UNIVERSIDAD DE SAN CARLOS DE GUATEMALA.

CENTRO UNIVERSITARIO DE OCCIDENTE.

DIVISIÓN DE CIENCIAS DE LA INGENIERÍA.

INGENIERÍA EN CIENCIAS Y SISTEMAS.

LABORATORIO DE ESTRUCTURA DE DATOS.

PROYECTO FINAL.

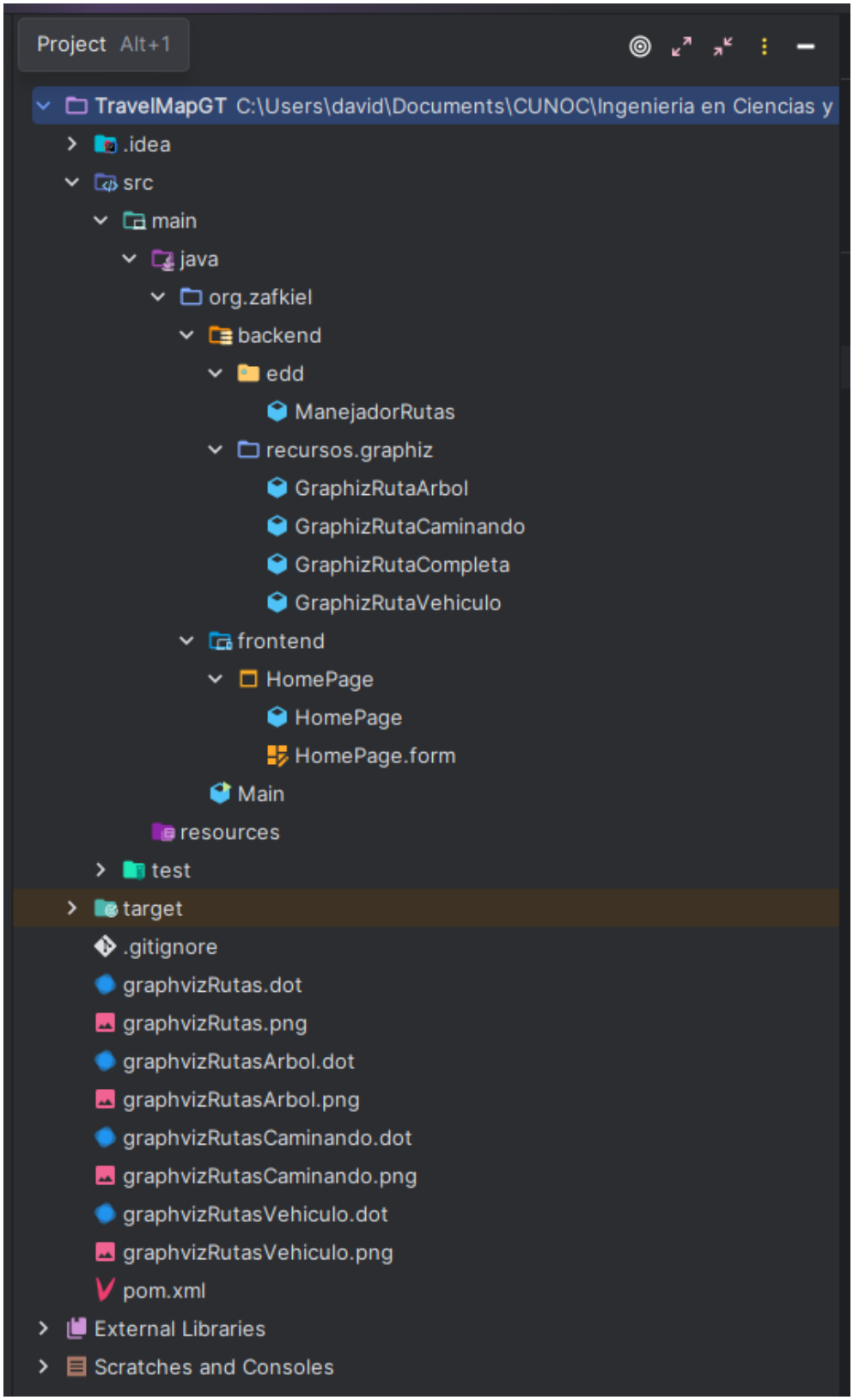
TRAVELMAP GT

MANUAL TÉCNICO.

ESTUARDO DAVID BARRENO NIMATUJ.

CARNÉ: 201830233.

ESTRUCTURA DEL PROYECTO.



El proyecto cuenta con las diferentes clases y métodos dentro de la carpeta, asi mismo todos los archivos .dot y .png usados en el mismo se generan de manera dinámica conforme avance el programa.

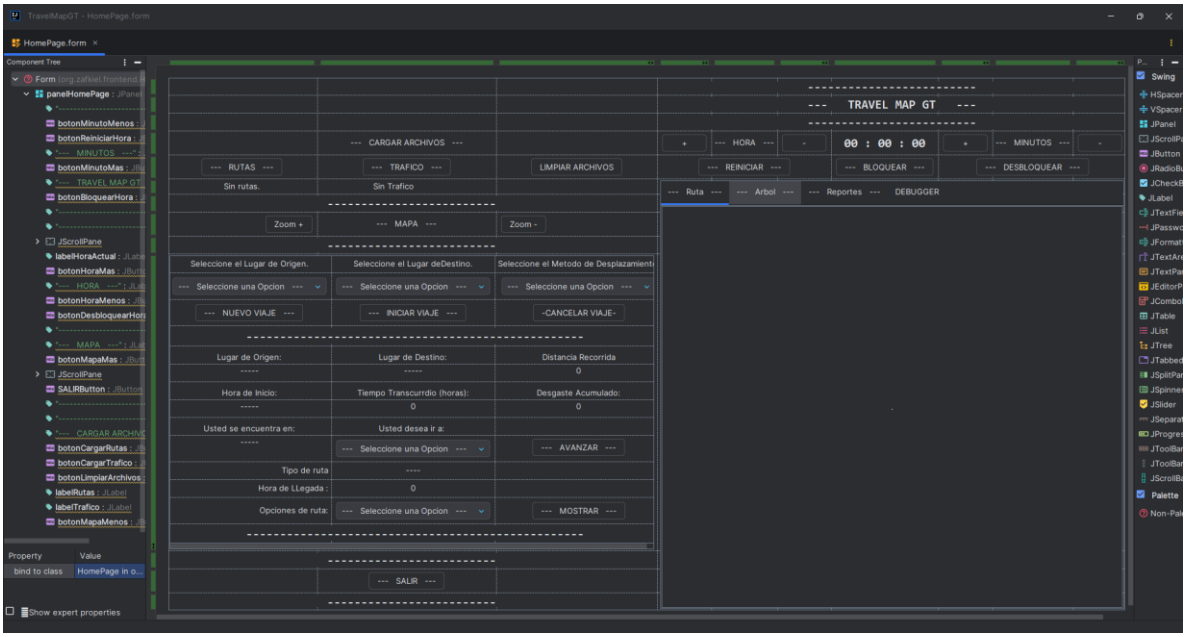
METODO MAIN.

```java
public class Main {
    public static void main(String[] args) {
        HomePage homePage = new HomePage();
        homePage.setContentPane(homePage.panelHomePage);
        homePage.setExtendedState(JFrame.MAXIMIZED_BOTH);
        homePage.setUndecorated(true);
        homePage.setVisible(true);

    }
}
```

Este método simplemente llama al panel HomePage que es la interfaz gráfica base del proyecto.

METODO INTERFAZ GRAFICA.



La interfaz esta echa en un solo panel llamado HomePage, y tiene sub paneles y todos sus detalles diseñados en una sola clase.

METODO CARGA DE ARCHIVOS – RUTAS.

Este código en Java maneja la acción de cargar rutas desde un archivo CSV:

1. Se crea un botón botonCargarRutas que espera la acción del usuario.

2. Cuando se presiona el botón, se abre un explorador de archivos para seleccionar un archivo CSV.

3. Se filtran solo los archivos CSV para la selección.

4. Si se elige un archivo CSV válido, se lee su contenido línea por línea.

5. Cada línea se procesa y se almacena en una estructura de datos llamada `datos`.

6. Se verifica que cada línea tenga 7 valores y se limpian los espacios en blanco.

7. Se manejan casos donde la combinación de ciertos valores ya existe o es única.

8. Se actualiza una etiqueta labelRutas con el nombre del archivo seleccionado.

9. Si hay algún error en el archivo seleccionado, se muestra un mensaje de error y se detiene el proceso.

10. Se habilitan botones y se actualiza una tabla con los datos cargados.

11. Se generan archivos de imagen y se preparan para mostrar mapas.

12. Se procesan y se llenan los datos de origen y destino en un menú desplegable.

13. Se muestra una imagen general relacionada con las rutas cargadas.

Para hacer todo esto se usa un meto Try Catch, en donde se implementa la lógica de los filtros , una vez terminado se manda a cargar los 3 mapas interactivos a partir de un archivo DOT y se muestran en el programa.

```
<origen>|<destino>|<tiempo_vehiculo>|<tiempo_pie>|<consumo_gas>|<desgaste_persona>|<distancia>
```

Ejemplo:

```
AltaVerapaz|SantaMaríadeJesús|24|51|24|456|85
Petén|CiudadVieja|24|42|24|453|24
Quetzaltenango|Retalhuleu|25|50|56|54|100
Retalhuleu|Quetzaltenango|36|45|60|80|100
SanMarcos|Quetzaltenango|30|35|62|15|250
Suchitepequez|Guatemala|70|250|60|2500|70
SanMarcos|SanBartoloméMilpasAltas|24|425|62|63|55
Huehuetenango|SanLucasSacatepéquez|24|123|32|47|62
Quiché|SantaLucíaMilpasAltas|24|62|42|46|32
```

METODO CARGA DE ARCHIVOS – TRAFICO.

Este código en Java maneja la acción de cargar rutas desde un archivo CSV:

1. Se crea un botón botonCargarRutas que espera la acción del usuario.

2. Cuando se presiona el botón, se abre un explorador de archivos para seleccionar un archivo CSV.

3. Se filtran solo los archivos CSV para la selección.

4. Si se elige un archivo CSV válido, se lee su contenido línea por línea.

5. Cada línea se procesa y se almacena en una estructura de datos llamada `datos`.

6. Se verifica que cada línea tenga 7 valores y se limpian los espacios en blanco.

7. Se manejan casos donde la combinación de ciertos valores ya existe o es única.

8. Se actualiza una etiqueta labelTrafico con el nombre del archivo seleccionado.

9. Si hay algún error en el archivo seleccionado, se muestra un mensaje de error y se detiene el proceso.

Para hacer todo esto se usa un meto Try Catch, en donde se implementa la lógica de los filtros , una vez terminado se manda a cargar los 3 mapas interactivos a partir de un archivo DOT y se muestran en el programa.

<origen>|<destino>|<hora_inicio>|<hora_finaliza>|<probabilidad_trafico>

Ejemplo:

```
Quetzaltenango|Retalhuleu|12|14|40
Retalhuleu|Quetzaltenango|5|8|45
Retalhuleu|Quetzaltenango|15|18|80
Suchitepéquez|Retalhuleu|13|15|40
Quetzaltenango|Salola|12|14|40
Retalhuleu|Suchitepequez|16|21|63
```

TABLA DEBBUGER.

```
//--- Iniciamos los parametros ---
DefaultTableModel modelo1 = new DefaultTableModel();
modelo1.addColumn( columnName: " Origen ");
modelo1.addColumn( columnName: " Destino ");
modelo1.addColumn( columnName: " Tiempo_Vehiculo ");
modelo1.addColumn( columnName: " Tiempo_Pie ");
modelo1.addColumn( columnName: " Consumo_Gas ");
modelo1.addColumn( columnName: " Desgaste_Persona ");
modelo1.addColumn( columnName: " Distancia ");
modelo1.addColumn( columnName: " Trafico_H_Inicio ");
modelo1.addColumn( columnName: " Trafico_H_Fin ");
modelo1.addColumn( columnName: " Trafico_Prob ");
tablaRutasDebugger.setModel(modelo1);
```

Esta se utiliza principalmente solo para verificar que los datos fueron ingresados de manera correcta, ya que se basan en un arry list “datos”.

```
for (String[] fila : datos) {
    modeloTabla.addRow(fila);
}
tablaRutasDebugger.setModel(modeloTabla);
```

CARGA DE DATOS.

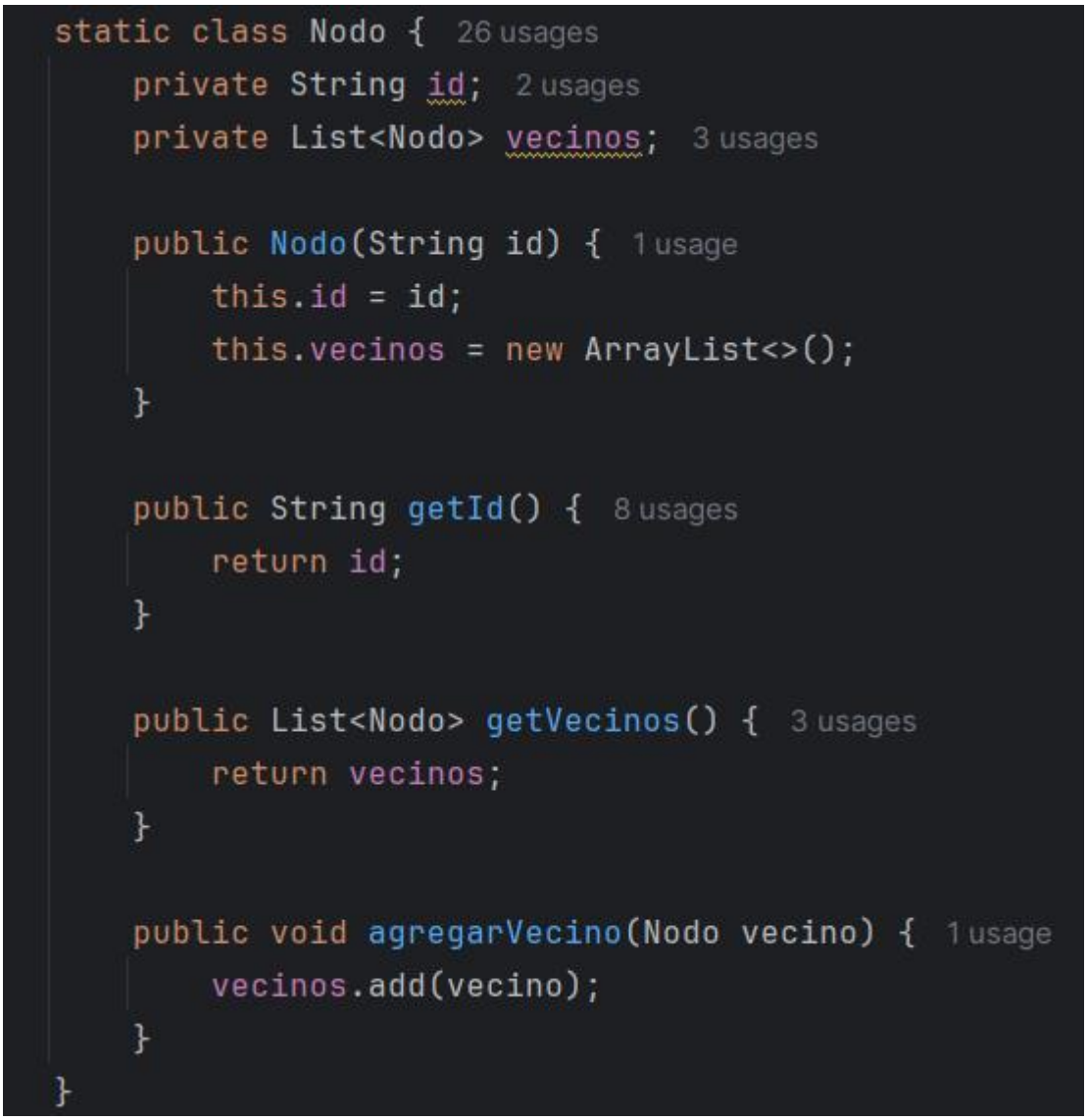
```
if(valores.length ==7 ){
    for (int i = 0; i < valores.length; i++) {
        valores[i] = valores[i].trim();

    }
    String combinacion = valores[1] + "|" + valores[2];
    if (combinacionesUnicas.contains(combinacion)) {
        for (int i = 0; i < 3; i++) {
            valores = Arrays.copyOf(valores, newLength: valores.length + 1);
            valores[valores.length - 1] = "0";
        }
        for (int i = 0; i < datos.size(); i++) {
            String[] fila = datos.get(i);
            if (fila[1].equals(valores[1]) && fila[2].equals(valores[2])) {
                datos.set(i, valores);
                break;
            }
                                                    }
    }else {
        // La combinación no existe, agregarla al conjunto de combinaciones únicas
        combinacionesUnicas.add(combinacion);
        // Agregar los valores al array datos
        for (int i = 0; i < 3; i++) {
            valores = Arrays.copyOf(valores, newLength: valores.length + 1);
            valores[valores.length - 1] = "0";
        }
        datos.add(valores);
    }
```

Una vez cargado el archivo, este se lee línea por línea hasta llenar la tabla datos, luego se genera un espacio de 3 extras donde se incluyen 0, ya que este es el lugar de los parámetros de tráfico.

Una vez cargado el archivo de tráfico, este se compara con el arry list datos y únicamente sustituye o adjunta los datos faltantes en donde la posición 0 y la posición 1 de trafico sean las mismas en rutas. Así nace nuestra base de datos.

ESTRUCTURAS DE NODOS.



```
static class Nodo {  26 usages
    private String id;  2 usages
    private List<Nodo> vecinos;  3 usages

    public Nodo(String id) {  1 usage
        this.id = id;
        this.vecinos = new ArrayList<>();
    }

    public String getId() {  8 usages
        return id;
    }

    public List<Nodo> getVecinos() {  3 usages
        return vecinos;
    }

    public void agregarVecino(Nodo vecino) {  1 usage
        vecinos.add(vecino);
    }
}
```

```java
private static Nodo buscarOCrearNodo(List<Nodo> nodos, String id) {
    for (Nodo nodo : nodos) {
        if (nodo.getId().equals(id)) {
            return nodo;
        }
    }
    Nodo nuevoNodo = new Nodo(id);
    nodos.add(nuevoNodo);
    return nuevoNodo;
}
```

1. `static class Nodo`: Define una clase estática llamada `Nodo`, que contiene la información y métodos relacionados con un nodo en un grafo.

- **`private String id`:** Representa el identificador único del nodo.

- **`private List<Nodo> vecinos`:** Es una lista que almacena los nodos vecinos conectados al nodo actual.

2. `public Nodo(String id)`: Constructor de la clase `Nodo` que inicializa un nodo con un identificador dado. Además, inicializa la lista de vecinos como una lista vacía.

3. `public String getId()`: Método que devuelve el identificador del nodo.

4. `public List<Nodo> getVecinos()`: Método que devuelve la lista de vecinos del nodo.

5. `public void agregarVecino(Nodo vecino)`: Método que agrega un nodo vecino a la lista de vecinos del nodo actual.

6. `private static Nodo buscarOCrearNodo(List<Nodo> nodos, String id)`: Este método estático recibe una lista de nodos y un identificador. Busca en la lista de nodos un nodo con el mismo identificador. Si lo encuentra, devuelve ese nodo. Si no lo encuentra, crea un nuevo nodo con el identificador dado, lo agrega a la lista de nodos y luego lo devuelve.

```
//NOODOSSSSSSSSSSSSSS
for (String[] dato : datos) {
    Nodo origen = buscarOCrearNodo(nodos, dato[0]);
    Nodo destino = buscarOCrearNodo(nodos, dato[1]);
    origen.agregarVecino(destino);
}
/*for (String[] dato : datos2) {
    Nodo origen = buscarOCrearNodo(nodos, dato[1]);
    Nodo destino = buscarOCrearNodo(nodos, dato[0]);
    origen.agregarVecino(destino);
}*/
```

Aca tenemos que aclara 2 cosas muy importantes , ya que nuestra clase nodos nos va a servir para crear todas las rutas posibles entre nodos, etc.

Y al momento de moverse entre nodos hay 2 formas.

1.Vehiculos: Esto quiere decir que el nodo solo tendrá 1 dirección.

2.Caminando: Esto quiere decir que el nodo tendrá dirección en ambos sentidos.

Para el modo de Vehículos, la carga de archivos se respeta ya que según nuestras condiciones en datos posición 0 es un origen y datos posición 1 es el vecino, cuya dirección es hacia el, y asi se repite con todos los datos.

Para el modo de Caminar, la carga de datos se hace doble, me explico, al momento de cargar la primera vez dejamos las rutas de vehículos, pero si cargamos por segunda vez únicamente invirtiendo las posiciones 0 y 1 a 1 y 0, esto genera los caminos faltantes entre los nodos.

Es por eso que se deja comentado para activar o desactivar cualquiera de los dos modos.

ESTRUCTURAS DE ARBOLES.

```
class BTreeNode {  12 usages
    int[] keys;  16 usages
    int t; // grado mínimo del árbol  1 usage
    BTreeNode[] children;  12 usages
    int n; // Número actual de claves  14 usages
    boolean leaf; // Indica si el nodo es una hoja  5 usages

    BTreeNode(int t, boolean leaf) {  3 usages
        this.t = t;
        this.leaf = leaf;
        this.keys = new int[2 * t - 1];
        this.children = new BTreeNode[2 * t];
        this.n = 0;
    }
}
```

Clase `BTreeNode`:

- Esta clase representa un nodo en un árbol B.

- `keys`: Es un arreglo que contiene las claves almacenadas en el nodo.

- `t`: Representa el grado mínimo del árbol.

- `children`: Es un arreglo que contiene los hijos del nodo.

- `n`: Es el número actual de claves en el nodo.

- `leaf`: Indica si el nodo es una hoja (true) o no (false).

```
class BTree {  2 usages
    private BTreeNode root;  10 usages
    private int t; // grado mínimo del árbol  13 usages

    BTree(int t) {  1 usage
        this.t = t;
        this.root = null;
    }
```

Clase `BTree`:

- Esta clase implementa un árbol B.
- `root`: Es el nodo raíz del árbol.
- `t`: Representa el grado mínimo del árbol.

```
void insert(int key) {  1 usage
    // Si el árbol está vacío
    if (root == null) {
        root = new BTreeNode(t,  leaf: true);
        root.keys[0] = key;
        root.n = 1;
    } else {
        // Si la raíz está llena, se divide
        if (root.n == 2 * t - 1) {
            BTreeNode newRoot = new BTreeNode(t,  leaf: false);
            newRoot.children[0] = root;
            splitChild(newRoot,  i: 0);
            int i = 0;
            if (newRoot.keys[0] < key)
                i++;
            insertNonFull(newRoot.children[i], key);
            root = newRoot;
        } else {
            insertNonFull(root, key);
        }
    }
```

Método `insert(int key)`:

- Inserta una nueva clave en el árbol.

- Si el árbol está vacío, crea un nuevo nodo raíz y agrega la clave.

- Si la raíz está llena, divide el nodo raíz y crea un nuevo nodo raíz.

- Llama a `insertNonFull()` para insertar la clave en un nodo no lleno.

```java
// Función auxiliar para insertar cuando el nodo no está lleno
private void insertNonFull(BTreeNode node, int key) {  3 usages
    int i = node.n - 1;
    if (node.leaf) {
        while (i >= 0 && node.keys[i] > key) {
            node.keys[i + 1] = node.keys[i];
            i--;
        }
        node.keys[i + 1] = key;
        node.n++;
    } else {
        while (i >= 0 && node.keys[i] > key)
            i--;
        i++;
        if (node.children[i].n == 2 * t - 1) {
            splitChild(node, i);
            if (node.keys[i] < key)
                i++;
        }
        insertNonFull(node.children[i], key);
    }
}
```

Método `insertNonFull(BTreeNode node, int key)`:

- Inserta una clave en un nodo que no está lleno.

- Si el nodo es una hoja, inserta la clave en orden.

- Si el nodo no es una hoja, encuentra el hijo adecuado y lo llama recursivamente.

- Si el hijo está lleno, divide el hijo.

```java
// Función para dividir el hijo del nodo x
private void splitChild(BTreeNode x, int i) {  2 usages
    BTreeNode y = x.children[i];
    BTreeNode z = new BTreeNode(t, y.leaf);
    z.n = t - 1;
    for (int j = 0; j < t - 1; j++)
        z.keys[j] = y.keys[j + t];
    if (!y.leaf) {
        for (int j = 0; j < t; j++)
            z.children[j] = y.children[j + t];
    }
    y.n = t - 1;
    for (int j = x.n; j >= i + 1; j--)
        x.children[j + 1] = x.children[j];
    x.children[i + 1] = z;
    for (int j = x.n - 1; j >= i; j--)
        x.keys[j + 1] = x.keys[j];
    x.keys[i] = y.keys[t - 1];
    x.n++;
}
```

Método `splitChild(BTreeNode x, int i)`:

- Divide el hijo del nodo `x` en dos hijos.
- Crea un nuevo nodo `z` y mueve las claves correspondientes.
- Actualiza los hijos y las claves del nodo `x`.

```java
// Función para recorrer el árbol y generar el formato de digraph
String generateDigraphTree() {  1 usage
    StringBuilder sb = new StringBuilder();
    sb.append("digraph Tree {\n");
    sb.append("    node [shape=record];\n");

    int counter = 1;
    sb.append(generateDigraphNode(root, counter));

    sb.append("}");
    return sb.toString();
}
```

Método `generateDigraphTree()`:

- Genera una representación en formato `digraph` del árbol.

- Utiliza un StringBuilder para construir el formato.

- Llama a `generateDigraphNode()` para generar la representación de cada nodo.

```java
private String generateDigraphNode(BTreeNode node, int counter) {  2 usages
    StringBuilder sb = new StringBuilder();
    sb.append("    ").append(counter).append("[label=\"");
    for (int i = 0; i < node.n; i++) {
        sb.append("<C").append(i).append(">").append(node.keys[i]);
        if (i < node.n - 1)
            sb.append("|");
    }
    sb.append("\"]\n");

    if (!node.leaf) {
        int childCounter = counter + 1; // Contador para los hijos
        for (int i = 0; i <= node.n; i++) {
            sb.append("    ").append(counter).append(":C").append(i).append(" -> ").append(childCounter).append("\n");
            sb.append(generateDigraphNode(node.children[i], childCounter));
            childCounter++; // Incrementa el contador para los hijos
        }
    }

    return sb.toString();
}
```

Método `generateDigraphNode(BTreeNode node, int counter)`:

- Genera la representación de un nodo en el formato `digraph`.

- Utiliza un contador para asignar un identificador único a cada nodo.

- Recorre los hijos del nodo y llama recursivamente a este método.

```
int[] arr = {5,5,5,5,5,5, 8, 1, 3, 9, 7, 4, 2, 6, 10,11,12,13,14,15,16,17};
Arrays.sort(arr); // Ordena el arreglo de manera ascendente

BTree bTree = new BTree( t: 5); // Crear un árbol B con grado 5

// Insertar los elementos ordenados en el árbol B
for (int key : arr) {
    bTree.insert(key);
}
rutaArbolB = bTree.generateDigraphTree();
GraphizRutaArbol graphizRutaArbol = new GraphizRutaArbol();
graphizRutaArbol.generarArchivoDOT();
graphizRutaArbol.generarImagenDesdeDOT();
iconoArbol = new ImageIcon( filename: "graphvizRutasArbol.png");
labelMostrarRutaArbolImagen.setIcon(iconoArbol);
```

Con todos los métodos del Arbol B, podemos recibir un arreglo de ints, en donde cada valor numerico representa el parámetro buscado en cada posición siguiente del nodo.

Una vez recibido se manda a generar un archivo en estructura DOT, en la estructura de nuestro proyecto para luego graficarlo.

```
digraph Tree {
    node [shape=record];
    1[label="<C0>5|<C1>6|<C2>11"]
    1:C0 -> 2
    2[label="<C0>1|<C1>2|<C2>3|<C3>4|<C4>5"]
    1:C1 -> 3
    3[label="<C0>5|<C1>5|<C2>5|<C3>5"]
    1:C2 -> 4
    4[label="<C0>7|<C1>8|<C2>9|<C3>10"]
    1:C3 -> 5
    5[label="<C0>12|<C1>13|<C2>14|<C3>15|<C4>16|<C5>17"]
}
```

MENEJADORES DE ARCHIVOS DOT. – CAMINOS.

```
public class GraphizRutaCompleta {  3 usages

    ArrayList<String[]> datos2 = new ArrayList<>();  2 usages

    public void  procesarArrayList(ArrayList<String[]> datos) {  1 usage

                datos2 = new ArrayList<>(datos);
    }

    public void generarArchivoDOT() {  1 usage
        String ciudades = "";


        for (String[] fila : datos2) {
            if (fila.length >= 6) {
                    ciudades +="\t"+ fila[0]+" -- "+ fila[1]+" [label = \""+ fila[6]+"KM\"];\n"  ;
            }
        }
```

Primero traemos nuestra base de datos “datos”, luego la pasamos a otro array para no alterar la base de datos original.

Sacamos los parámetros y los usamos a nuestro antojo.

```
String contenidoDOT = "graph rutas_completas {\n" +
        "\tfontname=\"Helvetica,Arial,sans-serif\"\n" +
        "\tnode [fontname=\"Helvetica,Arial,sans-serif\"]\n" +
        "\tedge [fontname=\"Helvetica,Arial,sans-serif\"]\n" +
        "\trankdir=LR;\n" +
        "\tnode [shape = doublecircle, style=filled, color=springgreen];\n" +
        "\tnode [shape = circle, color=lightskyblue];\n" +
        ciudades +
        "}";
```

Creamos una estructura contenidoDOT para luego crear el archivo DOT y asi graficarlo posteriormente.

```
    try {
        // Escribimos el contenido en un archivo DOT
        FileWriter writer = new FileWriter( fileName: "graphvizRutas.dot");
        writer.write(contenidoDOT);
        writer.close();
        System.out.println("Archivo DOT graphvizRutas.dot generado correctamente.");
    } catch (IOException e) {
        System.err.println("Error al escribir el archivo DOT: " + e.getMessage());
    }
}
```

```java
public void generarImagenDesdeDOT() {  1 usage
    try {
        // Ejecutamos el comando de Graphviz para generar la imagen desde el archivo DOT
        ProcessBuilder processBuilder = new ProcessBuilder( ...command: "dot", "-Tpng", "graphvizRutas.dot", "-o", "graphvizRutas.png");
        Process process = processBuilder.start();
        int exitCode = process.waitFor();
        if (exitCode == 0) {
            System.out.println("Imagen graphvizRutas.png generada correctamente.");
        } else {
            System.err.println("Error al generar la imagen. Código de salida: " + exitCode);
        }
    } catch (IOException | InterruptedException e) {
        System.err.println("Error al generar la imagen: " + e.getMessage());
    }
}
```

En ensencia es lo mismo para el archivo rutaCompleta y rutaCompletaVehiculo, la diferencia con el archivo rutaCompletaCaminando es que definirle en lugar de

```
degraph
```

se usa

```
graph
```

Esto por la cuestión de las señales.

MENEJADORES DE ARCHIVOS DOT. – ARBOLES.

El archivo DOT para graficar los arboles nace de una estructura de balanceo para arboles B, en donde recibe como parámetros los datos dependiendo el caso, el vecino del nodo actual, y su parámetro numero varia conforme a lo que estamos buscando balancear para encontrar la ruta mas corta.

```java
int[] arr = {5,5,5,5,5,5, 8, 1, 3, 9, 7, 4, 2, 6, 10,11,12,13,14,15,16,17};
Arrays.sort(arr); // Ordena el arreglo de manera ascendente

BTree bTree = new BTree( t: 5); // Crear un árbol B con grado 5
```

Como lo definimos en la clase manejadora de árboles B.

MANEJADOR DE PARAMETROS O RUTAS MAS CORTAS.

```
if(comboBoxOpcionesRuta.getSelectedItem().toString().equals("Mejor ruta en base a Distancia.")){
    Map<String, Map<String, Integer>> graph = new HashMap<>();
    for (String[] dato : datos) {
        String ciudad1 = dato[0];
        String ciudad2 = dato[1];
        int distancia = Integer.parseInt(dato[6]);
        graph.putIfAbsent(ciudad1, new HashMap<>());
        graph.putIfAbsent(ciudad2, new HashMap<>());
        graph.get(ciudad1).put(ciudad2, distancia);
        graph.get(ciudad2).put(ciudad1, distancia);
    }
    String inicio = labelRutaActual.getText();
    String fin = labelRutaFinal.getText();
    Map<String, Integer> distances = new HashMap<>();
    Map<String, String> parents = new HashMap<>();
    PriorityQueue<String> pq = new PriorityQueue<>(Comparator.comparingInt(distances::get));
    Set<String> visited = new HashSet<>();
    for (String city : graph.keySet()) {
        distances.put(city, Integer.MAX_VALUE);
    }
    distances.put(inicio, 0);
    pq.offer(inicio);
```

```
  while (!pq.isEmpty()) {
      String city = pq.poll();
      if (visited.contains(city)) continue;
      visited.add(city);

      Map<String, Integer> neighbors = graph.get(city);
      for (String neighbor : neighbors.keySet()) {
          int newDist = distances.get(city) + neighbors.get(neighbor);
          if (newDist < distances.get(neighbor)) {
              distances.put(neighbor, newDist);
              parents.put(neighbor, city);
              pq.offer(neighbor);
          }
      }
  }
```

```
List<String> path = new ArrayList<>();
String current = fin;
while (current != null) {
    path.add(current);
    current = parents.get(current);
}
Collections.reverse(path);
String reporte1 = "Distancia más corta de " + inicio + " a " + fin + ": " + distances.get(fin);
String reporte2 = "Camino recorrido: " + String.join( delimiter: " -> ", path);
JOptionPane.showMessageDialog( parentComponent: null,  message: reporte1+"\n"+reporte2,  title: "SUGERENCIA", JOptionPane.INFORMATION_MESSAGE);
```

1. Verificación de la selección: El código comienza verificando si la opción seleccionada en un JComboBox es "Mejor ruta en base a Distancia".

2. Creación del grafo: Se inicializa un HashMap llamado `graph` para representar el grafo. Se itera sobre los datos disponibles para agregar las ciudades y las distancias entre ellas al grafo.

3. Inicialización de variables: Se inicializan las variables necesarias para el algoritmo de Dijkstra, como las distancias, los padres, la cola de prioridad y el conjunto de ciudades visitadas.

4. Construcción del camino más corto: Se reconstruye el camino más corto desde la ciudad de destino hasta la ciudad de inicio utilizando los padres registrados durante la ejecución del algoritmo.

5. Presentación del resultado: Se crea un mensaje que incluye la distancia más corta entre las ciudades de inicio y destino, así como el camino recorrido. Este mensaje se muestra en un cuadro de diálogo emergente (`JOptionPane`).