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.

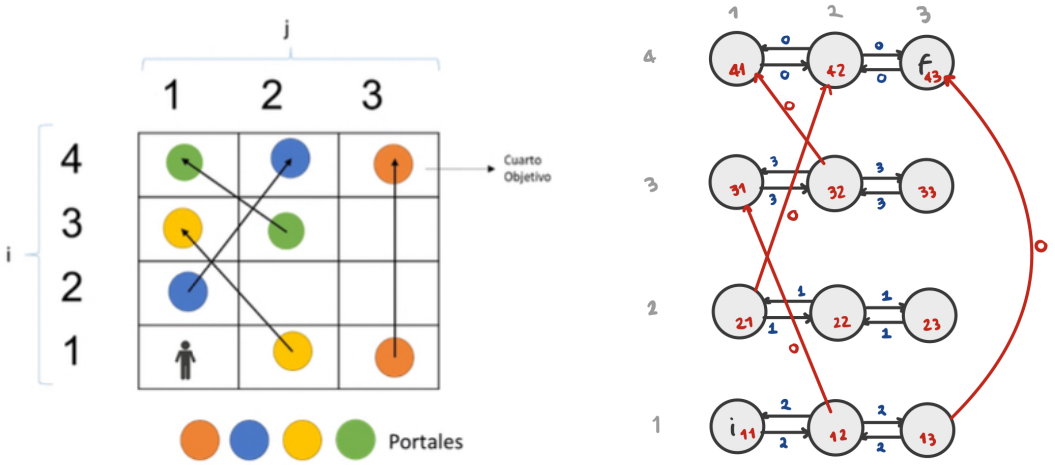
Tabla de contenidos

1 Algoritmos de solución

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

1.1 Primera aproximación (P1)

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).

1.1.1 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:

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

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

Poscondición: Se define el predicado `pesoRecorrido(i,f)` $= (\sum k \in nat | i < k <= f : p(i,k) \wedge k \in c)$

$$R : n = min(pesoRecorrido(c,p))$$

1.1.2 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
    ,adyacente)])
25
26         recorridos.append(w)
27
28     masCorto = distancias[llaves[-1]]
29     if masCorto == math.inf:
30         masCorto = "NO EXISTE"
31     return masCorto
```

El algoritmo funciona de la siguiente manera:

1. En la linea 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, o 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.

1.2 Segunda aproximación (P2)

Debido a la complejidad temporal tan alta del anterior algoritmo (ver sección 2.1 apartado 1.1), aunque pudiera resolver los distintos casos de manera satisfactoria, se buscó cambiar el algoritmo para reducir su complejidad temporal ya que esta no correspondía con la complejidad temporal teórica del algoritmo de Dijkstra. Para lograr esto, se desarolló el siguiente código.

1.2.1 Especificación

Para este segundo algoritmo se utilizaron las mismas estructuras de entrada que el anterior por lo que su precondición y entradas serán iguales. De la misma forma, se busca la misma respuesta al problema, por lo que las salidas y la poscondición también serán iguales.

1.2.2 Código

```
1  def torreDeTeletransportacion(caminos:dict, pesos:dict)->int:
2
3      distancias = {}
4      recorridos= {}
5      posiblesCaminos = []
6      llaves = list(caminos.keys())
7      for nodo in llaves:
8          distancias[nodo] = math.inf
9          recorridos[nodo] = False
10
11     for adyacente in caminos["11"]:
12         distancias[adyacente] = pesos[("11",adyacente)]
13         posiblesCaminos.append(("11",adyacente))
14     w = "11"
15     recorridos[w]=True
16     continuar = True
```

```
17      anterior = None
18      while (w!=llaves[-1] and continuar == True):
19          minimo = math.inf
20          for camino in posiblesCaminos:
21              if pesos[camino]<minimo:
22                  minimo = pesos[camino]
23                  w = camino[1]
24                  anterior = camino[0]
25          recorridos[w]=True
26          posiblesCaminos.pop(posiblesCaminos.index((anterior,w)))
27
28          if minimo == math.inf:
29              continuar = False
30
31          for adyacente in caminos[w]:
32              distancias[adyacente] = min(distancias[adyacente],distancias[w] + pesos[(w
    ,adyacente)])
33              if recorridos[adyacente] == False:
34                  posiblesCaminos.append((w,adyacente))
35          if len(posiblesCaminos)==0:
36              continuar = False
37      masCorto = distancias[llaves[-1]]
38      if masCorto == math.inf:
39          masCorto = "NO EXISTE"
40      return masCorto
```

El algoritmo funciona de la siguiente manera:

1. Primero se declaran las estructuras base que irán cambiando a lo largo del algoritmo (línea 3 a 5). El diccionario distancias guardará las menores distancias entre el nodo inicial y los demás. La estructura correspondiente a los nodos recorridos pasa de ser una lista a un diccionario para poder consultar estos en O(1) y no en O(n). Este diccionario tiene como llave un nodo y como valor un booleano, True si ya fue recorrido y False si todavía no. Por último, la lista posibles caminos ahora guardará tuplas con pares de nodos que no han sido recorridos aún.

2. Posteriormente, se inicializan los diccionarios de distancias y recorridos. Distancias tendrá todos sus valores en infinito y recorridos en False indicando que ningún nodo ha sido recorrido.

3. Ahora se inicializa el algoritmo de Dijkstra con el primer nodo ("11"), se relajan los pesos de este nodo a sus adyacentes y se añaden estos caminos a la lista de posibles Caminos. Se cambia su valor en el diccionario de nodos recorridos.

4. Para este ciclo se cambiaron las condiciones para que el algoritmo continúe su ejecución. Se declaró la variable continuar para los casos en que no haya otro camino que tomar y se toma en cuenta la condición de haber encontrado el nodo destino para terminar el ciclo.

5. La forma de escoger el siguiente nodo también cambia para esta versión del algortimo. Ya no se revisan todos los posibles caminos de todos los nodos recorridos para chequear que no hayan sido recorridos y compararlos con el mínimo. Sino que ahora, se toma en cuenta únicamente la lista de posibles caminos que no será tan grande y se recorrerá una sola vez. Se encuentra el menor de estos caminos y se toma su nodo de llegada como el siguiente nodo a recorrer. Se guarda también el nodo de salida.

6. Cuando se termina de escoger el siguiente nodo a recorrer, se cambia su valor a True en el diccionario de nodos recorridos. Se quita el camino que se ha recorrido de los posibles caminos. Esto es de suma importancia, pues si no se hace se pueden crear bucles en la ejecución y el algoritmo falla.

7. Si después de haber comparado los posibles caminos a recorrer el mínimo no cambia, quiere decir que no hay más caminos a recorrer y se termina el ciclo.

8. Se toman en consideración los nodos adyacentes al nodo seleccionado y se relajan los pesos del diccionario distancias si es el caso. Para cada nodo se mira si ha sido recorrido o no. Si no ha sido recorrido, se agrega a la lista de posibles caminos para ser tomado en cuenta en la siguiente iteración.

9. Si la lista de posiblesCaminos tiene cero elementos, quiere decir que que ya no hay caminos más para recorrer y el ciclo debe terminar.

10. Finalmente, se retorna el resultado.

2 Análisis de complejidad espacial y temporal

2.1 Algoritmo 1.1

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	$2n^3$
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.

2. Las operaciones get y `append()` son $O(1)$ en los diccionarios, luego las podemos considerar como constantes.

3. La operación in es O(n) en su peor caso y este caso ocurre relativamente seguido. Ya que está dentro de otros 2 ciclos, se ejecutaría en el orden de n^3 veces.

4. 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: $T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 = 3c + 7cn + 5cn^2 + 2cn^3$. Donde c es la suma de las constantes correspondientes. En notación big O es $O(n^3)$

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 Algoritmo 1.2

Se consideran las siguientes operaciones para el cálculo de la complejidad temporal.

Operación	Constante	Veces
Asignación (=)	c_1	$11 + 5n + 4n^2$
Suma (+)	c_2	n^2
Comparación (`!=`,`==`)	c_3	$1 + 4n + 2n^2$
In (`in`)	c_4	n^2
Append (`.append()`)	c_5	n
Get (`[index]`)	c_6	$n + n^2$

Aspectos a resaltar:

1. Las operaciones get y `append()` son $O(1)$ en los diccionarios, luego las podemos considerar como constantes.
2. 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 $O(1)$.
3. La función .Keys() es O(n), pero no tiene efecto sobre la complejidad temporal de este algoritmo en su peor caso.

Luego tenemos la siguiente ecuación para el orden de complejidad temporal: $T(n) = c_1 + c_2 + c_3 + c_4 + c_5 + c_6 = 12c + 11cn + 9cn^2$ Que en notación big O es $O(n^2)$

Se observa un aumento en la cantidad de operaciones utilizadas, pero también una reducción considerable en el tiempo de ejecución del algoritmo.

2.3 Creación de las estructuras de datos

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

```
numero_casos = int(sys.stdin.readline())
for __ in range(numero_casos):
    pesos = {}
    caminos = {}

    linea1 = sys.stdin.readline().split()
    pisos = int(linea1[0])
    cuartos = int(linea1[1])
    portales = int(linea1[2])
    energias =  sys.stdin.readline().split()

    for piso in range(1,pisos+1):
        for cuarto in range(1,cuartos):
            primerCuarto = str(piso)+str(cuarto)
            cuartoAdy = str(piso)+str(cuarto+1)
            pesos[(primerCuarto,cuartoAdy)] = int(energias[piso-1])
            pesos[(cuartoAdy,primerCuarto)] = int(energias[piso-1])
            listaCaminosPrimero = []
            listaCaminosPrimero.append(cuartoAdy)
            listaCaminosAdy = []
            listaCaminosAdy.append(primerCuarto)
            if primerCuarto not in caminos:
                caminos[primerCuarto] = listaCaminosPrimero
            else:
                caminos[primerCuarto].extend(listaCaminosPrimero)
            caminos[cuartoAdy] = listaCaminosAdy

    for p in range(portales):
        portales = sys.stdin.readline().split()
        inicioPortal = str(portales[0])+str(portales[1])
        finPortal = str(portales[2])+str(portales[3])
        pesos[(inicioPortal,finPortal)] = 0
        caminos[inicioPortal].append(finPortal)
    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 Djikstra 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.