22C:21: Computer Science II: Data Structures
Project 2

Project 2 has 3 parts: (I) implementing a new `myGraph` class, using the adjacency list representation, (II) implementing *breadth first traversal* and the computing shortest paths between pairs of vertices, and (III) implementing a geometric routing algorithm called *greedy perimeter stateless routing (GPSR)*. The three parts are due back separately: Part I on October 31, Part II on November 7, and Part III on November 14.

1 Part I: The new `myGraph` class

Implement the `myGraph` class using the adjacency list representation, as described in the lectures. You should allow for the edges to have associated real number weights. The main difference between the new `myGraph` class and the version you used in Project 1 is in the data structure used to store the edges of the graph. For this project, define the data member `Edges` as follows:

```
LinkList[] Edges;
```

Slot i of the array `Edges` stores the neighbors of the ith vertex as a singly linked list. Each `Link` object in the link list would contain an integer representing the index of a vertex and a real number representing the weight of an edge. There is no need for you to implement the `LinkList` class from scratch. You should start with Lafore's `LinkList` class (to download this class, follow the link from the course page) and make the following changes to this class.

- Add a data member called `numLinks` to the `LinkList` class. Use this to keep track of the number of `Link` objects there are in the linked list.
- Add a member function `int size()` that returns the number of `Link` objects in the linked list.
- Add a second `insertFirst` member function, which takes just an `int` parameter. This allows the specification of the `double` parameter to be optional. In other words, the new `LinkList` class should contain both of these functions:

```
void insertFirst(int id, double dd);
void insertFirst(int id);
```

Make sure that the interface of your new `myGraph` class contains all the methods provided in the `myGraph` class in *our* solution to Project 1. This is important because we would like to use your new `myGraph` class and our old `myGraph` class, interchangeably. For example, our `myGraph` class contains the function:

```
void addEdge(String vertex1, String vertex2, double distance);
```

You should make sure that your implementation contains a function with exactly this function header. In addition, also provide an `addEdge` function that just takes the endpoints, `vertex1` and `vertex2` as arguments.

After you complete your implementation of the new `myGraph` class, you need to compare the efficiency of your new implementation to that of the old `myGraph` class. To ensure that all of you are doing the same experiment, I would like you to compare your new `myGraph` class with our *old* `myGraph` class (from the solution to Project 1). Specifically, I would like you to time two functions from the `wirelessNetwork` class: (i) the `wirelessNetwork` constructor and (ii) the topology control function, to see if these two functions have very different running times depending on whether we use the old `myGraph` class or the new `myGraph` class. To do this comparison, construct 10 `wirelessNetwork` objects, each obtained by dropping points uniformly at random on a 10×10 square using varying the number of points. Use $500, 550, 600, \ldots, 950$ points for the 10 `wirelessNetwork` objects and report on the time Run topology control on each of these 10 wireless networks and report the time. Summarize your observations in 3-4 sentences.

What to submit: Three files (i) `myGraph.java`, (ii) `experiments.java`, and (iii) a file called `results`, containing tables of running times and your 3-4 sentence summary.

2 Part II: Breadth First Traversal

Like depth first traversal, breadth first traversal is a certain way of traversing graphs. The main advantage of using breadth first traversal is that it can be used to compute shortest paths between pairs of vertices. Breadth first traversal (BFT) is similar in structure to depth first traversal (DFT) in that BFT also uses a data structure to store vertices as they are being processed. Recall that DFT used a stack to store vertices that it discovered. BFT uses a data structure called a *queue*. A *queue* is a data structure that returns elements in *first in first out* (FIFO) order. More precisely, a queue data structure typically provides the following operations:

enqueue: takes an item and inserts it into the queue.

dequeue: returns the item in the queue that was inserted earliest and removes it.

isEmpty: returns a boolean value indicating if the queue is empty.

Notice that the queue data structure has functionality that is quite similar to that of the stack data structure, expect that the stack data structure returns elements in the LIFO order while the queue data structure returns them in the FIFO order. A typical implementation of the queue data structure will contain at least these three functions and possibly others.

Here is pseudocode for BFT.

```
visited[source] = true;
Q.enqueue(source);

while(Q not empty)do
          current = Q.dequeue();
          for each unvisited neighbor v of current do
                    visited[v] = true;
                    Q.enqueue(v);
```

Like DFT, BFT takes a `source` vertex and starts the traversal from there. Like DFT, BFT maintains a boolean array called `visited` to keep track of the vertices already visited. The difference between the two traversals is that BFT scans the neighbors of the current vertex and enqueues *every* unvisited neighbor of the current vertex. You may recall that DFT looks for and pushes one unvisited neighbor of the current vertex into the stack, before going back to the beginning of the outer while-loop.

Doing a BFT on a graph defines a *breadth first traversal tree*. The source vertex is the root of this tree and for every other vertex v in the graph, we define the parent of v in the BFT tree as the vertex u from which we first discovered v. The following figure illustrates the notion of BFT tree. A BFT tree has the nice property that every path in the tree from the root of the tree to a vertex, is a shortest path in the graph. Thus, one way to compute a shortest path between a pair of vertices s and t is to do a BFT from s, construct a BFT rooted at s, and then find the path from t to s in the BFT tree.

Here are some more details on how to implement BFT.

- Lafore has a simple array based implementation of the queue data structure. You will find a link to this from the course page. Use this implementation to define a queue for BFT. You will have to read Lafore's code to find out specific details of this implementation.

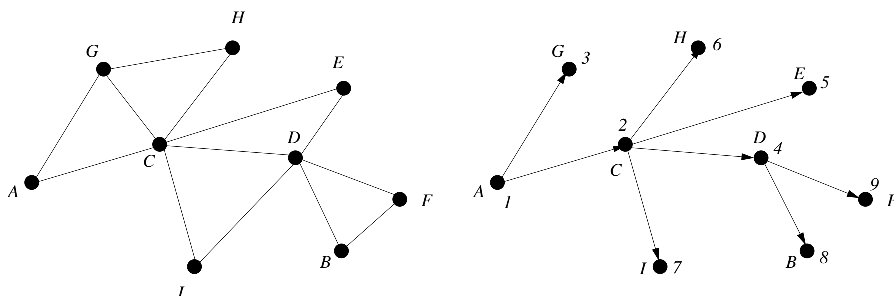

Figure 1: A BFT tree is shown on the right, for the graph on the left. It is assumed that the source of the traversal is A and the neighbors of each vertex are scanned in alphabetical order. In the BFT tree, arrows point from a parent to a child. The numbers shown next to the vertices show the order in which the vertices are discovered. It is easy to verify that this BFT tree gives shortest paths from vertex A to every other vertex in the graph.

- The pseudocode for BFT that I have shown above is missing the code needed to construct a BFT tree. It is your responsibility to supply this. You should mimic the approach used in constructing the DFT tree.
- Like DFT, BFT will only traverse the connected component of the graph that the source vertex lies in, unless it is forced to restart the traversal from some vertex in each unexplored component. The above pseudocode does not force BFT to continue exploring other connected components. Your implementation should. Again, you should mimic the approach we used in DFT.

2.1 Experiments

Perform experiments 2 and 3 from Project 1. But, this time I would like you to compare the length of the path returned by compass routing (assuming that a path is found) to the length of a shortest path between s and t. More precisely, here are the experiments you need to perform.

1. Generate a wireless network G with 1000 points distributed on a 10×10square. Then repeat the following 10 times. Uniformly at random, pick a vertex and designate it the source s and again uniformly at random, pick a vertex and designate it the destination t. Run compass routing on G, with source s and destination t. Report on whether compass routing was able to find the destination or not and if it was, report on the length of the path from s to t that was discovered. Then compute a shortest path between s and t and report its length. Report the ratio of the length of the path between s and t computed by compass routing to the length of a shortest path between s and t. Present your results in a tabular form.
2. Start with the same wireless network G generated for Experiment (2) and run topology control on it to get a sparser network H. Run Experiment (1) again, but use the network H this time. Tabulate your results as for Experiment (1). Write 2-3 sentences commenting on how the path lengths generated in Experiment (2), compare with path lengths generated in Experiment (1). Try to explain your observations.

3 Part III: Greedy Perimeter Stateless Routing (GPSR)

In a 2000 paper, Karp and Kung described a memoryless, geometric, routing protocol called *Greedy Perimeter Stateless Routing (GPSR)* that guarantees message delivery in planar graphs.

Like compass routing, GPSR takes as input a graph and a pair of vertices s and t, where s is the source vertex and t is the destination vertex. The goal of GPSR is to find a route from s to t in the given graph. As in the case of compass routing, the main restriction on GPSR is that it is allowed to remember only a very small amount of information about the graph it is traversing. The fact that GPSR is guaranteed to run correctly only on planar graphs is not a significant bottleneck. This is because topology control protocols such as XTC, produce a planar graph as output, when given a UDG as input. Thus we can start with a wireless network modeled as a UDG, run a topology control protocol such as XTC on it, and then run GPSR on the planar graph produced by the topology control protocol.

A graph is *planar* is it can drawn in the plane without any pair of edges crossing each other. For the rest of the handout, when we talk about a planar graph, we will be referring to a drawing of the graph in the plane with no edge crossings. A planar graph partitions the plane into faces. One of these is unbounded and is called the *external face*, while the rest are bounded and are called *internal faces*. See Figure 2 for an illustration. The GPSR algorithm requires the traversal

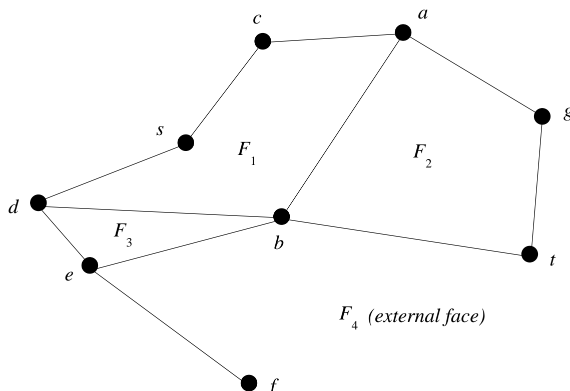

Figure 2: This planar graph has 4 faces. The three internal faces are labeled F_1, F_2, and F_3. The sole external face is labeled F_4.

of a face of a planar graph using the *right hand rule*. See Figure 3 for an illustration.

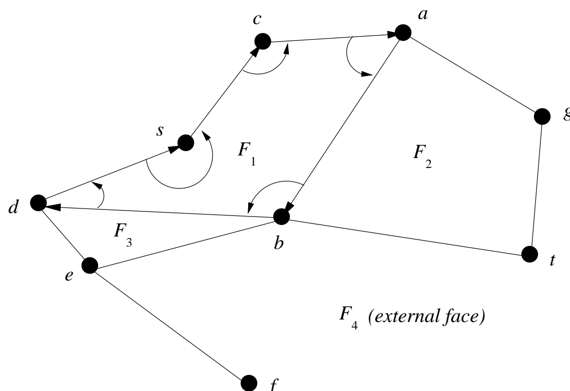

Figure 3: Suppose we start the traversal by going from s to c. The *right hand rule* requires that at c, we find the first neighbor of c after s, in counterclockwise order. This neighbor is a. At a, we find the first neighbor of a, after c, in counterclockwise order. This neighbor is b. If we continue to apply the right hand rule, we will traverse the entire boundary of F_1 and return to s.

The GPSR algorithm operates in two modes: (i) a *greedy* mode and (ii) a *perimeter* mode. The algorithm starts off at vertex s, in the greedy mode, and looks for a neighbor of s that is closer to the destination t (than s). If such a neighbor exists, then the algorithm "greedily" picks the neighbor of s that is closest to t. The algorithm proceeds in this greedy mode until it reaches a vertex x such that all neighbors of x are farther from t than x is. The vertex x is a "local minimum" because there is no way of reducing the distance to the destination t by simply moving to a neighbor. GPSR then switches to the perimeter mode, remembering the fact that it switched into perimeter mode at vertex x.

GPSR continues in perimeter mode until it reaches a vertex y whose distance to t is smaller than the distance between x and t. Recall that x is the vertex at which GPSR switched from greedy mode to perimeter mode. The perimeter mode is only intended to recover from a local minimum and is meant as a short interlude before the greedy mode can be resumed. What happens in the perimeter mode is described in next paragraph.

Refer to Figure 4 as you read this description. The line from x to t, lies in some face F_1

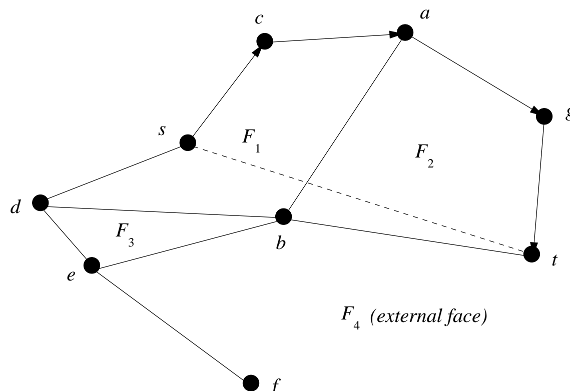

Figure 4: The source vertex s is a local minimum. So GPSR switches to the perimeter mode immediately. It then applies the right hand rule at s and finds the first neighbor of s, in counterclockwise order, after t. This neighbor is c. From c, by applying the right hand rule, we get to vertex a. Note that a is closer to t, than s is. So GPSR switches to greedy mode. It them goes to vertex g and then to vertex t in greedy mode.

containing x. GPSR starts traversing the face F_1 in counterclockwise order starting at x. GPSR uses the right hand rule described earlier to go from one edge to the next on a face. If GPSR keeps going, it will traverse the face F_1 completely and return to x. However, this will not happen because at some point GPSR will encounter the edge $\{a, b\}$ that intersects the line xt. One of the two end points of this edge, a or b, is guaranteed to be closer to t than x is. On encountering such a vertex GPSR switches to the greedy mode, if it had not already done so before getting to edge $\{a, b\}$. In any case, it is guaranteed that GPSR will get to a vertex closer to t than x by traversing a portion of the face F_1.

3.1 Details about implementation and experiments

Implement GPSR as a function in the `wirelessNetwork` class. It should have the following function header:

```
public String[] GPSR(String s, String t)
```

Like the function `compassRouting`, `GPSR` takes vertices s and t and returns a `String` array containing a path from s to t.

Perform the following experiment. Generate a wireless network G with 1000 points distributed on a 10×10 square. Then repeat the following 10 times. Run topology control on G

to get a planar graph H. Uniformly at random, pick a vertex and designate it the source s and again uniformly at random, pick a vertex and designate it the destination t. Run the two routing algorithms on H, with source s and destination t. Report on whether compass routing was able to find the destination or not and if it was, report on the length of the path from s to t that was discovered. Report on the length of the path that GPSR found between s and t. Then compute a shortest path between s and t and report its length. Report the ratio of the length of the path between s and t computed by compass routing to the length of a shortest path between s and t. Also report the ratio of the length of the path between s and t computed by GPSR to the length of a shortest path between s and t. Present your results in a tabular form. Discuss is 2-3 sentences the relative performance of compass routing and GPSR.

3.2 What to submit

Submit the files `wirelessNetwork.java`, `experiments.java`, and `results`. Use exactly these names and do not submit any other files. We will use our *myGraph* class from the solution to Project 2.2 to test your submission.