22C:21: Computer Science II: Data Structures
Project 3: Due on Wednesday, 12/14

Project 3 has 3 parts. Part (I) allows you to work more on GPSR. Part (II) asks that you implement Dijkstra's shortest path algorithm. Part (III) asks you to build a GUI for your project. All three parts are due at the same time, at midnight on Wednesday, 12/14.

1 Part I: More on GPSR

Several students felt that they needed more time for Project 2.3. By incorporating Project 2.3 into Project 3, I am allowing you more time for GPSR. From conversations with TAs and some students, I have gathered that some of you have implementations of GPSR that go into an infinite loop for certain kinds of networks. Below, I provide an example that will hopefully show you how to sort out this problem. Also, in Project 2.3, I described a somewhat simplified version of GPSR. There are some rare situations in which the simplified GPSR fails to find an st-path even when one exists. Below, I describe a more complete version of GPSR. This is what you should implement for Project 3.

Another example of how GPSR works. Consider the following network. GPSR starts at

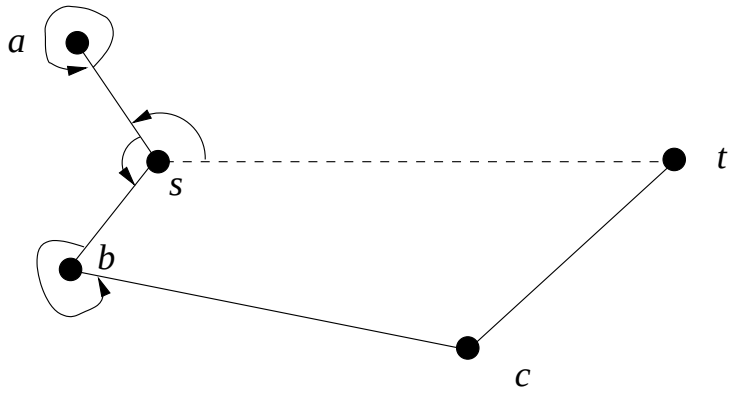

Figure 1: An example of how GPSR works.

vertex s in greedy mode and immediately switches to perimeter mode since its two neighbors a and b are further away from t. GPSR remembers the fact that it switched from greedy mode to perimeter mode at s. In general, GPSR always remembers the most recent vertex, say x, at which it switched from greedy mode to perimeter mode. GPSR remains in perimeter mode until it finds a vertex which is closer to t than x. At that point GPSR switches to greedy mode and forgets all about x. In this example, GPSR looks for a neighbor of s to go to, by going counter clockwise around s, starting in the direction $\vec{st}$. GPSR finds a and goes to it. GPSR remains in the perimeter mode and travels from a to s, from s to b and from b to c along the outer face of the network. At c, GPSR realizes that it is closer to t than s and therefore switches to greedy mode. It immediately finds t and the algorithm ends. Some students had implementations that cycled among a subset of the vertices in examples such as these. This may have been due to a confusion about when to switch back from perimeter mode to greedy mode or due to an incorrect implementation of the right hand rule.

Easy cases for the right hand rule. A minor point that might help your implementation of the right hand rule is the following. If GPSR is at a vertex of degree 1 or 2 in the perimeter

mode, it is very easy to find out where to go next. If u is of degree 1, then GPSR just travels to the only neighbor of u. If u is of degree 2, with neighbors a and b, then GPSR travels to b if it came to u from a (and vice versa). So in cases when a vertex has degree 1 or 2, it is not necessary to sort by angle to find out where to go next, in perimeter mode.

A more complete GPSR. In describing GPSR in the Project 2.3 handout, I described a simplified version of GPSR. For example, consider the following network. Suppose GPSR arrives

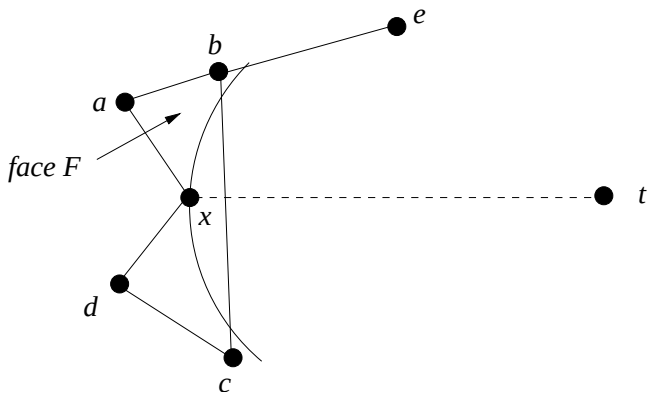

Figure 2: An example of GPSR in which multiple faces may have to be traversed before GPSR gets out of the perimeter mode.

at vertex x in the greedy mode. Then GPSR switches to perimeter mode and starts traversing the face marked F, first going to a, then to b, etc. Now notice that there are no vertices on face F that are closer to t than s. The figure above also shows part of the circle with center t and radius $|xt|$ to emphasize that all other vertices of F are farther away from t than x. So GPSR will continue to go around face F forever in the perimeter mode. This is a somewhat rare occurrence because vertices have to be arranged just so, for this to happen. However, GPSR has to deal with this situation also and here is how. As GPSR travels around face F, assuming that it has not found a vertex closer to t than x, it stops at the first edge on the face that intersects xt. In the above example, this edge would be bc. GPSR then immediately switches to traversing around the face on the other side of edge bc. Note that in a planar drawing of a graph, every edge is on the boundary of two faces, one on either side of the edge. GPSR continues in perimeter mode, possibly traversing several faces of the network, each closer to t than the previous, until it finds a vertex closer to t than x.

What does it mean for GPSR to switch to a new face, on the other side of edge bc? It simply means that instead of going from vertex b to vertex c, GPSR, finds the next neighbor of b, in counter clockwise order from c and goes to that neighbor. In the above figure that vertex is e.

2 Part II: Dijkstra's Shortest Path Algorithm

Here is Dijkstra's shortest path algorithm (DSP), as discussed in class. Recall that it takes an edge-weighted graph and a source vertex s as input and returns shortest paths from s to all vertices in the graph.

```
for each vertex v ∈ V do
        d[v] ← ∞
d[s] ← 0
S ← {s}; u ← s
for i ← 1 to n − 1 do
{
        for each neighbor v of u do
                d[v] ← min{d[v], d[u] + w(u, v)}


        u ← vertex in V − S with smallest d-value
        S ← S ∪ {u}
}
```

To implement DSP efficiently we need to maintain a binary min-heap in which each record consists of two fields: (i) a vertex index and (ii) a distance estimate. The heap is organized by distance estimates, with the vertex with smallest distance estimate at the top of the heap. The heap needs to support three operations:

1. `void INSERT(index, distance)`.
2. `int REMOVE-MIN()`.
3. `void DECREASE-DISTANCE(index, newDistance)`.

Using such a heap, DSP can be implemented as follows.

```
for each v ∈ V do
        d[v] ← ∞
d[s] ← 0
S ← {s}; u ← s
for each vertex v ∈ V do
        H.INSERT(v, d[v])

for i ← 1 to n − 1 do
{
        neighbors ← getNeighbors(u)
        for j ← 0 to neighbors.length-1 do
        {
                v ← neighbors[j]
                if (v ∈ S)
                        d[v] ← min{d[v], d[u] + w(u, v)}
                        H.DECREASE-DISTANCE(v, d[v])
        }


        u ← H.REMOVE-MIN()
        S ← S ∪ {u}
}
```

To implement the heap data structure mentioned above, start with Lafore's implementation of the binary max-heap data structure. A link to this implementation is posted on the course page. Modify this implementation so that it becomes a min-heap and supports the functionality mentioned above. The exact names of functions are not very important; feel free to use the names Lafore uses for his functions. However, one critical issue that needs to be addressed is that the `DECREASE-DISTANCE` operation takes as parameter a vertex index. As mentioned in class, a heap is not an efficient data structure for searching. So given a vertex index, how do we efficiently find its location in the heap? The solution is to maintain a mapping from vertex indices to heap locations in the heap class. Specifically, define as a data member of the heap class, an `int` array called `map` such that at all times, `map[i]` equals the location of the vertex with index `i` in the heap. If the vertex with index `i` has already departed from the heap, then `map[i]` would equal -1. As vertices move around in the heap, `map` would need to get updated immediately. This only imposes a constant amount of overhead per move and does not change the worst case asymptotic running time of the heap operations.

The above pseudocode does not contain any code for actually constructing shortest paths from s to the vertices in the graph. You will have to supply this code. The shortest paths can all be together maintained in an `int` array called `SPTree` — very similar to a `BFTTree` or a `DFTTree` that we have encountered earlier. Suppose vertex v has index i. Then `SPTree[i]` would contain the predecessor of v with in a shortest path from s to v. Thus to compute a shortest path from s to t, we would first run DSP with source s, construct `SPTree`, find the index, say i, of t, and follow the chain of predecessors in `SPTree` until we get to s.

To incorporate DSP into the `myGraph` class do the following. First, implement DSP as a public member function of the `myGraph` class using the following function header:

```
void DSP(String s)
```

Then implement a function to compute a shortest path between a given pair of vertices, using the following function header:

```
String[] shortestPath(String s, String t, boolean byHops)
```

The `boolean` flag `byHops` informs the function if the user wants a shortest st-path measured by hops or a shortest st-path measured by edge weights. Specifically, if `byHops` is sent in true, then the function needs to call breadth first traversal, otherwise it needs to call Dijkstra's shortest path algorithm. You may recall that the `myGraph` class already contains a function with the following header:

```
String[] shortestPath(String s, String t)
```

Modify this function so that it now computes a shortest path measured by edge weights. For this, it would simply need to call the above mentioned version of `shortestPath` with `byHops` set to false.

2.1 Experiments

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 GPSR on H, with source s and destination t. Report on the length of the path that GPSR found between s and t, measured in two ways: (i) the number of hops in the path and (ii) the sum of the edge weights of the edges in the path. Then compute a shortest path between s and t and also report its path length in the two ways mentioned above. Finally, report the ratio of the "hop length" of the st-path computed by GPSR to the "hop length" of a shortest st-path and the ratio of the "Euclidean distance length" of the st-path computed by GPSR to the "Euclidean distance length" of a shortest st-path. Present your results in a tabular form. Discuss in 2-3 sentences the performance of GPSR.

2.2 What to submit

Submit the files `myGraph.java`, `heap.java`, `wirelessNetwork.java`, `experiments.java`, and `results`. Use exactly these names and do not submit any other files.

3 Part III: A GUI for the project

For this part of the project feel free to unleash your creativity and get up to 50 extra credit points. To get 25 points, I would like to see an applet that does the following:

- When started, it displays a wireless network. The network may be generated randomly with some default and fixed values of the number of vertices in the network and the dimensions of the square in which they are distributed.
- Runs topology control on network and in a separate window displays the sparser version of the network.
- Picks two vertices s and t and shows (a) the st-path picked by compass routing, (b) the st-path picked by GPSR, and (c) the shortest st-path measured by edge weights.

To get the extra credit points, there are many things you can do beyond the minimum functionality mentioned above. For example, you might let the user select the number of vertices and the size of the square in which to distribute these. You might also allow the user to pick a bunch of vertices by just clicking on the window. You might also allow the allow the user to "step through" compass routing and GPSR in the sense that each time the user hits a key, the next edge selected by the routing algorithm may be highlighted. I am sure you can think of many other ways to make your GUI "cool," interesting, and easy to use.
