¿Por qué funciona el algoritmo de Prim?

(*) Por simplicidad asumiremos que todos los costos son distintos

Seguiremos la demostración de validez de Roughgarden, que consiste de dos pasos:

(1) Verificamos que el árbol generador producido por Prim satisface la MBP ("Minimum Bottleneck Property") que definimos abajo y

(2) Mostraremos que, bajo (*) un árbol generador con MBP debe ser de costo mínimo (un MST).

Def: Sea G un grafo con pesos en las aristas. Si $P = v_1, v_2, \ldots, v_k$ es un camino en G (es decir si $(v_i, v_{i+1}) \in E(G)\ \forall i$) definimos

$$b(P) = \max_{1 \le i \le k-1} \left[c(v_i, v_{i+1}) \right]$$

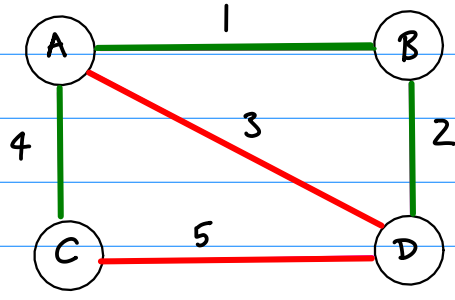
$b(P)$: ancho o "bottleneck" de P

Def: Una arista $(v, w) \in E(G)$ satisface el "minimum bottleneck property" MBP si

$$c(v,w) = \min \{ b(P) : P \text{ es un camino de } v \text{ a } w \}$$

En otras palabras si NO hay un camino de v a w que use sólo aristas e con $c(e) < c(v,w)$.

Ejemplo: Cuáles aristas satisfacen MBP?



(A,D) NO porque

$b(A,B,D) = \max(1,2)$

$c(A,D) = 3$

(A,B) Sí porque

$b(A,D,B) = 3 > 1$

$b(A,C,D,B) = 5 > 1$

Inmediato del ejercicio de árbol en [P1]

con $n = |V(G)|$ y $m = |E(G)|$

Lema: Sea G un grafo conexo, $T \subseteq E(G)$ con $|T| = n-1$ entonces las siguientes son equivalentes

(1) (V,T) es conexo

(2) (V,T) es acíclico (es decir (V,T) no contiene ningún ciclo)

Adicionalmente todo árbol generador tiene exáctamente $n-1$ aristas.

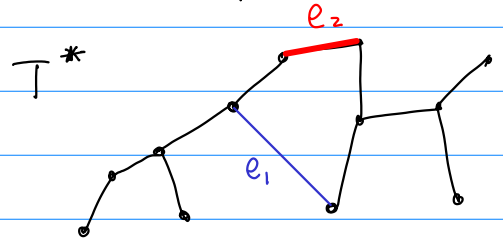
[MBP en toda arista $\Rightarrow$ óptimo]

Lema: Si toda arista de T satisface MBP entonces T es un árbol generador mínimo.

Dem: Suponga lo contrario, entonces existe un árbol generador $T^* \neq T$ con $c(T^*) < c(T)$.

Todo árbol generador tiene $n-1$ aristas, si $T^* \neq T$ entonces hay alguna arista $e_1 \in T \setminus T^*$

Uniendo e_1 a T^* creamos un ciclo que contiene a e_1



usamos que los costos son $\neq$ s.

Como $e_1 \in T$ y toda arista de T satisface la MST existe $e_2 \in T^*$ con $c(e_2) > c(e_1)$ a lo largo del nuevo ciclo

Definimos $T' = (T^* \setminus e_2) \cup e_1$

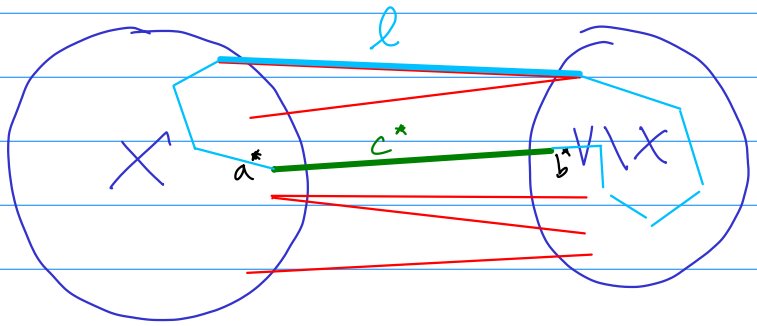
T' es conexo, tiene n vértices y $n-1$ aristas luego es un árbol generador para G

$$c(T') = c(T^*) - c(e_2) + c(e_1) < c(T^*)$$

lo que contradice la optimalidad de T^*.

Lema: Si T resulta de aplicar el algoritmo de Prim entonces MBP se cumple para toda arista $e \in T$.

Dem:



Sea P cualquier otro camino de a^* a b^*

$b(P) \geq \ell \geq c^*$ (podemos asumir que las desigualdades son estrictas)

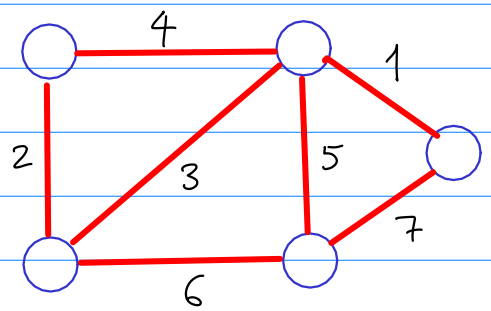
así que (a^*, b^*) satisface la MBP

Como toda arista de Prim se construye de esta manera hemos demostrado la afirmación.

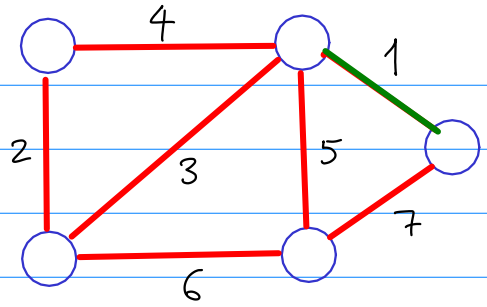
(2) Algoritmo de Kruskal

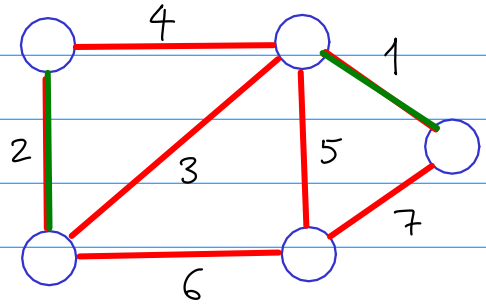
En el algoritmo de Kruskal empezamos con n vértices aislados y adicionamos aristas una a una. En cada paso escogemos la arista disponible de mínimo costo si al unirla con las que ya tenemos NO SE FORMA un ciclo.

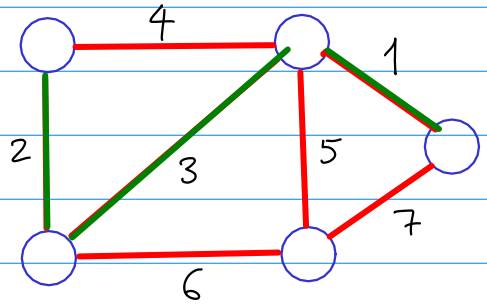
Ejemplo:



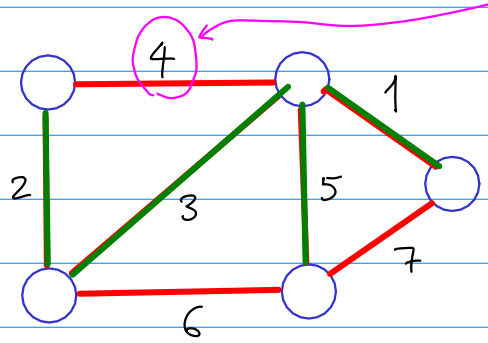
Ejemplo:







NO seleccionamos esta pues formaría un ciclo.



✓

——— = MST(G)

Obs: Una implementación ingenua tiene complejidad $O(mn)$. Podemos ir mucho mejor usando una estructura de datos llamada union_find.

③ UNION-FIND

Def: La estructura de datos UNION_FIND sirve para mantener una partición (set partition) de un conjunto estático de N objetos (la partición puede ir cambiando pero el conjunto de objetos no)

F Tiene a su disposición 3 operaciones muy eficientes:

F:=INITIALIZE(X) Recibe N objetos en un arreglo X y pone cada uno en su propio conjunto $\{x\}$ (tiempo $O(N)$)

F.FIND(x) Dice cuál parte (índice) contiene al elemento x

F.UNION(x,y) = Junta las partes que contienen a los elementos x y y.

(tiempo $O(\log(N))$)

Para implementar Kruskal:

(i) Ordenamos las aristas por peso

$O(m\log(m)) = O(m\log(n))$

(ii) Inicializamos un union_find F sobre los vértices $O(n)$

(iii) Recorremos las aristas en orden creciente chequeando si creamos o no un ciclo al adicionar (a,b) mediante

F.find(a) == F.find(b) ?

$O(\log(n))$

Si son distintas unimos dos componentes así que F.union(a,b)

En total $O((n+m)\log(n))$.