MA4701. Optimización Combinatorial. 2013.
Profesor: José Soto
Escriba(s): Martin Castillo, Gianluca Carniglia
Fecha: 30 de Agosto 2013

Cátedra 4

1. Tema I: Complejidad de un Algoritmo

1.1. Recuerdo

Definición 1 (Orden Asintótico)**.** Sean f y g funciones, entonces:

$$f \in O(g) \Leftrightarrow \exists n_0 \in \mathbb{N},\ \exists c > 0 \text{ tal que } \forall n \geq n_0 :\ |f(n)| \leq c|g(n)| \Leftrightarrow \limsup_n \frac{|f(n)|}{|g(n)|} < +\infty$$

$$f \in o(g) \Leftrightarrow \forall \epsilon > 0,\ \exists n_0 \in \mathbb{N} \text{ tal que } \forall n \geq n_0 :\ |f(n)| \leq \epsilon|g(n)| \Leftrightarrow \lim_{n\to\infty} \frac{|f(n)|}{|g(n)|} = 0$$

Recordemos que la entrada de un algoritmo típicamente está codificada como una secuencia de N números los cuales son escritos usando S bits.
Por ejemplo un algoritmo puede recibir como entrada una matriz de adyacencia de un grafo y los pesos asociados a cada arista. Esta información es codificada usando $N = n^2 + m$ números (las n^2 entradas de A y los m pesos). El largo total de dichos números escritos en binario es S).

Definición 2. Un algoritmo es **polinomial o débilmente polinomial** si el tiempo T que demora en devolver un objeto es polinomial con respecto a S.

$$T = O(S^c), \text{ con } c > 0.$$

Definición 3. Un algoritmo es **fuertemente polinomial** si el tiempo T que demora en devolver un objeto es polinomial con respecto a N.

$$T = O(N^c), \text{ con } c > 0.$$

1.2. Tiempos polinomiales

Supongamos que $P(x) = x^k$. Luego si tomamos una constante $c > 0$ tendríamos que $P(cx) = c^k P(x)$. Y así:

$$P(c\ \cdot) \in O(P(\cdot)),$$

o abusando de la notación:

$$P(cn) = O(P(n)) = O(n^k).$$

Del análisis anterior se puede deducir lo mismo para cualquier polinomio $P(x) = \sum_{i=o}^{k} a_i x^i$, pues

$$P(cn) \in O(a_k c^k n^k + \ldots + a_0) = O(n^k).$$

Esto es bastante importante pues si el tamaño de la entrada se multiplica por una constante, el tiempo de ejecución también se multiplica por a lo más una constante (no necesariamente igual a la primera). Ahora, supongamos que el tiempo T que un algoritmo demora en devolver un objeto de n números de entrada es $T(n) = n^k$. Si al mismo algoritmo le damos cn números de entrada (con $c \in \mathbb{N}$), entonces este demoraría $T(cn) = c^k n^k = c^k T(n)$. Así para una entrada de números c veces más grande este demoraría c^k veces (no depende de n) más tiempo que si el tamaño de la entrada fuera n.

Un caso no polinomial

Suponiendo $T(n) = 2^n$, entonces $T(2n) = T(n)^2$.

Comentario

Si suponemos cierta la *Ley de Moore* que sostiene que la capacidad de procesamiento de las computadoras se duplica cada 18 meses, entonces con los algoritmos polinomiales podríamos, cada 18 meses, resolver problemas el doble de grandes en la misma cantidad de tiempo que hace 18 meses atrás.

Definición 4. El tiempo (T) de un algoritmo es el número de operaciones realizadas sobre enteros de tamaño acotado:

- Asignaciones de variables.
- Comparaciones.
- Operaciones aritméticas.

Más detalles sobre el tiempo que demora cada operación realizada sobre enteros se deja para un curso de Complejidad.

1.3. Complejidad de algunos Algoritmos

1.3.1. Complejidad de PRIM

Procedamos a calcular la complejidad de PRIM dependiendo de las distintas formas en las que se implemente este algoritmo. Primero recordemos PRIM:

Algoritmo 1 Algoritmo de Prim (Prim 1957 – Jarník 1930)

1: Dado $G = (V, E)$ conexo
2: Sea $r \in V$
3: Iniciar $H = (U, T)$
4: $U = \{r\}$
5: $T = \emptyset$
6: **while** $\delta(U) \neq \emptyset$ **do**
7: Sea $e = uv \in \delta(U)$, $u \in U$, $v \notin U$ tal que e es la arista de menor peso en $\delta(U)$
8: $U \leftarrow U + v$
9: $T \leftarrow T + e$
10: **end while**
11: Devolver $H = (U, T)$

Primera forma de implementar PRIM

1: Dado $G = (V, E)$ conexo
2: Sea $r \in V$
3: Iniciar $H = (U, T)$
4: $U = \{r\}$, $T = \emptyset$ // Esto toma $O(1)$.
5: **for** $i = 1 : n - 1$ **do**
6: Pasar por todas las entradas de la matriz de adyacencia para encontrar $e_i = u_i v$ de menor peso en $E[U, V \setminus U]$ // Esto toma $O(n^2)$.
7: $U \leftarrow U + v$
8: $T \leftarrow T + e_i$
9: **end for** // El **for** se realiza n veces, entonces toma $O(n)$.
10: Devolver $H = (U, T)$

$$\text{Total } 1 = O(1) + O(n)O(n^2) = O(n^3).$$

Segunda forma de implementar PRIM

1: Dado $G = (V, E)$ conexo
2: Sea $r \in V$
3: Iniciar $H = (U, T)$
4: $U = \{r\}$, $T = \emptyset$ // Esto toma $O(1)$.
5: **for** $i = 1 : n - 1$ **do**
6: Pasar por todas las aristas incidentes a u_i para encontrar $e_i = u_i v$ de menor peso. En $E[U, V \setminus U]$ // Esto toma $O(m)$.
7: $U \leftarrow U + v$
8: $T \leftarrow T + e_i$
9: **end for** // $O(n)$
10: Devolver $H = (U, T)$

$$\text{Total 2} = O(1) + O(n)O(m) = O(nm).$$

La segunda forma de implementar PRIM es un poco mejor que la primera, pues $m \leq n^2$ y $m = n^2$ solo si G es completo. Así por ejemplo si G fuera un árbol entonces Total 2= $O(n^2)$.

Tercera forma de implementar PRIM

Antes de empezar definimos para $v \notin U$: $\text{cand}(v)$ como la arista más liviana de $E(U, v)$, suponiendo que esta existe.

1: Dado $G = (V, E)$ conexo
2: Sea $r \in V$
3: Iniciar $H = (U, T)$
4: $U = \{r\}$, $T = \emptyset$ // $O(1)$
5: Calcular $\text{cand}(v)$ para todo $v \notin U$ // $O(m) \leftarrow$ se miran todas las aristas.
6: **for** $i = 1 : n - 1$ **do**
7: Se elige el mejor candidato $(\min\limits_{v \notin U} \text{cand}(v) = u_i v_i = e_i)$ // $O(n)$.
8: $U \leftarrow U + u_i$
9: $T \leftarrow T + e_i$
10: Recalcular $\text{cand}(w)$ para todo $w \notin U$ // Esto toma $O(n)$.
11: **end for** // $O(n)$
12: Devolver $H = (U, T)$

$$\text{Total 3} = O(m + n^2) = O(n^2).$$

Es importante señalar la razón por la que el paso 10 toma $O(n)$: cada vez que recalcula $\text{cand}(w)$ con $w \notin U$ toma $O(1)$, pues solo basta comparar el antiguo $\text{cand}(w)$ con la nueva arista $u_i w$ (si es que existe) y tomar la mas liviana. Así en total recalcularlos todos toma $O(n)$.

Otras implementaciones de PRIM

Las mejores implementaciones conocidas de PRIM son de orden:

- $O((n + m)\log(n))$ en una estructura de datos llamada *Heaps Binarios*.
- $O(m + n\log(n))$ en una estructura de datos llamada *Heaps de Fibonacci*.

1.3.2. Complejidad de KRUSKAL

Recordemos KRUSKAL:

Algoritmo 2 KRUSKAL

1: Dado $G = (V, E)$ conexo, $w : E \to \mathbb{R}$
2: $T = \emptyset$
3: **while** T no sea generador **do**
4: Sea $e = uv$ la arista mas liviana de $E - T$ tal que $T + e$ no tiene ciclos
5: $T \leftarrow T + e$
6: **end while**
7: Devolver $H = (V, T)$

Implementación de KRUSKAL

1: Dado $G = (V, E)$ conexo, $w : E \to \mathbb{R}$
2: Ordenar E de menor a mayor peso // $O(m \log(m))$
3: $T = \emptyset$
4: **for** $i = 1 : m$ **do**
5: **if** $T + e_i$ es acíclico **then**
6: $T \leftarrow T + e_i$
7: **end if** // $O(|V(T + e_i)| + |E(T + e_i)|) = O(n)$ sabiendo que tengo un bosque (T) y comprobando si e_i cierra un ciclo
8: **end for** // $O(m)$
9: Devolver $H = (V, T)$

$$\text{Total} = O(m \log(m) + mn) = O(mn).$$

Implementación por partes conexas

Una mejor implementación de KRUSKAL es un algoritmo que identifica las componentes conexas de T y si e_i no conecta dos componentes conexas entonces genera un ciclo. Esta implementación se puede demostrar que tiene orden $O(m \log(m)) = O(m \log(n^2)) = O(m \log(n))$.

Implementación de Chazelle

Esta es la mejor implementación conocida de KRUSKAL y tiene orden $O(m\alpha(n))$. Donde $\alpha(n)$ es la inversa de Ackermann. Para hacernos una idea de cuan lento crece esta función ocupemos el hecho de que:

$$\alpha(n) = o(\log^*(n)).$$

Donde $\log^* : \mathbb{R}_+ \longrightarrow \mathbb{N}$,
$\log^*(n)$ =número de veces que hay que aplicar $\log_2$ a n para llegar a un número menor que 2.
Para hacernos una idea de cuan lento crece esta función notemos que

$$\log^*(2^{2^{2^2}}) = \log^*(65536) = 4.$$

Por otro lado

$$\alpha(2^{2^{65536}} - 3) = 4, \text{ donde } 2^{2^{65536}} - 3 \approx 2^{2^{10^{19729}}},$$

es el número más grande que da 4. En concreto para cualquier n que nos pueda salir $\alpha(n) < 5$. Así podemos decir que en la práctica $\alpha(n)$ es constante y que la complejidad de KRUSKAL es "prácticamente" lineal en m.

2. Tema II: Algoritmos Glotones

En general, un algoritmo glotón es aquel que busca un óptimo local en cada etapa, con esperanza de encontrar un óptimo global. Anteriormente vimos que KRUSKAL es un algoritmo glotón que busca dentro de todo el conjunto de aristas aquella de menor peso. Y también recordamos un algoritmo de este tipo para extraer un subconjunto l.i. de un conjunto finito de vectores.

2.1. Pares Hereditarios

Definición 5 (Par Hereditario)**.** Sea E un conjunto finito y $\emptyset \neq \mathcal{I} \subseteq 2^E$. El par ordenado $(E, \mathcal{I})$ se dice hereditario si:

$$X \in \mathcal{I} \wedge Y \subseteq X \Rightarrow Y \in \mathcal{I}$$

A los conjuntos de $\mathcal{I}$ los llamamos independientes.

Ejemplo 1 (Pares Hereditarios)**.** Veamos algunos casos de pares hereditarios:

- $(E, 2^E)$ y $(E, \emptyset)$.
- $(E, \{X \subseteq E : |X| \leq k\})$
- Dado $w : E \to \mathbb{R}_+$ $(E, \{X \subseteq E : w(x) \leq w^*\})$
- $(E, \cap_{i=1}^n \mathcal{I}_i)$, con $(E, \mathcal{I}_i)$ par hereditario para $i = 1, ..., n$

Ejemplo 2 (Pares Hereditarios en Grafos)**.** Dado un grafo $G = (V, E)$, son pares hereditarios:

- $(E, \{F \subseteq E : F$ acíclico$\})$
- $(E, \{F \subseteq E : F$ matching$\})$

Definición 6 (Bases)**.** Sea $(E, \mathcal{I})$ hereditario y $X \subseteq E$. Una base de X es un subconjunto maximal de X que esté en $\mathcal{I}$. A las bases de E las llamamos simplemente bases.

2.2. Resolución de Problemas

Dados $(E, \mathcal{I})$ hereditario, $w : E \to \mathbb{R}$, queremos estudiar cuándo un algoritmo glotón resuelve los siguientes problemas:

- Encontrar una base de E de cardinal máximo.
- Encontrar una base de E de peso máximo/mínimo.
- Encontrar un conjunto independiente de peso máximo/mínimo.

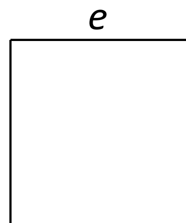

Figura 1: G

Ejemplo 3 (Falla de Glotón en el Problema de Cardinal máximo)**.** Consideremos Y ={matchings}.
Si elegimos e primero, Glotón no resuelve el problema.

Problema de Cardinalidad

Glotón-Cardinalidad

1: Dado $G = (V, E)$ hereditario.
2: $E = \{e_1, ..., e_m\}$ en cualquier orden.
3: Iniciar $S = \emptyset$
4: **for** $i = 1 : m$ **do**
5: **if** $S + e_i \in \mathcal{I}$ **then**
6: $S \leftarrow S + e_i$
7: **end if**
8: **end for**
9: Devolver S

Notemos que este algoritmo funciona, por ejemplo, para encontrar bases en espacios vectoriales.

Problema de la Base de peso Máximo

Glotón-Base(max)

1: Dado $G = (V, E)$ hereditario, $w : E \to \mathbb{R}$.
2: Ordenar $E = \{e_1, ..., e_m\}$ de mayor a menor peso.
3: Iniciar $S = \emptyset$
4: **for** $i = 1 : m$ **do**
5: **if** $S + e_i \in \mathcal{I}$ **then**
6: $S \leftarrow S + e_i$
7: **end if**
8: **end for**
9: Devolver S

Problema del Independiente de peso Máximo

Glotón-Independiente(max)

1: Dado $G = (V, E)$ hereditario, $w : E \to \mathbb{R}$.
2: Tomar $E^+ \subseteq E$, los objetos de peso positivo de E.
3: Ordenar $E^+ = \{e_1, ..., e_m\}$ de mayor a menor peso.
4: Iniciar $S = \emptyset$
5: **for** $i = 1 : m$ **do**
6: **if** $S + e_i \in \mathcal{I}$ **then**
7: $S \leftarrow S + e_i$
8: **end if**
9: **end for**
10: Devolver S

Estos tres algoritmos no siempre devuelven el objeto óptimo buscado.

2.3. Condiciones para que un Algoritmo Glotón funcione

Dado $(E, \mathcal{I})$, una condición necesaria para que el algoritmo Glotón-Cardinalidad funcione es:
(*) Todas las bases tienen el mismo cardinal.
Sin embargo, esta condición no es suficiente para Glotón-Base como veremos a continuación

Ejemplo 4 (Falla algoritmo Glotón-Base)**.** Notemos que (*) se cumple en la figura 2. Además $(A \cup B, \mathcal{I})$ hereditario para $\mathcal{I} = 2^A \cup 2^B$. Glotón devuelve $\{a, b\}$ de peso 3, pero la mejor base es $\{c, d\}$ de peso 4. Observemos que $(A \cup B, \mathcal{I})$ tiene solo 2 bases.

Podríamos pedir mejor que:
(**) $\forall X \subseteq E$ las bases de X tienen el mismo cardinal.

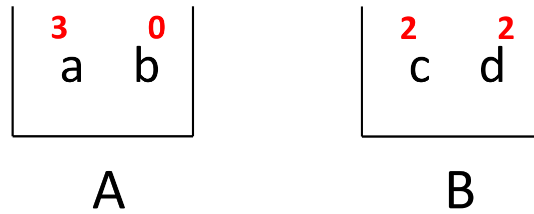

Figura 2:

2.4. Matroides

Definición 7 (Matroide)**.** $M = (E, \mathcal{I})$ hereditaria se dice Matroide si satisface alguno de los siguientes axiomas:

- **Aumentos:** $\forall X, Y \in \mathcal{I}, |Y| > |X| \quad \exists e \in Y \setminus X$ tal que $e + X \in \mathcal{I}$
- **Aumento Débil:** $\forall X, Y \in \mathcal{I}, |Y \setminus X| = 2, |X \setminus Y| = 1 \quad \exists e \in Y \setminus X$ tal que $e + X \in \mathcal{I}$
- **Cardinalidad de Bases:** $\forall F \subseteq E$ las bases de F tienen el mismo cardinal.

Demostraremos que los axiomas son equivalentes, pero primero veamos algunos ejemplos de Matroides.

Ejemplo 5. (Ejemplos de Matroides).

- **Matroide Uniforme:** $(E, \{X \subseteq E : |X| \leq k\})$
- **Matroides Vectoriales/Columnas/Representables:** Dada una matriz $A \in \mathbb{F}^{n \times m}$, con $\mathbb{F}$ cuerpo: ($\{$columnas de A$\}, \{x : x$ es l.i.$\}$) es Matroide.
  - Si una matroide es isomorfa a una matroide de columnas a coeficientes en $\mathbb{F}$ se dice representable en $\mathbb{F}$.
  - Las matroides representables en $\mathbb{G}F(2) = (\mathbb{Z}_2, +, \cdot)$ se llaman binarias.
  - Si una matroide es representable en cualquier cuerpo, se dice regular.