

Algoritmos y Estructuras de Datos - 2020

Unidad 6 - Grafos

Tabla de Contenidos

[Tabla de Contenidos](#)

[Descripción](#)

[Introducción](#)

[Definiciones](#)

[Caracterización](#)

[Grafo Simple](#)

[Grafo Conexo](#)

[Grafo Completo](#)

[Grafos Ponderados](#)

[Implementación](#)

[Matriz de Adyacencias](#)

[Listas de Adyacencias](#)

[Ventajas y Desventajas](#)

[Grafos Ponderados](#)

[Operaciones](#)

[Algoritmos](#)

[Búsqueda en anchura](#)

[Búsqueda en profundidad](#)

[Algoritmo de Prim](#)

[Algoritmo de Kruskal](#)

[Ordenación topológica](#)

Descripción

Contenidos Básicos:

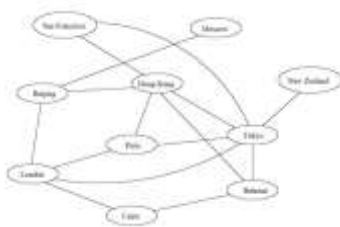
- Definiciones: vértice, grafo, sub-grafo, aristas, ciclos.
- Caracterización de grafos: grafos simples, grafos conexos, grafos completos.

- Operaciones y representaciones.
- Grafos ponderados.
- Implementación.
- Algoritmos: búsqueda en anchura, búsqueda en profundidad, algoritmo de Prim, árbol de cobertura mínimo (Prim, Kruskal), ordenación topológica.

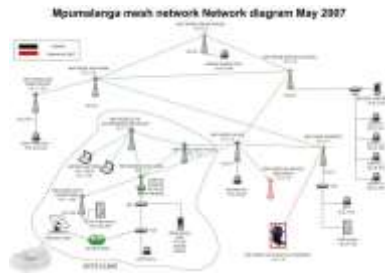
Introducción

Aquí describiremos construcciones utilizadas para representar datos en forma de grafos

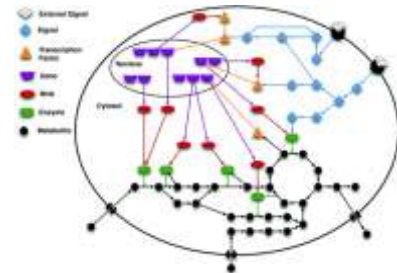
Ejemplos de grafos en la vida real:



Conexiones aéreas



Redes de comunicación



Red de regulación génica

Definiciones

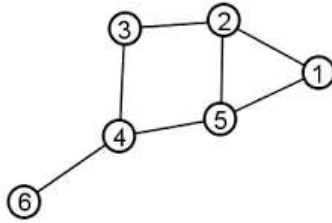
Un **grafo** es una colección de **nodos** (vértices) y **aristas** (edges), donde:

- Cada nodo contiene un valor
- Cada vértice conecta dos nodos (puede ser el mismo)
- Un **grafo dirigido** es un grafo donde los vértices tienen una dirección
- Un **grafo no dirigido** es un grafo donde los vértices no tienen dirección

Formalmente:

- Un grafo es un par ordenado $G(V,E)$ compuesto por
 - Un conjunto de vértices V
 - Un conjunto de aristas E
- En un **grafo dirigido**, una arista es un par ordenado (u,v) donde u y v son Vértices
- En un **grafo no dirigido**, una arista es un conjunto $\{u,v\}$ donde u y v son Vértices

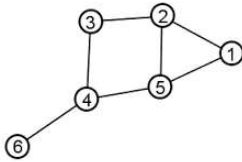
Ejemplo: el grafo definido por los siguientes vértices $V:=\{1,2,3,4,5,6\}$ y aristas $E:=\{\{1,2\},\{1,5\},\{2,3\},\{2,5\},\{3,4\},\{4,5\},\{4,6\}\}$, se puede visualizar como en la imagen abajo:



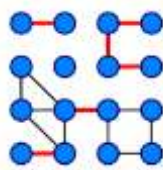
Grafo del ejemplo

Por notación, llamaremos simplemente **grafo** a un grafo no dirigido, y por **grafo dirigido**, o **digrafo**, a un grafo dirigido.

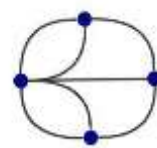
- El **tamaño** de un grafo es su cantidad de nodos



Tamaño = 6

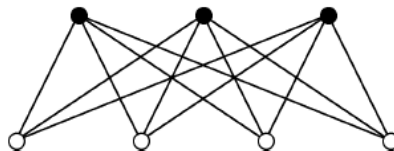


Tamaño = 16



Tamaño = 4

- Un **grafo vacío** es un grafo sin nodos.
- Si dos nodos están conectados por una arista, se dice que son **vecinos**, o **adyacentes**.
- El **grado** de un nodo es su número de aristas.
- Un grafo es **regular** si todos sus nodos tienen el mismo grado.
- Un grafo es **bipartito** si se pueden dividir los nodos en 2 conjuntos A y B, y todas las aristas del grafo tienen un vértice en un conjunto, y el otro vértice en el otro conjunto (no hay aristas de A a A, o de B a B).



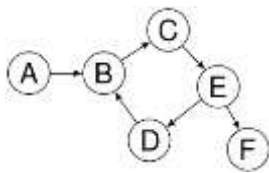
Ejemplo de grafo bipartito, donde A está definido por los nodos negros, y B por los nodos blancos.

En el caso de grafos dirigidos, agregamos las siguientes definiciones:

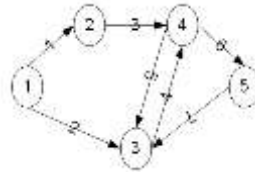
- Para una arista $e = (u, v)$, el vértice u se llama **origen** de la arista, y el vértice v se llama **destino** de la arista.
- Para una arista $e = (u, v)$, se dice que e es una arista **saliente** de u , y una arista **entrante** a v .
- Para una arista $e = (u, v)$, se dice que v es un **sucesor directo** de u , y que u es un **antecesor directo** de v .
- Un **Grafo Orientado** es un grafo dirigido en el que para cada par de nodos/aristas u y v , se cumple una sola de las tres situaciones:
 - a. Los nodos u y v no están conectados

- b. Existe una arista (u,v)
- c. Existe una arista (v,u)

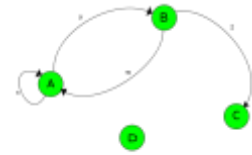
Los grafos orientados NO permiten la existencia de dos aristas, de sentido contrario, conectado los mismos nodos.



Grafo Dirigido y Orientado



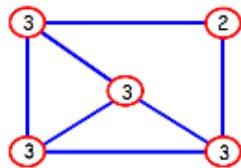
Grafo Dirigido, pero no Orientado (debido a la doble conexión entre los nodos 3 y 4)



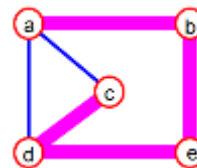
Grafo Dirigido y No Orientado

Más definiciones:

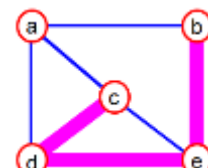
- Un **camino** es una secuencia de vértices $v_1, v_2, v_3, \dots, v_n$ tal que dos vértices consecutivos son adyacentes.



Grafo

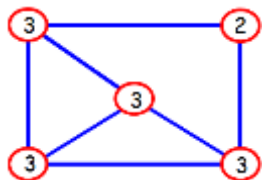


*Camino **abedc***

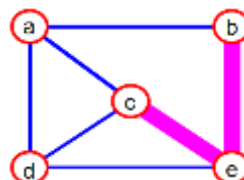


*Camino **bedc***

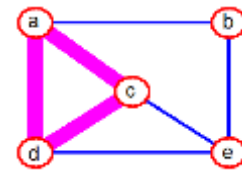
- Un **camino simple** es un camino en el que no se repiten los vértices. Por ejemplo, en el grafo anterior, el camino **abedc** es un camino simple, pero el camino **abedac** no lo es, porque el vértice **a** está dos veces.
- Un **ciclo** es un camino simple (no se repiten vértices), excepto en el que el primer nodo es el mismo que el último nodo (es la única repetición aceptada)



Grafo



Camino simple

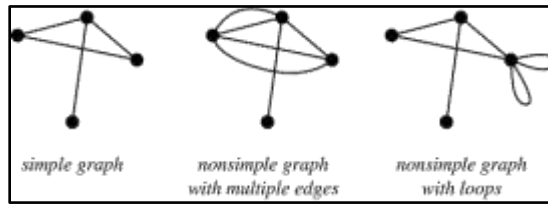


Ciclo

Caracterización

Grafo Simple

Un **grafo simple** es un grafo no dirigido que no posee bucles ni aristas paralelas



Es interesante determinar cuántos grafos simples diferentes (sin contar homomorfismos, que se obtienen por intercambiar los nodos) existen para una cantidad n de nodos. Es importante notar que un grafo simple no precisa ser conectado (ver abajo ejemplo de $n=2$). Algunos valores son:

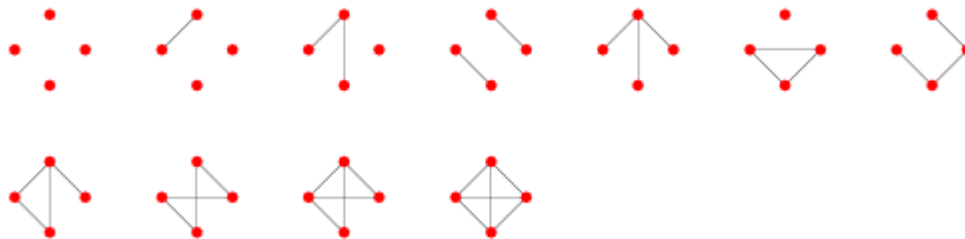
n	1	2	3	4	5	6	7
Cantidad	1	2	4	11	34	156	1044

Para $n=1$, el único grafo simple es aquel con un solo nodo, y sin aristas. También podemos ver todos los grafos simples para $n = 2, 3$ y 4



$n=2$

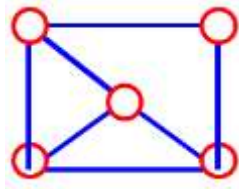
$n=3$



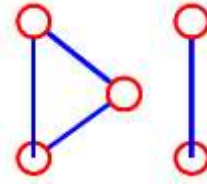
$n=4$

Grafo Conexo

- Un **grafo conexo** es un grafo en el que cualquier par de aristas está conectado por un camino.



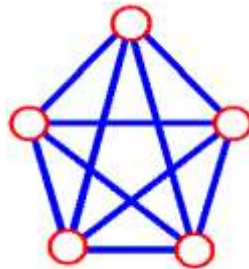
Grafo conexo



Grafo no conexo

Grafo Completo

- Un **grafo completo** es un grafo en el que todo par de vértices (o nodos) **u** y **v** está conectado por una única arista **{u,v}**
- Un **grafo dirigido completo** es un grafo dirigido en el que todo par de vértices **u** y **v** están conectados por dos aristas (**u,v**) y (**v,u**).
- Si **n** es la cantidad de nodos de un grafo, y **m** es su cantidad de aristas, entonces cada nodo de un **grafo completo** tiene grado **n-1**. Está conectado a otros n-1 nodos.
- En ese caso, tenemos $n(n-1)$ aristas (n-1 arista por cada uno de los n nodos). Pero cada arista la contamos dos veces (una por cada nodo que conecta), por lo que la cantidad de aristas es $n(n-1)/2$, para un grafo completo.



$$\begin{aligned} n &= 5 \\ m &= (5 * 4)/2 = 10 \end{aligned}$$

Ejemplo de un grafo completo con 5 nodos y 10 aristas.

Grafos Ponderados

En muchas aplicaciones de los grafos las aristas llevan asociada información adicional. En ese caso hablaremos de grafos etiquetados. Si esa información es numérica y tiene el significado del coste necesario para recorrer esa arista, entonces usaremos el nombre de grafo ponderado o red.

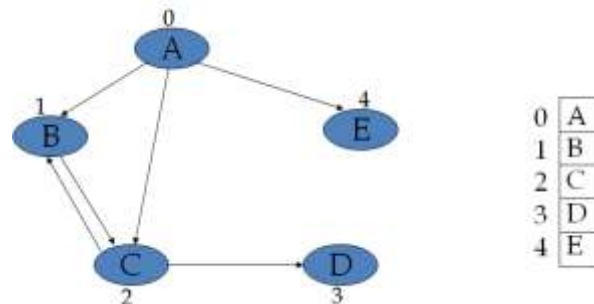
Grafo Ponderado, o Red: grafo en el que cada arista lleva asociado un coste (de aquí en adelante lo llamaremos longitud)

Definiremos la función longitud entre los vértices *i* y *j* de una red como:

$$long(i, j) = \begin{cases} 0 & \text{si } i = j \\ \infty & \text{si } \langle i, j \rangle \notin A \\ \text{coste}(\langle i, j \rangle) & \text{si } \langle i, j \rangle \in A \end{cases}$$

Implementación

Para representar un grafo, hace falta representar los vértices, o nodos, y las aristas. Usualmente, si hay N nodos, se les asigna un número, entre 1 y N, usado como índice del Nodo. Información adicional sobre los nodos se puede guardar en un vector de tamaño N, o una lista dinámica. Abajo se ve un ejemplo de la representación de los vértices por un arreglo.



Ejemplo de un vector de vértices A, B, ..., E, representado por un arreglo

Por el otro lado, la información de las aristas debe indicar que pares de nodos están conectados (y el sentido de la conexión en el caso de grafos dirigidos). Hay dos formas populares para representar la información de las aristas:

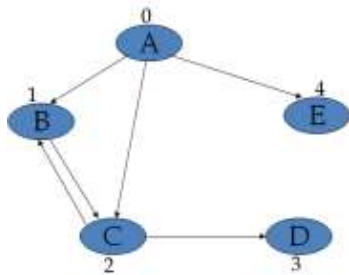
- Matriz de Adyacencias
- Lista de Adyacencias

Matriz de Adyacencias

Una **matriz de adyacencias** es una matriz A de tamaño NxN, donde N es la cantidad de nodos del grafo, y la entrada $A(i,j) = 1$ si los nodos/vértices v_i y v_j son adyacentes, y $A(i,j) = 0$ en caso contrario.

En el caso de nodos dirigidos, $A(i,j) = 1$ si existe una arista (v_i, v_j) , y $A(i,j) = 0$ en caso contrario.

El siguiente ejemplo muestra un grafo dirigido, la representación de sus vértices, y la representación de sus aristas por una matriz de adyacencias. Por ejemplo, existe una arista (A,E), que va del nodo A al nodo E. El nodo A tiene índice 0, y el nodo E tiene índice 4. Por lo tanto debe ser $A(0,4) = 1$. Como no hay una arista (E,A), la entrada $A(4,0) = 0$. En este ejemplo se observa que, para grafos dirigidos, la matriz puede no ser simétrica.



Grafo

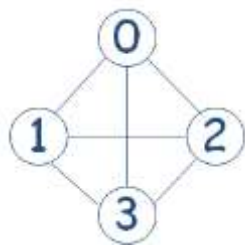
0	A
1	B
2	C
3	D
4	E

Vértices

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

Aristas

En este otro ejemplo se observan dos grafos, uno no dirigido (simétrico), y otro dirigido (no simétrico)



Grafo no dirigido

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Matriz asociada, simétrica



Grafo dirigido

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Matriz asociada, no simétrica

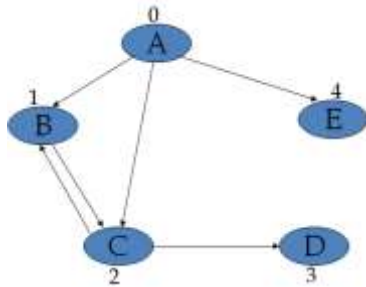
La representación por matriz de adyacencia es simple, y eficiente para algunas operaciones, pero su mayor problema reside en que requiere espacio para cada posible nodo. En una matriz dirigida de tamaño N , se precisa una matriz de tamaño N^2 . Por otro lado, muchos grafos son “no densos”, o sea, la cantidad de aristas es mucho menor que N^2 . Para estos grafos, se está usando mucha más memoria que la necesaria.

Listas de Adyacencias

La solución al problema de una representación más eficiente, con respecto al espacio, de grafos no densos, es la lista de adyacencias para representar las aristas.

Para un grafo de N nodos/vértices, una **Lista de Adyacencias** es un arreglo de N listas. Para cada vértice v_i , la entrada $A[i]$ del arreglo tiene una lista de vértices adyacentes a v_i .

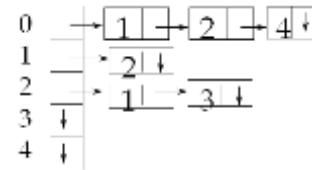
El siguiente ejemplo muestra el mismo grafo dirigido que en el ejemplo anterior, la representación de sus vértices, y la representación de sus aristas por una lista de adyacencias. Por ejemplo, existe una arista (A,E), que va del nodo A al nodo E. El nodo A tiene índice 0, y el nodo E tiene índice 4. Por lo tanto, en la lista A[0] (del nodo A) debe estar el nodo 4, como se ve en el ejemplo. Como los nodos E y D no tienen sucesores, sus listas (3 y 4) están vacías. Como el nodo B (1) tiene un solo sucesor (el Nodo C, etiquetado con 2), si lista A[1] debe tener un solo elemento, el 2.



Grafo

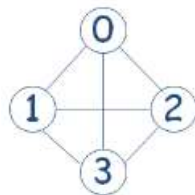
0	A
1	B
2	C
3	D
4	E

Vértices

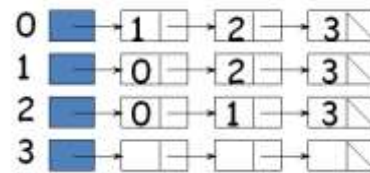


Aristas

Aquí vemos las listas de adyacencias para los dos ejemplos anteriores.



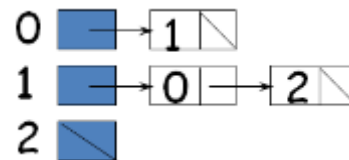
Grafo no dirigido



Lista de Adyacencia



Grafo dirigido



Lista de Adyacencia

Ventajas y Desventajas

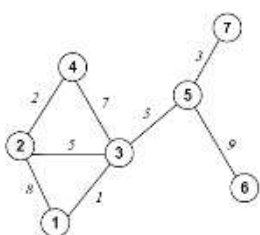
- Las listas de Adyacencias son más compactas que las matrices, si el grafo tiene pocas aristas.
- Las matrices de Adyacencia requieren mucha más memoria, del orden de N^2 .
- Las Listas de Adyacencias requieren más tiempo de búsqueda para determinar si existe una arista entre dos nodos, pues debe buscar en la lista asociada a una de los nodos,
- En cambio, con la matriz de Adyacencias, se realiza la búsqueda en una operación

Grafos Ponderados

En el caso de grafos ponderados, se debe guardar información adicional, de distancia entre nodos.

- Para la matriz de adyacencias, en lugar de 0 o 1, se guarda en $A[i,j]$ el valor de la distancia entre los nodos v_i y v_j (infinito si no están conectados).
- Para la lista de adyacencias, se deben guardar dos valores en cada entrada de las listas: el nodo descendiente, y el valor de la distancia.

En el siguiente ejemplo vemos las dos representaciones para un grafo ponderado:



Grafo

		vértice						
		1	2	3	4	5	6	7
vértice	1	0	8	1	∞	∞	∞	∞
	2	8	0	5	2	∞	∞	∞
	3	1	5	0	7	5	∞	∞
	4	∞	2	7	0	∞	∞	∞
	5	∞	∞	5	∞	0	9	3
	6	∞	∞	∞	∞	9	0	∞
	7	∞	∞	∞	∞	3	∞	0

Matriz de Adyacencias

1	•	→	(2,8)	(3,1)		
2	•	→	(1,8)	(3,5)	(4,2)	
3	•	→	(1,1)	(4,7)	(2,5)	(5,5)
4	•	→	(2,2)	(3,7)		
5	•	→	(3,5)	(7,3)	(6,9)	
6	•	→	(5,9)			
7	•	→	(5,3)			

Lista de Adyacencias

Operaciones

Las operaciones básicas sobre grafos son las de comprobación de existencia de arista entre dos vértices, recorrer la lista de vértices adyacentes a uno dado, la inserción y borrado de una arista, y la inserción y borrado (junto con las aristas asociadas) de un vértice. En muchas aplicaciones, sin embargo, el conjunto de vértices no varía durante la ejecución.

En el caso de las **matrices de adyacencias**, considerando un grafo no dirigido, algunas de estas operaciones son simples:

- Insertar una arista** entre los vértices v_i y v_j : se asigna valor $A[i,j] = 1$
- Borrado de una arista** entre los vértices v_i y v_j : se asigna valor $A[i,j] = 0$
- Detección** de existencia de una arista entre los vértices v_i y v_j : se observa el valor $A[i,j]$. Si es 1, la arista existe. Si es 0, la arista no existe.
- Insertar un vértice** (sin aristas): se agrega una nueva columna y una nueva fila a la matriz, con ceros. Inicialmente no tendrá aristas con ningún otro vértice.
- Eliminar un vértice**: se elimina la fila y columna asociados a dicho vértice. De esta forma, se eliminan automáticamente todas sus aristas.
- Recorrer la lista de vértices adyacentes** al vértice v_i : recorrer la fila $A[i,1], A[i,2], \dots, A[i,N]$. Si un valor $A[i,j] = 1$, entonces el vértice v_j es uno de los vértices adyacentes a v_i .

En el caso de **listas de adyacencia**, las operaciones se implementan de la siguiente manera

- Insertar una arista** entre los vértices v_i y v_j : se agrega una entrada a la lista $A[i]$, con el valor j .

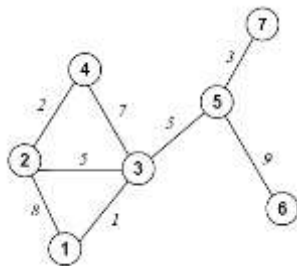
- b) **Borrado de una arista** entre los vértices v_i y v_j : se busca la entrada con valor j de la lista $A[i]$ y se la elimina de la lista.
- c) **Detección** de existencia de una arista entre los vértices v_i y v_j : se busca el valor j en la lista $A[i]$. Si está, la arista existe. Si no está, la arista no existe.
- d) **Insertar un vértice** (sin aristas): se agrega una nueva entrada $A[N+1]$, a la lista de adyacencias, con su lista asociada vacía. Inicialmente no tendrá aristas con ningún otro vértice.
- e) **Eliminar un vértice**: para el vértice v_i , se elimina la entrada $A[i]$ de la lista, y todos los valores i de las otras listas, $A[0]$, $A[1]$, $A[i-1]$, $A[i+1]$, ..., $A[N]$.
- f) **Recorrer la lista de vértices adyacentes** al vértice v_i : recorrer la lista $A[i]$. Si un valor j está en la lista $A[i]$, entonces el vértice v_j es uno de los vértices adyacentes a v_i .

Algoritmos

Los algoritmos de recorrido de grafos buscan un camino que recorre un grafo simple, no dirigido, y conexo, con las siguientes condiciones:

- a) Debe visitar todos los vértices
- b) Debe atravesar todas las aristas

El resultado del recorrido puede materializarse en una secuencia que contendrá todos los vértices del grafo. Por ejemplo, la figura muestra un grafo y algunos posibles recorridos:



Algunos posibles recorridos

- a) 4213576
- b) 5763421
- c) 1234567
- d)

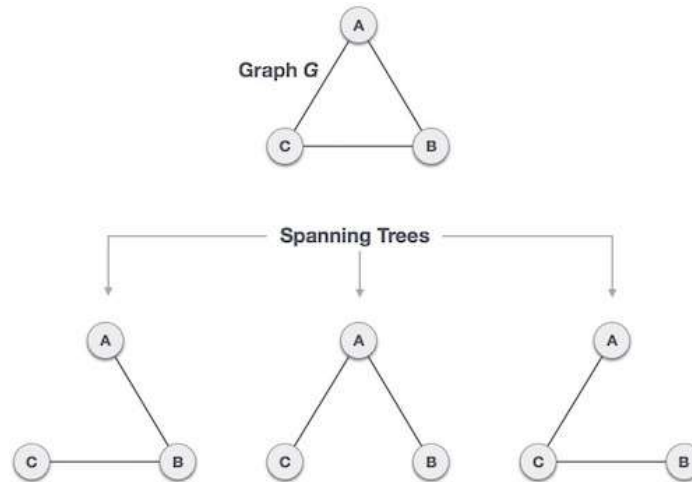
Los vértices y las aristas pueden ser visitados más de una vez, pero se espera que el recorrido presente una sola vez cada vértice. El recorrido del grafo puede ser útil, por ejemplo,

- a) Para mostrar la información de los vértices, uno por uno, sin repetición
- b) Para conocer el camino entre dos vértices.
- c) Para generar un Árbol de Cobertura

Un **árbol de cobertura** es un árbol que contiene todos los vértices, y cuyas aristas son aristas del grafo. Dichos árboles son útiles para varias aplicaciones. Por ejemplo, algoritmos de búsqueda de camino óptimo (a ver más adelante) utilizan un árbol de cobertura como paso intermedio. Un listado de posibles aplicaciones incluye:

- a) **Diseño de redes** - telefónicas, eléctricas, hidráulicas, Televisión, Informática, etc.

- b) **Soluciones de problemas NP** - problema del viajante, otros. (obs: NP, no se puede encontrar una solución en tiempo polinomial)
- c) **Clustering** - Se puede encontrar una partición de un conjunto generando su árbol de cobertura (donde las aristas corresponden a distancias, por ejemplo) y removiendo nodos en función de sus costos.



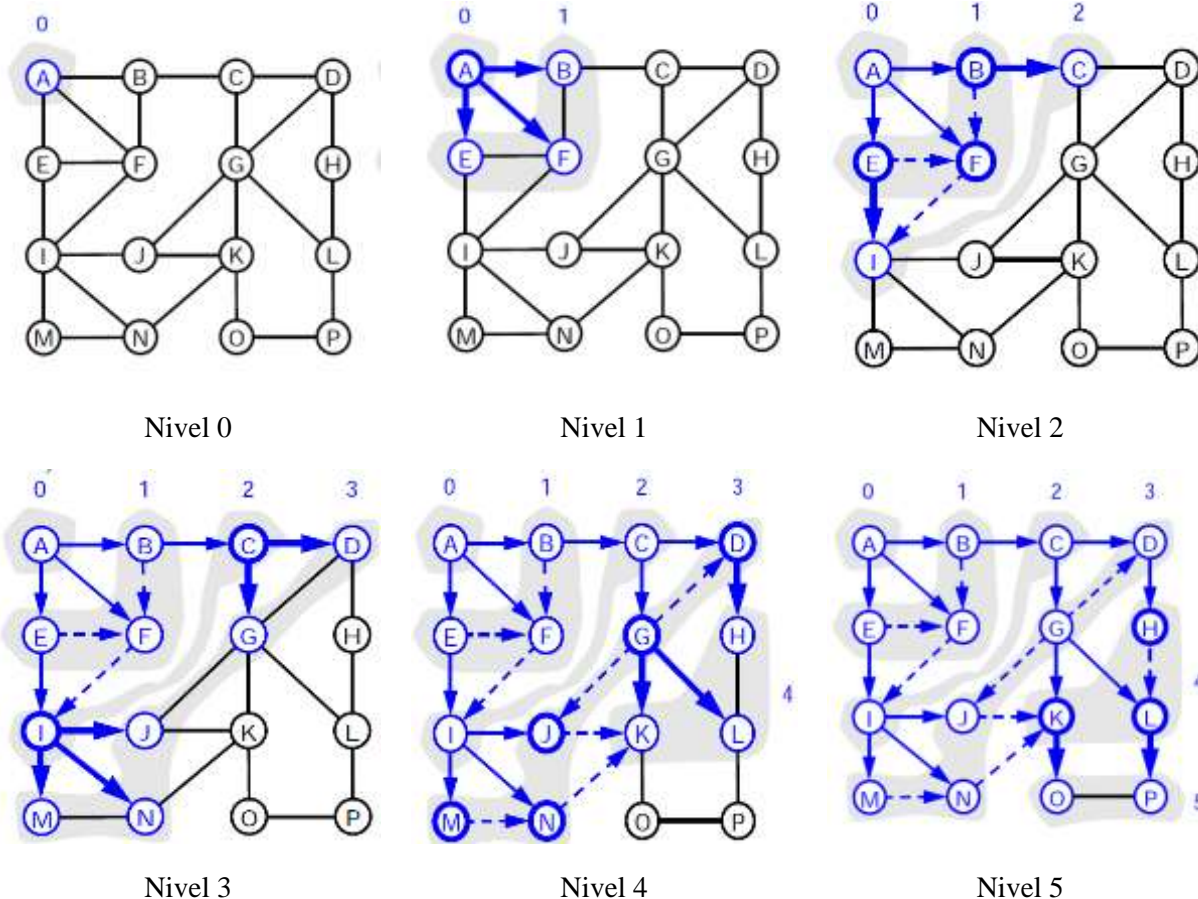
Ejemplo de tres diferentes árboles de cobertura para un grafo conexo no dirigido.

Búsqueda en anchura

En el caso de **búsqueda en anchura**, el proceso recorre el grafo de tal manera de visitar todos los vértices y aristas, y genera un árbol que cubre todos los vértices. La idea básica detrás de esta búsqueda es visitar primero un nodo, después todos sus adyacentes, después todos los adyacentes de los adyacentes, y continuar así hasta haber recorrido todo el grafo.

El procedimiento es el siguiente:

- a) Comenzar de un vértice S. Marcarlo como raíz del árbol. Asignar nivel 0 a S.
- b) En un primer paso, visitar y asignarles nivel 1 a todos los vértices adyacentes a S.
- c) En un segundo paso, visitar todos los vértices adyacentes a los vértices no nivel 1, que no hayan sido visitados antes. Asignarles nivel 2.
- d) Continuar hasta que todos los vértices fueran visitados.



El algoritmo usa una fila para guardar los vértices que va a recorrer. Para cada vértice que visita, guarda en la fila todos sus vecinos que no fueron visitados. Luego extrae de la fila un vértice, y lo visita, realizando el mismo procedimiento. Cuando no hay más elementos en la fila, es porque se ha recorrido el grafo completamente.

Con el proceso anterior, nos aseguramos de visitar todos los vértices una sola vez. El pseudocódigo de dicho algoritmo se describe a continuación:

```

Struct Fila* F;
Visitado[0] = verdadero;
FilaEnqueue(F,0)
Mientras ( NO FilaIsEmpty(F) ) {
    v = FilaDequeue(F);
    Para cada successor v' of v {
        Si Visitado[v'] ==falso {
            Fila.Enqueue(v');
            Visitado[v'] = verdadero;
        }
    }
}

```

Con dicho algoritmo podemos recorrer completamente el grafo. Por otro lado, el algoritmo puede requerir construir un árbol de cobertura. Para ello, cada vez que visita un vértice 'v' y agrega un vértice adyacente v' a la Fila, puede insertarlo como nodo hijo de v en el árbol. El algoritmo quedaría:

```

Struct Fila* F;
Struct Tree* T;
Visitado[0] = verdadero;
T = ArbolAgregaHijo(T,0);
FilaEnqueue(F,0)
Mientras ( NO FilaIsEmpty(F) ) {
    v = FilaDequeue(F);
    Para cada successor v' of v {
        Si Visitado[v'] ==falso {
            T = ArbolAgregaHijo(T,v,v');
            Fila.Enqueue(v');
            Visitado[v'] = verdaderp
        }
    }
}

```

Si lo que buscamos es el camino más corto entre un vértice s y otro vértice v , podemos agregar una información de “distancia” a cada nodo. Comenzamos el recorrido en s , asignandole distancia 0. Recorremos el grafo y aumentamos la distancia en $d+1$ a los hijos de un nodo con distancia d . Cuando llegamos a v , su distancia es la distancia a s .

El siguiente pseudocódigo muestra cómo obtener la distancia entre todos los vértices y un vértice s .

```

Struct Fila* F;
Visitado[s] = verdadero;
Distancia[s] = 0;
FilaEnqueue(F,s)
Mientras ( NO FilaIsEmpty(F) ) {
    v = FilaDequeue(F);
    Para cada successor v' of v {
        Si Visitado[v'] ==falso {
            Fila.Enqueue(v');
            Distancia[v'] = Distancia[v] + 1;
            Visitado[v'] = verdaderp
        }
    }
}

```

Búsqueda en profundidad

A diferencia de la búsqueda en anchura, que visita un vértice, luego todos sus vértices adyacentes, luego los adyacentes de estos, la búsqueda en altura profundiza cada adyacente hasta encontrar el final (no más adyacentes no visitados), con lo cual recién vuelve a visitar los siguientes nodos adyacentes.

El algoritmo de recorrido en profundidad, en inglés **depth-first search** y que denotaremos a partir de ahora DFS para abreviar, permite efectuar un recorrido sistemático del grafo

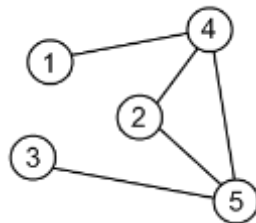
El orden de visita es tal que los vértices se recorren según el orden 'primero en profundidad', lo que significa que dado un vértice v que no haya sido visitado, la primera vez que el DFS lo alcanza, DFS lo visita y, a continuación, aplica DFS recursivamente sobre cada uno de los adyacentes/sucesores de v que aún no hayan sido visitados.

Para poner en marcha el recorrido se elige un vértice cualquiera como punto de partida y el algoritmo acaba cuando todos los vértices han sido visitados. Se puede establecer un claro paralelismo entre el recorrido en preorden de un árbol y el DFS sobre un grafo.

Algoritmo:

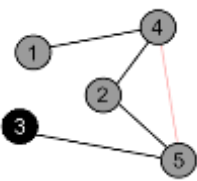
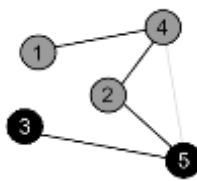
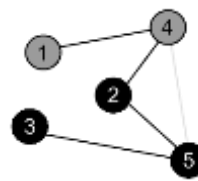
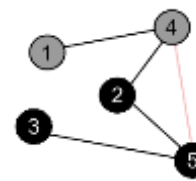
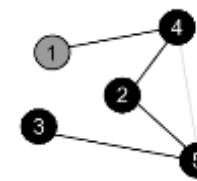
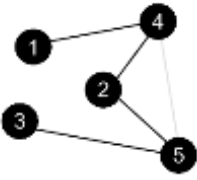
- a) Cada vértice tiene 3 colores posibles:
 - i) Blanco (No visitado)
 - ii) Gris (Visitado, en progreso)
 - iii) Negro (Se terminó de procesar)
- b) Se comienza con todos los vértices en blanco (no visitados)
- c) Se marca un vértice u como en gris (visitado)
- d) Para cada arista (u,v) , donde v es blanco, se aplica recursivamente el algoritmo DFS a v
- e) Se marca u con negro (terminado) y se vuelve al pariente

Como ejemplo, recorreremos el siguiente grafo a partir del nodo 1.



El cuadro abajo muestra todos los pasos del algoritmo DFS:

<p>Marco 1 con gris</p>	<p>Hay una arista (1,4) y 4 no fue visitado. Ir a 4</p>	<p>Marco 4 con gris</p>	<p>Hay una arista (4,2) y 2 no fue visitado. Ir a 2.</p>	<p>Marca 2 con gris</p>
<p>Hay una arista (2,5) y 5 no fue visitado. Ir a 5.</p>	<p>Marcar 5 con gris</p>	<p>Hay una arista (5,3) y 3 no fue visitado. Ir a 3.</p>	<p>Marcar 3 con gris</p>	<p>No hay aristas libres desde 3. Marcar 3 con negro. Volver a 5</p>

 <p>Hay una arista (5,4) pero 4 es gris.</p>	 <p>No hay más aristas desde 5. Marcar 5 con negro. Volver a 2.</p>	 <p>No hay más aristas desde 2 para visitar. Marcar con negro. Volver a 4.</p>	 <p>4 tiene otra arista (4,5) pero 5 es negro. No hacer nada.</p>	 <p>No hay más aristas. Marcar 4 con negro.</p>
 <p>1 no tiene más aristas. Marcar 1 con negro. Terminar.</p>				

El mismo algoritmo se puede aplicar sin utilizar colores, simplemente indicando si un vértice ya fue visitado.

Algoritmo:

- Se comienza con todos los vértices como no visitados
- Se marca un vértice u como visitado
- Para cada arista (u,v) , donde v es NO visitado, se aplica recursivamente el algoritmo DSF a v
- Cuando no hay más aristas se vuelve al pariente

El pseudocódigo es el siguiente:

```

n ← number of nodes
Initialize visited[ ] to false (0)
for(i=0;i<n;i++)
    visited[i] = 0;

void DFS(vertex i) [DFS starting from i]
{
    visited[i]=1;
    for each w adjacent to i
        if(!visited[w])
            DFS(w);
}

```


Algoritmo de Prim

Robert Prim en 1957 descubrió un algoritmo para la resolución del problema del **Árbol de coste total mínimo** (*minimumspanningtree* - *MST*). Este problema es un problema típico de optimización combinatoria.

El algoritmo de Prim encuentra un árbol de peso total mínimo conectando nodos o vértices con arcos de peso mínimo del grafo sin formar ciclos.

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

Objetivo de Algoritmo prim

- Encontrar el árbol recubrido más corto

Requisitos

- — Ser un grafo conexo
- — Ser un grafo sin ciclos
- — Tener todos los arcos etiquetados.

En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

La **idea básica** consiste en añadir, en cada paso, una arista de peso mínimo a un árbol previamente construido. Más explícitamente:

Paso 1. Se elige un vértice u de G y se considera el árbol $S=\{u\}$

Paso 2. Se considera la arista e de mínimo peso que une un vértice de S y un vértice que no es de S , y se hace $S=S+e$

Paso 3. Si el nº de aristas de T es $n-1$ el algoritmo termina. En caso contrario se vuelve al paso 2

Pseudocódigo

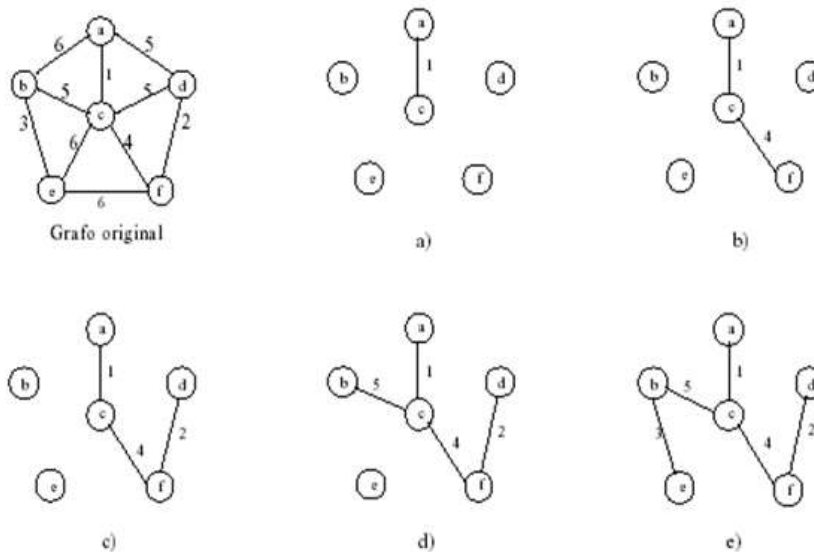
```

void Prim( GRAFO G, CONJUNTO T )
{
    CONJUNTO_V U;
    VERTICE    v;

    T =  $\emptyset$ ;
    U = { cualquier v3rtice de G };
    while( U != V ) {
        Sea (u,v) es el arco de menor costo tal que
            u  $\in$  U y v  $\in$  V-U;
        T = T  $\cup$  { (u,v) };
        U = U  $\cup$  { v };
    }
}

```

Ejemplo



Algoritmo de Kruskal

Joseph B. Kruskal investigador del Math Center (Bell-Labs), que en 1956 descubrió su algoritmo para la resolución del problema del Árbol de coste total mínimo

El objetivo del algoritmo de Kruskal es construir un árbol (subgrafo sin ciclos) formado por arcos sucesivamente seleccionados de mínimo peso a partir de un grafo con pesos en los arcos.

El Algoritmo de Kruskal que resuelve la misma clase de problema que el de Prim, salvo que en esta ocasión no partimos desde ningún nodo elegido al azar.

Para resolver el mismo problema lo que hacemos es pasarle a la función una lista con las aristas ordenada de menor a mayor, e iremos tomando una para formar el ARM.

En un principio cada nodo está en un digamos grupo distinto, al elegir una arista de la lista miraremos si no están los nodos conectados ya en el mismo grupo, de no estarlo fusionamos ambos grupos y comprobamos si hemos encontrado ya la solución, para devolver el resultado.

- Sea G un grafo con m nodos y e aristas, donde cada arista tiene un peso $W(e)$ asignado.
- Para todo grafo conectado G es posible asociar un árbol que contenga todos los vértices del grafo, denominado árbol de cobertura.
- Todo árbol de cobertura que a su vez tiene la sumatoria mínima de los valores de las aristas se denomina árbol de cobertura mínima.
- El algoritmo de Kruskal nos permite encontrar el árbol de cobertura mínima, para un grafo cuyas aristas han sido ordenadas de la forma:

$$W(e_1) \leq W(e_2) \leq \dots \leq W(e_m)$$

Pseudocódigo

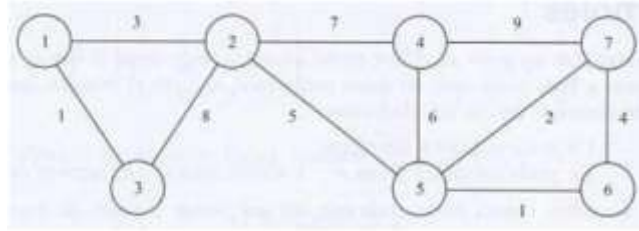
```
void Kruskal( GRAFO G, CONJUNTO T )
{
    CONJUNTO_V U;
    VERTICE v;

    T = Ø;
    for( cada vértice v de G )
        construye un árbol con v;
    Ordena los arcos de G en orden no decreciente;
    while( Haya más de un árbol ) {
        Sea (u,v) es el arco de menor costo tal que el árbol
        de u es diferente al árbol de v;
        Mezcla los árboles de u y de v en uno solo;
        T = T ∪ { (u,v) };
    }
}
```

El algoritmo de Kruskal permite hallar el árbol minimal de cualquier grafo valorado (con capacidades). Hay que seguir los siguientes pasos:

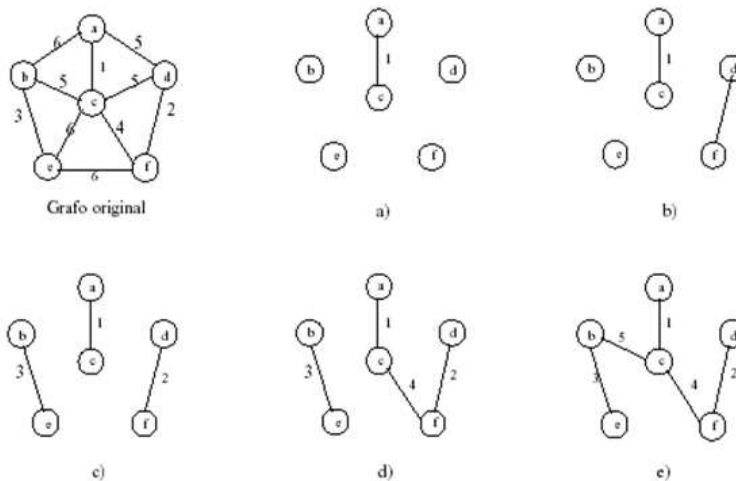
1. Se marca la arista con menor valor. Si hay más de una, se elige cualquiera de ellas.
2. De las aristas restantes, se marca la que tenga menor valor, si hay más de una, se elige cualquiera de ellas.
3. Repetir el paso 2 siempre que la arista elegida no forme un ciclo con las ya marcadas.
4. El proceso termina cuando tenemos todos los nodos del grafo en alguna de las aristas marcadas, es decir, cuando tenemos marcados $n-1$ arcos, siendo n el número de nodos del grafo.

Ejemplo: Determinar el árbol de mínima expansión para el siguiente grafo



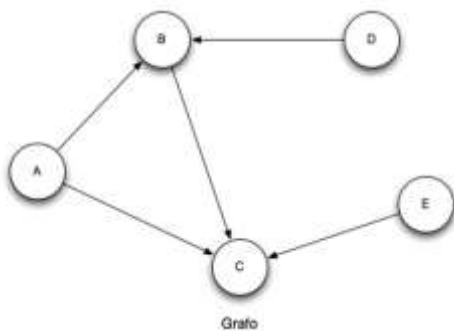
- Siguiendo el algoritmo de Kruskal, tenemos:
 - Elegimos, por ejemplo, la arista $(5, 6) = 1$ (menor valor) y la marcamos.
 - Elegimos la siguiente arista con menor valor $(1, 3) = 1$ y la marcamos.
 - Elegimos la siguiente arista con menor valor $(5, 7) = 2$ y la marcamos, ya que no forma ciclos con ninguna arista de las marcadas anteriormente.
 - Elegimos la siguiente arista con menor valor $(1, 2) = 3$ y la marcamos, ya que no forma ciclos con ninguna arista de las marcadas anteriormente.
 - Elegimos la siguiente arista con menor valor $(6, 7) = 4$ y la desechamos, ya que forma ciclos con las aristas $(5, 7)$ y $(5, 6)$ marcadas anteriormente.
 - Elegimos la siguiente arista con menor valor $(2, 5) = 5$ y la marcamos, ya que no forma ciclos con ninguna arista de las marcadas anteriormente.
 - Elegimos la siguiente arista con menor valor $(4, 5) = 6$ y la marcamos, ya que no forma ciclos con ninguna arista de las marcadas anteriormente.
 - FIN. Finalizamos dado que los 7 nodos del grafo están en alguna de las aristas, o también ya que tenemos marcadas 6 aristas $(n-1)$.
 - Por tanto el árbol de mínima expansión resultante sería:

Otro Ejemplo

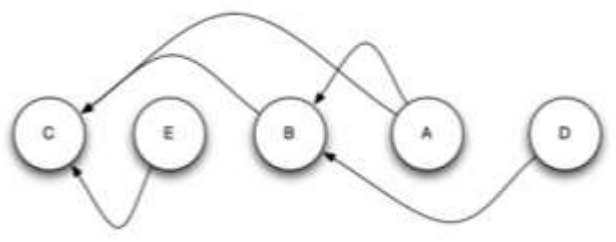


Ordenación topológica

Un Orden Topológico ordena los nodos de un grafo dirigido acíclico de forma que si hay una camino del nodo A al nodo B entonces A aparece antes que B en la ordenación.



Grafo Dirigido



Ordenado Topologicamente

El sentido de las flechas indica las dependencias. A depende de B y C, B depende de C, etc, etc
En pocas palabras no se visitara un vértice, hasta que todos sus predecesores hayan sido visitados.

Aplicaciones

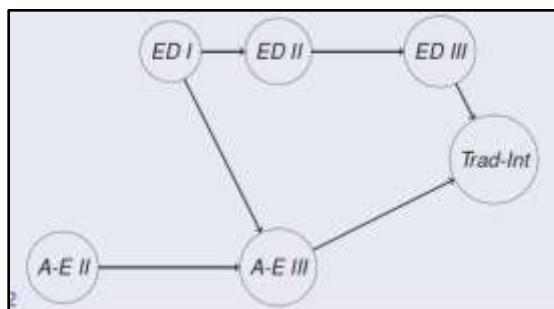
- Puede ser aplicable a la representación de las fases de un proyecto , donde los vértices representen tareas, y las aristas relaciones temporales a ellas.
- Evaluación en la fase semántica de un compilador.
- También para encontrar algún problema o defecto de cierto algoritmo que se ordene topológicamente.

Algunas definiciones

- Para un digrafo acíclico (dag) $G = (N, A)$ el orden lineal de todos los nodos tal que si G contiene el arco (u, v) , entonces u aparece antes de v en el orden dado.
- Si G tiene ciclos el orden lineal no es posible.
- Ordenamiento topológico de todos los nodos de un digrafo es una manera de visitar todos sus nodos, uno por uno, en una secuencia que satisface una restricción dada.
- Un orden topológico de los nodos de un digrafo $G = \{N, A\}$ es una secuencia (n_1, n_2, \dots, n_k) tal que $N = \{n_1, n_2, \dots, n_k\}$ y para todo (n_i, n_j) en A , n_i precede a n_j en la secuencia.

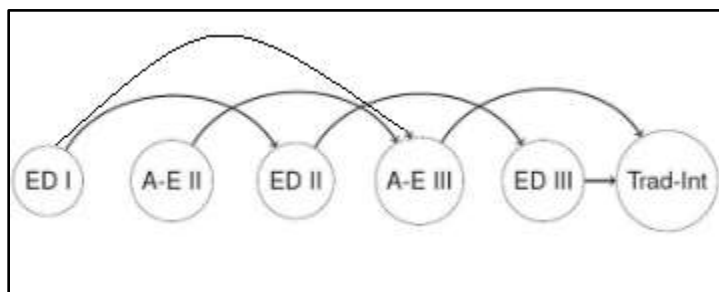
Ejemplo

Teniendo una gráfica acíclica dirigida (DAG, por sus siglas en inglés), nos interesa encontrar una forma en que podamos acomodar los vértices en línea recta de tal forma que la dirección de los arcos siempre apunte hacia un mismo lado. Un ejemplo de cuando se dan este tipo de gráficas, es en las materias de estudios superiores. Se tienen que tomar algunas materias antes que otras (materias seriadas), pero nunca se puede formar un ciclo (o no se podrían estudiar estas materias).



Ejemplo de relación entre materias

Como no hay forma en que nos podamos regresar, podemos ver a este tipo de gráfica en forma similar a un árbol. Podemos utilizar cualquier forma de recorrer un árbol para encontrar el ordenamiento topológico, pero es más conveniente utilizar una búsqueda en profundidad ya que es más fácil de implementar. El orden Topológico Seria:



Referencias

- [1] <http://mathworld.wolfram.com/SimpleGraph.html> Grafo simple
- [2] <http://algorithmics.lsi.upc.edu/docs/ada/MTA/grafos.pdf> - Algoritmos
- [3] <https://www.topcoder.com/community/data-science/data-science-tutorials/introduction-to-graphs-and-their-data-structures-section-1/> - Implementación
- [4] <http://www.cs.cornell.edu/courses/cs2112/2012sp/lectures/lec24/lec24-12sp.html> - Graph Traversal
- [5] http://opendatastructures.org/ods-java/12_3_Graph_Traversal.html - Graph Traversal Java
- [6] http://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm - Spanning trees.
- [7] <http://hansolav.net/sql/graphs.html>
- [8] <http://www.geeksforgeeks.org/applications-of-minimum-spanning-tree/> - Minimum Spanning Tree
- [9] http://www.algolist.net/Algorithms/Graph/Undirected/Depth-first_search - DFS
- [10] <http://www.thecrazyprogrammer.com/2014/03/depth-first-search-dfs-traversal-of-a-graph.html> - DFS en C
- [11] <https://fher9696.files.wordpress.com/2010/08/algoritmos.docx>
- [12] <https://ordenacion-topologica-grafos.wikispaces.com/4+Ordenaci%C3%B3n+topol%C3%B3gica> - Ordenacion topologica