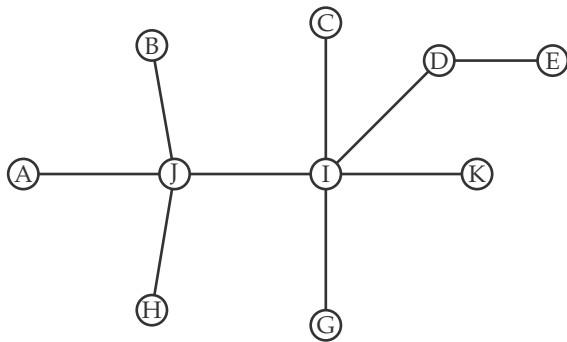


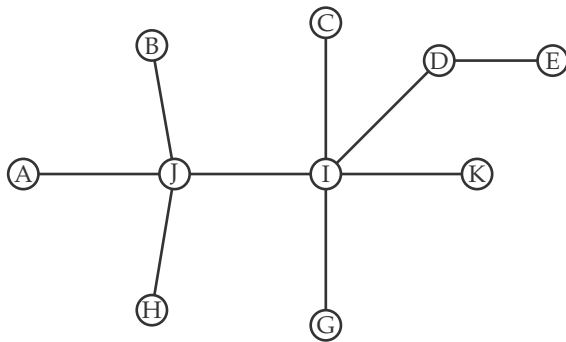
ÁRBOL GENERADOR MÍNIMO

Tecnología Digital V: Diseño de Algoritmos
Universidad Torcuato Di Tella



Definición

Un **árbol** es un grafo conexo sin circuitos.

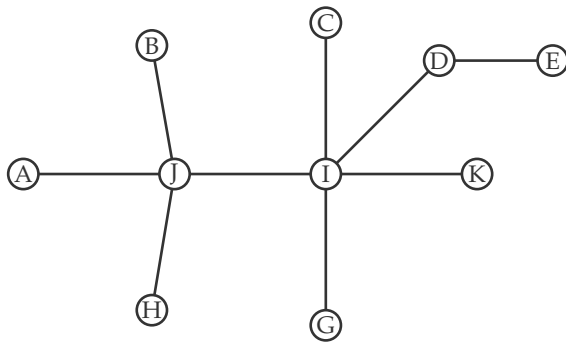


Definición

Un **árbol** es un grafo conexo sin circuitos.

Definición

Una **hoja** es un nodo de grado 1.



Definición

Un **árbol** es un grafo conexo sin circuitos.

Definición

Una **hoja** es un nodo de grado 1.

Teorema

Todo árbol no trivial tiene al menos dos hojas.

Teorema

Dado un grafo $G = (V, E)$ son equivalentes:

1. G es un árbol.
2. G no tiene circuitos y $m = n - 1$.
3. G es conexo y $m = n - 1$.
4. G es un grafo sin circuitos simples, pero si se agrega cualquier arista e a G resulta un grafo con exactamente un circuito simple, y ese circuito contiene a e .
5. Existe exactamente un camino simple entre todo par de nodos.
6. G es conexo, pero si se quita cualquier arista a G queda un grafo no conexo.

Árbol generador

Dado un grafo G , un **árbol generador** de G es un subgrafo de G que es un árbol y tiene el mismo conjunto de vértices que G .

Árbol generador

Dado un grafo G , un **árbol generador** de G es un subgrafo de G que es un árbol y tiene el mismo conjunto de vértices que G .

Longitud de un árbol generador

Sea $T = (V, E)$ un árbol y $l : E \rightarrow \mathbb{R}$ una función que asigna longitudes (o pesos) a las aristas de T . Se define la **longitud** de T como $l(T) = \sum_{e \in E} l(e)$.

Árbol generador

Dado un grafo G , un **árbol generador** de G es un subgrafo de G que es un árbol y tiene el mismo conjunto de vértices que G .

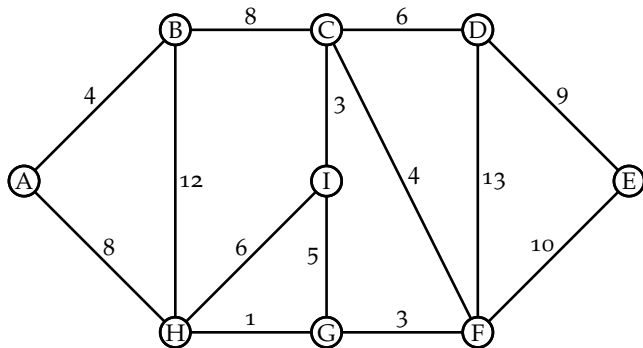
Longitud de un árbol generador

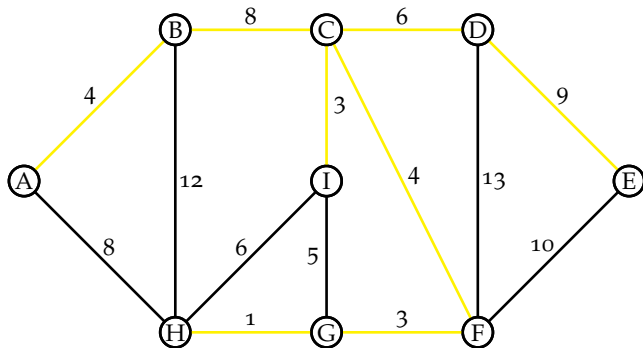
Sea $T = (V, E)$ un árbol y $l : E \rightarrow R$ una función que asigna longitudes (o pesos) a las aristas de T . Se define la **longitud** de T como $l(T) = \sum_{e \in E} l(e)$.

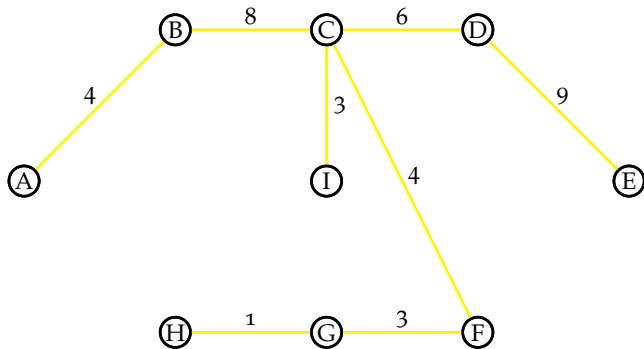
Árbol generador mínimo

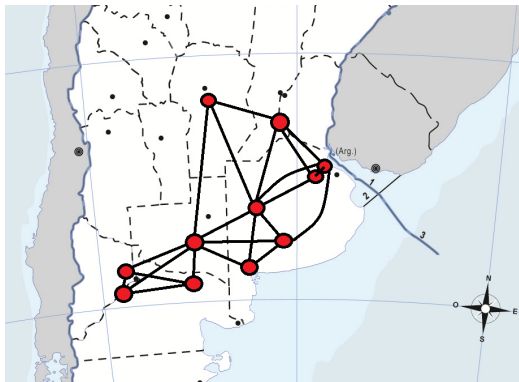
Dado un grafo $G = (V, E)$ un **árbol generador mínimo** T de G es un árbol generador de G de mínima longitud, es decir

$$l(T) \leq l(T') \quad \text{para todo } T' \text{ árbol generador de } G.$$



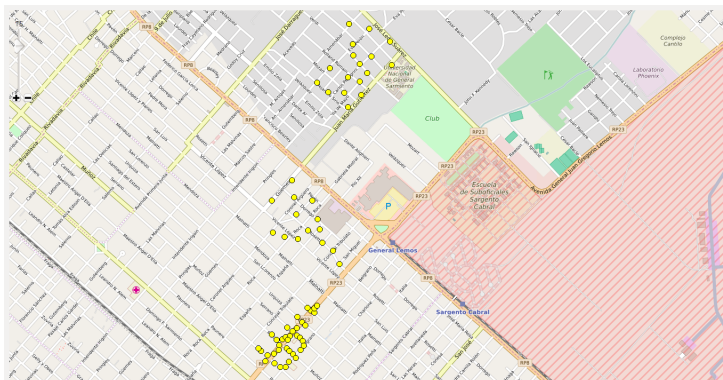






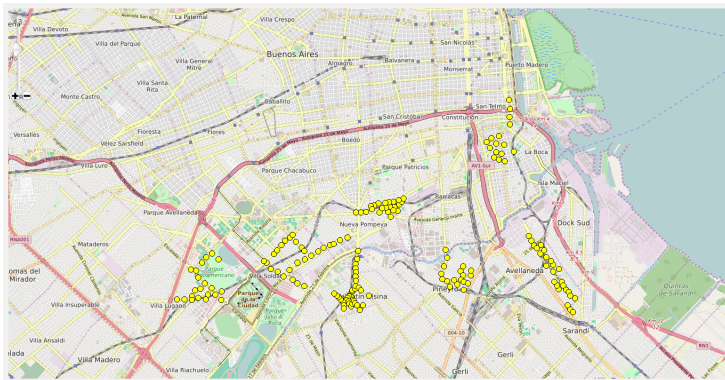
Conexión con costo mínimo de una **red eléctrica**.

- Si las longitudes de las aristas representan el **costo** de construir una línea de alta tensión, un árbol generador mínimo proporciona la estrategia de menor costo para interconectar la red completa.



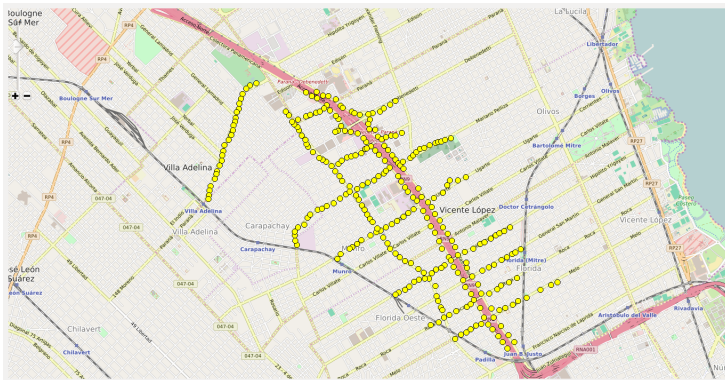
Algoritmos de clustering.

- Los algoritmos tradicionales de clustering están basados en distancias,



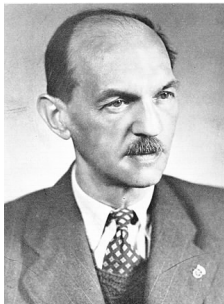
Algoritmos de clustering.

- Los algoritmos tradicionales de clustering están basados en distancias, pero pueden fallar cuando hay clusters con puntos “alineados” no necesariamente todos cercanos entre sí.



Algoritmos de clustering.

- Los algoritmos tradicionales de clustering están basados en distancias, pero pueden fallar cuando hay clusters con puntos “alineados” no necesariamente todos cercanos entre sí.



Vojtech Jarník
(1897–1970)



Robert Prim
(1921–2021)



Edsger Dijkstra
(1930–2002)

$V_T := \{u\}$ (u cualquier vértice de G)

$E_T := \emptyset$

$i := 1$

mientras $i \leq n - 1$ **hacer**

 elegir $e = (u, v) \in E$ tal que $l(e)$ sea mínima
 entre las aristas que tienen un extremo

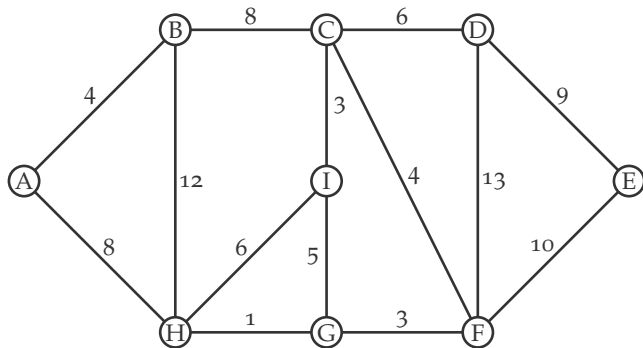
$u \in V_T$ y el otro $v \in V \setminus V_T$

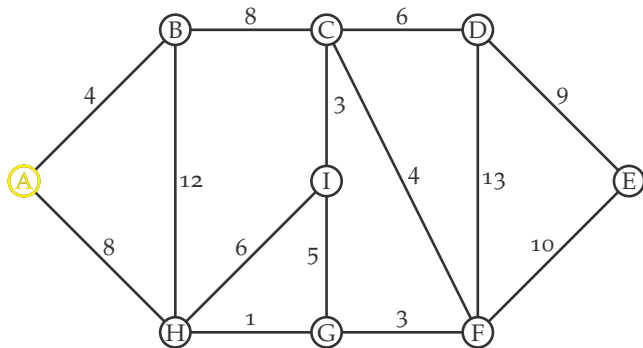
$E_T := E_T \cup \{e\}$

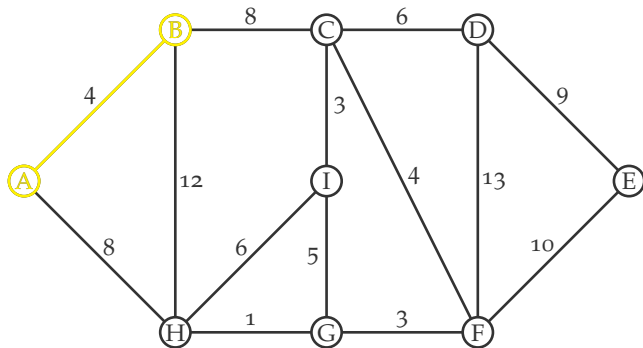
$V_T := V_T \cup \{v\}$

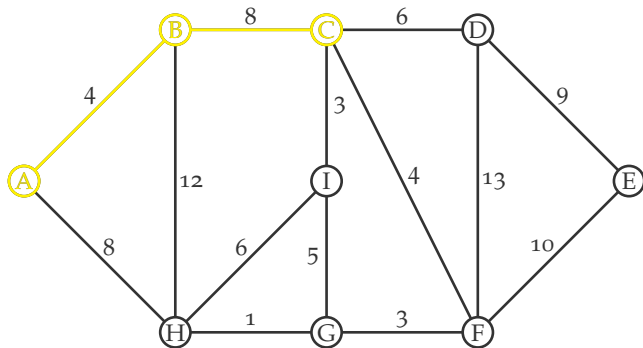
$i := i + 1$

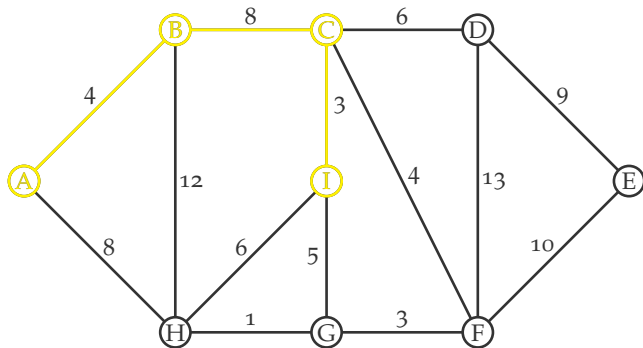
retornar $T = (V_T, E_T)$

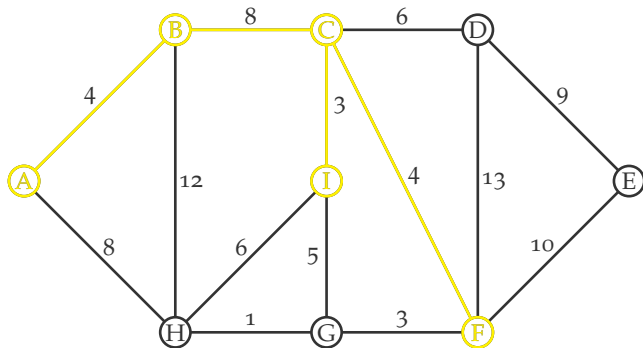


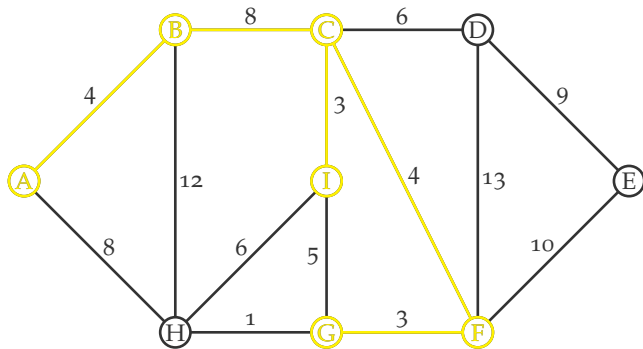


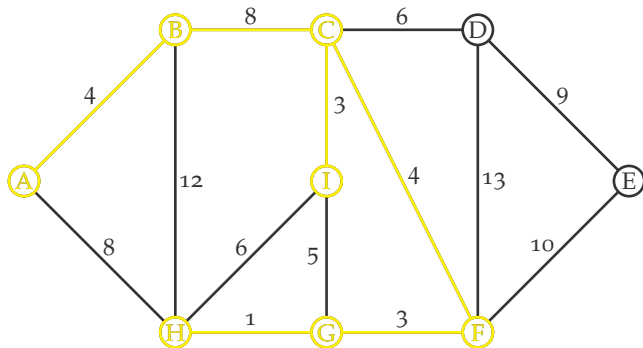


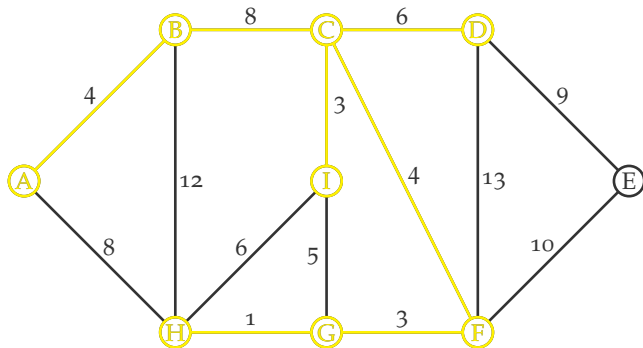


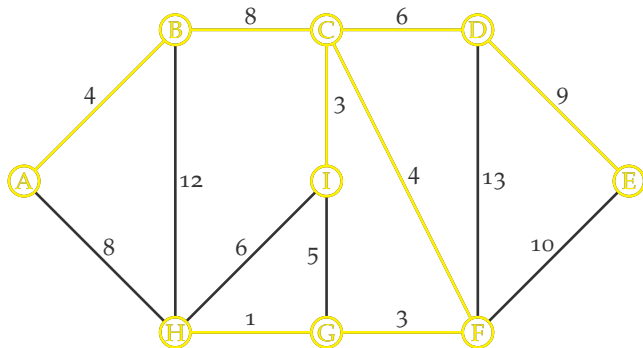














Joseph Kruskal
(1928–2010)

$E_T := \emptyset$

$i := 1$

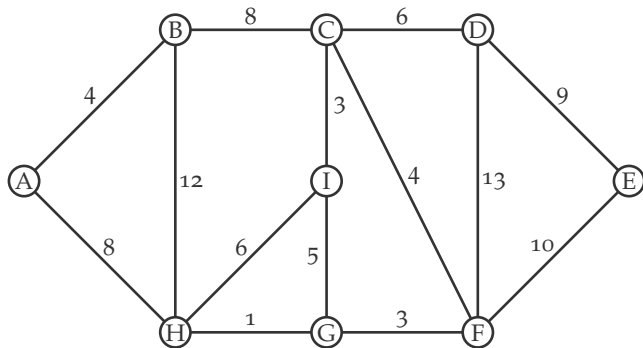
mientras $i \leq n - 1$ **hacer**

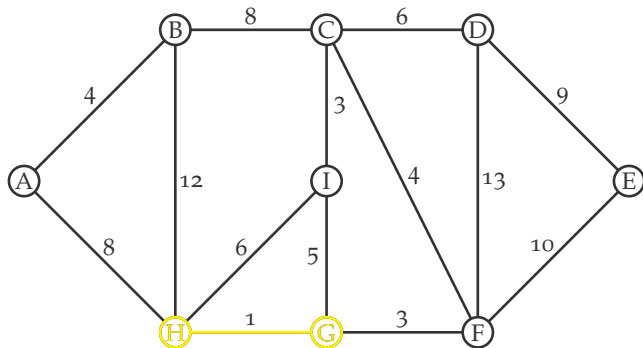
 elegir $e \in E$ tal que $l(e)$ sea mínima entre las
 aristas que no forman circuito con las
 aristas que ya están en E_T

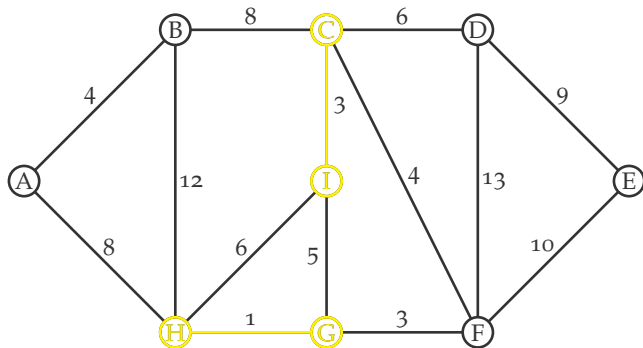
$E_T := E_T \cup \{e\}$

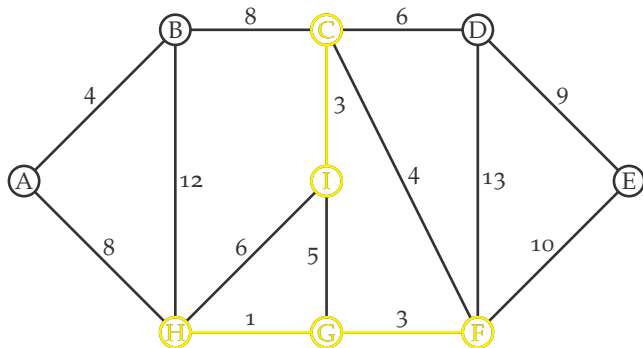
$i := i + 1$

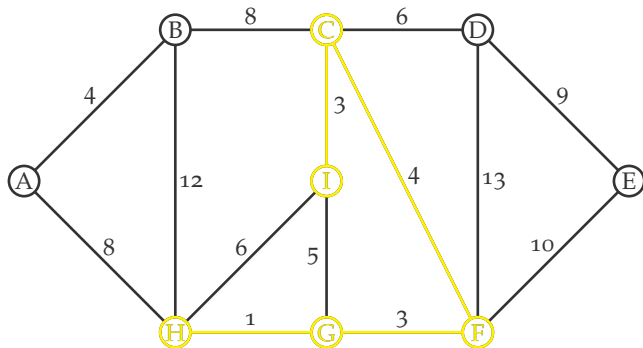
retornar $T = (V, E_T)$

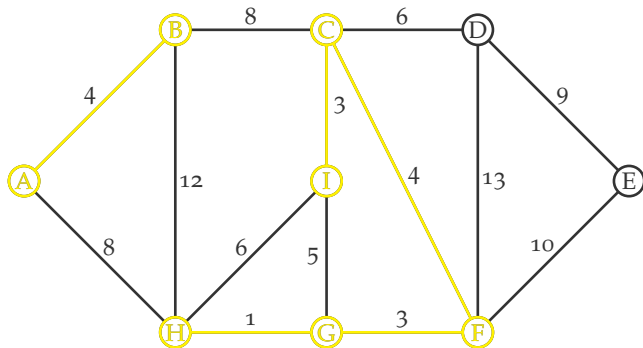


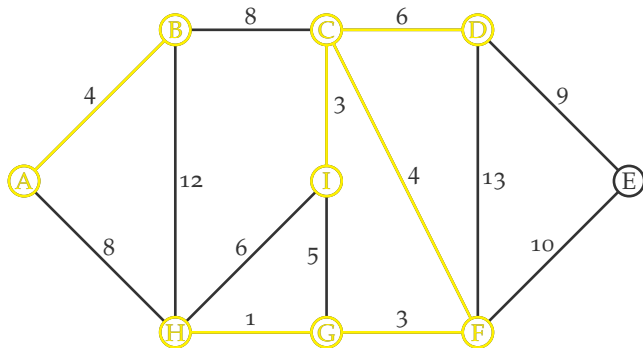


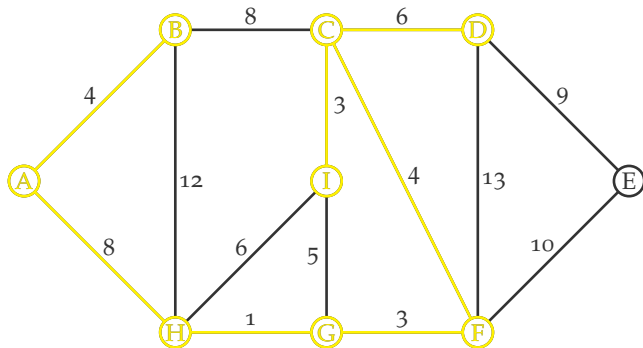


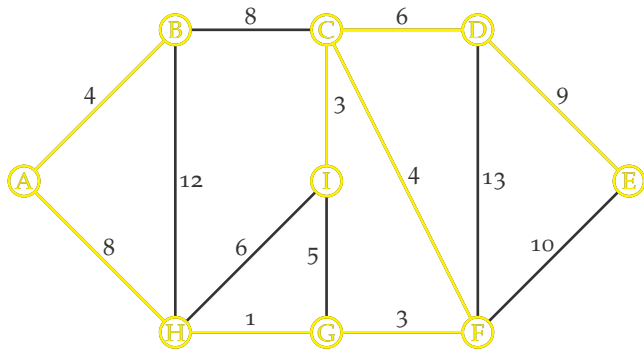














Bernard Galler (1928–2006)



Michael Fischer (1942)

- Podemos mejorar la complejidad computacional del Algoritmo de Kruskal por medio de la estructura de datos **union-find** (Galler y Fischer, 1964).

- Mantenemos un arreglo especificando en qué componente conexa está cada vértice.

- Mantenemos un arreglo especificando en qué componente conexa está cada vértice.
 1. El arreglo contiene el **padre** de cada vértice. Cada vértice está en la misma componente conexa que su padre.

- Mantenemos un arreglo especificando en qué componente conexa está cada vértice.
 1. El arreglo contiene el **padre** de cada vértice. Cada vértice está en la misma componente conexa que su padre.
 2. Se forma una estructura de **árbol**, cuya raíz es el **representante** de la componente conexa.

- Mantenemos un arreglo especificando en qué componente conexa está cada vértice.
 1. El arreglo contiene el **padre** de cada vértice. Cada vértice está en la misma componente conexa que su padre.
 2. Se forma una estructura de **árbol**, cuya raíz es el **representante** de la componente conexa.
- Inicialmente, cada vértice está en su propia componente conexa:

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

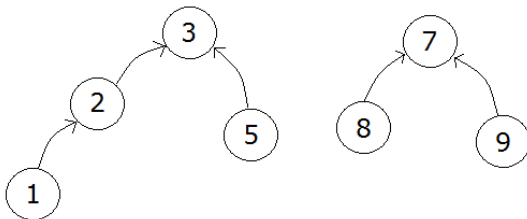
- Después de algunas iteraciones, tendremos varias componentes conexas:

0	2	3	3	4	3	6	7	7	7
---	---	---	---	---	---	---	---	---	---

- Después de algunas iteraciones, tendremos varias componentes conexas:

0	2	3	3	4	3	6	7	7	7
---	---	---	---	---	---	---	---	---	---

- Representamos esta situación con un árbol por cada componente conexas:



- Dado un vértice, es fácil encontrar su raíz:

```
int root(int i)
{
    while (A[i] != i)
        i = A[i];

    return i;
}
```

- Dado un vértice, es fácil encontrar su raíz:

```
int root(int i)
{
    while(A[i] != i)
        i = A[i];

    return i;
}
```

- Para determinar si dos vértices están en la misma componente conexa:

```
boolean find(int i, int j)
{
    return raiz(i) == raiz(j);
}
```

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.
- Hacemos que la raíz de una de ellas apunte a la raíz de la otra:

```
void union(int i, int j)
{
    int ri = root(i);
    int rj = root(j);

    A[ri] = rj;
}
```

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.
- Hacemos que la raíz de una de ellas apunte a la raíz de la otra:

```
void union(int i, int j)
{
    int ri = root(i);
    int rj = root(j);

    A[ri] = rj;
}
```

- ¿Qué complejidad computacional tienen estas operaciones?

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.
- Hacemos que la raíz de una de ellas apunte a la raíz de la otra:

```
void union(int i, int j)
{
    int ri = root(i);
    int rj = root(j);

    A[ri] = rj;
}
```

- ¿Qué complejidad computacional tienen estas operaciones?
 1. Union:

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.
- Hacemos que la raíz de una de ellas apunte a la raíz de la otra:

```
void union(int i, int j)
{
    int ri = root(i);
    int rj = root(j);

    A[ri] = rj;
}
```

- ¿Qué complejidad computacional tienen estas operaciones?
 1. Union: $O(n)$

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.
- Hacemos que la raíz de una de ellas apunte a la raíz de la otra:

```
void union(int i, int j)
{
    int ri = root(i);
    int rj = root(j);

    A[ri] = rj;
}
```

- ¿Qué complejidad computacional tienen estas operaciones?
 1. Union: $O(n)$
 2. Find:

- La única operación que modifica la estructura es la **unión** de dos componentes conexas.
- Hacemos que la raíz de una de ellas apunte a la raíz de la otra:

```
void union(int i, int j)
{
    int ri = root(i);
    int rj = root(j);

    A[ri] = rj;
}
```

- ¿Qué complejidad computacional tienen estas operaciones?
 1. Union: $O(n)$
 2. Find: $O(n)$

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor.

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union:

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union: $O(\log n)$
 2. Find:

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union: $O(\log n)$
 2. Find: $O(\log n)$

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union: $O(\log n)$
 2. Find: $O(\log n)$
- Podemos **compactar** los árboles cuando se hace find(), con las siguientes operaciones:

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union: $O(\log n)$
 2. Find: $O(\log n)$
- Podemos **compactar** los árboles cuando se hace find(), con las siguientes operaciones:
 1. **Path compression:** Hacer que todos los nodos apunten a la raíz.

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union: $O(\log n)$
 2. Find: $O(\log n)$
- Podemos **compactar** los árboles cuando se hace find(), con las siguientes operaciones:
 1. **Path compression:** Hacer que todos los nodos apunten a la raíz.
 2. **Path splitting:** Hacer que cada nodo apunte a su abuelo.

- Si llevamos la cuenta de la cantidad de elementos en cada componente conexa, podemos hacer que la raíz del menor árbol apunte a la raíz del mayor. Con esto, logramos:
 1. Union: $O(\log n)$
 2. Find: $O(\log n)$
- Podemos **compactar** los árboles cuando se hace `find()`, con las siguientes operaciones:
 1. **Path compression:** Hacer que todos los nodos apunten a la raíz.
 2. **Path splitting:** Hacer que cada nodo apunte a su abuelo.
 3. **Path halving:** Aplicar *path splitting* nodo de por medio.

Definición: Función de Ackermann

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

Definición: Función de Ackermann

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

- La función de Ackermann crece muy rápidamente! Por ejemplo, $A(4, 2)$ tiene 19.729 dígitos decimales.

Definición: Función de Ackermann

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

- La función de Ackermann crece muy rápidamente! Por ejemplo, $A(4, 2)$ tiene 19.729 dígitos decimales.
- Si se usa alguna de las técnicas de compactación y la unión por tamaño (o por altura), entonces el tiempo amortizado de cada operación es $O(\alpha(n))$, donde $\alpha(n)$ es la inversa de $A(n, n)$.

Definición: Función de Ackermann

$$A(m, n) = \begin{cases} n + 1 & \text{si } m = 0 \\ A(m - 1, 1) & \text{si } m > 0 \text{ y } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{si } m > 0 \text{ y } n > 0 \end{cases}$$

- La función de Ackermann crece muy rápidamente! Por ejemplo, $A(4, 2)$ tiene 19.729 dígitos decimales.
- Si se usa alguna de las técnicas de compactación y la unión por tamaño (o por altura), entonces el tiempo amortizado de cada operación es $O(\alpha(n))$, donde $\alpha(n)$ es la inversa de $A(n, n)$.
- La función $\alpha(n)$ crece muy lentamente! A fines prácticos, $\alpha(n) \leq 5$ para cualquier valor de n que utilicemos.