Flight Planner

Due: Monday, Nov 8, 2021 @ 6am pushed to GitHub Classroom Assignment Repo

Introduction

I'm sure we are all familiar with trips that take a couple (or more) shorter flights with layovers in between to get to our final destination. Sometimes this is done because of total price, total time, or the fact that there is no direct flight from where we are to our destination. In this project, you will use information on flights including the duration, cost, and carrier to determine all possible flight plans for a person wishing to travel between two different cities serviced by an airline (assuming a path exists). For each of these possible itineraries, you will also calculate the total cost and time incurred.

So we are all on the same page, let me define some terms with the hopes that I will stay consistent through the remainder of the write up.

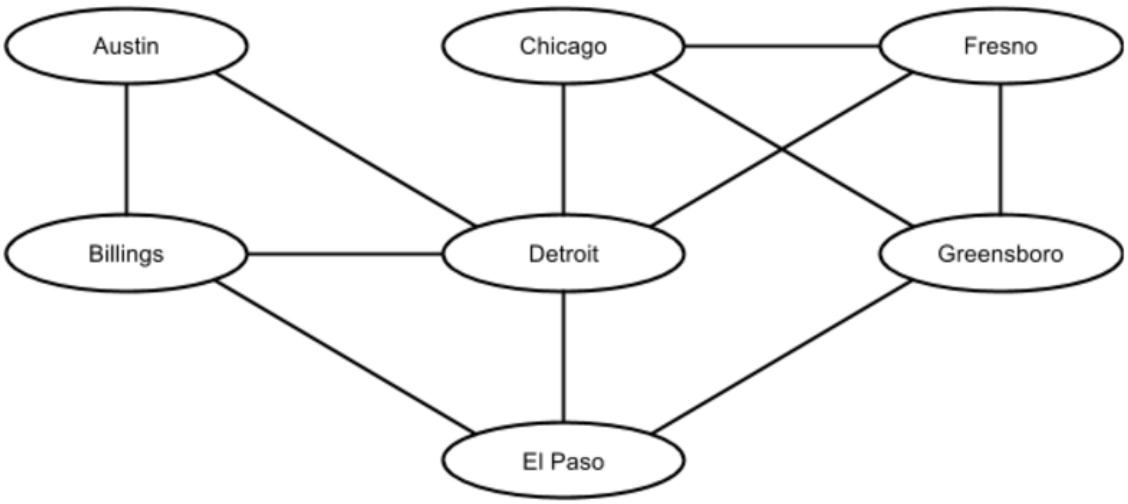
- **Originating City** - Where the flight will be starting from
- **Destination City** - Where we are trying to get to
- **Itinerary** - A set of flights that start at originating city and end in destination city
- **Leg** - one flight that is part of an itinerary.

There will be two input files.

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 - Example Undirected Graph of Flights

Flight Data

Consider a requested flight originating in Detroit with the destination of Fresno. From Figure 1, we can see that there is a direct flight, a flight that goes through Chicago (with 2 legs), and even longer paths like Detroit -> Austin -> Billings -> El Paso -> Greensboro -> Fresno (5 legs).

We can think of the complete set of flights between different cities serviced by our airlines as an undirected graph. In Figure 1, a line from one city to another indicates at least one bi-directional flight path between the cities. In other words, if Cities A and B are connected, that means there's a flight from A -> B and a flight from B -> A. Of course, 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. Of course, it is possible that there is no way to get to a particular destination city from a particular originating city.

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. No one wants to fly in a circle like that...

Data Files

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 there are in this file. Each subsequent row will contain two city names, the cost of the flight, and the number of minutes of the flight. Each component will be separated by a space. "What about cities with spaces in their names?" you may ask. If we use one of those in testing data, we will use an underscore for the space. For example, "El Paso" would appear as "El_Paso". But, you just treat it as a name with no spaces.

For flight paths, each layover incurs a time penalty of 43 minutes and it is assumed that the passenger will spend an average of $23 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 27 minutes for that layover.

Requested Itineraries

A sample input file for requested itineraries is shown below. The first line will contain an integer indicating the number of flight plans requested. The subsequent lines will contain a space-delimited list of city pairs with a trailing character to indicate sorting the output of flights by time (T) or cost (C). If T, your program will find the three itineraries with shortest overall durations. If C, your program will find the three cheapest itineraries overall. If there are fewer than three itineraries between the two cities requested, print all that exists. If no path exists between two cities, print an error message in the output file stating such.

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

Output File

For each flight in the Requested Itineraries 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 break the tie using the other condition. Here is an example for the output for requested flight 1:

```
Flight 1: Dallas, Houston (Time)
  Itinerary 1:
    Dallas -> Houston (Spirit)
    Totals for Itinerary 1:  Time: 51 Cost: 101.00
  Itinerary 2:
    Dallas -> Austin (Spirit)
    Austin -> Houston (United)
    Totals for Itinerary 2:  Time: 117 Cost: 216.00
  Itinerary 3:
    Dallas -> Austin (American)
    Austin -> Houston (United)
    Totals for Itinerary 3:  Time: 129 Cost: 216.00
```

Implementation Details and Requirements

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

The LinkedList Class

The LinkedList class shall be an implementation of a **doubly linked list**. A separate class to represent the node shall also be included. Both classes should be templated so that a doubly linked list can support any kind of data payload. Pay special attention to dynamic memory management and include method implementations consistent with Rule of 3.

You are responsible for determining the interface of your linked list class. Your LinkedList class must include some mechanism to allow other code (client code) to iterate over the list without exposing the underlying implementation. What this means is that a programmer using your LinkedList class should be able to step through the sequence of elements contained within (minimally, from front of list to end of list) without concern for how such iterating is accomplished (you may consider implementing an entire iterator class, or something similar).

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, 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. Of course, each node would include additional information such as cost, time, carrier, etc.

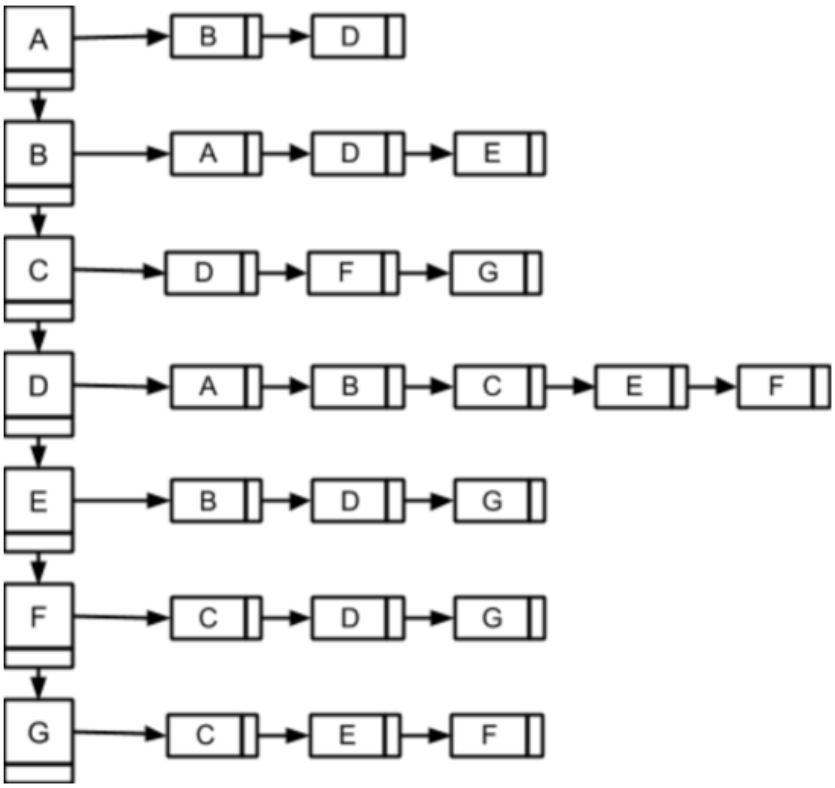


Figure 2 - Adjacency List

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 will implement an exhaustive search of all possible itineraries originating from a particular city that terminate in the destination 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.
Warning: You MAY NOT implement recursive backtracking.

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. The one exception here is that you may use std::string and associated string streams if you’d like.

Dynamic Memory Management

Please adhere to rule-of-3 and please minimize your memory leaks using Valgrind or some other memory profiler.

Testing Your Classes

It is expected that along with your Stack, LinkedList, and AdjacencyList implementations, you include CATCH2 Test.. 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
```

What to Submit

By the beginning of your lab period for the week of Oct 25, you should have the following ready to be checked in lab by your TA:

- Doubly Linked List class and CATCH tests
- Adjacency list class with CATCH tests
- Stack Class and CATCH tests

By Monday, November 8 @ 6am, you should have pushed the following to your Github Classroom Assignment Repo:

- Completed Project

All code submitted should :

- be well formatted and documented
- follow all best practices covered i1n lecture and lab
- follow good design principles for object oriented programming

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	9	
Mechanism for Iterating over Linked list	5	
Design and Implementation of Stack Class	9	
Design and Implementation of Adjacency List	9	
Tests for Linked List, Stack and Adjacency List	8	
Iterative Backtracking Algorithm Implementation	25	
Overall Project Software Design	10	
Source Code Quality	10	
Flight Planner Functionality and Correctness	15	