Teoría de Grafos I

Bach. Rodolfo Mercado Gonzales
Universidad Nacional de Ingeniería

Grafo

- Un grafo modela entes u objetos y las relaciones entre ellos.

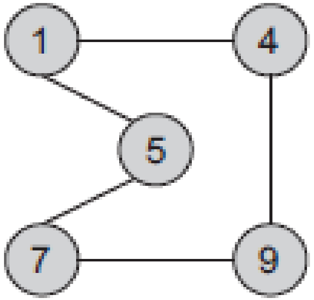
- Está formado por un conjunto **V** de vértices o nodos que representan a los entes y un conjunto **E** de aristas (*edges)* o arcos que representan las relaciones entre los entes.

Grafo

Un grafo es un par $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, donde $\boldsymbol{V}$ es un conjunto de vértices y $\boldsymbol{E} \subseteq \boldsymbol{V} \, \boldsymbol{x} \, \boldsymbol{V}$ es un conjunto de aristas.

Representación Gráfica

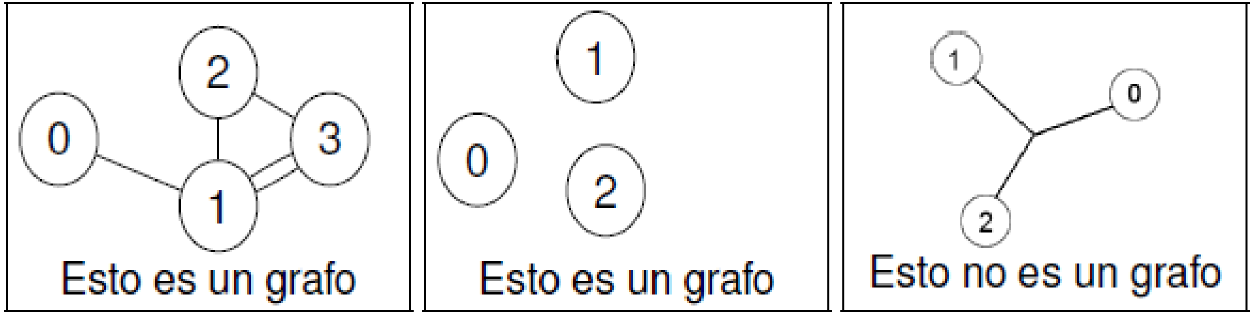
Generalmente los vértices son representados por puntos y las aristas por líneas que unen un par de puntos.



$$V = \{1, 4, 5, 7, 9\}$$

$$E = \{ (1, 4), (1,5), (4,9), (7,9), (5,7) \}$$

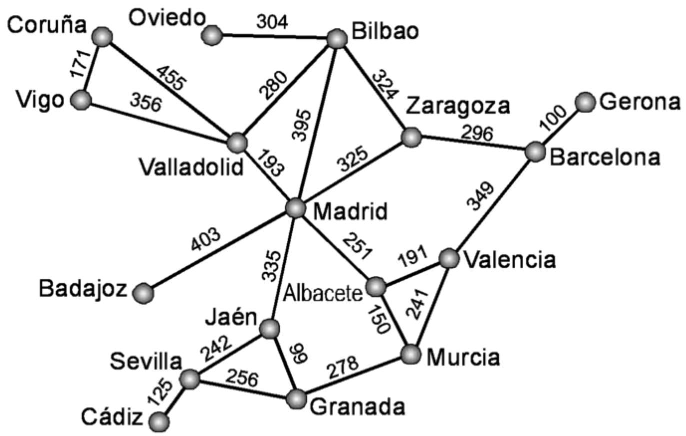
Representación de un Grafo



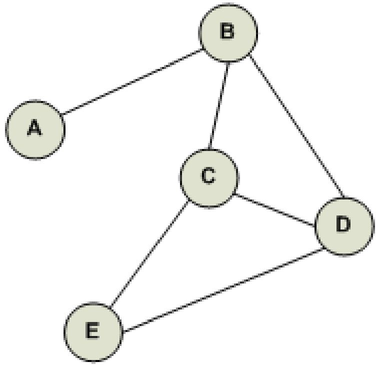
Representación de un Grafo

Algunas veces al modelar un grafo la relación entre dos vértices tiene asociada una magnitud, generalmente denominada peso *(weight).*
Al grafo con pesos se le denomina *weighted graph*, y si no tiene pesos es *unweighted graph*.

Por ejemplo el siguiente es un grafo de ciudades interconectadas, donde se le asocia a cada arista la distancia en km entre el par de ciudades que une.

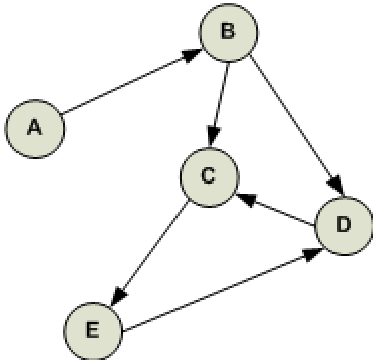


Tipos de Grafos



Grafo No Dirigido (*undirected graph)*

Las aristas no presentan orientación. La arista (u, v) es la misma que (v, u).



Grafo Dirigido (*directed graph*)

Las aristas presentan orientación. La arista (u, v) se dice que es dirigida de u a v, donde u es el vértice de inicio y v el vértice de fin.

Definiciones

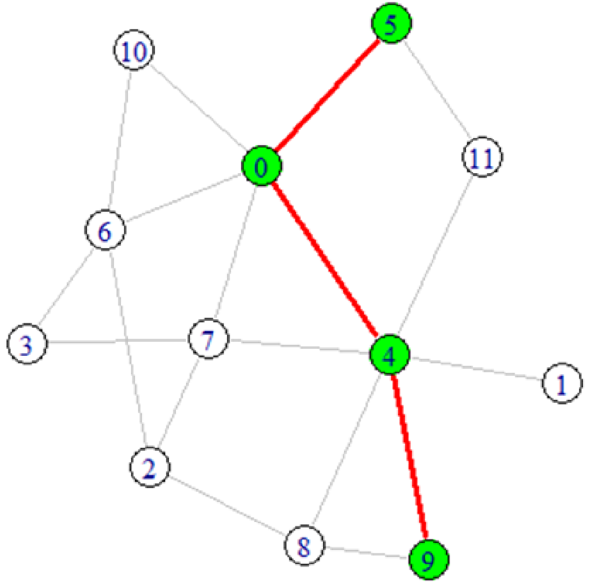
Camino

Un camino de longitud n es una secuencia de $n + 1$ vértices
$v_0, v_1, \dots, v_n$ tal que $(v_i, v_{i+1}) \in E$ para $0 \; <= \; i \; < \; n$.

Un **camino** es **simple** si es que no existen vértices repetidos en la secuencia.

Camino = *Walk*

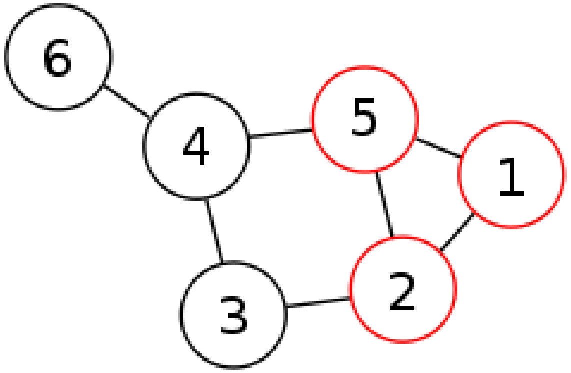
Camino Simple = *Path*



Definiciones

Ciclo (*cycle*)

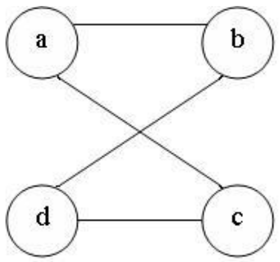
Es un camino donde el vértice inicial y el final son el mismo.



Ciclo simple (*simple cycle*)

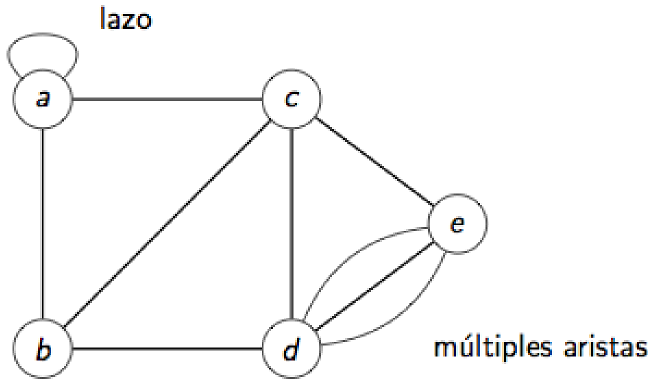
Es un ciclo en donde además ningún vértice se repite (a excepción del inicial y final).

Tipos de Grafos



Grafo Simple (*simple graph*)

Es el grafo más común, en el cual no se presentan aristas múltiples ni lazos.



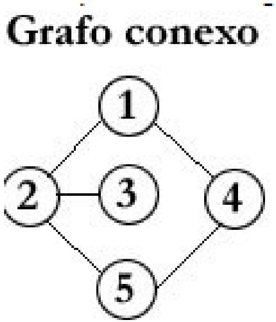
Multigrafo (*multigraph*)

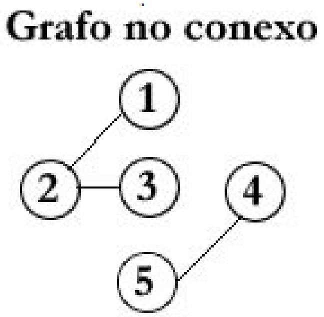
Posee aristas múltiples ***(multiple edges)***, es decir aristas que unen el mismo par de nodos ; y lazos ***(self loops***), que son aristas que unen un nodo consigo mismo.

Tipos de Grafos

Grafo Conexo *(connected graph)*

Un grafo no dirigido es conexo, si para todo par de vértice (u, v) existe un camino desde u hacia v.

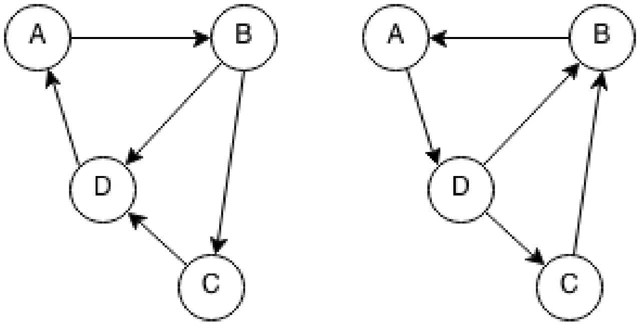




Tipos de Grafos

Grafo Fuertemente Conexo *(strongly connected)*

Un grafo dirigido se denomina fuertemente conexo si para cada par de vértices (u, v) existe un camino dirigido de ida (de u hacia v) y de regreso (de v hacia u).

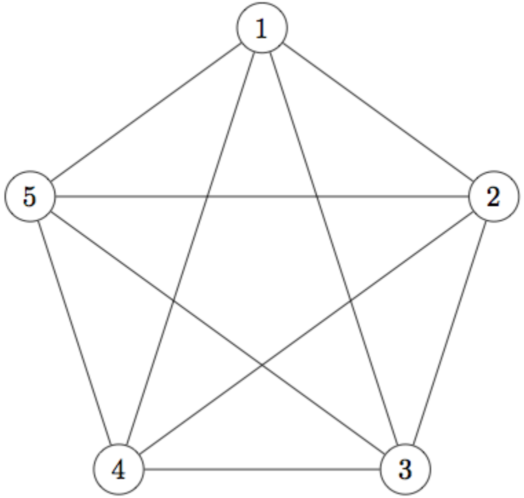


Tipos de Grafos

Grafo completo *(complete graph)*

Grafo simple no dirigido en el que todo par de vértices está relacionado por una arista.

Si el grafo posee n vértices, el número de aristas será $n * (n-1)/2$

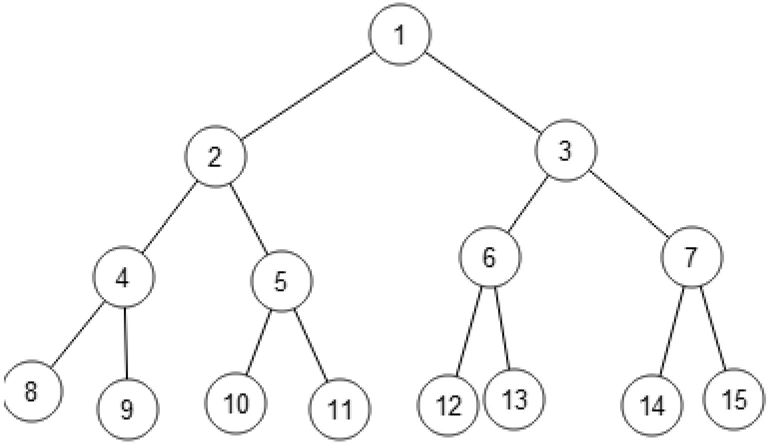


Tipos de Grafos

Árbol (*tree*)

Grafo no dirigido, conexo y acíclico. ***(undirected, connected, acyclic)***

Un árbol con n vértices tiene $n - 1$ aristas.



Bosque (*forest*)

Es un grafo conformado por varios árboles

Propiedades de árboles

Theorem 2.1.4. For an n-vertex graph ($n \geq 1$), the following are equivalent and characterize the trees with n vertices.

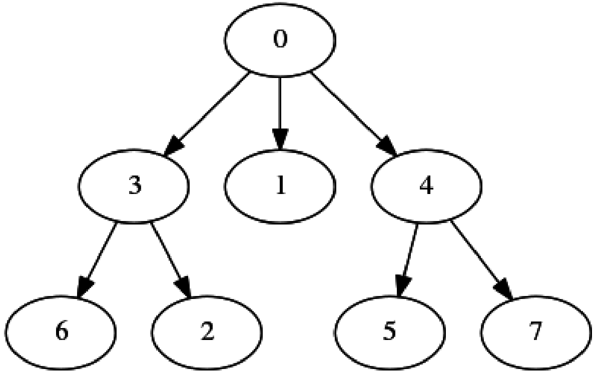
A) G is connected and has no cycles.
B) G is connected and has $n - 1$ edges.
C) G has $n - 1$ edges and no cycles.
D) For $u, v \in V(G)$, there is exactly one path from u to v.

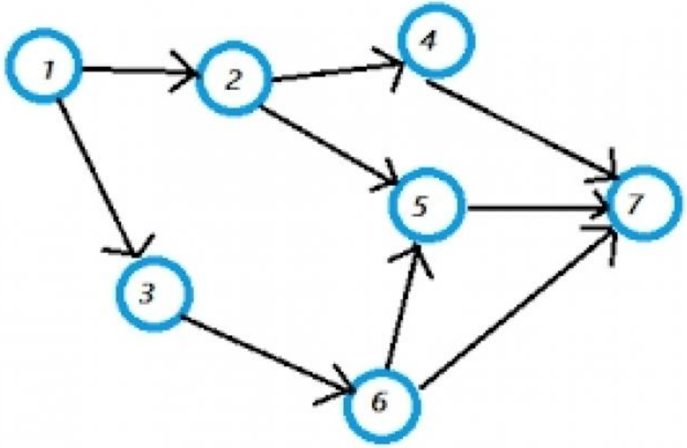
Basta cualquiera de esas combinaciones de propiedades (A o B o C o D) para que un grafo sea árbol

Tipos de Grafos

Directed Acyclic Graph (DAG)

Grafo dirigido y acíciclo





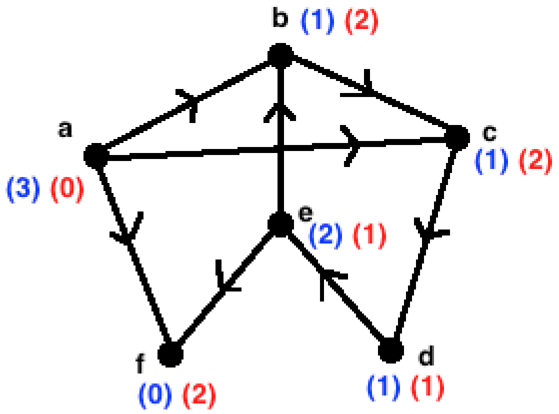
Definiciones

Indegree

Está definido solo en grafo dirigidos. El indegree de un vértice es la cantidad de aristas que entran a él.

Outdegree

Está definido solo en grafo dirigidos. El outdegree de un vértice, es la cantidad de aristas que salen de él.



Definiciones

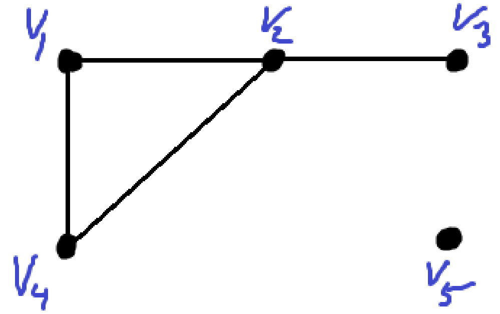
Degree

Está definido tanto en grafos dirigidos como en no dirigidos.

En grafos dirigidos es la suma del indegree y outdegree.

En grafos no dirigidos, es la cantidad de aristas incidentes en él.

Nota : Si en un grafo no dirigido hay self loops, esa arista se cuenta doble en el degree.



$deg(v_1) = 2$
$deg(v_2) = 3$
$deg(v_4) = 2$

$deg(v_3) = 1$
$deg(v_5) = 0$

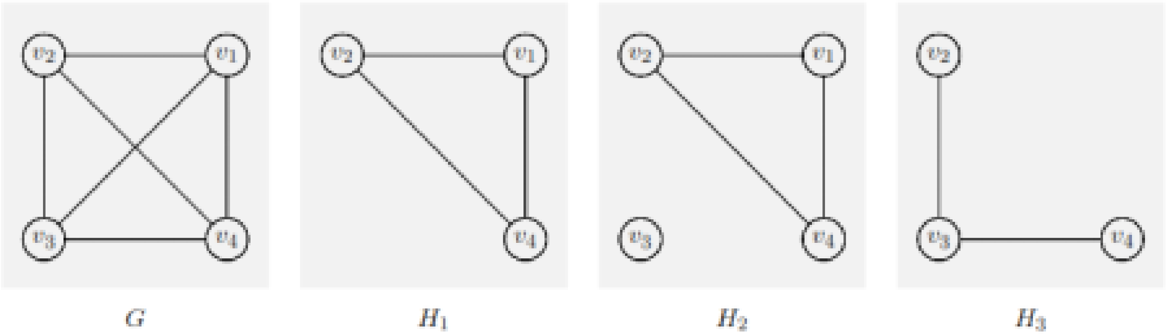
Definiciones

Handshaking lemma (degree sum formula)

$$\sum_{v \in V} \deg v = 2|E|$$

Subgrafo (subpgraph)

Un subgrafo de un grafo $G = (V, E)$ es un grafo $H = (V_H, E_H)$ tal que $V_H \subseteq V$ y $E_H \subseteq E$

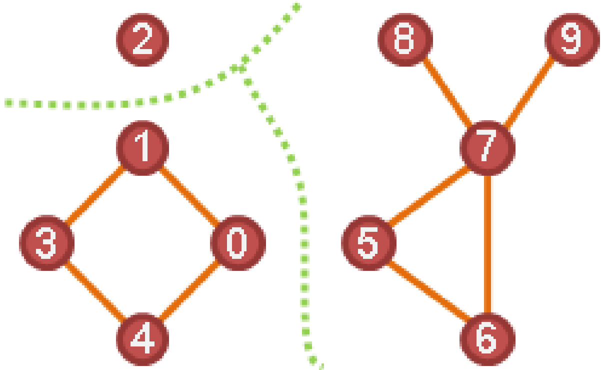


Subgrafos

Subgrafo

Componente Conexa (*connected component)*

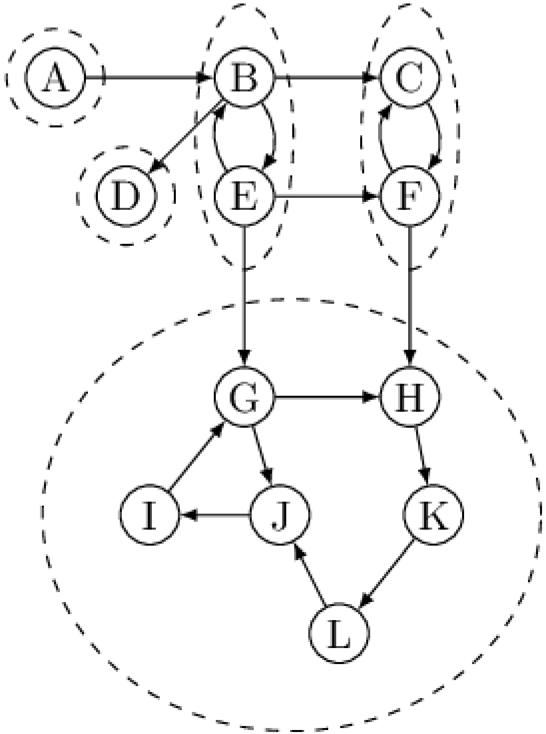
- ❑ Es un subgrafo conexo maximal de un grafo no dirigido.
- ❑ Un subgrafo conexo es maximal, si no existe otro subgrafo conexo que lo contenga.



Subgrafo

Componente Fuertemente Conexa
(*strongly connected component*)

- ❑ Es un subgrafo fuertemente conexo maximal de un grafo dirigido.



Representación Computacional

Para poder aplicar nuestros algoritmos, necesitamos representar un grafo de forma computacional.

Para ello las representaciones más usadas son:

- ❑ Matriz de adyacencia
- ❑ Lista de adyacencia

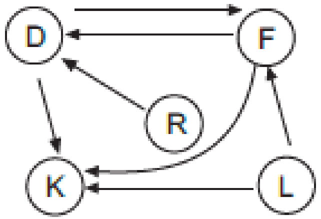
Matriz de Adyacencia

- Es la forma más sencilla de representar un grafo.
- Dado un grafo $G = (V, E)$ de n vértices, donde $V = \{v_0, v_1, \dots, v_n\}$ y $E = \{ (v_i, v_j)\}$
  - Los vértices se pueden representar por números consecutivos de 0 a $n - 1$.
  - Las aristas se representan mediante una matriz A de $n\ x\ n$ elementos, donde cada elemento a_{ij} toma los valores:

$$a_{ij} = \begin{cases} 1 & \text{si hay un arco } (v_i, v_j) \\ 0 & \text{si no hay arco } (v_i, v_j) \end{cases}$$

Matriz de Adyacencia

Grafo dirigido

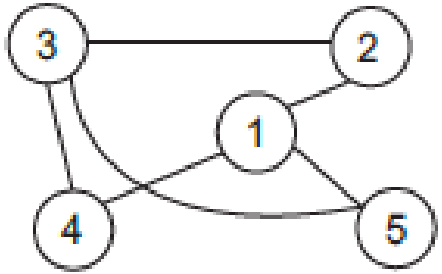


Grafo dirigido con los vértices {D, F, K, L, R}.

$$A = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{vmatrix}$$

Matriz de Adyacencia

Grafo no dirigido



$$A = \begin{vmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{vmatrix}$$

Matriz de Adyacencia

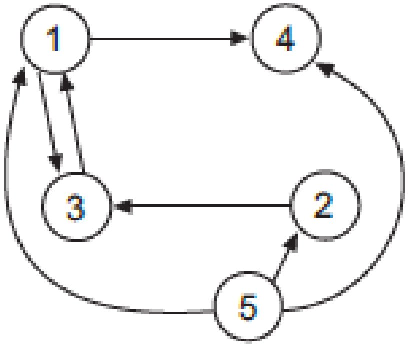
La matriz de adyacencia deja de ser eficiente cuando el grafo presenta muy pocas aristas, ya que **se reserva espacio de memoria innecesario**, por ello en la mayoría de casos se prefiere usar la lista de adyacencia.

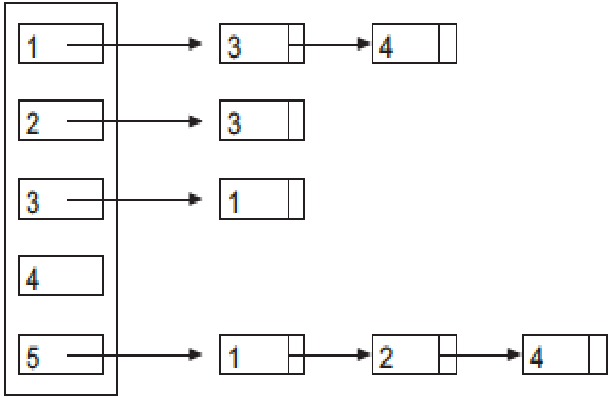
Lista de Adyacencia

Dado un grafo $G = (V, E)$, esta estructura consiste de un arreglo **adj** de $|V|$ listas (una por cada uno de los vértices). Además para cada vértice u, su lista de adyacencia adj[u] contiene todos los vértices v con los cuáles comparte una arista.

Lista de Adyacencia

Grafo Dirigido





Recorrido de un Grafo

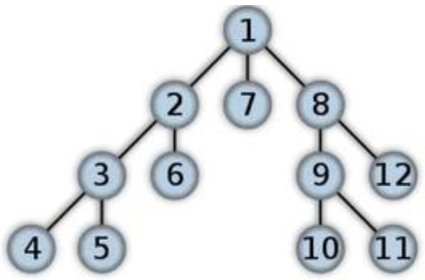
Recorrer un grafo consiste en visitar todos los vértices alcanzables desde un vértice fuente. Tenemos dos formas de realizar el recorrido:

- ❑ Búsqueda en Profundidad
- ❑ Búsqueda en Amplitud

Búsqueda en Profundidad (DFS)

- El objetivo de este algoritmo es explorar o visitar todos los vértices alcanzables desde un vértice fuente.

- Este algoritmo es conocido como DFS (depth first search), ya que sigue la estrategia de visitar los vértices teniendo como prioridad la profundidad.



Búsqueda en Profundidad (DFS)

ExplorarAlcanzables (u) : marcará como explorados todos los vértices alcanzables desde el vértice u.

ExplorarAlcanzables (u) = marcar u como explorado y
ExplorarAlcanzables(v) , $\forall v$ vecino de u

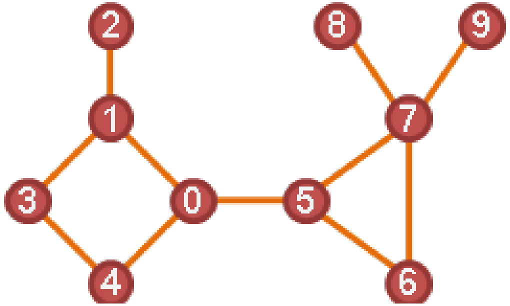
Búsqueda en Profundidad (DFS)

$$O(V + E)$$

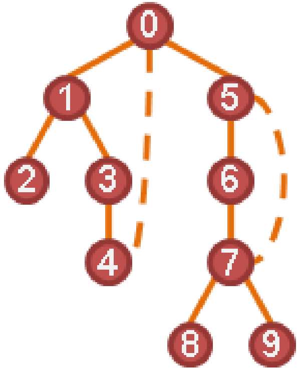
```
const int MAX_V = 100000;
vector<int> adj[ MAX_V + 1 ];
bool vis[ MAX_V + 1 ];

void dfs( int u ){
    if( vis[ u ] ) return;
    vis[ u ] = 1;
    int sz = adj[ u ].size();
    for( int i = 0; i < sz; ++i ){
        int v = adj[ u ][ i ];
        dfs( v );
    }
}

int main(){
    int V, E, x, y;
    cin >> V >> E;
    for( int i = 0 ; i < E; ++i ){
        cin >> x >> y;
        adj[ x ].push_back( y );
        adj[ y ].push_back( x );
    }
    dfs( 0 );
    for( int i = 0; i < V; ++i ) cout << vis[ i ] << endl;
}
```

Árbol del DFS

Grafo No Dirigido

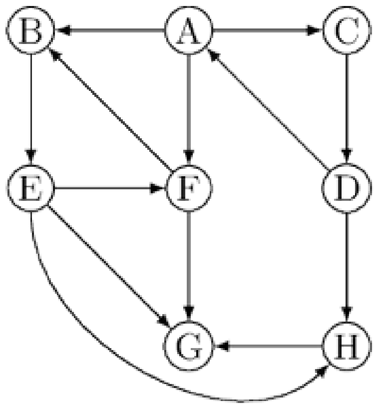


Árbol del DFS

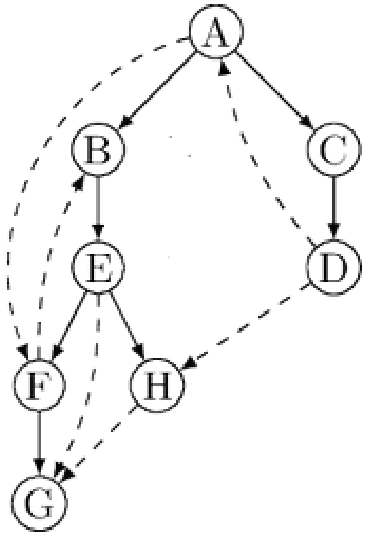


Árbol del DFS

Grafo Dirigido

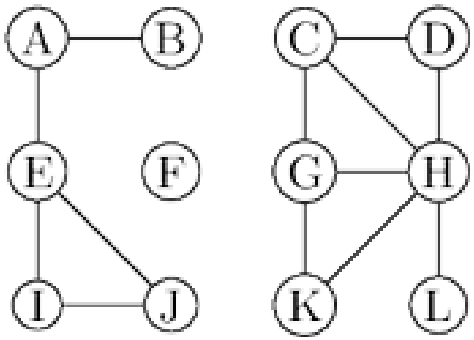


Árbol del DFS

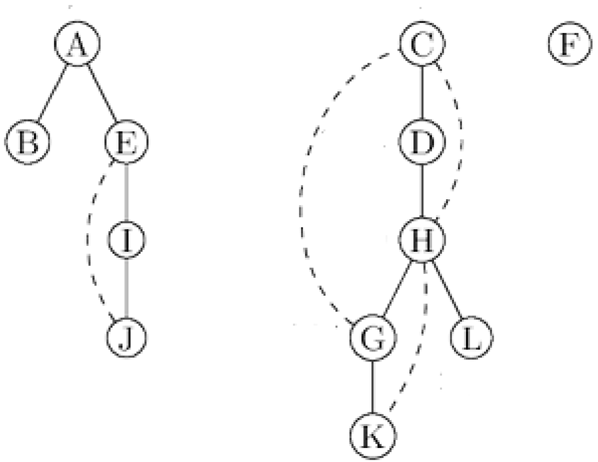


Bosque del DFS

Grafo No Dirigido

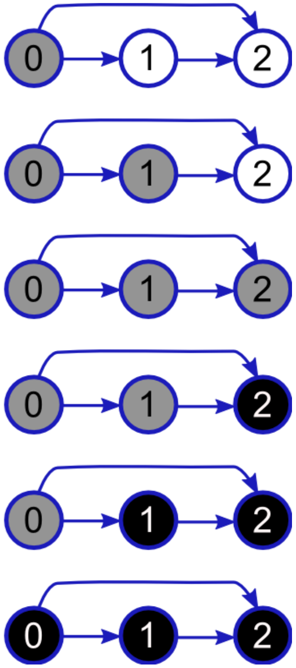


Bosque del DFS



Estado de los Vértices

- ❑ **Blanco (0) :** estado inicial.
- ❑ **Gris (1):** es visitado por primera vez (descubierto) y empieza a explorar sus vecinos.
- ❑ **Negro (2):** terminó de explorar sus vecinos (finalizado)

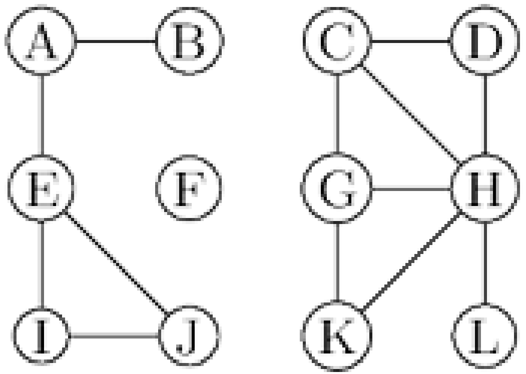


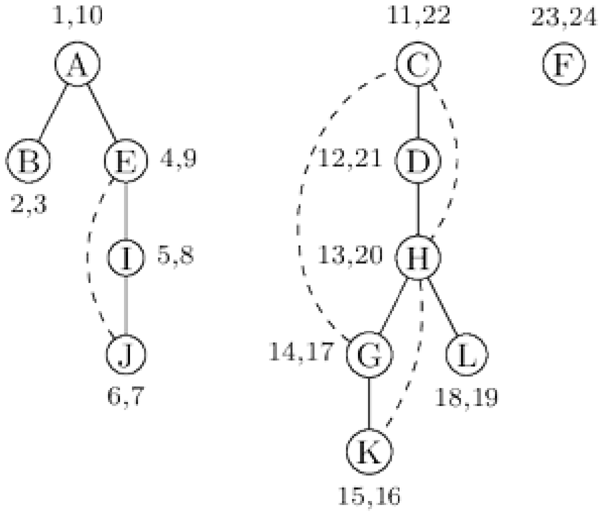
Tiempo de Descubrimiento y Finalización

Podemos usar marcas de tiempo entre $[0, 2n >$ para indicar:

- **Tiempo de Descubrimiento (td):** cuando el vértice se pinta de gris.
- **Tiempo de finalización (tf):** cuando el vértice se pinta de negro.

Tiempo de Descubrimiento y Finalización





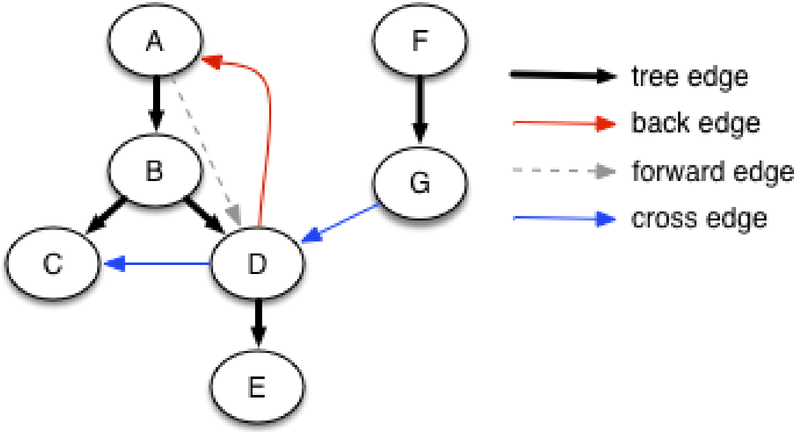
Ancestros y Descendientes

Sea G un grafo dirigido o no dirigido, el vértice v es un descendiente de u en su bosque de DFS, si y solo si:

$$td(u) < td(v) < tf(v) < tf(u)$$

Clasificación de Aristas

- **Tree edges:** (u, v), tal que v fue descubierta explorando la arista (u, v). Se viaja a un nodo blanco.
- **Back edges:** (u, v), tal que v es un ancestro de u en el árbol del DFS. Se viaja a un nodo gris.
- **Forward edges:** (u, v), tal que v es un descendiente de u. Se viaja a un nodo negro tal que $td(u) < td(v)$
- **Cross edges:** (u, v), tal que v no es descendiente ni ancestro de u. Se viaja a un nodo negro tal que $td(u) > td(v)$



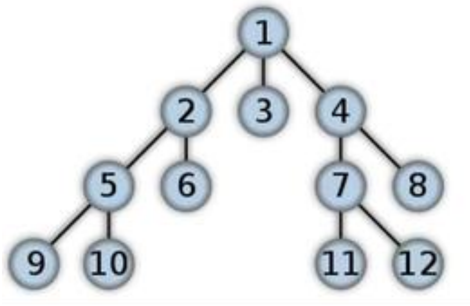
Propiedades

- En un grafo no dirigido solo existen tree edges y back edges.
- Un grafo es acíclico si no presenta back edges.
- En un grafo dirigido acíclico (DAG) para toda arista (u, v), $tf(u) > tf(v)$

Búsqueda en Amplitud (BFS)

- El objetivo de este algoritmo es explorar o visitar todos los vértices alcanzables desde un vértice fuente.

- Este algoritmo es conocido como BFS (breadth first search), ya que sigue la estrategia de visitar los vértices teniendo como prioridad la amplitud.



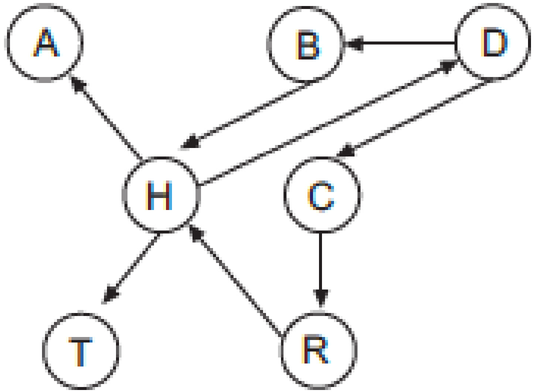
Búsqueda en Amplitud (BFS)

Desde un vértice fuente, exploramos primero todos los vértices a distancia $\boldsymbol{k}$, antes de explorar algún vértice a distancia $\boldsymbol{k+1}$.

Búsqueda en Amplitud (BFS)

```
BFS-B(G,s)
   for all v in V[G] do
      visited[v] := false
   end for
   Q := EmptyQueue
   visited[s] := true
   Enqueue(Q,s)
   while not Empty(Q) do
      u := Dequeue(Q)
      for all w in Adj[u] do
      if not visited[w] then
         visited[w] := true
         Enqueue(Q,w)
      end if
   end while
```

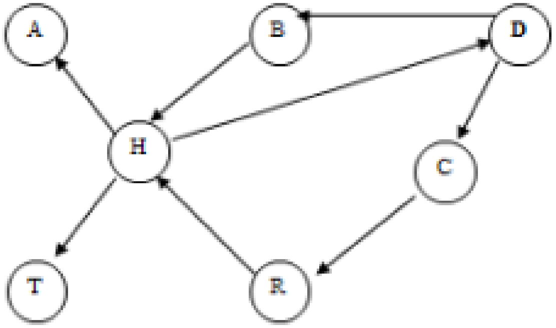
Búsqueda en Amplitud (BFS)



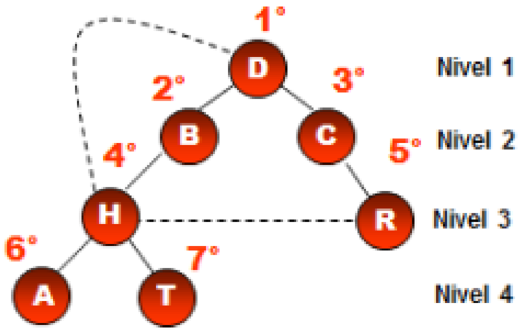
COLA	Vértices procesados
D	
B C	D
C *H*	B
H *R*	C
R *A T*	H
A T	R
T	A
cola vacía	T

Árbol del BFS

Grafo Dirigido



Árbol del BFS



Camino más corto

El BFS adicionalmente calcula los caminos más cortos (distancias) desde el vértice fuente hacia los demás.

Búsqueda en Amplitud (BFS)

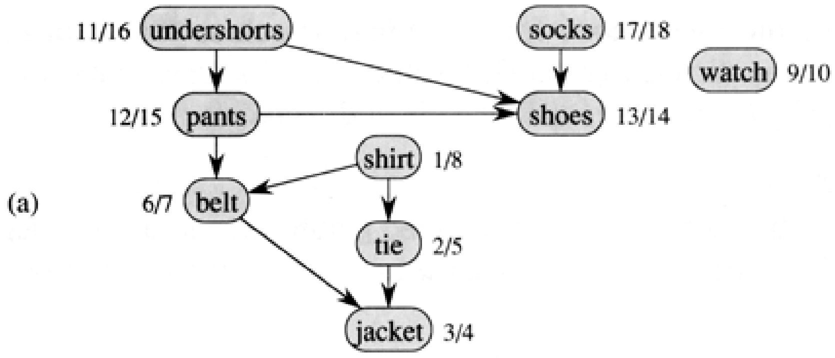
$$O(V + E)$$

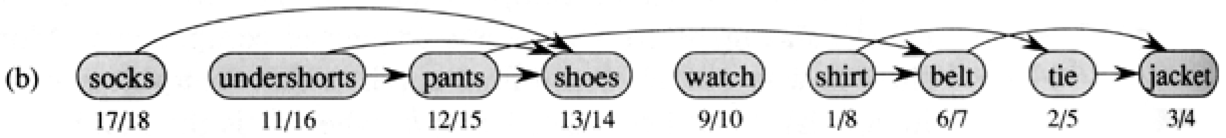
```
const int inf = 10000000
const int MAX_V = 1000000
int d[ MAX_V + 1 ]; // inicializar en inf
vector<int> adj[ MAX_V + 1 ];

void bfs(int s ){
    d[ s ]=0;
    deque<int> Q;
    Q.push_back( s );
    while(!Q.empty()){
        int u = Q.front();
        Q.pop_front();
        for(int i=0; i<adj[ u ].size(); ++i){
            int v =adj[ u ][ i ];
            if(d[ v ] == inf ){
                d[ v ] = d[ u ] + 1;
                Q.push_back( v );
            }
        }
    }
}
```

Ordenamiento Topológico

Un ordenamiento topológico de un DAG G es un ordenamiento lineal de todos sus vértices, tal que si G contiene una arista (u, v) entonces u aparece antes que v en el ordenamiento

Ordenamiento Topológico





Ordenamiento Topológico / Propiedades

- El ordenamiento topológico solo se aplica a un DAG.
- Generalmente la arista (u,v) indica precedencia de eventos.
- Luego de realizar el ordenamiento topológico, todas las aristas van de izquierda a derecha.
- Luego de realizar el ordenamiento topológico, los vértices quedan en orden inverso a su tiempo de finalización.

Ordenamiento Topológico (estilo BFS)

- Para cada nodo de un grafo dirigido:

In – degree : número de aristas que entran a un nodo.
Out – degree : número de aristas que salen de un nodo.
Degree : número de relaciones del nodo (in-degree + out-degree)

- En un grafo no dirigido solo tenemos degree.

Ordenamiento Topológico (estilo BFS)

En un ordenamiento topológico, siempre el primer elemento tiene in-degree igual a 0,

$$O(V + E)$$

```
deque<int> Q;
vector<int> sol;
for( int i = 0; i < MAXV; ++i ) if( inc[i] == 0 ) Q.pb( i );
while( !Q.empty() ){
    int u = Q.front();
    sol.pb( u );
    Q.pop_front();
    for(int i = 0; i < adj[ u ].size(); ++i ){
        int v = adj[ u ][ i ];
        inc[ v ]--;
        if( inc[ v ]==0 ) Q.pb(v);
    }
}
```

BFS 0/1

- Un BFS lo aplicábamos a grafos sin pesos, donde la distancia aumentaba en 1 unidad al procesar los hijos de un nodo.
- Podemos extender el BFS para grafos con aristas de peso 0 y 1.
- En un BFS, en cada instante la cola tiene como máximo nodos de hasta 2 niveles distintos.

BFS 0/1

- Debemos mantener la invariante que “todos los nodos al inicio estén a distancia x y al final con x+1”.
- Si llegamos a un nodo pasando por una arista de peso 1 lo enviamos al final de la cola, caso contrario al inicio.
- Ahora solo debemos tener cuidado ya que podemos llegar primero a uno nodo pasando por una arista con peso 1 y luego al mismo nodo pasando por una arista con peso 0.
- Ahora sí debemos actualizar las distancias.

BFS 0/1

$$O(V+E)$$

```
const int inf = 100000000;
const int MAXV = 100000;
int d[ MAXV ]; //inicializar en inf
bool used[ MAXV ];
vector<int>adj[ MAXV ] , cost[ MAXV ];

void bfs01( int s ){
    for( int i = 0; i < n; ++i ) d[ i ] = INF, used[ i ] = 0;
    d[ s ] = 0;
    deque<int> Q;
    Q.push_back( s );
    while( !Q.empty( ) ){
        int u  = Q.front( );
        Q.pop_front( );
        if( used[ u ] ) continue;
        used[ u ] = 1;
        for( int i = 0; i < adj[ u ].size( ); ++i ){
            int v = adj[ u ][ i ];
            int w = cost[ u ][ i ];
            int temp = d[ u ] + w ;
            if( temp < d[ v ] ){
                d[ v ] = temp;
                if( w == 0 ) Q.push_front( v );
                else Q.push_back( v );
            }
        }
    }
}
```

Problemas

Codechef – Chef and Reversing

UVA - Ordering tasks

Live archive – The Dueling Philosophers Problem

Referencias

- Cormen, Introduction to Algorithms

¡ Good luck and have fun !