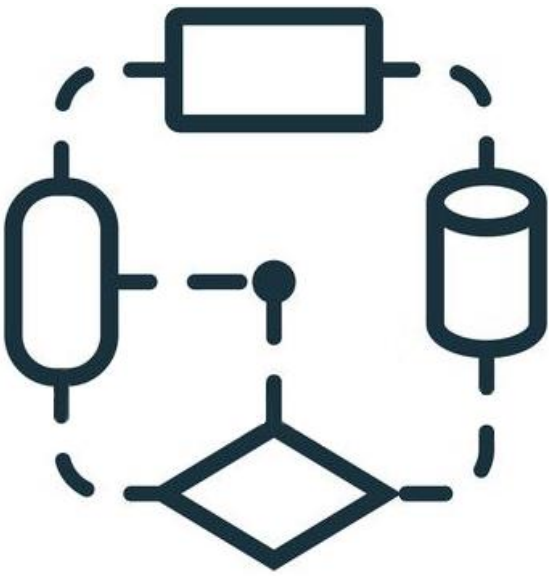
# Programación Dinámica

PRÁCTICA 3



**Autor: Pablo Gómez Rivas y Jesús Fornieles Muñoz**

**Materia: Estructura de Datos y Algoritmos II**

**Grupo de Prácticas: GTAB2**

**Fecha: Domingo, 14 de mayo de 2023**

# Índice

## Objetivo

La programación dinámica es una técnica utilizada en informática y matemáticas para resolver problemas complejos, dividiéndolos en subproblemas más pequeños y resolviendo cada subproblema solo una vez. Esta técnica se basa en la idea de que si se resuelve un subproblema varias veces, se pueden guardar los resultados y reutilizarlos en lugar de calcularlos nuevamente.

El enfoque de programación dinámica se utiliza para problemas que se pueden descomponer en subproblemas superpuestos y que exhiben la propiedad de optimalidad, es decir, la solución óptima a un problema se puede encontrar a partir de las soluciones óptimas de sus subproblemas (Principio de Optimalidad).

El proceso de programación dinámica consta de varios pasos:

1.Definir la estructura del problema: Se describe cómo se relacionan los subproblemas entre sí y cómo se puede construir la solución óptima del problema original a partir de las soluciones óptimas de los subproblemas.

2.Definir la función de recursión: Se establece una relación recursiva que expresa la solución óptima del problema en términos de las soluciones óptimas de los subproblemas más pequeños. Esta función de recursión se llama ecuación de recurrencia.

3.Diseñar la tabla de memorización: Se crea una tabla o matriz donde se almacenan los resultados de los subproblemas resueltos. Esto evita recalcular los mismos subproblemas en el futuro.

4.Resolver los subproblemas: Se resuelven los subproblemas más pequeños primero y se almacenan sus resultados en la tabla de memorización.

5.Construir la solución óptima: Utilizando los resultados almacenados en la tabla de memorización, se construye la solución óptima del problema original.

En esta ocasión aplicamos esta metodología de programación a la red de carreteras de EDAland, en concreto para la empresa EDARoute, cuya misión consiste en desarrollar un servicio web capaz de encontrar el camino más corto entre dos núcleos, misión para la cual acuden a TheBest Soft dado su alto prestigio en EdaLand como la mejor empresa informática del país.

La empresa debe de ser capaz de cumplir con el reto de los caminos mínimos sumado a algunas variaciones propuestas en el informe las cuales deben realizarse con distintos algoritmos asociados a la programación dinámica(Bellman-Ford, Dijsktra… ).

Por otro lado, en lo que respecta a este trabajo, hemos decidido hacer a Jesús Fornieles Muñoz líder de esta práctica, encargado de supervisar y verificar la calidad y el correcto funcionamiento de los códigos del trabajo del resto de miembros, Pablo Gómez Rivas en este caso, de forma que la elaboración del proyecto ha quedado de la siguiente forma:

| # | Tipo | Tarea | |
|---|---|---|---|
| 1 | Implementacion | Algortimo voraz MochilaFraccionada | Pablo |
| 2 | Implementacion | Algortimo voraz MochilaEntera | Jesús |
| 3 | Implementacion | Algortimo voraz MochilaMitad | Pablo |
| 4 | Implementacion | Implementación de Lingotes | Jesús |
| 5 | Implementacion | Implementación de Lingotes infinitos | Pablo |
| 6 | Implementacion | Generación de secuencuas aleatorias. Considerar casos especiales | Jesús |
| 7 | Implementacion | Main con medidas de tiempos de ejecución | Pablo |
| 8 | Implementacion | Juegos de prueba, considerando los casos especiales | Jesús |
| 9 | Documentación | **Estudio de la implementación** | Pablo |
| 10 | Documentación | **Estudio teórico** | Jesús |
| 11 | Documentación | **Estudio experimental** | Pablo |
| 12 | Documentación | **Anexos** y bibliografia | Jesús |

## Antecedentes

Para el desarrollo de la web que ofrezca los caminos más cortos entre los núcleos de EDALand hacemos uso de una serie de algoritmos de programación dinámica:

- Dijkstra:

El algoritmo de Dijkstra es un algoritmo de búsqueda de caminos más cortos que se utiliza para encontrar la ruta más corta desde un nodo de origen a todos los demás nodos en un grafo ponderado dirigido o no dirigido con pesos no negativos. Aunque el algoritmo de Dijkstra tiene un carácter greedy, se puede utilizar en combinación con la programación dinámica para resolver problemas relacionados con la búsqueda de caminos más cortos en problemas más grandes y complejos.

- Bellman Ford:

El algoritmo de Bellman-Ford es un algoritmo de programación dinámica que se utiliza para encontrar la ruta más corta desde un nodo de origen a todos los demás nodos en un grafo ponderado dirigido o no dirigido con pesos negativos. A diferencia del algoritmo de Dijkstra, el algoritmo de Bellman-Ford puede manejar aristas con pesos negativos, lo que le ofrece una mayor funcionalidad, pero puede ser menos eficiente en términos de tiempo de ejecución.

El algoritmo de Bellman Ford se basa en 3 pasos, instanciación en infinito a los nodos inaccesibles inicialmente(menos a nodo fuente), la relajación de las aristas y por último la comprobación de los bucles de pesos negativos.

- Floyd Warshall:

El algoritmo de Floyd-Warshall es un algoritmo de programación dinámica utilizado para encontrar los caminos más cortos entre todos los pares de nodos en un grafo dirigido o no dirigido con pesos. A diferencia de los algoritmos de Dijkstra y Bellman-Ford, que encuentran los caminos más cortos desde un nodo de origen a todos los demás nodos, el algoritmo de Floyd-Warshall encuentra los caminos más cortos entre todos los pares de nodos en el grafo.

El algoritmo de Floyd-Warshall utiliza una matriz bidimensional para almacenar y actualizar las distancias entre los nodos del grafo. Inicialmente, se llena la matriz con los pesos de las aristas directas entre los nodos. Si no hay una arista directa entre dos nodos, se utiliza un valor infinito para representar que no hay conexión directa.

## Preguntas apartado 3:

¿el resultado de la ejecución de cada algoritmo es único?, ¿por qué?.

Sí dado que ambos algoritmos devuelven la solución óptima.

¿El resultado de la ejecución de los dos algoritmos debe ser el mismo?, ¿por qué?

Sí dado que ambos algoritmos devuelven la solución óptima.

¿Qué sucedería al aplicar los algoritmos Bellman-Ford y Dijkstra sobre la red de carreteras reducida, en la que por error hemos introducido la distancia de Almería a Murcia como -224.0?

Dijkstra no funciona pues no trata el caso de los pesos negativos, mientras que bellman-ford puede manejar ese caso.

## Estudio de la Implementación

- **Dijkstra:**

Para entender este algoritmo tenemos que tener en cuenta su carácter greedy, es decir, Dijkstra tiende a elegir siempre la opción más óptima a nivel local. Pero claro, la razón por la hacemos uso de él es porque su forma de iterara garantiza la mejor solución a fin de cuentas.

De esta forma, el algoritmo procede de esta forma:

1.Empezará a buscar cual es el vértice más cercano a que que hayamos elegido como fuente (source), esto lo conseguimos mediante la función minDistance:

```
public static int minDistance(int dist[], Boolean sptSet[]) {
    // Initialize min value
    int min = Integer.MAX_VALUE, min_index = -1;

    for (int v = 0; v < dist.length; v++) {
        if (sptSet[v] == false && dist[v] <= min) {
            min = dist[v];
            min_index = v;
        }
    }

    return min_index;
}
```

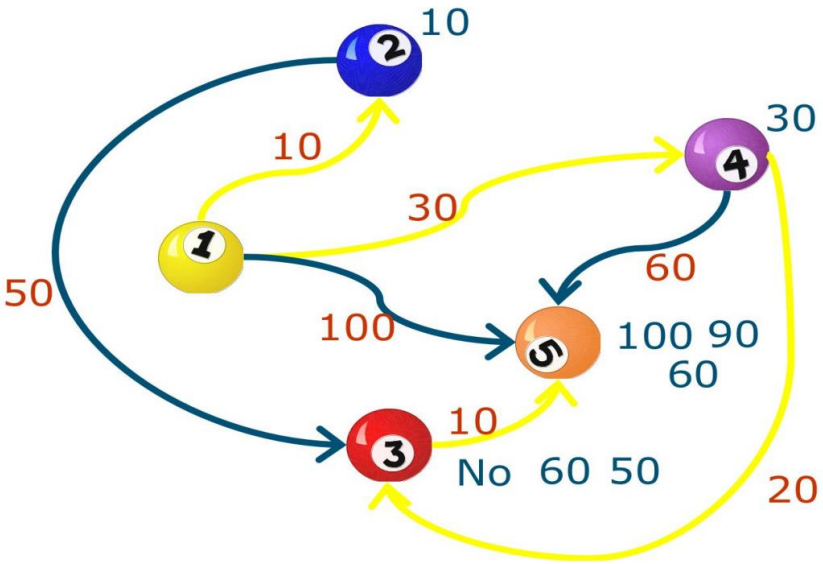
2.Salta al vértice cuyo camino sea el menor.

3.Valorar cuál será el siguiente salto comparando las distancias que nos ofrece el nuevo vértice o el anterior.

4.Repetir este proceso con el resto de vértices.

Ejemplo:

En este caso, 1 será nuestro vértice fuente y encontraremos el camino mínimo a cada uno de los demás vértices de este grafo:



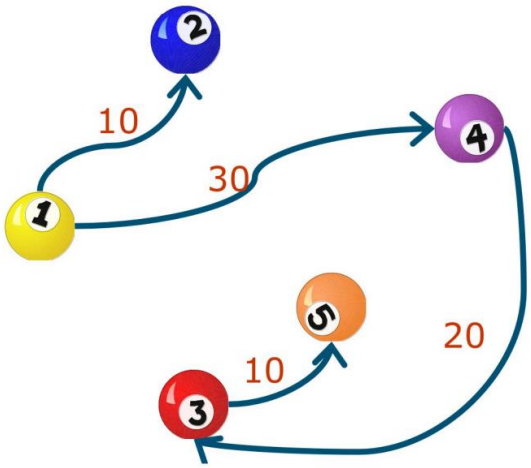

Como vemos tenemos un grafo dirigido conexo:

| Iter | Tratado | VCM | Al vértice 2 | | Al vértice 3 | | Al vértice 4 | | Al vértice 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Peso | Vert | Peso | Vert | Peso | Vert | Peso | Vert |
| Inicial | {1} | | 10 | 1 | ∞ | 1 | 30 | 1 | 100 | 1 |
| 1 | {1,2} | 2 | 10 | 1 | 60 | 2 | 30 | 1 | 100 | 1 |
| 2 | {1,2,4} | 4 | 10 | 1 | 50 | 4 | 30 | 1 | 90 | 4 |
| 3 | {1,2,4,3} | 3 | 10 | 1 | 50 | 4 | 30 | 1 | 60 | 3 |
| 4 | {1,2,4,3,5} | 5 | 10 | 1 | 50 | 4 | 30 | 1 | 60 | 3 |

Una vez creada la tabla nos fijamos en la última fila que nos ofrece el camino más corto del vértice 1 al resto de vértices. Esta fila se puede traducir en la siguiente tabla:

| V | Coste | Vértice |
|---|---|---|
| 1 | 0 | 1 |
| 2 | 10 | 1 |
| 3 | 50 | 4 |
| 4 | 30 | 1 |
| 5 | 60 | 3 |

Siendo esta la solución que nos ofrece el algoritmo, y siendo su representación en el grafo semejante a esta:




- **Bellman Ford:**

Como ya hemos hablado, a diferencia de Dijkstra, Bellman Ford si nos permite trabajar con pesos negativos. Esto es debido al uso de la clase interna Edge, que nos permite depositar pesos a nuestro gusto sin importar la naturaleza de este:

```
static class Edge {
    int src, dest, weight;

    Edge() { src = dest = weight = 0; }

    Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight; }
};
```

Así pues procedemos a almacenar en una lista de Edges aquellas aristas que existen de por si en el grafo, es decir, aquellas != INF:

```
ArrayList<Edge> edges = new ArrayList<Edge>();
for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
        if (graph[i][j] != INF) {
            edges.add(new Edge(i, j, graph[i][j]));
        }
    }
```

Una vez tenemos la lista, comenzamos a recorre de forma iterativa cada una de las aristas para cada uno de los vértices que tenemos en la lista mediante 2 bucles for anidados, de forma que si la distancia de algún Edge ha sido mejorada, lo guardamos en el array de distancia que vamos a ofrecer como respuesta (Relajación de las aristas):

```
for (int i = 1; i < V; i++) {
    for (Edge e : edges) {
        int u = e.src;
        int v = e.dest;
        int weight = e.weight;
        //System.out.println("(" + u + ", " + v + ") - " + weight);
        if ((dist[u] != INF) && (dist[u] + weight < dist[v])) {
            dist[v] = dist[u] + weight;
            pred[v] = u;
        }
    }
```

Por último, nos aseguramos de evitar la presencia de bucles de pesos negativos:

```
for (Edge e: edges) {
    int u = e.src;
    int v = e.dest;
    int weight = e.weight;
    if ((dist[u] != INF) && (dist[u] + weight < dist[v])) {
        System.out.println("Graph contains negative weight cycle");
        return;
    }
}
```

- **Floyd Warshall:**

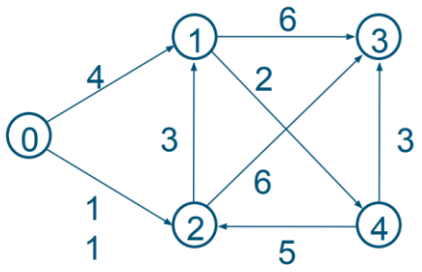
A diferencia de los dos algoritmos anteriores, nos encontramos con que Floyd es el único de ellos que no necesita un nodo fuente, esto es debido a que devuelve el camino mínimo de todos los pares de vértices entre sí. Como consecuencia inevitable, observamos como cuenta con un mayor orden de complejidad.

La forma en la que Floyd opera consiste en recorrer cada peso de cada vertice probando distintos vértices como puente para los demás, actualizando así el grafo con los pesos de los caminos mínimos creados durante la ejecución del método junto a un nuevo grafo que nos indica aquellos nuevos caminos que han sido creados a partir del grafo original(S[]):

```
for (k = 0; k < V; k++) {
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++) {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++) {
            // If vertex k is on the shortest path
            // from i to j, then update the value of graph[i][j]
            if (graph[i][k] + graph[k][j] < graph[i][j]) {
                graph[i][j] = graph[i][k] + graph[k][j];
                S[i][j] = k;
            }
        }
    }
```

Ejemplo:

Para mostrar como funciona el algoritmo planteamos un grafo junto a su matriz de adyacencias:




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 4 | 11 | ∞ | ∞ |
| 1 | ∞ | 0 | ∞ | 6 | 2 |
| 2 | ∞ | 3 | 0 | 6 | ∞ |
| 3 | ∞ | ∞ | ∞ | 0 | ∞ |
| 4 | ∞ | ∞ | 5 | 3 | 0 |

A partir de dicha matriz procedemos a identificar, vértice por vértice, cuál es el mejor puente para crear el camino mínimo hacia el vértice de destino. LLamaremos Dx a cada una de las iteraciones del bucle exterior donde x será el vértice que utilizaremos como puente:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 4 | 11 | ∞ | ∞ |
| 1 | ∞ | 0 | ∞ | 6 | 2 |
| 2 | ∞ | 3 | 0 | 6 | ∞ |
| 3 | ∞ | ∞ | ∞ | 0 | ∞ |
| 4 | ∞ | ∞ | 5 | 3 | 0 |

Como podemos ver, no se ha modificado el grafo, lo que significa que el vértice 0 no permite acortar ningún camino pasando por él (como se puede apreciar en el recuadro rojo).

D1:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 4 | 11 | 10 | 6 |
| 1 | ∞ | 0 | ∞ | 6 | 2 |
| 2 | ∞ | 3 | 0 | 6 | 5 |
| 3 | ∞ | ∞ | ∞ | 0 | ∞ |
| 4 | ∞ | ∞ | 5 | 3 | 0 |

En este caso podemos ver como si hemos actualizado el grafo utilizando el vértice 1 como puente, en concreto el camino que va de 0-3, 0-4 y 2-4.

Por ejemplo, se muestra en verde como el camino 0-3 pasa de ser infinito a tener 10 de peso, pues weight(0-1) = 4, weight(1-3) = 6;

¿4 + 6 < Infinito? La respuesta es sí, lo que hace que cambiemos el grafo otorgando un camino más corto del 0 al 3 que el que había antes.

Finalmente, tras probar todos los vertices como puente nos queda el siguiente resultado.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 4 | 11 | 9 | 6 |
| 1 | ∞ | 0 | 7 | 5 | 2 |
| 2 | ∞ | 3 | 0 | 6 | 5 |
| 3 | ∞ | ∞ | ∞ | 0 | ∞ |
| 4 | ∞ | 8 | 5 | 3 | 0 |

Asemejando este proceso al triple for anidado que utilizamos en nuestro código java:

```
for (k = 0; k < V; k++) {
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++) {
        // Pick all vertices as destination for the
        // above picked source
        for (j = 0; j < V; j++) {
            // If vertex k is on the shortest path
            // from i to j, then update the value of graph[i][j]
            if (graph[i][k] + graph[k][j] < graph[i][j]) {
                graph[i][j] = graph[i][k] + graph[k][j];
                S[i][j] = k;
            }
        }
    }
}
```

Podemos comprobar como el grafo cuenta ahora con pesos mucho más asequibles.

Por otro lado también es conveniente realizar algún método que nos permita saber cual ha sido el vertice puente que nos ha permitido realizar el salto para acortar el camino, tal y como se muestra en la siguiente tabla:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | -1 | -1 | -1 | 4 | 1 |
| 1 | -1 | -1 | 4 | 4 | -1 |
| 2 | -1 | -1 | -1 | -1 | 1 |
| 3 | -1 | -1 | -1 | -1 | -1 |
| 4 | -1 | 2 | -1 | -1 | -1 |

Donde -1 indica que no ha habido ningún cambio respecto al grafo original, mientras que los números naturales del 1 al 4 muestran el vértice puente utilizado para conseguir el peso concreto de la posición en la que se encuentra el número.

Asemejando esta matriz al código siguiente:

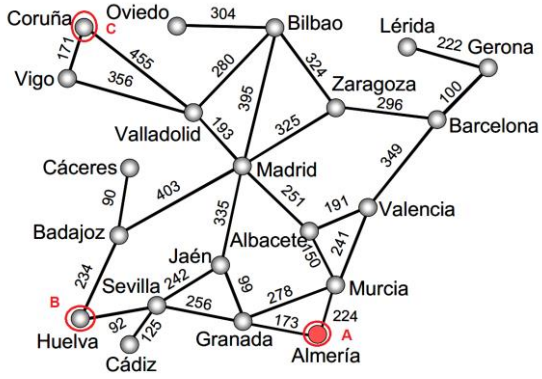
```
S = new int [V][V];
for (i = 0; i < V; i++) {
    for (j = 0; j < V; j++) {
        S[i][j] = -1;
    }
}
```

```
S[i][j] = k;
```

### Dijkstra/Bellman Ford por nodo intermedio:

Para este apartado tan solo aplicamos un método para ambos algoritmos debido a que ambos siguen el mismo principio lógico para la resolución del apartado.

Este principio consiste en que el camino más corto de un vertice A a un vertice C pasando por B




será siempre el camino mínimo de A a B y el camino mínimo de B a C.

Una vez claro esto, observamos que una posible resolución del problema puede ser aplicar un algoritmo de caminos mínimos dos veces, una del nodo fuente al nodo intermedio y otra del nodo intermedio al nodo destino.

De esta forma el path que nos devuelve el método será siempre el camino mínimo del nodo fuente al destino pasando por el nodo intermedio:

```
case Dijkstra:
    S1 = dijkstra(src)[1];
    S2 = dijkstra(verticeIntermedio)[1];
    break;
case BellmanFord:
    S1 = bellmanFord(src)[1];
    S2 = bellmanFord(verticeIntermedio)[1];
    break;
case FloydWarshall:
    return "Método no implementado para Floyd";
```

**Generador de redes aleatorias:**

Para realizar el apartado 6 de la práctica necesitamos un mecanismo de generación de redes aleatorias que nos permita crear grafos no orientados, valorados positivamente y conexos a partir de un número de vértices y aristas a nuestra elección.

Para conseguir esto podemos hacer uso de algoritmos ya existentes como cruzcal o prim, sin embargo optamos por una opción más intuitiva y de fácil comprensión:

Comprobamos que el número de aristas sea mayor que el numero de vertices - 1, garantizando así que el grafo puede ser conexo:

```
// Compruebo que el número de aristas es al menos número de vértices - 1
if (numA < numV - 1)
    throw new Exception("Un grafo necesita más aristas para ser conexo.");
```

Inicializamos la matriz con 0:

```
// Inicializo la matriz con todo 0
for (int i = 0; i < numV; i++)
    for (int j = 0; j < numV; j++)
        adjMatrix[i][j] = 0;
```

Colocamos las aristas mínimas necesarias para garantizar la conexidad del grafo con pesos aleatorios:

```
// Creo las mínimas aristas necesarias para que sea conexo, que son número de
// vértices - 1
int numAMin = numV - 1;
for (int i = 0; i < numAMin; i++)
    adjMatrix[i][i + 1] = rand.nextInt(500) + 1;
```

Introducimos aristas aleatorias con pesos aleatorios entre vértices aleatorios hasta llegar al numero de aristas deseado:

```
// Completamos con aristas adicionales hasta alcanzar el número de aristas
// deseado
int numAAct = numA - numAMin;
for (int i = 0; i < numAAct; i++) {
    int v1 = rand.nextInt(numV);
    int v2 = rand.nextInt(numV);
    if (adjMatrix[v1][v2] == 0)
        adjMatrix[v1][v2] = rand.nextInt(500) + 1;
    else
        i--;
}
```

## Estudio Teórico

**-Dijsktra:**

```
public static int[][] dijkstra(int src) {
          int V = grafo.length;
          int dist[] = new int[V];

          Boolean sptSet[] = new Boolean[V];
          int S[] = new int[V];

          for (int i = 0; i < V; i++) {
               dist[i] = grafo[src][i] != 0 ? grafo[src][i]
: Integer.MAX_VALUE;
               sptSet[i] = false;
               S[i] = grafo[src][i] != 0 ? src : -1;
          }

          dist[src] = 0;
          S[src] = src;
          sptSet[src] = true;
for (int count = 0; count < V - 1; count++) {
int u = minDistance(dist, sptSet);
```

```
            for (int count = 0; count < V - 1; count++) {

                int u = minDistance(dist, sptSet);

                sptSet[u] = true;
            for (int v = 0; v < V; v++) {
                for (int v = 0; v < V; v++) {

                    if (!sptSet[v] && grafo[u][v] != 0 &&
dist[u] != Integer.MAX_VALUE && dist[u] + grafo[u][v] <
dist[v]) {

                        dist[v] = dist[u] + grafo[u][v];
                        S[v] = u;

                    }
                }
            }
            return new int[][] { dist, S };
        }
```

- El orden de complejidad de las líneas no marcadas es $O(1)$ ya que son operaciones elementales de tiempo constante.
- El orden de complejidad del método minDistance: `int u = minDistance(dist, sptSet);` es $O(n)$:
- El orden de complejidad del bucle `for (int count = 0; count < V - 1; count++) {` es $O(n)$ ($\sum_{i=1}^{n} d$).
- El orden de complejidad del bucle `for (int v = 0; v < V; v++) {` es $O(n)$ ($\sum_{i=1}^{n} d$).
-

se calcula:

$$\sum_{Count=0}^{V-1} \left(\sum_{v=0}^{V} (c) + \sum_{v=0}^{V} (d)\right)$$

$$\in O(n^2).$$

**Bellman Ford:**

```
            public static int[][] bellmanFord(int src) {
```

```
            int V = grafo.length;
            int dist[] = new int[V];
            int pred[] = new int[V];

            ArrayList<Edge> edges = new
ArrayList<Edge>();
            for (int i = 0; i < V; i++) {
                  for (int j = 0; j < V; j++) {
                        if (grafo[i][j] != INF) {
                              edges.add(new Edge(i, j,
grafo[i][j]));
                        }
                  }
            }
            for (int i = 0; i < V; i++) {

                  dist[i] = INF;
                  pred[i] = -1;
            }
            dist[src] = 0;

            for (int i = 1; i < V; i++) {
                  for (Edge e : edges) {
                        int u = e.src;
                        int v = e.dest;
                        int weight = e.weight;
                        if ((dist[u] != INF) && (dist[u] +
weight < dist[v])) {
                              dist[v] = dist[u] + weight;
                              pred[v] = u;
                        }
                  }
            }
```

$$\sum_{i=0}^{V}\left(\sum_{j=0}^{V}(c)\right)+\sum_{i=0}^{V}(c)+\sum_{i=0}^{V}\left(\sum_{j=0}^{Edges}(c)\right)$$

$$\in O(n^2).$$

**Floyd Wharsall:**

```
public static int[][][] floydWarshall() {
          int i, j, k;
          int V = grafo.length;
          int[][] S = new int[V][V];

          for (int i = 0; i < V; i++) {
               for (int j = 0; j < V; j++) {

                    S[i][j] = -1;
               }
          }

          for (int k = 0; k < V; k++) {
               for (int i = 0; i < V; i++) {

                    for (int j = 0; j < V; j++) {

                         if (grafo[i][k] + grafo[k][j]
< grafo[i][j]) {
                              grafo[i][j] =
grafo[i][k] + grafo[k][j];
                              S[i][j] = k;
                         }
                    }
               }
          }
          return new int[][][] { grafo, S };
     }
```

$$\sum_{i=0}^{V}\left(\sum_{j=0}^{V}(c)\right)+\sum_{k=0}^{V}\left(\sum_{i=0}^{V}\left(\sum_{j=0}^{V}(c)\right)\right)$$

$$\in O(n^3).$$

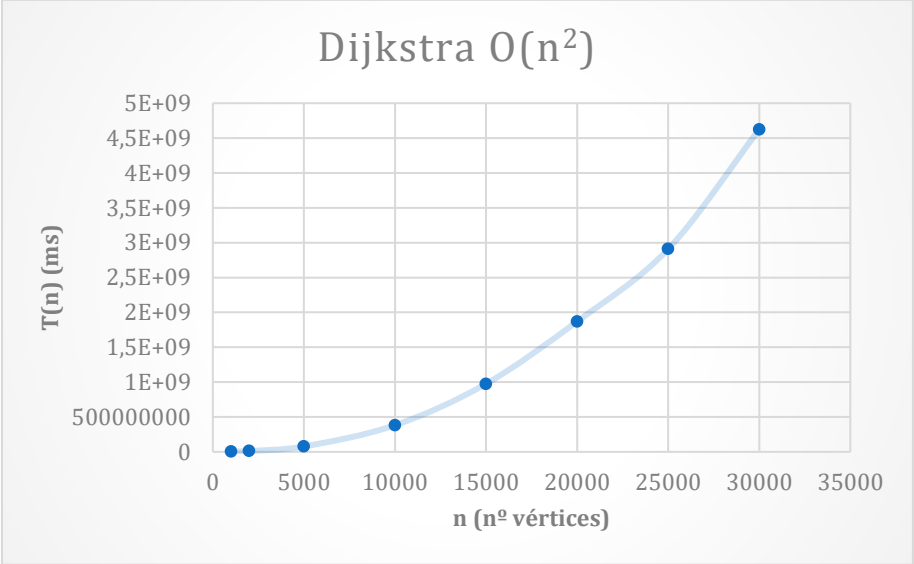
## Estudio Experimental

### Dijkstra O(n²)

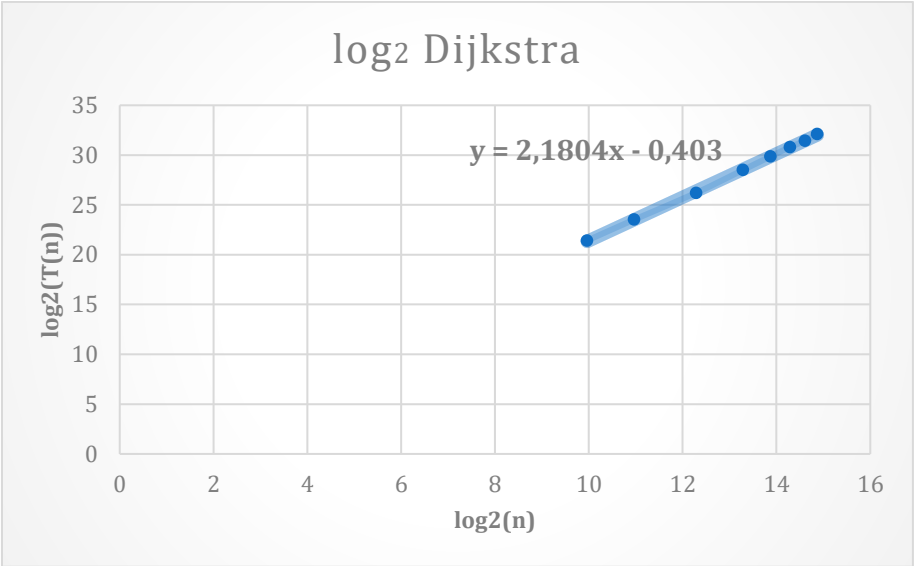
Para el conjunto de datos de graphEDAland.txt, el algoritmo Dijkstra ha tardado 50290 ns, y para graphEDAlandLarge.txt ha tardado 2876455 ns.

| n (nº vértices) | $\bar{T}(n)$ (ms) | log2(n) | log2($\bar{T}(n)$) |
|---|---|---|---|
| **1000** | 2810695 | 9,965784285 | 21,42249548 |
| **2000** | 1,20E+07 | 10,96578428 | 23,52002417 |
| **5000** | 7,76E+07 | 12,28771238 | 26,21001338 |
| **10000** | 3,82E+08 | 13,28771238 | 28,50778102 |
| **15000** | 9,76E+08 | 13,87267488 | 29,86160829 |
| **20000** | 1,87E+09 | 14,28771238 | 30,80045204 |
| **25000** | 2,91E+09 | 14,60964047 | 31,43780519 |
| **30000** | 4,62E+09 | 14,87267488 | 32,1067534 |

Tiempo de Ejecución VS Tamaño de la Entrada




Gráfica tiempo de ejecución VS Tamaño de la entrada



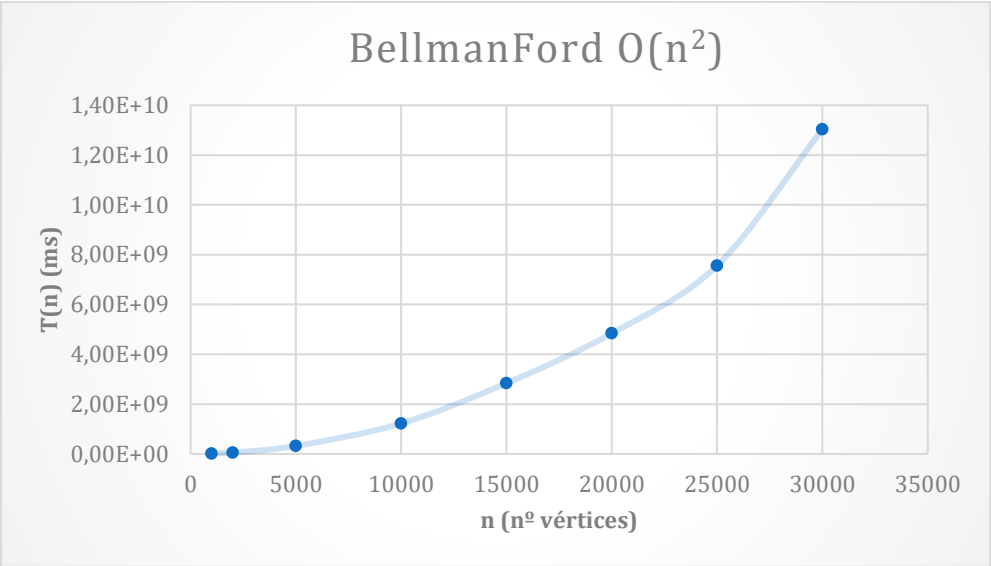

Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 2,1804**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad **$O(n^{2,1804})$.**

### BellmanFord $O(n^2)$

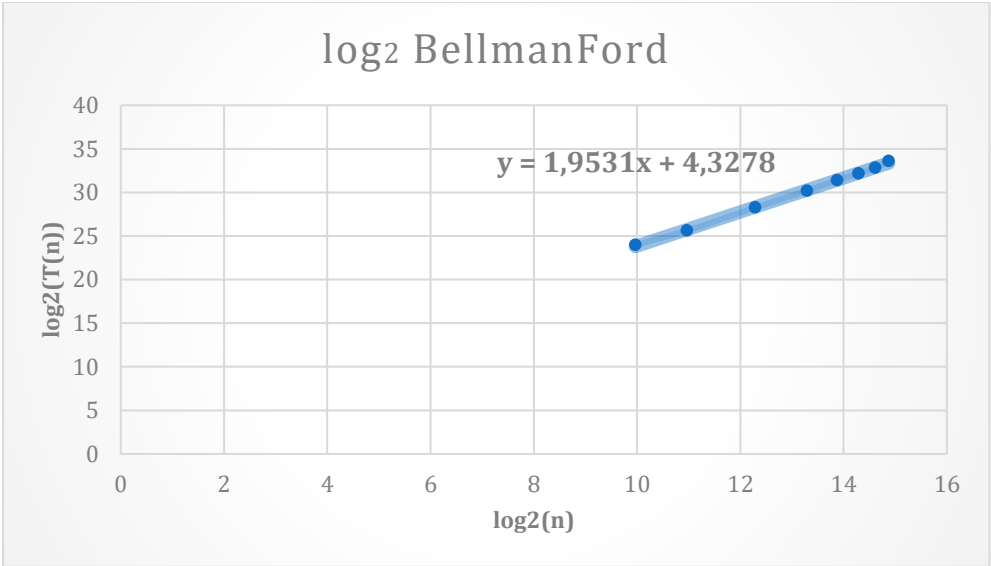
Para el conjunto de datos de graphEDAland.txt, el algoritmo BellmanFord ha tardado 209695 ns, y para graphEDAlandLarge.txt ha tardado 17388420 ns.

| n (nº vértices) | $\bar{T}(n)$ (ms) | log2(n) | log2($\bar{T}(n)$) |
|---|---|---|---|
| **1000** | 1,64E+07 | 9,965784285 | 23,96934172 |
| **2000** | 5,15E+07 | 10,96578428 | 25,61761367 |
| **5000** | 3,22E+08 | 12,28771238 | 28,26216923 |
| **10000** | 1,22E+09 | 13,28771238 | 30,18839767 |
| **15000** | 2.841.450.425 | 13,87267488 | 31,4039804 |
| **20000** | 4.844.808.875 | 14,28771238 | 32,17379261 |
| **25000** | 7.566.064.870 | 14,60964047 | 32,816896 |
| **30000** | 13.035.369.640 | 14,87267488 | 33,60171244 |

Tiempo de Ejecución VS Tamaño de la Entrada




Gráfica tiempo de ejecución VS Tamaño de la entrada



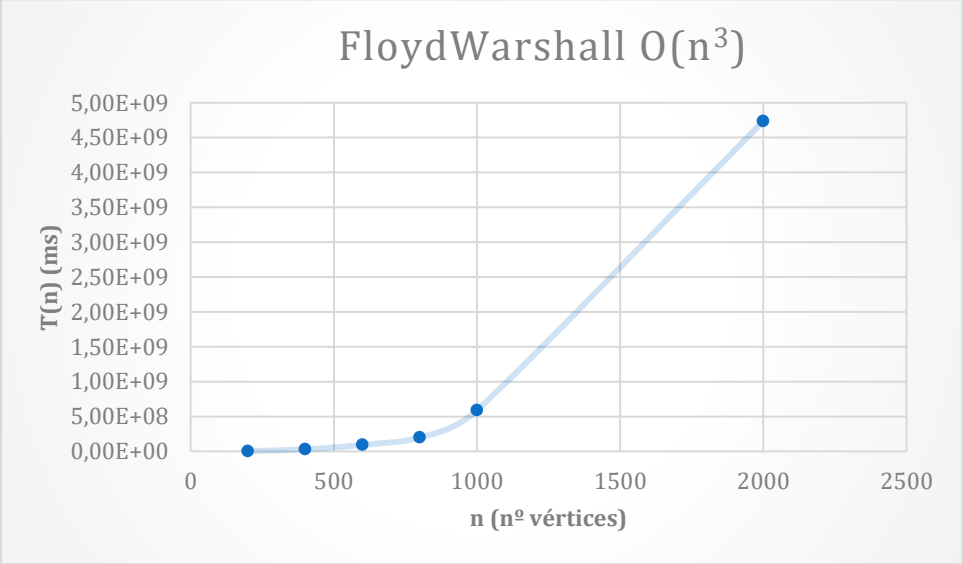

Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 1,9531**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad **$O(n^{1,9531})$.**

### FloydWarshall $O(n^3)$

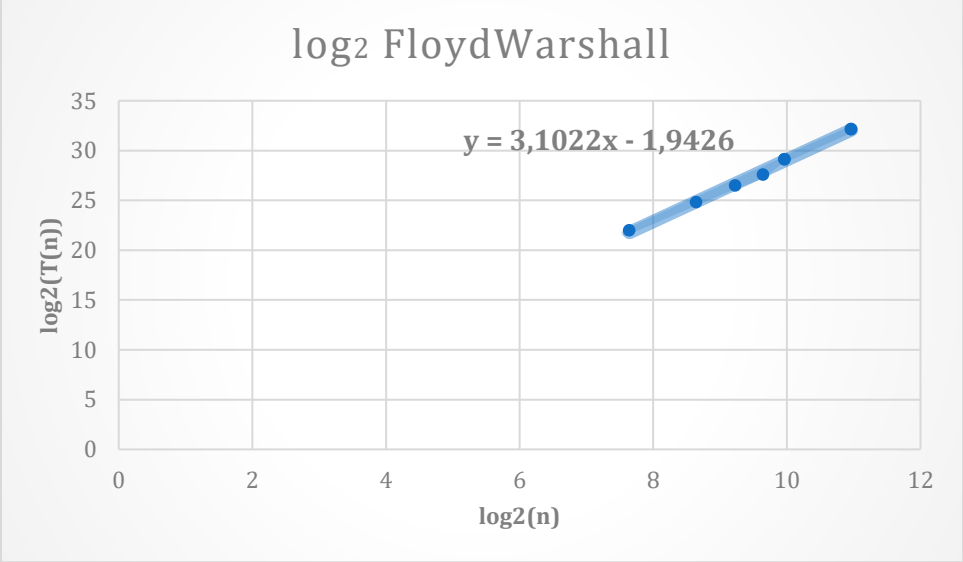
Para el conjunto de datos de graphEDAland.txt, el algoritmo FloydWarshall ha tardado 272815 ns, y para graphEDAlandLarge.txt ha tardado 730287915 ns.

| n (nº vértices) | $\bar{T}(n)$ (ms) | log2(n) | log2($\bar{T}(n)$) |
|---|---|---|---|
| **50** | 786080 | 9,965784285 | 29,13411746 |
| **100** | 1111555 | 10,96578428 | 32,14125418 |
| **200** | 4,15E+06 | 7,64385619 | 21,98572765 |
| **400** | 2,94E+07 | 8,64385619 | 24,80851666 |
| **600** | 9,30E+07 | 9,22881869 | 26,47106544 |
| **800** | 2,01E+08 | 9,64385619 | 27,58476749 |
| **1000** | 5,89E+08 | 9,965784285 | 29,13411746 |
| **2000** | 4,74E+09 | 10,96578428 | 32,14125418 |

Tiempo de Ejecución VS Tamaño de la Entrada



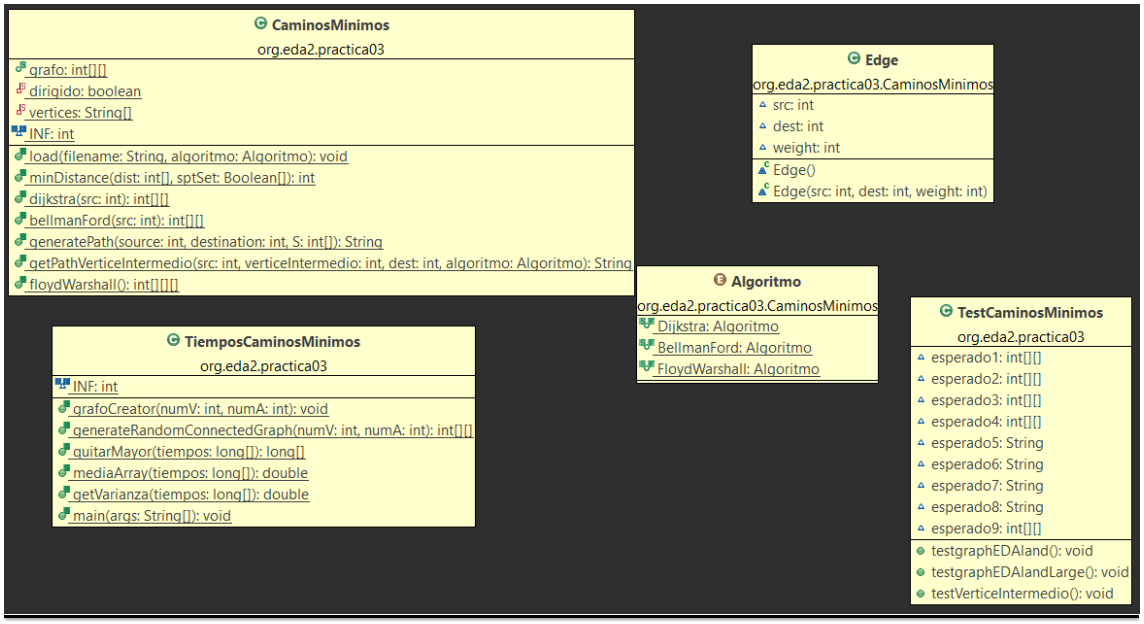

Gráfica tiempo de ejecución VS Tamaño de la entrada




Fijándonos en la anterior gráfica, podemos ver que la pendiente de la ecuación de la recta es **k = 3,1022**. Por tanto, podemos decir que este algoritmo tiene orden de complejidad **$O(n^{3,1022})$.**

# Anexo

## Diagrama de clases:




## Archivos fuente:

- **CaminosMinimos:**

En este archivo dispuesto como clase java podremos encontrar los 5 métodos exigidos durante la práctica destinados al cálculo de caminos mínimos entre nodos (y sus distintas variables), junto a una serie de métodos (getters y métodos auxiliares) y clases auxiliares (edge).

- **TiemposCaminosMinimos:**

En este archivo dispuesto como clase java podemos encontrar una amigable interfaz que nos permite comprobar el tiempo de cualquiera de los algoritmos de la práctica con el n(numero de vértices) que deseemos agregar.

- **TestCaminosMinimos:**

En este archivo dispuesto como clase java observamos la existencia de una serie de diversos test Junit 5 que nos permiten probar nuestro código con distintas variaciones de la red de carreteras de EDALand para probar nuestro código de la forma más completa posible.

## Fuentes Bibliográficas

1.Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Este libro clásico proporciona una introducción exhaustiva a los algoritmos, incluida una sección dedicada a la programación dinámica.
Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). Algorithms. McGraw-Hill.

2.Este libro de texto abarca diversos temas algorítmicos, y presenta una sección sobre programación dinámica con ejemplos y explicaciones detalladas.

kiena, S. S. (2008). The Algorithm Design Manual (2nd ed.). Springer. Este libro práctico y de fácil lectura aborda la programación dinámica y otros temas algorítmicos con ejemplos y problemas resueltos.

3.Kleinberg, J., & Tardos, É. (2005). Algorithm Design. Pearson Education.

Este libro de texto presenta algoritmos y estrategias de diseño de algoritmos, incluyendo una sección dedicada a la programación dinámica.

Bellman, R. (1957). Dynamic Programming. Princeton University Press.