

CSCI-2300: Data Structures and Algorithms

Project 3 — Shortest Paths in a Network

Due Date

The due date is **Friday, April 28, 2006 by 11:59:59 pm**. See the syllabus for late policies. See below for submission guidelines.

Introduction

Consider a data communication network that must route data packets (email or MP3 files, for example). Such a network consists of routers connected by physical cables or links. A router can act as a source, a destination, or a forwarder of data packets. We can model a network as a graph with each router corresponding to a vertex and the link or physical connection between two routers corresponding to a *pair of directed edges* between the vertices.

A network that follows the OSPF (Open Shortest Path First) protocol routes packets using Dijkstra's shortest path algorithm. The criteria used to compute the weight corresponding to a link can include the time taken for data transmission, reliability of the link, transmission cost, and available bandwidth. Typically each router has a complete representation of the network graph and associated information available to it.

For the purposes of this project, each link has associated with it the transmission time taken for data to get from the vertex at one end to the vertex at the other end. You will compute the best path using the criterion of minimizing the total time taken for data to reach the destination. The shortest time path minimizes the sum of the transmission times of the links along the path.

The network topology can change dynamically based on the state of the links and the routers. For example, a link may go down when the corresponding cable is cut, and a vertex may go down when the corresponding router crashes. In addition to these transient changes, changes to a network occur when a link is added or removed.

Example

Consider a very simplified example of a network at RPI, with vertices at Amos Eaton, Lally, DCC, Troy, Sage, and Folsom. See Figure 1 for an illustration. For this example, the shortest time (or fastest) path from Amos Eaton to DCC is: Amos Eaton–Troy–DCC with a total transmission time of 0.9 milliseconds.

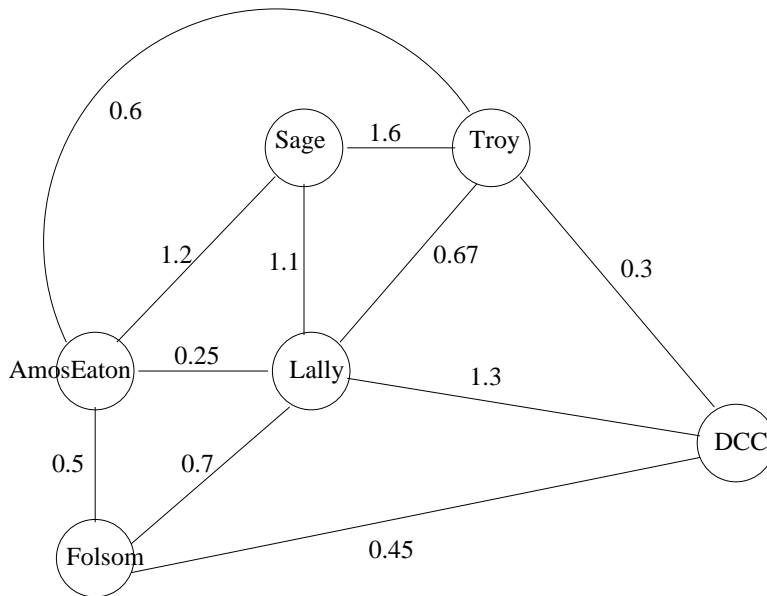


Figure 1: An example network graph. Each link shown represents two directed edges. Shown next to each link are the associated transmission times of its two edges in milliseconds.

Your Job

You have five tasks in this project: building the initial graph, updating the graph to reflect changes, finding the shortest path between any two vertices in the graph based on its current state, printing the graph, and finding reachable sets of vertices. These are described in turn.

Building the Graph

Your program should run from the command-line:

```
graph network.txt
```

where `network.txt` is a file that contains the initial state of the graph. (Other names besides `network.txt` may be used, so you cannot hard-code this.) The format of the file is quite simple. Each link representing two directed edges is listed on a line, and is specified by the names of its two vertices followed by the transmission time. Vertices are simply strings (vertex names with no spaces) and the transmission times are floating point numbers. For the above example, the file will look like:

```
AmosEaton Sage 1.2
AmosEaton Folsom 0.5
Troy AmosEaton 0.6
AmosEaton Lally 0.25
Lally Sage 1.1
Sage Troy 1.6
Folsom Lally 0.7
```

Folsom DCC 0.45
Lally DCC 1.3
Troy DCC 0.3
Lally Troy 0.67

For each input link, add **two** directed edges, one in each direction.

Changes to the Graph

Input about changes to the graph, requests for shortest paths, and requests to print the graph will come as input queries to be read from `cin`. Here are queries indicating changes to the graph.

addege `tailvertex headvertex transmit_time` — Add a single directed edge from `tailvertex` to `headvertex`. Here `headvertex` and `tailvertex` are the names of the vertices and `transmit_time` is a float specifying the transmission time of the edge. If a vertex does not exist in the graph already, add it to the graph. If the edge is already in the graph, change its transmission time. **Important note:** This can enable two vertices to be connected by a directed edge in one direction and not the other, or enable the transmission times in the two directions to be different.

deleteedge `tailvertex headvertex` — Delete the specified directed edge from the graph. Do not remove the vertices. If the edge does not exist, do nothing. **Note:** This may cause two vertices to be connected by a directed edge in one direction, but not the other.

edgedown `tailvertex headvertex` — Mark the directed edge as “down” and therefore unavailable for use. The response to an **edgedown** (or **edgeup**) query should mark only the specified directed edge as “down” (or “up”). So its companion directed edge (the edge going in the other direction) should not be affected.

edgeup `tailvertex headvertex` — Mark the directed edge as “up”, and available for use. Its previous transmission time should be used.

vertexdown `vertex` — Mark the vertex as “down”. None of its edges can be used. Marking a vertex as “down” should not cause any of its incoming or outgoing edges to be marked as “down”. So the graph can have “up” edges going to and leaving from a “down” vertex. However a path through such a “down” vertex cannot be used as a valid path.

vertexup `vertex` — Mark the vertex as “up” again.

Finding the Shortest Path

The query for finding the shortest path will look like

```
path from_vertex to_vertex
```

where `from_vertex` and `to_vertex` are names of vertices. This should compute the shortest time path from `from_vertex` to `to_vertex` using Dijkstra’s algorithm and based on the current state of the graph.

The most difficult task is the implementation of Dijkstra’s algorithm, and especially the priority queue (binary heap). In the textbook, Weiss describes two ideas. Here is an outline of the first.

- The binary heap must store pointers to vertices in the graph as well as the path distance.
- It must use a “percolate up” function to move a vertex in the binary heap after its path distance is decreased.
- Each vertex must store the index of its location in the binary heap.
- The author’s binary heap code must be altered slightly. It is not allowable to have a pre-determined heap size.

See the text for the second method for using a binary heap. You must implement one of these to obtain full credit for the project.

The output must be the vertices along the shortest path, followed by the total transmission time. For example, with our example graph and no changes to the state of the graph, the query

```
path AmosEaton DCC
```

should yield as output

```
AmosEaton Troy DCC 0.9
```

Print Graph

The simple query

```
print
```

must print the contents of the graph. Vertices must be printed in alphabetical order and the outward edge for each vertex must be printed in alphabetical order. For the example graph, the output should be

```
AmosEaton
  Folsom 0.5
  Lally 0.25
  Sage 1.2
  Troy 0.6
DCC
  Folsom 0.45
  Lally 1.3
  Troy 0.3
Folsom
  AmosEaton 0.5
  DCC 0.45
  Lally 0.7
```

```
Lally
  AmosEaton 0.25
  DCC 1.3
  Folsom 0.7
  Sage 1.1
  Troy 0.67
Sage
...
Troy
...
```

When a vertex is down, append the word **DOWN** after the vertex. When an edge is down, append the word **DOWN** after the edge.

Reachable Vertices

Since the states of the vertices and edges may change with time, it is useful to know the set of vertices reachable from each vertex by valid paths. You must develop and implement an algorithm that identifies for each vertex, the set of all its reachable vertices. **Important:** Additionally, you must describe your algorithm and justify its running time, using “O” notation, in the comments for this algorithm in your submitted code.

The query

```
reachable
```

must print for each “up” vertex, all vertices that are reachable from it. So vertices that are “down” or not reachable by “up” edges will not be printed. Vertices must be printed in alphabetical order, and the set of vertices reachable from each vertex must be printed in alphabetical order. For example, if only vertices **Folsom** and **Sage** are reachable from vertex **AmosEaton**, the output will look like

```
AmosEaton
  Folsom
  Sage
Folsom
...
```

Quit

The input query **quit** should simply cause the program to exit without printing anything.

Project Hints

The implementation of Dijkstra’s algorithm is perhaps the hardest part of the project. You can start from the graph code (Appendix A.7) and binary heap code provided from the textbook, which

we have posted on the course web page. You will need to modify this in several ways, including the addition of a simple Edge class.

Remember that the initial information about the network graph will come from a file specified as a command-line argument. The graph you create will be a directed graph with two edges, one in each direction, for each input link. The queries will come from `cin` and the output from your program should go to `cout`. Example input and output files will be posted on the course web page.

Grading Criteria

The total of 100 points for this project is broken up into:

- 20 points for proper construction of the graph and outputting a correct description.
- 35 points for efficiently finding shortest paths using Dijkstra's algorithm. You will lose 15 points if you do not use a binary heap to implement the priority queue.
- 15 points for updating the graph based on network changes.
- 15 points for printing all reachable vertex sets and analyzing the complexity of your algorithm.
- 15 points for compilation, structure, and documentation.

Within these criteria, your grade will depend on program structure, efficiency, and correct execution.

Your program must compile using VC++.NET or g++ (version 2.9.5 or higher, check using `g++ --version`). The structure of your code will be judged for quality of the comments, quality of the data structure design, and especially the logic of the implementation. The comments need not be extremely long: just explain clearly the purpose of each class and each function within each class.

Submission Guidelines

Your submission must include all your source code (`.h` and `.cpp` suffix), including any of the provided textbook code that you use, and a brief report as a `readme.txt` file. Your submission should NOT include any files with a `.exe`, `.dsp`, `.dsw`, `.ncb`, or `.opt` suffix. Every file should have your name in a comment line at the top. Your `readme.txt` file should have a brief description of your program design, the breakdown of the files, which compiler you used (VC++ .NET will be taken as the default), a summary of what you think works and fails in your program, **and** a short description of your data structure design.

You will follow the web-based submission procedure used for Project 2; details will be posted on the project web page. Briefly, you should submit a single zip, tar, or gzip file containing your source code files and `readme.txt` file. You can submit your project multiple times; only the most recent project submission will be graded. The most recent project submission will also be used to compute late days, if any.

Final Warning

A project that does not follow the submission guidelines will receive a 10 point deduction. Proper submission is entirely **your responsibility**. Contact your TA if you have any

doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to keep copies of your files and **do not change them after submitting**. Don't even copy them. It is a good idea to keep your submission files in a private (protected) directory on RCS to have a valid timestamp. After grades are posted, you have exactly one week to resolve all problems. After that week is up, all grades are final.