Teoría de Grafos

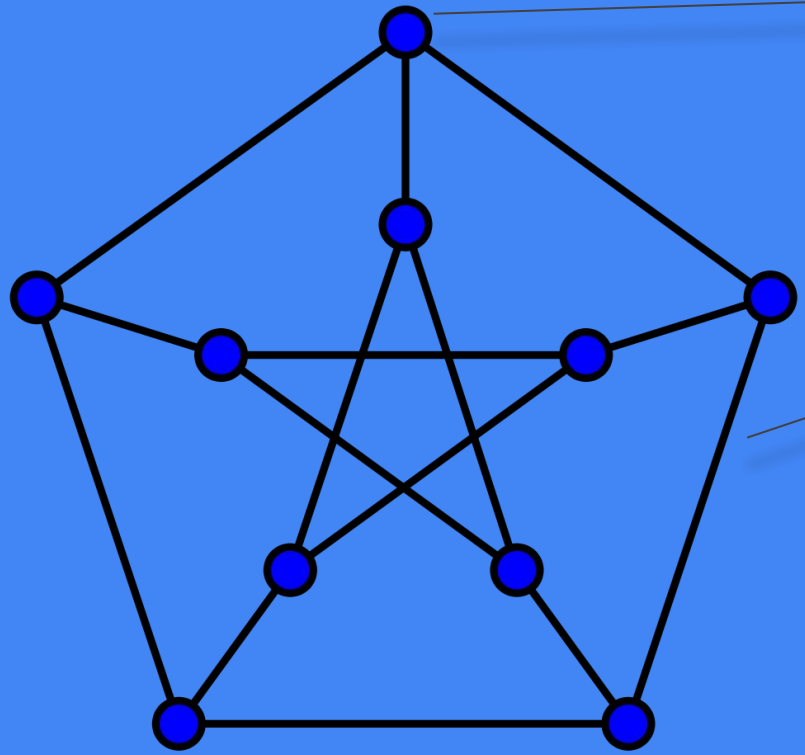
Curso Introducción a la Programación Competitiva 2019

Qué es un grafo?

Un grafo es un conjunto no vacío de objetos llamados nodos o vértices, y una selección de pares de vértices llamados aristas o lados.

Osea, una cosa que tiene puntos y que se unen con líneas.

Algo como esto:



NODO O VÉRTICE

ARISTA O LADO

Algunas definiciones

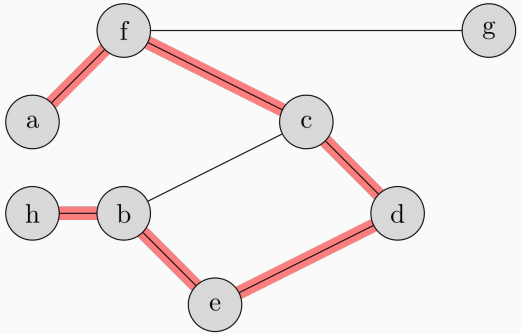
- Camino
- Longitud de un camino
- Grado de un vértice
- Componentes Conexas
- Distancia mínima

Camino

Un camino entre dos nodos U y W es una lista de vértices v0=U, v1, … , vk=W

tal que hay una arista entre v(i) y v(i+1) para todo i<k

Ejemplo:

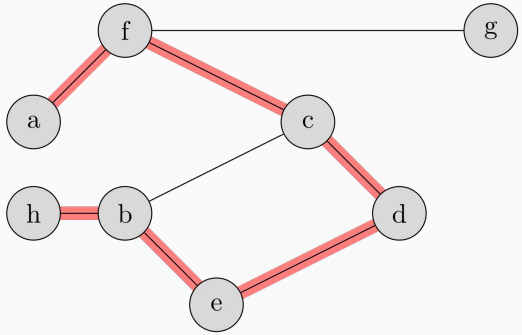


a,f,c,d,e,b,h es un camino de a hasta h

y a,f,c,b,h también lo es

Longitud de un camino

La longitud de un camino es la cantidad de aristas o lados por los que paso para llegar de un vertice U a otro W. Es decir, en el ejemplo anterior:



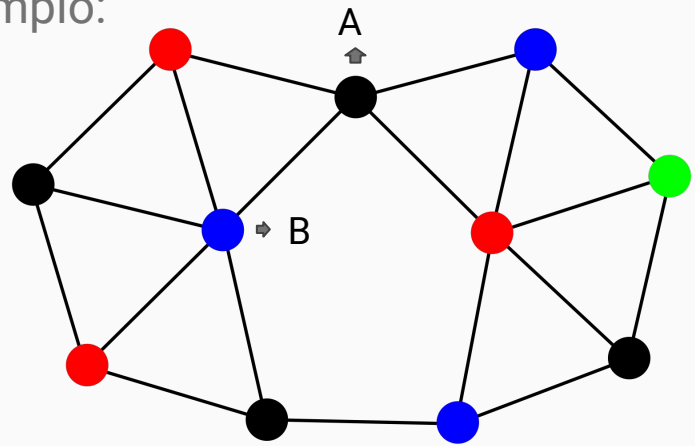
long(<a,f,c,d,e,b,h>) = 6

long(<a,f,c,b,h>) = 4

Grado de un vértice

El grado de un vértice V es la cantidad aristas que salen de V hacia otros nodos

Ejemplo:

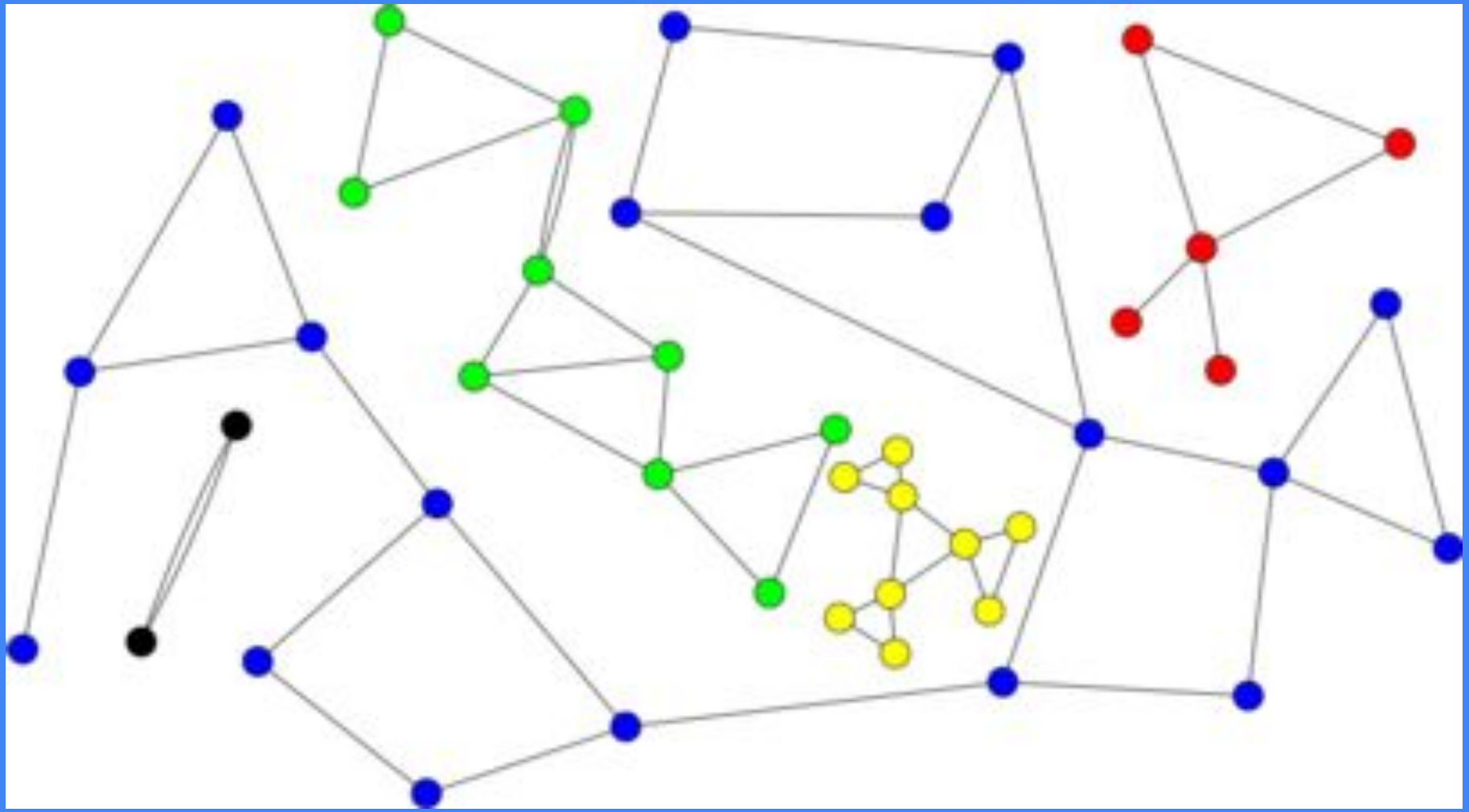


grado(A) = 4

grado(B) = 5

Componentes conexas

Sea G un grafo no dirigido. Las componentes conexas son subgrafos de G tales que de un nodo cualquiera del subgrafo yo puedo llegar a cualquier otro nodo moviendome por aristas del subgrafo. Mejor veamos un ejemplo:

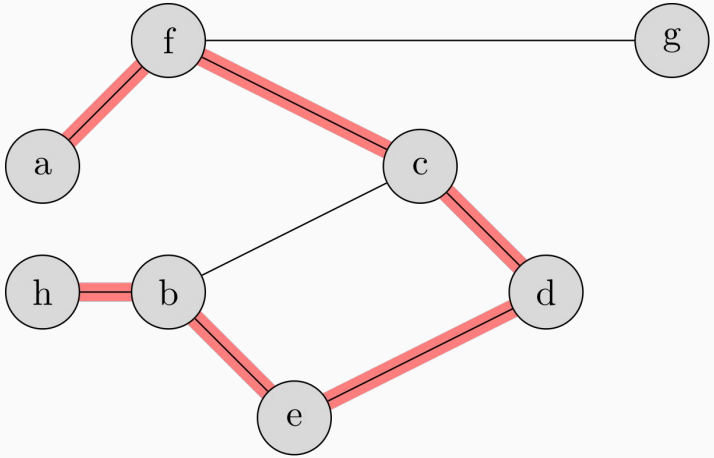


Distancia mínima

La distancia mínima entre dos vértices V y W se define como el menor número n tal que existe un camino entre V y W de longitud n. Si no existe un camino, decimos que la distancia es infinito.

Distancia mínima

Volvamos a uno de los primeros ejemplos:



minDist(a,h) = 4

minDist(b,c) = 1

minDist(g,b) = ????

Tipos de Grafos

- Conexos - No Conexos
- Dirigidos - No dirigidos
- Ponderados - No ponderados
- Árboles

Conexo vs No Conexo

Un grafo se dice conexo si la cantidad de componentes conexas es igual a 1

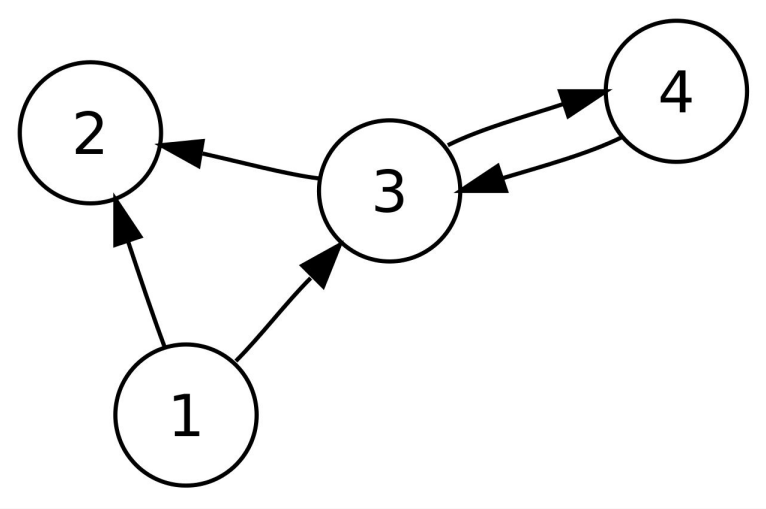
Por lo contrario un grafo es No Conexo si la cantidad de componentes

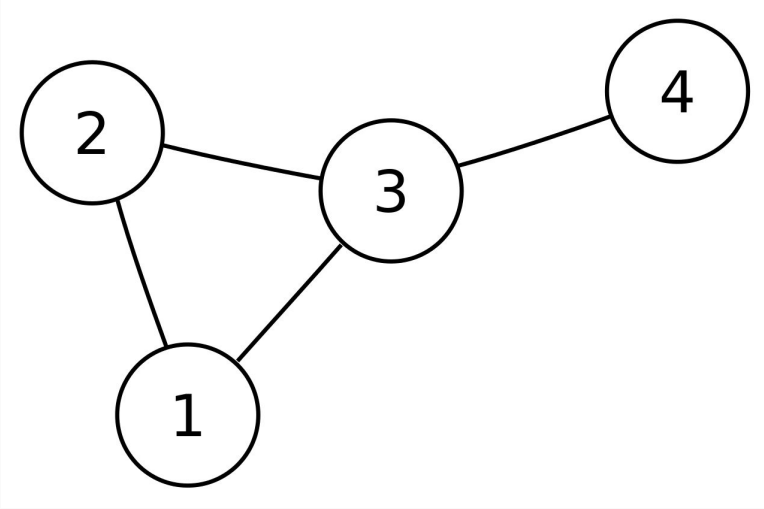
conexas es > 1

Dirigidos vs No Dirigidos

Hasta ahora siempre hablamos de grafos No Dirigidos, es decir las aristas que van de V a W significan que puedo ir tanto de V a W como de W a V. Ahora introducimos el concepto de grafo dirigido, es decir las aristas V a W significa que puedo ir de V a W, pero no puedo ir de W a V a no ser que la arista W a V exista

Dirigidos vs No Dirigidos

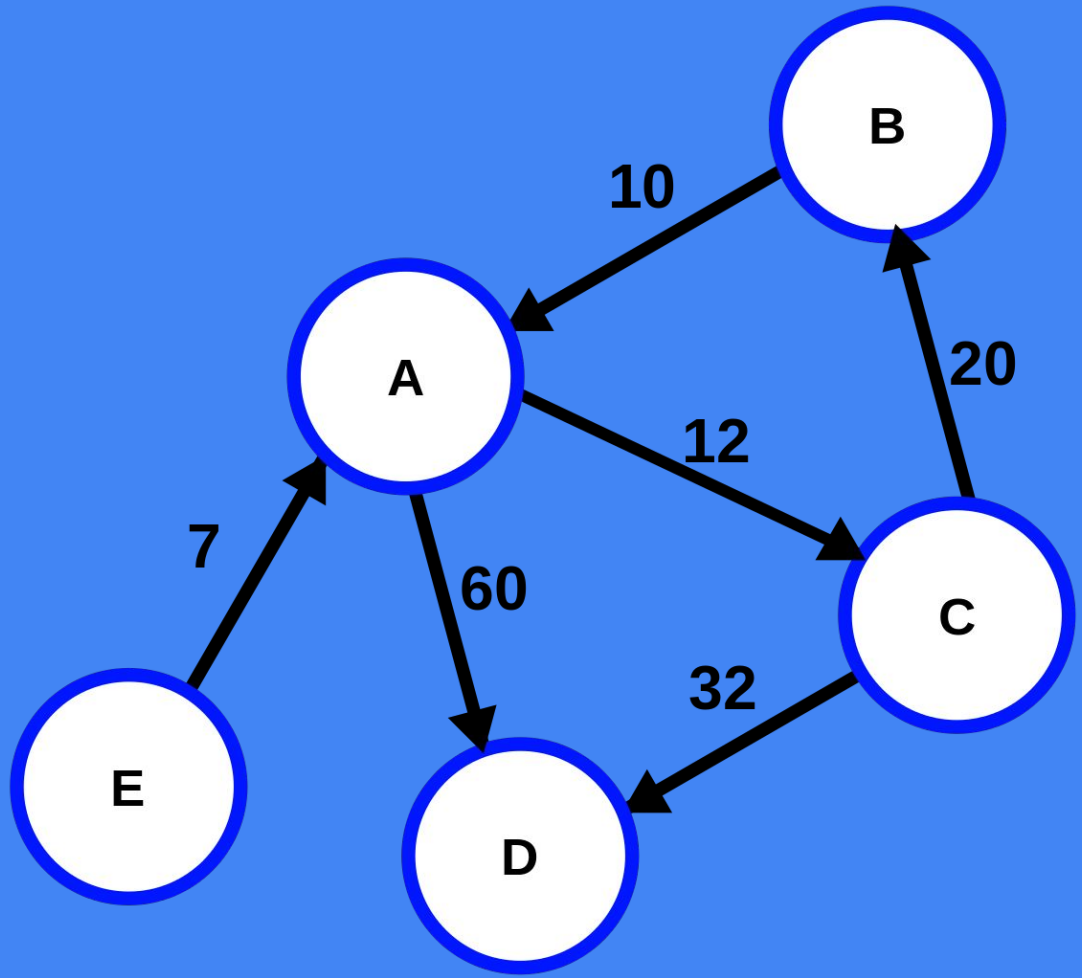




Ponderados vs No Ponderados

Hasta ahora solo estudiamos grafos no ponderados. Un grafo es ponderado si las aristas además tienen pesos. Es decir “valores arribita”.

Veamos ejemplos


B
10
A
20
12
7
60
C
32
E
D

Ponderados vs No Ponderados

Esto introduce una nueva definición de distancia mínima entre nodos,

si el grafo es ponderado, la distancia va a ser la suma de los valores de las

aristas por las que paso en el camino.

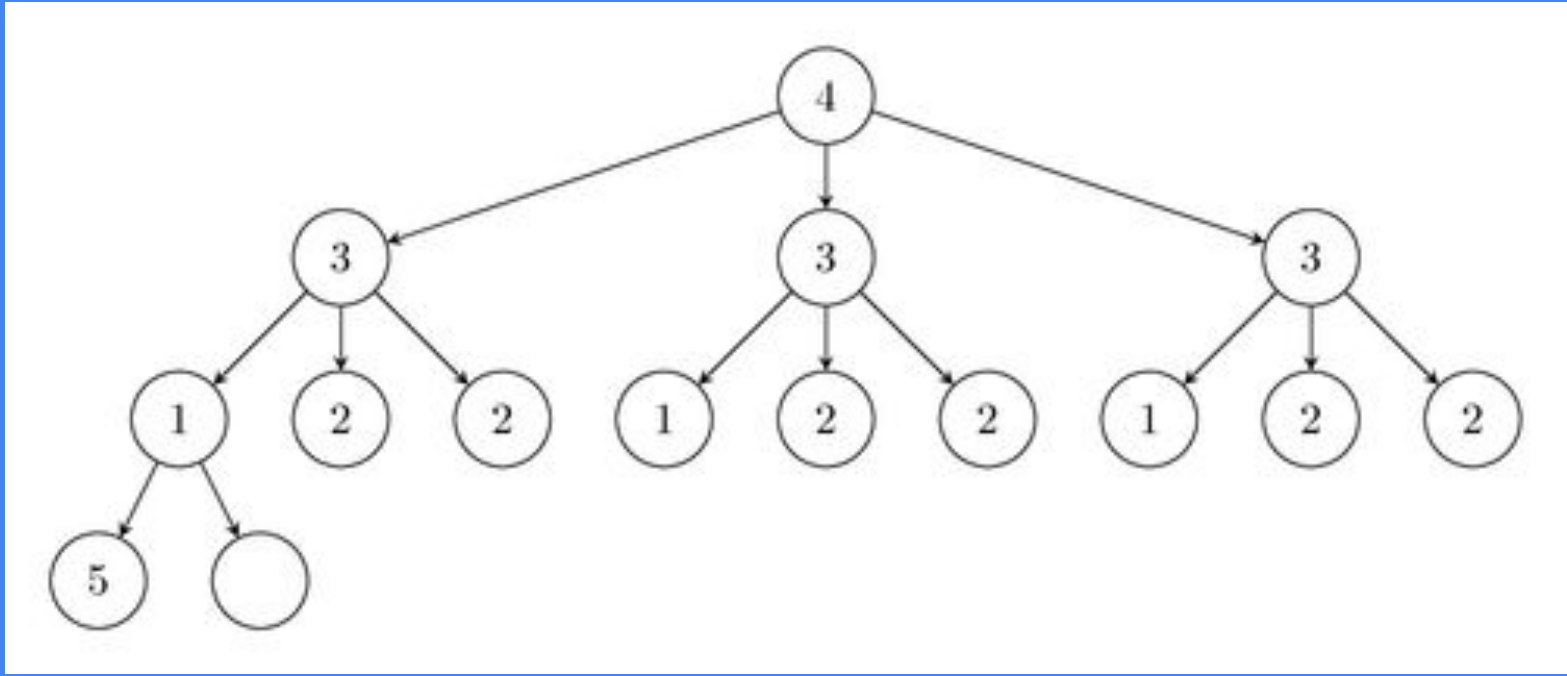
Ejemplo:

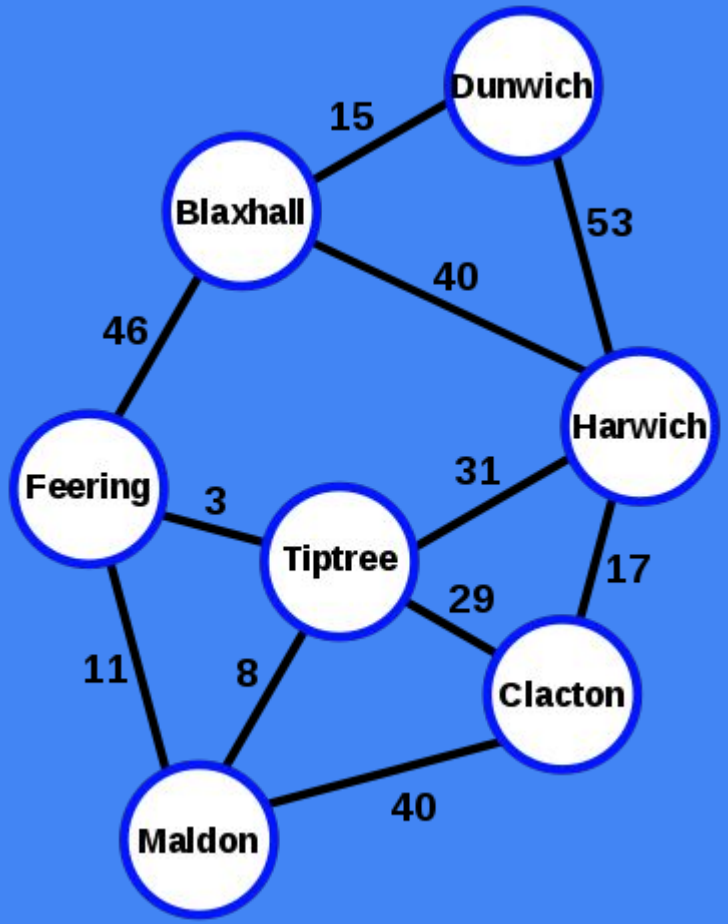
Árbol

Grafo conexo sin ciclos

Un grafo es un árbol si para todo par de vértices U V existe un único camino

Un grafo es un árbol si es conexo y tiene n-1 aristas


4
3
3
3
1
2
2
1
2
2
1
2
2
5



Imaginemos que esos nombres en los nodos son ciudades, y las aristas significan vuelos disponibles y el peso en las aristas es cuanto sale el vuelo.

Queremos viajar de Maldon a Clacton, me conviene comprar un pasaje directo de Maldon a Clacton?

No, es más barato viajar a TipTree y luego a Clacton


YA DAME EL MALDITO CODIGO
GENERADORMEMES.COM

Temario Parte 1

- Formas de representación
- DFS
- BFS
- Ejemplos
- Dijkstra

Formas de representación

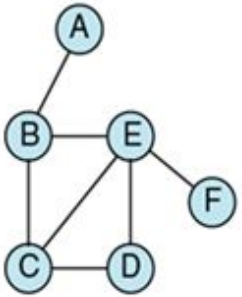
Matriz de adyacencia

Es una forma de representar un grafo de manera simple.

Vamos a tener una matriz G donde cada posición i, j va a tener un 1 si existe arista entre los nodos i, j. Y 0 de lo contrario.

G[i][j] = 1 if i---j

G[i][j] = 0 otherwise

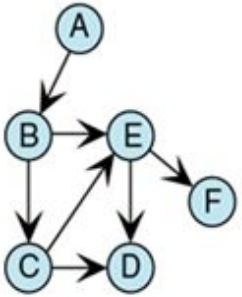


Vertex vector
A
B
C
D
E
F

Adjacency matrix

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

(a) Adjacency matrix for non-directed graph



Vertex vector
A
B
C
D
E
F

Adjacency matrix

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

(a) Adjacency matrix for directed graph

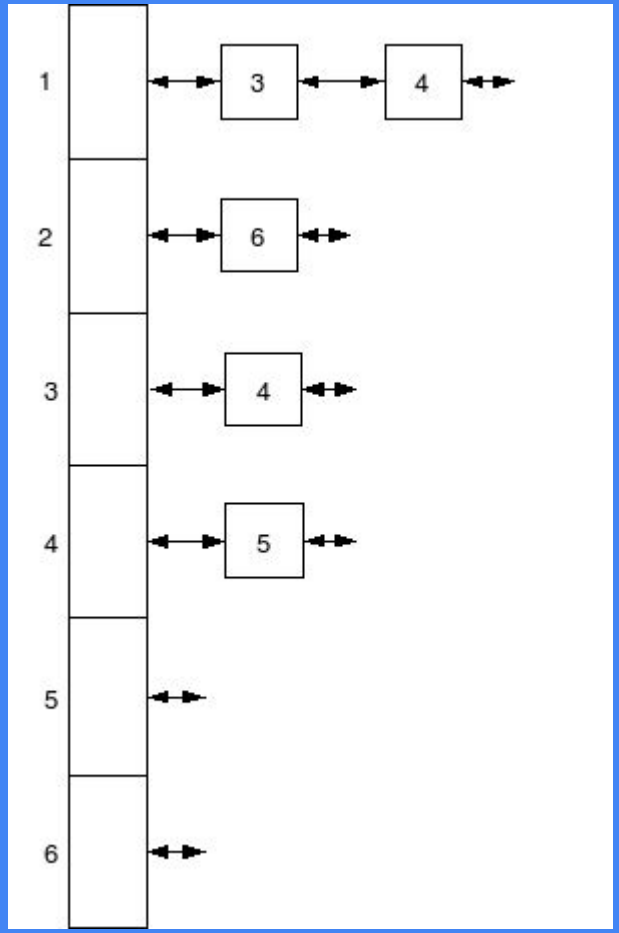
```
#include <bits/stdc++.h>
using namespace std;

int adj_matrix[100][100];

int main(){
        int n,m;
        cin>>n>>m;
        while(m--){
                int a,b;
                cin>>a>>b;
                adj_matrix[a][b]=1;
        }
        return 0;
}
```

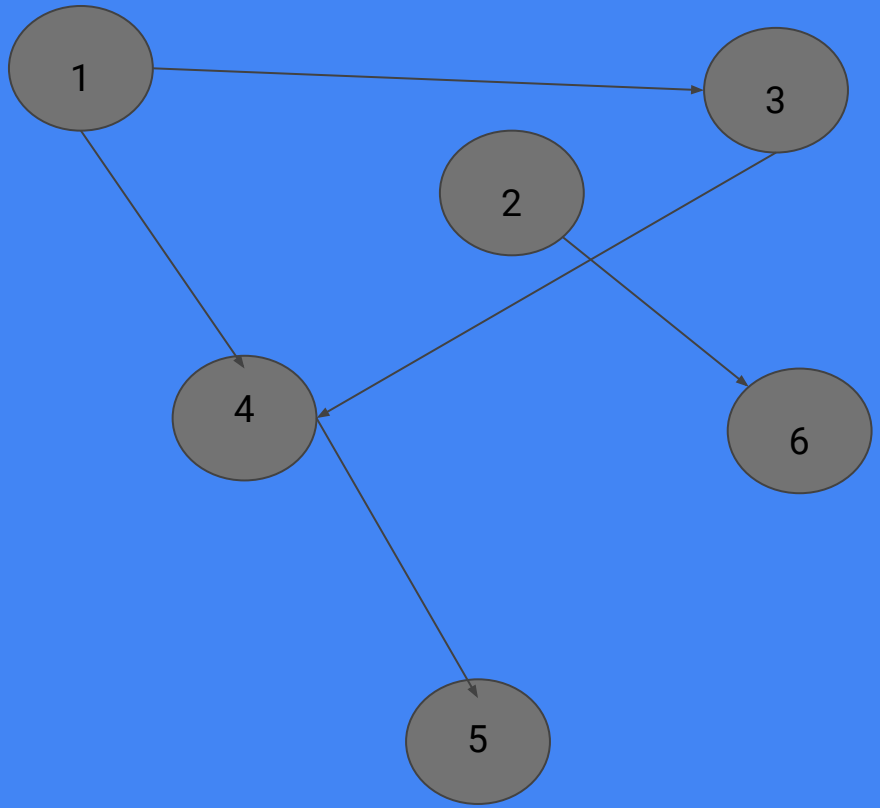
```
#include <bits/stdc++.h>
using namespace std;

int adj_matrix[100][100];

int main(){
        int n,m;
        cin>>n>>m;
        while(m--){
                int a,b;
                cin>>a>>b;
                adj_matrix[a][b]=1;
                adj_matrix[b][a]=1;
        }
        return 0;
}
```

Lista de Adyacencia

Es la forma más usada, y es la que probablemente más veces vayan a usar.

La idea es por cada nodo tener una lista de vecinos, es decir que en la lista del nodo V vamos a tener a todos los nodos P tal que exista una arista de V a P


1
3
4
2
6
3
4
4
5
5
6


1
3
2
4
6
5

```
#include <bits/stdc++.h>
using namespace std;

vector <int> g[100];

int main(){
	int n,m;
	cin>>n>>m;
	while(m--){
		int a,b;
		cin>>a>>b;
		g[a].push_back(b);
	}
	return 0;
}
```

```
#include <bits/stdc++.h>
using namespace std;

vector <int> g[100];

int main(){
	int n,m;
	cin>>n>>m;
	while(m--){
		int a,b;
		cin>>a>>b;
		g[a].push_back(b);
		g[b].push_back(a);
	}
	return 0;
}
```

Lets code!

DFS

DFS

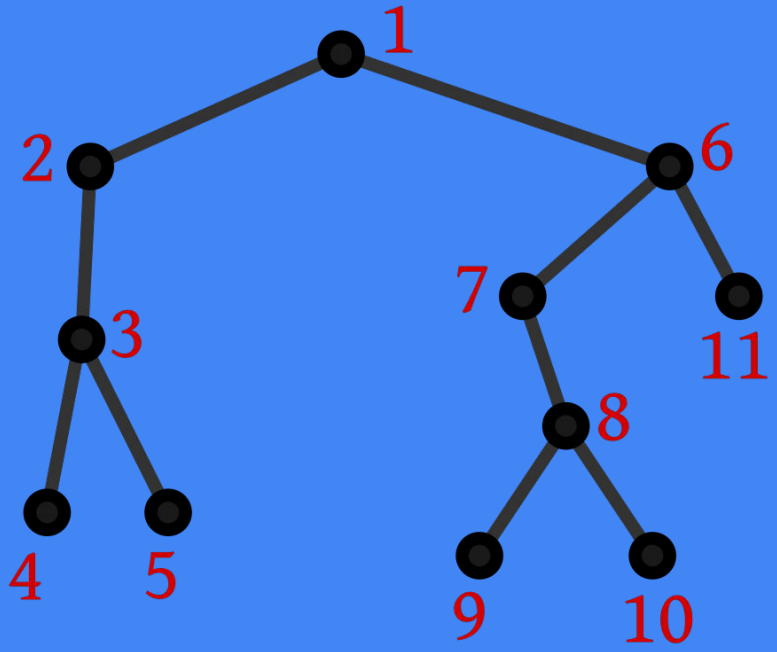
Es una de las formas de recorrer un grafo. DFS significa Depth First Search.

Es decir es un recorrido en profundidad. La idea es agarrar un Nodo e ir hacia abajo lo más que pueda, cuando no puedo más, voy a subir al nodo anterior y hacer exactamente lo mismo, así hasta terminar de recorrer todos los nodos.

(explicación “así noma”)(ojo, voy a ir bajando si y sólo si no pase por ahí antes)

Time Complexity: O(n+m)

Búsqueda en profundidad



```
void dfs(int nodo){
        cout<<”Estoy viendo el nodo:”<<node;
        visited[nodo] = 1;
        for(auto vecino : g[nodo]){
                if(!visited[vecino]){
                        dfs(vecino);
                }
        }
}
```

```
void dfs(int node){
        stack<int>s;
        s.push(node) ;
        encolado[node]=true;
        while(s.size()){
                int current=s.top();
                s.pop();
                cout<<”Estoy viendo el nodo:”<<current<<endl;
                for(int i=0; i<graph[current].size();i++){
                        int = graph[current][i];
                        if(!encolado[t]){
                                s.push(t);
                                encolado[t]=true;
                        }
                }
        }
}
```

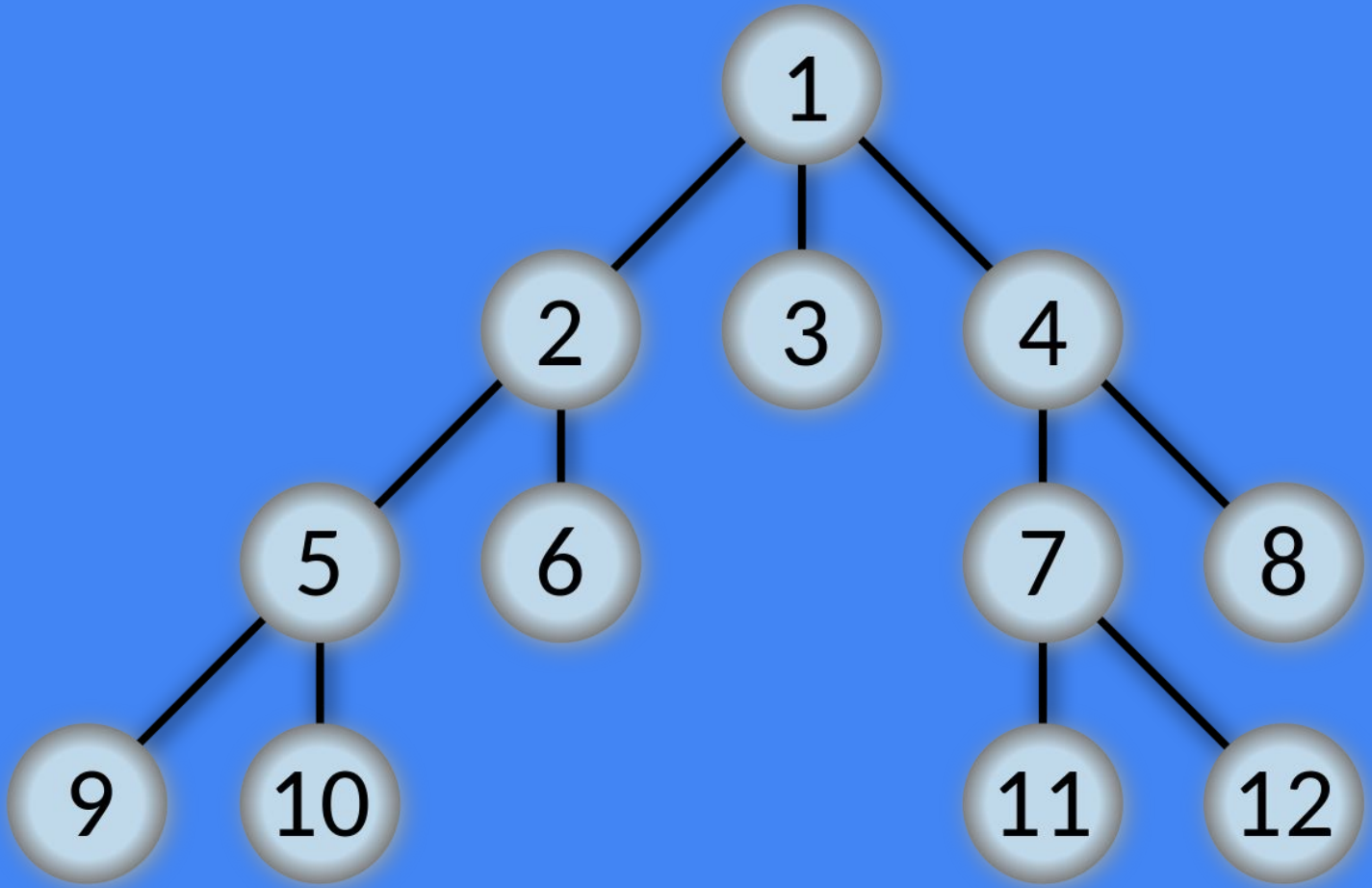
BFS

BFS

Es una de las formas de recorrer un grafo. BFS significa Breadth First Search.

Es decir es un recorrido por nivel. Intuitivamente, se comienza por la raiz y se visitan todos los vecinos, así sucesivamente para cada uno de los vecinos vamos a visitar a sus vecinos, etc..

Time Complexity: O(n+m). Veamos un ejemplo:


1
2
3
4
5
6
7
8
9
10
11
12

```
void bfs(int src){

        q.push(src);

        while(!q.empty()){
            int aux = q.front();
            q.pop();
            cout<<"Estoy viendo a "<<aux<<endl;
            if(!visited[aux]){
                visited[aux] = 1;
                for(int x : a[aux]){
                    if(!visited[x]){
                    q.push(x);
                }
            }
         }
}
```

Ejemplos

http://codeforces.com/contest/500/problem/A

http://codeforces.com/contest/580/problem/C

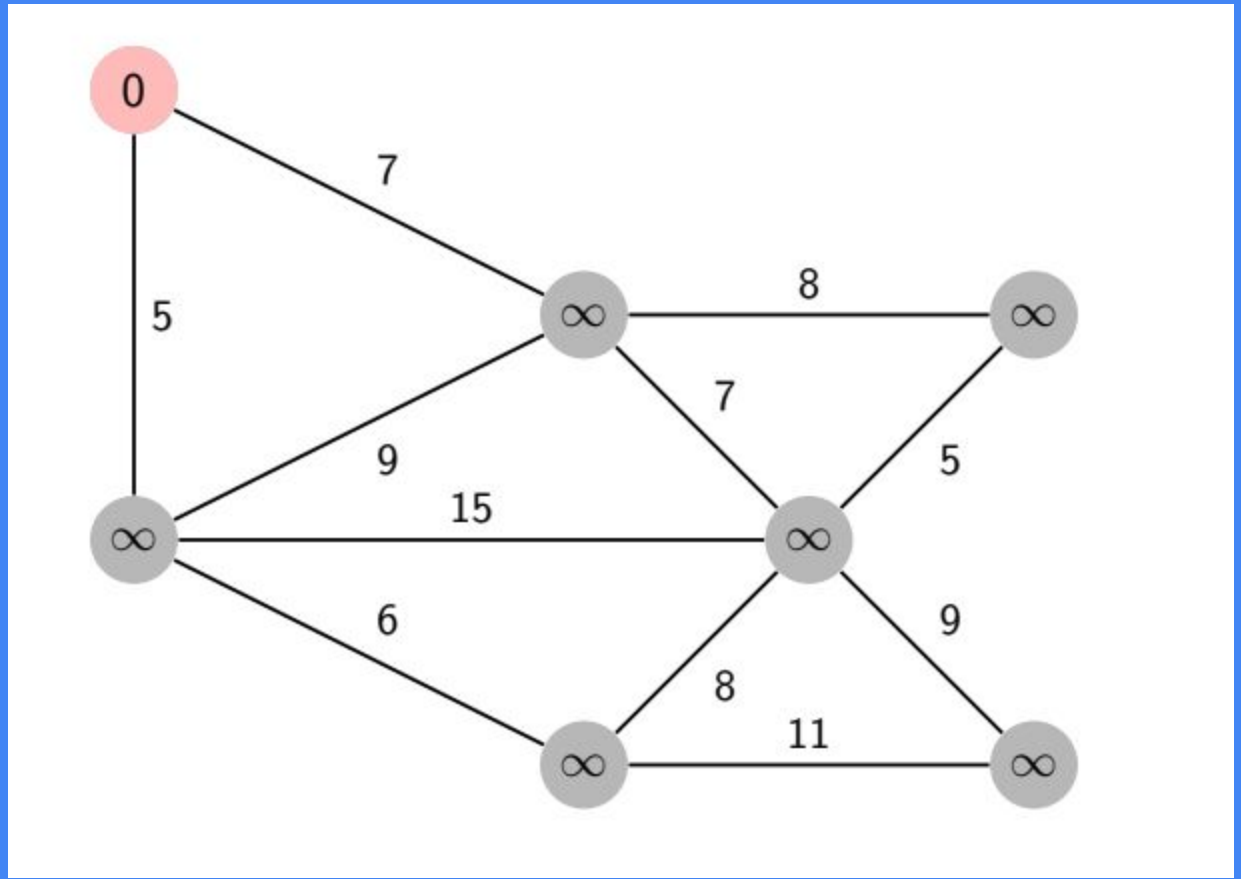
Calcular la cantidad de componentes conexas de un grafo No Dirigido.

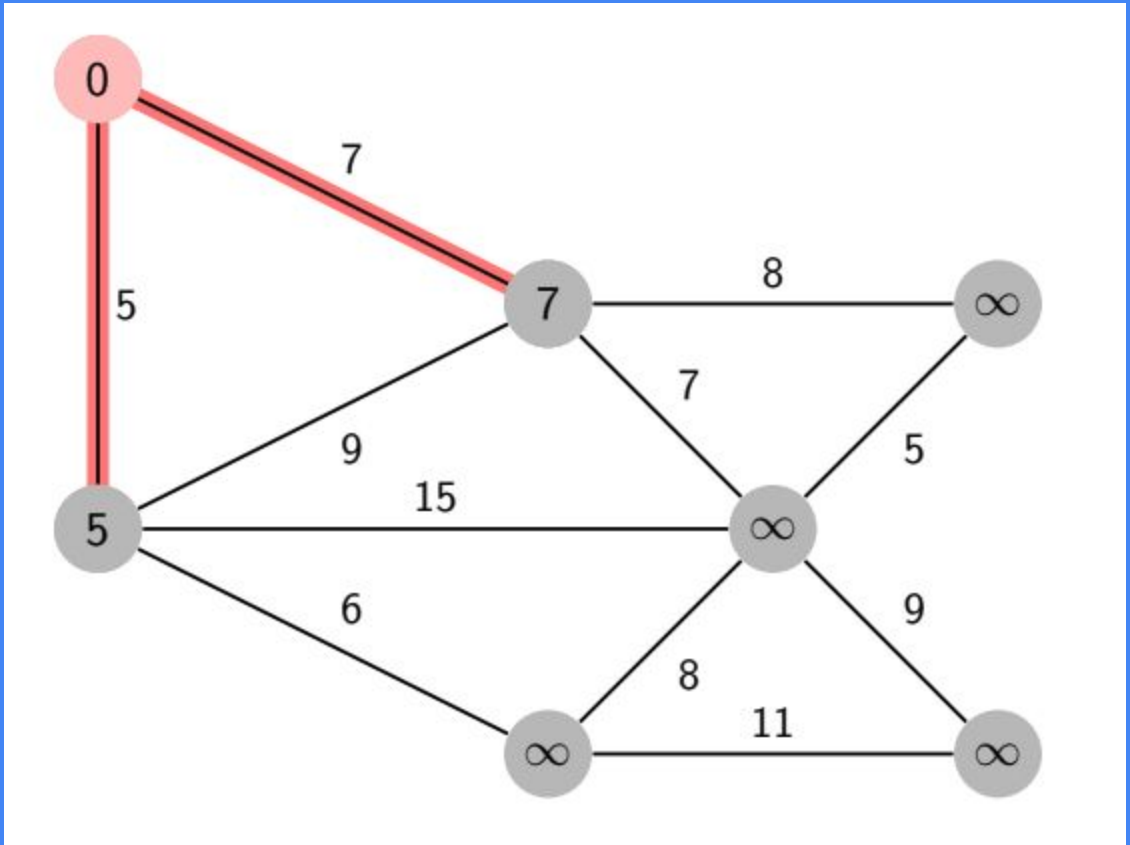
Dijkstra

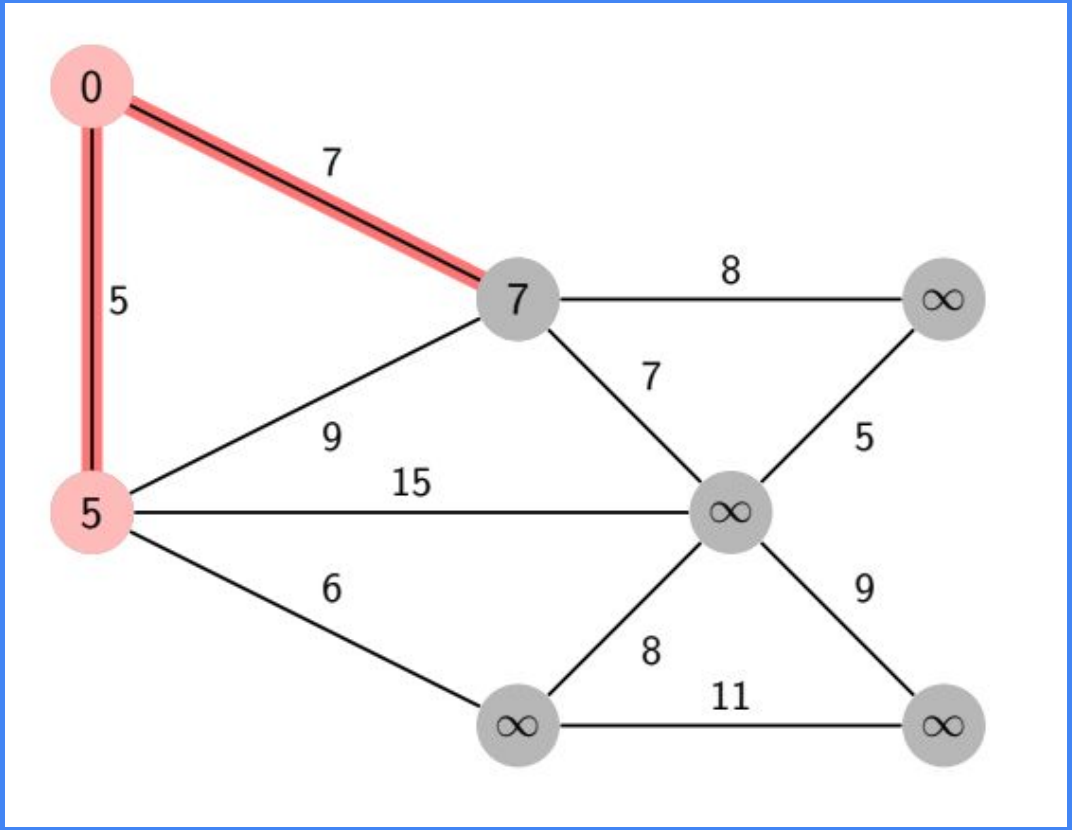
Dijkstra

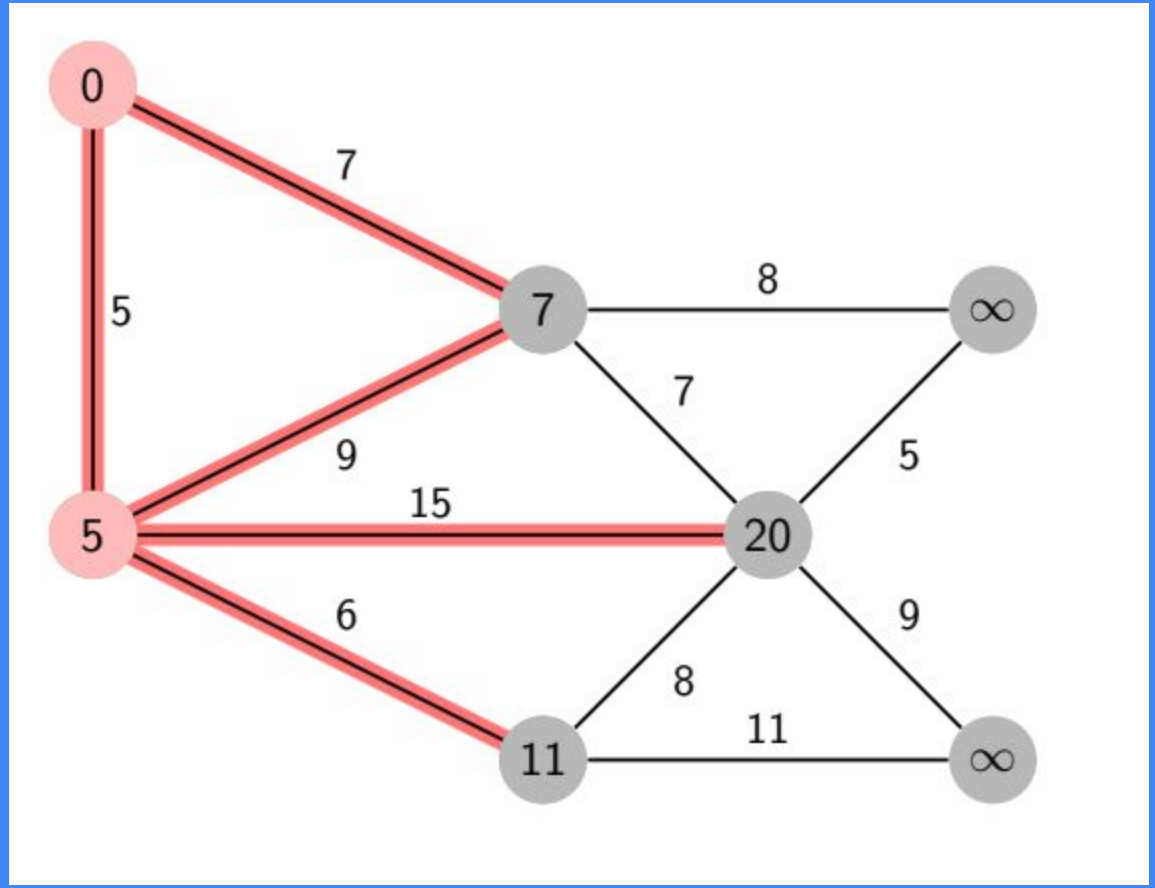
Si tenemos un grafo ponderado, es decir, con pesos en las aristas, y por alguna razón en alguna parte de nuestra solución queremos saber la mínima distancia de un nodo a todos los demás, usaremos Dijkstra. Nótese que es desde un nodo definido, no es la distancia de todos los nodos con todos.

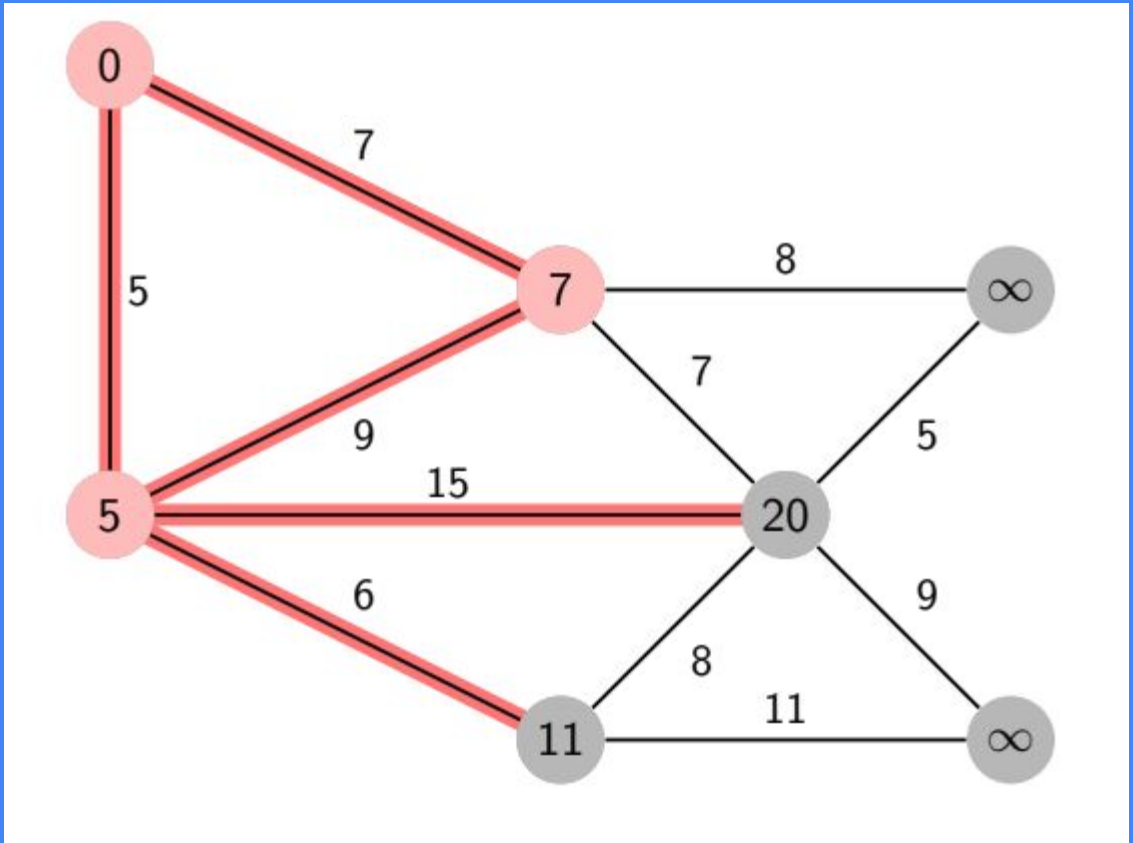
Time Complexity: O(m*log(n)). Veamos un ejemplo:

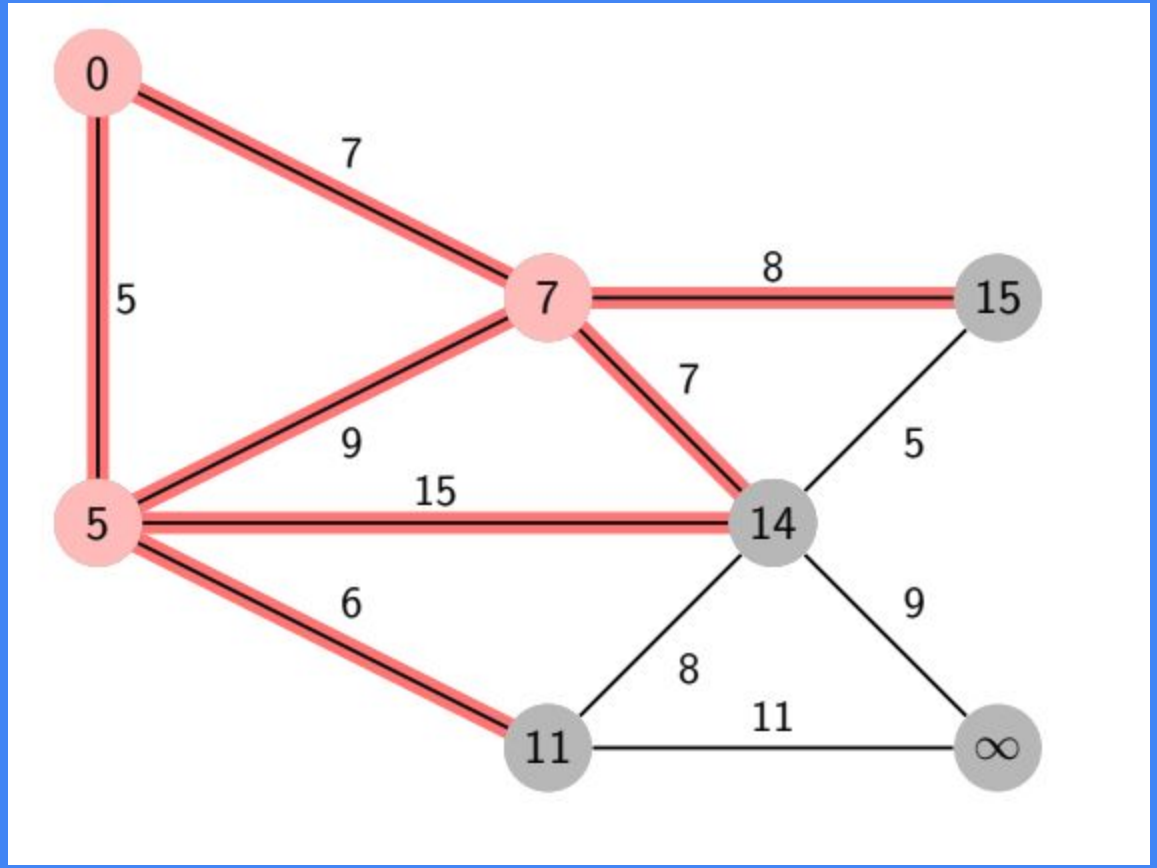

0
7
5
∞
8
∞
7
9
5
15
∞
∞
6
9
8
11
∞
∞

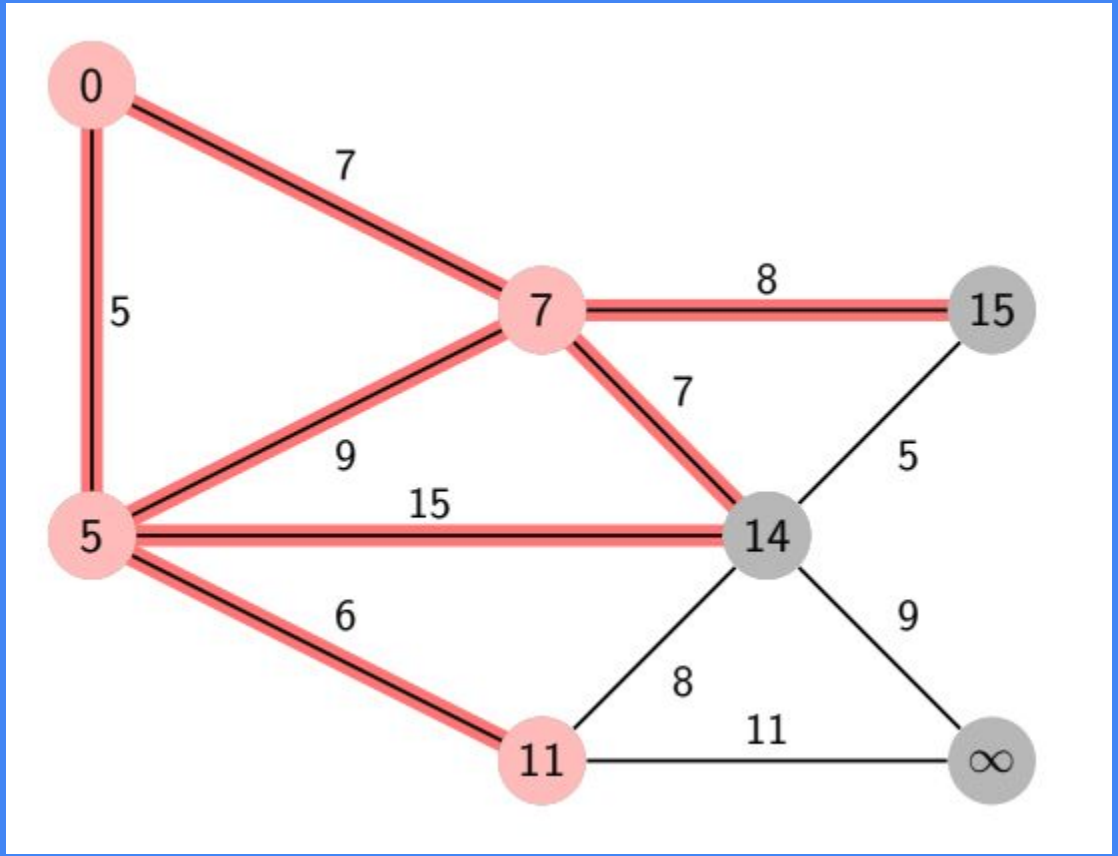

0
7
5
7
8
∞
7
9
5
15
5
∞
6
9
8
11
∞
∞

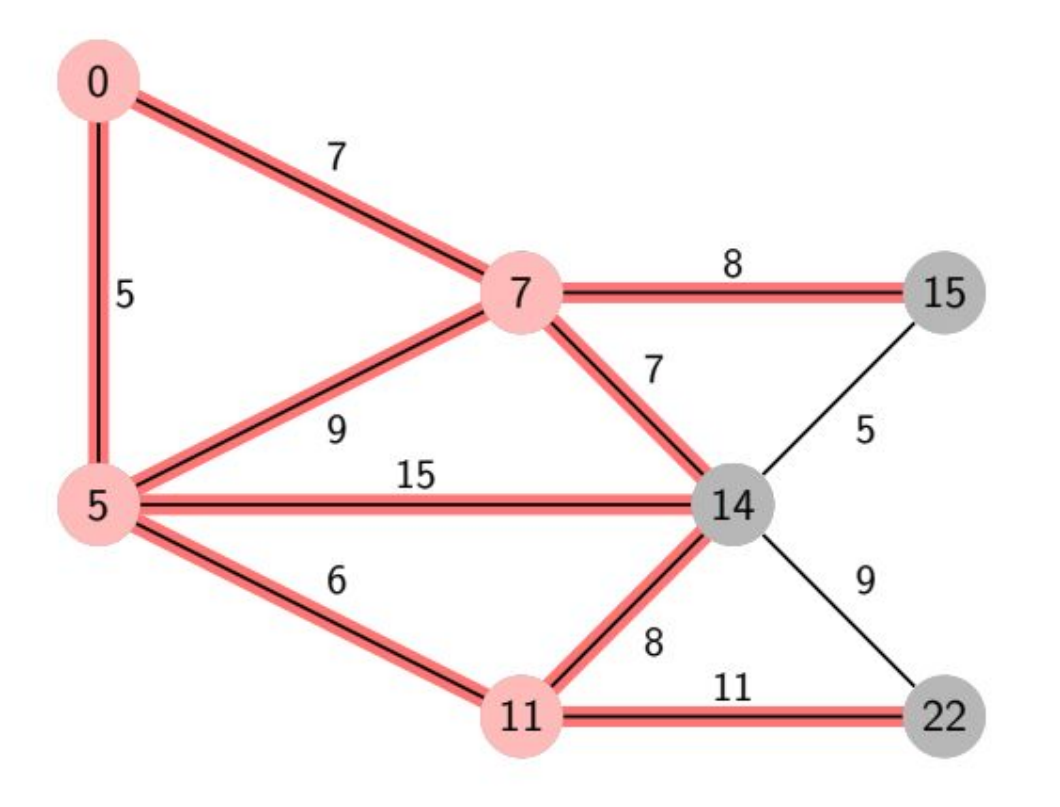

0
7
5
7
8
∞
7
9
5
15
5
∞
6
9
8
11
∞
∞

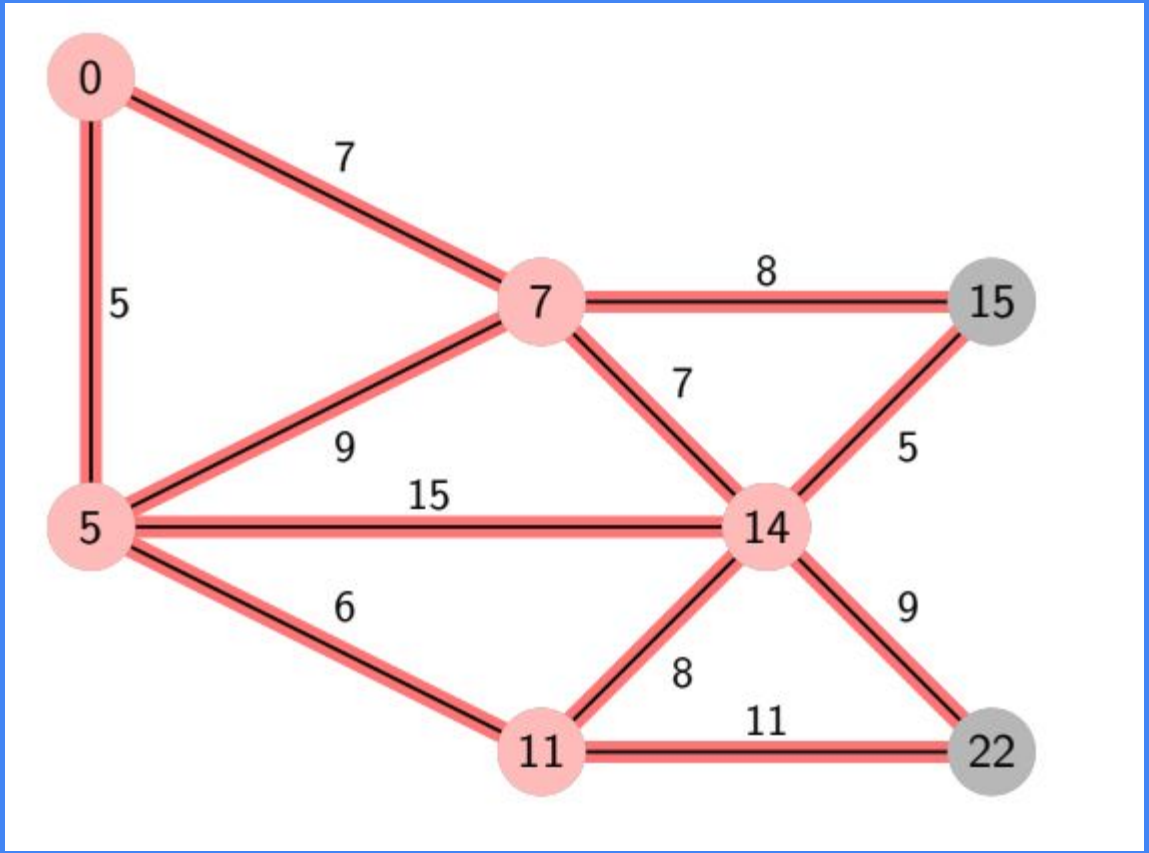

0
7
5
7
8
∞
7
9
5
15
5
20
6
9
8
11
11
∞

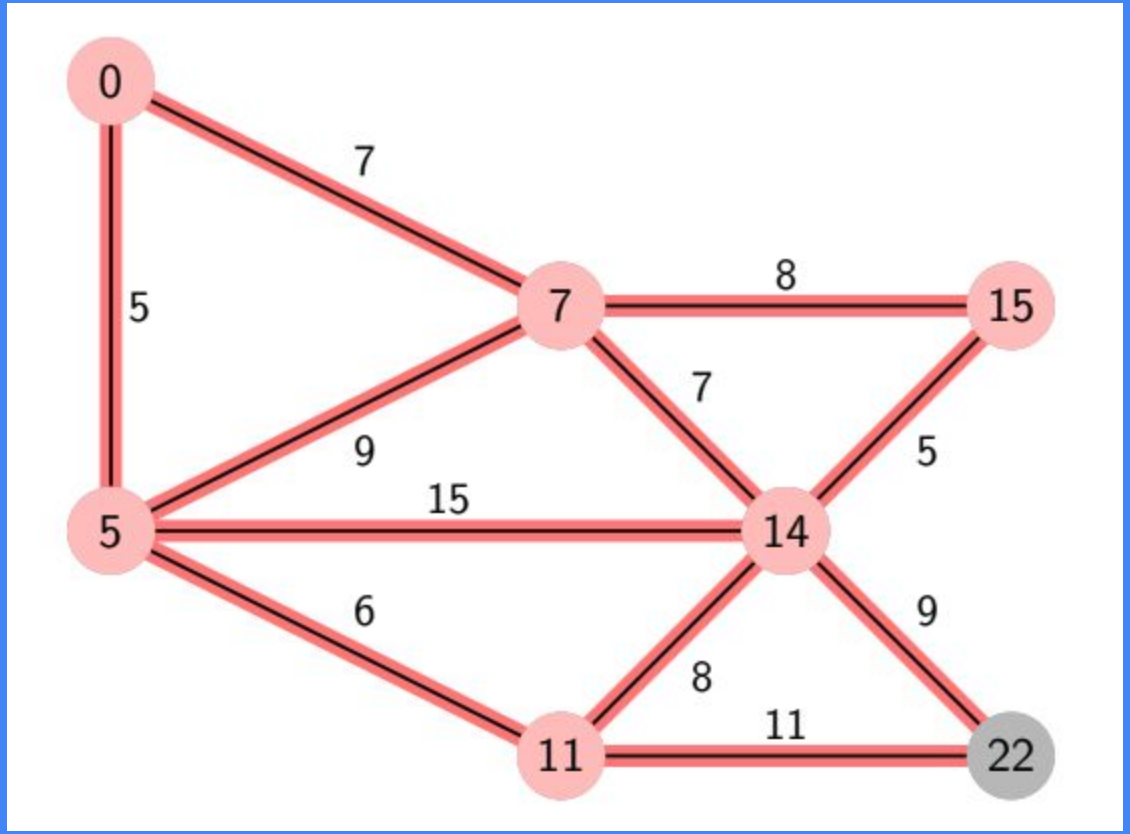

0
7
5
7
8
∞
7
9
5
15
5
20
6
9
8
11
11
∞

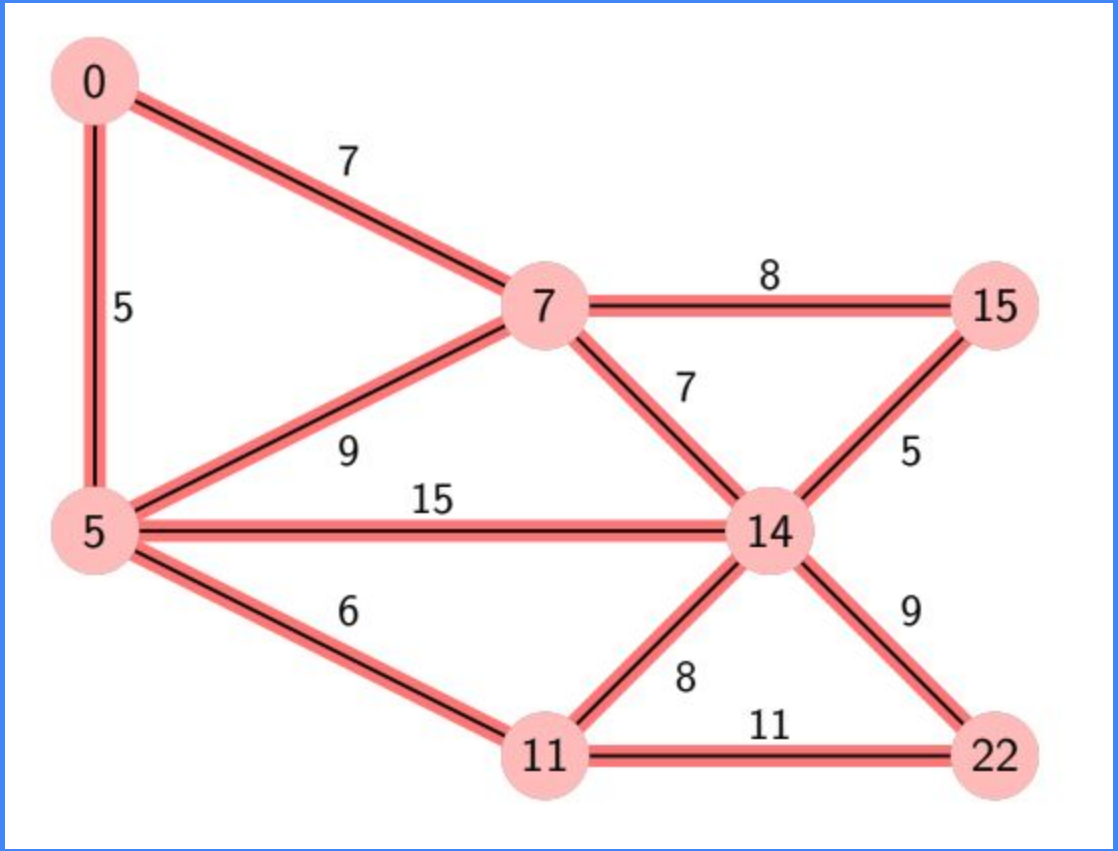

0
7
5
7
8
15
7
9
5
15
5
14
6
9
8
11
11
∞


0
7
5
7
8
15
9
7
5
15
5
14
6
9
8
11
11
∞


0
7
5
7
8
15
9
7
5
15
5
14
6
9
8
11
11
22


0
7
5
7
8
15
7
9
5
15
5
14
6
9
8
11
11
22


0
7
5
7
8
15
7
9
5
15
5
14
6
9
8
11
11
22


0
7
5
7
8
15
7
9
5
15
5
14
6
9
8
11
11
22

```
void dijkstra(int src){
	int u, v, i;
	long long w;
	pq.push({0, src});
	dist[src] = 0;
	while(!pq.empty()){
		w = pq.top().first;
		u = pq.top().second;
		pq.pop();
		if(w <= dist[u]){
			for(auto x : g[u]){
				v = x.second;
				w = x.first;
				if(dist[v] > dist[u] + w){
					dist[v] = dist[u] + w;
					pq.push({dist[v], v});
				}
			}
		}
	}
}
```

Ejercicio

https://www.spoj.com/problems/EZDIJKST/

Floyd - Warshall

Floyd - Warshall

Ahora bien, Dijkstra nos daba la mindist desde un nodo hasta todos los demás.

Pero qué pasa si quiero la distancia de todos los nodos hacia todos los otros?

Usamos en este caso Floyd-Warshall.

Floyd - Warshall

El algoritmo es una dp. Se basa en pensar, cuál es el costo mínimo de ir de i a j pasando solamente por nodos del conjunto {1,..,k}.

Formalmente :

minCost(i,j,0) = pesos[i][j];

minCost(i,j,k) = min(minCost(i,j,k-1), minCost(i,k,k-1) + minCost(k,j,k-1));

```
const long long INF = 1e14;
long long g[MAXN][MAXN];
int n;
void floyd(){
	for(int k=0; k<n; k++){
		for(int i=0; i < n; i++){
			if(g[i][k]<INF){
				for(int j=0; j<n;j++){
					if(g[k][j]<INF){
						g[i][j] = min(g[i][j],g[i][k]+g[k][j]);
					}
				}
			}
		}
	}
```

Para practicar

https://a2oj.com/category?ID=13

Los que tienen “difficulty level” = 1

https://codeforces.com/contest/910/problem/A ya lo hicimos con greedy y con dp, y si lo hacemos con grafos?

Kruskal

Kruskal

Para poder entender el algoritmo de Kruskal necesitamos saber otro llamado Union-Find, que además sirve para muchos otros algoritmos y es bastante poderoso para solucionar algunos problemas.

UF

E	F	I	D	C	A	J	L	G	K	B	H
0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

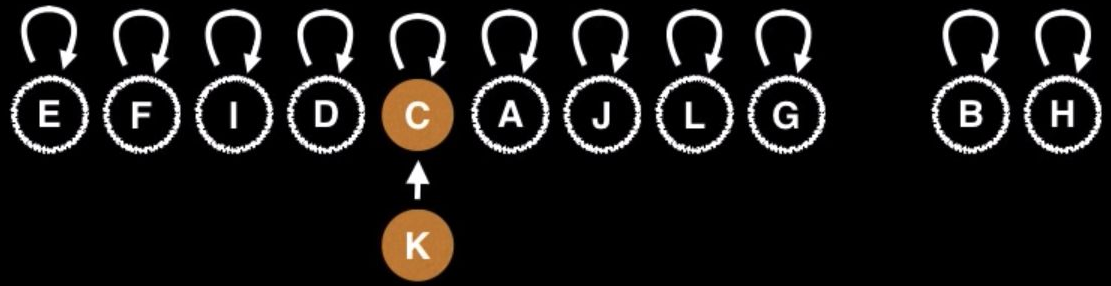
Union(C,K) ←
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)

(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	1	2	3	4	5	6	7	8	4	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

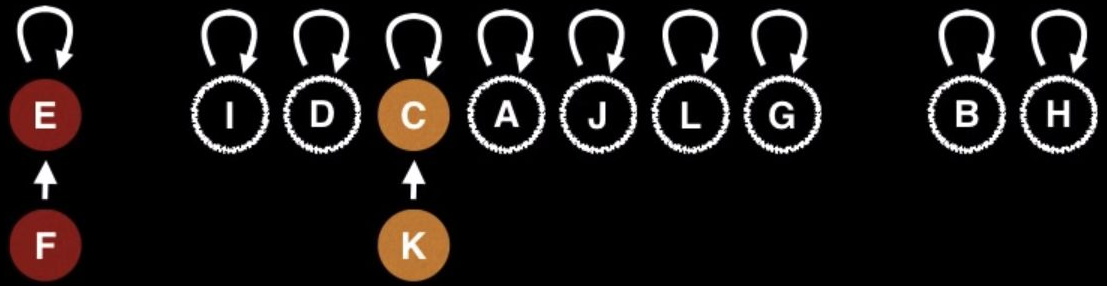


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	2	3	4	5	6	7	8	4	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

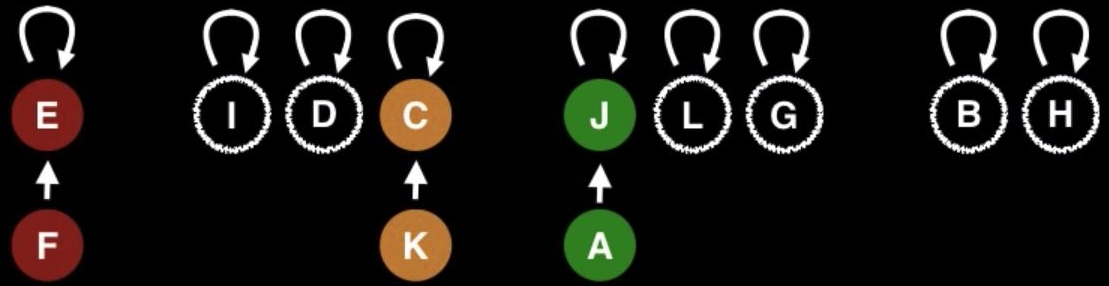


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	2	3	4	6	6	7	8	4	10	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

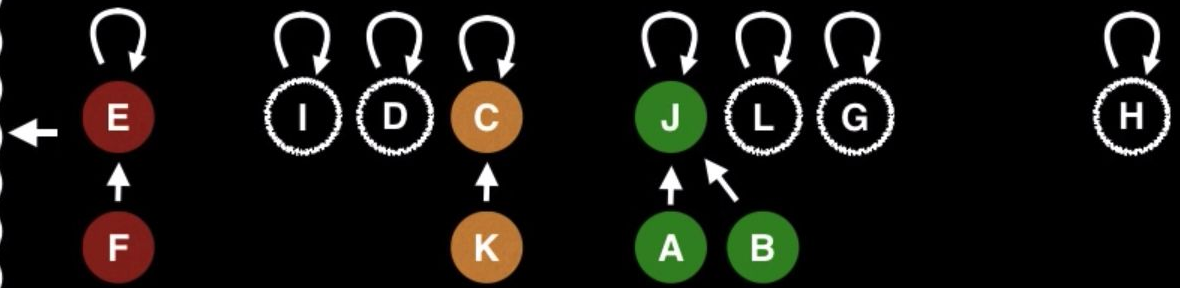


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	2	3	4	6	6	7	8	4	6	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)

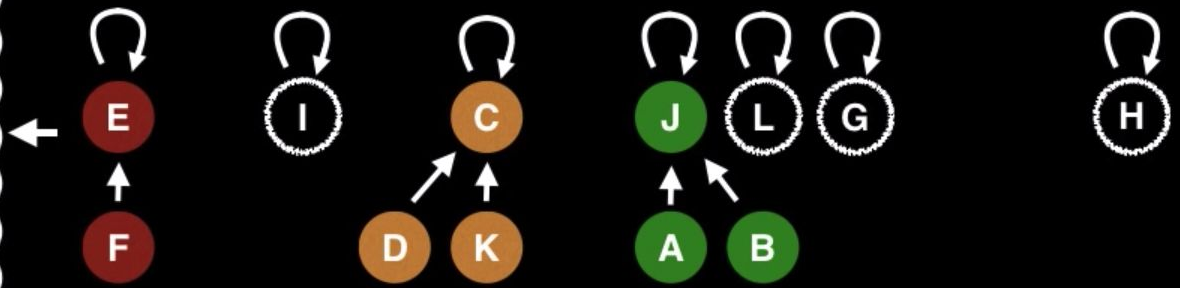


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	2	4	4	6	6	7	8	4	6	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

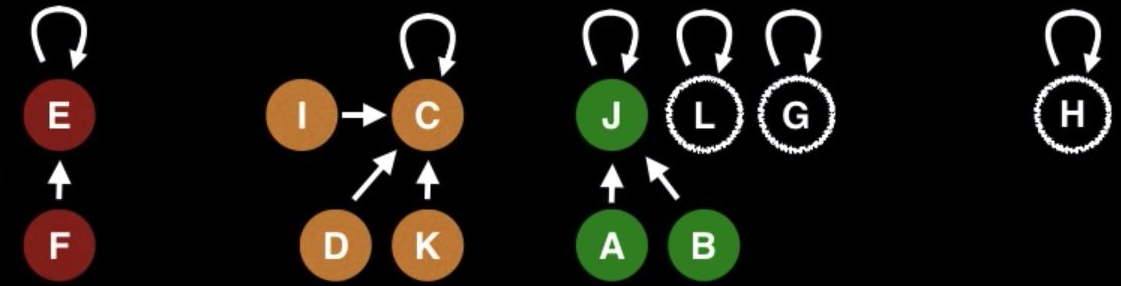


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	4	4	4	6	6	7	8	4	6	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)

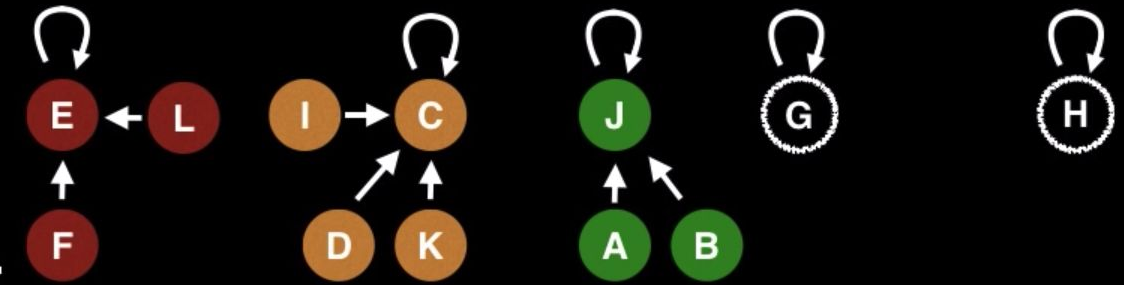


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	4	4	4	6	6	0	8	4	6	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

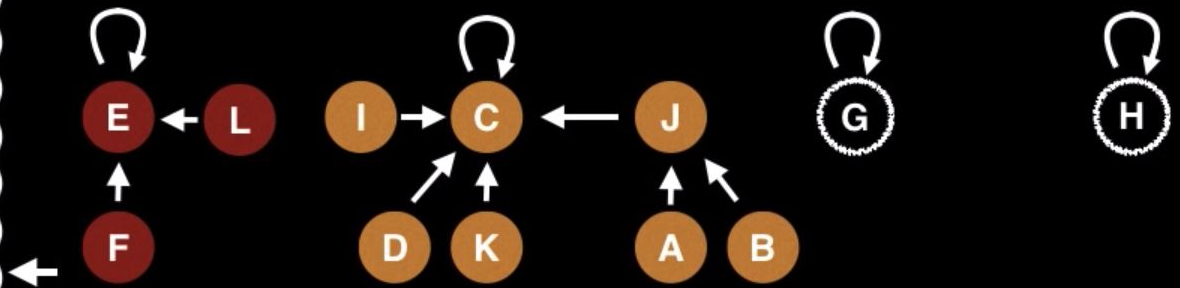


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	4	4	4	6	4	0	8	4	6	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```

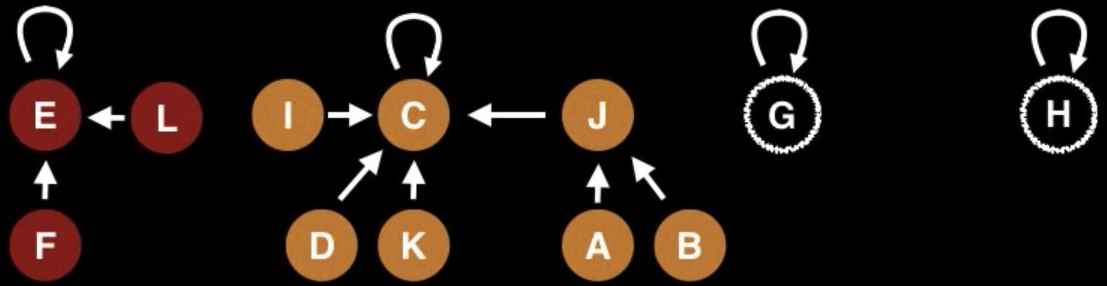


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	4	4	4	6	4	0	8	4	6	11
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B) ←
Union(H,G)
Union(H,F)
Union(H,B)
```

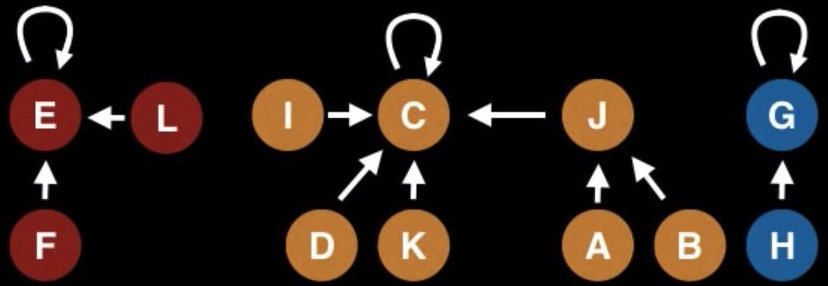


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	4	4	4	6	4	0	8	4	6	8
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G) ←
Union(H,F)
Union(H,B)
```

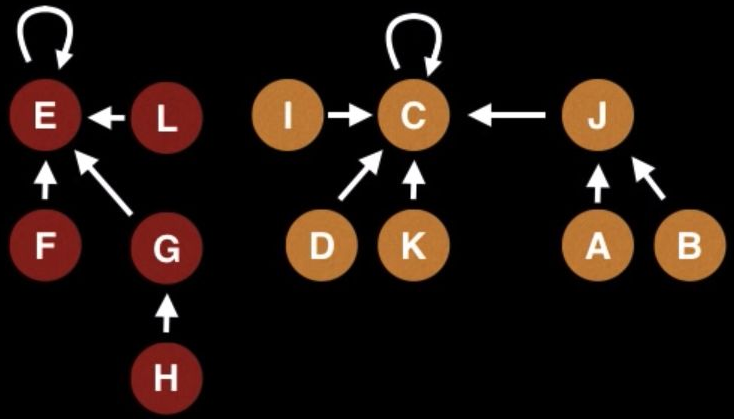


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
0	0	4	4	4	6	4	0	0	4	6	8
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F) ←
Union(H,B)
```

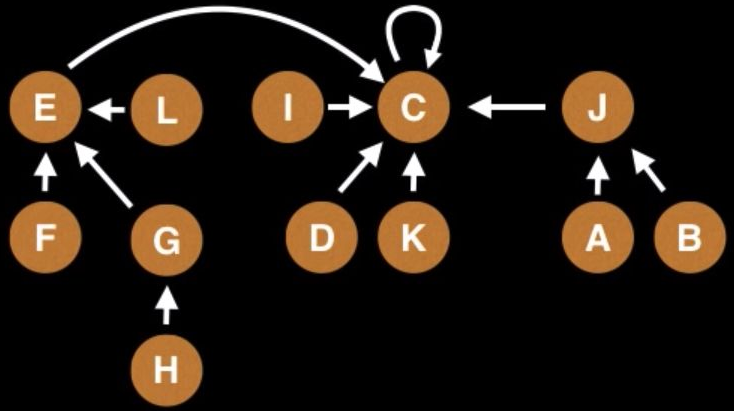


(This example does not use path compression)

E	F	I	D	C	A	J	L	G	K	B	H
4	0	4	4	4	6	4	0	0	4	6	8
0	1	2	3	4	5	6	7	8	9	10	11

Instructions:

```
Union(C,K)
Union(F,E)
Union(A,J)
Union(A,B)
Union(C,D)
Union(D,I)
Union(L,F)
Union(C,A)
Union(A,B)
Union(H,G)
Union(H,F)
Union(H,B)
```



(This example does not use path compression)

Problemas?

Cada vez que queremos unir algo es 0(n)

Resumen:

Find: trepar hasta el root node

Union: hacemos find de los dos, si es el mismo ya esta, y sino hacemos que una root es padre de la otra

Solucion?

Path Compression.

Using **path compression**

Instructions:

```
Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)
```

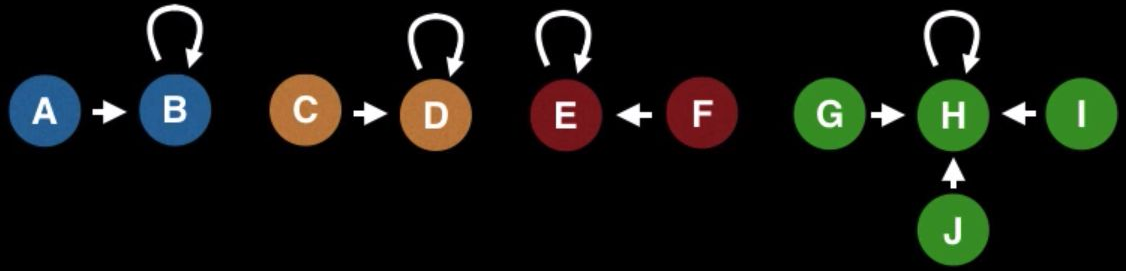
```
Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)
```

Using **path compression**

Instructions:

```
Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)
```

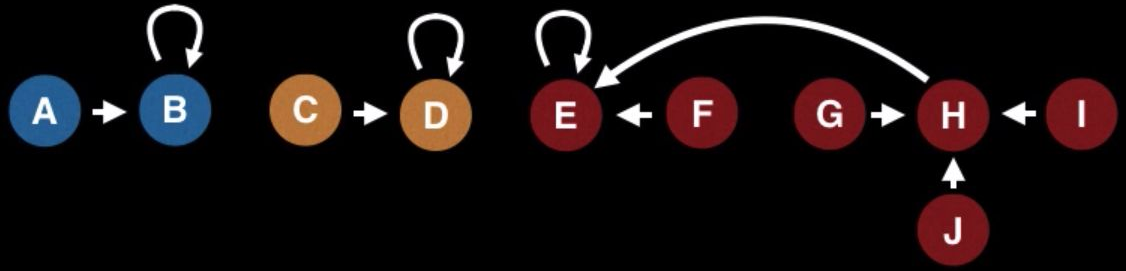
```
Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)
```



Using **path compression**

Instructions:

Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)
Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)

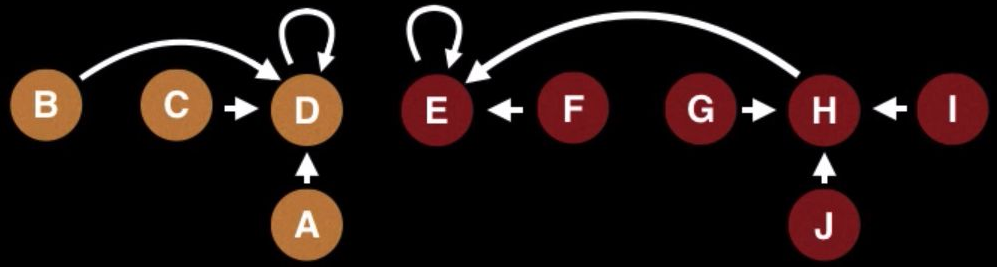


Using **path compression**

Instructions:

```
Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)
```

```
Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)
```

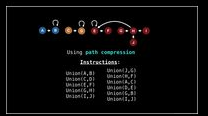


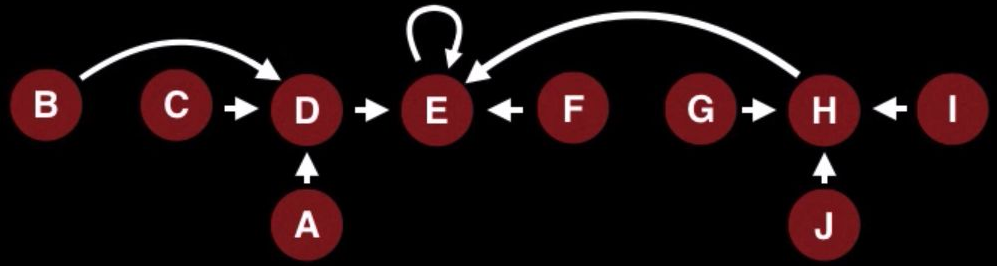
Using **path compression**

Instructions:

Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)

Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)

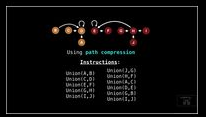


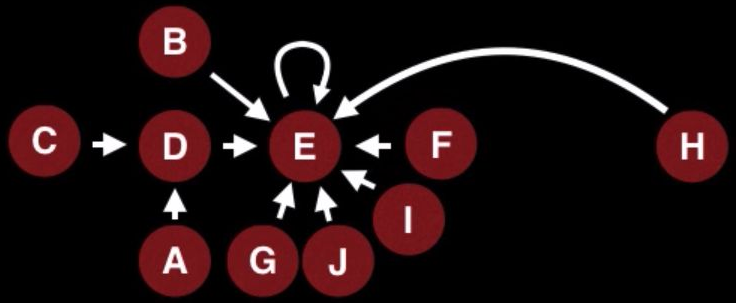


Using **path compression**

Instructions:

Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)
Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)





Instructions:

Union(A,B)
Union(C,D)
Union(E,F)
Union(G,H)
Union(I,J)

Union(J,G)
Union(H,F)
Union(A,C)
Union(D,E)
Union(G,B)
Union(I,J)

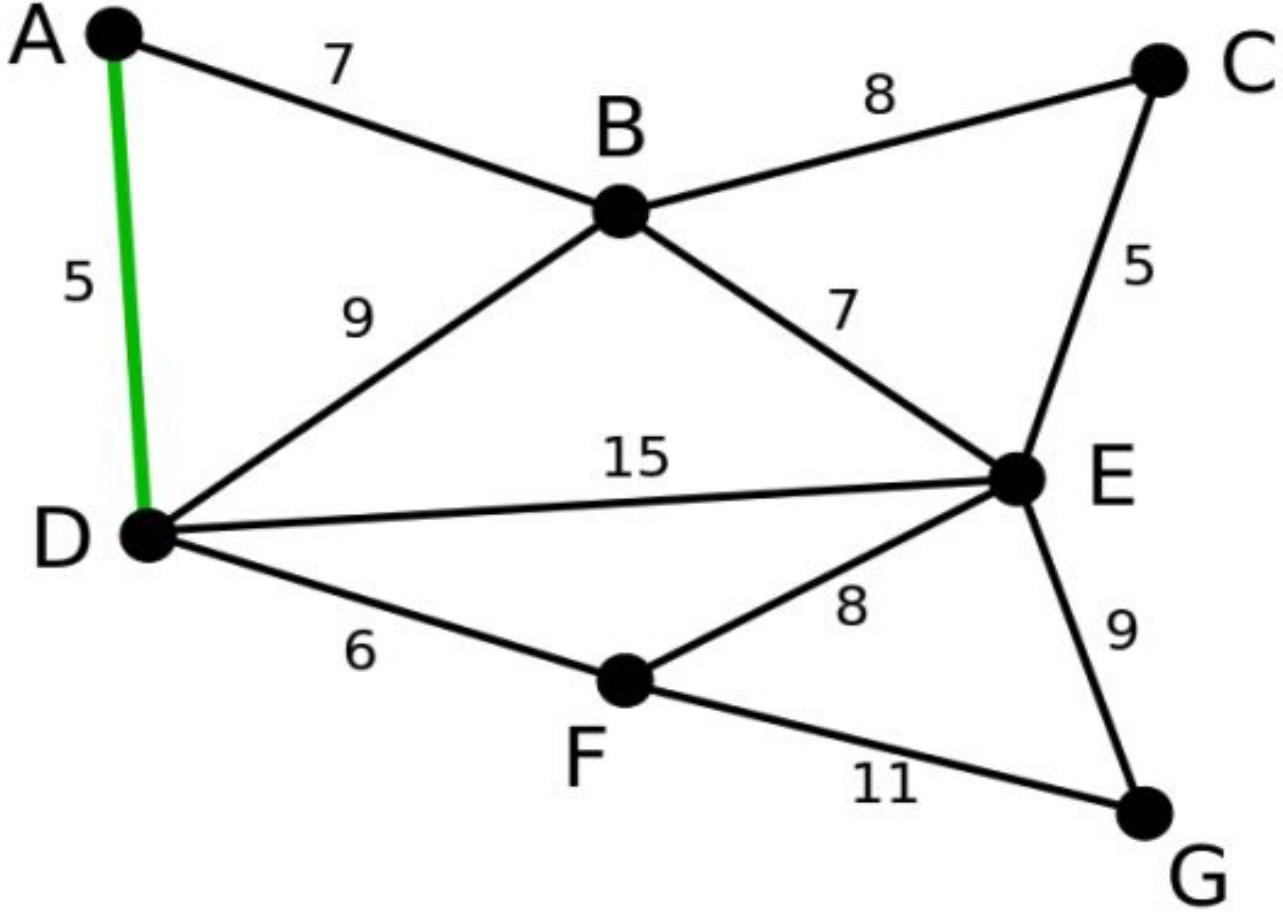
```cpp
#include <vector>

int reprs[1000];
int uf_rank[1000];

void initUF(int n){
  for(int i=0;i<n;i++){
    reprs[i] = i;
    uf_rank[i] = 0;
  }
}

int find(int x){
  vector<int> st;
  while(reprs[x] != x){
    st.push_back(x);
    x = reprs[x];
  }
  // actualizo los representantes (path compression)
  for(auto i : st) reprs[i] = x;
  return x;
}
```

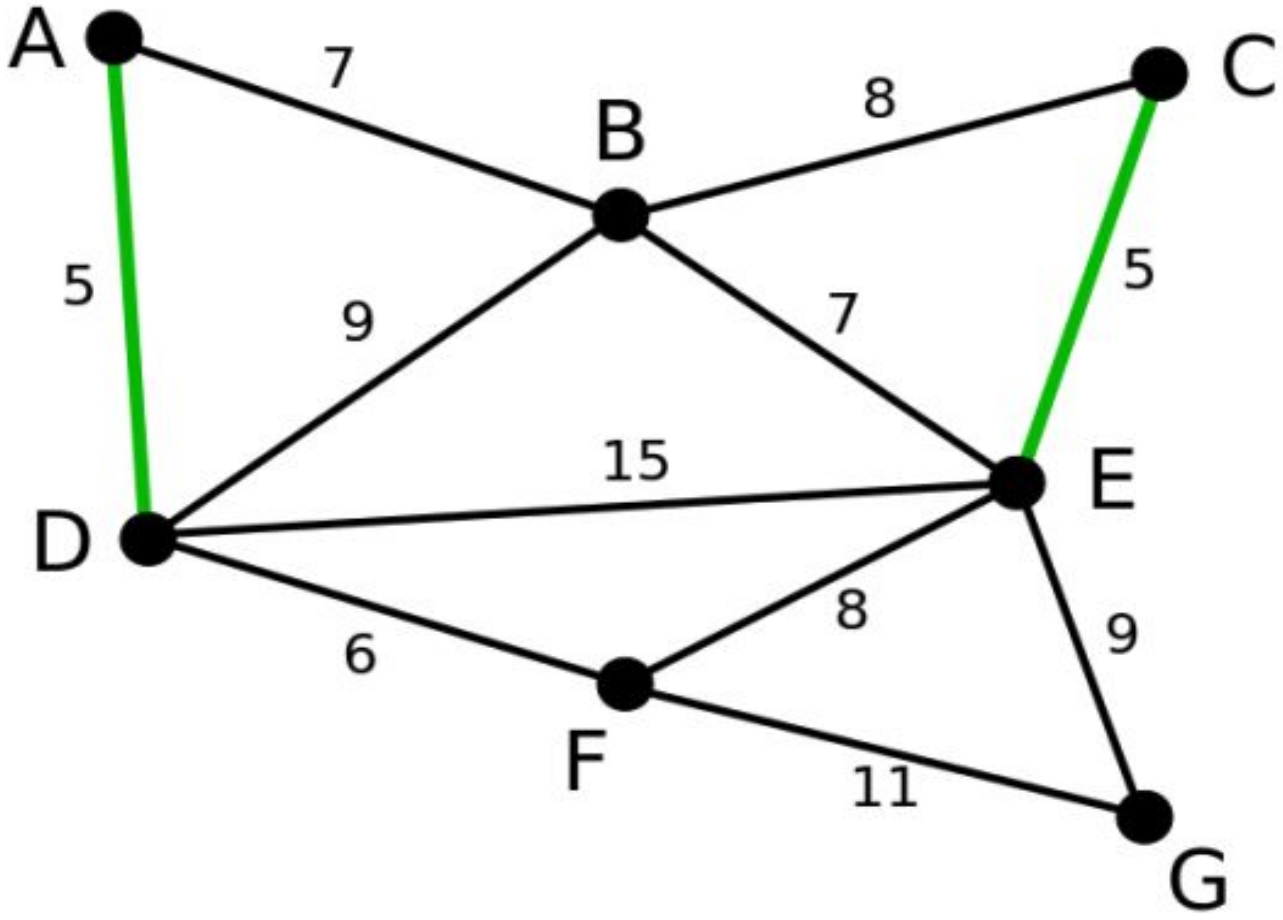
```
void join(int x, int y){
  // if(find(y) == find(x)) return;
  int repx = find(x);
  int repy = find(y);

  // Y se come a X
  if(uf_rank[repx] < uf_rank[repy]) reprs[repx] = repy;
  // X se come a Y
  else if(uf_rank[repx] > uf_rank[repy]) reprs[repy] = repx;
  else{
    // Tiene la misma cantidad, pero X se come a Y
    reprs[repy] = repx;
    // Lo unico que aca el rank crece
    uf_rank[repx] += 1;
  }
}
```

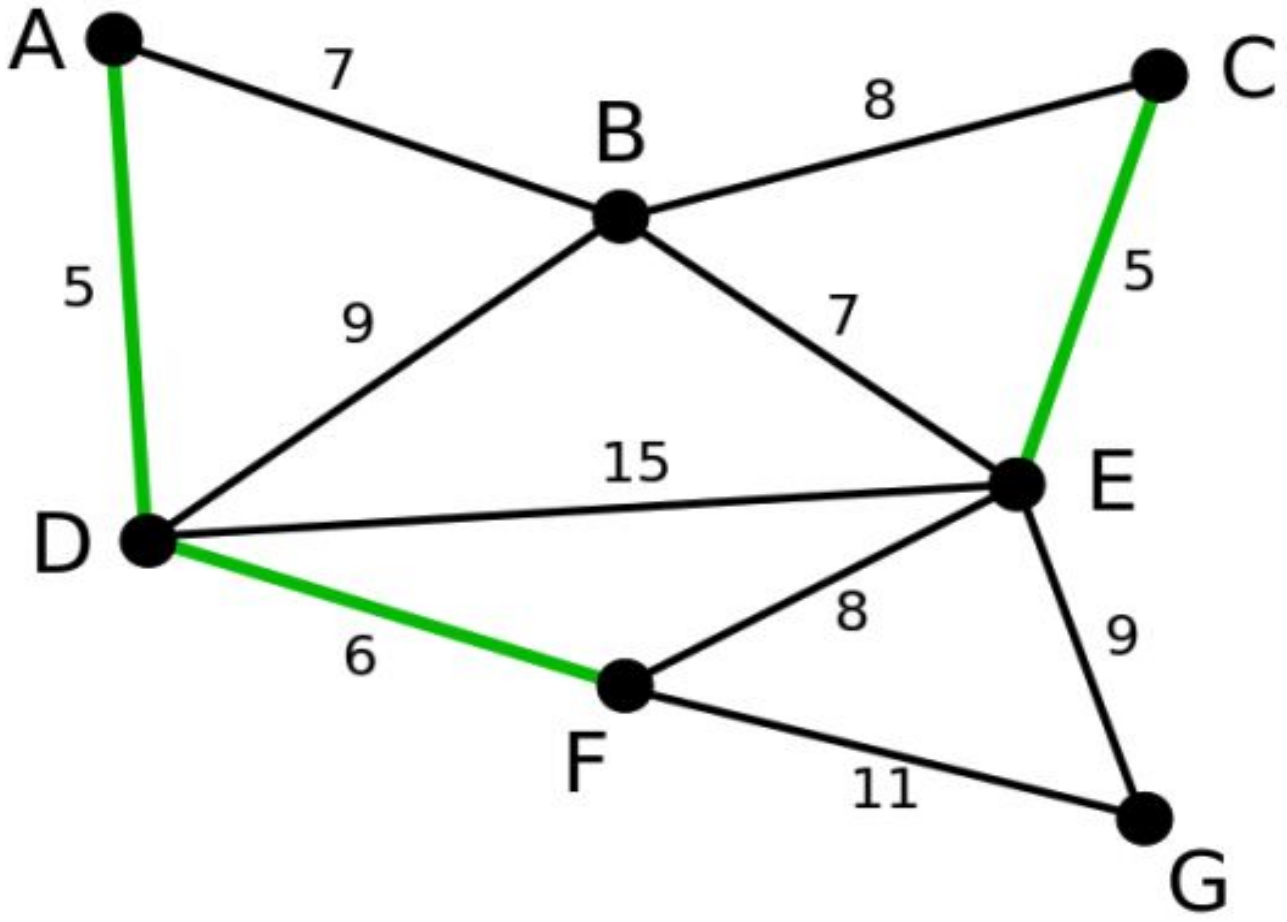
Ahora si, Kruskal

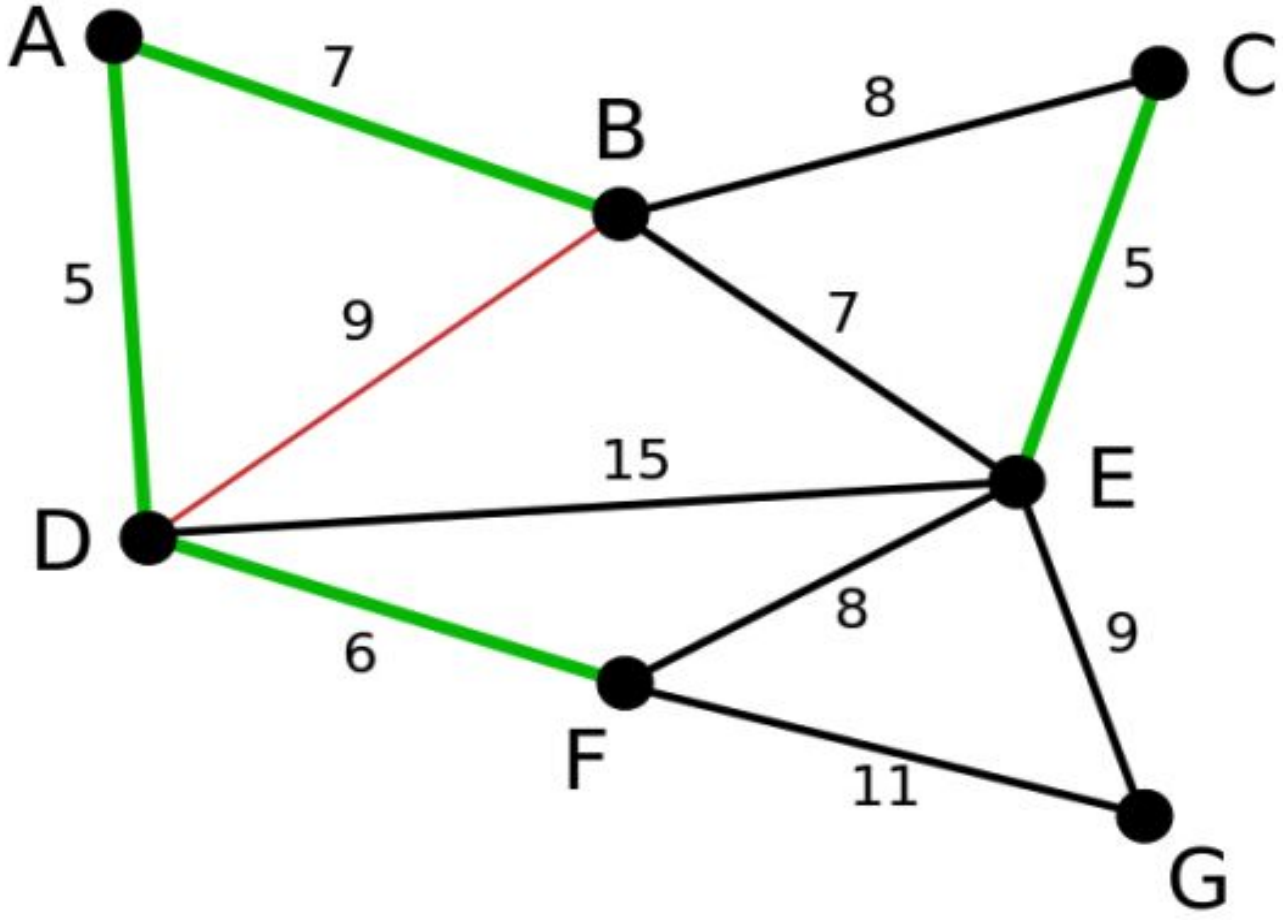
El algoritmo de Kruskal ordena las aristas por peso y va agregando desde la arista de menor peso hasta la de mayor peso, **evitando agregar las aristas que forman ciclos.**
Para saber si forman ciclos se usa la Union-Find. Y se usa de la siguiente manera:

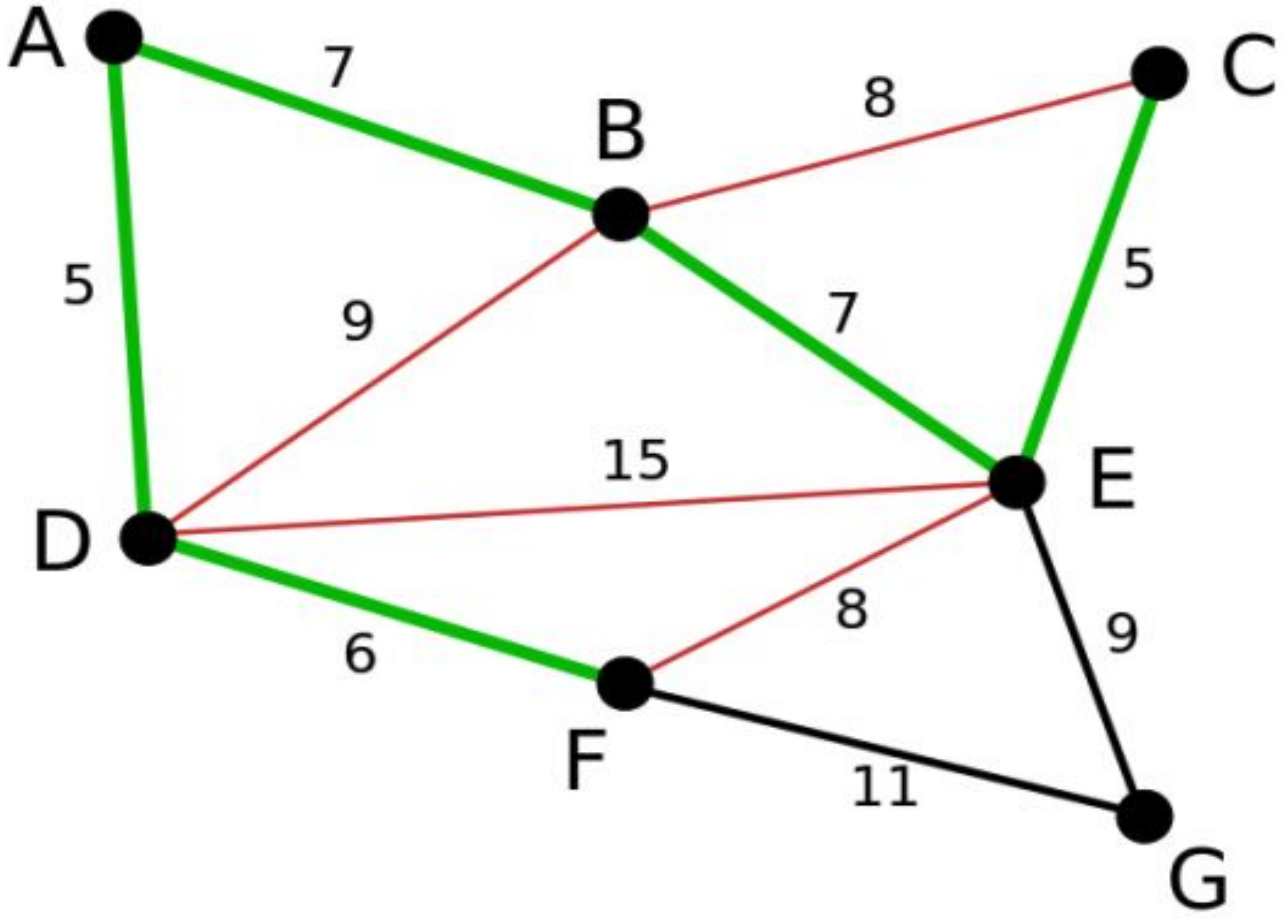
- En un principio todos los puntos son una componente conexa, entonces hay N conjuntos disjuntos.
- Cuando se unen dos componentes conexas, se hace un **union** de los conjuntos correspondientes.
- Para ver si dos aristas pertenecen a distintas componentes nos fijamos en el conjunto que pertenecen con la función **find**.

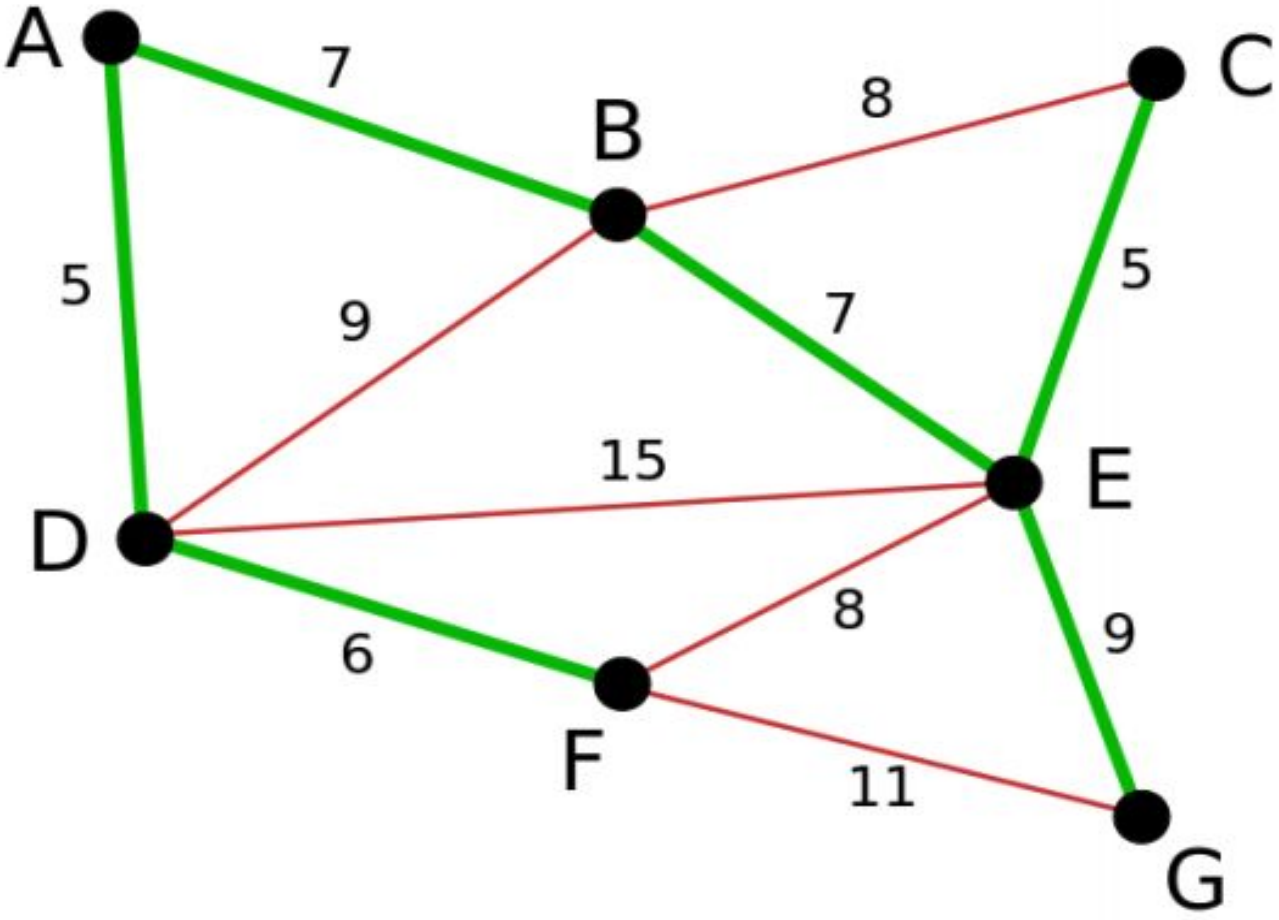

A
B
C
D
E
F
G
7
8
5
9
7
5
15
8
9
6
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
8
9
6
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
8
9
6
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
8
9
6
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
8
9
6
11


A
B
C
D
E
F
G
7
8
5
9
7
5
15
8
9
6
11

```
int kruskal(vector<pair<int ,pair<int ,int > > > ejes, int n){
  sort(ejes.begin(),ejes.end());
  initUF(n);
  int num_ejes = 0;
  long long costo = 0;
  for(int i=0;i<ejes.size();i++){
    // Si no pertenecen a la misma componente
    if(find(ejes[i].second.first) != find(ejes[i].second.second)){
      num_ejes++;
      costo+=ejes[i].first;
      // Si ya forme el arbol
      if(num_ejes == n-1)
        return costo;
      // junto dos componentes
      union(ejes[i].second.first,ejes[i].second.second);
    }
  }
  return -1;
}
```