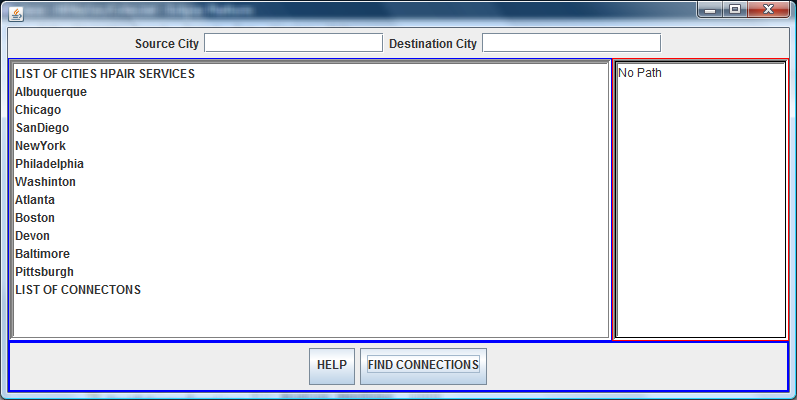
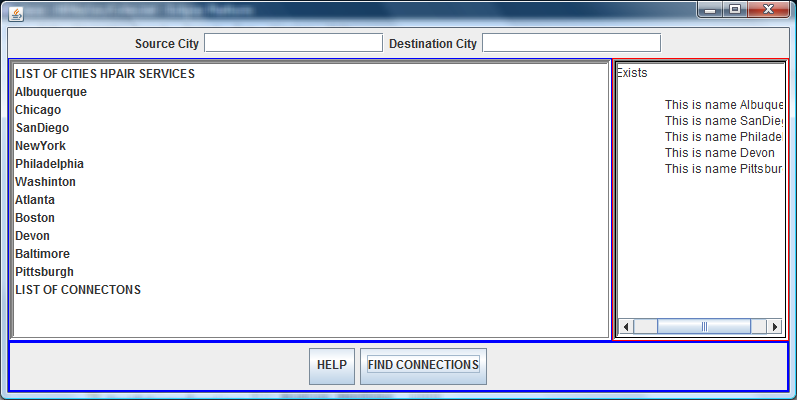
Casey Burkhardt

HPAir Screenshots





**import** java.util.Iterator;

**public** **class** GraphListDirected **extends** Graphlist

{

/\*\*

\* Construct a directed, adjacency-list based graph.

\*

\* **@post** constructs an directed graph

\*/

**public** GraphListDirected()

{

**super**(**true**);

}

/\*\*

\* Add an edge between two vertices within the graph. Edge is directed.

\* Duplicate edges are silently replaced.

\* Labels on edges may be null.

\*

\* **@pre** vLabel1 and vLabel2 are labels of existing vertices, v1 & v2

\* **@post** an edge is inserted between v1 and v2;

\* if edge is new, it is labeled with label (can be null)

\*

\* **@param** vLabel1 Source vertex.

\* **@param** vLabel2 Destination vertex.

\* **@param** label Label associated with the edge.

\*/

**public** **void** addEdge(Object vLabel1, Object vLabel2, Double label)

{

// get vertices with the label parameters

VertexS v1 = (VertexS) dict.get(vLabel1);

VertexS v2 = (VertexS) dict.get(vLabel2);

// create and edge with the vertices

Edge e = **new** Edge(v1.label(), v2.label(), label, **true**);

// add the edge to v1, the source vertex's linked list

v1.addEdge(e);

}

/\*\*

\* Remove a vertex from the graph. Associated edges are also

\* removed. Non-vertices are silently ignored.

\*

\* **@pre** label is non-null vertex label

\* **@post** vertex with "equals" label is removed, if found

\*

\* **@param** label The label of the vertex within the graph.

\* **@return** The label associated with the vertex.

\*/

**public** Object remove(Object label)

{

VertexS v = (VertexS)dict.get(label);

Iterator vi = dict.values().iterator();

**while** (vi.hasNext())

{

Object v2 = vi.next();

**if** (label.equals(v2)) removeEdge(v2,label);

}

dict.remove(label);

**return** v.label();

}

/\*\*

\* Remove possible edge between vertices labeled vLabel1 and vLabel2.

\* vLabel1 is the source.

\*

\* **@pre** vLabel1 and vLabel2 are labels of existing vertices

\* **@post** edge is removed, its label is returned

\*

\* **@param** vLabel1 Source vertex.

\* **@param** vLabel2 Destination vertex.

\* **@return** The label associated with the edge removed.

\*/

**public** Object removeEdge(Object vLabel1, Object vLabel2)

{

VertexS v1 = (VertexS) dict.get(vLabel1);

VertexS v2 = (VertexS) dict.get(vLabel2);

Edge e = **new** Edge(v1.label(), v2.label(), **null**, **true**);

e = v1.removeEdge2(e);

**if** (e == **null**)

**return** **null**;

**else**

**return** e.label();

}

/\*\*

\* Determine the number of edges in graph.

\*

\* **@post** returns the number of edges in graph

\*

\* **@return** Number of edges in graph.

\*/

**public** **int** edgeCount()

{

**int** count = 0;

Iterator i = dict.values().iterator();

**while** (i.hasNext())

count += ((VertexS) i.next()).degree();

**return** count;

}

/\*\*

\* Construct a string representation of graph.

\*

\* **@post** returns string representation of graph

\*

\* **@return** String representing graph.

\*/

**public** String toString()

{

**return** "<GraphListDirected: "+dict.toString()+">";

}

**public** Iterator neighbors(Object label)

{

// return towns adjacent to vertex labeled label

//Assert.condition(dict.containsKey(label), "Vertex exists");

**return** ((VertexS) dict.get(label)).adjacentVertices();

}

/\*

\* //---------------------------------------------------

//Determines whether a sequence of flights between

//two cities exists. Nonrecursive stack version.

// keeps track of cities visited in the path in Queue

//Precondition: originCity and destinationCity are

//the origin and destination cities, respectively.

//Postcondition: Returns true if a sequence of

//flights exists from originCity to destination City,

//otherwise returns false. Cities visited during the

//search are marked as visited. A queue of visited cities in the path is returned

//---------------------------------------------------\*/

//Method to determine if there is a sequence of flights

**public** CircularArrayQueue isPath(Label originCity, Label destinationCity)

{

StackReferenceBased stack = **new** StackReferenceBased();

CircularArrayQueue que = **new** CircularArrayQueue();

Label topCity, nextCity;

reset(); // clear marks on all cities

VertexS vs;

Iterator iter;

// push origin city onto stack, mark it visited

stack.push(originCity);

// peek at item on the top of the stack and store in variable topcity, you will have to cast it to a label

topCity =(Label)stack.peek();

// call neighbors method which returns an iterator of the adjacent cities to topcity e.g

iter = neighbors(topCity);

// loop while stack is not empty and the topcity is not equal destination ciy

**while** (!stack.isEmpty() && (topCity.compareTo(destinationCity) != 0))

{

// call method to get NextCity defined below

nextCity = (Label)getNextCity(iter);

// if next city is null

**if**(nextCity == **null**)

{

// peek at the stack again

topCity =(Label)stack.peek();

// call the neighbors method again with topcity

iter = neighbors(topCity);

// pop the stack

stack.pop();

}

**else**

{

// push nextcity on the stack and mark it visite

stack.push(nextCity);

// insert it into the queue

que.insert(nextCity);

// peek at the city on top of the stack again

stack.peek();

} // end else

} // end while

// If Stack is empty, no path exists, return null else return the queue

**if**(stack.isEmpty())

**return** **null**;

**else**

**return** que;

} // end isPath

// gets the next city on the path, determines it is either null or visited, otherwise returns the city

**public** Label getNextCity( Iterator iter)

{

Label nextCity;

**if**(!iter.hasNext())

**return** **null**;

**else**

{

nextCity = (Label) iter.next();

VertexS vs = (VertexS)dict.get(nextCity);

**if** (vs.isVisited())

**return** **null**;

**return** nextCity;

}

}

}// end class

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

// Demonstrates a simulation of an airline scheduling system using a

// weighted graph data structure and a HashTable

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

import java.io.\*;

import java.util.\*;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import java.util.Iterator;

public class HPAirGUI extends JFrame

{

//-----------------------------------------------------------------

// Creates a new network and populates it with data from a file

// which contains the distance of flights to various cities. The user

// enters two cities and the program returns whether or not there is a flight

// between the two cities.

//-----------------------------------------------------------------

private String fileIn = "Cities.txt";

private JTextArea textbox;

private DefaultListModel listModel;

private JList jList;

private JButton helpButton, FindIndirectButton;

private JTextField firstCity, secondCity;

private JLabel sourceCity; // Setup input labels

private JLabel destinationCity;

private ActionHandler action;

private GraphListDirected cities;

public HPAirGUI()

{

// creates the graph to used to store cities and the their connections

cities = new GraphListDirected();

// create the listener for the buttons

action = new ActionHandler();

// create a JTextArea to itineraries

textbox = new JTextArea(5, 15);

textbox.setText("THIS IS YOUR INTINERARY");

JScrollPane scrollingArea = new JScrollPane(textbox);

scrollingArea.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

scrollingArea.setBorder(

BorderFactory.createCompoundBorder(

BorderFactory.createCompoundBorder(

BorderFactory.createLineBorder(Color.red),

BorderFactory.createLoweredBevelBorder()),

BorderFactory.createCompoundBorder(

BorderFactory.createLineBorder(Color.black),

BorderFactory.createLoweredBevelBorder())));

// displays the list of cities HPAir Services

listModel = new DefaultListModel(); // Setup Listbox

jList = new JList(listModel);

jList.setSelectionMode(ListSelectionModel.SINGLE\_SELECTION);

jList.setSelectedIndex(-1);

// jList.addListSelectionListener(this);

JScrollPane listScrollPane = new JScrollPane(jList);

// set up a compound border for the Jlist object

listScrollPane.setBorder(

BorderFactory.createCompoundBorder(

BorderFactory.createCompoundBorder(

BorderFactory.createLineBorder(Color.blue),

BorderFactory.createLoweredBevelBorder()),

BorderFactory.createCompoundBorder(

BorderFactory.createLineBorder(Color.gray),

BorderFactory.createLoweredBevelBorder())));

// Create all the buttons

FindIndirectButton = new JButton("FIND CONNECTIONS"); // Setup Buttons

FindIndirectButton.setToolTipText("Finds Connections Between Two Cities");

FindIndirectButton.setMargin(new Insets(5,5,10,5));

FindIndirectButton.addActionListener(action);

helpButton = new JButton("HELP"); // Setup Buttons

helpButton.setToolTipText("Instructions ");

helpButton.setMargin(new Insets(5,5,10,5));

helpButton.addActionListener(action);

JPanel buttonPane = new JPanel();

buttonPane.setBorder(BorderFactory.createLineBorder(Color.blue,2));

buttonPane.add(helpButton);

buttonPane.add(FindIndirectButton); // Attach buttons

//sets up the textfields for entering source and destination cities

firstCity = new JTextField(16); // Setup input fields

secondCity = new JTextField(16);

sourceCity = new JLabel("Source City"); // Setup input labels

destinationCity = new JLabel("Destination City");

JPanel fieldPanel = new JPanel();

fieldPanel.add(sourceCity);

fieldPanel.add(firstCity);

fieldPanel.add(destinationCity);

fieldPanel.add(secondCity);

// read in cities and their connections

BuildGraph();

Container c = getContentPane();

c.add(listScrollPane, BorderLayout.CENTER);

c.add(buttonPane, BorderLayout.SOUTH);

c.add(scrollingArea, BorderLayout.EAST);

c.add(fieldPanel, BorderLayout.NORTH);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* BuildGraph Method : I put the two input files, cities and then cities and

\* and their connections into one file, so I did not need two methods to read them

\* in. I simply read in the cities, create a label for each one, and call a method in

\* graphlist which adds them to the hashtable. See the add method in GraphList

\* which uses the label to create a vertex and then adds the label and vertex to the

\* Hashtable

\* After reading in the cities I read in the cities and the connections

\*between cities from the input file and add edges to the graph from city1 to city2.

\* It also outputs the cities

\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public void BuildGraph()

{

try {

Scanner scan = new Scanner(new File("Cities.txt"));

Scanner linescan;

String line, city1, city2;

double cost, distance;

int numCities;

Label label;

listModel.addElement("LIST OF CITIES HPAIR SERVICES");

//loop though lines to get data

// read in cities, create labels with them and add them to the hashtable

line = scan.nextLine();//reads in a line from the file

while(line.length() != 0 )

{

label = new Label(line);

cities.add(label);

listModel.addElement(line);

line = scan.nextLine();//reads in a line from the file

}

// Read in the connections between cities from the input

// file and add edges to the graph from city1 to city2.

listModel.addElement("LIST OF CONNECTONS");

while(scan.hasNext())

{

line = scan.nextLine();

// create a scanner object to loop through text in each line

linescan = new Scanner(line);

// to break up the line of input into usable fields -

city1 = linescan.next();

city2 = linescan.next();

distance = linescan.nextDouble();

Label label1 = new Label(city1);

Label label2 = new Label(city2);

cities.addEdge(label1, label2, (Double)distance);

}

}

catch(FileNotFoundException e)

{

System.out.println("File Not Found: ") ; // !!FILE\_NOT\_FOUND

}

catch(IOException e)

{

System.out.println("IOException " + e.getMessage() ); // !!READ\_ONLY!!

}

}// close method

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* class A

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

class ActionHandler implements ActionListener

{

Label l1, l2;

CircularArrayQueue que;

boolean ispath = false;

public void actionPerformed(ActionEvent e) // 'ADD' button was pressed

{

if(e.getActionCommand().equals("HELP"))

JOptionPane.showMessageDialog (null, "Enter two cities to determine if there is a flight ",

"INSTRUCTIONS", JOptionPane.PLAIN\_MESSAGE );

if (e.getActionCommand().equals("FIND CONNECTIONS"))

{ // gets connections

textbox.setText("");

// get origin and destination city

String c1 = firstCity.getText();

String c2 = secondCity.getText();

// create a label for each city

Label l1 = new Label(c1);

Label l2 = new Label(c2);

// call the method isPath with the labels

// isPath returns aCcircularArrayQueue

// which is instantiated above as "que"

que = cities.isPath(l1, l2);

//if the que is null, output the appropriate message

if(que == null)

textbox.setText("No Path");

else

// output the appropriate message that there is a connection]

// and output the que ( you can use que.toString())

//System.out.println(" In find connection after getting que" + que.toString());

textbox.setText("A Path Exists\n" + que.toString());

firstCity.setText("");

secondCity.setText("");

}

} // close actionperformed

} //close ActionHandler

}//class CheapestFlight

/\* public static void readFiles (String[] args)

{

String file = "flights.txt";

/\* try

{

BufferedReader inFile = new BufferedReader(new FileReader(file));

BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));

// Network<String> cities = new Network<String>();

StringTokenizer tokenizer;

String line, city1, city2;

double cost;

int numCities;

// Read in the list of cities from the input file and add

// add them to the network. List of cities will be

// terminated by a blank line.

line = inFile.readLine();

while (line.length() != 0)

{

cities.addVertex(line);

line = inFile.readLine();

}

// Read in the connections between cities from the input

// file and add them to the network.

line = inFile.readLine();

while (line != null)

{

tokenizer = new StringTokenizer(line, "\t\n");

city1 = tokenizer.nextToken();

city2 = tokenizer.nextToken();

cost = Double.parseDouble(tokenizer.nextToken());

cities.addEdge(city1, city2, cost);

line = inFile.readLine();

}

// Display the list of cities

numCities = cities.size();

Iterator<String> cityiterator = cities.iteratorBFS(0);

System.out.println("Cities");

System.out.println("------");

while (cityiterator.hasNext())

{

System.out.println(cityiterator.next());

}

// Prompt the user to enter two cities

System.out.print("\nCity 1: ");

city1 = keyboard.readLine();

System.out.print("\nCity 2: ");

city2 = keyboard.readLine();

// Display the shortest path between the two cities and

// how much it would cost

cost = cities.shortestPathWeight(city1, city2);

if (cost < Double.POSITIVE\_INFINITY)

{

System.out.print("\nThe cheapest path from " + city1 + " to " + city2 + " is ");

NumberFormat money = NumberFormat.getCurrencyInstance();

System.out.println(money.format(cities.shortestPathWeight(city1, city2)));

System.out.print("You would have to travel from ");

Iterator<String> it = cities.iteratorShortestPath(city1, city2);

while (it.hasNext())

{

System.out.println(it.next());

if (it.hasNext())

System.out.print(" to ");

}

}

else

System.out.println("\nThere is no path from " + city1 + " to " + city2);

}

catch (FileNotFoundException e) {System.out.println("file " + file + " not found");}

catch (IOException e) {System.out.println("IO exception");}

}//method main

\*/