MST y algoritmo de Prim

Clase 22

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

Conectividad digital

Conectividad digital

Consideremos el problema de mejorar la conectividad digital de la Región del Maule

- Objetivo: instalar fibra óptica subterránea entre pares de puntos relevantes
- Cada instalación de ese cableado tiene un costo
- Es prioritario conectar ciudades más pobladas

El desafío es **cubrir** con el menor costo

Los MST resuelven este problema

Árboles de cobertura mínimos

Definición

Dado un grafo no dirigido G, un subgrafo $T \subseteq G$ se dice un **árbol de cobertura mínimo** o **MST** de G si

1. T es un árbol
2. $V(T) = V(G)$
3. No existe otro MST T' para G con menor costo total

La propiedad 1. se deduce de las otras dos

Árboles de cobertura mínimos

Dado G no dirigido

- Llamamos **corte** a una **partición** (V_1, V_2) de $V(G)$

$$V_1, V_2 \neq \varnothing, \qquad V_1 \cup V_2 = V(G), \qquad V_1 \cap V_2 = \varnothing$$

- Diremos que una arista **cruza el corte** si uno de sus extremos está en V_1 y el otro en V_2

Con esto planteamos el primer algoritmo para nuestro problema

Algoritmo de Prim

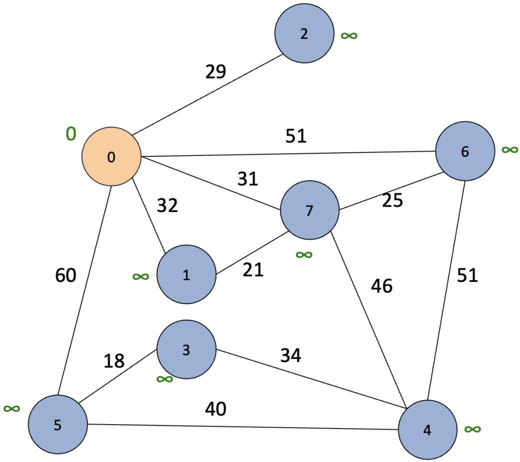
La idea detrás del **algoritmo de Prim** es utilizar las aristas que cruzan cortes para guiar la construcción

Para un grafo $G = (V, E)$ y un nodo inicial v

1. Sean $R = \{v\}$ y $\bar{R} = V - R$
2. Sea e la arista de menor costo que cruza de R a $\bar{R}$
3. Sea u el nodo de e que pertenece a $\bar{R}$
4. Agregar e al MST. Eliminar u de $\bar{R}$ y agregarlo a R
5. Si quedan elementos en $\bar{R}$, volver al paso 2.

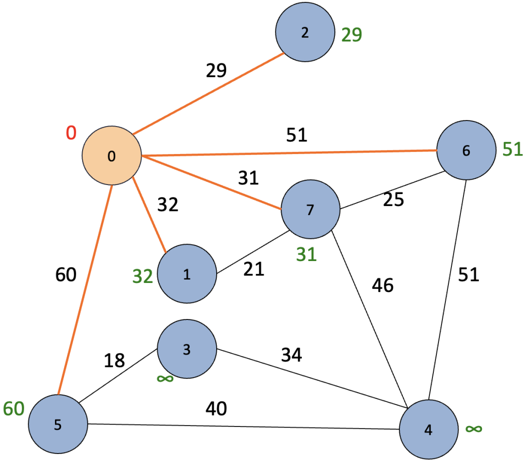
Usamos una cola de prioridad que usa los costos para conectarse a R

Algoritmo de Prim



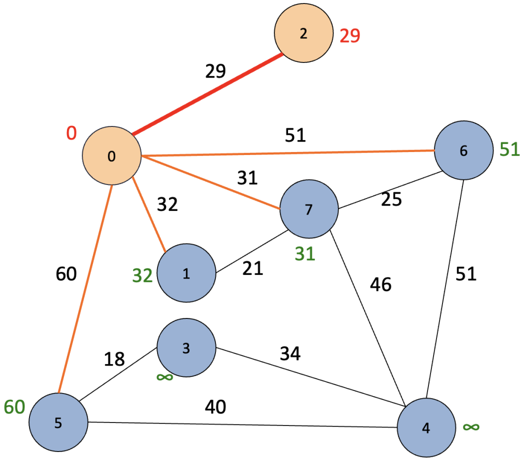
Usamos cortes para decidir qué arista agregar

Algoritmo de Prim



Usamos cortes para decidir qué arista agregar

Algoritmo de Prim



Usamos cortes para decidir qué arista agregar

Algoritmo de Prim: una versión más concreta

$\texttt{Prim}(s)$:

$Q \leftarrow$ cola de prioridades con s

$T \leftarrow$ lista vacía

$x.key \leftarrow 0; \quad x.parent \leftarrow \varnothing$

while Q *no está vacía* **:**

    $u \leftarrow \texttt{Extract}(Q); \quad u.color \leftarrow$ negro

    if $u.parent \neq \varnothing$ **:**

        $T \leftarrow T \cup \{(u.parent, u)\}$

    for $v \in \alpha[u] \wedge v.color \neq negro$ **:**

        if $v \in Q$ **:**

            $Insert(Q, v)$

        if $v.key > cost(u, v)$ **:**

            $v.key \leftarrow cost(u, v)$

            $v.parent \leftarrow u$

return T

Suponemos que inicialmente $v.key \leftarrow \infty$ para todo v

Una alternativa

Pensemos en otro principio para formar un MST

- Cada arista tiene un costo
- Existe una arista con costo mínimo (o varias con el mismo costo)

¿Dicha arista pertenece a un MST?
¿Podemos aprovecharlo en un algoritmo codicioso?

Sumario

Algoritmo de Kruskal

La idea detrás del **algoritmo de Kruskal** es crear un bosque que va convergiendo en un único árbol

Para un grafo $G = (V, E)$, iteramos sobre las aristas e en orden no decreciente de costo

1. Si e genera un ciclo al agregarla a T, la ignoramos
2. Si no genera ciclo, se agrega

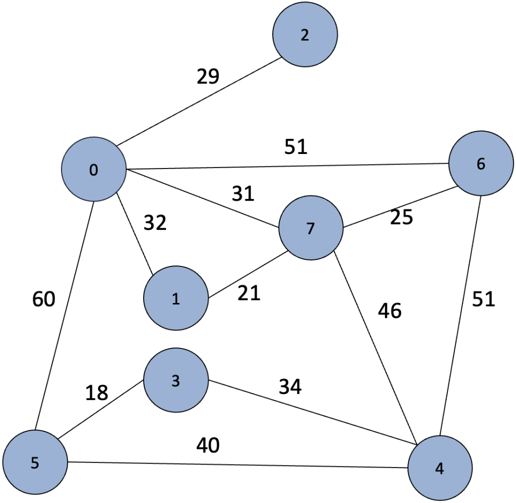
¿Es necesario revisar **todas** las artistas de E?

Algoritmo de Kruskal

```
Kruskal(G):
    E ← E ordenada por costo, de menor a mayor
    T ← lista vacía
    for e ∈ E :
        if Agregar e a T no forma ciclo :
            T ← T ∪ {e}
    return T
```

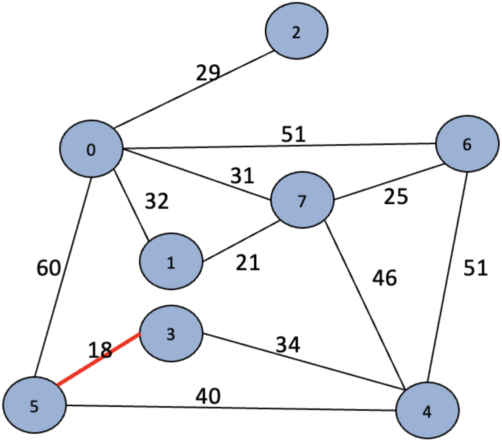
¿Este algoritmo usa cortes de manera implícita?

Algoritmo de Kruskal



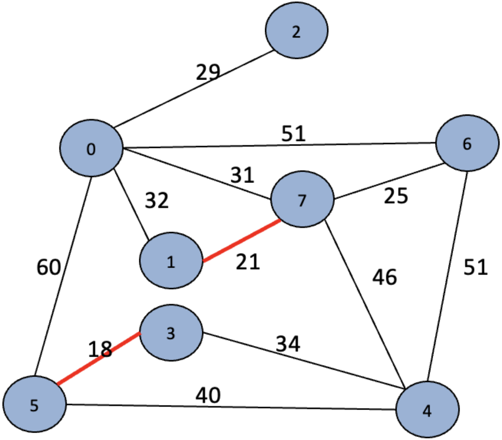
Al inicio, tenemos un bosque de $|V|$ árboles sin aristas

Algoritmo de Kruskal

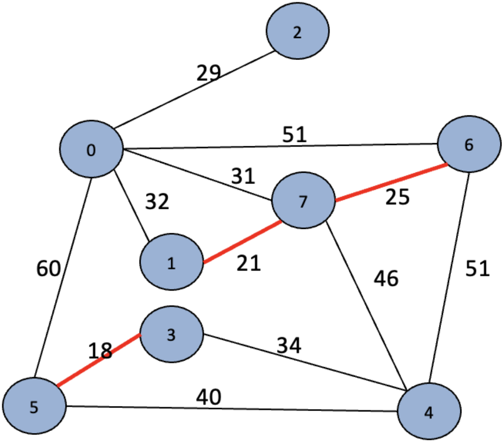


Reducimos la cantidad de árboles en 1 con cada arista añadida

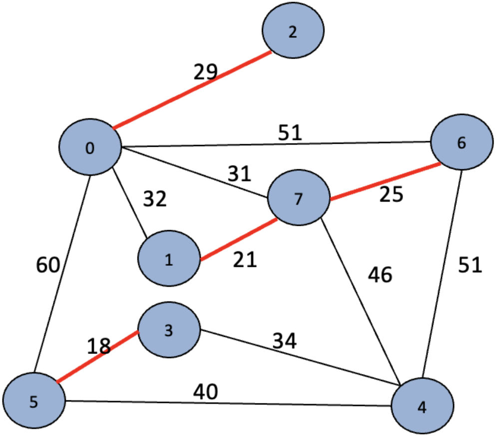
Algoritmo de Kruskal



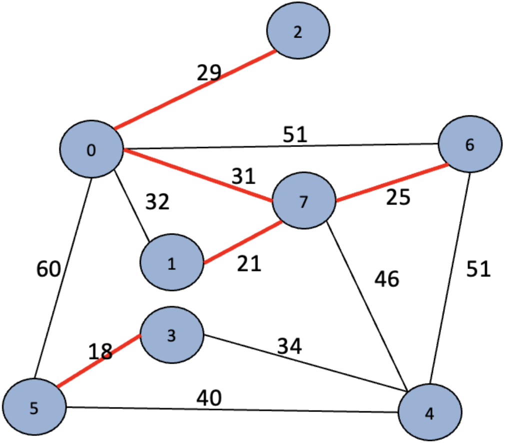
Algoritmo de Kruskal



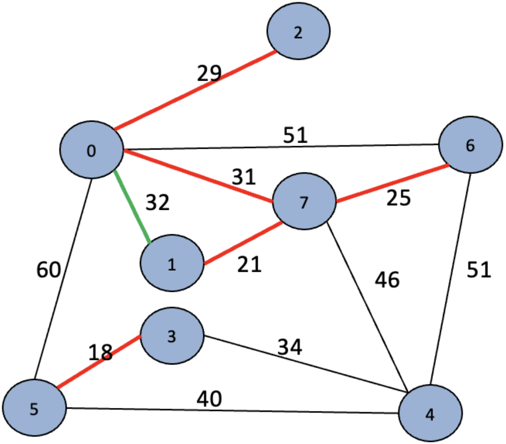
Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal



Algoritmo de Kruskal: una conexión con cortes

Dada una arista (u, v) que corresponde chequear

- Podemos considerar un corte dado por

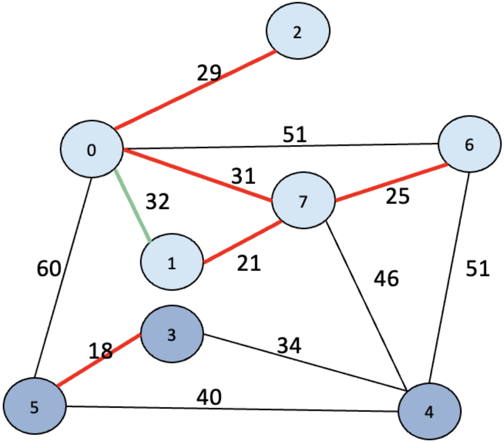
$$V_1 = \{w \mid w \text{ está conectado con u con aristas de } T\}, \quad V_2 = V - V_1$$

- Y otro corte dado por

$$V_1 = \{w \mid w \text{ está conectado con v con aristas de } T\}, \quad V_2 = V - V_1$$

Agregamos la arista si es de corte para estos dos cortes

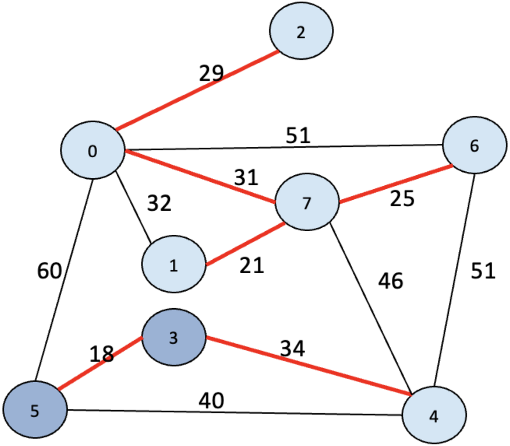
Algoritmo de Kruskal



La arista $(0, 1)$ no es de corte para ninguno de los cortes (ambos cortes son iguales porque 0 y 1 ya están conectados entre sí):

$$(\{0, 1, 2, 6, 7\}, \{3, 4, 5\})$$

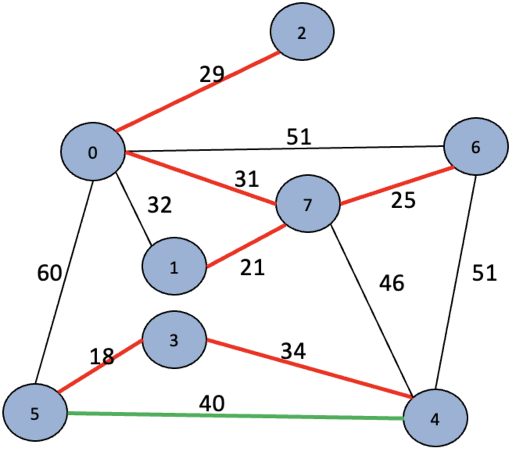
Algoritmo de Kruskal



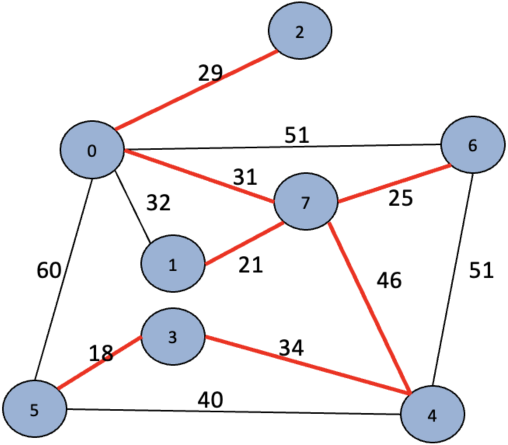
La arista $(3, 4)$ es de corte para los cortes:

$$(\{3, 5\}, \{0, 1, 2, 4, 6, 7\}) \qquad (\{4\}, \{0, 1, 2, 3, 5, 6, 7\})$$

Algoritmo de Kruskal

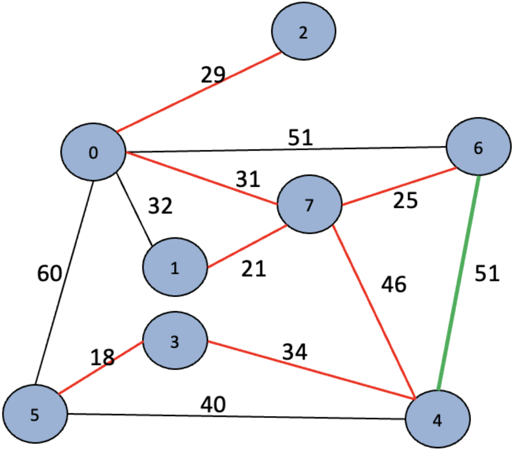


Algoritmo de Kruskal



En este punto, $|E| = |V| - 1$ y T es un árbol

Algoritmo de Kruskal



Cualquier arista adicional formaría un **ciclo**

Algoritmo de Kruskal

Kruskal(G):
    $E \leftarrow E$ ordenada por costo, de menor a mayor
    $T \leftarrow$ lista vacía
    for $e \in E$ **:**
        if *Agregar e a T no forma ciclo* **:**
            $T \leftarrow T \cup \{e\}$
    return T

¿Cómo revisamos eficientemente que e no forma un ciclo en T?

Algoritmo de Kruskal: conjuntos

Dada una arista (u, v) podemos considerar los conjuntos

- Nodos conectados con u en T

$$V_u = \{w \mid w \text{ está conectado con u con aristas de } T\}$$

- Nodos conectados con v en T

$$V_v = \{w \mid w \text{ está conectado con v con aristas de } T\}$$

La arista forma un **ciclo** en T si, y solo si, $V_u = V_v$

Notemos que los árboles del bosque T forman **conjuntos disjuntos**

¿Cómo modelar eficientemente estas estructuras?

Sumario

Conjuntos disjuntos

Definición

Una colección de **conjuntos disjuntos** $\{S_1, \ldots, S_n\}$ es una EDD que permite

- Identificar a qué conjunto **pertenece** un elemento
- **Unir** conjuntos S_i, S_j formando un nuevo conjunto

Para atacar la representación de los conjuntos usaremos un **representante**

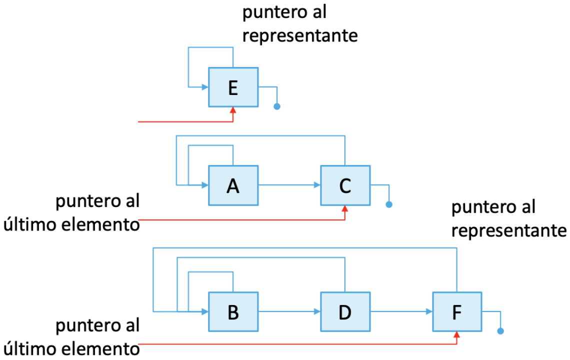
- Un elemento cualquiera de S que denotamos por $rep(S)$
- La consulta de representante debe ser consistente si no hay cambios en S: entrega el mismo elemento
- Cada elemento de S tiene referencia a $rep(S)$

Dos conjuntos S y T son iguales si y solo si $rep(S) = rep(T)$

Conjunto disjuntos: listas ligadas

Una primera representación puede ser con listas ligadas

Para los conjuntos $\{E\}, \{A, C\}, \{B, D, F\}$



¿Cuál es la complejidad de las operaciones de búsqueda y unión?

Setting

Supondremos que las operaciones son de la forma

- `Find`(A): entrega el conjunto al que pertenece A
- `Union`(A, B): entrega un conjunto resultante de unir los conjuntos de A y B

Además, supondremos que en un estado inicial, tenemos n conjuntos con **un solo elemento cada uno**. Un conjunto con un elemento se llama **singleton**

Definimos los parámetros

- n: número de elementos (cantidad de conjuntos iniciales)
- m: número de operaciones `Union` y `Find` realizadas en una rutina

Intuición detrás de las operaciones

Consideremos una matriz de $c \times c$ tal que nos sirve de base para construir un laberinto

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Cada elemento está en un **conjunto independiente**, i.e. $n = c^2$

¿Qué complejidad tiene m al conectar todos los conjuntos?

Intuición detrás de las operaciones

Nos interesa conectar a todos los elementos: botaremos muros del laberinto

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Conjuntos actuales:

$$\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}, \{8\}, \ldots$$

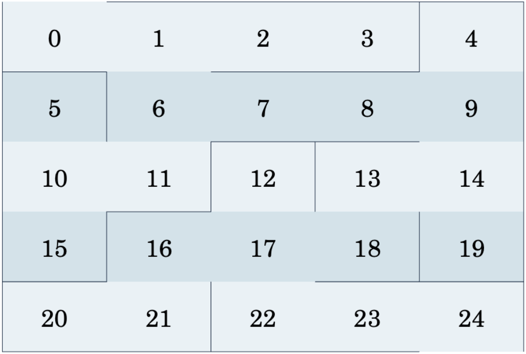
Intuición detrás de las operaciones

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

Conjuntos actuales:

$$\{0,1\},\{2\},\{3\},\{4,6,7,8,9,13,14\},\{5\},\{10,11,15\},\ldots$$

Intuición detrás de las operaciones



Conjuntos actuales:

$$\{0, 1, 2, \ldots, 23, 24\}$$

Intuición detrás de las operaciones

Como cada conjunto es disjunto

- Luego de una operación `Union`, el número de conjuntos se reduce en 1
- La cantidad de operaciones `Union` para lograr un solo conjunto es $n - 1$

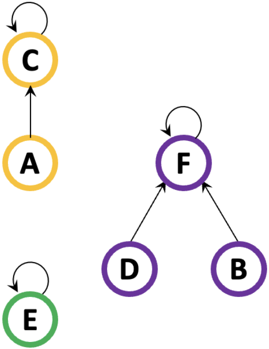
Además, para saber qué muros botar debemos usar `Find`

- Entre dos elementos separados por un muro: si `Find` da distinto, se bota ese muro
- Cantidad de `Find` es $\mathcal{O}(n)$

¿Cómo representamos los conjuntos?

Conjuntos como árboles

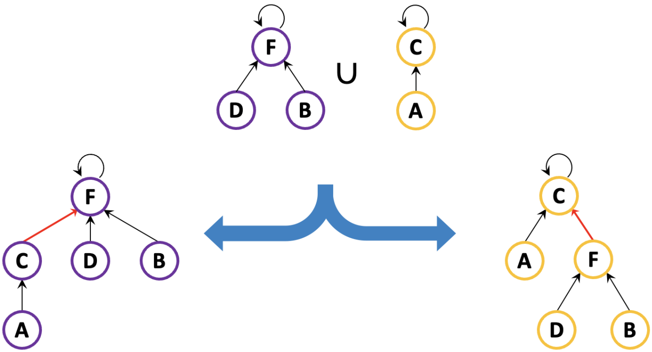
Una primera forma de representar los conjuntos es usar **árboles** donde los caminos llevan al **representante**



¿Cómo se implementan las operaciones `Union` y `Find`?

Conjuntos como árboles

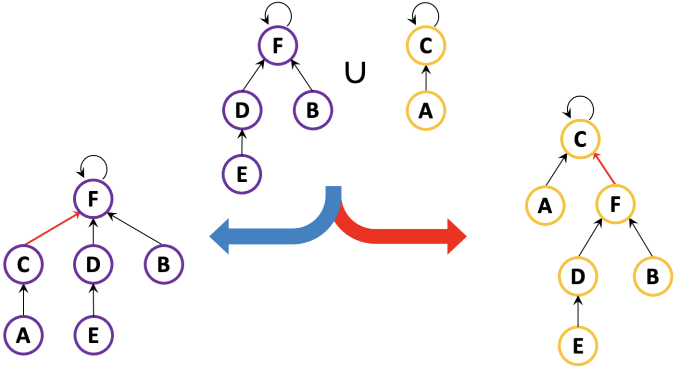
La operación `Union` corresponde a cambiar el autoloop del representante de un conjunto: $\mathcal{O}(1)$



¿Cuál de las dos opciones escogemos como resultado?

Conjuntos como árboles

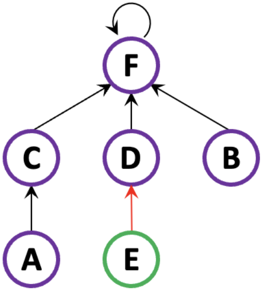
La operación `Find` requiere recorrer punteros, por lo que las dos opciones no son equivalentes



Hay que unir el árbol más corto al más largo

Conjuntos como árboles

La operación `Find` requiere recorrer punteros

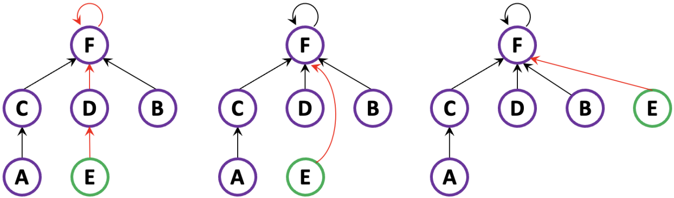


$$\texttt{Find}(E) = \texttt{Find}(D) = \texttt{Find}(F) = F$$

¿Podríamos mejorar estructuralmente?

Conjuntos como árboles

Podemos modificar punteros para simplificar las rutas



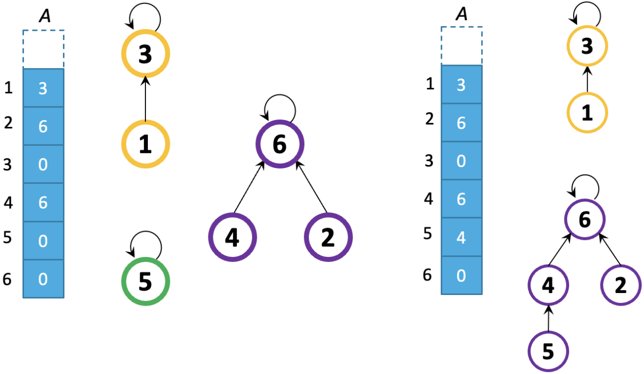
$$\texttt{Find}(E) == F$$

¿Cómo almacenamos los conjuntos?

Conjuntos como árboles

Podemos almacenar los árboles en un único arreglo de referencias:

$$A[k] \text{ es el padre de } k$$



Complejidad de `Find`

El costo de `Find` depende de qué elemento se busca

- Se puede demostrar que para un conjunto de n elementos, con rutas simplificadas, `Find` toma tiempo $\mathcal{O}(\alpha(n))$
- $\alpha(n)$ es una función *interesante* que tiene buenas propiedades
- En particular, crece **extremadamente lento**

$$\alpha(n) = 4, \quad 2048 \leq n \leq 16^{512}$$

En cualquier aplicación práctica, podemos considerar que $\alpha(n) \in \mathcal{O}(1)$

Algoritmo de Kruskal y conjuntos

Ahora que tenemos una implementación de conjuntos

- En Kruskal comenzamos con un conjunto para cada nodo
- Verificamos aristas viendo si sus extremos están **en el mismo conjunto**
- Si no lo están, las agregamos y **unimos** los conjuntos

Algoritmo de Kruskal y conjuntos

```
Kruskal(G):
    E ← E ordenada por costo, de menor a mayor
    for v ∈ V :
        MakeSet(v)
    T ← lista vacía
    for (u, v) ∈ E :
        if Find(u) ≠ Find(v) :
            T ← T ∪ {(u, v)}
            Union(u, v)
    return T
```

¿Qué complejidad tiene este algoritmo?

Complejidad

Usando la implementación de conjuntos disjuntos

- Ordenar las aristas $\mathcal{O}(E \log(E))$
- Construir V conjuntos singleton $\mathcal{O}(V)$
- Unir conjuntos $V - 1$ veces $\mathcal{O}(V)$
- Buscar $2E$ veces $\mathcal{O}(E\alpha(V)) = \mathcal{O}(E)$

En total

$$\mathcal{O}(E \log(E) + V + E) = \mathcal{O}(E \log(E))$$

y como $E \leq V^2$, tenemos que $\log(E) \in \mathcal{O}(\log(V))$

Kruskal tiene complejidad $\mathcal{O}(E \log(V))$, igual que Prim