Algoritmia/Algoritmos voraces

Una aproximación **voraz** consiste en que cada elemento a considerar se evalúa una única vez, siendo descartado o seleccionado, de tal forma que si es seleccionado forma parte de la solución, y si es descartado, no forma parte de la solución ni volverá a ser considerado para la misma. Una forma de ver los algoritmos voraces es considerar la estrategia de ../Vuelta atrás/, en la cual se vuelve recursivamente a decisiones anteriormente tomadas para variar la elección entonces tomada, pero eliminando esa recursión y eligiendo la mejor opción.

El término voraz se deriva de la forma en que los datos de entrada se van tratando, realizando la elección de desechar o seleccionar un determinado elemento una sola vez.

Al contrario que con otros métodos algorítmicos, no siempre es posible dar una solución a un problema empleando un algoritmo voraz. No todos los problemas son resolubles con algoritmos voraces.

Los algoritmos voraces tienden a ser bastante eficientes y pueden implementarse de forma relativamente sencilla. Su eficiencia se deriva de la forma en que trata los datos, llegando a alcanzar muchas veces una complejidad de orden lineal. Sin embargo, la mayoría de los intentos de crear un algoritmo voraz correcto fallan a menos que exista previamente una prueba precisa que demuestre la correctitud del algoritmo. Cuando una estrategia voraz falla al producir resultados óptimos en todas las entradas, en lugar de algoritmo suele denominarse heurística. Las heurísticas resultan útiles cuando la velocidad es más importante que los resultados exactos (por ejemplo, cuando resultados "bastante buenos" son suficientes).

http://es.wikipedia.org/wiki/Algoritmo_voraz

Ejemplos y algoritmos típicos

Algoritmo de Dijkstra del camino más corto

http://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra

Árbol de recubrimiento mínimo

Traer de http://en.wikipedia.org/wiki/Minimum_spanning_tree

Algoritmo de Prim

http://es.wikipedia.org/wiki/Algoritmo_de_Prim

Algoritmo de Kruskal

http://es.wikipedia.org/wiki/Algoritmo_de_Kruskal

Problemas resueltos

Problema de la mochila

- **Enunciado**: "Se tiene una mochila que es capaz de soportar un peso máximo *P*, así como un conjunto de objetos, cada uno de ellos con un peso y un beneficio. La solución pasa por conseguir introducir el máximo beneficio en la mochila, eligiendo los objetos adecuados. Cada objeto puede tomarse completo o fraccionado".
- **Solución**: La forma más simple de saber qué objetos se deben tomar es ordenar dichos objetos por la relación *beneficio / peso* de mayor a menor. De esta forma, tomaremos los objetos con mayor beneficio en este orden hasta que la bolsa se llene, fraccionando si fuera preciso, el último objeto a tomar.
- **Algoritmo**:

```
- V almacena los beneficios de cada objeto,
- P almacena el peso de cada objeto,
- sol devuelve el tanto por 1 de objeto que se toma,
- benef devuelve el beneficio total,
- valor_obten almacena el beneficio parcial,
- peso_ac almacena el peso parcial.
```

```
fun mochila (V [1..n], P[1..n] de nat; m: nat) dev <sol[1..n] '''de''' nat, benef: nat>
begin
    para j := 1 hasta n hacer
        sol [j] := 0;
    fpara
    valor_obten := 0; i:=0; peso_ac := 0;
    mientras peso_ac  < m hacer
        si peso [i] + peso_ac < m entonces
            valor_obten := valor_obten + valor [i];
            sol [i] := 1;
            peso_ac := peso_ac + peso [i];
        si no
            sol [i] := (m − peso_ac) / peso [i];
            peso_ac := m;
            valor_obten := valor_obten + valor [i] * sol [i];
        fsi
        i := i + 1;
    fmientras
benef := valor_obten;
ffun
```

Problema del cambio de moneda

- **Enunciado**: "Se pide crear un algoritmo que permita a una máquina expendedora devolver el cambio mediante el menor número de monedas posible, considerando que el número de monedas es limitado, es decir, se tiene un número concreto de monedas de cada tipo".
- **Solución**: La estrategia a seguir consiste en escoger sucesivamente las monedas de valor mayor que no superen la cantidad de cambio a devolver. El buen funcionamiento del algoritmo depende de los tipos de monedas presentes en la entrada. Así, por ejemplo, si no hay monedas de valor menor que diez, no se podrá devolver un cambio menor que diez. Además, la limitación del número de monedas también influye en la optimalidad del algoritmo, el cual devuelve buenas soluciones bajo determinados conjuntos de datos, pero no siempre. Considérense los dos siguientes ejemplos como demostración de lo dicho:

Monedas	50	25	5	1
Cantidad	3	4	1	6

Si hay que devolver la cantidad 110 siguiendo el método del algoritmo voraz, se tomaría primero una moneda de 50, quedando una cantidad restante de 60. Como 50 es aún menor que 60, se tomaría otra moneda de 50. Ahora la cantidad restante es 10, por tanto ya tenemos que devolver una moneda de 5, ya que 50 y 25 son mayores que 10, y por tanto se desechan. La cantidad a devolver ahora es 5. Se tomaría otra moneda de 5, pero puesto que ya no nos queda ninguna, deberán devolverse 5 de valor 1, terminando así el problema de forma correcta.

Monedas	6	4	1
Cantidad	3	4	1

Si queremos devolver la cantidad 8, siguiendo el procedimiento anterior, el algoritmo tomaría primero una moneda de 6, quedando un resto de 2. Tomaría 2 monedas de valor 1, habiendo devuelto por tanto 3 monedas, cuando es fácil ver que con 2 monedas de valor 4 se obtiene el resultado pedido.

- **Algoritmo**:

```
fun cambio (monedas_valor[1..n] de nat, monedas[1..n] de nat, importe: nat) dev cambio[1..n] de nat
   m := 1;
   mientras (importe > 0) and (m <= n) hacer
       si (monedas_valor[m] <= importe) and (monedas[m] > 0) entonces
              monedas[m] := monedas[m] − 1;
              cambio[m] := cambio[m] + 1;
              importe := importe − monedas_valor[m];
       si no
              m := m + 1;
       fsi
   fmientras
   si importe > 0 entonces devolver “Error”; fsi
ffun
```

Véase el apartado de Programación dinámica para una solución alternativa a un problema similar.

Problema del maratón de películas

Traer de http://es.wikipedia.org/wiki/Problema_del_maratón_de_películas_mediante_algoritmo_voraz

Problema de los intervalos cerrados de puntos

- **Enunciado**: "Se pide un algoritmo para que, dado un conjunto de puntos sobre una recta real, determine el menor conjunto de intervalos cerrados de longitud 1 que contenga a todos los puntos dados".
- **Solución**: Partiendo de un conjunto ordenado de puntos, se consideran los puntos sucesivos en cada iteración del algoritmo. Si la entrada es un conjunto ordenado, el coste del algoritmo es lineal respecto a la cantidad de puntos.
- **Algoritmo**:

```
tipos
    intervaloCerrado = vector [1..2] de real
ftipos
```

```
fun cubrePuntos(puntos[1..n] de real) dev conjuntoPuntos[intervaloCerrado]

      var
         intervaloAux : intervaloCerrado;
         i : entero;
      fvar

      conjuntoPuntos := conjuntoVacio();
      i := 1;

      mientras i <= n hacer

         intervaloAux[1] := puntos[i];
         intervaloAux[2] := puntos[i] + 1;
         añadir(intervaloAux, conjuntoPuntos);
         mientras i <= n Λ puntos[i] <= intervaloAux[2] hacer
            i := i + 1;
         fmientras

      fmientras

ffun
```

Fuentes y contribuyentes del artículo

Algoritmia/Algoritmos voraces *Fuente*: http://es.wikibooks.org/w/index.php?oldid=179679 *Contribuyentes*: Baiji, Gothmog, Jesjimher, ManuelGR, MarcoAurelio, Morza, 6 ediciones anónimas

Licencia