8

Backtracking y Branch and Bound

El método Backtracking

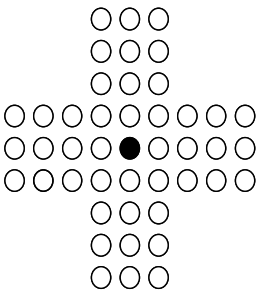
Aclaremos antes de nada que igual que hicimos con la denominación que otorgamos a los algoritmos greedy, manteniéndoles ese nombre, en todo lo que queda de capitulo también emplearemos los términos “backtracking” y “branch and bound” en lugar de su traducciones castellanas, por entender que las primeras están más extendidas que las segundas, sobre las que además no hay unanimidad, refiriéndose a ellas en ocasiones por vuelta atrás, progreso y retroceso, hacia adelante y hacia atrás, para el backtracking, o por ramificación y poda, acotación y generación, para el branch and bound, entre otras muchas.

Es interesante saber que el nombre backtrack fue acuñado por D.H. Lehmer en los años 50. Otros investigadores que contribuyeron a los fundamentos de esta técnica fueron R. J. Walker, que dio una versión algorítmica de ella en 1960, y Golomb y Baumert que presentaron una descripción general del backtracking asociada a una gran variedad de aplicaciones.

El backtracking representa una de las técnicas más generales en la búsqueda de principios fundamentales del diseño de algoritmos. Muchos problemas que tratan de búsquedas sobre un conjunto de soluciones o que buscan una solución óptima que satisfaga algunas restricciones, pueden resolverse usando la técnica backtracking.

Dentro del ámbito de la Inteligencia Artificial, o más generalmente de los Sistemas Inteligentes, algunos ejemplos típicos para la aplicación de esta técnica son los siguientes

1) Recorridos de laberintos. Una matriz de dimensión nxn puede representar un laberinto cuadrado, de modo que cada posición contiene un entero no negativo que indica si la casilla es transitable (0) o no lo es (∞). Las casillas [1,1] y [n,n] corresponden a la entrada y salida del laberinto y siempre serán transitables. Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la entrada a la salida. Para ello en cada etapa se van generando los posibles movimientos desde la casilla en la que nos encontremos.
2) Composición de rompecabezas o solución de solitarios. Por ejemplo en el conocido solitario de mesa llamado Continental, las piezas se colocan en un tablero tal y como indica la siguiente figura, en la que solamente queda vacía la casilla central:



Una pieza sólo puede moverse saltando sobre una de sus vecinas y cayendo en una posición vacía, al igual que en el juego de las damas, aunque aquí no están permitidos los saltos en diagonal. La pieza sobre la que se salta se retira del tablero. El problema consiste en diseñar un algoritmo que encuentre una serie de movimientos (saltos) que, partiendo de la configuración inicial expuesta en la figura, llegue a una situación en donde sólo quede una pieza en el tablero, que ha de estar en la posición central.

3) Los ya conocidos de coloreado de mapas, como sabemos, dado un grafo conexo y un número m > 0, se llama colorear el grafo a asignar un número i ($1 \leq i \leq m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales. Deseamos implementar un algoritmo backtracking que coloree un grafo dado.

Nótese que en estos problemas que hemos puesto como ejemplos, se dan circunstancias comunes en todos ellos, como son que no tenemos suficiente información para implementar una estrategia que nos conduzca a la solución que buscamos, que cada elección lleva a otro conjunto de elecciones, y que una sucesión de elecciones puede (o no) llevarnos a una solución

En ese sentido, y lo mismo que ocurría con los enfoques Greedy, Divide y Vencerás o basados en Programación Dinámica, la metodología backtracking podrá desarrollarse bajo una serie de hipótesis, que si bien no son ineludibles, su cumplimiento asegura la obtención de mejores resultados. Así para aplicar el método backtracking, la solución deseada debe ser expresable como una n-tupla (x_1, ..., x_n) en la que x_i se elige de algún conjunto finito S_i. A menudo el problema a resolver trata de encontrar un vector que maximiza (o minimiza) una función criterio $P(x_1, ..., x_n)$. A veces, también se trata de encontrar todos los vectores que satisfagan P.

Por ejemplo, ordenar los enteros en A(1..n) es un problema cuya solución es expresable mediante una n-tupla en la que x_i es el índice en A del i-ésimo menor elemento. La función de criterio P es la desigualdad $A(x_i) \leq A(x_{i+1})$, para $1 \leq i \leq n$. El conjunto S_i es finito e incluye a todos los enteros entre 1 y n. Aunque la ordenación no es uno de los problemas que habitualmente se resuelven con backtracking, si es un ejemplo de un problema familiar cuya solución puede formularse como una tupla. En este capítulo, estudiaremos algunos de los problemas cuyas soluciones pueden encontrarse usando backtracking.

La aplicabilidad de esta técnica reside en lo siguiente. Supongamos que m_i es el tamaño del conjunto S_i. Entonces hay $m = m_1 m_2 \cdot ... \cdot m_n$ n-tuplas posibles candidatos a satisfacer la función P. El enfoque enumerativo, de la fuerza bruta,

propondría formar todas esas tuplas y evaluar cada una de ellas con P, escogiendo la que diera un mejor valor. El algoritmo backtracking tiene como virtud el proporcionar la misma solución pero en mucho menos de m intentos. Su idea básica es construir el mismo vector escogiendo una componente cada vez, y usando funciones de criterio modificadas $P_i(x_1, ..., x_n)$, que a veces se llaman funciones de acotación, para testear si el vector que se está formando tiene posibilidad de éxito. La principal ventaja del método es que si se constata que a partir del vector parcial $(x_1, x_2, ..., x_i)$ no se podrá constituir una solución, entonces pueden ignorarse por completo $m_{i+1} \cdot ... \cdot m_n$ posibles test de vectores.

Destaquemos que en los algoritmos greedy se construía la solución buscada aprovechando la posibilidad de calcularla a trozos, pero con backtracking la elección de un sucesor en una etapa no implica su elección definitiva. Del mismo modo en Programación Dinámica, la solución se construía por etapas a partir del Principio del Óptimo, y los resultados se almacenaban para no tener que recalcular. Como iremos viendo eso no es posible en Backtracking.

Tanto estos problemas como otros muchos que resolveremos usando backtracking requieren que todas las soluciones satisfagan un complejo conjunto de restricciones. En cualquier problema estas restricciones podrán dividirse en dos categorías: explícitas e implícitas. Las restricciones explícitas son reglas que restringen cada x_i a tomar valores solo en un conjunto dado. Ejemplos comunes de restricciones explícitas son,

$x_i \geq 0$ $\quad S_i = \{\text{números reales no negativo}\}$
$x_i = 0,1$ $\quad S_i = \{0,1\}$
$l_i \leq x_i \leq u_i$ $\quad S_i = \{a: l_i \leq a \leq u_i\}$

Las restricciones explícitas pueden depender o no del caso particular del problema a resolver. Todas las tuplas que satisfagan las restricciones explícitas definen un posible espacio de soluciones del caso que estamos resolviendo. Las restricciones implícitas determinan cuál de las tuplas en ese espacio solución del caso que tratamos satisface la función de criterio. Por tanto, las restricciones implícitas describen la forma en la que los valores que pueden tomar las x_i deben relacionarse entre sí.

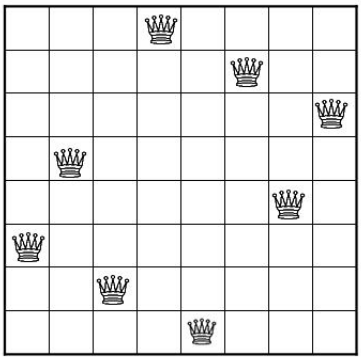
Como aclaración de estos conceptos, consideremos el siguiente problema, que se conoce como el Recorrido del Rey de Ajedrez: Dado un tablero de ajedrez de tamaño nxn, se coloca un rey en una casilla arbitraria de coordenadas (x,y). El problema consiste en determinar los n^2-1 movimientos que tiene que hacer el rey para visitar todas las casillas del tablero una sola vez, si tal secuencia de movimientos existe.

Si el tablero lo representamos por una matriz de dimensión nxn, cada elemento (x,y) de la matriz contendrá un número natural k indicando el número de orden en que ha sido visitada la casilla de coordenadas (x,y). Por tanto en lo que concierne a las restricciones explícitas, por la forma en la que hemos definido la estructura que representa la solución (en este caso una matriz bidimensional de números naturales), sabemos que sus componentes pueden ser números comprendidos entre cero (que indica que una casilla no ha sido visitada aún) y n^2, que es el orden del último movimiento posible. Entonces las restricciones implícitas limitarán el número de

hijos que se generan desde una casilla para comprobar que el correspondiente movimiento no lleve al rey fuera del tablero o sobre una casilla previamente visitada. En lo que sigue iremos desgranando la aplicación de la técnica y los elementos que necesitamos para ello sobre algunos ejemplos que, finalmente, resolveremos completamente.

Ejemplo 1: El problema de la ocho reinas

Un clásico problema combinatorio es el de colocar ocho reinas en una tablero de ajedrez 8 por 8, de modo que con las normas usuales de ese juego, no haya dos que se ataquen, es decir, que no haya dos entre ellas en la misma fila, columna o diagonal. Buscamos una situación como la que representa la siguiente figura,



Si numeremos las filas y columnas del tablero del 1 al 8, las reinas pueden también numerarse del 1 al 8. Como cada reina debe estar en una fila diferente, sin pérdida de generalidad podemos suponer que la reina i se coloca en la fila i.
Todas las soluciones para este problema, pueden representarse como 8-tuplas $(x_1,...,x_n)$ en las que x_i es la columna en la que se coloca la reina i. Las restricciones explícitas usando esta formulación son $S_i = \{1,2,3,4,5,6,7,8\}$, $1 \le i \le n$. Por tanto el espacio solución tendrá 8^8 tuplas. Las restricciones implícitas para este problema son que ningún par de x_i puedan ser iguales (es decir, todas las reinas deben estar en columnas diferentes) y que ningún par de reinas estén en la misma diagonal. La primera de estas restricciones implica que todas las soluciones son permutaciones de la 8-tupla (1,2,3,4,5,6,7,8). Esto lleva a reducir el tamaño del espacio solución de 8^8 tuplas a 8!. Veremos más adelante como formular la segunda restricción en términos de las x_i. Señalemos, a título de ejemplo, que una posible solución del problema es la (4,6,8,2,7,1,3,5) que muestra la anterior figura.

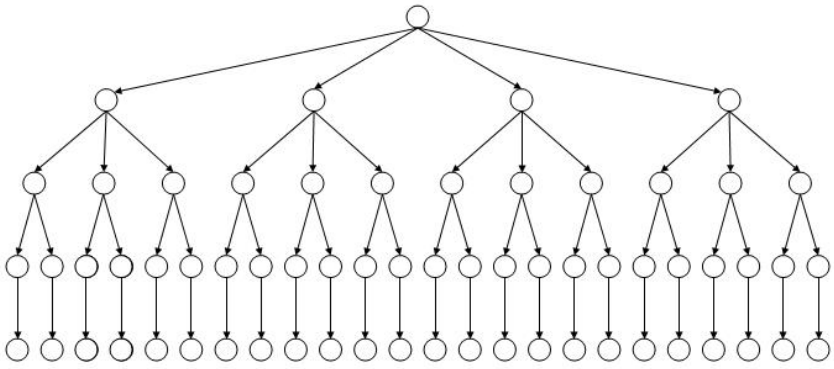
Ejemplo 2: La suma de subconjuntos.

Dados n+1 números positivos w_i, $1 \le i \le n$, y M, este problema trata de encontrar todos los subconjuntos de números w_i cuya suma valga precisamente M. Por ejemplo, si n = 4, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ y M = 31, entonces los

subconjuntos buscados son (11, 13, 7) y (24, 7). Más que representar el vector solución por los w_i que sumen M, podríamos representarlo dando los índices de tales w_i. Entonces las dos soluciones se describen por los vectores (1,2,4) y (3,4).
En general, todas las soluciones son k-tuplas $(x_1, x_2,...,x_n)$, $1 \le k \le n$, y soluciones diferentes pueden tener tamaños de tupla diferentes. Las restricciones explícitas requieren que $x_i \in \{j: j \text{ es entero y } 1 \le j \le n\}$. Las restricciones implícitas exigen que no haya dos iguales y que la suma de los correspondientes w_i sea M. Ya que deseamos evitar generar múltiples casos del mismo subconjunto, por ejemplo (1,2,4) y (1,4,2) representan el mismo subconjunto, otra restricción implícita que hay que imponer es que $x_i < x_{i+1}$, para $1 \le i < n$. En otra formulación del problema de la suma de subconjuntos, cada subconjunto solución se representa por una n-tupla $(x_1,...,x_n)$ tal que $x_i \in \{0,1\}$, $1 \le i < n$, y $x_i = 0$ si w_i no se elige y $x_i = 1$ si se elige w_i. Así, las soluciones del anterior caso son (1,1,0,1) y (0,0,1,1). Esta formulación expresa todas las soluciones usando un tamaño de tupla fijo. Por tanto puede haber diferentes modos de formular un problema de manera que todas las soluciones sean tuplas satisfaciendo algunas restricciones. Se puede comprobar que para estas dos formulaciones, el espacio solución tiene 2^4 tuplas distintas.
Los algoritmos backtracking determinan las soluciones del problema buscando sistemáticamente en el espacio de soluciones del caso considerado. Esta búsqueda se facilita usando una organización en árbol para el espacio solución. Para un espacio solución dado, pueden caber muchas organizaciones en árbol. Los siguientes ejemplos examinan algunas de las formas posibles para estas organizaciones.

Ejemplo 3: El problema de la n reinas.

Este es un problema que generaliza el anterior de las 8 reinas: Se tienen que colocar n reinas en un tablero nxn de modo que no haya dos que se ataquen, es decir, no puede haber dos reinas en la misma fila, columna o diagonal. Generalizando lo anterior, ahora el espacio de soluciones consiste en las n! permutaciones de la n-tupla (1,2,...,n). La figura muestra una posible organización en árbol para el caso n = 4.

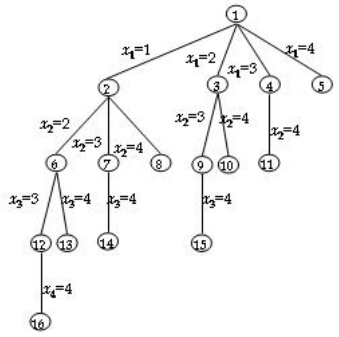


A un árbol como ese se le llama Árbol de Permutación. En él las aristas se etiquetan con los posibles valores de las x_i. Las aristas desde los nodos del nivel 1 al 2

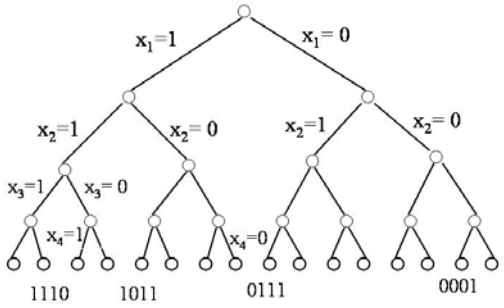
especifican los valores para x_1 ; el subarbol de la izquierda contiene todas las soluciones con $x_1 = 1$ y $x_2 = 2$, etc. Las aristas desde el nivel i al i+1 están etiquetadas con los valores de x_i. El espacio de soluciones está definido por todos los caminos desde el nodo raíz a un nodo hoja. Hay 4! = 24 nodos hoja en la figura.

Ejemplo 4: Suma de subconjuntos.

En el ejemplo 2 vimos dos versiones del espacio solución de la suma de subconjuntos. Las figuras siguientes muestran posibles organizaciones en árbol para cada una de estas formulaciones para n = 4. El primer árbol corresponde a la formulación por tamaño de la tupla. La aristas se etiquetan de modo que una desde el nivel de nodos i hasta el i+1 representa un valor para x_i. En cada nodo, el espacio solución se particiona en espacios subsolución, y se define para todos los caminos desde el nodo raíz a cualquier nodo en el árbol. Los posibles caminos son (), asociado al camino vacío desde la raíz a sí misma, (1), (12), (123), (1234), (124), (134), (2), (23), … Así, el subarbol de la izquierda define todos los subconjuntos conteniendo w_1, el siguiente todos los que contienen w_2 pero no w_1, etc.



El segundo árbol corresponde a la otra formulación. Una arista del nivel i al i+1 se etiqueta con el valor de x_i (0 o 1). Todos los caminos desde la raíz a las hojas definen el espacio solución. El subarbol de la izquierda define todos los subconjuntos que contienen w_1, y el de la derecha todos los subconjuntos que no contienen w_1, etc. Ahora hay 2^4 nodos hoja, que representan 16 posibles tuplas.



En lo que sigue desarrollaremos alguna terminología de cara a la organización en árbol de los espacios solución. Cada nodo en este árbol define un estado del problema. Todos los caminos desde la raíz a otros nodos definen el espacio de estados del problema. Estados solución son aquellos estados del problema S para los que el camino desde la raíz a S define una n-tupla en el espacio solución. En el primero de los arboles anteriores, todos los nodos son estados solución, mientras que en el segundo solo los nodos hoja son estados solución. Estados respuesta son aquellos estados solución S para los que el camino desde la raíz hasta S define una tupla que es miembro del conjunto de soluciones (es decir satisfacen las restricciones implícitas) del problema. La organización en árbol del espacio solución se referirá como el árbol de espacios de estado.

En cada nodo interno en el árbol de espacio de estados de los ejemplos 3 y 4 el espacio solución esta particionado en espacios sub-solución disjuntos. En todo lo que sigue, el espacio de estados se particionará en espacios subsolución disjuntos en cada nodo interno.

Las organizaciones en árbol de espacio de estados descritas en el ejemplo 4 se llaman arboles estáticos, porque las organizaciones en árbol son independientes del caso del problema que se vaya a resolver. En algunos problemas es ventajoso usar diferentes organizaciones de árbol para diferentes casos del problema. Entonces la organización en árbol se determina dinámicamente como el espacio solución que está siendo buscado. Las organizaciones en árbol que son dependientes del caso concreto del problema a resolver se llaman arboles dinámicos.

Cuando para algún problema se ha concebido un árbol de espacios de estado, podemos resolver este problema generando sistemáticamente sus estados, determinando cuales de estos son estados solución, y finalmente determinando que estados solución son estados respuesta.

Fundamentalmente hay dos formas diferentes de generar los estados del problema. Las dos comienzan con el nodo raíz y generan otros nodos. Un nodo que ha sido generado, pero para el que no se han generado aun todos sus nodos hijos, se llama un nodo vivo. El nodo vivo cuyos hijos están siendo generados actualmente se llama un E-nodo (nodo que está siendo expandido). Un nodo muerto es un nodo generado que o no se va a expandir o que todos sus hijos ya han sido generados. En ambos métodos de generar estados del problema tendremos una lista de nodos vivos.

Entonces,

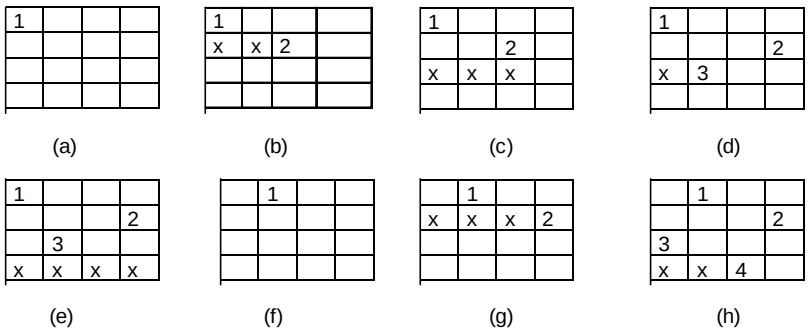
1) En el primero de estos dos métodos, tan pronto como un nuevo hijo C, del E-nodo en curso R, ha sido generado, este hijo se convierte en un nuevo E-nodo. R se convertirá de nuevo en E-nodo cuando el subarbol C haya sido explorado completamente. Esto corresponde a una generación primero en profundidad de los estados del problema.
2) En el segundo método de generación de estados, el E-nodo permanece como E-nodo hasta que se hace nodo muerto.

En ambos métodos, se usaran funciones de acotación para matar nodos vivos sin tener que generar todos sus nodos hijos. Esto habrá de hacerse cuidadosamente para que a la conclusión del proceso al menos se haya generado un nodo respuesta, o se hayan generado todos los nodos respuesta si el problema requiriera encontrar todas las soluciones. La generación de nodos primero en profundidad con

funciones de acotación se llama Backtracking. Los métodos de generación de estados en los que los E-nodos quedan como E-nodos hasta que se mueren originan los Métodos Branch and Bound que estudiaremos más adelante.

Aplicación al problema de las cuatro reinas.

Veamos cómo trabaja el backtracking en el problema de las cuatro reinas del ejemplo 3. Como función de acotación usaremos el criterio obvio de que si el camino hasta el E-nodo actualmente en curso es el $(x_1, x_2,...,x_n)$, entonces todos los nodos hijo con etiquetas padre-hijo x_{i+1} son tales que $(x_1,...,x_{i+1})$ representa una configuración del tablero en la que no hay dos reinas atacándose.
Comenzamos con el nodo raíz como único nodo vivo: Se convierte en E-nodo y el camino es el (). Generamos un hijo. Supongamos que los hijos se generan en orden ascendente. Así, se genera el nodo número 2 del árbol, y ahora el camino es el (1). Esto se corresponde a colocar la reina 1 en la columna 1. El nodo 2 se convierte en E-nodo. Entonces se genera el nodo 3 y se mata inmediatamente. El siguiente nodo generado es el 8 y el camino es el (1,3). El nodo 8 se convierte en E-nodo. Sin embargo es asesinado ya que todos sus hijos representan configuraciones del tablero que no pueden conducir a nodos respuesta. Entonces hacemos un backtracking al nodo 2 y generamos otro hijo, el nodo 13. El camino es ahora el (1,4). La siguiente figura muestra las configuraciones del tablero conforme procede el backtracking.



Esta figura muestra gráficamente las etapas que el algoritmo backtracking realiza para encontrar una solución. Las “x” representan posiciones de una reina que hemos ensayado y retirado debido a que había ya una reina atacando. En (b) la segunda reina se coloca en las columnas 1, 2 y finalmente queda en la columna 3. En (c) el algoritmo prueba las cuatro columnas y es incapaz de colocar la siguiente reina en alguna casilla. Entonces tiene lugar el backtracking. En (d) se mueve la segunda reina a la siguiente columna posible, la columna 4 y la tercera reina se sitúa en la columna 2. Los restantes tableros, (e,f,g,h) muestran las sucesivas etapas hasta que el algoritmo encuentra una solución.
Tras este ejemplo podemos dar ya una formulación general del backtracking. Supondremos que hay que encontrar todos los nodos respuesta, y no solo uno. Sea $(x_1, x_2,..., x_i)$ un camino desde la raíz hasta un nodo en el árbol de estados. Sea $T(x_1,$

x_2,..., x_i) el conjunto de todos los posibles valores de x_{i+1} tales (x_1, x_2,..., x_{i+1}) es también un camino hacia un estado del problema. Supondremos la existencia de funciones de acotación B_{i+1} (expresadas como predicados) tales que B_{i+1} (x_1, x_2,..., x_{i+1}) es falsa para un camino (x_1, x_2,..., x_{i+1}) desde el nodo raíz hasta un estado del problema solo si el camino no puede extenderse para alcanzar un nodo respuesta. Así los candidatos para la posición i+1 del vector solución X(1..n) son aquellos valores que son generados por T y satisfacen B_{i+1}. El algoritmo siguiente, Procedimiento Backtracking, representa el esquema general backtracking haciendo uso de T y B_{i+1}.

PROCEDIMIENTO BACKTRACK(n)
{Todas las soluciones se generan en X(1..n) y se imprimen tan pronto como se han determinado. T(X(1),...,T(k-1)) da todos los posibles valores de X(k) dado que habíamos escogido X(1),..., X(k-1). El predicado B_k(X(1),..., X(k)) determina los elementos X(k) que satisfacen las restricciones implícitas}

```
Comienzo
 k = 1
   Mientras k > 0 hacer
    Si queda algún X(k) no probado tal que
          X(k) ∈T(X(1),..., X(k-1)) y B_k (X(1),..., X(k)) = verdadero
    Entonces si (X(1),..., X(k)) es un camino hacia un nodo respuesta
          Entonces imprimir (X(1),..., X(k))
          k = k+1
    sino k = k-1
Fin
```

Nótese que T() dará el conjunto de todos los posibles valores que pueden colocarse como primera componente, X(1), del vector de soluciones. X(1) tomara aquellos valores para los que la función de acotación B_1(X(1)) es verdad. Nótese también como se generan los elementos de una manera primero en profundidad. k se incrementa continuamente, creciendo un vector de soluciones hasta que o bien se encuentra una solución, o queda un valor no ensayado de X(k). Cuando k se decrementa, el algoritmo debe resumir la generación de posibles elementos a la k-ésima posición que aún no haya sido ensayada. Por tanto se debe desarrollar un procedimiento que genere estos valores en algún orden. Si solo se desea una solución, bastara con colocar un "devuelve" tras la sentencia de imprimir.
El siguiente algoritmo, Recur_Backtrack, da la formulación recursiva del método,

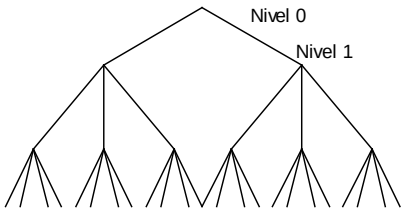
PROCEDIMIENTO RECUR_BACKTRACK(K)
{De entrada, los primeros k-1 valores X(1),...,X(k-1) del vector solución X(1..n) han sido asignados}

```
Comienzo
   Para cada X(k) tal que
      X(k) ∈T(X(1),..., X(k-1)) y B (X(1),..., X(k)) = verdadero hacer
            Si (X(1),..., X(k)) es un camino hacia un nodo respuesta
            Entonces imprimir (X(1),..., X(k))
         RECUR_BACKTRACK(K+1)
Fin
```

El vector solución (x_1,...,x_n) se trata como una array global X(1..n). Todos los posibles elementos de la k-ésima posición de la tupla que satisface B_k se generan, uno por uno, y se adjuntan al vector en curso X(1),..., X(k-1). Cada vez que X(k) se añade, se hace un chequeo para ver si hemos encontrado una solución, y entonces el algoritmo se llama recursivamente. Cuando el lazo para (for) se pone en marcha, no existen más valores para X(k) y la actual copia de RECUR_BACTRACK termina.
Nótese que cuando k excede a n, T(X(1),..., X(k-1)) devuelve el conjunto vacío y así el lazo para (for) no se efectúa. Nótese también que este programa imprime todas las soluciones, y supone que tuplas de diversos tamaños pueden constituir una solución. Si se desea solo una solución, puede hacerse alguna señal para indicar la primera ocurrencia de ese suceso.

Eficiencia.

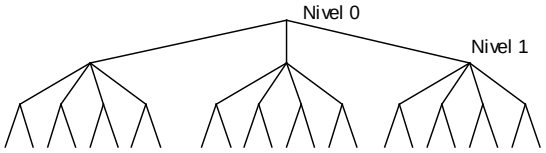
La eficiencia de los dos programas backtracking que acabamos de ver depende básicamente de cuatro factores: (i) el tiempo necesario para generar el siguiente X(k), (ii) el número de X(k) que satisfagan las restricciones explícitas, (iii) el tiempo para las funciones de acotación B_i, y (iv) el número de X(k) que satisfagan las B_i para todo i. Las funciones de acotación se entienden buenas si reducen considerablemente el número de nodos que generan. Sin embargo, estas funciones de acotación buenas, consumen mucho tiempo en evaluaciones, por lo que hay que buscar un equilibrio entre el tiempo global de computación, y la reducción del número de nodos generados.
Un principio general que proporciona búsquedas eficientes es el que se conoce con el nombre de re-arreglo. En muchos problemas los conjuntos S_i que definen las restricciones explicitas, pueden ordenarse de cualquier orden. Esto sugiere que todo lo demás no se altera y por tanto es más eficiente hacer la siguiente elección desde el conjunto con menos elementos. Esta estrategia no sirve para el problema de las n reinas, y pueden construirse ejemplos que demuestran que este principio no siempre funciona. Pero con intención teórico-informativa, puede mostrarse que en promedio la elección en el conjunto con menos elementos es la más eficiente.
El valor potencial de esta heurística se exhibe en la siguiente figura para dos árboles de búsqueda backtracking de un mismo problema.



(a)

Si podemos eliminar un nodo en el nivel 1 de la figura (a), estamos eliminando de consideración efectivamente doce posible 4-tuplas.

Sin embargo, si eliminamos un nodo en el nivel 1 de la figura (b), solo eliminamos ocho tuplas.



(b)

Como hemos comentado previamente, hay 4 factores que determinan el tiempo requerido por un algoritmo backtracking. Cuando se selecciona una organización en árbol de estados, las 3 primeras son relativamente independientes del caso concreto a resolver. Solo la cuarta, el número de nodos generados, varía de un caso a otro. Un algoritmo backtracking en un caso podría generar solo O(n) nodos, mientras que en otro (relativamente parecido) podría generar casi todos los nodos del árbol de espacio de estados.

Si el número de nodos en el espacio solución es 2^n o n!, el tiempo del peor caso para el algoritmo backtracking variaría generalmente entre $O(p(n)2^n)$ y O(q(n)n!) respectivamente, siendo p y q polinomios en n. La importancia del backtracking reside en su capacidad para resolver algunos casos con grandes valores de n en muy poco tiempo. La única dificultad está en predecir la conducta del algoritmo backtracking en el caso que deseemos resolver. Una alternativa para estimar el número de nodos que se generaran con un algoritmo backtracking trabajando sobre un cierto caso I se consigue empleando el método de Monte Carlo. Pero esto se sale de los objetivos que aquí perseguimos.

De todos modos es interesante comentar que una estimación razonable del número de nodos que se generaran por medio de un algoritmo backtracking, puede obtenerse con el Método de Monte Carlo seleccionando diferentes caminos aleatorios (normalmente no más de 20) y determinando el promedio de estos valores.

Solución del problema de las n reinas.

Generalizamos el problema para considerar un tablero nxn y encontrar todas las formas de colocar n reinas que no se ataquen. A partir del problema de las 4 reinas, observamos que podemos tomar $(x_1,...,x_n)$ representando una solución si x_i es la columna de la i-ésima fila en la que la reina i está colocada. Los x_i's serán todos distintos ya que no puede haber dos reinas en la misma columna. El problema es como comprobar que dos reinas no estén en la misma diagonal. Si imaginamos las casillas del tablero numeradas como los índices de un array bidimensional A(1..n,1..n), observamos que cada elemento en la misma diagonal que vaya de la parte superior izquierda a la inferior derecha, tiene el mismo valor ""fila-columna". También, cualquier elemento en la misma diagonal que vaya de la parte superior

derecha a la inferior izquierda, tiene el mismo valor "fila+columna". Esto se observa fácilmente a partir de la siguiente figura,

			(1,4)				
(2,1)		(2,3)					
	(3,2)						
(4,1)		(4,3)					
			(5,4)				(5,8)
(6,1)				(6,5)		(6,7)	
	(7,2)				(7,6)		
		(8,3)		(8,5)		(8,7)	

Supongamos que dos reinas están colocadas en las posiciones (i,j) y (k,l). Entonces estarán en la misma diagonal solo si,

$$i - j = k - l \text{ ó } i + j = k + l$$

La primera ecuación implica que

$$j - l = i - k$$

y la segunda que

$$j - l = k - i$$

Por tanto, dos reinas están en la misma diagonal si y solo si

$$|j-l| = |i-k|$$

El siguiente algoritmo devuelve un valor booleano que es verdadero si la k-ésima reina puede colocarse en el valor actual de X(k). Testea si X(k) es distinto de todos los valores previos X(1),...,X(k-1), y también si hay alguna otra reina en la misma diagonal, siendo en total su tiempo de ejecución de O(k-1). El esqueleto del algoritmo es

```
PROCEDIMIENTO COLOCA(K)
{X es un array cuyos k primeros valores han sido ya asignados}
Comienzo
  Para i:=1 hasta k hacer
     Si X(i) = X(k) o ABS(X(i)-X(k)) = ABS(i-k)
     Entonces Devolver (falso)
  Devolver (verdadero)
Fin
```

Ahora, usando este algoritmo podemos refinar el método backtracking general que dimos al principio, para dar una solución precisa al problema de las n reinas, y por tanto al de las ocho reinas.

PROCEDIMIENTO NREINAS(N)

{Usando backtracking este algoritmo imprime todos los posibles emplazamientos de n reinas en un tablero nxn de modo que no se ataquen}

```
Comienzo
  X(1) = 0, k = 1
  Mientras k > 0 hacer
      X(k) = X(k) + 1
      Mientras X(k) ≤ n y no COLOCA (k) hacer
          X(k) = X(k) + 1
      Si X(k) ≤ n
        Entonces Si k = n
              Entonces imprimir (X)
              Sino k = k + 1; X(k) = 0
        Sino k = k - 1
Fin
```

Nótese que en un tablero 8x8 hay $\mathbf{C}_{64,8}$ formas posibles de colocar 8 reinas, es decir, más de 4.400 millones de tuplas diferentes, un número de tuplas más que formidable. Sin embargo, permitiendo solo emplazamientos de reinas en filas y columnas distintas, necesitamos examinar, a lo sumo, 8!, es decir, 40.320 8-tuplas.

Solución para la suma de subconjuntos.

El problema de la suma de subconjuntos, un típico problema de la clase NP, es de suma importancia en distintos campos, pero especialmente destaca en el de la Criptografía. También es fundamental de cara a resolver problemas como el de la mochila.

Supongamos que tenemos n números positivos distintos (usualmente llamados pesos) y que queremos encontrar todas las combinaciones de estos números que sumen M. Esta es la más clásica formulación del problema de la suma de subconjuntos. Los anteriores ejemplos 2 y 4 mostraron como podríamos formular este problema usando tamaños de las tuplas fijos o variables. Consideraremos una solución backtracking usando la estrategia del tamaño fijo de las tuplas. En este caso el elemento X(i) del vector solución es uno o cero, dependiendo de si el peso W(i) está incluido o no.

Los hijos de cualquier nodo en el árbol se generan fácilmente. Para un nodo en el nivel i, el hijo de la izquierda corresponde a X(i) = 1, y el de la derecha a X(i) = 0. Una elección sencilla de las funciones de acotación es que $B_k(X(1), ..., X(k))$ = verdadero si y solo si,

$$\sum_{1..k} W(i)X(i) + \sum_{1..k} W(i) \geq M$$

Claramente X(1), ..., X(k) no pueden conducir a un nodo respuesta si no se verifica esta condición. Las funciones de acotación pueden fortalecerse si suponemos los W(i)'s en orden creciente. En este caso, X(1),..,X(k) no pueden llevar a un nodo respuesta si

$$\sum_{1..k} W(i)X(i) + W(k+1) > M$$

Por tanto las funciones de acotación que usaremos serán las definidas de la siguiente forma: $B_k(X(1), ..., X(k))$ es verdadero si y solo si

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M \text{ y } \sum_{1..k} W(i)X(i) + W(k+1) \leq M$$

Ya que nuestro algoritmo no hará uso de B_n, no necesitamos preocuparnos por la posible aparición de W(n+1) en esta función. Pero, aunque hasta aquí hemos especificado todo lo que es necesario para usar cualquiera de los esquemas Backtracking, resultaría un algoritmo más simple si diseñamos a la medida cualquiera de esos esquemas al problema que estemos tratando. Esta simplificación resulta de la comprobación de que si X(k) = 1, entonces

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) > M$$

Por simplicidad refinaremos el esquema recursivo. El algoritmo resultante es el siguiente,

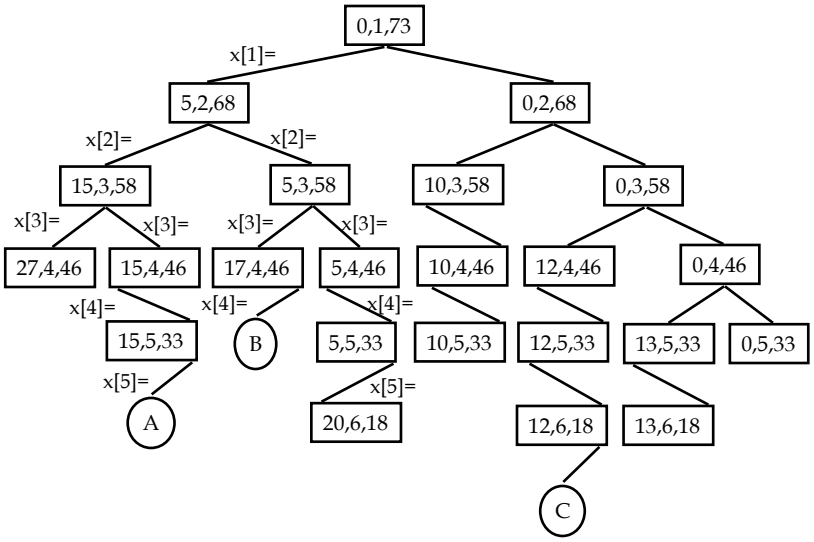
ALGORITMO SUMASUB (S, k, R)
{Notamos $S = \sum_{1..k-1} W(j)X(j)$ y $R = \sum_{k..n} W(j)$. Los W(j) están en orden creciente. Se supone que $W(1) \leq M$ y que $\sum_{1..n} W(i) \geq M$}
Comienzo
{Generación del hijo izquierdo. Nótese que $S+W(k) \leq M$ ya que B_{k-1} = verdadero}
X(k) = 1
{4} Si s + W(k) = M
{5} Entonces Para i = 1 hasta k imprimir X(j)
Sino
{7} Si S + W(k) + W(k+1) ≤ M
Entonces SUMASUB(S + W(k), k+1, R-W(k))
{Generación del hijo derecho y evaluación de B_k}
Si S + R - W(k) ≥ M y S + W(k+1) ≤ M
Entonces X(k) = 0
SUMASUB(S, K+1, R-w(K))
Fin

El algoritmo evita calcular cada vez $\sum_{1..k} W(i)X(i)$ y $\sum_{k+1..n} W(i)$ manteniendo estos valores en las variables S y R respectivamente. El algoritmo asume $W(1) \leq M$ y $\sum_{1..n} W(i) \geq M$. La llamada inicial se hace como SUMASUB(0, 1, $\sum_{1..n} W(i)$). Es interesante destacar que el algoritmo no usa explícitamente el test k > n para terminar la recursión. Este test no es necesario ya que, desde que empieza el algoritmo, $S \neq M$ y $S + R \geq M$. De aquí que $R \neq 0$ y por tanto k no pueda ser mayor que n. Nótese también que en la línea (7), como $S + W(k) < M$ y $S + R \geq M$, se sigue que $R \neq W(k)$ y por tanto $k + 1 \leq n$. Obsérvese también que si $S + W(k) = M$ en la línea (4), entonces X(k+1),...,X(n) deberían ser cero. Estos ceros se omiten del output de la línea (5). En la línea (7) no comprobamos si

$$\sum_{1..k} W(i)X(i) + \sum_{k+1..n} W(i) \geq M$$

puesto que ya sabemos que $S + R \geq M$ y que X(k) = 1.

Para ilustrar el algoritmo, desarrollemos el árbol correspondiente al siguiente caso. Tomamos $n = 6$, $M = 30$, $W = (5, 10, 12, 13, 15, 18)$. Entonces representando en los rectángulos los valores de S, k y R, en cada llamada, obtenemos



donde los tres círculos A, B y C se corresponden a las tres soluciones que tiene el problema: A = (1,1,0,0,1); B = (1,0,1,1) y C = (0,0,1,0,0,1). Nótese que de este modo se construyeron 26 nodos (del total de 2^7-1 = 127).

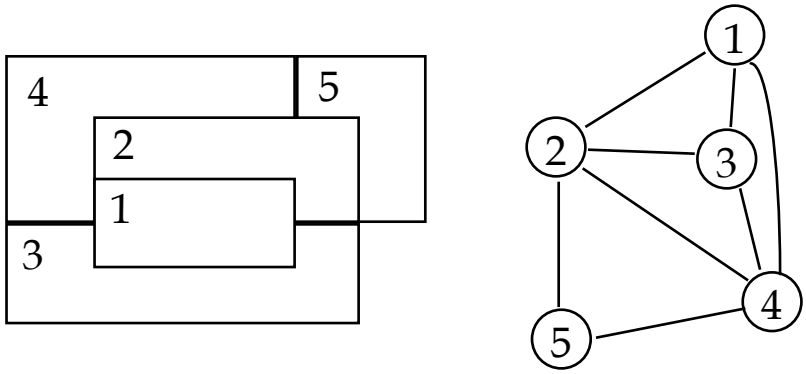
El problema del coloreo de grafos.

Sea G un grafo y m un número entero positivo. Queremos saber si los nodos de G pueden colorearse de tal forma que no haya dos vértices adyacentes que tengan el mismo color, y que solo se usen m colores para esa tarea. Este es el célebre problema de la m-colorabilidad. El problema de optimización de la m-colorabilidad, pregunta por el menor número m con el que el grafo G puede colorearse. A ese entero se le denomina Número Cromático del grafo.

Como sabemos, un grafo se llama plano si y solo si puede pintarse en un plano de modo que ningún par de aristas se corten entre sí. Un caso especial famoso del problema de la m-colorabilidad es el problema de los cuatro colores para grafos planos. Dado un mapa cualquiera el problema consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, de modo que para ese coloreo se empleen solo cuatro colores.

Como sabemos este problema, que tiene innumerables aplicaciones, siendo las más recientes las referidas a diseño de páginas web y a los procesos de recuperación de información, es fácilmente traducible a la nomenclatura de grafos, y evidentemente representable en esos términos, como se ilustra gráficamente en la siguiente figura, que considera un mapa con cinco regiones, a la izquierda, y su contrapartida en la versión de grafo plano (cada región se modela con un nodo, y si dos regiones son

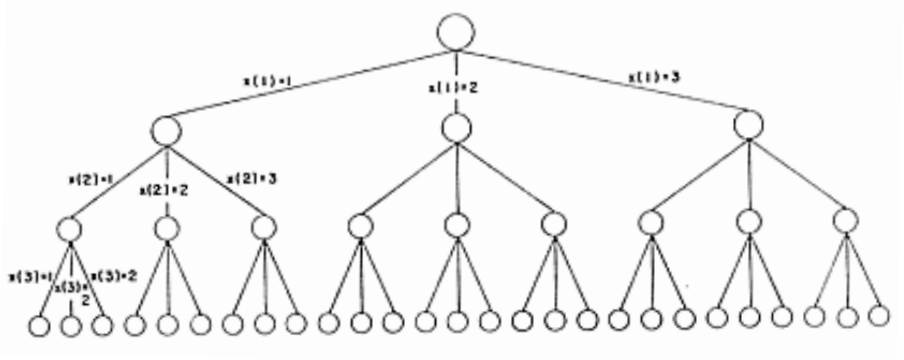
adyacentes, sus correspondientes nodos se conectan con un arco)



En lo que sigue consideraremos no solo los grafos que pueden obtenerse de mapas, sino cualquier grafo. Estamos interesados en determinar todas las diferentes formas en las que puede colorearse un grafo dado usando a lo más m colores.
Supongamos que representamos el grafo por su matriz de adyacencia, que llamaremos GRAFO(i,j), siendo

GRAFO(i,j) = verdadero si (i,j) es una arista de G.
GRAFO(i,j) = falso en caso contrario

Preferimos usar valores booleanos ya que al posible algoritmo solo le interesaría saber si existe o no una determinada arista, de ahí precisamente que empleemos la representación del grafo por su matriz de adyacencia.
Los colores se pueden representar por los enteros 1, 2, ..., m, y entonces las soluciones vendrán dadas por las n-tuplas (X(1), ..., X(n), donde X(i) será el color del vértice i. Así, para un caso con n = 3 y m = 3, el espacio de estados del correspondiente problema sería el ilustrado en la siguiente figura



en el que cada nodo en el nivel i tiene m hijos correspondientes a las m posibles asignaciones para X(i), $1 \le i \le n$, y donde los nodos en el nivel n+1 son nodos hoja.
Ahora, usando la formulación recursiva del procedimiento backtracking que propusimos con anterioridad, puede construirse el siguiente esqueleto de algoritmo para resolver este problema,

ALGORITMO M_COLOR(k)
Comienzo
  para v = 1 hasta m hacer
    X[k] = v
    Si SIGUIENTEVALOR(k) Entonces
      Si k = n Entonces imprimir(X)
      Sino M_COLOR(k+1)
Fin

ALGORITMO SIGUIENTEVALOR(k)
{Se emplean las siguientes variables b, booleana; j, con valores enteros, inicializadas en b = verdadero y j = 1}
Comienzo
    Mientras (j < k) y b = verdadero hacer
      si GRAFO[k,j] y (x[k] = x[j]) Entonces
        b = falso
      Sino j = j+1
    Devuelve(b)
Fin

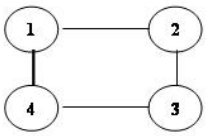
Para este algoritmo, el número de nodos internos en el espacio de estados es

$$\sum_{i=1..n-1} m^i$$

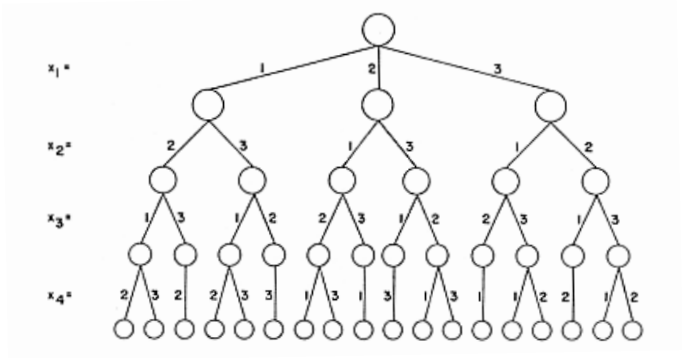
En cada nodo interno SiguienteValor invierte O(nm) en determinar el hijo correspondiente a un coloreo legal. Así, el tiempo total está acotado por

$$\sum_{i=1..n-1} m^i n = n(m^{n+1} - 1)/(m-1) = O(n\, m^n)$$

y el algoritmo trabaja en un tiempo $O(nm^n)$.
Por ejemplo para un grafo como el de la figura,



El árbol que genera M-Color es el siguiente



siendo el espacio de estados subyacente un árbol de grado m (3) y altura n+1 (4).

Ciclos Hamiltonianos

Sea G = (V, E) un grafo conexo con n vértices. Como sabemos un ciclo Hamiltoniano es un camino circular a lo largo de los n vértices de G que visita cada vértice de G una vez y vuelve al vértice de partida, que naturalmente es visitado dos veces, es decir, se trata de un camino $v_1 v_2 \ldots v_{n+1}$ tal que:

- $v_i \in V$, i=1,…,n+1,
- $(v_i, v_{i+1}) \in E$, i = 1,…,n,
- $v_1 = v_{n+1}$,
- $v_i \neq v_j$, $\forall$i,j=1,…,n tales que i ≠ j.

Estamos interesados en construir un algoritmo backtracking que determine todos los ciclos hamiltonianos de G, que puede ser dirigido o no.
El vector backtracking solución $(x_1, ..., x_n)$ se define de modo que x_i represente el i-ésimo vértice visitado en el ciclo propuesto. Todo lo que se necesita es determinar cómo calcular el conjunto de posibles vértices para x_k si ya hemos elegido $x_1, ..., x_{k-1}$. Entonces,

- Si k = 1, entonces X(1) puede ser cualquiera de los n vértices. Para evitar la impresión del mismo ciclo n veces, exigimos que X(1) = 1.
- Si 1 < k < n, entonces X(k) puede ser cualquier vértice v que sea distinto de X(1), X(2), ..., X(k-1) que esté conectado por una arista a X(k-1).
- X(n) puede ser solo el único vértice restante y debe estar conectado a X(n-1) y a X(1).

El siguiente algoritmo, en el que suponemos que tenemos definida la matriz de adyacencia del grafo, y la notamos como es habitual G[i, j], determina el siguiente vértice posible para el ciclo propuesto:

```
ALGORITMO HAMILTONIANO(k)
Comienzo
  Para v = 1 hasta n hacer
    X[k]:=v;
    si CERRABLE(k) Entonces
        si k=n entonces
          escribir(X)
        sino HAMILTONIANO(k+1)
Fin
```

```
ALGORITMO CERRABLE(k)
Comienzo
   b=  G[X[k-1], X[k]]
   Para i:=1 hasta k-1 Mientras b = verdadero hacer
     Si X[i] = X[k] Entonces b = falso
   si k = n y NO G[X[n], X[1]] Entonces
      b = falso
   Devolver (b)
Fin
```

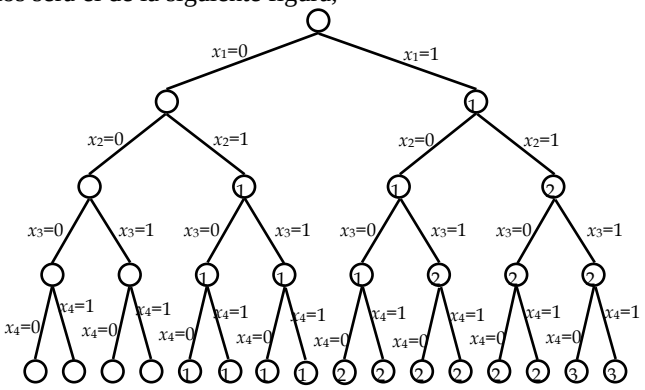
El algoritmo comienza inicializando la matriz de adyacencia G[i,j], se toma X(2) = 0 y X(1) = 1, y entonces se ejecuta la llamada HAMILTONIANO(2).

El problema de la mochila

Supuesta la versión booleana, como sabemos el problema consiste en determinar los elementos a escoger para incluir en la mochila, de manera que se maximice lo transportado, es decir, tenemos

$$\text{Max } \{\sum_{i=1..n} pi \cdot x_i \,/\, \sum_{i=1..n} w_i \cdot x_i \leq M;\ x_i \in \{0, 1\},\ i = 1...n\}$$

El espacio solución para este problema consiste en las 2^n formas distintas de asignar los valores cero o uno a las x_i. Así ese espacio solución es el mismo que el del problema de la suma de subconjuntos, siendo por tanto posibles dos organizaciones (por tamaños fijos o variables de tuplas). Si escogemos la primera de las representaciones, es decir, la de tamaño de tuplas fijo, el espacio de búsqueda que tendremos será el de la siguiente figura,



De todos modos, e independientemente de la organización que escojamos, para resolver el problema necesitamos funciones de acotación que ayuden a podar algunos nodos vivos sin que efectivamente haya que expandirlos. Se obtiene una buena función de acotación usando como cota superior el valor de la mejor solución factible obtenible expandiendo el nodo vivo dado y cualquiera de sus descendientes. Si esta cota superior no es mayor que el valor de la mejor solución determinada hasta ese momento, entonces ese nodo vivo puede podarse.

Sobre el anterior árbol, si en un nodo Z ya se han determinado los valores de las x_i, $1 \leq i \leq k$, entonces se puede obtener una cota superior para Z relajando el requerimiento de que $x_i = 0$ o 1, a que $x_i \in [0,1]$, para $k+1 \leq i \leq n$, y usando el algoritmo greedy que conocemos para resolver ese problema relajado, es decir, resolviendo lo que se conoce como la relajación lineal del problema de la mochila que se plantea en ese nodo.

El siguiente algoritmo COTA(P,W,k,M) determina una cota superior para la mejor solución obtenible expandiendo cualquier nodo Z en el nivel k+1 del árbol de estados. Los pesos y costos de los objetos son W(i) y P(i). Así mismo se nota

$$P = \sum P(i)X(i)$$

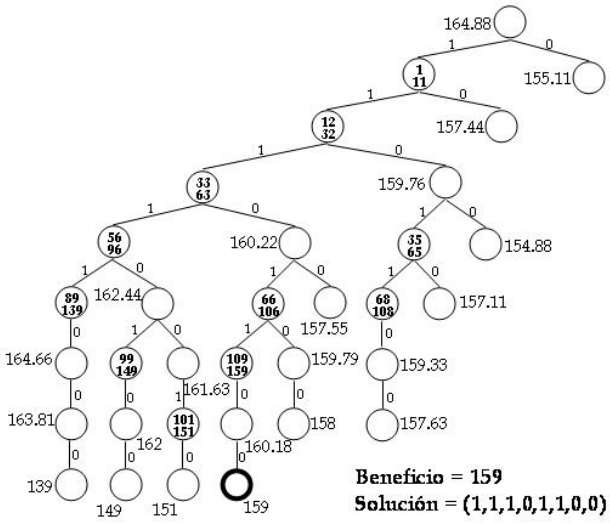
y se supone que $P(i)/W(i) \geq P(i+1)/W(i+1)$, $1 \leq i \leq n$. Evidentemente P es el beneficio total asociado a la solución en curso, w el peso total asociado a esa solución, k el índice del ultimo ítem eliminado, y como habitualmente M el tamaño de la mochila

ALGORITMO COTA(P,W,k,M)
```
Comienzo
  b = P; c = W
  Para i = k+1 hasta n hacer
     c = c + W(i)
     Si c < M Entonces b = b + P(i)
           Sino Devolver (b + (1 - (c-M)/W(i)P(i))
  Devolver (b)
Fin
```

A partir de este algoritmo se sigue que la cota para un hijo factible a la izquierda de un nodo Z es la misma que para Z. Así, la función de acotación no necesita usarse cada vez que el algoritmo backtracking haga un movimiento hacia el hijo a la izquierda de un nodo. Como el algoritmo backtracking intentará hacer un movimiento hacia un hijo a la izquierda siempre que tenga que elegir entre un hijo a la izquierda y a la derecha, vemos que la función de acotación hay que usarla solo después de una serie de movimientos exitosos a hijos a la izquierda (es decir movimientos hacia hijos a la izquierda factibles).
Si M, n, W y P mantienen el significado anterior, fw nota el peso final de la mochila, fp el beneficio final máximo, las variables son booleanas, X(k) = 0 si W(k) no está en la mochila (en caso contrario X(k) = 1) y como antes suponemos que $P(i)/W(i) \geq P(i+1)/W(i+1)$}, podemos proponer el siguiente algoritmo iterativo para resolver el problema:

ALGORITMO RETROMOCHILA (M, n, W, P, fw, fp, X)
```
Comienzo
  cw = cp = 0; k = 1; fp = -1;                  {cw y cp son peso y beneficio en curso}
    Mientras k ≤ n y cw + W(k) ≤ M hacer          {coloca k en la mochila}
      cw = cw+W(k); cp = cp+P(k); Y(k) = 1; k = k+1    {coloca W(k) en mochila}
    Repetir
      Si k > n Entonces fp = cp; fw = cw; k = n; X = Y      {actualiza la solución}
      Sino Y(k) = 0                                        {debido a k se
sobrepasa M}
    Mientras COTA(cp, cw, k, M) ≤ fp hacer                 {Aquí COTA = fp}
        Mientras k ≠ 0 y Y(k) ≠ 1 hacer
            k = k + 1                {busca el mayor peso incluido en la mochila}
        Si k = 0 Entonces Volver                 {aquí termina el algoritmo}
        Y(k) = 0; cw = cw - W(k); cp = cp - P(k)     {elimina el ítem k}
    k = k+ 1
Fin RETROMOCHILA
```

Para ilustrar el funcionamiento de este algoritmo, consideremos un problema definido por M = 110, n = 8, P = (11, 21, 31, 33, 43, 53, 55, 65) y W = (1, 11, 21, 23, 33, 43, 45, 55). Entonces el árbol que genera el algoritmo es el de la siguiente figura.



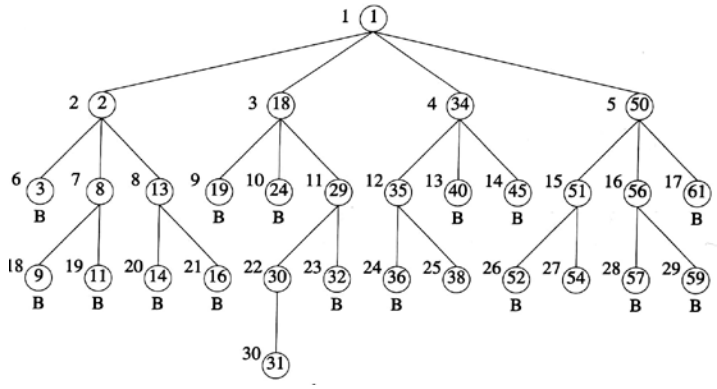
El árbol muestra las distintas elecciones que se van haciendo para el vector Y. El i-ésimo nivel corresponde a una asignación de uno o cero a Y(i), incluyendo o excluyendo el peso W(i). Los dos números dentro de un nodo son el peso (cw) y el beneficio (cp), dando la asignación inferior al nivel del nodo. Los nodos que no contienen números implican que el peso y el beneficio es el mismo que el de su padre. El número fuera de cada hijo a la derecha y fuera de la raíz es la cota correspondiente a ese nodo. La cota para el hijo de la izquierda es la misma que la de su padre. Cada vez que actualizamos fp, también actualizamos X. Al final fp = 159 y X = (1,1,1,0,1,1,0,0).

Hay que destacar que de los 2^9-1 = 511 nodos en el espacio de estados solo se han generado 33. Este número podría haberse reducido a 26 teniendo en cuenta que como todos los P(i) son enteros, el valor de todas las soluciones factibles también tiene que ser entero, y consecuentemente algunos nodos no tendrían que haberse expandido.

Branch and Bound

Con el término Branch and Bound queremos referirnos a un método de búsqueda en un espacio de estados que actúa en forma de ramificar y acortar, de tal modo que todos los hijos de un E-nodo se generan antes de que cualquier otro nodo vivo pueda convertirse en el E-nodo. Conocemos dos formas de búsqueda sobre grafos (la llamada primero en profundidad y la denominada primero en anchura), en las que la exploración de un nuevo nodo no puede comenzar hasta que el nodo actualmente bajo análisis haya sido explorado completamente. Ambas estrategias se generalizan

con el Branch and Bound, en cuya terminología una búsqueda en el espacio de estados primero en anchura, se llamara FIFO (Primer Llegado-Primer Servido), ya que la lista de los nodos vivos es una lista FIFO (una cola). Por otro lado, una búsqueda en el espacio de estados de tipo primero en profundidad se llamara LIFO (Ultimo Llegado-Primer Servido), ya que la lista de nodos vivos es una lista LIFO (una pila). Usaremos funciones de acotación como en el caso del backtracking, de manera que nos beneficiemos no generando sub-árboles que no contengan nodos respuesta. Los algoritmos generados por esta técnica son normalmente de orden exponencial o peor en su peor caso, pero su aplicación en casos muy grandes, ha demostrado ser eficiente (incluso más que backtracking).
Veamos, en la siguiente figura, sobre el problema de las cuatro reinas como buscaría en el árbol de estados un algoritmos FIFO Branch and Bound: Inicialmente hay solo un nodo vivo, el nodo 1. Este, representa el caso en que ninguna reina ha sido colocada en el tablero. Entonces este nodo se convierte en E-nodo. Se expande y se generan sus hijos, los nodos 2, 18, 34 y 50. Estos nodos representan un tablero con la reina 1 en la fila 1 y en las columnas 1, 2, 3 y 4 respectivamente. Ahora los únicos nodos vivos son los 2, 18, 34 y 50. Si los nodos fueron generados en este orden, entonces el siguiente E-nodo es el 2. Se expande, y entonces se generan los nodos 3, 8, y 13. Inmediatamente se elimina el nodo 3 usando la misma función de acotación que empleamos en este ejemplo cuando lo desarrollamos con la técnica backtracking, es decir, tales que B_{i+1} (x_1, x_2,..., x_{i+1}) es falsa para un camino (x_1, x_2,..., x_{i+1}) desde el nodo raíz hasta un estado del problema solo si el camino no puede extenderse para alcanzar un nodo respuesta. Los nodos 8 y 13 se añaden a la cola de nodos vivos. Ahora el nodo 18 se convierte en el siguiente E-nodo, y se generan los nodos 19, 24 y 29. Pero los nodos 19 y 24 se eliminan como resultado de la función de acotación, y el nodo 29 se añade a la cola de nodos vivos. El siguiente E-nodo es el 34.



Nodo Respuesta

Los nodos que han resultado eliminados como resultado de las funciones de acotación, se indican con una B bajo ellos. Los números dentro de los nodos representan referencias numéricas en la figura, y los números fuera de los nodos dan el orden en el que se generan por la regla FIFO Branch and Bound. En el momento en que se alcanza el nodo respuesta, el nodo 31, los únicos nodos vivos son los 38 y

54. Una simple comparación entre las figuras que mostraban el espacio de estados para este problema, cuando se resolvía con backtracking, y esta de arriba indican que aquel es un método de búsqueda mejor para este problema.

Estamos así en condiciones de dar una primera descripción del método. Branch and Bound es un método de búsqueda general que se aplica conforme a lo siguiente:

1) Explora un árbol comenzando a partir de un problema raíz (el problema original con su región factible completa)
2) Entonces se aplican procedimientos de cotas inferiores y superiores al problema raíz.
    a. Si las cotas cumplen las condiciones que se hayan establecido, habremos encontrado la solución óptima y el procedimiento termina.
    b. Si ese no fuera el caso, entonces la región factible se divide en dos o más regiones, dando lugar a distintos subproblemas. Esos subproblemas particionan la región factible. La búsqueda se desarrolla en cada una de esas regiones. El método (algoritmo) se aplica recursivamente en cada uno de los subproblemas.
    c. Si se encuentra una solución óptima para un subproblema, será una solución factible para el problema completo, pero no necesariamente el óptimo global.
    d. Cuando en un nodo (subproblema) la cota superior es menor que el mejor valor conocido en la región, no puede existir un óptimo global en el subespacio de la región factible asociada a ese nodo, y por tanto ese nodo puede ser eliminado en posteriores consideraciones.
3) La búsqueda sigue hasta que se examinan o "podan" todos los nodos, o hasta que se alcanza algún criterio pre-establecido sobre el mejor valor encontrado y las cotas superiores de los problemas no resueltos

Profundizando un poco más en la operatividad del método, en cada nodo i tendremos:

- Una cota superior (CS(i)) y una cota inferior (CI(i)) del beneficio (o coste) óptimo que podamos alcanzar a partir de ese nodo. Estas cotas son la esencia del método porque determinan cuándo se puede realizar una poda desde ese nodo.
- Una estimación del beneficio (o coste) óptimo que se puede encontrar a partir de ese nodo. Evidentemente, y por razones de efectividad, esa estimación perfectamente podrá ser una media de las cotas anteriores. La función de este valor es ayudar a decidir qué parte del árbol evaluar primero.

Así mismo, y como su propio nombre indica, asociado al método Branch and Bound dispondremos de una Estrategia de Poda. En efecto, supongamos sin pérdida de generalidad que estamos considerando un problema de maximización, y que hemos recorrido varios nodos del conjunto {1, 2, ...n}, estimando para cada uno de ellos la cota superior CS (j) e inferior CI (j), respectivamente, para $j \in \{1, 2, ...n\}$. En estas circunstancias distinguiremos dos casos:

1. Si a partir de un nodo siempre podemos obtener alguna solución válida, entonces podar un nodo i si:

$$CS(i) \leq CI(j)$$

para algún nodo j generado.

Esto es lo que ocurre por ejemplo en el problema de la mochila, utilizando un árbol binario: A partir de a, podemos encontrar un beneficio máximo de CS(a) = 4. A partir de b, tenemos garantizado un beneficio mínimo de CI(b) = 5. Entonces podemos podar el nodo a, sin perder ninguna solución óptima.

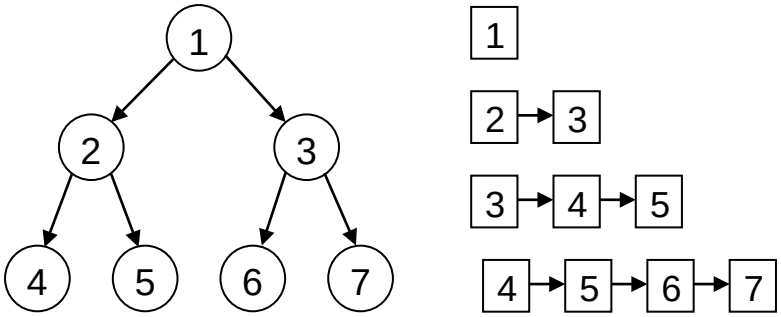
2. Si a partir de un nodo puede que no lleguemos a una solución válida, entonces podemos podar un nodo i si:

$$CS(i) \leq Beneficio(j)$$

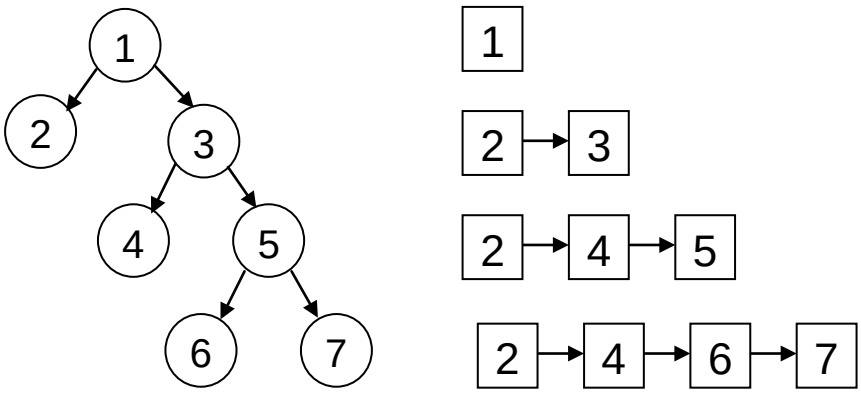
para algún nodo j, solución final (factible).

Por ejemplo en el problema de las reinas, a partir de una solución parcial, no está garantizado que exista una solución

En lo que concierne a la forma de ramificar, la denominada Estrategia de Ramificación, tendremos en cuenta que normalmente el árbol de soluciones es implícito, de forma que no tendremos que almacenarlo en ningún lugar. Desde este punto de vista, para hacer el recorrido se utiliza una lista de nodos vivos (LNV), que contiene todos los nodos que han sido generados pero que no han sido explorados todavía, es decir, está constituida por todos los nodos pendientes de ser considerados por el algoritmo. Según cómo sea la lista, el recorrido será de uno u otro tipo, y así tendremos una Estrategia FIFO y una Estrategia LIFO. Con la primera, FIFO, la LNV es una cola y el recorrido es en anchura, como describe la siguiente figura,



Con la segunda estrategia de ramificación, LIFO, la lista de nodos vivos es una pila y el recorrido es en profundidad, como la siguiente figura ilustra



Pero, sin tener en cuenta nada más, hay que destacar que las estrategias FIFO y LIFO realizarían una búsqueda "a ciegas", sin considerar los beneficios. Por tanto usando la estimación del beneficio, será mejor buscar primero por los nodos con mayor valor estimado. Con esto aparece una nueva y, para nuestros propósitos aquí, definitiva estrategia que se denomina Estrategias LC ("Least Cost"), que consiste en lo siguiente: entre todos los nodos de la lista de nodos vivos, elegir el que tenga mayor beneficio (o menor coste) para explorar a continuación. En caso de empate (de beneficio o coste estimado) deshacerlo usando un criterio FIFO o LIFO. De este modo nos quedan dos estrategias LC, a su vez:

- Estrategia LC-FIFO: Seleccionar de la LNV el nodo que tenga mayor beneficio y en caso de empate escoger el primero que se introdujo (de los que empatan).
- Estrategia LC-LIFO: Seleccionar de la LNV el nodo que tenga mayor beneficio y en caso de empate escoger el último que se introdujo (de los que empatan).

Cuando hablemos de búsqueda LC, muchas veces haremos referencia a una función de costo c(X) definida como sigue: Si X es un nodo respuesta entonces c(X) es el costo (beneficio estimado, nivel, dificultad computacional, etc.) de alcanzar X desde la raíz del árbol de estados. Si X no es un nodo respuesta entonces $c(X) = \infty$ si el subarbol X no contiene nodos respuestas, en otro caso c(X) es igual al costo de un nodo respuesta de mínimo costo en el subarbol X.

Como en cada nodo podremos tener una cota inferior de coste, un coste estimado y una cota superior del coste, el método Branch and Bound queda conformado podando según los valores de las cotas, y ramificando según los costes estimados.

Nótese que sólo se comprueba el criterio de poda cuando se introduce o se saca un nodo de la LNV. Por otro lado, si un descendiente de un nodo es una solución final entonces no se introduce en la LNV. Se comprueba si esa solución es mejor que la actual, y se actualiza C y el valor de la mejor solución óptima de forma adecuada.

De esta forma, si suponemos un problema de minimización, y que nos movemos en el caso en que existe solución a partir de cualquier nodo, un esquema algorítmico general para el método Branch and Bound, es el siguiente

```
BRANCHANDBOUND (NodoRaiz: tipo_nodo; var s: tipo_solucion)
Comienzo
   LNV = {NodoRaiz}
   C = CS (NodoRaiz)
   s = ∅
   Mientras LNV ≠ ∅ hacer
         x = Seleccionar (LNV);  {con criterio FIFO, LIFO, LC-FIFO o LC-LIFO }
         LNV = LNV - {x}
         Si CI (x) < C entonces                {Si no se cumple se poda x}
              Para cada y hijo de x hacer
                   Si y es una solución final mejor que s entonces
                        s = y
                        C = min (C, Coste (y))
                   Sino si y no es solución final y (CI(y) < C) entonces
                        LNV = LNV + {y}
                        C:= min (C, CS (y) )
Fin
```

Igual que en el caso de los algoritmos Backtracking, poco podemos decir en este caso de la eficiencia de los métodos Branch and Bound, salvo que básicamente depende de dos factores:

a) del número de nodos recorridos, y por tanto de la efectividad de la poda, y
b) del tiempo gastado en cada nodo, y por consiguiente del tiempo de hacer las estimaciones de coste y del tiempo de manejo de LNV.

Aunque en el peor caso, el tiempo suele ser igual que el de un algoritmo Backtracking (o peor si tenemos en cuenta el tiempo que requiere la LNV), en el caso promedio se suelen obtener mejoras respecto al backtracking.

Pensando en hacer que un algoritmo Branch and Bound sea más eficiente, las direcciones en las que podemos actuar son dos. La primera hacer estimaciones de costo muy precisas: Se realiza una poda exhaustiva del árbol, con lo que se recorren menos nodos, pero se gasta mucho tiempo en realizar las estimaciones. La segunda hacer estimaciones de costo poco precisas, con lo que se gasta poco tiempo en cada nodo, pero el número de nodos puede ser muy elevado, y por tanto no se hace mucha poda. Consiguientemente se debe buscar un equilibrio entre la exactitud de las cotas y el tiempo de calcularlas.

Desarrollaremos en lo que sigue esta técnica sobre algunos problemas que ya conocemos bien, como son el Problema de la Mochila 0-1 y el del Viajante de Comercio.

El Problema de la Mochila reconsiderado

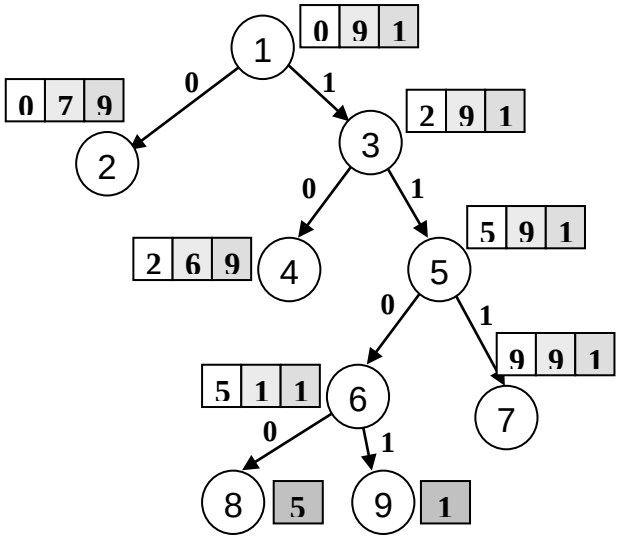
Puesto que es un problema bien conocido, no recordaremos su formulación y, directamente, comenzaremos dando el diseño del algoritmo Branch and Bound, es decir, teniendo en cuenta los tres siguientes aspectos claves:

1. Definir una representación de la solución, es decir, a partir de un nodo, cómo se obtienen sus descendientes.
2. Dar una manera de calcular el valor de las cotas y la estimación del beneficio, y
3. Definir la estrategia de ramificación y de poda.

Respecto a la representación de la solución, tenemos dos opciones. La primera es suponer un árbol binario: $(s_1, s_2, \ldots, s_n)$, con $s_i = (0, 1)$. Entonces, los hijos de un nodo $(s_1, s_2, \ldots, s_k)$ los representaremos por $(s_1, s_2, \ldots, s_k, 0)$ y $(s_1, s_2, \ldots, s_k, 1)$. La segunda hacerla mediante un árbol combinatorio: $(s_1, s_2, \ldots, s_m)$
donde $m \leq n$ y $s_i \in \{1, 2, ..., n\}$. En este caso, los hijos de un nodo $(s_1, s_2, \ldots, s_k)$ los representaremos por $(s_1, s_2, \ldots, s_k, s_{k+1})$, $(s_1, s_2, \ldots, s_k, s_{k+2})$, …, $(s_1, s_2, \ldots, s_k, s_n)$.

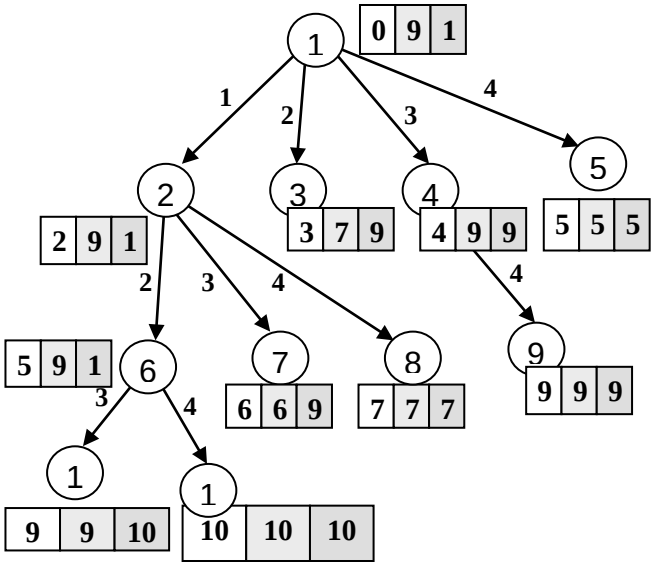
En lo que concierne al Cálculo de Cotas, la cota inferior podría ser el beneficio que se obtendría incluyendo sólo los objetos incluidos hasta ese nodo. La estimación del beneficio podríamos calcularla sumándole a la solución actual el beneficio de incluir los objetos enteros que quepan, utilizando por ejemplo un método greedy, para lo que supondremos que los objetos están ordenados por orden decreciente de vi/wi. Finalmente, la cota superior se determinaría de resolver el problema de la mochila continuo a partir de ese nodo (con un algoritmo greedy), y quedarse con la parte entera.

En último lugar, la Forma de realizar la poda puede ser la siguiente: En una variable C guardar el valor de la mayor cota inferior hasta ese momento dado. Si para un nodo, su cota superior es menor o igual que C entonces se puede podar ese nodo. Finalmente podemos definir una Estrategia de ramificación del siguiente modo: Puesto que tenemos una estimación del coste, usaremos una estrategia LC, según la cual exploraremos primero las ramas con mayor valor esperado (MB), y sobre si emplear LC-FIFO o LC-LIFO, usaremos la LC-LIFO, y en caso de empate seguiremos por la rama más profunda. (MB-LIFO).
Como ilustración, consideremos el siguiente caso del Problema de la Mochila 0-1: n = 4, M = 7, p = (2, 3, 4, 5) y w = (1, 2, 3, 4). Entonces la aplicación del método Branch and Bound recién descrito nos lleva al siguiente desarrollo:



C	LNV
0	**1**
2	**3** → 2
5	**5** → 2 → 4
9	**6** → 7 → 2 → 4
10	**7** → 2 → 4
10	**2** → 4
10	**4**

Sobre el mismo caso, utilizando un árbol combinatorio y LC-FIFO, la técnica Branch and Bound se desarrollaría conforme describe la siguiente figura,



C	LNV
0	**1**
5	**2** → 4 → 3
7	**4** → 6 → 3 → 7
9	**6** → 3 → 7
10	**3** → 7
10	**7**

Problema del Viajante de Comercio

A fin de no confundirnos con la notación que vamos a emplear, recordemos que en este problema partimos de un grafo orientado G = (V,A), siendo V = {1,2,…,n}, A ={(o_i, d_i)}, $o_i \in V$, $d_i \in V$, y

$D[o_i, d_i]$ la distancia de la arista. Llamamos hamiltoniano a un conjunto de n arcos tal que para cada vértice i, $0 < i < n+1$, debe haber en ese conjunto exactamente un arco de la forma (i,j) y uno de la forma (k,i).
Con esto las soluciones factibles están definidas por

$$\{X = ()| \forall i\ x_i \in \{0,1\}, \Sigma x_i = n, \forall u \in V\ ((\exists !i\ (x_i = 1 \text{ y } o_i = u) \wedge (\exists !j\ (x_j = 1 \text{ y } d_j = u))\ \}$$

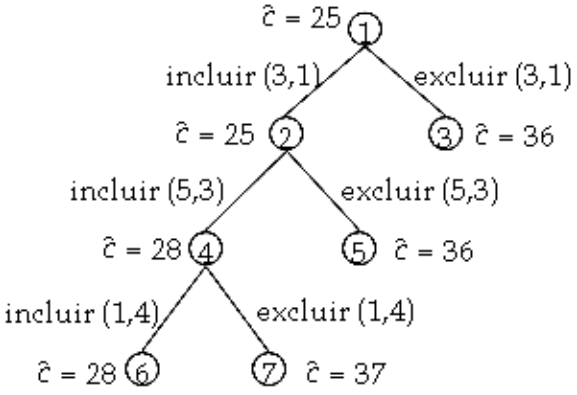
y la función objetivo está dada por

$$F(X) = \Sigma x_i\ D[o_i, d_i]$$

Como árbol de búsqueda tomamos un árbol binario tal que un hijo izquierdo representa la inclusión de un determinado arco en el hamiltoniano, mientras que su hermano derecho representa la exclusión de ese arco. Si se elige, para empezar, el arco (i,j): el subárbol izquierdo representa todos los recorridos que incluyen el arco (i,j), y el subárbol derecho los recorridos que no lo incluyen. Si hay un recorrido óptimo incluido en el subárbol izquierdo, entonces sólo faltan por seleccionar n-1 arcos para encontrarlo, mientras que si todos están en el derecho, hay que seleccionar todavía n arcos.
Por ejemplo en el grafo de matriz

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

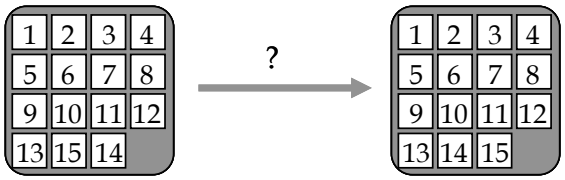
como ilustra la figura, se llega al nodo 6.



Se han elegido ya tres arcos: (3,1), (5,3), (1,4). Para los dos restantes, sólo queda ya una opción: (4,2) y (2,5). Así, se obtiene el recorrido: 5,3,1,4,2,5, con una distancia total de 28. El siguiente nodo en curso es el 3, con $\hat{c}(3) = 36$, valor mayor que la cota superior en curso, y por tanto el algoritmo acaba.

El problema del 15-puzzle.

El 15-puzzle (inventado por Sam Loyd en 1878) está definido por 15 fichas numeradas en un cuadrado con una capacidad para 16 fichas, como por ejemplo muestra la figura,



El objetivo

Nos dan una disposición inicial de las fichas y el objetivo es transformar esta disposición en la disposición deseada de la figura a través de una serie de movimientos legales.

Los únicos movimientos legales son aquellos en los que se mueve una ficha adyacente al hueco vacío (HV). Así desde la disposición inicial de la figura, son posibles dos movimientos. Podemos mover hacia el HV cualquiera de las fichas numeradas 12 o 14. Cada movimiento crea una nueva configuración de las fichas. Estas disposiciones se llaman estados del puzle. Las disposiciones inicial y deseada se llaman estados inicial y deseado. Un estado es alcanzable desde el estado inicial si y solo si existe una sucesión de movimientos legales desde el estado inicial hasta ese estado. El espacio de estados de un estado inicial consiste en todos los estados que se pueden alcanzar desde ese estado inicial. El modo más directo de resolver el puzle seria buscar el espacio de estados del estado deseado y usar el camino desde el estado inicial al deseado como respuesta. Es fácil constatar que hay 16! (16! = 20.9x10) diferentes disposiciones de las fichas en el tablero. De estas, solo la mitad son alcanzables desde el estado inicial dado.

Antes de buscar en este espacio de estados el estado deseado, sería interesante saber si ese estado es alcanzable desde el estado inicial, y hay una forma sencilla de hacerlo. Numeremos las posiciones del tablero del 1 al 16. La posición i es la casilla que contiene la ficha numerada i en el estado deseado de la siguiente figura.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Configuración Deseada

Así la posición 16 es el HV. Sea POSICION(i) el número de posición de la ficha numerada i en el estado inicial. POSICION(16) notara la posición del hueco vacío.

Sea, para cualquier estado, MENOS(i) el número de fichas j tales que j < i y POSICION(j) > POSICION(i). Para el estado de la siguiente configuración

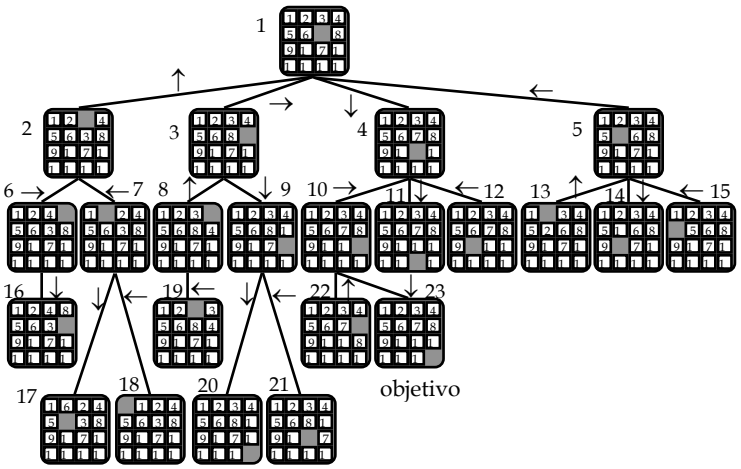
1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

tenemos que MENOS(1) = 0, MENOS(4) = 1 y MENOS(12) = 6. Sea X = 1 si en el estado inicial el HV está en una de las posiciones marcadas, y X = 0 si es una de las restantes posiciones. Se tiene entonces el siguiente resultado: El estado deseado de la Configuración Objetivo es alcanzable desde el estado inicial si y solo si

$$\sum_{i=1..16} \text{MENOS}(i) + X$$

es par.

Este resultado puede usarse para saber si el estado deseado está en el espacio de estados del estado inicial o no. Si lo está, podemos proceder a determinar una sucesión de movimientos encaminados hacia el estado deseado. Para efectuar esta búsqueda el espacio de estados puede organizarse en forma de árbol, en el que los hijos de un nodo X representan los estados alcanzables desde el estado X mediante un movimiento legal. Es conveniente pensar en un movimiento de modo que realmente suponga un movimiento del espacio vacío más que un movimiento de la ficha. En cada movimiento el espacio vacío se mueve arriba, abajo, a la derecha o a la izquierda. La siguiente figura muestra los tres primeros movimientos en el árbol de espacios de estado del 15-Puzzle comenzando con el estado inicial que se muestra en la raíz



Por la solución de este juego, que como hemos dicho fue publicado en un periódico de Nueva York en 1878, se ofrecieron 1000 dólares norteamericanos. Aún hoy sigue siendo un banco de pruebas para los algoritmos de búsqueda, y constituye un magnifico puente hacia las más recientes técnicas asociadas a los Sistemas Inteligentes, por lo que se le propone al lector interesado como ejercicio de reflexión y búsqueda bibliográfica sobre el mismo.