Semana 2

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

07-2021

Backtracking

- Introducción
- Algunos ejemplos
  - 8 reinas
  - Permutaciones
  - Suma de subconjuntos
  - Prime Ring Problem

Introducción

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

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

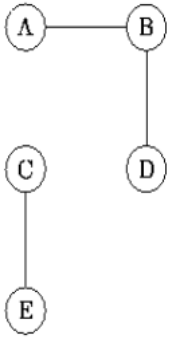
```
dip
lip
mad
map
maple
may
pad
pip
pod
pop
sap
sip
slice
slick
spice
stick
stock
*
spice stock
may pod
```

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*

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 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

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 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 ← 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 $k \geq required$

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 ← 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
```

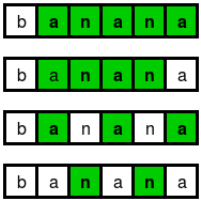
```
if count_zero = n then
  return 0
end if
if count_neg mód 2 = 1 then
  if count_neg = 1 and count_zero > 0
  and (count_neg + count_zero) = n then
    return 0
  end if
  product ← product/max_neg
end if
return product
```

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. [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
```

Introducción

- Existen problemas importantes para los cuales no se ha logrado encontrar un algoritmo eficiente para solucionarlos. Sin embargo, como son importantes, se tiene que buscar alguna forma de obtener su solución.
- En algunas ocasiones, la única herramienta con la que se cuenta es la búsqueda exhaustiva, esto es, buscar en todo el espacio de soluciones un elemento que cumpla con las condiciones para ser la solución.
- Cuando estos problemas son combinatorios, esto es, cuando su solución es una **permutación**, **combinación** o **subconjunto** de un conjunto de valores que pueden tomar las variables involucradas, el espacio de búsqueda es discreto

- La búsqueda exhaustiva podría funcionar para solucionar cualquier problema combinatorio, sin embargo, en algunos casos los espacios de solución son infinitos o muy grandes y este proceso se haría imposible de realizar en un tiempo adecuado debido a que el número posible de soluciones crece exponencialmente con el número de valores posibles de las variables.
- Backtracking es una mejora de la búsqueda exhaustiva. Va construyendo la solución paso a paso, en donde cada paso analiza los posibles valores que pueda tomar una variable. Utiliza una estructura de árbol cuya raíz es el inicio del problema y cada nivel está formado por los posibles valores que puede tomar una de las variables involucradas en el problema.

- Para no generar el árbol, utiliza el recorrido DFS.
- Al explorar un nodo, se verifica que la solución parcial hasta ese momento cumpla con las condiciones de una posible solución, si es así, este nodo es clasificado como "prometedor", por lo que se expande y se continúa el proceso de "primero en profundidad". Si, por el contrario, se encuentra que la solución parcial obtenida hasta ese momento no cumple con las condiciones para ser una solución al problema, ese nodo se clasifica como "no promisorio", por lo que se desecha, se regresa a su padre.

Forma general

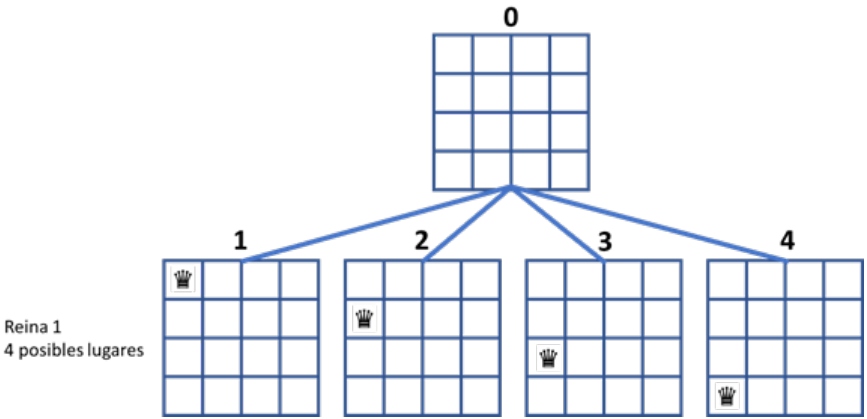
Procedure 9 BACKTRACKING

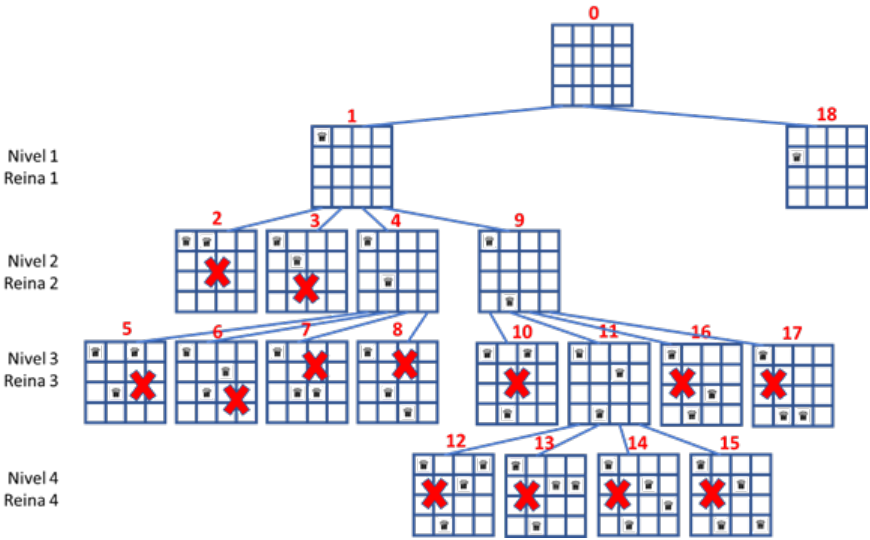
Input: *node*

```
if SOLUTION(node)) then
    print node
end if
if NOT(PROMISSORY(node))) then
    return
end if
for i ← 1 to CHILDS(node) do
    BACKTRACKING(i)
end for
```

Permutaciones

En un tablero de ajedrez (8X8 casillas) se deben colocar 8 reinas sin que se ataquen. Recordar que una reina puede atacar vertical, horizontal o diagonalmente, recorriendo cuantas casillas requiera. El problema se puede generalizar a n reinas, es decir, en un tablero de nXn casillas se colocan n reinas sin que se ataquen


0
1
2
3
4
Reina 1
4 posibles lugares


0
1
18
Nivel 1
Reina 1
2
3
4
9
Nivel 2
Reina 2
5
6
7
8
10
11
16
17
Nivel 3
Reina 3
12
13
14
15
Nivel 4
Reina 4

Procedure 10 CAN_PLACE

Input: $rows : Array, col : Integer, ren : Integer$

```
for i ← 1 to col do
  if rows[i] = ren or abs(rows[i] − ren) = abs(i − c) then
    return false
  end if
end for
return true
```

Procedure 11 EIGHT_QUEENS

Input: $rows : Array, a : Integer, b : Integer, col : Integer$

```
if c = 8 and rows[b] = a then
  print rows
end if
for ren ← 1 to 8 do
  if CAN_PLACE(rows, ren, col) then
    rows[col] ← ren
    EIGHT_QUEENS(rows, a, b, col + 1)
  end if
end for
```

Permutaciones

Hallar todas las permutaciones de un número N. Por ejemplo, las permutaciones de 123 son: 123, 231, 321, 312, 132, 213, 123.

Procedure 12 PERMUTATION

Input: $S : String, pos : Integer$

```
if pos = 0 then
  print S
else
  for i ← 1 to pos do
    SWAP(number, i, pos)
    PERMUTATION(number, pos − 1)
    SWAP(number, i, pos)
  end for
end if
```

Suma de subconjuntos

Dado un conjunto S de números enteros positivos, encontrar los subconjuntos que sumen la cantidad C. Si no se encuentra algún subconjunto que cumpla, se debe responder con el conjunto vacío.

Procedure 13 SUBSET_SUM

Input: $A : Array, S : Set, acum : Integer, target : Integer, level : Integer$

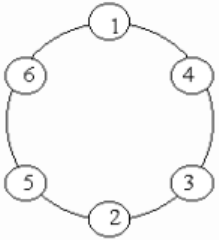
```
if acum = target then
  print solution
end if
if acum > target then
  return
end if
if level < A.length then
  SUBSET_SUM(A, S, acum, target, level + 1)
  S.INSERT(A[level])
  SUBSET_SUM(A, S, acum + solution[level], target, level + 1)
end if
```

Prime Ring Problem

524 Prime Ring Problem

A ring is composed of n (even number) circles as shown in diagram. Put natural numbers $1, 2, \ldots, n$ into each circle separately, and the sum of numbers in two adjacent circles should be a prime.

Note: the number of first circle should always be 1.



Input

n $(0 < n \leq 16)$

Output

The output format is shown as sample below. Each row represents a series of circle numbers in the ring beginning from 1 clockwisely and anticlockwisely. The order of numbers must satisfy the above requirements.

You are to write a program that completes above process.

[7] https://onlinejudge.org/external/5/524.pdf

Sample Input

```
6
8
```

Sample Output

```
Case 1:
1 4 3 2 5 6
1 6 5 2 3 4

Case 2:
1 2 3 8 5 6 7 4
1 2 5 8 3 4 7 6
1 4 7 6 5 8 3 2
1 6 7 4 3 8 5 2
```