CS 1501 Algorithm Implementation
Programming Project 5

Purpose: The purpose of this assignment is to give you practice implementing some simple graph algorithms and to see how they can be used in a somewhat practical way.

Procedure: You are to implement a simple information program for a fictional airline. Your program should be designed to be accessed by employees of the company, who may pass some of the information on to customers when needed. Details of the assignment follow below.

Initially, you need to load from a file (whose name was input by the user) a list of all of the service routes that your airline runs. These routes include the cities served and the various **non-stop** destinations from each city. Clearly, you will be interpreting these routes as a graph with the cities being the vertices and the non-stop trips being the edges. To keep things simple, assume that all routes are bidirectional, so that you can use an undirected graph (this is not necessarily an incorrect assumption, as airlines most often fly non-stop routes in both directions). Alternatively, you could use a directed graph, with a link in each direction for each trip. You can think of these as the current active routes, which would be updated periodically by some other program (in case a route is cancelled or perhaps snow closes an airport somewhere). The routes (edges) should have 2 different weights: one weight based on the **distance** between the cities and the other based on the **price** of a ticket between the cities.

- Your program should be able to handle the following queries:
  1) Show the entire list of direct routes, distances and prices. This amounts to outputting the entire graph (you do not need to use graphics) in a well-formatted way. Note that this operation does not require any sophisticated algorithms to implement.
  2) Display (not necessary to use graphics -- points and edges are ok) a minimum spanning tree for the service routes based on **distances**. This could be useful for maintenance routes and/or shipping supplies from a central supply location to all of the airports. If the route graph is not connected, this query should identify and show each of the connected subtrees of the graph.
  3) Allow for each of the three "shortest path" searches below. For each search, the user should be allowed to enter the source and destination cites (names, not numbers) and the output should be the cities in the path (starting at the source and ending at the destination), the "cost" of each link in the path and the overall "cost" of the entire trip. If multiple paths "tie" for the shortest, **you only need to print one out** (see extra credit options). If there is no path between the source and destination, the program should indicate that fact.
     a) Shortest path based on **total miles** (one way) from the source to the destination. Assuming distance and time are directly related, this could be useful to passengers who are in a hurry. It would also appeal to passengers who want to limit their carbon footprints.
     b) Shortest path based on **price** from the source to the destination. This option is a bit naïve, since prices are not necessarily additive for hops on a multi-city flight. However, to keep the algorithm fairly simple, you should assume the prices ARE additive. Since distance and price do NOT always correspond, this could be useful to passengers who want to save money.
     c) Shortest path based on **number of hops** (individual segments) from the source to the destination. This option could be useful to passengers who prefer fewer segments for one reason or other (ex: traveling with small children).
  4) Given a dollar amount entered by the user, print out **all trips** whose cost is less than or equal to that amount. In this case, a trip can contain an arbitrary number of hops (but it should not repeat any cities – i.e. it cannot contain a cycle). This feature would be useful for the airline to print out weekly "super saver" fare advertisements. Be careful to implement this option as efficiently as possible, since it has the possibility of having an exponential run-time (especially for long paths). Consider a recursive / backtracking / pruning approach.
  5) Add a new route to the schedule. Assume that both cities already exist, and the user enters the vertices, distance and price for the new route. Clearly, adding a new route to the schedule may affect the searches / algorithms indicated above.

6) Remove a route from the schedule. The user enters the vertices defining the route. As with 5), this may affect the searches / algorithms indicated above.
7) Quit the program. Before quitting, your routes should be saved back to the file (the same file and format that they were read in from, but containing the possibly modified route information).

- You must represent the graph as an **adjacency list**. The cities should minimally have a string for a name and any other information you want to add. The data should be input from a file when the program begins. The edges will have multiple values (distance, price) and can be implemented as either a single list of edges with two values each, or as two separate lists of edges, one for each value. You may use the author's various graph classes as a starting point.
- You must use the algorithms and implementations discussed in class for your queries. For example, to obtain the MST you must use either Prim's or Kruskal's algorithm and for the shortest distance and shortest price paths you must use Dijkstra's algorithm. To obtain the shortest hops path you must use breadth-first search. Clearly, since the algorithms are encapsulated in classes in the author's examples, you will need to extract / modify these in order to incorporate them as methods within your class.
- Your main program must have a menu-driven loop that asks the user for any of the choices above. Your output should be clear and well-formatted.
- **W Section Paper:** Rewrite your hashing paper from Assignment 2, taking into account the suggestions I have made on the first version. Your rewrite should have at least 2 sources from verifiable NON-WEB sources, such as a textbook or journal article (although it is ok if you locate the source from the web). I will have a commented version of your Hashing paper back to you by Monday, July 21.
- If you want to try some extra credit, here are some ideas:
  - Add a query to indicate the shortest route (or cheapest route) from a source city to a destination city through a third city. In other words, "What is the shortest path from A to B given that I want to stop at C for a while?" All three cities should be input by the user.
  - For the "minimum" queries above, in the event of a "tie", show all results. If you do this option, you must submit a test file that demonstrates that it works correctly.
  - Allow vertices to be added or removed from your graph (handling corresponding edges correctly).
  - Add graphics to improve your output.
- Below is an example input file, visual graph, and response to some of the queries listed above. The index numbers for the vertices are based on the order that the cities appear in the file (note that the indexing starts at 1). **Look online for an additional test file and more complete sample output.**

```
9
Pittsburgh
Erie
Altoona
Johnstown
Harrisburg
Philadelphia
Scranton
Reading
Allentown
1 2 127 200.00
1 4 66 125.00
1 5 205 125.00
1 6 306 550.00
3 4 43 150.00
3 5 134 225.00
5 6 105 175.00
5 8 59 200.00
6 7 125 275.00
6 9 62 150.00
8 9 35 175.00
```

Shortest distance path from Johnstown to Allentown:
Johnstown 43 Altoona 134 Harrisburg 59 Reading 35 Allentown: Total = 271

Shortest price path from Johnstown to Allentown:
Johnstown 125 Pittsburgh 125 Harrisburg 175 Philadelphia 150 Allentown: Total = 575

Shortest hops path from Johnstown to Allentown:
Johnstown Pittsburgh Philadelphia Allentown: Total = 3