Teórica 5: Algoritmos para determinar Caminos Mínimos en Grafos

1. Problemas de camino mínimo en grafos

Si queremos ir desde un punto a otro de una ciudad, caminando o en auto, y tenemos un mapa de las calles de la ciudad con las distancias entre cada par de intersecciones adyacentes, ¿cómo determinamos el camino más corto entre esos dos puntos?

Una posibilidad es enumerar todos los posibles caminos y calcular la distancia que debemos recorrer si seguimos cada uno de ellos, y con esa información seleccionar el más corto. Esta no es una buena estrategia, porque aunque no consideremos los caminos que contienen ciclos, hay una cantidad muy grande de posibilidades, y algunas es obvio que no serán útiles porque se alejan de los puntos de partida y llegada.

Se pueden presentar dos escenarios, uno en el que todas las aristas o arcos tengan igual longitud o peso, y otro donde no sucede ésto, es más, hasta podría haber aristas con peso negativo.

Los grafos (pesados o no) son la estrutura natural para modelar redes en las cuales uno quiere ir de un punto a otro de la red atravesando una secuencia de enlaces. Sobre estas estructuras se han desarrollado algoritmos eficientes para resolver muchos de estos problemas.

Podemos modelar el mapa de la ciudad mediante un grafo, donde los vértices representan las intersecciones de las calles, las aristas (o arcos si es orientado) los segmentos de calle entre dos intersecciones adyacentes y la función de peso la longitud de este segmento. Nuestro objetivo es encontrar un camino mínimo desde el vérticu6e que respresenta la esquina de salida al vértice que representa la de llegada.

El peso de cada arista puede ser interpretado como otras métricas diferentes a la distancia, como el tiempo que lleva recorrerlo, el costo que implica hacerlo o cualquier otra cantidad que se acumule linealmente a lo largo de un camino y que querramos minimizar.

Formalmente, sea $G = (V, X)$ un grafo y $l : X \to \mathbb{R}$ una función de longitud/peso para las aristas de G.

Definición 1.

- La ***longitud*** de un camino C entre dos vértices v y u es la suma de las longitudes de las aristas del camino:

$$l(C) = \sum_{e \in C} l(e)$$

- Un ***camino mínimo*** C^0 entre v y u es un camino entre v y u tal que $l(C^0) = \text{mín}\{l(C) | C \text{ es un camino entre } v \text{ y } u\}$.

Nota: Puede haber varios caminos mínimos.

Dado un grafo G, se pueden definir tres variantes de problemas sobre caminos mínimos:

Único origen - único destino: Determinar un camino mínimo entre dos vértices específicos, v y u.

Único origen - múltiples destinos: Determinar un camino mínimo desde un vértice específico v al resto de los vértices de G.

Múltiples orígenes - múltiples destinos: Determinar un camino mínimo entre todo par de vértices de G.

La versión único origen - único destino pareciera ser más fácil que la versión único origen - múltiples destinos. Sin embargo, no se conocen algoritmos con mejor complejidad para el primer caso que para el segundo, sólo parar la ejecución del algoritmo una vez que se alcanza el vértice de interés.

Una posibilidad para resolver la versión múltiples orígenes - múltiples destinos, es aplicar un algoritmo para el problema único origen - múltiples destinos tomando como vértice origen a cada vértice del grafo. No obstante, para resolver este problema específico se conocen algoritmos que generalmente son más rápidos que aplicar la estrategia anterior.

Todos estos conceptos se pueden adaptar cuando se estudian grafos orientados (digrafos) y en el resto de la clase vamos a trabajar con este tipo de grafos.

Generalmente, los algoritmos para resolver problemas de camino mínimo se basan en que todo subcamino de un camino mínimo entre dos vértices es un camino mínimo.

Proposición 1 (Subestructura óptima de un camino mínimo)**.** *Dado un digrafo $G = (V, X)$ con una función de peso $l : X \to \mathbb{R}$, sea $P : v_1 \dots v_k$ un camino mínimo de v_1 a v_k. Entonces $\forall 1 \leq i \leq j \leq k$, $P_{v_i v_j}$ es un camino mínimo desde v_i a v_j.*

Demostración: Podemos descomponer al camino P en $P_{v_1 v_i} + P_{v_i v_j} + P_{v_j v_k}$, entonces $l(P) = l(P_{v_1 v_i}) + l(P_{v_i v_j}) + l(P_{v_j v_k})$.

Por contradicción, asumamos que $P_{v_i v_j}$ no es un camino mínimo desde v_i a v_j y llamamos P' a un camino mínimo desde v_i a v_j, entonces $l(P') < l(P_{v_i v_j})$. $P'' : P_{v_1 v_i} + P' + P_{v_j v_k}$ es un camino desde v_1 a v_k y $l(P'') = l(P_{v_1 v_i}) + l(P') + l(P_{v_j v_k}) < l(P)$, contradiciendo que P es un camino mínimo desde v_1 a v_k. ■

Dos consideraciones a tener en cuenta:

Aristas con peso negativo: Si el digrafo G no contiene ciclos de peso negativo o contiene alguno pero no es alcanzable desde v, entonces el problema sigue estando bien definido, aunque algunos caminos puedan tener longitud negativa. Sin embargo, si G tiene algún ciclo con peso negativo alcanzable desde v, el concepto de camino de peso mínimo deja de estar bien definido. Si u pertenece a un ciclo de peso negativo, ningún camino de v a u puede ser un camino mínimo porque siempre se puede obtener un camino de peso menor siguiendo ese camino pero atravesando el ciclo negativo una vez más. Esto se puede extender a cualquier vértice u alcanzable desde un vértice del ciclo negativo.

Circuitos: Siempre existe un camino mínimo que no contiene circuitos (si el problema está bien definido). Ya vimos que no puede tener circuitos de longitud negativa porque pierde sentido la definición de camino mínimo. Tampoco puede tener ciclos de longitud positiva, ya que sacando el ciclo obtenemos otro camino desde el mismo origen hasta el mismo destino de menor longitud. Es decir, si $P : v_1 \dots v_k$ es un camino y $C : v_i \dots v_j = v_i$ es un ciclo de longitud positiva dentro de P, el camino $P' : v_1 \dots v_i v_{j+1} \dots v_k$ es un camino desde v_1 hasta v_k con $l(P'') < l(P)$. Lo que implica que P no puede ser camino mínimo desde v_1 a v_k.

La única posibilidad que queda es que el ciclo tenga peso 0. Si eso sucede, removiendo el ciclo del camino obtenemos otro camino con igual origen y destino e igual peso. Entonces, si hay un camino mínimo desde v a u que contiene un ciclo de peso 0, podemos obtener otro camino mínimo desde el mismo origen y destino sin ese ciclo. Repitiendo este procedimiento hasta que no contenga más ciclos de peso 0, obtenemos un camino mínimo sin ciclos. Entonces, en el resto de la clase podemos asumir que un camino mínimo no contiene ciclos.

2. Camino mínimo con único origen y múltiples destinos

Problema: Dado $G = (V, X)$ un digrafo, $l : X \to \mathbb{R}$ una función que asigna a cada arco una cierta longitud y $v \in V$ un vértice del digrafo, el objetivo es calcular los caminos mínimos desde v al resto de los vértices. Asumiremos que todo vértice es alcanzable desde v.

Distintas situaciones:

- Todos los arcos tienen igual longitud o no.
- Todos los arcos tienen longitud no negativa o no.

2.1. Problemas de camino mínimo en digrafos no pesados

En el caso de tener todas los arcos igual longitud, este problema se traduce en encontrar los caminos que definan las distancias (de menor cantidad de aristas/arcos). Para ésto podemos adaptar fácilmente el algoritmo BFS que vimos la clase pasada para calcular tanto la distancia como el camino mínimo desde v al resto de los vértices.

```
BFS(G,v)
    entrada: G = (V,X) de n vertices, un vertice v
    salida: pred[u] = antecesor de u en un camino minimo desde v
            dist[u] = distancia desde v a u

    pred[v] ← 0
    dist[v] ← 0
    COLA ← {v}
    para todo u ∈ V \ {v} hacer
        dist[u] ← ∞
    fin para
    mientras COLA ≠ ∅ hacer
        w ← sacarPrimerElem(COLA)
        para todo u tal que (w → u) ∈ X y dist[u] = ∞ hacer
            pred[u] ← w
            dist[u] ← dist[w] + 1
            insertarAlFinal(COLA, u)
        para
    fin mientras
    retornar pred y dist
```

Lema 1. *Dado $G = (V, X)$ un digrafo y $v \in V$. Sea $COLA = [v_1, \ldots, v_r]$. Se cumple que:*

- $dist[v_1] + 1 \geq dist[v_r]$ *y*
- $dist[v_i] \leq dist[v_{i+1}]$ *para todo* $i = 1, \ldots r - 1$.

Demostración: Haremos inducción en las iteraciones, k.

Caso base: Antes de entrar al ciclo se fija $dist[v] = 0$ y se encola a v. Como hay sólo un elemento en $COLA$ obviamente se cumple la propiedad.

Paso inductivo: Para demostrar el paso inductivo, consideremos la iteración k. Llamemos $COLA_k$, con $k \geq 1$, al valor de la varibale $COLA$ al finalizar la iteración k del algoritmo.

Nuestra hipótesis inductiva es: $COLA_{k'} = [v_1^{k'}, \ldots, v_{r_{k'}}^{k'}]$, $k' < k$ ***cumple que*** $dist[v_1^{k'}] + 1 \geq dist[v_{r_{k'}}^{k'}]$ ***y*** $dist[v_i^{k'}] \leq dist[v_{i+1}^{k'}]$ ***para todo*** $i = 1, \ldots, r_{k'} - 1$.

Si $COLA_{k-1}$ tiene un único elemento (el que se desencola), como todos los vértices que ingresan tienen igual valor de $dist$, se cumple la propiedad.

Ahora supongamos que $COLA_{k-1}$ tiene más de un elemento. Sean $COLA_{k-1} = [u, x, \ldots, z]$ y $w_1, \ldots, w_s$ los vértices que se encolan en la iteración k, es decir $COLA_k = [x, \ldots, z, w_1, \ldots, w_s]$. Por HI, $COLA_{k-1}$ cumple la propiedad. Al desencolar a u se sigue cumpliendo. Para los vértices que se encolan, se fija $dist[w_i] = dist[u] + 1$. Como por HI, $dist[z] \leq dist[u] + 1$, se cumple que $dist[z] \leq dist[w_i]$, $i = 1, \ldots, s$. Además, por HI, $dist[u] \leq dist[x]$. Por lo tanto, $dist[w_s] = dist[u] + 1 \leq dist[x] + 1$.

Esto muestra que $COLA_k$ sigue cumpliendo la propiedad.

■

Corolario 1. *Dado $G = (V, X)$ un digrafo y $v \in V$. Si el vértice u ingresa a $COLA$ antes que el vértice w, en todo momento se cumple que $dist[u] \leq dist[w]$.*

Lema 2. *Dado $G = (V, X)$ un digrafo y $v \in V$. Se cumple que $dist[u] \geq d(v, u)$ para todo $u \in V$ en todo momento del algoritmo.*

Demostración: Haremos inducción en las iteraciones, k.

Caso base: Para $k = 0$, esto es antes de entrar al ciclo, se fija $dist[v] = 0$, $dist[u] = \infty$, para todo $u \in V$, $u \neq v$ y se encola a v. Como $d(v, v) = 0$, se cumple que $dist[u] \geq d(v, u)$ para todo $u \in V$.

Paso inductivo: Para demostrar el paso inductivo, consideremos la iteración k.

Nuestra hipótesis inductiva es: $dist_{k-1}[u] \geq d(v, u)$ ***para todo*** $u \in V$, ***donde*** $dist_{k-1}[u]$ ***es el valor de la variable*** $dist[u]$ ***al finalizar la iteración*** $k - 1$.

Sea w el vértice que se desencola en la iteración k y $w_1, \ldots, w_s$ los vértices que se encolan en esta iteración. El valor de $dist_k[u] = dist_{k-1}[u]$ para todo $u \in V \setminus \{w_1, \ldots, w_s\}$. Por lo tanto, por HI, para esos vértices se cumple que $dist_k[u] \geq d(v, u)$.

Sólo tenemos que analizar $dist_k[w_i]$ para $i = 1, \ldots, s$. El algoritmo fija $dist_k[w_i] = dist_{k-1}[w] + 1$. Por HI, sabemos que $dist_{k-1}[w] \geq d(v, w)$. Por otro lado, como $(w \to w_i) \in X$, por desigualdad triangular de la función distancia, $d(v, w_i) \leq d(v, w) + 1$. Entonces $d(v, w_i) \leq d(v, w) + 1 \leq dist_{k-1}[w] + 1 = dist_k[w_i]$, para $i = 1, \ldots, s$.

■

Teorema 1. *Dado $G = (V, X)$ un digrafo y $v \in V$. El algoritmo BFS enunciado calcula $d(v, u)$ para todo $u \in V$.*

Demostración: Por Lema 2, sabemos que $dist[u] \geq d(v, u)$ para todo $u \in V$. Por contradicción, supongamos que para algún vértice no se cumple la igualdad y sea w el vértice con menor $d(v, w)$ tal que $dist[w] > d(v, w)$. Ese

vértice w seguro no es v, porque $dist[v] = 0$, al igual que $d(v, v)$.

Como todo vértice de G es alcanzable desde v, sea $P = v, \ldots, x, w$ (x podría ser v) un camino de mínima cantidad de arcos desde v a w. Entonces se cumple que $d(v, w) = d(v, x) + 1$. Como $d(v, x) < d(v, w)$, por la elección de w se cumple que $dist[x] = d(v, x)$ (w el vértice con menor $d(v, w)$ tal que $dist[w] > d(v, w)$).

Cuando se desencola x, se analiza w porque $(x \to w) \in X$. En ese momento tiene que pasar una de estas tres situaciones: w ya fue desencolado de $COLA$; w está en $COLA$; o w todavía no entró en $COLA$. Analicemos las tres posibilidades.

- Si w ya fue desencolado, entonces, por corolario 1, $dist[w] \leq dist[x]$. Esto es absurdo porque $d(v, w) = d(v, x) + 1 = dist[x] + 1$ y por Lema 2 $dist[w] \geq d(v, w)$, lo que implica que $dist[w] > dist[x]$.
- Si w está en $COLA$, por Lema 1, $dist[w] \leq dist[x]+1$. Esto también es absurdo, porque $dist[w] \leq dist[x]+1 = d(v, x) + 1 = d(v, w)$, contradiciendo que $dist[w] > d(v, w)$.
- Si w todavía no fue encolado, el algoritmo fijará $dist[w] = dist[x]+1$ y encolará a w. Esto también es absurdo, ya que $dist[w] = dist[x] + 1 = d(v, x) + 1 = d(v, w)$, contradiciendo que $dist[w] > d(v, w)$.

Esto demuestra que no existe vértice w tal que $dist[w] > d(v, w)$. Por lo tanto, $dist[u] = d(v, u)$, para todo $u \in V$.

■

La complejidad de este algoritmo es $\mathcal{O}(m)$, donde m es la cantidad de arcos del digrafo, ya que se examina cada arco exactamente una vez. Si es grafo no fuera dirigido, cada arista se examinaría dos veces, siendo también $\mathcal{O}(m)$.

2.2. Problemas de camino mínimo en digrafos pesados

En la gran mayoría de las situaciones que se quieren modelar, no se cumple que todos los arcos del digrafo tengan igual longitud o peso, lo que hace no aplicable BFS. Además, dependiendo de qué represente la función de peso, podría ser que algunos de estos valores sean negativos.

Vamos a ver dos algoritmos. El primero, es aplicable a grafos o digrafos con todos pesos positivos en sus aristas o arcos, mientras que el segundo también es correcto cuando hay valores negativos.

Algoritmo de Dijkstra (1959)

Este algoritmo asume que las longitudes de los arcos son positivas. El grafo puede ser orientado o no orientado.

Es un algoritmo goloso que construye un árbol de caminos mínimos (un árbol enraizado en v conteniendo un camino mínimo desde v hacia todo otro vértice de V), comenzando con el vértice v y agregando un vértice a este árbol en cada iteración. Es goloso porque en cada iteración agrega el vértice más cercano a v de entre todos los que todavía no fueron agregados al árbol. Es decir, en la iteración k agrega al árbol de caminos mínimos el k-ésimo vértice más cercano a v.

El algoritmo mantiene un conjunto S de vértices que ya han sido incorporados al árbol de caminos mínimos y cuya distancia está almacenada en el vector π. Inicialmente $S = \{v\}$ y $\pi[v] = 0$. Luego, para cada vértice $u \in V \setminus S$, determina el camino mínimo que puede ser construido siguiendo un camino dentro de S desde v hasta a algún $z \in S$ y luego el arco $(z \to u)$.

Es decir, $\forall u \in V \setminus S$ considera el valor $\pi'[u] = \min_{(z \to u) \in X, z \in S} \pi[z] + l((z,u))$ y elige el vértice w para el cual este valor es mínimo. Agrega w a S y fija $\pi[w] = \pi'[w]$.

Cuando se agrega el vértice w a S, se actualiza π' para los adyacentes de w que no están en S. En la implementación no es necesario utilizar dos vectores distintos, π y π', ya que en todo momento cada vértice tendrá sólo uno de estos valores activo (π para los $u \in S$ y π' para los $u \in V \setminus S$).

El pseudocódigo del algoritmo es el siguiente:

```
Dijkstra(G, v)
    entrada: G = (V, X) de n vertices, un vertice v
    salida: π[u] = valor del camino minimo desde v a u

    S ← {v}, π[v] ← 0
    para todo u ∈ V hacer
        si (v → u) ∈ X entonces
            π[u] ← l((v → u))
        si no
            π[u] ← ∞
        fin si
    fin para
    mientras S ≠ V hacer
        w ← arg mín{π[u], u ∈ V \ S}
        S ← S ∪ {w}
        para todo u ∈ V \ S y (w → u) ∈ X hacer
            si π[u] > π[w] + l((w → u)) entonces
                π[u] ← π[w] + l((w → u))
            fin si
        fin para
    fin mientras
    retornar π
```

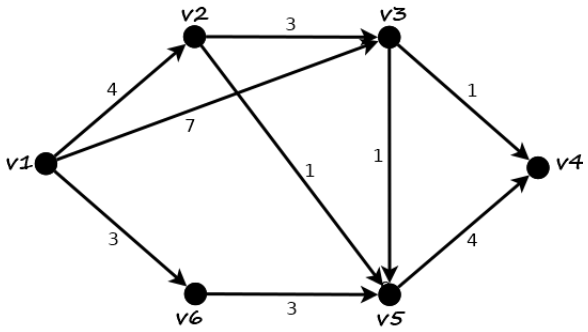
Si, además de la longitud de un camino mínimo, queremos obtener el camino correspondiente, podemos mantener un vector *pred*, donde para cada vértice $w \in V$ en $pred[w]$ se guarda su predecesor en el camino mínimo desde v hacia él. Cuando se agrega el vértice w a S, se guarda en $pred[w]$ el vértice z sobre el cual se alcanzó el valor $\min_{(z \to u) \in X, z \in S} \pi[z] + l((z \to u))$. El camino mínimo desde v a u es implícitamente representado por *pred*. Al finalizar el algoritmo, siguiendo la cadena de predecesores originada desde $pred[w]$ obtenemos hacia atrás recursivamente este camino mínimo hasta alcanzar v.

El pseudocódigo de esta opción es:

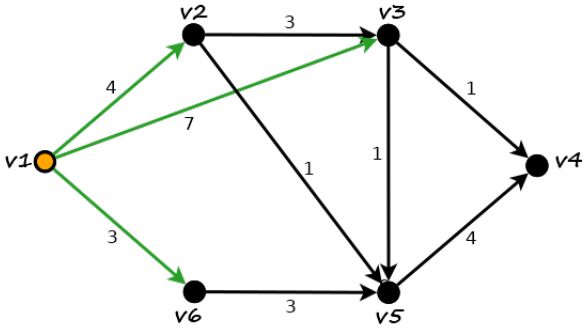
```
Dijkstra(G, v)
    entrada: G = (V, X) de n vertices, un vertice v
    salida: π[u] = valor del camino minimo desde v a u
            pred[u] = predecesor de u en un camino minimo desde v a u

    S ← {v}, π(v) ← 0, pred[v] ← 0
    para todo u ∈ V hacer
        si (v → u) ∈ X entonces
            π[u] ← l((v → u)), pred[u] ← v
        si no
            π[u] ← ∞, pred[u] ← ∞
        fin si
    fin para
    mientras S ≠ V hacer
        w ← arg mín{π[u], u ∈ V \ S}
        S ← S ∪ {w}
        para todo u ∈ V \ S y (w → u) ∈ X hacer
            si π[u] > π[w] + l((w → u)) entonces
                π[u] ← π[w] + l((w → u))
                pred[u] ← w
            fin si
        fin para
    fin mientras
    retornar π, pred
```

Ejemplo 1. *En el ejemplo, se marca en naranja los vértices que están en S. También se marca en naranja los arcos que definen el vector pred para los vértices de S. En verde se marcan los arcos candidatos de pred, es decir pred para los vértices de $V \setminus S$.*

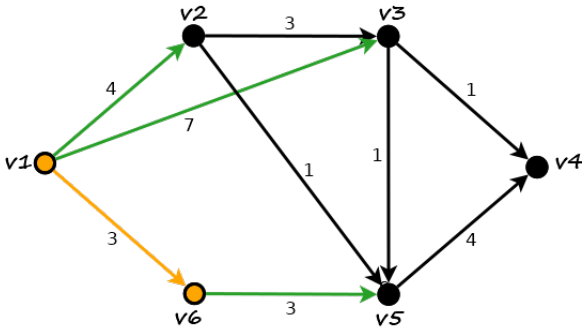
G



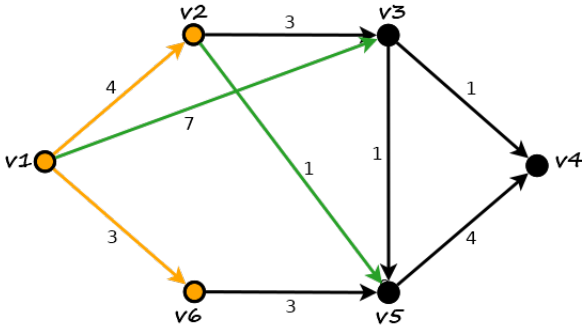
Iteración 0: $S = \{1\}$ $\pi = (0, 4, 7, \infty, \infty, 3)$



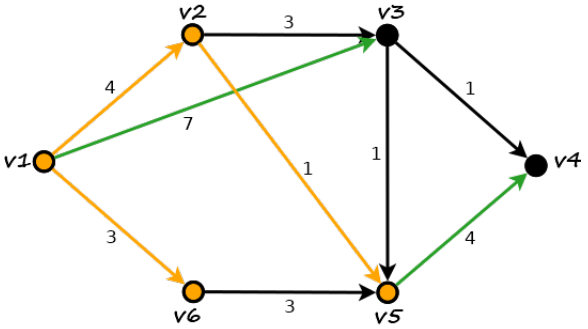
Iteración 1: $S = \{1, 6\}$ $\pi = (0, 4, 7, \infty, 6, 3)$



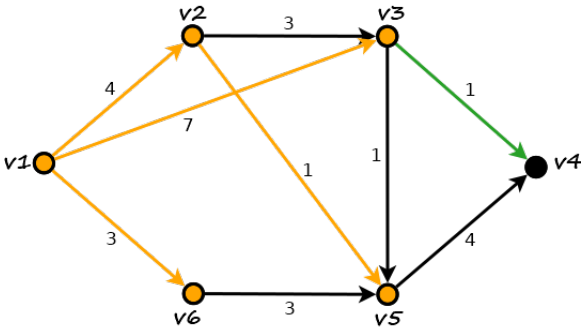
Iteración 2: $S = \{1, 6, 2\}$ $\pi = (0, 4, 7, \infty, 5, 3)$



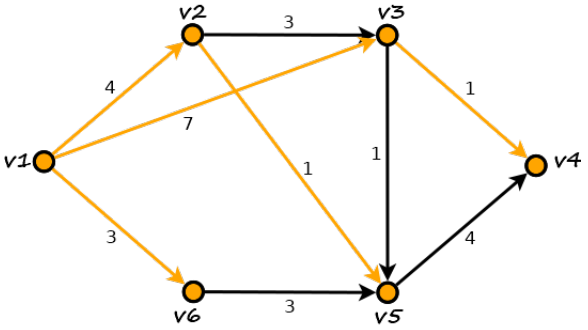
Iteración 3: $S = \{1, 6, 2, 5\}$ $\pi = (0, 4, 7, 9, 5, 3)$



Iteración 4: $S = \{1, 6, 2, 5, 3\}$ $\pi = (0, 4, 7, 8, 5, 3)$



Iteración 5: $S = \{1, 6, 2, 5, 3, 4\}$ $\pi = (0, 4, 7, 8, 5, 3)$



Lema 3. *Dado un digrafo $G = (V, X)$ con pesos positivos en las arcos, al finalizar la iteración k el algoritmo de Dijkstra determina, siguiendo* hacia atrás *pred hasta llegar a v, un camino mínimo entre el vértice v y cada vértice u de S_k, con longitud $\pi[u]$ (siendo S_k el valor del conjunto S al finalizar la iteración k).*

Demostración: Haremos inducción en las iteraciones, k.

Caso base: Antes de entrar por primera vez al ciclo, $k = 0$, vale $S_0 = \{v\}$ y $\pi[v] = 0$. Esto es correcto ya que el camino mínimo desde v hasta v es 0 por definición.

Paso inductivo: Consideremos una iteración k, $k \geq 1$.

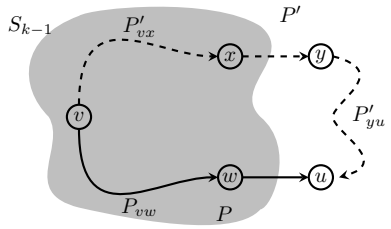
La hipótesis inductiva es: ***Al terminar la iteración*** k' ***(***$k' < k$***),*** $\forall u \in S_{k'}$***,*** $\pi[u]$ ***es la longitud de un camino mínimo de*** v ***a*** u ***finalizado en*** $pred[u]$***.***

Sea u el vértice agregado a S en la iteración k, $S_k = S_{k-1} \cup \{u\}$, con $pred[u] = w$. Como $w \in S_{k-1}$, por HI, $\pi[w]$ es el valor de un camino mínimo desde v a w definido por $pred$. Llamemos P_{vw} a ese camino mínimo desde v a w ($l(P_{vw} = \pi[w])$. Definimos $P = P_{vw} + (w \to u)$.

Sea P' otro camino de v a u y y el primer vértice de ese camino que no está en S_{k-1} (debe existir porque $v \in S_{k-1}$ y $u \notin S_{k-1}$) y x el inmediatamente anterior.

Si y es u, entonces $l(P') = \pi[x] + l((x \to u))$. Si x ingresó a S después que el u, en ese momento el algoritmo comparó $\pi[u]$ (que era $\pi[w] + l((w \to u))$ por ser $pred[u] = w$) con $\pi[x] + l((x \to u))$ y no actualizó $pred[u]$, por lo que asegura que $\pi[x] + l((x \to u)) \geq \pi[u]$. Si x ingresó a S antes que w, al momento de ingresar a w actualizó $pred[u]$, lo que también asegura que $\pi[x] + l((x \to u)) \geq \pi[u]$. Demostrando que $l(P) \leq l(P')$.

Ahora supongamos que y no es u. Tenemos el siguiente esquema:



En la iteración k los vértices y y u son candidatos para entrar a S y el algoritmo elige a u, lo que implica que $\pi[w] + l((w \to u)) = \pi_{k-1}[u] \leq \pi_{k-1}[y]$. Como $x \in S_{k-1}$, $\pi_{k-1}[y] \leq \pi[x] + l((x \to y))$ (hizo esta comparación cuando ingresó x a S) y, por HI, $\pi[x]$ es la longitud de un camino mínimo desde v a x, tenemos $l(P) = \pi[w] + l((w \to u)) = \pi_{k-1}[u] \leq \pi_{k-1}[y] \leq \pi[x] + l((x \to y)) \leq l(P'_{vx}) + l((x \to y)) \leq l(P'_{vy})$. Como ***no hay arcos con peso negativo***, se cumple que $l(P'_{vy}) \leq l(P')$.

Entonces $l(P) \leq l(P')$ para todo camino P' desde v a u y $\pi[u]$ es la longitud de un camino mínimo desde v a u.

■

Teorema 2. *Dado un digrafo $G = (V, X)$ con pesos positivos en las arcos y $v \in V$, el algoritmo de Dijkstra determina el camino mínimo entre el vértice v y el resto de los vértices.*

Demostración: En cada iteración un nuevo vértice es incorporado a S y el algorimto termina cuando $S = V$. Aplicando el Lema 3 al finalizar la última iteración del ciclo, el algoritmo de Dijkstra determina, siguiendo *hacia atrás pred* hasta llegar a v, un camino mínimo entre el vértice v y cada vértice u de V, con longitud $\pi[u]$. ■

Analicemos la complejidad de este algoritmo. Los pasos computacionales críticos son:

1. encontrar el próximo vértice a agregar a S,
2. actualizar π.

Cada uno de estos pasos se realiza n veces. La forma más fácil de implementar (1) es buscar secuencialmente el vértice que minimiza, ésto se hace en $\mathcal{O}(n)$. En el paso (2), el vértice elegido tiene a lo sumo n adyacentes y para cada uno podemos actualizar π en $\mathcal{O}(1)$. Considerando ésto, cada iteración es $\mathcal{O}(n)$, resultando $\mathcal{O}(n^2)$ el algoritmo completo.

Podemos ser más cuidadosos en el cálculo de (2), teniendo en cuenta que en total (no por cada iteración) se realiza m veces. Con ésto el algoritmo completo es $\mathcal{O}(m+n^2)$ que sigue siendo $\mathcal{O}(n^2)$. Ésto sugiere que si queremos mejorar su complejidad, debemos revisar la implementación del paso (1).

Para mejorar (1), se debe utilizar una estructura de datos que permita encontrar en forma más eficiente el mínimo. Por ejemplo, utilizando una cola de prioridades sobre heap con n elementos, crearla es $\mathcal{O}(n)$. Luego, es posible borrar el elemento mínimo e insertar uno nuevo en $\mathcal{O}(\log_n)$. Si se mantiene un arreglo auxiliar apuntando a la posición actual de cada vértice en el heap, también es posible modificar el valor de π de un vértice en $\mathcal{O}(\log_n)$.

Considerando todas las iteraciones, la operación (1) es $\mathcal{O}(nlog_n)$. Cada aplicación del paso (2) requiere, a lo sumo, $d(u)$ inserciones o modificaciones por cada vértice elegido u. Cada una de estas operaciones es $\mathcal{O}(log_n)$, dando en total $\mathcal{O}(d(u)log_n)$. Sumando sobre todas las iteraciones, la operación (2) es $\mathcal{O}(mlog_n)$. El algoritmo completo resultan $\mathcal{O}(n \log_n + m \log_n)$, que es $\mathcal{O}(m \log_n)$.

Esto es mejor que $\mathcal{O}(n^2)$ cuando m es $\mathcal{O}(n)$ (grafos ralos), pero peor si m es $\mathcal{O}(n^2)$ (grafos densos).

Algoritmo de Bellman - Ford (1956)

Este algoritmo resuelve, utilizando la técnica de programación dinámica, el problema de camino mínimo con único origen (v) - múltiples destinos, pero también es correcto para el caso general en el que el digrafo puede tener arcos de longitud negativa. En la primer versión del algoritmo vamos a asumir que no hay ciclos de longitud negativa alcanzables desde v.

Para cada vértice u, en $\pi[u]$ va guardando una cota superior de la longitud de un camino mínimo desde v a u, hasta obtener la verdadera longitud. El algoritmo considera sucesivamente caminos con más cantidad de arcos. Al terminar la k-ésima iteración, el algoritmo encuentra todos los caminos mínimos desde v con, a lo sumo, k arcos.

En cada iteración se revisan todos los arcos, y, si agregar un arco $(w \to u)$ al camino hasta w obtenido hasta el momento es más corto que ese camino, actualiza el valor del camino hasta u, $\pi(u)$, poniendo a w como su predecesor. El algoritmo se detiene cuando en una iteración no es posible mejorar ningún camino. Como los caminos mínimos son simples, a lo sumo realiza $n-1$ iteraciones.

El pseudocódigo es el siguiente:

```
BellmanFord(G, v)
    entrada: G = (V, X) de n vertices, un vertice v
    salida: π[u] = valor del camino minimo desde v a u

    π[v] ← 0
    para todo u ∈ V \ {v} hacer
        π[u] ← ∞
    fin para
    mientras hay cambios en π hacer
        π' ← π
    para todo  u ∈ V \ {v} hacer
        π[u] ← mín(π'[u], mín_{(w→u)∈X} π'[w] + l((w → u)))
        fin para
    fin mientras
    retornar π
```

Lema 4. *Dados un digrafo $G = (V, X)$ y $l : X \to \mathbb{R}$ una función de peso para las aristas de G. Si G no tiene circuitos de longitud negativa, al finalizar la iteración k el algoritmo de Bellman-Ford determina los caminos mínimos de a lo sumo k arcos entre el vértice v y los demás vértices.*

Demostración: Inducción en las iteraciones, k.

Caso base: Antes de ingresar por primera vez al ciclo, $k = 0$, el algoritmo fija $\pi[v] = 0$ y $\pi[u] = \infty$ para todo $u \in V \setminus \{v\}$. Ésto es correcto ya que el único camino posible con 0 arcos es de v a v.

Paso inductivo: Consideremos la iteración $k \geq 1$ del algoritmo.

La hipótesis inductiva es: ***Al terminar la iteración k' $(k' < k)$ el algoritmo determina en $\pi^{k'}$ las longitudes de los caminos mínimos desde v al resto de los vértices con a lo sumo k' arcos.***

Sea u un vértice tal que el camino mínimo desde v a u con menos cantidad de arcos contiene k arcos y P uno de estos caminos. Sea w el predecesor de u en P. Por optimalidad de subcaminos, P_{vw} es un camino mínimo desde v a w con $k-1$ arcos. Por HI este camino es correctamente identificado al terminar la iteración $k-1$ del algoritmo.

Luego, en la iteración k, $\pi[u]$ recibe el valor correcto cuando es examinado el arco $(w \to u)$. Como este valor no puede ser mejorado (porque implicaría que P no es camino mínimo), no será modificado en las iteraciones sucesivas del algoritmo. Luego, el algoritmo calcula correctamente los caminos mínimos desde v al resto de los vértices.

■

Teorema 3. *Dados un digrafo $G = (V, X)$ y $l : X \to \mathbb{R}$ una función de peso para las aristas de G. Si G no tiene circuitos de longitud negativa, el algoritmo de Bellman-Ford determina el camino mínimo entre el vértice v y todos los demás vértices.*

Este algoritmo es $\mathcal{O}(nm)$. La inicialización en el primer **para** toma $\mathcal{O}(n)$, el **mientras** a lo sumo lo repite n veces y en cada iteración de éste el **para** interior lo realiza m veces.

Es posible acelerar el cálculo de π. Si $v_1, \ldots, v_n$ es el orden en que los vértices son considerados, para calcular $\pi[v_i]$ se puede usar el valor de $\pi[v_j]$ con $j < i$ calculados al comienzo de esa iteración.

Si el grafo tiene ciclos de longitud negativa alcanzables desde v, el algoritmo no termina después de $n-1$ iteraciones. Podemos modificarlo para controlar esto. Con estas modificaciones, el algoritmo es el siguiente:

```
BellmanFord(G, v)
    entrada: G = (V, X) de n vertices, un vertice v
    salida: π[u] = valor del camino minimo desde v a u

    π[v] ← 0, i ← 0
    para todo u ∈ V \ {v} hacer
        π[u] ← ∞
    fin para
    mientras hay cambios en π y i < n hacer
        i ← i + 1
    para todo u ∈ V \ {v} hacer
        π[u] ← mín(π[u], mín_{(w→u)∈X} π[w] + l((w → u)))
        fin para
    fin mientras
    si i = n hacer
        retornar "Hay circuitos de longitud negativa."
    si no
        retornar π
    fin si
```

3. Camino mínimo para múltiples orígenes - Algoritmos matriciales

Sea $G = (V, X)$ un digrafo de n vértices y $l : X \to \mathbb{R}$ una función de peso para las aristas de G. Definimos las siguientes matrices:

- $L \in \mathbb{R}^{n \times n}$, donde los elementos l_{ij} de L se definen como:

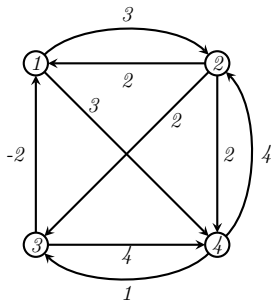
$$l_{ij} = \begin{cases} 0 & \text{si } i = j \\ l((v_i \to v_j)) & \text{si } (v_i \to v_j) \in X \\ \infty & \text{si } (v_i \to v_j) \notin X \end{cases}$$

- $D \in \mathbb{R}^{n \times n}$, donde los elementos d_{ij} de D se definene como:

$$d_{ij} = \begin{cases} \text{longitud del camino mínimo orientado de } v_i \text{ a } v_j & \text{si existe alguno} \\ \infty & \text{si no} \end{cases}$$

  D es llamada matriz de distancias de G.

Ejemplo 2.



$L =$

	1	2	3	4
1	0	3	∞	*3*
2	*2*	*0*	*2*	*2*
3	*-2*	∞	*0*	*1*
4	∞	4	4	0

3.1. Algoritmo de Floyd (1962)

El algoritmo de Floyd calcula el camino mínimo entre todo par de vértices de un digrafo pesado. Utiliza la técnica de programación dinámica y se basa en lo siguiente:

1. Si $D^0 = L$ y calculamos L^1 como
$$d^1_{ij} = \text{mín}(d^0_{ij}, d^0_{i1} + d^0_{1j})$$
d^1_{ij} es la longitud de un camino mínimo de v_i a v_j con vértice intermedio v_1 o directo.
2. Si calculamos D^k a partir de D^{k-1} como
$$d^k_{ij} = \text{mín}(d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj})$$
d^k_{ij} es la longitud de un camino mínimo de v_i a v_j cuyos vértices intermedios están en $\{v_1, \ldots, v_k\}$.
3. $D = D^n$

Primero asumiremos que no hay circuitos de longitud negativa.

```
Floyd(G)
    entrada: G = (V, X) de n vertices
    salida: D matriz de distancias de G

    D ← L
    para k desde 1 a n hacer
        para i desde 1 a n hacer
            para j desde 1 a n hacer
                d[i][j] ← mín(d[i][j], d[i][k] + d[k][j])
            fin para
        fin para
    fin para
    retornar D
```

Lema 5. *Al finalizar la iteración k del algoritmo de Floyd, $d[i][j]$ es la longitud de los caminos mínimos desde v_i a v_j cuyos vértices intermedios son elementos de $\{v_1, \ldots, v_k\}$.*

Demostración: Haremos inducción en las iteraciones, k.

Caso base: Antes de entrar por primera vez al ciclo, $k = 0$, vale $D = L$. Esto es correcto ya que son las longitudes de los caminos mínimos sin vértices intermedios, es decir, caminos que sólo son un arco.

Paso inductivo: Consideremos una iteración k, $k \geq 1$.

La hipótesis inductiva es: ***Al terminar la iteración*** k' $(k' < k)$, $D^{k'}$ ***tiene las longitudes de los caminos mínimos entre todos los pares de vértices cuyos vértices intermedios son elementos de*** $\{v_1, \ldots, v_{k'}\}$.

Dados dos vértices v_i y v_j, el camino mínimo de v_i a v_j cuyos vértices intermedios están $\{v_1, \ldots, v_k\}$, P, tiene a v_k o no tiene a v_k. Analicemos un camino mínimo, P^1, desde v_i a v_j que pasa por v_k y cuyos vértices intermedios son elementos del conjunto $\{v_1, \ldots, v_k\}$ y un camino mínimo, P^2, con vértices intermedios en el conjunto $\{v_1, \ldots, v_{k-1}\}$.

- $P^1 = P^1_{v_i v_k} + P^1_{v_k v_j}$. Por Proposición 1 (subestructura óptima de un camino mínimo), tanto $P^1_{v_i v_k}$ y $P^1_{v_k v_j}$ son caminos mínimos de v_i a v_k y de v_k a v_j respectivamente con vértices intermedios en $\{v_1, \ldots, v_{k-1}\}$. Por HI, $P_{v_i v_k}$ tiene longitud $d^{k-1}[i][k]$ y $P_{v_k v_j}$ longitud $d^{k-1}[k][j]$. Por lo tanto $l(P^1) = d^{k-1}[i][k] + d^{k-1}[k][j]$.
- Por HI, $l(P^2) = d^{k-1}[i][j]$.

P es el más corto entre P^1 y P^2. Por lo tanto, $l(P) = \text{mín}(l(P^1), l(P^2)) = \text{mín}(d^{k-1}[i][k]+d^{k-1}[k][j], d^{k-1}[i][j])$, que es el cálculo que realiza el algoritmo.

■

Teorema 4. *El algoritmo de Floyd determina los caminos mínimos entre todos los pares de nodos de un grafo orientado sin circuitos negativos.*

La complejidad del algoritmo de Floyd es $\mathcal{O}(n^3)$.

Si no sólo queremos conocer la longitud de los caminos mínimos, si no también un camino mínimo para cada par de vértices, debemos guardar la información sobre los vértices intermedios. Para ésto se puede utilizar una matriz *pred* que indica para cada par de vértices v_i, v_j el vértice anterior a v_j en el camino mínimo actual de v_i a v_j. La inicialización es $pred[i][j] \leftarrow i$ y cuando se actualiza $d[i][j]$, se debe fijar $pred[i][j] \leftarrow pred[k][j]$.

Si en el curso del algoritmo, para algún vértice i, $d[i][i] < 0$ significa que hay al menos un ciclo negativo que pasa por i. Se puede mejorar la eficiencia del algoritmo agregando dos chequeos que permiten no realizar algunas comparaciones y para inmediatamente si $d[i][i]$ se hace negativa para algún i. Con estas modificaciones, el algoritmo es:

```
Floyd(G)
  entrada: G = (V, X) de n vertices
  salida: D matriz de distancias de G

  D ← L
  para k desde 1 a n hacer
      para i desde 1 a n hacer
          si d[i][k] ≠ ∞ entonces
              si d[i][k] + d[k][i] < 0 entonces
                  retornar "Hay circuitos negativos"
              fin si
              para j desde 1 a n hacer
                  d[i][j] ← mín(d[i][j], d[i][k] + d[k][j])
              fin para
          fin si
      fin para
  fin para
  retornar D
```

3.2. Algoritmo de Dantzig (1966)

En la iteración k, el algoritmo de Dantzig genera una matriz de $k \times k$ de caminos mínimos en el subgrafo inducido por los vértices $\{v_1, \ldots, v_k\}$.

Calcula la matriz D^k a partir de la matriz D^{k-1} para $1 \leq i, j \leq k$ como:

- $d^k_{ik} = \text{mín}_{1 \leq j \leq k-1}\{d^{k-1}_{ij} + l((j \to k))\}$
- $d^k_{ki} = \text{mín}_{1 \leq j \leq k-1}\{l((k \to j)) + d^{k-1}_{ji}\}$
- $d^k_{ij} = \text{mín}(d^{k-1}_{ij}, {}_{ik}^{k} + d^k_{kj})$

Finalmente, $D = D^n$.
Asumimos que el grafo es orientado. Detecta si hay circuitos de longitud negativa.

```
Dantzig(G)
  entrada: G = (V, X) de n vertices
  salida: D matriz de distancias de G

  D ← L
  para k desde 2 a n hacer
      para i desde 1 a k hacer
          D[i][k] ← mín_{1≤j≤k−1}{D[i][j] + D[j][k]}
          D[k][i] ← mín_{1≤j≤k−1}{D[k][j] + D[j][i]}
      fin para
      t ← mín_{i≤j≤k−1}{D[k][i] + D[i][k]}
      si t < 0 entonces
          retornar "Hay circuitos de longitud negativa"
      fin si
      para i desde 1 a k − 1 hacer
          para j desde 1 a k − 1 hacer
              D[i][j] ← mín(D[i][j], D[i][k] + D[k][j])
          fin para
      fin para
  fin para
  retornar D
```

Lema 6. *Al finalizar la iteración k del algoritmo de Dantzig, la matriz de $k \times k$ generada contiene la longitud de los caminos mínimos en el subgrafo inducido por los vértices $\{v_1, \ldots, v_k\}$.*

Teorema 5. *El algoritmo de Dantzig determina los caminos mńimos entre todos los pares de nodos de un grafo orientado sin circuitos.*

La complejidad del algoritmo de Dantzig es $\mathcal{O}(n^3)$.