Trabajo Práctico 2

[7529/9506] Teoría de Algoritmos I
Segundo cuatrimestre de 2021

Grupo:	404
Repositorio:	github.com/lucashemmingsen/7529tp2
Entrega:	nº 1 (20/10/2021)

Integrantes del grupo 404

Padrón	Apellido, Nombre	Email
76187	Xxxxxxxxxx, Xxxxx Xxxxx	xxxxxxxxxxx@fi.uba.ar
97529	Xxxxxxxx, Xxxxx Xxxxxxx	xxxxxxxxx@fi.uba.ar
102593	Xxxxxx, Xxxxx Xxxxx	xxxxxxx@fi.uba.ar
102649	Xxxxxx, Xxxxxxx Xxxxx	xxxxxxx@fi.uba.ar
106713	Xxxx Xxxxx, Xxxxx	xxxxx@fi.uba.ar

Índice

I. Introducción

I.1. Resumen

El presente informe documenta el enunciado y la solución del segundo trabajo práctico de la materia Teoría de Algoritmos I. El mismo comprende el análisis del problema planteado, la comparación de posibles algoritmos y la resolución manual del algoritmo de Johnsosn; así como también respuestas a preguntas teóricas.

I.2. Lineamientos básicos

- El trabajo se realizará en grupos de cinco personas.
- Se debe entregar el informe en formato pdf y código fuente en (.zip) en el aula virtual de la materia.
- El lenguaje de implementación es libre. Recomendamos utilizar C, C++ o Python. Sin embargo si se desea utilizar algún otro, se debe pactar con los docentes.
- Incluir en el informe los requisitos y procedimientos para su compilación y ejecución. La ausencia de esta información no permite probar el trabajo y deberá ser re-entregado con esta información.
- El informe debe presentar carátula con el nombre del grupo, datos de los integrantes y y fecha de entrega. Debe incluir número de hoja en cada página.
- En caso de re-entrega, entregar un apartado con las correcciones mencionadas

P1. Parte 1: Minimizando costos

a. 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.

b. Formato de los archivos

Formato de los archivos: El programa debe recibir por parámetro el path del archivo donde se encuentran los costos entre cada depósito. El archivo debe ser de tipo texto y presentar por renglón, separados por coma un par de depósitos con su distancia.

Ejemplo: “depositos.txt”

```
A,B,54
A,D,-3
B,C,8
...
```

Debe resolver el problema y retornar por pantalla la solución. Debe mostrar por consola en en que ciudad colocar el depósito. Además imprimir en forma de matriz los costos mínimos entre cada uno de los depósitos.

1. Algoritmo de Johnson: funcionamiento

Enunciado P1.1

Investigar el algoritmo de Johnson y explicar cómo funciona. ¿Es óptimo?

El algoritmo de Johnson determina los caminos minimales entre todos los pares de vértices de un grafo dirigido. Es preferible aplicarlo sobre grafos de escasas aristas, ya que los procesa con mayor rapidez que otros algoritmos de símil uso.

Su funcionamiento se basa en reponderaciones del grafo. Además, utiliza los algoritmos de Dijkstra y Bellman-Ford como subrutinas.

La reponderación se ejecuta condicionalmente. Si todos los pesos de las aristas son positivos, no es necesaria una reponderación, basta con ejecutar Dijkstra una vez por vértice y devolviendo el resultado final. En cambio, si el grafo (G) tiene aristas de peso negativo, pero no ciclos negativos, computamos un nuevo set de aristas con pesos no negativos que nos permita utilizar Dijkstra, como en el caso anterior. A este proceso de cómputo de nuevas aristas se lo llama reponderación, o cambio de peso.

El set de aristas obtenido de la reponderación debe cumplir dos propiedades:

I Se preservan los caminos más cortos.

II No hay aristas de peso negativo.

El proceso de reponderación tiene un costo de $O(V \times E)$.

Dado que la reponderación asegura que no quedan aristas de peso negativo, se procesa por Dijkstra como en el caso anterior. El último caso se da cuando el grafo tiene un ciclo negativo.

En instancias de este tipo, el algoritmo reporta que el grafo contiene dicho ciclo y termina.

Optimalidad del algoritmo de Johnson

Supongamos una solución J para la instancia I, la cuál consta de un grafo del cual deseamos saber todos los pesos de los caminos minimales entre todos sus vértices. J es una solución obtenida mediante el algoritmo de Johnson. J será óptima si su matriz resultante, la obtenida al finalizar el algoritmo de Johnson, contiene a todos los pesos de los caminos minimales del grafo de I.

Para hallar los caminos minimales, el algoritmo de Johnson, si decide que puede hacerse, aplica el algoritmo de Djikstra sobre cada vértice del grafo y registra el resultado de cada uno de sus usos en una matriz. Dado que el algoritmo de Djikstra asegura la obtención óptima de caminos minimales, [1] el algoritmo de Johnson entonces guarda los pesos de dichos caminos minimales.

En los casos en los que Djikstra no es aplicable, y el grafo de I no contiene ciclos negativos, Johnson aplica un reponderamiento sobre las aristas, brevemente explicado anteriormente, el cual permite la aplicación de Djikstra. Esto ya fue probado que resulta en que el algoritmo de Johnson guardará los pesos de los caminos minimales.

Por último, el caso del grafo de I con ciclo negativo, es redundante para nuestro problema, ya que simplemente ejecutaríamos un loop infinito sobre dicho ciclo para obtener ganancias infinitas. Tarde o temprano esto se vuelve inviable en una situación real. En este caso, el algoritmo de Johnson devuelve un mensaje alertando sobre el ciclo negativo, ya que no es aplicable sobre un grafo de estas características.

En conclusión, el algoritmo de Johnson es óptimo, ya que retorna el resultado esperado, gracias a el uso de otros algoritmos óptimos a su vez.

[1] Cormen tercera edición, pg 659-660

2. Comparación de algoritmos: Complejidad

Enunciado P1.2

En una tabla comparar la complejidad temporal y espacial de las tres propuestas.

Algoritmo	Temporal	Espacial
Bellman-Ford	$O(V^2 \times E)$	$O(V + E)$
Floyd-Warshall	$\Theta(V^3)$	$O(V^2)$
Johnson	$O(V^2 lg(V) + V \times E)$	$O(V^2 + V + E)$

Cuadro 1: Complejidades. V=vértices, E=aristas.

La complejidad temporal es $O(V \times E)$ por vértice de origen; sin embargo a los fines de comparación se debe tomar en cuenta la complejidad para analizar todo el grafo (cada vértice como origen), lo que lo convierte en $O(V^2 \times E)$.

3. Comparación de algoritmos: Ventajas y desventajas

Enunciado P1.3

Analizar en qué situaciones una solución es mejor que otras.

El algoritmo de Johnson puede ser más rápido que el de Floyd-Warshall en el caso de grafos dispersos (pocas aristas), pero el de Floyd-Warshall es más rápido cuando el grafo es denso (muchas aristas). La razón por la cual el algoritmo de Johnson es mejor para grafos dispersos es debido a que su complejidad temporal depende de la cantidad de aristas en el grafo, mientras que el de Floyd-Warshall no.

El algoritmo de Johnson tiene una complejidad temporal de $O(V^2.log(V) + |V|.|E|)$. Por ende, si la cantidad de aristas es pequeña, correrá más rápido que con el tiempo $O(V^3)$ del de Floyd-Warshall.

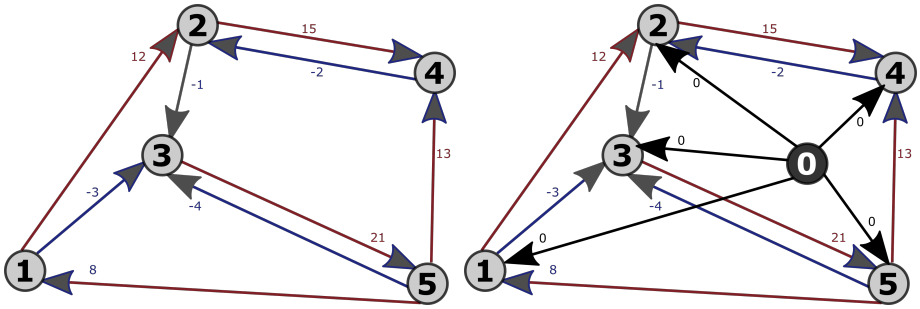
4. Algoritmo de Johnson: resolución manual

Enunciado P1.4

Crear un ejemplo con 5 depósitos y mostrar paso a paso cómo lo resolvería el algoritmo de Johnson.

4.1. Convertir a grafo sin negativos (Bellman-Ford)

Para el ejemplo tomaremos un grafo G arbitrario (figura 1a). Como tiene pesos negativos, a partir de éste, creamos un grafo G' (figura 1b) en principio idéntico pero al que le agregamos temporalmente un vértice V_0 con aristas de peso 0 hacia cada otro nodo preexistente.



(a) **Grafo G (original)** (b) **Grafo G' (con nodo temporal 0)**

A partir de este grafo, aplicamos el algoritmo de Bellman-Ford tomando como origen el vértice temporal V_0. Para el mismo, en cada iteración, recorreremos todas las aristas haciendo una reducción; en términos generales:

> Siendo $h(x)$ la distancia al origen estimada para el vértice x, y $w(u, v)$ el peso original de la arista que va desde el vértice u hasta v, comparamos $h(u)+w(u, v)$ contra $h(v)$. Si el primer valor es inferior al segundo, actualizamos $h(v) := h(u) + w(u, v)$ y anotamos u como padre de v: $\pi(v) := u$; esto es, qué otro vértice se tomó para calcular $h(v)$.

Iteramos estas reducciones sobre todas las aristas, un total de $|V| - 1$ veces, y luego una vez más para comprobar que no haya algún ciclo negativo (que no haya cambios). Adicionalmente, es posible finalizar con un resultado correcto (es decir, no hay ciclos) cuando durante una iteración externa no se produjo cambio alguno. [2]

[2] Si no hay cambios en la iteración, los datos de entrada de la siguiente iteración serán los mismos; por lo que el resultado también será el mismo (y seguirá sin haber cambios). Si bien no modifica big O, puede reducir el tiempo real de cómputo.

Cuadro 2: Primera iteración de Bellman-Ford en el algoritmo de Johnson

Fórmula	1ª iteración			V_1		V_2		V_3		V_4		V_5	
posible nuevo	Nuevo	$\lessgtr$	Ant.	h	p	h	p	h	p	h	p	h	p
$0 \to 1 \ldots 0 \to 5$	$0+0$	$<$	0	0	V_0	0	V_0	0	V_0	0	V_0	0	V_0
$h(1)+w(1,2)$	$0+12$	$<$	0	0	V_0	0	V_0	0	V_0	0	V_0	0	V_0
$h(1)+w(1,3)$	$0+(-3)$	$<$	0	0	V_0	0	V_0	-3	V_1	0	V_0	0	V_0
$h(2)+w(2,3)$	$0+-1$	$>$	-3	0	V_0	0	V_0	-3	V_1	0	V_0	0	V_0
$h(2)+w(2,4)$	$0+15$	$>$	0	0	V_0	0	V_0	-3	V_1	0	V_0	0	V_0
$h(3)+w(3,5)$	$-3+21$	$>$	0	0	V_0	0	V_0	-3	V_1	0	V_0	0	V_0
$h(4)+w(4,2)$	$0+(-2)$	$<$	0	0	V_0	-2	V_4	-3	V_1	0	V_0	0	V_0
$h(5)+w(5,1)$	$0+8$	$>$	0	0	V_0	-2	V_4	-3	V_1	0	V_0	0	V_0
$h(5)+w(5,3)$	$0+(-4)$	$<$	-3	0	V_0	-2	V_4	-4	V_5	0	V_0	0	V_0
$h(5)+w(5,4)$	$0+13$	$>$	0	0	V_0	-2	V_4	-4	V_5	0	V_0	0	V_0

Cuadro 3: Segunda iteración de Bellman-Ford en el algoritmo de Johnson

Fórmula	2ª iteración			Resultados
posible nuevo	Nuevo	$\lessgtr$	Ant.	
$0 \to 1 \ldots 0 \to 5$	$0+0$	$\geq$	h(x)	Sin cambios
$h(1)+w(1,2)$	$0+12$	$>$	-2	Sin cambios
$h(1)+w(1,3)$	$0+(-3)$	$>$	-4	Sin cambios
$h(2)+w(2,3)$	$-2+(-1)$	$>$	-4	Sin cambios
$h(2)+w(2,4)$	$-2+15$	$>$	0	Sin cambios
$h(3)+w(3,5)$	$-4+21$	$>$	0	Sin cambios
$h(4)+w(4,2)$	$0+(-2)$	$>$	-2	Sin cambios
$h(5)+w(5,1)$	$0+8$	$>$	0	Sin cambios
$h(5)+w(5,3)$	$0+(-4)$	$>$	-4	Sin cambios
$h(5)+w(5,4)$	$0+13$	$>$	0	Sin cambios

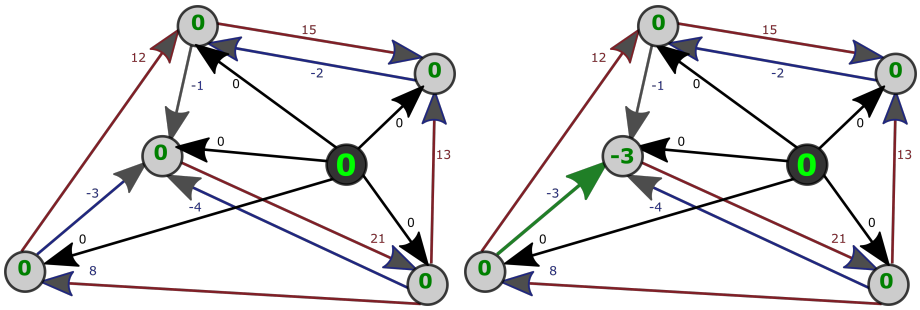
Como puede observarse en el Cuadro 2, en las primeras reducciones simplemente se actualiza la distancia estimada hasta cada vértices como `0`; esto siempre es así, por definición. [3] Por esto lo hemos marcado en una sola fila (véase figura 2a). En las siguientes reducciones, sólo hay tres en las que se realiza una actualización:

$V_1 \overset{-3}{\to} V_3$ (figura 2b) El vértice V_3 tenía una distancia estimada de 0, que es actualizada a -3 (que resulta de la suma de la distancia estimada de $h(V_1) = 0$ y el peso original de la arista $w(1,3) = -3$).

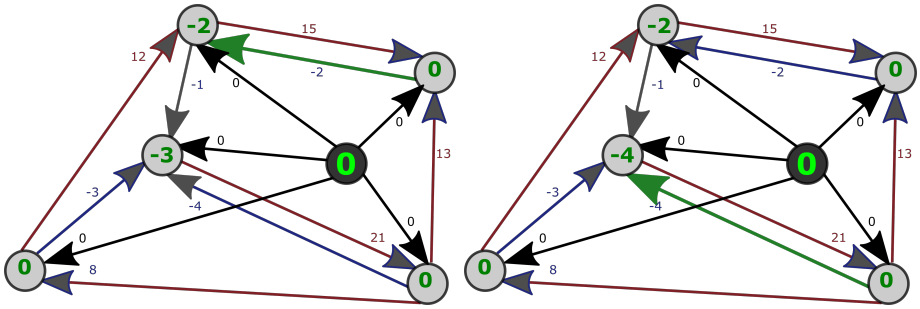
$V_4 \overset{-2}{\to} V_2$ (figura 2c) La distancia estimada al vértice V_2 era 0, y es actualizada a -2 (por la suma de la $h(V_4)$ y el peso original de la arista $w(4,2) = -2$).

$V_5 \overset{-4}{\to} V_3$ (figura 2d) Nos encontramos con otra arista que llega al nodo V_3. Su distancia estimada ahora se actualiza a -4, ya que $h(5) + w(5,4) = 0 + (-4) = -4$ es menor a la distancia estimada la momento (-3).

[3] Como se agregó una arista de peso 0 desde el origen hasta cada nodo, y las mejores distancias se inicializan en infinito excepto el origen mismo.



(a) **Estimación inicial** (b) **Estimación para vértice 3 es -3.**



(c) **Estimación para vértice 2 es -2.** (d) **Vuelve a cambiar la estimación para vértice 3.**

Figura 2: **Grafo con cambios mediante Bellman-Ford**.

Mientras que en la segunda iteración, si bien los valores de $h(x)$ son distintos, el resultado de la comparación es el mismo por lo que no hay ningún cambio en ésta ni las siguientesa, finalizando exitosamente el algoritmo de Bellman-Ford. En este punto, se crea un nuevo grafo H posteriormente empleado para calcular los caminos mediante el algoritmo de Dijkstra. Se toma como base el grafo G, actualizando los pesos de las aristas según la fórmula:

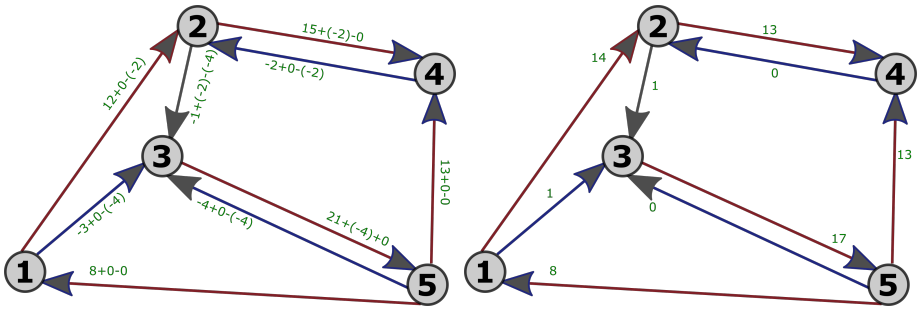
$$\hat{w}(u, v) \ := \ w(u, v) + h(u) - h(v)$$

Donde:

$w(u, v)$ es el peso original de la arista, en el grafo G, desde el vértice u hasta el vértice v.

$h(x)$ es la distancia estimada, en el grafo G', desde V_0 hasta x.

$\hat{w}$ es el nuevo peso, en el grafo H, desde el vértice u hasta el vértice v.



(a) **Cálculo de cada arista para sus pesos.** (b) **Grafo H con los valores calculados.**

Figura 3: **Grafo** H con cambios de pesos con $\hat{w}$.

4.2. Aplicar Dijkstra

Dijkstra desde vértice 1:

i Inicializar vŕtice de origen con distancia 0.

Visitando 1: con distancia estimada 0:

Arista hasta 2: de peso 14 es 14, mejor que anterior (inf), **se actualiza**.
Arista hasta 3: de peso 1 es 1, mejor que anterior (inf), **se actualiza**.

Visitando 3: con distancia estimada 1:

Arista hasta 5: de peso 17 es 18, mejor que anterior (inf), **se actualiza**.

Visitando 2: con distancia estimada 14:

Arista hasta 4: de peso 13 es 27, mejor que anterior (inf), **se actualiza**.

Visitando 5: con distancia estimada 18:

Arista hasta 4: de peso 13 es 31, **NO ES** mejor que anterior (27).

Visitando 4: con distancia estimada 27:

x No hay aristas a vértices sin visitar.

Dijkstra desde vértice 2:

i Inicializar vŕtice de origen con distancia 0.

Visitando 2: con distancia estimada 0:

Arista hasta 3: de peso 1 es 1, mejor que anterior (inf), **se actualiza**.
Arista hasta 4: de peso 13 es 13, mejor que anterior (inf), **se actualiza**.

Visitando 3: con distancia estimada 1:

Arista hasta 5: de peso 17 es 18, mejor que anterior (inf), **se actualiza**.

Visitando 4: con distancia estimada 13:

x No hay aristas a vértices sin visitar.

Visitando 5: con distancia estimada 18:

Arista hasta 1: de peso 8 es 26, mejor que anterior (inf), **se actualiza**.

Visitando 1: con distancia estimada 26:

x No hay aristas a vértices sin visitar.

Dijkstra desde vértice 3:

i Inicializar vŕtice de origen con distancia 0.

Visitando 3: con distancia estimada 0:

Arista hasta 5: de peso 17 es 17, mejor que anterior (inf), **se actualiza**.

Visitando 5: con distancia estimada 17:

Arista hasta 1: de peso 8 es 25, mejor que anterior (inf), **se actualiza**.

Arista hasta 4: de peso 13 es 30, mejor que anterior (inf), **se actualiza**.

Visitando 1: con distancia estimada 25:

Arista hasta 2: de peso 14 es 39, mejor que anterior (inf), **se actualiza**.

Visitando 4: con distancia estimada 30:

Arista hasta 2: de peso 0 es 30, mejor que anterior (39), **se actualiza**.

Visitando 2: con distancia estimada 30:

x No hay aristas a vértices sin visitar.

Dijkstra desde vértice 4:

i Inicializar vŕtice de origen con distancia 0.

Visitando 4: con distancia estimada 0:

Arista hasta 2: de peso 0 es 0, mejor que anterior (inf), **se actualiza**.

Visitando 2: con distancia estimada 0:

Arista hasta 3: de peso 1 es 1, mejor que anterior (inf), **se actualiza**.

Visitando 3: con distancia estimada 1:

Arista hasta 5: de peso 17 es 18, mejor que anterior (inf), **se actualiza**.

Visitando 5: con distancia estimada 18:

Arista hasta 1: de peso 8 es 26, mejor que anterior (inf), **se actualiza**.

Visitando 1: con distancia estimada 26:

x No hay aristas a vértices sin visitar.

Dijkstra desde vértice 5:

i Inicializar vŕtice de origen con distancia 0.

Visitando 5: con distancia estimada 0:

Arista hasta 1: de peso 8 es 8, mejor que anterior (inf), **se actualiza**.

Arista hasta 3: de peso 0 es 0, mejor que anterior (inf), **se actualiza**.

Arista hasta 4: de peso 13 es 13, mejor que anterior (inf), **se actualiza**.

Visitando 3: con distancia estimada 0:

x No hay aristas a vértices sin visitar.

Visitando 1: con distancia estimada 8:

Arista hasta 2: de peso 14 es 22, mejor que anterior (inf), **se actualiza**.

Visitando 4: con distancia estimada 13:

Arista hasta 2: de peso 0 es 13, mejor que anterior (22), **se actualiza**.

Visitando 2: con distancia estimada 13:

x No hay aristas a vértices sin visitar.

origen	$\mathbf{V_1}$	$\mathbf{V_2}$	$\mathbf{V_3}$	$\mathbf{V_4}$	$\mathbf{V_5}$
$\mathbf{V_1}$	0	14	1	27	18
$\mathbf{V_2}$	26	0	1	13	18
$\mathbf{V_3}$	25	30	0	30	17
$\mathbf{V_4}$	26	0	1	0	18
$\mathbf{V_5}$	8	13	0	13	0

(a) Distancia mínima

origen	$\mathbf{V_1}$	$\mathbf{V_2}$	$\mathbf{V_3}$	$\mathbf{V_4}$	$\mathbf{V_5}$
$\mathbf{V_1}$	-	V_1	V_1	V_2	V_3
$\mathbf{V_2}$	V_5	-	V_2	V_2	V_3
$\mathbf{V_3}$	V_5	V_4	-	V_5	V_3
$\mathbf{V_4}$	V_5	V_4	V_2	-	V_3
$\mathbf{V_5}$	V_5	V_4	V_5	V_5	-

(b) Nodo antecesor a cada destino

Cuadro 4: Resultados al aplicar Dijkstra

4.3. Restaurar valor de distancias

Finalmente, es importante tener en cuenta que el resultado de Dijkstra nos muestra los caminos mínimos, pero con las distancias alteradas $\hat{\delta}(u, v)$ (basadas en los pesos $\hat{w}$). Entonces un último paso, para cada iteración de Dijkstra (por cada vértice de origen), es restaurar la verdadera distancia hasta cada vértice destino v:

$$d(u, v) \ := \ \hat{w}(u, v) + h(v) - h(u)$$

Tomando entonces $h(x)$ que calculamos de Bellman-Ford, actualizamos (cuadro 5) cada distancia mínima de cuadro 4a para obtener la tabla cuadro 6 de distancias mínimas reales.

origen	$\mathbf{h(V_1)= 0}$	$\mathbf{h(V_2)=-2}$	$\mathbf{h(V_3)=-4}$	$\mathbf{h(V_4)= 0}$	$\mathbf{h(V_5)= 0}$
$\mathbf{h(V_1)= 0}$	0 +0-0	14+(-2)-0	1+(-4)-0	27 $\cancel{+0-0}$	18$\cancel{+0-0}$
$\mathbf{h(V_2)=-2}$	$26+0 \not/(\not/ 2)$	0 $\cancel{+(-2)-(-2)}$	$1+(-4) \not/(\not/ 2)$	$13+0 \not/(\not/ 2)$	$18+0 \not/(\not/ 2)$
$\mathbf{h(V_3)=-4}$	$25+0 \not/(\not/ 4)$	$30+(-2) \not/(\not/ 4)$	0 $\cancel{+(-4)-(-4)}$	$30+0 \not/(\not/ 4)$	$17+0 \not/(\not/ 4)$
$\mathbf{h(V_4)= 0}$	26 $\cancel{+0-0}$	0 +(-2)-0	1+(-4)-0	0 $\cancel{+0-0}$	18 $\cancel{+0-0}$
$\mathbf{h(V_5)= 0}$	8 $\cancel{+0-0}$	13+(-2)-0	0+(-4)-0	13 $\cancel{+0-0}$	0 $\cancel{+0-0}$

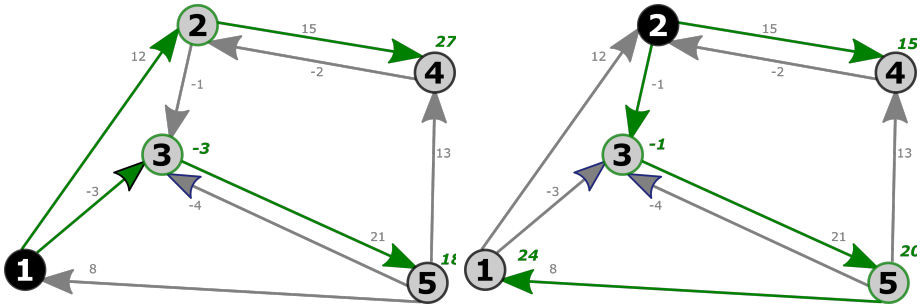
Cuadro 5: Cálculos de ajuste de distancias reales

origen	$\mathbf{V_1}$	$\mathbf{V_2}$	$\mathbf{V_3}$	$\mathbf{V_4}$	$\mathbf{V_5}$
$\mathbf{V_1}$	0	12	-3	27	18
$\mathbf{V_2}$	28	0	-1	15	20
$\mathbf{V_3}$	29	32	0	34	21
$\mathbf{V_4}$	26	-2	-3	0	18
$\mathbf{V_5}$	8	11	-4	13	0

Cuadro 6: Valores reales de distancias mínimas

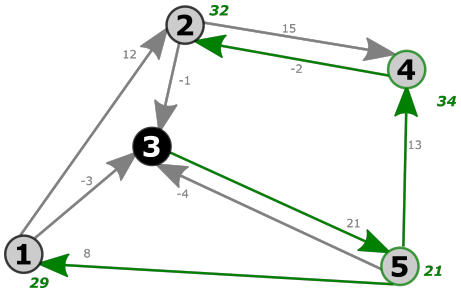
4.4. Reconstrucción caminos mínimos

Para cualquier `(origen, destino)`, a partir del cuadro 4b podemos (recursivamenta desde `destino` al `origen`) reconstruir el camino mínimo, sabiendo que su peso es el calculado en el cuadro 6. A continuación en la figura figura 4, se muestran a modo de comprobación los caminos mínimos según el vértice de `origen`:

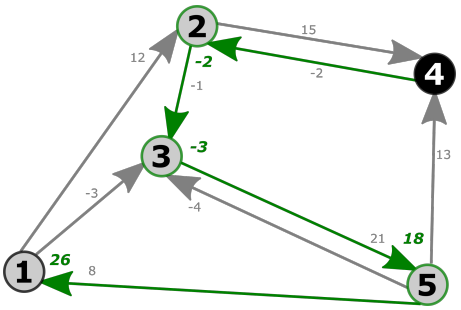


(a) **Caminos mínmos desde V_1.**

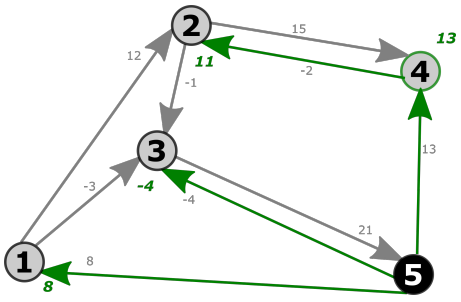
(b) **Caminos mínmos desde V_2.**



(c) **Caminos mínmos desde V_3.**



(d) **Caminos mínmos desde V_4.**



(e) **Caminos mínmos desde V_5.**

Figura 4: **Grafo** H con caminos mínimos (en verde, junto con la distancia) según el punto de origen.

5. Metodología Greedy

Enunciado P1.5

¿Puede decirse que Johnson utiliza en su funcionamiento una metodología greedy? Justifique.

Dijkstra es Greedy, ya que elige el vértice más 'ligero' (camino de menor peso desde el vértice origen) para iterativamente añadir vértices al set de vértices que utiliza para obtener el camino minimal (Cormen 659).

Por lo tanto, Johnson utiliza en su funcionamiento una metodología Greedy, debido a que utiliza Dijkstra, que es una metodología Greedy.

6. Programación dinámica

Enunciado P1.6

¿Puede decirse que Johnson utiliza en su funcionamiento una metodología de programación dinámica? Justifique.

Sí, el algoritmo de Johnson emplea la metodología de programación dinámica, ya que utiliza el algoritmo de Bellman-Ford, que es un algoritmo de programación dinámica, para computar una transformación al grafo inicial para eliminar los pesos negativos. Se almacenan resultados parciales de menor jerarquía (distancia mínima estimada) que serán empleados como parte del cálculo en iteraciones posteriores.

7. Programa con solución

Enunciado P1.7

Programar la solución usando el algoritmo de Johnson.

7.1. Criterio de selección

En el enunciado nos pide que el programa sirva para, con respecto a una nueva fábrica, «determinar dónde la ubicarán para minimizar los gastos de logística y distribución». Pero también se indica que: «Cada ruta une dos depósitos en un sentido. No todos los depósitos tienen rutas que los conecten».

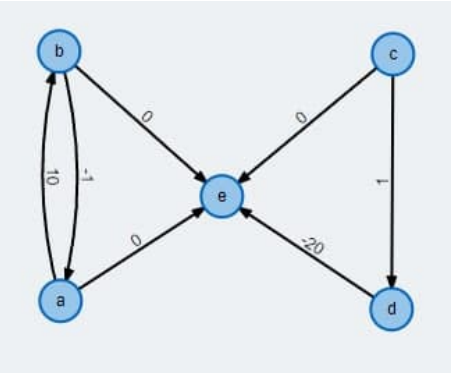
Así mismo se indica que se debe emplear el algoritmo de Johnson; éste pruduce una tabla de distancias. Sin embargo, para emplearlo a tal fin es necesario tomar una serie de decisiones relevantes para el criterio de selección, a saber:

1. Si todos los depósitos cuentan con la misma improtancia o si existe alguna ponderación.
2. Con las condiciones dadas es posible encontrarse con la disyuntiva de acceder a pocos depósitos pero con mejor costo (incluso negativo) o elegir una opción con un costo más alto pero con acceso a más depósitos (aunque aun así no a todas).

Para el primer punto, hemos asumido que todos cuentan con la misma importancia; de lo contrario suponemos que se nos hubiera indicado que también se provee una tabla de ponderaciones.

En cambio para el segundo entendemos que siendo una decisión con un posible impacto considerable y que dependa de circunstancias más allá del modelo actual (o, incluso, muy difíciles de modelar), le brindamos la información al usuario para que tome la decisión con el criterio que considere preferibel.

Mientras que si hay al menos un vértice que llegue al máximo de nodos con el mínimo costo, se tomará como la mejor recomendación (a pesar de que puedan existir otros nodos con el mismo costo pero menos alcance o el mismo alcance con mayor costo).



(a) **Grafo de ejemplo que nos llevaría a la disyuntiva**

```
PS D:\E\repos\7529tp2> python -m src.tp2 .\tests\entradas\test_ciclo_negativo.txt
Se detectó un bucle negativo
PS D:\E\repos\7529tp2> python -m src.tp2 .\tests\entradas\test_criterio_dificil.txt
Ubicaciones recomendadas:
 * Por costo promedio:  D
 * Por nº de ciudades:  C

      A     B     C     D     E
A:    0     2     inf   inf   0
B:    -1    0     inf   inf   -1
C:    inf   inf   0     1     -19
D:    inf   inf   inf   0     -20
E:    inf   inf   inf   inf   0
PS D:\E\repos\7529tp2> python -m src.tp2 .\tests\entradas\test_cormen.txt
Ubicación recomendada:  4

      1     2     3     4     5
1:    0     1     -3    2     -4
2:    3     0     -4    1     -1
3:    7     4     0     5     3
4:    2     -1    -5    0     -2
5:    8     5     1     6     0
PS D:\E\repos\7529tp2>
```

(b) **Salida del programa, con y sin disyuntiva.**

Por ejemplo, para el caso del figura 5a si la fábrica se ubicara en D tendría un costo promedio de -20 (es decir, genera ganancia), pero acceso a un sólo depósito adicional; y si se ubica en C tiene un costo de -18, pero no puede alcanzar a C ni D.

Entonces, será un usuario más conocdedor del problema quien podrá decidir si alguna de las dos es una situación aceptable y cuál es preferible (¿cuál es el costo de no acceder a ciertos depósitos?), o si sería preferible posponer la construcción de la fábrica y, en ese caso, qué acciones se deben tomar; tales como crear una ruta, etc.

7.2. Uso

Para ejecutar el programa es necesario descargar e instalar Python 3.10 y, desde la línea de comandos ubicado en la carpeta donde se descomprimió, [4] ejecutar:

```
python -m src.tp2 NOMBRE_DE_ARCHIVO
```

Siendo `NOMBRE_DE_ARCHIVO` la ruta (relativa a la carpeta de trabajo) al archivo con las rutas, según el formato indicado en Formato de archivos. Por ejemplo, para usarse el ejemplo dado en Cormen:

```
python -m src.tp2 tests/entradas/test_cormen.txt
```

7.3. Explicación general de la implementación

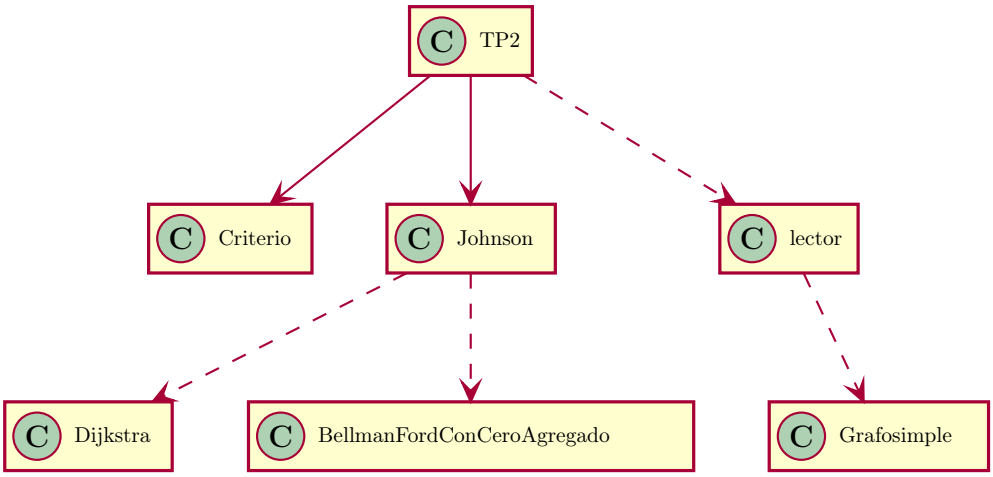


Figura 6: Diagrama de clases general.

Nos enfocaremos en los fragmentos del código más relevantes algorítmicamente.

src.Dijkstra

```
1       class Dijkstra:
2       def __init__(self,grafo,desde):
3           cantNodos = grafo.cantidadNodos()
4           if (desde < 0) or (desde >= cantNodos):
5               raise Exception("El nodo no existe")
6
7           self._estadoVisitado = [False    for i in range(cantNodos)]
8           self._visitables     = [i        for i in range(cantNodos)]
9           self.distancias      = [math.inf for i in range(cantNodos)]
10          self.distancias[desde] = 0
11
12          while(len(self._visitables)):
13              visitando = self._visitar()
14              arcos = grafo.arcoDesdeNodoId(visitando)
15              for(destino, peso) in arcos:
16                  if not self._fueVisitado(destino):
17                      distanciaAlternativa = self.distancias[visitando] + peso
```

[4]

```
18                      if distanciaAlternativa < self.distancias[destino]:
19                          self.distancias[destino] = distanciaAlternativa
20
21      def _visitar(self):
22          # Ordenar visitables de menor a mayor distancia
23          self._visitables = sorted(self._visitables, key=lambda
            ↪  nodo:self.distancias[nodo])
24
25          aVisitar = self._visitables.pop(0)
26          self._estadoVisitado[aVisitar] = True
27          return aVisitar
28
29      def _fueVisitado(self, nodo):
30          return self._estadoVisitado[nodo]
```

src.BellmanFordConCeroAgregado Para el caso de Bellman-Ford, nos decidimos por una implementación sutilmente modificada, considerando que:

1. Sólo se emplea dentro de Johnson.
2. Se agrega un vértice de grado de entrada cero y con aristas de peso 0; pero se descarta al finalizar.
3. Dentro del algoritmo no se modifica el grafo más allá del punto anterior.

En particular, se optó por evitar crear una copia del grafo o tener que modificarlo. Algunas observaciones al respecto:

13 Se agregaría una arista por nodo preexistente. En vez de eso, simplemente se calcula la cantidad.

20 Se inicializa en 0 porque siempre será el resultado al iterar por primera vez por las aristas agregadas.

22 Tal como se establece en el algoritmo se itera externamente una vez por cada vértice del grafo modificado.

25 Se implementa una parada temprana: en caso de que no haya modificaciones, cada iteración terminará con el mismo resultado.

28 Por claridad la detección de bucles negativos se especificó externamente. Si bien, como hay detección de cambios en la iteración interna podría hacerse una iteración más y si sale del bucle con cambios, encontró un bucle negativo.

33 Aquí está la parte de la iteración interna por cada arista.

```
1       class BellmanFordConCeroAgregado:
2       def __init__(self,grafo):
3           """Crea una instancia con las distancias para un grafo copia del
4           dado, al que se añade un nodo con arcos de peso 0 hacia el resto
5           de los nodos (pero sin nodo que llegue al mismo). Las distancias
6           serán calculandas con Bellman-Ford¹ para el nodo agregado, pero
7           sólo se devolverá las de los demás nodos.
8           ¹ Sabiendo que el nodo agregado no tiene nodos de entrada por lo
9           que no puede cambiar su distancia, y entonces desde ella hasta
10          otros siempre será 0, en esta implementación no se almacena."""
11          cantNodos = grafo.cantidadNodos()
```

```
12          cantArcosCon0 = grafo.cantidadArcos() + grafo.cantidadNodos()
13          self._arcos = list(grafo.arcos())
14
15          if cantNodos<1:
16              raise Exception("")
17          # En la primera iteración real de Beellman-Ford, como el nodo
18          # agregado tiene arcos de peso 0, todos pasan a tener esa distancia.
19          self.distancias = [0 for i in range(cantNodos)]
20
21          for k in range(cantArcosCon0 - 1):
22              self._iterar()
23              # Finalización temprana:
24              if(not self._huboCambio):
25                  break
26
27          # Comprobar bucle
28          self._iterar()
29          if(self._huboCambio):
30              self.distancias = False
31
32      def _iterar(self):
33          self._huboCambio = False
34          for (origen,destino,peso) in self._arcos:
35              distanciaAlternativa = self.distancias[origen] + peso
36              if distanciaAlternativa < self.distancias[destino]:
37                  self.distancias[destino] = distanciaAlternativa
38                  self._huboCambio = True
```

src.Johnson

9 Aquí verificamos que haya algún peso negativo, para aplicar la transformación si es necesaria:

10 Se obtienen las distancias al nodo temporal con Bellman-Ford.

11 Se modifica el grafo con la fórmula $\hat{w} = w + h(u) - h(v)$.

13 En caso de que no se haya aplicado ninguna modificación, se deja h en cero.

19 Aquí se aplica la corrección si se modificó (en caso contrario, es 0).

```
1       from src.bellmanford import BellmanFordConCeroAgregado
2       from src.dijkstra import Dijkstra
3
4       class Johnson:
5           def __init__(self, grafo):
6               if grafo.cantidadNodos()<1:
7                   raise Exception("No hay suficientes nodos")
8
9               if any(peso<0 for (u,v,peso) in grafo.arcos()):
10                  h = BellmanFordConCeroAgregado(grafo).distancias
11                  grafo.modificarPesos( lambda w,u,v: w +h[u] -h[v] )
12              else:
13                  h = [0 for i in range(grafo.cantidadNodos())]
14              self.h = h
15
16              self.matriz = []
```

```
17              for u in range(grafo.cantidadNodos()):
18                  d = Dijkstra(grafo, u).distancias
19                  self.matriz.append([ d[v] + h[v] -h[u] for v in
                    ↪  range(grafo.cantidadNodos()) ])
```

P2. Parte 2: Un poco de teoría

a. 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.

1. Formas de resolución de problemas

Enunciado P2.1

Hasta el momento hemos visto 3 formas distintas de resolver problemas. Greedy, división y conquista y programación dinámica.

1. Describa brevemente en qué consiste cada una de ellas.
2. Identifique similitudes, diferencias, ventajas y desventajas entre las mismas. ¿Podría elegir una técnica sobre las otras?

1.1. Breve descripción

Greedy Metodología de resolución de problemas de optimización (minimización o maximización) que consiste en dividir el problema en subproblemas con una jerarquía entre ellos. Cada subproblema se va resolviendo iterativamente mediante una elección heurística y habilita nuevos subproblemas.

Para cada problema existen diferentes algoritmos greedy, pero algunos no resultan en la solución óptima y no todos los problemas pueden resolverse mediante un algoritmo greedy.

Para poder aplicar esta metodología, un problema debe contener las siguientes propiedades:

- Elección greedy: Se selecciona una solución óptima local esperando que la misma nos acerque a la solución óptima global. Para solucionar un subproblema, analiza el conjunto de los elementos del problema en el estado en que llegaron al mismo y elige heurísticamente la "mejor solución" local. Un subproblema está condicionado por las elecciones de los anteriores problemas y condiciona a los subproblemas siguientes.
- Subestructura óptima: Un problema contiene una subestructura óptima si la solución óptima global del mismo contiene en su interior las soluciones óptimas de sus subproblemas. La elección greedy iterativamente resolverá los subproblemas óptimamente y nos llevará a la solución óptima global.

División y conquista Consiste en dividir el problema en subproblemas de igual naturaleza y menor tamaño. Se los conquista (resuelve) en forma recursiva (hasta un caso base) y se combina los resultados en una solución general. Generalmente se puede aplicar a problemas donde la solución por fuerza bruta ya tiene una complejidad polinómica. Analizar su complejidad requiere resolver una relación de recurrencia.

Programación dinámica Metodología de resolución de problemas de optimización (minimización o maximización) que consiste en dividir el problema en subproblemas con una jerarquía entre ellos (de menor a mayor tamaño). Cada subproblema puede ser utilizado ser reutilizado en diferentes subproblemas mayores.

Para poder resolverse de forma óptima utilizando programación dinámica, un problema debe cumplir las siguientes propiedades:

- Subestructura óptima: Un problema contiene una subestructura óptima si la solución óptima global del mismo contiene en su interior las soluciones óptimas de sus subproblemas.
- Subproblemas superpuestos: Un problema contiene subproblemas superpuestos si en la resolución de sus subproblemas vuelven a aparecer subproblemas previamente calculados.

La programación dinámica emplea **Memorización**: Técnica que consiste en almacenar los resultados de los subproblemas previamente calculados para evitar repetir su resolución cuando vuelva a requerirse. De esa forma reducen la cantidad total de subproblemas a calcular, consiguiendo reducir significativamente la complejidad temporal de la solución.

1.2. Ventajas y desventajas

Similitudes

- En todos ellos se divide un problema en subproblemas.
- Greedy/Programación dinámica: Subestructura óptima, los subproblemas tienen una jerarquía entre ellos.

Ventajas

Greedy

- Suelen ser rápidos.
- Fáciles de implementar.

Programación dinámica

- La técnica de memorización permite reducir la complejidad temporal.

División y conquista

- Permite la resolución de problemas complejos.
- Es fácil de medir su complejidad.

Desventajas

Greedy

- Es difícil de demostrar que para cada instancia de un problema se llega a la solución óptima mediante un algoritmo greedy.
- No todos los problemas se pueden solucionar con este método.

Programación dinámica

- Se necesita mucha memoria para almacenar los resultados de los subproblemas.
- Es posible que haya soluciones almacenadas en memoria que no se vuelvan a utilizar.

División y conquista

- Al utilizarse soluciones de forma recursiva, se utiliza mucha memoria del stack y esto puede traer problemas en casos grandes.

¿Podría elegir una técnica sobre las otras?

No podría elegir una técnica sobre la otra, siempre dependerá de las propiedades del problema que tenga que resolver y los recursos disponibles.

2. Caso teórico puntual

Enunciado P2.2

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?

Depende. Debe considerarse qué opción es la más eficiente, no sólo en su tiempo de ejecución, sino también con respecto a los recursos disponibles. Si, PD es más eficiente que G, el *conundrum* está en si los recursos son suficientes para hacer correr el algoritmo PD. Entonces, el algoritmo a ser ejecutado depende de la cantidad de elementos que posea la instancia a solucionar.

Si la instancia es pequeña, es decir aplicable PD en tanto a los recursos computacionales disponibles, se utilizará PD. En caso contrario, la instancia es tan grande que deberá utilizarse G, dado que no es posible ejecutar PD.

En conclusión, conviene tener programados ambos algoritmos, ya que dependiendo de la situación previamente explicada, y otros posibles factores como preferencias institucionales o de algún usuario demasiado insistente, convendrá utilizar o uno u otro algoritmo.