CS 251
Data Structures
Spring 2017

Project #06: Divvy Graph Analysis

Complete By: Tuesday, April 18th @ 11:59pm
Assignment: C++ program to perform Divvy Graph analysis
Policy: Individual work only, late work *is* accepted (see "Policy" section on last page for more details)
Submission: online via zyLabs ("Project06 Divvy Graph", section 5.12)

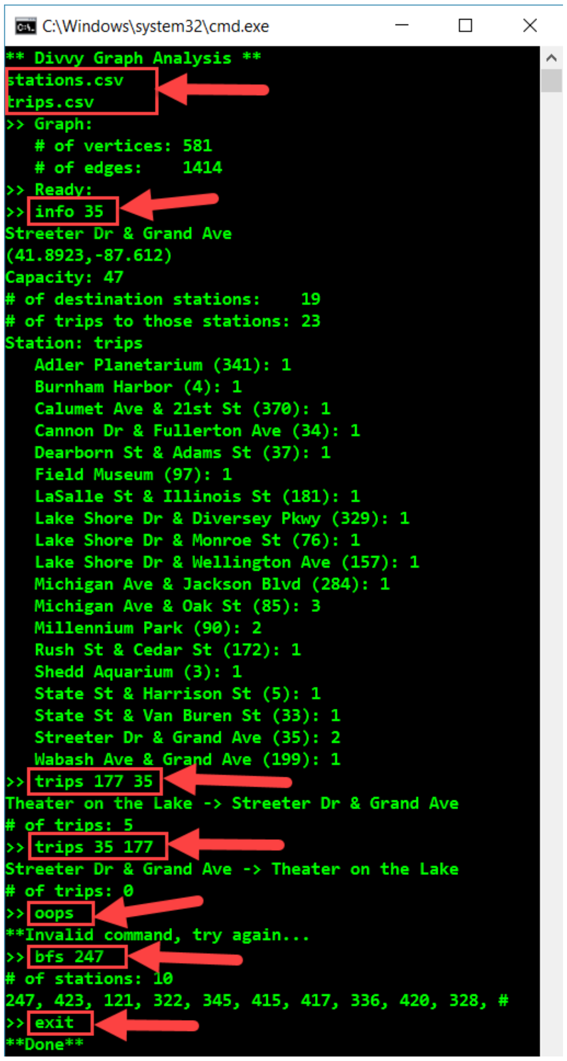
Assignment

We worked with Divvy data in an earlier project (#04). We're going to revisit the data and use a graph to analyze trips between stations. The assignment is to input Divvy station and trip data, build a graph of the # of trips between stations, and then interact with the user to support the following 5 commands:

1. **info** about a station
2. # of **trips** between 2 stations
3. breadth-first search (**bfs**) from a station
4. **debug**
5. **exit**

The data will come from 2 input files, both in CSV format (Comma-Separated Values). Your job is to write a C++ program to input this data, build a graph, and perform the requested analyses / output. You are required to use the Graph class we have been developing in class and homework (in particular HW #19 and #20), with implementation details private.

Here's a screenshot for the input files "stations.csv" and "trips.csv" --- the user input, with corresponding output, is highlighted in red. The user inputs the filenames (stations first), and then supplies one or more commands to the program. Your program must match this output exactly, since the first level of grading will use the zyLabs system.



```
** Divvy Graph Analysis **
stations.csv
trips.csv
>> Graph:
   # of vertices: 581
   # of edges:    1414
>> Ready:
>> info 35
Streeter Dr & Grand Ave
(41.8923,-87.612)
Capacity: 47
# of destination stations:    19
# of trips to those stations: 23
Station: trips
   Adler Planetarium (341): 1
   Burnham Harbor (4): 1
   Calumet Ave & 21st St (370): 1
   Cannon Dr & Fullerton Ave (34): 1
   Dearborn St & Adams St (37): 1
   Field Museum (97): 1
   LaSalle St & Illinois St (181): 1
   Lake Shore Dr & Diversey Pkwy (329): 1
   Lake Shore Dr & Monroe St (76): 1
   Lake Shore Dr & Wellington Ave (157): 1
   Michigan Ave & Jackson Blvd (284): 1
   Michigan Ave & Oak St (85): 3
   Millennium Park (90): 2
   Rush St & Cedar St (172): 1
   Shedd Aquarium (3): 1
   State St & Harrison St (5): 1
   State St & Van Buren St (33): 1
   Streeter Dr & Grand Ave (35): 2
   Wabash Ave & Grand Ave (199): 1
>> trips 177 35
Theater on the Lake -> Streeter Dr & Grand Ave
# of trips: 5
>> trips 35 177
Streeter Dr & Grand Ave -> Theater on the Lake
# of trips: 0
>> oops
**Invalid command, try again...
>> bfs 247
# of stations: 10
247, 423, 121, 322, 345, 415, 417, 336, 420, 328, #
>> exit
**Done**
```

Input Files

The input files are text files in **CSV** format — i.e. comma-separated values. The first file defines the set of **stations**, in no particular order. An example of the format is as follows:

```
id,name,latitude,longitude,dpcapacity,online_date
456,2112 W Peterson Ave,41.991178,-87.683593,15,5/12/2015
101,63rd St Beach,41.78101637,-87.57611976,23,4/20/2015
109,900 W Harrison St,41.874675,-87.650019,19,8/6/2013
.
.
.
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each data line represents one Divvy station for docking/undocking bicycles. Each station contains 6 values: a unique station ID (integer), the stations name (string), the position of the station in (longitude, latitude) format, the capacity of the station (integer, # of bikes you can dock there), and the date the station came online (string). Do not assume the station IDs are contiguous 1..N, they are not.

The second file defines a set of **trips**, in no particular order. With Divvy, a “bike trip” means a rider checks out a bike from one station, rides it around the city, and then checks it back into the same station, or a different station. The main purpose of Divvy is commuting, so most bike rides are short — less than 30 minutes. Here’s an example of the file format (you may need to open one of the data files to see this better):

```
trip_id,starttime,stoptime,bikeid,tripduration,from_station_id,from_station_name,to_station_id,to_station_name,usertype,gender,birthyear
10426648,6/30/2016 23:57,7/1/2016 0:22,4050,1466,259,California Ave & ...,123,California Ave & ...,Subscriber,Female,1986
10426638,6/30/2016 23:55,7/1/2016 0:40,4579,2713,177,Theater on the Lake,340,Clark St & Wrightwood Ave,Customer,,
.
.
.
```

The first line contains column headers, and should be ignored. The data starts on line 2, and each data line represents one Divvy trip. Each data line consists of 12 values:

• Trip id:	integer
• Start time:	date and time of trip start
• Stop time:	data and time of trip end
• Bike id:	integer
• Trip duration:	integer, in seconds
• From station id:	integer, where bike was checked out
• From station name:	string, where bike was checked out (**this value cannot be trusted --- sometimes wrong**)
• To station id:	integer, where bike was returned
• To station name:	string, where bike was returned (**this value cannot be trusted --- sometimes wrong**)
• User type:	a registered “subscriber” or a new “customer”
• Gender:	optional string, may be missing
• Birth year:	optional integer, may be missing

These values do not need to be stored --- they are used to augment the edge weights in the graph. In case you’re curious, the Divvy data is available for browsing on Chicago’s data portal; the raw data was downloaded from Divvy @ https://www.divvybikes.com/data.

Program Functionality

The Divvy system is best represented as a graph, where the stations are vertices and trips are edges between stations. In our graph, if there's an edge from station X to station Y, that means someone made a trip from X to Y: he/she checked out a bike from station X, rode it to station Y, and checked it in at station Y. In addition, the weight along this edge will represent the total # of trips that were taken from station X to station Y. The weight thus represents how popular common it is for a trip from station X to station Y. [*This idea is inspired by the visualization here, where the weight corresponds to the thickness of the edge.*]

The program starts by first inputting the stations from the stations file, adding each station as a vertex to the graph --- the vertex name should be the name of the station. You'll also need to store the station info in a separate data structure so that (a) you can output it as part of the "info" command, and (b) map from station id to station name (and vice versa). Define a class **Station** to hold the data, and then store the station objects however you want (since the # of stations is relatively small, a simple **vector<Station>** is fine).

Once the vertices have been added to the graph, then input the trip data from the trips file, and add / update edges as needed. The first time you encounter a trip from X to Y, add an edge from X to Y with a weight of 1; when you encounter subsequent trips from X to Y, increment the existing edge weight. After you have processed the trips file, the edge weights represent the # of trips between every pair of stations.

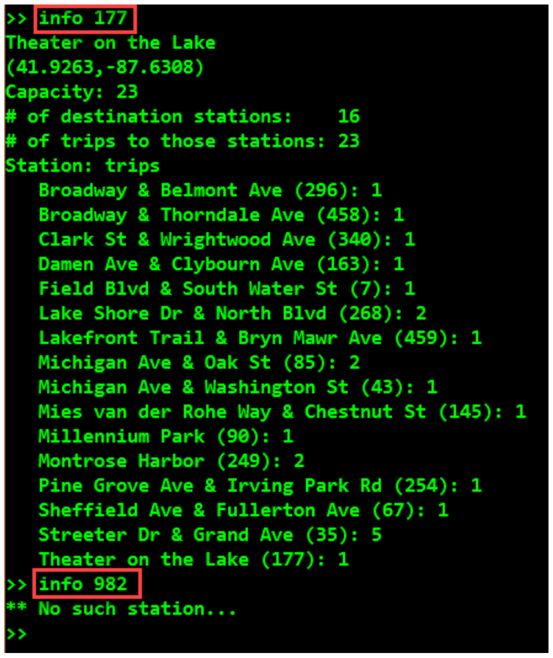
Once the graph is built, the user can input any number of commands. For simplicity assume the input is valid in the sense that if the user needs to input an integer, he/she will do so. The 4 commands are "info", "trips", "bfs", and "exit".

1. info

The "info" command provides info about a station. The user inputs a station id, and the program outputs the following information:

- Name
- Position
- Capacity
- # of adjacent stations in the graph
- Total # of trips to those adjacent stations
- A list of each adjacent station (name and id) with # of trips to that station

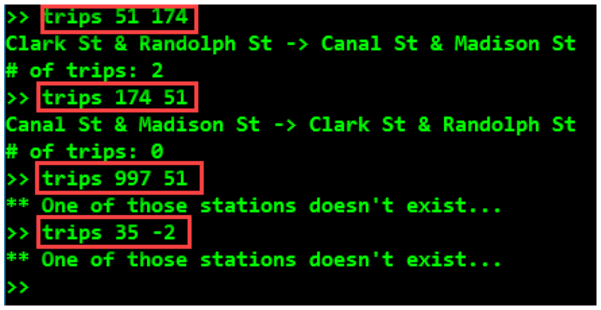
An example is shown in the screenshot to the right. If the user inputs the id of a station that does not exist, say so.



```
>> info 177
Theater on the Lake
(41.9263,-87.6308)
Capacity: 23
# of destination stations:    16
# of trips to those stations: 23
Station: trips
   Broadway & Belmont Ave (296): 1
   Broadway & Thorndale Ave (458): 1
   Clark St & Wrightwood Ave (340): 1
   Damen Ave & Clybourn Ave (163): 1
   Field Blvd & South Water St (7): 1
   Lake Shore Dr & North Blvd (268): 2
   Lakefront Trail & Bryn Mawr Ave (459): 1
   Michigan Ave & Oak St (85): 2
   Michigan Ave & Washington St (43): 1
   Mies van der Rohe Way & Chestnut St (145): 1
   Millennium Park (90): 1
   Montrose Harbor (249): 2
   Pine Grove Ave & Irving Park Rd (254): 1
   Sheffield Ave & Fullerton Ave (67): 1
   Streeter Dr & Grand Ave (35): 5
   Theater on the Lake (177): 1
>> info 982
** No such station...
>>
```

2. trips

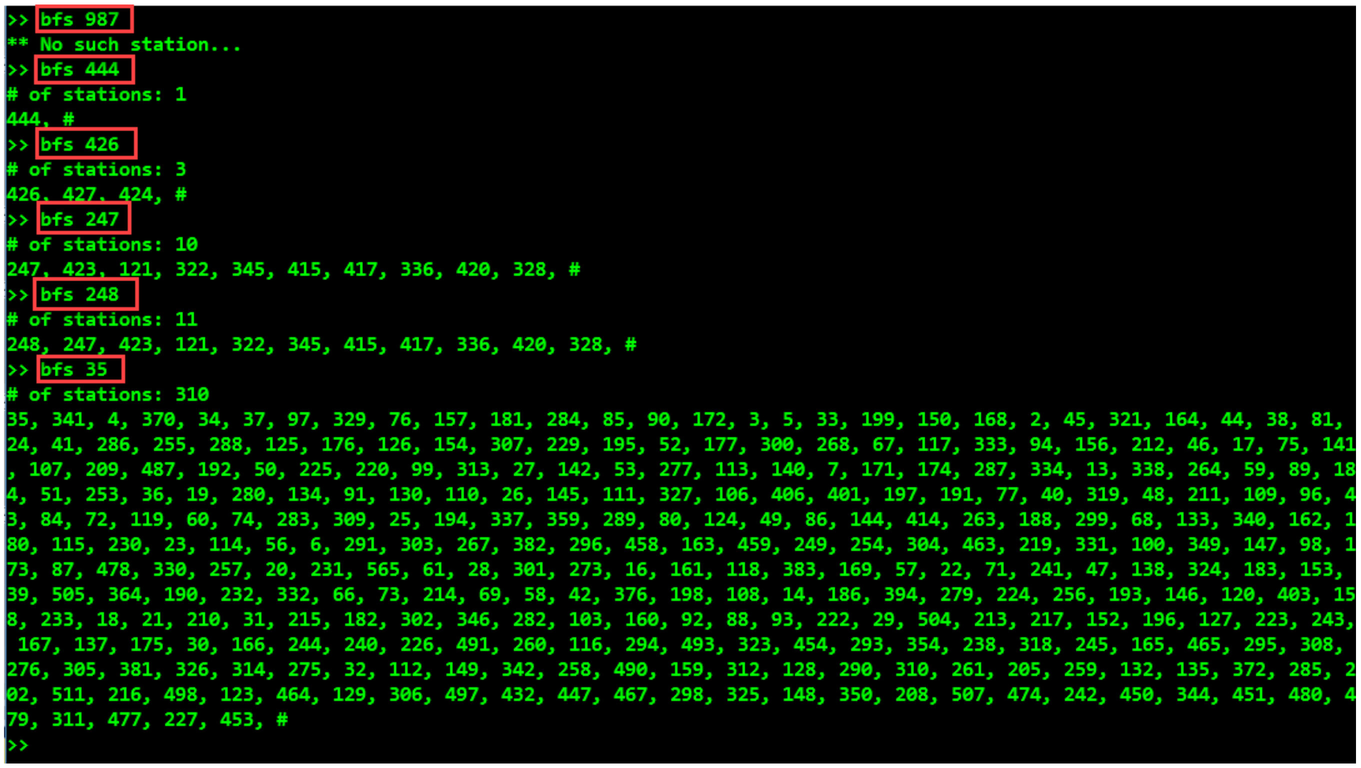
Given the starting station id and the destination station id, reports the # of trips taken from the starting station to the destination station. This is simply the weight on the edge from the starting station's vertex to the destination station's vertex. Examples are shown to the right. If the user enters a station id that doesn't exist, say so.



```
>> trips 51 174
Clark St & Randolph St -> Canal St & Madison St
# of trips: 2
>> trips 174 51
Canal St & Madison St -> Clark St & Randolph St
# of trips: 0
>> trips 997 51
** One of those stations doesn't exist...
>> trips 35 -2
** One of those stations doesn't exist...
>>
```

3. bfs

The "bfs" command performs a breadth-first search, starting from the given station id. The returned vector is then output to the screen in order. To reduce the amount of output, the station ids are output on the screen instead of the station names. If the user inputs a station id that doesn't exist, say so.



```
>> bfs 987
** No such station...
>> bfs 444
# of stations: 1
444, #
>> bfs 426
# of stations: 3
426, 427, 424, #
>> bfs 247
# of stations: 10
247, 423, 121, 322, 345, 415, 417, 336, 420, 328, #
>> bfs 248
# of stations: 11
248, 247, 423, 121, 322, 345, 415, 417, 336, 420, 328, #
>> bfs 35
# of stations: 310
35, 341, 4, 370, 34, 37, 97, 329, 76, 157, 181, 284, 85, 90, 172, 3, 5, 33, 199, 150, 168, 2, 45, 321, 164, 44, 38, 81,
24, 41, 286, 255, 288, 125, 176, 126, 154, 307, 229, 195, 52, 177, 300, 268, 67, 117, 333, 94, 156, 212, 46, 17, 75, 141
, 107, 209, 487, 192, 50, 225, 220, 99, 313, 27, 142, 53, 277, 113, 140, 7, 171, 174, 287, 334, 13, 338, 264, 59, 89, 18
4, 51, 253, 36, 19, 280, 134, 91, 130, 110, 26, 145, 111, 327, 106, 406, 401, 197, 191, 77, 40, 319, 48, 211, 109, 96, 4
3, 84, 72, 119, 60, 74, 283, 309, 25, 194, 337, 359, 289, 80, 124, 49, 86, 144, 414, 263, 188, 299, 68, 133, 340, 162, 1
80, 115, 230, 23, 114, 56, 6, 291, 303, 267, 382, 296, 458, 163, 459, 249, 254, 304, 463, 219, 331, 100, 349, 147, 98, 1
73, 87, 478, 330, 257, 20, 231, 565, 61, 28, 301, 273, 16, 161, 118, 383, 169, 57, 22, 71, 241, 47, 138, 324, 183, 153,
39, 505, 364, 190, 232, 332, 66, 73, 214, 69, 58, 42, 376, 198, 108, 14, 186, 394, 279, 224, 256, 193, 146, 120, 403, 15
8, 233, 18, 21, 210, 31, 215, 182, 302, 346, 282, 103, 160, 92, 88, 93, 222, 29, 504, 213, 217, 152, 196, 127, 223, 243,
 167, 137, 175, 30, 166, 244, 240, 226, 491, 260, 116, 294, 493, 323, 454, 293, 354, 238, 318, 245, 165, 465, 295, 308,
276, 305, 381, 326, 314, 275, 32, 112, 149, 342, 258, 490, 159, 312, 128, 290, 310, 261, 205, 259, 132, 135, 372, 285, 2
02, 511, 216, 498, 123, 464, 129, 306, 497, 432, 447, 467, 298, 325, 148, 350, 208, 507, 474, 242, 450, 344, 451, 480, 4
79, 311, 477, 227, 453, #
>>
```

4. debug

The "debug" command simply calls the **PrintGraph** function in the graph class, with the title "Divvy Graph". This is one way to see if you have built the graph correctly.

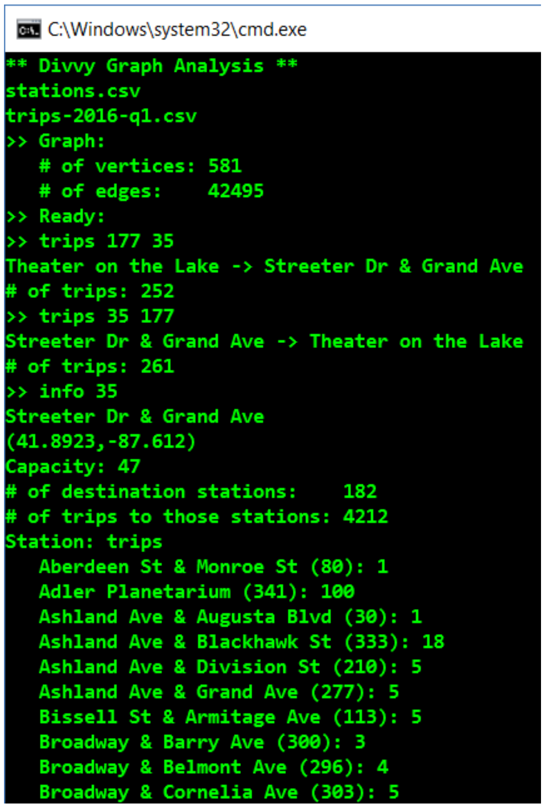
5. exit

The user types "exit" when he/she wishes to stop.

Requirements

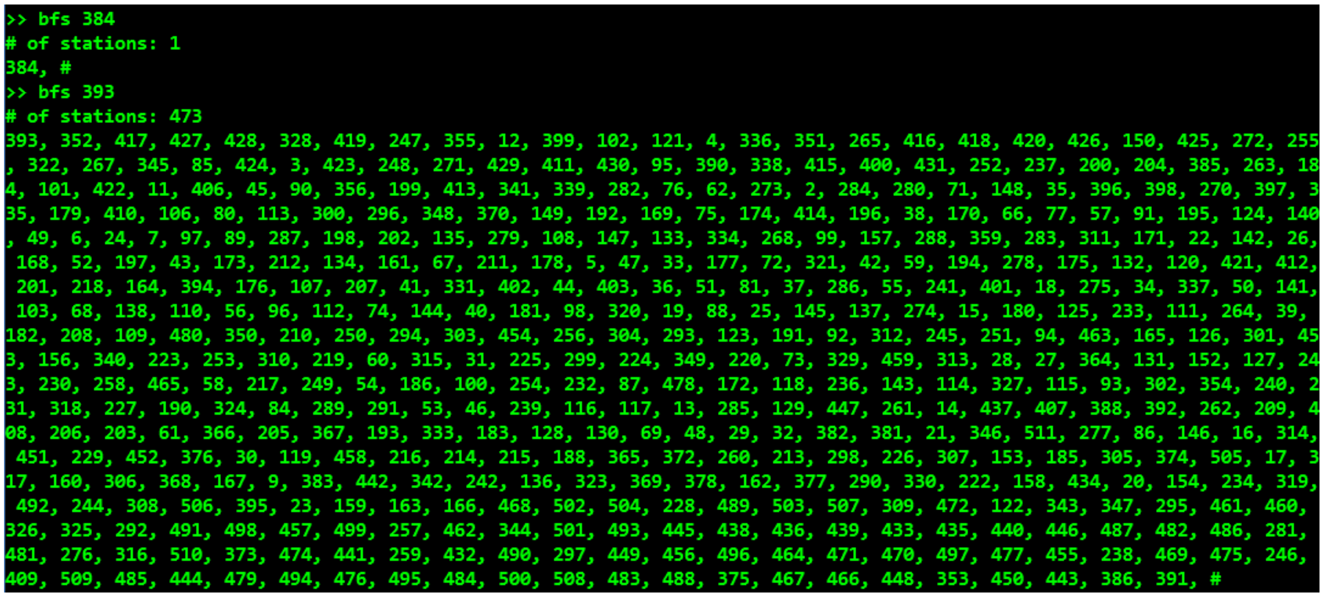
You are required to use C++, along with the Graph class we have been developing in class and homeworks (HW #19 and #20). Part of the exercise is to also work on "proper" software design --- e.g. what's the right way to update an edge's weight from the main() program? The proper approach is to add public functionality to the Graph class that allows an edge to be found & updated; the wrong way is to make the private implementation details public and just let the main() program reach into the linked-list and update the weight. Programs that take the latter approach of making the private implementation details public will not be graded, and instead scored 0/100.

You may assume at most 1,000 stations (vertices); we'll see how to lift that restriction later. You are not required to free memory, but go ahead if you want to experiment (and verify with valgrind). Do not get overly concerned about performance in this assignment; we'll focus on that in the next project. This assignment can be done efficiently with the existing Graph implementation of arrays and linked-lists, with a simple unordered vector of Station objects in the main() program. Feel free to optimize if you want, but focus on correctness first.



```
** Divvy Graph Analysis **
stations.csv
trips-2016-q1.csv
>> Graph:
   # of vertices: 581
   # of edges:    42495
>> Ready:
>> trips 177 35
Theater on the Lake -> Streeter Dr & Grand Ave
# of trips: 252
>> trips 35 177
Streeter Dr & Grand Ave -> Theater on the Lake
# of trips: 261
>> info 35
Streeter Dr & Grand Ave
(41.8923,-87.612)
Capacity: 47
# of destination stations:    182
# of trips to those stations: 4212
Station: trips
   Aberdeen St & Monroe St (80): 1
   Adler Planetarium (341): 100
   Ashland Ave & Augusta Blvd (30): 1
   Ashland Ave & Blackhawk St (333): 18
   Ashland Ave & Division St (210): 5
   Ashland Ave & Grand Ave (277): 5
   Bissell St & Armitage Ave (113): 5
   Broadway & Barry Ave (300): 3
   Broadway & Belmont Ave (296): 4
   Broadway & Cornelia Ave (303): 5
```

We are releasing additional test files, in particular a much larger "trips-2016-q1.csv" file to test how your Graph class handles more edges. While additional data structures are not needed (i.e. the arrays and linked-lists are sufficient), you *DO* need to have the compiler optimize the program, otherwise it may take 5+ minutes to input the data and build the graph. With g++ you want to use the -O option; with Visual Studio, you want to switch from "Debug" mode to "Release" mode via the drop-down on the toolbar.



```
>> bfs 384
# of stations: 1
384, #
>> bfs 393
# of stations: 473
393, 352, 417, 427, 428, 328, 419, 247, 355, 12, 399, 102, 121, 4, 336, 351, 265, 416, 418, 420, 426, 150, 425, 272, 255
, 322, 267, 345, 85, 424, 3, 423, 248, 271, 429, 411, 430, 95, 390, 338, 415, 400, 431, 252, 237, 200, 204, 385, 263, 18
4, 101, 422, 11, 406, 45, 90, 356, 199, 413, 341, 339, 282, 76, 62, 273, 2, 284, 280, 71, 148, 35, 396, 398, 270, 397, 3
35, 179, 410, 106, 80, 113, 300, 296, 348, 370, 149, 192, 169, 75, 174, 414, 196, 38, 170, 66, 77, 57, 91, 195, 124, 140
, 49, 6, 24, 7, 97, 89, 287, 198, 202, 135, 279, 108, 147, 133, 334, 268, 99, 157, 288, 359, 283, 311, 171, 22, 142, 26,
 168, 52, 197, 43, 173, 212, 134, 161, 67, 211, 178, 5, 47, 33, 177, 72, 321, 42, 59, 194, 278, 175, 132, 120, 421, 412,
 201, 218, 164, 394, 176, 107, 207, 41, 331, 402, 44, 403, 36, 51, 81, 37, 286, 55, 241, 401, 18, 275, 34, 337, 50, 141,
 103, 68, 138, 110, 56, 96, 112, 74, 144, 40, 181, 98, 320, 19, 88, 25, 145, 137, 274, 15, 180, 125, 233, 111, 264, 39,
182, 208, 109, 480, 350, 210, 250, 294, 303, 454, 256, 304, 293, 123, 191, 92, 312, 245, 251, 94, 463, 165, 126, 301, 45
3, 156, 340, 223, 253, 310, 219, 60, 315, 31, 225, 299, 224, 349, 220, 73, 329, 459, 313, 28, 27, 364, 131, 152, 127, 24
3, 230, 258, 465, 58, 217, 249, 54, 186, 100, 254, 232, 87, 478, 172, 118, 236, 143, 114, 327, 115, 93, 302, 354, 240, 2
31, 318, 227, 190, 324, 84, 289, 291, 53, 46, 239, 116, 117, 13, 285, 129, 447, 261, 14, 437, 407, 388, 392, 262, 209, 4
08, 206, 203, 61, 366, 205, 367, 193, 333, 183, 128, 130, 69, 48, 29, 32, 382, 381, 21, 346, 511, 277, 86, 146, 16, 314,
 451, 229, 452, 376, 30, 119, 458, 216, 214, 215, 188, 365, 372, 260, 213, 298, 226, 307, 153, 185, 305, 374, 505, 17, 3
17, 160, 306, 368, 167, 9, 383, 442, 342, 242, 136, 323, 369, 378, 162, 377, 290, 330, 222, 158, 434, 20, 154, 234, 319,
 492, 244, 308, 506, 395, 23, 159, 163, 166, 468, 502, 504, 228, 489, 503, 507, 309, 472, 122, 343, 347, 295, 461, 460,
326, 325, 292, 491, 498, 457, 499, 257, 462, 344, 501, 493, 445, 438, 436, 439, 433, 435, 440, 446, 487, 482, 486, 281,
481, 276, 316, 510, 373, 474, 441, 259, 432, 490, 297, 449, 456, 496, 464, 471, 470, 497, 477, 455, 238, 469, 475, 246,
409, 509, 485, 444, 479, 494, 476, 495, 484, 500, 508, 483, 488, 375, 467, 466, 448, 353, 450, 443, 386, 391, #
```

Getting Started

The course web page contains a set of files to help you get started. Look under "Projects", and then "project06-files". You'll find the following files:

1. main.cpp
2. stations.csv
3. trips.csv
4. trips-2016-q1.csv
5. trips-t2.csv

These files are also available within Zyante, section 5.12. Since most everyone is new to C++, some starter code is provided in "main.cpp" to help you get started. A couple highlights of the provided code:

```
class Station  // defines a Station object
{
  ...
}

//
// Inputs the stations from the given file, adding each station name as a
// vertex to the graph, and storing a new Station object into the vector.
// The graph is passed by reference --- note the & --- so that the changes
// made by the function are returned back.  The vector of Station objects is
// returned by the function.
//
vector<Station> InputStations(Graph& G, string filename)
{
  ...
}

//
// Inputs the trips, adding / updating the edges in the graph.  The graph is
// passed by reference --- note the & --- so that the changes made by the
// function are returned back.  The vector of stations is needed so that
// station ids can be mapped to names to lookup in the graph; it is passed by
// reference only for efficiency (so that a copy is not made).
//
void ProcessTrips(string filename, Graph& G, vector<Station> stations)
{
  ...
}
```

Passing by reference is much easier in C++. Just pass the parameter as you normally would, and the function definition simply adds **&** to denote pass by reference. Now the function can change without the need for explicit pointers. To compile the program at the command-line, use **g++** as follows:

```
g++ -std=c++11 -Wall main.cpp graph.cpp -o project06
```

You will need to provide your own versions of "graph.h" and "graph.cpp" files. If you are working in Visual

Studio or another IDE, create a new project and then add the source and headers files to the project: main.cpp, graph.cpp, and graph.h. [Note: C++11 is not available on bert / ernie, but is available on bertvm.]

Submission

The program is to be submitted via zyLabs: see "**Project06 Divvy Graph**", section 5.12. Submit your three files in the tabs provided: main.cpp, graph.h, and graph.cpp. When you're ready to submit, copy-paste your program into the zyLabs editor pane, switch to "Submit" mode, and click "SUBMIT FOR GRADING".

Keep in mind this is only a first level grading mechanism to ensure you have (a) submitted the correct files, and (b) are following the basic requirements and passing some simple tests. The grade you receive from your zyLabs submission is only a preliminary grade. After the deadline, the TAs will then manually review and test each program for the following:

- adherence to requirements
- overall design
- correctness against additional tests (involving other input files)
- commenting, indentation, whitespace and readability

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 25%. After 24 hours, no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is described here:

http://www.uic.edu/depts/dos/docs/Student%20Disciplinary%20Policy.pdf .

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml .