first creating a figure and then plotting the mapImage in a section. The controlPanel section contains the options for start and destination locations, and a Favorite option, which allows the user to Favorite the current pixelposition and add it to the head of the DefaultList property such that it shows first in the dropdown.

The Biking/Walking radio control generates different paths based on its value, taking into account the non bike accessible paths on campus.

The submit button callback packages the data acquired from the start/destination controls and from it creates a PathConstructor object, which returns the path from start to finish. This is then plotted, along with a plot of the elevation change over time.

Manual of Operation

Step 1: User Input

Users have the 3 different options of inputting data into the GUI: mouse input, text search, or drop down menu

In addition, specify whether you are walking or biking

- Tip: Make sure the type of input you enter for the start field matches the end field!

Step 2: Favorite Locations (Optional)

Users can add locations to their favorites list by pressing the “Favorite” button. This automatically saves this location for future reference, and displays it at the top of the dropdown menu.

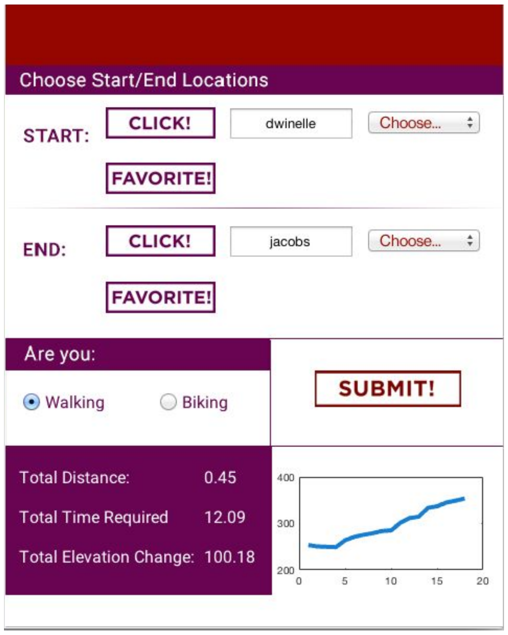
Step 3: Hit SUBMIT!

Calculate your optimal path and behold the results!

Step 4: Check out useful information about your path in the space below.

You can find out about total distance of the path, a time estimate (based on average walking/biking speeds), a total elevation change, and a handy plot of the elevation change along the length of the path.

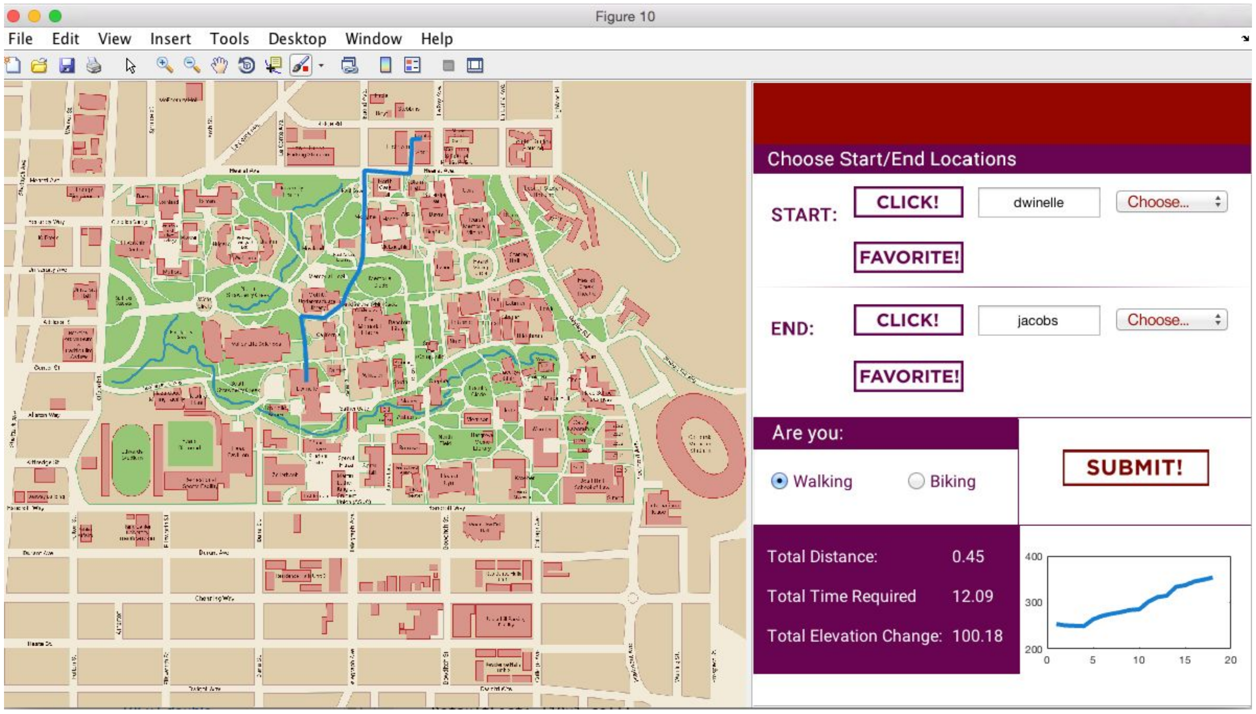
Step 5: Repeat!



Total Distance: 0.45
Total Time Required 12.09
Total Elevation Change: 100.18

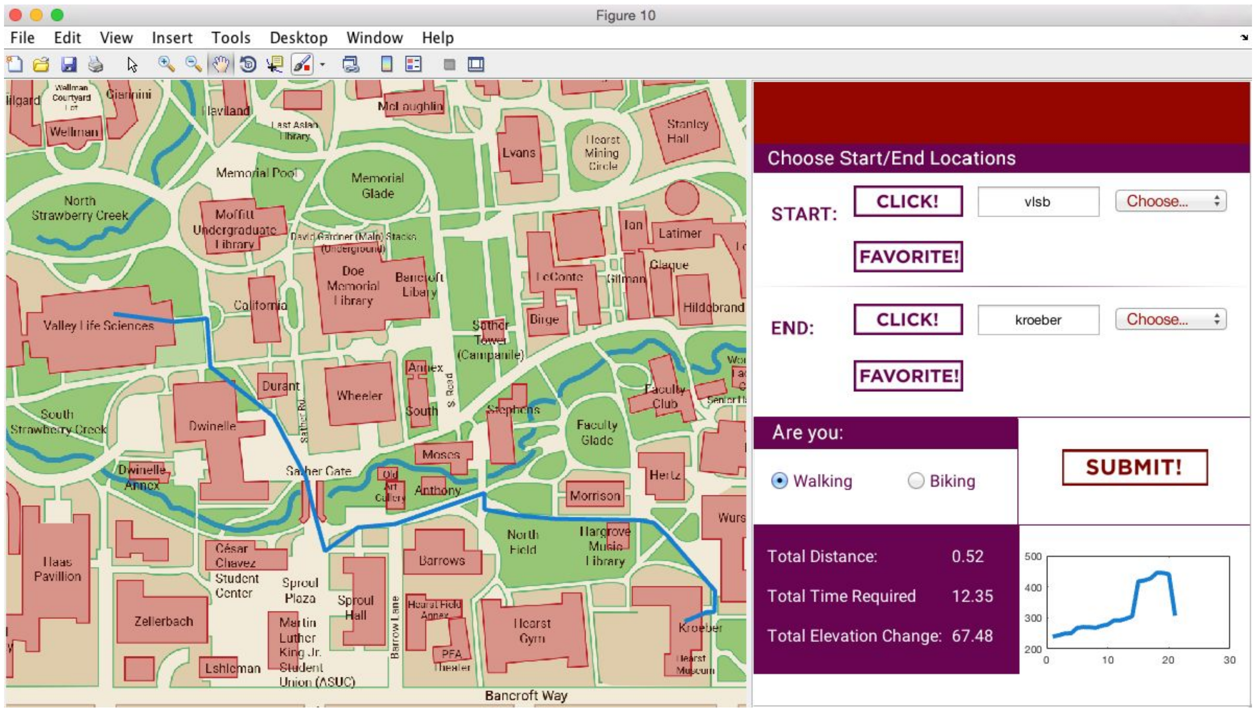
Example Outputs

Navigate from Dwinelle to Jacobs (walking)



Total Distance: 0.45
Total Time Required 12.09
Total Elevation Change: 100.18

Ex 2) Navigate from VLSB to Kroeber (walking)



Choose Start/End Locations

START: CLICK! vlsb Choose... FAVORITE!

END: CLICK! kroeber Choose... FAVORITE!

Are you: [✓ Walking] Biking

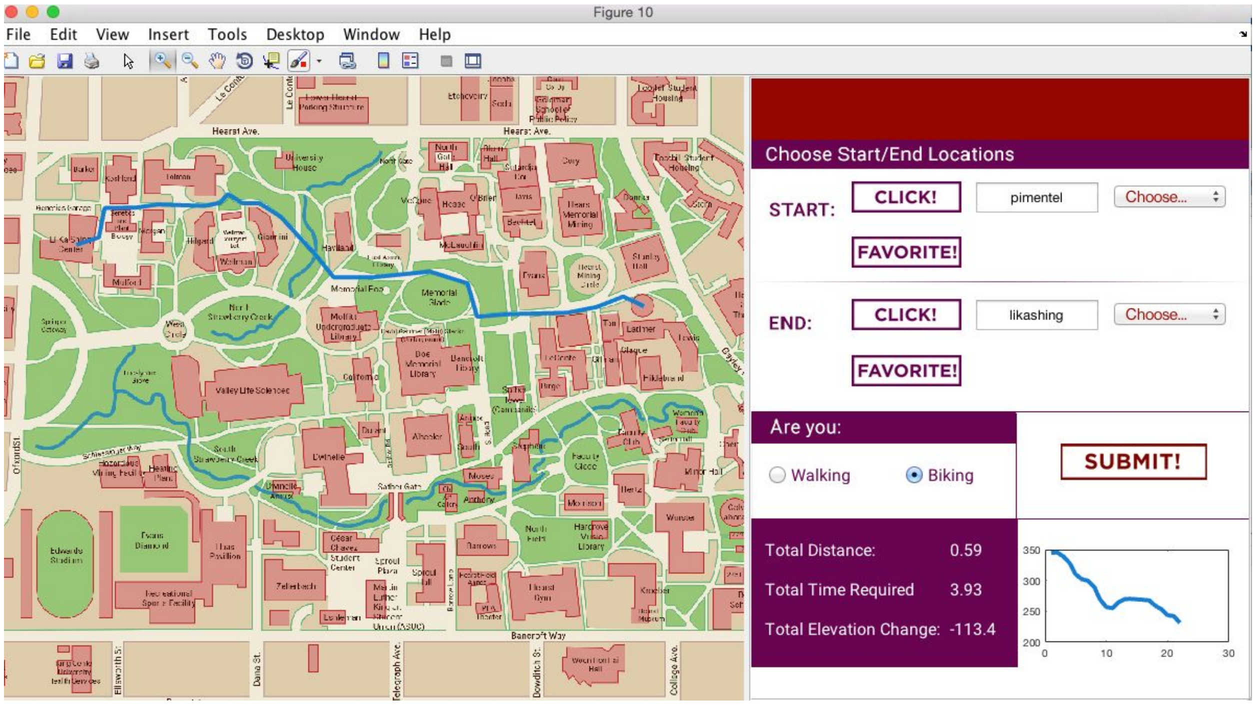
SUBMIT!

Total Distance: 0.52

Total Time Required 12.35

Total Elevation Change: 67.48

Ex 3)Navigate from Pimentel to Li Ka Shing (biking)



Choose Start/End Locations

START: CLICK! pimentel Choose... FAVORITE!

END: CLICK! likashing Choose... FAVORITE!

Are you: Walking [✓ Biking]

SUBMIT!

Total Distance: 0.59

Total Time Required 3.93

Total Elevation Change: -113.4

Project Team Roles and Responsibilities

Ben:

- Organized and documented ideas, class structure, and functionality from start to finish
- Researched graphs, graph traversal algorithms, and implemented the back end of the project (including the Node, Graph, GraphConstructor, and PathConstructor classes)
- Designed robust, yet simply to use interface to communicate with the GUI
- Wrote test scripts to rigorously test the functionality of algorithm, veracity of the Graph/Node objects, and the correctness of the Graph's underlying paths
- Cleaned up and thoroughly commented project code.
- Tested functionality of project
- Created UML Diagram
- Contributed to the project report
- Contributed to presentation slides

Vedang:

- Created the custom vector Berkeley map used in the GUI from a low-res image
- Researched geographical elevation data, recorded elevation data manually by seeding a maps API and creating a meshgrid/interpolation of Berkeley campus elevation values
- Made a test GUI to create individual nodes by using general knowledge of campus routes and Google Maps data (over 500 nodes)
- Made a test GUI to create paths from nodes and determined whether paths were accessible by bike
- Repeatedly tested for isolated nodes/ broken paths
- Made the central Map GUI that interfaces with the backend including custom graphics for GUI use and the path plotting methods
- Contributed to the project report
- Contributed to presentation slides

Future Improvements

Our project was implemented following a detailed design documentation that provided a high level framework for us to organize our system design around. This lends our project to a very modular program structure, and thus it is easy to build additional features and improvements upon the existing framework. However, because of the limited time constraints on our project, here is a list of future improvements, features, and optimizations that we plan to potentially implement in the future to increase the capabilities of our project:

- Optimize graph traversal algorithm by employing a heuristic that guides the algorithm in the direction of its target, which improves upon the inefficient bread-first traversal strategy that our simple implementation of Djikstra's Algorithm performs.
- Implement different user defined strategies to traverse the graph. Rather than simply calculating the path with shortest distance, modify the heuristic to calculate based on other goals. These can include: calculate the path with mildest gradient, calculate the path with minimum time, avoids congested areas, etc.
- Optimize the way the GUI communicates and interfaces with the back end computation class objects. Current implementation reinstantiates these objects upon every call of the GUI's submit button, which negatively impacts run time performance.
- Reconstruct our node and path data in the graph representation of campus to increase accuracy and allow for smoother interpolation when constructing path.
- Implement feature for multiple destinations or "waypoints"
- Integrate building accessibility and building hours when considering allowable paths
- Export the program into a mobile app

References/Acknowledgements

Matlab Documentation
Graphs and Graph Traversal: https://en.wikipedia.org/wiki/Graph_(abstract_data_type)
Pathfinding Algorithm (Dijkstra'): https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
Naismith's Rule: https://en.wikipedia.org/wiki/Naismith%27s_rule
VAM: https://www.cicerone-extra.com/vam-a-naismiths-rule-for-cyclists
Elevation Data: https://www.daftlogic.com/sandbox-google-maps-find-altitude.htm