Minimum Spanning Tree y Union Find

Emanuel Lupi

Universidad Nacional de Córdoba
FaMAF

11 de Mayo de 2018

Problema

Dado un grafo ponderado (con pesos). Lo queremos podar (quitándole ejes) sin dejar a ninguna arista sola y teniendo el arbol de menor peso

Definición: Sea $G = (V, E)$ un grafo ponderado conexo. Un árbol generador mínimo de G es un árbol $G_0 = (V, E_0)$ tal que la suma de los costos de las aristas de G_0 es mínima. Uno de los algoritmos utilizados es el algoritmo de Kruskal.

Algoritmo de Kruskal

Antes de ver ćomo funciona y saber qué hace el algoritmo de Kruskal hay que conocer una estructura de datos para resolver eficientemente un subproblema en el algoritmo de Kruskal.

Union Find data structure

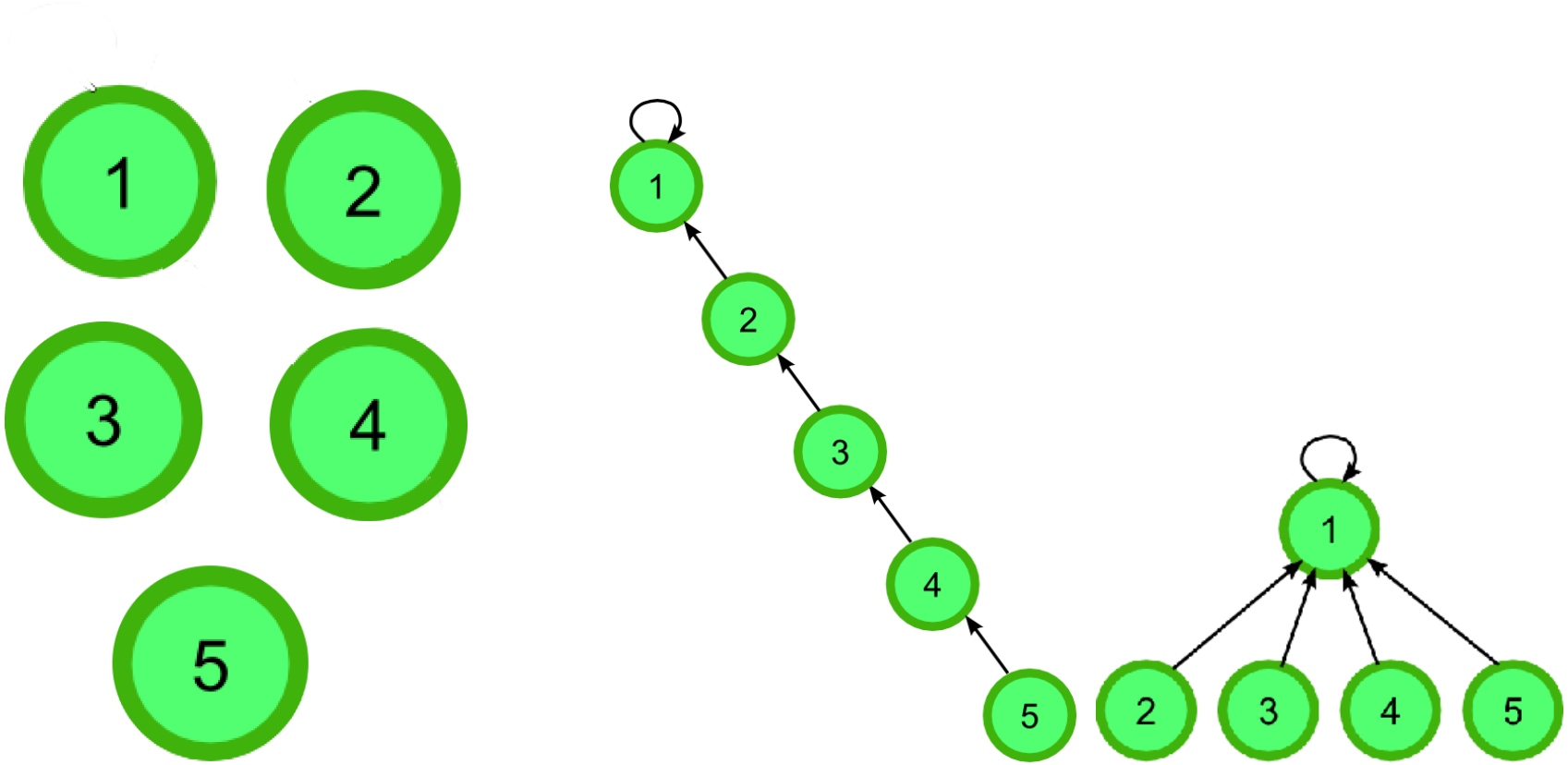
Teniendo conjuntos disjuntos esta estructura nos permite hacer de manera eficiente los siguientes procesos o funciones:

- Union: Une dos subconjuntos en uno solo.
- Find: Determina a cual subconjunto pertenece un elemento.

Idea

Digamos que cada conjunto disjuntos puede tener un representante del conjunto.
También permitiremos que un representante tenga un representante.
Entonces para saber si dos elementos arbitrarios están dentro de un conjunto solo hay que preguntar por sus representantes supremo.
Y para unirlos hay que hacer que un representante sea el representante del otro conjunto.

Si todos los elementos de la izquierda pertenecieran a un conjunto, dos posibles representaciones son los arboles de dependencias que están a la derecha.

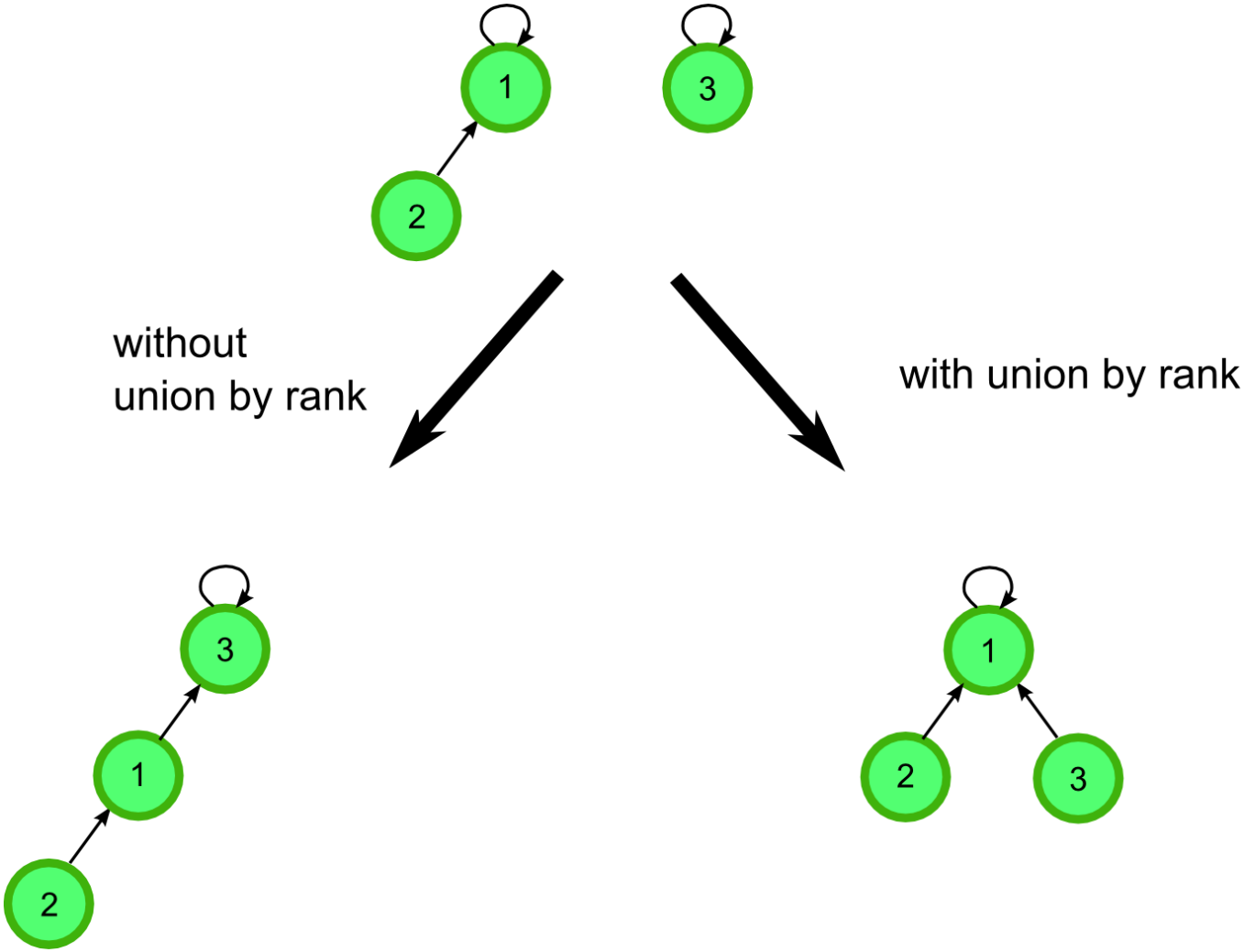


Primera implementación

```
int reprs[1000];

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

int find(int x){
  while(reprs[x] != x){
    x = reprs[x];
  }
  return reprs[x];
}


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

  //reprs[repx] = reprs[repy];
  reprs[repx] = repy;
}
```

Segunda implementación

Introducimos el rank para evitar armar el un árbol con una rama muy extensa



Segunda implementación

```
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}
}

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;
  }
}
```

La Implementación

Las idea dentro son:

- Union by rank (recientemente vista).
- Path compression (por verse).

Para tener **La implementación** hay que cambiar el código en tres lugares:

- initUF: Agregar un ranker para no unir siempre el que tenga la cadena mas larga al que la cadena tiene mas chica.
- Find: Cuando se busca el representante de algún elemento podes actualizar la lista de dependencias. (Path compression)
- Union: Anexar siempre el representante que tenga menos rank con el de mayor. (Union by Rank)

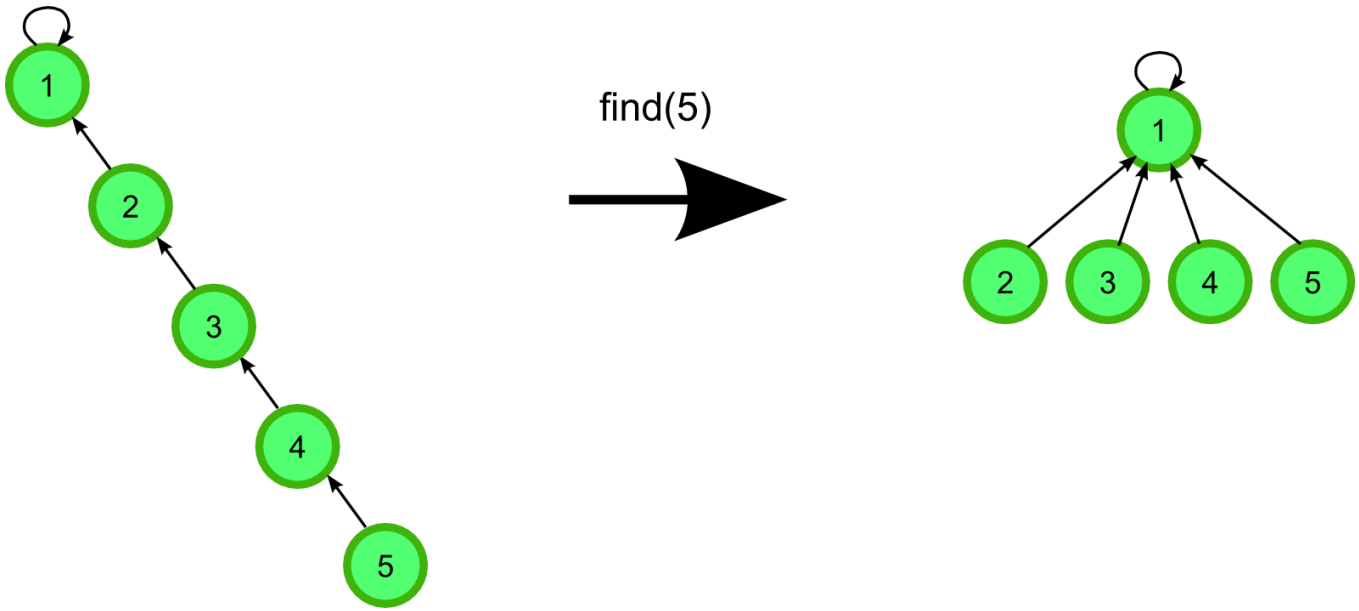
La Implementación

```
#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;
}
```

La Implementación

```
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;
  }
}
```

Path compression

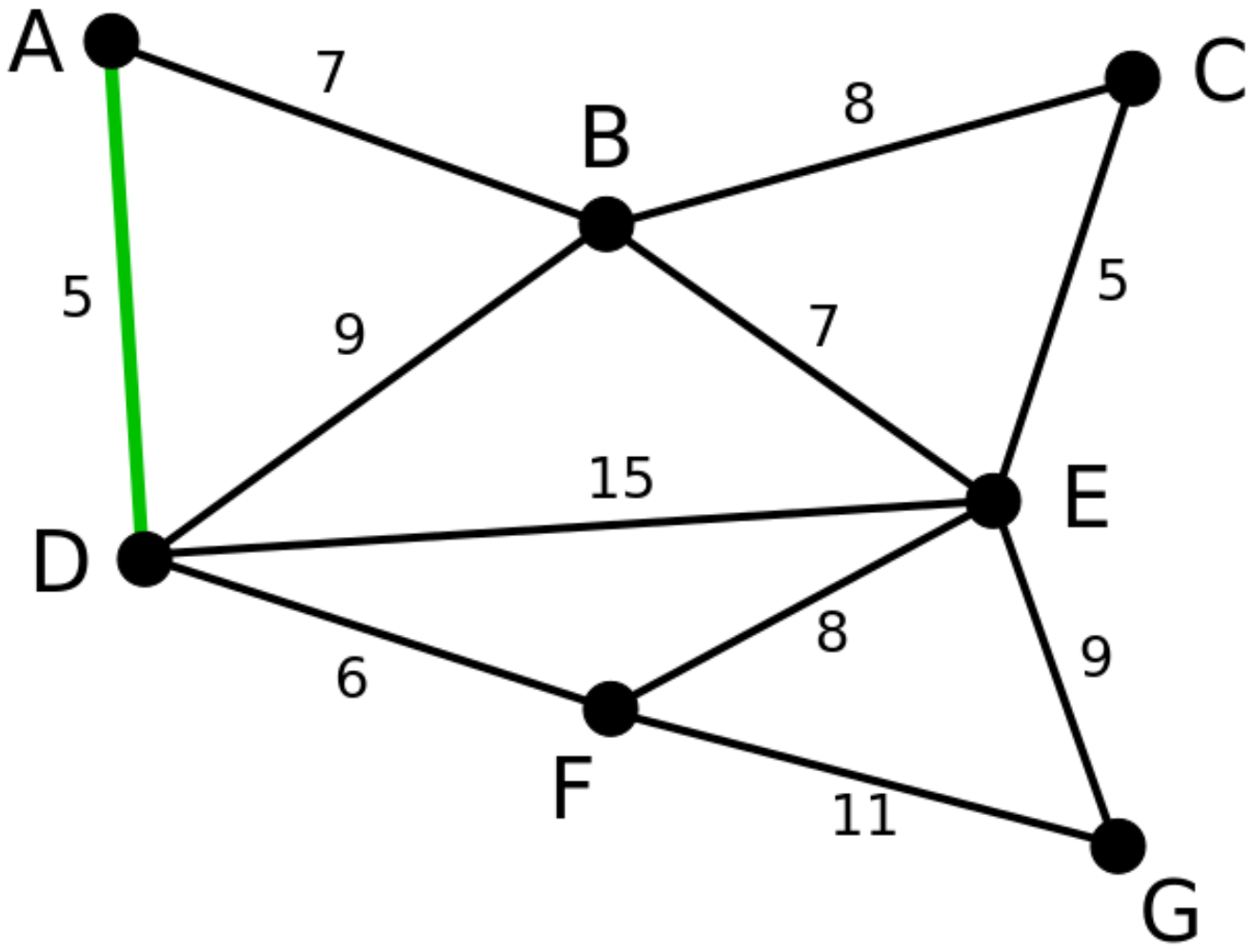


El algoritmo de Kruskal

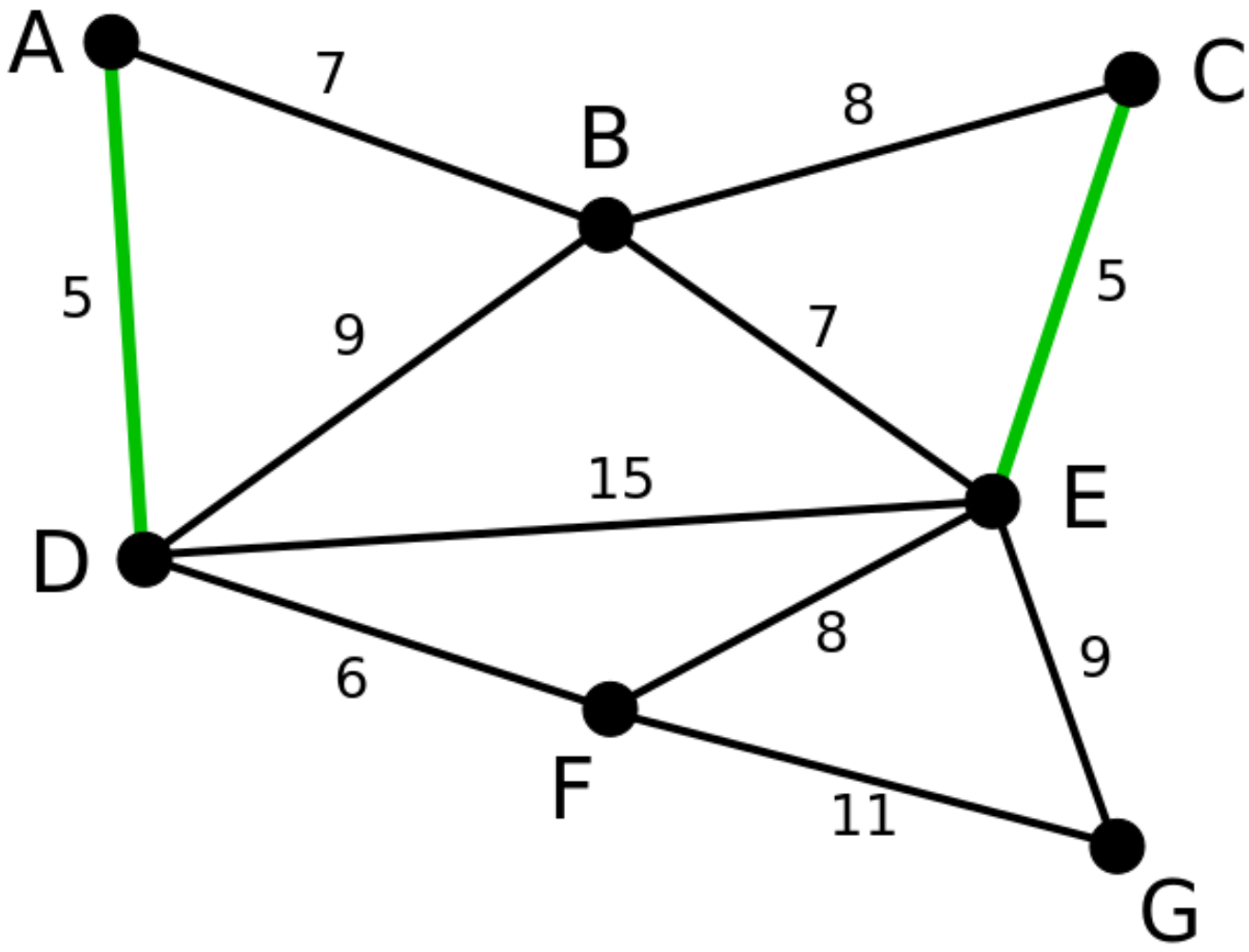
El algoritmo de Kruskal ordena las aristas por peso y va agregando desde la arista de menor peso hasta la de mayor peso, **evitando agregar las aristas que forman ciclos.**
Para saber si forman ciclos se usa la Union-Find. Y se usa de la siguiente manera:

- 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**.

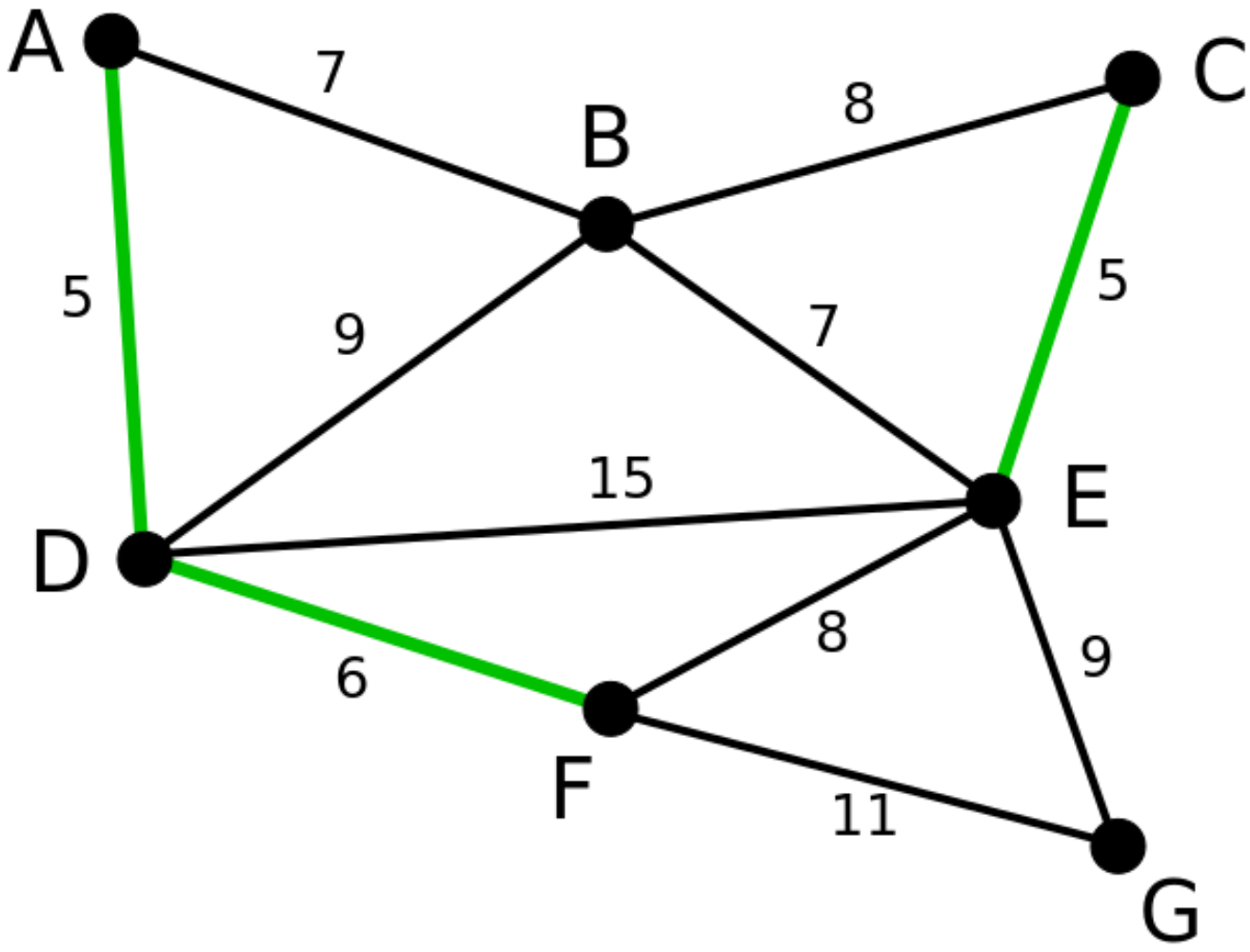
Secuencia



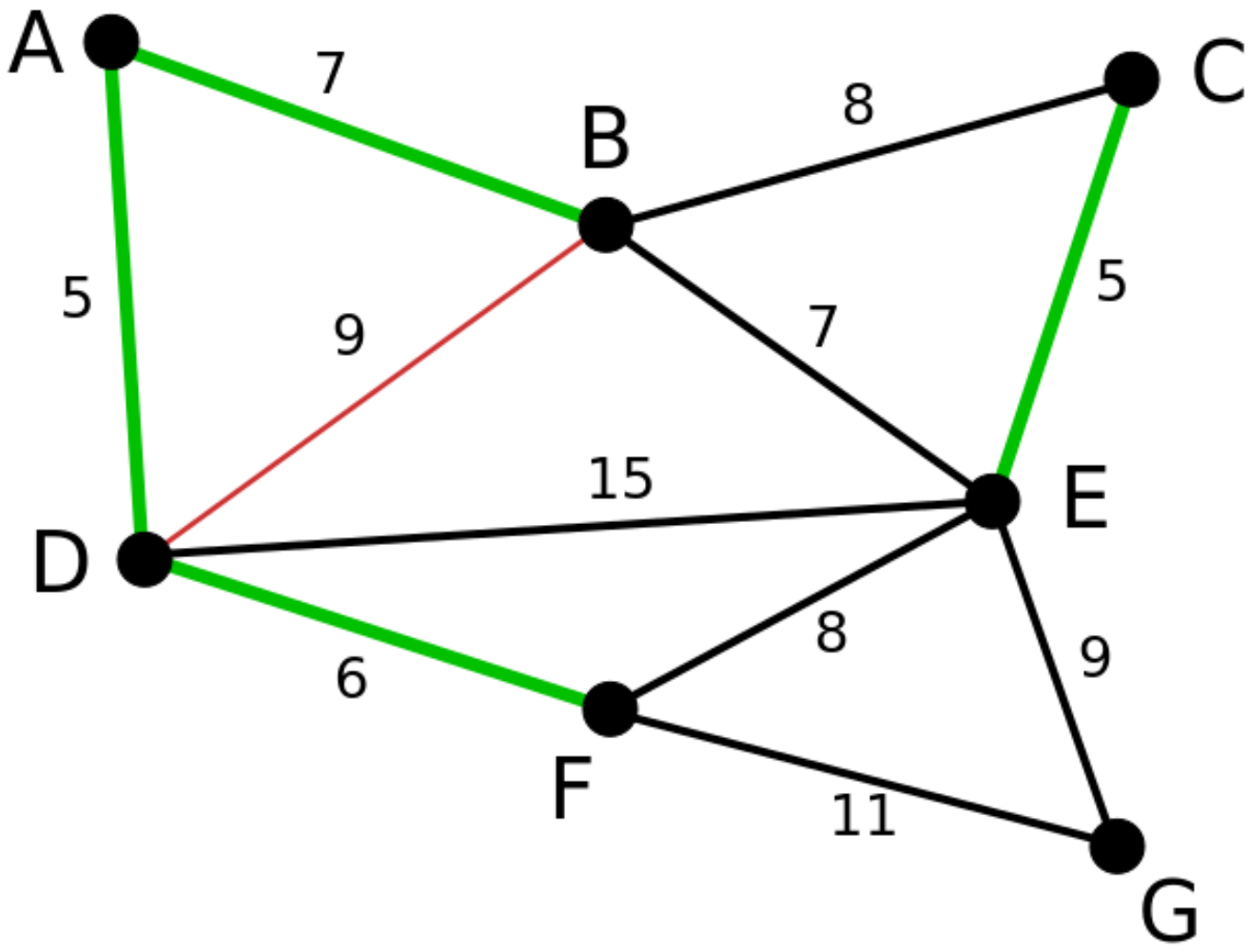
Secuencia



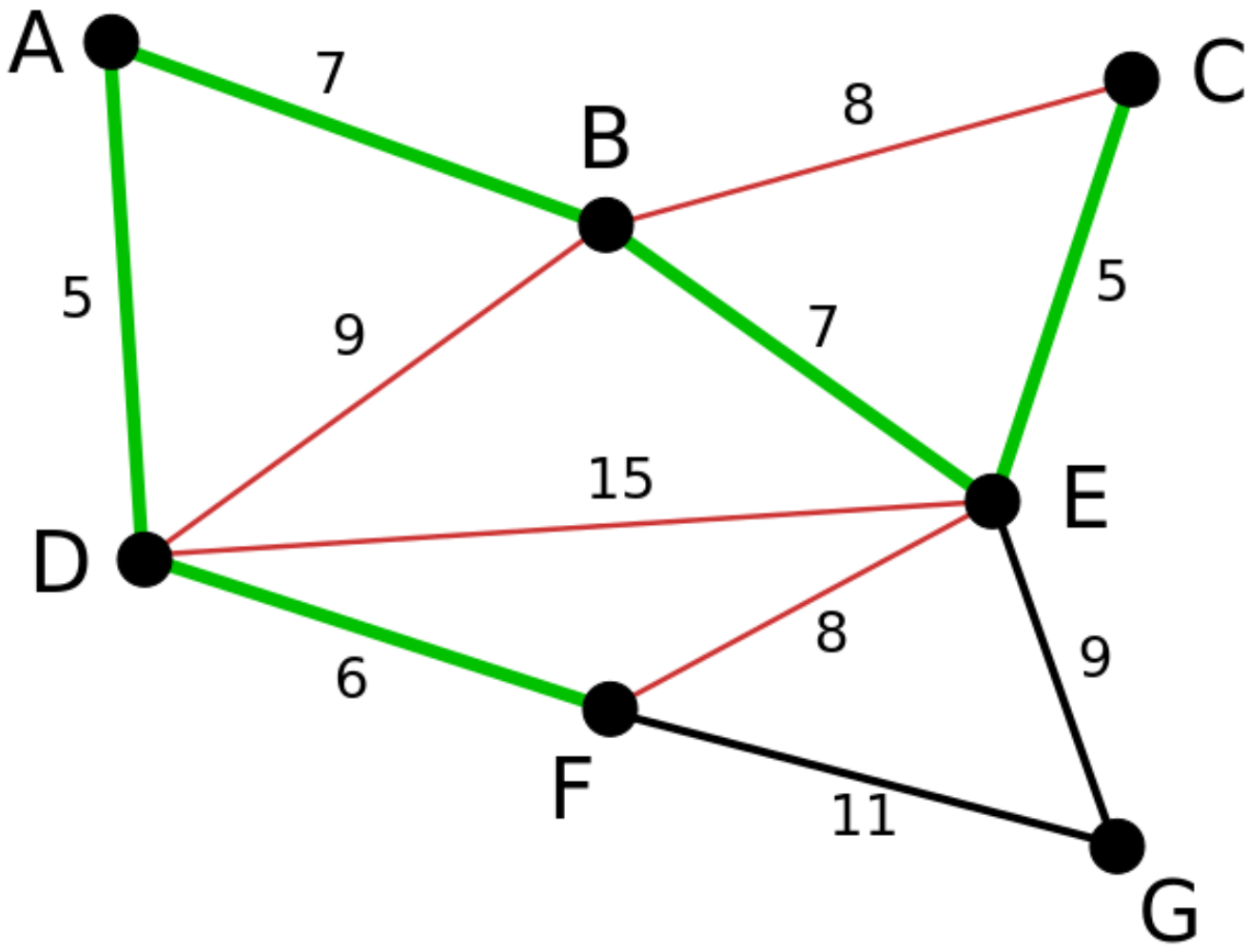
Secuencia



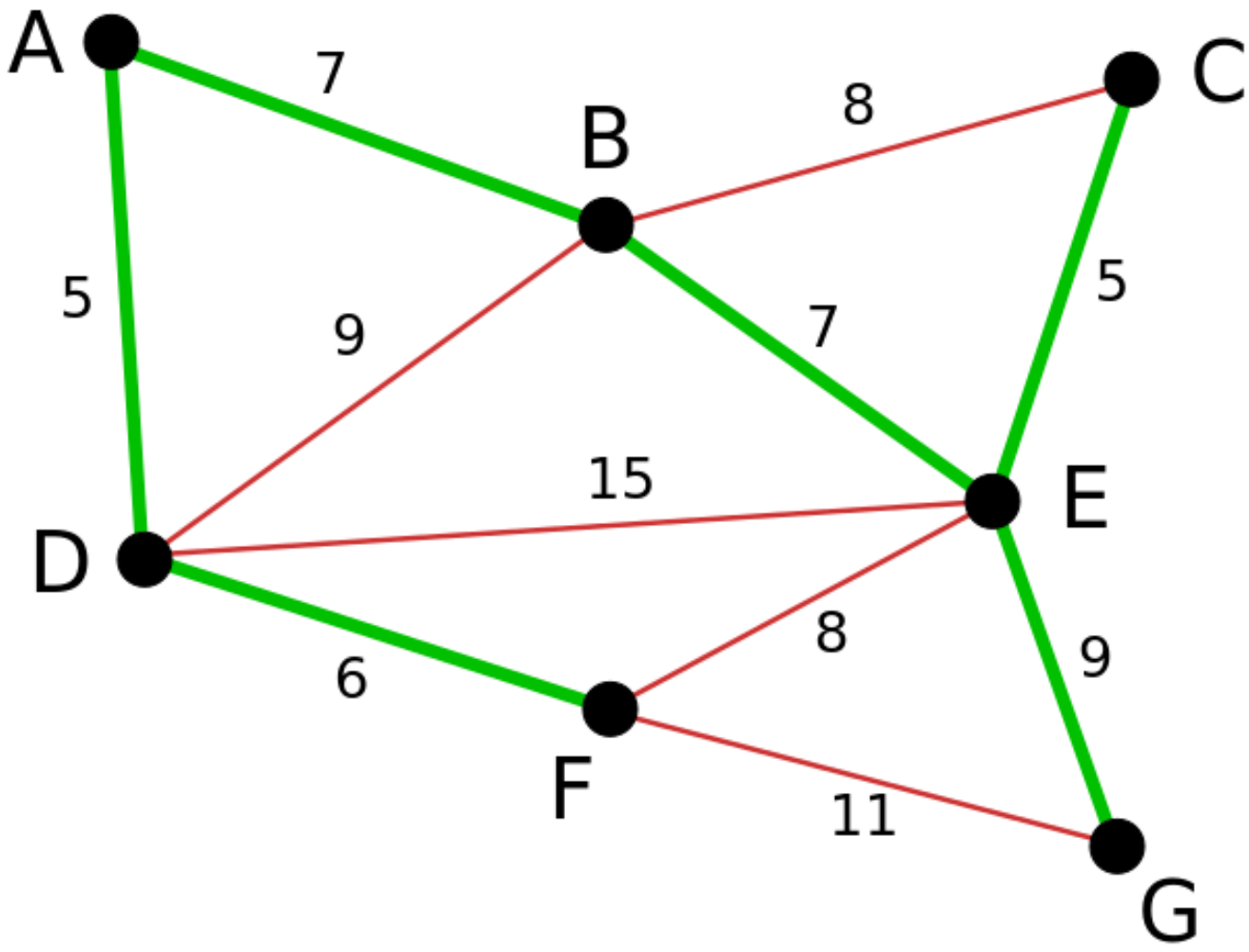
Secuencia



Secuencia



Secuencia



Secuencia

```
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;
}
```