Flight Planner

Due: Monday, Nov 09, 2020 @ 6am pushed to GitHub in the CSE 2341 repo created by following __this__ link.

Introduction

In this project, you will determine all possible flight plans for a person wishing to travel between two different cities serviced by an airline (assuming a path exists). You will also calculate the total cost incurred for all parts of the trip. For this project, you will use information from two different input files in order to calculate the trip plan and total cost.

Origination and Destination Data – This file will contain a sequence of city pairs representing 0-stop flights that can be used in preparing a flight plan. For each of these, the file will also contain a dollar cost for that leg and a time to travel. For each pair in the file, you can assume that it is possible to fly both directions.

Requested Flights – This file will contain a sequence of origin/destination city pairs. For each pair, your program will determine if the flight is or is not possible. If it is possible, it will output to a file the flight plan with the total cost for the flight. If it is not possible, then a suitable message will be written to the output file.

The names of the two input files as well as the output file will be provided via command line arguments.

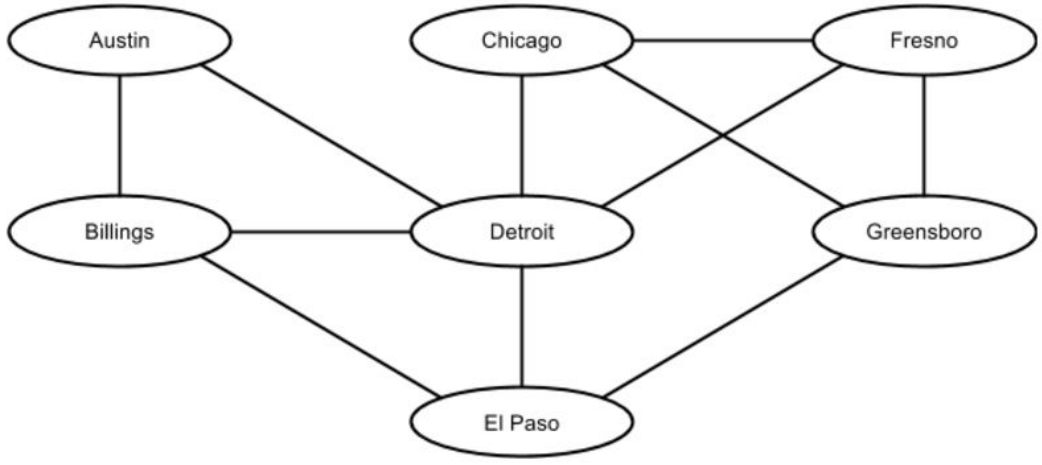

Figure 1 - Undirected Graph

Flight Data

Consider a flight from Detroit to Fresno. It's possible that there is a direct flight, or it may be the case that a stop must be made in Chicago. One stop in Chicago would mean the flight would have two legs. We can think of the complete set of flights between different cities serviced by our airline as an undirected graph. An example of an undirected graph is given in Figure 1.

In Figure 1, a line from one city to another indicates at least one bi-directional flight path between the cities. More than one airline could service flights between those cities. The price and flight time is the same for both directions for a particular airline. If we wanted to travel from El Paso to Chicago, we would have to pass through Detroit. This would be a trip with two legs and one included layover. It is possible that there might not be a path from one city to another city. In this case, you'd print an error message indicating such.

In forming a flight plan from a set of flight legs, one must consider the possibility of cycles. In Figure 1, notice that there is a cycle involving Chicago, Fresno, and Greensboro. In a flight plan from city X to city Y, a particular city should appear no more than one time.

The input file for flight data will represent a sequence of origin/destination city pairs with a cost of that flight. The first line of the input file will contain an integer which indicates the total number of origin/destination pairs contained in the file.

Sample Data

Flight Data

Here is an example of a flight data input file (Note: this does NOT match Figure 1):

```
5
Dallas|Austin|98|47|Spirit
Dallas|Austin|98|59|American
Austin|Houston|95|39|United
Dallas|Houston|101|51|Spirit
Austin|Chicago|144|192|American
```

The first line of the file will contain an integer indicating how many rows of data will be in the file. Each subsequent row will contain two city names, the cost of the flight, and the number of minutes of the flight. Each field will be separated with a pipe (shift-\ on most keyboards).

For flight paths, each layover incurs a time penalty of 43 minutes and it is assumed that the passenger will spend an average of $19 on food/magazines/etc during each layover.

Notice that the flight data also contains the airline and that one pair of cities can have two

flights between them on different airlines. Assume that changing from one airline to another requires a passenger to change terminals and thus incurs a travel penalty of an additional 22 minutes for that layover.

Requested Flight Plans

A sample input file for requested flight plans is shown below. The first line will contain an integer indicating the number of flight plans requested. The subsequent lines will contain a pipe-delimited list of city pairs with a trailing character to indicate sorting the output of flights by time (T) or cost (C). Your solution will find all flight paths between these two cities (if any exists) and calculate the total cost of the flights and the total time in the air.

```
2
Dallas|Houston|T
Chicago|Dallas|C
```

Output File

For each flight in the Requested Flight Plans file, your program will print the three most efficient flight plans available based on whether the request was to order by time or cost. If there are fewer than three possible plans, output all of the possible plans. If no flight plan can be created, then print an error message. If there is a tie between two flight plans on the sorting condition (time or cost), then the order of those paths that tie is inconsequential. Here is an example:

```
Flight 1: Dallas, Houston (Time)
Path 1: Dallas -> Houston. Time: 51 Cost: 101.00
Path 2: Dallas -> Austin -> Houston. Time: 86 Cost: 193.00

Flight 2: Chicago, Dallas (Cost)
Path 1: Chicago -> Austin -> Dallas. Time: 237 Cost: 242.00
Path 2: Chicago -> Austin -> Houston -> Dallas. Time: 282 Cost: 340.00
```

Implementation Details and Requirements

For this project, you'll need to implement a few data structures as described below.

The Stack Class

The Stack class shall represent a LIFO data structure, and it should include the standard methods associated with a stacks (push, pop, peek, and isEmpty). The Stack class should be composed of a LinkedList object (i.e. a Stack object should have a LinkedList as a private data member). You can think of the stack class as adapting the functionality of a LinkedList object to achieve the interface needed for said classes. You are responsible for developing your own interface to the stack class.

The Adjacency List Class

In order to store the structure representing flights serviced by the company, you will implement a simple adjacency list data structure. Essentially, it will be a linked list of linked lists. There will be one linked list for every distinct city. Each list will contain the cities (and other needed info) that can be reached from this city. Figure 2 is an example representation of an adjacency list for the graph in Figure 1.

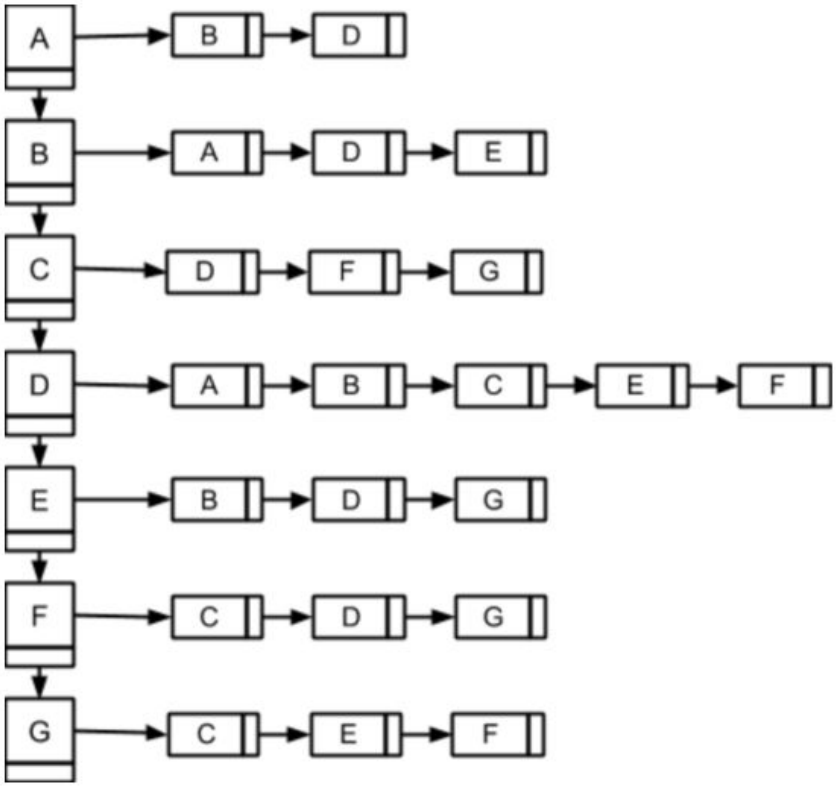

Figure 2 - Adjacency List

The larger squares on the left represent the container of cities (with one node for each city). The list to which each node is pointing represents a city from which you can get to from the parent node. For example, from city A, it is possible to fly to cities B and D.

Iterative Backtracking

To solve this problem, you'll need to implement an exhaustive search of all possible flights originating from a particular city. To achieve this, you'll implement an **iterative backtracking** algorithm (using a stack). As you are calculating the flight path, you will use the stack to "remember" where you are in the search. The stack will also be used in the event that you've gone down a path that does not lead to the destination city. This algorithm method will be discussed in lecture, and you are encouraged to do some of your own research. See https://en.wikipedia.org/wiki/Backtracking as one source of information.

For this project, you may NOT use any of the STL container classes or associated algorithms. You MAY use any classes which you've built for previous sprints or the ones built for this sprint.

Dynamic Memory Management

Any class which manages dynamic memory (the LinkedList will) needs to include an overloaded assignment operator, a copy constructor, and a destructor. Omitting any of these functions could cause segmentation faults, memory leaks, and shallow copy-related problems.

Testing Your Classes

It is expected that, along with your Stack, LinkedList, and AdjacencyList implementations, you include unit tests, run with the CATCH interface. Your test cases should convince a reasonable person (TA or Prof. Fontenot) that you've given a great deal of thought to edge cases and special scenarios. Thinking about what these edge cases and special scenarios are is a great task to take on with a partner from the class. Don't write code together. Instead, find a whiteboard and talk through all the ways to "break" a typical linked list or stack

Your implementation should be **object-oriented in both design and implementation**. Minimize the amount of code you have in your main method. Note that implementing a single class in which you have multiple methods does not make your solution object oriented in design.

Executing Your Program

The final version of your program will be run from the command line with the following arguments:

```
./flightPlan <FlightDataFile> <PathsToCalculateFile> <OutputFile>
```

Alternatively, if we call your program with no arguments, your CATCH tests should run:

```
./flightPlan
```

You will need to modify the CMakeLists.txt file and the workflow.yml file (for GitHub Actions) to support the above.

Grading

Your project will be graded by one of the TAs for the course.

Outcome	Points	Points Earned
Design and Implementation of Doubly Linked List Class	8	
Mechanism for Iterating over Linked list	8	
Design and Implementation of Stack Class	8	
Design and Implementation of Adjacency List	8	
Tests for Linked List and Stack	8	
Iterative Backtracking Algorithm Implementation	15	
Overall Project Software Design	15	
Source Code Quality	15	
Flight Planner Functionality and Correctness	15	