Assignment 4

Overview

This assignment combines several different data abstractions and algorithms that we have covered in class, including priority queues, on-line disjoint set operations, hashing, and sorting. The project is a simplification of the problem of inferring relationships from social networking data.

In this assignment, you are to write a program that processes captured communication data. The input is obtained from a file named on the command line. The file consists of lines of text of various types. A ***data line*** consists of the word `data` followed by two *distinct* telephone numbers followed by a positive integer. Each valid telephone number is in the form *xxx-xxx-xxxx*, where each '*x*' is a decimal digit. The positive integer represents an amount of money transferred from the first number to the second number, in whole dollar amounts. For example, the line

```
data    807-444-2100    201-222-1200    15400
```

represents the fact that $15,400 was transferred from the owner of 807-444-2100 to the owner of 201-222-1200. The number more generally represents the strength of the relationship between the two numbers. A telephone number x is ***linked*** to a telephone number y if either x and y are the same number or a transfer of money was made between them. Two numbers x and y are ***connected*** if x and y are ***linked*** or if there is a number z such that x and z are linked and z is connected to y. A ***cohort*** is the largest set of telephone numbers such that every number in the set is connected to every other number in the set. This implies that if a telephone number is not in the cohort, then no transfer was ever made between that number and any number in the cohort. It also implies that every pair of cohorts is disjoint, and that no cohort is empty. From this definition, it follows that the *connected* relation is symmetric, reflexive, and transitive, and hence, an equivalence relation, and that *the set of all cohorts forms a partition of the set of all telephone numbers.*

The ***size*** of a cohort is the cardinality of the set, i.e., the number of telephone numbers in it. The ***volume*** of a cohort is the total amount of money transferred between members of the cohort. From the definition of connectivity, cohorts of size 1 have volume 0. The ***activity*** of a cohort is the volume divided by the total number of *unordered pairs* of distinct numbers in the cohort, which is $N(N-1)/2$ where N is the size of the cohort; activity is the average amount of money that has been transferred between any pair of members of the cohort.

When a data line is read, the program must update all cohorts to reflect possibly new relationships. For example, if the two numbers in the data line are in distinct cohorts already, then these are merged (unioned) into a single cohort. The update must include any changes in the size, volume, and activity of all cohorts.

Every cohort must be identified by a ***positive integer ID***, called its ***cohort-id***. The value zero (`0`) is reserved.

The other types of lines that may appear in the input file represent commands of various types, to be carried out by the computer. They are specified in the table below. For each command, your program must take the action indicated in the right hand column. In the table, **boldface** indicates command keywords and *italics* represent placeholders for data. `Fixed font` indicates actual data. All commands take at most one argument. Specific input validation requirements are listed below.

Command	Argument Type	Description
`find`	*phone-number*	The program should display the ID of the cohort to which the phone-number belongs, in the form *phone-number*`:` *ID* If there is no such number, it should display *phone-number*`: 0`
`members`	*cohort-id*	The program should display a list of all telephone numbers in the given cohort, one per line. If there is no cohort with the given id, it should display a single blank line.
`max-size`		The program should display the cohort-id, activity, size, and volume of the cohort having the maximum size. If two or more cohorts have the same maximum size, then all that are maximal should be displayed. If there are no cohorts, it should display a line indicating there are no cohorts.
`max-activity`		The program should display the cohort-id, activity, size, and volume of the cohort having the maximum activity. If two or more cohorts have the same maximum activity, then all that are maximal should be displayed. If there are no cohorts, it should display a line indicating there are no cohorts.
`cohort-ids`		The program should display a list of all cohort ids in order of increasing id.
`info`	*cohort-id* / `0`	The program should display 1. the cohort-id, 2. the activity, 3. the size, and 4. the volume of the cohort whose cohort-id is given. If no such id exists, it should display an error message. If the argument is `0`, then the program should display this information for all of the cohorts, sorted by increasing cohort-id. These values should be listed on a single line, in the above order, separated by tab characters.

Input And Output Details

1. The program must get the name of the input file from its only command line argument.
   (a) If there is no command line argument, it must report this as a usage statement on the standard error stream and exit.
   (b) If the file name is supplied but cannot be opened for any reason, it must report this as an I/O error on the standard error stream and exit.
2. The program must write all output onto the standard output stream, *not into a file*; to repeat, it is not to place its output into a file.
3. Valid telephone numbers have no space in them, and will have hyphens as indicated above.

4. Tokens in the input line may be preceded or followed by any number of white space characters.

5. The number of distinct telephone numbers in the input file will not exceed 16,384.

Input Validation

There are a total of seven different words that can start a line in the file. The program should catch any line that does not start with one of these words and display an error message on the standard error stream for each such line that it finds. Even if a line begins with a valid word, the program ***cannot assume that the rest of the line is in the proper format***. If the rest of the line is not valid, it should catch this and display an error message on the standard error stream. It should continue to the next line of the file after flagging the bad commands. In particular,

- The program should catch when the two telephone numbers are identical.
- It should catch when the amount of money is not a positive integer.
- It should catch when the argument to the `info` command is not a non-negative integer.

Implementation Requirements

1. You must use the union-find data structure and associated algorithm with path compression and union-by-size, as described in class, to solve this problem. All set manipulation must be carried out in a suitably defined class, and the data structure that stores the set information must be either a dynamic array or a vector. (Cohorts are sets of telephone numbers with associated properties.)

2. Telephone numbers are strings, not integers. (No arithmetic is performed with them.) Because they are strings, they are not index values of arrays or vectors. However, in the union-find algorithm description, the array implementation treats each set as consisting of simple integers, and the name of the set is simply an array index value. How can string values be used as the elements of sets? The answer is to use a hash table to represent telephone numbers:

    - When a telephone number is read by the program, the program needs to store it and access it easily. Although you could use a vector and some type of $O(\log n)$ look-up structure such as a tree for it, a hash table and a hashing scheme will allow nearly $O(1)$ look-ups. The telephone number and possibly other data is stored in a hash table entry.
    - For the union-find algorithm, each new set must be stored in the next free position in an array or vector, which suggests that there must be a connection between the hash table entry and the place in the array that stores the new set. Since the program must be able to obtain a list of all telephone numbers in a given set, and to find the name of the set to which a given telephone number belongs, and it must do these tasks efficiently, the array must have an underlying type that can store information to make these operations efficient. In particular, it needs a reference back to the hash table entry containing the telephone number.

3. You should use a priority queue for finding the maximum size or maximum activity cohort as needed. Since the maximum may be with respect to either size or activity, this suggests that either the program needs two different priority queues, or that it uses a single node type with multiple links. In this case, the priority queue node stores pointers to something. When sets are updated as a result of reading new data, the priority queue(s) may become invalid because one set may become larger or smaller with respect to one or more properties than before, and because a new set may be created, or an existing one removed. Although you could update the priority queues after each data operation, you could use the on-demand approach of leaving them in their old states, and only when the `max` command is issued, heapifying and find the maximum element. For the sake of efficiency, you might consider keeping a 'dirty' flag to test if it is necessary to re-heapify. This is your choice; you can either update your queue after every operation that changes the properties of the sets, or only "on-demand."

4. You must use heapsort to sort cohorts for display.

5. All interaction with the file system and the terminal must be performed by the main program. Classes are free to create strings to pass to the main program, or to write onto ostream objects that are passed as parameters, but they may not interact with global streams such as `cin` or `cout`.

6. Activity must be computed with floating point division and reported with two decimal places of precision. Size and volume are whole numbers.

7. Your submission must include build instructions either in the main program file or in a separate README file, or it must include a Makefile.

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 own work alone.

Testing Your Program

You should design your own input files and test your program using your own input. You should carefully check that the output of your program is correct for the inputs you gave to it. Include files with bad lines, files with no lines, files that cannot be opened, files with all kinds of spacing, and so on. Include data that creates several cohorts of equal size, volume, and activity.

Grading Rubric

The program will be graded based on the following rubric.

- If the program does not compile on a cslab machine, it receives only 20%.
- For programs that compile:
  - Correctness 40%
  - Conformance to Implementation Requirements 25%
  - Design (modularity and organization) 10%
  - Documentation: 20%
  - Style and proper naming: 5%

Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on May 15, 2017. Create a directory named named *username*`_project4`. where *username* is to be replaced by your CS Department network login name. Put all project-related source-code files 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[1].

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

```
zip -r username_project4.zip ./username_project4
```

This will compress all of your files into the file named `username_project2.zip`. Do not use the tar compress utility.

The command requires two arguments: the number of the project and the pathname of your file. Thus, if your file is named `username_project4.zip` and it is in your current working directory you would type

```
submit_cs335_assignment 4  username_project4.zip
```

[1] 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 `project2` subdirectory

```
/data/biocs/b/student.accounts/cs335_sw/projects/project4/
```

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/project4
```

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.