Algoritmos ávidos

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

12-2022

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 grafos, 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
```

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: $S : Array, c : Integer$

$min \leftarrow 0$

$SORT\_DESC(S)$

for $i \leftarrow 1$ **to** $S.length$ **do**

  $min \leftarrow min + (c/S[i])$

  $c \leftarrow c \mod S[i]$

end for

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].

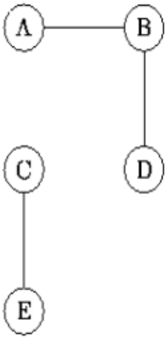
[1] `https://goo.gl/ZsAAzP`

Procedure 3 DFS

Input: $start : Vertex, g : Graph$

```
visited : Set
x_visit : Stack
x_visit.push(start)
while !x_visit.empty() do
    current ← x_visit.pop()
    if current ∉ visited then
        visited.add(current)
        for all v in current.connections() do
            x_visit.push(start)
        end for
    end if
end while
return visited
```

Considere un grafo G formado a partir de un gran número de vértices conectados por arcos. G se dice que está conectado si existe un camino entre cualquier par de vértices en G. Por ejemplo, el siguiente grafo no está conectado, porque no hay trayectoria de A a C.



Este grafo contiene, sin embargo, un número de subgrafos que están conectados, uno para cada uno de los siguientes conjuntos de vértices: (A), (B), (C), (D), (E), (A, B), (B,D), (C, E), (A, B, D). Un subgrafo conectado es máximo si no hay vértices y arcos en el grafo original que podrían añadirse al subgrafo y todavía dejarlo conectado. En la imagen anterior, hay dos subgrafos máximos, uno asociada con los vértices (A, B, D) y el otro con los vértices (C, E). Desarrollar un algoritmo para determinar el número de subgrafos máximos conectados de un gráfico dado.
`http://bit.do/eNTvC`

Procedure 4 COUNTING_GRAPHS

Input: G : $Graph$

$Reached$: Set

$acum \leftarrow 0$

Mark all the vertexes in G as $No\_Explored$

for $vertex$ in G **do**

  if $vertex$ is not marked $Explored$ **then**

    $DFS(vertex, G, Reached)$

    $acum \leftarrow acum + 1$

  end if

end for

return $acum$

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 5 ACTIVITIES_SELECTION

Input: $A : Activity\_Array$

$SORT\_ASC\_BY\_END(A)$
$i \leftarrow 1$
$S \leftarrow \emptyset + A[i]$
for $j \leftarrow 2$ **to** $A.length$ **do**
  if $A[j].start \geq A[i].end$ **then**
    $S \leftarrow S + A[j]$
    $i \leftarrow j$
  end if
end for
return S

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 6 BOOKING_PROBLEM

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

$SORT\_ASC(Arrival)$

$SORT\_ASC(Departure)$

$i \leftarrow 1$

$j \leftarrow 1$

$current \leftarrow 0$

$required \leftarrow 0$

while $i < Arrival.length$ **and** $j < Departure.length$ **do**
  if $Arrival[i] < Departure[j]$ **then**
    $current \leftarrow current + 1$
    $required = MAX(current, required)$
    $i \leftarrow i + 1$
  else
    $current \leftarrow current - 1$
    $j \leftarrow j + 1$
  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 $k \geq required$

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. [4]

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

[4] 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 es el producto de todos los números, excepto los ceros con una excepción. La excepción es cuando 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 ← 1 to A.length do
    if A[i] = 0 then
        count_zero ← count_zero + 1
    else
        if A[i] < 0 then
            count_neg ← count_neg + 1
            max_neg ← MAX(max_neg, count_neg)
        end if
        product ← product * A[i]
    end if
end for
```

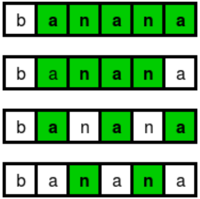
```
if count_zero = n then
  return 0
end if
if count_neg mód 2 = 1 then
  if count_neg = 1 and count_zero > 0
  and (count_neg + count_zero) = n then
    return 0
  end if
  product ← product/max_neg
end if
return product
```

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. [5]

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

Salida: nn



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

[5] `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 $i \leftarrow last$ **to** $S.length$ **do**
  if $S[i] = ch$ **then**
    $T \leftarrow T + ch$
    $new\_last \leftarrow i$
  end if
end for
$last \leftarrow new\_last$