Programming Project 2: Processing MTA Subway Data: Revised

1 Overview

This is a minor revision of the original assignment. There is one more command, some small changes to the `SubwaySystem` *class, and more suggestions and advice on how to develop this project. Changed sections are marked with* **[Changed]**.

The dataset for this project consists of (almost) all entrances and exits to the stations of the New York City Transit Authority. This dataset is not part of the ***New York City OpenData*** project, but is instead maintained and provided by New York State's ***Open NY Initiative***. The terms "subway entrance" and "subway exit" respectively mean an entrance or exit to any station, whether it is above or below the ground. I will henceforth use the general term ***portal*** to mean either an exit or an entrance. The dataset has a row for every distinct portal. Perhaps counter-intuitively, if an elevator, an escalator, and a staircase are adjacent to each other, they will each have their own distinct row and be treated as separate portals.

Your program will begin by reading and storing the rows of data, which is contained in a csv file whose name is the first command-line argument to the program. It will then process a sequence of queries about that data. The queries will be read from a file whose pathname is given as the second command-line argument to the program.

This particular dataset is relatively small - only 1839 rows. Your project will read the csv file and store each row into an element of an array (or vector). It will not store all of the columns of the rows; details about this appear below. 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. 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 is so much information in this data set that many interesting queries are possible. Unfortunately, there is not enough time to create a very robust and useful program to ask the most interesting types of questions, but in this assignment you will get a taste of what can be done with this type of data. In particular, you will design a program that determines which entrances are part of the same stations, which stations are transfer points, and which stations and/or trains are closest to a given GPS location. As you will soon see, solving these problems will require using hash tables, and disjoint set representations.

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 The Data Set

The data set in its original form is found at NYC Transit Subway Entrance And Exit Data. I have cleaned this data set, removed the header line, and added another column to it. You should not attempt to work with the original data, but you may want to inspect it on-line. This section describes the data set after my revisions to it.

Each row has 33 separate columns, not all of which have data in them. The information that is included includes items like the entrance's spatial location in GPS coordinates, the subway lines that are accessible from that entrance, the *division* (e.g. BMT, IRT, IND), the *line* (e.g. 4th Avenue), the *station name*, and the cross streets of the entrance (exit) and which compass direction it is at that intersection (e.g. Lexington Avenue and 63rd Street NW corner). It also has the access type (stairs, elevator, etc), whether there are vending machines, and so on.

Type	**Name**	**Description**
`string`	division	`IND, IRT, BMT`
`string`	line	Name for route, such as `"Sixth Avenue"`, or `"Lexington Ave"`
`string`	station_name	Official name of station on that line
`double`	station_latitude	Latitude and longitude of station that the
`double`	station_longitude	portal accesses
`string`	route1	Each of the route fields has a single train designator such
`string`	route2	as A,B,C,... or 1,2,3,.., including the two-letter
`string`	route3	designators, FS and GS.
`string`	route4	The routes are the trains that are accessed at that portal.
`string`	route5	
`string`	route6	
`string`	route7	
`string`	route8	
`string`	route9	
`string`	route10	
`string`	route11	
`string`	entrance_type	Stair, elevator, escalator, etc
`string`	entry	`YES` or `NO`
`string`	exit_only	`YES` or empty string
`string`	vending	`YES` or `NO`
`string`	staffing	`FULL` or `NONE` or `PART` or `"Spc Ev"`
`string`	staff_hours	string describing hours and days at which it is open
`string`	ada	`TRUE` or `FALSE`
`string`	ada_notes	string providing more details about ADA compliance
`string`	free_crossover	`TRUE` or `FALSE` (whether one can switch directions on lines)
`string`	north_south_street	Name of north-south street
`string`	east_west_street	Name of east-west street
`string`	corner	which corner: `NW, SW, NE, SE`
`int`	id	1 or 2, in case the rest of the location information is duplicated
`double`	entrance_latitude	Latitude and longitude of the portal
`double`	entrance_longitude	
`GPS`	station_location	The station's location in the form (lat, long)
`GPS`	entrance_location	The portal's location in the form (lat, long)

The data dictionary lists the fields in the order in which they occur in the csv file.

4 Subway Portals, Subway Stations, and Routes

4.1 Stations and Their Connections

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 the data set, they are separate stations.

There is a natural equivalence relation on subway stations:

- Two stations are equivalent if there is a free transfer between them without having to exit and re-enter (as is the case for the F train at 63rd Street and the 4-5-6 at 59th Street.) This is reflexive, symmetric (to the best of my knowledge), and transitive.

One objective of your program is to form the equivalence classes of the subway stations in the data set. The set of all stations in an equivalence class based on the preceding relation will be called a ***disjoint station set*** or a ***station set*** for short when the meaning is clear. Your program will use a heuristic to decide whether two stations are equivalent: it will call them equivalent if they have the exact same set of routes and they are "sufficiently close" to each other geographically.

There is no explicit column in the dataset that tells us whether two stations are connected. Therefore, we must use a heuristic rule to determine this. The rule we shall use is the following:

Definition 1. Two subway stations S_1 and S_2 are *connected* if either

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

or there is a third station S_0 such that S_1 is connected to S_0 and S_2 is connected to S_0.

The distance constraint is based on the lengths of trains and the historical minimum distance between non-equivalent stations. With this definition, the dataset can be used to construct the collection of ***disjoint station sets***.

4.2 Portals and Connectivity

Given any pair of subway portals, either they are part of the same subway station or they are not. When two portals 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.

The data set provides the name of the station to which a portal belongs. It is the third field of the row. If two portals have the same third field, then they are part of the same station and thus equivalent. But the equivalence class is larger than this, because if two stations are equivalent by Definition 1 above, then all portals for each of the equivalent stations are equivalent. For example, if entrances A and B are part of station X, and entrances C and D are part of station Y, and we discover that X and Y are equivalent by Definition 1, then A, B, C, and D are all equivalent - they all give access to the exact same set of routes in the small geographic vicinity of the two stations.

This shows that the set of all portals is a collection of disjoint sets, and this collection will be represented by a set of parent trees in your program.

4.3 Portal Names

Entrances and exits are not given unique names in the data set, but we can define a unique name as follows. The concatenation of the north-south-street field, the east-west-street field, the corner field, and the ID field, with all spaces squeezed so that any white-space is at most one space character, using a comma to separate the fields, forms a string that uniquely identifies the portal.

Example

[Changed] The row containing the fields

```
north-south street  east-west street   corner  id
Madison Ave         42nd St            NW      2
```

will be given the portal name "`Madison Ave,42nd St,NW,2`".

Being able to give a unique name to each portal means that we can use that name as a key that represents it uniquely.

4.4 Routes

A ***route*** consists of the set of all stations that service that route. One station may be part of many routes, and hence routes do not partition the collection of stations into disjoint sets. Although most routes define a sequence of stations, determined by the order in which a train running on that route visits each station, some do not because they branch. Therefore, in this project, we do not attempt to order the stations for a given route, but we will store the stations that are part of a route in a container that represents that route. A `SubwayRoute` encapsulates the set of the stations that are part of that route, in no particular order, as well as the operations that a route should support. I will provide the `SubwayRoute` class for you.

Note that the data set uses the term "line" as a portion of a route. For example, the "`R`" is a route that starts in Brooklyn on the *4th Avenue* line, then runs on the *Broadway* line in Manhattan, then runs on the *Queens Boulevard* line in Queens. ***Do not confuse lines with routes.***

4.5 Route Sets

A ***route set*** is a set of zero or more routes. At any given subway station, there is a set of routes that it accesses. Therefore, there is a route set associated with each station and hence each portal. This data set (which is not as current as it should be) only has 25 distinct routes. Therefore a set of routes can be represented easily by a 32-bit number, with each unique bit position representing a unique route. We will use a 64-bit number to represent a route set. This will be described below in more detail.

5 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 5.3 below.

5.1 Case Sensitivity and White Space Sensitivity

The program can treat all names of stations and portals ***case sensitively***. However, 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.

5.2 Processing the Input Data File

The main program must read the input file, parse its lines, construct a `SubwayPortal` object for each row, and make the calls to a `SubwaySystem` object to insert that object into the collection of parent trees of subway portals. Each `SubwayPortal` object represents a single subway portal, i.e., a row of the data set. Exactly what type of container the `SubwaySystem` needs for them will be described below. In addition, as the portal object is read, the route set for that portal should be computed and stored, and the station that it is part of should be added to the `SubwayRoute` objects of all routes that it services.

After all of the rows have been read and stored in the appropriate container, the main program should request the `SubwaySystem` object to form the equivalence classes of portals and subways stations (using Definition 1 and the union algorithm.)

5.3 Processing the Command File

[Changed] After the csv file has been read, 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_route_stations` *`route_identifier`*	Lists the *station_names* of all subway stations that service the given route. The route identifier is case insensitive - either upper or lower case should identify a route, e.g. "`A`" and "`a`" specify the `A` route.
`list_all_stations`	Lists the *station_names* of all subway stations in the entire subway system.
`list_routes` *`portal_name`*	Lists the routes that can be accessed at this portal. Routes should be printed in the form "`route`,`route`,..., `route`", where route is a route id like `A`, `B`, `7`, `FS`. portal_name is the unique string for the portal defined in Section 4.3.
`list_station_portals` *`station_name`*	Lists the *names* of all subway portals for the given station. The station name is ***case sensitive***. **(NEW)**
`nearest_portal` *`latitude longitude`*	Lists the *portal name* of the portal that is closest to the point (*`latitude`*, *`longitude`*). The two numbers should be checked for validity - no absolute value greater than 180 degrees is allowed. In the very unlikely event that two portals are exactly the same distance from the point, either one can be listed.
`nearest_routes` *`latitude longitude`*	Lists the *route_identifiers* of all subway routes that are closest to the point (*`latitude`*, *`longitude`*). There may be more than one because two or more routes might be at a station that is nearest to the point.

5.3.1 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

$$2r \cdot \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \quad (1)$$

You have already seen the code for this, so it is omitted this time.

6 Program Architecture and Algorithms

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

6.1 Subway Portals, Subway Stations, and Parent Trees

[Changed] The program will use ***parent trees*** to represent the collection of disjoint sets of stations. To do this it needs to create an array of objects that will be used by the union/find algorithms. The objects in this array cannot be simple integers. Instead the objects stored in this array are `SubwayStation` objects. The private data in each `SubwayStation` object will look like this:

```
private:
    int          parent_id;
    set<string>  station_names;  // Note change here
    list<int>    children;
    string       portal_name;
    SubwayPortal portal;
```

Notes

- Each cell has an integer `parent_id` that is the index of the cell's parent in the array, or is a negative number indicating that it is a root.
- The cell also stores a set of the names of the stations, `station_names`, that the cell represents. A cell can have more than one station name because when stations are identified by the program as being connected, their disjoint sets are unioned, and at that point, they become a single set with more than one name. By doing this, the array cells that are roots of their trees contain the names of all stations that are part of that set of connected stations. In the earlier version of this assignment, it was a list of names, **but now I suggest a set of names, because you need to avoid duplicates. Your choice.**
- The cell has a list of index values of the children of that cell, `children`, to make it easy to find the portals and other stations that are part of that disjoint set. This list is useful once all sets have been formed, and then only for the cells that are the roots of the trees. Initially it should have an empty list. Remember that the find algorithm can reorganize the tree and the list of children can become stale and invalid. **I suggest that you create the lists after all sets are finalized. It is easy if you use the `find` algorithm to do this.**
- It has the unique name of the portal that was constructed and stored into that cell, `portal_name`, and
- it has the data from the portal itself, in `portal`.
- As each row of the data set is read, a `SubwayPortal` object is created for that row and stored into the `portal` member of the next free cell in the array. Initially it has a `parent_id` of -1 and the `station_name` is copied from the row's data into the list of station names. The list of children is empty. The portal's unique name is constructed as described in Section 4.3and written into the `portal_name` member.

The `SubwayPortal` class must provide methods that process its data, which must be private. It does not have to store all of the data from the row, but it must store enough of it to process the commands that come from the command file.

A `SubwayStation` ***must implement*** at least the following interface. You may decide to include other public and/or private methods.

```
class SubwayStation
{
public:
    /** SubwayStation() is a default constructor
     *  It should initialize any private members with suiatbel default values.
     */
```

```
        SubwayStation();

        /** This is a constructor that creates a SubwayStation object from a portal
         *  It makes the portal the embedded portal.
         */
        SubwayStation(SubwayPortal portal );

        /** set_parent() sets the parent id of the station
         *  @param int [in] the id of the parent
         */
        void  set_parent(int newparent);

        /** add_child() adds a new child to the station's list of children
         *  @param int [in] the index of the child to add
         */
        void  add_child(int child);

        /** A friend function that determines when two stations are connected
         *  @param SubwayStation [in] s1
         *  @param SubwayStation [in] s2
         *  @return bool true iff s1 and s2 are connected according to rules defined
         *               in the assignment specification
         */
        friend bool connected(SubwayStation s1, SubwayStation s2);
        /** add_station_name() adds a new name to station
         *  @Note:  It does not add a name if it is already in the set of names for
         *          the station.
         *  @param  string [in] newname is name to be added
         *  @return 1 if name is added and 0 if not
         */
        int add_station_name(string newname);

        /** names() returns a list of the names of the station as a list of strings
         */
        list<string> names()  const;

        // primary_name() is the first station name in its set of names
        string  primary_name() const;

        // parent_id() is the index in the array of the parent of the station
        int    parent_id() const;

        /** portal_list() returns a list of the ids in the list of the portals in
         * this station set
         */
        list<int>   portal_list() const;

        // returns the name of the embedded portal
        string  portal_name() const;

        // returns the portal that is embedded in this station object
        void get_portal(SubwayPortal & ) const;
    private:
        int          m_parent_id;
        set<string>  m_station_names;
        list<int>    children;
        string       portal_unique_name;
        SubwayPortal portal;
    };
```

After all portals have been stored into the array, the program must run through the array and determine the connectivity of all portals and stations. It will repeatedly apply the definition of connectivity until all stations that should be connected to each other have been connected. When it is finished, the only array entries with negative parent_ids will be those that are the roots of disjoint sets.

6.2 Finding Subway Portals Efficiently

One of the problems the program must solve is how to find a particular subway portal efficiently. Commands supply portal names or station names, which are strings. The program has to be able to find the cells in the array that contain these particular names. Linear searching through this array is unacceptable.

Instead, the program will use a hash table whose keys are portal names, and whose values are the index values in the array or vector at which the portal was stored. As each portal is read and stored into the parent tree array, a pair (portal name, array index) can be inserted into the hash table. When the commands are processed, the hash table can be accessed to find that portal in O(1) time in the array.

6.3 Finding Stations Efficiently

A second hash table can be used for finding station names efficiently. This hash table will store station names and the positions in the array at which these are found. After the connectivity of all stations has been computed, the array entries with negative parent_ids are those of unique stations. No two such entries will have the same station name in their station name lists. For each array entry whose parent_id is negative, and for each name in the station name list in that entry, the pair (station name, array index) can be inserted into this hash table. When a command refers to a station name, it can be looked up in this hash table and its array entry found.

6.4 Routes, Bit Masks, and Bit Strings

Bit masks and bit-strings are important tools in computing. In this problem they can simplify the solution to ***route set*** problems.

Given two subway portals, each with its own set of routes that it accesses, how can you quickly decide if the two sets are identical? Of course you could design a solution that iterates over train routes, 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 string, using only bit-wise and/or integer operations

Similarly, given a route identifier, how can you find the set of all stations that service the route, assuming that this set is not stored for each route? For each station, you would need to check if the route identifier is part of the set of routes that it services. This can also be done in constant time if the station has a bit string and the route has a bit mask, also using nothing but bit-wise operations.

Given that there are at most 26 single letter routes, and 7 single digit routes, and two routes with more than one character ("FS" and "GS"), a 64-bit integer has more than enough bits to define bit strings to represent sets of routes. Namely each bit is associated with a specific route. If a bit is 1, the route is in the set, and if 0, it is not. Thus a 64-bit integer can represent a set of routes. Each route is then given a unique route mask, which is a 64 bit integer that has a 1 bit in the position assigned to that route, and 0 bits everywhere else.

Example

Suppose that bits `1,2,3, ...7` are assigned to represent the `1,2,3,..,7` routes respectively and that `route_set` is an integer storing the route set for some portal. If we want to know whether the 4 route stops at that station, we use the bit mask with a 1 in bit 4 and 0 everywhere else. Call this integer `route_mask`. Then `route_mask & route_set == 1` if and only if the 4 is at that portal.

Deciding whether two route sets are identical is nothing more than testing two integers for equality with this strategy.

When a portal row is read, the set of routes in it should be converted to a bit-string. That bit-string is what you should store in your `SubwayPortal` object. This way a simple integer represents the trains that can be accessed there, and it will be easy to check whether a train is at that station using a bit-mask for the train.

The routes are in separate fields of the row.

6.5 Routes

[Changed] I have written a `SubwayRoute` class that encapsulates the data and methods for subway routes. The class interface and bomary object file are on the server.

6.6 The Subway System Class

[Changed] The main program will rely on a class called the `SubwaySystem` class to carry out all processing related to subway portals, stations, and routes. This class must encapsulate the array of parent trees, the hash tables, and an array of route masks. All of this data must be hidden inside this class. These objects need not be exposed to the clients. Instead, the `SubwaySystem` should provide specific methods to perform all operations required by the program, so that the main program's only interaction is with the `SubwaySystem`. The methods that the main program will call are in the interface below. You are free to add more private methods, but you cannot modify the public interface of this class. I will provide this code on the server.

```
#define MAX_STATIONS  2048

using namespace std;


class _SubwaySystem
{
public:

    /** add_portal()  adds the given portal to the array of portals
     *  It also creates a hash table entry for this portal that points to
     *  its location in the array.
     *  @param  SubwayPortal [in] portal: an initialized portal
     *  @return int  1 if successful, 0 if portal is not added.
     */
    int add_portal(SubwayPortal portal);

    /** list_all_stations() lists all subway station names on the given stream
     *  @param [inout] ostream out is an open output stream
     */
    void list_all_stations(ostream & out) const;

    /** list_all_portals() lists all portals to a given station on given stream
     *  @param [inout] ostream is an open output stream
     *  @param [in]  string station_name is the exact name of a station,
     *          which must be the name of the set of portal names. These can
     *          be obtained from the output of list_all_stations().
     */
    void list_all_portals(ostream & out, string station_name) const;

    /** list_stations_of_route() lists all station names on the given route on
     *          the given output stream
     *  @param [inout] ostream is an open output stream
     *  @param [in]  route_id route is the name of the subway route whose
     *          stations are to be printed onto the stream
     */
    void list_stations_of_route( ostream & out, route_id route) const;


    /** form_stations()
     *  Note: form_stations should be called once after the array of portals
     *  has been created. It determines which portals are connected to each
     *  other and forms disjoint sets of connected stations and portals.
     *  After all sets are formed, it stores the names of the stations that
     *  name these sets (e.g., if parent trees, the ones at the root)
     *  in a hash table for station names for fast access.
     *  Finally, it sets an internal flag to indicate that the sets have been
     *  computed.
     *  @return int : number of sets created
     */
    int  form_stations();

    /** get_portal() searches for a portal whose name equals name_to_find
     *  @param string [in]  name_to_find is the portal name to look up
     *  @param SubwayPortal & [out] is filled with the data from the Portal
     *         if it is found, or is an empty Portal whose name is ""
     *  @return bool true if anf only if the portal is found
     */
    bool get_portal(string name_to_find, SubwayPortal & portal) const;


    /** nearest_portal() returns a string representation of the portal that
     *  is nearest to the given point
     *  @param  double [in]  latitude of point
     *  @param  double [in]  longitude of point
     *  @return string       portal's name (as defined in subway_portal.h)
     */
    string nearest_portal( double latitude, double  longitude) const;

    /** nearest_routes() returns a string representation of the routes that
     *  are nearest to the given point
     *  @param  double [in]  latitude of point
     *  @param  double [in]  longitude of point
     *  @return string       representation of set of routes
     */
    string nearest_routes( double latitude, double  longitude) const;

};
```

6.7 The Main Program and Command Processing

I will do as I did in the first project and provide a main program in binary. It will handle all details of processing the command line, and parsing the command file. The `main.o` that I supply depends on the `SubwayPortal` class whose interface I put on the server in the same directory. This means that your

`SubwayPortal` should contain the same interface. In fact all it needs are the two constructors, not more, so as long as you name it `SubwayPortal` and provide the two constructors, it will build and link correctly.

You are free to create your own for testing your code, but your program will be built with the one I supply. The main program is on the server.

7 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:

```
subway_portal.h
subway_portal.cpp
subway_station.h
subway_station.cpp
subway_system.h
subway_system.cpp
README
Makefile
```

as well as the `subway_route.h` and binaries that I provided.

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.

8 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.

9 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.

10 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.

11 Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on May 7, 2019. Create a directory named named *username*_project2.2 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 5% for each file that does not belong there, and you will lose 5% if you do not name the directory correctly**[1].

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

```
zip -r username_project2.2.zip ./username_project2.2
```

This will compress all of your files into the file named username_project2.2.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_project2.2.zip, are not in a properly named directory, your program will lose 5%.

The submit command that you will use is submit_cs335_hwk. It requires two arguments: the number of the assignment and the pathname of your file. Thus, if your file is named username_project2.2.zip and it is in your current working directory you would type,

```
submit_cs335_hwk 6  username_project3.zip
```

since this will be the sixth assignment. The program will copy your file into the hwk6 subdirectory

```
/data/biocs/b/student.accounts/cs335_sw/hwks/hwk6/
```

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/hwks/hwk6
```

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.

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