APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 3: TÉCNICAS DE DISEÑO DE ALGORITMOS

Recorrida de grafos

A pesar de aplicarse a problemas de diferente naturaleza (no necesariamente de grafos) backtracking puede describirse como una técnica que consiste en recorrer un cierto grafo, muchas veces implícito en el problema. Comenzamos esta sección con algunos algoritmos de recorrida de grafos: árboles binarios, árboles finitarios, grafos dirigidos y no dirigidos.

Recorriendo árboles binarios. Recorrer un grafo significa dar un algoritmo para visitar, o procesar, todos los vértices 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: no importa en qué consista realmente la visita o procesamiento de los vértices.

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 vértice por visitar, o bien en un vértice que tiene un elemento, un subárbol izquierdo y un subárbol derecho. Recorrer este árbol consiste en

- visitar este mismo vértice, procesando el elemento que se encuentra en él,
- visitar los vértices 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-order: Primero se visita el vértice, y luego se recorren los subárboles izquierdo y derecho.

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

pos-order: Primero se recorren los subárboles izquierdo y derecho y finalmente se visita el vértice.

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

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

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

pos-order, der-izq: Primero se recorren los subárboles derecho y izquierdo y finalmente se visita el vértice.

Para ver un ejemplo, pensemos en un algoritmo que devuelve una lista con todos los vértices del árbol. Para ello es necesario recorrer todo el árbol buscando los elementos

para devolver en la lista. Evidentemente, alcanza con visitar cada vértice una vez. Hay al menos 6 maneras de hacerlo según cuál de las 6 estrategias enumeradas se utilice. Por ejemplo, en la recorrida pre-order el algoritmo se puede escribir:

$\text{pre\_order}(<>) = [\,]$
$\text{pre\_order}(< l, e, r >) = e \rhd \text{pre\_order}(l) \mathbin{++} \text{pre\_order}(r)$

En la recorrida in-order, se obtiene el siguiente algoritmo

$\text{in\_order}(<>) = [\,]$
$\text{in\_order}(< l, e, r >) = \text{in\_order}(l) \mathbin{++} (e \rhd \text{in\_order}(r))$

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

$\text{pos\_order}(<>) = [\,]$
$\text{pos\_order}(< l, e, r >) = \text{pos\_order}(l) \mathbin{++} \text{pos\_order}(r) \lhd e$

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

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

$\text{in\_order\_der\_izq}(<>) = [\,]$
$\text{in\_order\_der\_izq}(< l, e, r >) = \text{in\_order\_der\_izq}(r) \mathbin{++} (e \rhd \text{in\_order\_der\_izq}(l))$

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-order se aplica a un ABB.

Recorriendo árboles finitarios. Un árbol finitario es similar a un árbol binario, sólo que en vez de tener a lo sumo dos subárboles, cada vértice tiene una cantidad (cualquiera) finita de subárboles. Asumimos que el árbol viene dado por un grafo $G = (V, \text{root}, \text{children})$ donde $\text{root} \in V$ es la raíz y children es la función que aplicada a un vértice del árbol devuelve el conjunto de los vértices que son sus hijos.[13]

Para el caso de árboles finitarios, las tres primeras estrategias naturales vistas anteriormente se reducen a dos. La que deja de ser una estrategia natural es la in_order (habiendo una cantidad arbitraria de hijos, ¿entre las recorridas de cuáles de ellos visitaríamos al propio vértice?). Las otras dos siguen siendo fácil de enunciar:

pre-order: Primero se visita el vértice, y luego se recorren los hijos.
pos-order: Primero se recorren los hijos, y luego se visita el vértice.

[13] Se asume que el grafo es un árbol finitario. En términos de root y children eso puede formalizarse: decimos que v_0 es ancestro propio de v_{n+1} si existen $v_1, \ldots, v_n \in V$ tales que si $0 \leq i \leq n$ entonces $v_{i+1} \in \text{children}(v_i)$. Para que G sea un árbol finitario se asume que root es ancestro propio de todos los demás vértices (es decir, es conexo), que ninguno es ancestro propio de sí mismo (es decir, no tiene ciclos) y que que si v y w son vértices diferentes, $\text{children}(v)$ y $\text{children}(w)$ son disjuntos (es decir, cada vértice tiene a lo sumo un padre).

El orden en que se recorren los hijos no es importante, pero si se quiere precisar se puede requerir que children devuelva una lista en vez de un conjunto, los hijos serían recorridos siguiendo el orden establecido en dicha lista.

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

```
fun pre_order(G=(V,root,children)) ret mark: marks
    init(mark)
    pre_traverse(G, mark, root)
end

proc pre_traverse(in G, in/out mark: marks, in v: V)
     visit(mark,v)
     for w ∈ children(v) do pre_traverse(mark, w) od
end
```

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

```
type marks = tuple
                ord: array[V] of nat
                cont: nat
             end

proc init(out mark: marks)
     mark.cont:= 0
end

proc visit(in/out mark: marks, in v: V)
     mark.cont:= mark.cont+1
     mark.ord[v]:= mark.cont
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 vértice será visitado exactamente una vez y en ese momento la celda correspondiente será inicializada.

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

```
fun pos_order(G=(V,root,children)) ret mark: marks
    init(mark)
    pos_traverse(G, mark, root)
end

proc pos_traverse(in G, in/out mark: marks, in v: V)
     for w ∈ children(v) do pos_traverse(mark, w) od
     visit(mark,v)
end
```

En lo que sigue, asumimos que la función children devuelve una lista en vez de un conjunto. Definimos para $v, w \in V$ la siguiente relación: $v \preceq w$ sii existen $p, q, r \in V$ tales que p y q aparecen en children(r) en ese orden, p es ancestro de v, y q es ancestro de w. Esta relación es antisimétrica, es decir, $v \preceq w \wedge v \succeq w \Rightarrow v = w$.

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

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

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 vértices inicialmente desconocidos $(v_1, w_1), \ldots, (v_M, w_M)$ si v_i es ancestro de w_i de manera eficiente, conviene calcular los arreglos pre_mark y pos_mark. Como ya hemos observado, v_i será ancestro de w_i sii pre_mark.ord$[v_i] \preceq$ pre_mark.ord$[w_i]$ y pos_mark.ord$[v_i] \succeq$ pos_mark.ord$[w_i]$.

fun ancestor(pre_mark, pos_mark: marks, v, w: V) **ret** b: **bool**
  b:= (pre_mark.ord[v] $\leq$ pre_mark.ord[w] $\wedge$ pos_mark.ord[v] $\geq$ pos_mark.ord[w])
end

El costo del método es principalmente el de precondicionar, en este caso es el de recorrer dos veces el árbol finitario, que es lineal en el número de vértices. Una vez calculadas las dos tablas, el costo de atender cada consulta es una llamada a la función ancestor, que es constante.

Búsqueda en profundidad (DFS). Todas las estrategias vistas para recorrer árboles binarios, y las 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 vértice 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 vértice. Si el grafo tiene ciclos, nuestra recorrida en profundidad puede llevarnos a un vértice 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 vértice fue visitado. Redefinimos el procedimiento de inicialización y definimos una nueva función booleana que dice si un vértice ya fue visitado.

```
proc init(out mark: marks)
      mark.cont:= 0
      for v ∈ V do mark.ord[v]:= 0 od
end
fun visited(mark: marks, v: V) ret b: bool
    b:= (mark.ord[v] ≠ 0)
end
```

Con estas definiciones es posible implementar 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=(V,neighbours)) ret mark: marks
    init(mark)
    for v ∈ V do
        if ¬visited(mark,v) then dfsearch(G, mark, v) fi
    od
end
proc dfsearch(in G, in/out mark: marks, in v: V)
     visit(mark,v)
     for w ∈ neighbours(v) do
         if ¬visited(mark,w) then dfsearch(G, mark, w) fi
     od
end
```

El procedimiento recursivo dfsearch es muy similar al procedimiento pre_traverse visto para árboles salvo que en vez de children(v) usa neighbours(v) (un nombre más apropiado para denominar la lista de todos los vértices vecinos al vértice v 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 vértice, y con ello que la recorrida se pierda en un ciclo del grafo.

El procedimiento 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 ciclo **for**. Es decir, la inicia tantas veces como sea necesario a partir de diferentes vértices. 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 vértice si el mismo no ha sido ya visitado en una búsqueda iniciada en un vértice 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 $w \in \text{neighbours}(v)$ sii $v \in \text{neighbours}(w)$.

¿Cómo hacer una versión análoga de dfs que corresponda a pos_recorrer en vez de a pre_recorrer? Es un poco más difícil, hay que tener cuidado con los ciclos.

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

```
proc dfsearch(in G, in/out mark: marks, in v: V)
    var p: stack of V
    empty(p)
    visit(mark,v)
    push(v,p)
    while ¬is_empty(p) do
        if existe w ∈ neighbours(top(p)) tal que ¬visited(mark,w) then
          visit(mark,w)
          push(w,p)
        else pop(p)
        fi
    od
end
```

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

En el caso de un árbol finitario, una recorrida en BFS equivale a visitar primero la raíz, luego todos los vértices del nivel 1, luego todos los del nivel 2, etc.

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. El algoritmo principal bfs es idéntico a dfs, sólo invoca a bfsearch en vez de invocar a dfsearch.

```
fun bfs(G=(V,neighbours)) ret mark: marks
    init(mark)
    for v ∈ V do
        if ¬visited(mark,v) then bfsearch(G, mark, v) fi
    od
end
```

```
proc bfsearch(in G, in/out mark: marks, in v: V)
    var q: queue of V
    empty(q)
    visit(mark,v)
    enqueue(q,v)
    while ¬is_empty(q) do
        if existe w ∈ neighbours(first(q)) tal que ¬visited(w) then
            visit(mark,w)
            enqueue(q,w)
        else dequeue(q)
        fi
    od
end
```

Backtracking y DFS. Resolver un problema utilizando backtracking equivale a recorrer en DFS un grafo **implícito**, es decir, un grafo que no está dado por el problema que se intenta resolver, sino que está dado por los intentos que realiza el algoritmo, las elecciones que toma o descarta, en su afán de hallar la solución.

Por ejemplo, al considerar el problema de la moneda se propuso el siguiente algoritmo que utiliza backtracking, y cuya llamada principal es $m(n, k)$ donde n es el número de denominaciones y k el monto a pagar de manera exacta:

$$m(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ m(i-1,j) & d_i > j > 0 \wedge i > 0 \\ \text{mín}(m(i-1,j), 1 + m(i, j-d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

Para descubrir cuál es el grafo implícito, se puede intentar determinar cuáles son las decisiones que toma el algoritmo. Principalmente se ve en el último caso que se elige el menor de dos posibilidades: utilizar o no monedas de denominación d_i. Esto nos dice que el grafo implícito será un árbol binario. Los vértices son pares (i, j), a lo que podemos agregarle el número de monedas acumuladas. Como la llamada principal es $m(n, k)$ donde n es el número de denominaciones y k el monto a abonar de manera exacta, la raíz del árbol estará dada por la terna $(n, k, 0)$, ya que hay 0 monedas acumuladas al momento de la llamada inicial. Desde un vértice (i, j, x), si $i, j > 0$ y $d_i < j$ existe una única arista a al vértice $(i-1, j, x)$. En cambio si $j \leq d_i$ existen dos aristas: una a $(i-1, j, x)$ y otra a $(i, j-d_i, x+1)$.

Por ejemplo, si las denominaciones son $d_1 = 1$, $d_2 = 10$ y $d_3 = 25$, y el monto es $k = 30$, se obtiene el árbol de la Fig. 1. Para entenderlo, consideremos por ejemplo el vértice $(2, 20, 1)$. El 2 indica que pueden usarse monedas de denominación d_1 o d_2, el 20 es el saldo a pagar y el 1 el número de monedas ya utilizadas. Sus hijos corresponden a la posibilidad de no utilizar más monedas de denominación $d_2 = 10$ (primer hijo: $(1, 20, 1)$) o la posibilidad de utilizar más monedas de esa denominación (segundo hijo: $(2, 10, 2)$).

Las soluciones se encuentran en las cinco hojas, de las cuales sólo $(2, 0, 3)$ es la óptima, ya el tercer elemento de la terna es el número de monedas.

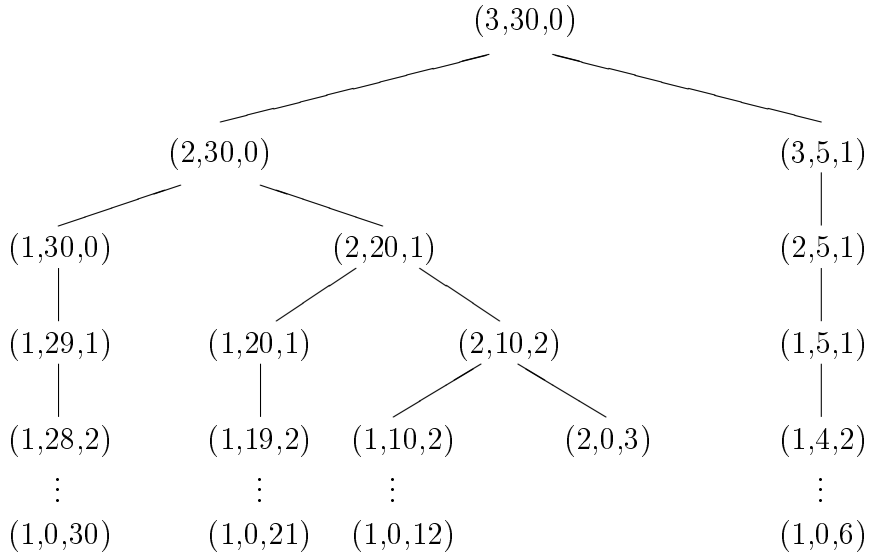

FIGURA 1. Grafo de búsqueda para la primera definición de m, para denominaciones $d_1 = 1$, $d_2 = 10$ y $d_3 = 25$ y monto $k = 30$.

Otra forma de definir $m(i, j)$ que se consideró es la siguiente:

$$m(i,j) = \begin{cases} 0 & j = 0 \\ 1 + \text{mín}(\{m(i', j - d_{i'}) | 1 \leq i' \leq i \wedge d_{i'} < j\}) & j > 0 \end{cases}$$

En este caso, la raíz resulta la misma que en el caso anterior, pero el grafo implícito es diferente ya que (i, j, x) puede tener varios hijos: todos los vértices de la forma $(i', j - d_{i'}, 1 + x)$ tal que $1 \leq i' \leq i$ y $d_{i'} < j$.

Con esta definición detallamos en la Fig. 2 el grafo implícito para las mismas denominaciones que el ejemplo anterior, pero con monto $k = 55$. Si bien el árbol resultante es más grande que el de la Fig. 1 no hay que engañarse: esto se debe a que se eligió un monto mayor. Si se quieren comparar los árboles obtenidos con cada una de las definiciones de m, debe compararse el árbol de la Fig. 1 con el subárbol de la Fig. 2 cuya raíz es el vértice $(3, 30, 1)$. Se observará que este último es ligeramente más pequeño.

Ocho reinas. Otro problema interesante para resolver usando la técnica del backtracking es el de calcular el número de maneras diferentes de ubicar ocho reinas (o damas) en un tablero de ajedrez (que tiene ocho filas por ocho columnas) 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 cuatro reinas en un tablero de cuatro filas por cuatro columnas. No es difícil encontrar manualmente la manera de colocar cuatro reinas en un tablero de cuatro por cuatro sin que ningún par de reinas compartan fila, columna ni diagonal.

Pero colocar ocho reinas en un tablero de ocho por ocho, de manera de que ningún par de reinas compartan fila, columna ni diagonal, es más difícil. Más aún, el problema puede generalizarse a colocar n reinas en un tablero de n por n sin que ningún par de reinas compartan fila, columna ni diagonal.

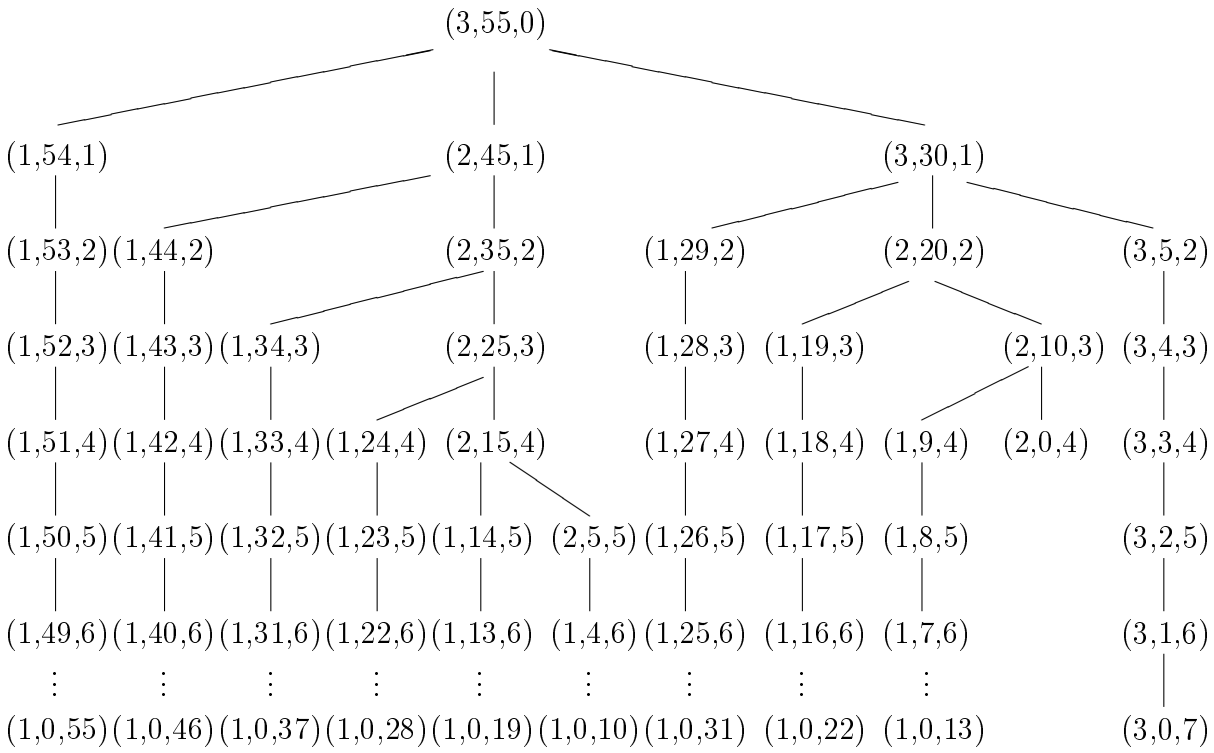

FIGURA 2. Grafo de búsqueda para la segunda definición de m, para denominaciones $d_1 = 1$, $d_2 = 10$ y $d_3 = 25$ y monto $k = 55$.

Primer intento. Lo primero que podríamos hacer es revisar todas las maneras posibles de ubicar ocho 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: nat
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    for i1:= 1 to 57 do
        for i2:= i1+1 to 58 do
            ...
                for i8:= i7+1 to 64 do
                    if solucion_1([i1,i2,i3,i4,i5,i6,i7,i8]) then r:= r+1 fi
                od
            ...
        od
    od
end
```

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

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

Para hacer explícito el grafo definimos el conjunto de vértices por

$$V = \{[p_1, p_2, \ldots, p_n] \in \{1, \ldots, 64\}^* \mid n \leq 8 \wedge p_1 < p_2 < \ldots < p_n \leq 56 + n\}$$

y decimos que dados $p = [p_1, p_2, \ldots, p_n] \in V$ y $q = [q_1, q_2, \ldots, q_m] \in V$ hay una arista de p a q sii $m = n + 1$ y $p_i = q_i$ para todo $1 \leq i \leq n$.

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 seis reinas.

A continuación presentamos un algoritmo mejor, que considera sólo las distintas maneras de colocar ocho 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: nat
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    for j1:= 1 to 8 do
        for j2:= 1 to 8 do
            ...
                for j8:= 1 to 8 do
                    if solucion_2([j1,j2,j3,j4,j5,j6,j7,j8]) then r:= r+1 fi
                od
            ...
        od
    od
end
```

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.

El conjunto de vértices del grafo implícito ahora es $V = \{p \in \{1, \ldots, 8\}^* \mid |p| \leq 8\}$ y las aristas se establecen de la misma manera que en el intento anterior.

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

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

donde el procedimiento or_2 se define recursivamente como sigue

```
proc or_2(in sol: list of nat, in/out r: nat)
                        {calcula el número de maneras de extender la solución parcial 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, digamos n, de reinas en un tablero de n filas por n columnas.

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: nat
                    {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 nat, in/out r: nat)
                        {calcula el número de maneras de extender la solución parcial 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 puede ser igual a solucion_2, con la salvedad de que no es necesario que solucion_3 verifique que las reinas estén en diferentes columnas ya que eso vale por la forma en que or_3 construye la solución.

El conjunto de vértices del grafo implícito excluye las listas con repeticiones, ahora tenemos $V = \{p \in \{1, \ldots, 8\}^* \mid |p| \leq 8 \wedge p \text{ sin repeticiones}\}$. Las aristas se establecen de la misma manera que en los intentos anteriores.

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: nat
                    {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    or_4([ ], [ ], [ ], r)
end
proc or_4(in sol, bajadas, subidas: list of nat, in/out r: nat)
                                          {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_4(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 ocho reinas 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+7 (sumamos siete solo para asegurar que el valor sea un número natural, otra alternativa es usar el tipo **int**) 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:nat) ret r: nat
    r:= i-j+7
end
fun subida(i,j:nat) ret r: nat
    r:= i+j
end
```

Ahora resulta más complicado explicitar el grafo implícito: V es el conjunto de listas $[p_1, \ldots, p_n] \in \{1, \ldots, 8\}^*$ tales que para todo $i \neq j$ las siguientes condiciones se cumplen:

1. $p_i \neq p_j$, es decir que no se repiten columnas,
2. $p_i - i \neq p_j - j$, es decir que no se repiten bajadas, y
3. $p_i + i \neq p_j + j$, es decir que no se repiten subidas.

Las aristas se establecen como en los intentos anteriores.