Algoritmos ávidos

Pedro O. Pérez M., MTI

Análisis y diseño de algoritmos
Tecnológico de Monterrey

pperezm@tec.mx

02-2019

Contenido

Definición

Se conocen también como *algoritmos miopes*, *golosos*, *ávidos* o *avaros*, y caracterizan por decisiones basados en la información que tienen a primera mano, sin tener en cuenta lo que pueda pasar más adelante. Además, una vez que toman una decisión nunca reconsideran otras posibilidades, lo que ocasionalmente los lleva a caer en puntos muertos o sin salida.

Los algoritmos ávidos también se caracterizan por la rapidez con la que encuentran una solución (cuando la encuentran), que casi nunca es la mejor. Normalmente son utilizados para resolver problemas en los cuales la velocidad de respuesta debe ser muy alta o el espacio de búsqueda es muy grande.

Ejemplos típicos de problemas que se pueden resolver mediante este paradigma están las búsquedas en árboles o graos, solución de laberintos y algunos juegos entre otros. También muchos problemas que requieren obtener máximos o mínimos.

Forma general

La estrategia general de este tipo de algoritmos se basa en la construcción de una solución que comienza sin elementos, y cada vez que debe tomar algún tipo de decisión lo hace con la información que tiene en ese momento, para, de alguna manera, agregar elementos y así avanzar hacia la solución final. Cada elemento se agrega al conjunto solución, y así hasta llegar a la solución completa o a un punto en el cual el algoritmo no puede seguir avanzando, lo cual no indica que no se encontró una solución al problema.

Procedure 1 GREEDY_ALGORITHM

```
Input: C : Set
  S : Set
  while C ≠ ∅ and SOLUTION(S) = false do
    x ← SELECT(C)
    S ← S + x
    C ← C − x
  end while
  if SOLUTION(S) then
    return S
  else
    return ∅
  end if
```

Cambio de monedas

Dado un sistema monetario S con N monedas de diferentes denominaciones y una cantidad de cambio C, calcular el menor número de monedas del sistema monetario S equivalente a C.

Ejemplos:

- **Input** : s[] = 1, 3, 4 c = 6
  Output : 3
  Explanation : The change will be (4 + 1 + 1) = 3

Procedure 2 COIN_CHANGE

Input: $S : Array, c : Integer$

```
min ← 0
SORT_DESC(S)
for i ← 1 to S.length do
    min ← min + (c/S[i])
    c ← c mód S[i]
end for
```

Búsqueda en profundidad

La búsqueda en profundidad (en inglés **DFS** o **Depth First Search**) es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado [1].

[1] https://goo.gl/ZsAAzP

Procedure 3 DFS

Input: $start : Vertex, g : Graph$

$visited : Set$

$x\_visit : Stack$

$x\_visit.push(start)$

while $!x\_visit.empty()$ **do**

  $current \leftarrow x\_visit.pop()$

  if $current \notin visited$ **then**

    $visited.add(current)$

    for all v in $current.connections()$ **do**

      $x\_visit.push(start)$

    end for

  end if

end while

return $visited$

Programación de actividades

Te dan N actividades con sus tiempos de inicio (S_i) y finalización (F_i). Selecciona el número máximo de actividades que puede realizar una sola persona, asumiendo que una persona solo puede trabajar en una sola actividad a la vez. [2]
Ejemplo:

- **Input** :
  start[] = (1, 3, 0, 5, 8, 5)
  finish[] = (2, 4, 6, 7, 9, 9)
  Output : 4

[2] `https://goo.gl/1RG25M`

Procedure 4 ACTIVITIES_SELECTION

Input: $A : Activity\_Array$

```
SORT_ASC_BY_END(A)
i ← 1
S ← ∅ + A[i]
for j ← 2 to A.length do
   if A[j].start ≥ A[i].end then
      S ← S + A[j]
      i ← j
   end if
end for
return S
```

Reservaciones de hotel

Un gerente de hotel debe procesar N reservas anticipadas de habitaciones para la próxima temporada. Su hotel tiene K habitaciones. Las reservas contienen una fecha de llegada y una fecha de salida. Quiere saber si hay suficientes habitaciones en el hotel para satisfacer la demanda. [3]

[3] `https://goo.gl/7e6idL`

Procedure 5 BOOKING_PROBLEM

Input: $Arrival : Array, Departure : Array, k : Integer$

```
SORT_ASC(Arrival)
SORT_ASC(Departure)
i ← 1
j ← 1
current ← 0
required ← 0
while i < Arrival.length and j < Departure.length do
    if Arrival[i] < Departure[j] then
        current ← current + 1
        required = MAX(current, required)
        i ← i + 1
    else
        current ← current − 1
        j ← j + 1
    end if
end while
```

while $i < n$ **do**
  $current \leftarrow current + 1$
  $required = MAX(current, required)$
  $i \leftarrow i + 1$
end while
while $j < n$ **do**
  $current \leftarrow current - 1$
  $j \leftarrow j + 1$
end while
return $k \geq required$

Fracciones egipcias

Cada fracción positiva puede representarse como la suma de fracciones unitarias únicas. Una fracción es una fracción unitaria si el numerador es 1 y el denominador es un entero positivo, por ejemplo, 1/3 es una fracción unitaria. Dicha representación se llama fracción egipcia, ya que fue utilizada por los antiguos egipcios. [4]

Representación en fracción egipcia de 2/3 is 1/2 + 1/6
Representación en fracción egipcia de 6/14 is 1/3 + 1/11 + 1/231
Representación en fracción egipcia de 12/13 is 1/2 + 1/3 + 1/12 + 1/156

[4] https://goo.gl/sUzBPd

Procedure 6 EGYPTIAN

Input: $num : integer, dem : Integer$

```
if num = 0 or dem = 0 then
    return
end if
if dem mód num = 0 then
    print "1/" + (dem/num)
    return
end if
if num mód dem = 0 then
    print (num/dem)
    return
end if
```

if $num > dem$ **then**
  print $(num/dem) + " + "$
  $EGYPTIAN(num \mod dem, dem)$
  return
end if
$n \leftarrow (dem/num) + 1$
print $"1/" + n + " + "$
$EGYPTIAN((num * n) - dem, dem * n)$

Subconjunto de producto máximo de un arreglo

Dado un arreglo A, tenemos que encontrar el producto máximo posible con el subconjunto de elementos presentes en el arreglo. El producto máximo puede ser solo uno de los elementos del arreglo. [5]

Ejemplos:

- **Input** : a[] = -1, -1, -2, 4, 3
  Output : 24
  Explanation : Maximum product will be (-2 * -1 * 4 * 3) = 24
- **Input** : a[] = -1, 0
  Output : 0
  Explanation : 0 (single element) is maximum product possible
- **Input** : a[] = 0, 0, 0
  Output : 0

[5] https://goo.gl/spb5Ka

Una solución simple sería generar todos los subconjuntos, encontrar el producto de cada subconjunto y regresa el máximo. Sin embargo, existe una mejor solución si tomamos en cuenta los siguiente factores:

- Si el número de elementos negativos es par, el resultado es, sencillamente, el producto de todos los elementos.
- Si el número de elementos negativos es impar, el resultado es la multiplicación de todos los elementos excepto el número negativo más grande.
- Si hay ceros, el resultado el producto de todos los números, excepto los ceros con una excepción. La excepción es cuadn hay un número negativo y todos los otros números son ceros. En este caso, el resultado es 0.

Procedure 7 MAXIMUM_PRODUCT

```
Input: A : Array
  if n = 1 then
    if A[1] < 1 then
      return 0
    else
      return A[1]
    end if
  end if
  max_neg ← INT_MIN
  count_neg ← 0
  count_zero ← 0
  product ← 1
```

for $i \leftarrow 1$ **to** $A.length$ **do**
  if $A[i] = 0$ **then**
    $count\_zero \leftarrow count\_zero + 1$
  else
    if $A[i] < 0$ **then**
      $count\_neg \leftarrow count\_neg + 1$
      $max\_neg \leftarrow MAX(max\_neg, count\_neg)$
    end if
    $product \leftarrow product * A[i]$
  end if
end for

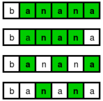
```
if count_zero == n then
   return 0
end if
if count_neg mód 2 = 1 then
   if count_neg = 1 and count_zero > 0 and (count_neg+count_zero) =
   n then
      return 0
   end if
   product ← product max_neg
end if
return product
```

Subsecuencia lexicográficamente más grande

Dada una cadena S y un entero K. La tarea es encontrar la subsecuencia lexicográficamente más grande de S, digamos T, de modo que cada carácter en T debe aparecer al menos K veces. [6]

Entrada: $S = banana$, $K = 2$

Salida: nn



De las opciones anteriores, nn es la lexicográficamente más grande.

[6] https://goo.gl/iwFFCA

Procedure 8 SUBSEQUENCE

Input: S : $String$, T : $String$, k : $Integer$

```
last ← 1
new_last ← 1
for ch ←' z' to 'a' do
    count ← 0
    for i ← last to S.length do
        if S[i] = ch then
            count ← count + 1
        end if
    end for
    if count ≥ k then
        NEXT SLIDE
    end if
end for
return T
```

```
for i ← last to S.length do
    if S[i] = ch then
        T ← T + ch
        new_last ← i
    end if
end for
last ← new_last
```

Problema de la mochila fraccionaria

Dado los pesos y valores de N artículos, debemos colocar estos artículos en una mochila de capacidad W para obtener el máximo valor total en la mochila. Siempre es posible tomar un parte o totalidad de cada uno de los artículos.
Ejemplos:
Input:
arr = [[60(b), 10(w)], [100(b), 20(w)], [120(b), 30(w)]]
W = 50
Output:
Maximum possible value = 220
by taking items of weight 20 and 30 kg

Procedure 9 FRACTIONAL_KNAPSACK

```
Input: A : Item, W : Integer
  SORT_DESC_BY_RATIO(A)
  currentWeight ← 0
  acum ← 0
  for i ← 1 to A.length do
    if currentWeight + A[i].weight <= W then
      currentWeight ← currentWeight + A[i].weight
      acum ← acum + A[i].value
    else
      remain ← W − currentWeight
      acum ← acum + A[i].value ∗ (remain/A[i].weight)
    end if
  end for
  return acum
```

Procedure 10 KRUSKAL

Input: $G : Graph(V, E)$

```
A ← ∅
for all v in V do
  INIT_SET(v)
  for all (u, v) ordered by weight(u, v), increasing do
    if FIND_SET(u) <> FIND_SET(v) then
      A ← A + (u, v)
      UNION(u, v)
    end if
  end for
end for
return A
```

- La estructura de datos Union-Find nos permite mantener conjuntos disjuntos. Tienen dos operaciones muy simples:
  - FIND(p, q) regresa verdadero si el conjunto al cual pertenece p está en el mismo conjunto que q.
  - UNION(p, q) une el conjunto al cual pertenece p con el conjunto al cual pertenece q.
- ¿Cómo podríamos impelementar esta estructura de datos?