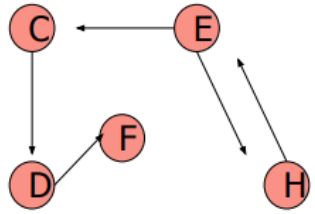
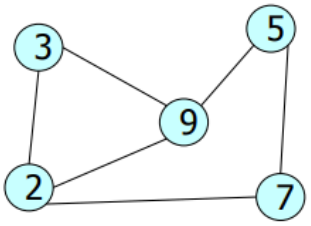
Grafo: (V,E), V es un conjunto de vértices o **nodos**, con una relación entre ellos; E es un **conjunto de pares (u,v)**, u,v Є V , llamados aristas o arco

Grafo dirigido: cada arista tiene una dirección. Esto significa que una arista conecta un nodo inicial con un nodo final, formando un par ordenado.

Grafo no dirigido: cada arista conecta dos nodos sin una dirección específica. A diferencia de los grafos dirigidos, las conexiones no tienen un sentido, lo que significa que si hay una arista entre los nodos u y v, se puede ir tanto de u a v como de v a u. (como se puede ir de u a v y de v a u, solo graficar las aristas una vez, no repetir por ejemplo de 2 a 3 y de 3 a 2)

Ejemplos





Grafo dirigido G(V,E).

V = {C,D,E,F,H}

E={(C,D),(D,F),(E,C),(E,H),(H,E)}

Grafo no dirigido G(V,E).

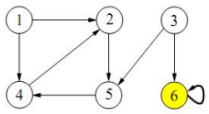
V = {2,3,5,7,9}

E= {{2,3},{2,7},{2,9},{3,9},{5,7},{5,9}}

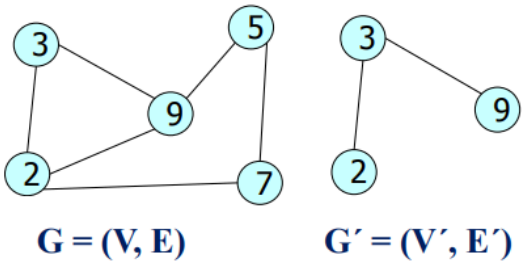
En grafos no dirigidos: el grado de un nodo: número de arcos que inciden en él.
En grafos dirigidos: existen el grado de salida (grado_out) y el grado de entrada (grado_in). el grado_out es el número de arcos que parten de él y el grado_in es el número de arcos que inciden en él.

- El grado del vértice será la suma de los grados de entrada y de salida.

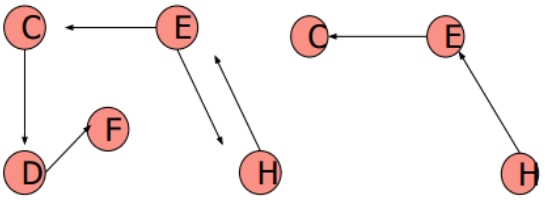
Grado de un grafo: máximo grado de sus vértices.
Camino: desde u Є V a v Є .
Longitud de un camino: número de arcos del camino.
Camino simple: camino en el que todos sus vértices, excepto, tal vez, el primero y el último, son distintos.
Ciclo: camino desde v1 , v2 , ... , vk tal que v1=vk.
Bucle: ciclo de longitud 1.



Grafo acíclico: grafo sin ciclos.
Dado un grafo G=(V, E), se dice que G´=(V´, E´) es un subgrafo de G, si V'⊆ V y E' ⊆ E



G = (V, E) G´ = (V´, E´)



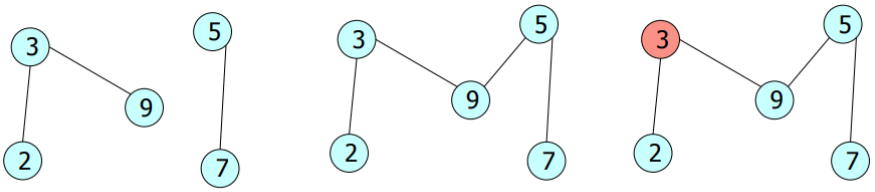
G = (V, E) G´ = (V´, E´)

Un grafo ponderado, pesado o con costos: cada arco o arista tiene asociado un valor o etiqueta.

CONECTIVIDAD EN GRAFOS NO DIRIGIDOS:
Un grafo no dirigido: es conexo si hay un camino entre cada par de vértices

Conexo No Conexo

- Un bosque es un grafo sin ciclos.
- Un árbol libre es un bosque conexo.
- Un árbol es un árbol libre en el que un nodo se ha designado como raíz.



Bosque Árbol libre Árbol

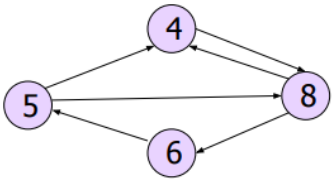
PROPIEDADES:
Sea G un grafo no dirigido con n vértices y m arcos, entonces:

✔*Siempre:* $m \leq (n*(n-1))/2$

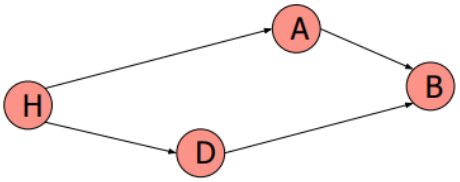
✔*Si G conexo:* $m \geq n-1$
✔*Si G árbol:* $m = n-1$
✔*Si G bosque:* $m \leq n-1$

CONECTIVIDAD EN GRAFOS DIRIGIDOS:

- v es alcanzable desde u, si existe un camino de u a v.
- Un grafo dirigido se denomina fuertemente conexo si existe un camino desde cualquier vértice a cualquier otro vértice.



Fuertemente Conexo



No Fuertemente Conexo
Débilmente Conexo

Si un grafo dirigido no es fuertemente conexo, pero el grafo subyacente (sin sentido en los arcos) es conexo, el grafo es débilmente conexo (como sucede en el nodo H).

COMPONENTES CONEXAS:

En un grafo no dirigido, una componente conexa es un subgrafo conexo tal que **no existe otra componente conexa que lo contenga. (esa propiedad hace que sea un subgrafo conexo maximal)**

Componentes fuertemente conexas:

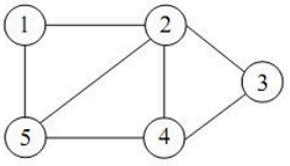
En un grafo dirigido, una componente fuertemente conexa, es el máximo subgrafo fuertemente conexo. Un grafo dirigido es no fuertemente conexo si está formado por varias componentes fuertemente conexas.

REPRESENTACIONES: Matriz de adyacencias y Lista de adyacencias.

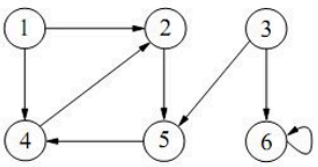
Matriz de adyacencias:

- $G = (V, E)$: matriz A de dimensión $|V| \times |V|$.
- Valor a_{ij} de la matriz:

$$a_{ij} = \begin{cases} 1 & \text{si } (i,j) \in E \\ 0 & \text{en cualquier otro caso} \end{cases}$$



	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



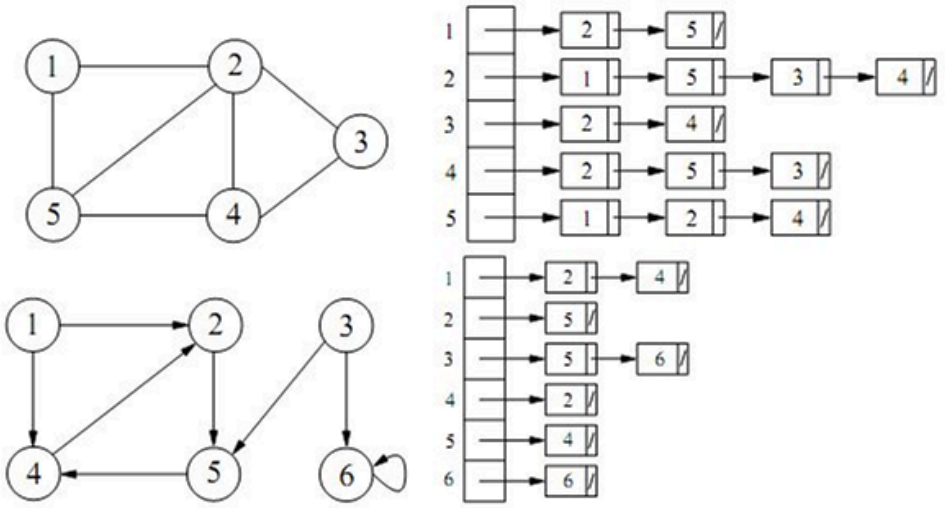
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

- La matriz de adyacencia es útil para grafos con número de vértices pequeño y es fácil comprobar si una arista (u,v) pertenece a E, consultando por su posición en la matriz A(u,v).
- Tendrá un costo de tiempo igual a T(|V|,|E|) = O(1).
- Cuando se trata de grafos pesados/ponderados, el peso de (i,j) se almacena en A(i,j)

Lista de adyacencias:

G =(V ,E): vector de tamaño |V|.
Posición i → puntero a una lista enlazada de elementos (lista de adyacencia).

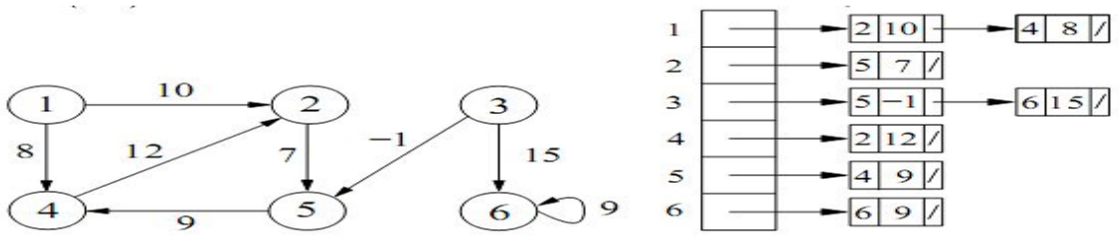
*Los elementos de la lista son los vértices adyacentes a **i***



- Si G es dirigido, la suma de las longitudes de las listas de adyacencia será |E|. (la cantidad total de aristas/conjuntos que tendrá).
- Si G es no dirigido, dado que cada arista se cuenta dos veces, una vez para cada uno de los dos vértices que conecta, la suma total de las longitudes de las listas de adyacencia será el doble del número de aristas, es decir, 2|E|)
- Costo espacial, sea dirigido o no: O (|V|+|E|) .
- Desventaja: si se quiere comprobar si una arista (u,v) pertenece a E ⇒ buscar v en la lista de adyacencia de u.

Representación aplicada a Grafos pesados:

El peso de (u,v) se almacena en el nodo de v de la lista de adyacencia de u.



RECORRIDOS DFS Y BFS

DFS: (mediante listas de adyacencia)

Estrategia:

- Partir de un vértice determinado v.
- Cuando se visita un nuevo vértice, explorar cada camino que sale de él.
- Hasta que no se haya finalizado de explorar uno de los caminos no se comienza con el siguiente.
- Un camino deja de explorarse cuando se llega a un vértice ya visitado.
- Si existían vértices no alcanzables desde v el recorrido queda incompleto; entonces, se debe seleccionar algún vértice como nuevo vértice de partida, y repetir el proceso.

Esquema recursivo:

1. Marcar todos los vértices como no visitados. (se suele almacenar en un vector de booleanos)
2. Elegir vértice u como punto de partida.
3. Marcar u como visitado.
4. ∀ v adyacente a u,(u,v) Є E, si v no ha sido visitado, repetir recursivamente (3) y (4) para v.

- Finalizar cuando se hayan visitado todos los nodos alcanzables desde u.
- Si desde u no fueran alcanzables todos los nodos del grafo: volver a (2), elegir un nuevo vértice de partida v no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

BFS:

Estrategia:

- Partir de algún vértice u, visitar u y, después, visitar cada uno de los vértices adyacentes a u.
- Repetir el proceso para cada nodo adyacente a u, siguiendo el orden en que fueron visitados.

Esquema iterativo:

1. Encolar el vértice origen u.
2. Marcar el vértice u como visitado.
3. Procesar la cola.
4. Desencolar u de la cola
5. ∀ adyacente a u,(u,v) Є E,
6. si v no ha sido visitado
7. encolar y visitar v

- Si desde u no fueran alcanzables todos los nodos del grafo: volver a (1), elegir un nuevo vértice de partida no visitado, y repetir el proceso hasta que se hayan recorrido todos los vértices.

BOSQUE DE EXPANSIÓN DEL DFS Y BFS VERLO EN EL PDF

ORDENACIÓN TOPOLÓGICA

- La ordenación topológica es una permutación: v1 , v2 , v3 , ..., v|V| de los vértices, tal que si (vi ,vj) Є E, vi ≠ vj , entonces vi precede a vj en la permutación.
- La ordenación no es posible si G es cíclico.
- La ordenación topológica no es única.
- Una ordenación topológica es como una ordenación de los vértices a lo largo de una línea horizontal, con los arcos de izquierda a derecha.

Aplicaciones:

- Para indicar la precedencia entre eventos .
- Para planificación de tareas.
- Organización curricular.

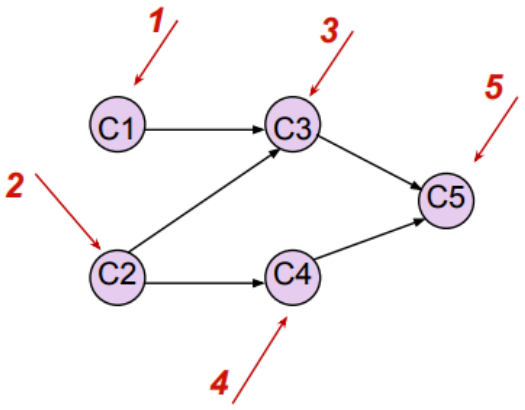
La ordenación topológica se refiere a la secuencia lineal de los nodos (o vértices) de un grafo de manera que para cada arista

dirigida (u,v), el nodo u aparece antes que el nodo v en la secuencia.

Versión 1:

Grado_in

C1	C2	C3	C4	C5
0	0	2	1	2
0	**0**	**1**	1	2
0	0	**0**	**0**	2
0	0	0	**0**	**1**
0	0	0	0	**0**



Sort Topológico :

C1 C2 C3 C4 C5

Pasos:

1. Calcular el grado de entrada (in-degree) de cada nodo.
2. Insertar todos los nodos con grado de entrada 0 en una cola.
3. Repetidamente extraer un nodo de la cola, añadirlo a la ordenación topológica, y reducir el grado de entrada de sus nodos adyacentes.
4. Si al final del proceso todos los nodos han sido añadidos a la ordenación, el grafo es un DAG. Si no, contiene ciclos.

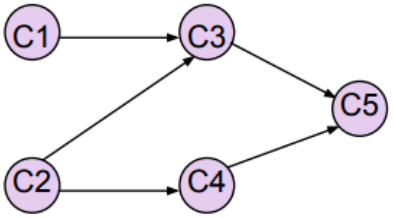
Versión 2 (Pila):

Una pila (se saca el último que nodo que se agregó) en donde se almacenan los vértices con grados de entrada igual a cero, conforme se “ejecuten” los nodos y decrementen sus adyacentes, cuando estos se hacen de grado = 0, se apilan.

→ *Tomando los vértices con grado_in = 0 de una Pila (o Cola)*

Grado_in

C1	C2	C3	C4	C5
0	**0**	2	1	2
0	0	**1**	**0**	2
0	0	1	0	**1**
0	0	**0**	0	**1**
0	0	0	0	**0**



Pila ***P*** : **C1** – **C2**
: C1 // C1 – **C4**
: C1 // C1
: // **C3**
: // **C5**

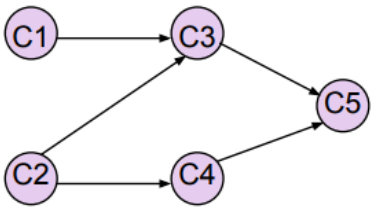
Sort Topológico :

C2 C4 C1 C3 C5

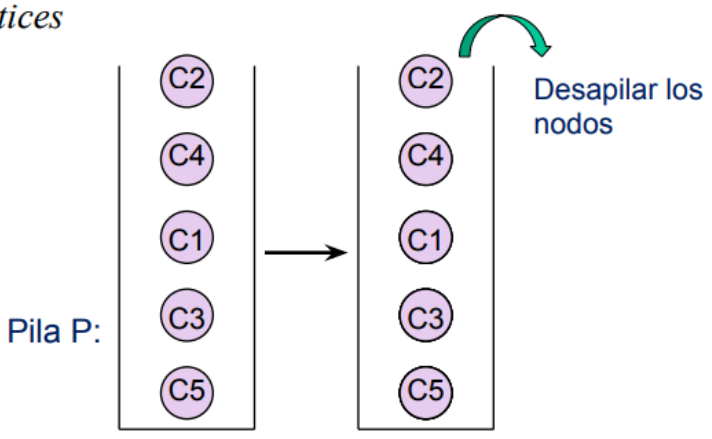
Versión 3 (Ordenación topológica con DFS):

Se realiza un recorrido DFS, marcando cada vértice en post-orden, es decir, una vez visitados todos los vértices a partir de uno dado, los colocamos en una pila P, luego se listan empezando por el tope.

*Opción **b)** - apilando los vértices*



Grafo dirigido acíclico



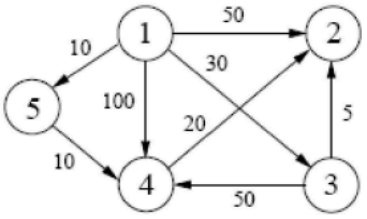
1.- Aplico DFS a partir de un vértice cualquiera, por ejemplo C1, y apilo los vértice en post-orden.
2.- Listo los vértices a medida que los desapilo.

Ordenación Topológica: ***C2 C4 C1 C3 C5***

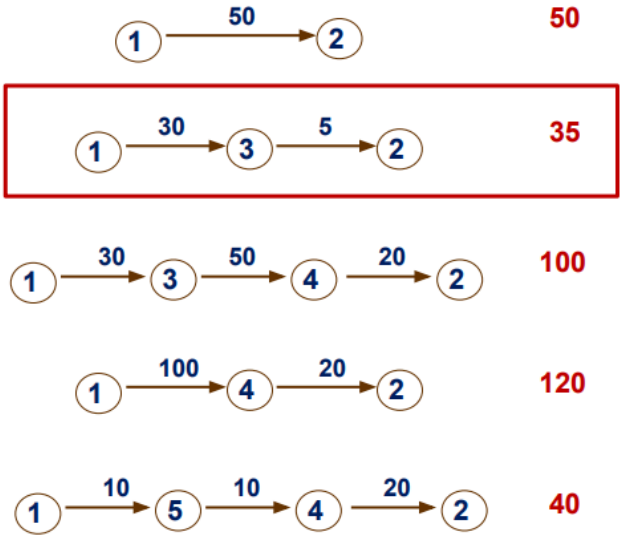
Caminos mínimos

El camino de costo mínimo desde un vértice vi a otro vértice vj es aquel en que la suma de los costos de las aristas es mínima.

Ejemplo:



Caminos posibles desde el **vértice 1** al **vértice 2**



Caminos mínimos en grafos sin peso:

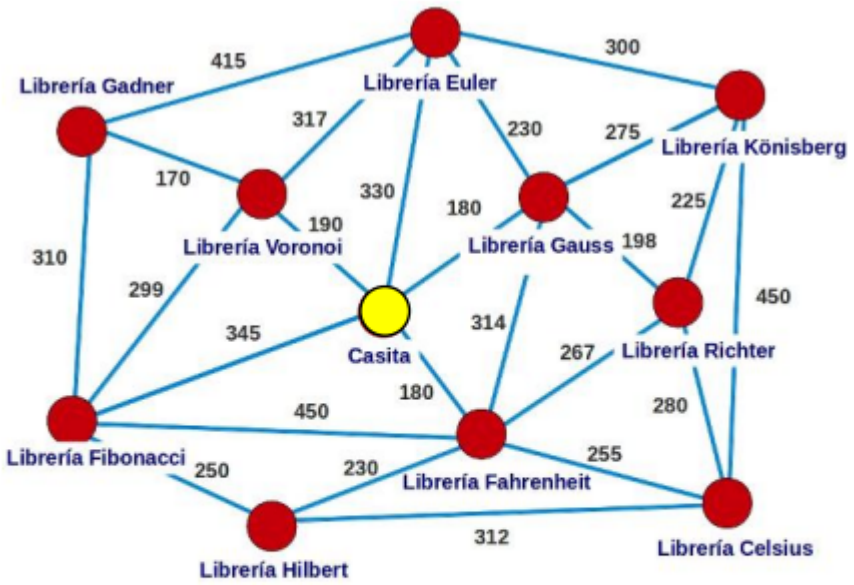
Estrategia: Recorrido de amplitud (BFS)

Para cada vértice v se mantiene la siguiente información:

1. Dv: distancia mínima desde el origen s (inicialmente ∞ para todos lo vértices **excepto el origen con valor 0**)
2. Pv: vértice por donde paso para llegar
3. Conocido : dato booleano que me indica si está procesado (inicialmente todos en 0)
4. (este último campo no va a ser necesario para esta clase de grafos)

Para resolver este problema se utiliza comúnmente el algoritmo de BFS. Este algoritmo explora el grafo por niveles, asegurando que el primer camino encontrado hacia un vértice es el más corto (en términos de número de aristas).

Caminos mínimos en grafos con peso positivo:



Encontrar los caminos más cortos desde Casita a cada una de las librerías

Caminos mínimos con Dijkstra:

Pasos:

1. Dado un vértice origen s, elegir el vértice v que esté a la menor distancia de s, dentro de los vértices no procesados
2. Marcar v como procesado
3. Actualizar la distancia de w adyacente a v

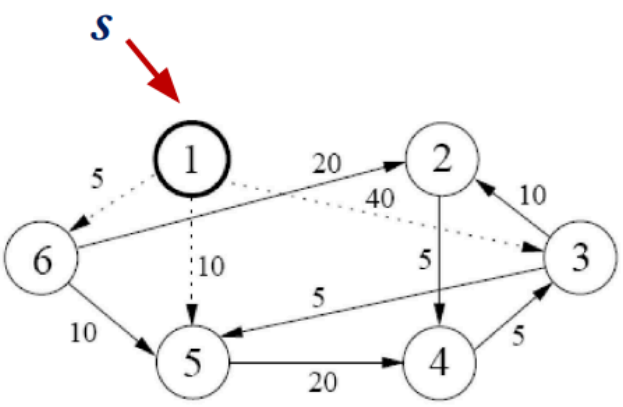
La actualización de la distancia de los adyacentes w se realiza con el siguiente criterio:

- Se compara D_w con $D_v + c(v,w)$

D_w: Distancia de s a w (sin pasar por v)

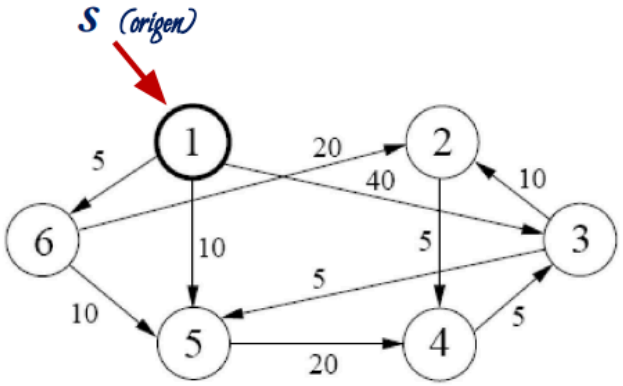
$D_v + c(v,w)$: Distancia de s a w, pasando por v

Se actualiza si D w > D v + c(v,w) ^



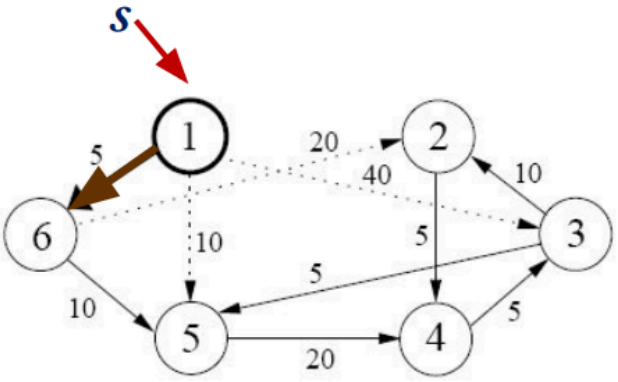
- Valores al seleccionar el vértice 1
- Actualiza la distancia de 3, 5 y 6

V	D_v	P_v	Conoc.
1	0	0	**1**
2	∞	0	0
3	**40**	**1**	0
4	∞	0	0
5	**10**	**1**	0
6	**5**	**1**	0



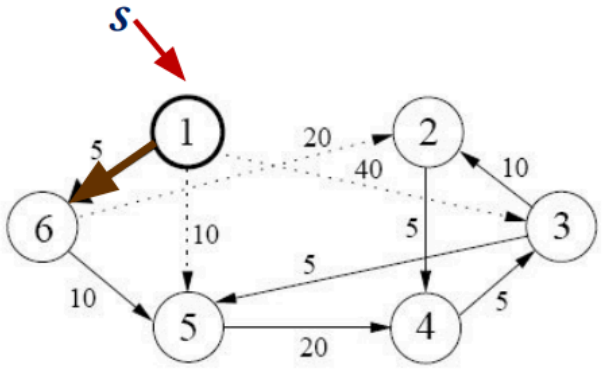
Valores iniciales de la tabla

V	D_v	P_v	Conoc.
1	**0**	0	0
2	∞	0	0
3	∞	0	0
4	∞	0	0
5	∞	0	0
6	∞	0	0



V	D_v	P_v	Conoc.
1	0	0	1
2	∞	0	0
3	40	1	0
4	∞	0	0
5	10	1	0
6	5	1	0

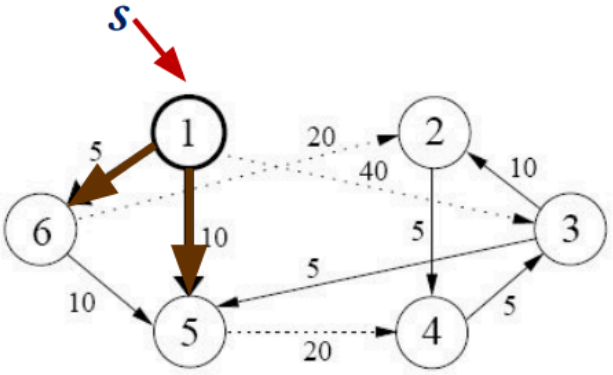
Próximo vértice a elegir →



V	D_v	P_v	Conoc.
1	0	0	1
2	**25**	**6**	0
3	40	1	0
4	∞	0	0
5	10	1	0
6	5	1	**1**

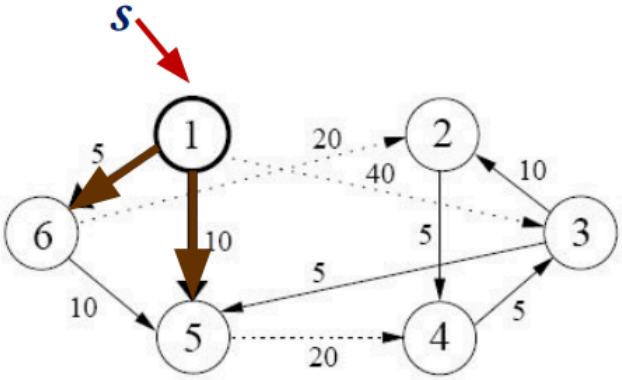
- Valores al seleccionar el vértice 6
- Actualiza la distancia de 2
- La distancia de 5 es mayor que la de la tabla (no se actualiza)

IMPORTANTE NO ACTUALIZAR SI LA NUEVA DISTANCIA ES MAYOR QUE LA DE LA TABLA.



Próximo vértice a elegir

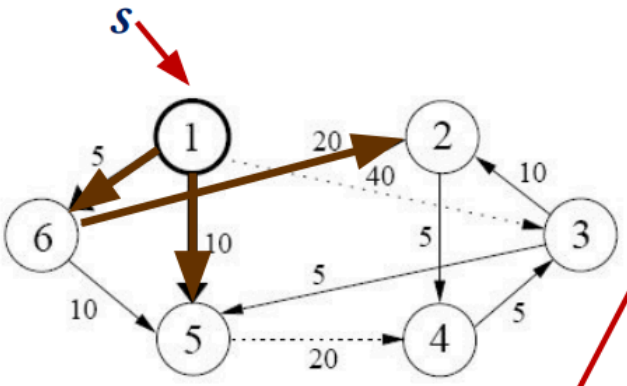
V	D_v	P_v	Conoc.
1	0	0	1
2	25	6	0
3	40	1	0
4	∞	0	0
5	10	1	0
6	5	1	1



- Valores al seleccionar el vértice 5
- Actualiza la distancia de 4

V	D_v	P_v	Conoc.
1	0	0	1
2	25	6	0
3	40	1	0
4	**30**	**5**	0
5	10	1	**1**
6	5	1	1

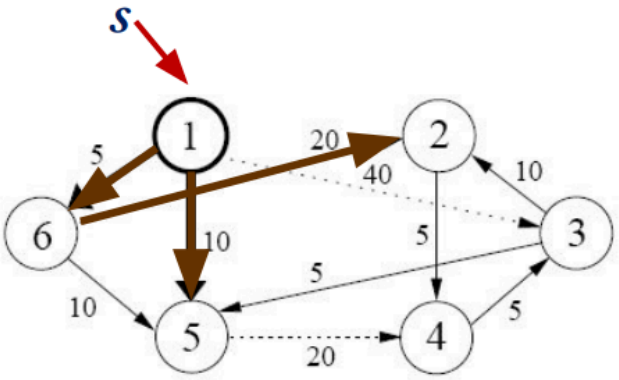
CUANDO SE ACTUALIZA UN NODO, CAMBIAR LA DISTANCIA Y TAMBIÉN EL NODO POR EL QUE HUBO QUE PASAR PARA QUE CAMBIE.



Próximo vértice a elegir

V	D_v	P_v	Conoc.
1	0	0	1
2	25	6	0
3	40	1	0
4	**30**	**5**	0
5	10	1	1
6	5	1	1

SIEMPRE TOMO EL NODO QUE TENGA LA MENOR DISTANCIA Y QUE NO HAYA SIDO CONOCIDO.



- Valores al seleccionar el vértice 2
- La distancia de 4 es igual que la de la tabla (no se actualiza)

V	D_v	P_v	Conoc.
1	0	0	1
2	25	6	**1**
3	40	1	0
4	30	5	0
5	10	1	1
6	5	1	1

- Los próximos vértices a elegir son: 2, 4 y 3 en ese orden.

El resultado final es:

V	D_v	P_v	Conoc.
1	0	0	1
2	25	6	1
3	35	4	1
4	30	5	1
5	10	1	1
6	5	1	1

Tiempo de ejecución:

- Si almacenamos las distancias en un vector, el costo total del algoritmo es O(|V|2)
- Si almacenamos las distancias en una heap, se hace más eficiente, teniendo un coste de O(|E| log|V|)

Caminos mínimos con grafo acíclico (orden topológico) + Dijkstra:

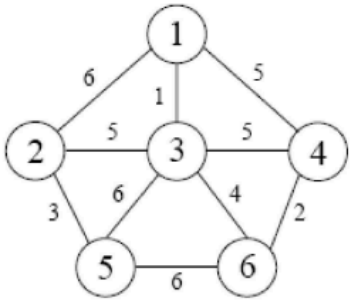
Estrategia: Orden Topológico

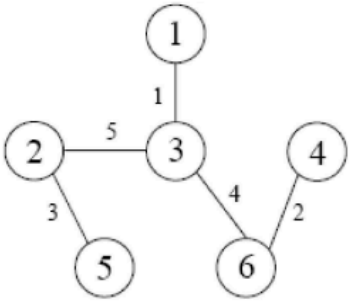
- Optimización del algoritmo de Dijkstra
- La selección de cada vértice se realiza siguiendo el orden topológico
- Esta estrategia funciona correctamente, dado que al seleccionar un vértice v, no se va a encontrar una distancia dv menor, porque ya se procesaron todos los caminos que llegan a él

ARBOL DE EXPANSION MINIMA

Dado un grafo G=(V, E) no dirigido y conexo

El árbol de expansión mínima es un árbol formado por las aristas de G que conectan todos los vértices con un costo total mínimo.





Algoritmo de Prim:

- Construye el árbol haciéndolo crecer por etapas
- Se elige un vértice como raíz del árbol.

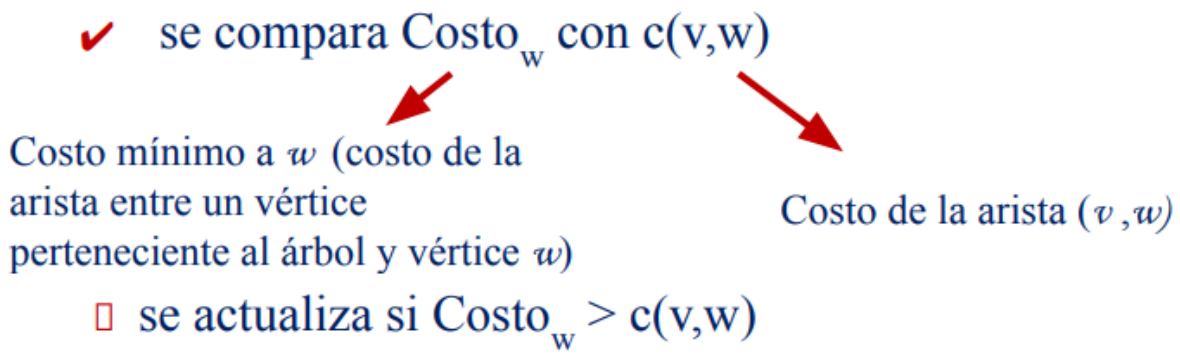
En las siguientes etapas:

A. se selecciona la arista (u,v) de mínimo costo que cumpla: u ∈ árbol y v ∉ árbol
B. se agrega al árbol la arista seleccionada en a) (es decir, ahora el vértice v ∈ árbol)
C. se repite a) y b) hasta que se hayan tomado todos los vértices del grafo

Implementación:

- Para la implementación se usa una tabla (similar a la utilizada en la implementación del algoritmo de Dijkstra).
- La dinámica del algoritmo consiste en, una vez seleccionado una arista (u,v) de costo mínimo tq u ∈ árbol y v ∉ árbol:

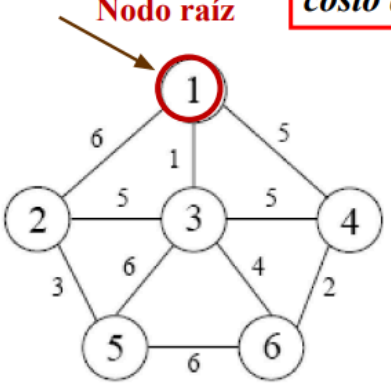
1. se agrega la arista seleccionada al árbol
2. se actualizan los costos a los adyacentes del vértice v de la sig. manera :



- Construye el árbol haciéndolo crecer por etapas

Ejemplo:

1° Paso



costo de la arista (v,w)

Vértice inicial

Vértice elegido

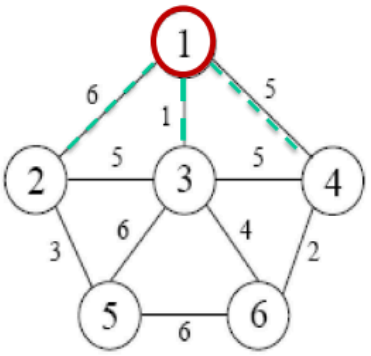
V	*Costo*	*W*	*Conoc.*
1	0	0	1
2	∞	0	0
3	∞	0	0
4	∞	0	0
5	∞	0	0
6	∞	0	0

1° Paso

Vértice elegido

Vértices actualizados

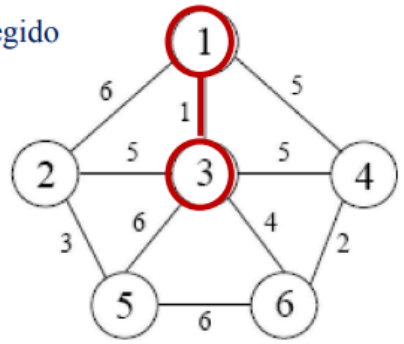
V	*Costo*	*W*	*Conoc.*
1	0	0	1
2	6	1	0
3	1	1	0
4	5	1	0
5	∞	0	0
6	∞	0	0



V	*Costo*	*W*	*Conoc.*
1	0	0	1
2	6	1	0
3	1	1	1
4	5	1	0
5	∞	0	0
6	∞	0	0

Vértice elegido

2° Paso



Se agrega la arista (1,3) y el vértice 3

2° Paso

Vértice elegido

V	*Costo*	*W*	*Conoc.*
1	0	0	1
2	6	1	0
3	1	1	1
4	5	1	0
5	∞	0	0
6	∞	0	0

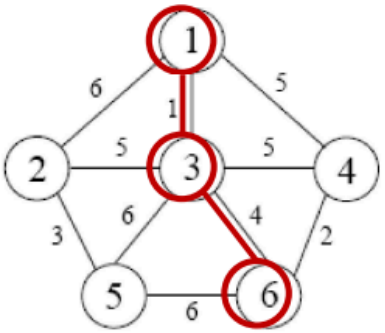
Vértices actualizados

V	*Costo*	*W*	*Conoc.*
1	0	0	1
2	5	3	0
3	1	1	1
4	5	1	0
5	6	3	0
6	4	3	0

V	Costo	W	Conoc.
1	∞	0	1
2	5	3	0
3	1	1	1
4	5	1	0
5	6	3	0
6	4	3	1

Vértice elegido

3° Paso



Se agrega la arista (3,6) y el vértice 6

3° Paso

V	Costo	W	Conoc.
1	∞	0	1
2	5	3	0
3	1	1	1
4	5	1	0
5	6	3	0
6	4	3	1

Vértices actualizados

Vértice elegido

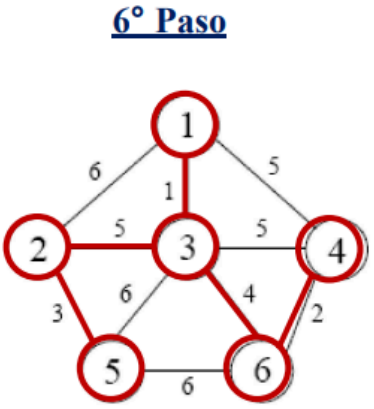
V	Costo	W	Conoc.
1	0	0	1
2	5	3	0
3	1	1	1
4	2	6	0
5	6	3	0
6	4	3	0

SE REPITE HASTA HABER CONOCIDO TODOS LOS VÉRTICES…

Estado final…

V	*Costo*	*W*	*Conoc.*
1	0	0	1
2	5	3	1
3	1	1	1
4	2	6	1
5	3	2	1
6	4	3	1

Vértice elegido



Se agrega la arista (2,5) y el vértice 5

El costo total del algoritmo es O(|E| log|V|)

La principal diferencia entre el Prim y Dijkstra es que en el Prim, cuando el costo es menor al que había antes se "pisa", en cambio en el Dijkstra se van sumando.

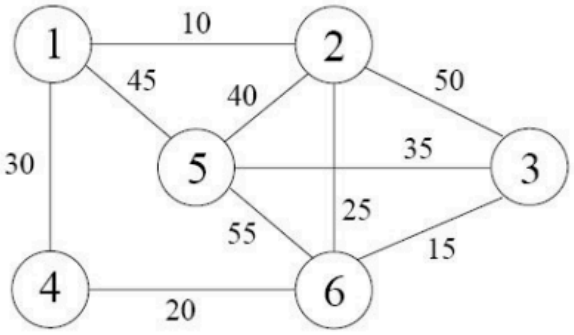
Algoritmo de Kruskal:

- Inicialmente cada vértice pertenece a su propio conjunto (|V| conjuntos con un único elemento)
- Al aceptar una arista se realiza la unión de dos conjuntos.
- Las aristas se organizan en una heap, para ir recuperando la de mínimo costo en cada paso.
- Selecciona las aristas en orden creciente según su peso y las acepta si no originan un ciclo.
- Si dos vértices u y v están en el mismo conjunto, la arista (u,v) es rechazada porque al aceptarla forma un ciclo.

Pasos del Algoritmo de Kruskal

Ordenar todas las aristas por su costo de menor a mayor

Ejemplo:

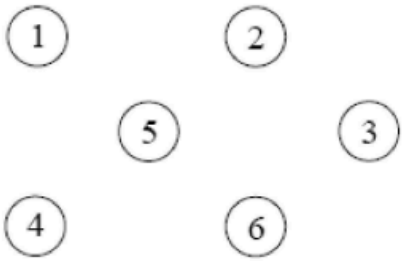


Aristas ordenadas por su costo de menor a mayor:

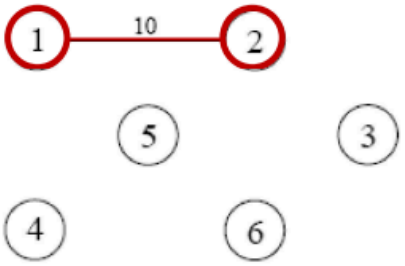
(1,2) □ 10
(3,6) □ 15
(4,6) □ 20
(2,6) □ 25
(1,4) □ 30
(5,3) □ 35
(5,2) □ 40
(1,5) □ 45
(2,3) □ 50
(5,6) □ 55

- Ordenar las aristas, usando un algoritmo de ordenación
- Construir una min-heap □ **más eficiente**

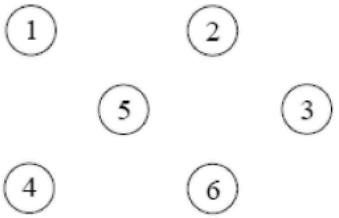
Inicialmente cada vértice está en su propio conjunto



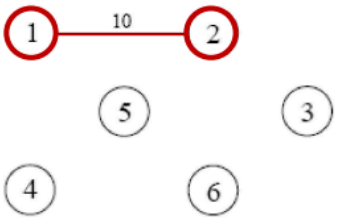
Se agrega la arista (1,2)



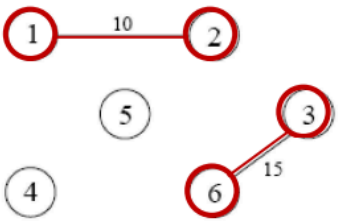
Inicialmente cada vértice está en su propio conjunto

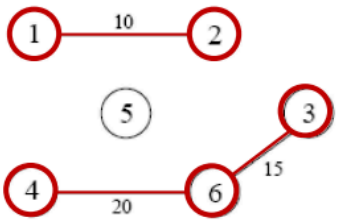


Se agrega la arista (1,2)



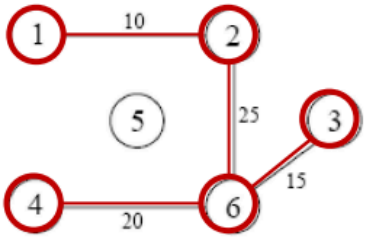
Se agrega la arista (3,6)



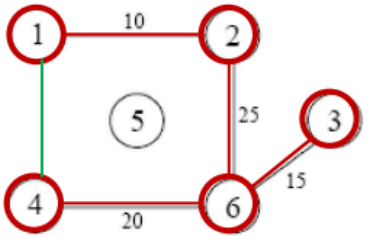


Se agrega la arista (4,6)

Se agrega la arista (2,6)



¿Se agrega la arista (1,4) con costo 30?



No, porque forma ciclo, ya que pertenece a la misma componente conexa

AL AGREGAR LA ARISTA (2,6) SE FORMA UNA SOLA COMPONENTE CONEXA, POR LO TANTO AL QUERER AGREGAR (1,4) NO PUEDO PORQUE YA PORQUE SI DOS VÉRTICES U Y V ESTÁN EN EL MISMO CONJUNTO, LA ARISTA (u,v) ES RECHAZADA PORQUE AL ACEPTARLA FORMARÍA UN CICLO.