Resumen Recorrido Mínimo

Tomás Felipe Melli

Noviembre 2024

Índice

1 Introducción

Sea $G = (V, X)$ un grafo y $l : X \to R$ una función de longitud/peso para las aristas de G.
Definiciones:

- La longitud de un recorrido R entre dos nodos v y u es la suma de las longitudes de las aristas del R:

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

- Un recorrido mínimo R_0 entre u y v es un recorrido entre u y v tal que

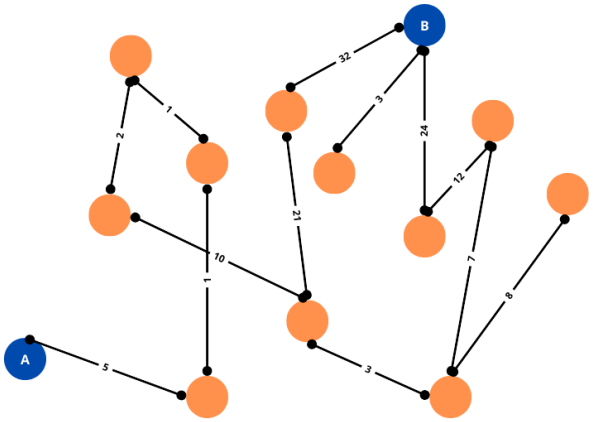
$$l(R_0) = \min\{l(R) \mid R \text{ es un recorrido entre } u \text{ y } v\}.$$

- Si un recorrido mínimo entre un par de nodos es un camino, entonces se lo llama como **camino mínimo** entre ese par de nodos. La existencia de recorridos mínimos es equivalente a la existencia de caminos mínimos. *Puede haber varios caminos mínimos.*

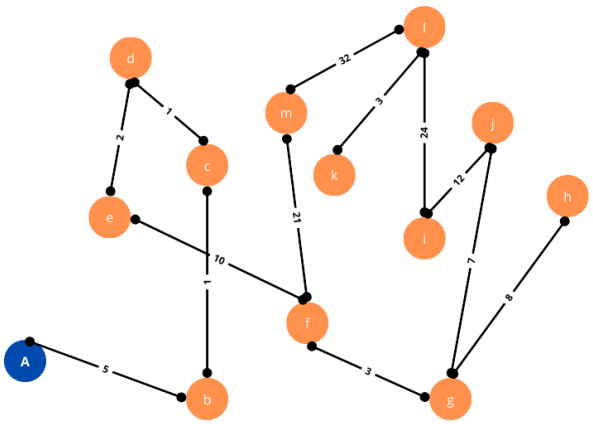
- La distancia entre u y v, $\text{dist}(u, v)$, es la longitud de un camino mínimo entre u y v en caso de existir algún camino entre u y v; en caso contrario, sería ∞.

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

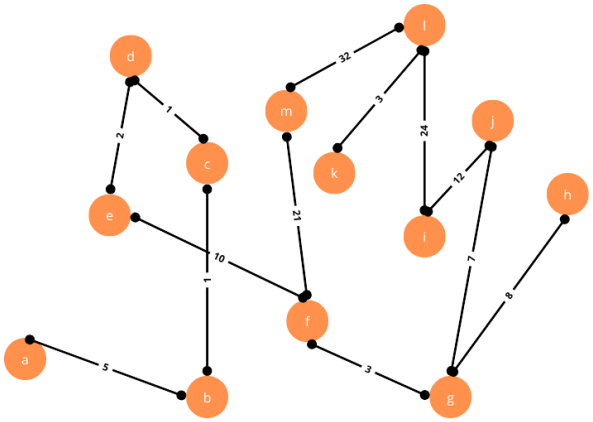
- **Único origen - único destino:** Determinar un camino mínimo entre dos vértices específicos, v y u. Llamado **Single-Source Single-Destination (SSSD)**.



- **Único origen - múltiples destinos:** Determinar un camino mínimo desde un vértice específico v al resto de los vértices de G. Llamado **Single-Source All-Destinations (SSAD)** o también *Single-Source Shortest Path (SSSP)*.

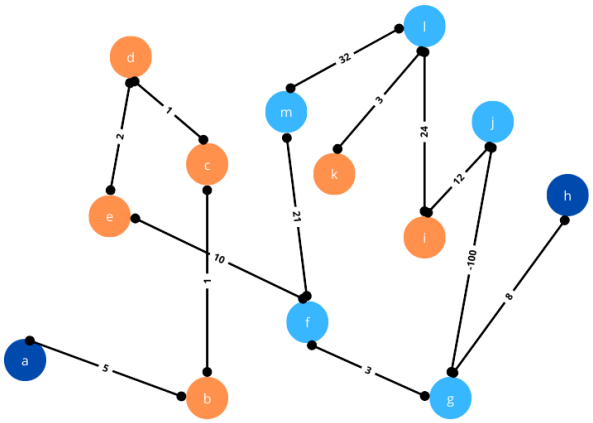


- **Múltiples orígenes - múltiples destinos:** Determinar un camino mínimo entre todo par de vértices de G. Llamado **All-Pairs Shortest Paths (APSP)**

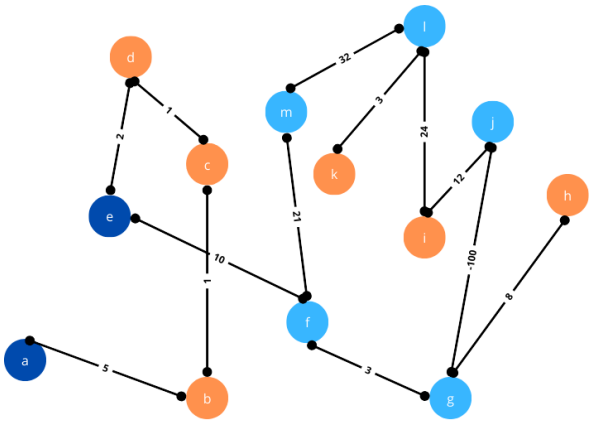


Qué pasa si en el grafo tenemos una arista con peso negativo ?

- **Si esa arista forma un ciclo y es alcanzable desde el vértice de origen** entonces el problema se **indefine**. Esto se debe a que el concepto de camino mínimo se indefine. Al incluir el ciclo negativo en el camino, podríamos recorrerlo indefinidamente para reducir el peso total del camino sin límite.



- **Si esa arista forma ciclo, pero no es alcanzable desde el vértice de origen** entonces el problema está bien definido. Esto se debe a que no podemos indefinidamente reducir el peso de un camino.



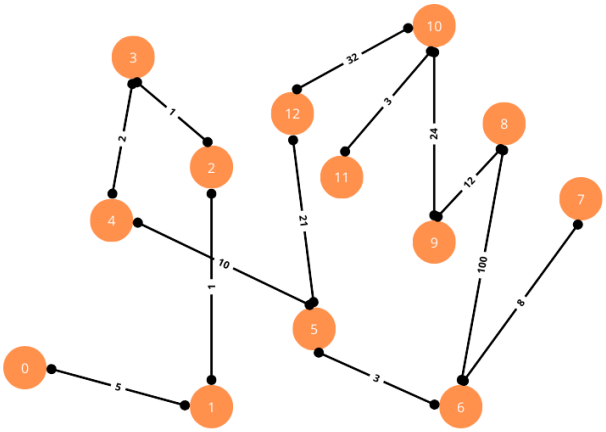
Es importante destacar que los **caminos mínimos** cumplen con la propiedad de **subestructura óptima**: Dado un dígrafo $G = (V, X)$ con una función de peso $l : X \to R$, sea $P : v_1 \dots v_k$ un camino mínimo de v_1 a v_k. Entonces, $\forall 1 \leq i \leq j \leq k$, $P_{v_i v_j}$ es un camino mínimo desde v_i a v_j.

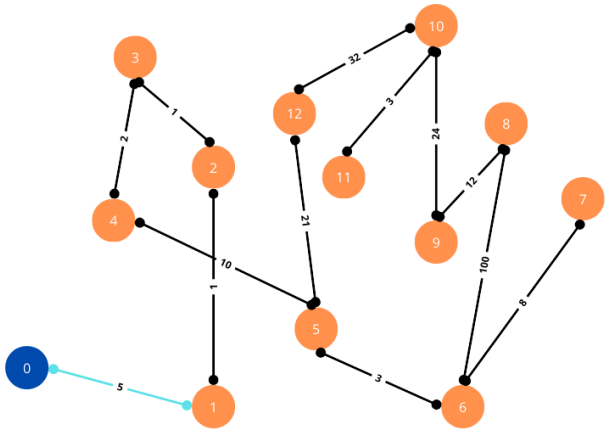
2 Algoritmos para resolver el problema de Single-Source-Multiple-Destination (SSMD)

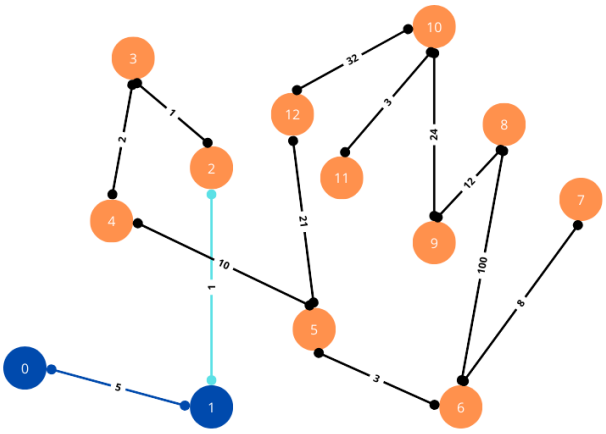
Ya mencionamos que el problema consiste en hallar **el camino mínimo desde un único origen hacia múltiples nodos en grafos pesados**, la aclaración no está de más, ya que para un grafo que tiene todas sus arista de peso 1, simplemente usamos BFS. El problema es el siguiente :
Dado $G = (V, X)$ un grafo y $l : X \to R$ una función que asigna a cada arco una cierta longitud y $v \in V$ un nodo del grafo, calcular los caminos mínimos de v al resto de los nodos.

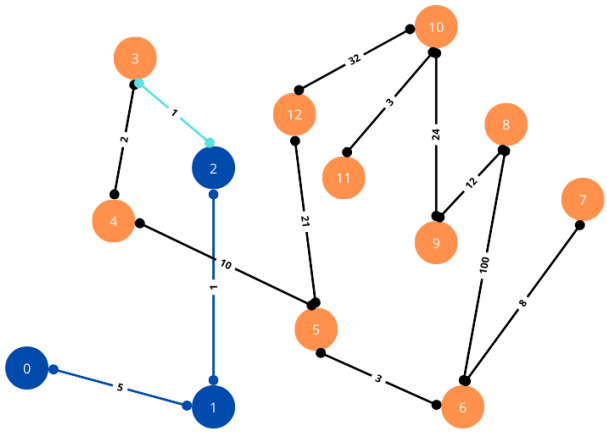
2.1 Algortimo de Dijkstra

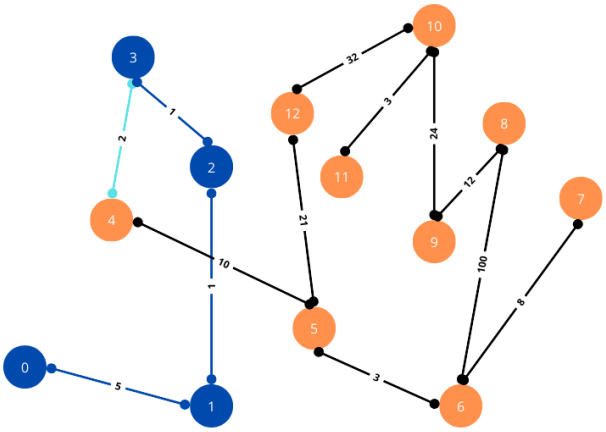
El algoritmo de Dijkstra nos permite encontrar los caminos mínimos desde un nodo a todos en grafos pesados con aristas **no negativas**. La idea del algoritmo es la siguiente, se mantiene un árbol enraizado en **src** con una *frontera* de vértices adyacentes a los del árbol. En cada paso, se agrega el vértice de la frontera más cercano a src. Es considerado un algortimos de tipo **greedy**.





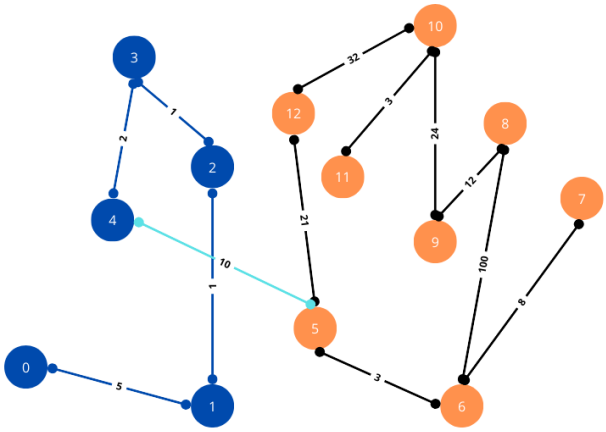






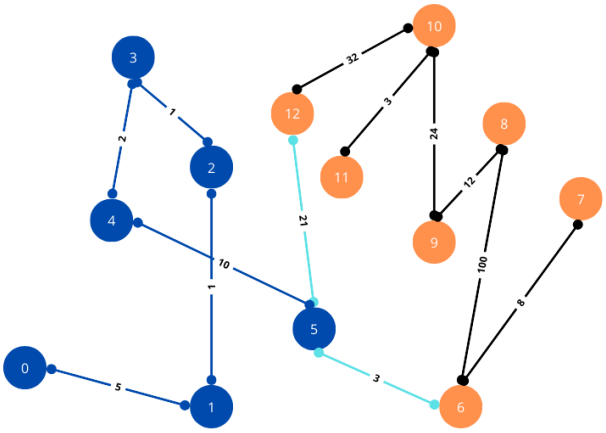
VISITADOS = { 0, 1, 2, 3}

distancias-mínimas= { 0, 5, 6, 7, 9,inf,inf, inf,inf,inf, inf,inf,inf. inf }



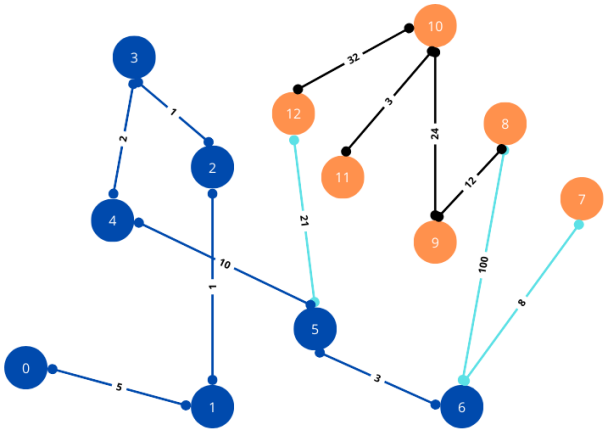
VISITADOS = { 0, 1, 2, 3, 4}

distancias-mínimas= { 0, 5, 6, 7, 9, 19,inf, inf,inf,inf, inf,inf,inf. inf }



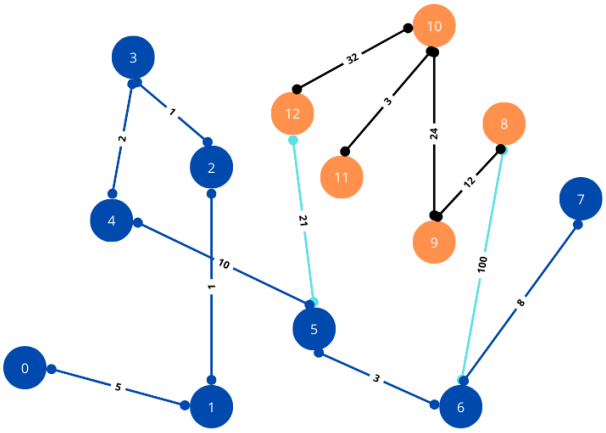
VISITADOS = { 0, 1, 2, 3, 4, 5}

distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , inf,inf,inf, inf,inf, 43}



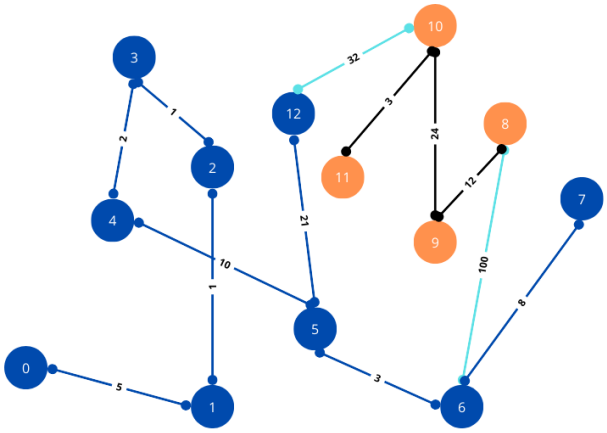
VISITADOS = { 0, 1, 2, 3, 4, 5, 6}

distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 122 ,inf, inf,inf, 43}



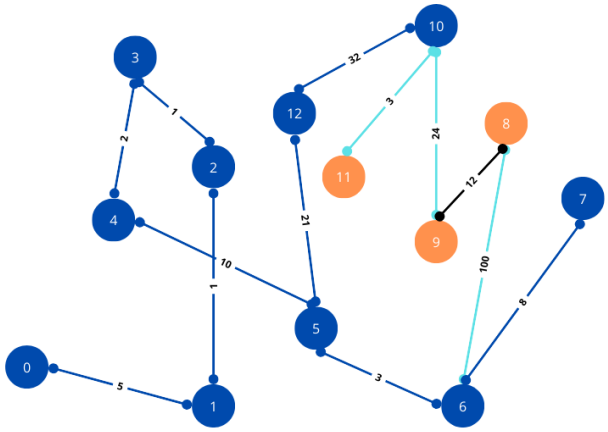
VISITADOS = { 0, 1, 2, 3, 4, 5, 6, 7}

distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 122 ,inf, inf,inf, 43}

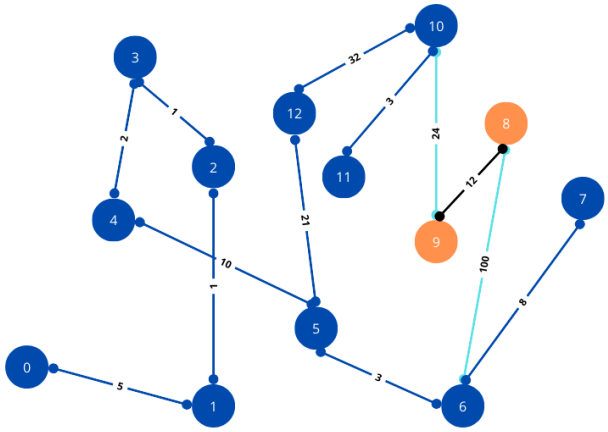


VISITADOS = { 0, 1, 2, 3, 4, 5, 6, 7, 12}

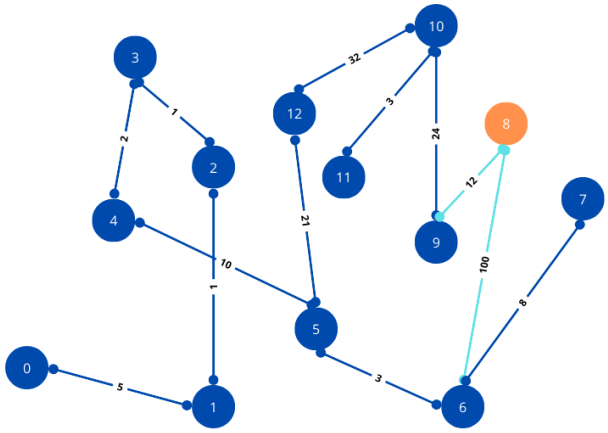
distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 122 ,inf, 75 ,inf, 43}



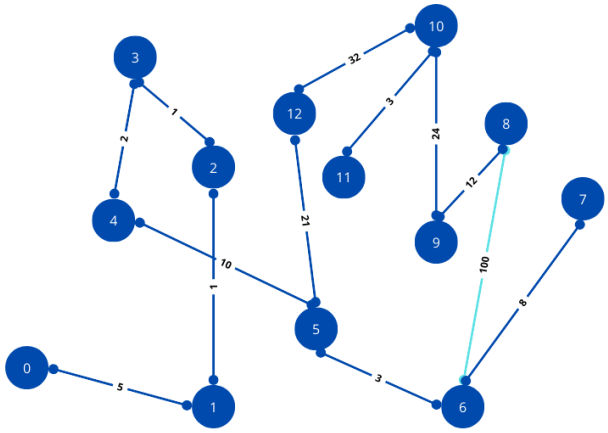
VISITADOS = { 0, 1, 2, 3, 4, 5, 6, 7, 12, 10}
distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 122 , 99 , 75 ,78 , 43}



VISITADOS = { 0, 1, 2, 3, 4, 5, 6, 7, 12, 10, 11}
distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 122 , 99 , 75 ,78 , 43}



VISITADOS = { 0, 1, 2, 3, 4, 5, 6, 7, 12, 10, 11, 9}
distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 111 , 99 , 75 ,78 , 43}



VISITADOS = { 0, 1, 2, 3, 4, 5, 6, 7, 12, 10, 11, 9, 8}
distancias-mínimas= { 0, 5, 6, 7, 9, 19 , 22 , 30 , 111 , 99 , 75 ,78 , 43}

```cpp
void dijkstra(int src)
{
    // Cola de prioridad para almacenar los v rtices que est n siendo procesados
    priority_queue<pair<int, int>, vector<pair<int,int>>, greater<pair<int, int>>> pq;

    // Distancias inicializadas todas en INF
    vector<int> dist(V, INF);

    // Pusheamos a la cola el src (nodo origen) y lo inicializamos con distancia 0
    pq.push(make_pair(0, src));
    dist[src] = 0;

    // Procesamos
    while (!pq.empty())
    {
        // Tomamos el de menor distancia
        int u = pq.top().second;
        pq.pop();
        // Para cada vecino
        for (auto &vecino : adj[u])
        {
            int v = vecino.first;
            int peso = vecino.second;
            // Si existe un camino m s corto hacia v ==>
            if (dist[v] > dist[u] + peso)
            {
                // Actualizamos la distancia, y la pusheamos al heap
                dist[v] = dist[u] + peso;
                pq.push(make_pair(dist[v], v));
            }
        }
    }
}
```

2.1.1 Análisis de complejidad

- **Espacial** : La cola de prioridad almacena V-vértices por tanto será $O(V)$. Lo mismo para el vector de distancias.
- **Temporal** : Crear la cola de prioridad cuesta O(V). Cualquier operación sobre ella O(log V). Si prestamos atención, de la pq vamos a sacar V-vértices y por cada uno de ellos haremos una operación de complejidad logarítmica. Es decir que tendremos O(V log V) en principio. Luego como recorremos las adyacencias de cada uno de ellos, las aristas del grafo y se hace una operación de heap, tendremos una complejidad O (E log V). El tiempo de ejecución del algoritmo de Dijkstra es $O((|E| + |V|) \log |V|)$. Si el número de aristas $|E| \in \Theta(|V|^2)$ (en grafos densos), entonces la complejidad temporal se convierte en:

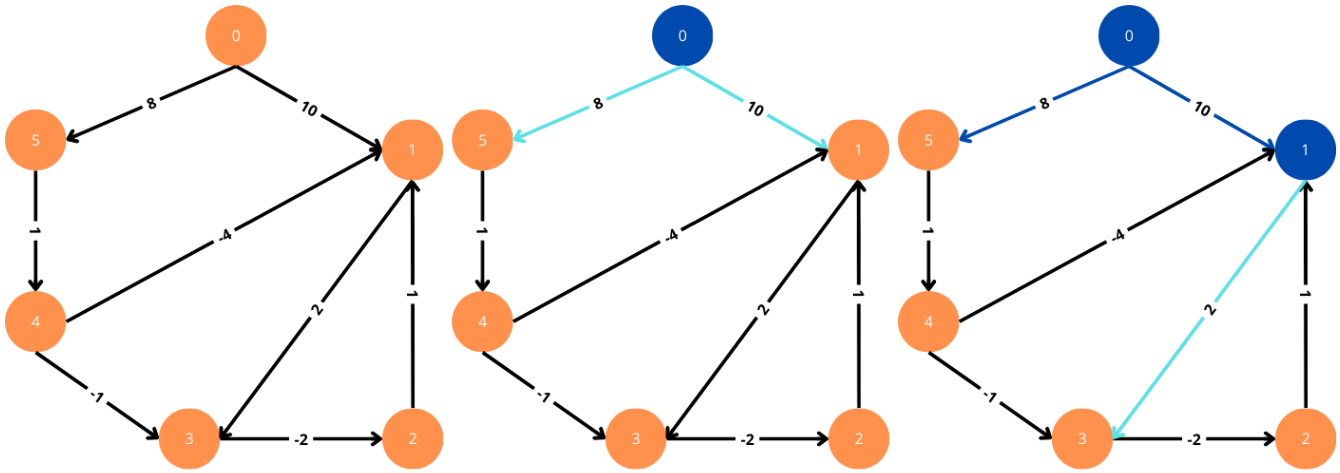
  $$O((|V|^2 + |V|) \log |V|) = O(|V|^2 \log |V|)$$

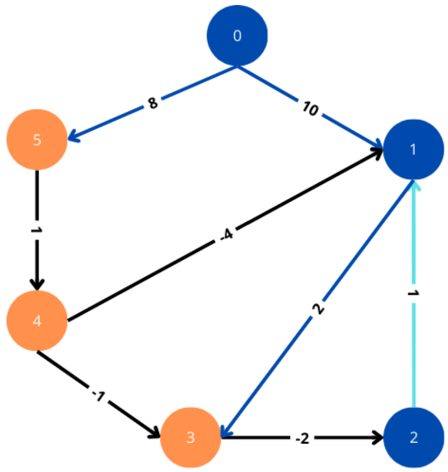
  Esto se puede mejorar si se utiliza otra estructura como el **fibonacci heap**, que para grafos ralos corre en $O(|E| + |V| \log |V|)$. Aunque si se trata de grafos densos cuando el número de aristas $|E| \in \Theta(|V|^2)$ entonces la complejidad es :

  $$O(|V|^2 + |V| \log |V|) = O(|V|^2)$$

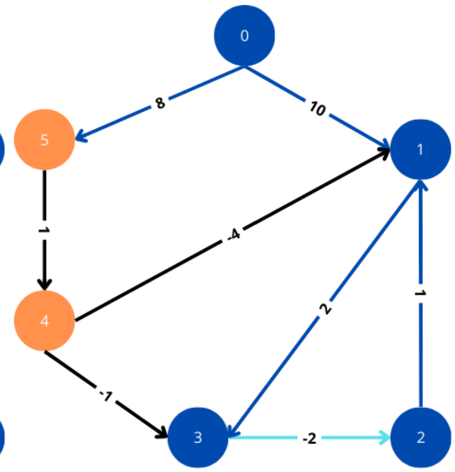
2.2 Algortimo de Bellman-Ford

A diferencia del algoritmo de Dijkstra, el algoritmo de Bellman-Ford puede ser utilizado para **digrafos con pesos negativos (pero sin ciclos negativos alcanzables desde el origen)** en caso de tener un ciclo de esta índole, el algortimo es capaz de detectarlo. La razón por la cuál Dijkstra no es útil para grafos con aristas de peso negativo es que **no hace una revisita** a los nodos que ya fueron visitados. En caso de que exista una ruta, más lejana con pesos negativos, Dijkstra no puede contemplarla. El principio de este algoritmo es la **relaxation of edges**, es decir, se verifica si se puede mejorar la distancia a un nodo mediante el uso de una arista (u,v) con peso w. Si $dist[v] > dist[u] + w$, significa que se encontró un camino más corto hacia v pasando por u. En este caso, actualizamos dist[v] = dist[u] + w. La *relajación* ocurre **V-1** veces, ya que en un camino entre dos nodos, el número máximo de aristas sin formar ciclos es (V - 1). Por tanto, con V - 1 relajaciones garantizamos que cualquier camino posible en el grafo será evaluado. Cada vez que realizamos una relajación completa de todas las aristas, acercamos cada nodo a su distancia mínima desde la fuente, si es posible. Donde ocurre la **detección de ciclos negativos** ? Justamente la condición de corte, luego de las V-1 iteraciones, si encontramos que **todavía se puede reducir alguna distancia** es porque efectivamente existe un ciclo de peso negativo. Entonces retorna -1. Para poder entender mejor cómo funciona la *relajación* veamos este ejemplo concreto :

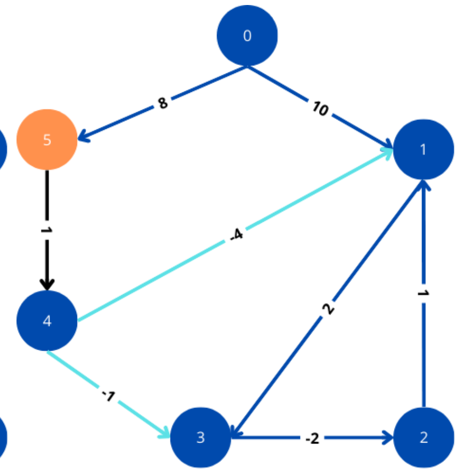




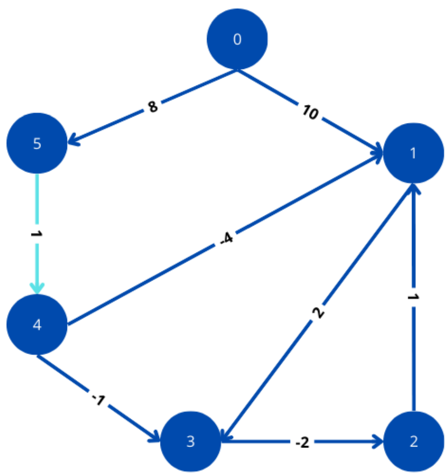
Distancias-mínimas= { 0, 10, inf, 12, inf, 8}
iteración : 1



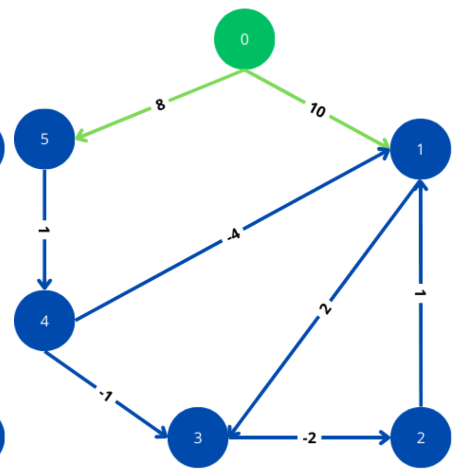
Distancias-mínimas= { 0, 10, 10, 12, inf, 8}
iteración : 1



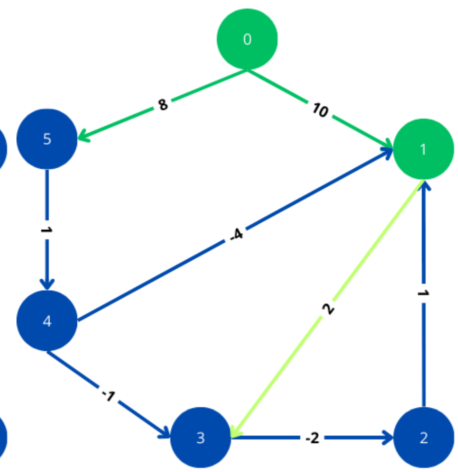
Distancias-mínimas= { 0, 10, 10, 12, inf, 8}
iteración : 1



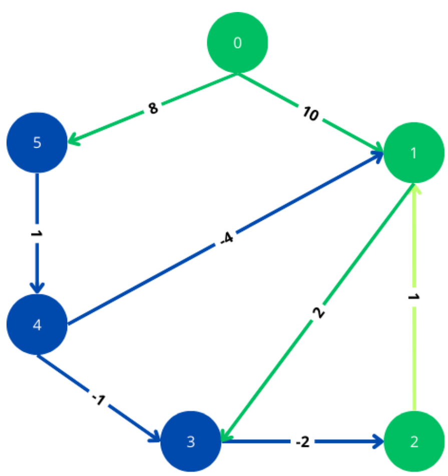
Distancias-mínimas= { 0, 10, 10, 12, 9, 8}
iteración : 1



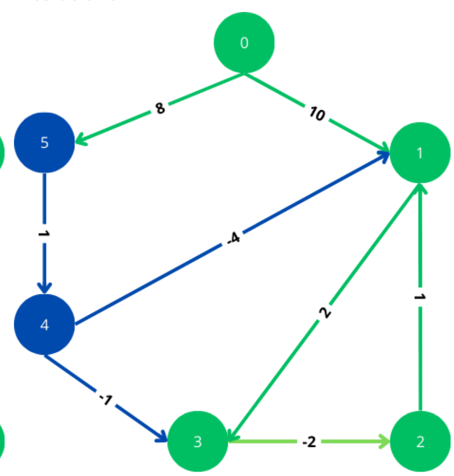
Distancias-mínimas= { 0, 10, 10, 12, 9, 8}
iteración : 2



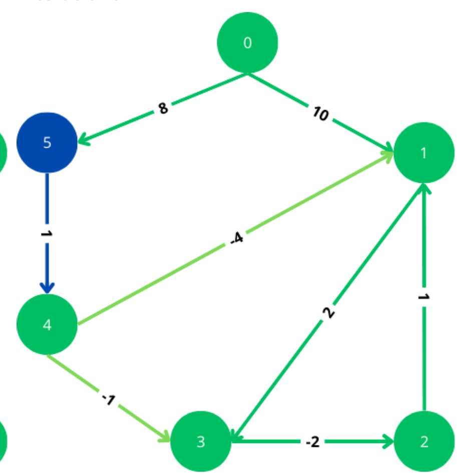
Distancias-mínimas= { 0, 10, 10, 12, 9, 8}
iteración : 2



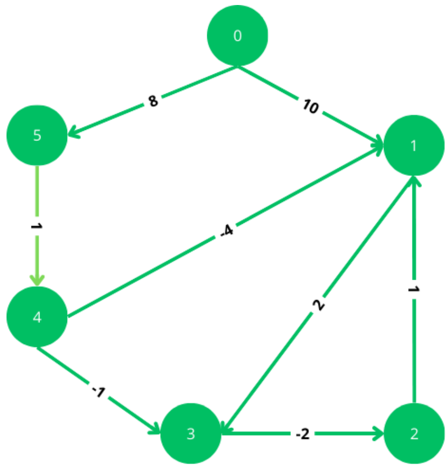
Distancias-mínimas= { 0, 10, 10, 12, 9, 8}
iteración : 2



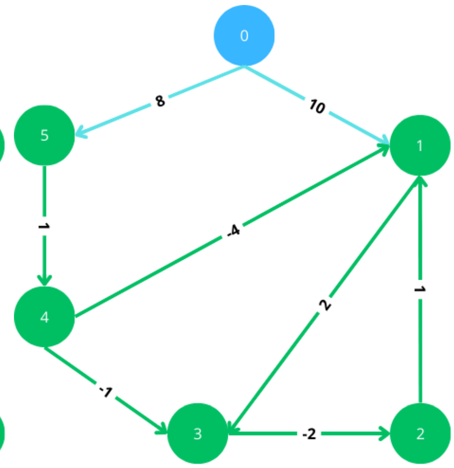
Distancias-mínimas= { 0, 10, 10, 12, 9, 8}
iteración : 2



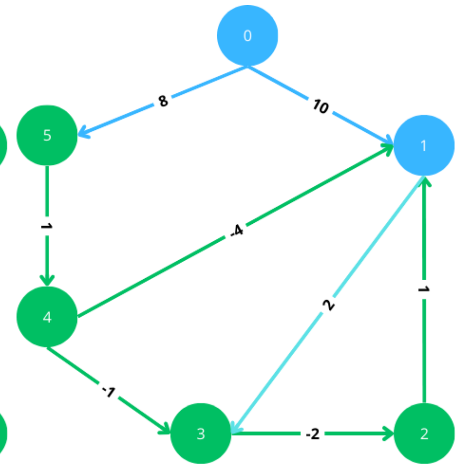
Distancias-mínimas= { 0, 5, 10, 8, 9, 8}
iteración : 2



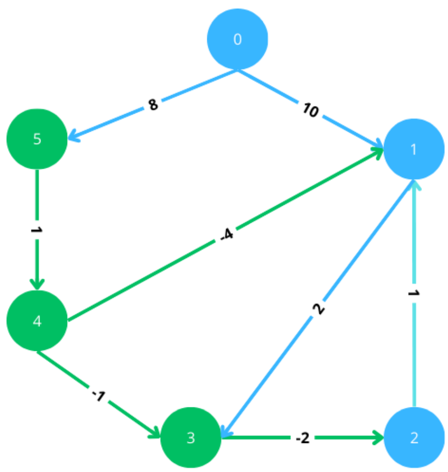
Distancias-mínimas= { 0, 5, 10, 8, 9, 8}
iteración : 2



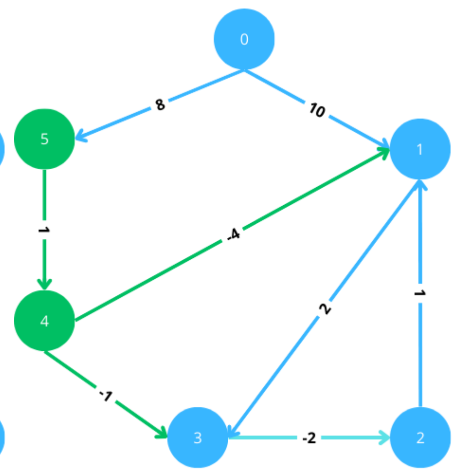
Distancias-mínimas= { 0, 5, 10, 8, 9, 8}
iteración : 3



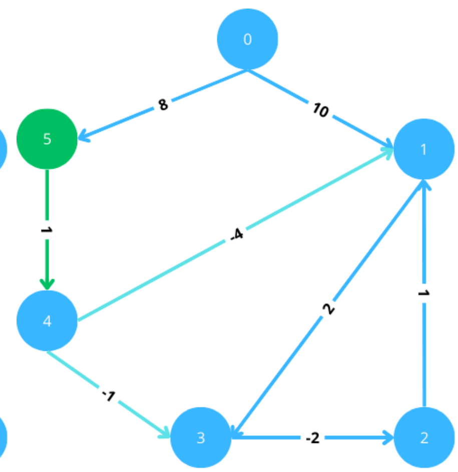
Distancias-mínimas= { 0, 5, 10, 7, 9, 8}
iteración : 3



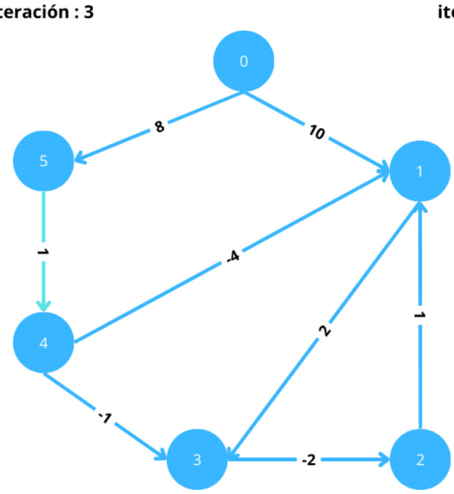
Distancias-mínimas= { 0, 5, 10, 7, 9, 8}
iteración : 3



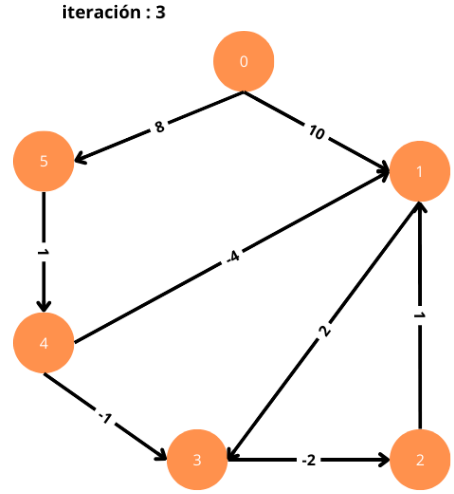
Distancias-mínimas= { 0, 5, 5, 7, 9, 8}
iteración : 3



Distancias-mínimas= { 0, 5, 5, 7, 9, 8}
iteración : 3



Distancias-mínimas= { 0, 5, 5, 7, 9, 8}
iteración : 3



Distancias-mínimas= { 0, 5, 5, 7, 9, 8}
iteración : 6

Así ..... hasta la iteración V ....

```cpp
vector<int> bellmanFord(int V, vector<vector<int>>& edges, int src) {

    // Se inicaliza el vector de distancias desde src en INF y a src con 0
    vector<int> dist(V, 1e8);
    dist[src] = 0;

    // Realizamos V iteraciones
    for (int i = 0; i < V; i++) {
        for (vector<int> edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int wt = edge[2];
            if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

                // Si se puede mejorar una distancia en la iteraci n V
                // es porque existe ciclo negativo
                if(i == V - 1)
                    return {-1};

                // Se actualiza la distancia m s corta al nodo src
                dist[v] = dist[u] + wt;
            }
        }
    }
    return dist;
}
```

2.2.1 Análisis de complejidad

- **Espacial** : Sólo estamos usando el arreglo de distancias para almacenar los valores. Por tanto la complejidad espacial es $O(V)$.
- **Temporal** : Iteramos V-veces sobre las aristas del grafo. Esto nos da una complejidad temporal $O(|V| \times |E|)$. Para ajustar esta complejidad se puede chequear si luego de varias iteraciones los valores de distancia se mantienen constantes. Recordar que si el si se trata de grafos densos cuando el número de aristas $|E| \in \Theta(|V|^2)$ entonces la complejidad se nos va de las manos a $O(|V| \times |V|^2)$ por tanto nos queda complejidad temporal **cúbica** : $O(|V|^3)$.

2.3 DAG : Directed Acyclic Graph

Para el caso de dígrafos acíclicos, el camino mínimo de uno a muchos puede resolverse utilizando un algoritmo más eficiente que los anteriores. Este se basa en la siguiente fórmula para la distancia entre dos nodos:

$$\delta(u, v) = \begin{cases} 0 & \text{si } u = v \\ \min\{\delta(u, z) + w(z \to v) \mid z \in N^-(v)\} & \text{en caso contrario} \end{cases}$$

Esto es intuitivo: la distancia mínima entre un par de nodos u, v es el peso de algún camino mínimo $P = u \cdots v$. El anteúltimo vértice del camino z está en el vecindario de entrada de v (ya que tiene una arista que apunta hacia él), mientras que el subcamino P_{uz} también es mínimo.

Esta relación no se puede utilizar para computar las distancias en un grafo con un ciclo $C = v \cdots v$, ya que la llamada $\delta(u, v)$ llevaría a un bucle de recursión infinita (porque existe un camino de v a v a través de los vecindarios de entrada).

Versión Top-Down

La fórmula se puede implementar como un algoritmo de programación dinámica top-down. Asumiendo que el arreglo de distancias (que funciona como estructura de memoización) D se inicializa con $\perp$ en todas las posiciones, el código es el siguiente:

Algorithm 1 DAG-Top-Down(G, s, v)

```
if s = v then
    return 0
end if
if D[v] =⊥ then
    D[v] ← min{SP-DAG-Top-Down(G, s, z) + w(z, v) | z ∈ N⁻(v)}
end if
return D[v] =0
```

Versión Bottom-Up

En esta versión, las distancias se calculan en orden topológico, asegurando que cada distancia $D[z]$ se calcule antes que $D[v]$ para cualquier $z \in N^-(v)$.

Algorithm 2 DAG-Bottom-Up(G, s)

```
0: Inicializar arreglo de distancias D
0: Calcular un orden topológico de G
0: for cada v ∈ V, en orden topológico do
0:   if s = v then
0:     D[v] ← 0
0:   else
0:     if N⁻(v) ≠ ∅ then
0:       D[v] ← min{D[z] + w(z → v) | z ∈ N⁻(v)}
0:     else
0:       D[v] ← ∞
0:     end if
0:   end if
0: end for
0: return D =0
```

3 Algortimos para resolver el problema de All-Pairs-Shortest-Paths (Matriciales)

Este problema consiste en encontrar el **camino más corto entre todos los pares de vértices de un grafo pesado** puede ser dirigido o no, con o sin aristas de pesos negativos.

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

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

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

2. $D \in R^{n\times n}$, donde los elementos d_{ij} de D se definen como:

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

D es llamada la matriz de distancias de G.

3.1 Algortimo de Floyd-Warshall

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

- Si $D^0 = L$ y calculamos D^1 como:

$$d^1_{ij} = \min(d^0_{ij}, d^0_{i1} + d^0_{1j})$$

  donde d^1_{ij} es la longitud de un camino mínimo de v_i a v_j con vértice intermedio v_1 o directo.
  Es decir, la inicialización de las distancias está basada en la Matriz L que vimos que pone en la diagonal un cero (distancia del vértice a sí mismo), si no hay un camino **directo** pone ∞ y si existe una arista de v_i a v_j, entonces $D^0_{ij} = L_{ij}$.
  La Matriz D^1 representa las distancias más cortas entre los vértices v_i a v_j **que considera sólo como vértice intermedio a** v_1 .Es decir, $(d^0_{i1} + d^0_{1j})$ es la distancia de ir de v_i a v_1 sumado a la distancia de ir de v_1 a v_j que será uno de los parámetros para comparar si el camino que existe pasando por este vértice intermedio mejora la distancia. El otro será el **default**, aquella distancia que ya **conocemos** (d^0_{ij}).
  **Conocemos está en negrita porque este algoritmo es de Programación Dinámica, y es importante contemplar si la información resguardada hasta cierta iteración mejora.

- Si calculamos D^k a partir de D^{k-1} como sigue:

$$d^k_{ij} = \min(d^{k-1}_{ij}, d^{k-1}_{ik} + d^{k-1}_{kj})$$

  donde d^k_{ij} es la longitud de un camino mínimo de v_i a v_j cuyos vértices intermedios están en $\{v_1, \ldots, v_k\}$.
  En este momento, en la *iteración k*, queremos comparar la **ruta anterior** (d^{k-1}_{ij}) con la **distancia de la ruta que se obtiene al pasar por el vértice intermedio** v_k .

- $D = D^n$.
  Finalmente, al llegar a $k = n$ obtendremos la matriz D^n con **las distancias mínimas entre todos los pares de vértices considerando a TODOS los posibles vértices intermedios**

Los elementos de la matriz de Floyd Warshall cumplen :

- **Propiedad reflexiva (diagonal):** Los elementos de la diagonal principal dist$[i][i]$ representan la distancia de un nodo a sí mismo.
  - Si no hay ciclos negativos que involucren el nodo i, entonces:

$$\text{dist}[i][i] = 0.$$

  - Si **existe un ciclo negativo en el grafo** que afecta el nodo i, entonces:

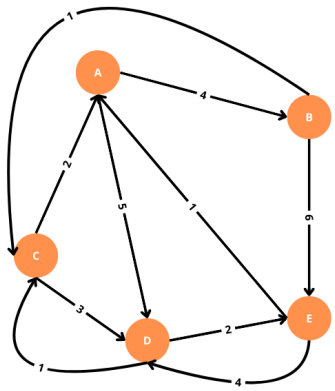
$$\text{dist}[i][i] < 0.$$

- **Desigualdad Triangular (propiedad del camino mínimo):** La matriz satisface la propiedad de camino más corto:

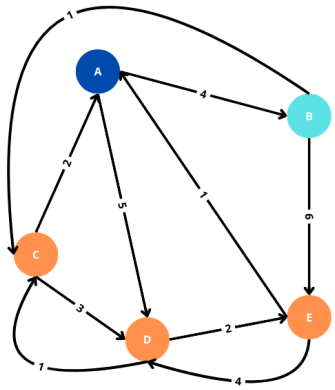
$$\text{dist}[i][j] \leq \text{dist}[i][k] + \text{dist}[k][j],$$

  para cualquier nodo intermedio k. Esto asegura que la distancia directa $i \to j$ no es mayor que cualquier ruta que pase por otros nodos intermedios.

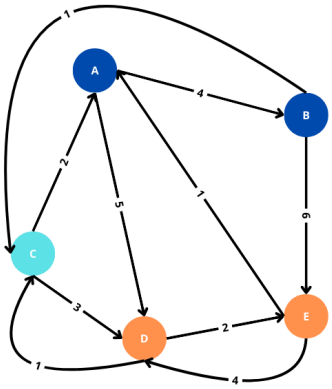
Para que funcione correctamente, **se debe asumir que NO existen ciclos negativos en el grafo**. Veamos el ejemplo :



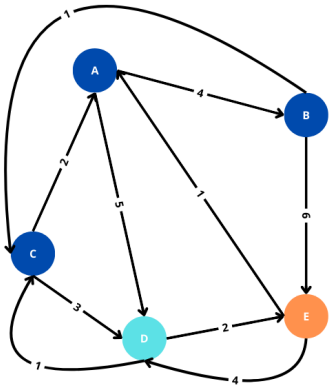
	A	B	C	D	E
A	0	4	INF	5	INF
B	INF	0	1	INF	6
C	2	INF	0	3	INF
D	INF	INF	1	0	2
E	1	INF	INF	4	0



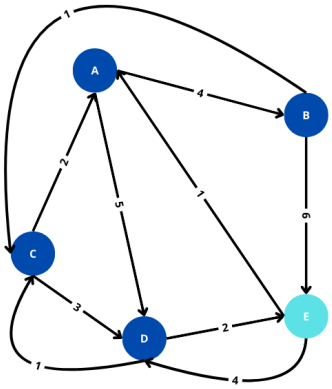
	A	B	C	D	E
A	0	4	5	5	10
B	INF	0	1	INF	6
C	2	6	0	3	12
D	INF	INF	1	0	2
E	1	5	6	4	0



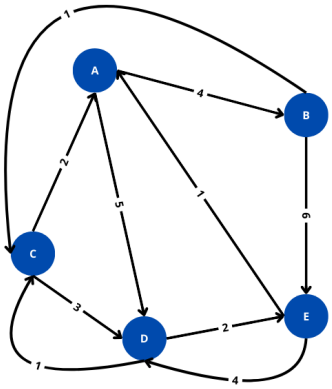
	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0



	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0



	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0



	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

```
void floydWarshall(int dist[][V])
{
    //   ndices
    int i, j, k;
    // Tomamamos cada v rtice como src
    for (k = 0; k < V; k++)
    {
        // Tomamos cada v rtice como dst
        for (i = 0; i < V; i++)
        {
            // Para cada v rtice
            for (j = 0; j < V; j++)
            {
            // Si el v rtice est  en el camino m s corto entre vi y vj
            if (dist[i][j] > (dist[i][k] + dist[k][j]) && (dist[k][j] != INF && dist[i][k] != INF))
                    // actualizamos el valor de  dist[i][j]
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }
}
```

3.1.1 Análisis de complejidad

- **Espacial** : dado que el algortimo hace uso de una matriz de distancias, la complejidad temporal es $O(|V|^2)$
- **Temporal** : el algortimo de Floyd-Warshall tiene 3 ciclos anidados de $|V|$ iteraciones cada uno. Dentro tenemos unn *if* que corre en tiempo constante. Dicho esto, la complejidad temporal es $O(|V|^3)$. Es mejor que correr *n-veces Bellman-Ford* ya que para un grafo conexo $|E| > |V| - 1 \implies |E| \in \Omega(|V|)$. Hay un detalle adicional sobre este algoritmo cuando **existen ciclos negativos** en el grafo. El asunto se ve **en la diagonal de la matriz**, en los caminos de un nodo a sí mismo. Si esta distancia es negativa, $\text{dist}[i][i] < 0$, estamos en presencia de un ciclo negativo. Esto se puede utilizar para **detectar ciclos negativos** o para **devolver que no existe una solución bien definida para el problema**.

3.2 Algoritmo de Dantzig

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

El cálculo de la matriz D_k a partir de D_{k-1} para $1 \leq i, j \leq k$ se hace de la siguiente manera:

- Para las distancias desde i hacia k:

$$d_k(i,k) = \min_{1 \leq j \leq k-1} \{d_{k-1}(i,j) + l(j \to k)\}$$

- Para las distancias desde k hacia i:

$$d_k(k,i) = \min_{1 \leq j \leq k-1} \{l(k \to j) + d_{k-1}(j,i)\}$$

- Para las distancias entre i y j (sin pasar por k o pasando por k):

$$d_k(i,j) = \min\left(d_{k-1}(i,j), d_k(i,k) + d_k(k,j)\right)$$

Finalmente, al completar todas las iteraciones, obtenemos la matriz de distancias mínimas en todo el grafo como $D = D_n$. Para detectar circuitos de longitud negativa, comprobamos si existe algún i tal que:

$$d_n(i,i) < 0$$

Si esta condición se cumple, el grafo contiene un circuito de longitud negativa.

- **Entrada:** $G = (V, X)$ un grafo dirigido con n vértices.
- **Salida:** D matriz de distancias de G.

Inicializar $D \leftarrow L$ (donde L es la matriz de distancias inicial).

Algorithm 3 Algoritmo de Dantzig

```
Para k = 2 hasta n hacer:
  Para i = 1 hasta k hacer:
    D[i][k] ← min_{1≤j≤k−1}{D[i][j] + D[j][k]}
    D[k][i] ← min_{1≤j≤k−1}{D[k][j] + D[j][i]}
  Fin para
  Calcular t ← min_{1≤j≤k−1}{D[k][i] + D[i][k]}
  Si t < 0, entonces:
    Retornar "Hay circuitos de longitud negativa"
  Fin si
  Para i = 1 hasta k − 1 hacer:
    Para j = 1 hasta k − 1 hacer:
      D[i][j] ← min(D[i][j], D[i][k] + D[k][j])
    Fin para
  Fin para
Fin para
Retornar D =0
```

3.2.1 Análisis de complejidad

- **Espacial** : hace uso de una matriz de distancias y por tanto la complejidad espacial es $O(|V|^2)$.
- **Temporal** : El algoritmo itera sobre k desde 2 hasta n, por lo que hay $n-1$ iteraciones principales.

  En cada iteración k, el algoritmo calcula nuevas distancias mínimas para $D[i][k]$ y $D[k][i]$ para i que va de 1 a k. Cada cálculo de $D[i][k]$ y $D[k][i]$ requiere encontrar el mínimo sobre $k-1$ elementos. Esto contribuye a una complejidad aproximada de $O(V^2)$ en cada iteración k. La verificación de ciclos negativos $t = \min_{i \leq j \leq k-1}\{D[k][i] + D[i][k]\}$ también tiene una complejidad de $O(V)$ en cada iteración k. Para cada par (i,j) en el subgrafo de los primeros $k-1$ vértices, el algoritmo recalcula $D[i][j]$ como el mínimo entre su valor actual y $D[i][k] + D[k][j]$. Esto implica una complejidad de $O(V^2)$ por cada iteración k. Como conclusión el algortimo de Dantzig tiene complejidad $O(|V|^3)$

3.3 Algoritmo de Johnson

El algortimo de Johnson resuleve también el problema APSP y es más eficiente que Floyd-Warshall en **grafos ralos** :

1. **Modificación del grafo:** El algoritmo empieza agregando un vértice ficticio q al grafo, con aristas dirigidas a todos los demás vértices con un peso de 0. Esto es para permitir que el algoritmo de Bellman-Ford calcule distancias mínimas correctamente en grafos con aristas de peso negativo.

2. **Aplicación de Bellman-Ford desde el vértice ficticio:** Se ejecuta el algoritmo de Bellman-Ford desde el nuevo vértice q. Este algoritmo calcula las distancias más cortas desde q a todos los demás vértices. Si el algoritmo de Bellman-Ford detecta un ciclo negativo, el algoritmo de Johnson se detiene, ya que el grafo contiene un ciclo negativo, lo que hace imposible encontrar caminos más cortos válidos. Las distancias resultantes de Bellman-Ford se almacenan en un arreglo $h[]$, donde $h[v]$ es la distancia más corta desde q hasta el vértice v.

3. **Reajuste de pesos:** Para eliminar los efectos de las aristas con pesos negativos y hacer que los pesos sean no negativos, se reajustan los pesos de todas las aristas del grafo original. Si $w(u, v)$ es el peso de la arista de u a v, el peso ajustado $w'(u, v)$ se calcula como:

   $$w'(u, v) = w(u, v) + h[u] - h[v]$$

   Este reajuste garantiza que las distancias mínimas calculadas a partir de cualquier vértice en el grafo original correspondan a las distancias correctas, sin importar si había pesos negativos inicialmente.

4. **Ejecutar Dijkstra desde cada vértice:** Ahora que todos los pesos se han reajustado para ser no negativos, se puede ejecutar el algoritmo de Dijkstra desde cada vértice del grafo con los pesos ajustados. Dijkstra es eficiente cuando los pesos de las aristas son no negativos, lo que permite calcular los caminos más cortos de manera eficiente en $O(|E| \log |V|)$. Este paso se repite para todos los vértices, lo que da una complejidad de $O(V^2 \log |V| + |E||V|)$ para todo el grafo.

5. **Restaurar las distancias originales:** Después de ejecutar Dijkstra, se deben restaurar las distancias originales usando el arreglo $h[]$. La distancia de un vértice u a v en el grafo original se calcula como:

   $$d(u, v) = d'(u, v) + h[v] - h[u]$$

   donde $d'(u, v)$ es la distancia calculada en el grafo con los pesos ajustados.

```cpp
void JohnsonAlgorithm(const vector<vector<int>>& graph)
{
    // Cantidad de V rtices
    int V = graph.size();
    vector<vector<int>> edges;

    // Capturamos las aristas del grafo
    for (int i = 0; i < V; ++i)
    {
        for (int j = 0; j < V; ++j)
        {
            if (graph[i][j] != 0)
            {
                edges.push_back({i, j, graph[i][j]});
            }
        }
    }

    // Almacenamos las distancias en el vector altered_weights
    vector<int> altered_weights = BellmanFord_Algorithm(edges, V);
    vector<vector<int>> altered_graph(V, vector<int>(V, 0));

    // Repesamos las aristas
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (graph[i][j] != 0) {
                altered_graph[i][j] = graph[i][j] + altered_weights[i] - altered_weights[j];
            }
        }
    }

    // Corremos Dijkstra desde cada v rtice
    for (int source = 0; source < V; ++source)
    {
        Dijkstra_Algorithm(graph, altered_graph, source);
    }
}
```

3.3.1 Análisis de complejidad

El algoritmo de Johnson tiene una complejidad combinada que depende principalmente de dos pasos clave: la ejecución de Bellman-Ford y la ejecución de Dijkstra desde cada vértice. Aquí está el análisis de la complejidad:

- **Aplicación de Bellman-Ford:** El algoritmo de Bellman-Ford se ejecuta una sola vez desde el vértice ficticio q. Este algoritmo tiene una complejidad de $O(|V| \times |E|)$, donde $|V|$ es el número de vértices y $|E|$ es el número de aristas del grafo. El número de iteraciones del algoritmo de Bellman-Ford es $O(|V|)$, y en cada iteración, se revisan todas las $|E|$ aristas.

- **Reajuste de pesos:** El paso de reajustar los pesos de las aristas del grafo tiene una complejidad de $O(|E|)$, ya que se debe ajustar el peso de cada arista del grafo original utilizando la información de los valores almacenados en el arreglo $h[]$. Esto es lineal con respecto al número de aristas.

- **Ejecutar Dijkstra desde cada vértice:** Después de reajustar los pesos, el algoritmo de Dijkstra se ejecuta desde cada vértice. La complejidad de Dijkstra es $O(|E| \log |V|)$ para cada ejecución, dado que se usa una cola de prioridad (por ejemplo, un montículo). Como Dijkstra se ejecuta desde cada uno de los $|V|$ vértices, el costo total para este paso es $O(|V| \cdot |E| \log |V|)$.

- **Restaurar las distancias originales:** El paso de restaurar las distancias originales, que implica actualizar las distancias calculadas en el grafo original utilizando el arreglo $h[]$, tiene una complejidad de $O(|E|)$, ya que se debe recorrer cada arista y ajustar las distancias según la fórmula proporcionada.

Complejidad total:

Sumando todas estas complejidades, tenemos: $O(|V||E| + |V| \cdot |E| \log |V|)$ Por lo tanto, la complejidad total del algoritmo de Johnson es: $O(|V||E| + |V| \cdot |E| \log |V|)$

Conclusión: La complejidad total es dominada por el término $O(|V| \cdot |E| \log |V|)$, por lo que este es el costo más significativo del algoritmo. En grafos densos, donde $|E|$ puede ser cercano a V^2, el algoritmo de Johnson puede ser eficiente, ya que tiene una complejidad mejor que la de los algoritmos de Floyd-Warshall, que es $O(V^3)$.