

**Taller 5**

*Fecha: Noviembre de 2024*

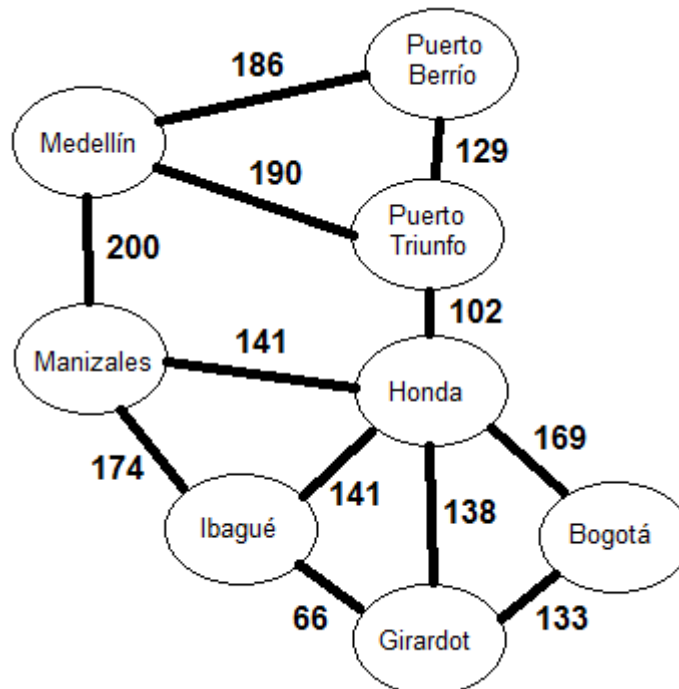
**Indicador de logro a medir:** Aplicar el concepto de Grafo en el desarrollo de una aplicación con interfaz gráfica de usuario basada en el lenguaje Java y un IDE adecuado.

**NOTAS:**

- Este taller se debe hacer como preparación para el quiz. En ningún caso representará una calificación.
- Se entregan ejercicios resueltos como ejemplo para el desarrollo de los demás.

Elaborar el diagrama de clases básico (sin las clases correspondientes a la interface de usuario según el lenguaje de implementación) y la respectiva aplicación en un lenguaje orientado a objetos para los siguientes enunciados:

1. Las distancias en kilómetros entre las principales ciudades de una región se almacenan en un grafo como el siguiente:



Implementar una solución que halle la distancia mínima entre 2 ciudades.

Por ejemplo, si se desea la ruta mínima entre Medellín y Bogotá, se tendrían las siguientes alternativas, siendo la más corta, la segunda:

**Ruta 1**

Ciudad	Km
Medellín	0
Puerto Berrio	186
Puerto Triunfo	315
Honda	417
Bogotá	586

**Ruta 2**

Ciudad	Km
Medellín	0
Puerto Triunfo	190
Honda	292
Bogotá	461

**Ruta 3**

Ciudad	Km
Medellín	0
Manizales	200
Honda	341
Bogotá	510

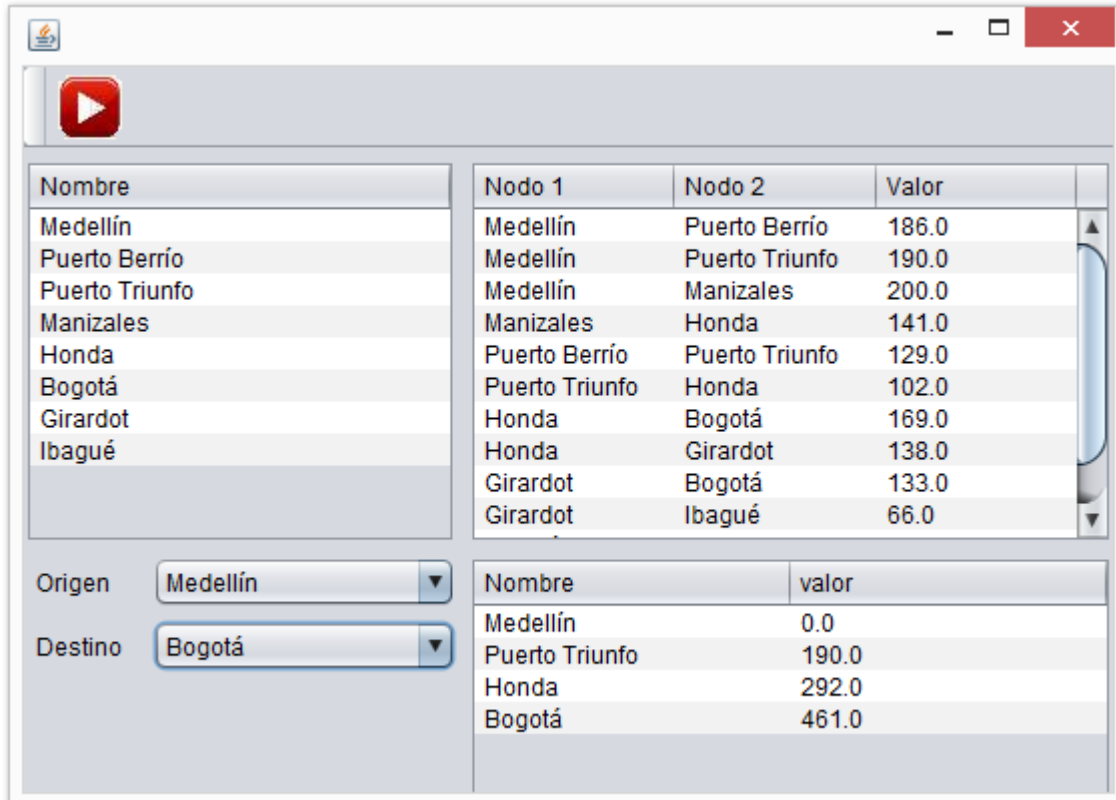
**Ruta 4**

Ciudad	Km
Medellín	0
Manizales	200
Ibagué	374
Girardot	440
Bogotá	573

**Ruta 5**

Ciudad	Km
Medellín	0
Manizales	200
Ibagué	374
Honda	515
Bogotá	684

La ejecución podría lucir así:



The interface shows a list of cities on the left: Medellín, Puerto Berrío, Puerto Triunfo, Manizales, Honda, Bogotá, Girardot, and Ibagué. Below this is a table of edges (aristas) with columns for 'Nodo 1', 'Nodo 2', and 'Valor'. The table lists connections between cities and their distances. At the bottom, there are dropdown menus for 'Origen' (set to Medellín) and 'Destino' (set to Bogotá). To the right of these is a small table showing the cumulative distance to the destination for each city.

Nodo 1	Nodo 2	Valor
Medellín	Puerto Berrío	186.0
Medellín	Puerto Triunfo	190.0
Medellín	Manizales	200.0
Manizales	Honda	141.0
Puerto Berrío	Puerto Triunfo	129.0
Puerto Triunfo	Honda	102.0
Honda	Bogotá	169.0
Honda	Girardot	138.0
Girardot	Bogotá	133.0
Girardot	Ibagué	66.0

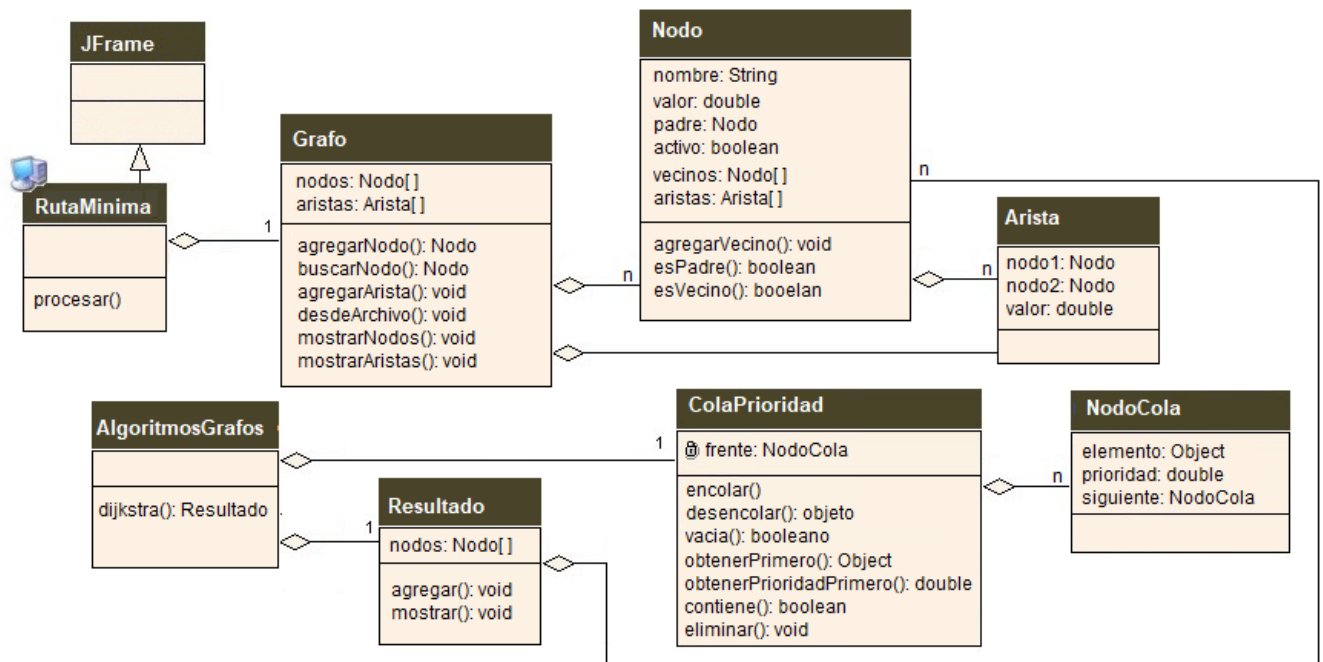
Nombre	valor
Medellín	0.0
Puerto Triunfo	190.0
Honda	292.0
Bogotá	461.0

Donde:

- Se listan los vértices o nodos (Ciudades)
- Se listan las aristas (distancias entre ciudades adyacentes)
- Se permite escoger la ciudad origen y la ciudad destino
- Se muestra la ruta más cercana en kilómetros entre las ciudades origen y destino

R/

- El modelado bajo el paradigma Orientado a Objetos se ilustra en el siguiente diagrama de clases:



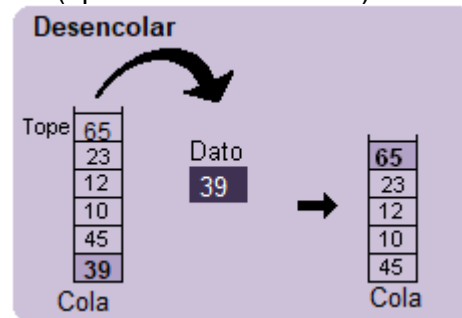
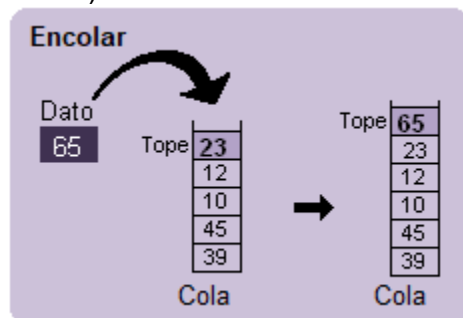
Para comprender este diagrama y el ejercicio, es importante tener en cuenta los siguientes fundamentos:

### Colas

Una **cola** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo y la operación de extracción se realiza por el otro. Es una estructura de tipo *FIFO* (del inglés *First In First Out*), debido a que el primer elemento que ingresa será también el primero en salir.

Son dos los cambios básicos que se pueden hacer en una cola:


- Agregar un nuevo elemento (operación **Encolar**)
- Quitar el primer elemento agregado (operación **Desencolar**)



Para realizar estas operaciones existen algunas restricciones:

- No se puede *desencolar* si no hay elementos en la cola (condición **Cola Vacía**).

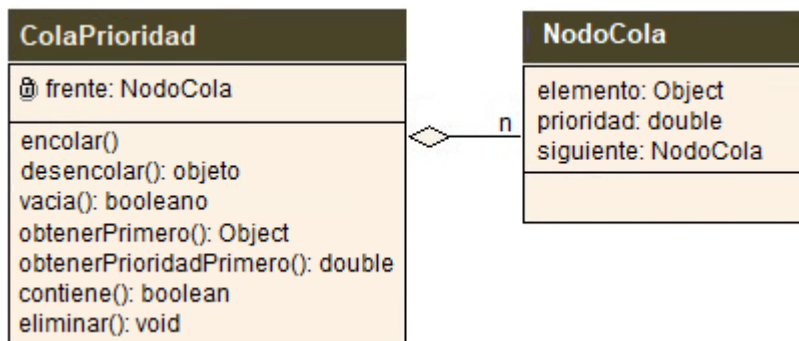
## ColaVacía

Tope   
Cola

## Cola de Prioridad

Una cola de prioridades es una cola en la que los elementos tienen adicionalmente, una prioridad asignada. En una cola de prioridades un elemento con mayor prioridad será desencolado antes que un elemento de menor prioridad. Si dos elementos tienen la misma prioridad, se desencolarán siguiendo el orden de cola.

La siguiente es la descripción de la clase *ColaPrioridad*:



- El atributo *frente* que corresponde al objeto con mayor prioridad
- El método *encolar()* permite encolar un dato en la cola.
- El método *desencolar()* permite desencolar un objeto de la cola (el que tenga mayor prioridad)
- El método *vacía()* devuelve un valor booleano que indica si la cola está vacía

- El método *obtenerPrimero()* permite obtener el objeto que tenga mayor prioridad sin desencolarlo
- El método *obtenerPrioridadPrimero()* retorna la prioridad del objeto a desencolar
- El método *contiene()* retorna un valor booleano indicando si un objeto ya está encolado
- El método *eliminar()* retira de la cola cualquier objeto que haya sido encolado

El siguiente sería el respectivo código en *Java*:

```

//Representa una cola con prioridades
public class ColaPrioridad {

    //Clase componente de la cola
    class NodoCola {

        Object elemento;
        double prioridad;
        NodoCola siguiente;
    }
    //Nodo al frente de la cola
    private NodoCola frente;

    //Metodo constructor
    public ColaPrioridad() {
        frente = null;
    }
}
    
```

```
}

//Indica si la cola esta vacia
public boolean vacia() {
    return (frente == null);
}

//devuelve el primer elemento de la cola
public Object obtenerPrimero() {
    if (vacía()) {
        return null;
    } else {
        return frente.elemento;
    }
}

//devuelve la prioridad del primer elemento de la cola
public double obtenerPrioridadPrimero() {
    if (vacía()) {
        return -1;
    } else {
        return frente.prioridad;
    }
}

//agrega un elemento y lo pone en la cola de acuerdo a la prioridad
public void encolar(Object elemento, double prioridad) {
    NodoCola p, q;
    if (!vacía()) {
        boolean encontrado = false;
        p = frente;
        while (p.siguiente != null && !encontrado) {
            if (p.prioridad < prioridad) {
                encontrado = true;
            } else {
                p = p.siguiente;
            }
        }
        q = p.siguiente;
        p.siguiente = new NodoCola();
        p = p.siguiente;
        p.siguiente = q;
    } else {
        p = new NodoCola();
        frente = p;
    }
    p.elemento = elemento;
    p.prioridad = prioridad;
} //encolar

//elimina el primer elemento de la cola
public Object desencolar() {
    if (vacía()) {
        return null;
    } else {
        Object elemento = frente.elemento;
```

```
        frente = frente.siguiente;
        return elemento;
    }
}

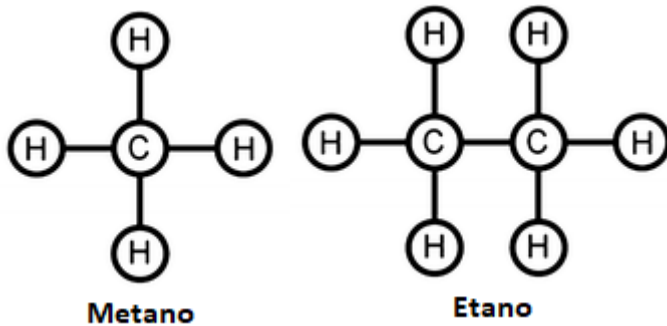
//indica si un elemento esta en la cola
public boolean contiene(Object elemento) {
    boolean encontrado = false;
    NodoCola p = frente;
    while (p != null && !encontrado) {
        if (p.elemento.equals(elemento)) {
            encontrado = true;
        } else {
            p = p.siguiente;
        }
    }
    return encontrado;
}

//Elimina un elemento de la cola
public void eliminar(Object elemento) {
    if (elemento != null && !vacía()) {
        //Buscar el elemento
        boolean encontrado = false;
        NodoCola p = frente;
        NodoCola q = null;
        while (p != null && !encontrado) {
            if (p.elemento.equals(elemento)) {
                encontrado = true;
            } else {
                q = p;
                p = p.siguiente;
            }
        }
        if (encontrado) {
            if (q == null) {
                frente = p.siguiente;
            } else {
                q.siguiente = p.siguiente;
            }
        }
    }
}
```

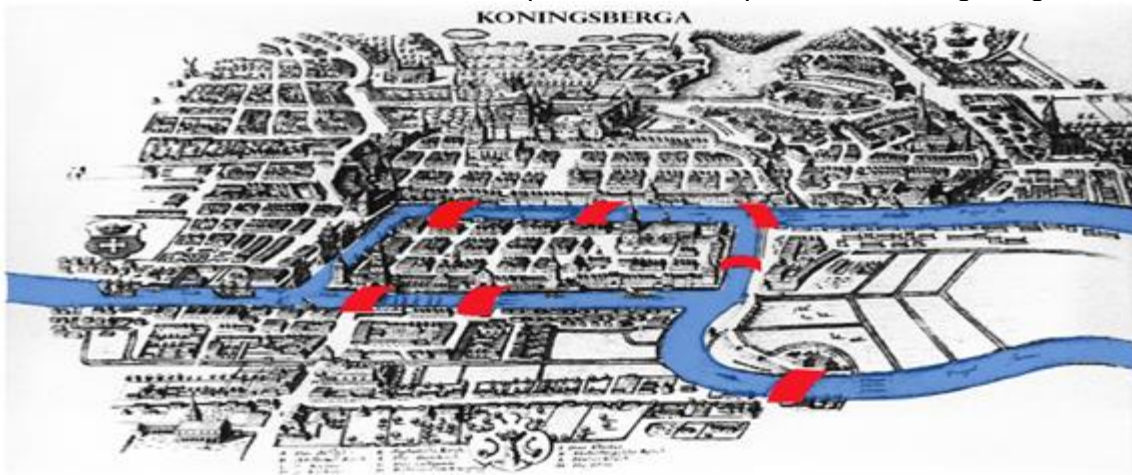
Donde se puede apreciar que se declara una clase interna *NodoCola* que corresponde a la estructura a encolar y que incluye además del objeto encolado, el valor de la prioridad y el apuntador al siguiente nodo, ya que la cola es una lista ligada.

## Grafos

En matemáticas y ciencias de la computación, un **Grafo** es un conjunto de objetos llamados **vértices** o nodos unidos por enlaces llamados **aristas** o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Mediante los grafos se puede describir multitud de procesos, situaciones, relaciones, estructuras, etc. Por ejemplo, un grafo puede servir para representar los enlaces moleculares de un elemento químico.



El primer artículo científico relativo a grafos fue escrito por el matemático suizo Leonhard Euler en 1736. Euler basó en su artículo en el problema de los puentes de Königsberg.

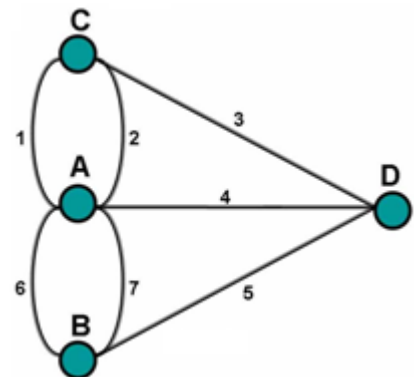


Se buscaba resolver el interrogante: ¿Es posible recorrer cada una de las regiones de esta ciudad pasando por los siete puentes, atravesando sólo una vez cada puente y regresando al mismo punto de donde se partió?

La idea genial de Euler fue representar la ciudad de Königsberg como un grafo en el que las cuatro partes de la misma eran los vértices y los siete puentes eran las aristas.

Por tanto, el problema de los puentes de Königsberg pasó a ser un problema de teoría de grafos cuya solución publicó Euler en su artículo *Solución de un problema relativo a la geometría de posición*.

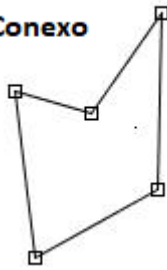
La conclusión fue que no era posible hacer tal recorrido debido a que los vértices son todos con grado impar



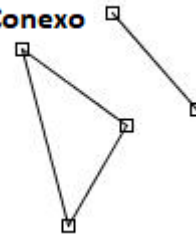
Algunos conceptos básicos de grafos son:

- Se llama **lazo** a una arista que une un vértice consigo mismo
- Se dice que dos vértices son **adyacentes** si existe una arista que los une
- Se dice que un grafo es **simple** si para cualesquiera dos vértices existe a lo sumo una arista que los une. En otro caso se denomina **multigrafo**
- Dado un vértice, se denomina **grado** al número de aristas que inciden en el mismo
- Se dice que un grafo es **conexo** si no puede expresarse como la unión de dos grafos de vértices disjuntos

**Grafo  
Conexo**



**Grafo  
No Conexa**



- Un camino de longitud  $n$  es una sucesión de vértices,  $v_i$ , y aristas,  $a_j$ , de la siguiente forma:  $v_0, a_1, v_1, \dots, v_{n-1}, a_n, v_n$
- Se dice que un camino es **cerrado** si  $v_n = v_0$ , es decir, el vértice inicial y el final son el mismo
- Se llama **trayectoria** a un camino simple en el que todos sus vértices (salvo probablemente el inicial y el final) son distintos
- se denomina **circuito** a una trayectoria cerrada con al menos una arista
- Se llama **camino euleriano** a un camino simple que contiene todas las aristas del grafo
- se denomina **circuito euleriano** a un camino euleriano cerrado
- Se dice que un **grafo es euleriano** si contiene un circuito euleriano
- Las aristas, además de ser dirigidas, pueden tener un valor o un peso que aporta cierta información acerca de las relaciones. Son los llamados **grafos ponderados** (como el del ejercicio a resolver)

Existen numerosos problemas que se pueden formular en términos de grafos. Ejemplos de ello son la planificación de las tareas que completan un proyecto, encontrar las rutas de menor longitud entre dos puntos geográficos, calcular el camino más rápido en un transporte, determinar el flujo máximo que puede llegar desde una fuente a, por ejemplo, una urbanización; entre otros. La resolución de estos problemas requiere examinar todos los nodos o todas las aristas del grafo que representa al problema; sin embargo, existen ocasiones en que la estructura del problema es tal que sólo se necesitan visitar algunos de los nodos o bien algunas de las aristas. Los algoritmos imponen implícitamente un orden en estos recorridos: visitar el nodo más próximo o las aristas más cortas, y así sucesivamente; otros algoritmos no requieren ningún orden concreto en el recorrido.

### Rutas mínimas

El algoritmo de rutas más cortas es uno de los análisis más importantes de los algoritmos de grafos. Este se encarga de detectar dentro de un grafo cuál es la ruta más eficiente o el recorrido de menor distancia entre un par de vértices que conforman un grafo. Dentro de esta categoría destaca el conocido algoritmo de **Dijkstra** (fue desarrollado por el científico del mismo nombre en 1956 mientras se proponía encontrar elementos innovadores para las computadoras ARMAC).

También se pueden mencionar los algoritmos de **Floyd** y **Warshall**. Los tres algoritmos utilizan una matriz de adyacencia ponderada o etiquetada: que es la misma matriz de adyacencia utilizada para representar grafos, pero con la diferencia que en lugar de colocar un número "1" cuando dos vértices son adyacentes, se coloca el peso o ponderación asignado a la arista que los une.

Por ejemplo, para el ejercicio a resolver, la siguiente sería la matriz de adyacencia:



Ciudad	Medellín	Puerto Berrio	Puerto Triunfo	Manizales	Honda	Bogotá	Girardot	Ibagué
Medellín	0	186	190	200	0	0	0	0
Puerto Berrio	186	0	129	0	0	0	0	0
Puerto Triunfo	190	129	0	0	102	0	0	0
Manizales	200	0	0	0	141	0	0	174
Honda	0	0	102	141	0	169	138	141
Bogotá	0	0	0	0	169	0	133	0
Girardot	0	0	0	0	138	133	0	66
Ibagué	0	0	0	174	141	0	66	0

A continuación, se presentan los pasos del algoritmo:

- 1) Seleccionar vértice de partida, es decir un origen
- 2) Determinar los caminos especiales desde el nodo de partida. Camino especial es aquel que solo puede trazarse a través de los nodos o vértices ya marcados
- 3) Para cada nodo no marcado, se debe determinar si es mejor usar el camino especial antes calculado o si es mejor usar el nuevo camino especial que resulte al marcar este nuevo nodo
- 4) Para seleccionar un nuevo nodo no marcado como referencia, deberá tomarse aquel cuyo camino especial para llegar a él es el mínimo, por ejemplo, si anteriormente se marcó el nodo o vértice 2, el cual tiene dos nodos adyacentes 3 y 4 cuyo peso en la arista corresponde a 10 y 5 respectivamente, se tomará como nuevo nodo de partida el 4, ya que el peso de la arista o camino es menor
- 5) Cada camino mínimo corresponde a la suma de los pesos de las aristas que forman el camino para ir del nodo principal al resto de nodos, pasando únicamente por caminos especiales, es decir nodos marcados

La implementación de este algoritmo usando cola de prioridades tendrá el siguiente pseudocódigo:

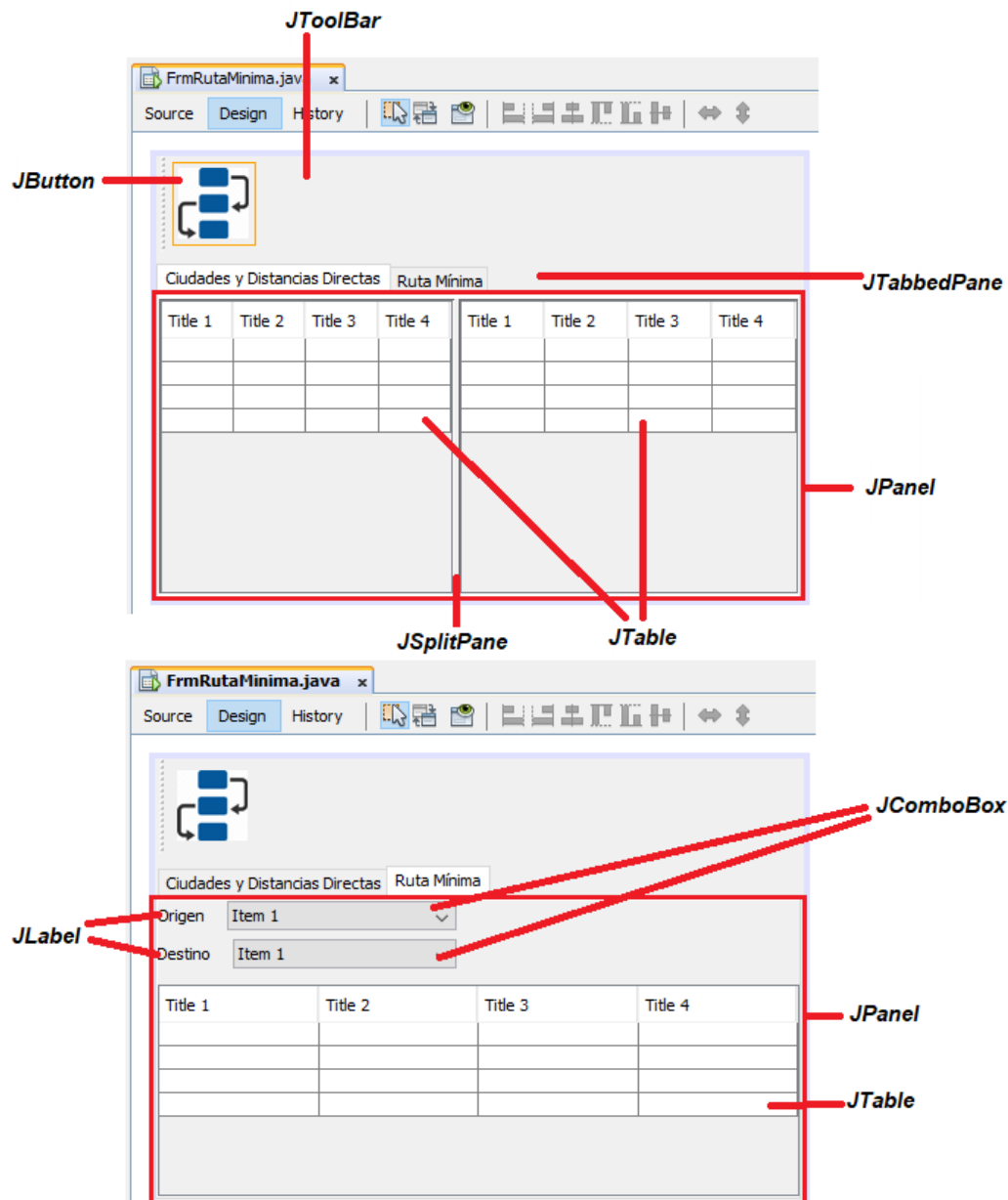
```

dijkstra (Grafo G, Nodo s)
  para u ∈ V[G] hacer
    distancia[u] = INFINITO
    padre[u] = NULO
  distancia[s] = 0
  encolar (cola, (s, distancia[s]))
  mientras no vacía(cola) hacer
    u = extraerMinimo(cola)
    para todos v ∈ adyacencia[u] hacer
      si distancia[v] > distancia[u] + peso (u, v) entonces
        distancia[v] = distancia[u] + peso (u, v)
        padre[v] = u
        encolar (cola, (v, distancia[v]))

```

## Programa

Para la implementación del aplicativo en *Java* se debe comenzar con el diseño de un formulario como el siguiente (Utilizando el *IDE Netbeans*):



Este formulario corresponde a la clase *RutaMinima* del anterior diagrama de clases. Al ser un formulario, hereda de la clase *JFrame*. La siguiente tabla relaciona los objetos a añadir con las propiedades cuyos valores deben ser cambiados:

Tipo Control	Nombre	Otras Propiedades
<b>JToolBar</b>	<i>jToolBar1</i>	
<b>JButton</b>	<i>btnProcesar</i>	icon = "Procesar.png" toolTiptext = "Hallar Ruta Mínima"
<b>JTabbedPane</b>	<i>tp</i>	
<b>JPanel</b>	<i>jPanel1</i>	text = "Ciudades y Distancias"

	<i>jPanel2</i>	toolTiptext = "Eliminar" icon= "Eliminar.gif"
<b>JSplitPane</b>	<i>jSplitPane1</i>	
<b>JTable</b>	<i>tblCiudades</i>	
	<i>tblDistancias</i>	
	<i>tblResultado</i>	
<b>JComboBox</b>	<i>cmbOrigen</i>	
	<i>cmbDestino</i>	
<b>JLabel</b>	<i>jLabel1</i>	text = "Origen"
	<i>jLabel2</i>	text= "Destino"

De acuerdo al modelo de clases planteado, se debe editar la clase *Nodo* la cual tendrá la funcionalidad asociada a la estructura base del grafo, ósea los vértices. Esta se compondrá de las siguientes propiedades:

Nodo
nombre: String valor: double padre: Nodo activo: boolean vecinos: Nodo[] aristas: Arista[]
agregarVecino(): void esPadre(): boolean esVecino(): boolean

- *nombre*: El nombre del vértice
- *valor*: almacena temporalmente el valor durante el proceso de hallar la ruta mínima
- *padre*: almacena temporalmente el nodo padre durante el proceso de hallar la ruta mínima
- *activo*: indica temporalmente si el nodo está activo durante el proceso de hallar la ruta mínima
- *vecinos*: almacena la lista de nodos adyacentes al nodo
- *aristas*: almacena las aristas del nodo

Y los siguientes métodos:

- *agregarVecino()*: añade a las listas *vecinos* y *aristas* un nodo y la respectiva arista
- *esPadre()*: indica si un nodo es padre del nodo actual durante el proceso de hallar la ruta mínima
- *esVecino()*: indica si un nodo hace parte de la lista de vecinos

El siguiente es el código completo de la clase en *Java*:

```
import java.util.ArrayList;
import java.util.List;

public class Nodo {

    private String nombre;

    private double valor; //almacenamiento temporal para procesos
    private Nodo padre; //almacenamiento temporal para procesos
    private boolean activo; //almacenamiento temporal para procesos

    private List<Nodo> vecinos;
    private List<Arista> aristas;

    public Nodo(String nombre) {
        this.nombre = nombre;
        activo = false;
        vecinos = new ArrayList<>();
    }
}
```

```
        aristas = new ArrayList<>();
    }

    public String obtenerNombre() {
        return nombre;
    }

    //asignar valor al nodo
    public void asignarValor(double valor) {
        this.valor = valor;
    }

    //Devuelve el valor del nodo
    public double obtenerValor() {
        return valor;
    }

    //asignar padre al nodo
    public void asignarPadre(Nodo n) {
        padre = n;
    }

    //Devuelve el padre del nodo
    public Nodo obtenerPadre() {
        return padre;
    }

    //hacer inactivo el nodo
    public void desactivar() {
        activo = false;
    }

    //Devuelve si el nodo esta activo
    public boolean estaActivo() {
        return activo;
    }

    public List<Nodo> obtenerVecinos(){
        return vecinos;
    }

    public List<Arista> obtenerAristas(){
        return aristas;
    }

    //Agrega un nodo vecino del actual
    public void agregarVecino(Nodo n, Arista a) {
        //asignar el nuevo vecino
        vecinos.add(n);
        aristas.add(a);
    }

    //Verificar si un nodo es padre del actual nodo
    public boolean esPadre(Nodo n) {
        if (padre.equals(n)) {
            return true;
        }
    }
}
```

```
    } else {  
        return false;  
    }  
}  
  
    //Verifica si un nodo es adyacente del actual  
public boolean esVecino(Nodo n) {  
    return vecinos.contains(n);  
}  
}
```

Ahora bien, la clase *Arista*, la cual implementa toda la funcionalidad relacionada con las aristas del grafo, se compondrá de los siguientes atributos:

Arista
nodo1: Nodo nodo2: Nodo valor: double

- *nodo1*: almacena el nodo origen de la arista
- *nodo2*: almacena el nodo destino de la arista
- *valor*: almacena el valor asignado a la arista (en este caso la distancia entre las ciudades)

El siguiente es el código completo de la clase en *Java*:

```
public class Arista {  
  
    private Nodo nodo1, nodo2;  
    private double valor;  
  
    public Arista(Nodo nodo1, Nodo nodo2, double valor) {  
        this.nodo1 = nodo1;  
        this.nodo2 = nodo2;  
        this.valor = valor;  
    }  
  
    //asignar valor a la arista  
    public void asignarValor(double valor) {  
        this.valor = valor;  
    }  
  
    //Devuelve el valor de la arista  
    public double obtenerValor() {  
        return valor;  
    }  
  
    //Devuelve el primer nodo de la arista  
    public Nodo obtenerNodo1() {  
        return nodo1;  
    }  
  
    //Devuelve el segundo nodo de la arista  
    public Nodo obtenerNodo2() {  
        return nodo2;  
    }  
}
```

Continuando con la clase *Grafo*, la cual implementa toda la funcionalidad relacionada con la estructura del grafo en sí, se compondrá de los siguientes atributos:

Grafo
<code>nodos: Nodo[]</code> <code>aristas: Arista[]</code>
<code>agregarNodo(): Nodo</code> <code>buscarNodo(): Nodo</code> <code>agregarArista(): void</code> <code>desdeArchivo(): void</code> <code>mostrarNodos(): void</code> <code>mostrarAristas(): void</code>

- *nodos*: almacena la lista de nodos del grafo
- *aristas*: almacena la lista de aristas del grafo

Y los siguientes métodos:

- *agregarNodo()*: añade a la lista *nodos* un nuevo vértice
  - *buscarNodo()*: devuelve el objeto *Nodo* correspondiente a un nombre determinado
  - *agregarArista()*: añade a la lista *aristas* una nueva arista
  - *desdeArchivo()*: permite crear el grafo a partir de la información almacenada en un archivo plano
- *mostrarNodos()*: permite mostrar la lista de vértices del grafo
  - *mostrarAristas()*: permite mostrar la lista de aristas del grafo

El siguiente es el código completo de la clase en *Java*:

```
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.table.DefaultTableModel;

public class Grafo {

    private List<Nodo> nodos;
    private List<Arista> aristas;

    public List<Nodo> obtenerNodos() {
        return nodos;
    }

    public List<Arista> obtenerAristas() {
        return aristas;
    }

    //agrega un nodo al grafo
    public Nodo agregarNodo(String nombre) {
        //instanciar el nuevo nodo
        Nodo n = new Nodo(nombre);
        nodos.add(n);
        return n;
    }

    //devuelve un nodo dado su nombre, si existe
    public Nodo buscarNodo(String nombre) {
        for (Nodo n : nodos) {
            if (n.obtenerNombre().equals(nombre)) {
                return n;
            }
        }
        return null;
    }
}
```

```
//agrega un borde al grafo
public void agregarArista(Nodo n1, Nodo n2, double valor) {
    //instanciar la nueva arista
    Arista a = new Arista(n1, n2, valor);
    //agregarla a la lista
    aristas.add(a);
    //actualiza los vecinos del primer nodo
    n1.agregarVecino(n2, a);
    //actualiza los vecinos del segundo nodo
    n2.agregarVecino(n1, a);
}

//Llena el grafo desde un archivo
public void desdeArchivo(String nombreArchivo) {
    //Limpiar el grafo
    nodos = new ArrayList<>();
    aristas = new ArrayList<>();
    //Abrir el archivo con los datos
    BufferedReader br = Archivo.abrirArchivo(nombreArchivo);
    //Se pudo leer algo?
    if (br != null) {
        try {
            //Leer la primer linea de texto
            String linea = br.readLine();
            //Mientras se hayan leído líneas
            while (linea != null) {
                //Partir la línea en los datos componentes
                String[] textos = linea.split(",");
                //verificar que haya el mínimo de datos
                if (textos.length >= 3) {
                    //verificar que no exista el primer nodo
                    Nodo n1 = buscarNodo(textos[0].trim());
                    if (n1 == null) {
                        n1 = agregarNodo(textos[0].trim());
                    }
                    //verificar que no exista el segundo nodo
                    Nodo n2 = buscarNodo(textos[1].trim());
                    if (n2 == null) {
                        n2 = agregarNodo(textos[1].trim());
                    }
                    //Convertir el valor numérico almacenado
                    double valor = 0;
                    try {
                        valor = Double.parseDouble(textos[2]);
                    } catch (Exception ex) {
                    }
                    //Agregar el borde
                    agregarArista(n1, n2, valor);
                }
                //Leer la siguiente línea
                linea = br.readLine();
            }
        } catch (IOException ex) {
        }
    }
}
```

```
}

//muestra la lista de nodos
public void mostrarNodos(JTable tbl) {
    //Verificar si hay nodos en el grafo
    if (nodos.size() > 0) {
        //Crear el modelo de datos de la tabla
        //Listar los nombres de los nodos
        String[][] datos = new String[nodos.size()][1];
        for (int i = 0; i < nodos.size(); i++) {
            datos[i][0] = nodos.get(i).obtenerNombre();
        }
        //definir las columnas
        DefaultTableModel dtm = new DefaultTableModel(datos,
            new String[]{"Nombre"});
        //Asignar el modelo a la tabla
        tbl.setModel(dtm);
    }
}

//muestra la lista de nodos
public void mostrarNodos(JComboBox cmb) {
    cmb.removeAllItems();
    for (int i = 0; i < nodos.size(); i++) {
        cmb.addItem(nodos.get(i).obtenerNombre());
    }
}

//muestra la lista de aristas
public void mostrarAristas(JTable tbl) {
    //Verificar si hay bordes en el grafo
    if (aristas != null) {
        //Crear el modelo de datos de la tabla
        //Listar los nombres de los nodos y el valor del valor del borde
        String[][] datos = new String[aristas.size()][3];
        for (int i = 0; i < aristas.size(); i++) {
            datos[i][0] = aristas.get(i).obtenerNodo1().obtenerNombre();
            datos[i][1] = aristas.get(i).obtenerNodo2().obtenerNombre();
            datos[i][2] = String.valueOf(aristas.get(i).obtenerValor());
        }
        //definir las columnas
        DefaultTableModel dtm = new DefaultTableModel(datos,
            new String[]{"Nodo 1", "Nodo 2", "Valor"});
        //Asignar el modelo a la tabla
        tbl.setModel(dtm);
    }
}
}
```

En este código es importante observar estos detalles:

- Los vértices del grafo son objetos de la clase *Nodo*
- Las aristas del grafo son objetos de la clase *Arista*



- El archivo plano es cargado en memoria en objetos de la clase *BufferedReader* el cual permite recorrer su contenido para luego ser utilizado para crear nodos de la lista ligada
- Debido a que la información del archivo plano corresponde a las aristas del grafo, los vértices generalmente se repiten, razón por la cual se debe buscar que cuando se agregue uno nuevo, éste no exista ya en la lista.

Distancias.txt	
Source	History
1	Medellín, Puerto Berrío, 186
2	Medellín, Puerto Triunfo, 190
3	Medellín, Manizales, 200
4	Manizales, Honda, 141
5	Puerto Berrío, Puerto Triunfo, 129
6	Puerto Triunfo, Honda, 102
7	Honda, Bogotá, 169
8	Honda, Girardot, 138
9	Girardot, Bogotá, 133
10	Girardot, Ibagué, 66
11	Ibagué, Honda, 141
12	Ibagué, Manizales, 174

- Las rejillas de datos para mostrar la información del grafo tales como vértices o aristas, se implementa mediante objetos de la clase *JTable*, la cual requiere estratégicamente pasar la información de la lista ligada a una matriz.
- Hay sobrecarga del método *mostrarNodos()* para permitir la posibilidad de listar en rejilla de datos y en lista desplegable

La clase *Archivo* incluye la funcionalidad para operar con archivos planos. El método que se implementa es el siguiente:

Archivo
abrirArchivo(): <i>BufferedReader</i>

- *abrirArchivo()*: Permite abrir el contenido de un archivo plano y dejarlo disponible en memoria para su lectura

El siguiente es el código completo de la clase en *Java*:

```
import java.io.*;

public class Archivo {

    public static BufferedReader Abrir(String nombreArchivo) {

        //verificar la existencia del archivo
        File f = new File(nombreArchivo);
        if (f.exists()) {
            try {
                FileReader fr = new FileReader(f);
```

```
        return new BufferedReader(fr);
    } catch (Exception ex) {
        return null;
    }
} else {
    return null;
}
}
```

En este código es importante observar estos detalles:

- La clase *File* permite verificar la existencia de un archivo mediante el método *exists()*. Esto sirve para poder abrir un archivo sin generar errores de no existencia
- Un archivo plano se lee mediante objetos de la clase *FileReader*. El resultado de la lectura se almacena en objetos de la clase *BufferedReader* quien almacena el contenido del archivo en memoria

Habiendo implementado las clases para definir el grafo, se procederá a implementar las clases para procesarlo y obtener la solución al problema.

Se comenzará desarrollando la clase *Resultado* que implementa la funcionalidad para almacenar el resultado esperado (la ruta mínima)

La descripción de esta clase es la siguiente:

Resultado
<b>nodos:</b> <i>Nodo</i> []
<b>agregar():</b> void <b>mostrar():</b> void

Atributos:

- *nodos*: Lista con los vértices que componen la ruta mínima

Métodos:

- *agregar()*: permite agregar un vértice a la solución
- *mostrar()*: permite desplegar la lista de vértices que componen la solución

El código en *Java* es el siguiente:

```
import java.util.*;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

public class Resultado {

    private List<Nodo> nodos;

    public Resultado() {
        nodos = new ArrayList<>();
    }

    //agrega un nodo al nodos
    public void agregar(Nodo n) {
        nodos.add(n);
    }

    //mostrar nodos del nodos
    public void mostrar(JTable tbl, boolean mostrarTotal) {
        //hay nodos en el nodos?
        if (nodos != null) {
```

```
//Crear el modelo de datos de la tabla
//Listar los nombres de los nodos y su peso
String[][] datos;
if (mostrarTotal) {
    datos = new String[nodos.size() + 1][2];
} else {
    datos = new String[nodos.size()][2];
}
double valorTotal = 0;
for (int i = 0; i < nodos.size(); i++) {
    datos[i][0] = nodos.get(i).obtenerNombre();
    datos[i][1] = String.valueOf(nodos.get(i).obtenerValor());
    valorTotal += nodos.get(i).obtenerValor();
}
if (mostrarTotal) {
    //Mostrar el valor total del nodos
    datos[nodos.size()][0] = "Valor Total";
    datos[nodos.size()][1] = String.valueOf(valorTotal);
}
//definir las columnas
DefaultTableModel dtm = new DefaultTableModel(datos,
    new String[]{"Nombre", "valor"});
//Asignar el modelo a la tabla
tbl.setModel(dtm);
}

public List<Nodo> obtenerNodos() {
    return nodos;
}
}
```

Luego se desarrolla la clase *AlgoritmosGrafos* que incluye la implementación del algoritmo de *Dijkstra*. La descripción de esta clase incluye:

AlgoritmosGrafos
dijkstra(): Resultado

Métodos:

- *dijkstra ()*: permite obtener la solución de la ruta mínima entre dos vértices de un grafo

Cuyo código en *Java* es el siguiente:

```
import java.util.Stack;

public class AlgoritmosGrafos {

    // Halla la ruta más corta desde el nodo inicial a todos los demás
    private static Resultado dijkstra(Grafo g, int inicio) {
        //declarar la lista resultante
        Resultado r = new Resultado();
        //Cola de nodos
        ColaPrioridad cola = new ColaPrioridad();
        //Agregar el nodo inicial a la cola
        Nodo n = g.obtenerNodos().get(inicio);
```

```
n.asignarPadre(null);
n.asignarValor(0);
cola.encolar(n, 0);
//Iterar hasta que la cola este vacia
while (!cola.vacia()) {
    //Desencolar nodo
    Nodo n = (Nodo) cola.desencolar();
    //agregarlo al resultado
    r.agregar(n);
    //Recorrer los nodos vecinos
    for (int j = 0; j < n.obtenerVecinos().size(); j++) {
        Nodo nv = n.obtenerVecinos().get(j);
        if (!r.obtenerNodos().contains(nv)) {
            double valor = n.obtenerValor() +
n.obtenerAristas().get(j).obtenerValor();
            // encolarlo si no esta en la cola
            if (!cola.contiene(nv)) {
                nv.asignarValor(valor);
                nv.asignarPadre(n);
                cola.encolar(nv, nv.obtenerValor());
            } else {
                if (nv.obtenerValor() > valor) {
                    nv.asignarValor(valor);
                    nv.asignarPadre(n);
                    cola.eliminar(nv);
                    cola.encolar(nv, valor);
                }
            }
        }
    }
}
return r;
}

// Halla la ruta más corta desde un nodo inicial a un nodo final
public static Resultado dijkstra(Grafo g, int inicio, int fin) {
    //aplicar el algoritmo de dijkstra al grafo desde el punto inicial
    Resultado r = dijkstra(g, inicio);
    //Obtener el nodo de destino
    Nodo nd = g.obtenerNodos().get(fin);
    //Verificar que este en el resultado
    if (r.obtenerNodos().contains(nd)) {
        // crea una pila para almacenar la ruta desde el nodo destino al
origen
        Stack pila = new Stack();
        while (nd != null) {
            pila.add(nd);
            nd = nd.obtenerPadre();
        }
        r = new Resultado();
        // recorre la pila para armar la ruta en el orden correcto
        while (!pila.isEmpty()) {
            r.agregar((Nodo) pila.pop());
        }
        return r;
    }
}
```

```

    } else {
        return null;
    }
}
}

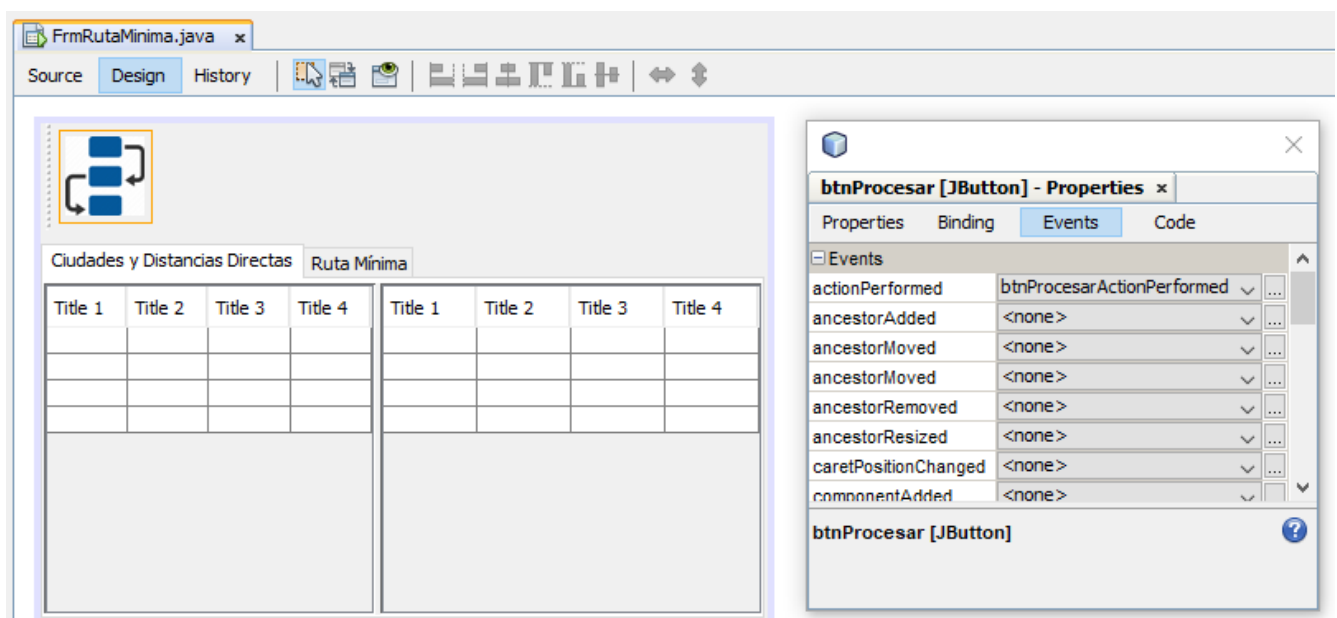
```

En este código es importante observar estos detalles:

- El algoritmo se divide en 2 subrutinas:
  - La primera permite que, dado un vértice inicial, hallar la ruta mínima que recorra todos los demás vértices, lo cual lo almacena en un objeto de la clase *Resultado*
  - La segunda toma el anterior resultado y busca el vértice de destino para devolverse hasta el vértice origen. Para ello utiliza una pila que recoge todos los nodos de interés para luego desapilarlos en el orden requerido en un nuevo objeto de la clase *Resultado*
- Los vértices que se van encolando se obtienen a partir de los vértices vecinos del vértice que ya ha sido agregado al resultado, pero se van descartando cuando aumentan el valor de la ruta mínima. Luego de recorrer todos los nodos vecinos, se desencola el que al final no quedó descartado, para agregarlo al resultado y seguir el proceso

Ahora bien, para terminar con la codificación, se debe programar el método *procesar()* de la clase *RutaMinima* y que corresponderá al evento del botón de comando agregado al formulario.

El cálculo de la ruta mínima se hará mediante un método que responde al evento ***actionPerformed*** del botón de comando ***btnProcesar***:



El código respectivo sería el siguiente:

```

private void btnProcesarActionPerformed(java.awt.event.ActionEvent evt) {
    Resultado r = AlgoritmosGrafos.dijkstra(g,

```

```
cmbOrigen.getSelectedIndex(),
cmbDestino.getSelectedIndex());
r.mostrar(tblResultado, false);
}
```

En este código, se llama a la implementación del algoritmo de *Dijkstra()* pasando como parámetros el grafo y los vértices elegidos como origen y destino. Luego se muestra en la rejilla de datos el resultado obtenido.

Por último, el código que corresponde el método constructor de la clase y que permite construir el grafo a partir de la información almacenada en el archivo plano, es el siguiente:

```
public class FrmRutaMinima extends JFrame {

    Grafo g=new Grafo ();

    public FrmRutaMinima() {
        initComponents();

        String nombreArchivo = System.getProperty("user.dir")
            + "/src/datos/Distancias.txt";
        g.desdeArchivo(nombreArchivo);
        g.mostrarNodos(tblCiudades);
        g.mostrarAristas(tblDistancias);

        g.mostrarNodos(cmbOrigen);
        g.mostrarNodos(cmbDestino);    }

    ...
}
```

2. Implementar en el anterior ejercicio una función que determine el grado de todos los vértices del grafo.

La respuesta para los datos mostrados sería una tabla como la siguiente:

Vértice	Grado
Medellín	3
Puerto Berrio	2
Puerto Triunfo	3
Manizales	3
Honda	5
Bogotá	2
Girardot	3
Ibagué	3

3. Implementar en el anterior ejercicio una función que determine si el grafo es conexo o no
4. Supongamos que tenemos un sistema de una facultad en el que cada alumno puede pedir hasta 10 libros de la biblioteca. La biblioteca tiene 3 copias de cada libro. Cada alumno desea pedir libros diferentes. Implementar un algoritmo que nos permita obtener la forma de asignar libros a alumnos de tal forma que la cantidad de préstamos sea máxima