Project 4 - Graphs
CSCI 230 T Th 11:10 am
Compiler: g++
OS: Windows 10/WSL

Michael Morikawa

June 4, 2020

Notes

Status

Required options completed and working

Extra Credit

Completed extra credit by implement option 6 (Least amount of stops) and option 9(Remove Airport)

Design Decisions

For the delete flight one I assumed that it be only a flight from the source to destination airport. So for the test case there was no flight to delete since there was no direct flight from LAX to SFO. If I had to consider both directions, then what I could have done would be to check in one direction and if there are no edges try the other direction instead. Also, for the adding a flight, I interepreted the replacing as only being done if the flight id mathced so I allowed multiple edges between vertices.

Data

Output

```
1. Display airport information
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
6. Fewest stops
7. All flights from one airport to another
8. Order to visit all airports from one airport
9. Delete airport
Q. Exit

Please Select from Above Options: 0

Vertex LAX
[outgoing] 2 adjacencies:
(SEA, UA1234, 199.99)
(DFW, AA1000, 189)

[incoming] 3 adjacencies:
(SFO, UA2000, 79)
(DFW, AA2000, 199)
(MSY, SW6000, 190)

Vertex SFO
[outgoing] 1 adjacencies:
(LAX, UA2000, 79)

[incoming] 1 adjacencies:
(DFW, DL4321, 99.99)

Vertex DFW
[outgoing] 2 adjacencies:
(LAX, AA2000, 199)
(SFO, DL4321, 99.99)

[incoming] 3 adjacencies:
(LAX, AA1000, 189)
(ORD, UA9999, 50)
(MSY, SW7654, 109)

Vertex ORD
[outgoing] 2 adjacencies:
(DFW, UA9999, 50)
(BOS, UA3000, 179)

[incoming] 3 adjacencies:
(BOS, UA4000, 149)
(JFK, JB5432, 99)
(SEA, UA5430, 179.5)

Vertex BOS
[outgoing] 2 adjacencies:
(ORD, UA4000, 149)
(JFK, JB2345, 99)
```

```
[incoming] 1 adjacencies:
(ORD, UA3000, 179)

Vertex JFK
[outgoing] 3 adjacencies:
(ORD, JB5432, 99)
(MIA, UA5000, 49)
(MSY, DL3555, 220)

[incoming] 1 adjacencies:
(BOS, JB2345, 99)

Vertex MIA
[outgoing] 1 adjacencies:
(MSY, DL6789, 50)

[incoming] 1 adjacencies:
(JFK, UA5000, 49)

Vertex MSY
[outgoing] 2 adjacencies:
(LAX, SW6000, 190)
(DFW, SW7654, 109)

[incoming] 2 adjacencies:
(JFK, DL3555, 220)
(MIA, DL6789, 50)

Vertex SEA
[outgoing] 1 adjacencies:
(ORD, UA5430, 179.5)

[incoming] 1 adjacencies:
(LAX, UA1234, 199.99)

1. Display airport information
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
6. Fewest stops
7. All flights from one airport to another
8. Order to visit all airports from one airport
9. Delete airport
Q. Exit

Please Select from Above Options: 1

Please enter an airport code: SFO
SFO:     San Francisco
1 Departing Flights:
Flight ID: UA2000
Flight Destination: LAX
Flight Cost: $79.00

1. Display airport information
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
```

```
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 2
 Please enter source airport then destination airport codes: LAX JFK
LAX —— UA1234 —> SEA —— UA5430 —> ORD —— UA3000 —> BOS —— JB2345 —> JFK
 Cost: $657.49

 1. Display airport information
 2. Cheapest flight from one airport to another
 3. Cheapest roundrtip from one airport to another
 4. Add a flight
 5. Delete a flight
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 2
 Please enter source airport then destination airport codes: JFK LAX
JFK —— UA5000 —> MIA —— DL6789 —> MSY —— SW6000 —> LAX
 Cost: $289.00

 1. Display airport information
 2. Cheapest flight from one airport to another
 3. Cheapest roundrtip from one airport to another
 4. Add a flight
 5. Delete a flight
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 5
 Please enter source airport then destination airport codes: LAX SFO
 Flight from LAX to SFO does not exist.

 1. Display airport information
 2. Cheapest flight from one airport to another
 3. Cheapest roundrtip from one airport to another
 4. Add a flight
 5. Delete a flight
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 4
 Please enter source airport then destination airport codes: DFW JFK

 Please Enter Flight number: UA8888
 Please enter the cost: 200.00
 Successfully added flight
 1. Display airport information
 2. Cheapest flight from one airport to another
 3. Cheapest roundrtip from one airport to another
 4. Add a flight
 5. Delete a flight
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 4
 Please enter source airport then destination airport codes: JFK MSY

 Please Enter Flight number: UA7777
 Please enter the cost: 199.00
 Successfully added flight

 1. Display airport information
 2. Cheapest flight from one airport to another
 3. Cheapest roundrtip from one airport to another
 4. Add a flight
 5. Delete a flight
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 3
 Please enter source airport then destination airport codes: LAX JFK
LAX —— AA1000 —> DFW —— UA8888 —> JFK
 Return: JFK —— UA5000 —> MIA —— DL6789 —> MSY —— SW6000 —> LAX
 Cost: $678.00

 1. Display airport information
 2. Cheapest flight from one airport to another
 3. Cheapest roundrtip from one airport to another
 4. Add a flight
 5. Delete a flight
 6. Fewest stops
 7. All flights from one airport to another
 8. Order to visit all airports from one airport
 9. Delete airport
Q. Exit

 Please Select from Above Options: 0

 Vertex LAX
 [outgoing] 2 adjacencies:
 (SEA, UA1234, 199.99)
 (DFW, AA1000, 189.00)

 [incoming] 3 adjacencies:
 (SFO, UA2000, 79.00)
 (DFW, AA2000, 199.00)
 (MSY, SW6000, 190.00)

 Vertex SFO
 [outgoing] 1 adjacencies:
```

```
(LAX, UA2000, 79.00)

[incoming] 1 adjacencies:
(DFW, DL4321, 99.99)

Vertex DFW
[outgoing] 3 adjacencies:
(LAX, AA2000, 199.00)
(SFO, DL4321, 99.99)
(JFK, UA8888, 200.00)

[incoming] 3 adjacencies:
(LAX, AA1000, 189.00)
(ORD, UA9999, 50.00)
(MSY, SW7654, 109.00)

Vertex ORD
[outgoing] 2 adjacencies:
(DFW, UA9999, 50.00)
(BOS, UA3000, 179.00)

[incoming] 3 adjacencies:
(BOS, UA4000, 149.00)
(JFK, JB5432, 99.00)
(SEA, UA5430, 179.50)

Vertex BOS
[outgoing] 2 adjacencies:
(ORD, UA4000, 149.00)
(JFK, JB2345, 99.00)

[incoming] 1 adjacencies:
(ORD, UA3000, 179.00)

Vertex JFK
[outgoing] 4 adjacencies:
(ORD, JB5432, 99.00)
(MIA, UA5000, 49.00)
(MSY, DL3555, 220.00)
(MSY, UA7777, 199.00)

[incoming] 2 adjacencies:
(BOS, JB2345, 99.00)
(DFW, UA8888, 200.00)

Vertex MIA
[outgoing] 1 adjacencies:
(MSY, DL6789, 50.00)

[incoming] 1 adjacencies:
(JFK, UA5000, 49.00)

Vertex MSY
[outgoing] 2 adjacencies:
(LAX, SW6000, 190.00)
(DFW, SW7654, 109.00)

[incoming] 3 adjacencies:
(JFK, DL3555, 220.00)
(MIA, DL6789, 50.00)
```

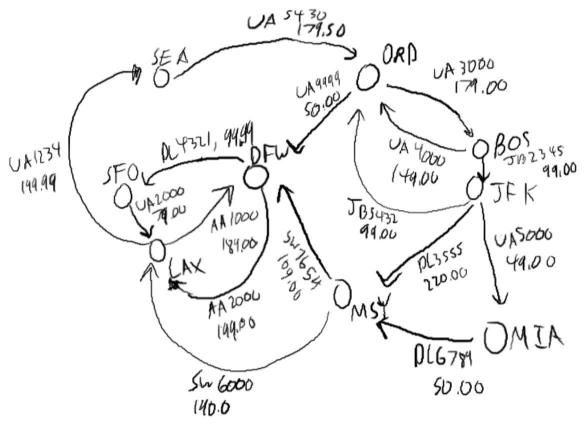
```
(JFK, UA7777, 199.00)

Vertex SEA
[outgoing] 1 adjacencies:
(ORD, UA5430, 179.50)

[incoming] 1 adjacencies:
(LAX, UA1234, 199.99)

1. Display airport information
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
6. Fewest stops
7. All flights from one airport to another
8. Order to visit all airports from one airport
9. Delete airport
Q. Exit

Please Select from Above Options: 6
Please enter source airport then destination airport codes: JFK SFO
JFK ---- JB5432 ---> ORD ---- UA9999 ---> DFW ---- DL4321 ---> SFO
Cost: $248.99

1. Display airport information
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
6. Fewest stops
7. All flights from one airport to another
8. Order to visit all airports from one airport
9. Delete airport
Q. Exit

Please Select from Above Options: 9
Please enter airport code that you wish to delete: MSY
Successfully deleted airport

1. Display airport information
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
6. Fewest stops
7. All flights from one airport to another
8. Order to visit all airports from one airport
9. Delete airport
Q. Exit

Please Select from Above Options: 0

Vertex LAX
[outgoing] 2 adjacencies:
(SEA, UA1234, 199.99)
(DFW, AA1000, 189.00)

[incoming] 2 adjacencies:
(SFO, UA2000, 79.00)
```

```
(DFW, AA2000, 199.00)

Vertex SFO
[outgoing] 1 adjacencies:
(LAX, UA2000, 79.00)

[incoming] 1 adjacencies:
(DFW, DL4321, 99.99)

Vertex DFW
[outgoing] 3 adjacencies:
(LAX, AA2000, 199.00)
(SFO, DL4321, 99.99)
(JFK, UA8888, 200.00)

[incoming] 2 adjacencies:
(LAX, AA1000, 189.00)
(ORD, UA9999, 50.00)

Vertex ORD
[outgoing] 2 adjacencies:
(DFW, UA9999, 50.00)
(BOS, UA3000, 179.00)

[incoming] 3 adjacencies:
(BOS, UA4000, 149.00)
(JFK, JB5432, 99.00)
(SEA, UA5430, 179.50)

Vertex BOS
[outgoing] 2 adjacencies:
(ORD, UA4000, 149.00)
(JFK, JB2345, 99.00)

[incoming] 1 adjacencies:
(ORD, UA3000, 179.00)

Vertex JFK
[outgoing] 2 adjacencies:
(ORD, JB5432, 99.00)
(MIA, UA5000, 49.00)

[incoming] 2 adjacencies:
(BOS, JB2345, 99.00)
(DFW, UA8888, 200.00)

Vertex MIA
[outgoing] 0 adjacencies:

[incoming] 1 adjacencies:
(JFK, UA5000, 49.00)

Vertex SEA
[outgoing] 1 adjacencies:
(ORD, UA5430, 179.50)

[incoming] 1 adjacencies:
(LAX, UA1234, 199.99)

1. Display airport information
```

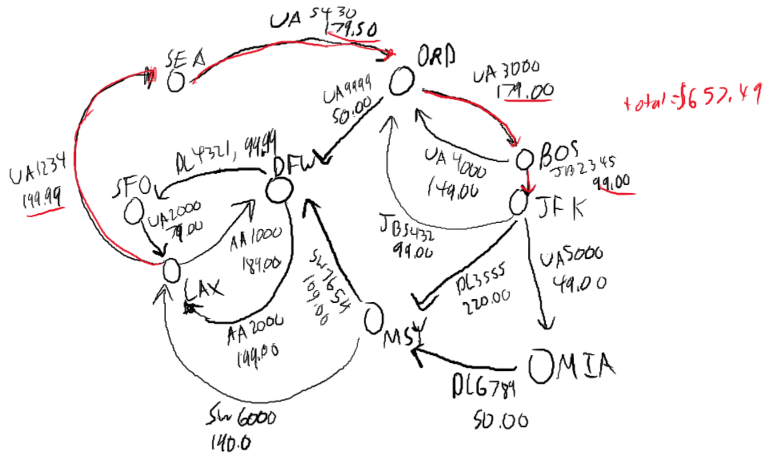
```
2. Cheapest flight from one airport to another
3. Cheapest roundrtip from one airport to another
4. Add a flight
5. Delete a flight
6. Fewest stops
7. All flights from one airport to another
8. Order to visit all airports from one airport
9. Delete airport
Q. Exit

Please Select from Above Options: Q
```

Graphs

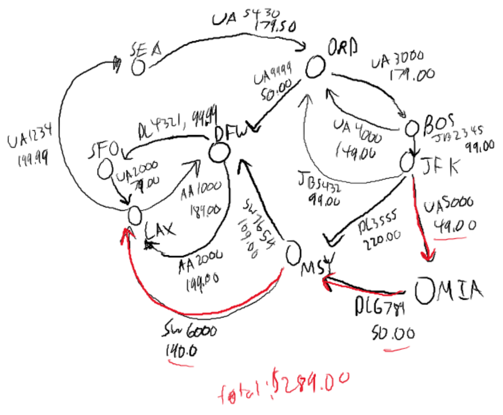
Original Graph



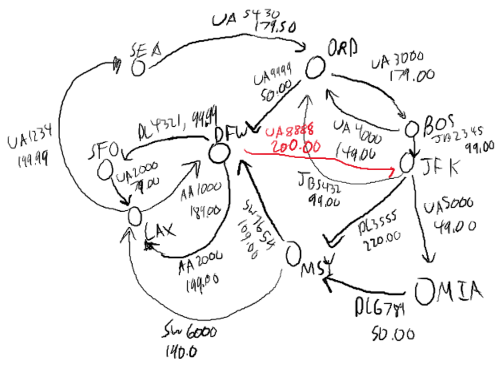
Test Case 3



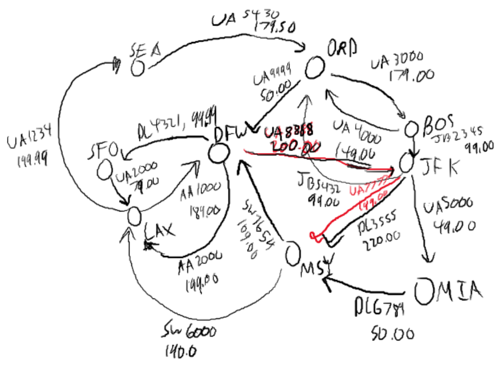
Test Case 4



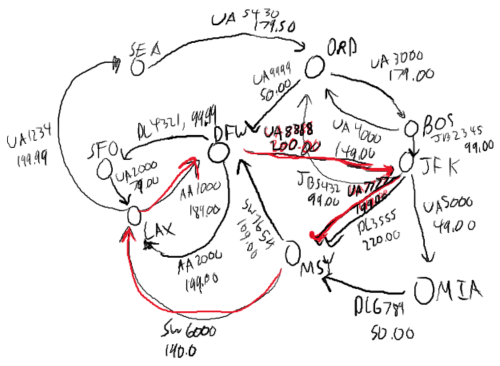
Test Case 6



Test Case 7

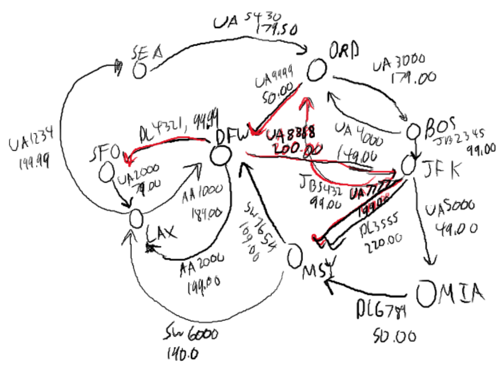


Test Case 8



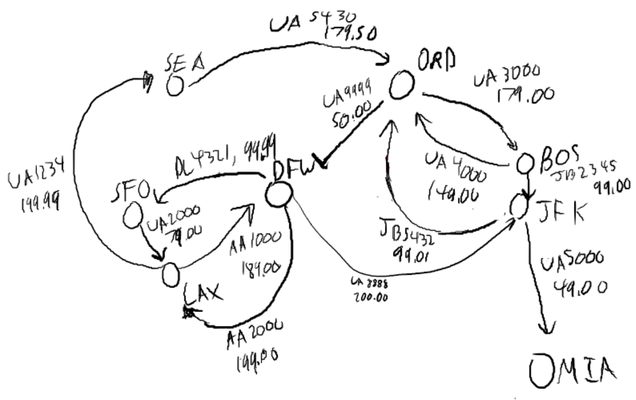
Total cost: $678.00

Test Case EC Option 6



2 stops
$248.99

Final Graph and Test Case EC Option 9



Remove MSY

Source Code

main.cpp

```
#include <map>
#include <iomanip>
#include <fstream>
#include "AdjacencyListGraph.hpp"

void loadVertices(std::string filepath, std::map<std::string, Vertex *> &vertexMap,
                  std::map<std::string, std::string> &nameMap, AdjacencyListGraph &G);

void loadEdges(std::string filepath, std::map<std::string, Edge *> &edgeMap,
               std::map<std::string, Vertex *> &vertexMap, AdjacencyListGraph &G);

void cheapestFlight(Vertex *src, Vertex *dest, AdjacencyListGraph &G, bool weighted = true);

void cheapestRoundTrip(Vertex *src, Vertex *dest, AdjacencyListGraph &G);

void updateFlights(AdjacencyListGraph &G);

void updateAirports(std::map<std::string, Vertex *> &vertexMap);

void printMenu();

int main()
{
    AdjacencyListGraph airportNetwork(true);
    std::map<std::string, Vertex *> vertexInfo;
    std::map<std::string, std::string> airportNames;
    std::map<std::string, Edge *> edgeInfo;
    loadVertices("docs/P4Airports.txt", vertexInfo, airportNames, airportNetwork);
    loadEdges("docs/P4Flights.txt", edgeInfo, vertexInfo, airportNetwork);
    int choice = 0;
    bool running = true;
    while (running)
    {
        printMenu();
        std::cin >> choice;
        std::cout << choice << '\n'; //for file redirection
        if (std::cin.fail())
        {
            std::cin.clear();
            std::cin.ignore();
            break;
        }
        switch (choice)
        {
        case 0:
        {
            airportNetwork.print();
        }
        break;
        case 1:
        {
            std::string code;
            std::cout << std::setprecision(2) << std::fixed;
            std::cout << "\nPlease enter an airport code: ";
            std::cin >> code;
            std::cout << code << '\n'; //file redirection
            if (vertexInfo.find(code) != vertexInfo.end())
            {
                Vertex *airport = vertexInfo[code];
                std::cout << airport->getElement() << ": " << airportNames[code]
                          << "\n"
                          << airportNetwork.outDegree(airport) << " Departing Flights: ";
                for (auto e : airportNetwork.outgoingEdges(airport))
                {
                    std::cout << "\nFlight ID: " << e->getElement()
                              << "\nFlight Destination: "
                              << airportNetwork.opposite(airport, e)->getElement()
                              << "\nFlight Cost: $" << airportNetwork.getWeight(e);
                }
                std::cout << "\n";
            }
            else
            {
                std::cout << "Could not find that airport\n";
            }
        }
        break;
        case 2:
        {
            std::string code1, code2;
            std::cout << "Please enter source airport then destination airport codes: ";
            std::cin >> code1 >> code2;
            std::cout << code1 << ' ' << code2 << '\n'; //file redirction
            if (vertexInfo.find(code1) != vertexInfo.end() && vertexInfo.find(code2) != vertexInfo.end())
            {
                Vertex *source = vertexInfo[code1];
                Vertex *dest = vertexInfo[code2];
                cheapestFlight(source, dest, airportNetwork);
            }
            else
            {
                std::cout << "Could not find inputted airports\n";
            }
        }
        break;
        case 3:
        {
            std::string code1, code2;
            std::cout << "Please enter source airport then destination airport codes: ";
            std::cin >> code1 >> code2;
            std::cout << code1 << ' ' << code2 << '\n'; //file redirction
            if (vertexInfo.find(code1) != vertexInfo.end() && vertexInfo.find(code2) != vertexInfo.end())
            {
                Vertex *source = vertexInfo[code1];
                Vertex *dest = vertexInfo[code2];
                cheapestRoundTrip(source, dest, airportNetwork);
            }
            else
            {
                std::cout << "Could not find inputted airports\n";
            }
        }
        break;
        case 4:
        {
            std::string code1, code2, flightCode;
```

```
            double cost;
            std::cout << "Please enter source airport then destination airport codes: ";
            std::cin >> code1 >> code2;
            std::cout << code1 << ' ' << code2 << '\n'; //file redirction
            std::cout << "\nPlease Enter Flight number: ";
            std::cin >> flightCode;
            std::cout << flightCode << '\n';
            std::cout << "Please enter the cost: ";
            std::cin >> cost;
            std::cout << cost << '\n';
            if (vertexInfo.find(code1) != vertexInfo.end() && vertexInfo.find(code2) != vertexInfo.end())
            {
                Vertex *source = vertexInfo[code1];
                Vertex *dest = vertexInfo[code2];
                if (edgeInfo.find(flightCode) == edgeInfo.end())
                {
                    Edge *e = airportNetwork.insertEdge(source, dest, flightCode, cost);
                    edgeInfo[flightCode] = e;
                    std::cout << "Successfully added flight\n";
                }
                else
                {
                    airportNetwork.setWeight(edgeInfo[flightCode], cost);
                    std::cout << "Successfully updated flight\n";
                }
            }
            else
            {
                std::cout << "Could not find inputted airports\n";
            }
        }
        break;
        case 5:
        {
            std::string code1, code2;

            std::cout << "Please enter source airport then destination airport codes: ";
            std::cin >> code1 >> code2;
            std::cout << code1 << ' ' << code2 << '\n'; //file redirction
            if (vertexInfo.find(code1) != vertexInfo.end() && vertexInfo.find(code2) != vertexInfo.end())
            {
                Vertex *source = vertexInfo[code1];
                Vertex *dest = vertexInfo[code2];
                Edge *e = airportNetwork.getEdge(source, dest);
                if (e != nullptr)
                {
                    edgeInfo.erase(e->getElement());
                    airportNetwork.removeEdge(e);
                    std::cout << "Successfully deleted flight\n";
                }
                else
                {
                    std::cout << "Flight from " << code1 << " to " << code2 << " does not exist.\n";
                }
            }
            else
            {
                std::cout << "Could not find inputted airports\n";
            }
        }
        break;
        case 6:
        {
            std::string code1, code2;
            std::cout << "Please enter source airport then destination airport codes: ";
            std::cin >> code1 >> code2;
            std::cout << code1 << ' ' << code2 << '\n'; //file redirction
            if (vertexInfo.find(code1) != vertexInfo.end() && vertexInfo.find(code2) != vertexInfo.end())
            {
                Vertex *source = vertexInfo[code1];
                Vertex *dest = vertexInfo[code2];
                cheapestFlight(source, dest, airportNetwork, false); //false to use all weights = 1
            }
            else
            {
                std::cout << "Could not find inputted airports\n";
            }
        }
        break;
        case 9:
        {
            std::string code;
            std::cout << "Please enter airport code that you wish to delete: ";
            std::cin >> code;
            std::cout << code << '\n'; //file redirection
            if (vertexInfo.find(code) != vertexInfo.end())
            {
                airportNetwork.removeVertex(vertexInfo[code]);
                vertexInfo.erase(code);
                std::cout << "Successfully deleted airport\n";
            }
            else
            {
                std::cout << "Could not find inputted airport\n";
            }
        }
        break;

        default:
        {
            std::cout << "Option unavailable\n";
        }
        break;
        }
    }

    updateAirports(vertexInfo);
    updateFlights(airportNetwork);
}

void loadVertices(std::string filepath, std::map<std::string, Vertex *> &vertexMap,
                  std::map<std::string, std::string> &nameMap, AdjacencyListGraph &G)
{
    std::ifstream infile(filepath);
    std::string code, name;
    while (true)
    {
        if (infile.eof())
        {
            break;
        }
        infile >> code;
        std::getline(infile, name);
        vertexMap[code] = G.insertVertex(code);
        nameMap[code] = name;
    }
}

void loadEdges(std::string filepath, std::map<std::string, Edge *> &edgeMap,
               std::map<std::string, Vertex *> &vertexMap, AdjacencyListGraph &G)
{
    std::ifstream infile(filepath);
    std::string src, dest, code;
    Vertex *source, *destination;
    double cost;
    while (true)
    {
        if (infile.eof())
        {
            break;
        }
        infile >> src >> dest >> cost >> code;
        source = vertexMap[src];
        destination = vertexMap[dest];
        G.insertEdge(source, destination, code, cost);
    }
}
void cheapestFlight(Vertex *src, Vertex *dest, AdjacencyListGraph &G, bool weighted)
{
    double cost = G.shortestPath(src, dest, weighted);
    G.printShortestPath(src, dest);
    std::cout << std::setprecision(2) << std::fixed;
    std::cout << "Cost: $" << cost << '\n';
}
void cheapestRoundTrip(Vertex *src, Vertex *dest, AdjacencyListGraph &G)
{
    double cost = G.shortestPath(src, dest);
    G.printShortestPath(src, dest);
    std::cout << "Return: ";
    cost += G.shortestPath(dest, src);
    G.printShortestPath(dest, src);
    std::cout << std::setprecision(2) << std::fixed;
    std::cout << "Cost: $" << cost << '\n';
}

void printMenu()
{
    std::cout << "\n1. Display airport information\n"
              << "2. Cheapest flight from one airport to another\n"
              << "3. Cheapest roundrtip from one airport to another\n"
              << "4. Add a flight\n"
              << "5. Delete a flight\n"
              << "6. Fewest stops\n"
              << "7. All flights from one airport to another\n"
              << "8. Order to visit all airports from one airport\n"
              << "9. Delete airport\n"
              << "Q. Exit\n"
              << "\nPlease Select from Above Options: ";
}
```

```
void updateAirports(std::map<std::string, Vertex *> &vertexMap)
{
    std::ofstream outfile("docs/P4AirportsRev1.txt");
    for (auto p : vertexMap)
    {
        outfile << p.second->getElement() << "    " << p.first << '\n';
    }
}

void updateFlights(AdjacencyListGraph &G)
{
    std::ofstream outfile("docs/P4FlightsRev1.txt");
    std::vector<Vertex *> endpoints;
    outfile << std::setprecision(2) << std::fixed;
    for (auto e : G.getEdges())
    {
        endpoints = G.endVertices(e);
        outfile << endpoints[0]->getElement() << "    " << endpoints[1]->getElement()
                << std::setw(10) << std::right
                << G.getWeight(e) << "    " << e->getElement() << '\n';
    }
}
```

Graph.h

```
#pragma once
#include <vector>
#include <list>
#include <string>

// Created by T. Vo for CSCI 230
// Based on Java version of Goodrich book w/o template
// string for vertex and int for edge
//Modified by Michael Morikawa for Proj 4

class Vertex // behaves like interface in Java
{
public:
        virtual std::string getElement() = 0;
};

class Edge // behaves like interface in Java
{
public:
        virtual std::string getElement() = 0;
};

class Graph
{
public:
        /* Returns the number of vertices of the graph */
        virtual int numVertices() = 0;

        /* Returns the number of edges of the graph */
        virtual int numEdges() = 0;

        /* Returns the vertices of the graph as an iterable collection */
        virtual std::list<Vertex *> getVertices() = 0;

        /* Returns the edges of the graph as an iterable collection */
```

```
virtual std::list<Edge *> getEdges() = 0;

/*
* Returns the number of edges leaving vertex v.
* Note that for an undirected graph, this is the same result
* returned by inDegree
* throws IllegalArgumentException if v is not a valid vertex?
*/
virtual int outDegree(Vertex *v) = 0; // throws IllegalArgumentException;

/**
* Returns the number of edges for which vertex v is the destination.
* Note that for an undirected graph, this is the same result
* returned by outDegree
* throws IllegalArgumentException if v is not a valid vertex
*/
virtual int inDegree(Vertex *v) = 0; // throws IllegalArgumentException;

/*
* Returns an iterable collection of edges for which vertex v is the origin.
* Note that for an undirected graph, this is the same result
* returned by incomingEdges.
* throws IllegalArgumentException if v is not a valid vertex
*/
virtual std::vector<Edge *> outgoingEdges(Vertex *v) = 0; // throws IllegalArgumentException;

/*
* Returns an iterable collection of edges for which vertex v is the destination.
* Note that for an undirected graph, this is the same result
* returned by outgoingEdges.
* throws IllegalArgumentException if v is not a valid vertex
*/
virtual std::vector<Edge *> incomingEdges(Vertex *v) = 0; // throws IllegalArgumentException;

/** Returns the edge from u to v, or null if they are not adjacent. */
virtual Edge *getEdge(Vertex *u, Vertex *v) = 0; // throws IllegalArgumentException;

/*
* Returns the vertices of edge e as an array of length two.
* If the graph is directed, the first vertex is the origin, and
* the second is the destination.  If the graph is undirected, the
* order is arbitrary.
*/
virtual std::vector<Vertex *> endVertices(Edge *e) = 0; // throws IllegalArgumentException;

/* Returns the vertex that is opposite vertex v on edge e. */
virtual Vertex *opposite(Vertex *v, Edge *e) = 0; // throws IllegalArgumentException;

/* Inserts and returns a new vertex with the given element. */
virtual Vertex *insertVertex(std::string element) = 0;

/*
* Inserts and returns a new edge between vertices u and v, storing given element.
*
* throws IllegalArgumentException if u or v are invalid vertices, or if an edge already exists between
*/
virtual Edge *insertEdge(Vertex *u, Vertex *v, std::string element, double weight) = 0; // throws Ille

/* Removes a vertex and all its incident edges from the graph. */
virtual void removeVertex(Vertex *v) = 0; // throws IllegalArgumentException;
```

```
        /* Removes an edge from the graph. */
        virtual void removeEdge(Edge *e) = 0; // throws IllegalArgumentException;

        virtual void print() = 0;
};
```

AdjacencyListGraph.hpp

```
#pragma once
#include <iostream>
#include <queue>
#include <list>
#include <vector>
#include "Graph.h"

// Created by T. Vo for CSCI 230
// Based on Java version of Goodrich book w/o template
// and minimal exception handling
// Some operations are incomplete and there are provisions
// to change from map to a list/vector for adjacency list
//Modified by Michael Morikawa for Proj 4

#define INF (double)99999999

class AdjacencyListGraph : public Graph
{
private:
    bool isDirected;
    std::list<Vertex *> vertices;
    std::list<Edge *> edges;

    /* A vertex of an adjacency list graph representation. */
    class InnerVertex : public Vertex
    {
    private:
        std::string element;
        //Position<Vertex<V>> pos;d
        Vertex *pos;
        InnerVertex *predecessor;
        double distance;

        std::list<std::pair<Vertex *, Edge *>> *outgoing;
        std::list<std::pair<Vertex *, Edge *>> *incoming;

    public:
        /* Constructs a new InnerVertex instance storing the given element. */
        InnerVertex(std::string elem, bool graphIsDirected = false)
        {
            element = elem;
            outgoing = new std::list<std::pair<Vertex *, Edge *>>();
            if (graphIsDirected)
                incoming = new std::list<std::pair<Vertex *, Edge *>>();
            else
                incoming = outgoing; // if undirected, alias outgoing map
        }

        /* Returns the element associated with the vertex. */
        std::string getElement()
        {
```

```
        return element;
    }

    /* Stores the position of this vertex within the graph's vertex list. */
    void setPosition(Vertex *p)
    {
        pos = p;
    }

    /* Returns the position of this vertex within the graph's vertex list. */
    Vertex *getPosition()
    {
        return pos;
    }

    //returns list of outgoing edges
    std::list<std::pair<Vertex *, Edge *>> *getOutgoing()
    {
        return outgoing;
    }

    /* Returns list of incoming edges. */
    std::list<std::pair<Vertex *, Edge *>> *getIncoming()
    {
        return incoming;
    }

    InnerVertex *getPredecessor()
    {
        return predecessor;
    }

    void setPredecessor(InnerVertex *u)
    {
        predecessor = u;
    }

    double getDistance()
    {
        return distance;
    }

    void setDistance(double n)
    {
        distance = n;
    }

}; //------------ end of InnerVertex class ------------

struct VertexComp
{
    bool operator()(InnerVertex *a, InnerVertex *b)
    {
        return a->getDistance() > b->getDistance();
    }
};

//---------------- nested InnerEdge class ----------------
/* An edge between two vertices. */
class InnerEdge : public Edge
```

```
    {
    private:
        std::string element;
        double weight;
        Edge *pos;
        std::vector<Vertex *> endpoints;

    public:
        /* Constructs InnerEdge instance from u to v, storing the given element. */
        InnerEdge(Vertex *u, Vertex *v, std::string elem, double w)
        {
            element = elem;
            weight = w;
            endpoints.push_back(u);
            endpoints.push_back(v);
        }

        /* Returns the element associated with the edge. */
        std::string getElement()
        {
            return element;
        }

        double getWeight()
        {
            return weight;
        }

        void setWeight(double w)
        {
            weight = w;
        }

        /* Returns reference to the endpoint array. */
        std::vector<Vertex *> getEndpoints()
        {
            return endpoints;
        }

        /* Stores the position of this edge within the graph's vertex list. */
        void setPosition(Edge *p)
        {
            pos = p;
        }

        /* Returns the position of this edge within the graph's vertex list. */
        Edge *getPosition()
        {
            return pos;
        }
    }; //------------ end of InnerEdge class ------------

public:
    /*
        * Constructs an empty graph.
        * The parameter determines whether this is an undirected or directed graph.
        */
    AdjacencyListGraph(bool directed = false)
    {
        isDirected = directed;
    }

    ~AdjacencyListGraph()
    {
        // should deallocate memory here
    }

    /* Returns the number of vertices of the graph */
    int numVertices()
    {
        return static_cast<int>(vertices.size());
    }

    /* Returns the number of edges of the graph */
    int numEdges()
    {
        return static_cast<int>(edges.size());
    }

    /* Returns the vertices of the graph as an iterable collection */
    std::list<Vertex *> getVertices()
    {
        return vertices;
    }

    /* Returns the edges of the graph as an iterable collection */
    std::list<Edge *> getEdges()
    {
        return edges;
    }

    /*
        * Returns the number of edges leaving vertex v.
        * Note that for an undirected graph, this is the same result
        * returned by inDegree
        * throws IllegalArgumentException if v is not a valid vertex?
        */
    int outDegree(Vertex *v) // throws IllegalArgumentException;
    {
        InnerVertex *vert = static_cast<InnerVertex *>(v);
        return static_cast<int>(vert->getOutgoing()->size());
    }

    /**
        * Returns the number of edges for which vertex v is the destination.
        * Note that for an undirected graph, this is the same result
        * returned by outDegree
        * throws IllegalArgumentException if v is not a valid vertex
        */
    int inDegree(Vertex *v) // throws IllegalArgumentException;
    {
        InnerVertex *vert = static_cast<InnerVertex *>(v);
        return static_cast<int>(vert->getIncoming()->size());
    }

    /*
        * Returns an iterable collection of edges for which vertex v is the origin.
        * Note that for an undirected graph, this is the same result
        * returned by incomingEdges.
        * throws IllegalArgumentException if v is not a valid vertex
        */
    std::vector<Edge *> outgoingEdges(Vertex *v) // throws IllegalArgumentException;
    {
        std::vector<Edge *> temp;
        std::list<std::pair<Vertex *, Edge *>> *listPtr = static_cast<InnerVertex *>(v)->getOutgoing();
        for (auto it = listPtr->begin(); it != listPtr->end(); ++it)
        {
            temp.push_back(it->second);
        }
        return temp;
    }

    /*
        * Returns an iterable collection of edges for which vertex v is the destination.
        * Note that for an undirected graph, this is the same result
        * returned by outgoingEdges.
        * throws IllegalArgumentException if v is not a valid vertex
        */
    std::vector<Edge *> incomingEdges(Vertex *v) // throws IllegalArgumentException;
    {
        std::vector<Edge *> temp;
        std::list<std::pair<Vertex *, Edge *>> *listPtr = static_cast<InnerVertex *>(v)->getIncoming();
        for (auto it = listPtr->begin(); it != listPtr->end(); ++it)
        {
            temp.push_back(it->second);
        }
        return temp;
    }

    /* Returns the edge from u to v, or null if they are not adjacent. */
    Edge *getEdge(Vertex *u, Vertex *v) // throws IllegalArgumentException;
    {
        Edge *temp = nullptr;
        std::list<std::pair<Vertex *, Edge *>> *listPtr = static_cast<InnerVertex *>(u)->getOutgoing();
        auto it = listPtr->begin();
        for (it, it = listPtr->begin(); it != listPtr->end(); it++)
        {
            if (it->first == v)
            {
                break;
            }
        }
        if (it != listPtr->end())
            temp = it->second;
        return temp; // origin.getOutgoing().get(v);    // will be null if no edge from u to v
    }

    /*
        * Returns the vertices of edge e as an array of length two.
        * If the graph is directed, the first vertex is the origin, and
        * the second is the destination.  If the graph is undirected, the
        * order is arbitrary.
        */
    std::vector<Vertex *> endVertices(Edge *e) // throws IllegalArgumentException;
    {
        std::vector<Vertex *> endpoints = static_cast<InnerEdge *>(e)->getEndpoints();
        return endpoints;
    }

    /* Returns the vertex that is opposite vertex v on edge e. */
```

```
Vertex *opposite(Vertex *v, Edge *e) // throws IllegalArgumentException;
{
    std::vector<Vertex *> endpoints = static_cast<InnerEdge *>(e)->getEndpoints();

    if (endpoints[0] == v)
        return endpoints[1];
    else
        return endpoints[0];
}

/* Inserts and returns a new vertex with the given element. */
Vertex *insertVertex(std::string element)
{
    Vertex *v = new InnerVertex(element, isDirected);
    vertices.push_back(v);
    static_cast<InnerVertex *>(v)->setPosition(vertices.back());
    return v;
}

/*
    * Inserts and returns a new edge between vertices u and v, storing given element.
    *
    * throws IllegalArgumentException if u or v are invalid vertices, or if an edge already exists between
    */
Edge *insertEdge(Vertex *u, Vertex *v, std::string element, double weight = 1) // throws IllegalArgumentEx
{
    Edge *e = new InnerEdge(u, v, element, weight);
    edges.push_back(e);
    static_cast<InnerEdge *>(e)->setPosition(edges.back());
    InnerVertex *origin = static_cast<InnerVertex *>(u);
    InnerVertex *dest = static_cast<InnerVertex *>(v);
    (origin->getOutgoing())->push_back(std::pair<Vertex *, Edge *>(v, e));
    (dest->getIncoming())->push_back(std::pair<Vertex *, Edge *>(u, e));

    return e;
}

/* Removes a vertex and all its incident edges from the graph. */
void removeVertex(Vertex *v) // throws IllegalArgumentException;
{
    for (Edge *e : outgoingEdges(v))
    {
        removeEdge(e);
    }
    for (Edge *e : incomingEdges(v))
    {
        removeEdge(e);
    }
    vertices.remove(v);
}

/* Removes an edge from the graph. */
void removeEdge(Edge *e)
{
    if (e == nullptr)
    {
        return;
    }
    std::vector<Vertex *> endpoints = static_cast<InnerEdge *>(e)->getEndpoints();
    static_cast<InnerVertex *>(endpoints[0])->getOutgoing()->remove(std::make_pair(endpoints[1], e));
    static_cast<InnerVertex *>(endpoints[1])->getIncoming()->remove(std::make_pair(endpoints[0], e));
    edges.remove(static_cast<InnerEdge *>(e)->getPosition());
}

double getWeight(Edge *e)
{
    return static_cast<InnerEdge *>(e)->getWeight();
}

void setWeight(Edge *e, double weight)
{
    static_cast<InnerEdge *>(e)->setWeight(weight);
}

double shortestPath(Vertex *src, Vertex *dest, bool weighted = true)
{
    InnerVertex *source = static_cast<InnerVertex *>(src);
    InnerVertex *destination = static_cast<InnerVertex *>(dest);
    std::priority_queue<InnerVertex *, std::vector<InnerVertex *>, VertexComp> PQ;
    for (auto v : vertices)
    {
        static_cast<InnerVertex *>(v)->setDistance(INF);
        static_cast<InnerVertex *>(v)->setPredecessor(0);
    }
    source->setDistance(0);
    PQ.push(source);
    InnerVertex *currentVert;
    InnerVertex *adjVert;
    double adjustedDist;
    double edgeWeight;
    while (!PQ.empty())
    {
        currentVert = PQ.top();
        if (currentVert == destination)
        {
            break;
        }
        PQ.pop();
        for (auto e : outgoingEdges(currentVert))
        {
            if (weighted)
            {
                edgeWeight = static_cast<InnerEdge *>(e)->getWeight();
            }
            else
            {
                edgeWeight = 1;
            }

            adjVert = static_cast<InnerVertex *>(opposite(currentVert, e));
            adjustedDist = currentVert->getDistance() + edgeWeight;
            if (adjustedDist < adjVert->getDistance())
            {
                adjVert->setDistance(adjustedDist);
                adjVert->setPredecessor(currentVert);
                PQ.push(adjVert);
            }
        }
    }
    if (!weighted)
    {
        double total = 0;
        Vertex *current = dest;
        std::list<Vertex *> path;
        while (current != 0)
        {
            path.push_front(current);
            current = static_cast<InnerVertex *>(current)->getPredecessor();
        }
        while (true)
        {
            auto v = path.front();
            if (v == dest)
            {
                break;
            }
            path.pop_front();
            total += getWeight(getEdge(v, path.front()));
        }
        return total;
    }
    return destination->getDistance();
}

void printShortestPath(Vertex *src, Vertex *dest)
{
    Vertex *current = dest;
    std::list<Vertex *> path;
    while (current != 0)
    {
        path.push_front(current);
        current = static_cast<InnerVertex *>(current)->getPredecessor();
    }
    while (true)
    {
        auto v = path.front();
        if (v == dest)
        {
            std::cout << v->getElement() << '\n';
            break;
        }
        std::cout << v->getElement() << " --- ";
        path.pop_front();
        std::cout << getEdge(v, path.front())->getElement() << " --> ";
    }
}

void print()
{
    for (auto itr = vertices.begin(); itr != vertices.end(); itr++)
    {
        std::cout << "\nVertex " << (*itr)->getElement() << '\n';
        if (isDirected)
            std::cout << "[outgoing]";
        std::cout << " " << outDegree(*itr) << " adjacencies:\n";
        for (auto e : outgoingEdges(*itr))
        {
            std::cout << "(" << opposite(*itr, e)->getElement() << ", " << e->getElement() << ", "
                      << getWeight(e) << ")\n";
        }
```

```
            std::cout << '\n';
            if (isDirected)
            {
                std::cout << "[incoming]";
                std::cout << " " << inDegree(*itr) << " adjacencies:\n";
                for (auto e : incomingEdges(*itr))
                {
                    std::cout << "(" << opposite(*itr, e)->getElement() << ", " << e->getElement() << ", "
                              << getWeight(e) << ")\n";
                }
            }
        }
    }
};
```