CSC111 Project: Navigating Your *Way* Through the Sub*way*

Alissa Lozhkin, Ayanaa Rahman, Jennifer Cao, Katherine Luo

Friday, April 16, 2021

Introduction

Subway systems are a very important mode of transportation. Subway systems provide many benefits, they are environmentally friendly, convenient, safer than other forms of transportation (e.g, cars) and cost effective (Silverwitz). People use the subway for one reason, and that is to get to another place. And the majority of people want to get to their final destination in the fastest time possible. People can achieve this by planning. There is a clear relationship between the population size of a city and the complexity of the city's subway (Florida). Areas with larger populations tend to have a greater number of subway stations (Florida). And as the subway becomes more and more complex, the harder it is for a subway user to plan their subway journey quickly. With our project, people will be able to plan their subway trips with ease and efficiency. We will be doing this by examining two different subway systems. These subway systems will serve as a model for our reference when solving this problem, but our project is applicable to any other subway system.

Finding the shortest path between subway stations is crucial to the overall experience of a subway user. Moreover, the complexity of large subway systems makes it difficult for the subway user to look at the system map and figure out the fastest route, especially if they want to avoid a station. Our project aims to provide subway users with the most efficient routes.

We chose to apply our project to the Vancouver SkyTrain subway system in British Columbia, Canada. We chose this system specifically because it is complicated enough to demonstrate our knowledge of graphs. Because there are many stations in the Vancouver SkyTrain subway system, we know it might be difficult for a user to plan their subway journey. By creating this project, we want to make the planning of a subway journey easier and more efficient. We also applied our project to the Kobe Subway system in Japan, solely to demonstrate how our project can be applicable to any subway system when given the appropriate dataset.

Our research question is: **How can we simplify the subway trip planning process of a subway user?** We implemented our project with this question in mind, focusing primarily on finding the shortest path between two subway stations and providing the user with the option to avoid stations. To determine this shortest path, our project will take into account the number of stations travelled to get from one station to another.

Datasets

The first dataset we used is from GitHub. The dataset was created by Dmitry Shkolnik and is a csv file called "skytrainstations.csv". This dataset includes information about the SkyTrain subway system in Vancouver, BC, Canada. Information in this dataset includes the stop id, station stop names, the longitude and latitude of the stations, zone ids and the location (Shkolnik). Within this dataset, we only used the station names, latitudes and longitudes of the stations for this project. We altered this dataset by deleting every duplicate row that contained the same station name twice. We added columns for the x- and y-coordinates of where the stations are located in the pygame window as well as the neighbours of each station. We called this file "vancouver_subway.csv". Within this file, the neighbours are separated using a comma (no space in between).

The second dataset is a csv file we created and called "kobe_subway.csv". We manually found the required data for this file, as we could not find it elsewhere. This dataset includes columns for the station names, the latitude and longitude of the stations, the x- and y- coordinates of where the stations are located in the pygame window, and the neighbours of each station. We used Google Maps to find the longitude and latitude of each station (Google).

Computational Overview

The ability to understand and visualize graphs was an integral part of our project. In our project, a graph represents a subway system (ex. the Vancouver SkyTrain subway system or the Kobe subway). The classes for the subway system are stored in the file "subway_system.py". Each vertex of the graph represents a subway station and each edge connects two neighbouring stations together. We created a private `_Station` class that stores the private instance attributes for the station. The "major" attributes are the name (represented as a string), latitude and longitude location (represented as a tuple, storing coordinate points), and its neighbouring stations (represented as a set of strings). We also made a public graph class `Subway` that has private instance attributes to keep track of the stations in this metro system, mapping the station's name to its corresponding `_Station` vertex.

In the same "subway_system.py" file, we created a method called `shortest_path` in the graph class `Subway` that determines the shortest path between station A and station B. It calls a method in the `_Station` class called `possible_paths` that determines all the possible paths from station A to station B using the graph traversal pattern taught in lecture (Liu). The shortest path was evaluated based on the least number of stations visited in the path. We also made sure not to visit any stations that a subway user wanted to avoid. If no path was found between station A and station B, an empty list is returned.

The "subway_system.py" file also has a method in the `Subway` class called `update_all_stations`, which loops through all the stations, finding the station the user selected (`mouse_position`) and changing the station to its correct colour (yellow if it's a station they want and red if it's a station they want to avoid). The method `update_selected_station` does the same thing except for only one station and takes in a `name` parameter for the name of the station rather than the user's mouse position. These methods call the `update` method in the `_Station` class that uses the `pygame.Rect` method `.collidepoint` to determine whether a station has "collided" with the user's mouse. In addition, there is also the `draw_stations` method that makes use of `pygame.sprite`. The `_Station` objects inherit from the `pygame.sprite.Sprite` class so that we can use the`pygame.sprite.Group` method `.draw` in the `Subway` class to draw the stations onto the pygame screen (Pygame). Note that a `Subway` class is created using the `read_csv_data` function found in the "data_wrangling.py".

For visualization, we used the python library `pygame` and `plotly`. The file "plotly_visualization.py" contains the function "plot_shortest_path" that plots the shortest path between two station on a real world map using latitude and longitude locations of each station (Plotly). The pygame visualization is found in the file "pygame_visualization". The most "important" function used in this file is the `run_visualization` function that calls helper functions in the same file (these functions draw on the pygame screen), and calls functions in the "pygame_mouse_click_handling.py" file (but more on this file later). The original map was the canvas we operated on. Drawing the background involved drawing the sidebar and uploading the image of the subway system map using the `pygame.image.load` function (Pygame). Running the visualization brought all the pieces together, drawing all the buttons that are available to the user's disposal ("GO!", "RESET", and "MAP VIEW" buttons), drawing all stations on the map, and drawing all required texts. A 'Sorry, no path was found.' text is drawn on the sidebar of the visualization when no path can be found between two subway stations. The local variables `selected_stations` (a list of length at most 2) and `removed_stations` (a set) keep track of the stations the user wants and the stations the user wants to avoid, respectively. This file uses adapted code from CSC111 Assignment 1 (CSC111 Department).

In order for the user to click on subway stations, buttons needed to be incorporated into our project. Buttons were represented in a similar way as stations were and are found in the "pygame_buttons.py" file. A private class `_Button` was created as a child of the `pygame.sprite.Sprite` class. A public class, `Buttons`, was created to keep track of all the buttons used in the pygame visualization. Each button has private instance attributes, including colour (represented as a string), image (represented as a `pygame.Surface` object), and `pygame.Rect` object, which represents the rectangular area that the button is enclosed in (Pygame). The public `Buttons` class contains a method called `draw_buttons` that draws the buttons onto the pygame screen using the `.draw` method of `pygame.sprite.Group`. There is also a method in the private `_Button` class called `was_pressed` that calls the `pygame.Rect` method `.collidepoint` to determine if the user pressed that button (Pygame).

The "pygame_mouse_click_handling.py" file contained functions that handle all mouse click events (i.e., when `event.type == pygame.MOUSEBUTTONDOWN`). The most "important" function in this file is the `handle_mouse_click` function, which calls helper functions to handle specific events depending on what/where the user pressed. The `handle_left_click_station` helper function checks whether the user left-clicked a station and "lights up" that station, turning it from a grey colour to a yellow colour. This function will add the station to the local variable `selected_stations` of the "pygame_visualization.py" file and remove it from the `removed_stations` if the station is in this set. It will also enable the "GO!" button, turning it from a grey to a blue colour if the user has selected two stations they want to go to. The `handle_right_click_station` helper function checks whether the user right-clicked a station and "lights up" that station, turning it from a grey colour to a red colour. This function will add the station to `removed_stations` and remove *all* instances of it from the `selected_stations` if the station is in this list (once or twice). Note that this can happen because we allow the user to start and end at the same station. The function will also disable the "GO!" button, turning it from a blue to a grey colour if the user has no longer selected two stations they want to go to.

In the same "pygame_mouse_click_handling.py" file, the `handle_click_reset` function resets the pygame visualization if the user pressed "RESET", disabling all buttons (turning them grey), deselecting all the stations (turning them grey as well), and clearing `selected_stations` and `removed_stations`, and drawing a rectangle over the 'Sorry, no path was found.' message, if necessary. The `handle_click_go` determines the shortest path between two stations if the user pressed "GO!". It displays this path on the pygame screen and may enable the "MAP VIEW" button if a path is found between the two stations. Finally, `handle_click_map_view` runs the plotly visualization of the shortest path when the user presses the "MAP VIEW" button. Note that all these helper functions call the necessary "update"- related methods of the `Subway` class and `Buttons` class to change the colour of the stations or buttons. The "draw" methods of these classes are called to display these changes. The images for the stations and the buttons come from Game Art 2D (Game Art 2D).

Instructions For Dataset Download and Running the Program

We used two libraries for our project: pygame and plotly. To install pygame, click on 'File' in the top-left corner and go to Settings. Click on 'Project: doc file' and you should see a drop-down menu. In the menu, click on 'Python Interpreter'. In the bottom-left corner, you can click '+' and search for 'pygame'. Once you have found it, you can click it and install it. Make sure you install version 2.0.1. You can specify the version by clicking on pygame and at the bottom clicking on the specify version box and scrolling through the options. To install plotly, follow the same steps for installing pygame only this time searching for 'plotly'. Make sure you install version 4.14.1.

There are three folders to be downloaded in zip file called "project.zip". The folders can be downloaded using the claim ID: **wr4T7fQkDXuRUf2T** and claim passcode: **iaNcbNm6vyNMeVWS**. The folders should be on the same "level" as the main file "main.py". One folder is called "data" and contains the following three datasets:

1. skytrainstations.csv
2. vancouver_subway.csv
3. kobe_subway.csv

Note that "skytrainstations.csv" is the original dataset, while "vancouver_subway.csv" is the modified dataset, and "kobe_subway.csv" is a dataset we created from scratch.

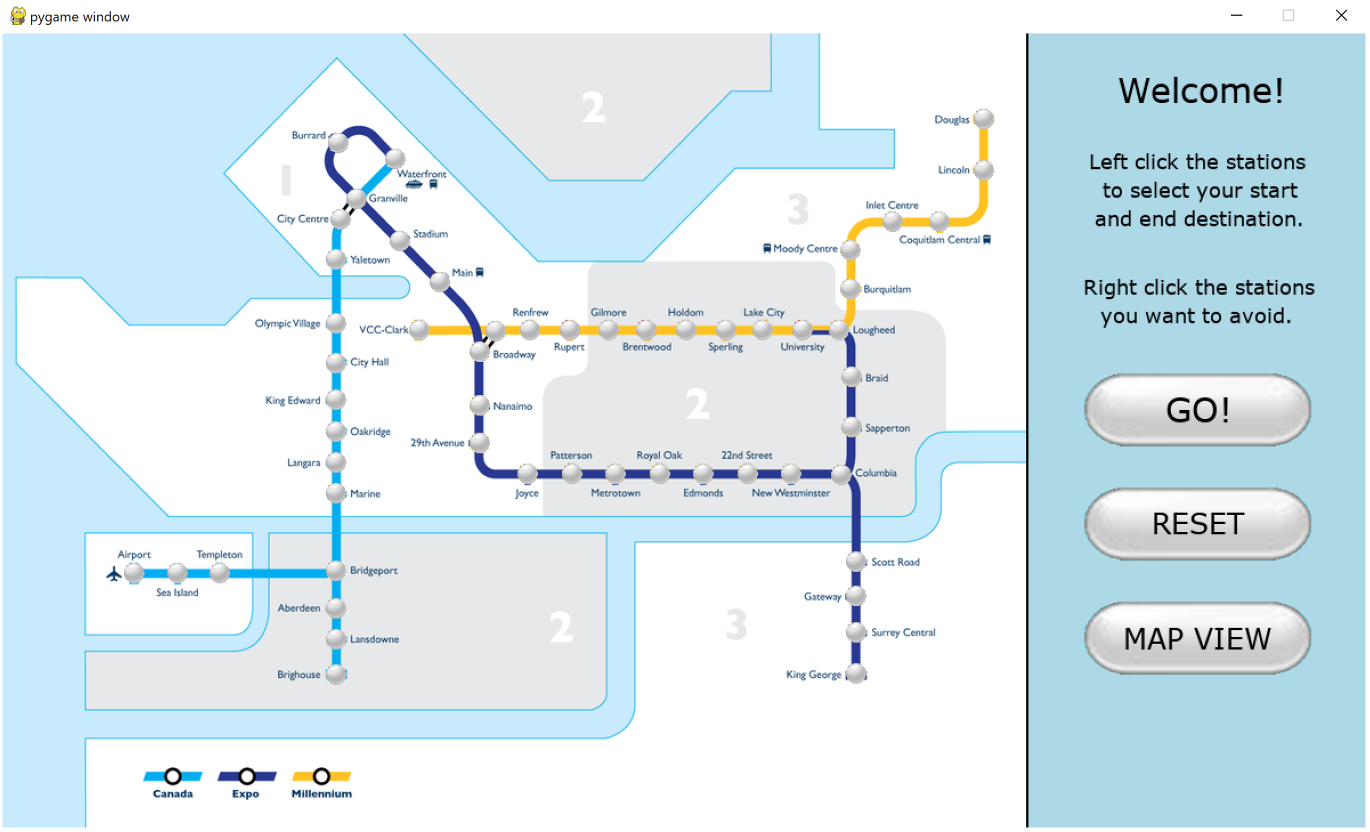
Make sure that there are also two other folders called "sounds" and "images". The folder "sounds" should contain the following:

1. background_music.mp3
2. click.mp3
3. path.mp3
4. complete.mp3

The folder "images" should include the following seven images:

1. vancouver_subway_system.png
2. kobe_subway_system.png
3. blue_button.png
4. grey_button.png
5. grey_circle.png
6. red_circle.png
7. yellow_circle.png

After downloading these three folders and running “main.py”, you should see this pygame window pop up:



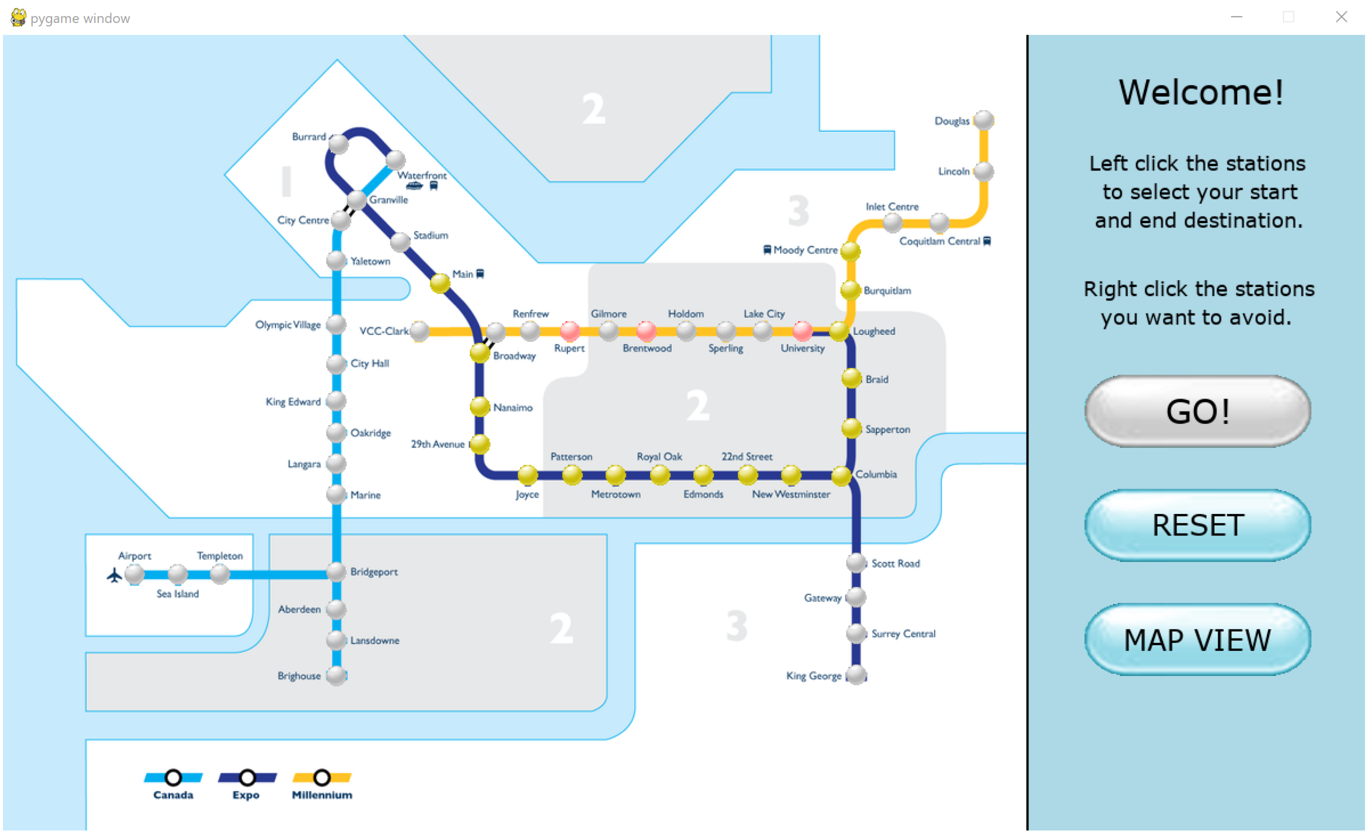
Welcome!

Left click the stations to select your start and end destination.

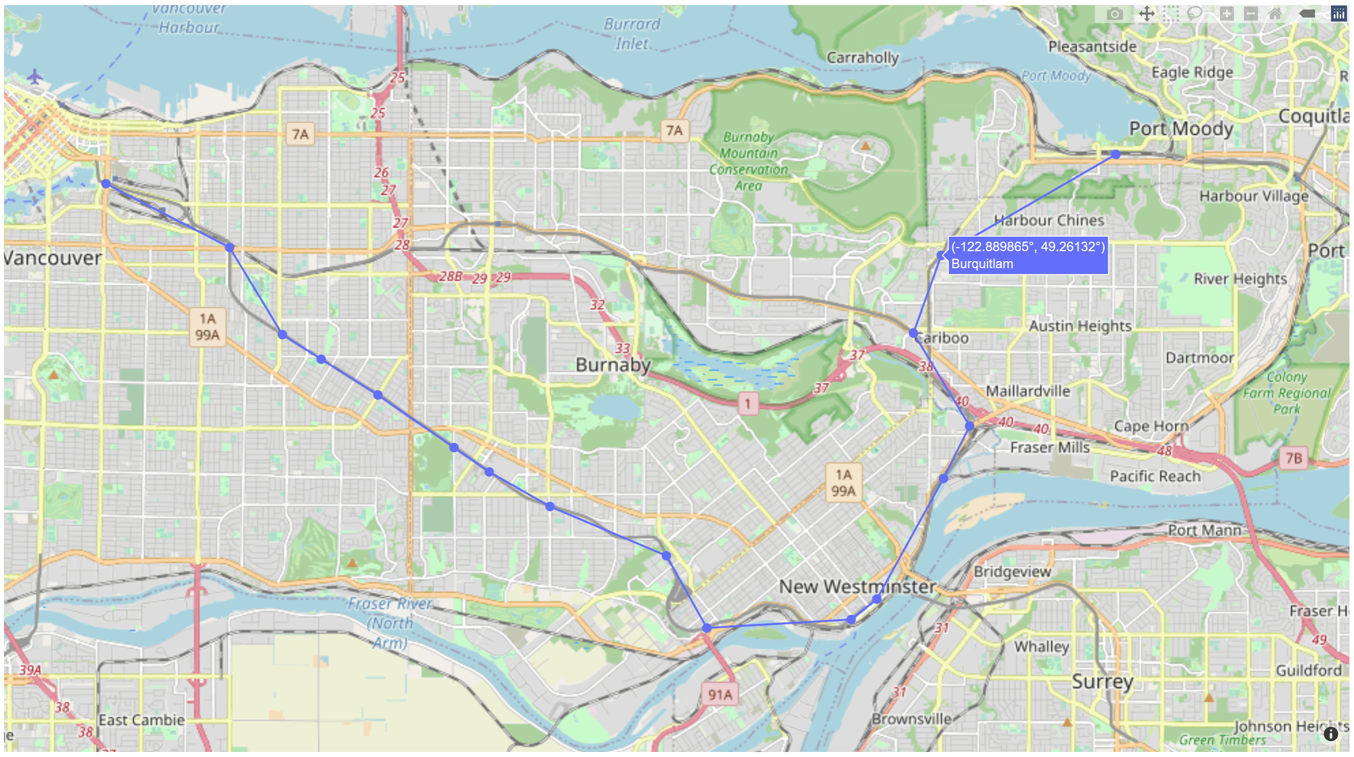
Right click the stations you want to avoid.

This map of the Vancouver SkyTrain subway system was from Wikimedia Commons (Wittal).

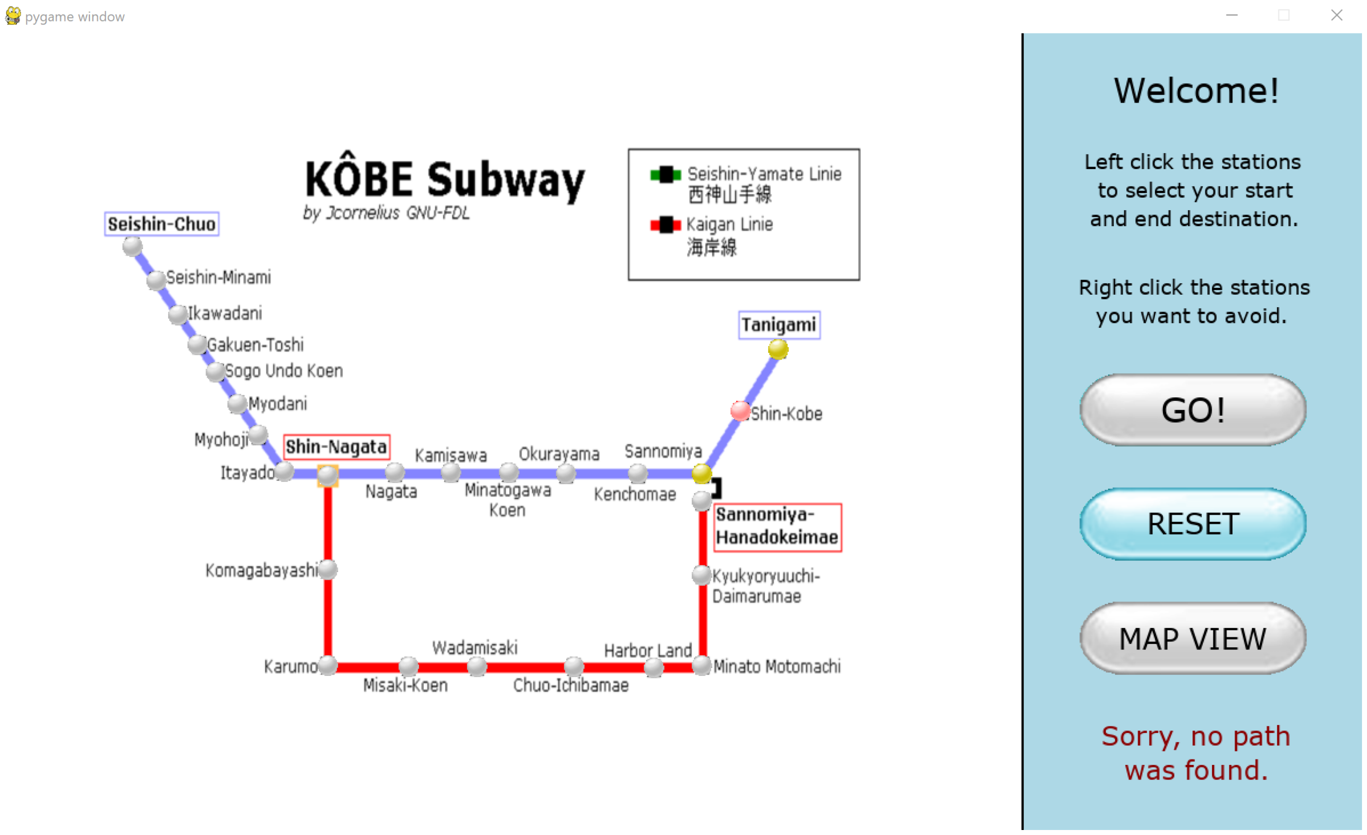
This is the pygame visualization of the Vancouver SkyTrain subway system. Left-click the two stations that they want to travel between. This will make the selected stations turn yellow. Right-click the stations you want to avoid. This will make the selected stations turn red. Then, press the “GO!” button which will display the shortest path between the two station, while avoiding the stations you do not want to visit. Note that selecting the same station twice **is** allowed here. When displaying the path, each station in the path will “light up” (turn yellow) one by one. If the user wants the path to load faster, click the screen anywhere rapidly. If no such path exists, the message “Sorry, no path was found” will be displayed in the sidebar. Here is what a possible path could look like (next page):


pygame window
Welcome!
Left click the stations to select your start and end destination.
Right click the stations you want to avoid.
GO!
RESET
MAP VIEW
Canada
Expo
Millennium

After the path is found, the user can click the "MAP VIEW" button, which will open a real plotly map on the users' browser. This will display the user's shortest path (note that if no path was found, this button cannot be pressed). The user can click the "RESET" button at any point in time to reset the subway system. If there is any confusion about which buttons are allowed to be pressed and which aren't, note that the buttons will turn blue when they are allowed to be pressed and are coloured grey otherwise. Here is what the plotly visualization could look like for the image of the path before:



In order to display how we can generalize our implementation of finding the shortest path, we applied our project to another subway system: the Kobe subway system in Japan. To see this visualization, uncomment the last two lines in the main file "main.py" (there is a comment in all caps in the file that gives the same instructions). Remember to comment out the lines related to the Vancouver SkyTrain subway system visualization before running the main file. The same instructions as for the Vancouver system apply here. The only difference is that this is another subway system. Here is what the second subway system should look like:



Left click the stations to select your start and end destination.

Right click the stations you want to avoid.

Sorry, no path was found.

This map of the Kobe subway system was from the Metro of Kobe (Metro of Kobe).

Discussion of Changes

Our original plan was to use the Hangzhou metro system as our sample subway system. However, we couldn't find datasets with useful information for this system and so we had to change the system we were going to use. Instead, we used the Vancouver SkyTrain subway system. We also added a second subway system, the Kobe subway system, with the sole purpose of showing how this can be generalized to any other subway system. So, the complexity of this second subway system was not a priority.

We also added the use of the plotly library as a way to visualize the path we've computed on a real map. Before, we were solely focused on having a pure pygame visualization. As a result, we added two columns to our dataset format that includes the latitude and longitude locations of the subway stations. This allows the user to see other places around the subway stations and is overall, a nice street view of the path they want to travel.

Unlike our plan in the proposal, we didn't use scrapy as it was unnecessary since the dataset was already available in a csv file when we found it on GitHub (Shkolnik). However, the neighbours and x- and y-coordinates for the stations in pygame had to be manually added to the dataset. For the Kobe subway system, we did not use a dataset but instead created our own. We manually found the latitude and longitude locations of the stations on Google Maps (Google).

In addition, originally, we only had a “GO!” button that the user can press when they have selected the two stations they wanted to go to and the stations they wanted to avoid. Now, we added a “RESET” button and a “MAP VIEW” button feature. The “RESET” button resets the map of the subway system to its original state. The “MAP VIEW” button causes a pop up window in the user’s browser that plots the shortest path (using the latitude and longitude locations of the stations in the path) on a real landscape map of the world.

On our proposal, we mentioned that the shortest distance between 2 stations will be measured in time that we obtain from a timetable. Instead, in our implementation, we used the least number of stations as an indicator of the shortest path. This is because no datasets were found with information about the time between two stations. Information found usually gave the time of an entire subway line rather than the individual times between each station.

As something extra, we added background music and sound effects, which we did not originally plan to include. The background music was composed by Benjamin Tissot and we chose this piece to give a more “relaxed” feel for the user (Tissot). Sound effects include a clicking sound when the mouse is pressed down and a different “clicking” sound when displaying the shortest path between two stations and a “completion” sound when the shortest path is complete (ZapSplat).

Discussion

We created this project to answer the research question of “How can we simplify the subway trip planning process of a subway user?”. A very important part of a user’s subway route is what gets them to their destination in the shortest amount of time. Allowing them to determine the shortest path easily reduces the amount of time they have to look at the map and go through each station that is in a potential path. While not having access to datasets with times between stations, we acknowledged this aspect by taking the shortest path according to the number of stations visited. Finding the shortest path required the use of the graph traversal approach taught in lecture (Liu). We recursively find all possible paths between two stations, and then choose the path with the least number of stations.

Another feature that we implemented was allowing the user to avoid certain stations. A user’s personal preferences may want them to avoid certain stations. Maybe they experience a certain level of discomfort passing through stations rumoured to be “bad areas” or they just do not want to travel on a certain subway line. All these features will simplify the day-to-day lives of a subway user, especially those living in areas with complex subway systems. Simplifying the planning process will encourage users to take advantage of the many benefits provided by subways systems (i.e., environmental benefits, enhanced safety, etc) and not switch to alternative less beneficial transportation systems (e.g., cars).

We implemented an interactive program with Pygame that allows the user to view the Vancouver SkyTrain subway system and plan out their subway route (as well as the Kobe subway system). Having the option to choose a start and end point and choosing any stations to avoid, the user can easily plan their subway journey so that it is the quickest possible journey. We also included a feature that allows the user to view their journey on a plotly map of the world. We think this is a nice feature because the user can see their subway route on a street map, in addition to a subway map. They can see the general areas and places around the stations.

Furthermore, our project can be generalized to any subway system. However, one limitation of our implementation is that a very specific dataset of the subway system is needed. Finding every station’s neighbours and also the coordinate points of each station in pygame is especially time-consuming. It would have been better to be able to find the shortest path by comparing the time of each route, but we were not able to do this because we couldn’t find a useful dataset with this information. Datasets found were restrictive, providing only the times of an overall subway line.

Another limitation is that it is not as accurate to count the number of stations to determine the shortest path. It would be better to have the time it takes to get to each station or the distance between each station. A third limitation is that our pygame visualization only works on a fixed screen size (1200px by 700px). It would be nice to put a boundary on the screen size but still be able to modify it slightly.

One way to expand on this project would be to extend it to not just subway systems, but public transit routes as well. For example, in Toronto, the bus system is a major part of public transportation, so including this application would be a good next step. We can incorporate the public transit routes and the subway routes together. In addition, we could include other factors (other than stations the user wants to avoid) to determine the ideal path for the user to take between the start and end destination. For example, the user may want to prioritize having the least number of transfers between public transit routes and subway lines and have the shortest path come second. We could also take into the account the public transit schedules and base the quickest path between the start and end destination on the current time of the user or the time they plan on starting their travels.

In summary, the main objective of our project was to simplify the user's subway experience to make complex subway systems easier to navigate. We achieved this by including multiple different features in our project such as finding the shortest path between stations and avoiding certain stations. We displayed these features in an interactive program that allowed the user to plan their subway route. There were some limitations in our project, but even with these limitations, we were able to successfully reach our goal. There are also many other next steps outlined in this section to further enhance the subway experience of a user.

References

CSC111 Department. “CSC111 Winter 2021 Assignment 1: Linked Lists.” *CSC111 Winter 2021 Assignment 1: Linked Lists*, CSC111 Department, 2021, `www.teach.cs.toronto.edu/~csc111h/winter/assignments/a1/handout/`.

Florida, Richard. “The Relationship Between Subways and Urban Growth.” *Bloomberg CityLab*, Bloomberg, 2016, `www.bloomberg.com/news/articles/2016-06-02/report-the-relationship-between-subways-and-urban-growth`.

Game Art 2D. “Free Casual Game Button Pack.” *Game Art 2D*, Game Art 2D, 2021, `www.gameart2d.com/free-casual-game-button-pack.html`.

Google. “Google Maps.” *Google Maps*, Google, 2021, `www.google.com/maps`.

Liu, David. “CSC111 Lecture 14: Representing Graphs in Python.” *CSC111 – CSC111 Lecture 14: Representing Graphs in Python*, David Liu, Department of Computer Science, 2021, `www.teach.cs.toronto.edu/~csc111h/winter/lectures/14-representing-graphs/david/14-slides.html#/title-slide`.

Metro of Kobe. “Kobe Metro.” *Subway: Kobe Metro Map, Japan*, Metro of Kobe, 2010, `https://mapa-metro.com/en/Japan/Kobe/Kobe-Subway-map.htm`.

Plotly. “Lines on Mapbox in Python.” *Plotly*, Plotly, 2021, `https://plotly.com/python/lines-on-mapbox/`.

Pygame. “Pygame Front Page.” *Pygame - Pygame Documentation*, Pygame, 2021, `www.pygame.org/docs/`.

Shkolnik, Dmitry. “dshkol / skytrainstations.csv.” *GitHub Gist*, GitHub, 2016, `https://gist.github.com/dshkol/d9132ea8d8e1b1d6c16c20f0485e27b9`.

Silverwitz, Phil. “Benefits of Subway Transportation.” *Azcentral: Part of the USA Today Network*, Azcentral, 2017, `https://getawaytips.azcentral.com/benefits-of-subway-transportation-12273529.html`.

Tissot, Benjamin. “The Elevator Bossa Nova: Muzak Royalty Free Music.” *Bensound*, Bensound, 2021, `www.bensound.com/royalty-free-music/track/the-elevator-bossa-nova`.

Wittal, Paul. “File:Vancouver Skytrain Map.png.” *Wikimedia Commons*, Wikimedia Commons, 2014, `https://commons.wikimedia.org/wiki/File:Vancouver_Skytrain_Map.png`.

ZapSplat. “Free sound effects & royalty free music.” *ZapSplat*, ZapSplat, 2021, `www.zapsplat.com/`.