Flight Delays

Design Specification Document

McMaster University
Software Engineering
Version: 1.0
04/13/2016

SFWRENG/COMPSCI 2XB3 - Lab 03
Software Engineering Practice and Experience:
Binding Theory to Practice

Group 24:
Abeed Alibhai - 1428809
Rahul Bablani - 1418434
Natasha DeCoste - 1206976
Chaoyi Kuang - 1402031
Muyu Zhang - 1405907

Table of Contents

Revision

Team Members:

Abeed Alibhai 1428809
Rahul Bablani 1418434
Natasha DeCoste 1206976
Chaoyi Kuang 1402031
Muyu Zhang 1405907

Revision History:

V1.0 - 04/12/2016

Scrum Master: Rahul Bablani
(responsible to manage the project to meet all the milestones and to produce the prototype)

Log Master: Rahul Bablani
(responsible to keep the log as a living document of the project)

Client(s): Anybody looking to minimize expected delay times on their flight to any domestic destination in the US.

Researcher(s): All*
Designer(s): Natasha DeCoste
Programmer(s): All*
Tester(s): All*

*Roles are constantly changing due to the team's decision to implement *Scrum*

By virtue of submitting this document we electronically sign and date that the work being submitted by all the individuals in the group is their exclusive work as a group and we consent to make available the application developed through [CS] or [SE]-2XB3 project, the reports, presentations, and assignments (not including my name and student number) for future teaching purposes.

Contributions

Name	Role(s)	Contributions	Comments
Abeed Alibhai	Product Owner Programmer Researcher Tester	Sorting Class Searching Class	--
Rahul Bablani	Scrum Master Log Master Programmer Researcher Tester	Sorting Class Searching Class Logs & Meeting Minutes Brainstorming and Project Proposal	--
Natasha DeCoste	UI Designer Programmer Researcher Tester	API Class Searching Class Graphing Class Shortest Path Class Brainstorming and Project Proposal	--
Chaoyi Kuang	Programmer Researcher Tester	ADT Flight Object's Class Brainstorming and Project Proposal	--
Muyu Zhang	--	--	Did not show up to any meetings

Executive Summary

This project is intended to find a shortest path between two airports based on the least amount of delay between the two points and not the distance or time of travel between. We obtained our information from an American dataset containing delays from airport to airport all across the United States. The purpose is to save the user from wasting time between travel due to delays at the airport.

Our intention was to graph the Vertices as being the different airports, sorted into a Binary Search Tree using the airport codes. We would then map directed edges between the airports that correspond to the flight between the airports and the edge weight is the average delay on that flight path (arrival delay and departure delay). Then we would be able to find the shortest path between the two chosen airports by relaxing all the edges with a shortest path algorithm. Our GUI was just a simple JavaFX application that would let the user pick the origin and destination and then display the route with the shortest delay.

Description of classes/modules

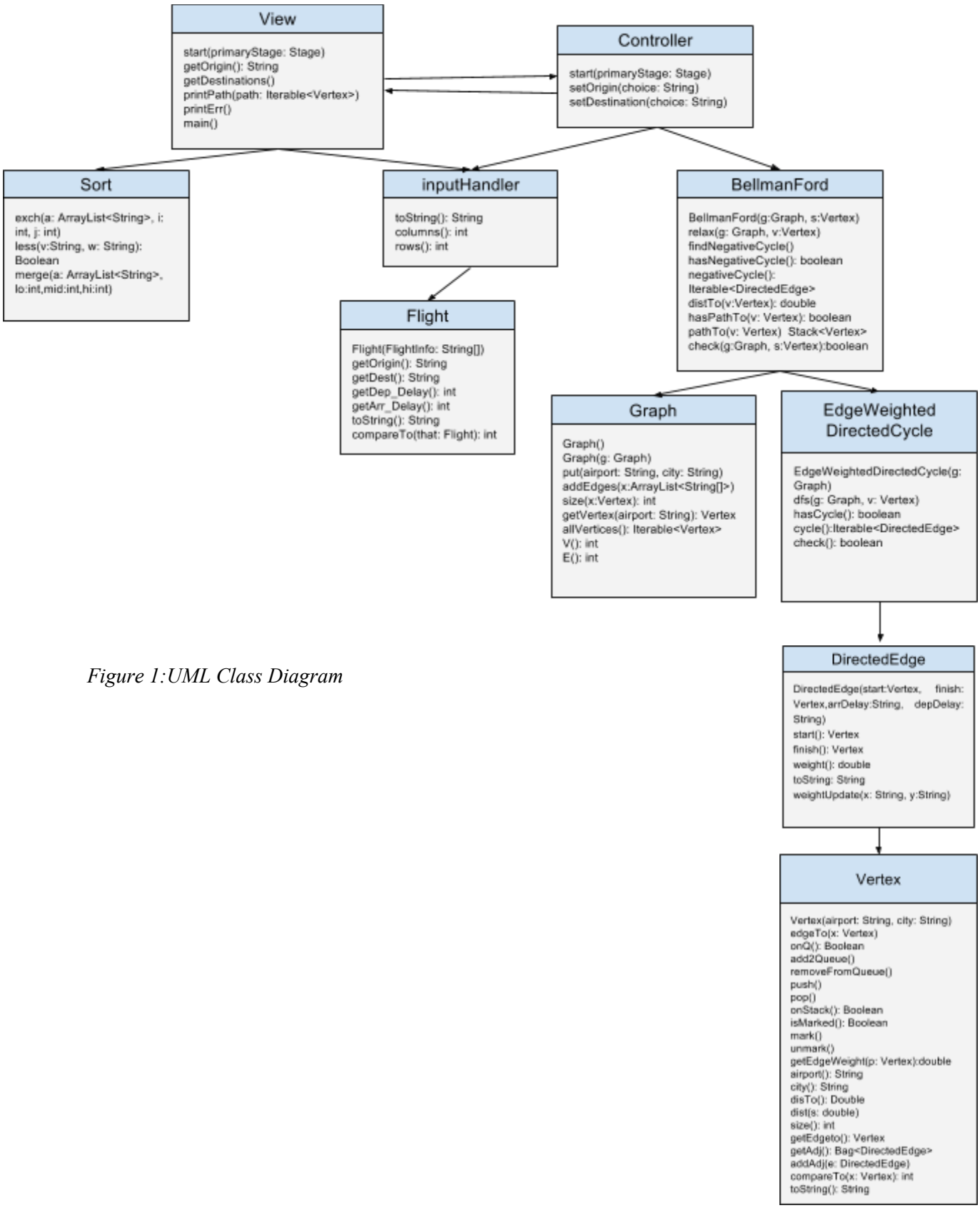

Figure 1:UML Class Diagram

MIS

CLASS: Flight

Implementation of the Flight ADT which extends the Comparable class, this makes the comparisons between arrival delays of two flights easier. Every Flight object contains the origin city, destination city, departure delay and arrival delay.

USES:

N/A

VARIABLES:

N/A

ACCESS PROGRAMS:

Flight(FlightInfo: String[])

The constructor of Flight objects.

getOrigin(): String

Return the string of the origin airport code of current Flight.

getDest(): String

Return the string of destination airport code of current Flight.

getDep_Delay(): int

Return the departure delay of current Flight.

getArr_Delay(): int

Return the arrival delay of current Flight.

toString(): String

Return a string contains all the information of current Flight.

compareTo(that: Flight): int

Override the compareTo method to make Flight objects are comparable by the arrival delays.

MID

CLASS: Flight

Implementation of the Flight ADT. Take a String from the input String list FlightInfo and create a Flight object based on that string. A typical String in FlightInfo contains the information of the origin airport code, origin city name, destination airport code, destination city name, departure time , departure delay and arrival delay of a flight. To make our application more efficient we only store the origin airport code, destination airport code, departure delay and arrival delay.

USES:

N/A

VARIABLES:

origin: String

Origin airport code.

destination: String

Destination airport code.

dep_delay: int

Departure delay of a flight.

arr_delay: int

Arrival delay of a flight.

ACCESS PROGRAMS:

Flight(FlightInfo: String[])

The constructor of Flight objects. Assign the value of origin, destination, dep_delay and arr_delay.

getOrigin(): String

Return the string of the origin airport code of current Flight.

getDest(): String

Return the string of destination airport code of current Flight.

getDep_Delay(): int

Return the departure delay of current Flight.

getArr_Delay(): int

Return the arrival delay of current Flight.

toString(): String

Return a string contains all the information of current Flight.

compareTo(that: Flight): int
Override the compareTo method to make Flight objects are comparable by the arrival delays. If the current flight delay is less than the other delay return -1; if they are the same return 0; if it has more delay return 1.

MIS

CLASS: EdgeWeightedDirectedCycle

Class to check for cycles in the graph.

USES:

edu.princeton.cs.algs4.Stack

VARIABLES:

N/A

ACCESS PROGRAMS:

EdgeWeightedDirectedCycle(g: Graph)

The constructor of the EdgeWeightedDirectedCycle Object.

dfs(g: Graph, v: Vertex)

Performs a depth first search on the Graph g, starting at Vertex v.

hasCycle(): boolean

Return the boolean representing whether there is a cycle in the graph.

cycle(): Iterable<DirectedEdge>

Returns the cycle as an iterable object.

check(): boolean

Returns whether or not there is a cycle all all Vertices are connected in that cycle.

MID

CLASS: EdgeWeightedDirectedCycle

Class that checks if a graph contains cycles.

USES:

edu.princeton.cs.algs4.Stack

VARIABLES:

g: Graph

Graph to search for a cycle within.

cycle: Stack<DirectedEdge>

Stack of the Edges in the cycle.

ACCESS PROGRAMS:

EdgeWeightedDirectedCycle(g: Graph)
The constructor of the EdgeWeightedDirectedCycle Object.

dfs(g: Graph, v: Vertex)
Performs a depth first search on the Graph g, starting at Vertex v.

hasCycle(): boolean
Return the boolean representing whether there is a cycle in the graph.

cycle(): Iterable<DirectedEdge>
Returns the cycle as an iterable object.

check(): boolean
Returns whether or not there is a cycle all all Vertices are connected in that cycle.

MIS

CLASS: Vertex

Implementation of the Vertex class builds objects to be used in order to graph the locations of the airports with their respective airport codes as well as allows for creating paths between airports/cities.

USES:

edu.princeton.cs.algs4.Bag

VARIABLES:

airport: String
The airport code

city: String
The location of the airport.

adj: Bag<DirectedEdge>
The adjacent airports.

N: int
The size of this Vertex's subtree.

left: Vertex
The Vertex connected to the left of this one in a binary search tree.

right: Vertex
The Vertex to the right of this one in a binary search tree.

ACCESS PROGRAMS:

Vertex(airport: String, city: String)
The constructor of Vertex objects.

edgeTo(x: Vertex)
Used for adding the Vertex that is leading to this Vertex in a path.

onQ(): Boolean
Return boolean whether or not this has been added to a queue. Used for finding shortest path.

add2Queue()
Indicating this Vertex is being added to the queue for shortest path.

removeFromQueue()
Removing this Vertex from being on the queue for shortest path.

push()
Indicating the Vertex is being added to a stack for a shortest path calculation.

pop()
Indicating the Vertex is being deleted to a stack for a shortest path calculation.

onStack(): Boolean
Return boolean whether or not this Vertex is on the stack for a shortest path calculation.

isMarked(): Boolean
Returns a boolean whether or not this Vertex has been marked in a path discovery algorithm.

mark()
Marking the Vertex as discovered.

unmark()
Mark the Vertex as undiscovered.

getEdgeWeight(p: Vertex): double
Returns the edge weight between the two vertices if there exists a weighted edge between them.

airport(): String
Returns a string of the airport code.

city(): String
Return a string of the airport's city/location as stated in the dataset.

distTo(): double
Returns a double of the distance to this point used in a shortest path algorithm..

dist(s: double)

Takes a double and updates the distance to this Vertex.

size(): int
Returns an integer representing the size of the subtree for this Vertex..

getEdgeto(): Vertex
Returns the Vertex that has an edge to this Vertex.

getAdj(): Bag<DirectedEdge>
Returns the adjacent edges to this Vertex.

addAdj(e: DirectedEdge)
Add an edge adjacent to this Vertex.

compareTo(x: Vertex): int
Override the compareTo method to make Vertex objects comparable by the airport codes.

toString(): String
The String representation of the Vertex object.

MID

CLASS: Vertex

Implementation of the Vertex class builds objects to be used in order to graph the locations of the airports with their respective airport codes as well as allows for creating paths between airports/cities.

USES:

edu.princeton.cs.algs4.Bag

VARIABLES:

airport: String
The airport code

city: String
The location of the airport.

adj: Bag<DirectedEdge>

The adjacent airports.

N: int
The size of this Vertex's subtree.

left: Vertex
The Vertex connected to the left of this one in a binary search tree.

right: Vertex
The Vertex to the right of this one in a binary search tree.

edgeTo: Vertex
The Vertex connecting to this in a path.

distTo: double
The distance to this Vertex in a shortest path implementation.

onQ: boolean
To keep track of whether the Vertex is on the queue.

onStack: boolean
To keep track of whether the Vertex is on the Stack.

marked: boolean
Whether or not this Vertex has been discovered.

ACCESS PROGRAMS:

Vertex(airport: String, city: String)
The constructor of Vertex objects. Assigns variables airport and city using the parameters and all other variables to null/false for booleans, except distTo is assigned to positive infinity.

edgeTo(x: Vertex)
Used for adding the Vertex that is leading to this Vertex in a path. Assigns the edgeTo variable to the parameter.

onQ(): Boolean
Return boolean variable onQ.

add2Queue()
Indicating this Vertex is being added to the queue for shortest path by changing the onQ variable to true.

removeFromQueue()
Removing this Vertex from being on the queue for shortest path by changing onQ to false.

push()
Indicating the Vertex is being added to a stack for a shortest path calculation by changing onStack variable to true.

pop()
Indicating the Vertex is being deleted to a stack for a shortest path calculation by changing onStack variable to false.

onStack(): Boolean
Return boolean variable onStack.

isMarked(): Boolean
Returns boolean variable marked.

mark()
Marking the Vertex as discovered by changing the variable marked to true.

unmark()
Mark the Vertex as undiscovered by changing the variable marked to false.

getEdgeWeight(p: Vertex): double
Iterates through the adjacent DirectedEdges in adj to find the one that connect this Vertex to p (from the parameter). It then return the weight of the edge connecting them.

airport(): String
Returns a string of the airport code by returning the airport variable.

city(): String

Return a string of the airport's city/location as stated in the dataset by returning the airport variable.

distTo(): double
Returns a double of the distance to this point by returning the distTo variable.

dist(s: double)
Takes a double and updates the distance to this Vertex by changing the distTo variable to s.

size(): int
Returns an integer representing the size of the subtree for this Vertex by returning the variable N.

getEdgeto(): Vertex
Returns the edgeTo variable.

getAdj(): Bag<DirectedEdge>
Returns the adjacent edges to this Vertex by returning the variable adj.

addAdj(e: DirectedEdge)
Add an edge adjacent to this Vertex by adding it to the Bag of DirectedEdges adj.

compareTo(x: Vertex): int
Override the compareTo method to make Vertex objects comparable by the airport codes. They are compared using the Strings.

toString(): String
The String representation of the Vertex object includes the airport code and the location.

MIS

CLASS: Graph

Implementation of Graph class

USES:

edu.princeton.cs.algs4.Queue, Vertex, DirectedEdge

VARIABLES:

N/A

ACCESS PROGRAMS:

Graph()

The constructor of Graph objects.

Graph(g: Graph)

The constructor of Graph objects based on another Graph object.

put(airport: String, city: String)

Add a new vertex to the graph.

addEdges(x: ArrayList<String[]>)

Creates Vertices from x, line by line, and adds them into the graph and created directed edges between them.

size(x: Vertex): int

Return the size of the subtree of vertex x.

getVertex(airport: String): Vertex

Return a certain vertex.

allVertices(): Iterable<Vertex>

Return all the vertices of the graph as an Iterable.

V(): int

Return the number of vertices.

E(): int

Return the number of edges.

MID

CLASS: Graph

USES:

edu.princeton.cs.algs4.Queue, Vertex, DirectedEdge

VARIABLES:

E: int

The number of the edges in the graph.

V: int

The number of the vertices in the graph.

root: Vertex

The root of the graph.

ACCESS PROGRAMS:

Graph()

The constructor of Graph objects. Initialize the number of edges and vertices to 0 and root to null.

Graph(g: Graph)

The constructor of Graph objects based on another Graph object. This method goes through all the vertex in the Graph g and add all the edges to current graph. It also finds all the DirectedEdges that connected to the destination airport.

put(airport: String, city: String)

Using airport and city, it creates a temporary vertex and put it into the binary search tree. And update the root.

put(n: Vertex, t: Vertex): Vertex

It takes the root vertex and the current vertex and put the current vertex into the binary search tree. The root might be changed during this progress so the latest root will also be returned.

addEdges(x: ArrayList<String[]>)

Creates Vertices from x by iterating through all the String [] and accessing the Destination and Origin and creating them as Vertices in the Graph. It then adds directed edges between the origin and the destination or if there already exists an edge, it updates the weight.

size(x: Vertex): int
Return the size of the subtree of vertex x.

getVertex(airport: String): Vertex
Search the vertex x in binary search tree and return the vertex once it's found.

getVertex(x: Vertex, airport: String): Vertex
Recursively go through the left node and right node of a certain vertex to find the target vertex and return it once it's found.

allVertices(): Iterable<Vertex>
Return all the vertices of the graph as an Iterable.

allVertices(x: Vertex, queue: Queue<Vertex>)
Recursively go through the left node and right node of a certain vertex to all the vertices and return them in queues.

V(): int
Return the number of vertices.

E(): int
Return the number of edges.

MIS

CLASS: DirectedEdge

The DirectedEdge object is used to connect vertices that are adjacent in a graph in order to get paths between airports. It is weighted as well to take into consideration the delays between airports rather than actual distance between them.

USES:

N/A

VARIABLES:

N/A

ACCESS PROGRAMS:

DirectedEdge(start: Vertex, finish: Vertex, arrDelay: String, depDelay: String)
The constructor of DirectedEdge creates an edge between start and finish.

start(): Vertex
Returns the start Vertex of the DirectedEdge.

finish(): Vertex
Returns the finish Vertex of the DirectedEdge.

weight() : double
Returns the weight of the edge.

toString() : String
The String representation of the DirectedEdge object.

weightUpdate(x: String, y: String)
Changes the weight on the edge using x and y. Used for path finding.

MID

CLASS: DirectedEdge

The DirectedEdge object is used to connect vertices that are adjacent in a graph in order to get paths between airports. It is weighted as well to take into consideration the delays between airports rather than actual distance between them.

USES:

N/A

VARIABLES:

start: Vertex

The Vertex representing the origin airport.

finish: Vertex

The Vertex representing the destination airport.

weight: double

The edge's weight, corresponding to the average flight delays from origin to destination airport.

n: double

Used in order to cumulatively average the delay for the weight.

ACCESS PROGRAMS:

DirectedEdge(start: Vertex, finish: Vertex, arrDelay: String, depDelay: String)

The constructor of DirectedEdge creates an edge between start and finish. Calculates the edge weight by averaging the arrival delays as well as departure delays.

start(): Vertex

Returns the start Vertex of the DirectedEdge.

finish(): Vertex

Returns the finish Vertex of the DirectedEdge.

weight() : double

Returns the weight of the edge.

toString() : String

The String representation of the DirectedEdge object.

weightUpdate(x: String, y: String)

Changes the weight on the edge using x and y. Used to update the weight to be the average when reading duplicates in from the dataset. Update the weight by parsing into doubles, the x and y strings (arrival delays and departure delays) and then adding them, subtracting the current edge weight and dividing by n (number of duplicates in dataset for this origin to destination..

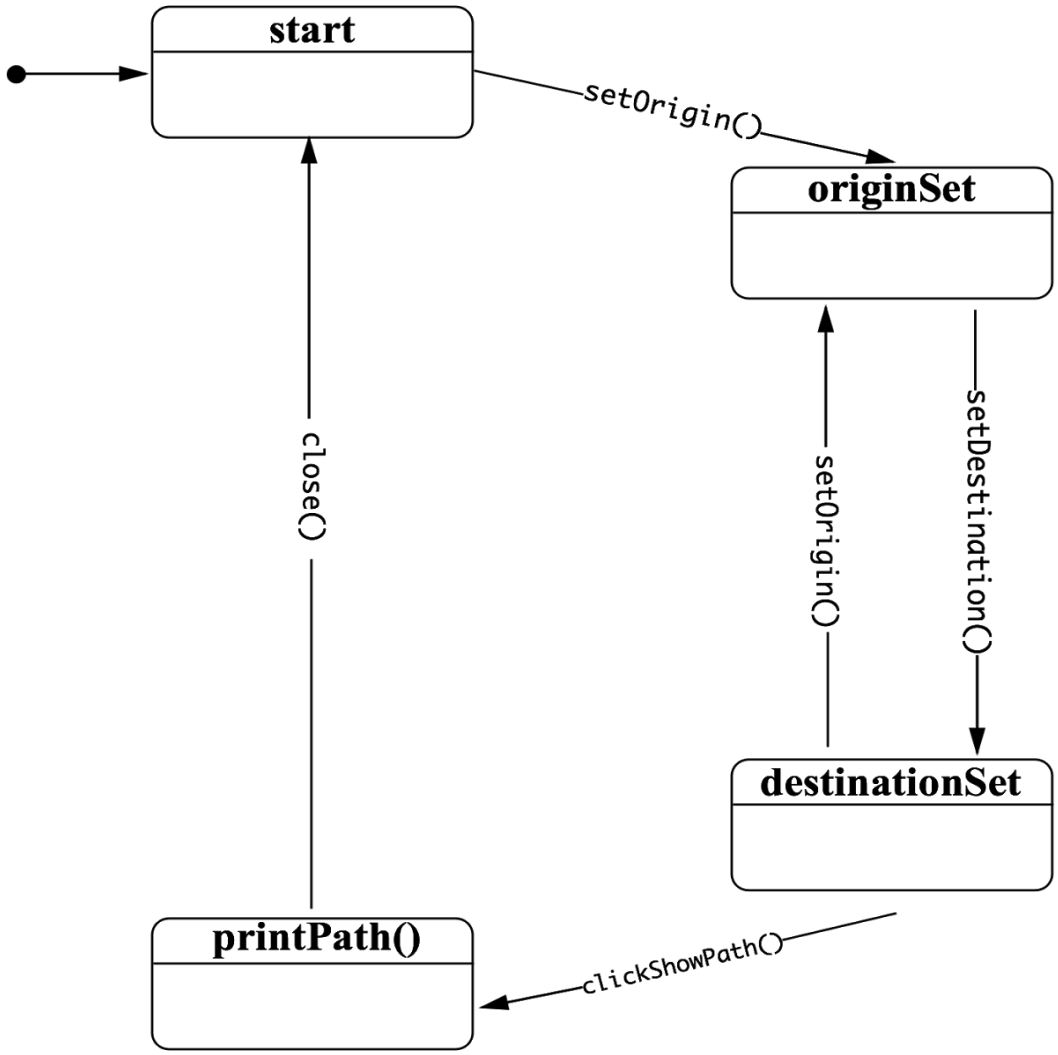

Figure 3: UML State Machine for View Class

MIS

CLASS: View

Class to display the GUI for the user.

USES:

App.fxml

The fxml file to use with the java SceneBuilder.

application.css

The css styling of the objects created with JavaFX.

inputHandler

Used to get the origin and destination airport codes for the User to pick from.

Sort
Used to sort the airport codes in alphabetical order for easier navigation.

VARIABLES:
N/A
ACCESS PROGRAMS:
start(primaryStage: Stage)
JavaFX required function to run when launch is called. Opens the main GUI screen.

getOrigin(): String
Returns the origin (airport code).

getDestinations()
Populates the destinations comboBox with destination airport codes sorted in alphabetical order.

printPath(path: Iterable<Vertex>)
Prints the path for the user in a separate dialog box.

printErr()
Prints an error statement in a new dialog box if there is no path between the origin picked and the destination picked.

main()
Runs the JavaFX required launch function.

MID
CLASS: View
Class to display the GUI for the user.
USES:
App.fxml
The fxml file to use with the java SceneBuilder.

application.css
The css styling of the objects created with JavaFX.

inputHandler
Used to get the origin and destination airport codes for the User to pick from.

Sort
Used to sort the airport codes in alphabetical order for easier navigation.

VARIABLES:

origin: String
Holding the origin choice the user clicks on in the comboBox in the GUI.

destination: String
To contain the destination the user chooses in the comboBox in the GUI.

comboDest: comboBox
Containing the list of possible destinations the user can pick from.

options: ArrayList<String []>
Populated with the options for the user for comboBoxes in the GUI.

ACCESS PROGRAMS:

start(primaryStage: Stage)
JavaFX required function to run when launch is called. Opens the main GUI screen. Runs the graphical animation.

getOrigin(): String
Returns the origin (airport code) by returning the variable origin.

getDestinations()
Populates the destinations comboBox with destination airport codes sorted in alphabetical order.

printPath(path: Iterable<Vertex>)
Prints the path for the user in a separate dialog box.

printErr()
Prints an error statement in a new dialog box if there is no path between the origin picked and the destination picked.

main()
Runs the JavaFX required launch function.

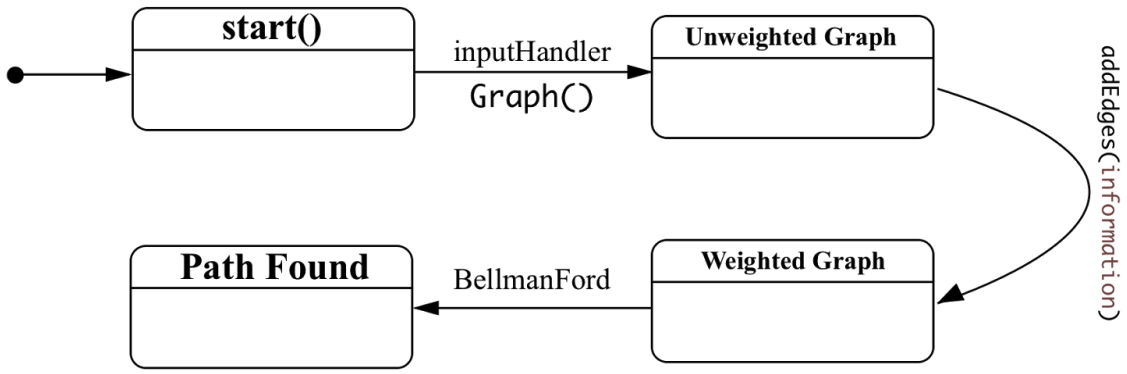

Figure 4: UML State Machine for Controller Class

MIS

CLASS: Controller

To communicate between the front end and back end.

.

USES:

inputHandler

To get the information from the datasets.

Graph

Created the graph to hold all the information.(Vertices and Edges)

BellmanFord

The Algorithm to get the shortest path.

View

The JavaFX GUI for the user frontend.

VARIABLES:

origin: String

Holding the origin choice the user clicks on in the comboBox in the GUI.

destination: String

To contain the destination the user chooses in the comboBox in the GUI.

Result: Label (@FXML Object)

Containing the list of possible destinations the user can pick from.

ACCESS PROGRAMS:

start(primaryStage: Stage)

Runs the graph in the backend and communicates with the View.

setOrigin(choice: String)

Sets the origin to the matching airport of the user's selection.

setDestination(choice: String)

Sets the destination variable to the matching airport of the user's selection.

MID

CLASS: Controller

To communicate between the front end and back end.

.

USES:

inputHandler

To get the information from the datasets.

Graph

Created the graph to hold all the information.(Vertices and Edges)

BellmanFord

The Algorithm to get the shortest path.

View

The JavaFX GUI for the user frontend.

VARIABLES:

origin: String

Holding the origin choice the user clicks on in the comboBox in the GUI.

destination: String

To contain the destination the user chooses in the comboBox in the GUI.

Result: Label (@FXML Object)

Containing the list of possible destinations the user can pick from.

ACCESS PROGRAMS:

start(primaryStage: Stage)

Creates a graph object and populates it with the information from the datasets. Communicates with the View to get the user's choice of origin and destination then tries to determine the shortest delay time between them.

setOrigin(choice: String)

Sets the origin variable to the matching airport of the user's selection.

setDestination(choice: String)

Sets the destination variable to the matching airport of the user's selection.

MIS

CLASS: BellmanFord

To traverse the graph in order to find the shortest path based on least amount of flight delay time.

.

USES:

Graph

Created the graph to hold all the information.(Vertices and Edges)

Vertex

Mapping the vertices from the Graph.

edu.princeton.cs.algs4.Queue;

edu.princeton.cs.algs4.Stack;

VARIABLES:

cycle: Iterable<DirectedEdge>

To hold a cycle between vertices in the Graph.

ACCESS PROGRAMS:

BellmanFord(g:Graph, s: Vertex)

The constructor for the Object takes a graph and vertex and relaxes the edges in the graph.

relax(g: Graph, v:Vertex)

Relaxes the edges in the graph updating them to map out the shortest path.

findNegativeCycle()

Creates an object of EdgeWeightedDirectedCycle and then tries to find a negative cycle in the graph.

hasNegativeCycle() : boolean

Returns a boolean indicating whether or not the graph contains a negative cycle.

negativeCycle() : Iterable<DirectedEdge>

Returns the negative cycle.

distTo(v:Vertex) : double
Rreturns the distance to v.

hasPathTo(v: Vertex) : boolean
Returns a boolean whether or not there is a path to vertex v.

pathTo(v:Vertex) : Stack<Vertex>
Returns the Vertices connected in the shortest path to v.

check(g : Graph, s: Vertex) : boolean
Checks that if a negative cycle exists it is legal and connected.

MID

CLASS: BellmanFord

To traverse the graph in order to find the shortest path based on least amount of flight delay time. Based on the idea of relaxing the edges of the graph and occasionally checking the graph for negative cycles.

.

USES:

Graph
Created the graph to hold all the information.(Vertices and Edges)

Vertex
Mapping the vertices from the Graph.

edu.princeton.cs.algs4.Queue;

edu.princeton.cs.algs4.Stack;

VARIABLES:

cycle: Iterable<DirectedEdge>
To hold a cycle between vertices in the Graph.

queue: Queue<Vertex>
To check Vertices that are in the queue so that we can check them in a FIFO order.

g: Graph
The Graph we are looking to find the shortest path in.

cost: int
Incremented and used to occasionally check the graph for negative cycles.

source: Vertex
The origin we want to use to map. It will be the starting point in which we discover the Vertices that are connected to this Vertex.

ACCESS PROGRAMS:

BellmanFord(g:Graph, s: Vertex)
The constructor for the Object takes a graph and vertex and relaxes the edges in the graph. Starts with s as the source, adds it to the queue and then examines and relaxes all edges from this source. Recursively relaxes all the edges in the graph as they are added to the queue as well.

relax(g: Graph, v:Vertex)
Relaxes the edges in the graph updating them to map out the shortest path. Examines the distance to Vertices and compares it in order to map out the edges that will be the shortest path.

findNegativeCycle()
Creates an object of EdgeWeightedDirectedCycle and then tries to find a negative cycle in the graph. See documentation for EdgeWeightedDirectedCycle.

hasNegativeCycle() : boolean
Returns a boolean indicating whether or not the graph contains a negative cycle by returning whether cycle is null or populated.

negativeCycle() : Iterable<DirectedEdge>
Returns the negative cycle if it exists.

distTo(v:Vertex) : double
Rreturns the distance to v using the Vertex class distTo() function on v.

hasPathTo(v: Vertex) : boolean
Returns a boolean whether or not there is a path to vertex v. If v's distance is still positive infinity then it hasn't been discovered on a path and there is no path to this Vertex.

pathTo(v:Vertex) : Stack<Vertex>
Returns the Vertices connected in the shortest path to v if it exists.

check(g : Graph, s: Vertex) : boolean
Checks that if a negative cycle exists it is legal and connected by checking that the end of the cycle is truly mapping to the start of the cycle.

MIS

CLASS: Sort

Using merge sort to sort all the airport codes.

USES:

N/A

VARIABLES:

N/A

ACCESS PROGRAMS:

sort(a: ArrayList<String>)

Sort a String ArrayList a.

MID

CLASS: Sort

Using merge sort to sort all the Flight objects based on their arrival delays.

USES:

N/A

VARIABLES:

aux: ArrayList<String>

Auxiliary array for merges.

ACCESS PROGRAMS:

exch(a: ArrayList<String>, i: int, j: int)

Exchange the element i in the ArrayList a with element j by using a temporary String t.

less(v: String, w: String): boolean

Compare two Strings v and w in lexical order. Return true if v is lexically less than w, otherwise, return false.

merge(a: ArrayList<String>, lo: int, mid: int, hi: int)

This method merges by first copying into the auxiliary array aux[] then merging back to a[]. In the merge (the second for loop), there are four conditions: left half exhausted (take from the right), right half exhausted (take from the left), current key on right less than current key on left (take from the right), and current key on right greater than or equal to current key on left (take from the left).

sort(a: ArrayList<String>, lo: int, hi: int)

To sort a subarray a[lo..hi] we divide it into two parts: a[lo..mid] and a[mid+1..hi], sort them independently (via recursive calls), and merge the resulting ordered subarrays to produce the result.

sort(a: ArrayList<String>)
Sort the input ArrayList a by calling sort method.

Uses Relationship

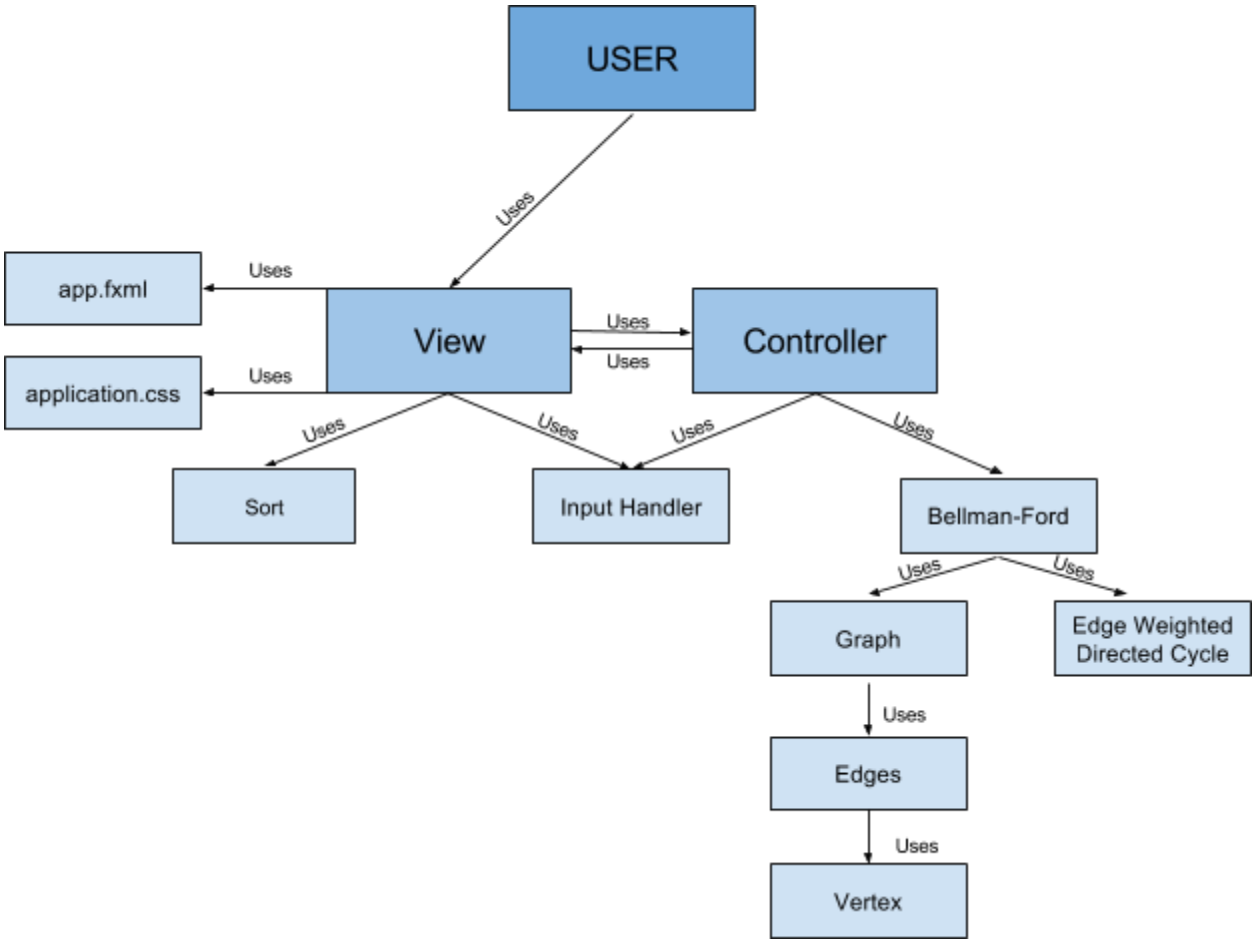

Figure 2:Uses Relationship

Traceback

Requirement	Module
Allow users to choose departure cities (airport code)	View
Sorts airports into Alphabetical order for User	Sort
Direct search to find average delay between cities	BellmanFord
Displays shortest path to UI	View
Find a Vertex in the Graph	Graph (Implementing a BST)
Reads from the dataset csv file	inputHandler
Finds Cycle	EdgeWeightedDirectedCycle
Creates DirectedEdges between the Vertices	Graph (addEdges)
Shows the User their Path (After pathfinding)	Controller, View
Gets the User selections for origin and destination	View
Adding Airports as Vertices	Graph, Vertex

For each class, a description of the implementation (private entities), including class variables in enough detail to show how the class variables are maintained by the methods in the class. 2 UML state machine diagrams for the two most interesting classes

Internal Review

When we set up our project we had scalability in mind because we wanted this to be a relevant implementation for other countries as well as the United States. Also we would want international flights to be considered as well as flights within the US.

The Vertices were implemented to hold a lot of information in order to be able to dfs, bfs, dijkstra's path and Bellman-Ford. We could definitely have been better off creating a symbol table and then just implementing everything else the same way as the textbook but we thought the OOP way would prove to be more versatile.

We initially implemented Dijkstra's Algorithm for our shortest path algorithm but then changed it to the Bellman-Ford algorithm because we realized, during debugging, that some of the delay times are negative (implying an early departure or arrival). In the end we concluded that our design is not very practical for scaling up since there is ~312 Vertices and ~470,000 edges between the vertices. The relaxing of the edges takes too much time to make the application a practical resource. There could be a negative cycle that the program is stuck on that creates an issue relaxing.