Project 4
Graph

Out: Saturday, April 11
In: Wednesday, April 29, 11:59 pm EST

1 Prologue

1. To install, type `cs0160_install graph` into a shell. The script will create the appropriate directories and deposit stencil files into them.
2. To compile your code, type `make` in your project directory. To run your code and launch the visualizer, type `make run` in the same directory. Make sure that your `Makefile` is in the same directory as your code.
3. To run tests, run `make run_tests` from your project directory.
4. To hand in your project, go to the directory you wish to hand in, and type `cs0160_handin graph` into a shell
5. To run a demo of this project, type `cs0160_runDemo graph` into a shell.
6. The documentation for support classes and NDS4 data structures can be found here: `http://cs.brown.edu/courses/csci0160/static/files/docs/doc/graph/index.html`. This is also linked off of the class website.
7. Remember to not include any identifying information in your hand in.

2 Introduction

In this assignment, you will implement a graph using an adjacency matrix as its underlying data structure. You will also implement the Prim-Jarnik algorithm for finding minimum spanning forests (MSFs) and the PageRank algorithm.

3 Using Eclipse

Follow the instructions located here to set up locally.

If you would like to use Eclipse, you may certainly do so. In order to set up your project and make Eclipse work with the support code, you should follow the website linked above. We've provided a few extra instructions below:

- **Please note that we have updated our external jars for this project.** If you are working on a department machine or over Fuse, you simply need to restart Eclipse to get the most updated jars. If you are working locally and have manually copied over our jars using SCP or another file-transfer system, you will need to re-copy and configure your Eclipse with the external jars. You will need the following jars:
  - Under the "libraries" tab choose "Add External JARs..."
  - Select `cs0160.jar`
  - Select `nds4.jar`
  - Select `junit-4.12.jar`
  - Select `hamcrest-core-1.3.jar`
  - Click "Finish"
- Right-click on App.java and select `Run As` → `Java Application`. Now you can run your program by pressing the green "play" button at the top of your screen and selecting "Java application" if prompted
- To run the tests in Eclipse, you can right-click on `TestRunner.java` and click `Run As` → `Java Application`.
- Alternatively, if you want to run a test file, you can right-click on that file and select `Run As` → `Junit test`
- To configure your Eclipse projects to run over FastX or SSH, follow these setup steps.
  - Right click on the package icon next to the project name. Go to properties.
  - Go to Run/Debug Settings, select the main window App and click Edit.
  - Go to the arguments tab and, and enter `-Dprism.order=sw` in the VM arguments block
  - Hit Apply and OK
  - You should be all set to work on this project remotely with Eclipse. Make sure to do this for each new project.

4 Working from Home

Follow the instructions located here to set up locally.

If you wish to work locally, you should be able to set up the project in Eclipse on your home computer by following the directions that are listed on the website linked above. Note that you will need to have the support libraries listed in that section copied onto your local computer in order to reference them – the link above has instructions on how to do this!

As always, be sure to test your code on department machines before handing it in!

5 Overview of Your Tasks

We have provided stencil code for the following classes: `AdjacencyMatrixGraph`, `MyPrimJarnik`, `MyPageRank`, and `MyDecorator`. You need to fill in these stencils. Feel free to add any helper functions you think are necessary as well. We will provide a brief overview and specific information about these classes and their methods in Section 8 of this handout.

1. `AdjacencyMatrixGraph`: You will implement your graph here. Its underlying data structure will be an adjacency-matrix.
2. `MyPrimJarnik`: Here, you will implement the Prim-Jarnik algorithm. You will use decorations to mark vertices with specific information, and should return a collection of edges that make up your MSF.
3. `MyPageRank`: Here, you will implement the PageRank algorithm. You should return a Map mapping each Vertex to its rank.
4. `MyDecorator`: Here, you will define methods that allow you to "decorate" vertices of your graph with specific information.

6 Reading

Refer to the slides from lecture and the help session to recall how Prim-Jarnik and PageRank work. Chapter 5.1 of Dasgupta et al. and Volume 3 of Roughgarden also discuss this.

7 Visualizer

7.1 Using the Visualizer

You must implement vertices, insertVertex, edges, insertEdge, and endVertices in AdjacencyMatrix before the visualizer will work. For the most part, using the visualizer is intuitive: to create a vertex, left-click. To create two vertices with an edge between them, click and drag, landing either on another pre-existing vertex or on nothing to create a new one. Here are some less intuitive controls:

- **select a vertex `v` as `v1`**: left-click on the vertex
- **select a vertex `v` as `v2`**: shift + click on the vertex (or option-click for some computers).
- **remove a vertex and any incident edges:** right-click on the vertex
- **select an edge:** left-click on the edge
- **remove an edge:** right-click on the edge

- **directedness:** Prim-Jarnik only works on an undirected graph, while PageRank only works on a directed graph.
  - To account for this, the visualizer has a "toggle directedness" option. The visualizer will always begin *undirected*, so make sure to toggle it to *directed* before testing MyPageRank!

A note on adding edges: You can choose to add edges with a random weight, with the distance between two vertices as the weight, or with custom weights that you give as input to the visualizer.

The visualizer should run without any issues. If you are getting a RuntimeException, this is an issue with your implementation.

- Your vertices may shift slightly after clicking PageRank. This is expected! You will not be deducted for this shifting.
- Don't worry if vertices expand to outside the white area representing the canvas of the visualizer.
- Your visualizer should behave like the one in the demo. If you are wondering about how something should behave, reference the visualizer.

7.2 Visualizer Methods

The following methods of AdjacencyMatrixGraph correspond to functionality in the visualizer:

- `areAdjacent(v1, v2)`: returns a boolean value describing whether there is an edge starting on `v1` and ending on `v2`. This is the same for both directed and undirected graphs.
- `connectingEdge(v1, v2)`: if the two selected vertices `v1` and `v2` are connected by an edge, returns the edge. If the graph is directed, it should only return the edge if there is an edge starting on `v1` and ending on `v2`, or return null. Keep in mind that this is different from `areAdjacent(v1, v2)`!
- `endVertices(e)`: returns the two endpoints of an edge `e`. The order in which the vertices are returned does not matter, regardless of whether the graph is directed or undirected.
- `opposite(v1, e)`: returns the second endpoint of an edge `e` (that is: if you select a vertex and a connected edge - this will return the *other* vertex attached to that edge). This method should be the same whether your graph is directed or undirected.

- `incomingEdges(v1)`: for an undirected graph, returns all edges attached to the selected vertex `v1`. For a directed graph, it should return all edges that end on this vertex.
- `outgoingEdges(v1)`: for an undirected graph, returns all edges attached to the selected vertex `v1`. For a directed graph, it should return all edges that start on this vertex.
- `toggleDirected()`: should set the graph to be undirected if it is currently directed, or directed if it is currently undirected. Your graph automatically starts undirected. When you toggle the graph's directedness, any existing vertices and edges are automatically cleared.

Note that in order for the visualizer to work at all, you need to implement the following methods: `vertices()`, `insertVertex()`, `edges()`, `insertEdge()`, and `endVertices()`.

7.3 Debugging PageRank with the Visualizer

- The visualizer will represent the vertices with respect to their relative ranks.
  - You can generally see which vertices should grow or shrink depending on how many incoming and outgoing edges they have.
  - It's easier to visualize the expected results on smaller graphs!
- The Olympics graph is larger and more complicated than hand-drawn graphs so it is a useful debugging tool. Make sure to run your PageRank on this graph before handing in. It is also a great way to visualize how the PageRank algorithm works in reality!
  - To run Pagerank on the Olympics graph (and all other provided graphs), you will have to load in the graphs using the "Load Graph" button. The Olympics graph is called `olympics_links.txt`, and is provided for you in the stencil code.
  - When loading in the Olympics graph, make sure you're set in directed mode! Otherwise, loading in the Olympic graph as an undirected graph will give you a graph with all edge weights of 1.
- As always, refer to the demo if you have any questions about how your PageRank is performing. If you enter the same graph into your program and the demo, the results should be very similar.

8 Your Code

8.1 Stencil

The stencil contains descriptions of the methods you'll write, their run-time requirements, parameters, and return values. **Unless otherwise noted, all functions in the support code are O(1)**.

8.2 AdjacencyMatrixGraph Class

8.2.1 Description

Your AdjacencyMatrixGraph should be able to represent either a directed or an undirected graph. The underlying data structure for your graph will be an adjacency matrix. An adjacency matrix is a 2D array. The i^{th} column and i^{th} row of this array represent the i^{th} vertex in the graph. Depending on if the graph is directed or undirected, the way edges are stored within the adjacency matrix will vary.

In an undirected graph, if vertex 1 and 5 are connected by an edge, then entries (1, 5) and (5, 1) in the adjacency matrix contain the `edge` that connects vertices i and j. Otherwise they are `null`. Note that every edge is in two places in the array. For a directed graph, each edge will only be stored once in the array. See the examples below.

Suppose we have an **undirected** graph containing vertices `v1, v2,` and `v3`, and that edge `e1` connects `v1` and `v2`, and that edge `e2` connects vertices `v1` and `v3`. Here's the corresponding adjacency matrix:

	v1	v2	v3
v1	null	e1	e2
v2	e1	null	null
v3	e2	null	null

Now suppose we have a **directed** graph containing vertices `v1, v2,` and `v3`, and that there is an edge `e1` pointing from `v1` to `v3`, and an edge `e2` pointing from `v3` to `v2`. Here's the corresponding adjacency matrix:

	v1	v2	v3
v1	null	null	e1
v2	null	null	null
v3	null	e2	null

Note that the adjacency matrix should be able to toggle between representing a directed and undirected graph. Keep in mind that there should not be separate methods to handle directed and undirected graphs, but that you should be able to handle both implementations within the same methods.

8.2.2 Implementing the Adjacency Matrix

Since we are using an adjacency matrix as our graph implementation, each vertex must have a "number," so that it can represent an index of a row and column in the array. This assignment is not as trivial as it may appear. Arrays have a fixed size, so you cannot indefinitely increase the number for each new vertex because you will exceed the size of your array. Note that the number associated with a given vertex must be unique - it must not be associated with any other vertex.

Your array should be able to hold up to `MAX_VERTICES` vertices, which is a constant defined in the support code.

8.3 Vertex and Edge Attributes

You will need to keep track of some information about the vertices and edges of your graph. To that end, the support code classes `GraphVertex` and `GraphEdge` have some methods that you may find helpful. View the Javadocs.

Note: In the support code, you'll notice there are references to `CS16Vertex` and `GraphVertex`. `CS16Vertex` is an interface that the class `GraphVertex` implements. You should be declaring vertices of type `CS16Vertex` but instantiating new instances of `GraphVertex`.

8.4 MyPrimJarnik Class

This class implements a slightly modified version of the Prim-Jarnik MST algorithm. The Prim-Jarnik algorithm has been extended to calculate an MSF (the collection of MSTs of each connected subgraph of your graph). This algorithm is required to run in $O(|V|^2 \log |V|)$ time.[1] Use `CS16AdaptableHeapPriorityQueue` (don't use the NDS4 one).

Here's an outline of the algorithm (which is presented in more depth in the reading – see Section 6).

1. First, decorate each vertex with a key: you can think of the key as the "cost" of adding a given vertex to the MSF. At each iteration of the algorithm, you will want to add the edge that connects to the cheapest vertex to the MSF. Initially, each node starts with a value of "infinity." (You can use the `Integer.MAX_VALUE` constant to represent infinity.)

2. Insert the vertices into a priority queue, using the keys that were assigned in the previous step.

[1] It *can* be made more efficient— it would run with complexity $O((|V| + |E|) \log |V|)$ if we were to use a binary heap and an adjacency list and with complexity $O(|E| + |V| \log |V|)$ if we were to use a Fibonacci heap and an adjacency list. Our implementation uses a binary heap and an adjacency *matrix*, which gives us runtime $O(|V|^2 \log |V|)$.

3. Remove the minimum vertex (`v`) from the priority queue, and add the edge that most recently updated it (if any) to the MSF.
4. Examine all (if any) of `v`'s incident edges `e` whose opposite vertex `u` remains unvisited. If `u` has a key greater than the weight of `e`, update the key to become the weight of `e` (and make sure this change is reflected in the priority queue—to do this, you'll need to know the `Entry` in the PQ associated with each vertex). Keep track of which edge most recently updated the key of vertex `u` (you may find a decoration useful here).
5. Continue to remove vertices from the priority queue (adding edges to the MSF and updating other vertices as you go along, as described in steps 3 and 4) until the PQ is empty.

Note that in addition to decorating edges as you go along, you must ultimately return a collection of the edges that are in the final MSF. You may add edges to the collection as you go along, or you may complete the entire algorithm and then check to see which edges you should add to the collection. Feel free to use any sensible implementation of the `java.util.Collection` interface.

8.5 MyDecorator Class

The `CS16Decorator` interface represents "decorations" that you will use to label vertices and edges with specific information. The `MyDecorator` class is the implementation of this interface that you will write and use. You may want to use this class to note that a specific edge is part of your minimum spanning forest. Think about what type of data structure you want to use in this class; it should be a data structure that you can quickly use (i.e., constant-time association and lookup) to associate edges and vertices with some type of information.

8.6 MyPageRank Class

This class implements the PageRank algorithm. The algorithm takes in a directed `Graph<V>`, where each vertex represents a page and each directed edge represents a link from one page to another. The algorithm outputs a mapping from each vertex to its pagerank. You can think of a page's *pagerank* as some amount of fluid which represents the importance of that page. If a page has many incoming links then that page will have a higher amount of pagerank.

Even though the algorithm takes in anything that implements the `Graph<V>` interface, we will only be testing `MyPageRank` on the `MyAdjacencyMatrix` class. The algorithm will output each page's pagerank. Remember that the sum of all the pageranks should equal 1. Also, note that the pagerank of a single page should be between 0 and 1 (inclusive). Refer to the slides from lecture and from the help session to recall how the algorithm works.

Here is an outline of how you should implement the algorithm:

1. The algorithm relies on two important constants which must be stored and used throughout your implementation. These are set for you and should not be changed:

   (a) `_error`: the algorithm needs to stop running when either of the following conditions hold (whichever one comes first): (1) for each vertex v, the *absolute value* of the difference between v's pagerank in the previous round and v's pagerank in the current round is less than or equal to `_error`; or (2) the algorithm has run for 100 iterations.

   (b) `_dampingFactor`: is a constant that the algorithm uses to make sure that pages don't pass the entirety of their pagerank to their neighbors. Without this, for certain graphs, all of the pagerank can accumulate in certain parts of the graph. Also, pages that are disconnected from other pages would never get any pagerank. For this project, the damping factor is set to 0.85.

2. Store the vertices of the graph in an ArrayList so you can access them easily by index. You can then use a second array to store, for each vertex, its number of outgoing edges. Finally, you can use two more arrays: one that stores the amount of pagerank of each vertex from the previous round and one that will store the amount of pagerank of each vertex for the current round. **Hint:** It will be very helpful if the indexing among these arrays is consistent; that is, the information you need for a page (e.g., its vertex, its number of outgoing edges or its previous or current pageranks) is stored at the same index across all these arrays. For example, if a vertex v is stored at index 0 in your vertices array, then v's number of outgoing edges would be stored at index 0 in your outgoing edges array.

3. Initialize every vertex's pagerank to be $1/N$, where N is the total number of vertices in the graph.

4. The PageRank algorithm runs in rounds and, as mentioned above, your implementation should run until the difference in pageranks between the current round and the previous round is less than or equal to `_error` or until it has executed 100 iterations. In each round your implementation should:

   (a) Decide how to handle sinks. See the lecture slides for a refresher on why sinks cause problems and different ways to address them. See the help session slides for two possible approaches.

   (b) Depending on your approach, update the ranks produced from the previous iteration by accounting for pages that are sinks (pages with no outgoing edges). Look at the help slides for pseudocode.

   (c) Use the pagerank values from the previous iteration of the algorithm to compute the current pagerank values of every vertex and store them in your current pagerank array. **Hint:** remember how *directed* edges are stored in an adjacency matrix: if there is an edge from vertex 3 to vertex 2, then there is an entry at

location $(3, 2)$ in the adjacency matrix (i.e., row 3 and column 2), but *not* at location $(2, 3)$. Make sure to keep this in mind when you compute the updated pagerank of a vertex.

9 Conceptual Question

There may be cases where, due to ethical considerations, we do not want to display the 'true' rank of all webpages and instead want to alter the search results to prevent the spread of false information. Suppose that, when someone searches 'global warming', we want to ensure that a website that claims global warming to be a hoax has the lowest rank of any page. (*You can see more examples of this by loading other graphs; we give you the following three graphs in the stencil:* `flatearth_links.txt`, `global_warming_links.txt`, `vaccines_links.txt`, *which represent the webpage results when a user searches, "Is the earth flat?," "Is global warming real?," and "Do vaccines cause autism?," respectively*). However, we do not want to censor the page (i.e. delete the vertex from our graph). How could you go about ensuring a page has the lowest rank of any vertex without simply deleting it from the graph? Please outline your strategy and describe an algorithm that could do this. We are looking for an algorithm that impacts how pagerank looks at a vertex; as such, manually setting the final rank of a blacklisted site to be 0 at the end of the algorithm is not a valid solution (it would also make your ranks not add up to 1!). Please write your answer to this question in your README.

10 Written Questions

In a separate document, please answer these questions thoroughly. Each answer should contain 3-4 sentences and will be graded on thoughtfulness. Note that a `graphReflection` folder should have been created when you ran the install script for `graph`.

Submit your file as a PDF by changing into the `graphReflection` directory and using the following handin script: `cs0160_handin graphReflection`

1. Aside from the three searches given in the provided text files (regarding vaccines, global warming, and the earth being flat), what are some other searches where it may be beneficial to modify the search results? Please give at least 2 examples and explain why the results should be altered.

2. What are the ethical implications of filtering search results? What are the benefits and the risks/costs? Do the pros outweigh the cons?

3. Who currently has the ability to alter the search results? Who should be responsible for it and why?

11 Extra Credit

Please only attempt the extra credit when you have a fully-functional page rank algorithm. In `MyPageRank.java`, implement the algorithm you describe in the conceptual question. We have provided a list of all of the websites from the three search results that we provided that have been flagged for providing misinformation that you can access by calling `PageRank.blacklist` in `MyPageRank.java`.

Please implement this in a helper method called `removeBlacklist`. If you implement the algorithm in your `calcPageRank` without a helper method, you will receive no credit.

12 Testing

As usual, we require a well thought-out set of tests. We won't give you too many hints – you've come a long way in CS16, so you're a pro at this by now! You will write these test functions in the stencil files provided: `GraphTest.java`, `MsfTest.java`, and `MyPageRankTest.java`.

In `MyPageRankTest.java`, we have provided two sample tests. The results of your PageRank algorithm should be within 0.03 of the values specified in the sample tests. In addition to these sample tests, you should write your own tests. While you don't know the exact pageranks of different pages in different graphs, **you should still test relative pageranks** (i.e. from the demo, you know that the pagerank of page X is greater/less than the pagerank of page Y, so you should check to make sure this holds true for the values returned by your own algorithm).

As always, please comment each testing function extensively. If there's anything particularly notable about your tests, include an explanation in your README.

13 What to Hand In

1. Code for the four classes, `AdjacencyMatrixGraph`, `MyPageRank`, `MyPrimJarnik`, and `MyDecorator`.
2. Code for the three testing classes, `GraphTest`, `MsfTest`, and `MyPageRankTest`.
3. A README pointing out any bugs in your code, any significant design choices (talk about the types of decorations you used, and how/why you used them), descriptions of your test functions and your answer to the conceptual question. For more information, see the README guide on the course website.
4. A PDF file containing your answers to the written questions.