Índice

Qué es un grafo

Un grafo es una estructura de datos, que se compone de

- Un número finito de vértices (o nodos)
- Un número finito de pares ordenados (u, v) de aristas que conectan los vértices

Hay dos tipos de grafos, los dirigidos (o digrafos) y los no dirigidos, veremos esto en profundidad más adelante, pero como primera aproximación podemos decir que en un grafo no dirigido, se puede recorrer la arista en cualquier sentido, para un lado o para el otro. Sin embargo en los digrafos, solo se puede ir en un sentido.

Algunas definiciones

- **Arista bucle**: arista de un solo punto extremo
- **Aristas paralelas**: son dos o más aristas conectadas a los mismos puntos extremos
- **Vértices adyacentes**: son dos vértices que se conectan por una (o más) arista. En el caso de los bucles, el vértice es adyacente a sí mismo
- **Aristas adyacentes**: son dos aristas que inciden en el mismo vértice
- **Vértice aislado**: es un vértice que no incide en ninguna arista
- **Grafo completo**: es un grafo simple de n vértices con una arista que conecta cada par de vértices
    - Si hay 1 vértice, hay 0 aristas
    - Si hay 2 vértices, hay 1 arista
    - Si hay 3 vértices, hay 3 aristas
    - Si hay 4 vértices, hay 6 aristas
    - Si hay 5 vértices, hay 10 aristas
    - Si hay 6 vértices, hay 15 aristas
    - Si hay n vértices, hay (n - 1) * n / 2
- **Grado** del vértice: la cantidad de aristas que indicen en él
    - Un bucle se cuenta como doble
    - La suma de los grados de todos los vértices es igual al doble de la cantidad de aristas
- **Camino**: la sucesión finita de vértices adyacentes y aristas
- **Sendero**: es un camino donde no se repiten las aristas
- **Trayectoria**: es un sendero que no repite vértices
- **Camino cerrado**: comienza y termina en el mismo vértice
- **Circuito**: es un camino cerrado que no repite aristas
- **Circuito simple**: es un circuito que no repite vértices (salvo el primero y el último)

Conectividad

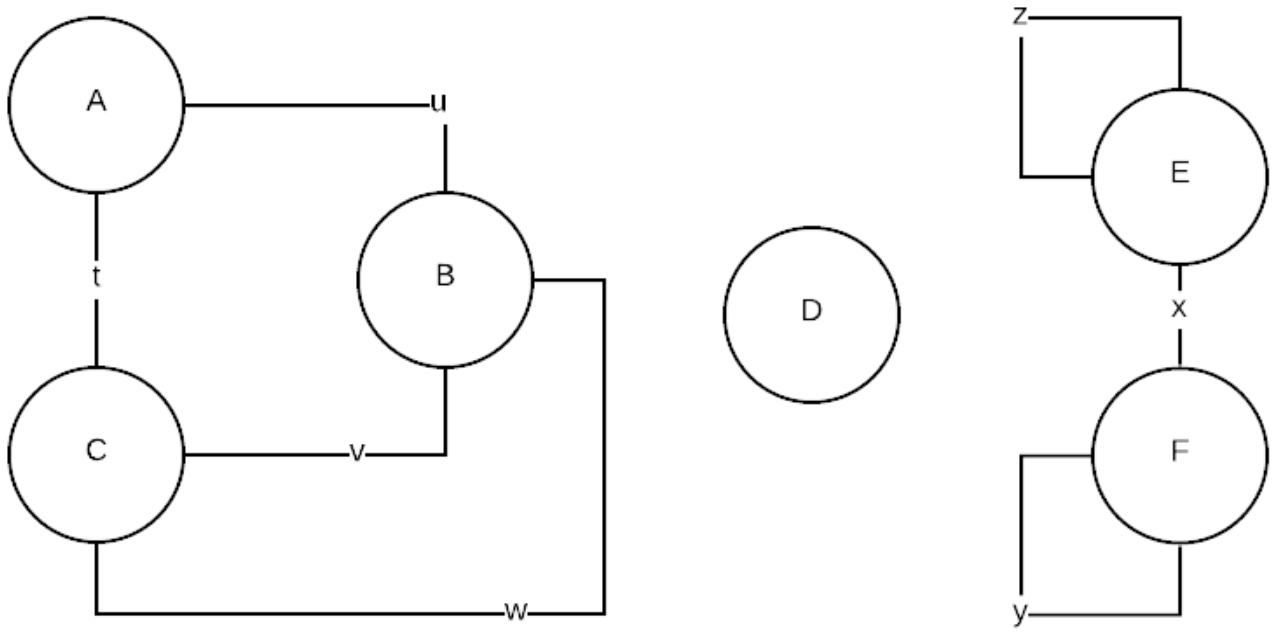
- **Vértices conexos**: sea G un grafo, y u y v dos vértices de G, se dice que u y v son conexos si y solo si
- **Grafo conexo**: un grafo es conexo cuando

Notas:

- *Si un grafo es conexo y tiene un circuito, se puede sacar una arista y el grafo seguira siendo conexo*
- *Si un grafo es conexo y está libre de circuitos, es un **arbol***

- *Si un grado está libre de circuitos pero no es conexo, se llama **bosque** (varios árboles)*

Ejemplo



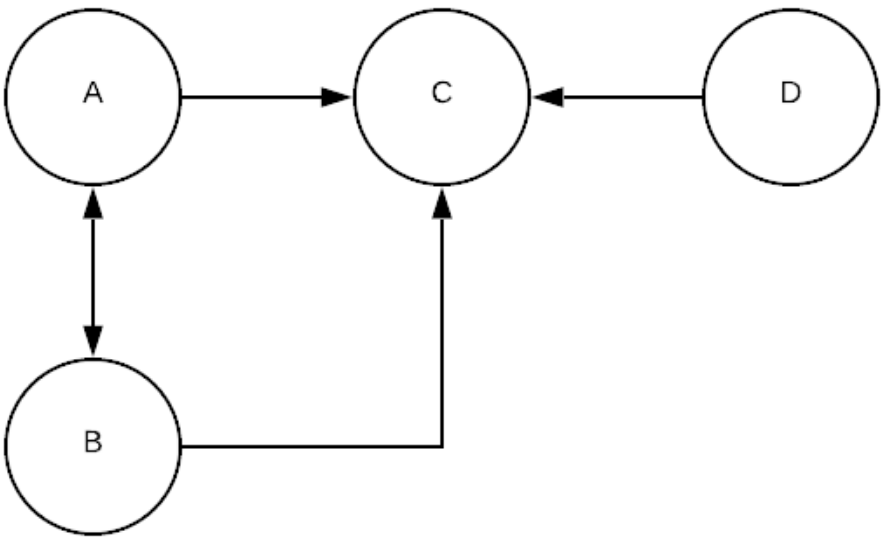
En la imagen anterior, podemos decir que

- **a** y **b** son vértices adyacentes
- **t** y **u** son aristas adyacentes
- **y** y **z** son bucles
- **v** y **w** son aristas paralelas
- **d** es un vértice aislado

Grafo dirigido o digrafo

Un grafo dirigido o digrafo consiste en una dupla formada por un conjunto V de vértices, y un conjunto de pares ordenados A (aristas orientadas) pertenecientes a VxV .

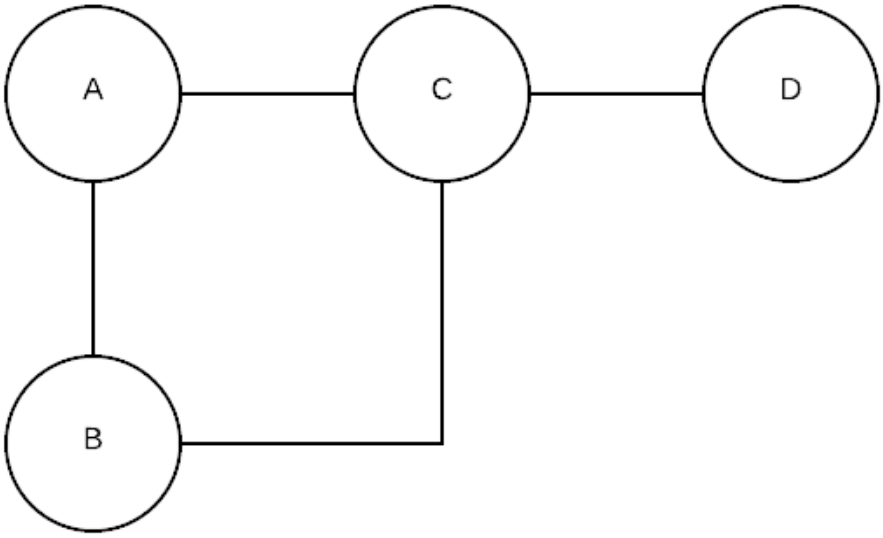
En símbolos el grafo dirigido G es G = (V ; A) donde A es un subconjunto de V xV (aristas orientadas o dirigidas).



Grafo no dirigido

Un grafo no dirigido (o no orientado) es una dupla formada por un conjunto V de vértices o nodos del grafo, y un conjunto de pares no ordenados A (aristas no orientadas) pertenecientes a V xV . La relación establecida entre los vértices es simétrica.

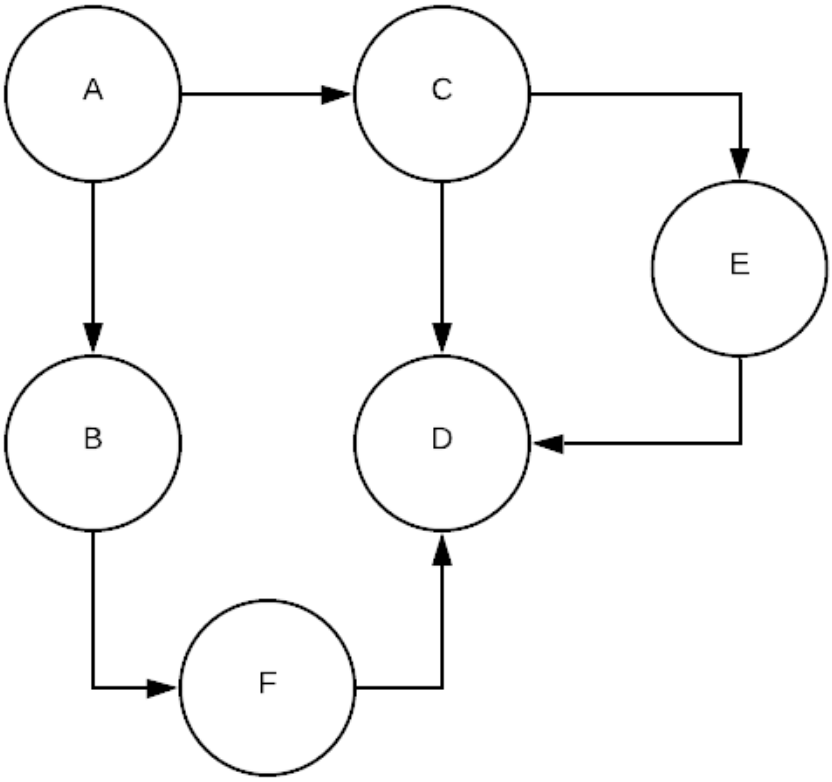
En símbolos el grafo dirigido G es G = (V ; A) donde A es un conjunto de pares no ordenados de V xV (Aristas no orientadas o no dirigidas). Esto significa que si hay un camino o modo de llegar desde F hasta G, también lo habrá de G a F .



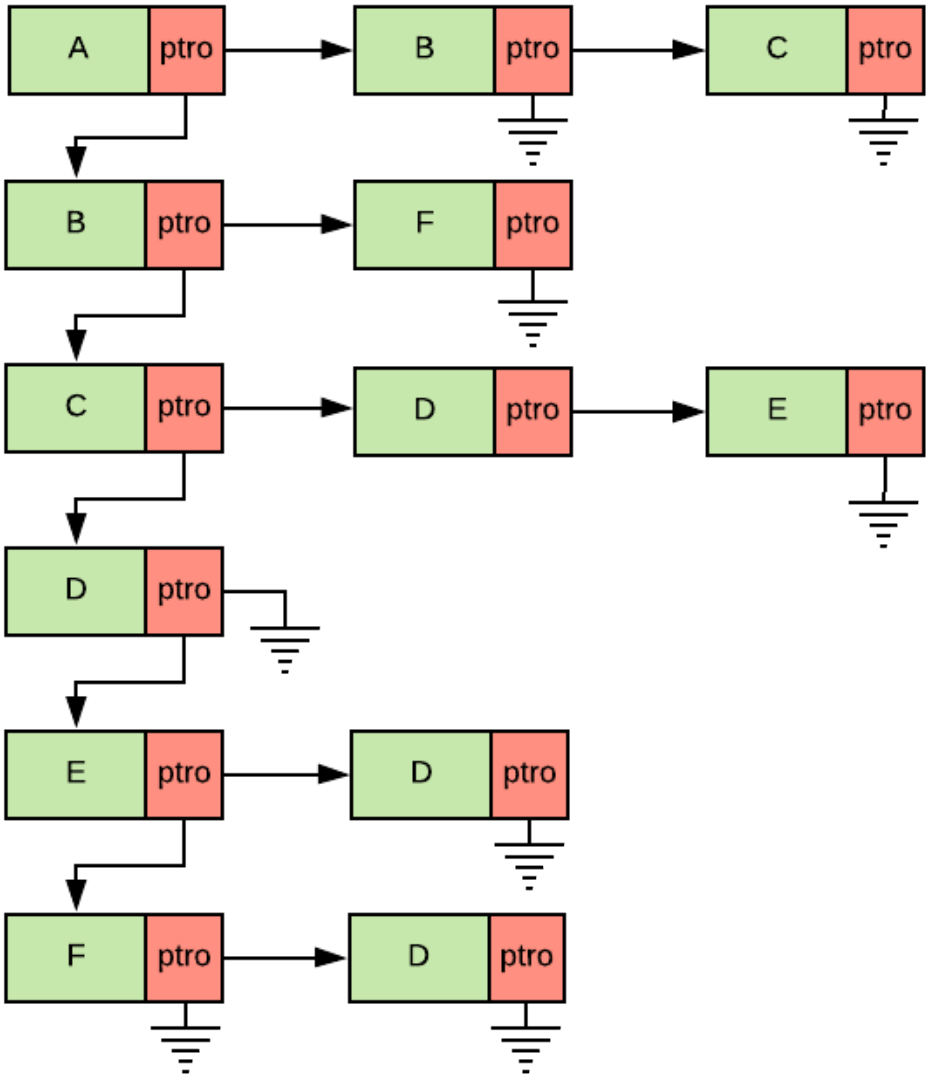
Estructuras para la implementación

Lista de adyacencias

Se representa el grafo como una lista de listas, en donde cada vértice tiene una lista de vértices vecinos. Para el grafo:

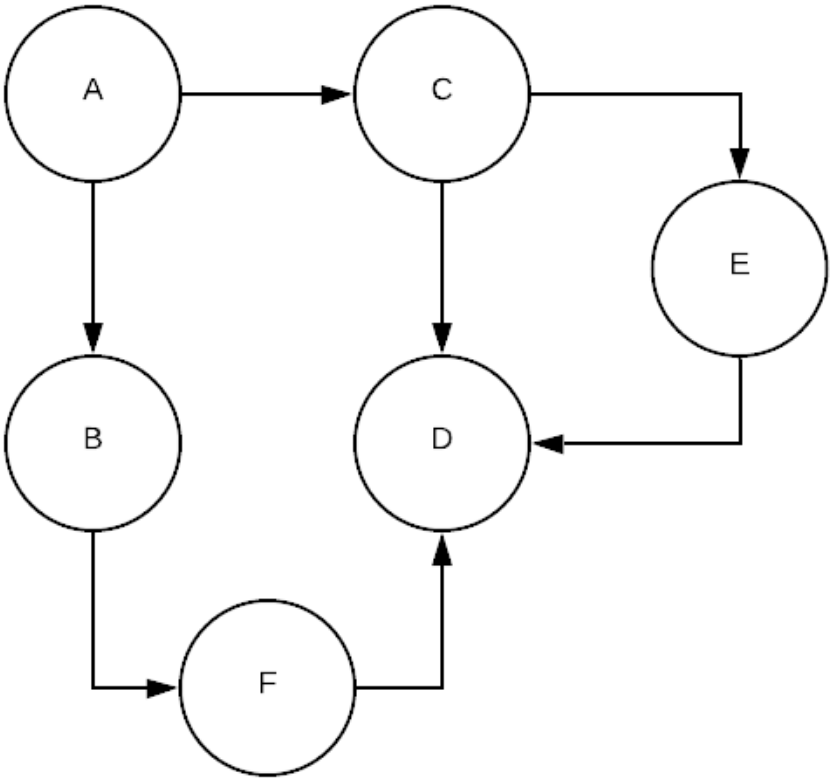


La lista de adyacencias es



Matriz de adyacencias

Se representa el grafo con una matriz que verifica lo siguiente: *si M[i] [j] = true, entonces la arista (i , j) pertenece al grafo*. Para el grafo:



La matriz de adyacencias es

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	0	0	0	1
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	0
F	0	0	0	1	0	0

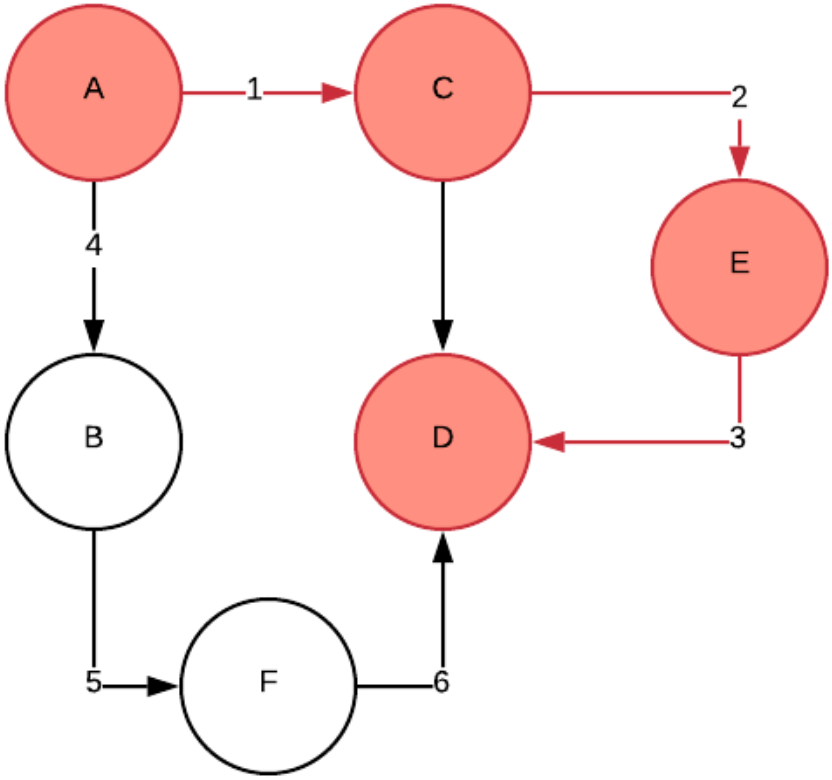
En el caso de ser un grafo o digrafo pesado, se reemplazan los 1 por el peso y los 0 por el nulo que corresponda.

Recorridos

En profundidad - DFS

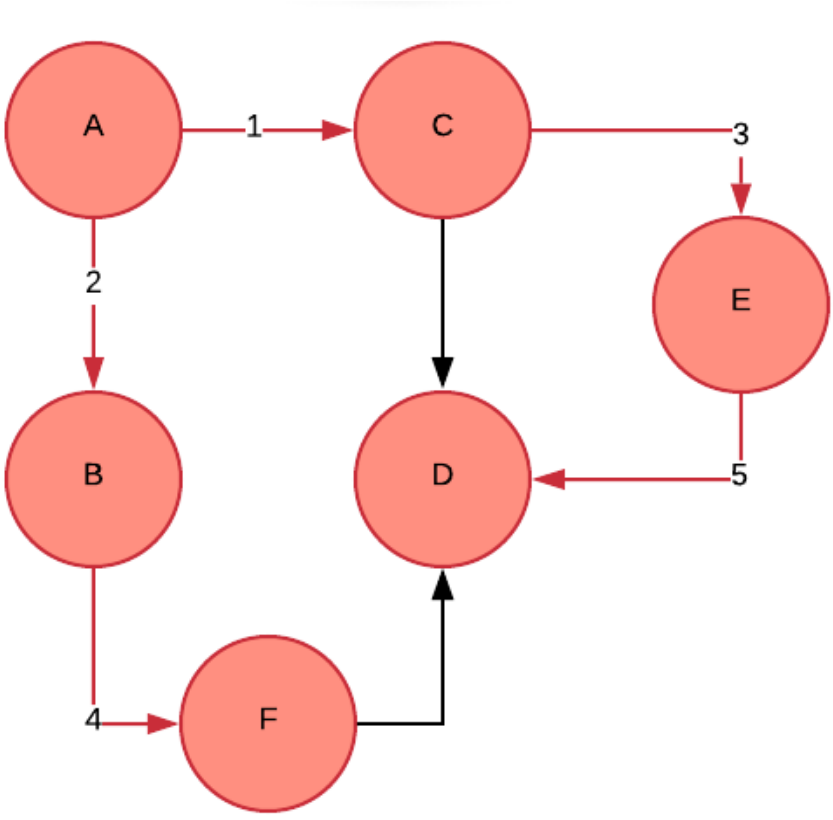
Depth first search o “busqueda primero en profundidad” consiste en para cada vértice del grafo que no ha sido visitado previamente, visitarlo y luego pasar al primer adyacente no visitado, luego al primer adyacente del adyacente que no haya sido visitado, y así hasta llegar a un nodo que no tenga adyacentes no visitados. Entonces se retrocede para pasar al siguiente

adyacente del anterior vértice y realizar el mismo proceso hasta agotar los adyacentes no visitados. El retroceso se hace cuando los vértices se agotan. Para implementar el algoritmo suele usarse una **pila** o un **algoritmo recursivo**.



En anchura - BFS

Breadth first search o "búsqueda primero en anchura" consiste visitar todos los nodos por niveles. Por cada nodo no visitado, se lo visita (nivel 0) y marca, y luego se visita a sus nodos adyacentes (nivel 1). Por cada nodo adyacente luego se visita a los adyacentes de los mismos(nivel 3), y así sucesivamente hasta visitar a todos los nodos. Para implementar el algoritmo suele usarse una **cola** o un **algoritmo recursivo**.

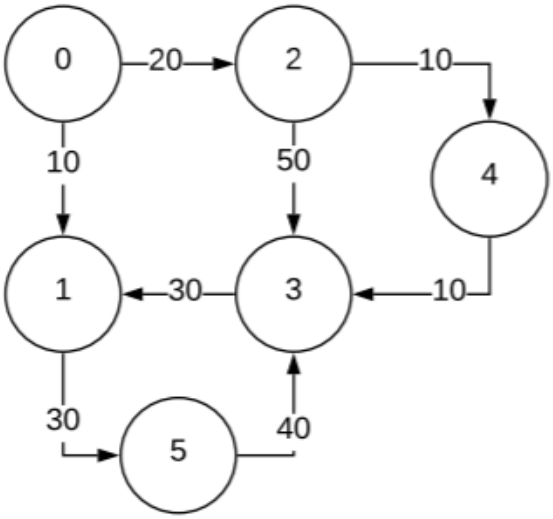


Recorridos mínimos

Dijkstra

Este algoritmo nos permite determinar la ruta con menor peso desde un origen hasta todos los vértices del grafo.

Supongamos el siguiente grafo



	0	1	2	3	4	5
0	∞	10	20	∞	∞	∞
1	∞	∞	∞	∞	∞	30
2	∞	∞	∞	50	10	∞
3	∞	30	∞	∞	∞	∞
4	∞	∞	∞	10	∞	∞
5	∞	∞	∞	40	∞	∞

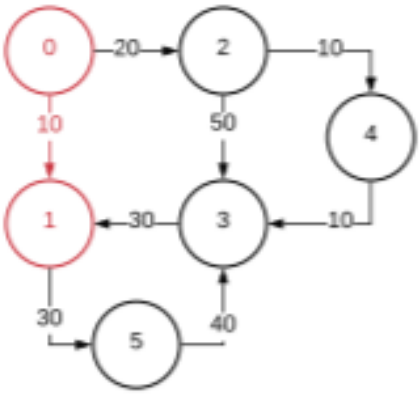
Vamos a tener que armar dos vectores, uno de tipo booleano y otro de tipo entero, que se van a inicializar así

0	1	2	3	4	5
true	false	false	false	false	false

0	1	2	3	4	5
∞	10	20	∞	∞	∞

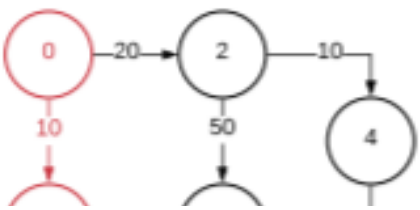
Y nos van a servir para poder ir comparando los pesos nuevos que vamos encontrando y para no hacer el mismo recorrido dos veces. La idea a medida que vamos recorriendo el grafo es ver si el nuevo valor para llegar a X es menor que el que esta guardado en nuestro vector, y en ese caso modificarlo.

Luego de inicializar los vectores empezamos a iterar

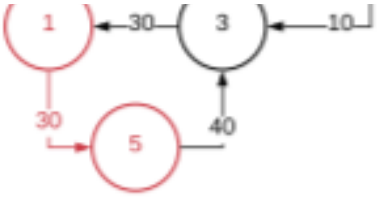


0	1	2	3	4	5
true	true	false	false	false	false

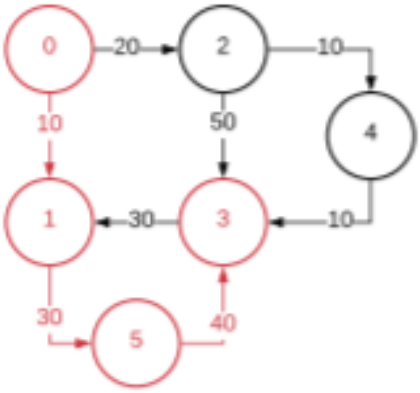
0	1	2	3	4	5
∞	10	20	∞	∞	∞



0	1	2	3	4	5
true	true	false	false	false	true

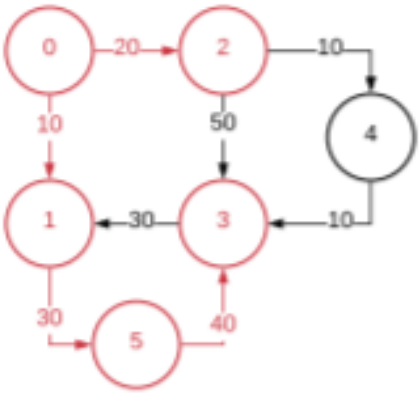


	1	2	3	4	5
∞	10	20	∞	∞	40



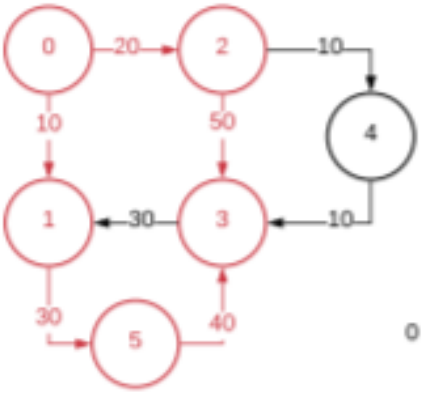
0	1	2	3	4	5
true	true	false	true	false	true

0	1	2	3	4	5
∞	10	20	80	∞	40



0	1	2	3	4	5
true	true	true	true	false	true

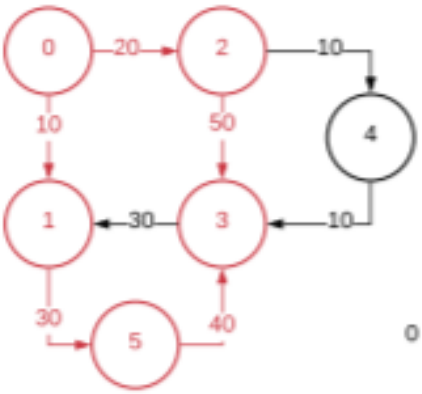
0	1	2	3	4	5
∞	10	20	80	∞	40



0	1	2	3	4	5
true	true	true	true	false	true

0	1	2	3	4	5
∞	10	20	70	∞	40

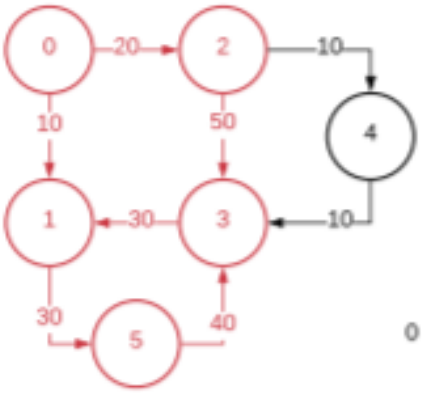
0 -> 2 -> 3: 20 + 50 = 70 como 70 < 80 se modifica



0	1	2	3	4	5
true	true	true	true	false	true

0	1	2	3	4	5
∞	10	20	70	∞	40

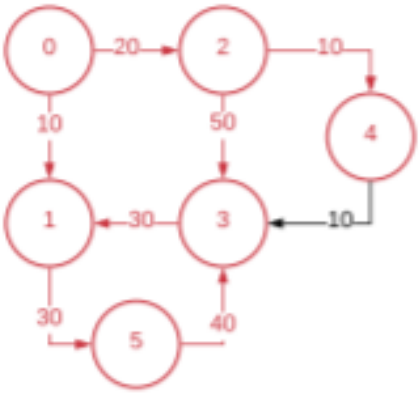
0 -> 2 -> 3: 20 + 50 = 70 como 70 < 80 se modifica



0	1	2	3	4	5
true	true	true	true	false	true

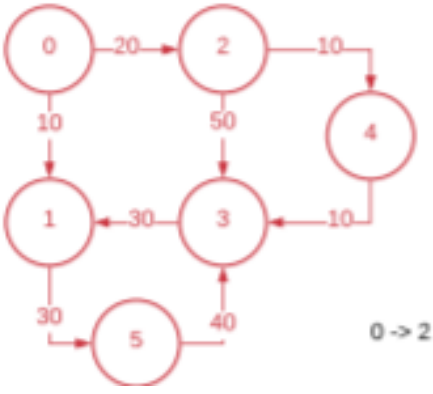
0	1	2	3	4	5
∞	10	20	70	∞	40

0 -> 2 -> 3 -> 1: 20 + 50 + 30 = 100 como 100 > 10 no se modifica



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

0	1	2	3	4	5
∞	10	20	70	30	40



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

0	1	2	3	4	5
∞	10	20	40	30	40

0 -> 2 -> 4 -> 3: 20 + 10 + 10 = 40 como 40 < 70 se modifica

Finalmente nos queda

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

0	1	2	3	4	5
∞	10	20	40	30	40

Entonces si queremos saber el menor costo desde 0 hasta 4 por ejemplo solo resta acceder a esa posicion del vector, y nos da 30.

xxxxxxxxxx

```
int* dijkstra (int origen, int** grafo, int elementos) {
    int distancias[elementos];
    bool visitados[elementos];

    for (unsigned i = 0; i < elementos; i++) {
        if (distancias[i] != origen)
            distancias[i] = INFINITO;
        visitados[i] = false;
    }

    for (unsigned i = 0; i < elementos - 1; i++) {
        int minPos = distanciaMinima(distancias, visitados);
        visitados[minPos] = true;
        for (unsigned j = 0; j < elementos; j++){
            int minDist = distancia[minPos] + grafo[min][j];
            if (!visitados[j] && distancia[min] != INFINITO && min < distancia[j])
                distancia[j] = minDist;
        }
    }
    return distancias;
}

int distanciaMinima(int distancias[], bool visitados[]) {
    int min = INFINITO;
    int indiceMin = 0;
    for (int i = 0; i < elementos; i++) {
        if (!visitados[i] && distancias[i] <= min) {
            min = distancias[i];
            indiceMin = i;
        }
    }
    return indiceMin;
}
```