Algoritmos Avaros (Greedy Algorithms)

Agustín J. González

ELO-320: Estructura de Datos y Algoritmos

1er. Sem. 2002

Introducción

- Hay muchos algoritmos que pueden ser clasificados bajo la categoría de Algoritmos Avaros, Oportunistas (Greedy Algorithms). La idea es optar por la mejor opción de corto plazo. Sacar mayor provecho inmediato.
- Éstos se caracterizan por hacer la elección que parece mejor en el momento.
- Éstos no conducen siempre a una solución óptima, pero para muchos casos si.
- Éstos producen una solución óptima cuando se puede llegar a ésta a través de elecciones localmente óptimas.
- Hay un parecido con programación dinámica (investigación de operaciones). La diferencia es que en programación dinámica, la elección en cada paso depende de la solución a un sub-problema. En los algoritmos avaros hacemos la elección que parece mejor en el momento y luego se resuelve el sub-problema que queda.
- Se puede pensar que algoritmos avaros resuelven el problema en forma top-down (de arriba hacia abajo); mientras que la programación dinámica lo hace bottom-up (de abajo hacia arriba).
- Éste método es muy poderoso y trabaja bien en muchos algoritmos que veremos en las próximas semanas; por ejemplo: Minimum-spanning-tree (árbol de mínima extensión), el algoritmo de Dijkstra para el camino más corto en un grafo y desde una única fuente.

Problema de la selección de actividades

- Se requiere itinerar el uso de un recurso entre varias actividades que compiten.
- Problema: Tenemos un conjunto de S={0, 1, 2, 3, .. , n-1} actividades que desean usar un recurso (como una sala de clases por ejemplo) del cual sólo uno puede usarse a la vez. Cada actividad tiene un tiempo de partida s_i y un tiempo de término f_i , donde $s_i \leq f_i$.
  La tarea consiste en seleccionar el conjunto de cardinalidad máxima de actividades mutuamente compatibles; i.e. que no se traslapen.
- Suponemos que las actividades están ordenadas por orden creciente de tiempo de término: $f_o \leq f_1 \leq f_2 \leq f_3 \leq f_4 ... \leq f_{n-1}$ Si no fuera así, sabemos que podemos ordenarlas en tiempo O(n lg n).
- **Int Selector_de_Actividades(float s[], float f [], int n, int A[]) {**

```
        /* s y f son arreglos de tiempos de partida y término */
        /* n: número de actividades compitiendo */
        /* A: Arreglo de salida indicando la asignación óptima para
                    máximo número de actividades*/
        /* se retorna el número de actividad itineradas en A*/
        int i, j = 0, m = 0;
        A[m++] = 0;
        for (i=1; i < n; i++)
                if (s [i] >= f [j]) {
                        A [m++] = i;
                        j = i;
                }
        return (m);
}
```

Ejemplo

- Se puede demostrar que la solución aquí propuesta es óptima.
- La estrategia fue asignar la actividad que dejara el recurso libre la mayor cantidad de tiempo en el futuro y que fuera compatible con las anteriores.
- Ejemplo: sea

	0	1	2	3	4	5	6	7	8	9	10
S	1	3	0	5	3	5	6	8	8	2	12
F	4	5	6	7	8	9	10	11	12	13	14