UNIVERSIDAD DIEGO PORTALES

FACULTAD DE INGENIERÍA Y CIENCIAS

ESCUELA DE INFORMÁTICA Y TELECOMUNICACIONES

Estructura de Datos y Algoritmos

Informe de Laboratorio N°4

Profesor:

Alan Toro

Estudiante:

Gabriel González

Santiago, Chile

Julio de 2024

Índice

1. Introducción **1**

2. Resumen **1**

3. Implementación **2**
3.1. Grafo 2
3.2. Dijkstra 4
3.3. HashTable 7

4. Ejecución del programa **9**

5. Conclusión **10**

1. Introducción

En este laboratorio se abordará el estudio de las estructuras de datos conocidas como *Grafos*, centrándose en la implementación y aplicación del algoritmo de `Dijkstra` para la determinación del camino más corto entre dos nodos (pathfinding).

Además, se implementarán métodos relacionados con el `hashing` y una `HashTable` en base a los datos de un archivo CSV.

2. Resumen

Se trabajará con un grafo de canciones obtenidas de un archivo CSV, el cual contiene dos nodos a conectar, el costo de la conexión (arista) y la letra o 'nombre' de la arista. A partir de otro archivo CSV con pares de canciones, se utilizará el algoritmo de `Dijkstra` para formar un camino entre dicho par. Este camino se almacenará como un `String`, formado por las letras características de cada arista recorrida. Este `string` será procesado por una función de *hashing* personalizada que retornará un valor entero, el cual se utilizará como índice para insertar el par de canciones en una `HashTable`. La `HashTable` no admitirá repeticiones y una vez se inserten todos los pares compatibles, se finalizará el proceso, en esta implementación, las claves de la HashTable son `strings` representando los caminos calculados, y los valores son objetos de la clase `Par`, que contienen dos canciones.

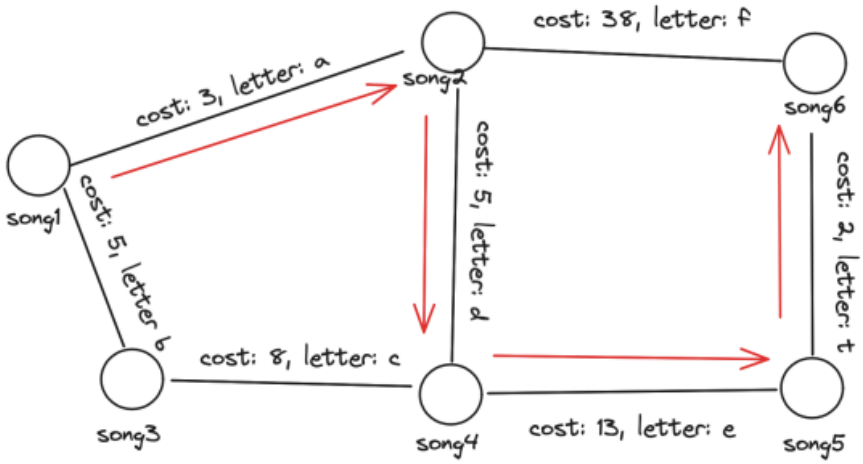

Figura 1: Ejemplo de `pathfinding`

3. Implementación

3.1. Grafo

A continuación se muestra y se explica parte por parte la implementación del grafo de canciones a utilizar.

```
public class Graph {
    public record Arista(String destino, int costoArista, char letra) {
    }

    private HashMap<String, ArrayList<Arista>> listaAdyacencia;

    public Graph(String rutaCSV) throws Exception {
        listaAdyacencia = new HashMap<>();

        BufferedReader br = new BufferedReader(new FileReader(rutaCSV));
        String lineaActual;

        // Leer el archivo CSV línea por línea
        while ((lineaActual = br.readLine()) != null) {
            String[] datos = lineaActual.split(",");
            String song1 = datos[0].trim();
            String song2 = datos[1].trim();
            int costoArista = Integer.parseInt(datos[2].trim());
            char letra = datos[3].trim().charAt(0);
            agregarConexion(song1, song2, costoArista, letra);
        }

        br.close();
    }

    // Método para agregar una conexión entre dos vértices (canciones)
    public void agregarConexion(String origen, String destino, int costoArista, char letra) {
        listaAdyacencia.computeIfAbsent(...));
        listaAdyacencia.computeIfAbsent(...));
    }

    // Método para obtener las conexiones de un vértice dado
    public ArrayList<Arista> obtenerConexiones(String vertice) {
        return listaAdyacencia.getOrDefault(vertice, new ArrayList<>());
    }

    // Método para imprimir la representación del grafo
    public void imprimirGrafo() {
        System.out.println("Grafo:");
        for (String vertice : listaAdyacencia.keySet()) {
            System.out.print(vertice + " -> ");
            ArrayList<Arista> conexiones = listaAdyacencia.get(vertice);
            for (Arista conexion : conexiones) {
                System.out.print(conexion.destino + " ");
            }
            System.out.println();
        }
    }
}
```

`Clase graph`
Se compone de una lista de adyacencia, la cual es un HashMap compuesto de un String y una lista de aristas.

`Métodos`

- `public record Arista`: Es la clase encargada de almacenar una arista (conexión) entre dos nodos de un grafo, esta, en una implementación común se compone de su destino y costo, pero en la implementación del laboratorio también incluye la letra que distingue a la arista.
- `Constructor de graph`: Lee el CSV con los datos y los separa, luego, llama a la funcion `agregarConexion`, la cual se encarga de unir ambos nodos (bidireccionalmente) y añadirlos a la lista de adyacencia en caso de no existir una entrada coincidente.
- `agregarConexion`: Conecta dos nodos bidireccionalmente, utiliza la función `computeIfAbsent`, la cual si es que existe la llave, retorna la arista, si no, crea la nueva arista en una dirección, para conectar por ambos lados se llama dos veces la función, invirtiendo el inicio y el destino.
- `obtenerConexiones`: obtiene los nodos adyacentes al nodo entregado.
- `imprimirGrafo`: Imprime el grafo

`Output de ejemplo - Grafo de canciones`

```
ytglqvwzdumijeosnbcr -> hcpdgbtajqfvkoirxeum fyxsoqmnauvcbjdhtlze zmbcapxoygehrlukidfj
dnerjzmukcysavxwhifg -> hcpdgbtajqfvkoirxeum xvwgipbrjktolsmuheac reoibxnsumftcypqzklj ...
xtfghanouebpdsvyrcqw -> szagikemqvdyxnwftpro kniwlemtfbxcdsgujpyr
nbjsitzokrqflyeuhawv -> dzxljiupchgvenofkbst sdnwvjfrqekpiotuxcbz fxeuhjbvklzwimcqgrps ...
mtucxfvpalkqenodwjyg -> dhwfxyobreiclsmpzunj wdrngkmqevoxclbhjpiy gnwlxcsquybpzmkifhto ...
dnepxzagfyritqbcwouj -> xclpkzthfbqywjignues kzojsxlvwmcqenihufpr syjrgfxobntehdwvmckz
mptgoskjqhfrldacivew -> cfjweulmbrsodxkivnth jpmgbxealhncwqkysduz vhlsnzrdfbipgwotxuqm
tgpqbjdsuokhynmwflvc -> bwfdepxyajzluvqhgosi ivtxyceuakrbmnhsqpfw nvlpeqmwytzkabfsdhcj ...
zcfjiaylktvgqhwxubde -> zjatgcsbyxekfmhudnop nmokbxvgfdpwqretscyl dizuowhaelvrxycqpskf ...
lkvabwjemtsipnfdzyxq -> dizuowhaelvrxycqpskf yxojrtdskneubaqzilgv tmwhgypuvzdeqikfrocb ...
flyrtxmaeskvjindohzg -> kosznadvcxbljurhmyep amhsbytezdlnovigxcqf werfzylotxamhvcibjqg ...
hbsjgkmcpiorxeuaqvwz -> whjckxfmzyodgeiparsq wosxzkdrqcgjvaepbilt vcflibenuxtkpszqjwrd
```

3.2. Dijkstra

A continuación se muestra y se explica parte por parte la implementación del Algoritmo `Dijkstra`.

`Clase Node (Clase interna)`
Se compone de un nombre(String), una variable de costo del camino(int) y un camino(lista de nombres).

`Métodos`

- `public int compareTo`: Implementa la interfaz `Comparable<Node>` para comparar nodos basándose en su costo.

`Código:`

```
public class Dijkstra {
    private class Node implements Comparable<Node> {
        String nombre;
        int costo;
        ArrayList<String> camino;

        public Node(String nombre, int costo, ArrayList<String> camino) {
            this.nombre = nombre;
            this.costo = costo;
            this.camino = camino;
        }

        public int compareTo(Node other) {
            return Integer.compare(this.costo, other.costo);
        }
    }
```

`Clase Dijkstra`

Continuando en la clase principal *Dijkstra*, esta usa principalmente el método **caminoMasCorto**, los atributos están definidos dentro del método aunque también se pueden definir dentro de la clase, estos atributos son los siguientes:

- `colaPrioridad`: Una `PriorityQueue` que almacena nodos ordenados por el costo acumulado hasta el nodo actual.
- `distancias`: Un `HashMap` que mapea cada nodo a su costo mínimo conocido desde el nodo de inicio.
- `anteriores`: Un `HashMap` que mapea cada nodo a su nodo previo en el camino más corto conocido desde el nodo de inicio.

`Inicialización de la clase`
`Código:`

```
public class Dijkstra {

    ... // Código de la clase Node

    public Dijkstra() {
    }

... // Resto del código más abajo

```

`Métodos de Dijkstra:`

`caminoMasCorto`: Este método recibe el grafo y los nombres de los nodos de inicio y destino, opera de la siguiente forma

1. Se inicializa la `colaPrioridad` con el nodo inicial y se establece su distancia como 0.
2. Mientras haya nodos en la cola de prioridad (while):
   - Se extrae el nodo con el costo mínimo actual.
   - Si este nodo es el nodo de destino, se reconstruye y devuelve el camino más corto utilizando el método `construirPalabra`.
   - Para cada conexión (`arista`) del nodo actual:
     - Se calcula el nuevo costo acumulado sumando el costo de la arista actual al costo acumulado del nodo actual.
     - Si este nuevo costo es menor que el costo mínimo conocido para el nodo destino de la arista:
       - Se actualiza el costo mínimo conocido y se guarda el nodo previo en `anteriores`.
       - Se crea un nuevo camino actualizado agregando el nodo destino de la arista al camino actual.
       - Se agrega este nuevo nodo a la cola de prioridad para explorar más tarde.

`Código:`

```
26      public String caminoMasCorto(Graph graph, String comienzo, String destino) {
27          PriorityQueue<Node> colaPrioridad = new PriorityQueue<>();
28
29          HashMap<String, Integer> distancias = new HashMap<>();
30          HashMap<String, String> anteriores = new HashMap<>();
31
32          colaPrioridad.add(new Node(comienzo, 0, new ArrayList<>()));
33          distancias.put(comienzo, 0);
34
35          while (!colaPrioridad.isEmpty()) {
36              Node current = colaPrioridad.poll();
37
38              if (current.nombre.equals(destino)) {
39                  return construirPalabra(graph, anteriores, comienzo, destino);
40              }
41
42              if (current.costo > distancias.getOrDefault(current.nombre, Integer.MAX_VALUE)) {
43                  continue;
44              }
45
46              for (var arista : graph.obtenerConexiones(current.nombre)) {
47                  String destinoArista = arista.destino();
48                  int costoArista = arista.costoArista();
49                  int nuevoCosto = current.costo + costoArista;
50                  if (nuevoCosto < distancias.getOrDefault(destinoArista, Integer.MAX_VALUE)) {
51                      distancias.put(destinoArista, nuevoCosto);
52                      anteriores.put(destinoArista, current.nombre);
53                      ArrayList<String> nuevoCamino = new ArrayList<>(current.camino);
54                      nuevoCamino.add(destinoArista);
55                      colaPrioridad.add(new Node(destinoArista, nuevoCosto, nuevoCamino));
56                  }
57              }
58          }
59          return null;
60      }
```

`construirPalabra`: Este método recibe el grafo, el HashMap de nodos previos en el camino más corto desde el nodo inicial hasta cada nodo., el nodo de inicio y el nodo de destino, funciona de la siguiente manera

- Utiliza `anteriores` para reconstruir el camino más corto desde el nodo destino hasta el nodo inicial.
- Construye una cadena (`palabra`) que representa el camino en función de las letras de las aristas del grafo.

`Código:`

```
61      private String construirPalabra(...) {
62          StringBuilder palabra = new StringBuilder();
63          String actual = destino;
64
65          while (!actual.equals(comienzo)) {
66              String anterior = anteriores.get(actual);
67              for (var arista : graph.obtenerConexiones(anterior)) {
68                  if (arista.destino().equals(actual)) {
69                      palabra.insert(0, arista.letra());
70                      break;
71                  }
72              }
73              actual = anterior;
74          }
75
76          return palabra.toString();
77      }
78  }
```

3.3. HashTable

`Clase Par` Consiste simplemente en un par de nombres de canciones, también puede implementarse como record class

`Inicialización de HashTable`

La clase HashTable cuenta con los siguientes atributos:

- `SIZE`: Tamaño de la tabla hash, establecido en 1,000,000.
- `Potencia`: Valor constante utilizado como base para el cálculo del hash.
- `tabla`: Arreglo de objetos tipo `Par` utilizado como tabla hash para almacenar pares de canciones.

`Código:`

```
public class HashTable {
    private static final int SIZE = 1_000_000;
    private static final int Potencia = 31;
    Par[] tabla;

    private static class Par {
        String song1;
        String song2;

        Par(String song1, String song2) {
            this.song1 = song1;
            this.song2 = song2;
        }
    }

    public HashTable() {
        tabla = new Par[SIZE];
    }

```

`Método hash:` Este método recibe un String y calcula un valor de hash en base a la siguiente función:

$$hash(s) = s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} m$$

```
    private int hash(String s) {
        if(s == null) return 0;

        long valorHash = 0;
        long potenciaELPRIMO = 1;

        for(char c: s.toCharArray()){
            valorHash = (valorHash + (c - 'a' + 1) * potenciaELPRIMO) % SIZE;
            potenciaELPRIMO = (potenciaELPRIMO * Potencia) % SIZE;
        }

        return (int) valorHash;
    }
```

`Método insert:` Este método recibe un par de canciones y un camino en forma de String (el retornado por Dijkstra), llama a la función *hash* y obtiene el índice, si es menor a 1, lo convierte a positivo, tras esto intenta insertar en el índice, si este está ocupado, imprime un error y no inserta en la posición

```
33      public void insert(String song1, String song2, String path) {
34          int indice = hash(path);
35          if(indice < 0) {
36              indice = -indice;
37          }
38          if(tabla[indice] == null) {
39              tabla[indice] = new Par(song1, song2);
40          } else {
41              System.out.println("Colisión, no se insertará en el índice: " + indice);
42              System.out.println("Canciones: " + song1 + " y " + song2 + ". ");
43          }
44      }
45
```

`Metodo printTable:` Imprime la tabla hash con un bucle for.

```
46      public void printTable() {
47          for(int i = 0; i < SIZE; i++) {
48              if(tabla[i] != null) {
49                  System.out.println("Index " + i + ": (" + tabla[i].song1 + ", " + tabla[i].song2 + ")");
50              }
51  }   }   }
```

4. Ejecución del programa

La función main es la encargada de realizar la unión de todas las clases y métodos implementados, esta hace lo siguiente (en orden):

1. Crea un objeto `Graph` cargando los datos desde el archivo `graph.csv`.
2. Crea una tabla hash (`HashTable`) y un objeto de la clase `Dijkstra`.
3. Lee pares de canciones desde el archivo `pairs.csv`.
4. Para cada par de canciones, calcula el camino más corto usando el algoritmo de Dijkstra.
5. Si se encuentra un camino, lo inserta en la tabla hash.
6. Si no se encuentra un camino, imprime un mensaje de error.
7. Imprime la representación del grafo y el contenido de la tabla hash.

```
public class Main {
    public static void main(String[] args) throws Exception {
        Graph graph = new Graph("D:\\Code\\...\\graph.csv");

        HashTable hashTable = new HashTable();
        Dijkstra dj = new Dijkstra();

        try(BufferedReader br = new BufferedReader(new FileReader("D:\\Code\\...\\pairs.csv"))){
            String lineaActual;
            while((lineaActual = br.readLine()) != null){
                String[] partes = lineaActual.split(",");
                String song1 = partes[0];
                String song2 = partes[1];
                String camino = dj.caminoMasCorto(graph, song1, song2);
                if(camino != null) {
                    hashTable.insert(song1, song2, camino);
                } else {
                    System.out.println("No se encontró camino");
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }

        graph.imprimirGrafo();
        hashTable.printTable();
    }
}
```

5. Conclusión

Tras probar la implementación completa del programa se obtiene el grafo completo y la tabla hash, como medida de prueba, el archivo CSV de pairs está hecho de forma que sólo deben haber tres colisiones, que es el caso de el programa implementado, las colisiones se obtienen del siguiente output:

```
Colisión, no se insertará en el índice: 496326
Canciones: pzdakwexvufibqnltohy y evlkczhbarxdfuwtmnjy.
Colisión, no se insertará en el índice: 571
Canciones: rucvjqoghkiytabmznew y gvtdbmuljyqhanpxfeiz.
Colisión, no se insertará en el índice: 932713
Canciones: pnudrcasgzqvwebfmjih y jhfdzxcgvslwpqtokeua.
```

Con esto, se da por finalizada la implementación y se considera exitosa, el programa compila, carga el grafo, carga los pares, crea el String con el camino más corto, convierte el String a Int mediante la función de hash y lo usa como índice para insertar en la HashTable, cumpliendo con todos los procesos detallados en la guía, con este Laboratorio se logró obtener un mejor conocimiento de la estructura de Grafos y HashTable, también se aprendió a implementar un Algoritmo de Dijkstra y de Hashing personalizado.