uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 6 Grafos

Profesores EDA

Objetivos

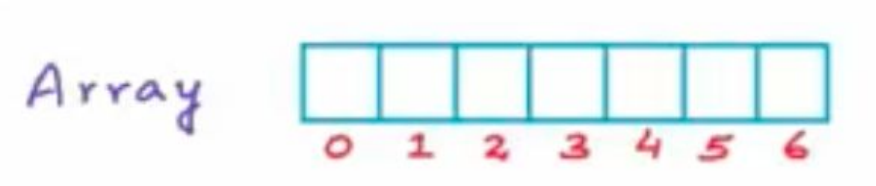
1. Conocer qué es un grafo y sus principales propiedades.
2. Implementar el tipo abstracto grafo usando distintas representaciones.
3. Comprender la complejidad espacial y temporal de cada representación.
4. Comprender e implementar algoritmos para visitar los vértices de un grafo.
5. Resolver problemas aplicando grafos.

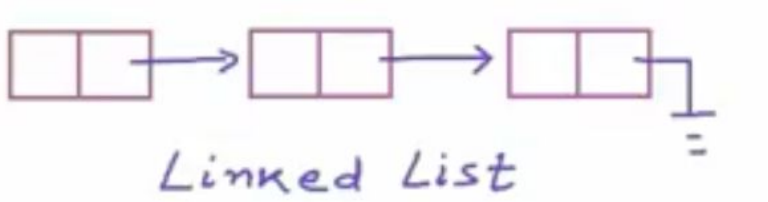
Índice

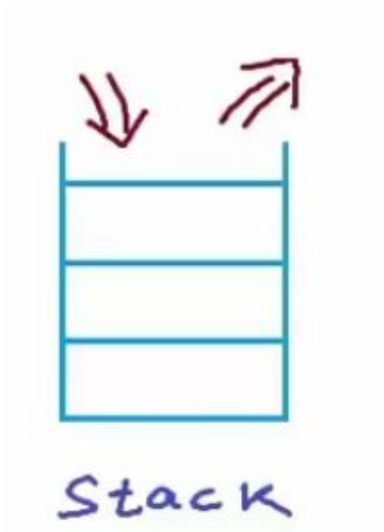
- **Introducción**
- Conceptos sobre grafos
- TAD Grafo
- Implementaciones:
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

Introducción

Estructuras Lineales

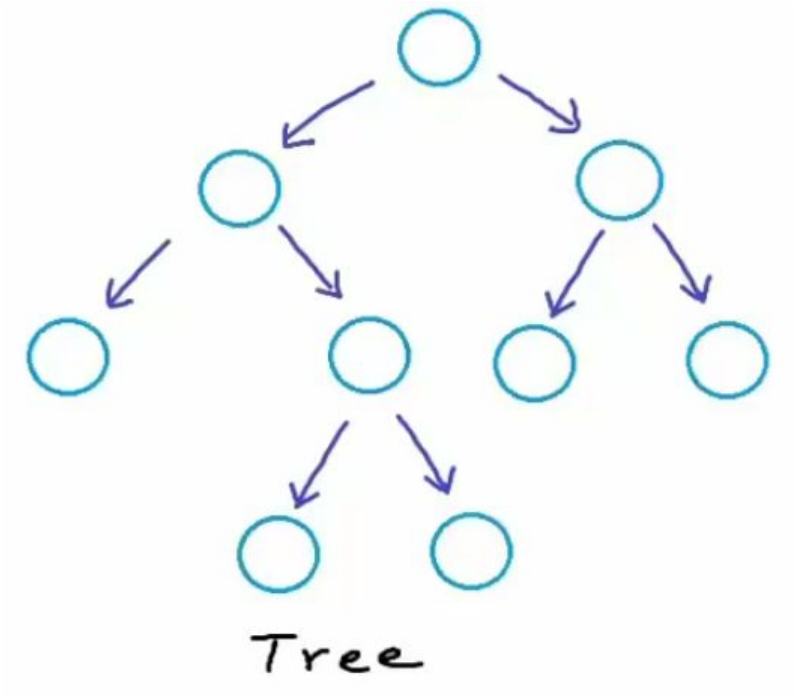






Introducción

Estructuras jerárquicas (no son lineales)

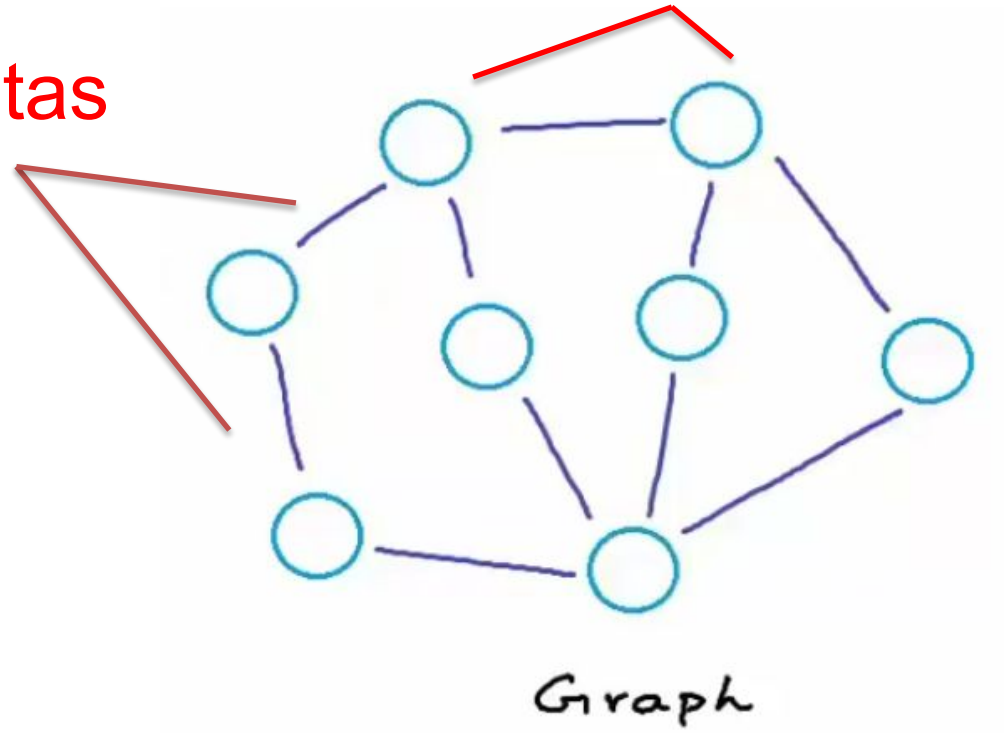


Introducción

Estructuras no lineales: grafos

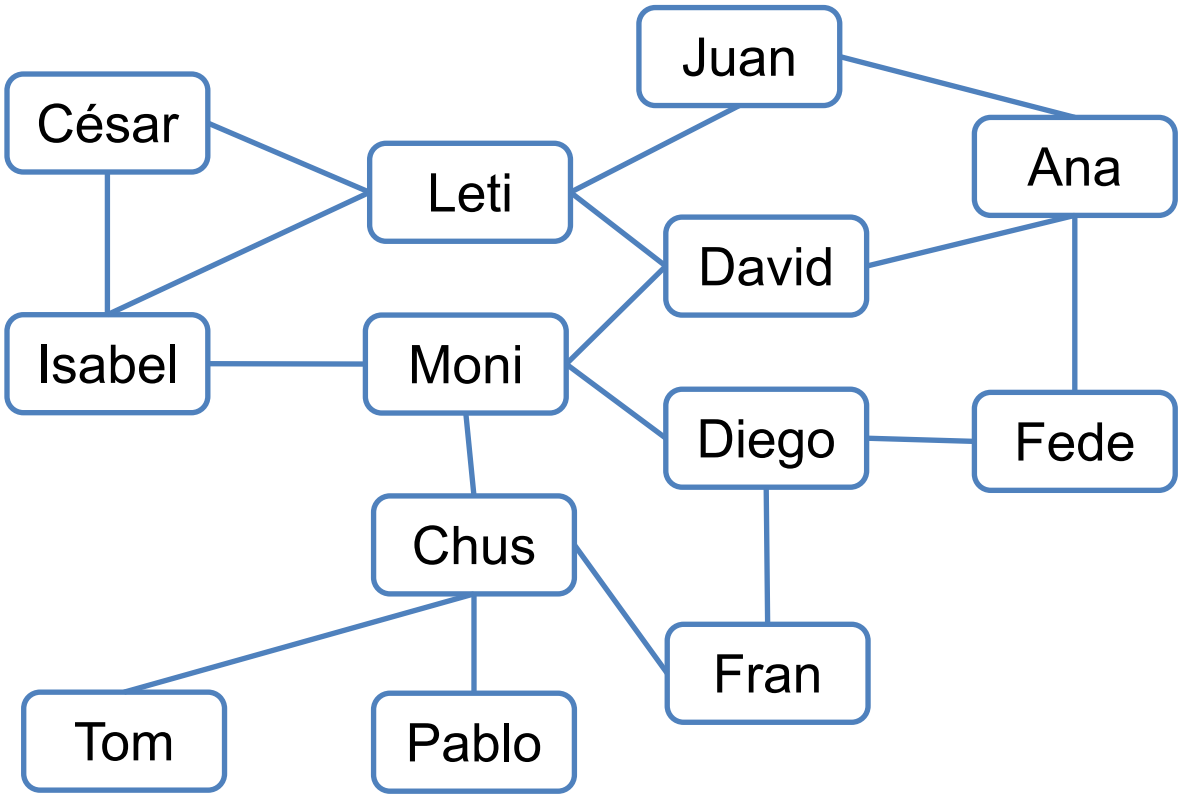
Nodos o vértices

Aristas



Permiten representar cualquier tipo de conexión

Introducción



Una red social se puede representar como un grafo no dirigido y no ponderado. Los usuarios son los vértices y sus relaciones de amistad las aristas.

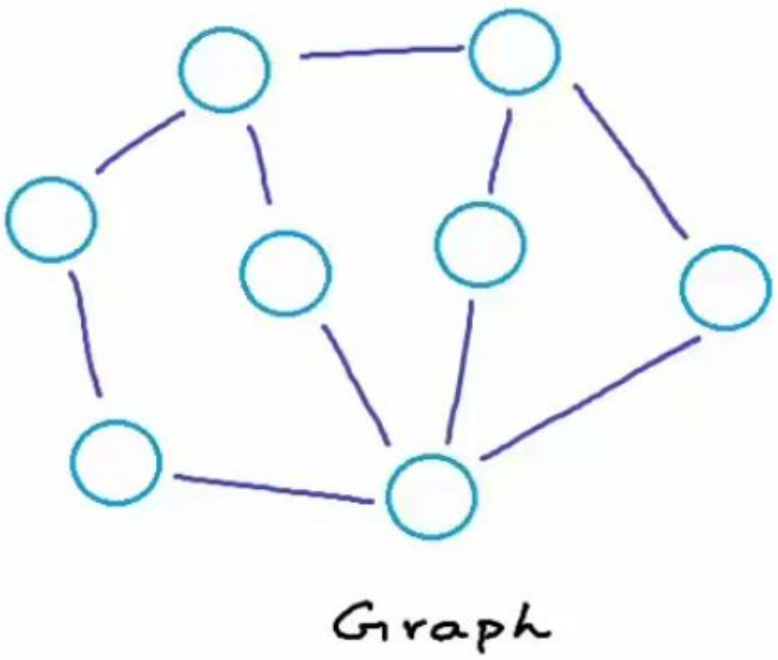
Índice

- Introducción
- **Conceptos sobre grafos**
- TAD Grafo
- Implementaciones:
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

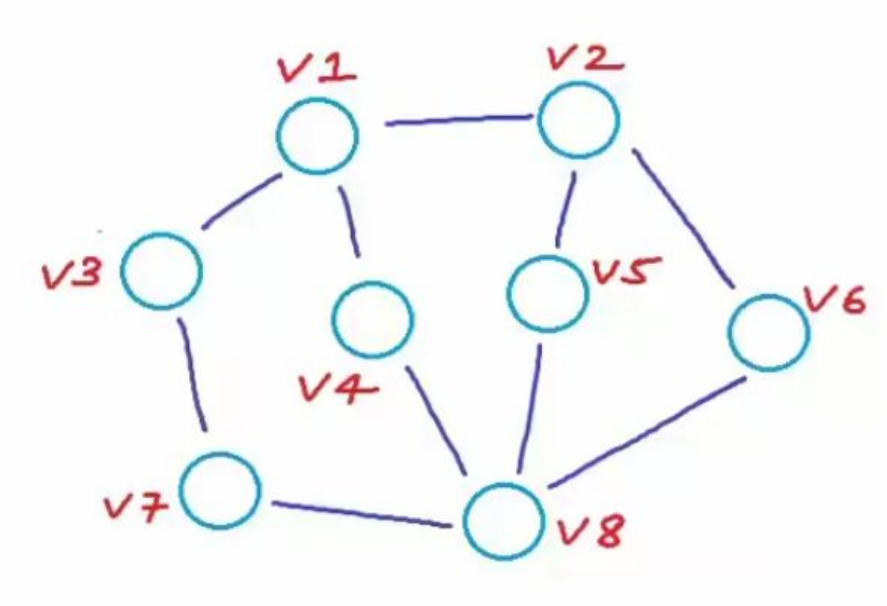
Conceptos sobre grafos

Grafo: G=(V,A)

Un grafo G es un par (V,A), donde V es un conjunto de vértices (nodos) y A un conjunto de aristas



Conceptos sobre grafos



Grafo: G=(V,A)

V = { v1, v2, v3, v4, v5, v6, v7, v8}

¿Cómo puedo representar una arista?

Conceptos sobre grafos

Tipos de Aristas

(u,v)
(u,v)!=(v,u) if u!=v

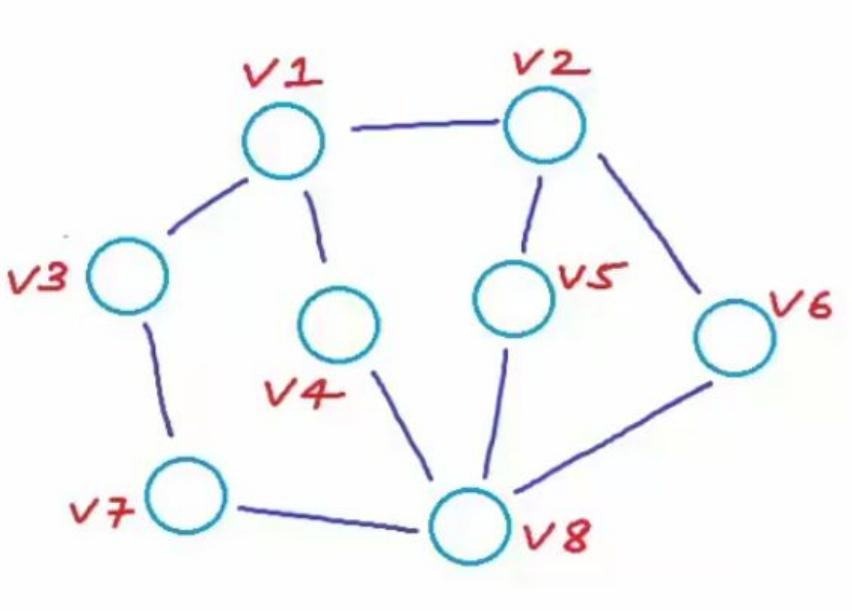
{u,v} ,
{u,v} = {v,u}

Conceptos sobre grafos

Grafo: G=(V,A)

V = { v1, v2, v3, v4, v5, v6, v7, v8}

A= { {v1, v2}, {v1, v3}, {v1, v4}, {v2,v5}, {v2,v6}, {v3,v7}, {v4,v8}, {v5,v8}, {v6,v8}, {v7,v8}}



IVI =número de vértices

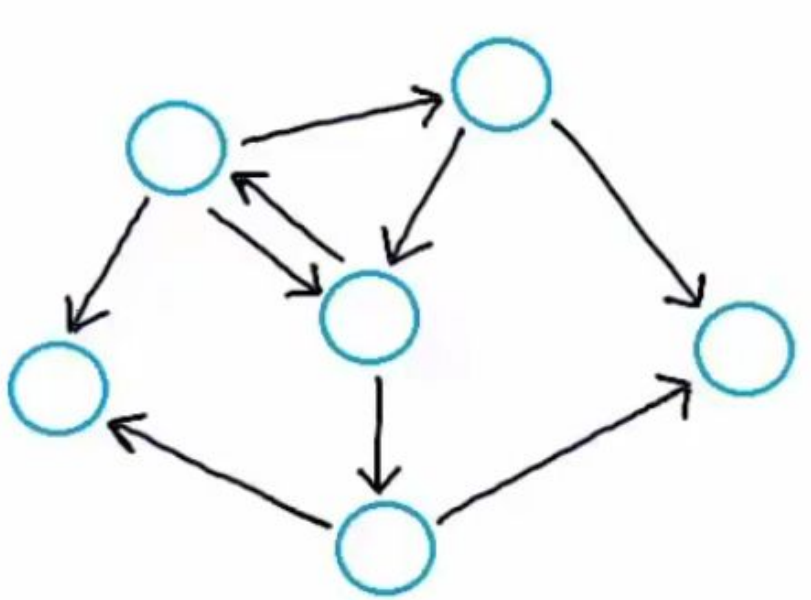
IAI =número de aristas

IVI = 8

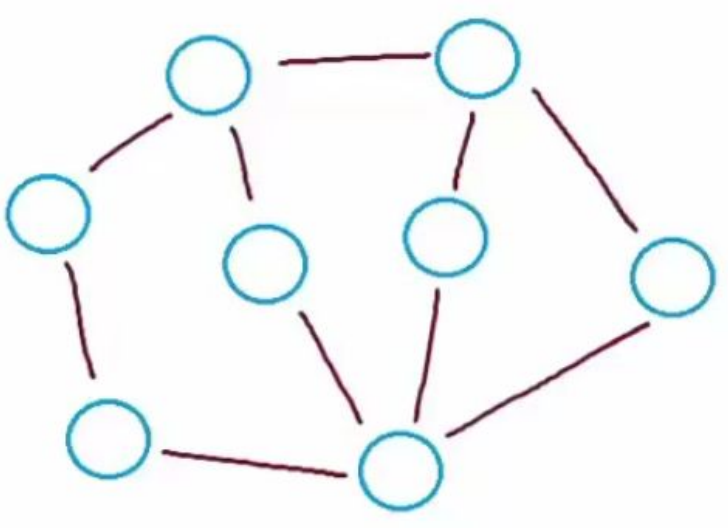
IAI=10

Conceptos sobre grafos

Grafo dirigidos

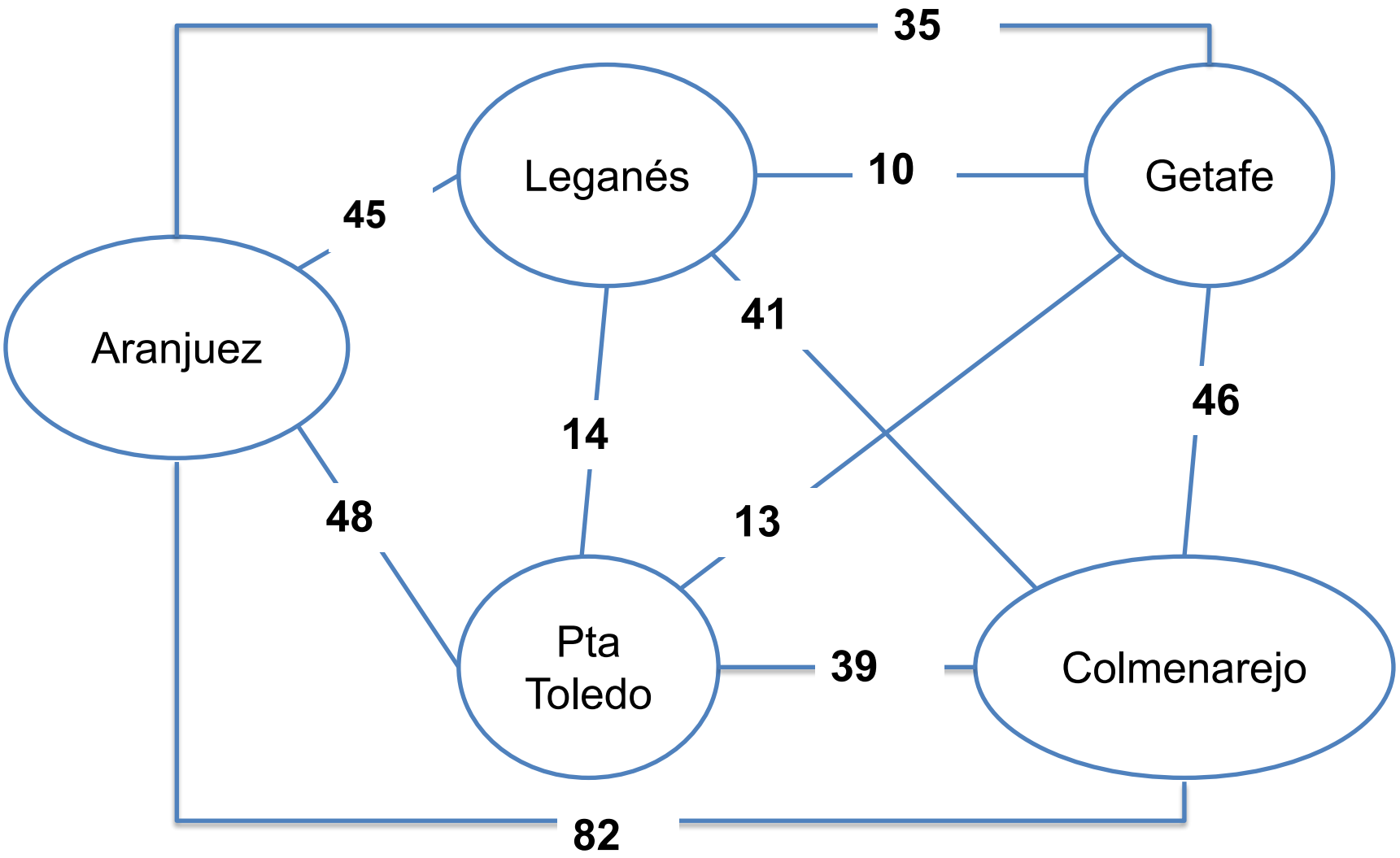


Grafo no dirigido



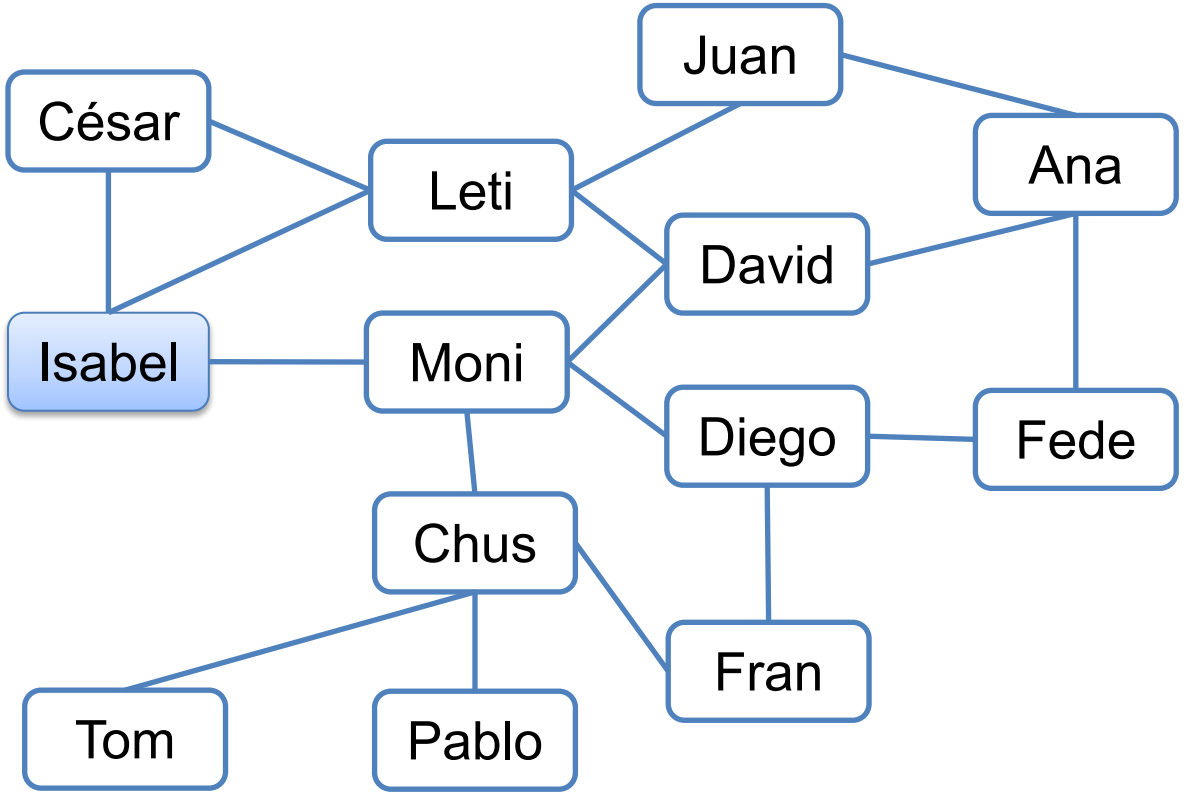
Conceptos sobre grafos

Grafo ponderado (weighted)



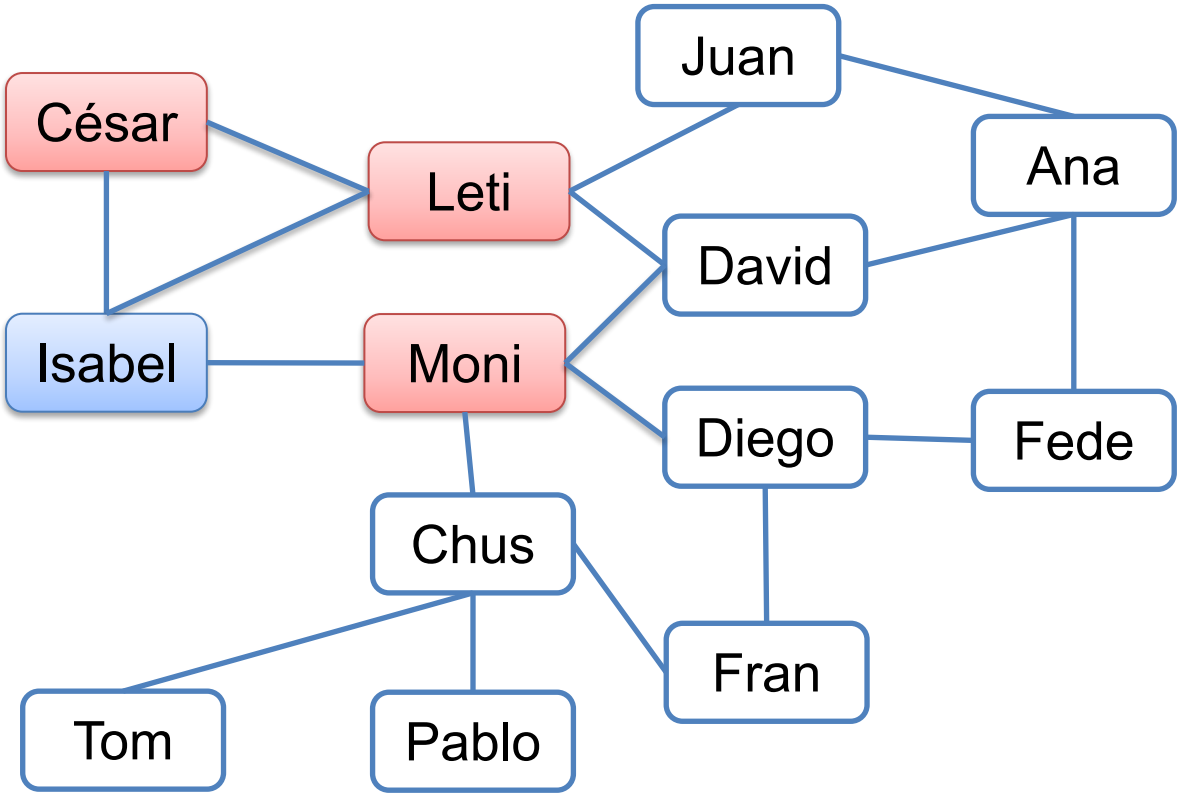
Campus de la UC3M (distancias en KM)

Conceptos sobre grafos

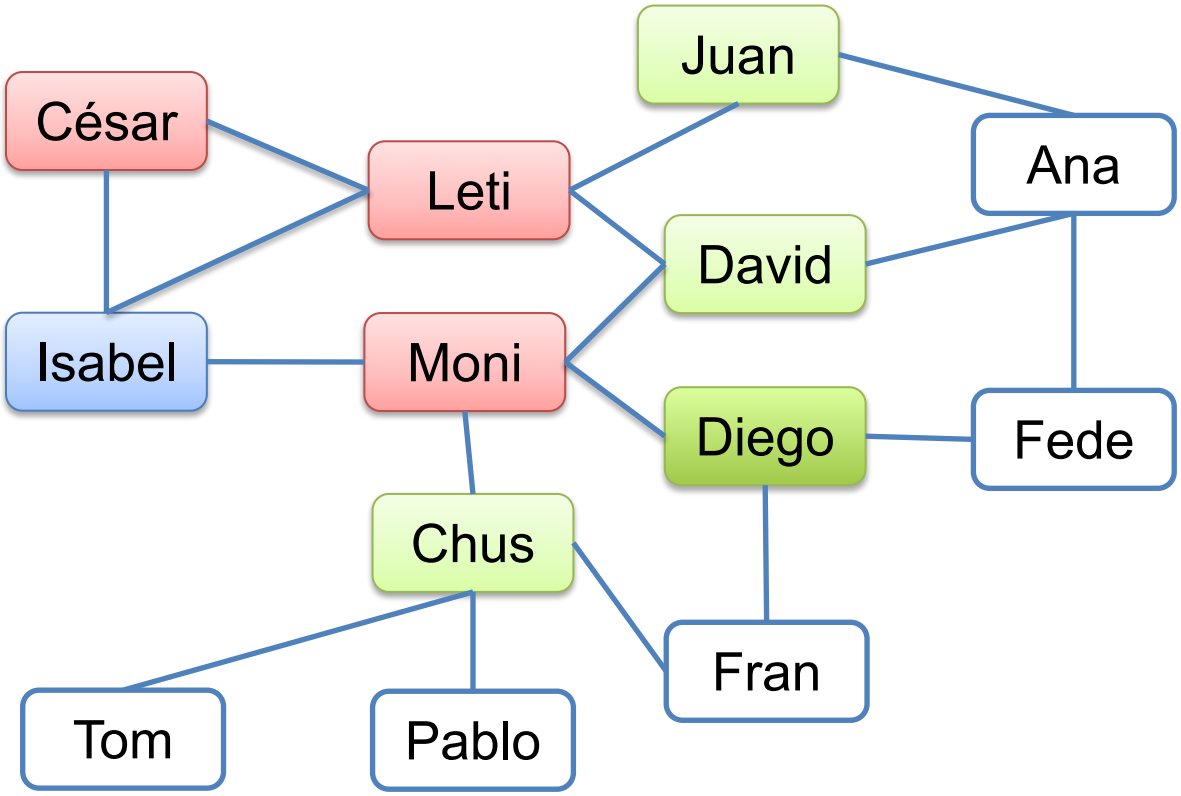


¿Cómo sugerir nuevos amigos a Isabel?

Conceptos sobre grafos

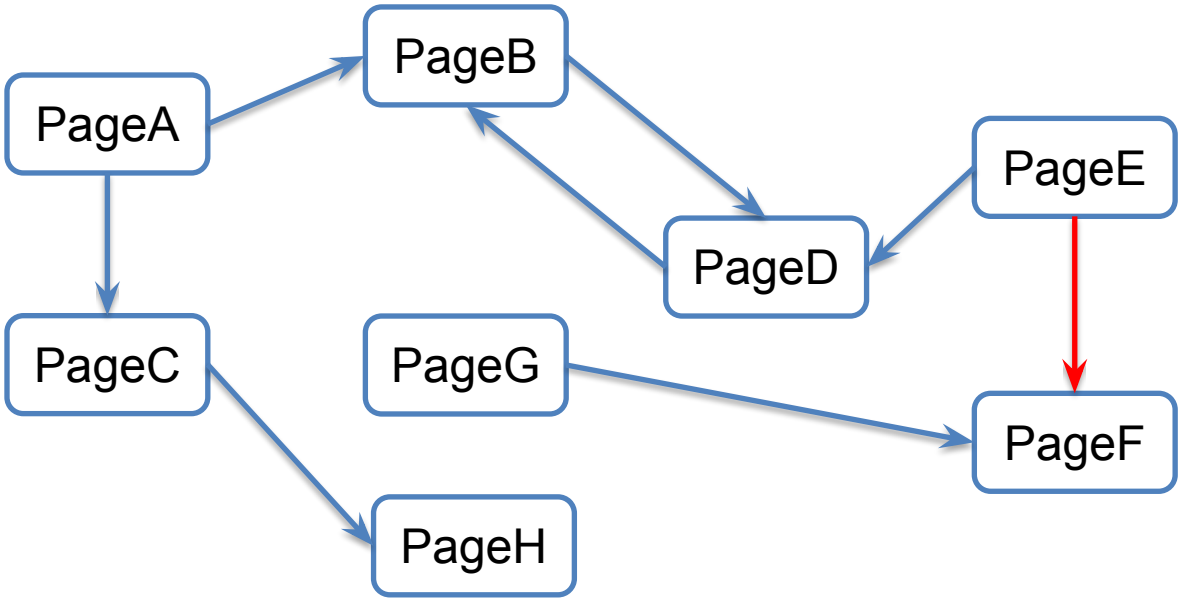


Conceptos sobre grafos



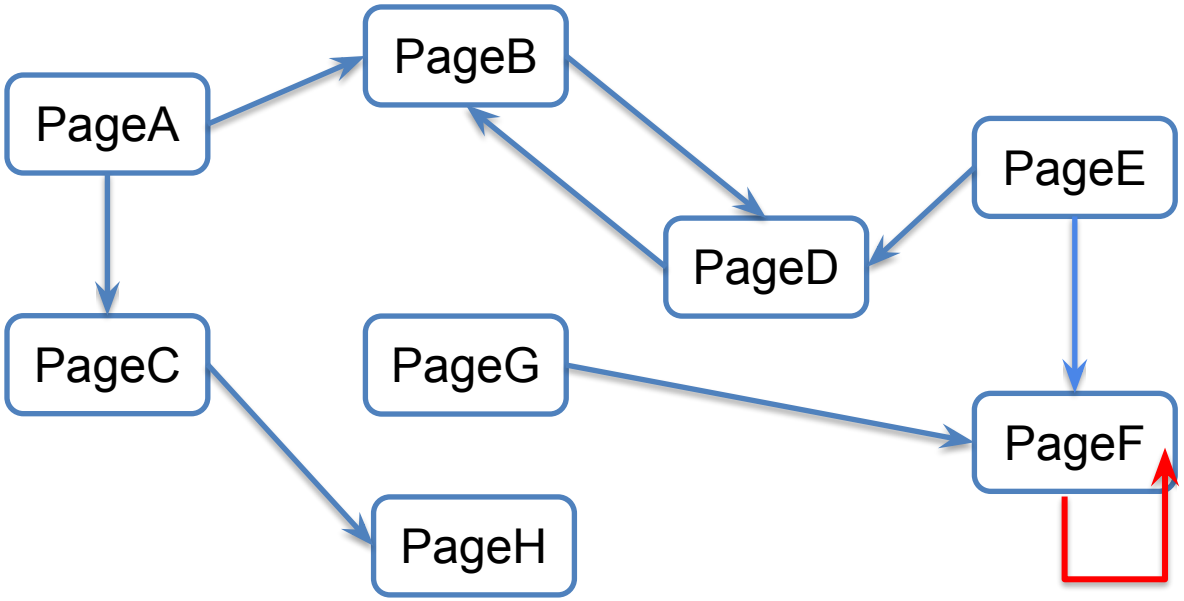
Encontrar todos los nodos para los que exista un camino de longitud 2

Conceptos sobre grafos



La Web se puede representar como un grafo dirigido. Los vértices son las páginas web y las conexiones entre estas son las aristas del grafo.

Conceptos sobre grafos

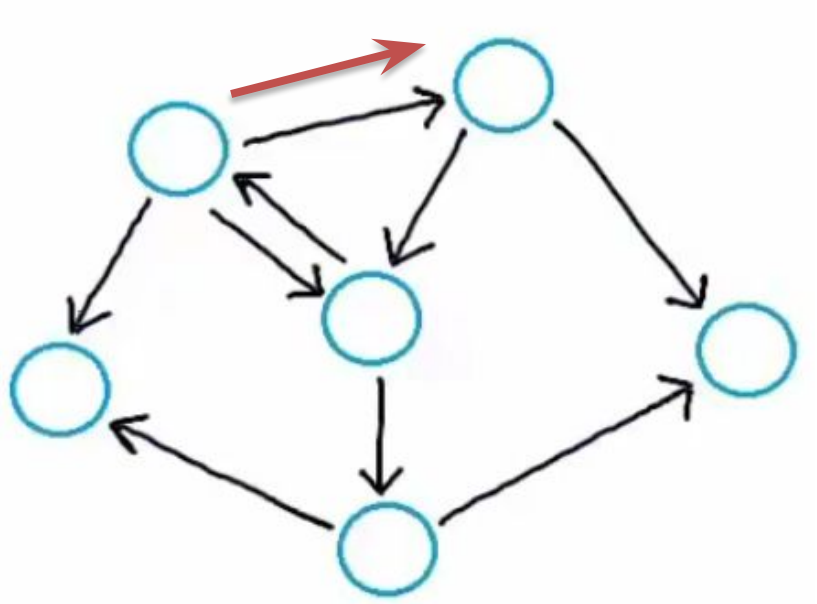


Bucle (loop)

Una página puede contener un enlace a ella misma. Ese tipo de aristas son conocidas como **bucles** (**loop**) y son aristas que conectan un vértice consigo mismo.

Conceptos sobre grafos

Aristas múltiples (aristas paralelas)

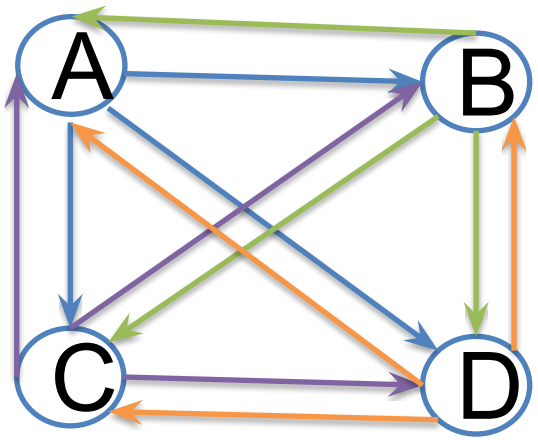


Conceptos sobre grafos

- Los bucles y las aristas paralelas tienden a hacer más complejos los algoritmos de grafos.
- Un **grafo simple** es un grafo que no tiene bucles ni aristas paralelas.

Conceptos sobre grafos

- ¿Cuál es el número mínimo y máximo de aristas en un **grafo simple dirigido**?



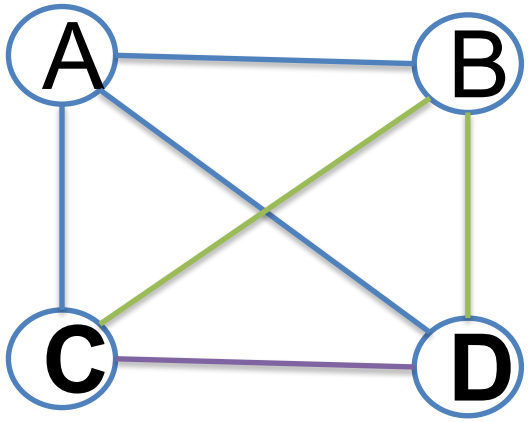
|V| = 4
|A| = 0 (mínimo)

|V| = 4
|A| = 12 (máximo)

Si */V/ = n, cada vértice podría tener un máximo de n-1 aristas. Por tanto,* ***0<= /A/<=n(n-1)***

Conceptos sobre grafos

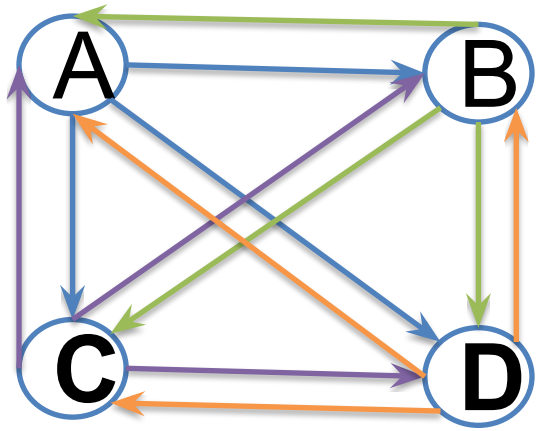
- ¿Cuál es el número máximo de aristas en un **grafo simple no dirigido**?



Si */V/ = n, cada vértice podría tener n-1 aristas.*
0<= /A/<=n(n-1)/2

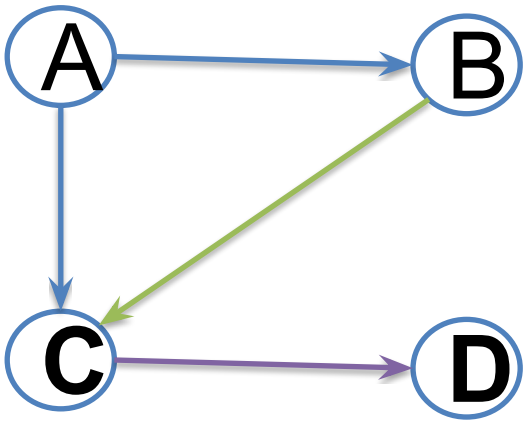
Conceptos sobre grafos

- Un grafo es **denso** si el número de sus aristas es cercano a su número máximo posible (n(n-1) o n(n-1)/2) ($\approx n^2$)



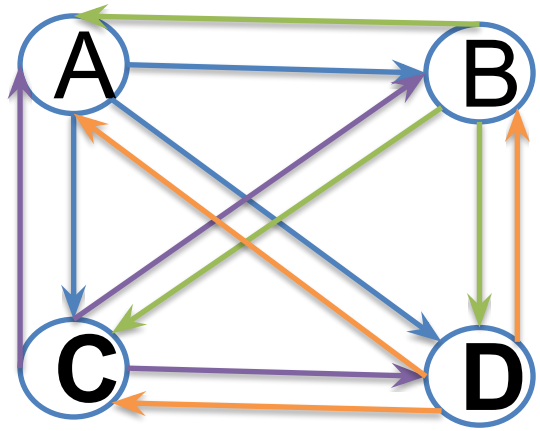
Conceptos sobre grafos

- Un grafo es **escaso** si el número de sus aristas es cercano a el número de sus vértices (≈ *n)*

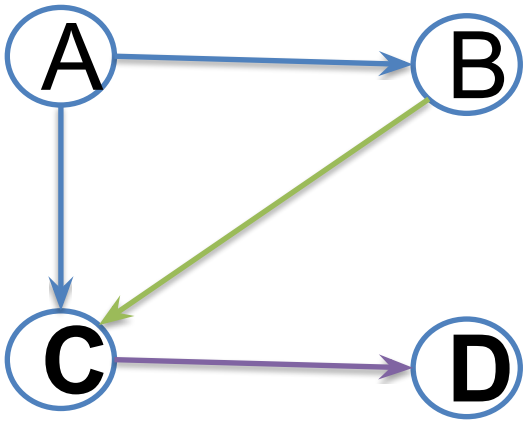


Conceptos sobre grafos

- Conocer si un grafo es denso o escaso nos ayudará a elegir la implementación más apropiada.



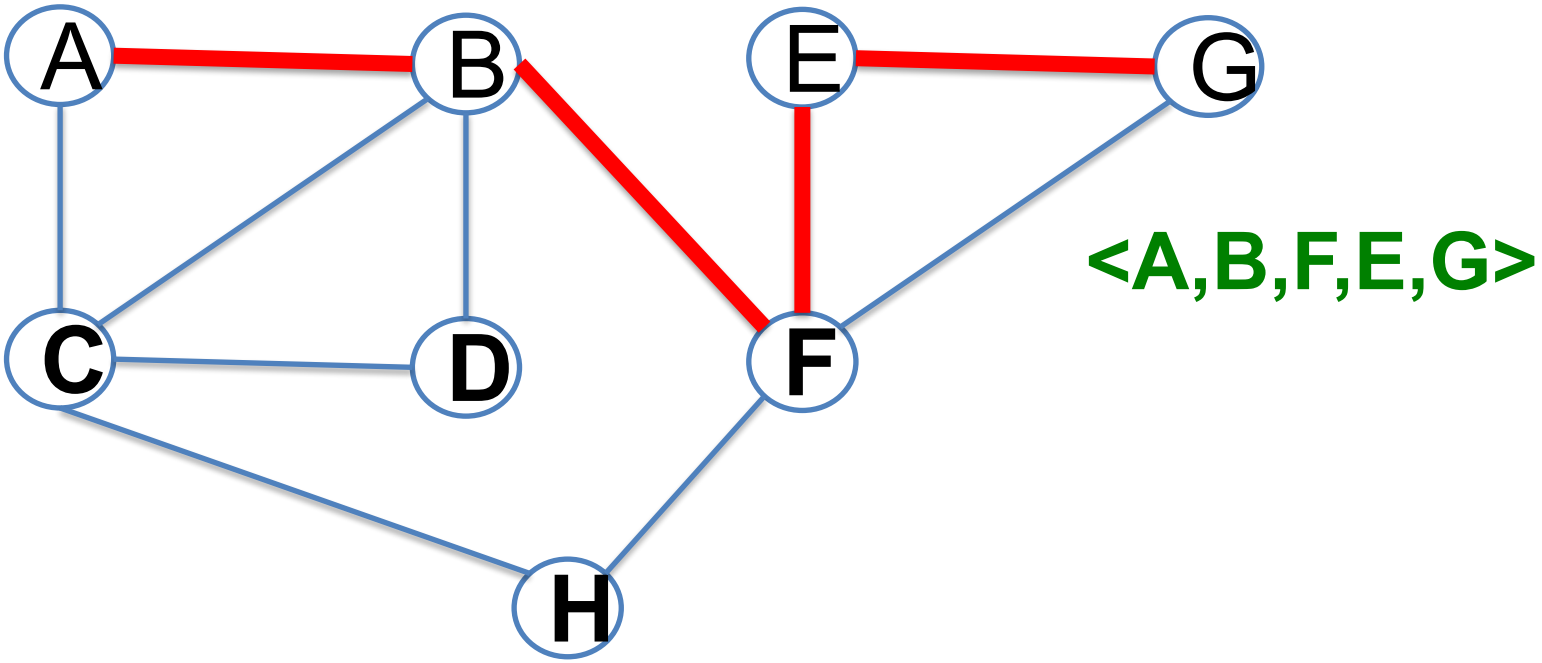
dense



sparse

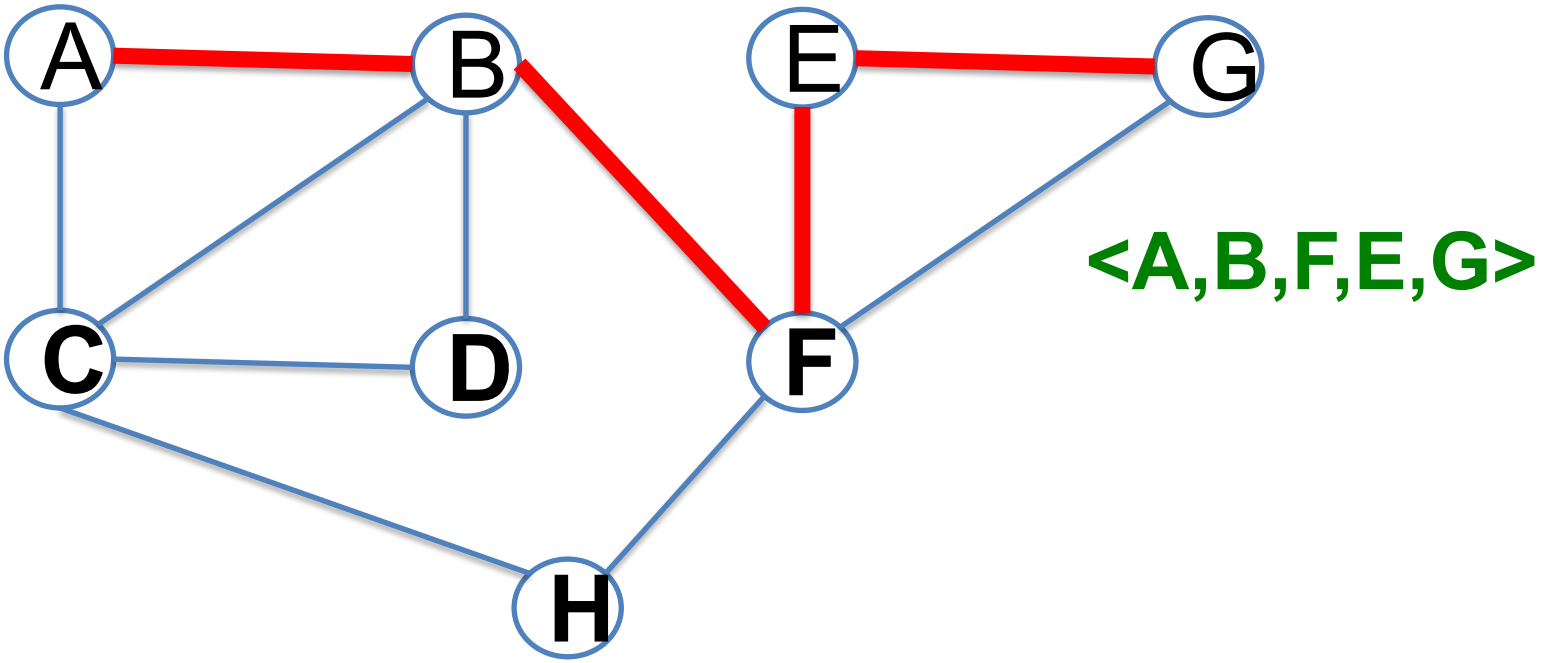
Conceptos sobre grafos

- Un **camino** es una secuencia de vértices tal que exista una arista entre cada vértice y el siguiente.



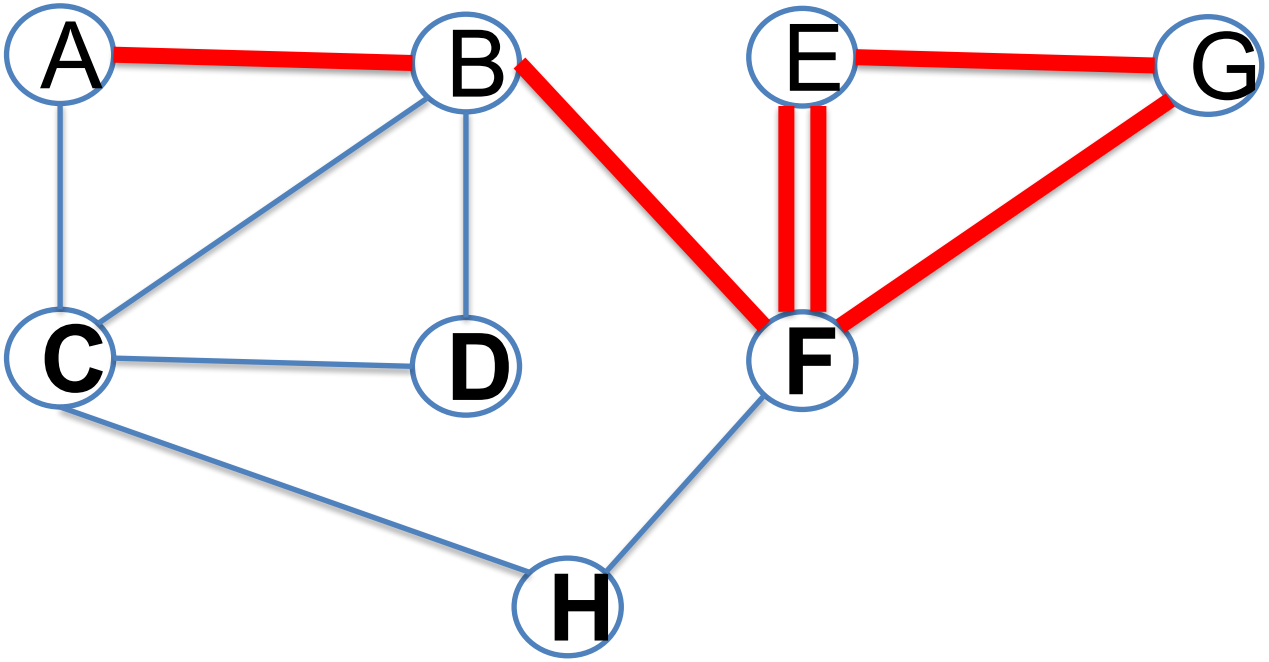
Conceptos sobre grafos

- Un **camino simple** es aquel que no repite vértices en su recorrido.



Conceptos sobre grafos

- Este camino no es simple porque hay dos vértices repetidos **<A,B,F,E,G,F,E>**



Índice

- Introducción
- Conceptos sobre grafos
- **TAD Grafo**
- Implementaciones:
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

TAD Grafo

G = (V, A), donde V es el conjunto de vértices y A es el conjunto de aristas.

Operaciones:

- **Constructor**. En nuestro caso, el constructor va a recibir como parámetro el conjunto de vértices. Además, también podemos incluir un parámetro para indicar si el grafo es dirigido o no.
- **add_edge(start, end, weight=1)**: crea una arista de start a end, con peso weight. Si el grafo no está dirigido, también añade la arista de end a start.

TAD Grafo

- **contains_edge(start,end)**: comprueba si existe la arista de start a end existe. Si existe, devuelve su peso asociado.
- **remove_edge(start, end)**: borra la arista de start a end.

TAD Grafo

Más operaciones:

- **bfs()**: devuelve (o imprime) el recorrido en anchura del grafo.
- **_bfs(v)**: devuelve (o imprime) el recorrido en anchura desde el vértice v.
- **dfs()**: devuelve (o imprime) el recorrido en profundidad del grafo.
- **_dfs(v)**: devuelve (o imprime) el recorrido en profundidad desde el vértice v.

Además, también se pueden incluir otras operaciones como los **algoritmos de camino mínimo**.

TAD Grafo

Más operaciones:

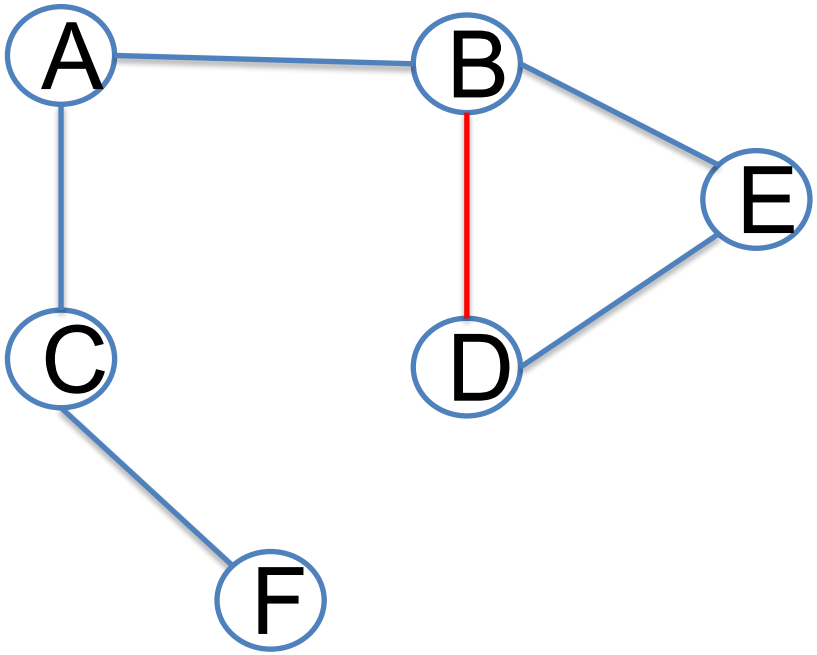
- **bfs()**: devuelve (o imprime) el recorrido en anchura del grafo.
- **_bfs(v)**: devuelve (o imprime) el recorrido en anchura desde el vértice v.
- **dfs()**: devuelve (o imprime) el recorrido en profundidad del grafo.
- **_dfs(v)**: devuelve (o imprime) el recorrido en profundidad desde el vértice v.
- **minimum_path**(start, end): devuelve una lista de vértices con el camino mínimo de vértice *start* a *end*.

Índice

- Introducción
- Conceptos sobre grafos
- TAD Grafo
- **Implementaciones:**
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

Implementación

G=(V,A), V vértices, A aristas

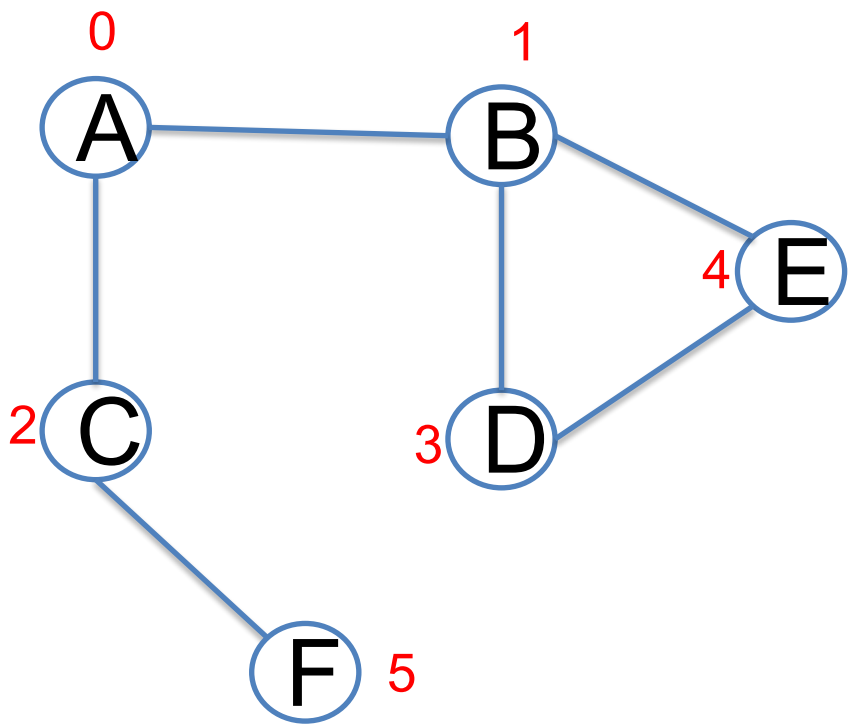


¿Cómo puede representar un grafo?

Índice

- Introducción
- Conceptos sobre grafos
- TAD Grafo
- **Implementaciones:**
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

Matriz de Adyacencia

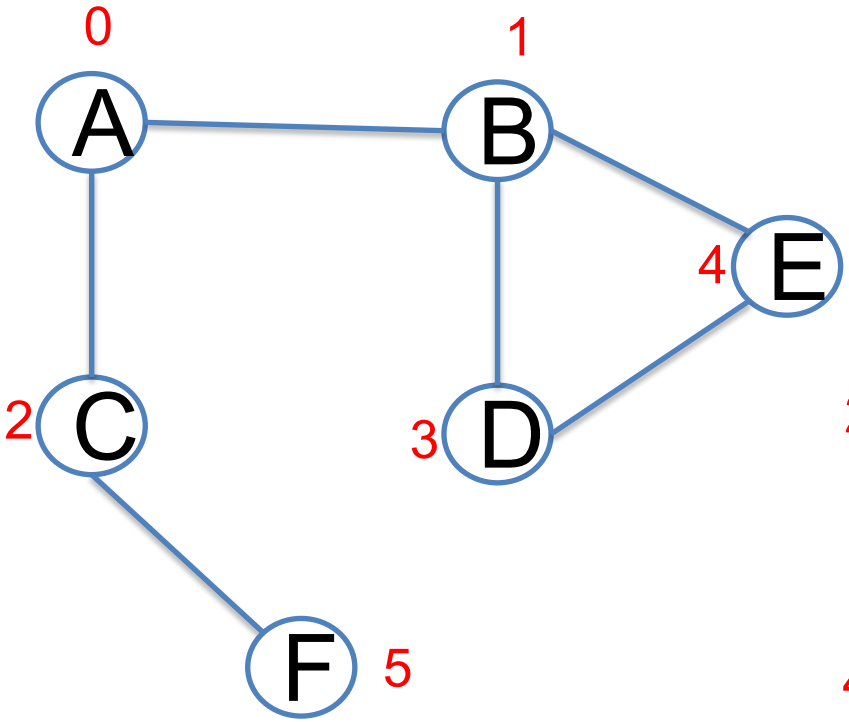


Lista de vértices

0	A
1	B
2	C
3	D
4	E
5	F

Cada vértice es representado por un índice. Podemos usar una lista de Python (array) para almacenar los vértices.

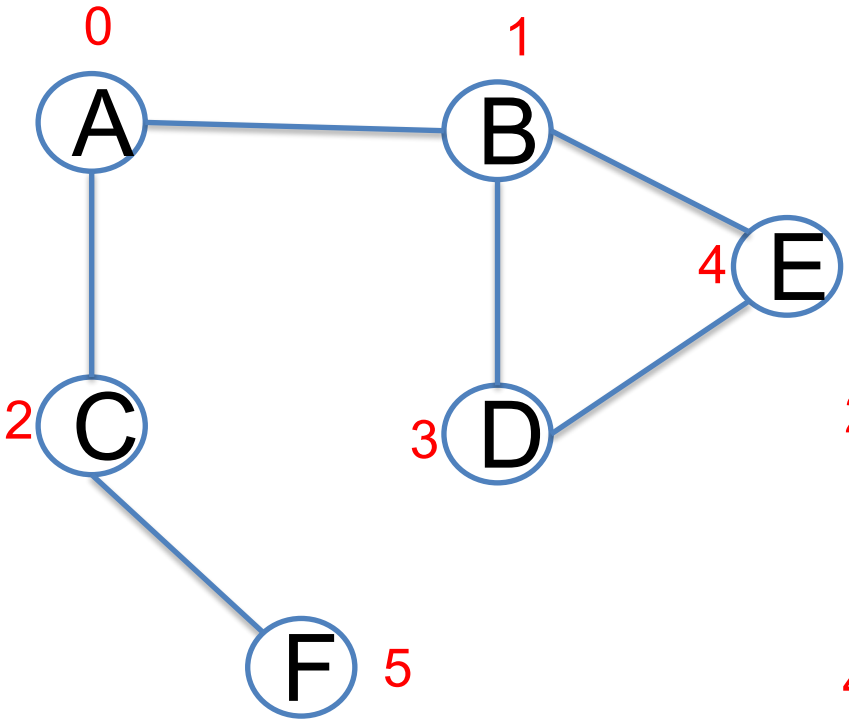
Matriz de Adyacencia (grafo no dirigido)



0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

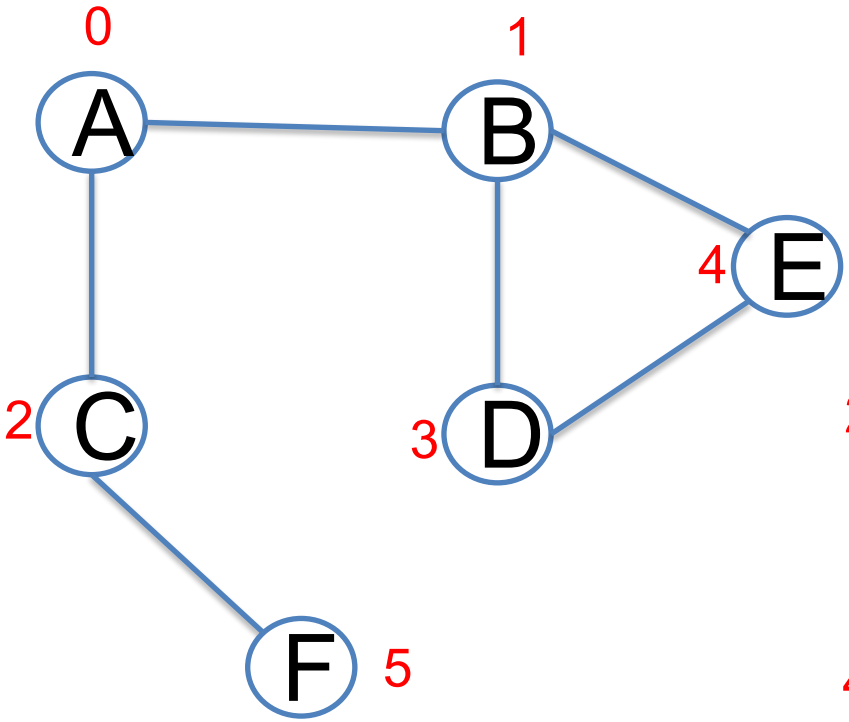
Matriz de Adyacencia (grafo no dirigido)



		0	1	2	3	4	5
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

$$Mij = \begin{cases} 1, & \text{if } \{i, j\} \text{ si es una arista} \\ 0, & \text{eoc} \end{cases}$$

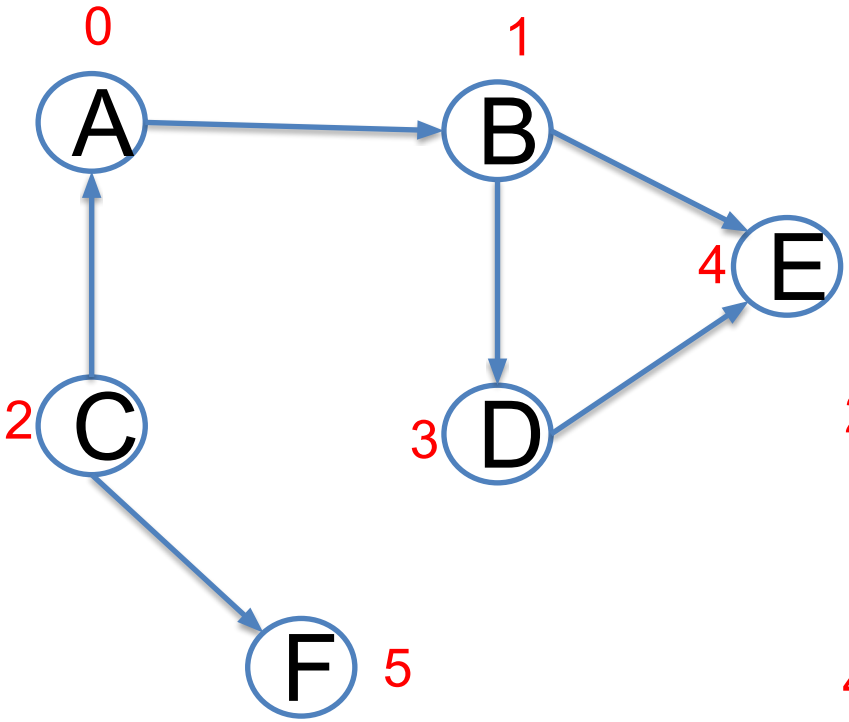
Matriz de Adyacencia (grafo no dirigido)



		0	1	2	3	4	5
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

En los grafos no dirigidos, la matriz de adyacencia va a ser simétrica: Mij = Mji

Matriz de Adyacencia (grafo no dirigido)

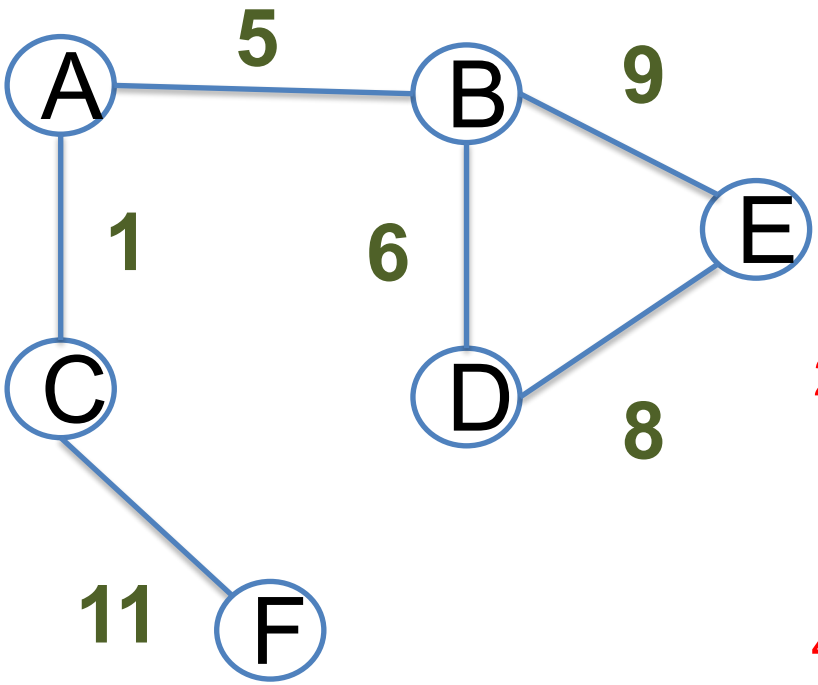


		0	1	2	3	4	5
0	A	0	1	0	0	0	0
1	B	0	0	0	1	1	0
2	C	1	0	0	0	0	0
3	D	0	0	0	0	1	0
4	E	0	0	0	0	0	0
5	F	0	0	0	0	0	0

$$Mij = \begin{cases} 1, & \text{if (i, j) si es una arista} \\ 0, & \text{eoc} \end{cases}$$

En este caso, **la matriz no es simétrica**

Matriz de Adyacencia (grafo no dirigido)

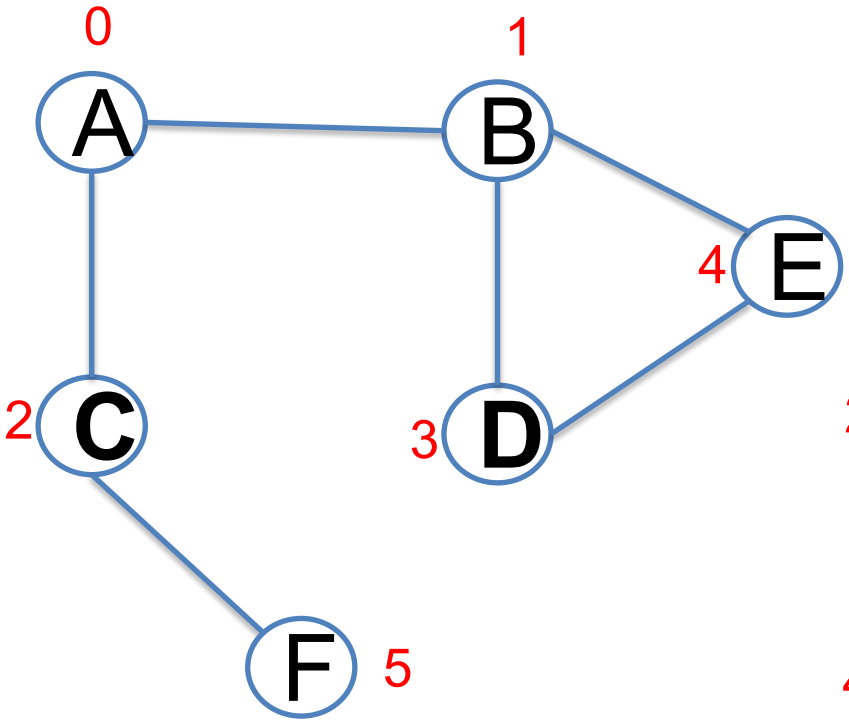


0	A
1	B
2	C
3	D
4	E
5	F

	0	1	2	3	4	5
0	∞	5	1	∞	∞	∞
1	5	∞	∞	6	9	∞
2	1	∞	∞	∞	∞	11
3	∞	6	∞	∞	8	∞
4	∞	9	∞	8	∞	∞
5	∞	∞	11	∞	∞	∞

¿Cómo representar grafos ponderados?

Matriz de Adyacencia - Complejidad Espacial



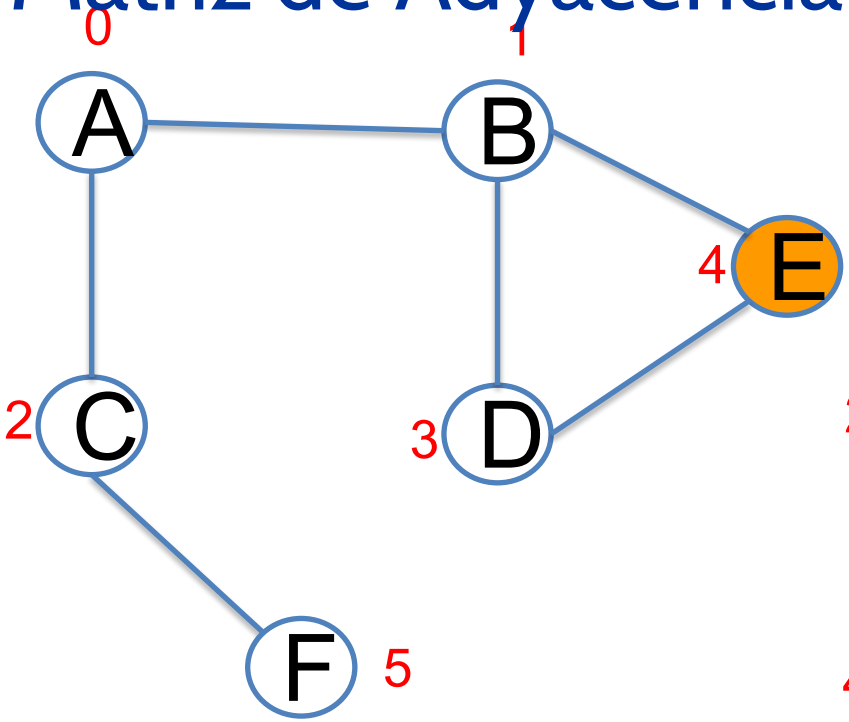
0	A
1	B
2	C
3	D
4	E
5	F

0	0	1	1	0	0	0
1	1	0	0	1	1	0
2	1	0	0	0	0	1
3	0	1	0	0	1	0
4	0	1	0	1	0	0
5	0	0	1	0	0	0

<u>Complejidad espacial</u>

If $|V| = n$, $O(n^2)$

Matriz de Adyacencia - Complejidad Temporal

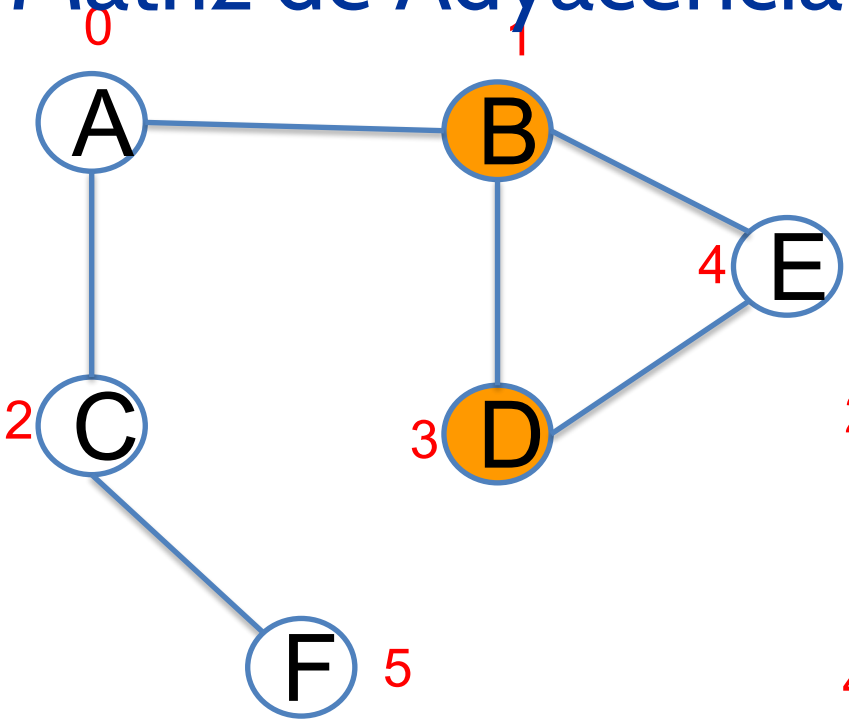


		0	1	2	3	4	5
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

Complejidad temporal
buscar los vecinos de E?

O(n)

Matriz de Adyacencia - Complejidad Temporal



		0	1	2	3	4	5
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

¿son vecinos? **O(n) + O(1)**

Matriz de Adyacencia - Implementación

- Implementación de grafo no ponderado y no dirigido basado en matriz de adyacencia.
- Implementación de grafo (cualquier tipo) basado en matriz de adyacencia.

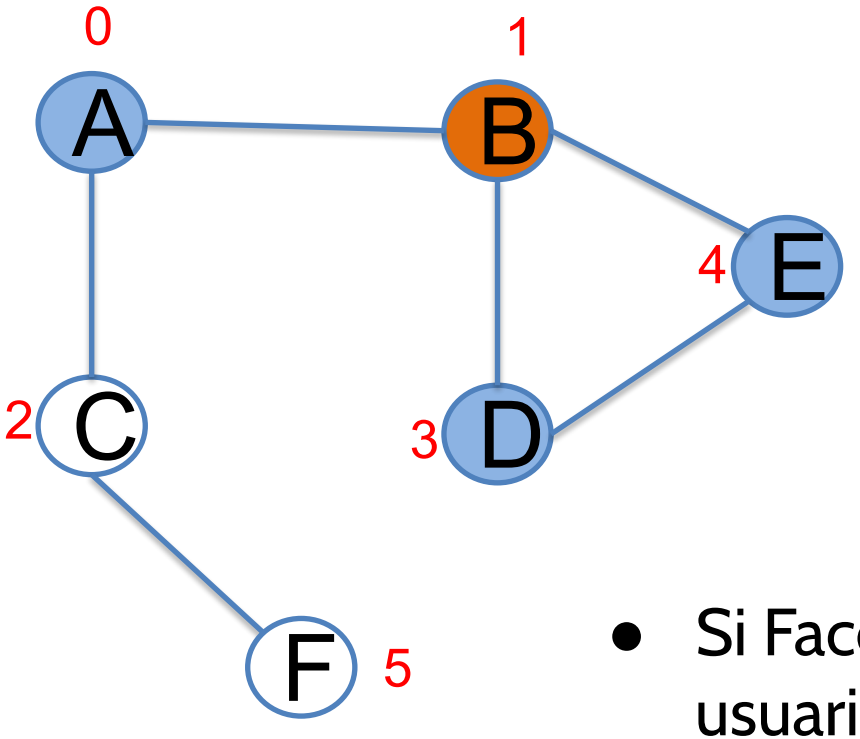
Matriz de Adyacencia - Conclusiones

- En términos de **complejidad temporal**, la matriz de adyacencia es una estructura eficiente (**O(n)**).
- Sin embargo, en términos de complejidad **espacial**, es una representación demasiado costosa (**$O(n^2)$**).
- Se considera una **implementación adecuada** cuando **n^2 es pequeño** o el **grafo es denso** (número de aristas próximo a n^2).
- La **mayoría** de los **grafos reales** son **escasos** (por ejemplo, WWW).

Índice

- Introducción
- Conceptos sobre grafos
- TAD Grafo
- **Implementaciones:**
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

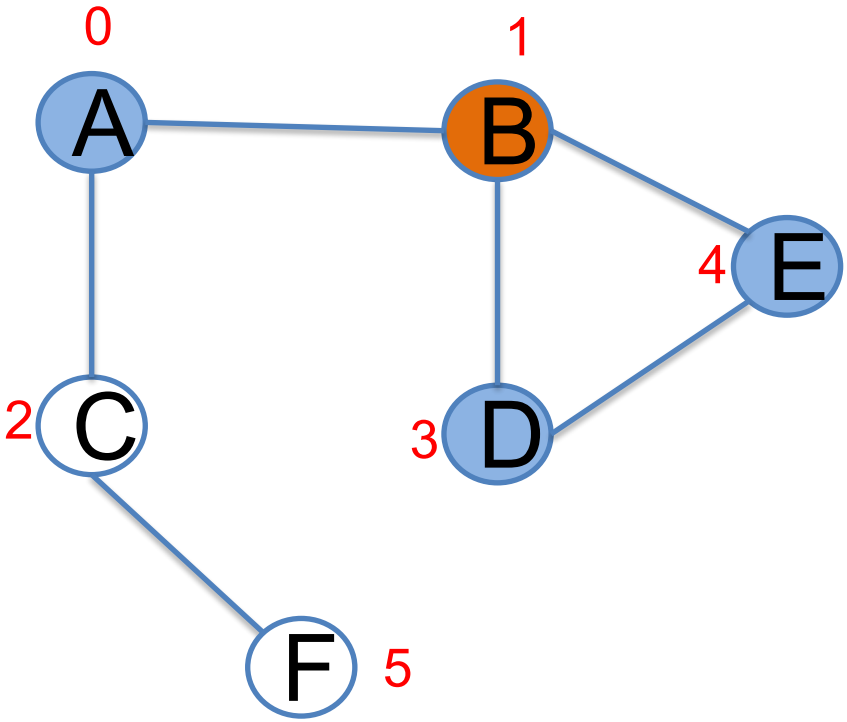
Matriz de Adyacencia - Conclusiones



0	1	2	3	4	5
1	0	0	1	1	0

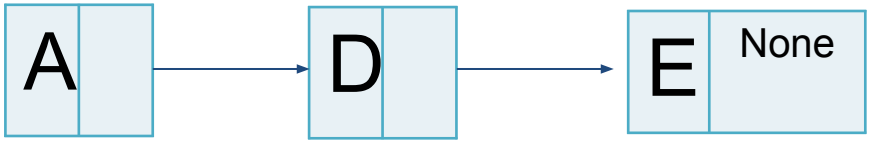
- Si Facebook tiene 1000 millones de usuarios (10^9), las filas de la matriz de adyacencia son de dimensión 10^9
- Si un usuario, B, tiene 1000 amigos, en su fila, habrá:
  - Número de 1s: 1000 = 1 KB y
  - Número de 0s: 10^9 - 1000 = 1 GB

Lista de Adyacencia

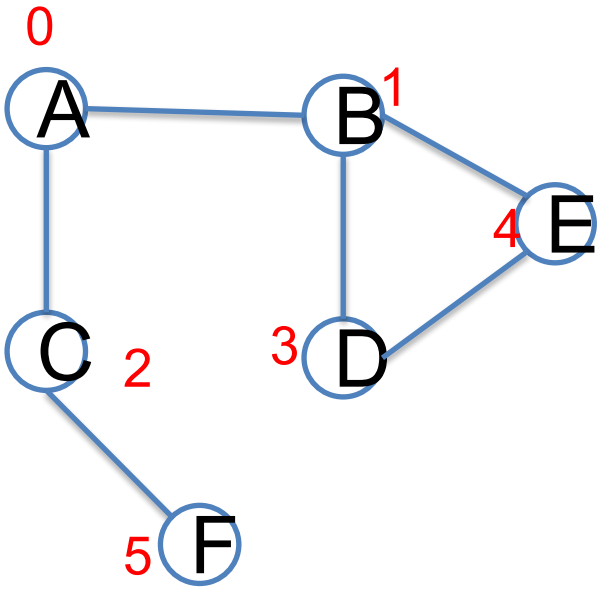


Para representar los **vecinos de B,** sería suficiente con almacenar sus índices en un **lista de python** o en una **lista enlazada**.

Vecinos de B:



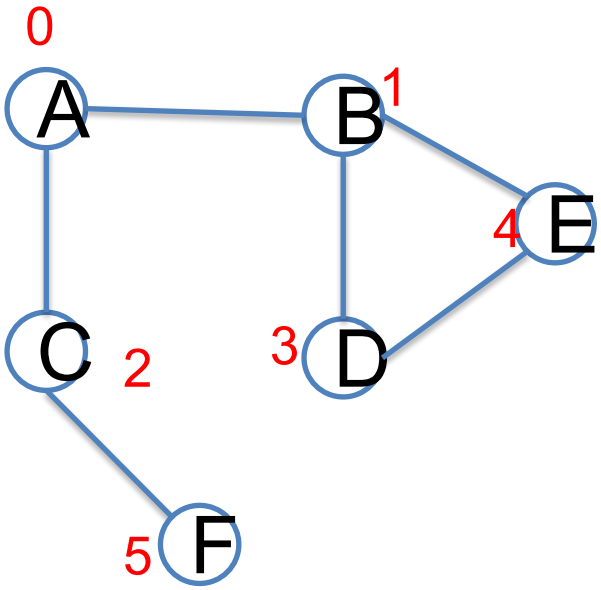
Lista de Adyacencia

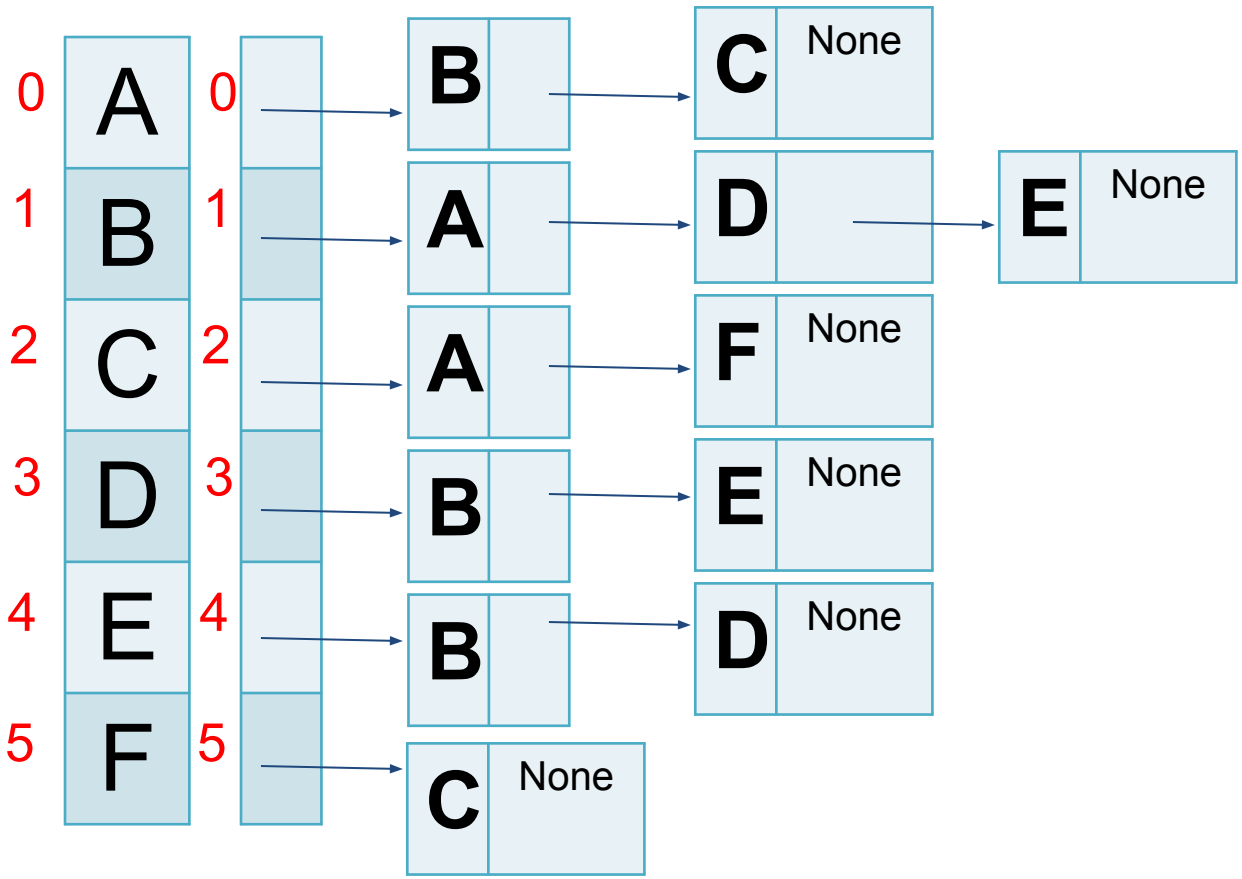


0	A
1	B
2	C
3	D
4	E
5	F

Seguiremos utilizando un array (lista de python) para almacenar los vértices

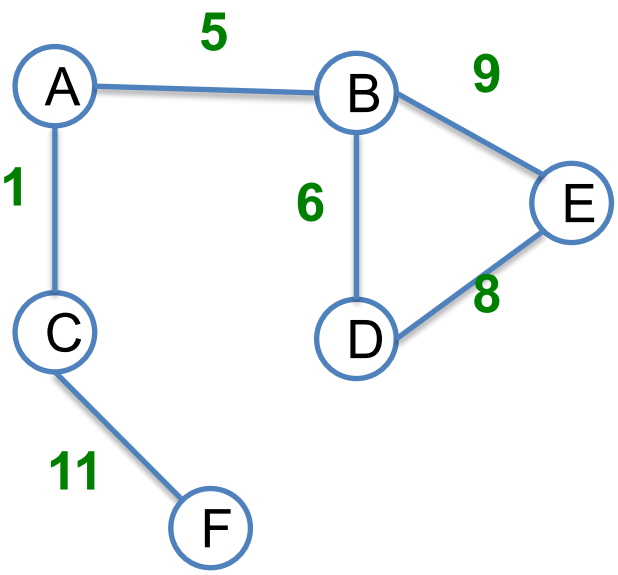
Lista de Adyacencia

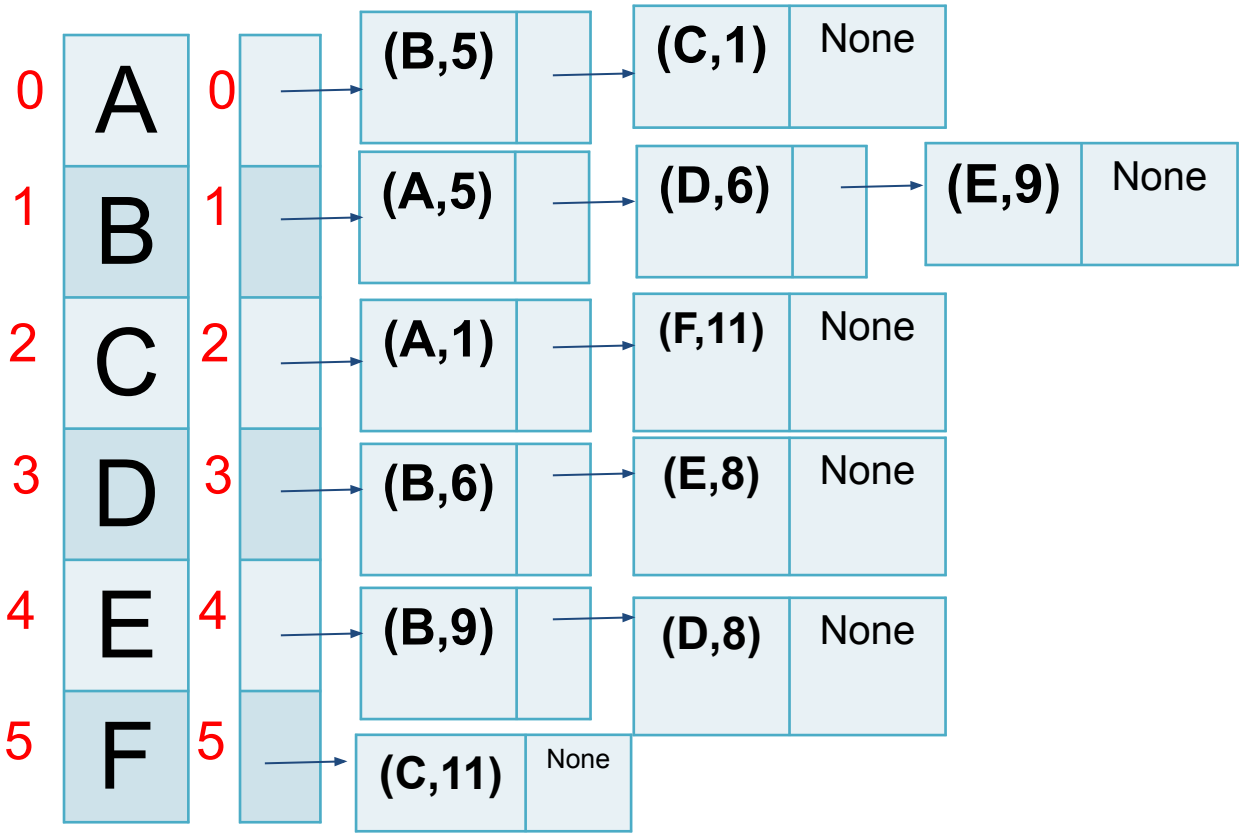




Usaremos un array de listas enlazadas para almacenar los vértices adyacentes a cada vértice.

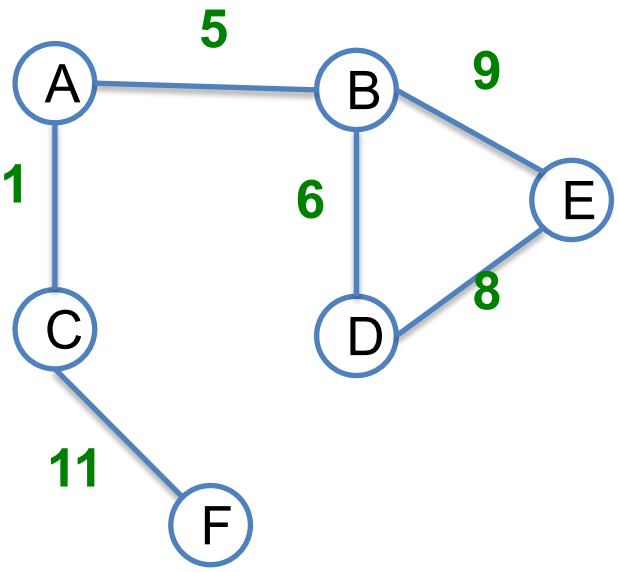
Lista de Adyacencia (grafo ponderado)

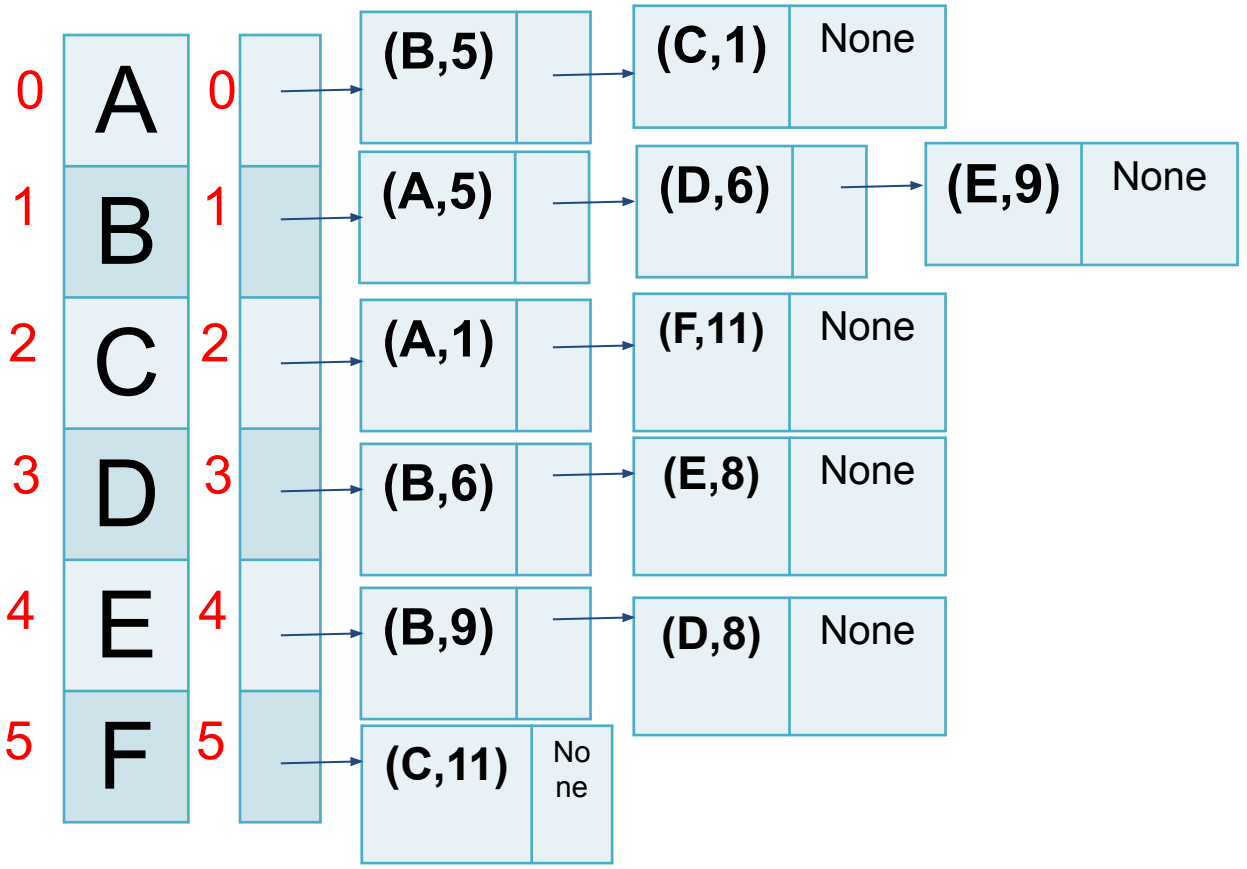




Cada vértice adyacente se representa como un **par (v,w)** donde **v es el vértice adyacente**, y **el peso de la arista**.

Lista de Adyacencia - Complejidad Espacial

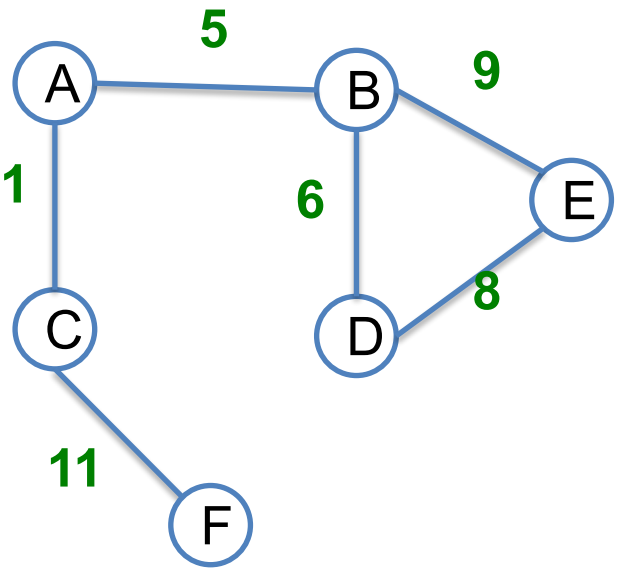


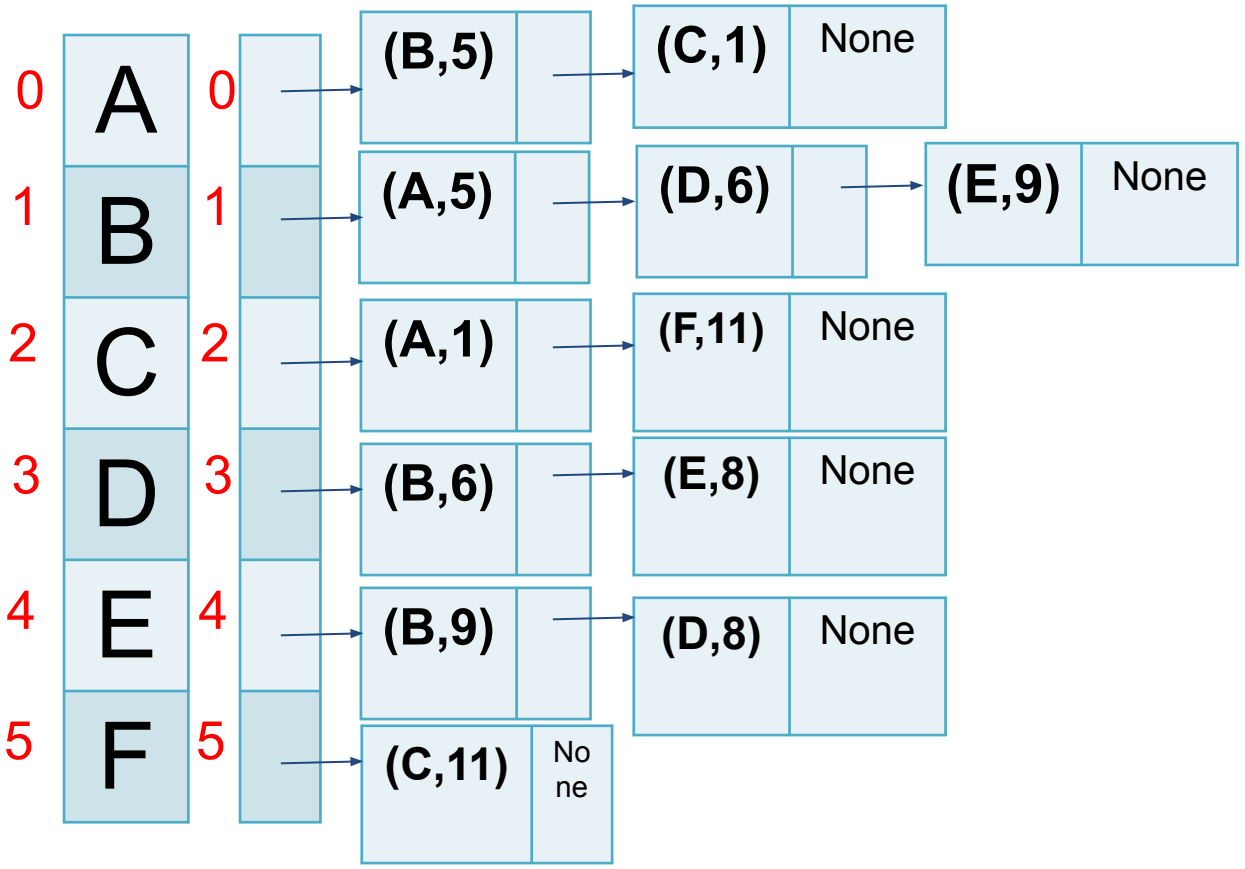


O(|A|)

Recuerda que el número máximo de aristas en un grafo simple es n(n-1)/2 (no dirigido) y n(n-1) (dirigido)

Lista de Adyacencia - Complejidad Espacial

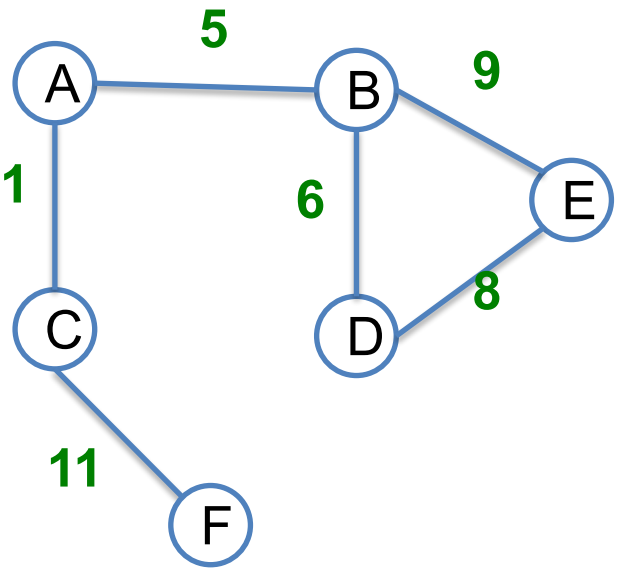


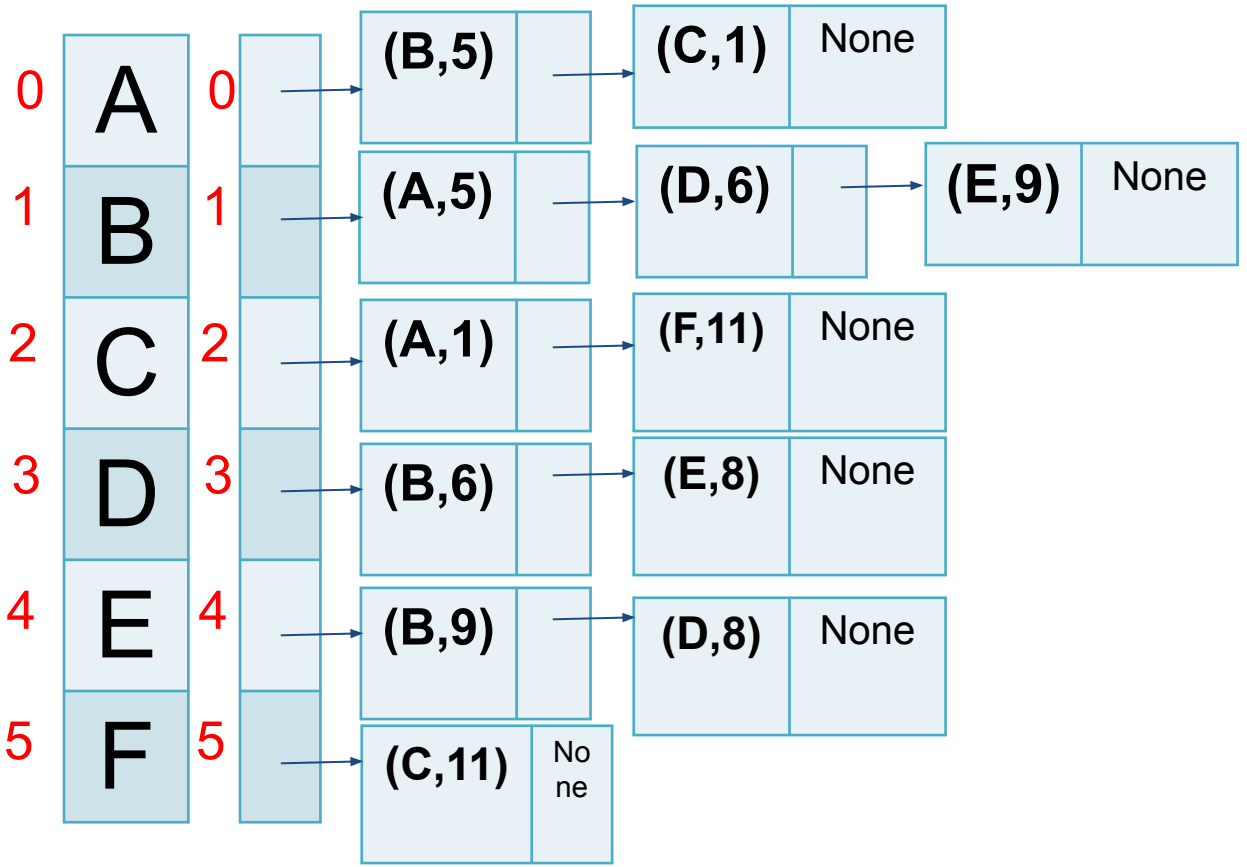


Si el grafo es denso: O(|A|) -> n2

Si el grafo es escaso: O(|A|) -> n

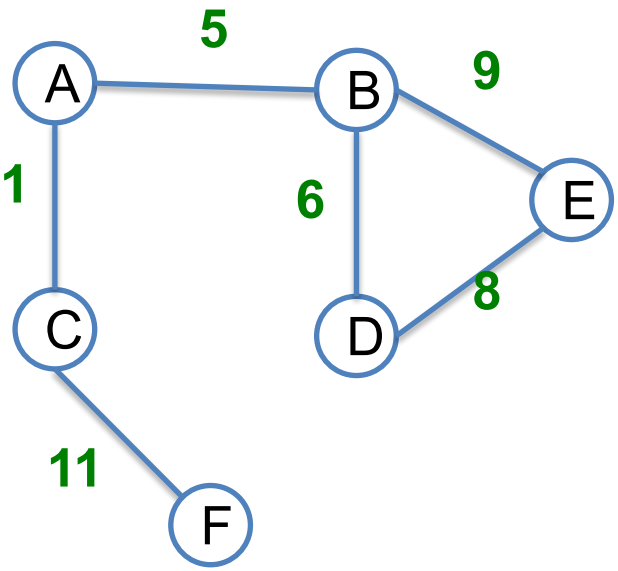
Lista de Adyacencia - Complejidad Temporal

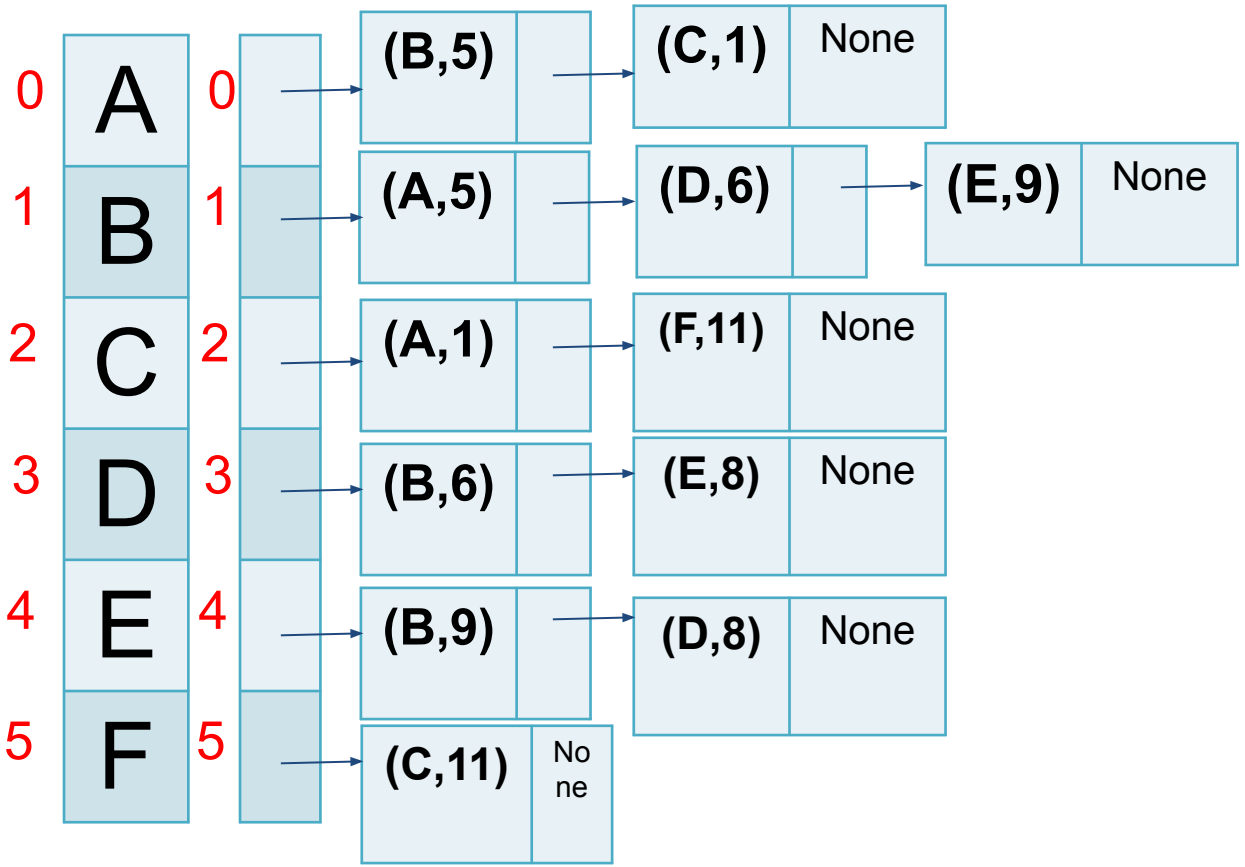




Obtener vértices adyacentes a E?
O(n) (obtener su índice=4) y O(1) devolver la lista de adyacencia asociada al índice

Lista de Adyacencia - Complejidad Temporal





Comprobar si E y F son vecinos?
O(n) (obtener el índice de E) y O(n) comprobar si F está en la lista de adyacencia de E

Lista de Adyacencia - Implementación

- Implementación de grafo (cualquier tipo) basado en lista de adyacencia.

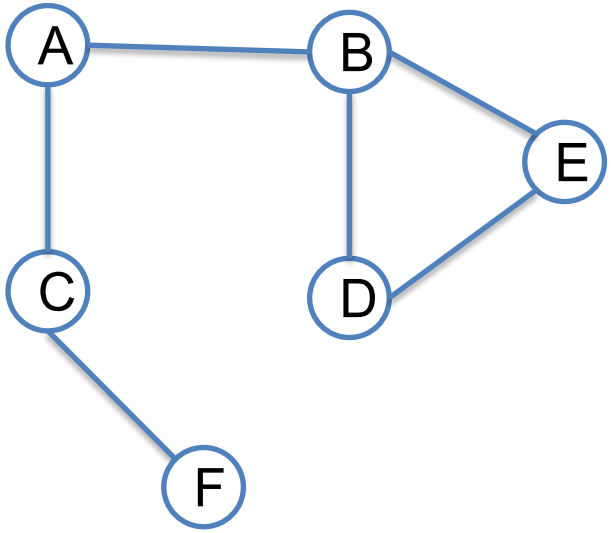
Lista de Adyacencia - Conclusiones

- En términos de **complejidad temporal**, la lista de adyacencia y la matriz de adyacencia son similares. Para la mayorías de las operaciones, su complejidad es **O(n) donde n=|V|.**
- Sin embargo, en términos de **complejidad espacial**, la lista de adyacencia es una estructura más eficiente **O(|A|)**.
- La mayoría de los **grafos reales son escasos**, y por tanto, **|A|≈|V|=n**. Por tanto, la **complejidad espacial será O(n)**.

Índice

- Introducción
- Conceptos sobre grafos
- TAD Grafo
- **Implementaciones:**
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- Algoritmo de camino mínimo (Dijkstra).

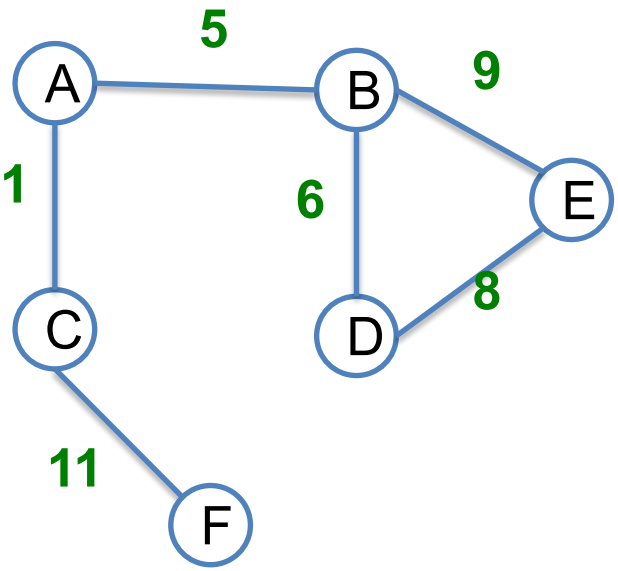
Diccionarios




```
graph = {
'A':['B','C'],

'B':['A','D','E'],

'C':['A','F'],

'D':['B','E'],

'E':['B','D'],

'F':['C']}
```

Los vértices del grafo se van a representar como las claves (keys)del diccionario.
En el diccionario, asociado a cada clave (vértice), se guarda la lista de sus vértices adyacentes

Diccionarios (grafos ponderados)




```
graph = {
'A':[('B',5),('C',1)],

'B':[('A',5),('D',6),('E',9)],

'C':[('A',1),('F',11)],

'D':[('B',6),('E',8)],

'E':[('B',9),('D',8)],

'F':[('C',11)]   }
```

Diccionarios - Implementación

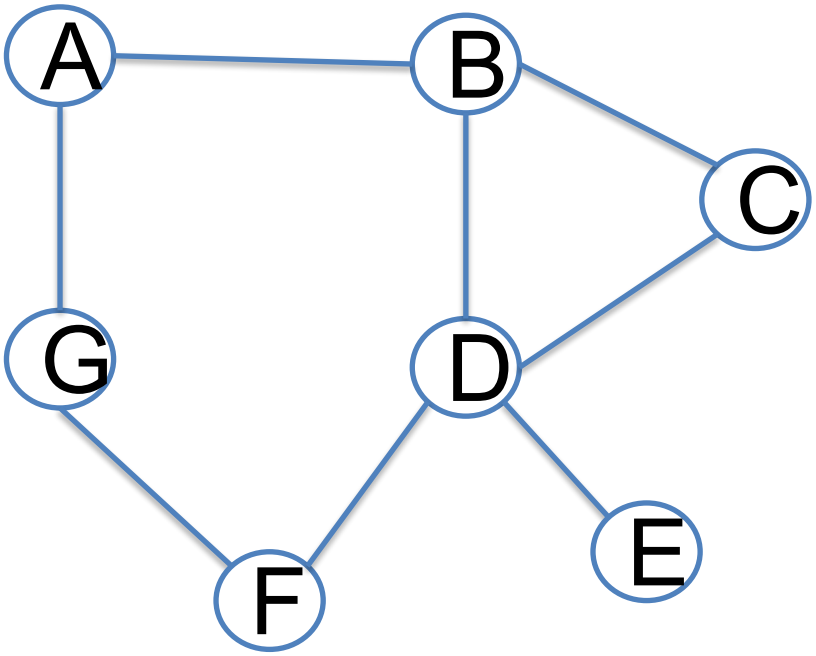
- Implementación de grafo (cualquier tipo) basado en diccionarios de Python (recomendada).

Índice

- Introducción
- Conceptos sobre grafos
- TAD Grafo
- Implementaciones:
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- **Recorridos**
- Algoritmo de camino mínimo (Dijkstra).

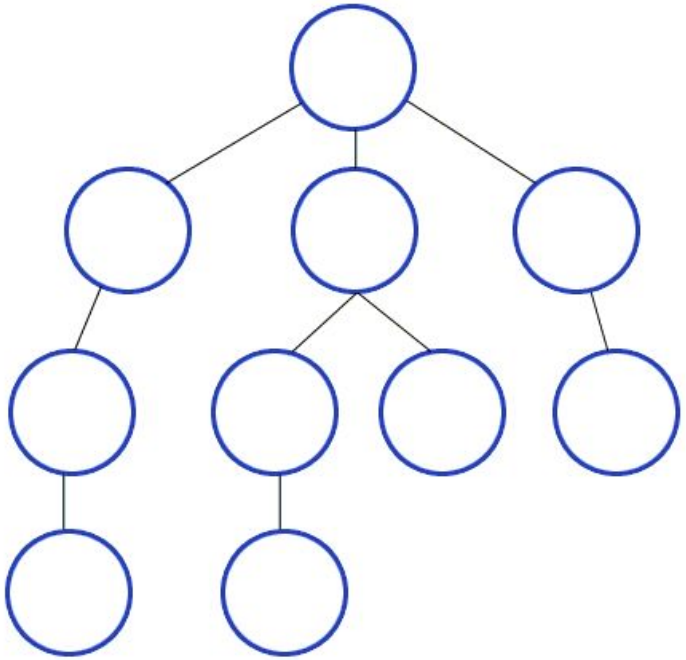
Recorridos (Graph Traversals)

- Problema del viajante: encontrar una ruta que le permita visitar todas las ciudades (vértices)

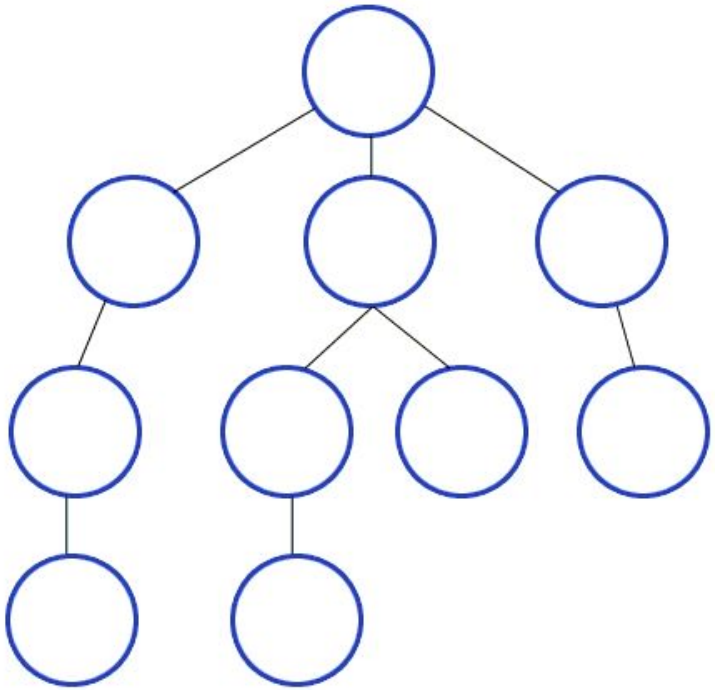


Recorridos de grafos

Recorrido en anchura
(Breadth first search)

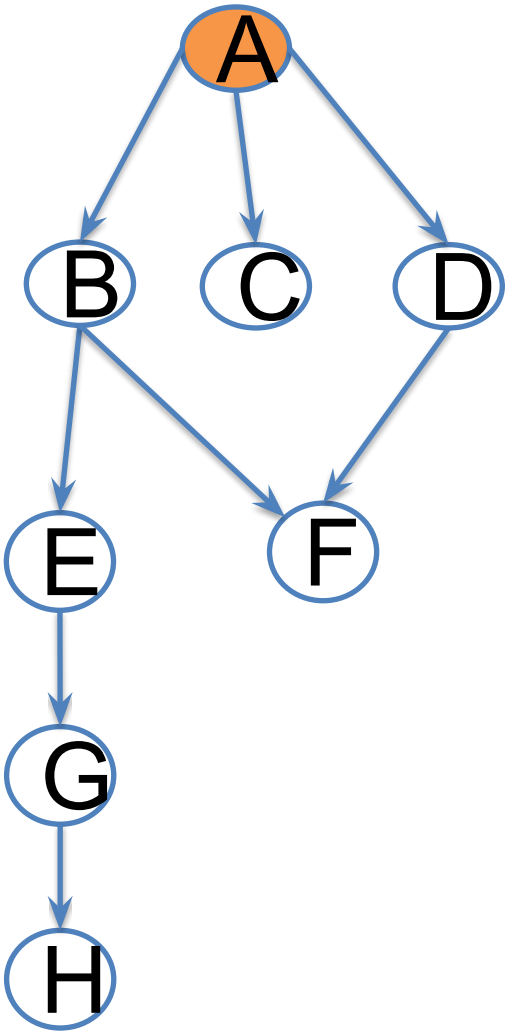


Recorrido en profundidad
(depth first search)



Source: https://www.thedshandbook.com/breadth-first-search/

Recorrido en anchura (bfs)

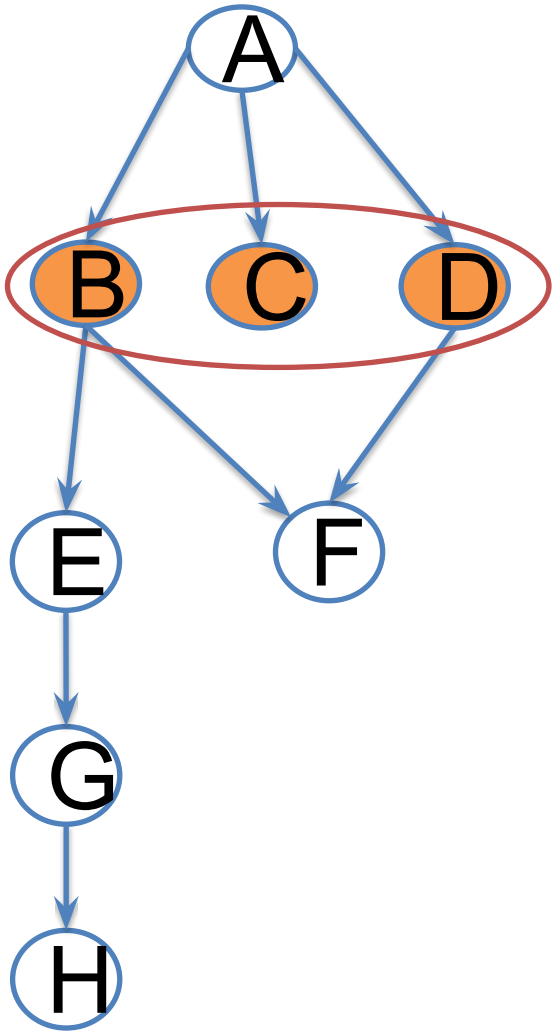


Breadth-first traversal (BFS)

- Idea: visitar todos los vértices por niveles.
- Algoritmo similar al recorrido por niveles de los árboles.

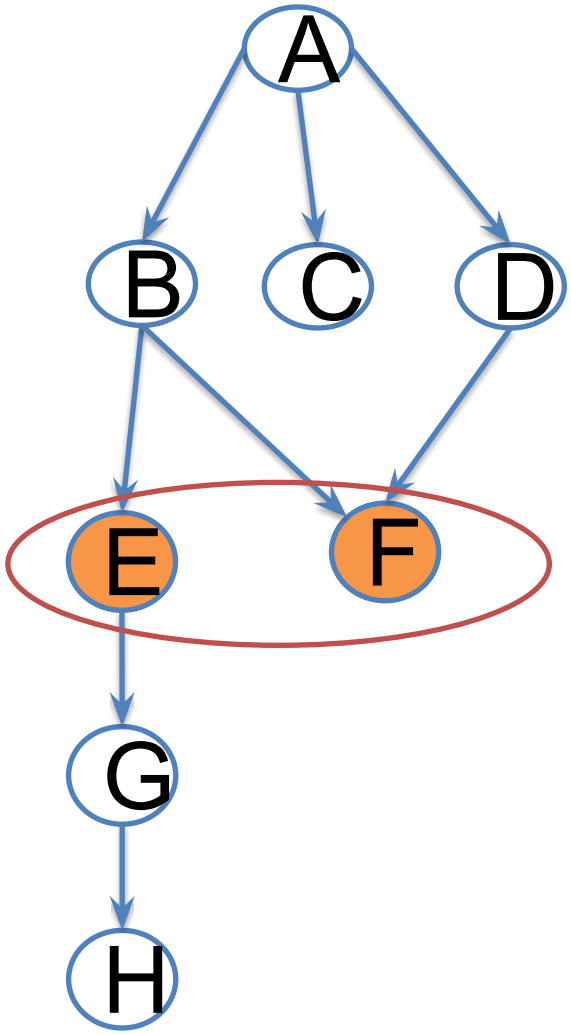
Output: A

Recorrido en anchura (bfs)



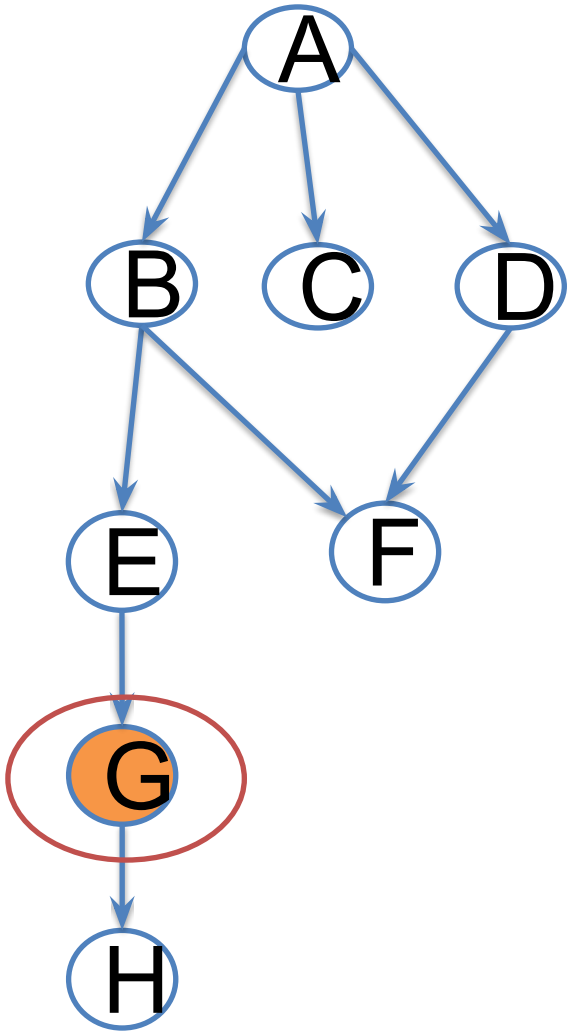
Output: A B C D

Recorrido en anchura (bfs)



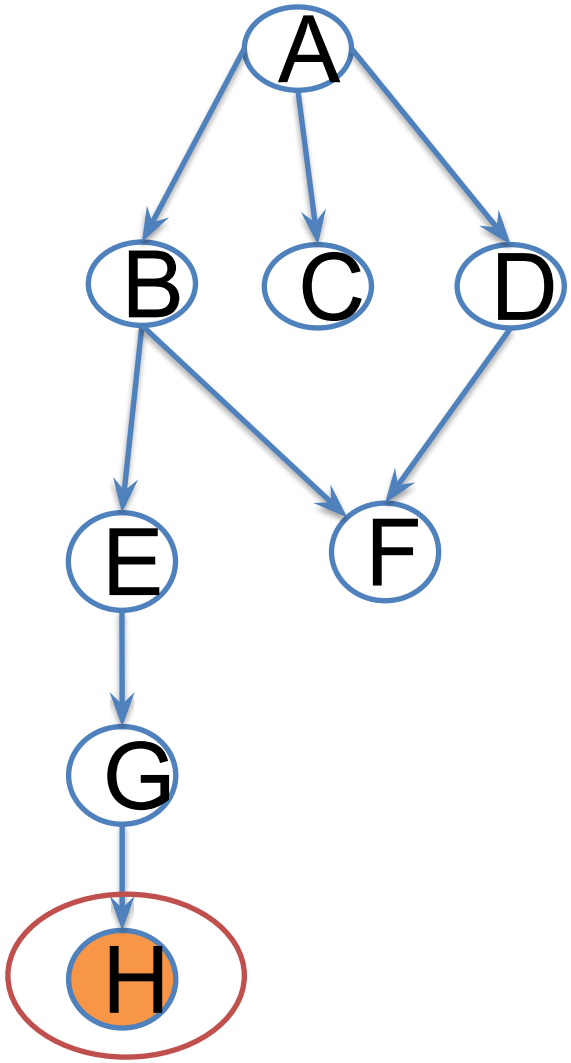
Output: A B C D E F

Recorrido en anchura (bfs)



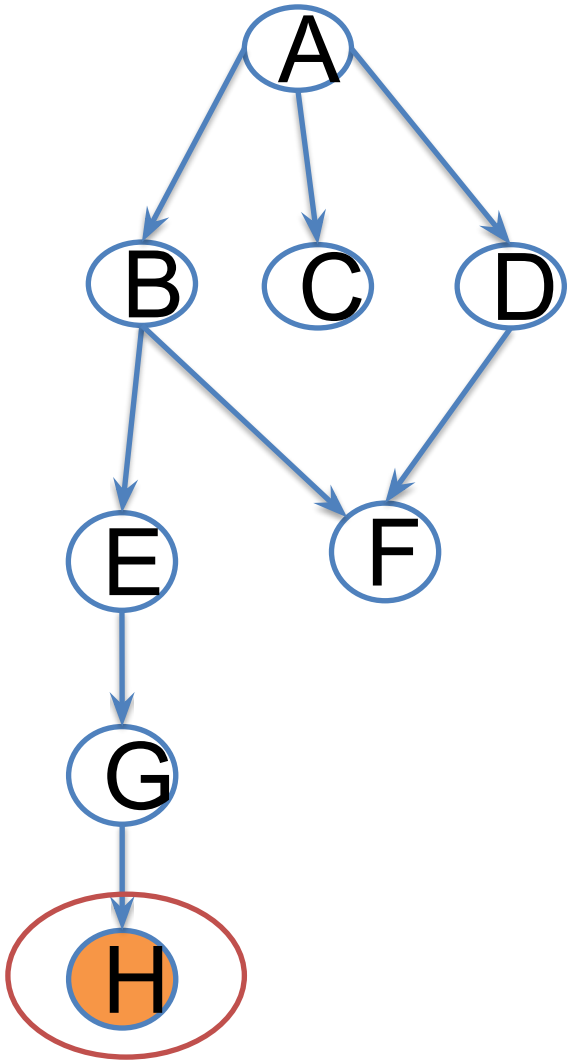
Output: A B C D E F G

Recorrido en anchura (bfs)



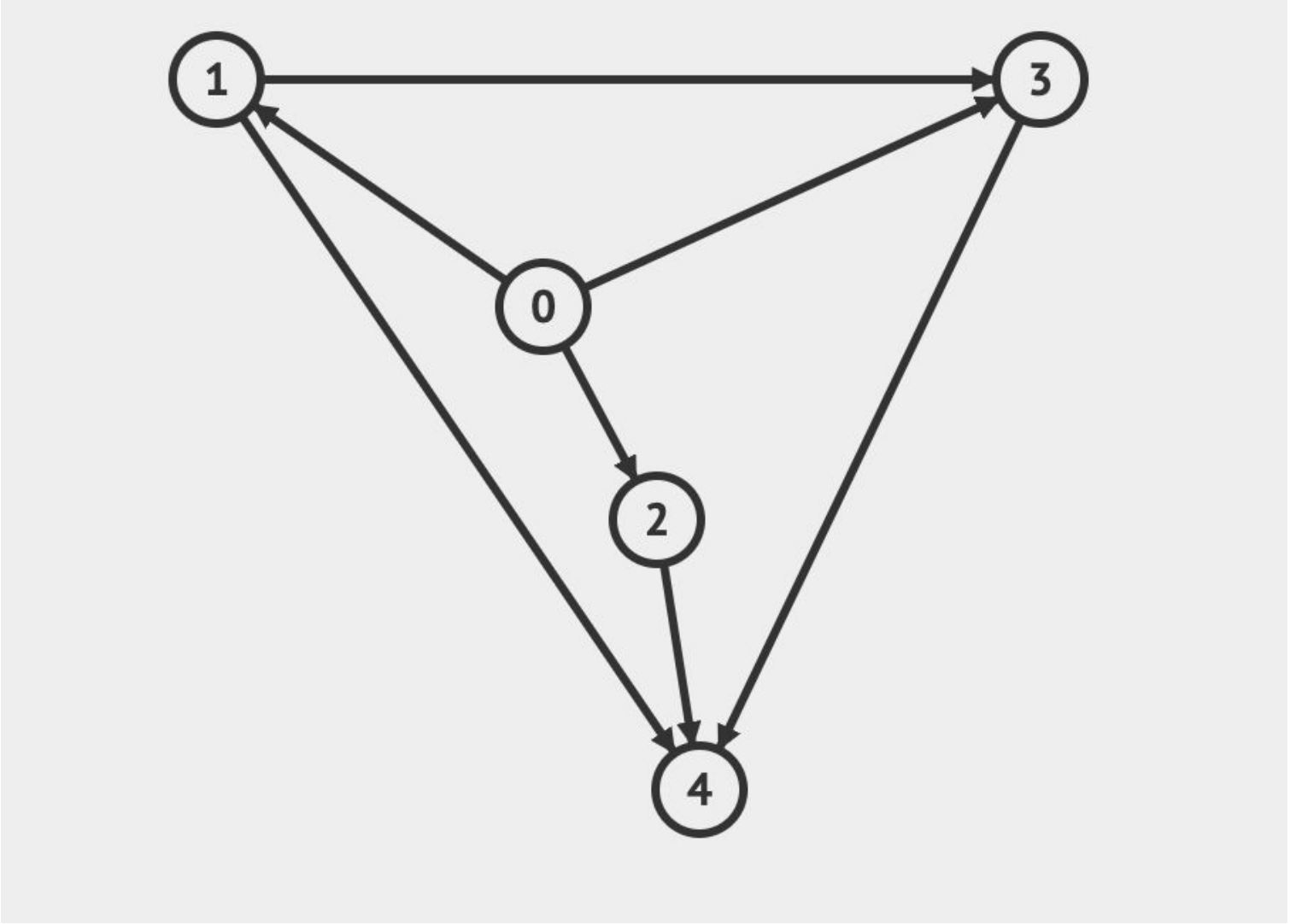
Output: A B C D E F G H

Recorrido en anchura (bfs)



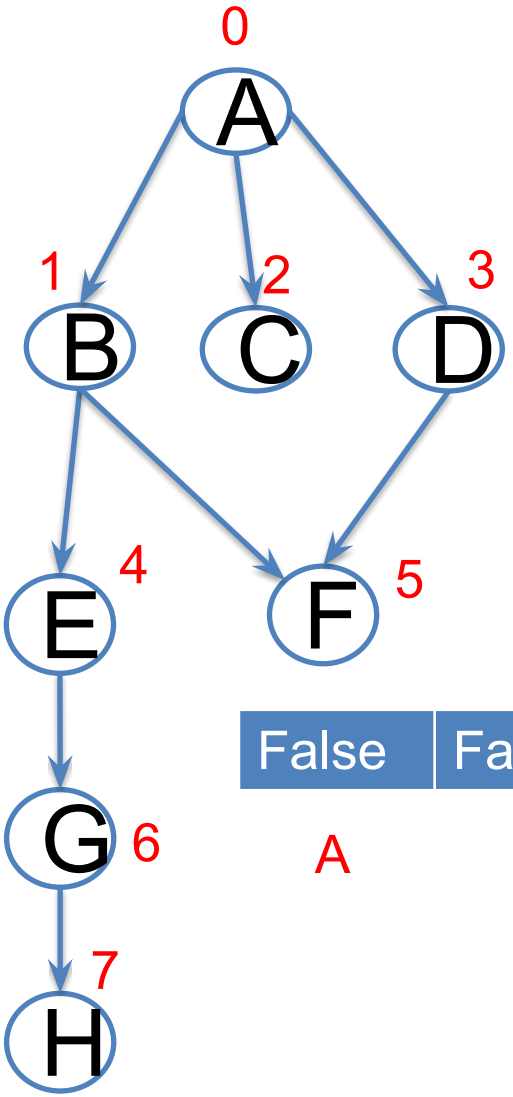
Output: A B C D E F G H

Recorrido en anchura (bfs)



https://visualgo.net/en/dfsbfs

Recorrido en anchura (implementación)

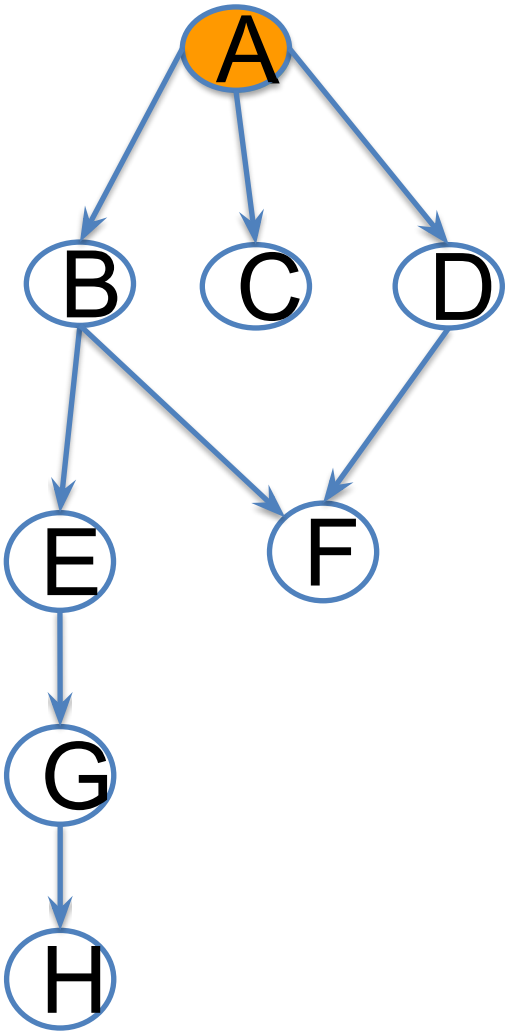


- Queue
- Lista (o diccionario) de booleanos para indicar que nodos ya han sido visitados

q

False	False	False	False	False	False	False	False
A	B	C	D	E	F	G	H

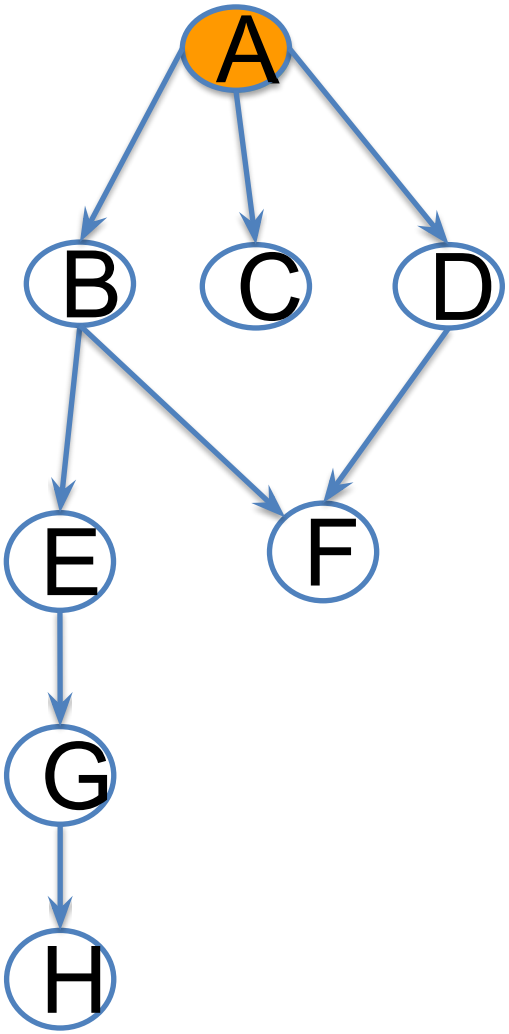
Recorrido en anchura (implementación)



Empezamos, por ejemplo, por el nodo A y lo añadimos en la cola

q | A |

Recorrido en anchura (implementación)

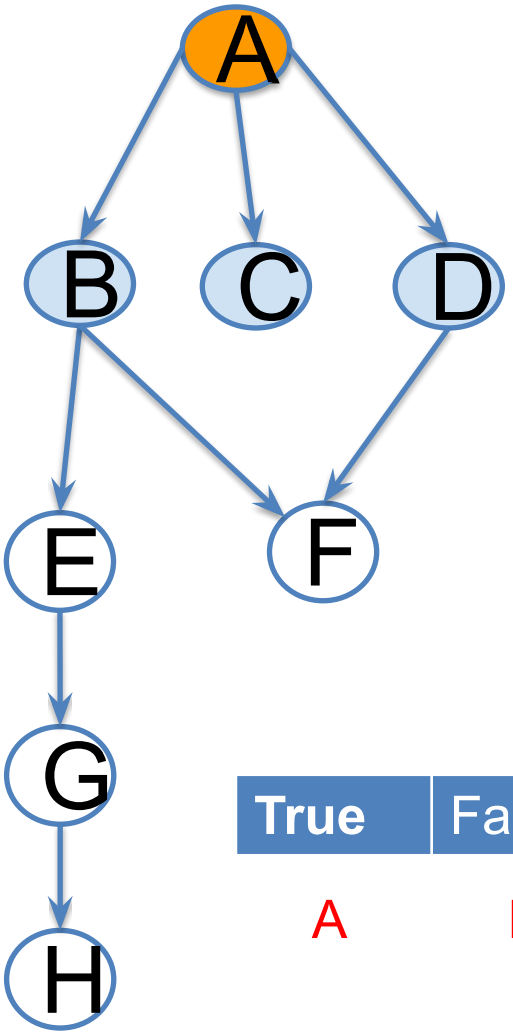


Mientras la cola no está vacía, hacer:

1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q | A |

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

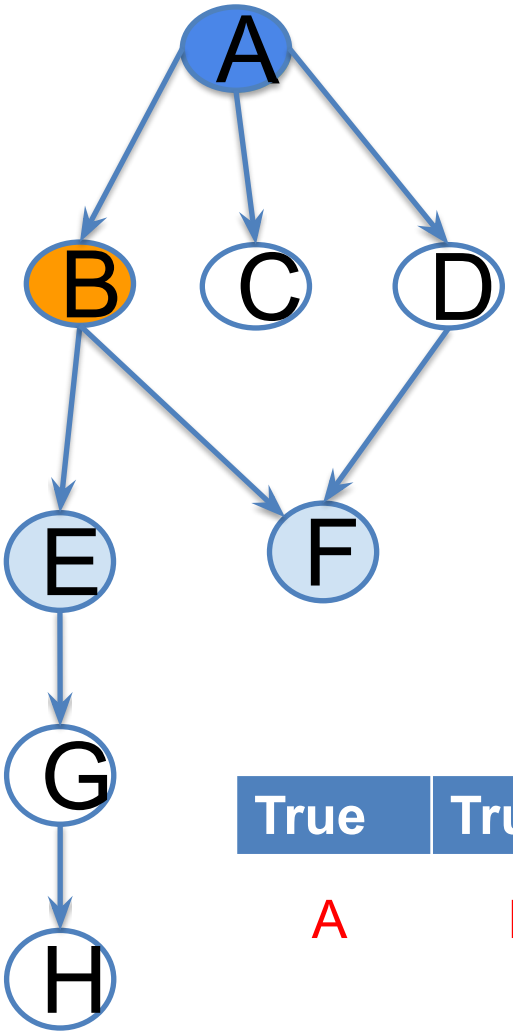
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q B C D

True	False	False	False	False	False	False	False
A	B	C	D	E	F	G	H

Output: A

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

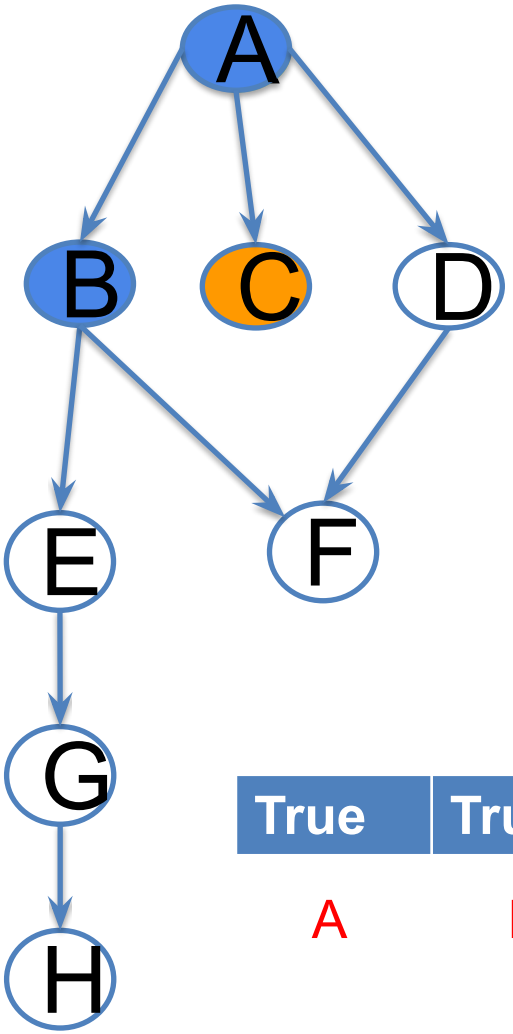
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q C D E F

True	True	False	False	False	False	False	False
A	B	C	D	E	F	G	H

Output: A B

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

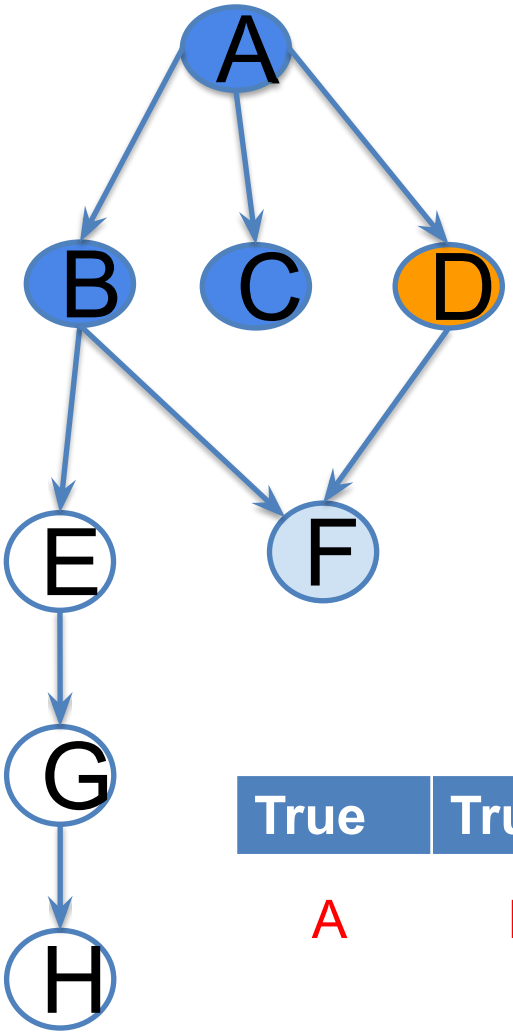
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q D E F

True	True	True	False	False	False	False	False
A	B	C	D	E	F	G	H

Output: A B C

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

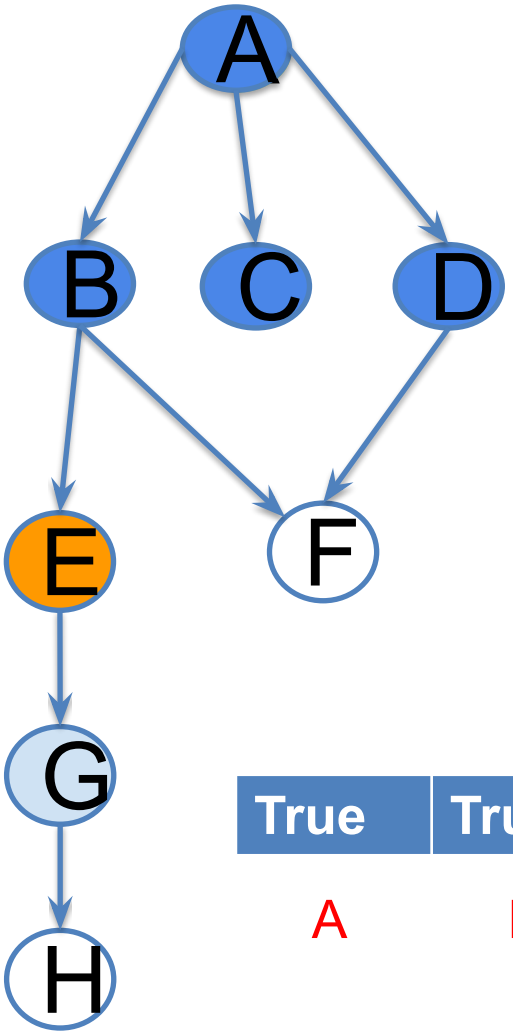
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q E F

True	True	True	True	False	False	False	False
A	B	C	D	E	F	G	H

Output: A B C D

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

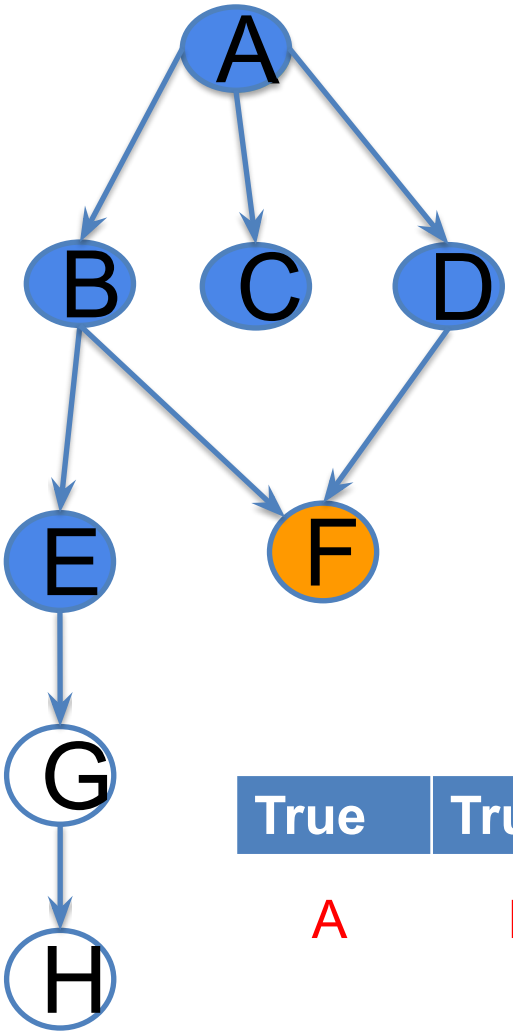
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q F G

True	True	True	True	True	False	False	False
A	B	C	D	E	F	G	H

Output: A B C D E

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

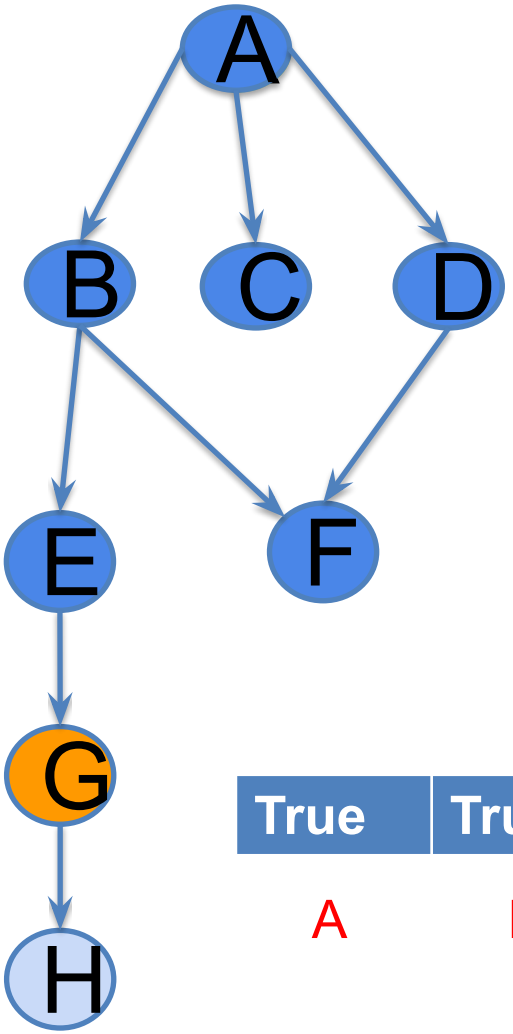
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q G

True	True	True	True	True	True	False	False
A	B	C	D	E	F	G	H

Output: A B C D E F

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

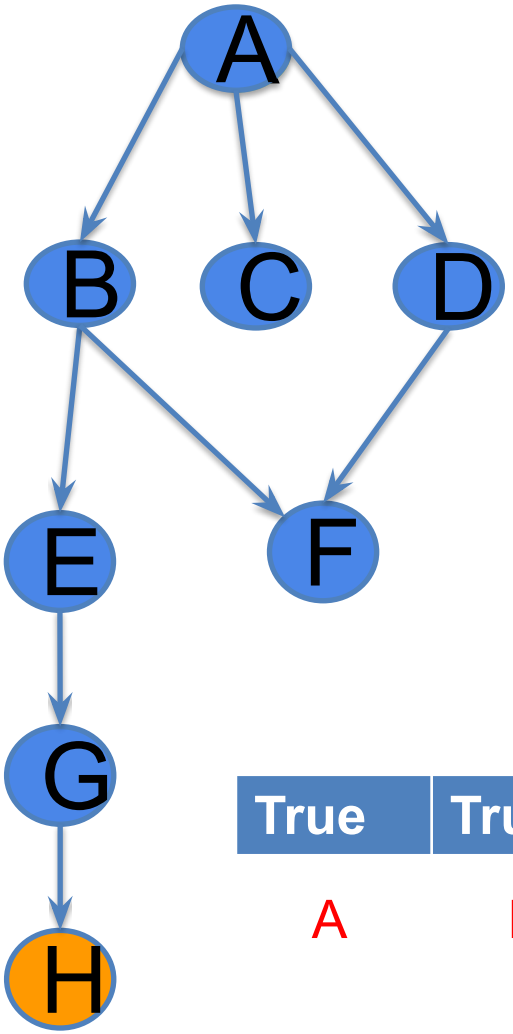
1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

q | H |

True	True	True	True	True	True	True	False
A	B	C	D	E	F	G	H

Output: A B C D E F G

Recorrido en anchura (implementación)



Mientras la cola no está vacía, hacer:

1. Sacamos el primer elemento de la cola
2. Imprimimos y marcamos como visitado
3. Obtenemos sus nodos vecinos y los añadimos a la cola

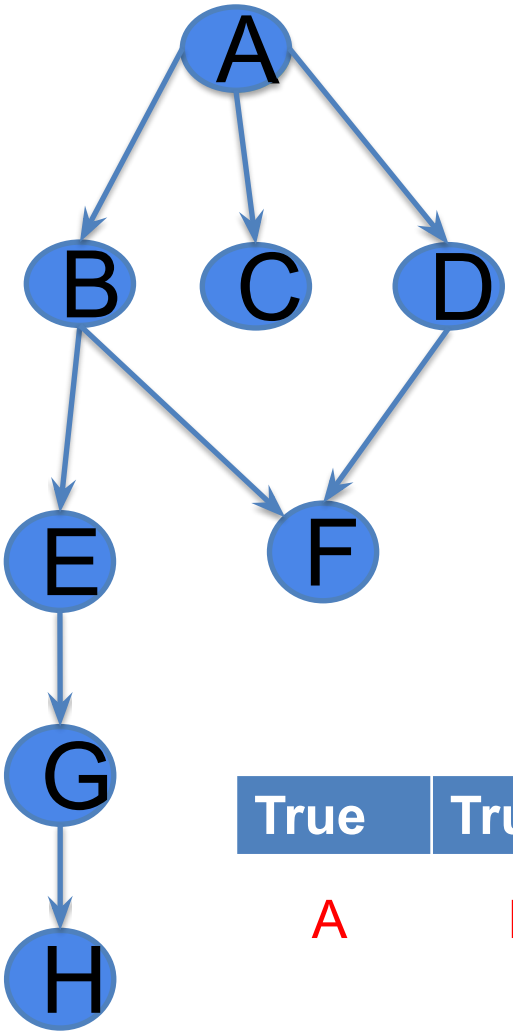
q _____

_____

True	True	True	True	True	True	True	True
A	B	C	D	E	F	G	H

Output: A B C D E F G H

Recorrido en anchura (implementación)



La cola está vacía y además todos los vértices ya han sido visitados!!!

q _____

_____

True	True	True	True	True	True	True	True
A	B	C	D	E	F	G	H

Output: A B C D E F G H

Complejidad Espacial - Recorrido en anchura

- Recordamos que la complejidad espacial de un grafo es: **|V| + |A|,** siendo |V|=n, número de vértices
  - Implementación basada en matriz: |V| + |A|= $n + n^2$
  - Implementación basada en lista de adyacencia o diccionarios*:
    - Si el grafo es denso |A| ≈ n2: |V| + |A|≈ $n + n^2$
    - Si el grafo es escaso |A| ≈ n: |V| + |A| ≈ n + n

(*) La complejidad espacial de un grafo basado en diccionario será el número de vértices (keys) + la suma del tamaño de todas las listas de adyacencia, que es igual, al número total de aristas. Es decir, |V| + |A|

Complejidad Espacial - Recorrido en anchura

- Por tanto, implícitamente todas las funciones ya tendrán una complejidad espacial de **|V| + |A|.**
- Si el grafo es denso o está implementado con matrices, la complejidad espacial requerida será de orden cuadrático.
- Si la implementación no está basada en matrices, y además, el grafo es escaso, la complejidad espacial será de orden n.

(*) La complejidad espacial de un grafo basado en diccionario será el número de vértices (keys) + la suma del tamaño de todas las listas de adyacencia, que es igual, al número total de aristas. Es decir, |V| + |A|

Complejidad Espacial - Recorrido en anchura

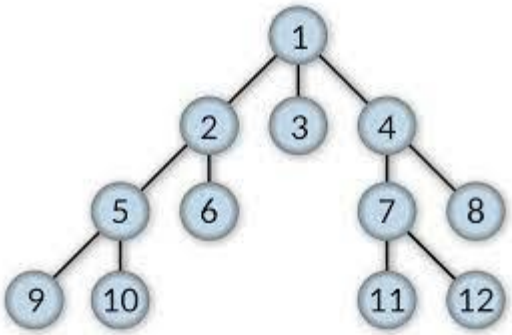
- Si nos limitamos a considerar únicamente la complejidad espacial de las estructura auxiliares que usa la función bfs:
  - Cola q: n, como máximo número de vértices
  - Diccionario visited: diccionario de n keys y con n booleanos asociados, es decir, este diccionario es como tener n tuplas (key, value). Su complejidad espacial es 2n
- Por tanto, la complejidad espacial (extra) del método bfs será de orden n.

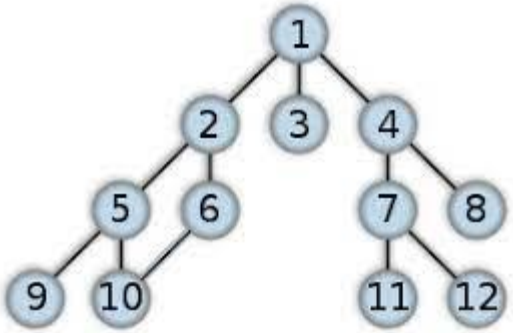
Complejidad Temporal - Recorrido en anchura

- La complejidad temporal del recorrido en anchura será O(|V| + |A|), porque en el peor caso será necesario recorrer todos los posibles caminos a todos los vértices.

Complejidad Temporal - Recorrido en anchura

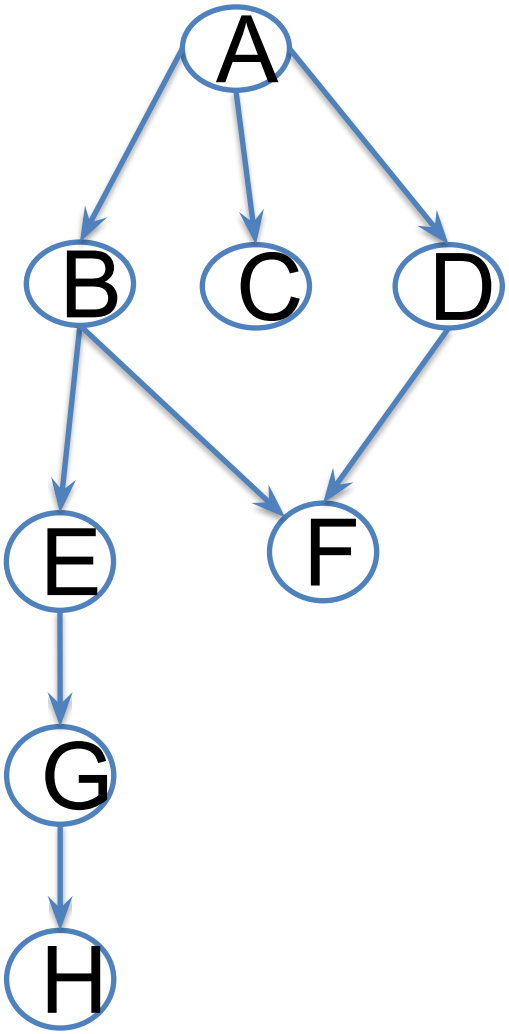
- Por ejemplo, en el grafo de la izquierda, es necesario recorrer todos los nodos adyacentes a uno dado.
- Sin embargo, en el grafo de la derecha, una vez que has visitado 10 (como hijo adyacente de 5), no es necesario volver a recorrerlo como adyacente de 6.





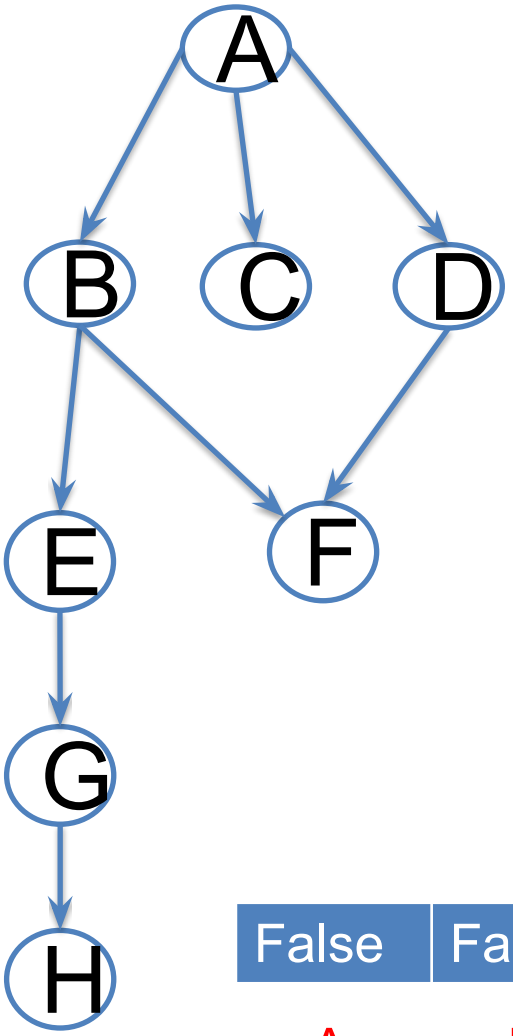
Source. https://www.techiedelight.com/check-undirected-graph-contains-cycle-not/

Recorrido en profundidad (dfs)



1) Recorrido en amplitud: Breadth-first traversal (BFS)
2) **Recorrido en profundidad: Depth-first traversal (DFS)**

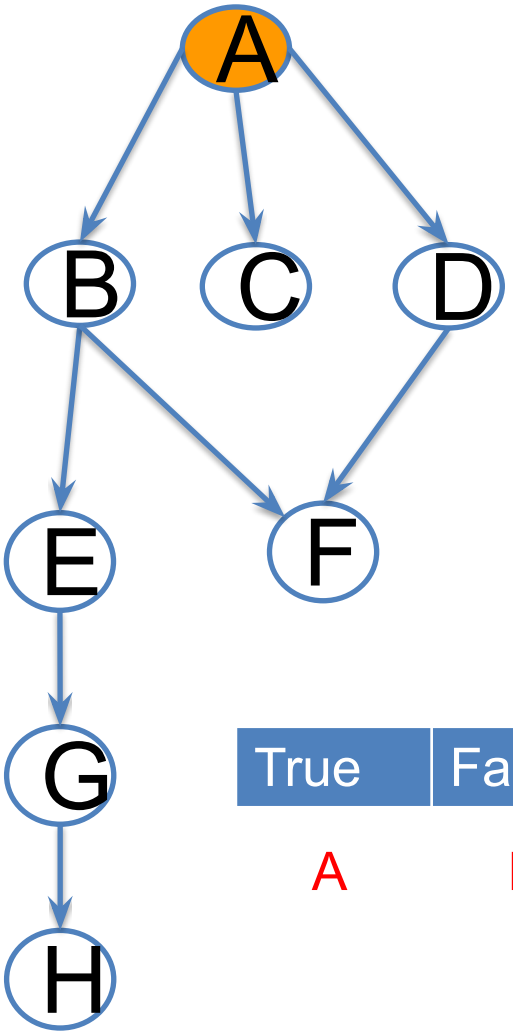
Recorrido en profundidad (dfs)



Idea: Partimos de un vértice y continuamos por una de sus ramas hasta que sea posible.
Cuando no sea posible, retrocedemos hasta encontrar el primer vértice ya visitado con varios caminos, para continuar por alguno de ellos.
Utilizamos un diccionario (o una lista) para marcar los nodos visitados.
El algoritmo sigue mientras haya vértices por visitar.

False	False	False	False	False	False	False	False
A	B	C	D	E	F	G	H

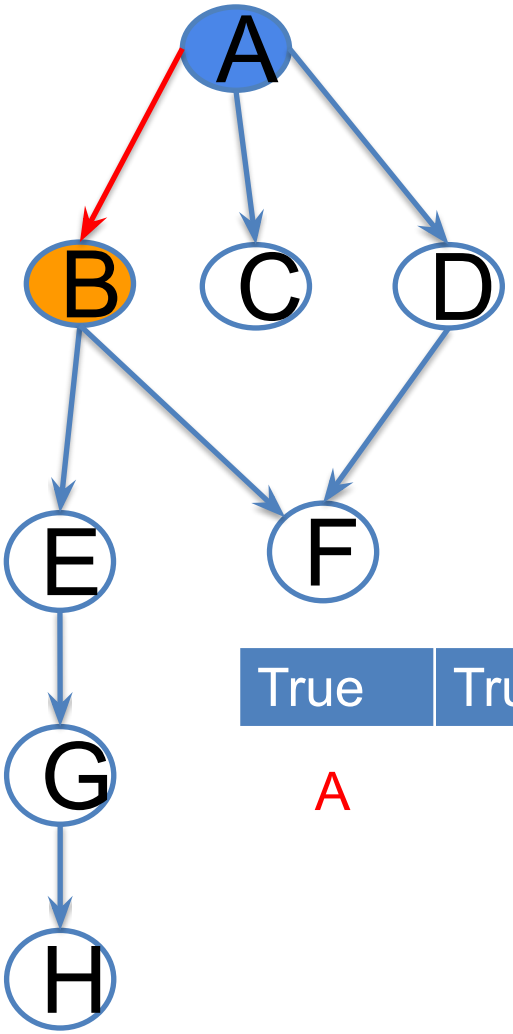
Recorrido en profundidad (dfs)



Output: A,

True	False	False	False	False	False	False	False
A	B	C	D	E	F	G	H

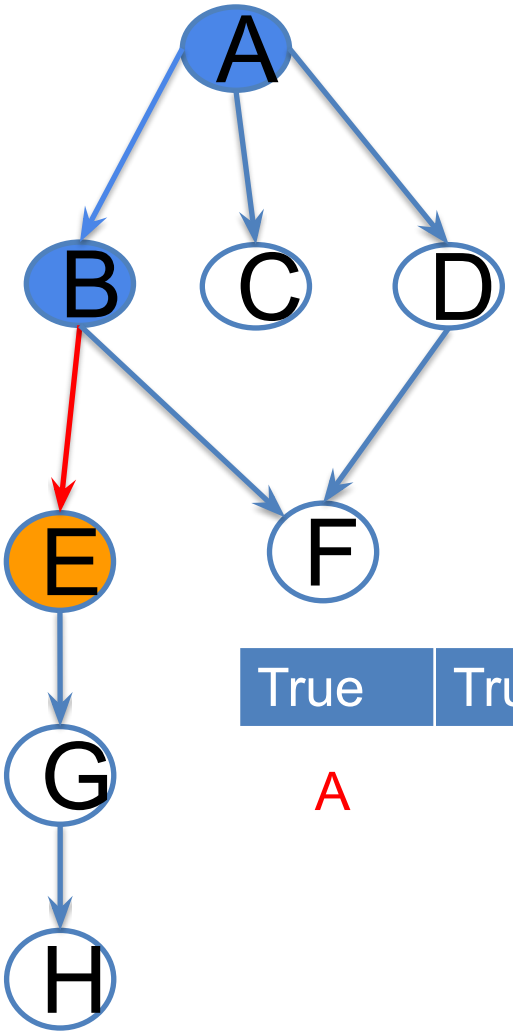
Recorrido en profundidad (dfs)



Output: A,B

True	True	False	False	False	False	False	False
A	B	C	D	E	F	G	H

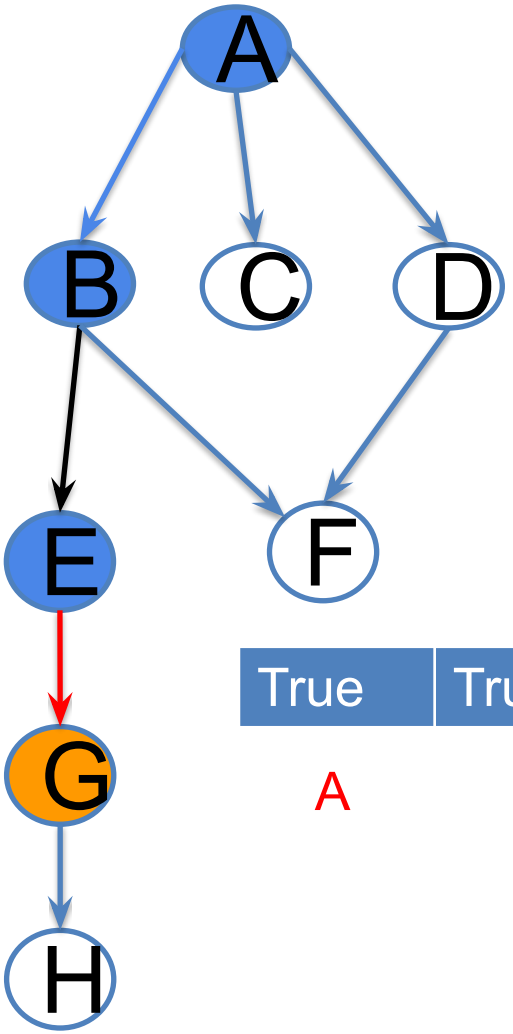
Recorrido en profundidad (dfs)



Output: A,B,E

True	True	False	False	**True**	False	False	False
A	B	C	D	E	F	G	H

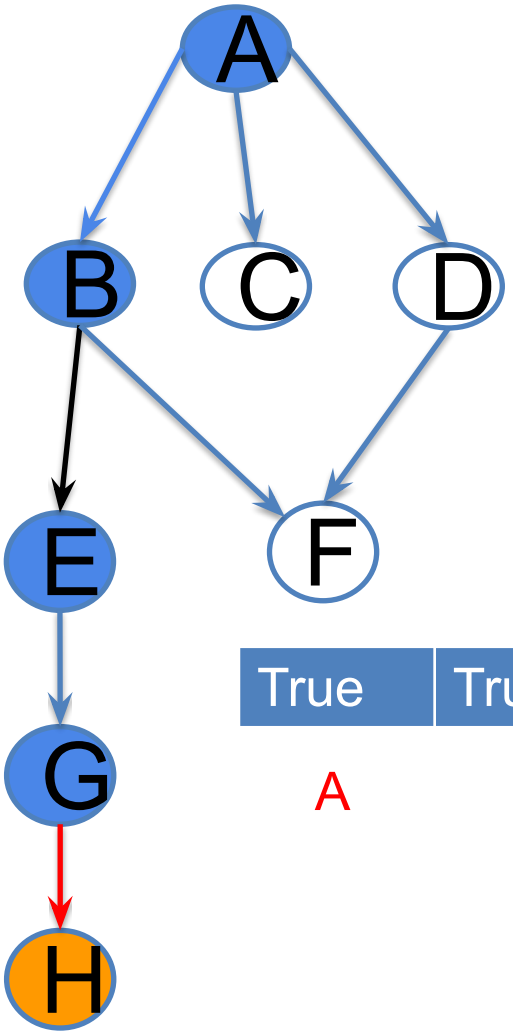
Recorrido en profundidad (dfs)



Output: A,B,E,G

True	True	False	False	True	False	**True**	False
A	B	C	D	E	F	G	H

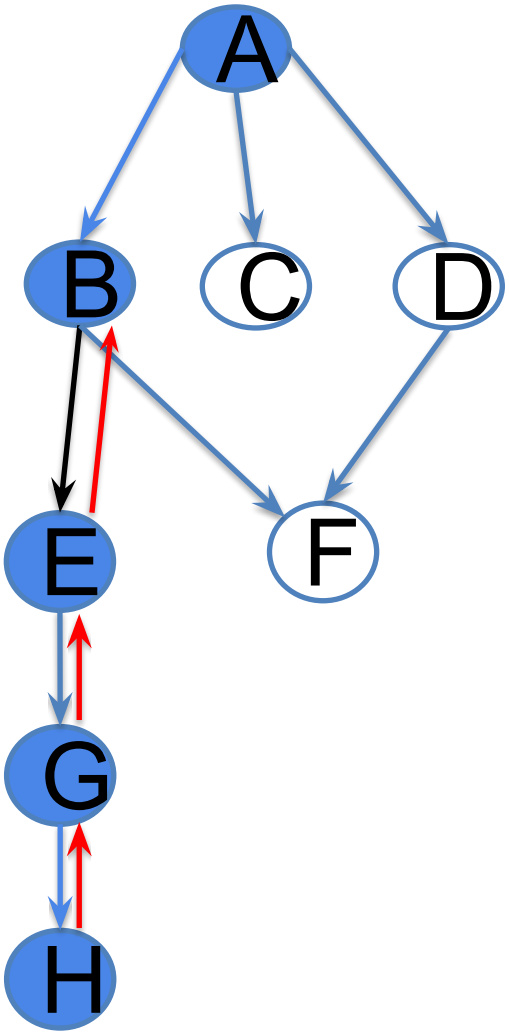
Recorrido en profundidad (dfs)



Output: A,B,E,G,H

True	True	False	False	True	False	True	**True**
A	B	C	D	E	F	G	H

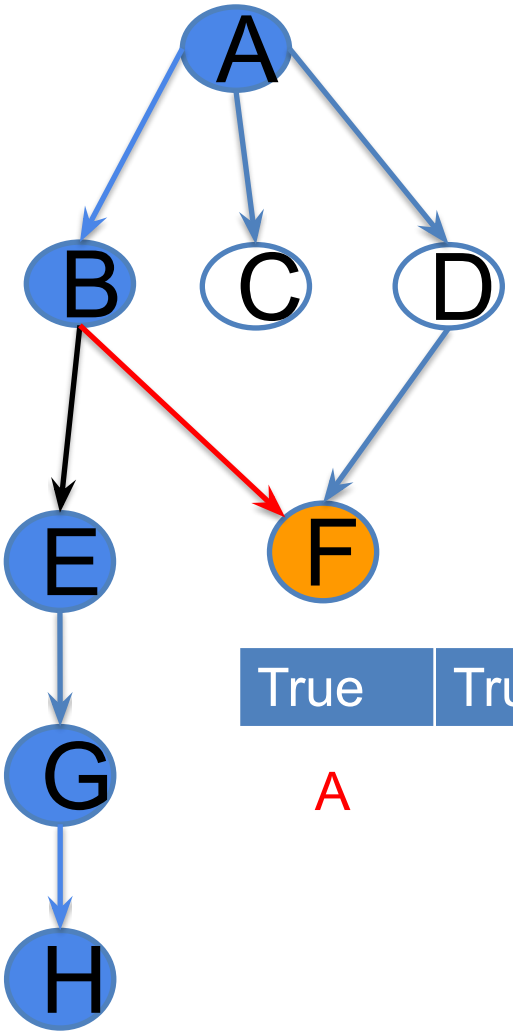
Recorrido en profundidad (dfs)



Output: A,B,E,G,H

En H, no podemos continuar, así que retrocedemos hasta encontrar otra rama alternativa donde podemos continuar

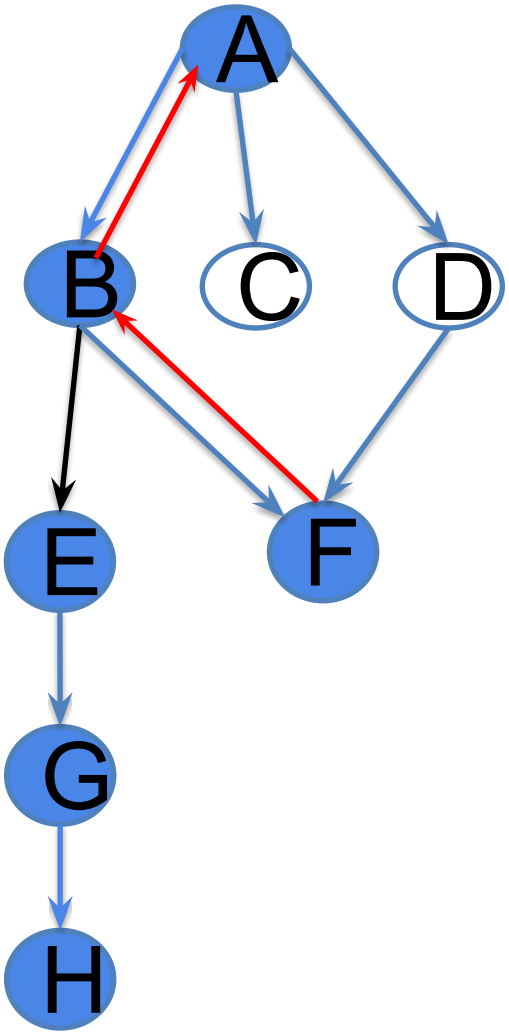
Recorrido en profundidad (dfs)



Output: A,B,E,G,H,F

True	True	False	False	True	**True**	True	True
A	B	C	D	E	F	G	H

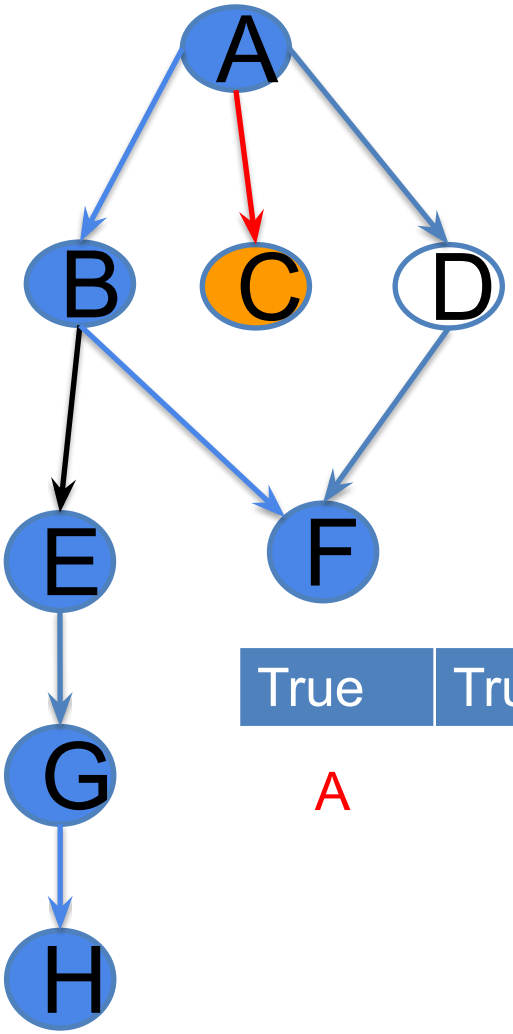
Recorrido en profundidad (dfs)



Output: A,B,E,G,H,F

En F, no podemos continuar, así que retrocedemos hasta encontrar un nodo ya visitado que tenga otras ramas que visitar.

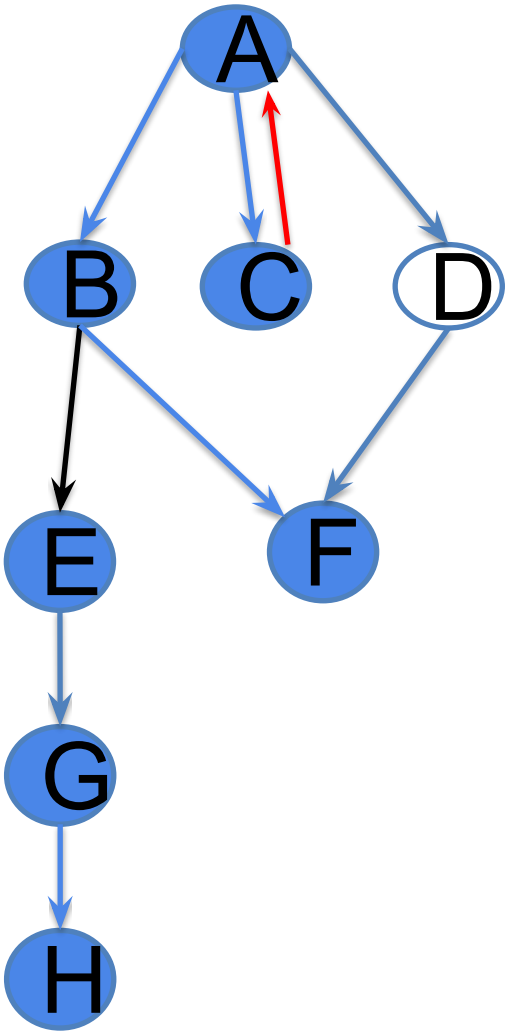
Recorrido en profundidad (dfs)



Output: A,B,E,G,H,F,C

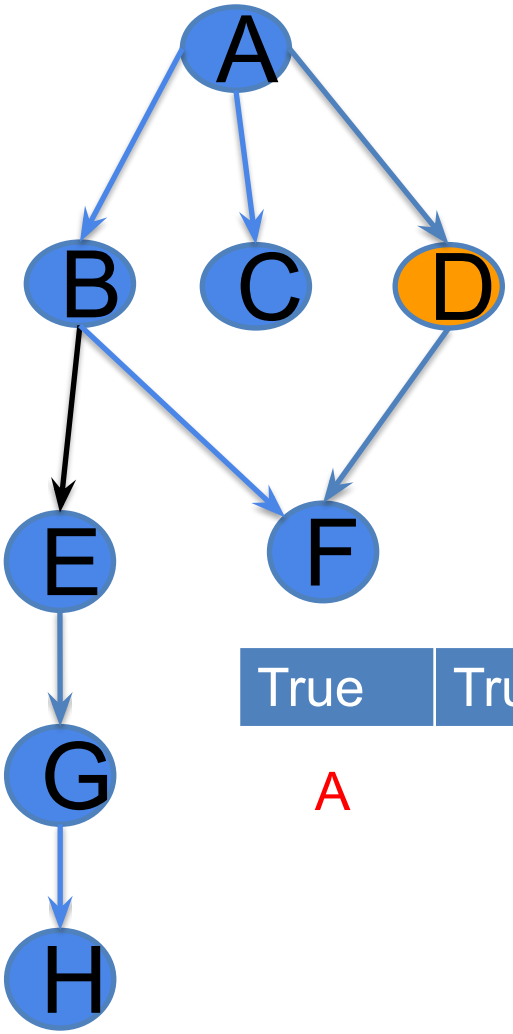
True	True	**True**	False	True	True	True	True
A	B	C	D	E	F	G	H

Recorrido en profundidad (dfs)



Output: A,B,E,G,H,F,C

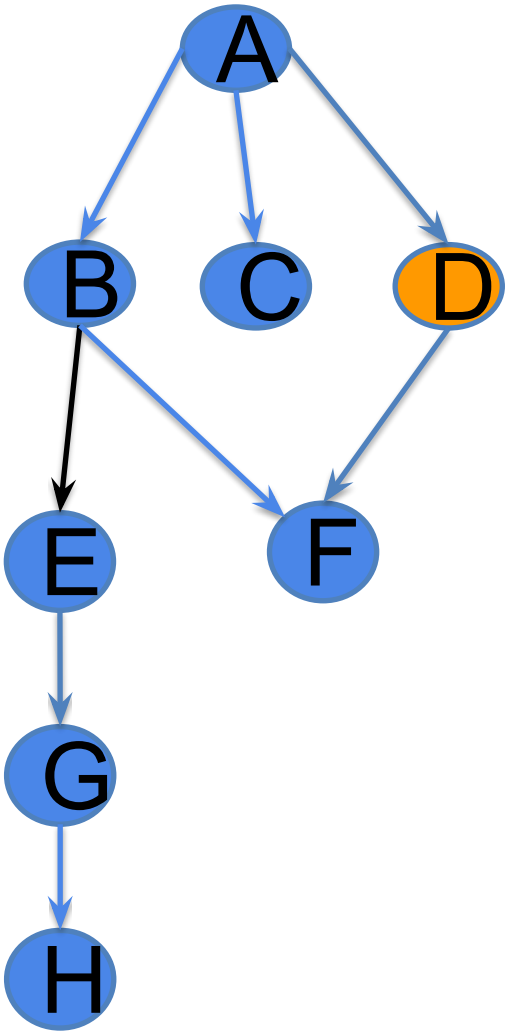
Recorrido en profundidad (dfs)



Output: A,B,E,G,H,F,C,D

True	True	True	**True**	True	**True**	True	True
A	B	C	D	E	F	G	H

Recorrido en profundidad (dfs)



Output: A,B,E,G,H,F,C,D

F es adyacente a D, pero ya ha sido visitado, así que no tenemos que volver a visitarlo.

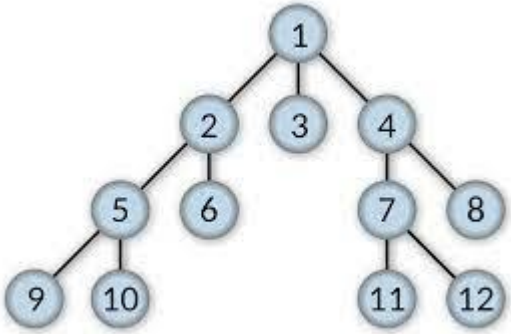
Terminamos porque ya no hay más caminos y todos los vértices han sido visitados.

Complejidad espacial - Recorrido en profundidad (dfs)

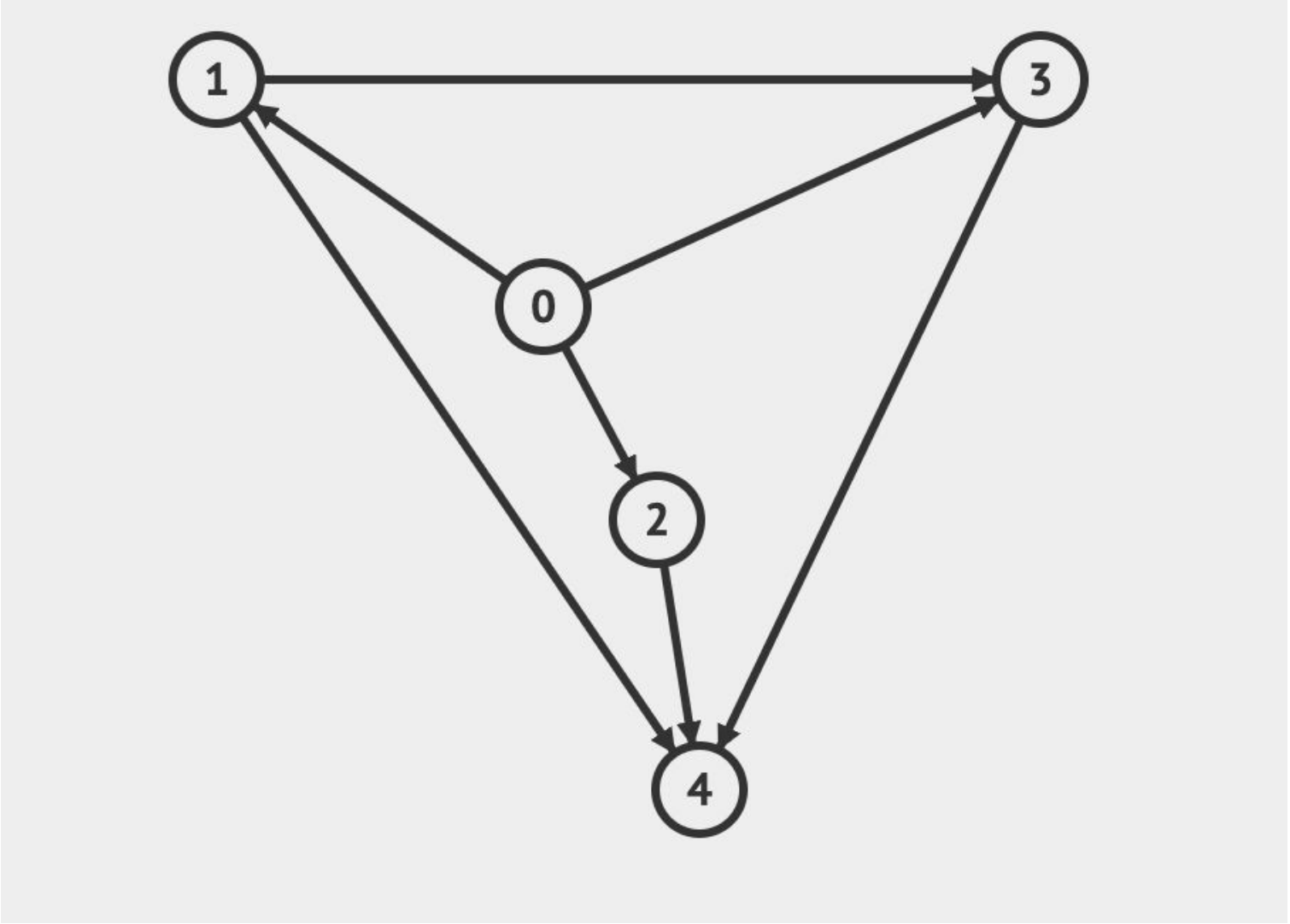
- Si nos limitamos a considerar únicamente la complejidad espacial de las estructura auxiliares que usa la función dfs:
  - Diccionario visited: diccionario de n keys y con n booleanos asociados, es decir, este diccionario es como tener n tuplas (key, value). Su complejidad espacial es 2n
- Por tanto, la complejidad espacial de dfs será menor que la complejidad espacial de breadth (cola + diccionario).

Complejidad espacial - Recorrido en profundidad (dfs)

- La complejidad temporal del recorrido en profundidad es O(|V| + |A|) porque en el peor caso para cada nodo será necesario visitar todos los caminos que parten de dicho nodo.
- Como ocurría en el recorrido en anchura, si el grafo es un árbol (es decir, no se forman ciclos), será necesario recorrer todos los posibles caminos del grafo.



Recorrido en profundidad (dfs)



https://visualgo.net/en/dfsbfs

Recorridos (implementación)

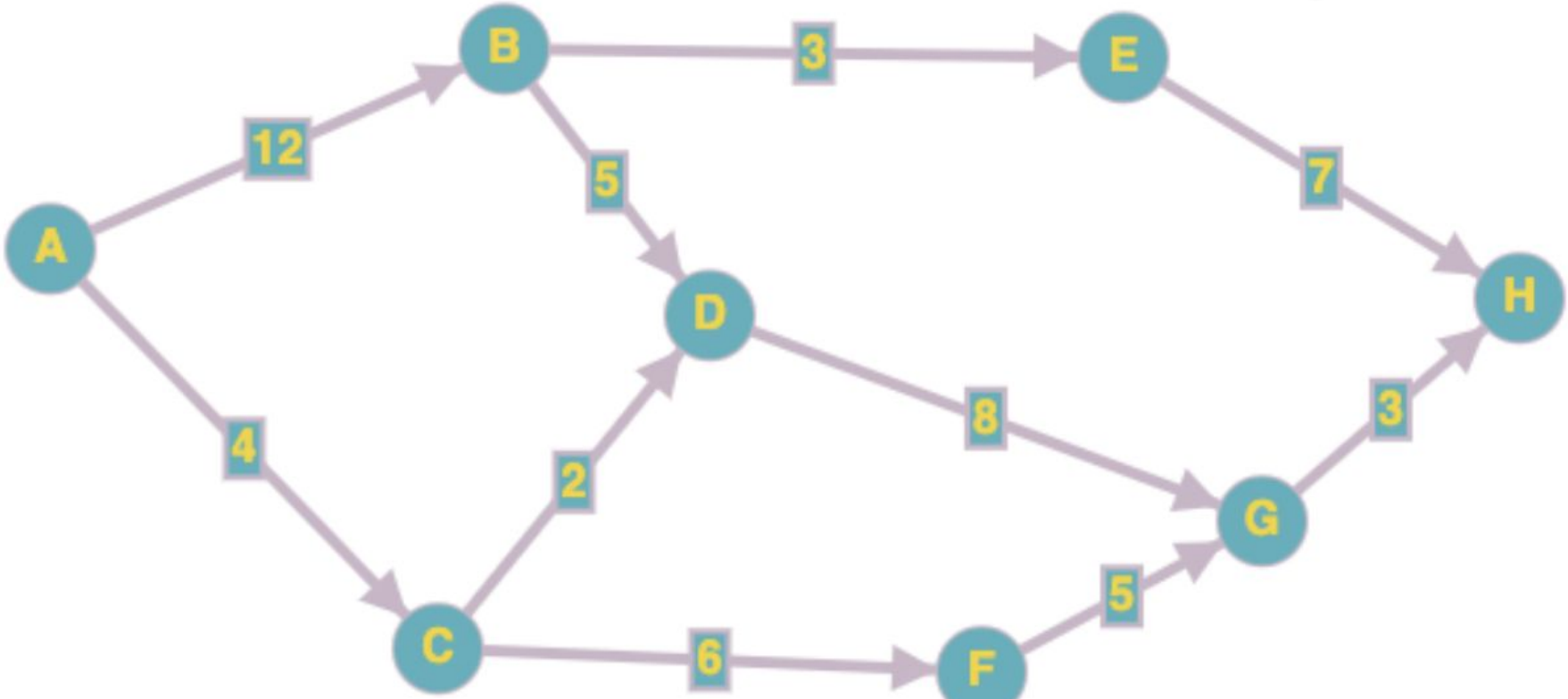
- Recorrido en anchura, métodos bfs y _bfs
- Recorrido en profundidad, métodos dfs y _dfs

Índice

- Introducción
- Conceptos sobre grafos
- TAD Grafo
- Implementaciones:
  - Matriz de adyacencia.
  - Lista de adyacencia.
  - Diccionarios (Python)
- Recorridos
- **Algoritmo de camino mínimo (Dijkstra).**

¿Cómo encontrar el camino más corto entre dos ciudades?

Por ejemplo, para ir de A a H

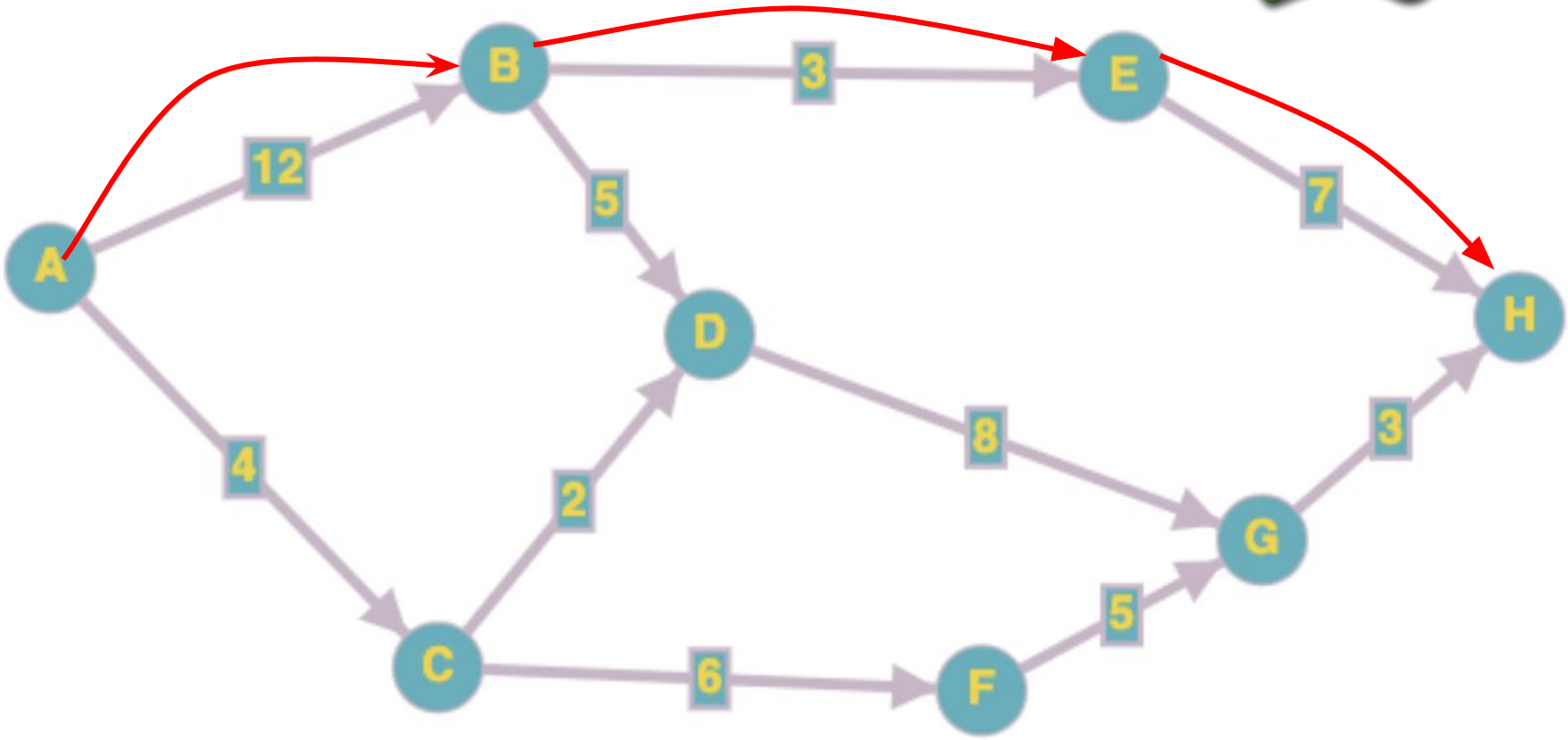


Creado con la herramienta online https://graphonline.ru/en/

http://graphonline.ru/en/?graph=SrfeylVWuybozZrc

¿Cómo encontrar el camino más corto entre dos ciudades?

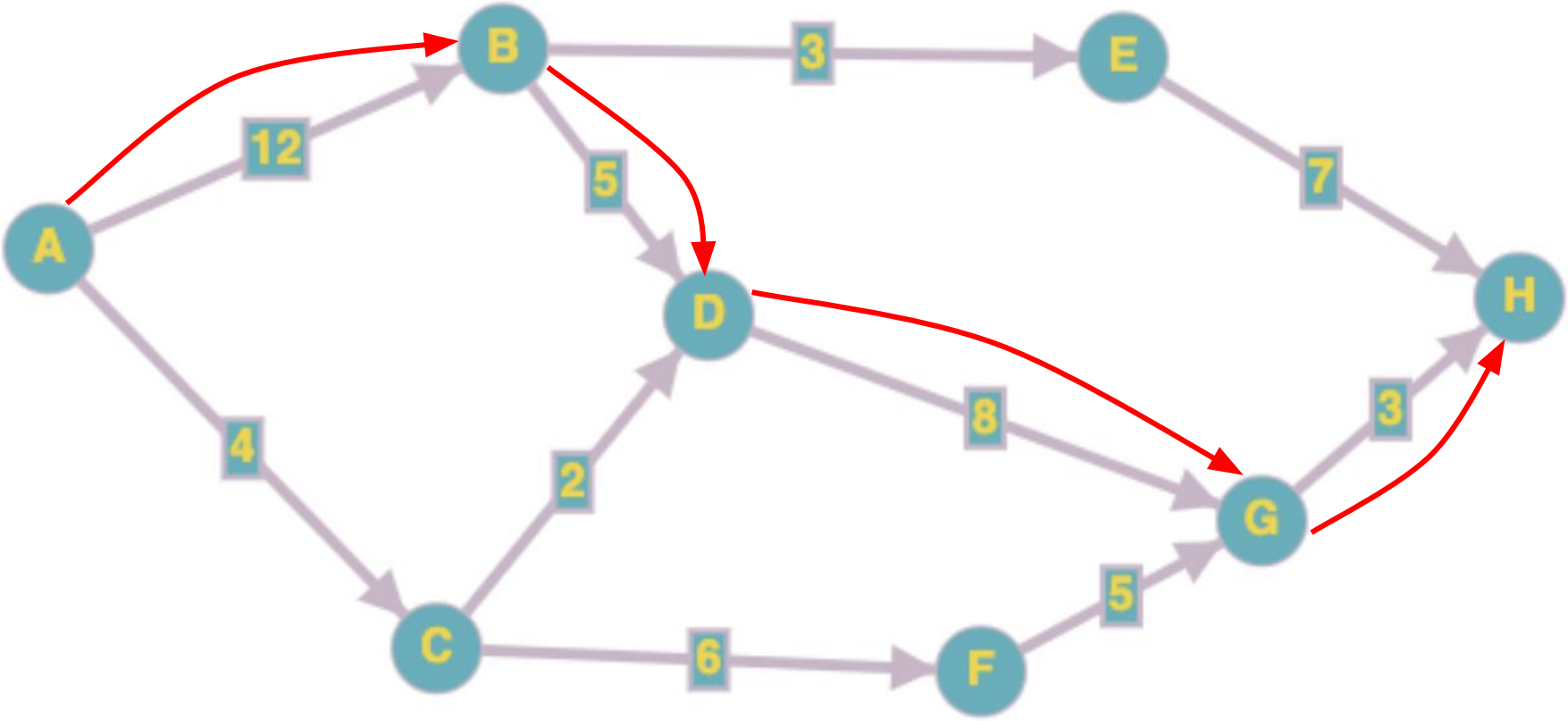
Ruta 1: A->B->E->H (22 km)



Creado con la herramienta online https://graphonline.ru/en/
http://graphonline.ru/en/?graph=SrfeyIVWuybozZrc

¿Cómo encontrar el camino más corto entre dos ciudades?

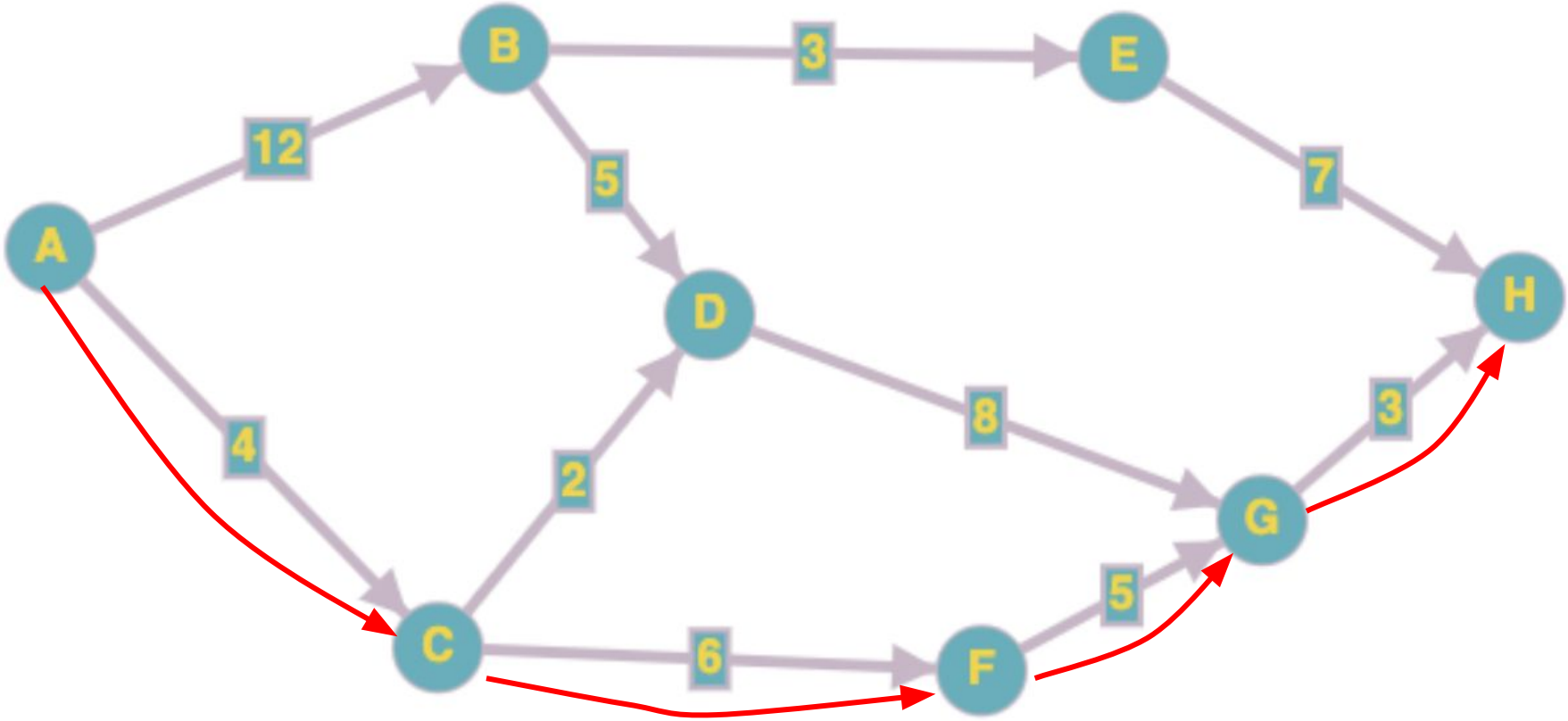
Ruta 2: A->B->D->G->H (28 KM)



Creado con la herramienta online https://graphonline.ru/en/
http://graphonline.ru/en/?graph=SrfeyIVWuybozZrc

¿Cómo encontrar el camino más corto entre dos ciudades?

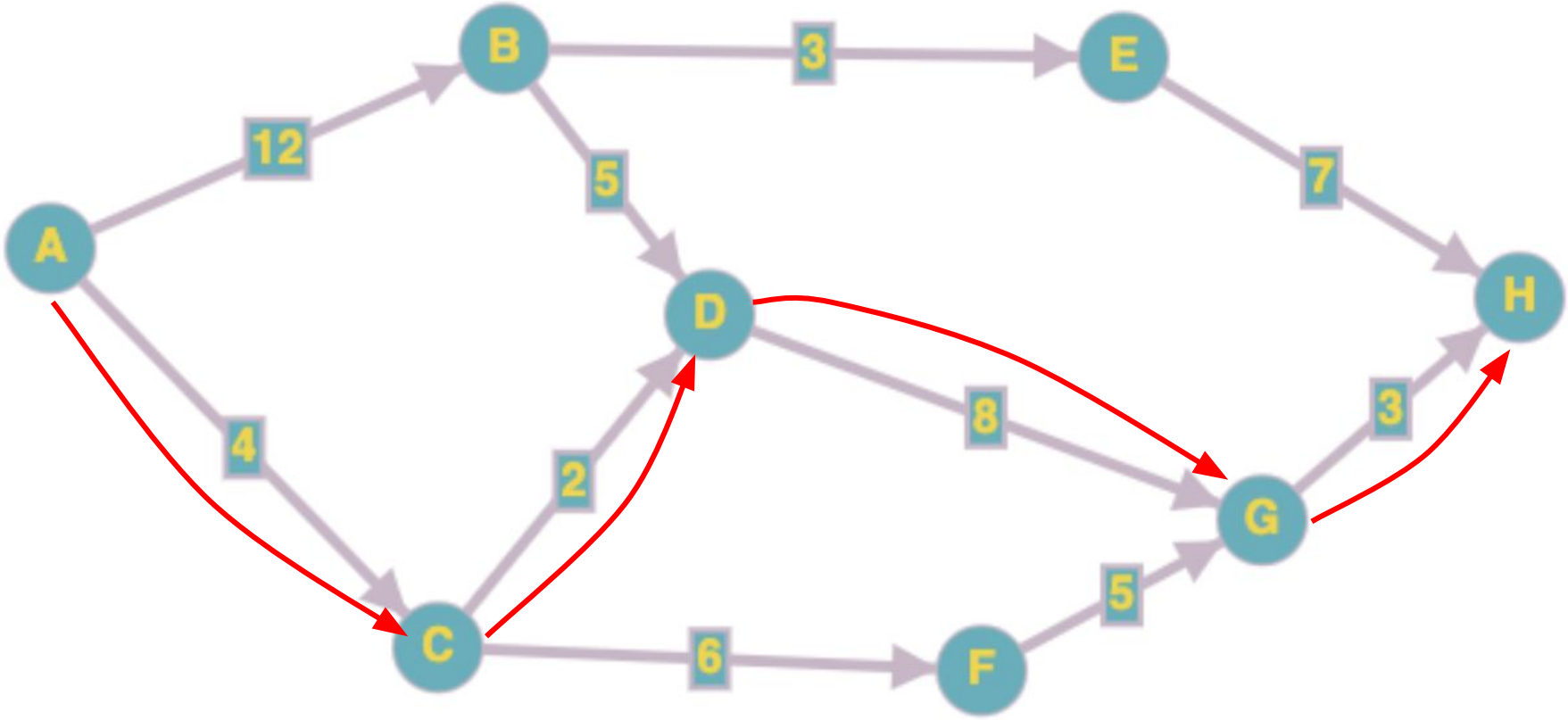
Ruta 3: A->C->F->G->H (18 km)



Creado con la herramienta online https://graphonline.ru/en/
http://graphonline.ru/en/?graph=SrfeyIVWuybozZrc

¿Cómo encontrar el camino más corto entre dos ciudades?

Ruta 4: A->C->D->G->H (17 km)



Creado con la herramienta online https://graphonline.ru/en/
http://graphonline.ru/en/?graph=SrfeyIVWuybozZrc

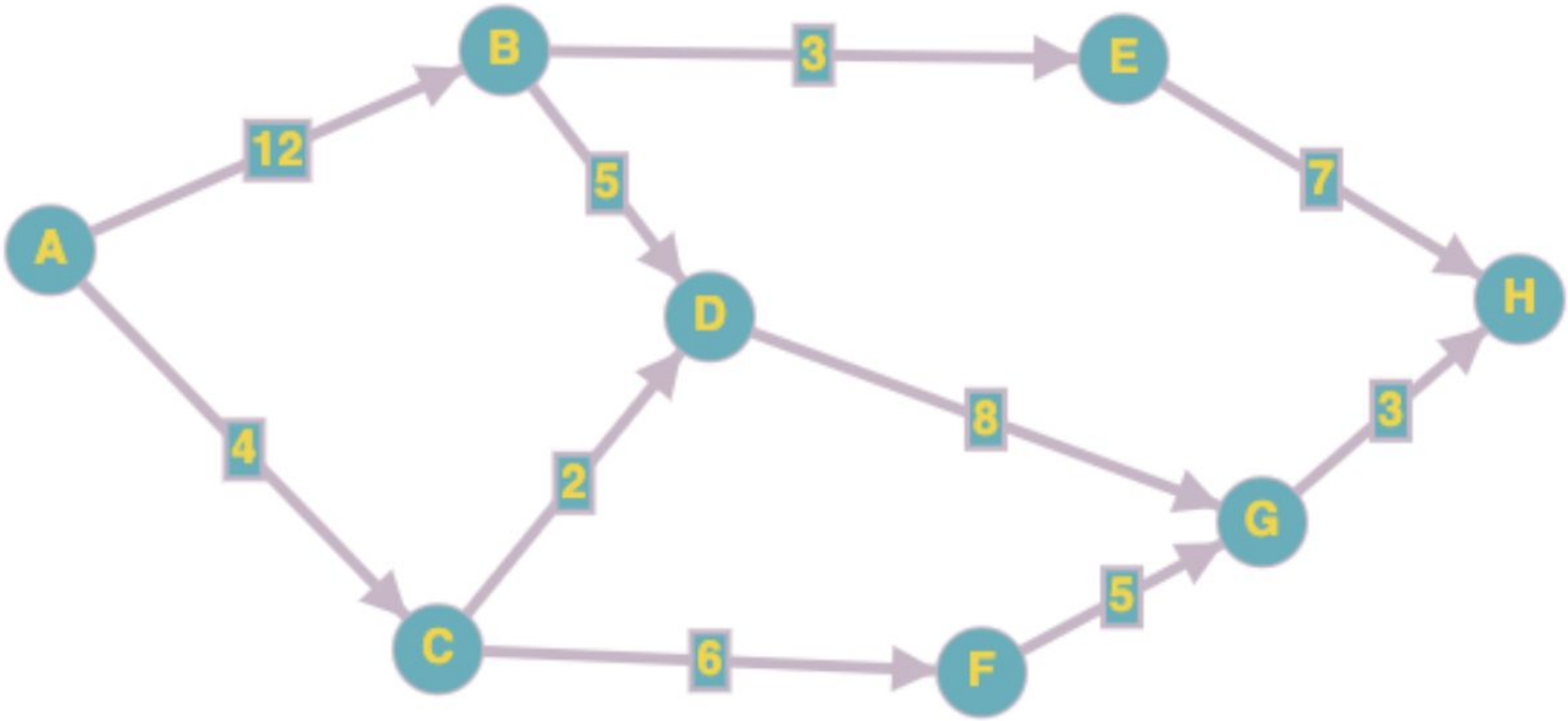
¿Cómo encontrar el camino más corto entre dos ciudades?

El problema se complica cuando tengo muchas ciudades (vértices) y carreteras (aristas)

Algoritmo de Dijkstra

- También conocido como algoritmo de camino mínimo.
- Permite obtener el camino más corto entre un vértice origen y el resto de vértices en el grafo.

Algoritmo de Dijkstra

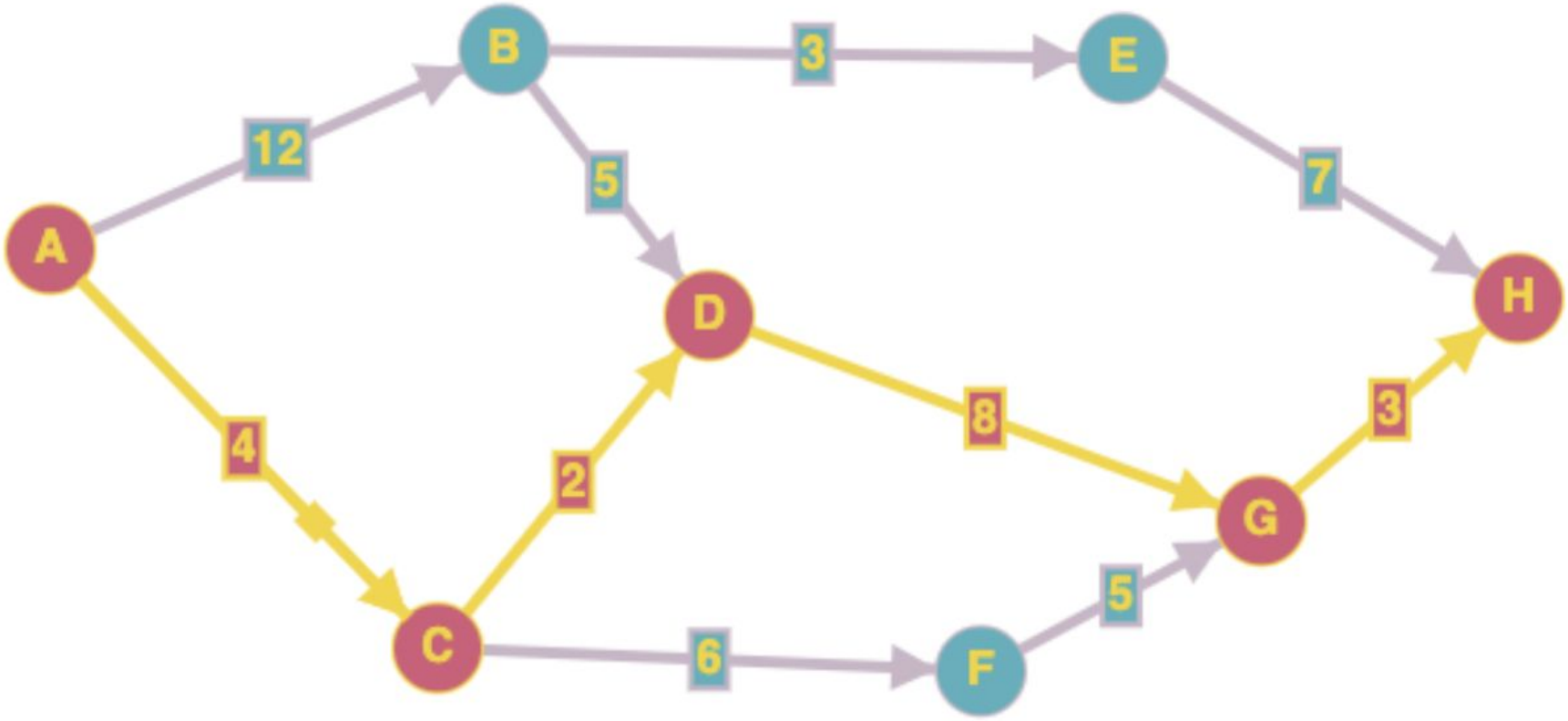


Creado con la herramienta online https://graphonline.ru/en/

http://graphonline.ru/en/?graph=SrfeyIVWuybozZrc

Algoritmo de Dijkstra

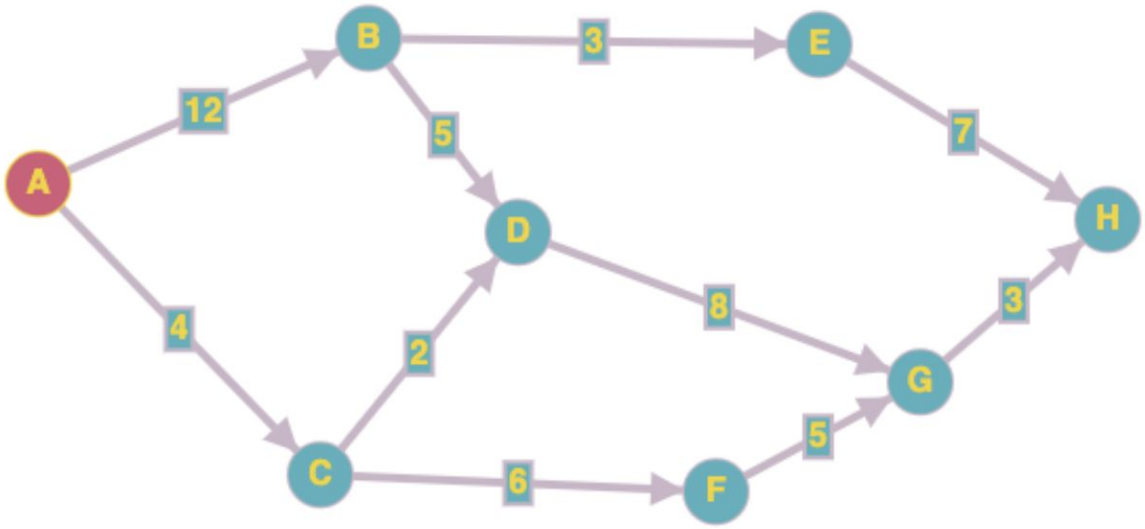
Camino mínimo de A a H:



Creado con la herramienta online https://graphonline.ru/en/

Algoritmo de Dijkstra - Resultado esperado

Nos permite calcular el camino más corto desde A al resto de vértices.



	Distancia acumuluda del camino más corto desde A	Vértice anterior
A	0	-
B	12	A
C	4	A
D	6	C
E	15	B
F	10	C
G	14	D
H	17	G

El vértice anterior nos permite reconstruir el camino de un vértice a otro.

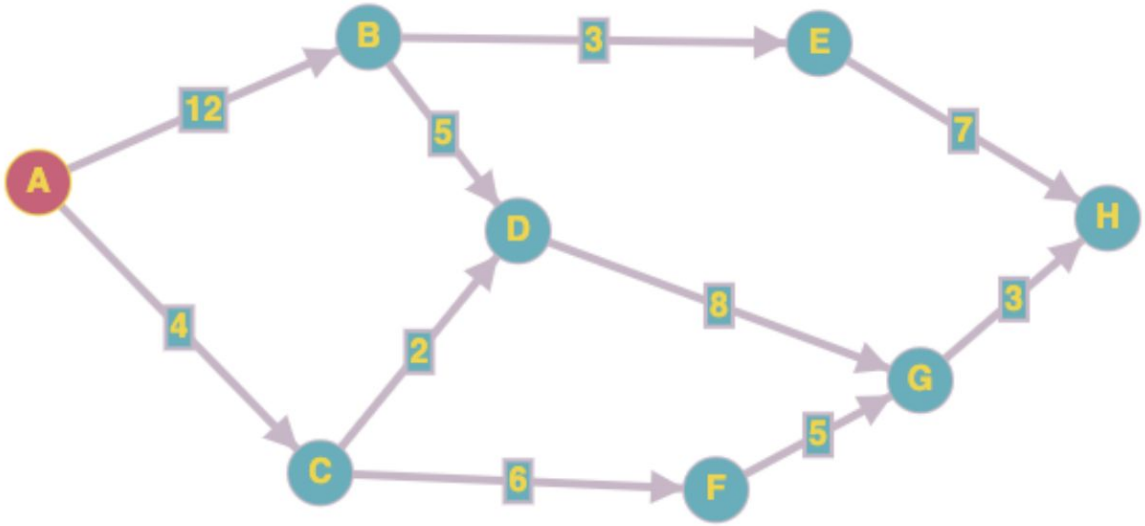
Algoritmo de Dijkstra - Inicialización

```
def dijkstra(self, origin: object) -> None:
    visited = {}    # for each vertex (key), the value is a boolean indicating if the vertex has been visited
    previous = {}   # for each vertex (key), the value is the previous node in the minimum path from origin
    distances = {}  # for each vertex (key), the value is minimum distance in the minimum path from origin

    # initialize dictionaries
    for v in self._vertices.keys():
        visited[v] = False
        previous[v] = None
        distances[v] = math.inf

    # The distance from origin to itself is 0
    distances[origin] = 0
    ...
```

En primer lugar, debemos crear tres diccionarios: visited, previous, distances e inicializarlos. Además, también marcamos distancia del vértice origen a 0.

Algoritmo de Dijkstra - Cómo?

Elegimos el vértice origen (A) e inicializamos las distancias a otros vértices como ∞. La distancia A a A se considera 0.



	Longitud más corto desde A	Vértice anterior
A	**0**	None
B	∞	None
C	∞	None
D	∞	None
E	∞	None
F	∞	None
G	∞	None
H	∞	None

visitados={}

Algoritmo Disjkstra- bucle principal

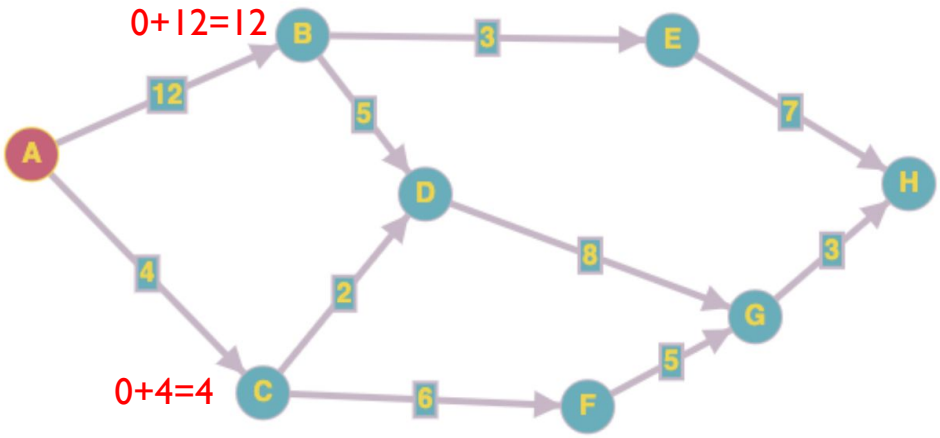
```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

En el siguiente bucle, iteramos n veces, siendo n el número de vértices.

En cada iteración vamos a obtener el vértice con menor distancia acumulada y lo vamos a visitar

Algoritmo de Dijkstra - Cómo?

Marcamos A como visitado y, obtenemos sus vértices adyacentes (B,C). Debemos calcular su distancia acumulada. También indicamos que el vértice anterior a esos dos vértices B y C, es A.



visitado[A]=True
Los adyacentes no visitados de A son {B, C}, ambos con distancia ∞, mayor que la que viene desde el vértice A. Por tanto, debemos actualizar la distancia de B y C, así como su nodo predecesor

	Longitud más corto desde A	Vértice anterior
A ✓	0	None
B	0+12	A
C	0+4	A
D	∞	None
E	∞	None
F	∞	None
G	∞	None
H	∞	None

Algoritmo Disjkstra- bucle principal

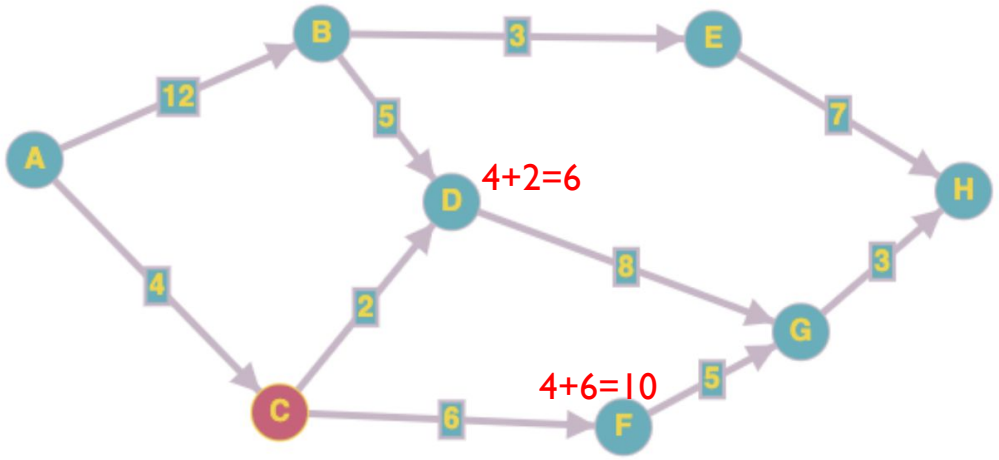
```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater
            distances[i] = distances[u]+w
            previous[i] = u
```

	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Volvemos a iterar, y en la segunda iteración deberemos tomar el vértice no visitado con menor distancia acumulada.

Algoritmo de Dijkstra - Cómo?

Marcamos a C como visitado, y recuperamos sus vértices adyacentes: {D, F}, que no han sido visitados.



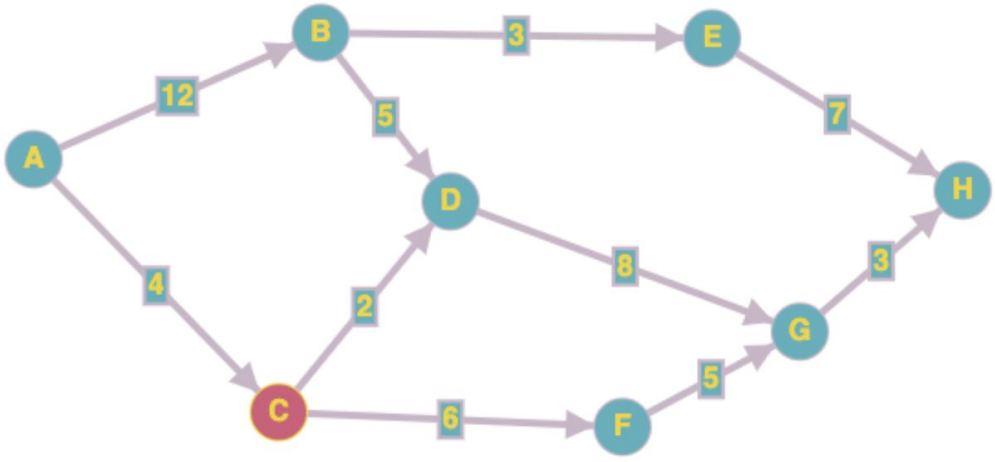
distancia[D]= ∞ > distancia[C] + 2 = 6 => Debemos actualizar

distancia[F]= ∞ > distancia[F] + 6 = 8 => Debemos actualizar

	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D	∞	
E	∞	
F	∞	
G	∞	
H	∞	

Algoritmo de Dijkstra - Cómo?

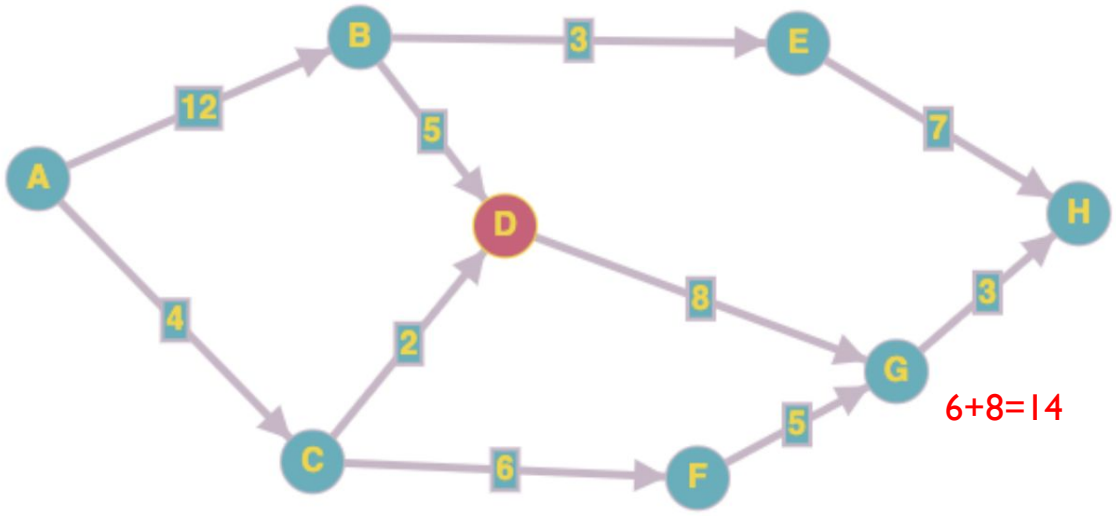
Además, guardamos C como vértice anterior para D y F.
Termina esa iteración, y el bucle sigue iterando (3 iteración < 8 número vértices).
Debemos seleccionar el vértice no visitado con menor distancia acumulada: D



	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D	4+2=6	C
E	∞	
F	4+6=10	C
G	∞	
H	∞	

Algoritmo de Dijkstra - Cómo?

Ahora el vértice no visitado con menor distancia acumulada es D. Lo marcamos como visitado y recuperamos sus vértices no visitados: G.

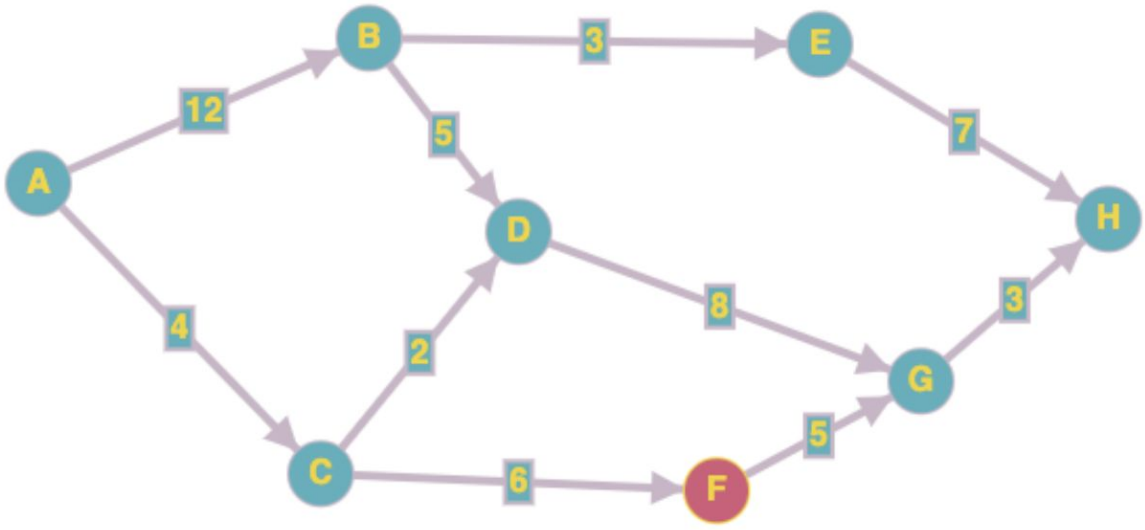


visitados[D]=True #[A,C,D] ya han sido
distancia[G]=∞ > distancia[D] + 8 = 14 =>
Debemos modificar

	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D ✓	6	C
E	∞	
F	10	C
G	∞	
H	∞	

Algoritmo de Dijkstra - Cómo?

Ha terminado la tercera iteración. Empezamos la 4 iteración < 8 (número de vértices). Ahora el vértice no visitado con menor distancia acumulada es F, y vamos a recuperar sus adyacentes: G.

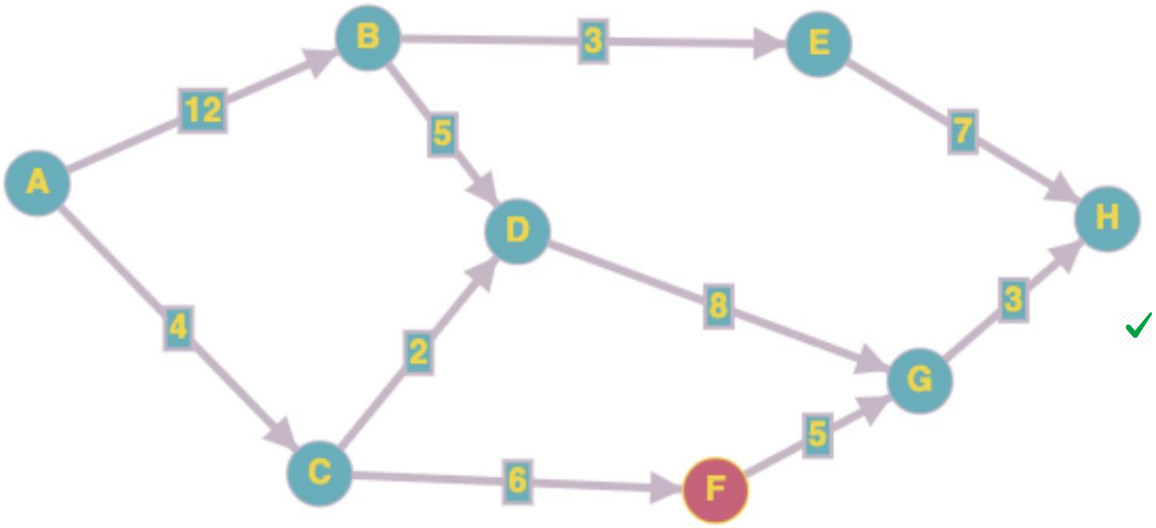


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D ✓	6	C
E	∞	
F ✓	10	C
G	14	D
H	∞	

visitado[F]= True
visitados=[A,C,D,F]

Algoritmo de Dijkstra - Cómo?

Para el único vértice adyacente no visitado a F, G, debemos comprobar: distancia[G] = 14 > distancia[F] + 5 = 10 +5. No se cumple!!!. Por tanto, no actualizamos la distancia y previo de G.

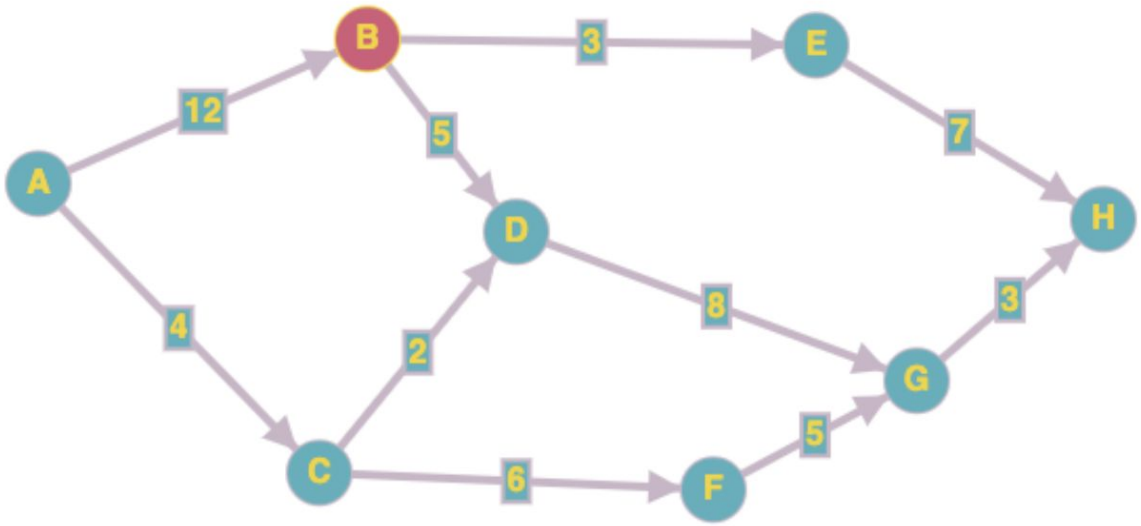


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D ✓	6	C
E	∞	
✓ F	10	C
G	14	D
H	∞	

Termina la cuarta iteración, Seguimos iterando porque 5 < 8

Algoritmo de Dijkstra - Cómo?

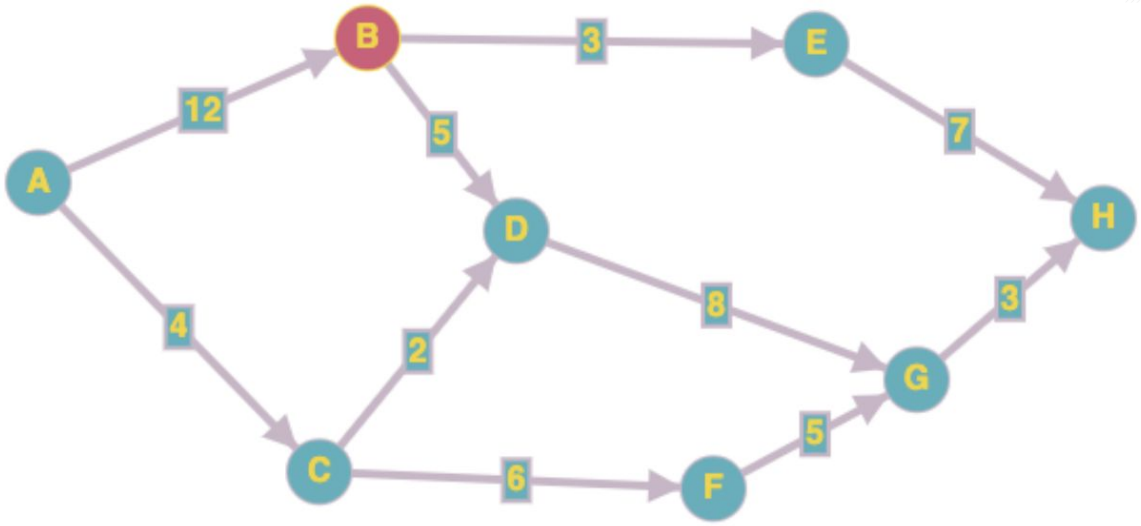
En la 5ª iteración, debemos seleccionar el vértice no visitado ({B, E, G, H},) con menor distancia acumulada: B



	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D ✓	6	C
E	∞	
F	10	C
G	14	D
H	∞	

Algoritmo de Dijkstra - Cómo?

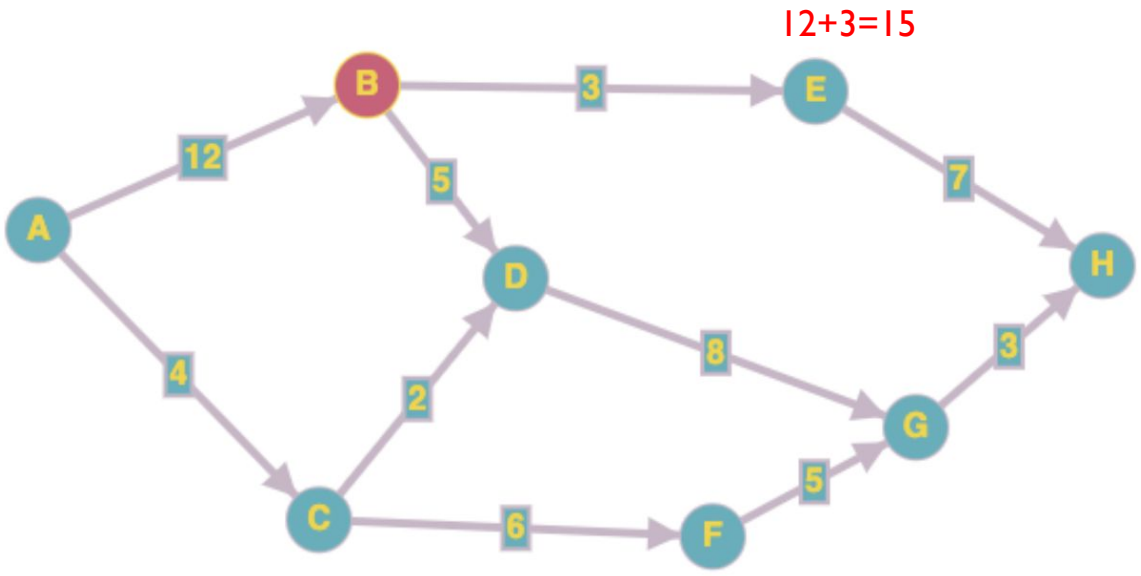
Marcamos B visitado, visitado[B]=True, y recuperamos sus adyacentes no visitados para comprobar:
d[E]= ∞ > d[B] + 3 = 15 => Modificar



	Longitud más corto desde A	Vértice anterior
A ✓	0	-
✓ B	12	A
C ✓	4	A
D ✓	6	C
E	∞	B
F ✓	10	C
G	14	D
H	∞	

Algoritmo de Dijkstra - Cómo?

Actualizamos también el previo de E para que sea B. Termina la quinta iteración y comienza la sexta (< 8). Otra vez debemos seleccionar el vértice no visitado {E, G, H}, con menor distancia acumulada: G

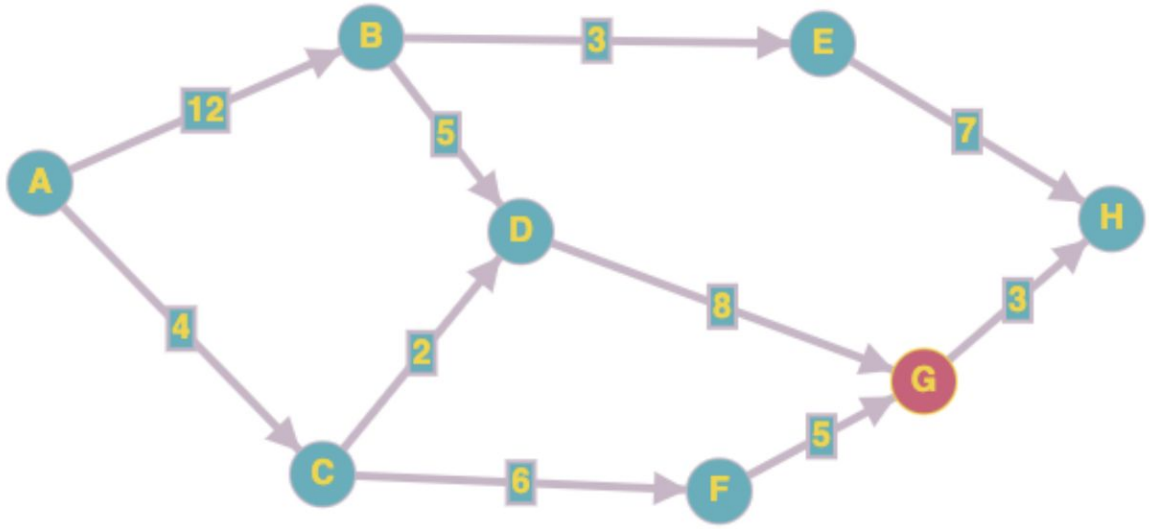


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E	15	B
F ✓	10	C
G	14	D
H	∞	

visitados=[A,C,D,F,B]

Algoritmo de Dijkstra - Cómo?

Marcamos G como visitados y tomamos su único vértice adyacente, que además no ha sido visitado: H. Calculamos d[H]=∞ > d[G] + 3 = 14 +3 = 18, y actualizamos su distancia acumulada.

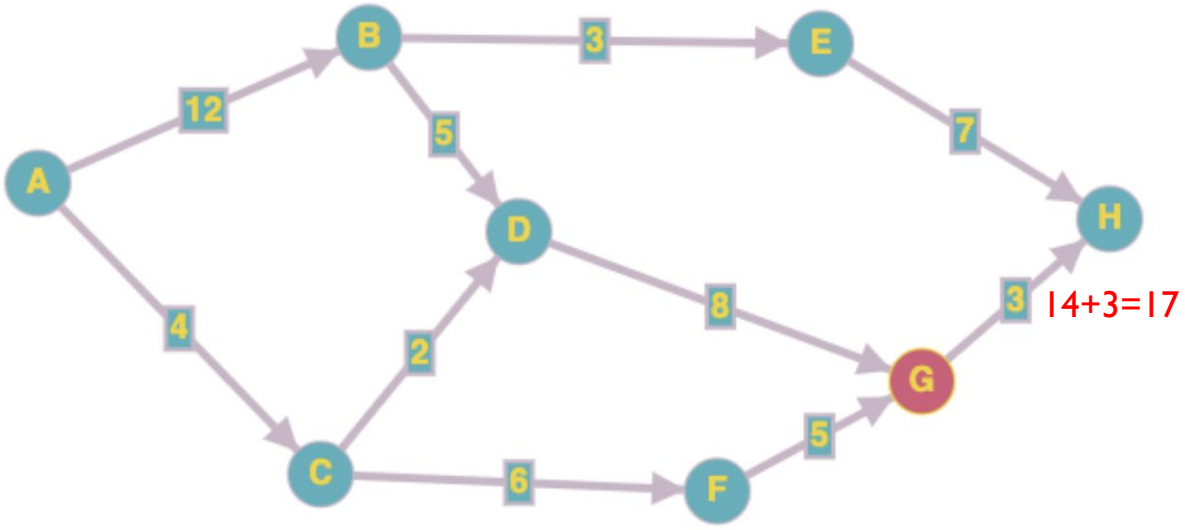


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E	15	B
F ✓	10	C
✓ G	14	D
H	∞	None

visitados[G]=True #[A,C,D,F,B,G]

Algoritmo de Dijkstra - Cómo?

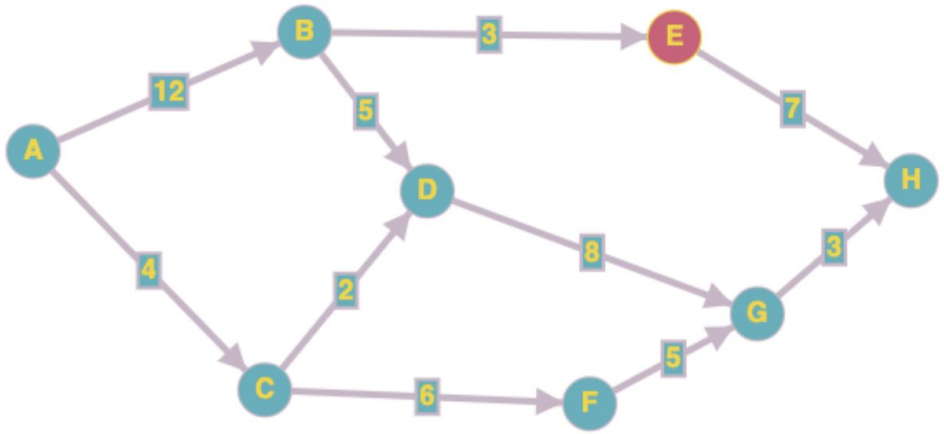
También debemos actualizar el predecesor de H a G.
Comenzamos la 7ª iteración (< 8), los vértices no visitados son {E, H}, siendo E el de menor distancia acumulada



	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E	15	B
F ✓	10	C
G ✓	14	D
H	14+3=17	G

Algoritmo de Dijkstra - Cómo?

Marcamos E como visitado, y obtenemos sus adyacentes. H es el único vértice adyacente no visitado. Debemos calcular:
d[H] = 17 > d[E] + 7 = 15 + 7 = 22. No se cumple, no actualizamos.

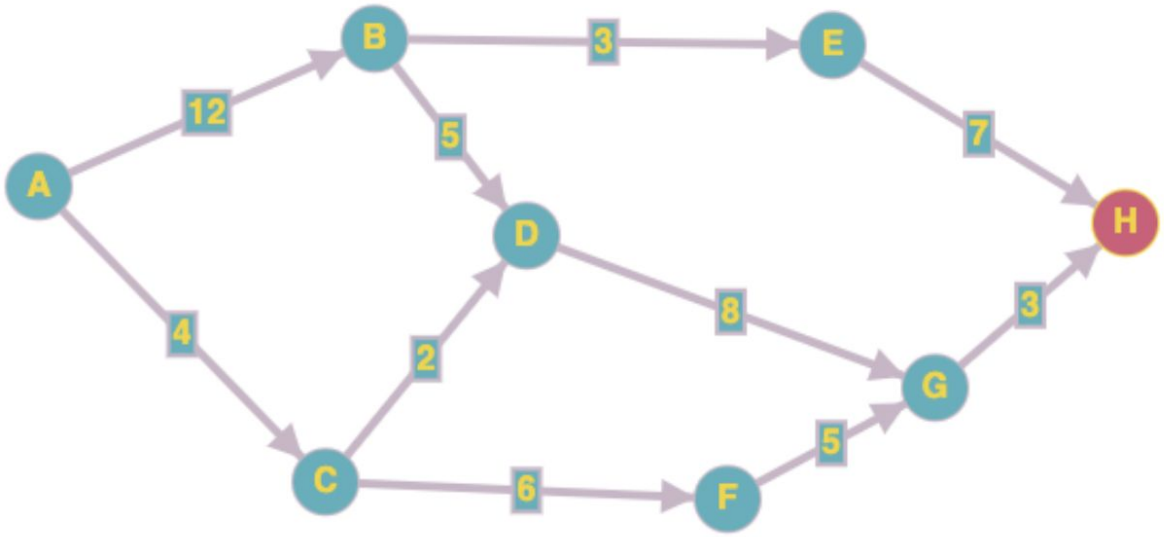


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
✓ E	15	B
F ✓	10	C
C ✓	14	D
H	17	G

visitados[E]=True #[A,C,D,F,B,G,E] ya han sido visitados

Algoritmo de Dijkstra - Cómo?

Ya estamos en la última iteración (8), y como era de esperar, sólo queda un vértice por visitar, que es H. Lo visitamos. H no tiene adyacentes (y aunque los tuviera todos estarían ya visitados). El algoritmo ha terminado.

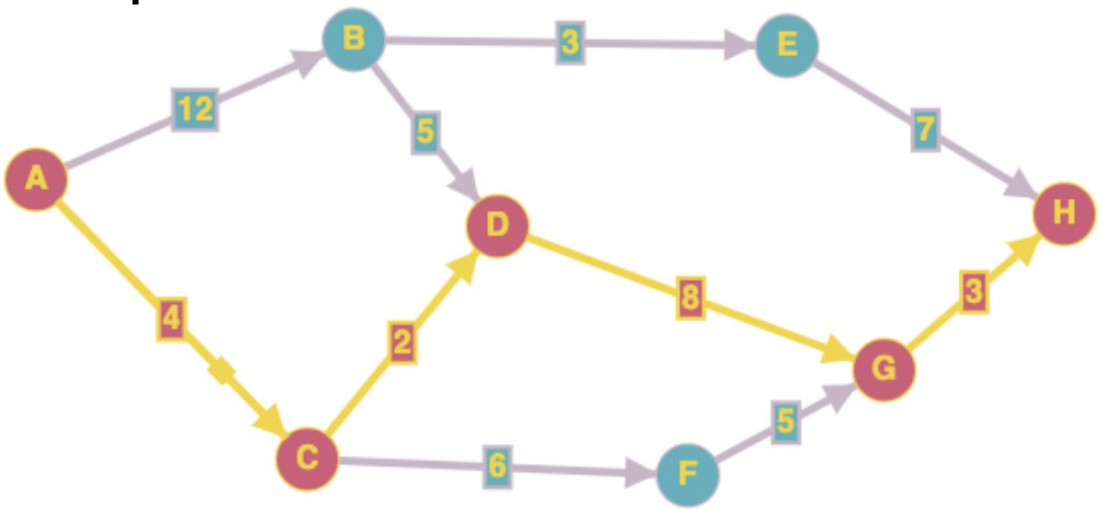


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E ✓	15	B
F ✓	10	C
G ✓	14	D
H	17	G

visitados[H]=True

Algoritmo de Dijkstra - Reconstruir caminos

La tabla contiene la distancia mínima del vértice A al resto de vértices. Además, la columna "Vértice anterior" nos permite reconstruir el camino mínimo para cualquiera de los vértices



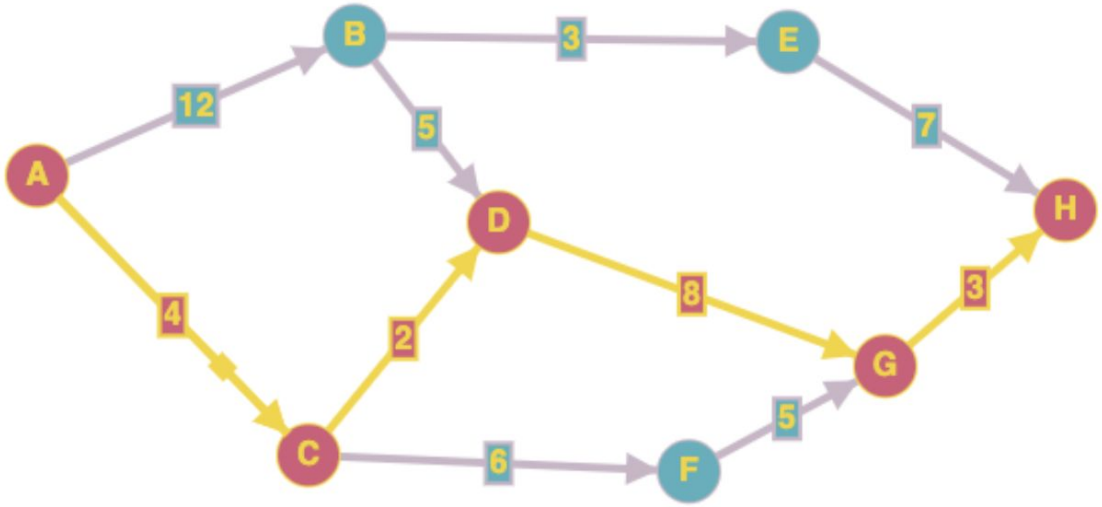
	Longitud más corto desde A	Vértice anterior
A	0	-
B	12	A
C	4	A
D	6	C
E	15	B
F	10	C
G	14	D
H	17	G

H->G>D->C->A
G>D->C->A
F->C->A
E->B->A

D->C->A
C->A
B->A

Algoritmo de Dijkstra - Reconstruir caminos

Por último, debemos invertir para obtener los caminos desde A al resto de vértices.



A->C>D->G->H
A>C->D->G
A->C->F
A->B->E

A->C->D
A->C
A->B

	Longitud más corto desde A	Vértice anterior
A	0	-
B	12	A
C	4	A
D	6	C
E	15	B
F	10	C
G	14	D
H	17	G

Análisis de complejidad espacial

- Usa 3 diccionarios. Cada diccionario son n keys con n values: 2n => 3 * 2n = 6n, donde n es el número de vértices

Análisis de complejidad temporal

```
def dijkstra(self, origin: object) -> None:
    visited = {}    # for each vertex (key), the value is a boolean indicating if the vertex has been visited
    previous = {}   # for each vertex (key), the value is the previous node in the minimum path from origin
    distances = {}  # for each vertex (key), the value is minimum distance in the minimum path from origin

    # initialize dictionaries
    for v in self._vertices.keys():
        visited[v] = False
        previous[v] = None
        distances[v] = math.inf

    # The distance from origin to itself is 0
    distances[origin] = 0
    ...
```

O(n)

En la primera parte, ya tenemos una complejidad lineal. Recorremos la lista de vértices (n vértices) e inicializamos cada diccionario. El resto de las instrucciones son tiempo constante.

Análisis de complejidad temporal

```
for _ in range(len(self._vertices)):
    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Tenemos un bucle principal que se va a ejecutar n veces, siendo n el número de vértices.

Análisis de complejidad temporal

```
for _ in range(len(self._vertices)):
    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

O(n) * n

1 * n

El método min_distance tiene complejidad lineal (O(n)) porque es necesario recorrer toda la lista de vértices para encontrar el vértice no visitado con menor distancia acumulada. Si tenemos en cuenta que min_distance se va a ejecutar n veces, ya tenemos complejidad O(n^2)
Visitar el vértice tiene complejidad 1.

Análisis de complejidad temporal

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True
```

Peor caso, n-1 * n

```
    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Este bucle interno se va a ejecutar tantas veces como vértices adyacentes a u. En el peor de los casos, puede ser n-1.
Todas las instrucciones tienen complejidad 1. Por tanto, la complejidad del bucle interno en el peor de los casos será n-1, que debemos de multiplicar por n, que es el número de veces que se ejecuta el bucle principal. Es decir, aquí volvemos a tener O(n^2)

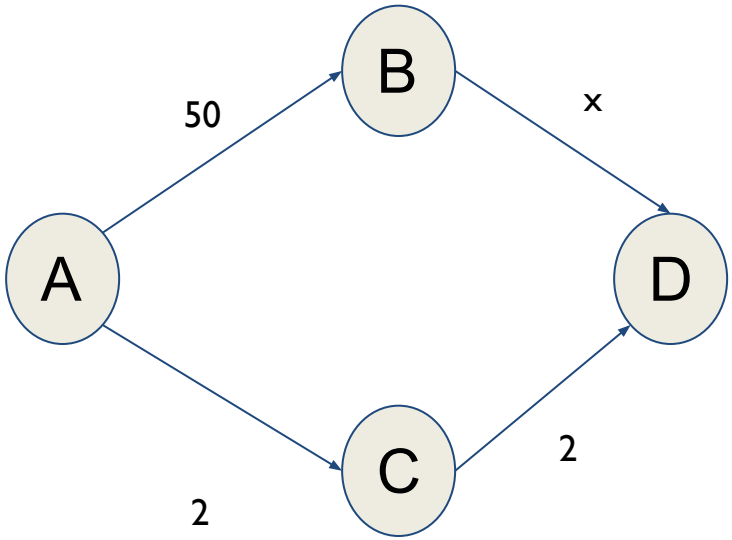
Análisis de complejidad temporal

Por tanto, la complejidad temporal del algoritmos de Dijkstra es $O(n^2)$

Ejercicio:

- Discute sobre la complejidad temporal de los recorridos (en profundidad y anchura) y el algoritmo de Dijkstra.

Ejemplo



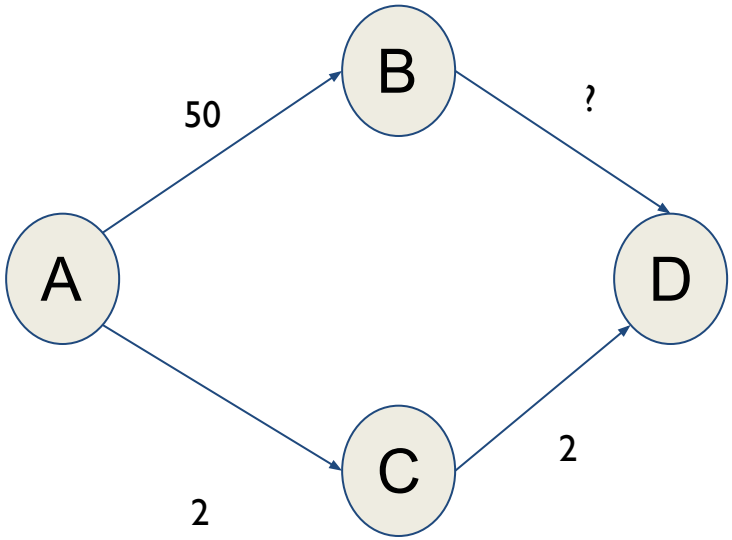
Aplicando el **algoritmo de Dijkstra**, vamos a calcular el camino mínimo de A al resto de vértices del grafo. Vamos a suponer que no conocemos el peso de la arista B -> D, únicamente que es positivo x>=0

Ejemplo

```
def dijkstra(self, origin: object) -> None:
    visited = {}    # for each vertex (key), the value is a boolean indicating if the vertex has been visited
    previous = {}   # for each vertex (key), the value is the previous node in the minimum path from origin
    distances = {}  # for each vertex (key), the value is minimum distance in the minimum path from origin

    # initialize dictionaries
    for v in self._vertices.keys():
        visited[v] = False
        previous[v] = None
        distances[v] = math.inf

    # The distance from origin to itself is 0
    distances[origin] = 0
    ...
```

En primer lugar, debemos inicializar los diccionarios y además, al vértice origen (en nuestro caso A), le asignamos la distancia mínima 0.

Ejemplo



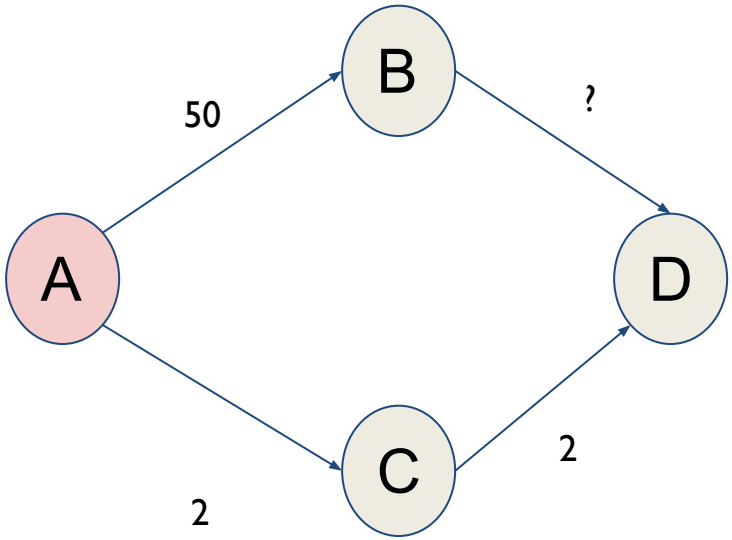
	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	∞	-
C	∞	-
D	∞	-

Ejemplo

```
for _ in range(len(self._vertices)):
    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

A continuación, vamos a tener un bucle que se ejecuta tantas veces como el número de vértices.
En cada iteración, siempre vamos a **seleccionar el vértice no visitado con la menor distancia y lo marcamos como visitado** (en la primera iteración, siempre será el vértice origen).

Ejemplo



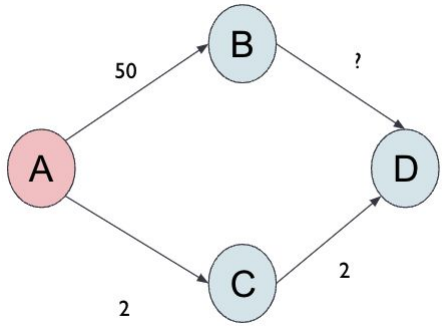
visited[A] = True

	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	∞	-
C	∞	-
D	∞	-

Ejemplo

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

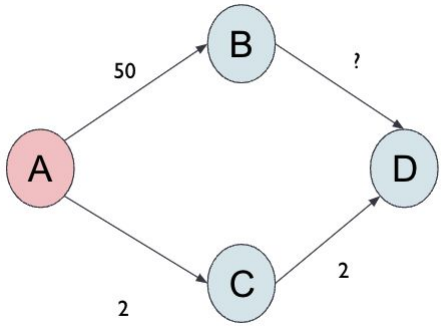
Recuperamos los vértices adyacentes del vértice que estamos visitando.
En nuestro caso A, sus adyacentes son B y C.



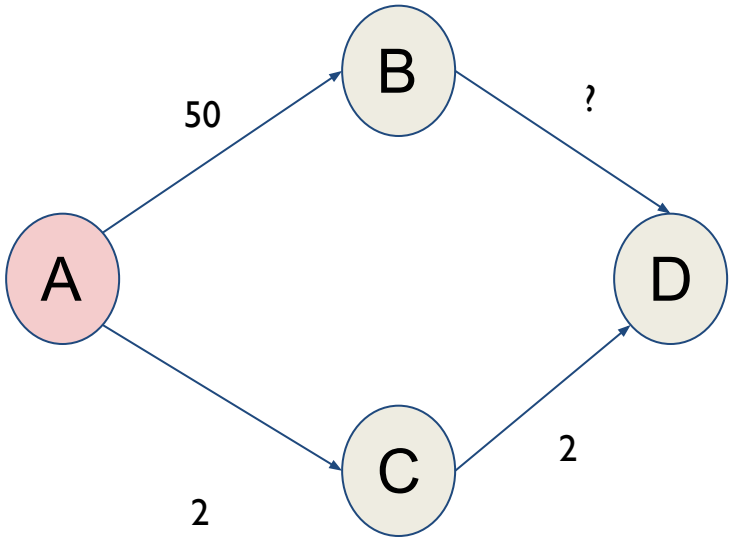
Ejemplo

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Para cada adyacente:
Como B no está visitado y distancia[B] = ∞ > distancia[A] + 50,
Debemos actualizar la distancia acumulada de B, e indicar que A es el vértice anterior en el camino mínimo desde A.



Ejemplo

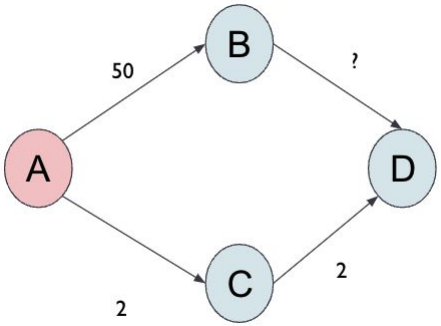


	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	0+50	A
C	∞	-
D	∞	-

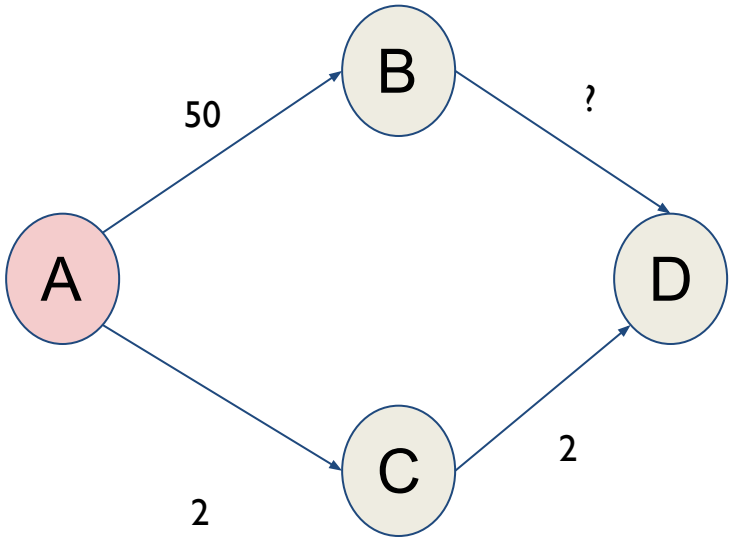
Ejemplo

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

El siguiente adyacente de A es C:
Como C no está visitado y distancia[C] = ∞ > distancia[A] + 2,
Debemos actualizar la distancia acumulada de C, e indicar que A es el vértice anterior en el camino mínimo desde A.



Ejemplo



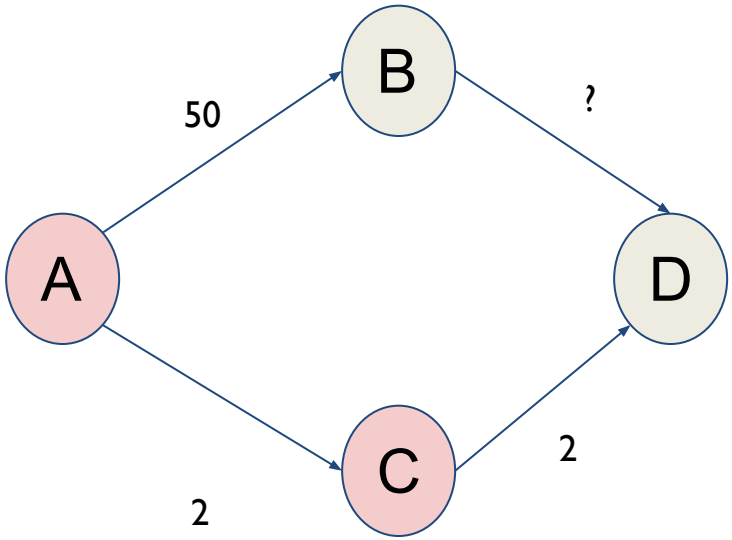
	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	∞	-

Ejemplo

```
for _ in range(len(self._vertices)):
    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Volvemos a iterar (el bucle se ejecutará 4 veces porque tenemos 4 vértices; esta es la segunda iteración).
Primer paso seleccionar el vértice no visitado con la menor distancia acumulada (en esta iteración es C) y lo visitamos.

Ejemplo



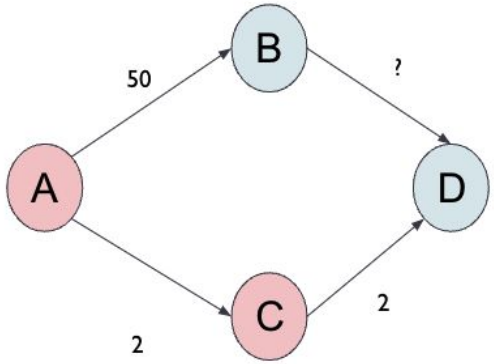
visited[C] = True

	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	∞	-

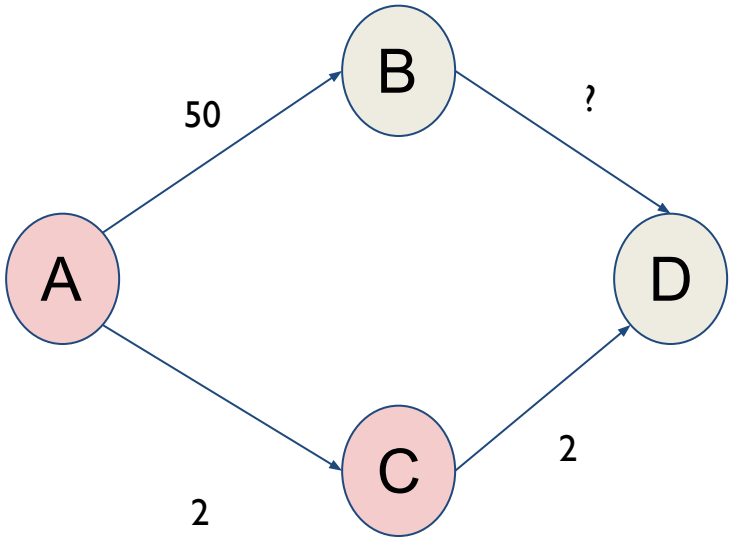
Ejemplo

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Recuperamos los adyacentes no visitados de C: únicamente D. Tenemos que comprobar si d[D]>d[C] + 2



Ejemplo



	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	2+2	C

distancia[D]=∞ > d[C] + 2 = 2 +2 =>
d[D] = 2 + 2 = 4, en indicamos que el vértice previo a D desde el camino mínimo de A es C.

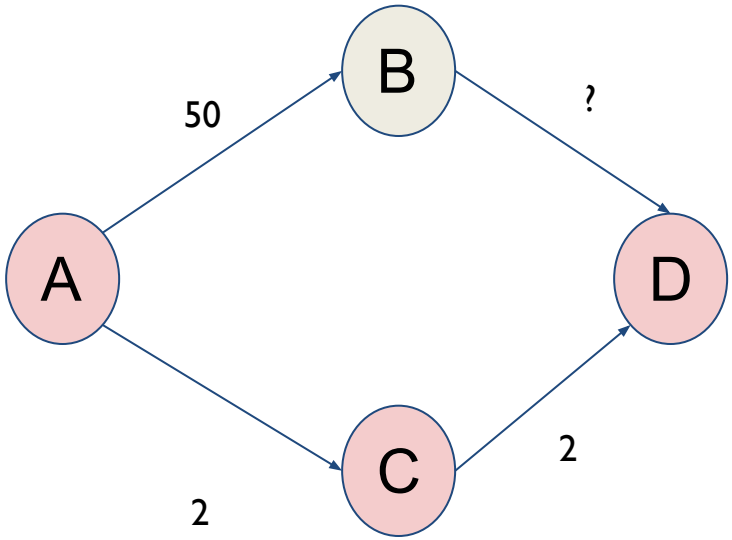
Ejemplo

```
for _ in range(len(self._vertices)):
    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Volvemos a iterar (esta es la tercera iteración).
El vértice no visitado con menor distancia acumulada es D

	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Ejemplo



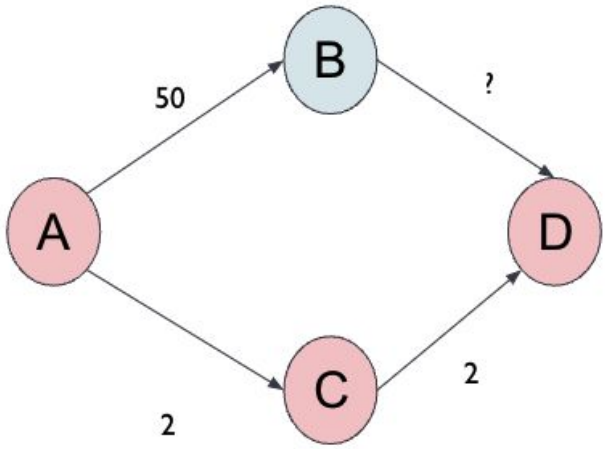
visited[D] = True

	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Ejemplo

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Como D no tiene adyacentes no se ejecuta el bucle interior. La tercera iteración ya ha terminado



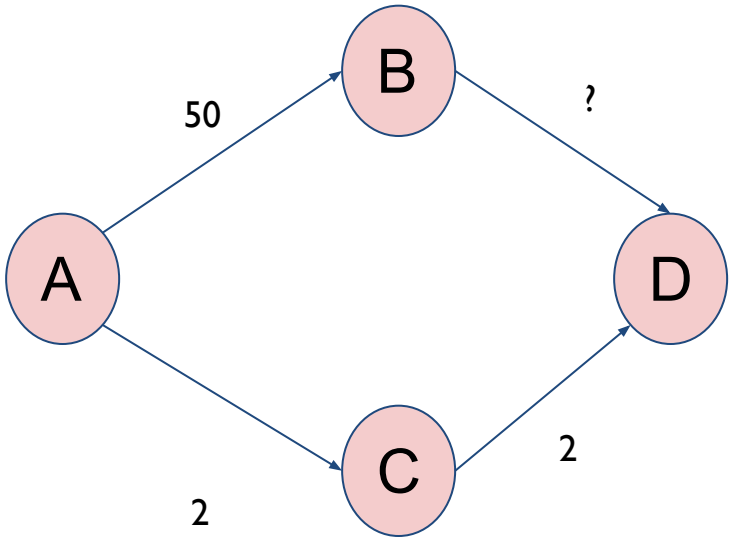
Ejemplo

```
for _ in range(len(self._vertices)):
    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

Cuarta y última iteración.
El último vértice que queda por visitar es B. Lo visitamos

	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Ejemplo



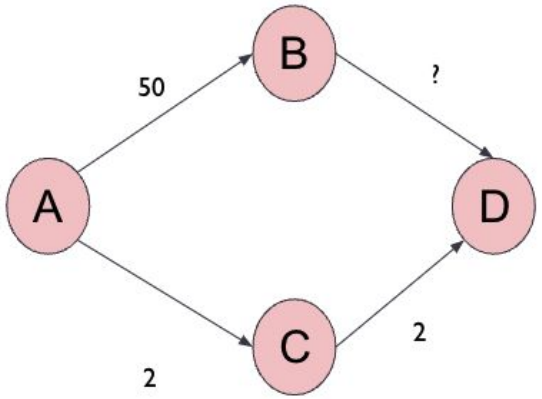
visited[B] = True

	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

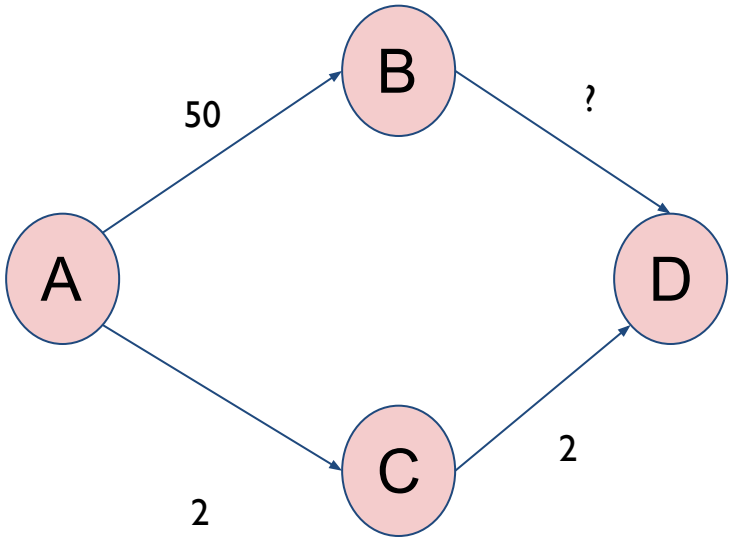
Ejemplo

```
for _ in range(len(self._vertices)):

    u = self.min_distance(distances, visited)
    visited[u] = True

    for adj in self._vertices[u]:
        i = adj.vertex
        w = adj.weight
        if not visited[i] and distances[i] > distances[u]+w:
            # we must update because its distance is greater than the new distance
            distances[i] = distances[u]+w
            previous[i] = u
```

B sólo tiene un vértice adyacente: D. D ya ha sido visitado, no habría que hacer nada más y **el algoritmo habría terminado** (independientemente del valor de la arista B -> D)



Ejemplo

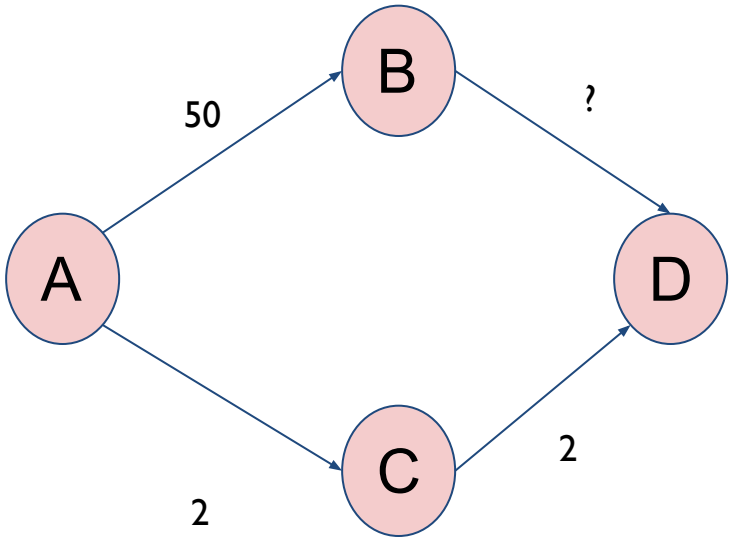


	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Caminos mínimos:

- A a B: [A, B], d = 50
- A a C: [A, C], d = 2
- A a D: [A, C, D], d = 4

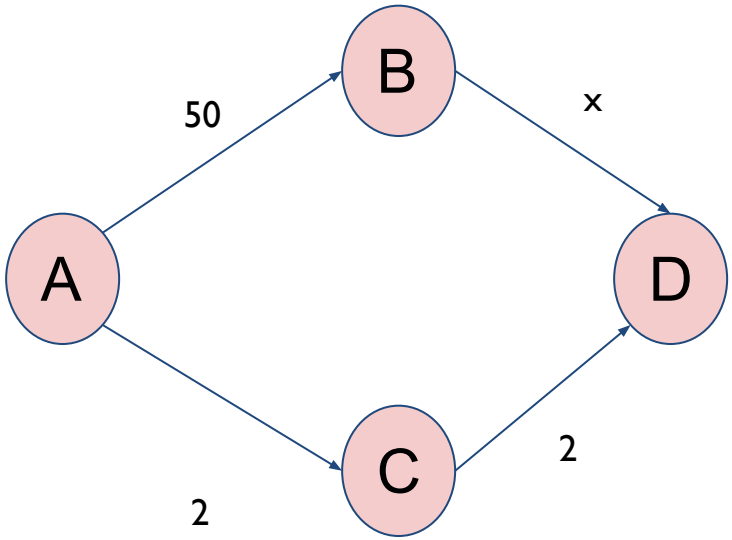
Ejemplo



	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

¿Podemos asegurar que [A, C, D], d = 4, es el camino mínimo de A a D?:

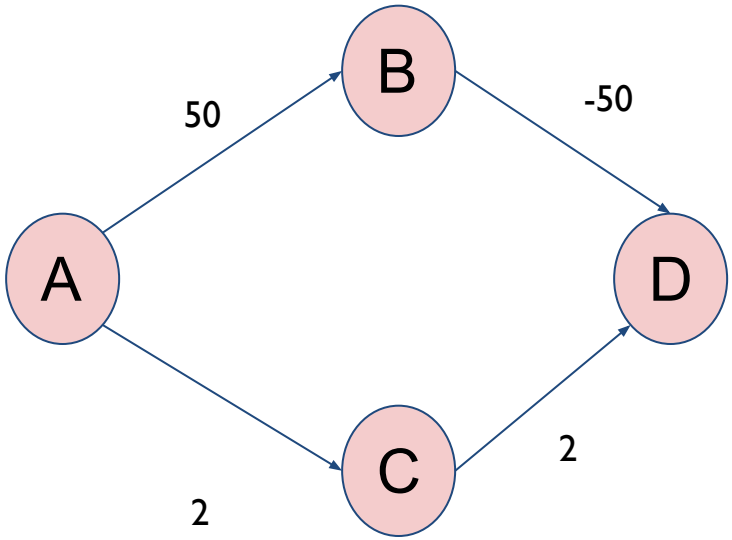
Ejemplo



	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Sea x>=0, la distancia de B a D.
No existe ningún valor de x para el que se cumpla:
d[D] = 4 > d[B] + x = 50 +x
4 > 50 +x

Ejemplo



	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Sin embargo, si x = -50, el algoritmo de Dijkstra no habría encontrado el camino mínimo de A a D!!!.
Por tanto, Dijkstra no funciona para grafos ponderados con pesos negativos.