

# Teoría de Algoritmos 1 [75.29/95.06]

## TP2

Gonzalo Sabatino	104609
Maria Paula Bruck	107533
Kevin Javier Torrez Maldonado	101698
Mateo Capón Blanquer	104258

## Índice

<b>1. Ejercicio 1</b>	<b>2</b>
1.1. Ítem 1 . . . . .	3
1.2. Ítem 2 . . . . .	4
1.3. Ítem 3 . . . . .	4
1.4. Ítem 4 . . . . .	5
1.5. Ítems 5 y 6 . . . . .	13
1.6. Ítem 7 . . . . .	13
<b>2. Ejercicio 2</b>	<b>15</b>
2.1. Ítem 1 . . . . .	15
2.1.1. Subítem a . . . . .	15
2.1.2. Subítem b . . . . .	16
2.2. Ítem 2 . . . . .	18

# 1. Ejercicio 1

## Enunciado

Una empresa productora de tecnología está planeando construir una fábrica para un producto nuevo. Un aspecto clave en esa decisión corresponde a determinar dónde la ubicarán para minimizar los gastos de logística y distribución. Cuenta con  $N$  depósitos distribuidos en diferentes ciudades. En alguna de estas ciudades es donde deberá instalar la nueva fábrica. Para los transportes utilizarán las rutas semanales con las que ya cuentan. Cada ruta une dos depósitos en un sentido. No todos los depósitos tienen rutas que los conecten. Por otro lado, los costos de utilizar una ruta tienen diferentes valores. Por ejemplo hay rutas que requieren contratar más personal o comprar nuevos vehículos. En otros casos son rutas subvencionadas y utilizarlas les da una ganancia a la empresa. Otros factores que influyen son gastos de combustibles y peajes. Para simplificar se ha desarrollado una tabla donde se indica para cada ruta existente el costo de utilizarla (valor negativo si da ganancia).

Los han contratado para resolver este problema.

Han averiguado que se puede resolver el problema utilizando Bellman-Ford para cada par de nodos o Floyd-Warshall en forma general. Un amigo les sugiere utilizar el algoritmo de Johnson.

Aclaración: No existen ciclos negativos! Se pide:

1. Investigar el algoritmo de Johnson y explicar cómo funciona. ¿Es óptimo?
2. En una tabla comparar la complejidad temporal y espacial de las tres propuestas.
3. Analizar en qué situaciones una solución es mejor que otras
4. Crear un ejemplo con 5 depósitos y mostrar paso a paso cómo lo resolvería el algoritmo de Johnson.
5. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología greedy? Justifique
6. ¿Puede decirse que Johnson utiliza en su funcionamiento una metodología de programación dinámica? Justifique
7. Programar la solución usando el algoritmo de Johnson.

## Marco Teórico

Previo a la solución del ejercicio, debemos presentar dos propiedades importantes para entender el algoritmo de Johnson.

Dado un grafo dirigido  $G = (V, E)$ , con su función de pesos  $w : E \rightarrow \mathbb{R}$ , creamos un nuevo grafo  $G' = (V', E')$  junto con la nueva función  $\hat{w} : E' \rightarrow \mathbb{R}$ , donde  $V' = V \cup \{s\}$  para un nuevo vértice  $s \notin V$  y  $E' = E \cup \{(s, v) : v \in V\}$ . Reponderamos los pesos con  $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$ , con  $h$  una función cualquiera tal que  $h : V \rightarrow \mathbb{R}$ . Se define además al peso de un camino cualquiera como la suma de los pesos de las aristas que lo generan.

**La reponderación de los pesos  $w_{i,j}$  por la transformación  $\hat{w}_{i,j}$  no modifica las distancias mínimas  $\delta(i, j)$ .**

Para mostrar este lema, consideramos un camino mínimo genérico  $p = \langle v_0, v_1, \dots, v_k \rangle$ . Queremos probar que  $p$  es el camino mínimo utilizando la función de peso  $w$  si y solo si,  $p$  es el camino mínimo con la función de peso  $\hat{w}$ . Esto es equivalente a mostrar que  $w(p) = \hat{w}(p) + h(v_0) - h(v_k)$ , pues

$h(v_0)$  y  $h(v_k)$  no dependen del camino, al ser  $h$  una función que mapea los vértices a números. Mostramos esta afirmación a continuación.

$$\begin{aligned}
 \hat{w}(p) &= \sum_{i=1}^k \hat{w}(v_{i-1}, v_i) \\
 &= \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)) \\
 &= \sum_{i=1}^k w(v_{i-1}, v_i) + h(v_0) - h(v_k) \\
 &= w(p) + h(v_0) - h(v_k)
 \end{aligned} \tag{1}$$

### Producción de pesos no negativos por la reponderación

Queremos asegurar ahora que reponderar los pesos  $\hat{w}_{i,j}$  se realizará de forma que queden todos no negativos.

Extendemos la función de peso  $w(s, v) = 0$  para todo  $v \in V$ . Ya que estamos agregando un nuevo nodo  $s$ , ningún nodo del grafo original  $G$  puede llegar a  $s$ . Además,  $G'$  no contiene ciclos negativos si y solo si  $G$  no contiene ciclos negativos. Esto es fácil de observar, considerando  $v_k = v_0$  en el lema anterior.

Ahora, suponiendo que  $G$  no contiene ciclos negativos, podemos definir nuestra  $h(v) = \delta(s, v)$  para toda  $v \in V'$ . Por la desigualdad triangular,  $h(v) \leq h(u) + w(u, v)$  para todo par de vértices  $(u, v) \in E'$ . Por ende, si definimos los nuevos pesos  $\hat{w}$ , reponderando todos los pesos, tenemos que  $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$ , cumpliendo con nuestro objetivo de que todos los pesos reponderados sean no negativos.<sup>1</sup>

## 1.1. Ítem 1

El objetivo del algoritmo de Johnson es utilizar el algoritmo de Dijkstra para calcular los caminos mínimos entre todo par de nodos  $(u, v)$ . Siendo que una precondition de Dijkstra es que todos los pesos sean no negativos, se debe reponderar el digrafo. La transformación de reponderación viene dada por la función  $h$  ya definida en el marco teórico. Para poder calcular las distintas  $h(v)$ , se utiliza el algoritmo Bellman-Ford.

El algoritmo utiliza la matriz  $D = d_{i,j}$ , donde  $d_{i,j} = \delta(i, j)$ . El paso a paso del algoritmo es el siguiente:

1. Agrega el nodo  $s$  al grafo, formando el grafo  $G'$ .
2. Linkea el nodo  $s$  a todos los demás nodos de forma unidireccional, asignándole cero a sus pesos.
3. Computa el algoritmo de Bellman-Ford para hallar los caminos mínimos entre el nodo  $s$  y todos los demás nodos originales. Como observación, podemos ver que estos nuevos pesos solo podrán ser negativos o cero (puesto que  $s$  ya está conectado a cada nodo con un peso de cero).
4. Actualiza los pesos con  $\hat{w}(u, v)$  definida previamente, para cada par de vértices  $(u, v) \in G'$ .
5. Para cada vértice en  $u \in G$ , actualiza la posición  $d_{u,v}$ , recorriendo todos los faltantes vértices  $v \in G$ , utilizando Dijkstra con los nuevos pesos.
6. Retorna la matriz  $D$ , siendo  $d_{u,v}$  el camino mínimo entre los vértices  $u$  y  $v$ .

<sup>1</sup>Cormen, T; Leiserson, C; Rivest, R; Stein, C (2009). *Introduction to Algorithms. Third Edition*

El algoritmo es óptimo, puesto que utiliza dos algoritmos óptimos que en conjunto resuelven el presente problema. El hecho de que la actualización de pesos funciona para cualquier instancia del problema, asegura que se cumplan las precondiciones del algoritmo de Dijkstra, no habrá pesos negativos. Además, como la transformación mencionada preserva los caminos mínimos, los encontrados por Dijkstra serán los mismos. Vale aclarar que esto es así, solo cuando no existen ciclos negativos en el grafo, aclarado ya en el enunciado.

## 1.2. Ítem 2

A continuación mostramos en una tabla la complejidad de los tres algoritmos propuestos, si se busca el camino mínimo **para cada par de nodos** del grafo.

Algoritmos	Complejidades	
	Temporal	Espacial
Bellman-Ford	$\mathcal{O}(m \cdot n^2)$	$\mathcal{O}(n^2)$
Floyd-Warshall	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$
Johnson	$\mathcal{O}(n^2 \log(n) + m \cdot n)$	$\mathcal{O}(n^2)$

Para Bellman-Ford, la complejidad temporal es  $\mathcal{O}(m \cdot n \cdot n)$ , pues en el loop principal (el de mayor iteraciones) recorre una vez todas las aristas por cada vértice del digrafo (a través de los predecesores). Además, como se busca el camino mínimo para todo par de puntos, se ejecutará el algoritmo Bellman-Ford  $n$  veces. La complejidad espacial para el algoritmo simple mostrado en clase es de  $\mathcal{O}(n^2)$ , condicionado por la matriz OPT de  $n \times n$ . Los predecesores no tienen peso aquí porque se guardan en a lo sumo  $\mathcal{O}(n^2)$ . Sin embargo, se conocen otras implementaciones de Bellman-Ford con complejidad espacial  $\mathcal{O}(n)$ . Se tienen que almacenar  $n$  resultados del algoritmo, que se pueden guardar cada uno en  $\mathcal{O}(n)$ , si se utiliza espacio solamente para los predecesores de los respectivos camino. Por lo tanto, la complejidad temporal en el peor de los casos es  $\mathcal{O}(n^2) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$ , el primer sumando debido a los  $n$  resultados, y el segundo, debido a la complejidad espacial de una sola ejecución.

Para Floyd-Warshall la complejidad temporal es  $\mathcal{O}(n^3)$  puesto que realiza tres iteraciones concatenadas: Una por cada vértice, creando una nueva matriz  $D$ ; luego, por cada par  $(u, v)$  (doble iteración), computa el camino mínimo y lo guarda en la matriz  $D[k][u, v]$ . La complejidad espacial es  $\mathcal{O}(n^2)$  ya que se guarda una matriz  $D$  de tamaño  $n \times n$ , siendo  $n$  la cantidad de vértices.

Johnson presenta una complejidad temporal  $\mathcal{O}(n^2 \log(n) + m \cdot n)$  puesto que utiliza  $n$  veces el algoritmo de Dijkstra, que presenta una complejidad de  $\mathcal{O}(n \log(n) + m)$ . Además, ejecuta una sola vez Bellman-Ford ( $\mathcal{O}(m \times n)$ ). La complejidad espacial es  $\mathcal{O}(n^2)$  puesto que guarda una matriz en memoria de tamaño  $n \times n$ . A su vez, Dijkstra y Bellman-Ford (para un solo nodo,  $s$ ) son algoritmos que requieren  $\mathcal{O}(n)$  y  $\mathcal{O}(n^2)$  en memoria respectivamente, por ende, cuando  $n$  tiende a infinito, el mayor peso se lo lleva  $\mathcal{O}(n^2)$ .

## 1.3. Ítem 3

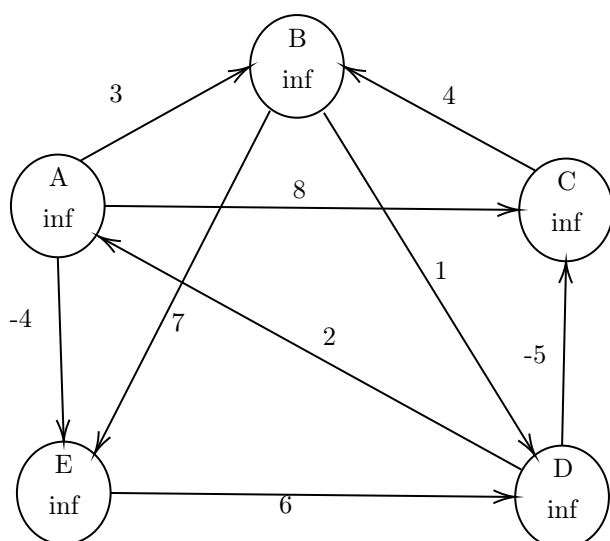
Mostramos en forma de punteros cuándo puede ser provechoso utilizar cada uno de los tres algoritmos en cuestión.

- **Bellman-Ford:** Este algoritmo es conveniente cuando el presente problema es calcular el camino mínimo desde un solo vértice a otro, o desde un vértice a todos los demás (siempre y cuando suponiendo que los pesos pueden ser negativos). Si el digrafo es disperso, esto es que  $m$  es chico, entonces este es el mejor algoritmo de los tres para resolver el problema. Sin embargo, si el digrafo es denso, es decir, que si  $m \sim 2 \cdot \frac{n}{2} \cdot n - 1$  (la cantidad máxima de aristas que puede tener), entonces la complejidad temporal será  $\mathcal{O}(n^3)$ , la misma que la de los otros dos algoritmos.

- **Floyd-Warshall:** Si bien este es el algoritmo con peor complejidad a priori, cuando la cantidad de aristas  $m$  tiende al máximo (es un grafo lo más cercano a ser completo), se puede dar una equivalencia entre los algoritmos de Floyd-Warshall y Johnson, siendo ambos  $\mathcal{O}(n^3)$ . A su vez, es un algoritmo muy fácil de implementar (más teniendo en cuenta que Johnson requiere de la inclusión de otros dos algoritmos), por ende si las complejidades son comparables, puede ser preferible utilizar este algoritmo.
- **Johnson:** Cuando se requiere buscar los caminos mínimos entre todos los vértices y el grafo es disperso, es el algoritmo más conveniente, puesto que la complejidad será  $\mathcal{O}(n^2 \log(n))$ , mucho menor que las presentadas por los otros dos algoritmos.

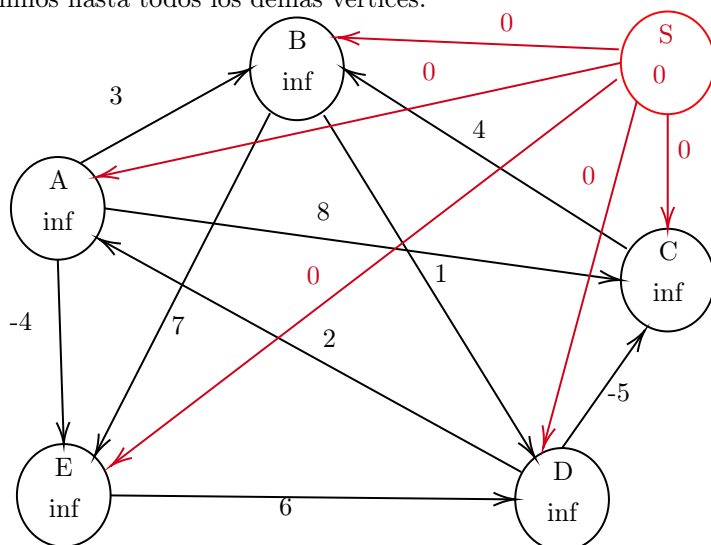
#### 1.4. Ítem 4

A continuación veremos el ejemplo del algoritmo de Johnson para cinco depósitos (A, B, C, D, E).

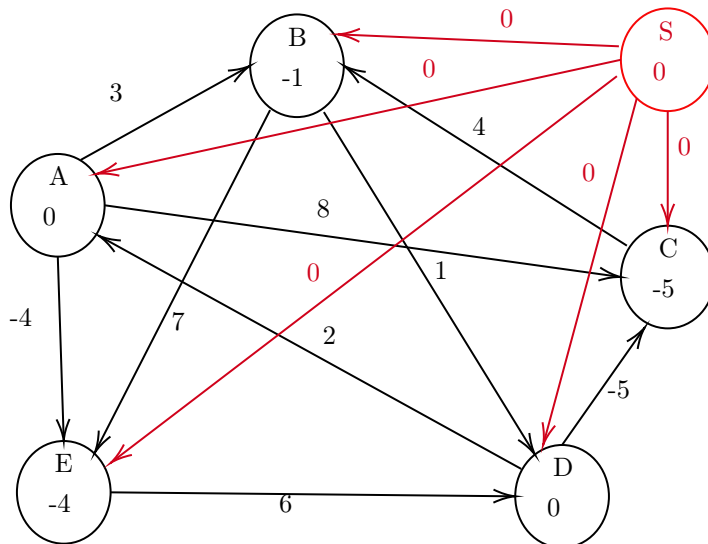


Todos los vértices contienen infinito en su función  $h(v)$ , que refleja cuál es valor del camino mínimo hasta ese vértice (se actualizará con Bellman-Ford).

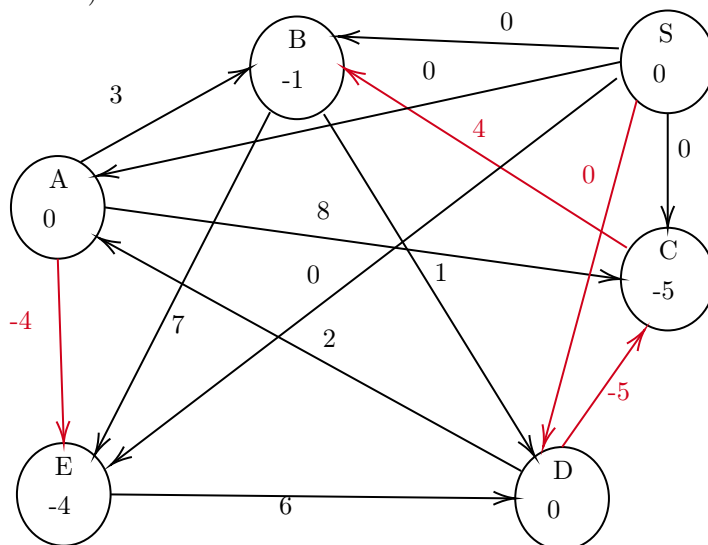
Ahora, se inserta un nuevo vértice  $S$  con peso cero tanto en su función  $h(S)$  como para los caminos hasta todos los demás vértices.



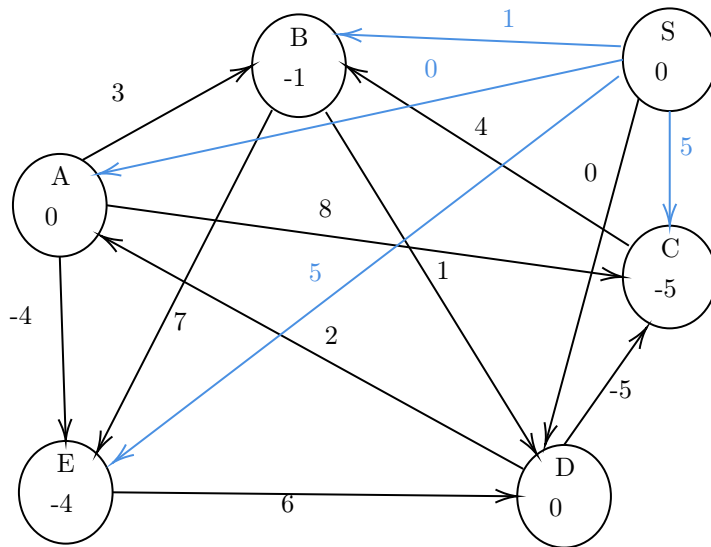
Se actualizan los pesos de todas las funciones  $h$  utilizando Bellman-Ford.



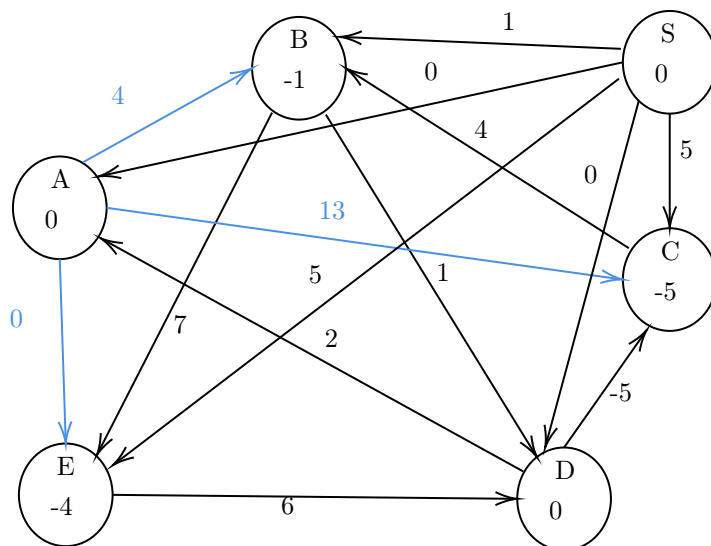
Ahora, como ya todos los vértices tienen indicada de forma correcta su función  $h$ , se procede a actualizar todos los pesos de las aristas (incluyendo las del vértice  $S$ ). Se indican en color rojo los caminos mínimos hasta los vértices (para el caso de  $B$ , se aprecia que el camino mínimo es desde  $D$  hacia  $C$ ).



Se actualizan las conexiones con  $S$ .

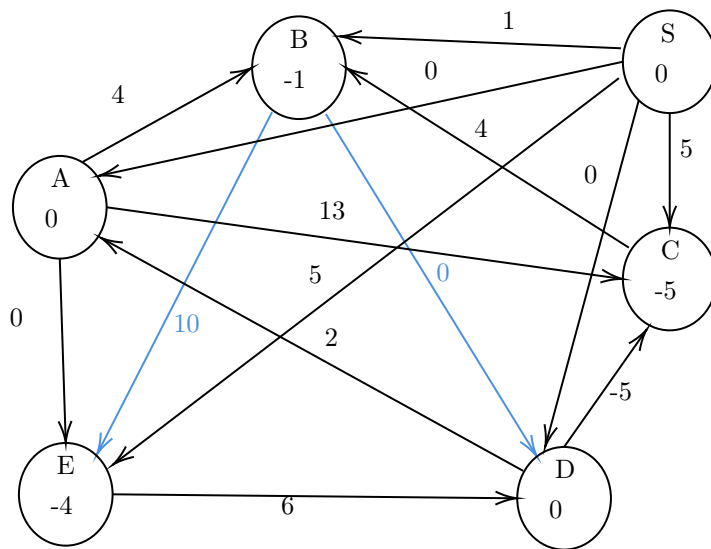


Se actualizan las conexiones con A.

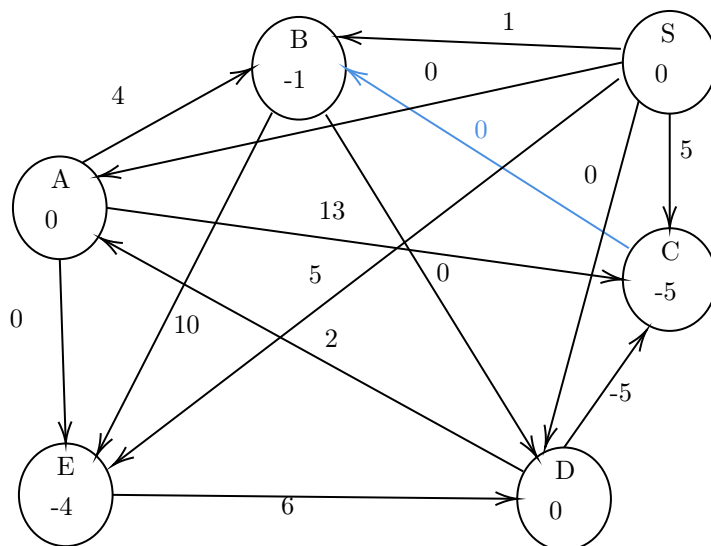


Se actualizan las conexiones con B.

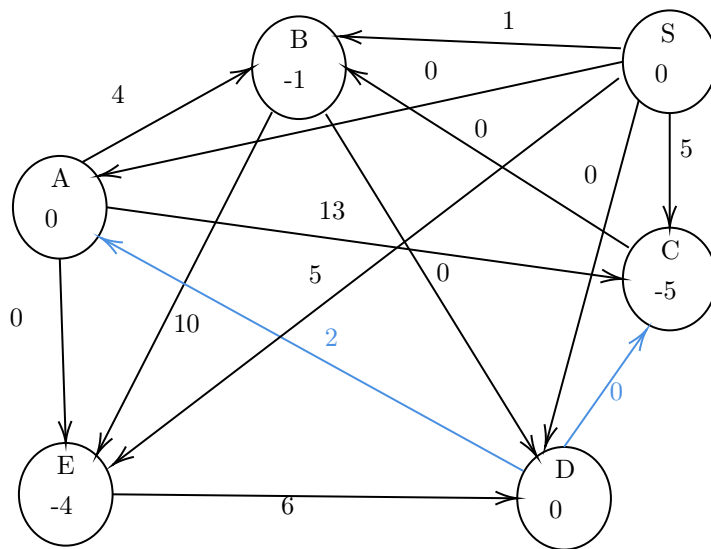




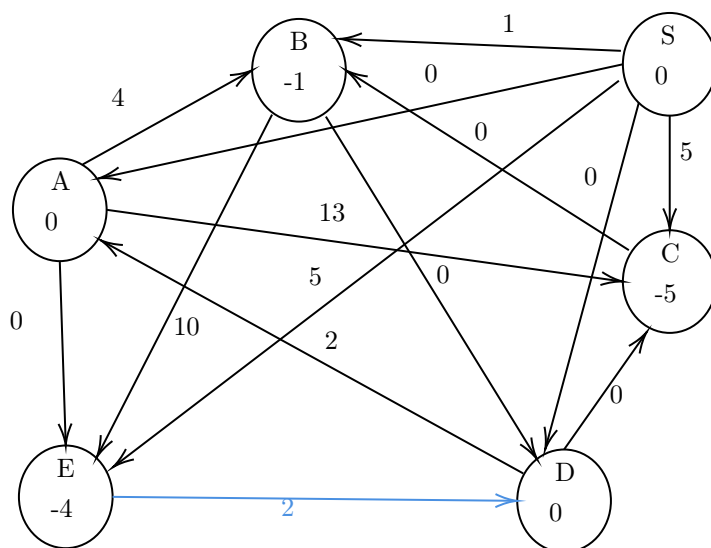
Se actualizan las conexiones con  $C$ .



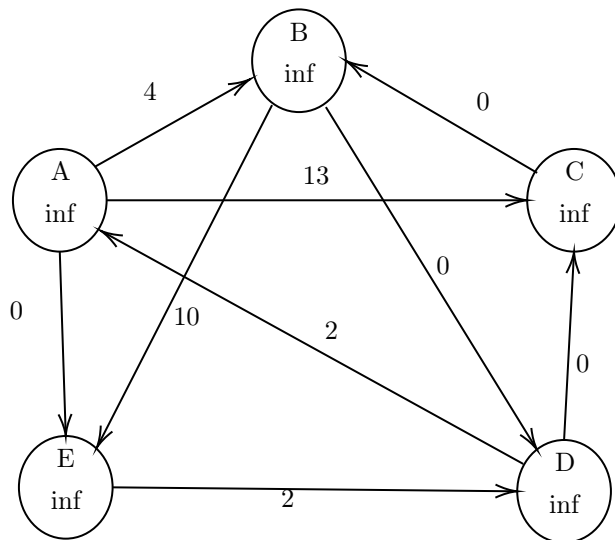
Se actualizan las conexiones con  $D$ .



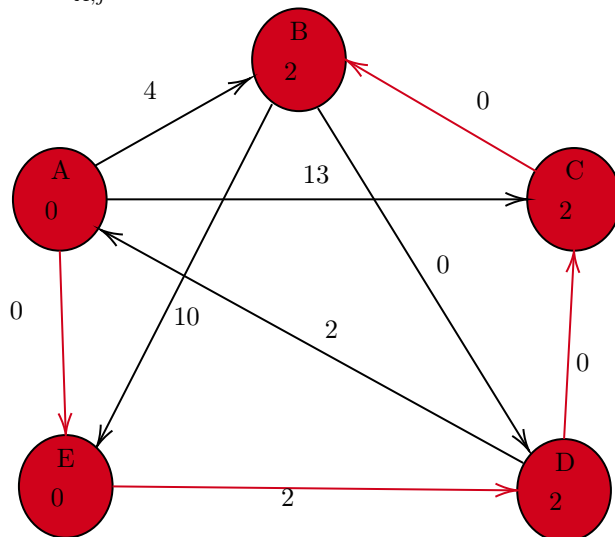
Se actualizan las conexiones con  $E$ .



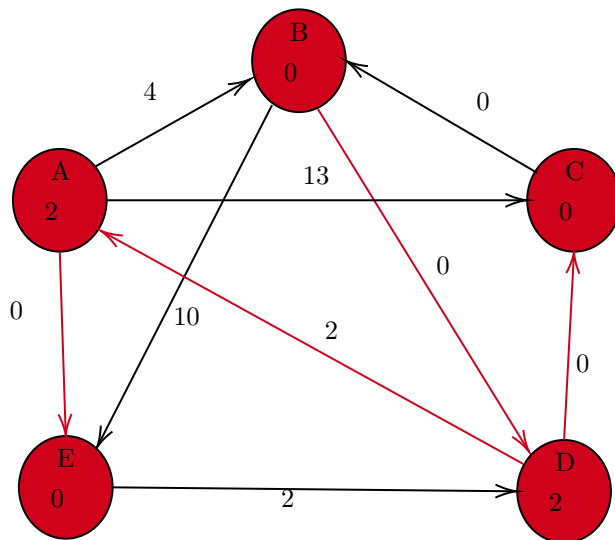
Se remueve el vértice  $S$  junto con sus conexiones.



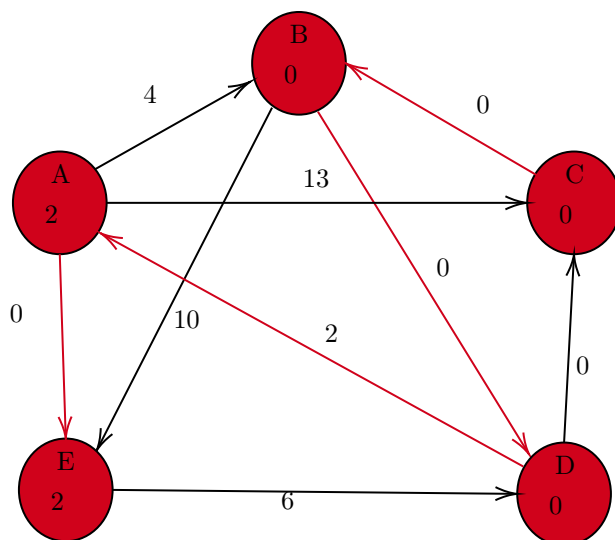
Se comienza a ejecutar el algoritmo de Dijkstra. Empezando por el vértice  $A$  se busca el camino más cercano a todas las demás. Dentro de cada vértice escribimos el valor del peso del camino mínimo  $\hat{\delta}_{A,j}$ .



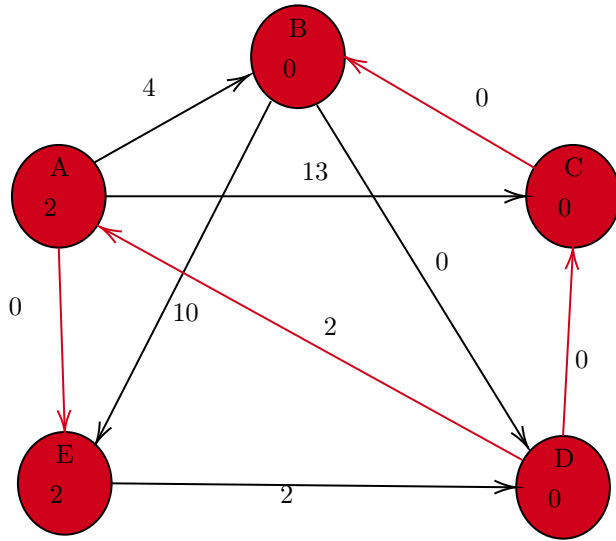
En rojo se aprecian los caminos mínimos a partir del vértice  $A$ . Se sigue con el vértice  $B$ :



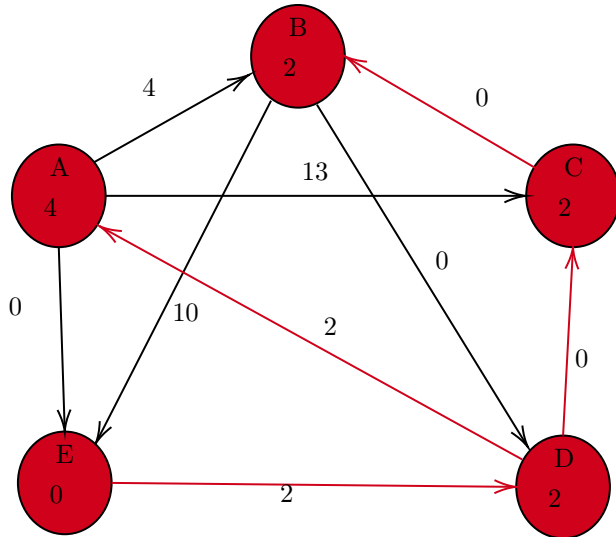
Con C:



Con D:



Con  $E$ :



Interpretamos que la locación ideal de la fábrica es aquella que minimice la suma del peso de los caminos mínimos. Por lo tanto, para elegir en qué ciudad colocarla, simplemente se suman  $\delta_{i,j} = \hat{\delta}_{i,j} - h(i) + h(j)$  de cada uno de los caminos mínimos, y se elige el mínimo de esos costos:

- $A : \hat{\delta}_{A,B} + \hat{\delta}_{A,C} + \hat{\delta}_{A,D} + \hat{\delta}_{A,E} - 4 \cdot h(A) + h(B) + h(C) + h(D) + h(E) = -4$
- $B : \hat{\delta}_{B,A} + \hat{\delta}_{B,C} + \hat{\delta}_{B,D} + \hat{\delta}_{B,E} - 4 \cdot h(B) + h(A) + h(C) + h(D) + h(E) = -1$
- $C : \hat{\delta}_{C,A} + \hat{\delta}_{C,B} + \hat{\delta}_{C,D} + \hat{\delta}_{C,E} - 4 \cdot h(C) + h(A) + h(B) + h(D) + h(E) = 19$
- $D : \hat{\delta}_{D,A} + \hat{\delta}_{D,B} + \hat{\delta}_{D,C} + \hat{\delta}_{D,E} - 4 \cdot h(D) + h(A) + h(B) + h(C) + h(E) = -6$
- $E : \hat{\delta}_{E,A} + \hat{\delta}_{E,B} + \hat{\delta}_{E,C} + \hat{\delta}_{E,D} - 4 \cdot h(E) + h(A) + h(B) + h(C) + h(D) = 20$

Por lo tanto, el costo mínimo será elegir la ciudad  $D$ .

**Respuesta del algoritmo de Johnson:** Elegir la ciudad  $D$  para colocar la fábrica, que dará un costo total para ir a cualquiera de las cinco ciudades de -6.

Vale aclarar que suponemos que se recorrerán todos los caminos de forma independiente. En otras palabras, no se busca encontrar el mejor ciclo hamiltoniano, sino que se busca minimizar la suma de los caminos mínimos desde el punto de la fábrica hacia los demás.

### 1.5. Ítems 5 y 6

Tal como mencionamos anteriormente, el algoritmo de Johnson utiliza al algoritmo de Bellman-Ford y el de Dijkstra. Por lo tanto, **utiliza en su funcionamiento las metodologías de estos dos algoritmos**, aunque no se puede decir que resuelve el problema de forma Greedy o utilizando programación dinámica.

Para resolver el problema de la actualización de los pesos, se usa el Bellman-Ford, quien a su vez utiliza una metodología de programación dinámica puesto que:

- Es un problema con una **subestructura óptima**: Contiene en su interior las soluciones óptimas de sus subproblemas, es decir, para encontrar el camino mínimo entre dos nodos  $(u, v)$ , se calculan los caminos mínimos entre los vértices intermedios.
- Presenta **problemas superpuestos**: Para encontrar el camino mínimo entre dos nodos  $(u, v)$ , se tiene que utilizar el cálculo del camino mínimo entre los nodos predecesores a  $v$  y compararlo con el camino mínimo del nodo  $u$  a  $v$ .

Además, para buscar los caminos mínimos del nuevo digrafo con pesos positivos, se vale del algoritmo de Dijkstra. Éste es un algoritmo Greedy, porque:

- Tiene **elección Greedy**: Seleccionar el mínimo camino entre los vértices de la frontera para un nodo  $u$ , acerca al óptimo global que es encontrar el camino mínimo entre los nodos  $(u, v)$ .
- Contiene **subestructuras óptimas**: Puesto que para encontrar el mínimo entre los vértices  $(u, v)$ , se empieza desde el nodo  $u$ , encontrando el mínimo entre todos los nodos de su frontera, hasta llegar al nodo  $v$ . Por ende, la solución óptima global contiene en su interior la solución óptima de sus subproblemas (llegar desde  $u$  hasta los nodos intermedios).

### 1.6. Ítem 7

A continuación se muestra un pseudocódigo resumido de lo que se encontrará en la implementación (utilizando python). Cabe notar que: para el pseudocódigo se llama a los algoritmos de Dijkstra y Bellman-Ford, sin implementarlos, aunque fueron implementados en código real para que el algoritmo funcione (y para preservar las complejidades mencionadas).

```

procedure JOHNSON(graph)
  new_graph = graph
  Add s to new_graph with weights 0
  for Every  $(u, v)$  in new_graph do
    bellman_ford(new_graph, u, v)
  end for
  for Every  $(u, v)$  in new_graph do
    new_graph.weights[u][v] = new_graph.weights[u][v] + h[u] - h[v]
  end for
  Update graph from new_graph
  Define D = [len(graph)][len(graph)]
  for Every u in graph do
    aux = dijkstra(graph, u)
    for Every v in aux do
      D[u][v] = aux[v] + h[v] - h[u]
    end for
  end for
  return D
end procedure

```

```
procedure FIND_LOCATION(graph)  
  D = Johnson(graph)  
  register = 0  
  min = inf  
  for Every vertex in graph do  
    Add up all the cost in D[vertex]  
    if Cost < min then  
      min = Cost  
      register = vertex  
    end if  
  end for  
  return register  
end procedure
```

La idea mencionada en el Ítem 4 sobre transformar el problema para hallar de forma correcta los costos y en qué ciudad conviene poner la fábrica, se ve reflejada en la línea donde se utiliza el algoritmo de Dijkstra para cada nodo, pero se le suma la función  $h$  de cada subnodo y se resta la propia  $h$  del nodo (es el camino inverso a la reponderación).

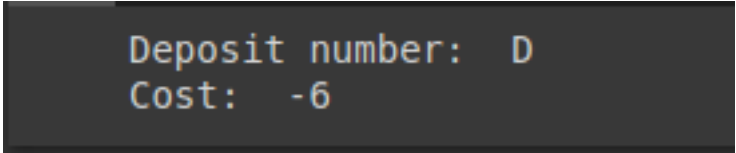
### Implementación en código

A continuación, figuran todos los pasos para poder ejecutar y probar el código entregado.  
La línea de ejecución es:

```
python johnson.py ruta_archivo/archivo.txt
```

Donde *ruta\_archivo* se puede omitir si el txt se encuentra dentro del mismo directorio que el código a ejecutar.

El resultado se mostrará de la forma:



```
Deposit number: D  
Cost: -6
```

## 2. Ejercicio 2

### Enunciado

1. Hasta el momento hemos visto 3 formas distintas de resolver problemas. Greedy, división y conquista y programación dinámica.
  - a) Describa brevemente en qué consiste cada una de ellas.
  - b) Identifique similitudes, diferencias, ventajas y desventajas entre las mismas. ¿Podría elegir una técnica sobre las otras?
2. Tenemos un problema que puede ser resuelto por un algoritmo Greedy (G) y por un algoritmo de Programación Dinámica (PD). G consiste en realizar múltiples iteraciones sobre un mismo arreglo, mientras que PD utiliza la información del arreglo en diferentes subproblemas a la vez que requiere almacenar dicha información calculada en cada uno de ellos, reduciendo así su complejidad; de tal forma logra que  $O(PD) < O(G)$ . Sabemos que tenemos limitaciones en nuestros recursos computacionales (CPU y principalmente memoria). ¿Qué algoritmo elegiría para resolver el problema?

Pista: probablemente no haya una respuesta correcta para este problema, solo justificaciones correctas

### 2.1. Ítem 1

#### 2.1.1. Subítem a

**Greedy:** Los algoritmos del tipo Greedy son también conocidos como algoritmos codiciosos. Se utilizan generalmente como metodología de resolución de problemas de optimización, tanto para obtener el mínimo como el máximo. Suelen ser fáciles y rápidos de implementar, toman las decisiones en función de la información que está disponible en cada momento y una vez que ésta ha sido tomada no vuelve a replantearse su futuro.

Para poder considerarse como un algoritmo Greedy, el problema debe contener:

- Elección Greedy: La solución de un óptimo local acerca a la solución del óptimo global.
- Subestructura óptima: La solución óptima global del problema a resolver debe contener las soluciones óptimas de sus subproblemas.

No siempre está garantizado alcanzar la solución óptima. Por lo tanto, siempre habrá que estudiar la corrección del algoritmo para demostrar si las soluciones obtenidas son óptimas o no.

**División y Conquista:** Los algoritmos del tipo de división y conquista son también conocidos como divide y vencerás, estos algoritmos son un conjunto de técnicas algorítmicas en el que se divide el problema en subproblemas de igual naturaleza y menor tamaño, se los conquista (resuelve) en forma recursiva hasta llegar a un caso base y se combinan los resultados en una solución general. Se suelen representar en forma de árbol, donde la raíz representa al problema en sí, y cada uno de los nodos, representan a los subproblemas. Las hojas simbolizan a los casos base. Generalmente, se pueden aplicar a problemas donde la solución por fuerza bruta ya tiene complejidad polinómica.

Para poder analizar su complejidad se requiere una relación de recurrencia.

**Programación Dinámica:** Es una metodología de resolución de problemas de optimización nombrada por Richard Bellman en 1950, su función es dividir el problema en subproblemas con una jerarquía entre ellos y cada subproblema puede ser reutilizado en diferentes subproblemas mayores.

Para poder resolverse con esta metodología un problema debe contener:



- Subestructura óptima: Igual que para los algoritmos Greedy.
- Subproblemas superpuestos: En la resolución de subproblemas, vuelven a aparecer subproblemas previamente calculados.

Se pueden resolver de manera recursiva y mediante la memorización que es una técnica que consiste en almacenar los resultados de los subproblemas previamente calculados con la finalidad de evitar repetir su resolución cuando se vuelva a requerir, de esta forma se reduce la cantidad total de subproblemas a calcular consiguiendo reducir significativamente la complejidad temporal de la solución.

#### **2.1.2. Subítem b**

A continuación, se muestra una tabla comparando los tres algoritmos

	GREEDY	DIVISIÓN Y CONQUISTA	PROGRAMACIÓN DINÁMICA
DIFERENCIAS	<ul style="list-style-type: none"> <li>- Las decisiones se llevan a cabo en base a cada óptimo local, tratando de acercarse, de esta manera, al óptimo global.</li> </ul>	<ul style="list-style-type: none"> <li>- Las soluciones de los subproblemas descompuestos por el problema se pueden combinar en la solución del problema.</li> <li>- Los subproblemas son resueltos de forma independiente, es decir, no tiene guardado un resultado anterior.</li> <li>- Mergea los resultados de los subproblemas anteriores.</li> <li>- Resolución recursiva, resuelve subproblemas de forma independiente.</li> <li>- Estructura de subproblema diferente para cada subproblema.</li> </ul>	<ul style="list-style-type: none"> <li>- Los subproblemas no son resueltos de forma independiente (interesa el resultado del subproblema menor).</li> <li>- En cada paso evalúa la solución considerando tanto las soluciones actuales como las anteriores para obtener la solución óptima.</li> <li>- Resuelve redundancias, utiliza los resultados de subproblemas anteriores para cálculos similares (problemas superpuestos).</li> <li>- Muchas subpreguntas están duplicadas (no son independientes).</li> </ul>
SIMILITUDES	<ul style="list-style-type: none"> <li>- Divide el problema en subproblemas.</li> </ul>	<ul style="list-style-type: none"> <li>- Divide el problema en subproblemas.</li> <li>- Se puede representar de forma recursiva.</li> </ul>	<ul style="list-style-type: none"> <li>- Divide el problema en subproblemas.</li> <li>- Se puede representar de forma recursiva.</li> </ul>
VENTAJAS	<ul style="list-style-type: none"> <li>- No es muy difícil construir una estrategia greedy, y si se demuestra su optimalidad, es un algoritmo eficiente.</li> <li>- Son eficientes en memoria.</li> </ul>	<ul style="list-style-type: none"> <li>- Proporciona una forma natural de diseñar algoritmos eficientes.</li> <li>- Es una herramienta potente para solucionar problemas complejos.</li> <li>- Se adapta de forma natural a la ejecución en entornos multiprocesador.</li> <li>- El paradigma de Divide y vencerás tiende naturalmente a hacer un uso eficiente de las memorias cachés.</li> <li>- En computaciones con aritmética redondeada podría dar resultados más exactos que un problema iterativo equivalente superficialmente.</li> </ul>	<ul style="list-style-type: none"> <li>- Resuelve eficientemente muchos problemas que no pueden resolverse con algoritmos greedy o algoritmos de divide y vencerás.</li> <li>- Para tareas de optimización (hallar máximos y mínimos) resulta ser una de las mejores opciones.</li> <li>- Acelera el procesamiento, ya que se usan referencias que fueron previamente calculadas.</li> <li>- Encuentra la solución óptima para los subproblemas, luego hace una elección informada combinando los resultados de esos subproblemas para encontrar realmente la solución más óptima.</li> </ul>
DESVENTAJAS	<ul style="list-style-type: none"> <li>- Al no almacenar valores calculados previamente para uso futuro, no asegura una solución eficiente.</li> <li>- Hay que trabajar mucho más para seleccionar una heurística correcta, que demuestre resolver el problema.</li> </ul>	<ul style="list-style-type: none"> <li>- Si los subproblemas no son independientes, debe buscar repetidamente subproblemas comunes (conviene en ese caso usar programación dinámica).</li> <li>- Lentitud en la repetición del proceso recursivo.</li> <li>- Inconveniencia de aplicar el método a situaciones en las que la solución al problema general no se deriva de la suma directa y simple de los subproblemas.</li> <li>- No todas las soluciones recursivas, tienen forma equivalente iterativa. Puede entonces generar stack overflow.</li> </ul>	<ul style="list-style-type: none"> <li>- Se puede necesitar mucha memoria para almacenar el resultado calculado de cada subproblema, sin poder garantizar que el valor almacenado se utilizará o no.</li> <li>- Funciones con llamadas recursivas haciendo que la memoria de pila se mantenga en constante aumento (si se elige la estrategia recursiva)..</li> </ul>

En cuanto a la pregunta de si podríamos elegir una de las tres formas como *mejor*, considerando los pros y contras de cada una, no podríamos seleccionar una. Siempre se debe analizar el problema en cuestión, teniendo en cuenta la complejidad temporal y espacial de cada solución propuesta, y comparándolas con las características de la máquina con la que se trabaja. Más aún, pueden existir diferentes problemas a los que no se les conoce la solución utilizando un método, y sí se conoce un algoritmo con otro.

## 2.2. Ítem 2

El principal problema que se presenta en este punto es qué elegir: Memoria o CPU.

El algoritmo Greedy (G) requiere una complejidad mayor que el algoritmo de Programación Dinámica (PD), pero a su vez, el algoritmo de programación dinámica (para lograr reducir la complejidad) debe ir almacenando la solución cada subproblema.

La respuesta, entonces, dependerá de cuál sean las características del problema y con qué limitaciones reales se cuentan.

### Información con la que se cuenta de antemano

Por las características del enunciado, se puede apreciar que:

- El algoritmo Greedy tiene una complejidad espacial de  $\mathcal{O}(n)$ , puesto que utiliza solo un arreglo en memoria para resolver el problema.
- Ambos algoritmos serán de tiempos polinómicos, en otro caso, ambos se consideran como intratables, por ende no se seleccionará ninguno.
- $\mathcal{O}(PD) < \mathcal{O}(G)$

### Elección del algoritmo

La verdadera elección dependerá de cuál es la característica del problema, y cuáles son las limitaciones reales con las que se cuentan.

Como ya sabemos que la complejidad temporal del algoritmo de Programación Dinámica es menor que el algoritmo Greedy, será muy importante evaluar qué tan diferentes son las complejidades, y qué recurso se quiere primar.

1. Si alguna de las diferencias de las complejidades (espacial y temporal) es mucho mayor a la otra, entonces deberá ser elegido el método que permita la mínima complejidad. Es decir, en caso de que la diferencia entre las complejidades espaciales sea mucho mayor que la diferencia entre las complejidades temporales, se elige el algoritmo Greedy puesto que contamos con limitaciones de ambos recursos, y es el que mejor logra preservar uno de ellos (memoria).
2. Si el algoritmo PD logra resolver el problema en una complejidad espacial  $\mathcal{O}(n)$  (por ejemplo, en problemas donde solo se debe guardar el óptimo dado un vértice), entonces el algoritmo de Programación Dinámica será el que tendremos que seleccionar por sobre el algoritmo Greedy.
3. Si el algoritmo PD presenta complejidades espaciales mayores que  $\mathcal{O}(n)$  (por ejemplo, en *Seam Carving* o *el problema de Knapsacks*), entonces habrá que analizar qué diferencia prima en las complejidades, y con qué recursos se cuenta:
  - a) Si la complejidad espacial de PD depende de otra constante  $\mathcal{O}(C * n)$ , en este caso se podrá pensar qué instancias del problema estamos resolviendo, es decir, sabiendo de antemano que las instancias de un problema particular determinan que  $C \ll n$ , entonces se elige PD como en el caso anterior. Si contamos de antemano con la información de que esa constante se asemeja a  $n$  o no contamos con dicha información, se cae en un caso de indecisión (desarrollado a continuación).
4. Llamamos a cualquier otro escenario posible donde la elección no es clara como un **caso de indecisión**. Cabe destacar que en estos casos, la diferencia entre complejidades no será demasiado grande:

- a) Si la diferencia de las complejidades espaciales entre PD y G es mayor que las complejidades temporales, entonces se deberá elegir qué recurso se quiere preservar, teniendo en cuenta que la elección del algoritmo Greedy logra preservar de una forma mejor la memoria que lo que logra Programación Dinámica con la CPU. Es decir, será más probable la elección de un algoritmo Greedy.
- b) Será análogo en caso de que la diferencia entre las complejidades temporales sea mayor que la diferencia entre las complejidades espaciales.
- c) Si las diferencias de complejidades son muy similares, entonces prima en mayor medida qué recurso se quiera lograr consumir en menor medida (por ejemplo, si se cuenta con gran espacio de almacenamiento, se elige PD contando con tiempos menores que G).

Otro procedimiento que se podría seguir para determinar qué algoritmo utilizar, son las pruebas empíricas. Muchos algoritmos tienen complejidad menor que otros, lo cual hace que se los considere mejores en la teoría. Sin embargo, llevado a la práctica, existen veces que esta diferencia de complejidad es casi imperceptible. Un ejemplo al respecto son los distintos algoritmos que se presentaron en la teórica para resolver multiplicaciones de números de orden altísimo. Teniendo esto en cuenta, en función de decidir qué algoritmo utilizar, se podrían implementar ambos (suponiendo que esto no conlleva un costo extra), y probarlos variando distintos parámetros de entrada. En el caso de algoritmos con grafos, se podrían hacer tests con grafos variando la cantidad de vértices  $n$  y la cantidad de aristas  $m$ , utilizando  $n$  y  $m$  similares o mayores a los valores reales que aparecerán en los problemas a resolver. Luego, se tendría que hacer un balance entre el consumo de CPU y memoria, y el tiempo empleado en resolver cada problema, para poder decidir qué algoritmo utilizar.

Estas pruebas, desde ya, se harían si son accesibles con el problema a tratar. Puede suceder que estos algoritmos tarden días (o semanas) en correr, y por lo tanto, no sea una buena idea, ejecutar muchos tests para tomar una decisión.

La elección, entonces, dependerá (como en todo problema informático) de las condiciones del problema y de los recursos con los que se cuentan.

## Referencias

- [1] Cormen, T; Leiserson, C; Rivest, R; Stein, C (2009). *Introduction to Algorithms. Third Edition*.
- [2] Kleinberg, J; Tardos, É (2006). *Algorithm Design*.
- [3] Videos de cátedra <https://www.youtube.com/user/vpode>.