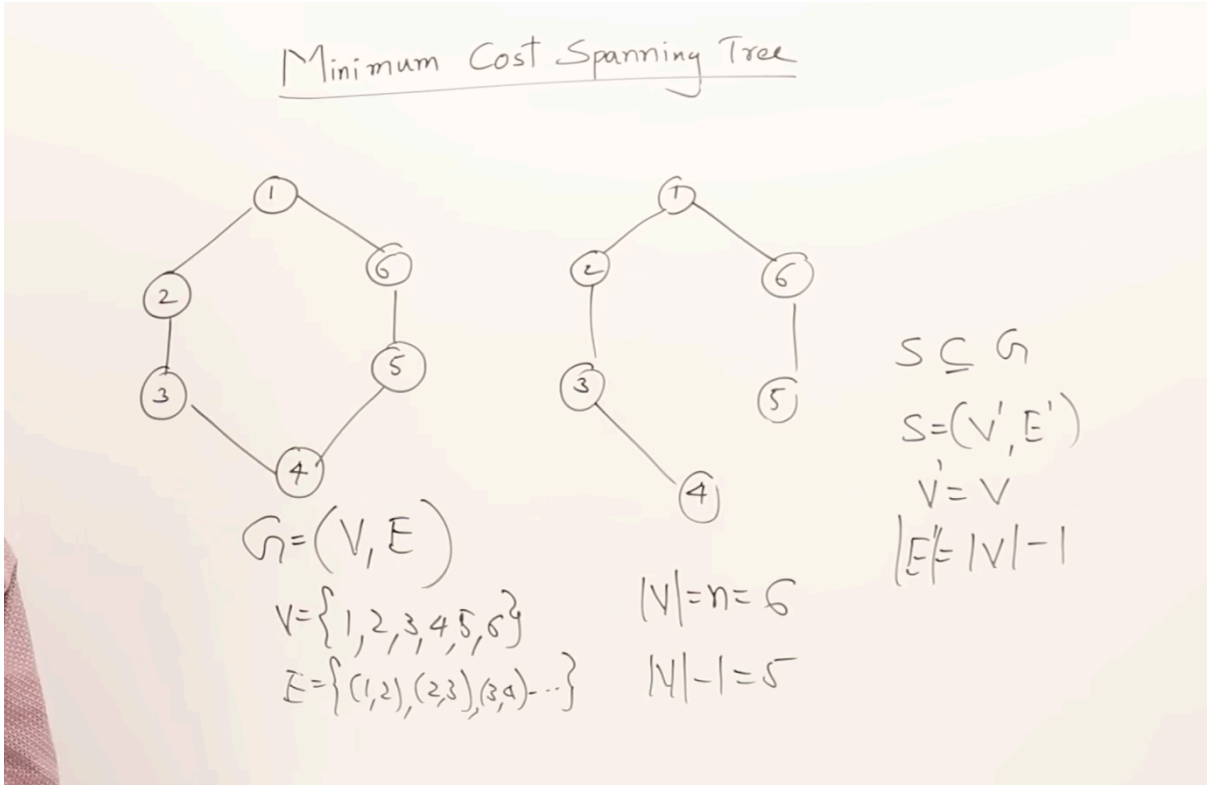
Grafos no dirigidos

Apuntes del video de Abdul:

Spanning tree:

Lo mismo que el grafo conexo normal, pero con una arista menos a la cantidad de vértices, que rompa el ciclo (n vértices, n - 1 aristas), se puede cortar cualquiera de las aristas :



Minimun Cost Spanning Tree

El coste del spanning tree resultante podrá variar su peso total dependiendo de cual vértice quites.

Prim’s Algorithm:

Para buscar el de menor peso posible es necesario ir vértice por vértice buscando el arco de menor costo hasta conseguir la cantidad de aristas necesaria. No es muy diferente de como funcionan los algoritmos de Floyd y Dijkstra, pero para grafos no dirigidos en este caso.

Orden de tiempo:

Lo normal es que sea O(n^2) cuando es una matriz

También puede ser

Usando cola de prioridad (heap binario) + lista de adyacencia:

- **Tiempo:** `O((n + m) log n)`
- Donde `n` es el número de vértices y `m` de aristas.

Kruskal Algorithm:

Lo mismo que el Prim’s, pero siempre evita crear un ciclo aunque tenga que buscar un arco con mayor coste.

Orden de tiempo: O(v*a), o O(n^2)

Con min Heap se puede hacer más rápido: O (n log n) **(Tener en cuenta este último más que nada)**

Siempre se puede encontrar más de un spanning tree para un mínimo costo posible.

Libro:

Un grafo no dirigido es lo mismo que un grafo dirigido, pero cada arista “A” no es un par ordenado de vértices, ya que (v,w) = (w,v)

Si (v,w) es un una arista, se dice que es incidente sobre los vértices v y w, y los vértices son adyacentes entre sí

- Un camino es una secuencia de vértices v1, v2, v3 ..... vN, tal que (vi, vi + 1) es una arista.
- Camino simple: Todos los vértices distintos, salvo que parta del mismo con el que terminó
- Camino conexo: Camino en donde todos los nodos se conectan
- Ciclo: Hay un bucle (3 o más vértices)

Subgrafo inducido: Subgrafo con todos los vértices y aristas entre ellos, presentes en el grafo original.

Árbol libre = spanning tree (si se agrega cualquier arista, resulta en un ciclo)

Métodos de representación:

- Matrices o listas de adyacencias
- Recordar que una arista no dirigida, implícitamente refiere a 2 aristas dirigidas, una arista de V a W no dirigida representa los pares (V,W) y (W,V)
- Las matrices de adyacencias serán siempre simétricas

Árboles abarcadores de costo mínimo

Dado un grafo G = (V,A) donde cada arista (u,v) de A tiene un costo asociad c(u,v):

- Un árbol abarcador de G es un árbol libre que conecta todos los vértices de V.
- El costo de ese árbol es la suma de costo de las aristas

Recordemos las propiedades de un árbol abarcador:

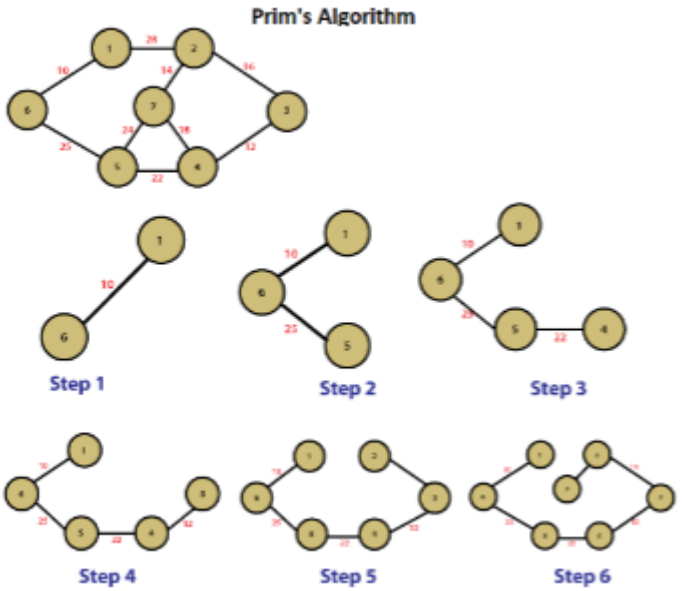
- Conecta todos los vértices (grafo conexo)
- No tiene ciclos
- La suma de costo es lo mínimo posible, sino se podrían hacer muchísimos árboles abarcadores

Métodos para hallar este árbol Abarcador:

Algoritmo de Prim (ver la sección del video de Abdul)

Para añadir, en pseudocódigo:

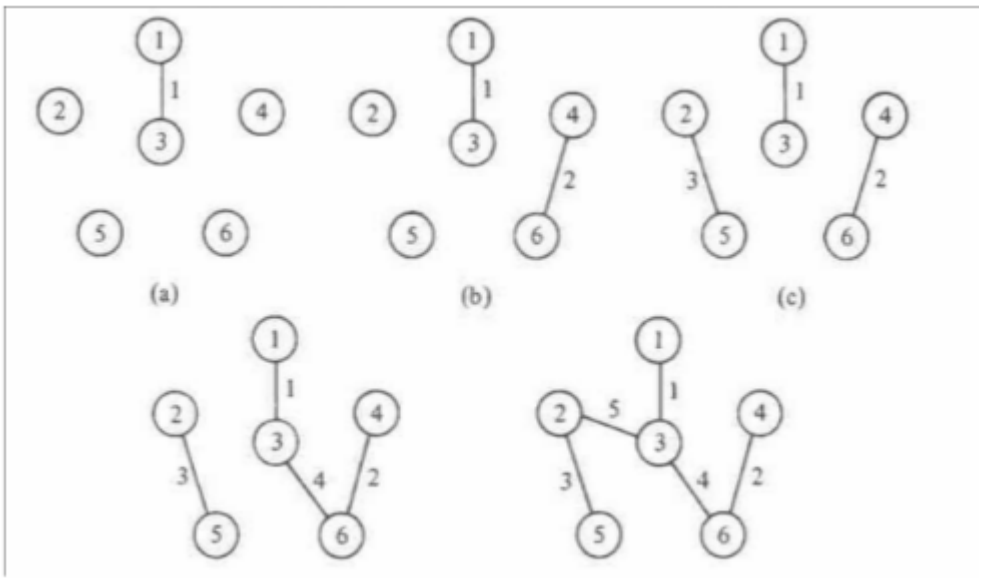
```
 Método TGRAFO.Prim ( conjunto de aristas T);
2 U: conjunto de vértices;
3 u, v: vértice;
4 // el TGRAFO representado por un conjunto de vértices V y un conjunto de
Aristas A
5 COMIENZO
   6 T.Vaciar();
   7 U.Agregar(1);
   8 MIENTRAS U <> V hacer
      9   elegir una arista (u,v) de costo mínimo tal que u está en U y v está en
      11 T.agregar (u,v);
      12 U.agregar(v);
   13 FIN MIENTRAS
14 FIN
```



Algoritmo de Kruskal (ver la sección del video de Abdul)

Para agregar:

```
 Método TGrafo.Kruskal;
2 F conjunto de aristas;
3 COM
    4 F.Vaciar;
    5 Repetir:
        6 Elegir una arista de costo mínimo que no esté en F ni haya sido elegida
        7 Si arista no conecta dos vértices del mismo componente entonces agre
    8 hasta que todos los vértices estén en un solo componente;
9 FIN
```



Búsqueda en profundidad y amplitud:

Ver resumen de la unidad anterior. (https://www.notion.so/Grafos-dirigidos-20563ae195d480af8678f0f9a25a52cc?source=copy_link)

Diferencias con la unidad anterior

Hay 2 tipos de arcos:

En la búsqueda en amplitud:

- De árbol
- De retroceso

Si el grafo es conexo, la búsqueda en profundidad se obtiene un sólo árbol

```
 Método Tvertice.bea() : String
2 // bea visita todos los vértices conectados a 'this' usando busq en amplitud
3
4 Variables:
5 C : ColaDeVértices
6 x, y : Vértice
7 tempstr : String ← ""
8
9 Inicio:
10 this.Visitar()
```

```
11 C.Insertar(this)
12 tempstr ← tempstr + this.etiqueta
13
14 mientras no C.vacía() hacer:
15 x ← C.Eliminar() // x toma el vértice al frente de la cola
16
17 para cada vértice y adyacente a x hacer:
18 si no y.Visitado() entonces:
19 y.Visitar()
20 C.Insertar(y)
21 tempstr ← tempstr + y.etiqueta
22 fin si
23 fin para
24 fin mientras
25
26 devolver tempstr
27 Fin
```

Tiempo de ambos: O(a) en donde a es el num de aristas

Punto de Articulación:

Vértice que al eliminarse junto con sus aristas, se divide en uno o más subgrafos conexos.

Algoritmo para encontrar dicho punto:

1. Realizar una búsqueda en profundidad (DFS) numerando los vértices en orden previo. A cada vértice v se le asigna un número número_bp[v] que indica el momento en que fue descubierto.
2. Para cada vértice v, calcular el valor bajo[v], que representa el menor número alcanzable desde v:
• Bajando por el árbol DFS (hacia sus descendientes),
• Y subiendo luego por una arista de retroceso.
3. Una vez calculados los valores bajo de los hijos de v, se define:

```
1 bajo[v] = min(número_bp[v],
3 número_bp[z] para cada z con una arista de retroceso desde v,
4 bajo[y] para cada hijo y de v)
```

Para entenderlo más fácil:

El low es menor cuando uno de sus descendientes tiene un arco de retroceso con el que accede a un nodo con menor tiempo de descubrimiento, ahí su valor bajo (low) será igual al de dicho nodo con menor tiempo de descubrimiento

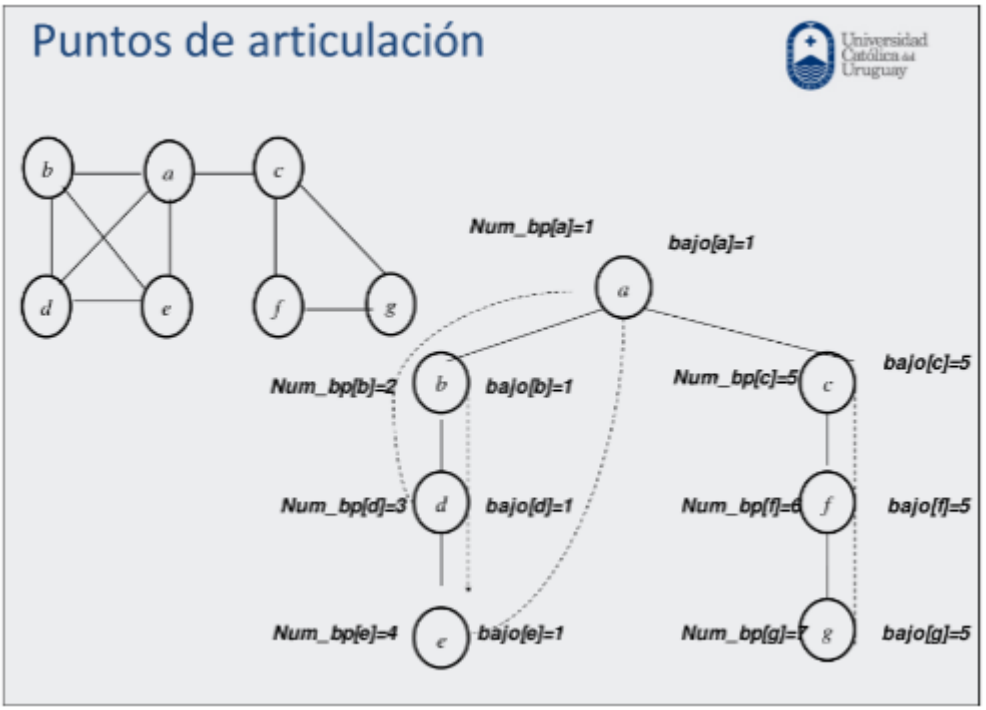
Condiciones para que lo sea:

- La raíz del árbol DFS es un punto de articulación si tiene 2 o más hijos
- Un vértice v (distinto de la raíz) es punto de articulación si existe un hijo w tal que:

```
bajo[w] ⇒ número_bp[v]
```

Esto indica que no existe una ruta desde w hacia un antecesor de v sin pasar por v, por lo que eliminar v desconectaría el subárbol de w

- Los valores bajo se calculan durante un recorrido en orden posterior.
- La verificación de puntos de articulación se puede hacer en tiempo lineal, O(a), siendo a el número de aristas, si se representa el grafo con listas de adyacencia



1. Prim eficiente → usa Priority Queue (cola de prioridad)

- En vez de buscar manualmente la arista de menor peso (como en la versión $O(n^2)$), usás una **cola que siempre te da el nodo con menor peso**.
- En Java, eso sería con `PriorityQueue<T>`.
- Esto permite que el algoritmo corra en **O((n + m) log n)** en lugar de $O(n^2)$.

---

2. Kruskal eficiente → usa Union-Find (conjuntos disjuntos)

- Sirve para saber si **dos vértices ya están conectados** (es decir, en el mismo componente).
- Tiene dos operaciones rápidas:
  - **Find(v)** → te dice a qué conjunto pertenece `v`
  - **Union(u, v)** → une los conjuntos de `u` y `v` si no estaban ya juntos
- Se implementa con arrays y compresión de caminos, y logra **O(m log m)** de complejidad.