



5. Backtracking

5.1 Introducción

La técnica backtracking o vuelta atrás es un método basado en la ramificación, con objetivo de realizar una búsqueda exhaustiva y sistemática del espacio de soluciones. Es una técnica general de resolución de problemas que se puede aplicar en problemas de optimización, juegos, decisión, estrategias, etc. La técnica backtracking siempre encuentra la solución óptima, y si no optimiza la búsqueda de la solución óptima es igual al método basado en explorar todo el espacio de soluciones (fuerza bruta).

Por lo tanto backtracking es un técnica por lo general poco eficiente en tiempo computacional. Y en este sentido se puede considerar una técnica opuesta a los algoritmos voraces, o de avance rápido. Hay que recordad que los algoritmos voraces no deshacen ninguna de las decisiones tomadas sobre cada uno de los candidatos, así por ejemplo si es un problema de decisión de coger o no un elemento una vez considerado no se vuelve a considerar en el futuro. Por contra backtracking en el caso más extremo reconsidera una y otra vez el mismo elemento hasta agotar todas las posibilidades con ese elemento.

5.2 Método General

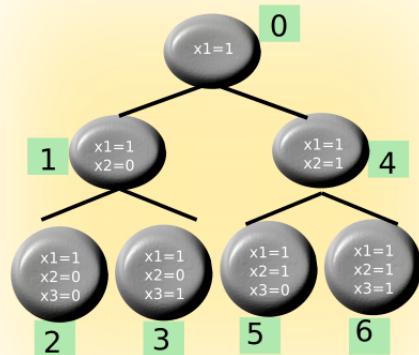
El método obtendrá una solución $S = (x_1, x_2, \dots, x_n)$ que cumple algunas restricciones y/o optimiza alguna función objetivo. En cada iteración el algoritmo se encontrará en cierto nivel k (rellenando el valor x_k en S) y teniendo así una solución parcial (x_1, x_2, \dots, x_k) (aún no se han tomado valores para (x_{k+1}, \dots, x_n)). Desde este nivel k las posibilidades que tenemos son:

1. Si se puede añadir otro elemento a la solución x_{k+1} se añade y se avanza al nivel $k + 1$
2. Si no se puede añadir se prueba con otros valores para x_k .
3. Si no existe ningún valor posible para probar, entonces se retrocede al nivel $k - 1$.
4. Se sigue hasta que la solución parcial sea una solución completa del problema o hasta que no haya más posibilidades.

Ejemplo 5.2.1

Veamos la forma de actuar de backtracking en el problema de la Mochila. Supongamos que tenemos una mochila con $M=6$, $p = (3, 2, 3)$ y $b = (4, 2, 1)$. El nivel en el que estamos es $k=1$ y tenemos la solución parcial ($x_1 = 1$).

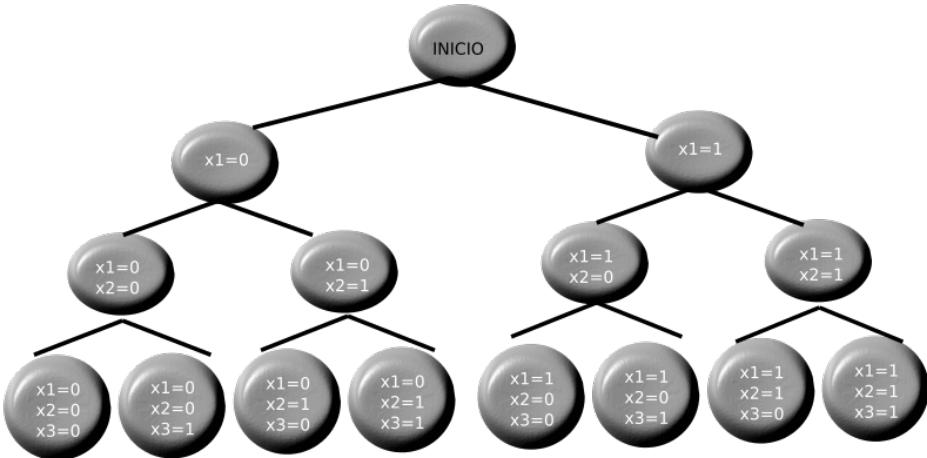
1. Como nos queda 3kg de mochila tras haber añadido el elemento 1 se pasa a evaluar el nivel 2, es decir el objeto 2. Por ejemplo podemos decidir no cogerlo $x_2 = 0$
2. Se avanza al nivel 3, para evaluar el objeto 3. Y se puede decidir no cogerlo. Y así obteniendo la solución $S = (1, 0, 0)$.
3. Como en el nivel 3 nos queda la posibilidad de coger el objeto 3. Obtenemos otra solución $S = (1, 0, 1)$.
4. Ya hemos agotado todas las posibilidades del objeto 3 hacemos backtracking y volver al nivel 2, a ver si nos queda más posibilidades. Como es así generamos la solución parcial ($x_1 = 1, x_2 = 1$). Y avanzamos al nivel 3.
5. Desde el nivel 3 podemos generar la solución $(x_1 = 1, x_2 = 1, x_3 = 0)$. La otra posibilidad para el objeto 3 no es válida ya que supera la capacidad de la mochila.



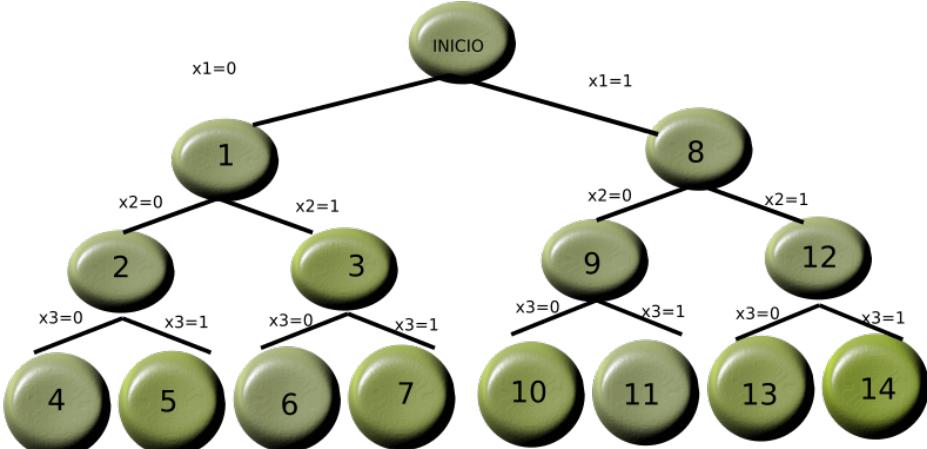
□

5.3 Espacio de Soluciones

En el ejemplo 5.2.1 hemos dado una introducción intuitiva del árbol de estado o espacio de soluciones que representa la forma de actuar del algoritmo backtracking. Por ejemplo si nuestro vector solución S se compone de tres variables x_i que puede adoptar valores 0 o 1, el espacio de soluciones o árbol de estados sería el siguiente:



Y una representación del mismo árbol, que también indica el orden en el que se analiza las soluciones parciales y totales:



Así por ejemplo el nodo 1 indica que la solución parcial ($x_1 = 0$) es la primera que se genera. A continuación el nodo 2 que se corresponden con la solución parcial ($x_1 = 0, x_2 = 0$), y así sucesivamente.

Con respecto al árbol de soluciones que explora la técnica backtracking hay que tener en cuenta que:

- El árbol de estados es una forma de representar la ejecución del algoritmo
- Es implícito y no se almacena. Esto quiere decir que no vamos a tener en memoria el árbol de soluciones.
- En la técnica backtracking el recorrido del árbol de soluciones subyacente es primero en profundidad y de izquierda a derecha. Esto quiere decir que primero se analiza los nodos hijos antes que los hermanos a un nodo.

Cuando nos enfrentamos a un problema que queremos resolver con la técnica backtracking tenemos que contestar a las siguientes preguntas:

1. **¿Cómo es la forma del árbol?**

2. ¿Qué significa el valor de la tupla solución?
3. ¿Cómo es la representación de la solución?

5.4 Tipos comunes de árboles backtracking

Con respecto a cual es la forma del árbol nos planteamos cuantos hijos tiene cada nodo. Desde este punto de vista tenemos:

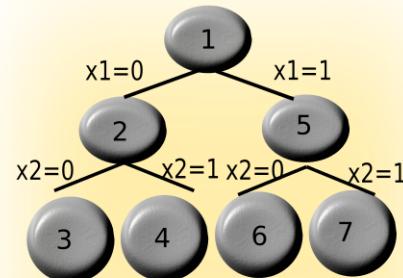
- Árboles binarios
- Arboles n-arios
- Arboles Permutacionales
- Arboles Combinatorios

5.4.1 Árboles Binarios

Se caracterizan porque el vector solución $S = (x_1, x_2, \dots, x_n)$ cada x_i adopta un valor 0 o 1. n es el número de niveles que se corresponde con el número de variables del vector solución.

Ejemplo 5.4.1

El siguiente árbol representa el árbol de estados binario para $n=2$



□

Los problemas que se adaptan a estos tipos de árboles son aquellos que deben elegir entre ciertos elementos de un conjunto sin importar el orden de los elementos. De forma que a cada elemento i del conjunto se le asocia un x_i indicando con $x_i = 1$ si se coge o con $x_i = 0$ que no se coge para construir la solución.

El número máximo de soluciones parciales sería $\sum_{i=0}^n 2^i = 2^{n+1} - 1$. Ese número se corresponden con todos los nodos del árbol de soluciones. Por otro lado, las soluciones totales son las hojas y el número que tenemos es 2^n .

Ejemplos típicos de Problemas que se adaptan.

- El problema de la mochila (0|1).

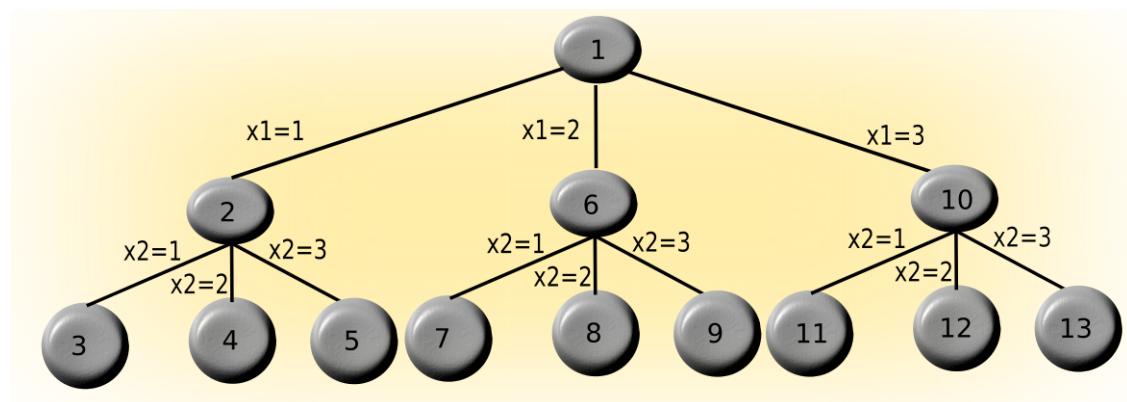
- El problema de encontrar un subconjunto de elementos de un conjunto que sume exactamente una cantidad.

5.4.2 Árboles k-arios

Se caracterizan porque el vector solución $S = (x_1, x_2, \dots, x_n)$ cada $x_i \in \{1 \dots, k\}$. n es el número de niveles que se corresponde con números de variables del vector solución.

Ejemplo 5.4.2

El siguiente árbol de estados es un árbol 3-ario, con dos variable ($n=2$). Cada variable por lo tanto puede adoptar los valores $\{1, 2, 3\}$.



□

El número de soluciones parciales (número de nodos) de este tipo de árbol viene dado por la ecuación:

$$\sum_{i=0}^n k^i = \frac{k^{n+1} - 1}{k - 1}$$

Siendo en este caso el número de soluciones (las hojas) k^n .

Ejemplos típicos de Problemas que se adaptan.-

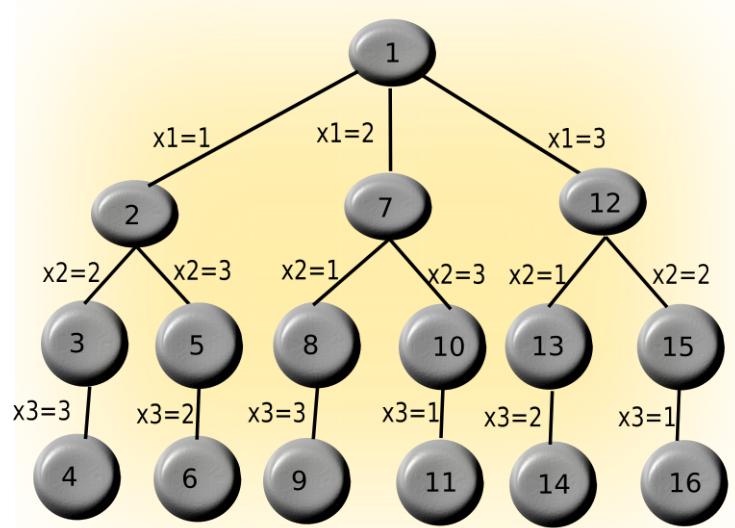
- El problema del cambio de monedas
- El problema de las n-reinas. Este problema consiste en dado un tablero de ajedrez de dimensiones $n \times n$, el objetivo es colocar cada reina o dama en una casilla sin atacar y ser atacada por otra dama. En este caso para n-reinas el numero de soluciones parciales sería n^n .

5.4.3 Árboles Permutacionales

En este caso el vector solución $S = (x_1, x_2, \dots, x_n)$ se caracteriza porque cada $x_i \in \{1, 2, \dots, n\}$ y $x_i \neq x_j$. Este árbol es adecuado en aquellos problemas en los que los valores de x_i no se puede repetir.

Ejemplo 5.4.3

El siguiente árbol de estados es un árbol permutacional, con tres variable ($n=3$). Cada variable por lo tanto puede adoptar los valores $\{1, 2, 3\}$. Pero en el mismo camino desde el nodo 1 hasta una hoja los valores de x_i tienen que ser diferentes.



□

El número de soluciones parciales o nodos son:

$$\sum_{i=0}^n \frac{n!}{n-i!} = 1 + n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \dots + n!$$

El número de soluciones totales son $n!$.

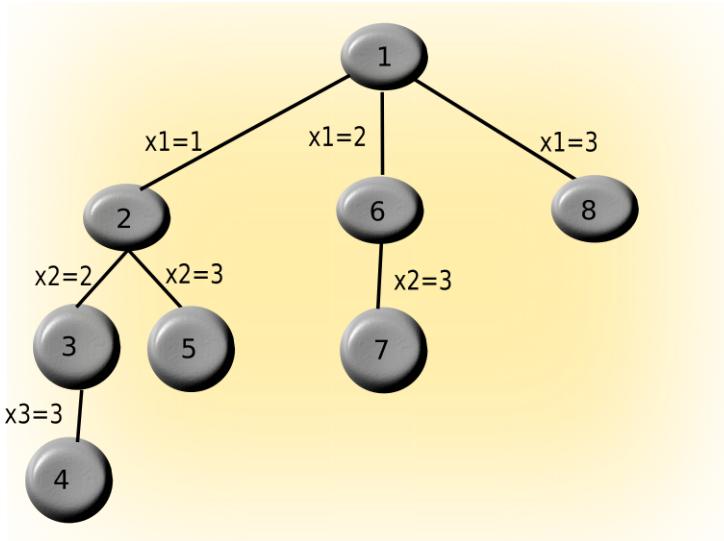
Un problema típico donde se adapta un árbol permutacional es el problema de asignar n trabajos a n trabajadores.

5.4.4 Árboles Combinatorios

Dado el vector solución $S = (x_1, x_2, \dots, x_m)$ con $m \leq n$, lo que quiere decir que tenemos un número de variables menor o igual al número de niveles del árbol. Además los valores que adoptan las variables cumplen que $x_i \in \{1, 2, \dots, n\}$ y $x_i < x_{i+1}$.

Ejemplo 5.4.4

El siguiente árbol de estados es un árbol combinatorio, con tres variable ($n=3$). En este ejemplo los valores que adoptan las variables son $\{1, 2, 3\}$. Pero en el mismo camino que conduce desde la raíz hasta una hoja se debe de dar que $x_i < x_j$ siendo $i < j$.



□

En estos árboles todos los nodos representan una solución total. Así por ejemplo en el problema de la mochila (0|1) se puede usar un árbol combinatorio. Ahora x_i representa que objeto entre los n se coge.

El número de soluciones totales en este caso son:

$$\sum_{i=1}^n \binom{n}{i} = 2^n$$

Si se cuenta la raíz a este número de sumamos 1.

Si las soluciones están solamente en las hojas entonces el número de soluciones es:

$$\sum_{i=1}^n \binom{n}{i} - \sum_{i=1}^n \binom{n-i}{i}$$

En general, cualquier tipo de problema que se adapte a un árbol de estados binario se puede adaptar también a un árbol de estados combinatorio.

5.5 Cuestiones a resolver para realizar el diseño

Para poder plantear una solución basada en un algoritmo backtracking debemos antes responder a las siguientes preguntas:

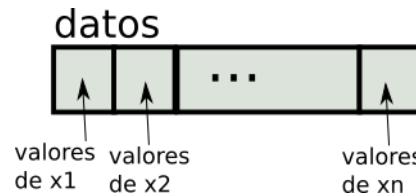
1. *¿Qué tipo de árbol es el adecuado para el árbol de soluciones?.*-Para responder a esta pregunta debemos plantear lo siguiente
 - Cómo se representa la solución. Por ejemplo un vector de n componentes
 - Cómo es la tupla solución en el sentido de que significa cada x_i y valores que puede adoptar. Por ejemplo cada elemento indica si se coge el elemento o no y por lo tanto cada $x_i \in 0, 1$.

2. *¿Cómo generar un recorrido según ese árbol?* En este sentido como añadir nuevos elementos a la solución y como eliminarlos para recorrer todos los posibles caminos del árbol implícito.
 - Se debe especificar como generar un nuevo nivel. Este es el proceso en el que la solución parcial crece.
 - Se debe especificar como generar los hermanos a un nivel. Por ejemplo como un x_i adopta todo el rango de valores
 - Retroceder en el árbol. Como decrece la solución parcial.
3. *¿Qué ramas se pueden descartar por no conducir a soluciones del problema?* .- Esto es lo que se conoce como backtracking (vuelta atrás). Esto puede ocurrir en general por dos cosas:
 - No se continua por ese camino y se poda porque la solución parcial no cumple las restricciones
 - Seguir por ese camino no nos lleva a algo mejor ya logrado, y por lo tanto se poda el camino.

5.6 Backtracking con A. Binario y A.k-arios

Si nuestra solución S tiene la forma $S = (x_1, x_2, \dots, x_n)$ la forma más directa de implementar la solución es con un vector tal como:

El vector datos de tipo entero, tiene que tener n posiciones de forma que $datos[i]$ hace referencia al valor de x_i :



Supongamos que nuestro árbol adopta valores 0,1 es decir es binario, por lo tanto $k = 1$ (máximo valor que adopta cada $datos[i]$) y $valor_inicial = -1$. Si tenemos tres variables datos se inicia con $datos = \{-1, -1, -1\}$ y $level=0$. El constructor sería:

```

1 Back_Anario::Back_Anario(unsigned int n,unsigned int kk,int vi){
2
3     valor_inicial = vi; //valor inicial para cada dato[i]
4     k=kk; //maximo valor para dato[i]
5     datos= vector<int >(n,valor_inicial);
6     level=0;
7     datos[level]++; //iniciamos el primero
8 }
```

Dejando datos con los valores 0,-1,-1 y level =0, en nuestro ejemplo. Ahora queremos obtener la siguiente secuencia que sería {0,0,−1}. Para obtenerla ejecutamos el método GeneraSiguiente:

```

1  bool Back_Anario::GeneraSiguiente(){
2
3      int nivel;
4      if (level==(int)datos.size()-1)
5          nivel=level;
6      else //avanzamos en profundidad
7          nivel = level+1;
8
9      //tenemos mas posibilidades para datos[nivel]
10     while (nivel>=0 && !MasHermanos(nivel) ){
11         //si no retrocedemos en nivel
12         datos[nivel]=valor_inicial;
13         nivel=nivel-1;
14     }
15     level=nivel;
16     //hemos obtenido todas las posibilidades
17     if (nivel<0) return false;
18     do{
19         //pasamos al siguiente valor en datos[nivel]
20         GenerarSiguiente(nivel);
21
22         //Si es una solucion parcial valida
23         if (EsSecuencia(nivel)){
24             level=nivel;
25             return true;
26         }
27         //podemos avanzar al siguiente nivel
28         if ( PosibleSecuencia(nivel))
29             level++;
30
31         else{
32             //miramos si hay mas posibilidades para datos[nivel]
33             while (nivel>=0 && !MasHermanos(nivel) ){
34                 //si no retrocedemos en nivel
35                 datos[nivel]=valor_inicial;
36                 nivel=nivel-1;
37             }
38         }
39     }
40

```

```

41     }while (nivel>=0 );
42     level = nivel;
43     return false;
44 }
```

En la sentencia 4 vemos si hemos llegado a una hoja, si es así no avanzamos level. En otro caso avanzamos en profundidad con la sentencia

```
nivel=level+1
```

A continuación el while en la sentencia 10 comprueba si existen más valores posibles para *datos[nivel]*. Esto se comprueba en la función *MasHermanos*:

```

bool Back_Anario::MasHermanos( int nivel){
    //si datos[nivel]<k hay mas valores para datos[nivel]
    return nivel>=0 && datos[nivel]<k;
}
```

Así si *MasHermanos* devuelve true significa que tenemos mas valores que probar para *datos[nivel]*. En ese caso en la línea 18 entramos en un bucle que en primer lugar genera un siguiente valor para *datos[nivel]*:

```

void Back_Anario::GenerarSiguiente(int nivel){
    datos[nivel]=datos[nivel]+1;
}
```

Ahora en la linea 23 se comprueba si la secuencia en datos es una solución válida mediante la función *EsSecuencia*:

```

bool Back_Anario::EsSecuencia( int nivel){
    //podemos exigir llegar a las hojas
    //entonces nivel==datos.size()-1
    return nivel<=(int)datos.size()-1 ;
}
```

Dependiendo del problema *EsSecuencia* puede ser true cuando el nivel está comprendido entre 0 y n-1 (n el numero de variables o dimensión de datos). Pero existen problemas que solamente nos interesan las soluciones totales es decir las hojas. Entonces EsSecuencia se podría implementar como:

```

bool Back_Anario::EsSecuencia( int nivel){
    //podemos exigir llegar a las hojas
    //entonces nivel==datos.size()-1
    return nivel==(int)datos.size()-1 ;
}
```

En este casi tenemos que comprobar que es un solución parcial válida con *PossibleSecuencia*

```

bool Back_Anario::PossibleSecuencia(int nivel){
    return  nivel<(int)datos.size()-1;
}
```

Si esta devuelve true entonces avanzamos level en otro caso retrocedemos level con el trozo de código:

```

while (nivel>=0&& !MasHermanos(nivel) ){
    datos[nivel]=valor_inicial;
    nivel=nivel-1;
}

```

Hacer vuelta atrás es muy parecido a el método *GeneraSiguiente*, la única diferencia es que en primer lugar no avanza level, sino busca otros valores posibles para datos[level] si no los hay retrocede level hasta encontrar un level tal que a datos[level] se le pueda asignar nuevos valores:

```

1  bool Back_Anario::Backtracking(){
2      int nivel;
3      if (level==0) return false;
4      else{
5          nivel = level;
6          while (nivel>=0 && !MasHermanos(nivel) ){
7              datos[nivel]=valor_inicial;
8              nivel=nivel-1;
9              level=level-1;
10         }
11         level = nivel;
12         if (nivel<0) return false;
13     }
14     do{
15         GenerarSiguiente(nivel); //ponemos el valor en el nivel
16         if (EsSecuencia(nivel)){ // es una posibilidad valida
17             level=nivel;
18             return true;
19         }
20         if ( PossibleSecuencia(nivel))
21             level++; // avanzamos en profundidad
22         else{
23             while (nivel>=0 && !MasHermanos(nivel) ){
24                 //no existen mas hermanos en ese nivel
25                 datos[nivel]=valor_inicial; //backtracking
26                 nivel=nivel-1;
27                 level=nivel;
28             }
29         }
30     }
31 }while (nivel>=0 );
32 return false;
33 }

```

5.6.1 Problema: La Mochila (0|1)

En esta sección usando *Back_Anario* vamos a dar un diseño backtracking al problema de la mochila 0|1. Los datos del problema son:

- n:numero de objetos
- M capacidad de la mochila
- $p = (p_1, p_2, \dots, p_n)$ pesos de los objetos
- $b = (b_1, b_2, \dots, b_n)$ beneficios de los objetos

El objetivo es:

$$\text{maximizar} \left(\sum_{i=1}^n x_i \cdot b_i \right) \text{ sujeto a } \sum_{i=1}^n x_i \cdot p_i \leq M$$

siendo $x_i \in \{0,1\}$ tal que es 0 si el objeto i no se coge o 1 si el objeto i se coge. En nuestra implementación x_i cuando no ha sido considerado adopta el valor -1. Veamos la implementación C++:

```

1  /**
2   * @brief Algoritmo de Mochila 0/1 aplicando Backtracking
3   * con poda solamente cuando se supera la capacidad de la mochila
4   * @param M: capacidad de la mochila
5   * @param objetos: todos los objetos.
6   * @param ab: la mejor secuencia de 0/1 con la mejor
7   *           solucion encontrada
8   * @return el mejor beneficio acumulado.
9  */
10 int Mochila_Backtracking(unsigned int M,const vector<objeto>&objetos,
11                           Back_Anario &ab){
12     Back_Anario P(objetos.size(),1,-1);
13     int best_beneficio=0;
14     bool seguir=true;
15     while (seguir){
16         int s = ObtainBeneficios(P,objetos);
17         unsigned int speso=ObtainPesos(P,objetos);
18         if (speso>M) //si superamos la capacidad
19             seguir = P.Backtracking();
20         else{
21             if (s>best_beneficio){
22                 best_beneficio=s;
23                 ab=P;
24             }
25             seguir = P.GeneraSiguiente();
26         }
27     }
28     return best_beneficio;
29 };

```

Ejemplo 5.6.1

Obtener el mejor beneficio para una mochila $M = 7$, $p = (1, 2, 3, 4)$ y $b = (2, 3, 4, 5)$.

La mejor solución que devuelve el algoritmo de backtracking es $B_T = 10$ con $S = (x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1)$. La siguiente tabla muestra los valores de las distintas variables en las iteraciones del while:

Iteración	$M = 7 \ p = (1, 2, 3, 4) \ b = (2, 3, 4, 5)$					
	Level	Peso	Beneficio	Best_Beneficio	Elección vector datos $x_1x_2x_3x_4$	Best Elección $x_1x_2x_3x_4$
0	0	0	0	0	0 -1 -1 -1	
1	1	0	0	0	0 0 -1 -1	
2	2	0	0	0	0 0 0 -1	
3	3	0	0	0	0 0 0 0	
4	3	4	5	0	0 0 0 1	
5	2	3	4	5	0 0 1 -1	0 0 0 1
6	3	3	4	5	0 0 1 0	0 0 0 1
7	3	7	9	5	0 0 1 1	0 0 0 1
8	1	2	3	9	0 1 -1 -1	0 0 1 1
9	2	2	3	9	0 1 0 -1	0 0 1 1
10	3	2	3	9	0 1 0 0	0 0 1 1
11	3	6	8	9	0 1 0 1	0 0 1 1
12	2	5	7	9	0 1 1 -1	0 0 1 1
13	3	5	7	9	0 1 1 0	0 0 1 1
BACKTRACKING 14	3	9	12	9	0 1 1 1	0 0 1 1
15	0	1	2	9	1 -1 -1 -1	0 0 1 1
16	1	1	2	9	1 0 -1 -1	0 0 1 1
17	2	1	2	9	1 0 0 -1	0 0 1 1
18	3	1	2	9	1 0 0 0	0 0 1 1
19	3	5	7	9	1 0 0 1	0 0 1 1
20	2	4	6	9	1 0 1 -1	0 0 1 1
21	3	4	6	9	1 0 1 0	0 0 1 1
BACKTRACKING 22	3	8	11	9	1 0 1 1	0 0 1 1
23	1	3	5	9	1 1 -1 -1	0 0 1 1
24	2	3	5	9	1 1 0 -1	0 0 1 1
25	3	3	5	9	1 1 0 0	0 0 1 1
26	3	7	10	9	1 1 0 1	0 0 1 1
27	2	6	9	10	1 1 1 -1	1 1 0 1
28	3	6	9	10	1 1 1 0	1 1 0 1
BACKTRACKING 29	3	10	14	10	1 1 1 1	1 1 0 1

Como se puede observar se han realizado 30 iteraciones. En tres ocasiones se ha realizado vuelta atrás debido a que los elementos que seleccionaba en el estado superaba la capacidad de la mochila, esto para la iteración: 14, 22, y 29. \square

Consideraciones con Mochila (0|1).-

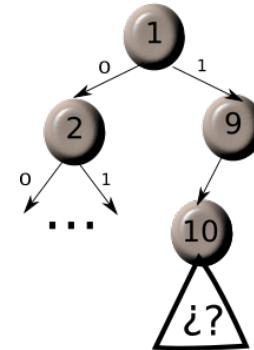
- El algoritmo es muy ineficiente en el sentido que puede llegar a un número de iteraciones $O(2^{n+1})$
- Solamente se hace backtracking cuando se supera la capacidad de la mochila.
- Tenemos que mejorar el algoritmo con una poda más inteligente, en el sentido de que cuando por un camino no vayas a obtener mejor beneficio que el actual no continuar.
- Para hacer esta poda más inteligente tenemos que hacer una estimación del beneficio total que se puede obtener por una rama y si este es menor que lo mejor obtenido podar, aunque esta sea una estimación

5.7 Estimación

Para poder mejorar nuestro esquema backtracking vamos a realizar una poda más eficiente. Para ello necesitamos una estimación de lo mejor (cota superior) o peor (cota inferior) que puedes llegar a obtener siguiendo por un camino del árbol de soluciones.

La estimación por lo tanto en el caso de que queramos obtener un beneficio máximo, es obtener una cota superior de lo mejor que podemos llegar a obtener, sin explorar el camino que nos queda. Por ejemplo en el problema de la mochila, la estimación del beneficio para el nivel y el nodo actual donde estamos sería:

$$b_{estimado} = b_{actual} + Estimacion(nivel + 1, n, M - pact)$$



Para no equivocarnos en hacer la estimación, tenemos que buscar una estimación que nos de una cota superior del valor real que podemos llegar a obtener si realizamos la exploración. De forma que si tenemos una solución actual con un beneficio mayor que el beneficio estimado (que es mayor o igual al real) del camino que aún no hemos explorado, entonces por ese camino no debemos continuar y realizamos una poda.

Por lo tanto nuestro objetivo será buscar una buena **Estimación**.

5.7.1 El problema de la Mochila (0|1). Estimación

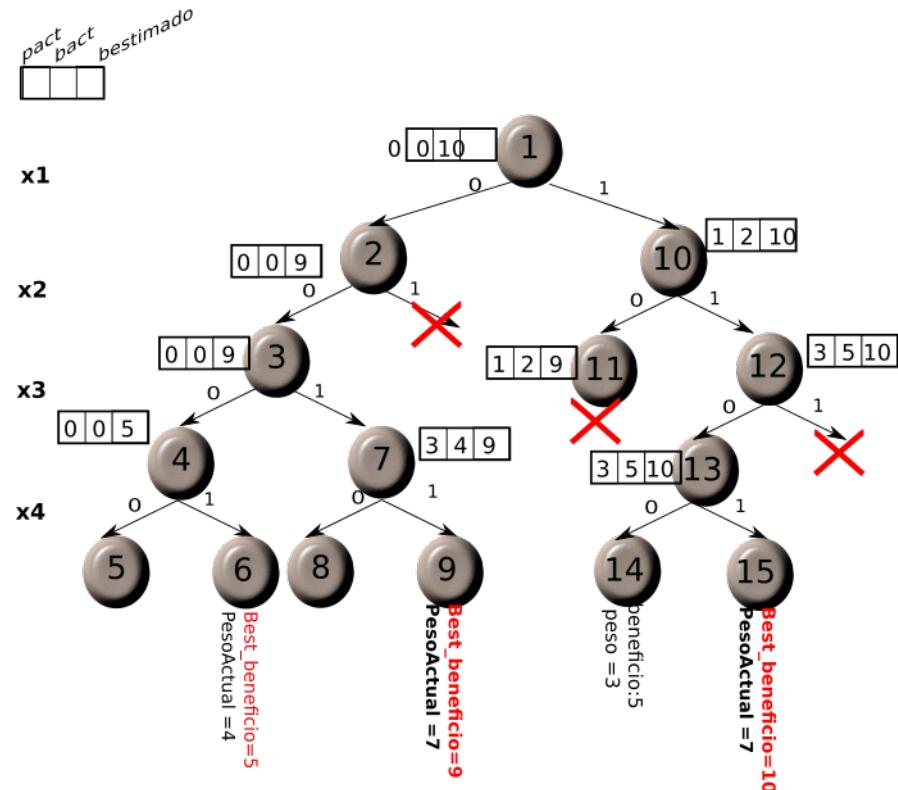
La función $Estimacion(k, n, Q)$ para el problema de la mochila nos da una estimación del beneficio que obtenemos para una mochila con los objetos $k \dots n$, con una capacidad Q . Si recordamos la mochila voraz no (0|1), es decir, permitía partir los objetos, daba una solución óptima. El valor que devuelve el algoritmo voraz podría ser una cota superior válida para el problema de la mochila (0|1). Así cuando $k=1$, tenemos en cuenta todos los objetos, la mochila voraz no (0|1) obtiene la solución óptima.

Ejemplo 5.7.1

Obtener el mejor beneficio para una mochila $M = 7$, $p = (1, 2, 3, 4)$ y $b = (2, 3, 4, 5)$, usando como estimación para podar el algoritmo voraz Mochila no (0|1).

En la siguiente imagen se puede observar que caminos se exploran. Para ello en cada nodo no hoja disponemos de una tripleta compuesta por: peso actual, beneficio actual, beneficio estimado.

Cuando el mejor beneficio es mayor que el beneficio estimado se poda la rama como es el caso del hijo del nodo 2 con valor de $x_2 = 1$.



Iteración	$M = 7 \ p = (1, 2, 3, 4) \ b = (2, 3, 4, 5)$						
	Level	Peso	Beneficio	BeneficioEstimado	BestBeneficio	Elección vector datos	Best Elección
Inicio	0	0	0	10	0	-1 -1 -1 -1	
0	0	0	0	9	0	0 -1 -1	
1	1	0	0	9	0	0 0 -1 -1	
2	2	0	0	5	0	0 0 0 -1	
3	3	0	0	0	0	0 0 0 0	
4	3	4	5	5	0	0 0 0 1	
5	2	3	4	9	5	0 0 1 -1	0 0 0 1
6	3	3	4	4	5	0 0 1 0	0 0 0 1
7	3	7	9	9	5	0 0 1 1	0 0 0 1
BACKTRACKING 8	1	2	3	9	9	0 1 -1 -1	0 0 1 1
9	0	1	2	10	9	1 -1 -1 -1	0 0 1 1
BACKTRACKING 10	1	1	2	9	9	1 0 -1 -1	0 0 1 1
11	1	3	5	10	9	1 1 -1 -1	0 0 1 1
12	2	3	5	10	9	1 1 0 -1	0 0 1 1
13	3	3	5	5	9	1 1 0 0	0 0 1 1
14	3	7	10	10	9	1 1 0 1	0 0 1 1
BACKTRACKING 15	2	6	9	10	10	1 1 1 -1	1 1 0 1

Como se puede observar el número de iteraciones o nodos generados en el árbol de estados se ha reducido considerablemente.

A continuación veamos el código C++:

```

1  /**
2   * @brief Algoritmo de Mochila 01 aplicando Backtracking con una
3   * estimacion para hacer mejor poda
4   * @param M: capacidad de la mochila
5   * @param objetos: todos los objetos. Debe estar ordenados
6   * por mayor razon beneficio peso
7   * @param ab: la mejor secuencia de 0 y 1 con la mejor solucion encontrada
8   * @return el mejor beneficio acumulado.
9  */
10 int Mochila_BacktrackingEstimacion(unsigned int M,
11         const vector<objeto>&objetos,Back_Anario &ab){
12     Back_Anario P(objetos.size(),1,-1);
13     int best_beneficio=0;
14     bool seguir=true;
15
16     while (seguir){
17
18         int bact = ObtainBeneficios(P,objetos);
19         unsigned int speso=ObtainPesos(P,objetos);
20         float be =bact+Voraz_Mochilano01(P.GetLevel()+1,M-speso,objetos);
21         if (speso>M) //superamos la capacidad de la mochila
22             seguir =P.Backtracking(); //poda
23         else{
24             if (bact>best_beneficio){//es mejor lo obtenido
25                 best_beneficio=bact;
26                 ab=P;
27                 seguir = P.GeneraSiguiente();
28             }
29             else{//hacemos una estimacion de
30                 //lo mejor que podemos llegar a
31                 //obtener por ese camino
32                 float bestimado=bact+
33                     Voraz_Mochilano01(P.GetLevel()+1,M-speso,objetos);
34                 //si es mejor que lo que tenemos
35                 if ((int)bestimado>best_beneficio)
36                     seguir = P.GeneraSiguiente();
37                 else//en otro caso se poda
38                     seguir = P.Backtracking();
39             }
40         }
41     }
42 }
```

```

43     return best_beneficio;
44 }

```

La implementación del algoritmo voraz Mochila no (0|1) es el siguiente:

```

1 /**
2 * @brief Implementa el algoritmo de mochila no01 usando voraces
3 * @param level: a partir de que objeto hay que tener en cuenta
4 *                 para aplicar el algoritmo
5 * @param M : capacidad de la mochila
6 * @param objetos: informacion de todos los objetos. Ordenados de
7 *                 mayor a menor razon beneficio peso
8 * @return el mejor beneficio obtenido.
9 */
10
11 float Voraz_Mochilano01(int level,unsigned int M,const vector<objeto>& objetos){
12     int n=objetos.size();
13     float bene_total=0.0;
14     unsigned long int pesoactual=0;
15     int i =level;
16
17     while (pesoactual <M && i<n){
18         //Seleccion del Objeto
19         objeto o=objetos[i];
20         //puede entrar en la mochila
21         if (pesoactual+o.peso<=M){
22             pesoactual= pesoactual+o.peso;
23             bene_total +=o.beneficio;
24         }
25         else { //ponemos lo que nos quepa.
26             int diff = M-pesoactual;
27             pesoactual= M;
28             bene_total +=((diff)/(double)o.peso)*o.beneficio;
29         }
30         i++;
31     }
32     return bene_total;
33 }

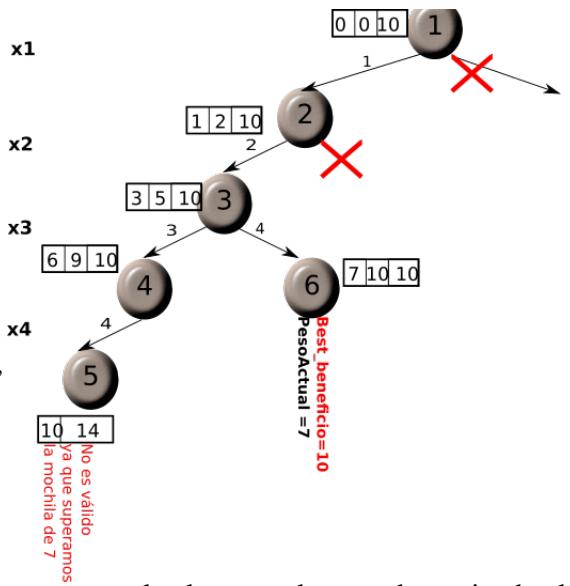
```

5.7.2 El problema de la Mochila (0|1). Árbol Combinatorio

El problema de la mochila se puede tratar usando un árbol combinatorio. Así la solución $S = (x_1, x_2, \dots, x_m)$ con $m \leq n$ tomando cada x_i un valor en $\{1, \dots, n\}$, indicando el objeto que toma. Además, por ser un árbol combinatorio debe cumplirse que $x_i < x_{i+1}$, para no repetir combinaciones. m representa el número total de objetos. Con este espacio de representación las soluciones pueden estar en cualquier nivel del árbol. Veamos un ejemplo:

Ejemplo 5.7.2

Supongamos que el número de objetos $n = 4$, la capacidad de la mochila $M = 7$ con $b = (2, 3, 4, 5)$ y $p = (1, 2, 3, 4)$. De la misma forma que en la solución con un árbol binario en cada nodo tenemos una triplete que especifica el peso actual, beneficio actual y beneficio estimado.

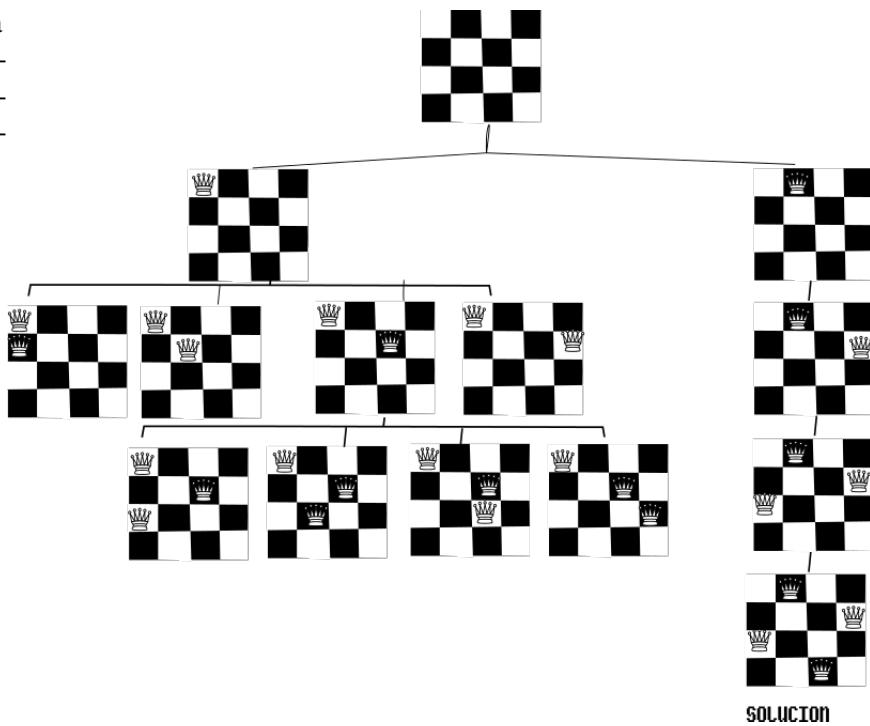


Como se puede observar en cada nivel, representa el valor que adopta cada x_i , siendo el objeto que escoge. Así la solución óptima se obtiene en el nodo 6 siendo $x_1 = 1, x_2 = 2, x_3 = 4$ con un beneficio de 10 y peso de 7. \square

5.8 Problema: N-Reinas

Dadas n reinas y un tablero de ajedrez $n \times n$, el objetivo en este problema es colocar la n reinas en el tablero sin que ninguna ataque a otra.

En esta imagen se puede ver un trozo del árbol de estados. Y la rama (a la derecha) que conduce a una solución.



Representación de la solución. La solución será un vector $S = (x_1, x_2, \dots, x_n)$ de forma que x_i representa en qué columna se coloca la reina i , fijado que la fila donde se coloca es la fila i . En este caso por cada reina i que se coloca en la fila i tenemos que probar como máximo n posibilidades que se corresponden con las n columnas de la fila. Como cada x_i toma valores entre 1 y n , el árbol de estados que nos hace falta es un n -ario. En cada nivel i se expresa dónde se pone la reina i .

Podar. El algoritmo realizará la vuelta atrás cuando dos reinas se atacan. Para ver si dos reinas se atacan hay que comprobar si comparten: fila, columna, o algunas de las dos diagonales. El código C++ para ver si dos reinas se atacan sería el siguiente:

```

1 bool Atacan( const Back_Anario &P,int filnueva){
2     Back_Anario::const_iterator it=P.begin();
3     Back_Anario::const_iterator itnueva=P.end(); --itnueva;
4
5     if (itnueva==it) return false; //solamente habia una dama
6     int fil=0;
7     for (; it!=itnueva;++it,++fil){
8         if (fil==filnueva) return true; //misma fila
9         if (*it==*itnueva) return true; //misma columna

```

```

10     if ((fil-(*it))==(filnueva-(*itnueva))){
11         return true; //misma diagonal
12     }
13     if ((fil+(*it))==(filnueva+(*itnueva))) {
14         return true;
15     }
16 }
17 return false;
18 }
```

El iterador recorre el vector datos del objeto Back_Anario que contiene la columna donde se coloca la reina i. La nueva reina se coloca en el ultimo nivel del objeto P y por lo tanto la columna donde se encuentra es *itnueva. Las líneas 10-15 permiten ver si la nueva reina comparte diagonales con otra existente. El algoritmo backtracking para colocar las n reinas es el siguiente:

```

1 void Nreinas(int n){
2     Back_Anario P(n,n,0); //arbol n-ario con n posiciones
3                         //para datos, cada datos[i]
4                         //puede tomar valores entre 1-n
5                         //se inician a 0
6     bool seguir=true;
7     int fila;
8     while (seguir){ //mientras no se hayan colocado
9         fila = P.GetLevel();
10        if (Atacan(P,fila)){
11            cout<<"Se atacan"<<endl;
12            P.Backtracking(); //hacemos backtracking
13            fila = P.GetLevel();
14        }
15        else
16            //las hemos colocado todas
17            if ((unsigned int)P.GetLevel()==n-1)
18                seguir=false; //nos salimos
19            else {
20                cout<<"No se atacan seguimos"<<endl;
21                seguir=P.GeneraSigiente();
22            }
23    }
24    cout<<endl<<"Solucion final: ";
25    MuestraPosicionDamas(P,n);
26    cout<<endl;
27 }
28 }
```

Finalmente el código que muestra las posiciones de las reinas en el tablero es el siguiente:

```

1 void MuestraPosicionDamas(const Back_Anario &P,int n){
2     Back_Anario::const_iterator it=P.begin();
3     cout<<endl;
4     int fila=0;
5     for (int i=0;i<n;i++){
6         for (int j=0;j<n;j++){
7             if (fila == i && it!=P.end() && *it==(j+1)){
8                 cout<<"D\t";
9                 ++it;
10                fila++;
11            }
12            else cout<<"X\t";
13        }
14        cout<<endl;
15    }
16 }
```

5.9 Backtracking con A. Permutacionales.

Cuando tenemos un espacio de soluciones tal que $S(x_1, x_2, \dots, x_k)$ siendo $x_i \in \{1, 2, \dots, n\}$ y cumpliendo que $x_i \neq x_j \wedge 1 \leq k \leq n$ necesitamos un árbol permutacional. Una permutación en términos generales es una nueva ordenación de una secuencia de elementos. Donde no se pueden repetir elementos. Si nuestra secuencia de elementos es n y consideramos todos los elementos después de obtener una permutación obtendremos también n elementos cambiados de sitio. Puede también ser el caso que queramos obtener una permutación de los n elementos pero al final nos quedamos solamente con los k primeros (en términos matemáticos esto sería una variación). Para poder usar este nuevo espacio de soluciones nuestro TDA Back_Anario se ha modificado obteniendo el nuevo tipo APermutacion. En este caso la representación del TDA Apermutacion sería:

```

1 class Apermutacion{
2     private:
3         vector<int>datos;
4         int level;
5     public:
6     ...
7 }
```

La diferencia con Back_Anario es que una secuencia es válida si no existen elementos repetidos desde 0 hasta level. Así por lo tanto el código que se modifica es:

```

1     bool Apermutacion::Repetidos(){
2         for (int i=0;i<=level-1;i++)
3             if (datos[i]==datos[level]) return true;
4         return false;
5     }
6     /*****************************************************************/
7     bool Apermutacion::EsSecuencia(int l){
```

```

8     if (Repetidos()) return false;
9     return level>=0 && level<=(int)datos.size()-1;
10    }
11
12    /*************************************************************************/
13    bool Apermutacion::PossibleSecuencia(int l){
14        if (Repetidos()) return false;
15        return level>=0 && level<(int)datos.size()-1;
16    }

```

Como se puede observar tanto el método PossibleSecuencia como EsSecuencia comprueba si hay repetidos en ese caso no es una secuencia válida.

El resto de métodos son iguales:

```

1 void Apermutacion::Next(int l){
2     datos[l]++;
3 }
4 /*************************************************************************/
5 bool Apermutacion::MasHermanos(int l){
6     return level>=0 && datos[level]<(int)datos.size();
7 }
8 /*************************************************************************/
9
10 bool Apermutacion::GeneraSiguiente(){
11
12     if (level<(int)datos.size()-1)//Podemos avanzar al siguiente nivel
13         level=level+1;
14     while (level>=0 && !MasHermanos(level)){
15         datos[level]=0;
16         level--;
17     }
18     do{
19         Next(level);
20         if (EsSecuencia(level))
21             return true;
22
23         if (PossibleSecuencia(level)) level++;
24         else
25             while (level>=0 && !MasHermanos(level)){
26                 datos[level]=0;
27                 level--;
28             }
29     }while (level>=0);
30     return false;
31 }
32
33 /*************************************************************************/
34
35 bool Apermutacion::Backtracking(){
36     if (level==0) return false;
37     else{
38         while (level>=0 && !MasHermanos(level)){
39             datos[level]=0;
40             level--;

```

```

41         }
42         if (level<0) return false;
43     }
44     do{
45         Next(level);
46         if (EsSecuencia(level))
47             return true;
48
49         if (PosibleSecuencia(level)) level++;
50         else
51             while (level>=0 && !MasHermanos(level)){
52                 datos[level]=0;
53                 level--;
54             }
55         }while (level>=0);
56         return false;
57     }
58
59
60     /*****************************************************************/
61     int Apermutacion::NumeroSecuenciasPosibles(){
62         unsigned int suma=0;
63
64         for (unsigned int k=1; k<=datos.size();k++){
65             int total =1;
66             for (unsigned int i=k;i<=datos.size();i++)
67                 total*=i;
68             suma+=total;
69
70         }
71         return suma;
72     }

```

5.9.1 El problema de la Asignación

En este problema tenemos n trabajadores y n tareas o trabajos. Tenemos una matriz que nos indica el beneficio de asignar un trabajador a una tarea. El objetivo es obtener la asignación trabajador-tarea que acumule el mayor beneficio posible. Es decir

$$\text{maximizar } \sum_{i=1}^n B[i, S(i)] \text{ sujeto a } i \neq j \implies S(i) \neq S(j)$$

La restricción es que a dos trabajadores diferentes no se le puede asignar la misma tarea.
La asignación debe ser uno a uno.

Veamos un primera aproximación al código Backtracking para resolver este problema.

```

1 /**
2  * @brief Obtiene el mejor beneficio de asignar a n trabajadores n trabajos
3  * usando backtracking
4  * @param n: el numero de trabajos o trabajadores
5  * @param ab: Arbol de permutaciones para obtener la mejor solucion
6  * @param B: matriz de beneficios

```

```

7   * @return el mejor beneficio total
8   */
9
10  int Asig_Trabajadores(int n, Apermutacion &ab, const Matriz<unsigned int> &B){
11      Apermutacion P(n);
12      bool seguir =true;
13      int bact=0; int best_beneficio=0;
14      unsigned int nodos_recorridos =0; //contabilizar el numero de nodos
15      while (seguir){
16          nodos_recorridos++;
17          bact=Suma_Beneficio(P,B);
18          if (P.GetLevel()==n-1){
19              if (bact>best_beneficio){
20
21                  best_beneficio=bact;
22                  ab=P;
23                  seguir =P.Backtracking(); //da igual hacer P.GeneraSiguiente()
24                                         //ya que estamos en una hoja
25              }
26              else seguir=P.GeneraSiguiente();
27          }
28          else seguir=P.GeneraSiguiente();
29      }
30      int total=P.NumeroSecuenciasPosibles();
31      cout<<"Numero de nodos recorridos "<<nodos_recorridos<< " total nodos "<<t
32      " Porcentaje "<<(nodos_recorridos/(double)total)*100.0<<endl;
33
34      return best_beneficio;
35  }

```

Ejemplo 5.9.1

Supongamos que tenemos la siguiente matriz de beneficios (en las filas los trabajadores o personas y en las columnas trabajos o tareas):

		Tareas		
		1	2	3
Personas	1	4	9	1
	2	7	2	3
	3	6	3	5

Como se puede observar la mejor asignación en este ejemplo es $S(2, 1, 3)$ con un beneficio de 21. Una traza del código visto anteriormente para este ejemplo es el que se muestra en la siguiente tabla:

Iteración	Bact	BestBeneficio	Asignación $x_1x_2x_3$	BestAsignación $x_1x_2x_3$
0	4	0	1	
1	6	0	1 2	
2	11	0	1 2 3	
3	7	11	1 3	1 2 3
4	10	11	1 3 2	1 2 3
5	9	11	2	1 2 3
6	16	11	2 1	1 2 3
7	21	11	2 1 3	1 2 3
8	12	21	2 3	2 1 3
9	18	21	2 3 1	2 1 3
10	1	21	3	2 1 3
11	8	21	3 1	2 1 3
12	11	21	3 1 2	2 1 3
13	3	21	3 2	2 1 3
14	9	21	3 2 1	2 1 3

Como se puede observar hasta que no tengamos una asignación completa el mejor beneficio no se modifica. Así es por ejemplo hasta la iteración 2 que se obtiene un beneficio de 11 con la asignación $S = (1, 2, 3)$ y por lo tanto ya en la iteración 3 vemos que el mejor beneficio es 11.

□

Como se puede observar esta primera aproximación es poco eficiente ya que explora todas las hojas y nodos intermedios del árbol de soluciones. Para mejorar este código vamos a especificar una cota que permita hacer vuelta atrás cuando se estime que no se puede mejorar por ese camino lo que tenemos hasta el momento. Como nuestro objetivo es maximizar el beneficio acumulado, una estimación tiene que dar un valor mayor o igual del beneficio acumulado que obtendríamos si siguieramos por ese camino, esto es una COTA SUPERIOR. En el caso de que esta COTA SUPERIOR sea inferior al beneficio de la actual solución por ese camino no continuamos. Por ejemplo una COTA SUPERIOR puede ser acumular el beneficio actual más el beneficio que se obtiene asignando a los trabajadores libres las tareas no asignadas que proporcionan más beneficio, incluso aunque en esta asignación surjan repeticiones. Es decir podemos a dos trabajadores libres asignarle la misma tarea libre si es la que mayor beneficio proporciona.

Así para obtener la COTA SUPERIOR el código C++ podría ser el siguiente:

```

1  /**
2   * @brief Establece una cota superior del beneficio para los trabajadores
3   * aun no asignados a ningun trabajo.
4   * @param asignados: vector con los trabajadores asignados ya a trabajos
5   * @param B: matriz de beneficios
6   * @return el beneficio estimado de asignar a los trabajadores aun no
7   * asignados trabajos aun libres.Puede que se repita dicha asignacion
8   * */

```

```

9  int CotaSuperior(vector<int>asignados,const Matriz<unsigned int> &B){
10     Matriz<bool>usados(asignados.size(),asignados.size(),false);
11     int n=asignados.size();
12     vector<bool>candidatos(n,true);
13
14     //Identificamos los trabajadores libres.
15     for (unsigned int i=0;i<asignados.size();i++){
16         if(asignados[i]>=0){
17             candidatos[i]=false;//trabajador asignado
18             for ( int j=0;j<n;j++)
19                 usados[j][asignados[i]]=true; //eliminamos su fila
20         }
21     }
22     unsigned int best_bene=0;
23     for (int i=0;i<n;i++){
24         if (candidatos[i]){// es un candidato
25             //buscamos entre los trabajos que quedan libres el mas beneficioso
26             int mejor=0;
27             for (int j=0;j<n;j++){
28                 if (usados[i][j]==false)
29                     if ((int)B.get(i,j)>mejor){
30                         mejor = B.get(i,j);
31                     }
32             }
33             best_bene +=mejor;
34         }
35     }
36     return best_bene;
37 }
```

Este código en las líneas 14-21 identifica los trabajadores libres. Para ello usa el vector candidatos de forma que si adopta un valor true está libre en otro caso es false. Igualmente para cada tarea que esta ya asignada se identifica para no cogerla, esto se hace mediante la matriz usados. A continuación en las líneas 22-35 para los trabajadores libres (candidatos[i] es true) miramos entre las tareas libres la que obtenga mayor beneficio.

El código para obtener la asignación usando la COTA SUPERIOR sería el siguiente:

```

1 /**
2  * @brief Obtiene el mejor beneficio de asignar a n trabajadores n trabajos
3  * usando backtracking y una poda basada en un cota superior
4  * @param n: el numero de trabajos o trabajadores
5  * @param ab: Arbol de permutaciones para obtener la mejor solucion
6  * @param B: matriz de beneficios
7  * @return el mejor beneficio total
8  */
```

```

9  int Asig_Trabajadores_Poda(int n, Apermutacion &ab,
10    const Matriz<unsigned int> &B){
11      Apermutacion P(n);
12      bool seguir =true;
13      int bact=0; int best_beneficio=0;
14      unsigned int nodos_recorridos =0;
15      while (seguir){
16          nodos_recorridos++;
17          bact=Suma_Beneficio(P,B);
18          if (P.GetLevel()==n-1){
19              if (bact>best_beneficio){
20                  best_beneficio=bact;
21                  ab=P;
22                  seguir =P.Backtracking();
23              }
24              else
25                  seguir=P.GeneraSiguiente();
26          }
27          else{
28              vector<int> C=ObtainAsignaciones(P,n);
29              //beneficio estimado usando la COTA SUPERIOR
30              int bestimado = bact+CotaSuperior(C,B);
31              if (bestimado >=best_beneficio)
32                  seguir=P.GeneraSiguiente();
33              else//hacemos vuelta atrás
34                  //no vamos a obtener nada mejor
35                  seguir =P.Backtracking();
36
37      }
38  }
39  int total=P.NumeroSecuenciasPosibles();
40  cout<<"Número de nodos recorridos "<<nodos_recorridos<< " total nodos "
41  <<total<<" Porcentaje "<<(nodos_recorridos/(double)total)*100.0<<endl;
42  return best_beneficio;
43 }
```

Ejemplo 5.9.2

Supongamos que planteamos el mismo ejemplo de asignación del ejemplo 5.9.1:

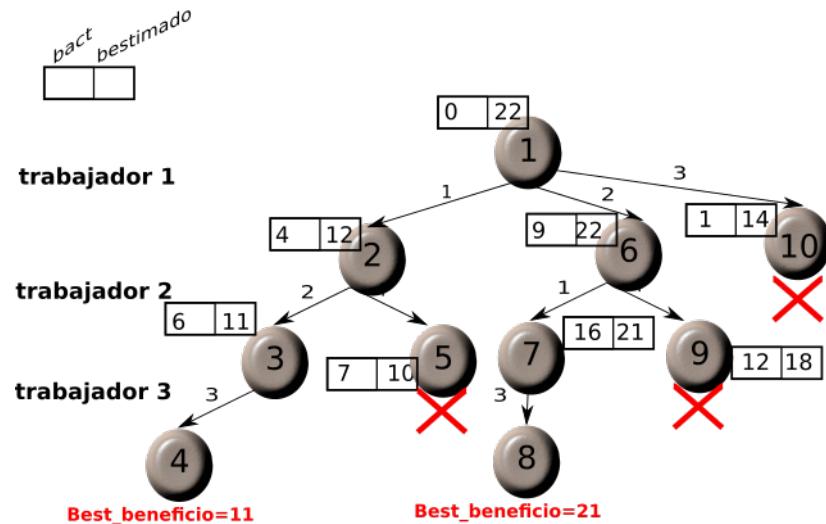
Tareas				
Personas		1	2	3
	1	4	9	1
	2	7	2	3
	3	6	3	5

Ahora el objetivo es plantear el algoritmo backtracking usando una poda más eficiente basada en la COTA SUPERIOR comentada anteriormente.

En este caso las iteraciones que realizaría el algoritmo backtracking son:

Iteración	Bact	BestBeneficio	Beneficio Estimado	Asignación $x_1x_2x_3$	BestAsignación $x_1x_2x_3$
Inicio	0	0	22		
0	4	0	12	1	
1	6	0	11	1 2	
2	11	0	11	1 2 3	
3	7	11	10	1 3	1 2 3
4	9	11	22	2	1 2 3
5	16	11	21	2 1	1 2 3
6	21	11	21	2 1 3	1 2 3
7	12	21	18	2 3	2 1 3
8	1	21	14	3	2 1 3

En la siguiente imagen se muestra los nodos explorados del árbol de soluciones. Además se muestra que caminos no se exploran porque el mejor beneficio (BestBeneficio en la tabla) es mejor que el beneficio estimado que se puede obtener por ese camino.



□

5.10 Resolución de Juegos

La técnica de backtracking se puede aplicar en problemas de juegos: damas, ajedrez, tres en raya, etc. El objetivo de usar esta técnica en este tipo de problemas es decidir el movimiento óptimo que debe realizar el jugador que empieza movimiento. Para ello se realiza un análisis en profundidad, describiendo: si juego este movimiento, que juega mi contrario, y si como respuesta a mi contrario juega esto otro, así hasta alcanzar un nivel máximo. Alcanzado este nivel máximo se valoran cada una de esas posibilidades, y esas valoraciones ascienden

por los niveles de análisis, mediante una técnica que describiremos hasta alcanzar la raíz. Con esta información el jugador que juega puede decidir qué movimiento hacer.

CARACTERÍSTICAS.

- En el juego participan dos jugadores *A* y *B*, que mueven alternativamente.
- En cada movimiento un jugador puede elegir entre un número finito de posibilidades.
- El resultado del juego puede ser: gana *A*, gana *B* o empatan. El objetivo de los dos jugadores es ganar.
- Los juegos se suponen que no influye el azar. Así es en el ajedrez, damas, damas chinas, tres en raya, cuatro en linea, etc.

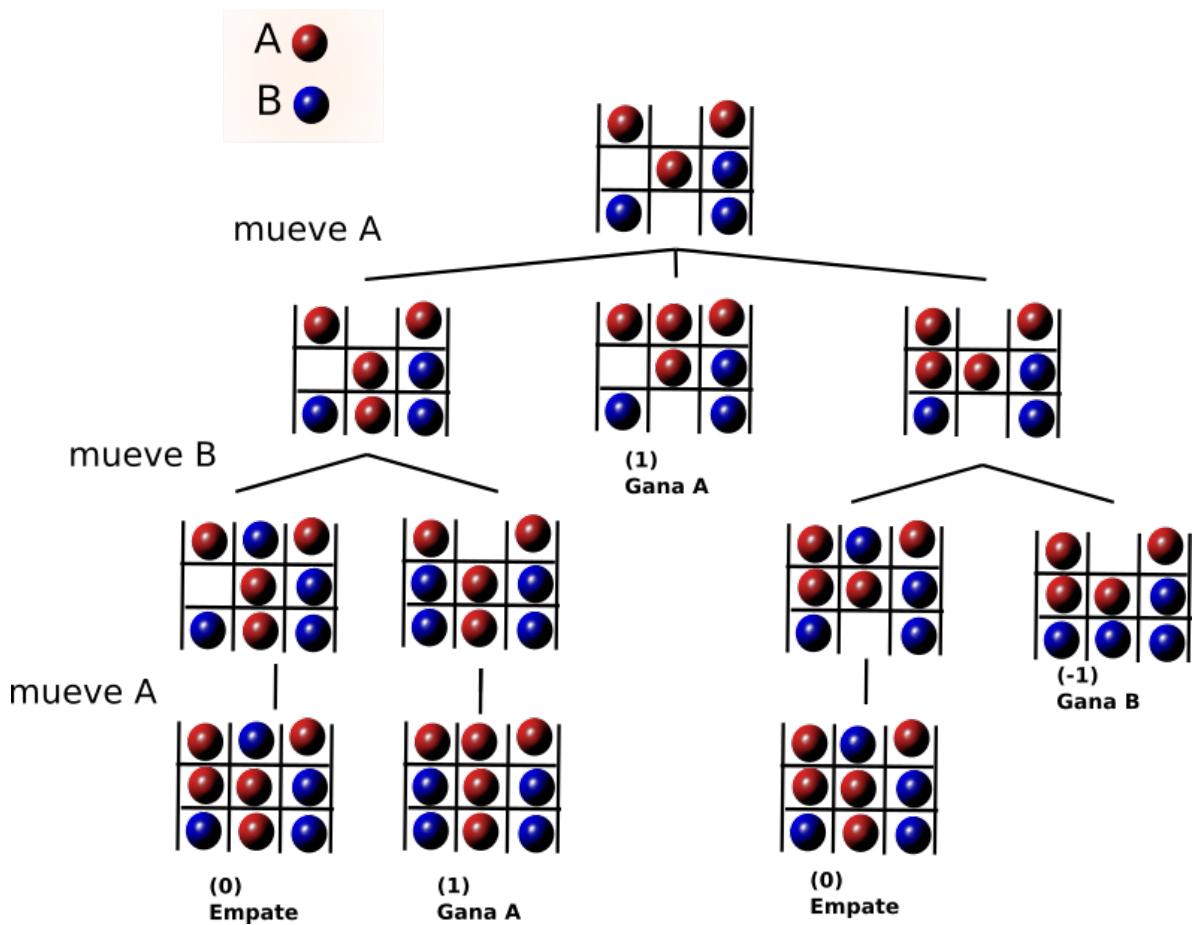
5.10.1 Árbol del Juego

En la técnica backtracking hemos hablado del árbol de estados, en este caso tratando de problemas de resolución de juegos, a este árbol lo denominamos árbol del juego. Así en el árbol tenemos:

- Cada nodo del árbol representa un posible estado del juego.
- La raíz representa el comienzo de la partida
- Los descendientes de un nodo son los movimientos para cada jugador. Por ejemplo:
 - ◊ En el nivel 1 mueve el jugador *A*,
 - ◊ En el nivel 2 mueve el jugador *B*
 - ◊ En el nivel 3 mueve el jugador *A*,
 - ◊ etc.
- Una hoja es una situación donde acaba el juego o llegamos a un nivel máximo de análisis (profundidad).

Ejemplo 5.10.1

Tres en Raya.- Dos jugadores *A* y *B* están jugando al 3 en raya. El jugador *A* lleva las piezas rojas y el jugador *B* las piezas azules. El jugador quiere hacer un análisis de su movimiento y para eso hace un estudio en profundidad de lo que puede pasar al hacer un determinado movimiento. Así en la raíz se muestra el estado actual del tablero. En el nivel 1 los diferentes movimientos del jugador *A*. En el nivel dos los movimientos del jugador *B* y en el tercer nivel los movimientos del jugador *B*. Como se puede observar con un solo movimiento el jugador puede ganar, y esta debería ser la opción que eligiese.



□ Para que el jugador A obtenga una recomendación sobre que jugar tiene que realizar un etiquetado de las hojas y ese valor debe propagarse hasta la raíz. De esta forma las etiquetas son:

- 1: si gana A
- -1: si gana B
- 0: si empatan.

El objetivo de A es encontrar un camino en el árbol que le lleve hasta un nodo hoja con valor 1. Pero, el jugador A tambien tiene que desarrollar una estrategia si desde el estado actual del tablero no se llega a un nodo hoja con valor 1. Hay que tener en cuenta que:

- En los movimientos de B, el jugador B intentará llegar a hojas con valor -1, por lo tanto el jugador A debe prever esto y jugar en función de evitarlo.
- En el caso de que A pueda llegar a una hoja etiquetada con 1 antes de una hoja etiquetada con -1, debería intentar buscar el camino que lo lleve a la hoja etiquetada con 1.

Para que el jugador A pueda decidir esta información de las hojas debe propagarse hacia los padres. Con tal fin se va a definir una estrategia denominada **MiniMax**.

5.10.2 Estrategia Minimax

La estrategia Minimax usa una función de utilidad, dando para cada nodo hoja un valor numérico. Esta valor indica como de buena es esa situación para el jugador A.

Proceso de resolución de juegos.-

Generar el árbol de juego hasta un nivel determinado, ya sea hasta un nivel máximo o se determine el juego (gana, pierde o empata). A continuación se aplica la función de utilidad a los nodos hoja. Una vez obtenido el valor numérico par a los nodos hoja propagar los valores de utilidad hasta la raíz, usando la estrategia **MiniMax**:

- En los movimiento impares tomar el máximo de los hijos.
- En los movimientos pares tomar el mínimo de los hijos.
- Solución Final. Escoger el movimiento indicado por el hijo de la raíz con mayor valor.

La implementación de la técnica **MiniMax** se hace mediante la técnica backtracking recursivo. El algoritmo en términos generales sería el siguiente:

Algorithm 2 Estrategia Minimax

```

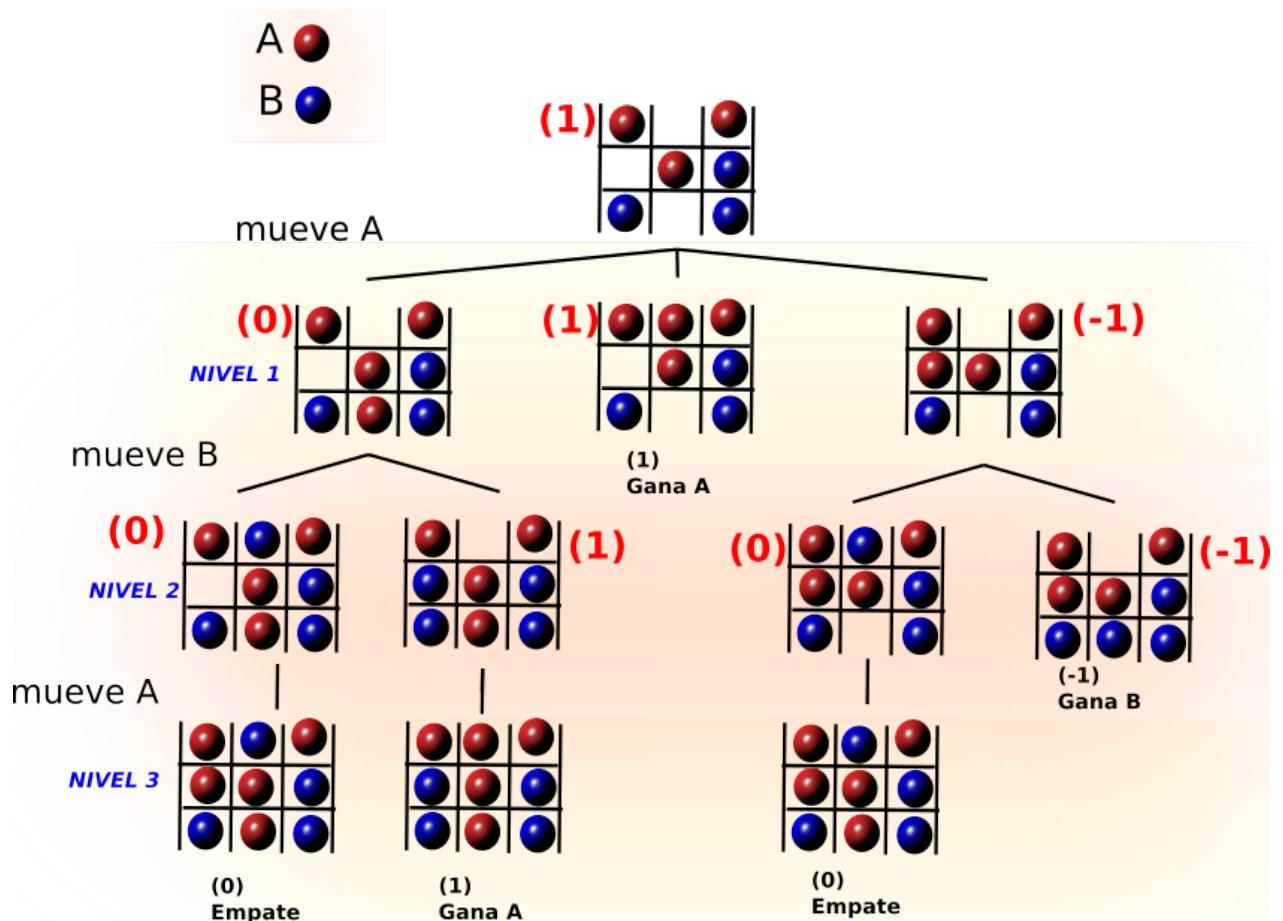
1: procedure BUSCAMINIMAX()(B : tipotablero, modo : (MAX, MIN)) : real
2:   if EsHoja(B) then return Utilidad(B,modo)
3:   end if
4:   if modo==MAX then
5:     valoract=-∞
6:   else valoract=∞
7:   end if
8:   for cada hijo C del tablero B do
9:     if modo==MAX then
10:       valoract=max(valoract,BuscaMinimax(C,MIN));
11:     else
12:       valoract=min(valoract,BuscaMinimax(C,MAX));
13:
14:   return valoract
15:

```

Si el árbol de juego es muy grande o infinito, como puede ocurrir en el ajedrez, entonces la función de utilidad se debe poder aplicar sobre situaciones no terminales (p.e en el ajedrez, no siempre tiene que llegar a ser mate o tablas). En este caso la función de utilidad es una medida heurística que indica como de prometedora es la situación para el jugador A.

Ejemplo 5.10.2

Volviendo al ejemplo 5.10.1 vamos a ver, usando la estrategia **MiniMax**, como se propagan los valores de las nodos hijos a los padres, empezando por los nodos hoja. Este proceso de propagación se puede observar en las siguiente figura:



En los tableros que no se corresponden a un nodo hoja se pueden ver los valores (rojo) que se propagan de los hijos. Así por ejemplo de izquierda a derecha en el Nivel 1 podemos observar:

- El nodo más a la izquierda toma un valor de 0. Los hijos toman un valor de 0 y 1. Se escoge 0 porque quié juega es (B) y se supone que (B) escogerá los mejor para él que será ir por el nodo izquierdo.
- Por otro lado en el Nivel 1 el nodo en el centro lleva a una posición ganadora, siendo este un nodo hoja, y se etiqueta directamente con 1.
- El nodo más a la derecha se etiqueta con (-1) ya que al ser B quién juega a continuación, éste se conducirá por lo mejor para él.

Con este análisis en el Nivel 1 el jugador A puede decidir que jugar buscando lo mejor entre 0, 1, -1, siendo 1 que le lleva a ganar la partida. \square

En general la estrategia **Minimax** actúa de la siguiente forma:

- Si juega B el nodo se etiqueta con el mínimo de los hijos.
- Si juega A el nodo se etiqueta con el valor máximo de entre los hijos.

Ejemplo 5.10.3

El juego come aceitunas.-Este juego es una ampliación del juego de los palillos, ya que además de quitar un palillo te comes una aceituna. En este juego tiene n filas de palillos con aceitunas. Cada jugador debe comerse una o varias aceitunas pero de la misma fila siempre. Aquel jugador que se tenga que comer la última aceituna pierde. El tablero que usamos es un vector indicando en cada posición el número de aceitunas que nos queda en la fila i-ésima. Las funciones que se describen en el algoritmo 5.10.2 son las siguientes:

```
EsHoja(B)
    return NumeroAceitunas(B)<=1
```

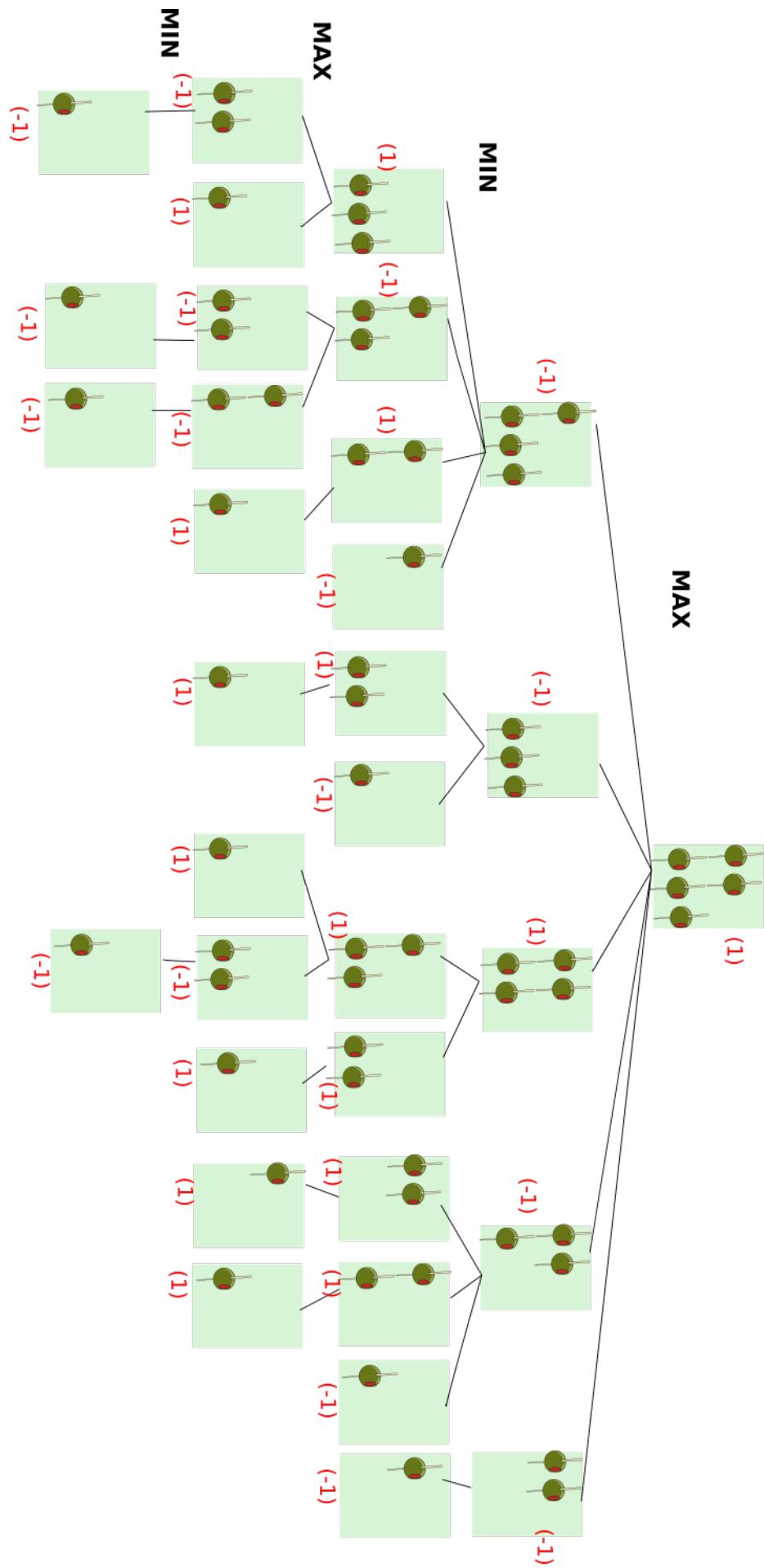
De esta forma un tablero es una hoja cuando nos quede solamente un numero menor o igual que 1.

```
Utilidad(B,modo)
    Si (NumeroAceitunas(B)==0 AND modo==MAX)
        or (NumeroAceitunas(B)==1 AND modo ==MIN) entonces
            return 1;
        sino
            return -1;
```

La forma de generar los hijos de B, tableros siguientes en la jerarquía sería:

```
for (int i=1;i<=n;i++)
    for (j=1;j<=B[i]; j++){
        C=B;
        C[i]=C[i]-j;
    }
```

En cada fila podemos quitar desde 1 hasta B[i] aceitunas, dando lugar a un nuevo nodo. En la siguiente imagen se muestra para 2 filas y teniendo 2 aceitunas en la primera fila y 3 aceitunas en la segunda fila, el jugador Max gana, siguiendo el camino que marca el tercer hijo empezando por la izquierda.



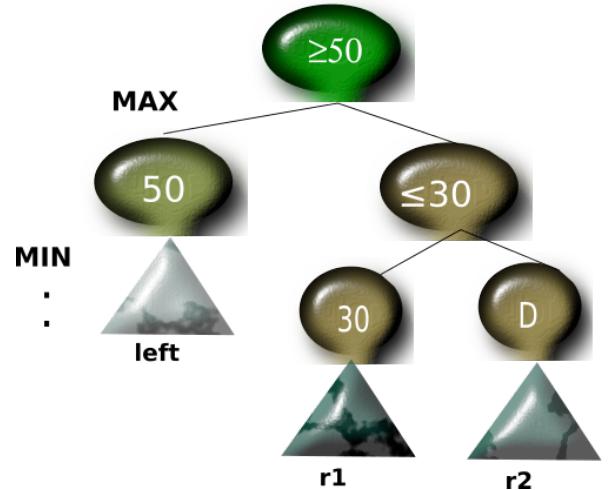
□

5.10.3 Poda Alfa-Beta

Sobre los árboles de juego se puede aplicar un tipo propio de poda, conocida como la poda $\alpha - \beta$.

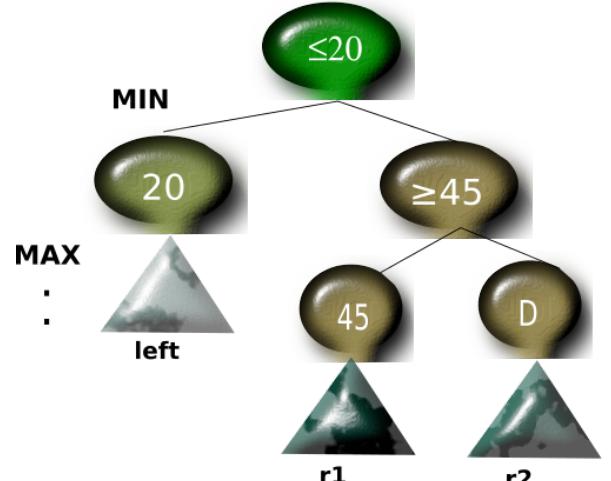
Poda α

Cuando analizamos left valoramos el nodo raíz con al menos 50 que es lo que se ha obtenido por la rama izquierda (left en la imagen). Ahora analizamos la parte derecha y en particular r1 para ver si podemos obtener algo mejor que 50. Pero en la imagen a la derecha vemos que por r1 obtenemos como máximo 30, pudiendo obtener incluso menos, ya que el hijo a la derecha de la raíz su valor se obtiene como una minimización de los valores hijos. Así que como mucho por la parte derecha llegamos a 30 que es menor que 50, por lo tanto no merece la pena explorar r2. Así que r2 se poda.



Poda β

Cuando analizamos left valoramos el nodo raíz con 20 como máximo cuando solamente se ha explorado left. Al pasar a explorar r1 obtenemos un valor de 45 luego el nodo hijo derecho de la raíz obtendrá como mínimo un valor de 45, por lo tanto la raíz no cambiará su valor de 20 si exploramos r2. Así que r2 se poda.



El algoritmo Minimax que aplica poda $\alpha - \beta$ sería

Algorithm 3 Estrategia Minimax con Poda Alfa-Beta

```

1: procedure ALFA_BETA(nodo, $\alpha$ , $\beta$ ,jugador)
2:   if EsHoja(nodo) then return Utilidad(nodo,jugador)
3:   end if
4:   if jugador==MAX then
5:     for cada hijo C de nodo do
6:        $\alpha \leftarrow \max(\alpha, ALFA\_BETA(C, \alpha, \beta, MIN))$ 
7:       if  $\beta \leq \alpha$  then return  $\alpha$ 
8:       end if
9:     end for
10:    return  $\alpha$ 
11:   end if
12:   if jugador==MIN then
13:     for cada hijo C de nodo do
14:        $\beta \leftarrow \min(\beta, ALFA\_BETA(C, \alpha, \beta, MAX))$ 
15:       if  $\beta \leq \alpha$  then return  $\beta$ 
16:       end if
17:     end for
18:     return  $\beta$ 
19:   end if

```

La poda β se puede observar que se realiza en la línea 7. Y la poda α se realiza en la línea 15.

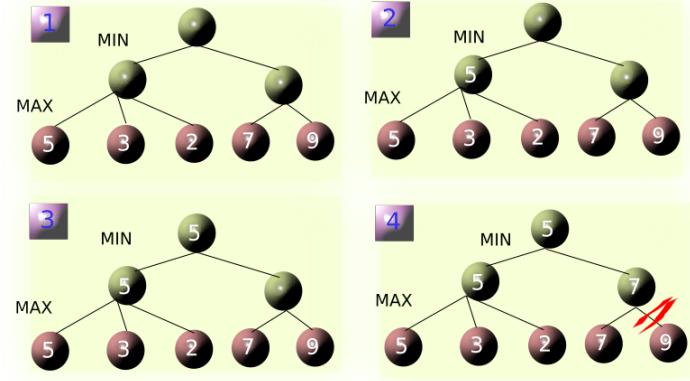
Ejemplo 5.10.4

En la siguiente imagen se puede ver los pasos para llenar los valores de los nodos vacíos tras haber valorado las hojas. La llamada incial que se hace al algoritmo sería:

Alfa_Beta(n,-∞,∞,MIN);

En la viñeta 1 se pueden ver los valores dados a las hojas. En la viñeta dos se puede ver que el nodo hijo a la izquierda de la raíz toma el valor 5 al ser un nodo MAX, en este caso se toma el máximo entre 5,3 y 2. Este valor es el que adopta en primer lugar la raíz un valor de 5, como se puede ver en la viñeta 3. Ahora se valora el nodo hijo a la derecha, adoptando un valor de 7. Con este valor se puede analizar si la raíz puede llegar a tomar un valor menor de 5. El hijo a la derecha si sigue analizando tomará un valor mayor o igual a 7 por lo tanto no es posible que la raíz tome el valor del hijo a la derecha, ya que el tiene un valor de 5 que es menor que 7. Como se puede ver en la viñeta 4 se poda la rama a la derecha del nodo hijo a la derecha de la raíz.

En este ejemplo se ha realizado una poda alfa.



□

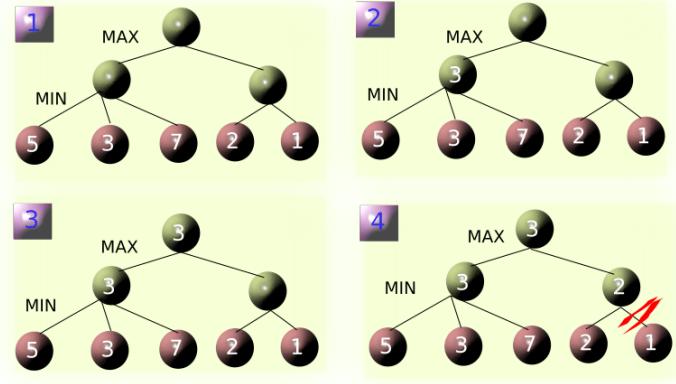
Ejemplo 5.10.5

En la siguiente imagen se puede ver los pasos para llenar los valores de los nodos vacíos tras haber valorado las hojas. La llamada inicial que se hace al algoritmo sería:

Alfa_Beta(n,-∞,∞,MAX);

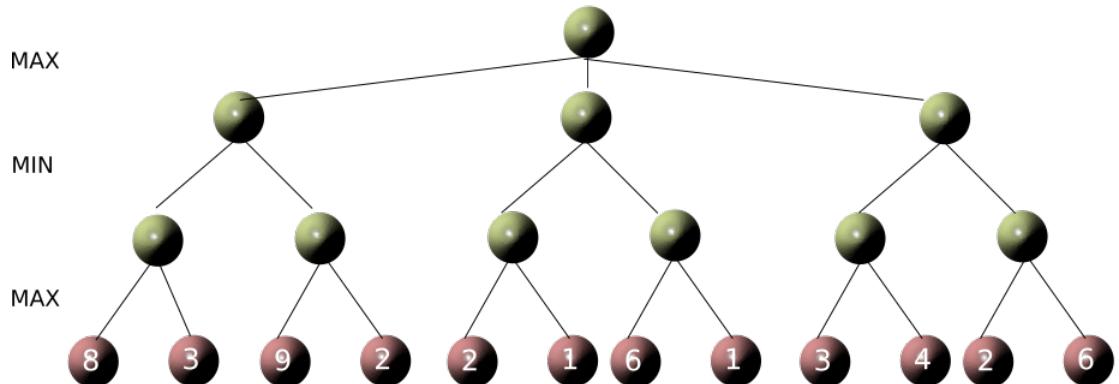
En la viñeta 1 se pueden ver los valores dados a las hojas. En la viñeta dos se puede ver que el nodo hijo a la izquierda de la raíz toma el valor 3 al ser un nodo MIN, en este caso se toma el mínimo entre 5,3 y 7. Este valor es el que adopta en primer lugar la raíz un valor de 3, como se puede ver en la viñeta 3. Ahora se valora el nodo hijo a la derecha, adoptando un valor de 2. Con este valor se puede analizar si la raíz puede llegar a tomar un valor mayor de 3. El hijo a la derecha si sigue analizando tomará un valor menor o igual a 2 por lo tanto no es posible que la raíz tome el valor del hijo a la derecha, ya que el tiene un valor de 3 que es mayor que 2. Como se puede ver en la viñeta 4 se poda la rama a la derecha del nodo hijo a la derecha de la raíz.

En este ejemplo se ha realizado una poda beta.



Ejemplo 5.10.6

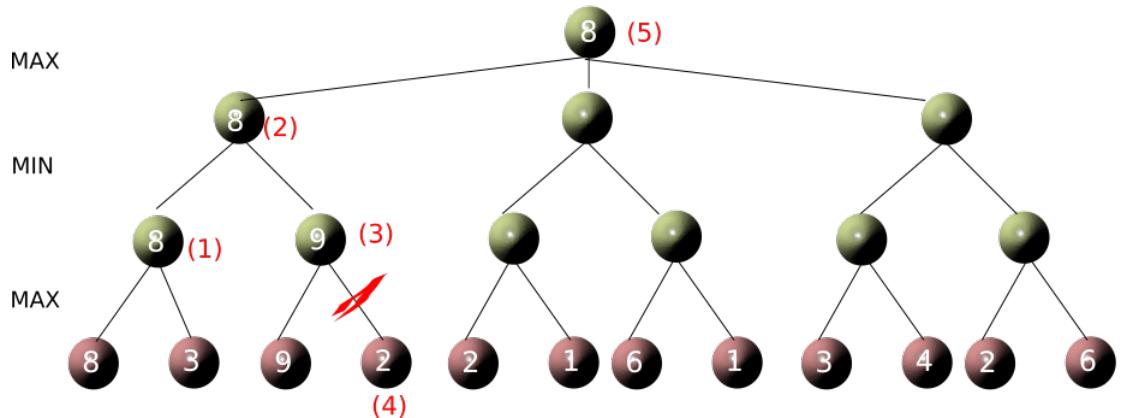
Aplicar el algoritmo Minimax con poda alfa beta al siguiente árbol



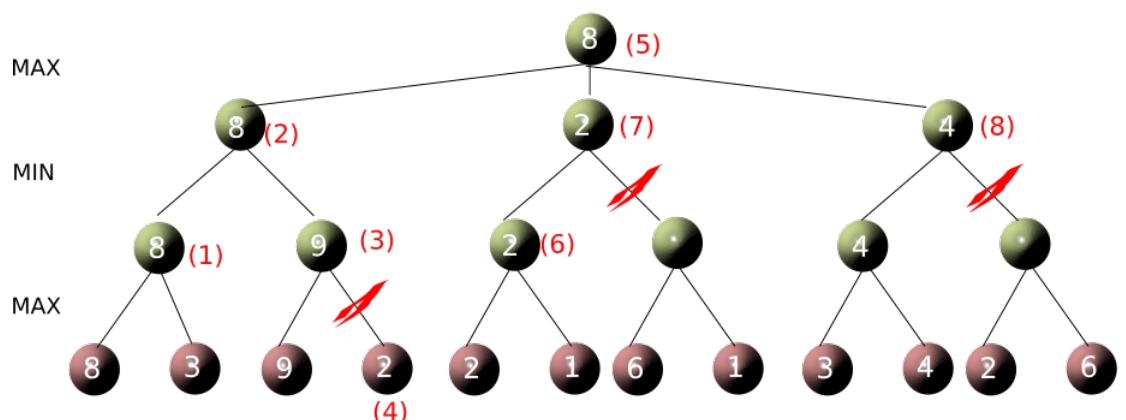
Analicemos en las siguientes imágenes como se han valorado los nodos

- El nodo (1) es un nodo MAX por lo tanto tiene que escoger entre el máximo de 8 y 3. Ahora mismo α tiene un valor de $-\infty$ y β de ∞ . Por lo tanto al ser en (1) un jugador MAX se α se modifica a 8.

2. En el nodo (2) $\beta = \infty$, y por lo tanto al escoger entre el mínimo entre 8 y ∞ el nodo (2) toma el valor $\beta = 8$.
3. En el nodo (3) es un nodo MAX por lo tanto el valor de α ahora mismo es $-\infty$, y al analizar el hijo con valor 9 α adopta el valor 9. A continuación desde el nodo (3) ocurre una poda β , ya que tenemos en el nodo (2) $\beta = 8$ y el hijo $\alpha = 9$. Por lo tanto es imposible que por el nodo (4) (3) adopte un valor más pequeño que 9. Así que el nodo (4) se poda.
4. En el nodo raíz (5) se modifica a 8, siendo este nuestro α en el nivel 1.



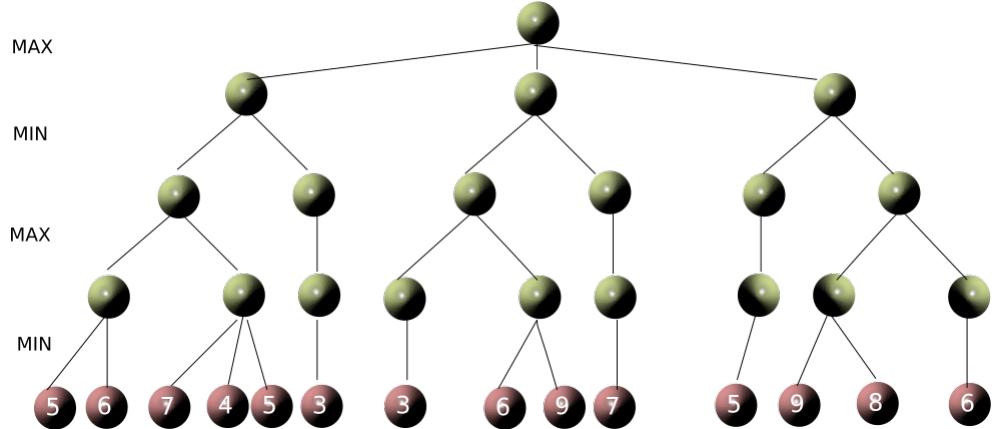
5. El nodo (6) toma un valor $\alpha = 2$. A su vez por el nodo (6), el nodo (7) toma un valor $\beta = 2$. Ahora desde el nodo (7) que es un nodo MIN tiene como $\beta = 2$ pero su α de referencia es $\alpha = 8$ del nodo (5). Por lo tanto por el nodo (7) es imposible obtener algo mayor que 8. Es decir no se va a modificar el valor de (5) por (7). Así que la rama izquierda del nodo (7) se poda.
6. Por otro lado el nodo (8) llega a obtener por su rama izquierda un valor de 4. Por lo tanto es imposible que por su rama derecha se obtenga un valor mayor que 4 y por lo tanto el nodo (5) que tiene un valor de 8 no se modificará por el nodo (8).



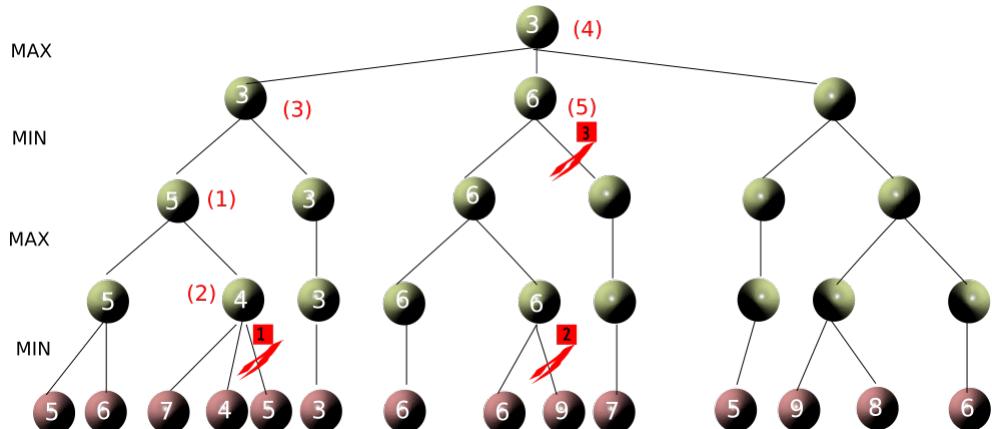
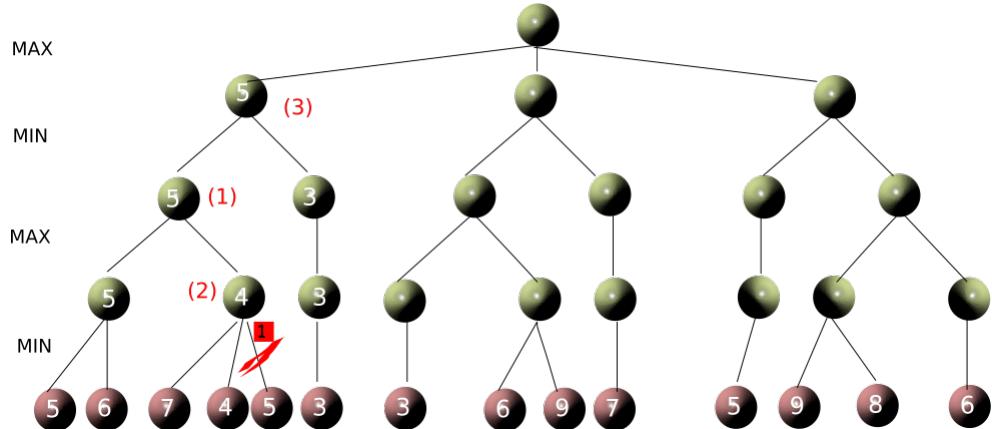
□

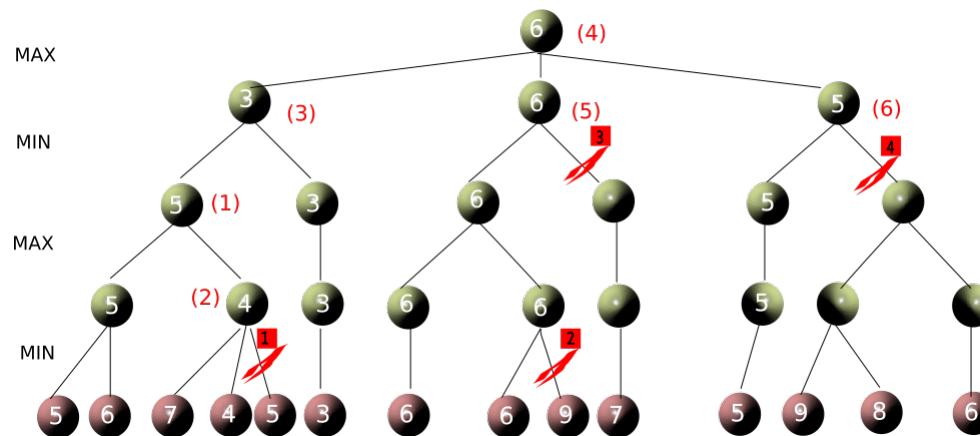
Ejemplo 5.10.7

Aplicar el algoritmo Minimax con poda alfa-beta al siguiente árbol:



En las siguientes imágenes se ve el proceso de valoración del nodo raíz, con las podas correspondientes:





□

