Caminos mínimos en grafos

Juan Gutiérrez

June 24, 2022

1 Representaciones en grafos

Dado un grafo $G = (V(G), E(G))$, podemos representarlo de dos maneras: lista de adyacencia o matriz de adyacencia.

1.1 Listas de adyacencia

Consiste en un arreglo Adj de $|V(G)|$ listas, una por cada vértice en $V(G)$. Para cada $u \in V(G)$, $Adj(u)$ contiene a los vértices v tal que $uv \in E(G)$. Si el grafo no dirigido, $\sum_v |Adj(v)| = 2|E(G)|$. Por lo tanto la memoria utilizada es $O(|V(G)| + |E(G)|)$.

1.2 Matrices de adyacencia

Es una matriz A indexada por los $V(G)$ en las filas y en las columnas tal que $A[u, v] = 1$ si $uv \in E(G)$ y $A[u, v] = 0$ en caso contrario. Por lo tanto la memoria utilizada es $O(|V(G)|^2)$.

Observación: si el grafo es denso (la cantidad de aristas es proporcional al cuadrado de la cantidad de vértices), es mejor usar matrices de adyacencia. Si el grafo es esparso (la cantidad de aristas es proporcional a la cantidad de vértices), es mejor usar listas de adyacencia.

2 Algoritmo de Dijkstra

Suponga que para cada $uv \in E(G)$, tenemos un número no negativo ℓ_{uv}.

Problema Caminos Mínimo. Dado un grafo G con longitudes ℓ y un vértice s, encontrar la distancia desde s hacia todos los vértices.

Ahora la *distancia* de u a v es la *longitud de un camino mínimo* entre u y v, tomando en cuenta las longitudes de cada arista. Podemos resolver el problema anterior haciendo una transformación y aplicando BFS:

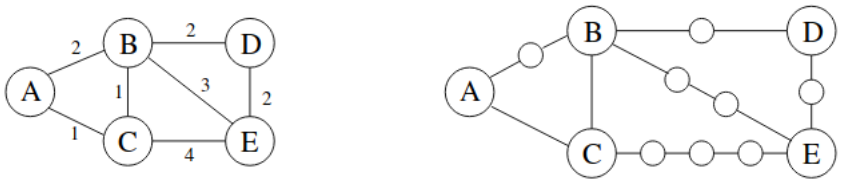

Figure 1: Tomada del libro Dasgupta et al, Algorithms

Sin embargo, el tiempo de ejecución sería $O(|V(G)| + |E(G)|\ell_{\max})$. Si ℓ_{max} es muy grande, este algoritmo no es eficiente.

Presentaremos a continuación un algoritmo eficiente conocido como Algoritmo de Dijkstra.

Idea: ir incrementando gradualmente el conjunto de vértices a los cuales ya se les ha seteado correctamente la distancia. Llamaremos a este conjunto R. Inicialmente, $R = \{s\}$ (antes de la primera iteración).

En una iteración cualquiera, escogeremos un vértice en $V(G) \setminus R$ **lo más cercano posible a** s. Al obtener un vértice de este tipo, podemos setearle correctamente su distancia y adicionarlo a R.

¿Por qué? Sea v un vértice a distancia mínima de s. Tome un camino mínimo de s a v. Sea u el vértice adyacente a v en dicho camino. Note que $u \in R$, ya que si $u \notin R$, entonces la distancia de s a u sería menor que la distancia de s a v y u sería escogido en lugar de v, contradicción a la elección de v.

Como $u \in R$, entonces su distancia ya ha sido seteada correctamente en un vector *dist*. Note que la distancia de s a v es igual a $dist(u) + \ell_{uv}$ (recuerde que cada subcamino de un camino mínimo es también mínimo).

Por lo tanto, para encontrar un vértice en $V(G) \setminus R$ con distancia mínima, basta evaluar cada $dist(u) + l_{uv}$ para todos los $u \in R$. Podemos setear estos valores en la iteración anterior e ir actualizando de una iteración para otra, solamente, posiblemente los valores de los vértices adyacentes al nuevo vértice que ha entrado en R.

Recibe: Un grafo G con longitudes ℓ no negativas en las aristas y un vértice $s \in V(G)$. Modifica *dist* de manera que $dist(v)$ guarda la distancia de s a v en G.

DIJKSTRA(G, s)

1: **for** $v \in V(G)$
2: $\quad dist(v) \leftarrow \infty$
3: $dist(s) \leftarrow 0$
4: $R \leftarrow \emptyset$
5: **while** $R \neq V(G)$
6: $\quad$ Sea $v \in V(G) \setminus R$ tal que $dist(v) = \min\{dist(w) : w \in V(G) \setminus R\}$
7: $\quad R = R \cup \{v\}$
8: $\quad$ **for** $vz \in E(G)$
9: $\quad\quad$ **if** $dist(v) + \ell_{vz} < dist(z)$

10: $dist(z) \leftarrow dist(v) + \ell_{vz}$

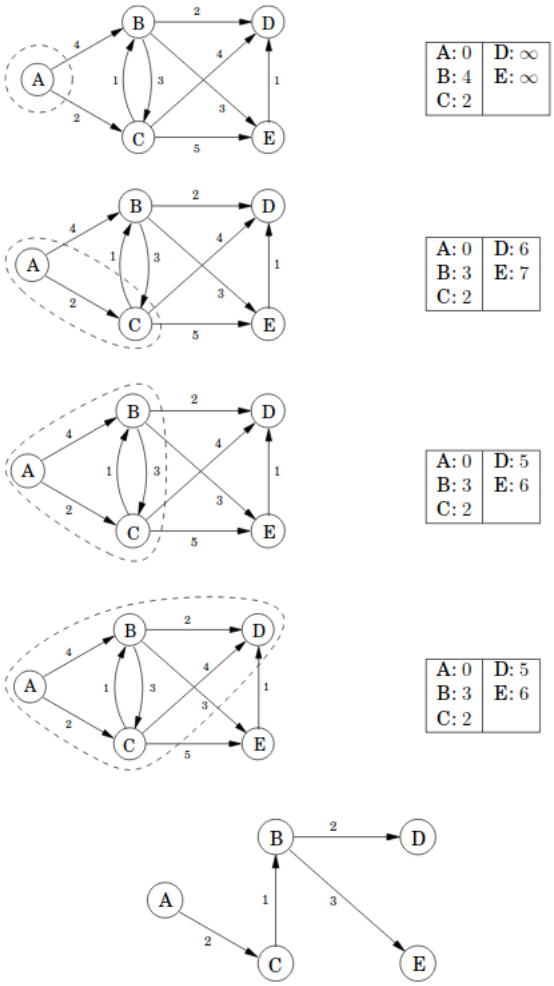

Figure 2: Tomada del libro Dasgupta et al, Algorithms

El tiempo de ejecución es cuadrático en el número de vértices, $O(|V(G)|^2)$, debido a la línea 6, que hace una búsqueda secuencia en el arreglo. Para grafos densos, este tiempo de ejecución es el mejor posible, sin embargo, podemos mejorar el tiempo de ejecución del algoritmo en grafos esparsos ($|E(G)| = O(|V(G)|)$) haciendo uso de fila de prioridades.

Recibe: Un grafo G con longitudes ℓ no negativas en las aristas y un vértice $s \in V(G)$. Modifica *dist* de manera que $dist(v)$ guarda la distancia de s a v en G.

DIJKSTRA-FP(G, s)

1: **for** $v \in V(G)$
2: $\quad dist(v) \leftarrow \infty$
3: $dist(s) \leftarrow 0$
4: $Q \leftarrow V(G)$ //fila de prioidades inicializada
5: **while** $Q \neq \emptyset$
6: $\quad v \leftarrow$ EXTRACT-MIN(Q)
7: $\quad$ **for** $vz \in E(G)$
8: $\quad\quad$ **if** $dist(v) + \ell_{vz} < dist(z)$
9: $\quad\quad\quad dist(z) \leftarrow dist(v) + \ell_{vz}$
10: $\quad\quad\quad$ DECREASE-KEY(Q,z)

Tiempo de ejecución: $O((|V(G)| + E(G)) \lg |V(G)|)$.
Con Fibonacci heap, el tiempo de ejecución mejora a $O((|V(G)| \lg |V(G)| + E(G))$

Implementation	`deletemin`	`insert/ decreasekey`	$\|V\|$ × `deletemin` + ($\|V\| + \|E\|$) × `insert`
Array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
Binary heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
d-ary heap	$O(\frac{d \log \|V\|}{\log d})$	$O(\frac{\log \|V\|}{\log d})$	$O((\|V\| \cdot d + \|E\|) \frac{\log \|V\|}{\log d})$
Fibonacci heap	$O(\log \|V\|)$	$O(1)$ **(amortized)**	$O(\|V\| \log \|V\| + \|E\|)$

Figure 3: Tomada del libro Dasgupta et al, Algorithms

3 Longitudes negativas (Algoritmo de Bellman-Ford)

Dijkstra funciona porque un camino mínimo de la raíz s hacia un vértice v contiene vértices que están más cerca que v. Esto no funciona cuando hay aristas negativas.

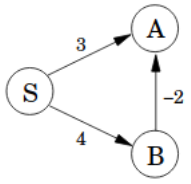

Figure 4: Tomada del libro Dasgupta et al, Algorithms

En la figura, un camino mínimo de S hacia A pasa a través del nodo B, el cual está más lejos que A.

Sin embargo, inclusive con arcos negativos, se sigue cumpliendo la siguiente propiedad: *para un vértice v y un vecino u* de v, la distancia de s a v es menor o igual que la distancia de s a u más ℓ_{uv}.

¿Por qué? Si P_{su} es un camino mínimo de s a u, entonces $P_{su} \cdot uv$ es un camino de s a v con longitud $|P_{su}|+\ell_{uv}$, es decir con longitud igual a la distancia de s a u más ℓ_{uv}. Luego, un camino mínimo de s a v tendrá tamaño menor o igual al tamaño de dicho camino.

Más aún, con un análisis similar, podemos deducir que para cada vértice v, existe un vecino u de v tal que la distancia de s a v es igual a la distancia de s a u más ℓ_{uv}.

Por lo tanto, para cada v, suponiendo que *dist* ya ha sido seteado correctamente a sus vecinos, nos interesa encontrar $\min_{uv \in E(G)} dist(u) + \ell_{uv}$

Nos podemos asegurar de esto si encontramos una manera correcta de utilizar el siguiente procedimiento:

UPDATE(u, v)

1: $dist(v) = \min\{dist(v), dist(u) + \ell_{uv}\}$

Dijkstra hace algunos updates, pero en algunos casos son insuficientes. El algoritmo de Bellman-Ford hace un update de cada arista $|V(G)| - 1$ veces, con lo cual se garantiza que las distancias han sido seteadas correctamente.

Recibe: Un grafo G con longitudes ℓ (positivas o negativas) en las aristas, **y sin ciclos negativos** y un vértice $s \in V(G)$. Modifica *dist* de manera que $dist(v)$ guarda la distancia de s a v en G.

BELLMAN-FORD(G, s)

1: **for** $v \in V(G)$
2: $\quad dist(v) \leftarrow \infty$
3: $dist(s) \leftarrow 0$
4: **for** $i = 1$ hasta $V|(G)| - 1$
5: $\quad$ **for** $uv \in E(G)$
6: $\quad\quad$ UPDATE(u, v)

La demostración de correctitud es complicada. Sin embargo, dejamos las siguientes invariantes que prueban la correctitud.

Al final de la i-ésima iteración de las líneas 4–6:

- Si $dist(w) \neq \infty$, entonces $dist(w)$ es igual a la longitud de un camino de s a w.
- Si existe un camino de s a w con máximo i aristas, entonces $dist(w)$ es como máximo la longitud de un camino mínimo de s a w con como máximo i aristas.

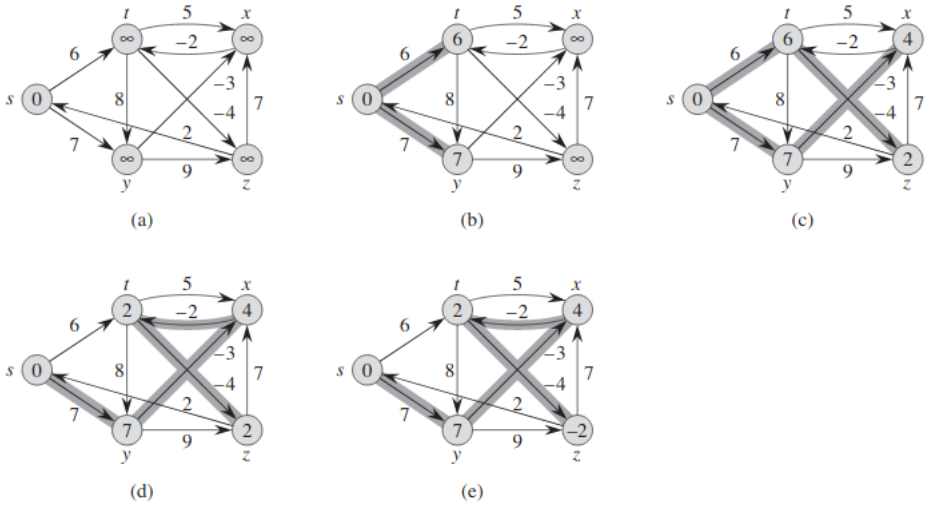

Figure 5: Tomada del libro Cormen et al, Introduction to Algorithms. El orden de procesamiento es $tx, ty, tz, xt, yx, yz, zx, zs, st, sy$.

3.1 Ciclos negativos

El algoritmo de Bellman-Ford falla en presencia de ciclos negativos.

¿Por qué? En realidad Bellman-Ford (y también Dijkstra) encuentra caminos mínimos no necesariamente simples (con posibilidad de vértices repetidos). Cuando no hay ciclos negativos estos caminos siempre son simples.

Por lo tanto, tanto Bellman-Ford (como Dijkstra), aunque están diseñados para encontrar caminos no necesariamente simples, siempre terminan encontrando caminos simples en la **no** presencia de ciclos negativos.

De aca hay dos opciones si existen ciclos negativos:

- Resolver el problema considerando caminos no necesariamente simples:

  El problema no está bien definido (ver discusión anterior). Sin embargo, una modificación a Bellman-Ford puede detectar si el grafo de entrada tiene un ciclo negativo. ¿Como lo detecta? Basta hacer una iteración más al primer bucle for y ver si hay alguna actualización más, si la hay, entonces existe ciclo negativo.

- Restringir el problema a encontrar caminos simples. Es decir, dado un grafo G con ciclos negativos y un vértice s, encontrar un camino simple de s a los demás vértices.

  Este problema es NP-dificil: podemos reducir el problema de camino máximos simples (longest path) en grafos con pesos no negativos a este problema.

  Y se sabe que longest path es NP-dificil.

4 Caminos mínimos entre todos los pares

Consideremos el siguiente problema.

- Entrada: Grafo dirigido G con pesos en los arcos y **sin ciclos negativos**.
- Salida: Matriz M tal que $M[u, v]$ guarda el peso de un camino mínimo de u a v en G.

Podemos resolver el problema anterior con un algoritmo ingenuo: para cada vértice, encontrar caminos mínimos hacia los demás usando un algoritmo conocido. En ese caso,

- si los pesos son no negativos, podemos usar Dijkstra y tendríamos un algoritmo $O(n(n+m)\lg n) = O(nm \lg n)$.
- si los pesos pueden ser negativos, usamos Bellman-Fordy y tendríamos un algoritmo $O(n^2 m)$.

A continuación veremos algoritmos específicos para el problema. Antes de ello, tenemos las siguientes convenciones.

- El conjunto de vértices del grafo en cuestión, $V(G)$, es $\{1, 2, \ldots, n\}$.
- Los pesos de las aristas son representados por una matriz $W = (w_{ij})$, donde

$$w_{ij} = \begin{cases} 0 & \text{si } i = j \\ \text{el peso de la arista } ij & \text{si } i \neq j \text{ y } ij \in E(G) \\ \infty & \text{si } i \neq j \text{ y } ij \notin E(G) \end{cases}$$

4.1 Primer algoritmo

Se basa en la siguiente propiedad.

Lema 4.1. *Todo subcamino de un camino mínimo es mínimo.*

Proof. Sea P un camino mínimo de s a t. Sea Q un subcamino de P, digamos de u a v. Suponga por contradicción que Q no es mínimo. Entonces existe un camino Q' de u a v con peso menor o igual a Q. Si reemplazamos Q por Q' en P, obtenemos un camino con peso menor o igual a P entre s y t, una contradicción. □

Luego, por el lema anterior, si $P = i \cdots kj$ es un camino mínimo entre i y j con como máximo ℓ aristas, entonces $P' = i \cdots k$ es un camino mínimo entre i y k con como mmáximo $\ell - 1$ aristas. Esto nos da un algoritmo de programación dinámica.

Para cada par (i, j), sea $\text{OPT}(i, j, \ell)$ el peso de un camino mínimo desde i hacia j que usa como máximo ℓ aristas. Tenemos la sgte recurrencia:

$$\text{OPT}(i,j,\ell) = \begin{cases} 0 & : \text{si } \ell = 0 \text{ e } i = j \\ \infty & : \text{si } \ell = 0 \text{ e } i \neq j \\ \min_{1 \leq k \leq n} \{\text{OPT}(i,k,\ell-1) + w_{kj}\} & : \text{si } \ell > 0 \end{cases}$$

La respuesta estará en $\text{OPT}((i, j, n-1))$, ya que siempre existe un camino mínimo con como máximo $n - 1$ aristas.

A continuación, veremos el pseudocódigo del algoritmo.

Recibe: Una matriz W asociada a un grafo G con pesos en las aristas, con n vertices, **y sin ciclos negativos**.

Devuelve: Una matriz que guarda las distancias entre todos los pares de vértices de G.

ALGO-1(W)
1: $M^{(1)} \leftarrow W$
2: **for** $\ell = 2$ hasta $n - 1$
3: $\quad M^{(\ell)} \leftarrow$ CALCULAR $(M^{(\ell-1)}, W)$
4: **return** $M^{(n-1)}$

La línea 3 del algoritmo calcula $M^{(\ell)}$ a partir de $M^{(\ell-1)}$ a partir de la recurrencia. Este procedimiento replica la idea de multiplicación de matrices.

CALCULAR($M^{(\ell-1)}, W$)
1: **for** $i = 1$ hasta n
2: $\quad$ **for** $j = 1$ hasta n
3: $\quad\quad M_{ij}^{(\ell)} \leftarrow \infty$
4: $\quad\quad$ **for** $k = 1$ hasta n
5: $\quad\quad\quad M_{ij}^{(\ell)} \leftarrow \min\{M_{ij}^{(\ell-1)}, M_{ik}^{(\ell-1)} + W_{kj}\}$
6: **return** $M^{(\ell)}$

El tiempo de ejecución del algoritmo ALGO-1 es $O(n^4)$. A continuación veremos como mejorar este tiempo.

4.2 Segundo algoritmo

El segundo algoritmo es escencialmente una mejora en tiempo de ejecución del primero. Note que solo nos interesa el resultado final. Además tenemos que para

matrices se cumple la propiedad asociativa para el procedimiento CALCULAR. Luego tenemos el siguiente algoritmo recursivo.

Recibe: Una matriz W asociada a un grafo G con pesos en las aristas, con n vertices, **y sin ciclos negativos**.

Devuelve: Una matriz que guarda las distancias entre todos los pares de vértices de G.

ALGO-2(W)

1: $M^{(1)} \leftarrow W$
2: $\ell \leftarrow 1$
3: **while** $\ell < n - 1$
4: $\quad M^{(2\ell)} \leftarrow$ CALCULAR $(M^{(\ell)}, M^{(\ell)})$
5: $\quad \ell \leftarrow 2\ell$
6: **return** M^{ℓ}

El tiempo de ejecución del algoritmo es $O(n^3 \lg n)$.

4.3 Tercer algoritmo - Floyd Warshall

Utiliza programación dinámica con un enfoque distinto. Recuerde que $V(G) = \{1, 2, ldots, n\}$. Definimos OPT$(i, j, k)$ como el costo de un camino mínimo de i hacia j cuyos vértices internos están en el conjunto $\{1, 2, \ldots, k\}$.

Si P es un camino de este tipo, entonces tenemos dos opciones dependiendo del vértice k.

- k es un vértice interno en P

  En ese caso, existen dos caminos P_1 y P_2 tales que P_1 es un camino de i hacia k y P_2 es un camino de k hacia j. Note que todos los vértices internos de P_1 y P_2 están en $\{1, 2, \ldots, k-1\}$.

- k no es un vértice interno en P.

  En ese caso, P es un camino mínimo entre i y j cuyos vértices internos están en $\{1, \ldots, k-1\}$.

Luego, tenemos la siguiente recurrencia.

$$\text{OPT}(i,j,k) = \begin{cases} w_{ij} & : \text{si } k = 0 \\ \min\{OPT(i,j,k-1), OPT(i,k,k-1) + OPT(k,j,k-1)\} & : \text{si } k \geq 1 \end{cases}$$

El pseudocódigo correspondiente es bastante directo, y nos da un algoritmo $O(n^3)$.

Recibe: Una matriz W asociada a un grafo G con pesos en las aristas, con n vertices, **y sin ciclos negativos**.

Devuelve: Una matriz que guarda las distancias entre todos los pares de vértices de G.

FLOYD-WARSHALL(W)

1: $M^{(0)} \leftarrow W$

2: **for** $k = 1$ hasta n
3: **for** $i = 1$ hasta n
4: **for** $j = 1$ hasta n
5: $M_{ij}^{(k)} \leftarrow \min\{M_{ij}^{(k-1)}, M_{ik}^{(k-1)} + M_{kj}^{(k-1)}\}$
6: **return** $M^{(n)}$

4.4 Cuarto algoritmo: Johnson

El algoritmo de Johnson resuelve el problema en tiempo de ejecución $O(n^2 \lg n + nm)$. Si el grafo es esparso, este tiempo de ejecución es $O(n^2 \lg n)$, por lo tanto es asintóticamente mejor que todos los algoritmos anteriores.

La idea del algoritmo consiste en cambiar los pesos del grafo de manera tal que los nuevos pesos sean no negativos, a la vez que los caminos mínimos se siguen manteniendo. Luego de eso, el algoritmo corre n veces dijkstra para cada vértice.

En este caso no es necesario construir una matriz de pesos como en los algoritmos anteriores para el input. Basta recibir el grafo como en los algoritmos de Dijkstra y Bellman-Ford.

Recibe: Un grafo G con pesos w en las aristas, con n vertices, **y sin ciclos negativos**.

Devuelve: Una matriz que guarda las distancias entre todos los pares de vértices de G.

JOHNSON(G, w)
1: $h \leftarrow$ Calcular-h(G, w)
2: **for** cada arista uv
3: $\hat{w}_{uv} = w_{uv} + h(u) - h(v)$
4: **for** cada vértice u
5: $dist \leftarrow$ Dijkstra$(G, \hat{w}, u)$
6: **for** cada vértice v
7: $M_{uv} \leftarrow dist(v) + h(v) - h(u)$
8: **return** M

La línea 5 del algoritmo invoca a la función $Dijkstra(G, \hat{w}, u)$. Esta función devuelve un arreglo $dist$, indexado por lo vértices del grafo, tal que $dist[v]$ guarda la distancia desde u hacia v usando los pesos $\hat{w}$.

La línea 1 devuelve un arreglo h, indexado por los vértices del grafo. En el for de las lineas dos y 3 crearemos los pesos auxiliares, que más adelante demostraremos siempre serán no negativos, y además se preservan los caminos mínimos. El corazón del algoritmo recae entonces en como calcular este arreglo h.

Recibe: Un grafo G con pesos w en las aristas, con n vertices, **y sin ciclos negativos**.

Devuelve: Un arreglo h, indexado por $V(G)$, tal que $h(v) \leq h(u) + w_{uv}$ para cada arista uv.

CALCULAR-H(G, w)

1: Sea G' el grafo auxiliar resultante de adicionar un vértice s a G y dar pesos 0 a todas las aristas que salen de s. Sea w' la función de pesos resultante.
2: $h \leftarrow$ BELLMAN-FORD(G', w', s)
3: **return** h

La subrutina BELLMAN-FORD(G', w', s) recibe un grafo G' con pesos w' en las aristas, y un vértice s, y devuelve un arreglo h, indexado por $V(G') \setminus \{s\}$, que guarda las distancias desde s hacia todos los otros vértices.

Notemos que la importancia de CALCULAR-H reside en su encabezado, el cual debe garantizar que el arreglo h devuelto cumple que $h(v) \leq h(u) + w_{uv}$ para cada arista uv. Esta propiedad es denominada *desigualdad triangular*. No es muy dificil de demostrar que dicha propiedad se cumple razonando por contradicción.

Finalmente, garantizando que CALCULAR-H cumple esto, podemos volver al análisis del algoritmo JOHNSON. Note que ya tenemos seguridad que los pesos $\hat{w}$ (definidos en la línea 3), serán no negativos. Finalmente, debemos demostrar que dichos pesos *preservan caminos mínimos*. Para ello basta mostrar que, al fijar un par de vértices, todos los caminos entre dichos vértices incrementan su peso en la misma cantidad al momento de hacer el cambio de w a $\hat{w}$ Es decir, basta probar el siguiente lema.

Lema 4.2. *Si P es un camino mínimo de u a v en el grafo G con pesos w, entonces $\hat{w}(P) = w(P) + h(u) - h(v)$.*

Proof. Sea $P = x_0 x_1 \dots x_k$ (note que $u = x_0$ y $v = x_k$). Note que

$$\begin{aligned}
\hat{w}(P) &= \sum_{i=1}^{k} \hat{w}(x_{i-1}x_i) \\
&= \sum_{i=1}^{k} (w(x_{i-1}x_i) + h(x_{i-1}) - h(x_i)) \\
&= \sum_{i=1}^{k} w(x_{i-1}x_i) + h(x_0) - h(x_k)) \\
&= w(P) + h(x_0) - h(x_k).
\end{aligned}$$

□

Finalmente discutiremos el tiempo de ejecuciónd el algoritmo. Tenemos una llamada a Bellman-Ford, lo cual nos da un tiempo $O(nm)$ y n llamadas a Dijkstra. Si utilizamos filas de prioridades el tiempo involucrado es $O(nm \lg n + nm)$, es decir $O(nm \lg n)$. La mejor implementación para Dijkstra conocida es

con filas de prioridades de Fibonacci. En ese caso obtenemos un tiempo total de $O(n^2 \lg n + nm)$. En ambos casos, si el grafo es esparso, se mejora el tiempo de ejecución de Floyd-Warhsall.