Project 3 Flight Graph Report

**Author:** Gabriel Bentley

**NetID:** Geb190000

**Course:** CS 3345.502

**Instructor:** Kamran Z. Khan

Project 3 Timeline

* **4/14/21**: created outline of several Classes that will be used in completing the project, such as:
  + flightNode
  + flightEdge
  + flightDataReader
  + flightPathsToCalculateReader
  + outputFileWriter
  + Stack # removed and used the Java class
  + LinkedList # removed and used the Java class
  + adjacencyList
* **4/18/21:** modified several classes giving them member variables, constructors, and getter methods. Classes modified:
  + flightNode
  + flightEdge
  + flightDataReader
  + flightPathToCalculateReader
* **4/19/21:** Completed the flightDataReader and flightPathToCalculateReader Classes so that they will read in the lines of the given file and separate them into ArrayList. Each class has methods to retrieve values from the array lists when given an index for the list. Also worked on the flightNode class giving it the methods getEdge and hasEdge which both take in strings for the destination city name in the edge.
* **4/21/21:** modified code to take the names of the input files and output files from the command line
* **4/24/21:** Completed the adjacency list class that takes in the flight data upon initialization and makes an adjacency list representation of the flight graph, first filling in a linked List of flight nodes with unique city names, then filling in each of those nodes internal edge linkedList with all the cities that are adjacent to it.
* **4/27/21:** created the flightPathFinder class and made use of a modified depth first search to add a method to the class that will find all paths from a start city to the desired destination city. The method will find the paths for one request and store those paths along with their weights in ArrayLists that will be passed to the outputFIleWriter class
* **4/29/21:** Modified the outputFileWriter class so that when given the flight paths and their weights, it will output the 3 most ideal flight paths to the output file given. Put a for loop in the main method so that flightPathFinder and outputFileWriter so that the program will go through all the requests.
* **4/30/21:** Ran tests on the program and finalized the report for the project

Introduction

Project 3 required the reading of flight edges from a text file to build a graph in the form of an adjacency list and reading flight requests from a separate text file. The flight edges consist of two connected cities and have two weight values for the edge; time and travel cost. The flight requests contain a starting city and a destination city along with an identifier telling what the weight priority is time or cost. Using the adjacency list the program must find the three most efficient paths for each request and output those paths to the output file. The program will make use of a stack to backtrack when finding paths from the start city to the destination city.

Approach

I first created an outline of what I would need to successfully read, flight edges and requests from text file, find the shortest paths for the given requests, and output those paths to the given output text file. I created the outline for two classes the flightDataReader and the flightPathsToCalculateRearder, which both make use of Java’s built in split method that separates a string based on the delimiter given. The classes have built in private methods which are called when an object for the class is constructed, that read in the lines of text from the given files and store the values in their private ArrayList variables.

With all the data from the text files stored in class objects, the next step was to use that data to build an adjacency list. To accomplish this required the use of three classes, the flightNode class, the flightEdge class, and the adjacencyList class. The flightNode class served as the vertices for cities that will be in the graph, it contains the city name as a string, a Boolean value for whether it has been visited or not, a linked list of flightEdges to show the cities that are adjacent to it, and helper variables for iterative backtracking, such as currentCost ..etc. The flightEdge class make up the second part of the adjacency list holding the weight of each edge in the graph as well as the city that is connected with this edge. The adjacencyList class uses the data from the edge text file to create a flightNode for every unique city name in the graph, store those flightNodes in a linked list, and then fill those flightNodes one by one with every edge that is connected to it, thus creating an adjacency list representation of a graph. The first linked list is of the flightNodes and each flightNode has its own linkedList of edges.

With the adjacency list built all that is left to do is find the paths for each request and write the three shortest paths to the output file. The flightPlanFinder class makes use of a depth first search (DFS) of the adjacency list to find all paths from the starting city of the request to the destination city of the request, storing the path and total weight values in ArrayLists so that they may be passed to an outputFileWriter object. Finding all paths from one city to another in a graph requires the use of backtracking so that when the end of a path is reached the program will go down a different branch of that path. I first wrote a DFS that made use of recursion since recursive functions are made for this type of problem and use the built in stack to execute calls of the recursive function so that the entire graph is traversed. Upon reviewing the project specifications for flightPlan I found that iterative backtracking was a requirement, so I set to work on an adjacent method in the flightPlanFinder class called DFS which makes use of the built-in java stack to find all the paths for a given request. Unfortunately, the DFS that makes use of iterative backtracking fails to function properly when given a complex graph, so I made the decision to use the paths gathered from the recursive findAllPaths method since they are confirmed to be accurate and still make use of backtracking.

With all the paths found they are sent to an outputFileWriter Object where the three paths with the smallest weights are appended to the output text file in the desired format. Smallest weight paths are found for both weight types and the priority weight type path is written to the output file. The flightPathFinder and outputFileWriter classes create new objects for every request that was read from the request file.

The entire program is executed from the flightPath class which holds the main method. Inside the main method the three arguments for the file names are read and stored in variables, a new output file is created destroying the old output file, and objects of each class are built, with the flightPathFinder and outputFileWriter being in a for loop that iterates once for every request read.

When running the program from the terminal make sure to use the full path name of each text file for the program to function properly examples of how the program runs are at the bottom of the report.

Challenges

* There was a typo on the project output example, Chicago -> Austin -> Dallas takes 239 minutes not 237, since 47 + 192 = 239. I kept trying to debug my code wondering why it didn’t match with the example output.
* I at first imagined that since we were looking for the minimum flight path, that it would be best to find it through the use of Dijkstra’s algorithm, but upon trying to implement it I quickly came to the conclusion that a modified depth first search method would be better suited to find multiple paths.
* The decision of whether to find the output for all request, then write it all at once to the output file, or each time the program finds the output for one request, that request is appended to the output file. I decided to append request results to the output file and run the needed classes in a for loop that iterates for every request.
* I have never run a java program from the command line before, so I had to spend a lot of time researching how to get the program to work with output from the command line.
* I first wrote a function that did recursive backtracking to find all paths from the source to the destination city, since it is better suited to the task, but upon rereading the project document I found that Iterative backtracking was requested, so I had to modify my code

Classes

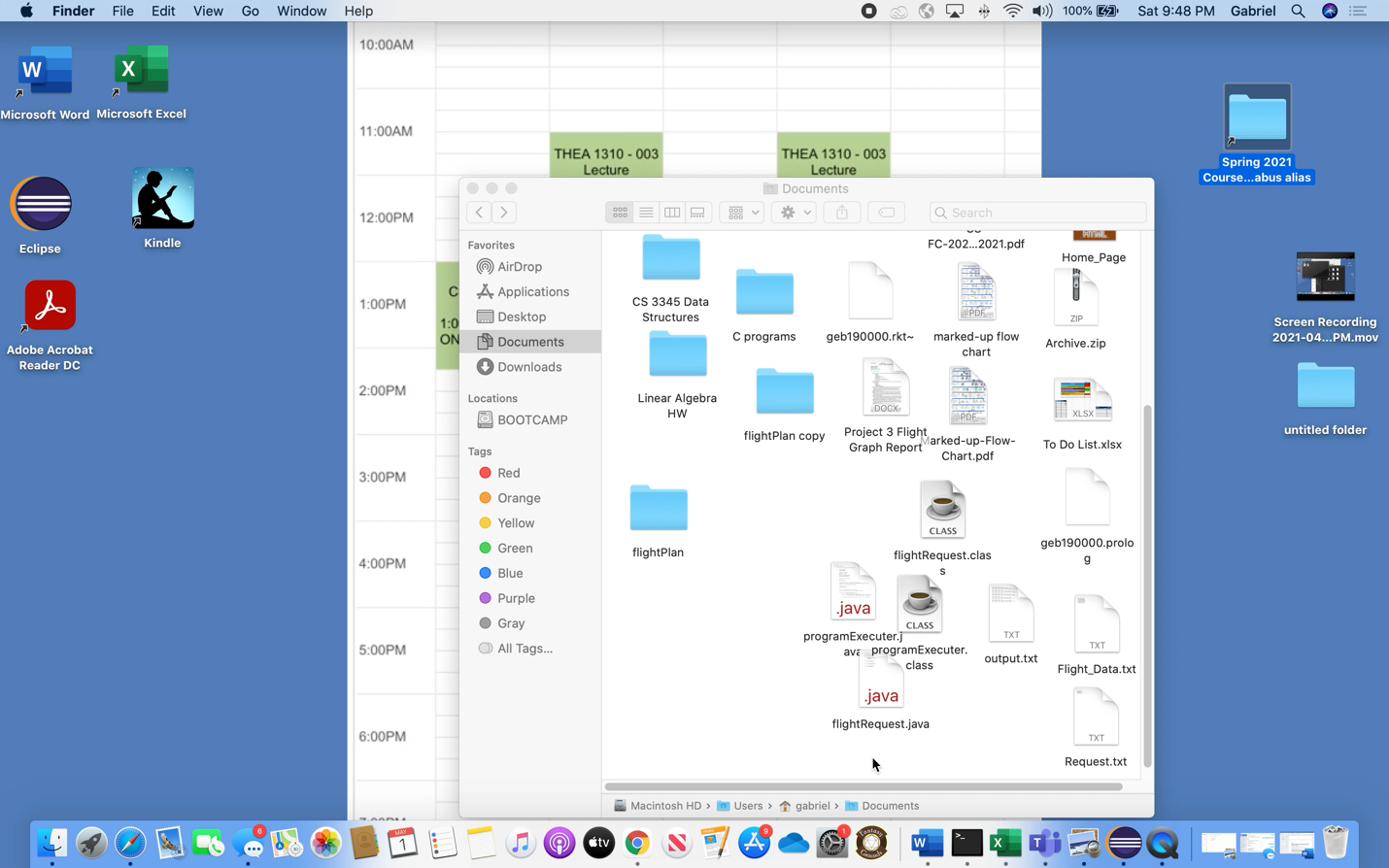
* **flightNode**
  + Purpose: The purpose of the flight Node class is to act as an object for the adjacency list holding the city name, and list of adjacent cities.
  + Variables:
    - Boolean visited
    - String cityName
    - LinkedList<flightEdge> adjacentCities
    - ArrayList<String> currentPath
    - ArrayList<Integer> currentTime
    - ArrayList<Double> currentCost
  + Methods:
    - Standard getter methods for the variables.
    - addEdge, added a given edge object to adjacentCities.
    - Setter methods for the array lists
* **flightEdge**
  + Purpose: The purpose of the flightEdge class is to hold the weights between two cities, the flightNode that has the flightEdge in its LinkedList will be the starting city, and the flightNode variable the flightEdge class has contains the destination city.
  + Variables:
    - flightNode endLocation
    - double costWeight
    - int timeWeight
  + Methods:
    - Standard getter methods for variables.
* **flightDataReader**
  + Purpose: The purpose of the flightDataReader class is to read in the edges from the provided text file and store the read information into 4 different lists.
  + Variables:
    - String filename
    - ArrayList<String> startingCityNames
    - ArrayList<String> destinationCityNames
    - ArrayList<Double> costWeightArrayList
    - ArrayList<Integer> timeWeightArrayList
    - Int numberOfEdges
  + Methods:
    - Standard getter methods for variables.
    - Private method setNumberOfEdges which reads the first line of the file to get the number of edges and stores it in the numberOfEdges variable. Called when flightDataReader object is constructed.
    - Private method fillArrayLists which read and stores the edge information in the ArrayList variables. Called when flightDataReader object is constructed.
* **flightPathsToCalculateReader**
  + Purpose: The purpose of this class is to read the flight requests from the given text file and store the data into ArrayLists.
  + Variables:
    - String requestsFile
    - Int numLines
    - ArrayList<flightRequest> requestArray
    - ArrayList<String> startCities
    - ArrayList<String> destinationCities
    - ArrayList<String> weightType
  + Methods:
    - Standard getter methods for the class variables.
    - Private method setNumberOfLines is a copy of the setNumberOfEdges method in the flightDataReader class.
    - Private method fillArrayLists does the same thing as the fillArrayLists method in the flightDataReader class.

**adjacencyList**

* + Purpose: The purpose of the adjacencyList class is to build an adjacency list representation of the graph made up of the edges read with the flightDataReader object.
  + Variables:
    - LinkedList<flightNode> nodeList
  + Methods:
    - Getter method for nodeList.
    - Private method inList which when given a city name string will determine if that city name is already in the nodeList. Returns a Boolean, used in the fillNodeLinkedList method.
    - Private method fillNodeLinkedList will create a flightNode object for each unique city name in the file ArrayLists given by the flightDataReader object passed to this class upon its construction. This is accomplished by looping through each string array list and using the inList method to look for unique names.
    - Private method fillNodeEdges will go through each node in the nodeList linked list and add flightEdges to that nodes flightEdge linked list for every city adjacent to that node. This satisfies the condition of an adjacency list, where there is a linked list of nodes and attached to each of those nodes there is a linked list of adjacent cities to that node.
* **flightPlanFinder**
  + Purpose: The purpose of the flight plan finder is to find all the paths from a request start city to its end city and find total weight for each of the paths. The paths and their weights are stored in ArrayLists for later use.
  + Variables:
    - adjacencyList graph
    - ArrayList<String> cityVisitedList
    - ArrayList<String> pathHolder
    - ArrayList<Double> costHolder
    - ArrayList<Integer> timeHolder
    - Stack<flightNode> nodeStack
    - flightPathsToCalculateReader travelPairs;
  + Methods:
    - Getter methods for the three array lists that will be used outside of the object; pathHolder, costHolder, and timeHolder.
    - Private method listSearch when given a cities name as a string this method will find the index of that city in the adjacency list and return it.
    - Private methods findAllPaths and findAllPathsUtil perform a DFS on the adjacency list making use of backtracking to find all paths from the start city in the given request to the end city in the given request pushing the city nodes onto the stack when going down one depth and popping that city node from the stack when backtracking.
    - Private method format will when given a path and weight values will format the path string to represent what it will be outputted as when written to the output text file.
    - Private Method DFS does a depth first search through the use of Iterative backtracking, unlike how the FindAllPaths methods use recursive backtracking.
* **outputFileWriter**
  + Purpose: To find the three most efficient paths from the ones given and write them to the output file
  + Variables:
    - String fileName;
    - int requestNum;
    - ArrayList<String> paths;
    - ArrayList<Integer> times;
    - ArrayList<Double> costs;
    - String weightPriority;
    - flightPathsToCalculateReader pair;
  + Methods:
    - Private method sortPaths will search for the three shortest paths by iterating through a for loop at most 3 times in which the path with the smallest priority weight is found, written to the file, and removed from the list of paths so that the second or third smallest weight can be found.
    - Private method findShortestWeightIndex will loop through the priority weight array to find the index with the smallest weight and return the index.
* **programExecutor**
  + Purpose: To be the code executor for the program, holds the main function that takes in the file names and creates class objects so that the adjacency list is created, and the paths for all requests are written to the output file
  + Variables:
    - String flightDataFile
    - String requestFile
    - String outputFile
    - flightDataReader flightData
    - flightPathsToCalculateReader flightRequests
    - adjacencyList graph
    - flightPlanFinder finder
    - outputFileWriter writer
  + Methods:
    - Only contains the main method which creates a flightDataReader object to read the edges, a filghtPathsToCalculateReader object to read the requests, an adjacencyList object to build an adjacny list with the flightData, and a for loop that runs for the number of requests that creates new flightPlanFinders and outputFileWriters to write the Request results to the output.txt file

How To Execute Code

Video of Code execution double click on the picture



Graph Representation of Flight\_Data.txtDiagram

Description automatically generated

Flight\_Data.txt

8

Dallas|Austin|98|47

Austin|Houston|95|39

Dallas|Houston|101|51

Austin|Chicago|144|192

Boston|Chicago|157|152

Boston|Seattle|222|205

Boston|Austin|173|114

Dallas|Seattle|122|78

Request.txt

5

Dallas|Houston|T

Chicago|Dallas|C

Chicago|Dallas|T

Boston|Dallas|C

Seattle|Austin|C