Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Matemática
MA47A: Optimización Combinatorial
Profesor: Roberto Cominetti
Auxiliares: Raul Aliaga Diaz, Cristóbal Guzmán

Clase Auxiliar 27/03/07

1. Definiciones previas

Para estimar el orden de magnitud del tiempo de ejecución de un programa, usaremos notación asintótica. Para ellos, introducimos las siguientes definiciones:

1. $O(g) = \{f : \mathbb{N} \to \mathbb{R} \text{ tq: } \exists c > 0 \text{ y } n_0 \in \mathbb{N} \text{ tq } |f(n)| \leq c|g(n)| \quad \forall n \geq n_0\}$.

   Esta definición nos permitirá en rigor, hablar del orden de magnitud de una función $f : \mathbb{N} \to \mathbb{R}$. Algunas propiedades:

   - $c \cdot O(f) = O(f)$ para todo $c \in \mathbb{R}$, $c > 0$.
   - $O(f) + O(g) = O(f + g) = O(max\{f + g\})$.
   - $O(f)O(g) = O(fg) = fO(g)$.
   - $O(f)^m = O(f^m)$, para todo $m \in \mathbb{N}$, $m > 0$.
   - $O(O(f)) = O(f)$.

2. $\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \text{ tq: } \exists c > 0 \text{ y } n_0 \in \mathbb{N} \text{ tq } |f(n)| \geq c|g(n)| \quad \forall n \geq n_0\}$.

3. $o(g) = \{f \in O(g) \text{ tq } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$.

4. $\omega(g) = \{f \in \Omega(g) \text{ tq } \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\}$.

5. $\Theta(g) = O(g) \cap \Omega(g)$.

Diremos, por comodidad, que $f(n) = O(g(n))$ cuando nos referimos a $f \in O(g)$. Para más detalles, ver notas sobre notación asintótica del Profesor Marcos Kiwi.

2. Algoritmos de ordenamiento

2.1. Bubblesort

Consideremos el siguiente algoritmo:

BubbleSort(A) 1 Ordena un arreglo A

```
1: N ← length(A)
2: for i = 1 ... N do
3:     for j = 1 ... N − i do
4:         if A[j + 1] < A[j] then
5:             tmp = A[j]
6:             A[j] = A[j + 1]
7:             A[j + 1] = A[j]
8:         end if
9:     end for
10: end for
```

¿Cuántas operaciones toma en ejecutar BubbleSort? Podemos notar fácilmente, que es $O(n^2)$. Pero, ¿Qué pasa si tenemos una recursión?

2.2. MergeSort

Consideremos el algoritmo de ordenamiento MergeSort:

MergeSort(A,p,f) 2 Ordena un arreglo A

```
1: if p < f then
2:     m ← ⌊(p+f)/2⌋
3:     MergeSort(A,p,m)
4:     MergeSort(A,m,f)
5:     Merge(A,p,m,f)
6: end if
```

En este caso, no es claro cuánto tiempo toma. Sea $T(n)$ el tiempo que toma en correr el algoritmo, según el número de pasos realizados. Podemos así, escribir la ecuación:

$$T(n) = 2T(\frac{n}{2}) + D(n) + C(n) \tag{1}$$

Donde $D(n)$ es el tiempo que nos toma dividir el problema en dos, y $C(n)$ el tiempo que nos toma hacer la operación `Merge`, que junta las soluciones obtenidas en las mitades previas. Esta recursión es válida, siempre que el tamaño de la instancia n, sea lo suficientemente grande ($n > 1$, pues ordenar un elemento es trivial, ¡se deja tal cual!). Como la operación que consideramos a contar es comparar, podemos asumir que $D(n)$ es despreciable frente a las comparaciones (en nuestro caso, es solo una operación aritmética) y observemos que $C(n) = n$. Luego, si desarrollamos la recursión, obtendremos:

$$\begin{aligned} T(n) &= 2T(\tfrac{n}{2}) + n \\ &= 2 \cdot (2T(\tfrac{n}{4}) + \tfrac{n}{2}) + n \\ &= 2 \cdot (2 \cdot (2T(\tfrac{n}{8}) + \tfrac{n}{4}) + \tfrac{n}{2}) + n \\ &= \ldots \\ &= 2 * 2 * \ldots * 2T(1) + n + \tfrac{n}{2} + \ldots + \tfrac{n}{2^k} \end{aligned}$$

Donde hemos iterado k veces, con k el número de veces antes de que $\frac{n}{2^k}$ sea menor o igual a uno. Así nos queda:

$$T(n) = 2^k T(1) + \sum_{j=0}^{k} n \tag{2}$$

Si consideramos entradas que son de tamaño siempre una potencia de dos, como $n = 2^l$, es directo observar que $T(n) = lT(1) + nl = \log nT(1) + n \log n$. Como $T(1)$ es constante, tenemos en consecuencia que $T(n) = O(n \log n)$.

Si consideramos entradas de cualquier tamaño en general, podemos demostrar (usando cotas apropiadas obtenidas a partir de considerar $\lfloor \frac{n}{2} \rfloor \leq \frac{n}{b}$ o $\lceil \frac{n}{2} \rceil \geq \frac{n}{2}$) que $T(n) = O(n \log n)$.

Se puede demostrar que cualquier algoritmo de ordenamiento de este estilo (que particiona la entrada para reordenar recursivamente) no puede hacerlo mejor que $O(n \log n)$[1], esta estrategia en general es una manera de abordar la técnica de "Dividir y reinar"[2]. Para recurrencias como la recién vista, puede obtenerse una cota en general vía "El Teorema Maestro"[3].

3. Algoritmo de Kruskal

Consideremos el algoritmo de Kruskal para encontrar un árbol generador de peso mínimo:

¿Cuántas operaciones toma el algoritmo?. Notemos que hay algunas operaciones "exógenas" a la descripción usual del algoritmo, que dice "si los extremos de los arcos están en componentes conexas distintas, entonces se agrega".

Sin embargo, en el momento de implementar el algoritmo, podemos considerarlo de la siguiente manera. Inicialmente, para cada nodo se le crea un conjunto, el cual es el árbol al que pertence (MakeSet(u)), luego, cuando se hace la pregunta ¿Cuál es la componente conexa a la que pertenece u? se realiza el llamado a FindSet(u), y una vez que se unen los arcos se actualiza el conjunto al que pertenecen con Union(u,v).

Dependiendo de como representemos el grafo mediante estructuras de datos apropiadas (o no tan apropiadas), las operaciones que tome el algoritmo cambiarán.

[1] Ver "Introduction to algorithms", Cormen, sección IV.9.1
[2] Ver Cormen, sección I.3.1
[3] Ver Cormen, secciónes I.4.3, I.4.4

Kruskal(G,w) 3 Encuentra un árbol generador del grafo G de peso mínimo

```
1: A ← ∅
2: for all v ∈ V[G] do
3:     MakeSet(v)
4: end for
5: Ordenar los arcos e ∈ E[G] en función de los pesos según w
6: for all arco (u, v) ∈ E, en el orden anterior do
7:     if FindSet(u) ≠ FindSet(v) then
8:         A ← A ∪ {(u, v)}
9:         Union(u,v)
10:    end if
11: end for
12: return A
```

La estructura de datos que escojamos para representar el grafo, debe (al menos) implementar las operaciones siguientes:

- MakeSet(u).
- FindSet(u).
- Union(u,v).

Consideremos dos maneras de representar un grafo. Mediante una matriz de adyacencia y una lista enlazada.

Las operaciones podemos contabilizarlas como se sigue (con $|V[G]| = n$ y $|E[G]| = m$):

	Matriz adyacencia	Lista enlazada
`MakeSet`(u)	$O(1)$	$O(1)$
`FindSet`(u)	$O(n)$	$O(1)$
`Union`(u,v)	$O(n)$	$O(\log n)$

La implementación con listas enlazadas, considera tener para un árbol, un nodo raíz, y punteros desde cada nodo a los nodos que le siguen. También, cada nodo tiene un puntero a la raíz del árbol.

El inicializar las estructuras, podemos considerar que toma tiempo constante, asumiendo en la matriz de adyacencia, que inicializar los valores del arreglo de la matriz toma un tiempo despreciable con respecto a las otras operaciones. La operación `FindSet`(u), en el caso de la matriz de adyacencia, debe en el peor caso preguntar si llega a cada uno de los otros nodos, para asi definir "on the fly" a que conjunto pertenece el nodo en cuestión. En cambio, en la lista enlazada, podemos guardar una etiqueta en el nodo raíz (podemos hablar de raíz, pues cada componente conexa será siempre un árbol) que nos diga cuál es el conjunto al que pertenece el nodo. Así, nos queda por saber como se diferencia la operación `Union` en ambos casos. En el caso de la matriz de adyacencia, no sería necesario, pues en cada ocasión calcularíamos el conjunto de los "alcanzables" a partir de un nodo, solo habría que actualizar la matriz

de adyacencia correspondiente en un valor. Sin embargo, en la lista enlazada, debemos actualizar los valores del nodo *padre* o *raíz*, para que asi el `FindSet` funcione como debe. Para ello, notemos que podemos hacer las asignaciones (o "cambios de padre") a lo más $O(\log n)$ veces. Pues si en cada caso consideramos el árbol más pequeño, podemos actualizar menos nodos si sólo cambiamos las referencias de ese árbol más pequeño. Así, cada vez que un nodo es "forzado" a cambiar su padre o raíz, es porque el otro árbol de la unión tiene tamaño al menos igual al del árbol en cuestión, por lo tanto, un nodo no puede ser cambiado más de $O(\log n)$ veces, pues $\forall k \leq n$, si a un nodo se le han realizado $\lceil \log k \rceil$ actualizaciones de padre, entonces el conjunto de nodos que lleva el árbol tiene al menos k nodos.

En el primer paso del algoritmo, se ordenan los arcos, que como vimos en la parte anterior puede hacerse en $O(m \log m)$. Luego, sumando las operaciones hechas en la parte de revisar los arcos, notamos que se realizan a lo más $n-1 \log n$ operaciones, y puesto que el grafo es conexo, entonces podemos concluir que el algoritmo toma tiempo $O(m \log n)$ o bien $O(m \log m)$.

4. Archivos adjuntos

Adjunto a este documento, están los siguientes archivos:

- En la carpeta "Estructuras", se encuentran algunos ejemplos de estructuras de datos.
- En la carpeta "Matlab", se encuentran implementaciones para los algoritmos de ordenamiento, junto con funciones de testeo de las mismas: `testBubbleSort` y `testMergeSort`, en que se grafica la función g del $O(g)$ en que toma correr el algoritmo, y los tiempos obtenidos empíricamente[4]. Dentro de ella, también hay dos carpetas que contienen una implementación en Matlab del algoritmo de Kruskal ("MST_Kruskal") y otra que permite dibujar grafos, proveyéndolos a una función en el formato adecuado ("GraphLayout").
- En la carpeta "Javas", se encuentra dos implementaciones de Kruskal, una muy "pro" (pero que para correr necesita una versión de java `1.5` o superior), y otra mucho más sencilla, junto con un archivo de prueba.

Se recomienda revisar los archivos y las implementaciones, junto con las notas sobre notación asintótica del profesor Marcos Kiwi.

[4]Se recomienda usar valores no mayores que 12 para invocar a las funciones, puesto que el tamaño de los arreglos crecen exponencialmente