APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 3: TÉCNICAS DE DISEÑO DE ALGORITMOS

Recorrida de grafos.

Recorriendo árboles binarios. Recorrer un grafo significa dar un algoritmo para visitar, o procesar, todos los nodos de un grafo. Nos interesan por ahora sólo las distintas estrategias para recorrer grafos, por ello nos conformamos con que visitar o procesar sea una actividad simbólica.

En el caso de árboles binarios, y dada la definición que hemos dado de ellos, naturalmente pensamos en un algoritmo que se aplicará o bien a un árbol vacío, en cuyo caso no hay ningún nodo por visitar, o bien en un nodo que tiene un elemento, un subárbol izquierdo y un subárbol derecho. Recorrer este árbol consiste en

- visitar este mismo nodo, por ejemplo, procesando el elemento que se encuentra en él,
- visitar los nodos del subárbol izquierdo, es decir, recorrer el subárbol izquierdo, y
- recorrer el subárbol derecho

Realizadas estas tres acciones el árbol en cuestión habrá sido recorrido. Se desprenden naturalmente 3 estrategias para recorrerlo:

pre-orden: Primero se visita el nodo, y luego se recorren los subárboles izquierdo y derecho.

in-orden: Primero se recorre el subárbol izquierdo, luego se visita el nodo, y finalmente se recorre el subárbol derecho.

pos-orden: Primero se recorren los subárboles izquierdo y derecho y finalmente se visita el nodo.

Otras 3 estrategias naturales se obtienen recorriendo de derecha a izquierda:

pre-orden, der-izq: Primero se visita el nodo, y luego se recorren los subárboles derecho e izquierdo.

in-orden, der-izq: Primero se recorre el subárbol derecho, luego se visita el nodo, y finalmente se recorre el subárbol izquierdo.

pos-orden, der-izq: Primero se recorren los subárboles derecho e izquierdo y finalmante se visita el nodo.

Para ver un ejemplo, pensemos en un algoritmo que devuelve una lista con todos los nodos elementos del árbol (respetando las repeticiones). Para ello es necesario recorrer todo el árbol buscando los elemento para devolver en la lista. Evidentemente, alcanza con visitar cada nodo una vez. Hay al menos 6 maneras de hacerlo según cuál de las 6 estrategias enumeradas se utilice. Por ejemplo, en la recorrida pre-orden el algoritmo se puede escribir:

pre_orden(<>) = [ ]
pre_orden($< l, e, r >$) = $e \triangleright$pre_orden(l) ++ pre_orden(r)

En la recorrida in-orden, se obtieneel siguiente algoritmo

in_orden(<>) = [ ]
in_orden($< l, e, r >$) = in_orden(l) ++ ($e \triangleright$in_orden(r))

Por último, en la recorrida pos-orden se obtiene:

pos_orden(<>) = [ ]
pos_orden($< l, e, r >$) = pos_orden(l) ++ pos_orden(r) $\triangleleft e$

Es importante notar que si utilizamos el algoritmo in_orden en un ABB obtenemos una lista ordenada de menor a mayor. Efectivamente, para todo nodo del árbol, in_orden lista primero todos los elementos del subárbol izquierdo (que por tratarse de un ABB son todos los menores al elemento del nodo), luego el elemento del nodo, y finalmente los elementos del subárbol derecho (que por tratarse de un ABB son todos los mayores al elemento del nodo).

Si dado un ABB quisiéramos listar los elementos de mayor a menor podríamos utilizar el algoritmo in_orden_der_izq:

in_orden_der_izq(<>) = [ ]
in_orden_der_izq($< l, e, r >$) = in_orden_der_izq(r) ++ ($e \triangleright$in_orden_der_izq(l))

Observar que los 6 algoritmos sirven para recorrer cualquier árbol binario. Sólo hemos mencionado una propiedad adicional que se cumple en el caso en que alguno de los algoritmos in-orden se aplica a un ABB.

Recorriendo árboles finitarios. Un árbol finitario es similar a un árbol binario, sólo que en vez de tener 2 subárboles cada nodo tiene una cantidad (cualquiera) finita de subárboles. Podemos pensar en un grafo G = (N,raíz,hijos) donde raíz es la raíz del árbol e hijos es la función que aplicada a un nodo del árbol devuelve una lista de los nodos que son sus hijos. El orden en que esa lista enumera los hijos es irrelevante. Sí es importante que si se aplica dos veces hijos al mismo nodo, en ambos casos listará los hijos en el mismo orden.

Para el caso de árboles finitarios, las 3 primeras estrategias vistas anteriormente se reducen a 2. En efecto, la que deja de ser una estrategia natural es la in_orden. Habiendo una cantidad arbitraria de hijos, ¿entre cuáles hijos visitaríamos al nodo en sí?. Las otras dos siguen siendo fácil de enunciar:

pre-orden: Primero se visita el nodo, y luego se recorren los hijos en el orden que son enumerados por hijos.

pos-orden: Primero se recorren los hijos en el orden que son enumerados por hijos y finalmante se visita el nodo.

Estas 2 estrategias abarcan las versiones de izquierda a derecha y de derecha a izquierda (y muchas más) dado que hemos abstraído el orden en que hijos enumera los hijos.

A continuación escribimos un algoritmo que visita todos los nodos de un árbol finitario en pre-orden:

fun pre_order(G=(N,root,children)) **ret** m: marks
    init(m)

```
    pre_order_rec(G, m, root)
end
proc pre_order_rec(in G, in/out m: marks, in n: N)
     visit(m,n)
     for c ∈ children(n) do pre_order_rec(m, c) od
end
```

Si queremos que el algoritmo le asigne a cada nodo un número que indique en qué orden los fue visitando, definimos

```
type marks = tuple
                ord: array[N] of int
                count: int
             end
proc init(out m: marks)
     m.count:= 0
end
proc visit(in/out m: marks, in n: N)
     m.cont:= m.count+1
     m.ord[n]:= m.count
end
```

Observar que no es necesario inicializar las celdas del arreglo ya que, tratándose de un árbol y dado que se comienza por la raíz cada nodo será visitado exactamente una vez y en ese momento la celda correspondiente inicializada.

A diferencia del anterior, el siguiente algoritmo recorre el árbol en pos-orden:

```
fun pos_order(G=(N,root,children)) ret m: marks
    init(m)
    pos_order_rec(G, m, root)
end
proc pos_order_rec(in G, in/out m: marks, in n: N)
     for c ∈ children(n) do pos_order_rec(m, c) od
     visit(m,n)
end
```

A continuación, definimos para $n, m \in N$ la siguiente relación: $n \preceq m$ sii existen $p, q, r \in N$ tales que p y q aparecen en children(r) en ese orden, p es ancestro de n, y q es ancestro de m. Esta relación es antisimétrica, es decir, $n \preceq m \wedge n \succeq m \Rightarrow n = m$.

Se puede demostrar que si pre_mark = pre_order(G) y pos_mark = pos_order(G) entonces para todo $n, m \in N$ se cumple:

$$\begin{array}{rcl} \text{pre\_mark}[n] \leq \text{pre\_mark}[m] & \iff & n \preceq m \vee n \text{ es ancestro de } m \\ \text{pos\_mark}[n] \geq \text{pos\_mark}[m] & \iff & n \succeq m \vee n \text{ es ancestro de } m \end{array}$$

Precondicionamiento. Estas observaciones nos permiten dar un ejemplo de una técnica que suele llamarse precondicionamiento. Se trata de elaborar los datos de determinada manera con el fin de facilitar luego su utilización. Por ejemplo, si tenemos n números naturales $m_1, m_2, \ldots, m_n$ y queremos averiguar para una gran cantidad de números

inicialmente desconocidos $k_1, k_2, \ldots, k_M$ si cada uno de ellos es divisible por $m_1, m_2, \ldots,$ y m_n o no. En vez de testear para cada $1 \leq i \leq M$ si k_i es divisible por m_1, luego por m_2, etc. podemos preprocesar los datos de la siguiente forma. Calculamos el mínimo común múltiplo m de $m_1, m_2, \ldots, m_n$ y luego simplemente testeamos para cada $1 \leq i \leq M$ si k_i es divisible por m. Al calcular ese mínimo común múltiplo estamos "precondicionando", creando condiciones favorables para realizar las cuentas siguientes.

Otro ejemplo se puede obtener para árboles finitarios. Si dado un árbol finitario quiero averiguar para una gran cantidad de pares de nodos inicialmente desconocidos $(n_1, m_1), \ldots, (n_M, m_M)$ si n_i es ancestro de m_i de manera eficiente, conviene calcular los arreglos pre_mark y pos_mark. Por las observaciones realizadas 2 párrafos más arriba, n_i será ancestro de m_i sii pre_mark[n] $\preceq$ pre_mark[m] y pos_mark[n] $\succeq$ pos_mark[m].

```
fun ancestro(pre_mark, pos_mark: marcas, n, m: int) ret b: bool
    b:= (pre_mark[n] ≤ pre_mark[m] ∧ pos_mark[n] ≥ pos_mark[m])
end
```

Búsqueda en profundidad (DFS). Las dos estrategias vistas para recorrer árboles finitarios tienen algo en común: ambas corresponden a realizar recorridas "en profundidad," es decir, antes de visitar el segundo hijo de cada nodo se visitan todos los descendientes del primer hijo. Esto es lo que se llama DFS (Depth-First Search), búsqueda en profundidad. Esta estrategia de búsqueda es muy útil en la práctica incluso para grafos que no sean necesariamente árboles. A continuación vemos cómo extender la estrategia a grafos arbitrarios.

Uno se enfrenta esencialmente con dos problemas al intentar extender los algoritmos de recorrida a grafos que no sean necesariamente árboles: la posibilidad de que no sean conexos y la posible existencia de ciclos. Si el grafo no es conexo, no podremos recorrer todo el grafo tan solo partiendo de un nodo. Si el grafo tiene ciclos, nuestra recorrida en profundidad puede llevarnos a un nodo ya visitado e incluso, si no se tiene precaución, a una recorrida que no termine.

Para cuidar estos aspectos modificamos ligeramente nuestras primitivas para manipular marcas de modo de poder averiguar en cualquier momento si un nodo fue visitado. Redefinimos el procedimiento inicializar y definimos una nueva función booleana visitado.

```
proc init(out m: marks)
      m.count:= 0
      for n ∈ N do m.ord[n]:= 0 od
end

fun visited(m: marks, n: N) ret b: bool
    b:= (m.ord[n] ≠ 0)
end
```

Con estas definiciones es posible definir la búsqueda en profundidad. Dicha búsqueda se asemeja a la que uno podría realizar de encontrarse encerrado en un laberinto teniendo la posibilidad de marcar con piedritas los lugares recorridos en la búsqueda de la salida. La búsqueda partiría del lugar donde nos encontramos dejando caer periódicamente piedritas de forma que queden marcados los lugares ya recorridos. Cuando llegamos

a una división del camino elegimos la primera de la derecha y continuamos, siempre marcando el camino elegido. Cuando llegamos a un lugar marcado, damos marcha atrás hasta encontrar la última división en la que nos queden alternativas sin marcar. Intentamos la primer alternativa de la derecha que aún no fue marcada y volvemos a avanzar marcando.

En pseudo-código este algoritmo de búsqueda se puede escribir:

```
fun dfs(G=(N,neighbours)) ret m: marks
    init(m)
    for n ∈ N do
        if ¬visited(m,n) then dfs_rec(G, m, n) fi
    od
end

proc dfs_rec(in G, in/out m: marks, in n: N)
      visit(m,n)
      for c ∈ neighbours(n) do
          if ¬visited(m,c) then dfs_rec(G, m, c) fi
      od
end
```

El procedimiento recursivo dfs_rec es muy similar al procedimiento pre_order_rec visto para árboles salvo que en vez de children(n) usa neighbours(n) (un nombre más apropiado para denominar la lista de todos los nodos vecinos al nodo n teniendo en cuenta que G no es necesariamente un árbol) y que antes de realizar una llamada recursiva se asegura de que el vecino en cuestión no haya sido visitado. Esto evita visitar más de una vez un mismo nodo, y con ello que la recorrida se pierda en un ciclo del grafo.

El procedimiento principal dfs también es similar al procedimiento pre_order que vimos para árboles salvo que en vez de iniciar la recorrida sólo a partir de la raíz la inicia dentro de un **for**. Es decir, la inicia tantas veces como sea necesario a partir de diferentes nodos. Esto soluciona el problema que podría presentarse al recorrer grafos no conexos: se iniciaría la búsqueda una vez por componente conexa. Observar que sólo se inicia la búsqueda a partir de un nodo si el mismo no ha sido ya visitado en una búsqueda iniciada en un nodo anterior.

Por último, es importante destacar que este algoritmo se comporta de forma adecuada tanto para grafos dirigidos como para grafos no dirigidos. En efecto, la única diferencia entre un caso y el otro estaría dada por la función neighbours que en el caso de grafos no dirigidos debe satisfacer la condición adicional n' ∈ neighbours(n) sii n ∈ neighbours(n').

¿Cómo hacer una versión análoga de dfs que corresponda a pos_order en vez de a pre_order? Hay que tener cuidado con los ciclos.

Algoritmo iterativo. Es posible dar una versión iterativa dfs_ite del procedimiento dfs_rec utilizando una pila de nodos. En este caso la pila almacenará el camino que va desde el nodo donde se inició el procedimiento dfs hasta el nodo que actualmente se está visitando, nodo que estará en el tope de la pila. Si este nodo tiene algún vecino sin visitar, se lo visita y agrega a la pila. Si no, se lo borra de la misma.

```
proc dfs_ite(in G, in/out m: marks, in n: N)
    var p: stack of N
    empty(p)
    visit(m,n)
    push(n,p)
    while ¬is_empty(p) do
        if existe c ∈ neighbours(top(p)) tal que ¬visited(m,c) then
            visit(m,c)
            push(c,p)
        else pop(p)
        fi
    od
end
```

En el procedimiento principal dfs se reemplaza la invocación a dfs_rec por una idéntica a dfs_ite.

Búsqueda a lo ancho (BFS). Como expresamos en la sección anterior, la búsqueda en profundidad es aquélla en que para cada nodo n se visita al i-ésimo vecino de n sólo después de que se hayan visitado todos los nodos alcanzables desde los vecinos anteriores a i. La posibilidad contraria, es decir, aquélla en que para cada nodo n se visite al i-ésimo vecino de n antes de que se visiten los nodos no-vecinos alcanzables desde los demás vecinos, se denomina BFS (Breadth-First Search) búsqueda a lo ancho.

A diferencia de DFS, no hay una definición recursiva natural del algoritmo BFS. De todas maneras, es muy fácil modificar la versión iterativa de DFS para que realice búsqueda a lo ancho: basta con reemplazar la pila por una cola.

```
proc bfs_ite(in G, in/out m: marks, in n: N)
    var q: queue of N
    empty(q)
    visit(m,n)
    enqueue(q,n)
    while ¬is_empty(q) do
        if existe c ∈ neighbours(first(q)) tal que ¬visited(m,c) then
            visit(m,c)
            enqueue(q,c)
        else dequeue(q)
        fi
    od
end

fun bfs(G=(N,neighbours)) ret m: marks
    init(mark)
    for n ∈ N do
        if ¬visited(m,n) then bfs_ite(G, m, n) fi
    od
end
```

Backtracking. Backtracking es una técnica de programación que se aplica a problemas que requieren la construcción de una solución de manera incremental, paso a paso, donde en cada paso hay una cantidad finita de posibilidades y no se sabe cuál o cuáles llevan a la solución o a las soluciones. Eso significa que después de varios pasos puede que se demuestre que no se esté arribando a la solución, por lo que algunas decisiones anteriores deben revisarse y reemplazarse por otras. Backtracking es un mecanismo adecuado que contempla intentar todas las combinaciones posibles de manera de que si una solución existe la misma será encontrada. Backtracking corresponde a realizar una búsqueda en profundidad en un grafo implícito. Es decir, un grafo que no está presente en el enunciado del problema, pero que puede explicitarse si uno lo desea expresando cuáles son en cada paso las diferentes posibilidades que podrían llevar hacia la solución.

Problema de la mochila. El primer problema que intentaremos resolver utilizando backtracking es el de la mochila. Recordemos que tenemos n objetos de valor $v_1, \ldots, v_n$ y peso $w_1, \ldots, w_n$ y una mochila de capacidad W. Se quiere obtener el máximo valor posible sin exceder la capacidad de la mochila.

Una manera de analizar este problema es pensando que en el paso i se decide si se incorpora o no el objeto i. Explicitando el grafo, tendríamos

$$N = \{p \in \{0,1\}^* \mid |p| \leq n \wedge w(p) \leq W\}$$
$$w([x_1, \ldots, x_k]) = \sum_{i=1}^{k} x_i * w_i$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{0,1\}.q = p \lhd b)$$

Los nodos son secuencias de 0's y 1's. El nodo $[0,0,1,1]$ corresponde al caso en que, de los primeros 4 objetos, se ha decidido guardar en la mochila solamente el tercero y el cuarto. Los demás objetos, desde el quinto al n-ésimo, aún no han sido considerados. Una arista de p a q consiste en tomar una decisión (0 ó 1) respecto al siguiente objeto. De $[0,0,1,1]$ sólo hay dos aristas: una a $[0,0,1,1,0]$ y otra a $[0,0,1,1,1]$.

En el algoritmo este grafo sólo es implícito. El siguiente algoritmo determina el máximo valor alcanzable sin exceder el peso utilizando backtracking:

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of peso, j: peso, i: int) ret r: valor
    {calcula el máximo valor alcanzable con los objetos i, ..., n sin exceder el peso j}
  if i > n then r:= 0
  else if w[i] >j then r:= mochila(v,w,j,i+1)
       else r:= máx(mochila(v,w,j,i+1),v[i]+mochila(v,w,j-w[i],i+1))
       fi
  fi
end
```

La función principal simplemente llama a la anterior con j igual a W e i igual a 1.

Otra posibilidad que da lugar a un grafo implícito más pequeño es pensar que a cada paso se decide incorporar cualquier objeto (que no supere el peso restante). El grafo se puede explicitar como sigue

$$N = \{p \subseteq \{1, \ldots, n\} \mid |p| \leq n \wedge w(p) \leq W\}$$
$$w(p) = \sum_{k \in p} w_k$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, n\}.q = p \lhd b \wedge \forall k \in p.k < b)$$

Los nodos son ahora conjuntos de objetos. Cada uno corresponde al caso en que dichos objetos hayan sido guardados en la mochila. Para no considerar más de una vez los mismos casos, sólo se pueden agregar objetos posteriores a los que ya se agregaron. Esto se refleja en la condición $\forall k \in p.k < b$ que dice justamente que el objeto b que se agrega es posterior a todos los que ya estaban.

Nuevamente este grafo da lugar a un algoritmo que utiliza backtracking:

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of peso, j: peso, i: int) ret r: valor
    {calcula el máximo valor alcanzable con los objetos i, ..., n sin exceder el peso j}
    r:= 0
    for k:= i to n do
        if w[k] ≤ j then r:= máx(r,v[k]+mochila(v,w,j-w[k],k+1)) fi
    od
end
```

Como en el caso anterior, la función principal es

```
fun mochila_ppal(v:array[1..n] of valor, w:array[1..n] of peso, W: peso) ret r: valor
    {calcula el máximo valor alcanzable con todos los objetos sin exceder el peso W}
    r:= mochila(v,w,W,1)
end
```

Ocho reinas. Otro problema interesante para resolver usando la técnica del backtracking es el de calcular el número de maneras diferentes de ubicar 8 reinas (o damas) en un tablero de ajedrez de manera de que ninguna de ellas amenace a ninguna de las demás. Recordemos que una reina amenaza todas las casillas que se encuentran en la misma fila, columna o diagonal que ella. A escala más pequeña puede formularse el problema con 4 reinas en un tablero de 4 filas por 4 columnas. No es difícil encontrar una solución al problema manualmente.

Primer intento. Lo primero que podríamos hacer es revisar todas las maneras posibles de ubicar 8 reinas en un tablero de ajedrez y controlar para cada una de esas distribuciones si hay una reina que ataque a otra. Para ubicar la primer reina tenemos 64 celdas posibles, para la segunda 63, etc. Si somos un poco más cuidadosos podemos evitar permutaciones (da lo mismo si la reina que está en una celda fue la primera o la quinta que acomodamos). Para ello cada reina será ubicada en casillas posteriores a las reinas que ya se ubicaron. Esto da lugar al algoritmo

```
fun ocho_reinas_1() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    var sol: list of int
    r:= 0
    for i1:= 1 to 57 do
        for i2:= i1+1 to 58 do
            for i3:= i2+1 to 59 do
                for i4:= i3+1 to 60 do
                    for i5:= i4+1 to 61 do
                        for i6:= i5+1 to 62 do
                            for i7:= i6+1 to 63 do
```

```
                                for i8:= i7+1 to 64 do
                                    sol:= [i1,i2,i3,i4,i5,i6,i7,i8]
                                    if solucion_1(sol) then r:= r+1 fi
                                od
                            od
                        od
                    od
                od
            od
        od
    od
end
```

donde asumimos que solucion_1 se encarga de verificar si la lista con las posiciones de las 8 reinas es o no una solución.

Para la primera reina hay 57 posibilidades en vez de 64 ya que debe dejar como mínimo 7 casillas libres al final para las siguientes 7 reinas.

Para hacer explícito el grafo tomamos

$$N = \{[p_1, p_2, \ldots, p_n] \in \{1, \ldots, 64\}^* \mid n \leq 8 \wedge p_1 < p_2 < \ldots < p_n \leq 56 + n\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 64\}.q = p \lhd b)$$

Segundo intento. Si bien ya tenemos una solución, evidentemente la misma considera demasiadas posibilidades que fácilmente podrían descartarse. Por ejemplo, si la primer reina y la segunda se colocan en la misma fila, no importa donde se coloquen las demás, no obtendremos una solución. Sin embargo el algoritmo anterior considera todas las (muchísimas) formas posibles de colocar las restantes 6 reinas.

A continuación presentamos un algoritmo mejor, que considera sólo las distintas maneras de colocar 8 reinas en filas diferentes, condición necesaria para obtener una solución. La primer reina irá a la primer fila, la segunda reina a la segunda fila, etc. Por ello, para cada reina se determina sólo un número entre 1 y 8, el de la columna que le corresponde en la fila que ya tiene asignada.

```
fun ocho_reinas_2() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    var sol: list of int
    r:= 0
    for j1:= 1 to 8 do
        for j2:= 1 to 8 do
            for j3:= 1 to 8 do
                for j4:= 1 to 8 do
                    for j5:= 1 to 8 do
                        for j6:= 1 to 8 do
                            for j7:= 1 to 8 do
                                for j8:= 1 to 8 do
                                    sol:= [j1,j2,j3,j4,j5,j6,j7,j8]
                                    if solucion_2(sol) then r:= r+1 fi
```

```
                    od
                od
            od
        od
    od
od
od
od
end
```

Para hacer explícito el grafo tomamos

$$N = \{p \in \{1, \ldots, 8\}^* \mid |p| \leq 8\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 8\}.q = p \lhd b)$$

Antes de pasar al tercer intento conviene presentar una versión recursiva de este algoritmo:

```
fun ocho_reinas_2() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    or_2([ ], r)
end
```

donde el procedimiento or_2 es como sigue

```
proc or_2(in sol: list of int, in/out r: int)
                                        {calcula el número de maneras de extender sol}
                                    {hasta ubicar en total 8 reinas sin que se amenacen}
    if |sol| = 8 then
      if solucion_2(sol) then r:= r+1 fi
    else for j:= 1 to 8 do
            or_2(sol ◁ j, r)
        od
    fi
end
```

El algoritmo recursivo es fácilmente modificable para resolver el mismo problema para una cantidad arbitraria de reinas (y tableros suficientemente grandes).

Observar que la función solucion_2 es diferente a la solucion_1 del primer intento, ya que ésta recibe listas de números entre 1 y 8 mientras que aquélla recibía listas de números entre 1 y 64.

Tercer intento. Así como observarmos que para encontrar una solución dos reinas no pueden ir en la misma fila y logramos mejorar el algoritmo para tener eso en cuenta, a continuación lo mejoraremos para tener en cuenta que, análogamente, dos reinas no pueden ir en la misma columna. Por suerte la lista sol contiene exactamente las columnas que ya han sido ocupadas.

```
fun ocho_reinas_3() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
```

```
    r:= 0
    or_3([ ], r)
end

proc or_3(in sol: list of int, in/out r: int)
                                        {calcula el número de maneras de extender sol}
                                    {hasta ubicar en total 8 reinas sin que se amenacen}
    if |sol| = 8 then
      if solucion_3(sol) then r:= r+1 fi
    else for j:= 1 to 8 do
            if j ∉ sol then or_3(sol ◁ j, r) fi
         od
    fi
end
```

En este caso, solucion_3 es idéntico a solucion_2.

Haciendo explícito el grafo tenemos

$$N = \{p \in \{1, \ldots, 8\}^* \mid |p| \leq 8 \wedge p \text{ sin repeticiones}\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 8\}.q = p \lhd b)$$

Cuarto intento. Por último, para reducir aún más el espacio de búsqueda podemos considerar, para cada nueva reina, sólo aquéllas diagonales que aún no han sido ocupadas. Llamaremos bajadas y subidas respectivamente a las listas que llevan la cuenta de las diagonales que bajan de izquierda a derecha y las que suben de izquierda a derecha. Asumimos que tenemos dos funciones bajada(i,j) y subida(i,j) que dicen a qué diagonal (en bajada y en subida respectivamente) pertenece la casilla (i,j).

```
fun ocho_reinas_4() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    or_3([ ], [ ], [ ], r)
end

proc or_3(in sol, bajadas, subidas: list of int, in/out r: int)
                                        {calcula el número de maneras de extender sol}
                                    {hasta ubicar en total 8 reinas sin que se amenacen}
                                {bajadas y subidas son las diagonales ya amenazadas}
    if |sol| = 8 then r:= r+1 fi
    else i:= |sol|+1
         for j:= 1 to 8 do
             if j ∉ sol ∧ bajada(i,j) ∉ bajadas ∧ subida(i,j) ∉ subidas
             then or_3(sol ◁ j, bajadas ◁ bajada(i,j), subidas ◁ subida(i,j), r)
             fi
         od
    fi
end
```

Observar que dadas las restricciones consideradas al agregar una nueva reina, una vez que se han ubicado las 8 no es necesario usar una función auxiliar solucion_4 ya que efectivamente ninguna reina atacará a otra.

Por último, dado que todas las casillas (i,j) que comparten una misma bajada (y sólo ellas) dan idéntico valor a la expresión i-j y todas las casillas que comparten una misma subida (y sólo ellas) dan idéntico valor a la expresión i+j, definimos

fun bajada(i,j:**int**) **ret** r: **int**
    r:= i-j
end
fun subida(i,j:**int**) **ret** r: **int**
    r:= i+j
end

Haciendo el grafo explícito tenemos

$$N = \{[p_1, \ldots, p_n] \in \{1, \ldots, 8\}^* \mid n \leq 8 \wedge$$
$$\wedge (i \neq j \Rightarrow p_i \neq p_j) \wedge (i \neq j \Rightarrow p_i - i \neq p_j - j) \wedge (i \neq j \Rightarrow p_i + i \neq p_j + j)\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 8\}.q = p \lhd b)$$