

# MST y algoritmo de Prim

Clase 22

IIC 2133 - Sección 3

Prof. Eduardo Bustos

# Sumario

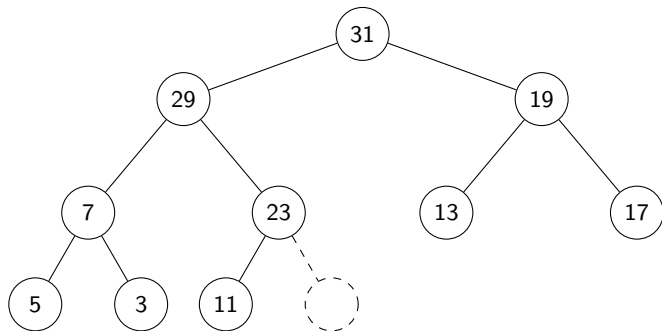
**Introducción**

Algoritmo de Prim

Análisis del algoritmo

Cierre

# Heaps binarios



# Construcción de un heap

La inserción que vimos permite agregar un **único** elemento a un heap  $H$  preexistente

Si tenemos un arreglo  $A$  y queremos obtener un heap podemos usar una de dos estrategias

1. Iterar para cada elemento de  $A$ , insertando sobre un heap originalmente vacío
2. Utilizar `SiftDown` para ciertos elementos de  $A$

Esta última forma es *in place* y sencilla

# Construcción de un heap

**input** : arreglo  $A[0 \dots n-1]$

BuildHeap( $A$ ):

**for**  $i = \lfloor n/2 \rfloor - 1 \dots 0$  :       $\triangleright$  loop decreciente

        SiftDown( $A, i$ )

**Observación:** los elementos de  $A$  en los cuales no se llama directamente SiftDown son hojas del último nivel del árbol

Respecto a su complejidad

- La complejidad asintótica *directa* es  $\mathcal{O}(n \log(n))$
- Se puede demostrar que una mejor cota es  $\mathcal{O}(n)$

BuildHeap deja  $A$  como un heap en tiempo  $\mathcal{O}(n)$

# Heaps para ordenar

Ya sabemos crear un heap a partir de un arreglo cualquiera

Y sabemos la propiedad de heap: cada nodo es estrictamente mayor que sus descendientes

¿Podemos aprovechar estos hechos para ordenar un arreglo  $A$ ?

# Ordenando con heaps

Dado un heap  $H$

- Su raíz es estrictamente mayor a todos los otros nodos
- Debe ser el último elemento del arreglo ordenado

Si sabemos que el último elemento del arreglo luego del intercambio **está ordenado**

- No queremos moverlo más
- Es decir, reducimos el **tamaño del heap**
- A este parámetro le llamamos `A.heap_size`

Cambiamos el tamaño del heap para que `SiftDown` sepa hasta dónde llegar moviendo elementos

# Ordenando con heaps

**input** : arreglo  $A[0 \dots n-1]$

HeapSort( $A$ ):

    BuildHeap( $A$ )

**for**  $i = n-1 \dots 1$  :      ▷ loop decreciente

$A[0] \rightleftharpoons A[i]$

$A.\text{heap\_size} = A.\text{heap\_size} - 1$

        ShiftDown( $A, 0$ )

Respecto a su complejidad

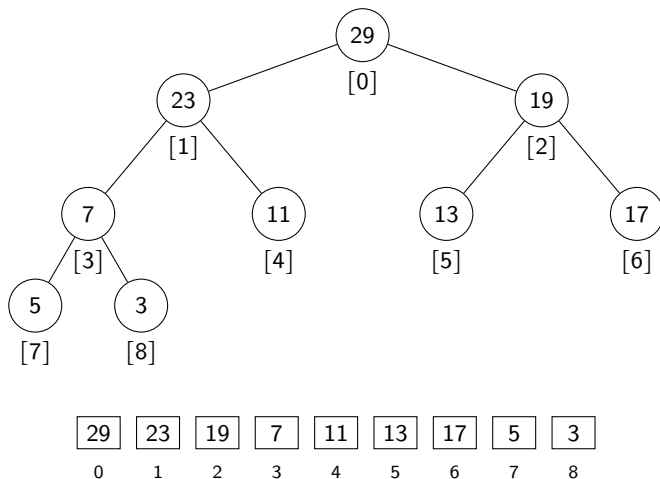
- BuildHeap  $\mathcal{O}(n)$
- SiftDown se repite  $\mathcal{O}(n)$  veces  $\mathcal{O}(n \log(n))$
- Total  $\mathcal{O}(n + n \log(n)) = \mathcal{O}(n \log(n))$

HeapSort ordena en tiempo  $\mathcal{O}(n \log(n))$



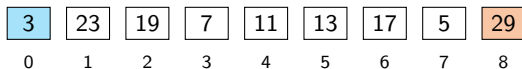
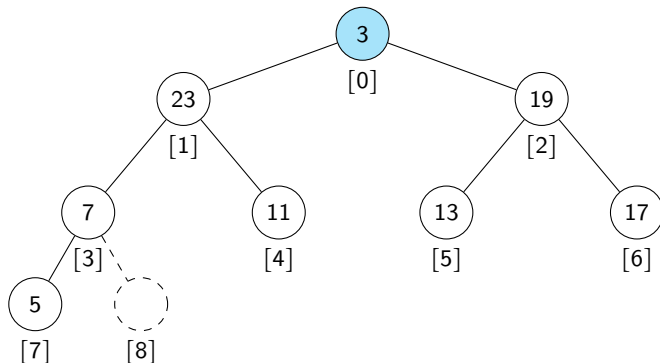
# Heapsort en acción

Supongamos que ya contamos con el heap resultante de BuildHeap



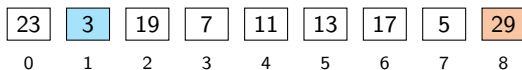
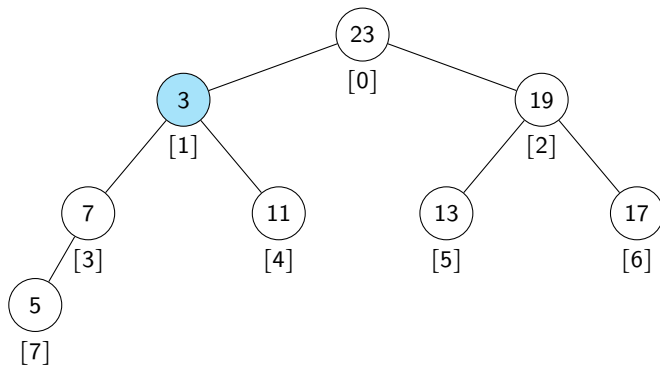
# Heapsort en acción

Movemos el primer elemento y reducimos el tamaño del heap en 1



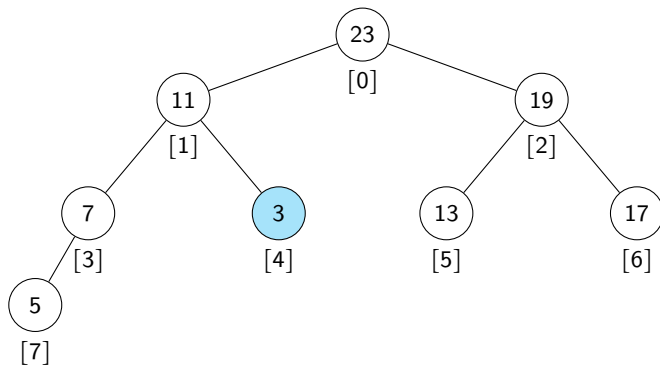
# Heapsort en acción

Aplicamos  $\text{SiftDown}(A, 0)$  (el heap es  $A[0 \dots 7]$ )



# Heapsort en acción

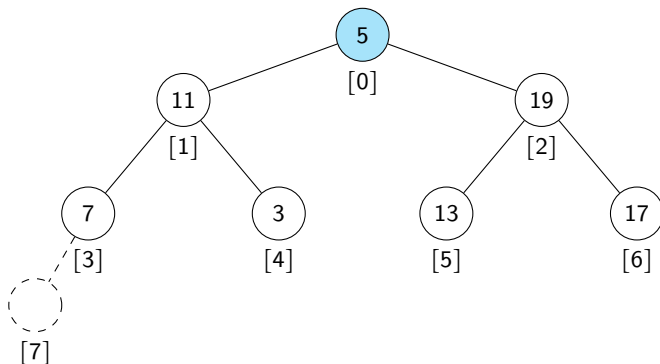
Aplicamos  $\text{SiftDown}(A, 1)$  (el heap es  $A[0 \dots 7]$ )



23	11	19	7	3	13	17	5	29
0	1	2	3	4	5	6	7	8

# Heapsort en acción

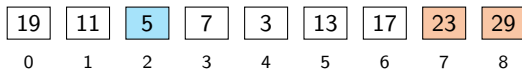
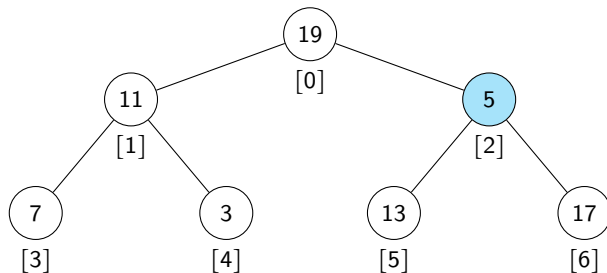
Repetimos el proceso con la nueva raíz



5	11	19	7	3	13	17	23	29
0	1	2	3	4	5	6	7	8

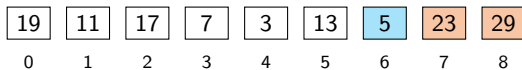
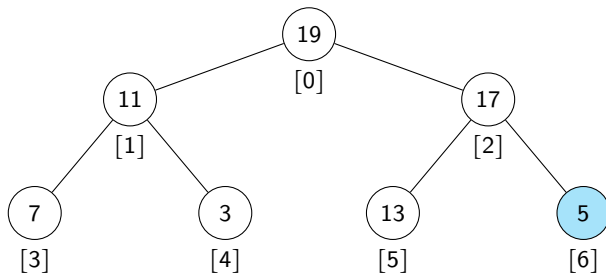
# Heapsort en acción

Aplicamos  $\text{SiftDown}(A, 0)$  (el heap es  $A[0 \dots 6]$ )



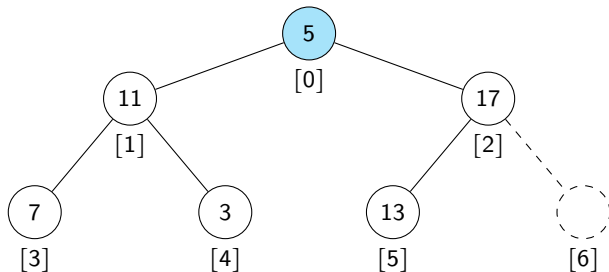
# Heapsort en acción

Aplicamos  $\text{SiftDown}(A, 2)$  (el heap es  $A[0 \dots 6]$ )



# Heapsort en acción

Repetimos el proceso con la nueva raíz

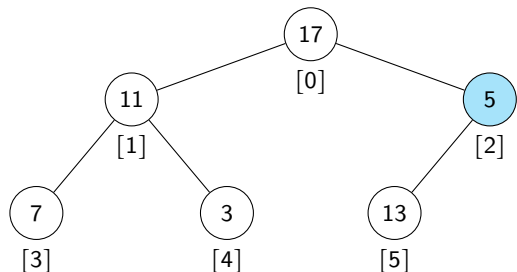


5	11	17	7	3	13	19	23	29
0	1	2	3	4	5	6	7	8



# Heapsort en acción

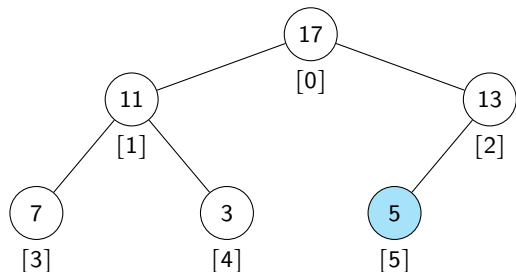
Aplicamos  $\text{SiftDown}(A, 0)$  (el heap es  $A[0 \dots 5]$ )



17	11	5	7	3	13	19	23	29
0	1	2	3	4	5	6	7	8

# Heapsort en acción

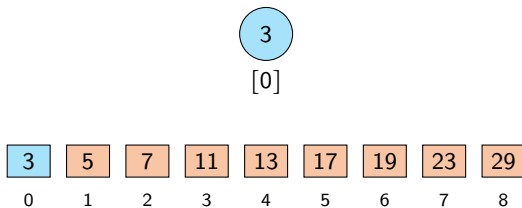
Aplicamos  $\text{SiftDown}(A, 2)$  (el heap es  $A[0 \dots 5]$ )



17	11	13	7	3	5	19	23	29
0	1	2	3	4	5	6	7	8

# Heapsort en acción

El proceso termina cuando queda solo un nodo en el heap: es el mínimo



# Uso de la cola de prioridad

Ya vimos que la cola de prioridad implementada como heap permite

- Insertar valores con prioridad
- Extraer el más prioritario

En algunos contextos además necesitamos **actualizar** prioridad cuando cambie el costo óptimo

1. Cambiamos prioridad del nodo del heap
2. Hacemos intercambios hacia arriba si es necesario

La propiedad de heap permite que esta operación solo afecte nodos en la ruta del nodo a la raíz

# Uso de la cola de prioridad

**input** : heap representado como arreglo  $H[0 \dots n-1]$ ,  
índice  $i$ ,  
nueva prioridad  $k > H[i]$

IncreaseKey( $H, i, k$ ):

$H[i] \leftarrow k$

**while**  $i > 0 \wedge H[\text{Parent}(i)] < H[i]$  :

$H[i] \rightleftharpoons H[\text{Parent}(i)]$

$i \leftarrow \text{Parent}(i)$

En tiempo  $\mathcal{O}(\log(V))$  actualizamos la prioridad y mantenemos el heap...  
¡Esta operación será necesaria en nuestro algoritmo de hoy!

# 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

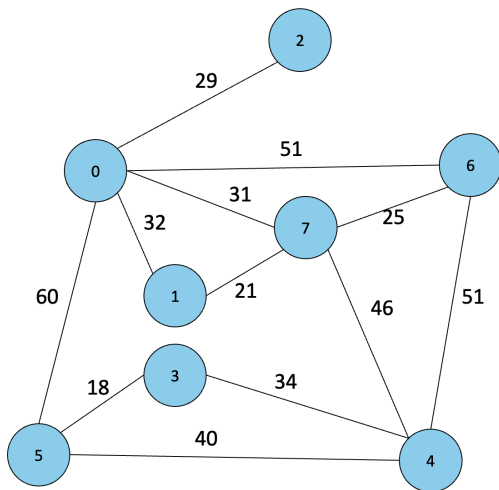
El desafío es **cubrir** con el menor costo

¿Ya tenemos una estrategia para atacar este problema?

## Conectividad digital



# Conectividad digital



Usamos un grafo **no dirigido con costos**



# Conectividad digital

Usamos un grafo no dirigido con costos

- Cada nodo es un punto a conectar (ciudad, pueblo, etc)
- Cada arista tiene un costo de conexión
- Consideramos que cada nodo presente en el grafo, debe ser conectado

Para resolver el problema seleccionamos un subconjunto de aristas

- La suma de los costos debe ser **mínima**
- El **subgrafo** que solo considera esas aristas, debe ser **conexo**

Buscamos un MST

# Objetivos de la clase

- ☐ Modificar prioridad dentro de un heap
- ☐ Comprender una segunda forma de resolver el problema de MST
- ☐ Demostrar correctitud y complejidad del algoritmo de Prim

# Sumario

Introducción

**Algoritmo de Prim**

Análisis del algoritmo

Cierre

# Recordemos: cortes

Para atacar el problema algorítmicamente, dado  $G$  no dirigido

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

$$V_1, V_2 \neq \emptyset, \quad V_1 \cup V_2 = V(G), \quad V_1 \cap V_2 = \emptyset$$

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

¿Qué podemos afirmar respecto a los MST y las aristas que cruzan un corte dado?

## Recordemos: cortes

Si  $T$  es un MST de  $G$ , y  $P = (V_1, V_2)$  es un corte de  $G$

- Sabemos que al menos una arista que cruza  $P$  está incluida en  $T$
- De lo contrario, no habría camino entre nodos de  $V_1$  y  $V_2$
- Por lo tanto:  $T$  no sería de cobertura

Si  $P = (V_1, V_2)$  es un corte de  $G$

- Sabemos que cada arista de corte tiene un costo asociado
- La arista más barata **siempre** se incluye en **algún** MST

¿Cómo podemos usar estos hechos para construir un MST desde cero?

# Algoritmo de Prim

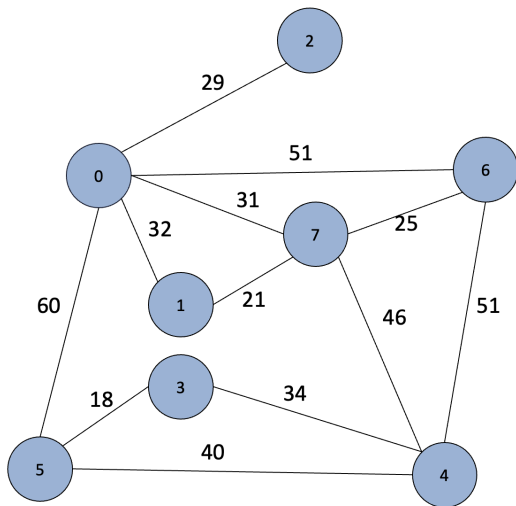
La idea detrás del **algoritmo de Prim** es utilizar las aristas que cruzan cortes para guiar la construcción

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

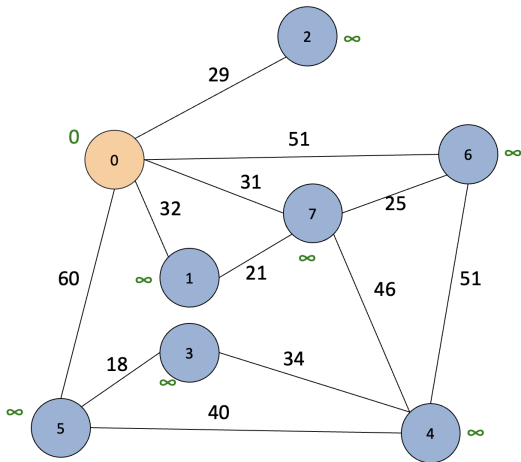
1. Sean  $R = \{v\}$  y  $\bar{R} = V - R$
2. Sea  $e$  la arista de menor costo que cruza de  $R$  a  $\bar{R}$
3. Sea  $u$  el nodo de  $e$  que pertenece a  $\bar{R}$
4. Agregar  $e$  al MST. Eliminar  $u$  de  $\bar{R}$  y agregarlo a  $R$
5. Si quedan elementos en  $\bar{R}$ , volver al paso 2.

¿Cómo hacer eficiente el paso 2?

## Árboles de cobertura mínimos

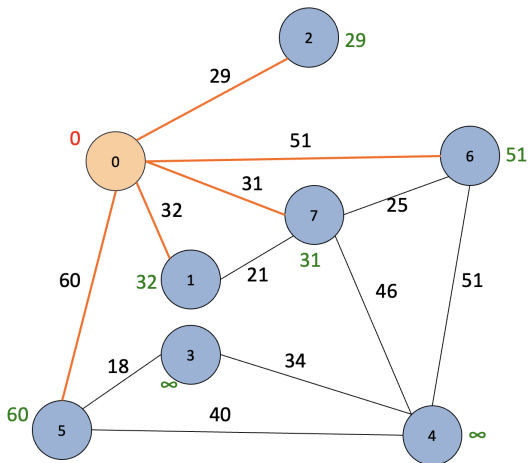


# Árboles de cobertura mínimos

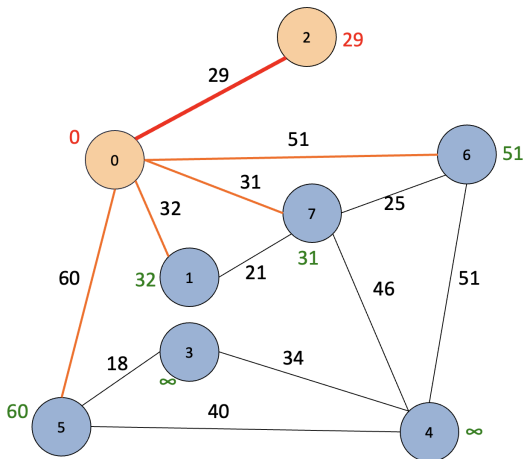




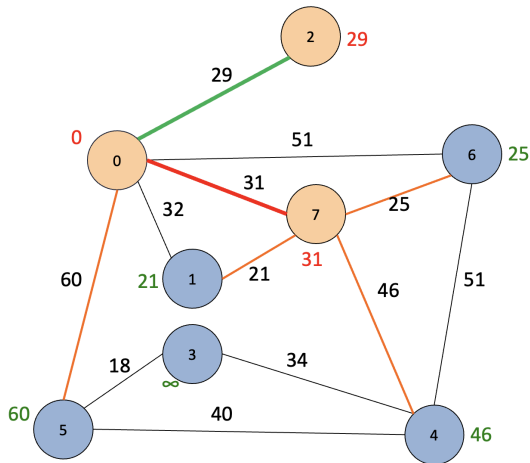
# Árboles de cobertura mínimos



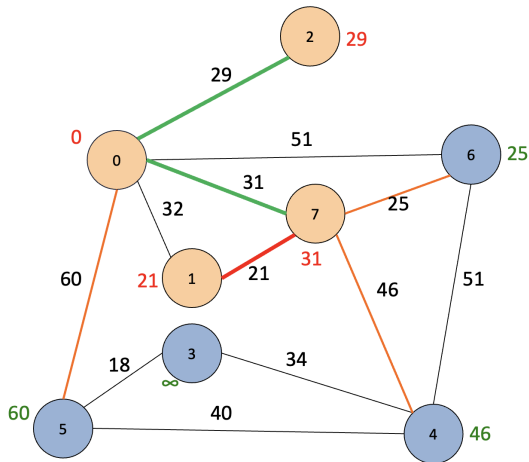
# Árboles de cobertura mínimos



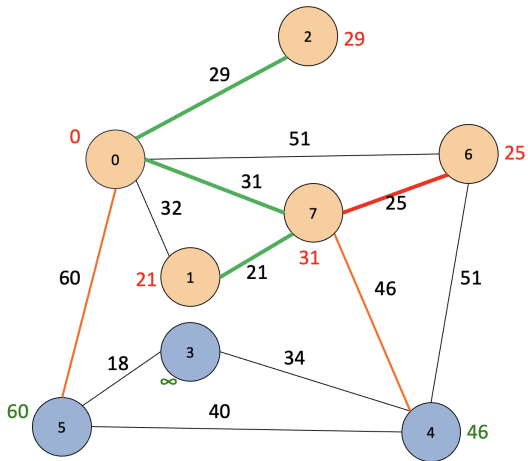
# Árboles de cobertura mínimos



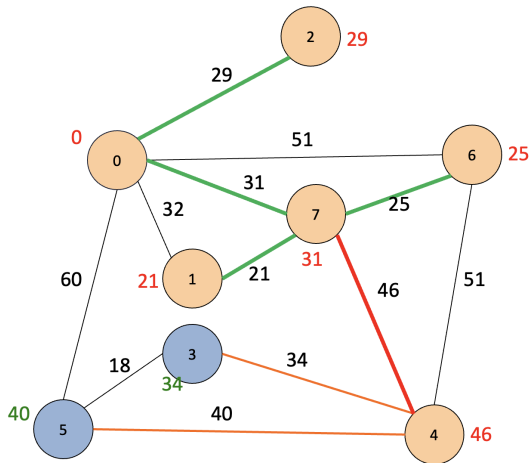
# Árboles de cobertura mínimos



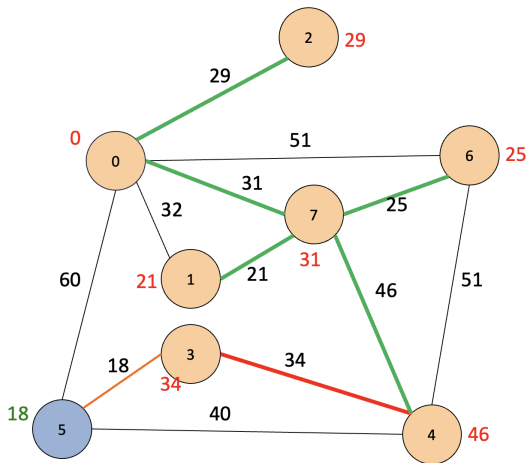
# Árboles de cobertura mínimos



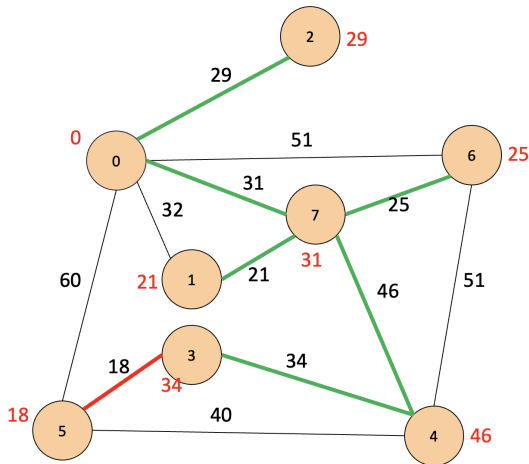
# Árboles de cobertura mínimos



# Árboles de cobertura mínimos

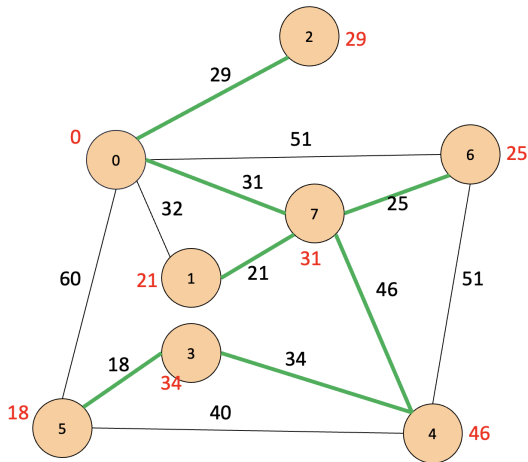


# Árboles de cobertura mínimos





# Árboles de cobertura mínimos



# Algoritmo de Prim

Prim( $s$ ):

```
1   $Q \leftarrow$  cola de prioridades vacía (Min Heap)
2   $T \leftarrow$  lista vacía
3  for  $u \in V - \{s\}$  :
4       $d[u] \leftarrow \infty$ ;  $\pi[u] \leftarrow \emptyset$ ; Insert( $Q, u$ )
5   $d[s] \leftarrow 0$ ;  $\pi[s] \leftarrow \emptyset$ ; Insert( $Q, s$ )
6  while  $Q$  no está vacía :
7       $u \leftarrow$  Extract( $Q$ )
8       $T \leftarrow T \cup \{(\pi[u], u)\}$ 
9      for  $v \in \alpha[u]$  :
10         if  $v \in Q$  :
11             if  $d[v] > cost(u, v)$  :
12                  $d[v] \leftarrow cost(u, v)$ ;  $\pi[v] \leftarrow u$ 
13                 DecreaseKey( $Q, v, d[v]$ )
14  return  $T$ 
```

$Q$  usa como prioridad el valor  $d[v]$ .

DecreaseKey( $Q, v, k$ ) cambia la prioridad del elemento  $v$

# Sumario

Introducción

Algoritmo de Prim

**Análisis del algoritmo**

Cierre

# Correctitud

## Demostración

**Correctitud.** Fijaremos nuestra atención en la línea 8 del algoritmo, justo luego de agregar a  $T$  una nueva arista. Denotaremos por  $G_n$  al subgrafo de  $G$  tal que considera solo los primeros  $n$  nodos extraídos de  $Q$  con todas sus aristas de  $G$ .

Probaremos por inducción sobre el número de iteraciones la propiedad

$P(n) :=$  En la iteración  $n$ -ésima, la línea 8 guarda en  $T$  un MST para el subgrafo  $G_n$

## Demostración

**C.B.** Para  $n = 1$ , extraemos de  $Q$  el nodo inicial y agregamos una arista *dummy*. Como el grafo que debemos cubrir es  $G_1 = (\{s\}, \emptyset)$  y  $T$  no cubre nada más que el nodo inicial, es efectivamente un MST para  $G_1$ .

**H.I.** Suponemos que en la iteración  $n$ ,  $T$  tiene guardado un MST para cubrir el subgrafo  $G_n$  con costo mínimo.

**T.I.** Sea  $u$  el  $(n + 1)$ -ésimo nodo extraído de la cola  $Q$ . Sea además  $T$  la variable guardado en la iteración anterior.

- Como  $u$  fue extraído en la iteración  $n + 1$ , significa que es el nodo con la arista más barata para conectar directamente con **algún nodo** de  $G_n$ .
- A saber, dicho nodo es  $\pi[u]$ .
- Al agregar la arista  $(\pi[u], u)$  a  $T$ , obtenemos un nuevo conjunto  $T'$  de  $G_{n+1}$ . Basta argumentar sus propiedades.

## Demostración

### $T'$ es **cubrimiento**

- Como  $T$  es MST para  $G_n$  por **H.I.**, entonces conecta todos los nodos de  $G_n$  (incluyendo a  $\pi[u]$ ).
- Al agregar  $(\pi[u], u)$ , se conecta también a  $u$ . Por lo tanto,  $T'$  cubre todo  $G_{n+1}$

### $T'$ es **árbol**

- Por **H.I.**,  $T$  es árbol.
- Al agregar  $(\pi[u], u)$ , se conecta un nodo ya incluído en  $G_n$  con uno **no incluído**, de forma que no se producen ciclos y  $T'$  no es árbol

### $T'$ es de **costo mínimo**

- Por **H.I.**,  $T$  es de costo mínimo para  $G_n$ .
- La elección de  $u$  asegura que se agrega la arista más barata para conectar  $u$  a  $G_n$ . Se concluye que  $T'$  es de costo mínimo.

## Demostración

**Finitud.** Es claro que el algoritmo termina, pues no visita nodos ya descubiertos y cada arista es revisada a lo más una vez. Como el grafo es finito, el algoritmo es finito y termina cuando  $n = |V|$ .

De lo anterior se concluye que  $P(n)$  es cierta. En particular, como  $G_{|V|} = G$

$P(|V|)$  verdadera  $\Leftrightarrow$  Prim entrega MST para  $G$



No olvidar: no necesariamente hay un único MST para  $G$

# Complejidad

Considerando que usamos una cola de prioridad implementada como heap

- Extracción del más prioritario  $\mathcal{O}(\log(V))$
- Actualización de prioridad cuando cambia el costo  $\mathcal{O}(\log(V))$

La complejidad en el peor caso viene dada por las iteraciones del **while**

- El loop ocurre  $|V|$  veces  $\mathcal{O}(V)$
- En cada loop se extrae un nodo  $\mathcal{O}(\log(V))$
- Cada arista se revisa exactamente una vez  $\mathcal{O}(E)$
- Por cada revisión se hace una actualización de costo  $\mathcal{O}(\log(V))$
- Total  $\mathcal{O}((V + E) \log(V))$

Podemos simplificar este último resultado



# Complejidad

Recordemos que para grafos conexos existe una relación entre  $|V|$  y  $|E|$

- Si  $G$  es un árbol,  $E = V - 1$
- Si  $G$  es conexo cualquiera,  $E \geq V - 1$

Concluimos que  $V \in \mathcal{O}(E)$  para grafos conexos

- Luego,  $(V + E) \in \mathcal{O}(E)$

El algoritmo de Prim toma tiempo  $\mathcal{O}(E \log(V))$

# Algoritmo de Prim: una versión más concreta

Prim( $s$ ):

```
1    $Q \leftarrow$  cola de prioridades con  $s$  (Min Heap)
2    $T \leftarrow$  lista vacía
3    $s.key \leftarrow 0$ ;  $s.parent \leftarrow \emptyset$ 
4   while  $Q$  no está vacía :
5        $u \leftarrow \text{Extract}(Q)$ ;  $u.color \leftarrow$  negro
6       if  $u.parent \neq \emptyset$  :
7            $T \leftarrow T \cup \{(u.parent, u)\}$ 
8       for  $v \in \alpha[u] \wedge v.color \neq \text{negro}$  :
9           if  $v \notin Q$  :
10              Insert( $Q, v$ )
11           if  $v.key > cost(u, v)$  :
12               $v.key \leftarrow cost(u, v)$ ;  $v.parent \leftarrow u$ 
13              DecreaseKey( $Q, v, v.key$ )
14   return  $T$ 
```

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

# Sumario

Introducción

Algoritmo de Prim

Análisis del algoritmo

**Cierre**

# Objetivos de la clase

- ☐ Modificar prioridad dentro de un heap
- ☐ Comprender una segunda forma de resolver el problema de MST
- ☐ Demostrar correctitud y complejidad del algoritmo de Prim