PROGRAMACION 3
CURSADA 2025
Inicio 14:00 hs.

Docente:
Federico Casanova.

Técnicas Algoritmicas: Backtracking

Programación 3 – TUDAI 2025

Dr. Federico Casanova

La técnica Backtracking

Supongamos que se nos plantea un problema sobre el cual se quiere obtener una solución, o todas las soluciones, o la mejor solución.

Una forma posible de resolverlo podría ser preguntarnos:

¿Cómo puedo generar todas las posibles soluciones del problema que nos plantean?

Y una vez que sé hacer eso, evaluamos cada solución posible y nos quedamos con aquella o aquellas que resuelven nuestro problema, o con la mejor de ellas.

De esta forma nos aseguramos que **siempre que exista solución, la vamos a encontrar**, ya que revisamos todas las soluciones posibles.

A esta técnica también se la conoce como de “búsqueda exhaustiva”, de “fuerza bruta” o algortimos de retroceso (backtracking).

La técnica Backtracking

Problema: Adivinar una contraseña de 4 números.

Y si pudiera contener letras ?

Cuántas combinaciones posibles ???

n cantidad de números posibles en cada dígito de la clave.
m cantidad de digitos de la clave.

Tendremos una cantidad de opciones igual a n.n….n (m veces) = $\mathbf{n^m}$

Problema: Ordenar un array de números.

20, 4, 5, 23, 7, 1, 9, 18

Sabemos decir si un array está ordenado, pero no sabemos cómo ordenarlo queremos aplicar la técnica de fuerza bruta.

Cuántas combinaciones posibles ???

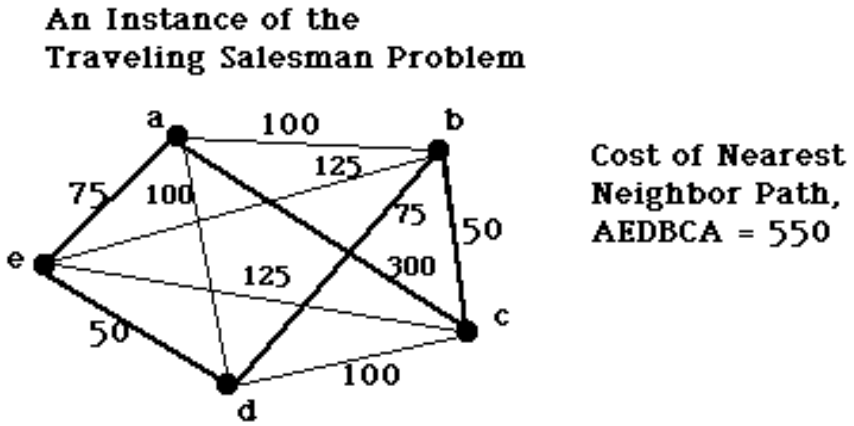
Si n es la cantidad de elementos del conjunto. Cuántas secuencias (arrays) distintos se pueden formar con esos n elementos?
Serían todas las permutaciones de esos n elementos.
Hay **n!** permutaciones posibles de n elementos.

Ejemplo NP-Completo(*): PAV

PAV (Problema del agente viajero – TSP Travelling salesman problem)

Dado un grafo no dirigido G, rotulado y completo (todos conectados con todos), encontrar un ciclo de costo mínimo que pase por todos los vértices una sola vez.

O sea, encontrar un recorrido desde un nodo origen, que pase sólo una vez por cada nodo del grafo y retorne al origen, y cuya suma de los rótulos de los arcos sea mínima.



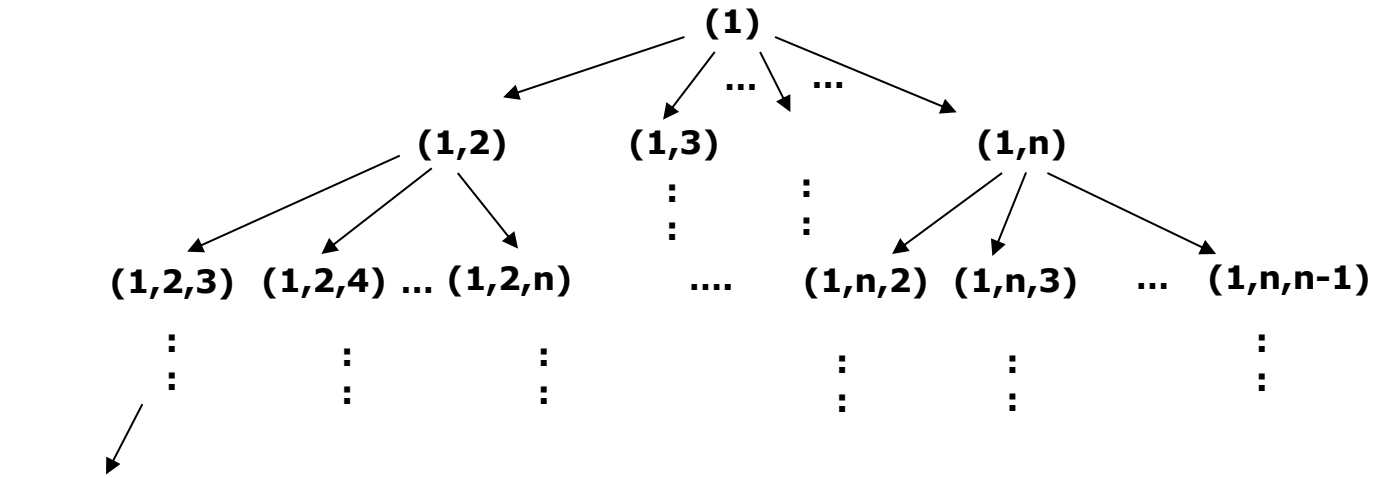
(*) Problemas para los que no se conoce algoritmo que lo resuleva en tiempo polinomial o sea que su tiempo sea O(polinomio en n), sino que resultan tiempos exponenciales (n en el exponente o n!).

Ejemplo NP-Completo: PAV

Supongamos que el grafo tiene *n* nodos (que representan ciudades). La solución la expresaremos como un vector de *n* elementos:

$S=<x_1,x_2,\ldots,x_n>$

Entonces el recorrido del agente será de x_1 a x_2, de x_2 a x_3, y asi sucesivamente hasta el último tramo del recorrido de x_n a x_1.



Es una candidata a solución del PAV

Espacio de Búsqueda

En cada nodo padre se genera un hijo por cada decisión ***posible***

La solución se va construyendo elemento a elemento

Ejemplo NP-Completo: PSS

PSS (Problema de la suma de subconjuntos):

Dado un conjunto $P \subset \mathbf{N}$, $t \in \mathbf{N}$. Decidir si existe $P' \subseteq P$ tal que $t = \sum_{j \in P'} j$.

Por ejemplo, para el conjunto P={4,6,2,1} y t=9, el algoritmo debe retornar SI, ya que existe al menos un subconjunto P' que lo cumple, por ej.: P'={6,2,1}, donde t=6+2+1=9

La solución la expresaremos como un vector de n elementos: $S=<x_1,x_2,\ldots,x_n>$ con $n=|P|$ y $x_i \in \{0,1\}$, $\forall i: 1 \leq i \leq n$.

Entonces en la solución S habrá un 1 en la posición *i* si el elemento *i-ésimo* del conjunto P pertenece a la solución P', o un 0 si no pertenece.

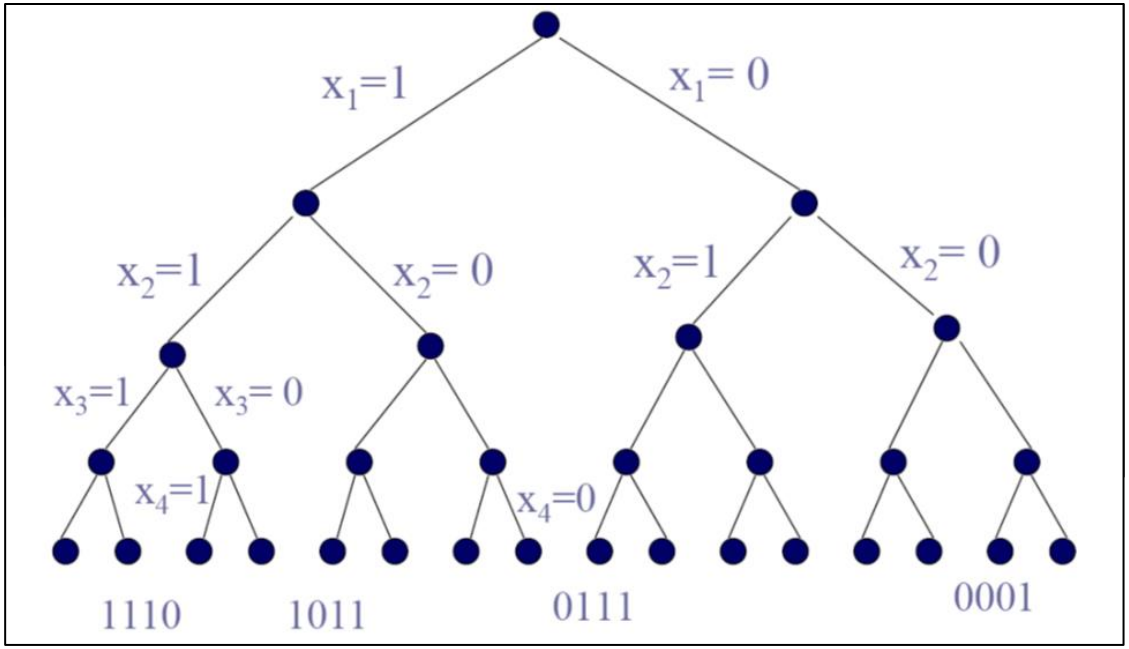
Para el ejemplo anterior, $n=|P|=4$, $S=<x_1,x_2,x_3,x_4>=<0,1,1,1>$, ya que existe P'={6,2,1}

Ejemplo NP-Completo: PSS

$S=<x_1,x_2,x_3,x_4>$

En cada nodo padre se genera un hijo por cada decisión posible

La solución se va construyendo elemento a elemento

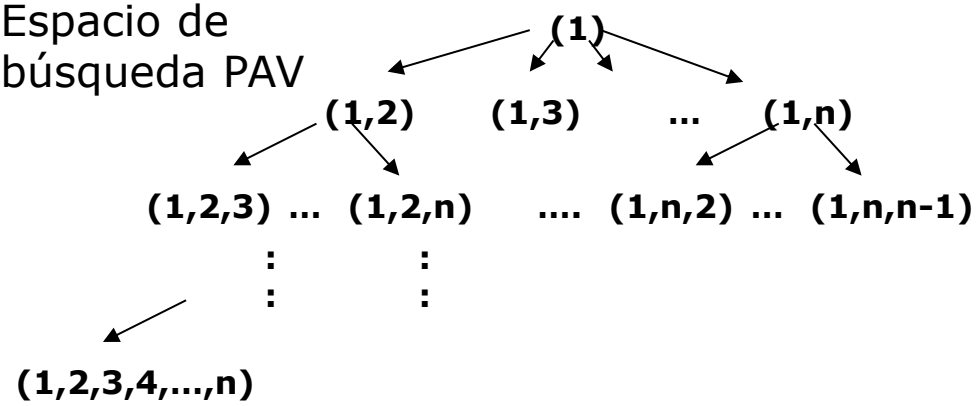


Candidatos= ...<1,1,1,0> ...<1,0,1,1,> ... <0,1,1,1> ... <0,0,0,1>

Espacio de Búsqueda

Backtracking

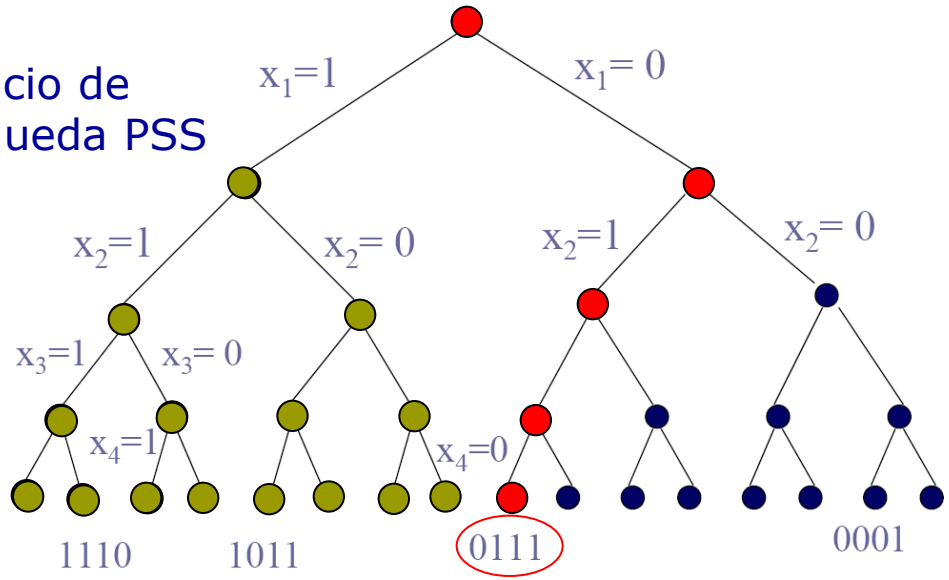
Espacio de búsqueda PAV



Dado un problema, generar y recorrer todo el espacio de búsqueda (todas las posibilidades).

La solución de cada problema se va construyendo de manera progresiva.

Espacio de búsqueda PSS



P={4,6,2,1} ; t=9 ; P'={6,2,1} ; S= <0,1,1,1>

BACKTRACKING
Lehmer (1950)

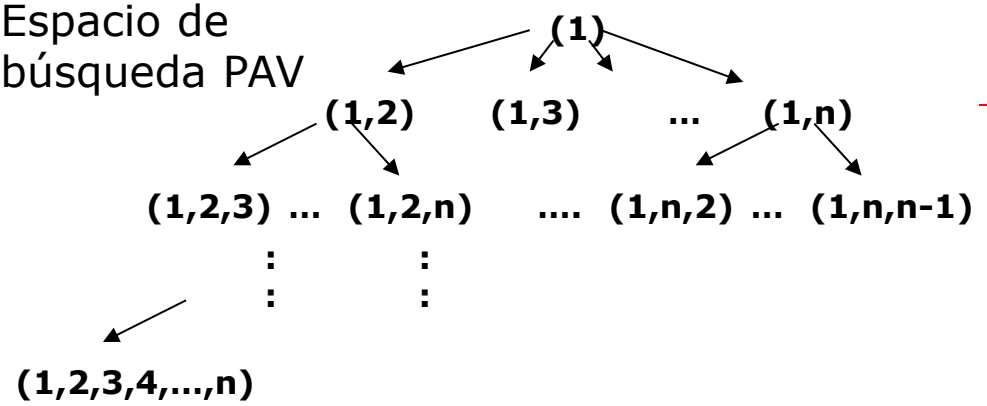
Es el recorrido en profundidad (DFS) del árbol implícito (árbol de recursión) que forma el espacio de búsqueda de un problema.

⇓

Si existe solución la encuentra

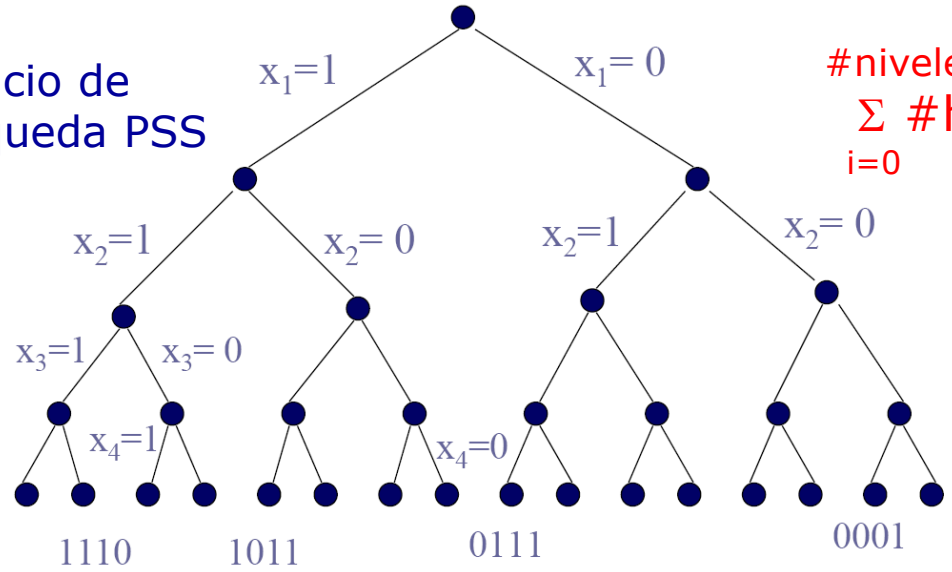
Backtracking - Complejidad

Espacio de búsqueda PAV



$\in$ **O (n!)**

Espacio de búsqueda PSS



#niveles

$\sum_{i=0}$ #hijos ^{i}

$\in$ O (#hijos $^{\#niveles}$)

$\in$ **O (2^n)**

Donde n es la cantidad de elementos del conjunto P.

La técnica Backtracking

Generalmente los algoritmos tienen un planteo recursivo con la siguiente estructura básica. Llevaremos un estado en cada momento:

```
Backtraking (estado e)
{
   Condición de Corte:
      ¿e es una posible solución?
      SI:
            operar con la solución
            Ej.: fijarse si es la mejor hasta el momento, o
                 agregarla a una lista de soluciones, o
                 imprimir, etc ,etc
      NO:
             Para cada hijo c del estado actual e:
                    Backtraking(c) /// EXPLORAR recursivamente a partir de c
}
```

Backtracking

Dado que las soluciones se van construyendo progresivamente, generalmente podemos encontrar restricciones que nos permitirán evaluar si una solución parcial es factible (o sea si nos puede llevar o no a una solución candidata):

- Restricciones Implícitas:

Determinan cuáles de las t-uplas (soluciones parciales o soluciones encontradas) dentro del espacio de búsqueda satisfacen el objetivo del problema.

PERMITIRA PLANTEAR
ESTRATEGIA DE PODA
Decidir si se corta o no la generación de la solución a partir de la solución parcial actual.

Backtracking

Para el PAV:

- Restricción Implícita: encontrar un ciclo de costo mínimo (los costos son positivos) que pase por todos los vértices una sola vez.

⇓

ESTRATEGIA DE PODA

- Que los vértices no se repitan en el camino que vamos construyendo.

- Guardar la mejor solución que vamos encontrando hasta el momento. Comparar el costo del camino que estamos construyendo versus el mejor encontrado, si es de mayor costo podar.

Si alguna de las dos condiciones no se cumple, PODAR.
O sea no continuar con ese camino,
cortar la recursión y volver para explorar otras alternativas.

Backtracking

Para el PSS (recordar el conjunto es de números Naturales):

- Restricción Implícita: $\sum_{i=1}^{n} x_{i*}p_i = t$

⇓

ESTRATEGIA DE PODA

Al agregar el elemento x_k a la solución evaluar:

$$\left(\sum_{i=1}^{k-1} x_{i*}p_i + x_{k*}p_k\right) <= t$$

Que sumando x_k a lo que ya tengo no se pase de t

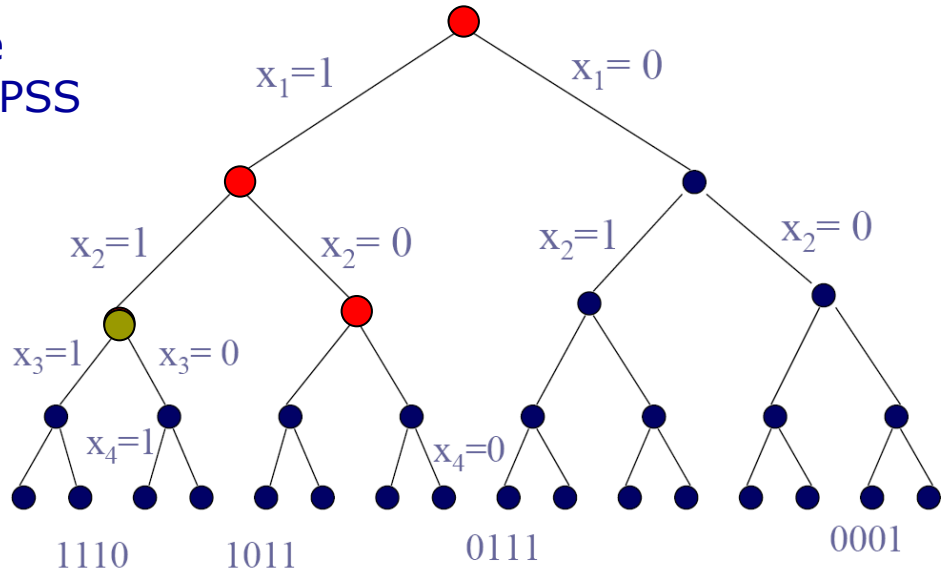
$$\left(\sum_{i=1}^{k-1} x_{i*}p_i + \sum_{i=k}^{n} 1*p_i\right) >= t$$

Lo que tengo hasta el momento, más todo lo que puedo sumar del conjunto restante, tiene que dar igual o más que t

Si alguna de las dos condiciones no se cumple, PODAR, cortar la generación de esa rama completa.

Backtracking

Espacio de búsqueda PSS



P={4,6,2,1} ; t=9 ; P'={6,2,1} ; S=<0,1,1,1>

4*1+6*1=10 > t $\Rightarrow$ PODAR

Backtracking

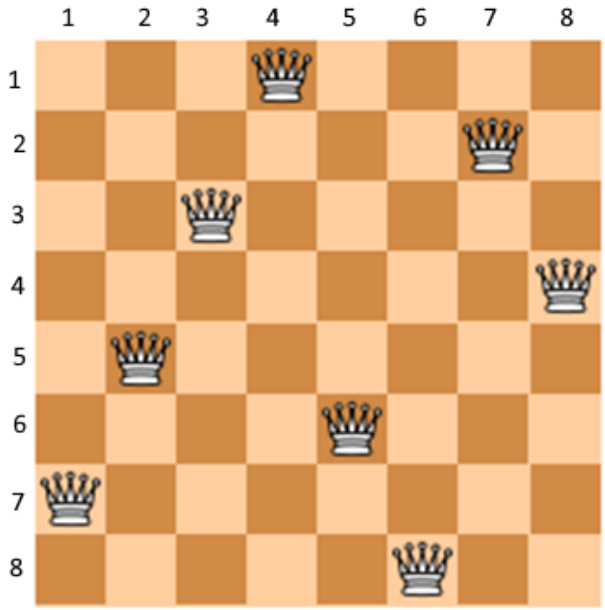
ESQUEMA GENERAL – varias alternativas

```
BACK (estado e, solucionActual *sol)
\\ e: nodo del árbol de soluciones
\\sol: solución actúa, que se va construyendo. También puede llevarse la mejor
encontrada hasta el momento en otro parámetro.
{
    if ( SOLUCION (e))
        OperarSobreSolución (e, sol);
     else {
        int nrohijo = 1;
        estado siguiente;
        while ( HIJOS (nrohijo, e, siguiente ) )
            {   if  ( !PODA ( siguiente, sol) )
                {  AgregarASolucionActual(siguiente, sol);
                   BACK ( siguiente, sol);
                   QuitarDeSolucionActual(siguiente, sol);
                }
               nrohijo++;
            }
        }
}
```

Backtracking
8 Reinas

En el juego del ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal.

El juego de las 8 reinas consiste en colocar sobre un tablero de ajedrez de 8x8, 8 reinas sin que estas se amenacen entre ellas.



Backtracking
8 Reinas

Como cada reina puede amenazar a todas las reinas que estén en la misma fila, cada una ha de situarse si o si en una fila diferente.

Podemos representar entonces, sin perder casos posibles, las 8 reinas mediante un vector[1-8].

Donde cada índice del vector representa una fila y el valor una columna. Así cada reina i estaría en la posición (i, v[i]) para i = 1 a 8.

Vector Solución:

V = {v1,v2,v3,v4,v5,v6,v7,v8}, cada vi toma valor entre 1 y 8

Restricciones implícitas:

- Ningún par (i, j) con v[i]=v[j] (todas las reinas deben estar en columnas diferentes).
- Ningún par (i, j) con |j-i|=|v[j]-v[i]| (todas las reinas deben estar en diagonales diferentes).

Backtracking
8 Reinas

```
// Comprobar si la reina de la fila k está bien colocada
// (o sea si no está en la misma columna ni en la misma diagonal
// que cualquiera de las reinas de las filas anteriores)
```

```
bool comprobar (int reinas[], int k)
{
  int i;
  for (i=0; i<k; i++)
    if ( ( reinas[i]==reinas[k] ) ||
       ( abs(k-i) == abs(reinas[k]-reinas[i]) ) )
       return false;
  return true;
}
```

Backtracking
8 Reinas

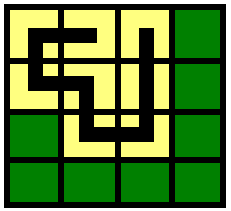
```
// muestra sólo la primer solución
// como cada nivel recursivo es una fila, nunca estarán dos en la misma fila
// n es la cantidad de reinas, k es la reina por la que iniciamos k=0 en la primer
// llamada. La eficiencia será O(n!)

bool NReinas (int reinas[], int n, int k)
{
    bool exito = false;
    if (k==n) { // Condición de corte: No quedan reinas por colocar
                exito = true;
    } else { // Aún quedan reinas por colocar (k<n)
              reinas[k]=0;
              while ((reinas[k]<n) && !exito) {
                    if (comprobar(reinas, k))
                          exito = NReinas (reinas, n, k+1);
                    reinas[k]++;
               }
    }
   return exito; // La solución está en reinas[] cuando ok==true
}
```

PROBLEMA: EL CABALLO DE ATILA

Enunciado:

Por donde pisa el caballo de Atila no vuelve a crecer el pasto. El caballo entró por una casilla al jardín de m x m casillas, empezó su paseo y volvió a casilla de entrada, es decir, hizo un recorrido cerrado. No visitó dos veces una misma casilla, se movió de una casilla a otra vecina en forma horizontal o vertical, pero nunca en diagonal. Por donde pisó el caballo, el pasto no volvió a crecer. Luego de terminado el recorrido en algunas casillas todavía había pasto (señal de que en ellas no había estado el caballo). Escriba un algoritmo que devuelva todos los recorridos válidos que pudo haber hecho el caballo.

Representación del jardín en una matriz:
Casillas amarillas donde pisó el caballo, casillas verdes donde no lo hizo.
Un recorrido válido sería:

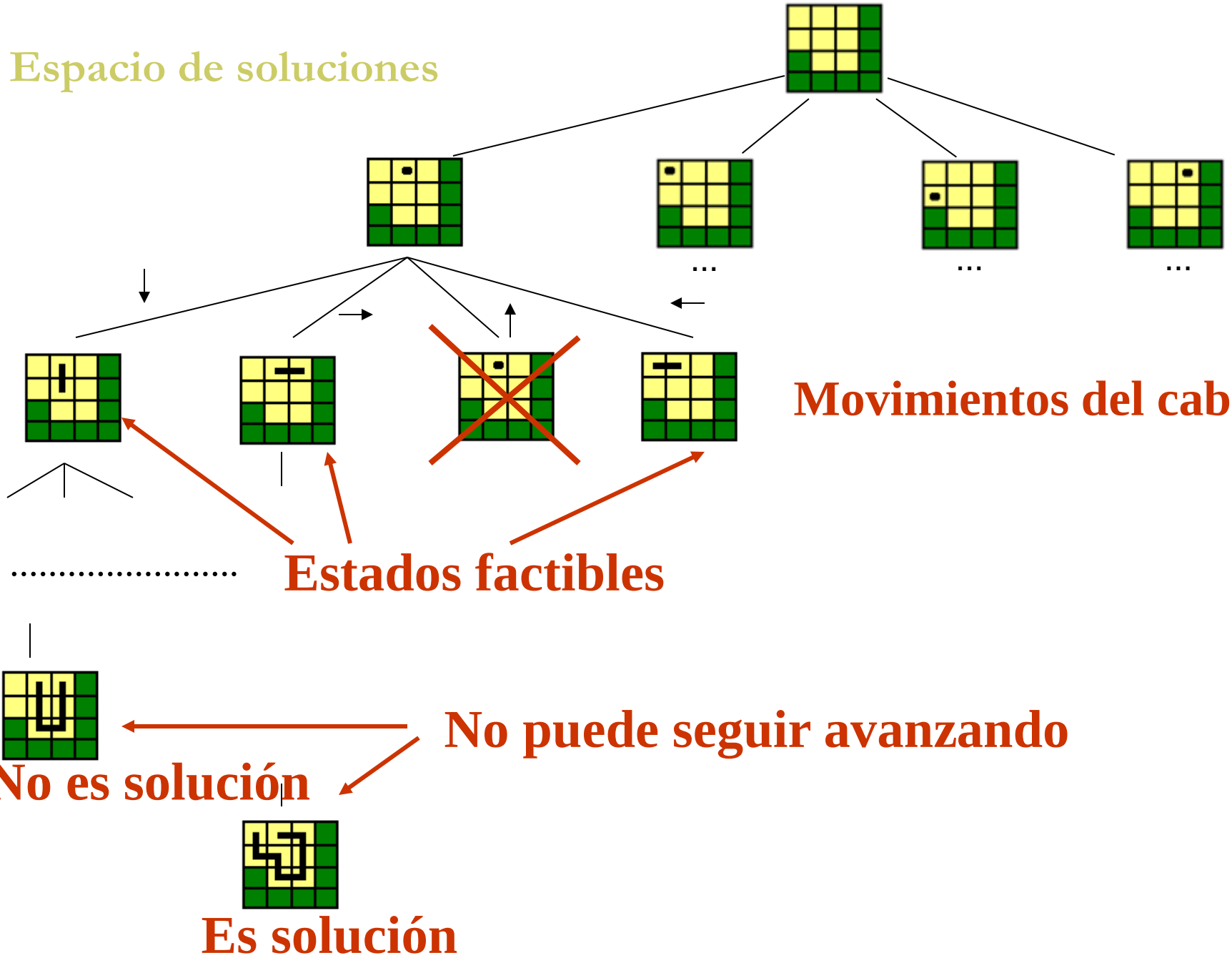


CARACTERIZACIÓN DEL PROBLEMA

- **Solución** → Vector $(pos_1, pos_2, pos_3, ...pos_n)$
  pos_i es una casilla de la matriz de m x m casillas

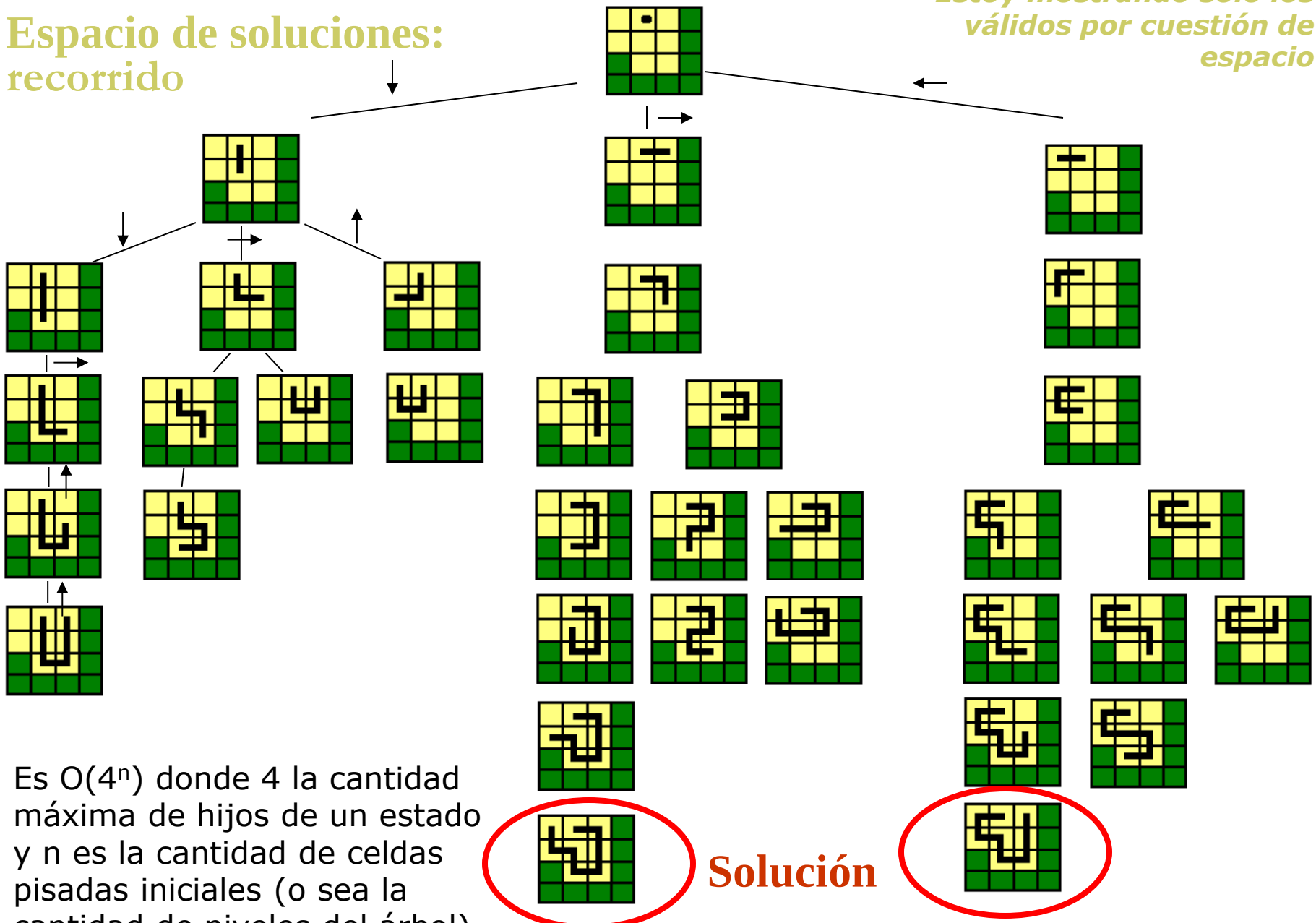
- **Restricciones Implícitas:** en el vector solución:
  - No hay dos posiciones iguales
  - pos_1 es una casilla del borde de la matriz
  - pos_i y pos_{i+1} son casillas adyacentes
  - pos_1 y pos_n son casillas adyacentes
  - todas las posiciones de casillas sin pasto deben estar en el vector

Espacio de soluciones



Espacio de soluciones: recorrido

Estoy mostrando sólo los válidos por cuestión de espacio



Es $O(4^n)$ donde 4 la cantidad máxima de hijos de un estado y n es la cantidad de celdas pisadas iniciales (o sea la cantidad de niveles del árbol).

Algoritmo para todas las soluciones

estado tendrá:
- Matriz ?
- lista casillas pisadas hasta el momento
- casilla actual

```
void BackAtila  (estado e,  int nroPisada) {

        if ( ! e.HayMovimientos() )
                { if (( nroPisada== e.cantPisadas() ) && (e.vecinaOrigen()))
                        imprimirSolucion(e);  }

        else
                {
                  movimiento movSgte;
                  int nrohijo=1;
                   while ( hijos(nrohijo, e, movSgte))
                        { if ( e.esFactible(movSgte) )
                              BackAtila( e.aplicarMov(movSgte), nroPisada+1);
                        nrohijo++;}
                }
  }
```

Backtracking

- Como vimos tendrá **costo exponencial** en el peor caso, lo cual puede llevar un problema a ser **intratable** computacionalmente (no termina).
- La estrategia de poda achica el espacio de búsqueda y por lo tanto el tiempo de ejecución, pero por lo general es difícil para una instancia particular de un problema predecir en cuánto de va a achicarse ese tiempo.

n	$\log_2 n$	n	n^2	2^n	n!
2	1	2	4	4	2
12	3,58	12	144	4096	4,7.10^8
25	4,64	25	625	3,3.10^7	1,55.10^25
50	5,64	50	2500	1,12.10^15	3,04.10^64
100	6,64	100	10000	1,26.10^30	9,33.10^157
200	7,64	200	40000	1,6.10^60	#¡NUM!

EVALUAR EL TAMAÑO DE LA ENTRADA
Y ESTUDIAR LA FACTIBILIDAD DE APLICAR LA TECNICA

POR LO GENERAL EN UN CASO REAL, LAS RESTRICCIONES SON MUCHO MAYORES QUE EN EL CASO TEORICO. UN PROBLEMA INTRATABLE PUEDE LLEGAR A SER TRATABLE Y ADEMAS LA POTENCIA COMPUTACIONAL VA EN AUMENTO….

Bibliografía

- Aho; Hopcroft; Ullman, “Estructuras de Datos y Algoritmos”.
- Sedgewich; Sahni; Rajasekaran, “Computer Algorithms”.