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

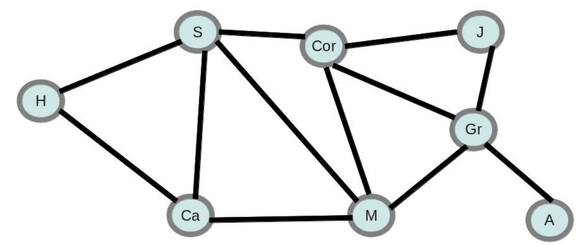
  x(i): 5 3 1 4 2
- La ordenación no es uno de los problemas que habitualmente se resuelven con backtracking

Ejemplo: Coloreo de mapas

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

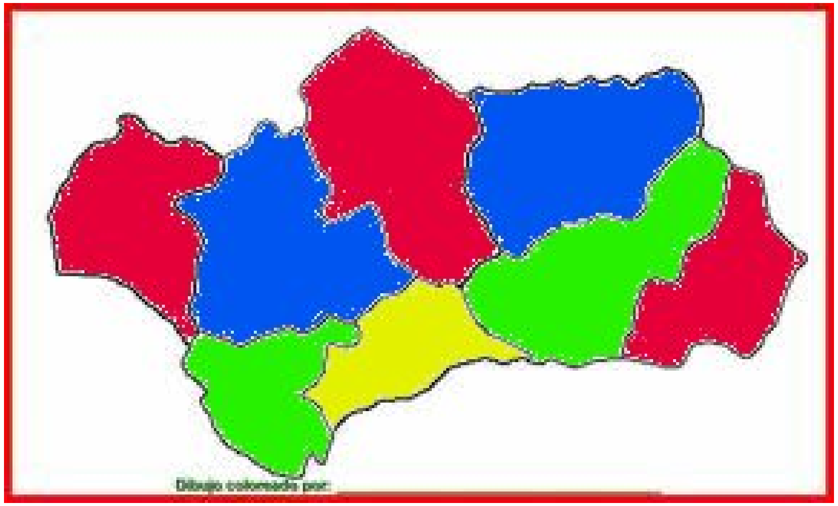
- Países adyacentes deben tener diferente color
- No tenemos suficiente información para elegir dichos colores
- Cada elección del un color nos abre un conjunto de alternativas para solucionar el problema
- Una o varias secuencias de colores nos pueden llevar a la solución

Colorear Mapa



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

Colorear mapa

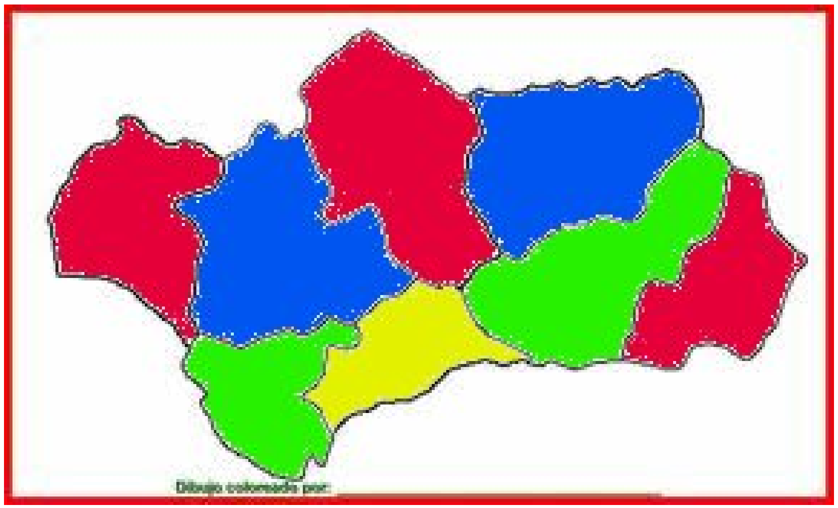


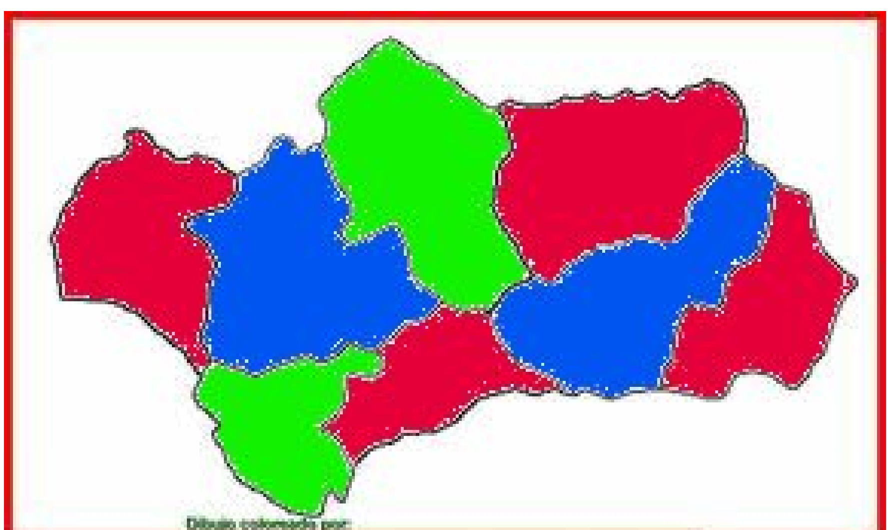
Es óptimo?

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

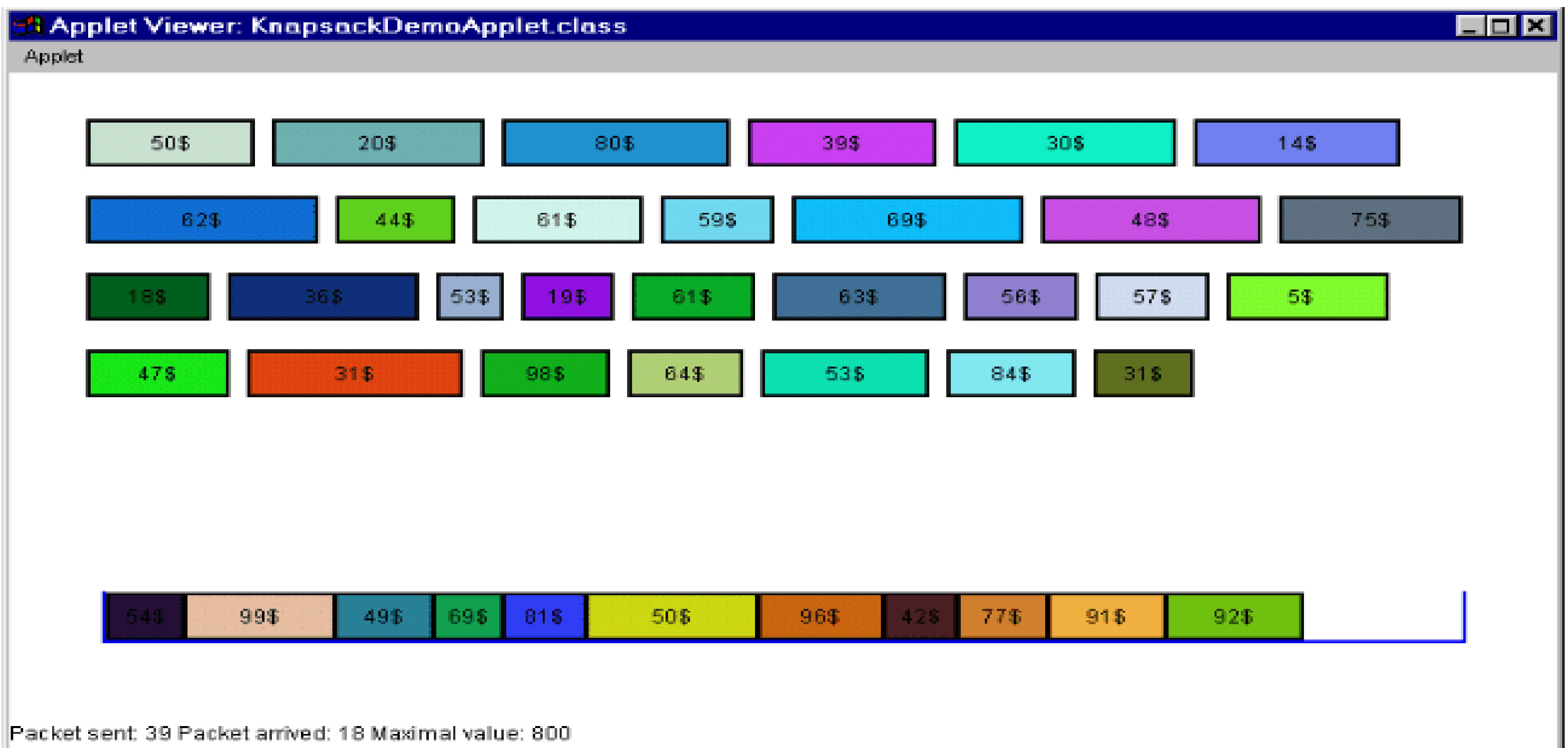




- Su pueden utilizar menos colores? Si
- Supongamos costos
  - Ro 80 Az 60 Ve 40 Am 20
    - Costo_4 = 460
    - Costo_3 = 520
- Cómo minimizar costos?

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 $m = m_1 \cdot m_2 \cdot \ldots \cdot m_n$ 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;
}
```

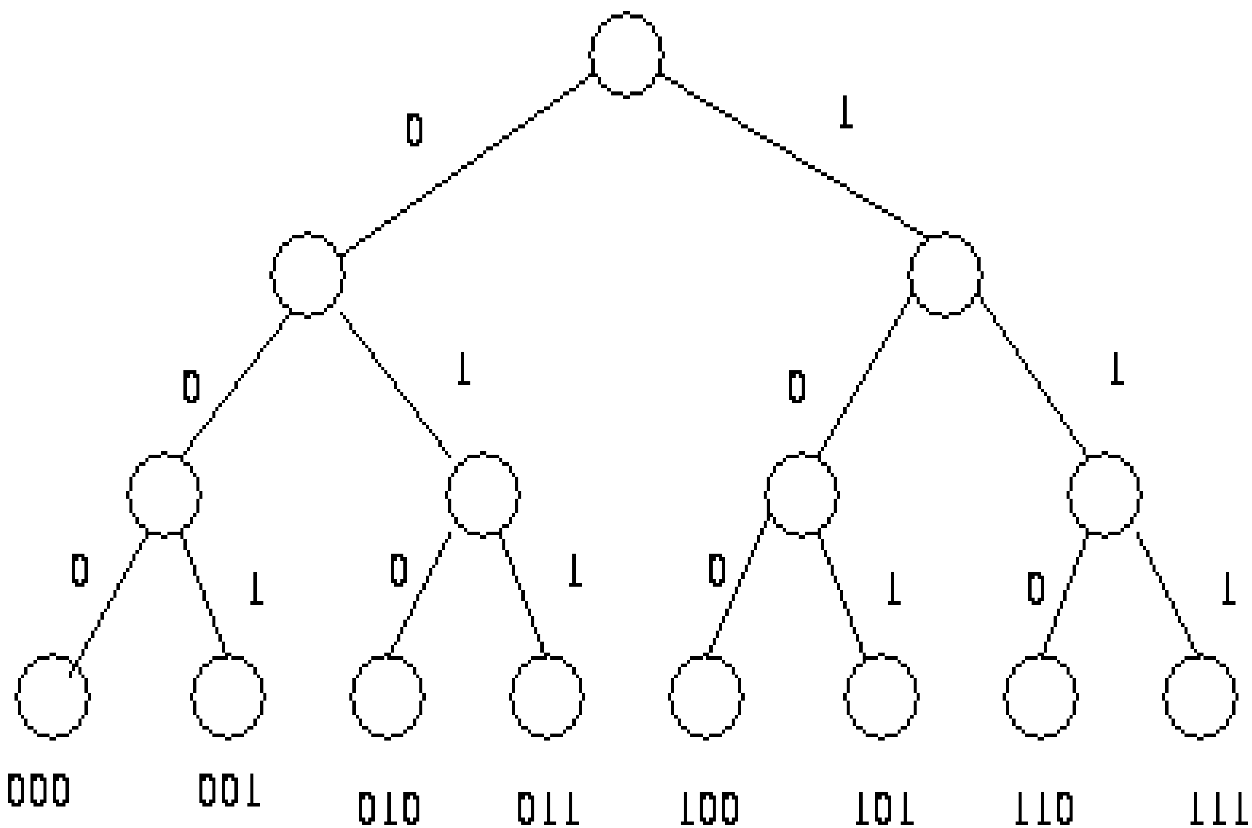
Análisis de Eficiencia

- 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.

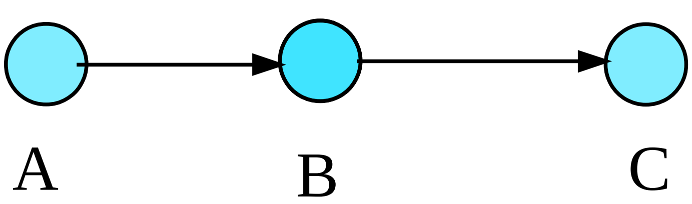
(Sabemos que si el grafo es plano, basta con 4 colores)

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);
  }
}
```

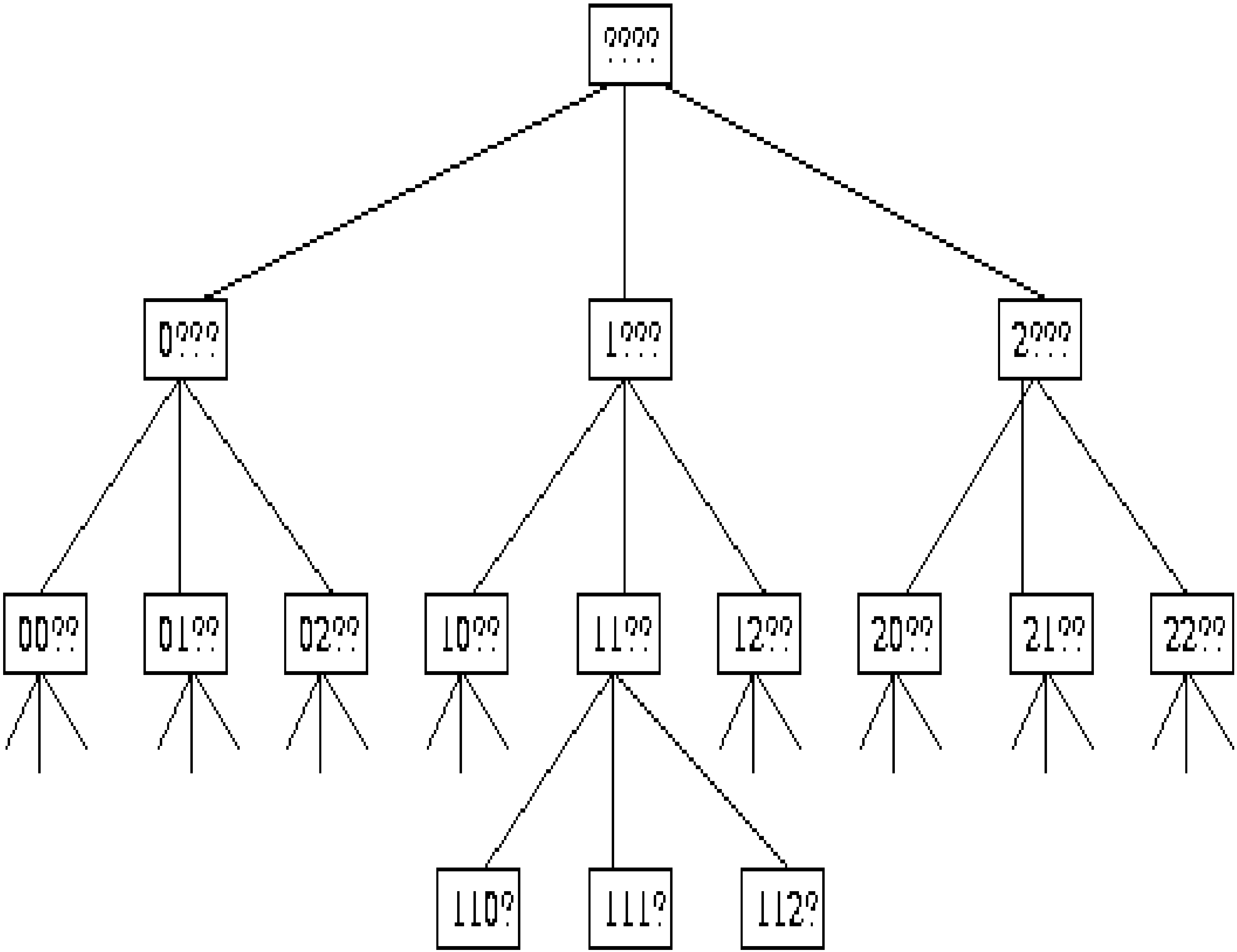


ABC
000
001
002
003
010 ok
011
012 ok
013 ok
020 ok
021 ok
022
023 ok
030 ok
031 ok
032 ok
033
100
101 ok
....

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


????
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 $(x_1, x_2,...,x_k)$, $1 \le k \le n$, y soluciones diferentes pueden tener tamaños de tupla diferentes.
- Restricciones explícitas: $x_i \in \{j\text{: } j \text{ es entero y } 1 \le j \le n\}$
- 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 $1 \le i < n$.

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

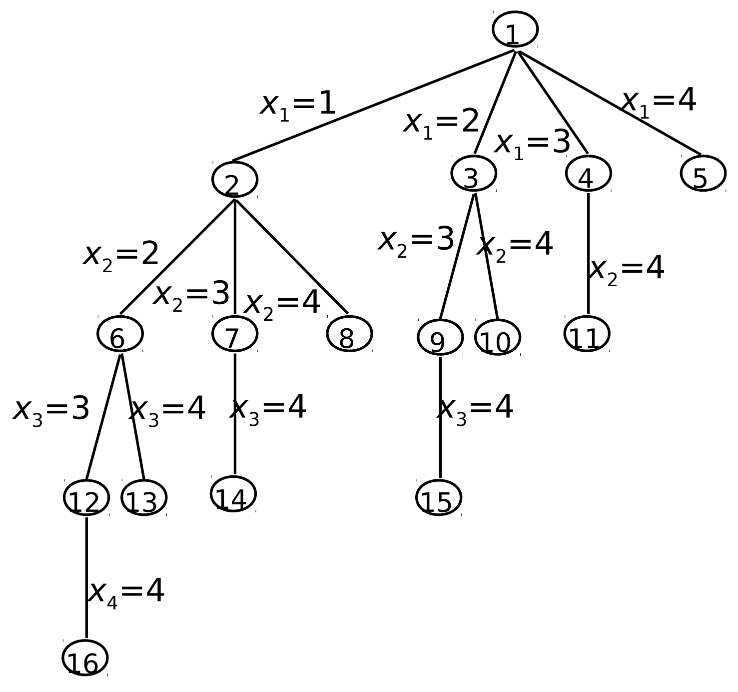
- Dos posibles formulaciones del espacio solución del problema de la suma de subconjuntos.
  - La primera corresponde a la formulación por el tamaño de la tupla variable
  - La segunda considera un tamaño de tupla fijo

- 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.

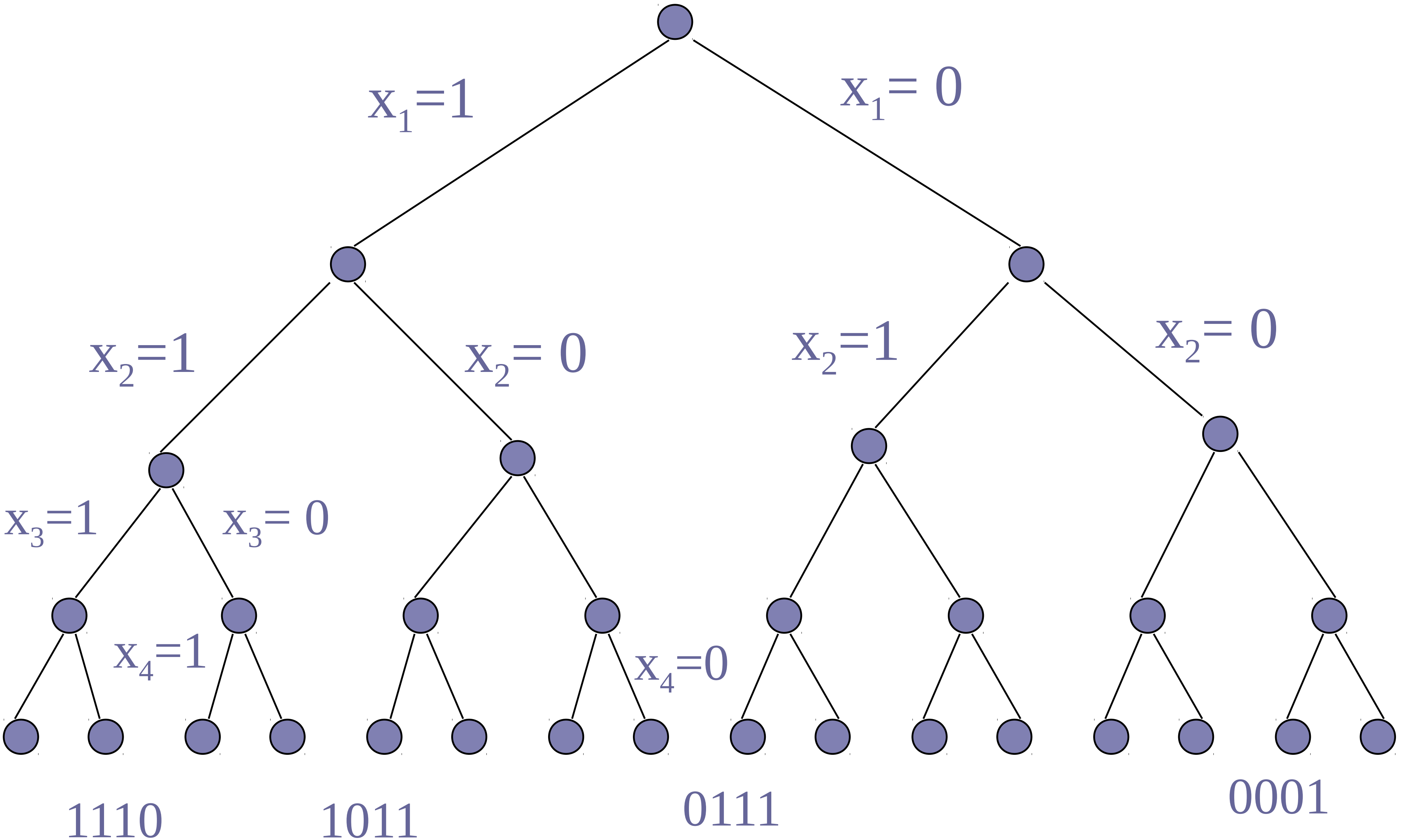
- Así, el subárbol de la izquierda define todos los subconjuntos conteniendo w_1, el siguiente todos los que contienen w_2 pero no w_1, etc.



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.

x			x			x	
	x		x		x		
		x	x	x			
x	x	x	Q	x	x	x	x
		x	x	x			
	x		x		x		
x			x			x	
			x				x

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

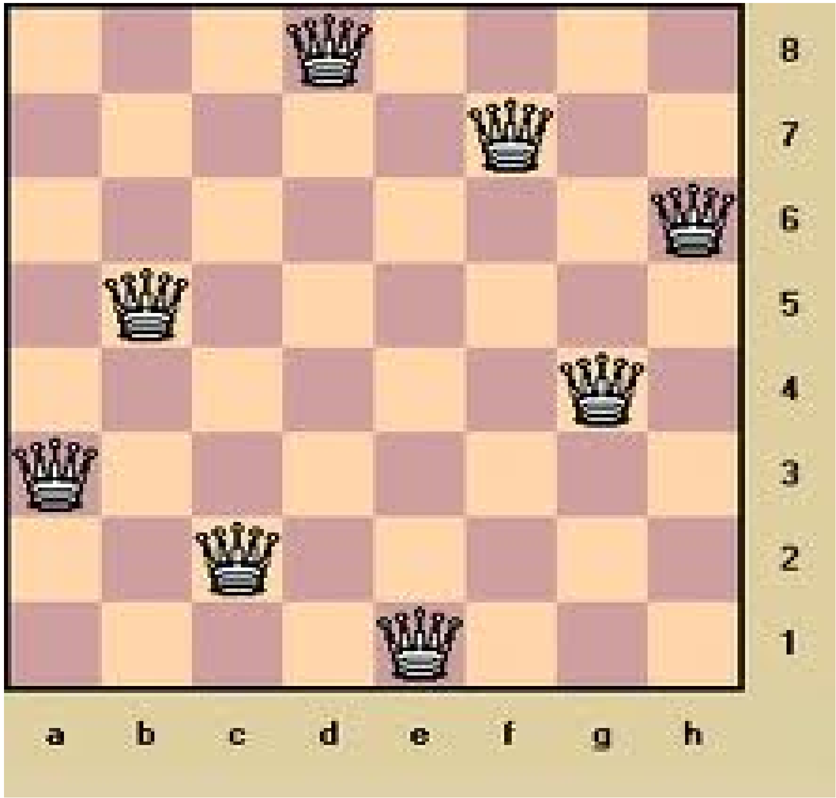
Restricciones implícitas: ningún par de x_i pueden ser iguales. Ningún par de reinas pueden estar en la misma diagonal.

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)

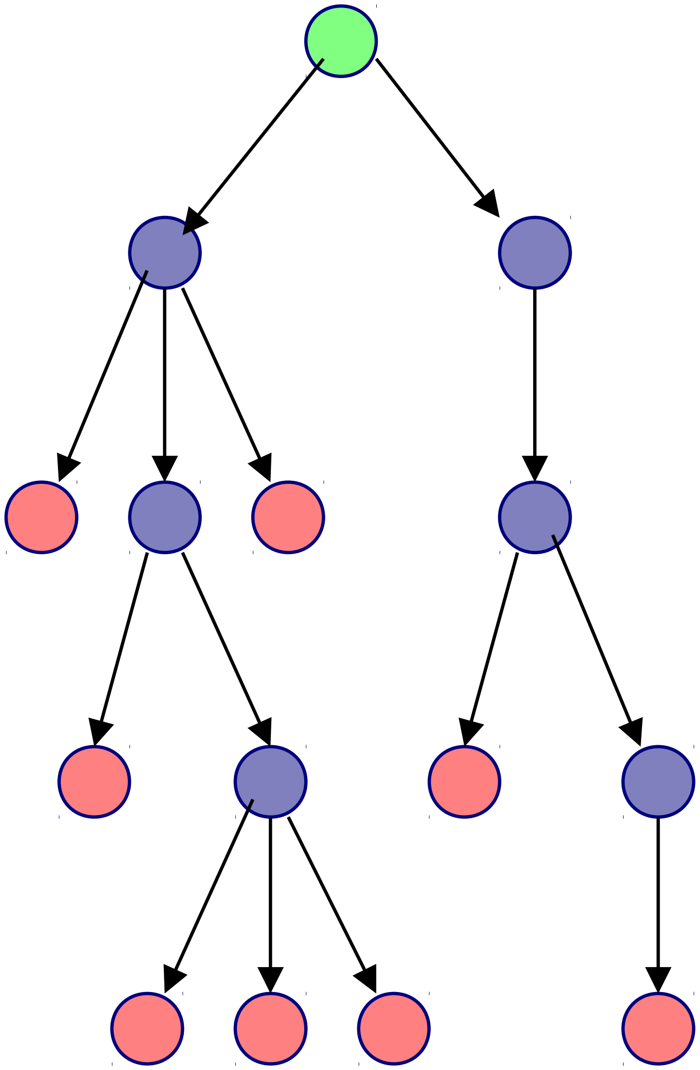


Generalización:

- 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:

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);
    }
  }
}
```

```
void back_iterativo (Solucion & sol)   //BK ITERATIVO
{  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] satisfisface restric
            { if (k == sol.Size() -1 )     // solucion 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)

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

{ // Una solución es factible sii:

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

y

$\sum_{1..k} W(i)X(i) + W(k+1) \leq M$ (si W(i) 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 $X(j)$, $1 \leq j < k$, 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 $W(1) \leq M$ y que $\sum_{1..n} W(i) \geq M$}

Begin

{Generación del hijo izquierdo. Nótese que $s+W(k)+r \geq M$ 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) )  ≤ 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) ≥ M and s + W(k+1) ≤ 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.