CIS 121—Data Structures and Algorithms with Java—Spring 2016

Due: Thursday, March 24, 10:30am online

6 Required Problems (75 points), Qualitative Questions (10 points), and Style and Tests (15 points)

DO NOT modify methods or method headers that were already provided to you.

DO NOT add public or protected methods.

DO introduce helper methods that are **package-private** (i.e., do not have a privacy modifier) as you deem necessary.

Motivation: Network Connectivity

In Homework 7, we considered lossless compression as a way to transmit more data over a fixed Internet link. Now we'll ask questions about the links themselves. The Internet is inherently 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 One: Graph Representation (10 points)

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

Recall the graph lab, where we discussed graph representations. Here, we'll implement an undirected, optionally-weighted graph.

We call the graph "optionally weighted" because it can be used by algorithms that use weights (like Kruskal's) and by algorithms that do not (like BFS). An algorithm like BFS would simply ignore any weights present.

The implementation details are largely up to you, but be sure to consider the runtimes of each operation. In the `Graph.java` stub, we've provided target runtimes for each method you need to implement; be sure to keep these in mind! (Particularly, for anything that says expected or amortized, it is fair game to use a `HashSet` or `HashMap` on if you deem it to be necessary. These data structures offer expected-case $O(1)$ `find`/`insert`/`delete`). Also, make sure that your solution only uses $O(m + n)$ space - please don't use adjacency matrices! Your solution also does not need to handle self-loops or parallel edges, although it may support these anyways.

Note that we've also provided a basic weighted edge class in `Edge.java` with `compareTo`, `equals`, and `hashCode` implementations to make it easy to use for hashing and sorting. This `Edge` class is tangential to the graph representation—it is just a wrapper for passing around edges, just as `Graph.getEdges` does. It will be useful in some later problems. It would probably be helpful for you to read through the `Edge` class to see how it works. Do *NOT* modify `Edge.java`!

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` to ensure that the user cannot modify the data structures you are passing them.

Part Two: Breadth-First Search (15 points)

Files to submit: `BFS.java`, `BFSTest.java`

Recall the discussion of breadth-first search in lecture and in recitation. We want to implement BFS as a single-source shortest path algorithm, which we will use as a tool later in this homework assignment. The

`BFS.java` comments contain necessary clarifying information, and you already know the algorithm, so give it your best shot!

Part Three: Union-Find (15 points)

Files to submit: `UnionFind.java`, `UnionFindTest.java`

Union-find is a data structure for tracking a set of elements partitioned into disjoint subsets. We can `union` these subsets together and then `find` which set an element belongs to. In particular, we can determine when two elements belong to the same set, which ends up being quite useful for Kruskal's algorithm.

We want to start with the pointer-based implementation, in which each element keeps track of a "parent" pointer. At the beginning, everyone is their own parent, and two elements are in the same set if they share an ancestor. This allows for fast `union` operations, only involving changing a parent pointer. Think carefully, though, about what parent pointer that should be.

As with everything in CIS 121, there's a tradeoff at play here. Fast `union` means slow `find` if we're not careful, since `find` requires us to follow the parent pointers up to a root node. If our tree gets very tall, this could take a while. To alleviate this, we'll implement the two optimizations discussed in lecture to help make `find` as fast as possible (and it turns out that that's pretty fast!).

We recommend implementing this without any optimizations at first. Donald Knuth, responsible for popularizing big-Oh notation and inventing TEX, once famously said "Premature optimization is the root of all evil." There are many ways to interpret this piece of wisdom, but here we take premature to mean "before it even works at all." It's also much easier to optimize working code when you have test cases to make sure you didn't break anything.

The two optimizations are described in the following sections.

Weighting

When we union two sets, one set essentially absorbs the other; the absorbing set continues to exist, but it's the end of the road for the absorbed set. How, then, should we decide which set does the absorbing, and which set gets absorbed?

You'll need to start keeping track of some property of your sets, and use that information to make a decision here. Remember that we're trying to make find as fast as possible, so we want to keep our tree heights low; take this into consideration when deciding what property to track.

If you implement this optimization correctly, `find()` and `union()` will run in $O(\log n)$ time.

Path Compression

Another technique we can use to keep our traversals back to the root node from getting too long is by making them take just one hop. We call this path compression; whenever we traverse the tree to perform a `find` operation, we can just point the nodes along the traversal to the root. This should only take one or two lines of code to implement!

The amortized running time of `find()` and `union()` is very hard to reason about (take CIS 502 if you would like to grapple with that proof, or look here). Unbelievably, the path compression optimization makes the running time of those operations "effectively constant." In reality, they are $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function. $\alpha(n)$ grows so slowly that for almost any practical value of n, $\alpha(n) < 5$. Hence, "effectively constant"! That's pretty cool.

Part Four: Kruskal's Algorithm (10 points)

Files to submit: `Kruskal.java`, `KruskalTest.java`

Kruskal's algorithm computes the minimum spanning tree of a graph by using the union-find data structure. You are to implement this algorithm, which is further described by the comments in `Kruskal.java`.

Part Five: Widest Path Algorithm (15 points)

Files to submit: `WidestPath.java`, `WidestPathTest.java`

Consider a graph and a path P. The path P is a *widest path* when the weight of smallest edge in P is maximal. This maximal minimum-weight edge is known as a *bottleneck edge*. In other words, if you wanted to transfer a large amount of something (internet packets, car traffic, etc.) along a graph, and you considered edge weights to be capacities, you would want to find a path with the largest overall capacity. If we think about Internet packets, then the overall capacity of a route would be limited by the lowest-bandwidth connection in that route, as we would not be able to send more information along that path. Thus, the widest path is the path along which we can send the most stuff. The constraining edge on that path, or in this example, the lowest-bandwidth connection, would be the bottleneck edge of the path.

How can we algorithmically find a graph's widest path? That's up to you to devise and implement. There are multiple ways to solve this problem, so consider how you can do it with what you've already built (namely, BFS and Kruskal's). Please do this without modifying the behavior of the BFS and Kruskal's implementations you have already built.

Part Six: Client Application (10 points)

Files to submit: `Client.java`, `ClientTest.java`

Let us take a step back. We have spent a lot of time in this class implementing APIs that do cool things. In this homework, we will briefly use an API we have developed as a code client.

A brief aside: **API** stands for **Application Program Interface**. You've probably heard the initialism used before. In the context of the web, it often refers to a set of URLs and requests that one can use to interact with a third-party service; for example, Candy Crush integrates with the Facebook API. More generally, it refers to the points of contact between any two independent pieces of a software system. In our context, this means the public methods in a Java class or interface that you are allowed to call. For example, the `WidestPath.java` API is simply `WidestPath.getWidestPath()`.

Now, imagine that you work for an Internet service provider (ISP). Imagine also that you have a list of all direct links between routers on your company's network, and you know the bandwidth capacity of each link. As an ISP, you're concerned about effectively routing Internet traffic.

Next, consider strategies one might use in choosing a path between two routers. Some common metrics include geographic distance (weighted shortest path) and number of hops (unweighted shortest path). Your company, however, wants to consider widest paths between routers in order to maximize available bandwidth. You have the `WidestPath.java` API available, so you do not need to implement a solution; instead, you need to marshall your own data into the format of the API and convert the result of the API call into a usable answer. Figuring out how to do this is the challenge of part six.

Part Seven: Conceptual Questions (10 points)

Files to submit: `questions.pdf`

Please answer the following questions in a LaTeX'd document called `questions.pdf`:

1. What is the running time of your implementation of Kruskal's algorithm? Was there any part of the code that acted as a bottleneck? If so, how could that be avoided?

2. What is the running time of your implementation of Widest Path? Can you think of any alternative ways to solve the Widest Path problem?

3. What were some of the advantages and disadvantages of the `Graph` class using `int`s to represent vertices? Consider this question in the context of `Client.java`.

4. Where possible, we used a functional structure in this homework, encouraging you to avoid statefulness. For example, the `BFS.getShortestPath()` method is static and takes in the graph as an argument, *à la* OCaml. Discuss some pros and cons of this kind of structuring.

Style & Tests (15 points)

The above parts together are worth a total of 85 points. The remaining 15 points are awarded for code style, documentation, and sensible tests. Style is worth 5 of the 15 points, and you will be graded according to the 121 style guide.

You will need to write comprehensive test cases for each of the methods you implement. This assignment in particular is highly compositional, with later parts building heavily on earlier parts, so we'd like to stress the importance of doing *bottom-up testing for each component before using it elsewhere*. We also encourage you to work out small examples of BFS, Kruskal's, and Widest Path by hand in order to write unit tests.

Make sure you consider edge cases, i.e., the more "interesting" inputs and situations. Use multiple methods instead of cramming a bunch of asserts into a single test method. Be sure to demonstrate that you considered "bad" inputs such as null inputs. Be sure to also demonstrate that you've tested for inputs to methods that should throw exceptions! Your test cases will be manually graded by the TAs, and are worth 10 of the 15 points. You will have to thoroughly test your code to get full points! This includes testing any helper methods you have written. **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`).