uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 6 Grafos

Profesores EDA

Objetivos

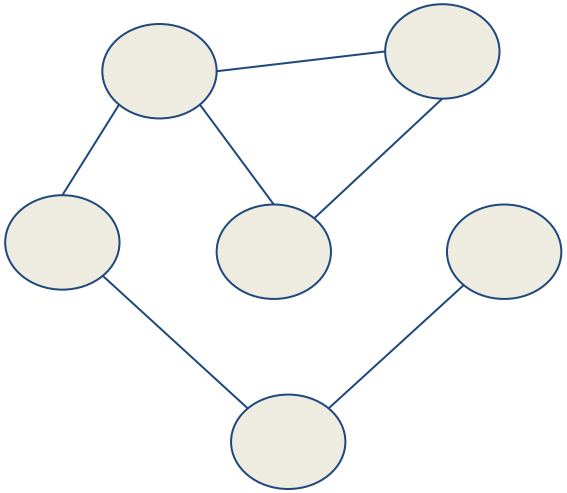
1. Conocer qué es un grafo y sus principales propiedades.
2. Implementar el tipo abstracto grafo usando distintas representaciones.
3. Comprender la complejidad espacial y temporal de cada representación.
4. Comprender e implementar algoritmos para visitar los vértices de un grafo.
5. Resolver problemas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

Grafo es una estructura no lineal, compuesta por vértices (nodos) y aristas (conexiones entre vértices).



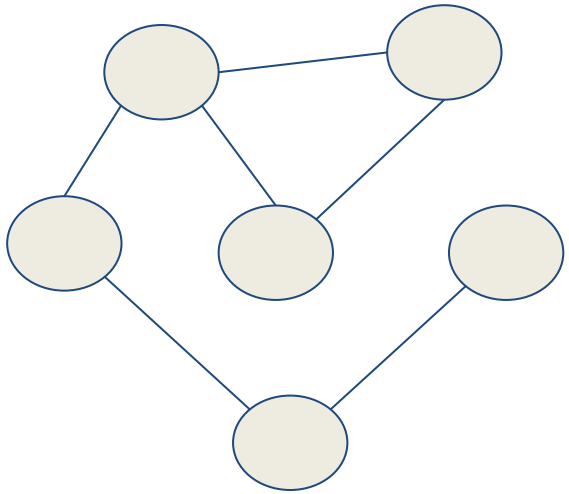
Índice

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

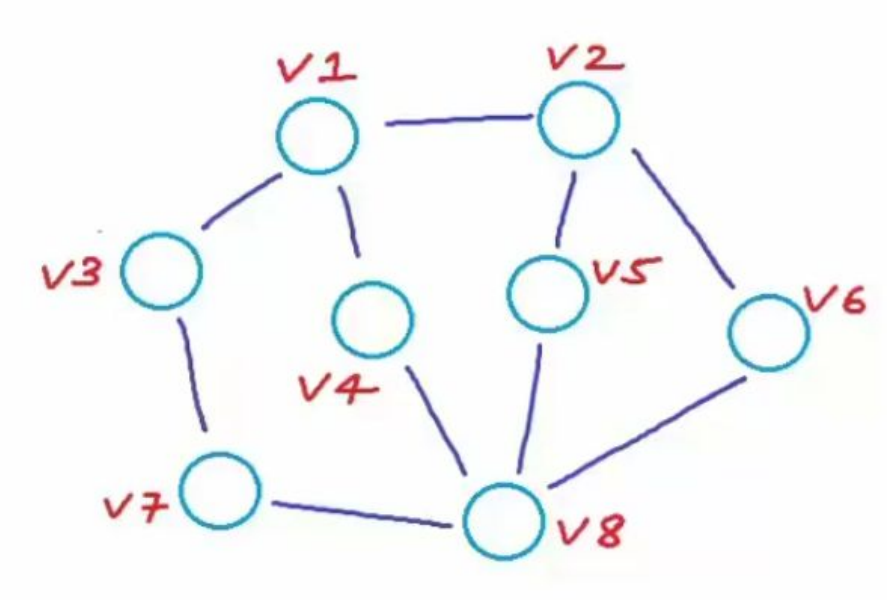
Conceptos sobre grafos

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

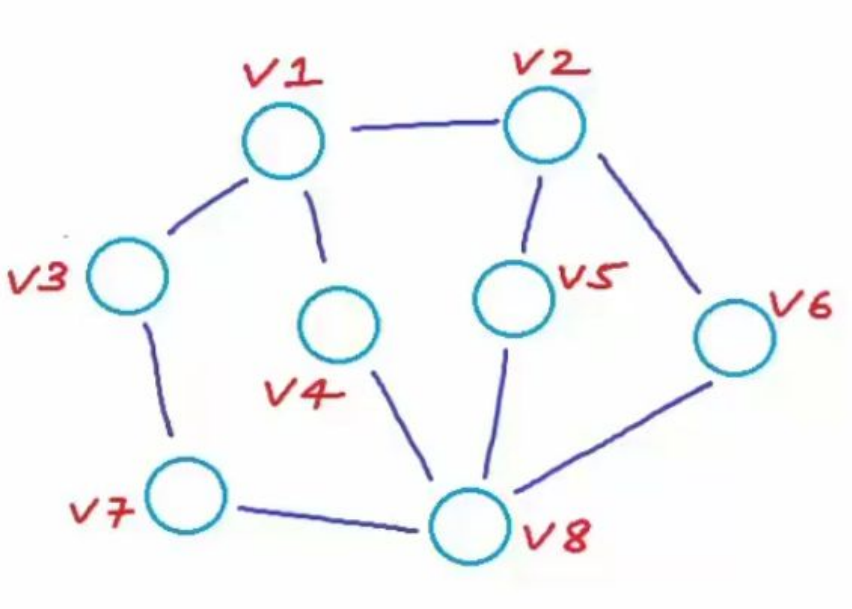
{u,v} ,
{u,v} = {v,u}

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

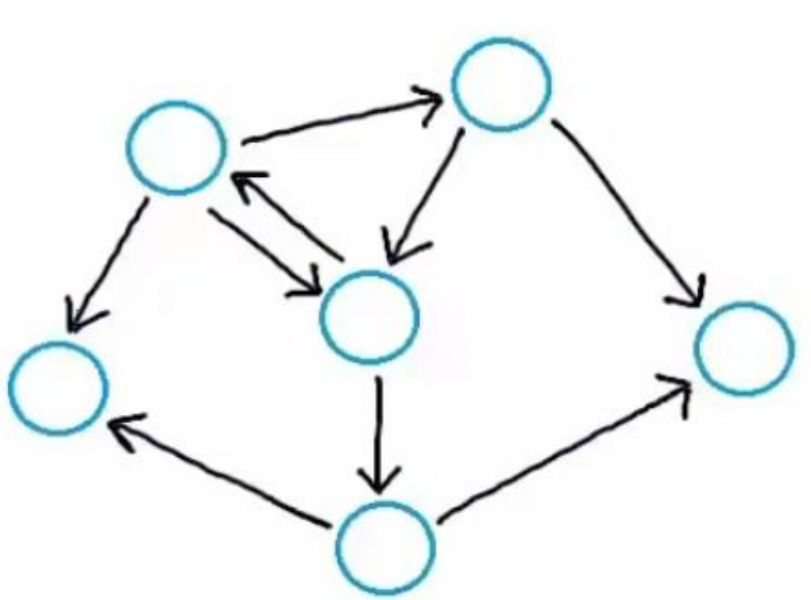
IAI =número de aristas

IVI = 8

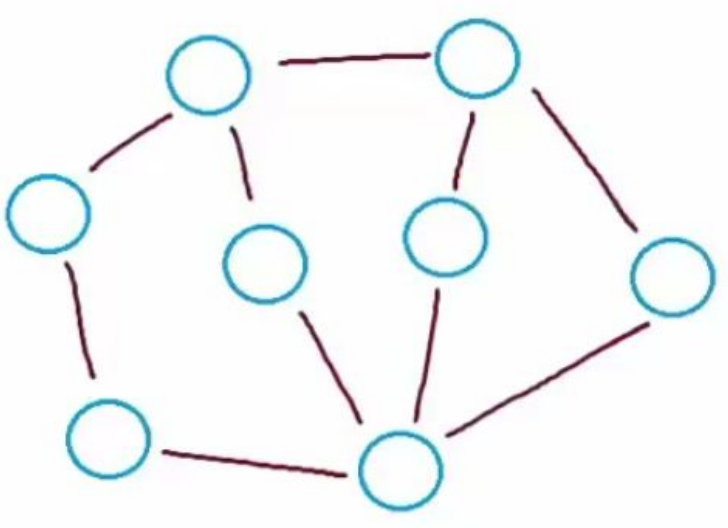
IAI=10

Conceptos sobre grafos

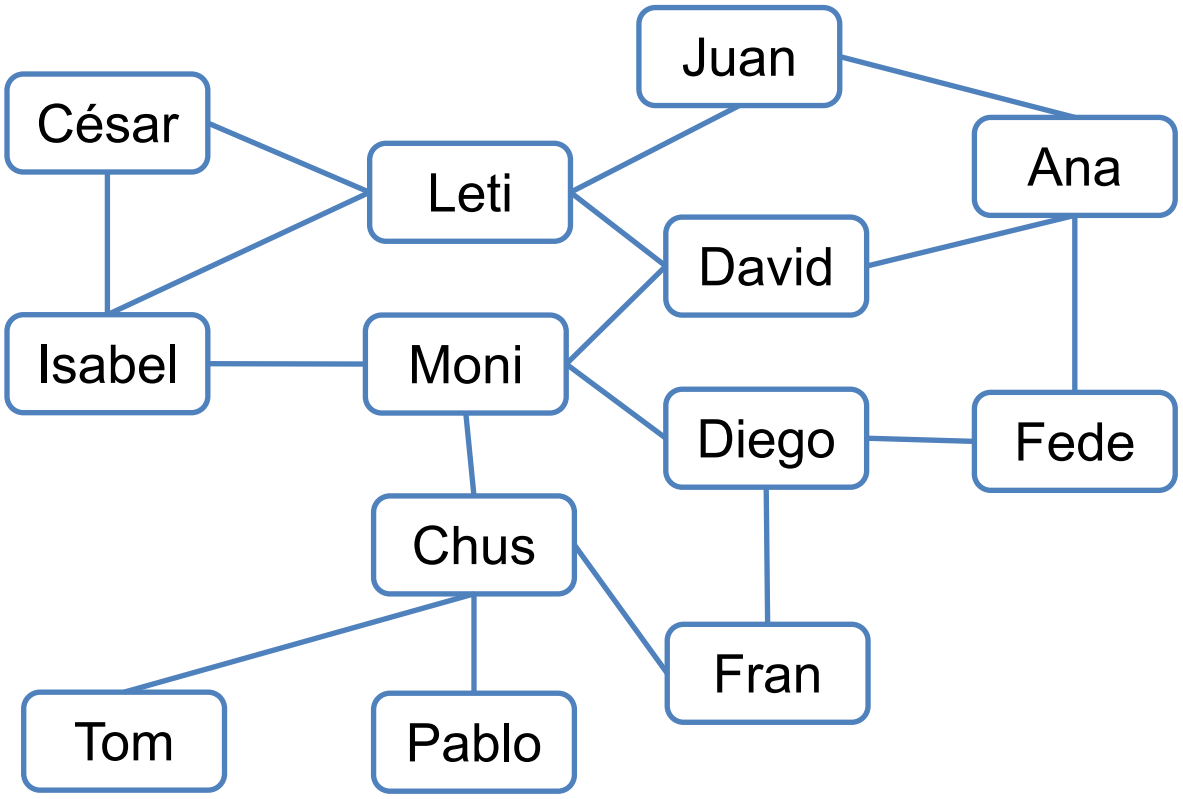
Grafo dirigidos



Grafo no dirigido

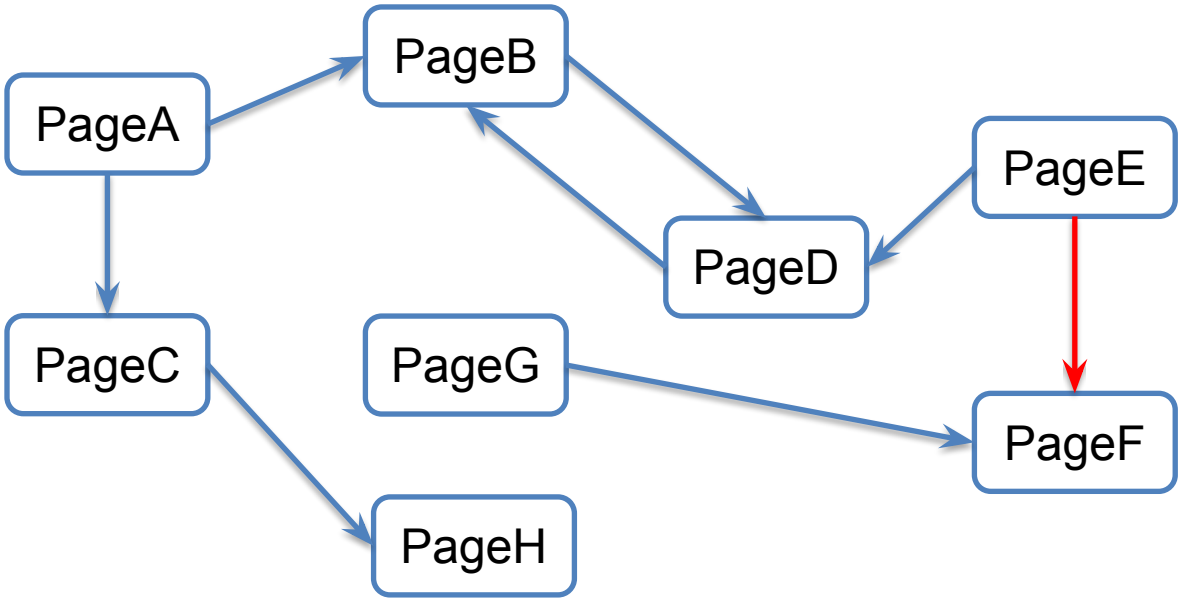


Ejemplo de grafo no dirigido: red social



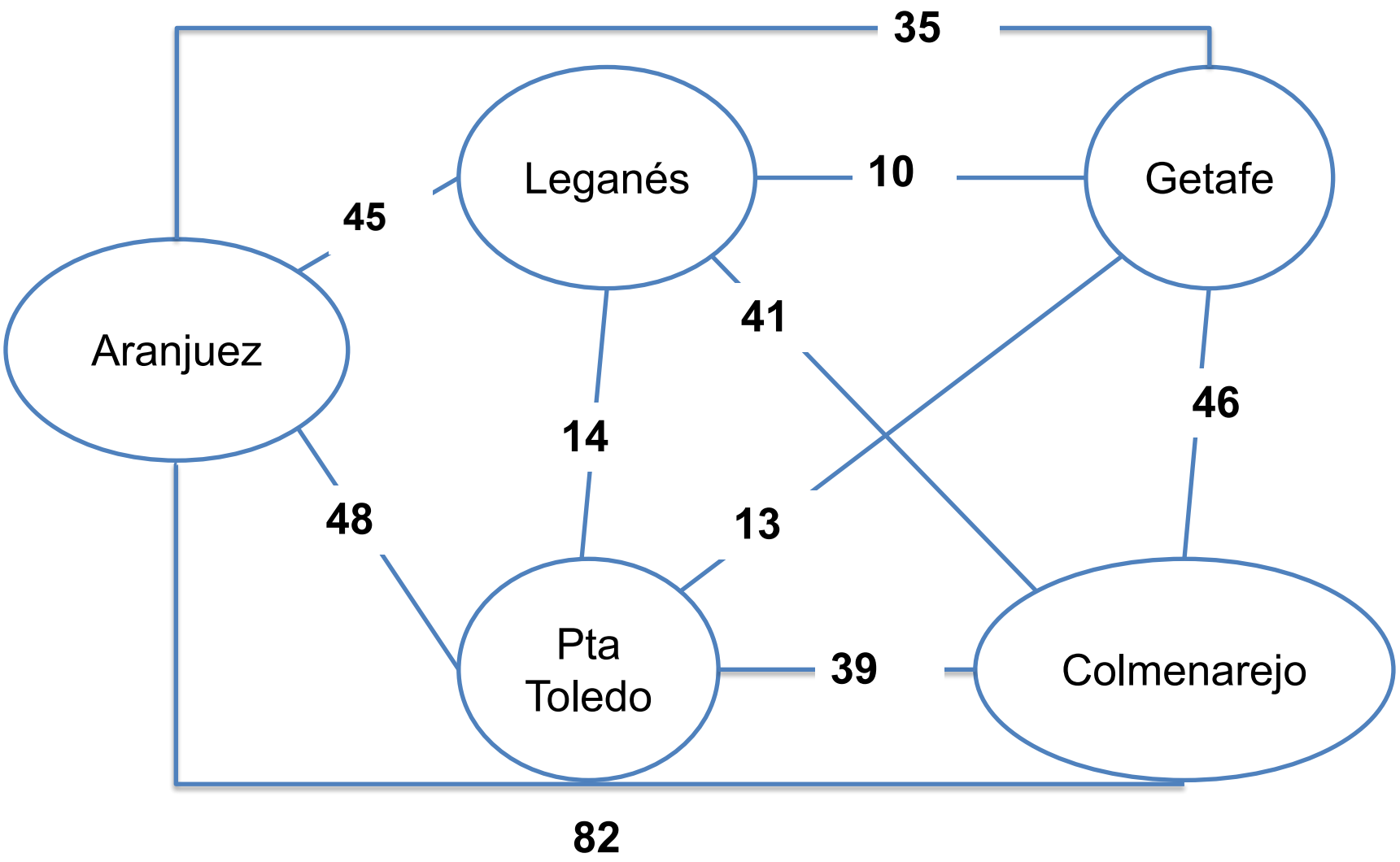
Una red social se puede representar como un grafo no dirigido y no ponderado. Los usuarios son los vértices y sus relaciones de amistad las aristas.

Ejemplo de grafo dirigido: web



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.

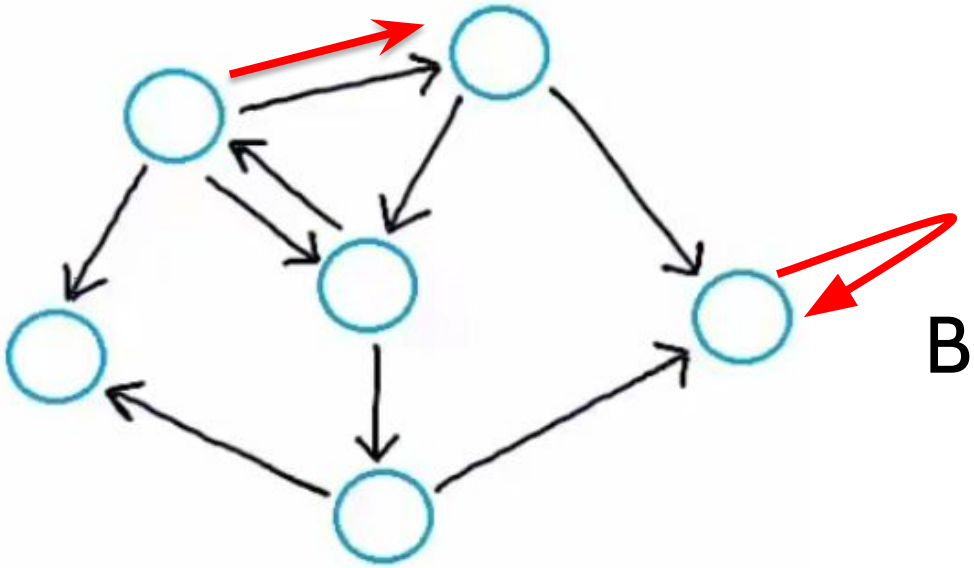
Grafos ponderados (weighted)



Campus de la UC3M (distancias en KM)

Conceptos sobre grafos: aristas paralelas y bucles (loops)

Aristas múltiples (aristas paralelas)

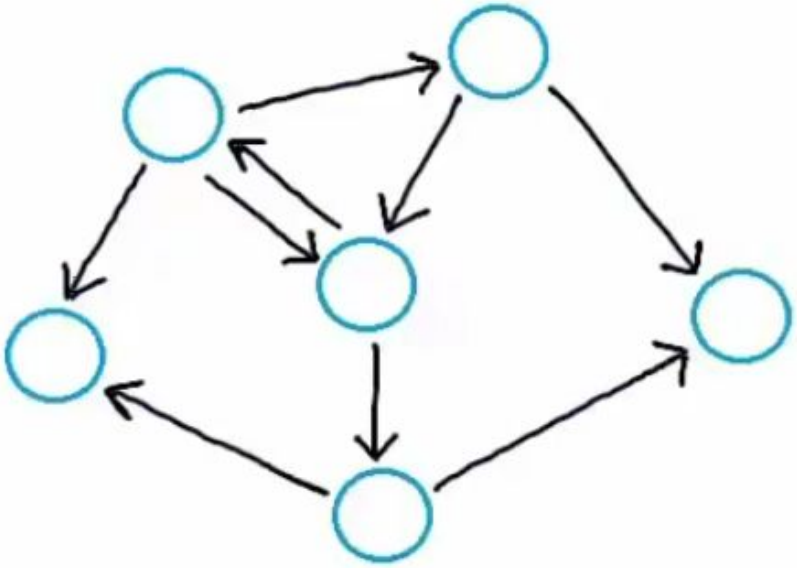


Bucle (loop)

- Los bucles y las aristas paralelas complican los algoritmos en grafos.

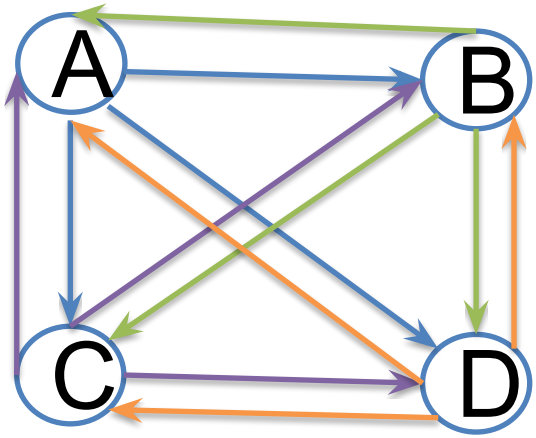
Conceptos sobre grafos: grafo simples

- 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**?



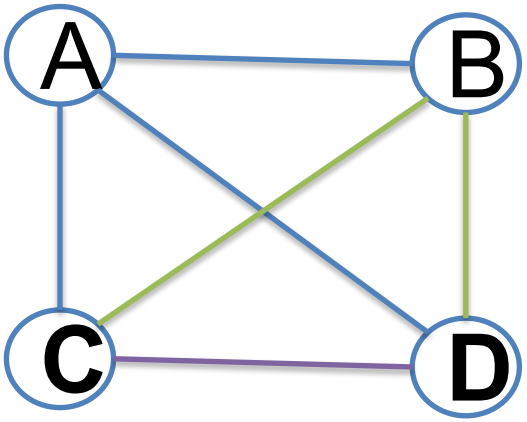
|V| = 4
|A| = 0 (mínimo)

|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**?



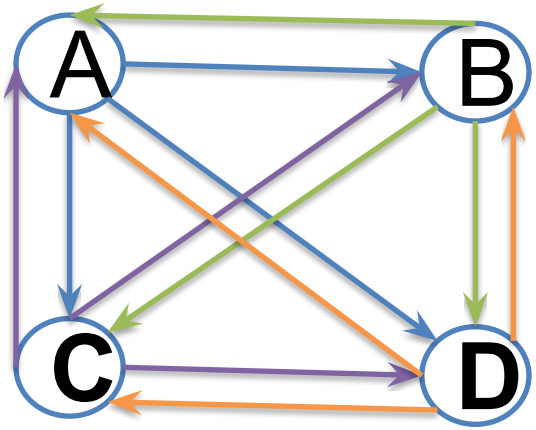
Si */V/ = n, cada vértice podría tener n-1 aristas.*
Las aristas son unidireccionales.

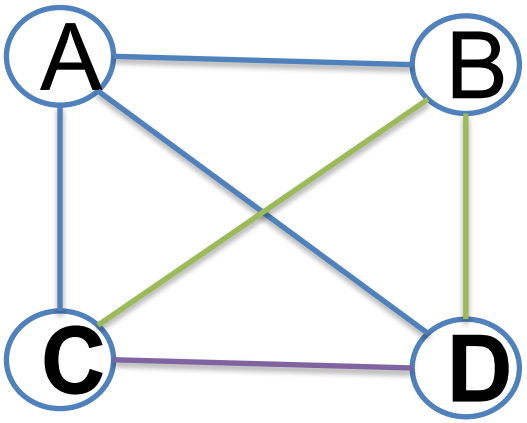
0<= /A/<=n(n-1)/2

Conceptos sobre grafos: grafo denso

- Un grafo es **denso** si el número de sus aristas es cercano a su número máximo posible, es decir,
  - n(n-1) si es un grafo dirigido,
  - n(n-1)/2 si es un grafo no dirigido.

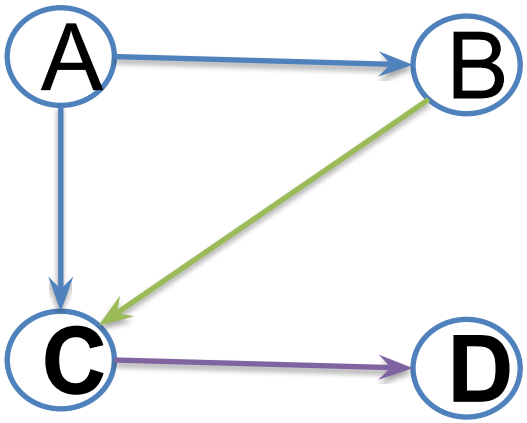
En ambos casos, el número de aristas es proporcional a n^2, siendo n el número de vértices.





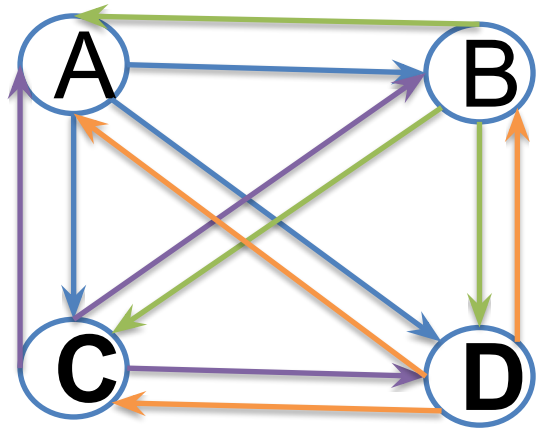
Conceptos sobre grafos: grafo escaso

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

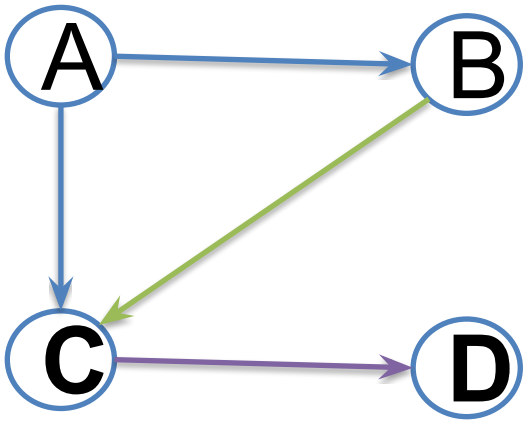


Conceptos sobre grafos

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



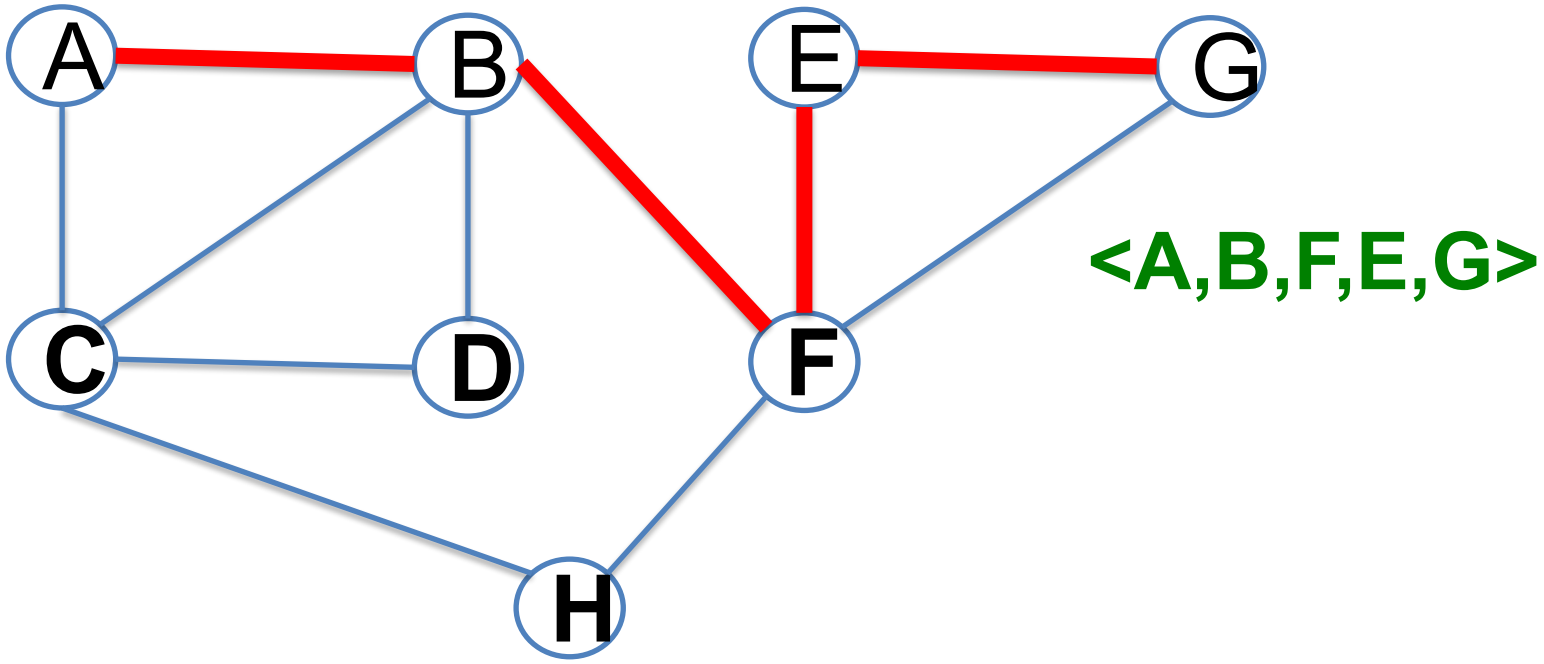
dense



sparse

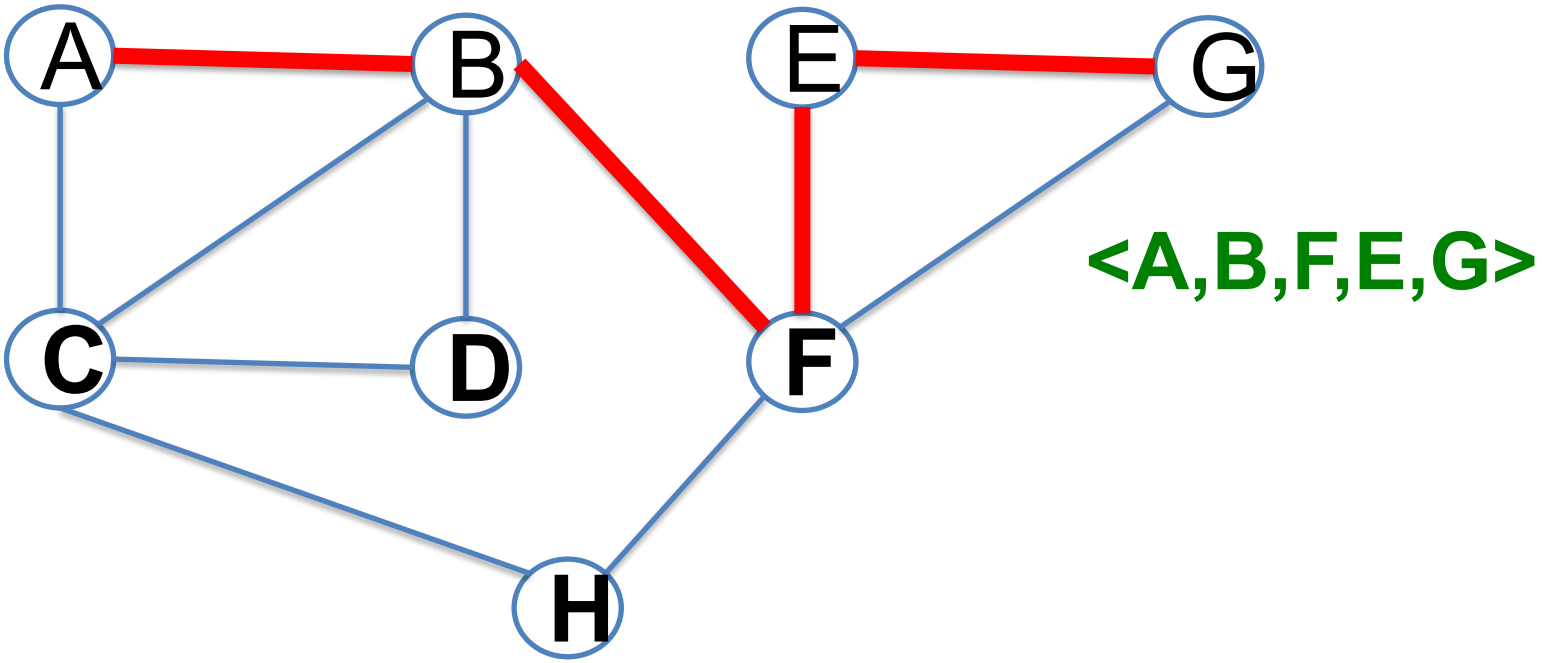
Conceptos sobre grafos: camino

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



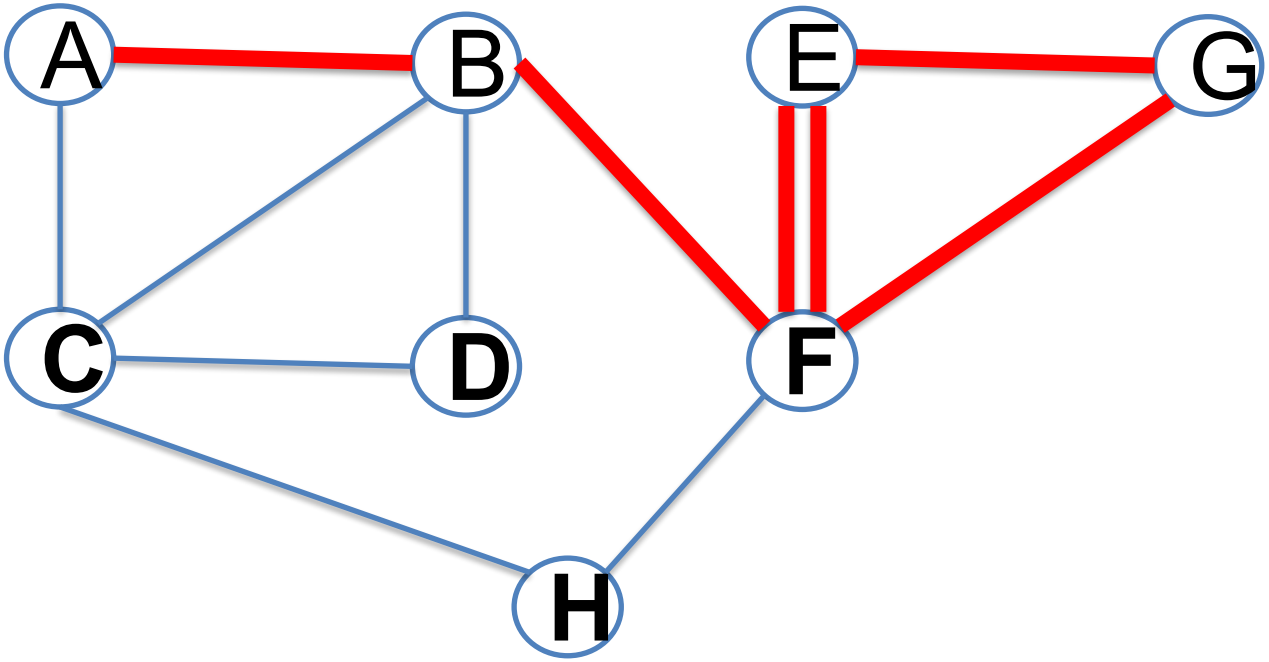
Conceptos sobre grafos

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

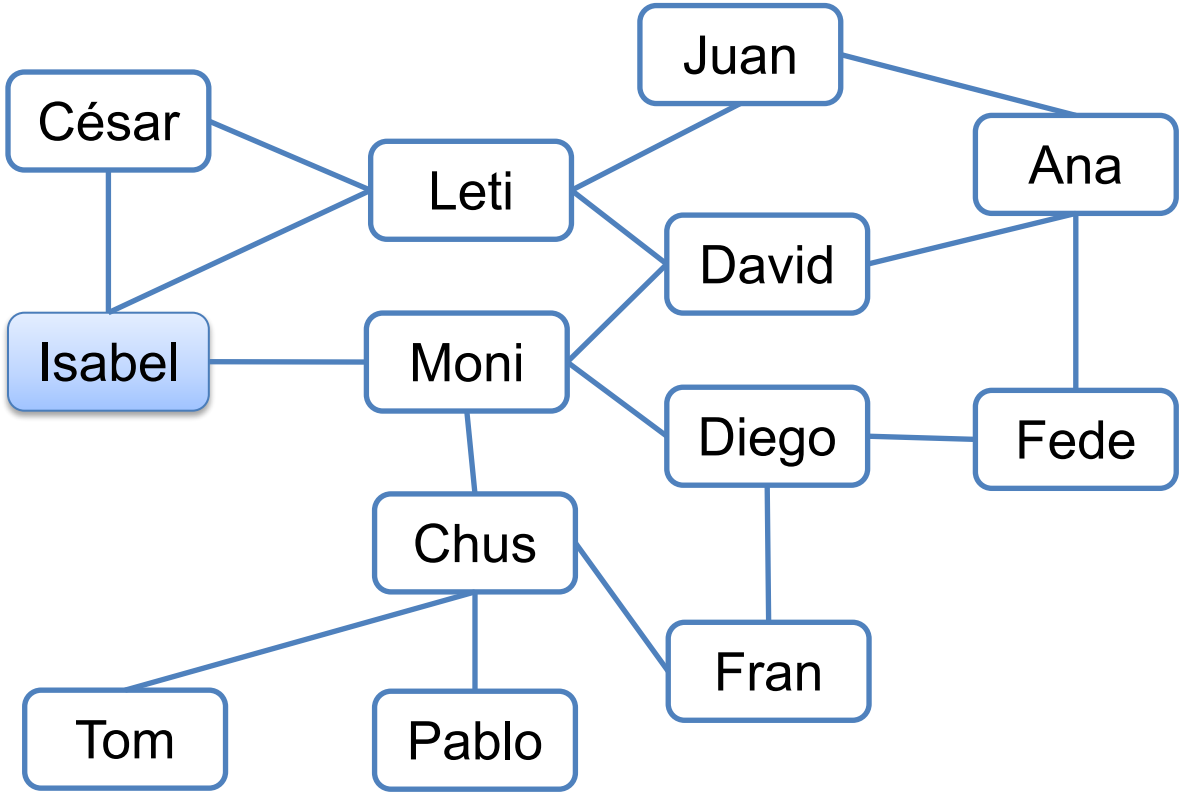


Conceptos sobre grafos

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

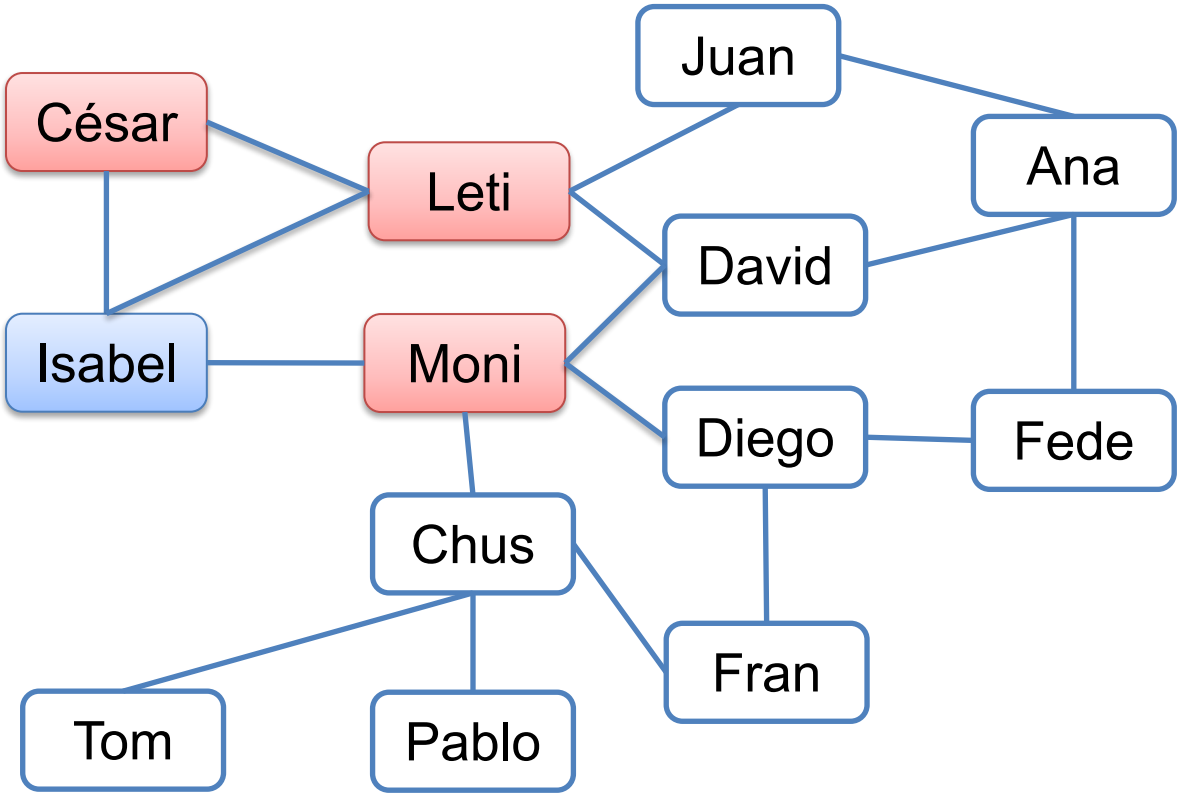


Conceptos sobre grafos: caminos

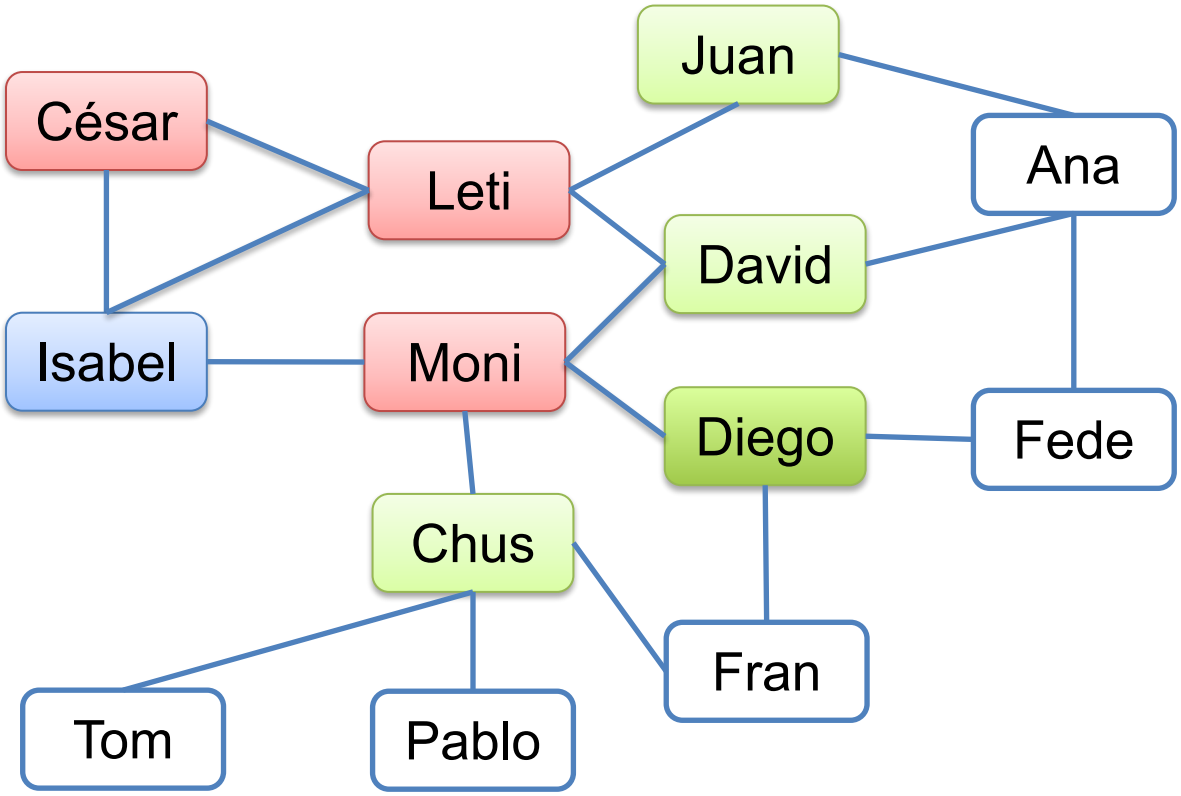


¿Cómo sugerir nuevos amigos a Isabel?

Conceptos sobre grafos: caminos



Conceptos sobre grafos



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

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**: permite crear un grafo a partir de una lista de vértices. No se añade ninguna arista. En esta operación, también se debe indicar si el grafo es dirigido o no.
- **add_vertex(v):** añade un nuevo vértice v al grafo. Si el vértice ya existe, no añade nada.
- **add_edge(start, end, weight=1)**: crea una arista de start a end, con peso weight. Si el grafo es no ponderado, weight es 1 por defecto. Si el grafo no es 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 la arista devuelve el peso de la arista. Si la arista no existe devuelve 0.
- **remove_edge(start, end)**: borra la arista de start a end. Recuerda que si el grafo no es dirigido, también debe eliminar la arista de end a start.
- **get_adjacent_vertices**(start): devuelve una lista con los vértices adyacentes de start.
- **get_origins**(end): devuelve una lista con los vértices v que tienen un arita de v a end.

TAD Grafo

Más operaciones:

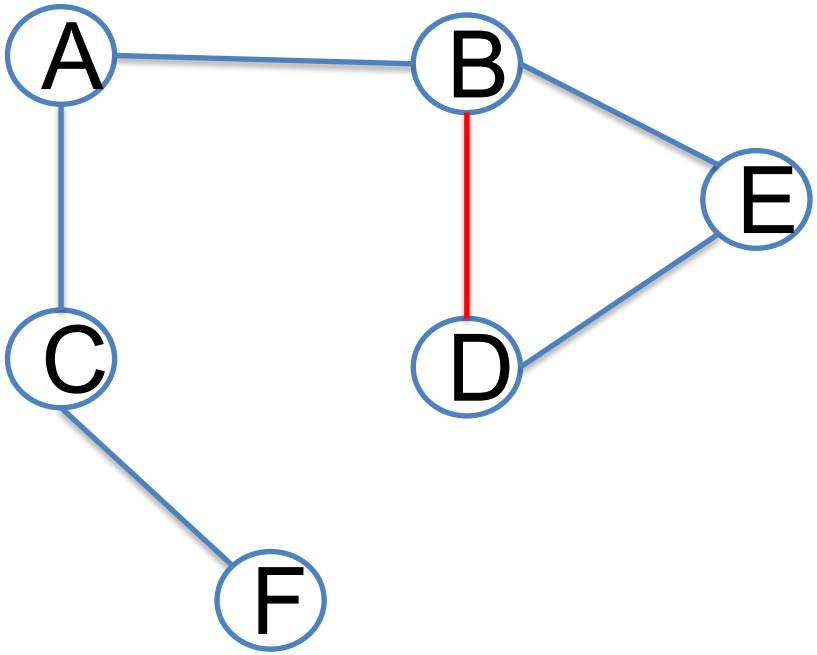
- **bfs(start)**: devuelve (o imprime) el recorrido en anchura desde el vértice start.
- **dfs(start)**: devuelve (o imprime) el recorrido en profundidad desde el vértice start.
- **minimum_path**(start, end): devuelve una lista de vértices con el camino mínimo de vértice *start* a *end*.

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

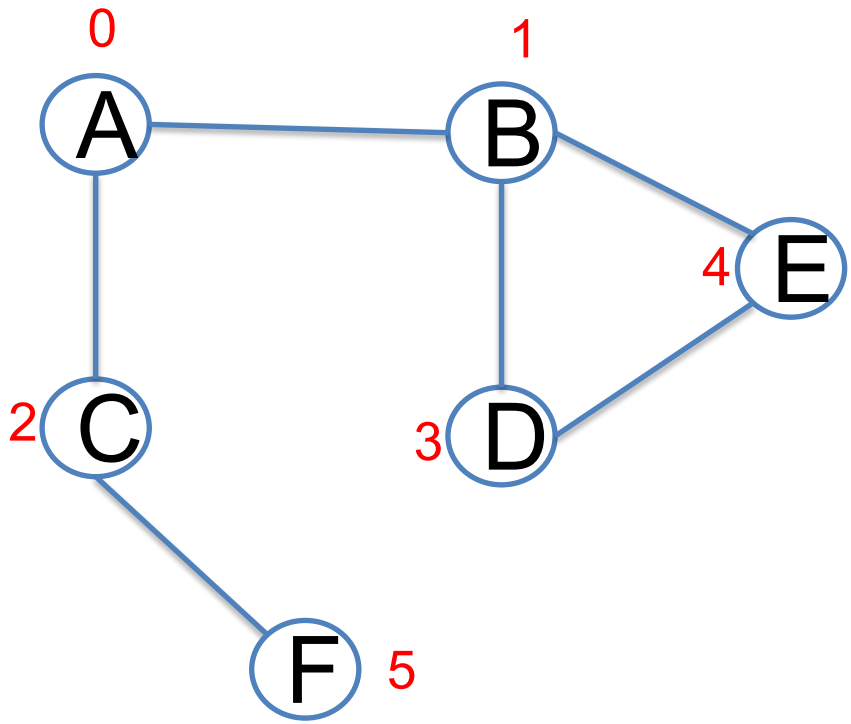


¿Cómo puede representar un grafo?

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

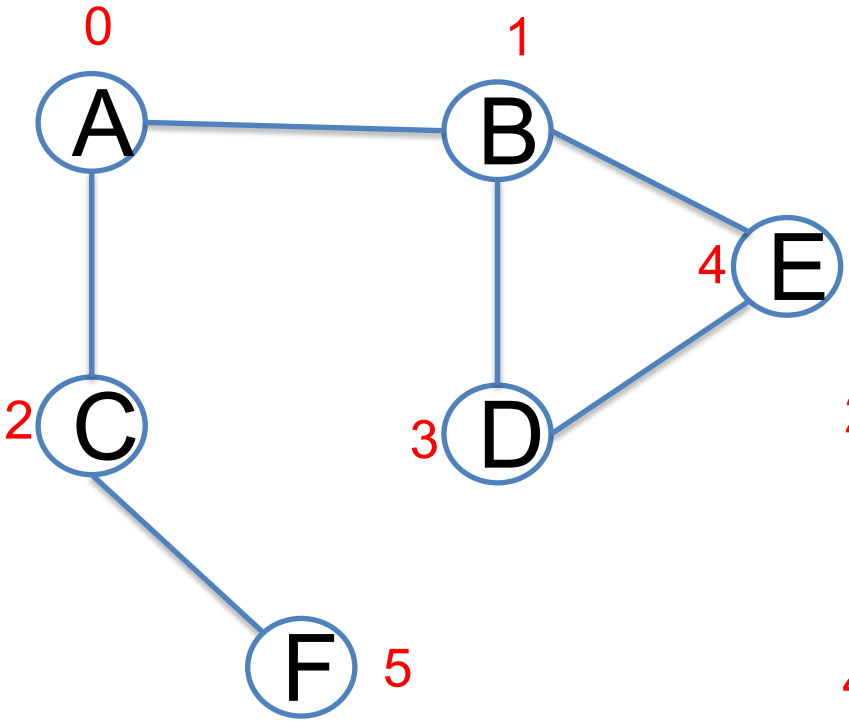


Lista de vértices

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.

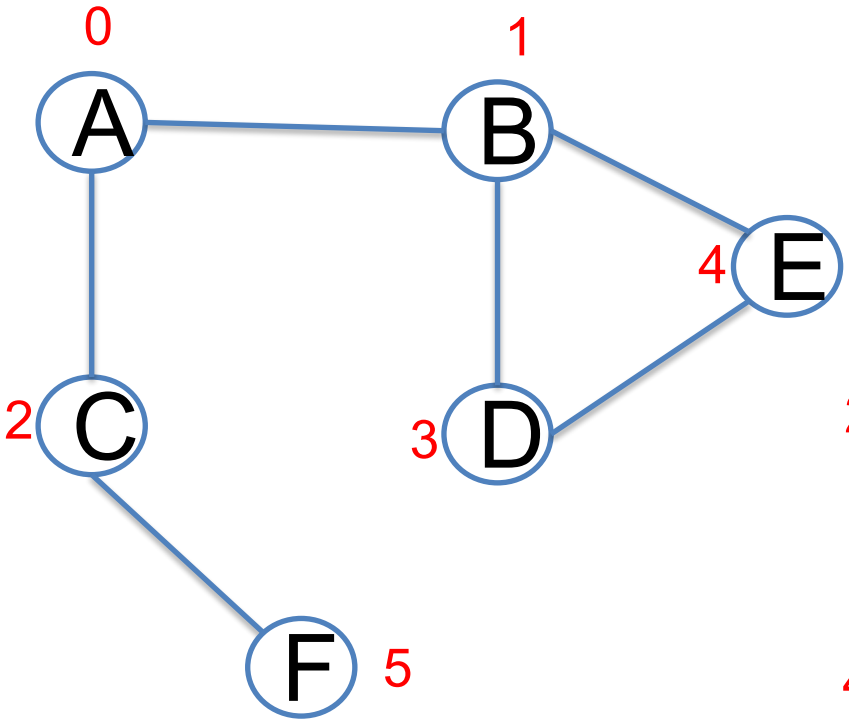
Matriz de Adyacencia (grafo no dirigido)



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

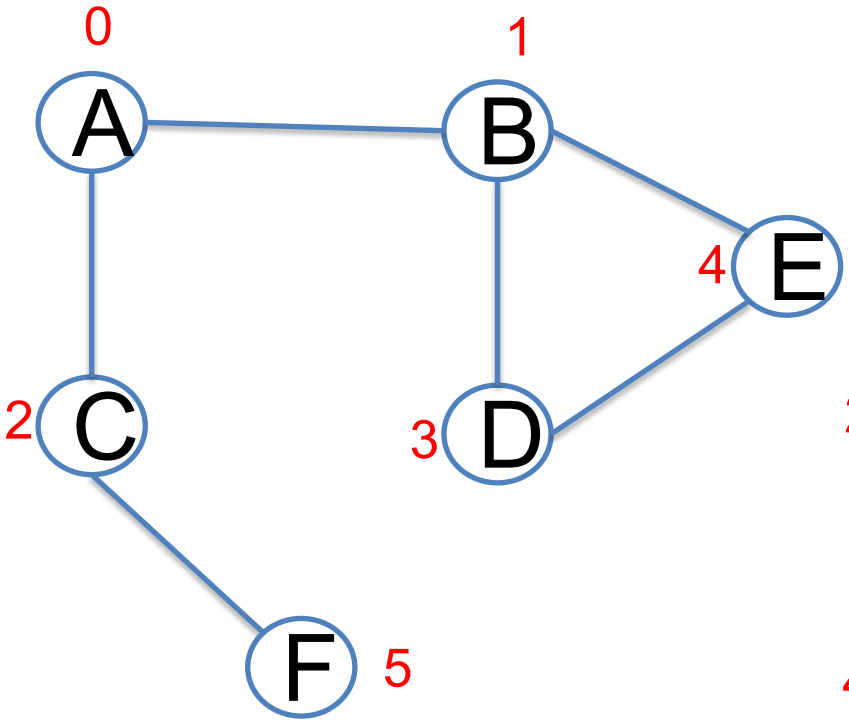
Matriz de Adyacencia (grafo no dirigido)



		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

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

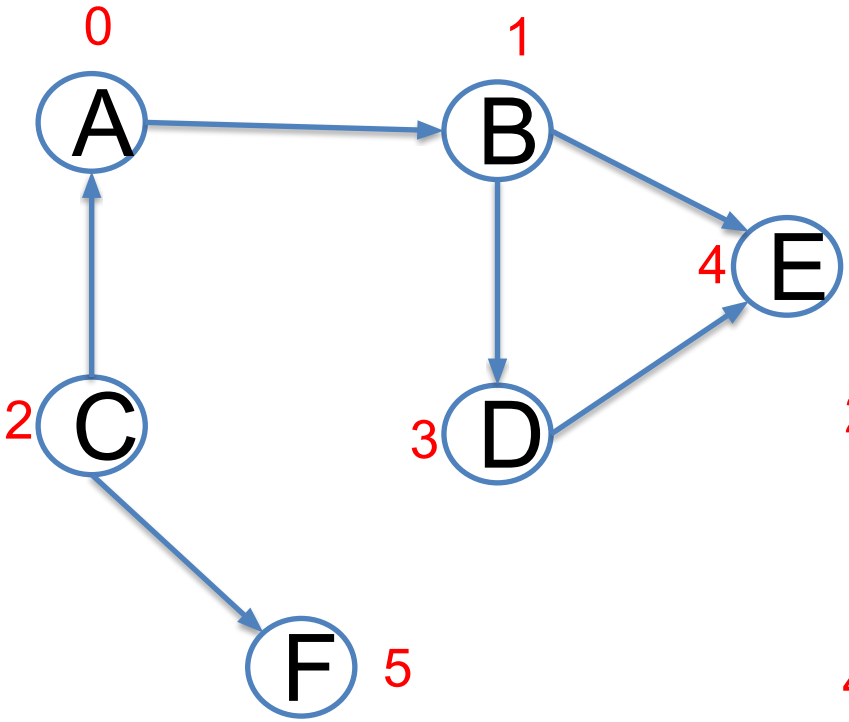
Matriz de Adyacencia (grafo no dirigido)



		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)

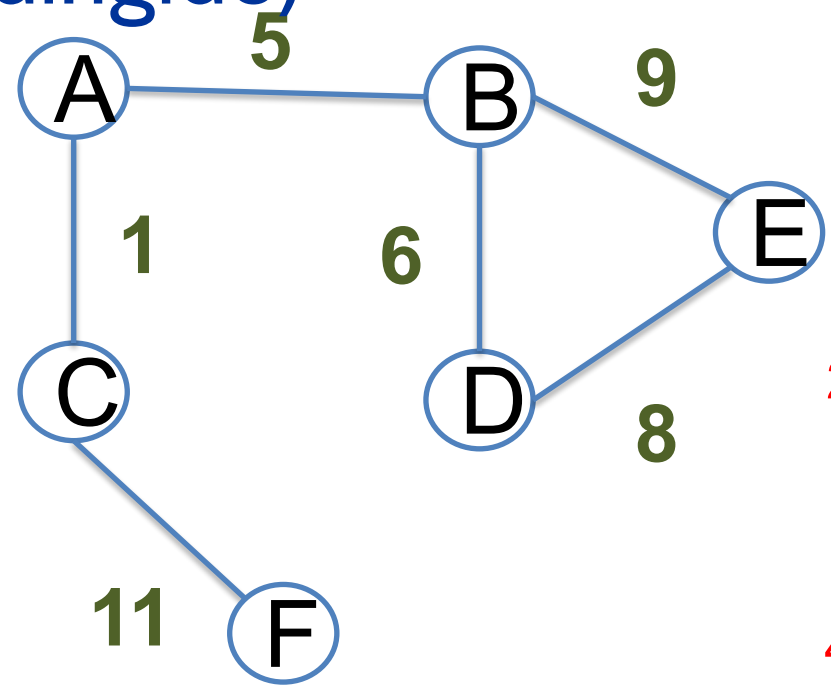


		0	1	2	3	4	5
0	A	0	1	0	0	0	0
1	B	0	0	0	1	1	0
2	C	1	0	0	0	0	0
3	D	0	0	0	0	1	0
4	E	0	0	0	0	0	0
5	F	0	0	0	0	0	0

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

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

Matriz de Adyacencia (grafo ponderado no dirigido)

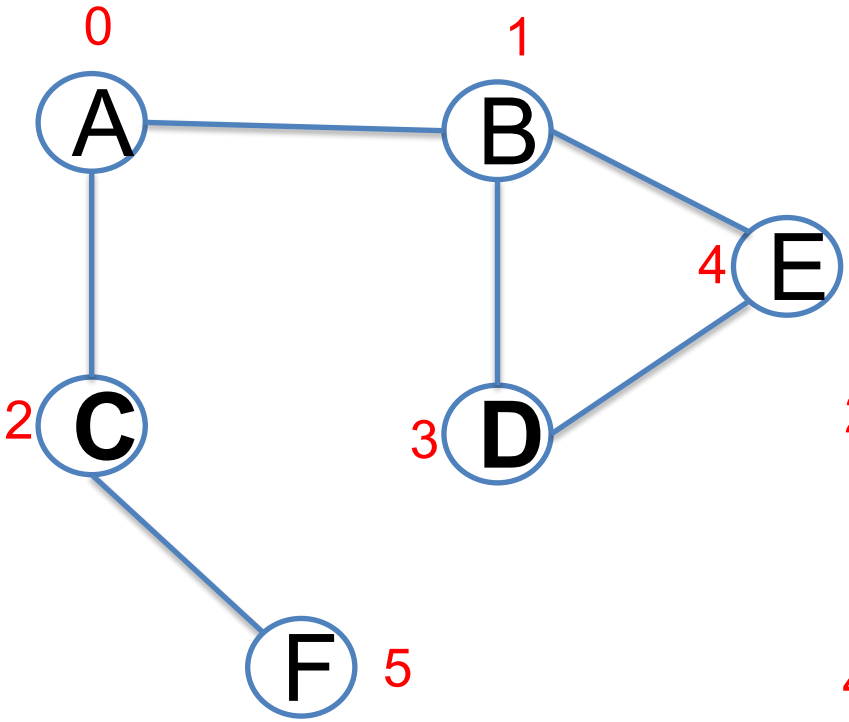


0	A
1	B
2	C
3	D
4	E
5	F

	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



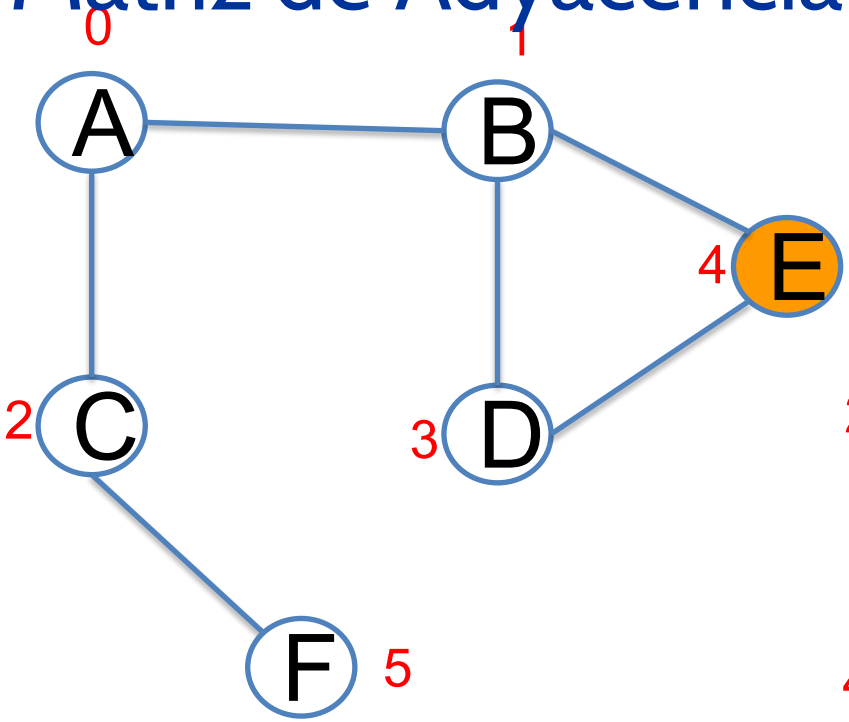
0	A
1	B
2	C
3	D
4	E
5	F

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



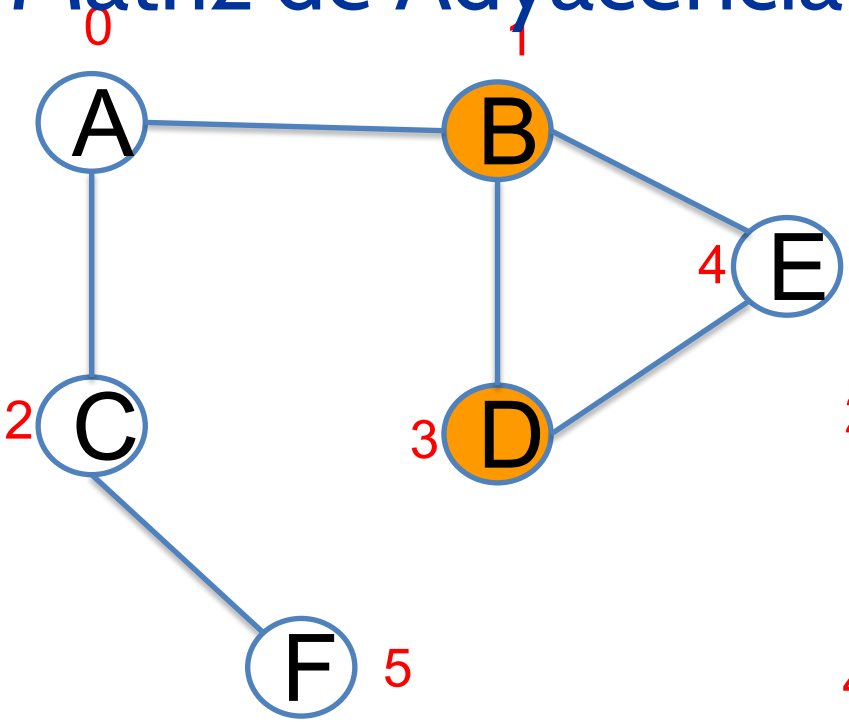
		0	1	2	3	4	5
0	A	0	1	1	0	0	0
1	B	1	0	0	1	1	0
2	C	1	0	0	0	0	1
3	D	0	1	0	0	1	0
4	E	0	1	0	1	0	0
5	F	0	0	1	0	0	0

Complejidad temporal **buscar los vecinos** de E?

O(n)

- O(n) para buscar el índice de E en la lista de vértices. En el peor de los casos, tengo que recorrer toda la lista.
- O(1) para obtener la fila de la matriz cuyo índice es el índice de E
- O(n) recorrer la fila para obtener los índices de las columnas donde el elemento de la matriz es 1. Siempre es necesario recorrer toda la fila.

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 B y D? **O(n) + O(1)**

- O(n) para buscar el índice de B en la lista de vértices. En el peor de los casos, debemos recorrer toda la lista.
- O(n) para buscar el índice de D en la lista de vértices. En el peor de los casos, debemos recorrer toda la lista.
- O(1) por acceder al elemento de la matriz con fila = índice de B, y columna = índice de D

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 (**$O(n^2)$**).

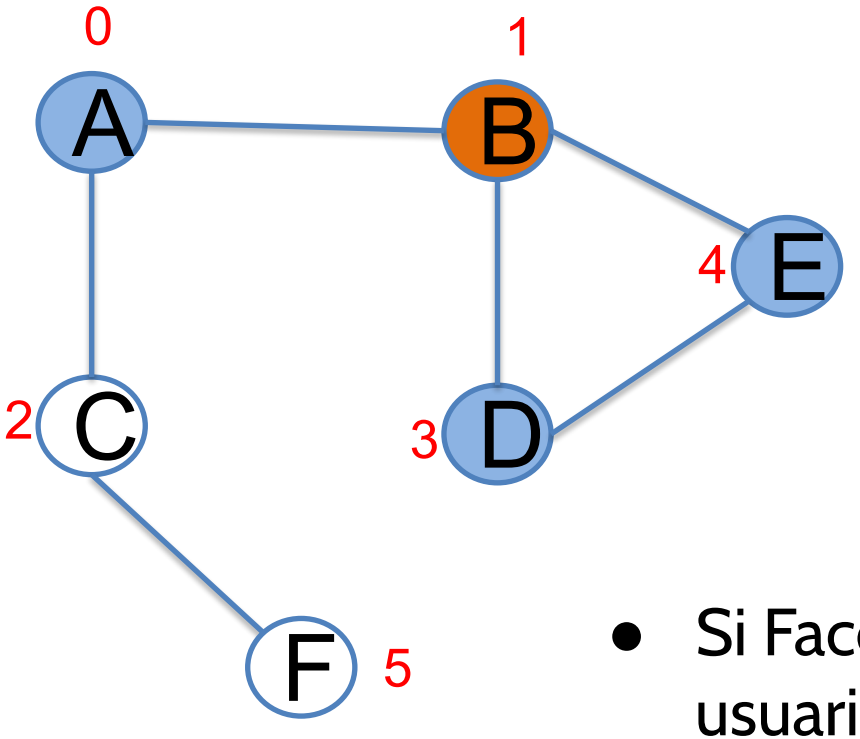
Matriz de Adyacencia - Conclusiones

- ¿Cuándo se considera una implementación adecuada?
  - **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).

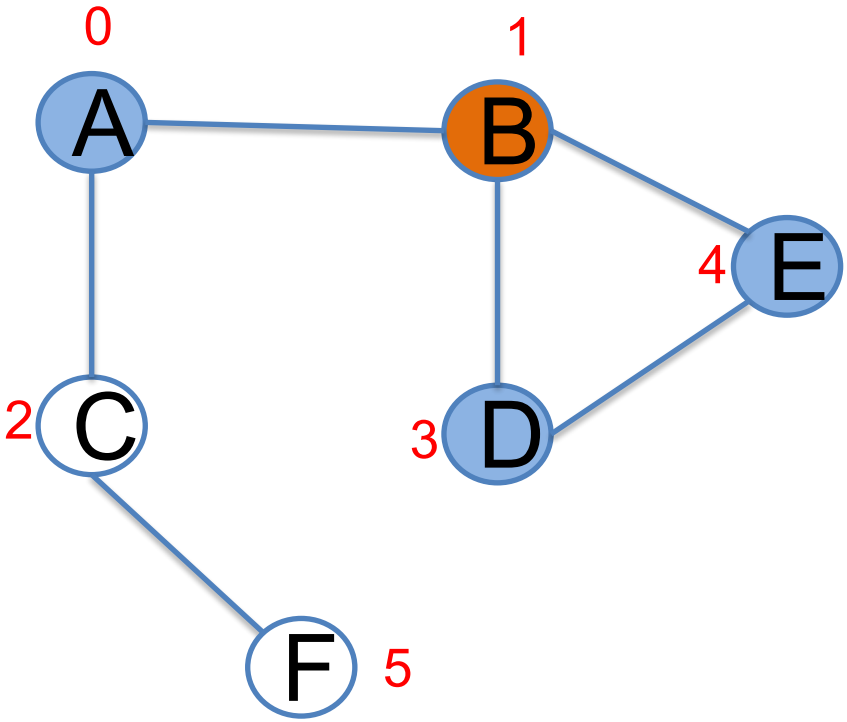
Matriz de Adyacencia - Conclusiones



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

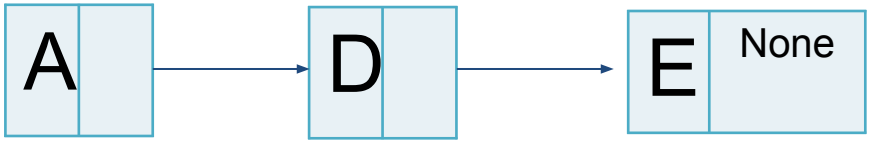
- Si Facebook tiene 1000 millones de usuarios (10^9), las filas de la matriz de adyacencia son de dimensión 10^9
- Si un usuario, B, tiene 1000 amigos, en su fila, habrá:
  - Número de 1s: 1000 = 1 KB y
  - Número de 0s: 10^9 - 1000 = 1 GB

Lista de Adyacencia

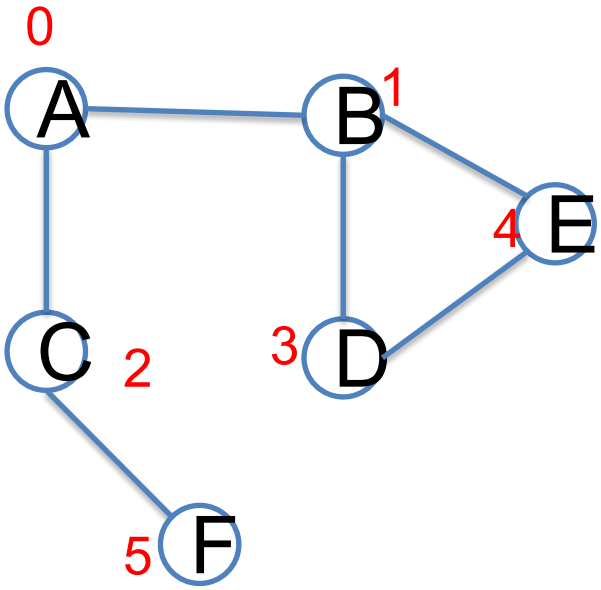


Para representar los **vecinos de B,** sería suficiente con almacenar sus índices en un **lista de python** o en una **lista enlazada**.

Vecinos de B:



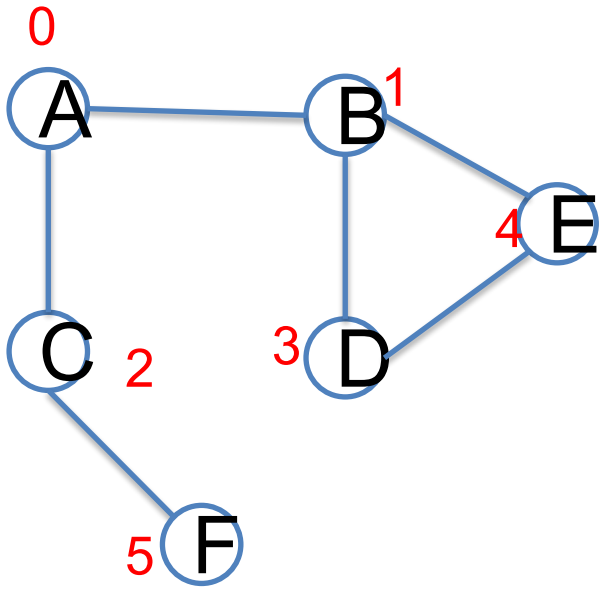
Lista de Adyacencia

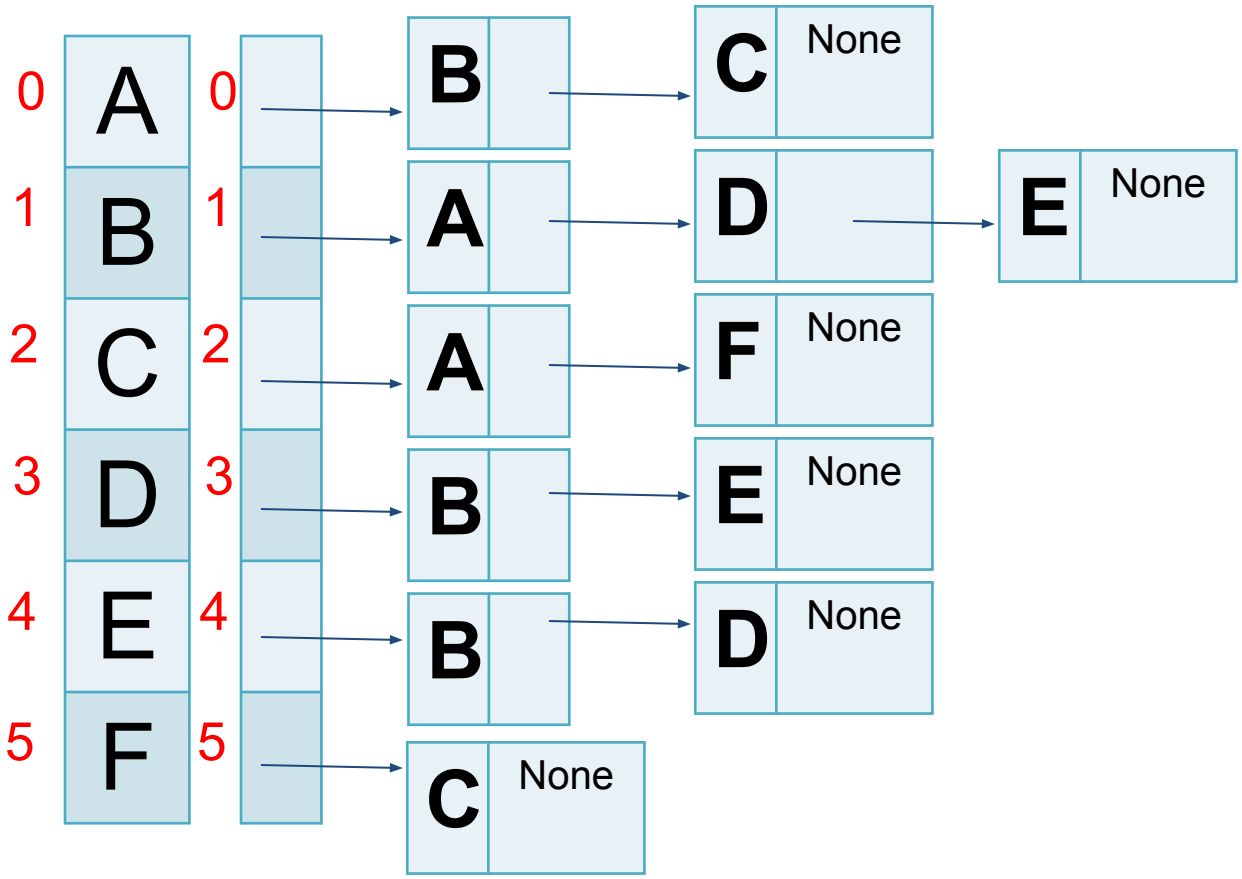


0	A
1	B
2	C
3	D
4	E
5	F

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

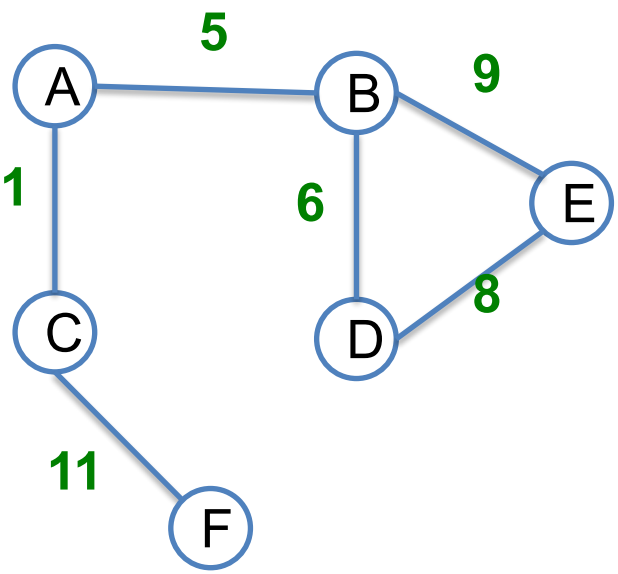
Lista de Adyacencia

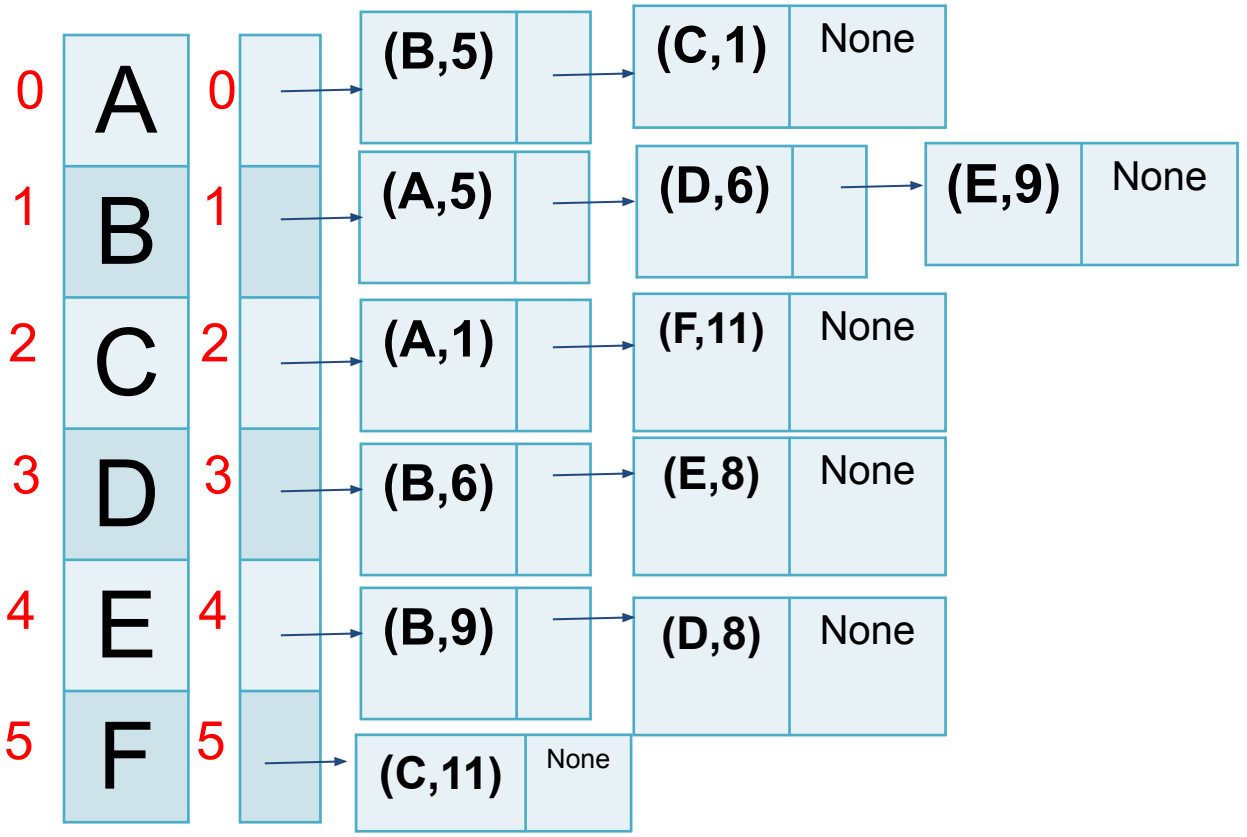




Además, usaremos una segunda lista de Python de listas enlazadas para almacenar los vértices adyacentes a cada vértice.

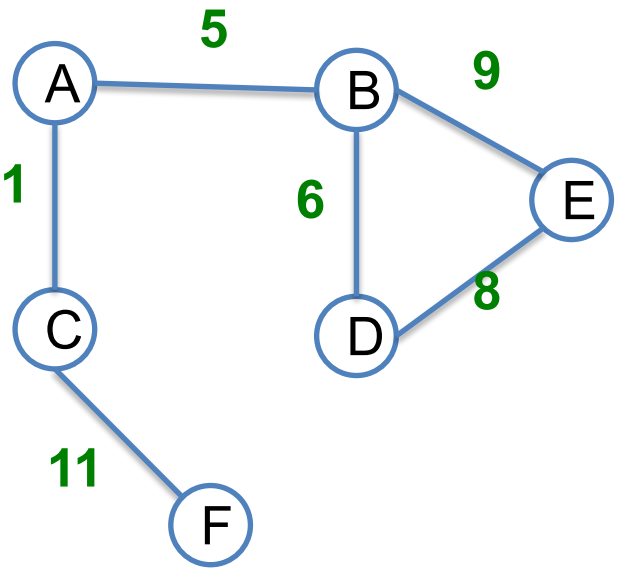
Lista de Adyacencia (grafo ponderado)

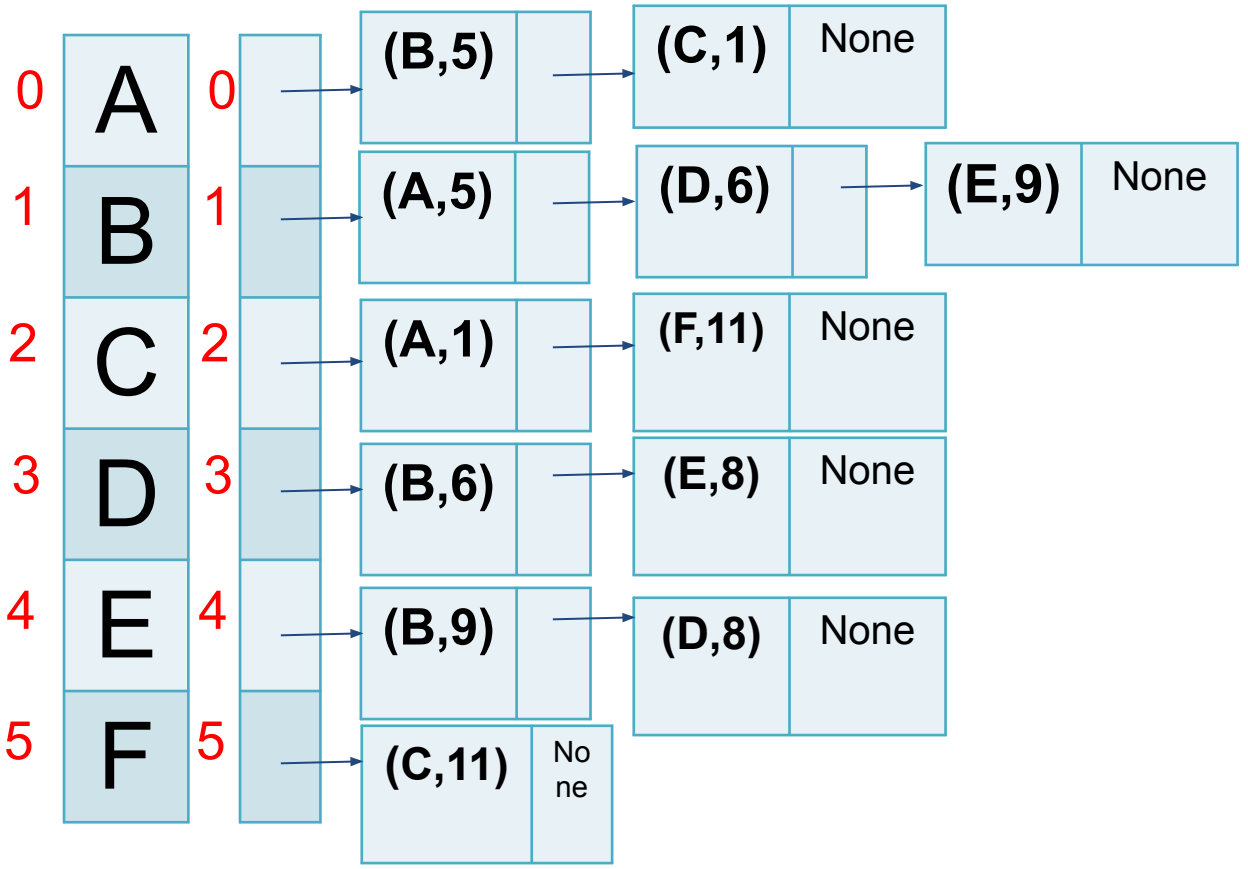




Cada vértice adyacente se representa como un **par (v,w)** donde **v es el vértice adyacente**, y **el peso de la arista**.

Lista de Adyacencia - Complejidad Espacial

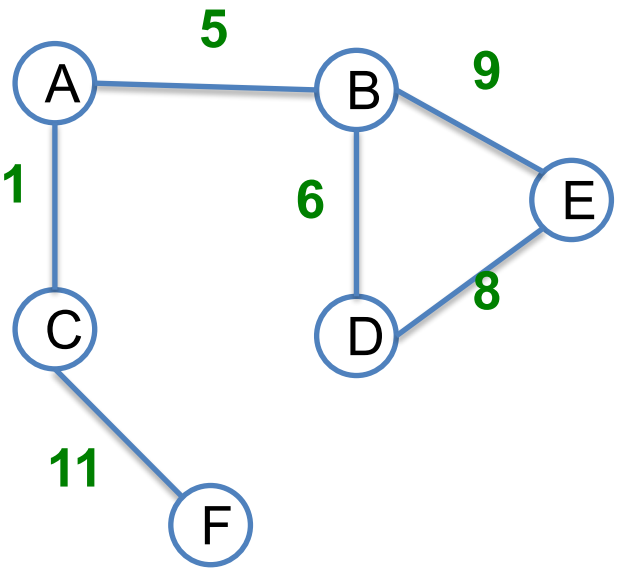


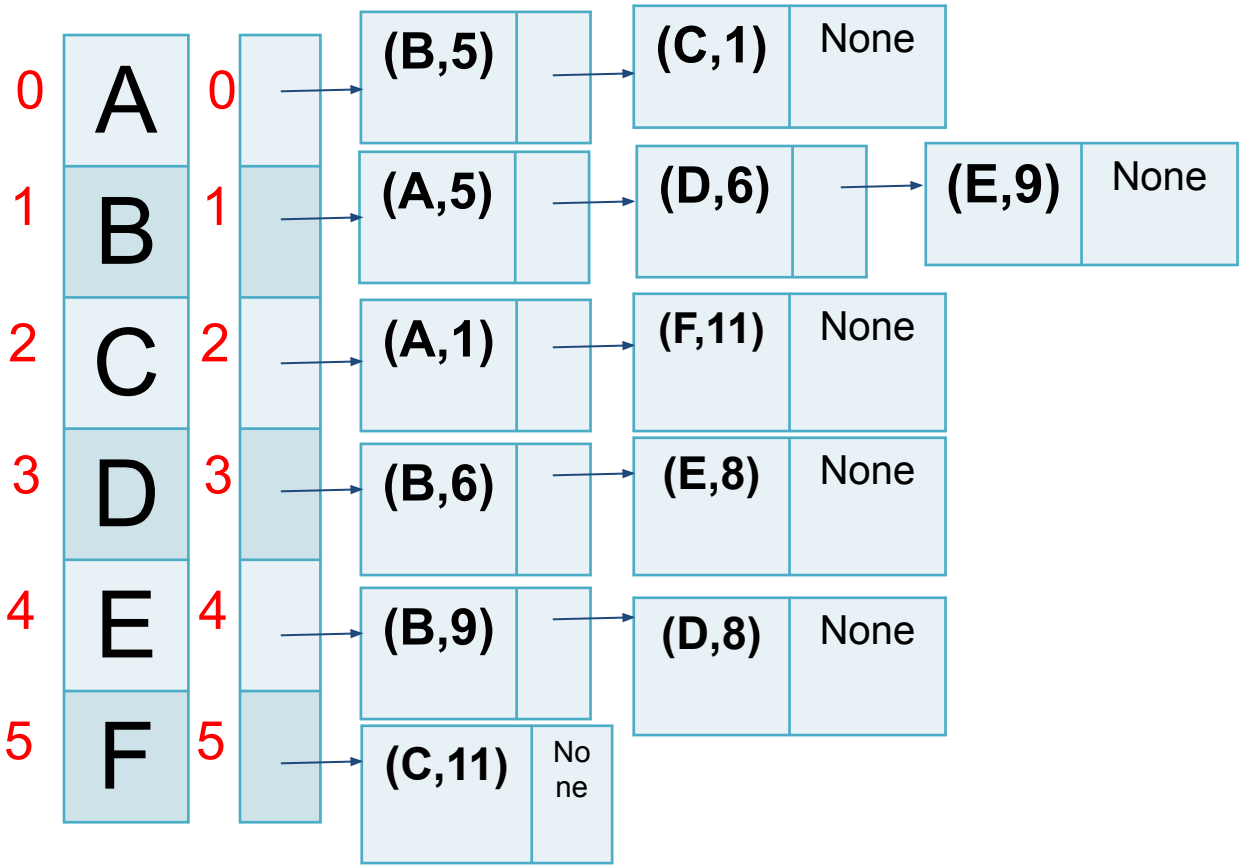


O(|V| + |A|)

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

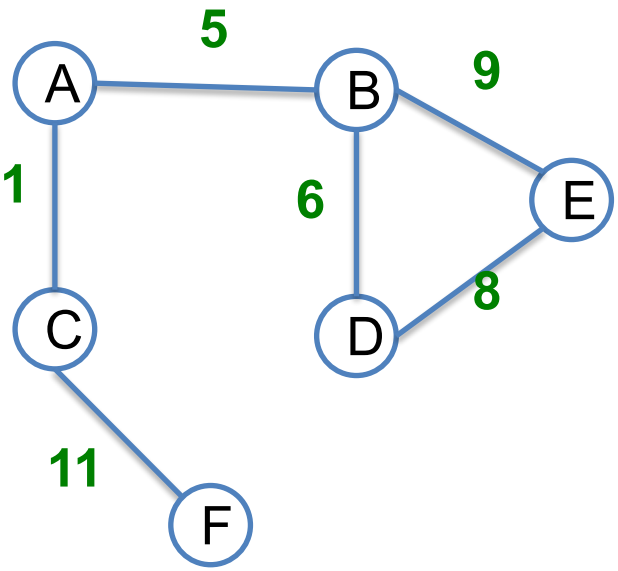


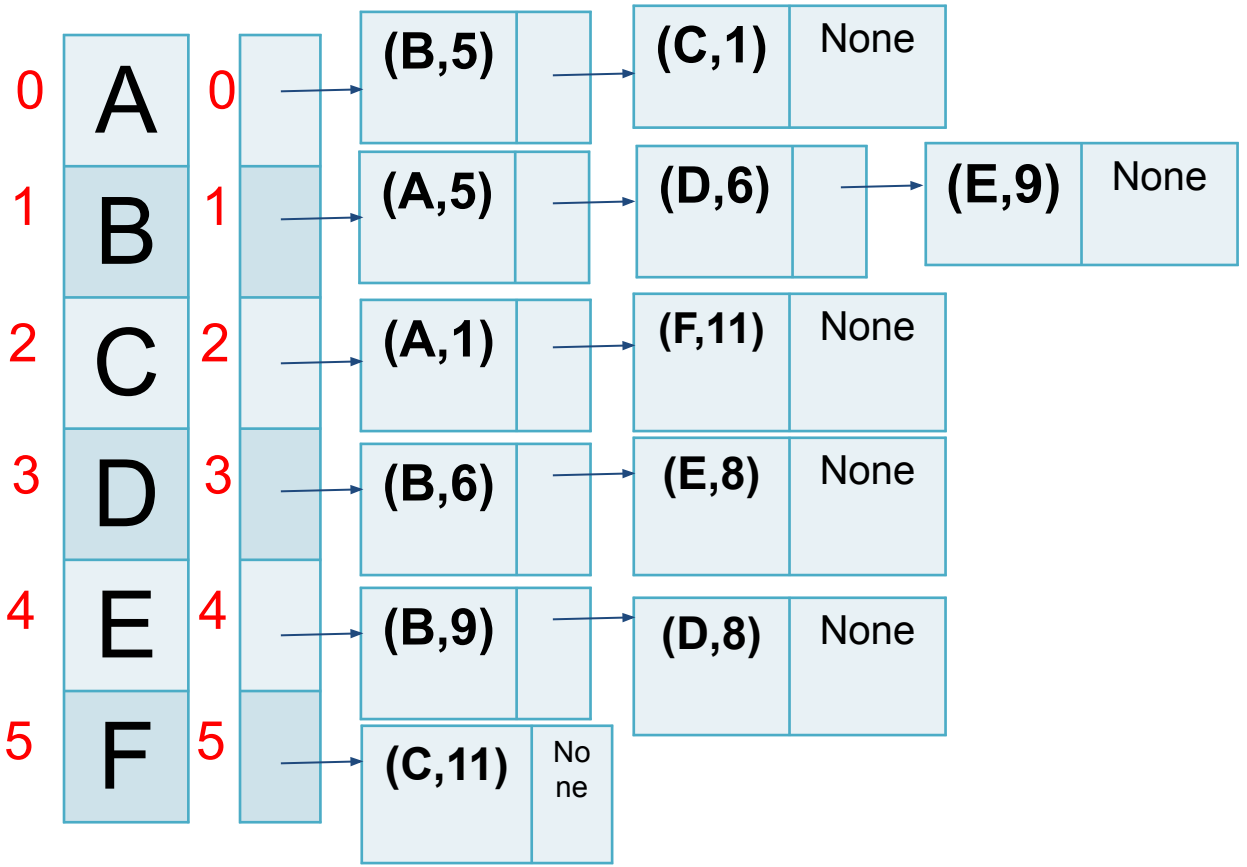


Si el grafo es denso: O(|V|+|A|) -> n + n^2 -> **$O(n^2)$**

Si el grafo es escaso: O(|V|+|A|) -> n+n = 2n -> **O(n)**

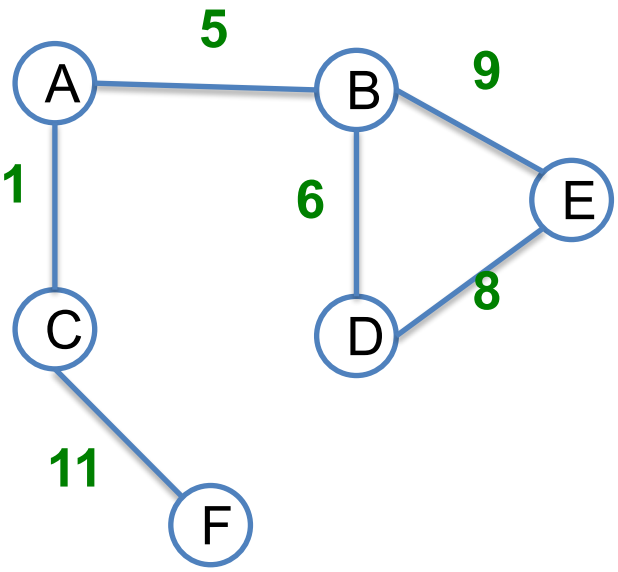
Lista de Adyacencia - Complejidad Temporal

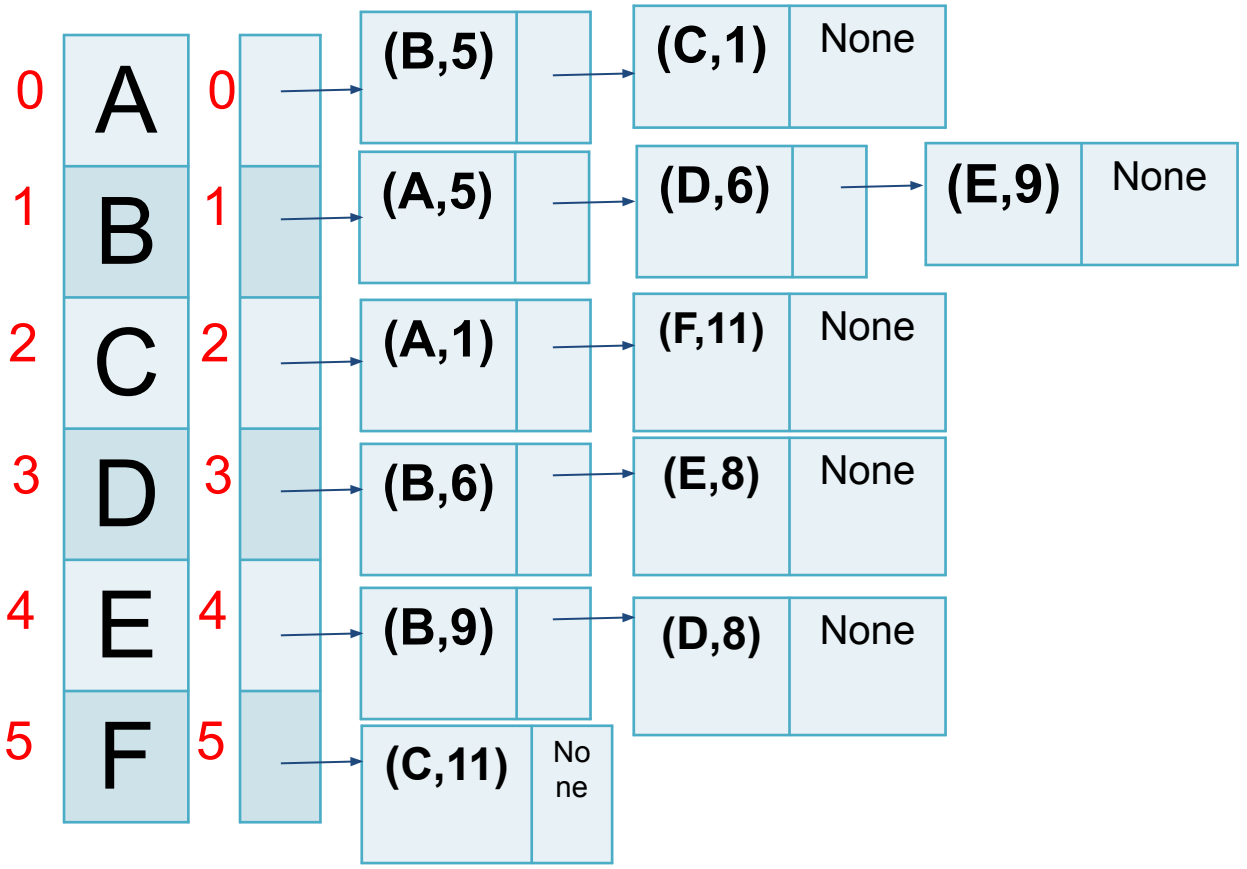




Obtener vértices adyacentes a E?
O(n) (obtener su índice=4) y O(1) devolver la lista de adyacencia asociada al índice

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.

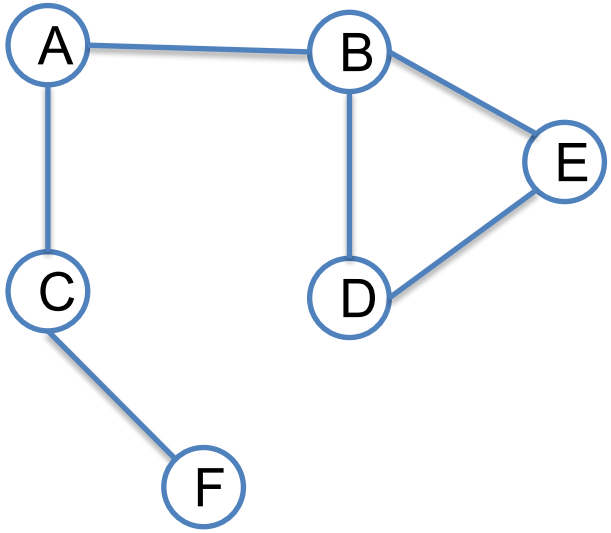
Lista de Adyacencia - Conclusiones

- 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(|V|+|A|)** -> n, si el grafo es escaso.

Índice

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

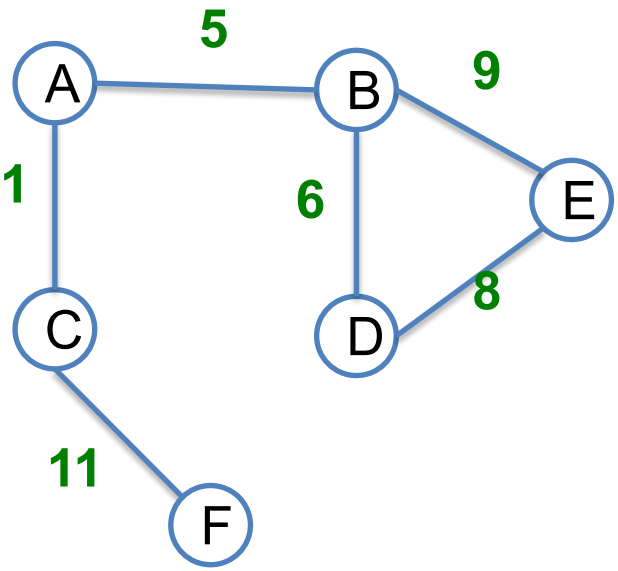
Diccionarios




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

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

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

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

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

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

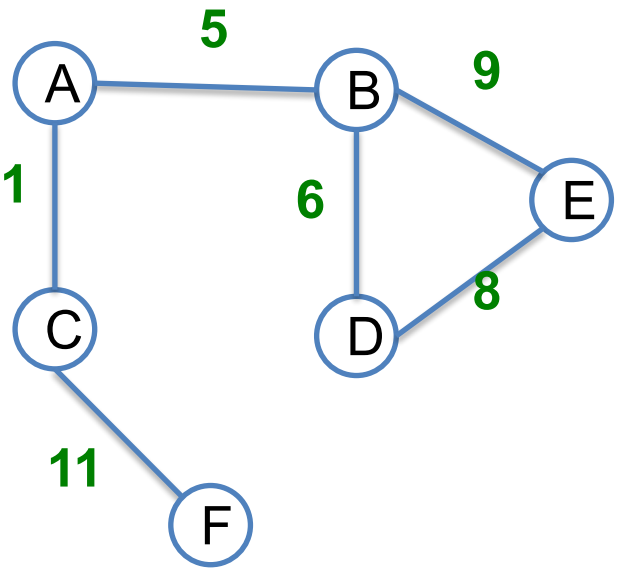
Los vértices del grafo se van a representar como las claves (keys)del diccionario.
En el diccionario, asociado a cada clave (vértice), se guarda la lista de sus vértices adyacentes

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 (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)] }

Aunque un vértice adyacente y el peso de la arista, pueden representarse directamente como una tupla, nosotros vamos a definir la siguiente clase **AdjacentVertex**:

```
class AdjacentVertex:
    def __init__(self, vertex: object, weight: int = 1) -> None:
        self.vertex = vertex
        self.weight = weight
```

El uso de la clase AdjacentVertex nos aporta una mayor legibilidad y semántica, porque utiliza nombres explícitos como vertex y weight, para referirnos a al vértice adyacente y al peso de la arista.
Además, hace que el código sea más fácil de mantener y extender.
La lista asociada a graph['A'] contiene dos objetos AdjacentVertex:

- El primer objeto tiene vertex='B', y weight=5
- El segundo objeto tiene vertex='C', y weight = 1

Diccionarios - Implementación

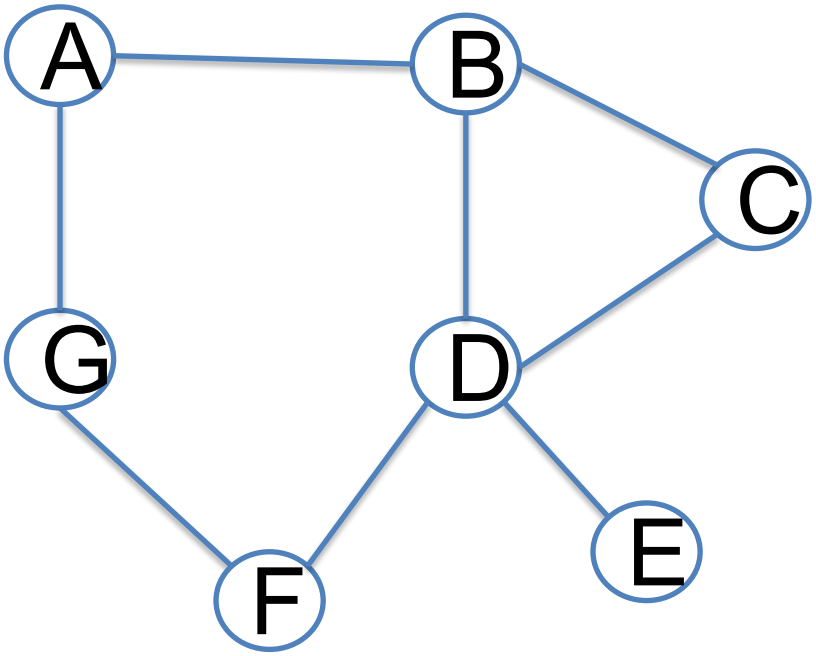
- Implementación de grafo (cualquier tipo) basado en diccionarios de Python (recomendada).

Índice

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

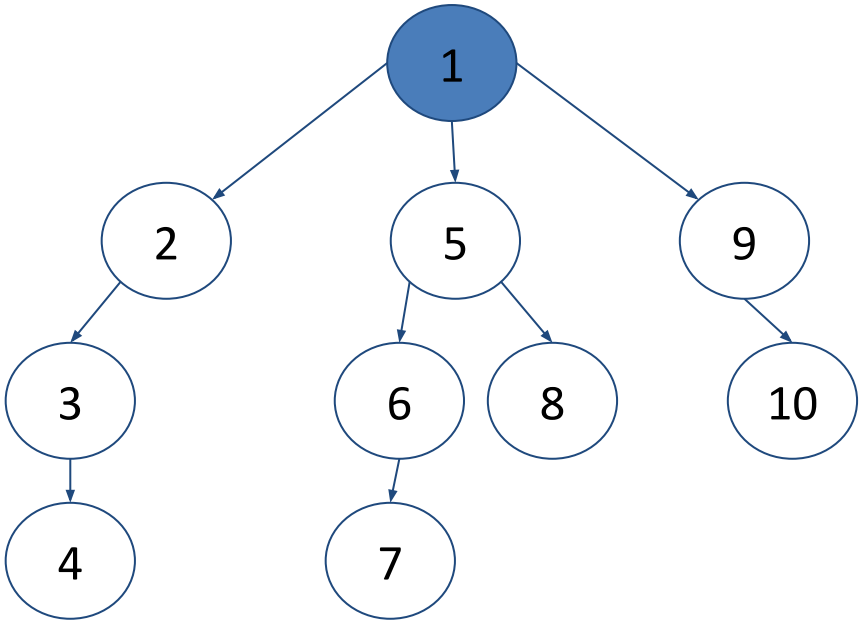
Recorridos (Graph Traversals)

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



Breadth first search

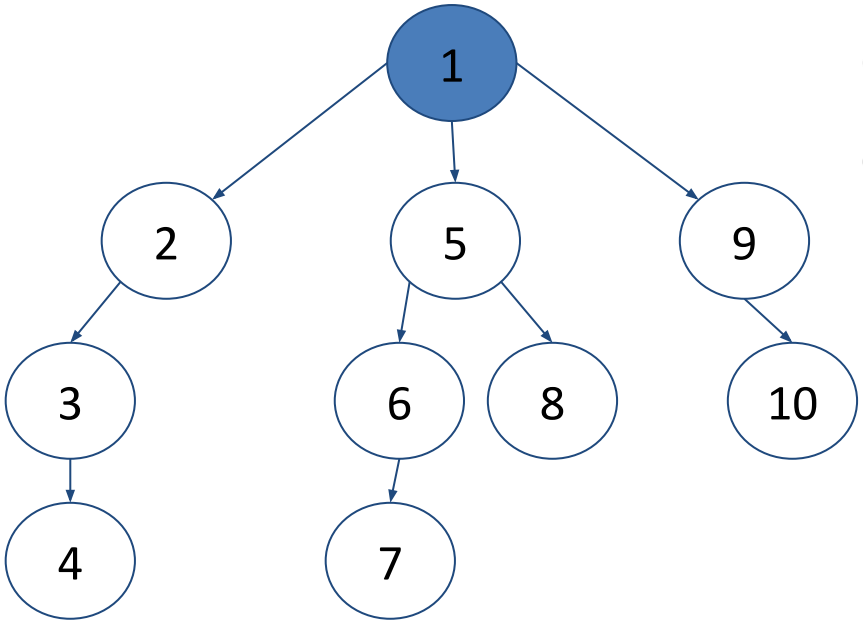
Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



El recorrido en amplitud comienza por un vértice dado (puede ser cualquier vértice) y explora todos sus nodos vecinos. Para cada uno de estos nodos, el algoritmo explora de nuevo sus nodos vecinos. Esto se continúa hasta que se han recorrido todos los nodos.

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



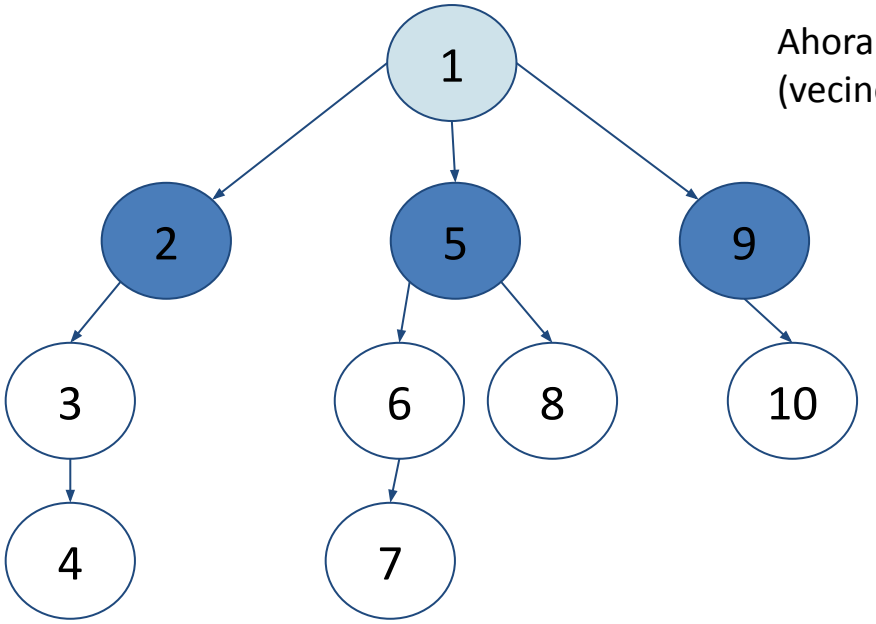
Comenzamos visitando el vértice 1 (podríamos empezar por cualquier otro vértice).

Recorrido: 1

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



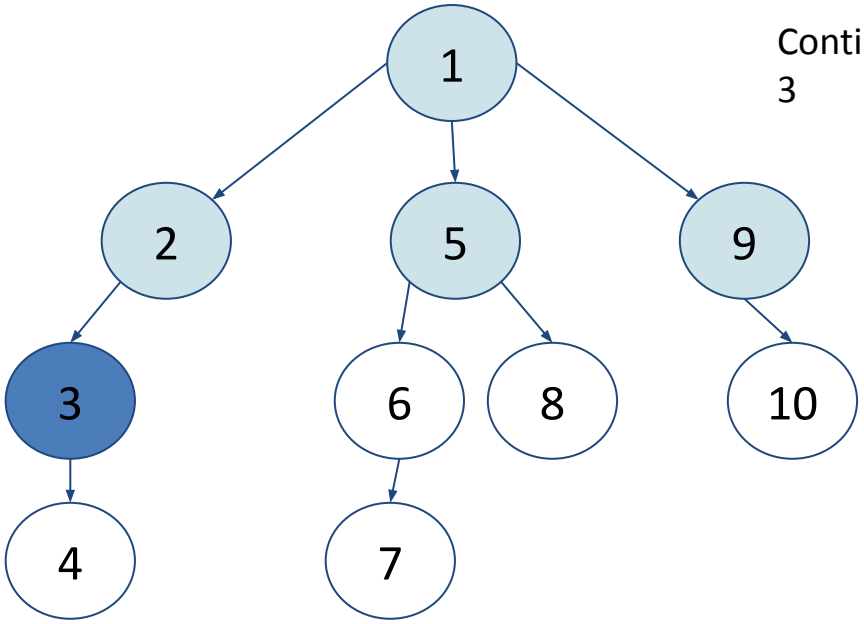
Ahora visitamos todos los adyacentes (vecinos) a 1: 2,5 y 9

Recorrido: 1, **2, 5, 9**

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



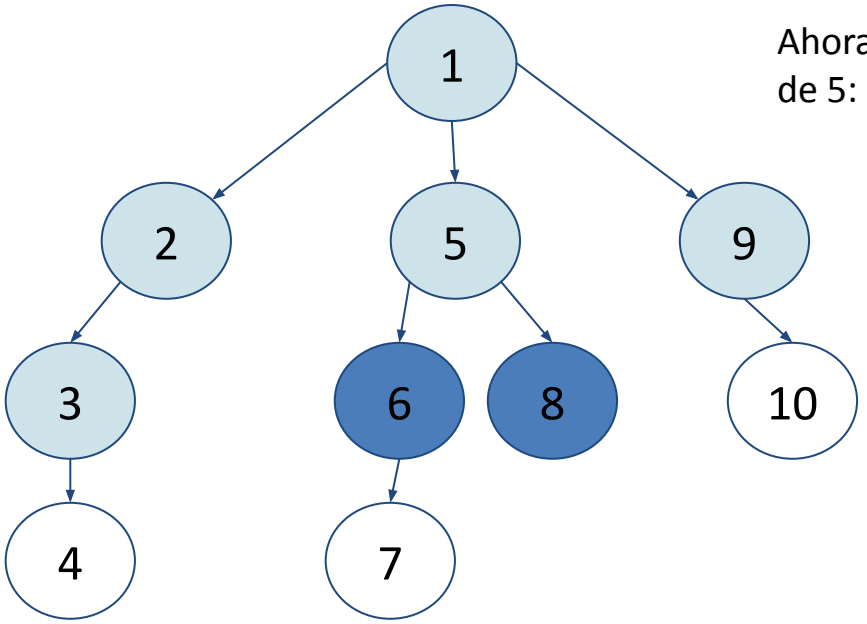
Continuamos con los adyacentes de 2: 3

Recorrido: 1, 2, 5, 9, **3**

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



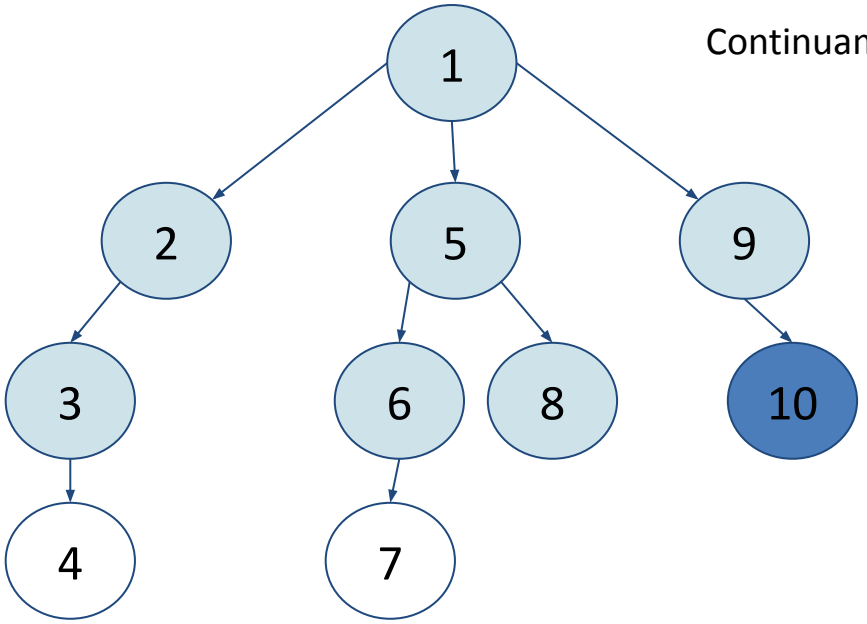
Ahora debemos visitar los adyacentes de 5: 6 y 8

Recorrido: 1, 2, 5, 9, 3, **6, 8**

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



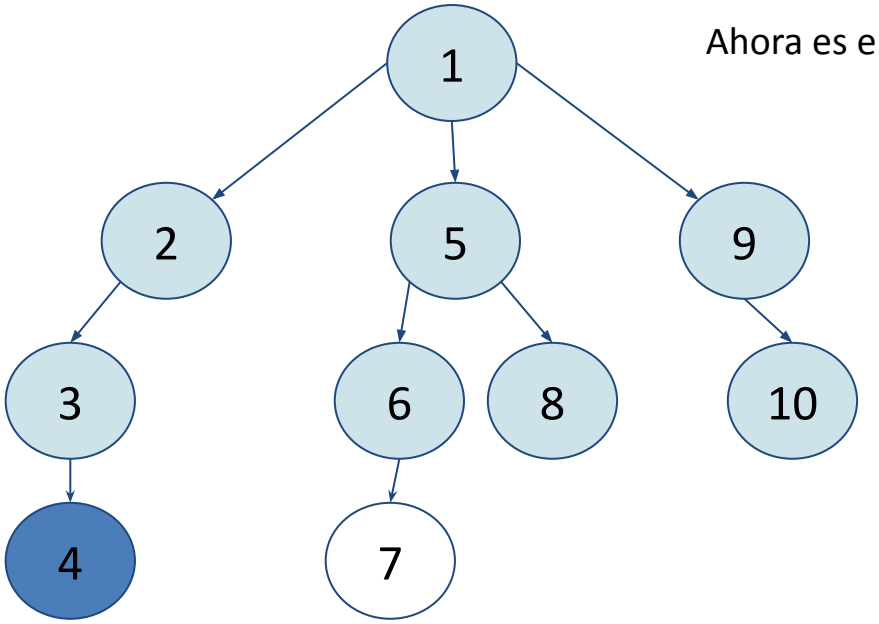
Continuamos con los adyacentes de 9: 10

Recorrido: 1, 2, 5, 9, 3, 6, 8, **10**

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



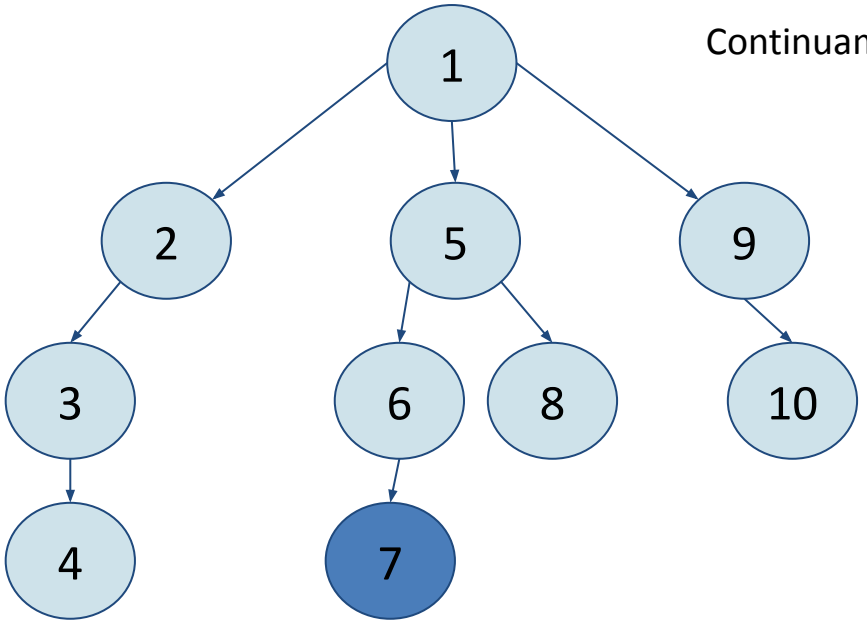
Ahora es el turno de los adyacentes de 3: 4

Recorrido: 1, 2, 5, 9, 3, 6, 8, 10, **4**

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



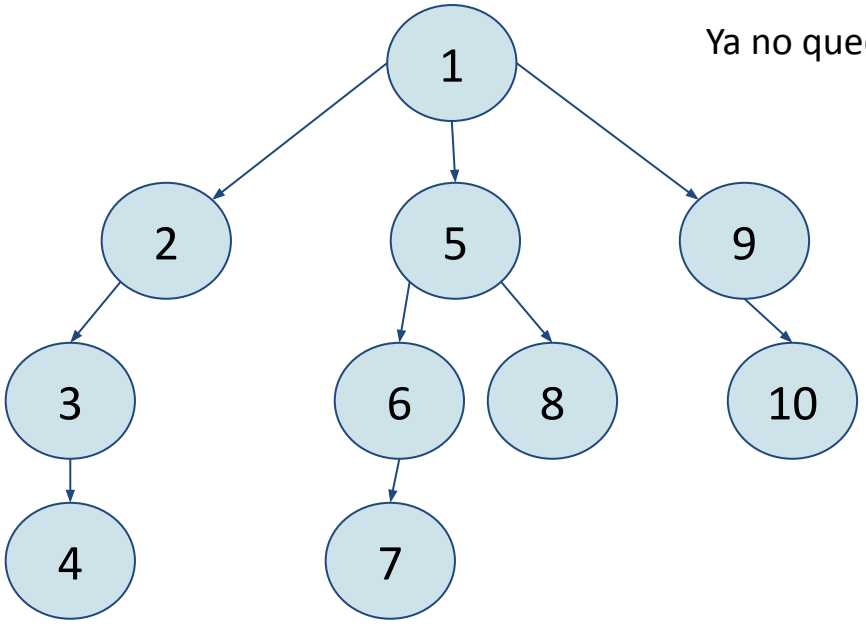
Continuamos con los adyacentes de 6: 7

Recorrido: 1, 2, 5, 9, 3, 6, 8, 10, 4, **7**

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search

Recorrido en anchura (Breadth first search) o amplitud: similar al recorrido por niveles en árboles.



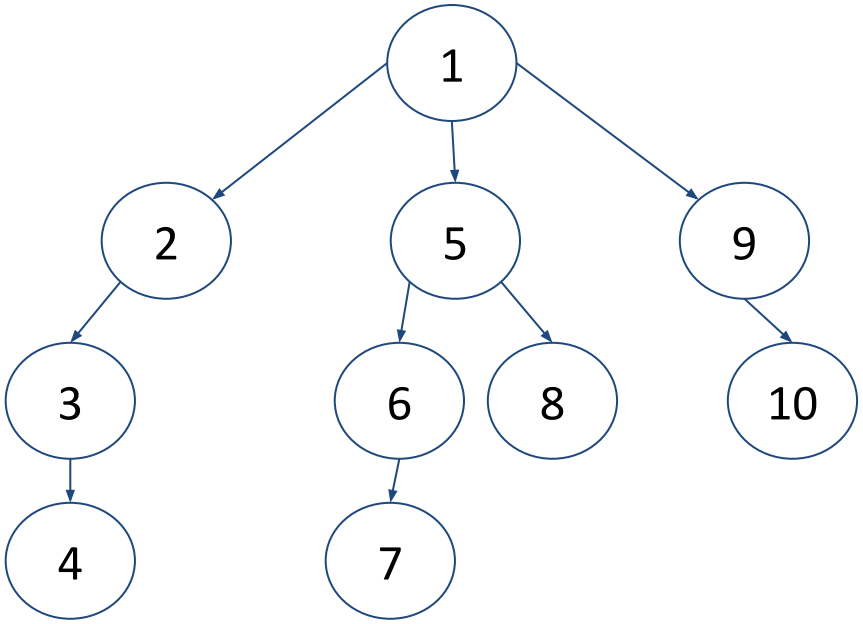
Ya no quedarían más nodos por recorrer.

Recorrido: 1, 2, 5, 9, 3, 6, 8, 10, 4, 7

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

Breadth first search - Implementación

Implementación similar a árboles. Usamos una cola para almacenar los vértices adyacentes del nodo que estamos visitando.

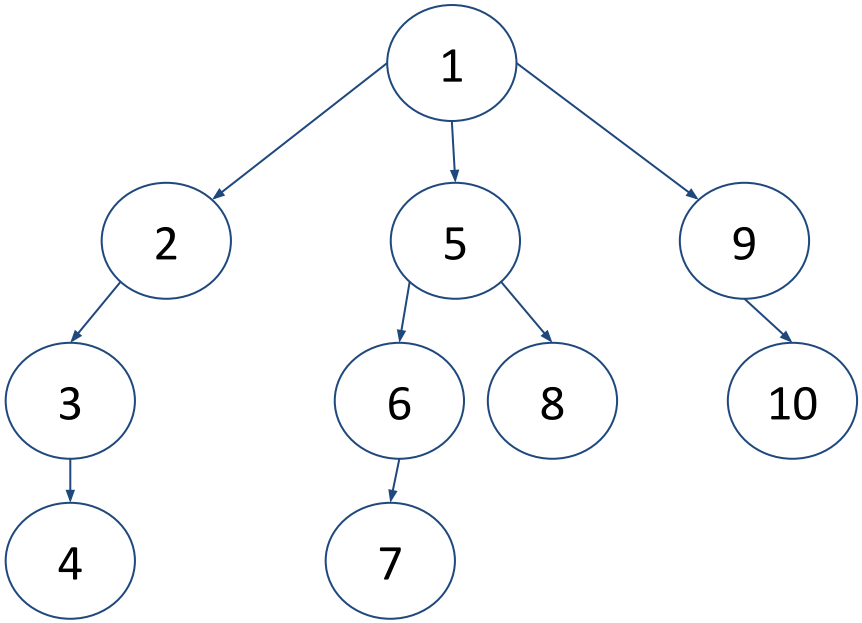


Comenzamos añadiendo a la cola el vértice por el que queremos comenzar el recorrido

Cola:

1			

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

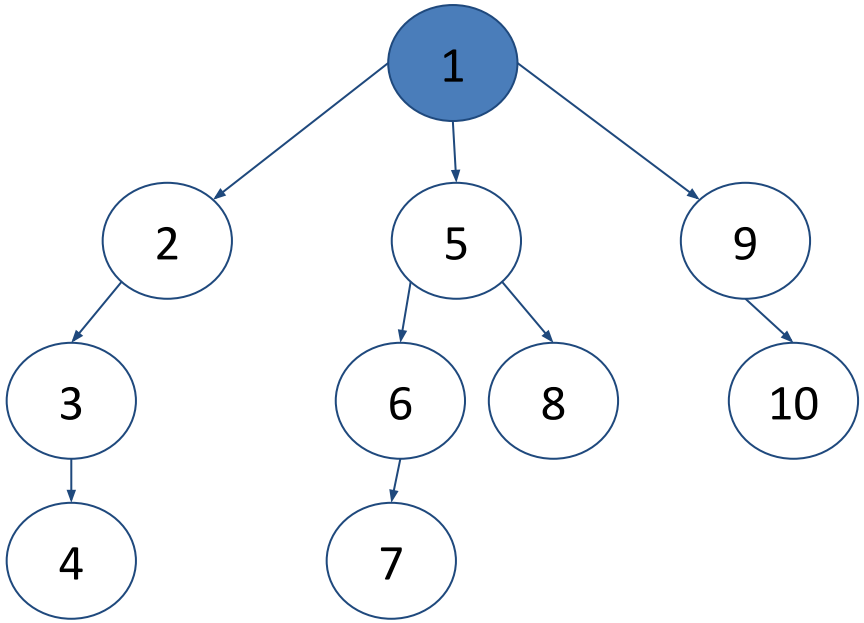
1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

1			

La cola no está vacía

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

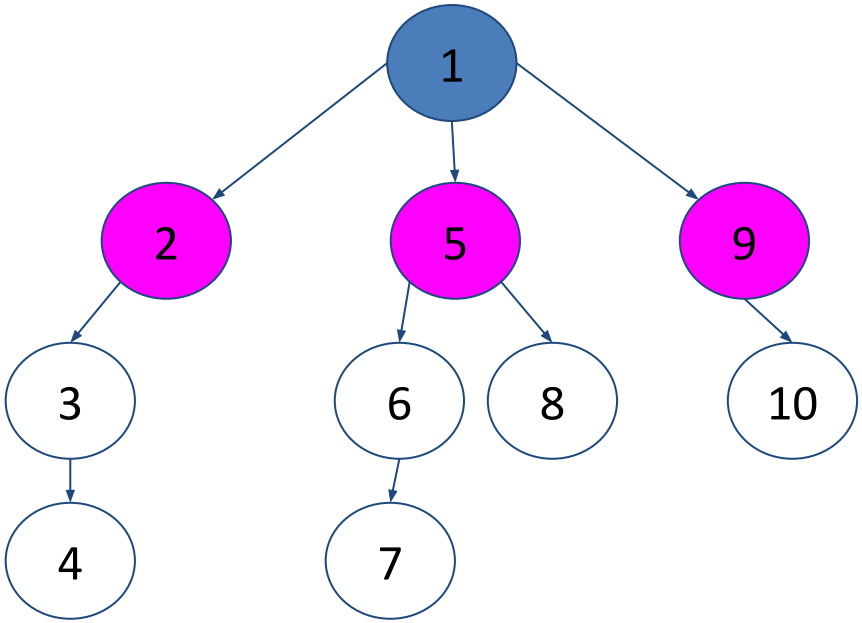
Cola: | ~~1~~ | | | |

v = cola.dequeue() -> 1

Cola: | | |

recorrido: | 1 | | | |

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

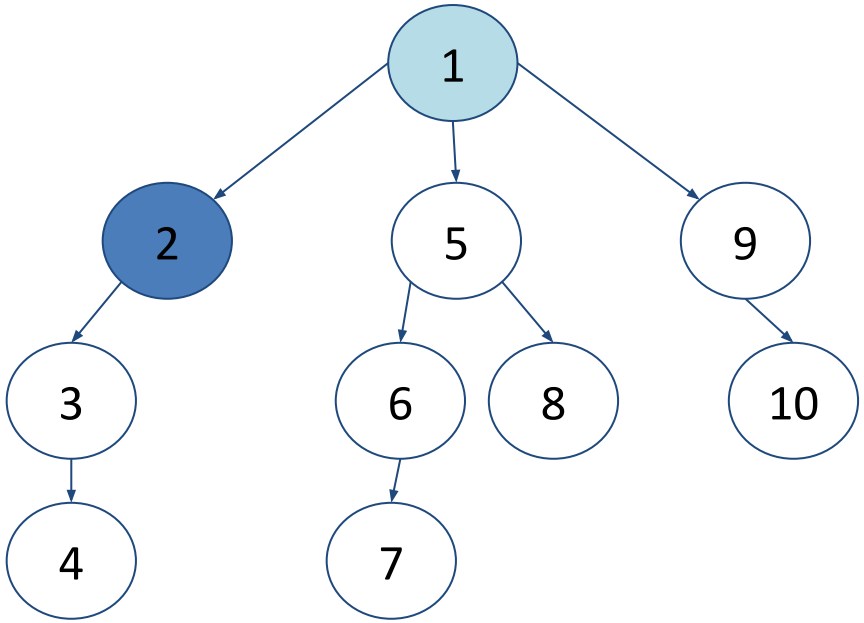
1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola: | |

Debemos añadir a la cola los adyacentes de 1: 2, 5 y 9

Cola: | **2** | **5** | **9** | |

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

~~2~~	5	9	

No está vacía

v = cola.dequeue() -> 2

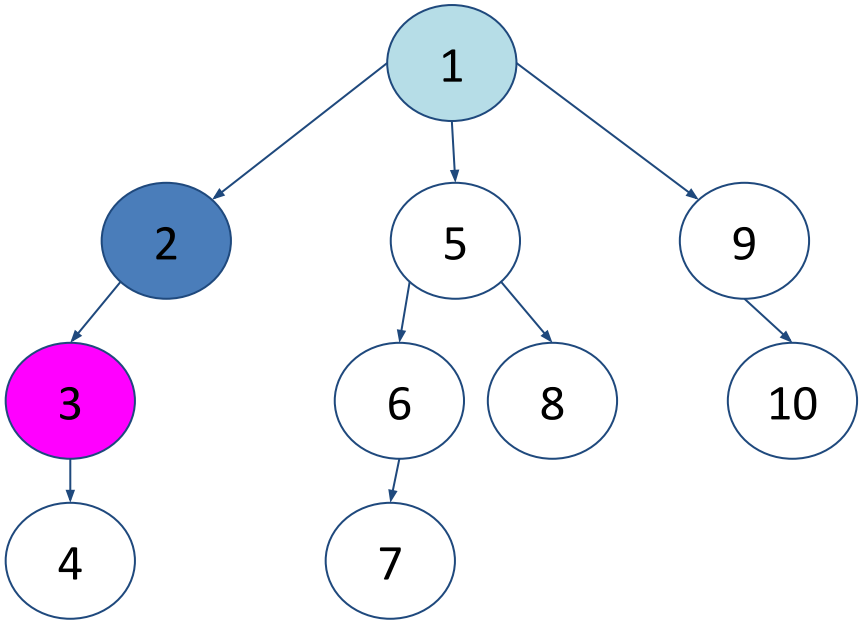
Cola:

5	9		

recorrido:

1	2		

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

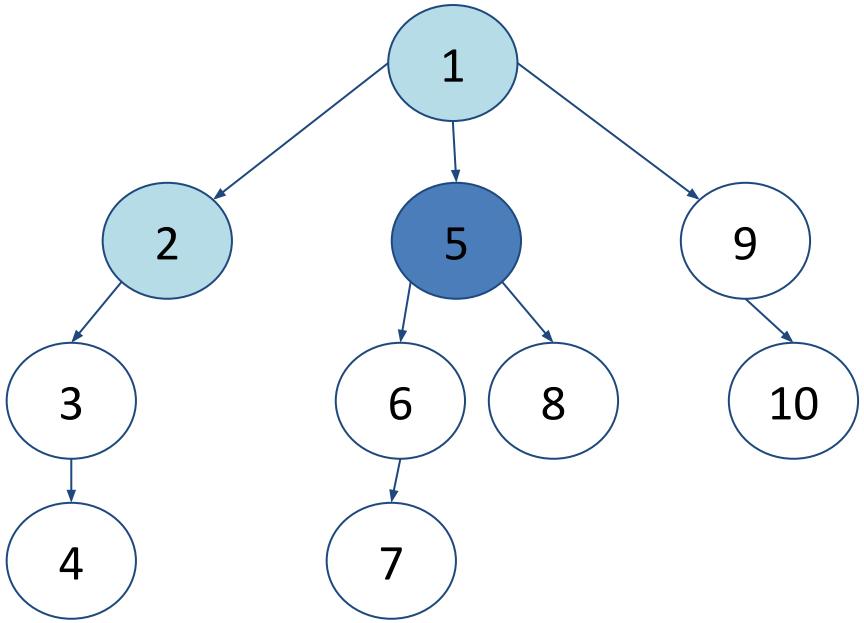
5	9		

Debemos añadir a la cola los adyacentes de 2: 3

Cola:

5	9	**3**	

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

~~5~~	9	3	

v = cola.dequeue() -> 5

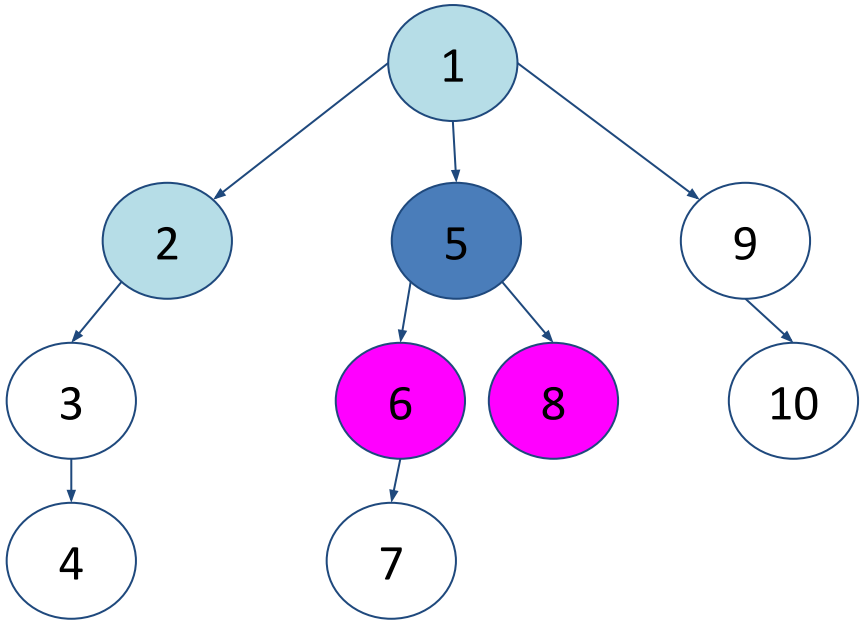
Cola:

9	3		

recorrido:

1	2	5	

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

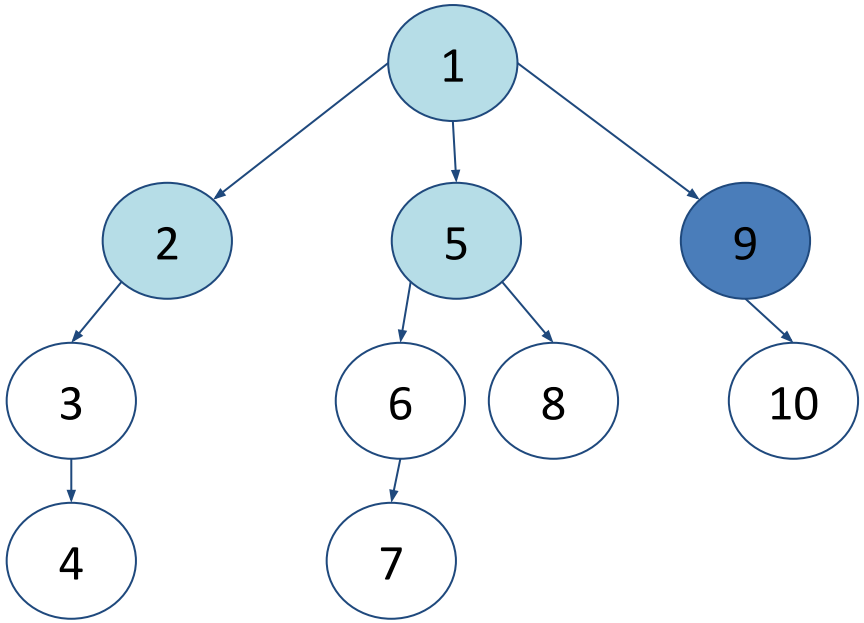
9	3

Debemos añadir a la cola los adyacentes de 5: 6 y 8

Cola:

9	3	**6**	**8**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

~~9~~	3	6	8

No está vacía

v = cola.dequeue() -> 9

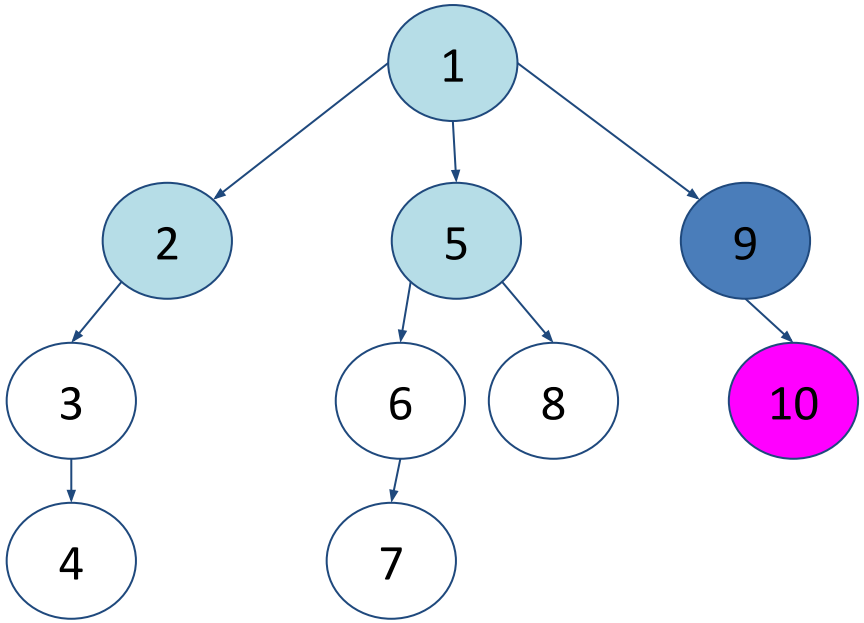
Cola:

3	6	8	

recorrido:

1	2	5	**9**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

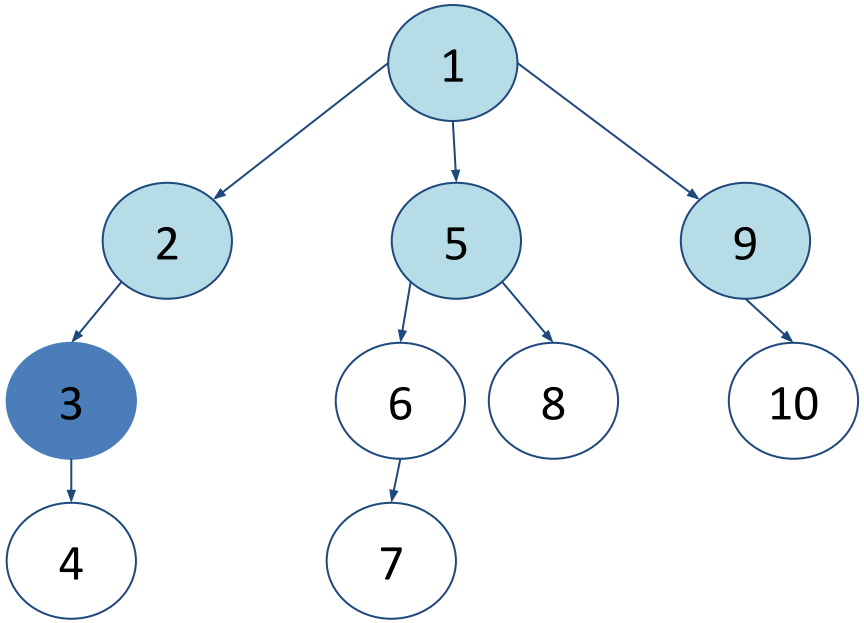
3	6	8	

Debemos añadir a la cola los adyacentes de 9: 10

Cola:

3	6	8	**10**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

3	6	8	10

No está vacía

v = cola.dequeue() -> 3

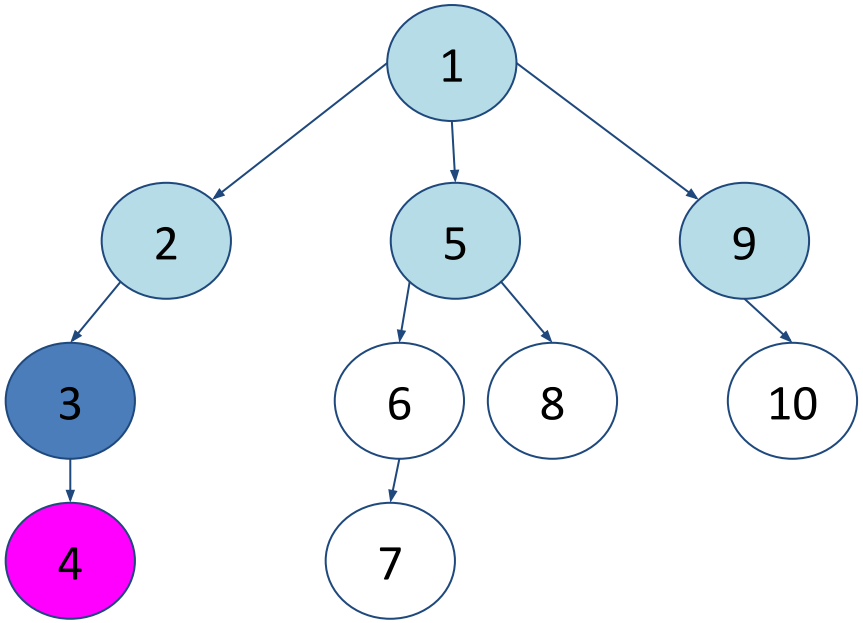
Cola:

6	8	10	

recorrido:

1	2	5	9	**3**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

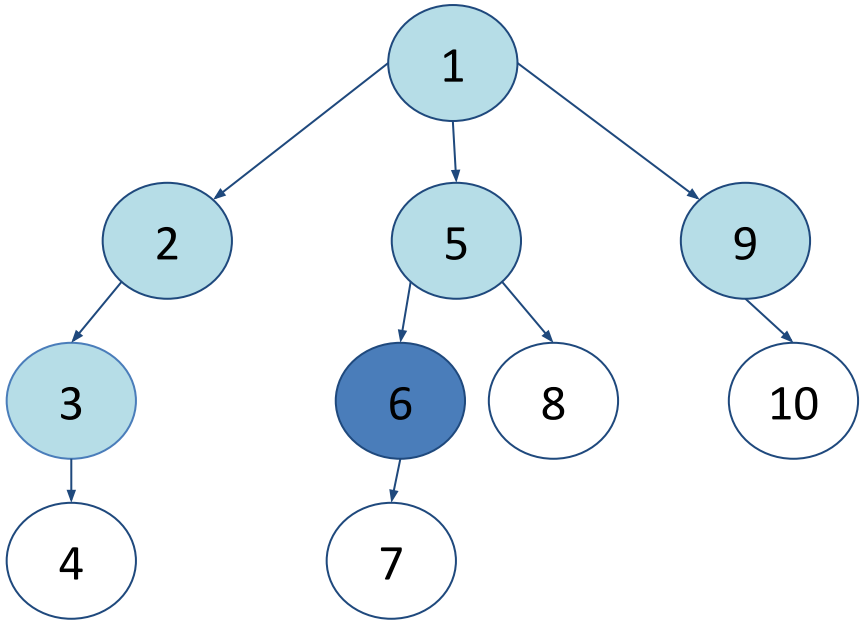
6	8	10	

Debemos añadir a la cola los adyacentes de 3: 4

Cola:

6	8	10	**4**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

~~6~~	8	10	4

No está vacía

v = cola.dequeue() -> 6

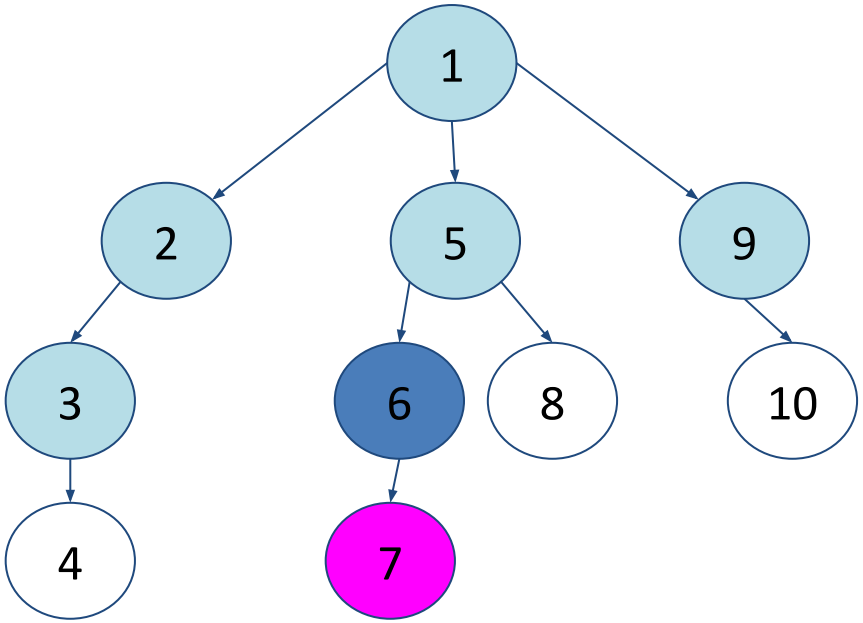
Cola:

8	10	4	

recorrido:

1	2	5	9	3	6		

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

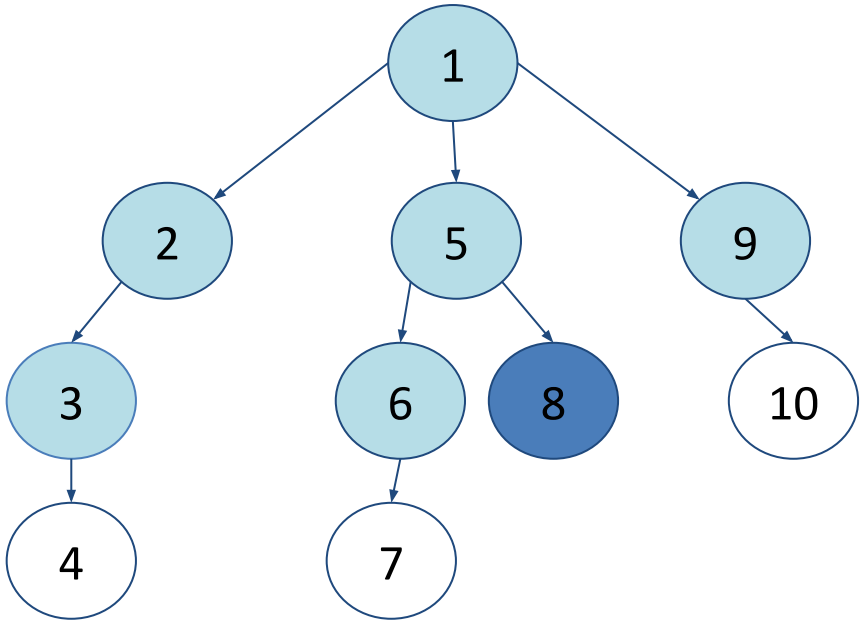
8	10	4	

Debemos añadir a la cola los adyacentes de 6: 7

Cola:

8	10	4	**7**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

8	10	4	7

No está vacía

v = cola.dequeue() -> 8

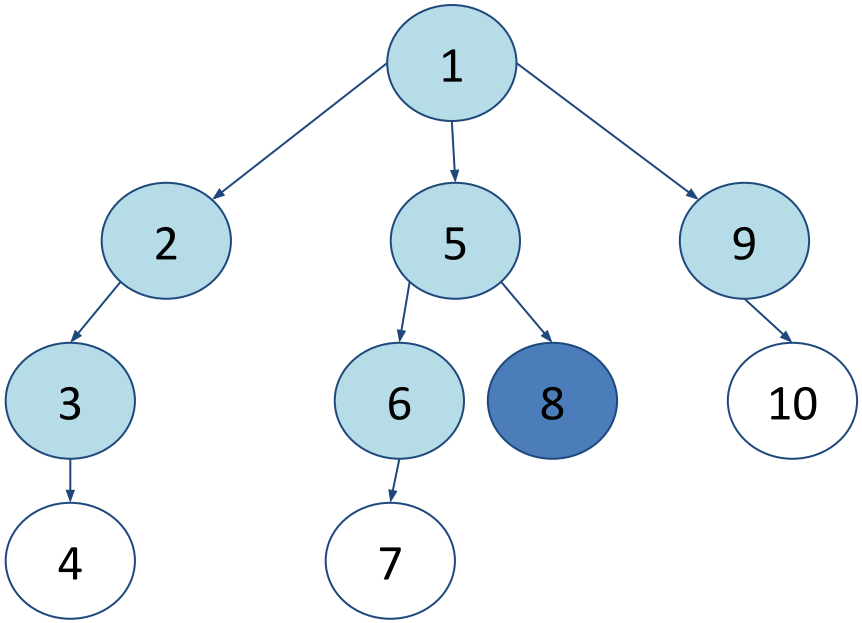
Cola:

10	4	7	

recorrido:

1	2	5	9	3	6	**8**	

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

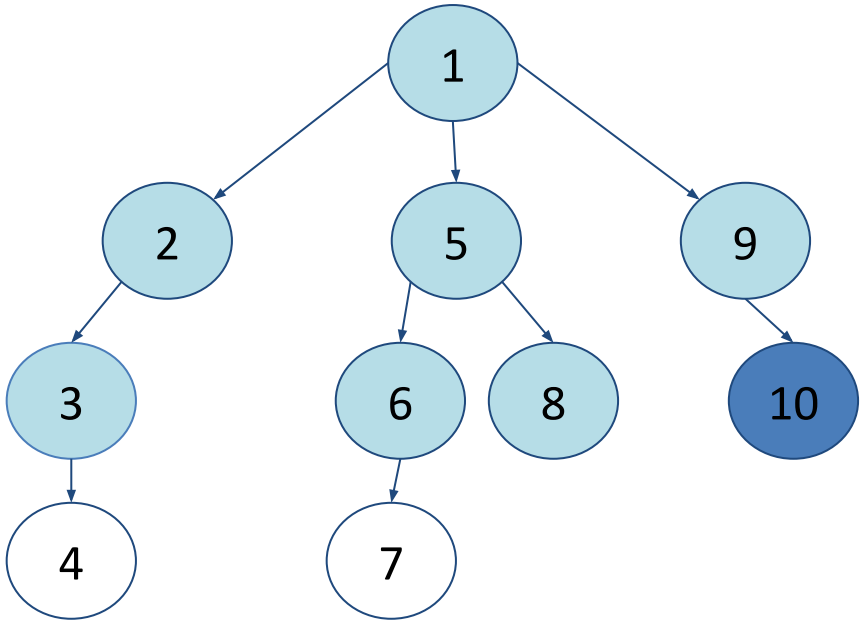
1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

10	4	7	

No añadimos nada a la cola, porque 8 no tiene adyacentes

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

10	4	7	

No está vacía

v = cola.dequeue() -> 10

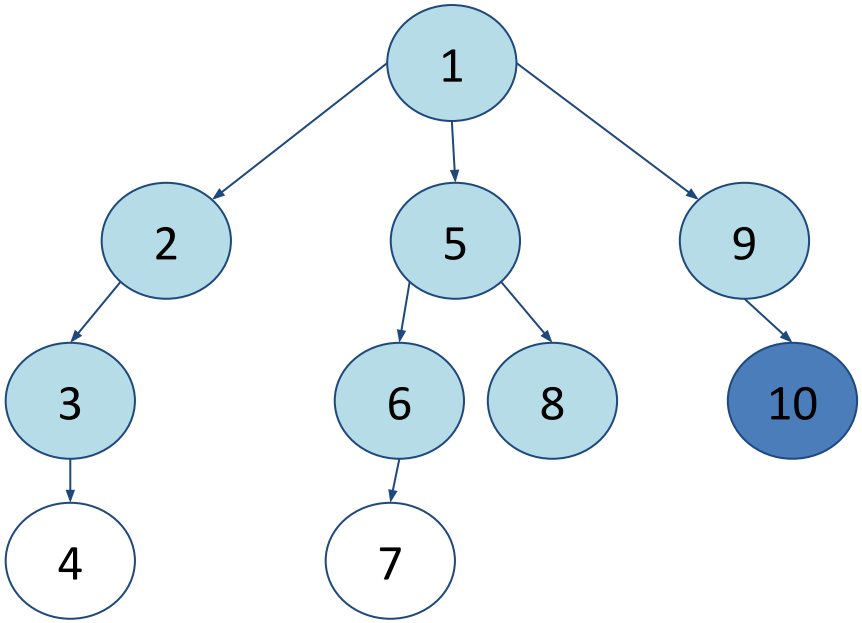
Cola:

4	7		

recorrido:

1	2	5	9	3	6	8	10	

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

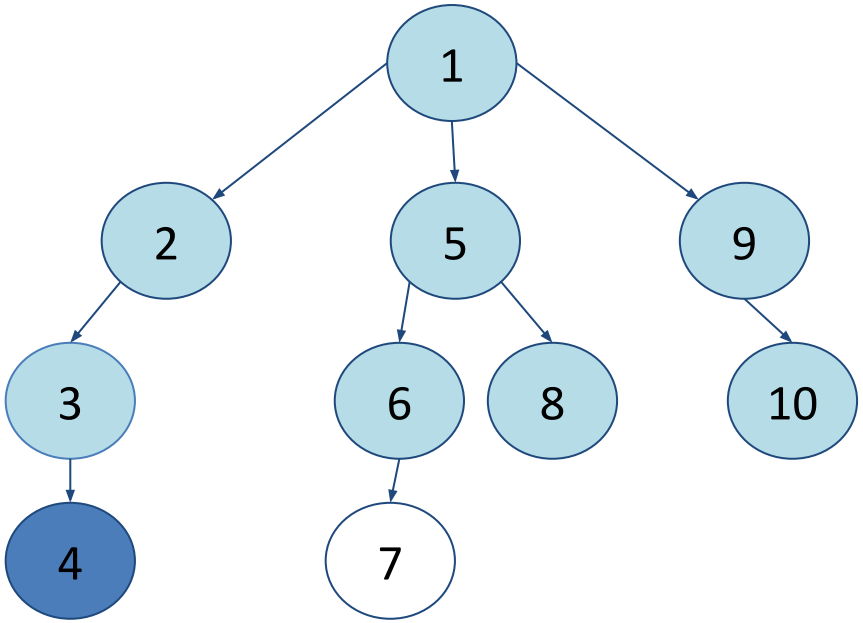
1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

4	7		

No añadimos nada a la cola, porque 10 no tiene adyacentes

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

4	7		

No está vacía

v = cola.dequeue() -> 4

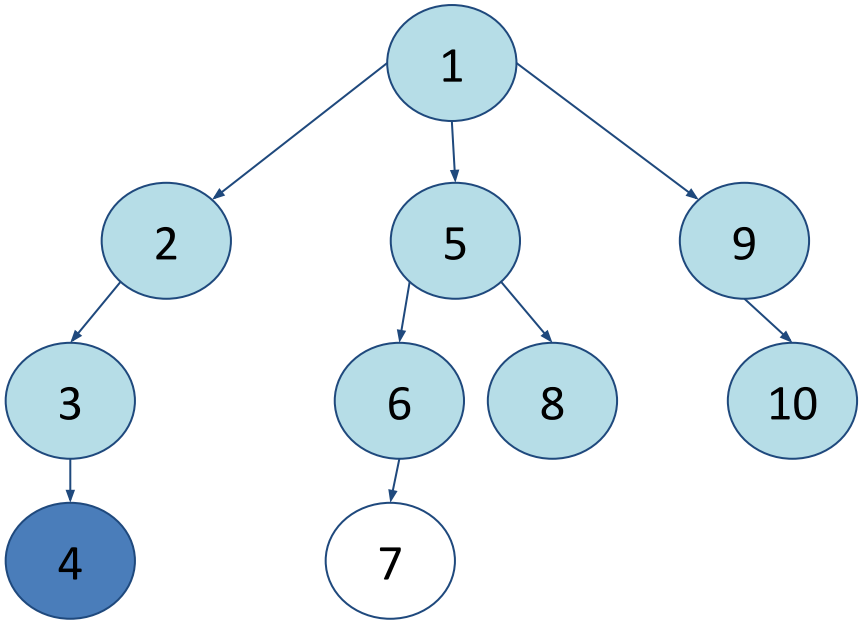
Cola:

7			

recorrido:

1	2	5	9	3	6	8	10	**4**

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

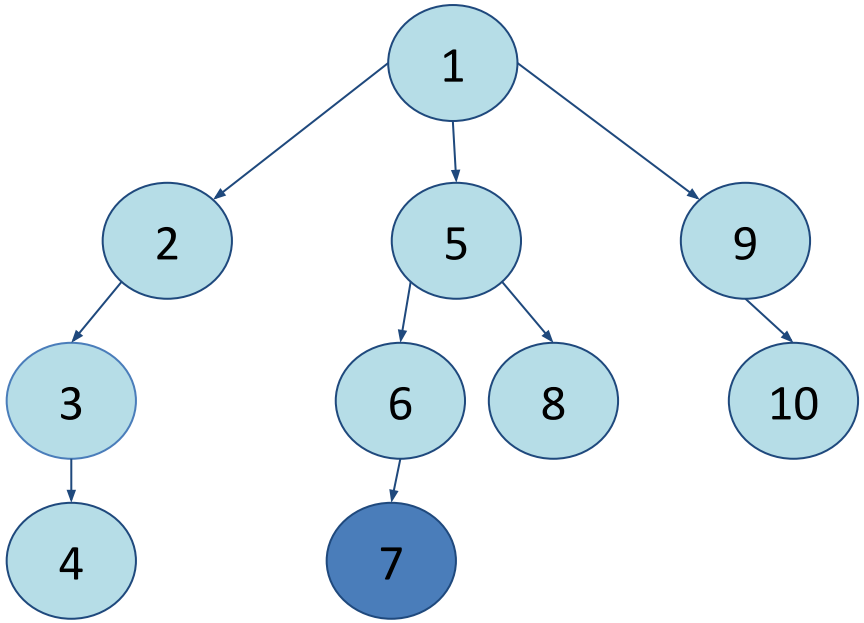
1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

7			

No añadimos nada a la cola, porque 4 no tiene adyacentes

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

7			

No está vacía

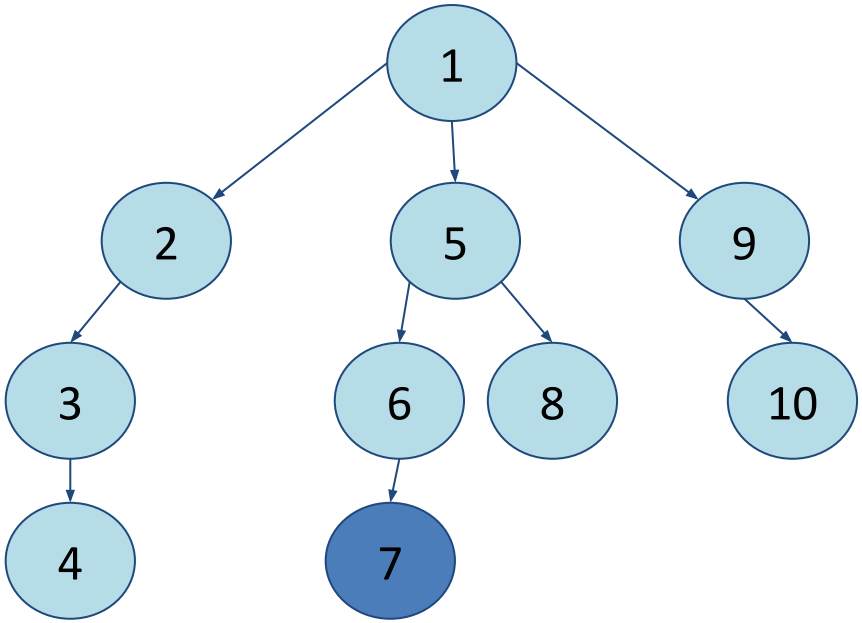
v = cola.dequeue() -> 7

Cola:

recorrido:

1	2	5	9	3	6	8	10	4	**7**	

Breadth first search - Implementación



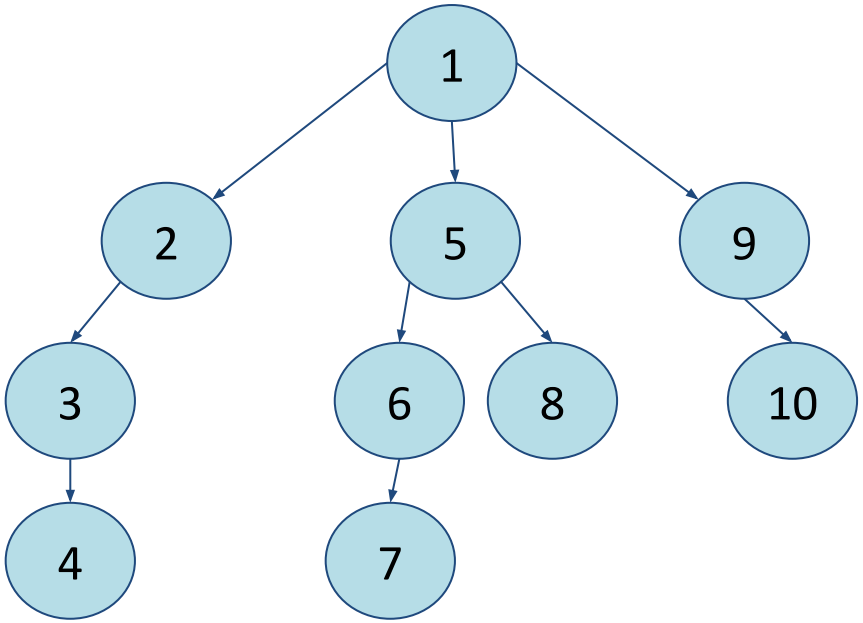
Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Cola:

No añadimos nada a la cola, porque 7 no tiene adyacentes

Breadth first search - Implementación



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Cola:

Terminamos porque la cola ya está vacía

recorrido:

1	2	9	3	6	8	10	4	7	

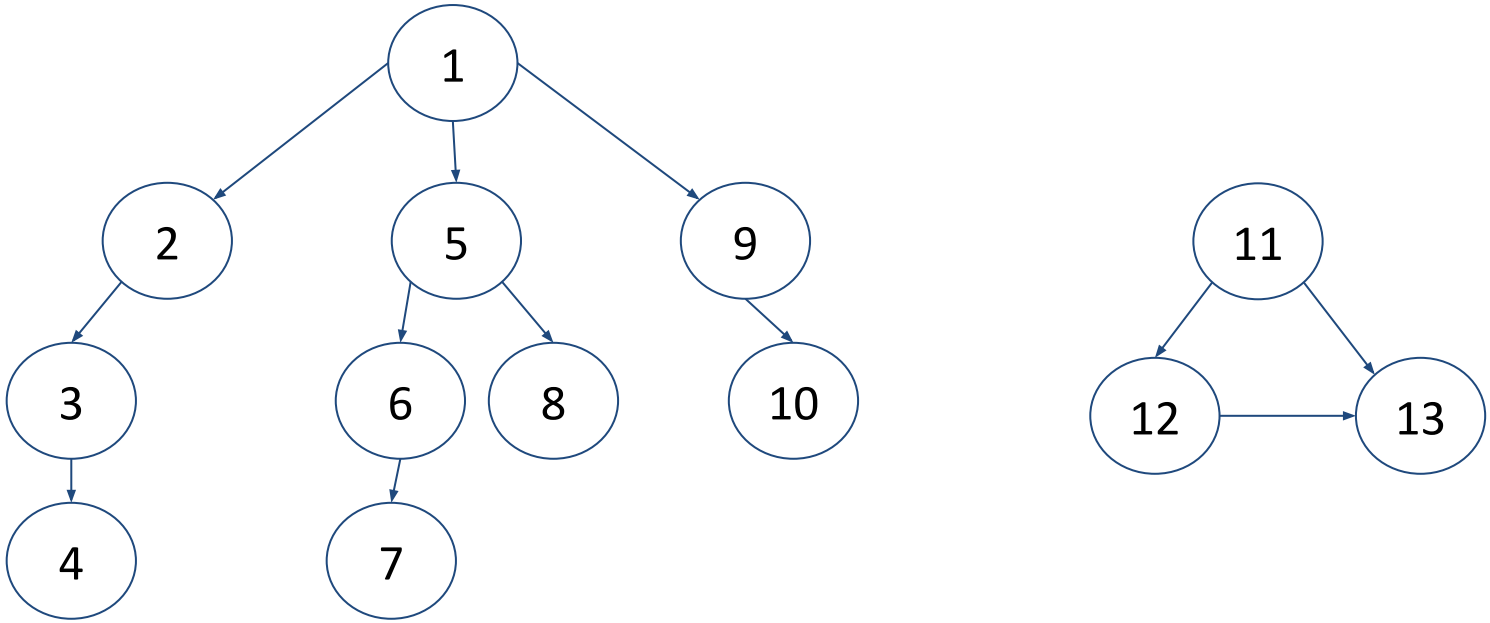
Breadth first search - Implementación

Completa la implementación en la clase Graph

```
def bfs(self, start: object) -> list:
    result = []

    ...

    return result
```

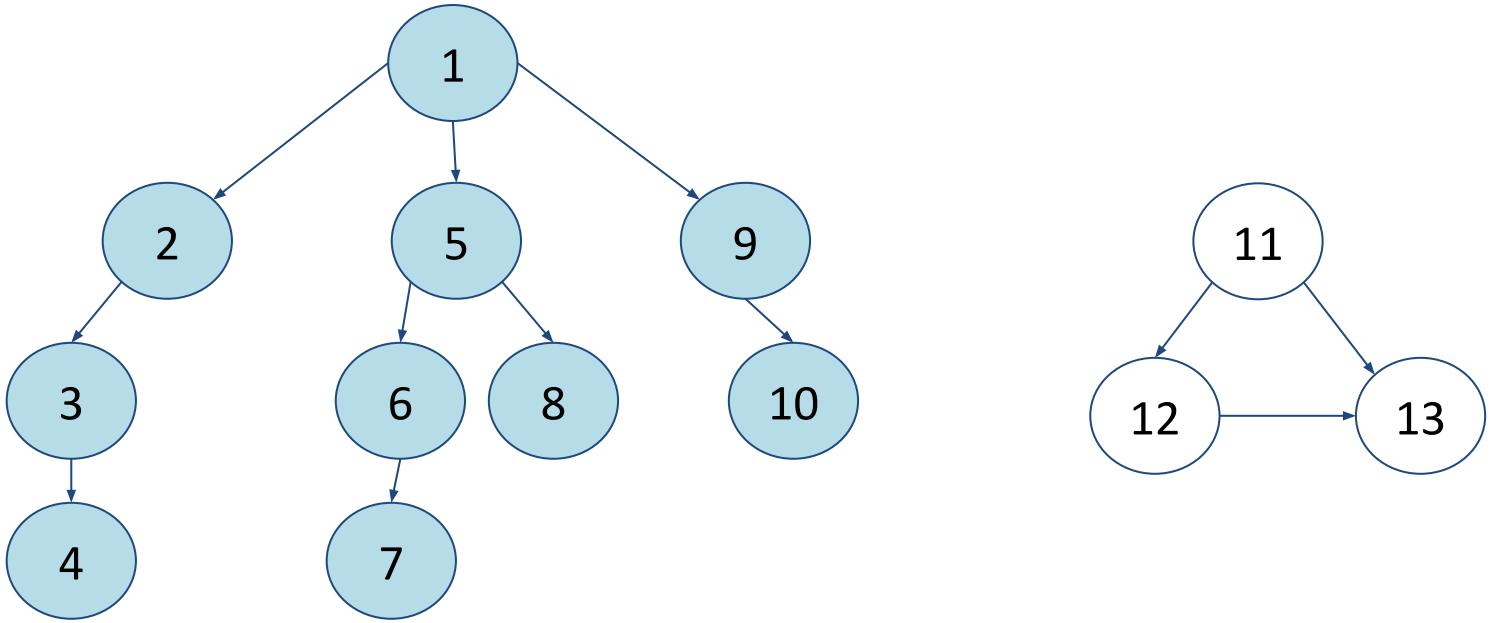
Solución.

Breadth first search - Grafo no conexo



¿Qué pasaría si el grafo es no conexo (es decir, no existe un camino que conecte todos los vértices)?
¿Cómo es el recorrido desde 1?

Breadth first search - Grafo no conexo

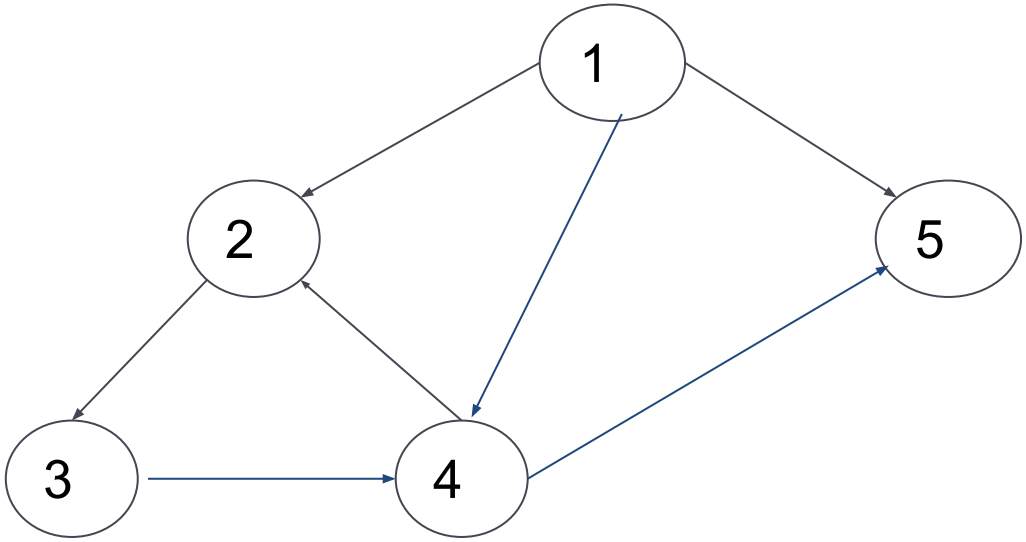


¿Cómo es el recorrido desde 1? Los vértices 11, 12 y 13 no formarían parte del recorrido.

recorrido:

1	2	5	9	3	6	8	10	4	7	

Breadth first search - grafos con ciclos

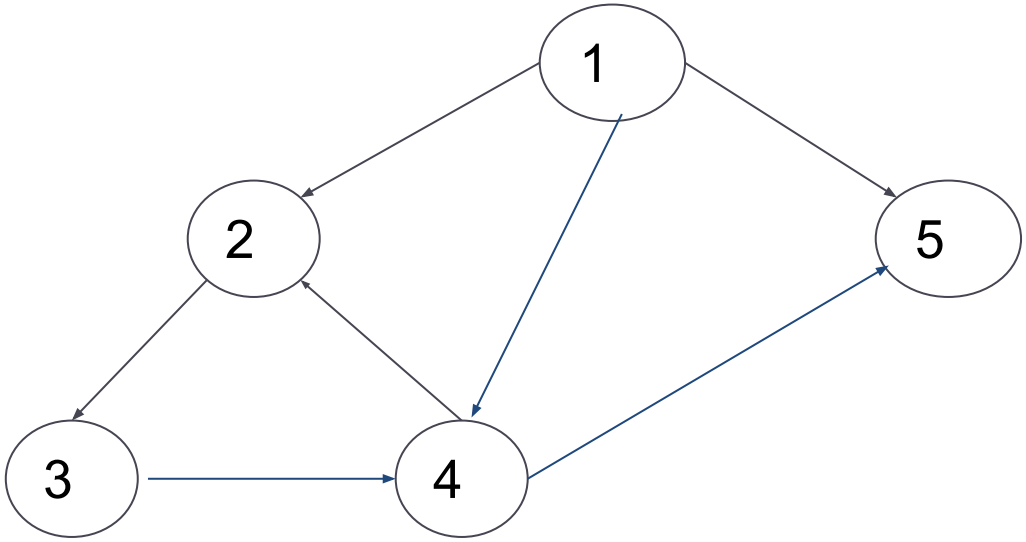


Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) Recuperamos sus vértices adyacentes y los añadimos la cola

¿Funciona el algoritmo si el grafo tiene ciclos?

Breadth first search - grafos con ciclos

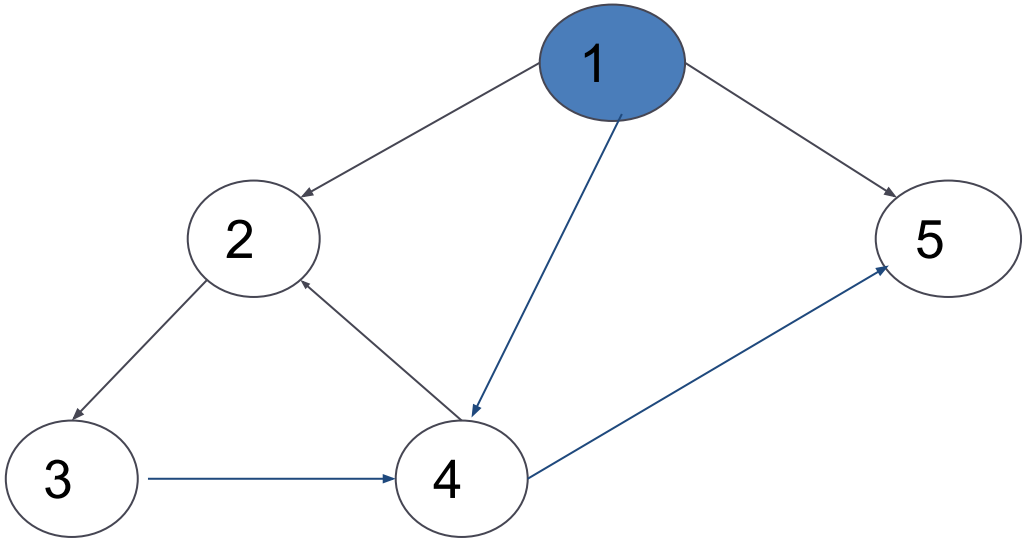


Comenzamos el recorrido por 1, lo añadimos a la cola

cola: | 1 | | | |

recorrido: | |

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

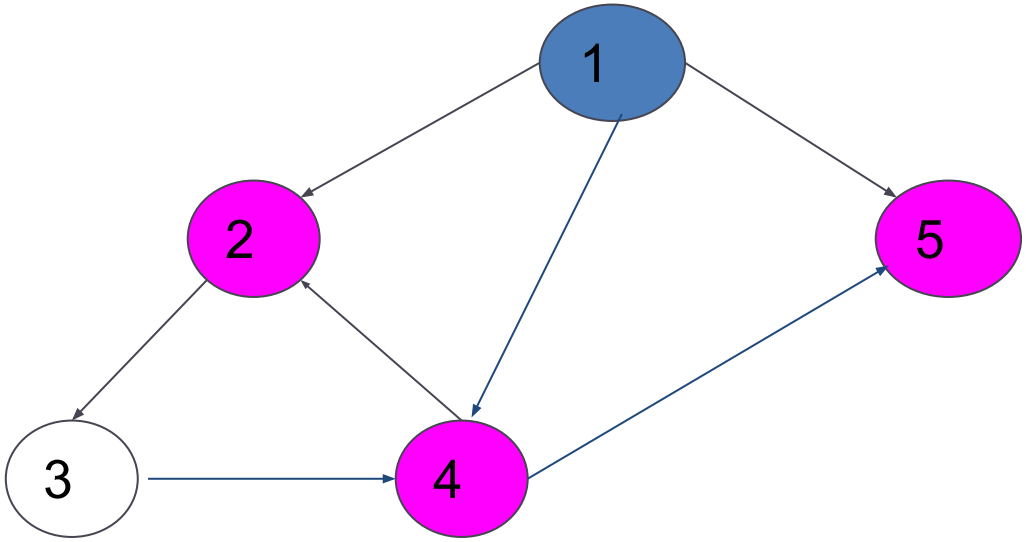
1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Como la cola no esta vacía, entramos en el bucle. Sacamos el 1 de la cola, y lo añadimos al recorrido

cola: | ~~1~~ | | | |

recorrido: | 1 |

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

Obtenemos los adyacentes de 1 (2, 4 y 5) y los añadimos a la cola

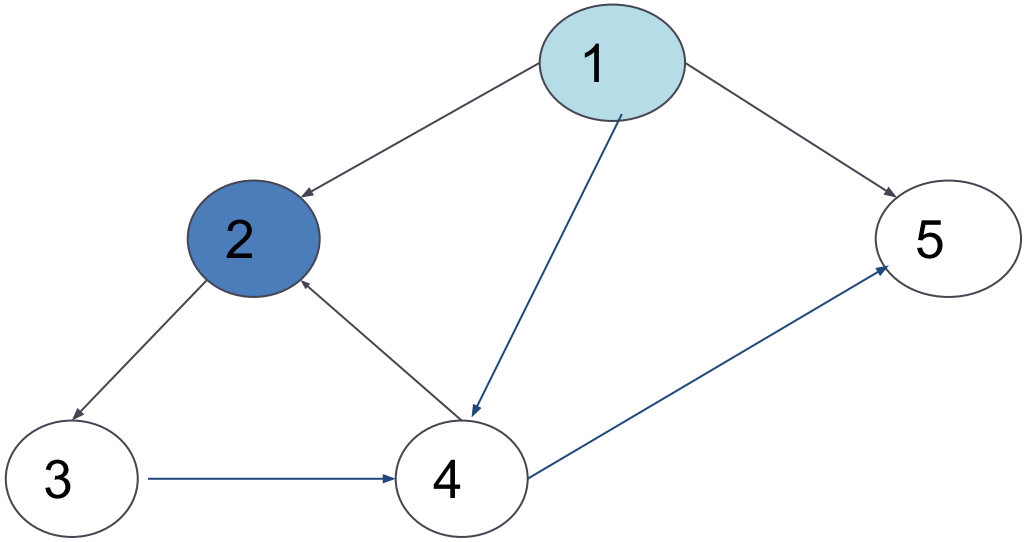
cola:

2	4	5	

recorrido:

1

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

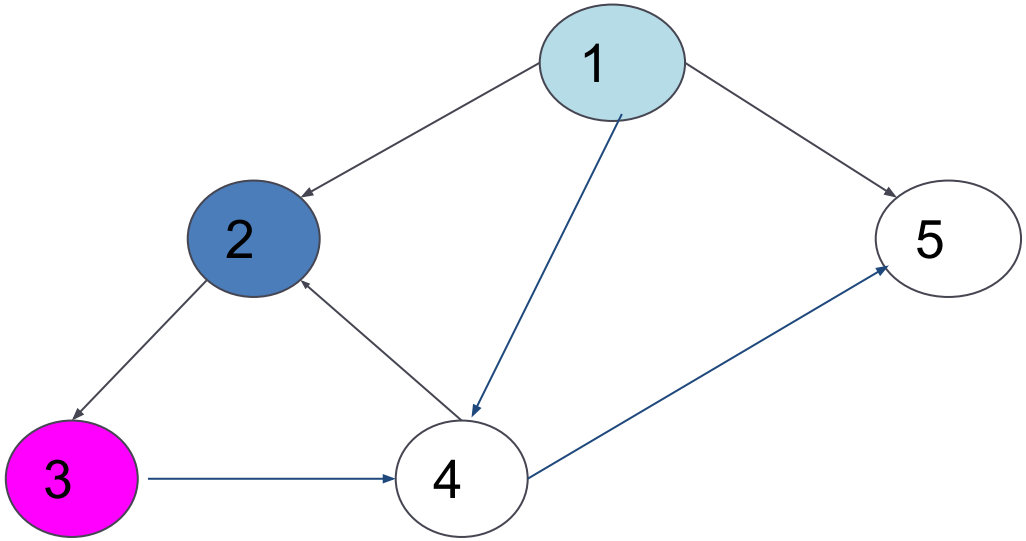
1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

La cola no está vacía. Entramos en el bucle. Sacamos el primero de la cola: 2, y lo añadimos al recorrido

cola:	~~2~~	4	5	

recorrido:	1	**2**

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los añadimos la cola**

2 sólo tiene un adyacente: 3. Lo añadimos a la cola.

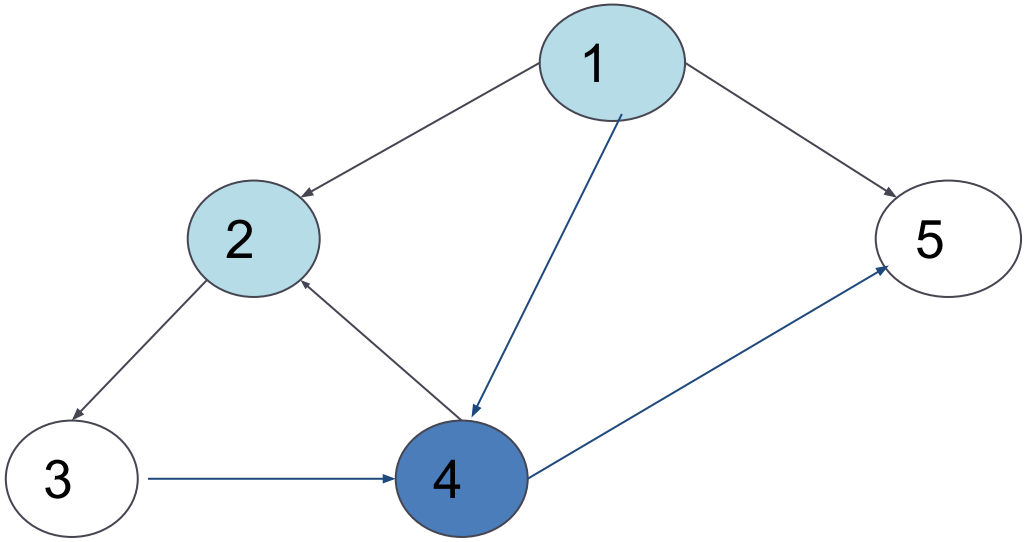
cola:

4	5	**3**

recorrido:

1	2

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

La cola no está vacía. Entramos en el bucle. Sacamos el primer elemento: 4, y lo añadimos al recorrido.

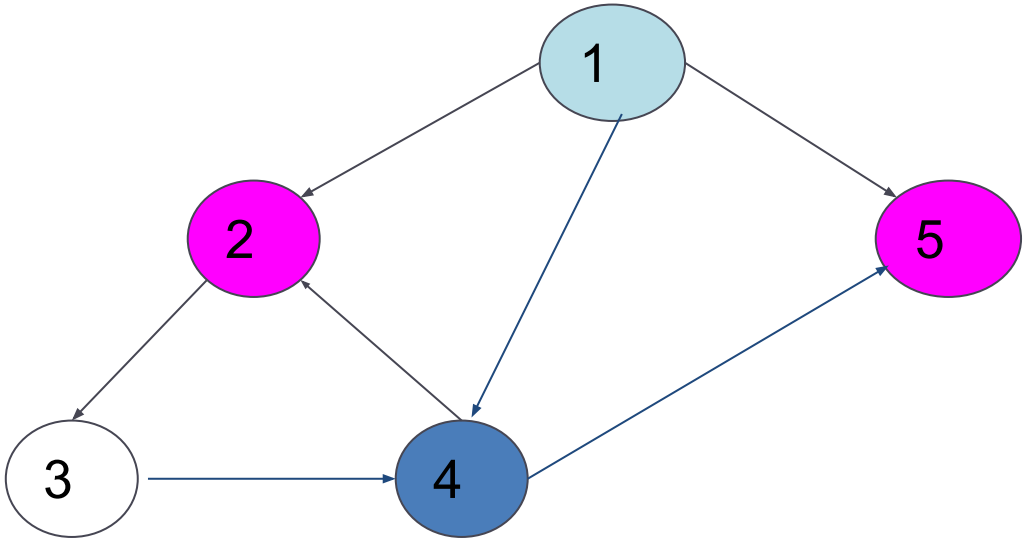
cola:

~~4~~	5	3

recorrido:

1	2	**4**

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los añadimos la cola

Recuperamos los adyacentes de 4: 2, y 5.

- 2 ya está en el recorrido. Si volvemos añadir el 2 a la cola, entraríamos en un bucle infinito.
- El 5 aún no ha sido añadido al recorrido, pero está en la cola. Si lo añadimos a la cola, entraríamos en un bucle infinito.

cola:

~~4~~	5	3

recorrido:

1	2	**4**

Breadth first search - grafos con ciclos

- Para evitar caer en un ciclo, sólo vamos a añadir a la cola aquellos vértices adyacentes que no están en la cola y que tampoco han sido añadidos al recorrido.
- Es posible implementarlo mirando en la cola y en la lista recorrido, pero esto aumentaría la complejidad temporal del algoritmo.
- Una posible alternativa es utilizar un diccionario de booleanos, **visited**, donde las claves son los vértices del grafo. Inicialmente todos los valores son False.
- Cuando un vértice se añade a la cola, también se modificar su valor a True.

1	2	3	4	5
False	False	False	False	False

Breadth first search - grafos con ciclos

- Antes de añadir un vértice v a la cola, se consulta su valor asociado en el diccionario **visited:**
  - Si visited[v]=False, significa que aún no ha sido visitado, por tanto, debemos añadirlo a la cola. Además, también debemos marcarlo como visitado (es decir, debemos modificar su valor a True).
  - Si visited[v]=True, quiere decir que ya ha sido visitado. En este caso, no debemos hacer nada con el vértice.

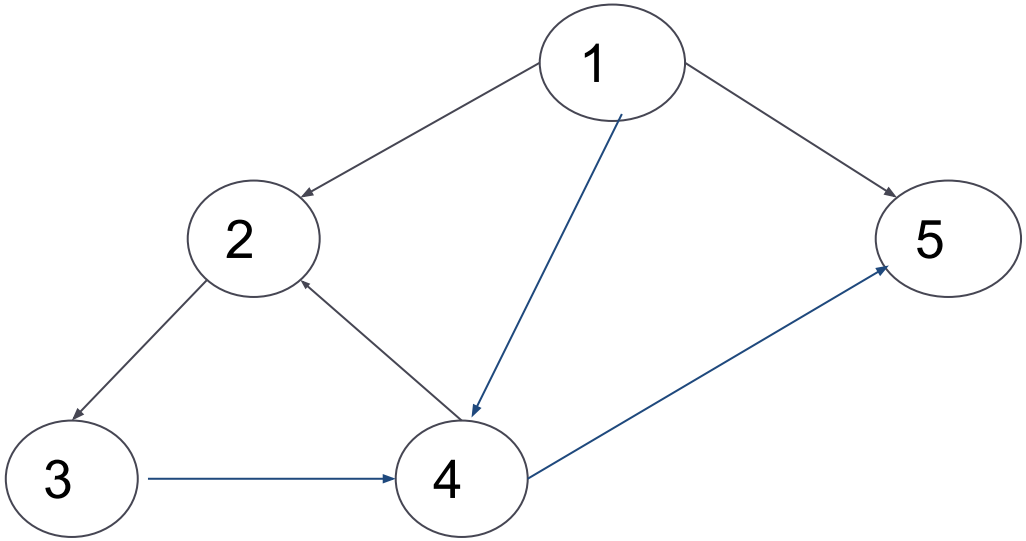
Breadth first search - grafos con ciclos

- El algoritmo modificado es:

Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) Recuperamos sus vértices adyacentes y los recorremos. **Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.**

Breadth first search - grafos con ciclos



El recorrido comienza en el vértice 1. Lo añadimos a la cola y lo marcamos como visitado.

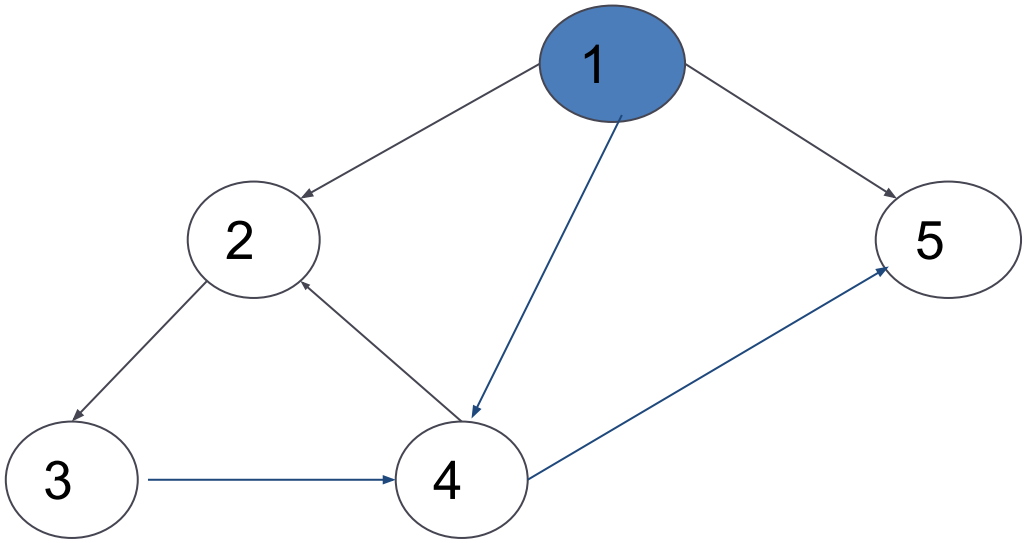
cola:

1			

visited:

1	2	3	4	5
True	False	False	False	False

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.

La cola no está vacía. Entramos en el bucle. Sacamos el primer elemento de la cola: 1, y lo añadimos al recorrido

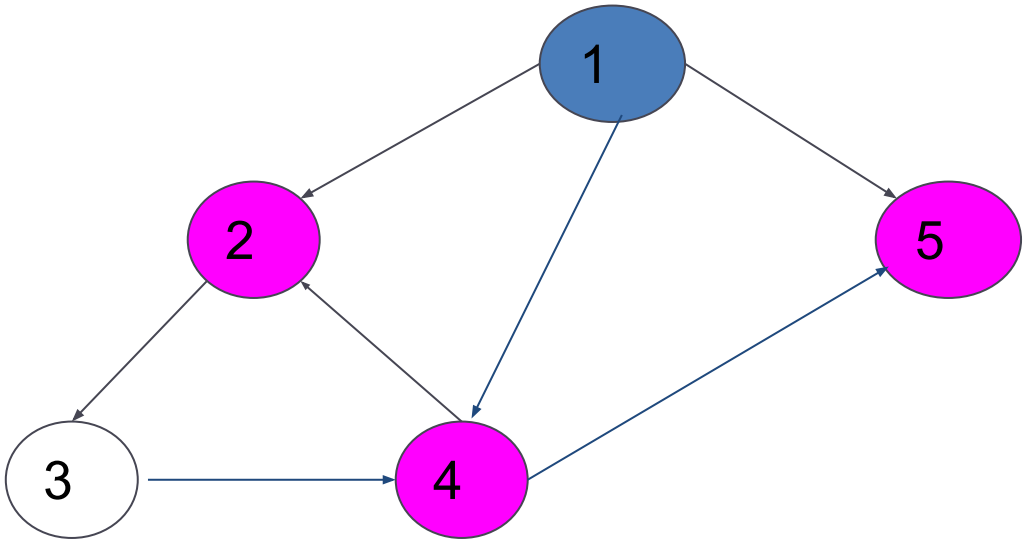
cola:

~~1~~			

recorrido:

1			

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.**

Los adyacentes de 1 son 2, 4 y 5. Como ninguno está marcado como visitado, los añadimos a la cola y los marcamos como visitados.

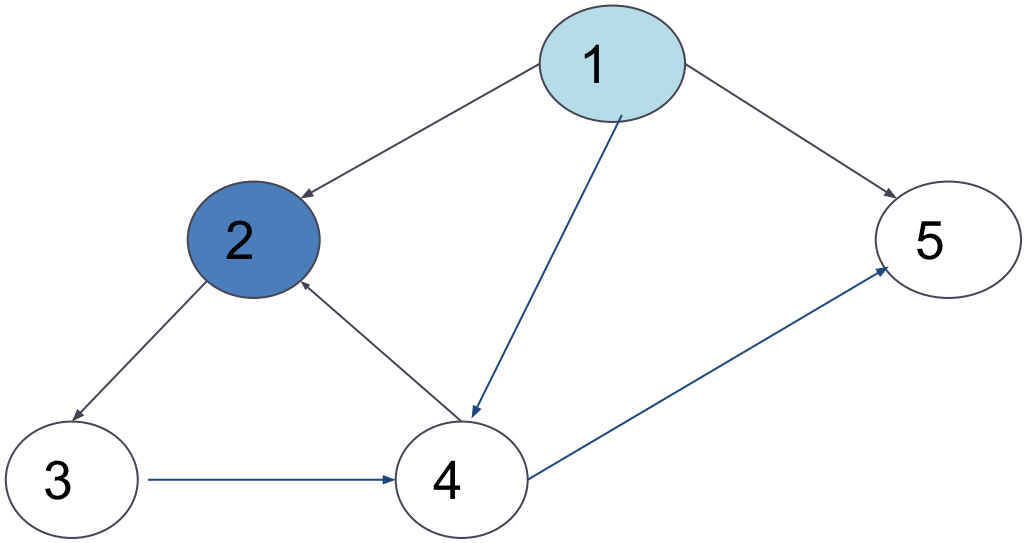
cola:

2	**4**	**5**	

visited:

1	**2**	3	**4**	**5**
True	**True**	False	**True**	**True**

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

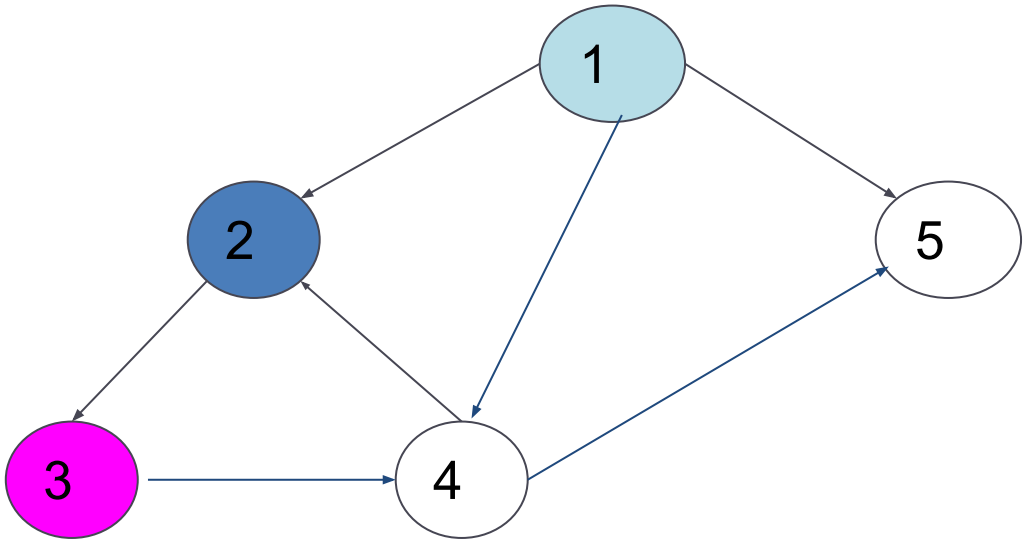
1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.

La cola no está vacía. Entramos en el bucle. Sacamos el primer elemento de la cola: 2, y lo añadimos al recorrido

cola:	~~2~~	4	5	

recorrido:	1	**2**		

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.**

El único adyacente de 2 es 3. Como no ha sido visitado, lo añadimos a la cola y lo marcamos como visitado.

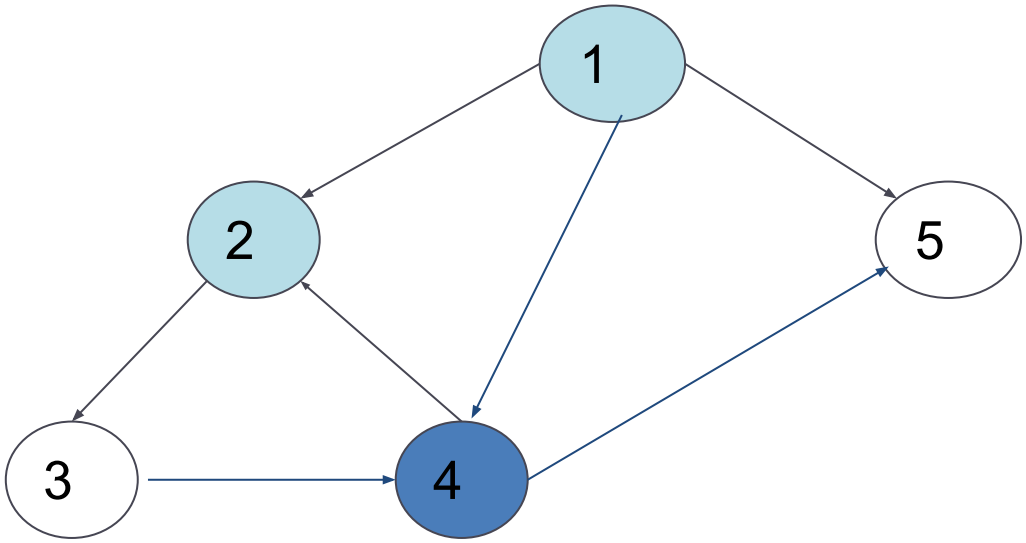
cola:

4	5	**3**	

visited:

1	2	**3**	4	5
True	True	**True**	True	True

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.

La cola no está vacía. Entramos en el bucle. Sacamos el primer elemento de la cola: 4, y lo añadimos al recorrido

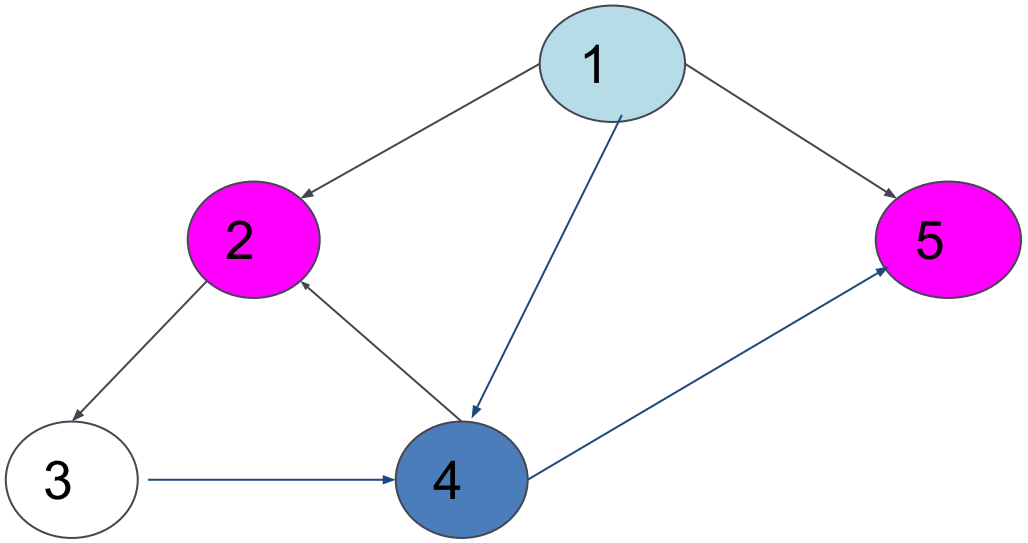
cola:

~~4~~	5	3	

recorrido:

1	2	**4**	

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.**

4 tiene dos vértices adyacentes: 2 y 5. Ambos ya han sido visitados. No añadimos nada a la cola.

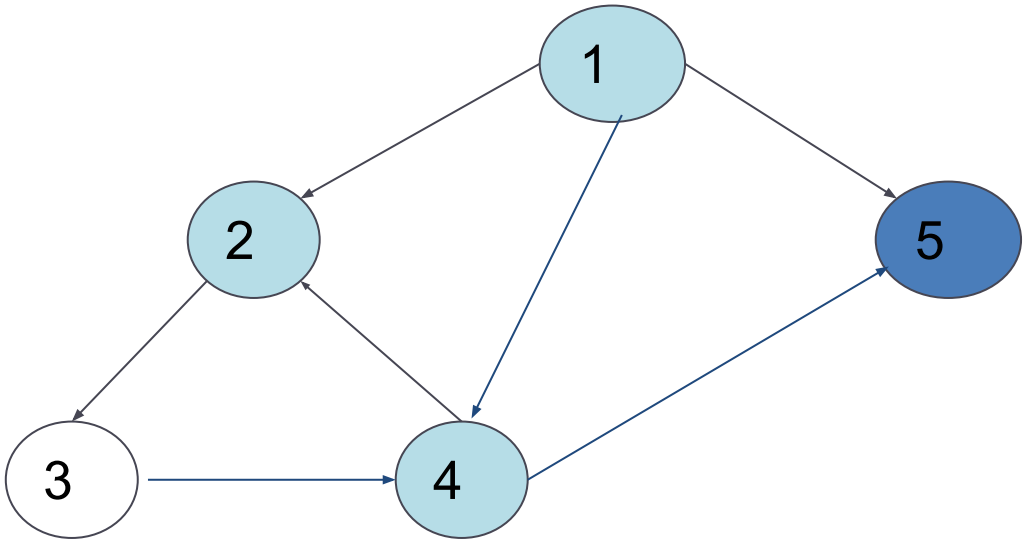
cola:

5	3		

visited:

1	2	3	4	5
True	True	True	True	True

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.

La cola no está vacía. Entramos en el bucle. Sacamos el primer elemento de la cola: 5, y lo añadimos al recorrido

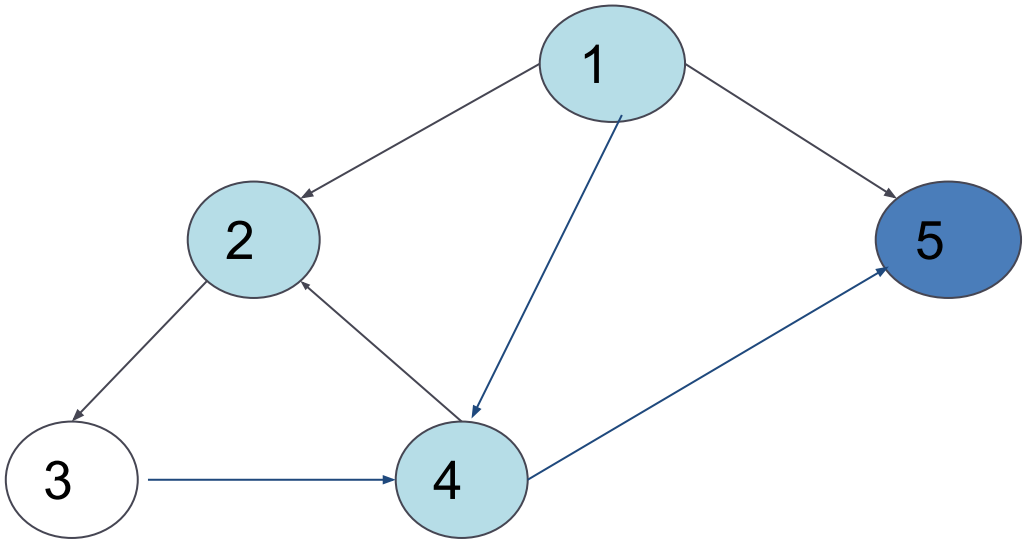
cola:

~~5~~	3		

recorrido:

1	2	4	**5**

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.**

5 no tiene adyacentes. NO añadimos nada a la cola.

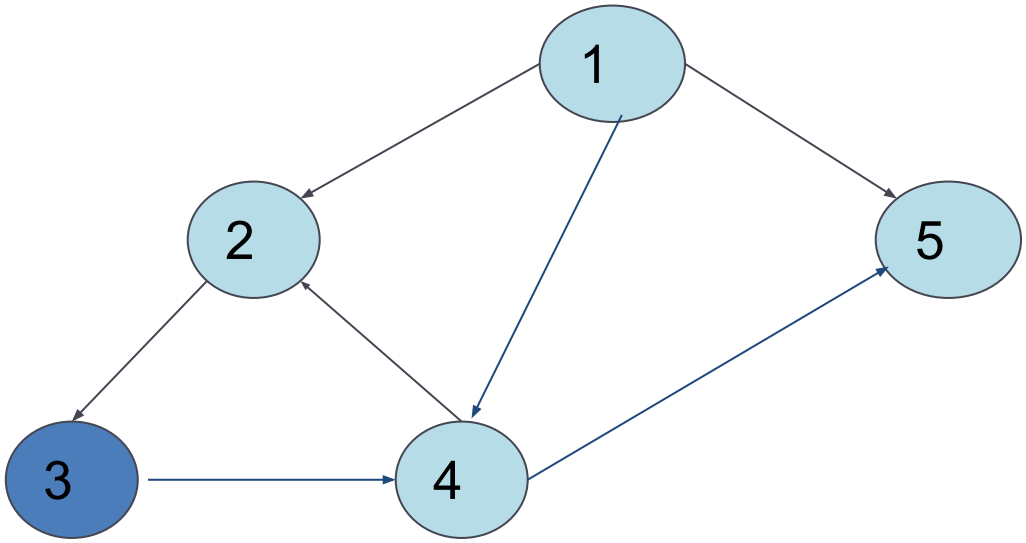
cola:

3	4		

visited:

1	2	3	4	5
True	True	True	True	True

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.

La cola no está vacía. Entramos en el bucle. Sacamos el primer elemento de la cola: 3, y lo añadimos al recorrido

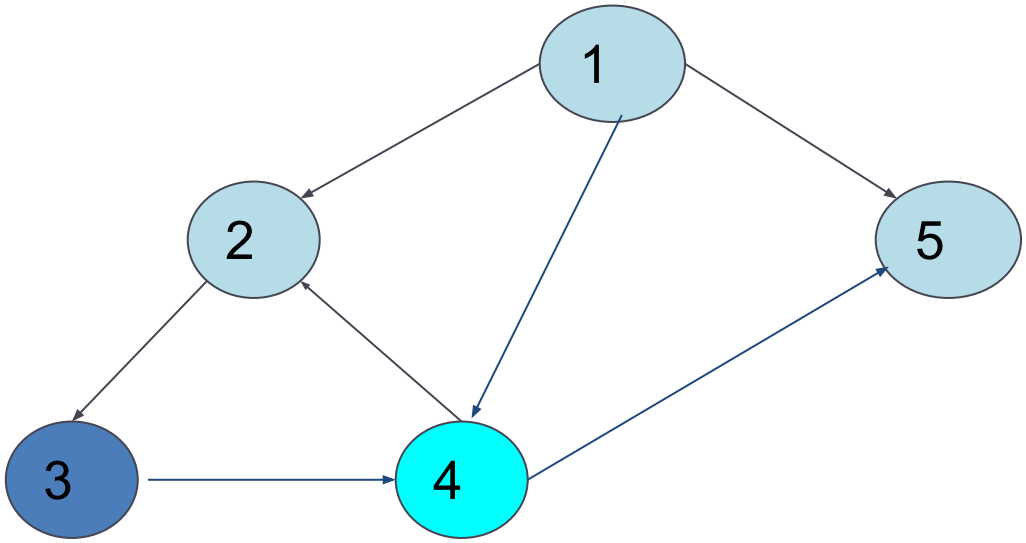
cola:

~~3~~			

recorrido:

1	2	4	5	**3**

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) Sacamos el primer elemento de la cola y lo añadimos al recorrido
2) **Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.**

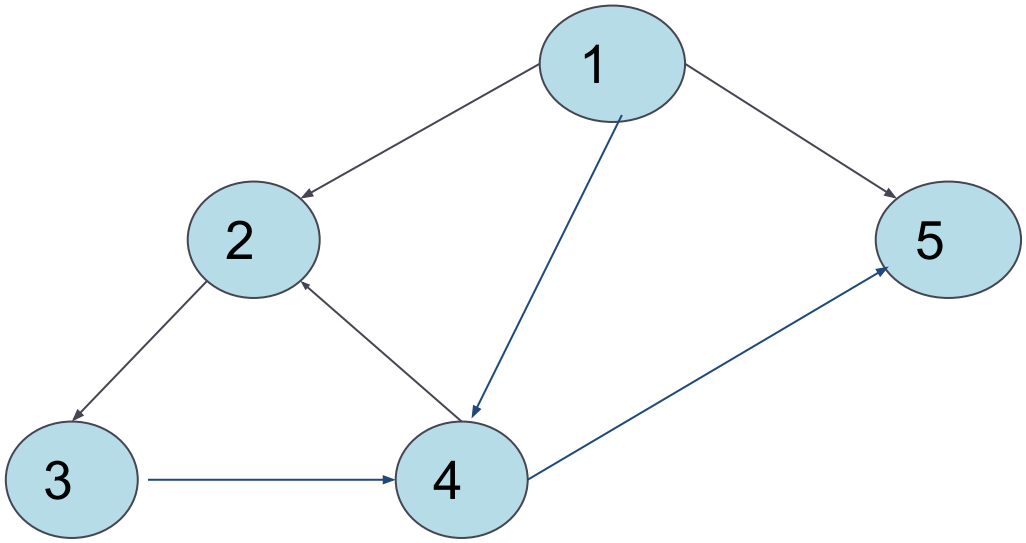
El único adyacente de 3 es 4. Como 4 ya ha sido visitado, no tenemos que añadirlo a la cola.

cola:

visited:

1	2	3	4	5
True	True	True	True	True

Breadth first search - grafos con ciclos



Mientras que la cola no esté vacía, realizamos los siguientes pasos:

1) **Sacamos el primer elemento de la cola y lo añadimos al recorrido**
2) Recuperamos sus vértices adyacentes y los recorremos. Si un vértice adyacente no ha sido visitado, lo marcamos como visitado y lo añadimos a la cola.

La cola está vacía. Hemos terminado el recorrido!!!

cola:

recorrido:

1	2	4	5	3

Breadth first search - grafos con ciclos

Ejercicio:
En la clase Graph, implementa el algoritmo **bfs(vertex)**, que devuelva una lista Python con el recorrido en amplitud desde el vértice vertex.
Prueba la implementación para varios grafos.
Solución.

Complejidad Espacial - Recorrido en anchura

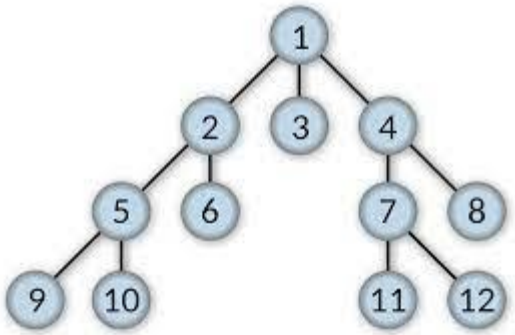
- El método bfs utiliza las siguientes estructuras auxiliares:
  - Lista recorrido: n, en el peor de los casos, tendrá todos los vértices del grafo.
  - Cola q: n, en el peor de los casos, contiene n-1 vértices (el primer vértice tiene como vecinos al resto de vértices).
  - Diccionario visited: 2n, porque es un diccionario de n keys (los vértices) y n booleanos asociados
- Por tanto, la complejidad espacial del método bfs será de orden n (sin contar el espacio del grafo).

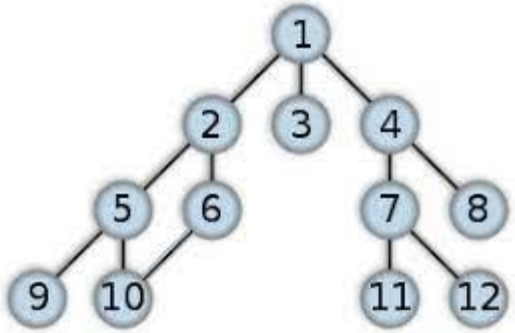
Complejidad Temporal - Recorrido en anchura

- El bucle principal, en el peor de los casos (todos los vértices están conectados), se va a ejecutar 1 vez por cada vértice del grafo.
- Dentro del bucle principal, por cada vértice que se está añadiendo al recorrido, es necesario recorrer todos sus vértices adyacentes, para comprobar cuales no han sido visitados, y entonces añadirlos a la cola.
  - Comprobar si un vértice ha sido visitado, tiene complejidad O(1).
  - Añadirlo a la cola, tiene complejidad O(1)
- El peor de los casos, es cuando no hay ciclos, y tenemos que añadir siempre todos los vértices adyacentes de cada vértice.
- La complejidad temporal del recorrido en anchura será O(|V| + |A|). Recuerda que |A|->n, si e grafo es escaso, y n^2, si el grafo es denso.

Complejidad Temporal - Recorrido en anchura

- Por ejemplo, en el grafo de la izquierda, no hay ciclos, y siempre se tendrán que visitar todos los nodos adyacentes a uno dado. Por tanto, se tendrán que visitar todas las aristas.
- 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.

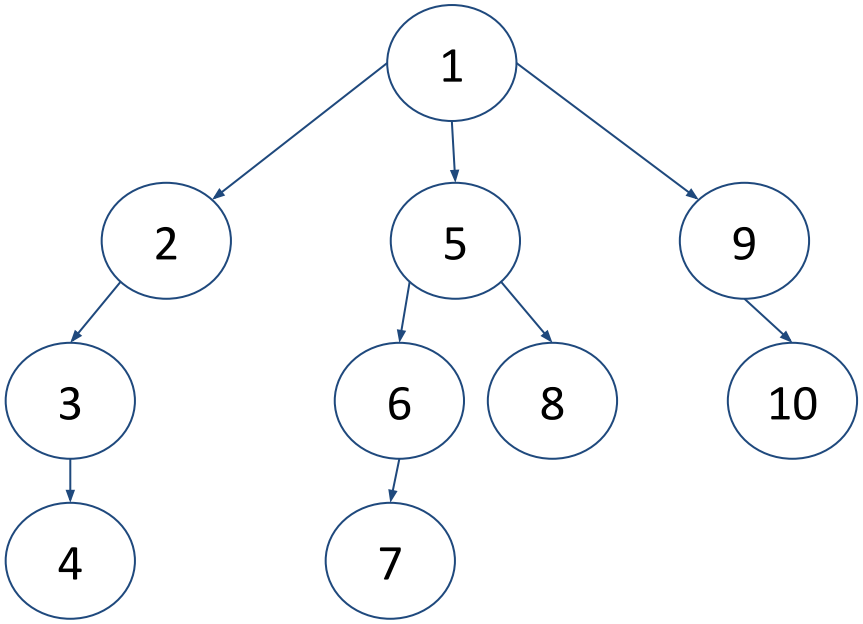




Source. https://www.techiedelight.com/check-undirected-graph-contains-cycle-not/

Depth first search

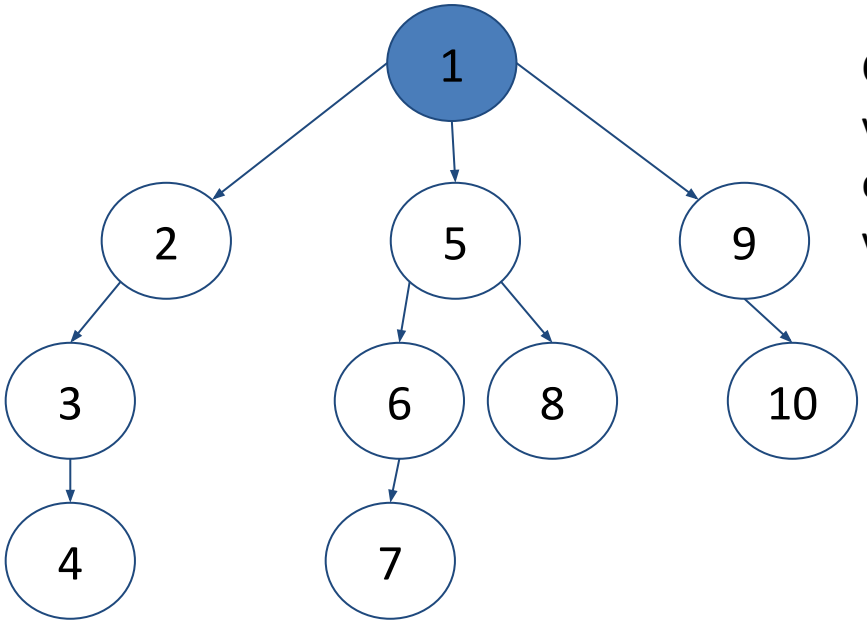
Recorrido en profundidad (Depth first search) o amplitud: similar al recorrido preorder en árboles.



Comienza en un vértice y explora cada rama secuencialmente, profundizando lo máximo posible antes de pasar a la siguiente rama no visitada.

Puedes ver una demo en: https://www.thedshandbook.com/breadth-first-search/

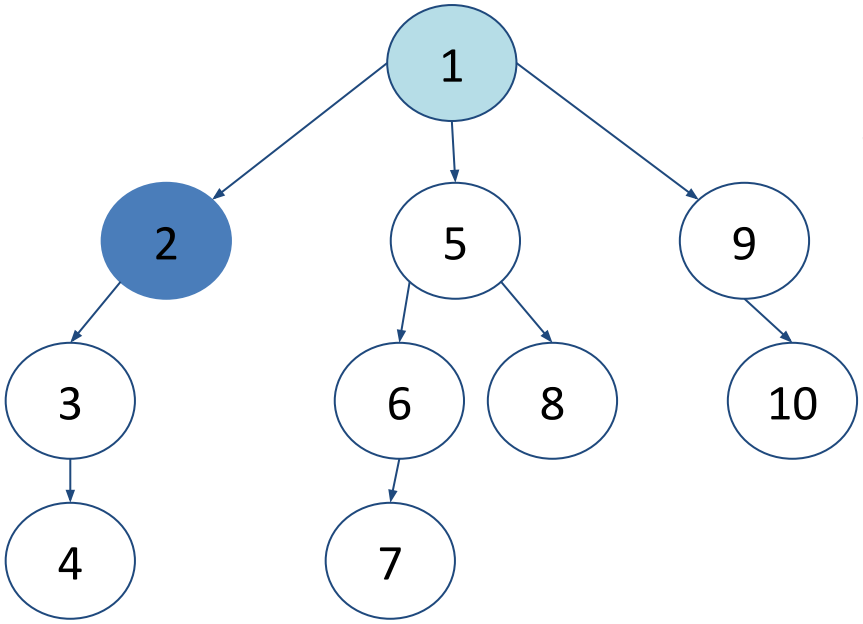
Depth first search



Comenzamos visitando el vértice 1 (podríamos empezar por cualquier otro vértice).

Recorrido: 1

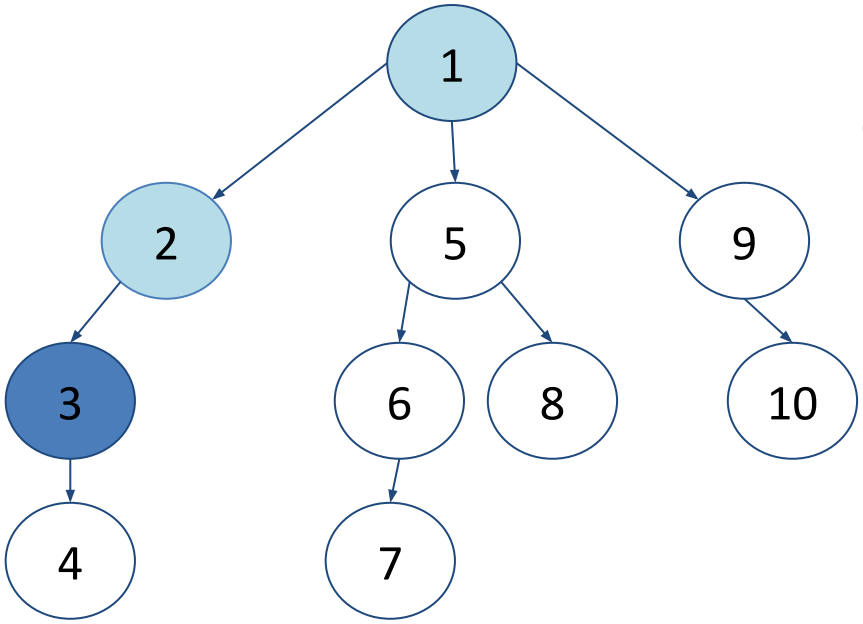
Depth first search



Continuamos por uno de sus vértices adyacentes, por ejemplo, el 2.

Recorrido: 1, **2**

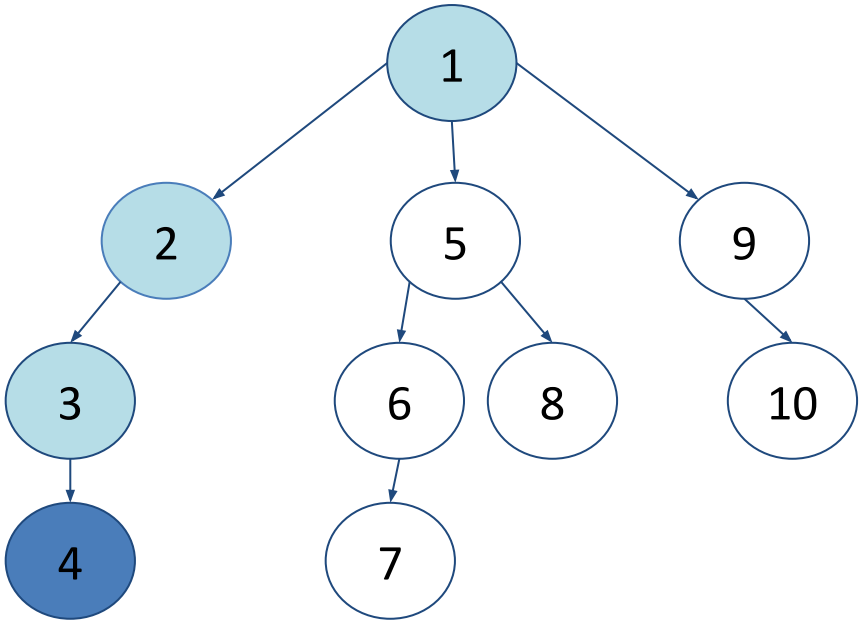
Depth first search



Continuamos explorando esa rama hasta que sea posible.

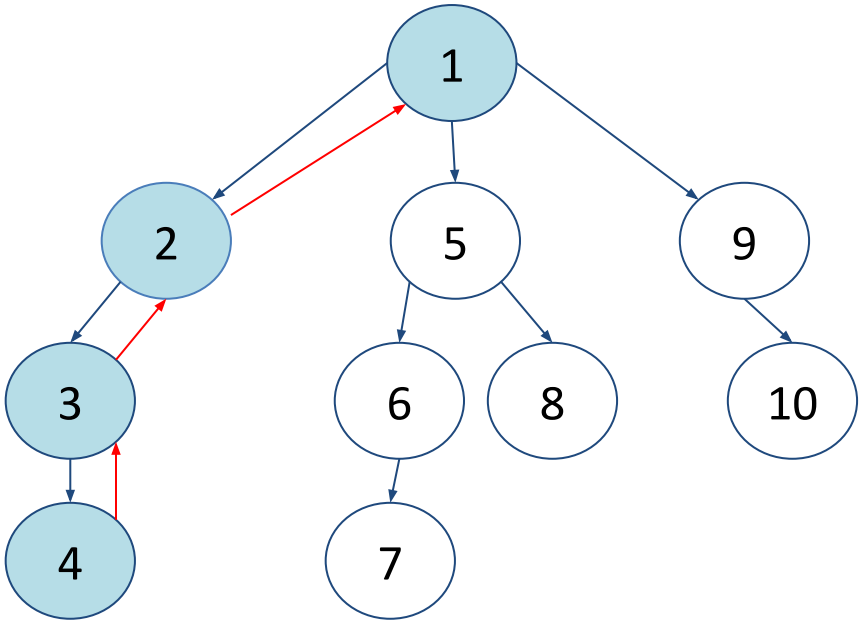
Recorrido: 1, 2, **3**

Depth first search



Recorrido: 1, 2, 3, **4**

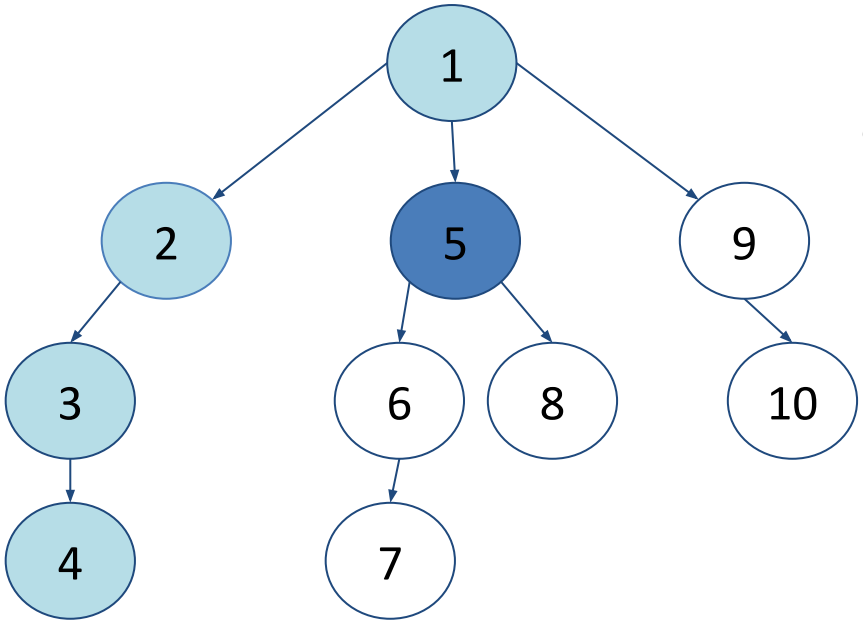
Depth first search



Ya no podemos continuar por esta rama. El algoritmo debe retroceder hasta encontrar un nodo que tenga algún vértice adyacente que aún no haya sido visitado. En este caso, es necesario retroceder hasta el vértice 1.

Recorrido: 1, 2, 3, 4

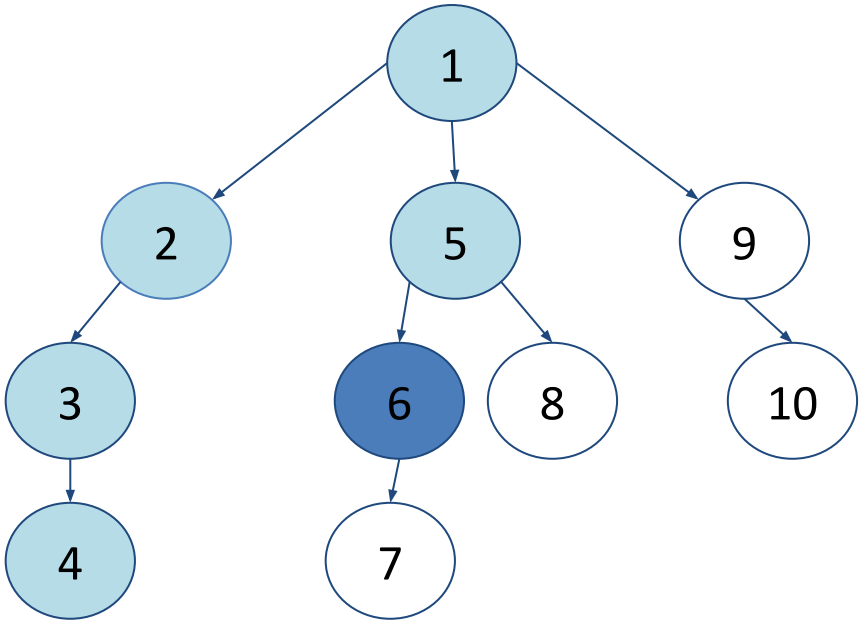
Depth first search



El 1 tiene 2 vértices adyacentes que aún no han sido visitados: 5 y 9. Continuamos por el 5.

Recorrido: 1, 2, 3, 4, **5**

Depth first search

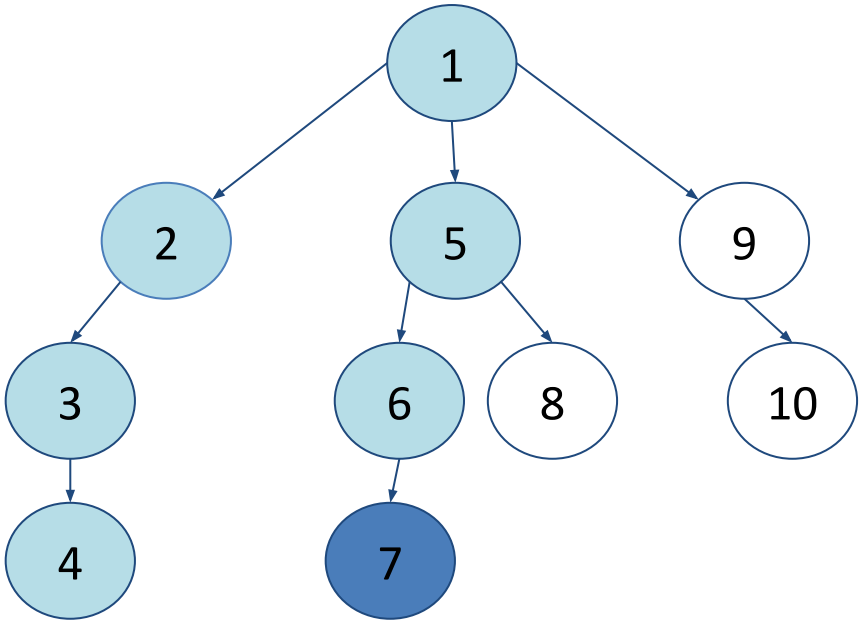


5 tiene tres vecinos: 1, 6 y 8.
1 ya ha sido visitados (así que esa rama la descartamos).

Continuamos por 6

Recorrido: 1, 2, 3, 4, 5, **6**

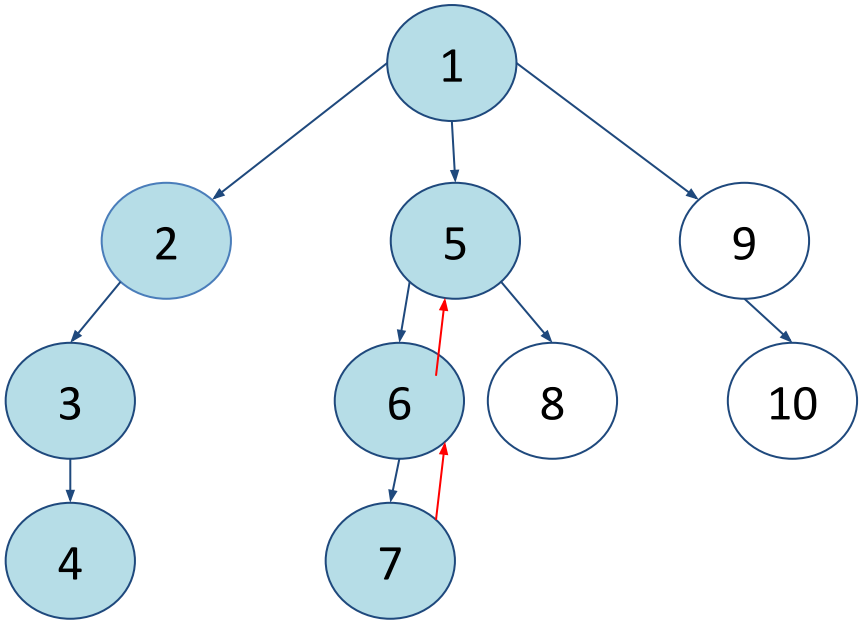
Depth first search



No podemos continuar más por esa rama.

Recorrido: 1, 2, 3, 4, 5, 6, **7**

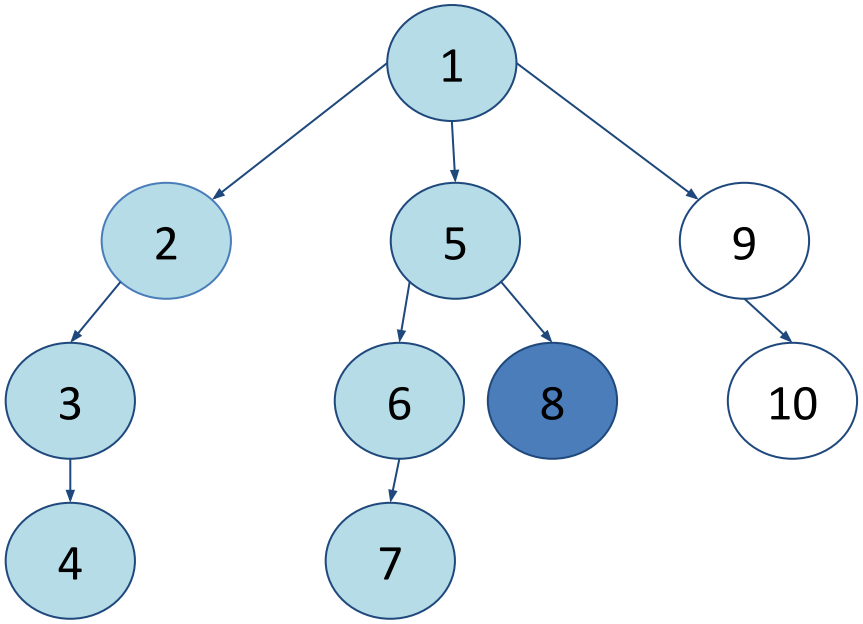
Depth first search



Retrocedemos hasta el primer nodo que tenga algún vecino adyacente que aún no haya sido visitado. Es decir, retrocedemos hasta el 5.

Recorrido: 1, 2, 3, 4, 5, 6, **7**

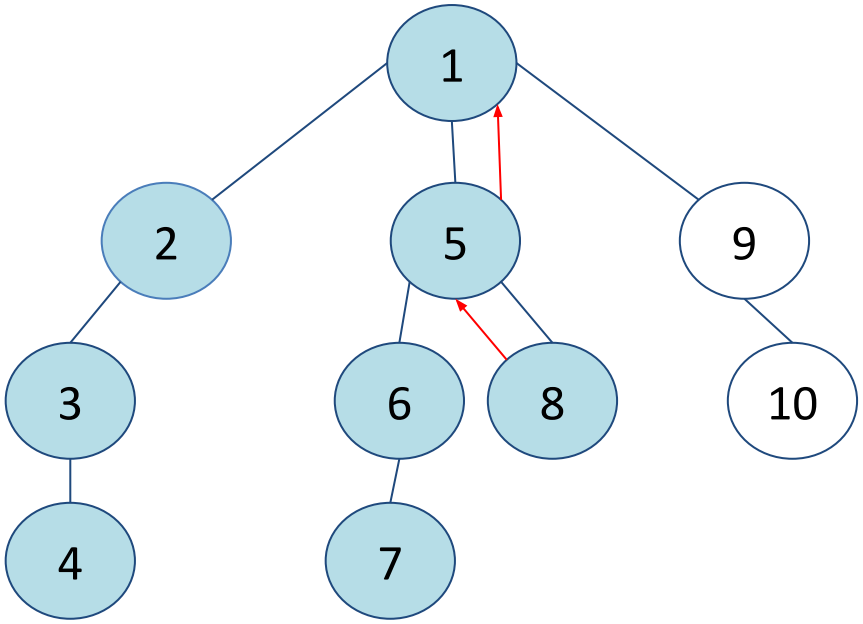
Depth first search



El 5 tiene como vecinos: 1, 6 y 8. El único que queda por visitar es 8. Continuamos por el 8.

Recorrido: 1, 2, 3, 4, 5, 6, 7, **8**

Depth first search

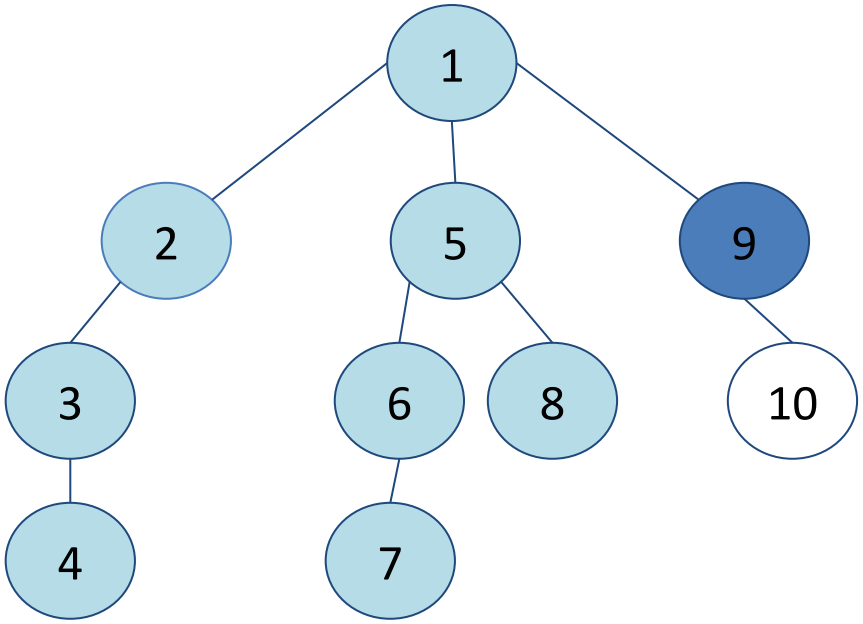


No es posible continuar por esa rama, retrocedemos hasta el primer nodo que tenga nodos adyacentes sin visitar.

Es decir, retrocedemos hasta el 1.

Recorrido: 1, 2, 3, 4, 5, 6, 7, **8**

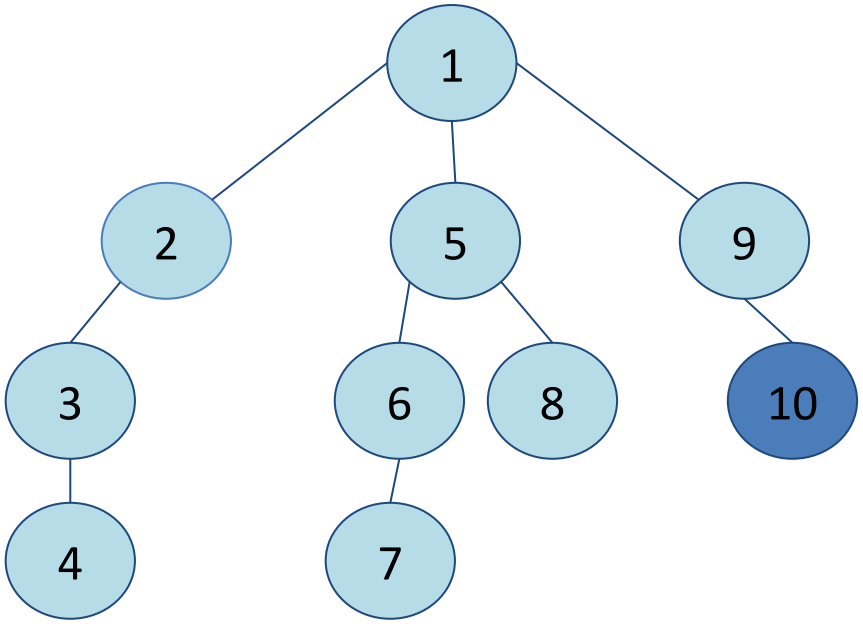
Depth first search



El 1 tiene como vecinos: 2, 5 y 9. Sólo podemos continuar por 9.

Recorrido: 1, 2, 3, 4, 5, 6, 7, 8, **9**

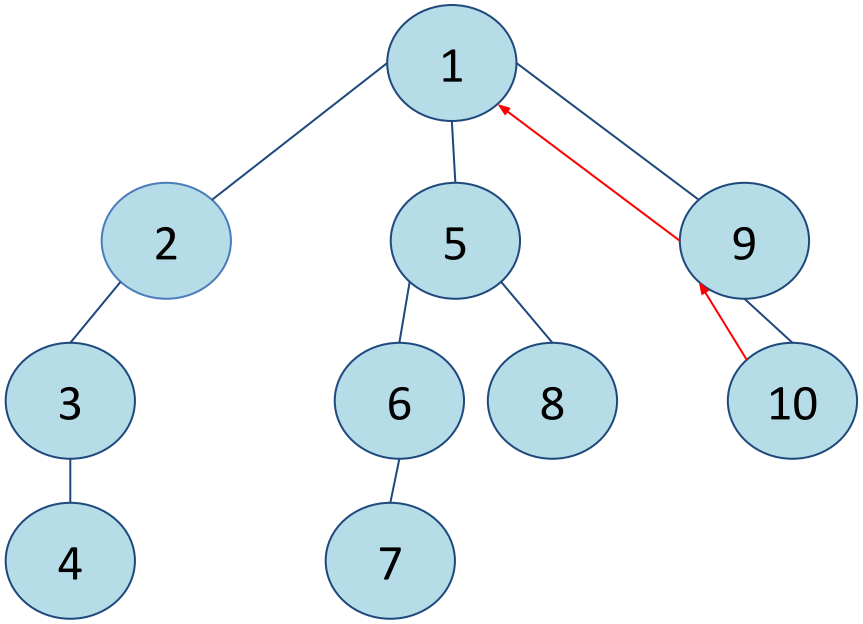
Depth first search



Visitamos el vértice 10. Termina esa rama.

Recorrido: 1, 2, 3, 4, 5, 6, 7, 8, 9, **10**

Depth first search



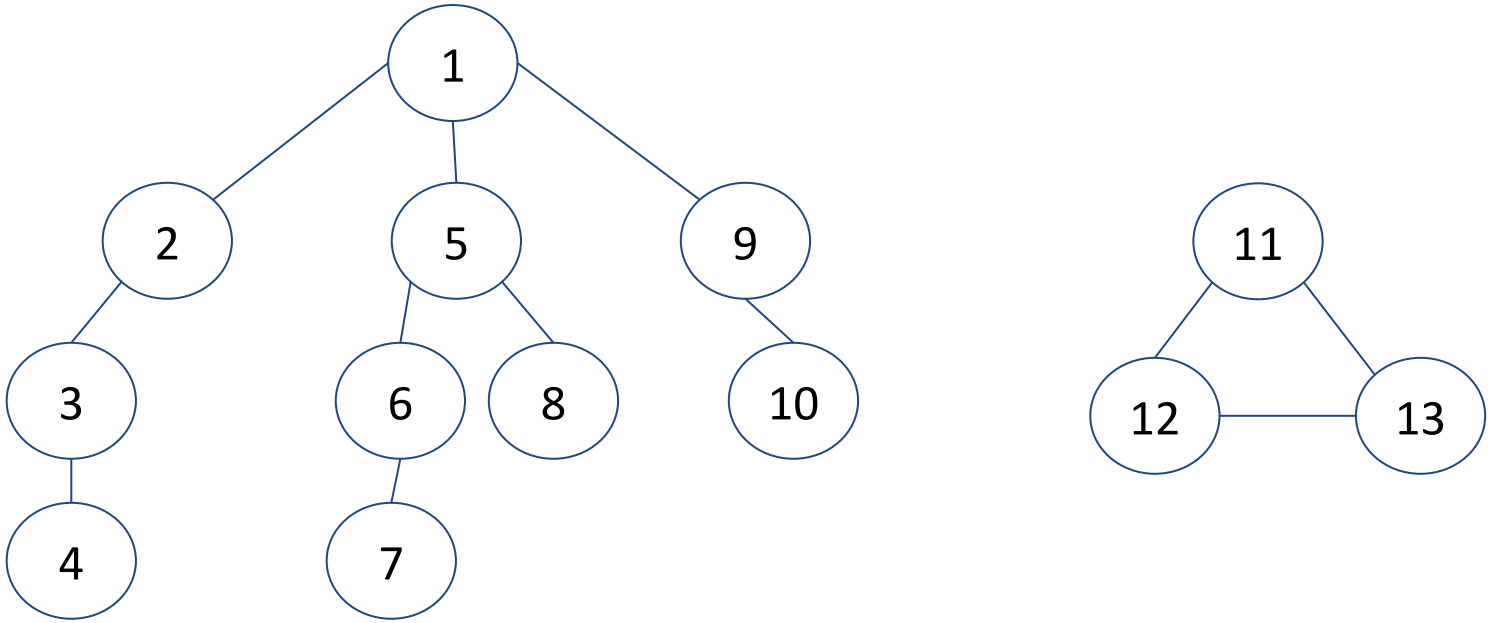
No podemos continuar por el 10.

Al retroceder por el camino no encontramos ninguno nodo con vértices adyacentes sin visitar.

Hemos terminado el recorrido.

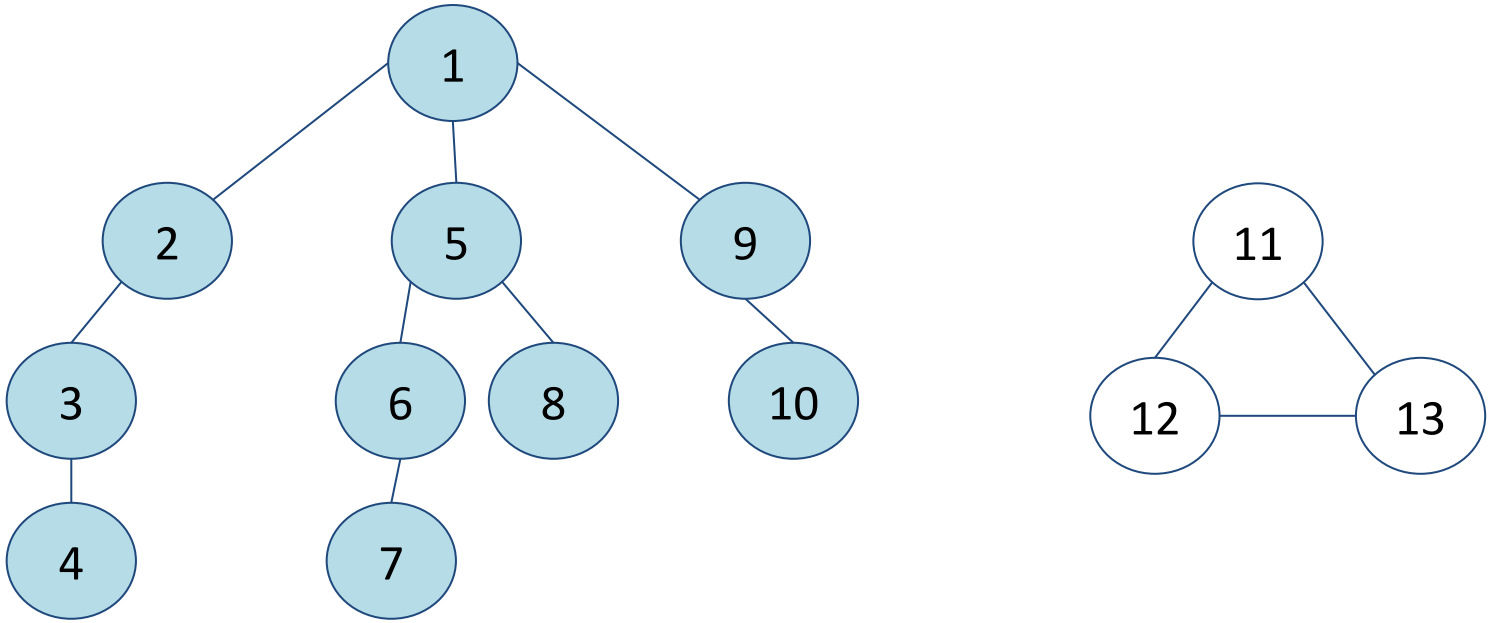
Recorrido: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Depth first search - Grafo no conexo



¿Qué pasaría si el grafo es no conexo (es decir, no existe un camino que conecte todos los vértices)?
¿Cómo es el recorrido desde 1?

Depth first search - Grafo no conexo



¿Cómo es el recorrido desde 1? Los vértices 11, 12 y 13 no formarían parte del recorrido.

recorrido:

1	2	3	4	5	6	7	8	9	10

Depth first search - Implementación

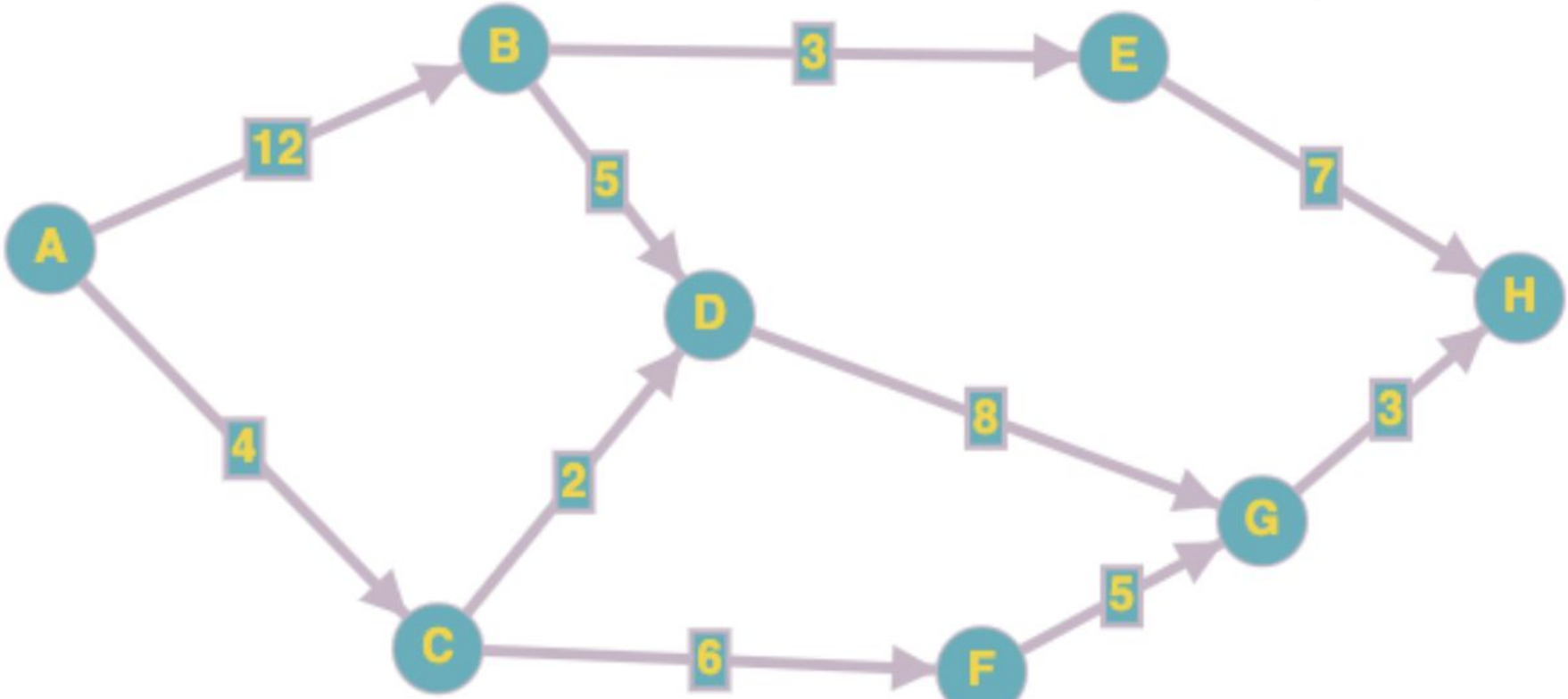
- Ejercicio: En la clase Graph, implementa:
  - El método recursivo dfs_rec(vertex) que devuelve una lista de Python con el recorrido en profundidad del grafo desde el vértice vertex.
    Pistas:
    - Puedes inspirarte en el recorrido recursivo preorder de los árboles
    - Para evitar caer en un ciclo, recuerda usar un diccionario visited de booleanos.
  - El método iterativo dfs_ite(vertex) que devuelve una lista de Python con el recorrido en profundidad del grafo desde el vértice vertex.
    Pistas:
    - Puedes inspirarte en el recorrido iterativo preorder de los árboles (usa una pila para almacenar los vértices adyacentes no visitados).
    - Para evitar caer en un ciclo, recuerda usar un diccionario visited de booleanos.
- Soluciones

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

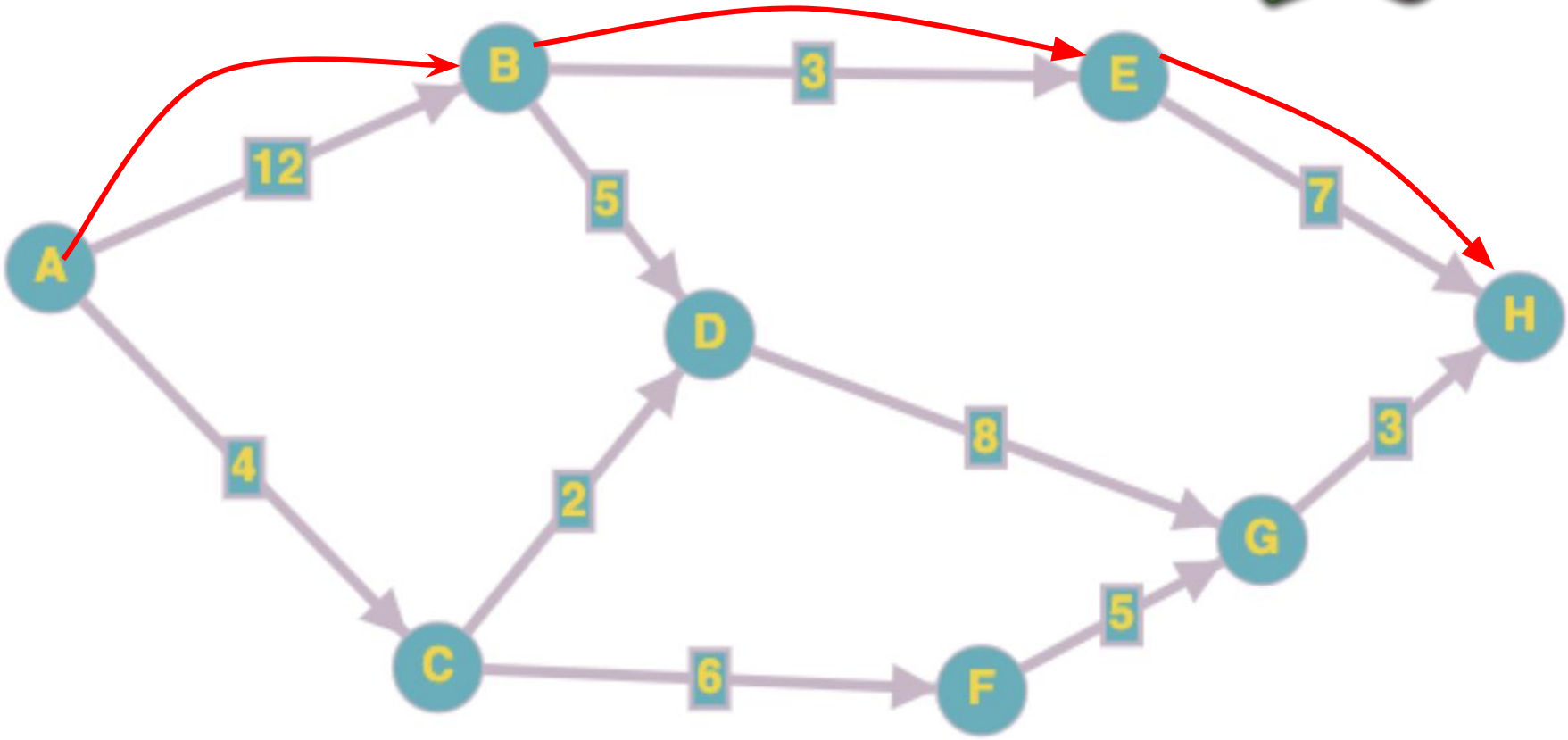


Creado con la herramienta online https://graphonline.ru/en/

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

¿Cómo encontrar el camino más corto entre dos ciudades?

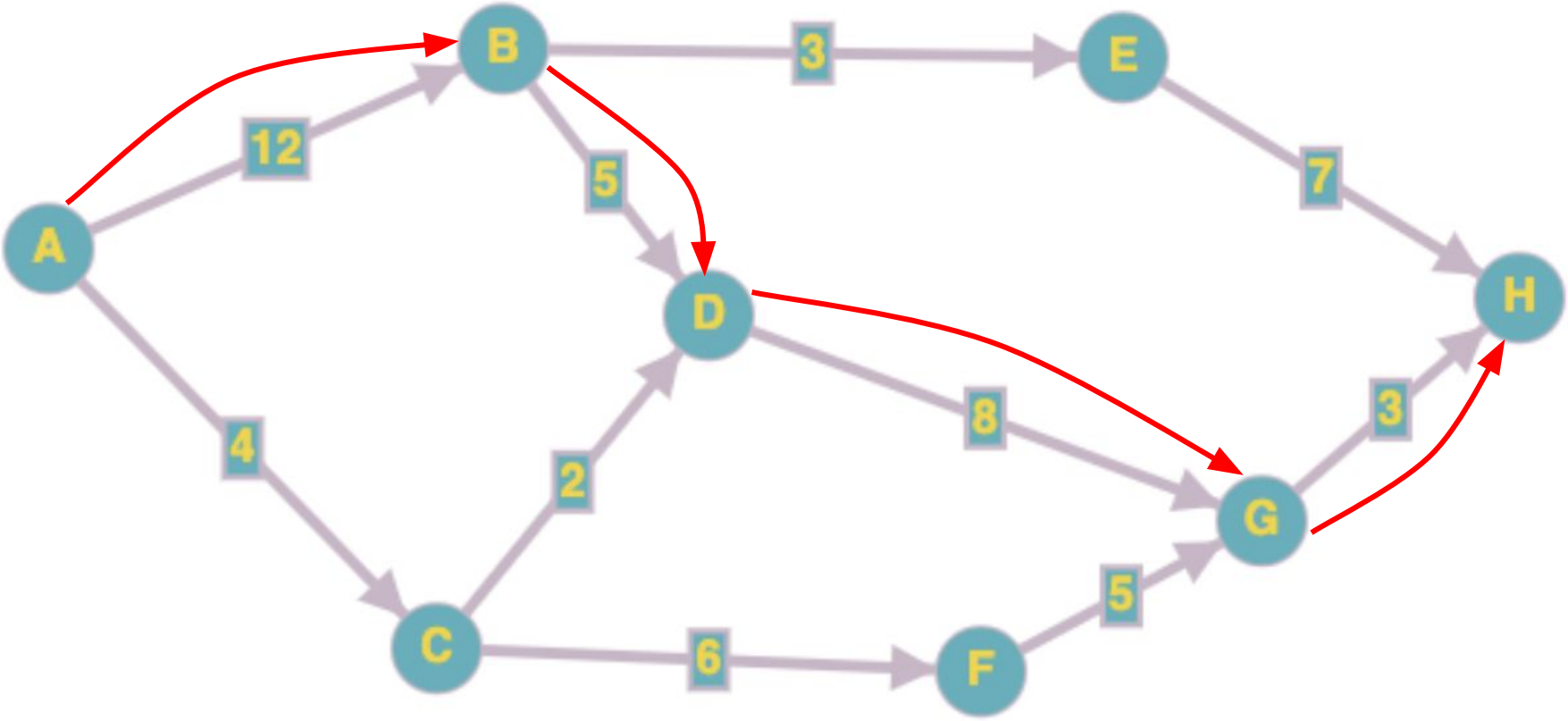
Ruta 1: A->B->E->H (22 km)



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

¿Cómo encontrar el camino más corto entre dos ciudades?

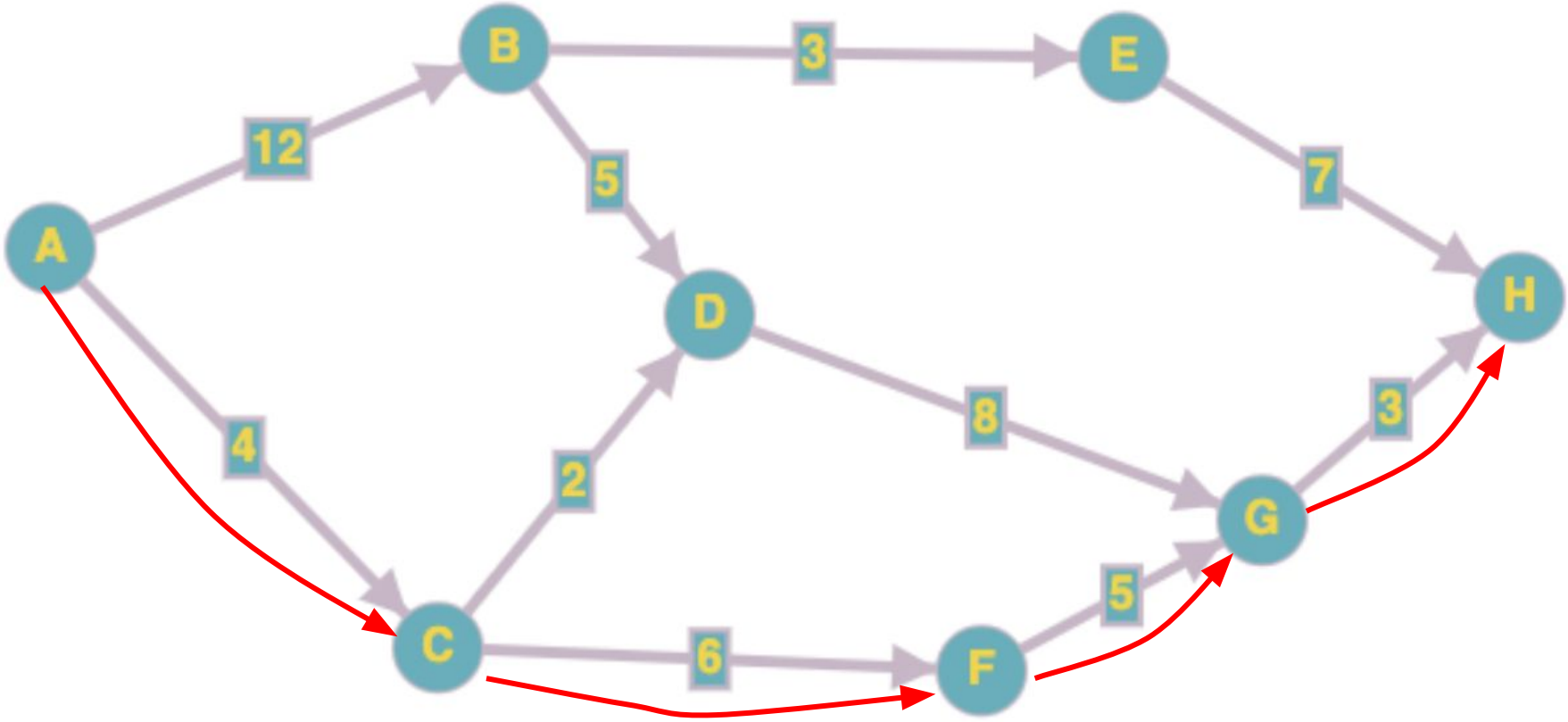
Ruta 2: A->B->D->G->H (28 KM)



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

¿Cómo encontrar el camino más corto entre dos ciudades?

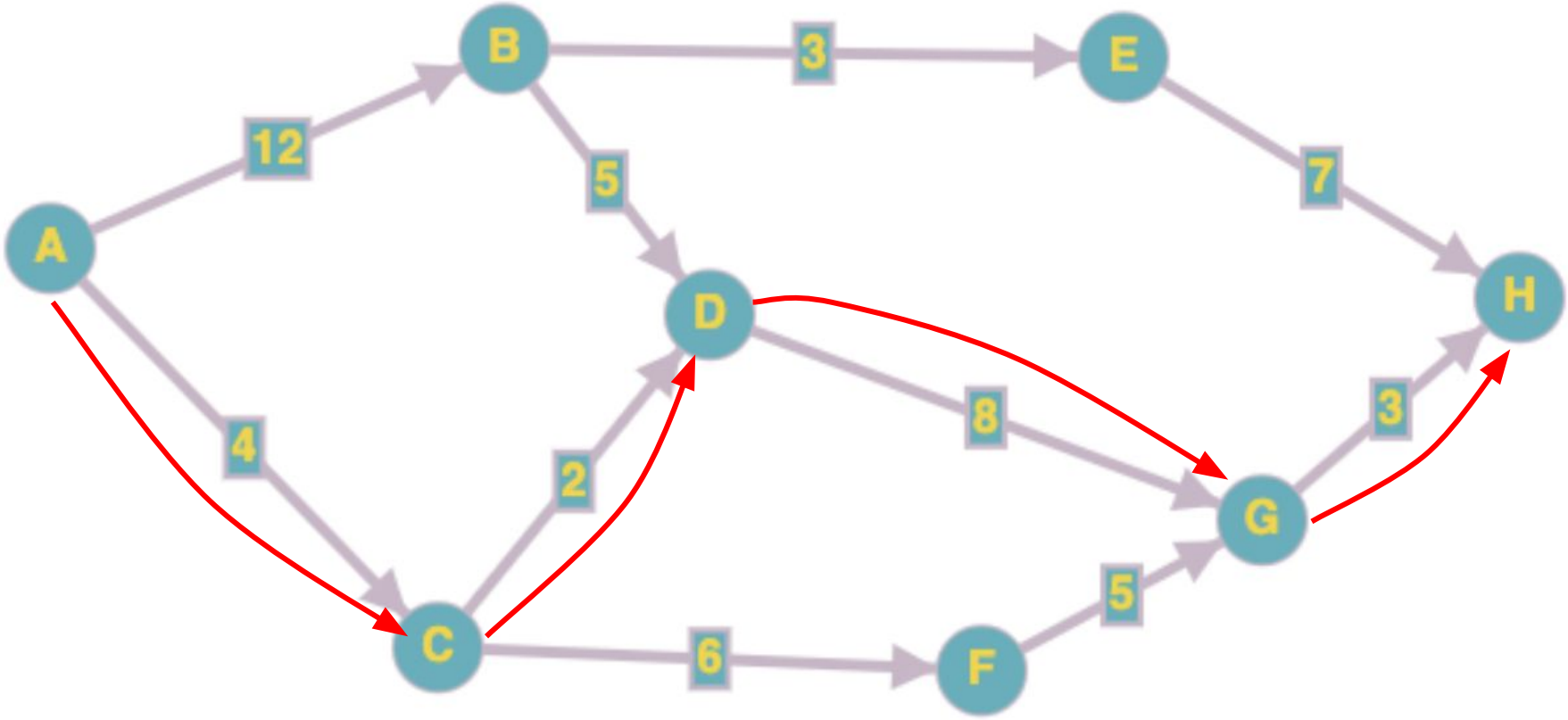
Ruta 3: A->C->F->G->H (18 km)



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

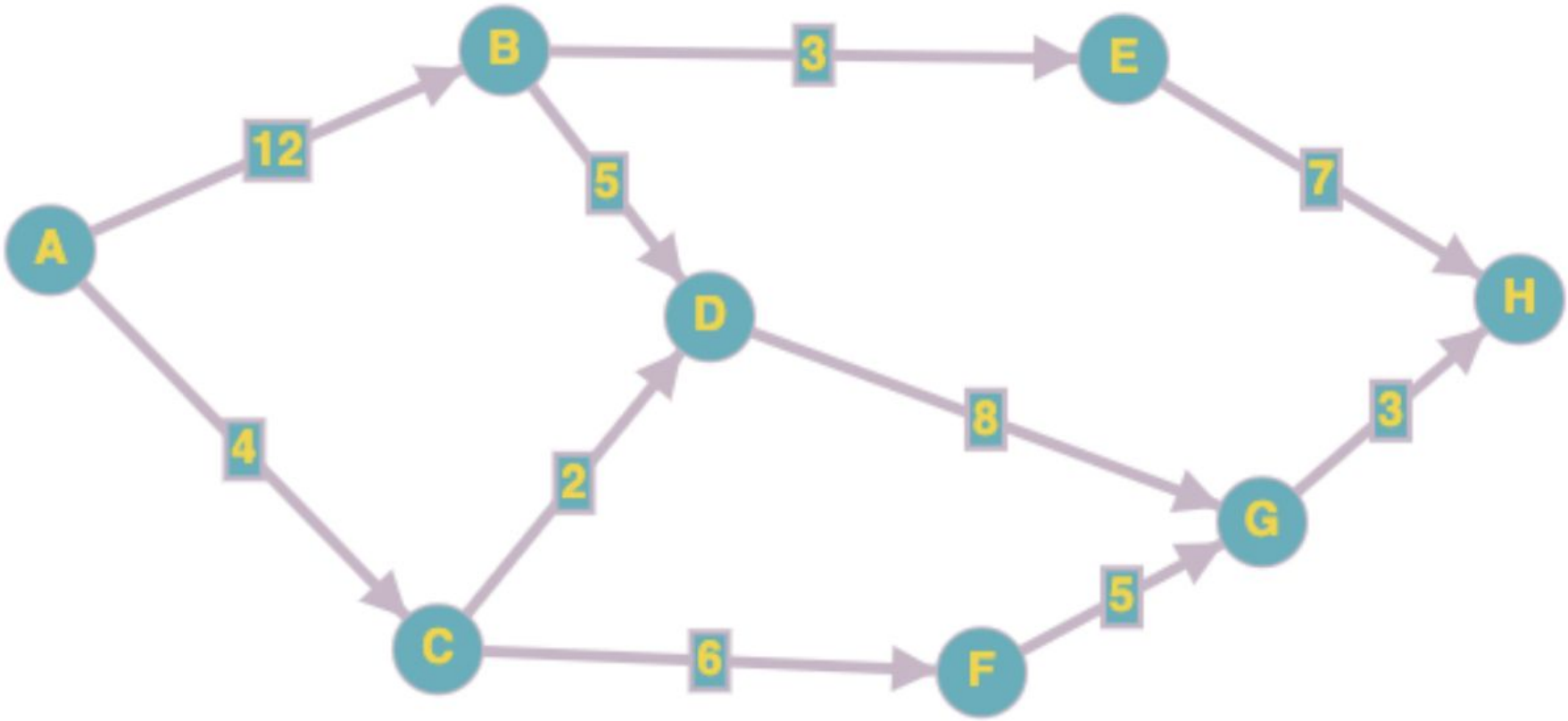
¿Cómo encontrar el camino más corto entre dos ciudades?

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

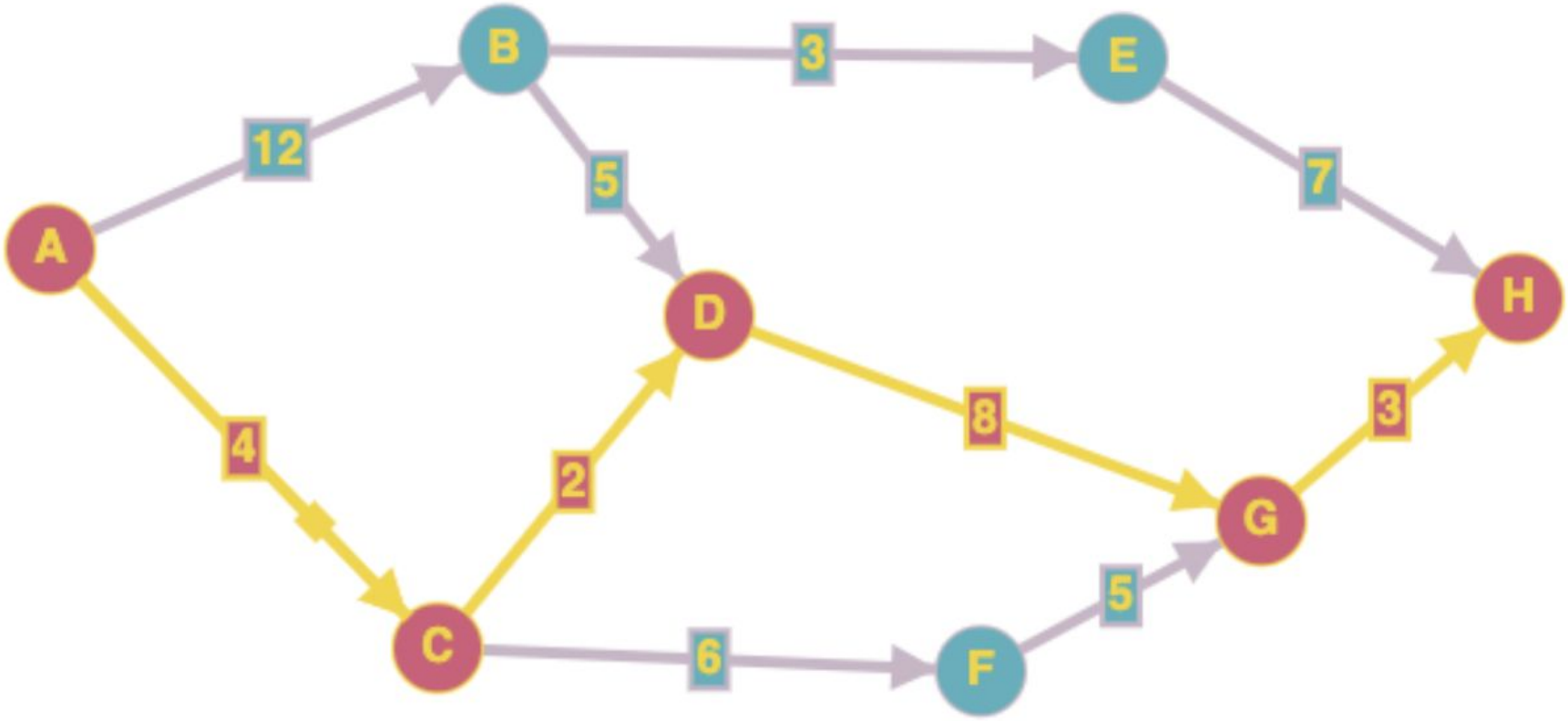


Creado con la herramienta online https://graphonline.ru/en/

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

Algoritmo de Dijkstra

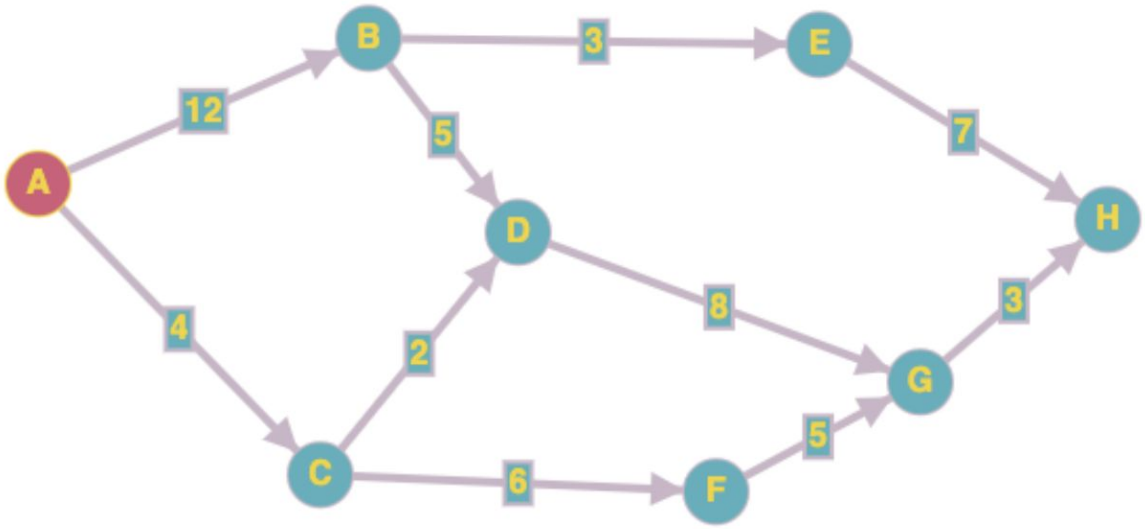
Camino mínimo de A a H:



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.

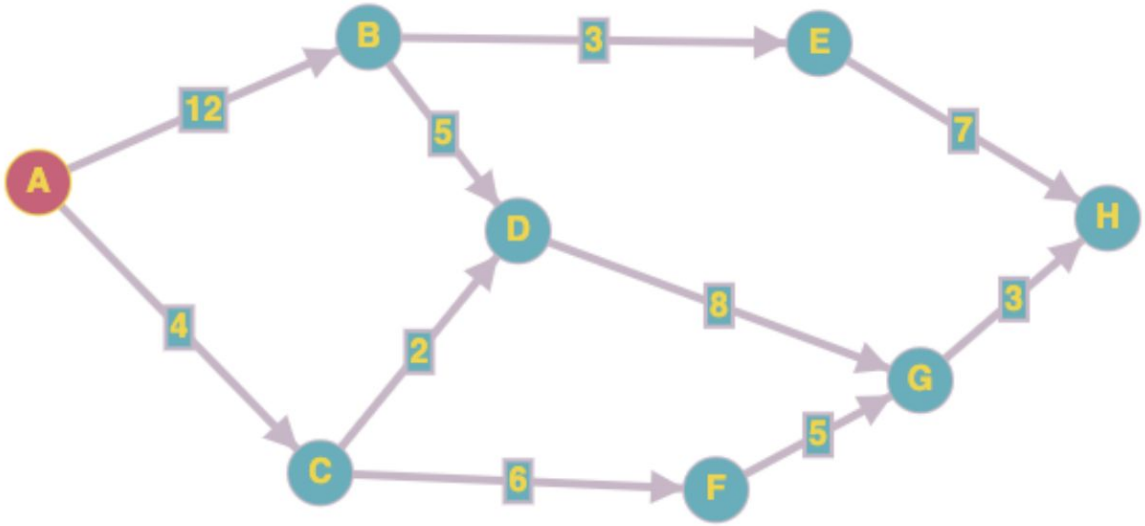
Algoritmo de Dijkstra - Inicialización

```
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

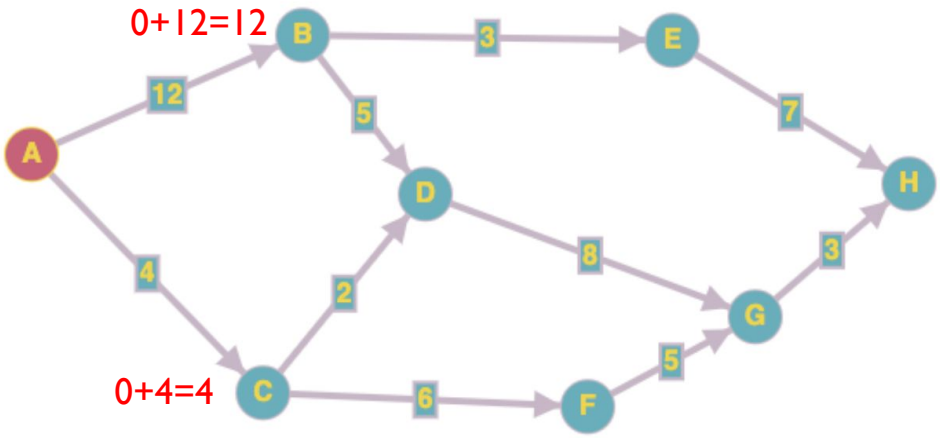
```
for _ in range(len(self._vertices)):

    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.



	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

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

Algoritmo Disjkstra- bucle principal

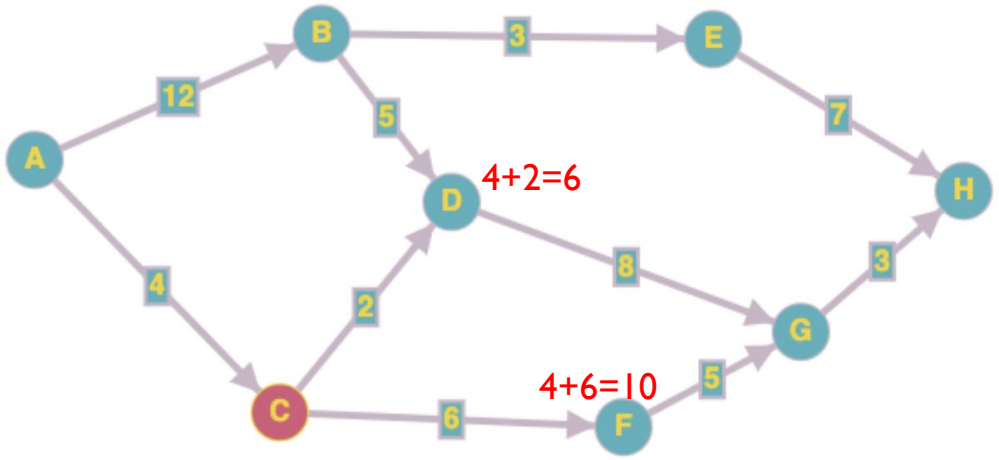
```
for _ in range(len(self._vertices)):

    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.



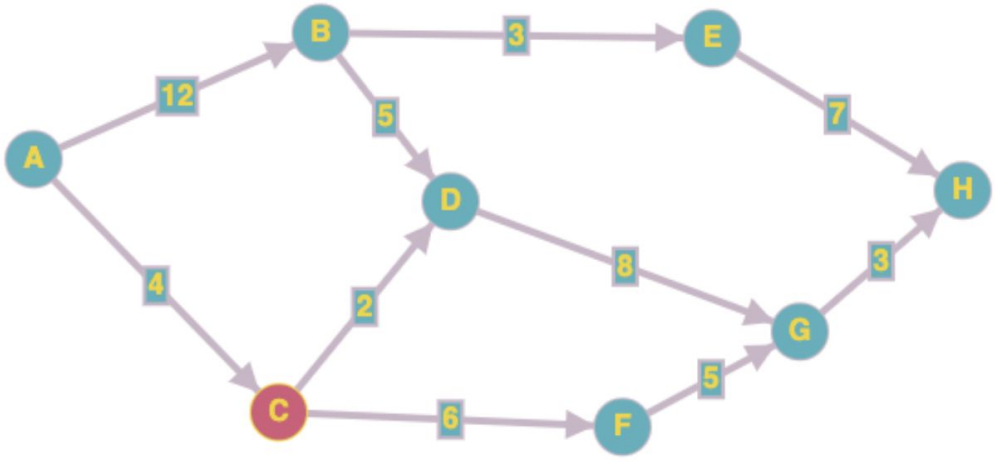
distancia[D]= ∞ > distancia[C] + 2 = 6 => Debemos actualizar

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?

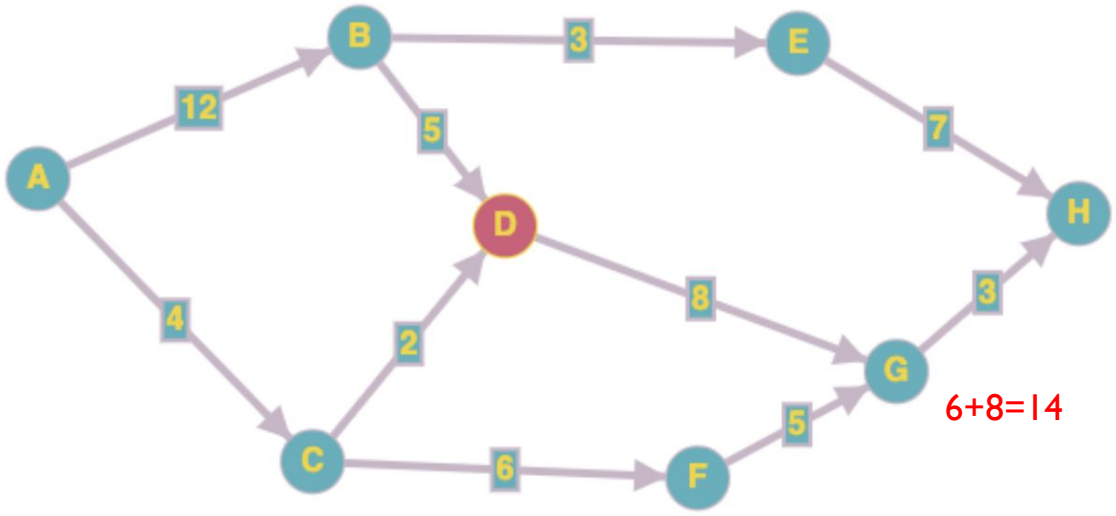
Además, guardamos C como vértice anterior para D y F.
Termina esa iteración, y el bucle sigue iterando (3 iteración < 8 número vértices).
Debemos seleccionar el vértice no visitado con menor distancia acumulada: D



	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.

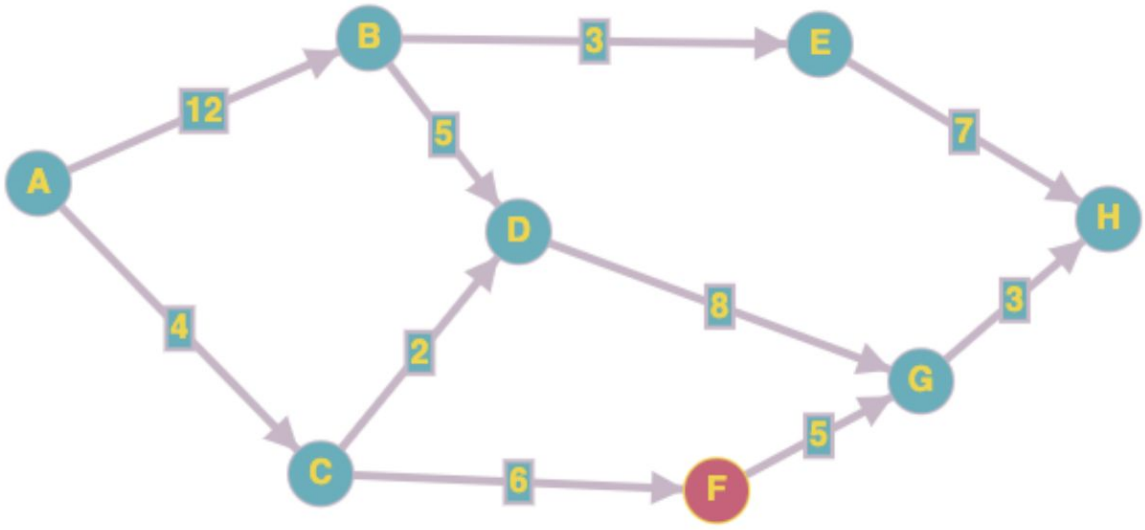


visitados[D]=True #[A,C,D] ya han sido
distancia[G]=∞ > distancia[D] + 8 = 14 =>
Debemos modificar

	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.

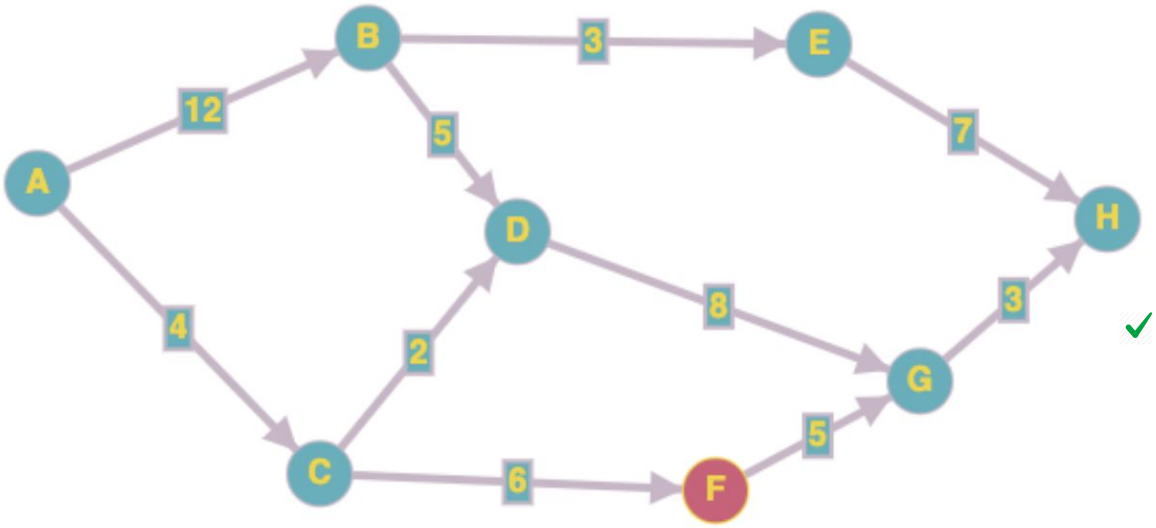


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D ✓	6	C
E	∞	
F ✓	10	C
G	14	D
H	∞	

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.

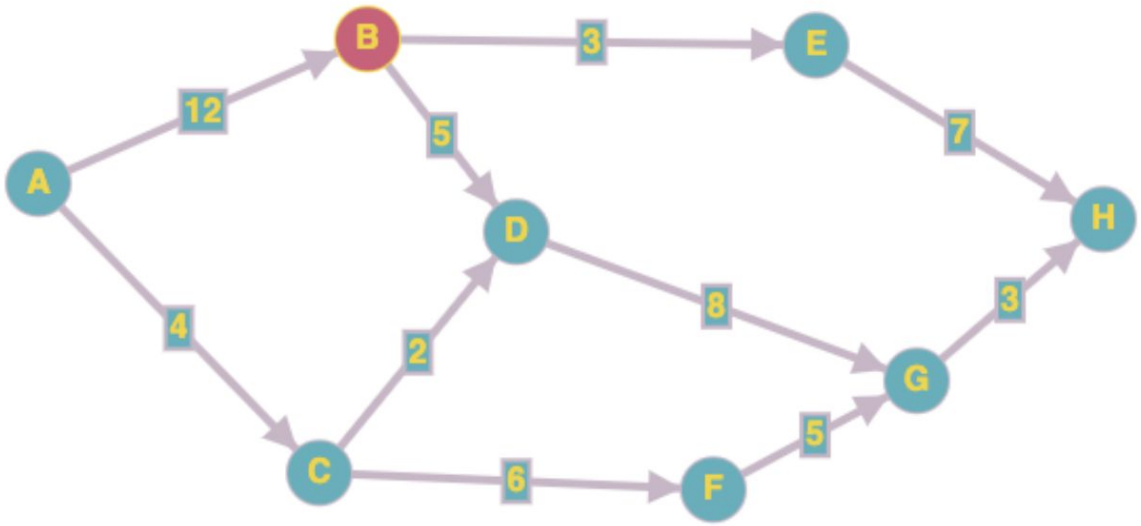


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B	12	A
C ✓	4	A
D ✓	6	C
E	∞	
✓ F	10	C
G	14	D
H	∞	

Termina la cuarta iteración, Seguimos iterando porque 5 < 8

Algoritmo de Dijkstra - Cómo?

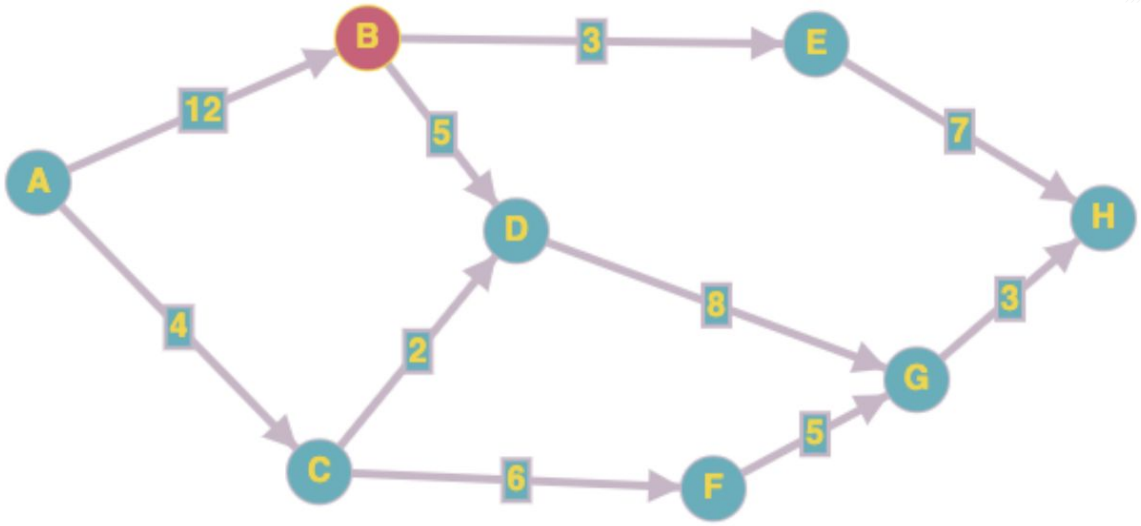
En la 5ª iteración, debemos seleccionar el vértice no visitado ({B, E, G, H},) con menor distancia acumulada: B



	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?

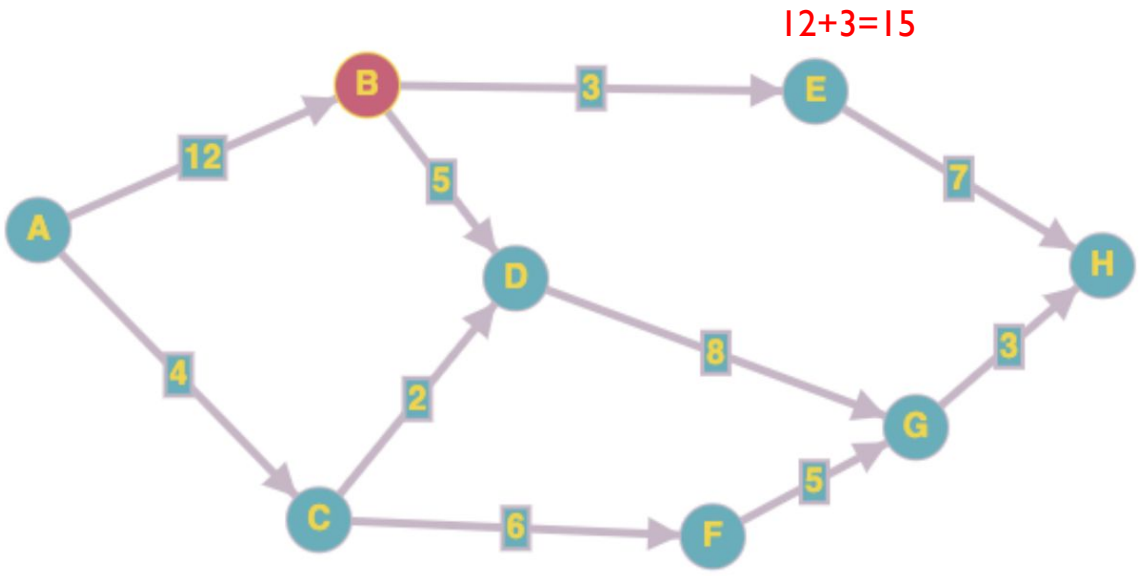
Marcamos B visitado, visitado[B]=True, y recuperamos sus adyacentes no visitados para comprobar:
d[E]= ∞ > d[B] + 3 = 15 => Modificar



	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

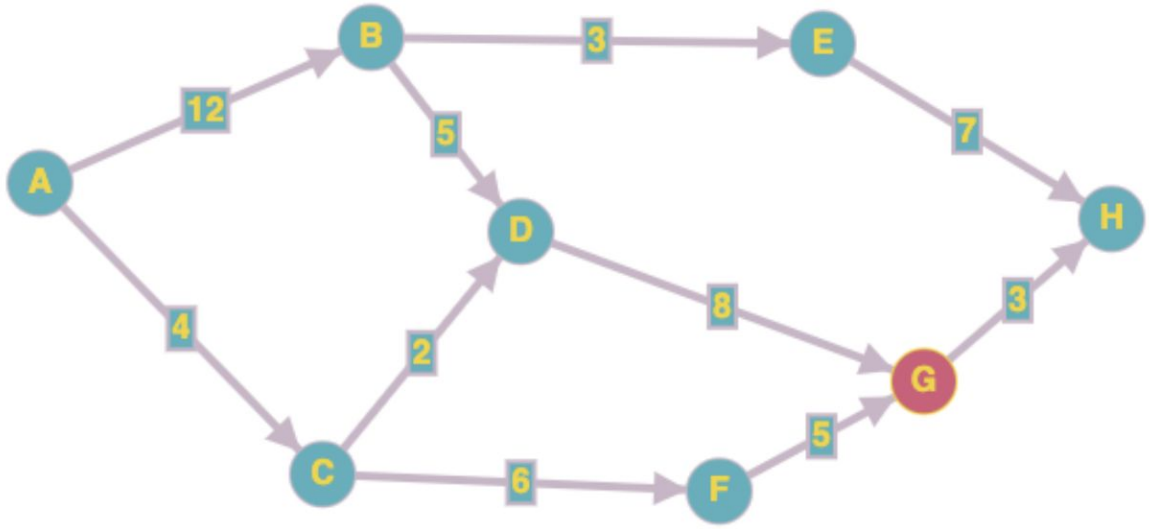


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E	15	B
F ✓	10	C
G	14	D
H	∞	

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.

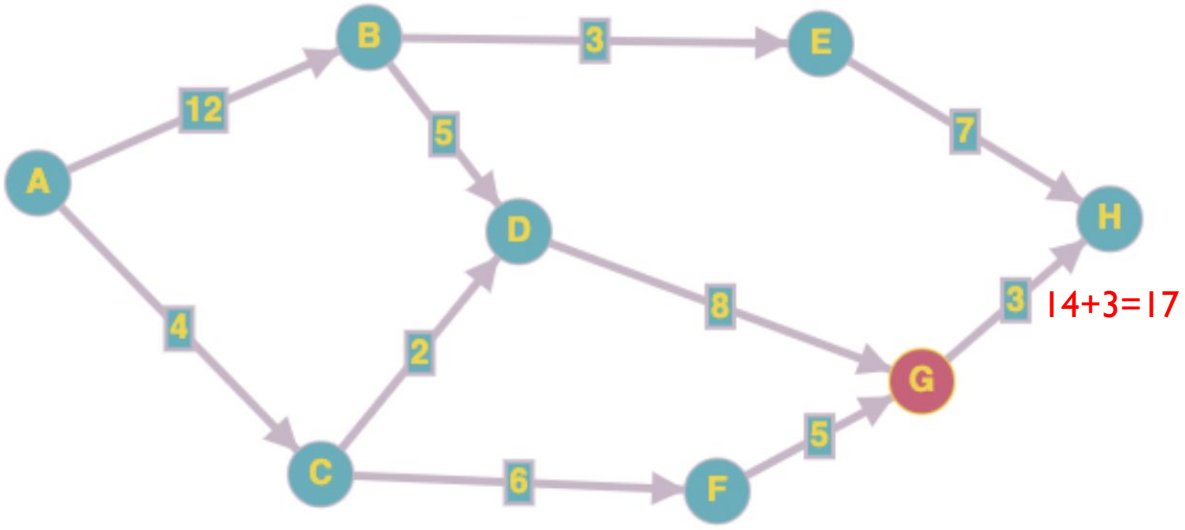


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E	15	B
F ✓	10	C
✓ G	14	D
H	∞	None

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

Algoritmo de Dijkstra - Cómo?

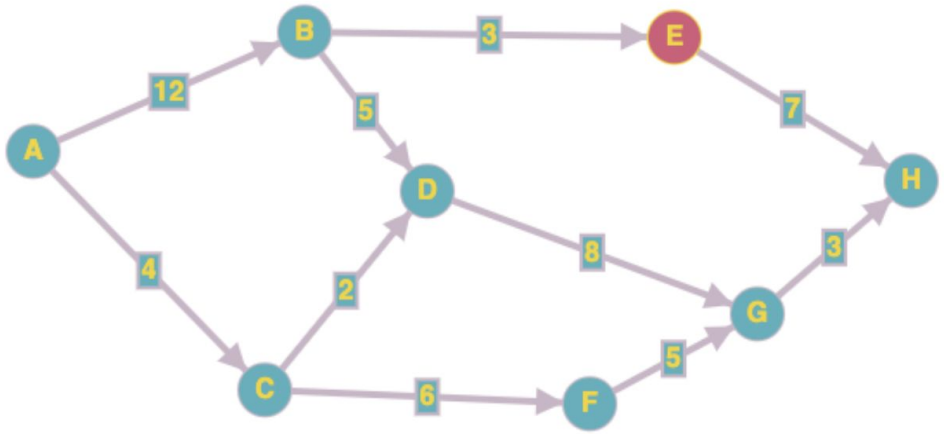
También debemos actualizar el predecesor de H a G.
Comenzamos la 7ª iteración (< 8), los vértices no visitados son {E, H}, siendo E el de menor distancia acumulada



	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.

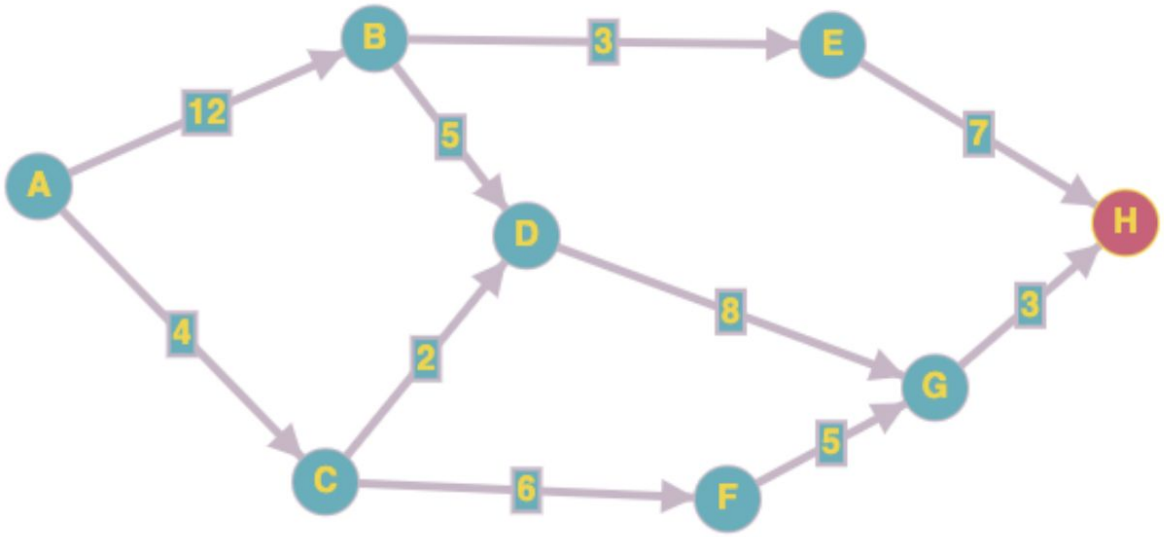


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
✓ E	15	B
F ✓	10	C
C ✓	14	D
H	17	G

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.

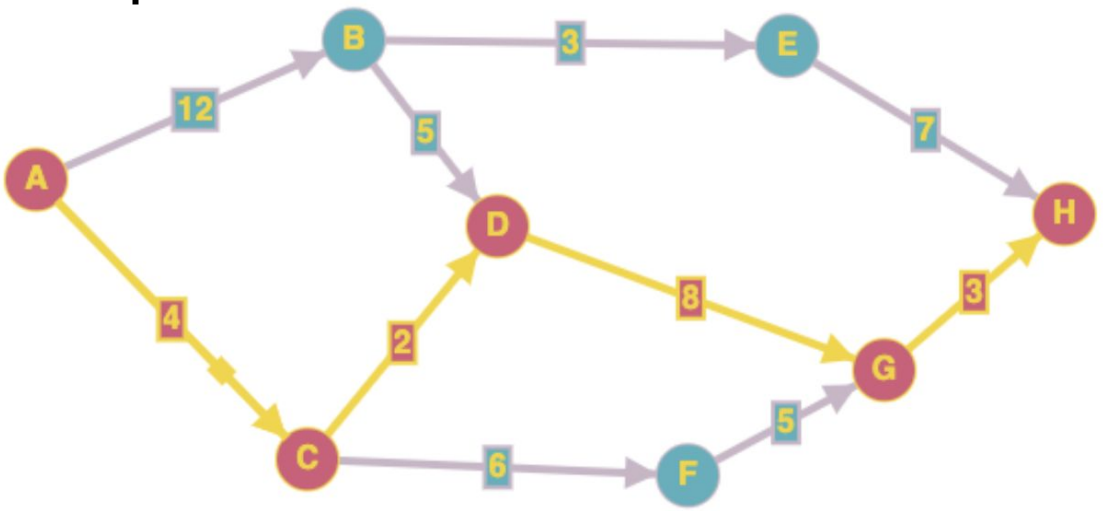


	Longitud más corto desde A	Vértice anterior
A ✓	0	-
B ✓	12	A
C ✓	4	A
D ✓	6	C
E ✓	15	B
F ✓	10	C
G ✓	14	D
H	17	G

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



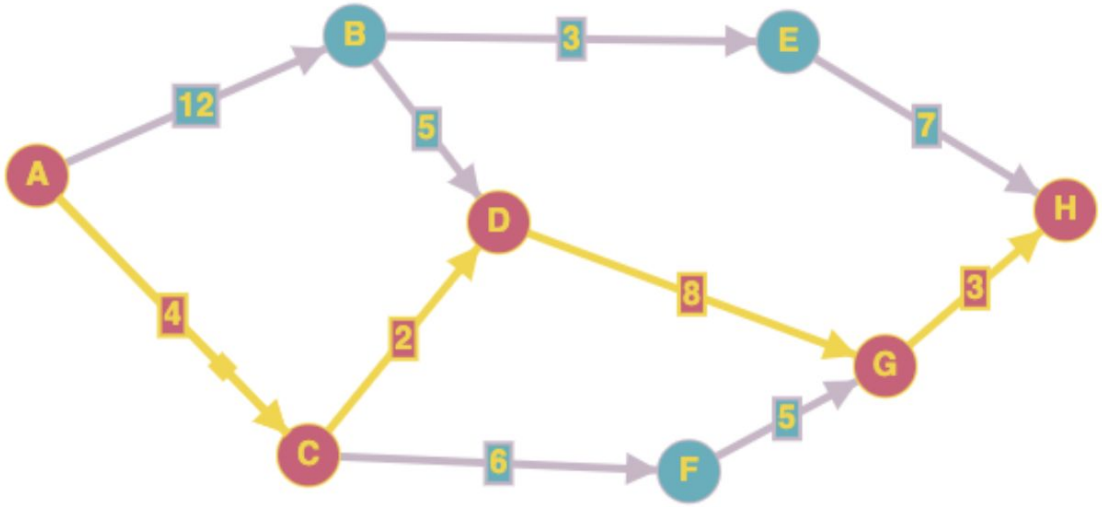
	Longitud más corto desde A	Vértice anterior
A	0	-
B	12	A
C	4	A
D	6	C
E	15	B
F	10	C
G	14	D
H	17	G

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

I * 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)

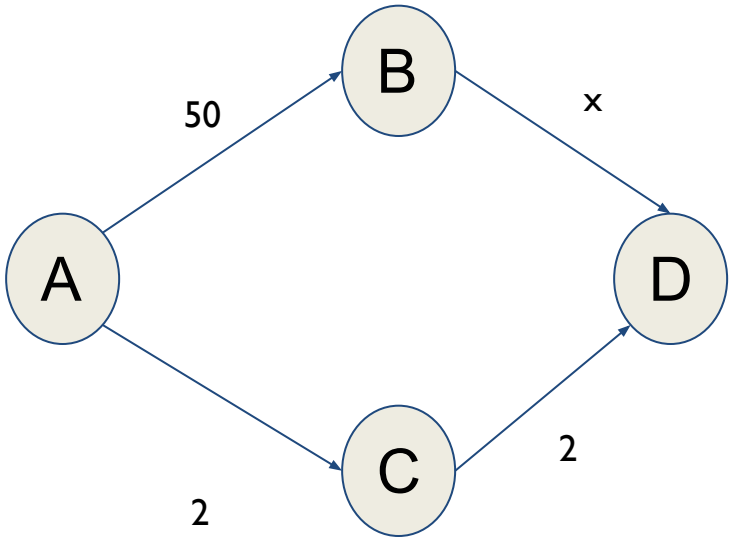
Análisis de complejidad temporal

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



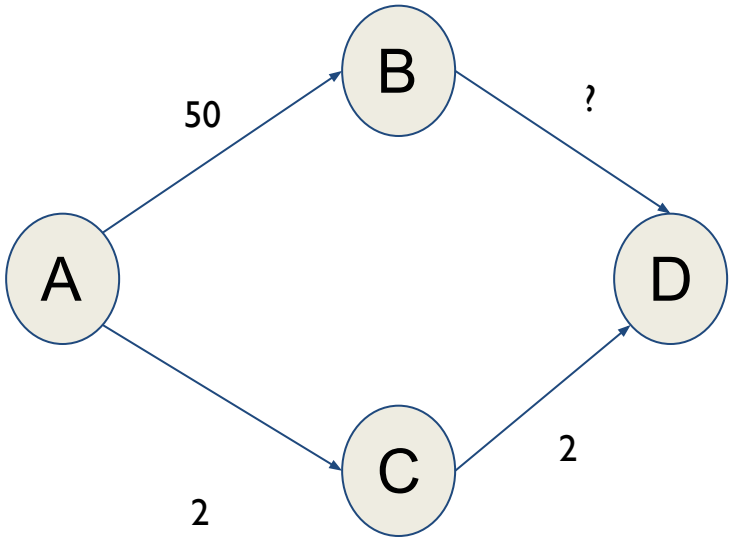
Aplicando el **algoritmo de Dijkstra**, vamos a calcular el camino mínimo de A al resto de vértices del grafo. Vamos a suponer que no conocemos el peso de la arista B -> D, únicamente que es positivo x>=0

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



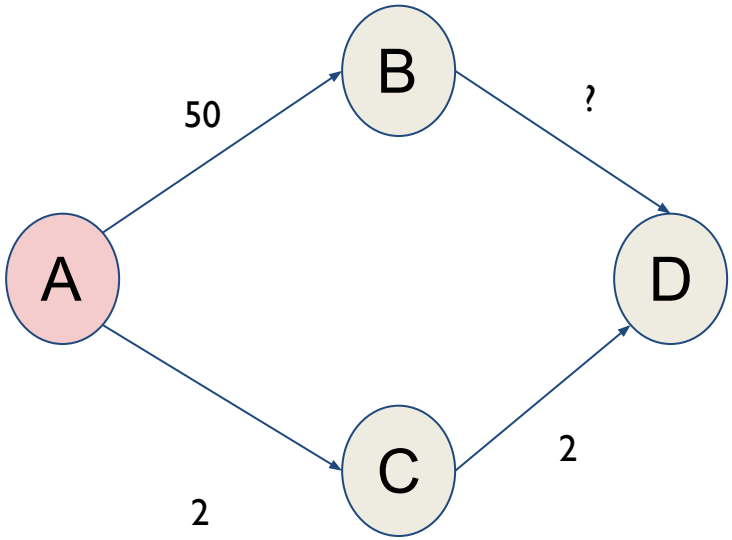
	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	∞	-
C	∞	-
D	∞	-

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



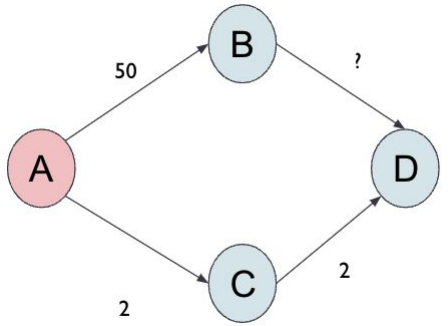
visited[A] = True

	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
```

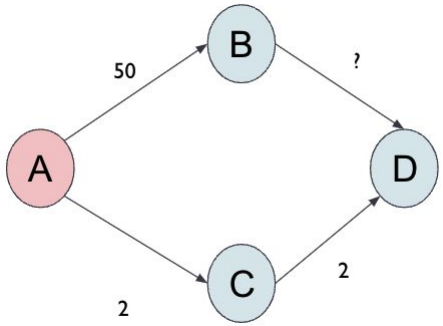
Recuperamos los vértices adyacentes del vértice que estamos visitando.
En nuestro caso A, sus adyacentes son B y C.



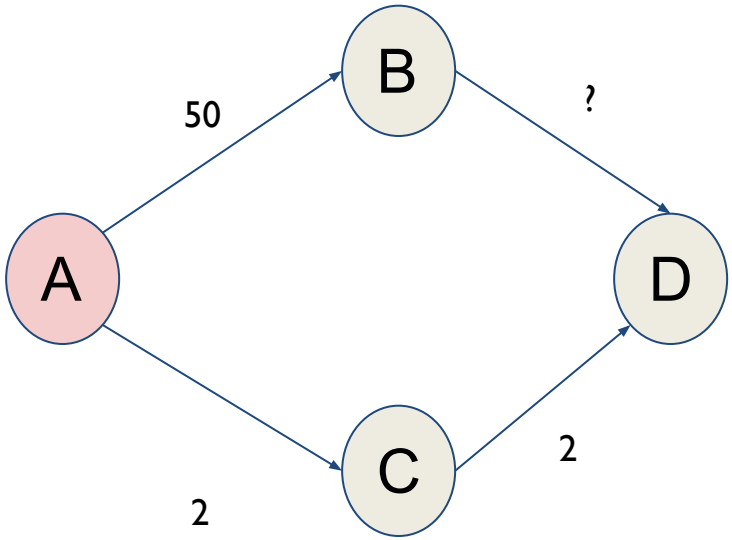
Ejemplo

```
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

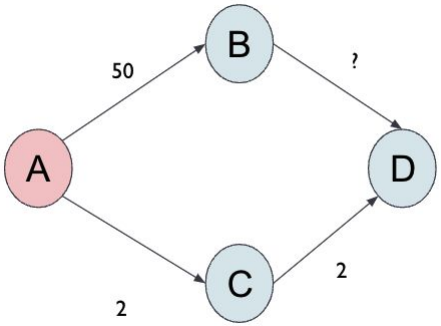


	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	0+50	A
C	∞	-
D	∞	-

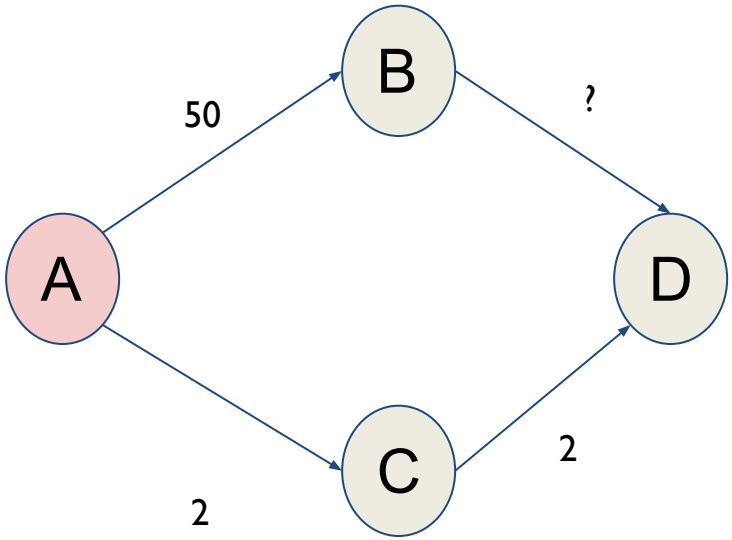
Ejemplo

```
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



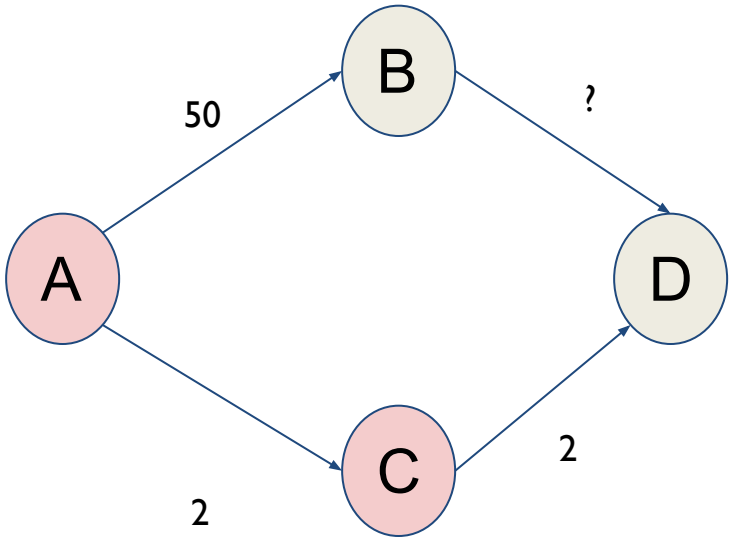
	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	∞	-

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



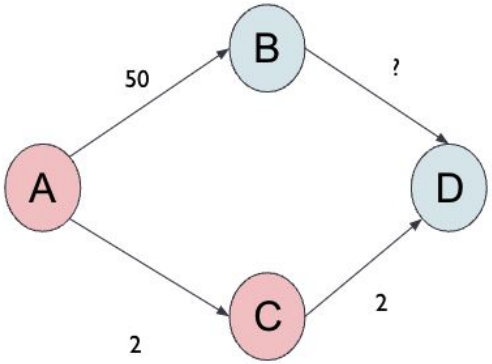
visited[C] = True

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

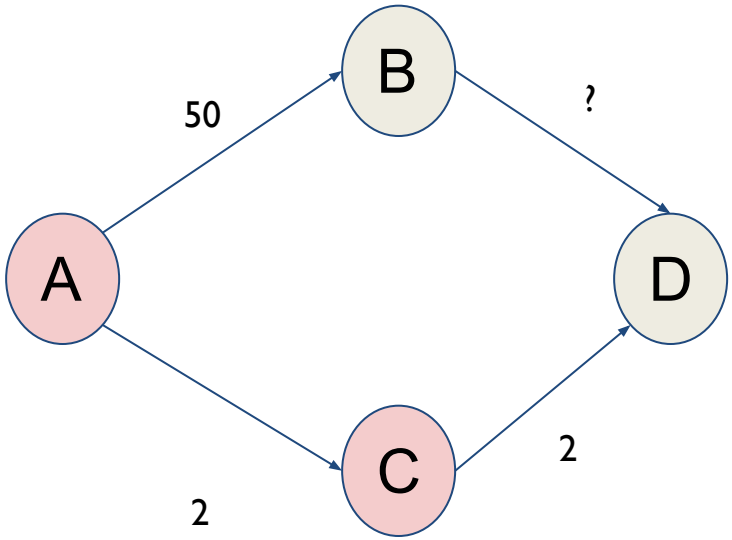
Ejemplo

```
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.

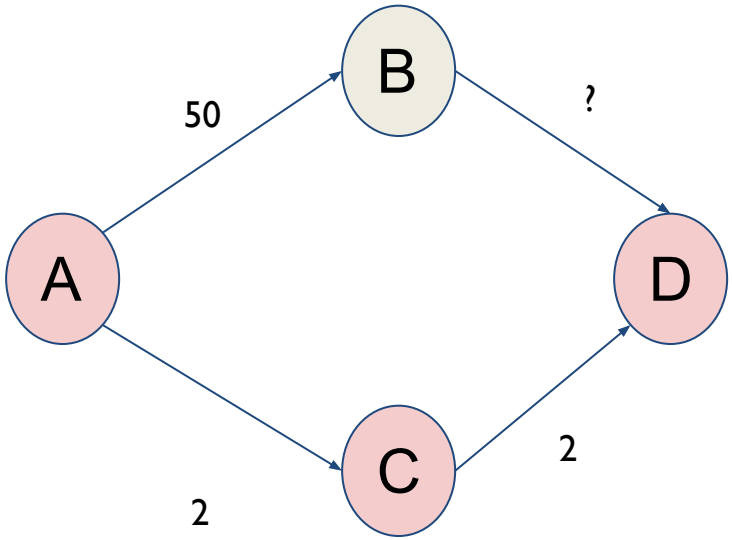
Ejemplo

```
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



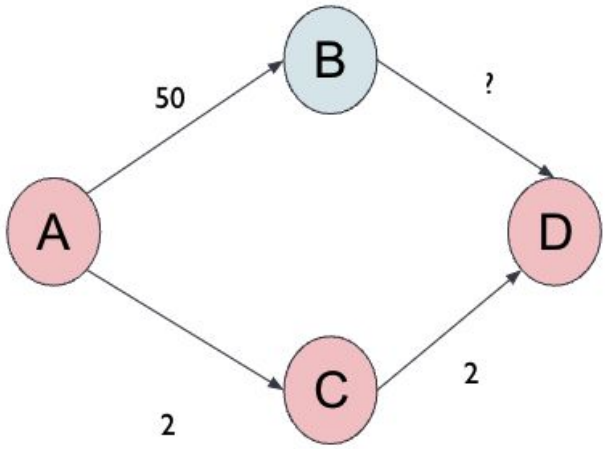
visited[D] = True

	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



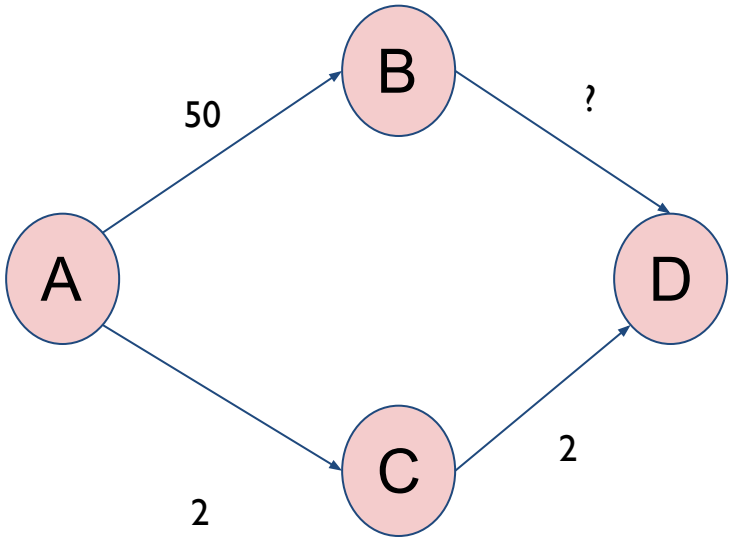
Ejemplo

```
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



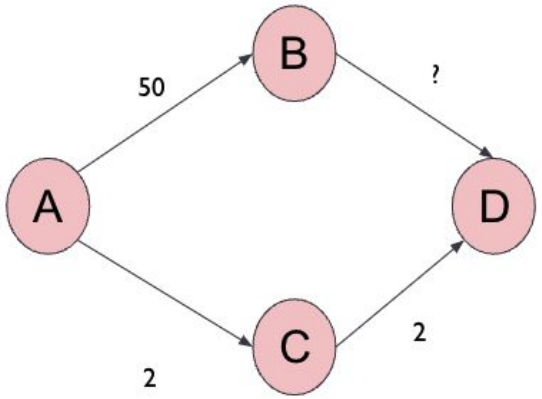
visited[B] = True

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

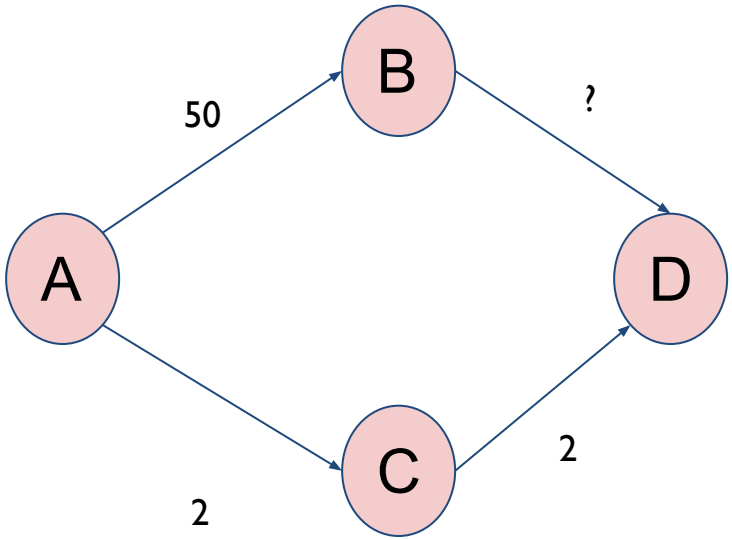
Ejemplo

```
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

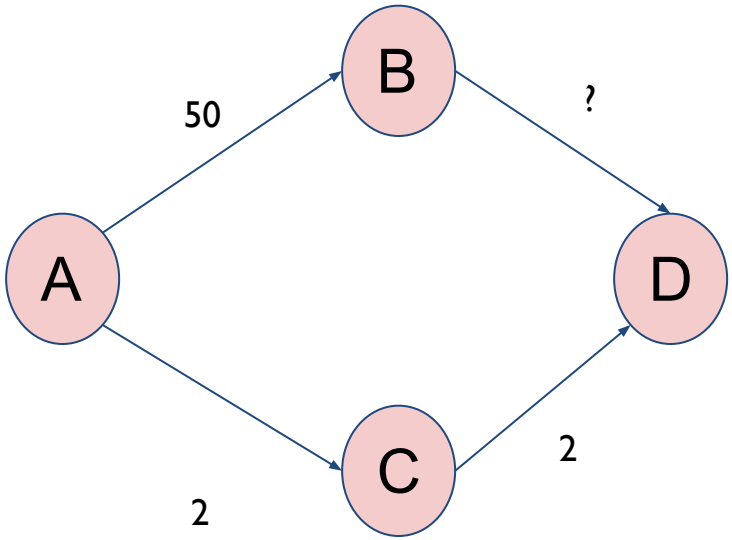


	Min. distancia desde A	Vértice anterior (en el camino desde A)
A	0	-
B	50	A
C	2	A
D	4	C

Caminos mínimos:

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

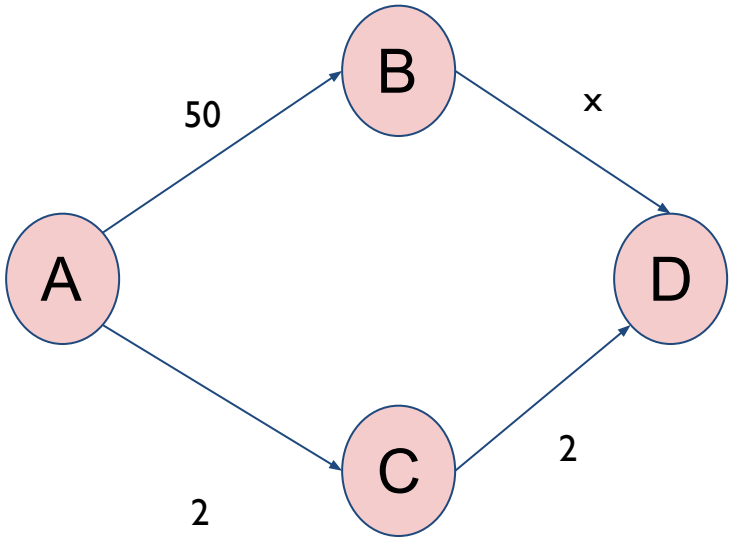
Ejemplo



	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?:

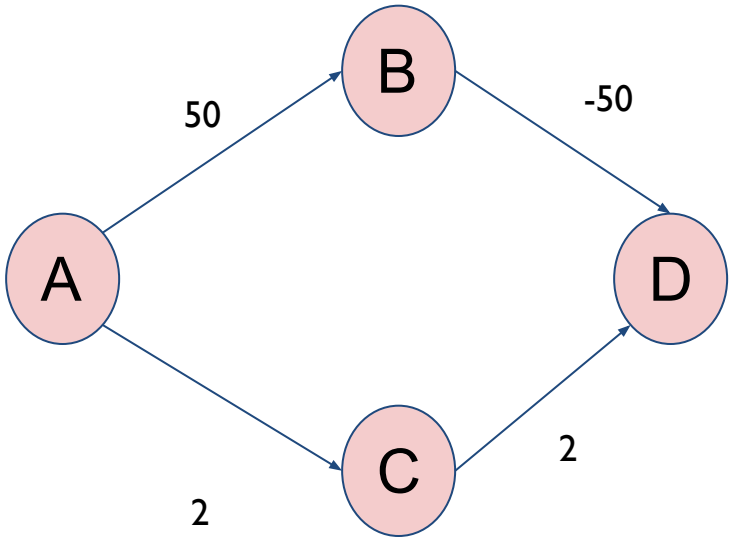
Ejemplo



	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.