Programming Assignment #2

Testing Due: Sunday, Mar 15 @ 11:59PM
Implementation Due: Sunday, Mar 29 @ 11:59PM
Assignment Link: https://classroom.github.com/a/gL7HLTx8

Objectives

1. Work with different data structures for storing graphs
   a. Generate an adjacency list for a graph that is stored as an edge list in order to make traversal more efficient
2. Implement algorithms for efficiently traversing graphs
   a. Search the graph to find a path connecting two intersections that minimizes the number of intersections traveled through
   b. Search the graph to find a path connecting two intersections that minimizes distance traveled
   c. Search the graph to find all points reachable within a certain distance from a given starting point

Useful Links

- The Java API
  - `Comparable`
  - `Comparator`
  - `HashMap`
  - `HashSet`
  - `PriorityQueue`
- JUnit Documentation
  - Assertions

Submission Process, Late Policy and Grading

Testing Due: Sunday, Mar 15 @ 11:59PM
Implementation Due: Sunday, Mar 29 @ 11:59PM
Total Points: 30 (5 for testing + 20 for correctness + 5 for runtime)

The project grade is the grade assigned to the latest (most recent) submission made to Autolab (or 0 if no submissions are made). Autolab will pull your submission from your GitHub repository, so you must make sure that any changes you want to be included in your grade have been committed and pushed. The AutoLab submission target **for the implementation phase** will remain open 2 days after the stated deadline.

- If your final submission is made before the deadline, you will be awarded 100% of the points your project earns.
- If your final submission is made up to 24 hours after the deadline, you will be awarded 75% of the points your project earns.
- If your final submission is more than 24 hours after the deadline, but within 48 hours of the deadline, you will be awarded 50% of the points your project earns.
- No submissions will be accepted more than 48 hours after the deadline.

You will have the ability to use three grace days throughout the semester, and at most two per assignment (since submissions are not accepted after two days). Using a grace day will negate the 25% penalty per day, but will not allow you to submit more than two days late. Please plan accordingly. You will not be able to recover a grace day if you decide to work late and your score is not higher. Grace days are automatically applied to the first instances of late submissions, and are non-refundable.

For example, if an assignment is due on a Sunday and you make a submission on Monday, you will automatically use a grace day, regardless of whether you perform better or not. You may then submit as many times as you want on Monday without penalty.

Be sure to test your code before submitting, especially with late submissions in order to avoid wasting grace days. Code that fails to compile and run on your machine will also fail to compile and run on AutoLab. You should not rely on feedback from AutoLab for debugging, but rather the feedback you get from your own tests.

Note: No late submissions will be accepted for the testing portion of the assignment, and no grace days can be used on the testing portion of the assignment.

Setup

In order to complete this project, you must have completed PA0. If you are working on a machine other than the one you used in PA0, you must at least complete steps 2 and 4 in order to get IntelliJ and GitHub working properly.

Once you have ensured your development environment is set up as in PA0, you can accept the PA2 assignment in GitHub Classroom (here), and create a new IntelliJ project from VCS with your newly created repository.

Instructions

In this assignment you will implement a set of tools for computing directions between intersections in downtown Buffalo. Specifically, you will implement methods in the **`MapUtils`** class, which includes the construction of an adjacency list, and three different implementations of graph traversal.

Note: You must **NOT** modify any files other than **`MapUtils.java`** and **`MapUtilsTests.java`** to complete the assignment.

As with PA1, the first phase of PA2 will be to write tests which must be able to pass on correct implementations of the **`MapUtils`** functions, and fail broken implementations. The second phase will be to implement the **`MapUtils`** functions yourself.

After you complete your tests, make sure to commit and push your work to GitHub, and submit to the PA2 Testing submission in Autolab. After completing your implementation, make sure to commit and push your work to GitHub, and submit to the PA2 Implementation submission in Autolab.

Hint: It is advised that you commit and push frequently rather than waiting until you've completed everything.

Hint: Although you will get feedback from Autolab about correctness of your solutions, you should be testing locally, and adding test cases as needed. This will be a more effective/efficient means of development, and will also give you a better understanding of the content of this programming assignment in the process.

Programming Tasks

Testing Phase

Modify the file **`MapUtilsTests.java`** to include new test cases.

Your test cases will first be run against a correct implementation of **`MapUtils`**. If your tests fail the correct implementation you will receive 0 points for the testing phase.

Your test cases will then be run against several broken implementations of **`MapUtils`**. You will get points for each broken implementation that at least one of your tests fail.

Implement the `MapUtils` Functions

Implement the functions in **`MapUtils`** according to the following specifications.

```
public static Map<String, List<Edge>> computeOutgoingEdges(StreetGraph graph)
```

The output of this function should be a **`Map`** containing an entry for every **`Intersection`** identifier that appears in the **`from`** field of at least one **`Edge`** in the graphs **`edges`** field.

- The key should be the identifier of the intersection.
- The value should be a **`List`** of every **`Edge`** whose **`from`** field is equal to the key.

For a graph with m edges, this function should run in $O(m)$ time.

```
public static Optional<List<Edge>> pathWithFewestIntersections(
    StreetGraph graph,
    Map<String, List<Edge>> outgoingEdges,
    String from,
    String to)
```

This function should return a path between the intersection with identifier **`from`**, to the intersection with identifier **`to`**, with the fewest number of edges possible. If there is no valid path then **`Optional.empty()`** should be returned.

The **`outgoingEdges`** argument passed will be the adjacency list computed by calling **`computeOutgoingEdges`** function on **`graph`**.

For a graph with m edges and n vertices, this function should run in $O(m)$ time.

```
public static Optional<List<Edge>> pathWithShortestDistance(
    StreetGraph graph,
    Map<String, List<Edge>> outgoingEdges,
    String from,
    String to)
```

This function should return a path between the intersection with identifier **`from`**, to the intersection with identifier **`to`**. The returned path should contain edges whose distance sums up to the smallest distance possible. If there is no valid path then **`Optional.empty()`** should be returned. The **`Edge.getDistance()`** method should be used to calculate the length of an edge.

The **`outgoingEdges`** argument passed will be the adjacency list computed by calling **`computeOutgoingEdges`** function on **`graph`**.

For a graph with m edges and n vertices, this function should run in $O(m \log m)$.

```
public static List<Intersection> intersectionsWithinRange(
    StreetGraph graph,
    Map<String, List<Edge>> outgoingEdges,
    String intersection,
    Double distance)
```

This function should return a list of ALL intersections that can be reached from the intersection with identifier **`intersection`** by traveling along paths whose edge weights sum up to at most **`distance`** km. The **`Edge.getDistance()`** method should be used to calculate the length of an edge.

The **`outgoingEdges`** argument passed will be the adjacency list computed by calling **`computeOutgoingEdges`** function on **`graph`**.

For an arbitrary graph with m edges and n vertices, if N of those intersections can be reached from **`intersection`** in at most **`distance`**, and M is the total number of edges incident to at least one of those N vertices, this function should run in $O(M \log(M))$ time.

Additional Notes

StreetGraph

Input to most operations in this assignment will be passed via an instance of the **`StreetGraph`** class. There are two useful instance fields:

1. **`graph.intersections`**: A collection of street intersections, organized by the intersections identifier. An **`Intersection`** consists of an identifier, and a pair of coordinates. These coordinates will either be interpretted as geospatial coordinates (longitude, latitude), or as Cartesian coordinates (x,y) depending on the type of **`Edge`** that is used. If you need to determine the geospatial or cartesian distance (both in km) between **`Intersections a`** and **`b`** you can call **`a.geoDistanceTo(b)`** or **`a.cartesianDistanceTo(b)`** respectively.
2. **`graph.edges`**: A collection of **directed** edges between intersections (i.e., street segments). An **`Edge`** object contains two intersection identifiers (see **`graph.intersections`**) for the **`from`** and **`to`** intersections it connects, as well as the name of the street. Two-way streets are represented as two individual edges, one for each direction.

An example data file (as passed to **`StreetGraph.load`**) can be downloaded from the course website (here). Right click on the link and click Save As to download. See **`Main.java`** for how to load the graph into your program.

Place the downloaded file in the data directory of your project. This file follows the standard OpenStreetMap XML format. Feel free to download your own examples from OSM, or create your own.

DO NOT COMMIT ANY DATASETS TO YOUR PROJECT, THEY ARE QUITE LARGE

Notice that **`StreetGraph`** is an implementation of the **`EdgeList`** data structure discussed in class. The first function in **`MapUtils`** that needs implementation generates an adjacency list for **`StreetGraph`**. Rather than storing the adjacency lists for each vertex in the vertex themselves, it creates a **`Map`** that associates vertices to lists of edges. As long as our **`Map`** implementation provides $O(1)$ lookups this accomplishes the thing as storing the edges in the vertex objects directly by allowing us quick access to the outgoing edges for a particular vertex.

If you are unsure of what the difference is between an edge list and adjacency list implementation or how it affects the search process please review the relevant lecture material.

Graph Traversals

Recall that we discussed several forms of graph traversal in class, using different data structures to control the order in which newly visited vertices are explored. Specifically, we considered:

- **Depth First Search (DFS):** Explores in LIFO order by using a **`Stack`** (the call stack) to manage vertices to explore
- **Breadth First Search (BFS):** Explores in a FIFO order by using a **`Queue`** to manage vertices to explore
- **Dijkstra's Algorithm:** Explores vertices in priority order, where vertices closer to the starting vertex have a higher priority. A **`PriorityQueue`** can be used to enforce this ordering on vertices to explore.

One of the properties of BFS is that when we first visit a vertex, we know we are visiting it via the shortest path from the origin in terms of number of edges. Similarly, if we are using Dijkstra's algorithm the first time we visit a vertex we know we are visiting it via the shortest path from the origin in terms of total distance.

Note: When we can consider a vertex as **visited** varies slightly when doing BFS vs Dijkstra's:

- With BFS we can mark a vertex as visited as soon as we enqueue it into our work list, since we know that at that point in the search there are no other shorter paths to that vertex in terms of number of edges.
- For Dijkstra's however, there could be a shorter path in terms of total distance that we have not discovered yet, so we can only consider a vertex as **visited** when we dequeue it from our **`PriorityQueue`**, since at that point we know it is the closest vertex to the origin that we have not yet visited.

`HashMap` and `HashSet` in Java

In the versions of the traversal algorithms discussed in class, we stored information about the state of the traversal directly in the **`Graph`** itself. We cannot (by design) do that in this programming assignment, so you will need to find another way to track what vertices and/or edges have been visited during the traversal.

The **`HashMap`** and **`HashSet`** classes are implementations of the **`Map`** and **`Set`** ADT that use a data structure called a **`HashTable`**. We will discuss and analyze hash tables later in the semester, but below are some operations you may find useful for this programming assignment along with their runtimes.

A **`HashSet`** is an implementation of the **`Set`** ADT that allows for efficient insertion, lookup, and removal. Some relevant methods are:

```
// Adds the specified element to this set if not already present
public void add(E e)

// Returns true if the set contains the specified element
public boolean contains(Object o)

// Removes the specified element if it is present
public boolean remove(Object o)
```

A **`HashMap`** is an associative collection that stores key-value pairs, and allows efficient lookup/updates by key. Some relevant methods are:

```
// Associates the specified value with the specified key in this map
public V put(K key, V value)

// Returns true if this map contains a mapping for the specified key
public boolean containsKey(Object key)

// Returns the value to which the specified key is mapped, or null if no mapping exists
public V get(K key)
```

In this programming assignment you may assume the above operations run in $O(1)$ time. (They actually run in *expected* $O(1)$ time...but we will cover this in more detail later in the semester)

`PriorityQueue` in Java

Java provides an implementation of the `PriorityQueue` ADT based on a binary heap. **`PriorityQueue`** supports the standard ADT operations, including **`add`**, **`poll`**, **`peek`** and **`size`**.

When you create a **`PriorityQueue`** in Java, the elements will be stored using their natural ordering. For user-defined classes, you can define their natural ordering by having the class implement the `Comparable` interface. If you would like to use an ordering other than the natural ordering, you can supply the **`PriorityQueue`** with a `Comparator` object at instantiation.

Hint: In order to utilize a **`PriorityQueue`** for Dijkstra's algorithm, you may need to define a custom element type based on what you would like to store in the **`PriorityQueue`**. You can make a **`static private class`** within **`MapUtils`** for this purpose. If you do so, you will have to tell your **`PriorityQueue`** how to order your objects as described above. To do so, you can make your private class implement the **`Comparable`** interface (recommended) **OR** create a custom **`Comparator`** object.

Other Tips

- Before you start writing tests draw out a number of example graphs. For each graph, think about what the different functions being implemented in this assignment should return under various circumstances. Then start asking "what if" questions that might lead to interesting situations:
  - "What if I add another edge here that creates a cycle?"
  - "What if there is not a path between these two nodes?"
  - "What if there are multiple paths between these two nodes (with certain properties)?"
  - etc.

  These questions might lead you to a few additional test cases or modifications to your example graphs you had not considered.
- When implementing, focus on a single function at a time and work in the order in which they are described. The functions in this programming assignment are all pretty self-contained. They also increase in complexity, so tackling **`computeOutgoingEdges`** first will be a good warm-up for the remaining functions and should allow you to start getting at least some points for your submission.
- For the traversal functions, remember you have complete control over what you store in your work list. In class we simply stored the vertices, but you have the freedom to store extra information as needed.
- For runtime complexity, remember to be aware of the cost of the various operations you are performing on your collections. Especially when dealing with paths, remember that for large graphs the paths can get quite long, and the cost of making copies will add up.

Academic Integrity

As a gentle reminder, please re-read the academic integrity policy of the course. I will continue to remind you throughout the semester and hope to avoid any incidents.

What Constitutes a Violation of Academic Integrity?

These bullets should be obvious things not to do (but commonly occur):

- Turning in your friend's code/write-up (obvious).
- Turning in solutions you found on Google with all the variable names changed (should be obvious). This is a copyright violation, in addition to an AI violation.
- Turning in solutions you found on Google with all the variable names changed and 2 lines added (should be obvious). This is also a copyright violation.
- Paying someone to do your work. You may as well not submit the work since you will fail the exams and the course.
- Posting to forums asking someone to solve the problem.

Note: Aggregating every [stack overflow answer—result from google—other source] because you "understand it" will likely result in full credit on assignments (if you aren't caught) and then failure on every exam. Exams don't test if you know how to use Google, but rather test your understanding (i.e., can you understand the problems to arrive at a solution on your own). Also, other students are likely doing the same thing, and then you will be wondering why 10 people that you don't know have your solution.

Other violations that may not be as obvious:

- Working with a tutor who solves the assignment with you. If you have a tutor, please contact me so that I may discuss with them what help is allowed.
- Sending your code to a friend to help them. If another student uses/submits your code, you are also liable and will be punished.
- Joining a chatroom for the course where someone posts their code once they finish, with the honor code that everyone needs to change it in order to use it.
- Reading your friend's code the night before it is due because you just need one more line to get everything working. It will most likely influence you directly or subconsciously to solve the problem identically, and your friend will also end up in trouble.

What Collaboration is Allowed?

Assignments in this course should be solved individually with only assistance from course staff and allowed resources. You may discuss and help one another with technical issues, such as how to get your compiler running, etc. There is a gray area when it comes to discussing the problems with your peers and I do encourage you to work with one another to solve problems. That is the best way to learn and overcome obstacles. At the same time you need to be sure you do not overstep and not plagiarize. Talking out how you eventually reached the solution from a high level is okay:

"I used a stack to store the data and then looked for the value to return."

but explaining every step in detail/pseudocode is not okay:

"I copied the file tutorial into my code at the start of the function, then created a stack and pushed all of the data onto the stack, and finished by popping the elements until the value is found and use a return statement."

The first example is OK but the second is basically a summary of your code and is not acceptable, and remember that you shouldn't be showing any code at all for how to do any of it. Regardless of where you are working, you must always follow this rule: Never come away from discussions with your peers with any written work, either typed or photographed, and especially do not share or allow viewing of your written code.

What Resources are Allowed?

With all of this said, please feel free to use any [files—examples—tutorials] that we provide directly in your code (with proper attribution). Feel free to directly use anything from lectures or recitations. You will never be penalized for doing so, but you should always provide attribution/ citation for where you retrieved code from. Just remember, if you are citing an algorithm that is not provided by us, then you are probably overstepping. More explicitly, you may use any of the following resources (with proper citation/attribu- tion):

- Any example files posted on the course webpage (from lecture or recitation).
- Any code that the instructor provides.
- Any code that the TAs provide.
- Any code from the Java API.

Amnesty Policy

We understand that students are under a lot of pressure and people make mistakes. If you have concerns that you may have violated academic integrity on a particular assignment, and would like to withdraw the assignment, you may do so by sending us an email **BEFORE THE VIOLATION IS DISCOVERED BY US.**

The email should take the following format:

```
Dear Dr. Mikida,

I wish to inform you that on assignment X, the work I submitted was not
entirely my own. I would like to withdraw my submission from consideration
to preserve academic integrity.

J.Q. Student
Person #12345678
UBIT: jqstuden
```

When we receive this email, student J would receive a 0 on assignment X, but would not receive an F for the course, and would not be reported to the office of academic integrity.