Analisis y diseño de algoritmos. Algoritmos en grafos

Juan Gutiérrez

December 7, 2020

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 Búsqueda en profundidad (en grafos no dirigidos)

Pregunta: ¿Qúe partes de un grafo son accesibles a partir de un vértice?

¿Como resolver este problema conociendo solo los vecinos de cada vértice?

Analogía con laberintos

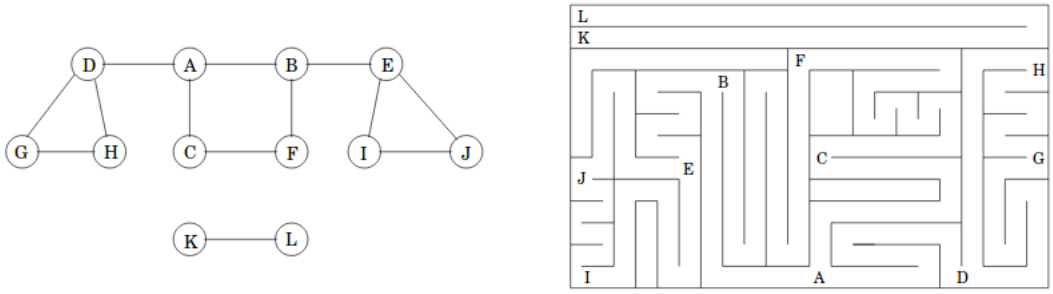

Figure 1: Tomada del libro Dasgupta et al, Algorithms

Recibe: Un grafo G, un vértice $v \in V(G)$. Llena un arreglo *visitado* tal que visitado(w)=TRUE si y solo si el vértice w es atingible desde v.

EXPLORAR(G, v)
1: *visitado*(v)= TRUE
2: **for** $uv \in E(G)$
3: **if** NOT *visitado*(u)
4: EXPLORAR(G, u)

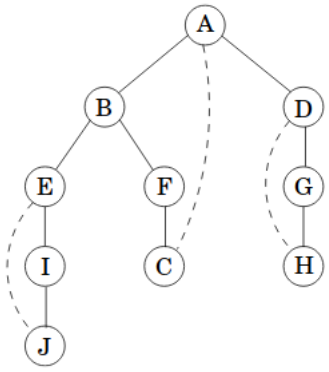

Figure 2: Tomada del libro Dasgupta et al, Algorithms

Propiedad 2.1. *El algoritmo* EXPLORAR *hace lo pedido.*

Proof. Debemos demostrar que todo vértice w alcanzable por v si y solo si tiene *visitado*(w) =TRUE al final del algoritmo.

(Solo si) Como w es alcanzable por v, existe un camino de v a w. Probaremos por inducción en k, que si existe un camino de tamaño k de v a w, entonces visitado(w)=TRUE. Si $k = 0$, entonces $w = v$ y por lo tanto *visitado*(v) se seteó en TRUE en la primera llamada.

Suponga ahora que $k > 0$. Sea w' el vértice anterior a w en el camino de v a w. Por hipótesis de inducción, $visitado(w')$ se seteó en TRUE dentro del algoritmo.

Luego, existió una llamada a EXPLORAR(G, w'). Si w ya estaba seteado en TRUE entonces no tenemos más que probar. Si no, en dicha llamada, como w es vecino de w', se ejecuta la línea 4 y la llamada recursiva $Explorar(G, w)$, que asigna TRUE a $visitado(w)$.

(Si) Como w ha sido visitado, entonces está en el árbol de búsqueda generado por el procedimiento Explorar. Luego, existe un camino de la raíz v hacia w. □

2.1 Recorriendo todo el grafo

El algoritmo EXPLORAR solo visita la porción de G alcanzable desde el vértice v. Para examinar el resto del grafo, debemos volver a invocar el procedimiento desde un vértice que aún no ha sido visitado.

El algoritmo DFS (Depth First Search) hace esto repetidamente hasta visitar todo el grafo.

Recibe: Un grafo G. Visita todos los vértices del grafo usando la técnica de búsqueda en profundidad

DFS(G)
1: **for** $v \in V(G)$
2: $visitado(v)$= FALSE
3: **for** $v \in V(G)$
4: **if** NOT $visitado(v)$
5: EXPLORAR(G, v)

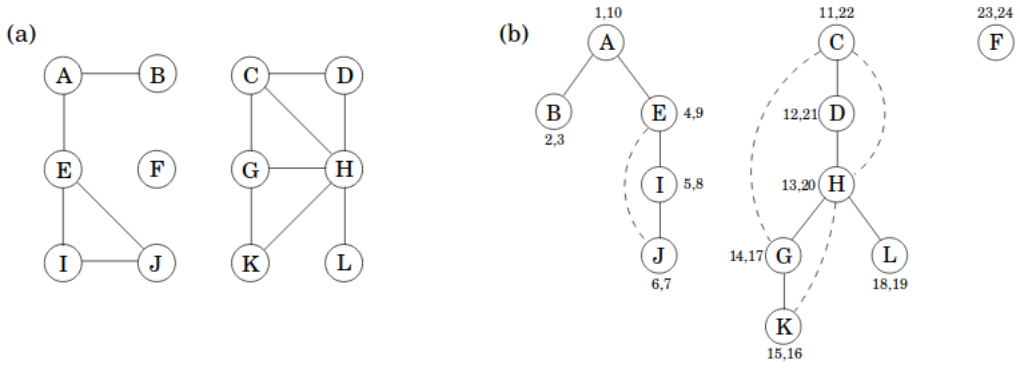

Figure 3: Tomada del libro Dasgupta et al, Algorithms

2.2 Análisis del tiempo de ejecución.

Al invocar a la función EXPLORAR a partir de un vértice v, el tiempo de ejecución, sin contar las llamadas recusivas que existan dentro, es $O(1)$ (línea 1) + $O(|d(v)|)$ (líneas 2 y 3, suponiendo que es implementado con listas de adyacencia), donde $d(v)$ es la cantidad de vecinos de v.

Como para cada vértice v solo invocamos a EXPLORAR(V) exactamente una vez, el tiempo de ejecución total sería:

$$\begin{aligned}\sum_{v \in V(G)} O(1) + \sum_{v \in V(G)} O(d(v)) &= O(|V(G)|) + O(2|E(G)|) \\ &= O(|V(G)| + |E(G)|).\end{aligned}$$

2.3 Aplicación: conectividad en grafos no dirigidos

Un grafo es *conexo* si existe un camino entre cualquier par de vértices. En la figura 3, el grafo no es conexo, porque no existe camino de A hacia K. Un *componente* de un grafo es un subgrafo conexo maximal. En la figura, los subgrafos con vértices $\{A, B, E, I, J\}$, $\{C, D, G, H, K, L\}$, $\{F\}$ son los componentes del grafos.

Ejercicio 2.1. *Modificar el algoritmo* DFS *para que determine si el grafo de entrada es o no es conexo.*

Ejercicio 2.2. *Modificar el algoritmo* DFS *para que determine la cantidad de componentes del grafo.*

Ejercicio 2.3. *Modificar el algoritmo* DFS *para que llene un arreglo componente, indexado por los vértices del grafo, tal que componente*$[v]$ = *componente*$[u]$ *si* u *y* v *están en el mismo componente de* G.

2.4 Aplicación: bicolorabilidad

Un grafo G es *bicolorable* si es que podemos asignar dos colores a los vértices de manera tal que ninguna arista tiene dos vértices del mismo color. Más formalmente, si existe una partición de los vértices de G en dos conjuntos A y B tales que A y B son conjuntos independientes.

Considere el siguiente algoritmo que resuelve el problema de decidir si un grafo conexo es bicolorable.

Recibe: Un grafo conexo G.
Devuelve: Verdadero si el grafo G es bicolorable y Falso en caso contrario.

DFS-BICOL(G, c)
1: **for** $v \in V(G)$
2: $\quad visitado(v) =$ Falso
3: **for** $v \in V(G)$
4: $\quad$ **if** NOT $visitado(v)$

```
5:      if NOT Colorear(G, v, 0)
6:         return Falso
7: return Verdadero
```

Colorear(G, v, c)

```
1: col(v) ← c
2: for uv ∈ E(G)
3:   if NOT visitado(u)
4:      if NOT Colorear(G, u, 1 − c)
5:         return Falso
6:   else
7:      if col(u) = col(v)
8:         return Falso
9: return Verdadero
```

Teorema 2.1. *El algoritmo DFS-Bicol hace lo pedido*

Proof. Debemos demostrar que DFS-Bicol devuelve Verdadero si y solo si G es bicolorable. (Solo Si) Suponga que DFS-Bicol devuelve Verdadero. Entonces, al finalizar el algoritmo, el arreglo col guarda una asignación de los vértices de G a dos valores, portanto G es bicolorable.

(Si) Suponga que DFS-Bicol devuelve Falso. Entonces, existen dos vertices u y v en el árbol de búsqueda tal que u es padre de v, uv es una arista de G y $col(u) = col(v)$. En ese caso, existe un circuito de tamaño impar en G y por lo tanto el grafo no es bicolorable. □

Corolario 2.1. *Un grafo es bicolorable si y solo si no tiene circuitos impares.*

Proof. (Solo si) Suponga que el grafo tiene un circuito impar. Suponga por contradicción que existe una bicoloración (A, B) del grafo. Entonces en ese circuito habrían dos vértices en el mismo lado de la bicoloración, contradicción.

(Si) Suponga que el grafo no tiene circuitos impares. Entonces el algoritmo devuelve Verdadero. Por el Teorema anterior, el grafo es bicolorable. □

2.5 Previsit y postvisit

Para cada vértice, guardaremos los momentos de dos importantes eventos: momento en el que el vértice comienza a ser procesado (previsit), y momento en el que deja de ser procesado (postvisit).

Para ello comienze con una variable clock seteada en uno y añada las siguiente líneas entre las líneas 1 y 2 de Explorar.

```
1: pre[v]= clock
2: clock= clock +1
```

y las siguientes línea después de la línea 4 (fuera del if) de Explorar.

```
1: post[v]= clock
2: clock= clock +1
```

Lema 2.1. *Para cada par de vértices u, v, o bien los intervalos $[pre[u], post[u]]$ y $[pre[v], post[v]]$ son disjuntos, o bien uno está contenido en el otro.*

Proof. Sin pérdida de generalidad, tenemos dos casos.

Caso 1: la llamada a EXPLORA(u) fue hecha dentro de la llamada a EXPLORA(v). En ese caso,

$$pre[v] \leq pre[u] \leq post[u] \leq post[v]$$

Caso 2: la llamada a EXPLORA(u) no fue hecha dentro de la llamada a EXPLORA(v). En ese caso, sin pérdida de generalidad,

$$pre[v] \leq post[v] \leq pre[u] \leq post[u]$$

□

3 Caminos mínimos en grafos

Aunque DFS consigue encontrar caminos en su árbol de búsqueda, estos no son necesariamente óptimos. Veremos como encontrar caminos mínimos.

La *distancia* entre dos vértices de un grafo es la longitud (número de aristas) de un camino mínimo entre ellos.

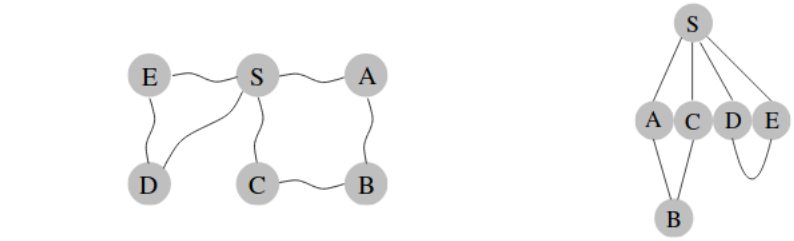

Figure 4: Tomada del libro Dasgupta et al, Algorithms

3.1 Búsqueda en largura

Una manera conveniente de computar las distancias de un vértice s a otros vértices, es proceder por capas: vértices a distancia 1, vértices a distancia 2, etc.

Esto se puede hacer porque es fácil encontrar los vértices a distancia d si se conoce cuales son los vértices a distancia $d-1$.

Recibe: Un grafo G y un vértice $s \in V(G)$. Modifica $dist$ de manera que $dist(v)$ guarda la distancia de s a v en G.

BFS(G, s)

1: **for** $v \in V(G)$
2: $\quad dist(v) = \infty$
3: $dist(s) = 0$
4: $Q = \{s\}$ //cola inicializada en s
5: **while** $Q \neq \emptyset$
6: $\quad u = DESENCOLAR(Q)$
7: $\quad$ **for** $uv \in E(G)$
8: $\quad\quad$ **if** $dist(v) = \infty$
9: $\quad\quad\quad ENCOLAR(Q)$
10: $\quad\quad\quad dist(v) = dist(u) + 1$

Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$

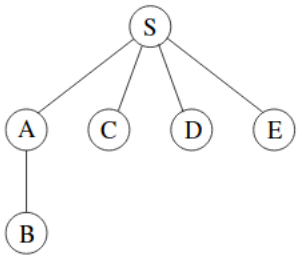

Figure 5: Tomada del libro Dasgupta et al, Algorithms

Probaremos la correctitud del algoritmo.

Mostraremos por inducción en d que para todo $d \leq n$ (donde n es el número de vértices), existe un momento del algoritmo (antes de comenzar el while) que cumple

(a) Todos los vértices a distancia $\leq d$ tienen sus distancias bien seteadas (osea, que $dist(v)$ guarda correctamente el valor)

(b) Todos los vértices a distancia $> d$ tienen su distancia seteada en ∞

(c) La cola tiene exactamente a los nodos con distancia d

Demostración.

Caso base: $d = 0$. Ese momento se da antes de la primera ejecución del while, ya que $Q = \{s\}$, $dist(s) = 0$ y $dist(v) = \infty$ para todo $v \neq s$.

Probaremos para $d > 0$. Por hipótesis de inducción, existe un i tal que al comienzo de la i-ésima iteración del while, se cumplen las tres propiedades, con $d - 1$ en lugar de d. Probaremos que al inicio de la $i + |Q|$-ésima iteración, se cumplen las tres propiedades con d.

- Propiedad (a). Sea un vértice v con distancia $\leq d$. Si su distancia es $\leq d-1$ entonces, por hipótesis de inducción, ya estaba bien seteada antes de la i-ésima iteración del while. Debido a la línea 8 del algoritmo, esta distancia no será modificada, por lo tanto continúa bien seteada.

  Suponga entonces que la distancia del vértice es igual a d. Entonces existe un camino mínimo de s a v con longitud d. Sea v' el vértice adyacente a v en este camino. Note que el subcamino de s a v' también es mínimo y por lo tanto la distancia de s a v' es $d-1$. Por (c), y por hipótesis de inducción, la cola Q contiene a v' antes del comienzo de la iteración i. Luego, en alguna iteración de $i+1$ a $i+|Q|$, por (b), la línea 10 del algoritmo, seteará correctamente la distancia de v.

- Propiedad (b). Sea $v \in V(G)$ con distancia $> d$. Suponga por contradicción que $dist(v) \neq \infty$. Entonces existe u, al inicio de la iteración i, que es vecino de v y está en la cola. En ese caso, $dist(v) \leq dist(u)+1 \leq d$, contradicción.

- Propiedad (c). Como estamos a inicio de la $i+|Q|$-ésima iteración, entonces todos los elementos de Q ya fueron retirados de la cola. En su lugar, se han insertado vértices con distancias $d-1+1=d$. Por (a), estas distancias están bien seteadas, por lo tanto todos los elementos en la cola están a distancia d.

Tiempo de ejecución: $O(|V(G)|+|E(G)|)$. Cada vértice es puesto en la cola exactamente una vez. Por lo tanto hay $|V(G)|$ operaciones con la cola (linea 6). Como cada arista es revisada exactamente dos veces en el for de la línea 7, el tiempo total de las líneas 7-10 es $O(|E(G)|)$.

4 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:

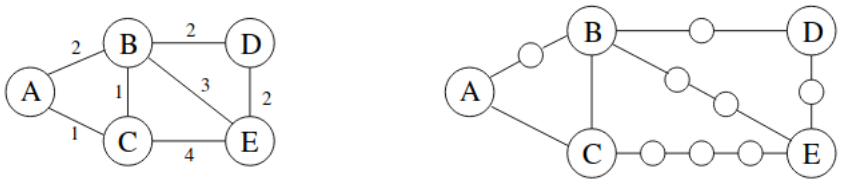

Figure 6: Tomada del libro Dasgupta et al, Algorithms

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 ∈ V(G)
2:     dist(v) = ∞
3: dist(s) = 0
4: R = ∅
5: while R ≠ V(G)
6:     Sea v ∈ V(G) \ R tal que dist(v) = min{dist(w) : w ∈ V(G) \ R}
7:     R = R ∪ {v}
8:     for vz ∈ E(G)
9:         if dist(v) + ℓ_vz < dist(z)
10:            dist(z) = dist(v) + ℓ_vz
```

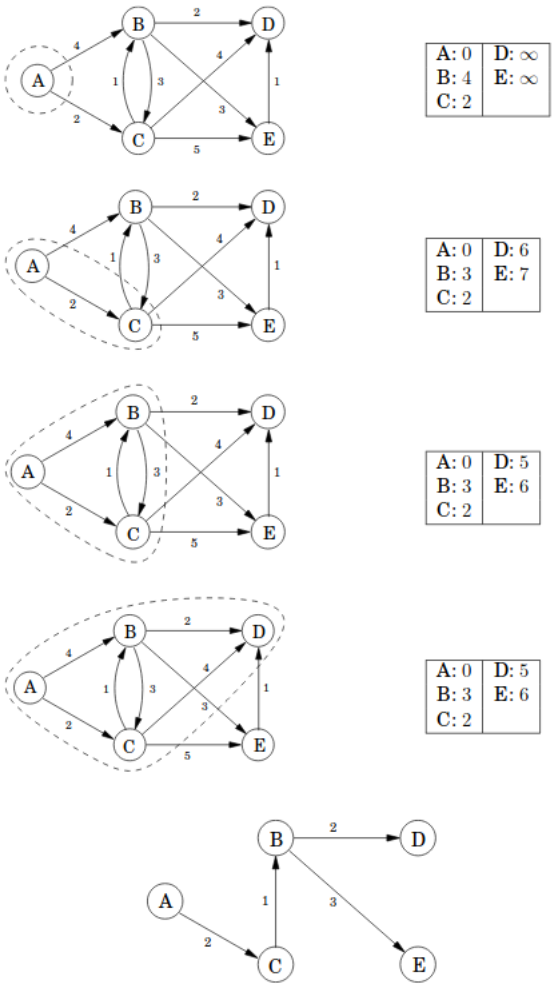

Figure 7: Tomada del libro Dasgupta et al, Algorithms

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: $\quad dist(v) = \infty$
3: $dist(s) = 0$
4: $Q = V(G)$ //fila de prioidades inicializada
5: **while** $Q \neq \emptyset$
6: $\quad v =$ EXTRACT-MIN(Q)
7: $\quad$ **for** $vz \in E(G)$
8: $\quad\quad$ **if** $dist(v) + \ell_{vz} < dist(z)$
9: $\quad\quad\quad dist(z) = dist(v) + \ell_{vz}$
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 8: Tomada del libro Dasgupta et al, Algorithms

5 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.

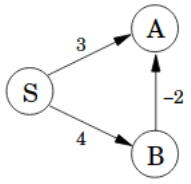

Figure 9: Tomada del libro Dasgupta et al, Algorithms

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: $\quad dist(v) = \infty$
3: $dist(s) = 0$
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.

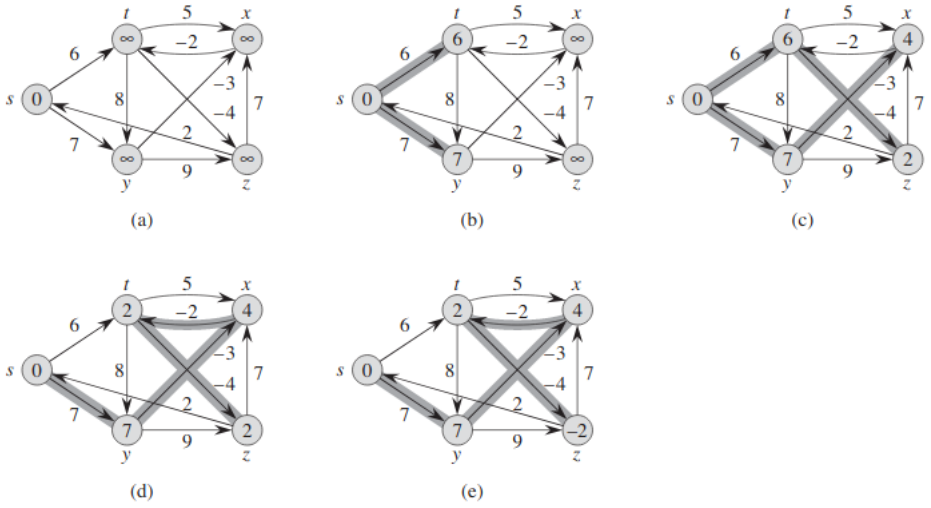

Figure 10: Tomada del libro Cormen et al, Introduction to Algorithms. El orden de procesamiento es $tx, ty, tz, xt, yx, yz, zx, zs, st, sy$.

5.1 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.