Algoritmos ávidos

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos
Tecnológico de Monterrey

pperezm@tec.mx

06-2020

Contenido

Introducción

Ejemplos clásicos

Otros ejemplos

Estructuras ávidas

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 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 $C \neq \emptyset$ **and** $SOLUTION(S) =$ **false do**
    $x \leftarrow SELECT(C)$
    $S \leftarrow S + x$
    $C \leftarrow C - x$
  end while
  if $SOLUTION(S)$ **then**
    return S
  else
    return $\emptyset$
  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 ← 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
```

Word Transformation [2]

Un rompecabezas común que se encuentra en muchos periódicos y revistas es la transformación de palabras. Al tomar una palabra de inicio y alterar sucesivamente una sola letra para formar una nueva palabra, se puede construir una secuencia de palabras que cambia la palabra original a una palabra final dada. Por ejemplo, la palabra "spice"se puede transformar en cuatro pasos a la palabra "stock"de acuerdo con la siguiente secuencia: spice, slice, slick, stick, stock. Cada palabra sucesiva difiere de la palabra anterior en una sola posición de carácter, mientras que la longitud de la palabra sigue siendo la misma. Dado un diccionario de palabras de las cuales hacer transformaciones, más una lista de palabras iniciales y finales, escribe un programa para determinar el número de pasos en la transformación más corta posible.

[2] https://onlinejudge.org/external/4/429.pdf

Entrada

La entrada tendrá dos secciones. La primera sección será el diccionario de palabras disponibles con una palabra por línea, terminada por una línea que contiene un asterisco (*) en lugar de una palabra. Puede haber hasta 200 palabras en el diccionario; todas las palabras serán alfabéticas y en minúsculas, y ninguna palabra tendrá más de diez caracteres. Las palabras pueden aparecer en el diccionario en cualquier orden. Después del diccionario hay pares de palabras, un par por línea, con las palabras en el par separadas por un solo espacio. Estos pares representan las palabras iniciales y finales en una transformación. Se garantiza que todas las parejas tendrán una transformación utilizando el diccionario dado. Las palabras iniciales y finales aparecerán en el diccionario. Dos conjuntos de entrada consecutivos se separarán por una línea en blanco.

Salida

La salida debe contener una línea por par de palabras para cada conjunto de prueba, y debe incluir la palabra inicial, la palabra final y el número de pasos en la transformación más corta posible, separados por espacios individuales.

Ejemplo de entrada

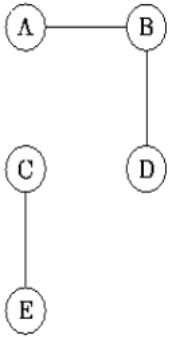
```
dip
lip
mad
map
maple
may
pad
pip
pod
pop
sap
sip
slice
slick
spice
stick
stock
*
spice stock
may pod
```

Ejemplo de salida

```
spice stock 4
may pod 3
```

Subgrafos máximos

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*

Topological Sort

Un "Topological Sort"de un Grafo Direccionado Acíclico (Directed Acyclic Graph, DAG) es un ordenamiento lineal de los vértices que aparecen en un DAG tal que si el vértice u aparece antes de v es porque existe un arco $(u \rightarrow v)$ en el DAG. Cada DAG tiene al menos, y posiblemente más, "topological sort".

Procedure 5 DFS2

Input: $u : Vertex, G : Graph, Reached : Set, TS : Stack$

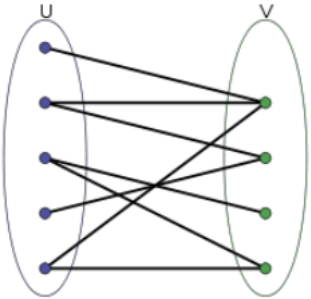
```
Mark u as Explored and add to Reached
for each (u, v) incident to u do
    if v is not marked Explored then
        DFS2(v, G, Reached, TS)
    end if
end for
TS.push(u)
```

Procedure 6 TOPOLOGIGAL_SORT

Input: G : *Graph*
  Reached : *Set*
  TS : *Stack*
  for each *vertex* in G **do**
    if *vertex* is not marked *Explored* **then**
      $DFS2(G, v, Reached, TS)$
    end if
  end for
  while TS is not empty **do**
    print $TS.top()$
    $TS.pop()$
  end while

Grafo bipartita

Un grafo bipartita (o bigrafo) $G = (V, E)$ es un grafo cuyos vértices pueden ser divididos en dos conjuntos disjuntos R y S tal que cada arco conecta a un vértice en R con un vértice en S.



Procedure 7 BIGRAPH

Input: G : *Graph*
  Q : *Queue*
  Color : *Array*
  isBipartite : *boolean*
  $INIT(Color, -1)$
  $isBipartite \leftarrow true$
  $vertex \leftarrow$ some vertex in G
  $Q.enqueue(vertex)$
  while Q is not empty **do**
    NEXT SLIDE
  end while
  return *isBipartite*

```
u ← Q.dequeue()
for each (u, v) incident in u do
  if Color[v] = −1 then
    Color[v] ← 1 − Color[u]
    Q.enqueue(v)
  else
    if Color[v] = Color[u] then
      isBipartite ← false
    end if
  end if
end for
```

Punto Articulado

Un punto articulado (o puente) se define como un vértice in un grafo $G = (V, E)$ cuya remoción (todos los arcos que inciden sobre él también son removidos desconecta el grafo G.Un grafo que no tiene ningún punto de articulación se le conoce como biconectado.

Procedure 8 FIND_POINT

Input: G : *Graph*

```
originalCC ← COUNTING_GRAPHS(G)
for each v in G do
    Remove v and its incident edges
    newCC ← COUNTING_GRAPHS(G)
    if newCC > originalCC then
        return true
    end if
    Restore v and its incident edges
end for
return false
```

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. [3]
Ejemplo:

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

[3] https://goo.gl/1RG25M

Procedure 9 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. [4]

[4] https://goo.gl/7e6idL

Procedure 10 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 ← 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 $k \geq required$

Dragón de Loowater [5]

Érase una vez, en el Reino de Loowater, una molestia menor se convirtió en un problema importante.
Las costas de Rellau Creek, en el centro de Loowater, siempre habían sido un excelente caldo de cultivo para los gansos. Debido a la falta de depredadores, la población de gansos estaba fuera de control. La gente de Loowater se mantuvo principalmente alejada de los gansos. Ocasionalmente, un ganso atacaría a una de las personas, y tal vez mordiera un dedo o dos, pero en general, la gente toleraba a los gansos como una molestia menor. Un día, ocurrió una mutación anormal, y uno de los gansos engendró un dragón de fuego múltiple con cabeza de fuego. Cuando el dragón creció, amenazó con quemar el Reino de Loowater hasta quebrarse. Loowater tuvo un gran problema. El rey se alarmó y pidió a sus caballeros que mataran al dragón y salvaran el reino.
Los caballeros explicaron: "Para matar al dragón, debemos cortar todas sus cabezas. Cada caballero puede cortar una de las cabezas del dragón. Las cabezas del dragón son de diferentes tamaños. Para cortar una cabeza, un caballero debe ser al menos tan alto como el diámetro de la cabeza. El sindicato de caballeros exige que, para cortar una cabeza, a un caballero se le pague un salario igual a una moneda de oro por cada centímetro de la altura del caballero".
¿Habría suficientes caballeros para derrotar al dragón? El rey llamó a sus asesores para que lo ayudaran a decidir cuántos caballeros contratar. Después de haber perdido mucho dinero construyendo Mir Park, el rey quería minimizar el gasto de matar al dragón. Como uno de los asesores, su trabajo era ayudar al rey. Lo tomaste muy en serio: si fallabas, ¡tú y todo el reino quedarían quemados!

[5] https://onlinejudge.org/external/112/11292.pdf

Entrada

La primera línea contiene dos números enteros entre 1 y 20000 inclusive, que indican el número n de cabezas que tiene el dragón y el número m de caballeros en el reino. Las siguientes n líneas contienen un número entero y dan los diámetros de las cabezas del dragón, en centímetros. Las siguientes líneas m contienen un número entero y especifican las alturas de los caballeros de Loowater, también en centímetros.

Salida

Genera una línea que contenga la cantidad mínima de monedas de oro que el rey debe pagar para matar al dragón. Si no es posible que los caballeros de Loowater maten al dragón, muestra la línea "¡Loowater está condenado!"

Ejemplo de entrada

2 3
5
4
7
8
4

Ejemplo de salida

11

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. [6]
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

[6] 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 11 MAXIMUM_PRODUCT

Input: $A : Array$
  if $n = 1$ **then**
    if $A[1] < 1$ **then**
      return 0
    else
      return $A[1]$
    end if
  end if
  $max\_neg \leftarrow INT\_MIN$
  $count\_neg \leftarrow 0$
  $count\_zero \leftarrow 0$
  $product \leftarrow 1$

```
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
```

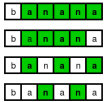
```
if count_zero = n then
  return 0
end if
if count_neg mód 2 = 1 then
  if count_neg = 1 and count_zero > 0
  and (count_neg + count_zero) = n then
    return 0
  end if
  product ← product/max_neg
end if
return product
```

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. [7]

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

Salida: *nn*



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

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

Procedure 12 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 13 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 14 KRUSKAL

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

$A \leftarrow \emptyset$
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**
$A \leftarrow A + (u, v)$
$UNION(u, v)$
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?

Amigos [8]

Hay una ciudad con N ciudadanos. Se sabe que algunos pares de personas son amigos. Según el famoso dicho que dice: "Los amigos de mis amigos también son mis amigos", se deduce que si A y B son amigos y B y C son amigos, entonces A y C también lo son.
Su tarea es contar cuántas personas hay en el grupo más grande de amigos.

[8] https://onlinejudge.org/external/106/10608.pdf

Entrada
La primera línea contiene los números N y M, donde N es el número de ciudadanos de la ciudad (1 <= N <= 30000) y M es el número de pares de personas (0 <= M <= 500000), que son conocidos por ser amigos. Cada una de las siguientes líneas M consta de dos enteros A y B (1 <= A <= N, 1 <= B <= N, A! = B) que describen que A y B son amigos. Podría haber repeticiones entre los pares dados. **Salida**
El resultado contiene (en una línea por sí mismo) un número que indica cuántas personas hay en el grupo más grande de amigos.

Ejemplo de entrada

10 12
1 2
3 1
3 4
5 4
3 5
4 6
5 2
2 1
7 1
1 2
9 10
8 9

Ejemplo de salida

7