**Universidad Nacional de San Agustín de Arequipa**

Facultad de Ingeniería de Producción y Servicios

Escuela Profesional de Ingeniería de Sistemas


# INFORME DE LABORATORIO 08

## Estructura de datos y Algoritmos

Alumnos:

- Payehuanca Riquelme Jhastyn Jefferson
- Zapata Butron Reyser Julio

Docente:
Quispe Vergaray Karen Melissa

28 de diciembre 2023
Arequipa - Perú

# Informe Laboratorio 08

## Tema: Grafos

## 1. URL de Repositorio Github

**Link del Repositorio en Github:** `https://github.com/ReyserLyn/Eda_lab08.git`

## 2. Ejercicios designados

1. Crear un repositorio en GitHub, donde incluyan la resolucion de los ejercicios propuestos y el informe.
2. Implementar el cogido de Grafo cuya representacion sea realizada mediante LISTA DE ADYACENCIA. (3 puntos)
3. Implementar BSF, DFS y Dijkstra con sus respectivos casos de prueba. (5 puntos)
4. Solucionar el siguiente ejercicio: (5 puntos)
   El grafo de palabras se define de la siguiente manera: cada vértice es una palabra en el idioma Inglés y dos palabras son adyacentes si difieren exactamente en una posición. Por ejemplo, las cords y los corps son adyacentes, mientras que los corps y crops no lo son.
   - Dibuje el grafo definido por las siguientes palabras: words cords corps coops crops drops drips grips gripe grape graph.
   - Mostrar la lista de adyacencia del grafo.
5. Realizar un metodo en la clase Grafo. Este metodo permitira saber si un grafo esta incluido en otro. Los parametros de entrada son 2 grafos y la salida del metodo es true si hay inclusion y false el caso contrario. (4 puntos)

## 3. Solución

### 3.1. Clase GraphLink

Este es la clase más importante y más compleja del trabajo, puesto que responde correctamente a los ejercicios designados.

Listing 1: GraphLink.java

```
1
2  import java.util.ArrayList;
3  import java.util.Comparator;
4  import java.util.HashMap;
5  import java.util.List;
6  import java.util.Map;
7  import java.util.PriorityQueue;
8
9  public class GraphLink<E extends Comparable<E>> {
10
11     protected ListLinked<Vertex<E>> listVertex;
```

```
12
13      public GraphLink() {
14          this.listVertex = new ListLinked<>();
15      }
16
17      public String insertVertex(E data) {
18          Vertex<E> v = new Vertex<>(data);
19          if (listVertex.search(v)) {
20              return "El vrtice con el dato " + data + " ya fue insertado";
21          } else {
22              listVertex.insertarOrdenado(v);
23              return "Vrtice agregado con xito";
24          }
25      }
26
27      public void insertEdge(E dataOri, E dataDes) {
28          Vertex<E> vOri = listVertex.searchData(new Vertex<>(dataOri));
29          Vertex<E> vDes = listVertex.searchData(new Vertex<>(dataDes));
30
31          if (vOri == null || vDes == null) {
32              throw new IllegalArgumentException("Los vrtices " + dataOri + " o " + dataDes + "
                    no existen");
33          }
34
35          Edge<E> e = new Edge<>(vDes);
36          if (vOri.listAdj.search(e)) {
37              throw new IllegalStateException("La arista (" + dataOri + "," + dataDes + ") ya
                    fue insertada");
38          }
39
40          vOri.listAdj.insertFirst(e);
41          vDes.listAdj.insertFirst(new Edge<>(vOri));
42      }
43
44      public void insertEdge(E dataOri, int weight, E dataDes) {
45          Vertex<E> vOri = listVertex.searchData(new Vertex<>(dataOri));
46          Vertex<E> vDes = listVertex.searchData(new Vertex<>(dataDes));
47
48          if (vOri == null || vDes == null) {
49              throw new IllegalArgumentException("Los vrtices " + dataOri + " o " + dataDes + "
                    no existen");
50          }
51
52          Edge<E> e = new Edge<>(vDes, weight);
53          if (vOri.listAdj.search(e) || vDes.listAdj.search(e)) {
54              throw new IllegalStateException("La arista (" + dataOri + "," + dataDes + ") ya
                    fue insertada");
55          }
56
57          vOri.listAdj.insertFirst(e);
58      }
59
60      public void removeEdge(E dataOri, E dataDes) {
61          Vertex<E> vOri = listVertex.searchData(new Vertex<>(dataOri));
62          Vertex<E> vDes = listVertex.searchData(new Vertex<>(dataDes));
63
```

```
64          if (vOri == null || vDes == null) {
65              throw new IllegalArgumentException("Los vrtices " + dataOri + " o " + dataDes + "
                    no existen");
66          }
67
68          Edge<E> e = new Edge<>(vDes);
69          if (vOri.listAdj.search(e)) {
70              vOri.listAdj.remove(e);
71              vDes.listAdj.remove(new Edge<>(vOri));
72          }
73      }
74
75      public void removeEdge(E dataOri, int weight, E dataDes) {
76          Vertex<E> vOri = listVertex.searchData(new Vertex<>(dataOri));
77          Vertex<E> vDes = listVertex.searchData(new Vertex<>(dataDes));
78
79          if (vOri == null || vDes == null) {
80              throw new IllegalArgumentException("Los vrtices " + dataOri + " o " + dataDes + "
                    no existen");
81          }
82
83          Edge<E> e = new Edge<>(vDes, weight);
84          if (vOri.listAdj.search(e)) {
85              vOri.listAdj.remove(e);
86              // Comentario: No eliminamos la arista inversa para grafos no dirigidos
87          }
88      }
89
90      public String removeVertex(E x) {
91          Vertex<E> vertex = listVertex.searchData(new Vertex<>(x));
92          if (vertex == null) {
93              return "El vrtice con el dato " + x + " no existe en el grafo";
94          }
95
96          Node<Vertex<E>> vAux = listVertex.getHead();
97          Edge<E> e = new Edge<>(vertex);
98          while (vAux != null) {
99              vAux.getData().listAdj.remove(e);
100             vAux = vAux.getNext();
101         }
102
103         listVertex.remove(vertex);
104         return "Vrtice eliminado con xito";
105     }
106
107     public boolean searchEdge(E dataOri, E dataDes) {
108         Vertex<E> vOri = listVertex.searchData(new Vertex<>(dataOri));
109         Vertex<E> vDes = listVertex.searchData(new Vertex<>(dataDes));
110
111         if (vOri == null || vDes == null) {
112             throw new IllegalArgumentException("Los vrtices " + dataOri + " o " + dataDes + "
                    no existen");
113         }
114
115         Edge<E> searchEdge = new Edge<>(vDes);
116         return vOri.listAdj.search(searchEdge);
```

```
117     }
118
119     public boolean searchEdge(E dataOri, int weight, E dataDes) {
120         Vertex<E> vOri = listVertex.searchData(new Vertex<>(dataOri));
121         Vertex<E> vDes = listVertex.searchData(new Vertex<>(dataDes));
122
123         if (vOri == null || vDes == null) {
124             throw new IllegalArgumentException("Los vrtices " + dataOri + " o " + dataDes + "
                    no existen");
125         }
126
127         Edge<E> searchEdge = new Edge<>(vDes, weight);
128         return vOri.listAdj.search(searchEdge);
129     }
130
131     public boolean searchVertex(E data) {
132         Vertex<E> vertex = listVertex.searchData(new Vertex<>(data));
133         return vertex != null;
134     }
135
136     public Vertex<E> buscarElemento(E data) {
137         return listVertex.searchData(new Vertex<>(data));
138     }
139
140     public ListLinked<Vertex<E>> getListVertex() {
141         return listVertex;
142     }
143
144     public void bfs(E startData) {
145         Vertex<E> startVertex = listVertex.searchData(new Vertex<>(startData));
146
147         if (startVertex == null) {
148             throw new IllegalArgumentException("El vrtice con el dato " + startData + " no
                    existe en el grafo");
149         }
150
151         SimpleQueue<Vertex<E>> queue = new SimpleQueue<>();
152         queue.enqueue(startVertex);
153         startVertex.visited = true;
154
155         while (!queue.isEmpty()) {
156             Vertex<E> currentVertex = queue.dequeue();
157             System.out.print(currentVertex.getData() + " ");
158
159             List<Edge<E>> adjList = new ArrayList<>(currentVertex.listAdj.length());
160             for (Node<Edge<E>> edgeNode : currentVertex.listAdj.getIterable()) {
161                 adjList.add(edgeNode.getData());
162             }
163
164             // Recorrer la lista de adyacencia en orden inverso
165             for (int i = adjList.size() - 1; i >= 0; i--) {
166                 Edge<E> edge = adjList.get(i);
167                 Vertex<E> neighbor = edge.refdest;
168                 if (!neighbor.visited) {
169                     queue.enqueue(neighbor);
170                     neighbor.visited = true;
```

```
171                 }
172             }
173         }
174
175         resetVisited();
176         System.out.println();
177     }
178
179     private void resetVisited() {
180         for (Node<Vertex<E>> vertexNode : listVertex.getIterable()) {
181             vertexNode.getData().visited = false;
182         }
183     }
184
185     public void dfs(E startData) {
186         Vertex<E> startVertex = listVertex.searchData(new Vertex<>(startData));
187
188         if (startVertex == null) {
189             throw new IllegalArgumentException("El vrtice con el dato " + startData + " no
                    existe en el grafo");
190         }
191
192         SimpleStack<Vertex<E>> stack = new SimpleStack<>();
193         stack.push(startVertex);
194         startVertex.visited = true;
195
196         while (!stack.isEmpty()) {
197             Vertex<E> currentVertex = stack.pop();
198             System.out.print(currentVertex.getData() + " ");
199
200             for (Node<Edge<E>> edgeNode : currentVertex.listAdj.getIterable()) {
201                 Edge<E> edge = edgeNode.getData();
202                 Vertex<E> neighbor = edge.refdest;
203                 if (!neighbor.visited) {
204                     stack.push(neighbor);
205                     neighbor.visited = true;
206                 }
207             }
208         }
209
210         resetVisited();
211         System.out.println();
212     }
213
214     public int dijkstra(E startData, E endData) {
215         Vertex<E> startVertex = listVertex.searchData(new Vertex<>(startData));
216         Vertex<E> endVertex = listVertex.searchData(new Vertex<>(endData));
217
218         if (startVertex == null || endVertex == null) {
219             throw new IllegalArgumentException("Los vrtices " + startData + " o " + endData +
                    " no existen");
220         }
221
222         Map<Vertex<E>, Integer> distances = new HashMap<>();
223
224         for (Node<Vertex<E>> vertexNode : listVertex.getIterable()) {
```

```
225             distances.put(vertexNode.getData(), Integer.MAX_VALUE);
226         }
227         distances.put(startVertex, 0);
228 
229         PriorityQueue<Vertex<E>> priorityQueue = new
                PriorityQueue<>(Comparator.comparingInt(distances::get));
230         priorityQueue.add(startVertex);
231 
232         while (!priorityQueue.isEmpty()) {
233             Vertex<E> currentVertex = priorityQueue.poll();
234 
235             if (currentVertex.equals(endVertex)) {
236                 break; // Salir si ya hemos alcanzado el vrtice de destino
237             }
238 
239             for (Node<Edge<E>> edgeNode : currentVertex.listAdj.getIterable()) {
240                 Edge<E> edge = edgeNode.getData();
241                 Vertex<E> neighbor = edge.refdest;
242                 int newDistance = distances.get(currentVertex) + edge.getWeight();
243 
244                 if (newDistance < distances.get(neighbor)) {
245                     distances.put(neighbor, newDistance);
246                     priorityQueue.add(neighbor);
247                 }
248             }
249         }
250 
251         return distances.get(endVertex);
252     }
253 
254     public boolean isIncludedIn(GraphLink<E> otherGraph) {
255         ListLinked<Vertex<E>> thisVertices = this.getListVertex();
256 
257         for (Node<Vertex<E>> thisVertexNode : thisVertices.getIterable()) {
258             Vertex<E> thisVertex = thisVertexNode.getData();
259 
260             if (!otherGraph.searchVertex(thisVertex.getData())) {
261                 return false;
262             }
263 
264             List<Edge<E>> thisEdges = thisVertex.listAdj.toList();
265             List<Edge<E>> otherEdges =
                    otherGraph.buscarElemento(thisVertex.getData()).listAdj.toList();
266 
267             for (Edge<E> thisEdge : thisEdges) {
268                 if (!otherEdges.contains(thisEdge)) {
269                     return false;
270                 }
271             }
272         }
273 
274         return true;
275     }
276 
277     @Override
278     public String toString() {
```

```
279         return listVertex.toString();
280     }
281 }
```

- `insertVertex(E data): String`
  Inserta un nuevo vértice con el dato `data` en el grafo. Retorna un mensaje indicando si la operación fue exitosa.
- `insertEdge(E dataOri, E dataDes): void`
  Inserta una arista entre los vértices con datos `dataOri` y `dataDes`. Lanza una excepción si alguno de los vértices no existe.
- `insertEdge(E dataOri, int weight, E dataDes): void`
  Inserta una arista ponderada entre los vértices con datos `dataOri` y `dataDes` con el peso especificado. Lanza una excepción si alguno de los vértices no existe.
- `removeEdge(E dataOri, E dataDes): void`
  Elimina la arista entre los vértices con datos `dataOri` y `dataDes`. Lanza una excepción si alguno de los vértices no existe.
- `removeEdge(E dataOri, int weight, E dataDes): void`
  Elimina la arista ponderada entre los vértices con datos `dataOri` y `dataDes` con el peso especificado. Lanza una excepción si alguno de los vértices no existe.
- `removeVertex(E x): String`
  Elimina el vértice con el dato `x` y todas las aristas asociadas. Retorna un mensaje indicando si la operación fue exitosa.
- `searchEdge(E dataOri, E dataDes): boolean`
  Verifica si existe una arista entre los vértices con datos `dataOri` y `dataDes`. Lanza una excepción si alguno de los vértices no existe.
- `searchEdge(E dataOri, int weight, E dataDes): boolean`
  Verifica si existe una arista ponderada entre los vértices con datos `dataOri` y `dataDes` con el peso especificado. Lanza una excepción si alguno de los vértices no existe.
- `searchVertex(E data): boolean`
  Verifica si existe un vértice con el dato `data` en el grafo.
- `buscarElemento(E data): Vertex<E>`
  Busca y retorna el vértice con el dato `data` en el grafo.
- `getListVertex(): ListLinked<Vertex<E`
  Retorna la lista de vértices del grafo.
- `bfs(E startData): void`
  Realiza un recorrido en amplitud (BFS) desde el vértice con dato `startData` e imprime los vértices visitados.
- `dfs(E startData): void`
  Realiza un recorrido en profundidad (DFS) desde el vértice con dato `startData` e imprime los vértices visitados.
- `dijkstra(E startData, E endData): int`
  Calcula la distancia mínima entre los vértices con datos `startData` y `endData` utilizando el algoritmo de Dijkstra.

- `isIncludedIn(GraphLink<E>otherGraph): boolean`
  Verifica si el grafo actual está incluido en otro grafo (`otherGraph`).
- `toString(): String`
  Retorna una representación en cadena del grafo.

## 3.2. Clase Main

Este es la clase en el que se hace un test de las clases realizadas y se encarga de responder los ejercicios designados, cumpliendo con lo que se pide y mostrando todo lo necesario.

Listing 2: Main.java

```
1
2  public class Main {
3
4      public static void main(String[] args) {
5          // Creamos un Grafo NO Dirigido
6          GraphLink<String> undirectedGraph = createUndirectedGraph();
7
8          System.out.println("Grafo No Dirigido:");
9          System.out.println(undirectedGraph);
10
11         System.out.println("Recorrido BFS desde el vrtice A:"); // BFS
12         undirectedGraph.bfs("A");
13
14         System.out.println("Recorrido DFS desde el vrtice A:"); // DFS
15         undirectedGraph.dfs("A");
16
17         System.out.println();
18
19         // Creamos un Grago Dirigido con Peso
20         GraphLink<String> weightedDirectedGraph = createWeightedDirectedGraph();
21
22         System.out.println("Grafo Dirigido con Peso:");
23         System.out.println(weightedDirectedGraph);
24
25         System.out.println("Recorrido BFS desde el vrtice A:"); // BFS
26         weightedDirectedGraph.bfs("A");
27
28         System.out.println("Recorrido DFS desde el vrtice A:"); // DFS
29         weightedDirectedGraph.dfs("A");
30
31         // DIJKSTRA
32         System.out.println("\nLa distancia mnima entre A y F es: " +
               weightedDirectedGraph.dijkstra("A", "F"));
33         System.out.println();
34
35         System.out.println("Agregamos un peso de 2 de B a F");
36         weightedDirectedGraph.insertEdge("B", 2, "F");
37         System.out.println("La nueva distancia mnima entre A y F es: " +
               weightedDirectedGraph.dijkstra("A", "F"));
38         System.out.println();
39
40         // Ejercicio 4, grafo con palabras - a) Creamos el grafo
41         GraphLink<String> wordGraph = createWordGraph();
42
```

```
43          // Ejercicio 4 - b) Mostramos la lista de adyacencia
44          System.out.println("Grafo de Palabras:");
45          System.out.println(wordGraph);
46
47          // Ejercicio 5 - Probamos la inclusion de un grafo dentro de otro
48          GraphLink<String> graph1 = createGraph1();
49          GraphLink<String> graph2 = createGraph2();
50
51          // Mostramos los grafos
52          System.out.println("Grafo 1:");
53          System.out.println(graph1);
54
55          System.out.println("Grafo 2:");
56          System.out.println(graph2);
57
58          // Verificar si graph1 est incluido en graph2
59          boolean isIncluded = graph1.isIncludedIn(graph2);
60
61          // Mostrar el resultado de la inclusin
62          System.out.println("Grafo 1 est incluido en Grafo 2? " + isIncluded);
63
64          isIncluded = graph2.isIncludedIn(graph1);
65          System.out.println("Grafo 2 est incluido en Grafo 1? " + isIncluded);
66      }
67
68      // Crear grafo no dirigido
69      private static GraphLink<String> createUndirectedGraph() {
70          GraphLink<String> undirectedGraph = new GraphLink<>();
71
72          undirectedGraph.insertVertex("A");
73          undirectedGraph.insertVertex("B");
74          undirectedGraph.insertVertex("C");
75          undirectedGraph.insertVertex("D");
76          undirectedGraph.insertVertex("E");
77          undirectedGraph.insertVertex("F");
78          undirectedGraph.insertVertex("G");
79
80          undirectedGraph.insertEdge("A", "B");
81          undirectedGraph.insertEdge("A", "C");
82          undirectedGraph.insertEdge("B", "D");
83          undirectedGraph.insertEdge("B", "F");
84          undirectedGraph.insertEdge("C", "F");
85          undirectedGraph.insertEdge("C", "G");
86
87          return undirectedGraph;
88      }
89
90      // Crear grafo dirigido con peso
91      private static GraphLink<String> createWeightedDirectedGraph() {
92          GraphLink<String> weightedDirectedGraph = new GraphLink<>();
93
94          weightedDirectedGraph.insertVertex("A");
95          weightedDirectedGraph.insertVertex("B");
96          weightedDirectedGraph.insertVertex("C");
97          weightedDirectedGraph.insertVertex("D");
98          weightedDirectedGraph.insertVertex("E");
```

```
99          weightedDirectedGraph.insertVertex("F");
100         weightedDirectedGraph.insertVertex("G");
101
102         weightedDirectedGraph.insertEdge("A", 5, "B");
103         weightedDirectedGraph.insertEdge("A", 6, "C");
104         weightedDirectedGraph.insertEdge("B", 7, "D");
105         weightedDirectedGraph.insertEdge("B", 15, "E");
106         weightedDirectedGraph.insertEdge("C", 25, "F");
107         weightedDirectedGraph.insertEdge("C", 234, "G");
108
109         return weightedDirectedGraph;
110     }
111
112     // Mtodo para crear el grafo con las palabras dadas
113     private static GraphLink<String> createWordGraph() {
114         GraphLink<String> wordGraph = new GraphLink<>();
115
116         String[] palabras = {"words", "cords", "corps", "coops", "crops", "drops", "drips",
                "grips", "gripe", "grape", "graph"};
117
118         for (String palabra : palabras) {
119             wordGraph.insertVertex(palabra);
120         }
121
122         for (int i = 0; i < palabras.length; i++) {
123             for (int j = i + 1; j < palabras.length; j++) {
124                 if (diferirEnUnaPosicion(palabras[i], palabras[j])) {
125                     wordGraph.insertEdge(palabras[i], palabras[j]);
126                 }
127             }
128         }
129
130         return wordGraph;
131     }
132
133     // Mtodo para verificar si dos palabras difieren exactamente en una posicin
134     private static boolean diferirEnUnaPosicion(String palabra1, String palabra2) {
135         if (palabra1.length() != palabra2.length()) {
136             return false;
137         }
138
139         int diferencia = 0;
140         for (int i = 0; i < palabra1.length(); i++) {
141             if (palabra1.charAt(i) != palabra2.charAt(i)) {
142                 diferencia++;
143                 if (diferencia > 1) {
144                     return false;
145                 }
146             }
147         }
148
149         return diferencia == 1;
150     }
151
152     private static GraphLink<String> createGraph1() {
153         GraphLink<String> graph = new GraphLink<>();
```

```
154         graph.insertVertex("A");
155         graph.insertVertex("B");
156         graph.insertEdge("A", "B");
157         return graph;
158     }
159 
160     private static GraphLink<String> createGraph2() {
161         GraphLink<String> graph = new GraphLink<>();
162         graph.insertVertex("A");
163         graph.insertVertex("B");
164         graph.insertVertex("C");
165         graph.insertEdge("A", "B");
166         graph.insertEdge("B", "C");
167         return graph;
168     }
169 }
```

Ejecutando nuestro Main, se evidencia el cumplimiento de los ejercicios designados. A continuación capturas de la ejecución.

## 3.3. Ejecución

```
> java .\Main.java
Grafo No Dirigido:
A -->    C -> B
B -->    F -> D -> A
C -->    G -> F -> A
D -->    B
E -->
F -->    C -> B
G -->    C

Recorrido BFS desde el vértice A:
A B C D F G
Recorrido DFS desde el vértice A:
A B D F C G

Grafo Dirigido con Peso:
A -->    C[6] -> B[5]
B -->    E[15] -> D[7]
C -->    G[234] -> F[25]
D -->
E -->
F -->
G -->

Recorrido BFS desde el vértice A:
A B C D E F G
Recorrido DFS desde el vértice A:
A B D E C F G

La distancia mínima entre A y F es: 31

Agregamos un peso de 2 de B a F
La nueva distancia mínima entre A y F es: 7
```

```
Grafo de Palabras:
coops -->       crops -> corps
cords -->       corps -> words
corps -->       coops -> cords
crops -->       drops -> coops
drips -->       grips -> drops
drops -->       drips -> crops
grape -->       graph -> gripe
graph -->       grape
gripe -->       grape -> grips
grips -->       gripe -> drips
words -->       cords

Grafo 1:
A -->    B
B -->    A

Grafo 2:
A -->    B
B -->    C -> A
C -->    B

¿Grafo 1 está incluido en Grafo 2? true
¿Grafo 2 está incluido en Grafo 1? false
```

# 4. Cuestionario

1. **¿Cuantas variantes del algoritmo de Dijkstra hay y cuál es la diferencia entre ellas? (1 puntos)**
   Hay dos variantes principales del algoritmo de Dijkstra: el algoritmo original y la versión con cola de prioridad. La principal diferencia radica en la implementación de la estructura de datos para manejar los vértices no visitados. La versión con cola de prioridad es más eficiente en términos de tiempo de ejecución, ya que mejora la complejidad temporal del algoritmo.

2. **Invetigue sobre los ALGORITMOS DE CAMINOS MINIMOS e indique, ¿Qué similitudes encuentra, qué diferencias, en qué casos utilizar y porque? (2 puntos)**
   Los algoritmos de caminos mínimos, como Dijkstra y Bellman-Ford, comparten la meta de encontrar la ruta más corta entre dos puntos en un grafo ponderado. La principal diferencia es que Dijkstra se utiliza para grafos con pesos no negativos, mientras que Bellman-Ford puede manejar pesos negativos, aunque con una complejidad temporal mayor.

# 5. Conclusiones

- La representación de grafos mediante listas de adyacencia ofrece flexibilidad y eficiencia para diversas operaciones, facilitando la implementación de algoritmos como DFS, BFS, y Dijkstra.
- La modularidad y reutilización de código se favorecen mediante la implementación de clases y métodos específicos, permitiendo construir y mantener sistemas más complejos con facilidad.
- Trabajar con clases propias en lugar de importar bibliotecas externas ofrece mayor control y comprensión sobre el funcionamiento interno del código.

# 6. Referencias

- `https://www.ecured.cu/Algoritmo_de_Dijkstra`

- https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/
- https://runestone.academy/ns/books/published/pythoned/Graphs/UnaListaDeAdyacencia.html