Manual Técnico

Brandon Josué Pinto Méndez

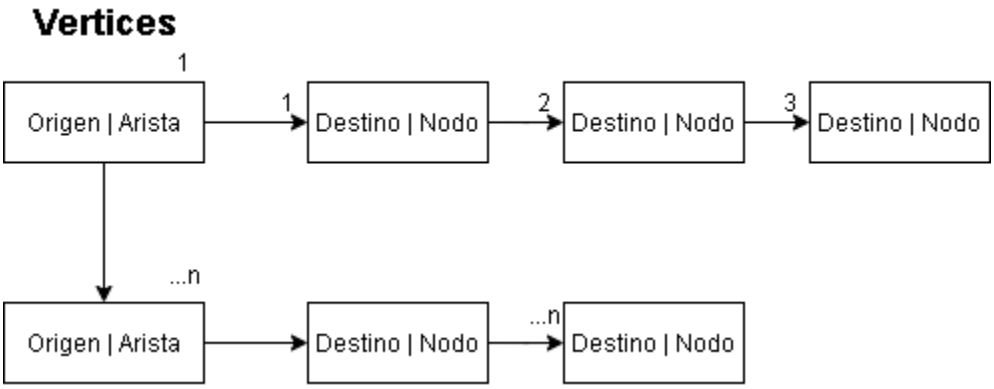
201930236

Estructura de datos del programa

El programa está hecho con las siguientes estructuras de datos:

Grafos:

Los grafos nos sirven para el manejo de los lugares y hacer los cálculos para encontrar las rutas dependiendo de las características que el usuario necesite, la estructura de la creación de grafos se basa en la siguiente gráfica:



Donde podemos observar los siguientes elementos:

- Vértice: estos contienen 2 elementos, una cadena String en la cual se guarda el lugar de origen y una Artista.
- Arista: Esta cuenta con una cadena String la cual guarda el lugar de destino y un Nodo
- Nodo: en este su guardan todos los datos para el transporte por el cual funcionara el programa, cuenta con datos como la distancia, trafico, gasto de gasolina, etc.

Como podemos observar podemos tener un numero ilimitados de Vértices los cuales cuenten con un numero cualquiera de aristas, esto es creado dinámicamente para poder optimizar la estructura del grafo, esto se hace a través de datos tipo ArrayList y de las clases de cada uno de los elementos mencionados anteriormente, a continuación, se detalla cada una de estas clases:

Las siguientes clases son clases de objetos y se mencionaran los elementos que guardan junto a ellas:

Clase vértices:

private String Origen;

private ArrayList<Arista> Aristas;

Clase Aristas:

private String Destino;

private Nodo info;

Clase Nodo:

```
private int tiempoVehiculo;
private int tiempoPie;
private int consumoGas;
private int desgastePersona;
private int distancia;
private ArrayList<Integer> HoraInicio = 0;
private ArrayList<Integer> HoraFinal = 0;
private ArrayList<Integer> Trafico = 0;
```

Con estos elementos podemos entender la construcción del grafo, y el como y donde se guardan los datos, para la implementación de este solo se crea un arrayList tipo Vértice "ArrayList<Vértice>". Luego solo guardamos los datos necesarios en este.

Inserción del grafo:

Para poder generar nuestro grafo dependemos de un archivo de entrada el cual iniciara todo, de este sacaremos todos los posibles recorridos y podremos clasificar toda la información, el algoritmo para esto es el siguiente:

```
public static ArrayList<Vertices> leerArchivo(String nombreArchivo) {
    ArrayList<Vertices> Grafo = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new FileReader(nombreArchivo))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            String[] partes = linea.split("\\|");
            if (partes.length == 7) {
                //separamos los elementos
                String origen = partes[0];
                String destino = partes[1];
                int tiempoVehiculo = Integer.parseInt(partes[2]);
                int tiempoPie = Integer.parseInt(partes[3]);
                int consumoGas = Integer.parseInt(partes[4]);
```

```
                int desgastePersona = Integer.parseInt(partes[5]);
                int distancia = Integer.parseInt(partes[6]);
                //Primero creamos los nodos, luego las aristas y de ultimo los vertices y los agregamos a la lista a mandar
                Nodo NuevoNodo = new Nodo(tiempoVehiculo, tiempoPie, consumoGas, desgastePersona, distancia);
                Arista NuevaArista = new Arista(destino,NuevoNodo);
                ArrayList<Arista> ExisteVertice = ManejadorDelGrafo.DevolverArregloArista(Grafo,origen);
                if(ExisteVertice!=null){
                   //Existen vertices
                   if(ManejadorDelGrafo.ExisteArista(Grafo,origen,destino)){
                      //actualizamos el nodo existente
                      ManejadorDelGrafo.ActualizarArista(Grafo,origen,destino,NuevoNodo);
                   }else{
                      //inseratamos una nueva arista en el vertice
                      ManejadorDelGrafo.InsertarArista(Grafo,origen,NuevaArista);
                   }
                }else{
                   //no existe vertice, creamos el nuevo vertice
                   Vertices nuevo = new Vertices(origen);
                   nuevo.getAristas().add(NuevaArista);
                   Grafo.add(nuevo);
                }
             } else {
                // Manejar el caso de una línea con un formato incorrecto
                System.err.println("Formato incorrecto en la línea: " + linea);
             }
          }
```

```
        } catch (IOException e) {
            e.printStackTrace();
        } catch (NumberFormatException e) {
            e.printStackTrace();
            System.err.println("Error de formato en el archivo.");
        }
        return Grafo;
    }
```

Como podemos observar con este algoritmo separamos las cadenas del archivo línea por línea gracias al formato en el que viene y luego vamos creando nuevos vértices, aristas y nodos los cuales nos sirven para ir agregando a nuestro elemento Grafo como tal, al final de este algoritmo devolvemos el grafo completo.

Búsqueda del grafo:

Para la búsqueda del Grafo tenemos un método general el cual nos ayudara para dividir la búsqueda si se viaja en vehículo o caminando, este método se desarrolla de la siguiente forma:

```
public static void encontrarCaminos(String Origen, Vertices inicio, String fin, ArrayList<NodoRecorridoDeGrafo> Nodos, HashSet<String> visitados, ArrayList<Vertices> Grafo, Vertices Anterior, boolean vehiculo) {
        if (vehiculo) {
            encontrarCaminosRecursivo(Origen, inicio, fin, Nodos, visitados, Grafo, Anterior, new NodoRecorridoDeGrafo());
        } else {
            ArrayList<NodoRecorridoDeGrafo> NodosInversos = new ArrayList<>();
            ArrayList<NodoRecorridoDeGrafo> Nodosnormales = new ArrayList<>();
            encontrarCaminosRecursivo(Origen, inicio, fin, Nodosnormales, visitados, Grafo, Anterior, new NodoRecorridoDeGrafo());
            encontrarCaminosRecursivoInversos(fin, obtenerVertice(fin, Grafo), Origen, NodosInversos, visitados, Grafo, Anterior, new NodoRecorridoDeGrafo());
            // Eliminar elementos duplicados de nodosInversos que también están en nodosNormales
```

```
        for (NodoRecorridoDeGrafo nodoInverso : NodosInversos) {
            if (Nodosnormales.contains(nodoInverso)) {
                NodosInversos.remove(nodoInverso);
            }
        }
        Nodosnormales.addAll(NodosInversos);
        Nodos.addAll(Nodosnormales);
    }
}
```

Como podemos observar este método se separa en 2 principales, pero básicamente siguen la misma estructura:

```
private static void encontrarCaminosRecursivo(String Origen, Vertices inicio, String fin, ArrayList<NodoRecorridoDeGrafo> Nodos, HashSet<String> visitados, ArrayList<Vertices> Grafo, Vertices Anterior, NodoRecorridoDeGrafo nuevoNodo) {
    if (inicio == null) {
        // Si el vértice de inicio es nulo, salimos de la recursión
        return;
    }
    if (inicio.getOrigen().equals(fin)) {
        // Si hemos llegado al nodo de destino, agregamos el camino actual a la lista de caminos
        nuevoNodo.getRecorrido().add(inicio.getOrigen());
        Nodos.add(nuevoNodo);
        return;
    }
    visitados.add(inicio.getOrigen());
    for (Arista arista : inicio.getAristas()) {
        if (!visitados.contains(arista.getLugar())) {
            NodoRecorridoDeGrafo nuevoNodoRecursivo = new NodoRecorridoDeGrafo();
            nuevoNodoRecursivo.getRecorrido().addAll(nuevoNodo.getRecorrido());
```

```
            nuevoNodoRecursivo.getDatosTotales().addAll(nuevoNodo.getDatosTotales());

nuevoNodoRecursivo.getCalcCompuesto().addAll(nuevoNodo.getCalcCompuesto());
            nuevoNodoRecursivo.getRecorrido().add(inicio.getOrigen());
            nuevoNodoRecursivo.getDatosTotales().add(arista.getInfo());
            if (obtenerVertice(arista.getLugar(), Grafo)==null) {
               if (arista.getLugar().equals(fin)) {
                  nuevoNodoRecursivo.getRecorrido().add(arista.getLugar());
                  Nodos.add(nuevoNodoRecursivo);
               }
            }
            encontrarCaminosRecursivo(Origen, obtenerVertice(arista.getLugar(), Grafo), fin,
Nodos, visitados, Grafo, Anterior, nuevoNodoRecursivo);
         }
      }
      visitados.remove(inicio.getOrigen());
   }
```

Este es el método para recorrer recursivamente los nodos, básicamente va buscando los posibles caminos y creando Nodos tipo NodoRecorridoDeGrafo, el cual sirve para poder insertar en el árbol B de una manera más simple y fácil, la segunda forma es para poder conseguir todos los caminos incluso si vamos a pie, para esto simplemente se utiliza un método para tomar los caminos si fuera de manera al revés y luego comparar si ya existen los caminos para no repetirlos.

Árbol B: el árbol B en este programa se utiliza básicamente para poder hacer la función de ordenamiento de los caminos entregados por el grafo, esto lo implementamos de la siguiente forma:

En la parte más baja tenemos NodoArbolB, estos sirven para guardar los caminos recogidos del grafo, este tiene un formato más simple y nos sirven para poder guardar los datos que se meterán en el Arbol B, su estructura es de la siguiente forma:

```
public class NodoArbolB {
   private ArrayList<String> Recorrido;
```

```
    private int calculo;
```

donde recorrido nos sirve para guardar los lugares que visitamos en este recorrido y el calculo es el numero según los parámetros que queramos, este se implemente en el NodoB, este tipo de Nodo ya es el que utilizamos directamente en el árbol, se compone de los siguientes elementos:

```
public class NodoB {
    NodoArbolB[] datos;
    int gradoMinimo;
    ArrayList<NodoB> hijos;
    boolean hoja;
    int numDatos;
```

como podemos ver tenemos nuestros datos el cual es un arreglo de NodoArbolB, esto con el fin de cumplir el concepto de pagina que tienen los nodos tipo B así podemos definir un tamaño especifico y podemos saber cuando tenemos que realizar la división celular.

Luego procederemos con el árbol en sí, el cual cuenta con la siguiente estructura:

```
public class ArbolB {

    NodoB raiz;
    int gradoMinimo;
```

este de una manera más fácil está compuesto de un NodoB que será nuestra Raiz y el grado de esta, el cual define el tamaño de las páginas.

Insertar en el árbol:

El método el cual se encarga de la inserción se divide en 3 partes las cuales son las siguientes:

```
public void insertar(NodoArbolB Recorrido) {
        if (raiz == null) {
            raiz = new NodoB(gradoMinimo, true);
            raiz.datos[0] = Recorrido;
            raiz.numDatos = 1;
```

```
    } else {
        if (raiz.numDatos == 2 * gradoMinimo - 1) {
            NodoB nuevaRaiz = new NodoB(gradoMinimo, false);
            nuevaRaiz.hijos.add(raiz);
            dividirHijo(nuevaRaiz, 0, raiz);
            raiz = nuevaRaiz; // Actualizamos la raíz con la nueva raíz creada
            insertar(raiz, Recorrido); // Insertamos en la nueva raíz
        } else {
            insertar(raiz, Recorrido); // Insertamos en la raíz existente
        }
    }
}
//inserción caso 1 La mejor ruta en base a la gasolina si es vehículo.

private void dividirHijo(NodoB padre, int indice, NodoB hijo) {
    NodoB nuevoHijo = new NodoB(hijo.gradoMinimo, hijo.hoja);
    nuevoHijo.numDatos = gradoMinimo - 1;
    // Copiar datos al nuevo hijo
    for (int i = 0; i < gradoMinimo - 1; i++) {
        nuevoHijo.datos[i] = hijo.datos[i + gradoMinimo];
        hijo.datos[i + gradoMinimo] = null; // Limpiar datos del hijo original
    }
    // Copiar hijos al nuevo hijo si no es hoja
    if (!hijo.hoja) {
        for (int i = 0; i < gradoMinimo; i++) {
            nuevoHijo.hijos.add(hijo.hijos.get(i + gradoMinimo));
            hijo.hijos.set(i + gradoMinimo, null); // Limpiar hijos del hijo original
        }
```

```
    }
    // Ajustar los datos y los hijos del padre
    for (int i = padre.numDatos; i > indice; i--) {
        padre.hijos.add(i + 1, padre.hijos.get(i));
    }
    padre.hijos.add(indice + 1, nuevoHijo);

    for (int i = padre.numDatos - 1; i >= indice; i--) {
        padre.datos[i + 1] = padre.datos[i];
    }
    padre.datos[indice] = hijo.datos[gradoMinimo - 1];
    hijo.datos[gradoMinimo - 1] = null; // Limpiar el dato copiado al padre
    hijo.numDatos = gradoMinimo - 1;
    padre.numDatos++;
}

private void insertar(NodoB nodo, NodoArbolB clave) {
    int i = nodo.numDatos - 1;
    if (nodo.hoja) {
        // Si es una hoja, simplemente insertamos el dato en la posición correcta
        while (i >= 0 && clave.getCalculo() < nodo.datos[i].getCalculo()) {
            nodo.datos[i + 1] = nodo.datos[i];
            i--;
        }
        nodo.datos[i + 1] = clave;
        nodo.numDatos++;
    } else {
        while (i >= 0 && clave.getCalculo() < nodo.datos[i].getCalculo()) {
```

```
            i--;
        }
        i++; // Aumentamos i después del bucle
        if (nodo.hijos.get(i).numDatos == 2 * gradoMinimo - 1) {
            dividirHijo(nodo, i, nodo.hijos.get(i));
            // Después de dividir, si la clave es mayor que el dato en la posición i,
            // necesitamos mover a la siguiente posición para insertarla en el hijo adecuado
            if (clave.getCalculo() > nodo.datos[i].getCalculo()) {
                i++;
            }
        }
        // Luego, insertamos en el hijo correspondiente después de la posible división
        insertar(nodo.hijos.get(i), clave);
    }
}
```

Como podemos ver el principal es donde mandamos solo la raíz como parámetro, en este elemento se realizarán los demás procesos, luego de esto debemos mandar el parámetro que queramos insertar, ahora para poder insertar debemos de comprobar si aún hay espacio en nuestra pagina o hay que realizar una división celular, dependiendo de cual de las 2 opciones se sigan se invocarán a uno de los 2 métodos auxiliares.

Buscar en Arbol B:

Para buscar en el árbol B utilizamos el siguiente método:

```
public void MeterOrdenado(NodoB nodo, ArrayList<NodoArbolB> ListaOrdenada) {
    if (nodo != null) {
        int i;
        // Recorrer todos los hijos
        for (i = 0; i < nodo.numDatos; i++) {
            try {
                // Recorrer el subárbol izquierdo si no es una hoja
                if (!nodo.hoja) {
```

```
                MeterOrdenado(nodo.hijos.get(i), ListaOrdenada);
            }

            // Agregar el nodo actual a la lista
            ListaOrdenada.add(nodo.datos[i]);

            // Recorrer el subárbol derecho si no es una hoja y no es la última clave
            if (!nodo.hoja && i == nodo.numDatos - 1) {
                MeterOrdenado(nodo.hijos.get(i + 1), ListaOrdenada);
            }
        } catch (NullPointerException e) {
            // Manejar la excepción si es necesario
        }
    }
}
```

Como podemos observar vamos recorriendo las hojas hasta llegar a la hoja más baja y por lo tanto encontrar el mejor parámetro, luego de esto vamos subiendo en la escala, con esto podemos aprovecharlo para poder crear una arraylist de nodos ordenados el cual se llama ListaOrdenada, solo debemos ir sacando los nodos ordenados y añadirlos a la lista para tener el funcionamiento del árbol.

Estas serían las estructuras principales de este proyecto de programación y la explicación de su funcionamiento y razonamiento detrás de su uso.