octubre 2025

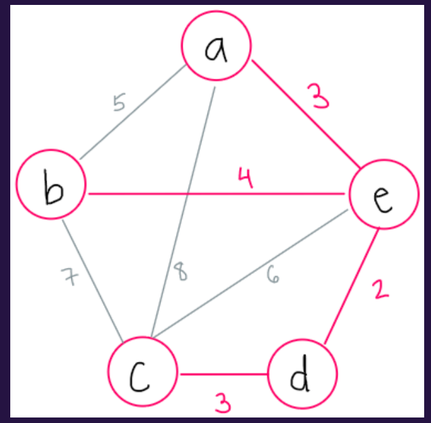
KRUSKAL, PRIM Y HEAPS

Joaquín - Steven - Elías - Alexander - Paula

MST:

MINIMUM SPANNING TREE

MST

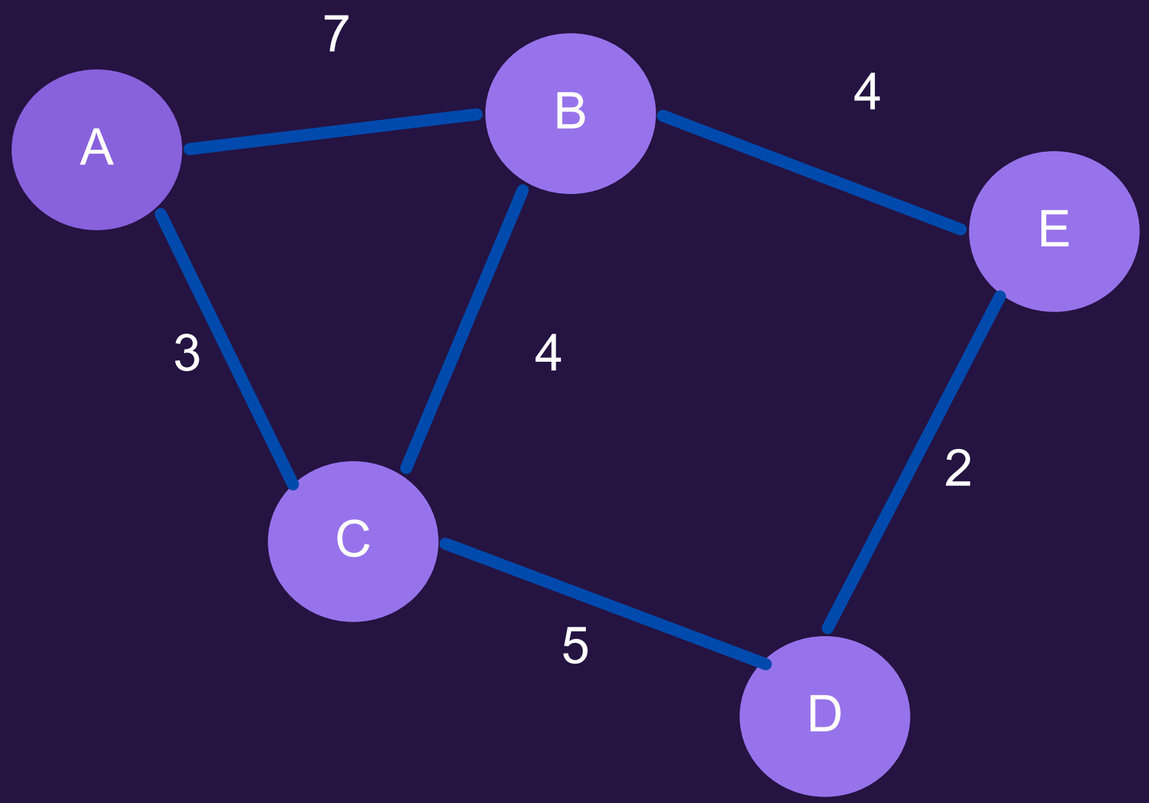


Definición

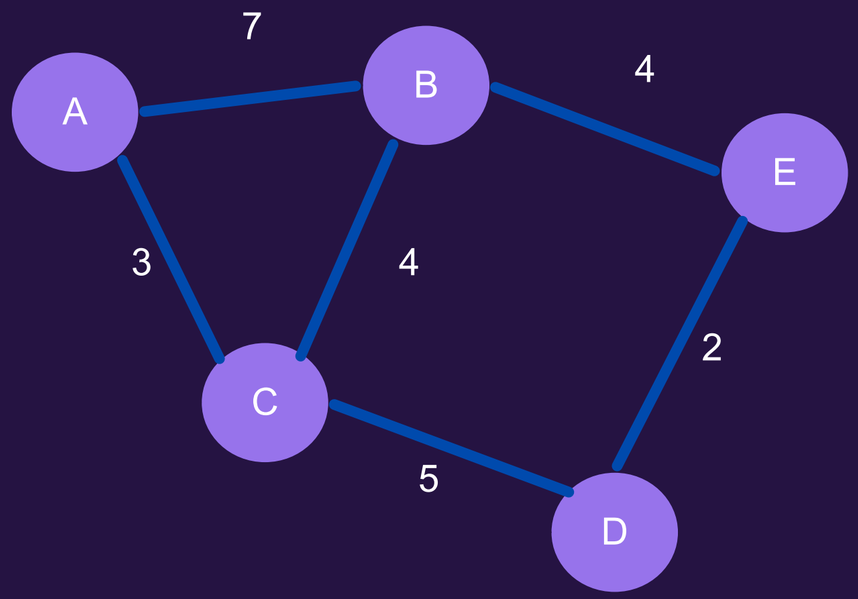
- Dado un grafo G no dirigido, un subgrafo T se dice MST de G si:
- T es un árbol
- V(T) = V(G)
- No existe otro MST T’ para G con menor costo total (notar que no dice que pueda haber otro con igual costo)

EJEMPLO

Cual es el mst de este grafo?

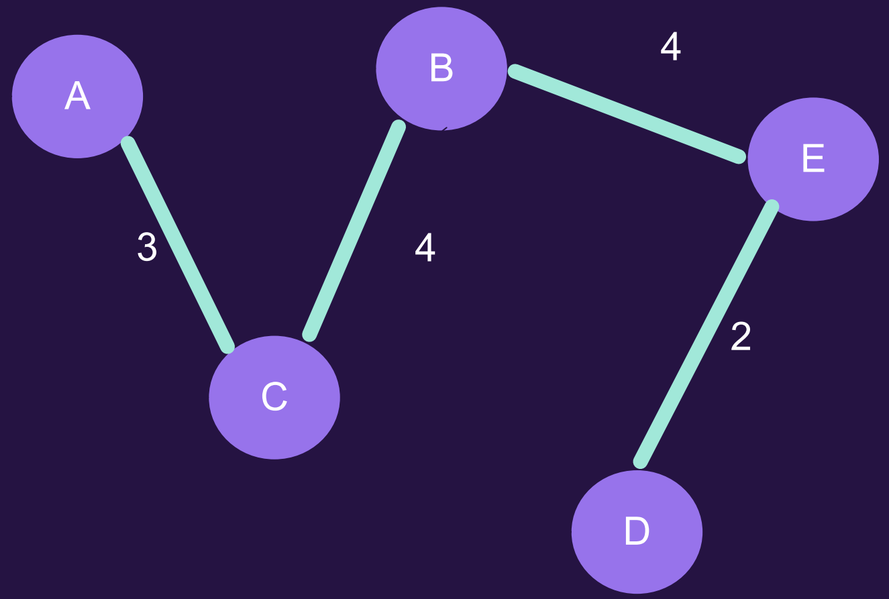


EJEMPLO



su mst

MST con costo = 13



CÓMO OBTENEMOS EL MST DE UN ÁRBOL DE FORMA EFICIENTE?

RESP: CON KRUSKAL Y PRIM :)

KRUSKAL

KRUSKAL

Idea base: Crear un bosque que converge en un único árbol

Sea G = (V, E) grafo no dirigido 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

HAY UN PROBLEMA CLAVE EN ESTE ALGORITMO

¿CÓMO DETECTAMOS SI UNA ARISTA PRODUCE O NO UN CICLO?

USANDO CONJUNTOS DIJUNTOS

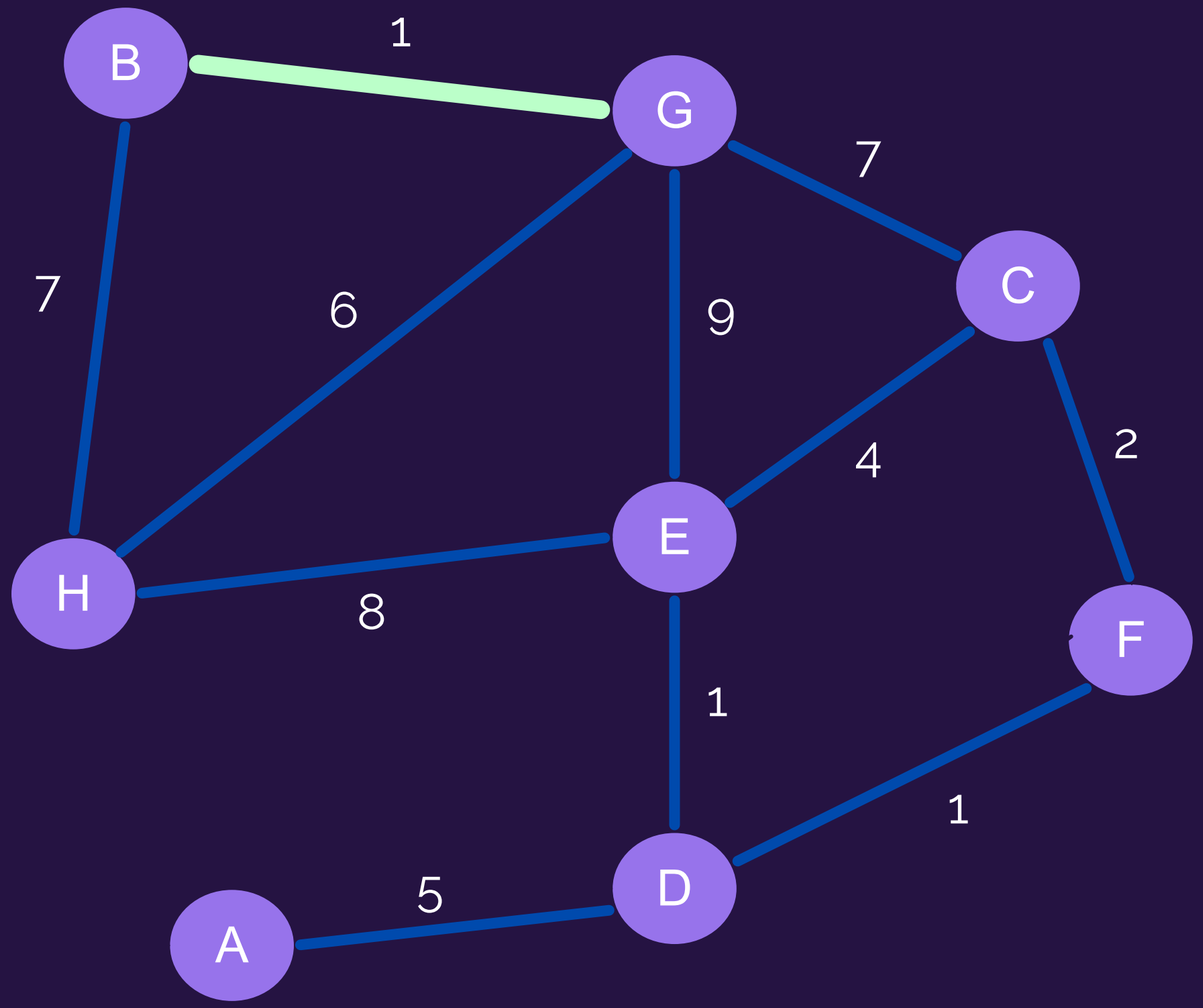
1. Asignamos un set a cada nodo dentro del grafo
2. Cuando agregamos una arista unimos sus sets
3. Al revisar más aristas a futuro, sabemos que si dos nodos se encuentran en el mismo set, entonces ya existe un camino que los conecta, por lo que agregar esa arista nos generaría un ciclo

ALGORITIMO

```
Kruskal(G):
1     E ← E ordenada por costo, de menor a mayor
2     for v ∈ V :
3         MakeSet(v)
4     T ← lista vacía
5     for (u, v) ∈ E :
6         if Find(u) ≠ Find(v) :
7             T ← T ∪ {(u, v)}
8             Union(u, v)
9     return T
```

* Find(u) entrega el conjunto al que pertenece el nodo u

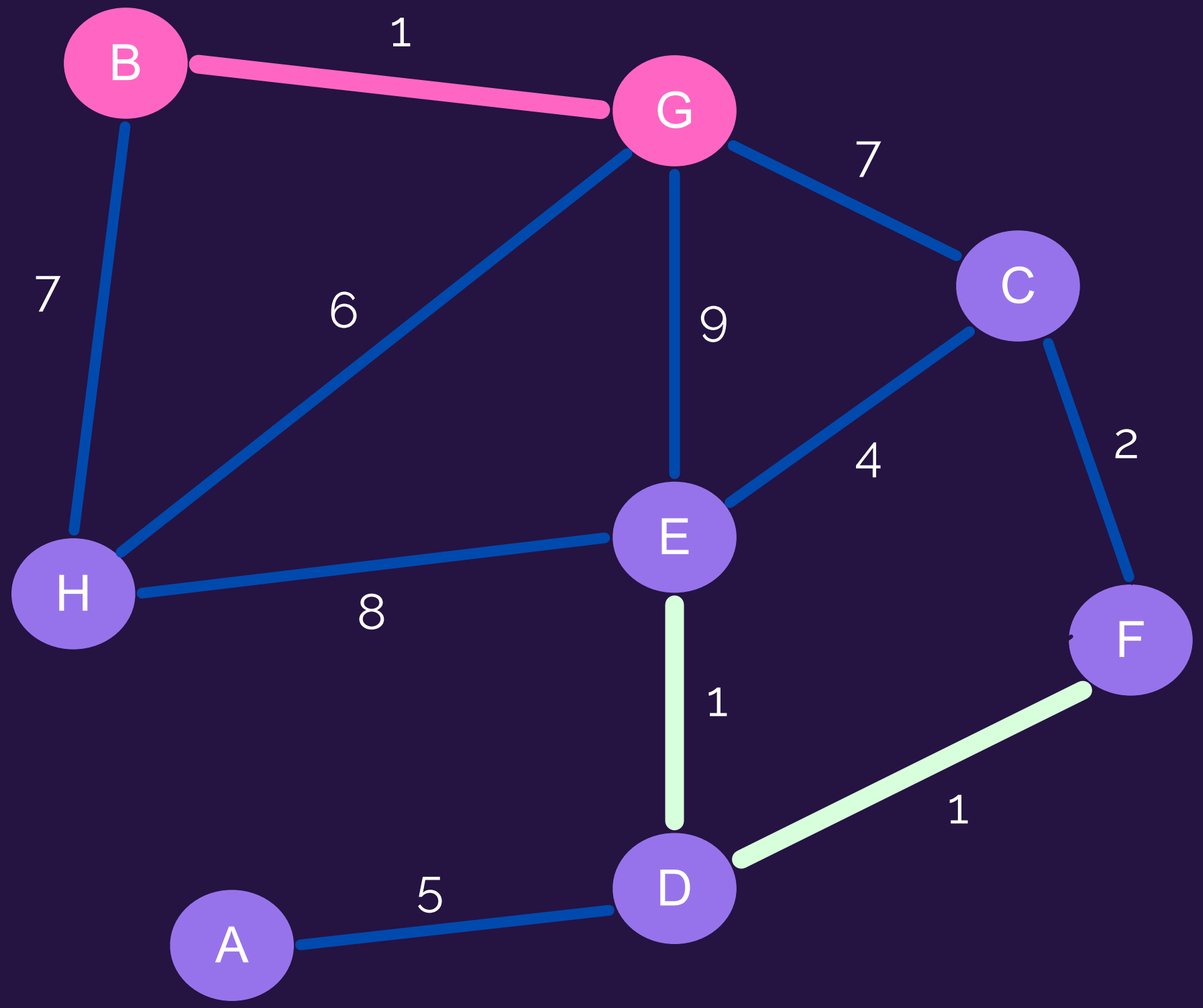
EJEMPLO



Aristas candidatas

Aristas en MST

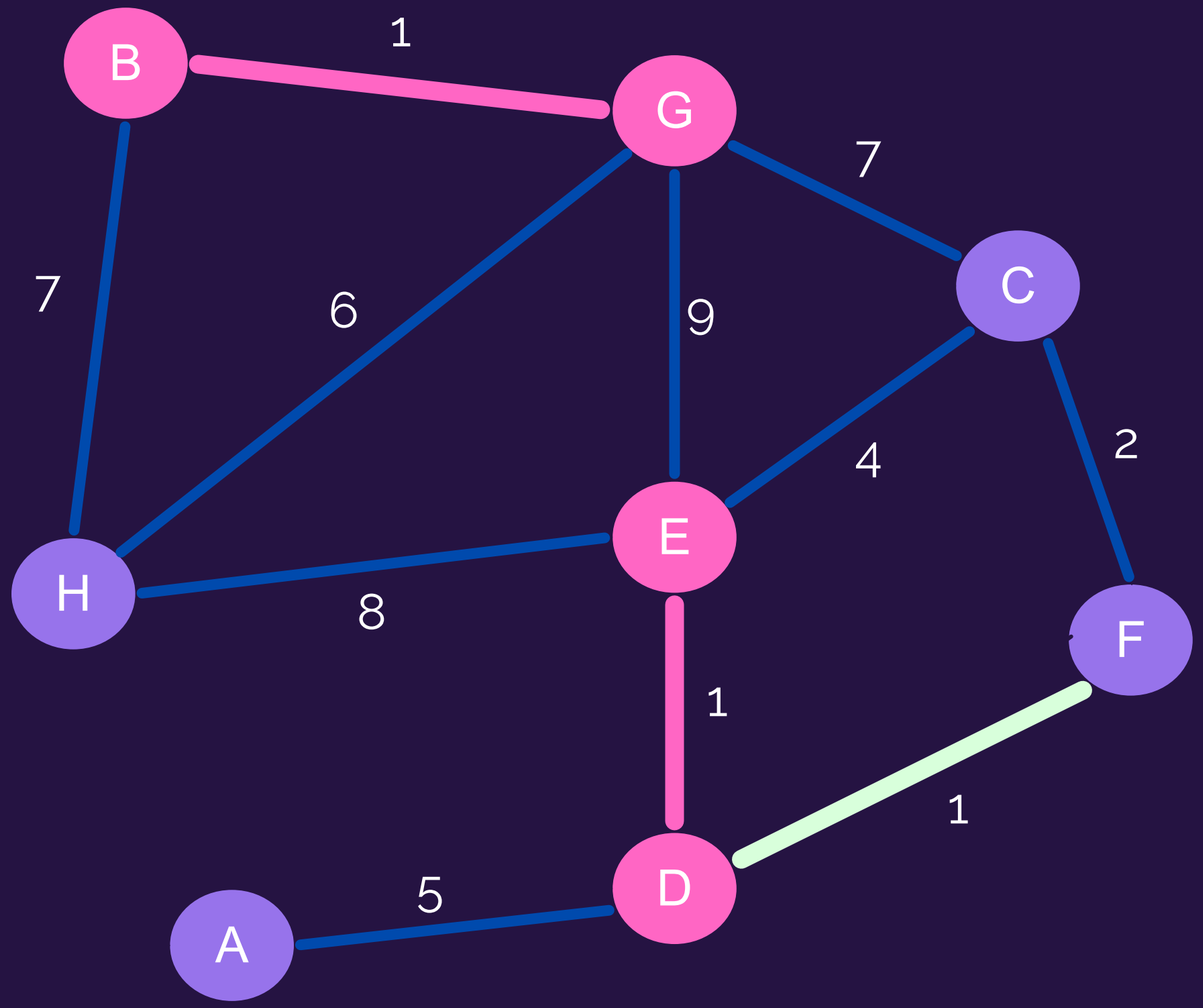
EJEMPLO



Aristas candidatas

Aristas en MST

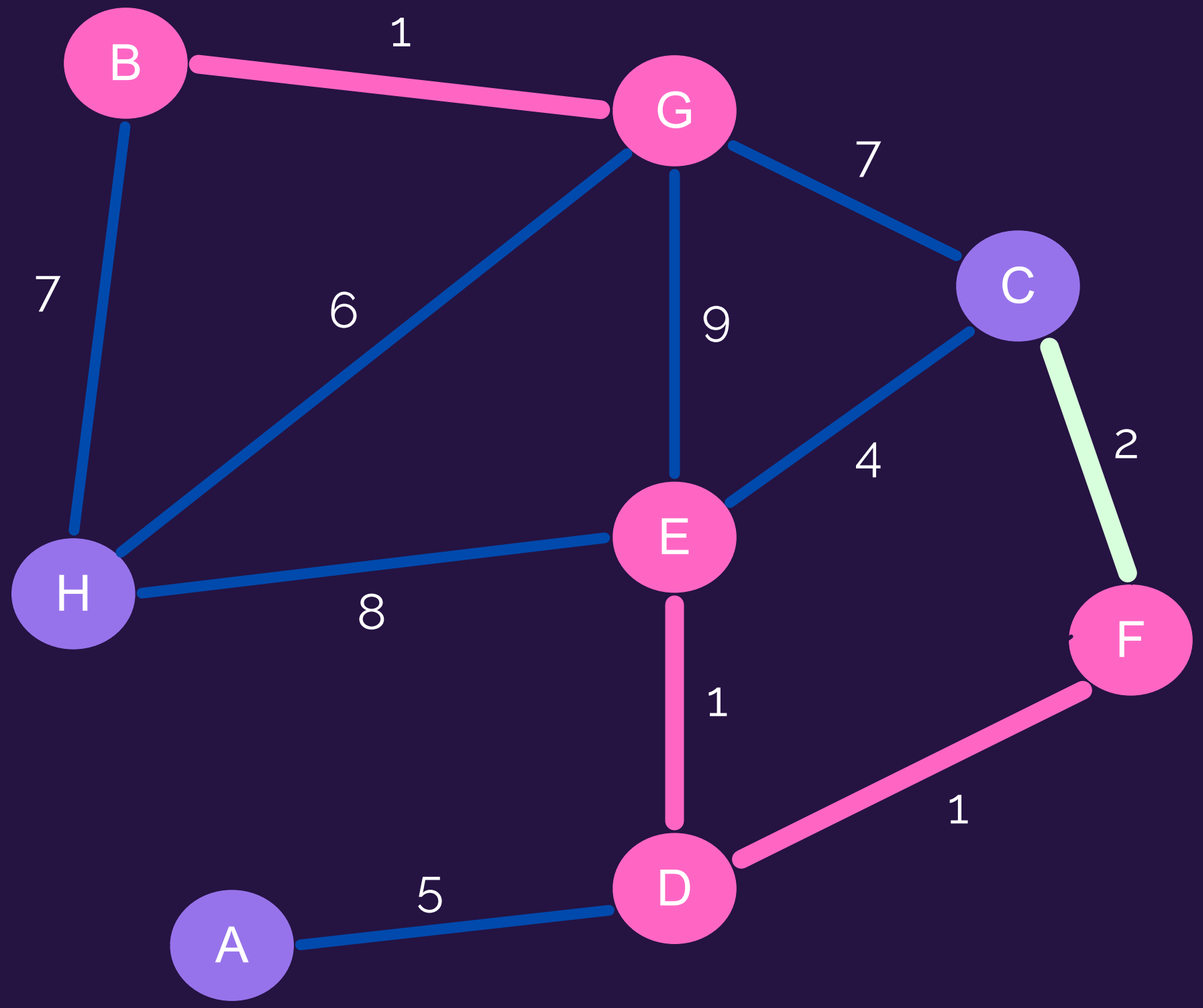
EJEMPLO



Aristas candidatas

Aristas en MST

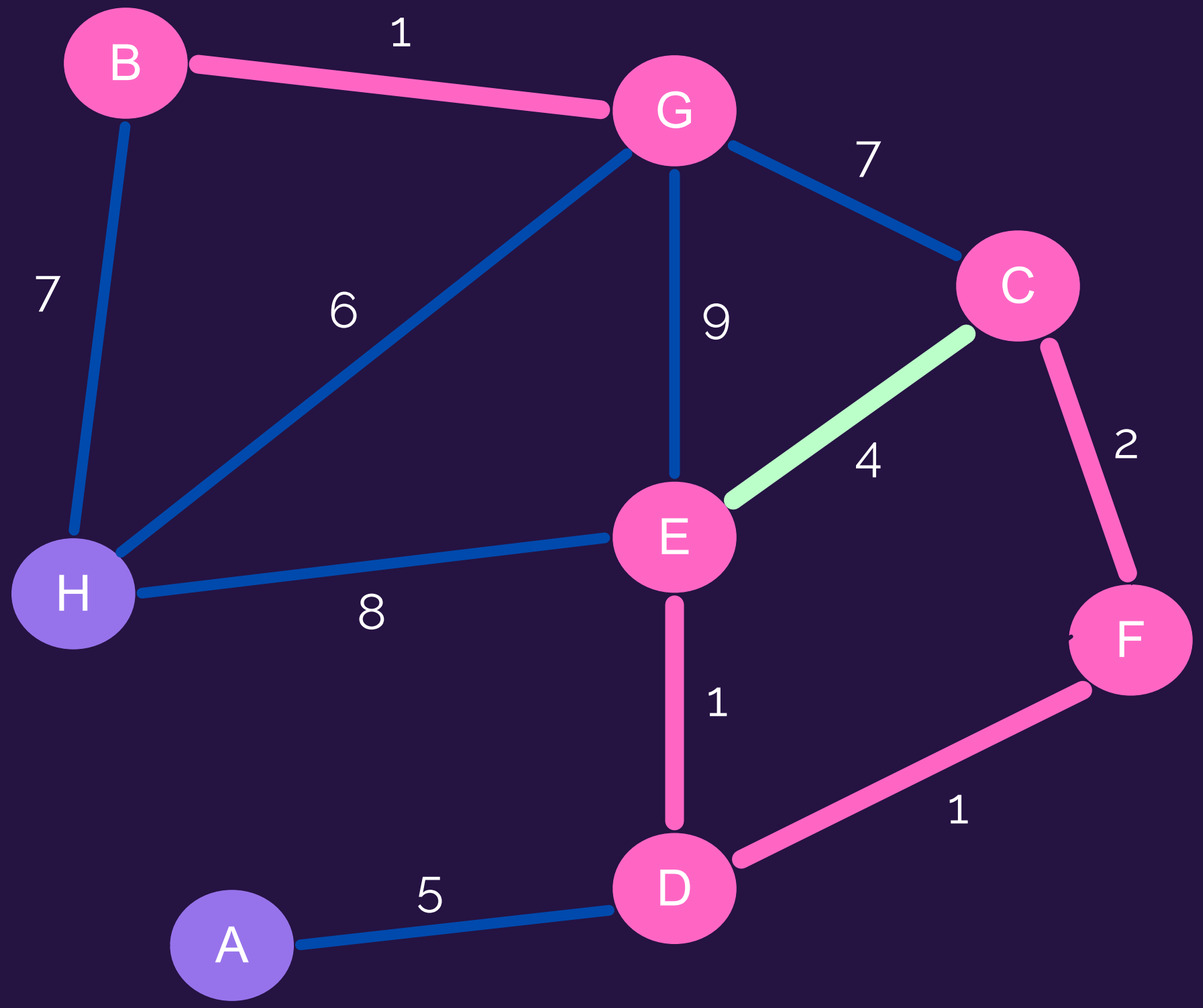
EJEMPLO



Aristas candidatas

Aristas en MST

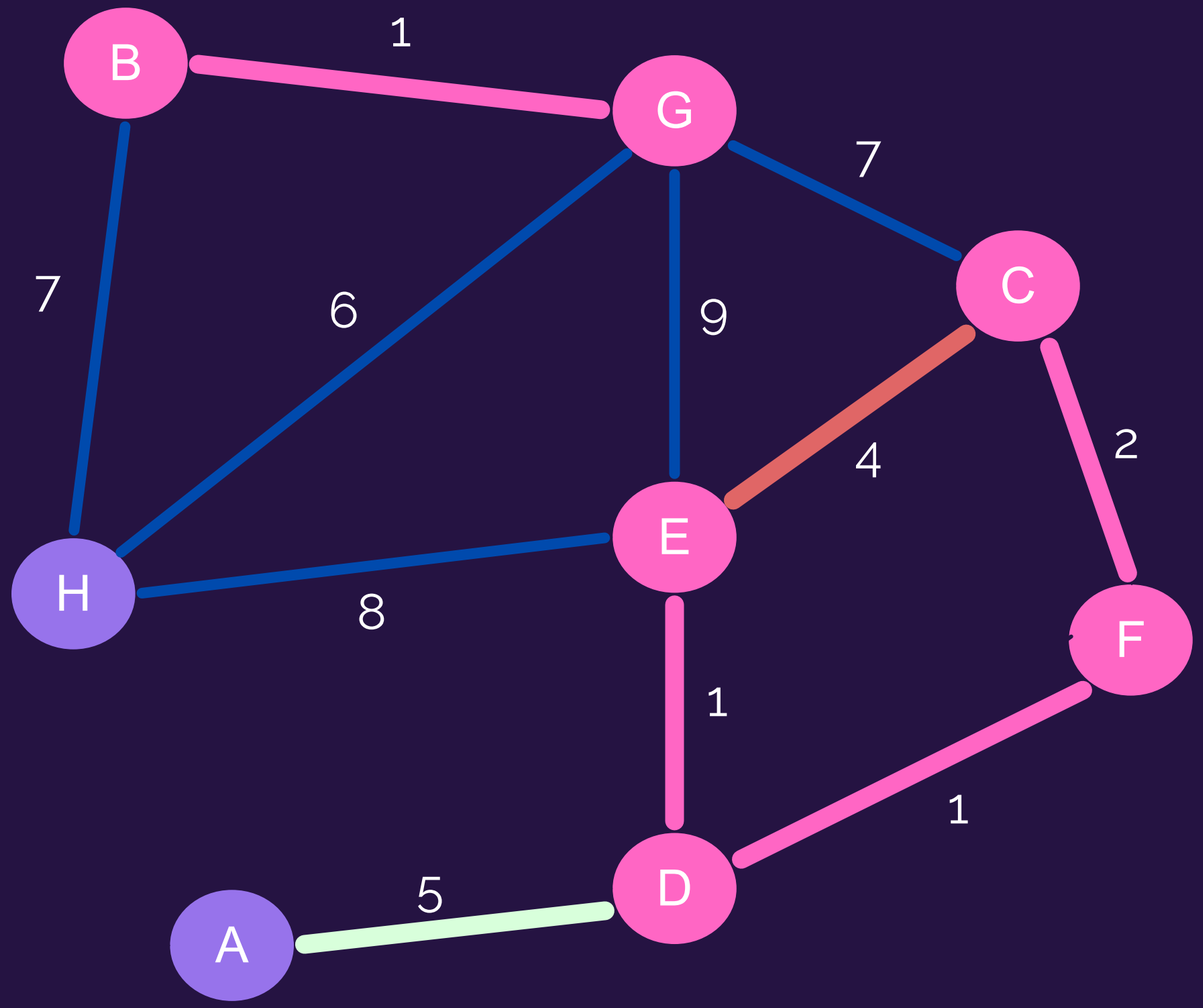
EJEMPLO



Aristas candidatas

Aristas en MST

EJEMPLO

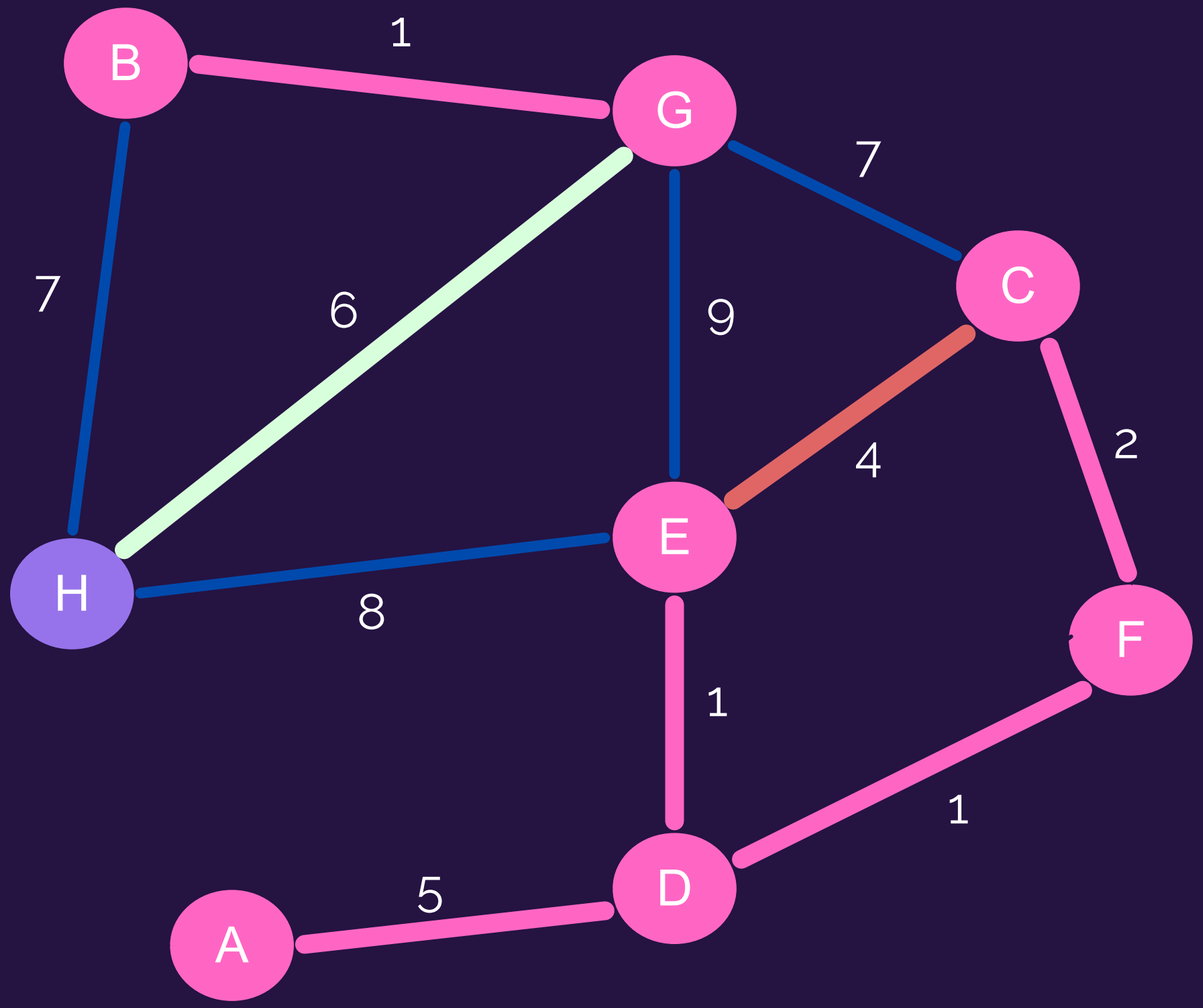


Aristas ignoradas

Aristas candidatas

Aristas en MST

EJEMPLO

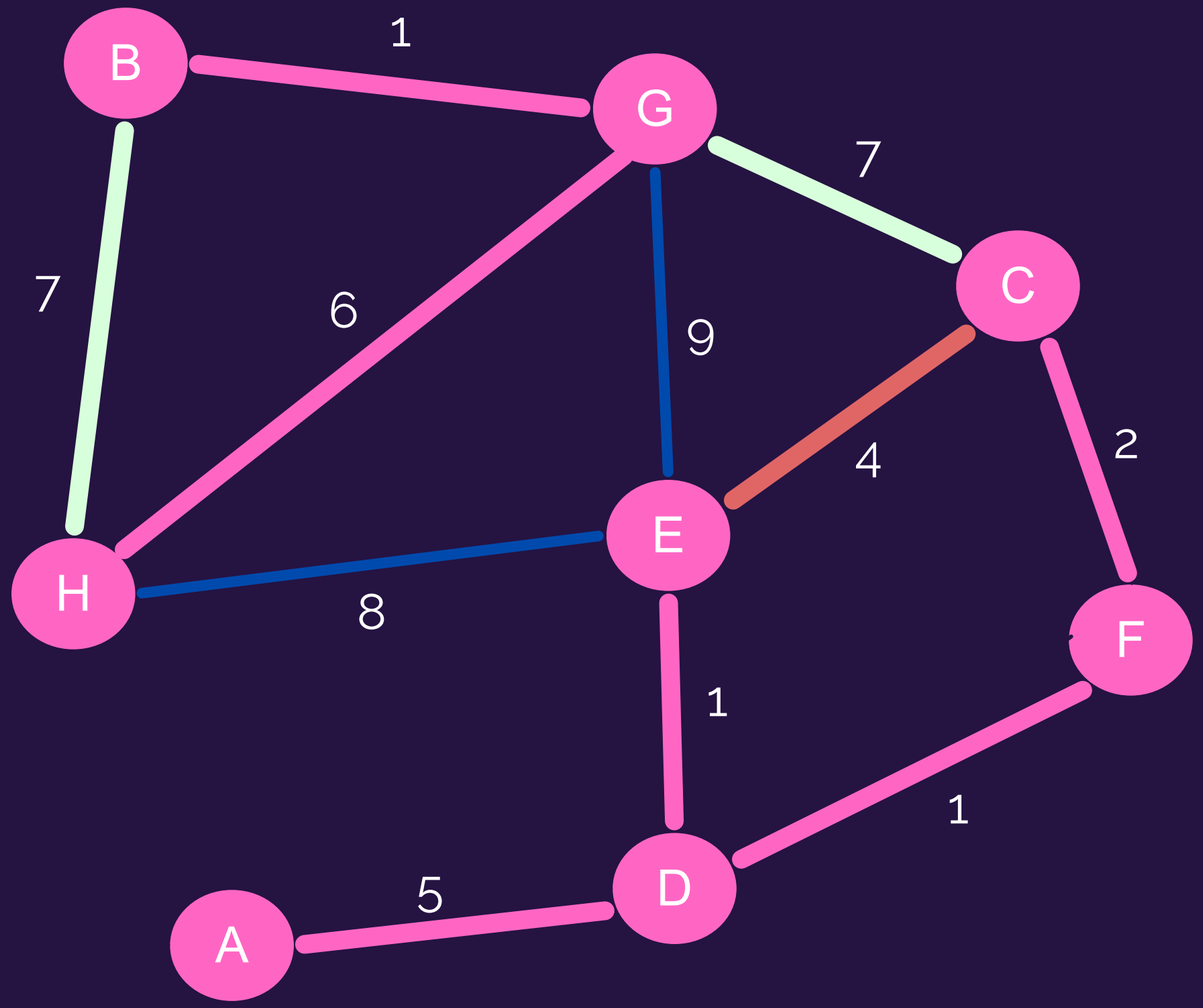


Aristas ignoradas

Aristas candidatas

Aristas en MST

EJEMPLO

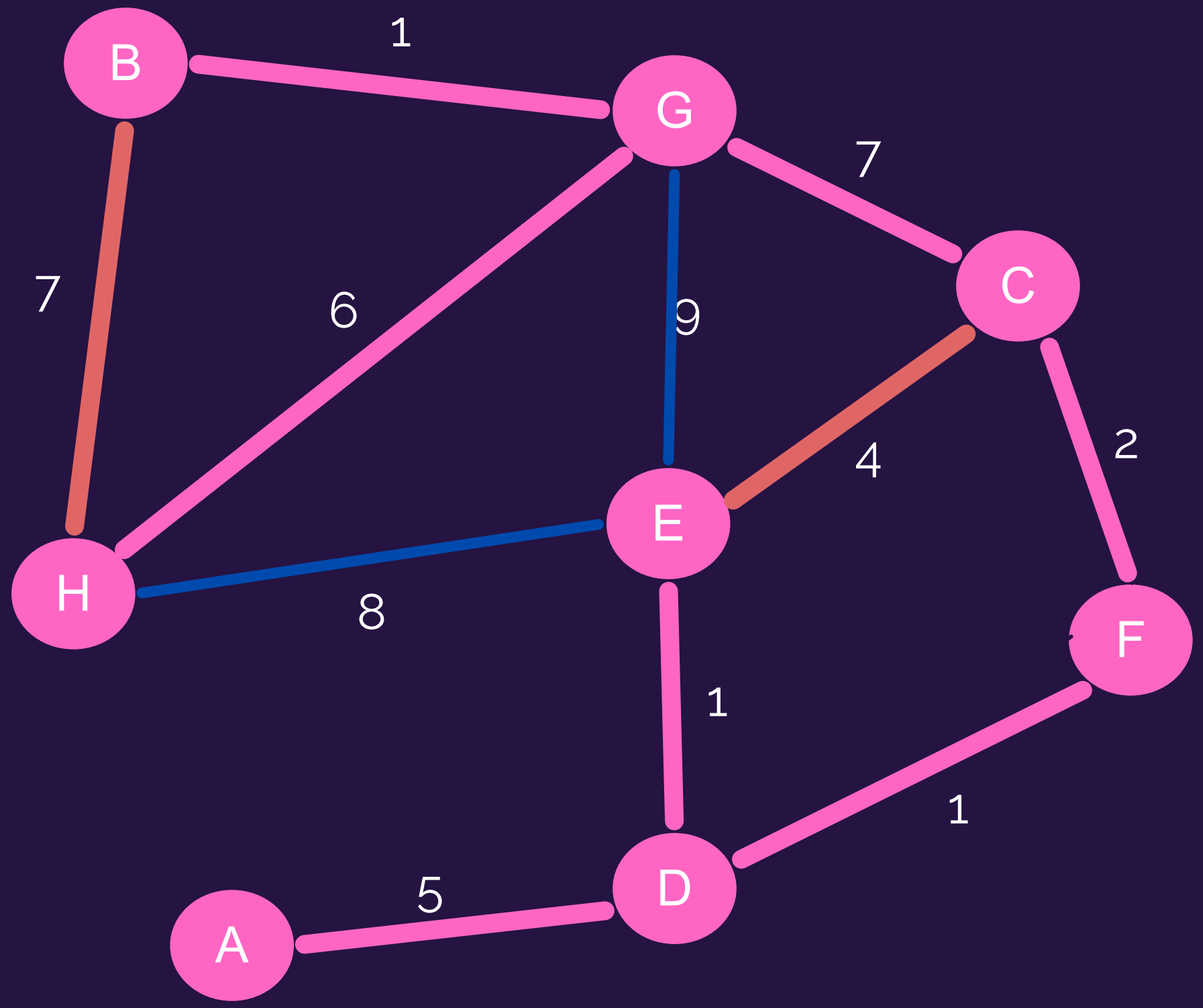


Aristas ignoradas

Aristas candidatas

Aristas en MST

EJEMPLO

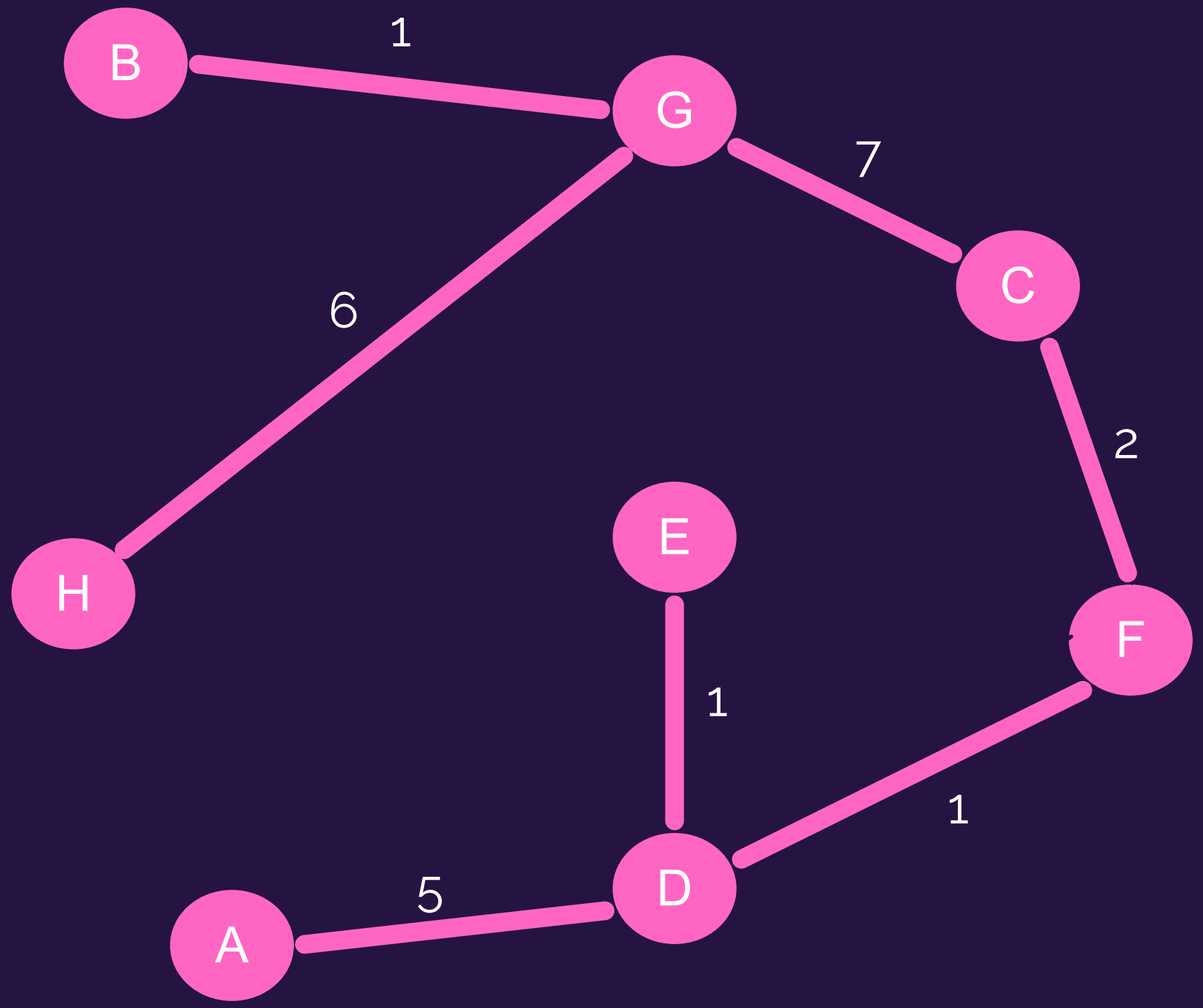


Aristas ignoradas

Aristas candidatas

Aristas en MST

RESULTADO



Aristas en MST

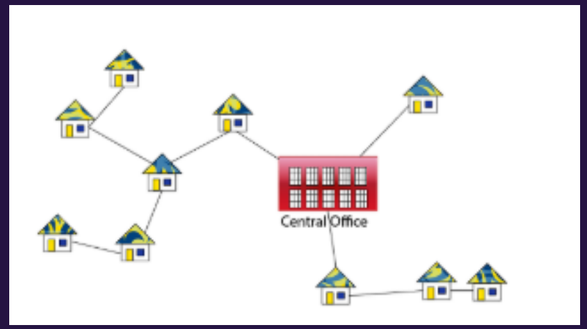
KRUSKAL
→ EJERCICIO

I3 2023-1 P3

Una estrategia base para **asegurar cobertura de comunicaciones** es contar con **diferentes servicios de conexión.** Por ejemplo, el servicio de Tesorería (TGR) puede conectar sus oficinas y puntos de atención mediante:

Redes de fibra de proveedores, Redes de telefonía móvil, Red StarLink, Redes Satelitales, Radioaficionados y Banda Local.

Naturalmente, cada uno de ellos tienen diferentes costos, velocidades y latencias, y pueden estar disponibles o no entre diferentes oficinas y puntos de atención, o perderse al ocurrir una emergencia.



a) ¿De qué forma puede TGR decidir la mejor forma de conectar sus oficinas y puntos de atención en función de los servicios de conexión disponibles en cada oficina o punto?

Indique la forma de representar el problema y las estrategias conocidas para determinar la solución.

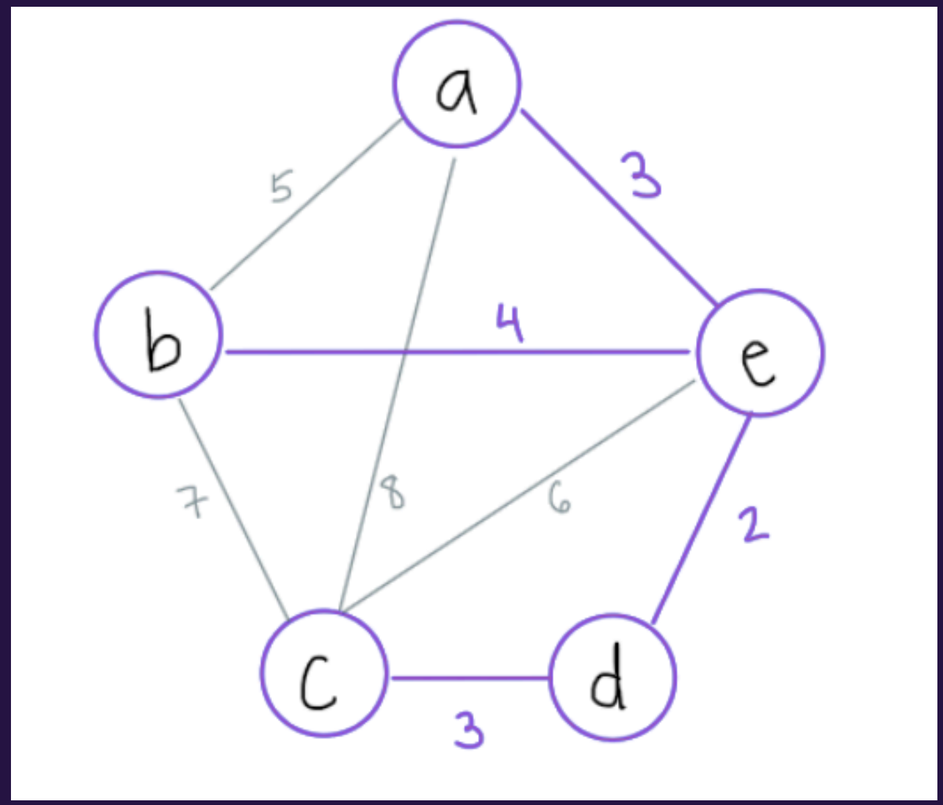
Pista: Define primero como determinar la mejor conexión en cada caso.

El problema se puede representar como un grafo no dirigido con costos en las aristas.

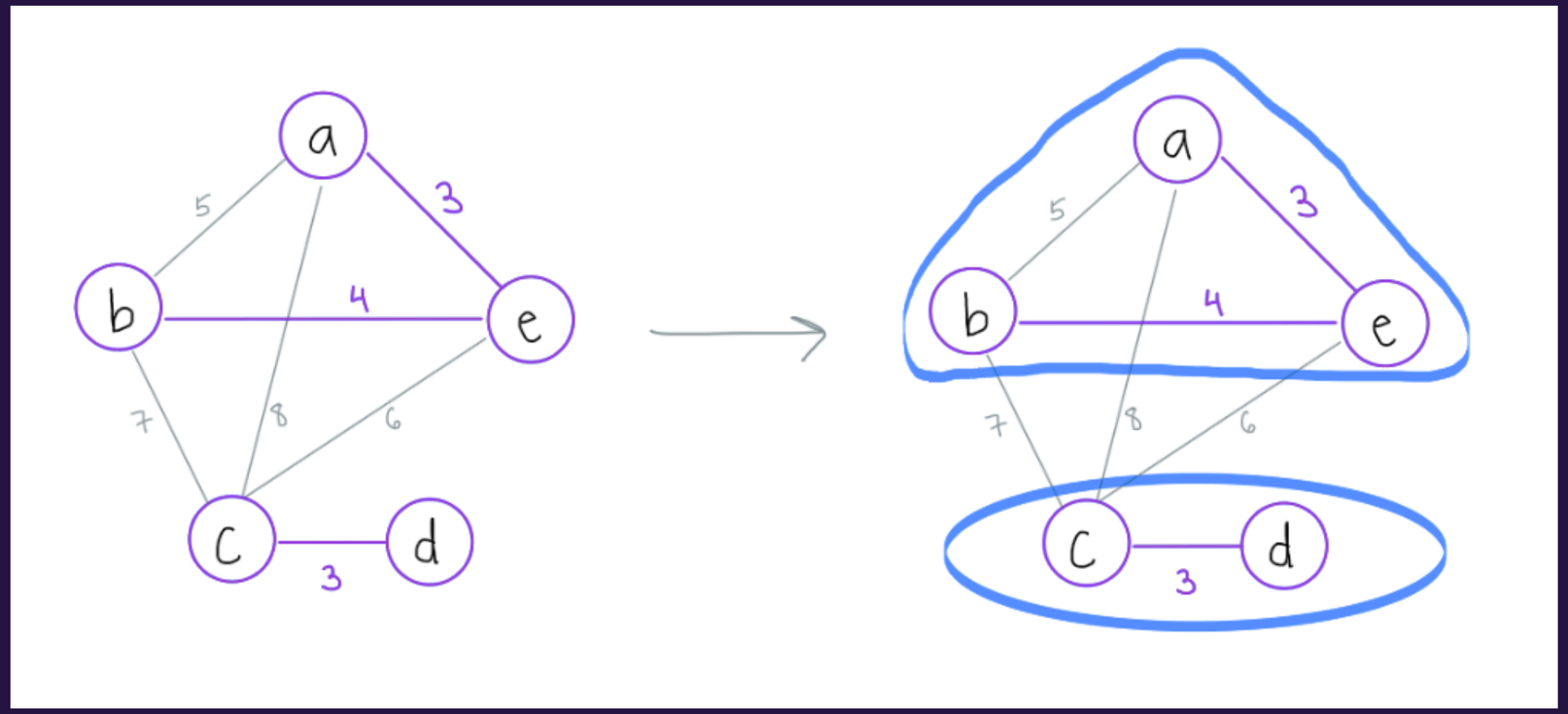
- **Nodos**: oficinas y puntos de atención
- **Aristas**: conexiones existentes entre nodos, con una arista por cada tipo de conexión disponible (puede haber más de una arista entre dos nodos).
- **Costo de cada arista:** función que relaciona costo, velocidad y latencia como un valor que describe el costo real de un tipo de conexión entre nodos. La mejor forma de conectar, basándose en minimizar costo total, es determinar el MST de este grafo con algún algoritmo estudiado como Prim o Kruskal.

b) Al ocurrir una emergencia, **TGR** pierde la conectividad entre dos oficinas a través de uno de los enlaces que es parte de la mejor forma de conectar ya escogida. Proponga una forma de recuperar la conectividad de la mejor forma disponible.
Justifique por qué es la mejor opción.

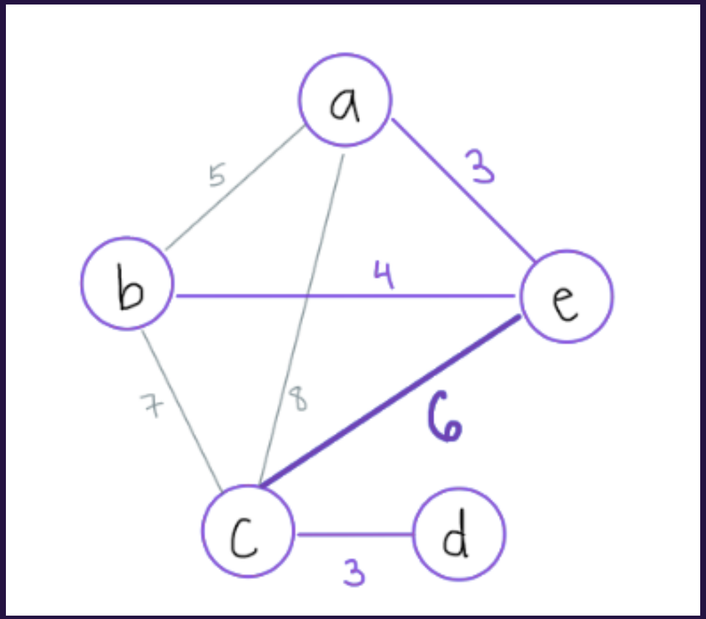
Supongamos que este es el grafo que representa el MST de las oficinas y puntos de atención.



Supongamos que este es el grafo que representa el MST de las oficinas y puntos de atención.



Sabemos por la estrategia de construcción de los MST que ambos MST son óptimos para conectar los nodos que cubren.
Luego, si consideramos que lo que tenemos al perder la arista es un “corte” entre los MST (como en Kruskal) si determinamos la **siguiente arista de las disponibles que tiene el menor costo y no genera un ciclo** podemos recuperar el MST con las aristas disponibles.



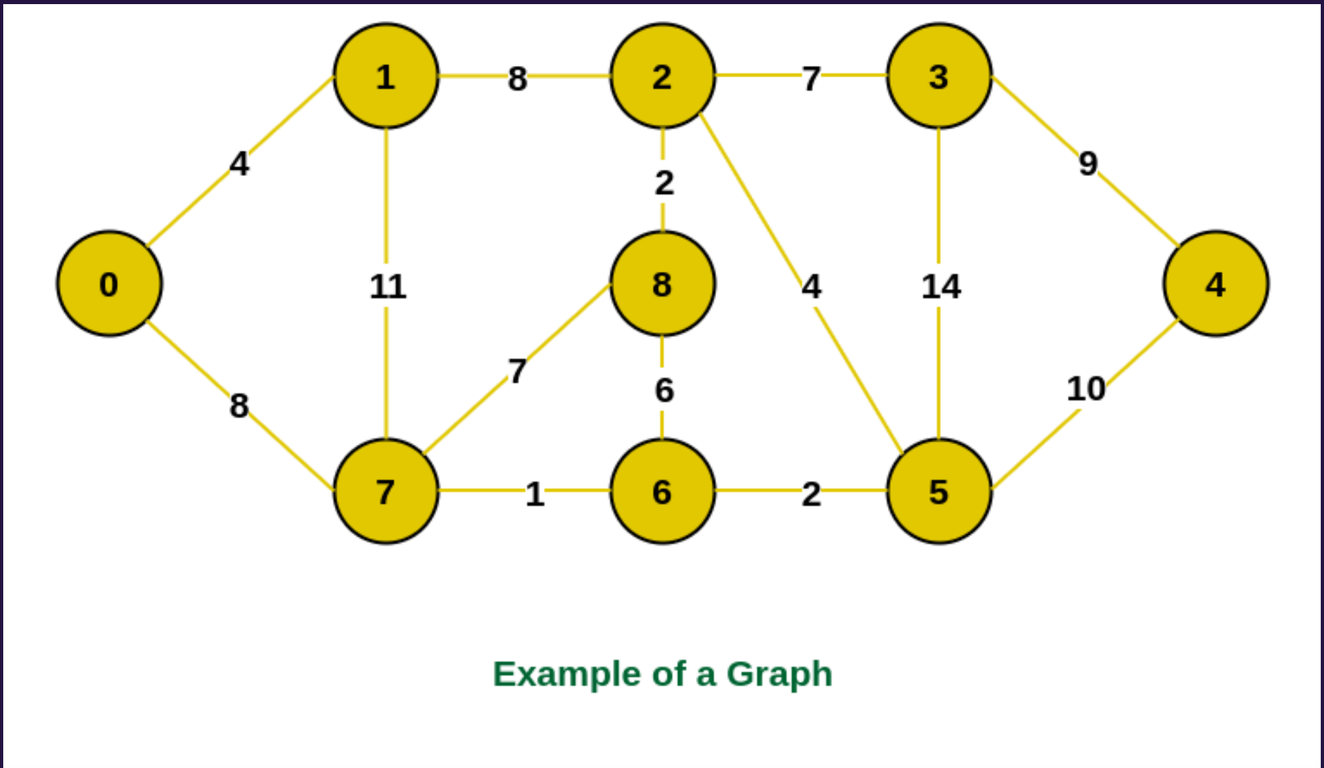
PRIM

PRIM

```
Prim(s):
1    Q ← cola de prioridades vacía (Min Heap)
2    T ← lista vacía
3    for u ∈ V − {s} :
4        d[u] ← ∞;  π[u] ← ∅;  Insert(Q, u)
5    d[s] ← 0;  π[s] ← ∅;  Insert(Q, s)
6    while Q no está vacía :
7        u ← Extract(Q)
8        T ← T ∪ {(π[u], u)}
9        for v ∈ α[u] :
10           if v ∈ Q :
11               if d[v] > cost(u, v) :
12                   d[v] ← cost(u, v);  π[v] ← u
13                   DecreaseKey(Q, v, d[v])
14   return T
```

Q usa como prioridad el valor $d[v]$.
$\texttt{DecreaseKey}(Q, v, k)$ cambia la prioridad del elemento v

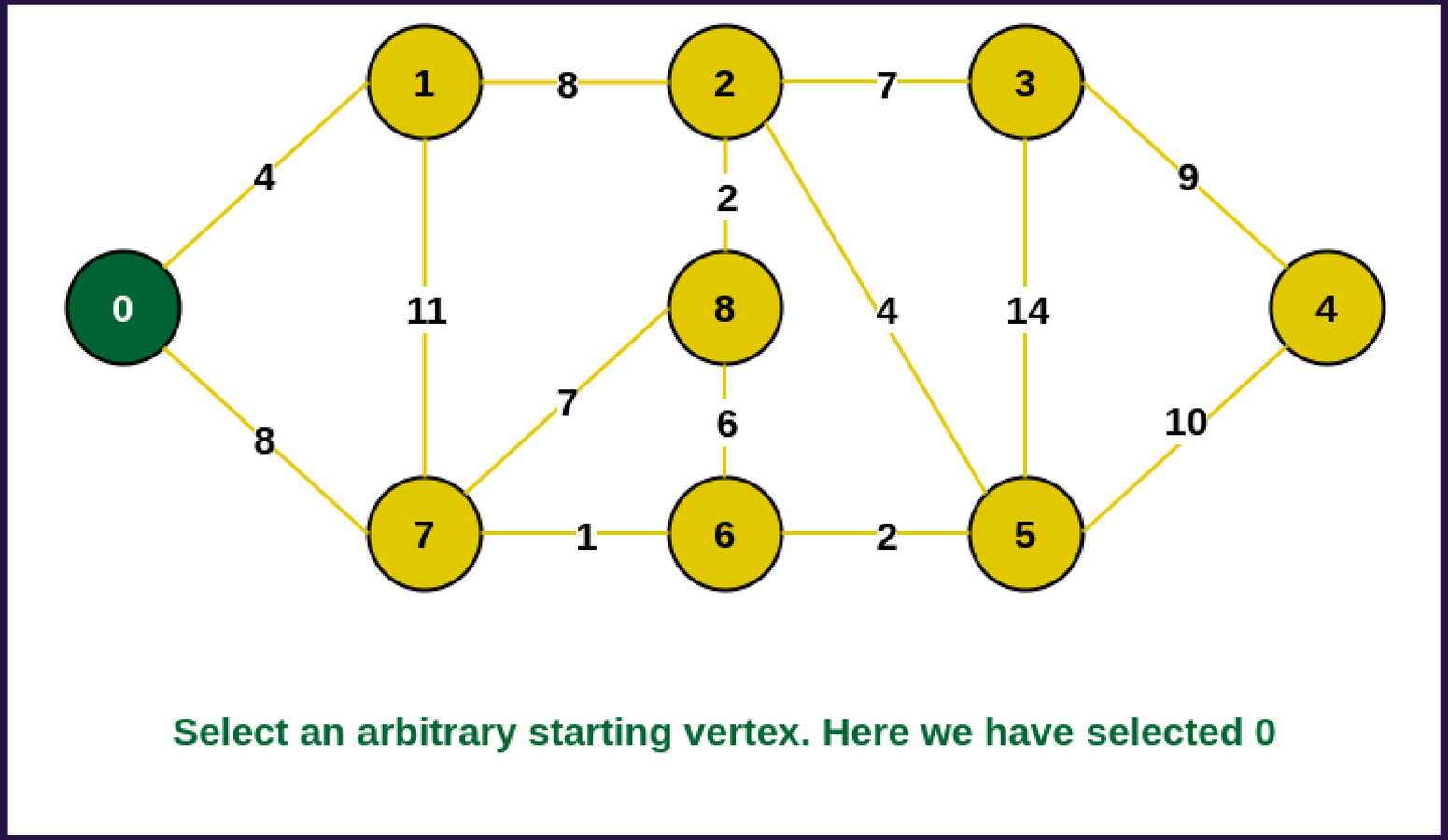
EJEMPLO



R = { }
R'= {0, 1, 2, 3, 4, 5, 6, 7, 8}

Fuente: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

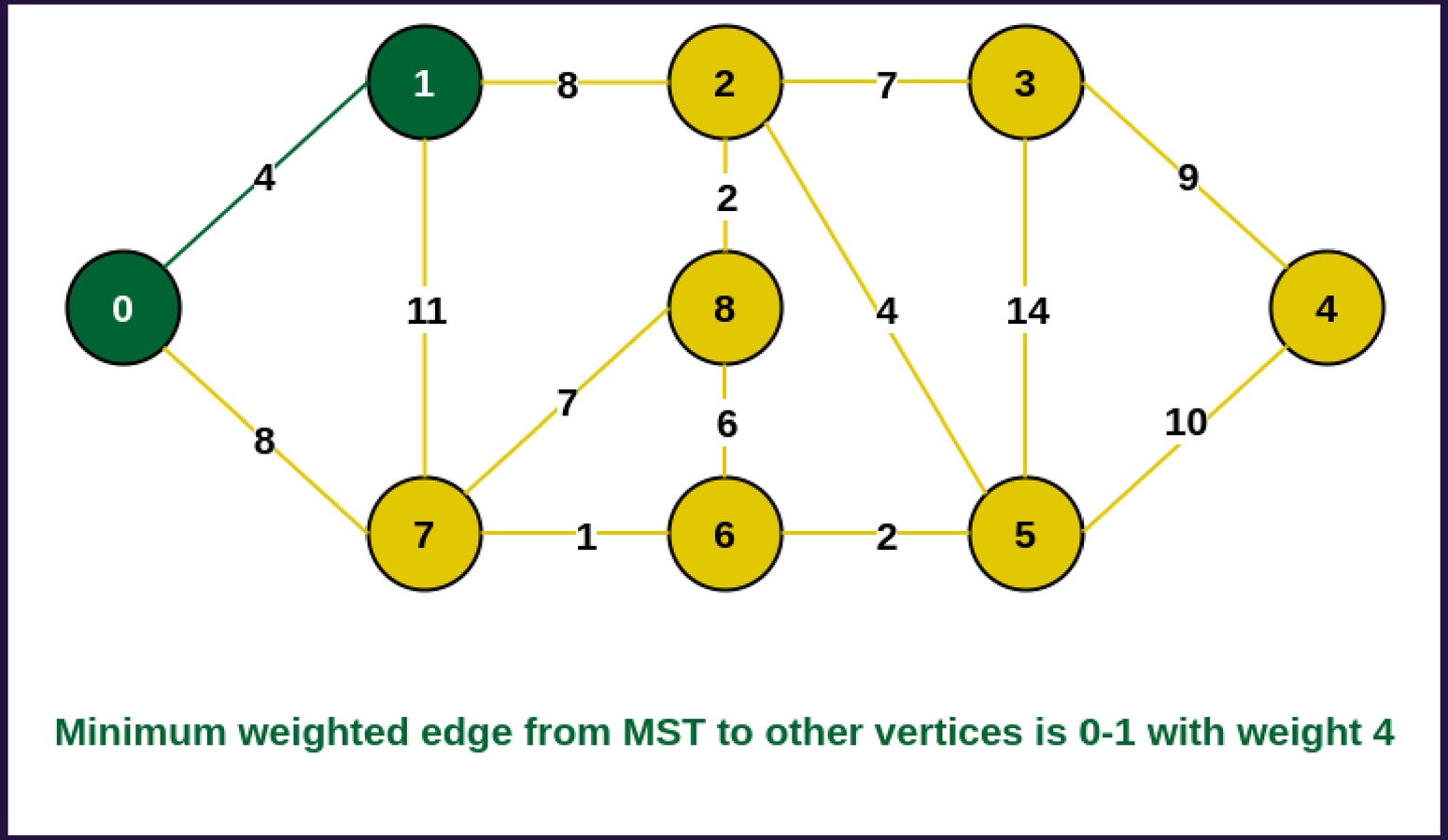
Escogemos arbitrariamente el nodo 0



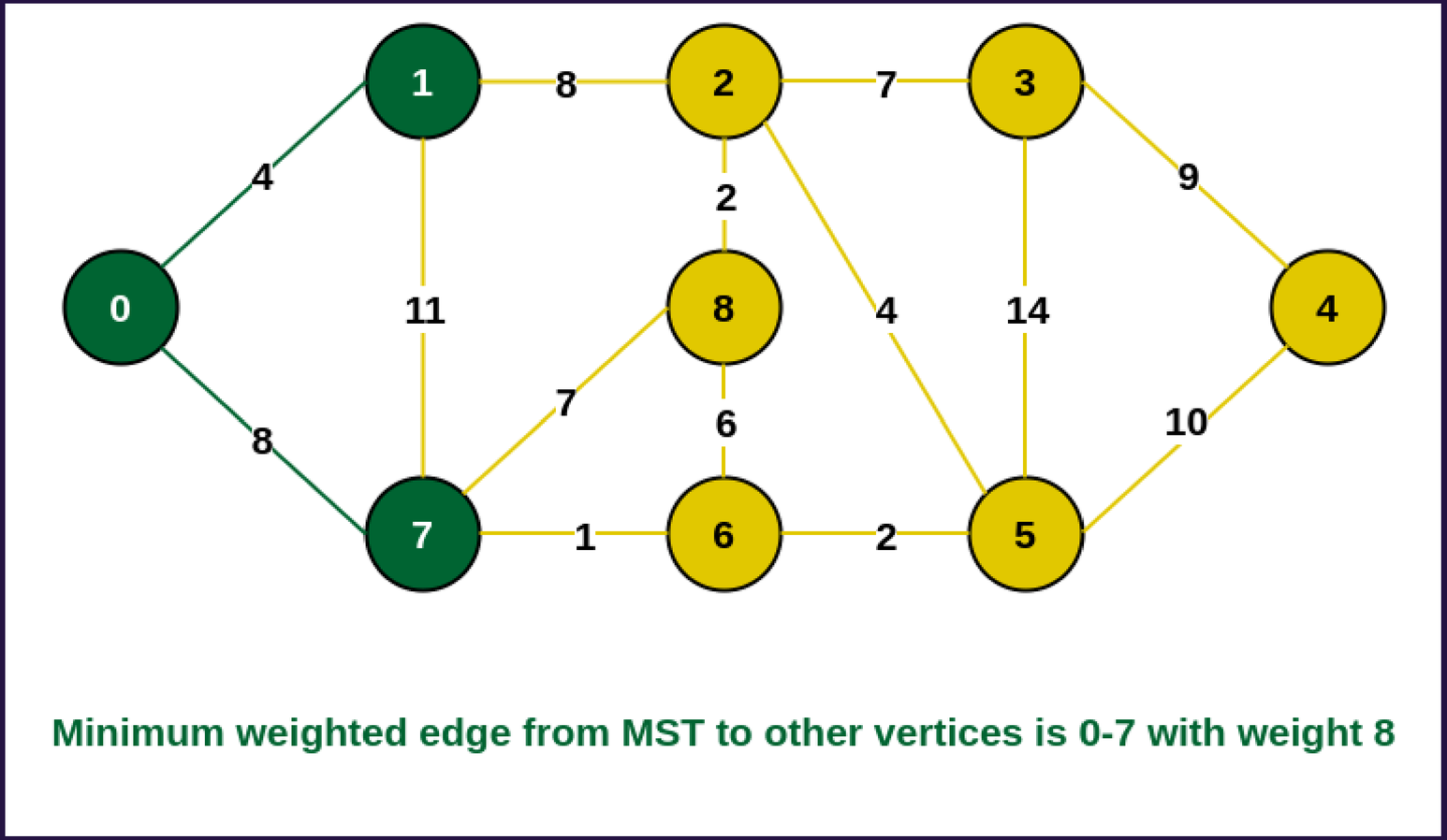
R = {0}

R'= {1, 2, 3, 4, 5, 6, 7, 8}

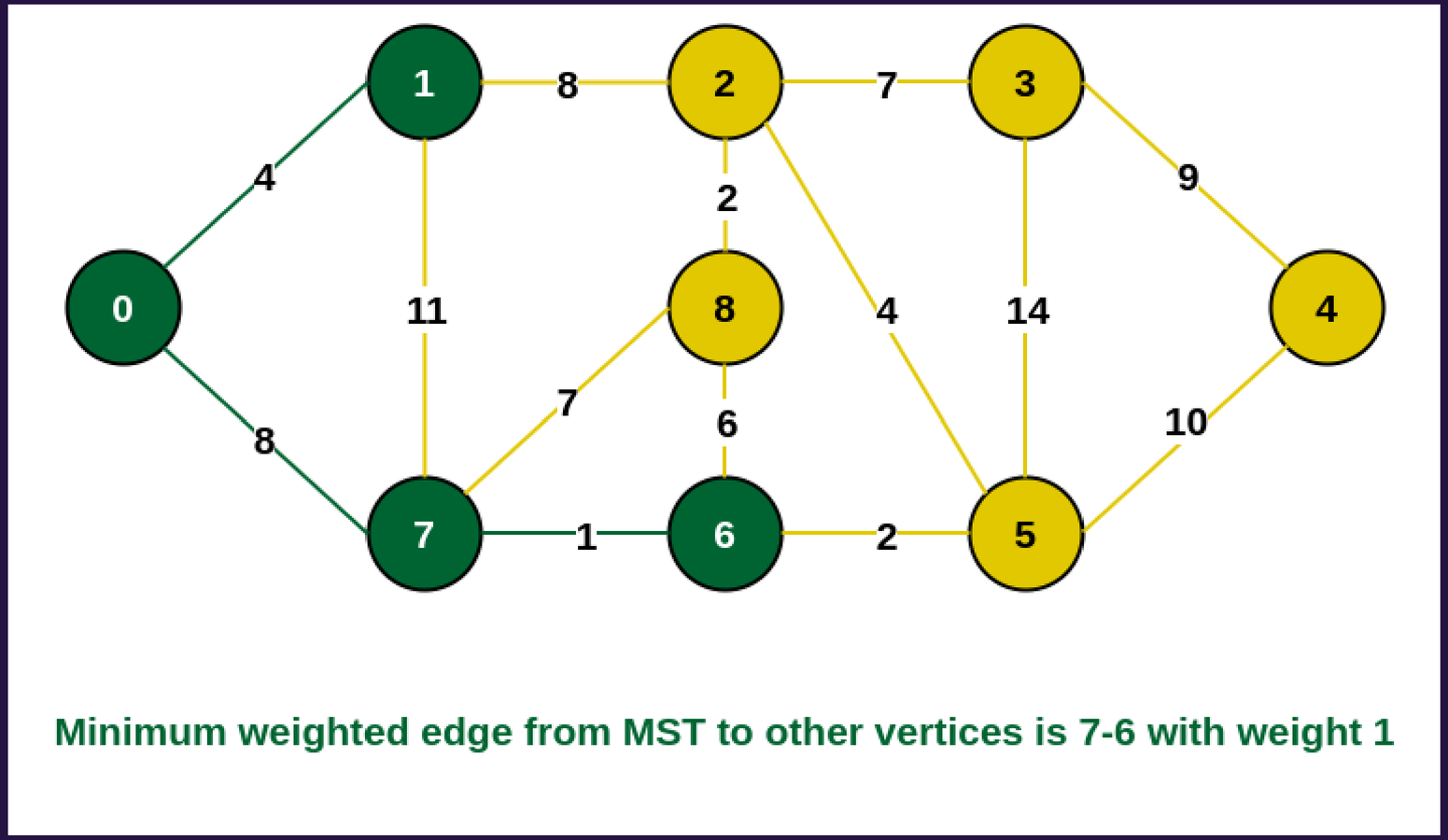
De todas las aristas disponibles que van desde un nodo visitado a un nodo no visitado, escogemos la de menor costo



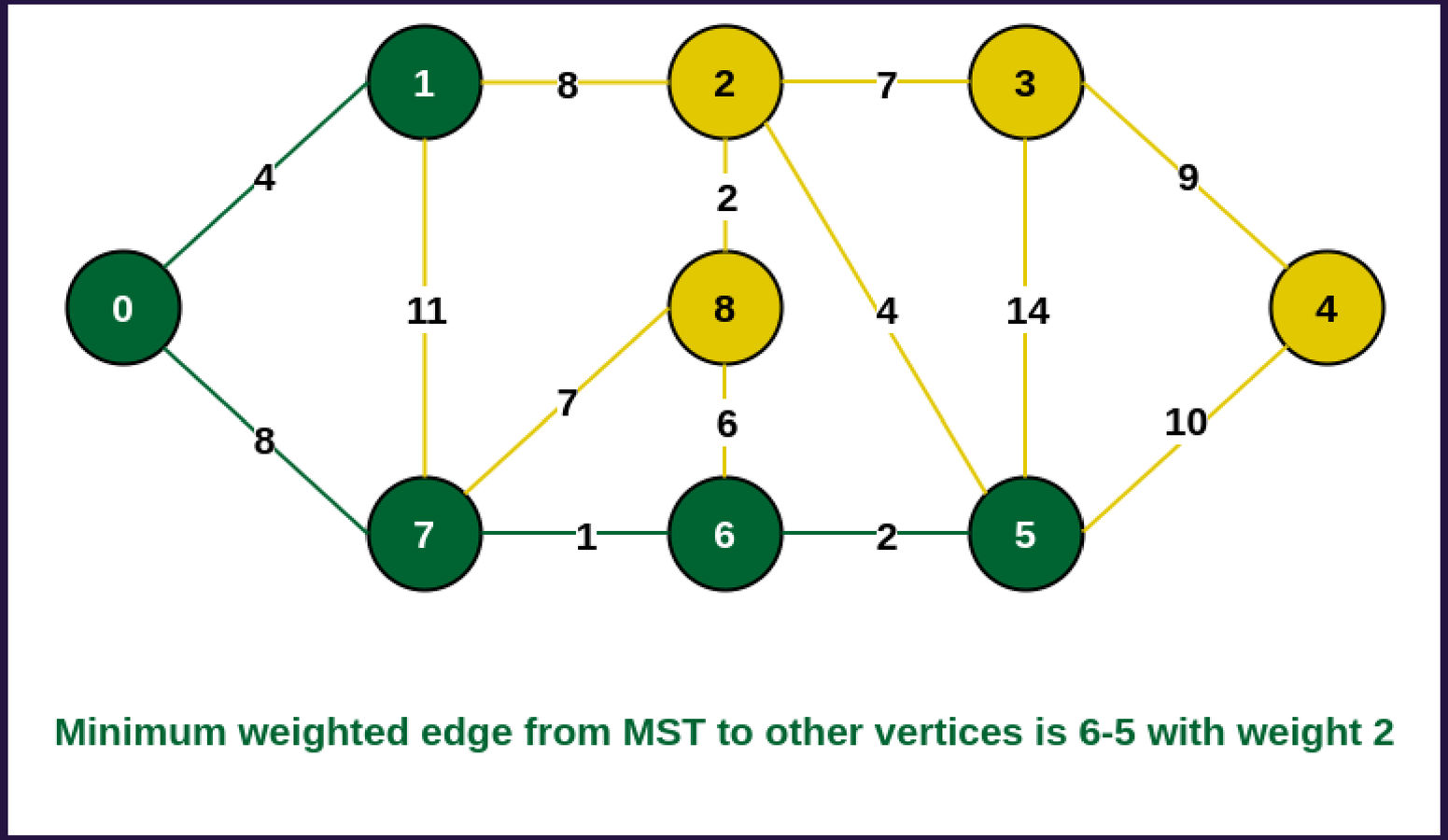
R = {0, 1}
R'= {2, 3, 4, 5, 6, 7, 8}



R = {0, 1, 7}
R'= {2, 3, 4, 5, 6, 8}

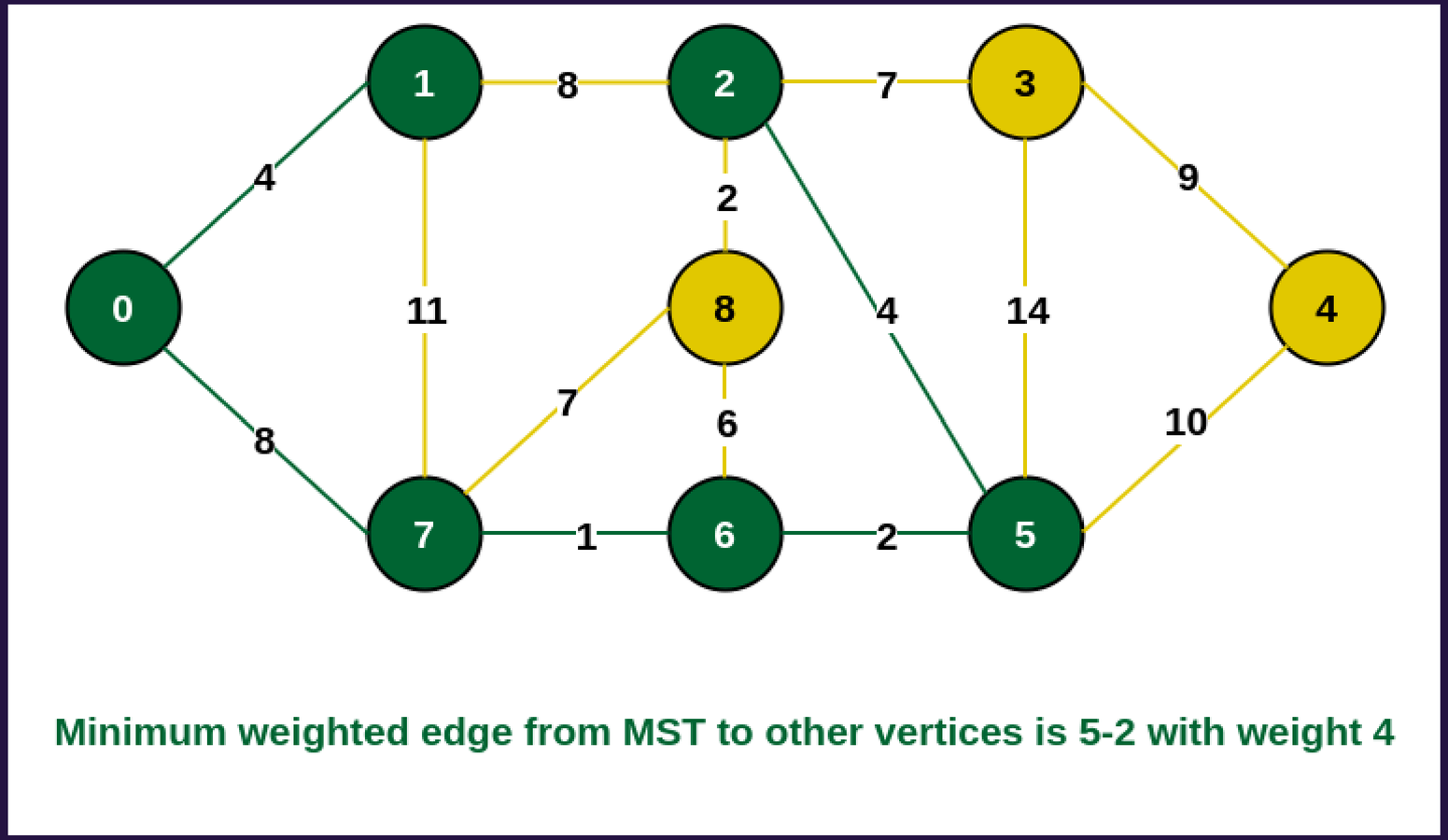


R = {0, 1, 7, 6}
R'= {2, 3, 4, 5, 8}

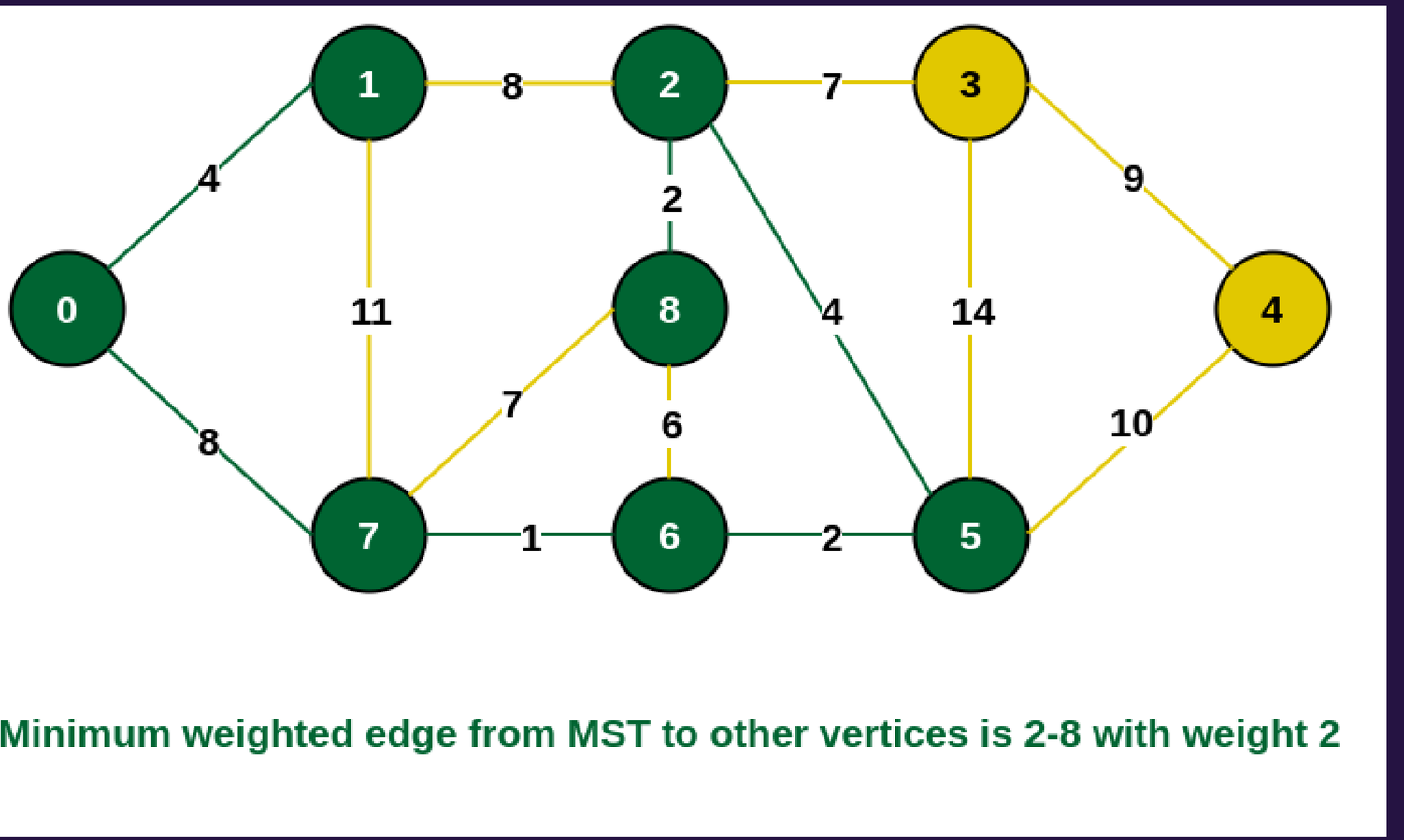


R = {0, 1, 7, 6, 5}

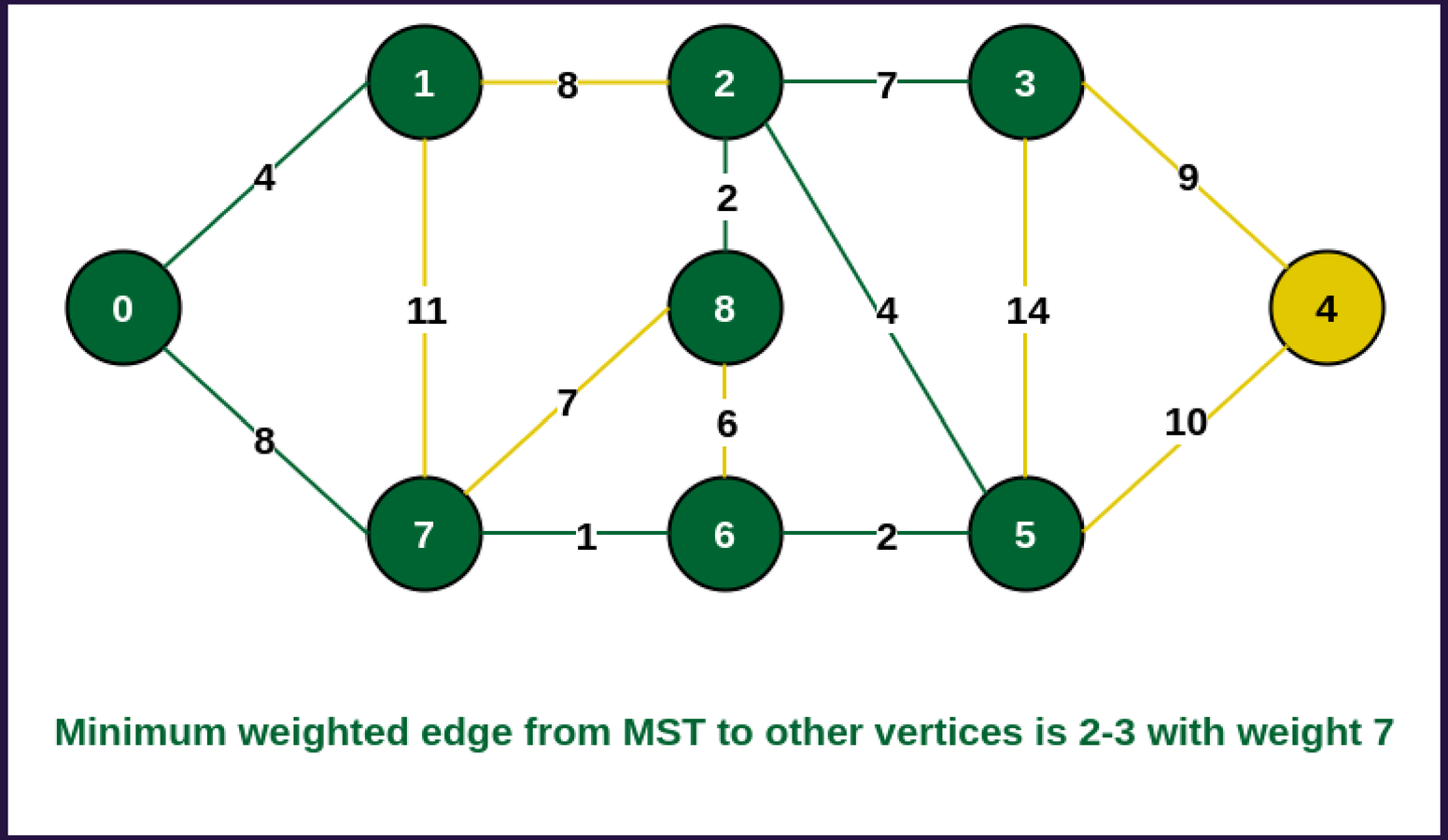
R'= {2, 3, 4, 8}



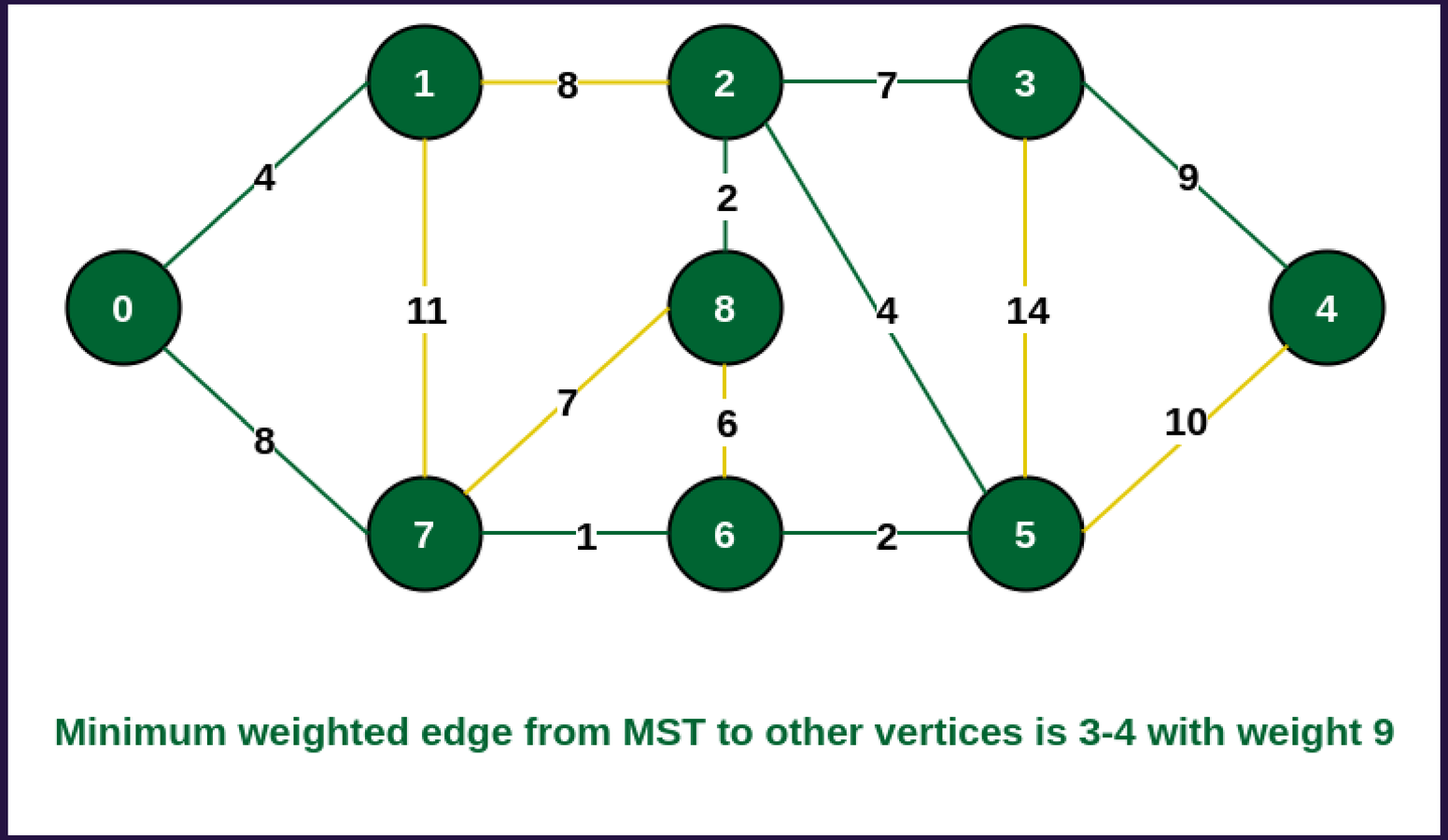
R = {0, 1, 7, 6, 5, 2}
R'= {3, 4, 8}



R = {0, 1, 7, 6, 5, 2, 8}
R'= {3, 4}



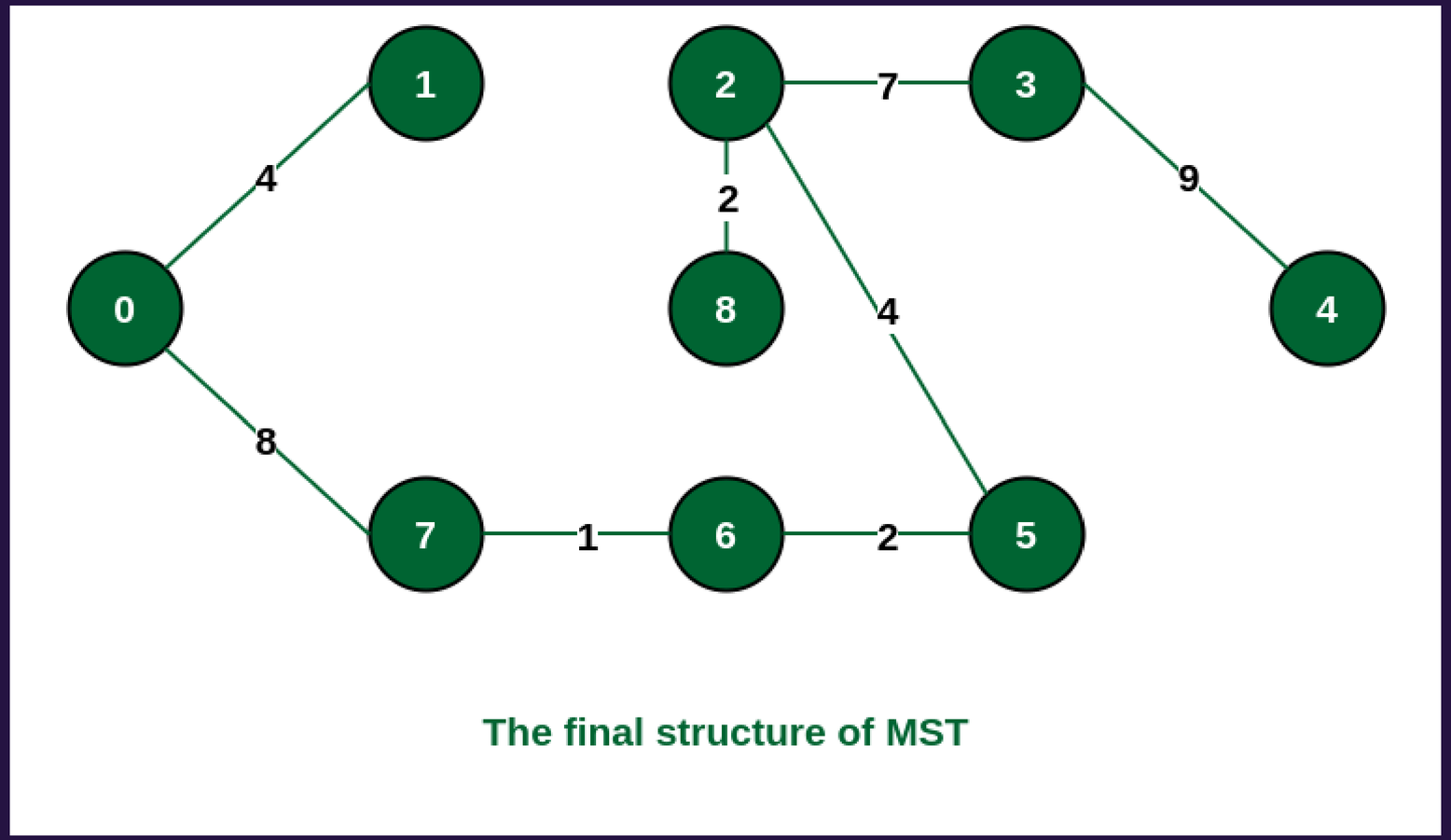
R = {0, 1, 7, 6, 5, 2, 8, 3}
R'= {4}



R = {0, 1, 7, 6, 5, 2, 8, 3, 4}

R'= { }

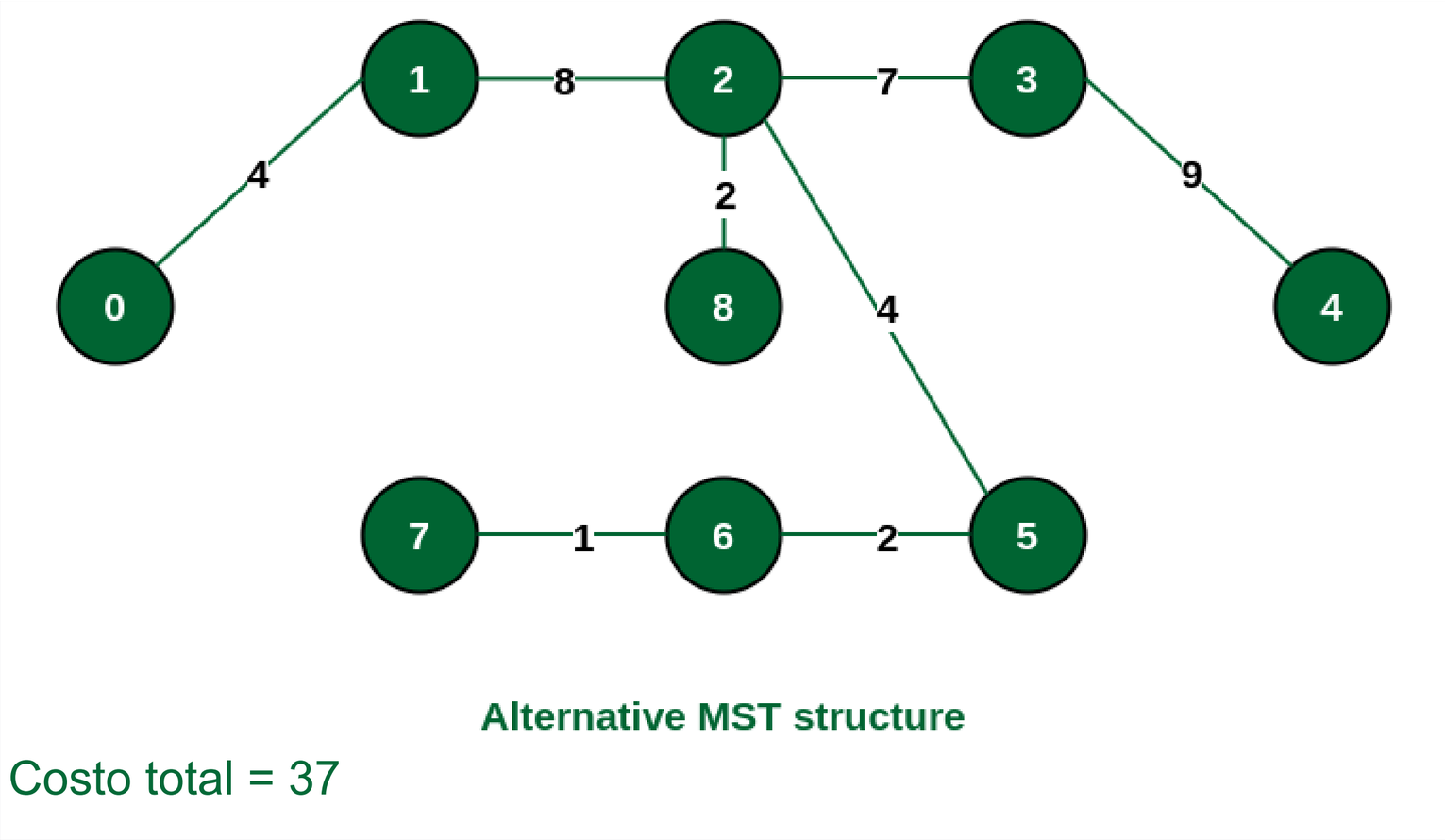
MST, hemos minimizado los costos totales!



Costo total anterior = 93

Nuevo Costo total = 37

MST Alternativo



Alternative MST structure

Costo total = 37

Por definición de MST, no puede existir otro árbol con menor costo total, pero sí puede existir otro con el mismo costo total!

HEAPS

COLA DE PRIORIDAD

- Una estructura de datos que nos permite almacenar datos según su prioridad (dada por un valor). Por ejemplo, pruebas para las que hay que estudiar o urgencia de atención de una de una persona en una sala de espera.

- Un criterio de prioridad: Orden de llegada (FIFO, LIFO). Importa la posición en la cola

- Podría interesarnos otro criterio, pero siempre nos interesará mantener estas funciones:

1. Insertar un dato con prioridad dada
2. Extraer el dato con mayor prioridad
3. Idealmente, cambiar la prioridad de un dato

HAY UNA PREGUNTA IMPORTANTE QUE TENEMOS QUE HACERNOS ACÁ

EN QUÉ ESTRUCTURA ALMACENAREMOS NUESTRA COLA DE PRIORIDADES

ARRAY VS LISTA LIGADA

Ganador: ARRAY

¿Por qué?

Hay casos donde usar una lista ligada nos dará problemas

Buscar el máximo valor → O(n)

Insertar en la posición correcta/mantener orden → O(n) } caro

Usar arrays nos limitará la cantidad de elementos!

Pero logramos tener accesos en o(1)

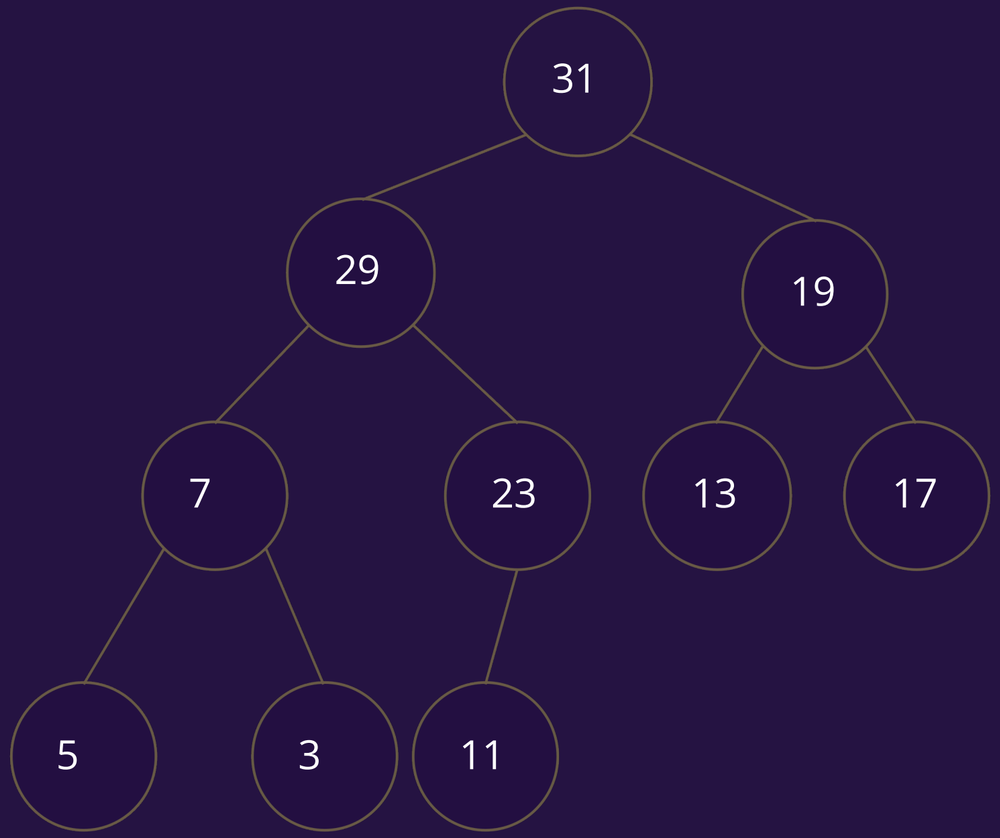
Para las colas de prioridad usaremos Heaps

¿Qué es un Heap?

- Un heap es un árbol binario que nos permite mantener un **orden de prioridad de los elementos** de un conjunto. La prioridad la podemos determinar según un MIN-Heap o un MAX-heap A medida que bajamos de nivel, los nodos hijos tendrán menor prioridad que el padre. Entre hermanos no hay ninguna restricción

- Para nuestro objetivo de extraer e insertar de forma eficiente, no necesitamos un orden estricto. Basta tener un orden entre subsectores

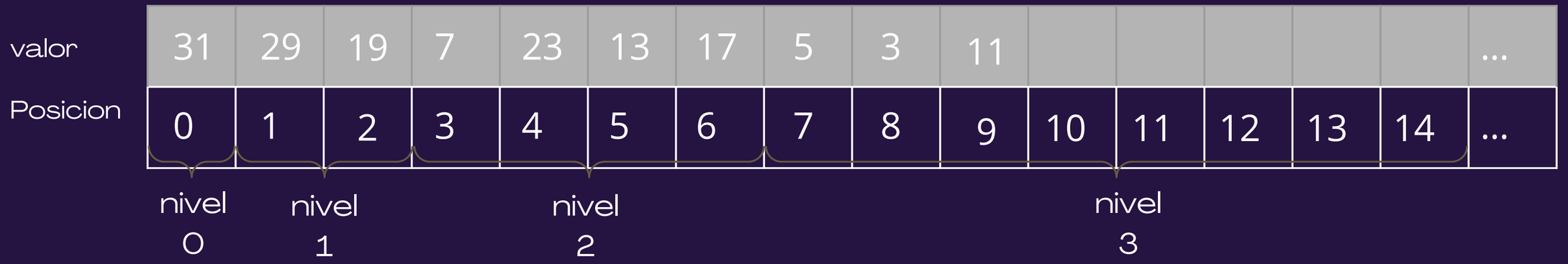
Ejemplo de Heap



valor	31	29	19	7	23	13	17	5	3	11						...
posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Al ir llenando el árbol por nivel ganamos:

- Minimizar la altura del árbol y compactar en
- el array Una relación entre padres y hijos:
    - El elemento H[k] es padre de H[2k + 1] y H[2k + 2]
    - El padre del elemento H[k] es H[⌊(k − 1)/2⌋]
- agrupar los niveles



A lo que vinimos... extracción e inserción eficiente

- Efectuamos la operación manteniendo un árbol binario casi-lleno
- Restablecemos las propiedades de heap

Extracción eficiente

```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

$O(?)$

Extracción eficiente

```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```

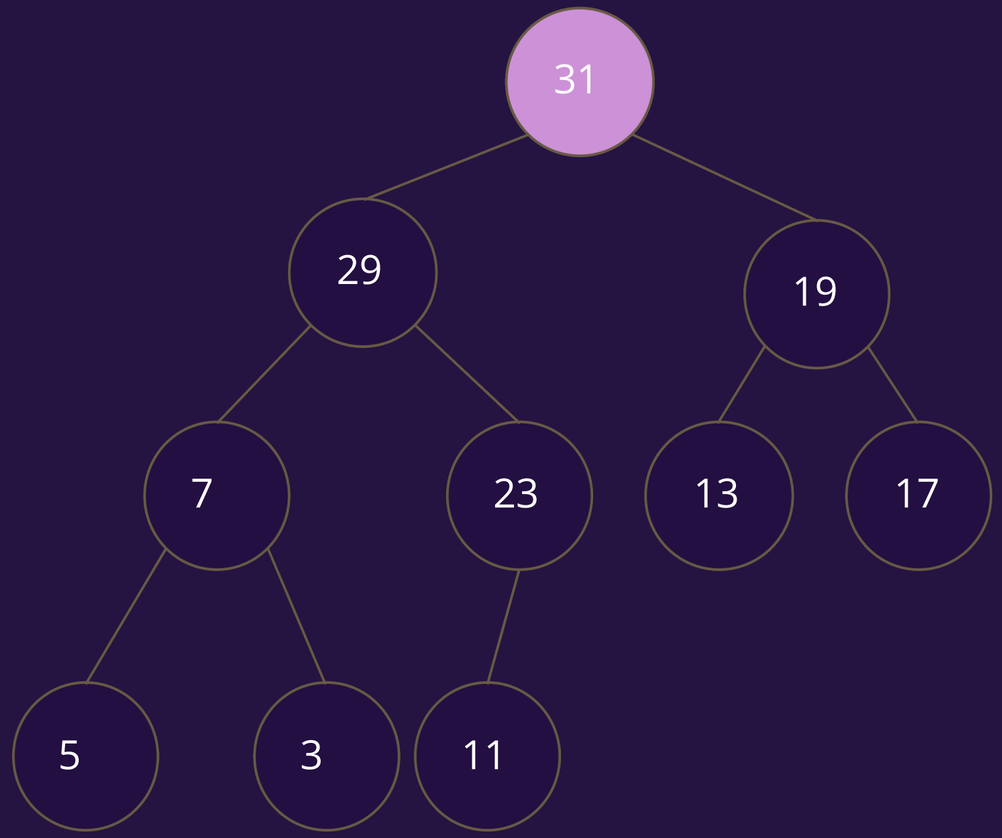
```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

O(log(n))

Extracción eficiente

Queremos sacar el elemento más prioritario

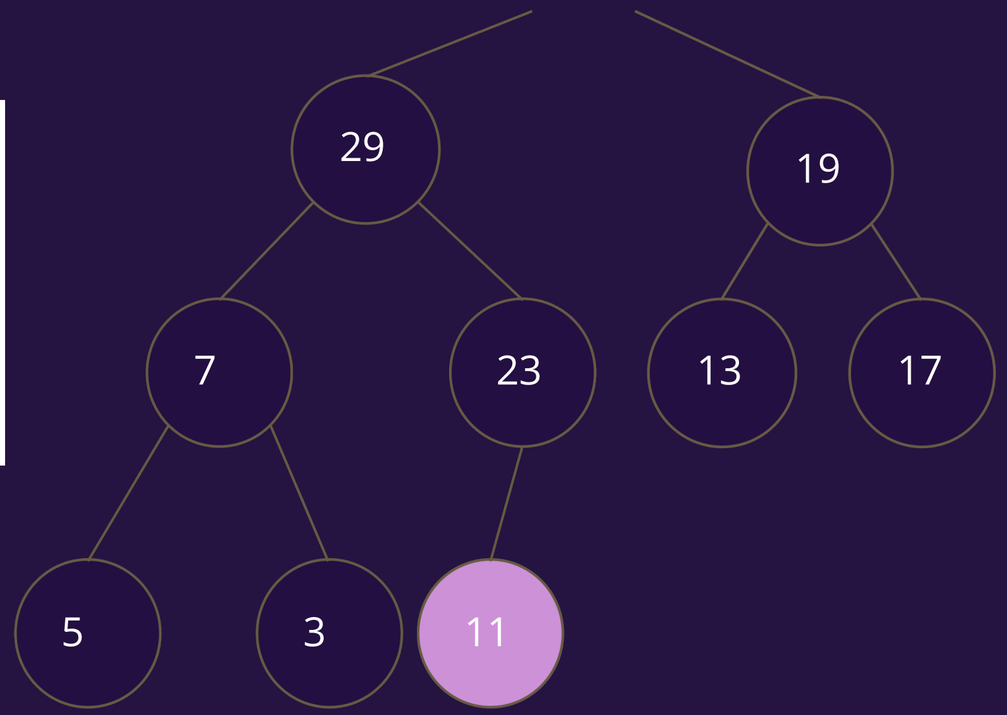
```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)
    return best
```



valor	31	29	19	7	23	13	17	5	3	11					...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Extracción eficiente

```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H,0)
    return best
```



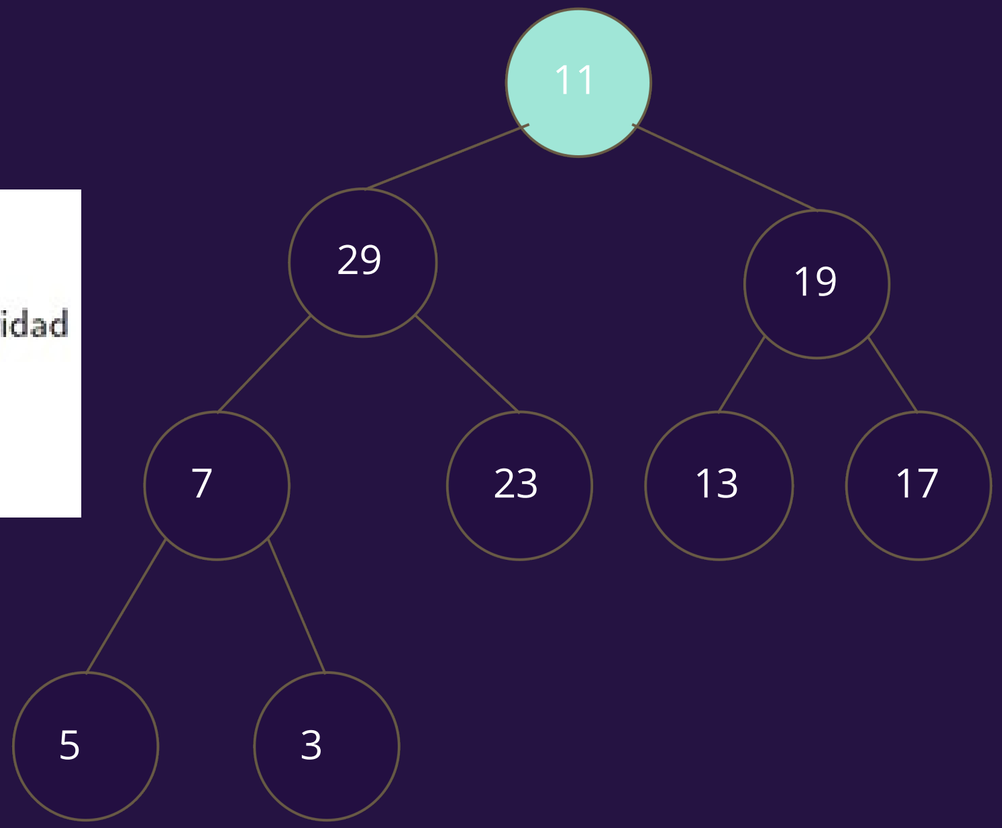
31

Extraemos

valor		29	19	7	23	13	17	5	3	11						...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Extracción eficiente

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```

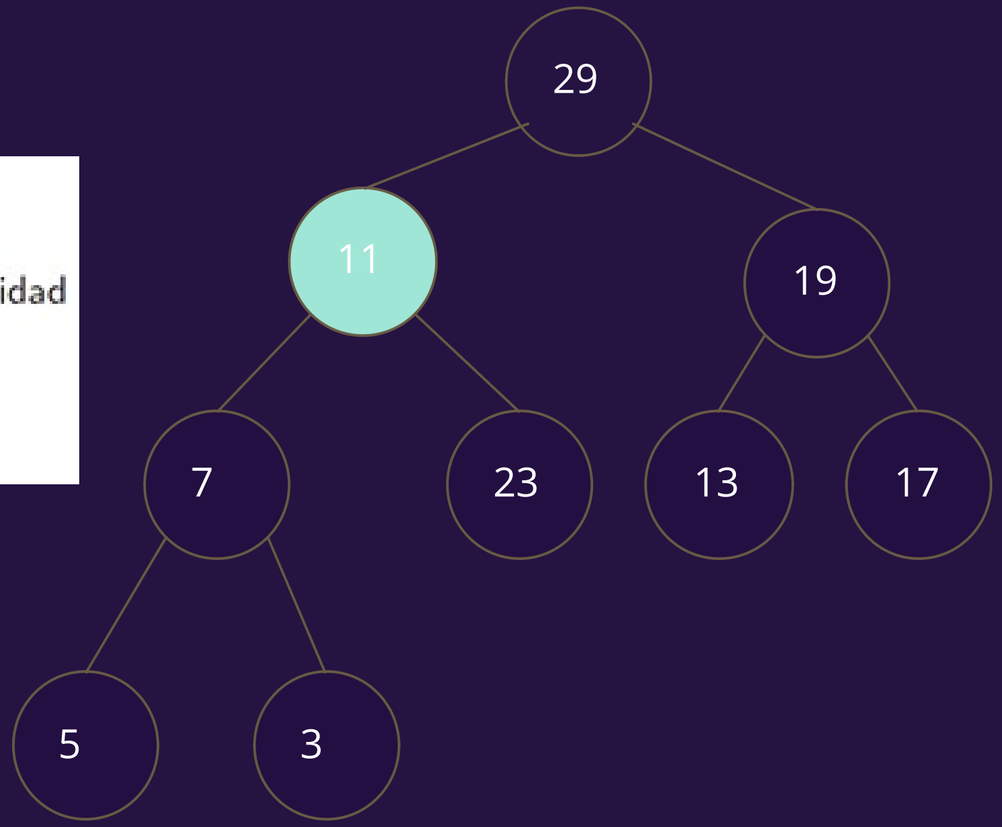


Empezamos con el ShiftDown para reacomodar

valor	11	29	19	7	23	13	17	5	3						...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Extracción eficiente

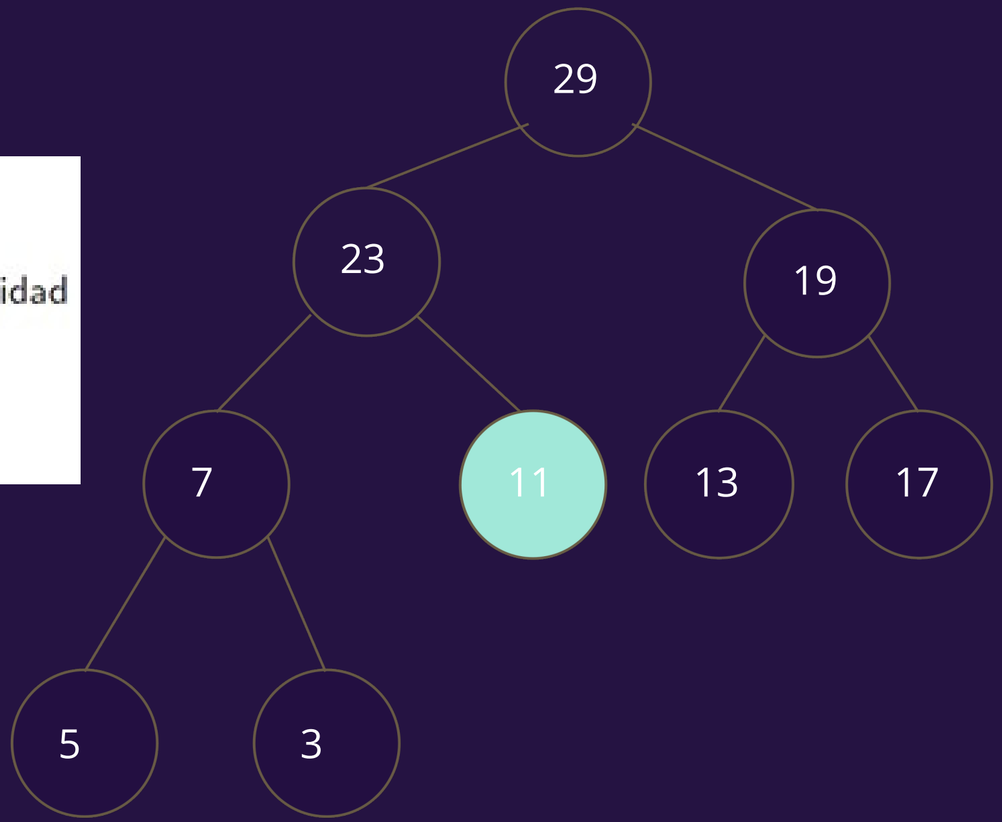
```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```



valor	29	11	19	7	23	13	17	5	3						...
Posición	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Extracción eficiente

```
SiftDown(H, i):
    if i tiene hijos :
        j ← hijo de i con mayor prioridad
        if H[j] > H[i] :
            H[j] ⇋ H[i]
            SiftDown(H, j)
```



valor	29	23	19	7	11	13	17	5	3						...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Inserción eficiente

```
Insert(H):
    i ← primera celda vacía de H
    H[i] ← e
    SiftUp(H, i)
```

```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```

$O(?)$

Inserción eficiente

```
Insert(H):
    i ← primera celda vacía de H
    H[i] ← e
    SiftUp(H, i)
```

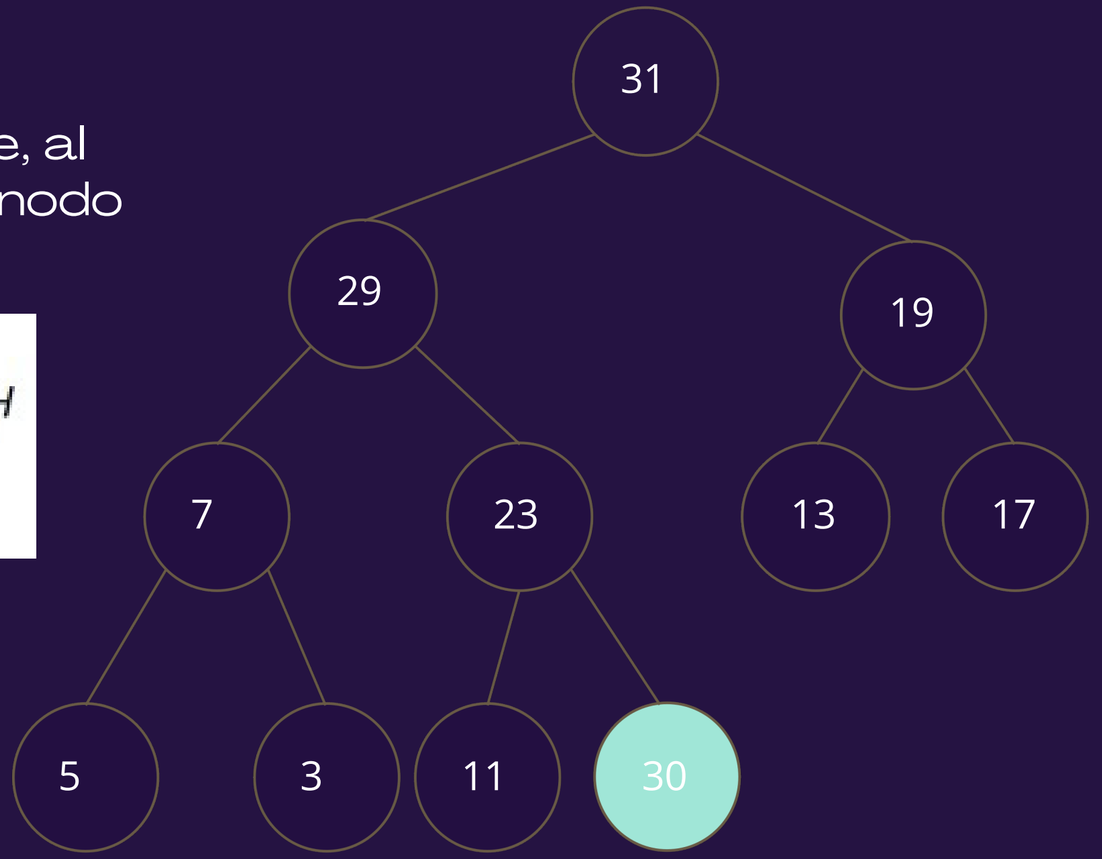
```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```

O(log(n))

Inserción eficiente

Insertamos, inicialmente, al final del array al nuevo nodo

```
Insert(H):
    i ← primera celda vacía de H
    H[i] ← e
    SiftUp(H, i)
```

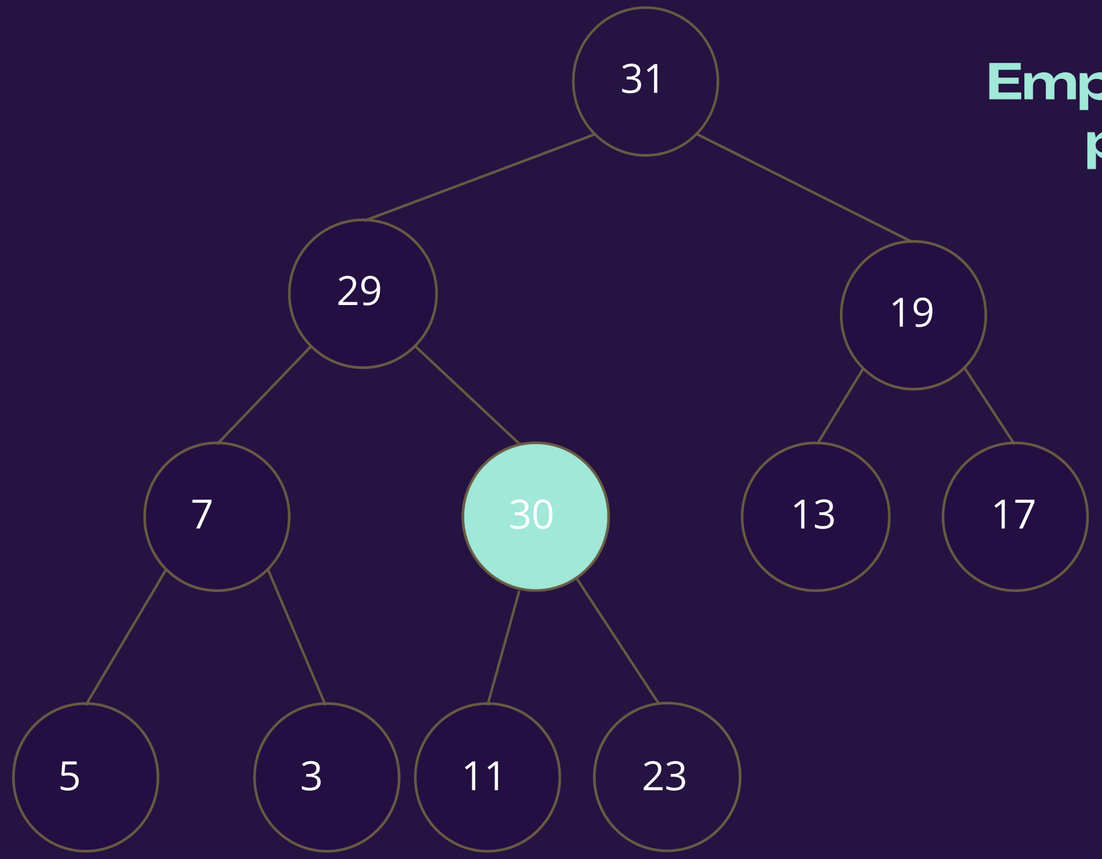


valor	31	29	19	7	23	13	17	5	3	11	30				...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Inserción eficiente

Empezamos a subir el nodo

```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```



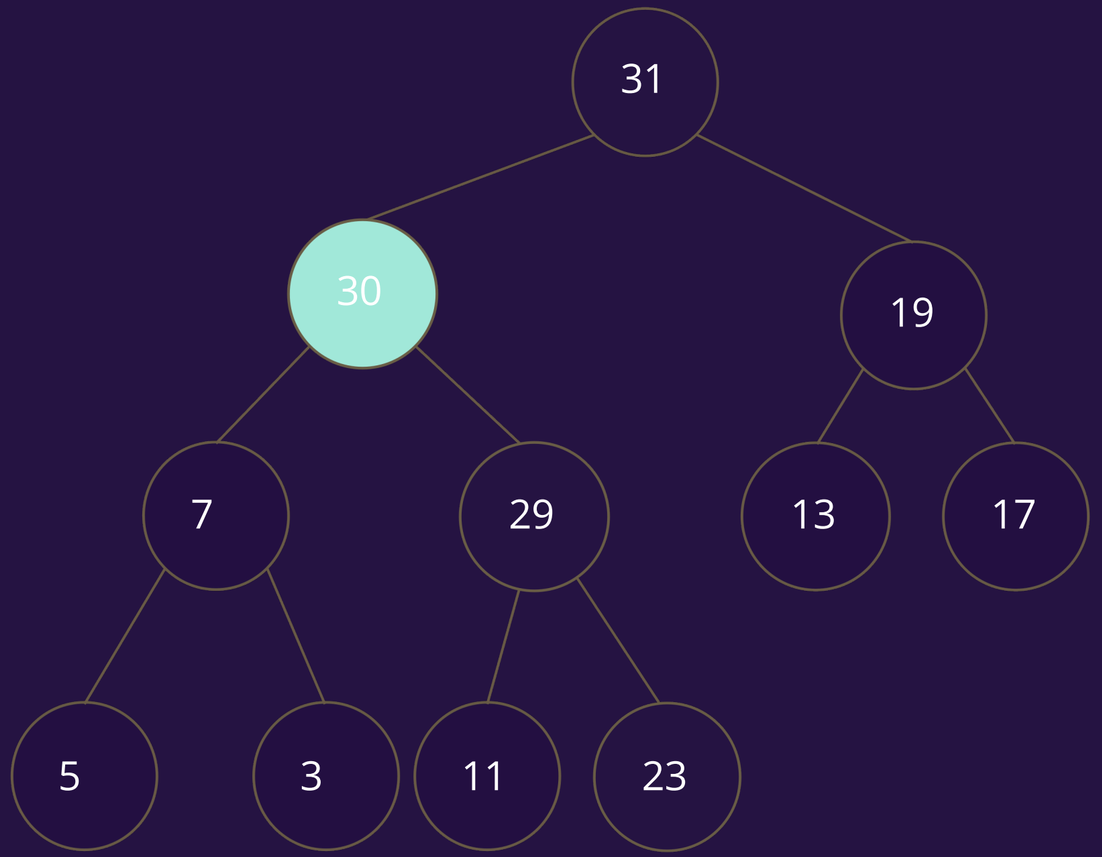
Empezamos con el ShiftUp para r eac omo d ar

valor	31	29	19	7	30	13	17	5	3	11	23					...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

Inserción eficiente

Empezamos a subir el nodo

```
SiftUp(H, i):
    if i tiene padre :
        j ← ⌊i/2⌋
        if H[j] < H[i] :
            H[j] ⇋ H[i]
            SiftUp(H, j)
```



valor	31	30	19	7	29	13	17	5	3	11	23				...
Posicion	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	...

¿Qué ganamos?

	Con una lista ligada	Con un array
Extracción del elemento más prioritario	O(n)	O(1)
Insertar manteniendo orden	O(n)	O (log(n))

¿Y la actualización de valores?

¿Qué pasa si actualizamos el valor?

Hay 3 casos

1. El nodo se actualiza con un valor que no supera al padre ni queda por debajo de alguno de los hijos.
   → No pasa nada
2. El nodo se actualiza con un valor que supera al del padre
   → Hay que subir el nodo
3. El nodo se actualiza con un valor que lo hace quedar por debajo de alguno de sus hijos
   → Hay que bajar el nodo y cambiarlo por ese hijo

input: Heap, índice del nodo a actualizar, nuevo valor del nodo

`Update`(H, i, v):

H[i] = v

if H tiene padre **and** H[i] > H[⌊(i−1)/2⌋]:

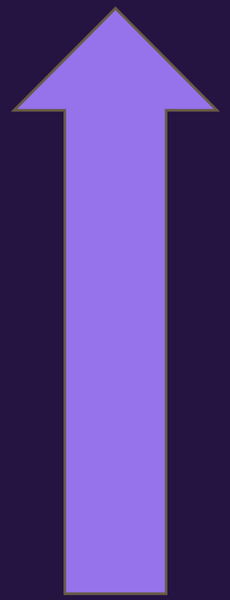
`ShiftUp` (H, i)

else if H tiene hijos **and** H tiene menor prioridad que alguno de ellos:

`Shiftdown` (H, i)

Max Heap

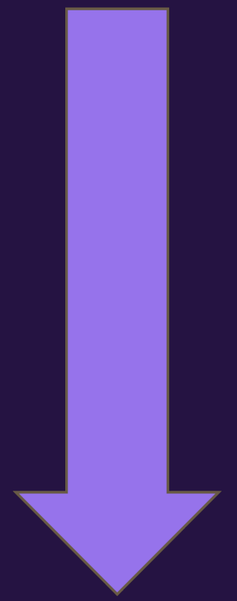
Valor
mayor



Valor
menor

Min Heap

Valor
menor



Valor
mayor

Ejercicio

I3 2023-1 P1

La atención al afiliado en la sucursal de una Isapre requiere que, para ser atendidos, éstos deben registrar su RUT y el tipo de atención solicitada (venta de bonos, reembolsos, pago de cotizaciones, cambio de planes, etc.) en un tótem de auto-atención. Este entrega un número secuencial ascendente, los que son invocados en orden por los ejecutivos de la sucursal al estar disponibles para atender al afiliado.

(a) El tótem utiliza la función **new**(**A, b, tipo, sec**) para registrar en A los datos del afiliado b que solicita atención con secuencia sec entregada por el tótem. El sistema utiliza la llamada b = **next**(**A**) para sacar de A al siguiente afiliado b a atender según la secuencia. Escriba el pseudocódigo de ambas funciones e indique explícitamente qué estructura de datos utiliza para representar A.

I3 2023-1 P1

(a) El tótem utiliza la función new(A, b, tipo, sec) para registrar en A los datos del afiliado b que solicita atención con secuencia sec entregada por el tótem. El sistema utiliza la llamada b = next(A) para sacar de A al siguiente afiliado b a atender según la secuencia. Escriba el pseudocódigo de ambas funciones e indique explícitamente qué estructura de datos utiliza para representar A.

Solución:

Utilizando como estructura una cola FIFO implementada como lista ligada, los métodos son:

```
new(A, b, tipo, sec):
    b.tipo ← tipo
    b.sec ← sec
    Insert(A, b)
```

```
next(A):
    return Remove(A, 0)
```

Donde Insert(A, b) agrega b al final de la lista ligada A, y Remove(A,0) retorna el primer elemento (cabeza) de la lista ligada A luego de extraerlo.

I3 2023-1 P1

(b) La gerente de la oficina desea mejorar la atención de los adultos mayores (> 65 años), considerando que al registrar al afiliado es posible incluir en b su edad (en meses según b.edad). Pide entonces promover que los adultos mayores sean atendidos primero, de mayor a menor edad, y luego los demás afiliados según la secuencia entregada por el tótem. Modifique el pseudocódigo de las funciones de (a) e indique explícitamente qué estructura de datos usa para **A**. Puede suponer que no hay dos afiliados con la misma edad en meses.

Pista: La secuencia del tótem es creciente no negativa

I3 2023-1 P1

(b) **Solución**

En este caso, se utiliza un MIN-heap donde la prioridad se almacena en el atributo **sec**. Para adultos mayores, este valor se reemplazará por su edad en negativo, de forma que el atributo sec por sí solo sirve como prioridad del MIN-heap. Con esto, los métodos quedan...

```
new(A,b,tipo,sec)
        b.tipo ← tipo

        if b.edad < 65 * 12:
            b.sec ← sec
        else:
            b.sec ←  -1 * b.edad
        Insert(A, b)

next(A):
        return Extract(A, 0)
```

donde **Extract(A)** es el método de extracción de raíz en MIN-heap.

I3 2023-1 P1

(**c**) Al medir el comportamiento de la mejora anterior se observa que el tiempo promedio de espera para ser atendido depende del tipo de atención de quienes llegaron antes, ya que algunas atenciones son rápidas (venta de bonos) y otras lentas (cambio de planes). Un consultor propone aplicar la estrategia SJF (Short Job First, Tarea Más Corta Primero) para privilegiar a los que requieren atenciones breves. Así, se puede usar un factor $0 \leq f \leq 1$ donde 1 representa la tarea más breve. Indique (no se requiere el pseudocódigo) qué debería modificar en su respuesta de (b) para incorporar este cambio.

I3 2023-1 P1

(c) **Solución:**

Se debe actualizar la forma en que se calcula la prioridad en el método new. Una estrategia es dividir la prioridad por el factor f de manera que las tareas más largas tienen mayores valores de prioridad luego del cambio.

PRIM
→ EJERCICIO PROPUESTO

I3-2023-2-P3

a) Considere un grafo no dirigido G y un árbol de cobertura de costo mínimo T para G. Suponga que se añade un conjunto de k nodos adicionales con ciertas aristas, resultando en un grafo conexo G'.

Explique cómo obtener un MST para G' a partir de T e indique la complejidad de su propuesta. No es necesario hacer un pseudocódigo.

a) Considere un grafo no dirigido G y un árbol de cobertura de costo mínimo T para G. Suponga que se añade un conjunto de k nodos adicionales con ciertas aristas, resultando en un grafo conexo G'.
Explique cómo obtener un MST para G' a partir de T e indique la complejidad de su propuesta. No es necesario hacer un pseudocódigo.

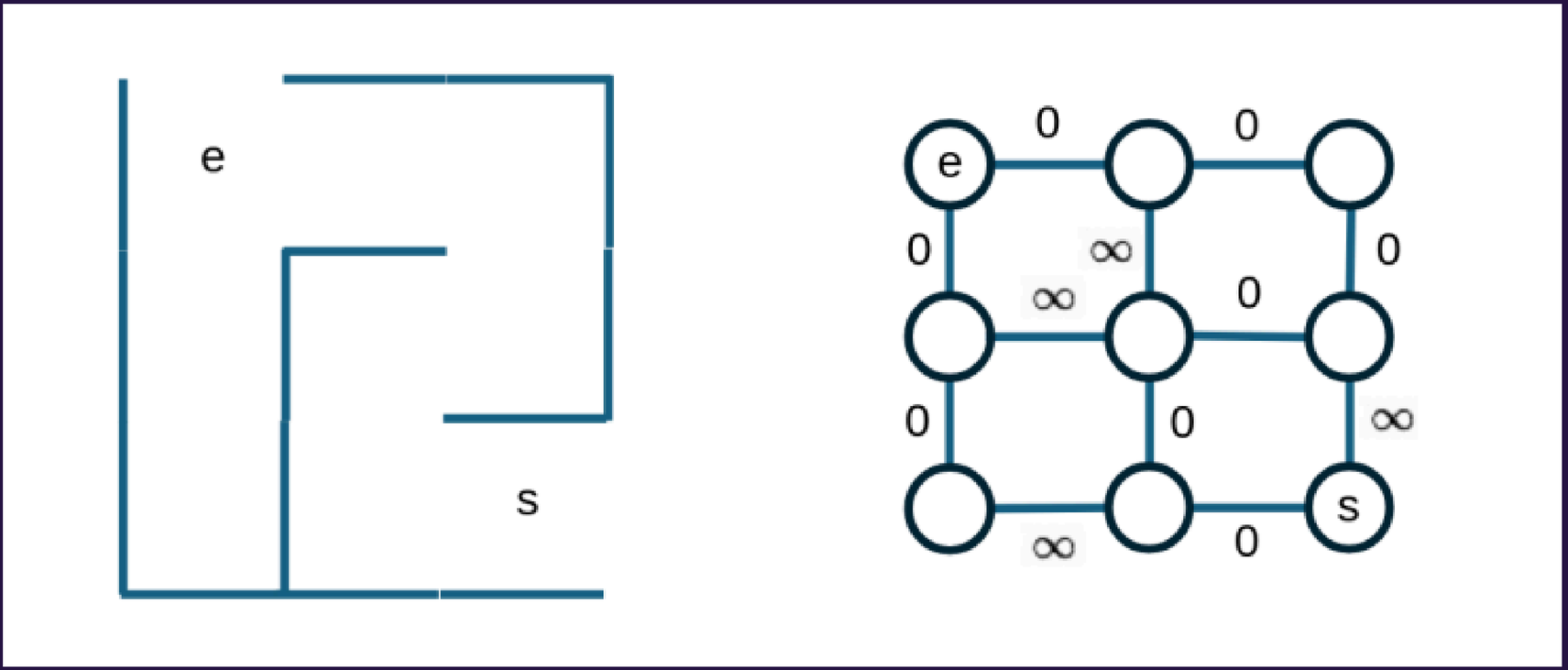
Sean
E' = conjunto de aristas nuevas para agregar a G

V' = conjunto de nodos nuevos para agregar a G

Para decidir qué aristas de E' se deben incluir para extender el árbol T, podemos usar el **algoritmo de Prim**, escogiendo la arista de menor costo que conecta con un **nodo no conectado previamente a T**.
Resumen: Continuar el proceso normal del algoritmo de Prim.

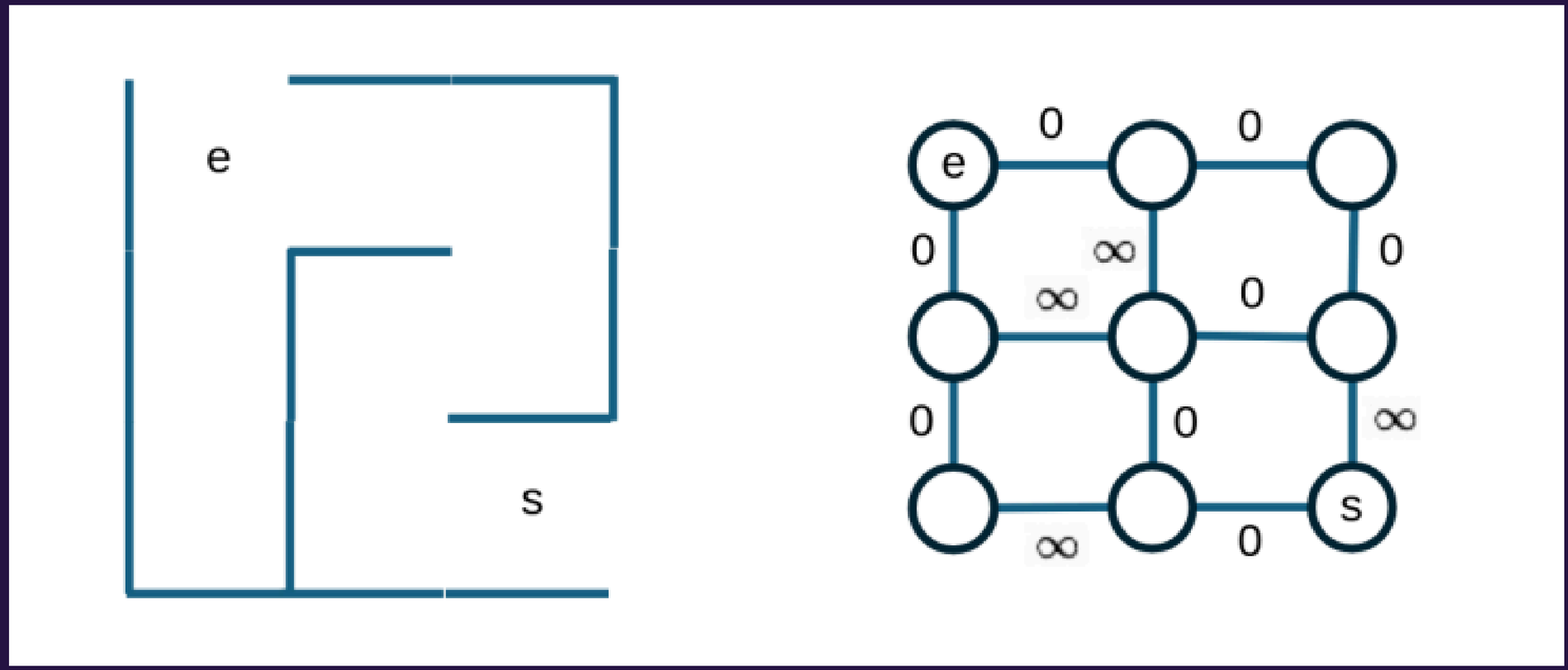
I3-2024-1-P2

Un laberinto se puede representar como un grafo conexo en que los nodos representan las celdas y las aristas conectan a las celdas con sus vecinas, dos celdas conectadas en el laberinto son unidas por una arista de costo 0 y un muro entre dos celdas es representado por una arista de costo ∞. Por ejemplo:



I3-2024-1-P2

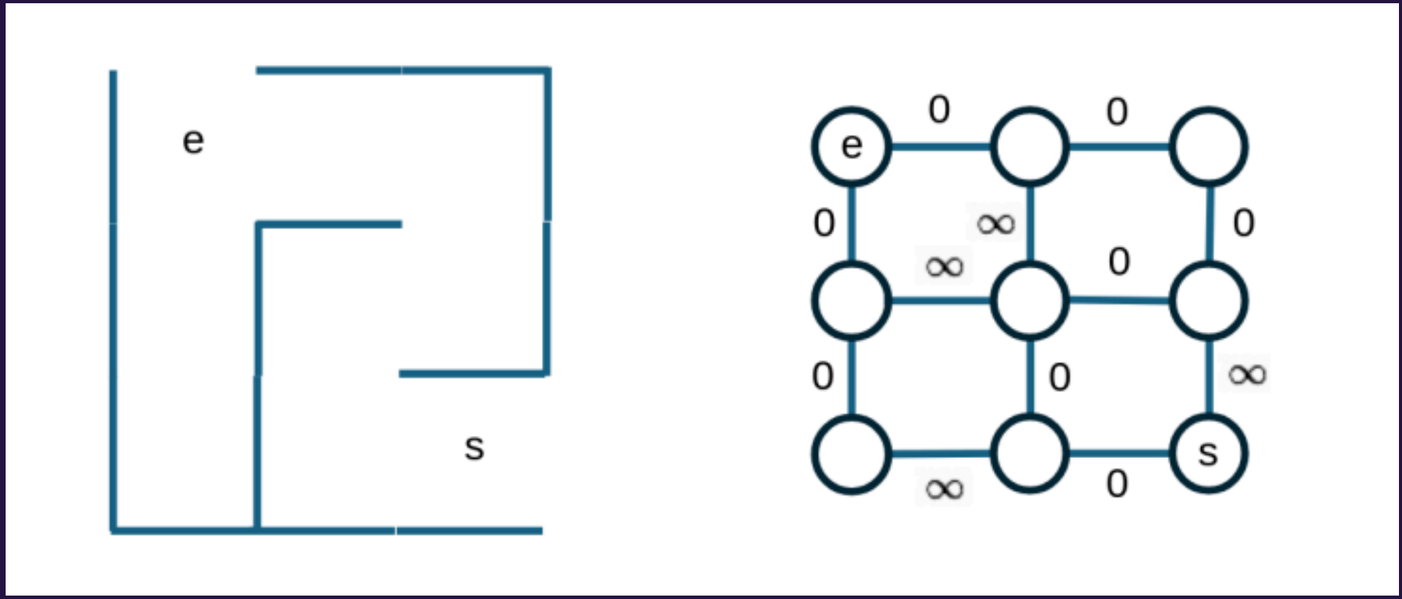
(a) En este escenario, ¿es posible utilizar el algoritmo de Prim para determinar el camino más corto desde un nodo de partida e hasta el nodo de salida s del laberinto? Argumente su respuesta.



I3-2024-1-P2

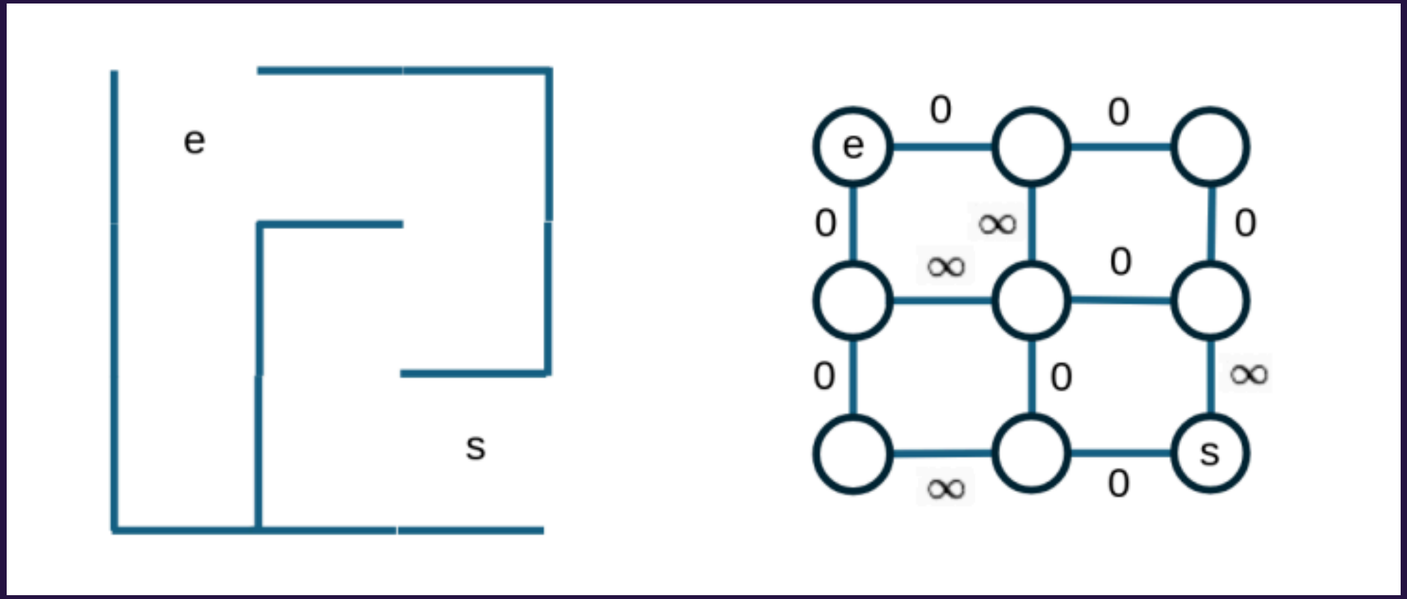
(a) En este escenario, ¿es posible utilizar el algoritmo de Prim para determinar el camino más corto desde un nodo de partida e hasta el nodo de salida s del laberinto? Argumente su respuesta.

Solución: Si, es posible. Dado que Prim permite determinar el MST de un grafo conexo, en particular este MST conecta a costo míinimo a e y s, utilizando las aristas de costo cero.



I3-2024-1-P2 (Propuesto)

b) Algunos juegos requieren crear laberintos de manera dinámica. Modifique el algoritmo de Prim (entregue el pseudocódigo) para crear un laberinto sin ciclos a partir de un grafo G con un nodo de partida e y un nodo de salida s. Hint: Puede suponer que inicialmente todas las celdas del grafo G están rodeadas por muros.



Solución Propuesto

Solución: La solución asume que el grafo tiene inicialmente todas las aristas de costo ∞.

```
   Prim(s):
1      Q ← cola de prioridades con s
2      T ← lista vacía
3      x.key ← 0;  x.parent ← ∅
4      while Q no está vacía :
5          u ←Extract(Q);  u.color ← negro
6          if u.parent ≠ ∅ then
7              T ← T ∪ {(u.parent, u)}
8          for v ∈ α[u] ∧ v.color ≠ negro :
9              if v ∉ Q then
10                 Insert(Q, v)
11                 v.key ← 0
12                 v.parent ← u
13                 DecreaseKey(Q, v, v.key)
14     return T
```

Los nodos en Q son aquellos que ya fueron conectados (linea 5). Luego en la línea 9 los que no estaban en Q son nodos que no generan ciclos, entonces los agregamos Q y botamos el muro que lo separa de u en la linea 11 y lo conectamos, ajustando su prioridad en el heap en la linea 13.

Algoritmo de Prim

d[x]: key del vértice x

vértice desde el que parte

```
Prim(s):
1    Q ← cola de prioridades vacía (Min Heap)
2    T ← lista vacía
3    for u ∈ V − {s} :                                  Inserta el resto de nodos.
4        d[u] ← ∞;  π[u] ← ∅;  Insert(Q, u)
5    d[s] ← 0;  π[s] ← ∅;  Insert(Q, s)                 vértice del que parte
6    while Q no está vacía :
7        u ← Extract(Q)                                 la raíz (mayor prioridad)
8        T ← T ∪ {(π[u], u)}                            Agrega la arista al MST
9        for v ∈ α[u] :                                 Cada nodo adyacente a u
10           if v ∈ Q :                                 Si no ha sido visitado (sigue en el heap)
11               if d[v] > cost(u, v) :
12                   d[v] ← cost(u, v);  π[v] ← u        } Actualiza el heap
13                   DecreaseKey(Q, v, d[v])            } con los nuevos costos
14   return T                                           lista de aristas del MST
```

input : heap representado como arreglo $H[0 \ldots n-1]$
output: elemento más prioritario

```
Extract(H):
    i ← última celda no vacía de H
    best ← H[0]
    H[0] ← H[i]
    H[i] ← ∅
    SiftDown(H, 0)        restablece propiedad heap
    return best
```

octubre 2025

KRUSKAL, PRIM Y HEAPS

Joaquín - Steven - Elías - Alexander - Paula