Técnicas de Exploración de Grafos

Backtracking (Vuelta Atrás)
Branch and Bound (Ramificación y Poda)

Introducción

- Supongamos que tenemos que tomar un conjunto de decisiones entre varias alternativas donde
  - No tenemos suficiente información sobre qué decisión tomar
  - Cada decisión nos abre un nuevo abanico de alternativas
  - Alguna secuencia de decisiones (probablemente más de una) puede ser una solución del problema
- Backtracking y B&B son metodologías que se pueden utilizar para buscar varias secuencias de decisiones hasta encontrar la que sea “correcta”

Backtracking Y Branch&Bound

- Características del problema:
  - La solución debe poder expresarse mediante una tupla

    $$(x_1,x_2,x_3,...,x_n)$$

    donde cada x_i es seleccionado de un conjunto finito S_i.
  - Unas veces el problema a resolver trata de encontrar una tupla que maximiza (o minimiza) una función criterio u objetivo $P(x_1,x_2,...,x_n)$.
  - Otras veces solo se trata de encontrar una tupla que satisfaga (no optimice) el criterio.
  - Otras veces se trata de encontrar todas las tuplas que satisfagan el criterio.
- ¿Se parece esto a las técnicas greedy?

Ejemplo: ordenación

- Ordenar los enteros en A(1..n) es un problema cuya solución es expresable mediante una n-tupla en la que x_i es el índice en A del i-esimo menor elemento.
- La función de criterio P es la desigualdad

$$A(x_i) \leq A(x_{i+1}), 1 \leq i \leq n.$$

- El conjunto S_i es finito e incluye a todos los enteros entre 1 y n. Ejemplo:

  i: 1 2 3 4 5

  A(i): 4 7 3 6 1

  x(i): 5 3 1 4 2

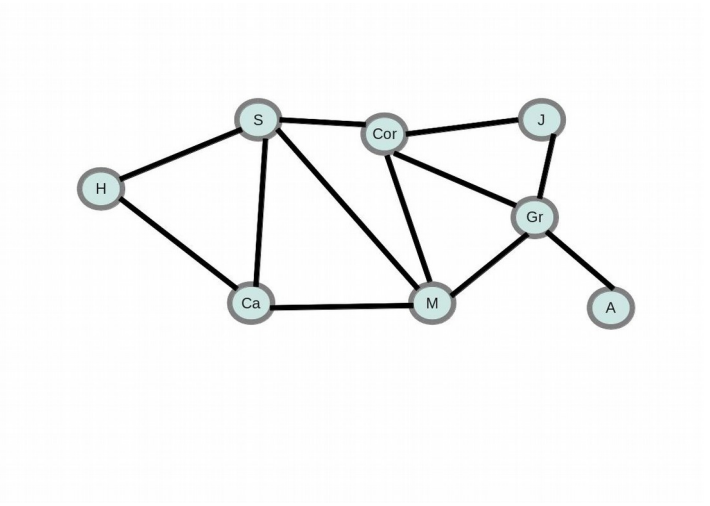
- La ordenación no es uno de los problemas que habitualmente se resuelven con backtracking

Ejemplo: Coloreo de mapas

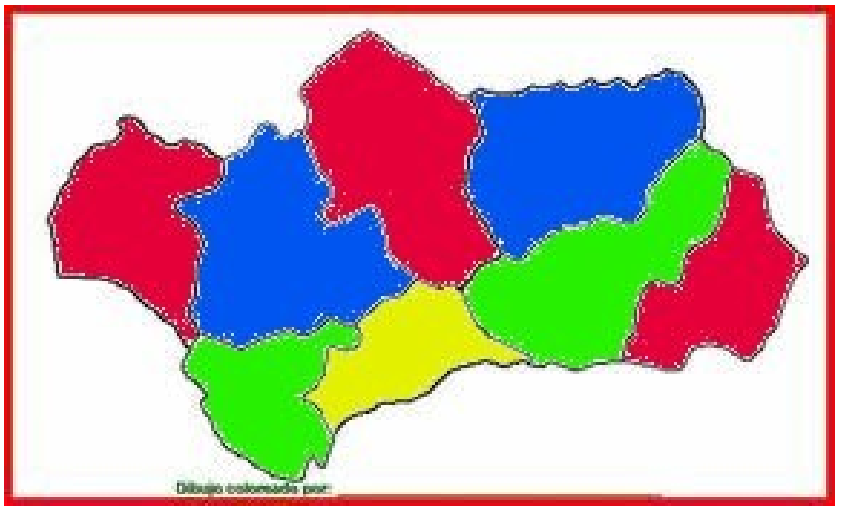
- Queremos colorear un mapa con no más de cuatro colores
  - Rojo, amarillo, azul, verde

- Países adyacentes deben tener diferente color
- No tenemos suficiente información para elegir dichos colores
- Cada elección del un color nos abre un conjunto de alternativas para solucionar el problema
- Una o varias secuencias de colores nos pueden llevar a la solución

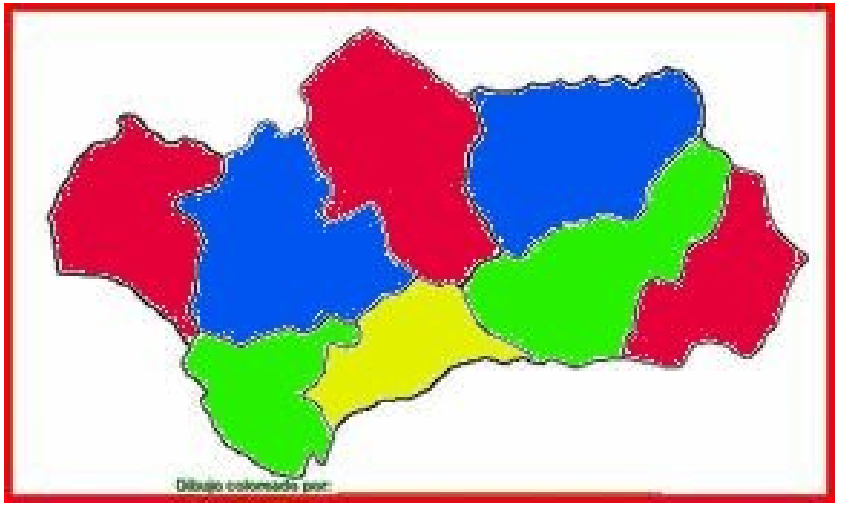
Colorear Mapa



- Algoritmo
- Recorrer nodos 1 al 8 y asignar el primer color factible en orden Ro Az Ve Am



Colorear mapa

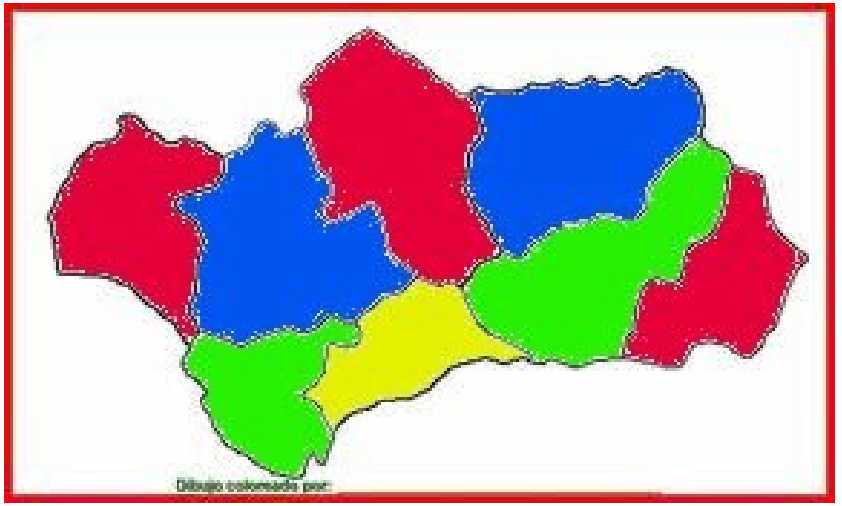


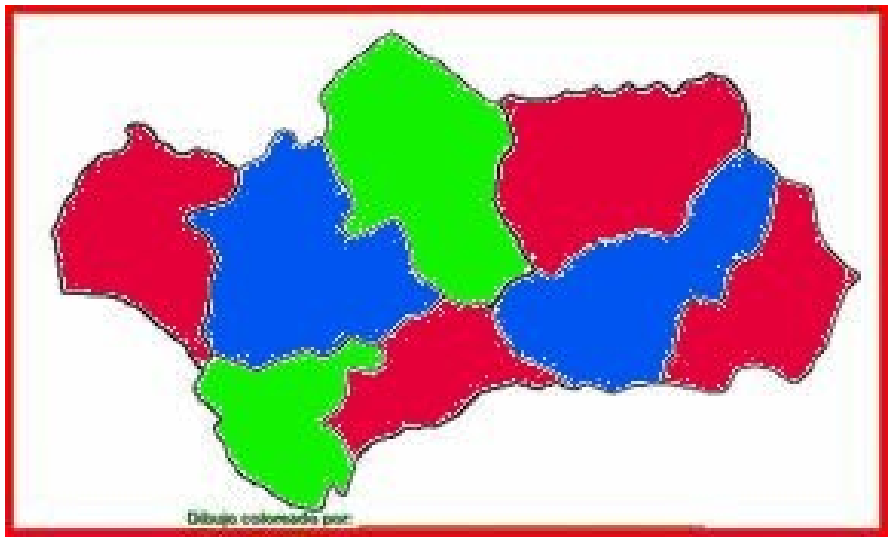
Es óptimo?

Criterio de optimalidad:

- Su pueden utilizar menos colores?
- Supongamos costos, minimizar costo final
  - Ro 80 Az 60 Ve 40 Am 20
    - Costo = 460

Minimizar colores

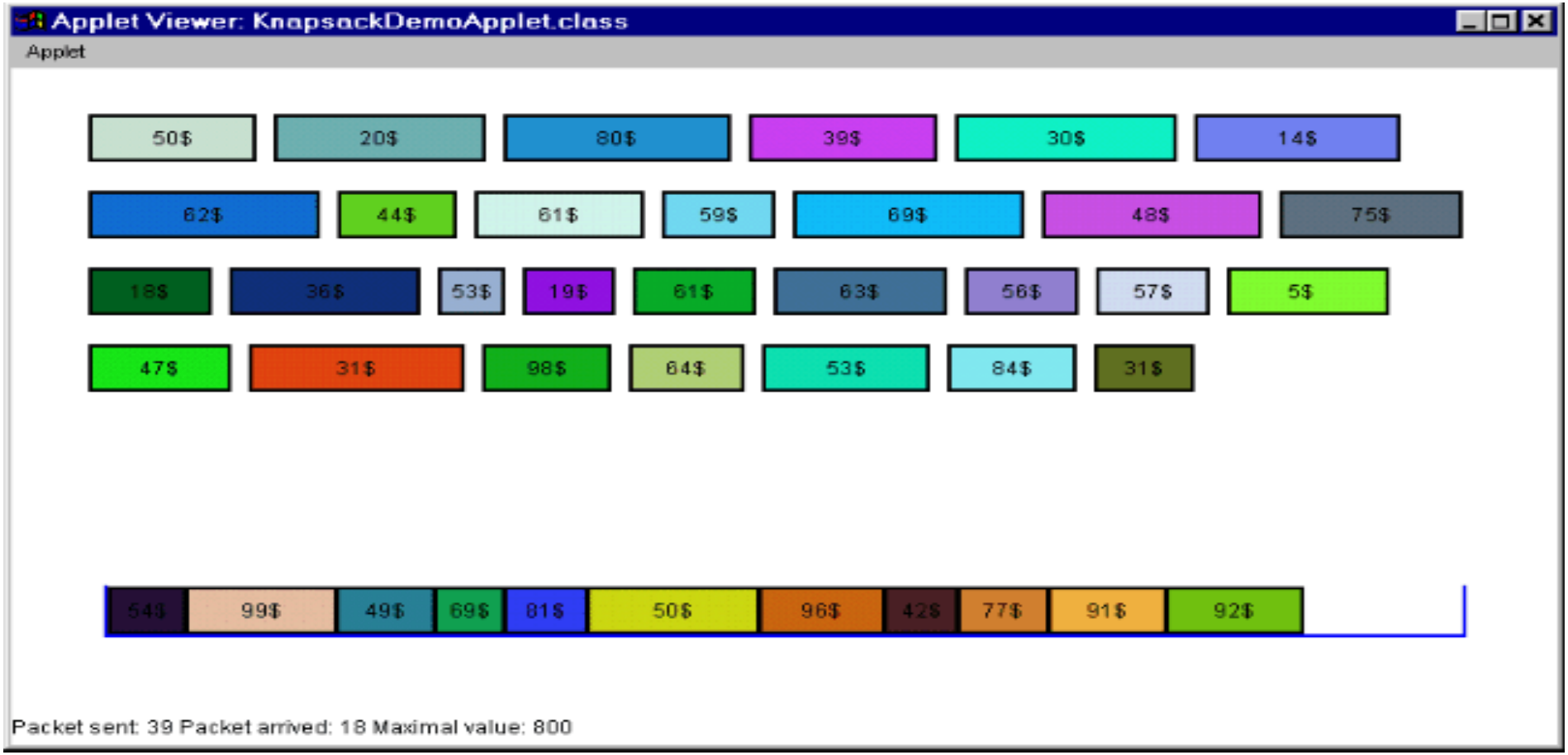




- Su pueden utilizar menos colores? Si
- Supongamos costos
  - Ro 80 Az 60 Ve 40 Am 20
    - Costo_4 = 460
    - Costo_3 = 520
- Cómo minimizar costos?

Ejemplo: Problema de la mochila

- Disponemos de N paquetes de datos a transmitir, donde cada paquete necesita un tiempo t_i para transmitirse y tiene una ganancia g_i. Se pide seleccionar el conjunto de paquetes que se transmiten en un tiempo delimitado T, maximizando la ganancia final.



Ejemplo: Problema del laberinto

- Dado un laberinto, encontrar el camino desde el comienzo a la salida
- En cada intersección tenemos que decidir entre varias alternativas
  - Seguir recto, ir a la derecha, a la izquierda, ..
- No tenemos información sobre cual decisión es la correcta
- Cada elección nos lleva a un nuevo conjunto de decisiones
- Puede haber una o varias secuencias de decisiones que sean una solución al problema

Fuerza bruta

- Sea m_i el tamaño del conjunto S_i.
- Hay $m = m_1 \cdot m_2 \cdot ... \cdot m_n$ n-tuplas posibles candidatos a satisfacer la función P.
- El enfoque de la fuerza bruta propondría formar todas esas tuplas y evaluar cada una de ellas con P, escogiendo la que diera un mejor valor de P o satisfaciera P.
- Backtracking y Branch&Bound proporcionan la misma solución pero en mucho menos de m intentos (no siempre).

Back y B&B versus fuerza bruta

- La idea básica es construir la tupla escogiendo una componente cada vez,
- y usando funciones de criterio modificadas $P_i(x_1, ..., x_n)$, que a veces se llaman funciones de acotación o poda, para testear si la tupla que se está formando tiene posibilidad de éxito.
- La principal ventaja de este método es que si a partir de la tupla parcial $(x_1, x_2, ..., x_i)$ se deduce que no se podrá construir una solución, entonces pueden ignorarse por completo $m_{i+1} \cdot ... \cdot m_n$ posibles test de tuplas.

Ejemplo de Fuerza Bruta

- Problema:
  - Generar todas las posibles combinaciones de n bits.
- Aplicaciones:
  - Selección de elementos en un conjunto
    - Selección de actividades
    - Mochila
    - etc.

00000

00001

00010

00011

00100

00101

00110

00111

.....

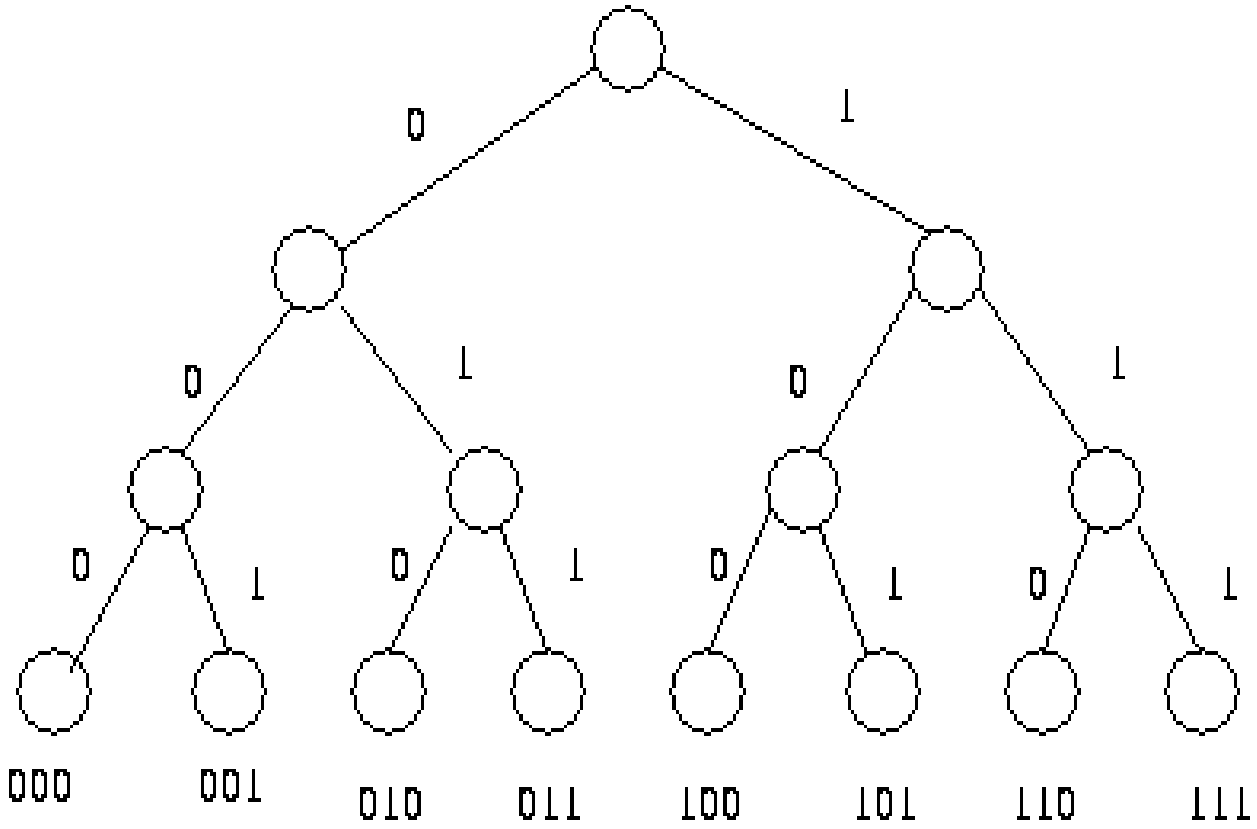
```
void completa_binario( vector<int> & V, int pos)
{  if (pos==V.size())
            procesa_vector(V);
  else {
    V[pos]=0;
    completa_binario(V,pos+1);
    V[pos]=1;
    completa_binario(V,pos+1);
  }
}

void procesa_vector(vector<int> & V)
{  int i;
    for (i=0;i<V.size();i++)
            cout << V[i];
     cout << endl;
}
```

Análisis de Eficiencia

- Ecuación de Recurrencia

T(n) = 2 T(n-1) +1 , es del orden O(2^n).

Arbol de recurrencia



Ejemplo: Coloreo de Grafos.

- Pintar los vértices de un grafo de forma que dos vértices adyacentes no tengan el mismo color.

Solución

$$X = (x1,x2,x3,x4,x5,x6)$$

con Xi el color con el que se pinta el vértice i-esimo.

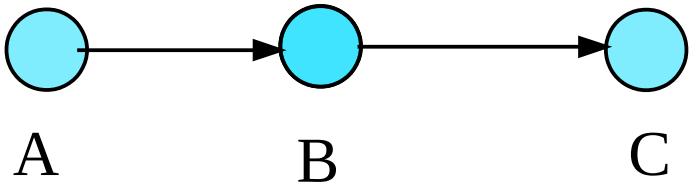
S_i representa al conjunto de colores disponible.

(Sabemos que si el grafo es plano, basta con 4 colores)

Por ejemplo,

X = (rojo,verde,azul,negro,rojo,azul)

```
void completa_grafo( vector<int> & X, int pos)
{

  if (pos==X.size())
    chequear_factibilidad(X);
  else {
    X[pos]=0;       //Rojo
    completa_grafo(X,pos+1);
    X[pos]=1;      // Verde
    completa_grafo(X,pos+1);
    X[pos]=2;     // Azul
    completa_grafo(X,pos+1);
    X[pos]=3;    // Negro
    completa_grafo(X,pos+1);
  }
}
```

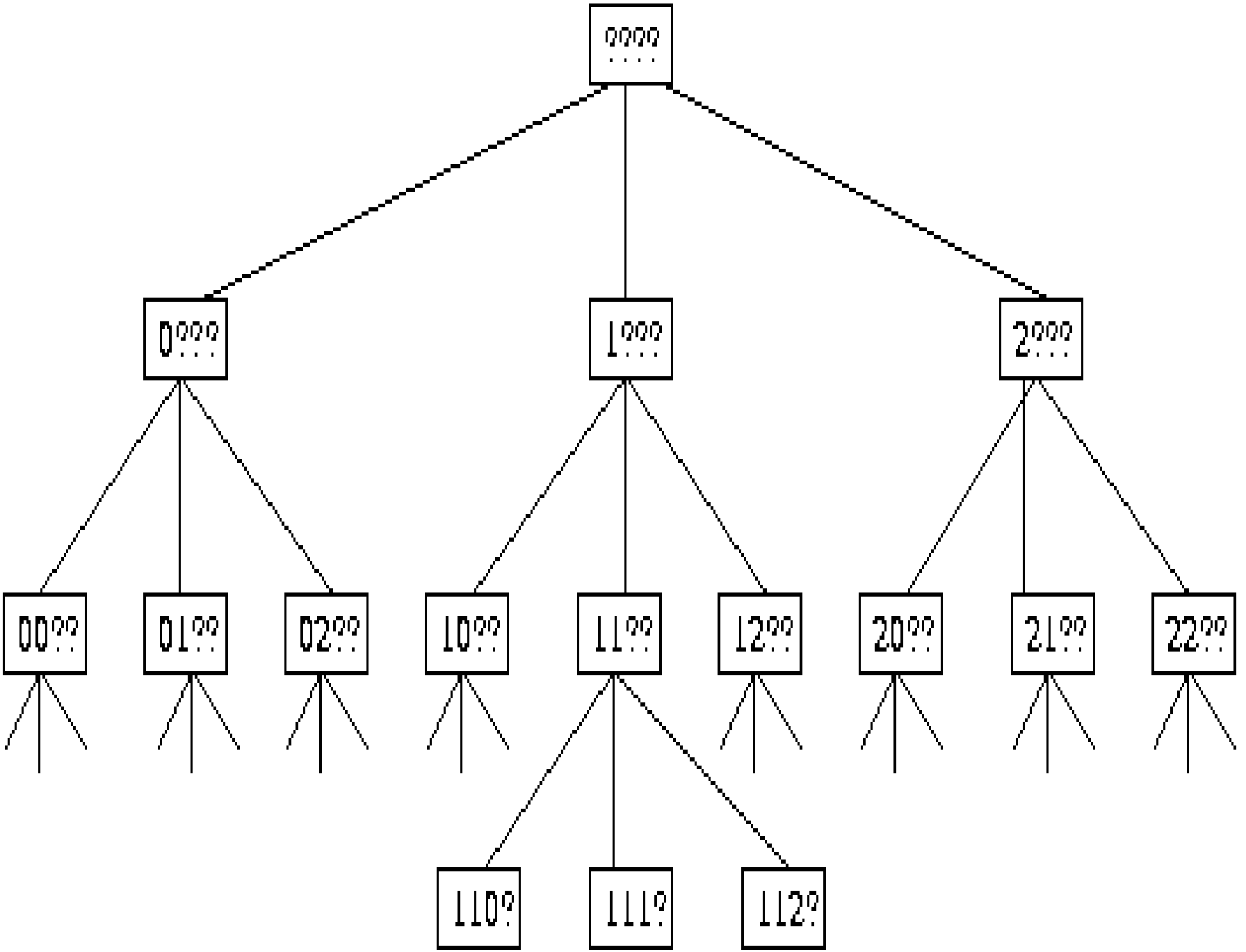



```
ABC
000
001
002
003
010  ok
011
012  ok
013  ok
020  ok
021  ok
022
023  ok
030  ok
031  ok
032  ok
033
100
101  ok
.............
```

CASO GENERICO.....

```
void completa_k-ario( vector<int> & X, int pos, int k)
{
  int j;
  if (pos==X.size())
        procesa_vector(X);
  else {
      for (j=0; j<k; j++){
         x[pos]=j;
         completa_k-ario(X,pos+1,k);
        }
    }
}
```

Escribir el árbol de recurrencia (4 elementos) para k=3


0000
0???
1???
2???
00??
01??
02??
10??
11??
12??
20??
21??
22??
110?
111?
112?

Qué hemos visto?

- Se impone una estructura (virtual) de árbol sobre el conjunto de posibles soluciones (espacio de soluciones)
- La forma en la que se generan las soluciones es equivalente a realizar un recorrido en pre-orden (en profundidad) del árbol, el espacio de soluciones.
- Se procesan las hojas (que se corresponden con soluciones completas).
- Pregunta:
  - ¿Se puede mejorar el proceso? ¿Cuándo? ¿Cómo?

BK y B&B !!!!

Se puede mejorar el proceso?

- Si, eliminando la necesidad de alcanzar una hoja para procesar.

  Cuando para un nodo interno del árbol podemos asegurar que no alcanzamos una solución (no nos lleva a nodos hoja útiles), entonces podemos **podar la rama.**

  Ventaja: Alcanzamos la misma solución con menos pasos.

Espacios de soluciones

- Los algoritmos backtracking y B&B determinan las soluciones del problema buscando en el espacio de soluciones del caso considerado sistemáticamente.

- Esta búsqueda se facilita usando una organización en árbol para el espacio solución.

- Para un espacio solución dado, pueden caber muchas organizaciones en árbol.

Diferencias con otras técnicas

- En los algoritmos greedy se construye la solución buscada, aprovechando la posibilidad de calcularla a trozos, pero con backtracking y B&B la elección de un sucesor en una etapa no implica su elección definitiva. Greedy explora una sola rama del árbol, BK y B&B exploran más de una rama.

- En Programación Dinámica, la solución se construye por etapas (a partir del principio de optimalidad), y los resultados se almacenan para no tener que recalcular, lo que no es posible en Backtracking y B&B.

Notación:

- **Solución Parcial:** tupla o vector solución para el que aun no se han asignado todos sus componentes.
- **Función de Poda**: Aquella función que nos permite identificar cuando una solución parcial no conduce a una solución del problema.
- **Restricciones Explícitas:** Reglas que restringen el conjunto de valores que puede tomar cada una de las componentes X[i] del vector solución.

  Todas las tuplas que satisfacen las restricciones explícitas definen un espacio de soluciones del caso que estamos resolviendo.

Notación (cont.)

- **Restricciones Implícitas**: Son aquellas que determinan cuando una solución parcial nos puede llevar a una solución (verifican función objetivo).

  Las restricciones implícitas describen la forma en la que las x_i deben relacionarse entre sí.

Notación (cont.)

- La organización en árbol del espacio solución se llama **árbol de estados.**
- **Estado del problema**: Cada uno de los nodos del árbol.
- **Estado solución**: Son los nodos del árbol que representan una posible solución al problema (el camino desde la raíz al nodo).
- **Estado respuesta**: representa una solución del problema (satisface las restricciones implícitas).

Notación (cont.)

- **Nodo vivo**: Nodo (estado del problema) que ya ha sido generado, pero del que aun no se han generado todos sus hijos.
- **Nodo muerto:** Nodo que ha sido generado, y o bien se ha podado o bien se han generado todos los descendientes.
- **e-nodo (nodo en expansión)**: Nodo vivo del que actualmente se están generando los descendientes.

Problema de la suma de subconjuntos

- Dados n+1 números positivos:

  w_i, $1 \leq i \leq n$, y uno más M,

- se trata de encontrar todos los subconjuntos de números w_i cuya suma valga M.
- Por ejemplo, si $n = 4$,

  $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ y $M = 31$,

  entonces los subconjuntos buscados son (11, 13, 7) y (24, 7).

Problema de la suma de subconjuntos

- Para representar la solución podríamos notar el vector solución con los índices de los correspondientes w_i.
- Las dos soluciones se describen por los vectores (1,2,4) y (3,4).
- Todas las soluciones son k-tuplas $(x_1, x_2,...,x_k)$, $1 \leq k \leq n$, y soluciones diferentes pueden tener tamaños de tupla diferentes.
- Restricciones explícitas: $x_i \in \{j: j \text{ es entero y } 1 \leq j \leq n\}$
- Restricciones implícitas: que no haya dos iguales y que la suma de los correspondientes w_i sea M.
- Como, por ejemplo (1,2,4) y (1,4,2) representan el mismo subconjunto, otra restricción implícita que hay que imponer es que $x_i < x_{i+1}$, para $1 \leq i < n$.

Problema de la suma de subconjuntos

- Puede haber diferentes formas de formular un problema de modo que todas las soluciones sean tuplas satisfaciendo algunas restricciones.

- Otra formulación del problema:
  - Cada subconjunto solución se representa por una n-tupla $(x_1,...,x_n)$ tal que $x_i \in \{0,1\}$, $1 \leq i < n$, y $x_i = 0$ si w_i no se elige y $x_i = 1$ si se elige w_i.
  - Las soluciones del anterior caso son (1,1,0,1) y (0,0,1,1).
  - Esta formulación expresa todas las soluciones usando un tamaño de tupla fijo.

- Se puede comprobar que para estas dos formulaciones, el espacio solución consiste en ambos casos de 2^4 tuplas distintas.

Suma de subconjuntos: árbol

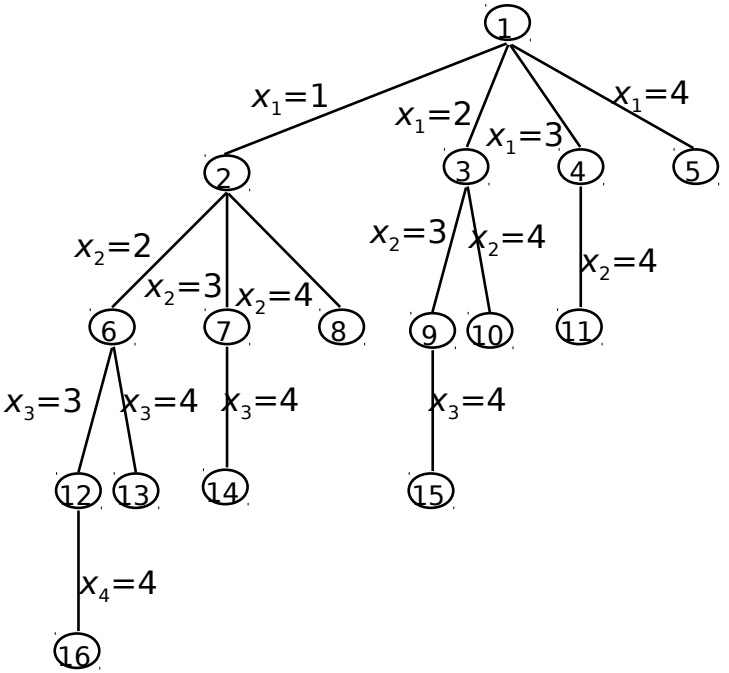
- Dos posibles formulaciones del espacio solución del problema de la suma de subconjuntos.
  - La primera corresponde a la formulación por el tamaño de la tupla variable
  - La segunda considera un tamaño de tupla fijo

- Con ambas formulaciones, tanto en este problema como en cualquier otro, el numero de soluciones tiene que ser el mismo

Suma de subconjuntos: árbol 1

- Las aristas se etiquetan de modo que una desde el nivel de nodos i hasta el i+1 representa un valor para x_i.
- En cada nodo, el espacio solución se particiona en espacios subsolución.
- Los posibles caminos son (), que corresponde al camino vacío desde la raíz a si misma, (1), (12), (123), (1234), (124), (13), (134), (14), (2), (23), etc.
- En esta representación todos los nodos son estados solución.

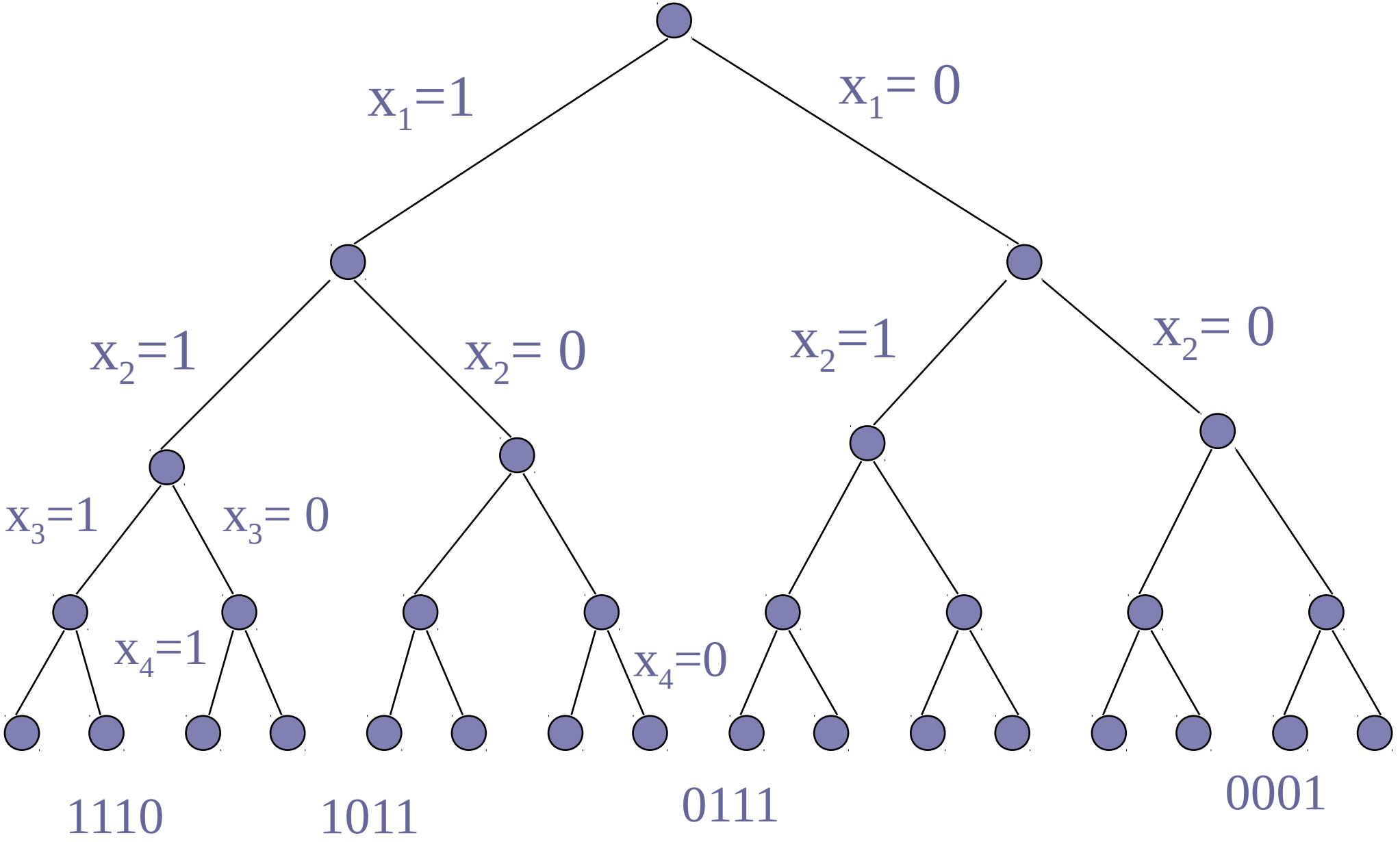
- Así, el subárbol de la izquierda define todos los subconjuntos conteniendo w_1, el siguiente todos los que contienen w_2 pero no w_1, etc.



Suma de subconjuntos: árbol 2

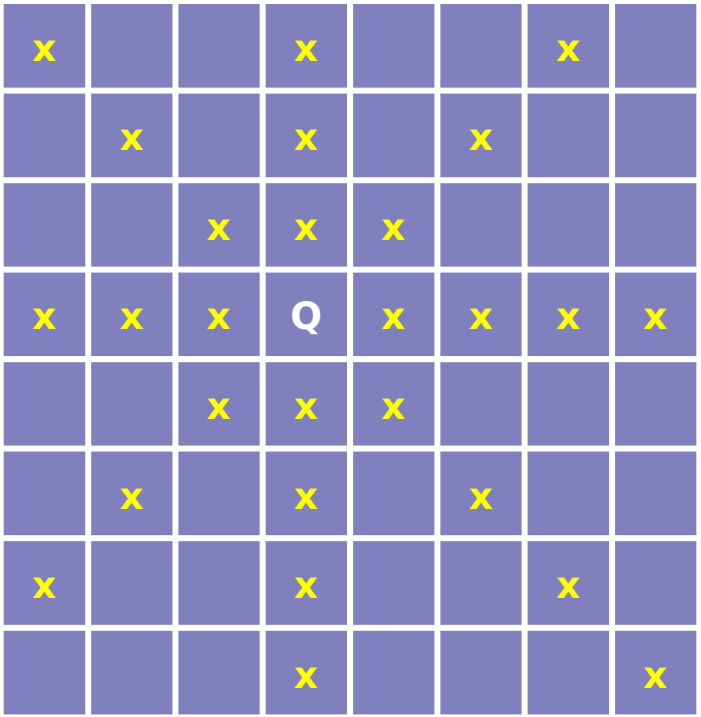
- Una arista del nivel i al i+1 se etiqueta con el valor de x_i (0 o 1).
- Todos los caminos desde la raíz a las hojas definen el espacio solución.
- El subárbol de la izquierda define todos los subconjuntos conteniendo w_1, mientras que el de la derecha define todos los subconjuntos que no contienen w_1, etc.
- Consideramos el caso de n = 4
- Hay 2^4 nodos hoja, que representan 16 posibles tuplas.
- En esta representación sólo los nodos hoja son estados solución.

Suma de subconjuntos: árbol 2



El problema de las ocho reinas

- Un clásico problema combinatorio es el de colocar ocho reinas en un tablero de ajedrez de modo que no haya dos que se ataquen, es decir, que estén en la misma fila, columna o diagonal.
- Las filas y columnas se numeran del 1 al 8.
- Las reinas se numeran del 1 al 8.



Si representamos la posible solución como una 8-tupla de las posiciones de las reinas, y la posición de cada reina como un par (fila,columna), tenemos $64^8=2^{48}=$ 281,474,976,710,656 posibilidades.

El problema de las 8 reinas

Como cada reina debe estar en una fila diferente, sin pérdida de generalidad podemos suponer que la reina i se coloca en la fila i.

Todas las soluciones para este problema, pueden representarse como 8 tuplas $(x_1,...,x_n)$ en las que x_i es la columna en la que se coloca la reina i. Esto reduce el tamaño del espacio de soluciones a $8^8=2^{24}=16,777,216$.

Restricciones explícitas: $x_i \in \{1,2,3,4,5,6,7,8\}$

Restricciones implícitas: ningún par de x_i pueden ser iguales. Ningún par de reinas pueden estar en la misma diagonal.

La primera restricción implica que todas las soluciones son permutaciones de {1,2,3,4,5,6,7,8}. Esto reduce el tamaño a 8! =40,320

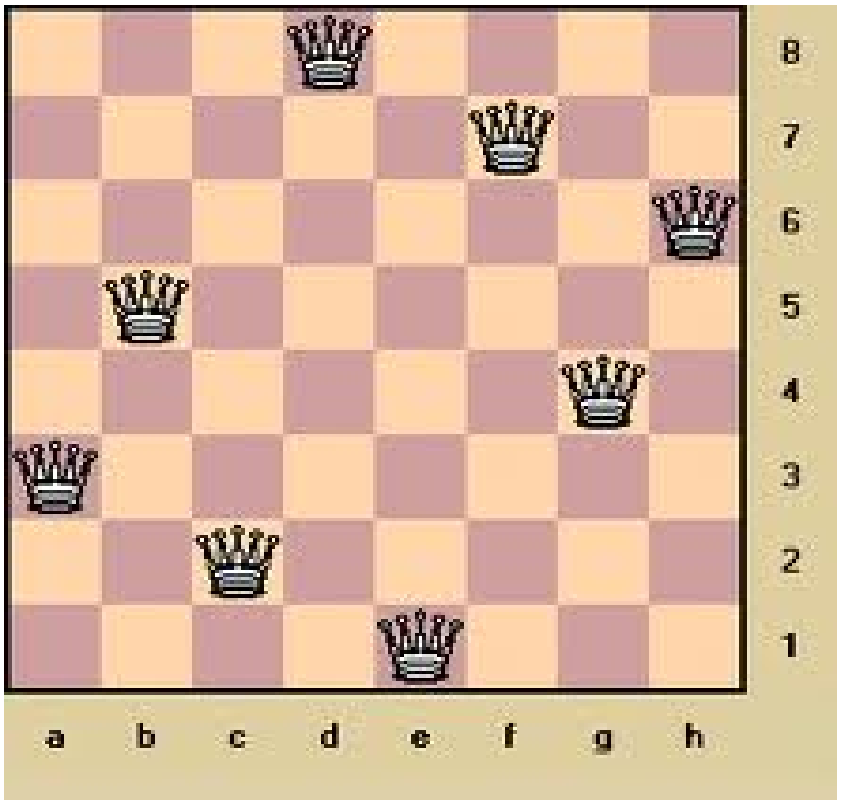
El problema de las 8 reinas

Posible solución del problema:

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

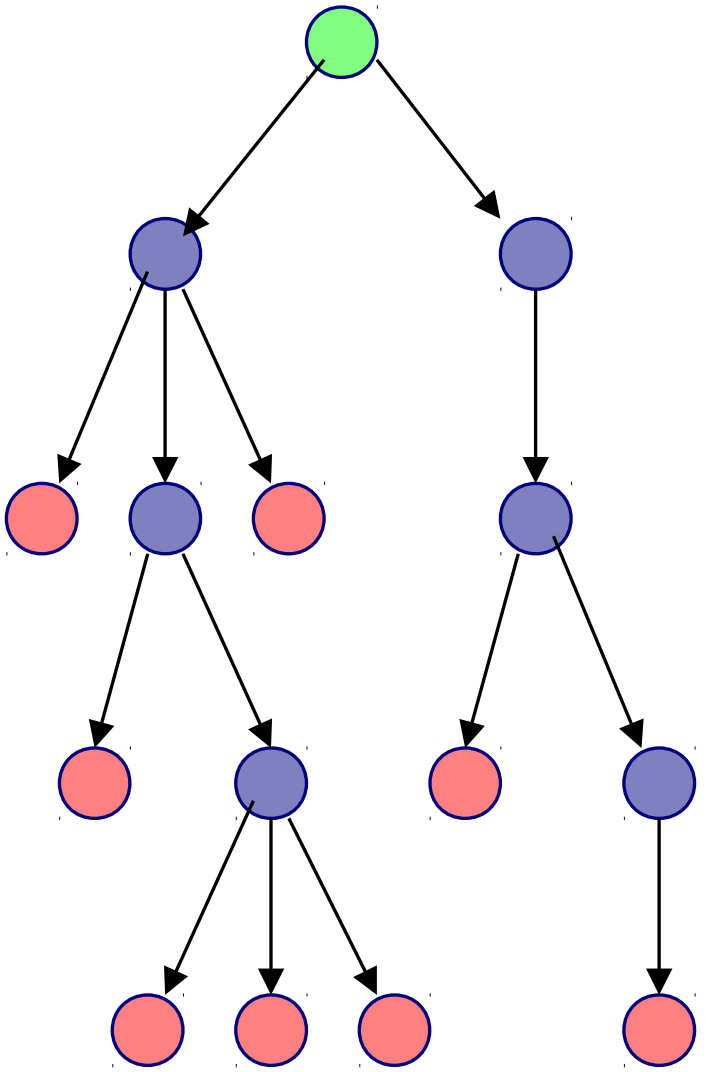
Generalización:

- Problema de las N reinas



Generación de estados de un problema

- Cuando se ha concebido un árbol de estados para algún problema, podemos resolver este problema generando sistemáticamente sus estados, determinando cuáles de estos son estados solución, y finalmente determinando qué estados solución son estados respuesta.



Backtracking vs Branch and Bound

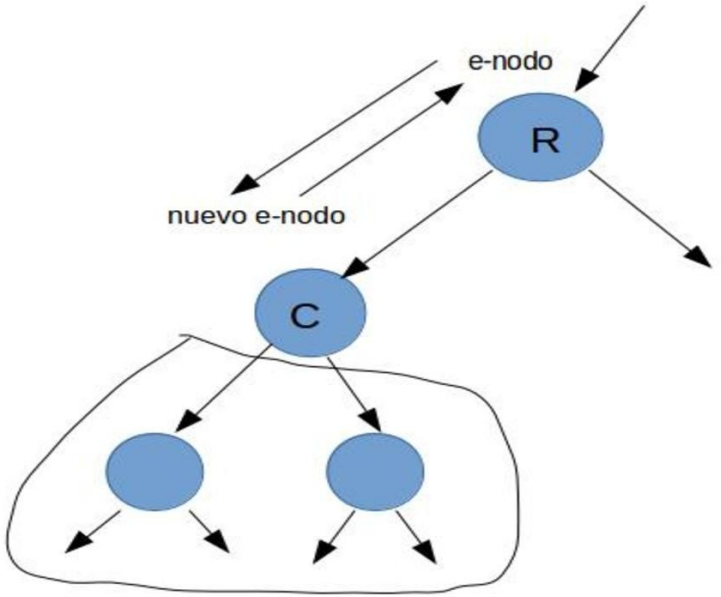
- Ambos métodos, recorren el árbol de estados de forma sistemática.
- Ambos métodos utilizan funciones de poda para eliminar ramas que no conducen a soluciones.
- BK: Cuando un nuevo hijo, C, del e-nodo R ha sido generado, entonces C se convierte en el nuevo e-nodo. R no vuelve a ser e-nodo hasta que no se han explorado todos los descendientes de C.

Recorrido primero en profundidad.

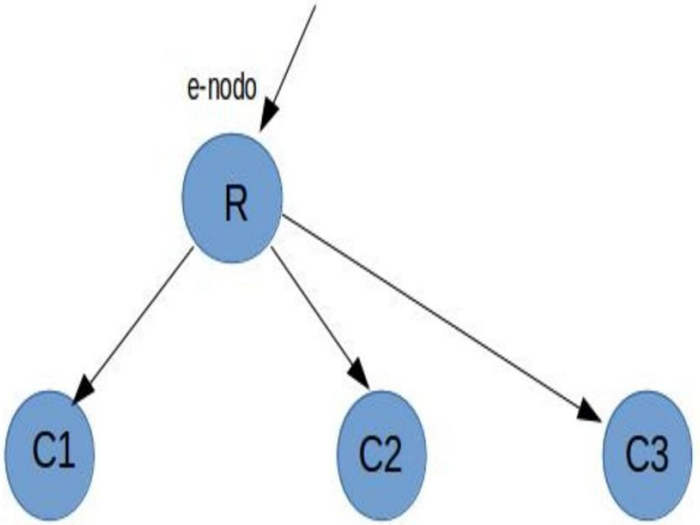
- B&B: el e-nodo continua siéndolo hasta que se han generado todos sus descendientes o se poda.

Distintos criterios de recorrido (FIFO,LIFO,LC).

Backtracking vs Branch and Bound



Backtracking



Branch and Bound

Backtracking:

Veremos .....

- TDA Solución.
- Algoritmo genérico recursivo.
- Esquema genérico iterativo.
- Aplicación al problema de suma de subconjuntos
- Aplicación al problema de las n-reinas
- Aplicación al problema del coloreo de grafos
- Aplicación al Problema de la Mochila.
- Problema Viajante Comercio
- Sobre eficiencia de algoritmos BK

Backtracking: TDA Solución.

Dominio de las posibles decisiones es de tipo TDec:

Ejemplo: Suma Subconjuntos {0,1}; Nreinas {1,2,3,...,N}
Incluimos dos estados mas: NULO y END
Se asume un orden sobre TDec, p.e. NULO<1<2<3< ...< END

```
class Solucion {
  private:
  vector<TDec> X;      // Almacena la solución
  ...                  // otra información relevante
 public:
  Métodos públicos
  Solucion(const int tam_max);   // Constructor
  ........                       // otros métodos información relevante
 ~Solucion();
};
```

Backtracking: TDA Solución.

Métodos:

Solucion:: Solucion(const int tam_max); // Constructor

Entre otras cosas reserva memoria para almacenar el vector X;
lo inicializa con la decisión NULA

int Solucion::size() const;
Devuelve el tamaño del vector solución

void Solucion::IniciaComp(int k);
Asigna valor nulo a X[k], p.e. X[k]= NULO

Backtracking: TDA Solución.

```
void Solucion::SigValComp(int k);
// Siguiente valor válido del dominio
```

Por ejemplo, si TDec = {NULO,1,2,3,4,5,6,7,8,END}

Si (X[k] == NULO) entonces tras llamar al método X[k] <-- 1

Si (X[k] == 5) entonces tras llamar al método X[k] <-- 6

Si (X[k]==8) entonces tras llamar al método X[k]<-- END !!!!

- Sólo genera valores válidos para X[k]: Usa restricciones explícitas.

Backtracking: TDA Solución.

bool Solucion::TodosGenerados(int k);
testea si quedan valores de S_k por generar, (return X[k]==END)

TDec Solucion::Decision(int k) const;
Obtener valor componente k, return X[k]

void Solucion::ProcesaSolucion();
// Representa el proceso que se realiza cuando se alcanza una solución.
Permite quedarnos con la mejor solución

Por ejemplo;

- Imprimir la solución;
- Si es un problema de optimización comparar con la mejor solución alcanzada hasta el momento.

Backtracking: TDA Solución.

bool Solucion::Factible(int k) const;
Es la función mas importante del mecanismo backtracking
Devuelve true si la solución actual, almacenada en (x1,x2, ..., xk) cumple las restricciones y false en caso contrario.

• Usa las restricciones implícitas:

•Podemos suponer la existencia de funciones de acotación (expresadas como predicados) tales que, Factible(x1,x2,...,xk) es falsa para un camino (x1, x2,..., xk) desde el nodo raíz hasta un estado del problema si el camino no puede extenderse para alcanzar un nodo respuesta

Esquema Recursivo

```
void back_recursivo(Solucion & Sol, int k)
{
if ( k == Sol.size())
    Sol.ProcesaSolucion();
else {
    Sol.IniciaComp(k);
    Sol.SigValComp(k);
    while (!Sol.TodosGenerados(k) {
      if (Sol.Factible(k))
          back_recursivo(Sol, k+1);
      Sol.SigValComp(k);
     }
  }
}
```

//BK ITERATIVO

```
void back_iterativo (Solucion & sol)
{  int k = 0;        // k representa la componente actual
  sol.IniciaComp(k);  //Se inicializa la primera componente a NULO
  while (k >= 0) {
          sol.SigValComp(k); // Probamos el sig. valor para X[k]
          if (sol.TodosGenerados(k))
                    k--; //Generados todos, por tanto backtracking
          else {
          if (sol.Factible(k))   // X[k] satisface restric
             { if (k == sol.Size() -1 )     // solución completa
                   sol.ProcesaSolucion();
               else  {
                   k++; // En caso contrario, ir a siguiente componente
                   sol.IniciaComp(k);
               }
             }  else {   ....  // Si el vector solución actual no es factible }
          }
}
```

Suma de Subconjuntos

Funciones de la Clase Solución:

```
void IniciaComp(int k)
        {  X[k] = 2 // Valor NULO   }
void SigValComp(int k)
        { X[k]--;     // Siguiente valor del dominio. }
bool TodosGenerados(int k)
        {return X[k]== -1; //END}
bool Solucion::Factible(int k)
{ // Una solución es factible sii:
```

!!! Añade un O(n) en cada nodo interno del árbol!!!

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

y

$$\sum_{1..k} W(i)X(i) + W(k+1) \leq M \text{ (si } W(i) \text{ ordenados en orden creciente)}$$

ó $\sum_{1..k} W(i)X(i)=M$

Suma de Subconjuntos (Eficiente)

Se puede hacer más eficiente:

Incluimos unos acumuladores sobre las decisiones tomadas en la clase solución

Class solucion {

private: // *Asumimos que los W(j) están en orden creciente*.

vector<int> X;

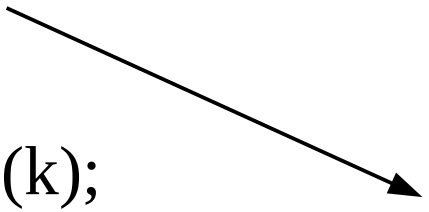
int s; // s = ∑1..k W(j)X(j)

int r; // r = ∑k+1..n W(j)

Funciones de la Clase Solución:

```
void SigValComp(int k)
        {  // Orden de los valores   2 -> 1 -> 0
           X[k]--;
          if  (X[k] == 1) {  s = s +w[k];  r = r-w[k]; }
          if  (X[k] == 0)   s = s - w[k]; // Descontamos  el valor
        }
bool Solucion::Factible(int k)
    { bool fact = false;
    if ( ( (s+w[k+1] <= M) && (s+r >= M) ) || (s==M) ) fact = true;
      return fact;
}
```

```
void back_recursivo(Solucion & Sol, int k) {
if ( k == Sol.Size()) Sol.ProcesaSolucion();
else {   Sol.IniciaComp(k);
         Sol.SigValComp(k);
         while (!Sol.TodosGenerados(k)) {
             if ( Sol.Factible(k))
                 { back_recursivo(Sol, k+1);
                   Sol.VueltaAtras(k+1);  // Actualizamos contadores
                 }
             Sol.SigValComp(k);
          } // While
       } // Else
}
```



```
void VueltaAtras( int pos )
{  if (pos==X.size()) {  return;}
    r = r+w[pos];
   X[pos] = 2;
}
```

- Aunque hasta aquí hemos especificado todo lo que es necesario para usar cualquiera de los esquemas Backtracking, resultaría un algoritmo más simple si diseñamos a la medida del problema que estemos tratando cualquiera de esos esquemas.

Procedimiento SUMASUB (s,k,r)

{Los valores de X(j), $1 \le j < k$, ya han sido determinados. $s = \sum_{1..k-1} W(j)X(j)$ y $r = \sum_{k..n} W(j)$. **Los W(j) están en orden creciente.**

Se supone que $W(1) \le M$ y que $\sum_{1..n} W(i) \ge M$}

Begin

{Generación del hijo izquierdo. Nótese que $s+W(k)+r \ge M$ ya que Fact(k-1) =true}

X(k) = 1

If s + W(k) = M Then For i = 1 to k print X(j)

Else If $(s + W(k) + W(k+1)) \le M$

Then SUMASUB (s + W(k), k+1, r-W(k))

{Generación del hijo derecho y evaluación de Fact(k) }

If $s + r - W(k) \ge M$ and $s + W(k+1) \le M$

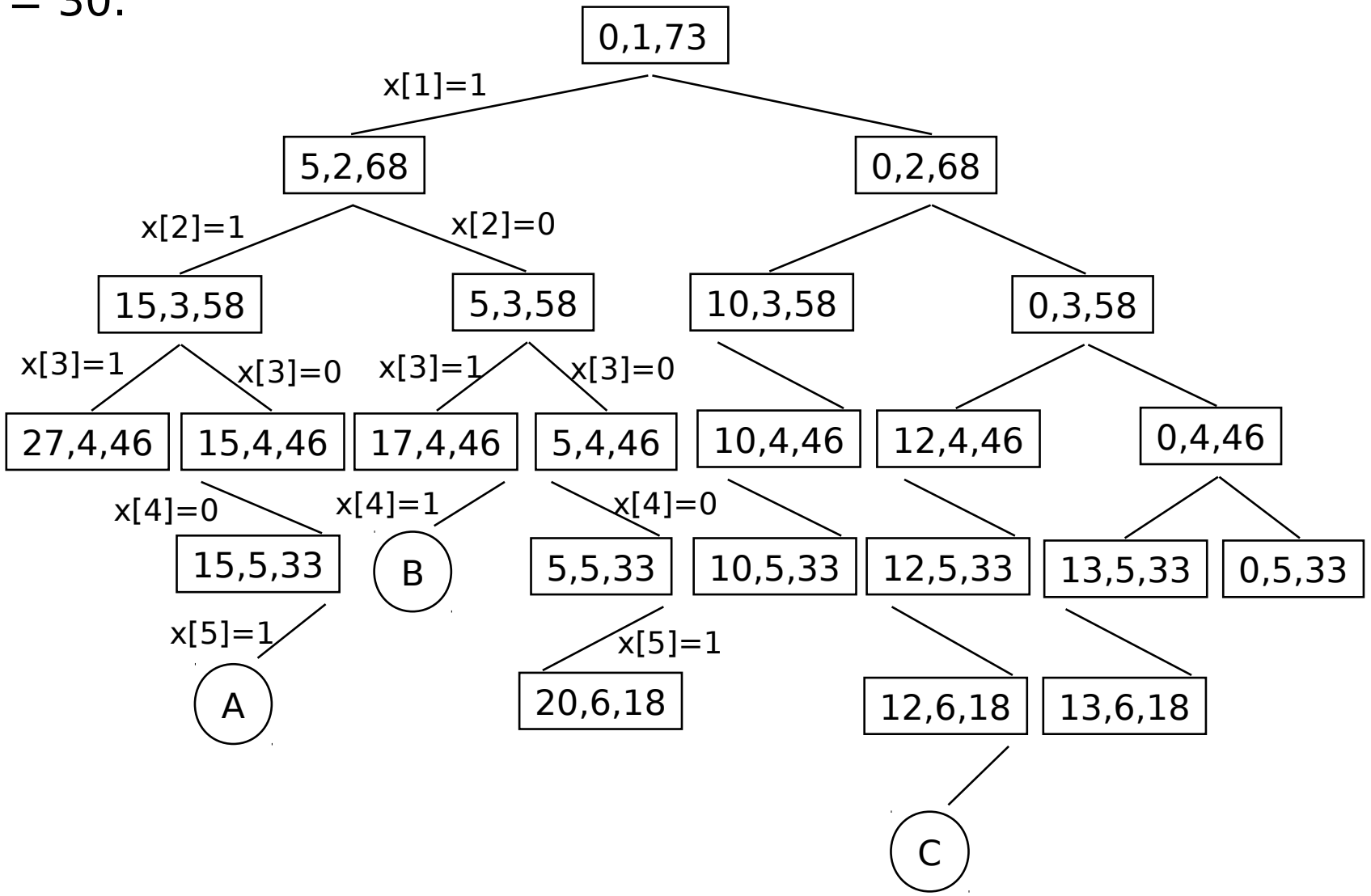
Then X(k) = 0

SUMASUB(s, k+1, r-W(k))

end

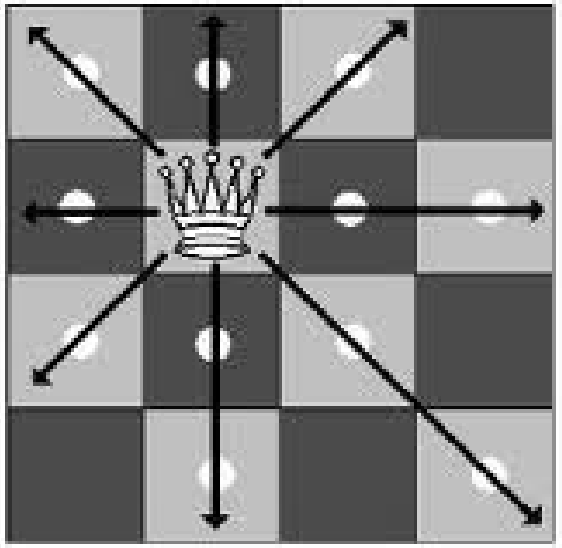
Ejemplo

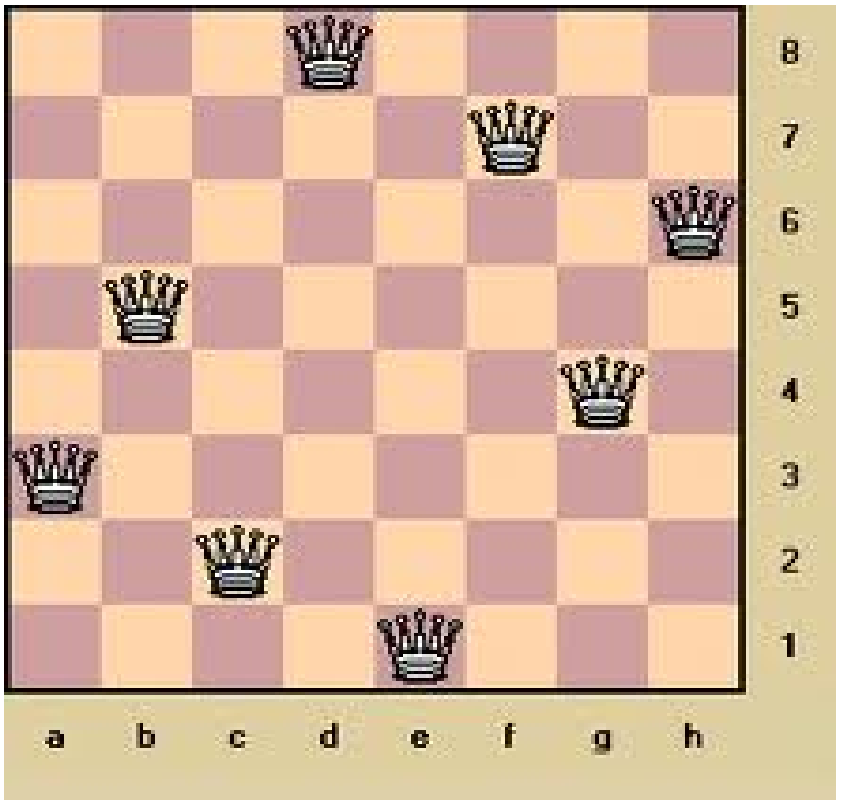
Como trabaja SUMASUB para el caso en que: W = (5, 10, 12, 13, 15, 18) y M = 30.



Problema de las 8 reinas

- Colocar 8 reinas en un tablero de tamaño 8x8 sin que se ataquen entre ellas.





Problema de las 8-reinas

Representación solucion:

- **vector<int> X[8];**

  X[i] representa la fila en la que coloca la reina de la columna i-esima ==> X[i] puede tomar 8 valores.

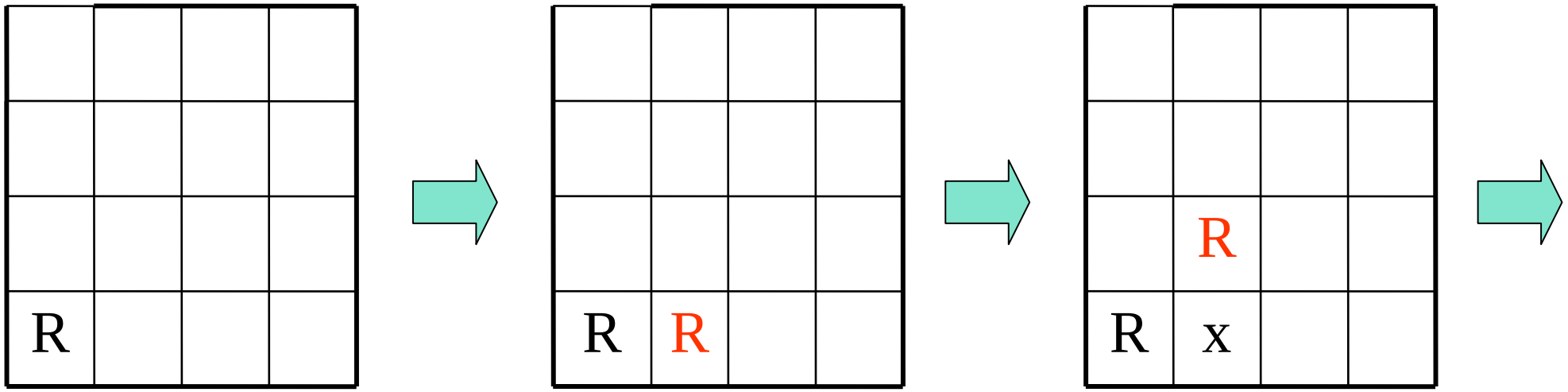
  (antes i representaba la fila en que se colocaba la reina i y X[i] la columna, son representaciones equivalentes)

  Además, dos reinas no pueden estar en la misma fila

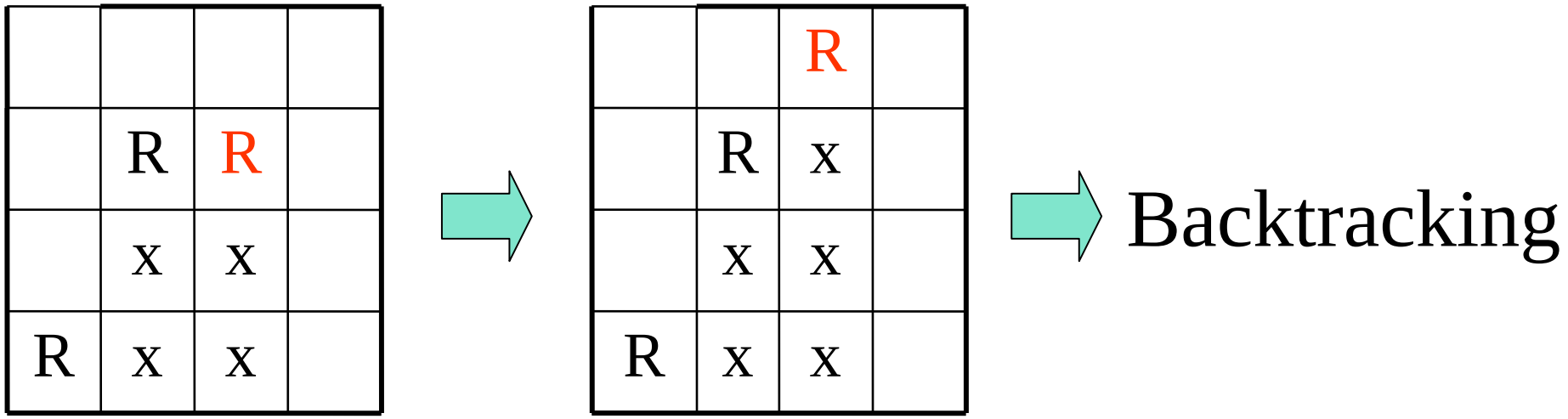
  Arbol de Estados: Arbol de permutaciones

  El numero de nodos solución es 8! = 40320.

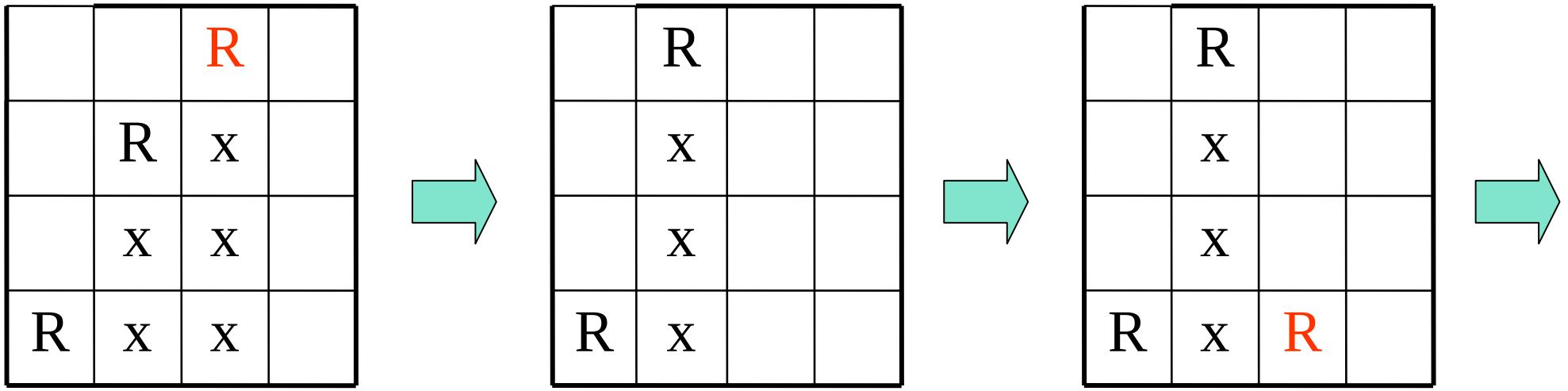
4-Reinas



4-Reinas



4-Reinas


R
R x
x x
R x x
R
x
x
R x
R
x
x
R x R
R
x
x R
R x x R
R
x
x R
R x x
R
x
x R
R x x R
R
x
x R R
R x x x

4-Reinas

	R		
	x		R
	x	R	x
R	x	x	x

→

	R		R
	x		x
	x	R	x
R	x	x	x

→

	R		
	x	R	
	x	x	
R	x	x	

→

	R	R	
	x	x	
	x	x	
R	x	x	

→

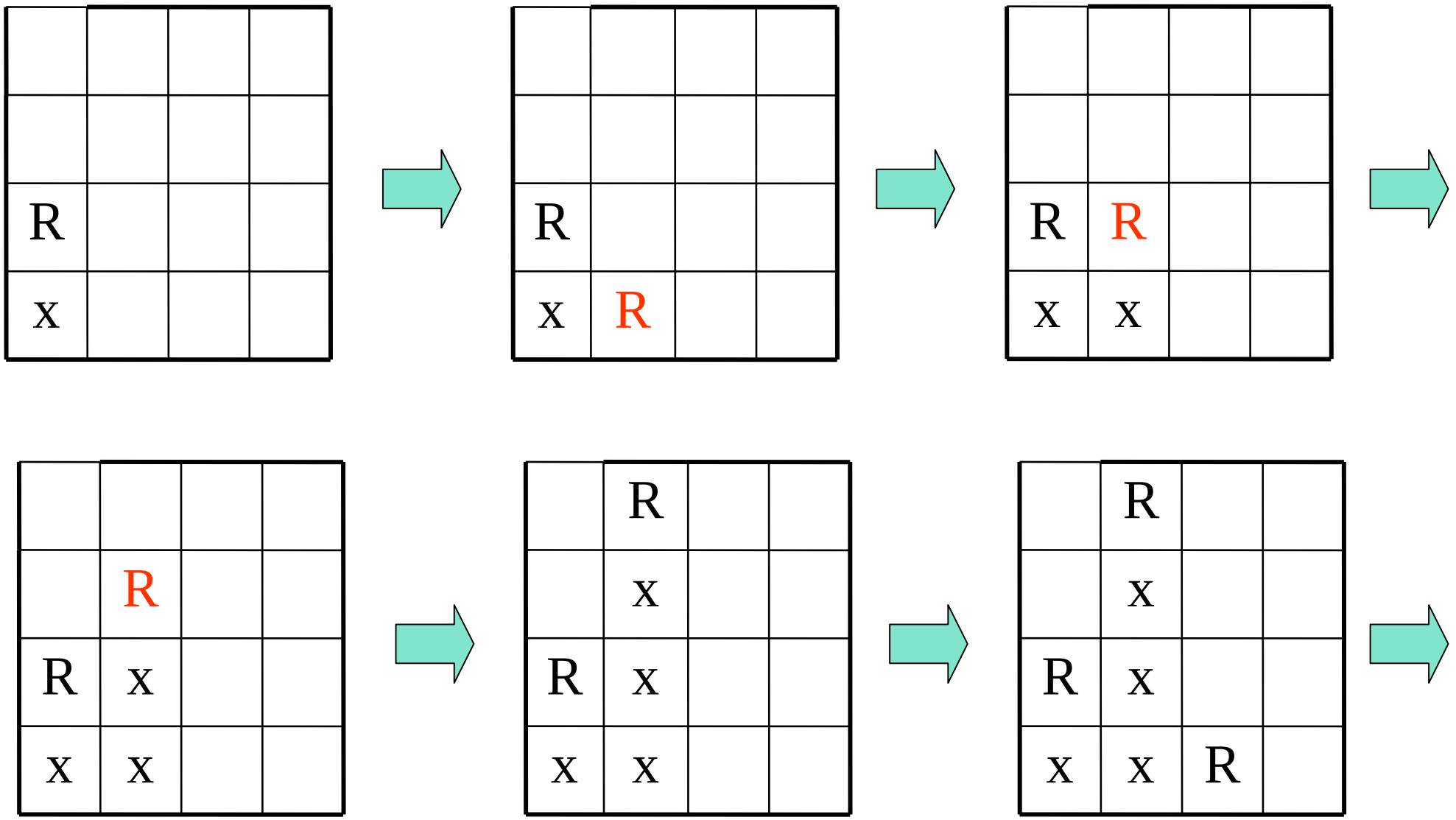
	R		
	x		
	x		
R	x		

→

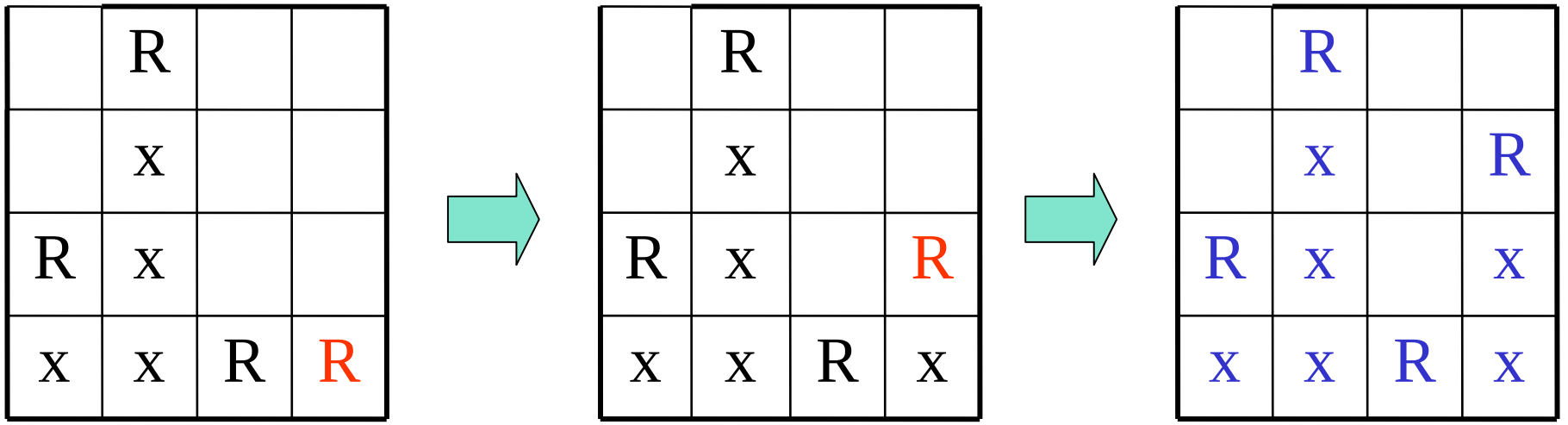
R			
x			

→

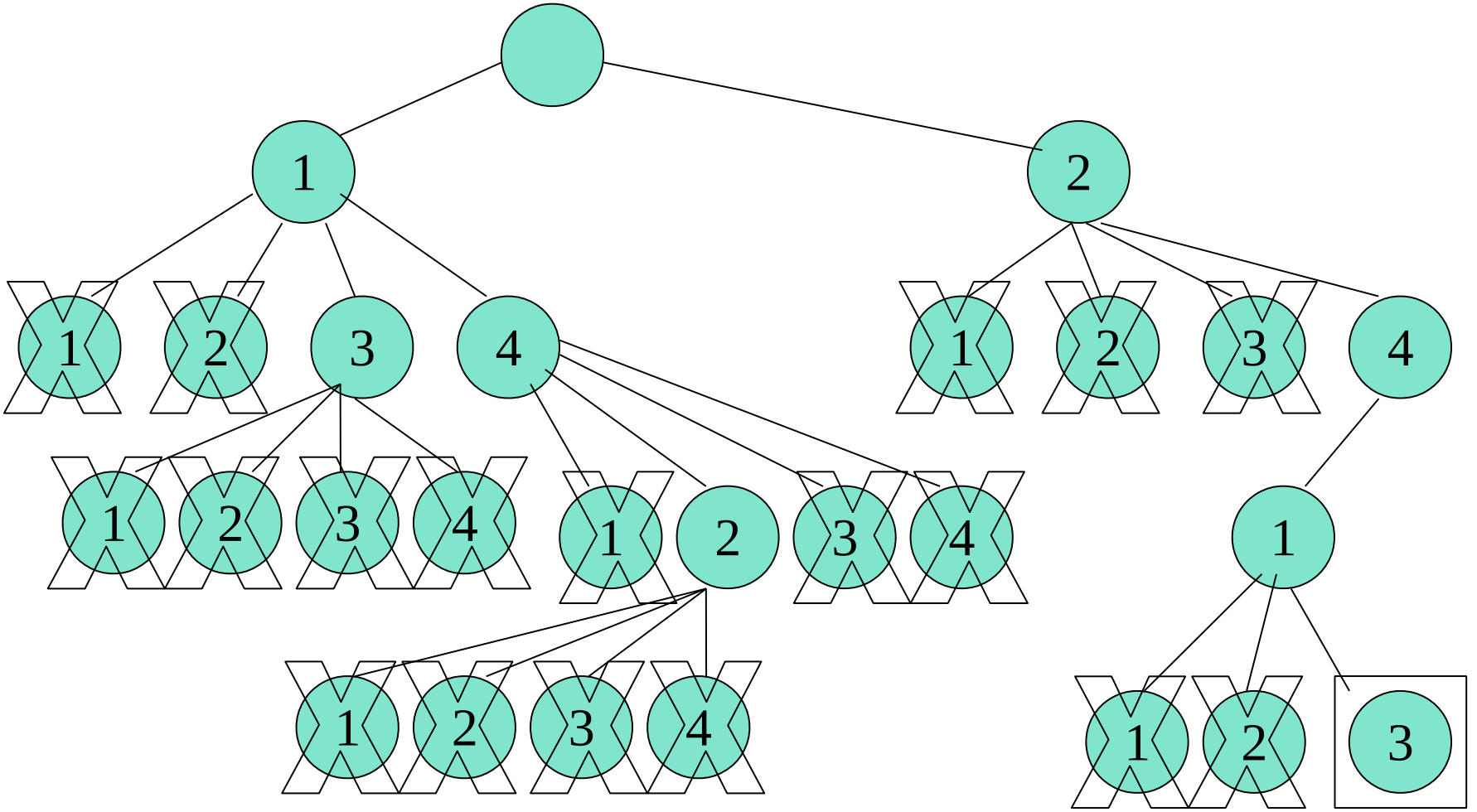
4-Reinas



4-Reinas



4-Reinas



Funciones de la Clase Solución:

```
void IniciaComp(int k)   {  X[k] =NULO; }   // columna k, fila 0
void SigValComp(int k)   { X[k]++; }    // Siguiente fila
bool TodosGenerados(int k) const  {return X[k]==END;}
bool Solucion::Factible(int k)
{ // Si X[k]=j,  la reina de la columna k, situada en la fila j,
 // no puede acosar a ninguna de las k-1 anteriores reinas.
```

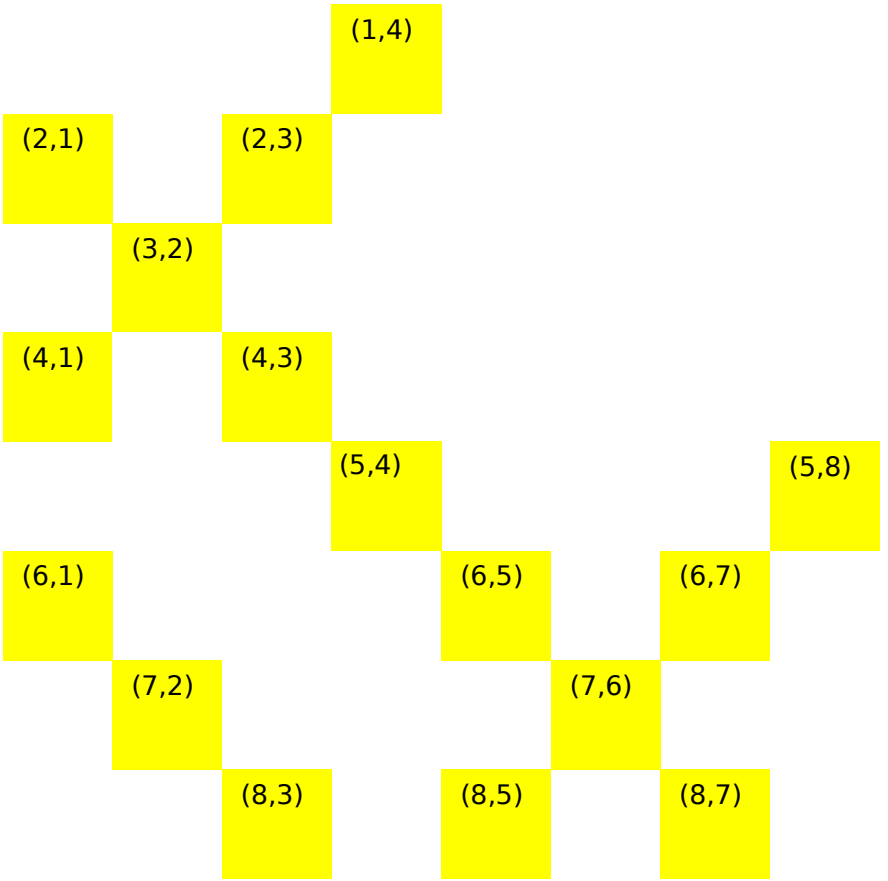
Por tanto DEBE

- Estar en una columna libre (esta implícito en la formulación de la solución)
- Estar en fila libre
- Estar en una diagonal izquierda libre
- Estar en una diagonal derecha libre

```
}
```

8 reinas: factibilidad

- ¿Cómo comprobar que dos reinas no están en la misma diagonal?



8 reinas: diagonales

- Si las casillas del tablero se numeran como una matriz A(1..n,1..n), cada elemento en la misma diagonal que vaya de la parte superior izquierda a la inferior derecha (diagonal derecha), tiene el mismo valor "fila-columna".
- También, cualquier elemento en la misma diagonal que vaya de la parte superior derecha a la inferior izquierda (diagonal izquierda), tiene el mismo valor "fila+columna".
- Si dos reinas están colocadas en las posiciones (i,j) y (k,l), estarán en la misma diagonal si y solo si,

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

- La primera ecuación implica que

$$j - l = i - k$$

- La segunda que

$$j - l = k - i$$

- Así, dos reinas están en la misma diagonal si y solo si

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

8 reinas: factibilidad

- Si asignamos un valor a la reina de la columna k, X[k], esa solución será factible si

```
factible(k) {
For i=1 to k-1 do
        If X[i] = X[k] or ABS(X[i]-X[k]) = ABS(i-k)
            return (false)
 return (true)
}
```

Pero esto consume O(k)!!

Chequear X[k]=j es válido?

- Testear si la fila j-ésima está libre

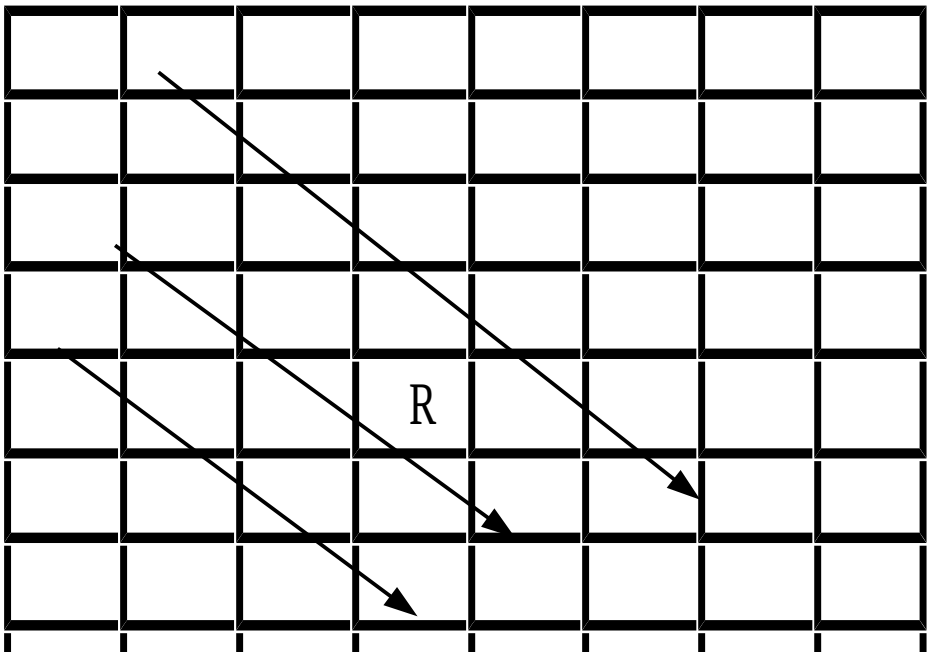
bool filas[8]={1,1,1,1,1,1,1,1}; // inicio.

if (fila[j]==1) fila[j]=0; La fila está libre, la ocupamos

- Problema diagonal derecha?

Fila-columna es fijo

(Rango -7,..,7)



Chequear X[k]=j es válido? (cont.)

- Diagonal derecha DD

```
bool DD[15]={1,1,1,...,1};
 if (DD[k-j+7]) DD[k-j+7]=0; // Marcamos ocupada
```

- Problema diagonal Izquierda (DI)

  Fila + Columna es fijo:(Rango 0,..,14) (ó 2..16)

```
bool DI[15]={1,1,1,...,1};
  if (DI[k+j]) DD[k+j]=0;  // Marcamos ocupada
```

```
bool Solucion::Factible(int k)

{ // Si X[k]=j, entonces la reina de la columna k, situada en fila j,
  // no puede acosar a ninguna de las k-1 anteriores reinas.

  If (fila[X[k]] && DD[k-X[k]+7] && DI[k+X[k]])

     { fila[X[k]]=0;

       DD[k-X[k]+7] =0;

        DI[k+X[k]]=0;

        return 1;

  } else return 0;

}
```

```
Void n_reinas(Solucion & sol, int i)
{
if ( i == Sol.size() )   Sol.ImprimeSolucion();
else {
    Sol.IniciaComp(i);
    Sol.SigValComp(i);
    while (!Sol.TodosGenerados(i)) {
      if (Sol.Factible(i)){
           n_reinas(Sol, i+1);
           Sol.LiberarPosiciones(i);  // tras vuelta atras,
      }                                //  libera posiciones ocup.
      Sol.SigValComp(i);
     }
  }
}
```

El algoritmo imprime todas soluciones,....

Cómo para al encontrar la 1ª?

```
Void n_reinas(Solucion & sol, int i, bool & solfound)
{
if ( i == Sol.size() )   {Sol.ImprimeSolucion(); solfound=true;}
else {
    Sol.IniciaComp(i);
    Sol.SigValComp(i);
    while (!Sol.TodosGenerados(i) && !solfound) {
      if (Sol.Factible(i)){
           n_reinas(Sol, i+1,solfound);
           Sol.LiberarPosiciones(i);  // tras vuelta atras,
      }                                    //  libera posiciones ocup.
      Sol.SigValComp(i);
     }
  }
}
```

El algoritmo imprime todas soluciones,....

Cómo para al encontrar la 1ª?

Deteniendo las llamadas recursivas usando un valor booleano

Solución 8 Reinas

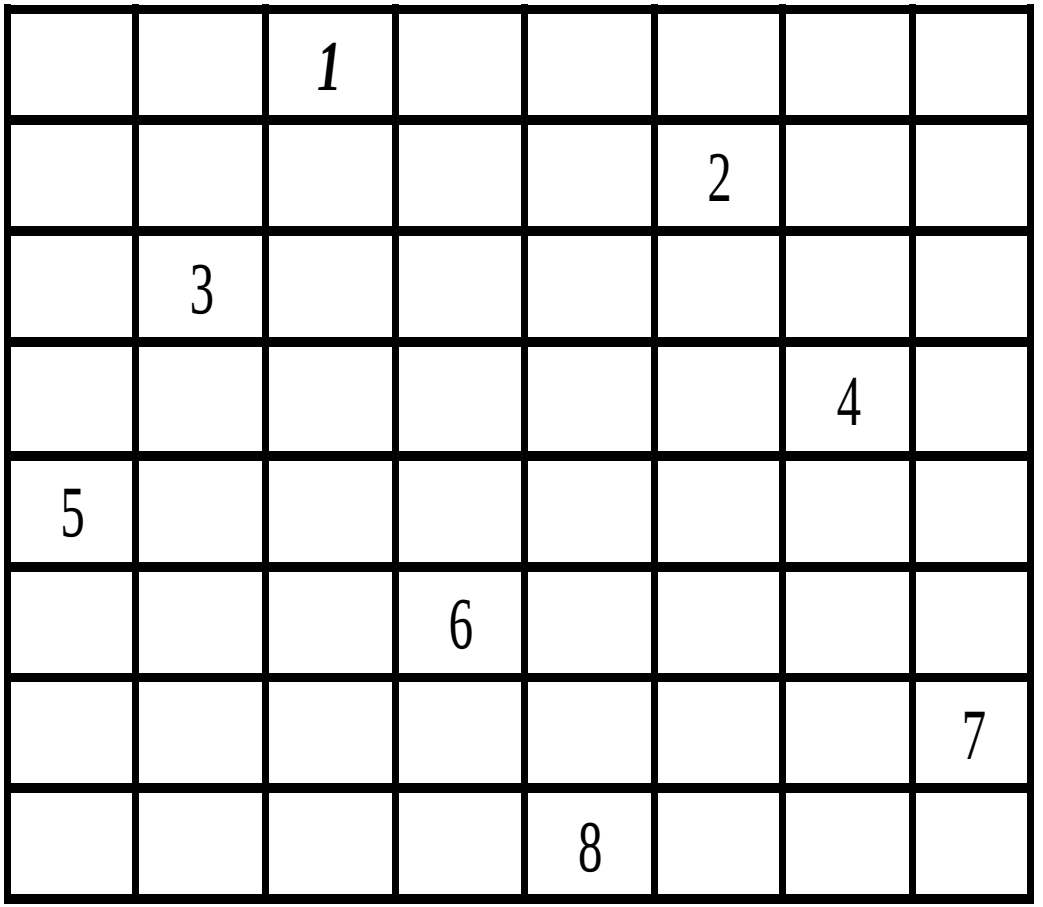
- Nodos posibles

$$1+\sum_{j=0}^{7}\left(\prod_{i=0}^{j}(8-i)\right)$$

$$= 69281.$$

Se generan 112 nodos para alcanzar la primera solución.

570 nodos para obtener todas.



Solución para las N reinas no recursivo

Procedimiento NREINAS(n)

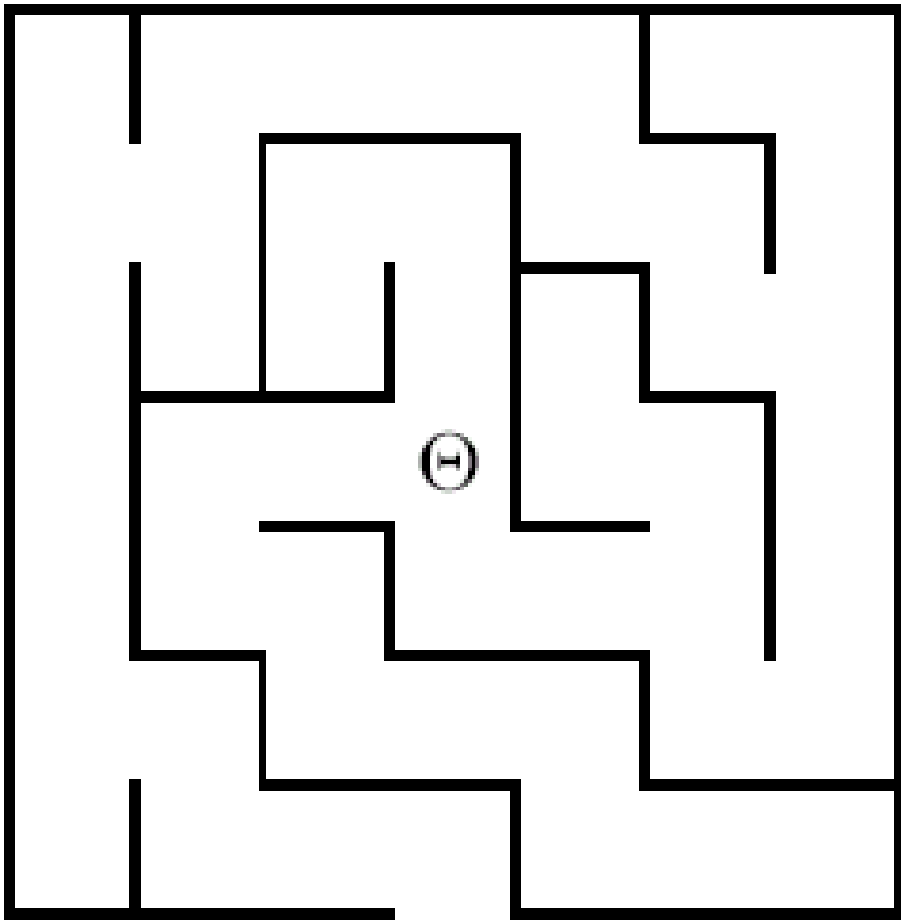
{Usando backtracking este procedimiento imprime todos los posibles emplazamientos de n reinas en un tablero nxn sin que se ataquen}

```
X(1) = 0; k = 1;                          {k es la columna actual}
while (k > 0) {                           {hacer para todas las columnas}
  X(k) = X(k) + 1;                        {mover a la siguiente fila}
  while ((X(k) <= n) && (!factible(k))    {puede moverse esta reina?}
       X(k) = X(k) + 1;
  if (X(k) <= n)                          {Se encontró una posición}
     if (k== n)                           {Es una solución completa?}
          print (X)
     else {k = k + 1; X(k) = 0;}          {Ir a la siguiente fila}
  else k = k – 1;                         {Backtrack}
}
```

Laberintos y Backtracking

Un laberinto puede modelarse como una serie de nodos.
En cada nodo hay que tomar una decisión que nos conduce a otros nodos.

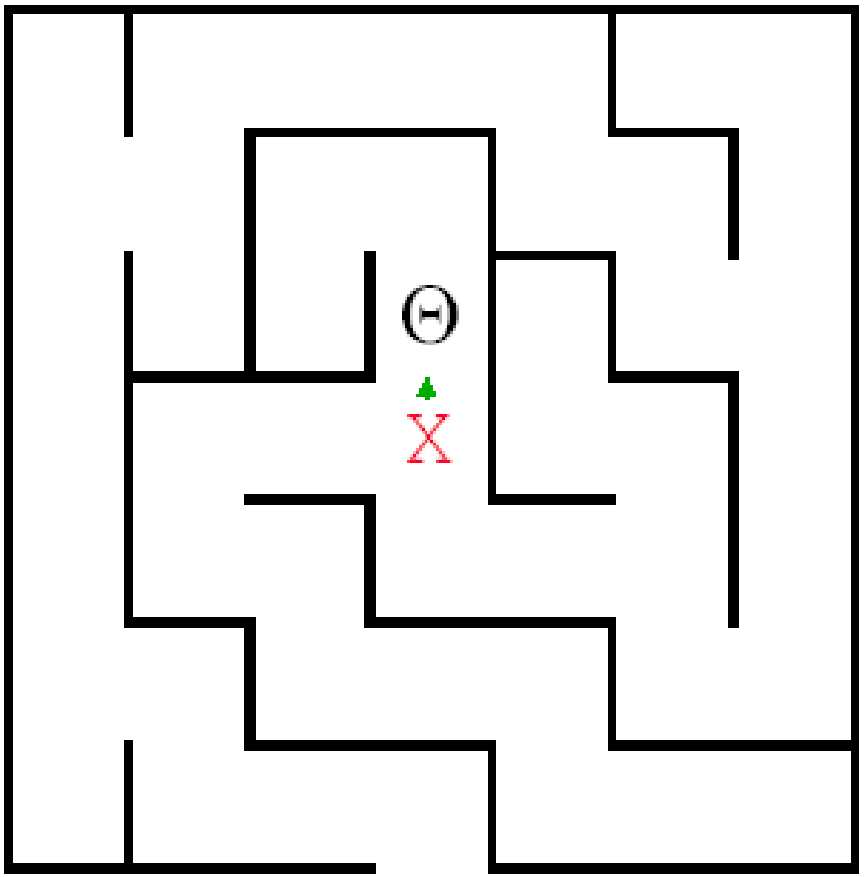
Un laberinto sencillo



Buscar en el laberinto hasta encontrar una salida. Si no se encuentra una salida, informar de ello.

Hay 4 movimientos posibles.

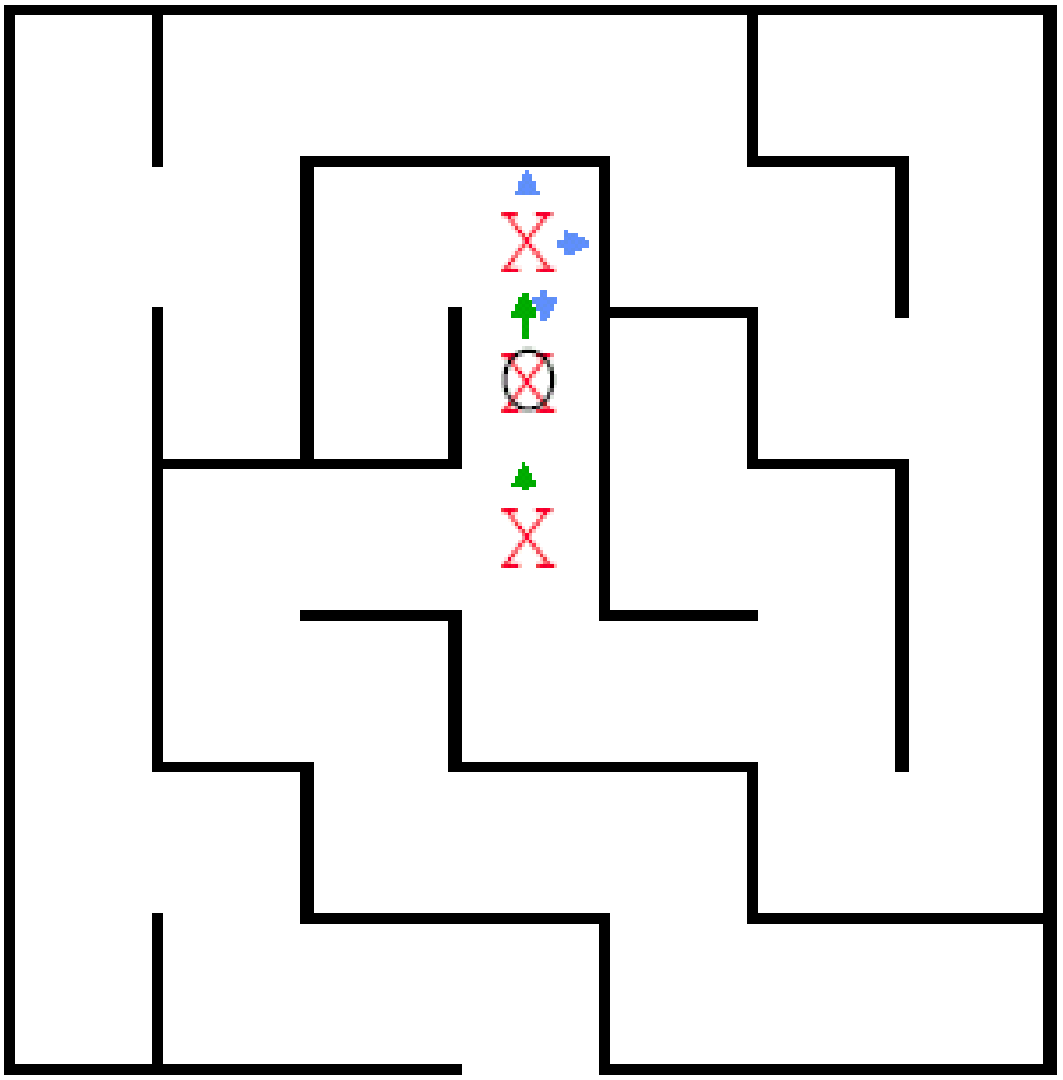
Backtracking en Acción



Intentamos sucesivamente movernos en cada una de las 4 direcciones.

Inicialmente nos movemos hacia el norte (marcando las posiciones por las que pasamos) .

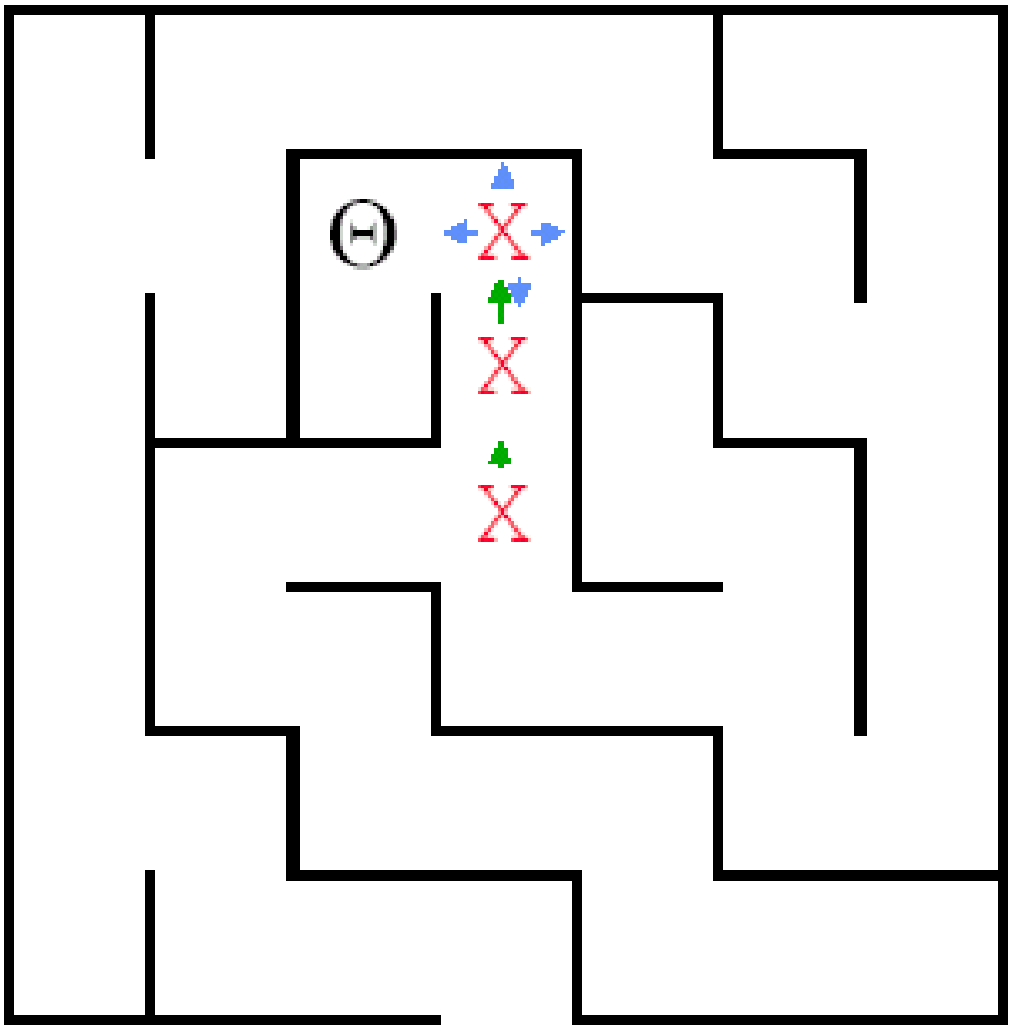
Backtracking en Acción



Aqui nos movemos hacia el Norte de nuevo, pero ahora la dirección Norte esta bloqueada por un muro.

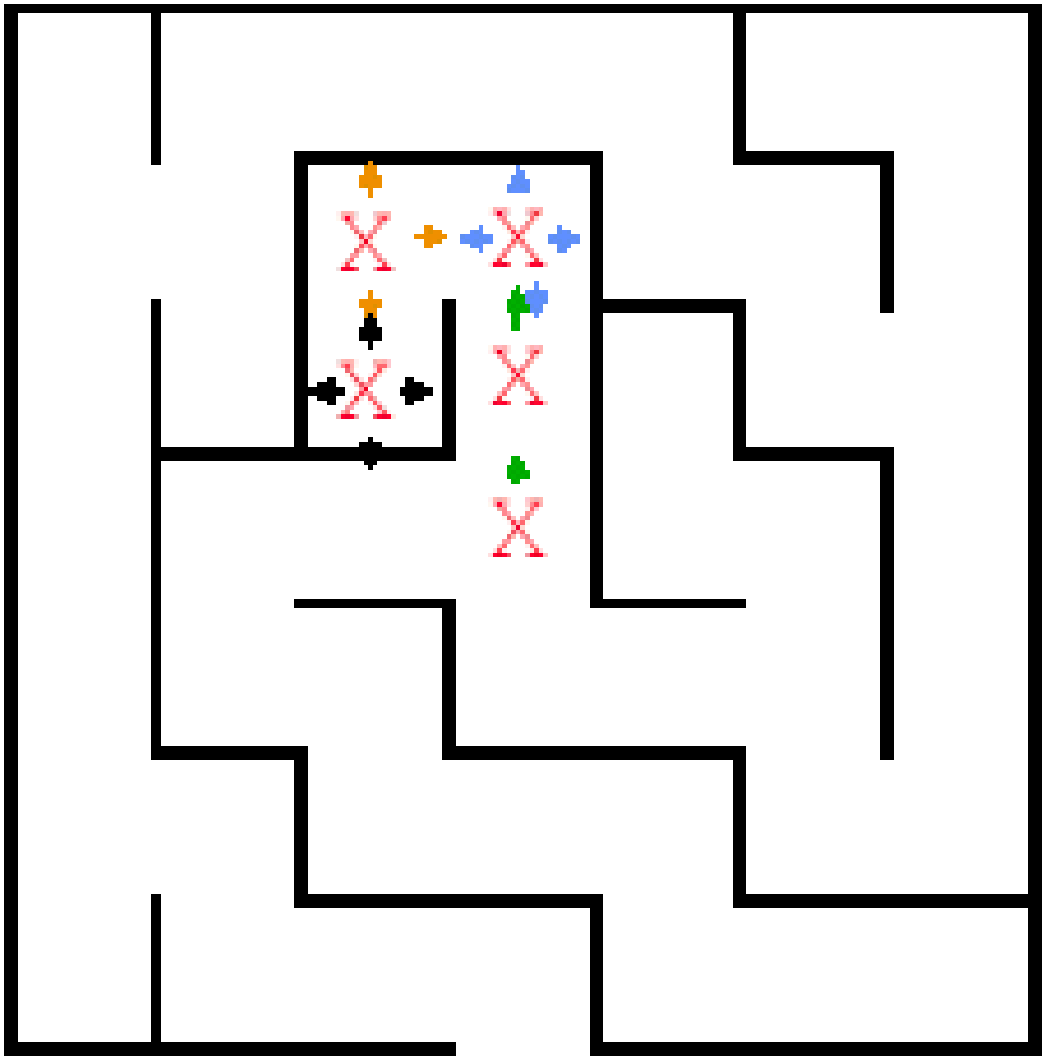
El Este también esta bloqueado, por lo que intentamos el Sur. Esa acción descubre que ese punto esta marcado.

Backtracking en Acción



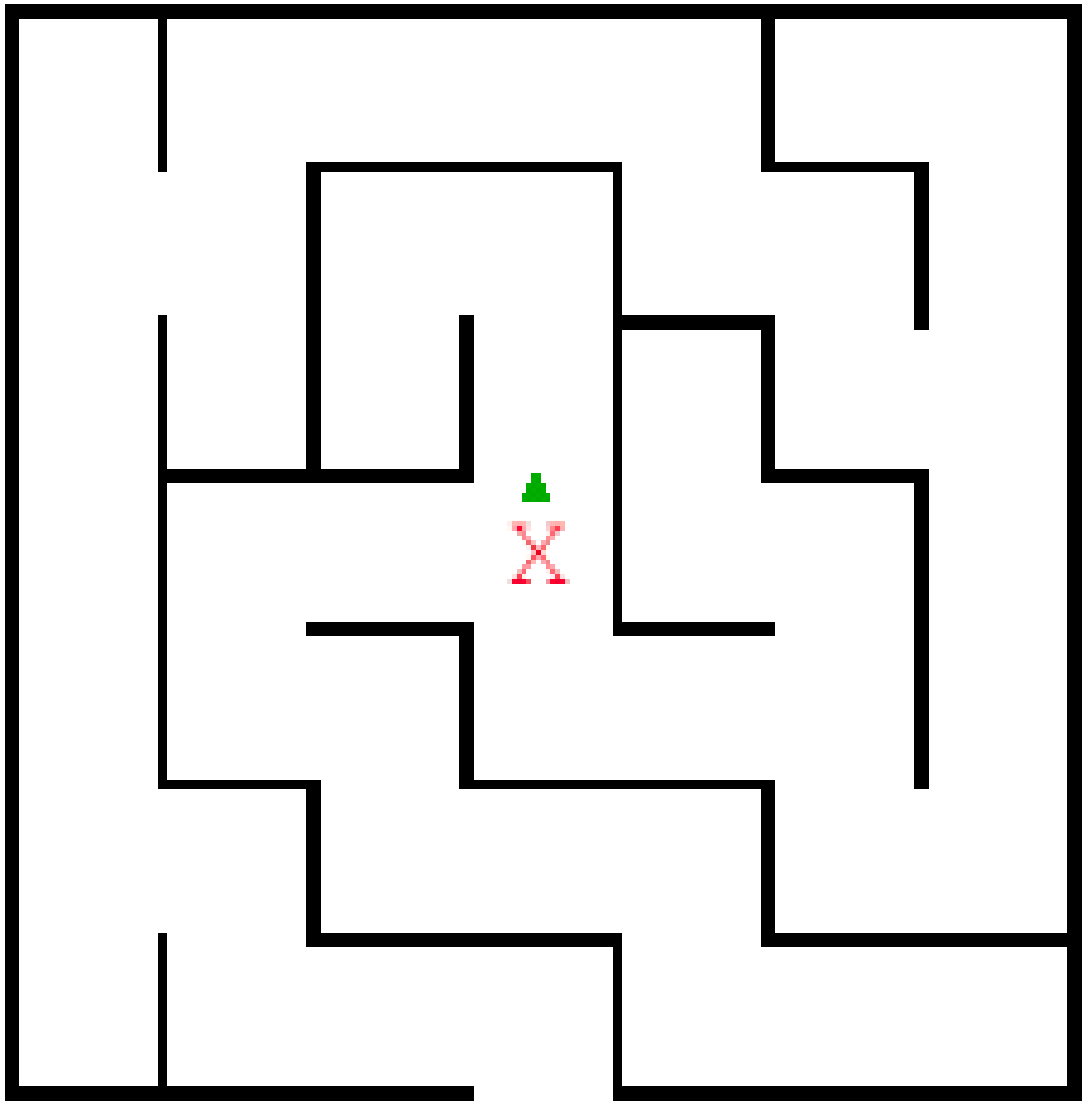
Por tanto el siguiente movimiento que podemos hacer es hacia el Oeste

Backtracking en Acción



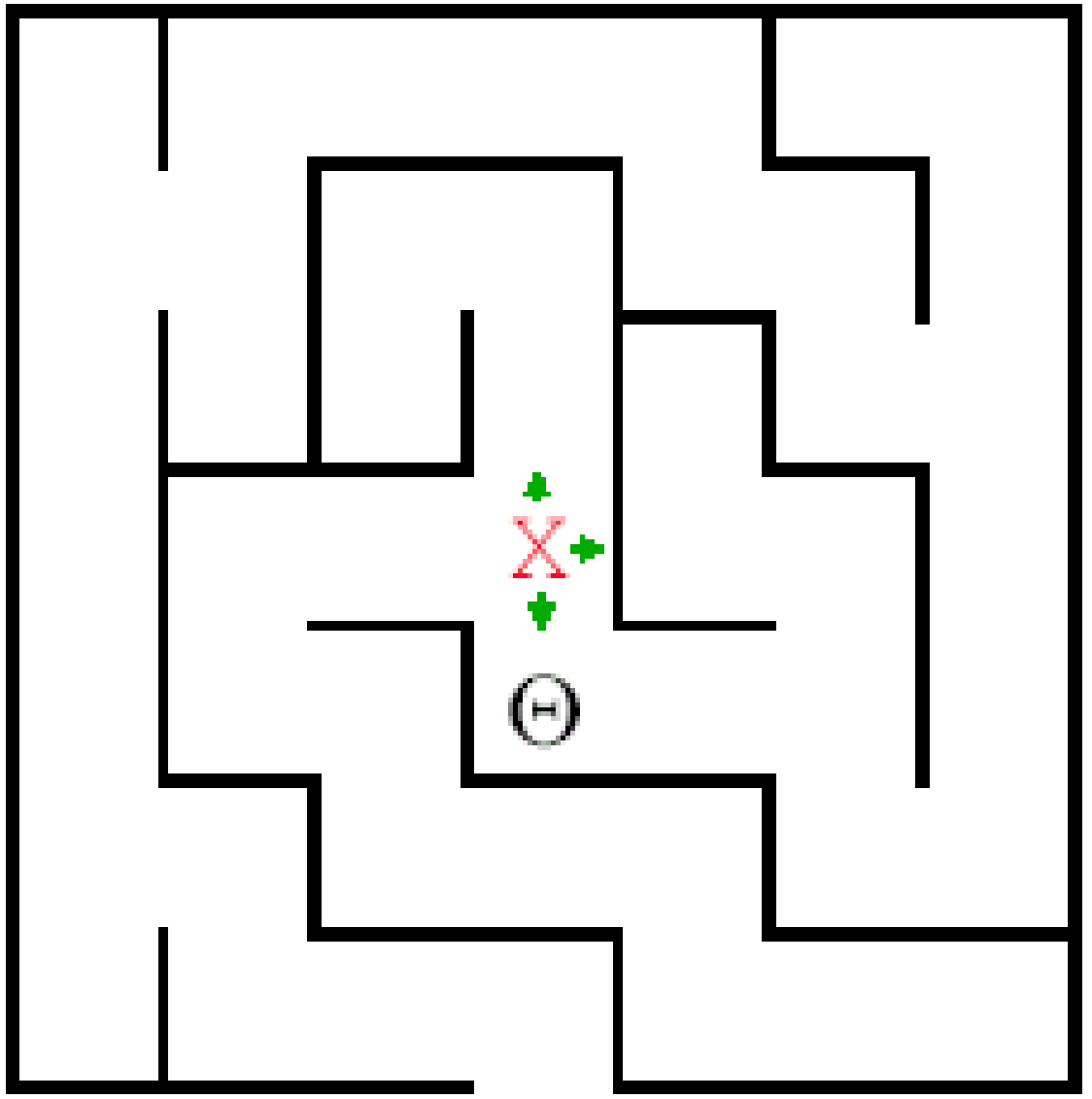
Este camino llega a un nodo (final) muerto .
¡Por tanto es el momento de hacer un backtrack!

Backtracking en Acción



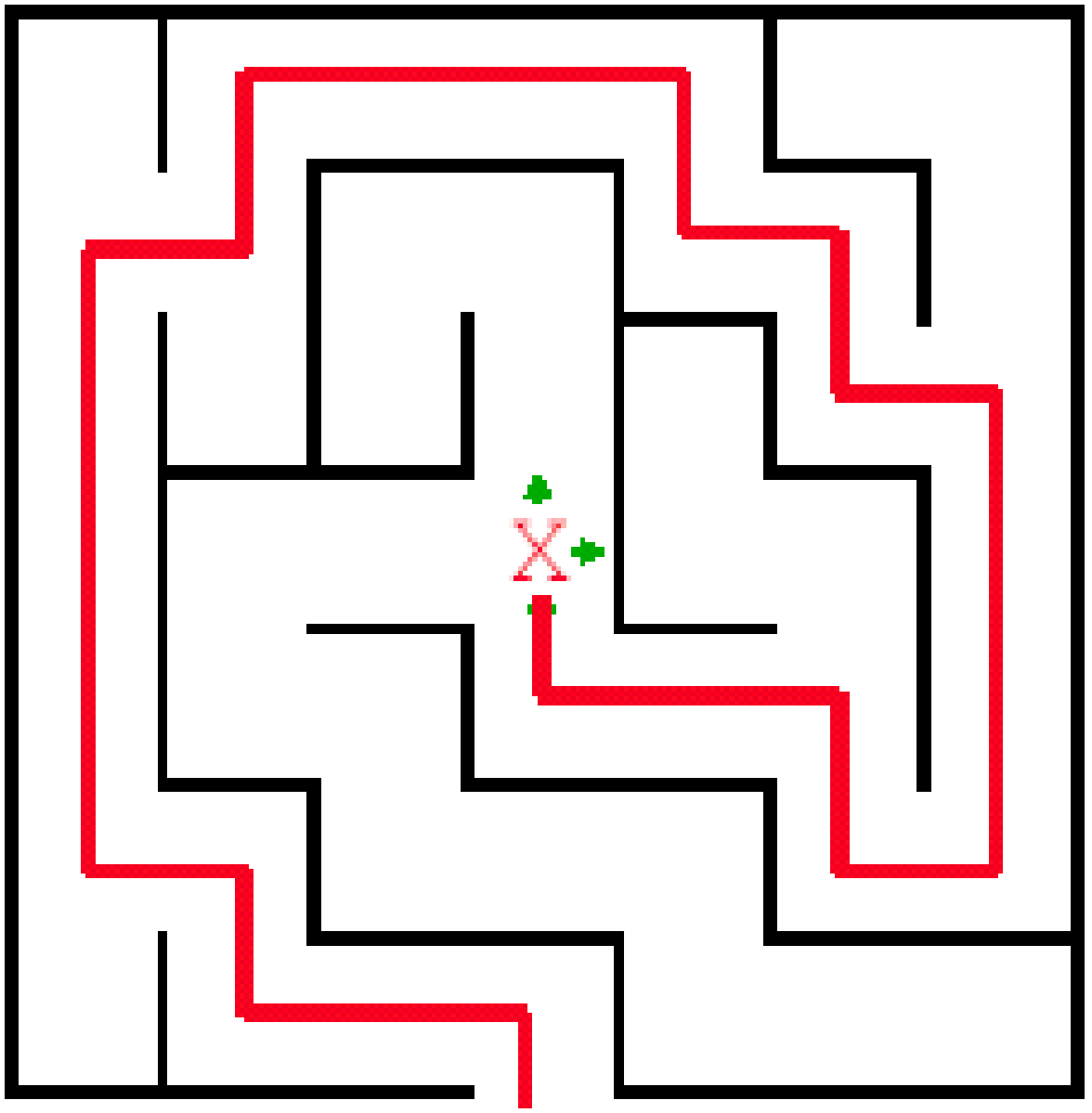
Se realizan sucesivas llamadas recursivas hasta volvernos a encontrar aquí

Backtracking en Acción



Intentamos ahora el Sur

Primer camino que se encuentra



En este problema la longitud del vector solución no es constante, es la lista de posiciones hasta llegar a la salida.

El problema del coloreo de un grafo

- Sea G un grafo y n un número entero positivo. Queremos saber si los nodos de G pueden colorearse de tal forma que no haya dos vértices adyacentes que tengan el mismo color, y que solo se usen n colores para esa tarea.

- Este es el problema de la n-colorabilidad.

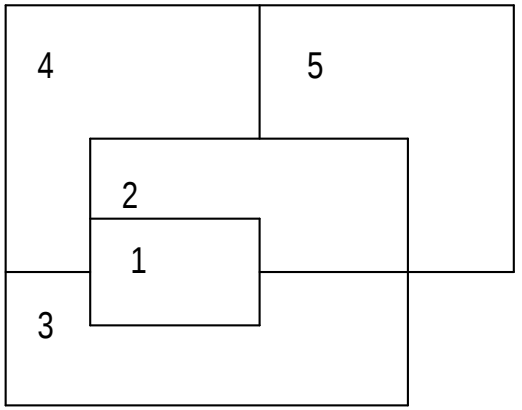
- El problema de optimización de la n-colorabilidad, pregunta por el menor número n con el que el grafo G puede colorearse. A ese entero se le denomina Número Cromático del grafo.

El problema del coloreo de un grafo

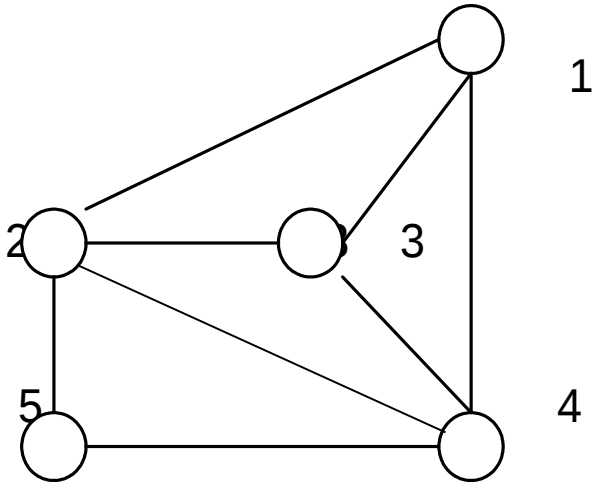
- Un grafo se llama plano si y solo si puede pintarse en un plano de modo que ningún par de aristas se corten entre sí.
- Un caso especial famoso del problema de la n-colorabilidad es el problema de los cuatro colores para grafos planos que, dado un mapa cualquiera, consiste en saber si ese mapa podrá pintarse de manera que no haya dos zonas colindantes con el mismo color, y además pueda hacerse ese coloreo solo con cuatro colores (sí se puede).
- Este problema es fácilmente traducible a la nomenclatura de grafos.

El problema del coloreo de un grafo

- El mapa

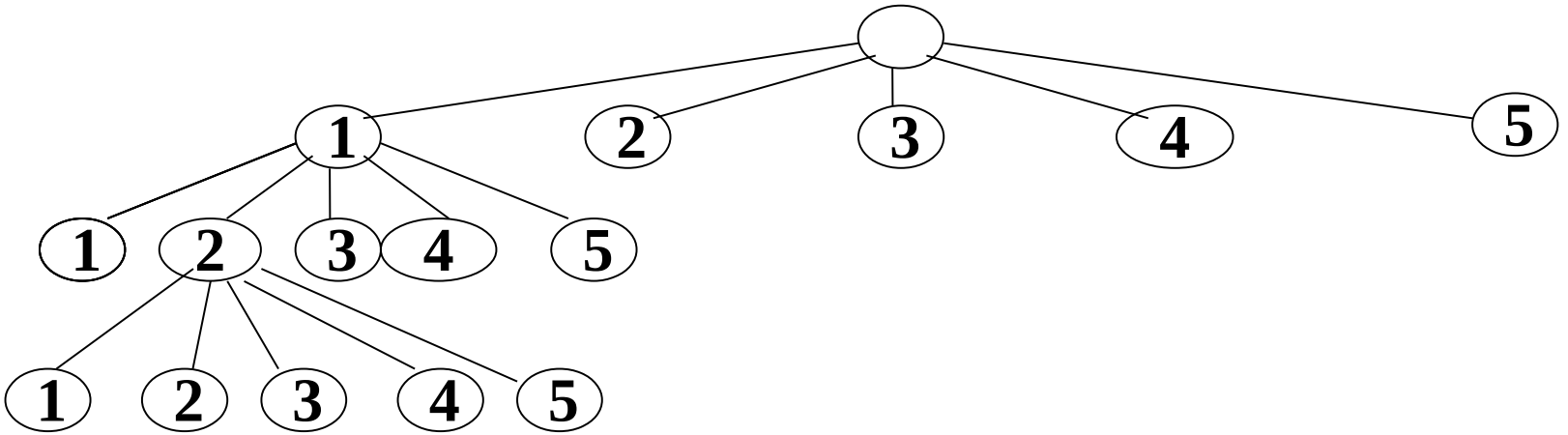


- puede traducirse en el siguiente grafo



Coloreo: Restricciones

- Restricciones Explícitas
  - X[i] toma valores en {NULO, 1, 2, 3, 4, ..., N, END}
- Restricciones Implícitas.
  - X[i] !=X[j] si i y j son paises adyacentes
- Arbol de Estados
  - Arbol de Combinaciones



Coloreo: Estruct. De datos

- Utilizaremos una estructura de datos simple que nos permita chequear para cada pais, todos sus adyacentes
  - vector<vector<int> > Mapa;
  - Mapa[i][j] es el j^{th} pais adyacente al pais i
    - `Mapa[5][3]==8` significa que el 3^{r} pais adyacente a 5 es 8

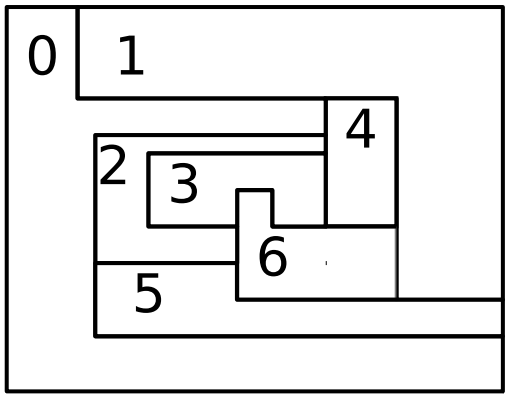
Coloreo: Ejemplo Mapa

mapa.resize(7);

mapa[0].push_back(1);

mapa[0].push_back(2);

mapa[0].push_back(4); mapa[0].push_back(5);



mapa[1].push_back(0); mapa[1].push_back(4);

mapa[1].push_back(6); mapa[1].push_back(5);

mapa[2].push_back(0); mapa[2].push_back(3);

mapa[2].push_back(4);mapa[2].push_back(5);mapa[2].push_back(6);

…….

Clase Solucion:

```
void IniciaComp(int pais)
   {  X[pais] = 0;  // Valor NULO   }


void SigValComp(int pais)
{ // Orden de los valores   0 -> 1 -> 2 -> ...-> N → N+1
  X[pais]++;
}
bool TodosGenerados(int pais) const
{
    return X[pais]== N+1; //END
}
```

```
bool Factible(int pais) {
for (int i = 0; i < M[pais].size(); i++) {
   int ith_adyacente = M[pais][i];
   if (X[ith_adyacente] == X[pais])
      return false;
    }
 return true;
}
void VueltaAtras(int pais ){
  if (pais==X.size()) {  return;}
  X[pais] = 0;
}
```

```
void back_coloreo(Solucion & Sol, int k) {
if ( k == Sol.Size())   Sol.ProcesaSolucion();
else {
    Sol.IniciaComp(k);
    Sol.SigValComp(k);
    while (!Sol.TodosGenerados(k)) {
       if ( Sol.Factible(k)) {
            back_coloreo(Sol, k+1);
            Sol.VueltaAtras(k+1); } // no es imprescindible
      Sol.SigValComp(k);
    } // While
 } // Else
}
```

Soluciones:
1221334
2112334
3223114

Ojo: Son la misma solución, !!!

Problema de la Mochila

- Cuál es el árbol de estados?

  El espacio de soluciones consiste en las 2^n formas de asignar valores 0 ó 1 a cada objeto (seleccionar o no el objeto)

  Es por tanto igual al del problema de la suma de subconjuntos.

  Emplearemos un árbol binario: los hijos de un nodo (x1,...,xk) son (x1,...,xk,0) y (x1,...,xk,1) (también se podría emplear un árbol combinatorio)

Funciones de la Clase Solución:

```
void IniciaComp(int k)
        {  X[k] = -1;  // valor NULO }
void SigValComp(int k)
        { X[k]++;     // Siguiente valor del dominio. -1->0->1->2}
Bool TodosGenerados(int k) const
        {return X[k]==2; // END}
bool Solucion::Factible(int pos) const
{//El peso de los objetos seleccionados no sobrepase la capacidad de la mochila
 float ps =0.0;
  for (int k=0; k<=pos ; k++)   ps += X[k]*p[k];
   return  (ps<=M) ;   }
```

Ejemplo:

Número de objetos: 7

Tamaño M=11.0

Beneficio ={30, 20, 36, 40, 48, 6, 2};

Pesos ={ 3, 2, 4, 5, 6, 3, 5};

Rel B/P ={ 10, 10, 9, 8, 8, 2, 0.4};

Nodos Posibles: (tamaño árbol de estados)

255 (2^8-1) , de los que 128 (2^7) son nodos solución

Nodos Generados: 162, de los cuales 43 son solución

Solución óptima: 98, X={1,1,0,0,1,0,0}

Se podría mejorar el algoritmo?.

Mejorando el algoritmo ...

1) Mejorar la función de Factibilidad:

Una solución no es factible cuando:

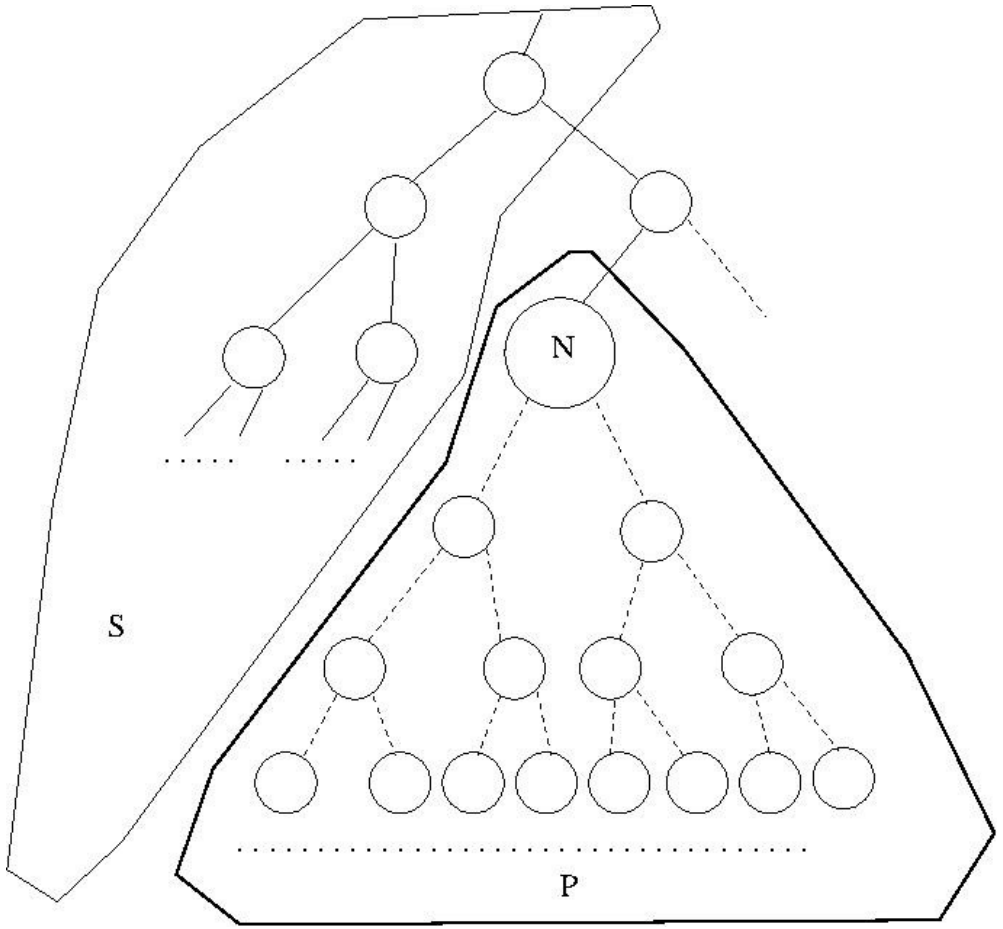
- La suma del peso de los objetos seleccionados sobrepasa la capacidad de la mochila (ya considerado)
- Sea N un nodo interno del árbol y sea S la mejor solución que se obtiene al evaluar todos los nodos solución (hojas) que se visitan antes que N

  Entonces, podemos podar N si la mejor solución, P, que se podría alcanzar al expandir el subárbol no pueda superar el valor de la mejor solución obtenida hasta el momento, S: USAR COTAS

2) Expandir primero la rama a priori más prometedora, con la idea de alcanzar antes la solución

Para ello ordenamos los objetos de forma que $b_i/p_i >= b_{i+1}/p_{i+1}$

Mejorar la función de factibilidad: cotas



S mejor solución encontrada hasta ahora

P mejor solución que encontraríamos explorando el Subárbol con raíz en N

Si P peor que S, podar N

1) TRAS mejorar función factibilidad:

Nodos Posibles: 255, (128 son nodos solución)

Antes: Nodos Generados: 162, (43 son solución).

Ahora: Nodos Generados: 72.

(10 son nodos solución)

(16 se han podado por sobrepasar M)

(11 se podan al no superar mejor solución S)

(35 se han explorado completos)

Solución Optima: 98, X={1,1,0,0,1,0,0}

Problema: Expandimos menos nodos, pero tardamos más en evaluar la factibilidad de los mismos.

2) Expandir **primero la rama más prometedora**, con la idea de alcanzar antes una solución.

```
void IniciaComp(int k) {  X[k]=2;  }
void SigValComp(int k)  { X[k]-- ; }
bool TodosGenerados(int k) const  {return X[k]==-1;}
```

No tiene costo, y sin embargo mejora considerablemente el rendimiento del algoritmo

Nodos Generados: 34 (antes 72).

- 3 (10) son nodos solución
- 11 (16) se han podado por sobrepasar M
- 4 (11) se podan al no superar mejor solución S
- 16 (35) se han explorado completos

Cálculo de cotas....Problema de Maximización (similar para min.)

Necesitamos 2 cotas sobre la ganancia :

- COTA GLOBAL:

  - Representa el valor de la mejor solución estudiada hasta el momento. Inicialmente, se le puede asignar el valor dado por un algoritmo greedy o empezar con valor -infinito.

  - Es una cota inferior, CI, de la solución optimal:

CI <=Optimo.

Cálculo de cotas....Problema de Maximización (similar para min.)

- La Cota Global se actualiza siempre que alcancemos una solución, H, (nodo hoja) con ganancia mejor

```
void ActualizaSolucion()
{ Si ganancia(H) > CI
  entonces { CI=ganancia(H); MejorSolucion = H;}
}
```

Cálculo de cotas.... Maximización

- COTA LOCAL:
  - Se calcula para cada nodo interno, N, generado, siendo una estimación optimista del mejor valor que se podría alcanzar al expandir el nodo, LOptimal(N).
  - Es una cota superior, CS(N): LOptimal(N)<= CS(N)
  - Debe haber un equilibrio entre eficiencia de cómputo y calidad de la cota.
  - Puede incluirse dentro de la función de Factibilidad(int k)

Cálculo de cotas.... Maximización

- PODA? Podamos el nodo N siempre que CS(N)<=CI

•Podemos asegurar que alcanzamos el óptimo? SI, porque

LOptimal(N) <= CS(N) <= CI <= Optimo

Cuanto menor sea la cota superior local (más próxima a LOptimal(N)) más probable es que podamos podar.

Cuanto mayor sea la cota inferior global (más próxima al verdadero óptimo) mejor

Cálculo Cota Local ...(max)

- Dado un nodo interno N, tenemos el vector solución que está incompleto:

  X=[decisiones tomadas]+[decisiones por tomar]

  La Cota Superior CS(N) dependerá de ambas

  CS(N) = f(DecTomadas X) + estimador(DecPorTomar X)

Interesa que la CS(N) esté lo más cerca posible de Loptimal(N).

Hay que optimizar el estimador.

Problema de la Mochila: Cota Local

X=[decisiones tomadas]+[decisiones por tomar]

X=[1,1,0,0,1,?,?,?,?,?]

Sea k la última decisión.

f(DecTomadas) = suma ganancia objetos seleccionados

$$f\left(DecTomadas\right)=\sum_{i=1}^{k} x[i]*b[i]$$

Problema de la Mochila: Cota Local

X=[decisiones tomadas]+[decisiones por tomar]

X=[1,1,0,0,1,?,?,?,?,?]

Sea k la última decisión.

Estimador(decisiones por tomar X): Ganancia del algoritmo Greedy No 0/1 para un problema de la mochila con los objetos k+1, k+2,...,n y capacidad M'

$$M' = M - \sum_{i=1}^{k} x[i] * p[i]$$

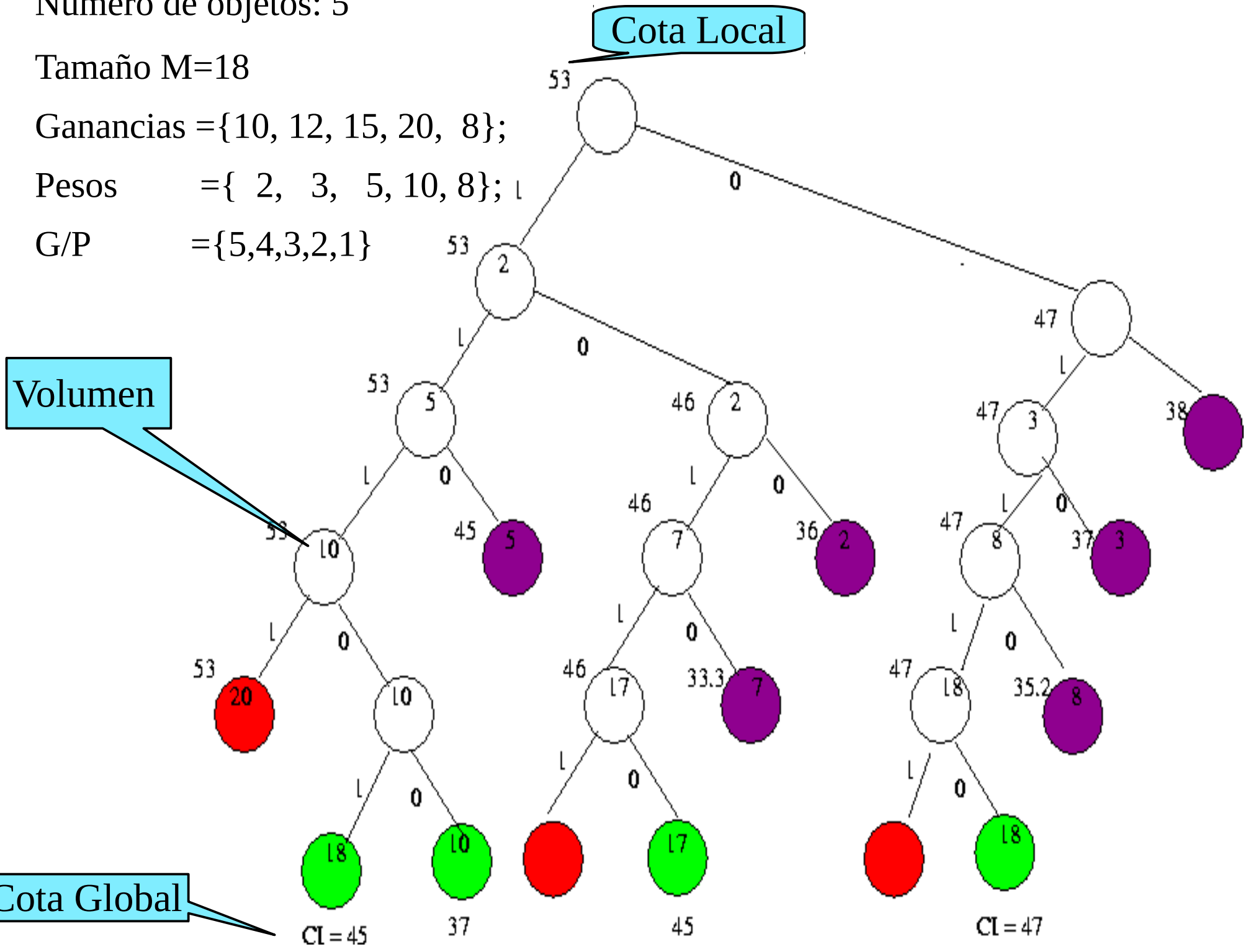
Número de objetos: 5

Tamaño M=18

Ganancias ={10, 12, 15, 20, 8};

Pesos ={ 2, 3, 5, 10, 8};

G/P ={5,4,3,2,1}



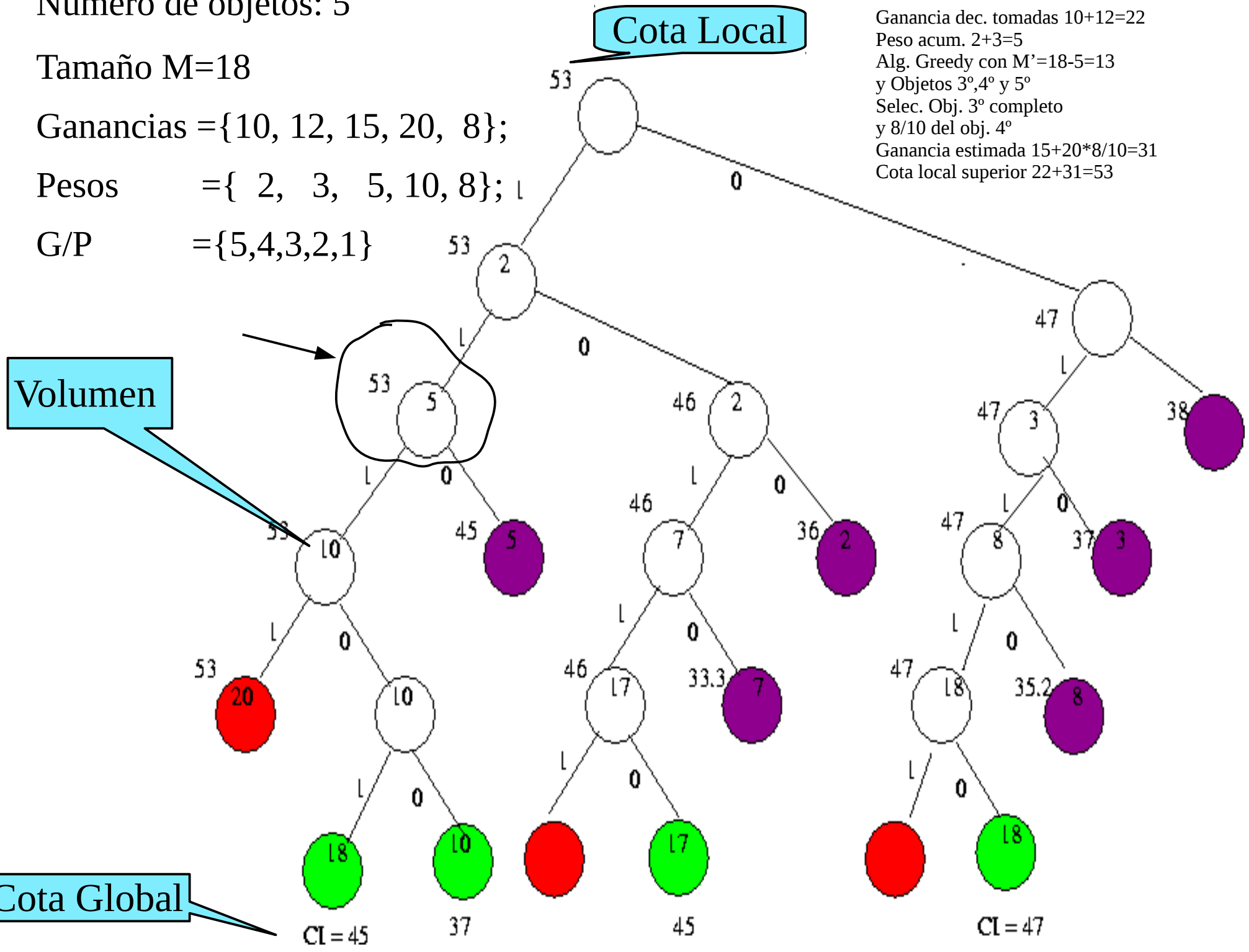
Número de objetos: 5

Tamaño M=18

Ganancias ={10, 12, 15, 20, 8};

Pesos ={ 2, 3, 5, 10, 8};

G/P ={5,4,3,2,1}



Número de objetos: 5

Tamaño M=18

Ganancias ={10, 12, 15, 20, 8};

Pesos ={ 2, 3, 5, 10, 8};

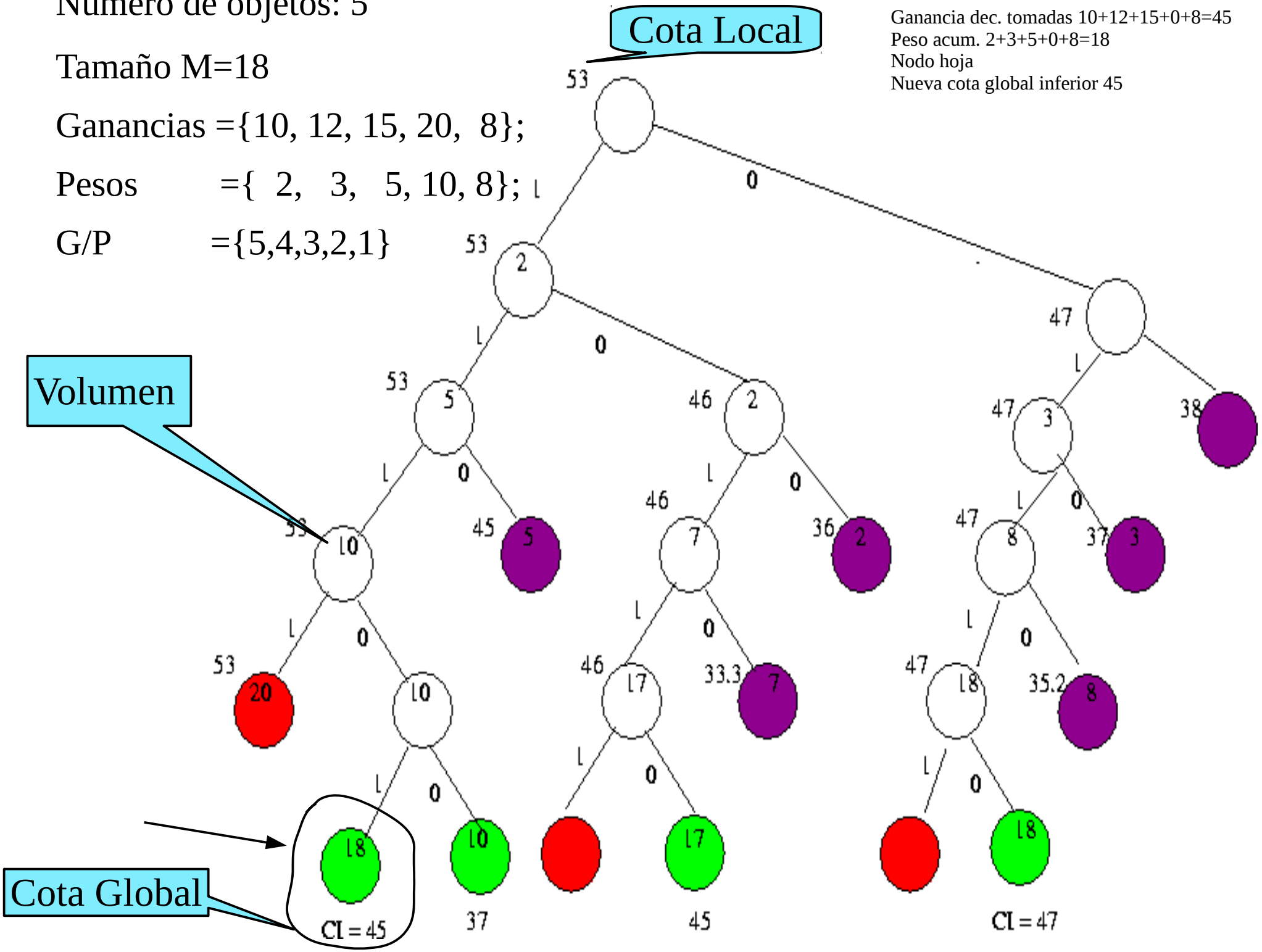
G/P ={5,4,3,2,1}

Cota Local

Ganancia dec. tomadas 10+12+15+0+8=45
Peso acum. 2+3+5+0+8=18
Nodo hoja
Nueva cota global inferior 45

Volumen

Cota Global



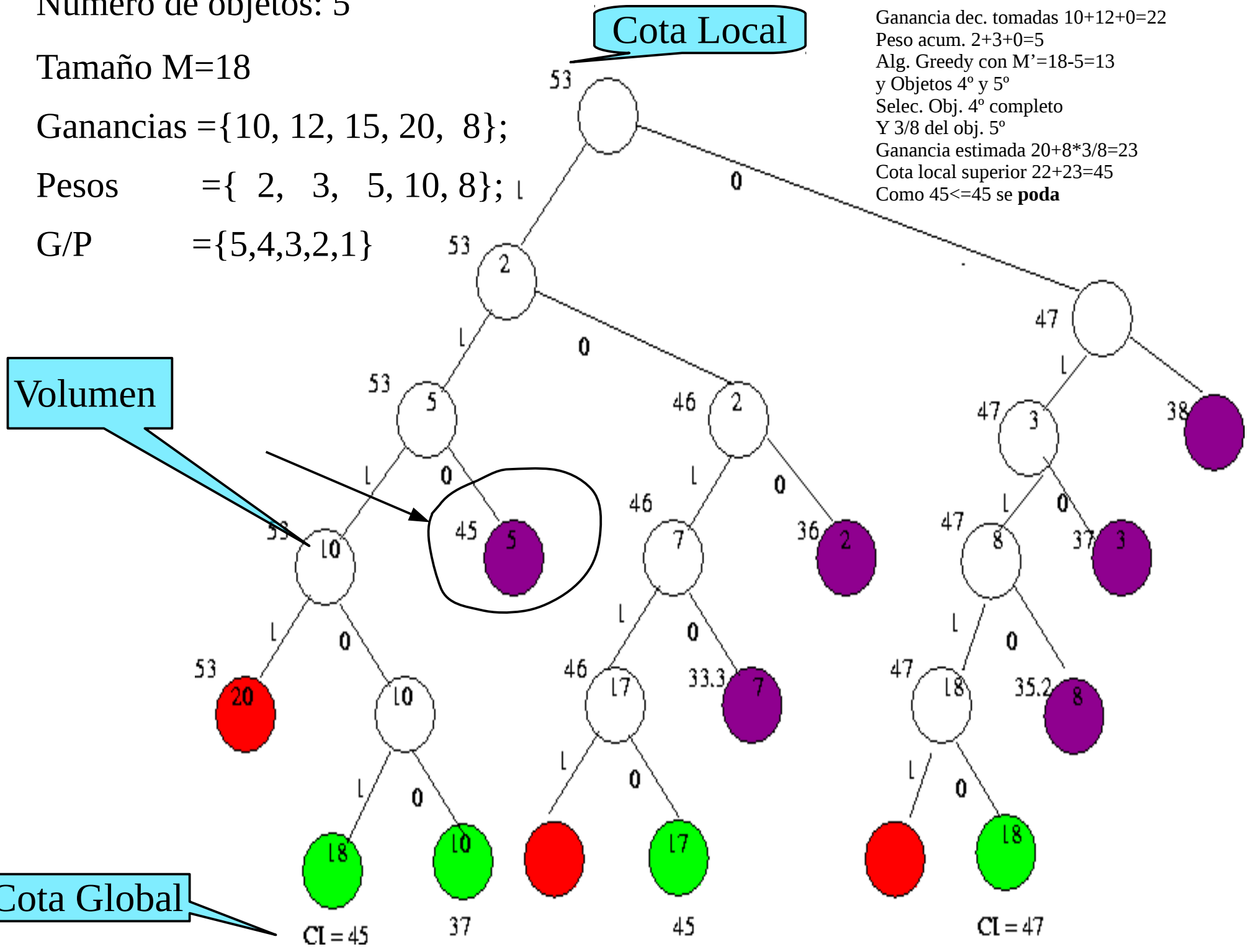
Número de objetos: 5

Tamaño M=18

Ganancias ={10, 12, 15, 20, 8};

Pesos ={ 2, 3, 5, 10, 8};

G/P ={5,4,3,2,1}



Optimizar en tiempo función factibilidad

- No es necesario volver a realizar todos los cálculos. Asumimos i el objeto que consideramos.

Podemos almacer inform. (objetos 0,..,i-1).

  - VolumenOcupado: VO, inicialmente VO=0
  - GanaciaAcumulada: GA, inicialmente GA = 0
  - Estimador: E inicialmente E = Gr NO 0/1

- Si X[i] = 1 => VO += p[i]; GA += b[i];
- Si X[i] = 0 => VO -= p[i]; GA -= b[i];
- E = GA + (M-VO)(b[i+1]/p[i+1])

Un estimador más optimista que antes: asume que puede rellenar lo que falta con una fracción del objeto más valioso que queda (el siguiente).

Todas las operaciones son O(1)!!!!

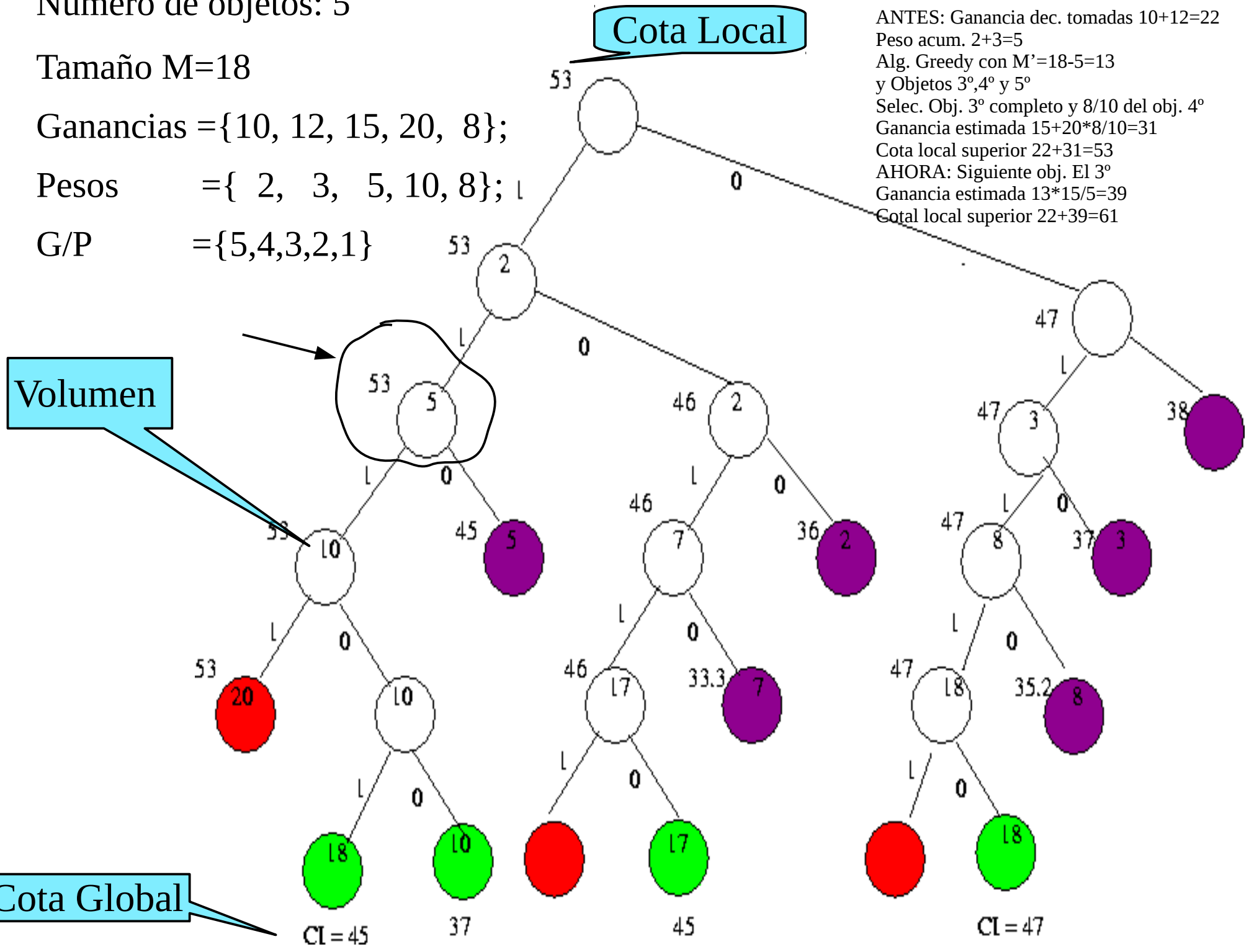
Número de objetos: 5

Tamaño M=18

Ganancias ={10, 12, 15, 20, 8};

Pesos ={ 2, 3, 5, 10, 8};

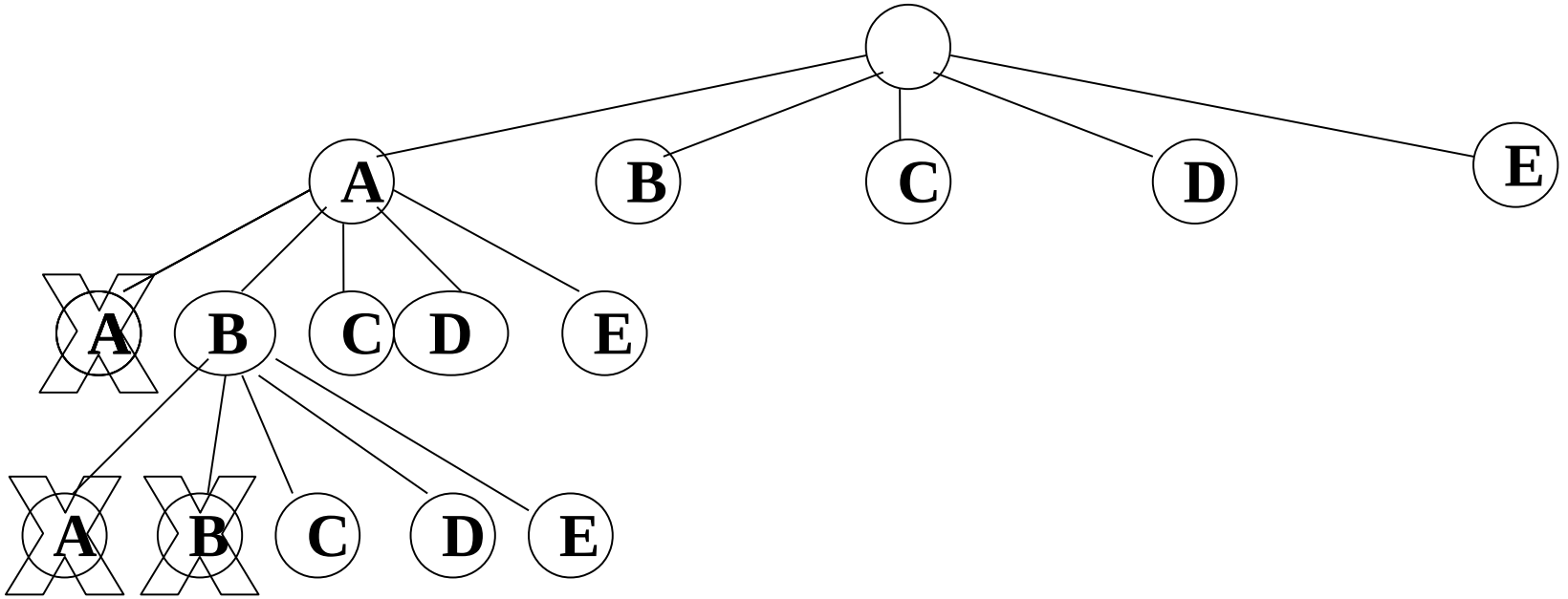
G/P ={5,4,3,2,1}



Problema Viajante de Comercio

- Supongamos N=5 ciudades.
  - Solución? X=[1,3,0,4,2]
  - Restricciones Explícitas: X[i] pertenece a {0,..,N-1}
  - Restricciones Implícitas: Una ciudad no puede aparecer dos veces en el recorrido.
  - Espacio de Soluciones: Permutaciones de N elementos

Viajante de comercio: Arbol de estados



Para evitar repetir varias veces las mismas soluciones, se fija una ciudad como punto de partida (la ciudad 0)
El circuito ABCDEA es lo mismo que BCDEAB o que CDEABC …
Tenemos pues un espacio de estados de (n-1)! en vez de n!

- Necesitamos definir cotas para este problema.
  - Es un problema de minimización, por tanto:
    - COTA GLOBAL es una cota superior (CS).

    Inicialmente, solución obtenida por Algoritmo Greedy

    Se actualiza cada vez que encontremos una solución de menor costo.

    - COTA LOCAL (estimador) es una cota inferior (CI(N)).
    - Esta cota indica que NO puede existir un ciclo con una distancia menor. Pero NO implica que necesariamente exista un ciclo con dicha distancia.

    Debemos considerar:

      - Costo del camino que representa la solución parcial
      - Un estimador (por debajo) del costo para ir al resto de ciudades

- Podamos si CI(N) >=CS.

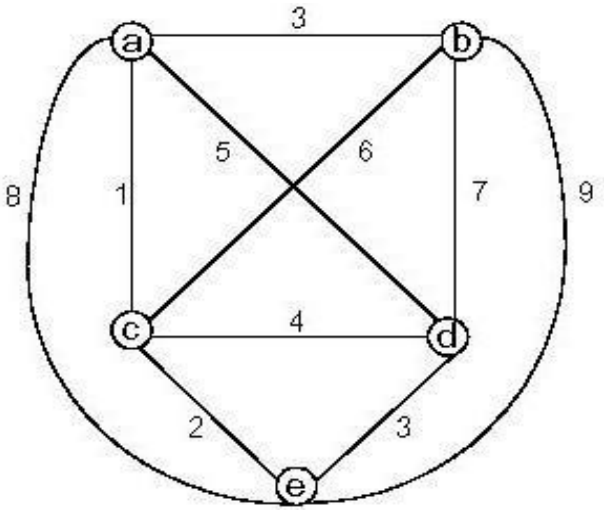
- COTA LOCAL (estimador) es una cota inferior (CI(N)).

X=[decisiones tomadas]+[decisiones por tomar]

X=[a,d,e,?,?]

  - Costo del camino que representa la solución parcial
  - Un estimador (por debajo) del costo para ir al resto de ciudades

Cómo?

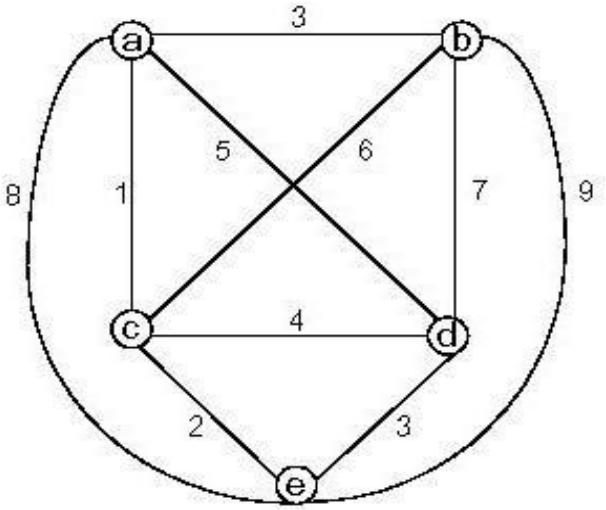


Viajante de comercio: Cota 1

E1: La selección directa sería encontrar el arco de menor peso en el grafo y multiplicar dicho peso por el número de ciudades que quedan por visitar.

Ejemplo: " a -- d – e .... X .... Y .... (a) "

Conocemos 2 arcos, pero nos falta por conocer 3 .



A – D		5
D – E		3
3 VECES	1x3	3

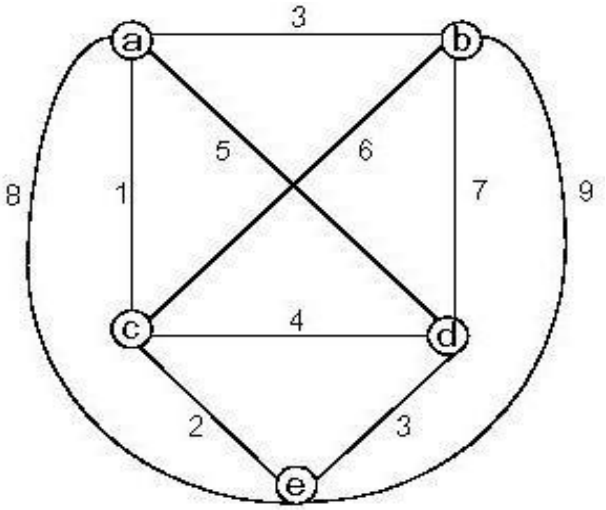
Estimador = 5 +3+3= 11

Viajante de comercio: Cota 2

- Ya que un ciclo debe pasar exactamente una vez por cada vértice, un estimador (cota inferior) de la longitud del ciclo se tiene al considerar el MINIMO costo de “SALIR” de cada vértice.
  - Es el minimo valor de todas las entradas no nulas de la matriz de costos (matriz de adyacencia)

    Ejemplo: “ a -- d – e .... X .... Y .... (a) “

    Conocemos 2 arcos, pero nos falta por conocer 3 .



A – D	5
D – E	3
Salir de E	2
Salir de B	3
Salir de C	1

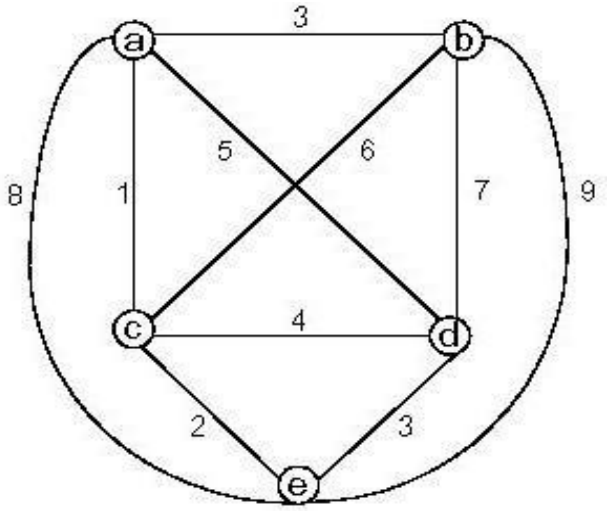
Estimador = 14

Viajante de comercio: Cota 3

- Ya que un ciclo debe pasar exactamente una vez, un estimador (cota inferior) de la longitud del ciclo se tiene al considerar el MINIMO costo de “SALIR Y ENTRAR” en cada vértice.
  - Para cada arista (u, v), podemos considerar la mitad de su peso como el costo de salir de u, y la otra mitad como el costo de entrar a v.
  - El costo total de un ciclo = suma de visitar (entrar y salir) un vértice cada vez
  - Cálculo:
    - Para cada vértice, considerar los dos arcos adyacentes con menor coste. Su suma, dividida por 2 es una cota inferior de pasar por cada vértice
-

Ejemplo: “ a -- d – e .... X .... Y .... (a) “

Conocemos 2 arcos, pero nos falta por conocer 3



.... A – D (1+5) / 2 = 3

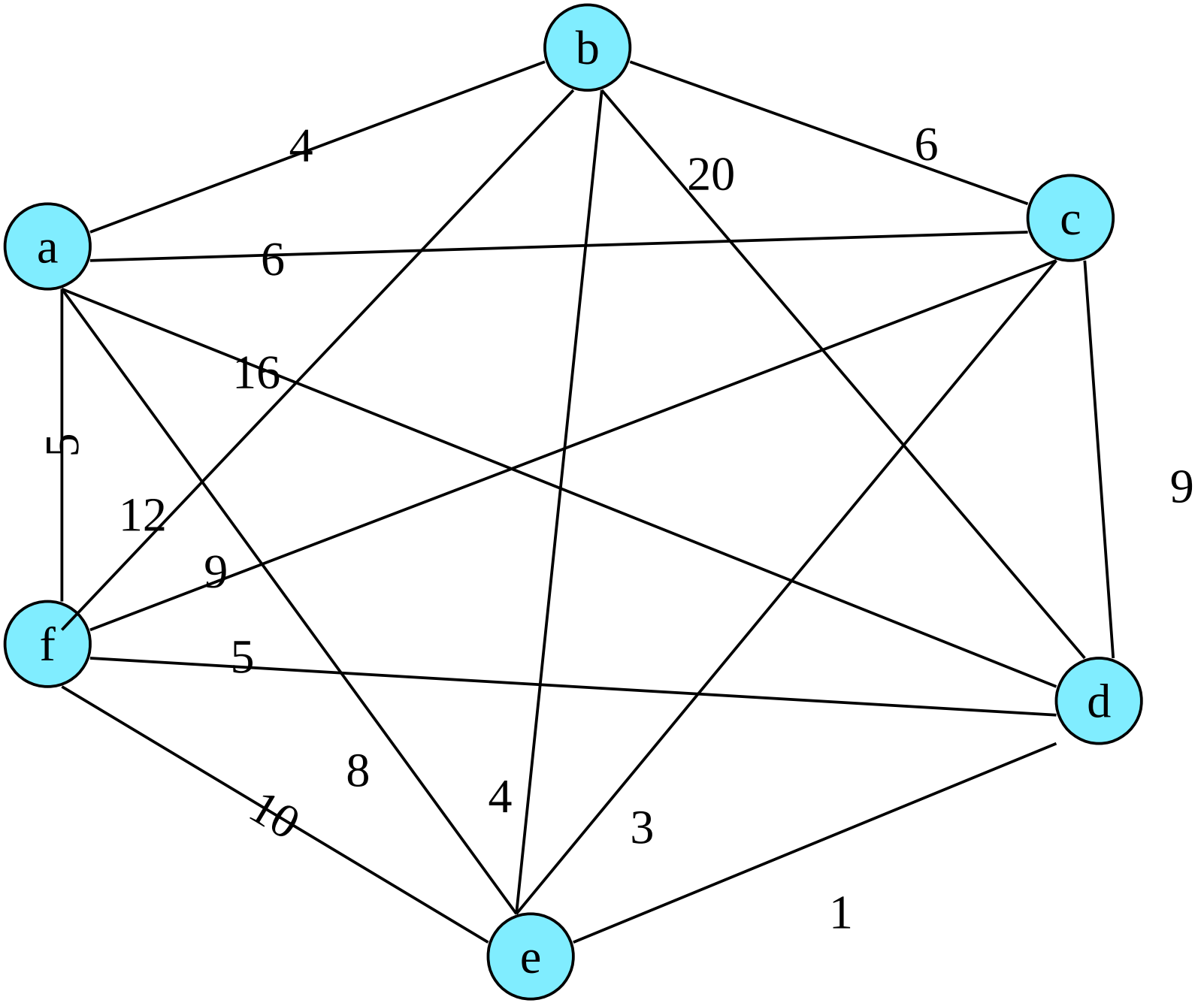
– D – E (5+3) / 2 = 4

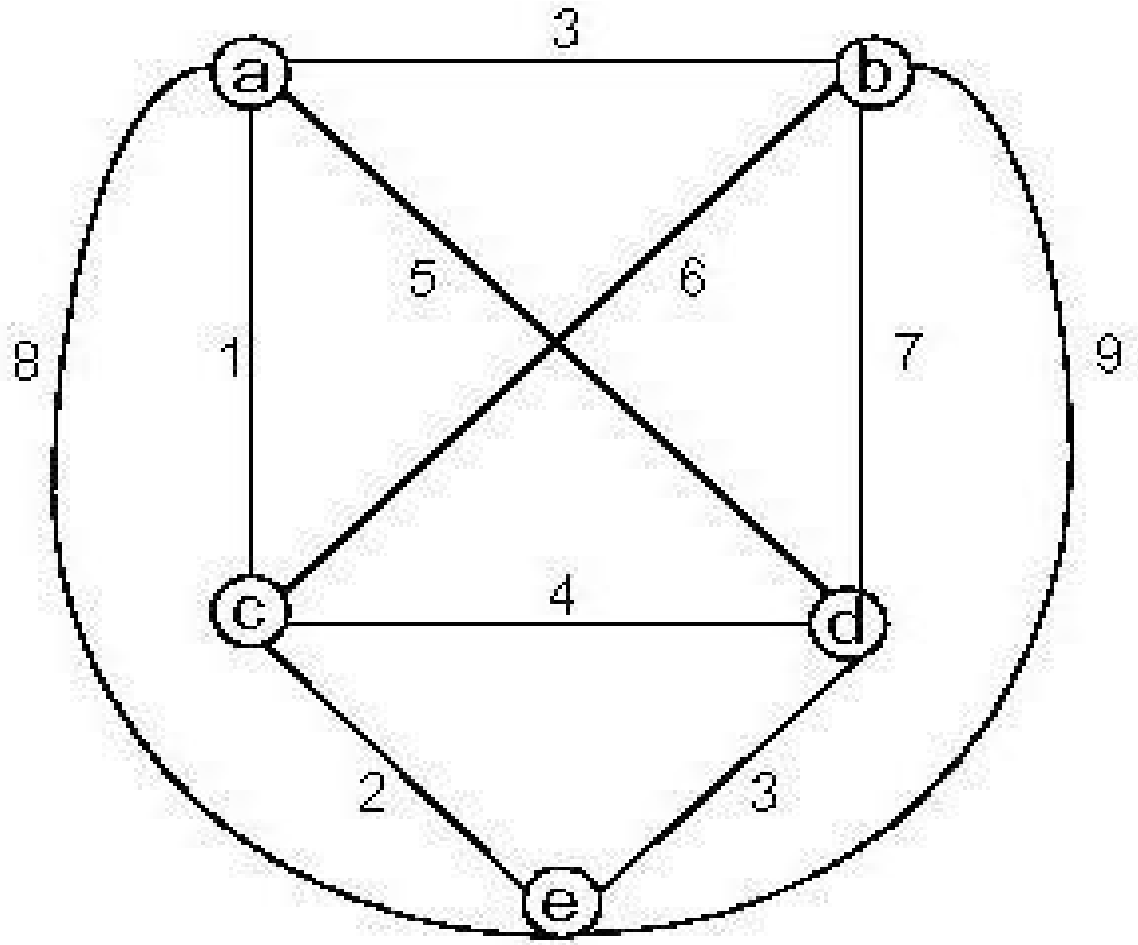
– E .... (3+2)/2 = 2.5

... B .... (3+6)/2 = 4.5

....C .... (1+2)/2 = 1.5

Estimador = 15.5


b
a
c
f
d
e
4
6
20
6
16
5
12
9
9
5
8
4
3
10
1


a
b
c
d
e
3
5
6
8
1
7
9
4
2
3

Eficiencia de Backtracking

- Depende de:
  - El tiempo necesario para generar la siguiente componente de la solución, X(k)
  - El número de X(k) que satisfacen las restricciones explícitas
  - El tiempo de determinar la función de factibilidad (acota las soluciones)
  - El número de X(k) que satisfacen la función de factibilidad (número de nodos generados)

Eficiencia de Backtracking

- Una función de factibilidad es buena sí:
  - permite reducir considerablemente (podar) el número de nodos generados.
  - El tiempo de ejecución es razonable
- Objetivo:
  - Reducir el tiempo global de ejecución
  - Hay que buscar un equilibrio entre esos factores: cuanto mejor (más nodos poda) la función también es mayor su tiempo de ejecución

Eficiencia de backtracking

- De los 4 factores que determinan el tiempo requerido por un algoritmo backtracking, solo el cuarto, el número de nodos generados, varía de un caso a otro.

- Un algoritmo backtracking en un caso podría generar solo O(n) nodos, mientras que en otro (relativamente parecido) podría generar casi todos los nodos del árbol de espacio de estados.

- Si el número de nodos en el espacio solución es d^n o n!, el tiempo del peor caso para el algoritmo backtracking sería generalmente $O(p(n)d^n)$ u O(q(n)n!) respectivamente, con p y q polinomios en n.

- La importancia del backtracking reside en su capacidad para resolver casos con grandes valores de n en muy poco tiempo.

- La dificultad está en predecir la conducta del algoritmo backtracking en el caso que deseemos resolver.

Branch and Bound

- Se recorre el árbol de estados de forma sistemática.
- Utiliza funciones de poda para eliminar ramas que no conducen a soluciones.
- En B&B se generan todos los hijos del e-nodo antes de que cualquier otro nodo vivo pase a ser el nuevo e-nodo.
- En backtracking tan pronto como se genera un nuevo hijo del e-nodo, este hijo pasa a ser el e-nodo.

Branch and Bound

- En backtracking los únicos nodos vivos son los que están en el camino de la raíz al e-nodo.
- En B&B puede haber más nodos vivos: se tienen que almacenar en alguna estructura de datos, llamada lista o contenedor de nodos vivos.
- En B&B el uso de cotas nos sirve, además de para podar, para determinar el orden de ramificación, seleccionando el nodo más prometedor como siguiente nodo a expandir.
- B&B se usa principalmente en problemas de optimización

Branch and Bound

- Backtracking es más fácil de implementar que B&B.
- Backtracking necesita muchos menos requisitos de memoria que B&B.
- B&B suele revisar menos estados para llegar a la solución (búsqueda más guiada).
- B&B puede ser (pero no necesariamente) más eficiente (en tiempo) que backtracking.

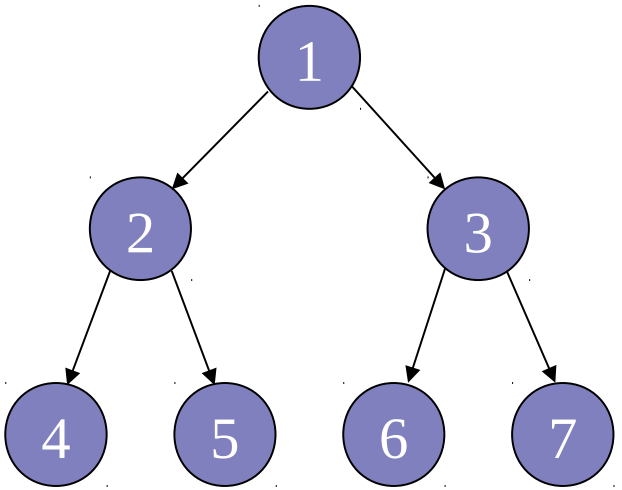
Esquema B&B

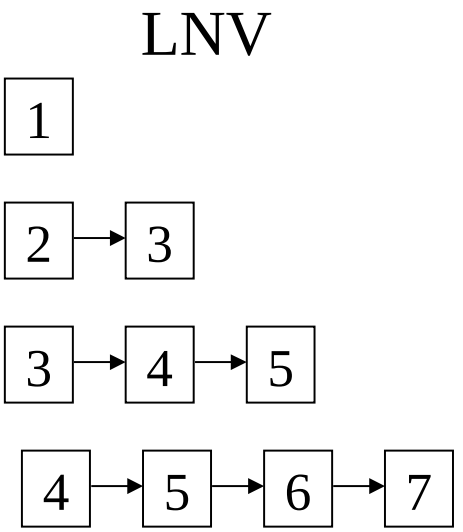
```
Algoritmo_B&B( )
{  contenedor<solucion> C;   solucion sol;
C.inserta(sol); // Raiz del arbol de estados
do {   sol=C.Selecciona_Siguiente_Nodo();
        if (sol.Factible()) {
            k = sol.Comp(); // Componente del e nodo;
            k++; // Siguiente componente
            for (sol.PrimerValorComp(k); sol.HayMasValores(k); sol.SigValComp(k))
                if (sol.Factible())  // Incluye las cotas
                    if (sol.NumComponentes()==sol.size())  //Es solucion?
                        sol.ActualizaSolucion();
                    else C.insert(sol);
        }
} while (!C.empty() )
}
```

¿Qué contenedor tenemos que utilizar?

Qué contenedor tenemos que utilizar?

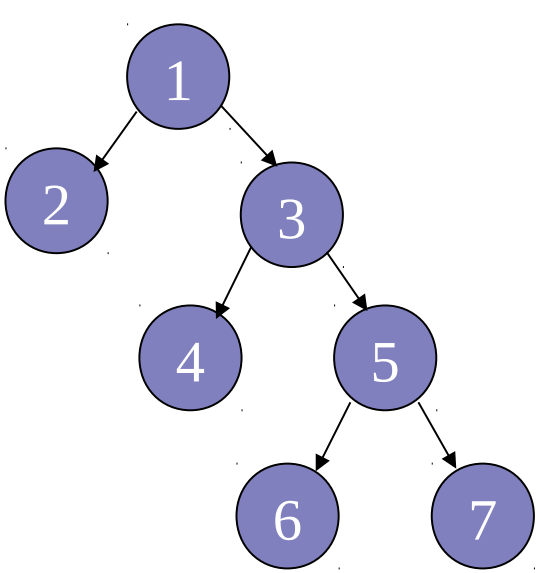
- El tipo de contenedor determina la estrategia o criterio de ramificación:
  - Cola para criterio FIFO
    - queue<solucion> C

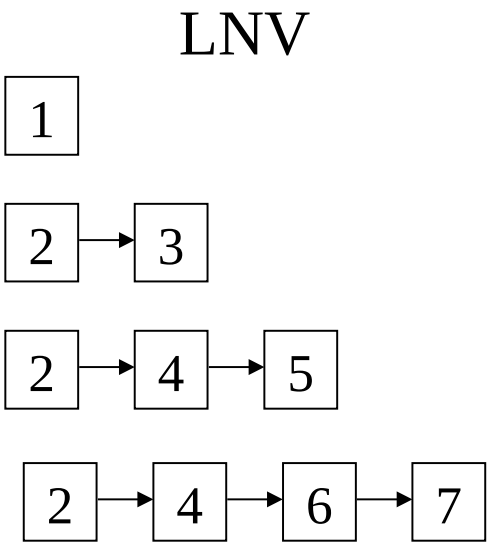




Qué contenedor tenemos que utilizar?

- El tipo de contenedor determina la estrategia o criterio de ramificación:
  - Pila para criterio LIFO
    - stack<solucion> C





Qué contenedor tenemos que utilizar?

- Las estrategias FIFO y LIFO realizan una búsqueda sistemática “a ciegas”.
- Si dispusiéramos de una estimación del beneficio o coste óptimo que se puede conseguir a partir de un nodo,
- sería mejor buscar primero por los nodos con mejor valor estimado (mayor beneficio o menor costo)

Qué contenedor tenemos que utilizar?

- El tipo de contenedor determina la estrategia o criterio de ramificación:
  - Cola con prioridad para criterio LC (Least Cost)
    - priority_queue<solucion, comparacion<solucion> > C

  donde comparacion<solucion> es la función que nos permite ordenar los elementos en el Heap, depende de si el problema es maximizar o minimizar un determinado objetivo

Estrategia LC

- En problemas de optimización, la estimación del beneficio o coste (lo prometedor que es un nodo) suele coincidir con la cota (superior o inferior, respect.) local.
- En problemas de otro tipo (p.e. juegos) se suele estimar el coste de hallar la solución a partir de cada nodo.

B&B simplificado

- Vamos a utilizar una versión simplificada de B&B en la que para cada nodo solo se usa una cota local (superior para problemas de maximización e inferior para problemas de minimización).
- El valor de esa cota local es el que se utiliza, además de para podar (comparándolo con la cota global), para decidir cuál es el mejor nodo para expandir.
- Interesa (más que en backtracking) que la cota sea precisa (no demasiado optimista).

B&B general

- En su versión general, B&B emplea 2 cotas locales (inferior y superior) por nodo.
- En un problema de maximización, si la cota superior de un nodo es menor o igual que la cota inferior de otro (o menor o igual que la cota global), se puede podar.
- En un problema de minimización, si la cota inferior de un nodo es mayor o igual que la cota superior de otro (o mayor o igual que la cota global), se puede podar.
- El estimador para decidir el orden expansión de los nodos se puede calcular a partir de esas dos cotas o es un valor intermedio.

Esquema B&B LC

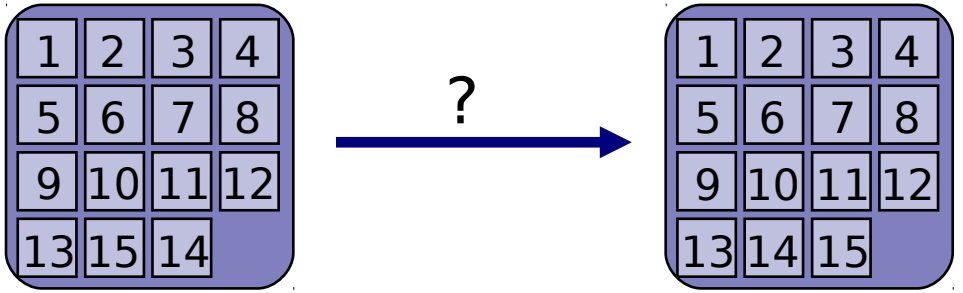
```
Algoritmo_B&B( )
{  priority_queue<solucion> apo;   solucion sol; fin=false;
apo.insert(sol); // Raiz del arbol de estados
do {   sol=apo.top();
        if (sol.Factible()) { // incluye cotas
            k = sol.Comp(); // Componente del e nodo;
            k++; // Siguiente componente
            for (sol.PrimerValorComp(k); sol.HayMasValores(k); sol.SigValComp(k))
                if (sol.Factible())  // Incluye las cotas
                    if (sol.NumComponentes()==sol.size())  //Es solucion?
                        sol.ActualizaSolucion();
                    else apo.insert(sol);
        } else fin=true;
} while ( (!apo.empty()) && (!fin))
}
```

Ejemplo: el juego del 15

Samuel Loyd: El juego del 15 o “taken”.

Problema publicado en un periódico de Nueva York en 1878 y por cuya solución se ofrecieron 1000 dólares.

- El problema original:

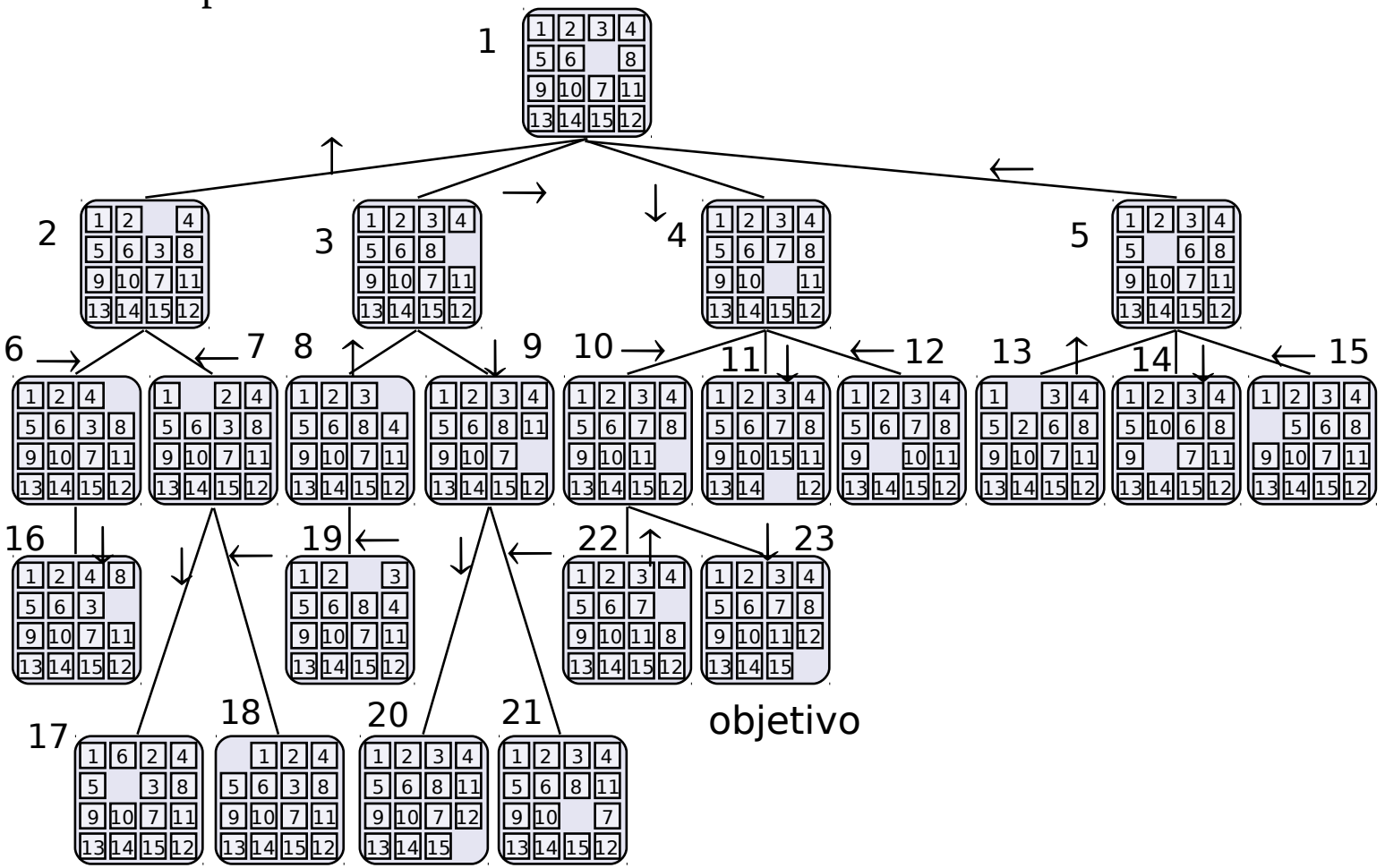


Problema de Lloyd El objetivo

- Decisión: encontrar una secuencia de movimientos que lleve al objetivo
- Optimización: encontrar la secuencia de movimientos más corta

Ejemplo: el juego del 15

- Configuración: permutación de (1,2,...,16)
- Solución: secuencia de configuraciones que
  Empiezan en el estado inicial y acaban en el final.
  De cada configuración se puede pasar a la siguiente.
  No hay configuraciones repetidas



Ejemplo: el juego del 15

- **Problema muy difícil para backtracking**
  - **El árbol de búsqueda es potencialmente muy profundo (16! niveles), aunque puede haber soluciones muy cerca de la raíz.**

- **Se puede resolver con branch and bound (aunque hay métodos mejores): Algoritmo A***

- **Estimación del coste:**
  - número de fichas mal colocadas (puede engañar)
  - suma, para cada ficha, de la distancia a la posición donde le tocaría estar

Ejemplo Mochila 0/1

- Supongamos $n = 4$, $M = 16$, y los siguientes objetos:

i	b_i	p_i	b_i / p_i
1	\$40	2	\$20
2	\$30	5	\$6
3	\$50	10	\$5
4	\$10	5	\$2

- Para el cálculo de las cotas locales utilizamos el algoritmo greedy NO-0/1 (fraccional)

ganancia
peso
estimador

Ejemplo

X – No Factible
O – No optimal

M=16

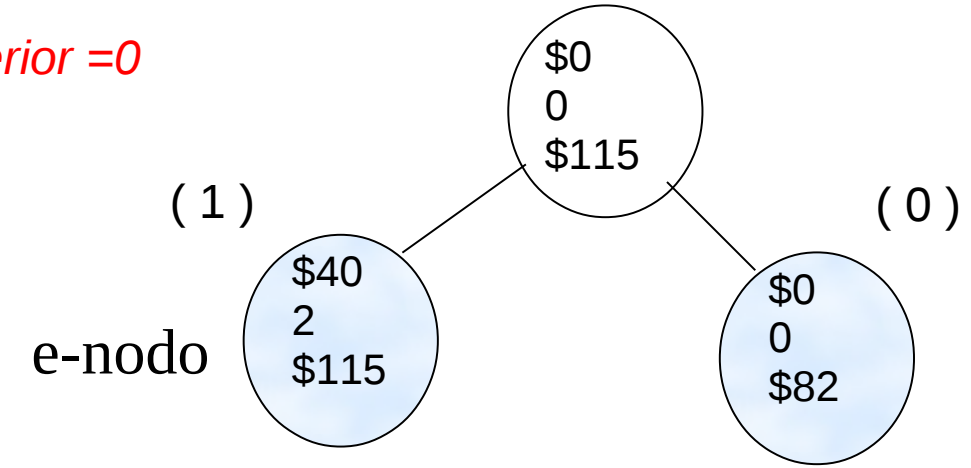
Objeto 1 [$40, 2]

Objeto 2 [$30, 5]

Objeto 3 [$50, 10]

Objeto 4 [$10, 5]

Cota Inferior =0



ganancia
peso
estimador

Ejemplo

X – No Factible
O – No optimal

M=16

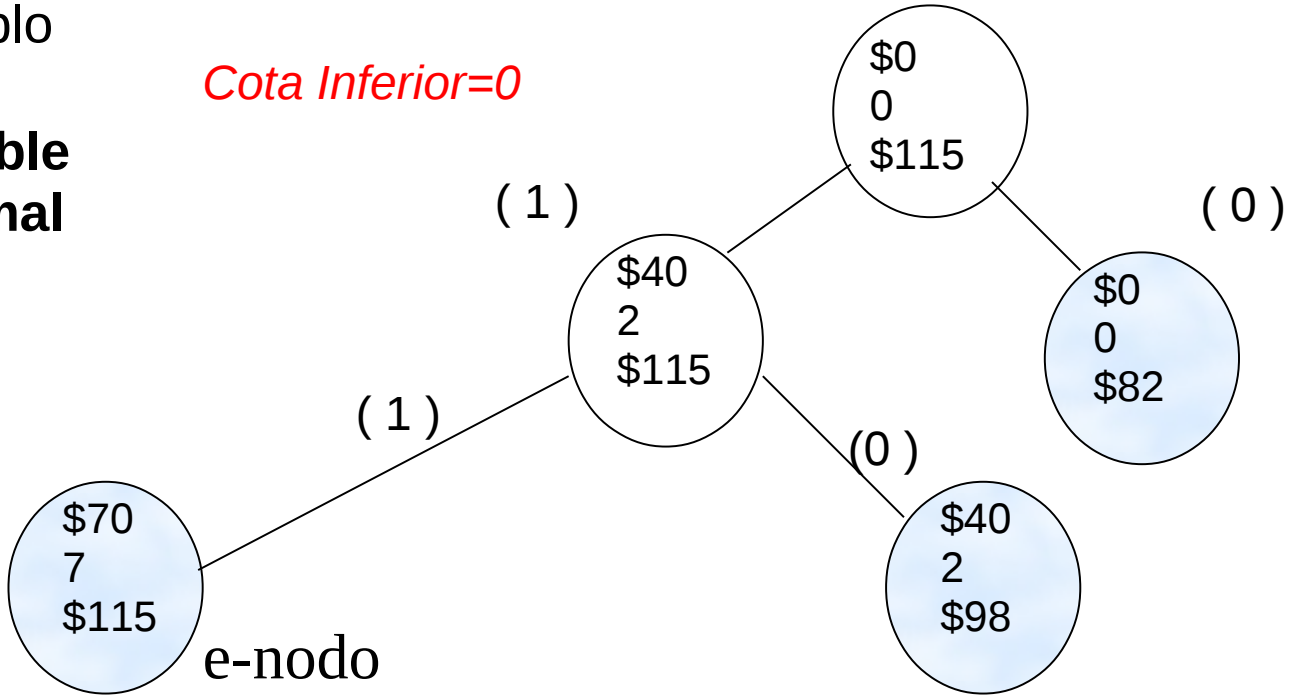
Objeto 1 [$40, 2]

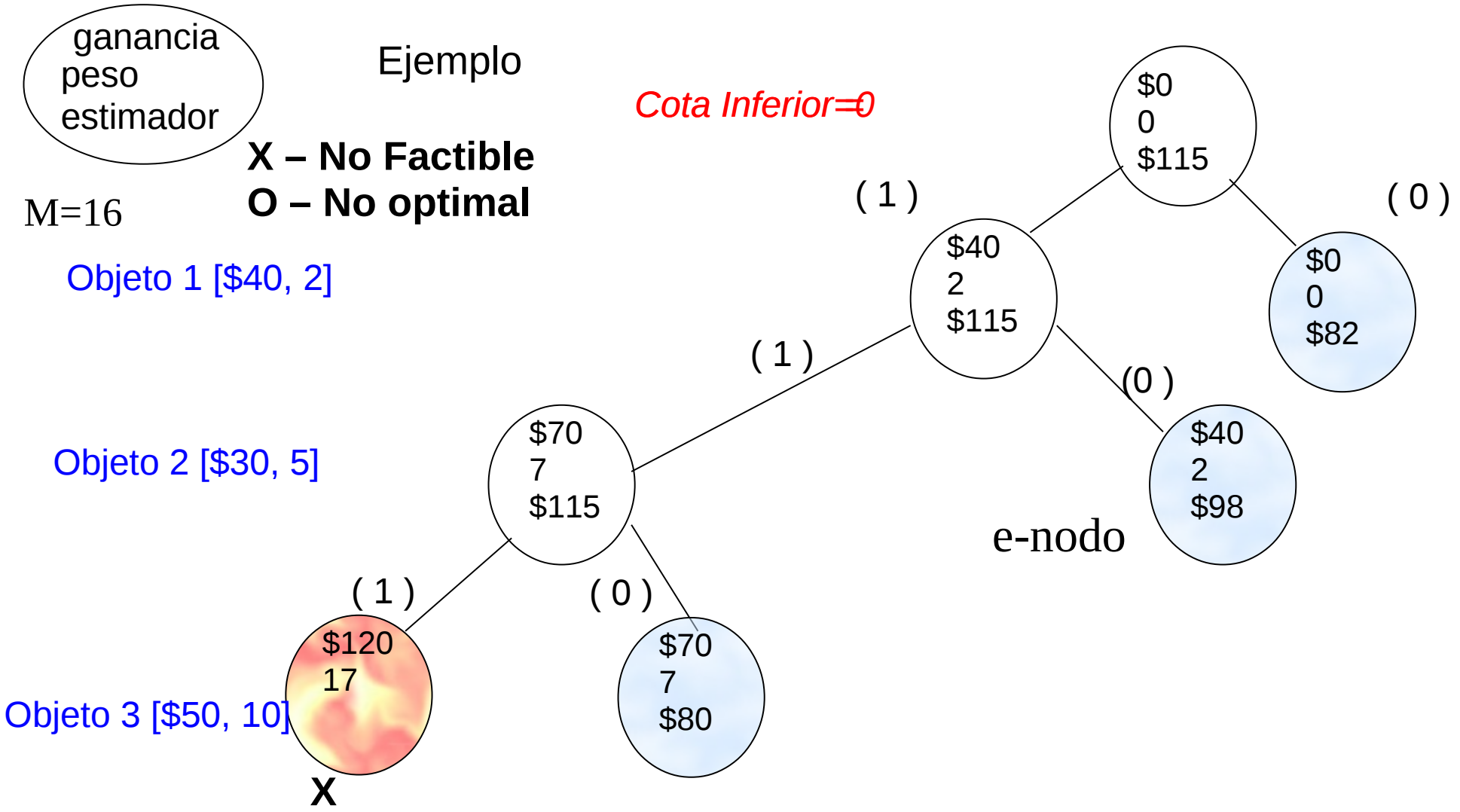
Objeto 2 [$30, 5]

Objeto 3 [$50, 10]

Objeto 4 [$10, 5]

Cota Inferior=0




ganancia
peso
estimador
Ejemplo
Cota Inferior=0
X – No Factible
O – No optimal
M=16
Objeto 1 [$40, 2]
Objeto 2 [$30, 5]
Objeto 3 [$50, 10]
Objeto 4 [$10, 5]
$0
0
$115
(1)
(0)
$40
2
$115
$0
0
$82
(1)
(0)
$70
7
$115
$40
2
$98
e-nodo
(1)
(0)
$120
17
$70
7
$80
X

ganancia
peso
estimador

Ejemplo

X – No Factible
O – No optimal

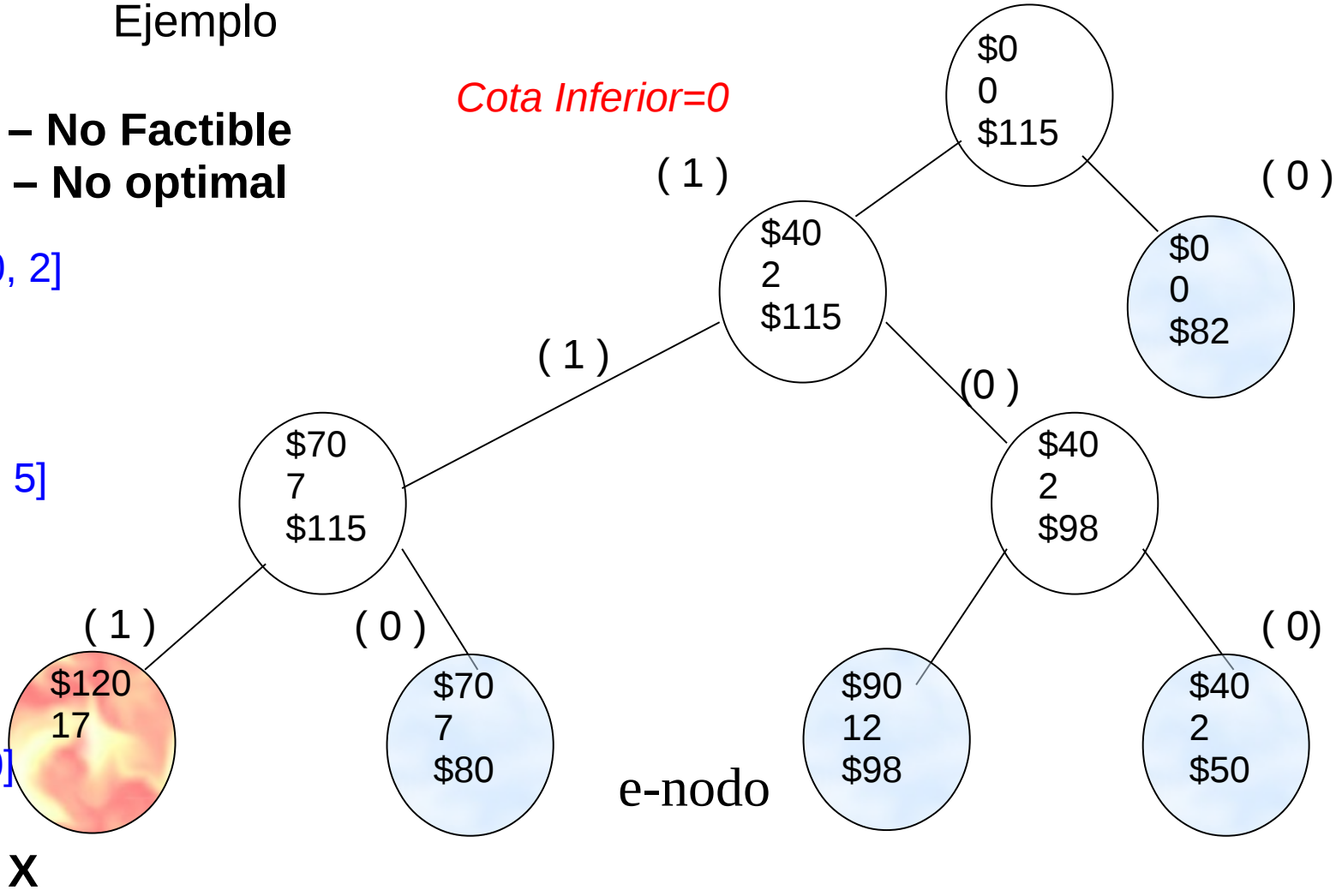
M=16

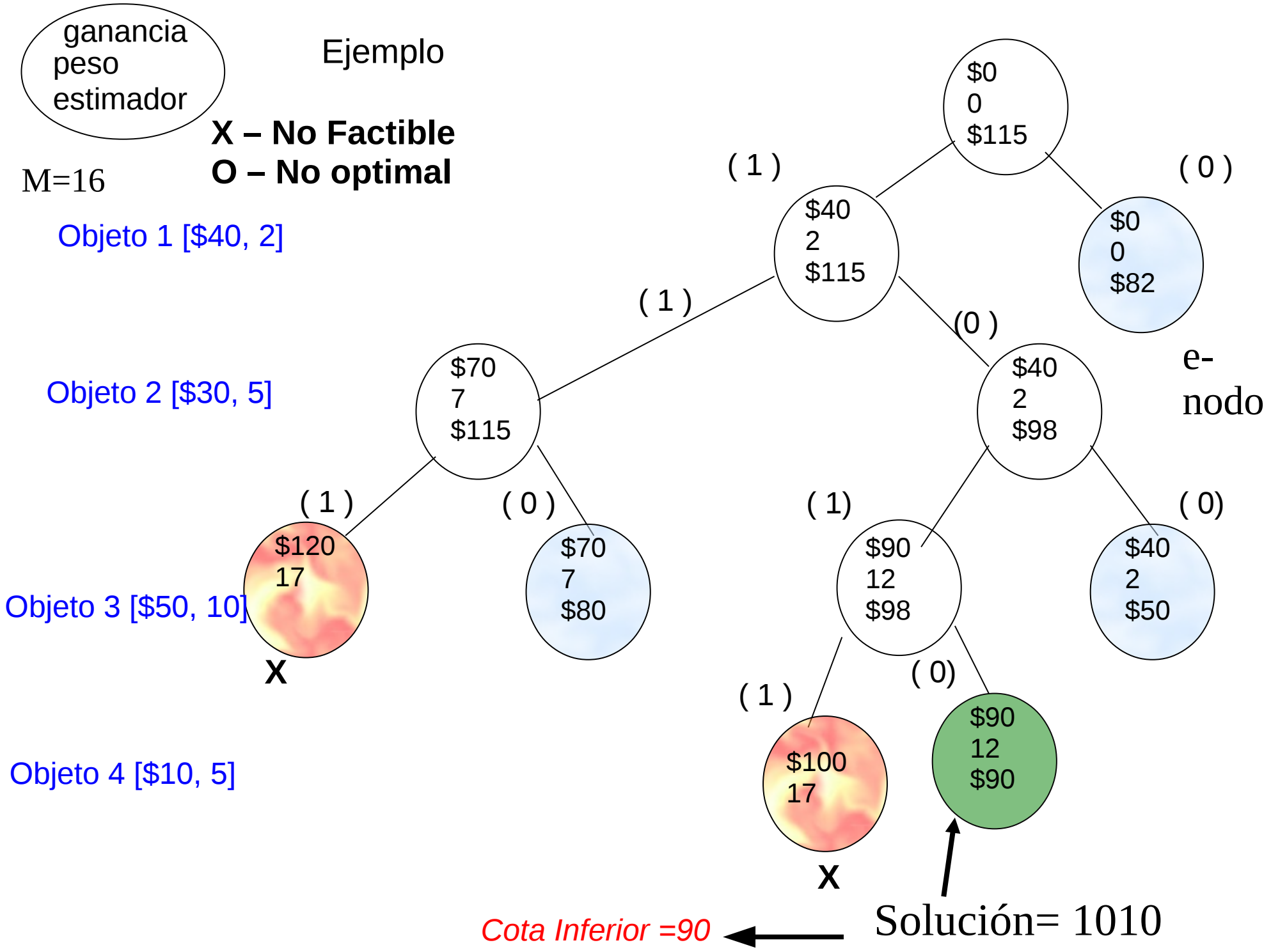
Objeto 1 [$40, 2]

Objeto 2 [$30, 5]

Objeto 3 [$50, 10]

Objeto 4 [$10, 5]




ganancia
peso
estimador
Ejemplo
X – No Factible
O – No optimal
M=16
Objeto 1 [$40, 2]
Objeto 2 [$30, 5]
Objeto 3 [$50, 10]
Objeto 4 [$10, 5]
$0
0
$115
(1)
(0)
$40
2
$115
$0
0
$82
(1)
(0)
$70
7
$115
$40
2
$98
e-
nodo
(1)
(0)
(1)
(0)
$120
17
$70
7
$80
$90
12
$98
$40
2
$50
X
(1)
(0)
$100
17
$90
12
$90
X
Cota Inferior =90
Solución= 1010

```
class Solucion {
public:
  Solucion(const int tam_max);
  bool Factible( ) const;
  void PrimerValorComp(int k);
  void SigValComp(int k);
  TipoBase Comp(int k) const;
  float CotaLocal() const
  bool HayMasValores(int k) const
  float Evalua();
  int NumComponentes() const;
  bool EsSolucion() const
  int CompActual() const
  bool operator<( const Solucion & s) const
   {return Estimador < s.Estimador;}

private:
  vector<int> X;   // Almacenamos la solucion
  int pos_e;     // Posicion  de la ultima decision en X
  float GA; // Ganancia Acumulada
  float VO; // Volumen ocupado
  float Estimador;  // Valor del estimador para el nodo X

};
```

```
Solucion Branch_and_Bound(int n_objetos )
{
  priority_queue<Solucion> Q;
  Solucion n_e(n_objetos), mejor_solucion(n_objetos) ;  //nodoe en expansion
  int k;
  float CG=0;   // Cota Global
  float ganancia_actual;

 Q.push(n_e);
 while ( !Q.empty() && (Q.top().CotaLocal() > CG) ){
    n_e = Q.top();
   Q.pop();
    k = n_e.CompActual();
    for ( n_e.PrimerValorComp(k+1); n_e.HayMasValores(k+1);n_e.SigValComp(k+1)) {
            if ( n_e.EsSolucion() ){
              ganancia_actual = n_e.Evalua();
              if (ganancia_actual > CG) { CG = ganancia_actual; mejor_solucion = n_e; }
          }  else if ( n_e.Factible( ) && n_e.CotaLocal()>CG )
                  Q.push( n_e );
     }
  }
 return mejor_solucion;
}
```

ganancia
peso
estimador

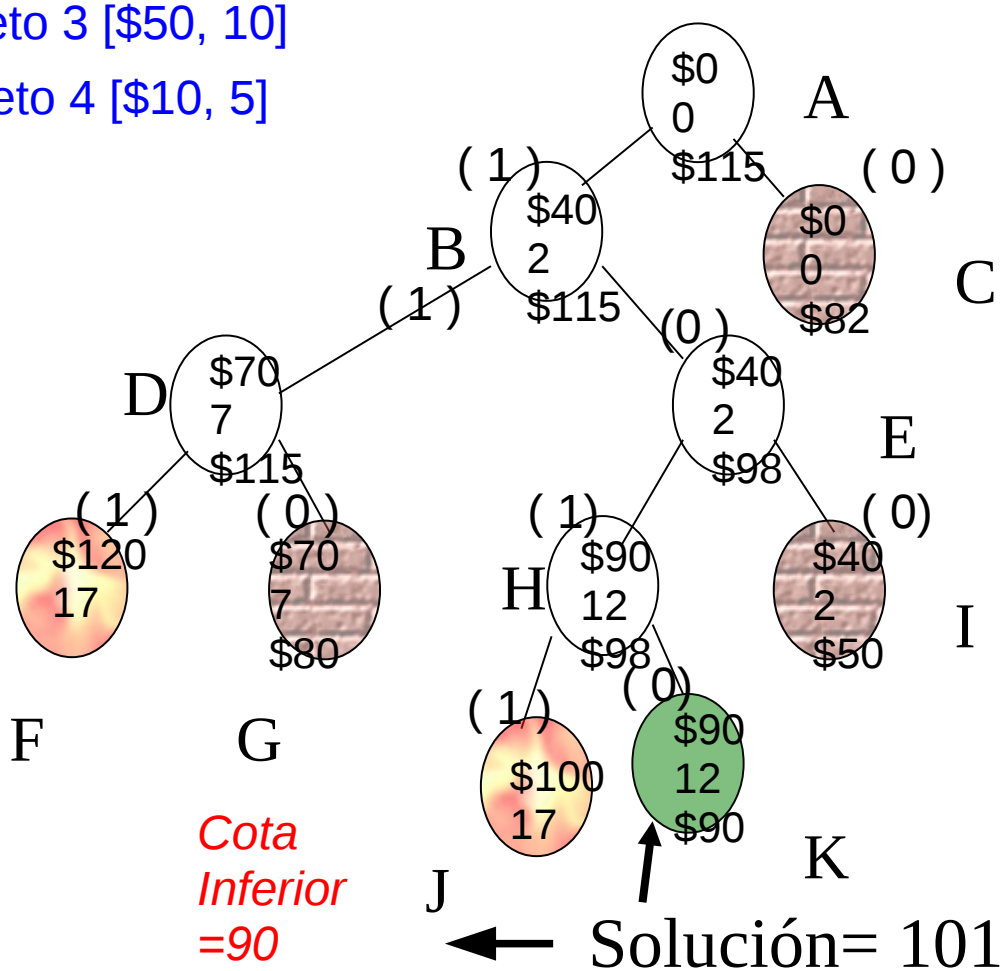
Ejemplo

Objeto 1 [$40, 2]

Objeto 2 [$30, 5]

Objeto 3 [$50, 10]

Objeto 4 [$10, 5]



Cola con Prioridad C

Estado		Cota Local	
A	X X X	115	
B	1 X X	115	
C	0 X X	82	
D	1 1 X	115	
E	1 0 X	98	
C	0 X X	82	
E	1 0 X	98	
C	0 X X	82	
G	1 1 0	80	
H	1 0 1	98	
C	0 X X	82	
G	1 1 0	80	
I	1 0 0	50	
K	1 0 1 0	90	Soluc
C	0 X X	82	
G	1 1 0	80	
I	1 0 0	50	

Problema de asignación

Dadas n personas y n trabajos, con C[i][j] el costo que tiene que la persona i-ésima realice el trabajo j-ésimo.

Ejemplo

	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4

Se pide:
Realizar una asignación de tareas de forma que:

- Todos los trabajos sean ejecutados
- Todas las personas realicen un trabajo (no hay persona ociosa)
- El costo de la asignación es mínimo

Problema de asignación

Espacio de estados:

hay que tomar n decisiones, cada decisión d_i se corresponde con el trabajo que se le asigna a la persona i-ésima.

Restricciones explícitas:

x[i] toma valores en {1..n}

Restricciones implícitas:

Un trabajo se le asigna a una única persona:

x[i] != x[j], para todo i!=j

Espacio de estados:

Árbol de permutaciones

Problema de asignación: Cotas

Cota Global:

Es un problema de minimización, por tanto actúa como cota superior

Se puede iniciar con +infinito o con un alg. Greedy: buscar el mínimo en la matriz, tachar la fila y columna correspondiente y repetir el proceso de buscar el mínimo en la matriz reducida.

Ejemplo

	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	**1**	8
Persona *d*	7	6	9	4

Cota = 1+2+4+6 = 13

Problema de asignación: Cotas

Cotas Locales

Para cada nodo, una cota inferior del coste de la mejor solución que se puede alcanzar desde el nodo

Para el nodo raíz: Cualquier solución debe ser mayor que:

Mínimos filas 2 + 3 + 1 + 4

(o mínimos columnas 5 + 2 + 1 + 4)

Ejemplo

	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4

Problema de asignación: Cotas

Cotas Locales

<u>Para un nodo con algunas asignaciones</u>: p.e. a<--Tr 3:

7 + Mínimos filas no asignadas 4 + 5 + 4

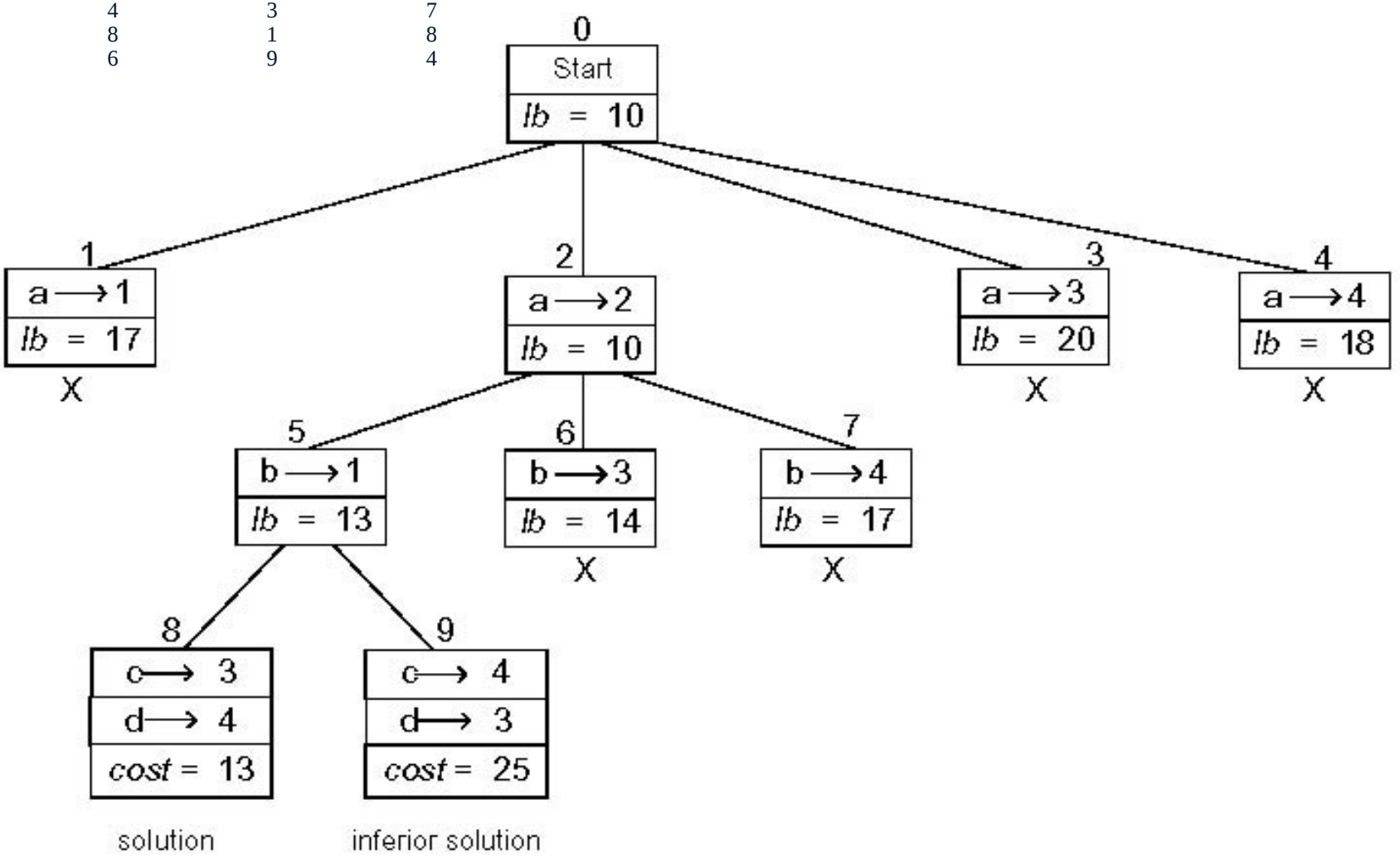
Ejemplo

	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4

	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	~~9~~	~~2~~	~~7~~	~~8~~
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4

Árbol de estados

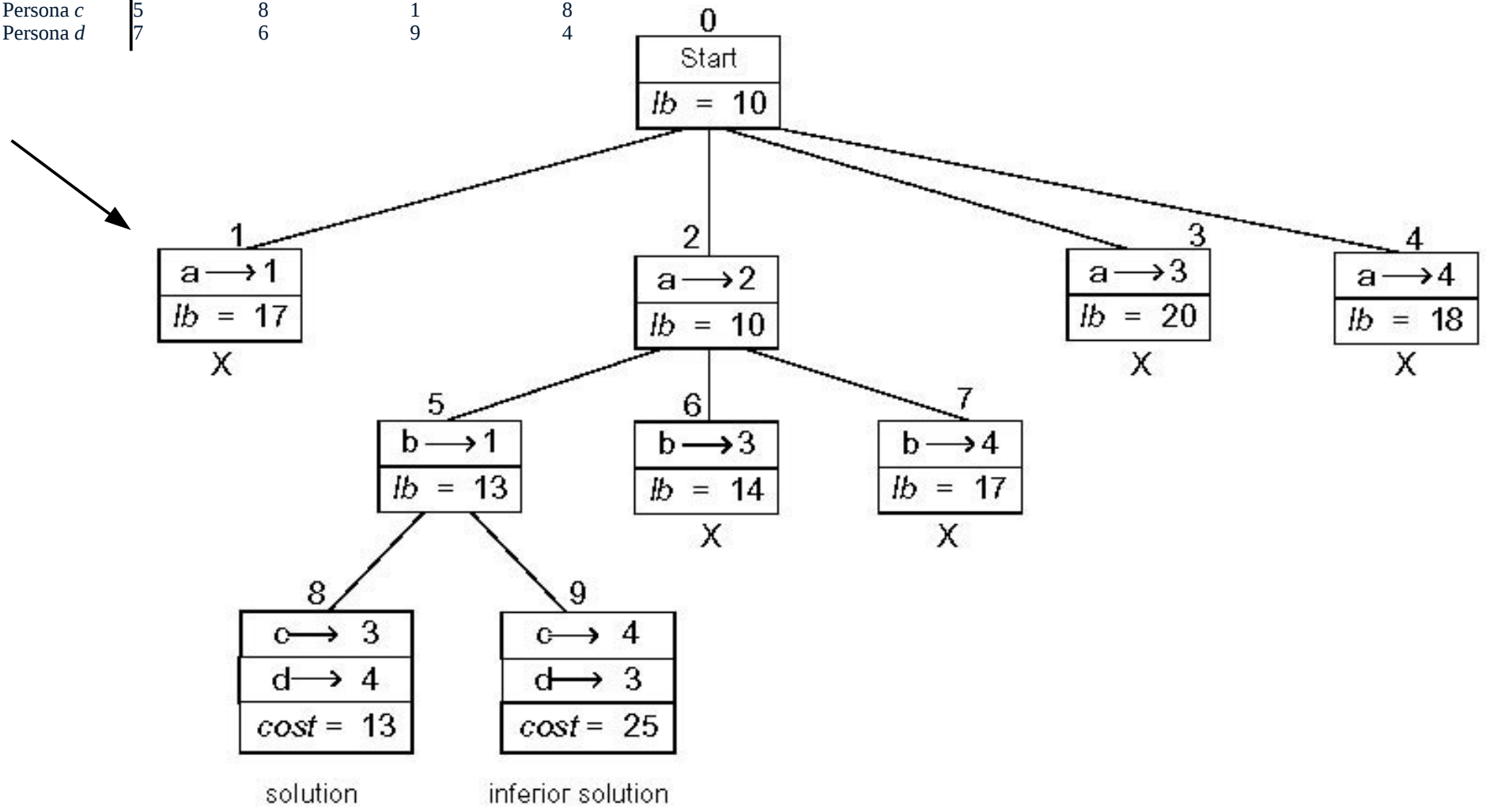
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

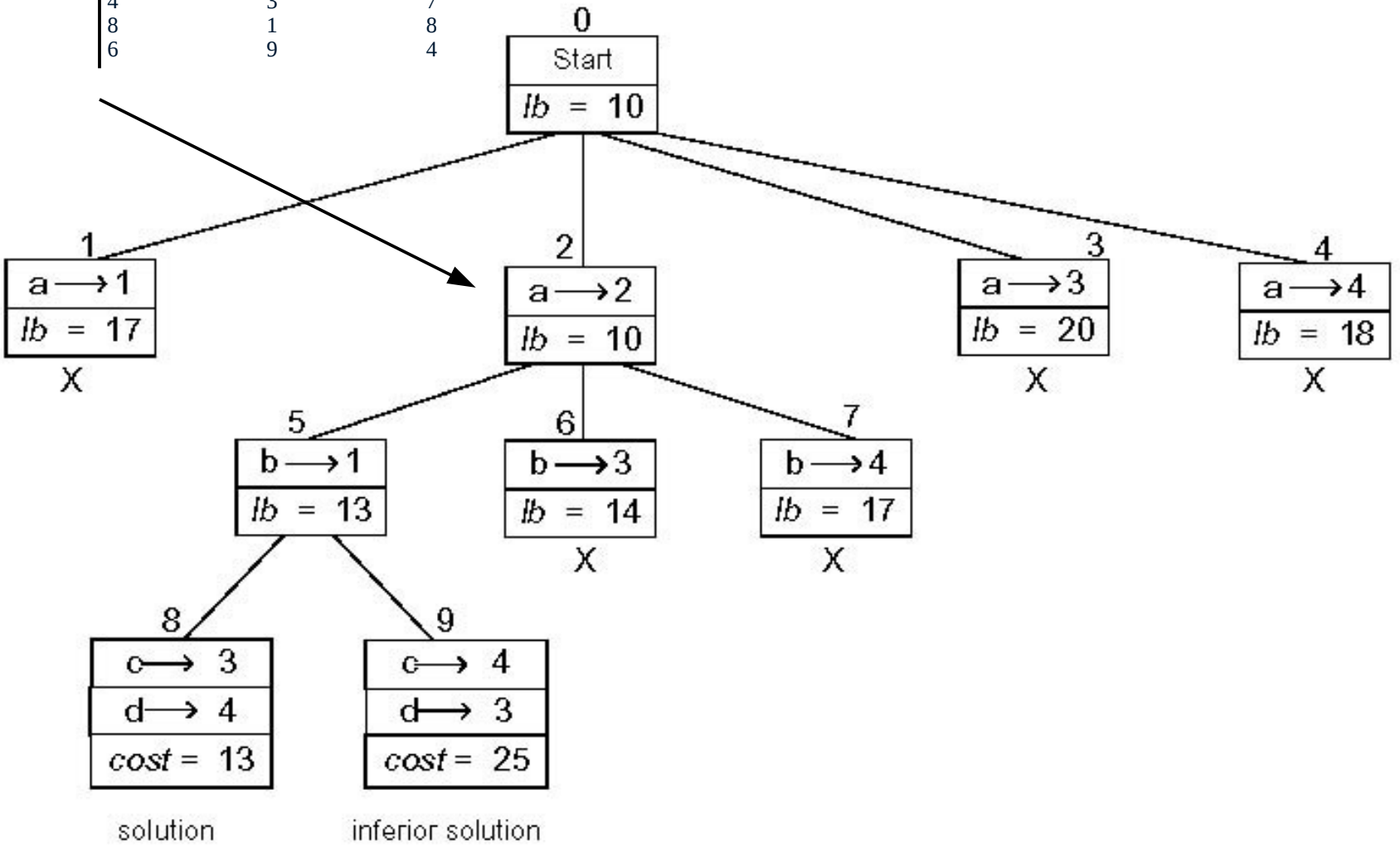
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

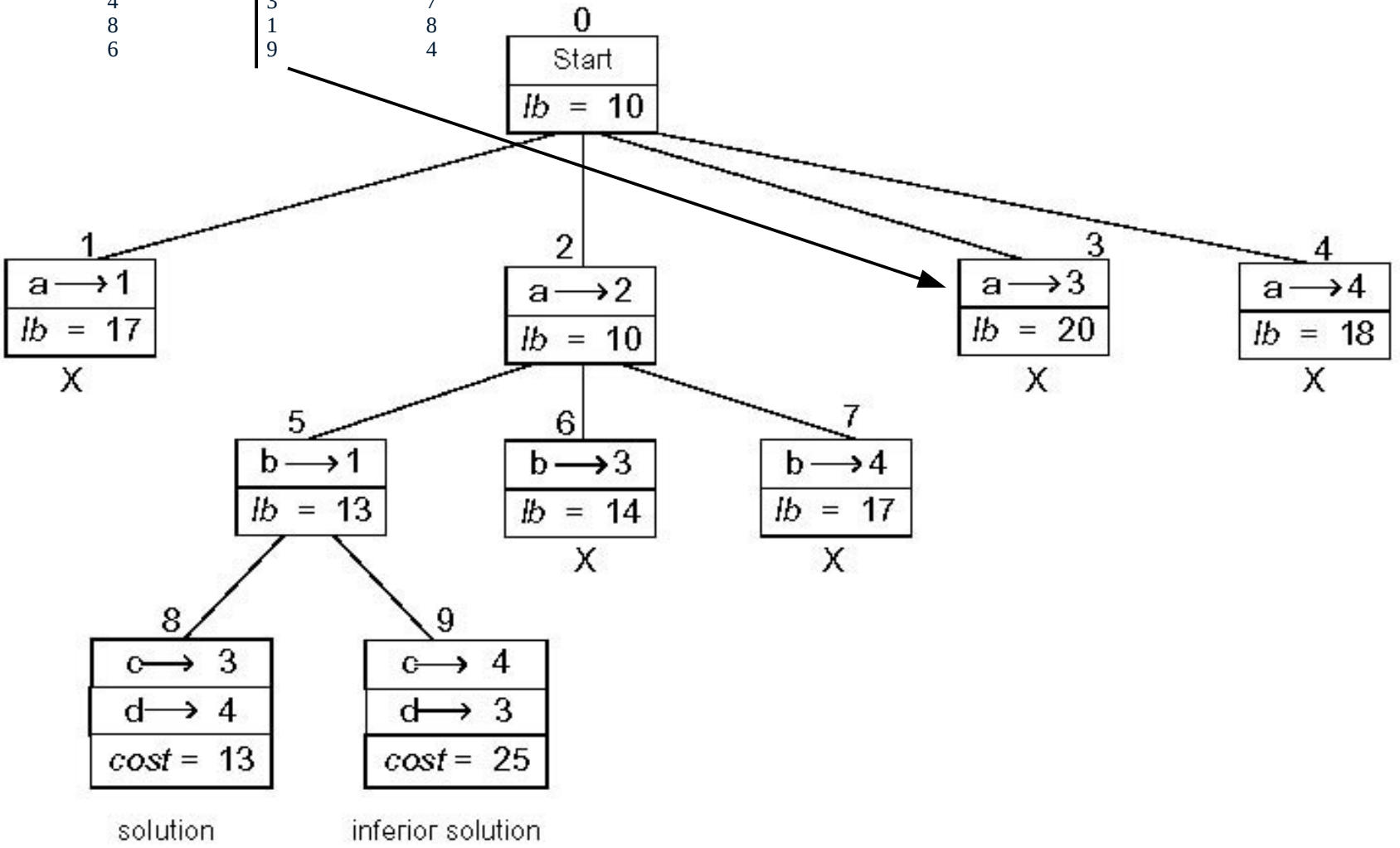
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

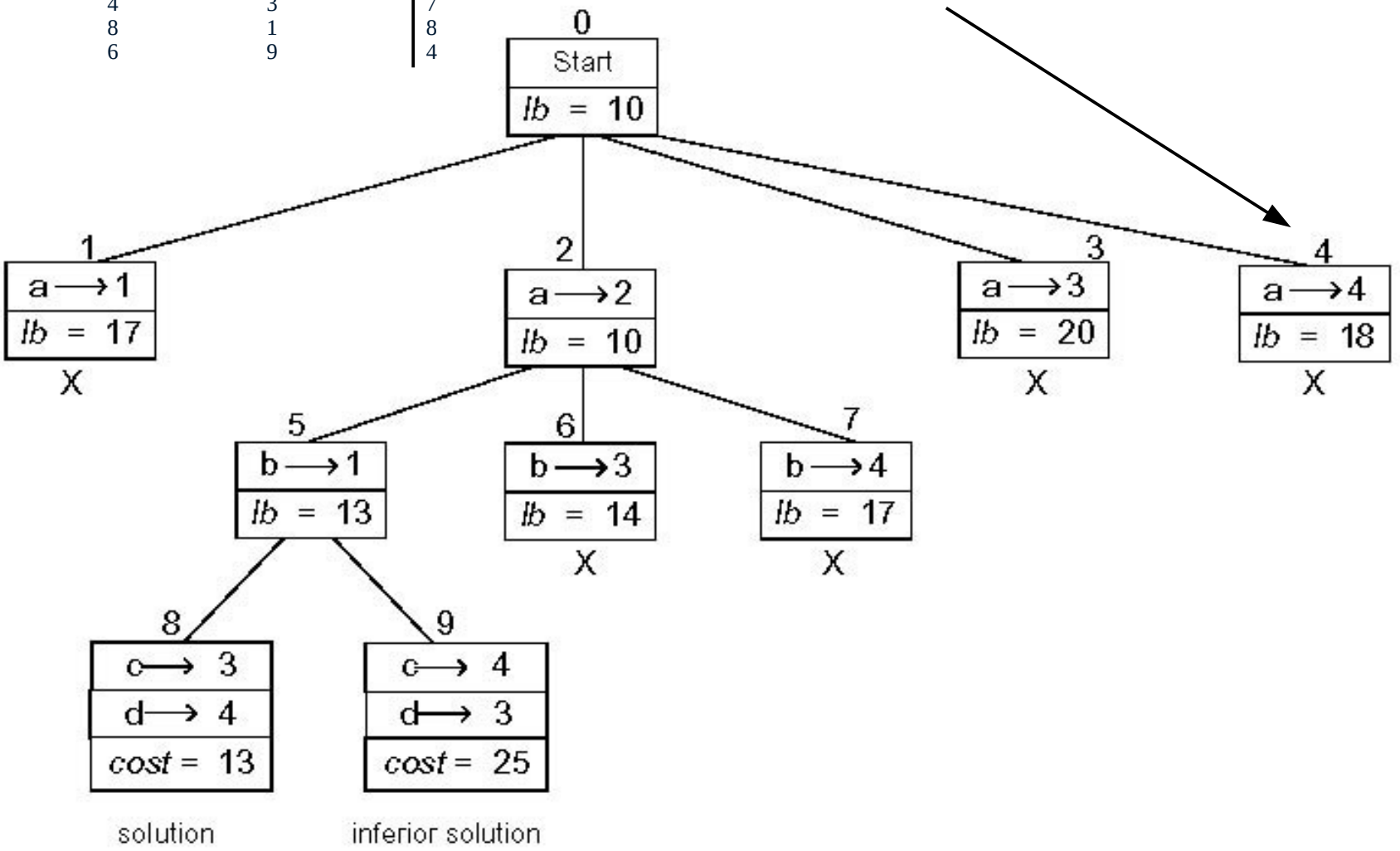
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

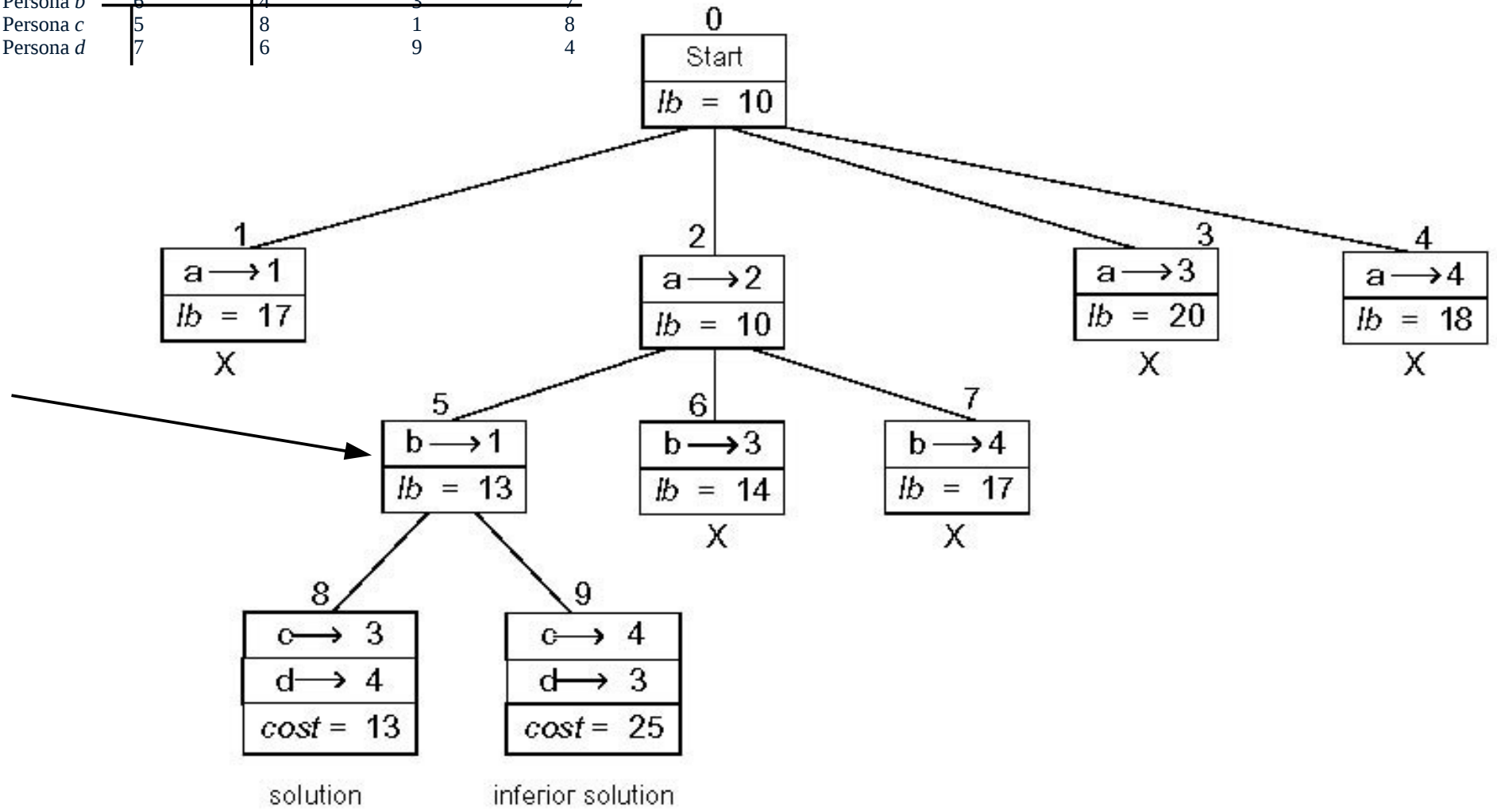
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

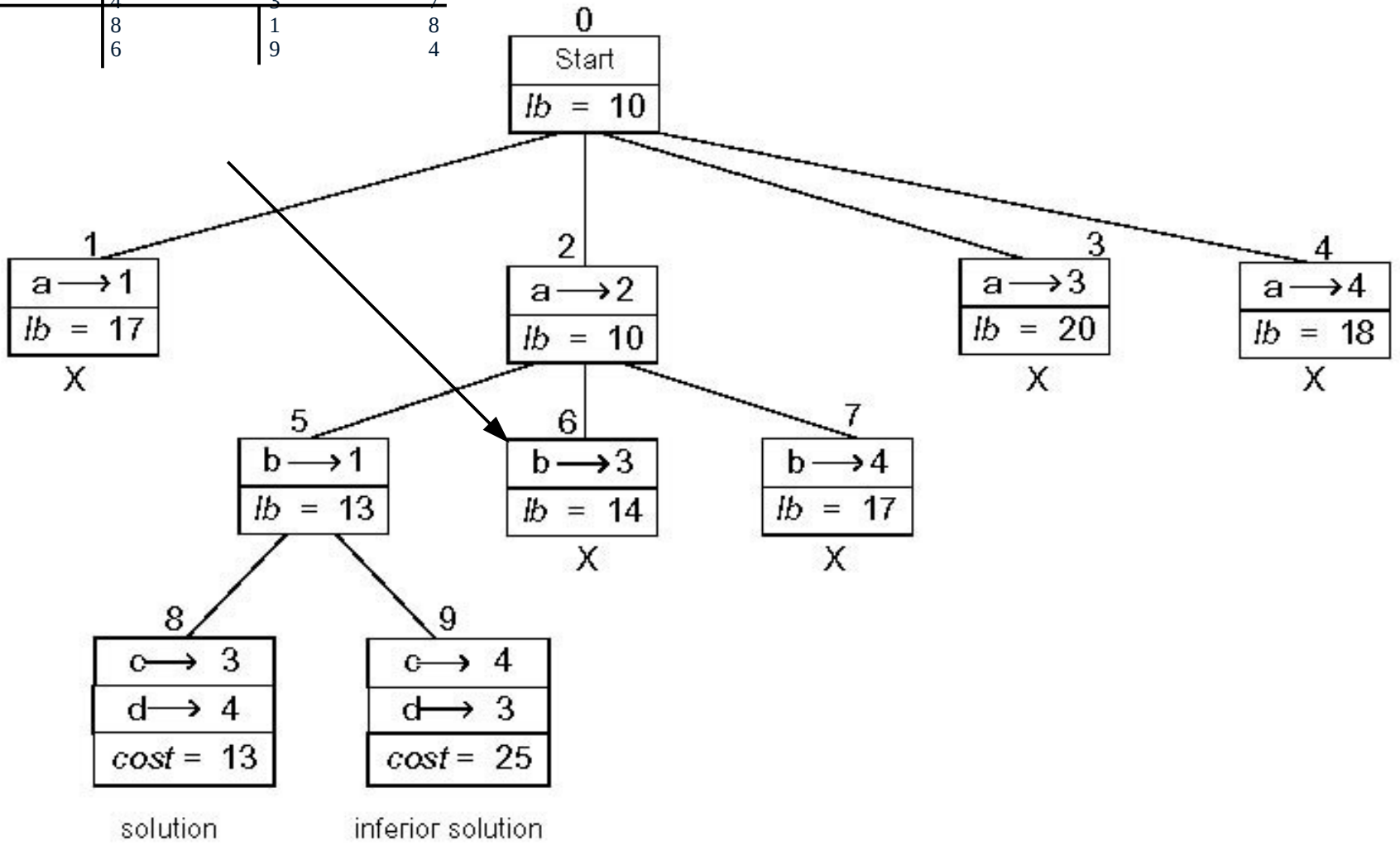
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

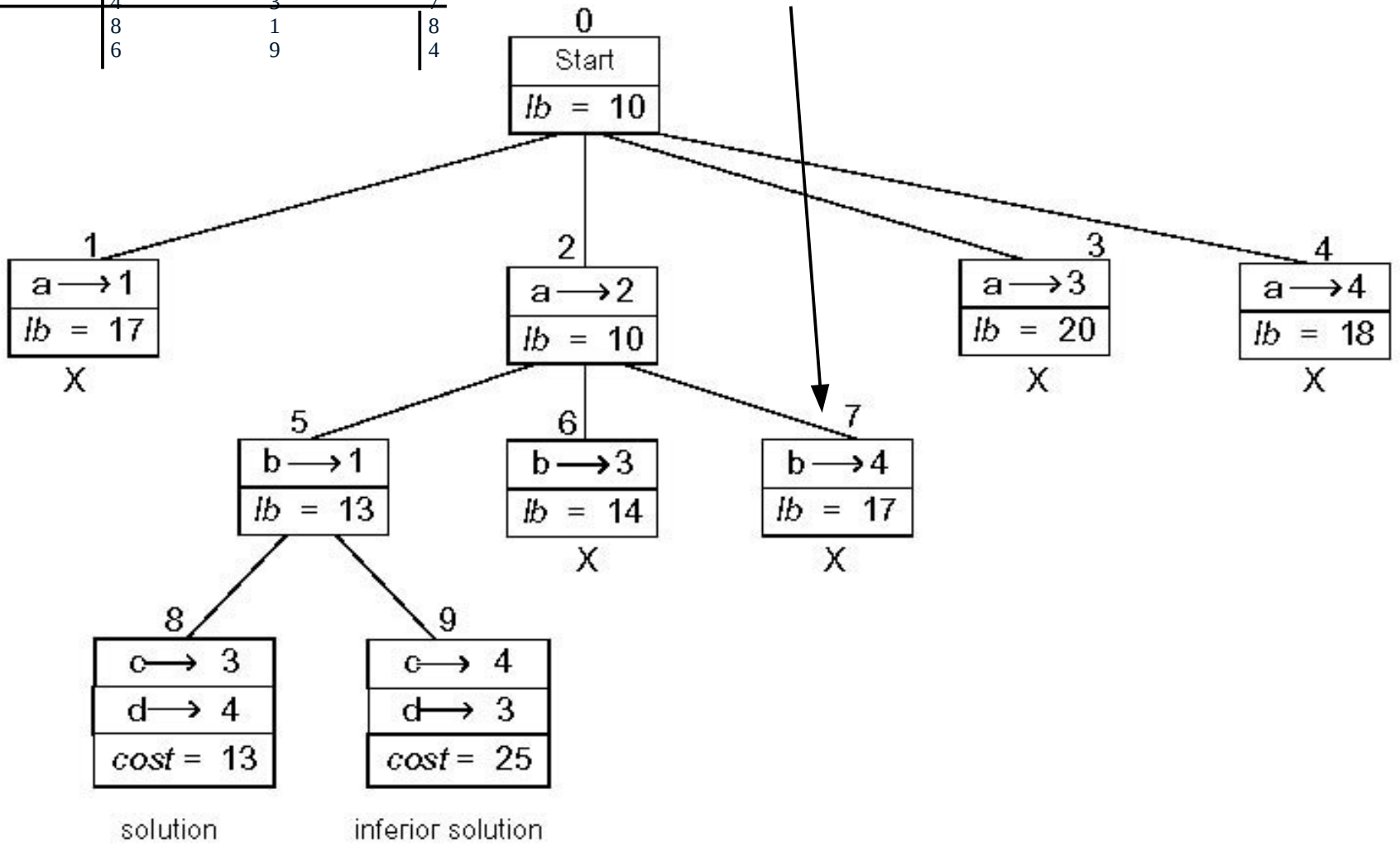
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

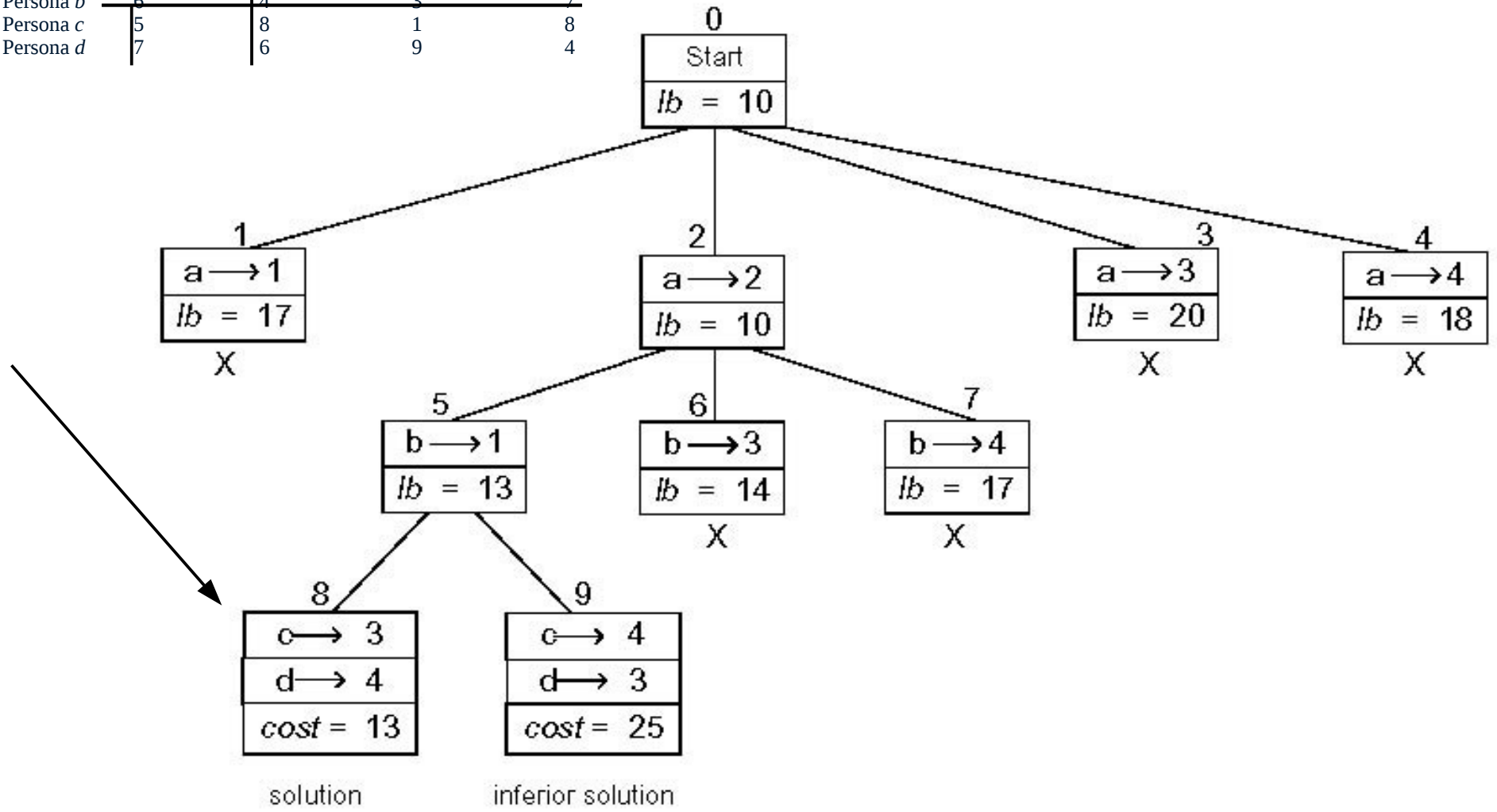
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

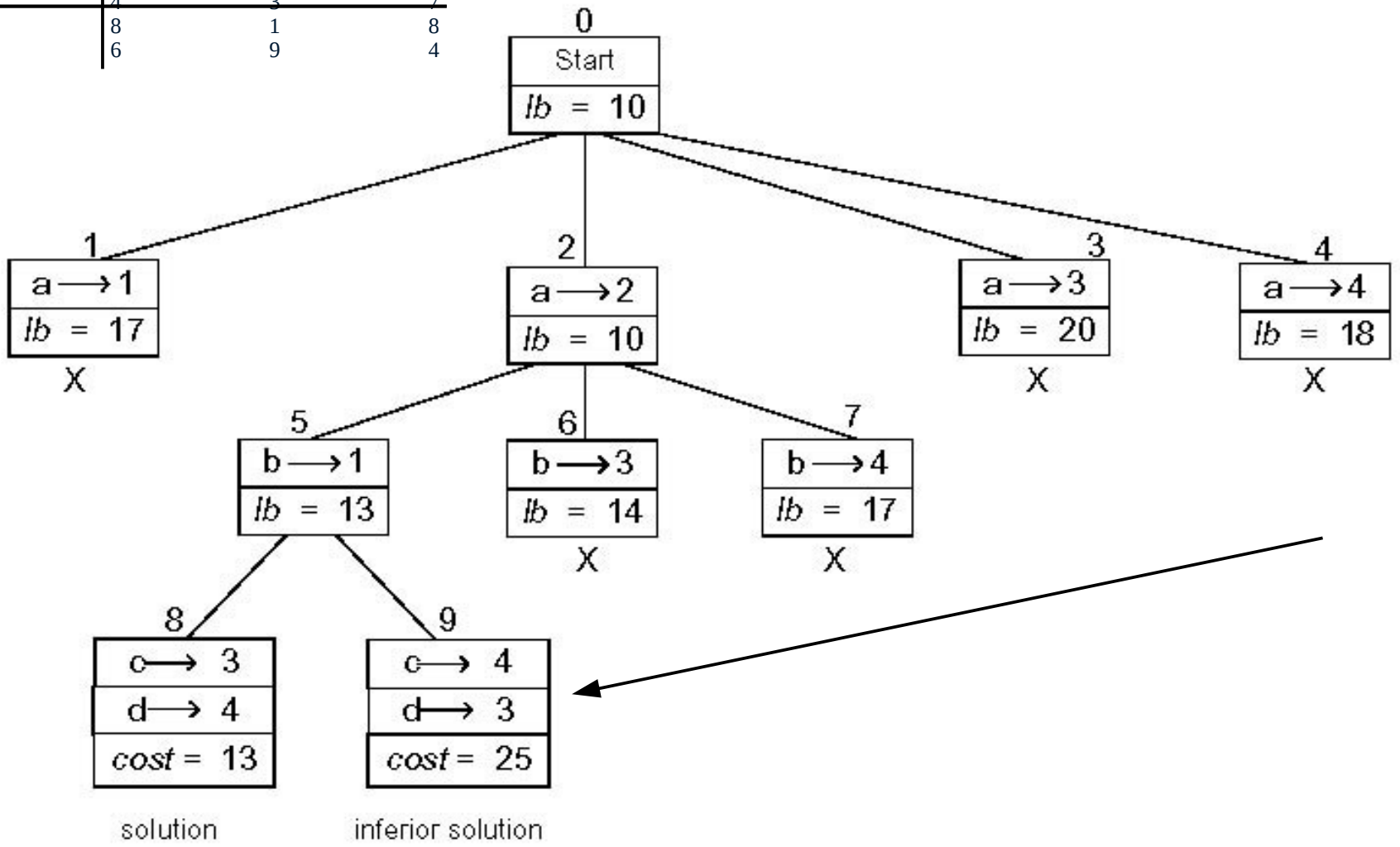
	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Árbol de estados

	Tr 1	*Tr* 2	*Tr* 3	*Tr* 4
Persona *a*	9	2	7	8
Persona *b*	6	4	3	7
Persona *c*	5	8	1	8
Persona *d*	7	6	9	4



En cada nodo se indica la decisión y el valor de la cota inferior (lb)

Problema del viajante de comercio

- Podemos usar cualquiera de las cotas inferiores comentadas anteriormente (aunque en B&B interesa que las cotas sean bastante precisas).
- Emplearemos en el ejemplo una variación de la cota 2 (el costo del camino acumulado más la suma de los menores arcos salientes de cada vértice del que aun no se ha salido)

Ejemplo

- Dada la siguiente matriz de adyacencias, ¿cuál es el costo mínimo posible de visitar todos los nodos una sola vez?

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

→ Mínimo = ***4***

→ Mínimo = ***7***

→ Mínimo = ***4***

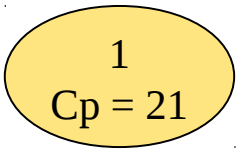
→ Mínimo = ***2***

→ Mínimo = ***4***

TOTAL = 21

Ejemplo

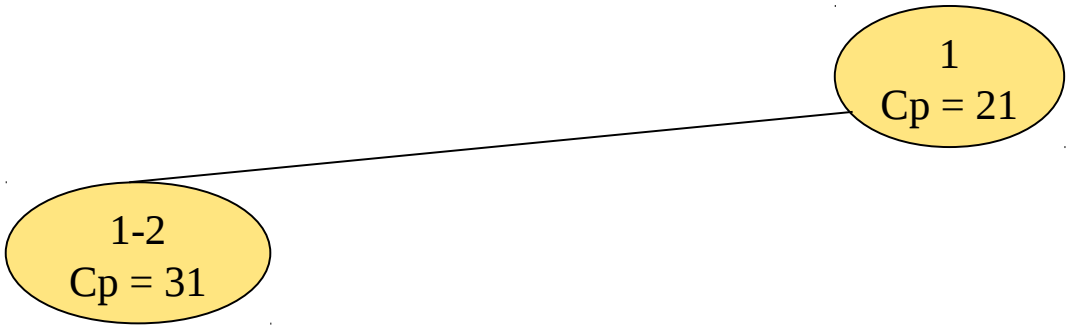
0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Costo mínimo = ∞

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



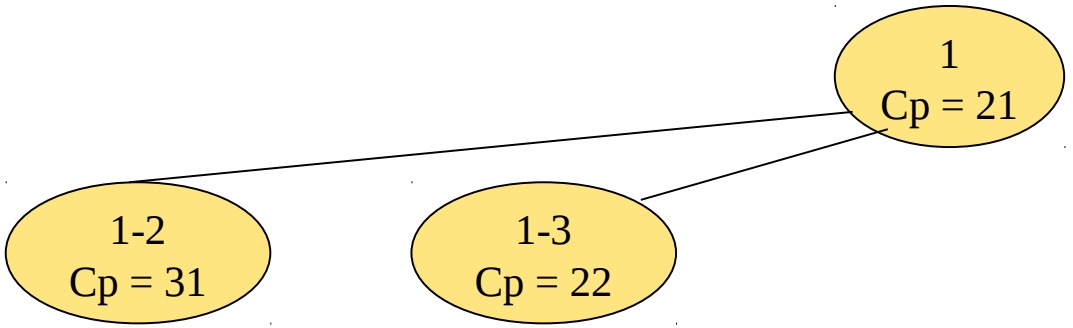
Costo mínimo = ∞

<u>Cálculo del Costo posible:</u>

Acumulado de 1-2 : **14**
Más mínimo de 2-3, 2-4 y 2-5: **7**
Más mínimo de 3-1, 3-4 y 3-5: **4**
Más mínimo de 4-1, 4-3 y 4-5: **2**
Más mínimo de 5-1, 5-3 y 5-4: **4**
TOTAL = 31

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



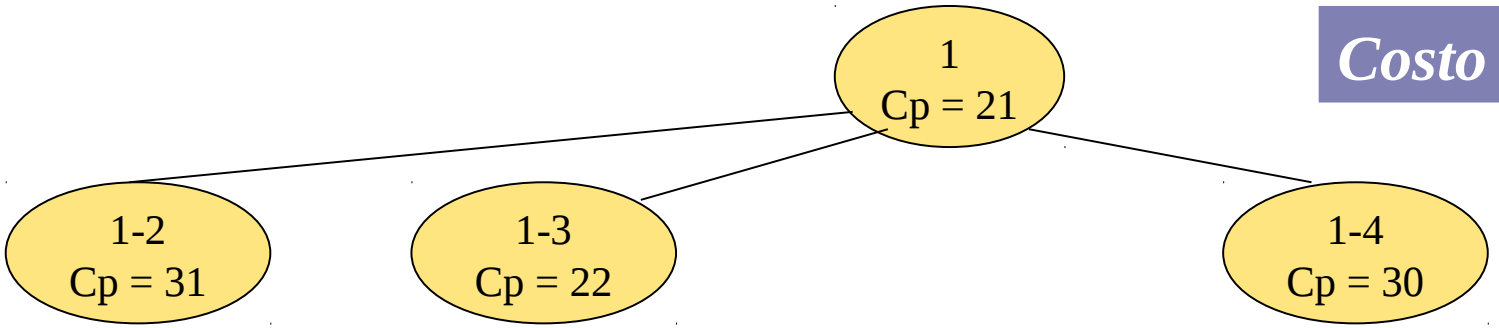
Costo mínimo = ∞

<u>Cálculo del Costo posible:</u>

Acumulado de 1-3 : **4**
Más mínimo de 3-2, 3-4 y 3-5: **5**
Más mínimo de 2-1, 2-4 y 2-5: **7**
Más mínimo de 4-1, 4-2 y 4-5: **2**
Más mínimo de 5-1, 5-2 y 5-4: **4**
TOTAL = 22

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



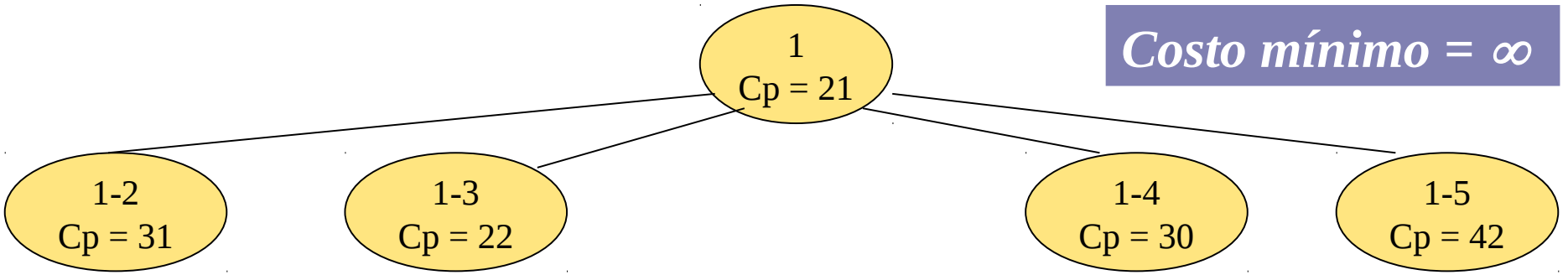
Costo mínimo = ∞

<u>Cálculo del Costo posible:</u>

Acumulado de 1-4 : **10**
Más mínimo de 4-2, 4-3 y 4-5: **2**
Más mínimo de 3-1, 3-2 y 3-5: **4**
Más mínimo de 2-1, 2-3 y 2-5: **7**
Más mínimo de 5-1, 5-2 y 5-3: **7**
TOTAL = 30

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



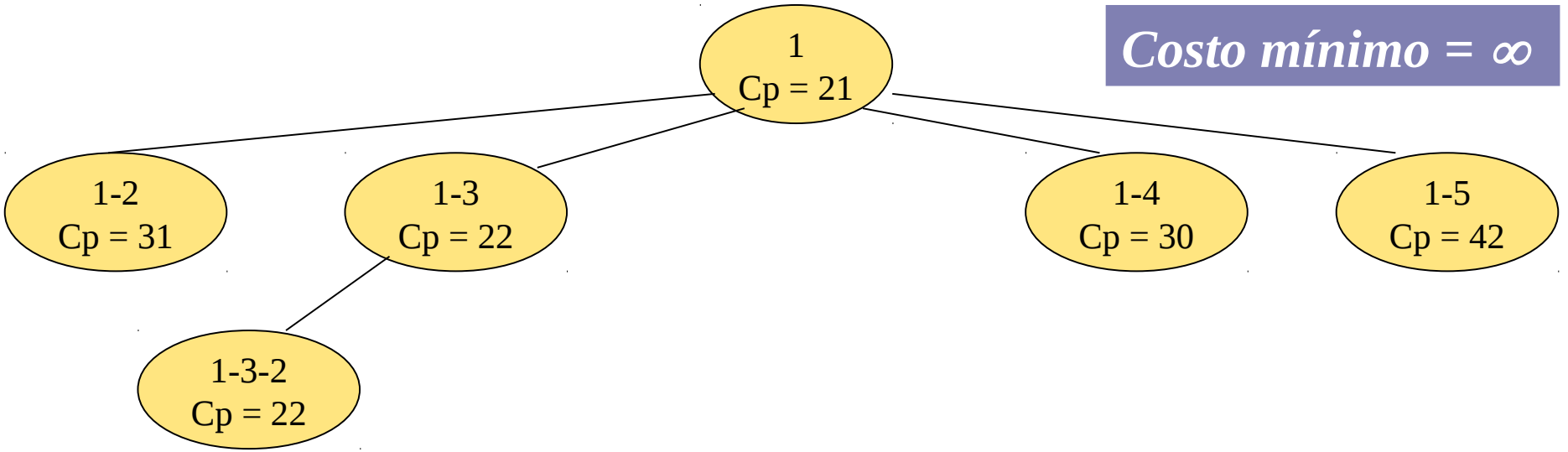
¿Cuál es el mejor nodo para expandir?

Cálculo del Costo posible:

Acumulado de 1-5 : **20**
Más mínimo de 5-2, 5-3 y 5-4: **4**
Más mínimo de 4-1, 4-2 y 4-3: **7**
Más mínimo de 3-1, 3-2 y 3-4: **4**
Más mínimo de 2-1, 2-3 y 2-4: **7**
TOTAL = 42

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



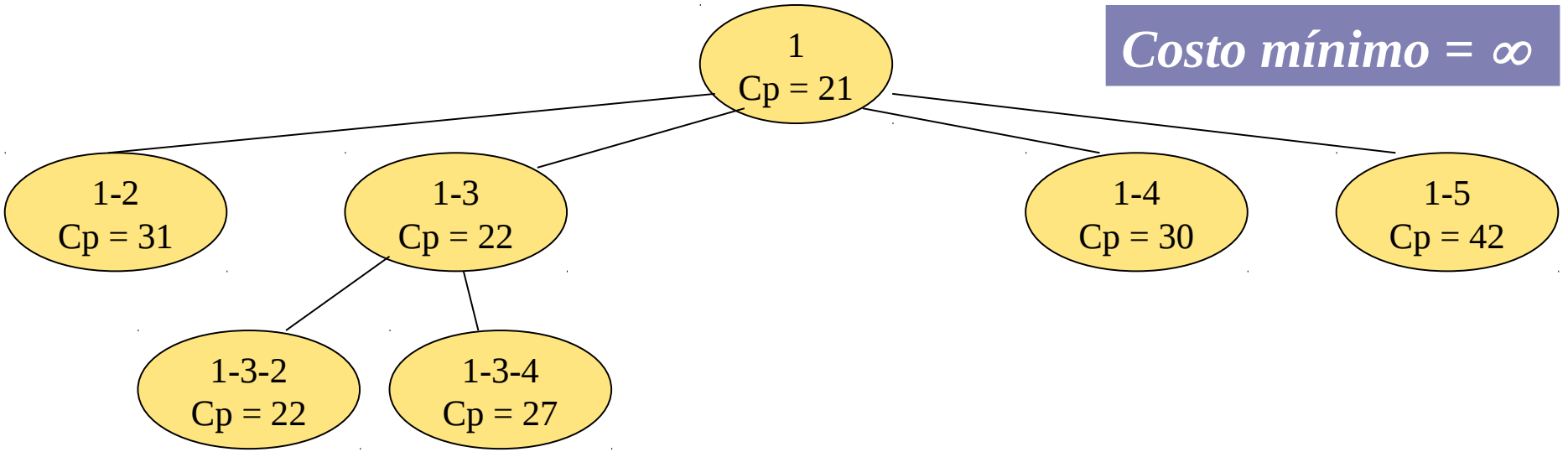
Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-3-2 : **9**
Más mínimo de 2-4 y 2-5: **7**
Más mínimo de 4-1 y 4-5: **2**
Más mínimo de 5-1 y 5-4: **4**
TOTAL = 22

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



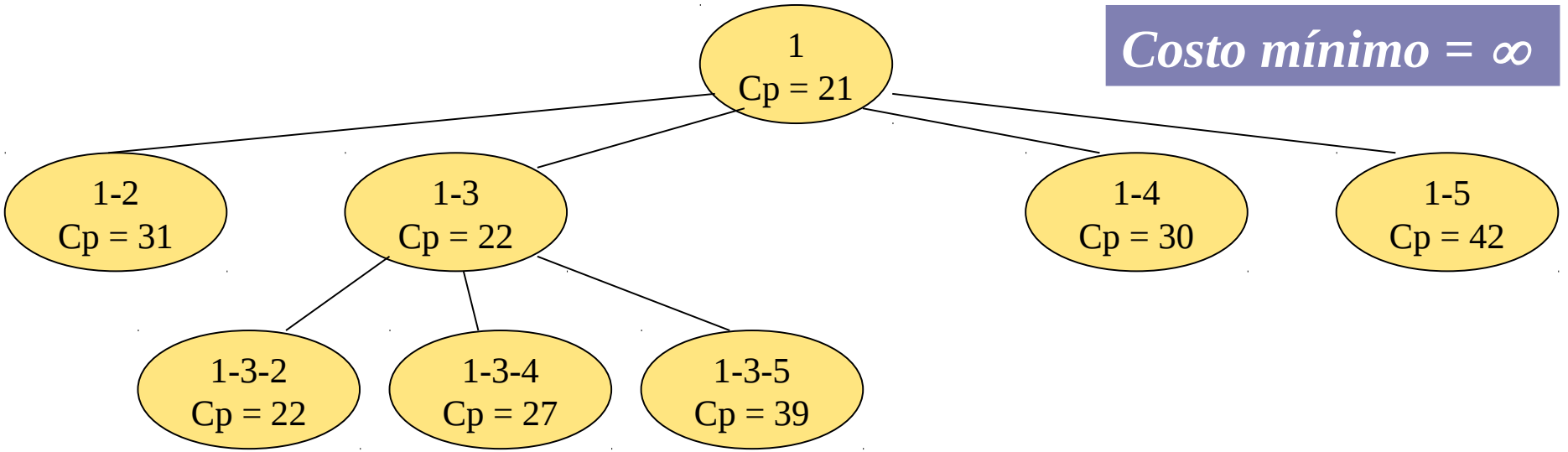
Costo mínimo = ∞

Cálculo del Costo posible:

Acumulado de 1-3-4 : **11**
Más mínimo de 4-2 y 4-5: **2**
Más mínimo de 2-1 y 2-5: **7**
Más mínimo de 5-1 y 5-2: **7**
TOTAL = 27

Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0



Costo mínimo = ∞

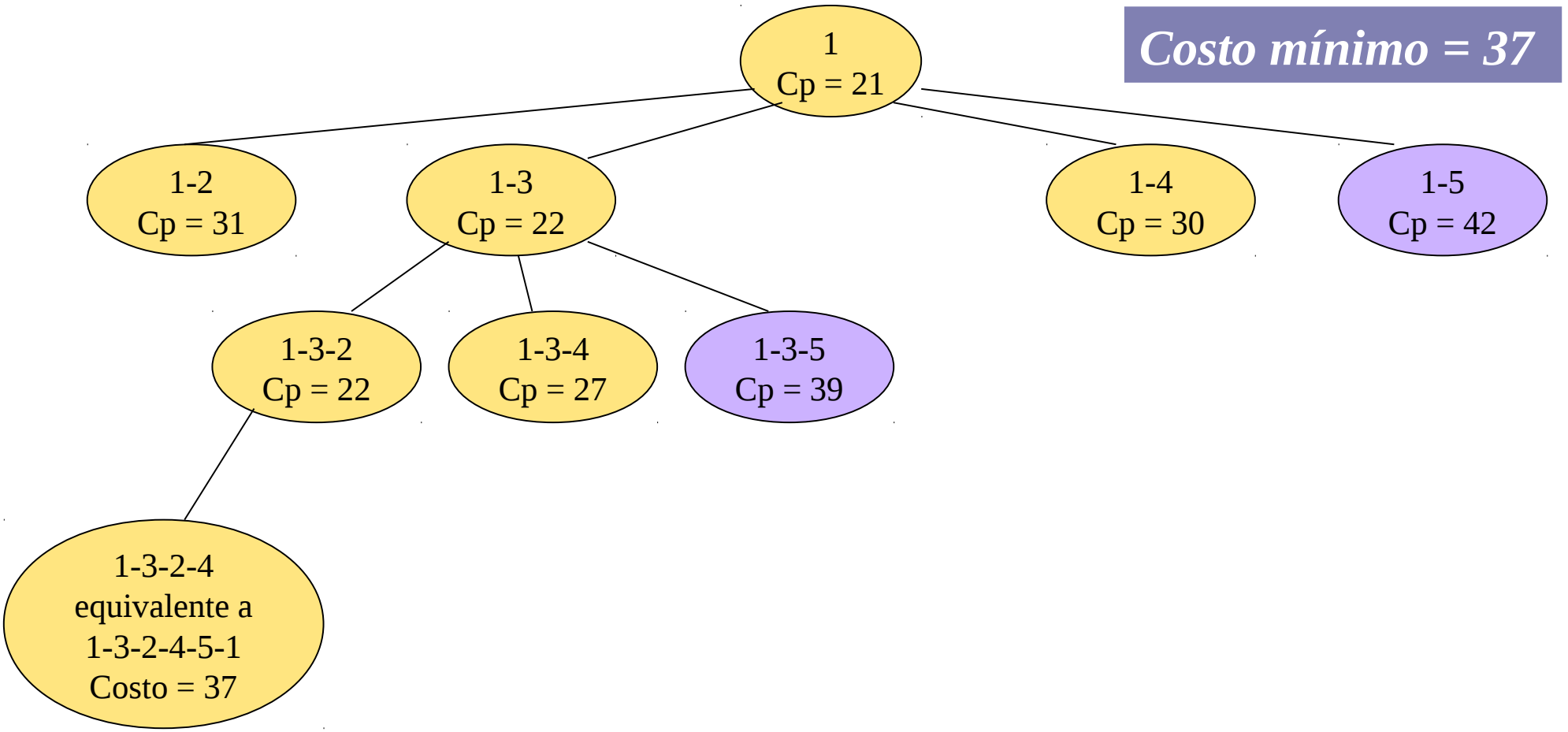
¿Cuál es el mejor nodo para expandir?

Cálculo del Costo posible:

Acumulado de 1-3-5 : **20**
Más mínimo de 5-2 y 5-4: **4**
Más mínimo de 2-1 y 2-4: **8**
Más mínimo de 4-1 y 4-2: **7**
TOTAL = 39

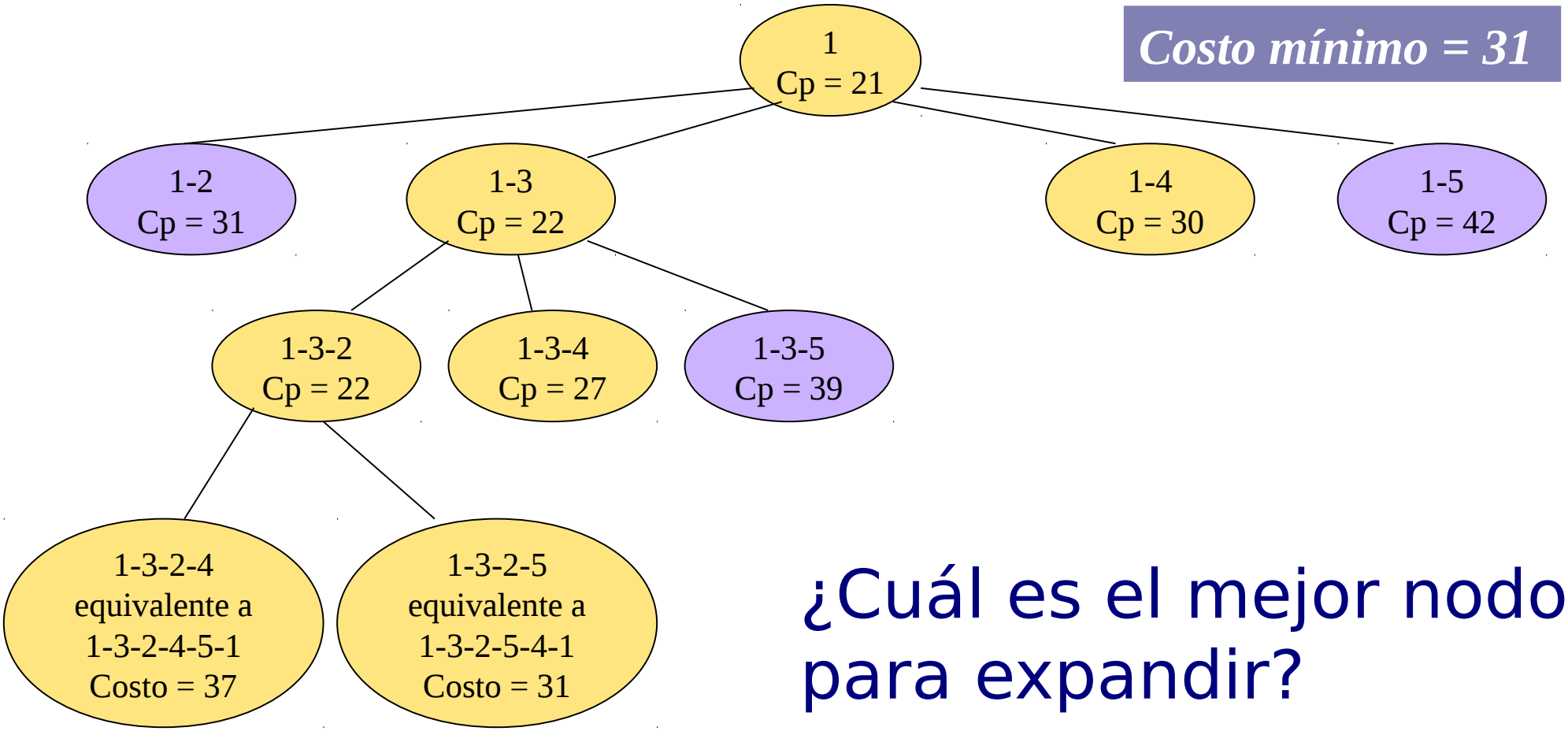
Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Ejemplo

0	14	4	10	20
14	0	7	8	7
4	5	0	7	16
11	7	9	0	2
18	7	17	4	0

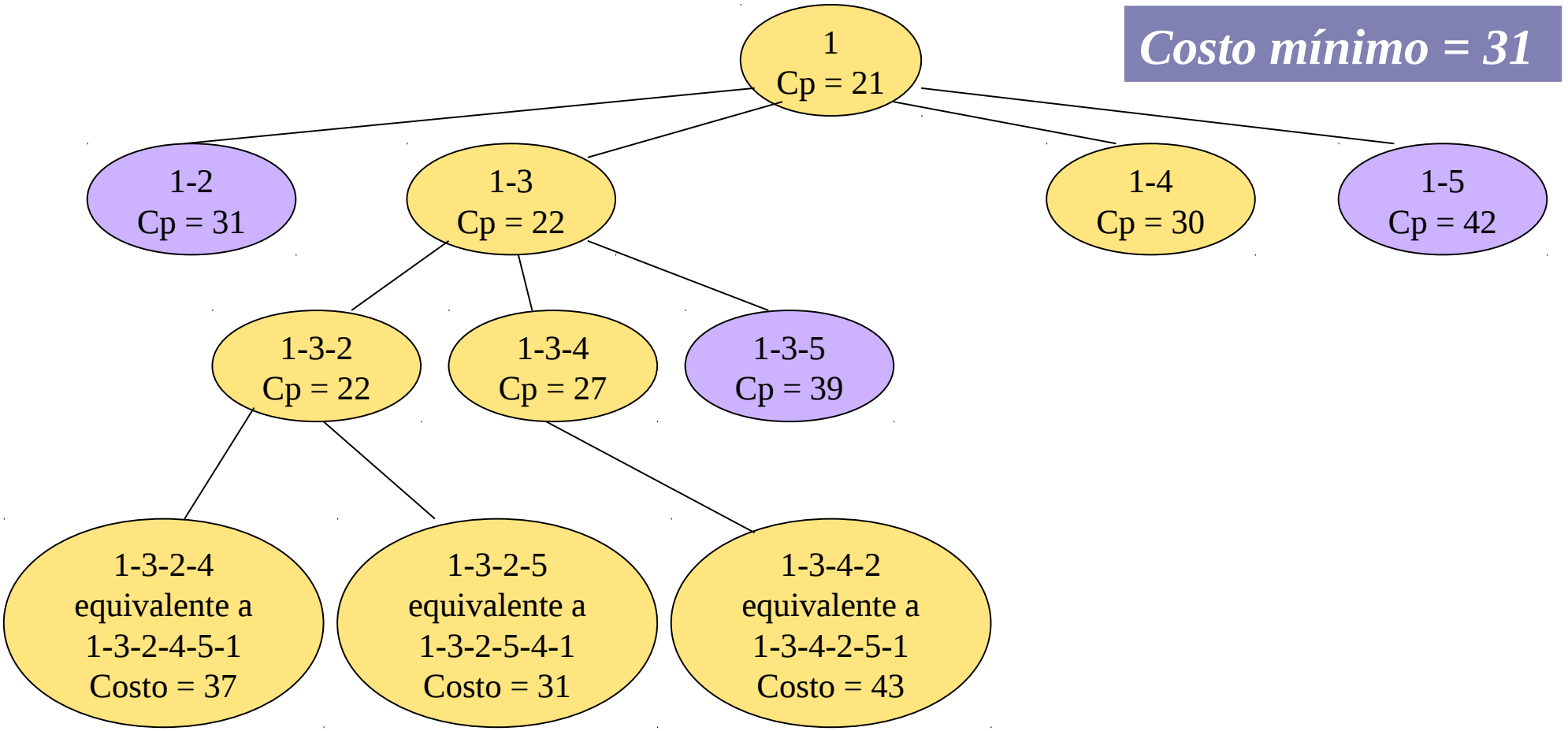


Costo mínimo = 31

¿Cuál es el mejor nodo para expandir?

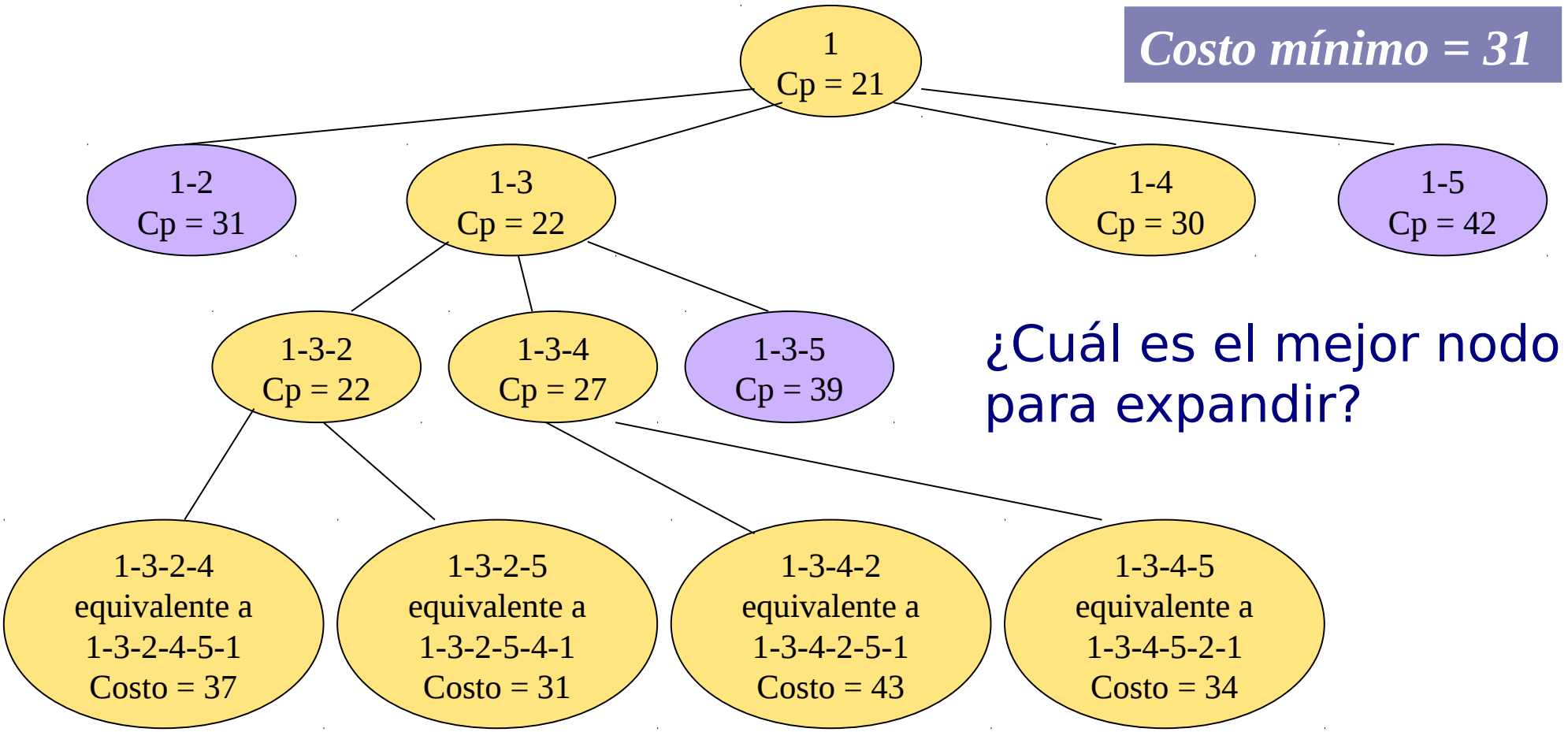
Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



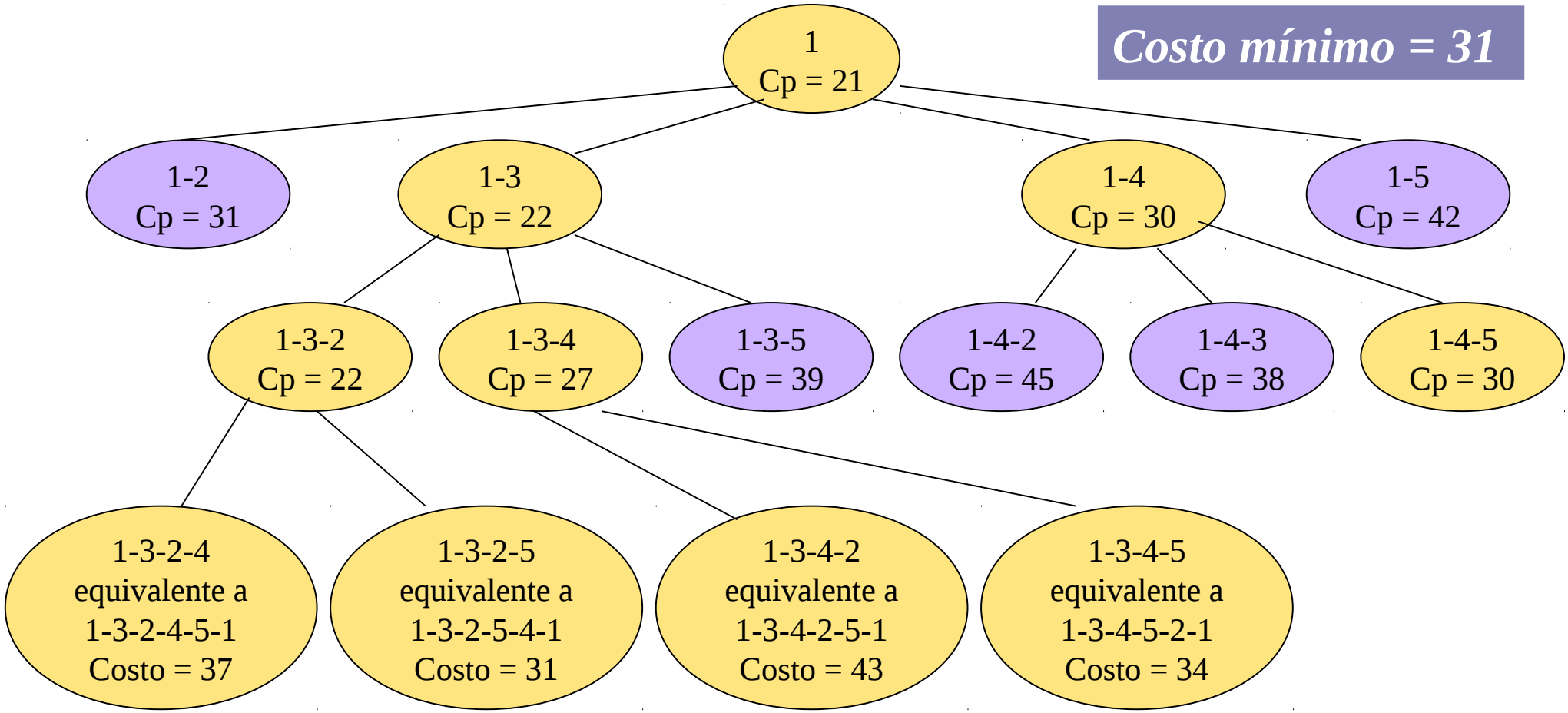
Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Ejemplo

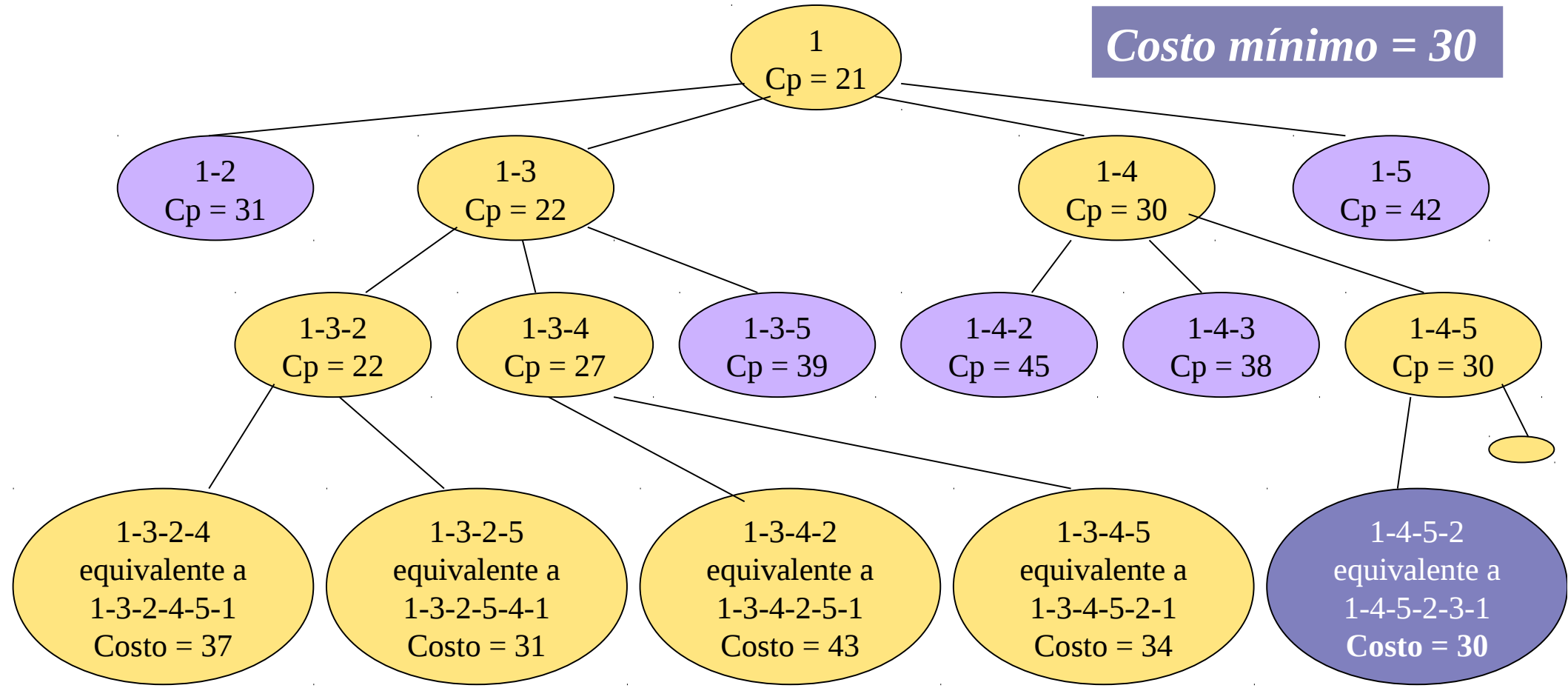
$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



¿Cuál es el mejor nodo para expandir?

Ejemplo

$$\begin{pmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{pmatrix}$$



Eficiencia en B&B

- Al igual que BK depende de:
  - El tiempo necesario para generar la siguiente componente de la solución, sol(k)
  - El número de sol(k) que satisfacen las restricciones explícitas
  - El tiempo de determinar la función de factibilidad (acota las soluciones)
  - El número de sol(k) que satisfacen la función de factibilidad
- Pero además de
  - El tiempo necesario para gestionar el contenedor de nodos vivos

Eficiencia en Branch and bound

- Básicamente, el tiempo de ejecución depende de:
  - **Número de nodos recorridos**: depende de la efectividad de la poda.
  - **Tiempo gastado en cada nodo**: tiempo de hacer las estimaciones de coste y tiempo de manejo de la lista de nodos vivos.
- En el peor caso, el tiempo es igual que el de un algoritmo con backtracking (o peor si tenemos en cuenta el tiempo que requiere la LNV).
- En el caso promedio se suelen obtener mejoras respecto al backtracking.
- ¿Cómo hacer que un algoritmo B&B sea más eficiente?
  - **Hacer estimaciones de costo muy precisas**: Se realiza una poda exhaustiva del árbol. Se recorren menos nodos pero se gasta mucho tiempo en realizar las estimaciones.
  - **Hacer estimaciones de costo poco precisas**: Se gasta poco tiempo en cada nodo, pero el número de nodos puede ser muy elevado. No se hace mucha poda.
- Se debe buscar un equilibrio entre la exactitud de las cotas y el tiempo de calcularlas.