Programming Project 3: Processing MTA Subway Entrance Data

1 Overview

In this programming project, you will work with a dataset consisting of all entrances and exits to the stations of the New York City Transit Authority. This dataset, like the others you have used in this course, is part of the *New York City OpenData* project, and is provided by the Metropolitan Transit Authority (MTA). It has been cleaned up by NYC's Department of Information Technology and Telecommunications (DoITT). The term "subway entrance (exit)" is used here to mean an entrance (exit) to any station, whether it is above or below the ground. The dataset has a row for every distinct subway entrance and exit[1]. Each row provides the entrance's spatial location, name, which is really its street address, and the train lines that are accessible from it. With this information, a rich set of queries is possible. In particular, you will design a program that determines which entrances are part of the same stations, which stations are part of the same train lines, and which stations and/or trains are closest to a given GPS location. As you will soon see, solving these problems will require using algorithms from Chapter 8 (on-line disjoint set operations), and hash tables, as well as other material you have learned about in this course.

2 Objectives

This project is designed with a few objectives in mind:

- to give you more experience working with real, large, open data sets.
- to give you experience programming the on-line disjoint set problem.
- to give you experience creating a very simple hash table.
- to give you a problem that has practical application and that can be extended to become a useful application.

3 About the Data Set

The data set is visualized on a map at this URL:

`https://data.cityofnewyork.us/Transportation/Subway-Entrances/drex-xx56/data`

You can download it in CSV format with this link:

`https://data.cityofnewyork.us/api/views/he7q-3hwy/rows.csv?accessType=DOWNLOAD`.

The dataset that is downloaded will have as its first row, the labels of its columns. For this assignment, you should delete that row, so that the program can assume all rows are actual data rows. A version of it without the first line (which contains column labels) is also posted on the server in the directory

`/data/biocs/b/student.accounts/cs335_sw/resources/project3_files/`

This particular dataset is relatively small - only 1928 lines, each of which is just five columns. Your project will read the CSV file and store its rows into an array (or vector). As it reads the rows and stores them, it will do some preliminary processing. After the data is stored in memory, the program will process various queries about the dataset. If you need a refresher about CSV files, please read assignment 1 or 2. Remember that fields can contain embedded commas. (Fields can contain commas if they are within quoted strings, e.g., "Brooklyn, New York" is a single field.)

There does not appear to be an on-line data dictionary for this dataset. The table below defines its five fields. In the table, the following terms are used:

decimal A fixed decimal numeric literal, with an optional leading negative sign, such as -40.32 or 7.1234567

[1]Some stations have exits that are not also entrances, and in this case the "name" field of the row indicates this.

line_identifier A string used by the MTA to define a transit line. These are either uppercase letters A through Z, or single digits 1 through 7, or two- or three-letter uppercase strings such as GS or SIR.

Field Number	Field Name	Type and Format
1	`ObjectID`	A unique positive integer identifying the particular subway entrance.
2	`URL`	A URL that provides service information about the subway entrance.
3	`Name`	A string that specifies a unique name for this subway entrance, which is always its street address or a similar locator. This field might contain a substring "(exit)" that indicates that it is an exit-only.
4	`The_Geom`	The GPS coordinates, in the format `POINT (decimal decimal)` where there is white space between the word "`POINT`" and the first parenthesis, and between the two decimals. ***Note: the first number is the longitude and the second is the latitude*** (the reverse of the tree dataset!)
5	`Line`	A string that consists of either a single `line_identifier`, or a sequence of `line_identifiers` separated by hyphens.

A sample row from this file, wrapped here because of the limited page width, looks like:

```
151,http://web.mta.info/nyct/service/,Parsons Blvd & Archer Ave at NE corner,
POINT (-73.799784000399 40.70235300071414),E-J-Z
```

3.1 Subway Entrances, Subway Stations, and Lines

Anyone who uses the New York City subway system knows that there is some ambiguity in the meaning of the term "subway station". For example, the 51st Street station of the 6 line is connected to the Lexington Avenue station of the E-M line. Is this one station or two? In this project, the answer is one. The term subway station refers to the transitive closure of the set of all stations that are connected to each other by pedestrian passages without having to pay an additional fare, and without having to exit the subway system, even if there is a free entrance by doing so. Thus, the 59th Street 6 line is not part of the Lexington Avenue F line, because one has to leave the station to get to the other.

3.1.1 Connectivity

Given any pair of subway entrances, either they are part of the same subway station or they are not. When two entrances are part of the same station, we say they are ***connected***. Otherwise they are ***disjoint***. A moment's thought should convince you that connectivity is *symmetric*, *reflexive*, and *transitive*, and hence, an equivalence relation, and that this relation partitions the collection of all subway entrances into a set of non-empty, disjoint sets such that every entrance belongs to a unique set, which we call a ***subway station***.

Every subway station has a unique name, which we call its ***station_name***. In this project, the `station_name` of a given station is any one of the names of the subway entrances that belong to that station. By the nature of how the stations will be identified, their names may change as the file is being read, but once the entire file is read, every station will have a unique, stable, station_name. This will become clear after you understand how stations are created.

A ***line*** consists of the set of all stations that service that line. One station may be part of many lines, and hence lines do not partition the collection of stations into disjoint sets. Although most lines define a sequence of stations, determined by the order in which a train running on that line visits each station, some do not because they branch. Therefore, in this project, we do not attempt to order the stations for a given line. Each line is identified by its `line_identifier`, defined above.

Determining Connectivity. The problem in determining connectivity, meaning which entrances are part of the same station, is that there is no explicit column in the dataset that tells us this. Therefore, we must use a heuristic rule to determine this. The rule we shall use is the following:

Definition. Two subway entrances E_1 and E_2 are *connected* if either

- the set of `line_identifiers` for each is identical, and
- the distance between them is at most 0.28 kilometers.

or there is a third entrance E_0 such that E_1 is connected to E_0 and E_2 is connected to E_0.

The distance constraint may be subject to change. Your program should make this distance an easily changeable parameter. With this definition, the dataset can be used to construct the collection of distinct subway stations. In addition, we can assign a unique location to each station by making it the centroid of the locations of its entrances. The centroid is the arithmetic mean of the locations of each of its entrances. We can approximate that mean using the average of the latitude and longitude values. Namely, if a station has entrances whose GPS points are $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)$ then the centroid is

$$c = (1/n)(\sum_{k=1}^{n} x_k, \sum_{k=1}^{n} y_k) \tag{1}$$

4 Program Invocation, Usage, and Behavior

The program is invoked from the command line and expects two command line arguments, which specify respectively (1) the ***csv*** file to be opened for reading and (2) the sequence of user commands to be processed. If two files are not specified, it is a usage error and the program must write an appropriate and meaningful error message onto the ***standard error stream***, after which it must exit. If a file that is specified does not exist or cannot be opened for some reason, the program must write an appropriate and meaningful error message onto the standard error stream and then exit.

Assuming that both files are opened successfully, the program must read the entire csv file, line by line, and process these lines according to the rules specified below. Once the entire csv file has been read and processed, the program will read the commands from the command file and process them one after the other. These commands are described and explained in Section 4.3 below.

4.1 Case Sensitivity and White Space Sensitivity

The program should treat all names of stations and station entrances case insensitively. This means that the following strings all refer to the same name:

```
Nassau St & Frankfort St at SE corner
nassau St & frankfort St at SE corner
Nassau st & frankfort St at se corner
```

It should treat any sequence of space characters as a single space character. Thus, the following strings refer to the same name:

```
Nassau St & Frankfort St at SE corner
Nassau St   &   Frankfort   St at SE corner
```

This is true of names entered as arguments to commands as well. This makes it easier for the user of the program, but harder for the programmer.

4.2 Processing the Input Data File

It is the task of the main program to read the input file, parse its lines, construct a `Subway_Entrance` object for each row, and make the calls to a `Subway_System` class to insert that object into the collection of entrances. The `Subway_Entrance` object should represent a single subway entrance. The `Subway_System` class must manage the collection of entrances so that, as each new entrance is found, it is checked against all other entrances to determine if it is connected to any of them, and if so, to update its sets. It is therefore a class that contains a container storing `Subway_Entrance` objects and, as you will see, a container storing `Line` objects.

4.3 Command Processing

After the CSV file has been processed, the program enters a command processing loop, in which it reads commands from the second file specified on the command line. The syntax of the valid commands from that file is as follows. The `bold` text is the command literal and the *`italicized`* text is its parameter list. ***All output lists should use the names as they are found in the original file, not names whose case has been changed.***

Command	Description
`list_line_stations` *`line_identifier`*	Lists the *station_names* of all subway stations that service the given line.
`list_all_stations`	Lists the *station_names* of all subway stations in the subway system.
`list_entrances` *`station_name`*	Lists the *names* of all subway entrances for the given station. The list should not include exit-only entrances.
`nearest_station` *`longitude latitude`*	Lists the *station_names* of all subway stations that are closest to the point (*`longitude,latitude`*). There may be more than one because two or more might be the same distance from the point.
`nearest_lines` *`longitude latitude`*	Lists the *line_identifiers* of all subway lines that are closest to the point (*`longitude,latitude`*). There may be more than one because two or more lines might be at a station that is nearest to the point, or because two stations might be at the same distance from the point.
`nearest_entrance` *`longitude latitude`*	Lists the *names* of all subway entrances that are closest to the point (*`longitude,latitude`*). There may be more than one because two or more might be the same distance from the point.

4.3.1 Distance Between Two Points on Sphere (The Haversine Formula)

The *Haversine* formula (see https: //en.wikipedia.org/wiki/Haversine_formula) can be used to compute the approximate distance between two points when they are each defined by their latitude and longitude in degrees. The distance is approximate because (1) the earth is not really a sphere, and (2) numerical round-off errors occur. Nonetheless, for points that are no more than ten kilometers apart, the formula is accurate enough. Given the following notation

d : the distance between the two points (along a great circle of the sphere,

r : the radius of the sphere,

φ_1, φ_2: latitude of point 1 and latitude of point 2, in radians

λ_1, λ_2: longitude of point 1 and longitude of point 2, in radians

the formula is

$$2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2-\varphi_1}{2}\right)+\cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2-\lambda_1}{2}\right)}\right) \tag{2}$$

A C++ function to compute this formula in a numerically efficient way is given in Listing 1. Remember that latitude and longitude are not interchangeable, and that the values of these are given in the commands and in the file with longitude followed by latitude.

Listing 1: Haversine Function (corrected version)

```
#include <cmath>
# To build, link to the math library using -lm
```

```
const double R = 6372.8              // radius of earth in km
const double TO_RAD= M_PI / 180.0; // conversion of degrees to rads

double haversine( double lat1, double lon1, double lat2, double lon2)
{
    lat1        = TO_RAD * lat1;
    lat2        = TO_RAD * lat2;
    lon1        = TO_RAD * lon1;
    lon2        = TO_RAD * lon2;
    double dLat = (lat2 - lat1)/2;
    double dLon = (lon2 - lon1)/2;
    double a    = sin(dLat);
    double b    = sin(dLon);

    return 2 * R * asin(sqrt(a*a + cos(lat1)*cos(lat2)*b*b));
}
```

5 Program Organization and Logic

The program must process the commands efficiently. This section discusses some of the logical issues, technical tools, and algorithms that this suggests you use.

5.1 Bit Masks

Bit masks are an important tool in solving several of the problems in this assignment. For example, given two subway entrances, each with its own set of lines that it services, how can you quickly decide if the two sets are identical? Of course you could design a solution that iterates over train lines, and which would be very inefficient, but you can also design a solution that uses constant time to solve this problem, if each set is represented by a bit mask. How?

Similarly, given a `line_identifier`, how can you find the set of all stations that service the line, assuming that this set is not stored for each line? For each station, you would need to check if the `line_identifier` is part of the set of lines that it services. This can also be done in constant time if the station has a bit mask and the line has a bit mask. How?

Given that there are at most 26 single letter lines, and 7 single digit lines, and just a few lines with more than one character, a 64-bit integer has more than enough bits to define bit masks to represent sets of lines.

5.2 Determining Connectivity (Subway Station Creation)

Determining the disjoint sets of subway stations and which subway entrances are part of the same station requires using the find and smart union operations from Chapter 8, which implies that `Subway_Entrance` objects should be stored in an array or a vector. In the lecture notes for Chapter 8, simple integers are used as an example to illustrate the algorithms, but your project must define an array whose elements are class objects and that also have a member that is the parent index. Thus you need to generalize those algorithms.

Each time a new `Subway_Entrance` object is created, it is placed into a set of size one. The object in this set is then compared against all currently existing sets to see if they should be connected to this one. If so, this is a union operation. If it is not connected to any existing sets, it remains a set of size one. Eventually, all subway entrances are compared to all others in this way, when the entire file has been read, and the sets have been formed. At this point the subway stations can be given stable subway station names, which are simply the names of the subway entrances at the roots of the parent trees that represent them. In addition, their locations as GPS coordinates can be computed based on Equation 1.

5.3 Finding Stations Efficiently

One of the problems the program must solve is how to find a particular station efficiently. If the only representation of a subway station is the parent tree for that station stored in an array or vector of such trees, then when given a station name to find, as when a command is given to list all entrances for that station, the program would have to do an iterative search, taking too much time. Instead, it can create a hash table whose keys are station names, and whose values are the index values in the array or vector at which the parent tree root is found. (It can also store the subway station location there too, if it is a structure or class object.) Therefore, once all stations have been created, a hash table can be created to store `Subway_Station` objects for fast access later. (What data and operations should a `Subway_Station` object encapsulate?) This hash table should be represented by a class named `Subway_Station_Hash`.

5.4 Lines

A subway line is a set of stations. Should the program construct these sets as it reads the input file, or only when a command is issued to list the stations belonging to the line? How would you represent these sets? Since subway stations are not stabilized until after all data has been read, it is a challenging task to construct the sets as the input file is processed. Instead, it is better to do so in a lazy way, only when a command is invoked to list the line's stations. Then how can this be done efficiently? As was mentioned in Section 5.1 above, if a line has a bit mask and the hash table has a method to enumerate all `Subway_Station` objects in it, each of which has a bit mask, then the set of all stations serving that line can be determined with a single pass across the set of subway stations, and these stations can then be stored, or at least some type of reference to them can be stored, in a linked list or vector associated with a `Line` object. Thus a Line object stores a bit mask and some type of reference to a container of `Subway_Station` objects. How can you create this bit mask efficiently?

How can you represent the set of all `Line` objects in such a way that there is fast access to them? Since you never delete lines and only insert them and search for them, this suggests that they should be stored in a hash table. Your program should create a hash table whose keys are line_identifiers and whose values are `Line` objects. It is possible to design a very simple hash function for this table. In fact, it is possible to define a perfect hash function that is extremely fast, given the simple form of `line_identifiers`. Your program will be evaluated in part on your accomplishing this task. This hash table should be a class in itself, named `Subway_Line_Hash`.

5.5 The Subway System

The subway system is represented by a `Subway_System` object. This object must encapsulate the two hash tables, `Subway_Station_Hash` and `Subway_Line_Hash`, and the vector that stores the `Subway_Station` parent trees. These objects need not be exposed to the clients. Instead, the `Subway_System` should provide methods to perform all operations required by the program, so that the main program's only interaction is with the `Subway_System`. You are free to design this class interface as you see fit.

5.6 Required Files

Regardless of how you define the class interfaces for your classes, each class must be represented by a separate pair of files, its header file and its implementation file. Minimally, this implies that your program would need to contain the following files, assuming they are named in the obvious way:

```
main.cpp
subway_entrance.h
subway_entrance.cpp
subway_station.h
subway_station.cpp
subway_system.h
subway_system.cpp
subway_station_hash.h
subway_station_hash.cpp
subway_line_hash.h
subway_line_hash.cpp
README
Makefile
```

The README file must contain a running log of your progress and changes and thoughts and possibly frustrations during this project, or your "eureka" moments. It can also contain documentation of the program. There is no hard rule about it. I will provide a Makefile that can be used regardless of how you name the files.

6 Testing Your Program

You should make sure that you test the program on a much smaller data set for which you can determine the correct output manually. You should create your own small test files for that purpose. (Feel free to share those with other students on Piazza.)

You should make sure that your program's results are consistent with what is described in this specification by running the program on carefully designed test inputs and examining the outputs produced to make sure they are correct.

7 Programming Rules

Your program must conform to the programming rules described in the ***Programming Rules*** document on the course website. It is to be your work and your work alone.

8 Grading Rubric

The program will be graded based on the following rubric, based on 100 points.

- A program that cannot run because it fails to compile or link on a `cslab` host receives only 20%. This 20% will be assessed using the rest of the rubric below.
- Meeting the functional requirements of the assignment: 50%
- Performance and Design. These are inseparable in this assignment. It includes efficient solutions to the problems as well as choices of algorithms, data structures, modularity, organization: 25%
- Documentation: 20%
- Style and proper naming: 5%

This implies that a program that does not compile on a `cslab` host cannot receive more than 20 points.

9 Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on May 14, 2018. Create a directory named named *username*_project3 where *username* is to be replaced by your CS Department network login name. Put all project-related source-code files and the README and Makefile into that directory. ***Do not place any executable files, data files, or object files into this directory.*** You will lose 1% for each file that does not belong there, and you will lose 2% if you do not name the directory correctly[2].

Next, create a zip archive for this directory by running the zip command

```
zip -r username_project3.zip ./username_project3
```

This will compress all of your files into the file named `username_project3.zip`. Do not use the tar compress utility. If you do not zip the directory correctly so that all files, when extracted with the command `unzip username_project3.zip`, are not in a properly named directory, your program will lose 2%.

The submit command that you will use is `submit_cs335_project`. It requires two arguments: the number of the project and the pathname of your file. Thus, if your file is named `username_project3.zip` and it is in your current working directory you would type

```
submit_cs335_project 3  username_project3.zip
```

[2]I have scripts that process your submissions automatically and misnamed files force me to manually override them.

The program will copy your file into the `project3` subdirectory

```
/data/biocs/b/student.accounts/cs335_sw/projects/project3/
```

and if it is successful, it will display the message, "`File ... successfully submitted.`"

You will not be able to read this file, nor will anyone else except for me. But you can double-check that the command succeeded by typing the command

```
ls -l /data/biocs/b/student.accounts/cs335_sw/projects/project3
```

and making sure you see a non-empty file there.

If you put a solution there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.