Capítulo 8

Backtracking

8.1. Introducción

En los temas precedentes se han analizado algunos métodos que permiten obtener la solución de un problema a partir de ciertas propiedades de la misma. Sin embargo, existen problemas que hasta ahora no se han podido resolver con tales técnicas, de forma tal que la única forma de resolverlos consiste en partir de un conjunto a *priori* de *posibles soluciones*, en el cual sabemos que están las soluciones del problema, y después analizar cuales realmente lo son. Esta forma de proceder es aplicable a problemas de localización de soluciones en los que se trata de encontrar una o todas las soluciones del problema en cuestión, aunque también se podría utilizar en problemas de optimización comparando simplemente las soluciones obtenidas y estimando cual de ellas es la óptima. Esta es la forma de proceder del método denominado Backtracking.

Por su forma de obtener la solución o soluciones de un problema, el Backtracking se puede considerar como un método de exploración del conjunto de posibles soluciones de un problema, aplicable cuando dichas soluciones son susceptibles de dividirse en etapas. De esta forma, el conjunto de posibles soluciones se puede representar en forma de árbol o de grafo (apartir de ahora usaremos el término árbol), en el que cada nodo representa una etapa k, que sería el último trozo de subsolución formado por las k primeras etapas. Por otra parte, las $k-1$ primeras etapas, correspondientes al nodo de la etapa k, están representadas por los nodos que van desde la raíz al nodo en cuestión. Adicionalmente, los hijos del nodo de la etapa k serán posibles prolongaciones al añadir una nueva etapa $k+1$. Para recorrer el conjunto de posibles soluciones bastaría con recorrer el árbol previamente creado. Básicamente, el backtracking se asemeja al recorrido en profundidad en un grafo dirigido.

La solución o soluciones obtenidas se suelen expresar mediante una n-tupla $(x_1, x_2, \ldots, x_n)$ donde los x_i son elementos seleccionados de algún conjunto finito

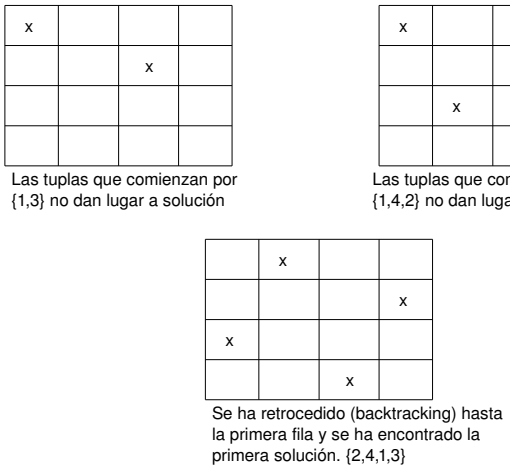

Figura 8.1: Búsqueda de soluciones en el problema de las 4-reinas.

S_i. En problemas de optimización se busca la tupla que optimiza una función criterio que satisface el predicado $P(x_0, x_1, \ldots, x_n)$. En otro tipo de problemas basta con encontrar la tupla o tuplas que satisfacen a P. Un ejemplo de este último caso será el problema de las n reinas. La solución sería expresable mediante una tupla, donde x_i indicaría la columna donde se colocaría la reina que está en la i-ésima fila. En este caso, la función criterio será aquella que comprueba que ninguna reina es amenazada, o apuntada, por las reinas ya colocadas.

8.2. El método general

En primer lugar, es necesario fijar la descomposición en etapas de la solución, lo que establecerá, una vez analizadas las opciones posibles en cada etapa, la estructura del árbol a analizar. Hay que resaltar que las opciones posibles de cada etapa dependerán de:

- La etapa en la que nos encontremos.
- Del trozo de solución construido hasta ese momento.

A la vez que se va construyendo el árbol, y llegado el caso, hay que identificar qué nodos corresponden a posibles soluciones, y cuales por el contrario son sólo etapas previas de dichas soluciones. De esta forma, una vez alcanzados los nodos que son posibles soluciones, se compruebe si realmente lo son. Por otra parte, puede ocurrir

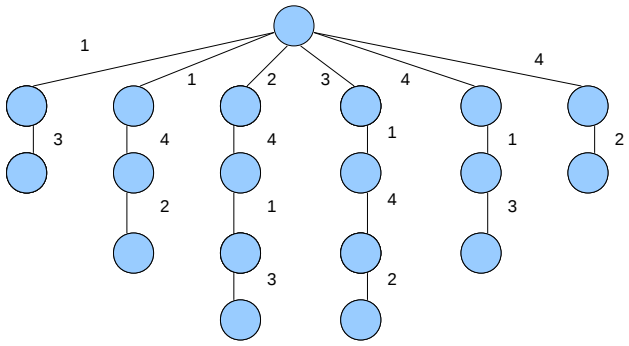

Figura 8.2: Árbol resultante en el problema de las 4-reinas

que al llegar a un cierto nodo del árbol se compruebe que ninguna continuación del mismo origine una solución del problema, en cuyo caso no se sigue buscando en los descendientes de ese nodo, aunuqe hay que seguir buscando en la prolongación de otros caminos alternativos. A los nodos de este tipo se les denomina nodos *fracaso*.

También es posible que un nodo no detectado en un principio como nodo fracaso en una primera exploración, se compruebe que todos sus descendientes son nodos fracaso y por tanto él también lo es. En este caso habrá que retroceder subiendo niveles en el árbol y buscando caminos alternativos en busca de posibles soluciones. De esta acción de retroceso proviene el nombre de Backtracking.

Para aclarar algunos conceptos vamos a analizar dos ejemplos cuya solución se verá más adelante:

- **Problema de las n-reinas**. Si particularizamos el problema de las n-reinas para $n = 4$, si excluimos todas aquellas posibles soluciones donde dos reinas ocupan la misma fila o columna, se puede representar cualquier solución como una tupla de 4 elementos (x_1, x_2, x_3, x_4), donde cada tupla es una permutación de los 4 primeros números naturales. El índice de cada elemento designará la fila que ocupa la reina y el valor designará la columna, eso implica que dos reinas jamás ocuparán la misma fila o la misma columna. El número total de tuplas posibles será de $4! = 24$. En la figura 8.1 aparece un esquema de funcionamiento para el problema hasta que se obtiene la primera solución y la figura 8.2 muestra el árbol generado en la búsqueda de las soluciones, donde en cada lado aparece la columna que ocuparía la reina ubicada

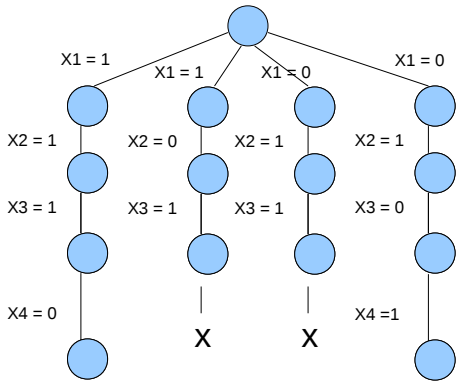

Figura 8.3: Árbol resultante en el problema de la suma de los subconjuntos

en la fila correspondiente al nivel de cada rama del árbol (la raíz tendría nivel 0 y sus hijos nivel 1). Solamente conforman soluciones aquellos caminos que tienen una longitud de 4 ramas, los demás caminos reflejan intentos de búsqueda de soluciones. Las tuplas solución serían $(2, 4, 1, 3)$ y $(3, 1, 4, 2)$.

- **Problema de la suma de subconjuntos**. En este caso se tiene un conjunto S de n números enteros positivos $S = \{a_1, a_2, \ldots, a_n\}$ y otro entero positivo M y se pretenden encontrar todos los subconjuntos de S cuya suma sea igual a M. Por ejemplo, supongamos que $S = \{5, 10, 15, 20\}$ y que $M = 30$. En este caso los subconjuntos solución serían $S_1 = \{5, 10, 15\}$ y $S_2 = \{10, 20\}$. En este caso tendríamos dos formas de representar las soluciones:

  1. Mediante tuplas de i elementos, donde $i \leq n$ y cada elemento de la tupla indica la posición que ocupa el índice del elemento del conjunto que forma parte de la solución. Para el subconjunto solución S_1 tendríamos la tupla $m_1 = (1, 2, 3)$ y para el subconjunto S_2 tendríamos la tupla $m_2 = (2, 4)$. En este caso las tuplas pueden tener distinta longitud.
  2. Mediante tuplas de n elementos en las cuales se indica mediante un 1, o un 0 en la posición i de la tupla, si el elemento x_i forma o no parte de la solución. Para el subconjunto solución S_1 tendríamos la tupla $m_1 = (1, 1, 1, 0)$ y para el subconjunto S_2 tendríamos la tupla $m_2 = (0, 1, 0, 1)$. En este caso las tuplas tendrían siempre la misma longitud. La figura 8.3 refleja parte del árbol formado en la búsqueda de las dos

soluciones.

Una vez vistos los conceptos básicos y su aclaración mediante los ejemplos anteriores, ahora se puede detallar el método general que sigue cualquier algoritmo basado en el Backtracking. Suponemos que se quieren encontrar todas las soluciones de un problema.

Sea $(x_1, x_2, \ldots, x_i)$ un camino desde el nodo raíz hasta un nodo cualquiera del árbol y $C(x_1, x_2, \ldots, x_i)$ el conjunto de posibles candidatos x_{i+1} que se añadan a $(x_1, x_2, \ldots, x_i)$ para formar parte de una posible solución. Sean P_{i+1} el conjunto de predicados tales que $P_{i+1}(x_1, x_2, \ldots, x_{i+1})$ es cierto si para el camino $(x_1, x_2, \ldots, x_{i+1})$ se puede alcanzar la solución. De esta forma los candidatos x_{i+1} para la etapa $i + 1$ son aquellos valores generados por la función C, que satisfacen P_{i+1}. Teniendo en cuenta estas particularidades, se podría formular un algoritmo general, en el cual las soluciones generadas serán almacenadas en el vector X y se mostrarán justo al ser obtenidas. Observese como el vector X almacena en cada moment los candidatos que fforman parte de la rama del árbol que se está explorando, y que pueden ser parte de una posible solución. De esta forma, el árbol como tal, no existe, solo se almacena en cada momento una rama del mismo. La función C nos devuelve los posibles valores de $X(k)$ cuando $X(1), X(2), \ldots, X(k-1)$ ya han sido seleccionados y los predicados P_k evalúan los $X(k)$ que cumplen las restricciones de la solución.

Algoritmo $Backtracking(n;;X)$
  inicio
    $k \leftarrow 1$
    mientras $k > 0$ **hacer**
      si $\exists X(k) \in C(X(1), \ldots, X(k-1))$ **y** $P_k(X(1), \ldots, X(k)) = true$ **entonces**
        $X(k) \leftarrow C(X(1), X(2), \ldots, X(k-1))$
        si $X(1), \ldots, X(k)$ es $solucion$ **entonces**
          escribir $X(1), \ldots, X(k)$
        sino
          $k \leftarrow k + 1$ $pasa$ al $siguiente$ $nodo$
        finsi
      sino
        $k \leftarrow k - 1$ se $retrocede$ al $nodo$ $anterior$
      finsi
    finmientras
  fin

En la etapa inicial, para $k = 1$, hay que resaltar que la función C producirá todos los candidatos posibles que se puedan usar como primer elemento $X(1)$ de la tupla solución. En este caso $X(1)$ tomará aquellos valores para los que $P_1(X(1))$ sea cierto. Por otra parte, también se puede apreciar que el árbol se va construyendo mediante un recorrido en profundidad ya que k se va incrementando y el vector solución va creciendo hasta que una solución es encontrada, o no se produce ningún valor válido para $X(k)$. Por esta razón, cuando k es decrementado se generan

nuevos $X(k)$ que antes no habían sido probados. Ello implica que los $X(k)$ hay que generarlos siguiendo un cierto orden.

A continuación se detalla la versión recursiva del algoritmo anterior. El funcionamiento es similar al del recorrido prefijo de un árbol multicamino, ya que antes de seguir explorando posibles soluciones con las llamadas recursivas, se comprueba si se ha encontrado una solución. Esta versión recursiva se invoca la primera vez con $BacktrackingRecursivo(n, 1; ; X)$. Se supone que los $k-1$ primeros valores del vector X ya han sido asignados.

Algoritmo $BacktrackingRecursivo(n, k; ; X)$
  inicio
    mientras $\exists X(k) \in C(X(1), \ldots, X(k-1))$ **y** $P_k(X(1), \ldots, X(k)) = true$ **hacer**
      $X(k) \leftarrow C(X(1), X(2), \ldots, X(k-1))$
      si $X(1), \ldots, X(k)$ es $solucion$ **entonces**
        escribir $X(1), \ldots, X(k)$
      finsi
      $BacktrackingRecursivo(n, k+1; ; X)$
    finmientras
  fin

Todos los posibles candidatos en la etapa k de la tupla que satisfacen P_k son generados uno a uno y son añadidos al actual $(X(1), \ldots, X(k-1))$. Cada vez que se añade un $X(k)$ se comprueba si se ha encontrado una solución y después se invoca el algoritmo recursivamente. Cuando el esquema **mientras** termina, no existen más valores de $X(k)$ y termina la llamada actual. La última llamada no resuelta se reanuda y continúa examinando los elementos restantes, suponiendo que sólo $k-1$ valores se han determinado.

Resaltar que cuando k excede a n, $C(X(1), X(2), \ldots, X(k-1))$ devuelve un valor vacío y entonces no entra en el esquema **mientras**. Al igual que la versión no recursiva, esta versión proporciona todas las soluciones.

En este método, existen dos aspectos muy importantes, que a la postre determinarán la eficiencia del mismo en aquellos problemas en los que se pueda aplicar:

- El conjunto de partida, con las posibles soluciones del problema, ha de ser del menor tamaño posible lo cual redundará en la eficiencia del método. Por ejemplo en el problema de las 8 reinas, el conjunto de partida podría contener desde $C_{64}^{8} = 4{,}426{,}165{,}368$ posibles soluciones, en el caso más desfavorable (cuando no hay restricciones a la hora de colocar las reinas a la hora de generar la solución), hasta $8! = 40,320$ en el caso más favorable, que se produce cuando se generan las soluciones de forma tal que dos reinas no coincidan en la misma fila. Evidentemente, no es lo mismo explorar en un conjunto de $4{,}426{,}165{,}368$ posibles soluciones, que en un conjunto de $40,320$.

- La complejidad de la prueba o condición que se utilizará para detectar nodos fracaso. Lógicamente estas pruebas pretenden ahorrar tiempo ya que reducirán el trozo de árbol a explorar, aunque hay que tener en cuenta que estas pruebas consumen tiempo, por lo que no serán interesantes cuando detecten pocos nodos fracaso. Como regla general, son deseables pruebas o condiciones sencillas, mientras que las pruebas complejas solo serán deseables cuando se trate de evitar árboles con tamaños gigantescos (conjunto enorme de posibles soluciones).

8.3. Ejemplos

8.3.1. El problema de las 8 reinas

Este problema consiste en situar 8 reinas en un tablero de ajedrez, de forma que ninguna de ellas amenace a las demás. Se podría generalizar para un tablero de nxn, situando en él n reinas. Si usamos indexación desde 0 como en C++, la solución del problema la podemos representar como una tupla $(x_0, x_1, x_2, x_3, \ldots, x_{n-1})$ en la que x_j es la columna de la fila j donde la reina j-ésima es ubicada. Como el índice de cada x_j refleja la fila en la que está ubicada la reina j-ésima, ocupan filas distintas. Para evitar que las reinas se ubiquen en la misma columna (se amenazarían en vertical), las x_j son todas distintas. También habría que establecer la condición para que dos reinas no estén en la misma diagonal. Si los escaques del tablero se etiquetaran como si el tablero fuese una matriz bidimensional, se observa que en las diagonales que crecen de izquierda a derecha, la diferencia entre filas y columnas de cada escaque permanece constante; mientras que en las diagonales que crecen de derecha a izquierda, lo que permanece constante es la suma de la fila y la columna de cada escaque.

Suponiendo que dos reinas se colocan en escaques (i, j) y (k, l), ocupan la misma diagonal si: $i - j = k - l$ ó $i + j = k + l$. De estas dos condiciones se puede deducir que: $j-l = i-k$ y $j-l = k-i$ de donde se desprende que: $|j-l| = |k-i|$. En conclusión, el valor absoluto de la diferencia entre filas y el valor absoluto de la diferencia entre columnas permanece constante.

A continuación figura el algoritmo $Lugar$ que devuelve un valor cierto si la reina k-ésima puede ser ubicada en la fila k y columna x_k, una vez colocadas las $k-1$ anteriores. Para ello se prueba que x_k sea distinto de $x_0, x_1, x_2, \ldots, x_{k-1}$ (no se amenazan en vertical) y que además ninguna de las reinas anteriores la amenaza en diagonal.

Algoritmo $Lugar(k, x; ;)$
  inicio
    para i **de** 0 **a** $k-1$ **hacer**
      si $(x(i) = x(k)$ **o** $|x(i) - x(k)| = |i - k|$ **entonces**

```
            devolver falso
          finsi
        finpara
        devolver cierto
    fin
```

Este algoritmo será invocado en el algoritmo final para verificar si una reina es amenazada por las anteriores. Se supone que indexamos desde 0 hasta $n-1$. El algoritmo final se basa en el siguiente esquema:

```
Algoritmo n − reinas(n;;)
    inicio
      colocar primera  reina en columna -1 (fila 0)
      mientras no se tengan todas las soluciones hacer
        desplazar reina a la siguiente columna
        mientras no salga del tablero y sea amenazada por una anterior
hacer
          desplazar reina a la siguiente columna
        finmientras
        si una posicion correcta ha sido encontrada entonces
          si es la ultima reina, se tiene una solucion entonces
            escribir la solucion
          sino No es la ultima reina y hay que probar la siguiente
            pasar a probar la siguiente reina ubicandola en columna
-1
          finsi
        sino la reina no se puede ubicar
          volvemos a reina anterior
        finsi
      finmientras
    fin
```

El algoritmo en pseudocódigo, indexando el vector de soluciones desde 0 a $n-1$ sería el siguiente:

```
Algoritmo n − reinas(n)
    inicio
      x(0) ← −1
      k ← 0
      mientras k > −1 hacer
        x(k) ← x(k) + 1
        mientras x(k) < n y Lugar(k, x) = falso hacer
          x(k) ← x(k) + 1
        finmientras
        si x(k) < n entonces
         No se ha salido del tablero
          si k = n − 1 entonces
```

```
        Ha encontrado una solución
          escribir x(0), x(1), x(2), ..., x(k)
        sino
        Pasa a la siguiente fila para colocar la siguiente reina
          k ← k + 1
          x(k) ← −1
        finsi
      sino
      Retrocede a la fila anterior, para probar otra posición de esa reina
        k ← k − 1
      finsi
    finmientras
  fin
```

Para ver la efectividad del método se puede destacar que existen $C_{64}^{8} = 4,426,165,368$ formas posibles de ubicar 8 reinas en un tablero y que con este algoritmo solo se ensayarían $8! = 40,320$ tuplas. Finalmente, indicar que este problema fue planteado a Gauss en su época, y consiguió obtener 68 soluciones del problema. Años más tarde un matemático, que era ciego, obtuvo las 92 soluciones que tiene el problema para un tablero de $8x8$.

8.3.2. El problema de la suma de subconjuntos.

Este problema ya se ha citado en la introducción del tema. Su enunciado es el siguiente: se tienen n enteros positivos, de valores $V(i), (i = 1, .., n)$, y un entero positivo M, y se trata de obtener los subconjuntos del conjunto de enteros que suman M.

En este apartado se va a proponer una solución usando tuplas de tamaño fijo, es decir con un número de elementos similar al del tamaño del conjunto de enteros, donde cada elemento será 0 o 1 en función de que forme parte o no de la solución. El árbol que genera las soluciones se irá construyendo de una forma muy simple, para un nodo $X(i)$ cualquiera de nivel i, el hijo izquierdo se corresponderá con $X(i) = 1$ y el derecho con $X(i) = 0$.

Una elección simple para el conjunto de predicados P_k, que en este caso se reducirían a uno solo, sería que $P_k(X(1), X(2), \ldots, X(k)) = true$ sí y solo sí:

$$\sum_{i=1}^{k} X(i) * V(i) + \sum_{i=k+1}^{n} V(i) \geq M$$

Es decir, que la suma de los candidatos ya seleccionados más los que quedan por seleccionar ha de ser como mínimo M. Está claro que no se puede obtener una solución si este predicado no se cumple, ya que su suma no llegaría a valer M aun-

que se usaran todos los candidatos que quedan por evaluar (elementos del segundo sumatorio).

Este predicado puede ser fortalecido si asumimos que el conjunto de números está en orden no decreciente (orden creciente). En este caso $X(1), \ldots, X(k)$ no pueden generar una solución si:

$$\sum_{i=1}^{k} X(i) * V(i) + V(k+1) > M$$

Es decir, si vamos seleccionando los candidatos en orden creciente, y al sumar a los valores de los candidatos ya seleccionados el valor del más pequeño de los candidatos que queda por seleccionar ($V(k+1)$), se supera el valor de M. Es evidente que, en este caso, cualquier otro candidato que se seleccione hará que la suma también supere a M, ya que dichos candidatos serán mayores que el candidato $k+1$.

Por tanto, el predicado quedaría como:

$$P_k(X(1), \ldots, X(k)) = true \; si \; y \; solo \; si \sum_{i=1}^{k} X(i) * V(i) + \sum_{i=k+1}^{n} V(i) \geq M \; y$$

$$\sum_{i=1}^{k} X(i) * V(i) + V(k+1) \leq M$$

En el algoritmo que se va a plantear no se va a hacer uso de P_k. En este caso se va a adaptar el esquema general del backtracking al problema en cuestión para obtener un algoritmo más simple. A continuación se detalla el algoritmo recursivo que resuelve el problema. Como se puede comprobar, el algoritmo es similar al recorrido prefijo de un árbol binario, comprobando, antes de bajar al hijo izquierdo o derecho, si la solución es viable o no, de esta forma el árbol se poda antes de seguir una rama que no conduce a una solución. En el algoritmo se asume lo siguiente:

- Los valores de $X(j), 1 \leq j < k$ ya han sido determinados.
- $s = \sum_{j=1}^{k-1} X(j) * V(j)$. s es la suma de los valores seleccionados entre los $k-1$ primeros.
- $r = \sum_{j=k}^{n} V(j)$. r es la suma de los valores desde k hasta n.
- Los $V(j)$ están en orden no decreciente.
- $V(1) \leq M$ y $\sum_{i=1}^{n} V(i) \geq M$. Condición para que haya posibles soluciones. Si de primeras no se cumplen estas condiciones no puede haber solución.

Algoritmo $SumaSubconjuntos(s, k, r, n, M, V; X;)$
inicio
Genera hijo izquierdo. $s+V(k) \leq M$ *ya que* $P_{k-1}(X(1), \ldots, X(k-1)) = true$
$X(k) \leftarrow 1$
si $s + V(k) = M$ **entonces** `Subconjunto encontrado`
escribir $(X(1), \ldots, X(k)$
finsi
`Para poder recorrer el hijo izquierdo, comprueba si el` $k+1$ `es viable`
si $s + V(k) + V(k+1) \leq M$ **entonces** $P_k(X(1), \ldots, X(k)) = true$
$SumaSubconjuntos(s + V(k), k + 1, r - V(k), n, M, V; X;)$
finsi
`Para poder recorrer el hijo derecho, comprueba si el` $k+1$ `es viable`
si $s+r-V(K) \geq M$ **y** $s+V(k+1) \leq M$ **entonces** $P_k(X(1), \ldots, X(k)) = true$
`La primera parte comprueba si al eliminar` $V(k)$ `del resto, se sobrepasa o iguala` M
`La segunda parte comprueba que al añadir` $V(k+1)$ `no se sobrepasa` M
$X(k) \leftarrow 0$
`Se sigue buscando solucion sin incluir al` $V(k)$
$SumaSubconjuntos(s, k + 1, r - V(k), n, M, V; X;)$
finsi
fin

En este algoritmo conviene aclarar las siguientes cuestiones:

- El uso de las variables s y r evita calcular los sumatorios $\sum_{j=1}^{k-1} X(j) * V(j)$ y $\sum_{j=k+1}^{n} V(j)$ continuamente.
- La llamada inicial será $SumaSubconjuntos(0, 1, \sum_{j=1}^{n} V(j), n, M, V; ; X)$
- No se necesita comprobar la finalización de la recursión con $k > n$, ya que en la entrada del algoritmo $s \neq M$ y $s + r \geq M$. Por lo tanto $r \neq 0$ y así k no puede ser mayor que n.
- Cuando se evalúa el predicado $(s + r - V(k) \geq M$ **y** $s + V(k + 1) \leq M)$ no se sale del tamaño de V, ya que $s + r - V(k) \geq M$, se deduce que al menos ha de existir un elemento detrás del $k - simo$.
- Cuando se encuentra un subconjunto solución, entonces $X(k + 1), \ldots, X(n)$ deben ser cero, y estos ceros se omiten al escribir la solución.

Para aclarar el funcionamiento del algoritmo vamos a suponer un ejemplo en el cual $n = 6$, $M = 30$ y $V = \{5, 10, 12, 13, 15, 18\}$. En la figura 8.4 aparece parte del árbol que se genera en la búsqueda de soluciones. En cada nodo se representan los valores de s, k y r respectivamente en cada llamada recursiva. Los nodos circulares representan nodos solución. En la figura solo aparecen 23 nodos, pero el total sería $2^6 - 1 = 63$.

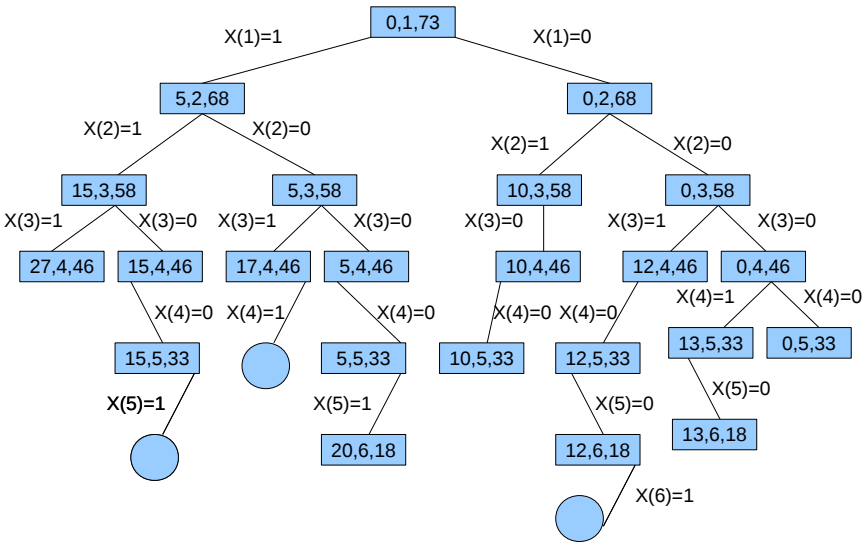

Figura 8.4: Árbol resultante para el ejemplo de la suma de los subconjuntos

Las soluciones obtenidas serían $(1, 1, 0, 0, 1)$, $(1, 0, 1, 1)$ y $(0, 0, 1, 0, 0, 1)$.

8.3.3. Ciclos hamiltonianos.

En un grafo conexo (dirigido o no dirigido) de n es posible que existan caminos que recorran todos los nodos, pasando por cada nodo una sola vez, o ciclos que recorran todos los nodos pasando por un nodo una sola vez, excepto en el caso del primer nodo del ciclo por el cual se pasa dos veces, al principio y al final. A estos caminos o ciclos se les denomina Hamiltonianos en honor a William Hamilton.

Cuando se aborda la solución del problema de obtener todos los ciclos hamiltonianos mediante Backtracking, el vector solución $(X(1), \ldots, X(n))$, se define de forma tal que el i-ésimo elemento nodo visitado queda representado por $X(i)$.

La clave de la solución está en determinar el conjunto de candidatos para $X(k)$ una vez que se han determinado los $k - 1$ primeros $(X(1), X(2), \ldots, X(k - 1))$.

Si $k = 1$, entonces $X(1)$ puede ser cualquiera de los n nodos. Para evitar que el mismo ciclo se calcule n veces (mismo ciclo, pero cambiando el nodo inicial) se va a considerar que $X(1) = 1$. Si $1 < k < n$, entonces $X(k)$ puede ser cualquiera de los nodos no seleccionados previamente y que además el nodo $X(k-1)$ esté conectado a $X(k)$. Finalmente $X(n)$ solo puede ser el último nodo restante, de forma tal que el nodo $X(n-1)$ esté conectado al $X(n)$ y $X(n)$ esté conectado a $X(1)$.

A continuación se detalla un algoritmo denominado $SiguienteNodo$ que determina el posible siguiente nodo para el ciclo que se está calculando. En este algoritmo, k representa al nodo k-ésimo del ciclo, C sería la matriz de conexión lógica (no necesitamos pesos) del grafo y X sería el vector que va almacenando la solución. Se supone que los $k-1$ primeros elementos del vector ya han sido calculados. Si $X(k) = 0$ indica que aún no se le ha asignado ningún nodo a $X(k)$. Si no se puede encontrar un nodo que cumpla la condición, $X(k)$ seguirá siendo 0. Cuando $k = n$, se comprueba además que $X(n)$ está conectado a $X(1)$.

Algoritmo $SiguienteNodo(k, n, C; X;)$
  inicio
    iterar
      $X(k) \leftarrow (X(k)+1)\mathbf{mod}(n+1)$ *recorre de forma cíclica los nodos del grafo*
      salir si $X(k) = 0$ *Ha recorrido todos los nodos sin encontrar ningún candidato*
      si $C(X(k-1), X(k)) = cierto$ **entonces** *hay lado que conecta el anterior con el evaluado*
        $j \leftarrow 1$
        mientras $X(j) \neq X(k)$ **hacer** *Comprobamos si coincide con alguno anterior*
          $j \leftarrow j + 1$
        finmientras
        si $j = k$ **entonces** *No coincide con ninguno de los anteriores. Puede ser candidato*
          Se comprueba también si es el último, para comprobar que está conectado al primero
          salir si $k < n$ **o** $(k = n$ **y** $C(X(n), 1) = cierto)$
        finsi
      finsi
    finiterar
  fin

A continuación se refleja el algoritmo recursivo $CicloHamiltoniano$ que hace uso del algortimo anterior para obtener todos los ciclos. Indicar que todos los ciclos comenzarán por el nodo 1, y que al principio el vector X está inicializado a 0, excepto $X(1) = 1$. La primera llamada se realizará con $k = 2$.

```
Algoritmo CicloHamiltoniano(k, n, C; X;)
  inicio
    iterar
      SiguienteNodo(k, n, C; X;)
      salir si X(k) = 0  No ha encontrado candidatos
      si k = n  entonces  ha encontrado un ciclo y lo escribe
        escribir  X(1), X(2), ..., X(n), X(1)
      sino  Se busca el siguiente nodo del ciclo
        CicloHamiltoniano(k + 1, n, C; X;)
      finsi
    finiterar
  fin
```

Dado que el número de posibilidades a explorar es $(n-1)!$, este algoritmo sería $O(n!)$.

El algoritmo se podría adaptar para resolver el problema del viajante de comercio guardando el ciclo de coste mínimo de entre todos los ciclos calculados. En este caso se podría usar también un algoritmo de poda, de forma tal que se guardase el ciclo de menor coste encontrado hasta el momento, y en caso de que un camino en exploración llegase a tener un coste peor que el mejor encontrado hasta ese momento, se interrumpiría la exploración de ese camino. En este caso, el algoritmo de poda solo podría reducir el tiempo de ejecución en los casos en los que la poda fuese efectiva pero en ningún caso podría reducir la complejidad computacional del algoritmo.

8.3.4. El problema de la mochila (versión 3).

En este apartado vamos a analizar el problema de la mochila en la versión analizada en el tema de *Programación dinámica*, pero afrontándolo mediante Backtracking. Recordamos que los datos del problema son los siguientes:

- El volumen de la mochila V.
- Los n materiales $m_1, m_2, ..., m_n$
- Sus volúmenes $v_1, v_2, ..., v_n$
- Sus precios unitarios $p_1, p_2, ..., p_n$ por unidad de volumen.

Hay que ver cual sería la combinación de materiales a escoger de forma tal que el valor de la misma sea máximo. En este caso los materiales son indivisibles, ya que un material se selecciona por completo o no se slecciona. La solución se va a almacenar en un vector $X(n)$ de ceros y unos, donde cada elemento i del vector

indica si se ha seleccionado el material m_i o no se ha seleccionado. El árbol de búsqueda de soluciones nos daría 2^n posibles soluciones, con lo cual sería similar al de la suma de los subconjuntos. Se podrían utilizar dos posibles organizaciones para el árbol, en función de que se seleccionase una tupla de tamaño n fijo, o una tupla de tamaño variable. Debido al tamaño que puede alcanzar el árbol es necesario establecer una condición que elimine posibles nodos fracaso sin llegar a expandirlos. Una buena condición consiste buscar un límite superior a la mejor solución que se pueda obtener expandiendo el nodo en cuestión y sus descendientes. En el caso de que este límite superior no supere al valor de la mejor solución determinada hasta ese momento, ese nodo se puede eliminar ya que en ningún caso obtendrá una solución mejor que la obtenida hasta ese momento.

Para implementar el algoritmo se van a usar tuplas de tamaño n fijo. Si en un nodo N los valores $X(i), 1 \leq i \leq k$ ya han sido determinados, entonces un límite superior para N puede obtenerse mediante la relajación del requerimiento de que $X(i)$ es 0 o 1, considerando que está entre 0 y 1 para los valores de i comprendidos entre $k+1$ y n, como en la versión voraz del algoritmo, y usando dicha versión resolver el problema. Esta relajación implica que los materiales se pueden seleccionar parcialmente, lo cual hace que las soluciones obtenidas sean aún mejores y se puedan usar como límite superior en los caminos que quedan por explorar.

El procedimiento $Limite$ calculará un límite superior para la mejor solución obtenible expandiendo cualquier nodo N en el nivel $k+1$ del árbol. Los materiales **están ordenados en orden descendiente de precios**, al igual que en el problema de la mochila en su versión voraz, y el valor final de la mochila será $p = \sum_{i=1}^{n} p_i * v_i * X(i)$. En los algoritmos $Limite$ y $Mochila$, $pActual$ y $vActual$ son el valor y volumen de la mochila en el estado que se está nalizando, y k es la posición del último nodo considerado. El algoritmo $Limite$ devuelve el valor nuevo de la mochila usando el algoritmo voraz a partir del estado que se analiza. Este valor siempre será un límite superior al resultado que se obtendría mediante el algoritmo que se está implementando.

```
Algoritmo Limite(n, pActual, vActual, k, p, v, V; ; )
  inicio
    valor ← pActual
    volumen ← vActual
     para i de k+1 a n hacer
       Se irán introduciendo los materiales restantes más caros
hasta llegar al volumen de la mochila
       volumen ← volumen + v(i)
        si volumen < V entonces Se puede introducir el material i
completo
          valor ← valor + p(i) * v(i)
        sino Se puede introducir el material i parcialmente
```

```
            devolver valor + (V − (volumen − v(i))) ∗ p(i)
        finsi
    finpara
    devolver valor
fin
```

Del algoritmo anterior se deduce que el límite para un posible hijo izquierdo de un nodo es el mismo que para ese nodo. Por lo tanto, la función de límite no tiene por qué ser utilizada siempre que el algoritmo de backtracking se mueve al hijo izquierdo de un nodo. Dado que el algoritmo tratará de hacer un movimiento al hijo de la izquierda siempre que pueda escoger entre un izquierdo y derecho, vemos que la función límite debe ser utilizada sólo después de una serie de movimientos con éxito hacia la izquierda (movimientos factibles a hijo izquierdo). A continuación se detalla un algoritmo iterativo que resuelve el problema.

```
Algoritmo Mochila(n, p(n), v(n), k, V, pActual, vActual; Y(n); Solucion, precioFinal)
  inicio
    Genera hijo izquierdo.
    Y(k) ← 1
    Actualizamos pActual y vActual
    vActual ← vActual + v(k)
    pActual ← pActual + p(k) ∗ v(k)
    Comprobamos si cabe en la mochila
    si V ≥ vActual entonces
       Calculamos el límite
       limite ← Limite(n, pActual, vActual, k, p, v, V)
       si limite > precioFinal entonces La solución puede mejorar
          si pActual > precioFinal entonces La solución se actualiza
             precioFinal ← pActual
             Solucion ← Y
          finsi
          si k < n entonces Se puede bajar un nivel
             Mochila(n, p(n), v(n), k+1, V, pActual, vActual; Y(n); Solucion, precioFinal)
          finsi
       finsi
    finsi
    Genera hijo derecho.
    Y(k) ← 0
    Actualizamos pActual y vActual quitando el material k
    vActual ← vActual − v(k)
    pActual ← pActual − p(k) ∗ v(k)
    Comprobamos si cabe en la mochila
    si V ≥ vActual entonces
       Calculamos el límite
```

$limite \leftarrow Limite(n, pActual, vActual, k, p, v, V)$
si $limite > precioFinal$ **entonces** `La solución puede mejorar`
si $pActual > precioFinal$ **entonces** `La solución se actualiza`
$precioFinal \leftarrow pActual$
$Solucion \leftarrow Y$
finsi
si $k < n$ **entonces** `Se puede bajar un nivel`
$Mochila(n, p(n), v(n), k{+}1, V, pActual, vActual; Y(n); Solucion, precioFinal)$
finsi
finsi
finsi
fin

Para ver mejor el funcionamiento del algoritmo se va a detallar un ejemplo práctico en el cual se tiene una mochila de volumen $V = 100$ y 5 materiales de volúmenes $v_i = \{20, 30, 40, 50, 60\}$ y de precios unitarios $p_i = \{5, 4, 3, 2'5, 1\}$.

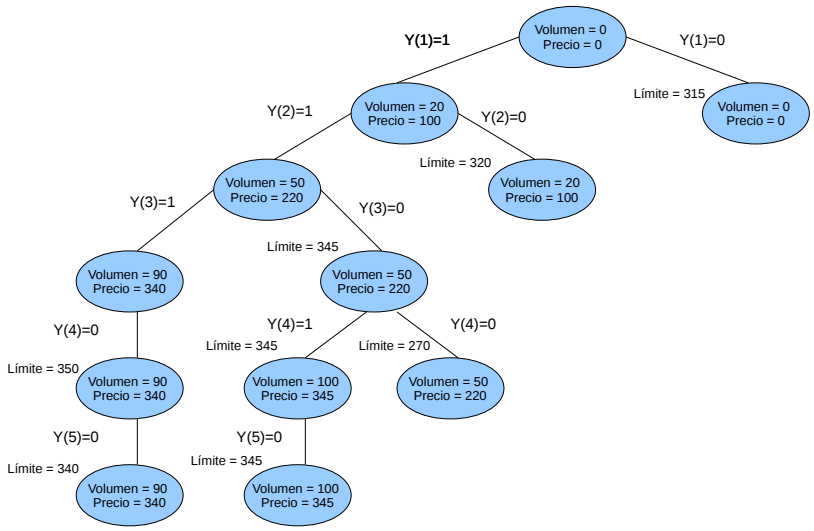

Figura 8.5: Árbol resultante para el ejemplo de la mochila resuelto por Backtracking

La figura 8.5 ilustra los pasos del algoritmo para este ejemplo. Dichos pasos son los siguientes:

- Inicialmente el volumen actual, el precio actual y el precio final son 0. Todos los valores de Y son -1 y k vale 1.

- Primero se recorren los hijos izquierdos y aplicando al algoritmo voraz el límite será $20x5 + 30x4 + 40x3 + 10x2{,}2 = 365$. Este límite será el mismo para toda la rama que se recorre por los hijos izquierdos

- Para $k = 1$ se crea el primer nodo, $X(1) = 1$, con $precio = 5 * 20 = 100$ y $volumen = 20$.

- Para $k = 2$ se crea el segundo nodo, $X(2) = 1$, con $precio = 5*20+30*4 = 220$ y $volumen = 20 + 30 = 50$.

- Para $k = 3$ se crea el tercer nodo, $X(3) = 1$, con $precio = 5 * 20 + 30 * 4 + 40 * 3 = 340$ y $volumen = 20 + 30 + 40 = 90$.

- Para $k = 4$ no genera hijo izquierdo ya que el cuarto material ya no cabe en la mochila, $X(4) = 0$.

- Se pasa a recorrer el hijo derecho $X(4) = 0$, y $limite(90, 340, 4, 100) = 340 + v_5 * p_5 = 340 + 10 * 1 = 350$ que es mayor que el precioFinal actual 340. Por lo tanto se explora esa rama pero no entra ningún material por la limitación del volumen.

- Ahora retrocede hasta llegar al segundo nivel, y comienza a explorar el hijo derecho $Y(3) = 0$. Al calcular el límite resulta un valor de $limite = 345$. Como es mayor que $pFinal = 340$, esa rama hay que explorarla porque la solución puede mejorar.

- Como se aprecia en la figura, al explorar esa rama, la solución mejora cuando $Y(4) = 1$ e $Y(5) = 0$. Por tanto, se actualiza el precio final a $precioFinal = 345$. En la rama generada por $Y(3) = 0$, el camino derecho presenta un límite de 270, que al ser inferior al precio final, no mejorará la solución actual y por tanto, no se explora.

- Al retroceder al primer nivel, y explorar el hijo derecho $Y(2) = 0$, al calcular el límite resulta un valor de $limite = 320$, que es inferior al precio final. Por tanto, se poda esa rama y no se explora.

- Lo mismo ocurre al retroceder al primer nivel y explorar el hijo derecho $Y(1) = 0$, ya que su límite es 315.

- La solucion final será $precioFinal = 345$, $volumenFinal = 100$ y $Y(1, 1, 0, 1, 0)$

8.4. Cuestiones sobre el tema

1. Describe el método general del bactracking.

2. ¿Cuales son los dos aspectos más importantes que influyen en la eficiencia de un algoritmo que se resuelva usando el método del Backtracking?.

3. Describe el algoritmo de las n-reinas usando un tablero de 4x4.

4. ¿Cómo modificarías el algoritmo de las n reinas si en vez de colocar reinas colocasemos torres (las torres mueven en horizontal y vertical)?. ¿Cuantas soluciones tendría?

5. ¿Cómo modificarías el algoritmo de las n-reinas si la reina también tuviese el movimiento del caballo?. En ese caso, el número de soluciones ¿sería mayor o menor que el número de soluciones del problema original?. Justifica esta última respuesta.

6. ¿Cual es el orden de complejidad del algoritmo de las n reinas?. Justifica tu respuesta.

7. Deducir la condición para que las reinas no se apunten en diagonal.

8. ¿Cuántas posibles soluciones explora el algoritmo de las n reinas para un tablero de 6x6?.

9. Describe brevemente como funciona el algoritmo que resuelve el problema de la suma de subconjuntos usando Backtracking.

10. En el problema de la suma de los subconjuntos, ¿qué condiciones se han de dar inicialmente para que el problema pueda tener solución?. Nota: Si estas condiciones no se diesen no habrá que explorar nada.

11. En el problema de la suma de los subconjuntos, ¿qué condiciones se han de cumplir para que a los k primeros candidatos se les pueda añadir al candidato k+1 para formar parte de la solución?.

12. ¿Cual es el orden de complejidad del algoritmo de la suma de subconjuntos?. Justifica tu respuesta.

13. ¿Se puede reducir el orden de complejidad del algoritmo de la suma de los subconjuntos si mejoramos la condición de poda?. Justifica tu respuesta.

14. Describe brevemente como funciona el algoritmo que resuelve el problema de los ciclos hamiltonianos usando Backtracking.

15. En el problema de los ciclos hamiltonianos para un grafo de n nodos. Si sabemos que existen soluciones, y consideramos soluciones distintas si hay una alteración en el orden de visita de los nodos ¿cual sería el número mínimo de soluciones que se pueden obtener?.

16. ¿Cómo adaptarías el algoritmo para obtener los ciclos hamiltonianos para resolver el problema del viajante de comercio?.

17. Indica alguna idea que sirva para reducir el número de caminos a explorar en la solución del problema del viajante de comercio basada en la obtención de los ciclos hamiltonianos.

18. ¿Se puede reducir el orden de complejidad del algoritmo del viajante de comercio usando ciclos hamiltonianos si usamos un algoritmo de poda?. Justifica tu respuesta.

19. Describe brevemente como funciona el algoritmo de la mochila en su versión del Bactracking.

20. ¿Cual es la condición de poda que se utiliza para resolver el problema de la mochila por el método del backtracking?. ¿Cómo hay que organizar los datos para poder aplicar dicha condición?.

21. ¿Cual es el orden de complejidad del algoritmo de la mochila resuelto por backtracking?. Justifica tu respuesta.

22. ¿Se puede reducir el orden de complejidad del algoritmo de la mochila resuelto por backtracking si mejoramos la condición de poda?. Justifica tu respuesta.

23. ¿Cómo se obtiene el valor del límite para la poda en el problema de la mochila usando el método del backtracking?.

24. De los tres algoritmos usados para resolver el problema de la mochila (voraz, programación dinámica y bactracking) ¿cual de los tres proporciona una solucion de más valor para la mochila?. Justifica tu respuesta.