CS310 – Advanced Data Structures and Algorithms

Review of Project 4

Using S&W Graph

- We want to use the S&W Graph class to represent this map of stations
- The Graph has vertices numbered 0 to N, with edges identified by the two vertex numbers being connected
- We’re not using vertex 0, however.
- Each vertex represents a station, and the class Station describes that station, with info on station name, train line names, id, and location.

StationGraph

- StationGraph is a class that wraps the S&W Graph and the app information on stations (the Station objects) together.
- A StationGraph object is available from MetroSystem via
  StationGraph stationGraph = mS.getStationGraph();
- Example ParkStreet: look up info on it
  Station ps = stationGraph.stationOf("ParkStreet");
- Find its neighbors: get its S&W Graph, use its adj info:
  Graph g = stationGraph.getGraph();
  for (int n : adj(ps.getStationId())) {
    // n is a neighbor vertex no.
    Station neighbor = stationOf(n);  // then whatever...
  }

A vertex here is a Station

- Our graph has one vertex for a station even if it is on two train lines
- Example Park St.:
  Station s = stationGraph.stationOf("ParkStreet");
- Park Street station is on the Red line and the Green line, so s.getTrainLines() returns "Red", "Green".
- Its name can be found from s by s.getStationName()
Platforms

- To find the fastest route realistically, we need to penalize train line transfers, and taking the Silver Line, since it is not really rapid transit.
- We note that people actually go from one platform to another when they transfer at Park Street station.
- Park St. has two red-line platforms (one in each direction) and two green-line platforms.
- People can move from any platform at Park St. to any other platform.
- The train tracks allow them to go to the adjacent stations on the same train line.

Park St. Station has PlatformGraph vertices on the Red and Green lines

More on edges

- Our PlatformGraph has platform numbers for vertices (platforms) and edges connecting platforms (a 3-6 edge could connect platform 3 with platform 6).
- This is a directed edge. Many edges connect platforms on the same line together, following the tracks.
- Some edges connect platforms at the same station, but on different lines, for passengers to connect from one line to the other.
- A few edges connect lines to each other, such as the RedA and RedB lines south of JFK/UMass connecting to the Red line at and north of JFK/UMass.

Platforms

- A client can get a Platform object for each vertex of the PlatformGraph. From main of PlatformGraph:
- System.out.println("Platform 2 is for station ", p.getStation().getStationName());
- Its train line name is p.getTrainLine() and its platformId is p.getPlatformId(), which is the vertex number in the Graph.

Neighboring Platforms

- A client can get the Digraph for PlatformGraph by calling platformGraph.getDigraph()
- To find neighboring platforms to Platform p: use adj from this Digraph:
  Digraph dg = platformGraph.getDigraph();
  for (int n : adj(p.getPlatformId())) {
    Platform neighbor = platformOf(n);  // then whatever...}

Data structures and modeling

- This is the most complicated data structure we have tackled in this course.
- You can see that defining the objects involved and how they are related to the Graph on one hand and the real-life concepts on the other is a real challenge.
- We always simplify the real world somehow in doing this.
- Here trains go both ways along the pairs of tracks we model as simple connections between "stations", themselves combinations of real-life platforms. End up with the undirected station graph.
- Or we incorporate more detail in the PlatformGraph, a digraph.
- This process is called "modeling".
Pa4 steps

• Although we have a beautiful diagram of the real system, we need to verify that our Graph follows it properly. So we print out the sequence of station names along each train line.
• The train line names in the MBTA data include RedA and RedB for the two parts of the Red line south of the JFK/UMass station. Also GreenB, GreenC, GreenD, and GreenE.

Step 2: finding an end station

Working on trainline: String line, i.e. "Red" or whatever.
We have found that vertex v is on this line.
G.adj(v) = Iterable for collection of adjacent vertices of v.
Count neighbors of v that are on the same trainline line:
int count = 0;
for (int n : G.adj(v)) { // loop over neighbors n of v
  if (sG.stationOf(n).getTrainLines().contains(line))
    count++;
}
Then if this count == 1, we’ve found an end station.

Step 2: finding down the line

• Starting from the end station for a line, we want to find an adjacent vertex that is on the same line.
• That’s pretty easy.
• But on the next step, we want to find an adjacent vertex that’s on the same line, but is not the one we came from.
• This can be tricky, need to track the previous vertex (or vertices) so as to avoid choosing it.

Pa4 graph algorithms

• Once we are confident of the Graph setup, we turned to using some of these powerful graph algorithms that are available in the S&W library alg4.jar.
• We first used BFS to find the shortest paths through the station graph.
• That works fine but can give us a routes using the Silver line, which is relatively slow, when it could give us a train-only route. Or a route involving extra transfers.
• So back to the drawing board...
Pa4 part 4

• How can we model the time wasted waiting for a train when transferring from one train line to another?
• It only happens when we traverse the edge from one platform to another at a certain station.
• The idea of an edge-weighted graph allows us to "charge for this". We'll put weight 1 on the ordinary edges connecting stations along a line, and 7 on the in-station transfer edges, and 3 on edges of the Silver line.
• So now we can ask for the least-cost route through this edge-weighted graph, using a different algorithm— which one??

With this setup, the cost of going two stops on the same line (top to bottom or vice versa, or left to right or vice versa) is 2, but the cost of going from the left to the bottom or top is 9, because of the estimated 7 minute wait of the transfer.

Pa4 part 4

• See pa4.html for a tour through the book looking for this algorithm.
• Finally find it: it's the famous Dijkstra's algorithm, implemented in the book for an edge-weighted digraph.
• But we have a plain directed graph, no weights...
• We're can convert our Digraph to an edge-weighted digraph, then do Dijkstra's algorithm.
• It's straightforward, since each edge of the digraph determines a weighted edge in the edge-weighted one.

Converting the Digraph to an EdgeWeightedDigraph

• Create dG with the same number of vertices as G, G.V().
• For each edge in G from vertex i to vertex j, add a directed edge in dG from vertex i to vertex j with weight w
• For our case, edge weight = 1 unless the edge connects two vertices in the same station, in which case it is 7, or on Silver line, in which case it is 3.
• Code for "connects two vertices in same station":
  pG.platformOf(i).getStation().getStationName().equals(pG.platformOf(j).getStation().getStationName()).

Using DijkstraSP

One execution of Dijkstra's algorithm gives you best routes to all other vertices in the graph: say from JFK/UMass:

DijkstraSP dep = new DijkstraSP(ewdg, umbVertex);
printRoute(pG, dep.pathTo(wonderlandVertex));

Iterable<DirectedEdge>

Note: I'll post a solution to pa4 on Sunday.