

# **Universidad de los Andes**

FACULTAD DE INGENIERÍA  
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS  
DISEÑO Y ANÁLISIS DE ALGORITMOS



PROYECTO: ETAPA I

Sergio Pardo Gutierrez  
Juan Diego Calixto  
Juan Diego Yepes

29 de marzo de 2022  
Bogotá D.C.

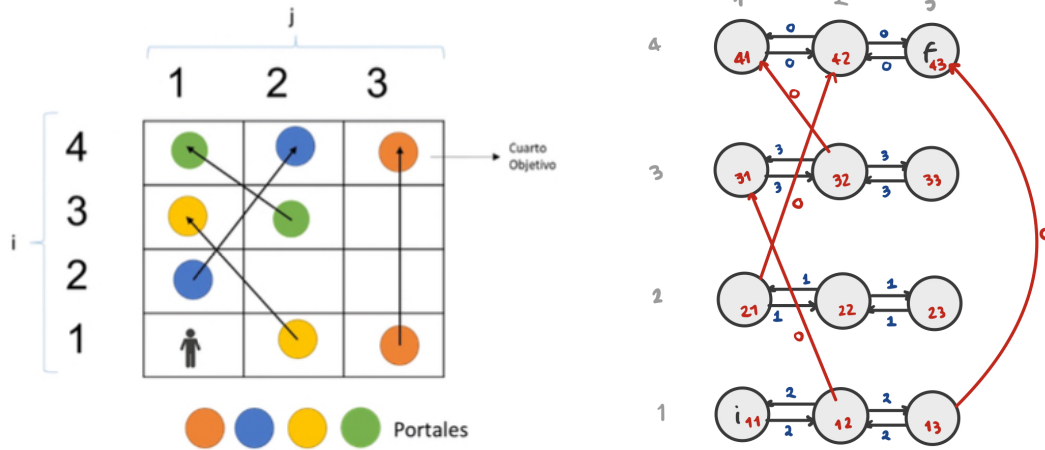
## Tabla de contenidos

<b>1</b>	<b>Algoritmo de solución</b>	<b>2</b>
1.1	Especificación . . . . .	2
1.2	Código . . . . .	3
<b>2</b>	<b>Análisis de complejidad espacial y temporal</b>	<b>4</b>
2.1	Algoritmo . . . . .	4
2.2	Creación de las estructuras de datos . . . . .	5
<b>3</b>	<b>Respuestas a los escenarios de comprensión de problemas algorítmicos</b>	<b>6</b>

# 1 Algoritmo de solución

Teniendo en cuenta el problema del proyecto, decidimos implementar el siguiente algoritmo

Para la solución del proyecto desarrollamos el siguiente algoritmo. Este algoritmo está basado en el algoritmo de Dijkstra, el cual maneja un grafo. Es por ello que utilizaremos terminología de grafos, para poder explicar mejor cómo funciona. De forma ilustrativa, se muestra ahora el grafo con uno de los datos de entrada del ejemplo, junto con la imagen ilustrativa del enunciado:



En el grafo los vértices estarán denominados con su primer dígito como el piso de la matriz (filas  $i$ ) y el segundo como el número del cuarto (columnas  $j$ ).

## 1.1 Especificación

Para el desarrollo del algoritmo hicimos las siguientes entradas y salidas para la función `torreDeTeletransportacion()`

E/S	Nombre	Tipo	Significado
E	c	HashMap	Llave: Vértice $v$ del grafo, Valor: Lista de vértices a los que puede llegar $v$
E	p	HashMap	Llave: Tupla que representa un arco, Valor: Peso del arco
S	n	nat	$e$ gastada en el camino más corto desde el inicio hasta el final

**Precondición:**

$$P : (\forall \langle k, v \rangle \in c \mid \langle k, v \rangle \in p.keys()) \wedge c.size() = ij$$

Donde  $i$  y  $j$  son el numero de filas y de columnas de la matriz del problema.

**Poscondición:** Se define el predicado  $\text{pesoRecorrido}(i, f) = (\sum k \in \text{nat} \mid i < k \leq f : p(i, k) \wedge k \in c)$

$$R : n = \min(\text{pesoRecorrido}(c, p))$$

## 1.2 Código

```
1 def torreDeTeletransportacion(caminos:dict, pesos:dict)->int:
2     distancias = {}
3     recorridos = ["11"]
4     llaves = list(caminos.keys())
5
6     for nodo in llaves:
7         distancias[nodo] = math.inf
8
9     for adyacente in caminos["11"]:
10         distancias[adyacente] = pesos[("11",adyacente)]
11
12     w = None
13
14     while (len(recorridos)!= len(llaves)):
15         minimo = math.inf
16
17         for node in recorridos:
18             for camino in caminos[node]:
19                 if camino not in recorridos and pesos[(node,camino)]<minimo:
20                     minimo = pesos[(node,camino)]
21                     w = camino
22
23         for adyacente in caminos[w]:
24             distancias[adyacente] = min(distancias[adyacente],distancias[w] + pesos[(w
25             ,adyacente)])
26
27         recorridos.append(w)
28
29     masCorto = distancias[llaves[-1]]
30     if masCorto == math.inf:
31         masCorto = "NO EXISTE"
32     return masCorto
```

El algoritmo funciona de la siguiente manera:

1. En la línea 2 se crea el diccionario de distancias, que es donde se irán guardando las distancias mínimas del nodo inicial a todos los nodos del grafo.
2. Después de esto, en la línea 3, se inicializa una lista donde se guardarán los nodos que se van recorriendo y se agrega el nodo "11", que siempre será el primer nodo a recorrer.
3. Posteriormente, en la línea 4, se guardan las llaves de todos los nodos en una lista para inicializar las el diccionario de las distancias en la línea 6
4. Después de esto, se relajan las distancias del nodo "11" a sus nodos adyacentes (línea 9)
5. Se inicializa la variable w donde se guardará el siguiente nodo a recorrer en cada iteración.
6. Se empieza el ciclo
7. Inicializamos un mínimo para realizar un algoritmo de búsqueda secuencial y poder seleccionar el siguiente nodo a recorrer.

8. Se recorre la lista de los nodos que han sido recorridos y se consultan los pesos a los posibles nuevos caminos. Si el nodo no ha sido recorrido y su peso es el menor dentro de los posibles nodos a recorrer, este será el próximo nodo a recorrer (línea 17).
9. Se recorren los posibles caminos de  $w$  para relajar los pesos y seleccionar el peso menor entre los posibles caminos.
10. Habiendo relajado las distancias, guardamos el nodo en la lista de nodos recorridos y volvemos a hacer el recorrido hasta tener el camino más corto a todos los nodos.
11. Finalmente se revisa el camino más corto entre la casilla inicial y la final, si la distancia corresponde a infinito quiere decir que no hay un camino posible entre ambos.

## 2 Análisis de complejidad espacial y temporal

### 2.1 Algoritmo

Para el cálculo de **complejidad temporal** del algoritmo, es necesario hacer una tabla de las operaciones básicas:

Operación	Constante	Veces
Asignación (=)	$c_1$	$3 + 2n + 2n^2$
Suma (+)	$c_2$	$n^2$
Comparación (!=, ==)	$c_3$	$n + n^2$
In (in)	$c_4$	$2n^2$
Append (.append())	$c_5$	$n$
Get ([index])	$c_6$	$3n + n^2$

Vale la pena resaltar varios aspectos:

1. En el código al que hace referencia el punto 8 de la explicación del algoritmo (un doble for para encontrar el siguiente nodo a recorrer) se podría argumentar que la complejidad del algoritmo es  $O(n^3)$ . Sin embargo, dada la estructura del diccionario caminos, cada nodo solo tiene 3 posibles caminos; ir a la izquierda, a la derecha o si hay un portal atravesarlo. Por esta razón, la cota del ciclo interno será 3 y se convierte en una constante. De esta manera, se descarta la posibilidad de que el algoritmo sea  $O(n^3)$ .
2. Las operaciones in, get y append() son  $O(1)$  en los diccionarios, luego las podemos considerar como constantes.
3. Según la documentación de Python, la función min es  $O(n)$ , sin embargo como se utiliza con solo dos parámetros (y no recorriendo una lista) podemos decir que en nuestra implementación es  $O(1)$ .

Luego tenemos la siguiente ecuación para el orden de complejidad temporal:  $T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 = 3 + 7n + 7n^2$  Que en notación big  $O$  es  $O(n^2)$

Por otro lado, para la **complejidad espacial**, Las estructuras creadas van a ser máximo de tamaño  $n$  en el peor de los casos, luego en complejidad espacial el algoritmo sería de  $S(n) = n$  que en big  $O$  es  $O(n)$ .

## 2.2 Creación de las estructuras de datos

Para crear los diccionarios de pesos y caminos utilizamos el siguiente código:

```

1 numero_casos = int(sys.stdin.readline())
2 for _ in range(numero_casos):
3     pesos = {}
4     caminos = {}
5
6     linea1 = sys.stdin.readline().split()
7     pisos = int(linea1[0])
8     cuartos = int(linea1[1])
9     portales = int(linea1[2])
10    energias = sys.stdin.readline().split()
11
12    for piso in range(1,pisos+1):
13        for cuarto in range(1,cuartos):
14            primerCuarto = str(piso)+str(cuarto)
15            cuartoAdy = str(piso)+str(cuarto+1)
16            pesos[(primerCuarto,cuartoAdy)] = int(energias[piso-1])
17            pesos[(cuartoAdy,primerCuarto)] = int(energias[piso-1])
18            listaCaminosPrimero = []
19            listaCaminosPrimero.append(cuartoAdy)
20            listaCaminosAdy = []
21            listaCaminosAdy.append(primerCuarto)
22            if primerCuarto not in caminos:
23                caminos[primerCuarto] = listaCaminosPrimero
24            else:
25                caminos[primerCuarto].extend(listaCaminosPrimero)
26            caminos[cuartoAdy] = listaCaminosAdy
27
28    for p in range(portales):
29        portales = sys.stdin.readline().split()
30        inicioPortal = str(portales[0])+str(portales[1])
31        finPortal = str(portales[2])+str(portales[3])
32        pesos[(inicioPortal,finPortal)] = 0
33        caminos[inicioPortal].append(finPortal)
34    print(torreDeTeletransportacion(caminos,pesos))

```

Para el cálculo de **complejidad temporal** del código, es necesario hacer una tabla de las operaciones básicas:

Operación	Constante	Veces
Asignación (=)	$c_1$	$5k + n + n^2$
Suma (+)	$c_2$	$n$
Append (.append())	$c_3$	$n + n^2$
Get ([index])	$c_4$	$n + n^2$

Donde  $k$  son el número de líneas de la entrada. Luego tenemos la siguiente ecuación para el orden de complejidad temporal:  $T(n) = c_1 + c_2 + c_3 + c_4 = 5k + 3n + 3n^2$  Que en notación big  $O$  es  $O(n^2)$

En complejidad espacial ya se analizaron las estructuras de datos, el diccionario de caminos debe ser de tamaño  $n$  porque tiene todos los vértices y el diccionario de pesos debe ser de tamaño  $2n + x$  donde  $x$  es el número de portales, luego en complejidad espacial el algoritmo sería de  $S(n) = 3n + x$  que en big  $O$  es  $O(n)$ .

### 3 Respuestas a los escenarios de comprensión de problemas algorítmicos

Para las siguientes preguntas se plantean respuestas desde una perspectiva analítica, mas no se presentan algoritmos ni especificación de los problemas.

**ESCENARIO 1:** Suponga ahora que los portales son bidireccionales.

1. ¿Qué nuevos retos presupone este nuevo escenario -si aplica-?  
Desde la implementación que tenemos, no supone ningún reto mayor, ya que podemos seguir utilizando nuestro algoritmo que trabaja desde grafos. Con este escenario el grafo dejaría de ser dirigido pero los caminos se calcularían de la misma forma.
2. ¿Qué cambios -si aplica- le tendría que realizar a su solución para que se adapte a este nuevo escenario?  
Lo que deberíamos cambiar es que ahora podemos encontrar otro arco del fin del portal al inicio del mismo; luego nuestra estructura de diccionario de caminos debe contener ahora un elemento más en la lista que corresponde a la llave del nodo del final del portal, indicando que se puede devolver, pero eso no va a ocupar un espacio considerablemente más grande.

**ESCENARIO 2:** Se le pide ahora calcular el número de rutas que existen.

1. ¿Qué nuevos retos presupone este nuevo escenario -si aplica-?  
Dado que el algoritmo de Dijkstra está implementado ambos desde la forma de un algoritmo greedy y de programación dinámica, esto implica que estamos guardando los recorridos para llegar al final. Por esto, el cambio en el algoritmo no sería mayor.
2. ¿Qué cambios -si aplica- le tendría que realizar a su solución para que se adapte a este nuevo escenario?  
Para satisfacer este nuevo requisito, podría inicializarse un contador y cada vez que se deba hacer una comparación entre los caminos para llegar al nodo final se le suma 1 a este contador. Al final, el contador daría información sobre cuantas veces se analizó un posible camino hacia el nodo final, y por lo tanto, el número de posibles caminos del nodo inicial al final.

**ESCENARIO 3:** Se le pide ahora calcular la ruta óptima en la cual el estudiante puede visitar todos los cuartos y finaliza en  $(n,m)$ .

1. ¿Qué nuevos retos presupone este nuevo escenario -si aplica-?  
Este escenario puede suponer un reto algo mayor para el algoritmo, pues es ahora no se quiere hallar el camino más corto entre el nodo inicial y el resto de los nodos, sino la ruta más corta tal que el estudiante pueda pasar por todos los cuartos. Esto podría cambiar el algoritmo al punto que ya no sea el algoritmo de Dijkstra.

2. ¿Qué cambios -si aplica- le tendría que realizar a su solución para que se adapte a este nuevo escenario?

Se podría encontrar un árbol orden topológico que asegure que sea el camino óptimo. Podría dársele una aproximación más hacia el lado greedy del algoritmo para que este tome el camino más corto y vaya recorriendo todos los nodos de esta manera. Otra alternativa, podría ser más dinámica y hallar todos los posibles caminos entre el nodo inicial y final, tal que se recorran todos los caminos, para dar una respuesta óptima.