Backtracking

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

01-2023

Contenido

- 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.

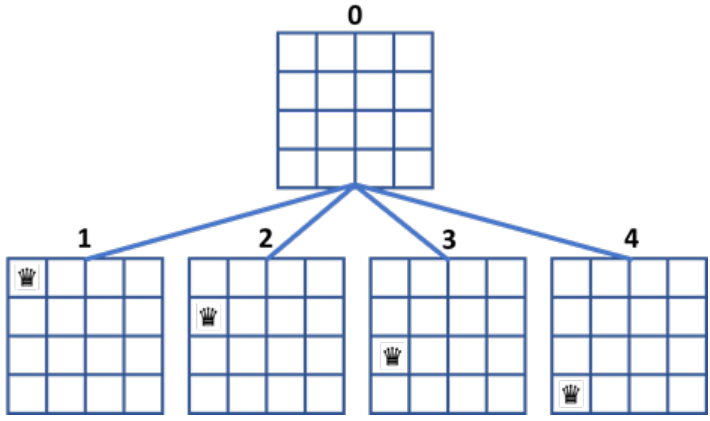
Procedure 1 BACKTRACKING

```
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
```

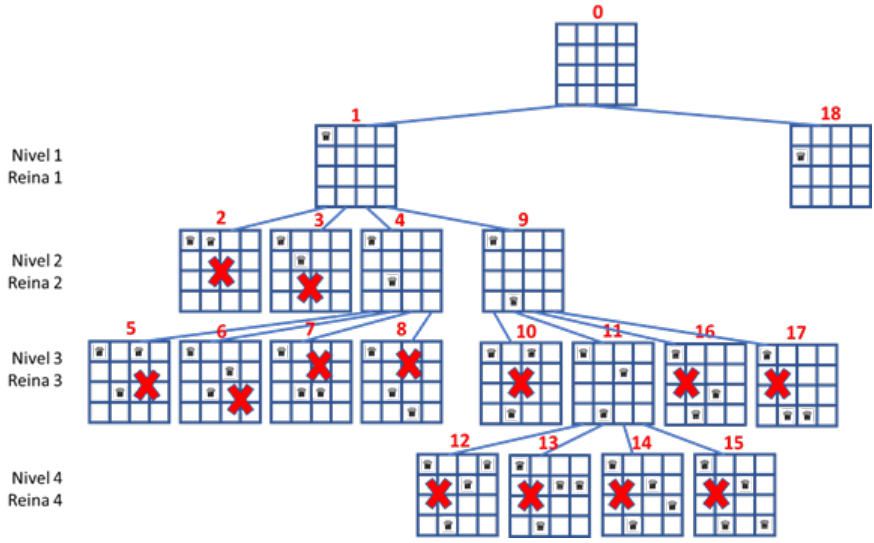
En un tablero de ajedrez ($8X8$ casillas) se deben colocar 8 reinas sin que se ataquen. Recordar que una reina puede atacar vertical, horizontal o diagonalmente, recorriendo cuantas casillas requiera. El problema se puede generalizar a N reinas, es decir, en un tablero de NXN casillas se colocan n reinas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

Procedure 2 EIGHT_QUEENS

```
EIGHT_QUEENS(ROWS, START_ROW, START_COL, 1)
```



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 3 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 4 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
```

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

Procedure 5 PERMUTATION

PERMITATION(S, S.length)

Procedure 6 PERMUTATION

Input: $S : String, pos : Integer$

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

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 7 SUBSET_SUM

```
SUBSET(A, S, 0, TARGET, 1)
```

Procedure 8 SUBSET_SUM

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

```
if acum = target then
  print solution
  return
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 + A[level], target, level + 1)
end if
```