6. Branch and Bound

6.1 Introducción

La técnica Branch and Bound o Ramificación y Poda, al igual que Backtracking realiza una exploración sistemática del árbol de soluciones. Y por lo tanto se usa en optimización y en problemas de juegos. La diferencia principal con Backtracking es que el recorrido no tiene por qué ser necesariamente primero en profundidad. El orden vendrá dado por el nodo activo que tenga un mejor beneficio estimado. Además para optimizar la exploración se usarán cotas en cada nodo que determine el rango en el que está el beneficio que se puede llegar a obtener explorando por el camino que sigue a ese nodo.

6.2 Estimación de Cotas

Dada una solución parcial en la que nos encontramos en el nodo i debemos establecer, antes de seguir explorando por i, una acotación del beneficio que se puede llegar a alcanzar. ***Para cada nodo*** i. Se estima una cota superior $CS(i)$, una cota inferior $CI(s)$ y el beneficio estimado que se puede llegar a obtener $BE(i)$. De forma que:

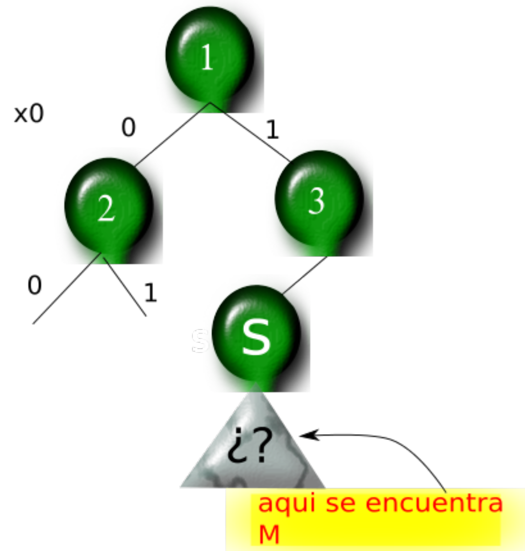
- $CS(i)$. Cota superior del beneficio óptimo que podemos alcanzar a partir del nodo i. El valor real que se puede llegar a obtener por i, $Valor(i)$, cumple que $Valor(i) \leq CS(i)$.
- $CI(i)$. Cota inferior del beneficio óptimo que podemos alcanzar a partir del nodo i. El valor real que se puede llegar a obtener por i, $Valor(i)$ cumple que $Valor(i) \geq CI(i)$.
- $BE(i)$. Beneficio estimado óptimo que se puede encontrar a partir del nodo i. Este valor permite decidir cual es el siguiente nodo que se explora.

La cota superior y cota inferior deben ser valores fiables, en el sentido de que el valor real si exploramos por ese nodo esta comprendido entre ambos, y no existe un rango muy amplio entre la cota inferior y cota superior.

En la imagen a la derecha se puede observar que antes de explorar s hay que acotar el beneficio mayor que se puede alcanzar por M. El valor de M cumple que:

$$\underbrace{CI(s)}_{\text{cota inferior}} \leq Valor(M) \leq \underbrace{CS(s)}_{\text{cota superior}}$$

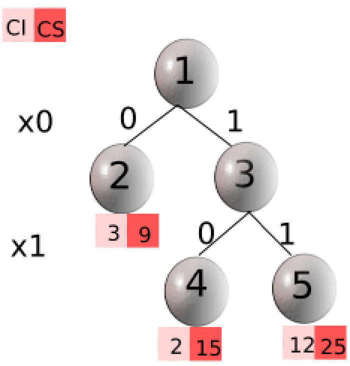
Si la cota inferior coincide con la cota superior tenemos el valor real de M.



6.3 Estrategia de Poda

La forma en como se realiza la poda depende del problema: si es un problema de maximización o si es un problema de minimización. Si estamos en un problema de maximización, dejaremos de explorar un nodo cuando su cota superior, CS, sea menor que lo mejor obtenido hasta el momento o CS sea menor que lo mínimo estimado que se puede alcanzar. Esto se puede ver en el siguiente ejemplo.

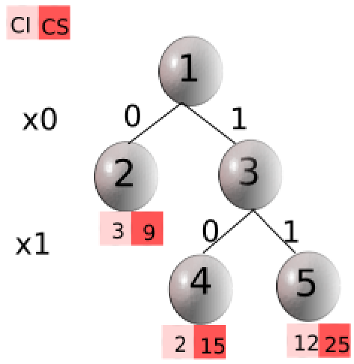
Ejemplo 6.3.1
Supongamos que estamos en un problema de maximización y hemos recorrido varios nodos, estimando para cada uno de ellos la cota superior $CS(i)$ y la cota inferior $CI(i)$. Hemos alcanzado el nodo 5 con $CS(5) = 25$ y $CI(5) = 12$. Ahora nos planteamos si debemos explorar el nodo 2. Claramente no tenemos que explorarlo ya que lo mejor que podemos obtener por 2 es un valor de beneficio de 9 ya que $CS(2) = 9$. Y por el nodo 5 como mínimo tenemos un beneficio de 12. Por lo tanto el nodo 2 no se explora. Por otro lado el nodo 4 si debemos explorarlo ya que tenemos una cota superior, que podría ser el valor real, de 15 que es mayor que la cota inferior de 5 que es 12.



□

En el siguiente ejemplo se muestra ahora una situación cuyo objetivo es minimizar.

Ejemplo 6.3.2
Supongamos que estamos en un problema de minimización y hemos recorrido varios nodos, estimando para cada uno de ellos la cota superior $CS(i)$ y la cota inferior $CI(i)$.
Hemos alcanzado el nodo 2 con $CS(2) = 9$ y $CI(2) = 3$. Ahora nos planteamos si debemos explorar el nodo 5. Claramente no tenemos que explorarlo ya que lo mejor que podemos obtener por 5 es un valor de beneficio de 12 ya que $CI(5) = 12$. Y por el nodo 2 como máximo tenemos un beneficio de 9. Por lo tanto el nodo 5 no se explora. Por otro lado el nodo 4 si debemos explorarlo ya que tenemos una cota inferior, que podría ser el valor real, de 2 que es menor que la cota inferior del nodo 2, $CI(2)$, que es 3.



□

Maximización: Criterio de Poda

El criterio de poda en problemas de optimización, que tienen que obtener el máximo de una función objetivo, se basa en dos criterios:
Se poda el nodo *i* si cumple alguna de las dos condiciones siguientes:

1. $CS(i) \leq CI(j)$ siendo j otro nodo considerado
2. $CS(i) \leq Valor(s)$ siendo s un nodo que representa una solución final y actual (la mejor hasta el momento).

Implementación.- Se usa una variable C que es la cota actual. Esta se obtiene como:

$$C = max\left(\underbrace{\{CI(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Podar el nodo i si $CS(i) \leq C$.

Minimización: Criterio de Poda

El criterio de poda en problemas de optimización que quiere obtener el mínimo de una función objetivo, se basa en dos criterios:
Se poda el nodo *i* si cumple alguna de las dos condiciones siguientes:

1. $CI(i) \geq CS(j)$ siendo j otro nodo considerado
2. $CI(i) \geq Valor(s)$ siendo s un nodo que representa una solución final y actual (la mejor hasta el momento).

Implementación.- Se usa una variable C que es la cota actual. Esta se obtiene como:

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Podar el nodo i si $CI(i) \geq C$.

6.4 Estrategia de Ramificación

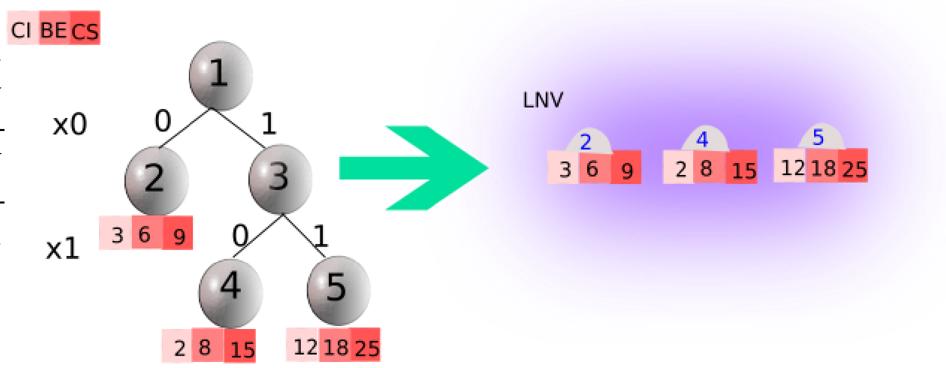
La estrategia de ramificación especifica como se recorre el árbol de soluciones. Así tenemos diferentes opciones: 1) hacer un recorrido primero en profundidad, 2) hacer un recorrido primero en anchura o 3) hacer un recorrido priorizando los nodos de mayor o menor beneficio según sea nuestro criterio de optimización.
Para hacer el recorrido se usa una lista de nodos vivos **LNV**: esta lista contiene todos los nodos que han sido generados pero no han sido explorados todavía (se puede seguir ramificando por ellos o se podan). En definitiva estos nodos son los pendientes que el algoritmo tiene que analizar.

Los pasos esenciales del algoritmo **Branch & Bound** son:

1. Sacar un elemento de la **LNV**, si este nodo no se poda
2. Generar sus descendientes
3. Los descendientes que no se podan ponerlos en **LNV**.

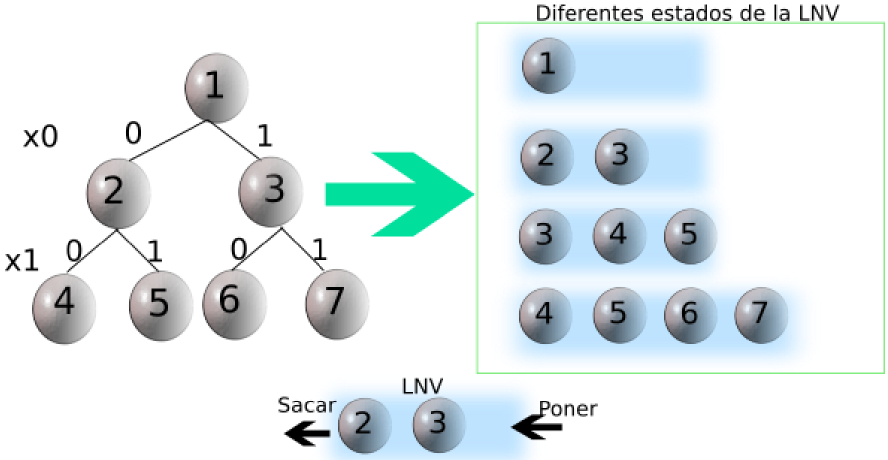
Como se puede apreciar en los pasos del algoritmo la poda puede suceder en dos puntos: 1) al sacar el nodo de la **LNV** o 2) se pueden podar los nodos que se generan como descendientes del último extraido de la **LNV**.

Según como se introduzcan y saquen los nodos de la **LNV** el recorrido será de un tipo u otro.



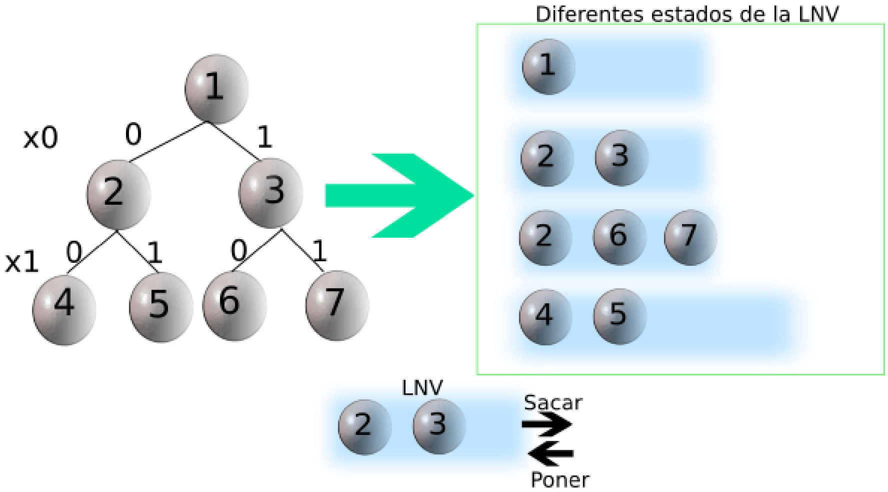
Estrategia de Ramificacion en Anchura

Se hace un recorrido del árbol de soluciones primero en anchura. En este caso la **LNV** se representa como una cola ya que el primero que entra es el primero que sale (política FIFO *first input first output*). En la siguiente imagen se puede ver como se modifica la **LNV** conforme se explora el árbol de soluciones.



Estrategia de Ramificacion en Profundidad

Se hace un recorrido del árbol de soluciones primero en profundidad. En este caso la **LNV** se representa como una pila ya que el último que entró es el primero que sale (política LIFO *last input first output*). En la siguiente imagen se puede ver como se modifica la **LNV** conforme se explora el árbol de soluciones.



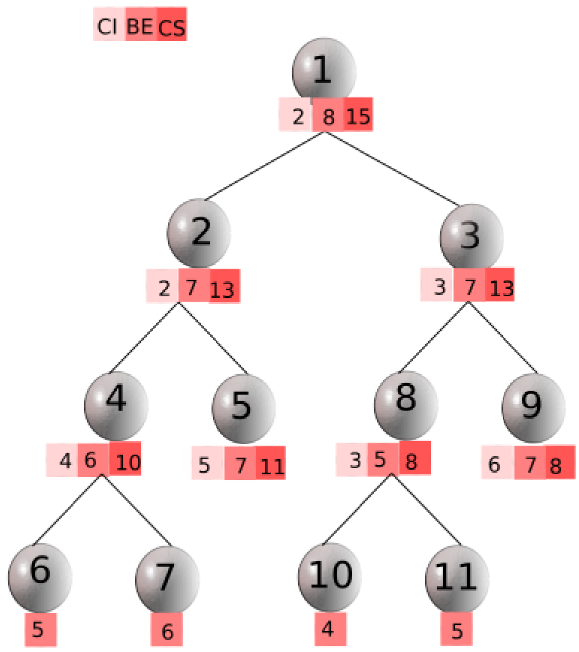
Tanto la estrategia de ramificación en Anchura como en Profundidad hacen una exploración del árbol de soluciones a ciegas sin tener en cuenta los beneficios. Una mejora a estas dos políticas de ramificación es explorar antes por aquellos nodos con mayor beneficio estimado.

Estrategia de Ramificacion LC (least cost)

Esta estrategia ramifica eligiendo de entre todos los nodos de **LNV** el que tenga el mayor beneficio (o menor coste) para explorar a continuación. Cuando diferentes nodos tiene el mismo beneficio o coste estimado se puede usar la política primero en anchura o primero en profundidad para elegir entre ellos. Por lo tanto para poder implementar esta estrategia tenemos:

- En cada nodo i debemos tener tres valores: $CS(i)$,$CI(i)$ y $BE(i)$.
- Se poda según los valores de CI o CS
- Ramificar según el nodo con mejor valor de BE de la **LNV**

Ejemplo 6.4.1
Supongamos que tenemos entre manos un problema de minimización, con el siguiente árbol de soluciones:



Para cada nodo tenemos una tripleta con cota inferior, beneficio estimado y cota superior: CI, BE, CS. Para resolver la priorización con igual valor de beneficio estimado usaremos una priorización primero en entrar primero en salir (LC-FIFO). En otro caso se prioriza por el nodo de menor beneficio estimado. Como es un problema de minimización la cota C se obtiene como:

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

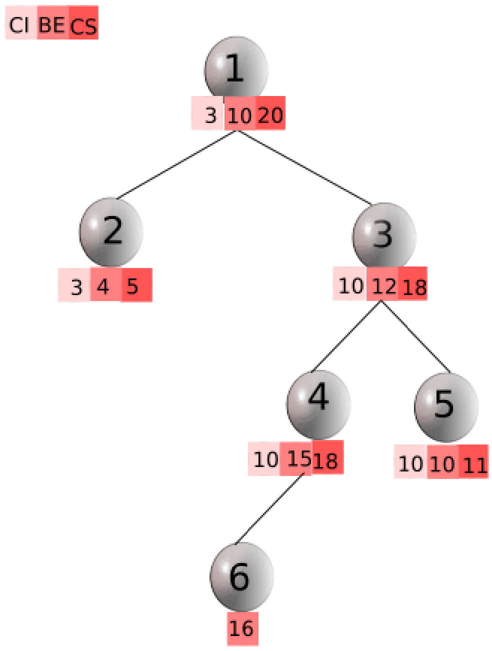
Podar el nodo i si $CI(i) \geq C$.
A continuación para cada iteración del algoritmo se muestra el estado de **LNV** y el valor de la cota C:

Iteración	*LNV*	*C*	**Comentario**
0	1	15	min{15}
1	2-3	13	min{C=15,{13,13}}
2	4-3-5	10	la LNV se ordena por BE
3	3-5	5	la cota C cambia a 5 por la solucion del nodo 6 si se revisará la LNV se eliminaría el nodo 5
4	8-5	5	
5	5	4	el valor de C se modifica a 4 por la solucion del nodo 10 el nodo 9 no se introduce ya que tiene CI=6>C
6		4	

□

Normalmente para obtener las cotas se usan algoritmos basados en la técnica voraz.

Ejemplo 6.4.2
Supongamos que tenemos entre manos un problema de maximización, con el siguiente árbol de soluciones:



Para cada nodo tenemos una tripleta con cota inferior, beneficio estimado y cota superior: CI, BE, CS. Para resolver la priorización con igual valor de beneficio estimado usaremos una priorización primero en entre primero en salir (LC-FIFO). En otro caso se prioriza por el nodo de menor beneficio estimado. Como es un problema de minimización la cota C se obtiene como:

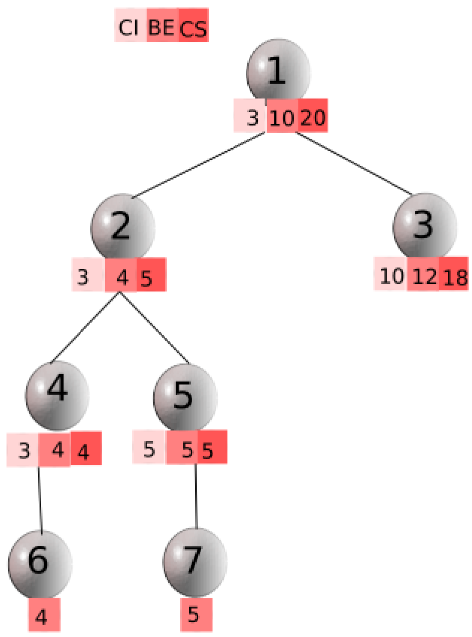
$$C = max\left(\underbrace{\{CI(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Podar el nodo i si $CS(i) \leq C$.
A continuación para cada iteración del algoritmo se muestra el estado de **LNV** y el valor de la cota C:

Iteración	*LNV*	*C*	**Comentario**
0	1	3	
1	3-2	10	Al generar el nodo (2) se mantiene el valor de C a 3. Al genera el nodo 3 se modifica C a 10
2	4-5-2	10	Ahora se sacaría el nodo 4 y se genera su hijo el nodo 6 El nodo 6 es una solución final con valor 16. Como C=10 y es menor que 16 C se modifica a 16
3	5-2	16	Sacamos el nodo 5 ya que CS(5)=11<16 por lo tanto se poda el nodo 5
4	2	16	Igualmente se poda el nodo 2 ya que CS(2)=5<16

□

Ejemplo 6.4.3
Supongamos que tenemos un problema de minimización con el siguiente árbol de soluciones:



Para cada nodo tenemos una tripleta con cota inferior, beneficio estimado y cota superior: CI, BE, CS. Para resolver la priorización con igual valor de beneficio estimado usaremos una priorización primero en entre primero en salir (LC-FIFO). En otro caso se prioriza por el nodo de menor beneficio estimado. Como es un problema de minimización la cota C se obtiene como:

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}} \right)$$

Podar el nodo i si $CI(i) \geq C$.
A continuación para cada iteración del algoritmo se muestra el estado de **LNV** y el valor de la cota C:

Iteración	*LNV*	*C*	**Comentario**
0	1	20	
1	2	5	Al generar el nodo 2 se obtiene C=min{C,CS(2)}=5 Al generar el nodo 3 vemos que CI(3)=10 y es mayor que 5. Por lo tanto el nodo 3 se poda directamente
2	4	4	Al generar el nodo 4 se obtiene C=min{C,CS(4)}=4 Al generar el nodo 5 vemos que CI(5)=5 y es mayor que 4. Por lo tanto el nodo 5 se poda directamente
3		4	

□

6.5 Arboles de Soluciones

Los tipos de árboles de soluciones en Branch & Bound son los mismos que hemos visto en Backtracking: A. Binario, A. Permutacionales, A.n-arios y A. Combinatorios. La representación es la misma:

```
class ABinario{ //sirve para A.nario Apermutacional
  private:
     vector<int> datos;
     int level;
     int valor_inicial; //0: A.permutacional -1:A.binario
     int k; // valor maximo que adopta cada dato[i]
     ....
}
```

Para poder resolver nuestros problemas usando Branch & Bound tenemos que ampliar la clase añadiendo las siguiente funciones:

- *GeneraSiguienteenAnchura*: este método genera el siguiente nodo en Anchura. Por ejemplo si **datos** contiene | 0 | 1 | 0 | teniendo *level*=*2*. Este método actualiza **datos** a | 0 | 1 | 1 |. Además devuelve true. Si no tiene un siguiente en Anchura devuelve false
- *GeneraHijos*: Este método devuelve todos los vectores que se corresponden con los nodos hijos del nodo actual.
- *EsPadre*: Dado un árbol (ya sea binario,kario o permutacional) devuelve si el nodo en el árbol que apunta this es padre del nodo que representa el árbol dado como parámetros.

Veamos el código del método *GeneraSiguienteenAnchura*:

```
bool ABinario::GeneraSiguienteAnchura(){
  int nivel=(int)level;

   while (nivel>=0  && !MasHermanos(nivel)  ){
              datos[nivel]=valor_inicial; // se va al nivel anterior
              nivel=nivel-1;
   }
   if (nivel<0){
    if (level<(int)datos.size()-1){
      level=level+1;
      nivel=0;
    }
    else
    return false;
   }

  do{
       GenerarSiguiente(nivel); //datos[nivel]++

       if (EsSecuencia(nivel)){// nivel ha alcanzado level
                               // es secuencia cuando se genera
                               //cadenas de longitud level
          return true;
       }
       if ( PosibleSecuencia(nivel)) //la longitud de la secuencia
                                      // es menor que level
           nivel=nivel+1;
       else{
          while (nivel>=0  && !MasHermanos(nivel)  ){
           //buscamos otra combinacion
              datos[nivel]=valor_inicial;
              nivel=nivel-1;
         }
         if (nivel<0){
            if (level<(int)datos.size()-1){
              level=level+1;
              nivel=0;
            }
         }
       }

   }while (nivel>=0 );
```

```
    return false;
}
```

Fijado un level se genera todas las secuencias de longitud level primero, antes de pasar a generar una secuencia de longitud mayor. A continuación veamos la implementación de la función *EsPadre*

```
bool ABinario::Espadre(ABinario &v){

  if (level>=v.level) return false;

  for (int i=0;i<=level;i++)
   if  (datos[i]!=v.datos[i]) return false;
  return true;
}
```

Para que una secuencia sea padre de otra su longitud debe ser menor (en particular un caracter menos). Y deben coincidir en estos caracteres. La función para obtener todos los nodos hijos es la siguiente:

```
 vector<vector<int> > ABinario::GeneraHijos(){
   vector< vector< int> > out;
      ABinario vaux = *this;
      if (vaux.GeneraSiguienteProfundidad()){
         int nivel = vaux.Nivel();
        bool seguir=true;

        while(seguir && nivel==level+1 && Espadre(vaux)) {
         out.push_back(vaux.GetSecuencia());

         seguir=vaux.GeneraSiguienteAnchura();
         if (seguir){
           nivel = vaux.Nivel();
        }
      }
      }
      return out;
}
```

6.6 Esquema Branch & Bound

A continuación vamos a describir los pasos esenciales en un algoritmo basado en la técnica Branch & Bound. En primer lugar los nodos que se almacenan en la lista de nodos vivos **LNV** tiene la siguiente representación:

```
struct nodo{
 ABinario v; // o Akario, Apermutacional
 int benefi; //beneficio real hasta el nodo
```

```
 int CI,BE,CS; //cotas inferior y superior
               // y beneficio estimado
};
```

Las variables que necesitamos siempre son:

```
priority_queue<nodo> pq; //LNV
nodo actual; //nodo actual.
int C; //cota de poda
nodo s; //solucion actual
```

La cota de poda C si es un problema de maximización se inicia al mayor CI de los nodos hijos de la raíz. Si es un problema de minimización C se inicia al menor CS de los nodos hijos de la raíz.
El algoritmo en términos generales sería el siguiente:

Algorithm 5 Branch & Bound

procedure BRANCH_BOUND()
  Crear un A Binario de longitud n
  Obtener cotas para el nodo que no contiene ninguna elección aún
  Iniciar C como CI
  Generar todos los hijos de este A.Binario y ponerlos en la cola de prioridad pq
  Comprobar si C se modifica con alguna de las cotas inferiores de los hijos
  repetir
   $actual \leftarrow$ Sacar el nodo de pq
   $pq \leftarrow pq - \{actual\}$
   Si actual.CS>=C
    para cada y hijo de actual
     Si y.benefi>s.benefi
      $s \leftarrow y$
      $C \leftarrow max(C, y.benefi)$
     Si y.CS>C AND y no es hoja
      $pq \leftarrow pq + \{y\}$
      $C \leftarrow max(C, y.CI)$
  while (!pq.empty())

En el algoritmo s es la mejor solución obtenida. En caso de que aún no hemos obtenido una mejor solución entonces s.benefi es $-\infty$.

6.7 Problema: La mochila 0/1

En esta sección vamos a resolver el problema de la mochila con n objetos diferentes, con un mochila de capacidad M, de forma que cada objeto se coge entero o no se coge.
Datos del problema.-

- n:numero de objetos
- M:capacidad de la mochila

- $p = (p_1, p_2, \cdots, p_n)$ pesos de los objetos.
- $b = (b_1, b_2, \cdots, b_n)$ beneficio de los objetos.

Formulación Matemática.-

$$Maximizar \sum_{i=1}^{n} x_i \cdot b_i \text{ sujeto a } \sum_{i=1}^{n} x_i \cdot p_i \leq M \text{ con } x_i \in \{0,1\}$$

Representación de la Solución.-

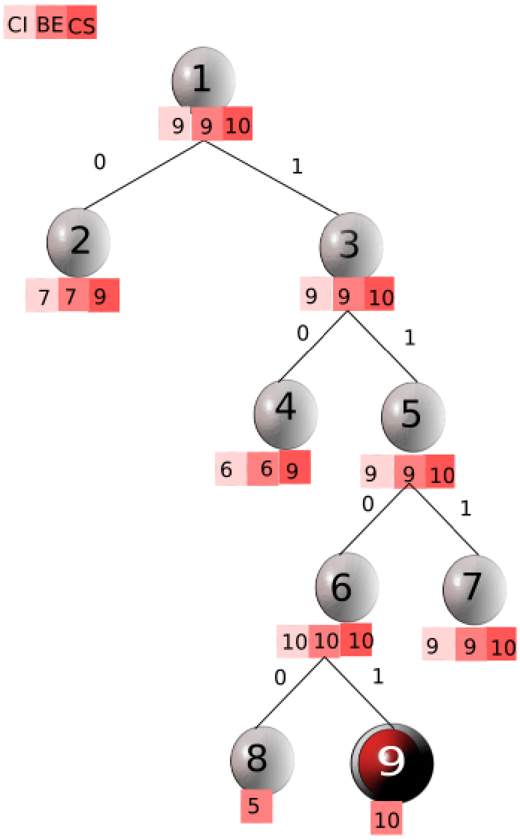
$$s = (x_1, x_2, \cdots, x_n) \text{ con } x_i \in \{0,1\} \text{ si } x_i = 0 \text{ no se coge el objeto } x_i = 1 \text{ se coge}$$

Cálculo de las cotas y beneficio estimado.- Para obtener las cotas vamos a usar la mochila voraz. Un cota inferior la da la mochila voraz 0|1. El beneficio de la solución que de la mochila voraz 0|1 es menor o igual que el beneficio óptimo, ya que los objetos no se pueden partir. Y una cota superior la da la mochila voraz no 0|1 (los objetos se pueden partir) con criterio: escoger por mayor razón beneficio peso. Las cotas por lo tanto se calculan como:

$$CS = b_{act} + \lfloor MOCHILA\_Vorazno01(\text{objetos restantes, resto de Mochila}) \rfloor$$
$$CI = b_{act} + \lceil MOCHILA\_Voraz01(\text{objetos restantes, resto de Mochila}) \rceil$$
$$BE = CI$$

Siendo b_{act} el beneficio acumulado por lo objetos que se han decidido escoger. En este problema el beneficio estimado será igual a la cota inferior. Otra posibilidad podría ser obtener el promedio de CS y CI.

Ejemplo 6.7.1
Supongamos que tenemos una mochila M=7 con 4 objetos, es decir n=4. El vector de pesos es $p = (1,2,3,4)$ y el vector de beneficios es $b = (2,3,4,5)$. En la siguiente imagen tenemos el árbol de soluciones que explora el algoritmo. En cada nodo tenemos la tripleta correspondiente a cota inferior, beneficio estimado y cota superior. El nodo 9 representa la solución final y óptima dando lugar a un beneficio máximo de 10, y el vector solución es $s = (1,1,0,1)$, indicando que se cogen los objetos 1,2 y 4.

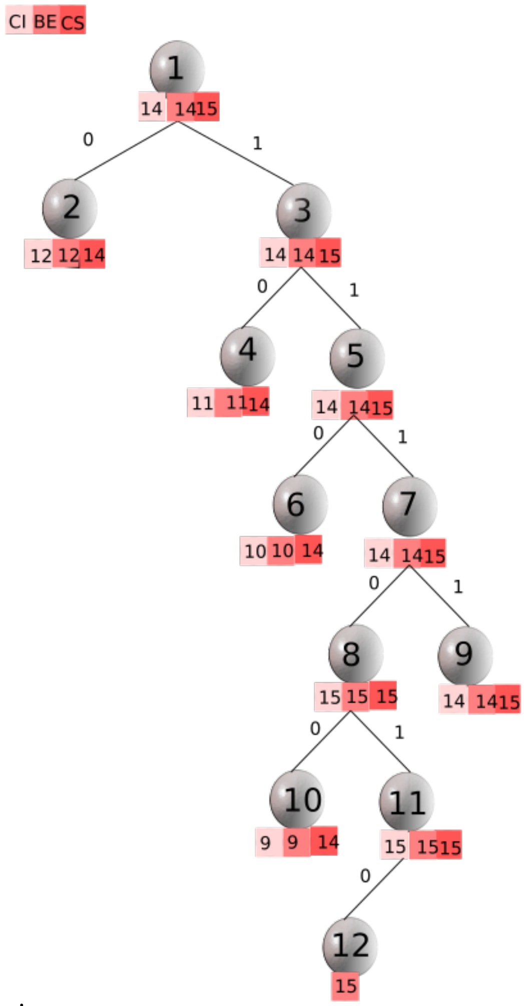


Los pasos que da el algoritmo son los siguientes:

Iteración	LNV	C	Comentario
0	1	9	
1	3	9	Se genera el nodo 2 que tiene una cota superior de 9 y por lo tanto se poda Se genera el nodo 3 que tiene CS=10 y por lo tanto se pone en **LNV**
2	5	9	Se genera el nodo 4 que tiene una cota superior de 9 y por lo tanto se poda Se genera el nodo 5 que tiene CS=10 y por lo tanto se pone en **LNV**
3	6	10	Se genera el nodo 6 que tiene una cota superior de 10 y se pone en LNV también se modifica C a 10. Se genera el nodo 7 que tiene CS=10 y se poda.
4		10	Se genera el nodo 8 que un nodo solución final con valor 5. Luego C no se modifica. Se genera el nodo 9 que tiene valor final 10 y es la solución.

□

Ejemplo 6.7.2



Supongamos que tenemos una mochila M=11 con 6 objetos, es decir n=6. El vector de pesos es $p = (1,2,3,4,5,6)$ y el vector de beneficios es $b = (2,3,4,5,6,7)$. En la siguiente imagen tenemos el árbol de soluciones que explora el algoritmo. En cada nodo tenemos la tripleta correspondiente a cota inferior, beneficio estimado y cota superior. El nodo 9 representa la solución final y óptima dando lugar a un beneficio máximo de 15, y el vector solución es $s = (1,1,1,0,1,0)$, indicando que se cogen los objetos 1,2,3 y 5.

Los pasos que da el algoritmo son los siguientes:

Iteración	**LNV**	**C**	**Comentario**
0	1	14	
1	3	14	Se genera el nodo 2 que tiene una cota superior de 14 y se poda Se genera el nodo 3 que tiene CS=15 y por lo tanto se pone en **LNV**
2	5	14	Se genera el nodo 4 que tiene una cota superior de 14 y se poda Se genera el nodo 5 que tiene CS=15 y por lo tanto se pone en **LNV**
3	7	14	Se genera el nodo 6 que tiene una cota superior de 14 y se poda Se genera el nodo 7 que tiene CS=15 y por lo tanto se pone en **LNV**
4	8	15	Se genera el nodo 8 que tiene una cota superior de 15 se pone en **LNV** Ademas C se modifica a 15 Se genera el nodo 9 que tiene CS=15 y por lo tanto se poda
5	11	15	Se genera el nodo 10 que tiene una cota superior de 14 y se poda Se genera el nodo 11 que tiene CS=15 y por lo tanto se pone en **LNV**. Este nodo no se poda ya que es el camino por donde 8 obtuvo CS 15 y aún no tenemos solución final
6		15	Se genera el nodo 12 que es solución final con valor 15

□

Para llevar a cabo la implementación C++ los nodos los vamos a representar de la siguiente forma:

```
/**
 * @brief Estructura que representa un nodo del
          arbol de soluciones
 */
struct nodo{
  ABinario V;//en V.datos tenemos la solucion parcial
  int bact,//beneficio acumulado de la solucion parcial
      pact;//peos acumulado de la solucion parcial
  int CI,BE,CS;//cotas y beneficio estimado
  //constructor por parametros
  nodo (ABinario &p, int b,int pes, int ci,int be,int cs)
       :V(p),bact(b),pact(pes),CI(ci),BE(be),CS(cs){}
};
```

Para usar la cola de prioridad en el algoritmo Branch & Bound necesitamos sobrecargar el operador < para dos nodos. Además para poder visualizar los datos de un nodo también hemos sobrecargado el operador de salida.

```
/**
 @brief Comparacion entre dos nodos
 **/
bool operator<(const nodo & n1, const nodo &n2){
   if (n1.BE<n2.BE) return true;
   else return false;
}
/**
 * @brief FLujos de salida para nodo
 */
ostream & operator<<(ostream & os, const nodo &O){
   os<<"Nodo: "<<O.V<<" beneficio: "<<O.bact<<
       " peso "<<O.pact<< " CI= "<<O.CI<<" BE="<<O.BE<<
       "  CS= "<<O.CS<<endl;;
   return os;
}
```

A continuación veamos el código C++ del algoritmo:

```
 /**
 * @brief Algoritmo de Mochila 0/1 aplicando Branch and Bound
 * @param M: capacidad de la mochila
 * @param objetos: todos los objetos.
 * @param Best_objs: la mejor solucion encontrada
 * @return el mejor  beneficio acumulado.
 */
```

```
int Mochila_Branch_and_Bound(unsigned int M,const vector<objeto>&objetos,
                              ABinario &Best_objs){

  ABinario P(objetos.size(),-1,1);

  priority_queue<nodo> pq;


  int best_beneficio=0;
  int bact=0,pact=0;

  //COtas sin considerar aun ningun objeto


  int CS= (Voraz_Mochilano01(0, M,objetos));//beneficio superior
  int BE= Voraz_Mochila01(0, M,objetos)+bact;//beneficio estimado
  int CI= BE;//beneficio inferior
  int C=CI;
  nodo a(P,bact,pact, CI,BE,CS);
  pq.push(a);
  int n_nodos=0;//contabilizar el numero de nodos explorados

  do{

     n_nodos ++;
     //sacamos un nodo de la cola
     a =pq.top();
     pq.pop();

     //si tiene un CS mayor a a la cota

     if (a.CS>=C && (unsigned int)a.pact<M){

      vector<vector<int> > hijos = a.V.GeneraHijos();
      //Generamos los hijos
      for (int i=0;i< (int)hijos.size();i++){
        ABinario H (hijos[i],a.V.Nivel()+1,-1,1);

        bact=ObtainBeneficios(H,objetos);//beneficio del hijo
        pact=ObtainPesos(H,objetos); //peso del hijo
        if ((unsigned int)pact<=M){
          //cotas del hijo
          //beneficio superior
```

```
          CS=bact+ (Voraz_Mochilano01(H.Nivel()+1, M-pact,objetos));
          //beneficio estimado
          BE=bact+ Voraz_Mochila01(H.Nivel()+1, M-pact,objetos);
          CI= BE;//beneficio inferior


          if (bact>best_beneficio){//Si el beneficio del hijo es mayor
                                    //que el mejor obtenido.
            Best_objs=H;
            best_beneficio=bact;
            C=(C<bact)?bact: C;// Modificamos el valor para podar

          }
         if (H.Nivel()<objetos.size()){// Si aun quedan objeto por ver
          if (CS>=C && pact<(int)M){ // Si por esa rama podemos obtener un
                                     //mayor beneficio al limite de poda
                                     //y aun no nos hemos pasado de peso
            //lo ponemos en la cola
            nodo anew (H, bact,pact,CI,BE,CS);

            pq.push(anew);
            C= (C<CI)? CI:C;
          }
          else{
            //NO se hace nada
          }
         }
      }
      }
    }
  }while (!pq.empty());
  cout<<"Nodos recorridos "<<n_nodos<< endl;
  return best_beneficio;
};
```

6.8 Problema: Cambio de Monedas

Dado un conjunto de monedas $\{c_1, c_2, \cdots, c_n\}$, con n monedas, y una cantidad P el objetivo es dar el menor número de monedas, de entre las del conjunto, que sumen le cantidad P. Este problema es un problema de minimización, para resolverlo con la técnica Branch & Bound debemos buscar como definir la cota inferior y cota superior, y a con estas cotas definimos el numero de monedas estimadas para conducir la exploración del árbol de soluciones. El árbol de soluciones analiza en cada nivel que pasa si se escoge cada una

de las monedas. En este sentido cada nodo del árbol vuelve a tener n hijos. Por lo tanto el árbol que necesitamos es n-ario. La solución $S = (x_1, x_2, \cdots, x_m)$ tiene m componentes que se corresponden con los niveles máximos analizados del árbol, y cada $x_i \in \{1, 2, \cdots, n\}$, especifica el tipo de moneda que se ha escogido en el nivel i.
Al ser un problema de minimización tenemos que recordar que la cota se obtiene como:

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall \, s\text{que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Y aplicaremos la poda sobre el nodo i si $CI(i) \geq C$.
La cota superior, CS debe de dar un número de monedas mayor o igual al número óptimo de monedas. Para asegurar que este número será mayor o igual una cota posible sería:

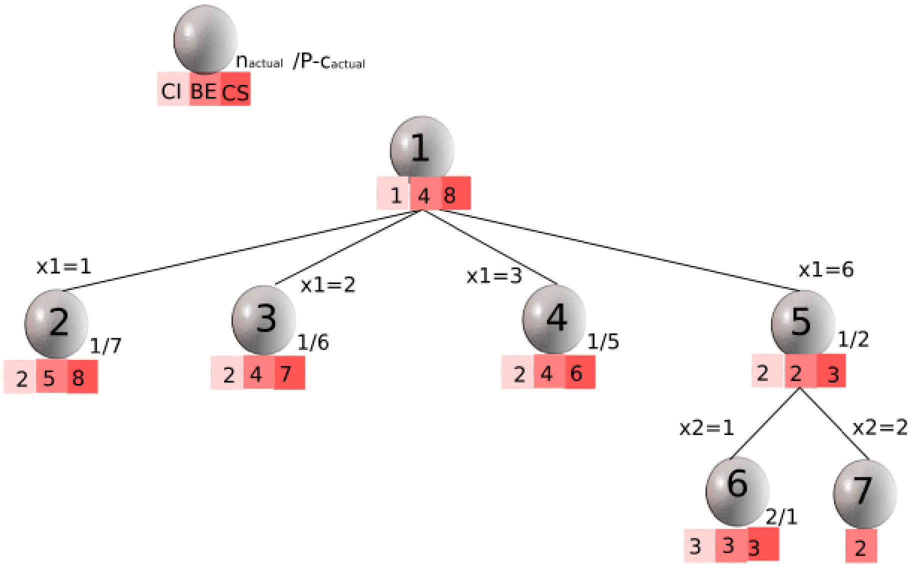
$$CS = n_{actual} + \left\lceil \frac{P - c_{actual}}{min\{c_i\}} \right\rceil$$

Siendo n_{actual} el número de monedas que se necesitan con la solución parcial ya establecidas, y c_{actual} es el valor acumulado de las monedas ya usadas en la solución parcial.
De la misma forma, la cota inferior tiene que ser un valor menor o igual al valor óptimo que se debe alcanzar. En este caso para dar una cota inferior usaremos el numero actual de monedas y le sumaremos 1.

$$CI = n_{actual} + \left\lfloor \frac{P - c_{actual}}{P - c_{actual}} \right\rfloor$$

El numero de monedas estimadas, BE, en este caso lo definimos como $BE = \frac{CI+CS}{2}$.

Ejemplo 6.8.1
Supongamos que disponemos de cuatro monedas con valores $\{1, 2, 3, 6\}$. La cantidad que queremos devolver es $P = 8$.
El árbol de soluciones que se explora se muestra en la siguiente imagen:

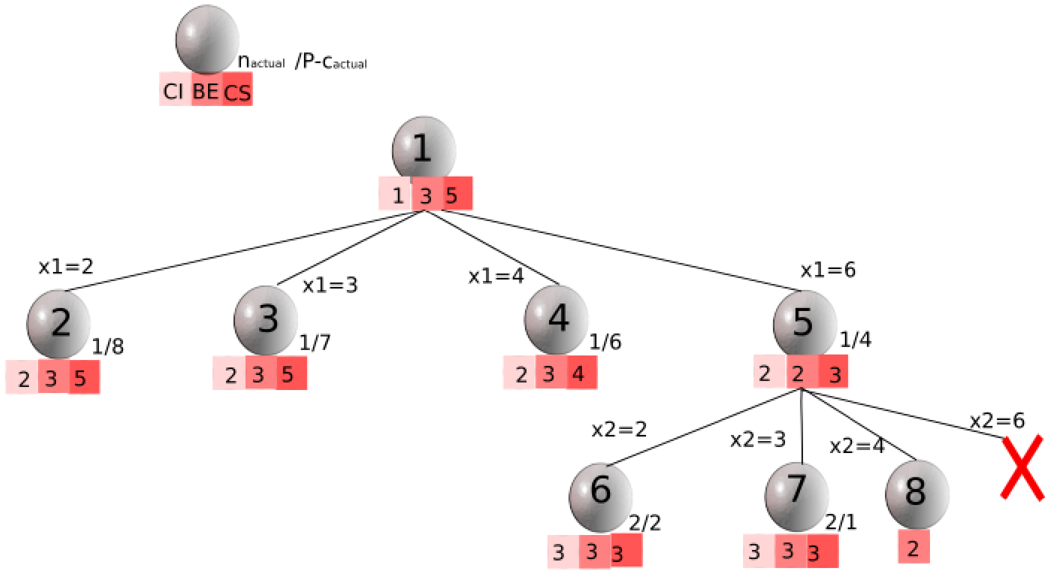


Los pasos que da el algoritmo son los siguientes:

Iteración	LNV	C	Comentario
0	1	8	
1	2		
2	3-2	7	Se genera el nodo 3 que tiene CS=7 y CI=2. Como la CI es menor que C no se poda. Además C se modifica a 7
3	3-2-4	6	Se genera el nodo 4 que tiene CS=6 y CI=2. Como la CI es menor que C no se poda. Además C se modifica a 6
4	5-3-2-4	3	Se genera el nodo 5 que tiene CS=3 y CI=2. Como la CI es menor que C no se poda. Además C se modifica a 3
5	3-2-4	2	Se genera el nodo 6 que tiene CS=3 y CI=3. Como la CI es igual que C se poda. Se genera el nodo 7 que es solución final con valor 2 monedas y C se modifica a 2
6		2	Se saca y podan todos los nodos de la **LNV** ya que su CI es mayor o igual a 2

□

Ejemplo 6.8.2
Supongamos que disponemos de cuatro monedas con valores $\{2,3,4,6\}$. La cantidad que queremos devolver es $P = 10$.
El árbol de soluciones que se explora se muestra en la siguiente imagen:



Suponiendo que nuestra política es LNV LC-FIFO, los pasos que da el algoritmo son los siguientes:

Iteración	**LNV**	C	**Comentario**
0	1	5	
1	2		
2	3-2		
3	4-3-2	4	Se genera el nodo 4 que tiene CS=4 y CI=2. Como la CI es menor (igual) que C no se poda. Además C se modifica a 4
4	5-4-3-2	3	Se genera el nodo 5 que tiene CS=3 y CI=2. Como la CI es menor que C no se poda. Además C se modifica a 3
5	7-6-4-3-2	2	Se genera el nodo 6 que tiene CS=3 y CI=3. Como la CI es menor igual se pone en **LNV**. Se genera el nodo 7 que tiene CS=3 y CI=2. Como la CI es menor igual se pone en **LNV**. Se genera el nodo 8 que es solución final con valor 2. C se modifica a 2
6		2	Se sacan y podan todos los nodos de la **LNV** ya que su CI es mayor o igual a 2

□

Veamos el código C++ de este algoritmo

```
/**
 *NO USA VORAZ porque no da una cota razonable
 * @brief Algoritmo cambio de monedas aplicando Branch & Bound
 * @param monedas: valores de las monedas ordenadas de mayor a menor
 * @param P: cantidad a devolver
 * @param Best_objs: la mejor solucion
```

```
  * @return el menor numero de monedas
  */

int Cambio_Branch_and_Bound(const vector<int> & monedas, unsigned int P,
                            ABinario &Best_objs){

  priority_queue<nodo> pq;
  int num_monedas=monedas.size();

  unsigned int best_sol=numeric_limits<unsigned int>::max();

  nodo n;
   //caso trivial cantidad 0
   //o la cantidad no alcanza el
   //valor de la menor moneda
  if (P==0 || P<monedas[0]) return 0;

  int CS= (int)(ceil((double)P/monedas[0]));//maximo numero de monedas
  int CI= 1;//numero de monedas minimo
  int BE= (CS+CI)/2;
  n.CS=CS;
  //El arbol tendra como maximo CS niveles.
  n.V=ABinario(CS, 0,num_monedas);
  //el nodo raiiz
  n.CI= CI;
  n.BE=BE;
  n.na=0;
  n.ca=0;
  pq.push(n);
  unsigned int C=n.CS;//cota de poda
  int n_nodos=0;
  do{
     n_nodos ++;
     n =pq.top(); pq.pop();

     //alcanza P
     if ((int)P-(int)n.ca==0){
        if (best_sol>n.na){//es mejor solucion
            best_sol=n.na;
            Best_objs=n.V;
            if (C>n.na) C=n.na;
        }
     }
```

```
      else{
        vector<vector<int> > hijos = n.V.GeneraHijos();
        for (unsigned int i=0;i<hijos.size();i++){
           nodo h; h.na=n.na+1; h.ca=ObtainSuma(hijos[i],monedas);

           if ((int)P-(int)h.ca>=0){
             h.CS=h.na+(int)ceil((double)((int)P-(int)h.ca)/monedas[0]);

             h.V=ABinario(hijos[i],GetLevel(hijos[i],0),0,num_monedas);
             h.CI= h.na+1;
             h.BE = (h.CI+h.CS)/2;

             if ((int)P-(int)h.ca==0){//alcanza P
                 if (best_sol>h.na){//es mejor solucion
                     best_sol=h.na;
                     Best_objs=h.V;
                     if (C>h.na) C=h.na;
                 }
             }
             else
              if (C>h.CI){
                if (C>h.CS) C=h.CS;
                pq.push(h);
             }
           }
        }
      }


  }while (!pq.empty());
  cout<<"Nodos recorridos "<<n_nodos<< endl;
  return best_sol;

}
```

La función GetLevel devuelve el nivel de un hijo (dado por un vector), de la siguiente forma

```
int GetLevel(const vector<int> & v,  int vi){
int i=0;
int n= v.size();
while(i<n && v[i]!=vi)
  i++;
i--;
return i;
```

```
}
```

En este problema se usa Anario que es igual que ABinario excepto porque los valores que almacena están en el rango $\{1,2,\cdots,n\}$ y el valor de inicio es 0. De esta forma un nodo viene representado por:

```
struct nodo{
  Anario V;
  unsigned int na,//numero de monedas
               ca;//cantidad acumulada
//   unsigned int CI,BE,CS;//cotas y numero de monedas estimadas

};
```

6.9 Problema: Asignación de Tareas

En este problema el objetivo es dado n trabajadores y n tareas obtener la mejor asignación trabjador-tarea (asignación 1 a 1) que produzca el mejor beneficio acumulado. La asignación de un trabajador a una tarea produce un beneficio, por lo tanto nuestro objetivo es buscar la mejor asignación que obtenga el mayor beneficio global.
Datos del Problema.-

- n: número de trabajadores y de tareas diponibles
- B: matriz indicando para un trabjador i (fila) y una tarea j (columna) el beneficio que se obtiene, almacenado en $B(i,j)$

Formulación matemática.-

$$Maximizar \sum_{i=1}^{n} B(i,s(i)) \text{sujeto a que si } i \neq j \Longrightarrow s(i) \neq s(j)$$

Representación de la solución.- El vector solución será $S=(t_1,t_2,\cdots,t_n)$ siendo t_i la tarea asignada al trabajador i. El árbol de soluciones que debe explorar el algoritmo Branch & Bound es una árbol permutacional ya que a dos poersonas diferentes no se le pueden asignar la misma tarea.
Definición de la Cotas y Beneficio Estimado.- Si recordamos el algoritmo voraz para este problema, asignaba a cada trabjador libre de entre las tareas libres la que mayor beneficio obtenía. Este algoritmo daba una solución aproximada en general, por lo tanto el beneficio que obtenga el algoritmo voraz será un valor menor o igual al beneficio que se obtiene con la solución óptima. Por otro lado una cota superior a este valor óptimo se obtiene si a los trabajadores aún libres se le asigna la tarea libre con mayor beneficio, aunque esta asignación de trabajadores libres tareas libres haya repeticiones. Esto quiere decir que podemos asignar a dos trabajadores la misma tarea si es con la que se obtiene el mayor beneficio. El beneficio estimado será un promedio de la cota superior y la cota inferior. Así

la formulación de las cotas y beneficio estimado es:

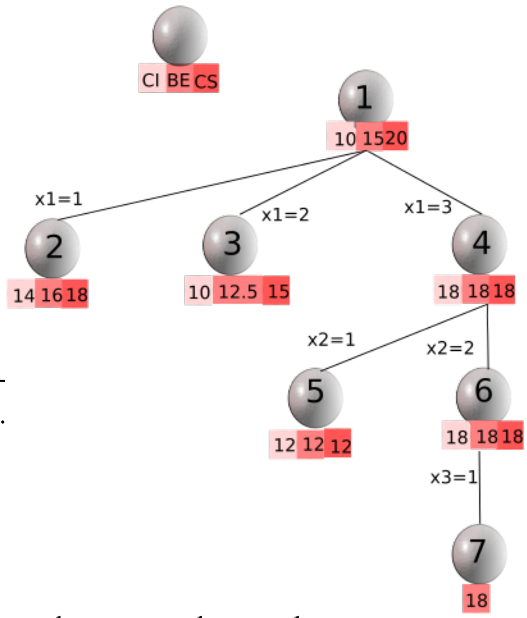
$$CI \leftarrow b_{act} + Asignacion\_Voraz(\text{resto trabajadores, restos tareas})$$

$$CS \leftarrow b_{act} + \text{ Asignar la tarea con mayor beneficio, aunque se repetian entre las libres}$$

$$BE \leftarrow \frac{CI+CS}{2}$$

Ejemplo 6.9.1
Supongamos que en el problema de la asignación con 3 trabajadores y 3 tareas tenemos la siguiente matriz de beneficio:

	Tareas			
		1	**2**	**3**
	1	5	6	4
Trabajadores	**2**	3	8	2
	3	6	5	1



A la derecha se puede ver el árbol de soluciones que explora el algoritmo Branch & Bound.

Al ser un problema de maximización el esquema de cota y poda que adoptamos es:

$$C = max\left(\underbrace{\{CI(j) | \forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s) | \forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Podar el nodo i si $CS(i) \leq C$.
Veamos como se modifica la cota C y la **LNV**.

Iteración	**LNV**	C	**Comentario**
0	1	10	
1	4-2-3	18	Se genera el nodo 2 que tiene CS=18 por lo tanto se introduce en **LNV** Se genera el nodo 3 que tiene CS= 15 y se introduce en **LNV** Se genera el nodo 4 que tiene CS= 18 y se introduce en **LNV** Además C se modifica a 18 que es la CI del nodo 4.
2	6-2-3	18	Se genera el nodo 5 que tiene CS=12 y como C=18 se poda Se genera el nodo 6 que tiene CS= 18 y se introduce en **LNV**
3	2-3	18	Se genera el nodo 7 que es solución
4		18	Se poda los nodos 2 y 3 ya que tienen CS menores a 18

□

A continuación veamos el código C++ del algoritmo Branch & Bound.

```
/**
 * @brief Obtiene el mejor beneficio de asignar a n trabajadores
          n tareas usando branch & bound
 * @param n: el numero de trabajos o trabajadores
 * @param ab: Arbol de permutaciones para obtener la mejor solucion
 * @param B: matriz de beneficios
 * @return el mejor beneficio total
 * */
int Asig_Trabajadores_Branch_Bound(int n, Apermutacion &ab,
                                   const Matriz<unsigned int> &B){
    Apermutacion P(n); //arbol permutacional

    int bact=0; int best_beneficio=0;
    unsigned int nodos_recorridos =0;

    vector<int> aux(n,-1);

    int CS=CotaSuperior(aux,B);;//beneficio superior
    int CI= CotaInferior(aux,B); // beneficio inferior
    int BE; // beneficio optimo estimao
    int C=CI;
    priority_queue<nodo> pq;


    nodo a(P,bact, CI,BE,CS);
    pq.push(a);
```

```
    do{
          nodos_recorridos++;
          nodo a = pq.top(); pq.pop();
          if (a.CS>=C){
             vector<vector<int> > hijos = a.V.GeneraHijos();
              for (int i=0;i< (int)hijos.size();i++){
                Apermutacion H (hijos[i],a.V.GetLevel()+1);

                bact=Suma_Beneficio(H,B);

                aux = ObtainAsignaciones(H,n);
                CS= bact+CotaSuperior(aux,B);//beneficio superior
                CI= bact+CotaInferior(aux,B);
                BE= (CS+CI)/2;
                //Si el beneficio del hijo es mayor que el mejor obtenido.
                if (H.GetLevel()==n-1 && bact>best_beneficio){
                    ab=H;
                    best_beneficio=bact;
                    C=(C<bact)?bact: C;// Modificamos el valor para podar

                }
                // Si no hemos analizado ya todos los trabajadores
                if (H.GetLevel()<n-1)
                // Si por esa rama podemos obtener un mayor beneficio
                  if (CS>=C ){
                    nodo anew (H, bact,CI,BE,CS);
                    pq.push(anew);
                    C= (C<CI)? CI:C;
                  }
              }
            }
   }while (!pq.empty());
    int total=ab.NumeroSecuenciasPosibles();
   cout<<"Numero de nodos recorridos "<<nodos_recorridos<< " total nodos "
       <<total<<" Porcentaje "<<(nodos_recorridos/(double)total)*100.0<<endl;
   return best_beneficio;

}
```

En este caso la representación de nodo que se usa es:

```
 /**
 * @brief Estructura  para establecer la prioridad
           de exploracion del arbol de soluciones
 */
struct nodo{
  Apermutacion V;
  int bact;//beneficio
  int CI,BE,CS;//cotas y beneficio estimado.
  //constructor con parametros
  nodo (Apermutacion &p, int b, int ci,int be,int cs):
         V(p),bact(b),CI(ci),BE(be),CS(cs){}
};

/**
 * @brief Sobrecarga del operado menos para nodo
 */
bool operator<(const nodo & n1, const nodo &n2){
   if (n1.BE<n2.BE) return true;
   else return false;
}
/**
 * @brief FLujos de salida
 */
ostream & operator<<(ostream & os, const nodo &O){
   os<<"Tupla : "<<O.V<<" beneficio: "<<O.bact<< " CI= "<<O.CI
   <<" BE="<<O.BE<< "  CS= "<<O.CS<<endl;;
   return os;
}
```

Las funciones CotaSuperior y CotaInferior son las siguientes:

```
/**
 * @brief Establece una cota inferior del beneficio para los trabajadores
       aun no asignados a ningun trabajo.Para ello aplica la tecnica voraz
 *     entre los trabajadores y trabajos libre.
 * @param asignados: vector con los trabajadores asignados ya a trabajos
 * @param B: matriz de beneficios
 * @return el beneficio estimado de asignar a los trabajadores
        aun no asignados trabajos aun libres.
  Puede que se  repita dicha asignacion
 **/
int CotaInferior(vector<int>asignados,const Matriz<unsigned int> &B){
   Matriz<bool>usados(asignados.size(),asignados.size(),false);
   int n=asignados.size();
   vector<bool>candidatos(n,true);
```

```

   for (unsigned  int i=0;i<asignados.size();i++){
      if(asignados[i]>=0){
       candidatos[i]=false;
       for ( int j=0;j<n;j++)
         usados[j][asignados[i]]=true;
      }
   }
   unsigned int best_bene=0;
   for (int i=0;i<n;i++){
      if (candidatos[i]){ // es un candidatos
        //buscamos entre los trabajos
        //que quedan libres el mas beneficioso
        int mejor=0;
        int work=0;
        for (int j=0;j<n;j++){
            if (usados[i][j]==false)
              if ((int)B.get(i,j)>mejor){
                 work=j;
                 mejor = B.get(i,j);
              }
        }
        for (int t=0;t<n;t++){
          usados[i][t]=true;
          usados[t][work]=true;
        }
        best_bene +=mejor;
      }
   }
   return best_bene;
}
/**
 * @brief Establece una cota superior del beneficio para los
          trabajadores aun no asignados a ningun trabajo.
 * @param asignados: vector con los trabajadores asignados ya a trabajos
 * @param B: matriz de beneficios
 * @return el beneficio estimado de asignar a los trabajadores
     aun no asignados trabajos aun libres.
     Puede que se  repita dicha asignacion
 * */
int CotaSuperior(vector<int>asignados,const Matriz<unsigned int> &B){
   Matriz<bool>usados(asignados.size(),asignados.size(),false);
   int n=asignados.size();
```

```
    vector<bool>candidatos(n,true);
    for (unsigned  int i=0;i<asignados.size();i++){
       if(asignados[i]>=0){
        candidatos[i]=false;
        for ( int j=0;j<n;j++)
          usados[j][asignados[i]]=true;
       }
    }
    unsigned int best_bene=0;
    for (int i=0;i<n;i++){
       if (candidatos[i]){ // es un candidatos
         //buscamos entre los trabajos que quedan libres el mas beneficioso
         int mejor=0;
         //int work=0;
         for (int j=0;j<n;j++){
             if (usados[i][j]==false)
               if ((int)B.get(i,j)>mejor){
                  mejor = B.get(i,j);
               }
         }
         best_bene +=mejor;
       }
    }
    return best_bene;
}
```

6.10 Eficiencia

En general la eficiencia de los algoritmos basados en la técnica Branch & Bound dependen de:

- Número de nodos recorridos.-El algoritmo será más eficiente si expande menos nodos. Cuanto la poda sea mejor la eficiencia será mejor.
- Tiempo que se dedica a cada nodo.- El tiempo dedicado puede contabilizar: 1) Tiempo para obtener la solución parcial hasta ese nodo 2) Calcular Cotas 3) Generar hijos del nodo 4) Gestionar la lista de nodos vivos.

En promedio Branch & Bound obtiene mejores tiempos computacionales que Backtracking. En el peor caso se generan tantos nodos como Backtracking y el tiempo en conjunto puede ser peor si se dedica mucho tiempo en cada nodo.
Por ejemplo en el problema de la *Asignación de Tareas* al tener un árbol Permutacional podemos llegar a generar:

$$\sum_{i=1}^{n} \prod_{j=n-(i-1)}^{n} j = n + n\cdot(n-1) + n\cdot(n-1)\cdot(n-2) + \cdots + n!$$

Por otro lado el tiempo para calcular las cotas tanto para la cota superior como la cota inferior en el peor de los caso es $\mathbf{O}(n)$. Por lo tanto en el peor de los casos tiende a $\mathbf{O}(n \cdot n!)$. En general tenemos que encontrar el equilibrio entre generar buenas cotas para producir el mayor número de podas y por lo tanto el menor números de nodos a explorar. Pero exigir mejores cotas va de la mano de un mayor tiempo computacional para obtener en cada nodo estas cotas.

6.11 Otros Problemas

6.11.1 Problema.-Subconjunto que suma una cantidad

Dado un conjunto de elementos enteros A y una cantidad M el objetivo en este problema es encontrar un subconjunto de A que sume como máximo M.

Ejemplo 6.11.1
Dado el conjunto $A = \{40, 12, 11, 4, 8\}$ y $M = 24$ un subconjunto de A que suman 24 es 4+8+12 □

Formulación Matemática. Sea $X = (x_1, x_2, \cdots, x_n)$ una solución, tal que $x_i \in \{0, 1\}$ representa si el elemento i no se coge ($x_i = 0$) o se coge ($x_i = 1$). Los suma de los elementos escogidos del conjunto no puede superar la cantidad M. De forma matemática podemos expresarlo de la siguiente forma:

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

Para diseñar un algoritmo Branch & Bound usaremos un árbol de soluciones binario, de forma que en cada nivel indicamos si se coge el elemento i o no.

Cotas y Suma estimada Para definir la cota inferior y superior definimos las siguientes funciones:

- $CotaInferior(elementos, cantidad)$: Esta función ordena el conjunto $elementos$ de menor a mayor. Recorre los elementos y si lo coge lo suma siempre y cuando no superemos la cantidad. Por ejemplo si $elementos = \{40, 12, 11, 4, 8\}$ y $cantidad = 24$ esta función ordena los elementos de menor a mayor $\{4, 8, 11, 12, 40\}$ y devuelve 23 que es la suma de $4 + 8 + 11$.
- $CotaSuperior(elementos, cantidad)$: Esta función ordena el conjunto $elementos$ de menor a mayor. Recorre los elementos y para cada elemento si el elemento supera la cantidad restante se coge y se sale de la funcion. En otro caso se coge y se pasa analizar el siguiente elemento. Por ejemplo si $elementos = \{40, 12, 11, 4, 8\}$ y $cantidad = 24$ esta función ordena los elementos de menor a mayor $\{4, 8, 11, 12, 40\}$ y devuelve 35 que es la suma de $4 + 8 + 11 + 12$.

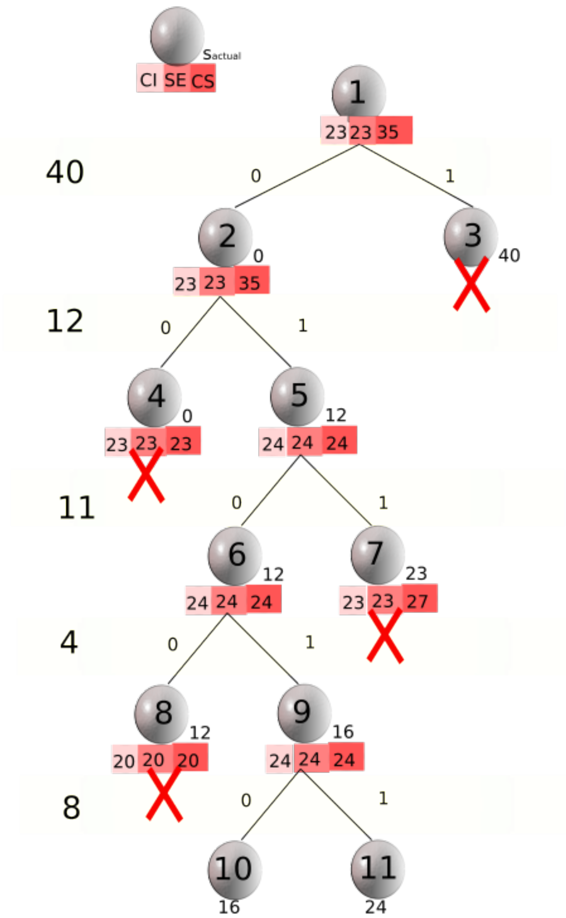
De esta forma la cota inferior y superior se define como:

$$CI = s_{actual} + CotaInferior(\text{resto elementos, cantidad restante})$$
$$CS = s_{actual} + CotaSuperior(\text{resto elementos, cantidad restante})$$

Siendo s_{actual} la suma actual conseguida. Por otro lado la suma estimada la vamos a definir igual a la cota inferior $SE = CI$

Veamos un ejemplo usando estas definiciones.

Ejemplo 6.11.2



Sea el conjunto $A = \{40, 12, 11, 48\}$ y sea $M = 24$. Obtener el subconjunto de A que más aproxime a 24. En la figura a la derecha se puede observar el árbol de soluciones y los nodos que se han podado.

Veamos como se modifica la cota C y la **LNV**.

Iteración	LNV	C	Comentario
0	1	23	
1	2	23	Se genera el nodo 2 en el que no se echa el elemento 40, por lo tanto nuestro $s_{actual} = 0$ Como el nodo 2 tiene un CS de 35 se inserta en la **LNV** El nodo 3 se poda directamente ya que su $s_{actual} = 40$ y sobrepasa la cantidad de 24
2	5-4	24	Se genera el nodo 4 y como tiene CS(4)=23 se inserta en la **LNV** Se genera el nodo 5 y se inserta **LNV** C se modifica a 24
3	6-4-7	24	Se genera el nodo 6 y se inserta en la **LNV** Se genera el nodo 7 y se inserta **LNV**
4	9-4-7	24	Se genra el nodo 8 y se poda ya que CS(8)=20 <24 Se genera el nodo 9 y pone en **LNV**.
5	4-7	24	
6		24	Se saca el nodo 4 que tiene CS=23<C y se poda Se saca el nodo 7 deberiamos expandirlo si no tuviesemos un valor solución exacto a 24 pero como lo tenemos lo podemos podar.

□

Veamos a continuación la implementación C++

```
/**
 @brief Obtiene la suma mas aproximada a un valor
        con elementos de  un conjunto.
 @param M : cantidad a alcanzar
 @param Elementos: conjunto de elementos
 @param best: subconjunto de elementos que aproximan M
 @return el valor que alcanza el mejor subconjunto
*/

int SumaElementos_Branch_Bound(int M, vector<int> &Elementos,
                                ABinario & best){

  ABinario tree(Elementos.size());
  int CS,CI,SE;

  CS=CotaSuperior(Elementos,0,M);
  CI=CotaInferior(Elementos,0,M);

  SE = CI;
  int bestsuma=0;
  nodo n (tree,0,CI,SE,CS);
  priority_queue<nodo>pq;
  pq.push(n);
```

```
  int C= CI;

 do{
  //nodo mas prioritario
  n = pq.top(); pq.pop();
  if ( n.CS>= C  && n.sumaactual<=M){
    //si el nodo tiene un mejor
    //aproximacion a M
    if (bestsuma<n.sumaactual){
        best=n.V;
        bestsuma=n.sumaactual;
        C=(C<bestsuma)?bestsuma:C;
    }
    //generamos los hijos
    vector<vector<int> > hijos = n.V.GeneraHijos();
    for (unsigned int i=0;i<hijos.size();i++){
      nodo H;
      H.V= ABinario(hijos[i],n.V.Nivel()+1);
      H.sumaactual = Suma(H.V,Elementos);
      //la suma actual es menor que M
      if (H.sumaactual<=M){
        //cota superior
        H.CS= H.sumaactual+
              CotaSuperior(Elementos,H.V.Nivel()+1,M-H.sumaactual);
        //cota inferior
        H.CI= H.sumaactual+
              CotaInferior(Elementos,H.V.Nivel()+1,M-H.sumaactual);
        //suma estimada
        H.SE = H.CI;
        //cumple la condicion de poda
        if (C<=H.CS){
          //es mejor solucion
          if (H.sumaactual>bestsuma){
            bestsuma = H.sumaactual;
            best=H.V;
          }
          //se modifica C?
          C=(C<H.CI)?H.CI:C;
          pq.push(H);
        }
      }
    }
  }
```

```
 }while (!pq.empty());
 return bestsuma;
}
```

A continuación detallamos la implementación de las funcions *ContaSuperior* y *CotaInferior*.

```
/**
 @brief Obtiene el valor mas aproximado por debajo
 a un cantidad sumando elementos de un conjunto
 @param Elementos: conjunto de elementos
 @param pos: posicion del primer elemento a analizar en Elementos
 @param M: cantidad  a alcanzar
 @return el valor aproximado a M menor o igual
*/

 int CotaInferior(const vector<int> &Elementos, int pos,int M){
  //consideramos solamente los elementos de pos has el ultimo
  vector<int>aux(Elementos.begin()+pos,Elementos.end());
  //ordenamos de menor a mayor
  sort(aux.begin(),aux.end());
  int suma=0;
  vector<int>::iterator it=aux.begin();
  while (it!=aux.end() && suma<M){
      if (suma+*it<=M)
         suma+=*it;
      ++it;
  }
  return suma;
}

/**
 @brief Obtiene el valor mas aproximado igual o por encima
 a un cantidad sumando elementos de un conjunto
 @param Elementos: conjunto de elementos
 @param pos: posicion del primer elemento a analizar en Elementos
 @param M: cantidad  a alcanzar
 @return el valor aproximado a M mayor o igual
*/

int CotaSuperior(const vector<int> &Elementos, int pos,int M){
  //consideramos solamente los elementos de pos has el ultimo
  vector<int>aux(Elementos.begin()+pos,Elementos.end());
  //ordenamos de menor a mayor
  sort(aux.begin(),aux.end());
```

```
   int suma=0;
   vector<int>::iterator it=aux.begin();
   while (it!=aux.end() && suma<M){
         //si no superamos a M
         if (*it+suma<=M)
          suma+=*it;
         else{
           //si superamos cogemos
           //el elemento y devolvemos
           //esta suma
           suma+=*it;
           return suma;
         }
       ++it;
   }
   return suma;
 }
```

6.11.2 Problema.-El fontanero con penalizaciones

Supongamos que un fontanero tiene N avisos pendientes y cada uno de ellos lleva asociado una duración (días que tardan en realizarse), un plazo límite y una penalización en caso de que no se ejecute en el plazo límite.

Ejemplo 6.11.3
Supongamos que el fontanero tiene cuatro avisos, con los siguiente datos:

	1	2	3	4
Duración	2	1	2	3
Plazo Límite	3	4	4	3
Penalización	5	15	13	10

□

El instante máximo para realizar una tarea a tiempo es $plazo - \text{duración} + 1$.
<u>Técnica Voraz.-</u> Una técnica o heurística es realizar antes aquellas tareas que me van a suponer mayor penalización.
Sea el vector S que contiene la información de la tarea que esoty haciendo en cada día. Si $S[i] = -1$ indica que en ese día estoy libre. En el ejemplo 6.11.3 si $S = (\underbrace{-1}_{\text{día 1}}, \underbrace{-1}_{\text{día 2}}, \underbrace{-1}_{\text{día 3}}, \underbrace{-1}_{\text{día 4}})$, con Penalización $\leftarrow 5 + 15 + 13 + 10 = 43$.
Los pasos que haría la técnica voraz serian los siguientes:

Paso	**Tarea**	**Penalización**	S día1|día2| día3|día4|
1	t_2	43-15=28	$t_2 \vert \; \vert \; \vert \; \vert$
2	t_3	28-13=15	$t_2 \vert t_3 \vert t_3 \vert \; \vert$

Para resolver este problema con Branch & Bound:
Representación de la Solución.- Sea $S = (x_1, x_2, \cdots, x_n)$ donde x_i indica en el día i la tarea que se realiza. Asociado a cada nodo del árbol tenemos un vector *tareas_dias* que indica las tareas que se han fijado ya a dias concretos. Si $tareas\_dias[i] = 0$ indica que en ese dia no se realiza ninguna tarea. Para representar el árbol de soluciones usaremos un árbol de soluciones permutacional.
Cotas.- Para que Branch & Bound funcione adecuadamente debemos definir la cotas de poda. Ya que nuestro objetivo será minimizar la penalización total, el criterio de cota y poda es:

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\, s \text{que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Aplicaremos la poda sobre el nodo i si $CI(i) \geq C$.
Las cotas inferior y superior la vamos a definir como:

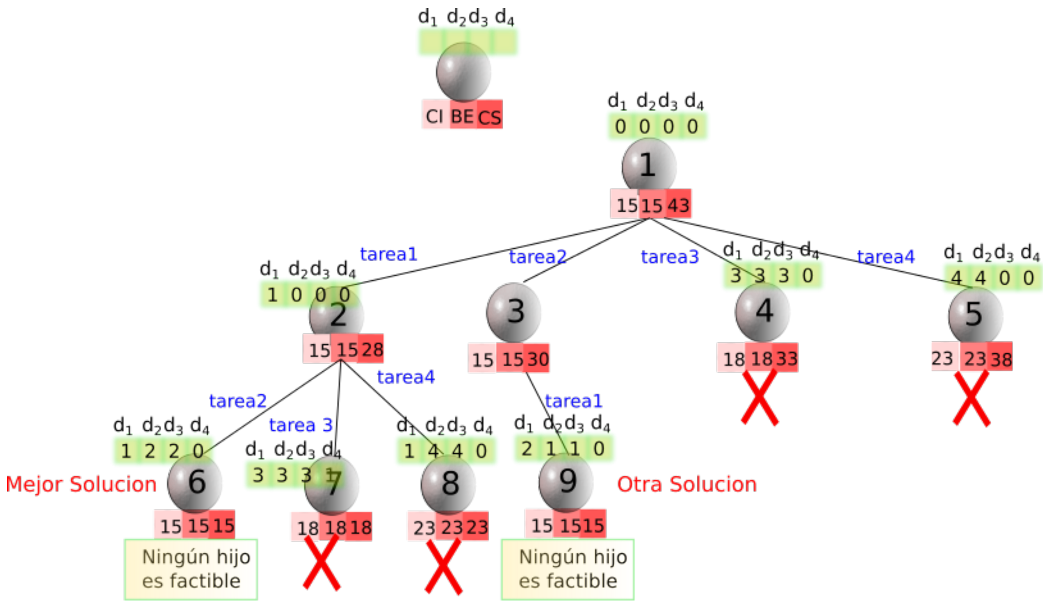
- $CI = \text{Penalización}_{actual} - \text{CotaVoraz(con las tareas que quedan)}$
- $CS = \text{Penalización}_{actual}$
- $BE = CI$

Así nuestra cota inferior vendrá dada por la penalización actual menos la solución que nos de voraces con la las tareas que quedan. Y en este caso nuestra cota superior es la penalización actual. Por otro lado el beneficio estimado se iguala a la cota inferior. Veamos un ejemplo:

Ejemplo 6.11.4
Dadas las tareas siguientes:

	1	2	3	4
Duración	1	2	3	2
Plazo Límite	4	4	3	3
Penalización	15	13	10	5

Decidir que tareas hacer y cuando para minimizar la penalización. El árbol de soluciones que explora el algoritmo Branch & Bound sería el siguiente:



Veamos como se modifica la cota C y la **LNV**.

Iteración	**LNV**	C	**Comentario**
0	1	43	
1	2	28	Se genera el nodo 2 que realiza en el dia 1 la tarea 1 Se modifica C a 28
2	2-3-4-5	28	Se genera el nodo 3 y se inserta en la LNV El nodo 4 y 5 se inserta en LNV
3	3-6-4-5	15	Se saca el nodo 2 y se generan sus hijos El nodo 6 se inserta en LNV y modifica la cota a 15. Además el nodo 6 representa la mejor solución El nodo 7 y 8 se poda
4	6-4-5	15	Se generan los nodos de 3. El único factible es el nodo 9 que es otra solucion. Este nodo se pone en la LNV
5	9-4-5	15	Se saca el nodo 6 y ningun hijo es factible. El nodo 7 y 8 se podan
6		15	Se saca el nodo 9 y ningún hijo es factible. Se saca el nodo 4 y se poda Se saca el nodo 5 y se poda.

□

Ejemplo 6.11.5
Dadas las tareas siguientes:

	1	2	3	4	5
Duración	2	2	1	1	2
Plazo Límite	3	4	4	4	4
Penalización	15	13	13	9	2

El algoritmo voraz, que escoge la tarea entre las candidatas la de mayor penalización y que sea factible hacerla, da lugar a la siguiente solución:

Día	Tarea
1	1
2	1
3	2
4	2
Penalización	24

En cambio si ejecutamos Branch & Bound la solución que obtenemos es:

Día	Tarea
1	1
2	1
3	3
4	4
Penalización	15

Los detalles de los nodos y cotas asociadas que se generan se pueden ver en la siguiente tabla:

Tareas	**CI**	**Penalización Estimada**	**CS**	**Penalización**	**Planning** d1|d2|d3|d4	C
	24	24	52	52	0 0 0 0	52
5	37	37	50	50	5 5 0 0	37
5 2	37	37	37	37	5 5 2 2	37
5 3	28	28	37	37	5 5 3 0	37
5 4	28	28	41	41	5 5 4 0	28
5 3 4	28	28	28	28	5 5 3 4	28
5 4 3	28	28	28	28	5 5 4 3	28
1	24	24	37	37	1 1 0 0	28
1 2	24	24	24	24	1 1 2 2	24
2	17	17	39	39	2 2 0 0	24
2 3	17	17	26	26	2 2 3 0	24
2 4	17	17	30	30	2 2 4 0	17
2 3 4	17	17	17	17	2 2 3 4	17
2 4 3	17	17	17	17	2 2 4 3	17
1 4	15	15	28	28	1 1 4 0	17
1 3	15	15	24	24	1 1 3 0	15
4	15	15	43	43	4 0 0 0	15
1 4 3	15	15	15	15	1 1 4 3	15
1 3 4	15	15	15	15	1 1 3 4	15
4 3	15	15	30	30	4 3 0 0	15
3	15	15	39	39	3 0 0 0	15
4 1	15	15	28	28	4 1 1 0	15
3 1	15	15	24	24	3 1 1 0	15
3 4	15	15	30	30	3 4 0 0	15
4 1 3	15	15	15	15	4 1 1 3	15
3 1 4	15	15	15	15	3 1 1 4	15
4 3 1	15	15	15	15	4 1 1 3	15
3 4 1	15	15	15	15	3 1 1 4	15

En la columna *Planing* un valor cero en el día significa que en ese día aún no tiene asignada ninguna tarea.

□

La implementación usando voraces sería la siguiente:

```
/*
 * @brief Busca un dia a partir del cual se pueda realizar la tarea
 * @param t_d: tareas en que dias se hacen. Y que dias hay libres
 * @param a: tarea sobre la que se busca dias
 * @return dia a partir del cual se puede realizar la tarea.
 * Si no se ha encontrado ninguno se da el dia maximo posible
 **/

```

```
int Busca_dia (const vector<int> & t_d,const tarea &a){
 int dia;
 bool encontrado=false;
 for (int i=1; i<= a.plazo-a.duracion+1 && !encontrado; i++){
     bool salir=false;
     for (int j=i;j<i+a.duracion && !salir; j++)
        if (t_d[j]!=-1) salir=true;
     if (!salir){
         dia =i;
         encontrado=true;
     }
 }

 if (!encontrado)
     dia = a.plazo-a.duracion+1;
 return dia;
}
/************************************************************/
/**
 * @brief Establece si se puede poner una tarea a partir de un dia
        a realizarse
 * @param dia: dia que comienza la tarea
 * @param dia_tarea: dias ocupados por tareas. Es modificado si se
  puede poner
 *  la tarea en el dia.
 * @param T: conjunto de tareas
 * @param t: indice de la tarea a poner
 * @return true si es factible ponerla. false en caso contrario
 */

bool Factible (int dia,  vector<int> & dia_tarea,
              const vector<tarea> &T, int t){
   vector<int> aux (dia_tarea.size(),-1);
   //copiamos hasta dia todo en aux

   for (int i=0;i<dia;i++)
     aux[i]=dia_tarea[i];
   //ponemos la tarea T[t] en aux
   for (int i=dia;i<dia+T[t].duracion;i++)
      aux[i]=t;

   unsigned int inicio =T[t].duracion;
   unsigned int i=dia;
```

```
    //recoloco las que habia despues de dia
    //si se puede
    while (i<dia_tarea.size()){
     if (dia_tarea[i]==-1){
       if (i+inicio<dia_tarea.size()){
         aux[i+inicio]=-1;

       }
       i++;
     }
     else{
       if (i+T[dia_tarea[i]].duracion+inicio-1
           >=dia_tarea.size()) return false;
       if ((int)(i+T[dia_tarea[i]].duracion+inicio-1)
             >(int)T[dia_tarea[i]].plazo) return false;

       for (int j=0;j<T[dia_tarea[i]].duracion;j++)
           aux[i+j+inicio]=dia_tarea[i+j];
       i+=T[dia_tarea[i]].duracion;

     }
    }
    for (unsigned int j=0;j<dia_tarea.size();j++)
      dia_tarea[j]=aux[j];
    return true;

}

/**************************************************************/
/**
 * @brief Obtiene la solucion voraz para decidir la tareas a realizar por
 * el fontanero
 * @param T: conjunto de tareas a realizar. Deben estar ordenadas de
 * mayor a menor penalizacion
 * @return la penalizacion que queda tras la asignacion
 * de tareas a realizar
 */
int Voraz_Fontanero(const vector<tarea> & T,vector<int> &Solucion){
  //Obtenemos el maximo plazo
  int max_pl=T[0].plazo;
  for (unsigned int i=1; i<T.size();i++)
      if (T[i].plazo>max_pl)
         max_pl=T[i].plazo;
```

```
  //Definimos el vector que dice que tarea hacer cada dia
  vector<int>tareas_dias(max_pl+1,-1);

  //Obtengo la penalizacion maxima
  int s_penalizacion=0;
  for (unsigned int i=0;i<T.size();i++)
     s_penalizacion+=T[i].penalizacion;
  //empiezo a realizar las tareas y decidir el dia
  for (unsigned int i=0;i<T.size();i++){
    int dia = Busca_dia(tareas_dias,T[i]);

    if (Factible(dia,tareas_dias,T,i)){
      s_penalizacion-=T[i].penalizacion;
    }
  }
  Solucion=tareas_dias;
  return s_penalizacion;
}
```

Para poder implementar este problema con la técnica Branch & Bound el árbol de soluciones que debemos usar es un árbol que indique en cada nivel la tarea a la que queremos asignarle días. De esta forma la información que contiene un nodo de este árbol es:

```
struct nodo{
   Apermutacion P;//guardar las tareas analizadas
   int CS,BE,CI;//cotas y mejor penalizacion estimada (BE)
   int pa;//penalizacion actual
   vector<int>tareas_dias;//por cada dia que tarea hago
                               //-1 si no hago ninguna

};
```

Para obtener la cota inferior una pequeña modificación que se hace al algoritmo voraz en el siguiente:

```
 /**
 * @brief Obtiene la cota inferior para decidir la tareas a realizar por
 * el fontanero
 * @param T: conjunto de tareas a realizar. Deben estar ordenadas de
 * mayor a menor penalizacion
 * @param tareas_dias: por cada dia la tareas que ya han sido asignada
 * @return la penalizacion que queda tras la asignacion
 * de tareas a realizar
 */
int Cota_Voraz_Fontanero(const vector<tarea> & T,vector<int> &tareas_dias){
  vector<bool> asignadas = Obtain_TareasAsig(T,tareas_dias);
```

```

  //Obtengo la penalizacion maxima
  int s_penalizacion=0;//lo inicio a 0
  //empiezo a realizar las tareas y decidir el dia
  for (unsigned int i=0;i<T.size();i++){
    if (asignadas[i]==false){
      int dia = Busca_dia(tareas_dias,T[i]);
      if (Factible(dia,tareas_dias,T,i)){
        s_penalizacion+=T[i].penalizacion;
      }
    }
  }
  return s_penalizacion;
}
```

El algoritmo Branch & Bound podría ser el siguiente:

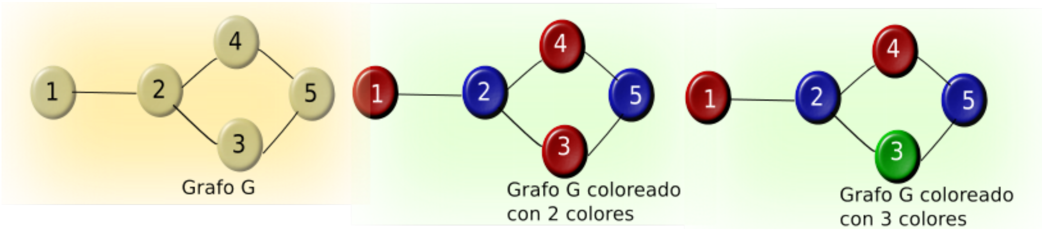
```
/**
 * @brief Obtiene la solucion branch and bound para el problema de
 * las tareas a realizar un fontanero.
 * @param T: conjunto de tareas a realizarion
 * @param solucion: las tareas que se hacen en que dias.
 * @return la penalizacion no completada
 */
int Fontanero_Branch_Bound(vector<tarea>&T, vector<int> &solucion){
 int C;
 nodo n;
 n.pa=0;
 int best_pena;

 //penalizacion total
 for (unsigned int i=0;i<T.size();i++)
    n.pa+=T[i].penalizacion;

 best_pena=n.pa;//menor penalizacion

//numero maximo de dias
 int max_pl=T[0].plazo;
 for (unsigned int i=1;i<T.size();i++)
   if (T[i].plazo>max_pl)
      max_pl=T[i].plazo;

 n.tareas_dias=vector<int>(max_pl+1,-1);
 vector<int> aux = n.tareas_dias;
 n.CS=n.pa;
```

```
n.CI=n.pa-Cota_Voraz_Fontanero(T,aux);
n.BE=n.CI;

priority_queue<nodo>pq;
n.P=Apermutacion(T.size());
C=n.CS;  //cota de poda
pq.push(n);
while (!pq.empty()){
  //sacamos el nodo con menor BE
  n= pq.top(); pq.pop();

  if (n.CI<=C){ //cumple la cota de poda
   if (n.pa<best_pena){ //si tiene la menor penaliazion
     best_pena=n.pa;
     solucion=n.tareas_dias;
     if (C>n.pa) C=n.pa;
   }
    //Genero sus hijos
   vector<vector<int> >hijos=n.P.GeneraHijos();
   for (unsigned int i=0;i<hijos.size();i++){
     int level=n.P.GetLevel()+1;
     int dia = Busca_dia(n.tareas_dias,T[hijos[i][level]-1]);
     nodo h; h.tareas_dias=n.tareas_dias;
     h.P=Apermutacion(hijos[i],level);

     if (Factible(dia,h.tareas_dias,T,hijos[i][level]-1)){

       h.pa = n.pa -T[hijos[i][level]-1].penalizacion;
       h.CS=h.pa;
       vector<int> aux = h.tareas_dias;
       h.CI=h.pa-Cota_Voraz_Fontanero(T,aux);
       h.BE=h.CI;

       if (h.CI<=C){//cumple la cota
         pq.push(h);
         if (h.CS<C){
             C=h.CS;
             best_pena= h.pa;
             solucion= h.tareas_dias;
         }
       }
     }
   }
```

```
    }
  }
  return best_pena;
}
```

6.11.3 Problema.-El coloreo de grafos

Este problema consiste en dado un grafo $G = (V,E)$, V es el conjunto de vértices y E conjunto de aristas, el objetivo es colorear los vértices usando el mínimo número de colores, de forma que dos vértices adyacentes no pueden tener el mismo color. Este problema ya fue visto en el tema de algoritmo voraces (ver sección 3.9.3), y dimo una solución voraz a este prblema. Este problema se clasifica NP, ya que no se puede obtener una solución en tiempo polinomial. Por lo tanto los algoritmos voraces darán soluciones aproximadas, en general.



Como vimos en la sección 3.9.3 existen diferentes heurísticas para establecer el siguiente nodo que se debe colorear:
- Ordenar los vértices por mínimo grado
- Ordenar los vértices por mayor grado
- Escoger el siguiente nodo de forma aleatoria.

En el algoritmo que propusimos en la sección 3.9.3 se suponia que los vértices ya se daban ordenados según un criterio y este simplemente recorría los nodos e iba coloreando. Se creaba un nuevo color si los que ya se habian usado no era posible mantener la restricción de que dos nodos adyacentes no se pueden colorear con el mismo color.

Solución Branch & Bound

Representación de la solución.- Sea $S = (x_1, x_2, \cdots, x_n)$ una solución a nuestro problema, tal que x_i representa el color asignado al vértice i. n representa el número de vértices. Como máximo el número de colores necesarios será n que se da en el caso de que nuestro grafo sea completo (tiene un numero de arista $n * \frac{n-1}{2}$). El menor número de colores será 1 que es cuando el grafo se compone de n componentes conexas. En el caso que nuestro grafo sea un camino el menor número de colores es 2. Por lo tanto para realizar la exploración de nuestro espacio de soluciones necesitamos un árbol n-ario. En cada nivel de árbol expresamos el color que le asignamos al vértice i.

Existen numerosos contextos donde este problema se puede aplicar. Por ejemplo:
- Disponer el menor número de empresas gasolineras en un conjunto de pueblos.
- Definir el menor número de especialidades en una fabrica suponiendo que tiene n grupos de trabajos, y cada grupo de trabajo se puede comunicar con algunos de los

otros grupos.

- Colorear las n regiones de interés en una imagen

Estos serian ejemplos reales donde podemos aplicar la solución a este problema.

Cotas y número de colores estimado.- Tenemos que dar cotas lo suficientemente buenas para que la búsqueda de la solución sea lo más rápida posible, ya que sin las cotas nuestro algoritmos tardaría en el peor de los casos $O(n^n)$ y por lo tanto sería imposible usarlo. Para dar una cota superior tenemos que tener en cuenta que la solución Voraz nos da una cota superior del número exacto de colores que podemos llegar a obtener por un camino. Asi nuestra cota superior se define como:

$$CS \leftarrow \text{colores}_{actuales} + \text{Voraces(resto de nodos)}$$

Una cota inferior podría ser el propio número de colores asignados hasta el momento. Pero esto lo podemos refinar un poco más usando la propiedad de que si el nodo que pretendemos colorear es adyacente a todos los nodos (coloreados y no coloreados) entonces necesitamos un color más, en otro caso simplemente el número de colores asignados. De esta forma la cota inferior la podemos definir como:

$$CI \leftarrow \text{colores}_{actuales} + \underbrace{\text{CotaInferior(restos de nodos)}}_{\substack{\text{por cada nodo no coloreado} \\ \text{si es adyacente a todo los demás} \\ \text{vértices incrementamos en uno el} \\ \text{número de colores}}}$$

Con estas cotas el yq podemos establecer el número estimado de colores necesarios como $BE = \frac{CI+CS}{2}$.

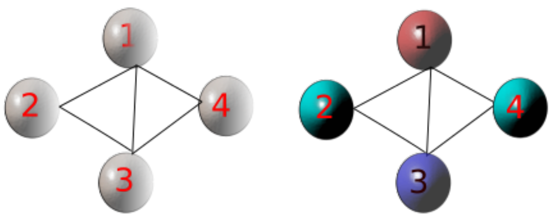
Al se un problema de minimización la cota de poda se define como:

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Y se poda si el nodo i cumple que $CI(i) \geq C$.

Ejemplo 6.11.6

Dado el siguiente grafo (izquierda), el número de colores mínimo necesarios son dos, como se puede observar en el grafo a la derecha.



Como se puede observar en la figura el número de colores necesarios es 3.
Los pasos mas relevantes de nuestro algoritmo Branch & Bound, se pueden observar en la siguiente tabla:

Iter	Colores actuales	Numero de Colores actuales	CI	Colores estimados	CS	C	Best Coloreo	Colores asignados a la solución
0	1	1	2	2	3	3		
1	1 2	2	3	3	4	3		
2	1 4	2	3	3	4	3		
3	1 2 3	3	3	3	4	3		
4	1 2 4	3	3	3	4	3	3	1|2|3|2|
5	1 4 2	3	3	3	4	3	3	1|2|3|2|
6	1 3	2	3	3	4	3	3	1|2|3|2|
7	1 4 3	3	3	3	4	3	3	1|2|3|2|
8	4	1	2	2	3	3	3	1|2|3|2|
9	4 1	2	3	3	4	3	3	1|2|3|2|
10	4 2	2	3	3	4	3	3	1|2|3|2|
11	4 3	2	3	3	4	3	3	1|2|3|2|
12	3	1	2	2	3	3	3	1|2|3|2|
13	3 1	2	3	3	4	3	3	1|2|3|2|
14	3 2	2	3	3	4	3	3	1|2|3|2|
15	3 4	2	3	3	4	3	3	1|2|3|2|
16	2	1	2	2	3	3	3	1|2|3|2|
17	2 1	2	3	3	4	3	3	1|2|3|2|
18	2 3	2	3	3	4	3	3	1|2|3|2|
19	2 4	2	3	3	4	3	3	1|2|3|2|

Analizando la tabla, empezamos en la iteración 0 asignado al vértice 1 el color 1. En la iteración 4 asignamos al vértice 1 el color 1, al vértice 2 el color 2 y al vértice 3 el color 3, ahora asignado al vértice 4 el color 2 obtenemos un número de colores necesarios de 3. Como se puede observar esta mejor solución es la solución que obtiene nuestro algoritmo. □

La implementación del algoritmo Branch Bound necesita la función que implementan las cotas inferior y superior como siguen:

```
/**
@brief Obtiene una cota inferior del numero minimo necesario
para colorear los nodos aun no coloreados de un grafo. Esta
cota suma para cada vertice no coloreado 1 si es adyacente
al resto de nodos 0 en caso contrario.
@param A: grafo que contiene los vertices y aristas
@param no_coloreados: conjunto de vertices no coloreados
@param coloreados: conjunto de  vertices coloreados
@return cota inferior del numero de colores necesarios,
*/

int CotaInferior(Graph &A, vector<unsigned int> &no_coloreados,
                 vector<unsigned int> & coloreados){
  int n_color=0;
```

```
  unsigned int nv = no_coloreados.size();
  for (unsigned int i=0;i<nv;i++){
    //si es adyacente a todos los nodos
    if (Adyacente_a_todos(A,no_coloreados[i],coloreados) &&
    Adyacente_a_todos(A,no_coloreados[i],no_coloreados)){
        n_color++;
    }
  }
  return n_color;
}

/****************************************************************/
/**
@brief Obtiene la cota superior del numero minimo necesario
para colorear los nodos aun no coloreados. Esta se obtiene
dando a cada nodo un color entre los existentes si se puede
en caso contrario se crea un nuevo color
@param A: grafo que contiene los vertices y aristas
@param no_coloreados: conjunto de vertices no coloreados
@return cota superior del numero de colores necesarios,
*/
int CotaSuperior(Graph & A, vector<unsigned int> & no_coloreados){
  unsigned int nv=no_coloreados.size();
  int ncolores=0;
  int total_vertices=A.num_vertices();

  vector<int>result(total_vertices,-1);

  vector<bool>colores_usados(nv,false);
  vector<bool>libre_color(result.size(),true);

  for (unsigned int u=0; u<nv;++u){
    //obtenemos los adyacentes al vertice no coloreado
    Graph::vertex_set out=GetAdyacentes(A,no_coloreados[u]);

    //Por cada uno de los adyacentes
    for (Graph::vertex_set::const_iterator q= out.begin();
         q != out.end(); q++){
      unsigned int v= *q;
      v--;
      if (result[v]!=-1)// esta coloreado ?
        //ese color no esta libre para el vertice
        libre_color[result[v]]=false;
```

```
      }
      //buscamos el primer color libre
      unsigned int cr;
      bool find=false;
      for (cr=0;cr<nv && !find; cr++)
         if (libre_color[cr]){ //esta libre el color?
           //usamos ese color para el vertice
           find=true;
           result[no_coloreados[u]-1]=cr;
         }


      //si no hemos reusado un color
      if (colores_usados[cr]==false){
        //creamos un nuevo color para el vertice
        ncolores++;
        colores_usados[cr]=true;
      }

      //reseteamos los colores libre para el siguiente vertice
      //a colorear
      for (Graph::vertex_set::const_iterator q= out.begin();
           q != out.end(); q++){
         unsigned int v= *q;
         v--;
         if (result[v]!=-1)
           libre_color[result[v]]=true;
      }

    }
    return ncolores;
}
```

La siguiente función indica que vértices están coloreados hasta un vértice menor igual a n.

```
void AsigColoreados(int n, vector<bool>&coloreados){
  for (int i=0;i<=n;i++)
     coloreados[i]=true;
  for (unsigned int i=n+1;i<coloreados.size();i++)
     coloreados[i]=false;

}
```

Cuando asignamos colores a los vértices tenemos que comprobar que es correcto y no incumplimos la restricción de que si dos vértices son adyacentes le asignemos el mismo color. Eso es lo que comprueba la función *Valido_Coloreo*:

```
/**
@brief Comprueba que una asignacion de colores a los vertices
es valida.
@param P:  asignaciones de colores
@param A: grafo con todos los vertices y aristas
@return true si el coloreo es valido. false en caso contrario
*/
bool Valido_Coloreo(Anario &P, Graph &A){


  for ( int i=0; i<=P.GetLevel();i++){

    unsigned int d=i+1;
    Graph::vertex_set out=GetAdyacentes(A,d);

    for (Graph::vertex_set::const_iterator q= out.begin();
        q != out.end(); q++){
       unsigned int v= *q; v--;
       if ((int)v<=P.GetLevel() && i!=v){//si es un vertice ya coloreado?
         if (P[i]==P[v]) return false;
       }
    }
  }
  return true;
}
```

Hay que tener en cuenta que P expresa en para cada vértice i en $P[i]$ el color que se le asigna. La siguiente función contabiliza dada una asignación el número de colores usados para los n vértices.

```
/**
@brief Obtiene el numero de colores usados en una asignacion
@param P : asignaciones de colores a lo vertices
@n: numero de vertices
@return numero de colores usados
*/
unsigned int ObtainColores(Anario &P,int n ){
  Anario:: iterator it;
  vector<bool> colores(n,false);
  int nc=0;
  int i;
  for (i=0, it=P.begin();it!=P.end();++it,++i){
     int col =*it; col--;
     if (colores[col]==false){
```

```
          nc++;
          colores[col]=true;

        }
    }
    return nc;

}
```

Con estas funciones ya podemos describir el código del algoritmo Branch & Bound:

```
/**
@brief Obtiene el coloreo de un grafo usando la tecnica Branch & Bound
@param A: grafo con los vertices y aristas
@param ab: el mejor coloreo
@return numero minimo de colores necesarios
*/

int Colorea_Branch_Bound(Graph &A,Anario &ab){

    vector<bool > coloreados(A.num_vertices(),false);
    vector<unsigned int>ver_color;
    vector<unsigned int>ver_nocolor=ObtainNoColoreados(coloreados);

    int colores_actual, best_ncolors=A.num_vertices()+1;
    int CS=CotaSuperior(A,ver_nocolor);//beneficio superior
    int CI;// cota inferior
    int BE; // beneficio optimo estimao
    int C=CS;
    //cout<<"Cota Inicial "<<CS<<endl;

    priority_queue<nodo> pq;
    bool seguir=true;
    Anario P(A.num_vertices());

    //Insertamos en la cola de prioridad los hijos del
    //nodo raiz
    do{
        AsigColoreados(P.GetLevel(),coloreados);
        ver_color=ObtainColoreados(coloreados);//nodos coloreados
        ver_nocolor= ObtainNoColoreados(coloreados);//nodos no coloreados
        CI=1+ CotaInferior(A,ver_nocolor,ver_color);//cota inferior

        if (CI<=C){
          CS=1+CotaSuperior(A,ver_nocolor);
```

```
          BE=(CS+CI)/2;//numero estimado de colores
          nodo a(P,1,CI,BE,CS);
          pq.push(a);//lo ponemos en la cola

        }
        //vamos al siguiente nodo
        seguir=P.GeneraSiguienteAnchura();
        //si no es un hermano nos salimos
        if (P.GetLevel()>0) seguir=false;
    }while (seguir);

    int nodos_recorridos=0;
    bool solucion=false;


  do{

    nodos_recorridos++;
    nodo a= pq.top(); pq.pop();
    //Si aun no hemos encontrado solucion
    //y se la cota de poda es mayor o igual
    //o tenemos ya una solucion final
    //y l cota es mayor estricta que la cota inferior
   if ((!solucion && a.CI<=C) || (solucion && a.CI<C)){

    //Generamos todos los hijos
    vector<vector<int> > hijos = a.V.GeneraHijos();
    for (int i=0;i< (int)hijos.size();i++){

      Anario H (hijos[i],a.V.GetLevel()+1);
      if (Valido_Coloreo(H,A)){//si es un coloreo valido
        AsigColoreados(H.GetLevel(),coloreados);

        ver_color=ObtainColoreados(coloreados);
        ver_nocolor= ObtainNoColoreados(coloreados);

        colores_actual= ObtainColores(H,A.num_vertices());
        CS= colores_actual+CotaSuperior(A,ver_nocolor);
        CI= colores_actual+CotaInferior(A,ver_nocolor,ver_color);

        BE= (CS+CI)/2;
        //Si hemos coloreado todos lo verties
        // y es mejor solucion
```

```
        if (H.GetLevel()==(int)A.num_vertices()-1
            && colores_actual<best_ncolors){
            ab=H;
            best_ncolors=colores_actual;
            // Modificamos el valor para podar
            C=(C<colores_actual)?C: colores_actual;
            solucion=true;  //ya tenemos una solucion final

        }
        else{
        // Si no hemos analizado ya todos los vertices
        if (H.GetLevel()<(int)A.num_vertices()-1)
          if (CI<=C ){ // Si por esa rama podemos obtener un
                        //un menor numero de colores
            nodo anew (H, colores_actual,CI,BE,CS);
            pq.push(anew);
            C= (C<CS)? C:CS;

          }

        }
      }
    }
  }
  }while (!pq.empty());
  return best_ncolors;
}
```

6.11.4 Problema.-Pasajeros a sus barcos

Tenemos un conjunto de barcos B y un conjunto de tipos de pasajeros P. El coste de que un determinado conjunto de pasajeros p_i viaje en el barco b_j viene dada por la matriz de costos C en $C(i,j)$. El objetivo en este problema es diseñar un algoritmo para asignar los pasajeros a los barcos con el mínimo coste, con la restricción de que cada barco se le asigna a un tipo de pasajero y viceversa (relación uno a uno).

Ejemplo 6.11.7
Dada la siguiente matriz que representa el coste de viajar un tipo de pasajero en un barco,

Coste	b_1	b_2	b_3	b_4
p_1	95	2	55	69
p_2	75	11	89	83
p_3	63	89	9	77
p_4	12	75	82	22

queremos encontrar la mejor asignación para minimizar el coste acumulado.

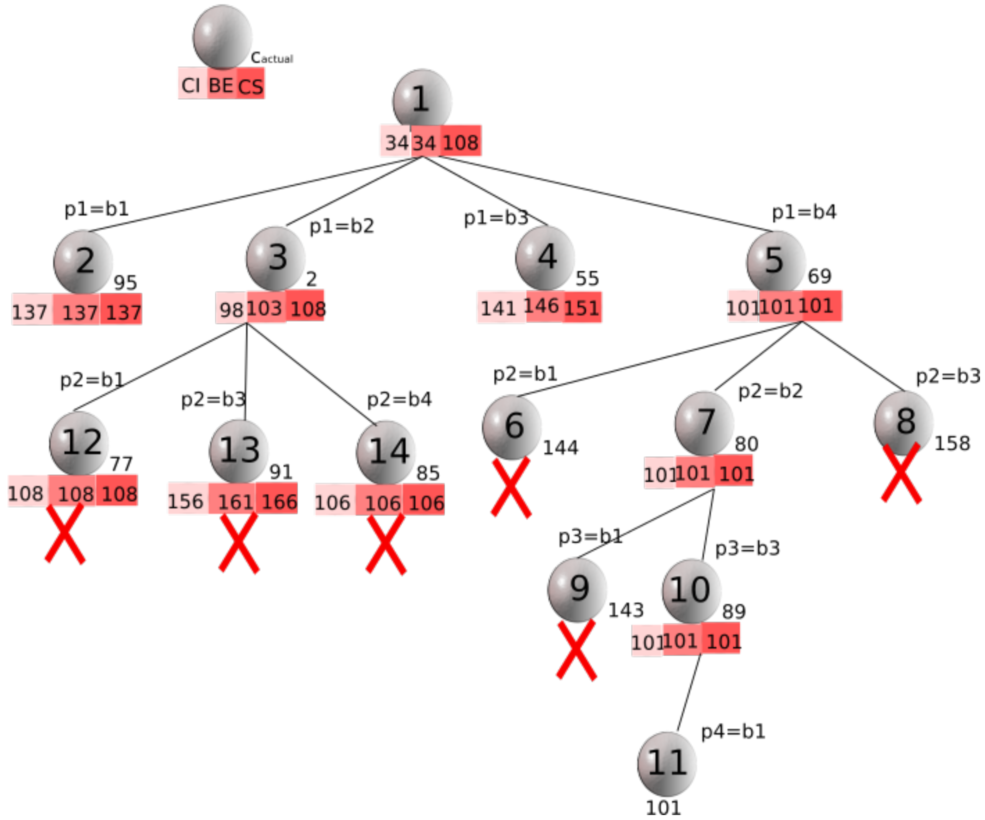
Para llevar a cabo este ejercicio definiremos las cotas como:

- $CI = c_{actual} +$ CotaInferior(pasajeros libres,barcos libres)
- $CS = c_{actual} +$ CotaSuperior(pasajeros libres,barcos libres)

Donde c_{actual} es el coste actual y $CotaInferior$ el coste de asignar a los pasajeros libres un barco libre, el de menor coste, aunque en esta asignación de pasajeros libres haya repeticiones.
Y $CotaSuperior$ es el el coste de asignar a los pasajeros libres un barco libre sin repeticiones, escogiendo para cada pasajero el barco libre que tenga menor coste.
Por otro lado el coste estimado $BE = \frac{CI+CS}{2}$.
Al ser un problema de minimización la cota de poda C se define como

$$C = min\left(\underbrace{\{CS(j)|\forall j \text{ generado }\}}_{\text{Nodos generados}}, \underbrace{\{Valor(s)|\forall\ s \text{ que sea solución final}\}}_{\text{Soluciones finales}}\right)$$

Y se poda si el nodo i cumple que $CI(i) \geq C$.
El árbol de soluciones es el siguiente:



Veamos como se modifica la cota C y la **LNV**.

Iteración	**LNV**	C	**Comentario**
0	1	108	
1	5-3	101	Se genera el nodo 2 que tiene CI=137 mayor que 108 por lo tanto no se introduce en **LNV** Se genera el nodo 3 que tiene CI= 98 y se introduce en **LNV** Se genera el nodo 4 que tiene CI= 141 y no se introduce en **LNV** Se genera el nodo 5 con CI=101 luego se introduce en **LNV** Se modifica C a 101
2	7-3	101	Se saca el nodo 5 y se generan sus hijos El nodo 6 se poda ya que CI es mayor que 144 El nodo 7 se introduce en la **LNV**
3	10-3	101	Se genera el nodo 9 y se poda se genera el nodo 10 y se inserta en **LNV**
4	3	101	Se genera el nodo 11 que es solucion con coste 101.
5		101	Se saca el nodo 3 con CI=98 luego se generan sus hijos. Los hijos, nodos 12,13 y 14 tienen valores de CI mayores a 101. Luego se podan

La solución por lo tanto es : $p_1 - b_4$, $p_2 - b_2$,$p_3 - b_3$ y $p_4 - b_1$ con coste 101. □ Veamos el código como quedaría.

```
/**
 * @brief Obtiene el menor coste de asignar a n pasajeros a
          n barcos usando branch & bound
 * @param n: el numero de tipos de pasajeros y barcos
 * @param ab: Arbol de permutaciones para obtener la mejor solucion
 * @param B: matriz de coste
 * @return el menor coste total
 * */
int Asig_PasajerosBarcos_Branch_Bound(int n, Apermutacion &ab,
                                          const Matriz<unsigned int> &B){
Apermutacion P(n);

  int cact=0; int best_coste=numeric_limits<int>::max();
  unsigned int nodos_recorridos =0;

  vector<int> aux(n,-1);

  int CS=CotaSuperior(aux,B); //coste inferior;//coste superior
  int CI= CotaInferior(aux,B); //coste inferior

  int C=CS;
  int BE=(CI+CS)/2; // coste optimo estimado
  priority_queue<nodo> pq;
// bool seguir=true;

nodo a(P,cact, CI,BE,CS);
pq.push(a);

bool solucion=false;

do{
  nodos_recorridos++;
  nodo a = pq.top(); pq.pop();
  //si aun no tenemos solucion CI<=C
  //si tenemos solucion menor estricto
  if ((!solucion && a.CI<=C) || (solucion && a.CI<C)){
      vector<vector<int> > hijos = a.V.GeneraHijos();
      //por cada uno de los hijos
      for (int i=0;i< (int)hijos.size();i++){
        Apermutacion H (hijos[i],a.V.GetLevel()+1);
        //coste actual del hijo
```

```
        cact=Suma_Beneficio(H,B);

        aux = ObtainAsignaciones(H,n);
        CS= cact+CotaSuperior(aux,B);//coste superior
        CI= cact+CotaInferior(aux,B);//coste inferior
        BE= (CS+CI)/2;
        //Si el coste del hijo es menor que el mejor obtenido.
        if (H.GetLevel()==n-1 && cact<best_coste){
            ab=H;
            best_coste=cact;
            C=(C<cact)?C:cact;// Modificamos el valor para podar
            solucion=true;
        }
        else{
        //Si no hemos analizado ya todos los trabajadores
        if (H.GetLevel()<n-1)
         // Si por esa rama podemos obtener un mayor beneficio
          if (C>=CI ){
            nodo anew (H, cact,CI,BE,CS);
            pq.push(anew);
            //se modifica la cota
            C= (CS<C)? CS:C;
          }
        }
      }
    }
  }while (!pq.empty());
  return best_coste;
}
```

Las funciones *CotaSuperior* y *CotaInferior* serían las siguientes:

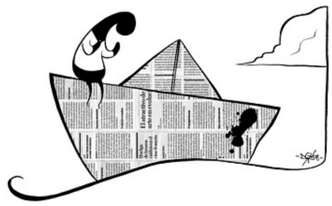
```
 /**
 * @brief Establece una cota inferior del
     coste de asignar tipos de pasajeros libres a barcos libres
 * @param asignados: vector con los pasajeros asignados a los barcos
 * @param B: matriz de costes
 * @return cota inferior del coste de asignar pasajeros libres
          a barcos libres.
 * */
int CotaInferior(vector<int>asignados,const Matriz<unsigned int> &B){
   Matriz<bool>usados(asignados.size(),asignados.size(),false);
   int n=asignados.size();
   vector<bool>candidatos(n,true);
   //establecemos que filas y columnas de la matriz usados
```

```
   //estan habilitadas
   for (unsigned  int i=0;i<asignados.size();i++){
      if(asignados[i]>=0){
       //no es un candidato el pasajero i
       candidatos[i]=false;
       for ( int j=0;j<n;j++)
       //ni el barco asignado
         usados[j][asignados[i]]=true;
      }
   }
   unsigned int best_coste=0;
   for (int i=0;i<n;i++){
      if (candidatos[i]){ // es un candidatos
        //buscamos entre los barcos  que quedan libres
        //los mas baratos aunque haya repeticiones
        int mejor=numeric_limits<int>::max();
        //int work=0;
        for (int j=0;j<n;j++){
            if (usados[i][j]==false)
              if ((int)B.get(i,j)<mejor){

                 mejor = B.get(i,j);
              }
        }
        best_coste +=mejor;
      }
   }
   return best_coste;
}

/**
 * @brief Establece una cota superior del
     coste de asignar tipos de pasajeros libres a barcos libres
 * @param asignados: vector con los pasajeros asignados a los barcos
 * @param B: matriz de costes
 * @return cota inferior del coste de asignar pasajeros libres
           a barcos libres.
 * */
int CotaSuperior(vector<int>asignados,const Matriz<unsigned int> &B){
   Matriz<bool>usados(asignados.size(),asignados.size(),false);
   int n=asignados.size();
   vector<bool>candidatos(n,true);
   //establecemos que filas y columnas de la matriz usados
```

```
    //estan habilitadas

    for (unsigned  int i=0;i<asignados.size();i++){
       if(asignados[i]>=0){
       //no es un candidato el pasajero i
        candidatos[i]=false;
        for ( int j=0;j<n;j++)
        //ni el barco asignado
         usados[j][asignados[i]]=true;
       }
    }
    unsigned int best_coste=0;
    for (int i=0;i<n;i++){
       if (candidatos[i]){ // es un candidatos
          // es un candidatos
         //buscamos entre los barcos  que quedan libres
         //el mas barato
         int mejor=numeric_limits<int>::max();
         int barco=0;
         for (int j=0;j<n;j++){
             if (usados[i][j]==false)
               if ((int)B.get(i,j)<mejor){
                  barco=j;
                  mejor = B.get(i,j);
               }

         }
         //este barco ya no esta disponible
         //por lo tanto lo anulamos
         for (int t=0;t<n;t++){
           usados[i][t]=true;
           usados[t][barco]=true;
         }
         best_coste +=mejor;
       }
    }
    return best_coste;
 }
```

6.11.5 Problema.-Pasajeros a sus barcos (segunda versión)

En esta segunda versión del problema de asignar tipos de pasajeros a barcos se diferencia en que podemos asignar a diferentes tipos de pasajeros el mismo barco. Así el problema consiste en dado un conjunto de barcos B y un conjunto de tipos de pasajeros P, queremos asignar el tipo de pasajero a un barco. El coste de que un determinado conjunto de pasajeros p_i viaje en el barco b_j viene dada por la matriz de costos C en $C(i,j)$. El objetivo en este problema es diseñar un algoritmo para asignar los pasajeros a los barcos con el mínimo coste.



Ejemplo 6.11.8
Dada la siguiente matriz que representa el coste de viajar un tipo de pasajero en un barco,

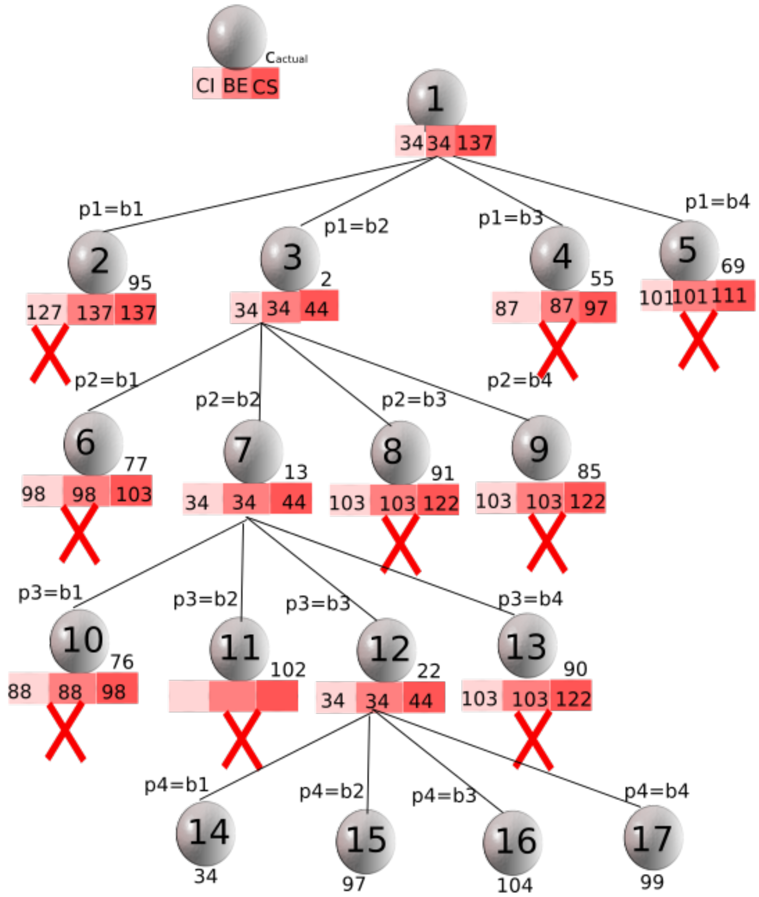
Coste	b_1	b_2	b_3	b_4
p_1	95	2	55	69
p_2	75	11	89	83
p_3	63	89	9	77
p_4	12	75	82	22

queremos encontrar la mejor asignación para minimizar el coste acumulado.

En este caso las cotas son

- $CI = c_{actual} +$ CotaInferior(pasajeros libres,barcos libres)
- $CS = c_{actual} +$ CotaSuperior(pasajeros libres,barcos libres)

siendo $CotaInferior$ el coste de asignar a los pasajeros libres un barco libre, el de menor coste, aunque en esta asignación de pasajeros libres haya repeticiones.
Y $CotaSuperior$ asignar al pasajero i el barco b_i. Cualquier otro barco podría ser posible.
Por otro lado el coste estimado ahora lo vamos a definir igual a la cota inferior $BE = CI$.
El árbol de soluciones es el siguiente:



La mejor solución en este caso es: $p_1 - b_2$, $p_2 - b_2$,$p_3 - b_3$ y $p_4 - b_1$ con coste 34.
Se deja como ejercicio al alumno para que analice como se modifica la cota de poda C y la lista de nodos vivos. Por otro lado modificar la función CotaSuperior y CotaInferior para adapatarlo a esta modificación. Hay que tener en cuenta que cada nodo pertenece a un arbol n-ario y no permutacional.