Javier Isaías Coyoy Marín **202030556**

Manual Técnico

Cargar Archivos

Este método es utilizado cuando el usuario ingresa un archivo que contiene los destinos a cargar, primero verifica que el usuario haya seleccionado un archivo, de lo contrario le notificara al usuario, luego crea un objeto ImportarDestinosYTrafico en su método cargarDestinos() que será el encargado de organizar la información del archivo seleccionado y por ultimo mostrara todas las opciones dentro de los JComboBox.

```
private void cargarDestinosActionPerformed() {
      JFileChooser jFileChooser = new JFileChooser();


      int respuesta = jFileChooser.showOpenDialog(this);


      if (respuesta == JFileChooser.APPROVE_OPTION) {
         String path = jFileChooser.getSelectedFile().getPath();


         ImportarDestinosYTrafico importar = new ImportarDestinosYTrafico(lugares, informaciones);
         importar.cargarDestinos(path);
         seleccionar = true;
         mostrarCombos();
      } else {
         JOptionPane.showMessageDialog(this,"Debe seleccionar una Archivo a
Leer","Archivo",JOptionPane.ERROR_MESSAGE);
      }
   }
```

Importar Destinos Y Trafico

Cargar Destinos

Este método es invocado desde la página principal y es el encargado de ordenar y guardar toda la información proveniente del archivo de entrada, primero obtiene los datos del documento, luego procedemos a buscar el Lugar correcto donde debemos guardar la información del destino.

```
public void cargarDestinos(String path) {
    List<String> lineas = manipulacion.leerArchivo(path);

    for (String linea : lineas) {
        String datos[] = manipulacion.obtenerDatos(linea);

        if (!lugaresRepetidos(datos[0], datos[1])) {
            Lugar lugarOrigen = obtenerOCrearLugar(datos[0]);
            lugarOrigen.guardarDestinoCercano(obtenerOCrearLugar(datos[1]));
            informaciones.add(new InformacionRecorrido(datos[0], datos[1], datos[2], datos[3], datos[4],
datos[5], datos[6]));
        }
    }
}
```

Cargar Trafico

Este método es utilizado para guardar la información de transito que nos es proporcionado por el archivo de entrada, primero obtiene los datos del documento y luego procede a comparar los valores de origen y destino para determinar en donde se debe ingresar dicha información y lo guarda todo.

```
public void cargarTrafico(String path) {
    List<String> lineas = manipulacion.leerArchivo(path);

    for (String linea : lineas) {
```

```
        String datos[] = manipulacion.obtenerDatos(linea);


        for (InformacionRecorrido informacion : informaciones) {
          if (informacion.getOrigen().equals(datos[0]) && informacion.getDestino().equals(datos[1])) {
            informacion.guardarInformacionDelTrafico(datos[2], datos[3], datos[4]);
          }
        }
      }
    }
```

Leer Archivo

Este método recibe de parámetro el String de path que contiene la ubicación del archivo a leer y por medio de un lector de archivo extrae cada una de las líneas que están dentro del documento.

```
  public List<String> leerArchivo(String path) {
    File file = new File(path);
    BufferedReader entrada = null;
    List<String> contenido = new ArrayList<>();
    String linea;


    if (file.exists()) {
      try {
        entrada = new BufferedReader(new FileReader(file));


        while ((linea = entrada.readLine()) != null) {
          contenido.add(linea);
        }
      } catch (FileNotFoundException e) {
```

```
            System.out.println(e.getMessage());
        } catch (IOException e) {
            System.out.println(e.getMessage());
        }
    }
    return contenido;
}
```

Obtener Datos

Con este método podemos obtener los datos que se encuentran dentro de cada una de las líneas de nuestro documento que están separadas por “|” retornar un arreglo de String con cada una de las palabras que contenga esa línea.

```
public String[] obtenerDatos(String linea) {
    String caracteres[] = linea.split("");
    String datos[] = new String[7];
    int indice = 0, contador = 0;

    while (indice < 7) {
        String dato = "";

        while (contador < caracteres.length) {
            if (!"|".equals(caracteres[contador])) {
                dato += caracteres[contador];
                contador++;
            } else {
                break;
            }
        }
        contador++;
```

```
        datos[indice] = dato;
        dato = "";
        indice++;
    }
    return eliminarEspacios(datos);
}
```

Obtener o Crear Lugar

Esta función es utilizada para localizar un Lugar que ya haya sido creado previamente con el objetivo de no tener datos repetidos dentro del programa, caso contrario en donde haya terminado de recorrer toda la lista y no haya encontrado igualdad procede a instanciar un objeto de la misma clase y pasarle los valores correspondientes.

```
private Lugar obtenerOCrearLugar(String destino) {
    for (Lugar lugar : lugares) {
        if (lugar.getNombre().equals(destino)) {
            return lugar;
        }
    }

    Lugar nuevoLugar = new Lugar(destino);
    lugares.add(nuevoLugar);
    return nuevoLugar;
}
```

Árbol

Analizar Lugar

Este método es utilizado para agregar un nuevo nodo al árbol y para esto primero verifica si el nodo raíz del árbol ya llego a su capacidad máxima que en este caso es de 5 si con el dato actual llega a 5 se debe hacer una división del árbol para que podamos seguir los lineamientos del árbol B, en caso contrario solo debemos ingresar le valor nuevo.

```
public void analizarLugar(int llave, String contenido) {
    NodoArbol r = raiz;

    if (r.cantidadValoresGuardados == (rango-1)) { //4
        NodoArbol nuevo = new NodoArbol(rango);
        raiz = nuevo;
        nuevo.esUnaHoja = false;
        nuevo.cantidadValoresGuardados = 0;
        nuevo.hijos[0] = r;
        dividir(nuevo, 0, r);
        insertar(nuevo, llave, contenido);
    } else {
        insertar(r, llave, contenido);
    }
}
```

Dividir

Este método es utilizado cuando la raíz o cualquiera de las hojas de nuestro árbol haya llegado a su capacidad máxima y sea necesario realizar un ajuste para que se pueda seguir el modelo de un árbol b y para esto debemos dividir el arreglo en dos y el número que este en el centro lo debemos subir al árbol anterior en caso sea la raíz debemos agregarla de igual manera y para los dos casos tenemos comprobar que ese arreglo tampoco haya llegado a su capacidad máxima sino tocara repetir el proceso.

```
private void dividir(NodoArbol raiz, int i, NodoArbol anterior) {
    NodoArbol temporal = new NodoArbol(rango);
    temporal.esUnaHoja = anterior.esUnaHoja;
    temporal.cantidadValoresGuardados = ((rango/2)-1); //1

    int limite = rango/2+1;
    for (int j = 0; j < (rango/2); j++) { //2
        temporal.valores[j] = anterior.valores[(j + limite)]; //3
    }

    if (!anterior.esUnaHoja) {
        for (int k = 0; k < limite; k++) { //3
            temporal.hijos[k] = anterior.hijos[(k + limite)]; //3
        }
    }
    anterior.cantidadValoresGuardados = (rango/2); //2

    for (int j = raiz.cantidadValoresGuardados; j > i; j--) {
        raiz.hijos[(j + 1)] = raiz.hijos[j];
    }
    raiz.hijos[(i + 1)] = temporal;

    for (int j = raiz.cantidadValoresGuardados; j > i; j--) {
        raiz.valores[(j + 1)] = raiz.valores[j];
    }

    raiz.valores[i] = anterior.valores[(rango/2)]; //2
    raiz.cantidadValoresGuardados++;
}
```

Insertar

Este método para ingresar el valor correspondiente en el lugar correcto para esto verificamos que el lugar a ingresar sea o una hoja o un centro, si es una hoja solo debemos ingresar el valor sin hacer cambios, de lo contrario debemos recorrer el arreglo de hijos hasta localizar el lugar correcto para ingresar el nuevo valor.

```
private void insertar(NodoArbol x, int llave, String contenido) {


  if (x.esUnaHoja) {
    int i = x.cantidadValoresGuardados;


    while (i >= 1 && llave < x.valores[i - 1].getLlave()) {
      x.valores[i] = x.valores[i - 1];
      i--;
    }
    x.valores[i].setLlave(llave);
    x.valores[i].setContenido(contenido);
    x.cantidadValoresGuardados++;
  } else {
    int j = 0;
    while (j < x.cantidadValoresGuardados && llave > x.valores[j].getLlave()) {
      j++;
    }


    if (x.hijos[j].cantidadValoresGuardados == (rango-1)) { //4
      dividir(x, j, x.hijos[j]);


      if (llave > x.valores[j].getLlave()) {
        j++;
      }
    }
    insertar(x.hijos[j], llave, contenido);
```

```
    }
}
```

Imprimir

Este método es utilizado para crear toda la información con respecto a la creación del árbol B con ayuda de Graphivz por lo tanto este método crea las conexiones entre nodos para que puedan ser utilizadas por la gráfica.

```
private String imprimir(NodoArbol actual, int indice, int contador) {
    String contenido = actual.imprimirValores(indice) + "\n";


    if (!actual.esUnaHoja) {
        indice++;
        for (int j = 0; j <= actual.cantidadValoresGuardados; j++) {
            if (actual.hijos[j] != null) {
                contenido += imprimir(actual.hijos[j], indice, (contador+1));
                contenido += "\t\"node" + contador + "\":f0 -> \"node" + (indice) + "\":f0;\n";
                indice++;
            }
        }
    }
    return contenido;
}
```

NodoArbol

Iniciar Valores

Este método es utilizado para inicializar la lista de Valores y así no tener valores nulos al momento de realizar las comparaciones.

```
private void iniciarValores() {
    for (int i = 0; i < valores.length; i++) {
        valores[i] = new Valor();
    }
}
```

Imprimir Valores

Crea las conexiones entre todas las rutas que son los valores que nosotros vamos a insertar en el árbol B.

```
public String imprimirValores(int indice) {
    int contador = 0;
    String contenido = "\tnode"+indice+" [label = \"";

    for (int i = 0; i < cantidadValoresGuardados; i++) {
        if (i < cantidadValoresGuardados - 1) {
            contenido += "<f" + contador + "> " + valores[i].getContenido() + " | ";
            contador++;
        } else {
            contenido += "<f" + contador + "> " + valores[i].getContenido() + "\" shape = \"record\"];";
        }
    }
    return contenido;
}
```

Valor

Esta clase es utilizada para agregar información al árbol b, y dado que necesita un índice para poder ingresar entonces esta clase guarda el índice y guarda el mensaje que en este caso es la ruta que puede tomar el usuario. Tiene tanto su constructor inicializado como sus getters.

```
public class Valor {
    private String contenido;
    private int llave;

    public Valor() {
        this.contenido = "";
        this.llave = 0;
    }

    public String getContenido() {
        return contenido;
    }

    public void setContenido(String contenido) {
        this.contenido = contenido;
    }

    public int getLlave() {
        return llave;
    }

    public void setLlave(int llave) {
        this.llave = llave;
    }
}
```

Acciones

Esta clase es utilizada para realizar las acciones de rutas con relación al establecimiento de rutas y el cálculo de las rutas más eficientes.

Establecer Caminos

Este método es utilizado para determinar los caminos factibles que el usuario puede tomar para llegar a su destino, además marca el nodo visitado con el objetivo que no se vuelva a pasar sobre el mismo y así evitamos ciclos infinitos. Además que funciona de manera recursiva ya que visita a los nodos del arreglo y luego visita a los nodos de ese arreglo así hasta llegar al final.

```
  public void establecerCaminos(Lugar lugarOrigen, String nombreDestino, String union, List<String>
caminos) {
    List<Lugar> lugaresCercanos = lugarOrigen.getLugaresCercanos();

    if (!lugarOrigen.isVisitado()) {
      lugarOrigen.setVisitado(true);

      if (!lugaresCercanos.isEmpty()) {
        for (Lugar lugarCerca : lugaresCercanos) {
          if (lugarCerca.getNombre().equals(nombreDestino)) {
            caminos.add(union + "," + lugarCerca.getNombre());
          } else {
            establecerCaminos(lugarCerca, nombreDestino, (union + "," + lugarCerca.getNombre()),
caminos);
          }
        }
      }
    }
  }
```

Marcar Caminos

Este método es utilizado cuando queremos graficar el mapa y por medio de los caminos previamente definidos podemos determinar que nodos necesitamos graficar.

```
public void marcarCaminos(List<Lugar> lugares, List<String> caminos) {
    desmarcarCaminos(lugares);
    for (String camino : caminos) {
      String[] partes = camino.split(",");

      for (int i = 0; i < partes.length; i++) {
        for (Lugar lugar : lugares) {
          if (lugar.getNombre().equals(partes[i])) {
            lugar.marcarRuta();
            break;
          }
        }
      }
    }

    String[] partes = caminos.get(0).split(",");
    int contador = 0, indice = 0;

    while (contador != 2) {
      for (Lugar lugar : lugares) {
        if (lugar.getNombre().equals(partes[indice])) {
          lugar.marcarOrigenODestino();
          indice = (partes.length-1);
          contador++;
          break;
        }
      }
```

```
    }
  }
```

Desmarcar Caminos

Este método es utilizado para desmarcar todos los nodos de la lista de lugares con el objetivo de volver a recalcular las rutas y por lo tanto lo debemos limpiar cualquier tipo de conexión que exista previamente.

```
  public void desmarcarCaminos(List<Lugar> lugares) {
    for (Lugar lugar : lugares) {
      lugar.desmarcarOrigenODestino();
      lugar.desmarcarRuta();
    }
  }
```

Calcular Caminando

Este método es utilizado para calcular la eficiencia de cada una de las rutas con las que podemos llegar al destino y esto lo hacemos analizando cada nodo y realizando operaciones entre cada nodo con el objetivo de tener la suma completa del recorrido y poder compararlos entre sí.

```
  public void calcularCaminando(List<String> caminos, List<InformacionRecorrido> informaciones) {
    desgaste = new DatoUno[caminos.size()];
    distancia = new DatoUno[caminos.size()];
    desgasteYDistancia = new DatoDos[caminos.size()];
    rapidez = new DatoDos[caminos.size()];
    int indice = 0, sumaDesgaste = 0, sumaDistancia = 0, contador = 0;
    double sumaRapidez = 0.0;

    for (String camino : caminos) {
      String partes[] = camino.split(",");
      sumaDesgaste = 0; sumaDistancia = 0; sumaRapidez = 0; indice = 0;
```

```
        while (indice != (partes.length - 1)) {
          for (InformacionRecorrido informacion : informaciones) {
            if (informacion.getOrigen().equals(partes[indice]) &&
informacion.getDestino().equals(partes[indice + 1])) {
              sumaDesgaste += informacion.getGastoPersona();
              sumaDistancia += informacion.getDistancia();

              double distanciaDecimal = informacion.getDistancia();
              double tiempoDecimal = (informacion.getTiempoPersona()/60.0);

              sumaRapidez += (distanciaDecimal / tiempoDecimal);
              indice++;
              break;
            }
          }
        }
        desgaste[contador] = new DatoUno(camino, sumaDesgaste);
        distancia[contador] = new DatoUno(camino, sumaDistancia);
        rapidez[contador] = new DatoDos(camino, sumaRapidez);
        contador++;
      }
      ordenarArreglosUno(desgaste);
      ordenarArreglosUno(distancia);
      ordenarArreglosDos(desgasteYDistancia);
      ordenarArreglosDos(rapidez);
    }
```

Calcular en Vehículo

Este método es utilizado para determinar todos los valores con respecto a las rutas cuando nos estamos movilizando en carro y en este caso hay que tomar más valores y además

debemos analizar si existe o existirá tráfico durante todo el recorrido y sea que hubiera o no debemos notificarle al usuario.

```
   public void calcularEnVehiculo(List<String> caminos, List<InformacionRecorrido> informaciones, int
horaRecorrido) {

      desgaste = new DatoUno[caminos.size()];

      distancia = new DatoUno[caminos.size()];

      desgasteYDistancia = new DatoDos[caminos.size()];

      rapidez = new DatoDos[caminos.size()];

      int indice = 0, sumaDesgasteGasolina = 0, sumaDistancia = 0, contador = 0;

      double sumaRapidez = 0.0;


      for (String camino : caminos) {

         String caminoTrafico = camino;

         String partes[] = camino.split(",");

         sumaDesgasteGasolina = 0; sumaDistancia = 0; sumaRapidez = 0;

         indice = 0;


         while (indice != (partes.length - 1)) {

            for (InformacionRecorrido informacion : informaciones) {

               if (informacion.getOrigen().equals(partes[indice]) &&
informacion.getDestino().equals(partes[indice + 1])) {

                  sumaDesgasteGasolina += informacion.getGastoVehiculo();

                  sumaDistancia += informacion.getDistancia();

                  double distanciaDecimal = informacion.getDistancia(), tiempoDecimal =
informacion.getTiempoVehiculo()/60.0;


                  // Toca verificar la hora

                  if (horaRecorrido >= informacion.getHoraInicio() && horaRecorrido <=
informacion.getHoraFinal()) {

                     caminoTrafico += " (CON TRÁFICO)";

                     sumaRapidez += (distanciaDecimal / (tiempoDecimal * (1 +
informacion.getProbabilidad())));
```

```
                } else {
                    int minutosActuales = informacion.getTiempoVehiculo();

                    while (minutosActuales >= 60) {
                        minutosActuales -= 60;
                        horaRecorrido++;
                    }

                    if (horaRecorrido >= informacion.getHoraInicio() && horaRecorrido <=
informacion.getHoraFinal()) {
                        caminoTrafico += " (CON TRÁFICO)";
                        sumaRapidez += (distanciaDecimal / (tiempoDecimal * (1 +
(informacion.getProbabilidad()))));
                    } else {
                        sumaRapidez += (distanciaDecimal / tiempoDecimal);
                    }
                }
                indice++;
                break;
            }
        }
    }
    desgaste[contador] = new DatoUno(camino, sumaDesgasteGasolina);
    distancia[contador] = new DatoUno(camino, sumaDistancia);
    desgasteYDistancia[contador] = new DatoDos(camino, (double) sumaDistancia / (double)
sumaDesgasteGasolina);
    rapidez[contador] = new DatoDos(caminoTrafico, sumaRapidez);
    contador++;
  }
  ordenarArreglosUno(desgaste);
  ordenarArreglosUno(distancia);
```

```
    ordenarArreglosDos(desgasteYDistancia);
    ordenarArreglosDos(rapidez);
}
```

Ordenar Arreglo

Ordena todos los arreglos de las rutas para así tener todos los resultados menores al inicio del arreglo y los mayores al final.

```
private void ordenarArreglosUno(DatoUno[] arreglo) {
    for (int i = 1; i <= arreglo.length - 1; i++) {
        DatoUno datoAMover = arreglo[i];

        for (int j = i - 1; j >= 0; j--) {
            if (arreglo[j].getCantidad() > datoAMover.getCantidad()) {
                arreglo[j + 1] = arreglo[j];
                arreglo[j] = datoAMover;
            }
        }
    }
}

private void ordenarArreglosDos(DatoDos[] arreglo) {
    for (int i = 1; i <= arreglo.length - 1; i++) {
        DatoDos datoAMover = arreglo[i];

        for (int j = i - 1; j >= 0; j--) {
            if (arreglo[j].getCantidadDecimal()> datoAMover.getCantidadDecimal()) {
                arreglo[j + 1] = arreglo[j];
                arreglo[j] = datoAMover;
            }
        }
```

```
    }
}
```

Retornar Arreglos

Estos métodos retornar los arreglos de las rutas.

```
public DatoUno[] getDesgaste() {
    return desgaste;
}

public DatoUno[] getDistancia() {
    return distancia;
}

public DatoDos[] getDesgasteYDistancia() {
    return desgasteYDistancia;
}

public DatoDos[] getRapidez() {
    return rapidez;
}
```

Datos Uno

Esta clase se encarga de almacenar la ruta y la velocidad del camino, pero en este caso no necesitamos de un valor tan preciso, por lo tanto, realizamos una aproximación que nos brindara una idea de la distancia, el desgaste o el tiempo en llegar a nuestro destino.

```
public class DatoUno {

    private String ruta;
    private int cantidad;
```

```
    public DatoUno(String ruta, int cantidad) {
        this.ruta = ruta;
        this.cantidad = cantidad;
    }

    public String getRuta() {
        return ruta;
    }

    public int getCantidad() {
        return cantidad;
    }
}
```

Datos Dos

Esta clase se encarga de almacenar la ruta y la velocidad del camino, pero en este caso para tener un valor más exacto en cuanto a los datos se busca que el valor este representado en doublé para que pueda ser visto de una mejor manera, además de que, al momento de obtener la cantidad de la velocidad, hacemos una conversión para que el valor nos quede con un punto decimal y 2 cifras decimales.

```
public class DatoDos {

    private String ruta;
    private double cantidadDecimal;

    public DatoDos(String ruta, double cantidadDecimal) {
        this.ruta = ruta;
        this.cantidadDecimal = cantidadDecimal;
```

```
    }

    public String getRuta() {
        return ruta;
    }

    public double getCantidadDecimal() {
        String conversion = String.format("%." + 2 + "f", cantidadDecimal);
        return Double.parseDouble(conversion);
    }
}
```

Información Recorrido

Esta clase es utilizada para poder guardar la información del recorrido del mapa como es el origen y el destino, el desgaste del usuario cuando va caminando, el desgaste cuando va en vehículo, el tiempo promedio en llegar a su destino y la distancia a la que se encuentra un lugar del otro, además guarda la información del trafico presente en dicha ruta a diferentes horas del día.

```
public class InformacionRecorrido {

    // Datos Iniciales
    String origen;
    String destino;
    int tiempoVehiculo, tiempoPersona;
    int gastoVehiculo, gastoPersona;
    int distancia;
    // Datos Extras
    List<Trafico> horarios;
```

```
   public InformacionRecorrido(String origen, String destino, String tiempoVehiculo, String
tiempoPersona, String gastoVehiculo, String gastoPersona, String distancia) {
      this.origen = origen;
      this.destino = destino;
      this.tiempoVehiculo = Integer.parseInt(tiempoVehiculo);
      this.tiempoPersona = Integer.parseInt(tiempoPersona);
      this.gastoVehiculo = Integer.parseInt(gastoVehiculo);
      this.gastoPersona = Integer.parseInt(gastoPersona);
      this.distancia = Integer.parseInt(distancia);
      horarios = new ArrayList<>();
   }

   public void guardarInformacionDelTrafico(String horaInicio, String horaFinal, String probabilidad) {
      horarios.add(new Trafico(Integer.parseInt(horaInicio), Integer.parseInt(horaFinal),
Integer.parseInt(probabilidad)));
   }

   public String getOrigen() {
      return origen;
   }

   public String getDestino() {
      return destino;
   }

   public int getTiempoVehiculo() {
      return tiempoVehiculo;
   }

   public int getTiempoPersona() {
      return tiempoPersona;
```

```
    }

    public int getGastoVehiculo() {
        return gastoVehiculo;
    }

    public int getGastoPersona() {
        return gastoPersona;
    }

    public int getDistancia() {
        return distancia;
    }

    public List<Trafico> getHorarios() {
        return horarios;
    }
}
```

Lugar

Esta clase es utilizada para guardar el lugar de origen y por medio de una lista de Lugares guarda el lugar siguiente al cual podemos acceder, además tenemos valores booleanos que nos van a servir para realizar el análisis correspondiente al momento de realizar los caminos y de determinar cuál es el lugar de inicio y fin del recorrido al momento de realizar la gráfica del mapa.

```
public class Lugar {

    private String nombre;
    private List<Lugar> lugaresCercanos;
    private boolean parteDeLaRuta = false;
```

```
private boolean origenODestino = false;
private boolean visitado = false;
private boolean pasado = false;

public Lugar(String nombre) {
   this.nombre = nombre;
   this.lugaresCercanos = new ArrayList<>();
}

public void guardarDestinoCercano(Lugar destino) {
   lugaresCercanos.add(destino);
}

public String getNombre() {
   return nombre;
}

public List<Lugar> getLugaresCercanos() {
   return lugaresCercanos;
}

public void marcarRuta() {
   parteDeLaRuta = true;
}

public void desmarcarRuta() {
   parteDeLaRuta = false;
}

public boolean isParteDeLaRuta() {
```

```
    return parteDeLaRuta;
}

public void marcarOrigenODestino() {
    origenODestino = true;
}

public void desmarcarOrigenODestino() {
    origenODestino = false;
}

public boolean isOrigenODestino() {
    return origenODestino;
}

public boolean isVisitado() {
    return visitado;
}

public void setVisitado(boolean visitado) {
    this.visitado = visitado;
}

public boolean isPasado() {
    return pasado;
}

public void setPasado(boolean pasado) {
    this.pasado = pasado;
}
```

Trafico

Esta clase es utilizada para guardar toda la información relacionada con el transporte, es decir la hora de inicio del tráfico, la hora de finalización del tráfico y la probabilidad de que efectivamente en ese rango de tiempo se encuentre el tráfico, y dado de que puede haber tráfico a distintas horas es necesario guardar cada posibilidad en una lista de opciones que nos servirá más tarde al momento de determinar las rutas.

```
public class Trafico {


   private int horaInicio;
   private int horaFinal;
   private int probabilidad;


   public Trafico(int horaInicio, int horaFinal, int probabilidad) {
      this.horaInicio = horaInicio;
      this.horaFinal = horaFinal;
      this.probabilidad = probabilidad;
   }


   public int getHoraInicio() {
      return horaInicio;
   }


   public int getHoraFinal() {
      return horaFinal;
   }


   public int getProbabilidad() {
      return probabilidad;
   }
}
```

Mapa Frame

Esta clase es la primera ventana que se ejecuta al momento de realizar el iniciar nuestro programa, y donde el usuario va poder interactuar, entre los métodos que tenemos están los siguientes:

Botón Caminar

Este botón al seleccionarlo se apaga y marca por medio de un booleano que el botón fue seleccionado.

```
private void botonCaminarActionPerformed(java.awt.event.ActionEvent evt) {
    caminarSeleccionado = true; vehiculoSeleccionado = false;
    botonCaminar.setEnabled(false);
    botonCarro.setEnabled(true);
}
```

Botón Vehículo

Este botón al seleccionarlo se apaga y marca por medio de un booleano que el botón fue seleccionado.

```
private void botonCarroActionPerformed(java.awt.event.ActionEvent evt) {
    caminarSeleccionado = false; vehiculoSeleccionado = true;
    botonCarro.setEnabled(false);
    botonCaminar.setEnabled(true);
}
```

Iniciar Viaje

Este botón al pulsarlo hace unas verificaciones previas para determinar que todos los datos estén correctos, por ejemplo, verifica que el origen y el destino no sean el mismo, además de verificar que se haya elegido un método de trasporte para el viaje, por ultimo si resulta que no hay caminos que se puedan seguir no desactiva los comandos.

```
private void iniciarViajeActionPerformed(java.awt.event.ActionEvent evt) {
    if (!lugares.isEmpty()) {
        if (relojBloqueado) {
```

```
            if (!getNombreOrigen().equals(getNombreDestino())) {
               if (caminarSeleccionado || vehiculoSeleccionado) {
                  boolean respuesta = recalcularDatos(getNombreOrigen());

                  if (respuesta) {
                     activarODesactivarControles(false, true);
                     origen = getNombreOrigen();
                  }
               } else { JOptionPane.showMessageDialog(this,"Elige un método de
Viaje","Viaje",JOptionPane.ERROR_MESSAGE); }
            } else { JOptionPane.showMessageDialog(this,"No puedes seleccionar el mismo Lugar de
Origen","Error",JOptionPane.ERROR_MESSAGE); }
         } else { JOptionPane.showMessageDialog(this,"Debes Bloquear el Reloj para establecer una
Hora","Error",JOptionPane.ERROR_MESSAGE); }
      } else { JOptionPane.showMessageDialog(this,"Debes Cargar un Archivo para poder
Seleccionar","Error",JOptionPane.ERROR_MESSAGE); }
   }
```

Siguiente

Este método es utilizado cuando el usuario a seleccionado uno de los siguientes destinos a los cuales puede ir, primero verifica que no haya llegado a su destino en caso fuera el caso le notifica al usuario que ha llegado a su destino, se procede a desmarcar toda la ruta que nos queda y se procede a volver a dibujar el mapa a su manera predeterminada, por otro lado, se deshabilita la sección de seleccionar y se vuelve a activar el de Configuración y del Reloj.

En caso todavía no llega a su destino tómanos el valor del siguiente movimiento del usuario y lo mandamos a recalcular con el método correspondiente, y modificamos la hora para que el usuario sepa en qué momento llega a ese destino y volvemos a recargar el mapa y se lo volvemos a mostrar.

```
   private void botonSiguienteActionPerformed(java.awt.event.ActionEvent evt) {
      if (getNombreLugarEscogido().equals(getNombreDestino())) {
         JOptionPane.showMessageDialog(this,"Ha llegado a su
Destino","Final",JOptionPane.INFORMATION_MESSAGE);
```

```
        actualizarHora();
        activarODesactivarControles(true, false);
        caminarSeleccionado = false; vehiculoSeleccionado = false; relojBloqueado = false;
        botonRestaurarHora.setEnabled(true); botonEditarHora.setEnabled(true);
botonBloquearHora.setEnabled(true);
        Acciones accion = new Acciones();
        accion.desmarcarCaminos(lugares);
        comboSiguiente.removeAllItems();
        opciones.setText("");

        if (caminarSeleccionado) {
           for (Lugar lugarDoble : lugaresDobles) {
              lugarDoble.setPasado(false);
           }
        } else {
           for (Lugar lugar : lugares) {
              lugar.setPasado(false);
           }
        }

        colocarImagen(1);
      } else {
        if (caminarSeleccionado) {
           for (Lugar lugarDoble : lugaresDobles) {
              if (lugarDoble.getNombre().equals(origen)) {
                 lugarDoble.setPasado(true);
              }
           }
        } else {
           for (Lugar lugar : lugares) {
```

```
            if (lugar.getNombre().equals(origen)) {
                lugar.setPasado(true);
            }
          }
        }
        actualizarHora();
        recalcularDatos(getNombreLugarEscogido());
    }
}
```

Reloj

Estos métodos que se presentan a continuación son utilizados para el manejo del reloj, con el primer método lo que hacemos es obtener la hora actual del dispositivo y proyectarlo en el programa, con el segundo método abrimos una nueva vista el programa en donde se le va solicitar al usuario que seleccione una hora y los minutos al momento de darle en aceptar obtendremos esos valores, se los pasaremos al reloj y este iniciara con esos valores. Por ultimo esta la opción de bloquear que establecerá a la hora de inicio del viaje y bloqueará los otros botones.

```
private void botonRestaurarHoraActionPerformed(java.awt.event.ActionEvent evt) {
    reloj.detener();
    activarReloj();
}


private void botonEditarHoraActionPerformed(java.awt.event.ActionEvent evt) {
    reloj.detener();
    EditarReloj editarReloj = new EditarReloj(this);
    editarReloj.setVisible(true);
    int horas = editarReloj.getHora();
    int minutos = editarReloj.getMinutos();
    reloj = new Reloj(labelRelojMostrar, horas, minutos);
    reloj.start();
```

```
    }


    private void botonBloquearHoraActionPerformed(java.awt.event.ActionEvent evt) {
        String obtenerHora = labelRelojMostrar.getText();
        reloj.detener();
        String[] partesHora = obtenerHora.split(":");
        horaComienzo = Integer.parseInt(partesHora[0]);
        relojBloqueado = true;
        botonRestaurarHora.setEnabled(false);                    botonEditarHora.setEnabled(false);
botonBloquearHora.setEnabled(false);
    }
```

Por otra parte, el siguiente método es solo para iniciar al conteo del reloj vigilando que este cumpla con ciertos parámetros como lo son que las horas no deben sobrepasar a las 23 horas, y que los minutos no deben sobrepasar los 59 minutos.

```
    private void actualizarHora() {
        for (InformacionRecorrido informacion : informaciones) {
            if ((informacion.getOrigen().equals(origen) &&
informacion.getDestino().equals(getNombreLugarEscogido())) ||
(informacion.getOrigen().equals(getNombreLugarEscogido()) &&
informacion.getDestino().equals(origen))) {
                String obtenerHora = labelRelojMostrar.getText();
                String[] partesHora = obtenerHora.split(":");
                String horaEscrita = "", minutoEscrito = "";
                int minutosASumar = 0;


                if (caminarSeleccionado) {
                    minutosASumar += informacion.getTiempoPersona();
                } else {
                    minutosASumar += informacion.getTiempoVehiculo();
                }
                int horasActuales = Integer.parseInt(partesHora[0]);
```

```
int minutosActuales = Integer.parseInt(partesHora[1]);

minutosActuales += minutosASumar;

while (minutosActuales >= 60) {
   minutosActuales -= 60;
   horasActuales++;
}

int extra = 0;

while (horasActuales > 24) {
   horasActuales--;
   extra++;
}
if (horasActuales == 24) {
   horasActuales = extra;
}

if (horasActuales < 10) {
   horaEscrita = "0"+horasActuales;
} else {
   horaEscrita = ""+horasActuales;
}

if (minutosActuales < 10) {
   minutoEscrito = "0"+minutosActuales;
} else {
   minutoEscrito = ""+minutosActuales;
}
```

```
            labelRelojMostrar.setText(horaEscrita + ":" + minutoEscrito);
            origen = getNombreLugarEscogido();
        }
    }
}
```

Recalcular Datos

Este método es utilizado para recalcular los caminos por los cuales puede pasar el usuario en caso los caminos nos dieran que están vacíos le notificaríamos al usuario que no existe conexión entre el origen y el destino que el usuario escogió, por el contrario, si existen caminos, procedemos a marcar, generar la gráfica y mostrárselo al usuario.

```
private boolean recalcularDatos(String origen) {
    Lugar lugarOrigen = null;
    List<Lugar> opciones;

    if (caminarSeleccionado) {
        opciones = lugaresDobles;
    } else {
        opciones = lugares;
    }

    for (Lugar lugar : opciones) {
        if (lugar.getNombre().equals(origen)) {
            lugarOrigen = lugar;
            break;
        }
    }
    caminos = new ArrayList<>();
    Acciones accion = new Acciones();
    accion.establecerCaminos(lugarOrigen, getNombreDestino(), lugarOrigen.getNombre(), caminos);
```

```
if (!caminos.isEmpty()) {

    for (Lugar lugar : opciones) {
        lugar.setVisitado(false);
    }

    accion.marcarCaminos(opciones, caminos);

    if (caminarSeleccionado) {
        colocarImagen(2);
    } else {
        colocarImagen(1);
    }



    if (caminarSeleccionado) {
        accion.calcularCaminando(caminos, informaciones);
    } else {
        accion.calcularEnVehiculo(caminos, informaciones, horaComienzo);
    }

    if (!caminos.isEmpty()) {
        desgastes = accion.getDesgaste();
        distancias = accion.getDistancia();
        desgasteYDistancia = accion.getDesgasteYDistancia();
        rapidez = accion.getRapidez();
        mostrarOpciones();
        mostrarSiguiente();
    }
    return true;
```

```
    } else {
        JOptionPane.showMessageDialog(this,"No hay conexión entre el Origen y el Desino","Error",JOptionPane.ERROR_MESSAGE);
        return false;
    }
  }
```

Mostrar Opciones

Este método muestras las opciones que el usuario tiene para seguir una ruta, y en este caso se le mostrarán las rutas que son más eficiente y su contraparte, tomando como referencia una serie de parámetros que el usuario podrá visualizar, es solo una guía y no es necesario que lo tome en cuenta ya que al final es el usuario el que toma la decisión de a dónde ir.

```
  private void mostrarOpciones() {
    String contenido = "";
    contenido += "Mejores rutas por:\n";
    contenido += "\t1) Desgaste: " + desgastes[0].getRuta() + " -> " + desgastes[0].getCantidad() + " RPE\n";
    contenido += "\t2) Distancia: " + distancias[0].getRuta() + " -> " + distancias[0].getCantidad() + " km\n";
    contenido += "\t3) Desgaste y Distancia: " + desgasteYDistancia[desgasteYDistancia.length-1].getRuta() + " -> " + desgasteYDistancia[desgasteYDistancia.length-1].getCantidadDecimal()+ " km/RPE\n";
    contenido += "\t4) Rapidez: " + rapidez[rapidez.length-1].getRuta() + " -> " + rapidez[rapidez.length-1].getCantidadDecimal()+ " km/h\n";
    contenido += "\n";
    contenido += "Peores rutas por:\n";
    contenido += "\t1) Desgaste: " + desgastes[desgastes.length-1].getRuta() + " -> " + desgastes[desgastes.length-1].getCantidad() + " RPE\n";
    contenido += "\t2) Distancia: " + distancias[distancias.length-1].getRuta() + " -> " + distancias[distancias.length-1].getCantidad() + " km\n";
    contenido += "\t3) Desgaste y Distancia: " + desgasteYDistancia[0].getRuta() + " -> " + desgasteYDistancia[0].getCantidadDecimal()+ " km/RPE\n";
    contenido += "\t4) Rapidez: " + rapidez[0].getRuta() + " -> " + rapidez[0].getCantidadDecimal()+ " km/h";
```

```
    contenido += "\n";
    opciones.setText("");
    opciones.setText(contenido);
  }
```

Colocar Imagen

El siguiente método es utilizado para colocar la imagen en el jLabel y para esto al momento de construir la imagen se verifica que el archivo exista y al momento que sabemos que el archivo existe, esperamos 2 segundos y lo mostramos en el panel, esto con el fin de evitar algún error al momento de cargar la imagen.

```
  private void colocarImagen(int numero) {
    try {
      File file = new File("ImagenMapa.png");
      file.delete();
      GraficaMapa grafica = new GraficaMapa(lugares, lugaresDobles);
      grafica.generarImagen(numero);
      file = new File("ImagenMapa.png");

      while (true) {
        if (file.exists()) {
          Thread.sleep(2000);
          labelMapa.setIcon(new ImageIcon(new
ImageIcon("ImagenMapa.png").getImage().getScaledInstance(-1, -1, java.awt.Image.SCALE_SMOOTH)));
          break;
        }
      }
    } catch (InterruptedException ex) {
      System.out.println("Error durante la Espera de tipo: " + ex.getMessage());
    }
  }
```