Riding the Moo-tro System: A Study in Subway Congestion

Robert Chen, Alexander Nie, Annie Rak, Evan Yao

CS 51 Final Project
TF: Frederick Widjaja
May 1, 2015

Project Overview

We model the congestion dynamics of metropolitan subway systems for our CS 51 Final Project. To do this, we first represented networks of subway systems in Boston and Paris as graphs and then used shortest path algorithms to determine what route people using the subway would take. By aggregating the routes of all individuals, we obtain how busy different parts of the subway system would be. Upon making a visualization for this, we present a tool that can be used to analyze the crowdedness of a subway network and experiment with how changes to the network would impact congestion.

Because we created a user interface, running the code is simple; just choose an option from the menu and hit the corresponding keystroke.

Link to video: http://youtu.be/3ix55o5EkCI

Descriptions of Images

figure_1: Shows how a user can find the shortest path between his/her desired start and end locations. Yields the names of all stations along the route.

figure_2: Displays Boston congestion map.

figure_3: Displays Paris congestion map.

figure_4: Shows Boston with added edge between Harvard and Kenmore stations. Note that from this, we can see the new path doesn't get used much, and thus doesn't relieve much of the congestion elsewhere in the network.

figure_5: Shows Boston congestion map with congestion adjustment to edge weights. Note that the congestion levels near the center of the subway map change compared to figure_2.png as a result of the adjustment, because people begin taking different routes as a result of certain lines being overly congested.

Project Components:

Graph Class:

We originally planned to have a few extra classes, however ultimately we decided that it was more efficient and elegant to maintain much of our functionality within the

Graph class. This caused a slight amount of confusion between team members, since there were some unexpected functions at first within the Graph class; we also had trouble understanding what some of these functions were doing, but better documentation and better communication ultimately resolved these issues pretty quickly.

The Graph class initializes adjacency list/adjacency matrix representations of the graph and provides data structures to look up station names, info, weights, coordinates, indices, etc. It also has methods for adding routes/stations, deleting routes, and generating subway network maps and congestion level maps. Our configuration of the Graph class has methods to call the A-Star and Dijkstra's algorithms located in other files.

Obtaining/Inputting Data

The process of gathering subway network data was not much different from what we had expected when writing the specifications. It took some, but fairly limited, tedious labor to create a CSV file containing information about the station names/usage rates/coordinates and subway connections for Boston. We used quanthub.com (http://www.quanturb.com/data.html) to obtain similar data for Paris, where we found a Pajek file for the adjacency matrix of the Paris subway system, as well as entrance data. This information is in the CSV files. You can also look in the files to figure out the names of all the stations.

Dijkstra's Algorithm

Dijkstra's algorithm was initially used to calculate the shortest paths and distances between all points on the graph. Because Dijkstra's calculates the shortest path between a source and all other points upon each iteration, the algorithm was implemented as the class ShortestPathsDijkstra (shortest_paths.py) so that previously calculated shortest paths could be stored between actual calls to the algorithm. This design was then exploited to increase efficiency: rather than calling Dijkstra's for every pair of points, the ShortestPathsDijkstra class calculated all shortest paths upon initialization, stored paths and distances in arrays, and merely extracted the path information from these arrays upon user request. Since the path between every pair of points was eventually extracted, this amortized cost (calculate everything upfront rather than as needed like A*) meant Dijkstra's ran n times rather than nC2 for a map with n vertices. To further optimize for space efficiency, paths were stored using a prev_array, where the jth element of the ith list represented the next node a rider at station j would take to arrive at i. In the extract_path method of the ShortestPathsDijkstra object, this prev_array was then "unwound" by accessing element i,j, updating j, and adding the next node to the path until either j = i (reached destination) or j = None (no path or same point).

To calculate the shortest paths, ShortestPathsDijkstra's allDist method was usually called soon after initialization as a path_finder object in graph.py. allDist then iterated over every single node, calling Dijkstra's algorithm using the singleSourceDist method. An overview of the implementation follows. Upon each iteration, Dijkstra's pops the node with the shortest distance from the source in a priority queue

(self.data_structure) which tracks nodes on the edge of exploration. The function then explores nodes adjacent to this one (found in self.adj_list), adds them to the priority queue for exploration if a new shortest path to these nodes is found going through the popped node, and updates various arrays that store information about each node for quick lookup, such as dist (current shortest distance to source), searched (is the node in the priority queue), and prev (stores previous node on shortest path between a node and the source). (These were done to prevent linear searching over the priority queue to check whether nodes were). Upon total depletion of the queue, the algorithm has explored every accessible node from our source, and data is transferred to the matrices self.prev_array and self.dist_array (stores distance of shortest path between source and sink) and the algorithm repeated with the next source.

The minimum priority queue used by the singleSourceDist method is implemented in the data_structures.py class as a naive list (Priority) and a more efficient d-ary heap (DaryHeap). For space efficiency, the d-ary heap nodes were actually stored in a list, rather than as a series of linked objects. For a node at index n in the queue, its children were then given by *nd+1, nd+2...n(d+1),* thereby allowing faster manipulation on indices and storage of each node as simply (key, value). For both classes, self.data_structure represents the actual queue while self.indices stores the index of each node in the queue for fast lookup by key. Of note, an object-oriented design was useful not only for the instance variables, but also for the inheritance of decrease Key. For rebalancing, both classes include a push_up function in addition to the standard priority queue functions.

While Dijkstra's formed the core of the initial algorithmic portion of the technical spec, its design evolved slightly during implementation, usually for efficiency considerations. Most importantly, the idea of calculating all shortest paths up front rather than as needed yielded significant improvements as seen in the graph construction times between Dijkstra's and A*. Second, a prev-array was used rather than a dict to store the shortest paths between points. In addition to greatly saving on space, runtime efficiency improved as well, as lookup time for a prev array depends on path length (O(n)) rather than performing linear search on all pairs ($O(n^2)$). With respect to the data structures, the binary heap was extended to provide a polymorphic d-ary heap option. Although this marginally improved efficiency (most nodes in a subway have degree 2), this extension encouraged us to think about generalizations of the methods used on a binary heap.

By far the most challenging portion of the Dijkstra implementation was the d-ary heap, as it was very easy to botch index manipulation while popping and rebalancing the heap, usually with off-by-one and out of bounds errors. From debugging, we realized that using an implementation with pointers in a linked list probably would not have been as prone to error, although the ease of performing calculations on indices rather than pushing around pointers justified the tradeoff. In retrospect, implementing the naive priority queue was a bit superfluous, and had we more time, the Fibonacci heap would have been our next step.

A-Star Algorithm

The A* algorithm uses heuristics to improve the shortest-edge search process. Unlike Dijkstra's, each iteration of the algorithm only finds the shortest path between a single start-goal pair. The reason for this is that one of the core components of the search process, the heuristic, directly depends on the goal. The heuristic, assigned to each node, can be thought of as a sort of 'hint' as to how close that node is to the goal. The A* algorithm requires that the heuristic be admissible, meaning that it cannot overestimate the distance from that node to the goal. Our natural choice of heuristic was as-the-bird-flies straight-line geographical distance between stations.

Given a start node and an end node, the algorithm initializes a list of lists containing the first node. Each iteration begins by checking to see if the last element of the first list (path) is the goal, and if it is, that path is returned. If not, the algorithm selects the first list, finds the last element (node), and replaces this path with all the possible paths that are formed by extending the first path by one edge. If any two lists end at the same node, the longer one is deleted. Then, each list (path) in the list of paths is sorted in increasing order of a metric that combines the heuristic at the final node plus the length of the path thus far. Then, the algorithm repeats.

The A* algorithm was tested against the scipy shortest path feature.

Visualization, User Interface, Etc.

We used networkx to generate visually appealing graphs in Python. After calculating the congestion on each edge by running a shortest paths algorithm on all pairs of vertices, we individually add edges to the graph with a corresponding color based on the congestion of that edge. We also created a user interface with a menu displaying all options for our tool.

Then, in addition to having an option to just run the shortest path algorithms once and display the congestion map, we implemented a feature to make the edge weight depend on not only geographical distance like before, but also level of congestion. We did this by first obtaining a congestion map like before and then multiplying the existing edge weights by a transformation of the standardized Z-scores based on congestion, and then re-running the shortest path algorithm/congestion map. This "double-run" simulates the extra crowdedness in busy parts of the subway network that would delay travelers; we can then observe how these travelers change their routes as a result of the congestion. (To run this version, use "Run Simulation" instead of "Display a Congestion Map".)

Conclusion

In this project, we were able to divide work effectively and had few issues keeping all team members engaged and on the same page. Alex focused mostly on implementing Dijkstra's algorithm along with the different underlying data structures used in the algorithm. Annie implemented and tested the A-Star algorithm and obtained and formatted the Paris subway data. Robert worked mainly on setting up the Graph class, gathering and inputting Boston subway data, making edge weights depend on congestion in addition to distance, and helping with project checkpoints. Evan developed the visualizations for the congestion map and was the main

contributor to the project checkpoint documents. All members were involved in brainstorming ideas/features and doing the writeup/video.

If there were more time, we could consider expanding our project in two main directions. From an algorithms/theory standpoint, we could try adding additional algorithms (like Floyd-Warshall) or more data structures to Dijkstra's (like Fibonacci heaps), as well as run simulations to test the time efficiency of the different approaches; presumably, with the subway graphs, the results would be significantly different from respective asymptotic complexities. From a more practical/feature-oriented perspective, possibilities include loading more city subway maps and comparing different cities (such as Paris from the QuantUrb website), adding functionality to calculate different statistics about congestion and commute time, actually using our tool to give discover social implications or policy recommendations (ex. where to best construct new lines), or factoring in time of day or capacity of different subway routes (number of trains running along the line) into our calculations.

Although several members had used some Python in the past, we learned a lot more about the language through doing the project, especially about doing object-oriented programming (we had mainly run quick Python scripts before). Many of the concepts were analogous in Python as in OCaml, despite the large differences between the languages. Learning about the different shortest path algorithms was also insightful. The project process went quite smoothly, so most of the improvements we wish we had made were feature-specific and mentioned in the earlier section. We do, however, believe we could have spent a bit more time planning how different functions/classes would interact together; we occasionally had small deviations from the interfaces and signatures that required us to read through/reorganize other team members' code. We will keep in mind for future coding projects the high importance of planning ahead of time and then clearly communicating any changes from those plans that become necessary.

Here is the annotation of the final specification:

CS 51 Final Project Specification

Robert Chen, Alex Nie, Annie Rak, Evan Yao

Annotations in red.
We will be exploring expected congestion in subway networks using Python.

Included Files:

- Graph.py
- Shortest_Paths.py
- Data_Structures.py

Boston MBTA data
http://www.mbta.com/uploadedfiles/documents/2014%20BLUEBOOK%2014th%20Edition.pdf We did indeed use this website to obtain the Boston data and had few problems; it supplied most of the info we needed. We printed out the map and put a piece of graph

paper on top of it in order to estimate the coordinates of each station. We also looked into quanturb.com to find data about other cities like Paris.

Main Features:

- Given a starting and ending station, we will inform the user how congested each possible path they can take will be.
- Display graph of congestion, which takes into account how many shortest paths will require a particular segment.
  We did both of these and incorporated all options for our tool into a user interface.

Object Oriented Design Structure:

- Graph Representation:
  - *Class*: Graph – General transportation network graph
    - *Abstract Class*: ShortestPathAlg – adds shortest path methods features to superclass Graph.
      - *Class*: Dijkstra's
      - *Class*: A Star
        We maintained more or less the same structure, but just incorporated methods calling Dijkstra's and A-Star directly into Graph, rather than having an abstract class.

- Data Structure for Algorithms
  - *Abstract Class*: Dijkstra's Data Structure
    - *Class*: Priority Queue
    - *Class*: Binary Heap
    - *Class*: Fibonacci Heap
      Rather than doing a Fibonacci heap, we decided to implement a D-ary heap data structure as well as focus more on doing the A-Star algorithm and running rigorous tests for both our algorithms.

Overview of Graph Implementation

- Fields include:
  - Adjacency List
  - Adjacency Matrix
  - Station ID to Station Name dictionary
  - Station Name to Station ID dictionary
  - Number of Stations
    We used all these instance variables and more as necessary.
- Constructor:
  - Reads a graph from a txt file of the following format:

```
SAMPLE INPUT
3 ← Number of Stations
Harvard 40 ← Station Name, # of people entering
per minute
Central 20
Kendall 30
Harvard Central 2 ← Each Edge, Time needed
Central Harvard 2
Central Kendall 3
Kendall Central 3
```

We read from a CSV file instead and change formatting since it was easier to keep track of our data when there were explicit cells in Microsoft Excel. But more or less the same thing.

- Methods:
    - See actual code for documentation

Task Division:

- **Person 1**: Implementation of Graph Data Structure
    - Define methods in *Graph*
    - Look into displaying graph in a more visually appealing way
- **Person 2**: A Star Algorithm
    - Research A Star Algorithm and implement it
    - Define *ShortestPathsAStar*
- **Person 3**: Dijkstra's Algorithm
    - Implement at least 2 underlying Dijkstra's data structures such as *Priority* and *Heap*
    - Define methods in *ShortestPathsDijkstra*
- **Person 4**: Data Collection, Miscellaneous
    - Collect Data and format into necessary input structure
    - Test code that others have written
    - Help out with one of the above components
      This was indeed the way we originally split the work up for the first few days and then we communicated about how to best adjust how to split up work afterward.

Timeline

- Week 1 (by 4/24 at 11:59 PM)
    - Collect data on Boston's MBTA and convert into input format. Start finding data on other transportation networks too
    - Finish implementation of *Graph* with basic features not involving shortest paths
    - Finish researching Dijkstra's and start implementing some versions of its underlying data structure
    - **Goal:** Have a working version reading input, creating graph and finding shortest paths between any two stations.
- Week 2 (by 5/1 at 5:00 PM)
    - Create more complicated Dijkstra's Algorithm Data Structures.
    - Research A Star and implement it
    - Research more visually appealing ways of representing the heat map as well as the graph itself
    - Create presentation video
    - **Goal:** Improve on basic features and hopefully make everything more visually appealing!
      We followed this pretty closely. We also added additional features such as "second iteration" method to make edge weights a function of geographical distance + congestion level, getting data for Paris, being able to see effects of adding/deleting edges, and making a user-friendly interface.