# Graph Algorithms*

Due Wednesday Dec 9, 2015

## Introduction

As residents of southern California, most of us face the realities of having to drive or ride from one place to another on streets and freeways. Given that this is a heavily populated area, we also have to contend with traffic. If we attempt to drive on a local freeway during rush hour, we often experience traffic jams and long delays, requiring us to find alternative routes or simply put up with the traffic and wait.

Fortunately, technology offers at least some assistance. With ubiquitous wireless Internet connections, powerful devices embedded into cars and available in a mobile form, we have easy access to information that can help. Aside from providing the obvious ability to download traffic reports and maps on demand, these devices can go a step further; given up-to-the-minute traffic information and with a little computing power, your device can actively aid you in finding the best way to get from one place to another, optimized not only for distance, but also for the shortest driving time given the current traffic conditions. Further, if all cars used such a system, as drivers were diverted around the scene of an accident, traffic conditions would change, and the advice offered by drivers' in-car or mobile devices would also change, optimally routing cars around a traffic jam by sending different cars down different alternate paths. This way, even the alternatives might flow as quickly as possible. (And, taking things a step further, Google has made a lot of recent progress on self-driving cars, which can optimize the traffic problem even further.)

This is hardly science fiction; many of these features are already available on smartphones and in-car systems. And while there are a lot of different kinds of technology you need in order to build a system like this, the core of the problem is actually one that's familiar to us in this course. For this project, you will write a simplified version of an important piece of such a system: given a map of streets and freeways, along with a snapshot of the current traffic between points on the map, your program will be capable of finding the shortest distance or fastest route to get from one location on the map to another.

## Getting Started

You will find the starter code for this project in the usual place. You have been provided a fair amount of pre-written code; after reading through the project write-up, take a look through the provided code and be sure you understand what problems it solves, so you can understand what parts of the problem you'll need to solve yourself.

## Our view of a street map

Real-life street maps, such as those you see online like Google Maps or those displayed by navigation systems in cars, are a handy way for people to determine an appropriate route to take from one location to another. They present an abstraction of the world as a scaled-down drawing of the actual streets. In order to be useful to us, a street map needs to give us the names of streets and freeways, to accurately demonstrate distances and directions, and to show us where the various streets and freeways intersect.

For our program, we'll need to develop a different abstraction of a street map. Our abstraction must contain the information that is pertinent to the problem we're trying to solve, presented in a way that will make it as easy as possible for our program to solve it. Not surprisingly, a picture made up of lines and words is not an abstract that is useful to our program; it would require a tremendous amount of effort to design and implement an algorithm to interpret the lines and words and build up some alternative representation that's more convenient. It's better that we first design the more convenient representation, then train our program to read and understand an input file that specifies it. To do so, we'll need to consider the problem a bit further.

*Originally written by Alex Thornton at UCI, Spring 2014 for ICS 62(C++), with heavy influence from a similarly-named project from ICS 23(Java). Modified by Michael J. Bannister at Pomona College for CS 62(C).

Our program's main job is to discover the shortest distance or driving time between two locations. There's no reason we couldn't think of locations as being any particular point on a street map (for example, any valid street address, or even any valid GPS coordinate). For simplicity, though, we'll think of them as points on the map in which decisions would need to be made, such as:

- The intersection of two or more streets
- A point of a freeway at which there is an entrance and/or an exit

Connecting pairs of locations on the map are stretches of road. In order to solve our problem, we'll need to know two things about each stretch of road:

- Its length, in miles
- The current speed of traffic traveling on it, in miles per hour

Our map will consist of any kind of road that a car can pass over (e.g., streets or freeways), though it will not make a clear distinction between them. In general, all of these are simply stretches of road that travel in a single direction. For example, you could think of a simple two-lane street as a sequence of intersections, connected by stretches of road of equal length running in opposite directions; note, though, that the speed of traffic on either side of the street might be different.

In real life, many intersections control traffic using stop signs or traffic lights. Our program will ignore these controls; we'll instead assume that the traffic speeds on streets have been adjusted appropriately downward to account for the average time spent waiting at stop signs and lights.

Also, to keep the problem relatively simple, absolute directions (i.e., north, south, east, and west) will not be considered by our program or reported in its output. For that reason, they won't be included in our abstraction of a street map, except optionally in the names of locations.

The output of our program will be a trip. A trip is a sequence of visits to locations on the map. For example, a typical trip around Irvine, CA would looked like this:

- Begin at Peltason & Los Trancos
- Continue to Bison & Peltason Continue to Bison & California
- Continue to Bison & 73N on-ramp
- Continue to 73N @ Birch
- Continue to 73N @ 73N-to-55N transition
- Continue to 55N @ Baker
- Continue to 55N Baker/Paularino off-ramp & Baker
- Continue to Baker & Bristol

In addition to the information above, your program will also output information about the distance in miles and (sometimes) driving time of each of the segments of the trip, as well as the overall distance and (sometimes) driving time for the whole trip.

## Representing our view of a street map

If you consider all of the data that we'll need to represent this abstraction, the task of organizing it can seem overwhelming. However, there is a well-known data structure that represents this system in a straightforward way: a directed graph. Using a directed graph, locations on our map can be represented as vertices, and the stretches of road connecting locations can be represented as edges. (Since traffic travels in only one direction on a given stretch of road, it makes good sense that the graph should be directed.)

Each vertex in the graph will have a human-readable name for the location it represents. For example, a vertex might be named Culver & Harvard, or it might be named I-405N @ Jamboree. The name will be used only for display purposes; it won't have any significance in our algorithm. The vertices should be numbered uniquely and consecutively, starting at zero. If there are $n$ vertices, they should be numbered $0, 1, \ldots, n-1$.

Each edge will be associated with two necessary pieces of information about the stretch of road it represents: the distance between the two vertices (in miles, stored as a double) and the current speed of traffic (in miles per hour, also stored as a double).

Since a trip is a sequence of visits to adjacent locations on the map, locations are represented as vertices, and two locations are adjacent only when there is a stretch of road (i.e., an edge) connecting them, a trip can be represented as a path in the graph. So our main goal is to implement a kind of shortest path algorithm.

## The program

The goal of your program is to read a file containing a description of all of the locations on a map and the stretches of road that connect them. It then performs two tasks:

1. Ensures that it is possible for every location to be reached from every other location. If we think of the locations and roads as a directed graph, that boils down to the problem of determining whether the graph is strongly connected. If not, the message "Disconnected Map" should be output and the program should end.

2. Determines, for a sequence of trip requests listed in the input, shortest distances or shortest times between pairs of locations.

Check out the sample input, which you'll find in the data directory in your project directory. A description of its format follows.

The input is separated into three sections: the locations, the road segments connecting them, and the trips to be analyzed. Blank lines (and, similarly, lines containing only spaces) should be ignored. Lines beginning with a # character indicate comments and should likewise be ignored. This allows the input to be formatted and commented, for readability.

The first section of the input defines the names of the map locations. First is a line that specifies the number of locations. If there are $n$ locations, the next $n$ lines of the input (not counting blank lines or comments) will contain the names of each location. The locations will be stored in order in an array of structs each containing a vertex id and string for the name. Each vertex is to be given a number, with the numbers assigned consecutively in the order their names appeared in the input, starting at 0.

The next section of the input defines the road segments. Each road segment will be an edge in the directed graph. The first line of this section specifies the number of segments. Following that are the appropriate number of road segment definitions, with each segment defined on a line with four values on it:

1. The vertex number where the segment begins

2. The vertex number where the segment ends

3. The distance covered by the segment, in miles

4. The current speed of the traffic on the segment, in miles per hour

Each road segment will be stored in a struct containing these four values, and the collection of all road segment will initially be stored in an array of such structs.

Finally, the desired trips are defined. Again, the section begins with a line specifying the number of trips. Following that are an appropriate number of trip requests, with each trip request appearing on a line with three values on it:

**1**. The starting location for the trip

**2**. The ending location for the trip

**3**. `D` if the program should determine the shortest distance, or `T` if the program should determine the shortest driving time

As with the last two sections trips will be stored as structs containing their three defining values, and the collection of all trips will be stored in an array.

Your program should read the vertices and edges from the input, build a graph (or multiple graphs), then process the trip requests in the order that they appear. The output for each trip request is described later in this write-up.

You may assume that the input will be formatted according to the rules described above, but you may not assume that the input we'll use to test the program will be identical to the sample. Different numbers of vertices, different configurations of edges, different names, difference distances and speeds, etc., are possible. *You may assume no line in the input will be more than 100 characters, and that no input will have more than 1000 vertices.*

## Implementing your directed graph

A directed graph implementation is provided for you in the files `graph.h` and `graph.c`. You may feel free to modify this implementation as you like. However, if you modify the graph make sure you do not introduce memory leaks.

## Finding the shortest paths

The problem we need to solve, that of finding the fastest or shortest trip along a network of roads, is not an uncommon one in computing. In fact, it's so common that it's already been solved abstractly. Our problem is an instance of the single-source, positive-weighted, shortest-path problem. In other words, from one particular vertex (a "single source"), we'll be finding the shortest path to another vertex, where all of the edges have a "positive weight" (in our case, distance or speed, neither of which will ever be negative or zero) associated with them. We'll use a well-known algorithm called Dijkstra's Shortest-Path Algorithm to solve this problem. You will want to use the version Dijkstra Algorithm described in class, as it is the simplest to implement and often the fastest. You will need a priority queue to implement Dijkstra Algorithm. A priority queue has been provided for you and you coded one as a solution to a previous assignment. You may use either for this assignment.

## The output

For each of the trip requests in the input file, your program should output a neatly-formatted report to the console that includes each leg of the trip with its distance and/or time (as appropriate), and the total distance and/or time for the trip.

If the trip request asks for the shortest distance, the output might look something like the following. (These are phony trips, to show you the output format; they are not related to the sample data file provided above.)

```
Shortest distance from Alton & Jamboree to MacArthur & Main
  Begin at Alton & Jamboree
  Continue to Main & Jamboree (1.1 miles)
  Continue to Jamboree & I-405N on ramp (0.3 miles)
  Continue to I-405N @ MacArthur (1.3 miles)
  Continue to MacArthur & I-405N off ramp (0.1 miles)
  Continue to MacArthur & Main (0.2 miles)
Total distance: 3.0 miles
```

On the other hand, if the trip request asks for the shortest time, the output might look like this:

```
Shortest driving time from Alton & Jamboree to MacArthur & Main
  Begin at Alton & Jamboree
  Continue to Alton & MacArthur (2.7 miles @ 33.7mph = 4 mins 48.8 secs)
  Continue to Main & MacArthur (1.1 miles @ 40.1mph = 1 min 38.7 secs)
Total time: 6 mins 27.5 secs
```

When outputting a time, you should separate it into its components—hours, minutes, and seconds—as appropriate. Here are some examples:

```
32.5 secs
2 mins 27.8 secs
13 mins 0.0 secs
3 hrs 13 mins 12.3 secs
6 hrs 0 mins 0.0 secs
```

Don't show hours if there are zero of them. Don't show hours or minutes if there are zero of both of them.

## The provided code

A fair amount of code has been provided already in your project directory, so some of what is described above is actually already implemented. In particular, you'll find the following has been implemented: all input parsing, an abstract graph and a priority queue. Any of the provided code can be replaced or removed; it's up to you. This code is provided primarily to let you focus on the interesting parts of the project, without spending too much time on grunt work.

## Grading

| criterion | points |
|---|---|
| properly building graph from input data | 2 |
| strong connectivity test | 4 |
| Dijkstra algorithm | 6 |
| proper handling on input | 2 |
| output properly formatted | 2 |
| appropriate comments (including JavaDoc) | 2 |
| style and formatting | 2 |
| submitted correctly | 1 |

## Submitting Your Work

Submit your three files in a folder named "Assignment12_LastNameFirstName". Make sure you and your partner's name are at the top of every file. Make sure your code follows the specifications above *exactly* and that your files are well-documented in the usual way with Javadoc comments, etc.