Algoritmos avaiciosos (greedy) para minimum spanning tree.

(1) Preliminares sobre grafos

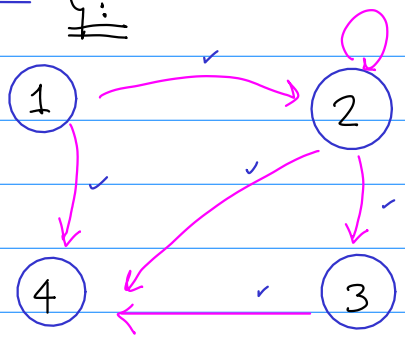
Def: Un grafo dirigido es una pareja $(V(G), E(G))$ donde

$V(G)$ — Conjunto finito llamado los vértices de G

$E(G) \subseteq V(G) \times V(G)$

↖ Colección de parejas de vértices llamadas aristas (v_1, v_2) se lee "arista de v_1 a v_2"

Ejemplo: G:



$V(G) = \{1, 2, 3, 4\}$

$E(G) = \{(1,2), (2,4), (2,3), (1,4), (3,4), (2,2)\}$

¿Cómo se representa un grafo en un computador?

Hay dos maneras comunes, dependiendo del objetivo una o la otra pueden ser más útiles

(i) Matriz de adyacencia

	1	2	3	4
1	0	1	0	1
2	0	1	1	1
3	0	0	0	1
4	0	0	0	0

(ii) Lista de adyacencia:

Para cada vértice v definimos

$$Out(v) = \{a \in V(G) : (v, a) \in E(G)\}$$

↑ vértices a los que puedo llegar con flechas que SALEN de v.

En lista de adyacencia representamos a G como un "diccionario"

$\{ v_1 : [\text{— } Out(v_1) \text{ —}]$
$v_2 : [\text{— } Out(v_2) \text{ —}] \}$
$\vdots$

Ejemplo: $\{ 1 : [2,4]$
$2 : [2,3,4]$
$3 : [4]$
$4 : [] \}$ = Lista(G).

¿Por qué son importantes los grafos?

Porque sirven para modelar relaciones binarias entre objetos y estas relaciones son MUY comunes como muestran los siguientes ejemplos:

Ejemplo 1: Redes físicas : T:

$V(T)$ = "Ciudades del mundo"

$(c_1, c_2) \in E(T) \iff$ "hay al menos un vuelo de c_1 a c_2 cada día"

Ejemplo 2: Grafos de conocimiento A

$V(A)$ = "Actores de cine"

$(a_1, a_2) \in E(A) \iff$ "a_1 y a_2 actúan en al menos una película"

World Wide Web W

$V(W)$ = "páginas web"

$(p, q) \in E(W) \iff$ "hay un link de p a q"

Ejemplo 3: Grafos de planeación: R

$V(R)$ = "Posibles configuraciones de un cubo de rubik"

$(p_1, p_2) \in E(R) \iff$ es posible pasar de a a b mediante un movimiento básico.

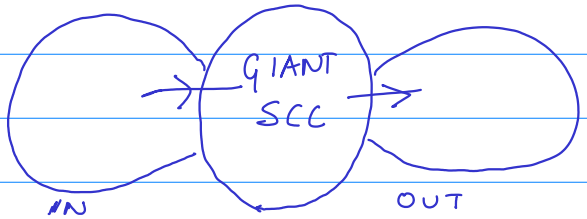
¿Qué podemos aprender de organizar la información mediante grafos? Mucho...

www.oracleofbacon.org /

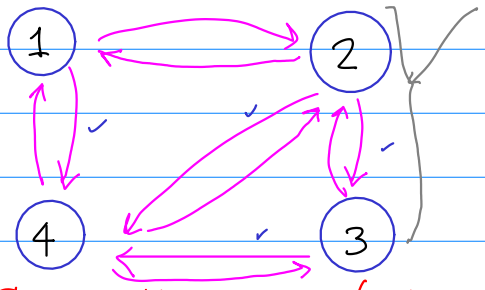
¿Cómo imaginarse el INTERNET?

$|V| = 200 \times 10^6$ millones

$|E| = 1.5 \times 10^9$ billones

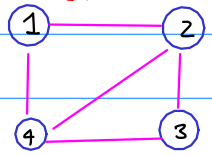


Def: Un grafo es "no dirigido" y "sin loops" si $\forall (a,b) \in E(G)$, $(b,a) \in E(G)$ y $a \neq b$



Las flechas no aportan nada así que las borramos así:

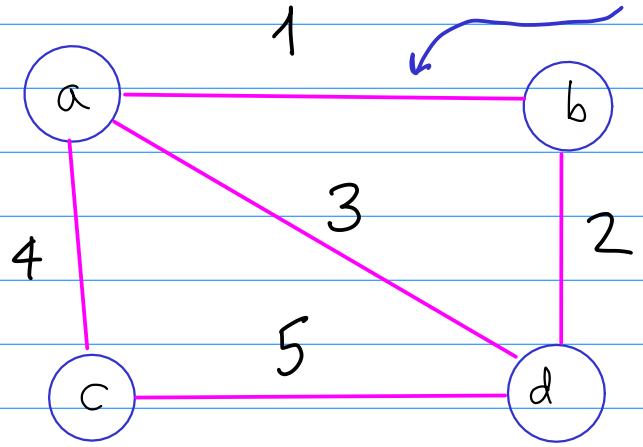
$\{v,u\} \in E(G)$



Def: Un grafo $(V(G), E(G))$ se dice "con pesos positivos en las aristas" si lo dotamos de una función de costos

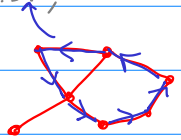
$c : E(G) \longrightarrow \mathbb{R}$.

$c(\{a,b\}) = 1$



Def: Un árbol generador para G es un conjunto T de aristas que cumple dos propiedades:

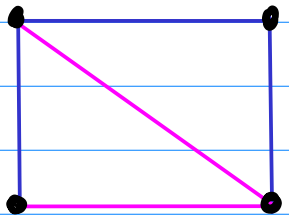
(1) T NO contiene ningún ciclo (Recuerda que un ciclo es una sucesión de k aristas distintas $e_1 = (v_0, v_1)$, $e_2 = (v_2, v_3)$, ..., $e_k = (v_{k-1}, v_k)$ con $v_0 = v_k$ para algún k)



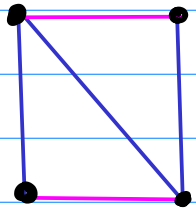
(2) T debe "generar" G en el sentido en que $\forall a, b \in V(G)$ debe existir un camino desde a hasta b usando sólamente aristas del conjunto T.

Ejemplos:

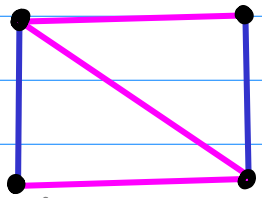
T



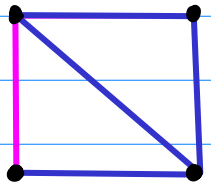
T



NO EJEMPLOS



T árbol pero no genera

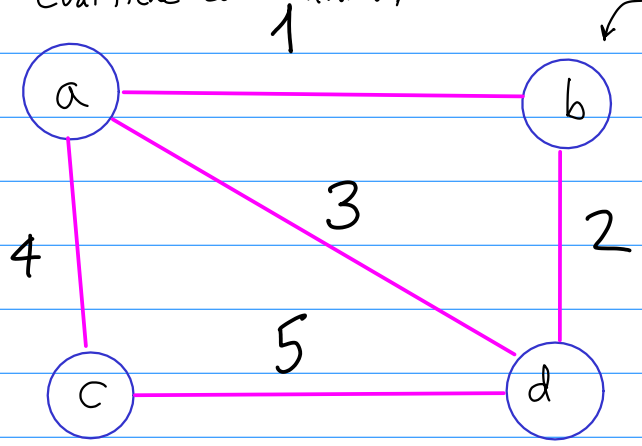


T genera pero no es árbol. (contiene ciclos)

Definimos el costo de un árbol generador $T \subseteq E(G)$ como

$$c(T) = \sum_{e \in T} c(e)$$

Ejercicio: Enumere todos los árboles generadores de G. Cuál tiene costo mínimo?



Problema [MST = Minimum spanning tree]

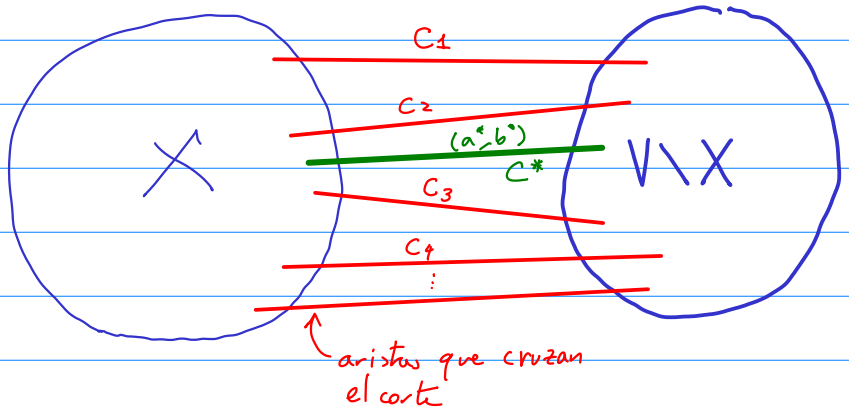
Sea G un grafo (CONEXO) no dirigido con costos en las aristas $c: E(G) \longrightarrow \mathbb{R}$. El MST es el siguiente problema de optimización:

$$MST(G) = \min \{ c(T) : T \subseteq E(G) \text{ es un árbol generador para } T \}$$

Queremos: (i) encontrar el costo mínimo $MST(G)$
(ii) Encontrar un árbol generador T^* de costo mínimo.

Algoritmo I: [Robert C. Prim 1957]

Idea: X – vértices hasta ahora analizados
T – aristas seleccionadas hasta ahora



$c^* := c((a^*, b^*))$ donde (a^*, b^*) es la arista de costo mínimo entre las que unen algún vértice de X con alguno de $V \setminus X$

Actualizamos:

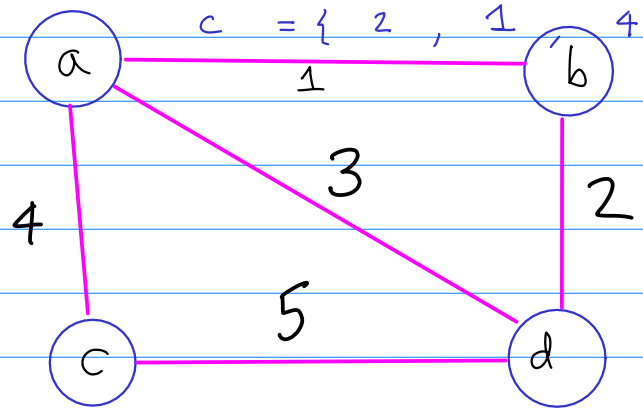
$$X \leftarrow X \cup \{b^*\}$$
$$T \leftarrow T \cup \{(a^*, b^*)\}.$$

El algoritmo de Prim inicia con $X = \{s\}$ (vértice arbitrario), $T = \emptyset$ y realiza la actualización de arriba

Ejemplo: $X = \{d, b, a, c\}$

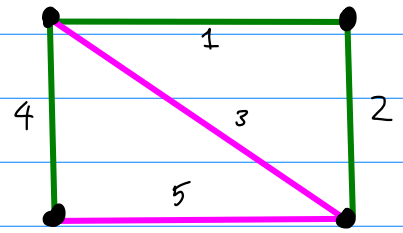
$T = \{(d,b), (a,b), (a,c)\}$

$c = \{2, 1, 4\}$



Concluimos que $MST(G) = 2+1+4 = 7$

y que T^*:



Obs: Una implementación sencilla hace lo siguiente:
dado X encuentra las aristas que van de X a $V \setminus X$ ($\leq m$ pasos).
Hay que hacer esto 1 vez por vértice, es decir en $\leq n$ pasos.

Overall complejidad $\leq nm \in O(mn)$

Haremos una implementación mucho mejor del mismo algoritmo mediante una estructura de datos adecuada, un priority queue.

Def: Un priority queue H es una colección de (key, value) pairs donde los keys estan totalmente ordenados

y permiten las siguientes operaciones: muy eficientes (en $O(\log N)$ donde N es el número de datos):

(1) H.extract_min() ← retorna la pareja (k^*, v^*) donde k^* es el mínimo key en H y quita el item (k^*, v^*) de H.

(2) H.insert((k,v)) ← adiciona (k,v) al queue.

(3) H.delete(p) ← elimina la pareja (k,v) a la que apunta p.

Son MUY, MUY útiles. Por ejemplo si queremos ordenar una lista L podemos usar **Heapsort**:

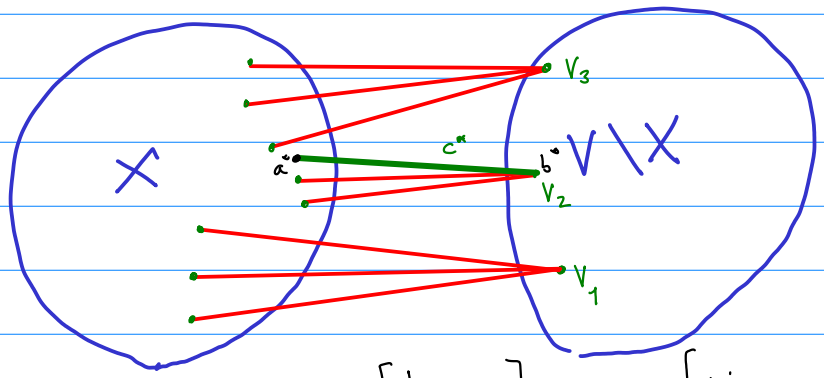
(i) insertamos los elementos de L en H $O(N \log(N))$

(ii) llamamos H.extract_min() N veces $O(N \log(N))$

Tiempo total: $O(N \log N)$

Es posible usar un priority queue para hacer una implementación mucho más eficiente del algoritmo de Prim.

Idea: H será $(key(v), v)$ donde $key(v) = \min \{ c(x,v) : x \in X \}$



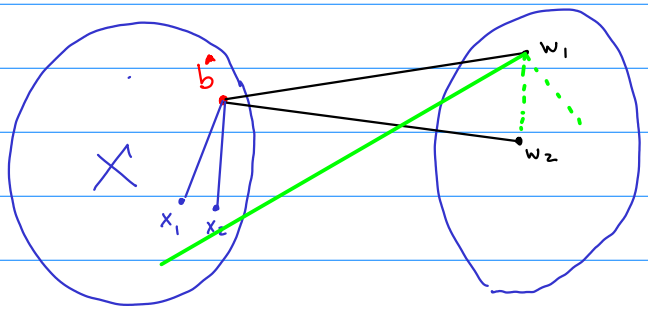
$\min_{(a,b): a \in X, b \in V \setminus X} c(a,b)$

Notar que $c^* = \min_{v \in V \setminus X} [key(v)] = \min_{v \in V \setminus X} \left[\min_{(x,v): x \in X} c(x,v) \right]$

↑ EXTRACT_MIN!

El único problema es que, al encontrar la mejor arista (a^*, b^*) con $c^* = c(a^*, b^*)$ los keys cambian. Cómo deberíamos actualizarlos?

$X := X' \cup \{b^*\}$



$w_1 \notin X$ Las aristas que terminan en w_1 se dividen en:

(i) ——— entre w_1 y X' — en key (w_1)

——— entre w_1 y b^* ← cambios:

- - - - - entre w_1 y $(V \setminus X') \setminus \{b^*\} = V \setminus X$ } no importa

Redefinimos (borrando w_1 e insertándolo de nuevo)

$$key(w_1) = \min\left(key(w_1),\ c(b^*, w_1) \right)$$

para todo vértice w_1 adyacente a b^* por fuera de X'.

Cuánto trabajo realizamos?

En cada paso: (1) H.extract_min() $O(\log n)$

(2) Actualizamos los vértices adyacentes a b^* $O(d_{b^*} \log(n_1))$

Luego en total:

$$n \log(n_1) + \sum_{b \in V(G)} d_b \log(n_1) = O\left((n+m)\log(n_1)\right)$$

Casi tan fácil como leer la información que depone el grafo!

(FOR FREE PRIMITIVE)