MST y algoritmo de Prim

Clase 23

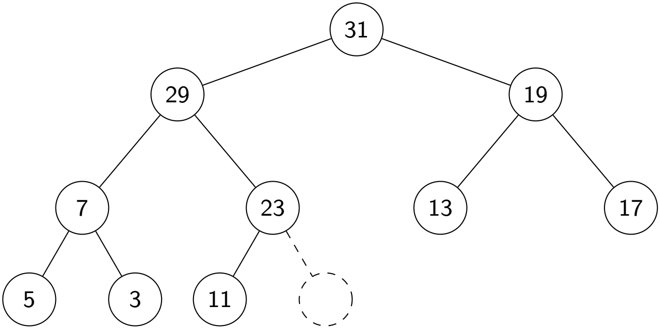
IIC 2133 - Sección 1

Prof. Sebastián Bugedo

Sumario

¿Cómo están?

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 $i = \lfloor n/2 \rfloor - 1 \dots 0$: ▷ 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

Rspecto 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 heps

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

`HeapSort`(A):

  `BuildHeap`(A)

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

    $A[0] \leftrightarrows A[i]$

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

    `ShiftDown`$(A, 0)$

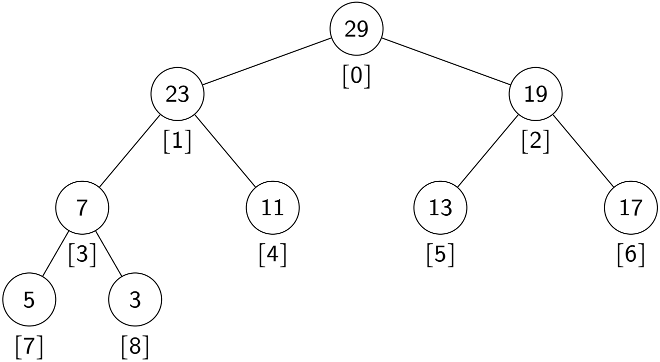
Respecto a su complejidad

- `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

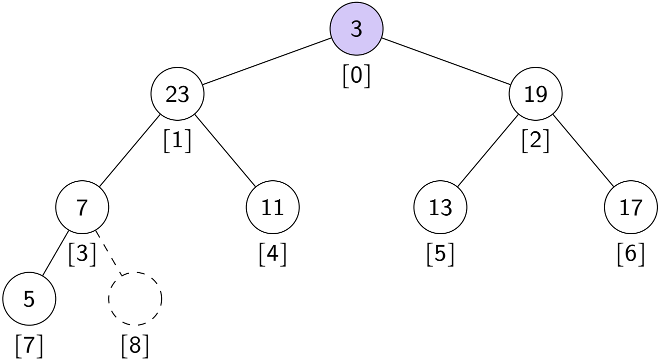
Supongamos que ya contamos con el heap resultante de `BuildHeap`



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

Heapsort en acción

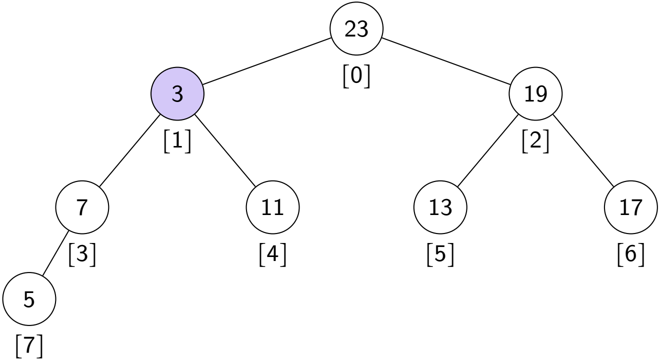
Movemos el primer elemento y reducimos el tamaño del heap en 1



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

Heapsort en acción

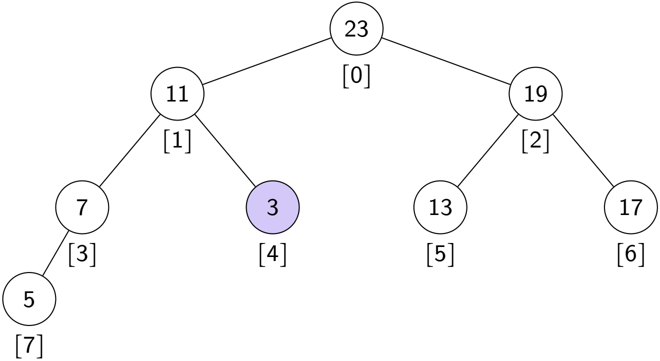
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \ldots 7]$)



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

Heapsort en acción

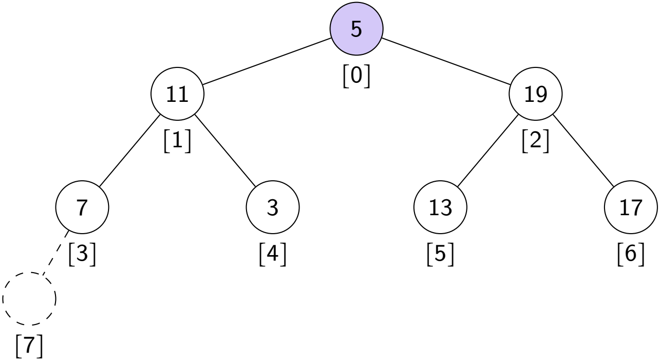
Aplicamos `SiftDown`$(A, 1)$ (el heap es $A[0 \ldots 7]$)



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

Heapsort en acción

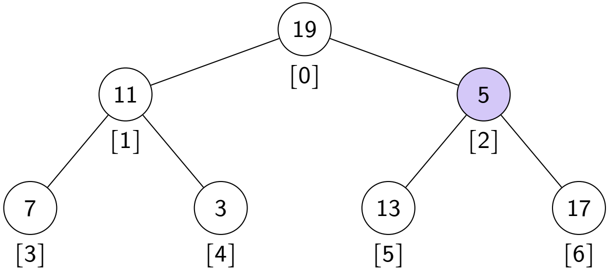
Repetimos el proceso con la nueva raíz



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

Heapsort en acción

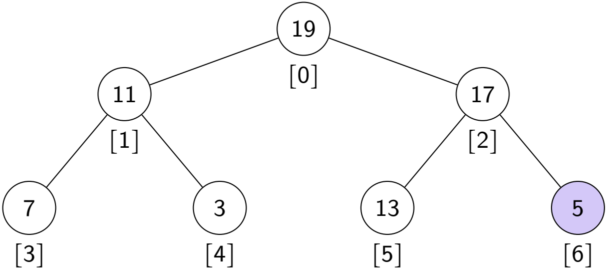
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \ldots 6]$)



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

Heapsort en acción

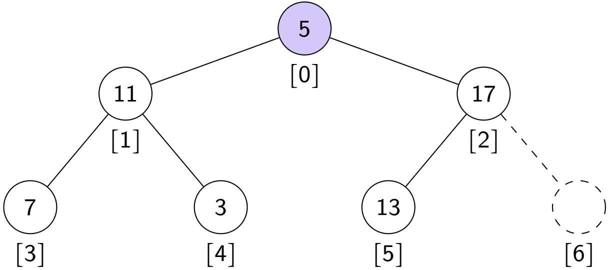
Aplicamos `SiftDown`$(A, 2)$ (el heap es $A[0 \ldots 6]$)



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

Heapsort en acción

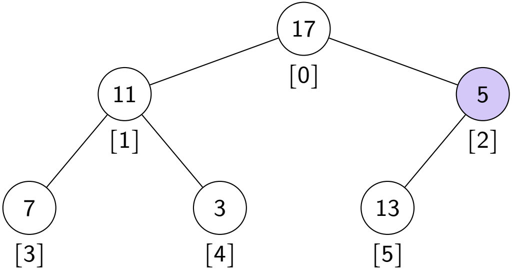
Repetimos el proceso con la nueva raíz



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

Heapsort en acción

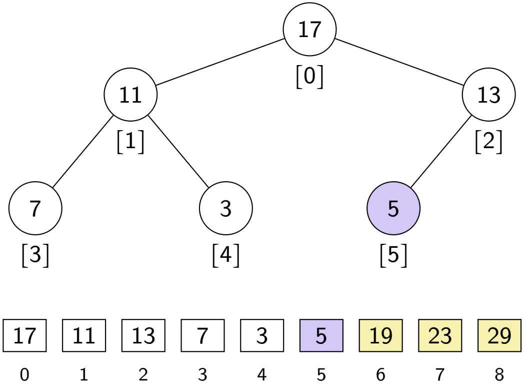
Aplicamos `SiftDown`$(A, 0)$ (el heap es $A[0 \dots 5]$)



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

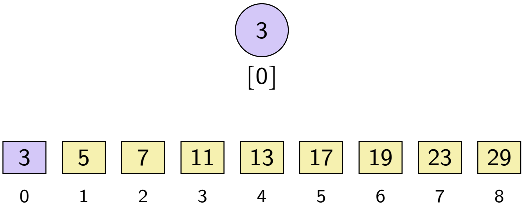
Heapsort en acción

Aplicamos `SiftDown`$(A, 2)$ (el heap es $A[0 \ldots 5]$)



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 $k > H[i]$

$\texttt{IncreaseKey}(H, i, k)$**:**

$\quad H[i] \leftarrow k$

$\quad$ **while** $i > 0 \land H[\texttt{Parent}(i)] < H[i]$ **:**

$\quad\quad H[i] \leftrightharpoons H[\texttt{Parent}(i)]$

$\quad\quad i \leftarrow \texttt{Parent}(i)$

En tiempo $\mathcal{O}(\log(V))$ actualizamos la prioridad y mantenemos el heap

Conectividad digital

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

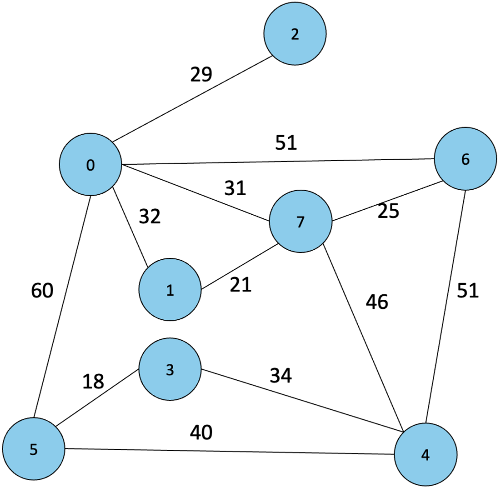
- Objetivo: instalar fibra óptica subterránea entre pares de puntos relevantes
- Cada instalación de ese cableado tiene un costo
- Es prioritario conectar ciudades más pobladas

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

¿Qué tipo de grafo es más adecuado para representar el 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**

¿Qué forma tiene el **subgrafo** que buscamos?

Objetivos de la clase

- ☐ Comprender el uso de heaps para ordenar
- ☐ Modificar prioridad dentro de un heap
- ☐ Comprender el concepto de árbol de cobertura mínimo
- ☐ Resolver el problema de MST con el algoritmo de Prim

Sumario

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

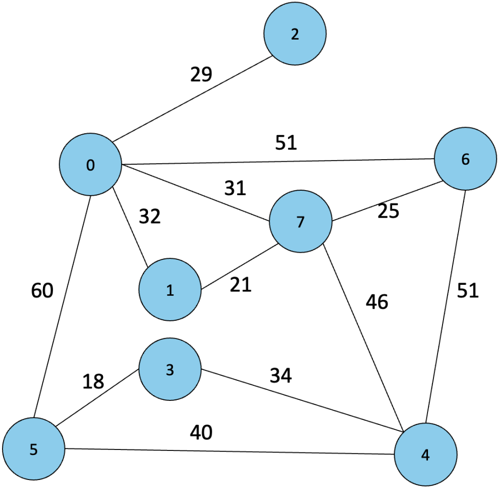
1. T es un árbol
2. $V(T) = V(G)$
3. No existe otro MST T' para G con menor costo total

Es decir, si T es MST de G,

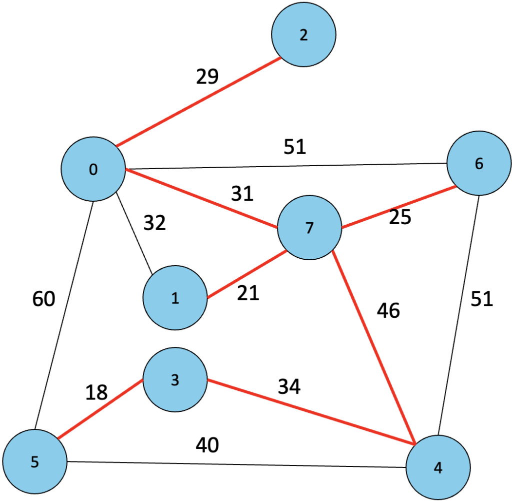
1. no tiene ciclos
2. es una **cobertura** de los nodos de G
3. tiene costo mínimo

¿Puede haber más de un MST diferente para G?

Árboles de cobertura mínimos



Árboles de cobertura mínimos



Árboles de cobertura mínimos

Los MST están presentes en la solución de múltiples problemas de conectividad

- Redes de distribución eléctrica
- Redes telefónicas
- Comunicaciones, computacionales, tráfico aéreo. . .
- Incluso redes biológicas, químicas y físicas

Árboles de cobertura mínimos

Para atacar el problema algorítmicamente, dado G no dirigido

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

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

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

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

Árboles de cobertura mínimos

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?

Sumario

Algoritmo de Prim

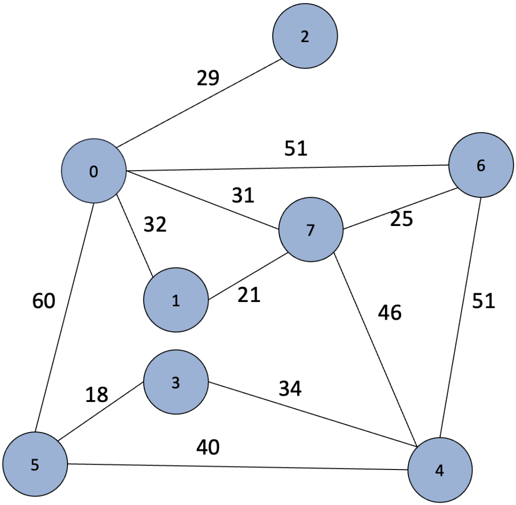
La idea detrás del **algoritmo de Prim** es utilizar las aristas que cruzan cortes para guiar la construcción

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

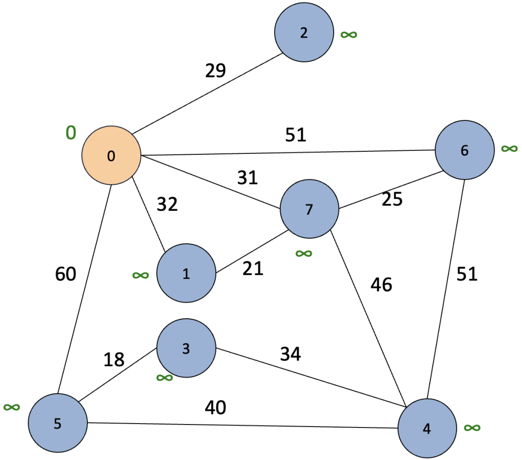
1. Sean $R = \{v\}$ y $\bar{R} = V - R$
2. Sea e la arista de menor costo que cruza de R a $\bar{R}$
3. Sea u el nodo de e que pertenece a $\bar{R}$
4. Agregar e al MST. Eliminar u de $\bar{R}$ y agregarlo a R
5. Si quedan elementos en $\bar{R}$, volver al paso 2.

¿Cómo hacer eficiente el paso 2?

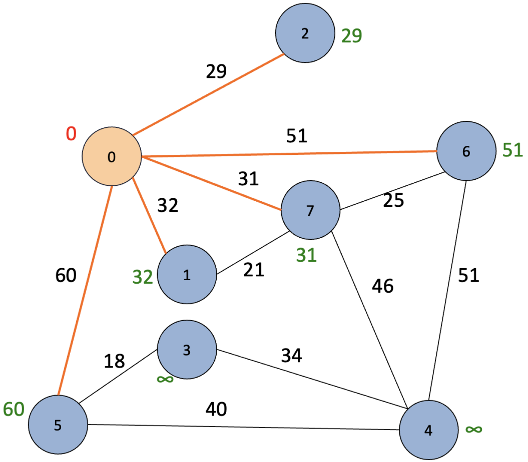
Árboles de cobertura mínimos



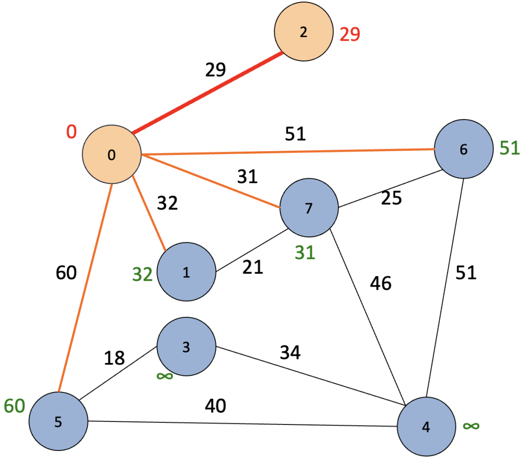
Árboles de cobertura mínimos



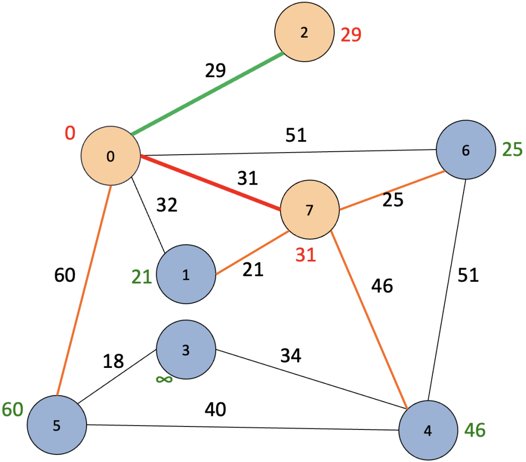
Árboles de cobertura mínimos



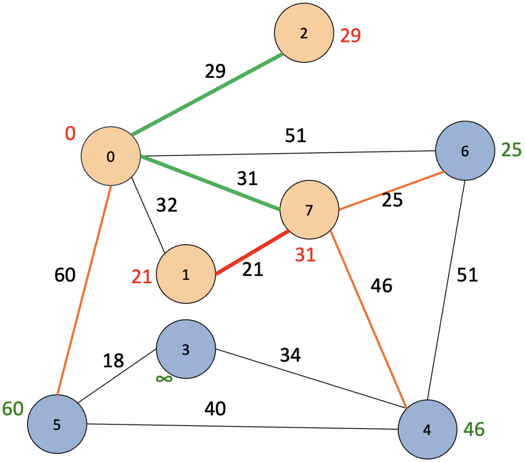
Árboles de cobertura mínimos



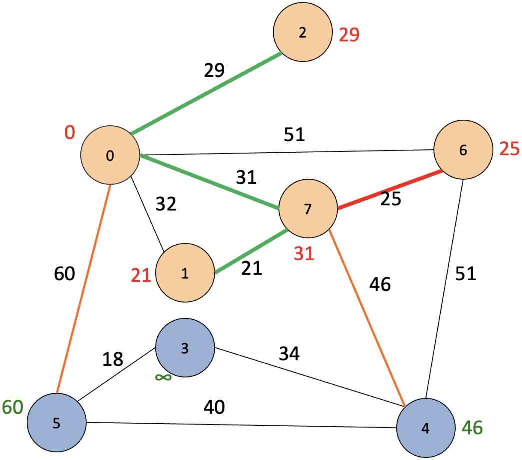
Árboles de cobertura mínimos



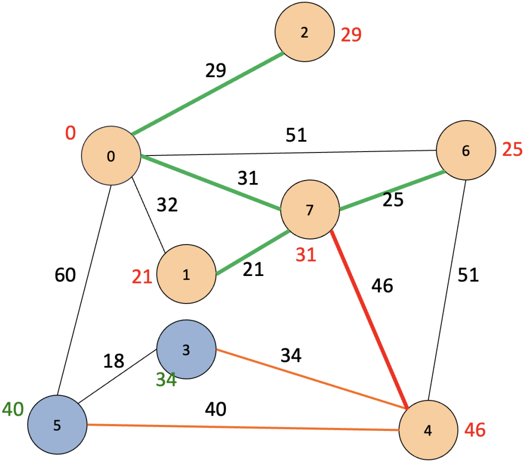
Árboles de cobertura mínimos



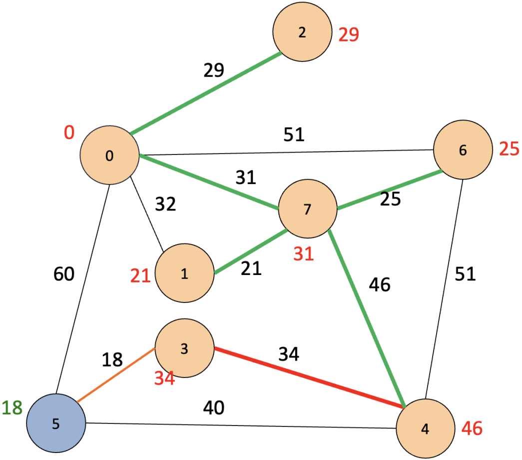
Árboles de cobertura mínimos



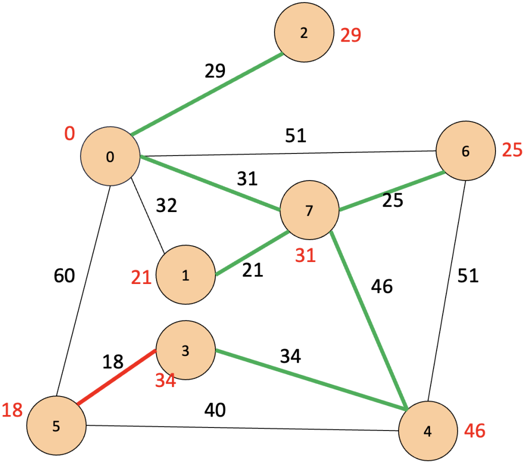
Árboles de cobertura mínimos



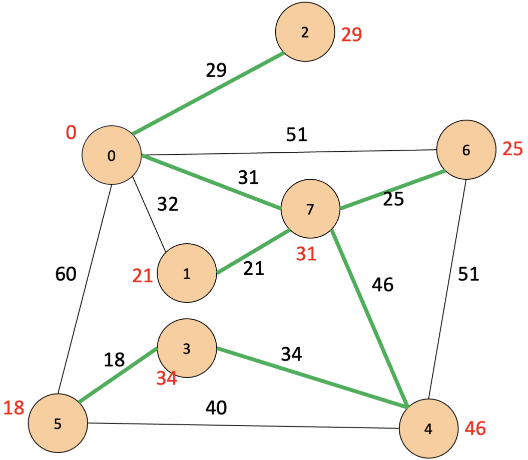
Árboles de cobertura mínimos



Árboles de cobertura mínimos



Árboles de cobertura mínimos



Algoritmo de Prim

```
Prim(s):
    Q ← cola de prioridades vacía
    T ← lista vacía
    for u ∈ V − {s} :
        d[u] ← ∞;  π[u] ← ∅;  Insert(Q, u)
    d[s] ← 0;  π[s] ← ∅;  Insert(Q, s)
    while Q no está vacía :
        u ← Extract(Q)
        T ← T ∪ {(π[u], u)}
        for v ∈ α[u] :
            if v ∈ Q :
                if d[v] > cost(u, v) :
                    d[v] ← cost(u, v)
                    π[v] ← u
    return T
```

Importante: la cola de prioridades usa $d[v]$ como prioridad de v

Correctitud

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.

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

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

Correctitud

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.

Correctitud

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.

Correctitud

Demostración

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

$$P(|V|) \text{ verdadera} \quad \Leftrightarrow \quad \texttt{Prim} \text{ entrega MST para } G$$

$\square$

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, $E \geq V - 1$

Concluímos 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):
    Q ← cola de prioridades con s
    T ← lista vacía
    x.key ← 0;   x.parent ← ∅
    while Q no está vacía :
        u ← Extract(Q);   u.color ← negro
        if u.parent ≠ ∅ :
            T ← T ∪ {(u.parent, u)}
        for v ∈ α[u] ∧ v.color ≠ negro :
            if v ∉ Q :
                Insert(Q, v)
            if v.key > cost(u, v) :
                v.key ← cost(u, v)
                v.parent ← u
    return T
```

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

Sumario

Objetivos de la clase

- ☐ Comprender el uso de heaps para ordenar
- ☐ Modificar prioridad dentro de un heap
- ☐ Comprender el concepto de árbol de cobertura mínimo
- ☐ Resolver el problema de MST con el algoritmo de Prim

Epílogo

Ve a

www.menti.com

Introduce el código

4236 2491

O usa el código QR