Homework 6: Graph Algorithms - Programming

Due by 11:59PM EST on Thursday, November 5, 2020

Required Problems (110 points)

Setup and Logistics

We have put together several stub files — **click here to download them (/~cis121/current/hw/6-graphs/hw6-stub-files.zip)**.

At least 8 files to submit: `Graph.java` , `GraphTest.java` , `MazeSolver.java` , `MazeSolverTest.java` , `IslandBridge.java` , `IslandBridgeTest.java` , `Dijkstra.java` , `DijkstraTest.java` , **Make sure to submit any other files you write which are required to compile your code!**

This assignment is longer than the previous ones, and you have two weeks. Each part will involve implementing and applying graph algorithms you learned in class to solve different problems!

It's imperative you start early! We're giving you lots of time to complete it.

Be sure to follow these guidelines:

- Ensure source files are not located in any package and that none of the files have a package declaration (e.g. `package src;`).
- When submitting, submit each file individually at the root level (not contained in a folder or zip files).
- Ensure that your own test files pass and do not run into infinite loops.
- You have unlimited submissions until the deadline on the autograder marked `[compilation]` , and **two free submissions** on the autograder marked `[full]` .
- For this assignment:
    - **Number of Free Submissions**: 2
    - **Deduction Per Additional Submission**: 10 points
    - **Max Code Coverage Deduction**: up to 20 points
    - **Max Style Score Deduction**: up to 5 points
    - **Manual TA Efficiency/Style Grading**: yes
- Do not wait until the last minute to try submitting. You are responsible for ensuring your submission goes through by the deadline.
- Before asking questions on Piazza, be sure to check and keep updated with the pinned clarifications post.
- **Gradescope issues?** Be sure to check out our **Gradescope help page (/~cis121/current/gradescope_help.html)** before posting to Piazza.

Motivation: Network Connectivity

You can think of the Internet as a graph where machines and routers are vertices, and the wires between them are edges. Various questions concern the implementation and structure of the Internet, such as “How many hops must a message take to reach a destination?” and “What is the cheapest way to connect and route all computers to each other?”

In this assignment, you will implement several data structures and algorithms to answer these questions.

Part 0: Binary Min-Heap (0 Points)

Files to submit: None

We will provide a Gradescope submission so you can test your implementation from the Huffman assignment with unlimited submissions allowed and instant feedback. This submission will not count towards your grade. It is necessary to have a working version so that you can test your Dijkstra implementation which will make use of the `BinaryMinHeapImpl` class.

To test your heap, submit *only* `BinaryMinHeapImpl.java` to the Binary Min-Heap Autograder on Gradescope. You will have unlimited submissions.

Regardless, a working BinaryMinHeap will be provided when you submit to the autograder for this assignment. That is, you will **not** be submitting `BinaryMinHeapImpl` ; we will run your code by providing a working version ourselves.

Part 1: Graph Representations (20 Points)

Files to submit: `Graph.java` , `GraphTest.java`

Since this homework will require implementing algorithms that run on both directed and undirected graphs, you will implement an optionally weighted-directed graph in `Graph.java` . That is, you have the option to use it as a weighted or unweighted graph, as well as a directed or undirected graph.

To use this class as a weighted/unweighted graph, you can choose to ignore edge weights to treat it like an unweighted graph and take into account the weights for a weighted (think about how BFS doesn’t use edge weights but Dijkstra’s does).

To use this class as a directed graph, each edge represents a single direction edge between two vertices. That is, to add the edge from *u* to *v*, add an edge from *u* to *v*.

To use this class as an undirected graph, inserting an *undirected* edge between *u* and *v* can be accomplished by inserted an edge from *u* to *v* AND an edge from *v* to *u*. That is, we are representing an undirected edge as two directed edges.

The implementation details are largely up to you, but be sure to consider the runtimes of each operation, which are provided in `Graph.java` . Please make sure that your solution only uses $O(m+n)$ $O(m+n)$ space - **don’t use adjacency matrices!**

Also, to reach the time complexity we require, your adjacency list must be implemented as an array of `HashMap` s. That is, instead of using linked lists to maintain the outgoing edges for each vertex, you should use hashmaps that map the endpoints of the edges to the edge weights.

Your solution *does not* need to handle self-loop and parallel edges, but it must support negative edge weights. Overall, the graph implementation is largely up to you for design as long as it meets the space and time requirements. The graph implementation also assumes that the vertices will be represented by integers in $[0..n-1]$ $[0..n-1]$. The stub file provides more specific details.

For Fun: Defensive Programming

Something else to keep in mind is what data your methods expose. Remember, you should practice *defensive coding*. If you return the data structures containing your graph to a client, then they can modify them as they wish (which is very bad!). Thus, you should use `Collections.unmodifiableXXX` (where the XXX is Map, Set, etc.) to ensure that the user cannot modify the data structures you are passing them. We won't be checking/grading for this, so this is just for fun!

Part 2: Let's Talk About Design

Now that you have your basic data structures set up, it's time to get into the algorithms. Take some time to read through the problems below, and think through what you need to solve them. Some problems may require you to write BFS, DFS, etc., so it may be a good idea for you to abstract this logic so that you do not need to code the same thing repeatedly.

Thinking about design now will help you in the long run, and will make this assignment more manageable. Many of the questions have hints about what sort of algorithms you will need to solve the problems, others are more open ended and will require you to put in more thought.

NOTE: this means you *are* allowed to create additional classes, and to submit these with your submission! How you design those classes is up to you – that's part of the challenge of this assignment.

Part 2.5: java.util.ArrayDeque

In order to implement the graph algorithms in this homework, you'll need some notion of a stack and a queue. This is where deques come in.

A **deque** (pronounced *deck*) is a **d**ouble-**e**nded **que**ue. It combines the functionality of queues and stacks. Queues typically only support adding to the back (enqueueing) and deleting from the front (dequeueing). Stacks support adding to the back (pushing) and deleting from the back (popping). A deque allows you to add and remove items to either the front or the back, and thus can function as either a stack or a queue.

Unique to this semester, we are allowing you to import Java's ArrayDeque class (https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html) to use in your graph algorithms. Note that `enqueue` and `dequeue` aren't actual methods of `ArrayDeque` , but there are plenty of methods specified in the API that do the same thing.

Part 3: Maze Solving Revisited (30 points)

Files to submit: `MazeSolver.java` , `MazeSolverTest.java`

It's now halfway through the course, the perfect time for retrospection. Recall the very first homework: HW 0, Part 2 (https://www.cis.upenn.edu/~cis121/current/hw/0-maze-solver/#part-2-recursive-algorithms---maze-solving-30-points), in which we gave you a maze and asked you to find a path to go from a specific source cell to a target cell on the maze. We solved that question using recursion, with an instruction on how to traverse through the maze in some specific order.

Realize that we can represent the maze as an unweighted undirected graph, where the vertices are cells in the maze, and the neighbors for each vertex are its up, down, left, and right adjacent cells, if they exist.

It is easy to see that our recursive approach from HW0 was essentially a depth-first search. However, as we now know, DFS is not guaranteed to return the **shortest path** between two vertices in the graph. This was also the case in HW0, where the path you ultimately found may not have been the shortest path from the given source to the given target cell.

Therefore, instead of using DFS, we'll use BFS to find the shortest path from the source cell to the target cell.

In this problem you will implement the `getShortestPath(int[][] maze, Coordinate src, Coordinate tgt)` method in `MazeSolver.java` , whose input is the $m \times n$ 2D array consisting of 0s and 1s representing the maze, the source, and the target coordinate. The method returns a `List<Coordinate>` of the vertices on the shortest path from `src` to `tgt` . For more details, please refer to the JavaDoc in the stub file.

Your algorithm should run in $O(m \times n)$ time.

Important: The `Coordinate` class functions the same as in HW0, with minor changes. Please make sure you use the one that's included in the stub file for this assignment.

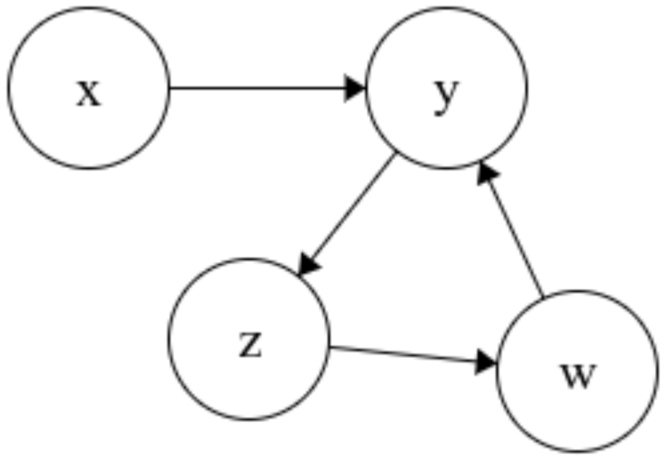
Part 4: Islands and Bridges (30 Points)

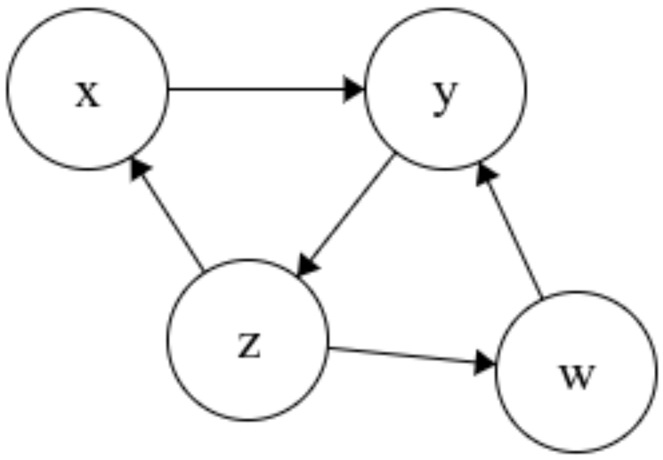
Files to Submit: `IslandBridge.java` , `IslandBridgeTest.java`

Consider a directed graph $G = (V, E)$ where the vertices are islands and the edges are one-way bridges between two islands. For a given island x, we are interested in the following claim: For any island v that can be reached from x through a series of bridges, there also exists a series of bridges to go from v to x.

Design an algorithm that takes in as input a directed graph G and an island x and returns true if the claim holds for x, and false otherwise.

Your algorithm should run in $O(n + m)$ time.





Above are two example graphs. Your algorithm should return False for the left image (since x can reach w, y, and z but none of them can reach x). To contrast, your algorithm should return True for the right image (since x can reach w, y, and z, and this time they can all reach x)

> **Important**: This problem is not the same as checking if the graph is strongly connected! Be sure to draw out some more examples **before** coding an algorithm.

Part 5: Shortest Paths and Dijkstra's Algorithm (30 Points)

Files to Submit: `Dijkstra.java`, `DijkstraTest.java`

In lecture you learned how to solve the single-source shortest path problem for weighted graphs using Dijkstra's algorithm. Specifically, given a weighted directed OR undirected graph $G = (V, E)$ and a source $s \in V$, Dijkstra's algorithm can compute the shortest path from s to every other vertex in expected $O((n + m)\lg n)$ time (assuming you use a binary min heap implementation of the priority queue).

For this part of the assignment, you will use your BinaryMinHeap implementation from Part 1 to implement Dijkstra's algorithm. Your min heap will act as a priority queue, storing each vertex and its distance away from the source s. At each step of the algorithm, you will need to extract from the heap the vertex with the smallest distance from s (this should tell you what the Key and Value should be).

While Dijkstra's algorithm computes the shortest path from s to all vertices in V, your implementation should only compute the shortest path from s to a specific vertex t. See the stub file for more details. Your implementation should run in expected $O((n + m)\lg n)$ time.

Note: You may assume that the weights of all edges will be non-negative! Additionally, your implementation should work on both directed *and* undirected graphs.

Style & Tests

The above parts together are worth a total of 110 points. Style and code coverage are graded on a subtractive basis with deductions as specified in the *Setup and Logistics* section of the writeup, and you will be graded according to the CIS 121 style guide (/~cis121/current/java_style_guide.html). Gradescope will give you your grade on this section immediately upon submission to either autograder.

IMPORTANT: Please **DO NOT** use any external files (.txt, etc.) to write your JUnit tests for this assignment. Our code coverage tool will fail, and you will not be able to submit your code due to failing test cases.

You will need to write comprehensive unit test cases for each of the classes, inner classes, and methods you implement (including helper methods!) in order to ensure correctness. Make sure you consider edge cases and exceptional cases in addition to typical use cases. Use multiple methods instead of cramming a bunch of asserts into a single test method. Your test cases will be auto-graded for code coverage. Be sure to read the testing guide (/~cis121/current/testing_guide.html) for some pointers on what level of testing we expect from you. Finally, due to the Code Coverage tools we use, **please be sure to use JUnit 4** as it is the only version supported.

Note: You will not be able to write JUnit test cases for any private methods. Instead, make them package-private by leaving off any privacy modifier (i.e., do not write `public` or `private`).

This assignment was developed by the CIS 121 Staff.
Last updated on Mon, Nov 2 at 12:43 PM.