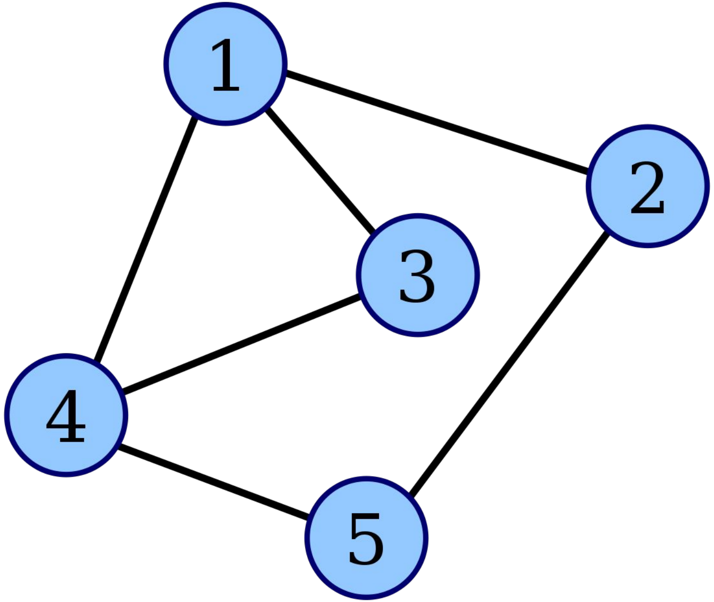
Grafos


GRAFOS
GRAFOS EVERYWHERE
imgflip.com

Intro y clasificaciones

Intro

Un grafo es un conjunto de objetos llamados **vértices** unidos por enlaces llamados **aristas**, que permiten representar relaciones binarias entre elementos de un conjunto.



Vértices: 1,2,3,4,5

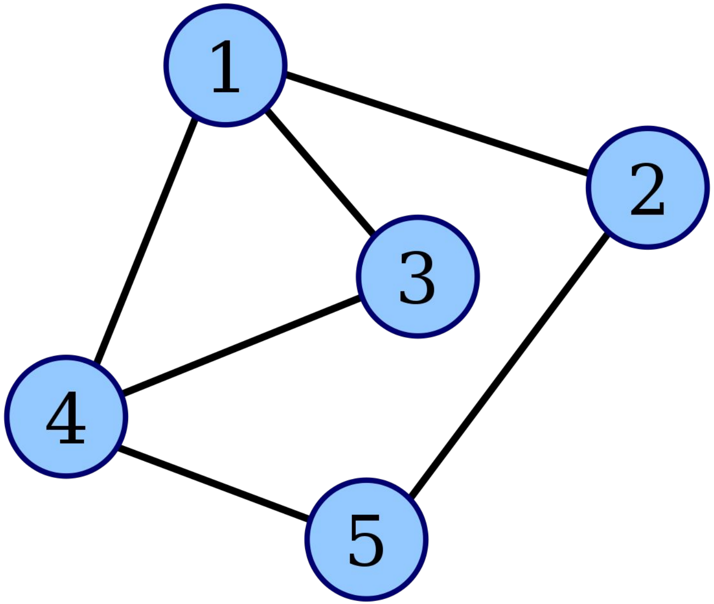
Aristas:
1-2
1-3
4-1
3-4
2-5
5-4

Intro

Los vértices forman relaciones/uniones con otros pares de vértices (incluso ellos mismos).

Esas relaciones se llaman **aristas** y pueden tener dos características básicas:

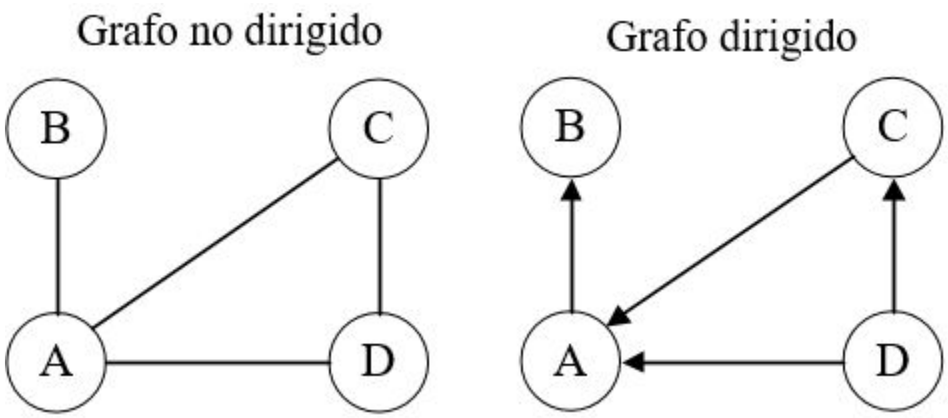
- Dirección
- Ponderación



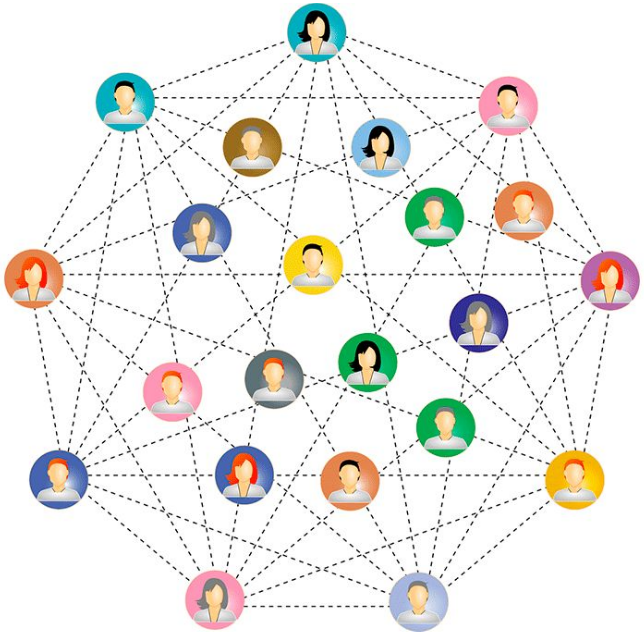
Clasificación: ordenación/dirección

La primera clasificación es según la dirección (o la ausencia de ésta) en las aristas.

Si el grafo es no dirigido y tiene una arista (v,w), entonces (w,v) representa la misma arista. Por lo tanto si w es adyacente a v, entonces v es adyacente a w.



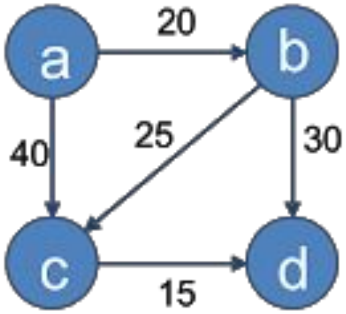
Clasificación: ordenación/dirección



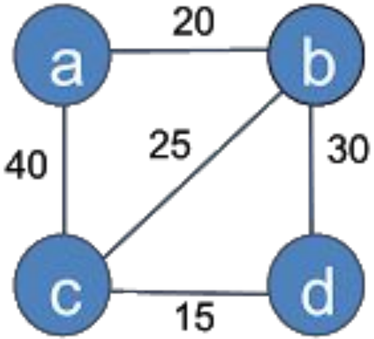
Clasificación: ponderación

Además de los vértices que une, la arista puede tener (o no) un peso asociado a dicha unión.

Llamaremos grafos **ponderados** a aquellos grafos que tengan “peso” o “costo” en sus aristas.



Grafo Dirigido con Costo



Grafo No Dirigido con Costo

Ordenación + ponderación

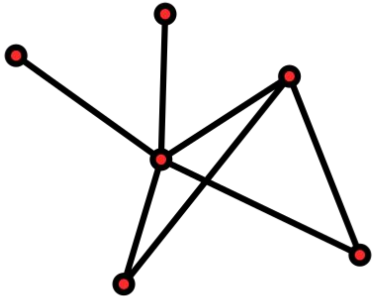
Ambas clasificaciones son totalmente <u>independientes</u> y pueden darse las 4 combinaciones:

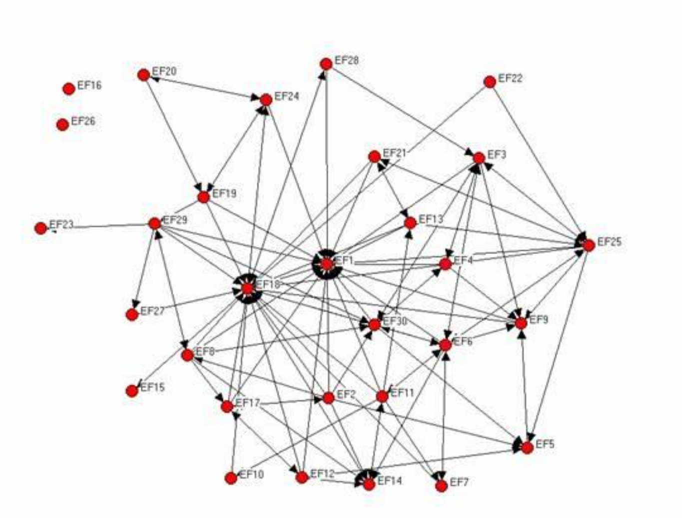
- Dirigido y ponderado: esquinas y calles
- No dirigido y ponderado: aeropuertos
- Dirigido y no ponderado: árbol genealógico
- No dirigido y no ponderado: amistades

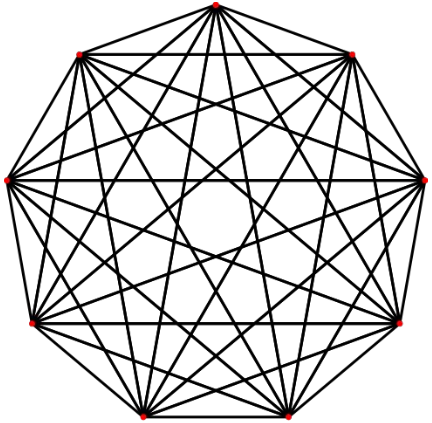
Clasificación: densidad

Otro tipo de clasificación es la relación de aristas que existen con respecto a la cantidad de vértices.

¿Cuáles son los valores posibles de |A| (cantidad de aristas)?







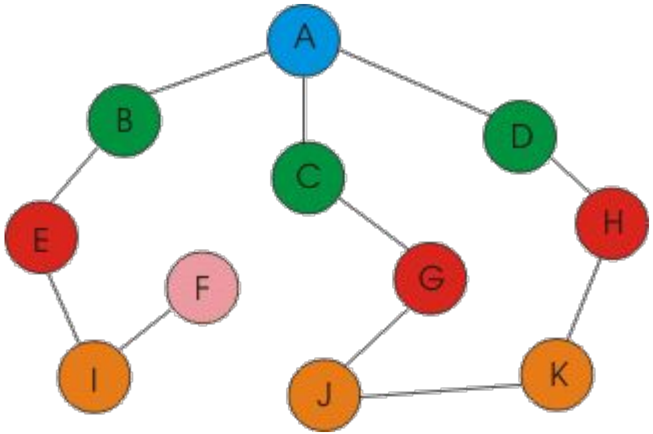
¿Cuáles son los valores posibles de |A| (cantidad de aristas)?

Grafos dirigidos -> V^2

Grafos no dirigidos -> V(V-1)/2

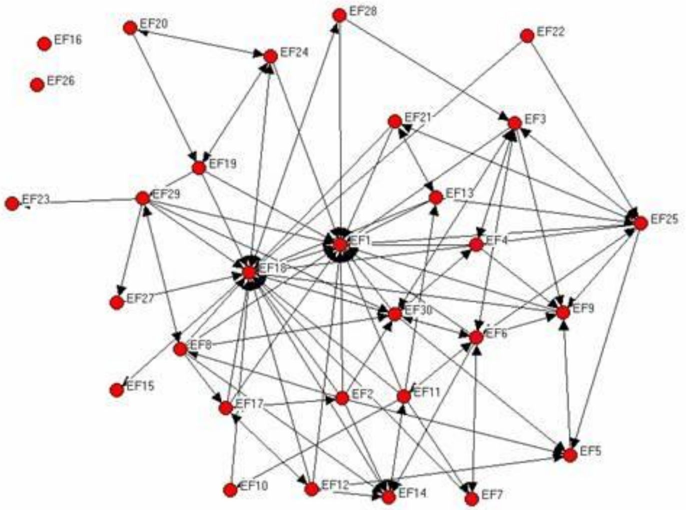
Clasificación: densidad

Aquellos grafos donde la cantidad de aristas es "*bastante menor*" que la cantidad posible de aristas, se les llama: **grafos dispersos**



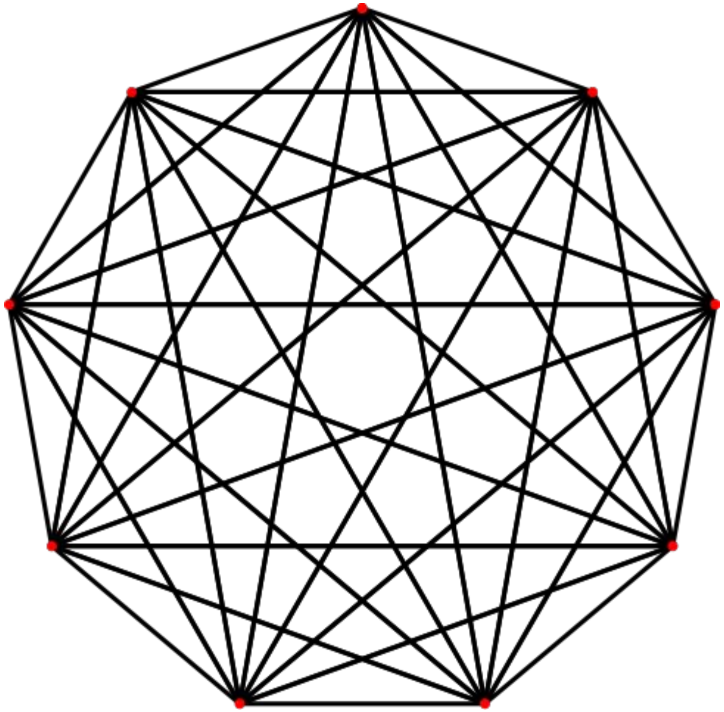
Clasificación: densidad

Aquellos grafos donde la cantidad de aristas se "*aproxima*" a la cantidad posible de aristas, se les llama: **grafos densos**



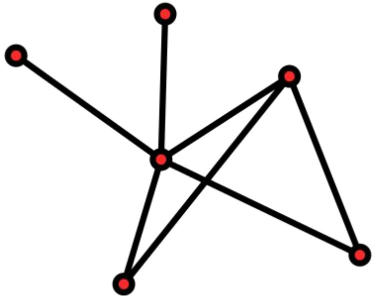
Clasificación: densidad

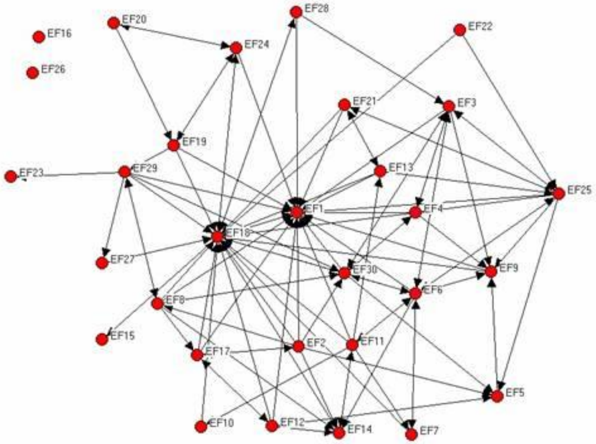
Hay un tipo especial de grafo denso: un **grafo completo**. Es aquel grafo donde existe una arista para cualquier par de vértices.

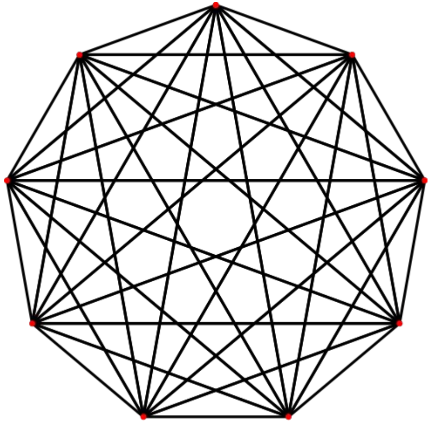


Clasificación: densidad

Quizás no parezca muy significativo en este momento pero esta propiedad del grafo puede que nos determine la forma de implementarlo.







Clasificación: ciclicidad

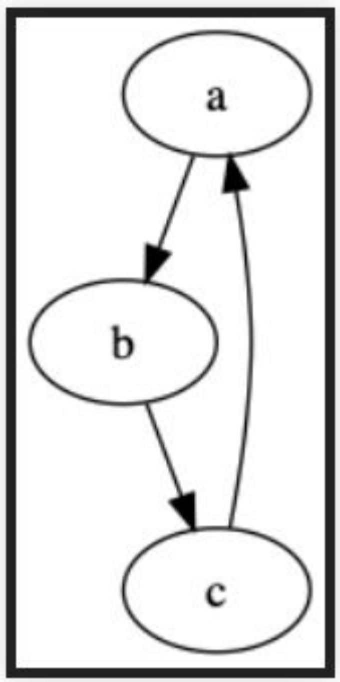
Existe una clasificación según los caminos(*) posibles dentro de un grafo.

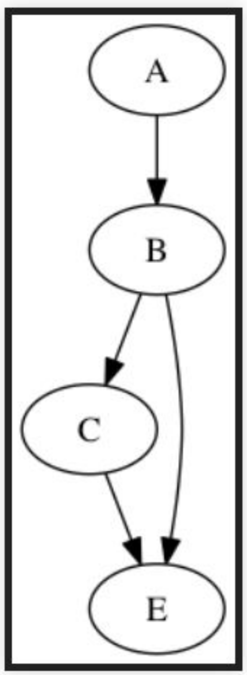
Un grafo es **cíclico** si contiene al menos un ciclo(**), de lo contrario lo llamaremos grafo **acíclico**.

(*) Un camino en un grafo es una lista de vértices de forma que dos vértices consecutivos son adyacentes.
Utilizaremos la siguiente notación: v0↦v1↦...vN.

(**) Un ciclo es un camino donde el vértice de inicio y el final son iguales.
Para los grafos no dirigidos se pide además no usar la misma arista, v->w->v no es un ciclo.

Clasificación: ciclicidad





Clasificación: ciclicidad

Si bien el hecho de que tenga o no ciclos un grafo no afecta su implementación, hay ciertos algoritmos de grafos que deben tenerlo en cuenta. Ej: ordenación topológica, DFS, BFS.

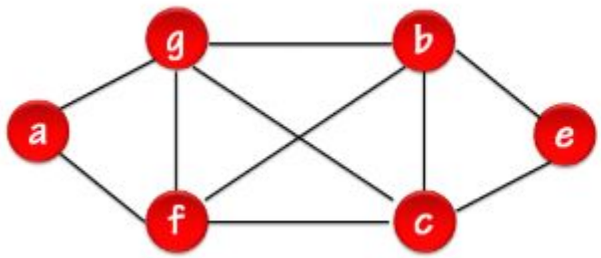
Clasificación: conexidad

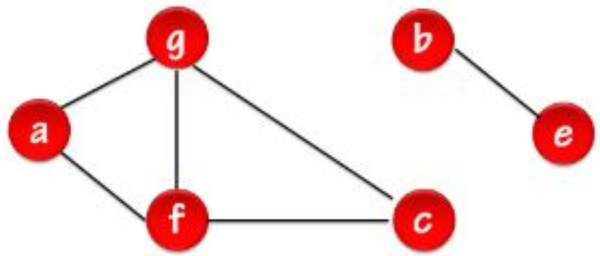
Un grafo no dirigido es **conexo** siempre y cuando haya un camino entre todo par de vértices.

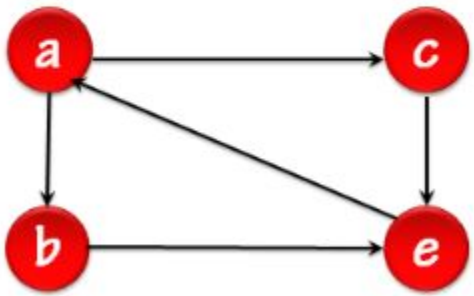
Un grafo dirigido puede ser:

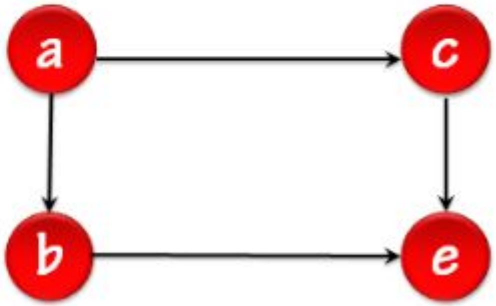
- Fuertemente conexo: hay un camino entre todo par de vértices
- Débilmente conexo: hay un camino entre todo par de vértices si ignoramos la dirección (lo tratamos como un grafo no dirigido, grafo subyacente)
- Ninguno de los anteriores

Clasificación: conexidad







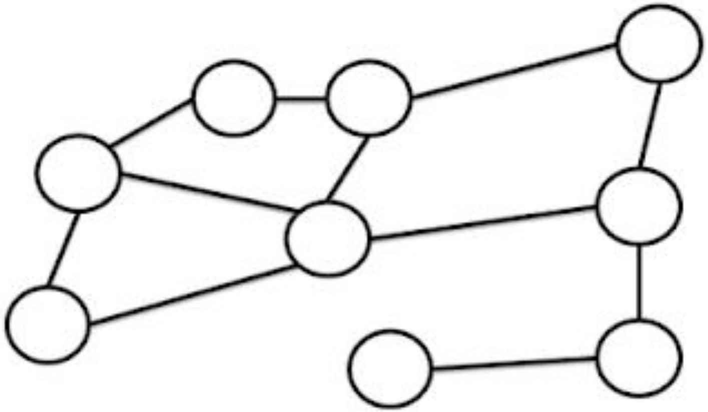


A practicar

Ejercicio: clasificación

Posibilidades:

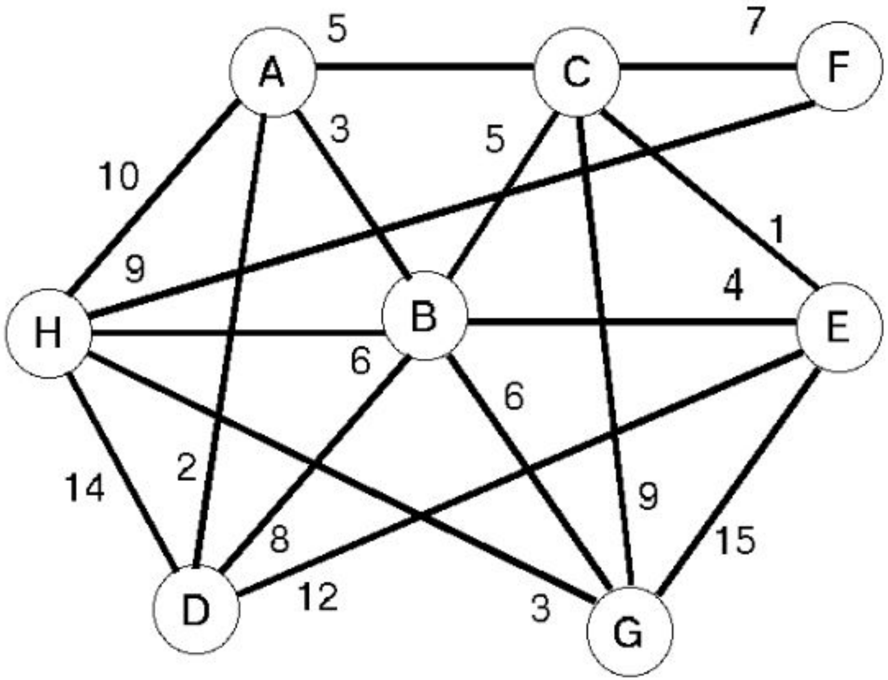
- Dirigido, no dirigido
- Ponderado, no ponderado
- Disperso, denso, ninguna
- Cíclico, acíclico
- Conexo, fuertemente conexo, débilmente conexo, ninguna



Ejercicio: clasificación

Posibilidades:

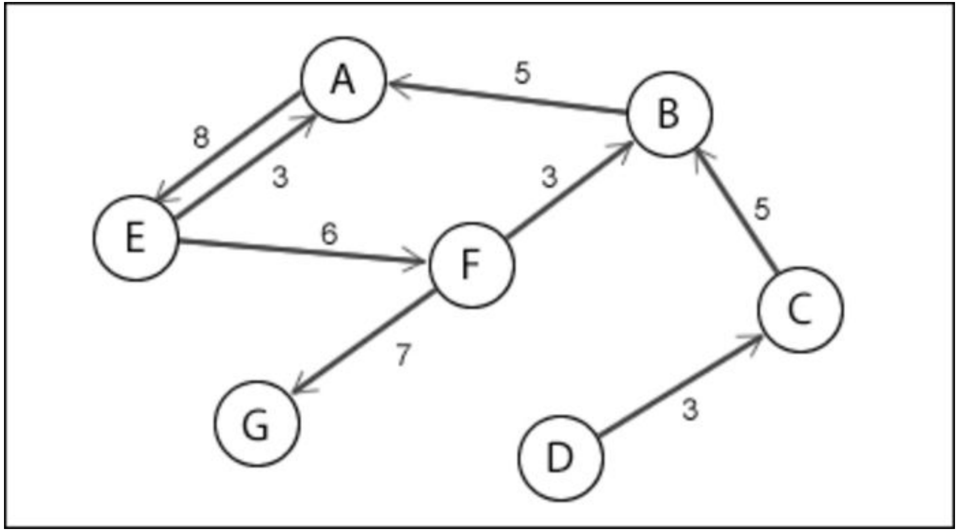
- Dirigido, no dirigido
- Ponderado, no ponderado
- Disperso, denso, ninguna
- Cíclico, acíclico
- Conexo, fuertemente conexo, débilmente conexo, ninguna



Ejercicio: clasificación

Posibilidades:

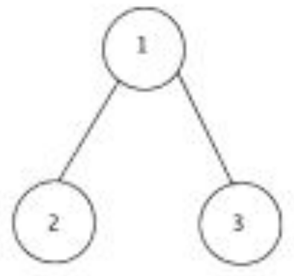
- Dirigido, no dirigido
- Ponderado, no ponderado
- Disperso, denso, ninguna
- Cíclico, acíclico
- Conexo, fuertemente conexo, débilmente conexo, ninguna

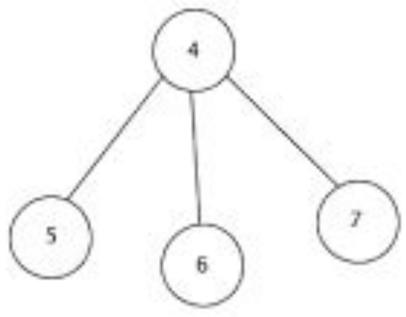


Ejercicio: clasificación

Posibilidades:

- Dirigido, no dirigido
- Ponderado, no ponderado
- Disperso, denso, ninguna
- Cíclico, acíclico
- Conexo, fuertemente conexo, débilmente conexo, ninguna



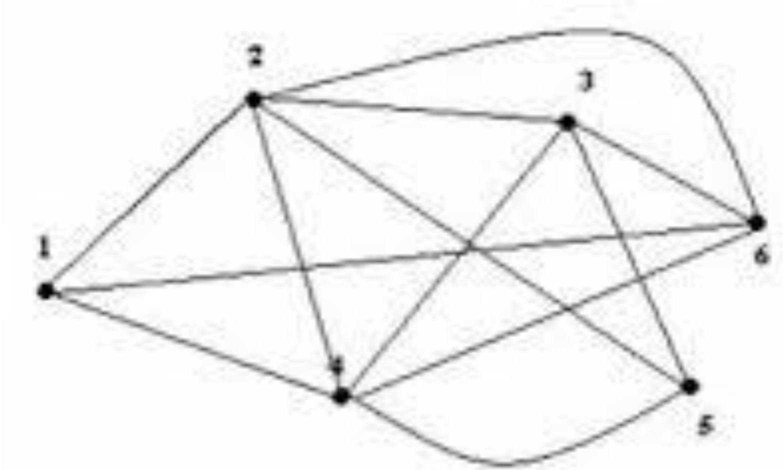


Implementaciones

Implementaciones: intro

Existen dos implementaciones ampliamente usadas:

1. Matriz de adyacencia
2. Lista de adyacencia



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

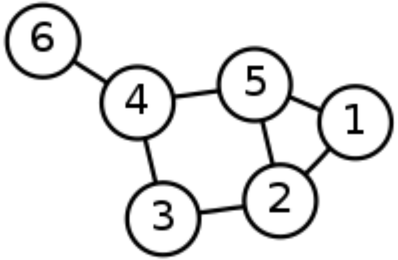
Lista de Adyacencia

1, 2, 4, 6
2, 1, 3, 4, 5, 6
3, 2, 4, 5, 6
4, 1, 2, 3, 5, 6
5, 2, 3, 6
6, 1, 2, 3, 4

Implementaciones: matriz de adyacencia

La matriz de adyacencia representa las aristas entre los pares de vértices en forma de coordenada.

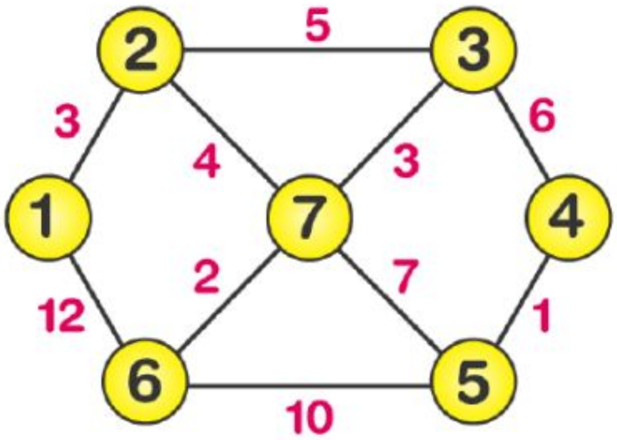
Si es un grafo no ponderado, los 1's pueden representar una arista, mientras que los 0's la ausencia de ella.



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Implementaciones: matriz de adyacencia

Para los grafos ponderados, se puede reemplazar los 1's por el peso de la arista.

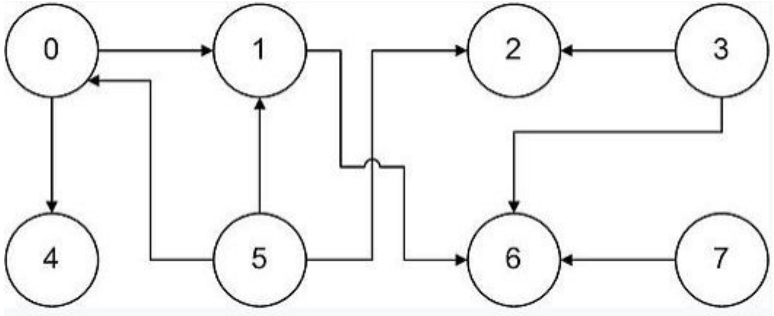


$$A = \begin{bmatrix} 0 & 3 & 0 & 0 & 0 & 12 & 0 \\ 3 & 0 & 5 & 0 & 0 & 0 & 4 \\ 0 & 5 & 0 & 6 & 0 & 0 & 3 \\ 0 & 0 & 6 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 10 & 7 \\ 12 & 0 & 0 & 0 & 10 & 0 & 2 \\ 0 & 4 & 3 & 0 & 7 & 2 & 0 \end{bmatrix}$$

nota: los 0's no son la única representación de ausencia de arista, también se pueden utilizar número "muy grandes" o "muy chicos".

Implementaciones: matriz de adyacencia

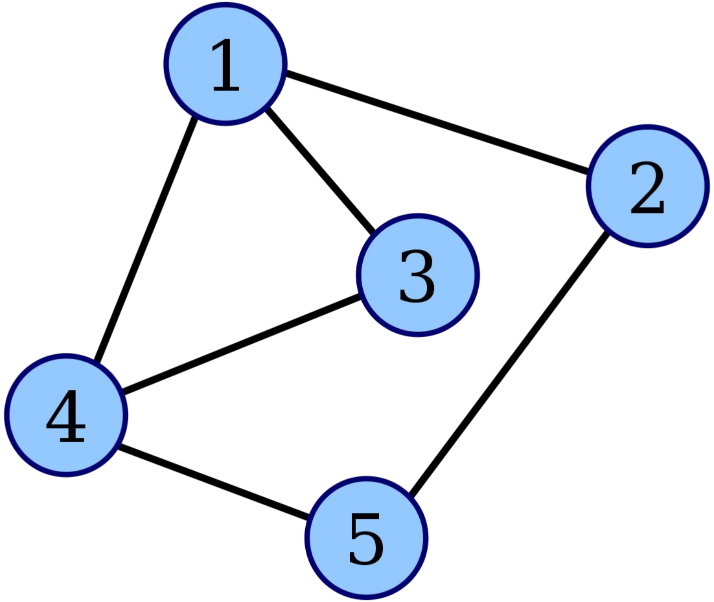
Los grafos dirigidos se representan de la misma manera, a diferencia de los no dirigidos, la matriz no queda simétrica por la diagonal.



$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

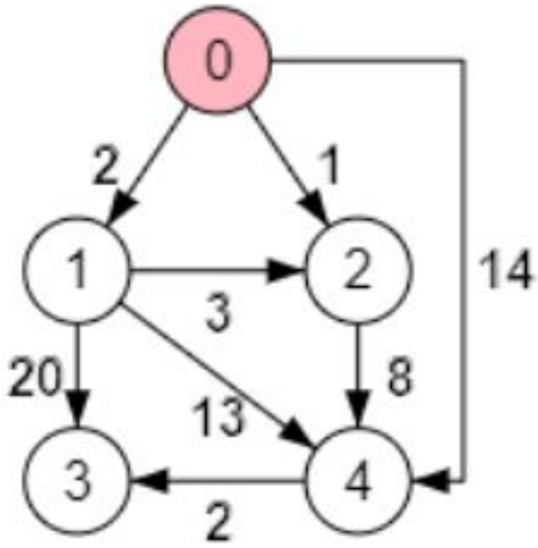
Implementaciones: matriz de adyacencia

Ejercicio: representar este grafo con una matriz de adyacencia. link: Ejercicio matriz de adyacencia / hoja 1



Implementaciones: matriz de adyacencia

Ejercicio: representar este grafo con una matriz de adyacencia. link: Ejercicio matriz de adyacencia / hoja 2



Matriz de adyacencia: pros y cons

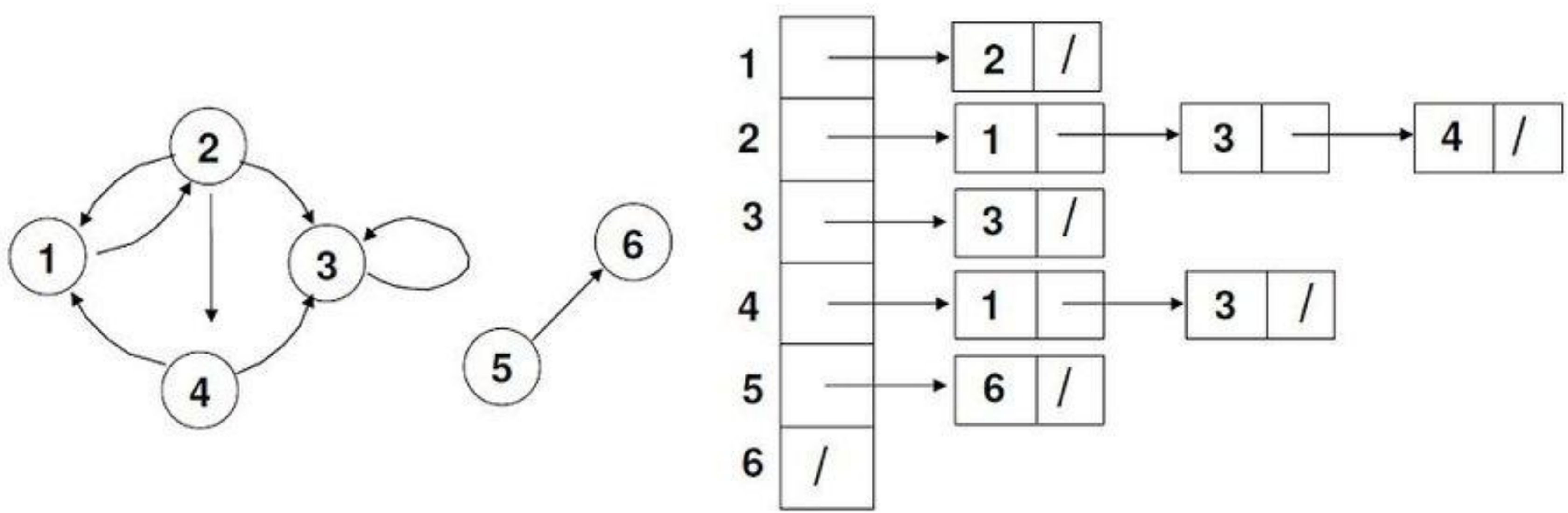
La mayor desventaja de usar una matriz de adyacencia es su **orden espacial** que es de V^2 lo cual es un costo elevado si estamos tratando con grafos dispersos. La mayoría de los casos prácticos tratan de un $|A| << |V\^2|$.

Una ventaja notoria es que podemos saber en O(1) pc si existe una arista entre cualquier par de vértices dado.

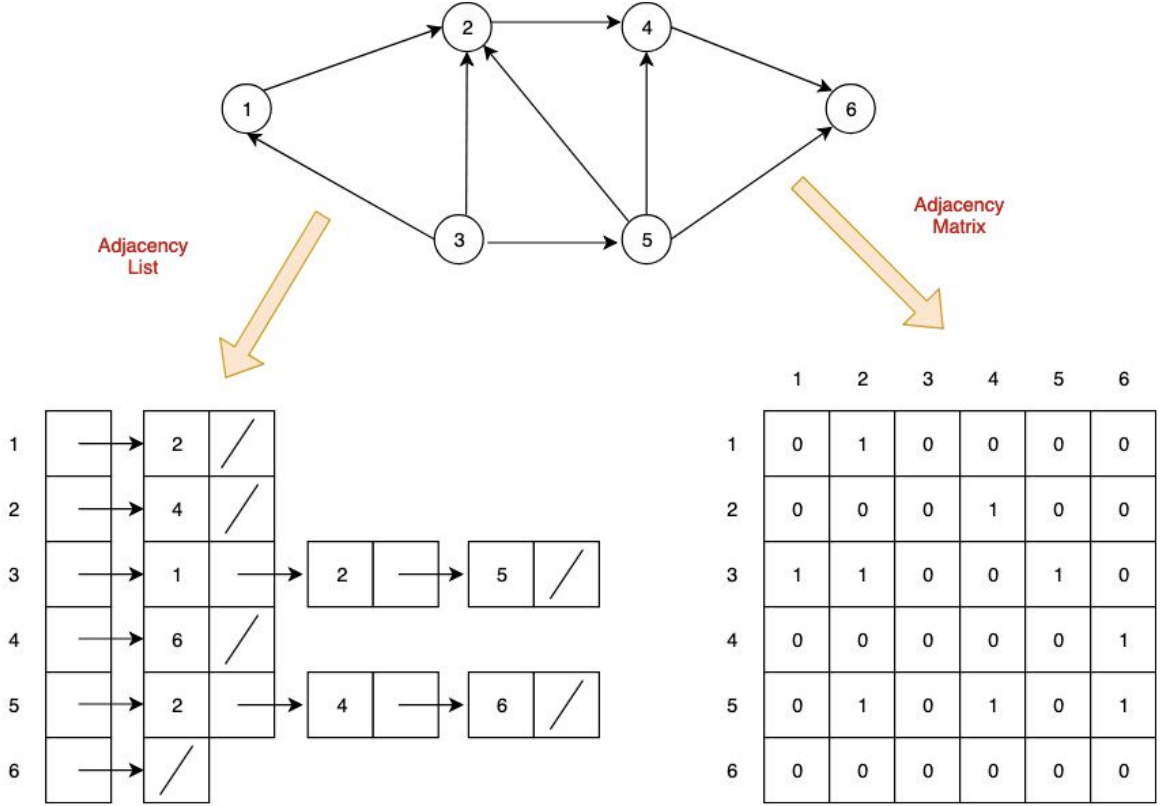
Existen ciertos órdenes de ejecución que veremos en particular a la hora de llevarlo a código. Pero a tener en cuenta: ¿qué orden tiene obtener todos los adyacentes de un vértice dado?

Implementaciones: lista de adyacencia

Se compone de un arreglo de listas, donde cada casillero/celda del arreglo tiene una lista que contiene los vértices adyacentes.



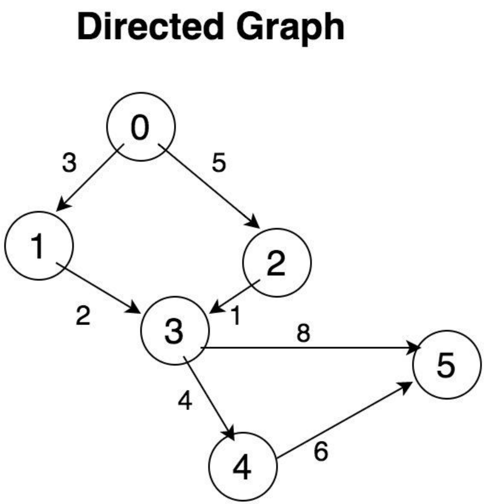
Implementaciones: lista de adyacencia

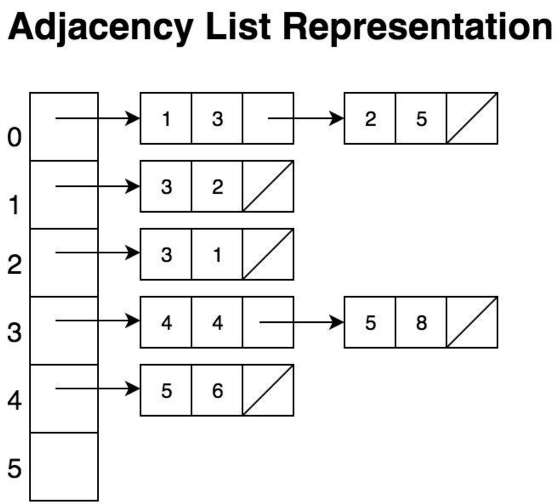


	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	0	1	0	0
3	1	1	0	0	1	0
4	0	0	0	0	0	1
5	0	1	0	1	0	1
6	0	0	0	0	0	0

Implementaciones: lista de adyacencia

Para representar las ponderaciones, simplemente añadimos más datos a los nodos de la lista.





Lista de adyacencia: pros y cons

La mayor desventaja la lista de adyacencia es entenderla conceptualmente. La matriz parece ser algo más intuitivo.

La ventaja principal del uso de la lista de adyacencia es el **orden espacial** |A| + |V|.

A tener en cuenta:
-Obtener los adyacentes de un vértice en O(1) (sin tener en cuenta la recorrida).
-Recorrer todas las aristas es de O(A) y no O(V^2) como en la matriz de ady.

¿Qué orden tiene saber si dos vértices son adyacentes?

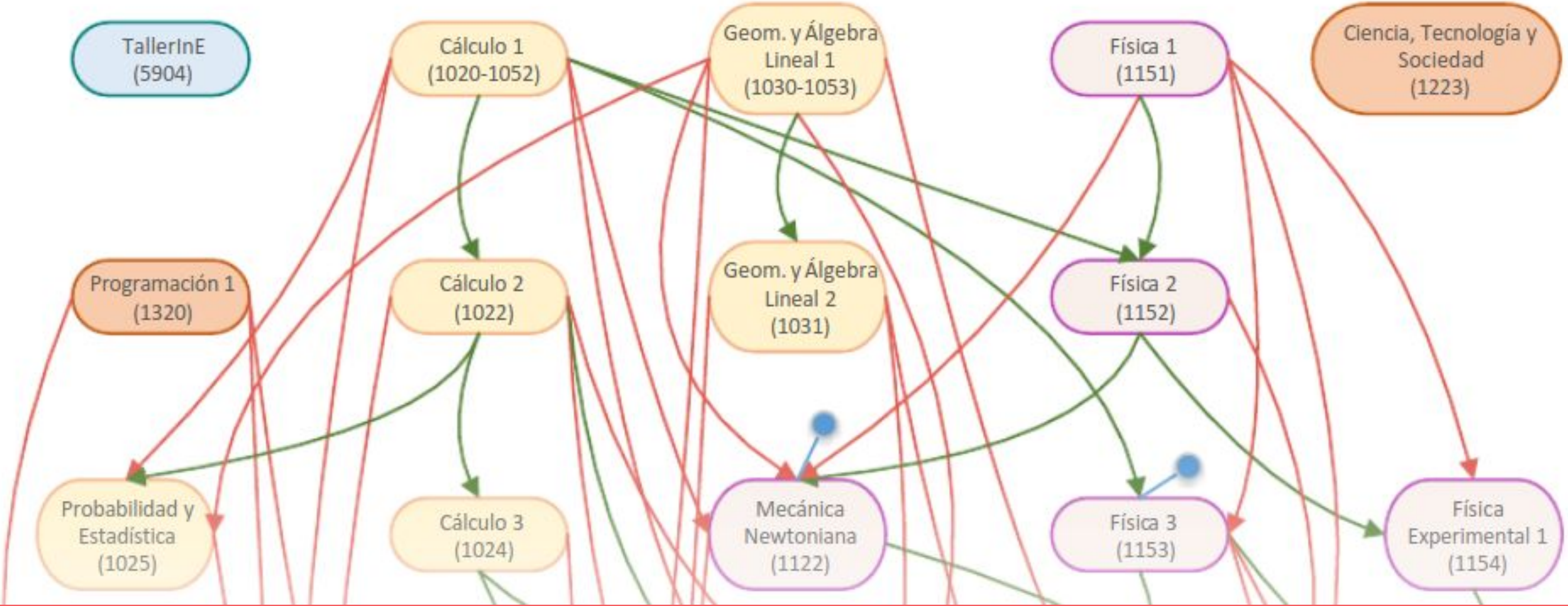
Entonces ...

¿cuál elijo?

Hora de codificar

Ordenación topológica

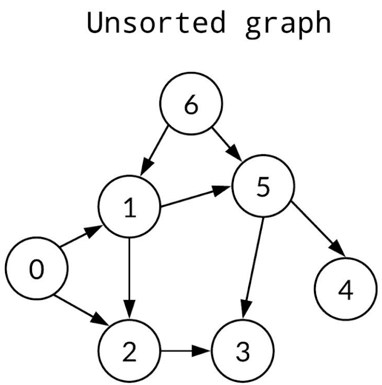
Ejemplo previaturas

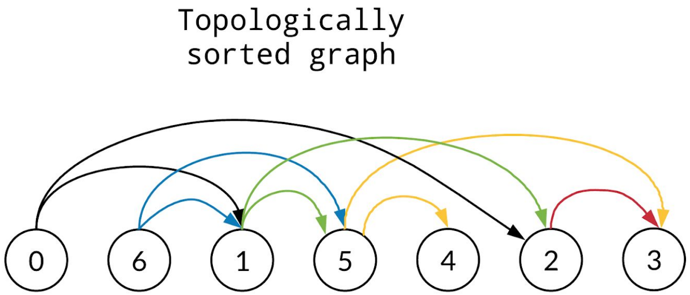


Intro

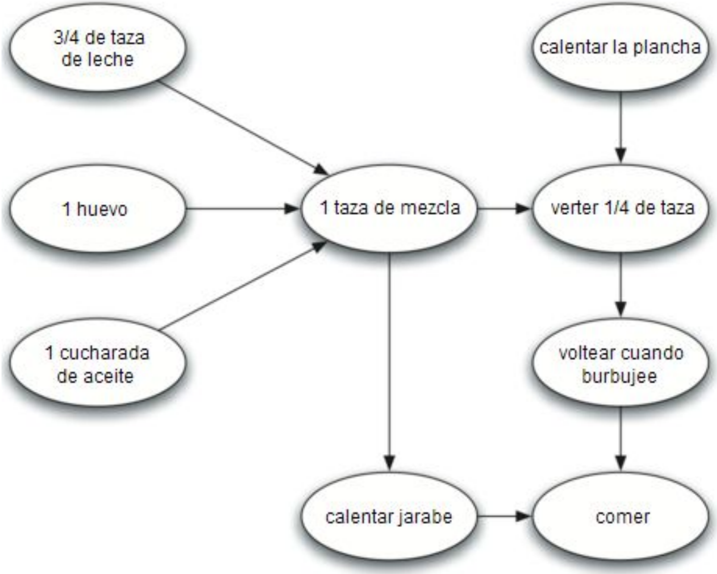
La ordenación topológica una manera de representar el grafo de tal modo que se vea claramente la relación de dependencia.

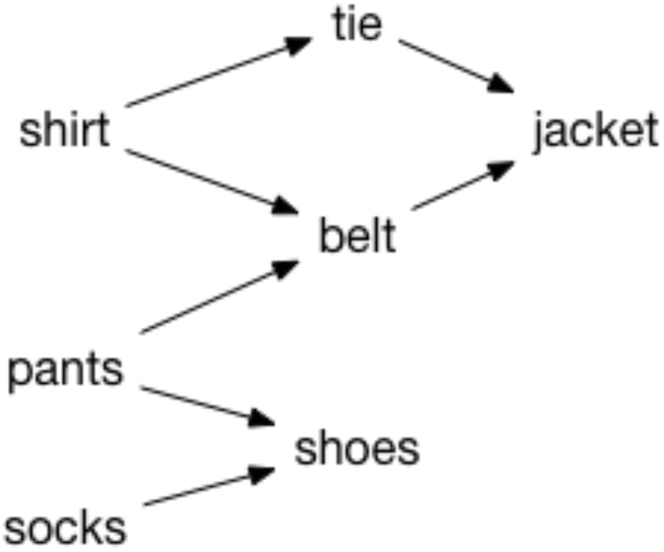
Es un algoritmo que se aplica a grafos dirigidos y acíclicos. Y dado un grafo puede tener más de un orden topológico.





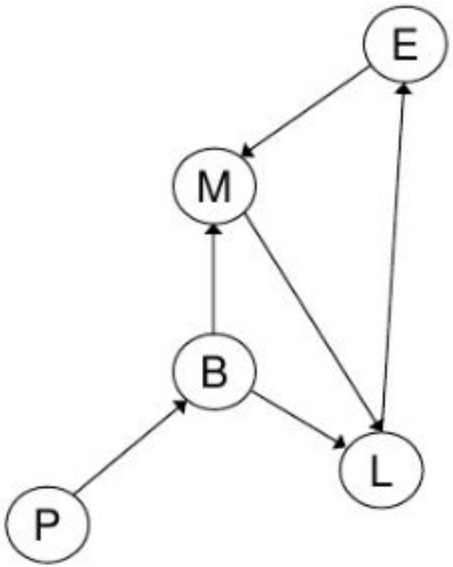
Ejemplos de uso





Grado de entrada _ definición express

El grado de entrada de un vértice es la cantidad de los vértices el cual el es adyacente. En otras palabras, cuántas aristas tiene como destino dicho vértice.



Algoritmo

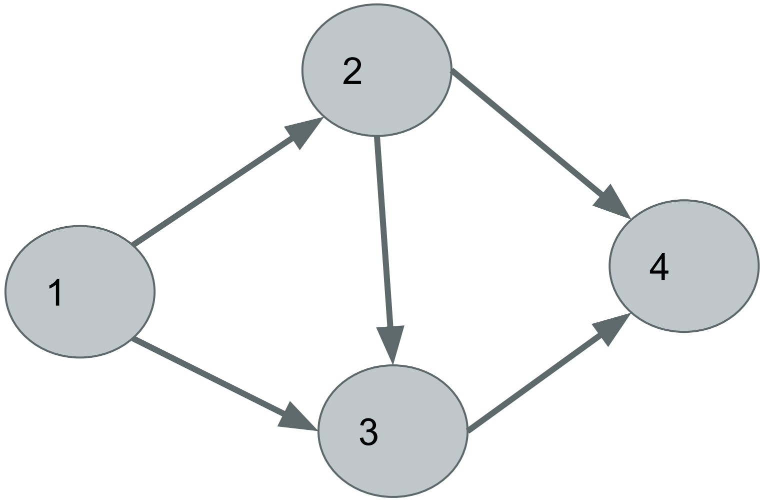
Calcular los índices de entrada de los vértices

Para cada vértice de grado de entrada cero no visitado

- Opero sobre el vértice (lo imprimo por ejemplo)
- Lo marco como visitado
- Para cada adyacente al vértice
  - Reducir su índice de entrada en 1

Ejemplo

Ejercicio OT



Algoritmo

```
void ordenacionTopologica() {
	int * gradoEntrada =  initGradoDeEntrada();
	int * visitados = initVisitados();
	for(int cont = 1; cont <= |V|;cont ++) {
		vertice = verticeGradoEntranteCeroNoVisitado(gradoEntrada  , visitados);
		if (vertice == -1) //No existe
			return cout<<"HAY UN CICLO";
		else{
			visitados[vertice] = true;
			cout << vertice << endl; // lo "proceso"
			for(int w = 1; w <= |V|: w++)
				if(w adyacente a v)
					gradoEntranda[w]--;
		}
	}
}
```

Algoritmo

```
void ordenacionTopologica() {
	int * gradoEntranda =  initGradoDeEntrada(); // O(V^2) MA || O(A) LA
	int * visitados = initVisitados(); // O(V)
	for(int cont = 1; cont <= |V|;cont ++) { // O(V^2)
		vertice = verticeGradoEntranteCeroNoVisitado(gradoEntrante , visitados); // O(V)
		if (vertice == -1) //No existe
			return cout<<“HAY UN CICLO”;
		else{
			visitados[vertice] = true;
			cout << vertice << endl;
			for(int w = 1; w <= |V|: w++)  // O(V) MA <- se puede mejorar con LA
				if(w adyacente a v)
					gradoEntranda[w]--;
		}
	}
}
```

Algoritmo v2

https://www.cs.usfca.edu/~galles/visualization/TopoSortIndegree.html

Algoritmo v2

```
void OrdenacionTopologicaV2() {
	int * gradoEntrada =  initGradoDeEntrada();
	Cola<int> cola; int vertice, cont = 0;
	for(int v = 1; v <= |V|;v ++)
		if (gradoEntranda [v] == 0)
			cola.Encolar(v);

	while (!cola.EsVacia()) {
		vertice = cola.desencolar();
		cont++;
		cout << vertice << endl;
		paraCada w adyacenteA vertice // usamos lista de adyacencia
			gradoEntrada [w]--;
			if(gradoEntrada [w] == 0)
				cola.encolar(w);
	}
	if(cont < V)
		cout << “Error: el grafo tiene ciclos” << endl;
```

Algoritmo v2

```
void OrdenacionTopologicaV2() {
	int * gradoEntranda =  initGradoDeEntrada(); // O(A) LA
	Cola<int> cola; int vertice, cont = 0;
	for(int v = 1; v <= |V|;v ++)
		if (gradoEntranda [v] == 0)
			cola.Encolar(v);

	while (!cola.EsVacia()) { // O(V + A) usando lista de adyacencia
		vertice = cola.desencolar();
		cont++;
		cout << vertice << endl;
		paraCada w adyacenteA vertice // usamos lista de adyacencia
			gradoEntranda [w]--;
			if(gradoEntranda [w] == 0)
				cola.encolar(w);
	if(cont < V)
		cout << “Error: el grafo tiene ciclos” << endl;
```

Otros usos

El algoritmo de ordenación topológica puede ser utilizado para saber si un grafo tiene ciclos. Si no se puede realizar la ordenación topológica quiere decir que hay un ciclo.

Hora de codificar

Camino más corto

en grafos no ponderados

Idea general

Dado un vértice de origen, se recorren las aristas.

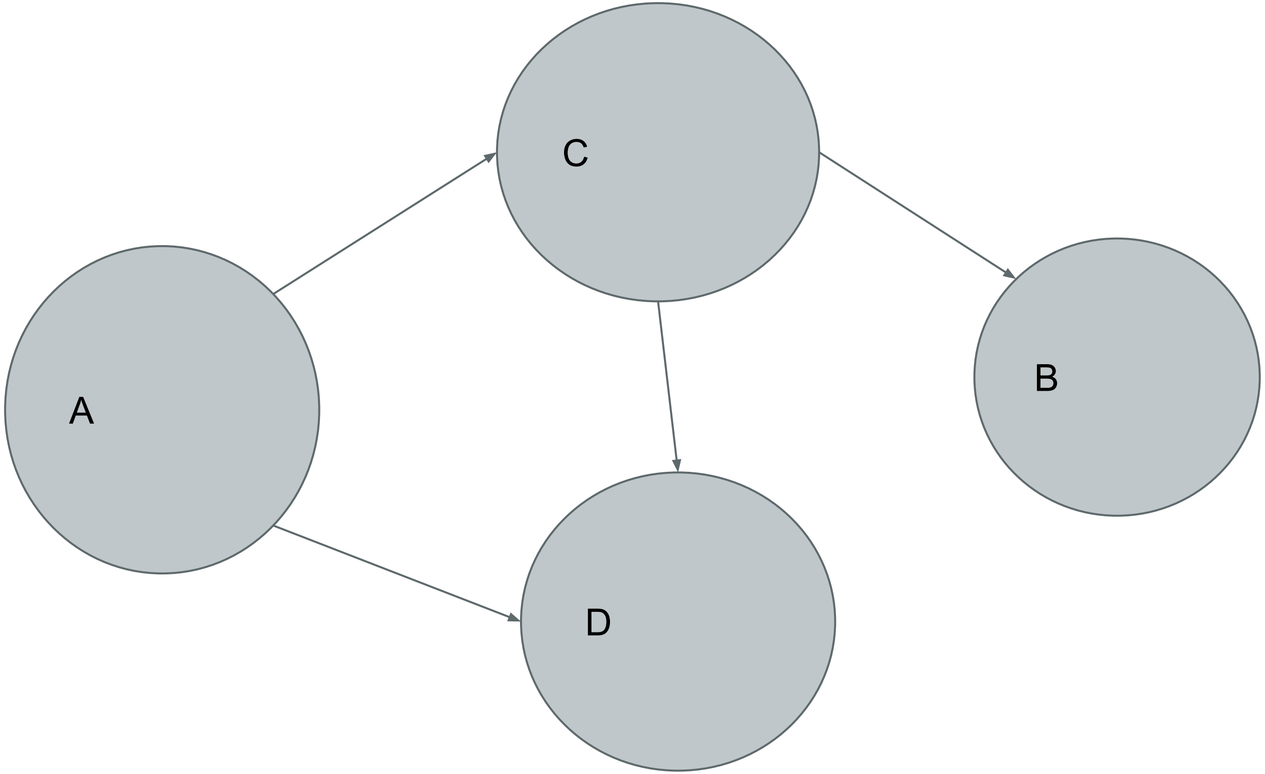
De cada arista de dicho vértice, se puede conocer los vértices adyacentes (a un paso), por lo cual, todos esos vértices tienen un camino de 1 paso (usando una arista).

La idea es aplicar este concepto a los nuevos vértices encontrados siempre y cuando los vértices nuevos encontrados no hayan sido visitados anteriormente.

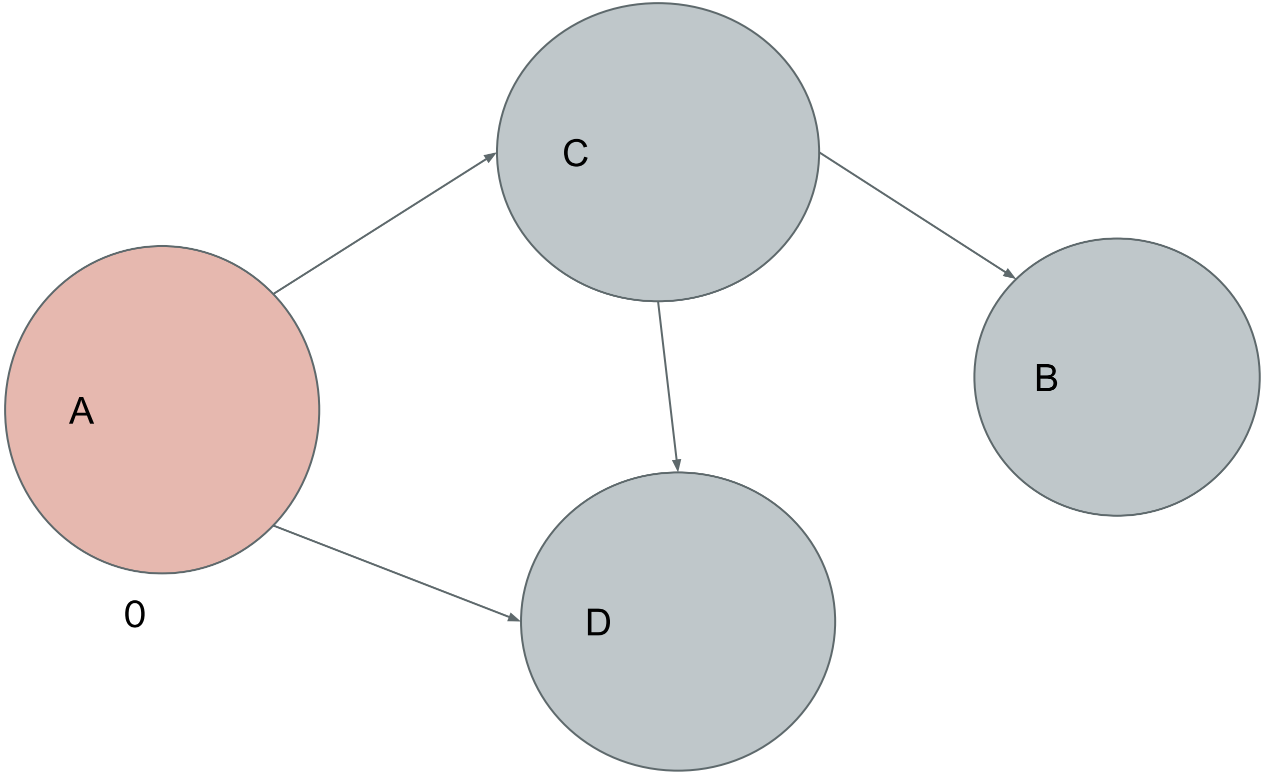
El costo de llegar a ese nuevo vértice será igual a el costo de por donde vine + 1.

Nota: se puede suponer que el costo de llegar al vértice origen es 0.

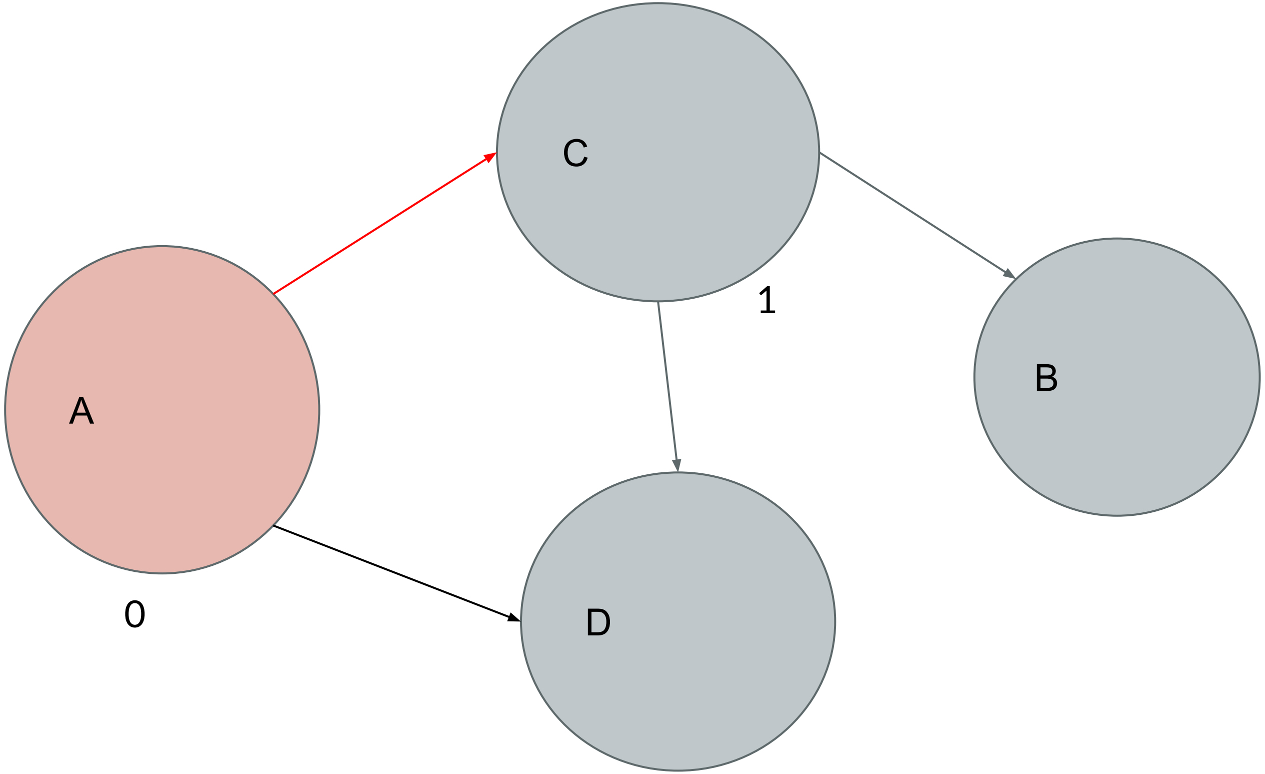
Ejemplo



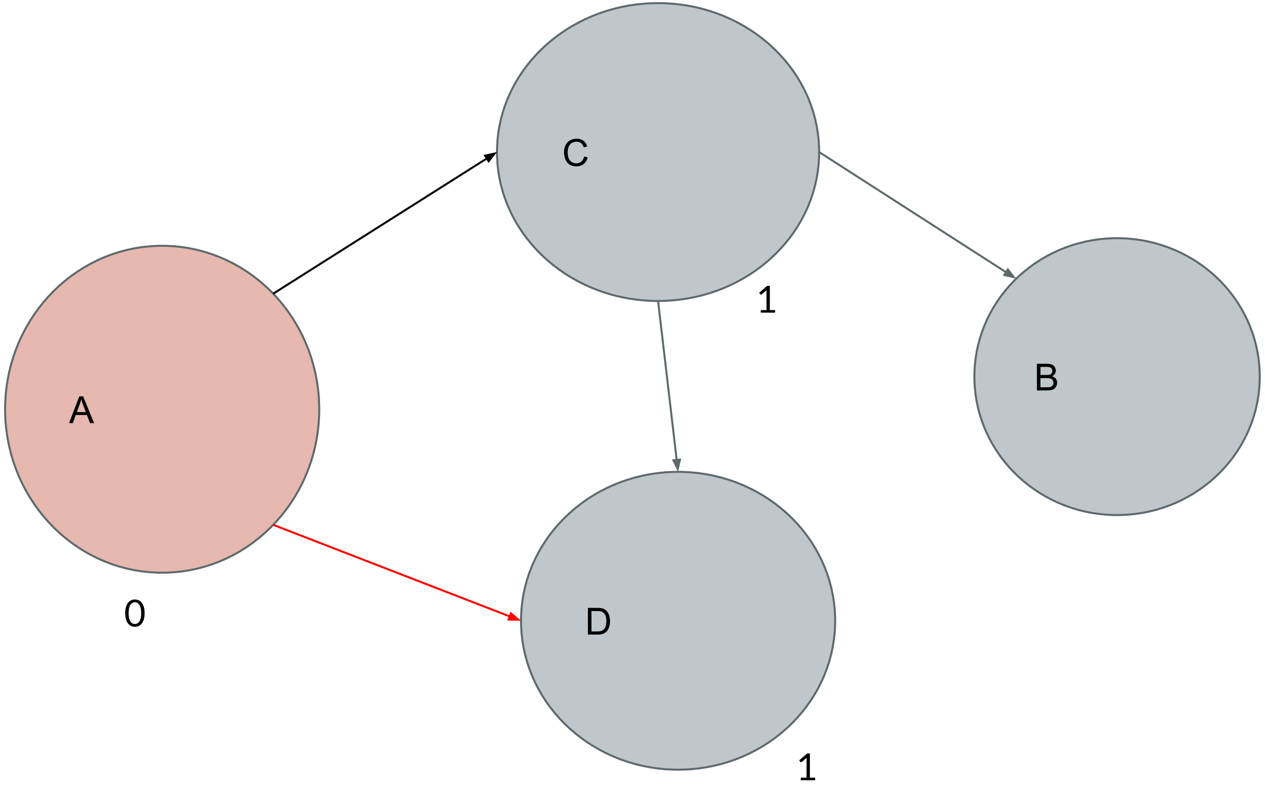
Ejemplo



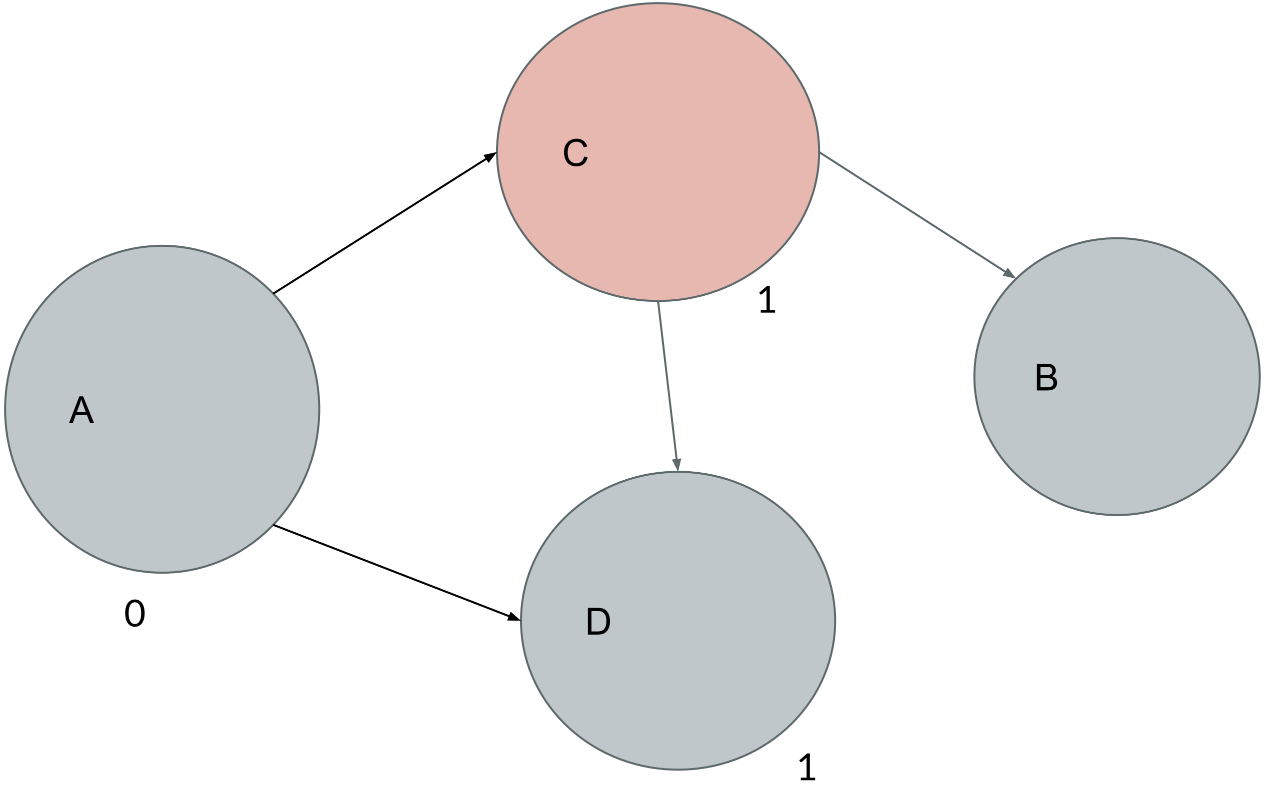
Ejemplo



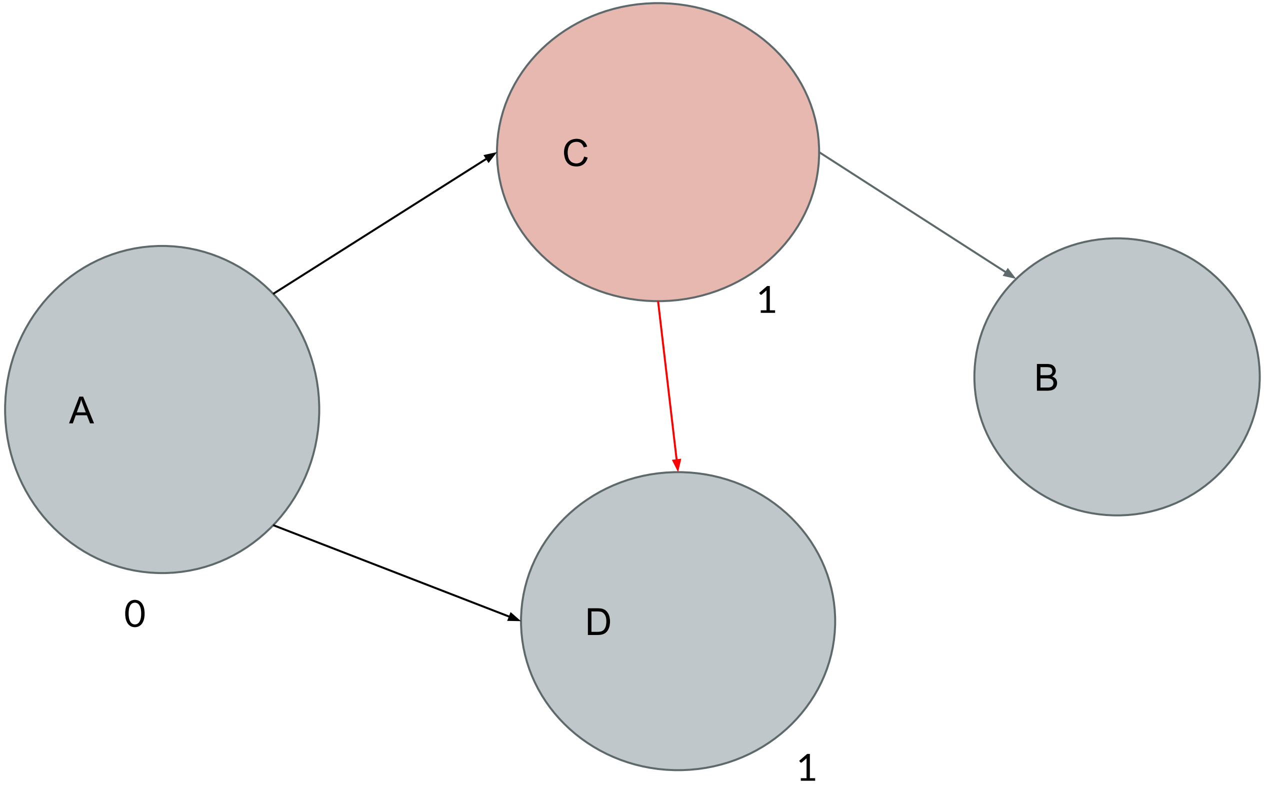
Ejemplo



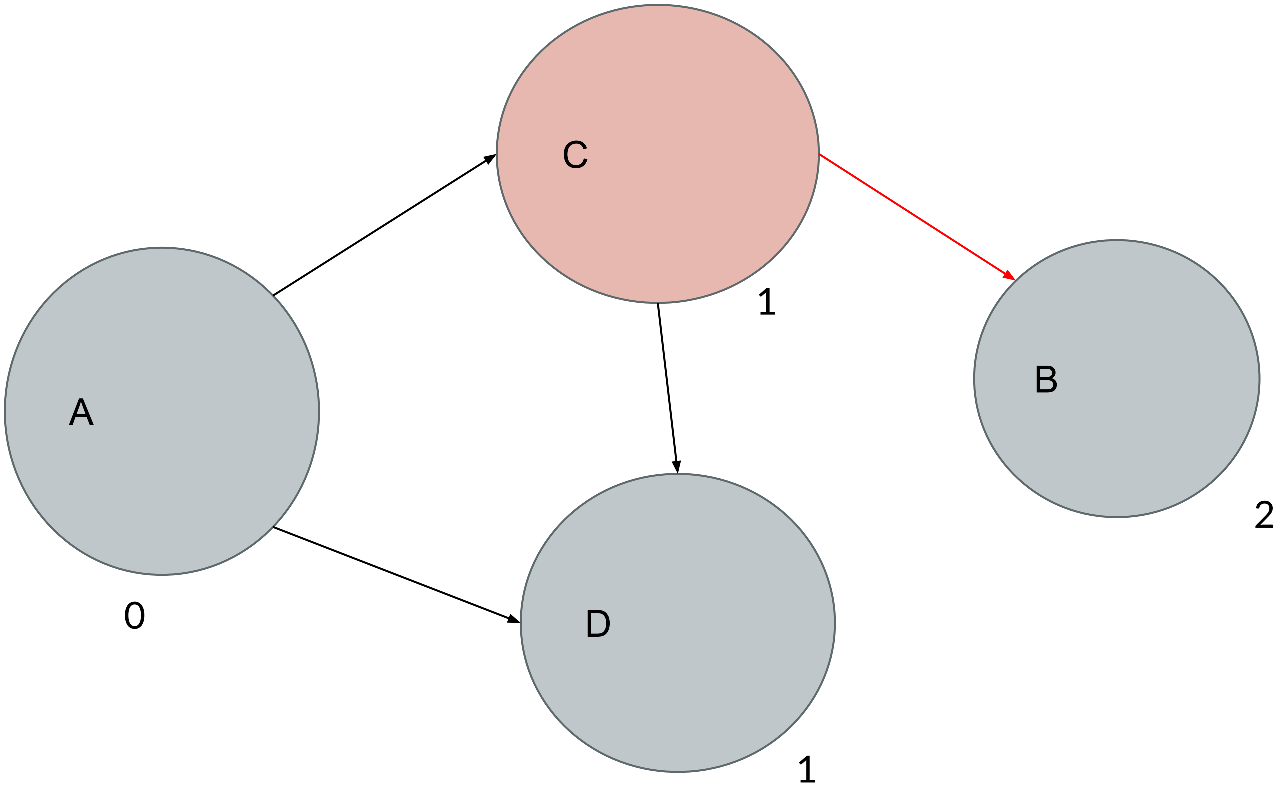
Ejemplo



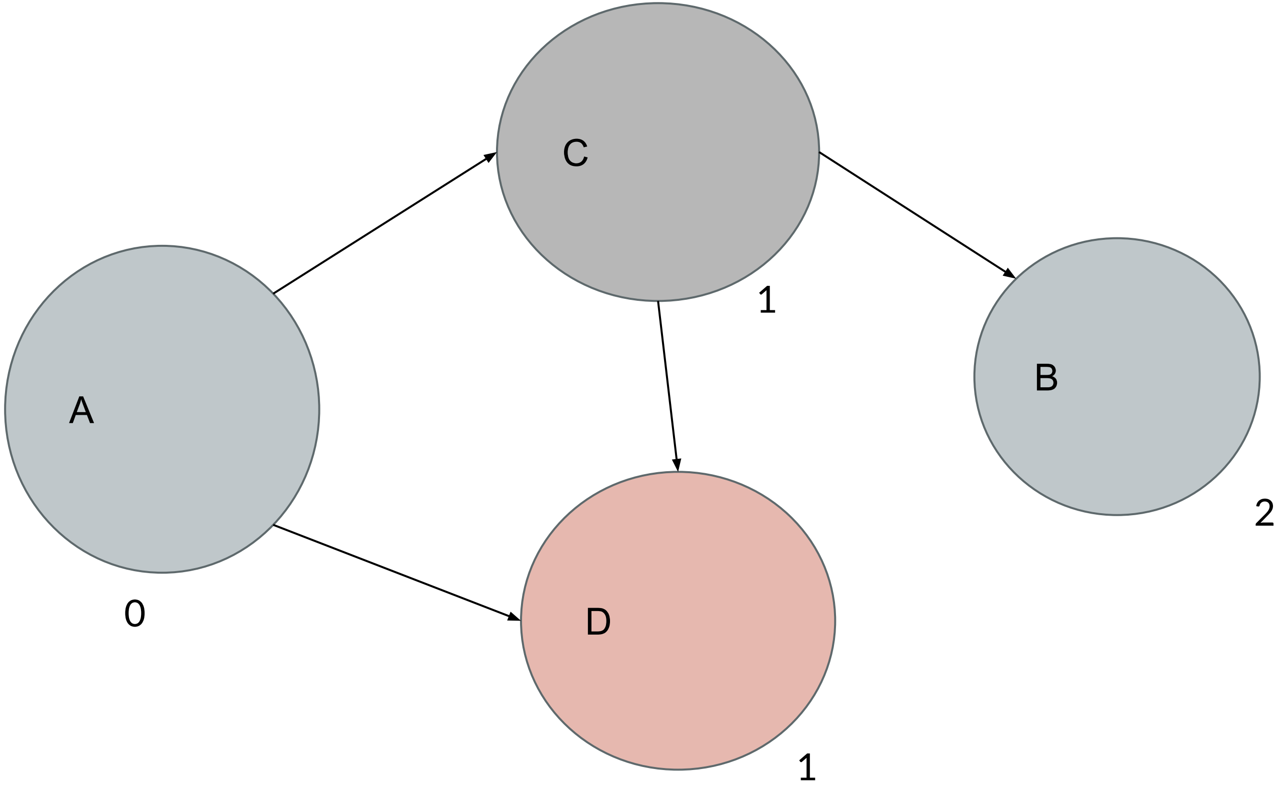
Ejemplo



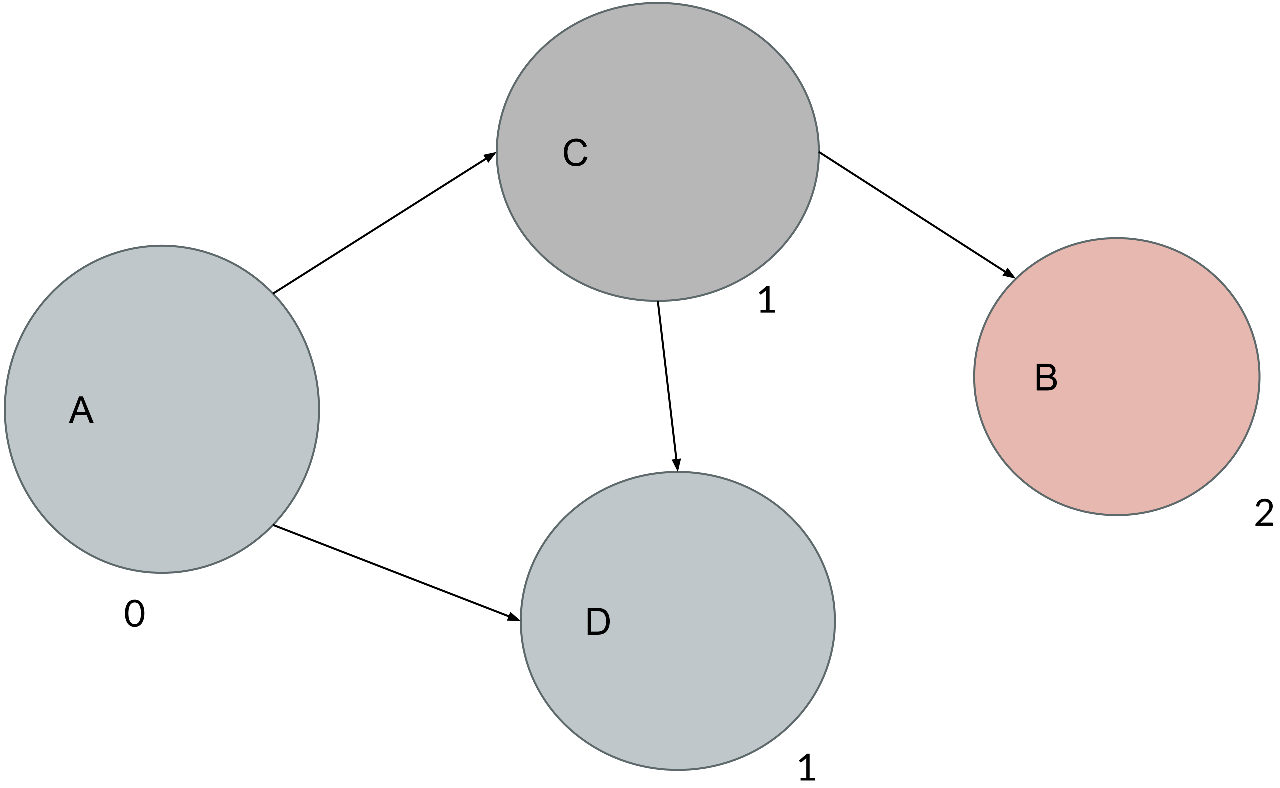
Ejemplo



Ejemplo



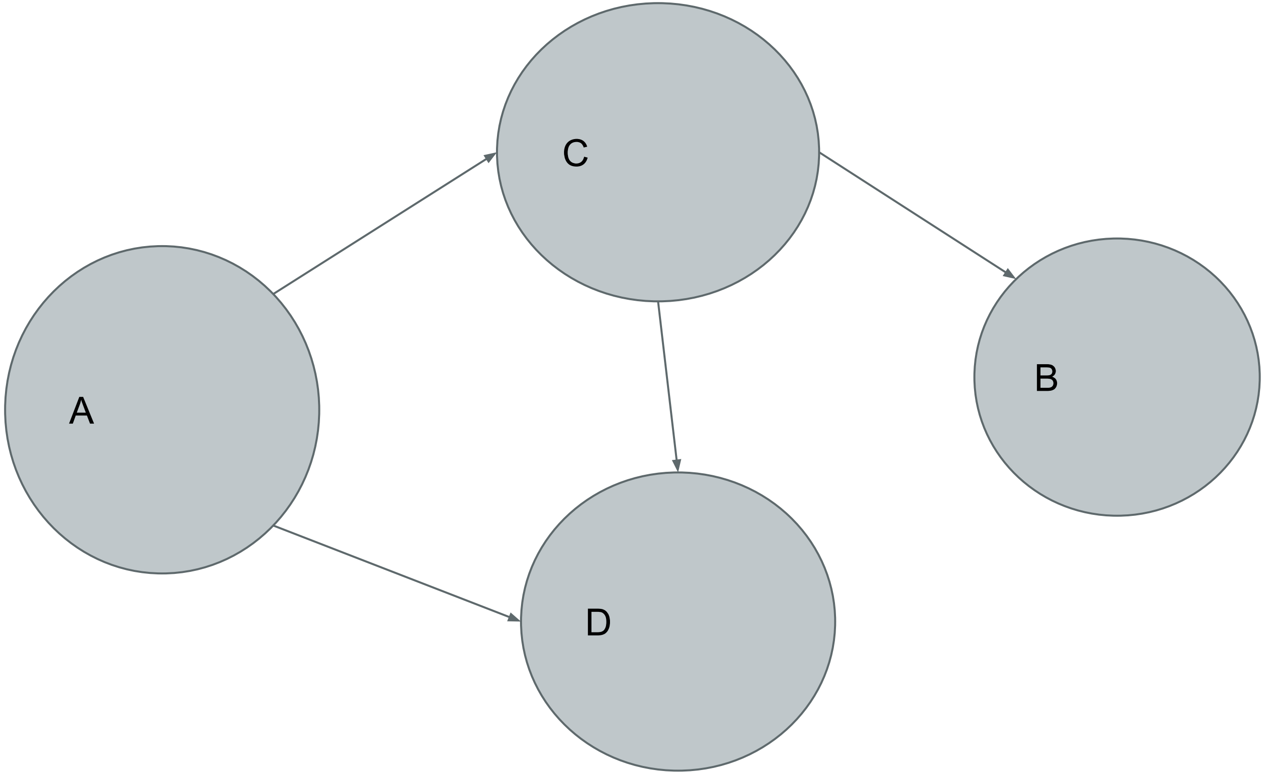
Ejemplo



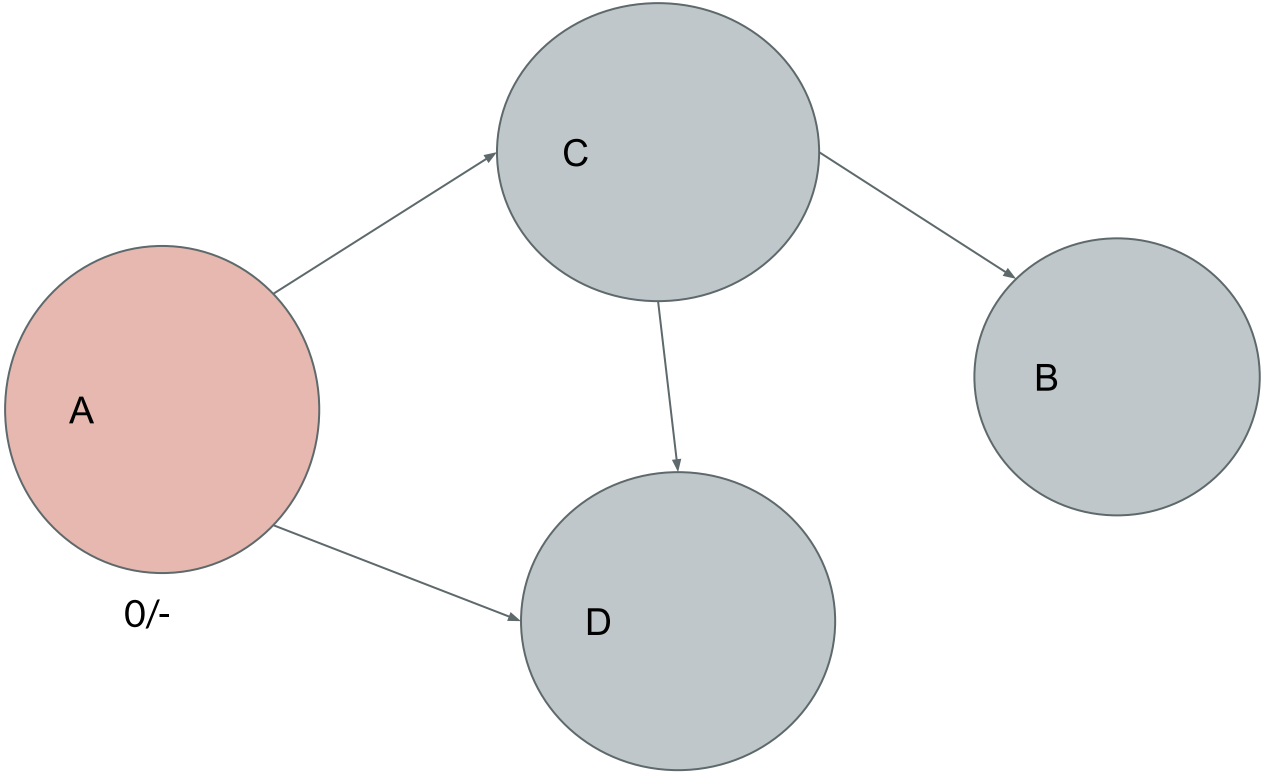
Ejemplo

Costo + camino

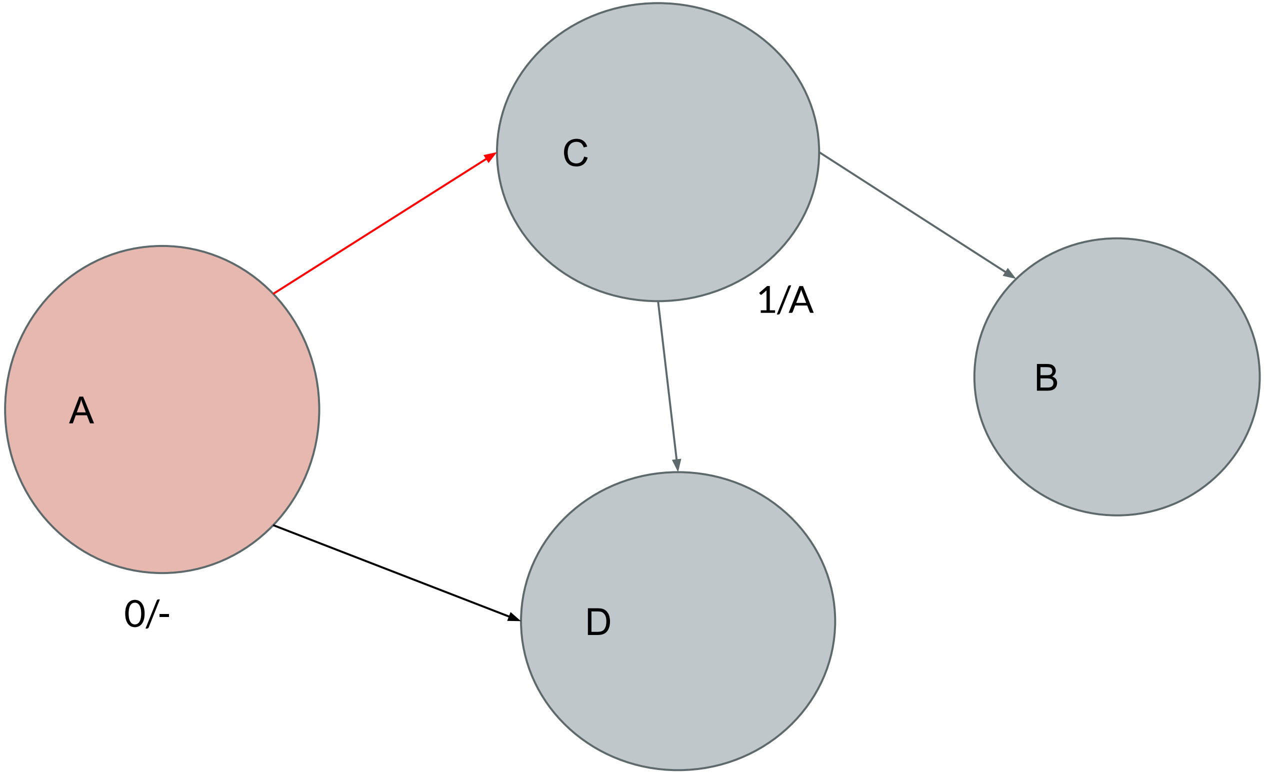
Ejemplo



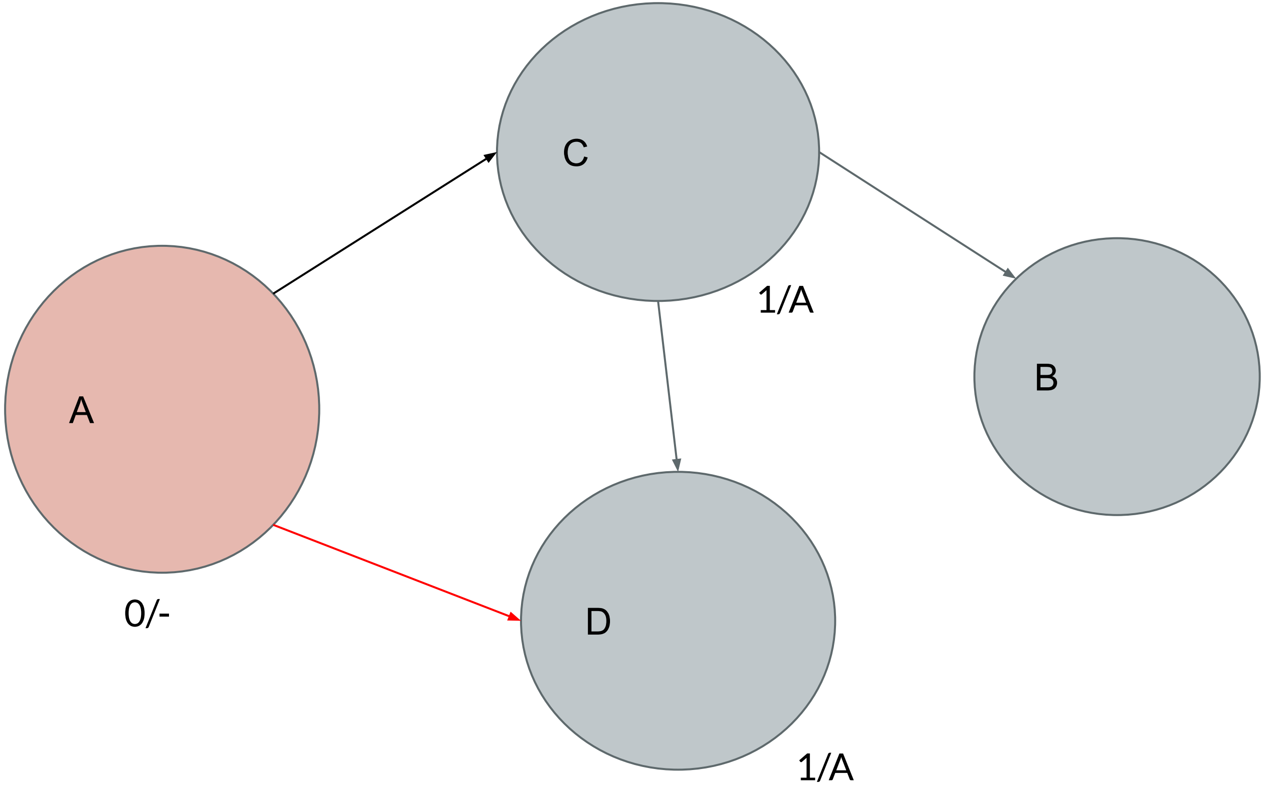
Ejemplo



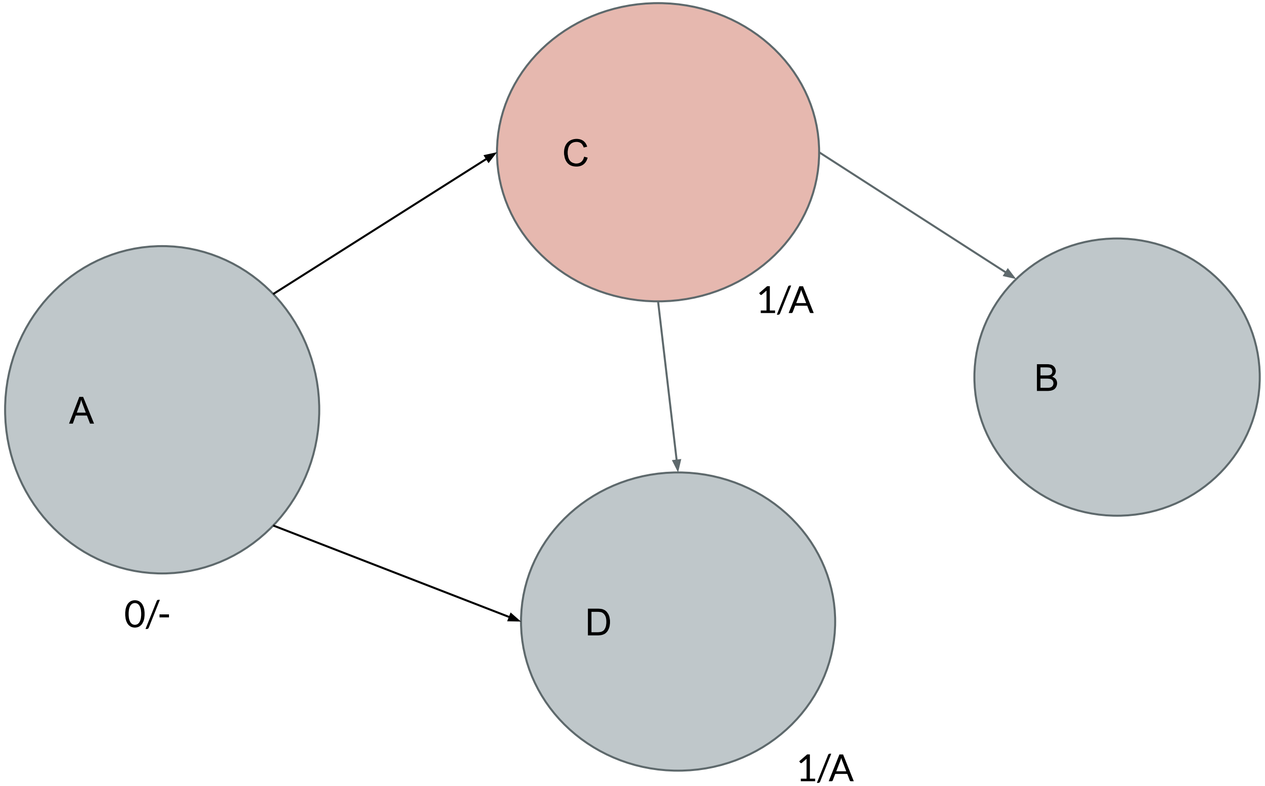
Ejemplo



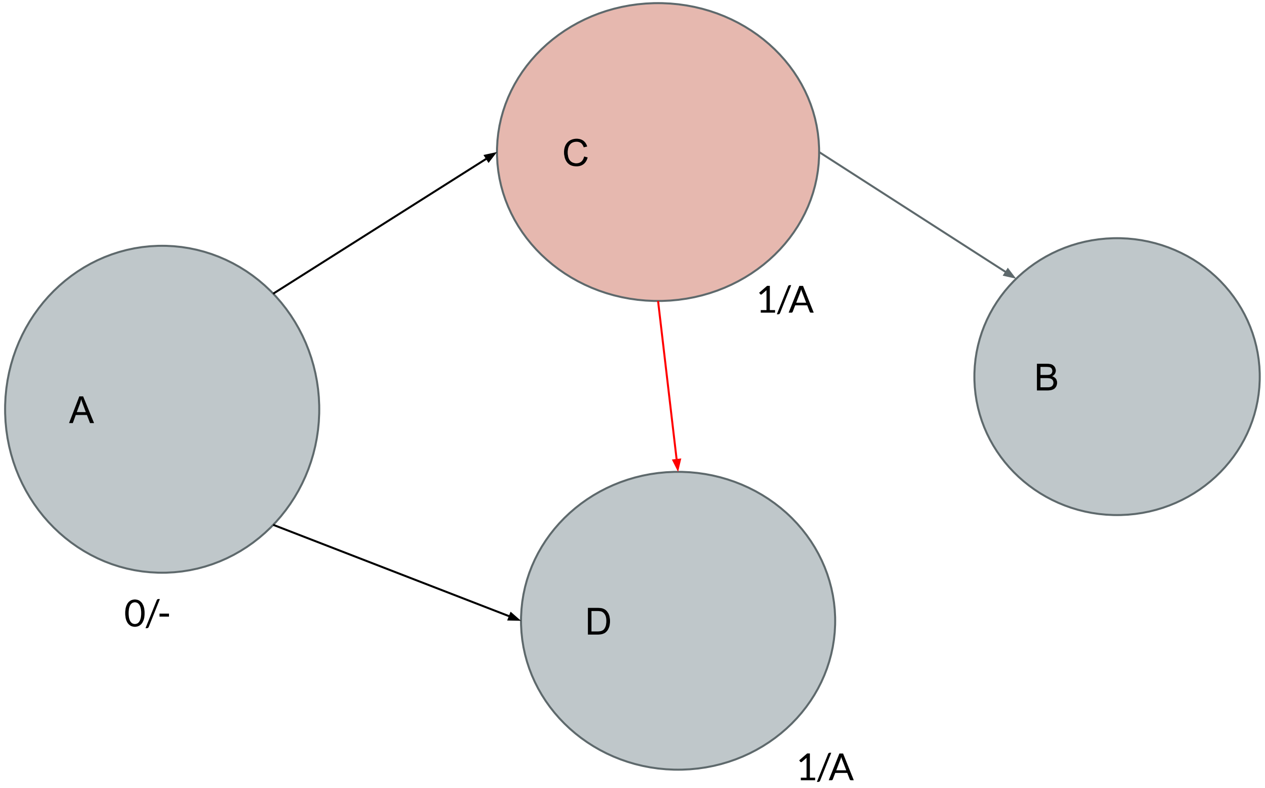
Ejemplo



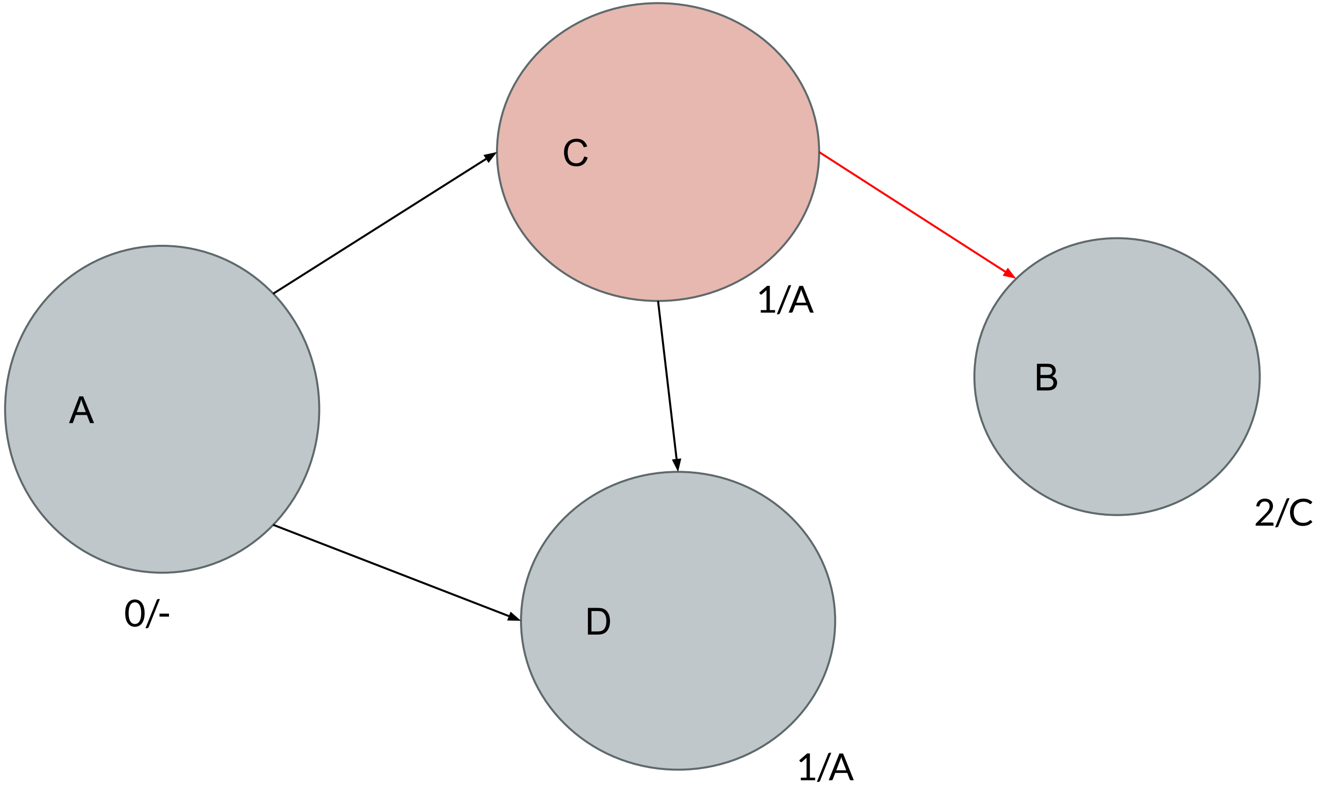
Ejemplo



Ejemplo



Ejemplo



Otro ejemplo

Otro ejemplo

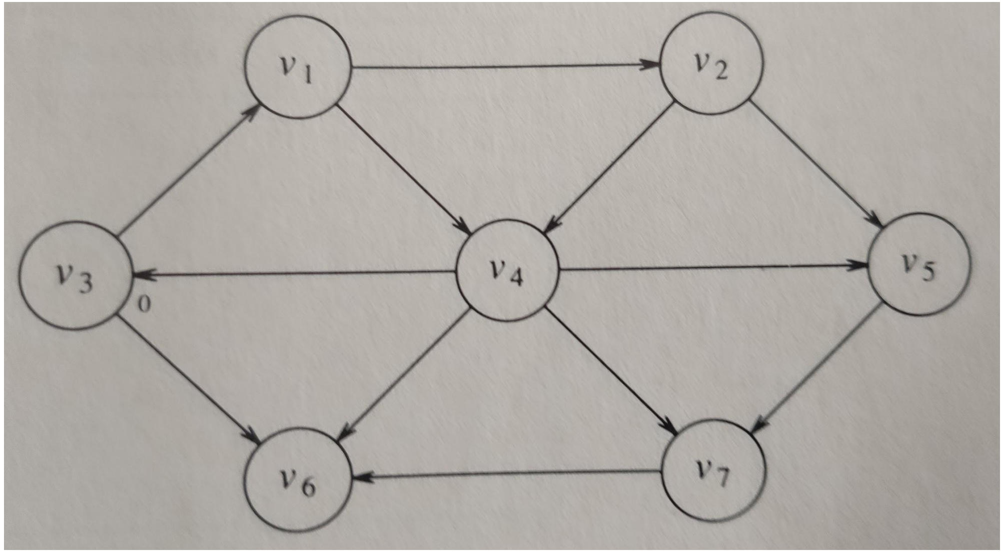


Tabla de trabajo:

[Ejercicio Camino más corto no ponderado]

Pseudocódigo

Pseudocódigo

```
void CaminoMasCortoNoPonderado(int origen) {
    int * visitados =  initVisitados(); // array de V casilleros, empiezan todos en false
    int * costo = initCosto(origen); // array de V casilleros, todos con valor “INF” a excepción de origen (con 0)
    int * vengo = initVengo(); // array de V casilleros, todos con valor -1

    Cola<int> cola;
    cola.encolar(origen);

    while (!cola.EsVacia()) {
        int vertice = cola.desencolar();
        visitados[vertice]= true; // lo marco como visitado
        paraCada w adyacenteA vertice
            if(costo[w] > costo[vertice] + 1) // solo procesamos los que no hayan sido visitados antes
                cola.encolar(w);
                costo[w] = costo[vertice] + 1;
                vengo[w] = vertice;

    // ... Utilizo las tablas, por ejemplo imprimir camino o saber el costo deseado.
```