Programming Assignment #3

Tests due: 4/9/23 @ 11:59pm
Implementation due: 4/16/23 @ 11:59pm

Assignment Link: https://classroom.github.com/a/3ZW7B3W_

Please read through the entire writeup before beginning the programming assignment

Objectives

1. Define methods to efficiently explore a graph representing a street map
   a. Generate an adjacency list for an graph that only stores an edge list to make searching the graph more efficient
   b. Search the graph to find a path from a given starting intersection to a given ending intersection that minimizes the number of intersections traveled through
   c. Search the graph to find a path from a given starting intersection to a given ending intersection that minimizes distance traveled

Useful Links

1. The Scala API
   a. Scala Collections
   b. scala.collections.mutable.Map
   c. scala.collection.mutable.Buffer
   d. scala.collection.mutable.Queue
   e. scala.collection.mutable.PriorityQueue
   f. scala.math.Ordering
2. Scala Tour
3. Scala Resources
4. ScalaTest: Writing your first test
5. Maps in Scala
6. Scala File I/O (Scala Cookbook Excerpt)
7. Scala Exercises

Submission Process, Late Policy and Grading

Testing due date: 4/9/23 @ 11:59PM
Implementation due date: 4/16/23 @ 11:59PM
Total points: 30 (5 for testing + 20 for implementation + 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.**

- If your submission is made before the deadline, you will be awarded 100% of the points your project earns.
- If your submission is made up to 24 hours after the deadline, you will be awarded 75% of the points your project earns.
- If your 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.
- If your submission is made more than 48 hours after the deadline, it will not be accepted.

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 sufficiently 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 Friday and you make a submission on Saturday, you will automatically use a grace day, regardless of whether you perform better or not. Be sure to test your code before submitting, especially with late submissions in order to avoid wasting grace days.

Keep track of the time if you are working up until the deadline. Submissions become late after the set deadline. Keep in mind that submissions will close 48 hours after the original deadline and you will not be able to submit your code after that time.

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 setup as in PA0, you can accept the PA3 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 `MapUtils`, which includes the construction of an adjacency list, and two different implementations of graph traversal that compute shortest paths between locations according to two different metrics.

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

As with PA2, the first phase of PA3 will be to write tests which must be able to pass on perfect 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 PA3 testing submission in Autolab. After completing your implementation, make sure to commit **and push** your work to GitHub, and submit to the PA3 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 get in the habit of 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.

0. Testing Phase (due 4/9/23)   [5/30 points]

Modify the file `MapUtils.scala` 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.

1. Generate Adjacency Lists   [4/30 points]

Implement the following function in `MapUtils`:

```
def computeOutgoingEdges(
    graph: StreetGraph
  ): mutable.Map[String, mutable.Seq[Edge]]
```

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 graph's `edges` field.
- The key for this entry should be the identifier of the intersection
- The value for this entry should be a collection of every `Edge` who's `from` field is equal to the key.

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

2. Find the Path with the Fewest Intersections   [8/30 points]

Implement the following function in `MapUtils`:

```
def pathWithFewestIntersections(
    graph: StreetGraph,
    outgoingEdges: mutable.Map[String, mutable.Seq[Edge]],
    from: String,
    to: String
  ): Seq[Edge]
```

This function should return a path between the intersection with identifier `from`, to the intersection with `to`. The returned path should contain the fewest path segments possible.

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

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

3. Find the Path with the Shortest Distance   [8/30 points]

Implement the following function in `MapUtils`:

```
def pathWithShortestDistance(
    graph: StreetGraph,
    outgoingEdges: mutable.Map[String, mutable.Seq[Edge]],
    from: String,
    to: String
  ): Seq[Edge]
```

This function should return a path between the intersection with identifier `from`, to the intersection with `to`. The returned path should contain path segments who's distance sums up to the smallest distance possible.

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

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

4. Runtime complexity   [5/30 points]

The final 5 points for this assignment will come after the submission deadline from separate testing that ensures your functions have the desired runtime complexity.

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 intersection's identifier. An `Intersection` consists of an identifier, and a pair of geospatial coordinates. You can find the (approximate) distance in km between two Intersections `a` and `b` by calling `a.distanceTo(b)`

2. `graph.edges`: A collection of edges between intersections (i.e., street segments). Assume that edges for segments of streets with two-way traffic are present in both directions (i.e., if traffic flows both ways on "W Tupper Street", then there is an edge from "W Tupper Street and Delaware Ave" to "W Tupper Street and Franklin Street", as well as the back edge. An `Edge` object contains two intersection identifiers (see `graph.intersections`) for the `from` and `to` intersections it connects (remember, these are directed edges), as well as the name of the street.

An example data file (as passed to `StreetGraph.load`) is provided in `data/buffalo_map.xml`. This file follows the standard [OpenStreetMap XML format](#). Feel free to download your own examples from OSM, or create your own.

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` from vertices to lists of edges, but it accomplishes the same goal of allowing us quick access to the outgoing edges for a particular vertex in order to make traversal more efficient. 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), which uses a `Stack`; Vertices adjacent to the most recently explored vertex are explored first.
- Breadth First Search (BFS), which uses a `Queue`; Vertices are explored in the order in which they are discovered.
- Djikstra's algorithm, a variant of Breadth First Search, where we explore vertices according to some sort of priority order, by using a `PriorityQueue`. For example, if each edge is assigned the distance one needs to travel to cross it, then we can assign a higher priority to vertices closer to the origin vertex.

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 Djikstra'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 Djiktra'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 Djikstra'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.

Priority Queues in Scala

Scala provides an implementation of the **PriorityQueue** ADT based on a binary heap.
PriorityQueue supports the standard ADT operations, including **enqueue**, **dequeue**, and
length.

PriorityQueue must be defined with an **Ordering**. Scala will automatically try to find an
ordering that 'fits' the base type of the **PriorityQueue** but you can also provide one manually.
An example of an ordering is given in **Path.scala**, and you can use it to create an empty
PriorityQueue with:

```
val queue = PriorityQueue[Seq[Edge]].empty(Path.decreasingPathDistanceOrder)
```

Alternatively, you can define your own for any type **T**

```
val queue = PriorityQueue.empty[T](
  new Ordering[T] {
    def compare(x: T, y: T): Int =
    {
      if(x is less than y){ return -1 }
      if(y is less than x){ return 1 }
      return 0
    }
  }
)
```

Immutable Lists in Scala

Scala's **List** class implements a linked list that is immutable and singly linked. As we discussed
in the first week of classes, immutability means that the value of the class can not change once
it has been allocated.

In the case of a linked list, immutability gives us an interesting benefit: The tail of the linked list
can be safely shared between multiple linked lists. For example, consider the lists:

1. M, N, D, C, B, A
2. E, D, C, B, A
3. P, Q, C, B, A

Since all of these lists share their tails, we can avoid duplication (or unnecessary copies) by
keeping only one copy of the shared tail. Pictorially this would look like:
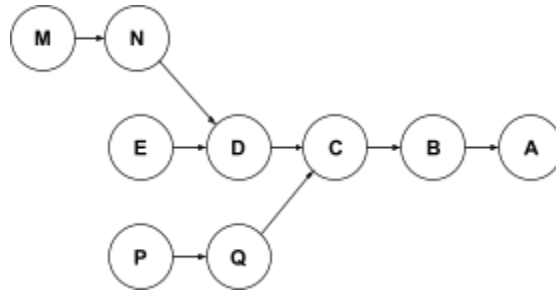
Notice how there is only a single copy of nodes D, C, B, and A even though they are part of multiple lists.

Immutability forces most linked list operations (**insert**, **append**, **remove**, **update**) to copy all preceding elements, making these operations *O(n)*. However, since the head has no preceding elements, operations on the head (**prepend**, **removeHead**) remain *O(1)*. Clever use of this property can allow you to store multiple lists sharing a common tail without making expensive copies.

Useful operations on the **List** class include:

```
Nil           // An empty list (O(1))
x :: Y        // The List Y with element x prepended (O(1))
Y.head        // The head element of List Y (O(1))
Y.tail        // List Y, excluding its head element (O(1))
```

Other Tips

1. Before you actually start writing tests, first draw out a number of example graphs. They don't have to be particularly large or complex. 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, or for less thorough implementations to fail.
   a. "What if I add another edge here that creates a cycle?"
   b. "What if there is not a path between these two nodes?"
   c. "What if there are multiple paths between these two nodes (with certain properties)?
   d. etc.

   These questions might lead you to a few additional test cases or modifications to your example graphs you had not considered.
2. Once you start 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.
3. 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 considering you are now computing entire paths, you may want to store additional information in your `Queue`/`PriorityQueues` so that when you reach your destination you have all of the information you need about the path you took to get there.
4. 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. Look to the section on immutable lists for a way to avoid the cost of copying lists that you know won't be changing.

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 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/attribution):
- 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 tour of Scala (https://docs.scala-lang.org/tour/tour-of-scala.html)
- Any code from Scala Collections (https://docs.scala-lang.org/overviews/collections-2.13/introduction.html)
- Any code from Scala API (https://www.scala-lang.org/api/2.13.0/)

Omitting citation/attribution will result in an AI violation (and lawsuits later in life at your job). This is true even if you are using resources provided.

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 me an email BEFORE THE VIOLATION IS DISCOVERED BY ME. 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.