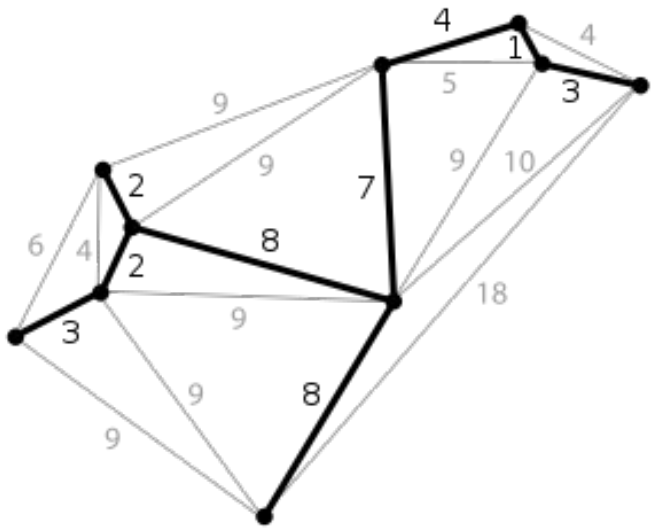
Grafos - parte 3

Árbol de cubrimiento mínimo

Prim & Kruskal

Árbol de cubrimiento mínimo

Dado un grafo no dirigido, ponderado y conexo, un árbol de cubrimiento mínimo es un subgrafo con **solo** las aristas necesarias para dejar el grafo conexo con el coste mínimo.

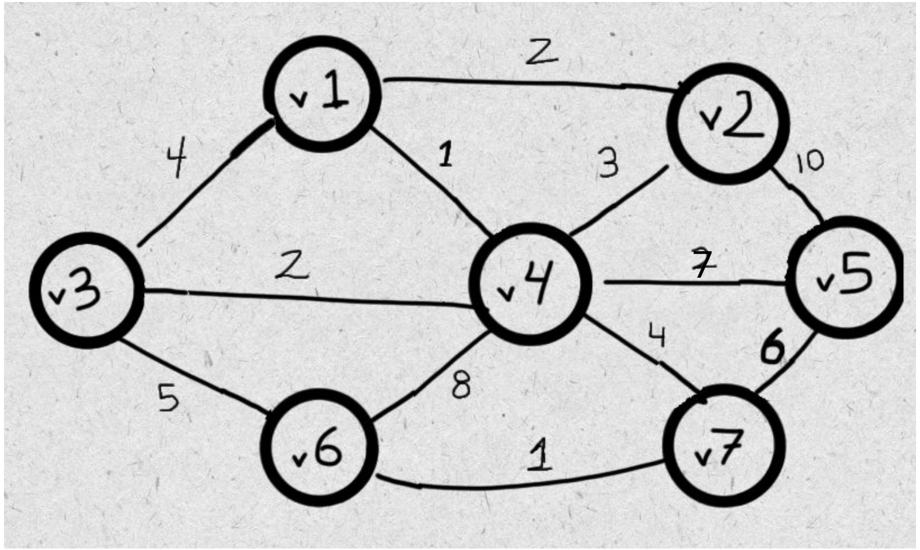


Ejemplo cables de teléfono

Prim

Algoritmo que nos sirve para descubrir el árbol de cubrimiento mínimo.

Esencialmente es un Dijkstra con diferencia de dos líneas de código.



Prim ejemplo1

Prim - otros ejemplos

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

Prim

```
void prim(int origen)
	int* visitados = initVisitados(); // array V casilleros, todos en false
	int* costos = initCostos(origen); // array V casillero, todos en INF menos origen con 0
	int* vengo= initVengo(); // array V casilleros, todos en -1
	for(int i=0; i<V; i++)
		int v = verticeDesconocidoDeMenorCosto(visitados, costos); // vértice a procesar
		visitados[v] = true;
		para cada w adyacente a v
			if(!visitados[w] && costos[w] > dist(v,w))
				costos[w] = dist(v,w);
				vengo[w] = v;
```

Prim

También nos podemos basar en las otras implementaciones de Dijkstra (V2) para mejorar el orden.

El orden es el mismo que Dijkstra (depende de su implementación).

A codificar

Kruskal

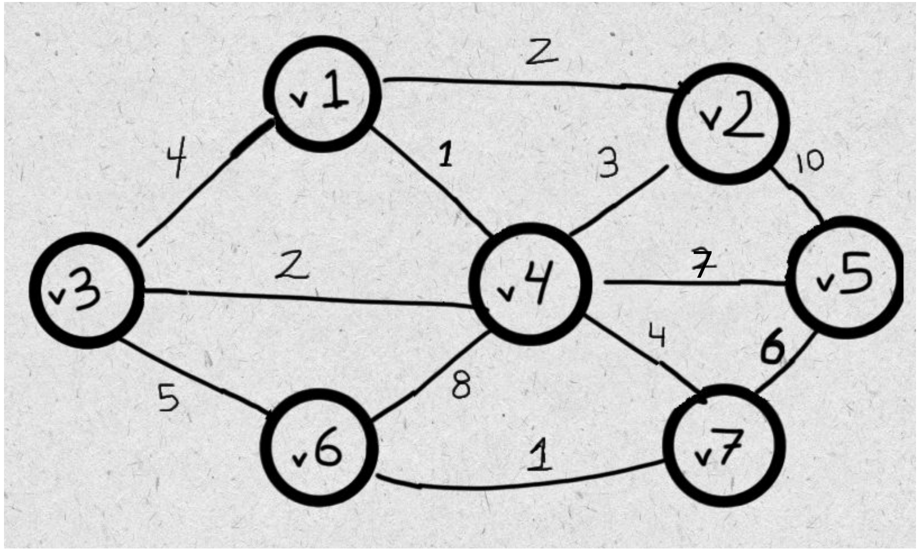
El algoritmo de Kruskal busca el mismo objetivo: ACM, pero con otra perspectiva.

En vez de ir agregando nodos como Prim, trata de agregar aristas.

En un principio se ordenan por peso y luego se recorren de esa manera (menos costo antes).

Se acepta usar esa arista siempre y cuando conecte dos árboles diferentes (no forme un ciclo).

Kruskal - ejemplo



[Kruskal ejemplo1]

Kruskal - otros ejemplos

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

Kruskal - pseudo v1

```
void kruskal() {
	Arista * aristas = initAristas(); // array de tamaño A, con todas la aristas del grafo
	bool * procesados =  initProcesados(); // array de tamaño A, todos los valores en false
	bool * aceptadas=  initAceptados(); // array de tamaño A, todos los valores en false

	for(int i=0; i<A; i++) {
		int indexMenor= obtenerMenorAristaNoProcesada(aristas, procesados);
		Arista a = aristas[indexMenor];
		procesados[indexMenor] = true;
		if(!formaCiclo(a)) {
			aceptadas[indexMenor] = true;
		}
	}
}
```

Kruskal - pseudo v2

```
void kruskal() {
	ColaPrioridad cp(); // imp. heap
	insertarAristas(cp); // inserto todas las aristas del grafo
	List<Arista> solucion(); // guardo las aristas que serán parte de la solución

	while(!cp.estaVacia()) {
		Arista a = cp.pop();// obtenemos la misma arista sin procesar
		if(!formaCiclo(solucion, a)) {
			solucion.agregar(a);
		}
	}
}
```

Kruskal - forma ciclo?

Kruskal es bastante trivial, sólo nos queda por cubrir cómo sabemos si forma un ciclo una nueva arista. Para ello utilizaremos una estructura auxiliar: **MFset**

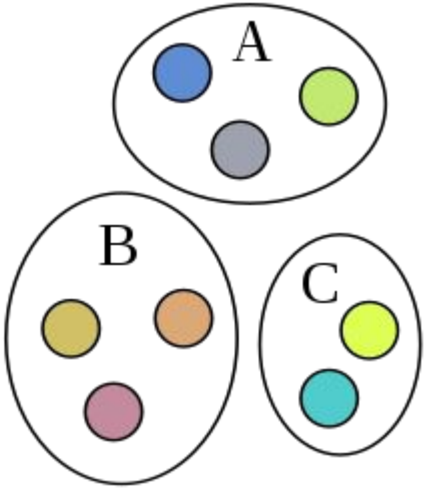
(

Merge-Find Set

También conocido como Disjoint-set/Conjuntos disjuntos.

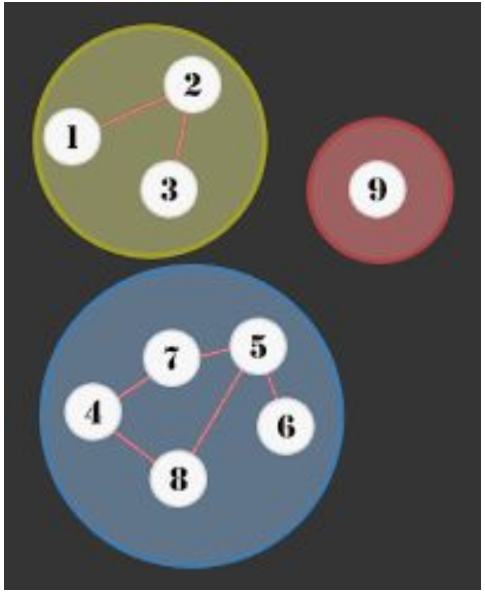
Es una estructura que nos permite saber a qué conjunto pertenece determinados elementos (**find**). En consecuencia saber si dos elementos distintos se encuentran dentro del mismo conjunto.

Además podremos unir diferentes conjuntos en uno solo (**merge**).



Merge-Find Set

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



A codificar

)

Kruskal - pseudo v3

```
void kruskal() {
	ColaPrioridad cp(); // imp. heap
	insertarAristas(cp); // inserto todas las aristas del grafo
	List<Arista> solucion(); // guardo las aristas que serán parte de la solución
	MFset mfset(V+1); // Merge-Find Set de V+1 elementos
	int aristasAceptadas = 0;

	while(!cp.estaVacia() || aristasAceptadas < V-1)
		Arista a = cp.pop();// obtenemos la misma arista sin procesar
		if(mfset.find(a.origen) != mfset.find(a.destino)) // !formaCiclo(solucion, a)
			mfset.merge(a.origen, a.destino);
			solucion.agregar(a);
			aristasAceptadas++;
```

Kruskal - orden

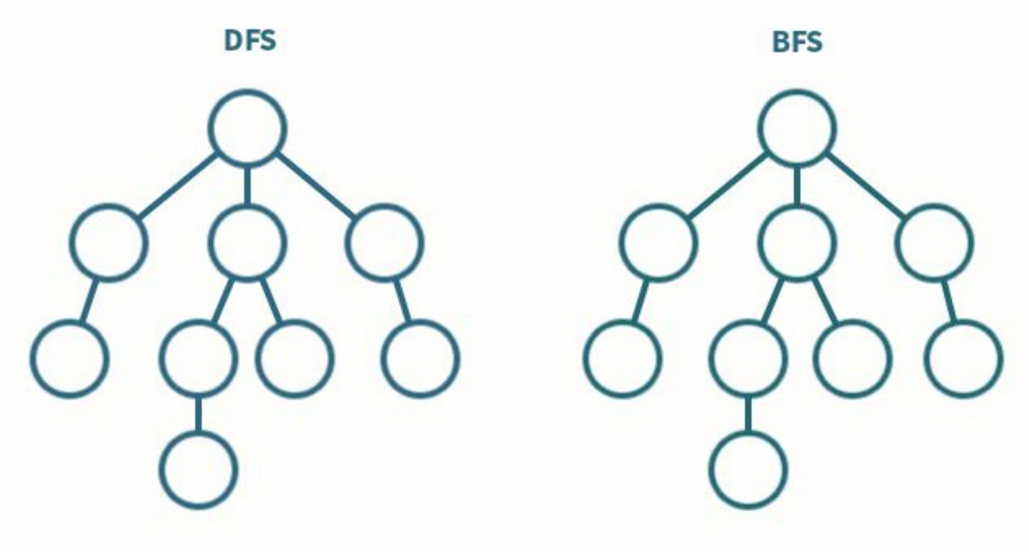
```
void kruskal() { // ALogV
	ColaPrioridad cp();
	insertarAristas(cp); // ALogA = ALog(V^2) = 2ALogV = ALogV
	List<Arista> solucion();
	MFset mfset(V+1); // V
	int aristasAceptadas = 0;

	while(!cp.estaVacia() || aristasAceptadas < V-1) // ALogA = ALog(V^2) = 2ALogV = ALogV
		Arista a = cp.pop(); // LogA
		if(mfset.find(a.origen) != mfset.find(a.destino)) // LogV -> Link
			mfset.merge(a.origen, a.destino); // LogV -> Link
			solucion.agregar(a); // 1
			aristasAceptadas++;
```

Recorridos/búsquedas

Amplitud(BFS) & Profundidad(DFS)

Búsquedas

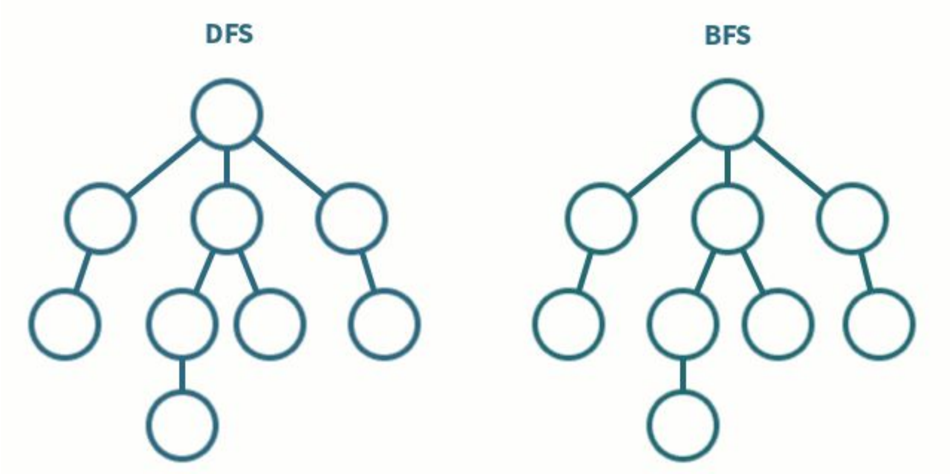
Son estrategias en las cuales se recorren los grafos desde un punto dado, tienen diferentes usos que veremos más adelante.



BFS/DFS

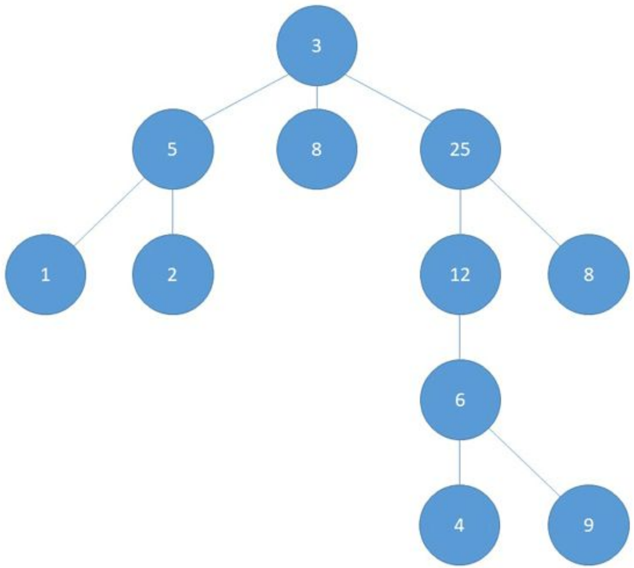
Las dos búsquedas más utilizadas son:

- Recorrido en amplitud / Breadth-first search (**BFS**)
- Recorrido en profundidad / Depth-first search (**DFS**)



BFS

La idea es recorrer desde el origen atravesando por niveles/pasos.



BFS - otros ejemplos

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

BFS - pseudo

```
void BFS(int origen)
    bool * encolados = initEncolados(); // array de V+1 pos, todos en false
    Cola c; //FIFO
    c.encolar(origen);
    encolados[origen] = true;
    while(!c.estaVacia())
        int v = c.desencolar();
        para todo w adyacente a v
            if(!encolados[w])
                c.encolar(w);
                encolados[w] = true;
```

BFS - Orden

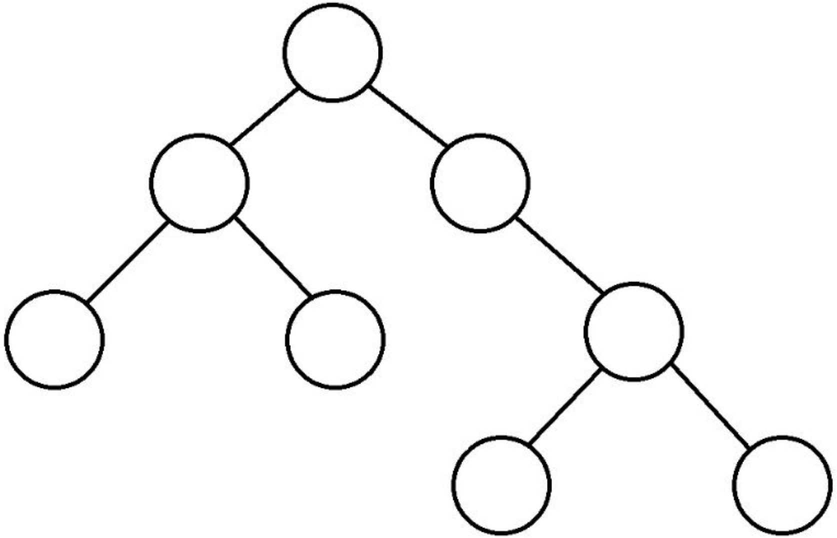
```
void BFS(int origen)
	bool * encolados = initEncolados(); // array de V+1 pos, todos en false
	Cola c; //FIFO
	c.encolar(origen);
	encolados[origen] = true;
	while(!c.estaVacia()) // O(V+A) análisis
		int v = c.desencolar();
		para todo w adyacente a v
			if(!encolados[w])
				c.encolar(w);
				encolados[w] = true;
```

BFS - usos

1. Sabes si dos vértices están conectados
2. Camino más corto en grafos no ponderados
3. Peer to Peer Networks: ayuda a encontrar nodos cercanos
4. Crawlers en Search Engines: para crear index desde una página
5. Social Networking Websites: encontrar personas a una distancia dada

DFS

La idea de la búsqueda en profundidad es recorrer y alejarse lo más posible antes de volver (backtracking).



DFS

Se puede implementar tanto con una PILA (LIFO) como con recursividad donde la pila está implícita por el stack de llamadas.

DFS - otros ejemplos

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

DFS - pseudo

```
void DFS(int origen, int * visitados)
	visitados[origen] = true;
	// procesar/imprimir origen
	para cada w adyacente a origen
		if(!visitados[w])
			DFS(w, visitados);
```

DFS - orden

```
void DFS(int origen, bool* visitados) // O(V+A) análisis
	visitados[origen] = true;
	// procesar/imprimir origen
	para cada w adyacente a origen
		if(!visitados[w])
			DFS(w, visitados);
```

DFS - usos

1. Saber si dos vértices están conectados
2. Se puede obtener el orden topológico de un grafo. Topological Sorting
3. Descubrir puntos de articulación. Biconnected graph
4. Hallar componentes fuertemente conexas en un grafo dirigido. Strongly Connected Components
5. Detectar ciclos. Detect Cycle in a Directed Graph

Resumen

Nombre	Propósito	Orden	Restricciones/comentarios
OT	Obtener el orden topológico de un grafo	O(V+A)*	No acepta ciclos (pero sirve para detectarlos)
Camino + corto con aristas no ponderadas	Camino + corto desde un origen	O(V+A)*	
Dijkstra	Camino + corto desde un origen	O(V^2) O(ALogV)* con heap	No acepta aristas negativas
Camino + corto con aristas negativas	Camino + corto desde un origen	O(V.A)*	
Floyd	Camino + corto entre todos los pares de vértices	O(V^3)	Acepta aristas negativas
Warshall	Matriz de Clausura Transitiva	O(V^3)	Nos puede servir para saber la conexidad de un grafo
Prim	Árbol de cubrimiento mínimo	O(V^2) O(ALogV)* con heap	Muy parecido a Dijkstra
Kruskal	Árbol de cubrimiento mínimo	A(A+V) O(ALogV)* con heap & MFset (mejorado)	
BFS	Múltiples	O(V+A)*	
DFS	Múltiples	O(V+A)*	

(*) con Lista de Ady