Algoritmos Elementales de Grafos

Agustín J. González

ELO-320: Estructura de Datos Y Algoritmos

1er.Sem. 2002

Introducción

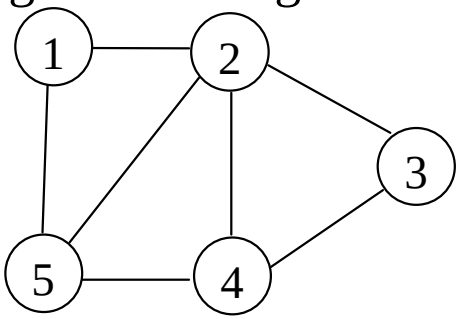
- Estudiaremos métodos para representar y explorar o recorrer grafos.
- Explorar un grafo significa seguir sistemáticamente los arcos de un grafo para visitar sus vértices.
- Las dos representaciones más comunes para representar grafos son: Lista de adyacencia y matriz de adyacencia.
- **Representación de grafos**
- Un Grafo G =(V, E) , V : conjunto de vértices y E conjunto de arcos, se representa preferiblemente con una lista de adyacencia porque ésta permite una representación compacta cuando el grafo es **disperso**; i.e. cuando $|E| << |V|^2$
- Es preferible usar una representación con Matriz de Adyacencia cuando el grafo es **denso**; i.e. $|E| \sim |V|^2$,.o cuando es preciso saber rápidamente si hay un arco conectando dos vértices.

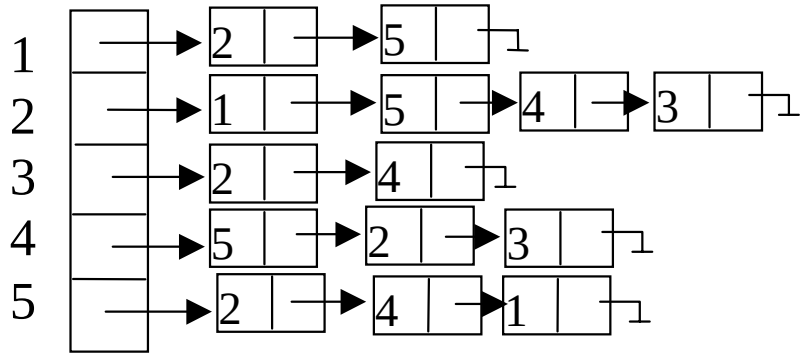
Representación con Listas de Adyacencia

- En este caso el Grafo G=(V, E) consiste de un arreglo Adj que almacena |V| listas, una para cada vértice en V.
- Para cada $u \in$ V, la lista de adyacencia Adj[u] contiene (punteros a) todos los vértices *v* tal que hay una arco $(u,v) \in$ E.
- Si el grafo es dirigido, se cumple que la suma de los largos de las listas de adyacencia es |E|.
- Si el grafo no es dirigido, se cumple que la suma de los largos de las listas de adyacencia es 2*|E|. Dado que cada arco aparece dos veces.
- En cualquier caso la memoria requerida es O(max(|V|,|E|)) = O(|V|+|E|).

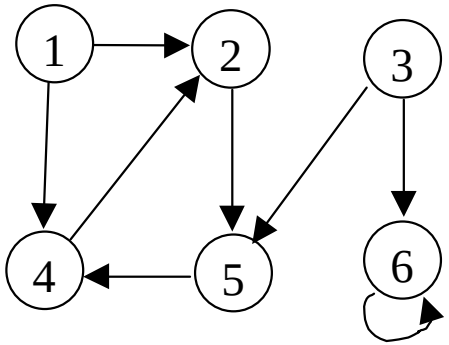
Representación con Listas de Adyacencia: Ejemplo

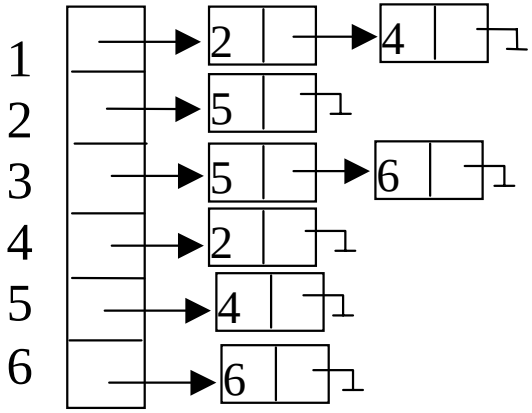
- Caso grafo no dirigido





- Caso Grafo dirigido

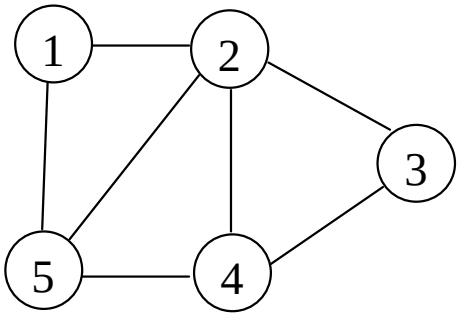




- Las listas de adyacencia pueden ser fácilmente adaptadas para representar ***grafos con peso.*** En estos un peso es asociado a cada arco a través de una función de peso ***w***: E --> **R.**
  Así el peso del arco (u,v) es puesto en el nodo ***v*** de la lista ***u***.

Representación con Matriz de Adyacencia: Ejemplo

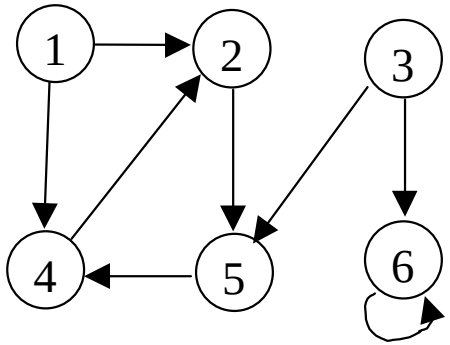
- Caso grafo no dirigido



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

  - Notar la simetría. Para ahorrar memoria se puede almacenar sólo la mitad.

- Caso Grafo dirigido



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

- Si el grafo es con peso, el peso se almacena en la matriz. Cuando un arco no existe se toma algún valor que represente su ausencia 0, -1 etc. Dependiendo de la aplicación.
  La matriz de adyacencia es preferible cuando el grafo es pequeño por su simplicidad.

Algoritmos de Exploración de un grafo.

- La idea es visitar todos los vértices siguiendo los arcos.
- “Breadth-first search” búsqueda (visitar) primero por distancia (todos de igual distancia se visitan primero)
- Dado un vértice fuente ***s***, Breadth-first search sistemáticamente explora los arcos del grafo G para “descubrir” todos los vertices alcanzables desde ***s***.
- También calcula la distancia (menor número de arcos) desde ***s*** a todos los vértices alcanzables.
- También produce un árbol con raíz en ***s*** y que contiene a todos los vértices alcanzables.
- El camino desde s a cada vértice en este recorrido contiene el mínimo número de arcos. Es el camino más corto medido en número de vértices.
- Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto. Llega a los nodos de distancia k, sólo luego de haber llegado a todos los nodos a distancia k-1.

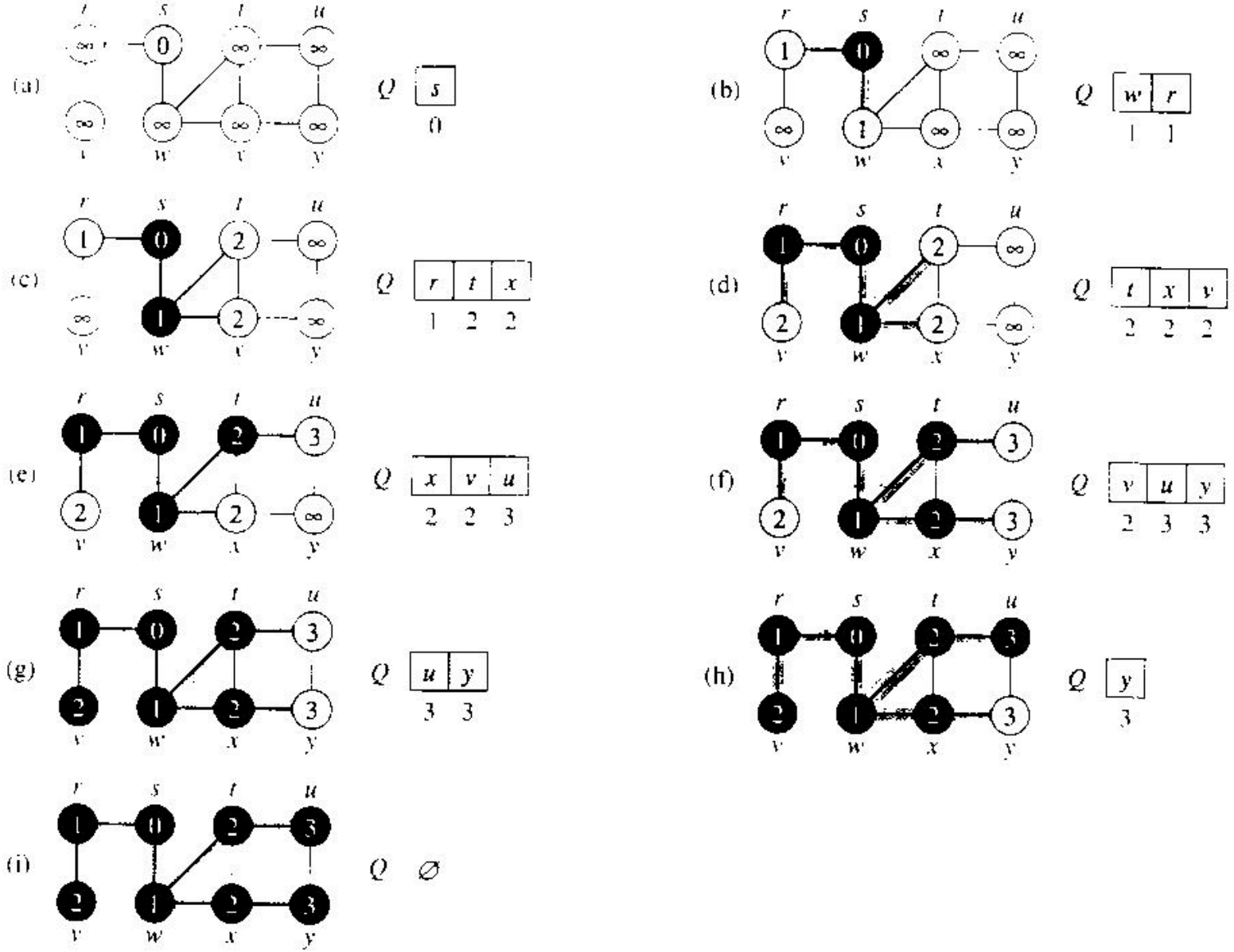
Algoritmos “Breadth-first search” (BFS)

- Inicialmente el algoritmo colorea los vértices con blanco. Luego éstos pasan a plomo y luego negro. El color plomo es usado para definir la frontera entre lo ya descubierto o explorado y lo por visitar.
- BFS(G,s) { /* pseudo-código */

```
        int d[N], p[N], color[N]; /* Arreglos de distancia, de padres, y de color */
        QUEUE Q; /* Cola usada como estructura auxiliar */
        for ( cada vértice u ∈ V[G] -{s}) {
                color [u] =Blanco;
                d[u] = ∞;  /* distancia infinita si el nodo no es alcanzable */
        }
        color[s] =Plomo;
        d[s] = 0;
        p [s]=NULL;
        Enqueue(Q, s);
        while ( !Queue_Vacía(Q) ) {
                u = Cabeza(Q);
                for ( cada v ∈ Adj [u] ) {
                        if (color [v] == Blanco) {
                                color[v]=Plomo;
                                d [v]=d [u] +1;
                                p [v] = u;
                                Enqueue(Q, v);
                        }
                Dequeue(Q); /* se extrae u */
                color [u] = Negro;
        }
}
```

- El tiempo de ejecución es O(|V|+|E|). Notar que cada nodo es encolado una vez y su lista de adyacencia es recorrida una vez también.

Ejemplo de Breadth-first search
“Recorrido o Búsqueda de nodos en amplitud”



Algoritmos “Depth-first search” (DFS)

- Como en BFS, inicialmente el algoritmo colorea los vértices con blanco. Luego éstos pasan a plomo y luego negro. Aquí el color plomo es usado para definir nodos cuyos descendientes están siendo visitados.
- int tiempo; /* global */
  int d[N], f[N], p[N], color[N]; /* Arreglos de tiempo de entrada, tiempo de salida, padres, y color */
- DFS(G) { /* pseudo-código */

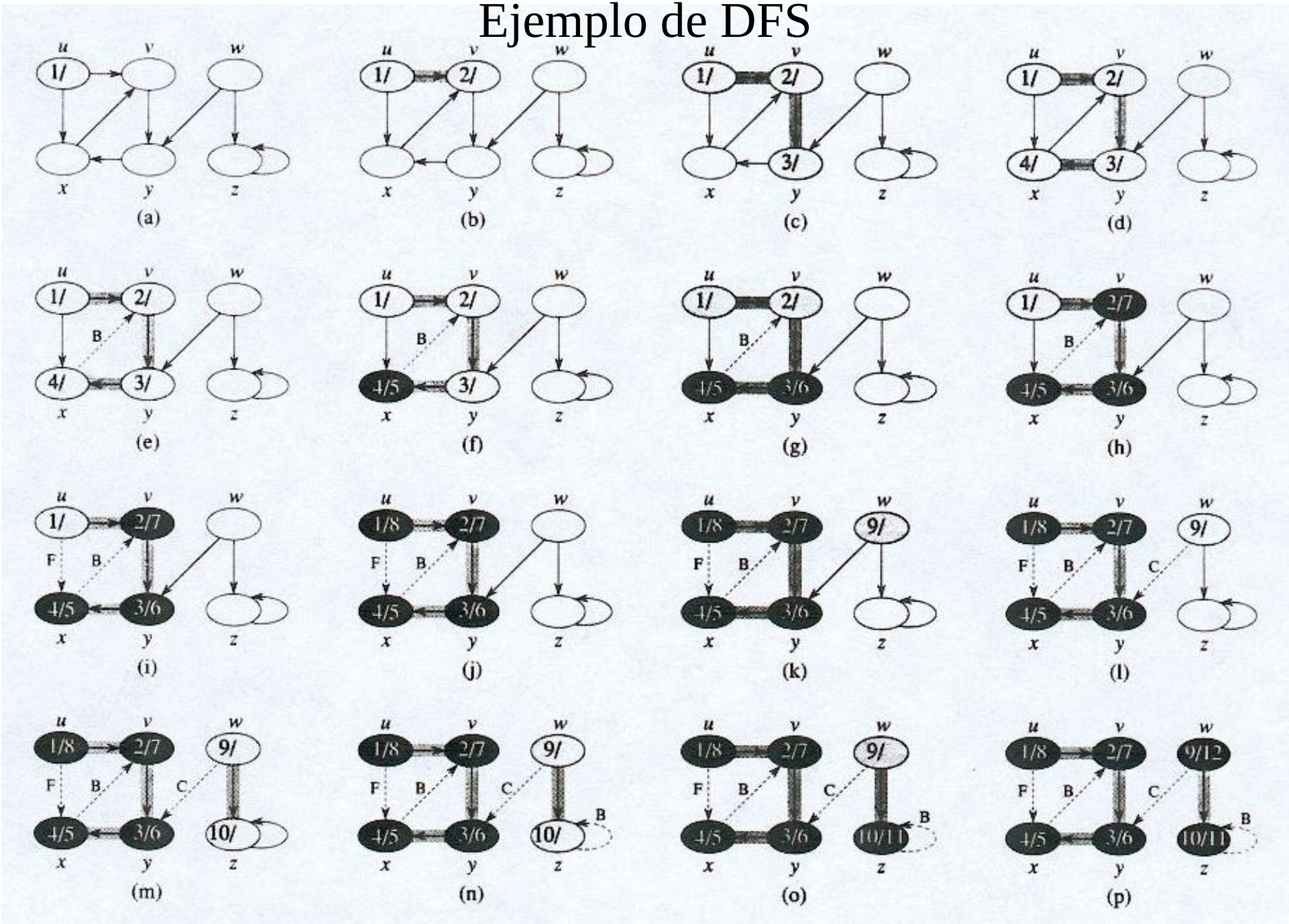
```
        for ( cada vértice u ∈ V[G]) {
                color [u] =Blanco;
                p[u] = NULL;
        }
        tiempo = 0;
        for (cada vértice  u ∈ V[G])
                if (color[u] == Blanco)
                        DFS_visit(u);
}
```

- DFS_visit (u) /* pseudo-código */

```
        color [u]= Plomo;  /* Vértice Blanco u es visitado, ingresamos a su sub-árbol */
        d[u] = ++tiempo; /* el tiempo se incrementa cada vez que “entramos y salimos” de un nodo*/
        for ( cada v ∈ Adj [u] ) {  /* explora arcos (u,v) */
                if (color [v] == Blanco) {
                        p [v] = u;
                        DFS_visit(v);
                }
        }
        color [u] = Negro;   /* ennegrezca u, salimos de su sub-árbol */
        f [u] = ++tiempo;
}
```

- El tiempo de ejecución de DFS es también O(|V|+|E|). Cada arco y nodo es recorrido una vez.

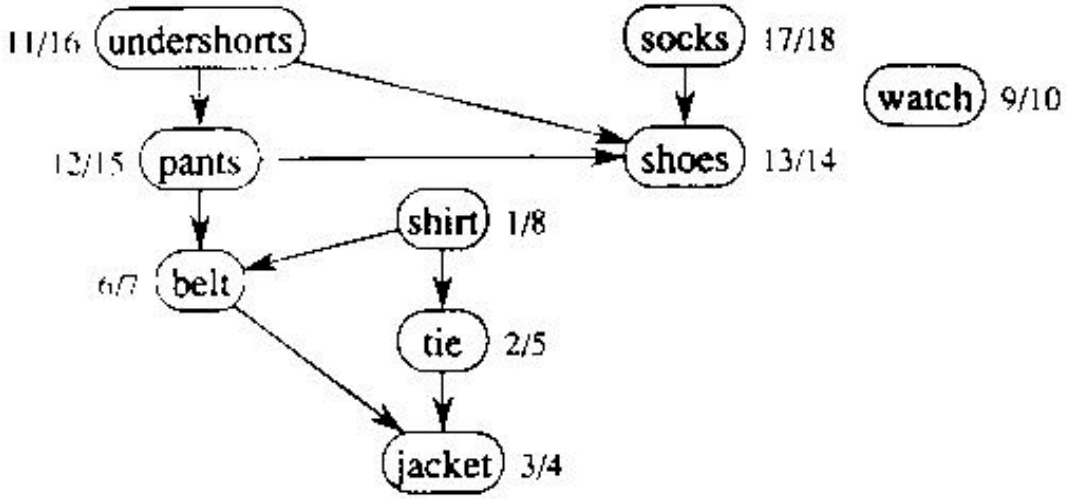
Ejemplo de DFS

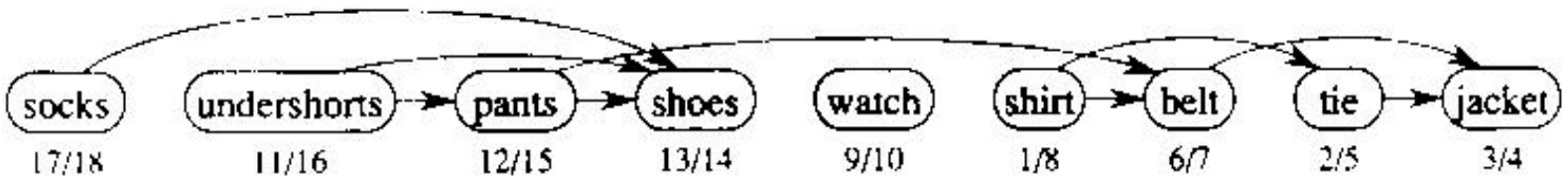


Orden Topológico

- El orden topológico tiene sentido sólo en grafos aciclicos dirigidos (DAG).
- Orden topológico de un dag G=(V,E) es un orden lineal de todos los vértices tal que si G contiene el arco (u,v), entonces ***u*** aparece antes que ***v*** en el orden.
- Cuando se tienen muchas actividades que dependen parcialmente unas de otras, éste orden permite definir un orden de ejecución sin conflictos.
- Gráficamente se trata de poner todos los nodos en una línea de manera que sólo haya arcos hacia delante.
- Algoritmo:
  - Topological_Orden(G)
    Llamar a DFS(G) para calcular el tiempo de término f[v] para cada vértice.
    Insertar cada nodo en una lista enlazada según su orden de término.
    Retornar la lista enlazada

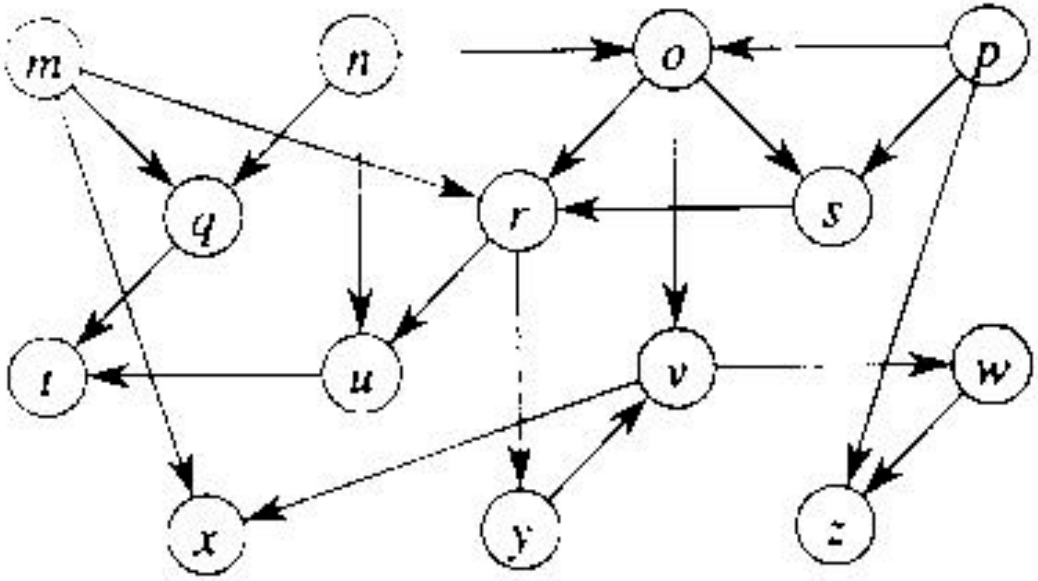
Ejemplo: Orden topológico





¿Es teste el único orden topológico?

Ejemplo: Orden topológico



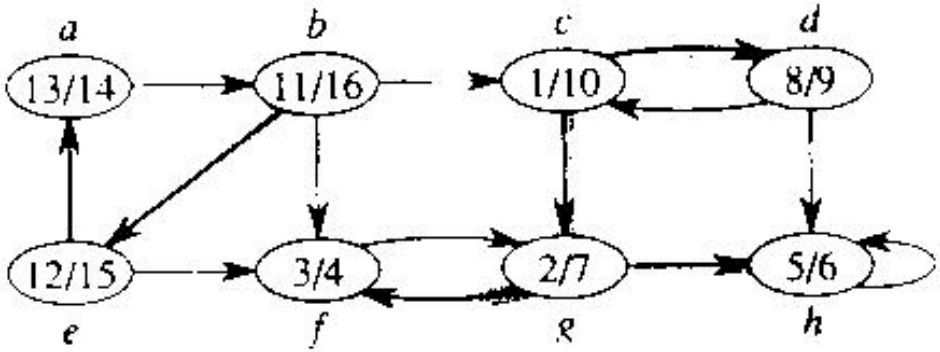
¿Cuál es el orden topológico?

Detección de componentes fuertemente Conexas

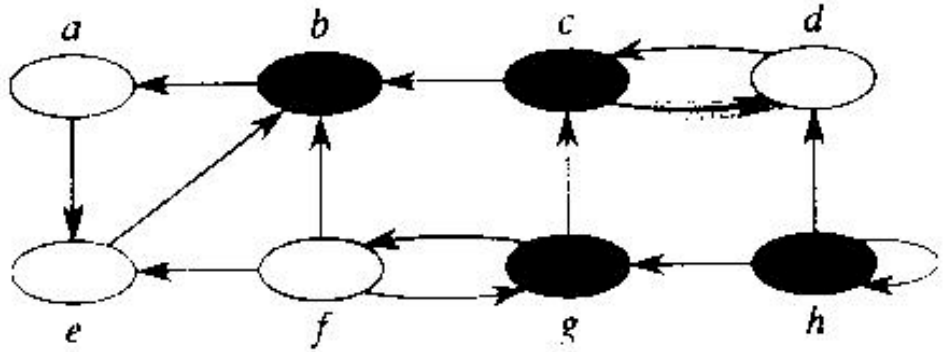
- Una componente fuertemente conexa de un grafo G=(V,E) es el máximo conjunto de vértices U subconjunto de V tal que para cada par de vértices u, v en U, existan caminos desde u a v y viceversa.
- El algoritmo descubre todas las componentes fuertemente conexas. Para ello define el grafo traspuesto de G, G^T= (V,E^T), donde E^T={(u,v) tal que (v,u) pertenece a E}. En otras palabras, invierte el sentido de todas los arcos.
- Algoritmo:
- Strongly_Connected_Components(G)
  1.- Llamar a DFS(G) para obtener el tiempo de término f[u], para cada vértice u;
  2.- Calcular G^T;
  3.- Llamar a DFS(G^T), pero en el loop principal de DFS, considerar los vértices en orden decreciente de f [u].
  4.- La salida son los vértices de cada árbol de la foresta del paso 3. Cada árbol es una componente fuertemente conexa separada.

Ejemplo de Detección de Componentes fuertemente conexas

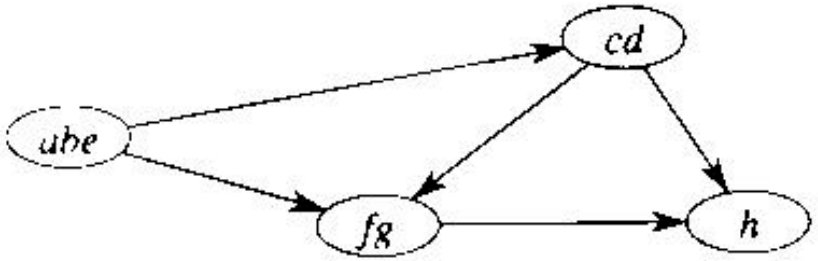
(a)



(b)



(c)



¿Por qué funciona?

- No haremos una demostración rigurosa, pero si daremos algunos elementos que ayudan a su entendimiento.
- Cuando se recorre un grado en DFS se tiene: si v es un descendiente de u entonces f[v]<f[u].
- Si v es descendiente de u, v es alcanzable desde u.
- La conectividad de nodos en una componente conexa es es invariante con respecto a la inversión de arcos. Si de v llegamos a u, y de u llegamos a v, al invertir los arcos esta propiedad se mantiene.
- Al visitar los nodos de G^T usando orden de término decreciente, para cada árbol estaremos partiendo por los mismos nodos que usamos en el recorrido de G. Tendremos que llegar a todos los conectados de todas formas y éstos definirán las componentes fuertemente conexas.