Universidad Nacional Autónoma de México
Facultad de Ingeniería
Departamento de Computación
Estructuras de datos y algoritmos II

Tema 3

Algoritmos de grafos

3 Algoritmos de grafos

Objetivo: Aplicar las formas de representar y operar la estructura de datos grafo para representarlo en la computadora.

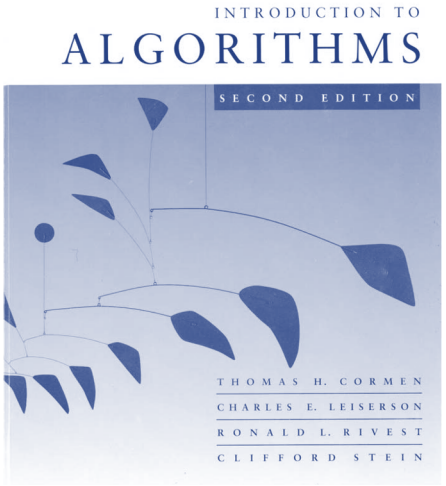
3 Algoritmos de grafos

3.1 Representación de grafos.

3.2 Búsqueda por expansión.

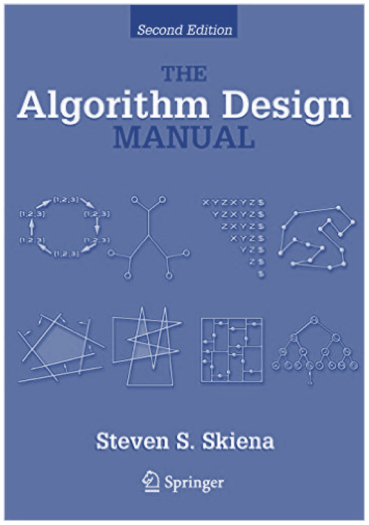
3.3 Búsqueda por profundidad.

Bibliografía



- Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, McGraw-Hill.

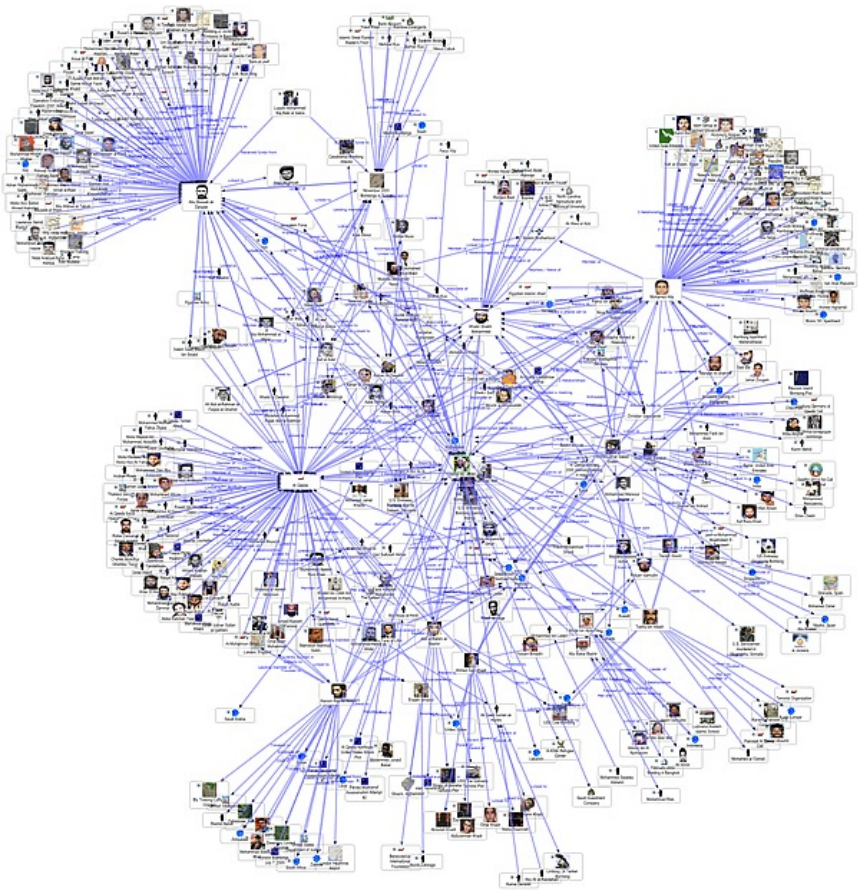
Bibliografía



- The Algorithm Design Manual. Steven S. Skiena, Springer.

3.1 Representación de grafos.

Algoritmos de grafos



Las estructuras o listas no lineales son estructuras de datos cuyas relaciones son multidimensionales, es decir, a cada nodo de la estructura pueden estar ligados uno o varios nodos.

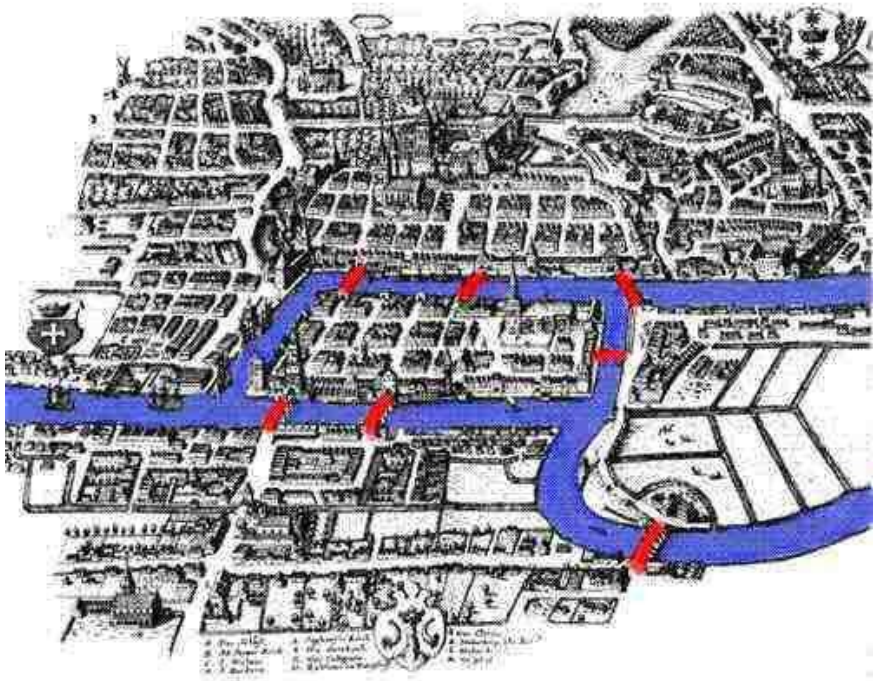
Las listas no lineales más representativas son las gráficas (o grafos) y los árboles.

Gráficas o grafos

Las gráficas son estructuras de datos no lineales donde cada elemento puede tener uno o más predecesores y uno o más sucesores.

Un grafo está compuesto por dos elementos básicos: nodos (vértices) y arcos (aristas). Las aristas se encargan de conectar los nodos del grafo.

Leonhard Euler (1707-1783) escribió lo que se conoce como el primer paper de la teoría de grafos, los 7 puentes de Königsberg. La ciudad de Königsberg, que en el siglo XVIII era una ciudad de Prusia a las orillas del río Pregel.



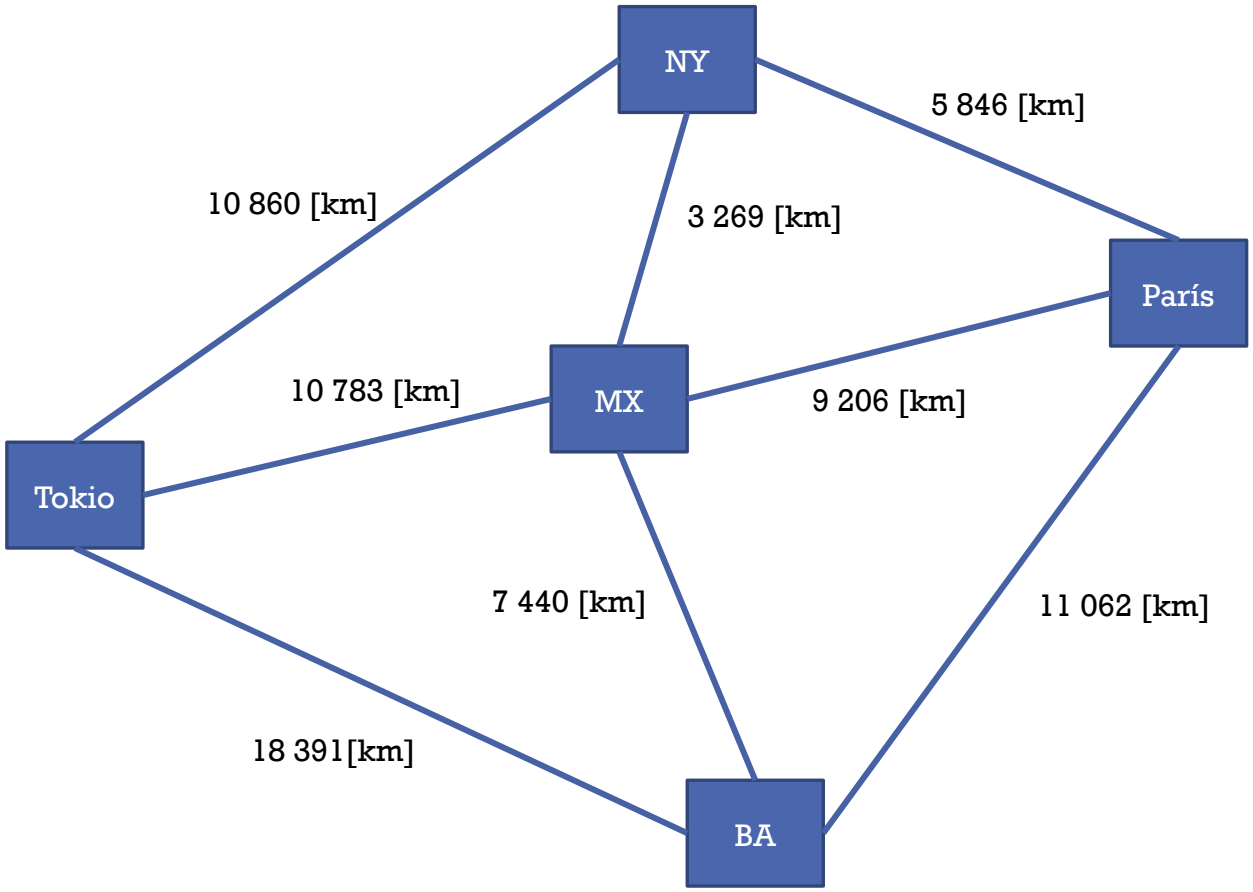
De manera general, el problema de los 7 puentes de Königsberg consiste en la siguiente pregunta: ¿Puedes caminar a través de la ciudad visitando cada parte de la ciudad y cruzando cada puente una sola vez?

Este problema fue resuelto por Leonhard Euler a través de teoría de grafos.

Como se puede deducir, la teoría de grafos brinda solución a muchas situaciones cotidianas:

- Durante la guerra permitió decidir qué calles se deberían bombardear para desconectar la capital de la ciudad de las ciudades aledañas.
- En un navegador permite buscar una página de internet o localizar un teléfono viajando a través de routers.
- En una aplicación de localización permite encontrar la distancia más corta entre un lugar y otro.

Los vértices almacenan información y las aristas representan la relación que existe entre dicha información.



Una gráfica G es una estructura matemática utilizada para modelar relaciones entre pares. Se denota como G = {V, A} y está formada por dos conjuntos: V(G) y A(G).

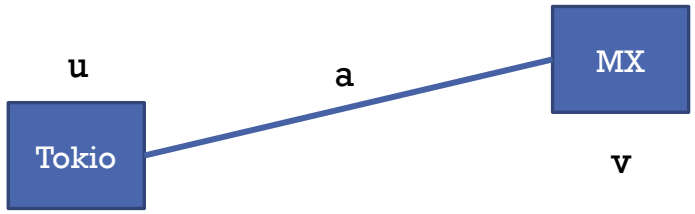
V(G) se refiere a los nodos, puntos o vértices de la gráfica G. A(G) se refiere a los arcos, aristas o líneas de G que unen a los nodos.

A menos que se especifique lo contrario, se entiende que los conjuntos V(G) y A(G) son finitos.

Cada arista solo puede unir un par de nodos del conjunto V(G). Una arista que enlaza al nodo *u* con el nodo *v* se denota como:

$$a=(u, v)$$

Si *u* y *v* están conectados por ***a,*** se dice que *a* es incidente en *u* y *v*.



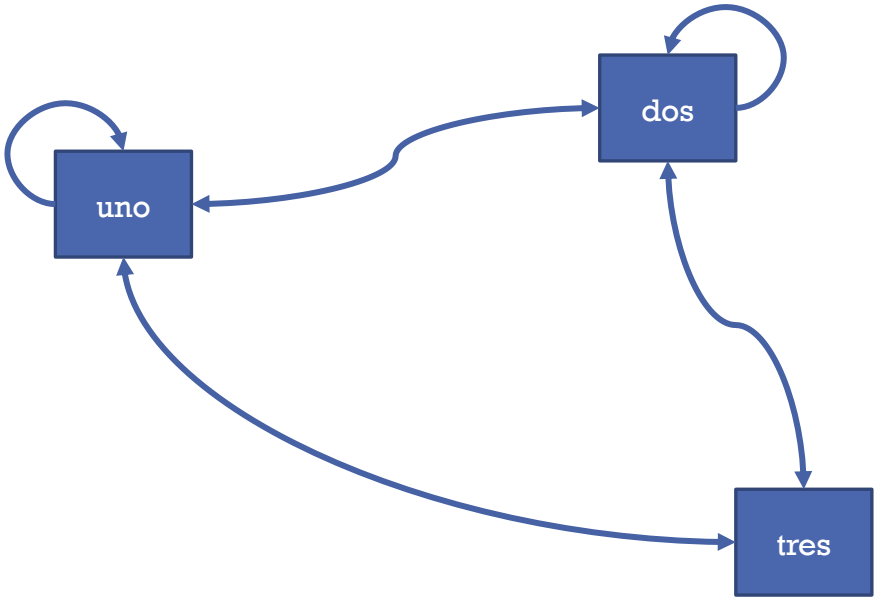
Sea A un conjunto de elementos $\{n_1, n_2, n_3, n_4, \ldots\}$, el producto cartesiano AxA está dado por todas las posibles combinaciones de los elementos del conjunto, es decir:

$$\langle n_1, n_1\rangle \langle n_1, n_2\rangle \langle n_1, n_3\rangle \ldots$$
$$\langle n_2, n_1\rangle \langle n_2, n_2\rangle \langle n_2, n_3\rangle \ldots$$
$$\langle n_3, n_1\rangle \langle n_3, n_2\rangle \langle n_3, n_3\rangle \ldots$$

Entonces, $G \subseteq AxA$.

Ejemplo

Sea V = {uno, dos, tres} y A = {<uno, uno> <uno, dos> <uno, tres> <dos, uno> <dos, dos> <dos, tres> <tres, uno> <tres, dos>}, la gráfica G = {V, A} es:

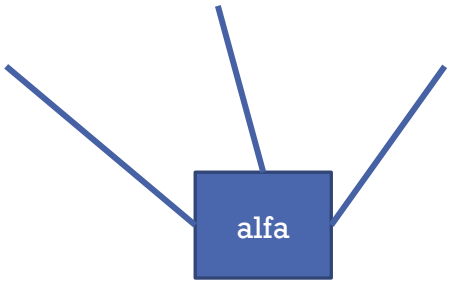


Grafos

Conceptos básicos

Grado de un vértice

El grado de un vértice v se denota como grado(v) y está definido por el número de aristas que contienen a v. Si grado(v)=0, entonces v no tiene aristas y se dice que es un nodo aislado.

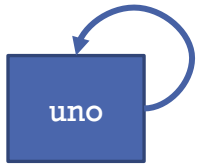


grado (alfa) = 3

grado (cuatro) = 0

Lazo

Un lazo es una arista que conecta a un vértice consigo mismo, es decir, a = {u, u}.

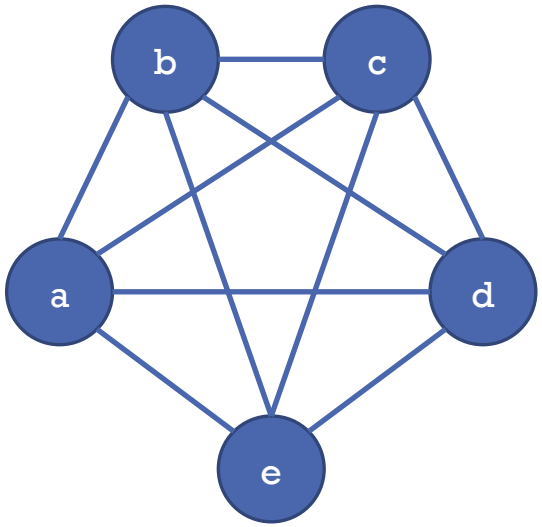


Camino

Un camino P de longitud ***n*** se define como la secuencia de ***n*** vértices que se deben recorrer para llegar del vértice v_1 (origen) al vértice v_n (destino):

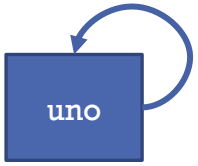
$P = (V_1, \ldots, V_n)$

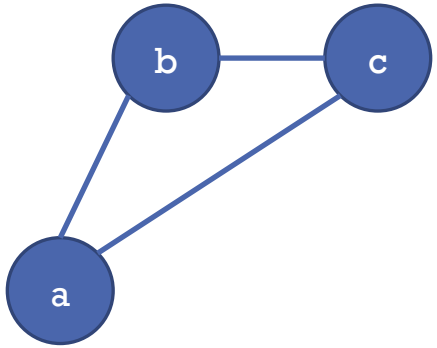
¿P(a, d)?
P (a, d) = {a, b, c, d}
P (a, d) = {a, e, d}
P (a, d) = {a, d}



Camino cerrado

Un camino P es cerrado si el primero y el último vértice son iguales, es decir, si $v_1 = v_n$.

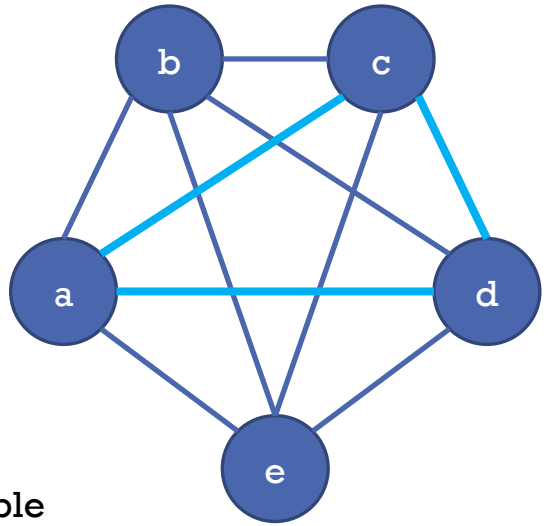




A-B-C-A es un camino cerrado
A-B-C no es un camino cerrado

Camino simple

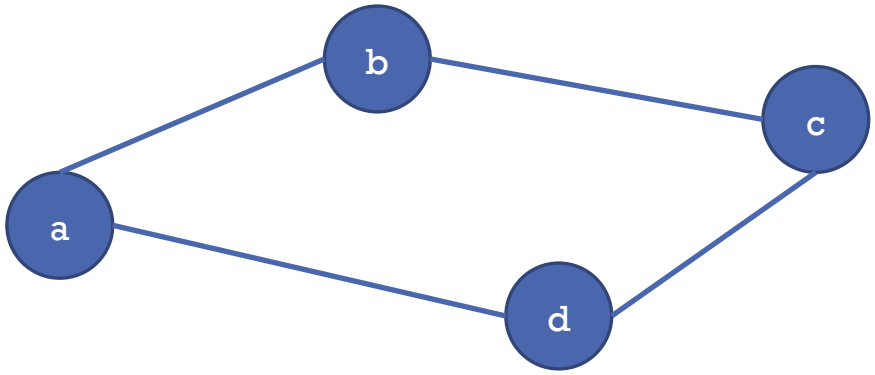
Es una ruta P en la que todos sus nodos son distintos. El primero y el último nodo pueden o no ser iguales.



A-C-D-A es un camino simple
A-C-B-D-C no es un camino simple

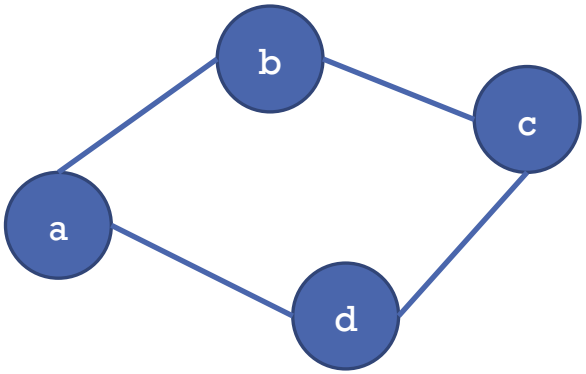
Ciclo

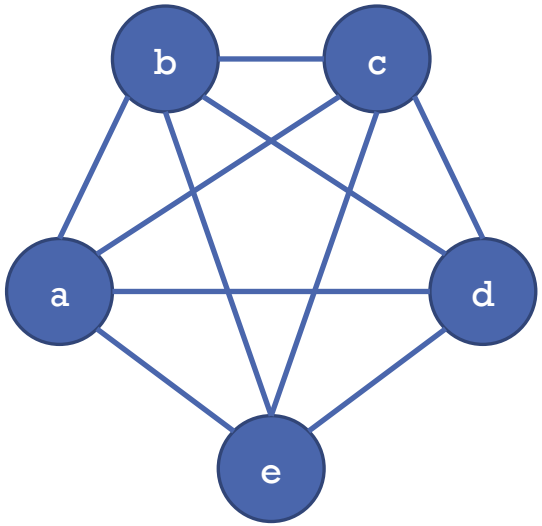
Un ciclo es un camino simple cerrado de longitud mayor o 3.



Gráfica conexa

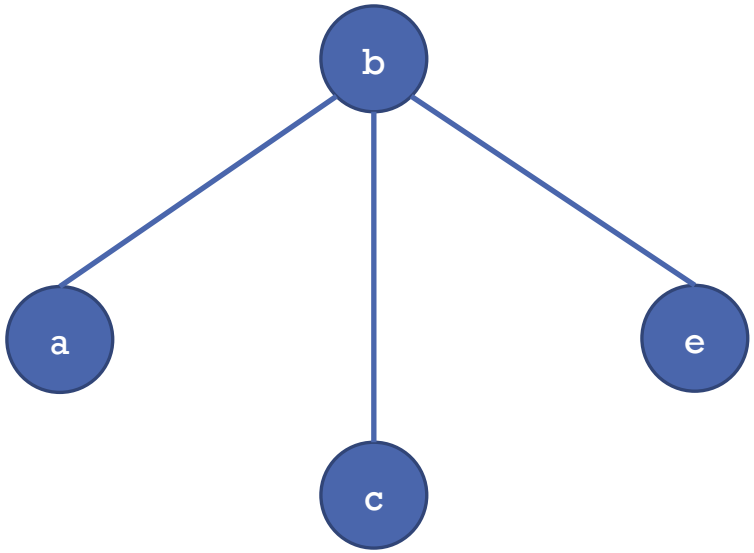
Una gráfica es conexa si existe un camino simple entre cualesquiera dos de sus nodos.





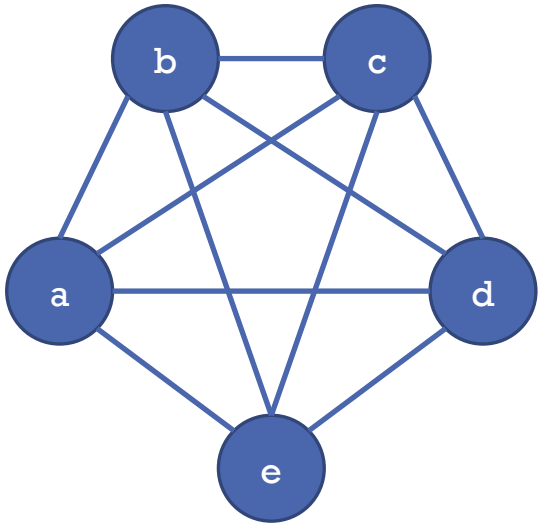
Gráfica árbol

Una gráfica es de tipo árbol (o árbol libre) si es una gráfica conexa sin ciclos.



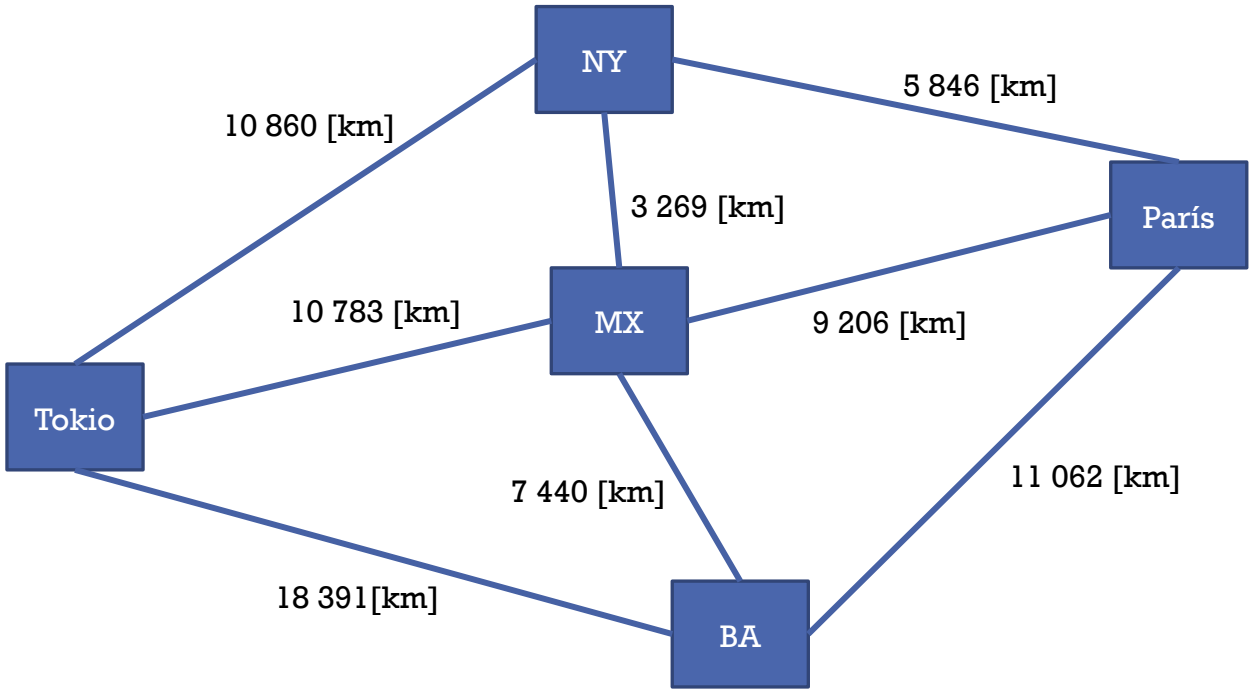
Gráfica completa

Una gráfica es completa si cada vértice ***v*** es adyacente a todos los demás vértices de G.



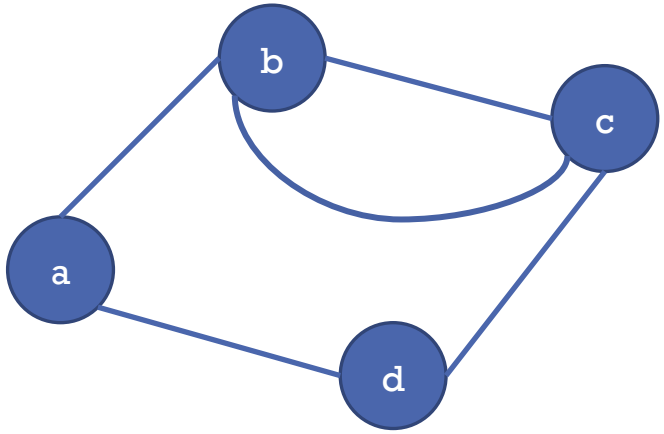
Gráfica etiquetada

Una gráfica G está etiquetada si sus aristas a tienen asignado un valor numérico no negativo c(a), llamado costo, peso o longitud.



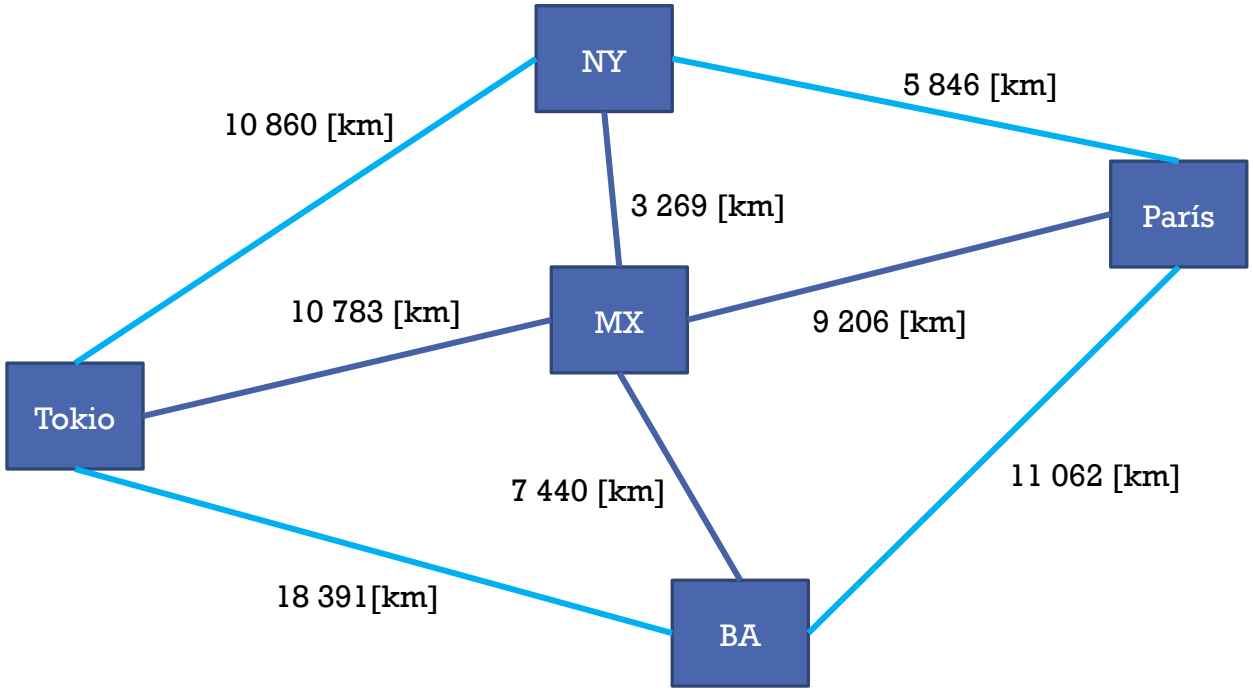
Multigráfica

Una gráfica G se considera multigráfica cuando al menos dos de sus vértices están conectados entre sí por medio de dos aristas (aristas múltiples o paralelas).



Subgráfica

A partir de una gráfica G = {V, A}, G' = {V', A'} es una sub gráfica de G si V' ≠ φ, V' es un subconjunto de V y A' es un subconjunto de A, donde cada arista A' es incidente con los vértices de V'.

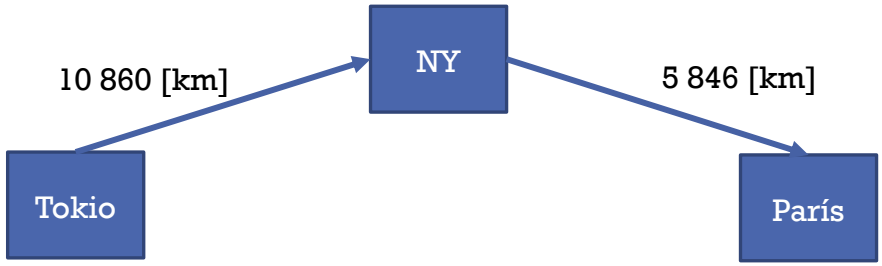


Gráfica dirigida

Una gráfica dirigida se caracteriza porque todas sus aristas tienen asociada una dirección, es decir, está constituida por pares ordenados.

En una gráfica dirigida los nodos representan información y las aristas representan una relación con dirección (o jerarquía) entre los vértices.

Una gráfica dirigida (o digráfica) G se caracteriza porque cada arista a tiene una dirección asignada y, por tanto, cada arista está asociada a un par ordenado (u, v) de vértices.



Una arista dirigida a = (u, v) se llama arco y se expresa como:

$$u \rightarrow v$$

De la expresión anterior se puede afirmar que:

- a empieza en u y termina en v.
- u es el origen (punto inicial) y v es el destino (o punto final) de a.
- u es predecesor de v y v es sucesor de u.

Dentro de las gráficas dirigidas se tienen dos tipos de grados, los internos y los externos.

El grado interno de un nodo está formado por el número de arcos que llegan a él. El grado externo de un nodo está dado por el número de líneas que emergen de él.

Las implementaciones más utilizadas para crear una gráfica en la computadora son:

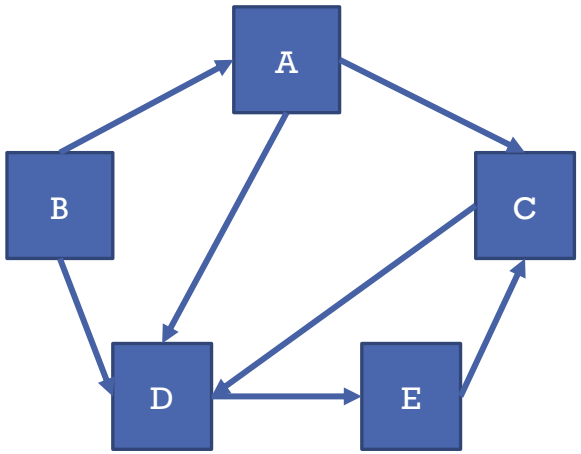
- Matriz de adyacencia
- Lista de adyacencia.

Matriz de adyacencia

Una matriz de adyacencia es una matriz booleana de orden ***n***, donde ***n*** indica el número de vértices de G. Tanto los renglones como las columnas de la matriz representan los nodos de la gráfica y su contenido representa la existencia (1) o no (0) de arcos entre los nodos i y j.

Sea G = (V, A) con V = {1, 2, 3, ..., n}, la matriz de adyacencia M que representa a G tiene n x n elementos, donde M[i, j] ($1 \leq j \leq n$ y $1 \leq j \leq n$) es 1 solo si existe un arco que vaya del nodo i al j, y 0 en caso contrario.

Dada la gráfica G, obtener su matriz de adyacencia.



Gráfica G (v, a)

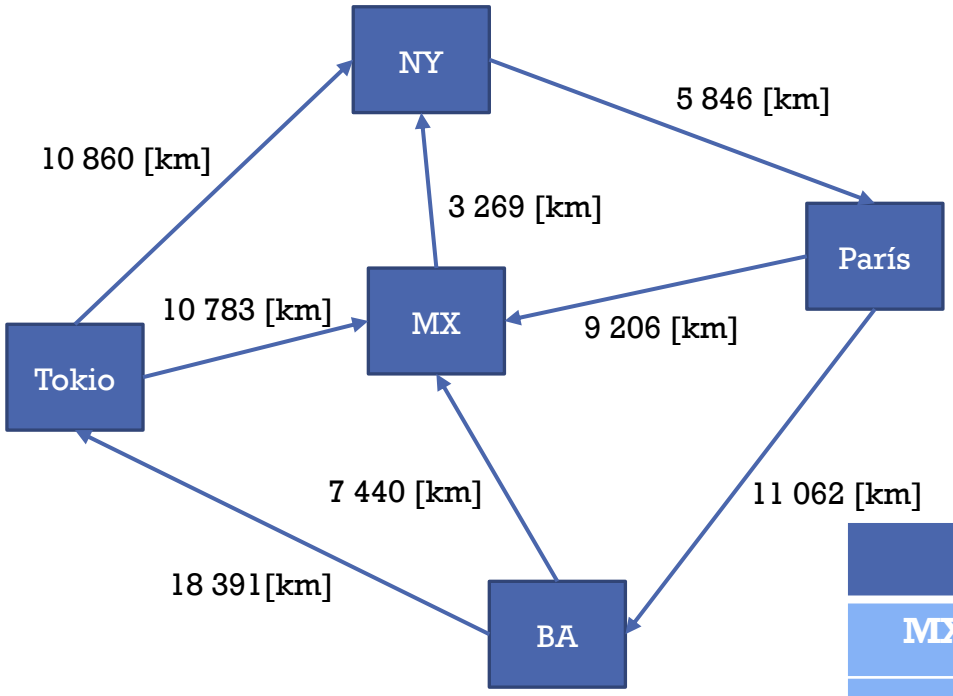
	A	B	C	D	E
A	0	0	1	1	0
B	1	0	0	1	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	1	0	0

Matriz de adyacencia

Dentro de una matriz de adyacencia el tiempo de acceso a un elemento es independiente del tamaño de V y A. Por otro lado, el tiempo de búsqueda es del orden O(n).

Una matriz de este tipo requiere un espacio n^2 en memoria aunque el número de arcos de G no llegue a ese número.

Dada la gráfica G, obtener su matriz de adyacencia.



Gráfica G (v, a)

	MX	París	Tokio	BA	NY
MX	0	0	0	0	3269
París	9206	0	0	11062	0
Tokio	10783	0	0	0	10860
BA	7440	0	18391	0	0
NY	0	5846	0	0	0

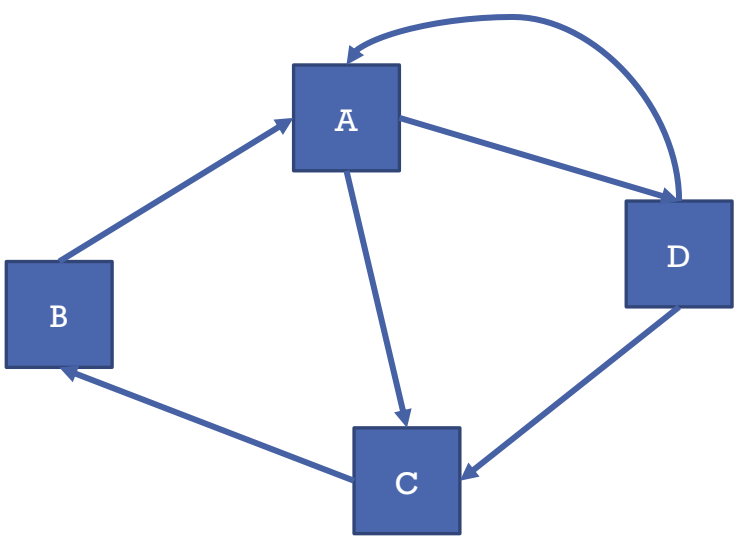
Lista de adyacencia

Una lista adyacente de una gráfica dirigida está formada por tantas listas como nodos tenga G y cada lista representa a todos los vértices adyacentes del vértice v.

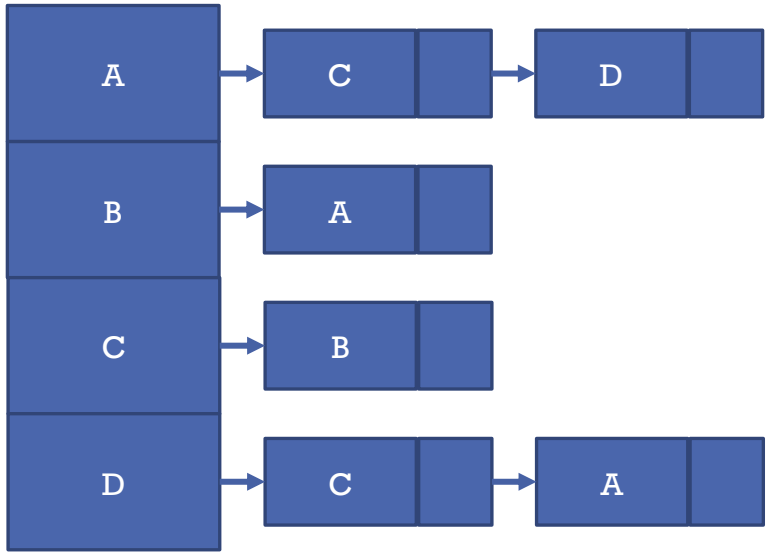
Esta disposición es recomendable cuando el número de aristas es menor a n^2. Sin embargo, el acceso a las aristas de cada nodo puede llegar a ser lento.

Ejemplo

Dada la gráfica G, obtener su lista de adyacencia.



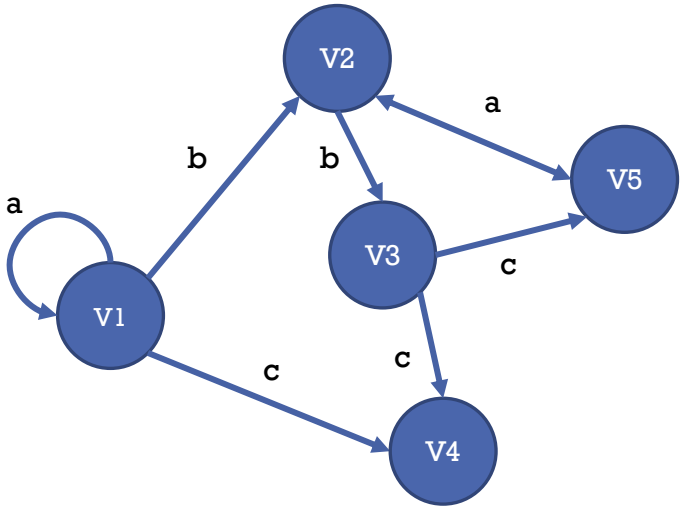
Gráfica G (v, a)



Lista de adyacencia

Ejemplo

Dada G = {V, A}, determinar el número de vértices internos y externos de G.

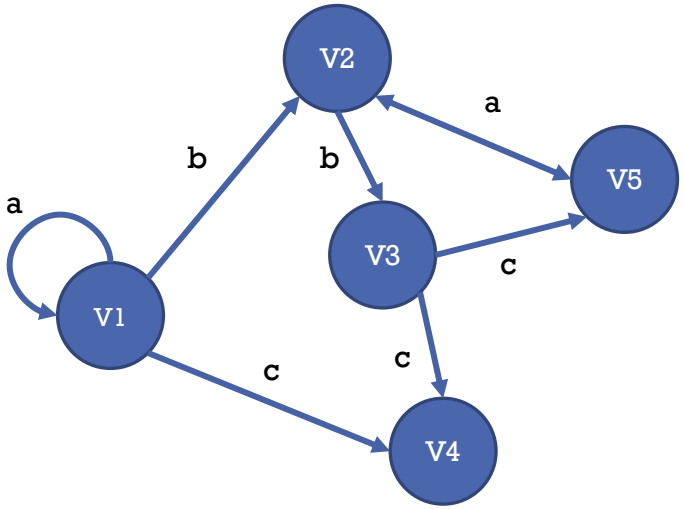


	V1	V2	V3	V4	V5
Externos	3	2	2	0	1
Internos	1	2	1	2	2

Ejemplo

Dada G = {V, A}, la matriz de adyacencia está definida por:

$$M[u, v] = \begin{cases} 1 \text{ si existe un arco } (u, v) \\ 0 \text{ en caso contrario} \end{cases}$$

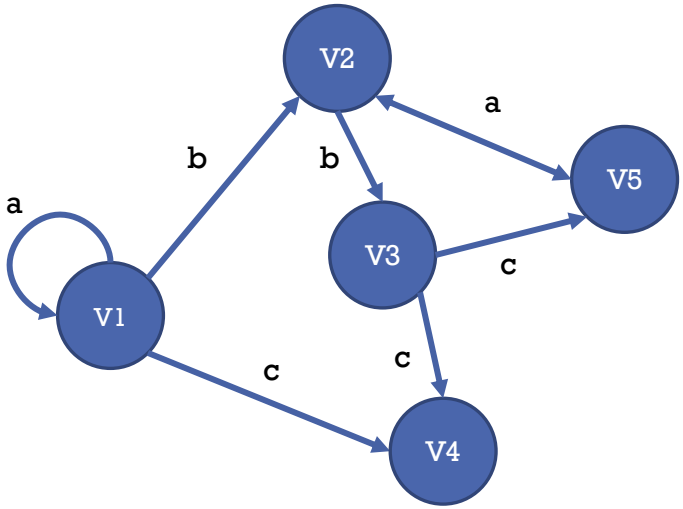


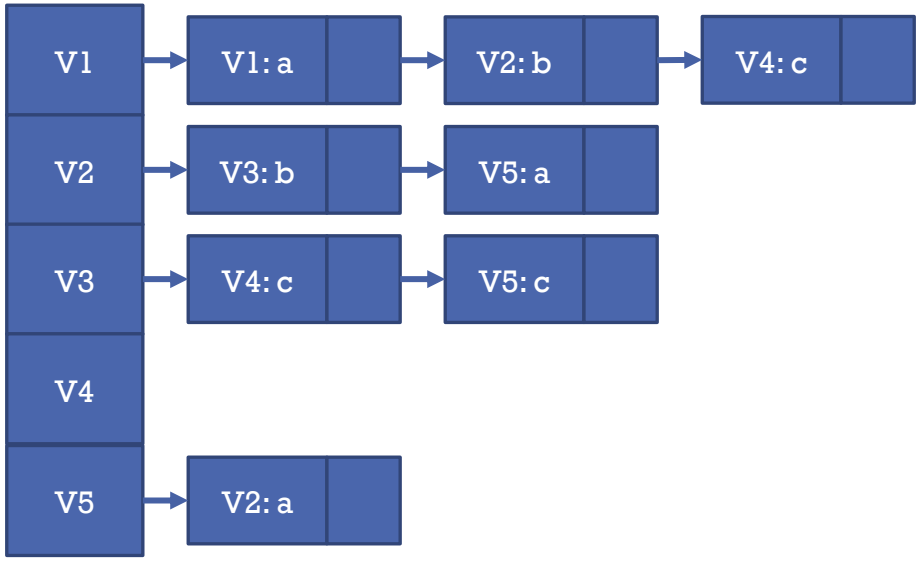
	V1	V2	V3	V4	V5
V1	A	B	0	C	0
V2	0	0	B	0	A
V3	0	0	0	C	C
V4	0	0	0	0	0
V5	0	A	0	0	0

Ejemplo

Dada G = {V, A}, la lista de adyacencia está definida por:

$$L_v = \{ v \in V \mid (u, v) \in A \}$$





Gráfica no dirigida

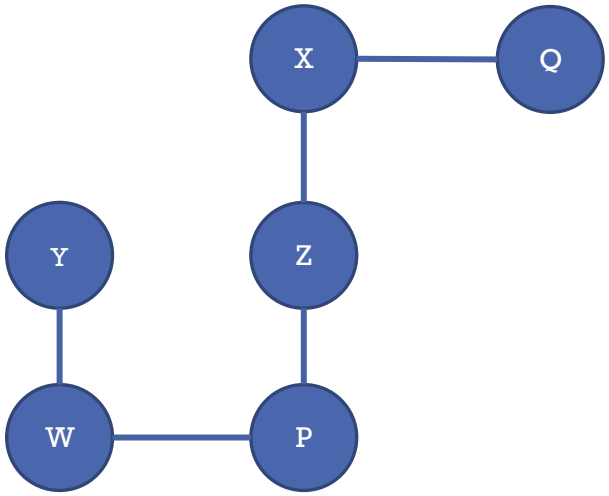
Una gráfica no dirigida G = (V, A) está formado por dos conjuntos finitos, uno de vértices V y otro de aristas A, donde cada arista en A es un par no ordenado de vértices, es decir, si (u, v) es una arista no dirigida, entonces (u, v) = (v, u).

Una gráfica no dirigida se puede representar mediante una matriz de adyacencia o mediante una lista de adyacencia (de manera similar a las gráficas dirigidas).

Debido a la relación entre los nodos, cada arista no dirigida entre u y v se debe cambiar por dos aristas dirigidas (de u→v y de v→u).

En este tipo de gráficas la matriz de adyacencia será simétrica y en la lista de adyacencia el vértice u estará en la lista de adyacencia del vértice v y viceversa.

Dada la gráfica G obtener su matriz de adyacencia.



Gráfica G (v, a)

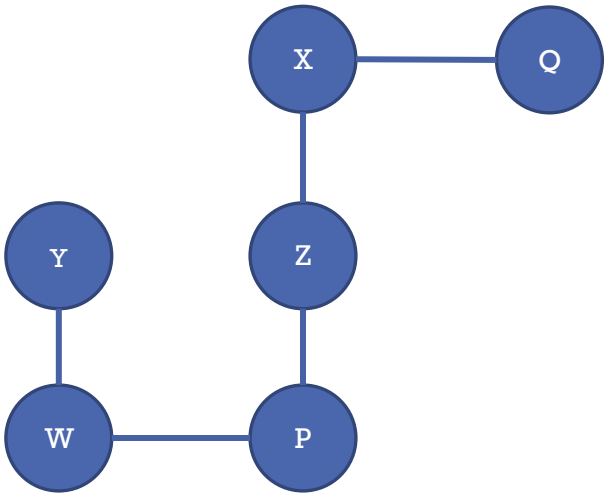
	P	Q	W	X	Y	Z
P	0	0	1	0	0	1
Q	0	0	0	1	0	0
W	1	0	0	0	1	0
X	0	1	0	0	0	1
Y	0	0	1	0	0	0
Z	1	0	0	1	0	0

Matriz de adyacencia

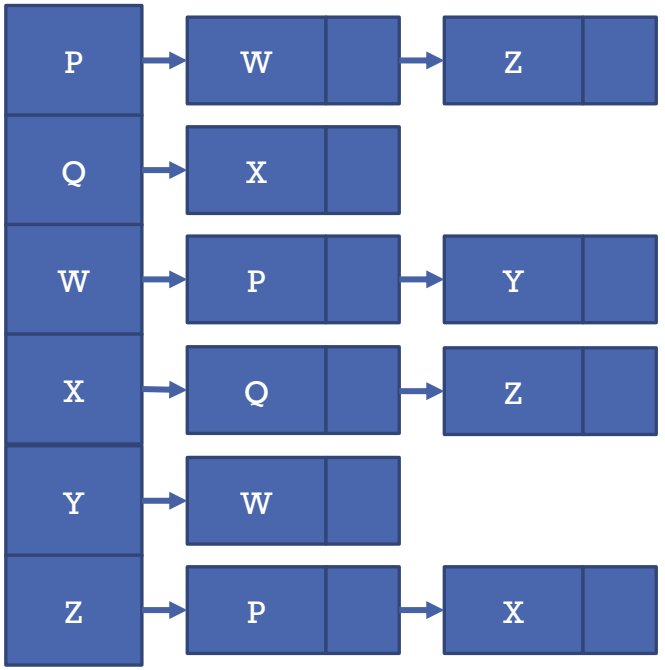
Nótese que la gráfica es simétrica, el triángulo superior es igual al triángulo inferior.

Ejemplo

Dada la gráfica G obtener su lista de adyacencia.



Gráfica G (v, a)



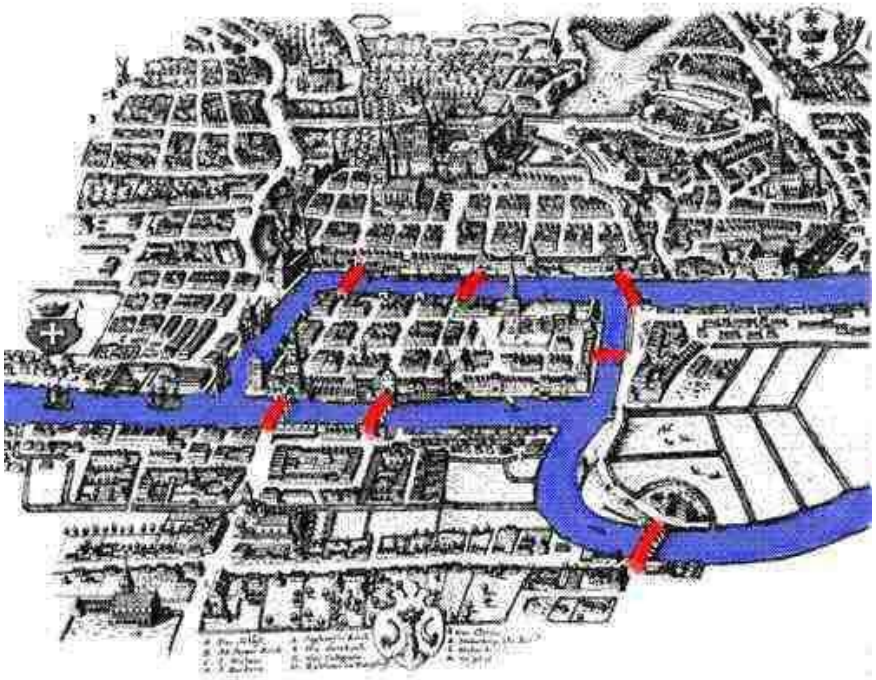
Matriz de adyacencia

La teoría de gráficas o teoría de grafos es aplicada en una gran cantidad de áreas tales como ciencias sociales, lingüística, ciencias físicas, ingeniería de comunicación y otras.

En la ciencia de la computación se utilizan para modelar, entre otras cosas, la teoría de cambio y lógica de diseño, inteligencia artificial, lenguajes formales, gráficos por computadora, sistemas operativos, compiladores y organización y recuperación de información.

Una red social se puede modelar como un grafo si se considera una gráfica donde los vértices son personas y existe una arista entre dos personas si y solo si son amigos.

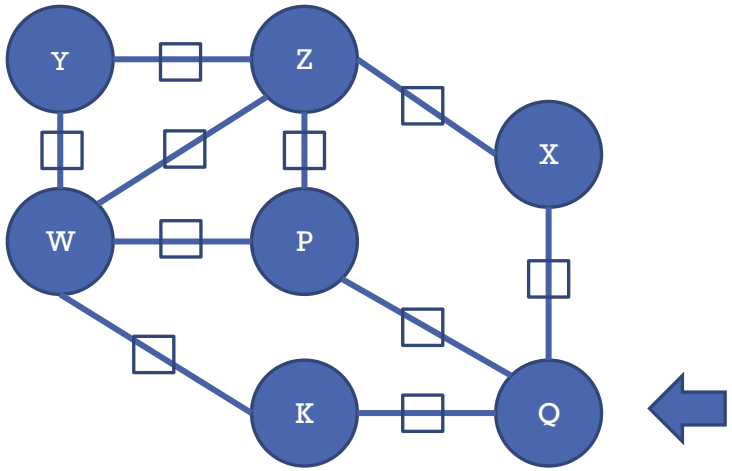
En los últimos años ha surgido una ciencia completa que se encarga del estudio de las redes sociales debido a que las personas y sus comportamientos se entienden mejor como propiedades de un grafo.



En un grafo que modela una red social se pueden realizar diversas preguntas:

¿Si soy "amigo" de X significa que X es "amigo" mío?
¿Qué tan cercano es un "amigo"?
¿Soy amigo mío?
¿Quién tiene más amigos?
¿Mis amigos viven cerca de mi?
¿Eres un individuo o una multitud sin rostro?

Dado el siguiente grafo, el cual representa una red social (los nodos son actores y las aristas películas), se desea visitar (ver) todas las películas (aristas) sin repetir, es decir, solo se puede pasar una vez por cada arista, y se debe partir del artista Q.



Dentro de las ciencias de la computación pueden existir diversos tipos de grafos:

- Anillo
- Árbol
- Cadena
- Estrella
- Hipercubo

Dependiendo el tipo de implementación que se desee una gráfica puede ser más eficiente cuando se maneja mediante una matriz de adyacencia o mediante una lista de adyacencia.

¿Más rápido para probar si los nodos (x, y) están en la gráfica?
Matriz de adyacencia

¿Más rápido para encontrar el grado externo de un nodo?
Lista de adyacencia

¿Ocupa menos memoria en gráficas pequeñas?
Lista de adyacencia ($m+n$ vs n^2)

Dependiendo el tipo de implementación que se desee una gráfica puede ser más eficiente cuando se maneja mediante una matriz de adyacencia o mediante una lista de adyacencia.

¿Ocupa menos memoria en gráficas grandes?
Matriz de adyacencia

¿Facilidad para agregar o eliminar vértices?
Matriz de adyacencia (O(1) vs O(d))

¿Facilidad para recorrer la gráfica?
Lista de adyacencia ($\theta(m + n)$ vs $\theta(n^2)$)

“You might not think that programmers are artists, but programming is an extremely creative profession. It’s logic-based creativity.”

John Romero
(Is an American director, designer, programmer,
and developer in the video game industry.)

Implementación

Realizar un programa en Python que permita guardar las relaciones de una gráfica. La gráfica puede ser dirigida o no dirigida. Así mismo, puede ser etiquetada o no etiquetada.

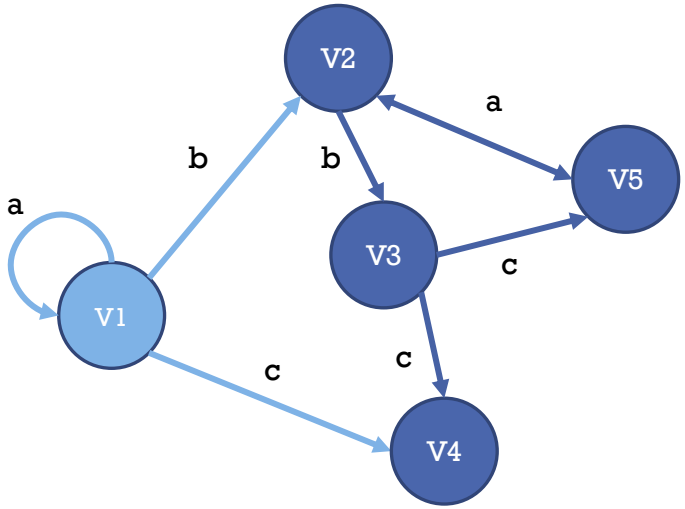
La implementación del grafo se debe realizar utilizando tanto matriz de adyacencia como lista adyacente.

En lenguaje C, ¿cómo se podría definir una estructura?

En un lenguaje orientado a objetos, ¿cómo se podría definir una clase?

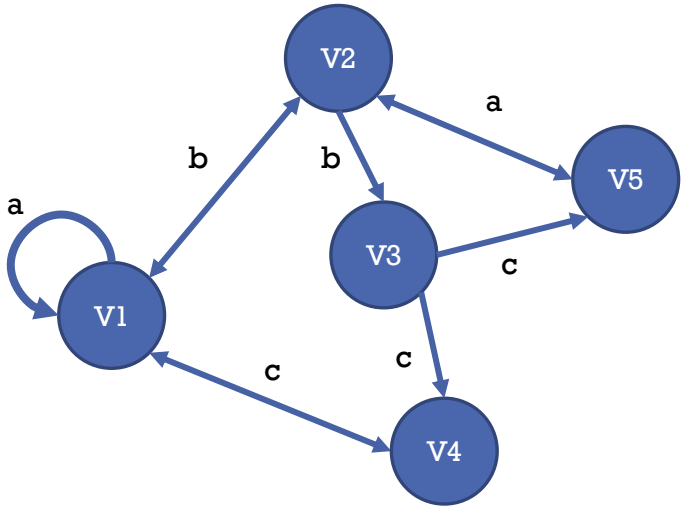
¿Qué se requiere para modelar una gráfica con una lista de adyacencia?

Un grafo está constituido por nodos, por lo tanto, lo primero que hay que modelar es el nodo.




```
class Node:
    # edge to v
    to = 0
    # weight or cost
    cost = 0
    # Next edge-node
    nxt = None
```

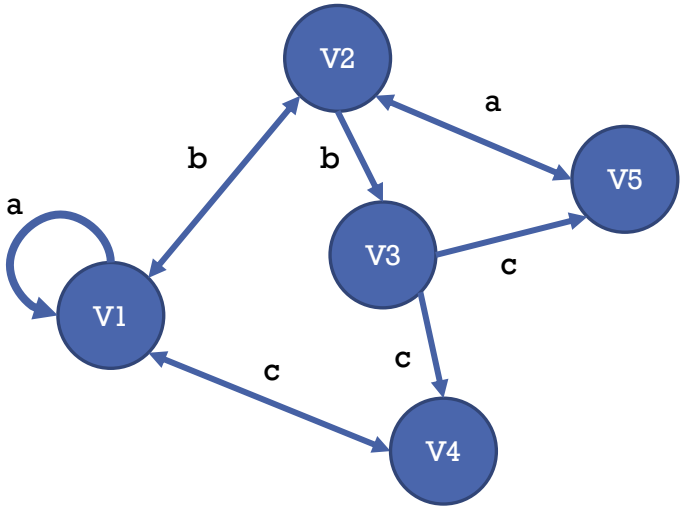
Debido a que una gráfica es un conjunto de nodos unidos entre sí, una vez modelado el nodo, se puede construir una gráfica.




```
class Graph:
    # edges adjacent
    edges = []
    # extern grade
    grade = []
    # nodes number
    numNodes = 0
    # edges number
    numEdges = 0
    # directed or not
    directed = False
```

Dada la definición de la clase graph cada nodo de la gráfica corresponderá a una posición de la lista de nodos, por lo tanto, es importante inicializar los grados y nodos de la gráfica.

```
def start_graph (objGraph):
    i = 0
    while i<=objGraph.numNodes:
        objGraph.grade.append(0)
        objGraph.edges.append(None)
        i += 1
```



¿Cómo se inserta un nodo en la gráfica?

```
def insert_edge(objGraph, intU, intV, intCost, isDirected):
    item = node()
    item.cost = intCost
    item.to = intV;
    item.nxt = objGraph.edges[intU]
    objGraph.edges[intU] = item
    objGraph.grade[intU] += 1
    if isDirected == False and intV != intU:
        insert_edge(objGraph, intV, intU, cost, True)
```

¿Qué datos se requieren para crear la gráfica?

```
def create_graph (objGraph, hasCost):
    i = 0
    while i <= objGraph.numEdges:
        u = int(input('u: '))
        v = int(input('v: '))
        if hasCost == True :
            cost = input('Cost / weight: ')
        else:
            cost = 1
        insert_edge(objGraph, u, v, cost, objGraph.directed);
        i += 1
```

¿Cómo se imprimen los valores de la gráfica?

```
def print_graph(objGraph):
    i = 1
    item = None
    string = ""
    while i <= objGraph.numNodes:
        string += str(i) + "\t"
        item = objGraph.edges[i]
        while item != None:
            string += str(item.to) + ": " + str(item.cost) +"\t"
            item = item.nxt;
        string += "\n"
        i += 1
    print(string)
```

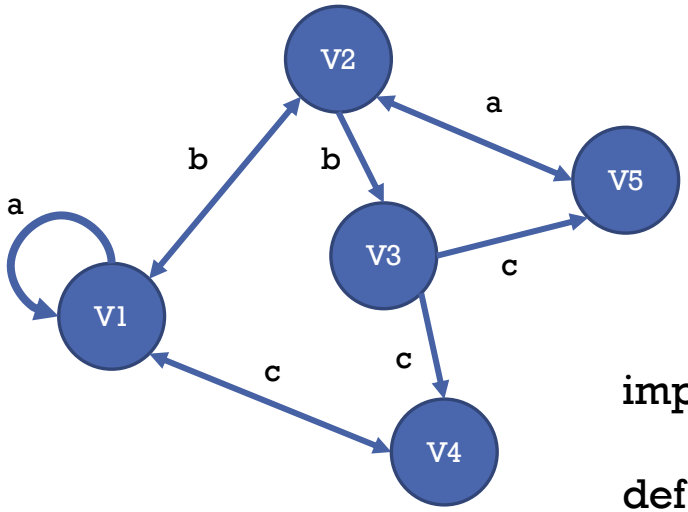
¿Qué datos se requieren para iniciar la gráfica?

```
def main():
    directed = int(input("¿Es dirigida? 1)SÍ, 2)NO: "))
    cost = int(input("¿Peso en las aristas? 1)SÍ, 2)NO: "))
    objGraph.numNodes = int(input("Número de nodos: "))
    objGraph.directed = True if directed == 1 else False
    objGraph.numEdges = int(input("Número de aristas: "))

    start_graph(objGraph)
    create_graph(objGraph, cost)
    print_graph(objGraph)

obj_graph = Graph()
main()
```

Para representar un grafo en una matriz de adyacencia se debe conocer el número de nodos, para reservar el espacio necesario.




```
import numpy as np

def main():
    num_nodes = int(input())
    hasCost = int(input())
    matrix = np.zeros((num_nodes, num_nodes))
```

¿Qué datos se requieren para insertar los datos en la gráfica?

```
def create_graph (matrix, hasCost, num_edges):
    i = 0
    while i < num_edges :
        u = int(input('u: '))
        v = int(input('v: '))
        if hasCost == True :
            cost = int(input('Cost / weight: '))
        else:
            cost = 1
        matrix[(u-1)][(v-1)] = cost;
        if objGraph.directed == False:
            matrix[(v-1)][u-1] = cost;
        i += 1
```

¿Cómo se imprimen los valores de la gráfica?

```
def print_graph(intMatrix):
    print (intMatrix)
```

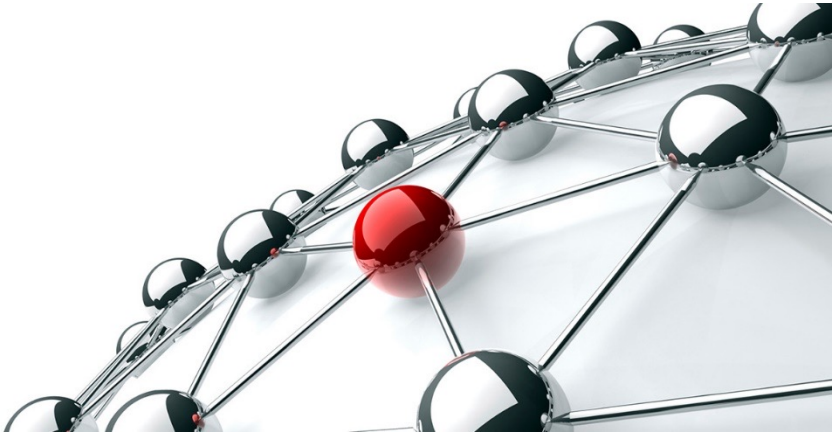
Algoritmos de exploración de grafos

La exploración de grafos consiste en realizar un recorrido de forma eficiente por cada nodo de la gráfica a través de las aristas disponibles. Los dos algoritmos más conocidos para la exploración de grafos son:

- Búsqueda por expansión.
- Búsqueda por profundidad.

3.2 Búsqueda por expansión.

Algoritmos de grafos



La búsqueda por expansión o breadth-first search es uno de los algoritmos más simples para realizar búsquedas en un grafo.

Dada una gráfica G = {V, A} y un vértice fuente s, el algoritmo de búsqueda por expansión (amplitud o anchura) explora de manera sistemática todas las aristas de G para descubrir cada vértice que es accesible desde s.

El algoritmo breadth-first search genera un breadth-first tree, cuya raíz es s y contiene todos los nodos accesibles desde éste. El algoritmo trabaja sobre gráficas dirigidas y no dirigidas.

Para mantener una traza del progreso el algoritmo pinta los vértices con colores blanco, gris o negro. Todos los vértices inician en color blanco y durante la búsqueda, según van siendo descubiertos, cambian de color a gris o negro.

Si un nodo es gris o negro significa que éste ya ha sido descubierto. Si (u, v) $\in$ A y el vértice u es negro, entonces, sin importar si v es gris o negro, significa que todos los vértices adyacentes a u han sido descubiertos.

Por otro lado, los vértices grises pueden tener nodos adyacentes blancos y, por tanto, representan la frontera entre los vértices descubiertos y los no descubiertos.

El algoritmo breadth-first search al mismo tiempo que va descubriendo los nodos va creando un árbol (breadth-first tree). Al inicio del proceso el árbol contiene solo el nodo raíz, el cual es el vértice origen s.

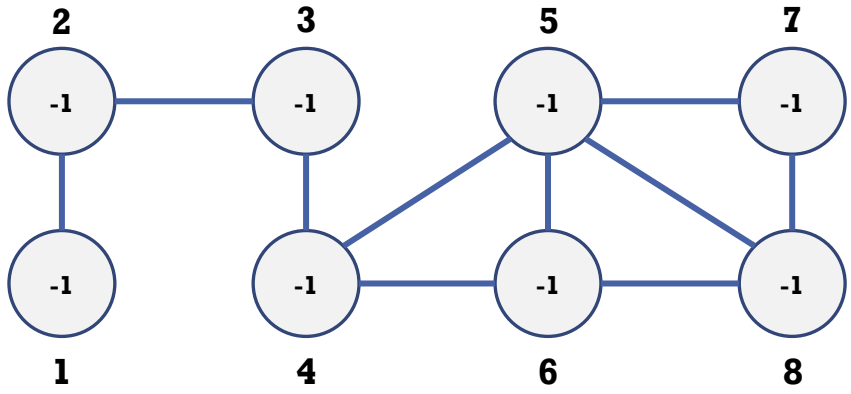
El proceso de búsqueda parte de un nodo u (origen) a un nodo v (destino). Si se encuentra un nodo blanco v dentro de la lista de adyacencia de u el vértice v y la arista (u, v) se ingresan en el árbol, de tal manera que u es el predecesor de v en el árbol.

Durante el proceso de búsqueda se cuenta con una estructura de datos lineal tipo cola, la cual permite guardar todos los nodos adyacentes a u, los cuales representan las rutas que se pueden seguir a partir del nodo origen.

En la cola se agregan los nodos de color blanco, es decir, los nodos que no han sido visitados y se recorren los nodos adyacentes para dibujar las rutas posibles. Así, se guarda la distancia que existe desde el nodo fuente a los nodos descubiertos.

Ejemplo

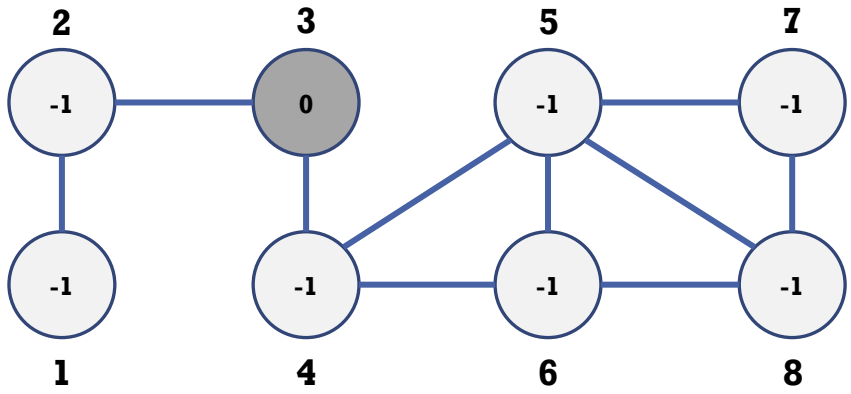
Dado la siguiente gráfica G realizar una búsqueda por expansión dentro de la misma a partir del nodo 3.



Ejemplo

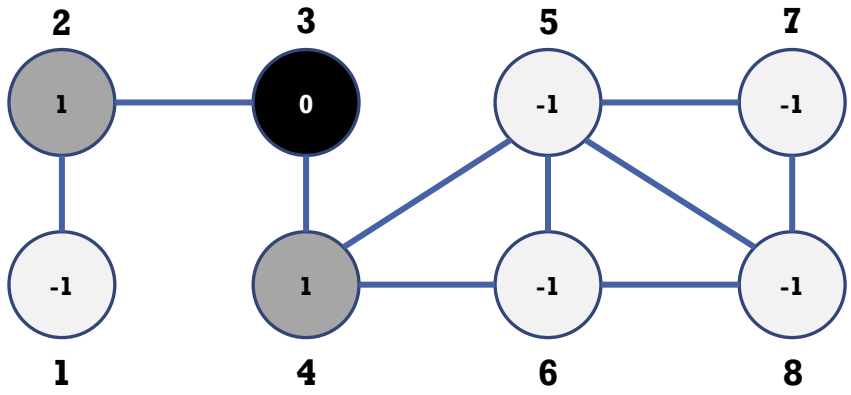
Se inicia en el nodo 3, por lo tanto, el único valor de la queue es 3. El valor del nodo fuente es 0, su color es gris y no tiene predecesor (nodo raíz).

queue 3



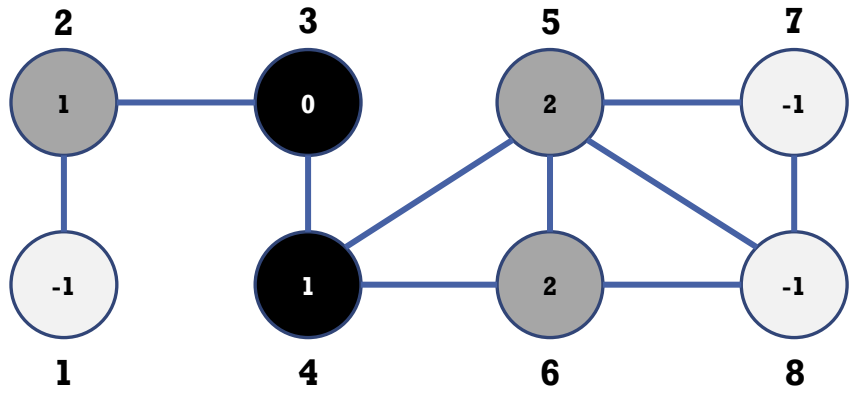
Se extrae el nodo 3 de la cola y se buscan los vértices contiguos al nodo fuente, los cuales se cambian de color a gris y se ingresan en la cola. La distancia del nodo fuente a estos nodos es 1. El nodo extraído cambia de color a negro.

queue | 4 | 2 |



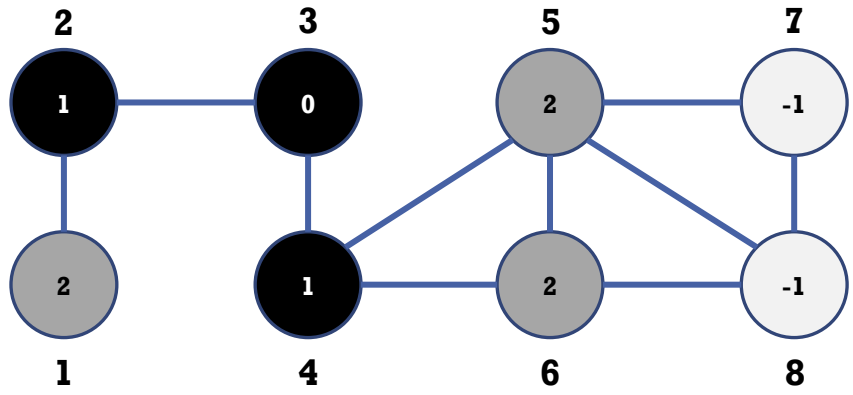
Se extrae el nodo 4 de la cola y se buscan los vértices contiguos al nodo fuente, los cuales se cambian de color a gris y se ingresan en la cola. La distancia del nodo fuente a estos nodos es 2. El nodo extraído cambia de color a negro.

queue



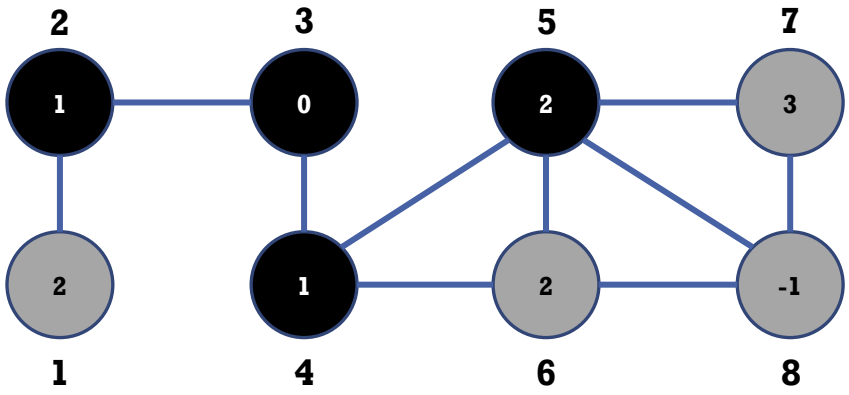
Se extrae el nodo 2 de la cola y se buscan los vértices contiguos al nodo fuente, los cuales se cambian de color a gris y se ingresan en la cola. La distancia del nodo fuente a estos nodos es 2. El nodo extraído cambia de color a negro.

queue



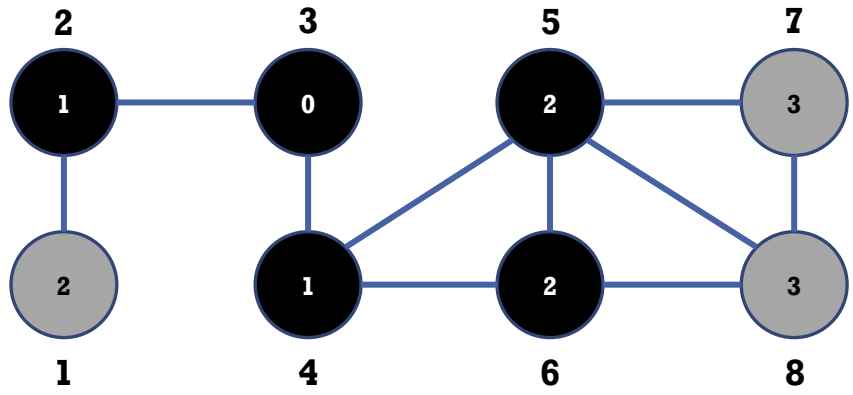
Se extrae el nodo 5 de la cola y se buscan los vértices contiguos al nodo fuente, los cuales se cambian de color a gris y se ingresan en la cola. La distancia del nodo fuente a estos nodos es 3. El nodo extraído cambia de color a negro.

queue



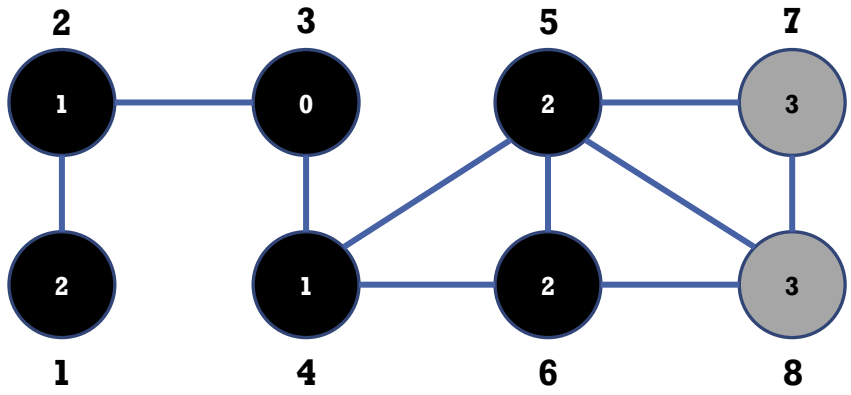
Se extrae el nodo 6 de la cola y se buscan los vértices contiguos al nodo fuente, debido a que estos ya habían sido descubierto, ya no se ingresan a la cola ni se modifica su distancia. El nodo extraído cambia de color a negro.

queue



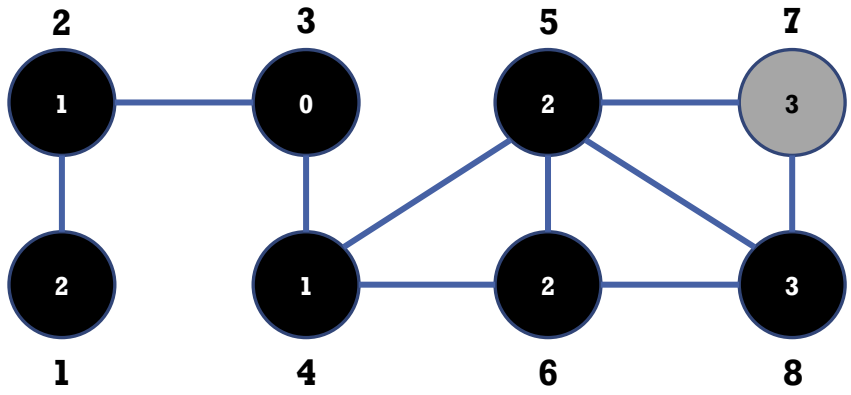
Se extrae el nodo 1 de la cola y se buscan los vértices contiguos al nodo fuente, debido a que estos ya habían sido descubierto, ya no se ingresan a la cola ni se modifica su distancia. El nodo extraído cambia de color a negro.

queue | 8 | 7 |



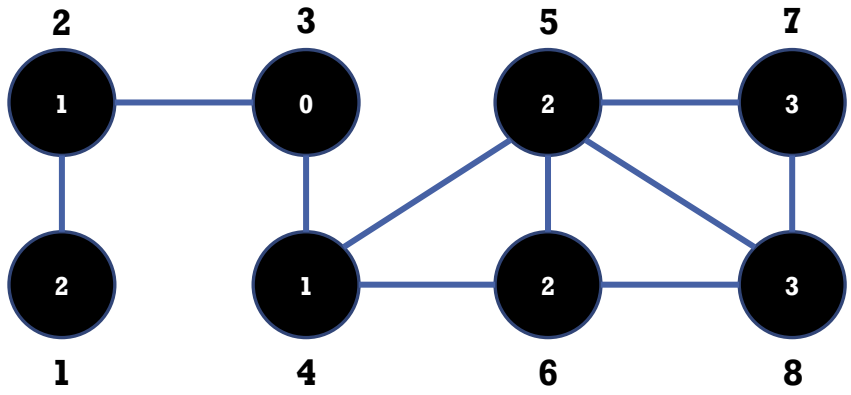
Se extrae el nodo 8 de la cola y se buscan los vértices contiguos al nodo fuente, debido a que estos ya habían sido descubierto, ya no se ingresan a la cola ni se modifica su distancia. El nodo extraído cambia de color a negro.

queue 7

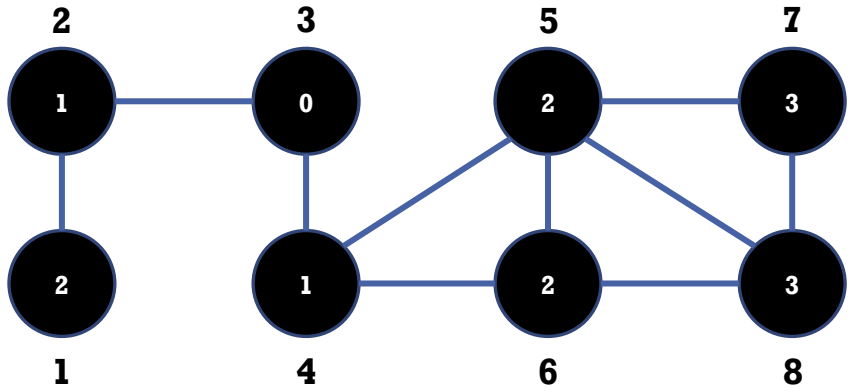


Se extrae el nodo 7 de la cola y se buscan los vértices contiguos al nodo fuente, debido a que estos ya habían sido descubierto, ya no se ingresan a la cola ni se modifica su distancia. El nodo extraído cambia de color a negro.

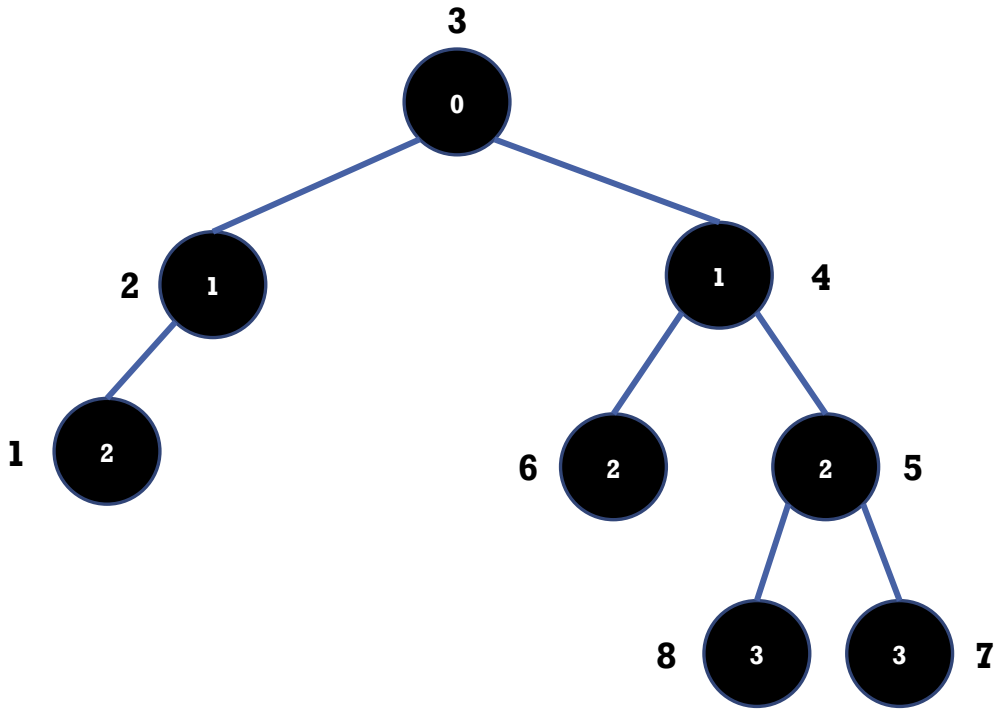
queue



El proceso termina cuando la cola ya no tiene elementos. La distancia desde el nodo fuente hasta los últimos nodos descubiertos crean el árbol breadth first.



El proceso termina cuando la cola ya no tiene elementos. La distancia desde el nodo fuente hasta los últimos nodos descubiertos crean el árbol breadth first.



"Code is like a humor. When you have to explain it, it's bad."

Cory House
(Author @pluralsight, Speaker, Software Architect,
@microsoft MVP, @JSJabber panelist #JavaScript)

El siguiente pseudocódigo representa el algoritmo básico de breadth first search (BFS):

```
breadth_first_search(G, s)
        for each vertex u ∈ V[G] − {s}
                color[u] ← WHITE
                distance[u]←∞
                 predecessor[u]← NIL
        color[s] ← GRAY
        distance[s] ← 0
        predecessor[s] ← NIL
        Q ← ∅
        ENQUEUE(Q, s)
        while Q ≠ ∅
                u ← DEQUEUE(Q)
                for each v ∈ Adj[u]
                        if color[v] = WHITE
                                color[v] ← GRAY
                                distance[v] ← distancia[u] + 1
                                 predecessor[v]← u
                                ENQUEUE(Q, v)
                color[u] ← BLACK
```

Análisis de complejidad

Dada una gráfica G = {V, A} sobre la que se realiza una búsqueda BFS, la operación de encolar y desencolar toma O(1). El tiempo que toma la operación de encolar está directamente relacionada con el número de vértices, es decir, O(V).

La longitud de toda la lista de adyacencia está dada por el número de aristas A de la gráfica, por tanto, el tiempo total que se requiere para leer toda la lista de adyacencia es O(A).

El tiempo total de ejecución del algoritmo breadth first search es O(V + A).

La ruta más corta

La ruta más corta se define como la distancia $\delta(s, v)$, es decir, el número de aristas que se deben recorrer para llegar del nodo s al nodo v. Si no existe una ruta se dice que $\delta(s, v) = \infty$.

Dada $G = \{V, A\}$ de una gráfica dirigida o no dirigida y dado un vértice arbitrario $s \in V$, para cualquier arista $(u, v) \in A$ se tiene:

$$\delta(s, v) \leq \delta(s, u) + 1$$

Algoritmos de grafos. Parte I.

Práctica 6

6. Algoritmos de grafos. Parte I.

- Implementar en lenguaje Python el algoritmo de búsqueda por expansión dentro de un grafo dirigido y no dirigido, con costo y sin costo, a partir de un nodo fuente s.
- Obtener la complejidad algorítmica del algoritmo búsqueda por expansión.
- Crear las gráficas de la complejidad que tiene el algoritmo de búsqueda por expansión dentro de un grafo para el mejor caso, el peor caso y el caso promedio.

3.3 Búsqueda por profundidad

Algoritmos de grafos

Un algoritmo de búsqueda por profundidad o depth-first search consiste (como su nombre lo indica) en buscar profundamente en el grafo como sea posible.

El algoritmo explora las aristas que no han sido exploradas del último vértice v descubierto. Cuando todas las aristas de v han sido exploradas la búsqueda vuelve hacia atrás (backtrack) para explorar las aristas no exploradas del vértice por el cual se descubrió v.

El proceso continúa hasta que se descubren todos los vértices alcanzables desde el nodo fuente. Si existe un vértice sin explorar, entonces éste se selecciona como el nuevo nodo fuente y el proceso se repite. El proceso entero se repite hasta que todos los vértices hayan sido descubiertos.

Cuando a partir del vértice u un vértice v ha sido descubierto, el algoritmo guarda a u como el predecesor de v. Las subgráficas de predecesores pueden dibujar varios árboles (bosque), debido a que pueden existir varios nodos fuente.

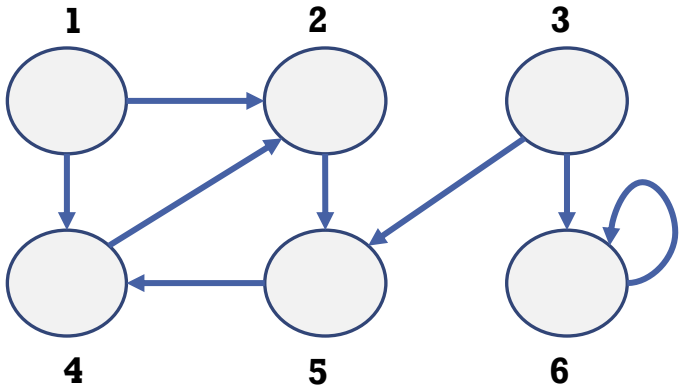
Durante la búsqueda los vértices se colorean para indicar su estado. Los vértices inicialmente son blancos, se colorean de gris cuando han sido descubiertos y se colorean de negro cuando se terminan, esto es, cuando toda la lista de adyacencia del vértice ha sido examinada por completo.

El algoritmo también genera marcas de tiempo en cada vértice. Cada vértice v tiene dos marcas de tiempo: la primera marca guarda cuando el vértice ha sido descubierto (cuando se colorea de gris), la segunda marca guarda cuando la búsqueda termina de examinar la lista de adyacencia de v (cuando se colorea de color negro).

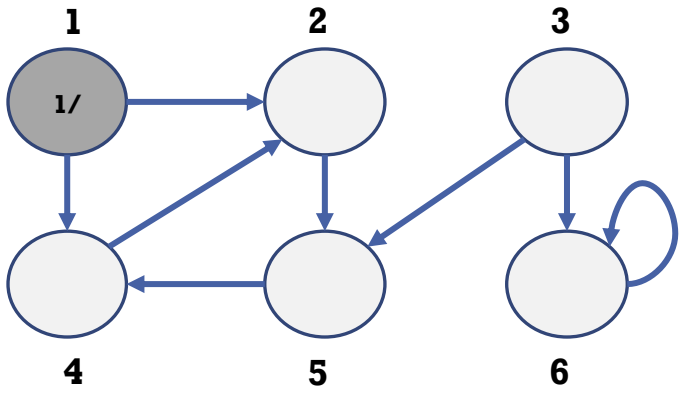
El algoritmo graba cuando un vértice u es descubierto (distance[u]) y cuando se termina con toda la lista de adyacencia de u (finish[u]). Las marcas de tiempo son valores enteros en el rango de 1 a 2|V|, cumpliendo que para cada vértice u la distance[u] < finish[u].

El vértice u es blanco antes de la marca distance[u], gris entre las marcas distance[u] y finish[u] y negro al final de la marca finish[u]. La marca de tiempo es común para todos los nodos.

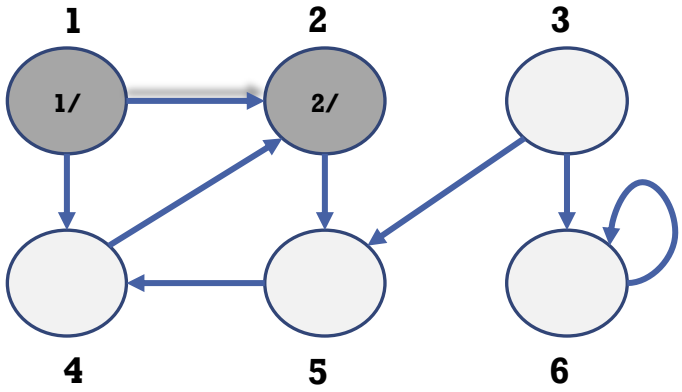
Dado la siguiente gráfica G realizar una búsqueda por profundidad dentro de la misma.



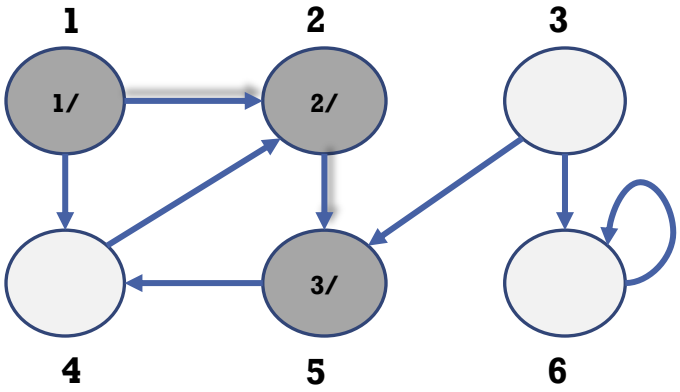
Se inicia en el primer nodo de la gráfica, siendo éste u, se colorea de gris y se inicia el proceso de visita por todos sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



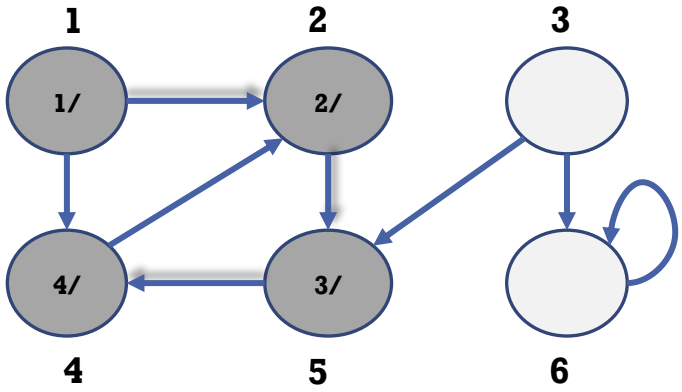
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



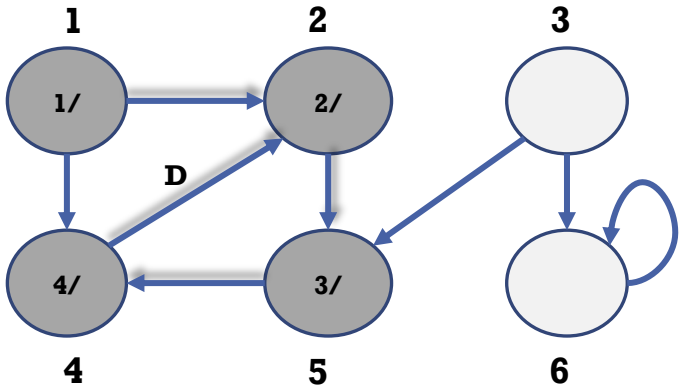
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



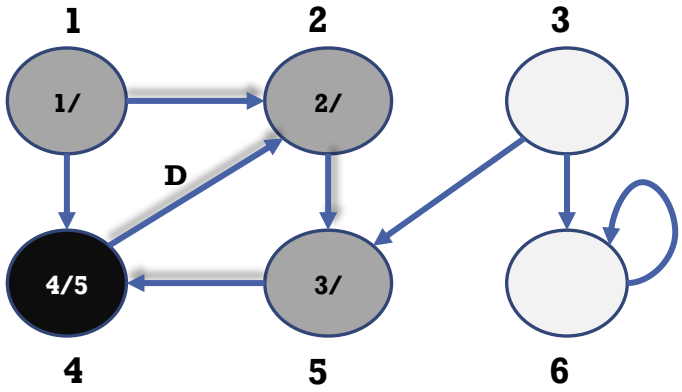
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



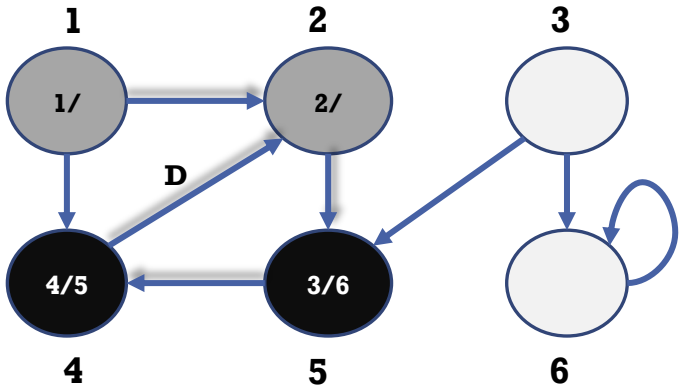
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



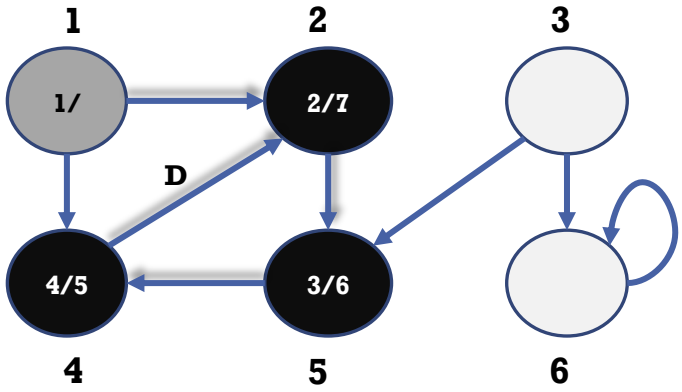
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



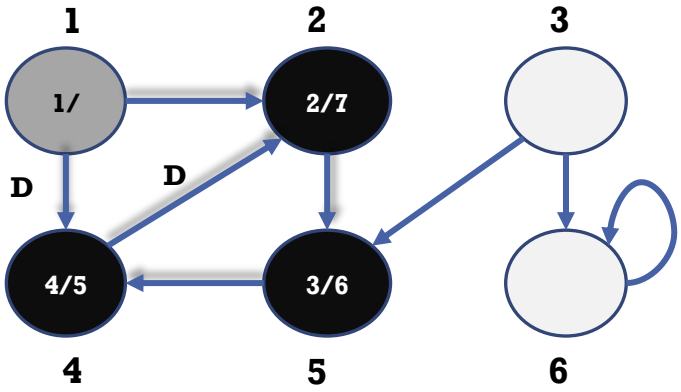
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



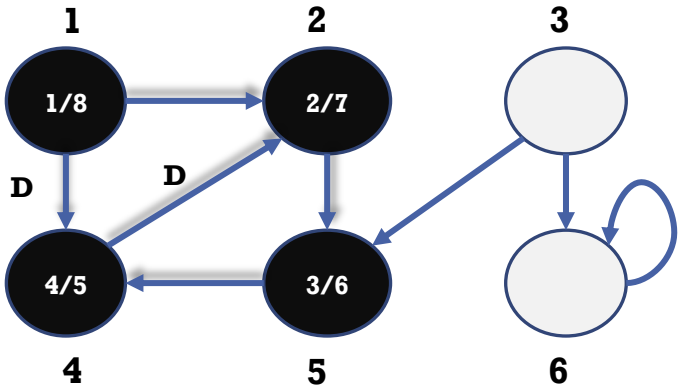
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



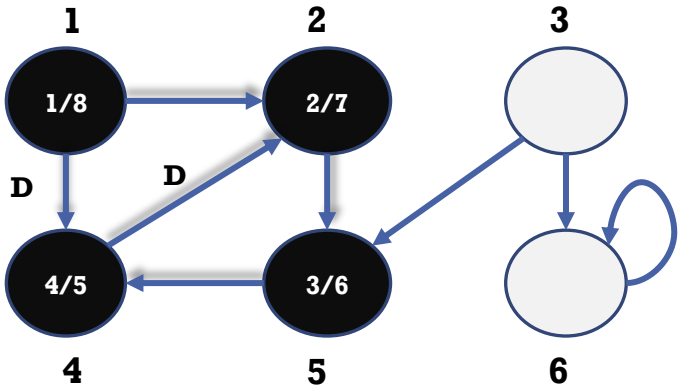
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



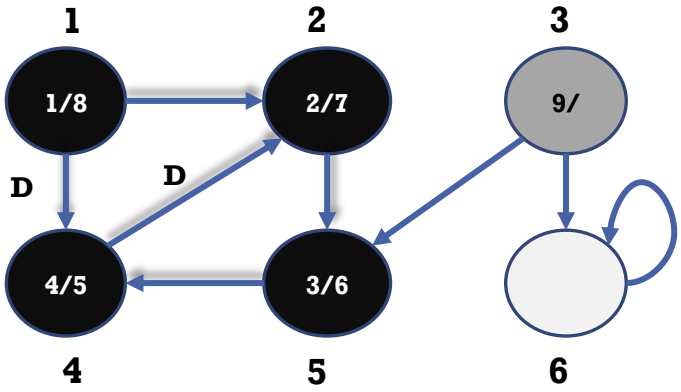
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



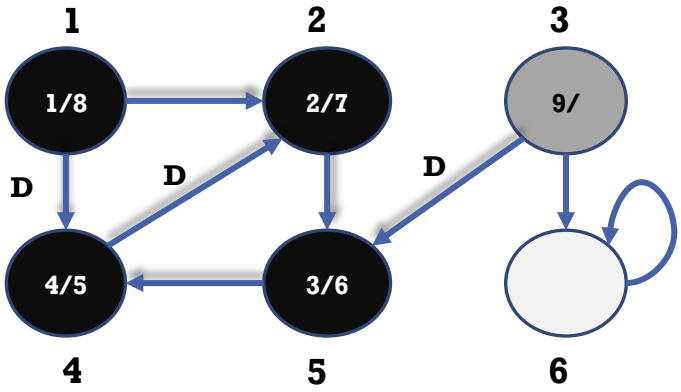
Una vez finalizado el recorrido de todos los nodos adyacentes del primer nodo, se continúa con el siguiente nodo. Sin embargo, debido a que el siguiente nodo (dos) ya fue descubierto, se sigue con el siguiente nodo, el nodo tres, siendo éste el nuevo u.



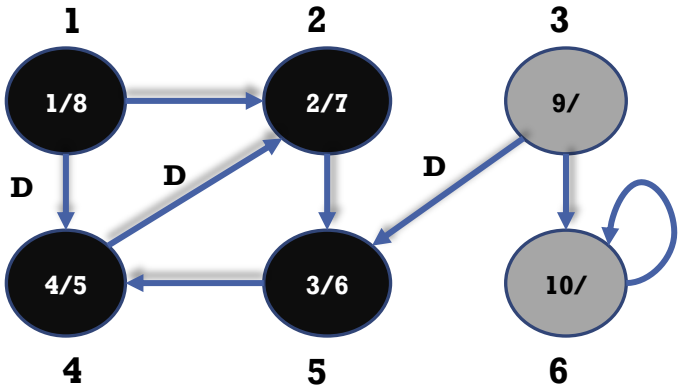
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



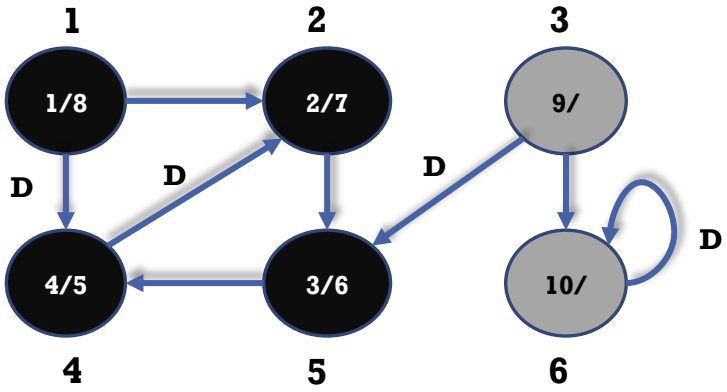
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



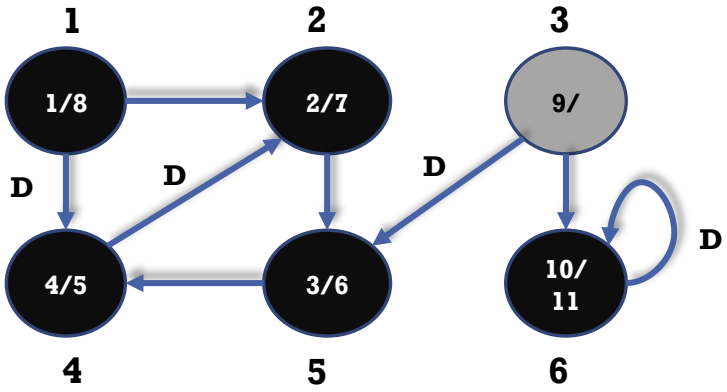
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



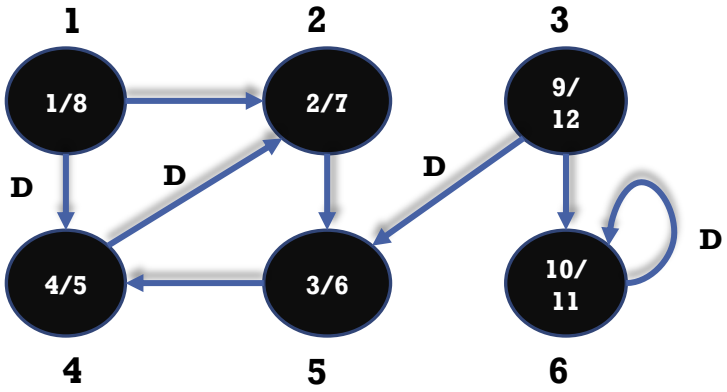
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



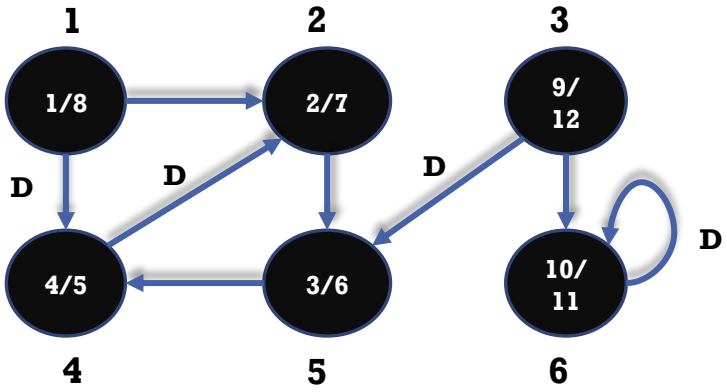
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



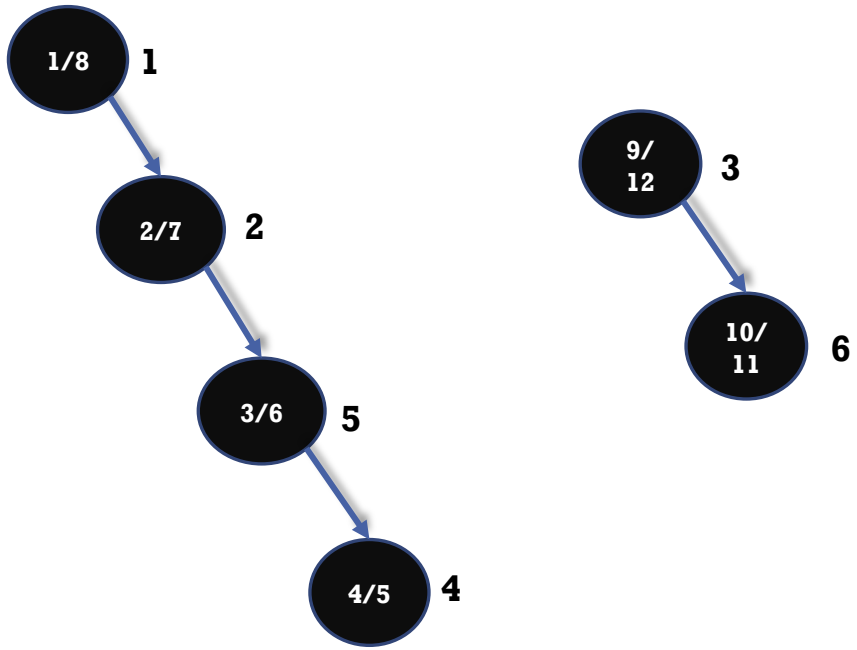
Se visita cada nodo adyacente de u, es decir, v. Cada v, a su vez, se convierte en un nodo inicial y el proceso se repite con sus nodos adyacentes. Cada que se visita un nodo se graba la marca de tiempo.



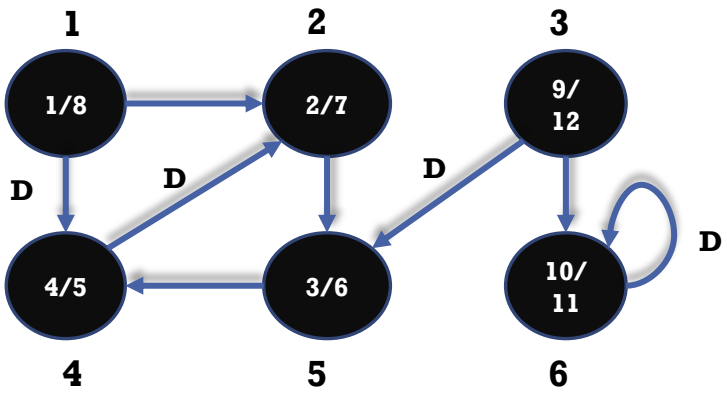
Después de terminar el recorrido del nodo 3, se visitan el resto de los nodos. Para este caso, todos ya fueron descubiertos (son negros), por tanto, el recorrido de los nodos adyacentes ya no se realiza.

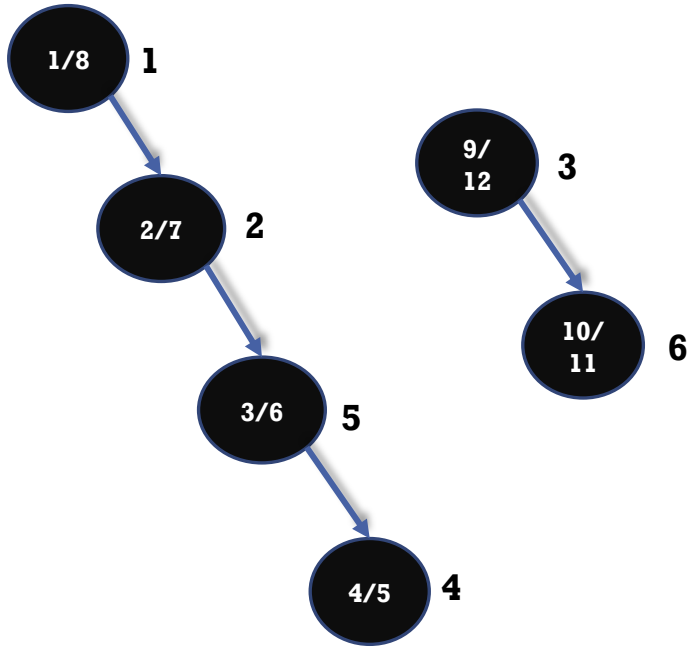


Cada que se realiza un recorrido de los nodos adyacentes de u se genera un árbol, es este caso se generaron dos.



Nótese que los nodos marcados como ya descubiertos (D) no son parte del árbol de predecesores.





El siguiente pseudocódigo representa el algoritmo básico de depth first search (DFS):

```
depth_first_search (G)
        for each vertex u ∈ V[G]
                color[u] ← WHITE
                 predecessor[u]← NIL
        time ← 0
        for each vertex u ∈ V[G]
                if color[u] = WHITE
                        DFS-VISIT(u)
```

```
DFS-VISIT (u)
        color[u] ← GRAY
        times ← times+1
        distance[u] ← times
        for each v ∈ Adj[u]
                if color[v] = WHITE
                        predecessor[v] ← u
                        DFS-VISIT(v)
        color[u] ← BLACK
        finish [u] ← times ← times+1
```

Análisis de complejidad

La función depth_first_search accede a cada nodo de la gráfica G, lo que toma un tiempo de O(V).

La función DFS-VISIT es llamada una vez por cada vértice v ∈ V. Internamente, la función visita todos los nodos adyacentes de v lo que representa un costo O(A).

Por tanto, el tiempo total de ejecución del algoritmo depth first search es O(V + A).

"What is programming?... Some people call it a science, some people call it an art, some people call it a skil or trade."

Charles Simonyi
(Is a Hungarian-born American computer businessman.)

Algoritmos de grafo. Parte II.

Práctica 7

7. Algoritmos de grafo. Parte II.

- Implementar en lenguaje Python el algoritmo de búsqueda por profundidad dentro de un grafo dirigido y no dirigido, con costo y sin costo.
- Obtener la complejidad algorítmica del algoritmo búsqueda por profundidad.
- Crear las gráficas de la complejidad que tiene el algoritmo de búsqueda por profundidad dentro de un grafo para el mejor caso, el peor caso y el caso promedio.

3 Algoritmos de grafos

Objetivo: Aplicar las formas de representar y operar los grafos y listas lineales para representarlos en la computadora.

3.1 Representación de grafos.

3.2 Búsqueda por expansión.

3.3 Búsqueda por profundidad.