MA4701. Optimización Combinatorial. 2013.
Profesor: José Soto.
Escriba(s): Stefanni Cavaletto, Juan Pablo Contreras y Christopher Urra.
Fecha: 26 de Agosto de 2013.

Cátedra 3

Al estudiar grafos, nos interesa determinar cuaǹdo un algoritmo devuelve un árbol generador de peso mínimo, comenzaremos esta cátedra estudiando un teorema que garantiza que en cada momento, el algoritmo PRIM nos devuelve un árbol de peso mínimo y que nos ayudara a probar que el algoritmo KRUSKAL también lo hace. Para esto necesitamos definir el siguiente concepto.

Definición 1. Dado G conexo, ω función de peso, $\omega : E \to \mathbb{R}$. Diremos que $F \subseteq E(G)$ es *extensible* si existe un árbol generador de peso mínimo $H = (V, T)$ tal que $F \subseteq T$.

Teorema 1. *Sea F extensible y $e \notin F$. $F + e$ es extensible si y sólo si existe un corte $\delta(U)$, $\emptyset \subsetneq U \subsetneq V$ (no trivial) tal que $F \cap \delta(U) = \emptyset$ y e es de mínimo peso en $\delta(U)$.*

Demostración:

$\Leftarrow$ | Sabemos que existe $H = (V, T)$ óptimo tal que $F \subseteq T$

Caso 1: Si $e \in T$, entonces tenemos fácilmente que $F + e \subseteq T$.

Caso 2: Si $e \notin T$, entonces $e \in \delta(U)$, además por hipótesis, e es mínimo en $\delta(U)$ y $\delta(U) \cap F = \emptyset$.

Como T es conexo, cruza $\delta(U)$ (es decir $\delta(U) \cap T \neq \emptyset$). Sea $f \in \delta(U) \cap T$, luego f pertenece al único ciclo en $(T + e)$, de esta forma $(T + e - f)$ es un árbol generador.

Solo falta probar que $(T + e - f)$ es de peso mínimo. En efecto, como $\omega(e) \leq \omega(f)$ (ya que es de peso mínimo en $\delta(U)$) tenemos que

$$\begin{aligned} \omega(T + e - f) &= \omega(T) + \omega(e) - \omega(f) \\ &\leq \omega(T) \end{aligned}$$

Luego $F + e$ es extensible pues $F + e \subseteq (T + e - f)$ que es de peso mínimo.

$\Rightarrow$ | Como $F+e$ es extensible, existe un árbol de peso mínimo $H = (V, T)$ tal que $F+e \subseteq T$. Sea U una de las componentes conexas de $T - e$, luego $e \in \delta(U)$ y $\delta(U) \cap F = \emptyset$.

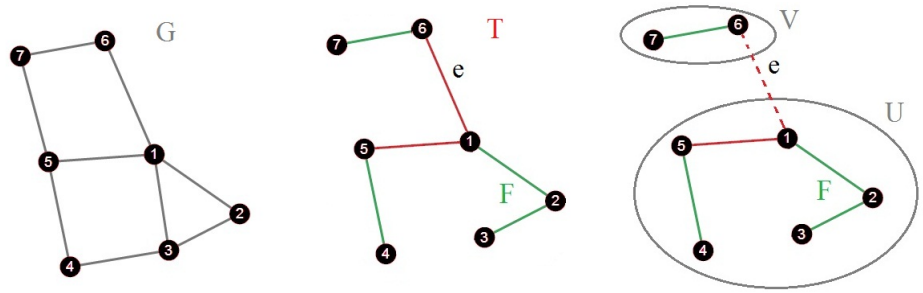


Figura 1: *Al quitar la arista e del conjunto T vemos como los cortes de las componentes conexas U y V contienen a e y no intersectan al conjunto de aristas F.*

Si existiera un $f \in \delta(U)$, con $\omega(f) < \omega(e)$, tendríamos que $(T + f - e)$ es un árbol generador y $\omega(T - e + f) < \omega(T)$, lo cual es una contradicción, ya que $H = (V, T)$ es de peso mínimo. □

Observación 1. Con el Teorema 1 , se prueba directamente que el algoritmo PRIM es correcto.

Demostración:

En cada etapa de PRIM, T es extendible y en la última etapa (cuando $\delta(U) = \emptyset$) tenemos que T es un árbol. □

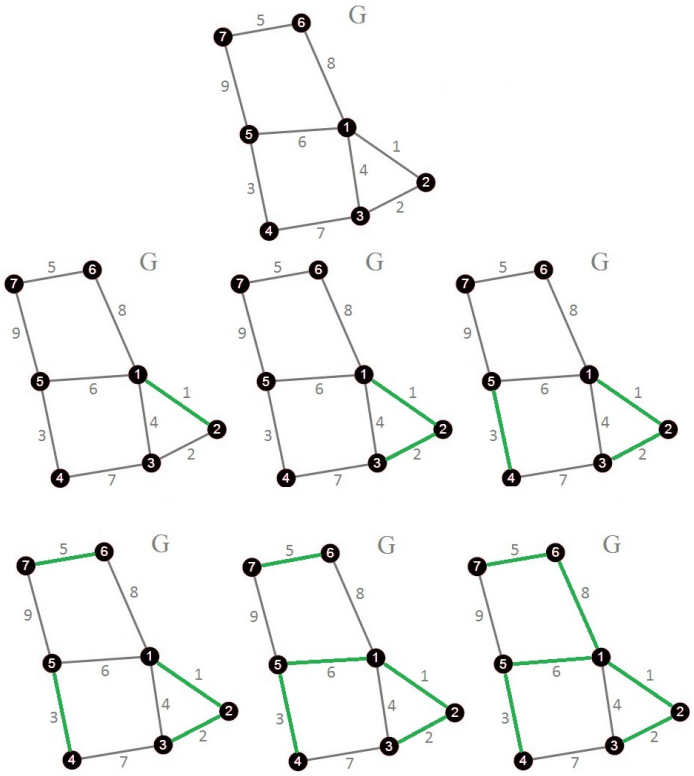
1. Algoritmo KRUSKAL

Estudiemos ahora un algoritmo "verdaderamente glotón" que también resuelve el problema de encontrar un árbol generador de peso mínimo.
A diferencia del algoritmo PRIM el algoritmo KRUSKAL elige la arista más liviana sin darle importancia a mantener la conexidad en todo momento y es por ello que decimos que "en más glotón". Se presenta ahora una implementación en pseudocódigo de KRUSKAL.

Algoritmo 1 KRUSKAL, (por Kruskal 1956)

Require: $H = (U, T)$ inicializando $U = \{r\}$ y $T = \phi$
1: **while** T no sea generador y conexo **do**
2: Elegir e la arista más liviana de $E \backslash T$ tal que $T + e$ no tiene ciclos
3: $T \leftarrow T + e$
4: **end while**
5: **return** $H = (V, T)$

Ejemplo 1. (KRUSKAL)
A continuación se presenta con un ejemplo el procedimiento que realiza el algoritmo KRUSKAL para el grafo G de la figura con los pesos de cada arista ahí especificados.



Procedimiento del algoritmo en cada iteración

- **Iteración 0:** Se considera como raíz el vértice 1 y ninguna arista.
- **Iteración 1:** Se identifica la arista 1-2 como la más barata (peso 1) y se agrega a T.
- **Iteración 2:** Se identifica la arista 2-3 como la más barata (peso 2) y se agrega a T.
- **Iteración 3:** Se identifica la arista 4-5 como la más barata (peso 3) y se agrega a T.
- **Iteración 4:** Se identifica la arista 1-3 como la más barata (peso 4), como esta arista genera el ciclo 1-2-3 no satisface la condición. Se busca la siguiente arista más barata que es la arista 7-6 de peso 5 y esta no genera ciclos por lo que se agrega a T.
- **Iteración 5:** Se identifica la arista 5-1 como la más barata (peso 6) y se agrega a T pues no genera ciclos.
- **Iteración 6:** Se identifica la arista 4-3 como la más barata (peso 7) pero esta genera el ciclo 1-2-3-4-5, la descartamos y buscamos la siguiente más liviana que resulta ser 1-6 de peso 8, esta última no genera ciclos así que se agrega a T.

Finalizada la sexta iteración tenemos un grafo $H = ([7], T)$ que resulta ser generador y conexo.

Teorema 2. *El algoritmo KRUSKAL devuelve un árbol de pesos mínimos.*

Demostración:

Probemos que T es extendible en cada momento.

1. $T = \emptyset$ es extendible trivialmente.
2. Sean $U_1, U_2, ..., U_k$ las componentes conexas de (V, T).

   Si e es tal que $T + e$ no tiene ciclos, entonces e conecta 2 componentes conexas, denominemos estas componentes por U_i y U_j. Luego $e \in \delta(U_i)$, $\delta(U_i) \cap T = \emptyset$ y además e es de peso mínimo en $\delta(U_i)$.

   Aplicando el Teorema 1, tenemos entonces que $T + e$ es extendible.
3. Al final del proceso, T es generador conexo, esto implica que T es de peso mínimo. □

Cabe destacar que la idea de este algoritmo puede ser empleada para resolver otro tipo de problemas, como por ejemplo en Álgebra para encontrar bases de espacios vectoriales.

Ejemplo 2. (Espacios vectoriales)
Sea V un espacio vectorial sobre $\mathbb{R}$, y sea $W \subseteq V$ un conjunto finito de v vectores. Encontrar $W_0 \subseteq W$, una base de $< W >$.

Para hacer esto, podemos usar un algoritmo igual al de Kruskal, en este caso modificando "tener ciclo" por "ser linealmente dependiente".

2. Notación Asintótica.

Dado que existen muchos tipos de algoritmos que resuelven un mismo problema, es ideal saber cual de todos estos es el más rápido en tiempo de ejecución. Para esto, nos interesa estudiar el comportamiento de los algoritmos con tamaños de entradas muy grandes (es decir, cuando la cantidad de entradas tiende a infinito). Además queremos dejar expresado este comportamiento en función de las entradas. De esta forma surge la idea de estudiar los algoritmos asintóticamente. Queremos formalizar el concepto de crecimiento de funciones como $n^2 \ll 2^n$, donde n representa el número de entradas de los algoritmos.

Definición 2. En esta sección, diremos que f es una función si y sólo si $f : A \to \mathbb{R}$, con $A \subseteq \mathbb{N}$, $\mathbb{N} \backslash A$ es finito.

Definiremos ahora distintos tipos de conjuntos de funciones que representaran cotas para el comportamiento asintótito de nuestro algoritmo.

Definición 3. Dado f, g funciones:

$$[f \preceq g] \quad f \in O(g) \iff \exists n_0 \in \mathbb{N}, \exists c < 0 \quad \text{tal que} \quad \forall n \geq n_0, \quad |f(n)| \leq c\,|g(n)| \iff \limsup_n \left|\frac{f(n)}{g(n)}\right| < +\infty.$$

$$[f \prec g] \quad f \in o(g) \iff \forall \epsilon > 0, \exists n_0 \in \mathbb{N} \quad \text{tal que} \quad \forall n \geq n_0, \quad |f(n)| \leq \epsilon\,|g(n)| \iff \lim_n \left|\frac{f(n)}{g(n)}\right| = 0.$$

$$[f \succeq g] \quad f \in \Omega(g) \iff \exists n_0 \in \mathbb{N}, \exists M > 0 \quad \text{tal que} \quad \forall n \geq n_0 \quad |f(n)| \geq M|g(n)| \iff \lim_n \left|\frac{f(n)}{g(n)}\right| > 0.$$

$$[f \succ g] \quad f \in \omega(g) \iff \forall M > 0, \exists n_0 \in \mathbb{N}, \quad \text{tal que} \quad \forall n \geq n_0 \quad |f(n)| \geq M|g(n)| \iff \liminf_n \left|\frac{f(n)}{g(n)}\right| = \infty.$$

$$[f \approx g] \quad f \in \Theta(g) \iff f \in O(g) \cap \Omega(g).$$

Ejemplo 3. (Orden de algunas funciones)

1. $\log(n) \in \Theta(H_n)$ con $H_n = \sum_{i=1}^{n} \frac{1}{i}$.
2. $\sin(n) \in O(1)$, $1 \notin O(\sin(n))$.
3. $n^2 \log(n) \in \omega(n^2)$.

Definición 4. (Abuso de notación)

Reemplazaremos el signo "$\in$" por "$=$" en todas las definiciones anteriores. Además, la relación "$=$" en este contexto no es simétrica y se lee de izquierda a derecha, es decir

1. $n^2 = O(n^3) \neq n^2$.
2. $n^2 + 5n = O(n^2) + O(n) = O(n^3) \neq O(n^2) + O(n)$.

Se lee $n^2 + 5n$ **es** orden n^2 más orden n que a su vez **es** orden n^3 el cual **no es** orden n^2 más orden n.

3. Complejidad de un Algoritmo

Dado que el tiempo de ejecución en diferentes computadores de un mismo algoritmo no es siempre el mismo por diversos factores (como por ejemplo el procesador, la memoria RAM, la memoria SWAP, etc.), se desea comparar diversos algoritmos de una forma en que no importe en la máquina en que sean ejecutados. Para efectos de este curso, cada operación de variables o números enteros (como la lectura, escritura, asignaciones, comparaciones, operaciones aritméticas, etc.) contará como una unidad de tiempo. La notación asintótica nos permite ignorar los cambios de escala entre estas unidades y el tiempo "real" de computo.

La definición rigurosa de modelos matemáticos de computación se dejará para futuros cursos en complejidad de algoritmos.

Ejemplo 4. Los objetos con los que trabajamos se pueden codificar usando bits (en binario).

$$k \in \mathbb{Z} \quad \longrightarrow \quad \begin{array}{c} \langle k \rangle \\ |\langle k \rangle| = \Theta(\log |k|). \end{array}$$

Donde $\langle k \rangle$ denota una secuencias de bits.

Ejemplo 5. Supongamos que queremos ordenar un arreglo de n números positivos. Se puede representar de la siguiente forma

$$n \in \mathbb{Z} \quad \longrightarrow \quad \begin{array}{c} V[n] \\ |V[n]| = O(n\log(n)) \end{array}$$

Donde n denota el número de entradas al arreglo.

Ejemplo 6. Consideremos para un grafo, las listas de incidencia que pueden ser vistas como

$$(n,m) \in \mathbb{Z} \times \mathbb{Z} \quad \longrightarrow \quad \begin{array}{c} A[n][m] \\ |A[n][m]| = O(n+2m) \end{array}$$

Donde n denota el índice de los vértices, m es el índice del vértice de incidencia y A es la matriz de incidencia.

Definición 5. Un algoritmo que toma como entrada una secuencia de N números enteros (digamos de tamaño total en binario S), y devuelve un objeto demorandose T unidades de tiempo, se dice:

Polinomial: Si $\exists c > 0$ tal que $T = O(S^c)$.

Fuertemente Polinomial: Si $\exists c > 0$ tal que $T = O(N^c)$.

Ejemplo 7. La complejidad del algoritmo PRIM es $O(n^2)$ por lo tanto PRIM es fuertemente polinomial.

Los conceptos de polinomial o fuertemente polinomial representan una medida de eficiencia del algoritmo en relación al tamaño de la entrada. Si P es polinomio y c una constante entonces $P(cn) = O(P(n))$, es decir, al amplificar la cantidad de entradas por una constante el algoritmo demora un tiempo proporcional al aumento de la entrada. En contraste si suponemos algoritmos con ordenes exponenciales, por ejemplo $T(n) = 2^n$ vemos como $T(cn) = O(T(n)^c)$ lo que indica que un aumento en la entrada produce un aumento mucho mayor en tiempo.