A

PROJECT

ON

JOURNEY PLANNER

By:

PuneetJyot Singh (007/CSE1/2016)

Department of Computer Science & Engineering
Guru Tegh Bahadur Institute of Technology

Guru Gobind Singh Indraprastha University
Dwarka, New Delhi

ACKNOWLEDGEMENT

I hereby take this opportunity to thank all those people whose knowledge and experience helped me bring this report in its present form. It would have been a tough task for me to complete the report without their help.

I express my sincere thanks and gratitude to my mentor **Mr. Paul Blaer, Columbia University.** For providing me the opportunity to pursue my training at the organization.

I would like to express my gratitude to my colleague friends for helping me in every aspect during the training phase as well as in carrying out the whole project.

Finally I would like to express my deep appreciation to my family and friends who have been a constant source of inspiration. I am internally grateful to them for always encouraging me wherever and whenever I needed them.

PuneetJyot
00713202714

DECLARATION

I, **PuneetJyot Singh**, student of B.Tech - CSE programme at Guru Tegh Bahadur Institute of Technology, Rajouri Garden, New Delhi, hereby declare that I have undergone Summer Training at **Columbia University, New York** under the mentorship of **Mr. Paul Blaer** from **July 6th, 2016** to **August 12th**, **2016** and learned **Data Structures and Algorithms in Java**.

I also declare that the present project report is based on the above summer training, and is my original work. The content of this project report has not been submitted to any other university or institute either in part or in full for the award of any degree, diploma or fellowship.

PuneetJyot

00713202714

Cse-1

ABSTRACT

The Project focuses on finding the shortest path between 2 destination using Dijkstra Algorithm. Dijkstra Algorithm is used in this project because it is one of the efficient algorithm to find the shortest method than other search algorithms because of its less time and space complexity.

A **journey planner** is a specialized electronic search engine used to find the best journey between two points by some means of transport. Journey planners have been widely used in the travel industry since the 1970s by booking agents accessed through a user interface on a computer terminal, and to support call center agents providing public transport information. With the advent of the internet, self-service browser based on-line journey planner interfaces for use by the general public have become widely available. A journey planner may be used to provide schedule information and plan your journey properly. It even provides a flexibility to make changes to your journey as per the time constraints and places of interest.

We placed great emphasis on the complexity of the algorithm to make it much faster than other search algorithms by using various abstract data types of data structure.

CONTENTS

Introduction

JAVA

- Java was created by a team of members called "Green" led by James Arthur Gosling.
- When Java was created in 1991, it was originally called Oak.
- It is a free and open source software (FOSS) under GNU General Public License(GPL)
- First version of Java was released in 1995.
- Java is an Object oriented language, simple, portable, platform independent, robust and secure.
- We will be focusing on Java 1.6.

Types of Java

- JSE
  - Java Standard Edition formerly known as J2SE.
  - This forms the core part of Java language.
- JEE
  - Java Enterprise Edition formerly known as J2EE.
  - These are the set of packages that are used to develop distributed enterprise-scale applications.
  - These applications are deployed on JEE application servers.
- JME
  - Java Micro Edition formerly known as J2ME.
- These are the set of packages used to develop application for mobile devices and embedded systems.

Features of Java Language

- Simple

- Object Oriented Language
- Portable and platform independent
- Robust
- Multithreaded
- Dynamic Linking
- Secure
- Performance

JVM: perspectives

JVM can be looked as

- A runtime instance: JVM life cycle begins when applications starts to run (that is when main method is called) and ends when the application ends.
- The abstract specification: Specification that Java team at Sun (Oracle) provides which tells JVM makers how they must design JVM for their OS.
- A concrete implementation: JVM that is built specifically targeted for an OS based on abstract specification.

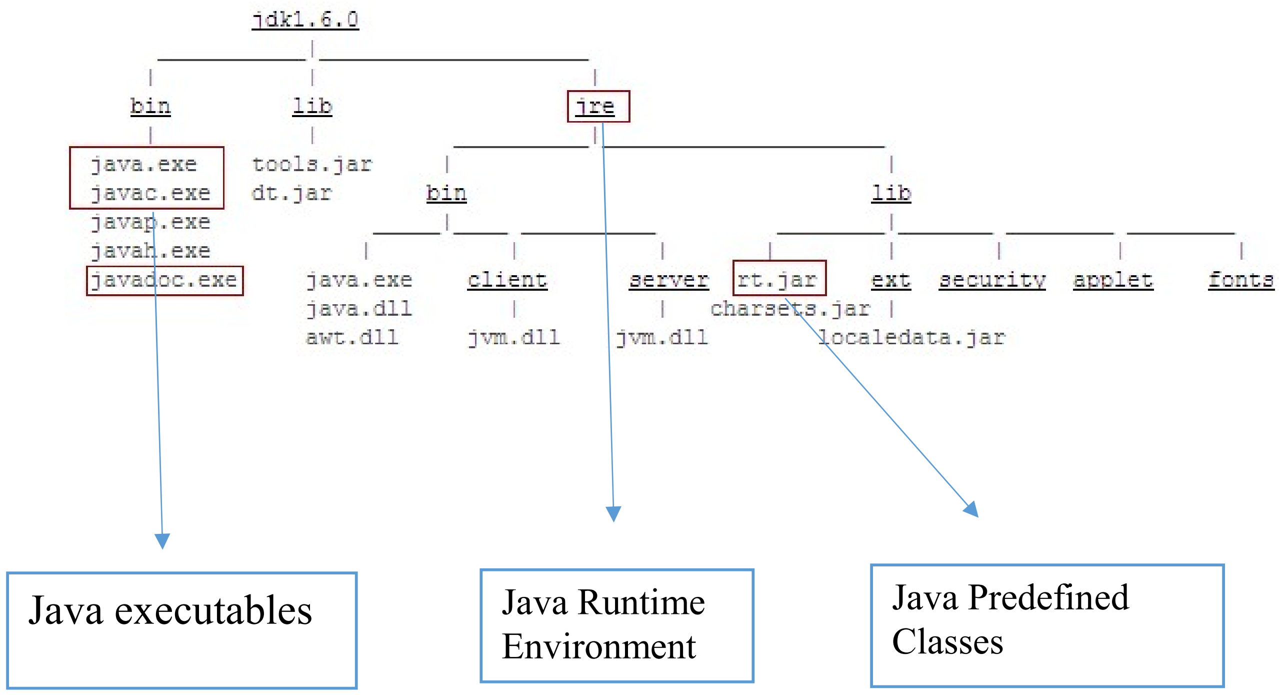
Environment to compile and execute

- Compile java programs
  - From command prompt
  - Through an IDE (Integrated development environment)
    - Eclipse →Apache
    - NetBeans →Oracle SDN
    - JBuilder → Borland
    - Integrated Development Environment → IBM

Setup JDK

- http://java.com/en/download/index.jsp or find appropriate link in http://www.oracle.com/technetwork/indexes/downloads
- Download Java based on the type of OS
  - Windows
  - Linux
  - Mac OS
  - Solaris
- Install JDK

JDK installation directory



About Columbia University Computer Science Department

The computer science curriculum at Columbia places equal emphasis on theoretical computer science and mathematics and on experimental computer technology. A broad range of upper-level courses is available in such areas as artificial intelligence, computational complexity and the analysis of algorithms, combinatorial methods, computer architecture, computer-aided digital design, computer communications, databases, mathematical models for computation, optimization, and software systems
The department has well-equipped lab areas for research in computer graphics, computer-aided digital design, computer vision, databases and digital libraries, data mining and knowledge discovery, distributed systems, mobile and wearable computing, natural-language processing, networking, operating systems, programming systems, robotics, user interfaces, and real-time multimedia.

The computer facilities include a shared infrastructure of Linux multiprocessor servers, NetApp file servers, a student interactive teaching and research lab of high-end multimedia workstations, a load balanced web cluster with 6 servers and business process servers, a large student laboratory, featuring 17 Mac-mini machines and 33 Linux towers each with 8 cores and 24GB memory; a remote Linux cluster with 17 servers, a large Linux computer cluster and a number of computing facilities for individual research labs. In addition, the data center houses a computer cluster consisting of a Linux cloud with 43 servers each with 2 Nehalem processors, 8 cores and 24GB memory. This cloud can support approximately 5000 of VMware instances.

The research facility is supported by a full-time staff of professional system administrators and programmers

REQUIREMENT ANALYSIS

HARDWARE REQUIREMENTS –

Processor	Intel Pentium Processor 4, 2.4GHz or above.
RAM	4 GB RAM
HARD DISK	MINIMUM 20 GB

SOFTWARE REQUIREMENTS -

Operating System	Windows
Software	Java Development Kit 1.8
Programming Language	Java

Introducing Eclipse Helios 3.6 IDE

Eclipse is an integrated development environment (IDE) used in computer programming, and is the most widely used Java IDE. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages through the use of plugins, including: Ada, ABAP, C, C++, COBOL, D, Fortran, Haskell, JavaScript, Julia, Lasso, Lua, NATURAL, Perl, PHP, Prolog, Python, R, Ruby (including Ruby on Rails framework), Rust, Scala, Clojure, Groovy, Scheme, and Erlang. It can also be used to develop documents with LaTeX (through the use of the TeXlipse plugin) and packages for the software Mathematica. Development environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others.

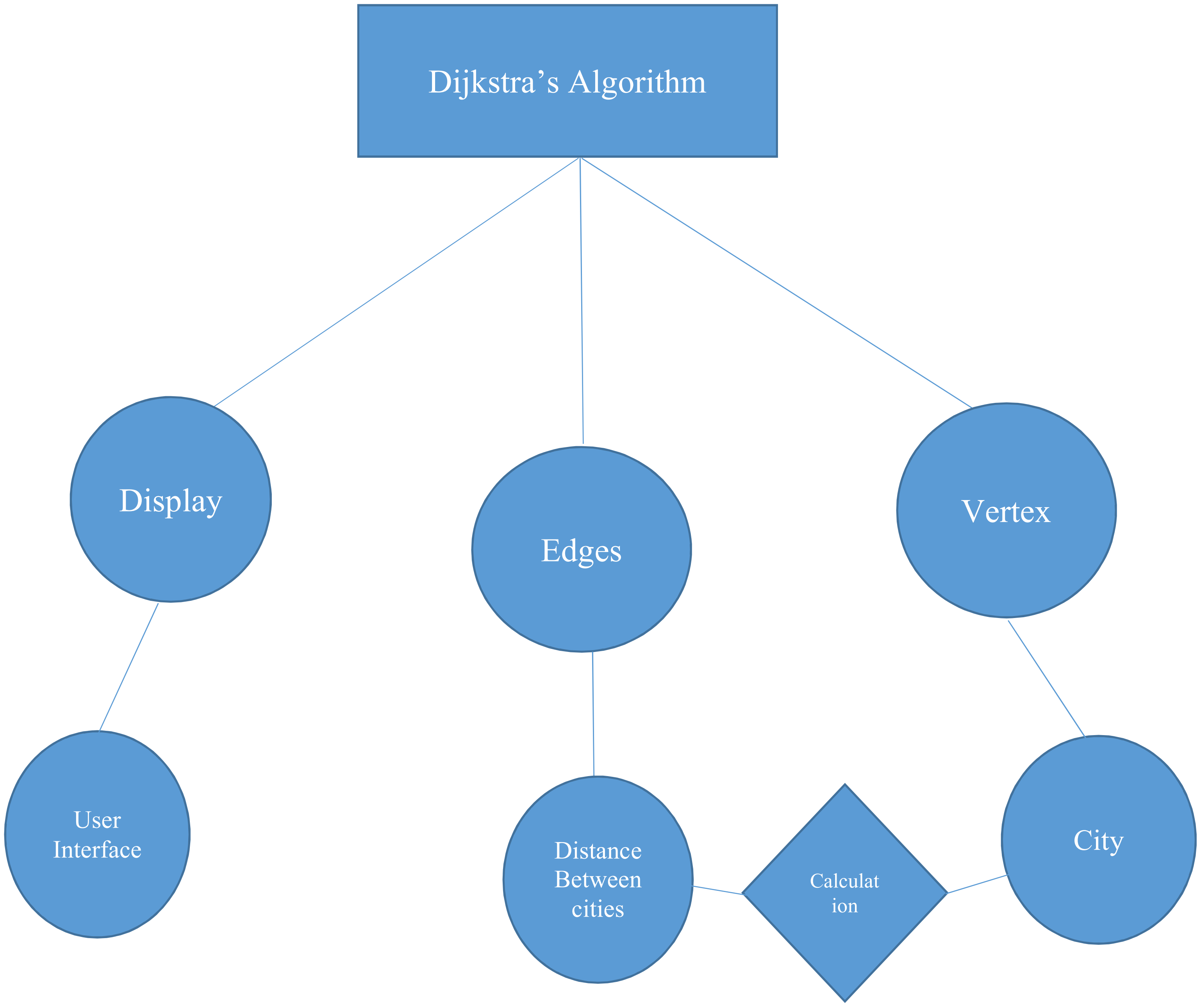
- IDE is where the Java programs are written, compiled and executed.
- Eclipse is an open source project
  - Launched in November 2001
  - Designed to help developers with specific development tasks
- GUI and non-GUI application development support
- Easy integration of tools
- Supported by multiple operating systems like Windows, Mac, Linux, Solaris, IBM AIX and HP-UX.
- UI Monitoring
- Docker Tools
- Git Flow

Eclipse as an IDE

- Java Development Tooling (JDT) is used for building Java code
- Provides set of workbench plug-ins for manipulating Java code
    - Java projects, packages, classes, methods,
- Java compiler is built in
    - Used for compiling Java code
    - Creates errors (special markers of code) if compilation fails
- Numerous content types support
    - Java, HTML, C, XML, ...

SYSTEM DESIGN

E-R Diagram



Introduction to Data Structure

In computer science, a data structure is a particular way of organizing data in a computer so that it can be used efficiently. Data structures can implement one or more particular abstract data types (ADT), which specify the operations that can be performed on a data structure and the computational complexity of those operations. In comparison, a data structure is a concrete implementation of the specification provided by an ADT.

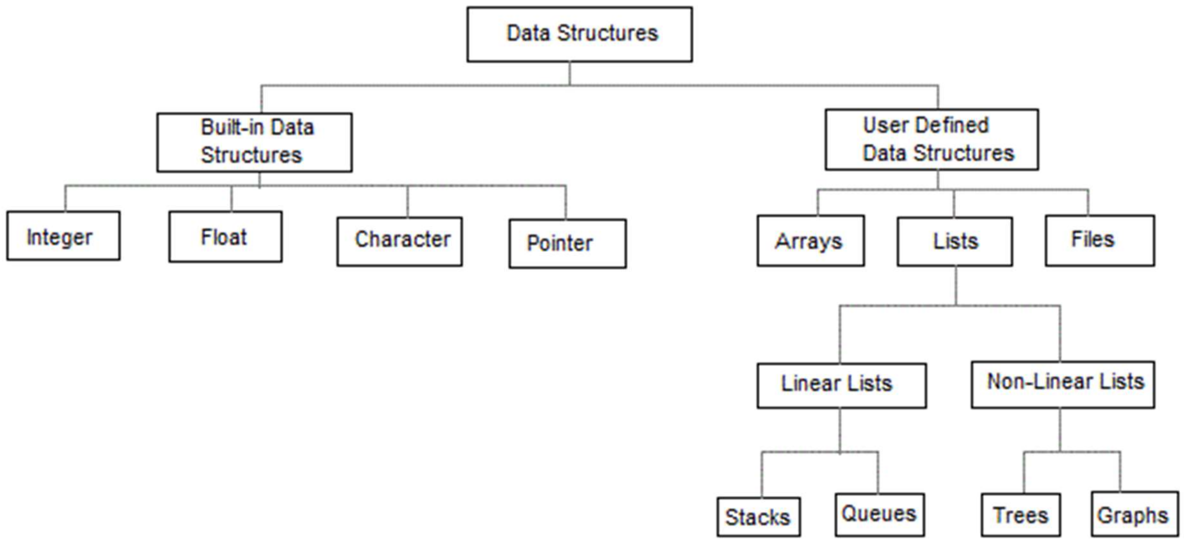
Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory.

Some example of **Abstract Data Structure** are:

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

What is Algorithm?

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Space Complexity

It’s the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components:

- **Instruction Space:** It’s the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** It’s the space required to store all the constants and variables value.
- **Environment Space:** It’s the space required to store the environment information needed to resume the suspended function.

Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion.

Dijkstra’s Algorithm

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. Like Prim’s MST, we generate a SPT (shortest path tree) with given source as root. We maintain two sets, one set contains vertices included in shortest path tree, and other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph?

Algorithm

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

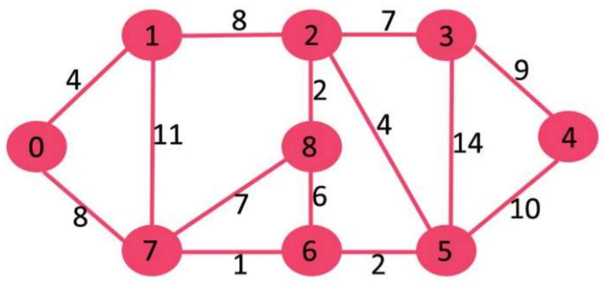
3) While sptSet doesn’t include all vertices

….a) Pick a vertex u which is not there in sptSetand has minimum distance value.

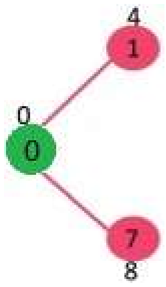
….b) Include u to sptSet.

….c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

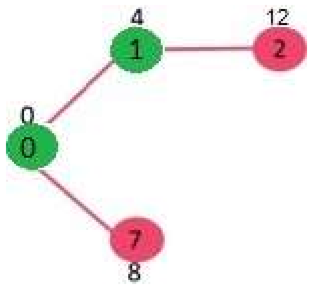
Let us understand with the following example:-



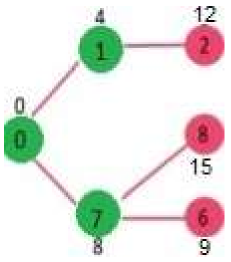
The set *sptSet*is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in *sptSet*. So *sptSet* becomes {0}. After including 0 to *sptSet*, update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices in ~cluded in SPT are shown in green color.



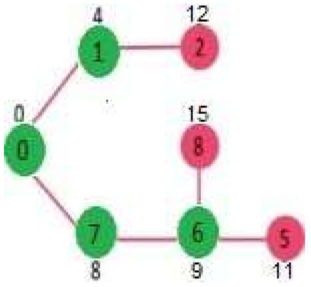
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet. So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.



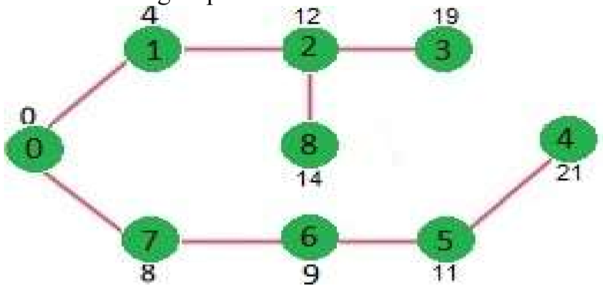
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes {0, 1, 7}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).



Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes {0, 1, 7, and 6}. Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



After following steps



Prim’s Algorithm

Prim’s algorithm is also a Greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.
A group of edges that connects two set of vertices in a graph is called cut in graph theory. So, at every step of Prim’s algorithm, we find a cut (of two sets, one contains the vertices already included in MST and other contains rest of the verices), pick the minimum weight edge from the cut and include this vertex to MST Set (the set that contains already included vertices).

How does Prim’s Algorithm Work? The idea behind Prim’s algorithm is simple, a spanning tree means all vertices must be connected. So the two disjoint subsets (discussed above) of vertices must be connected to make a Spanning Tree. And they must be connected with the minimum weight edge to make it a Minimum Spanning Tree.

Algorithm
1) Create a set mstSet that keeps track of vertices already included in MST.
2) Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3) While mstSet doesn’t include all vertices
….a) Pick a vertex u which is not there in mstSet and has minimum key value.
….b) Include u to mstSet.
….c) Update key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge u-v is less than the previous key value of v, update the key value as weight of u-v

Kruskal's Algorithm

Given a connected and undirected graph, a spanning tree of that graph is a subgraph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?
A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

Below are the steps for finding MST using Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.

2. Pick the smallest edge. Check if it forms a cycle with the spanning tree
formed so far. If cycle is not formed, include this edge. Else, discard it.

3. Repeat step#2 until there are (V-1) edges in the spanning tree.

FUTURE WORK

Our project aims at planning a trip easily. In future, we can make our website more interactive and attractive by implementing some of the features like whenever a user clicks on a particular place on the map, it should either pop out a message giving the description of that place or some useful information about that place. We can make our website more attractive by applying different themes depending on the specialty of the requested place. This can be implementing using dynamic web page creation.

This project is being developed as the prototype of a journey planner. We have tried our best to include as many places as possible according to the time and budget. Our project can be enhanced to provide more and more information as possible by adding information about places into our database. Our project has been designed to easily upgrade according to the time and needs.

CONCLUSION

It was a wonderful learning experience while working in this training. This training took me through various phases of the app development and gave me real insight into the world of JAVA Development using Abstract Datatype. The joy of working and the thrill involved while tackling the various problems and challenges gave me a real feel of the Developer's Industry.

It was due to this training that I realized how real application projects are made, and how things work.

I enjoyed each and every bit of my journey in the training and I look forward to more such endeavors in the future.

BIBLIOGRAPHY

Here I would like to mention about the sources of information in due course of writing the project report & would like to state that the information obtained through different presentation & news article has greatly contributed to the successful completion of the project. The list of different sources is mentioned below:

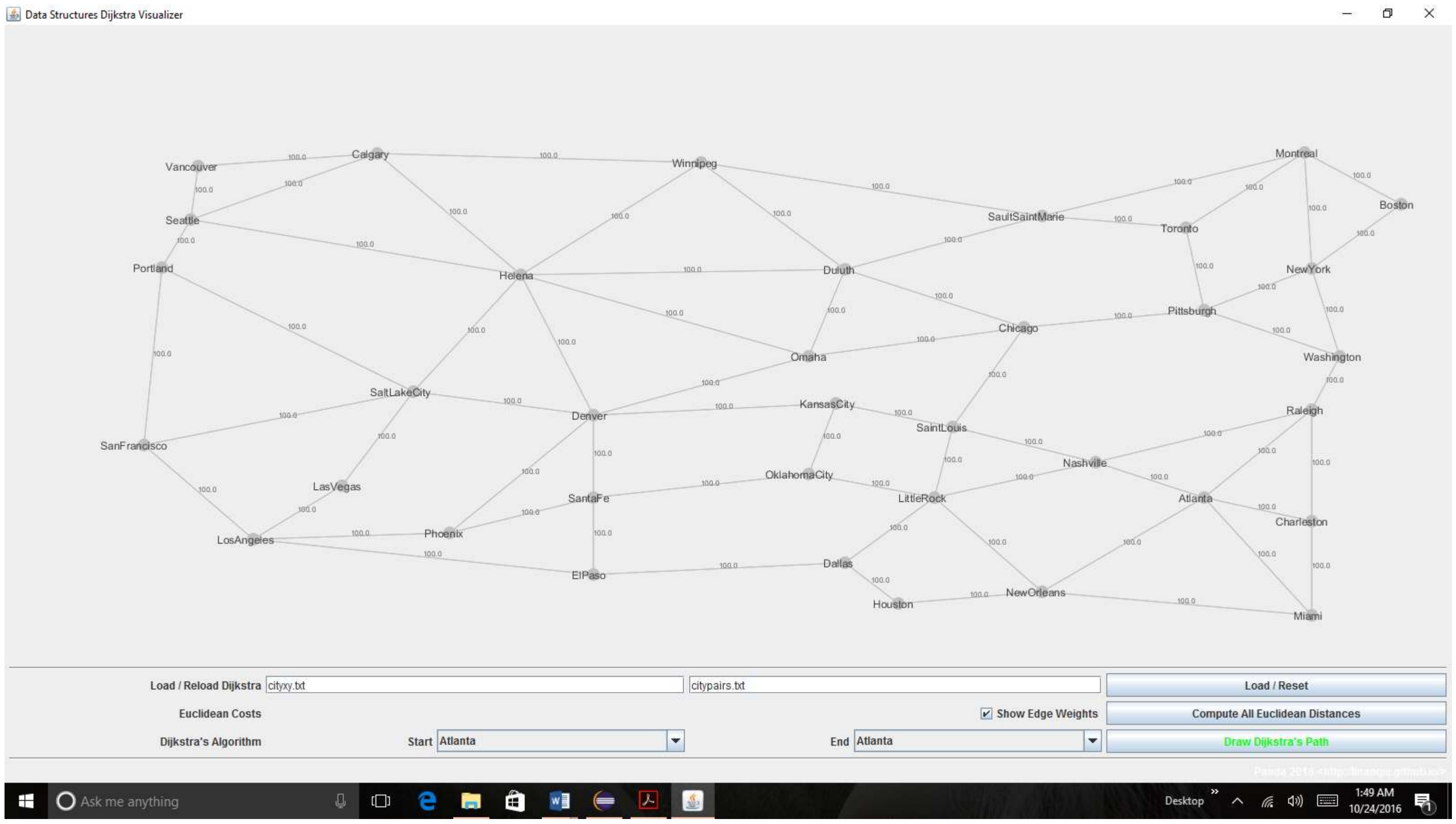
Books:
Mark Allen Weiss- Data Structure in Java
Big Early Objects of java-6th edition

Websites:
http://bulletin.engineering.columbia.edu/computer-science
http://documents.mx/download/link/srs5472f223b4af9fb6338b470a
https://en.wikipedia.org/wiki/Abstract_data_type
https://en.wikipedia.org/wiki/Eclipse_(software)
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

Appendix A

Screen Shots

Path Finder Screen



Load / Reload Dijkstra: cityxy.txt citypairs.txt

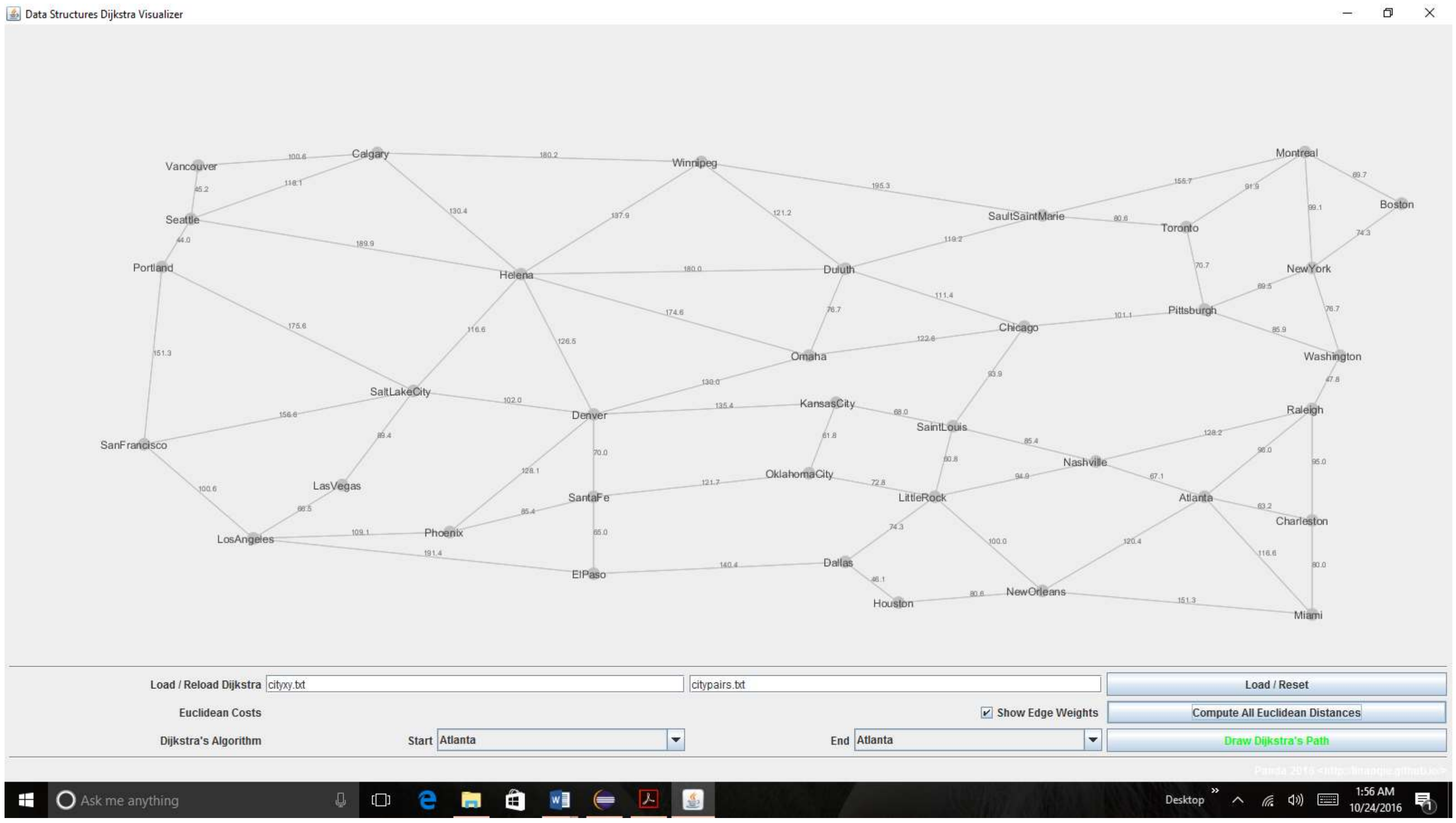
☑ Show Edge Weights

Start: Atlanta End: Atlanta

This is a prototype model of a desktop application known as Journey Planner.
At this time Euclidean distances between all the cities are set as 100 which is the default value and we just have loaded 2 text file which is present in same package.

These files contains all information about the geographical coordinates of different cities used in this model and all possible pairs which can be created using these cities.

<u>Calculation of Euclidean Distances</u>



Load / Reload Dijkstra cityxy.txt citypairs.txt

Euclidean Costs ✓ Show Edge Weights

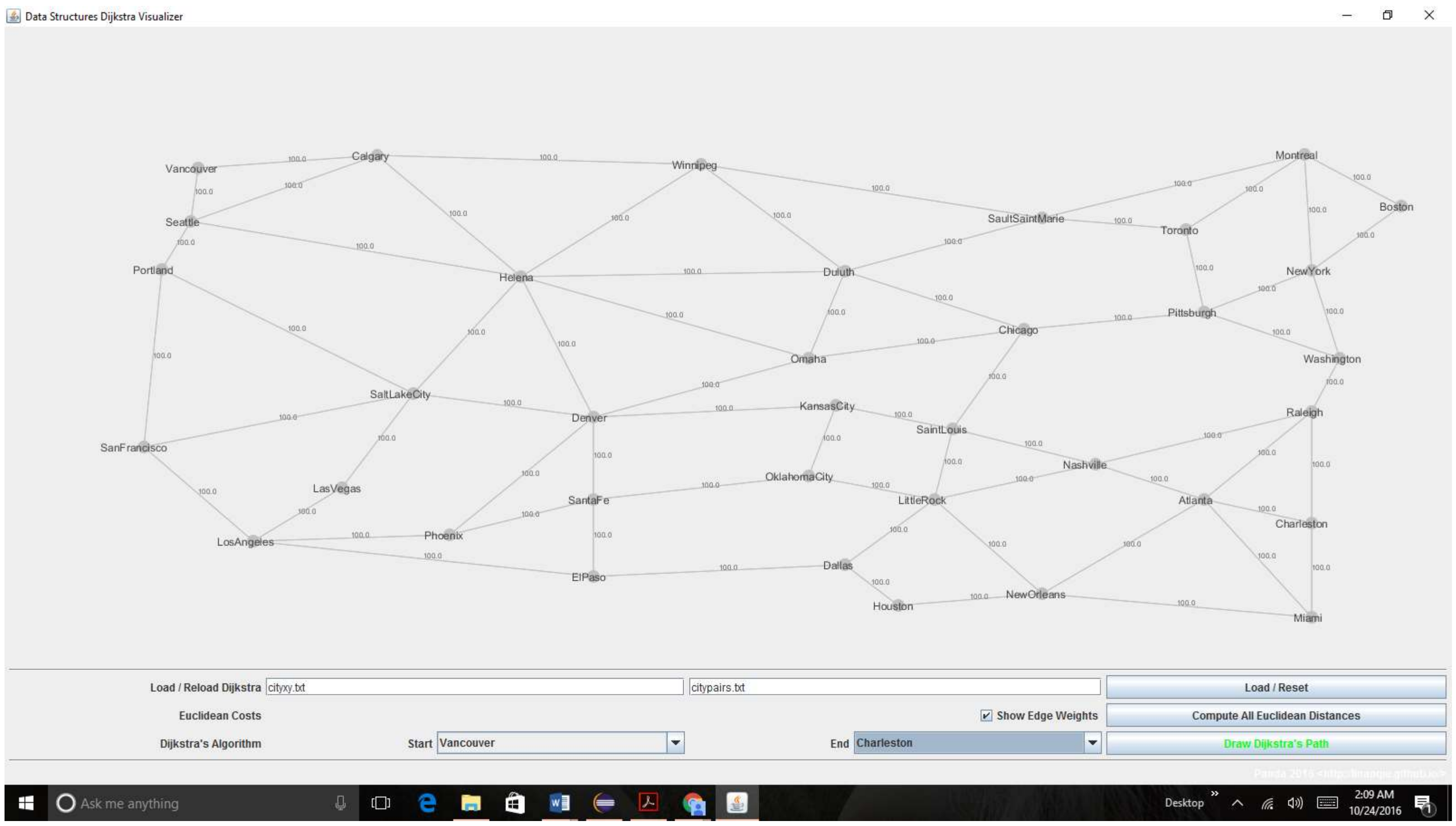
Dijkstra's Algorithm Start Atlanta End Atlanta

After Journey planner application is launched and this screen appears on your desktop, click on Compute All Euclidean Distance.

This will give you distance between every pair of cities which first was initialized with 100 as a default value.

After calculating distances you can move further and select our source and destination because before computing distances our project would consider every distance equal to other

<u>Choosing Source and Destination</u>



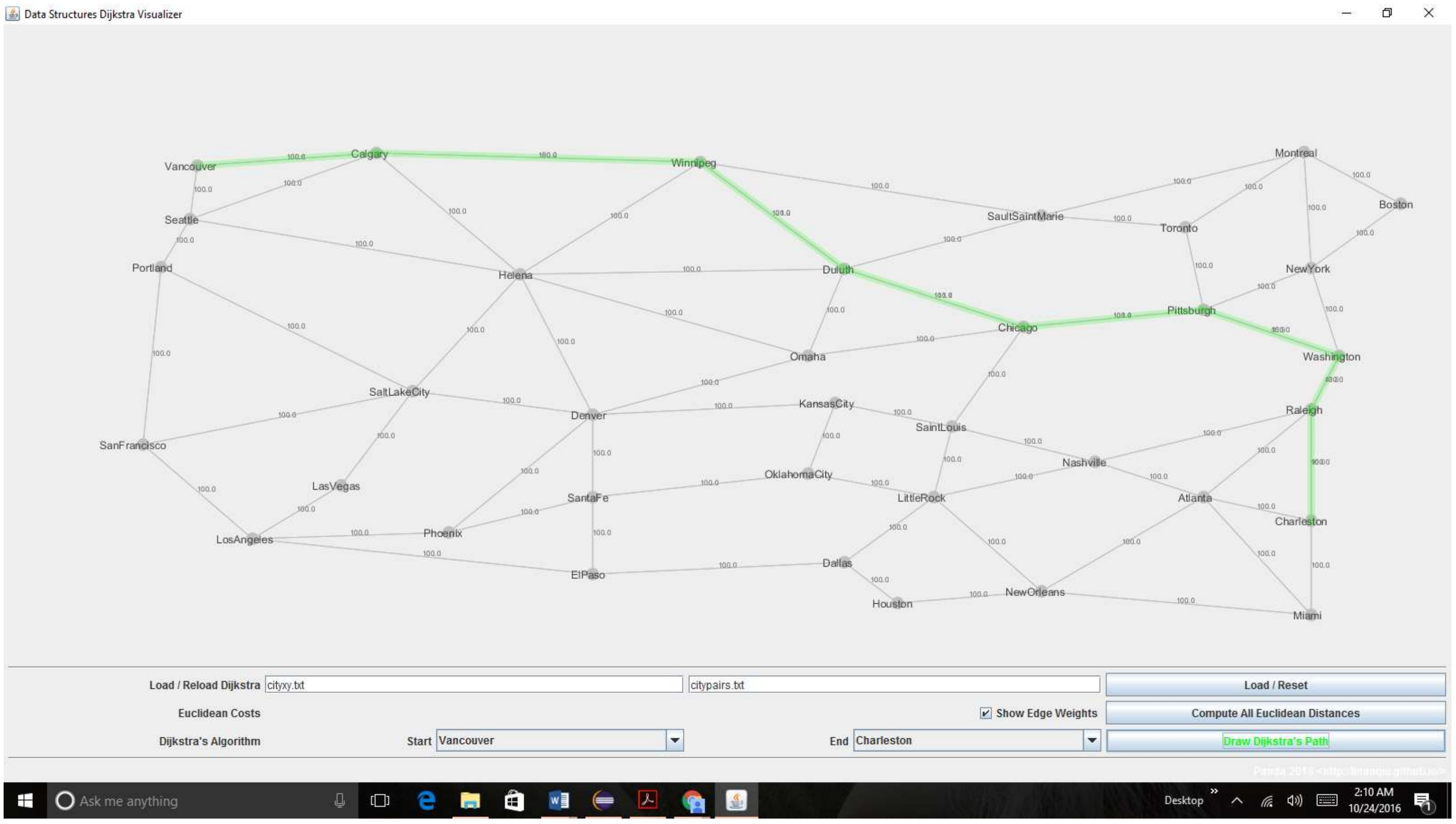
Load / Reload Dijkstra cityxy.txt citypairs.txt

Euclidean Costs ✓ Show Edge Weights

Dijkstra's Algorithm Start Vancouver End Charleston

In this part of the project user has to select the source i.e.-From where he wants to start and the destination. Once he calculates all the distances, now user is ready to calculate shortest path using Dijkstra’s algorithm.

Shortest Path between 2 locations



Load / Reload Dijkstra cityxy.txt citypairs.txt

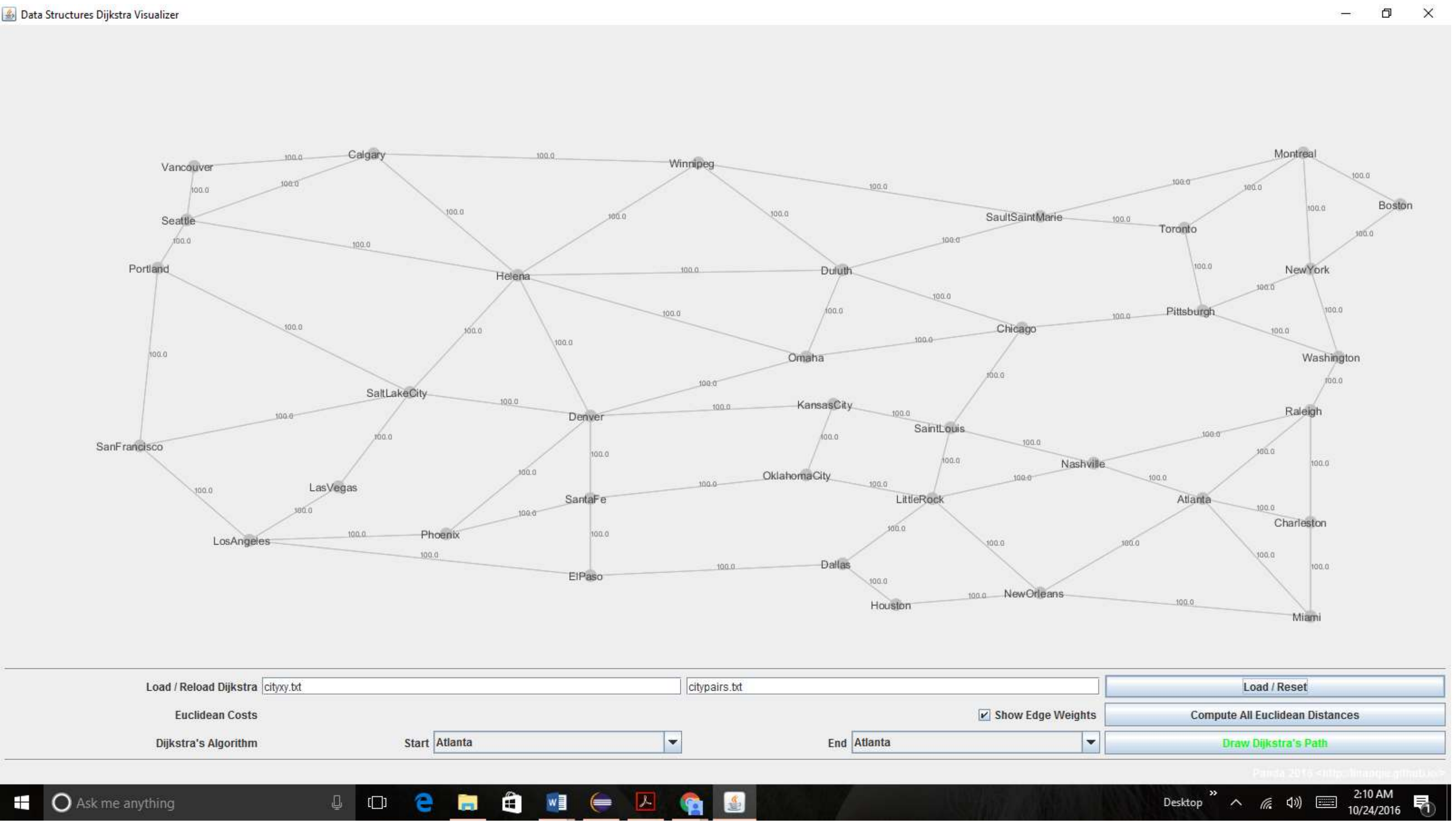
Euclidean Costs ☑ Show Edge Weights

Dijkstra's Algorithm Start Vancouver End Charleston

In this part of the project you just have to click on Draw Dijkstra's Path button and a green line will be formed between 2 locations which you have selected in minimum time.

So here if you want to go from **Vancouver** to **Charleston**, you have to follow this path to reach your destination in minimum time.

Load/Reset



Load / Reload Dijkstra cityxy.txt citypairs.txt

Euclidean Costs ☑ Show Edge Weights

Dijkstra's Algorithm Start Atlanta End Atlanta

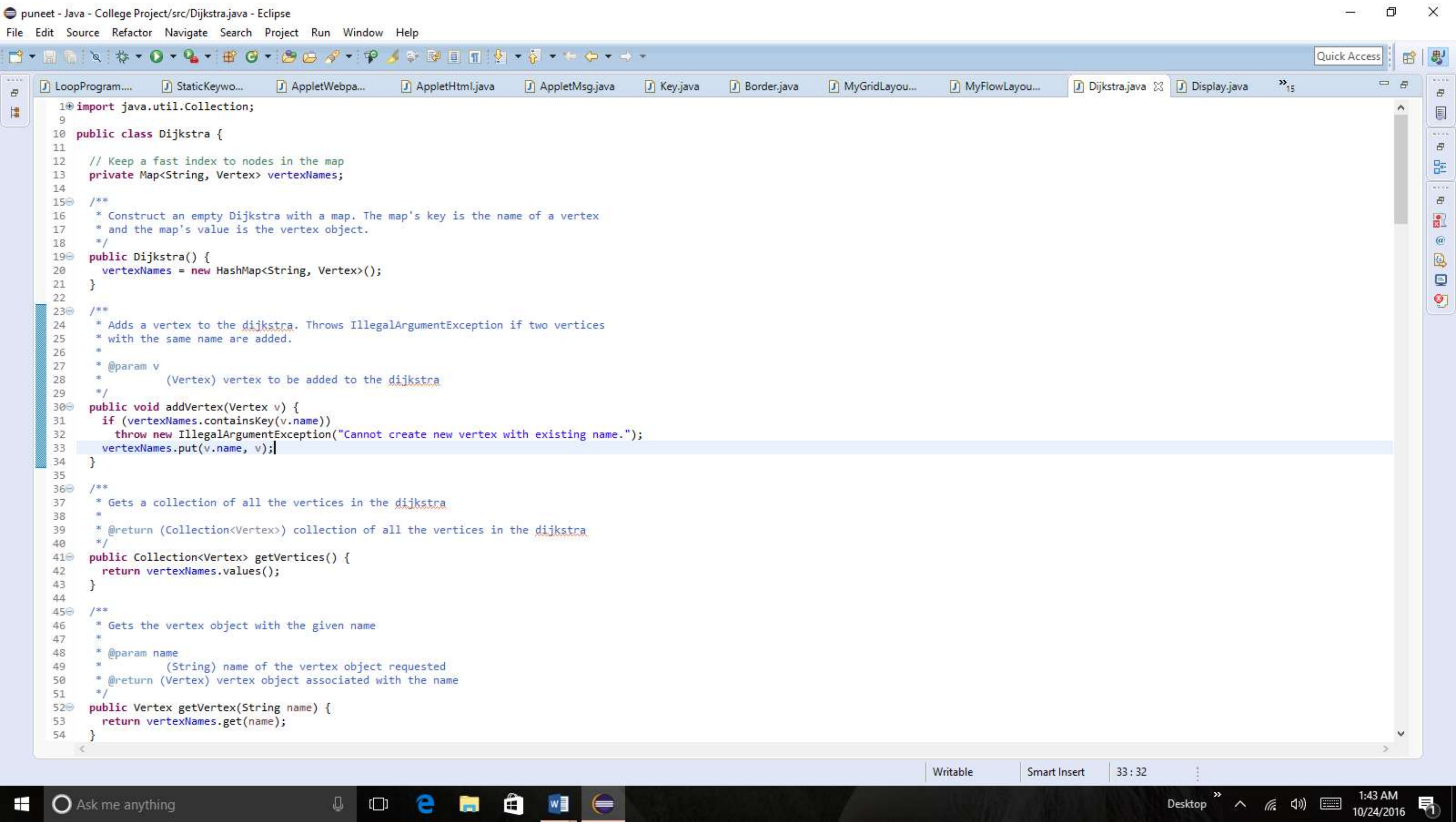
After finding the shortest path if you want to modify your route or want to visit some other location which is not on the route, you can just click on load and reset button and can find new shortest path from that particular location after reaching there.

This button will take our model again in the initial step and you have to calculate Euclidean distance and follow the procedure given upward again.

Appendix B

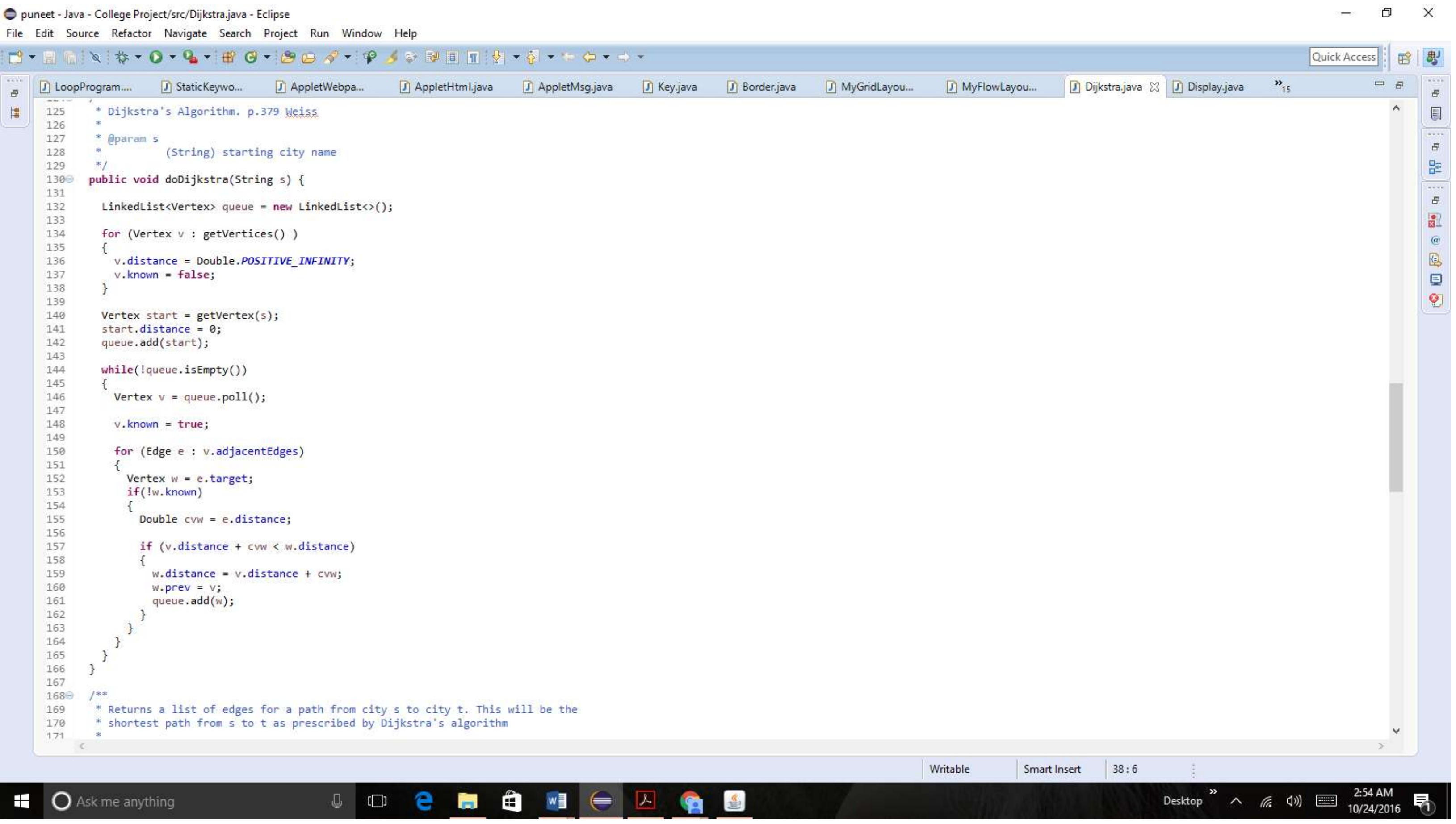
Source Code

Dijkstra Class-



```java
import java.util.Collection;

public class Dijkstra {

  // Keep a fast index to nodes in the map
  private Map<String, Vertex> vertexNames;

  /**
   * Construct an empty Dijkstra with a map. The map's key is the name of a vertex
   * and the map's value is the vertex object.
   */
  public Dijkstra() {
    vertexNames = new HashMap<String, Vertex>();
  }

  /**
   * Adds a vertex to the dijkstra. Throws IllegalArgumentException if two vertices
   * with the same name are added.
   *
   * @param v
   *          (Vertex) vertex to be added to the dijkstra
   */
  public void addVertex(Vertex v) {
    if (vertexNames.containsKey(v.name))
      throw new IllegalArgumentException("Cannot create new vertex with existing name.");
    vertexNames.put(v.name, v);
  }

  /**
   * Gets a collection of all the vertices in the dijkstra
   *
   * @return (Collection<Vertex>) collection of all the vertices in the dijkstra
   */
  public Collection<Vertex> getVertices() {
    return vertexNames.values();
  }

  /**
   * Gets the vertex object with the given name
   *
   * @param name
   *          (String) name of the vertex object requested
   * @return (Vertex) vertex object associated with the name
   */
  public Vertex getVertex(String name) {
    return vertexNames.get(name);
  }
```

Dijkstra Algorithm

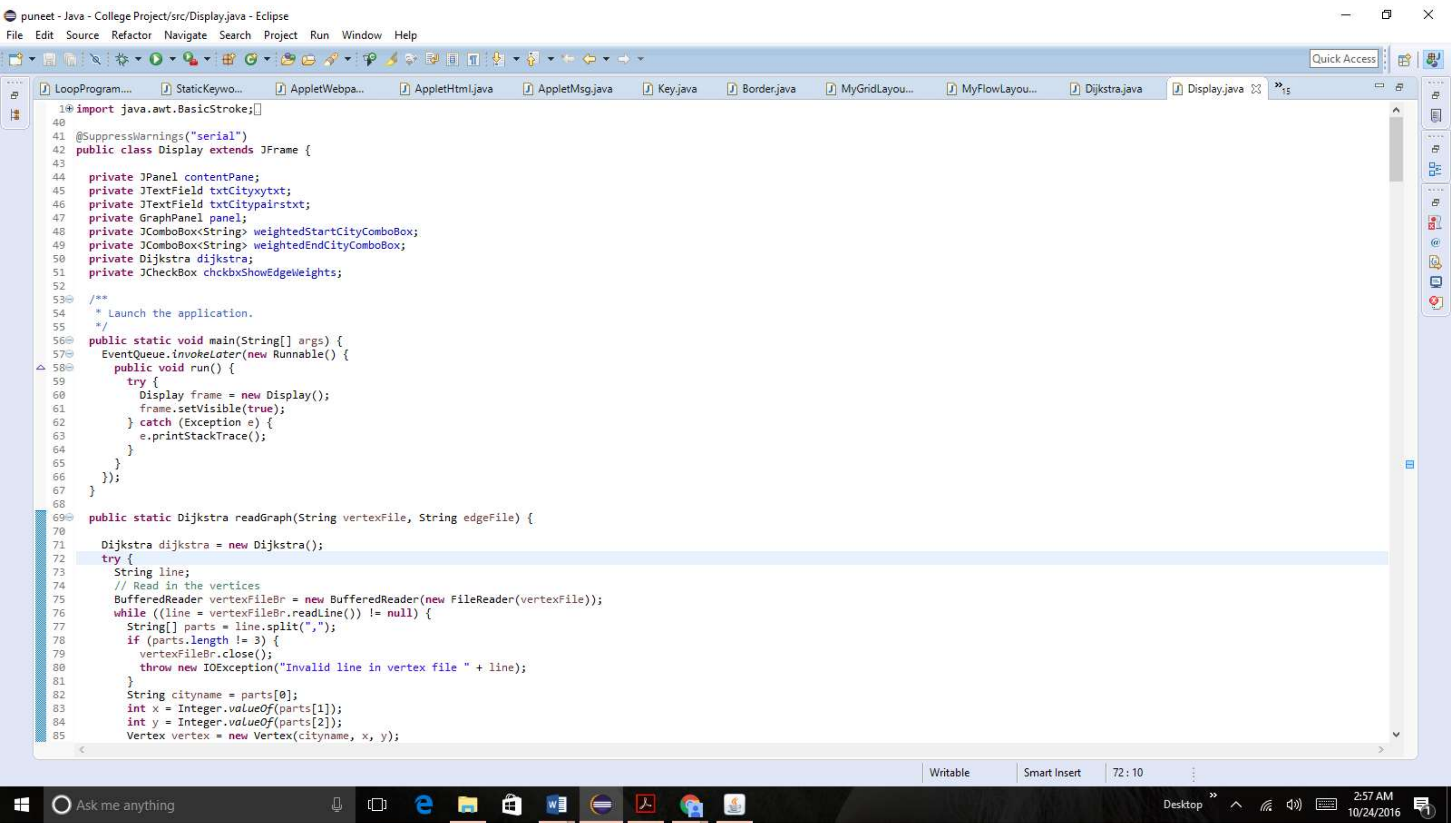


```
 * Dijkstra's Algorithm. p.379 Weiss
 *
 * @param s
 *          (String) starting city name
 */
public void doDijkstra(String s) {

  LinkedList<Vertex> queue = new LinkedList<>();

  for (Vertex v : getVertices() )
  {
    v.distance = Double.POSITIVE_INFINITY;
    v.known = false;
  }

  Vertex start = getVertex(s);
  start.distance = 0;
  queue.add(start);

  while(!queue.isEmpty())
  {
    Vertex v = queue.poll();

    v.known = true;

    for (Edge e : v.adjacentEdges)
    {
      Vertex w = e.target;
      if(!w.known)
      {
        Double cvw = e.distance;

        if (v.distance + cvw < w.distance)
        {
          w.distance = v.distance + cvw;
          w.prev = v;
          queue.add(w);
        }
      }
    }
  }
}

/**
 * Returns a list of edges for a path from city s to city t. This will be the
 * shortest path from s to t as prescribed by Dijkstra's algorithm
 *
```

It represents a graph and will contain your implementation of Dijkstra's algorithm.

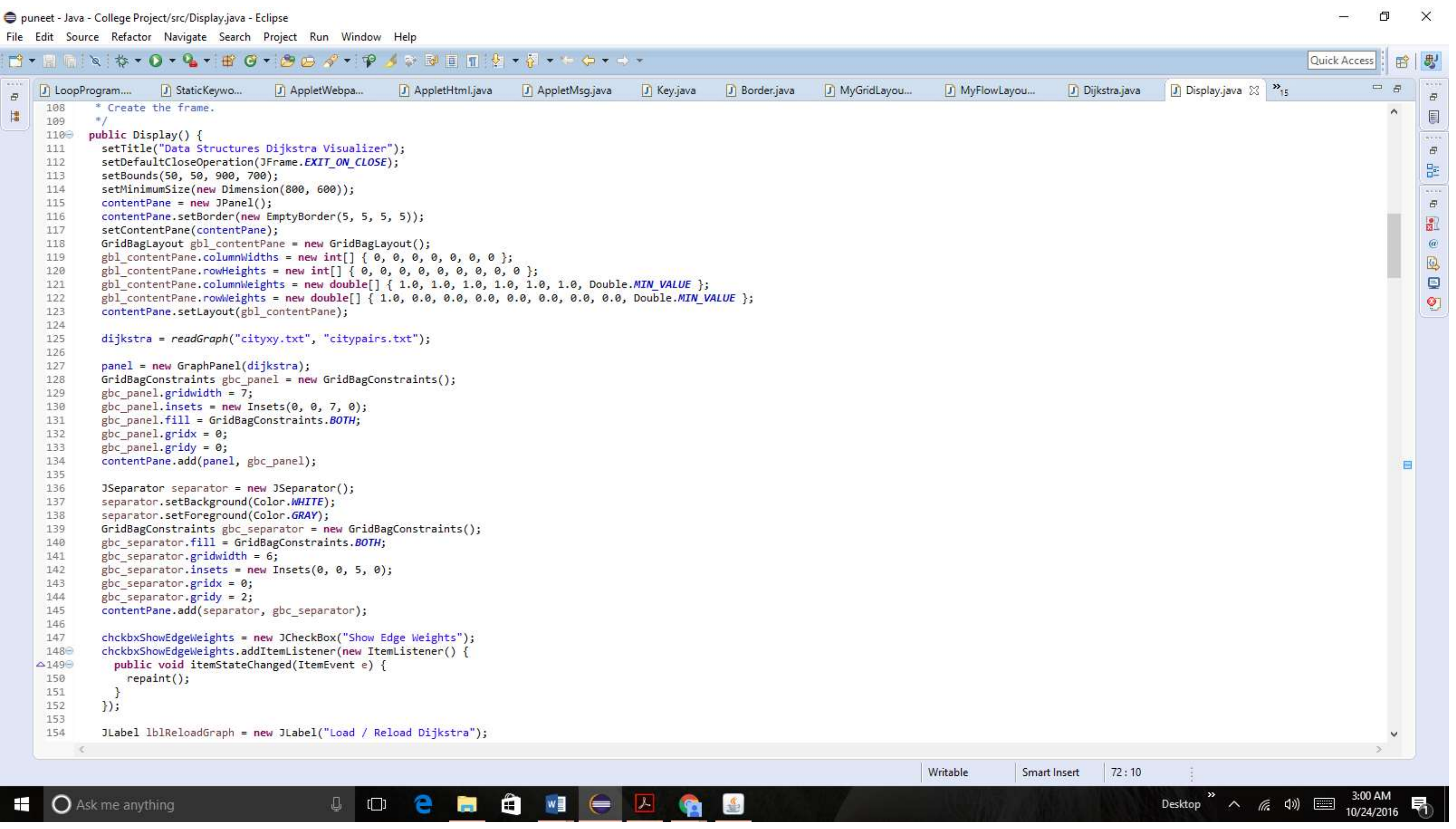
Here we have the instance variable vertexNames, which contains a mapping from city names to Vertex objects after the graph is read from the data files. The main method of Dijkstra illustrates how the class is used by the GUI and might be useful for testing your implementation on the command line.

<u>Display class</u>


puneet - Java - College Project/src/Display.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
LoopProgram.... StaticKeywo... AppletWebpa... AppletHtml.java AppletMsg.java Key.java Border.java MyGridLayou... MyFlowLayou... Dijkstra.java Display.java
Writable Smart Insert 72 : 10
Ask me anything
Desktop
2:57 AM
10/24/2016

```
import java.awt.BasicStroke;

@SuppressWarnings("serial")
public class Display extends JFrame {

  private JPanel contentPane;
  private JTextField txtCityxytxt;
  private JTextField txtCitypairstxt;
  private GraphPanel panel;
  private JComboBox<String> weightedStartCityComboBox;
  private JComboBox<String> weightedEndCityComboBox;
  private Dijkstra dijkstra;
  private JCheckBox chckbxShowEdgeWeights;

  /**
   * Launch the application.
   */
  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        try {
          Display frame = new Display();
          frame.setVisible(true);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
  }

  public static Dijkstra readGraph(String vertexFile, String edgeFile) {

    Dijkstra dijkstra = new Dijkstra();
    try {
      String line;
      // Read in the vertices
      BufferedReader vertexFileBr = new BufferedReader(new FileReader(vertexFile));
      while ((line = vertexFileBr.readLine()) != null) {
        String[] parts = line.split(",");
        if (parts.length != 3) {
          vertexFileBr.close();
          throw new IOException("Invalid line in vertex file " + line);
        }
        String cityname = parts[0];
        int x = Integer.valueOf(parts[1]);
        int y = Integer.valueOf(parts[2]);
        Vertex vertex = new Vertex(cityname, x, y);
```

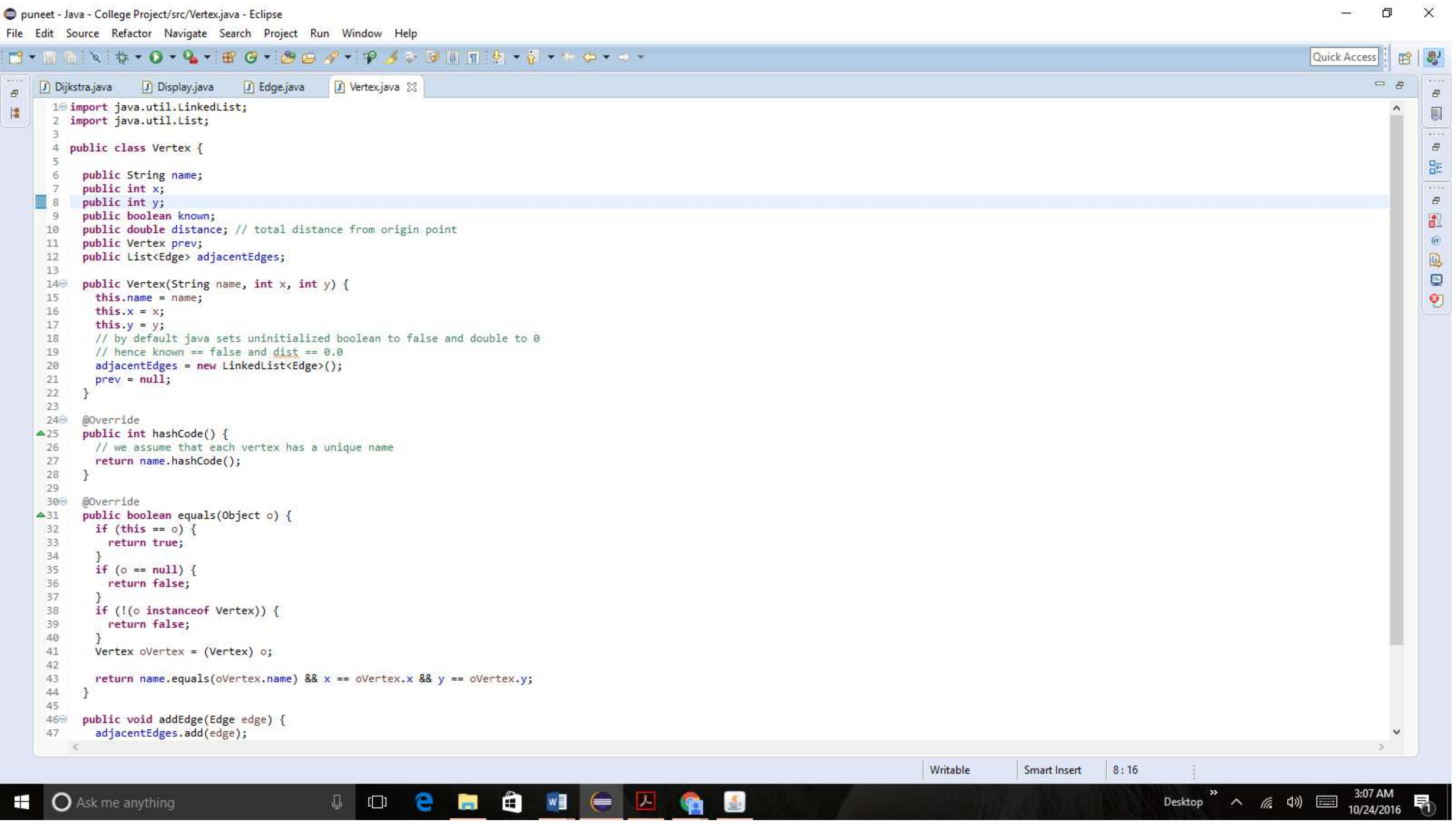
Layout in Display Class



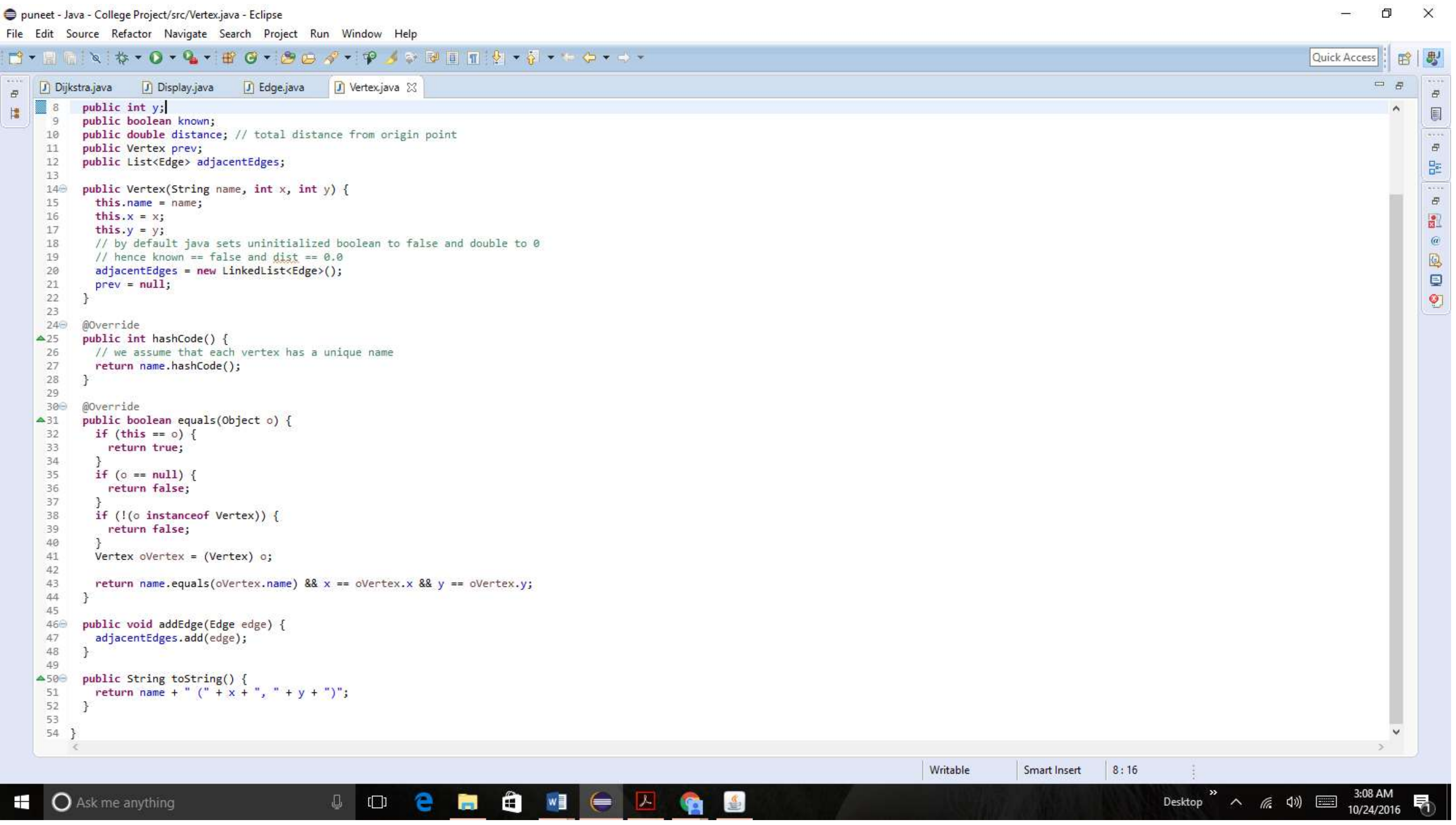
```
   * Create the frame.
   */
  public Display() {
    setTitle("Data Structures Dijkstra Visualizer");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(50, 50, 900, 700);
    setMinimumSize(new Dimension(800, 600));
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    GridBagLayout gbl_contentPane = new GridBagLayout();
    gbl_contentPane.columnWidths = new int[] { 0, 0, 0, 0, 0, 0, 0 };
    gbl_contentPane.rowHeights = new int[] { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    gbl_contentPane.columnWeights = new double[] { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, Double.MIN_VALUE };
    gbl_contentPane.rowWeights = new double[] { 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, Double.MIN_VALUE };
    contentPane.setLayout(gbl_contentPane);

    dijkstra = readGraph("cityxy.txt", "citypairs.txt");

    panel = new GraphPanel(dijkstra);
    GridBagConstraints gbc_panel = new GridBagConstraints();
    gbc_panel.gridwidth = 7;
    gbc_panel.insets = new Insets(0, 0, 7, 0);
    gbc_panel.fill = GridBagConstraints.BOTH;
    gbc_panel.gridx = 0;
    gbc_panel.gridy = 0;
    contentPane.add(panel, gbc_panel);

    JSeparator separator = new JSeparator();
    separator.setBackground(Color.WHITE);
    separator.setForeground(Color.GRAY);
    GridBagConstraints gbc_separator = new GridBagConstraints();
    gbc_separator.fill = GridBagConstraints.BOTH;
    gbc_separator.gridwidth = 6;
    gbc_separator.insets = new Insets(0, 0, 5, 0);
    gbc_separator.gridx = 0;
    gbc_separator.gridy = 2;
    contentPane.add(separator, gbc_separator);

    chckbxShowEdgeWeights = new JCheckBox("Show Edge Weights");
    chckbxShowEdgeWeights.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
        repaint();
      }
    });

    JLabel lblReloadGraph = new JLabel("Load / Reload Dijkstra");
```

Display class in java is responsible for the graphical user interface of this application. Without this class it would be impossible to use this application as a Real-Time application.
It helps us in visualizing our map and set our path accordingly.

Vertex class


puneet - Java - College Project/src/Vertex.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Dijkstra.java
Display.java
Edge.java
Vertex.java
Writable
Smart Insert
8 : 16
Ask me anything
Desktop
3:07 AM
10/24/2016

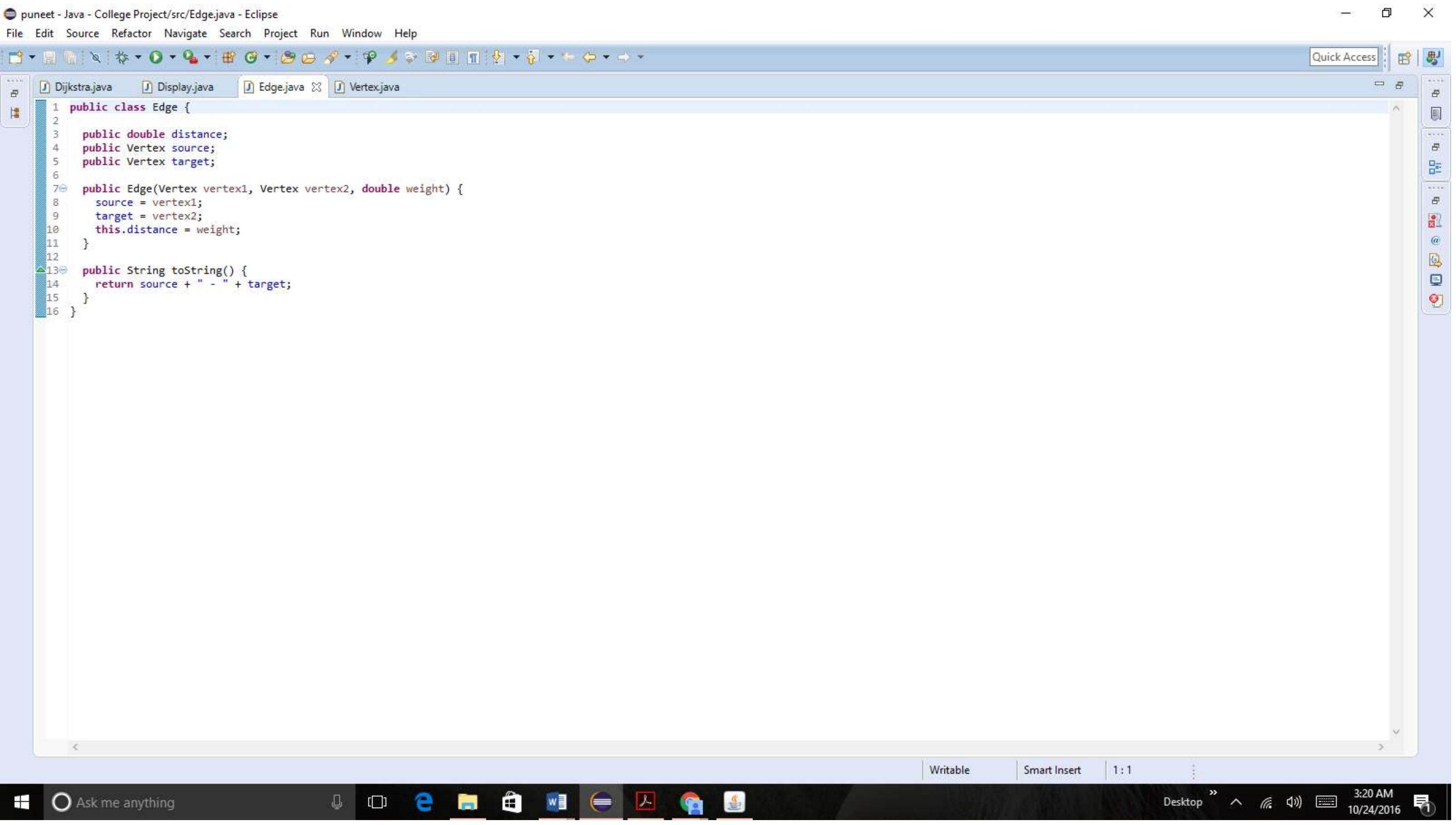
```java
import java.util.LinkedList;
import java.util.List;

public class Vertex {

  public String name;
  public int x;
  public int y;
  public boolean known;
  public double distance; // total distance from origin point
  public Vertex prev;
  public List<Edge> adjacentEdges;

  public Vertex(String name, int x, int y) {
    this.name = name;
    this.x = x;
    this.y = y;
    // by default java sets uninitialized boolean to false and double to 0
    // hence known == false and dist == 0.0
    adjacentEdges = new LinkedList<Edge>();
    prev = null;
  }

  @Override
  public int hashCode() {
    // we assume that each vertex has a unique name
    return name.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null) {
      return false;
    }
    if (!(o instanceof Vertex)) {
      return false;
    }
    Vertex oVertex = (Vertex) o;

    return name.equals(oVertex.name) && x == oVertex.x && y == oVertex.y;
  }

  public void addEdge(Edge edge) {
    adjacentEdges.add(edge);
```


puneet - Java - College Project/src/Vertex.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Dijkstra.java
Display.java
Edge.java
Vertex.java
Writable
Smart Insert
8 : 16
Ask me anything
Desktop
3:08 AM
10/24/2016

```java
  public int y;
  public boolean known;
  public double distance; // total distance from origin point
  public Vertex prev;
  public List<Edge> adjacentEdges;

  public Vertex(String name, int x, int y) {
    this.name = name;
    this.x = x;
    this.y = y;
    // by default java sets uninitialized boolean to false and double to 0
    // hence known == false and dist == 0.0
    adjacentEdges = new LinkedList<Edge>();
    prev = null;
  }

  @Override
  public int hashCode() {
    // we assume that each vertex has a unique name
    return name.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null) {
      return false;
    }
    if (!(o instanceof Vertex)) {
      return false;
    }
    Vertex oVertex = (Vertex) o;

    return name.equals(oVertex.name) && x == oVertex.x && y == oVertex.y;
  }

  public void addEdge(Edge edge) {
    adjacentEdges.add(edge);
  }

  public String toString() {
    return name + " (" + x + ", " + y + ")";
  }

}
```

Vertex class in this project is shown as a city with given coordinates, each vertex in the model has some definite coordinates which makes possible for the system to calculate distance between 2 vertices i.e. To calculate distance between 2 different pair of cities if they are directly in contact with each other.

Edge Class



```java
public class Edge {

  public double distance;
  public Vertex source;
  public Vertex target;

  public Edge(Vertex vertex1, Vertex vertex2, double weight) {
    source = vertex1;
    target = vertex2;
    this.distance = weight;
  }

  public String toString() {
    return source + " - " + target;
  }
}
```

Edge class in this model is represented as the distance between 2 cities.

It means Euclidean distances are the weight of these edges and edges used in this projects are bidirectional, if you can from New York to Boston, you can get from Boston to New York.

Dijkstra class

```
import java.util.Collection;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.io.IOException;
import java.io.FileReader;
import java.io.BufferedReader;

public class Dijkstra {

 // Keep a fast index to nodes in the map
 private Map<String, Vertex> vertexNames;

 /**
  * Construct an empty Dijkstra with a map. The map's key is the name of a vertex
  * and the map's value is the vertex object.
  */
 public Dijkstra() {
  vertexNames = new HashMap<String, Vertex>();
 }

 /**
  * Adds a vertex to the dijkstra. Throws IllegalArgumentException if two vertices
  * with the same name are added.
  *
  * @param v
  *         (Vertex) vertex to be added to the dijkstra
  */
 public void addVertex(Vertex v) {
  if (vertexNames.containsKey(v.name))
    throw new IllegalArgumentException("Cannot create new vertex with existing name.");
  vertexNames.put(v.name, v);
 }

 /**
  * Gets a collection of all the vertices in the dijkstra
  *
  * @return (Collection<Vertex>) collection of all the vertices in the dijkstra
  */
 public Collection<Vertex> getVertices() {
  return vertexNames.values();
 }

 /**
  * Gets the vertex object with the given name
  *
  * @param name
  *         (String) name of the vertex object requested
  * @return (Vertex) vertex object associated with the name
  */
 public Vertex getVertex(String name) {
  return vertexNames.get(name);
 }

 /**
  * Adds a directed edge from vertex u to vertex v
  *
  * @param nameU
  *        (String) name of vertex u
  * @param nameV
  *        (String) name of vertex v
  * @param cost
  *        (double) cost of the edge between vertex u and v
  */
 public void addEdge(String nameU, String nameV, Double cost) {
  if (!vertexNames.containsKey(nameU))
    throw new IllegalArgumentException(nameU + " does not exist. Cannot create edge.");
  if (!vertexNames.containsKey(nameV))
    throw new IllegalArgumentException(nameV + " does not exist. Cannot create edge.");
  Vertex sourceVertex = vertexNames.get(nameU);
  Vertex targetVertex = vertexNames.get(nameV);
  Edge newEdge = new Edge(sourceVertex, targetVertex, cost);
  sourceVertex.addEdge(newEdge);
 }

 /**
  * Adds an undirected edge between vertex u and vertex v by adding a directed
  * edge from u to v, then a directed edge from v to u
  *
  * @param nameU
  *        (String) name of vertex u
  * @param nameV
  *        (String) name of vertex v
  * @param cost
  *        (double) cost of the edge between vertex u and v
  */
 public void addUndirectedEdge(String nameU, String nameV, double cost) {
  addEdge(nameU, nameV, cost);
  addEdge(nameV, nameU, cost);
 }

 // STUDENT CODE STARTS HERE

 /**
  * Computes the euclidean distance between two points as described by their
  * coordinates
  *
  * @param ux
  *         (double) x coordinate of point u
  * @param uy
  *         (double) y coordinate of point u
  * @param vx
  *         (double) x coordinate of point v
  * @param vy
  *         (double) y coordinate of point v
  * @return (double) distance between the two points
  */
 public double computeEuclideanDistance(double ux, double uy, double vx, double vy) {
     return Math.sqrt( Math.pow((ux - vx), 2) + Math.pow((uy - vy),2) );
 }

 /**
  * Calculates the euclidean distance for all edges in the map using the
  * computeEuclideanDistance method.
  */
 public void computeAllEuclideanDistances() {
     for (Vertex vertex : getVertices() )
       for (Edge edge : vertex.adjacentEdges)
         edge.distance = computeEuclideanDistance(edge.source.x,
                          edge.source.y, edge.target.x, edge.target.y);
 }

 /**
  * Dijkstra's Algorithm. p.379 Weiss
  *
  * @param s
  *         (String) starting city name
  */
 public void doDijkstra(String s) {

  LinkedList<Vertex> queue = new LinkedList<>();

  for (Vertex v : getVertices() )
  {
    v.distance = Double.POSITIVE_INFINITY;
    v.known = false;
  }

  Vertex start = getVertex(s);
  start.distance = 0;
  queue.add(start);

  while(!queue.isEmpty())
  {
    Vertex v = queue.poll();

    v.known = true;

    for (Edge e : v.adjacentEdges)
    {
        Vertex w = e.target;
        if(!w.known)
        {
          Double cvw = e.distance;

          if (v.distance + cvw < w.distance)
          {
            w.distance = v.distance + cvw;
            w.prev = v;
            queue.add(w);
          }
        }
      }
    }
  }

  /**
   * Returns a list of edges for a path from city s to city t. This will be the
   * shortest path from s to t as prescribed by Dijkstra's algorithm
   *
   * @param s
   *         (String) starting city name
   * @param t
   *         (String) ending city name
   * @return (List<Edge>) list of edges from s to t
   */
  public List<Edge> getDijkstraPath(String s, String t) {
    doDijkstra(s);

    LinkedList<Edge> path = new LinkedList<>();
    for (Vertex v = getVertex(t); v.prev != null; v = v.prev)
      path.addFirst(new Edge(v.prev, v,
        computeEuclideanDistance(v.prev.x, v.prev.y, v.x, v.y)));

    return path;
  }

  // STUDENT CODE ENDS HERE

  /**
   * Prints out the adjacency list of the dijkstra for debugging
   */
  public void printAdjacencyList() {
    for (String u : vertexNames.keySet()) {
      StringBuilder sb = new StringBuilder();
      sb.append(u);
      sb.append(" -> [ ");
      for (Edge e : vertexNames.get(u).adjacentEdges) {
        sb.append(e.target.name);
        sb.append("(");
        sb.append(e.distance);
        sb.append(") ");
      }
      sb.append("]");
      System.out.println(sb.toString());
    }
  }


  /**
   * A main method that illustrates how the GUI uses Dijkstra.java to
   * read a map and represent it as a graph.
   * You can modify this method to test your code on the command line.
   */
  public static void main(String[] argv) throws IOException {
    String vertexFile = "cityxy.txt";
    String edgeFile = "citypairs.txt";

    Dijkstra dijkstra = new Dijkstra();
    String line;

    // Read in the vertices
    BufferedReader vertexFileBr = new BufferedReader(new FileReader(vertexFile));
    while ((line = vertexFileBr.readLine()) != null) {
      String[] parts = line.split(",");
      if (parts.length != 3) {
        vertexFileBr.close();
        throw new IOException("Invalid line in vertex file " + line);
      }
      String cityname = parts[0];
      int x = Integer.valueOf(parts[1]);
      int y = Integer.valueOf(parts[2]);
      Vertex vertex = new Vertex(cityname, x, y);
      dijkstra.addVertex(vertex);
    }
    vertexFileBr.close();

    BufferedReader edgeFileBr = new BufferedReader(new FileReader(edgeFile));
    while ((line = edgeFileBr.readLine()) != null) {
      String[] parts = line.split(",");
      if (parts.length != 3) {
        edgeFileBr.close();
        throw new IOException("Invalid line in edge file " + line);
      }
      dijkstra.addUndirectedEdge(parts[0], parts[1], Double.parseDouble(parts[2]));
    }
    edgeFileBr.close();

    // Compute distances.
    // This is what happens when you click on the "Compute All Euclidean Distances" button.
    dijkstra.computeAllEuclideanDistances();

    // print out an adjacency list representation of the graph
    dijkstra.printAdjacencyList();

    // This is what happens when you click on the "Draw Dijkstra's Path" button.

    // In the GUI, these are set through the drop-down menus.
    String startCity = "SanFrancisco";
    String endCity = "Boston";

    // Get weighted shortest path between start and end city.
    List<Edge> path = dijkstra.getDijkstraPath(startCity, endCity);

    System.out.print("Shortest path between "+startCity+" and "+endCity+": ");
    System.out.println(path);
  }

}
```

Display class

```
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Desktop;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.RenderingHints;
import java.awt.Stroke;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.ArrayList;
import java.util.List;
import java.util.Collections;
import javax.swing.DefaultComboBoxModel;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JSeparator;
import javax.swing.JTextField;
import javax.swing.UIManager;
import javax.swing.border.EmptyBorder;

@SuppressWarnings("serial")
public class Display extends JFrame {

  private JPanel contentPane;
  private JTextField txtCityxytxt;
  private JTextField txtCitypairstxt;
  private GraphPanel panel;
  private JComboBox<String> weightedStartCityComboBox;
  private JComboBox<String> weightedEndCityComboBox;
  private Dijkstra dijkstra;
  private JCheckBox chckbxShowEdgeWeights;

  /**
   * Launch the application.
   */
  public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
      public void run() {
        try {
          Display frame = new Display();
          frame.setVisible(true);
        } catch (Exception e) {
          e.printStackTrace();
        }
      }
    });
  }

  public static Dijkstra readGraph(String vertexFile, String edgeFile) {

    Dijkstra dijkstra = new Dijkstra();
    try {
      String line;
      // Read in the vertices
      BufferedReader vertexFileBr = new BufferedReader(new FileReader(vertexFile));
      while ((line = vertexFileBr.readLine()) != null) {
        String[] parts = line.split(",");
        if (parts.length != 3) {
          vertexFileBr.close();
          throw new IOException("Invalid line in vertex file " + line);
        }
        String cityname = parts[0];
        int x = Integer.valueOf(parts[1]);
        int y = Integer.valueOf(parts[2]);
        Vertex vertex = new Vertex(cityname, x, y);
        dijkstra.addVertex(vertex);
      }
      vertexFileBr.close();
      // Now read in the edges
      BufferedReader edgeFileBr = new BufferedReader(new FileReader(edgeFile));
      while ((line = edgeFileBr.readLine()) != null) {
        String[] parts = line.split(",");
        if (parts.length != 3) {
          edgeFileBr.close();
          throw new IOException("Invalid line in edge file " + line);
        }
        dijkstra.addUndirectedEdge(parts[0], parts[1], Double.parseDouble(parts[2]));
      }
      edgeFileBr.close();
    } catch (IOException e) {
      System.err.println("Could not read the dijkstra: " + e);
      return null;
    }
    return dijkstra;
  }

  /**
   * Create the frame.
   */
  public Display() {
    setTitle("Data Structures Dijkstra Visualizer");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(50, 50, 900, 700);
    setMinimumSize(new Dimension(800, 600));
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    setContentPane(contentPane);
    GridBagLayout gbl_contentPane = new GridBagLayout();
    gbl_contentPane.columnWidths = new int[] { 0, 0, 0, 0, 0, 0, 0 };
    gbl_contentPane.rowHeights = new int[] { 0, 0, 0, 0, 0, 0, 0, 0, 0 };
```

```
    gbl_contentPane.columnWeights = new double[] { 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
Double.MIN_VALUE };
    gbl_contentPane.rowWeights = new double[] { 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
Double.MIN_VALUE };
    contentPane.setLayout(gbl_contentPane);

    dijkstra = readGraph("cityxy.txt", "citypairs.txt");

    panel = new GraphPanel(dijkstra);
    GridBagConstraints gbc_panel = new GridBagConstraints();
    gbc_panel.gridwidth = 7;
    gbc_panel.insets = new Insets(0, 0, 7, 0);
    gbc_panel.fill = GridBagConstraints.BOTH;
    gbc_panel.gridx = 0;
    gbc_panel.gridy = 0;
    contentPane.add(panel, gbc_panel);

    JSeparator separator = new JSeparator();
    separator.setBackground(Color.WHITE);
    separator.setForeground(Color.GRAY);
    GridBagConstraints gbc_separator = new GridBagConstraints();
    gbc_separator.fill = GridBagConstraints.BOTH;
    gbc_separator.gridwidth = 6;
    gbc_separator.insets = new Insets(0, 0, 5, 0);
    gbc_separator.gridx = 0;
    gbc_separator.gridy = 2;
    contentPane.add(separator, gbc_separator);

    chckbxShowEdgeWeights = new JCheckBox("Show Edge Weights");
    chckbxShowEdgeWeights.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {
```

```
    repaint();
  }
});

JLabel lblReloadGraph = new JLabel("Load / Reload Dijkstra");
GridBagConstraints gbc_lblReloadGraph = new GridBagConstraints();
gbc_lblReloadGraph.anchor = GridBagConstraints.EAST;
gbc_lblReloadGraph.insets = new Insets(0, 0, 5, 5);
gbc_lblReloadGraph.gridx = 0;
gbc_lblReloadGraph.gridy = 3;
contentPane.add(lblReloadGraph, gbc_lblReloadGraph);

txtCityxytxt = new JTextField();
txtCityxytxt.setText("cityxy.txt");
GridBagConstraints gbc_txtCityxytxt = new GridBagConstraints();
gbc_txtCityxytxt.gridwidth = 2;
gbc_txtCityxytxt.insets = new Insets(0, 0, 5, 5);
gbc_txtCityxytxt.fill = GridBagConstraints.HORIZONTAL;
gbc_txtCityxytxt.gridx = 1;
gbc_txtCityxytxt.gridy = 3;
contentPane.add(txtCityxytxt, gbc_txtCityxytxt);
txtCityxytxt.setColumns(10);

txtCitypairstxt = new JTextField();
txtCitypairstxt.setText("citypairs.txt");
GridBagConstraints gbc_txtCitypairstxt = new GridBagConstraints();
gbc_txtCitypairstxt.gridwidth = 2;
gbc_txtCitypairstxt.insets = new Insets(0, 0, 5, 5);
gbc_txtCitypairstxt.fill = GridBagConstraints.HORIZONTAL;
gbc_txtCitypairstxt.gridx = 3;
gbc_txtCitypairstxt.gridy = 3;
```

```
contentPane.add(txtCitypairstxt, gbc_txtCitypairstxt);
txtCitypairstxt.setColumns(10);

JButton btnReloadGraph = new JButton("Load / Reset");
GridBagConstraints gbc_btnReloadGraph = new GridBagConstraints();
gbc_btnReloadGraph.fill = GridBagConstraints.HORIZONTAL;
gbc_btnReloadGraph.insets = new Insets(0, 0, 5, 0);
gbc_btnReloadGraph.gridx = 5;
gbc_btnReloadGraph.gridy = 3;
contentPane.add(btnReloadGraph, gbc_btnReloadGraph);

btnReloadGraph.addMouseListener(new MouseAdapter() {
  @Override
  public void mouseReleased(MouseEvent e) {
    // update JPanel
    updateGraphPanel();
  }
});

JLabel lblEuclideanCosts = new JLabel("Euclidean Costs");
GridBagConstraints gbc_lblEuclideanCosts = new GridBagConstraints();
gbc_lblEuclideanCosts.anchor = GridBagConstraints.EAST;
gbc_lblEuclideanCosts.insets = new Insets(0, 0, 5, 5);
gbc_lblEuclideanCosts.gridx = 0;
gbc_lblEuclideanCosts.gridy = 4;
contentPane.add(lblEuclideanCosts, gbc_lblEuclideanCosts);
chckbxShowEdgeWeights.setSelected(true);
GridBagConstraints gbc_chckbxShowEdgeWeights = new GridBagConstraints();
gbc_chckbxShowEdgeWeights.anchor = GridBagConstraints.EAST;
gbc_chckbxShowEdgeWeights.gridwidth = 4;
gbc_chckbxShowEdgeWeights.insets = new Insets(0, 0, 5, 5);
gbc_chckbxShowEdgeWeights.gridx = 1;
gbc_chckbxShowEdgeWeights.gridy = 4;
contentPane.add(chckbxShowEdgeWeights, gbc_chckbxShowEdgeWeights);

JButton btnComputeAllEuclidean = new JButton("Compute All Euclidean Distances");
GridBagConstraints gbc_btnComputeAllEuclidean = new GridBagConstraints();
gbc_btnComputeAllEuclidean.fill = GridBagConstraints.BOTH;
gbc_btnComputeAllEuclidean.insets = new Insets(0, 0, 5, 0);
gbc_btnComputeAllEuclidean.gridx = 5;
gbc_btnComputeAllEuclidean.gridy = 4;
contentPane.add(btnComputeAllEuclidean, gbc_btnComputeAllEuclidean);

btnComputeAllEuclidean.addMouseListener(new MouseAdapter() {
  @Override
  public void mouseReleased(MouseEvent e) {
    panel.dijkstra.computeAllEuclideanDistances();
    repaint();
  }
});

JLabel lblGetWeightedShortest_1 = new JLabel("Dijkstra's Algorithm");
GridBagConstraints gbc_lblGetWeightedShortest_1 = new GridBagConstraints();
gbc_lblGetWeightedShortest_1.anchor = GridBagConstraints.EAST;
gbc_lblGetWeightedShortest_1.insets = new Insets(0, 0, 5, 5);
gbc_lblGetWeightedShortest_1.gridx = 0;
gbc_lblGetWeightedShortest_1.gridy = 5;
contentPane.add(lblGetWeightedShortest_1, gbc_lblGetWeightedShortest_1);

JLabel lblStart_1 = new JLabel("Start");
GridBagConstraints gbc_lblStart_1 = new GridBagConstraints();
gbc_lblStart_1.insets = new Insets(0, 0, 5, 5);
```

```
gbc_lblStart_1.anchor = GridBagConstraints.EAST;
gbc_lblStart_1.gridx = 1;
gbc_lblStart_1.gridy = 5;
contentPane.add(lblStart_1, gbc_lblStart_1);

weightedStartCityComboBox = new JComboBox<>();
weightedStartCityComboBox.setToolTipText("");
GridBagConstraints gbc_weightedStartCityComboBox = new GridBagConstraints();
gbc_weightedStartCityComboBox.insets = new Insets(0, 0, 5, 5);
gbc_weightedStartCityComboBox.fill = GridBagConstraints.HORIZONTAL;
gbc_weightedStartCityComboBox.gridx = 2;
gbc_weightedStartCityComboBox.gridy = 5;
contentPane.add(weightedStartCityComboBox, gbc_weightedStartCityComboBox);

JLabel lblEnd_1 = new JLabel("End");
GridBagConstraints gbc_lblEnd_1 = new GridBagConstraints();
gbc_lblEnd_1.insets = new Insets(0, 0, 5, 5);
gbc_lblEnd_1.anchor = GridBagConstraints.EAST;
gbc_lblEnd_1.gridx = 3;
gbc_lblEnd_1.gridy = 5;
contentPane.add(lblEnd_1, gbc_lblEnd_1);

weightedEndCityComboBox = new JComboBox<>();
GridBagConstraints gbc_weightedEndCityComboBox = new GridBagConstraints();
gbc_weightedEndCityComboBox.insets = new Insets(0, 0, 5, 5);
gbc_weightedEndCityComboBox.fill = GridBagConstraints.HORIZONTAL;
gbc_weightedEndCityComboBox.gridx = 4;
gbc_weightedEndCityComboBox.gridy = 5;
contentPane.add(weightedEndCityComboBox, gbc_weightedEndCityComboBox);

JButton btnDrawWeightedShortest = new JButton("Draw Dijkstra's Path");
```

```
btnDrawWeightedShortest.setForeground(Color.GREEN);
GridBagConstraints gbc_btnDrawWeightedShortest = new GridBagConstraints();
gbc_btnDrawWeightedShortest.fill = GridBagConstraints.HORIZONTAL;
gbc_btnDrawWeightedShortest.insets = new Insets(0, 0, 5, 0);
gbc_btnDrawWeightedShortest.gridx = 5;
gbc_btnDrawWeightedShortest.gridy = 5;
contentPane.add(btnDrawWeightedShortest, gbc_btnDrawWeightedShortest);

JLabel lblPanda = new JLabel("Panda 2016 <http://linanqiu.github.io/>");
lblPanda.setForeground(UIManager.getColor("TextField.light"));
lblPanda.addMouseListener(new MouseAdapter() {
  @Override
  public void mouseReleased(MouseEvent e) {
    try {
      Desktop.getDesktop().browse(new URI("http://linanqiu.github.io/"));
    } catch (IOException e1) {
      // TODO Auto-generated catch block
      e1.printStackTrace();
    } catch (URISyntaxException e1) {
      // TODO Auto-generated catch block
      e1.printStackTrace();
    }
  }
});

JSeparator separator_1 = new JSeparator();
separator_1.setForeground(Color.GRAY);
separator_1.setBackground(Color.WHITE);
GridBagConstraints gbc_separator_1 = new GridBagConstraints();
gbc_separator_1.fill = GridBagConstraints.BOTH;
gbc_separator_1.gridwidth = 6;
```

```
    gbc_separator_1.insets = new Insets(0, 0, 5, 0);
    gbc_separator_1.gridx = 0;
    gbc_separator_1.gridy = 6;
    contentPane.add(separator_1, gbc_separator_1);
    GridBagConstraints gbc_lblPanda = new GridBagConstraints();
    gbc_lblPanda.gridwidth = 3;
    gbc_lblPanda.anchor = GridBagConstraints.EAST;
    gbc_lblPanda.gridx = 3;
    gbc_lblPanda.gridy = 7;
    contentPane.add(lblPanda, gbc_lblPanda);

    btnDrawWeightedShortest.addMouseListener(new MouseAdapter() {
      @Override
      public void mouseReleased(MouseEvent e) {
        String startCity =
weightedStartCityComboBox.getItemAt(weightedStartCityComboBox.getSelectedIndex());
        String endCity =
weightedEndCityComboBox.getItemAt(weightedEndCityComboBox.getSelectedIndex());
        System.out.println("Calculating shortest weighted path for " + startCity + " to " + endCity);
        List<Edge> weightedPath = dijkstra.getDijkstraPath(startCity, endCity);
        panel.overlayEdges.put("weighted", weightedPath);
        repaint();
      }
    });

    updateGraphPanel();
  }

  private void updateGraphPanel() {
    String vertexFile = txtCityxytxt.getText();
    String edgeFile = txtCitypairstxt.getText();
  dijkstra = readGraph(vertexFile, edgeFile);
  panel.dijkstra = dijkstra;
  System.out.println("Constructing new file from " + vertexFile + " and " + edgeFile);
  System.out.println("Data read: " + panel.dijkstra.getVertices());

  List<String> cityNameList = new ArrayList<>();
  for (Vertex v : dijkstra.getVertices())
    cityNameList.add(v.name);
  Collections.sort(cityNameList);
  String[] cityNames = cityNameList.toArray(new String[cityNameList.size()]);
  weightedStartCityComboBox.setModel(new DefaultComboBoxModel<>(cityNames));
  weightedEndCityComboBox.setModel(new DefaultComboBoxModel<>(cityNames));

  panel.overlayEdges.put("weighted", new LinkedList<Edge>());
  panel.overlayEdges.put("unweighted", new LinkedList<Edge>());
  panel.overlayEdges.put("mst", new LinkedList<Edge>());

  repaint();
}

public class GraphPanel extends JPanel {

  // dijkstra layout parameters
  public static final int VERTEX_RADIUS = 10;
  public static final int SPACE = 3;

  public static final int MARGIN_X = 50;
  public static final int MARGIN_Y = 50;

  public static final int DEFAULT_THICKNESS = 1;
  // scale factors
  public float xFactor, yFactor;

  public Dijkstra dijkstra;

  public HashMap<String, List<Edge>> overlayEdges;

  public GraphPanel(Dijkstra dijkstra) {
    this.dijkstra = dijkstra;
    overlayEdges = new HashMap<>();
    overlayEdges.put("weighted", new LinkedList<Edge>());
    overlayEdges.put("unweighted", new LinkedList<Edge>());
    overlayEdges.put("mst", new LinkedList<Edge>());
  }

  public void paintComponent(Graphics g) {
    // make everything smooth like butter
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2.setRenderingHint(RenderingHints.KEY_DITHERING,
RenderingHints.VALUE_DITHER_ENABLE);
    g2.setRenderingHint(RenderingHints.KEY_RENDERING,
RenderingHints.VALUE_RENDER_QUALITY);
    g2.setRenderingHint(RenderingHints.KEY_FRACTIONALMETRICS,
RenderingHints.VALUE_FRACTIONALMETRICS_ON);
    g2.setRenderingHint(RenderingHints.KEY_ALPHA_INTERPOLATION,
RenderingHints.VALUE_ALPHA_INTERPOLATION_QUALITY);
```

```
    g2.setRenderingHint(RenderingHints.KEY_COLOR_RENDERING,
RenderingHints.VALUE_COLOR_RENDER_QUALITY);
    g2.setRenderingHint(RenderingHints.KEY_STROKE_CONTROL,
RenderingHints.VALUE_STROKE_PURE);

    // scale the dijkstra
    int minX = 0;
    int maxX = 1;
    int minY = 0;
    int maxY = 1;
    for (Vertex v : dijkstra.getVertices()) {
      if (v.x < minX)
        minX = v.x;
      if (v.x > maxX)
        maxX = v.x;
      if (v.y < minY)
        minY = v.y;
      if (v.y > maxY)
        maxY = v.y;
    }
    xFactor = (this.getBounds().width - 2 * MARGIN_X) / (float) (maxX - minX);
    yFactor = (this.getBounds().height - 2 * MARGIN_Y) / (float) (maxY - minY);
    super.paintComponent(g2); // paint the panel
    try {
      paintGraph(g2); // paint the dijkstra
    } catch (NullPointerException e) {
      e.printStackTrace();
    }
  }

  public void paintGraph(Graphics g) {
    for (Vertex v : dijkstra.getVertices()) {
      for (Edge edge : v.adjacentEdges) {
        paintEdge(g, edge.source, edge.target, edge.distance, Color.LIGHT_GRAY,
DEFAULT_THICKNESS, 255);
      }
    }
    for (Vertex v : dijkstra.getVertices()) {
      paintVertex(g, v);
    }
    for (String overlayType : overlayEdges.keySet()) {
      if (overlayType.equals("unweighted")) {
        for (Edge edge : overlayEdges.get(overlayType)) {
          paintEdge(g, edge.source, edge.target, edge.distance, Color.RED, 8, 50);
        }
      }
      if (overlayType.equals("weighted")) {
        for (Edge edge : overlayEdges.get(overlayType)) {
          paintEdge(g, edge.source, edge.target, edge.distance, Color.GREEN, 8, 50);
        }
      }
      if (overlayType.equals("mst")) {
        for (Edge edge : overlayEdges.get(overlayType)) {
          paintEdge(g, edge.source, edge.target, edge.distance, Color.BLUE, 8, 50);
        }
      }
    }
  }

  public void paintVertex(Graphics g, Vertex v) {
    Graphics2D g2 = (Graphics2D) g;
```

```
      int x = Math.round(xFactor * (float) v.x + (float) MARGIN_X);
      int y = Math.round(yFactor * (float) v.y + (float) MARGIN_Y);
      g2.setColor(Color.LIGHT_GRAY);
      Stroke oldStroke = g2.getStroke();
      g2.setStroke(new BasicStroke(4));
      g2.drawOval(x - VERTEX_RADIUS / 2, y - VERTEX_RADIUS / 2, VERTEX_RADIUS,
VERTEX_RADIUS);
      g2.setStroke(oldStroke);
      g2.setColor(Color.LIGHT_GRAY);
      g2.fillOval(x - VERTEX_RADIUS / 2, y - VERTEX_RADIUS / 2, VERTEX_RADIUS,
VERTEX_RADIUS);
      g2.setColor(Color.DARK_GRAY);
      g2.drawString(v.name, x - v.name.length() * 8 / 2, y + VERTEX_RADIUS / 2);
    }

    public void paintEdge(Graphics g, Vertex u, Vertex v, double weight, Color color, int
thickness, int alpha) {
      Graphics2D g2 = (Graphics2D) g;
      int x1 = Math.round(xFactor * (float) u.x + (float) MARGIN_X);
      int y1 = Math.round(yFactor * (float) u.y + (float) MARGIN_Y);
      int x2 = Math.round(xFactor * (float) v.x + (float) MARGIN_X);
      int y2 = Math.round(yFactor * (float) v.y + (float) MARGIN_Y);
      g2.setColor(new Color(color.getRed(), color.getGreen(), color.getBlue(), alpha));
      Stroke oldStroke = g2.getStroke();
      g2.setStroke(new BasicStroke(thickness));
      g2.drawLine(x1, y1, x2, y2);
      g2.setStroke(oldStroke);
      if (chckbxShowEdgeWeights.isSelected()) {
        Font oldFont = g2.getFont();
        g2.setFont(new Font("Helvetica", Font.PLAIN, 8));
        g2.setColor(Color.GRAY);
        g2.drawString(String.format("%.1f", weight), (x1 + x2) / 2, (y1 + y2) / 2);
        g2.setFont(oldFont);
      }
    }
  }
}
```

Edge class

```
public class Edge {

  public double distance;
  public Vertex source;
  public Vertex target;

  public Edge(Vertex vertex1, Vertex vertex2, double weight) {
    source = vertex1;
    target = vertex2;
    this.distance = weight;
  }

  public String toString() {
    return source + " - " + target;
  }
}
```

Vertex Class

```
import java.util.LinkedList;
import java.util.List;

public class Vertex {

  public String name;
  public int x;
  public int y;
  public boolean known;
  public double distance; // total distance from origin point
  public Vertex prev;
  public List<Edge> adjacentEdges;

  public Vertex(String name, int x, int y) {
    this.name = name;
    this.x = x;
    this.y = y;
    // by default java sets uninitialized boolean to false and double to 0
    // hence known == false and dist == 0.0
    adjacentEdges = new LinkedList<Edge>();
    prev = null;
  }

  @Override
  public int hashCode() {
    // we assume that each vertex has a unique name
    return name.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) {
      return true;
    }
    if (o == null) {
      return false;
    }
    if (!(o instanceof Vertex)) {
      return false;
    }
    Vertex oVertex = (Vertex) o;

    return name.equals(oVertex.name) && x == oVertex.x && y == oVertex.y;
  }

  public void addEdge(Edge edge) {
    adjacentEdges.add(edge);
  }
  public String toString() {
    return name + " (" + x + ", " + y + ")";
  }

}
```