COMP2521 20T2 Data Structures and Algorithms

Week 07 Lab Exercise

Weighted Graphs and Geo Data

Objectives

- to implement a variant of path finding in a weighted graph
- to see how graphs might be used with real-world data

Admin

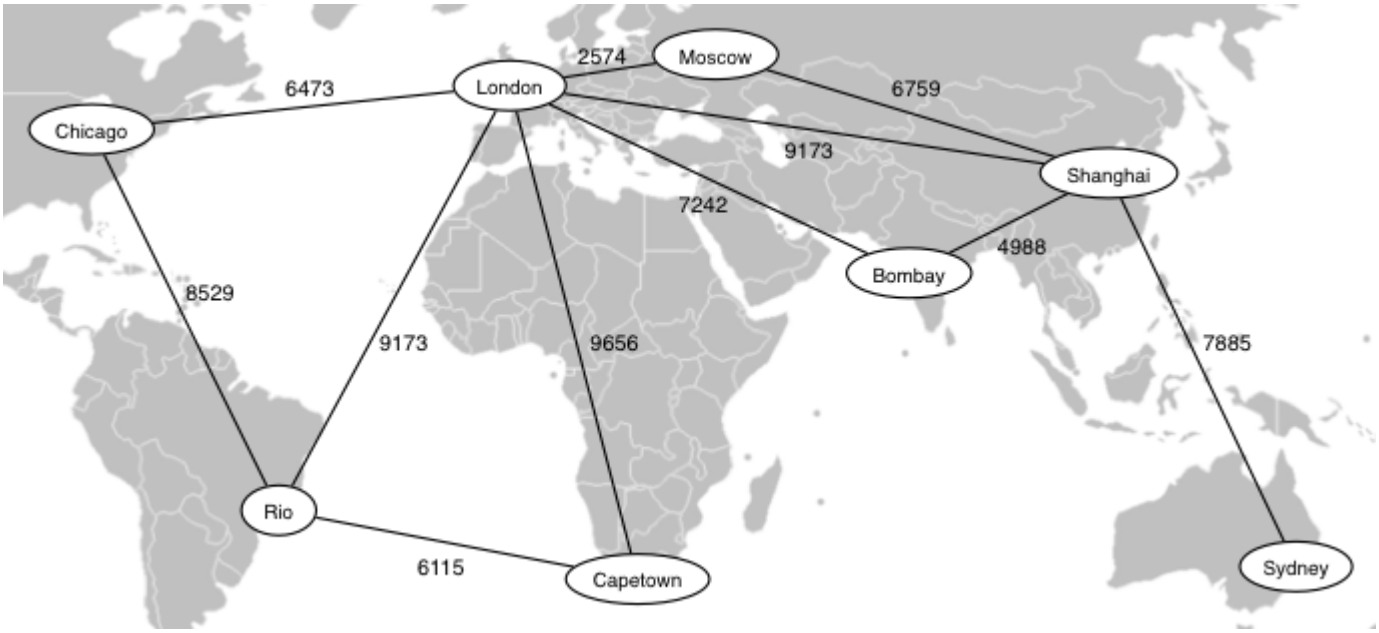
Marks	5=outstanding, 4=very good, 3=adequate, 2=sub-standard, 1=hopeless
Demo	in the Week 08 Lab
Submit	`give cs2521 lab07 Graph.c` or via WebCMS
Deadline	submit by Sun 19 July 2020, 23:59:00

Background

Geographic data is widely available, thanks to sites such as GeoNames. For this lab, we have downloaded data files from City Distance Dataset by John Burkardt in the Department of Scientific Computing at Florida Statue University. The dataset that we will use contains information about distances between 30 prominent cities/locations around the world. It measures "great circle" distances; we'll assume that these measure the distances that an aircraft might fly between the two cities. The data that we have forms a complete graph in that there is a distance recorded for every pair of cities.

While there may not be much scope for international travel at the moment, we can look forward to when we can travel freely again, and start planning our journeys now.

The following diagram shows a subset of the data from the City Distance Dataset.



Map from "BlankMap-World-v2" by original uploader: Roke

Own work. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons

The data comes in two files:

`ha30_dist.txt`

> This file contains a matrix of distances between cities. This is essentially the adjacency matrix for a weighted graph where the vertices are cities and the edge weights correspond to distances between cities. As you would expect for an adjacency matrix, the leading diagonal contains all zeroes (in this case, corresponding to the fact that a city is zero distance from itself).

`ha30_name.txt`

> This file contains one city name per line. If we number the lines starting from zero, then the line number corresponds to the vertex number for the city on that line. For example, the Azores is on line 0, so vertex 0 corresponds to the Azores, and the first line in the distance file gives distances from the Azores to the other 29 cities. The last line (line 29) tells us that Tokyo is vertex 29, and the last line in the distance files gives distances between Tokyo and all other cities.

Setting Up

Set up a directory for this lab under your COMP2521 labs directory, change into that directory, and run the following command:

```
$ unzip /web/cs2521/20T2/labs/week07/lab.zip
```

If you're working at home, download `lab.zip` and then work on your local machine.

If you've done the above correctly, you should now find the following files in the directory:

`Makefile`	a set of dependencies used to control compilation
`travel.c`	main program to load and manipulate the graph

`Graph.h`	interface to Graph ADT
`Graph.c`	implementation of Graph ADT
`Queue.h`	interface to Queue ADT
`Queue.c`	implementation of Queue ADT
`Item.h`	definition of Items (Edges)
`ha30_name.txt`	the city name file described above
`ha30_dist.txt`	the distance matrix file described above

The `Makefile` produces a file called `travel` based on the main program in `travel.c` and the functions in `Graph.c`. The `travel` program takes either zero or two command line arguments.

```
$ ./travel
... displays the entire graph ...
... produces lots of output, so either redirect to a file or use less ...
$ ./travel from-city to-city
... display a route to fly between specified cities ...
```

If given zero arguments, it simply displays the graph (in the format described below). If given two arguments, it treats the first city as a starting point and the second city as a destination, and determines a route between the two cities, based on "hops" between cities with direct flights.

Read the `main()` function so that you understand how it works, and, in particular, how it invokes the functions that you need to implement.

The `Graph` ADT in this week's lab has a `GraphRep` data structure that is a standard adjacency matrix representation of the kind we looked at in lectures. However, some aspects of it are different to the `GraphRep` from lectures.

Note that city names are not stored as part of the `GraphRep` data structure. The `Graph` ADT deals with vertices using their numeric id. The `main` program maintains the list of city names and passes this list to the `showGraph()` function when it is called to display the graph. This means that the calling interface for the `showGraph()` function is different to the `showGraph()` function from the `Graph` ADT in lectures.

Another difference between this `Graph` ADT and the one used in lectures is that the values stored in the matrix are not simply zero and one, but represent the distances between vertices. In other words, we're dealing with a *weighted* graph.

The `Graph` ADT includes a sub-ADT for `Edges`. The implementation of `insertEdge()` in lectures only required the vertices at the endpoints of the `Edge` (i.e. `insertEdge(g,v,w)`). The version of `insertEdge()` for this lab also requires a weight for the edge (i.e. `insertEdge(g,v,w,weight)`).

The main program makes some changes to the edges implied by the distance matrix as it copies them into the `Graph`. The values in the `ha30_dist.txt` file are given in units of "hundreds of miles"; we want them in units of kilometres so each distance is converted before it is added to the graph as the weight of an edge. Since every city has a distance to every other city (except itself), this gives us a *complete graph*.

As supplied, the `Graph.c` file is missing implementations for the `findPath()` function. If you compile the `travel` program and try to find any route, it will simply tell you that there isn't one. If you run `travel` with no arguments, it will print a representation of the graph (you can see what this should look like in the file graph.txt).

Exercise

Implement the `findPath(g,src,dest,max,path)` function. This function takes a graph `g`, two vertex numbers `src` and `dest`, a maximum flight distance, and fills the `path` array with a sequence of vertex numbers giving the "shortest" path from `src` to `dest` where no edge on the path is longer than `max`. The function returns the number of vertices stored in the `path` array; if it cannot find a path, it returns zero. The `path` array is assumed to have enough space to hold the longest possible path (which would include all vertices).

This could be solved with a standard BFS graph traversal algorithm, but there are two twists for this application:

- The edges in the graph represent real distances, but the user of the `travel` program (the traveller) isn't necessarily worried about real distances. They are more worried about the number of take-offs and landings (which they find scary), so the length of a path is measured in terms of the number of edges, *not* the sum of the edge weights. Thus, the "shortest" path is the one with the minimum number of edges.
- While the traveller isn't worried about how far a single flight goes, aircraft *are* affected by this (e.g. they run out of fuel). The `max` parameter for `findPath()` allows a user to specify that they only want to consider flights whose length is at most `max` kilometres (i.e. only edges whose weight is not more than `max`).

Your implementation of `findPath()` must satisfy both of the above.

In implementing `findPath()`, you can make use of the `Queue` ADT that we've supplied. This will create a queue of `Vertex` numbers.

Note that the default value for `max`, set in the `main` program is 10000 km. Making the maximum flight distance smaller produces more interesting paths (see below), but if you make it too small (e.g. 5000km) you end up isolating Australia from the rest of the world. With maximum flights of 6000km, the only way out of Australia in this data is via Guam. If you make the maximum flight length large enough (e.g. aircraft technology improves significantly), every city will be reachable from every other city in a single hop.

Some example routes (don't expect them to closely match reality):

```
# when no max distance is given on the command line,
# we assume that planes can fly up to 10000km before refuelling
$ ./travel Berlin Chicago
Least-hops route:
Berlin
->Chicago
$ ./travel Sydney Chicago
```

```
Least-hops route:
Sydney
->Honolulu
->Chicago
$ ./travel Sydney London
Least-hops route:
Sydney
->Shanghai
->London
$ ./travel London Sydney
Least-hops route:
London
->Shanghai
->Sydney
$ ./travel Sydney 'Buenos Aires'
Least-hops route:
Sydney
->Honolulu
->Chicago
->Buenos Aires
# if no plane can fly more than 6000km without refuelling
$ ./travel Sydney London 6000
Least-hops route:
Sydney
->Guam
->Manila
->Bombay
->Baghdad
->London
# if no plane can fly more than 5000km without refuelling
# you can't leave Australia under this scenario
$ ./travel Sydney 'Buenos Aires' 5000
No route from Sydney to Buenos Aires
# if no plane can fly more than 7000km without refuelling
$ ./travel Sydney 'Buenos Aires' 7000
Least-hops route:
Sydney
->Guam
->Honolulu
->Chicago
->Panama City
->Buenos Aires
# planes can fly up to 8000km without refuelling
$ ./travel Sydney 'Buenos Aires' 8000
Least-hops route:
Sydney
->Guam
->Honolulu
->Mexico City
```

```
->Buenos Aires
# planes can fly up to 11000km without refuelling
$ ./travel Sydney 'Buenos Aires' 11000
Least-hops route:
Sydney
->Bombay
->Azores
->Buenos Aires
# planes can fly up to 12000km without refuelling
# can reach Buenos Aires on a single flight
$ ./travel Sydney 'Buenos Aires' 12000
Least-hops route:
Sydney
->Buenos Aires
$ ./travel Bombay Chicago 5000
Least-hops route:
Bombay
->Istanbul
->Azores
->Montreal
->Chicago
$ ./travel Sydney Sydney
Least-hops route:
Sydney
```

The above routes were generated using an algorithm that checked vertices in order (vertex 0 before vertex 1 before vertex 2, etc.). If you check in a different order, you may generate different, but possibly equally valid, routes.

Demonstration/Submission

You can use Help Sessions to get help for this lab. The submission deadline is midnight on the Sunday at the start of Week 08 (July 19). Once you have completed the lab, you should demonstrate your work to your tutor, either in the Week07 lab (if you work quickly) or in the Week 08 lab.

Have fun, *jas*

COMP2521 20T2: Data Structures and Algorithms is brought to you by
the School of Computer Science and Engineering at the University of New South Wales, Sydney.
For all enquiries, please email the class account at cs2521@cse.unsw.edu.au
CRICOS Provider 00098G