Programming Assignment #2

Programming assignments are to be done individually. You may discuss the problem and general concepts with other students, and you may post testcases and solutions on Piazza to share with *everyone*, but there should be no sharing of code. You may not submit code other than that which you write yourself or is provided with the assignment. This restriction specifically prohibits downloading code from the Internet. If any code you submit is in violation of this policy, you will receive no credit for the entire assignment, and there may be other penalties, including reporting you to SJS.

This project has several goals:

- Ensure that you understand graphs and greedy algorithms.
- Explore the trade-offs between memory and run-time efficiency, in particular, by designing and analyzing an efficient representation of a graph.
- Get a hands-on and empirical understanding of the suitability of greedy algorithms as opposed to non-greedy algorithms and the trade-offs involved in algorithm selection and design, as the problem size is varied. In particular, note that a greedy algorithm is often not optimal, but can still be "good enough" for many instances of certain problems.

We have provided partial Java implementations that you should use to begin your solution for each required section. You do not have to implement any of the input processing or output generation.
This has all been done for you.

If you add your own print statements to any part of the solution other than in `Driver.java`, you are responsible for removing them before submitting your solution.
If your debug output interferes with the grader, you may lose points for an otherwise correct submission.

Problem Description

In this programming assignment, we will be looking at the problem of routing packets in a mobile, wireless network. Greedy Perimeter Stateless Routing (GPSR) [1], is a responsive and efficient routing protocol for mobile, wireless networks. GPSR exploits the correspondence between geographic position and connectivity in a wireless network by using the positions of nodes to make packet forwarding decisions. GPSR uses greedy forwarding to forward packets to nodes that are always progressively closer to the destination. In regions of the network where such a greedy path does not exist (i.e. the connecting paths require us to move temporarily further away from the destination), GPSR recovers by forwarding in perimeter mode in which a packet traverses successively closer faces of a planar subgraph of the full wireless network connectivity graph until reaching a node closer to the destination where greedy forwarding resumes. However, for the purposes of this programming assignment, we will assume that GPSR fails if it reaches a node from which it cannot find a forwarding node that is closer to the destination than itself.

The nodes are represented as vertices of a graph with associated locations specified by x and y coordinates. The radius of any vertex is the maximum transmission range of that node. A vertex can connect (i.e. form an edge) to only those vertices that are within its radius. A vertex which initializes a connection is called the source and the intended recipient of the source's communication is called the sink. If the sink is located outside of the radius of the source, the GPSR algorithm connects the source to a vertex within its radius that is closest to the sink. This intermediate vertex is called the first hop. The first hop then repeats this process and determines whether it can connect to the sink directly or whether it has to connect to a second intermediate vertex – the second hop. This process continues until either a path is established

between the source and the sink or an intermediate hop cannot find a vertex within its radius that is closer to the sink than itself. In this case we declare the GPSR algorithm to have failed.

Despite the usual correlation of distance with connectivity between nodes, we cannot rely on distance to find the shortest (lowest-latency) path from source to sink. Using distance as a metric will achieve a solution that has, on average, very low latency; however, sometimes a node in the path is unresponsive or slower than expected, and it is more efficient to route around that node, even if that means taking many more steps. Therefore, we must consider the amount of time it takes a packet to travel between each pair of nodes in a network and use this information to choose a path through the network which minimizes latency from the source to the sink. For nodes that are out of transmission range of each other, the latency is infinite. You will be given the latency between the nodes in the system assuming they can be reached. The maximum transmission range will be specified independently of these latencies, so your implementation will be responsible for determining whether to use the given latency or infinity, based on whether a node is in range.

Provided Starter Code

You are provided with the following partial Java implementations. **DO NOT rename any of the given classes or their fields and methods.** You are free to add any fields and/or methods to `Program2.java` that you think will be useful. Do not forget to modify the constructors when you add new fields to ensure that your new fields are initialized, but do not change the signatures of the constructors, or add any new ones.

- The `Vertex` class, which has fields `x` and `y` that define the `x` and `y` coordinates of the vertex respectively.
- The `VertexNetwork` class, which has fields `location` (an `ArrayList` of type `Vertex`) that defines the list of vertices in the graph along with their coordinates and `transmissionRange` that defines the radius of each vertex. We assume that all vertices have the same transmission radius. You should add a third field to represent the graph defined by the vertex coordinates in `location` and the radius in `transmissionRange`. In addition, the `VertexNetwork` class includes the following methods:

  (a) The `gpsrPath` method, which takes as inputs the indices of the source and the sink and returns a path between them using the GPSR algorithm. This path is represented as an `ArrayList` of type `Vertex` that lists the vertices in the path in their correct order. If the GPSR algorithm fails to find a path, the `gpsrPath` method returns an empty `ArrayList`. You need to implement this method.

  (b) The `dijkstraPathLatency` method, which is identical to the `gpsrPath` method except that it uses Dijkstra's algorithm (discussed below) to find the shortest path (in latency) between the source and the sink. You need to implement this method.

  (c) The `dijkstraPathHops` method, which is identical to the `gpsrPath` method except that it uses Dijkstra's algorithm (discussed below) to find the shortest path (in hops) between the source and the sink. You need to implement this method.

  (d) The `gpsrAllPairs` method, which takes as input a boolean print and finds a path, using the `gpsrPath` method, between all possible pairs of source and sink vertices. These paths are optionally displayed if `print` is `true`. This method also displays the number of times the GPSR algorithm was successful in finding a path and the average time taken in doing so. This method has been implemented for you.

  (e) The `dijkstraLatencyAllPairs` method, which is identical to the `gpsrAllPairs` method except that it uses the `dijkstraPathLatency` method to find paths. This method has been implemented for you.

(f) The `dijkstraHopsAllPairs` method, which is identical to the `gpsrAllPairs` method except that it uses the `dijkstraPathHops` method to find paths. This method has been implemented for you.

The `Driver.java` file tests your implementations on various inputs. You may modify this file if you wish, but please note that we will replace your `Driver.java` with our own when we grade your solution. Therefore, ensure that you don't intend for any code in `Driver.java` to be graded and that the correctness of your solution is not dependent on any code in `Driver.java`.

Part 1: Implement the GPSR Algorithm [30 points]

Implement the GPSR algorithm described above by filling in the `gpsrPath` method. You have been given 2 test input files `SmallInputGraph.in` and `LargeInputGraph.in`, which contain the location information of five vertices and hundred vertices respectively. Use the default `transmissionRange` value of one meter and call `gpsrAllPairs` with `print = true`. This has been implemented for you in `Driver.java`.

Part 2: Dijkstra's Algorithm [40 points]

An alternative algorithm for finding the shortest path between pairs of vertices in a graph is the well-known Dijkstra's algorithm [2]:

The node at which we start is called the source node. The distance of any node is defined as the distance from the source node to that particular node. For this project, we will consider distance in Dijkstra's algorithm in terms of both network latency and the number of network hops. For example, if transmitting a packet between nodes A and B takes 20 ns, the distance is either 20 ns or 1 hop, respectively.

In Dijkstra's algorithm, we assign some initial distance values to all nodes and then try to improve them step by step as follows:

a) Assign a tentative distance value of zero for the source node and infinity for all other nodes.
b) Mark all nodes as unvisited. Set the source node as the current node. Create a set of unvisited nodes called the unvisited set that consists of all the nodes except the source node.
c) For the current node, consider all of its unvisited neighbors and calculate their tentative distances. For example, if the current node *A* is marked with a distance of 6, and an edge connects it to a neighboring node *B*, then the distance to *B* (through *A*) will be 6 + 1 = 7. If this distance is less than the previously recorded distance, then overwrite that distance. Even though a neighbor has been examined, it is not marked as visited at this time, and it remains in the unvisited set.
d) When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.
e) If the sink node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal), then stop. The algorithm has completed.
f) Set the unvisited node marked with the smallest tentative distance as the next current node and go back to step 3.

Pseudo code for Dijkstra's algorithm is as follows [2]:

Algorithm

```
1: Function Dijkstra(Graph, source,sink)
2: for each vertex v in Graph // Initializations do
3:    dist[v] := infinity ; // Unknown distance function from source to v
4:    previous[v] := undefined ; // Previous node in optimal path from source
5: end for
6: dist[source] := 0 ; // Distance from source to source
7: Q := the set of all nodes in Graph ; // All nodes in the graph are
unoptimized - thus are in Q
8: while Q is not empty: // The main loop
9:    u := vertex in Q with smallest distance in dist[] ;
10:   if dist[u] = infinity then
11:         break;
12:   end if
13:   remove u from Q ;
14:   if u = sink then
15:         break;
16:   end if
17:   for each neighbor v of u: // where v has not yet been removed from Q. do
18:         alt := dist[u] + dist_between(u, v) ;
19:         if alt < dist[v]: // Relax (u,v,a) then
20:               dist[v] := alt ;
21:               previous[v] := u ;
22:               decrease-key v in Q; // Reorder v in the Queue
23:         end if
24:   end for
25: end while;
26: return dist[] ;
27: end Dijkstra
end
```

Implement Dijkstra's algorithm, described above, by filling in the `dijkstraPathLatency()` and `dijkstraPathHops()` methods. In the first method, you will use the latencies given to you in the edges as the weights/distances in Dijkstra's algorithm. For this project, remember that a node can only transmit to other nodes within its radius. Treat the weight of an edge which goes to a node outside the transmission range as infinity (in other words, you cannot take that edge).

The second method will act as an approximation for GPSR by treating the weight of every edge as having weight 1 and trying to minimize the weight of the graph. Note that the version which minimizes hops is a special case of the previous implementation of Dijkstra's algorithm (which treats every edge as having weight 1), so you should be able to re-use a large portion of the implementation between these methods.

You have been given test input files `SmallInputGraph.in` and `LargeInputGraph.in`, which contain the location information of five vertices and hundred vertices respectively and every edge between them. Use the default `transmissionRange` value of one meter and call `dijkstraAllPairsLatency` and `dijkstraAllPairsHops` with `print = true`. These methods have been implemented for you in `VertexNetwork.java`, and you can call them from an instance of `Program2`.

The `transmissionRange` can be set from the Program2 constructor, or by using the `setTransmissionRange()` method from `VertexNetwork` (the superclass of `Program2`). If you would like to change the behavior of `setTransmissionRange()` - for instance, you would like to modify your graph representations when you change the transmission range – override `setTransmissionRange()` from `Program2.java`. Note again that if you modify the `setTransmissionRange()` in `VertexNetwork`, it will not be graded as part of your solution.

Part 3: Report [30 points]

Write a report that analyzes various aspects of your implementation as described below.

Proofs

Include a section in your report where you discuss the following:

a) Give a brief proof which shows that GPSR, while greedy, is not an optimal solution for the wireless routing problem.

b) Argue that `dijkstraPathHops()` in general does a better job than GPSR at finding a path with the minimum number of hops, but is still not an optimal solution to the routing problem.

Efficiency Analysis

Analyze the various metrics of efficiency in your implementations:

a) Give a Big-O analysis of the memory space efficiency of your graph representation in terms of nodes and edges. Note that the graphs we give you have specified every possible edge between nodes, but you may not be able to use all of the edges because of the limitation of transmission distance.
Discuss any optimizations you made. If you used a different graph representation for GPSR than for Dijkstra's algorithm, explain the trade-offs you made, and analyze both. (Some extra points may be awarded if you have done something particularly clever, not to exceed the highest possible score on the project.)

b) Give a Big-O analysis of the runtime complexity of the algorithm you used to generate those graph representations. You may write pseudo-code here if it will help clarify your discussion, but it is not necessary. Justify the design decisions you made in optimizing for space versus runtime.

c) Bearing in mind the graph representations you have devised for this project and the trade-offs between space requirements and runtime, give a Big-O analysis of the runtime for your implementation of Dijkstra's algorithm. If the runtime is worse than expected, explain why.

Runtime Efficiency and Success Rate

Investigate the running time, space requirements, and success rate of GPSR and the implementations of Dijkstra's algorithm for different transmission radii. For this, you have been given a test input file called `LargeInputGraph.in`, which contains the location information of fifty vertices.

You need to do the following:

a) Call `gpsrAllPairs` with `print = false` and use the following values for vertex radii: `[5 10 15 20 25]`. Note that you can run the `gpsrAllPairs` method five times by giving the `transmissionRange` values after the '-t' option.

b) Call `dijkstraAllPairsLatency` with `print = false` and use the following values for vertex radii: `[5 10 15 20 25]`. Note that you can run the `dijkstraAllPairsLatency` method five times by giving the `transmissionRange` values after the '-t' option.

c) Repeat the previous task using `dijkstraAllPairsHops`.

d) Plot the success rate and the average running time versus the vertex radius for all the above algorithms. Explain the impact of the vertex radius on the performance of the algorithms, and on the basis of that explain when you might prefer GPSR over Dijkstra and vice versa.

Instructions and Tips

a) Implement your solution in `Program2.java`. You may also modify `Driver.java` as you see fit, but please note that we will replace your `Driver.java` with our own Driver program when we grade your solution. Therefore, ensure that you don't intend for any code in `Driver.java` to be graded, and that the correctness of your solution is not dependent on any code in `Driver.java`.

b) You will be required to generate a graph representation for this project. We have left up to you what kind of representation to use, and you will analyze your choice in your report. It may be helpful to note that the graph we give you specifies every possible edge, but you must treat the weight of an edge as infinite (or in other words, non-existent) if the node at the other end of the edge is outside of the transmission radius, because the nodes are not connected. This rule applies to both the GPSR and Dijkstra implementations.

c) Please include your name and UT EID at the top of any of the files you submit, including `Program2.java`, and your report. This will help us identify your solution in case something strange happens.

d) Submit .java files (only the ones that you modified), not .class files. Identifying this problem, and then decompiling your code so we can grade it is not our idea of a fun weekend.

e) **Submit only a single .zip file containing both the report and code.**

f) You are responsible for ensuring that your solution is compatible with the grader, by following the recommendations below.
   a) Aside from `Program2.java` and `Driver.java`, do not modify any of the other files we have given you, or your implementation will become incompatible with the grader and may appear to be incorrect, even if it is correct on your machine.
   b) DO NOT rename any of the given classes or their fields and methods. You are free to add any fields and/or methods to `Program2.java` that you think will be useful. Do not forget to modify the constructors when you add new fields to ensure that your new fields are initialized, but do not change the signatures of the constructors, or add any new ones.
   c) You may add additional files/classes. But be sure not to rename any classes or methods we have provided for you, or your solution will become incompatible with the grader and your submission will fail to compile. Of the files we have given you, we will overwrite all files except `Program2.java` in order to grade your program. Any additional files you have added will be left untouched and will be compiled with your solution.
   d) If you add your own print statements to any part of the solution other than in `Driver.java`, you are responsible for removing them before submitting your solution. If your **debug output interferes with the grader**, you may **lose points for an otherwise correct submission**. You may wrap your print statements with the `if (DEBUG) /* print */` paradigm, just be sure to set `DEBUG=false` before submitting. Alternatively, you may print to `System.err` instead of `System.out`, because we will ignore any output written to `System.err`.
   e) DO NOT USE PACKAGE STATEMENTS. Your code will **fail to compile** in our grader if you do and you will **lose points**.
   f) When you create your .zip file as described in the submission section below, please open up your file in your archive viewer to ensure that the files are visible from the root of the archive, i.e. there should be no folder inside your zip file. If there is, it will cause issues when we try to

grade your program, and points will be deducted because we will have to manually correct the issue.
In particular, if you right-click on a folder and select an option to create a zip file, that entire folder, including the folder itself, will be present in the archive. Instead, select the specific files you intend to zip up, and then right-click and create the archive (or however your particular zip software works).
We recommend 7-Zip for Windows users. (http://www.7-zip.org/) With 7-Zip, you can perform the above task with right click - 7-Zip - Add to "[name].zip".

Submission

You should submit a single file titled `EID_LastName_FirstName.zip`, which includes your implementation in `Program2.java`, and any other source files you have produced along with a report (.pdf file) titled `EID_LastName_FirstName.pdf` that contains your plots and discussion of the algorithms' performance. Your solution must be submitted via Canvas by 11:00 pm on the day it is due. You may use up to 1 slip day.
If you name the .zip file or the .pdf file incorrectly, points will be deducted.

Checklist

- ☐ Did you add a completed header for *all* your submitted .java files?
- ☐ Did you remove all package statements from your .java files?
- ☐ Did you put in your name and EID on the report file?
- ☐ Did you add all your source files and the report file in the zip file, with no directory structure?
- ☐ Did you remember not to include .class files, and any files that we did not ask you to submit?
- ☐ Did you remember to name the zip file `eid_lastname_firstname.zip`?
- ☐ Did you remember to work on your own and follow the honor code?

References

[1] B. Karp and H. T. Kung, "GPSR: greedy perimeter stateless routing for wireless networks," in MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking. New York, NY, USA: ACM, 2000, pp. 243-254.

[2] "Dijkstra's algorithm." [Online]. Available: http://en.wikipedia.org/wiki/Dijkstra's algorithm