Assignment: Project: Implementing Graphs

You have not submitted. You must earn 23/28 points to pass.

Deadline Pass this assignment by декабрь 27, 11:59 вечера PT

Instructions (/learn/advanced-data-structures/programming/8IM0S/project-implementing-graphs) Discussions (/learn/advanced-data-structures/programming/8IM0S/project-implementing-graphs/discussions)

Project for Week 1: Implementing basic graphs for real transportation data

A PDF of these instructions will be made available soon...

If at any point this week you feel like you are stuck or need some more hints, check out the support videos for the week. They not only provide you with some hints for completing the methods below, but also discuss a cool trick for computing the 2-hop neighbors of a vertex using matrix multiplication!

Getting Set Up

Before you begin this assignment, make sure you have the starter code and that you check Part 2 in the setup guide (https://www.coursera.org/learn/advanced-data-structures/supplement/cREMI/setting-up-java-eclipse-and-the-starter-code) to make sure the starter code has not changed since you downloaded it. If there have been any changes, follow the instructions in the setup guide for updating your code base before you begin.

Also make sure you have read through the starter code and front end orientation guide (https://www.coursera.org/learn/advanced-data-structures/supplement/LxK6a/project-orientation-to-the-starter-code-data-files-and-front-end) so you are familiar with what is included with the starter code.

In this project, you will implement the Graph ADT and compare graphs that come from real transportation data mapping intersections, roads, and flights.

Open the starter code for this week by expanding the basicgraph folder to see the package basicgraph. You will add code to the files Graph.java, GraphAdjMatrix.java and GraphAdjList.java .

Before you begin coding, make sure you know the definitions of a graph and the two implementation strategies for nodes and edges discussed in the core videos (https://www.coursera.org/learn/advanced-data-structures/lecture/PEHpu/core-graph-definitions): adjacency matrices and adjacency lists.

Assignment and Submission Details

Your goals in this assignment are to see how real life data can be modeled by graphs, and how implementation choices affect the performance of the algorithms. You'll also explore how different data properties impact the structure of the graph, both by computing the degree sequence of the graph and by visualizing the graph structure using a tool we provide.

Part 1: Implement the degreeSequence method

Start by tracing the code in the basicgraph package. There are three classes: one abstract (Graph.java) and two concrete (GraphAdjMatrix.java and GraphAdjList.java). The concrete classes give two possible implementations of the edge relation in a graph: one using an adjacency matrix and the other using an adjacency list. The first goal of this assignment is to work through these two implementations and find both their similarities and differences when working with real data.

implementations and find both their similarities and differences when working with real data.

The following two methods are already provided in the starter code:

getNeighbors(int v) an abstract method in Graph.java that is implemented in both derived classes. When given the input vertex index v, this method returns a list containing the integer indices of vertices appearing as the end point of edges starting at v. Note: if there are multiple edges between v and one of its neighbors, then that neighbor appears multiple times in this list.

getInNeighbors(int v) similar to above except returning list of indices of vertices which are the start points of edges that end at v.

Your task is to complete the following method in the Graph class (in Graph.java):

public List<Integer> degreeSequence( )

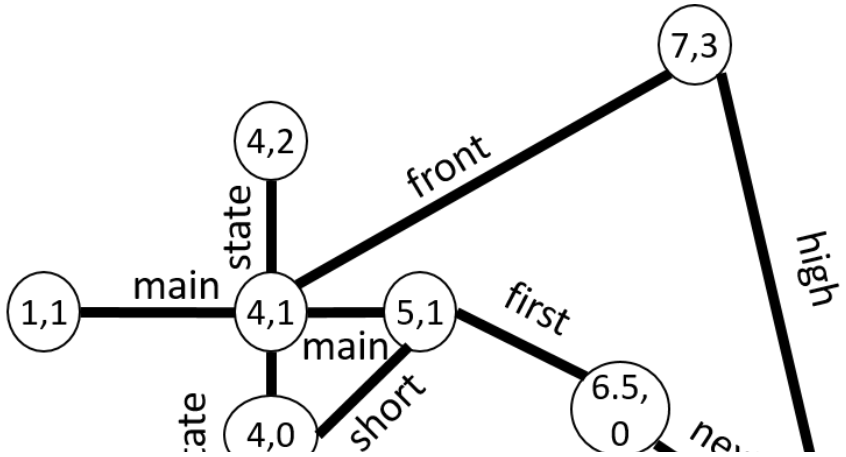
The degree sequence of a graph is a sorted (organized in numerical order from largest to smallest, possibly with repetitions) list of the degrees of the vertices in the graph.

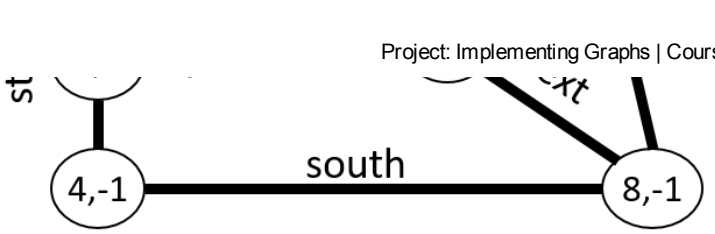
The degree sequence (https://en.wikipedia.org/wiki/Degree_(graph_theory)) can tell us a lot about the underlying structure of the graph: Are there **hubs**, vertices that are adjacent to many others? Are there **isolated vertices**, vertices that have no neighbors? Is the graph (almost) **regular**, with all vertices roughly adjacent to the same number of other vertices? The amazing thing is that even though these all sound like abstract technical notions, they have real meaning in our applications. For example, a hub in the airline network data is a major airport with many nonstop flights leaving and arriving through it (for example, London's Heathrow International airport or Chicago's O'Hare International airport). If a snowstorm grounds all flights at a hub airport, the impact on the transportation network on a whole is far greater than if the snowstorm were to impact a small regional airport.

Testing your code

Our main method in Graph.java provides a skeleton for testing the methods you wrote in this section. Use and add to this method to do thorough testing of your methods. The starter code and front end orientation guide (https://www.coursera.org/learn/advanced-data-structures/supplement/LxK6a/project-orientation-to-the-starter-code-data-files-and-front-end) gives you an overview of the data provided for testing. We recommend you test your methods with at least the following files:

- data/testdata/simpletest.map A file with simulated road data, representing the following (fake) street map:





All edges in the above map are considered 2-way. That is, for each line in the map, there are actually two edges in the graph, one in each direction.

You can use this very simple map for testing, and you can also look at it to see how the map files are set up. You can make a copy of this file and modify it to modify the map you are testing with.

Next, you can test on real-world data. However, before running your code on the real data files we provide (see below for descriptions of these files), make some predictions:

1. What will be the maximum degree of the road intersections graph?
2. What will be the maximum degree of the nonstop flights graph?
3. Which of these graphs will be more regular?

Here are the real-world files:

- data/maps/ucsd.map: A small region of the streets near the UCSD campus. This is real-world data

There are also several other real-world road data files in the data/maps folder. You probably want to stick with the ones that are labeled small until you're reasonably sure things are working correctly.

- data/airports/routesUA.dat: A list of the non-stop routes that United Airlines files.
- Custom data of your choosing (optional): If you like, you can generate custom raw-map files for any part of the world that you are interested in. Follow the instructions at the end of the starter code and front end orientation guide (https://www.coursera.org/learn/advanced-data-structures/supplement/LxK6a/project-orientation-to-the-starter-code-data-files-and-front-end). Make sure to generate the .intersections file also. You'll use it below.

Visualizing the data

In addition to submitting your code implementing the degree sequence (see instructions below), you can confirm your work with the visualization tool we're providing. Note that this file works only for the files in the data/intersections folder, or any custom .intersections files you have created.

To run the tool, download and open the following HTML file:

OSM_Graph_Viewer.0.1.4.html (https://d3c33hcgiwev3.cloudfront.net/_d09434285f7...

You can load data into this graph viewer using the following process:

1. Open the file containing the intersection data you want to visualize. NOTE: This must be a .intersections file and not the raw .map files. Copy everything in the file and paste it into the window in the graph viewer window.
2. Click "Import" to load the data.

Play around with the graph visualization tool for different data. What would the graph structure of the intersections of a busy downtown city look like? How about the intersections around a major highway?

Part 1: What and how to submit

Create a zip file containing Graph.java, GraphAdjList.java, and GraphAdjMatrix.java. Submit this file for part 1 of the assignment. We'll run some tests to make sure your code is correct - if it's not, we'll let you know.

As with all Coursera custom graders, it takes a few minutes. While the grader is running, you will see a 0 for your score. Do not be alarmed. This just means the grader is still running. When it completes the page will refresh.

If you found you had errors, and you can't figure them out, we've provided the Java code we use for testing your files. DegreeGrader in the basicgraph package is our part 1 grader, and the graph files it uses can be found in data/graders/mod1.

Part 2: Implement the getDistance2 method in both GraphAdjList and GraphAdjMatrix implementations

In this step of the assignment, you'll implement a method which takes the name of a node in the graph and returns the list of nodes that can be reached in two hops. Just like the getNeightbors( ) and getInNeighbors( ) methods, the implementation of the abstract method must go in the classes GraphAdjList and GraphAdjMatrix.

public List<Integer> getDistance2(int v)

To facilitate auto-grading, you need to read the method descriptions carefully to ensure they have the correct behavior. Also, make sure you are consistent with our test cases/examples in the main method.

In particular, if there are multiple two-hop paths to get to a vertex, then that vertex should be listed multiple times.

Hints:

- For the list representation you'll use a hard-coded search to implement **getDistance2(int v)**. In other words, first find the neighbors of v, and then expand them out for the two-hop cities.
- For the matrix representation, **getDistance2(int v)** can be implemented with matrix multiplication (square the matrix then read the non-zero entries from the appropriate row). The support video guides you through some more of the details. However, implementing this method using general matrix multiplication is not required as long as your method works correctly.

Part 2 Submission

Again create a zip file containing Graph.java, GraphAdjList.java, and GraphAdjMatrix.java. Submit this file for part 2 of the assignment. We'll run tests on your getDistance2 functions, and you'll get feedback on any tests that fail.

GraphGrader in the basicgraph package is our part 2 grader, and the graph files it uses can be found in data/graders/mod1.

How to submit

When you're ready to submit, you can upload files for each part of the assignment on the "My submission" tab.