Algoritmos Y Estructuras de Datos

Compendio: Grafos No Dirigidos (UT8)

Santiago Blanco

14-06-2025

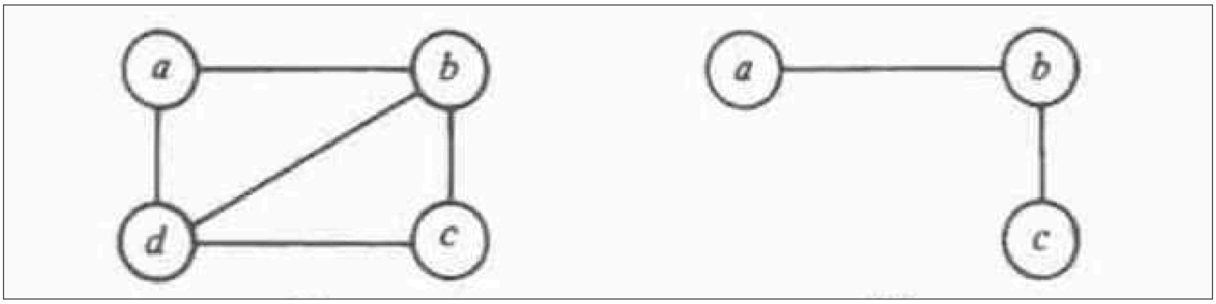
Grafos No Dirigidos

→ Un grafo NO dirigido $G = (V, A)$ es lo mismo que un grafo dirigido, excepto que cada arista en A no es un par ordenado de vértices. → $(v, w) = (w, v)$

- Si (v,w) es 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 $v_1, v_2, ..., v_n$, tal que (v_i, v_{i+1}) es una arista.
- **Camino simple:** Todos los vértices son distintos (excepto inicio y fin si son iguales).
- Un grafo es **conexo** si todos sus pares de vértices están conectados.
- Un **ciclo** (simple) es un camino (simple) de longitud mayor o igual a tres (3), que conecta un vértice consigo mismo.

Un **subgrafo** de un grafo $G = (V, A)$ es un grafo $G' = (V', A')$ tal que:
- V′ ⊆ V (toma un subconjunto de vértices del grafo original),
- A′ ⊆ A y toda arista de A′ conecta vértices que están en V′.

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



Árboles libres
- Un **grafo no dirigido conexo acíclico** se conoce también como árbol libre
- Un árbol libre puede convertirse en uno ordinario eligiendo un vértice como raíz.
- Todo árbol libre con n >= 1 vértices tiene exactamente n-1 aristas.
- Si se agrega cualquier arista a un árbol libre, resulta un ciclo.

Métodos de representación de grafos no dirigidos
- Se pueden usar los mismos que para grafos dirigidos: matrices o listas de adyacencias.
- Una arista no dirigida entre v y w se representa mediante dos aristas dirigidas de v a w y de w a v.
- Notar que la matriz de adyacencias es simétrica.

Árboles abarcadores de costo mínimo

Dado un grafo G = (V, A) donde cada arista (u,v) de A tiene un costo asociado 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 los costos de todas las aristas.

→ Un árbol abarcador es una forma de “recorrer” todo el grafo, sin repetir caminos, y sin dejar vértices desconectados.

Sea S un arbol abarcador de un grafo G. S es un subgrafo de G que tiene la misma cantidad de vértices y su número de aristas es (la cantidad de vértices menos uno):

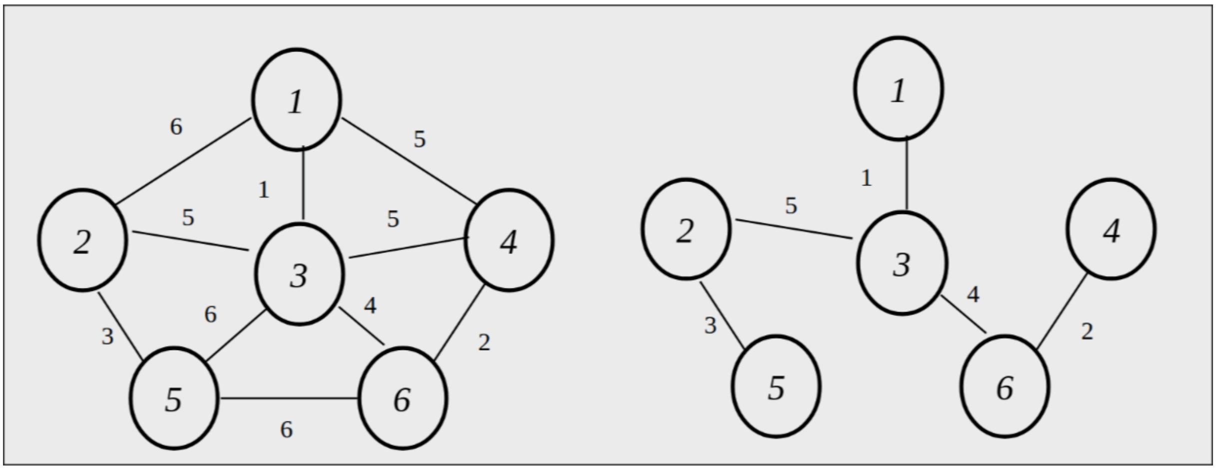
$S \subseteq G$

$S = (V', E')$

$V' = V$

$|E'| = |V| - 1$

¿Cuántos árboles abarcadores posibles hay para un grafo?

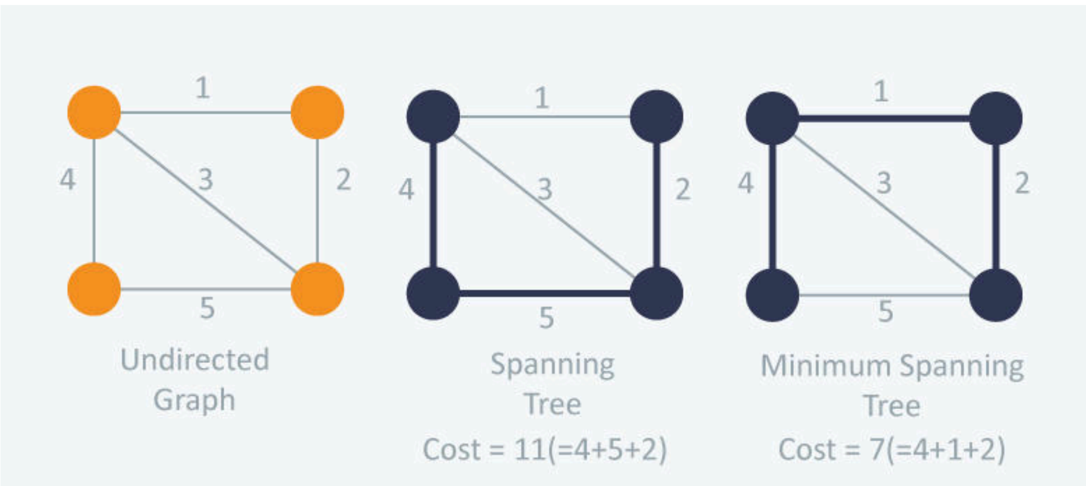


Propiedad AAM

- Cuando las aristas del grafo tienen un costo o peso, hay muchas formas posibles de construir un árbol abarcador. Pero no todas cuestan lo mismo.

El **árbol abarcador de costo mínimo** es aquel que:
- Conecta todos los vértices.
- No tiene ciclos.
- Y la suma de los pesos de sus aristas es la menor posible.

Sea G = (V, A) un grafo conexo con una función de costo definida para sus aristas. Sea U algún subconjunto propio del conjunto de vértices V.
- Si (u,v) es una arista de costo mínimo tal que u pertenece a U y v pertenece a V-U, existe un AAM que incluye a (u,v) entre sus aristas.
- Dos algoritmos hacen uso de esta propiedad: Prim y Kruskal



Algoritmo de Prim

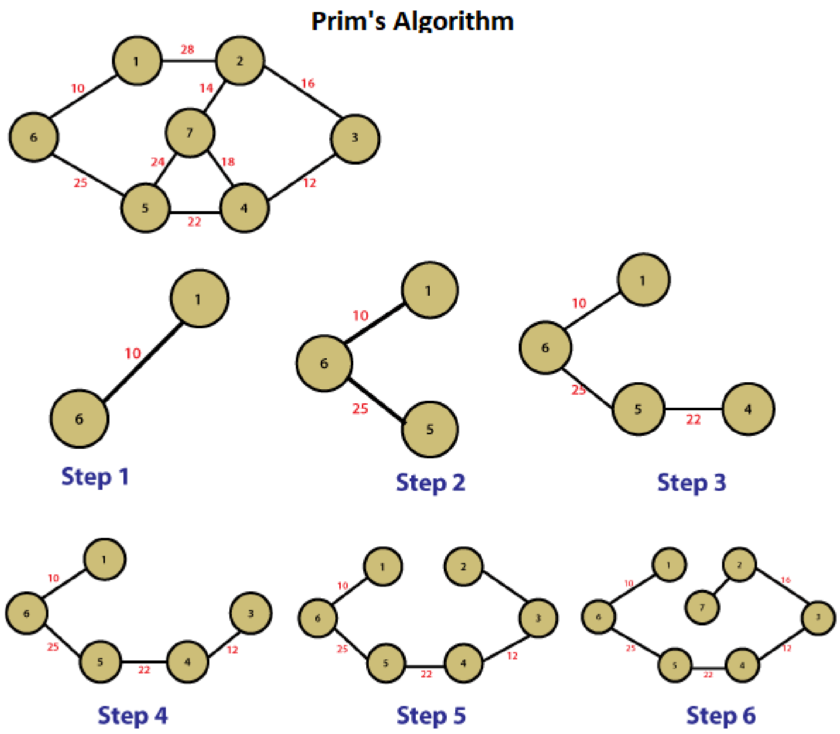
Método "ávido" (greedy), que permite construir un árbol abarcador de costo mínimo (Minimum-Cost Spanning Tree) de un grafo dado.

- G = (V , A)
- V = {1, 2, 3.....n) y una función de costo definida en las aristas de A

El algoritmo de Prim comienza cuando se asigna a un conjunto U un vértice inicial {1}, en el cual el árbol abarcador "crece" arista por arista.

- En cada paso, localiza la arista más corta (u,v) que conecta U y V-U, y después agrega v, el vértice en V, a U.
  - Este paso se repite hasta que U = V.
- Tiene una complejidad de $O(n^2)$ en su forma básica.

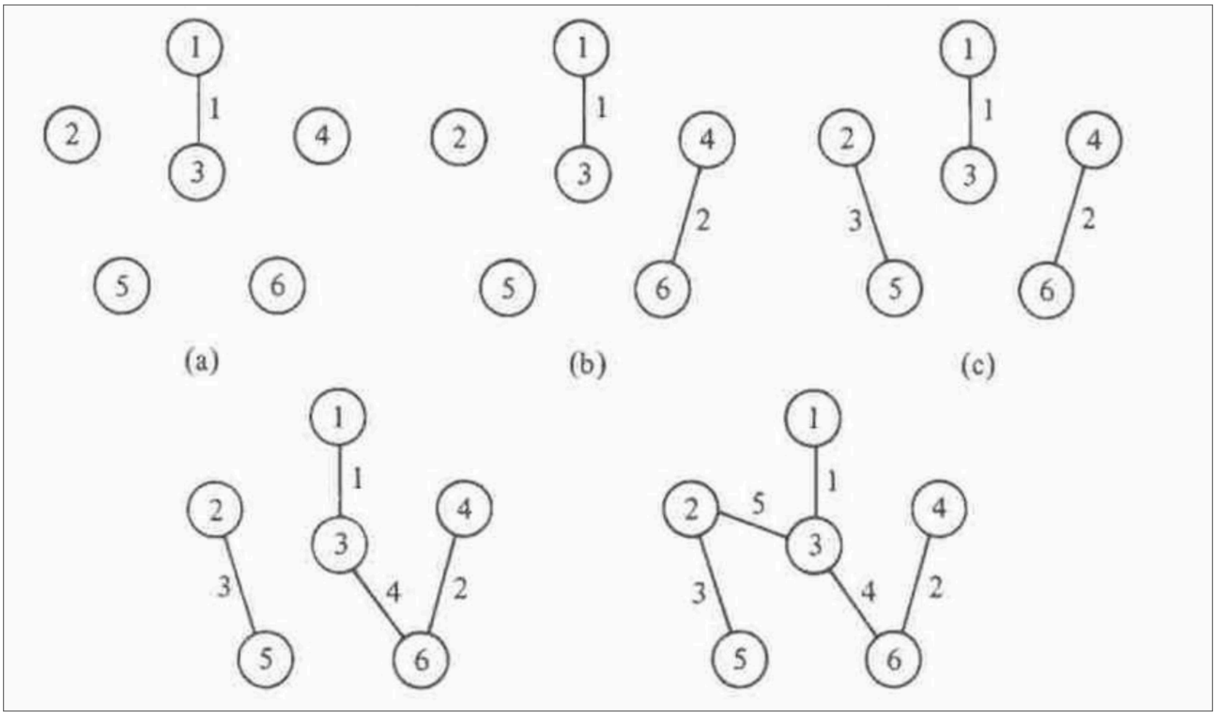
```
Método TGRAFO.Prim ( conjunto de aristas T);
U: conjunto de vértices;
u, v: vértice;
// el TGRAFO representado por un  conjunto de vértices V y un conjunto de Aristas A
COMIENZO
  T.Vaciar();
  U.Agregar(1);
  MIENTRAS U <> V hacer
    elegir una arista (u,v) de costo mínimo
    tal que u está en U y v está en V-U;
    T.agregar (u,v);
    U.agregar(v);
  FIN MIENTRAS
FIN;
```



Algoritmo de Kruskal

Otro método ávido. “Siempre seleccionar una arista de costo mínimo, excepto si esta genera un ciclo”

1. Empiezo con un grafo T = (V, ∅), sólo con los vértices de G y sin aristas.
2. Al avanzar, habrá siempre una colección de componentes conexos
3. Para cada componente se seleccionarán las aristas que formen un árbol abarcador.
4. Para construir componentes cada vez mayores, se agrega la arista de costo mínimo que conecte dos componentes distintos.
    - La arista se descarta si conecta dos vértices que están en el mismo componente conexo, pues crearía un ciclo.
5. Cuando todos los vértices están en un sólo componente, T es un árbol abarcador de costo mínimo para G.




```
Método TGrafo.Kruskal;
F conjunto de aristas;
COM
  F.Vaciar;
  Repetir:
    Elegir una arista de costo mínimo que no esté en F ni haya sido elegida;
    Si arista no conecta dos vértices del mismo componente entonces agrego a F;
  hasta que todos los vértices estén en un solo componente;
FIN
```

Tiempo de ejecución del algoritmo: $O(a \log a)$, donde a es el número de aristas del grafo.

Búsqueda en profundidad (Depth-First Search)

Véase Resumen UT7

- Mismo algoritmo que para grafos dirigidos.
- En este caso, si el grafo es conexo, de la búsqueda en profundidad se obtiene un sólo árbol.
- Para grafos no dirigidos, hay dos clases de arcos:
  - De árbol
  - De retroceso.

1. Comienzo por elegir un vértice inicial (v). Desde allí, exploro recursivamente todos los vértices adyacentes no visitados.
2. Cuando encuentro un nuevo vértice (w), detengo momentáneamente la exploración del vértice actual (`v`), y me concentro en `w`, como si abriera un nuevo camino.
3. Este proceso se repite: en cada nuevo vértice, suspendo al anterior y avanzo más profundo en el grafo, siempre priorizando avanzar lo más lejos posible por un camino antes de retroceder.
4. Cuando llego a un vértice sin adyacentes no visitados, vuelvo atrás (retrocedo en la recursión) y retomo la exploración desde donde la había dejado en el vértice anterior.

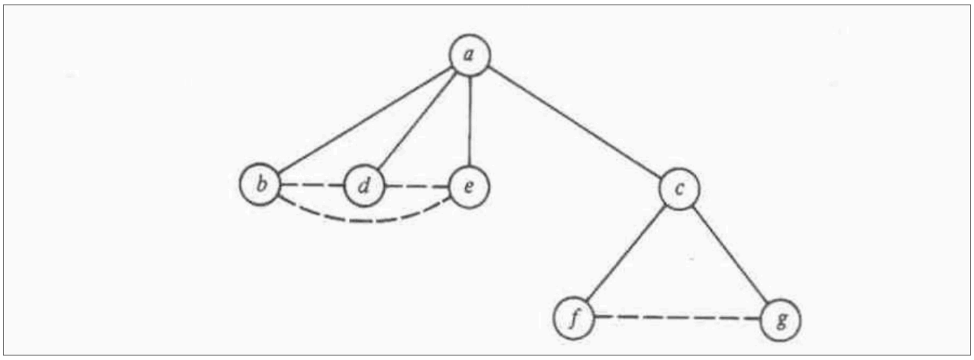
Tiempo de ejecución: $O(a)$, donde a es el número de aristas del grafo.

Búsqueda en amplitud (Breadth-First Search)

La búsqueda en amplitud es una forma de recorrer un grafo, explorando todos los adyacentes de un vértice antes de avanzar a los siguientes niveles. En grafos no dirigidos:

- Las aristas se clasifican como de árbol (si llevan a un vértice nuevo) o cruzadas (si conectan vértices ya visitados que no son ancestros entre sí).
- Se construye un bosque abarcador, con una arista por cada descubrimiento de vértice nuevo.

Para evitar repetir vértices, cada uno se marca como visitado antes de colocarlo en la cola.



1. Comienzo seleccionando un vértice inicial (`v`) y visito todos sus adyacentes antes de seguir. EEs decir, eploro en "capas", primero los más cercanos, luego los que están a dos pasos, luego a tres, y así sucesivamente.
2. Cuando descubro los adyacentes de un vértice, los agrego a una cola, y luego proceso uno por uno en el orden en que fueron agregados.
   - Este orden es importante: garantiza que se exploren los vértices en el mismo orden en que se encuentran a partir del vértice inicial.
3. A diferencia de DFS, no suspendo la exploración de un vértice para irme por uno nuevo. En BFS, termino de explorar todos los adyacentes del vértice actual, y luego paso al siguiente vértice en la cola.

```
Método Tvertice.bea() : String
// bea visita todos los vértices conectados a 'this' usando busq en amplitud

Variables:
    C : ColaDeVértices
    x, y : Vértice
    tempstr : String ← ""

Inicio:
    this.Visitar()
    C.Insertar(this)
    tempstr ← tempstr + this.etiqueta

    mientras no C.vacía() hacer:
        x ← C.Eliminar()  // x toma el vértice al frente de la cola

        para cada vértice y adyacente a x hacer:
            si no y.Visitado() entonces:
                y.Visitar()
                C.Insertar(y)
                tempstr ← tempstr + y.etiqueta
            fin si
        fin para
    fin mientras

    devolver tempstr
Fin
```

Tiempo de ejecución: $O(a)$, donde a es el número de aristas del grafo.

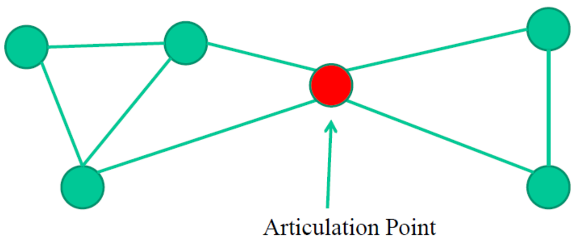
Puntos de articulación y componentes biconexos

Punto de articulación

Es un vértice v tal que, cuando se elimina, junto con todas las aristas incidentes sobre él, se divide un componente conexo en dos o más partes.

- A un grafo sin puntos de articulación se le llama **"grafo biconexo"**.
- Un grafo tiene conectividad k si la eliminación de k-1 vértices cualesquiera no lo desconecta.

→ La búsqueda en profundidad es muy útil para encontrar los componentes biconexos de un grafo.



Algoritmo para encontrar puntos de articulación

Pasos del algoritmo

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:

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

Condiciones para ser punto de articulación

- La **raíz** del árbol DFS es punto de articulación si tiene **dos 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.