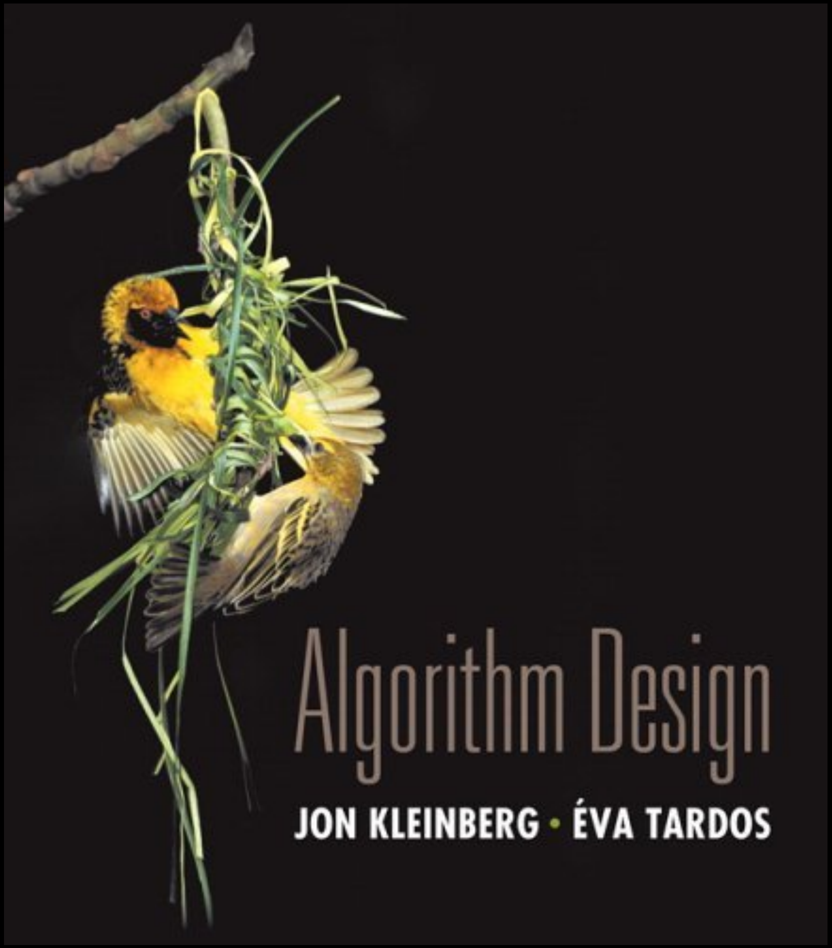


Capítulo 4

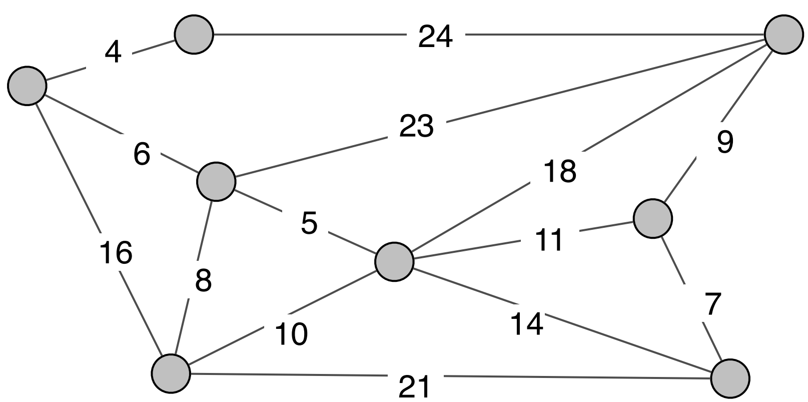
Algoritmos Greedy

Slides by Kevin Wayne.

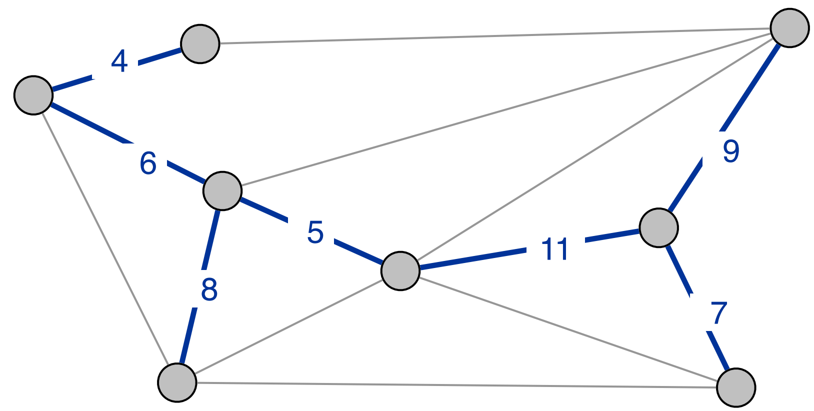
4.5 Árboles de cubrimiento de costo mínimo

Árbol de cubrimiento de costo mínimo

Minimum spanning tree (MST). Dado un grafo conexo G = (V, E) con valores reales en las aristas c_e, un MST es un subconjunto de aristas $T \subseteq E$ tal que T es un árbol de cubrimiento y la suma de los pesos de las aristas es mínima.



$G = (V, E)$



$T, \ \Sigma_{e \in T} c_e = 50$

Teorema de Cayley's. Hay n^{n-2} árboles de cubrimiento.

No se puede resolver con fuerza bruta

Algunas aplicaciones

- MST es un problema fundamental con aplicaciones diversas.
  - Diseño de redes.
    - telefónicas, eléctricas, hidráulicas, TV cable, computadoras, viales
  - Algoritmos de aproximación para problemas difíciles.
    - Problema del viajante

Algoritmo de Kruskal. Empezar con T = ϕ. Considerar las aristas en orden creciente de costo. Insertar la arista **e** en T al menos que al hacerlo cree un ciclo en T.

Algoritmo de Reverse-Delete. Empezar con **T = E**. Considerar las aristas en orden decreciente de costo. Eliminar la arista **e** de T a menos que al hacerlo desconecte a T.

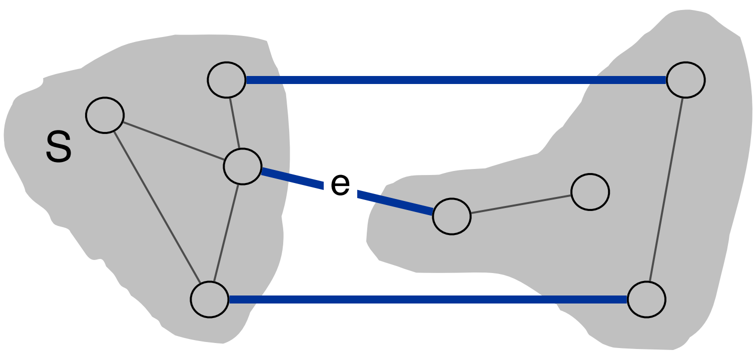
Algoritmo de Prim. Empezar con un nodo raíz **s** y hacer crecer el árbol **T** desde **s** hacia afuera. En cada paso, agregar la arista de menor costo **e** a T, que tiene exactamente uno de sus extremos en **T**.

Observación. Los tres algoritmos producen un MST.

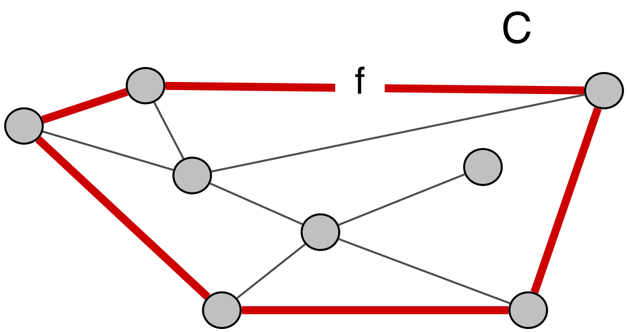
Para simplificar asumimos. Todos los costos de las aristas $\mathbf{c_e}$ son distintos.

Propiedad de corte. Sea **S** cualquier subconjunto de nodos, y sea **e** la arista de menor costo con exactamente un extremo en **S**. Entonces el MST contiene a **e**.

Propiedad de ciclo. Sea **C** cualquier ciclo, y sea **f** la arista de mayor costo que pertenece a **C**. Entonces el MST no contiene a **f**.



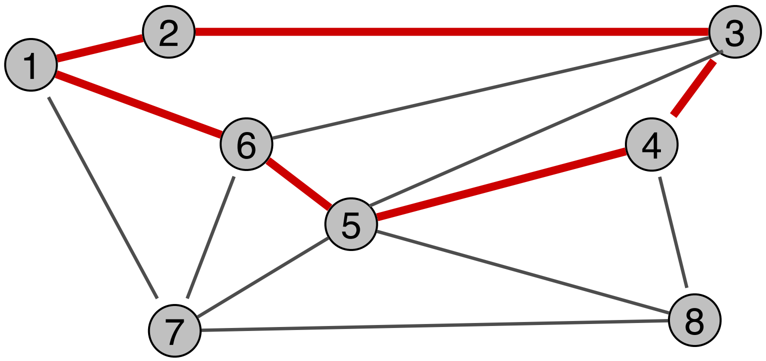
e está en el MST



f no está en el MST

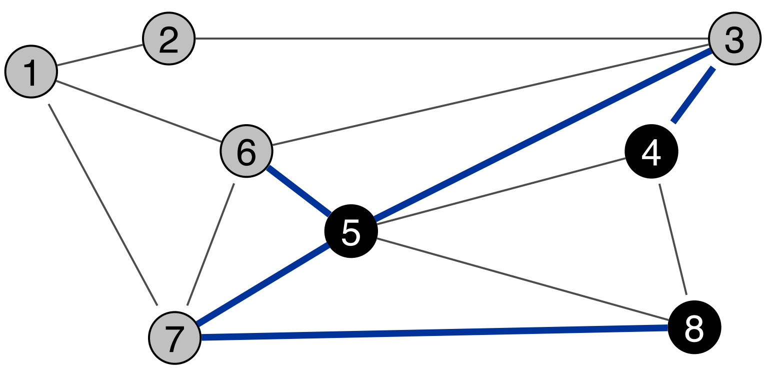
Ciclos y cortes

Ciclo. Conjunto de aristas de la forma a-b, b-c, c-d, …, y-z, z-a.



Ciclo C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

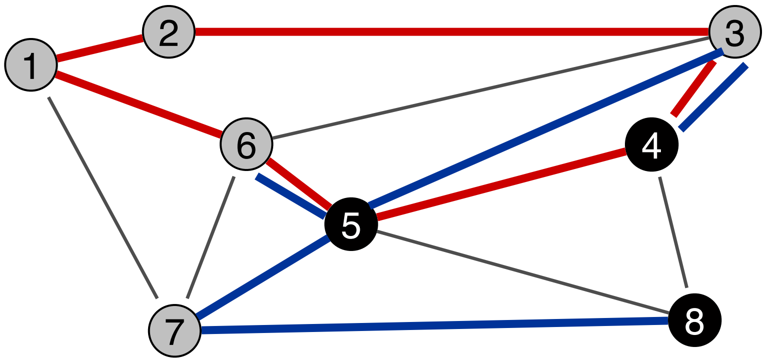
Conjunto de corte (*cutset*). Un corte es un subconjunto de nodos **S**. El conjunto de corte, *cutset*, correspondiente **D** es el subconjunto de aristas con exactamente un extremo en **S**.



Corte S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

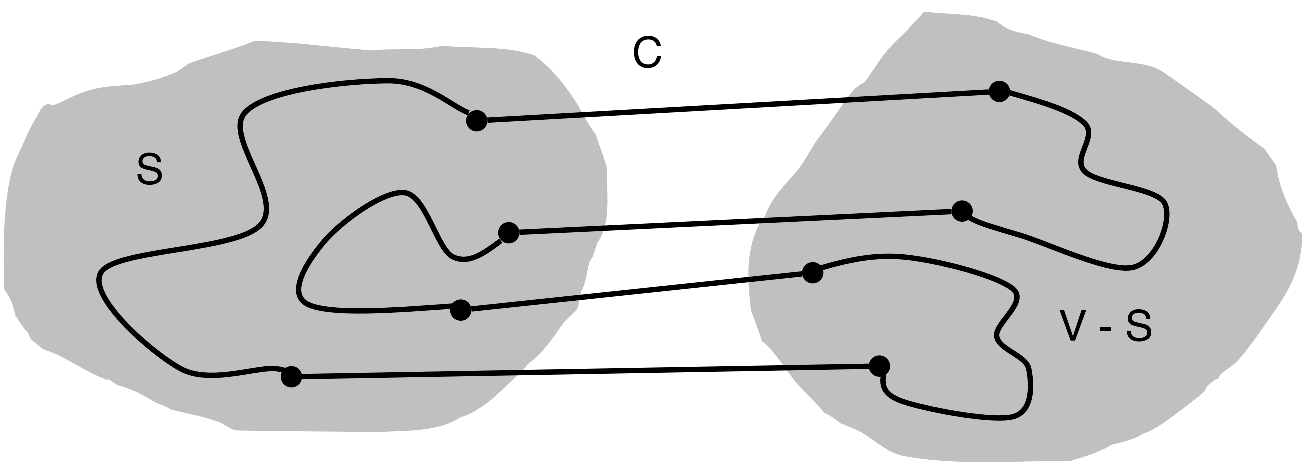
Intersección Ciclo-Cutset

Proposición. Un ciclo y un cutset intersectan en un número **par** de aristas.



Ciclo C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersección = 3-4, 5-6

Dem. (por la imagen)

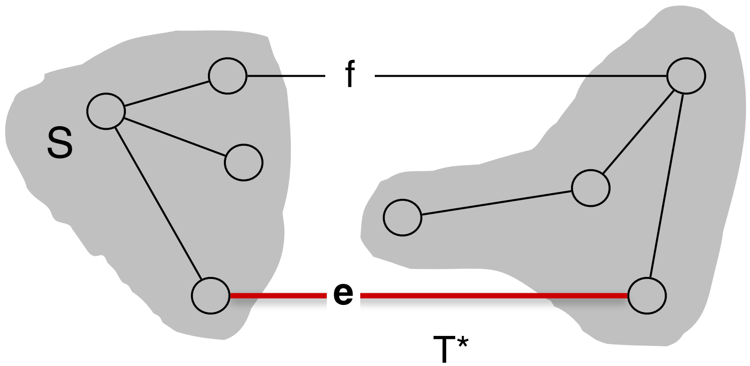


Para simplificar asumimos. Todos los costos de las aristas $\mathbf{c_e}$ son distintos.

Propiedad de corte. Sea **S** cualquier subconjunto de nodos, y sea **e** la arista de menor costo con exactamente un extremo en **S**. Entonces el MST **T*** contiene a **e**.

Dem. (Argumento de intercambio - Por absurdo)

- Supongamos que **e** no pertenece a **T***, y veamos qué sucede.
- Agregar **e** a **T*** crea un ciclo **C** en **T***.
- La arista **e** está tanto en **C** y en el *cutset* **D** correspondiente a **S** $\Rightarrow$ existe otra arista, digamos **f**, que está tanto en **C** como en **D**.
- Sea $\mathbf{T'} = \mathbf{T^*} \cup \{\, \mathbf{e} \,\} - \{\, \mathbf{f} \,\}$ también es un árbol de cubrimiento.
- Como $\mathbf{c_e} < \mathbf{c_f}$, **costo(T') < costo(T*)**.
- Llegamos a una contradicción. ▪

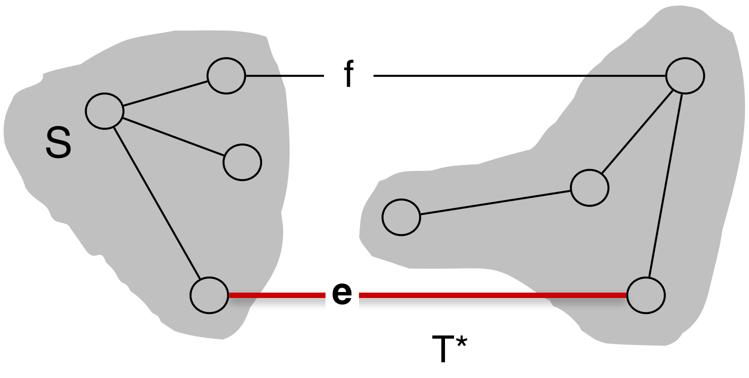


Para simplificar asumimos. Todos los costos de las aristas $\mathbf{c_e}$ son distintos.

Propiedad de ciclo. Sea **C** cualquier ciclo, y sea **f** la arista de mayor costo que pertenece a **C**. Entonces el MST **T*** no contiene a **f**.

Dem. (Argumento de intercambio - Por absurdo)

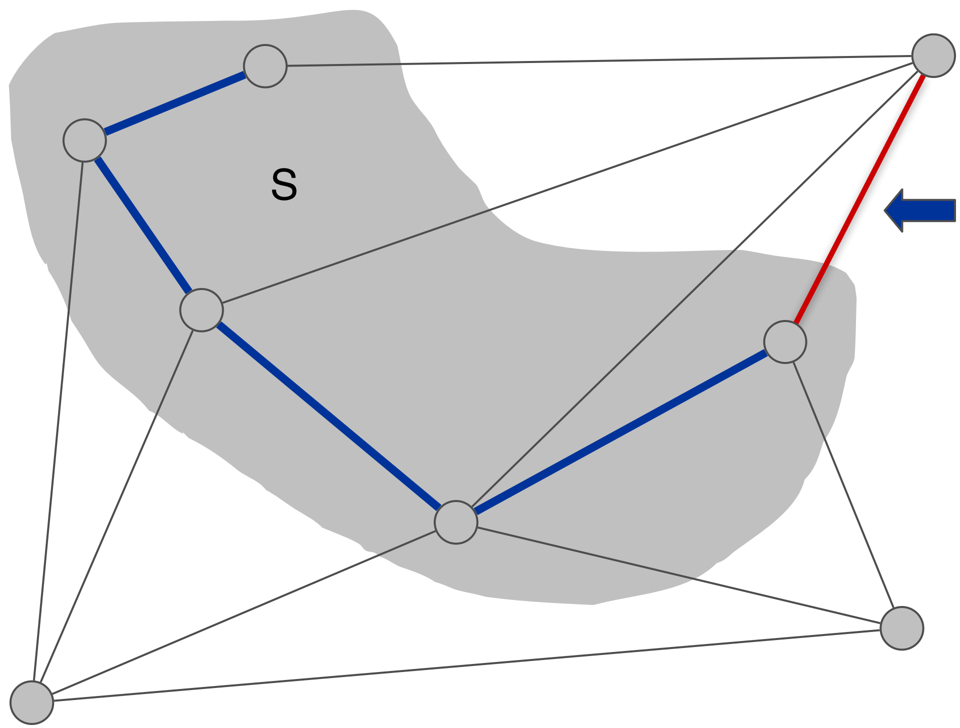
- Supongamos que **f** pertenece a **T***, y veamos qué sucede.
- Eliminar **f** de **T*** crea un corte **S** en **T***.
- La arista **f** está tanto en el cliclo **C** como en el *cutset* **D** correspondiente a **S** $\Rightarrow$ existe otra arista, digamos **e**, que está tanto en **C** como en **D**.
- $\mathbf{T' = T^* \cup \{e\} - \{f\}}$ también es un árbol de cubrimiento.
- Como $\mathbf{c_e < c_f}$, $\mathbf{costo(T') < costo(T^*)}$.
- Esto es una contradicción. ▪



Algoritmo de Prim: Prueba de corrección

Algoritmo de Prim. [Jarník 1930, Prim 1957, Dijkstra 1959]

- Inicializar **S** = {cualquier nodo}.
- Aplicar la propiedad de corte a **S**.
- Agregar la arista de menor costo en el cutset correspondiente a **S** a **T**, y agregar un nuevo nodo explorado **u** a **S**.



Implementación: Algoritmo de Prim

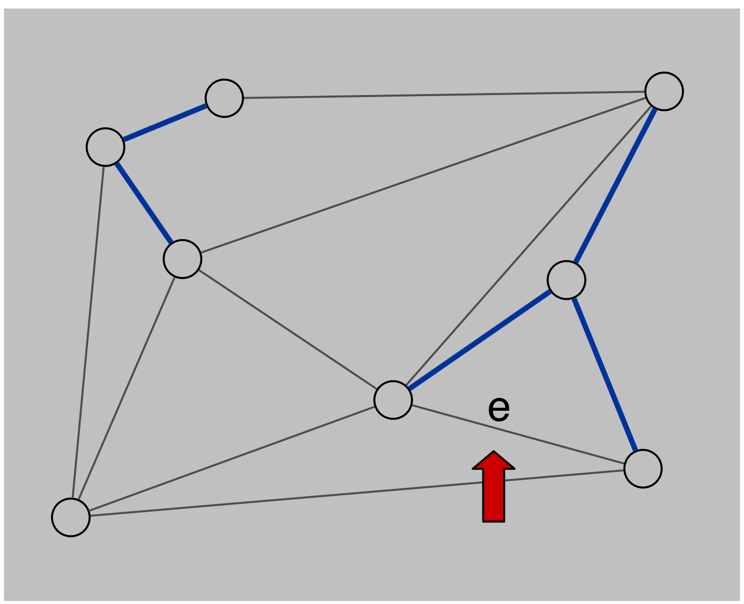
Implementación. Utilizar una cola de prioridad como en Dijkstra.

- Mantener un conjunto de nodos explorados **S**.
- Por cada nodo no explorado **v**, mantener un costo de conexión **a[v]** = *costo de la arista más barata desde* **v** *a un nodo en* **S**.
- **O(n²)** con un array; **O(m log n)** con un **heap**.

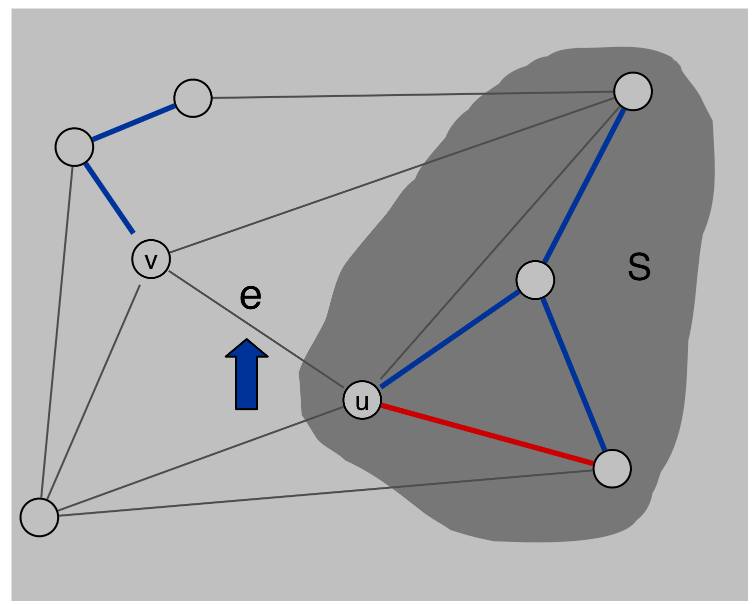
```
Prim(G, c) {
  foreach (v ∈ V) a[v] ← ∞
  Inicializar una cola de prioridad vacía Q
  foreach (v ∈ V) insertar v en Q
  Inicializar el conjunto de nodos explorados S ← φ

  while (Q no sea vacía) {
    u ← eliminar el mínimo elemento de Q
    S ← S ∪ { u }
    foreach (arista e = (u, v) incidente a u)
       if ((v ∉ S) y (c_e < a[v]))
          decrementar prioridad a[v] = c_e
}
```

Algoritmo de Kruskal. [Kruskal, 1956]

- Considerar las aristas en orden ascendente de costo.
- **Caso 1:** Si agregar **e** a **T** crea un ciclo, descartamos **e** según la propiedad del ciclo.
- **Caso 2:** De otra manera, insertar **e = (u, v)** en **T** según la propiedad de corte, donde **S** = conjunto de nodos en la componente conexa de u.



Case 1



Case 2

Implementación: Algoritmo de Kruskal

Implementación. Utilizar la estructura union-find.

- Construir el conjunto de aristas T del MST.
- Mantener un conjunto por cada componente conexa.
- O(m log n) por sorting y O(m α (m, n)) por union-find.

$m \le n^2 \Rightarrow \log m$ es O(log n)

Kruskal(G, c) {
  Ordenar aristas según su costo, $c_1 \le c_2 \le \ldots \le c_m$.
  $T \leftarrow \phi$

  foreach ($u \in V$) crear un conjunto que contenga al singleton u

  for i = 1 to m
    (u,v) = e_i
    if (u y v están en diferentes conjuntos) {
      $T \leftarrow T \cup \{e_i\}$
      unir los conjuntos que contienen a u y v
    }
  retornar T
}

están u y v en diferentes componentes conexas?

unir las dos componentes

Implementación: Algoritmo de Kruskal

Implementación. Utilizar la estructura union-find.

- Construir el conjunto de aristas T del MST.
- Mantener un conjunto por cada componente conexa.
- $O(m \log n)$ por sorting y $O(m\, \alpha\, (m, n))$ por union-find.

$m \le n^2 \Rightarrow \log m$ es $O(\log n)$

```
Kruskal(G, c) {
  Ordenar aristas según su costo, c_1 ≤ c_2 ≤ ... ≤ c_m.
  T ← φ

  MakeUnionFind (v ∈ V)

  for i = 1 to m
    (u,v) = e_i
    if (FIND(u) != FIND(v)) {
      T ← T ∪ {e_i}
      UNION(FIND(u), FIND(v))
    }
  retornar T
}
```

Implementación: Union-Find

Implementación. Utilizar la estructura union-find.

- Utilizamos un arreglo ***componente[v]***, con una entrada por nodo indicando a qué componente pertenece.
- Utilizamos un arreglo de listas ***l_componente[A]***, con una entrada por componente guardando qué nodos pertenecen a la componente.
- Utilizamos un arreglo de tamaños ***size[A]***, con una entrada por componente guardando su tamaño.

```
Union(A, B) {
  if size(A) >= size(B)
     • Recorrer la l_componente[B] y definir componente[v] = A para
       todo v en la lista.
     • Concatenar la lista de B al final de la lista de A.
     • Hacer size(A) = size(A) + size(B)
  else …
}
```

Implementación: Union-Find

Implementación. Utilizar la estructura union-find.

- El paso *más costoso* en tiempo de ejecución en el peor caso de la ***union*** es recorrer una de las listas re-asignando las componentes.

- Si los dos conjuntos tienen tamaño ***n/2***, este paso es **O(n)**.

- Sin embargo, realizando un análisis *amoritzado*, podemos acotar el tiempo de ejecución de ***k*** operaciones, y encontrar una cota más ajustada.

Prop. La ejecución de cualquier secuencia de ***k*** operaciones de **Union** requiere un tiempo de ejecución ***O(k . log (k))***.

Implementación: Union-Find

Prop. La ejecución de cualquier secuencia de ***k*** operaciones de **Union** requiere un tiempo de ejecución ***O(k . log (k))***.

Dem.

- La ejecución de ***k*** operaciones **Union** involucra *no más* de **2k** elementos de **S**, ya que en cada **Union** o tengo 2 elementos nuevos, o 1 nuevo, o ninguno nuevo. (1)
- Veamos la cantidad máxima de actualizaciones individuales de componente que puede recibir un elemento particular.
- Sea **v** perteneciente a **S** cualquiera.
- Cada vez que ***componente[v]*** se actualiza, el tamaño de la componente al menos duplica su tamaño, ya que sólo se actualiza la componente de **v** si es la de menor tamaño.
- Como el tamaño final del conjunto no puede superar **2k** (1), la cantidad de actualizaciones está acotada superiormente por **log(2k)**, que es **O(log(k))**.
- El tiempo total de ejecución, sumando sobre todos los elementos que se actualizan, es **O(k.log(k))**.