Árbol Generador Mínimo

Federico Pousa

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Algoritmos y Estructuras de Datos III

Contenidos

Repaso de definiciones

Definición (Árbol generador)

Sea G un grafo. Decimos que T es un *árbol generador* de G si se cumplen estas condiciones:

- T es subgrafo de G.
- T es un árbol.
- T tiene todos los vértices de G.

Definición (Costo de un árbol generador)

Sea G un grafo con pesos en sus aristas. Si T es un árbol generador de G, definimos el *costo* de T como la suma de los pesos de las aristas de T.

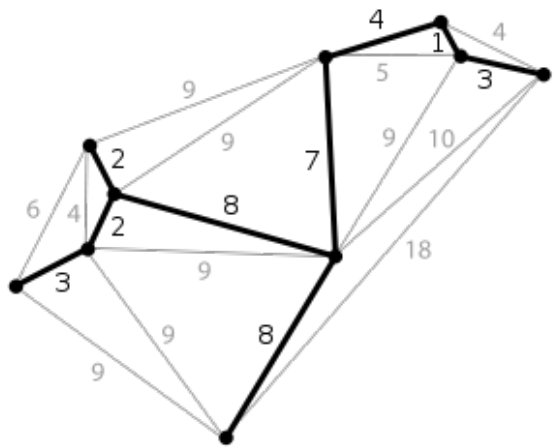
¿Qué es un AGM?

Definición (AGM)

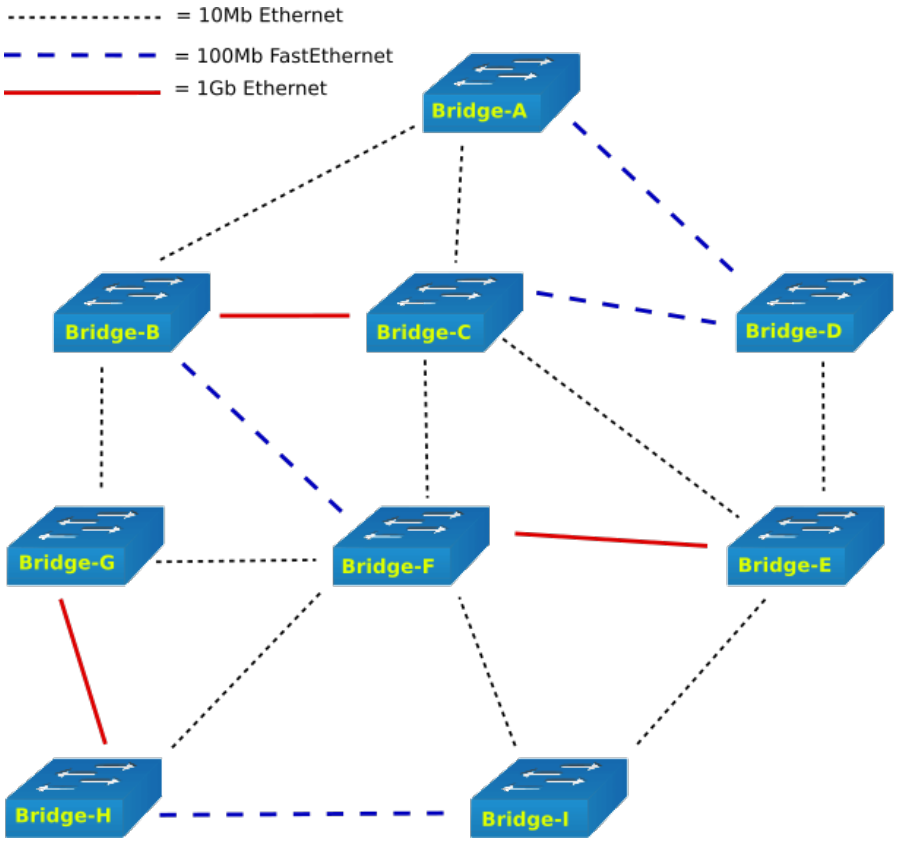
Sea G un grafo. Decimos que un grafo T es un árbol generador mínimo (AGM) de G si:

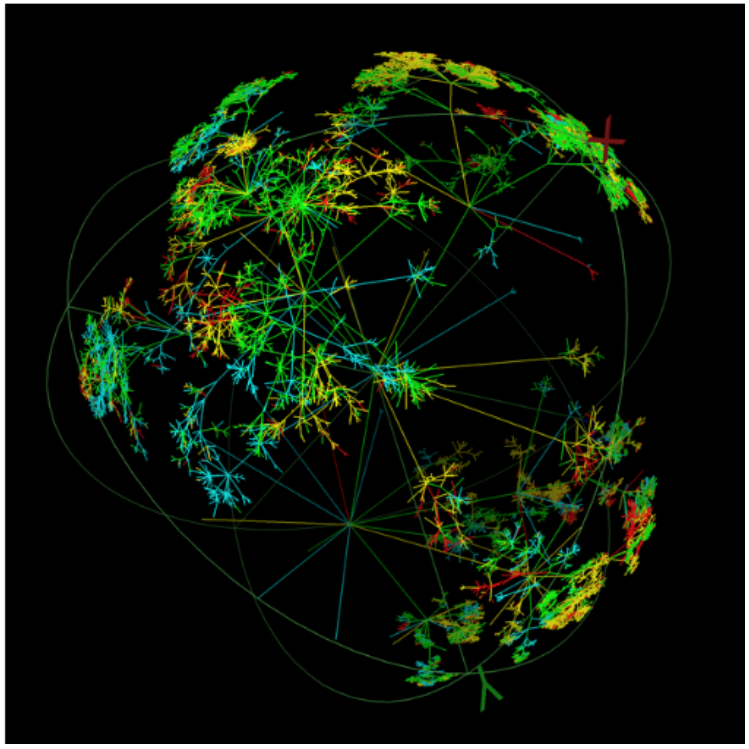
- T es un árbol generador de G.
- El costo de T es mínimo con respecto a todos los árboles generadores de G.

Aca vemos un grafo G (en gris) y un subgrafo T (en negro) que es AGM de G.

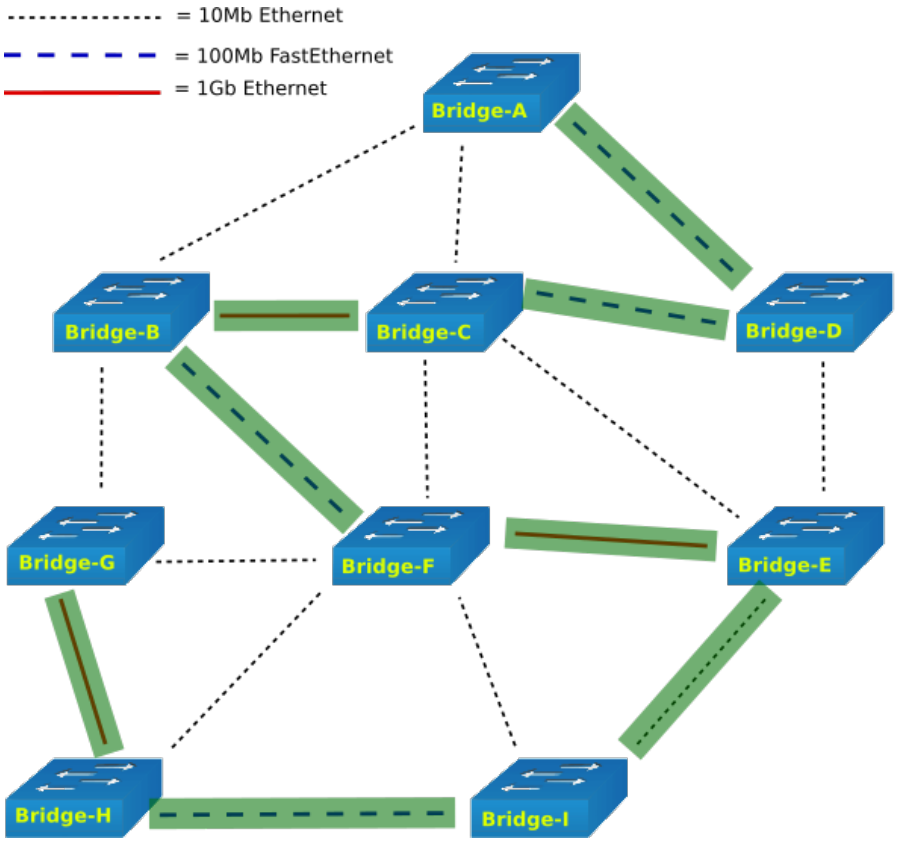


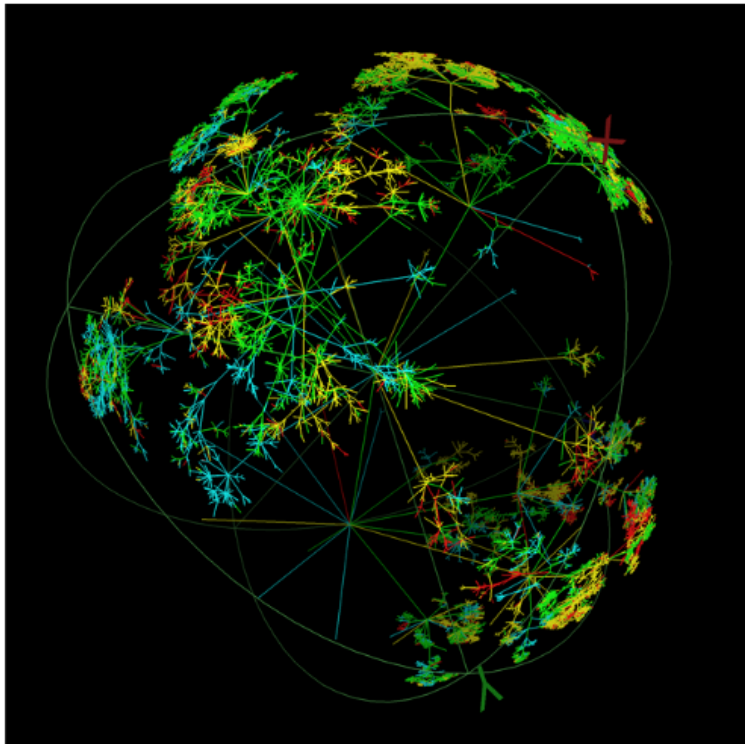
Spanning Tree Protocol (STP)





Spanning Tree Protocol (STP)





¿Como hallamos un AGM de un grafo?

El algoritmo de Kruskal sirve para resolver el problema de hallar un AGM de un grafo. Un pseudocódigo del algoritmo es el siguiente:

```
1 lista(aristas) Kruskal(lista(aristas)):
2     solucion = lista vacia // inicializo la solucion
3     sort(aristas) // ordeno las aristas por peso (de menor a mayor)
4     for e in aristas:
5         si agregar e a solucion no genera un ciclo:
6             agregar e a la solucion
7     return solucion
```

El código parece muy simple pero hay un paso no trivial clave. ¿Cómo verificamos (rápido) que agregar una arista no genera un ciclo?

Detectando ciclos

Notemos que al ejecutar Kruskal sobre un grafo G:

- En cada paso intermedio la solución (parcial) nos divide los vértices de G en varias componentes conexas.
- Una nueva arista nos generaría un ciclo si (y sólo si) nos une dos vértices de la misma componente conexa.
- Al agregar una nueva arista a la solución juntamos dos componentes conexas en una sola más grande.

Sabiendo esto, la siguiente estructura nos será de utilidad.

UDS

La estructura de datos Union Disjoint Set (UDS) es una estructura que nos permite manejar conjuntos disjuntos de elementos (como las componentes conexas).
Cada conjunto tiene un representante que lo identifica.
La estructura nos debe proveer las siguientes dos operaciones:

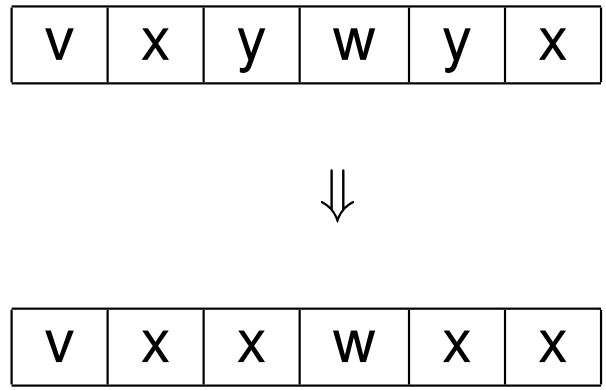
- find(x): Dado un elemento x, nos dice quien es el representante del conjunto al que pertenece x.
- union(x, y): Dados dos elemenos x e y, une los conjuntos a los que pertenecen x e y en uno solo.

En el caso de Kruskal, inicialmente cada elemento x está en un conjunto distinto y es su propio representante.

Primer approach

Una posible implementación de DSU es tener un arreglo $C[]$ con los representantes de cada elemento. Podemos implementar las dos operaciones de esta manera:

```
1  find(x):
2      return C[x]
3
4  union(x, y):
5    for z tal que C[z] = C[y]:
6        C[z] = C[x]
```



El problema es que con este algoritmo si hay revisamos todos los z posibles, hacer $O(n)$ unions donde n es la cantidad de elementos es $O(n^2)$. Buscamos algo mejor.

Optimizaciones

Esto se puede mejorar con las siguientes dos obervaciones:

- Podemos mantener una lista con los elementos del conjunto representado por x para cada x.
- Cuando unimos dos conjuntos elegimos recorrer los elementos del conjunto más chico. Para esto hay que llevar una cuenta de cuantos elementos tiene cada conjunto.

¿Cambió la complejidad?

Optimizaciones

Esto se puede mejorar con las siguientes dos obervaciones:

- Podemos mantener una lista con los elementos del conjunto representado por x para cada x.
- Cuando unimos dos conjuntos elegimos recorrer los elementos del conjunto más chico. Para esto hay que llevar una cuenta de cuantos elementos tiene cada conjunto.

¿Cambió la complejidad?
Con estas dos optimizaciones se puede ver que el costo de hacer $O(n)$ operaciones union o find es $O(nlog(n))$. Pero como somos ambiciosos vamos a buscar algo mejor todavía.

Versión con listas

```
1  | init(n):
2  |     for i en [0, n):
3  |         componente[i] = {i} // conjunto solo con el elemento i
4  |         padre[i] = i
5  |
6  | find(x):
7  |     return padre[x]
8  |
9  | union(x, y):
10 |     x = find(x), y = find(y)
11 |     if longitud(componente[x]) > longitud(componente[y]):
12 |         intercambiar x e y
13 |
14 |     for z en componente[x]:
15 |         padre[z] = y
16 |         agregar z a componente[y]
17 |     vaciar componente[x]
```

Versión con listas

```
1  | init(n):
2  |     for i en [0, n):
3  |         componente[i] = {i} // conjunto solo con el elemento i
4  |         padre[i] = i
5  |
6  | find(x):
7  |     return padre[x]
8  |
9  | union(x, y):
10 |     x = find(x), y = find(y)
11 |     if longitud(componente[x]) > longitud(componente[y]):
12 |         intercambiar x e y
13 |
14 |     for z en componente[x]:
15 |         padre[z] = y
16 |         agregar z a componente[y]
17 |     vaciar componente[x]
```

Como siempre agregamos los elementos del conjunto más chico al conjunto más grande, cada elemento que movemos de un conjunto a otro pasa a estar en un conjunto de al menos el doble de tamaño del anterior. Por lo tanto, cada elemento es agregado a lo sumo $O(\lg n)$ veces: esto implica una complejidad total de $O(n \lg n)$.

¡Un conjunto es un árbol!

La idea clave es que vamos a representar cada conjunto como un árbol.
El representante de un conjunto será la raíz del árbol al que pertenece.
Cada elemento sabe quién es su padre en el árbol. Si un elemento es raíz su padre es sí mismo.
Los algoritmos quedan así:

```
1 | find(x):
2 |     if padre[x] != x:
3 |         return find(padre[x]);
4 |     return x;
5 |
6 | union(x, y):
7 |     padre[find(x)] = padre[find(y)];
```

Optimizando II

Podemos hacer dos mejoras a este algoritmo:

- Colgar el árbol de menor altura al de mayor altura, pues el costo de un find es a lo sumo la altura del árbol.
- Cada vez que realizamos un find(x) actualizamos el padre de x con el resultado del find para ahorrarnos volver a tener que subir por el árbol ante futuros find.

Con estas dos optimizaciones se puede probar que realizar k operaciones init, union o find donde n operaciones son init es $O(k\alpha(n))$, donde $\alpha(n)$ es la inversa de Ackermann.

El final de la película

Así nos queda nuestro algoritmo que implementa DSU (incluyendo la inicialización):

```
1  init(n): \\ Inicializa los arreglos sabiendo que hay n elementos
2      for i = 1 to n:
3          altura[i] = 1; // "altura" es una cota superior de la altura
4          padre[i] = i;
5
6  find(x):
7      if padre[x] != x:
8          padre[x] = find(padre[x]);
9      return padre[x];
10
11 union(x, y):
12     x = find(x), y = find(y) // Tomo los representantes de cada conjunto
13     if altura[x] < altura[y]:
14         padre[x] = y;
15     else
16         padre[y] = x;
17     if altura[x] = altura[y]:
18         altura[x] = altura[x] + 1;
```

Contenidos

La verdad de la milanesa

Ahora sí podemos dar un algoritmo completo y eficiente de Kruskal:

```
1  lista(aristas) Kruskal(lista(aristas), int n):
2      solucion = lista vacia // inicializo la solucion
3      init(n)
4      sort(aristas) // ordeno las aristas por peso (de menor a mayor)
5      for e in aristas:
6          if find(e.inicio) != find(e.fin): // si los vertices que une
               la arista estan en componentes distintas
7              agregar e a la solucion
8              union(e.inicio, e.fin) // uno las dos componentes
9      return solucion
```

Este algoritmo es $O(V + E \lg V)$ con V la cantidad de vértices y E la cantidad de aristas.

Contenidos

Hiperconectados

La planificación de las ciudades es una herramienta fundamental para mitigar los problemas del crecimiento poblacional. Al tener muchas ciudades, un problema clásico a resolver es el del diseño de redes de conexión entre las mismas, ya sea para los medios de transporte como los automóviles o los trenes, así como para los diferentes tipos de servicios básicos como el agua potable o la electricidad. En este caso resolveremos un problema asociado a la conectividad del servicio más básico e importante para la humanidad, internet.
Tenemos un conjunto de n ciudades, para las cuales queremos crear una red de conexiónes que cumpla que todo par de ciudades está conectado por al menos un camino de nuestra red. Para lograr esto, tenemos m potenciales conexiones entre pares de ciudades. Cada potencial conexión posee tres atributos: dos ciudades a y b, que indica simétricamente que ciudades serían conectadas y c que representa el costo de instalar dicha conexión. Nuestro objetivo es crear la red indicada de manera tal que el costo total de la misma sea el mínimo, para lo cual ya sabemos que existen algoritmos que nos pueden ayudar específicamente con dicha tarea. Sin embargo, nuestro problema no se reduce simplemente a esto. Si bien las conexiones tienen un valor asociado que nos indica cuanto nos costará su instalación, hay otras variables asociadas a las conexiones que se están ocultando por una cuestión de simplicidad. Una conexión podría tener un costo de instalación bajo, pero a la vez podría ser una conexión no muy deseada por encontrarse en una zona sísmica, lo cual aumentaría las chances de tener que ser sometida a reparaciones. También podría haber una conexión con costo asociado bajo, pero que su instalación genere un descontento público por el lugar donde debe realizarse la instalación.

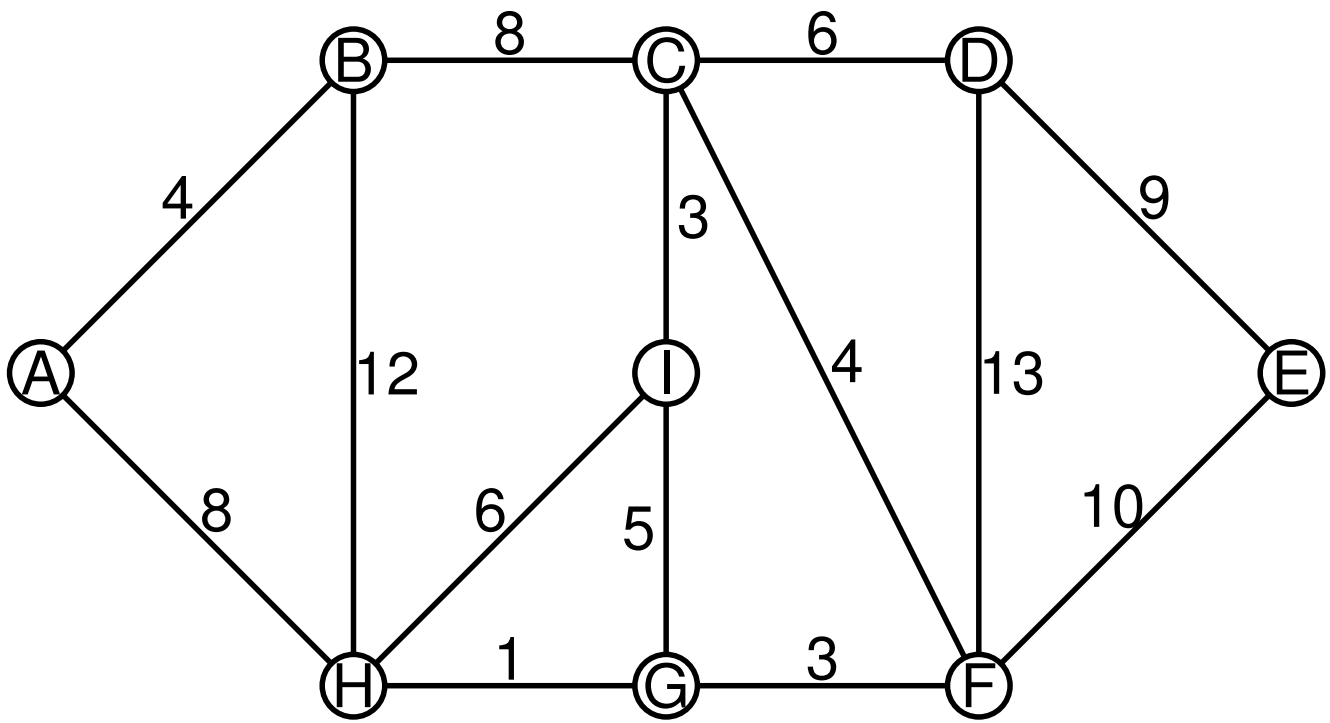
Es por estas situaciones que más alla de encontrar la forma menos costosa de armar una red entre las n ciudades tal que cada par de ciudades se encuentre conectado de al menos una manera (situación que denominaremos Anexión General Minúscula o AGM por sus siglas), tendremos aparte el siguiente problema. Por nuestros conocimientos en el área sabemos que la solución AGM del problema no tiene porque ser única, queremos saber entonces por cada posible conexión si la misma tiene la propiedad de 1) Estar en todo AGM del problema 2) No estar en ningún AGM del problema 3) Estar en algún AGM del problema.

Contenidos

Solución

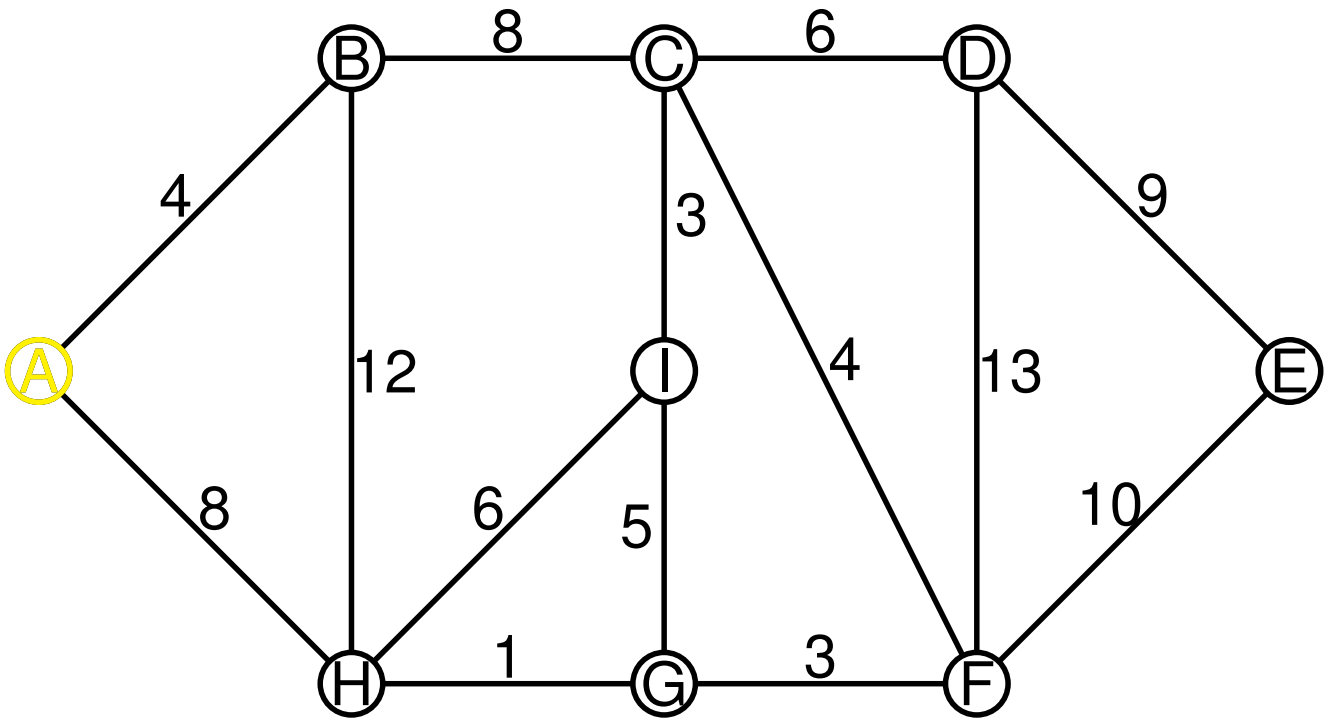
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



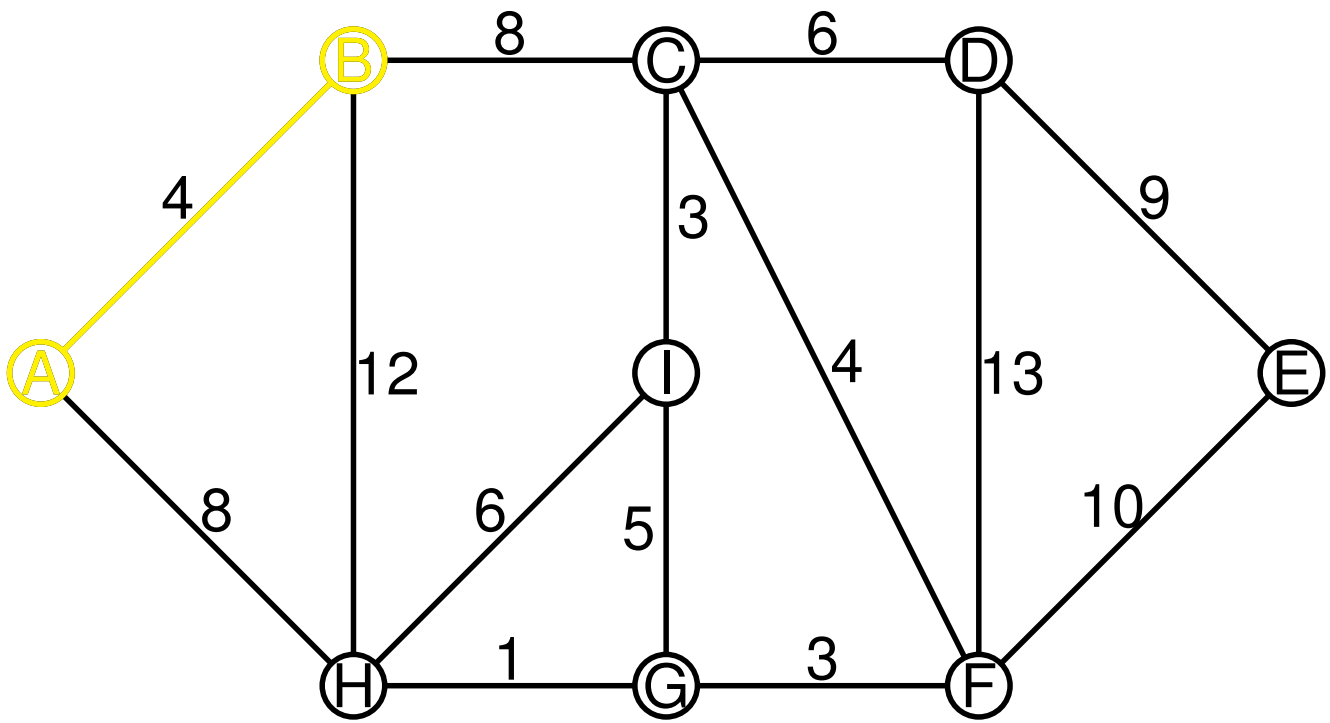
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



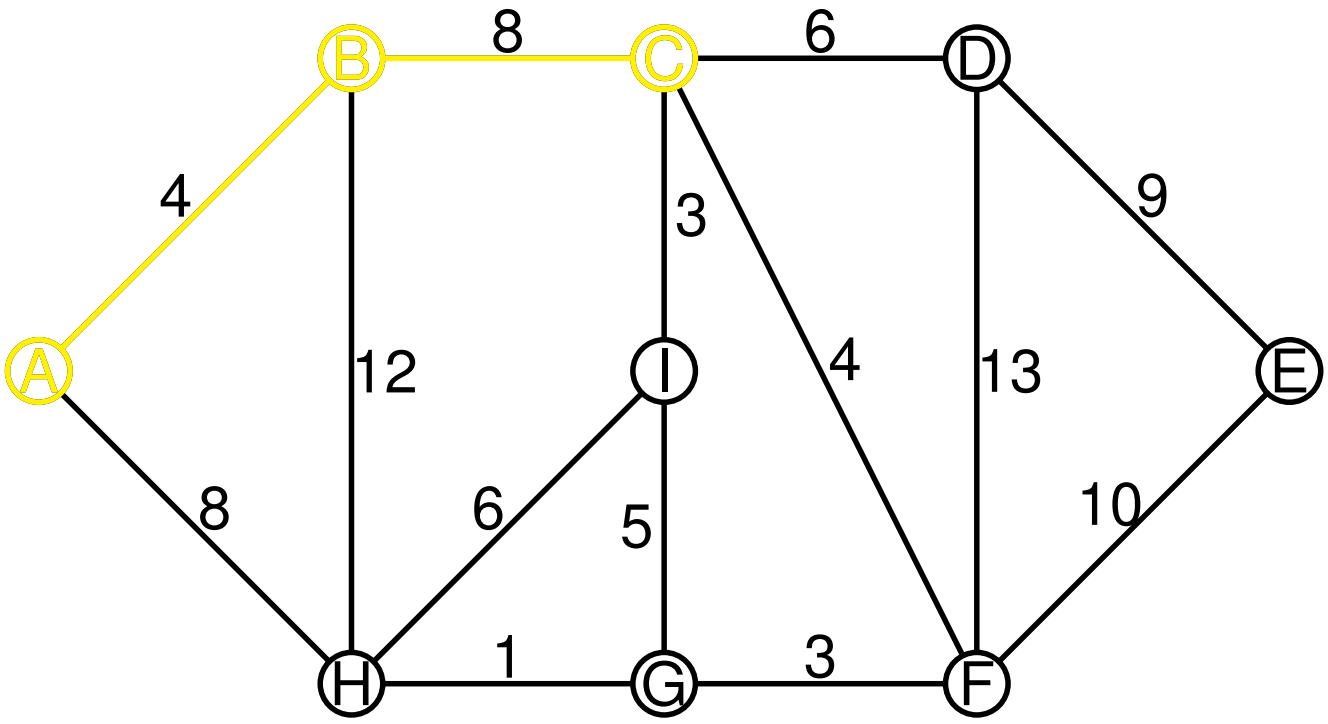
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



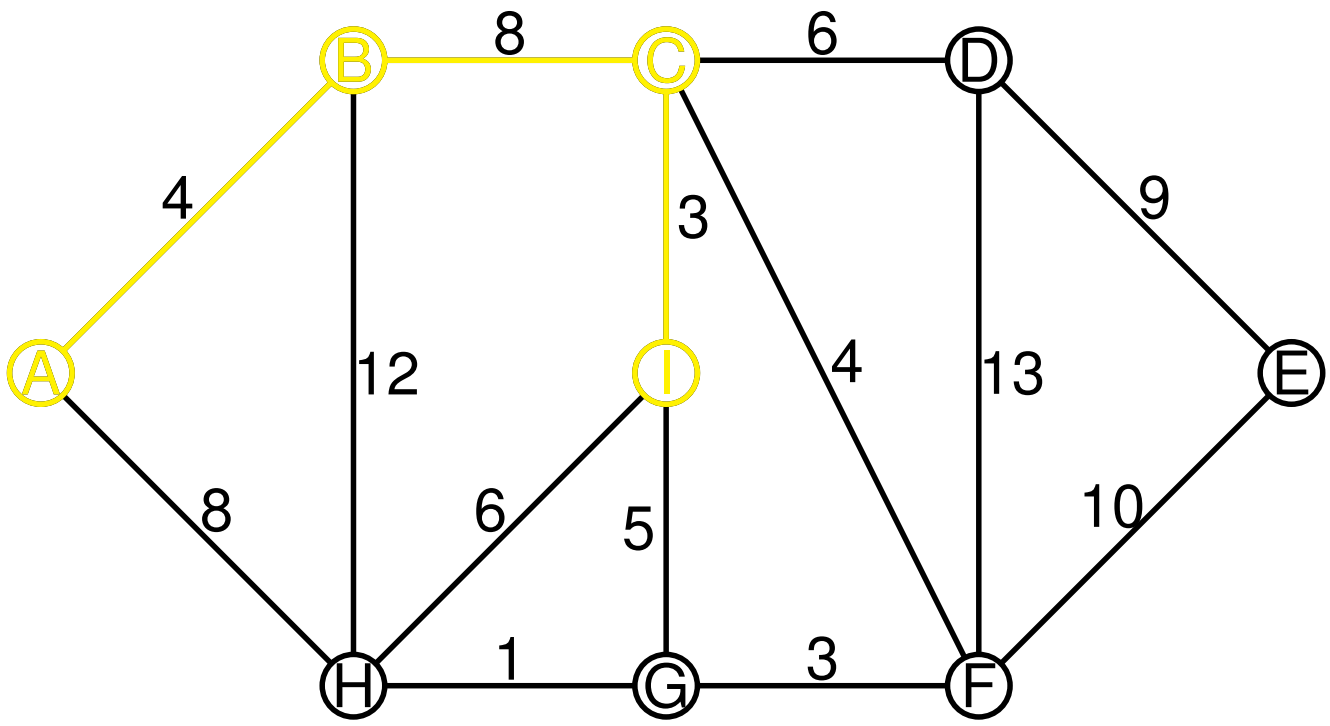
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



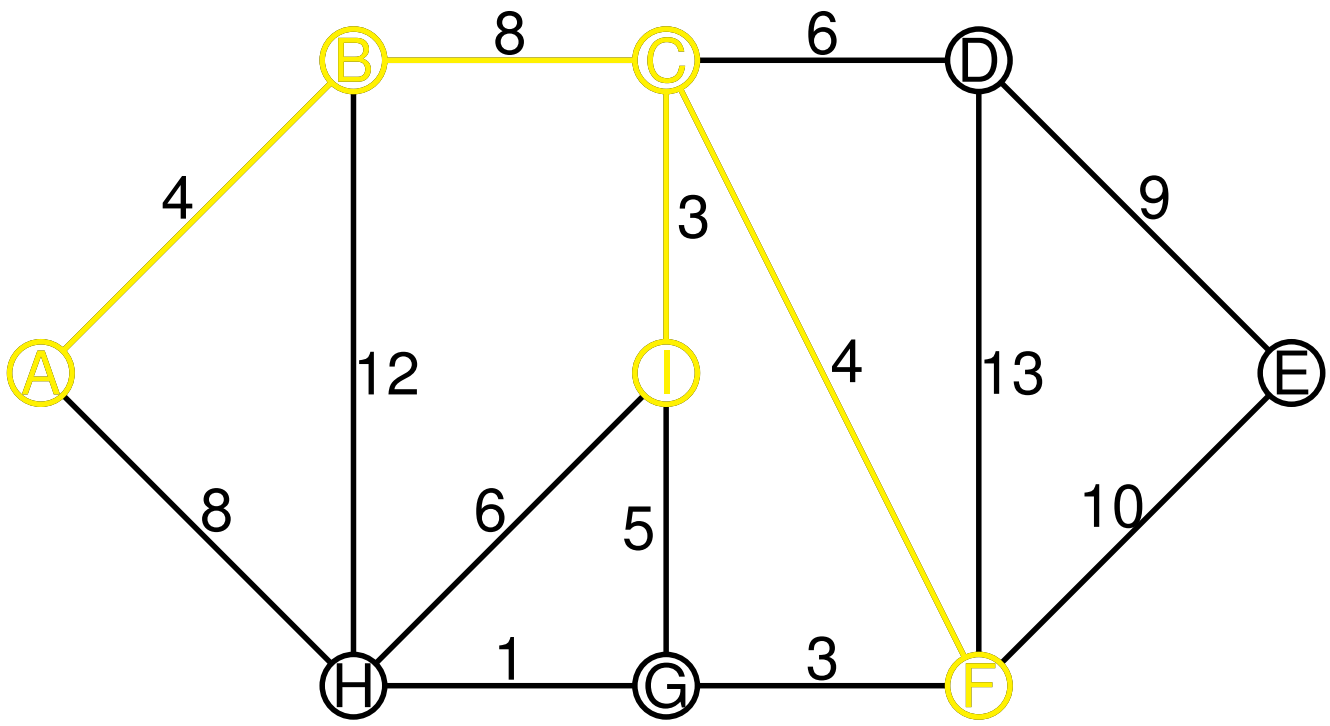
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



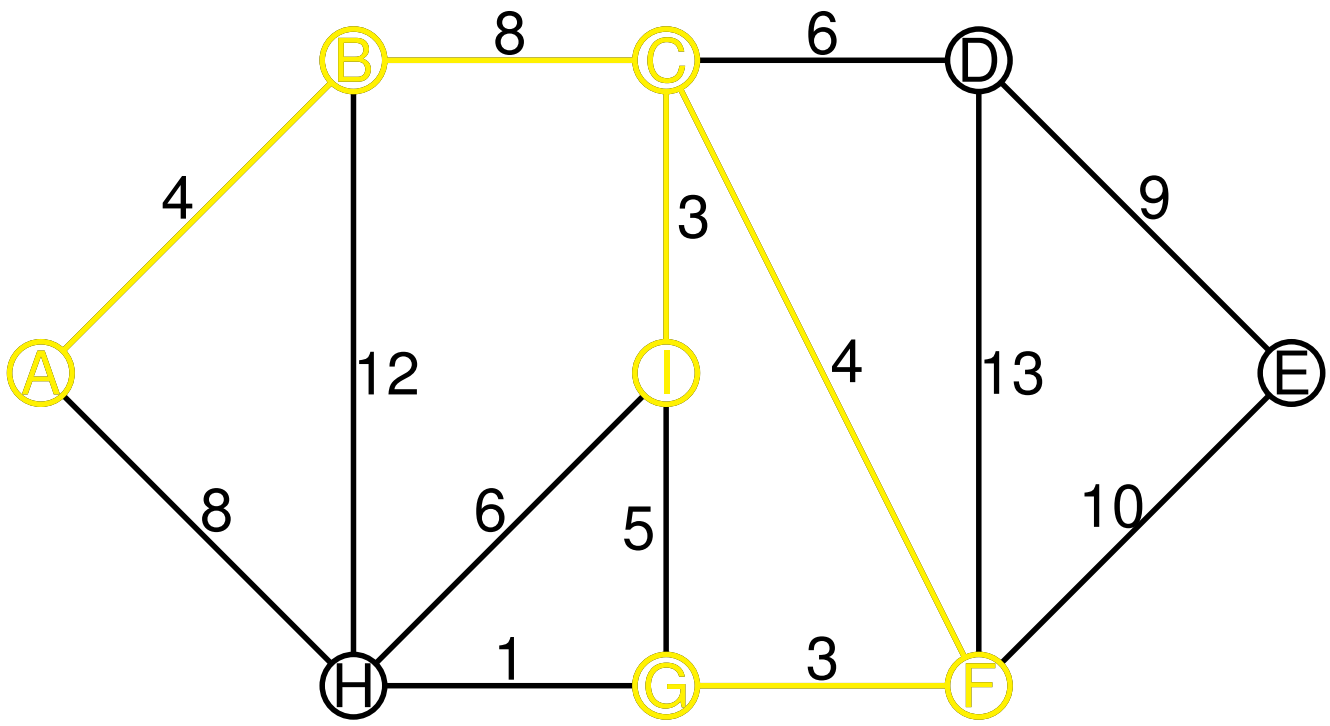
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



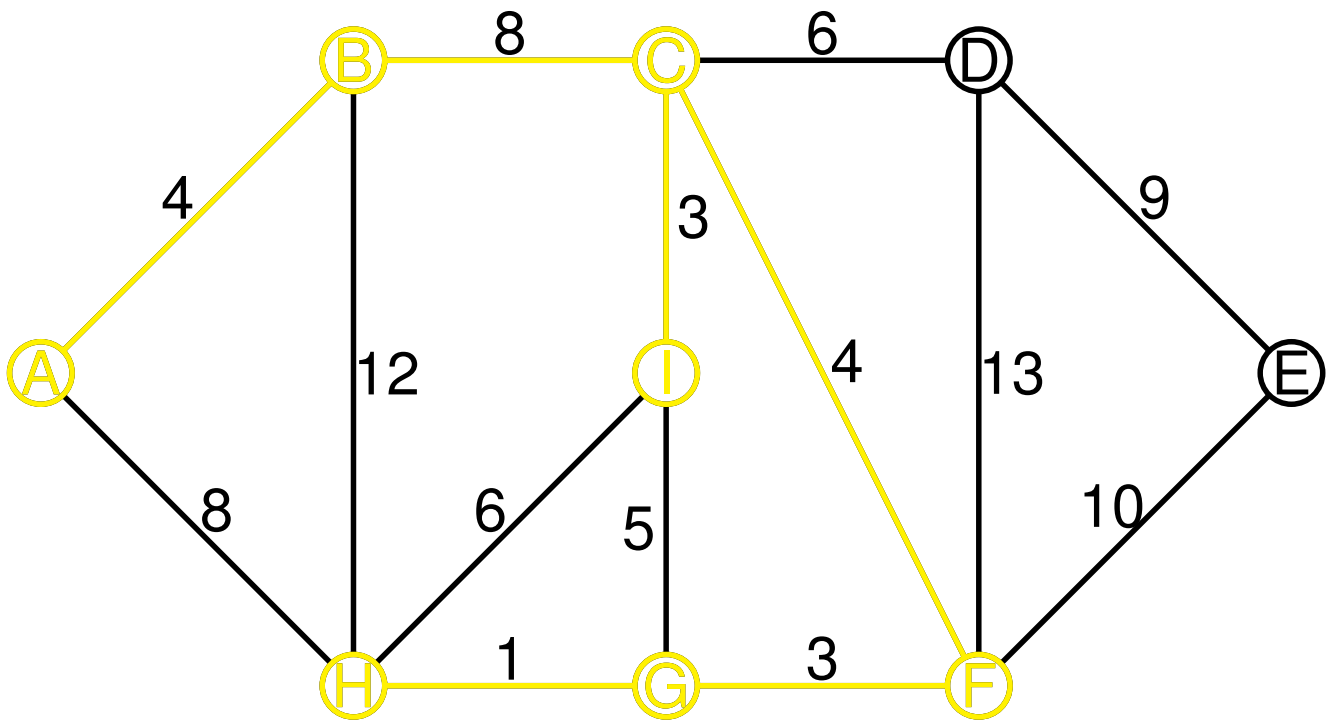
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



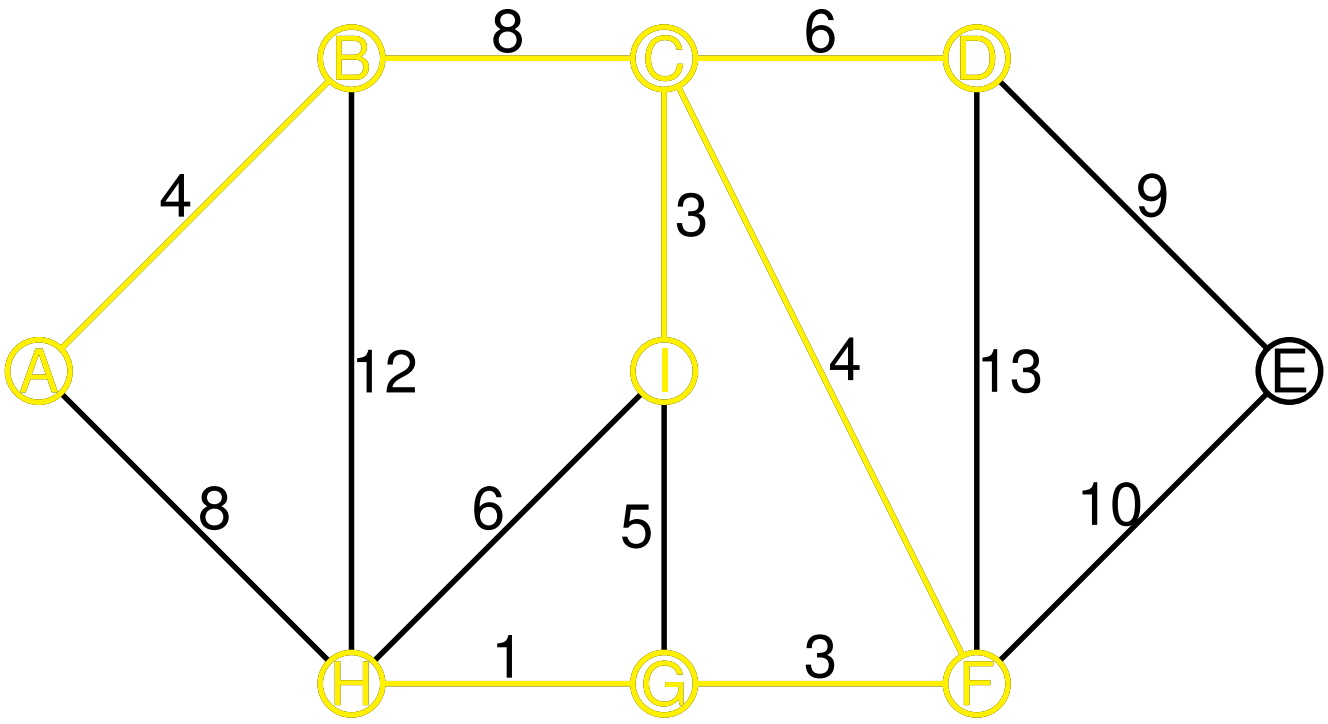
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



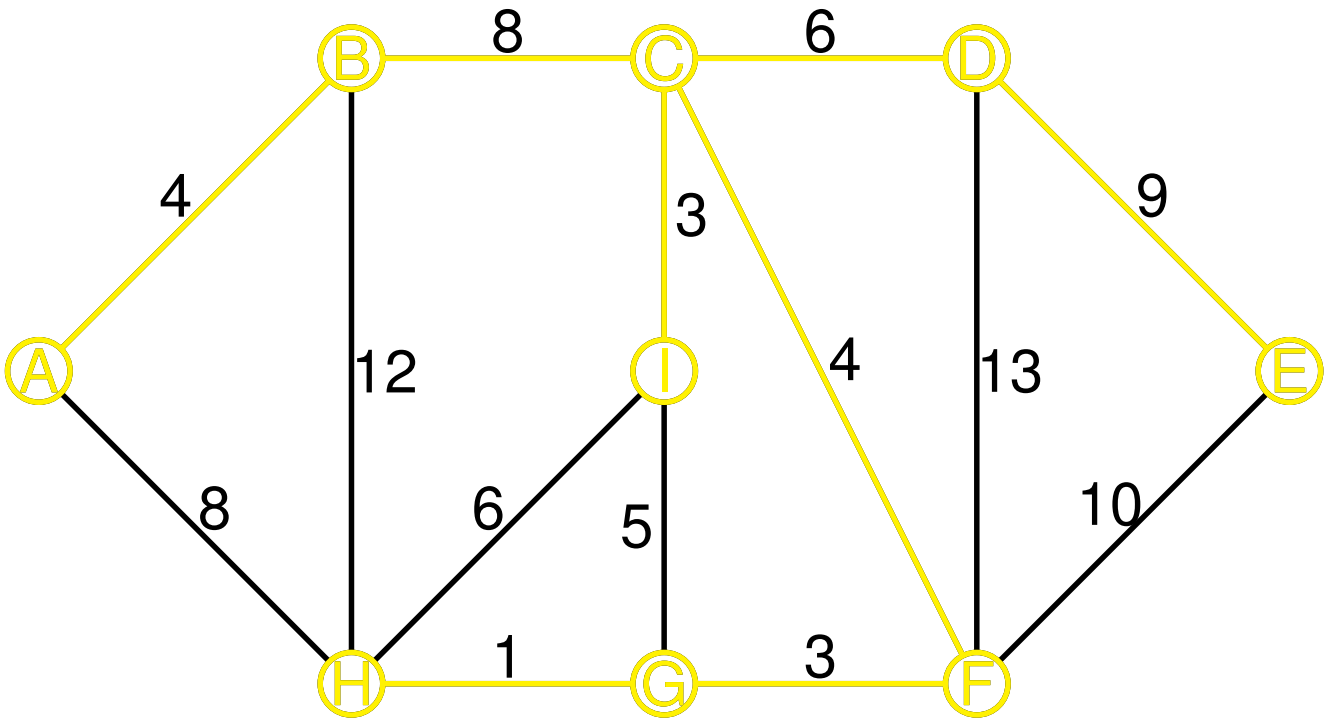
Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



Algoritmo de *Prim*

Veamos un ejemplo robado de la teórica para recordar cómo funciona:



Pseudocódigo abstracto de Prim

```
1  Prim (Grafo G)
2     visitado[n] = {false, ..., false} // guarda si un nodo ya fue
          alcanzado por algun eje
3
4     s = elegir un nodo inicial (puede ser cualquiera de G)
5     visitado[s] = true
6
7     mientras que no esten todos los nodos visitados hacer
8        (v, w) = eje de menor costo entre los que conectan un nodo ya
              visitado v con un nodo no visitado w
9        visitado[w] = true
```

Ya vieron el algoritmo de Prim en su forma más abstracta.
Pero, ¿cómo lo implementamos?

Contenidos

Posibles implementaciones de Prim

- Existen dos implementaciones muy famosas del algoritmo de Prim. Deberíamos analizar sobre qué tipo de grafo queremos correr este algoritmo y en función de eso elegir cuál programar
- La implementación básica es de $O(|V|^2)$
- La implementación con cola de prioridad es $O(|E|log|V|)$
- Cuando da lo mismo cuál implementación elegir, **elijan la que sientan que entienden mejor y van a tener menos errores al programar**

Posibles implementaciones de Prim

- Si el grafo es **ralo**, o sea, tiene pocas aristas, conviene utilizar la implementación con cola de prioridad ($O(|E|log|V|)$)

Posibles implementaciones de Prim

- Si el grafo es **ralo**, o sea, tiene pocas aristas, conviene utilizar la implementación con cola de prioridad ($O(|E|log|V|)$)
- Si el grafo es **denso**, o sea, tiene muchas aristas, conviene utilizar la implementación básica ($O(|V|^2)$)

Conceptos clave para ambas implementaciones

- Cuando agregamos un nodo al subárbol ya explorado por Prim, diremos que el mismo fue *visitado*. Tendremos un arreglo que indique en todo momento cuáles nodos fueron visitados y cuáles no.
- Llamaremos *distancia* a la distancia de un nodo al subárbol ya explorado por Prim. Mantendremos un arreglo con estos valores, y en cada momento *distancia*[*i*] contendrá el peso del eje de menor costo desde el subárbol explorado al nodo *i* que hallamos hasta el momento. Si no existe tal nodo, esta distancia será ∞.
- Llamaremos *padre* al nodo desde el que proviene el mejor eje hallado por *distancia* hasta el momento. Durante el algoritmo pueden no coincidir con los predecesores del AGM final, pero una vez terminado el algoritmo lo contendrán.

Pseudocódigo de Prim sin cola de prioridad

```
1  | Prim (Grafo G)
2  |   visitado[n] = {false, ..., false}
3  |   distancia[n] = {Infinito, ..., Infinito}
4  |   padre[n] = {null, ..., null}
5  |
6  |   s = tomo un nodo cualquiera de G como nodo inicial
7  |   para cada w en V[G] hacer
8  |       si existe arista entre s y w entonces
9  |           distancia[w] = peso (s, w)
10 |           padre[w] = s
11 |
12 |   distancia[s] = 0
13 |   visitado[s] = true
14 |
15 |   mientras que no esten visitados todos hacer
16 |       v = nodo de menor distancia para agregar que no fue visitado aun
17 |       visitado[v] = true
18 |       para cada w en sucesores (G, v) hacer
19 |           si distancia[w] > peso(v, w) entonces
20 |               distancia[w] = peso(v, w)
21 |               padre[w] = v
22 |
23 |   devolver padres
```

Pseudocódigo de Prim con cola de prioridad

```
1  Prim (Grafo G)
2      para todo u en V[G] hacer
3          distancia[u] = INFINITO
4          padre[u] = NULL
5          visitado[u] = false
6
7      s = tomo un nodo cualquiera de G como nodo inicial
8      distancia[s] = 0
9      adicionar (cola, (s, distancia[s]))
10
11     mientras que cola no sea vacia hacer
12         u = extraer_minimo(cola) // u es el nodo que realiza el minimo
13         si visitado[u] continuar con el proximo ciclo
14         visitado[u] = true
15         para todo v en adyacencia[u] hacer
16             si no visitado[v] y distancia[v] > peso(u, v) hacer
17                 distancia[v] = peso(u, v)
18                 padre[v] = u
19                 adicionar(cola, (v, distancia[v]))
20     devolver padres
```