CS310 – Advanced Data Structures and Algorithms

Intro to Project 4: Graphs

Intro to pa4

- We’ll work with the S&W’s graph code, adopting their Graph, Digraph, and within the S&W code, uses of their Queue, Stack, etc.
- Our application will be finding the best routes through the Boston metro system, as an example of graph that’s well known and has some complexity, but not too much.
- Also it has a nice graphic...

The MBTA map, linked from pa4.html

The edges?

- We know the stations are connected together, and those connections will be our edges, undirected edges.
- So are stations the vertices?
- They can be, that’s the station graph, an undirected graph.
- (For parts 4 and 5, we’ll use another more detailed graph, a digraph)

What are the vertices?

- We see the stations, and the fact that some stations are on multiple lines

Park St. Station is on the Red and Green lines

What are the vertices?

- Our station graph has one vertex for each station, even a station on two train lines
- Example: Park Street station is on the Red line and the Green line
- Many edges connect stations on the same line together, creating the Red line, the Blue line, and so on.
- 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.
How do we build such a graph?

- We need data on all the stations
  - That's in the file bostonmetro.csv
- It has one line for each station-on-line, with its own number (starting from 1), the name of its line, the numbers of its two neighbors (or 0 if none), and its latitude and longitude.
- This is enough information to chain together all the stations on each line

Duplicate edges

- Note that we end up with lots of duplicate edges because if stations A and B are connected on a train line, we see A as a neighbor of B and B as a neighbor of A in the input file.
- So for a tidy graph we have to "dedup" the edges
- Deduplication methods:
  - Traditional way: Make an array and sort it, to bring dups together, then scan it and pick up one each of values
  - Set way: Drop them all into a Set that accepts only one of each set of dups.

Deduplication by Set

- For a Set to squeeze out dups, the dup elements must be equals() to each other, and the hashValue has to be consistent, thus also equal for dups (or compareTo, to use a TreeSet).
- Here the elements are edges, defined by pairs of int vertex ids. So we set up a SimpleEdge class (not "Edge", since used by text) with equals meaning the same int pair (either order), and hashValue the XOR of the two ints—see details in MetroSystem.java.
- After the set insertions, the code reinitializes the Graph, and connects up the smaller number of unduplicated edges.

Stations

- A client can get a Station object for each vertex. From main of MetroSystem, where mS is of type MetroSystem :
  ```java
  MetroSystem mS = new MetroSystem(args[0]);
  StationGraph stationGraph = mS.getStationGraph();
  Station s = stationGraph.stationOf(10);
  System.out.println("station 10: " + s.getStationName());
  ```
- This code prints:
  ```plaintext
  station 10: Alewife
  ```

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.
- This process is called "modeling".
Pa4 parts 1 and 2

- Although we have a beautiful diagram of the real system, we need to verify that our Graph follows it properly. So the first task is to 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 for various parts of the Green line, so just adopt these names here.

Pa4 part 3 unweighted BFS

- Once we are confident of the StationGraph setup, we want to use some of these powerful graph algorithms that are available in the S&W library algs4.jar.
- We'll use BFS to find the shortest paths through the graph
- That works fine but could give us a routes using three train lines where a slightly different route uses only two lines, which is clearly better when waiting time for a train is significant.
- Also it chooses routes with the Silver Line buses that would be faster using the real trains.
- So back to the drawing board for part 4...

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 walk from one platform to another at a certain station.
- We can make a graph of platforms and have more detail for accurately describing a person's journey through the system.
- Each station has platforms for each direction on each train line.
- Consider Park Street station and its neighbors again: Here is its basic platform graph...

Platform Graph with passageways

- Now we add passenger passageways connecting the platforms

MBTA's own map

- Central part of MBTA track map. Shows that the Green line tracks cross above the Red line tracks at Park St., and also above the Blue line tracks at Government Center. They were there first, apparently. The colored rectangles represent the platforms. So we see that the transfers between lines involve vertical movement, usually using stairs.
With this PlatformGraph 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.

Wait time -- edge cost

- When we change trains, we have to wait for another train to show up, say about 7 minutes.
- 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.
- Also we put weight 3 on Silverline edges, since it runs slower.
- So now we can ask for the least-cost route through this edge-weighted graph, using a different algorithm—which one??

Pa4 part 4

- See pa4.html for a tour through the book looking for this algorithm for shortest paths using weighted edges.
- Finally find it: it’s the famous Dijkstra’s algorithm, implemented in the book for a digraph.
- So far we have a digraph, but not a weighted digraph.
- We can convert our unweighted digraph to a weighted digraph, then do Dijkstra’s algorithm.
- No problem, it’s all very doable.

Pa4: the optional part

- There’s an optional part to think about
- We all (normally, anyway) use our phones to lead us around the city
- We get routes that usually work out great — how does this work?
- There’s a huge graph in the map service somewhere and Dijkstra is running, some descendant of it anyway, with edge costs that vary with traffic, etc.
- We could do an app that could find the best route to walk to a station, use the subway, and walk to the destination, in the case of no overload in the subway system and no spatial barriers to walking.

Pa4: getting started

- We need to use algs4.jar, the library of S&W code we’ve been avoiding so far.
- But the JDK doesn’t have graph support, so it’s great to have this resource.
- We used to use JGraphT, a well-known open-source Java graph package, but I think the S&W approach is easier for simple apps, lighter-weight (no class inheritance), but it has no visualization capability, a serious drawback.
- Both algs4 and JGraphT are supposed to be open source, with slightly different licenses. JGraphT has Javascript and Python versions as well as its original Java version. However, the book’s publisher recently required github to take down the algs4.jar sources.

Understanding jar files

- S&W’s code, including all the graph algorithms as well as Graph, Digraph, etc., is all in algs4.jar
- A jar file is a compressed collection of .class files, organized by packages
- Uncompressed, it’s just a directory system defined by the packages in the usual way, with .class files in the appropriate directories.
- When we add the library to our build commands, Java has access to all these classes, so won’t output “cannot find symbol ...” for things in them.
Understanding the classpath

- What we need to do is put the jar file "on the classpath"
- For running the program in src, we need, on Mac/Linux:
  `java -cp ../lib/algs4.jar: . MetroSystem ../bostonmetro.csv`
  
  This has put two things on the classpath, the library and ".", the current directory. If you leave off the "," java won't find MetroSystem. The "."/bostonmetro.csv is naming the input file, located in the base directory, one level up from src.
- For Windows, replace the : with ;
  
  `java -cp ../lib/algs4.jar; . MetroSystem ../bostonmetro.csv`
  
  Similarly, use StationGraph or PlatformGraph instead of MetroSystem here.

Summary on build commands with algs4.jar

- To compile on any OS: cd src, then
  
  `javac -cp ../lib/algs4.jar *.java`

- To run on Mac/Linux, use a colon between classpath parts, access file in base directory: cd src, then
  
  `java -cp ../lib/algs4.jar: . MetroSystem ../bostonmetro.csv`

- To run on Windows, use a semicolon between classpath parts, access file in base directory: cd src, then
  
  `java -cp ../lib/algs4.jar; . MetroSystem ../bostonmetro.csv`

  Similarly, use StationGraph or PlatformGraph instead of MetroSystem here.

For Help

- As usual, there is a topic “project 4” at Piazza
- You can use a private Piazza message or email elizabeth.oneil at umb.edu if you need to include code snippets.