MA3705. Algoritmos Combinatoriales. 2014.
Profesor: José Soto
Escriba(s): Daniel Castro, Camila Fernández y Sebastián Tapia.
Fecha: 8 de Agosto 2014 .

Cátedra 4

En la siguiente clase, presentaremos criterios que nos resultarán cómodos a la hora de analizar la complejidad y eficiencia de un algoritmo, lo que concierne tanto al costo temporal como computacional que tiene llevarlo a cabo y al final presentaremos como ejemplo el análisis del algoritmo PRIM. Para esto, comenzaremos con la llamada notación asintótica.

1. Notación asintótica.

Para evitar las diferencias entre distintas formas de medir la complejidad de un algoritmo usaremos notación asintótica, para lo cual serán necesarias las siguientes definiciones, en las que f y g son funciones que van de $\mathbb{N}$ a $\mathbb{R}$, sin embargo, la notación es válida para funciones con dominios más complicados, por ejemplo, permitimos que f o g tengan una cantidad finita de puntos donde se indefinen.

Definición 1. ($O(g)$) Decimos que:

$$f \in O(g) \Leftrightarrow \exists n_0 \in \mathbb{N} \;\; \exists c > 0, \; \forall x \geq n_0 \;\; |f(x)| \leq c|g(x)|.$$

Se suele decir que f es "O-grande" de g o es "orden" g, lo que significa que f crece a lo más tan rápido como g.

Definición 2. ($o(g)$) Decimos que:

$$f \in o(g) \Leftrightarrow \forall \epsilon > 0 \; \exists n_o \in \mathbb{N}, \; \forall x \geq n_0 \;\; |f(x)| \leq \epsilon |g(x)|.$$

Se dice que f es "o-chica" de g, lo que significa que f crece estrictamente más lento que g.

Definición 3. ($\Omega(g)$) Decimos que:

$$f \in \Omega(g) \Leftrightarrow g \in O(f) \Leftrightarrow \exists n_0 \in \mathbb{N} \; \exists M > 0, \; \forall x \geq n_0 \;\; |f(x)| \geq M|g(x)|.$$

Se dice que f es "Omega-grande" de g, lo que significa que f crece a lo menos tan rápido como g.

Definición 4. ($\omega(g)$) Decimos que:

$$f \in \omega(g) \Leftrightarrow g \in o(f) \Leftrightarrow \forall M > 0 \; \exists n_o \in \mathbb{N}, \; \forall x \geq n_0 \;\; |f(x)| \geq M|g(x)|.$$

Se dice que f es "omega-chica" de g, lo que significa que f crece estrictamente más rápido que g.

Definición 5. ($\Theta(g)$) Decimos que:

$$f \in \Theta(g) \Leftrightarrow f \in O(g) \cap \Omega(g).$$

Se dice que f es "Theta" de g, lo que significa que f crece del mismo modo que g.

Se puede notar también que las primeras cuatro definiciones se pueden reformular de la siguiente manera pidiendole una condición extra a g.

Proposición 1. *Sean $f, g : \mathbb{N} \to \mathbb{R}$, con g tal que su conjunto de ceros es finito. Las siguientes sentencias son verdaderas.*

1. $f \in O(g) \Leftrightarrow \limsup_{x \to \infty} \left| \frac{f(x)}{g(x)} \right| < +\infty.$

2. $f \in o(g) \Leftrightarrow \lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| = 0.$

3. $f \in \Omega(g) \Leftrightarrow \liminf_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| > 0.$

4. $f \in \omega(g) \Leftrightarrow \lim_{x\to\infty} \left|\frac{f(x)}{g(x)}\right| = +\infty.$

Observación: Esta notación esconde constantes multiplicativas y sumas de funciones de menor orden, tal como se muestra a continuación:

1.1. Ejemplos.

1. $5n^2 + 2n^4 \in O(n^5)$.
2. $2n \in o(n\log(n))$.
3. $\log(n!) \in \Theta(n\log(n))$.
4. $2^n \in \omega(n^{1000000})$.
5. $\log_k(n) \in \Omega(\log(n)) \quad \forall k \in \mathbb{N}$.

1.2. Abuso de notación.

La O-notacón resulta ser muy cómoda, pero dada la tradición en su uso, existe abuso de notación, el cual será explicado a continuación:

Signo = en vez de $\in$: En vez de escribir $f \in O(g)$, escribiremos $f = O(g)$, esto es como el significado computacional que tiene el signo " = " y su sentido de asignación (tal como cuando escribimos "$n = n + 1$"). Para recalcar más aún su sentido, al signo " = " lo leemos como *es*, así que se utilizará como en el ejemplo: " un pájaro es un animal, pero un animal no es un pájaro". Ahora llevado a la O-notación:

$$n\log(n) = O(n^2) = O(n^3) \neq O(n^2).$$

A modo de otro ejemplo: $n^3 + 5n^2 + 40n^{1,5} = n^3 + O(n^2) = O(n^3)$.

2. Eficiencia o Complejidad de un Algoritmo.

Todo algoritmo recibe una entrada que pueden ser N números enteros (en una lista por ejemplo) los que, como información, se codifican usando S bits. Al analizar un algoritmo, se busca dar buenas cotas superiores para su complejidad (lo más ajustadas posible). En análisis teórico un algoritmo se considera eficiente cuando su complejidad está acotada por un polinomio.

Definición 6. (Algoritmo polinomial o débilmente polinomial) Si un algoritmo tiene complejidad $T(S) = O(S^k)$ para algún k fijo, decimos que el algoritmo es polinomial o débilmente polinomial.

Definición 7. (Algoritmo fuertemente polinomial) Si un algoritmo tiene complejidad $T(N) = O(N^k)$, donde N representa el número de datos que le entregamos, para algún k fijo, decimos que el algoritmo es fuertemente polinomial.

Donde notamos de estas definiciones que todo algoritmo fuertemente polinomial es débilmente polinomial, ya que $N = O(S)$.

2.1. Ejemplos

1. Algoritmo BFS: La entrada de BFS es la representación de un grafo, el cual se puede llevar a cabo de las siguientes dos maneras (por lo menos):

   a) Matriz de adyacencia: Esta es una matriz de $n \times n$ donde cada fila (respectiva columna) representa un vértice y cada coordena le corresponde un 1 o un 0, donde 1 significa que existe una arista entre ambos, y 0 que no. Entonces ocuparemos n^2 números (donde $n = |V|$). También cabe destacar que si usamos matriz de adyacencia para codificar un grafo, tanto N (el número de datos) como S (el número de bits) son $\Theta(n^2)$.

   b) Lista de incidencia: Esta es una lista enlazada de tamaño n, cada espacio representa un vértice, donde tiene enlazada otras lista con los vertices incidentes correspondientes. Entonces ocuparemos $O(n + m)$ números (donde $m = |E|$). También cabe destacar que si usamos lista de incidencia N es $\Theta(n + m)$, mientras que S es $\Theta((n + m)\log(n))$, pues los números involucrados tienen magnitud entre 1 y n.

   También, la clase anterior demostramos que la complejidad de BFS es $O(n + m)$, lo cual en ambas representaciones es fuertemente polinomial. Como regla general, todo algoritmo sobre grafos cuya complejidad está acotada por un polinomio en n y en m es fuermente polinomial.

2. Tenemos un problema que involucra N datos.
   Si un algoritmo demora $T(N)$ en resolver el problema y duplican el número de datos ($2N$ datos), entonces demorará $T(2N)$.
   Si T es fuertemente polinomial, es decir, $T(N) = O(N^k)$, entonces:

   $$T(2N) = O((2N)^k) = O(2^k N^k) = O(N^k)$$

   El tiempo necesario para resolver el problema grande se multiplica por una constante, en este caso 2^k.

3. Si ahora el algoritmo demora $T(N) = 2^N$, entonces:

   $$T(2N) = 2^{2N} = (2^N)^2$$

   El que ya no es una multiplicación por una constante, por tanto ya no es polinomial.

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.

3. Análisis del algoritmo PRIM

Recordemos el algoritmo PRIM de la clase anterior:

Algoritmo 1: Algoritmo de Prim (Prim 1957 - Jarník 1930)

```
Elegir r ∈ V;
U ← {r}
F ← ∅
while δ(U) ≠ ∅ do
    Sea e = uv ∈ δ(U), u ∈ U, v ∉ U tal que e es la arista de menor peso en δ(U)
    U ← U + w;
    F ← F + e;
end
return T = (U, F)
```

Vamos a mostrar la implementación de PRIM de dos formas distintas, pero antes nos haremos dos preguntas que nos ayudaran en nuestro propósito:

1. ¿ Cuál es la complejidad de ordenar N números?

$$\Theta(N\log(N)) \quad \text{vía mergesort, por ejemplo.}$$

2. ¿Complejidad de encontrar el mínimo en una lista?

Sin ordenar	$O(N)$
Ordenada	$O(1)$

3.1. Implementación

1.

Algoritmo 2: 1ra implementación del Algoritmo Prim

Elegir $r \in V$;
$U \leftarrow \{r\}$
$F \leftarrow \emptyset$
while $\delta(U) \neq \emptyset$ **do**
  (Revisar todas las aristas y marcar aquellas que están en $\delta(U)$)
  Sea $e = uv \in \delta(U),\ u \in U,\ v \notin U$ tal que e es la arista de menor peso en $\delta(U)$
  (sin haber ordenado previamente a $\delta(U)$)
  $U \leftarrow U + v$;
  $F \leftarrow F + e$;
end
return $T = (U, F)$

Está implementación, por los comentarios hechos en parentesis, se puede concluir facilmente que es de orden $O(m^2)$, donde el algoritmo recibe un grafo $G = (V, E)$, $m = |E|$ y $n = |V|$.

En la siguiente implementación, se ocupará la función cand$(\cdot)$, la que toma un vértice y calcula la arísta incidente a él más liviana.

2.

Algoritmo 3: 2da implementación del Algoritmo Prim

Elegir $r \in V$;
$U \leftarrow \{r\}$
$F \leftarrow \emptyset$
$\forall v \in V$ calcular cand(v) **for** $i = 1, ..., n$ **do**
  Elegir el mejor candidato en $\arg\min_{v \in U} \text{cand}(v)$.
  $e = uv$ con $u \in U$ y $v \in V$; $U \leftarrow U + v$;
  $F \leftarrow F + e$;
  Recalcular $\forall w \notin U$ cand(w)
end
return $T = (U, F)$

Para analizar esta implementación, la divideremos en fuera y dentro del for.

Fuera del for:

a)Para calcular por primera vez cand(v) para todo vértice v, sólo debemos mirar las aristas incidentes a r, lo que toma $O(n + \deg(r)) = O(n)$.

Dentro del for:

b) Elegir el mejor candidato toma $O(n)$, pues es el mínimo entre a lo más n números.

c) Por último, recalcular candidato para un w toma tiempo constante, ya que es sólo reemplazar el antiguo cand(w) por el mejor entre cand(w) y el peso de la arista vw (si es que existe), donde v es el vértice que acaba de entrar a U. Como esto lo haremos a lo más n veces, el trabajo es de $O(n)$.

Juntando todo lo anterior, el trabajo fuera del for es de $O(n)$, el trabajo adentro del for es de $O(n)$, repetido $O(n)$ veces, por lo que el trabajo total es $O(n^2)$.

Usando estructuras de datos más avanzadas (colas de prioridad llamadas Heaps), al algoritmo Prim se puede implementar más rápido, como muestran los siguientes ejemplos.

3. Usando Heaps binarios, se puede hacer una implementación de orden $O((n+m)\log(n))$.
4. Usando Heaps de Fibonacci, se puede hacer una implementación de orden $O(m + n\log(n))$,

4. Próxima Clase.

En la próxima sesión, se estudiarán los algoritmos denominados Algoritmos Glotones (o Avaros). Estos corresponden a algoritmos que en cada decisión incorpora a la solución el objeto más barato. Un ejemplo de algoritmo glotón para calcular bosques generadores de peso mínimo es el algoritmo de Kruskal, que expondremos a continuación.

La entrada del algoritmo, es un grafo $G = (V, E)$ y $w : E \to \mathbb{R}$, una función que respresenta al peso asignado a cada arista.

Algoritmo 4: Algoritmo de Kruskal

Ordenar todas las aristas en orden creciente $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_n)$
$F \leftarrow \emptyset$
for $i = 1, ..., n$ **do**
  if $F + e_i$ *es acíclico* **then**
    $F \leftarrow F + e_i$
  end
end
return $T = (V, F)$