UNIVERSIDAD DE LAS FUERZAS ARMADAS-ESPE SEDE SANTO DOMINGO

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN - DCCO-SS

CARRERA DE INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN

PERIODO	:	202351 noviembre 23 – marzo24
ASIGNATURA	:	Estructura de Datos
TEMA	:	Grafos
INTEGRANTES	:	Saldarriaga Gissela.
DOCENTE	:	Ing. Javir Cevallos, Mgtr
FECHA DE ENTREGA	:	22/02/2024

SANTO DOMINGO - ECUADOR

2024

Índice de contenido.

Índice de figuras.

1. Introducción

los algoritmos de búsqueda de rutas más cortas son fundamentales en el ámbito de la informática y la ingeniería para una variedad de aplicaciones, desde la planificación de rutas en sistemas de navegación hasta la optimización de redes de comunicación. Uno de los algoritmos más conocidos y utilizados para este propósito es el algoritmo de Dijkstra. presentar una implementación del algoritmo de Dijkstra en el contexto de un grafo ponderado, utilizando el lenguaje de programación Java. La implementación proporciona una herramienta útil para encontrar la ruta más corta entre dos puntos en un grafo y calcular el peso total de dicha ruta. Además, se incluyen funciones para visualizar la estructura del grafo, como la matriz de adyacencia y la lista de adyacencia, lo que facilita la comprensión y verificación de los resultados obtenidos.

2. Objetivos

2.1. General

- Implementar el algoritmo de Dijkstra en Java para encontrar la ruta más corta entre dos vértices en un grafo ponderado, proporcionando una herramienta eficaz y versátil para la resolución de problemas de optimización de rutas en diversos contextos informáticos y de ingeniería.

2.2. Especifico

2.2.1. Desarrollar una implementación eficiente del algoritmo de Dijkstra en Java, capaz de manejar grafos con un gran número de vértices y aristas.

2.2.2. Crear funciones adicionales para visualizar la estructura del grafo, incluyendo la matriz de adyacencia y la lista de adyacencia, para facilitar la comprensión y verificación de los resultados obtenidos.

2.2.3. Validar la implementación del algoritmo de Dijkstra mediante pruebas exhaustivas en diferentes escenarios y grafos de prueba, garantizando su precisión y fiabilidad en la determinación de rutas más cortas en grafos ponderados.

3. Desarrollo

3.1.1. Código copiado

Main

```
public class GrafoBusqueda {
   public static void main(String[] args) {
     Grafo grafo = new Grafo(9);

      grafo.agregarArista('A', 'B', 10);
      grafo.agregarArista('A', 'C', 15);
      grafo.agregarArista('B', 'D', 12);
      grafo.agregarArista('B', 'F', 15);
      grafo.agregarArista('C', 'E', 10);
      grafo.agregarArista('C', 'G', 8);
      grafo.agregarArista('D', 'E', 5);
      grafo.agregarArista('D', 'H', 7);
      grafo.agregarArista('E', 'F', 7);
      grafo.agregarArista('E', 'I', 8);
      grafo.agregarArista('F', 'I', 6);
      grafo.agregarArista('G', 'H', 12);
      grafo.agregarArista('H', 'I', 10);

      System.out.println("Lista de Adyacencia:");
      grafo.mostrarAristas();

      char origen = 'A';
      char destino = 'I';
      System.out.println("El recorrido es: " + grafo.dijkstra(origen, destino));
      System.out.println("El peso es: " + grafo.sumaPesosAristas(grafo.dijkstra(origen,
destino)));
      grafo.mostrarMatrizAdyacencia();
   }
}
```

Grafo

```
public class Grafo {
   private final int cantidadVertices;
   private final List<List<Arista>> listaAdyacencia;

   // Constructor para inicializar el grafo con el número de vértices
   public Grafo(int cantidadVertices) {
      this.cantidadVertices = cantidadVertices;
      this.listaAdyacencia = new ArrayList<>(cantidadVertices);
      for (int i = 0; i < cantidadVertices; i++) {
         this.listaAdyacencia.add(new ArrayList<>());
      }
   }

   // Método para agregar una arista entre dos vértices con un peso dado
   public void agregarArista(char origen, char destino, int peso) {
      this.listaAdyacencia.get(origen - 'A').add(new Arista(destino - 'A', peso));
   }
```

```
   // Algoritmo de Dijkstra para encontrar la ruta más corta desde un vértice de origen a
uno de destino
   public List<Character> dijkstra(char origen, char destino) {
      PriorityQueue<Nodo> colaPrioridad = new PriorityQueue<>(cantidadVertices,
Comparator.comparingInt(a -> a.peso));
      int[] distancias = new int[cantidadVertices];
      int[] previos = new int[cantidadVertices];
      Arrays.fill(distancias, Integer.MAX_VALUE);
      Arrays.fill(previos, -1);
      colaPrioridad.add(new Nodo(origen - 'A', 0));
      distancias[origen - 'A'] = 0;

      // Tabla para mostrar el cálculo del peso y la ruta
      System.out.println("Vertice\t| Peso Temporal\t| Peso Final\t| Ruta");
      System.out.println("-----------------------------------------------");
      while (!colaPrioridad.isEmpty()) {
         Nodo nodoActual = colaPrioridad.poll();
         int u = nodoActual.vertice;
         for (Arista arista : listaAdyacencia.get(u)) {
            int v = arista.destino;
            int peso = arista.peso;
            if (distancias[u] + peso < distancias[v]) {
               distancias[v] = distancias[u] + peso;
               previos[v] = u;
               colaPrioridad.add(new Nodo(v, distancias[v]));
            }
         }

         // Mostrar el cálculo del peso y la ruta en este paso
         System.out.print((char) (u + 'A') + "\t| " + distancias[u] + "\t\t| ");
         if (distancias[u] == Integer.MAX_VALUE) {
            System.out.print("INF\t\t| ");
         } else {
            System.out.print(distancias[u] + "\t\t| ");
         }
         List<Character> ruta = new ArrayList<>();
         for (int i = destino - 'A'; i != -1; i = previos[i]) {
            ruta.add((char) (i + 'A'));
         }
         Collections.reverse(ruta);
         System.out.println(ruta);
      }
      System.out.println("-----------------------------------------------");

      List<Character> rutaFinal = new ArrayList<>();
      for (int i = destino - 'A'; i != -1; i = previos[i]) {
         rutaFinal.add((char) (i + 'A'));
      }
      Collections.reverse(rutaFinal);
      return rutaFinal;
   }

   // Método para obtener la suma de los pesos de las aristas en una ruta dada
   public int sumaPesosAristas(List<Character> ruta) {
```

```
        int suma = 0;
        for (int i = 0; i < ruta.size() - 1; i++) {
            char origen = ruta.get(i);
            char destino = ruta.get(i + 1);
            int peso = obtenerPesoArista(origen, destino);
            suma += peso;
        }
        return suma;
    }

    // Método para mostrar la matriz de adyacencia del grafo
    public void mostrarMatrizAdyacencia() {
        System.out.println("Matriz de Adyacencia es ");
        for (int i = 0; i < cantidadVertices; i++) {
            for (int j = 0; j < cantidadVertices; j++) {
                int peso = obtenerPesoArista((char) ('A' + i), (char) ('A' + j));
                System.out.print((peso == Integer.MAX_VALUE ? "0" : peso) + " ");
            }
            System.out.println();
        }
    }

    // Método para obtener el peso de una arista entre dos vértices dados
    private int obtenerPesoArista(char origen, char destino) {
        for (Arista arista : listaAdyacencia.get(origen - 'A')) {
            if (arista.destino == destino - 'A') {
                return arista.peso;
            }
        }
        return Integer.MAX_VALUE;
    }

    // Método para mostrar la lista de adyacencia del grafo
    public void mostrarAristas() {
        System.out.println("Lista de adyacencia:");
        for (int i = 0; i < cantidadVertices; i++) {
            for (Arista arista : listaAdyacencia.get(i)) {
                System.out.println((char) ('A' + i) + " -> " + (char) ('A' + arista.destino) + " Peso: "
+ arista.peso);
            }
        }
    }

    // Clase interna que representa una arista en el grafo
    private static class Arista {
        private final int destino;
        private final int peso;

        public Arista(int destino, int peso) {
            this.destino = destino;
            this.peso = peso;
        }

        public int getDestino() {
```

```
        return destino;
    }

    public int getPeso() {
        return peso;
    }
}

// Clase interna que representa un nodo con su peso
private static class Nodo {
    private final int vertice;
    private final int peso;

    public Nodo(int vertice, int peso) {
        this.vertice = vertice;
        this.peso = peso;
    }
}}
```

Figura 1: Código copiado.

3.1.2. Código capturado y explicación.

El código proporciona una implementación del algoritmo de Dijkstra para encontrar la ruta más corta en un grafo ponderado.
Clase Grafo:
Esta clase representa el grafo y contiene métodos para agregar aristas, ejecutar el algoritmo de Dijkstra, obtener la suma de los pesos de las aristas en una ruta y mostrar la matriz de adyacencia del grafo.
Método agregarArista(char origen, char destino, int peso):

Este método agrega una arista al grafo entre dos vértices dados (origen y destino) con un peso especificado.
Método dijkstra(char origen, char destino):

Este método implementa el algoritmo de Dijkstra para encontrar la ruta más corta desde un vértice de origen hasta un vértice de destino en el grafo.
Utiliza una cola de prioridad para explorar los nodos de manera ordenada según sus pesos.
Calcula y actualiza las distancias mínimas desde el origen hasta cada vértice en el grafo.
Registra el camino más corto hacia cada vértice en un array de previos.
Muestra una tabla detallada del cálculo del peso y la ruta en cada paso del algoritmo.
Método sumaPesosAristas(List<Character> ruta):

Este método calcula la suma de los pesos de las aristas en una ruta dada.
Método mostrarMatrizAdyacencia():

Este método muestra la matriz de adyacencia del grafo, que representa los pesos de las aristas entre los vértices del grafo.
Método mostrarAristas():

Este método muestra la lista de adyacencia del grafo, que enumera las aristas salientes desde cada vértice.
Clase interna Arista:

Esta clase representa una arista en el grafo, con un destino y un peso.
Clase interna Nodo:

Esta clase representa un nodo en el grafo, con su índice y su peso.
Método main(String[] args):

Este método prueba la implementación del grafo y el algoritmo de Dijkstra creando un grafo de ejemplo, agregando aristas, ejecutando el algoritmo de Dijkstra desde un vértice de origen a un vértice de destino y mostrando la ruta más corta y su peso. También muestra la matriz de adyacencia del grafo y la lista de adyacencia para verificar la correcta construcción del grafo

```
public class GrafoBusqueda {
    public static void main(String[] args) {
        Grafo grafo = new Grafo( cantidadVertices: 9);

        grafo.agregarArista( origen: 'A',  destino: 'B',  peso: 10);
        grafo.agregarArista( origen: 'A',  destino: 'C',  peso: 15);
        grafo.agregarArista( origen: 'B',  destino: 'D',  peso: 12);
        grafo.agregarArista( origen: 'B',  destino: 'F',  peso: 15);
        grafo.agregarArista( origen: 'C',  destino: 'E',  peso: 10);
        grafo.agregarArista( origen: 'C',  destino: 'G',  peso: 8);
        grafo.agregarArista( origen: 'D',  destino: 'E',  peso: 5);
        grafo.agregarArista( origen: 'D',  destino: 'H',  peso: 7);
        grafo.agregarArista( origen: 'E',  destino: 'F',  peso: 7);
        grafo.agregarArista( origen: 'E',  destino: 'I',  peso: 8);
        grafo.agregarArista( origen: 'F',  destino: 'I',  peso: 6);
        grafo.agregarArista( origen: 'G',  destino: 'H',  peso: 12);
        grafo.agregarArista( origen: 'H',  destino: 'I',  peso: 10);

        System.out.println( x: "Lista de Adyacencia:");
        grafo.mostrarAristas();

        char origen = 'A';
        char destino = 'I';
        System.out.println("El recorrido es: " + grafo.dijkstra(origen, destino));
        System.out.println("El peso es: " + grafo.sumaPesosAristas( ruta: grafo.dijkstra(origen, destino)));
        grafo.mostrarMatrizAdyacencia();
    }
}
```

Figura 2: Explicación del código.

3.1.3. Ejecución

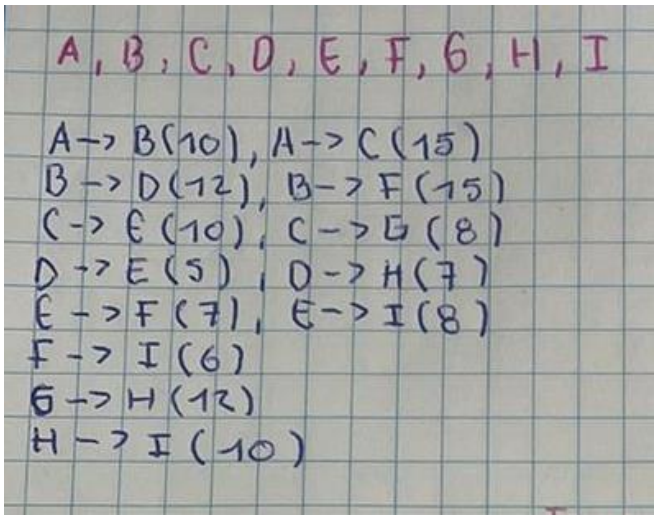
```
Lista de adyacencia:
A -> B Peso: 10
A -> C Peso: 15
B -> D Peso: 12
B -> F Peso: 15
C -> E Peso: 10
C -> G Peso: 8
D -> E Peso: 5
D -> H Peso: 7
E -> F Peso: 7
E -> I Peso: 8
F -> I Peso: 6
G -> H Peso: 12
H -> I Peso: 10
Vertice | Peso Temporal | Peso Final
-----------------------------------
A       | 0             | 0
B       | 10            | 10
C       | 15            | 15
D       | 22            | 22
G       | 23            | 23
F       | 25            | 25
E       | 25            | 25
H       | 29            | 29
I       | 31            | 31
-----------------------------------
Vertice | Peso Temporal | Peso Final
-----------------------------------
A       | 0             | 0
B       | 10            | 10
C       | 15            | 15
D       | 22            | 22
G       | 23            | 23
F       | 25            | 25
E       | 25            | 25
H       | 29            | 29
I       | 31            | 31
-----------------------------------
El recorrido es: [A, B, F, I]
```

```
El peso es: 31
Matriz de Adyacencia es
0 10 15 0 0 0 0 0 0
0 0 0 12 0 15 0 0 0
0 0 0 0 10 0 8 0 0
0 0 0 0 5 0 0 7 0
0 0 0 0 0 7 0 0 8
0 0 0 0 0 0 0 0 6
0 0 0 0 0 0 0 12 0
0 0 0 0 0 0 0 0 10
0 0 0 0 0 0 0 0 0
```

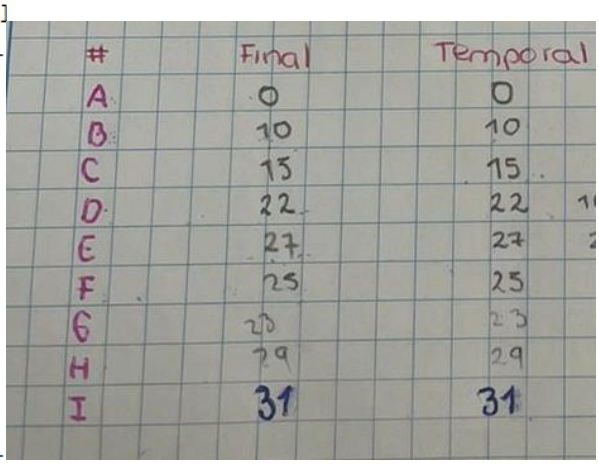
Figura 3: Ejecución del código.

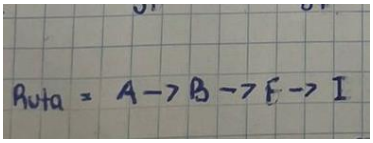
3.1.4. Ejecución de código en Hoja.

```
Lista de adyacencia:
A -> B Peso: 10
A -> C Peso: 15
B -> D Peso: 12
B -> F Peso: 15
C -> E Peso: 10
C -> G Peso: 8
D -> E Peso: 5
D -> H Peso: 7
E -> F Peso: 7
E -> I Peso: 8
F -> I Peso: 6
G -> H Peso: 12
H -> I Peso: 10
```




```
Vertice | Peso Temporal | Peso Final
-------------------------------------
A       | 0             | 0
B       | 10            | 10
C       | 15            | 15
D       | 22            | 22
G       | 23            | 23
F       | 25            | 25
E       | 25            | 25
H       | 29            | 29
I       | 31            | 31
-------------------------------------
```






```
El recorrido es: [A, B, F, I]
```

```
El peso es: 31
Matriz de Adyacencia es
0  10  15  0  0  0  0  0  0
0  0  0  12  0  15  0  0  0
0  0  0  0  10  0  8  0  0
0  0  0  0  5  0  0  7  0
0  0  0  0  0  7  0  0  8
0  0  0  0  0  0  0  0  6
0  0  0  0  0  0  0  12  0
0  0  0  0  0  0  0  0  10
0  0  0  0  0  0  0  0  0
```

Figura 4: Código comprado con prueba a mano.

- Evidencia del trabajo realizado a mano:

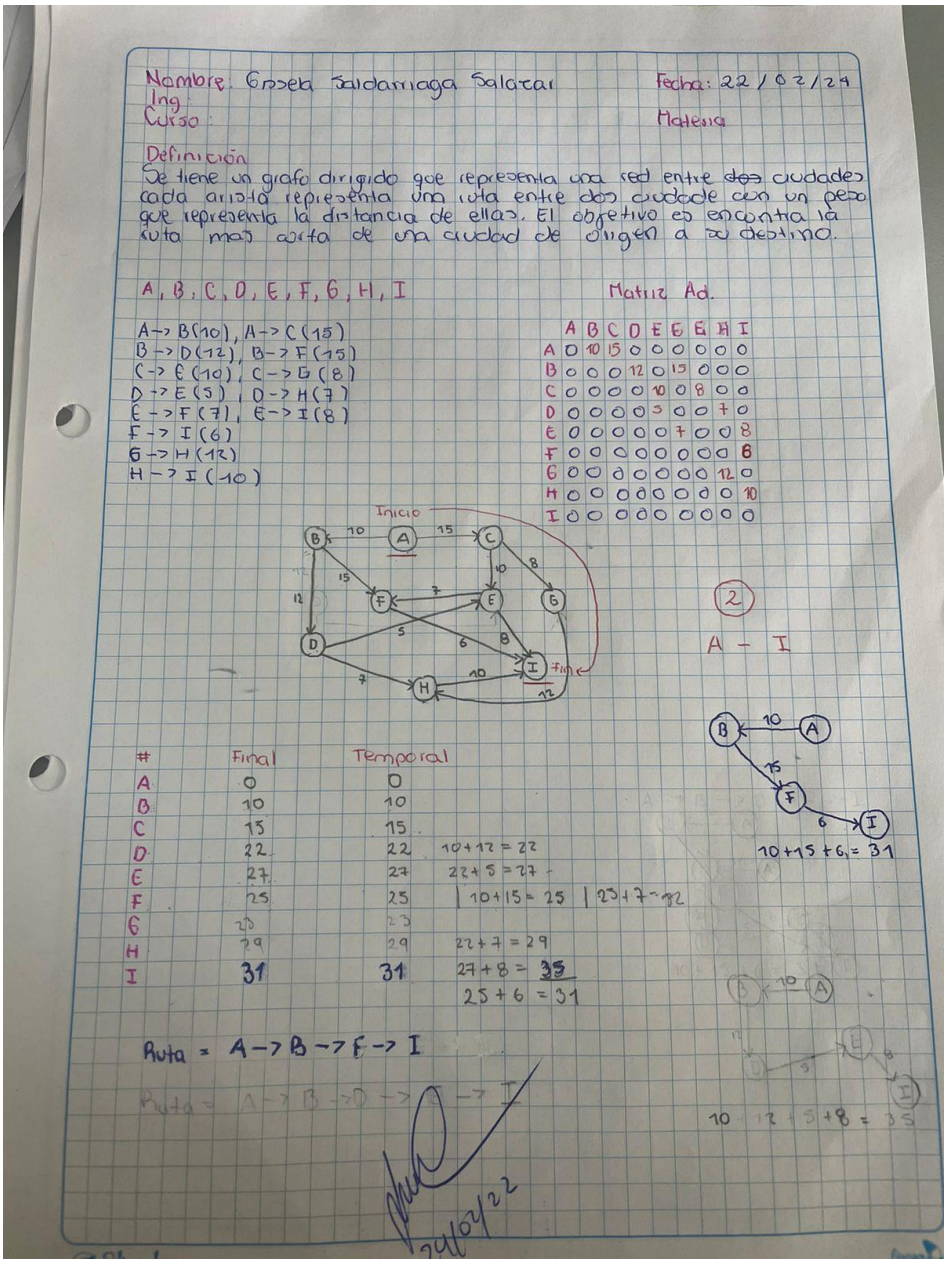


Figura 5: Trabajo realizado a mano.

4. Conclusiones.

- La implementación del algoritmo de Dijkstra en Java proporciona una herramienta robusta y eficiente para encontrar rutas más cortas en grafos ponderados, con aplicaciones prácticas en diversos campos de la informática y la ingeniería.
- La visualización de la estructura del grafo mediante la matriz y lista de adyacencia facilita la comprensión y verificación de los resultados obtenidos, permitiendo una mejor depuración y validación del algoritmo implementado.
- La validación exhaustiva del algoritmo de Dijkstra en diferentes escenarios y grafos de prueba garantiza su precisión y confiabilidad, lo que es fundamental para su aplicación en situaciones reales donde la exactitud de las rutas es crucial.

5. Recomendaciones

- Optimizar la implementación del algoritmo de Dijkstra para mejorar su eficiencia en grafos de gran tamaño, considerando técnicas de optimización y estructuras de datos más avanzadas para reducir el tiempo de ejecución y el consumo de recursos.
- Documentar adecuadamente el código fuente y proporcionar comentarios claros y concisos en cada parte del algoritmo para facilitar su comprensión y mantenimiento por parte de otros desarrolladores en el futuro.
- Explorar la posibilidad de integrar la implementación del algoritmo de Dijkstra en aplicaciones y sistemas existentes que requieran funcionalidades relacionadas con la búsqueda de rutas más cortas, aprovechando su versatilidad y adaptabilidad para resolver una amplia gama de problemas.

6. Bibliografía

GraphEverywhere, E. (2019, julio 1). *Qué son los grafos*. GraphEverywhere; Graph Everywhere SL. https://www.grapheverywhere.com/que-son-los-grafos/