Exploración de grafos

- Grafos
- Recorridos sobre grafos
  - Búsqueda primero en profundidad
  - Búsqueda primero en anchura
- Backtracking ("vuelta atrás")
  - Descripción general
  - Espacio de soluciones
  - Implementación
  - Ejemplos
- Branch & Bound ("ramificación y poda")
  - Descripción general
  - Estrategias de ramificación
  - Implementación
  - Ejemplos

Backtracking

Supongamos que tenemos que tomar una serie de decisiones pero…

- No disponemos de suficiente información como para saber cuál elegir.
- Cada decisión nos lleva a un nuevo conjunto de decisiones.
- Alguna secuencia de decisiones (y puede que más de una) puede solucionar nuestro problema.

Necesitamos un método de búsqueda
que nos permita encontrar una solución…

Backtracking

Características de un problema resoluble con backtracking (o branch & bound)

- La solución puede expresarse como una n-tupla,

  $$(x_1, x_2, x_3, \ldots, x_n)$$

  donde cada x_i es seleccionado de un conjunto finito S_i.

- El problema se puede formular como la búsqueda de aquella tupla que optimiza (maximiza o minimiza) un determinado criterio $P(x_1, \ldots, x_n)$.

Backtracking

Resolución por fuerza bruta

- En principio, podemos solucionar nuestro problema probando todas las combinaciones $(x_1, x_2, x_3, \ldots, x_n)$.

Ejemplo

Generando todas las posibles combinaciones de n bits podemos resolver el problema de la mochila 0/1 para n objetos.

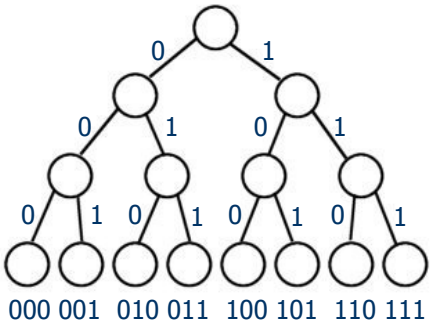
$$T(n) = 2\,T(n-1) + 1 \in O(2^n)$$

Backtracking

Resolución por fuerza bruta

$T(n) = 2\,T(n-1) + 1 \in O(2^n)$

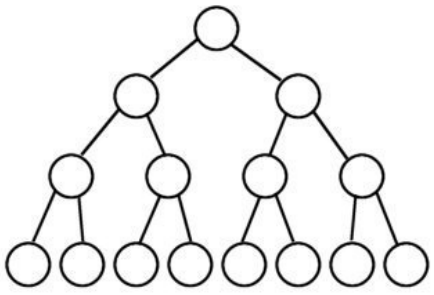
```
void combinaciones_binarias (vector<int> &V, int pos)
{
  if (pos==V.size())
     procesa_combinación(V);
  else {
     V[pos]=0;
     combinaciones_binarias (V,pos+1);
     V[pos]=1;
     combinaciones_binarias (V,pos+1);
  }
}
```



Backtracking

Resolución por fuerza bruta

- Implícitamente, se impone una estructura 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 del árbol, cuyas hojas corresponden a posibles soluciones del problema.



¿Se puede mejorar el proceso?

- **Backtracking**
- **Branch & Bound**

Backtracking

¿Se puede mejorar el proceso?

Sí, si eliminamos la necesidad de llegar hasta todas las hojas del árbol.

¿Cuándo?

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

¿Cómo?

Realizamos una vuelta atrás ("backtracking").

VENTAJA: Alcanzamos antes la solución.

Backtracking

Diferencias con otras técnicas

- En los algoritmos greedy, se construye la solución aprovechando la posibilidad de calcularla a trozos: un candidato nunca se descarta una vez elegido. Con backtracking, sin embargo, la elección de un candidato en una etapa no es irrevocable.

- El tipo de problemas en los que aplicaremos backtracking no se puede descomponer en subproblemas independientes, por lo que la técnica "divide y vencerás" no resulta aplicable.

Backtracking

- **Solución**: $(x_1,x_2,x_3,\ldots,x_n)$, $x_i \in S_i$.
- **Espacio de soluciones** de tamaño $\Pi|S_i|$
- **Solución parcial**: $(x_1,x_2,\ldots,x_k,?,?,\ldots,?)$, $x_i \in S_i$.
  Vector solución para el que aún no se han establecido todas sus componentes.
- **Función de poda/acotación**:
  Función que nos permite identificar cuándo una solución parcial no conduce a una solución del problema.

Backtracking

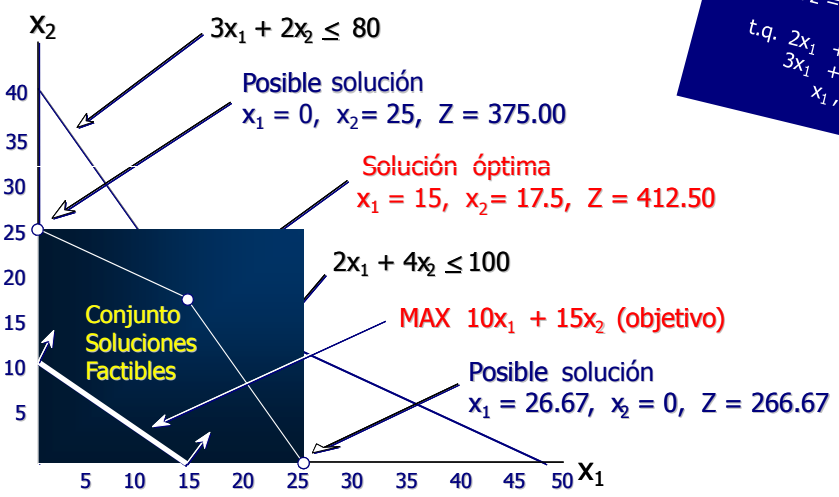
Restricciones asociadas a los problemas

- **Restricciones explícitas:** Restringen los posibles valores de las variables x_i (definen el espacio de soluciones del problema ΠS_i)

  p.ej. $x_i \geq 0 \Rightarrow S_i$ = {números reales no negativos}
  $x_i \in \{0,1\} \Rightarrow S_i = \{0,1\}$
  $l_i \leq x_i \leq u_i \Rightarrow S_i = \{a: l_i \leq a \leq u_i\}$

- **Restricciones implícitas**: Establecen relaciones entre las variables x_i (determinan las tuplas que satisfacen el criterio $P(x_1, \ldots,x_n)$: nos indican cuándo una solución parcial nos puede llevar a una solución).

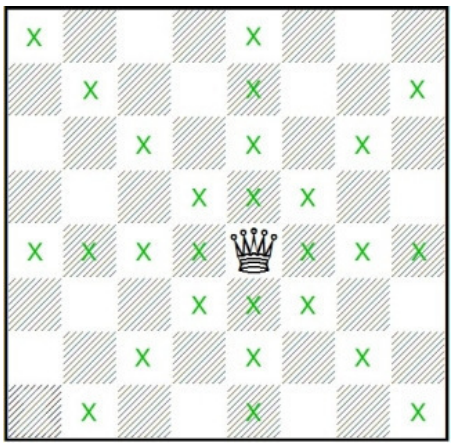
Backtracking



Backtracking
El problema de las 8 reinas

Problema combinatorio clásico:

Colocar ocho reinas en un tablero de ajedrez de modo que no haya dos que se ataquen; (que estén en la misma fila, columna o diagonal).



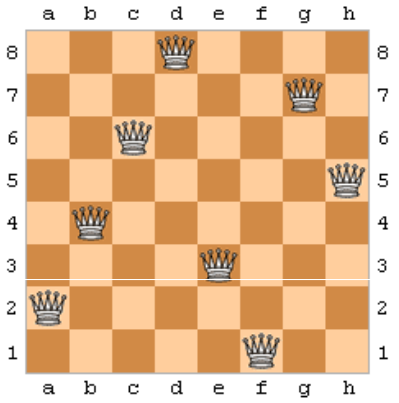
- Como cada reina debe estar en una fila diferente, sin pérdida de generalidad podemos suponer que la reina i se coloca en la fila i.
- Todas las soluciones para este problema pueden representarse como 8-tuplas $(x_1,...,x_8)$ en las que x_i indica la columna en la que se coloca la reina i.

Backtracking
El problema de las 8 reinas

Restricciones explícitas:

$S_i = \{1,2,3,4,5,6,7,8\}$, $1 \le i \le 8$

Espacio de soluciones:
Tamaño $|S_i|^8 = 8^8 = 2^{24} = 16M$



Restricciones implícitas:

- Ningún par (x_i, x_j) con $x_i=x_j$
  (todas las reinas deben estar en columnas diferentes).
- Ningún par (x_i, x_j) con $|j-i|=|x_j-x_i|$
  (todas las reinas deben estar en diagonales diferentes).

NOTA:
La primera de las restricciones implícitas implica que todas las soluciones son permutaciones de {1,2,3,4,5,6,7,8}, lo que reduce el espacio de soluciones de 8^8 tuplas a 8! = 40320.

Backtracking
El problema de la suma de subconjuntos

Dados
n+1 números positivos w_i, $1 \le i \le n$,
y un número M,
se trata de encontrar
todos los subconjuntos de números w_i cuya suma sea M.

Ejemplo:
n = 4
$(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$
M = 31
Subconjuntos buscados: (11, 13, 7) y (24, 7).

Backtracking
El problema de la suma de subconjuntos

Para representar la solución, podríamos usar un vector solución con los índices de los números w_i elegidos.

p.ej. Soluciones (1,2,4) y (3,4)

Las soluciones son, por tanto, k-tuplas $(x_1, x_2,...,x_k)$, $1 \leq k \leq n$, y pueden tener tamaños diferentes

Restricciones explícitas: $x_i \in \{j: j \text{ entero y } 1 \leq j \leq n\}$

Restricciones implícitas:

- Que no haya dos x_i iguales.
- Que la suma de los correspondientes w_i sea igual a M.
- Como (1,2,4) y (1,4,2) representan la misma solución, hay que imponer una restricción adicional: $x_i < x_{i+1}$, para $1 \leq i < n$

Backtracking
El problema de la suma de subconjuntos

Puede haber diferentes formas de formular un problema:

Otra formulación del problema de la suma de subconjuntos:

Cada subconjunto solución se representa por una n-tupla $(x_1,...,x_n)$ tal que $x_i \in \{0,1\}$, $1 \leq i < n$, con $x_i = 0$ si w_i no se elige y $x_i = 1$ si w_i se elige.

p.ej. Soluciones (1,1,0,1) y (0,0,1,1)

En esta formulación, todas las soluciones se expresan usando un tamaño de tupla fijo (n).

Para ambas formulaciones,
el espacio de soluciones es de tamaño 2^n.

Backtracking

Espacio de soluciones

- Los algoritmos backtracking determinan las soluciones del problema buscando sistemáticamente en el espacio de soluciones del problema.
- Esta búsqueda se puede representar en el **árbol de soluciones** asociado al conjunto de soluciones del problema.

Backtracking

Espacio de soluciones: Terminología

- **Estado del problema:** Cada uno de los nodos del árbol.
- **Estado solución**: Aquel estado (nodo) del problema (árbol) para el que el camino desde la raíz hasta el nodo representa una *n*-tupla en el espacio de soluciones del problema.
- **Estado respuesta:** Estado solución que representa una *n*-tupla del conjunto de soluciones del problema (esto es, una tupla que satisface todas las restricciones implícitas del problema).

Backtracking
El problema de las N reinas

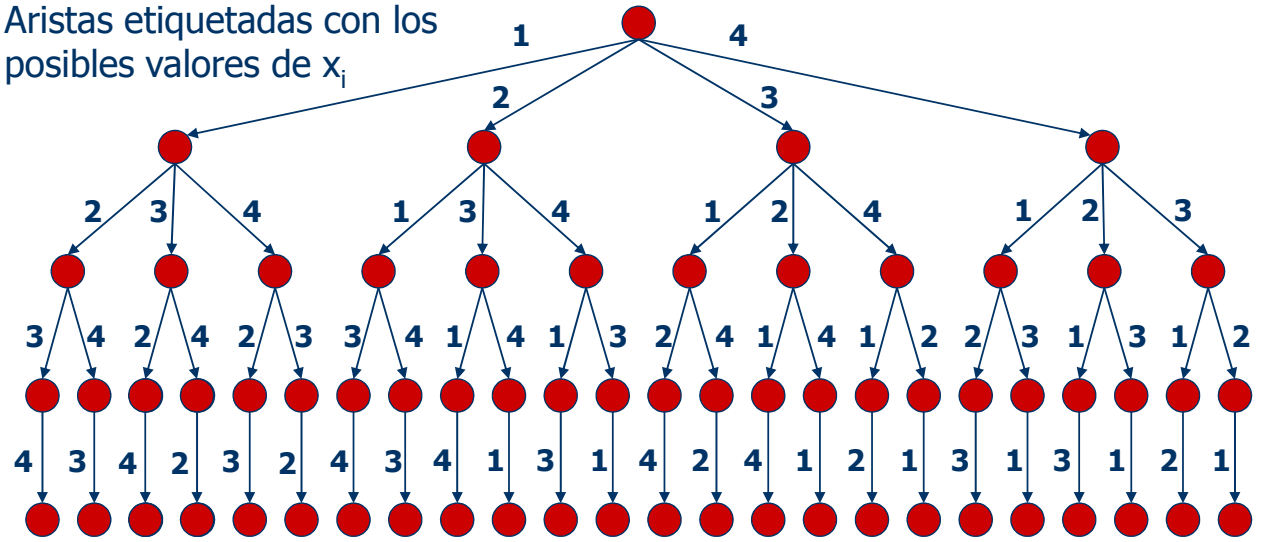
Espacio de soluciones

Generalización del problema de las 8 reinas:
Colocar N reinas en un tablero NxN
de modo se ataquen entre ellas.

El espacio de soluciones consiste en las N!
permutaciones de la N-tupla (1,2,...,N)

La generalización nos sirve, a efectos didácticos, para poder hablar del problema de las 4 reinas, que se puede representar en un árbol de permutaciones...

Backtracking
El problema de las N reinas



Las aristas desde los nodos del nivel i al i+1 están etiquetadas con los valores de x_i
p.ej. La rama más a la izquierda representa la solución $x_1=1$, $x_2=2$, $x_3=3$ y $x_4=4$.

El espacio de soluciones viene definido por todos los caminos desde el nodo raíz a un nodo hoja (4! permutaciones ⇒ 4! nodos hoja).

Backtracking
El problema de la suma de subconjuntos

Espacio de soluciones

Alternativa 1: Tuplas de distinto tamaño

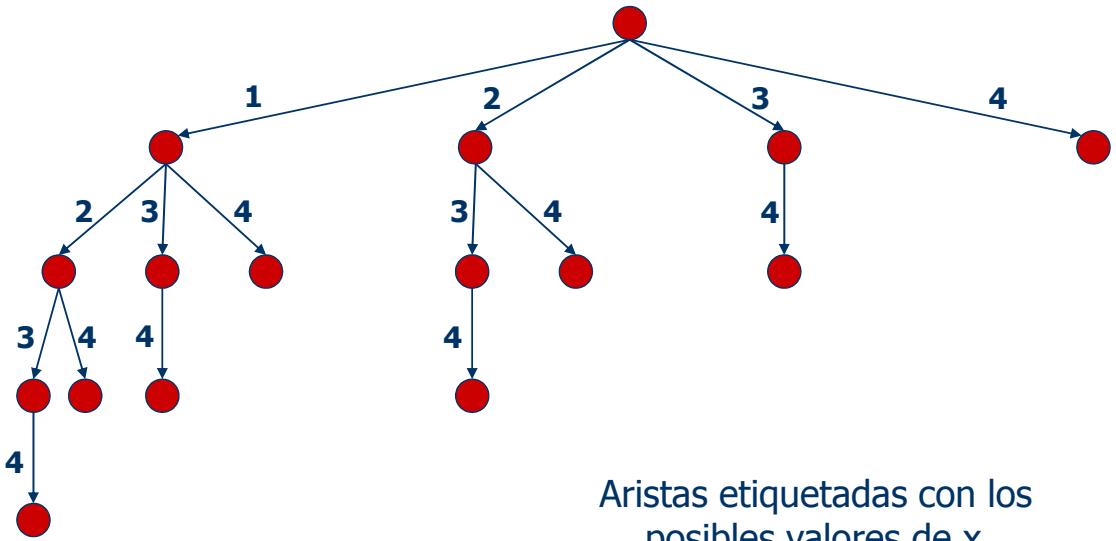
- Las aristas se etiquetan de modo que una arista desde el nivel i hasta el nivel i+1 representa un valor para x_i.
- Posibles soluciones: Tuplas que corresponden al camino desde la raíz hasta cada nodo del árbol (todos los nodos del árbol son estados solución).

  p.ej. {}, {1}, {1,2}, {1,2,3}, {1,2,3,4}
  {1,2,4}, {1,3,4}, {1,4}, {2}…
- El árbol resultante es un árbol combinatorio (o árbol de enumeración de subconjuntos)…

Backtracking
El problema de la suma de subconjuntos

Espacio de soluciones

Alternativa 1: Tuplas de distinto tamaño



Aristas etiquetadas con los posibles valores de x_i (siendo i el nivel del árbol)

Backtracking
El problema de la suma de subconjuntos

Espacio de soluciones

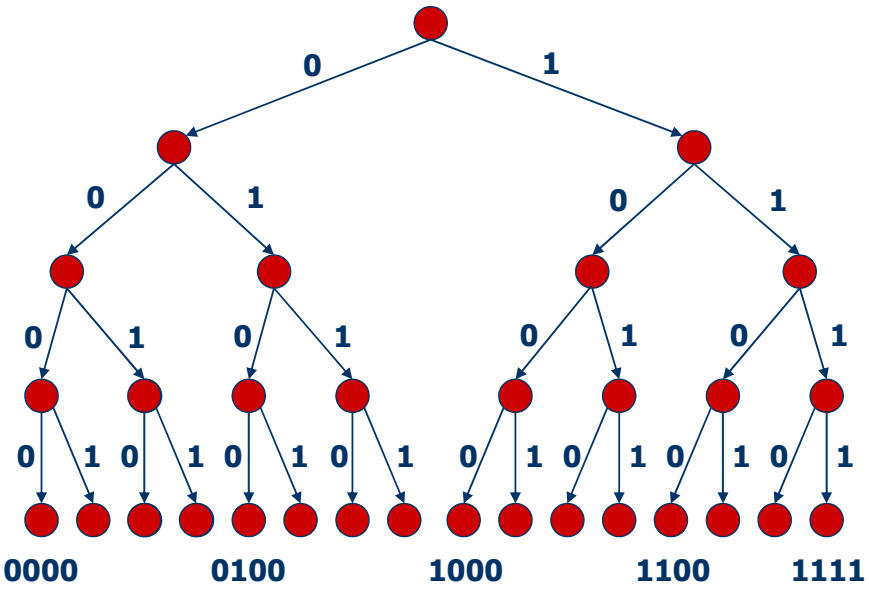
Alternativa 2: Tuplas del mismo tamaño

- Una arista del nivel i al i+1 se etiqueta con el valor de x_i (que, en este caso, sólo puede ser 0 ó 1)
- Cada camino desde la raíz hasta una hoja define una solución del espacio de soluciones del problema (sólo los nodos hoja son estados solución).
- En este caso, tenemos un árbol binario completo con 2^n nodos hoja que representan las 2^n posibles soluciones.

Backtracking
El problema de la suma de subconjuntos

Espacio de soluciones

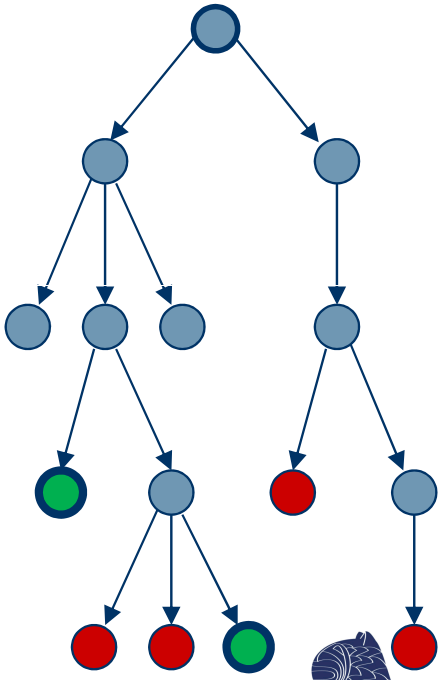
Alternativa 2: Tuplas del mismo tamaño



Backtracking

Generación de estados

Cuando se ha concebido el árbol de estados para un problema, podemos resolver el problema generando sistemáticamente sus **estados**, determinando cuáles de éstos son **estados solución** y, finalmente, comprobando qué estados solución son **estados respuesta**.



Backtracking

Generación de estados

- **Nodo vivo:** Estado del problema que ya ha sido generado pero del que aún no se han generado todos sus hijos.
- **Nodo muerto:** Estado del problema que ya ha sido generado y, o bien se ha podado, o bien ya se han generado todos sus descendientes.
- **E-nodo (nodo de expansión):** Nodo vivo del que actualmente se están generando sus descendientes.

Backtracking

Generación de estados

- Para generar todos los estados de un problema, comenzamos con un nodo raíz a partir del cual se generan otros nodos.
- Al ir generando estados del problema, mantenemos una lista de nodos vivos.
- Se usan funciones de acotación para "matar" nodos vivos sin tener que generar todos sus nodos hijos.

Backtracking

Generación de estados

Existen distintas formas de generar los estados de un problema (en función de cómo exploremos el árbol de estados).

p.ej. **backtracking** (en profundidad)

branch & bound (en anchura)

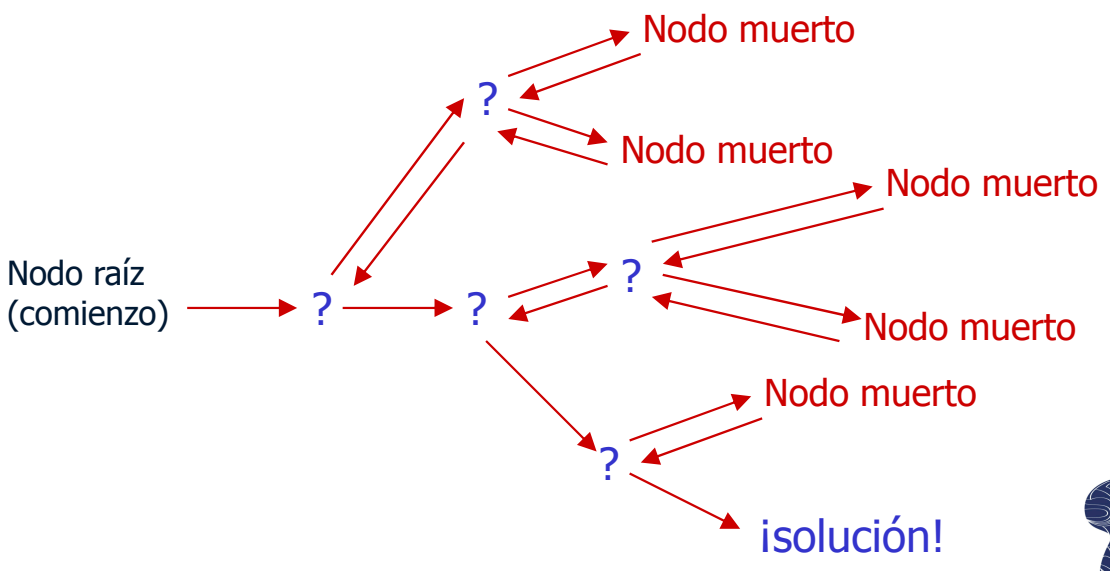
Backtracking

Generación de estados usando backtracking

- **Backtracking** corresponde a una generación primero en profundidad de los estados del problema.
- Tan pronto como un nuevo hijo C del E-nodo en curso R ha sido generado, este hijo se convierte en un nuevo E-nodo.
- R se convertirá de nuevo en E-nodo cuando el subárbol C haya sido explorado por completo.

Backtracking

Generación de estados usando backtracking



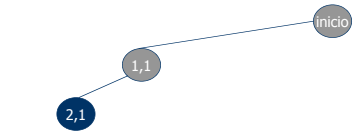
Backtracking
El problema de las 4 reinas

Generación de estados usando backtracking

OK! Adelante con la búsqueda…

Backtracking
El problema de las 4 reinas

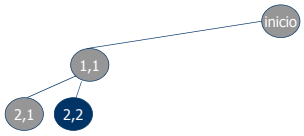
Generación de estados usando backtracking



No se cumplen las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

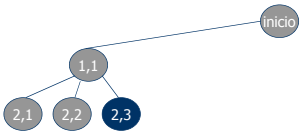
Generación de estados usando backtracking

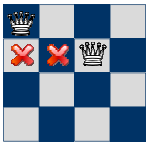


No se cumplen las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

Generación de estados usando backtracking

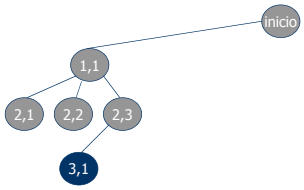




OK! Adelante con la búsqueda…

Backtracking
El problema de las 4 reinas

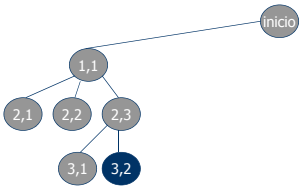
Generación de estados usando backtracking

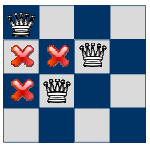


No se cumplen las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

Generación de estados usando backtracking

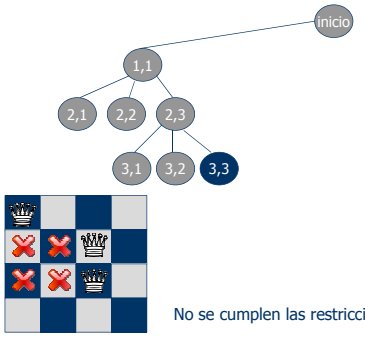




No se cumplen las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

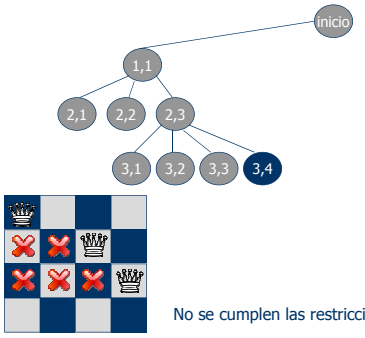
Generación de estados usando backtracking



No se cumplen las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

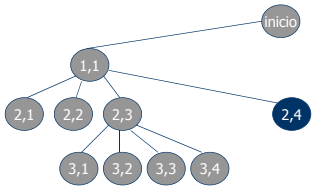
Generación de estados usando backtracking



No se cumplen las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

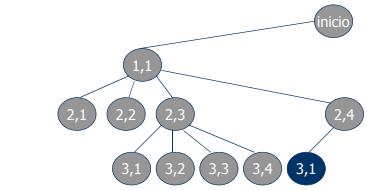
Generación de estados usando backtracking



OK! Adelante con la búsqueda…

Backtracking
El problema de las 4 reinas

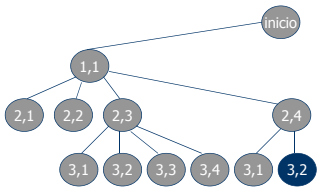
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

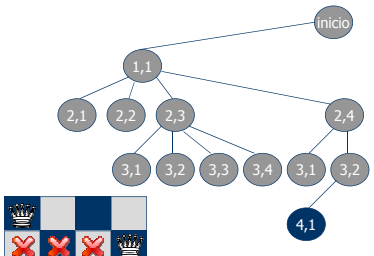
Generación de estados usando backtracking



OK! Adelante con la búsqueda…

Backtracking
El problema de las 4 reinas

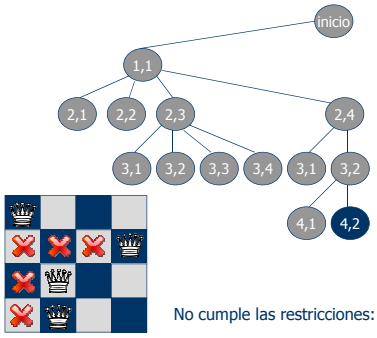
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

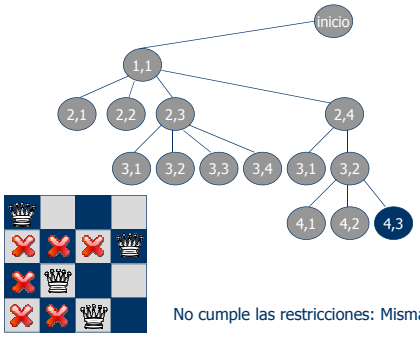
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

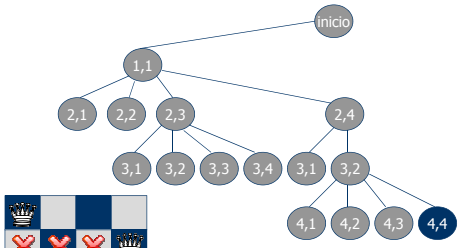
Generación de estados usando backtracking



No cumple las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

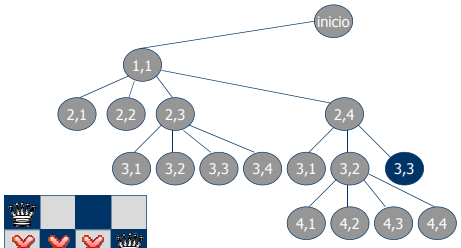
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

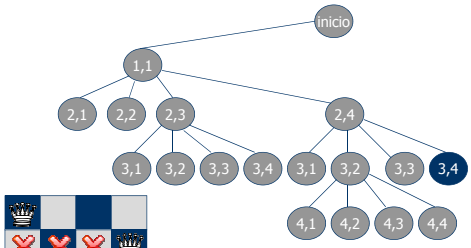
Generación de estados usando backtracking



No cumple las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

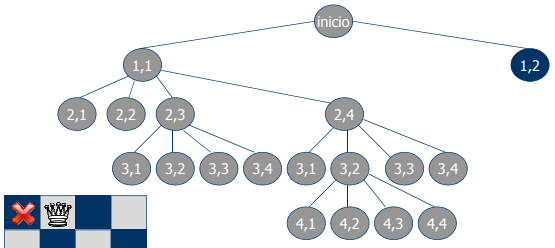
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

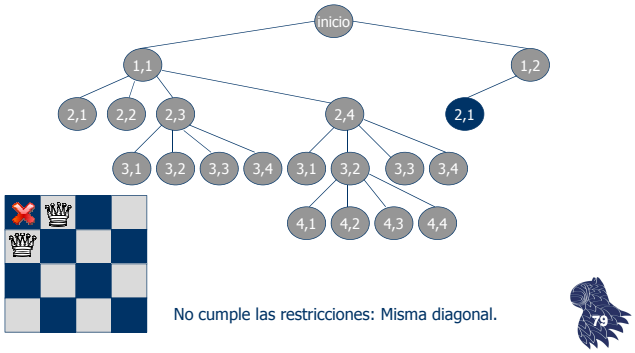
Generación de estados usando backtracking



OK! Adelante con la búsqueda…

Backtracking
El problema de las 4 reinas

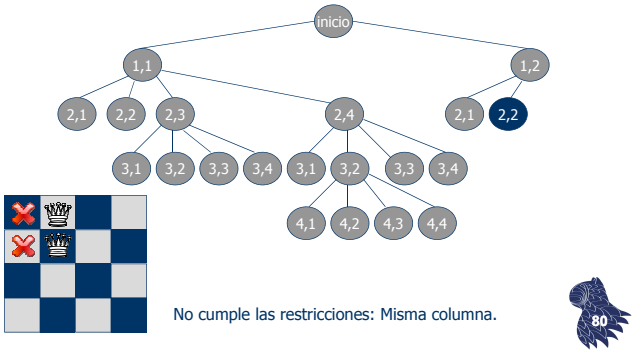
Generación de estados usando backtracking



No cumple las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

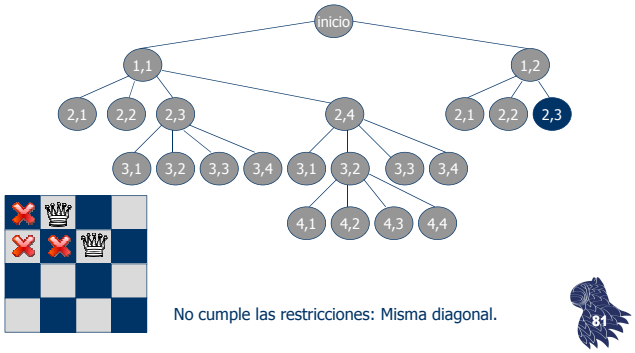
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

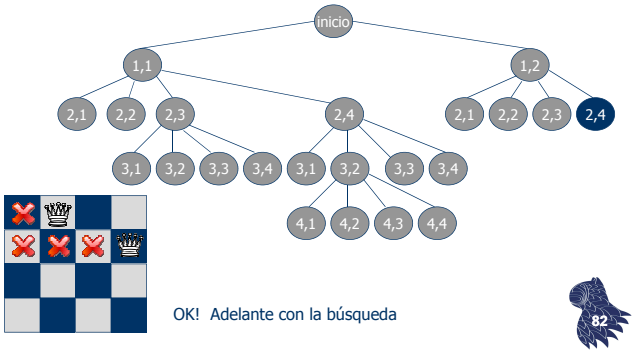
Generación de estados usando backtracking



No cumple las restricciones: Misma diagonal.

Backtracking
El problema de las 4 reinas

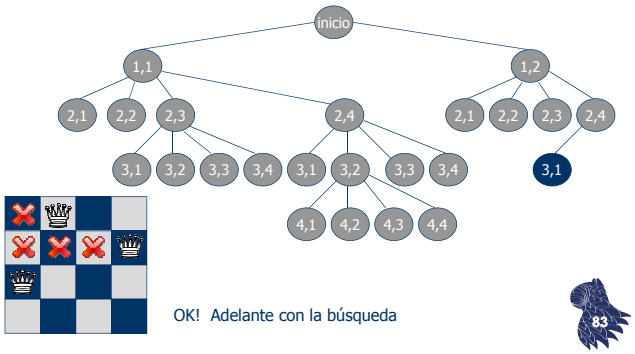
Generación de estados usando backtracking



OK! Adelante con la búsqueda

Backtracking
El problema de las 4 reinas

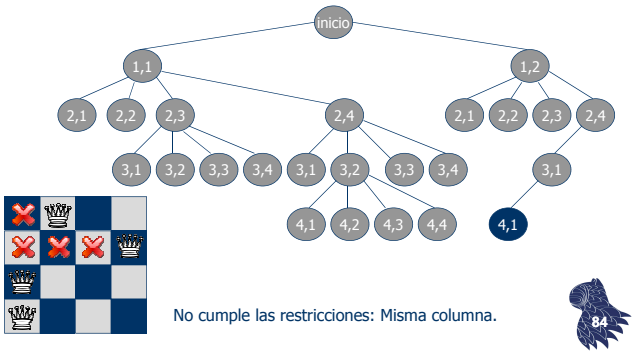
Generación de estados usando backtracking



OK! Adelante con la búsqueda

Backtracking
El problema de las 4 reinas

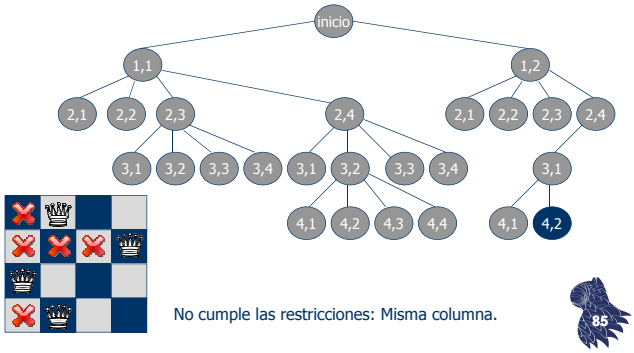
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

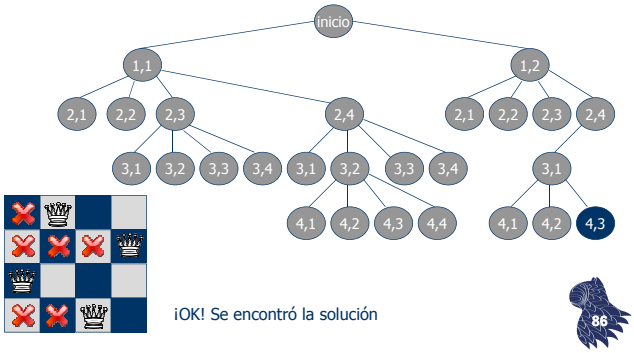
Generación de estados usando backtracking



No cumple las restricciones: Misma columna.

Backtracking
El problema de las 4 reinas

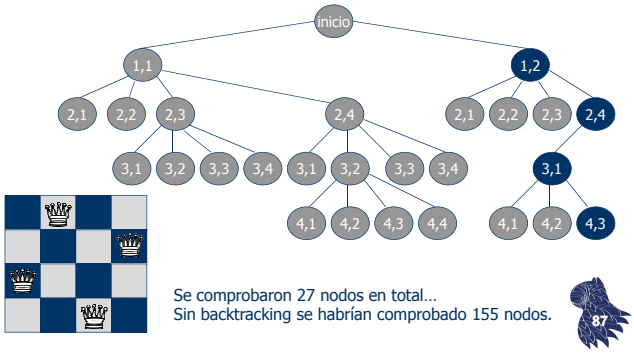
Generación de estados usando backtracking



¡OK! Se encontró la solución

Backtracking
El problema de las 4 reinas

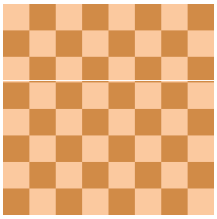
Generación de estados usando backtracking



Se comprobaron 27 nodos en total...
Sin backtracking se habrían comprobado 155 nodos.

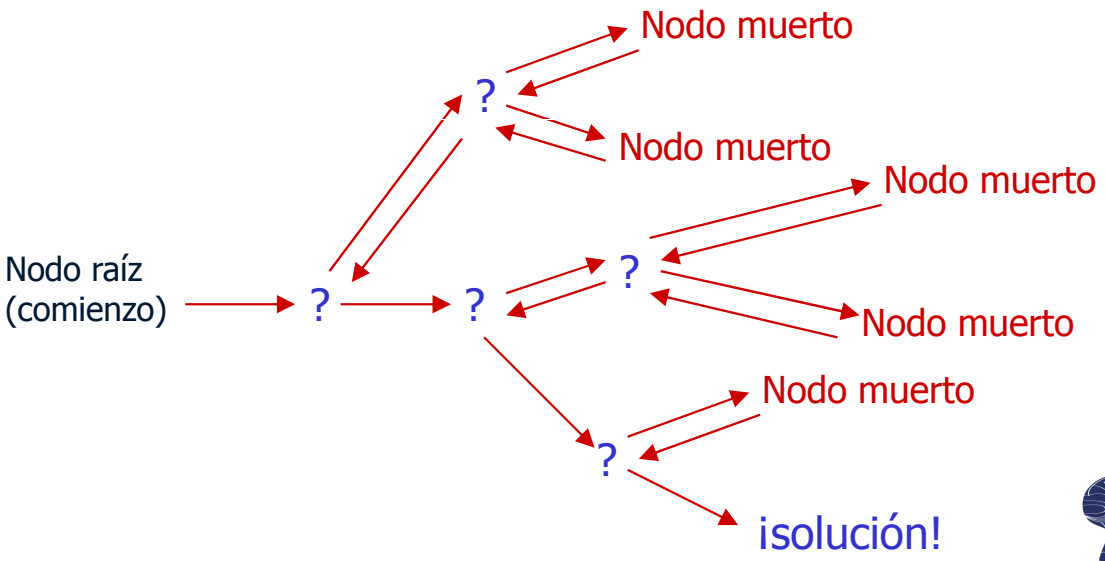
Backtracking
El problema de las 8 reinas

Generación de estados usando backtracking



Backtracking

Generación de estados usando backtracking



Backtracking

Generación de estados usando branch & bound

- **Branch & Bound** corresponde a una exploración en anchura del espacio de soluciones del problema.
- Un E-nodo se mantiene como E-nodo hasta que se convierte en un nodo muerto.
- Se usan funciones de acotación para detener la exploración de un subárbol: el diseño adecuado de **funciones de acotación** es fundamental para podar el espacio de búsqueda.

Backtracking

Implementación

Supondremos que hay que encontrar no sólo uno, sino todos los nodos respuesta:

$(x_1,x_2,\ldots,x_i)$

Camino desde la raíz hasta un nodo del árbol de estados.

$T(x_1,x_2,\ldots,x_{k-1})$

Conjunto de todos los posibles valores x_k tales que $(x_1,x_2,\ldots,x_{k-1},x_k)$ es también un camino válido hasta un estado del problema. Es decir, los posibles valores de x_k una vez que ya hemos escogido $(x_1,x_2,\ldots,x_{k-1})$.

Backtracking

Implementación

Funciones de acotación B_k

(expresadas como predicados):

$B_k(x_1,x_2,\ldots,x_k)$ es falso para un camino $(x_1,x_2,\ldots, x_k)$ si el camino no puede extenderse para alcanzar un nodo respuesta: nos indica si $(x_1,x_2,\ldots, x_k)$ satisface las restricciones implícitas del problema.

Los candidatos para la posición k del vector solución $X=(x_1,x_2,\ldots,x_n)$ son aquellos valores generados por $T(x_1,x_2,\ldots,x_{k-1})$ que satisfacen B_k.

Backtracking

Implementación

```
procedimiento backtrack (x[1..n])
{
  k=1;
  while (k>0) {
     if ( queda algún x[k] no probado tal que
          X[k]∈T(x[1..k-1]) && Bk(x[1..k]) ) {
        if (x[1..k] es un camino hasta un nodo respuesta)
           print ( x[1..k] ); // Mostrar solución x[1..k]
        k = k+1;
     } else {
        k = k-1;
     }
  }
}
```


Backtracking

Implementación recursiva

Llamada inicial: backtrack(x,1);

```
procedimiento backtrack ( x[1..n], k)
{
  foreach (x[k]∈T(x[1..k-1]) {
     if ( Bk(x[1..k]) ) {
        if (x[1..k] es un camino hasta un nodo respuesta)
           print ( x[1..k] ); // Mostrar solución x[1..k]
        backtrack (x, k+1);
     }
  }
}
```

Backtracking

Eficiencia de backtracking

La eficiencia de los algoritmos backtracking depende básicamente de cuatro factores:

1. el tiempo necesario para generar el siguiente x_k,
2. el número de valores x_k que satisfagan las restricciones explícitas,
3. el tiempo requerido por las funciones de acotación B_k y
4. el número de x_k que satisfagan las funciones B_k.

Backtracking

Eficiencia de backtracking: Sobre las funciones de acotación

- Las funciones de acotación se consideran buenas si reducen considerablemente el número de nodos que hay que explorar.
- Sin embargo, las buenas funciones de acotación suelen consumir mucho tiempo en su evaluación, por lo que hay que buscar un equilibrio entre el tiempo de evaluación de las funciones de acotación y la reducción del número de nodos generados.

Backtracking

Eficiencia de backtracking

De los 4 factores que determinan el tiempo de ejecución de un algoritmo backtracking, sólo el cuarto varía de un caso a otro: el **número de nodos** generados.

En el mejor caso, backtracking podría generar sólo O(n) nodos, aunque podría llegar a tener que explorar el árbol completo del espacio de estados del problema.

En el peor caso, si el número de nodos es 2^n o $n!$, el tiempo de ejecución del algoritmo backtracking será generalmente de orden $O(p(n)2^n)$ u $O(q(n)n!)$ respectivamente, con p y q polinomios en n.

Backtracking

Eficiencia de backtracking:
Estimación del número de nodos generados usando el método de Monte Carlo

Se genera un camino aleatorio en el árbol de estados:

- Dado un nodo X del nivel i del árbol perteneciente a ese camino aleatorio, las funciones de acotación se usan en el nodo X para determinar el número m_i de hijos del nodo que no se podan.
- El siguiente nodo del camino se obtiene seleccionando aleatoriamente uno de esos m_i hijos.
- La generación del camino termina cuando llegamos a un nodo hoja o a un nodo del que se podan todos sus hijos.

Backtracking

Eficiencia de backtracking:
Estimación del número de nodos generados usando el método de Monte Carlo

Tras generar unos cuantos caminos aleatorios sobre el árbol del espacio de soluciones, podremos usar los valores m_i obtenidos en cada camino aleatorio para estimar el numero total, m, de nodos del árbol que se generarán durante la ejecución del algoritmo.

Backtracking

Soluciones backtracking para distintos problemas

- El problema de las 8 reinas.
- El problema de la suma de subconjuntos.
- El problema del viajante de comercio.
- El problema del coloreo de un grafo.
- Laberintos.

Backtracking
El problema de las N reinas

Implementación

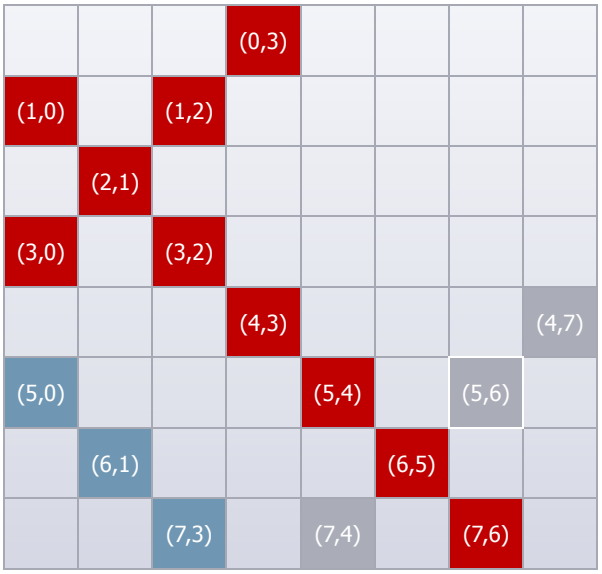
Tablero NxN en el que colocar N reinas que no se ataquen.

- **Solución**: $(x_0,x_2,x_3,\ldots,x_{n-1})$, donde x_i es la columna de la i-ésima fila en la que se coloca la reina i.
- **Restricciones implícitas**: $x_i \in \{0..n-1\}$.
- **Restricciones explícitas**: No puede haber dos reinas en la misma columna ni en la misma diagonal.

Backtracking
El problema de las N reinas

Implementación

- Distinta columna:
  Todos los x_i diferentes.
- Distinta diagonal:
  Las reinas (i,j) y (k,l) están en la misma diagonal si **i-j=k-l** o bien **i+j=k+l**, lo que se puede resumir en |j-l| = |k-i|.



En términos de los x_i, tendremos $|x_k-x_i|=|k-i|$.

Implementación

```
// Comprobar si la reina de la fila k está bien colocada
// (si no está en la misma columna ni en la misma diagonal
// que cualquiera de las reinas de las filas anteriores)
// Eficiencia: O(k-1).

bool comprobar (int reinas[], int n, 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
El problema de las N reinas

Implementación recursiva mostrando todas las soluciones

```
// Inicialmente, k=0 y reinas[i]=-1.

void NReinas (int reinas[], int n, int k)
{
  if (k==n) {            // Solución (no quedan reinas por colocar)
     print(reinas,n);
  } else {                  // Aún quedan reinas por colocar (k<n)
     for (reinas[k]=0; reinas[k]<n; reinas[k]++)
         if (comprobar(reinas,n,k))
            NReinas (reinas, n, k+1);
  }
}
```

Backtracking
El problema de las N reinas

Implementación recursiva mostrando sólo la primera solución

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

Backtracking
El problema de las N reinas

Implementación iterativa mostrando todas las soluciones

```
void NReinas (int reinas[], int n)
{
  int k=0;                                  // Fila actual = k
  for (int i=0; i<n; i++) reinas[i]=-1;  // Configuración inicial

  while (k>=0) {
        reinas[k]++;  // Colocar reina k en la siguiente columna…
        while ((reinas[k]<n) && !comprobar(reinas,n,k))
              reinas[k]++;
        if (k<n) {                              // Reina colocada
           if (k==n-1) { print(reinas,n);    // Solución
           } else { k++; reinas[k] = -1; }  // Siguiente reina
        } else { k--; }
  }
}
```

Backtracking
El problema de la suma de subconjuntos

Implementación

Tenemos n números positivos distintos (usualmente llamados pesos) y queremos encontrar todas las combinaciones de estos números que sumen M.

Solución backtracking usando un tamaño de tupla fijo:

- Solución: El elemento X[i] del vector solución es 1 ó 0 dependiendo de si el peso W[i] está incluido o no.
- Funciones de acotación:
  X[1..k] no puede conducir a una solución si no se verifica la siguiente condición: $\Sigma_{1..k} W[i]X[i] + \Sigma_{k+1..n} W[i] \geq M$.

Backtracking
El problema de la suma de subconjuntos

Implementación

La función de acotación anterior puede "fortalecerse" si consideramos los W[i] en orden creciente: En este caso, X[1..k] no puede llevar a una solución si

$$\Sigma_{1..k} W[i]X[i] + W[k+1] > M.$$

Por tanto, usaremos la función de acotación $B_k(X[1..k])$:

$B_k(X[1..k])$=true si, y sólo si,

$$\Sigma_{1..k} W[i]X[i] + \Sigma_{k+1..n} W[i] \geq M$$

y $$\Sigma_{1..k} W[i]X[i] + W[k+1] \leq M.$$

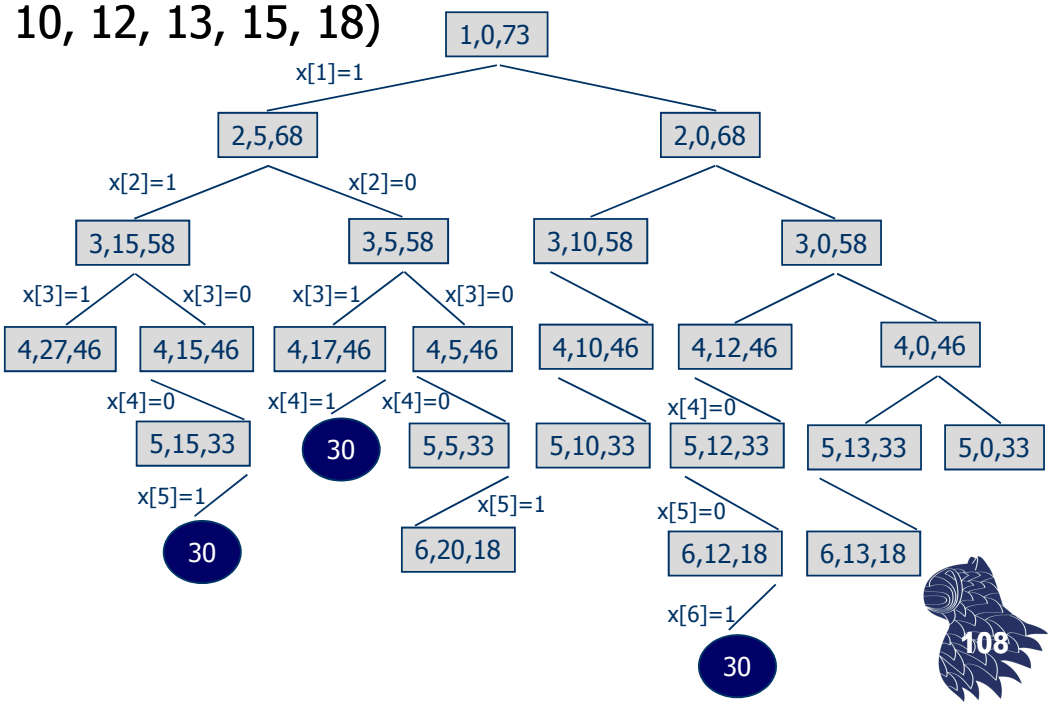
Implementación

```
// X[1..k-1] ya determinados, W[j] en orden creciente.
// Se supone que W[1]<=M y que Σ1..nW[i]>=M.

void SumaSubconjuntos (k,s,r)    // s = Σ1..k-1W[i]X[i]
{                                // r = Σk..nW[j]
  X[k] = 1;  // Nótese que s+W[k] <= M, ya que Bk-1 == true

  if (s + W[k] == M ) { // Solución
     print (X[1..k]);
  } else {
     if (s + W[k] + W[k+1] <= M)
        SumaSubconjuntos (k+1, s+W[k], r-W[k]);
     if ( (s + r - W[k] >= M) && (s + W[k+1] <= M) ) {
        X[k] = 0; SumaSubconjuntos (k+1, s, r-W[k]);
     }
  }
}
```


Implementación

W = (5, 10, 12, 13, 15, 18)

M = 30



Backtracking
El problema del viajante de comercio

Implementación

Algoritmo backtracking que determine todos los ciclos hamiltonianos de un grafo conexo G=(V,E) con n vértices

- Un ciclo hamiltoniano es un camino que recorre los n vértices del grafo, visitando una sola vez cada vértice, hasta finalizar en el vértice de partida.
- Vector solución (x_1, ..., x_n) tal que x_i representa el i-ésimo vértice visitado en el ciclo propuesto.
- Todo lo que se necesita es determinar el conjunto de posibles vértices x_k una vez elegidos x_1, ..., x_{k-1}.

Backtracking
El problema del viajante de comercio

Implementación

- Si k=1, x[1] puede ser cualquiera de los n vértices, aunque para evitar encontrar el mismo ciclo n veces, exigiremos que x[1] = 1.
- Si 1 < k < n, entonces x[k] puede ser cualquier vértice v que sea distinto de x[1], x[2], ..., x[k-1] y que esté conectado mediante una arista a x[k-1].
- x[n] sólo puede ser el único vértice restante y debe estar conectado tanto a x[n-1] como a x[1].

Backtracking
El problema del viajante de comercio

Implementación

```
// La función siguienteValor(k) devuelve el siguiente vértice
// válido para la posición k (o 0 si no queda ninguno)

int siguienteValor (k)
{
  // x[1..k-1] almacena los vértices ya asignados y G[][] es la
  // matriz de adyacencia que representa el grafo de N vértices
  do {
        x[k]++;
        if (  G[x[k-1]][x[k]]]
           && x[k]∉x[1..k-1]
           && ( k<N || (k==N && G[x[k]][x[1]]) ) )
              return x[k];
  } while (x[k]<N);
  x[k]=0; return 0;
}
```


Backtracking
El problema del viajante de comercio

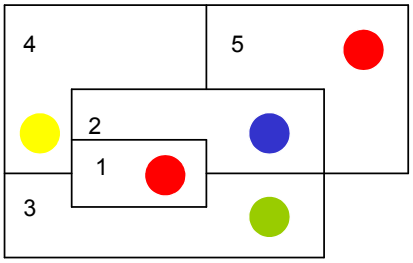
Implementación

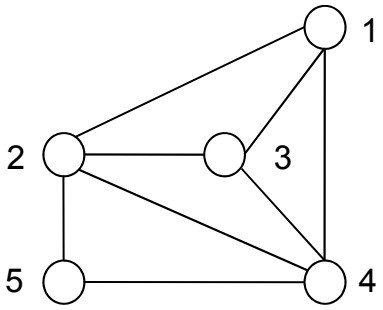
```
// x[1..k-1] almacena los vértices ya asignados

void hamiltoniano (k)
{
  if (k==N) {
     print(x[1..N]);  // Solución
  } else {
     do {
        x[k] = siguienteValor(k);
        if (x[k]!=0)
           hamiltoniano(k+1);
     } while (x[k]!=0);
  }
}
```

Backtracking
El problema del coloreo de un grafo

Problema de la m-colorabilidad:

Sea G un grafo y m 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 sólo se usen m colores.





NOTA: El menor número m con el que el grafo G puede colorearse se denomina número cromático del grafo.

Backtracking
El problema del coloreo de un grafo

Implementación

- Vector solución (x_1, ..., x_n) tal que x_i representa el color del vértice i, con $x_i \in \{1..m\}$.
- El espacio de estados subyacente es un árbol de grado m y altura n+1, en el que cada nodo del nivel i tiene m hijos correspondientes a las m posibles asignaciones para x_i y en el que los nodos del nivel n+1 son nodos hoja.

Implementación

```
// x[1..k-1] almacena los colores asignados a los vértices 1..k-1
// Inicialmente, x[i] = 0

void colorea (m, k)
{
  if (k==N) {

     print(x[1..N]);  // Solución con m colores

  } else {

     do {
        x[k] = siguienteValor(m,k);
        if (x[k]!=0)
           colorea(m, k+1);
    } while (x[k]!=0);
  }
}
```

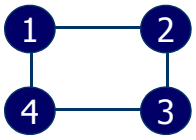

Implementación

```
// La función siguienteValor(k) devuelve el siguiente color
// válido para el vértice k (o 0 si no queda ninguno)
int siguienteValor (m,k)
{
  x[k]++;                                    // Siguiente color…
  while (x[k]<=m) {
        conflictos = 0;
        for (i=0; i<k; i++)
            if ( G[i][k] && x[i]==x[k] )     // Vértice adyacente
               conflictos++;                 // del mismo color.
        if (conflictos==0)
           return x[k];
        else
           x[k]++;
  }
  return 0;  // Ya se han probado los m colores permitidos.
}
```

Eficiencia del algoritmo

- Nodos internos del árbol de estados: $\Sigma_{i=0..n-1}\ m^i$
- En cada nodo interno, se llama a la función siguienteValor(m,k), que es de orden O(nm).
- El tiempo total de ejecución del algoritmo está acotado por $\Sigma_{i=1..n}\ nm^i = n(m^{n+1}-1)/(m-1) \in O(nm^n)$.
- Por tanto, el algoritmo es $O(nm^n)$.

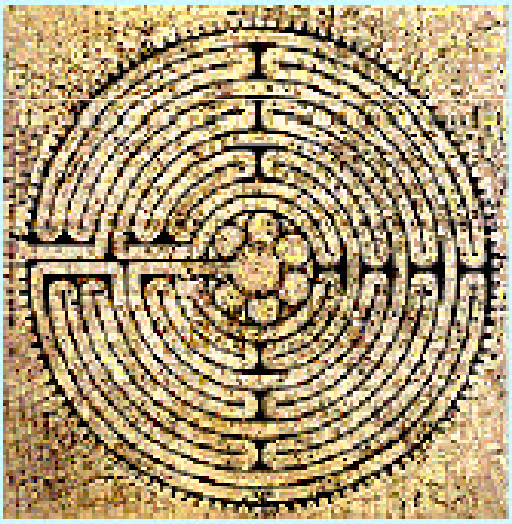
Ejemplo (m=3)



Backtracking
Laberintos

Problema

Buscar la salida de un laberinto…



Un laberinto puede modelarse como un grafo:

En cada nodo (cruce) hay que tomar una decisión que nos conduce a otros nodos (cruces).

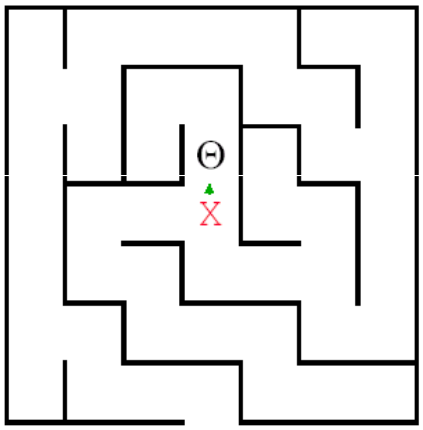
Backtracking
Laberintos

Solución con backtracking

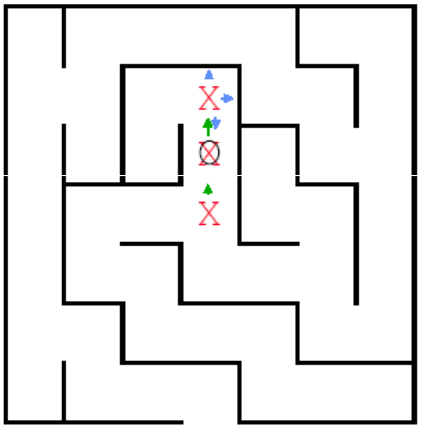
```
Si la posición actual está fuera del laberinto,
   devolver TRUE para indicar que hemos encontrado una solución.
Si la posición actual está marcada,
   devolver FALSE para indicar que ya habíamos estado aquí.
Marcar la posición actual.

Para cada una de las 4 direcciones posibles (N,S,E,W) {
   Si (en la dirección elegida no chocamos contra un muro) {
      Moverse un paso en la dirección indicada
      Intentar resolver el laberinto desde ahí recursivamente.
      Si la llamada recursiva nos permite salir del laberinto,
         devolver TRUE para indicar que ya hemos terminado.
   }
}

Quitar la marca de la posición actual.
Devolver FALSE para indicar que ninguna de las 4 direcciones
   nos permitió llegar a una solución desde la posición actual.
```

Backtracking
Laberintos

Ejemplo



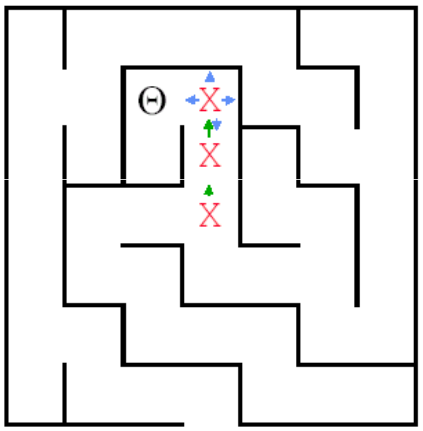
Dirección Norte…



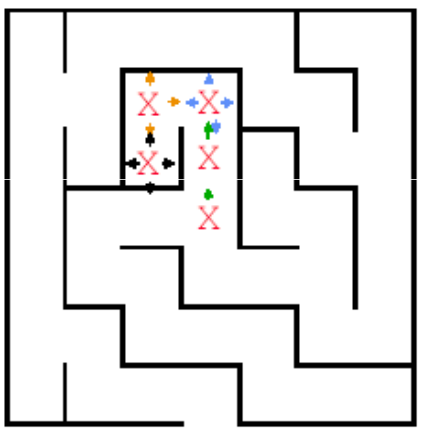
No podemos ir al Sur.

Backtracking
Laberintos

Ejemplo



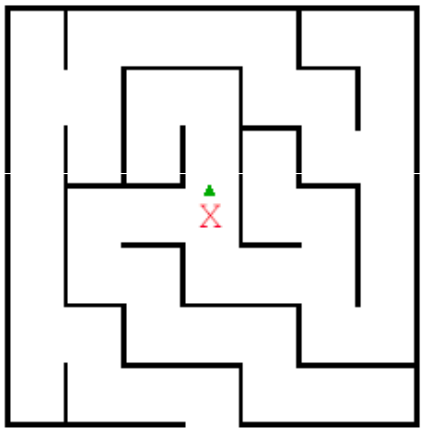
Giramos al Oeste…



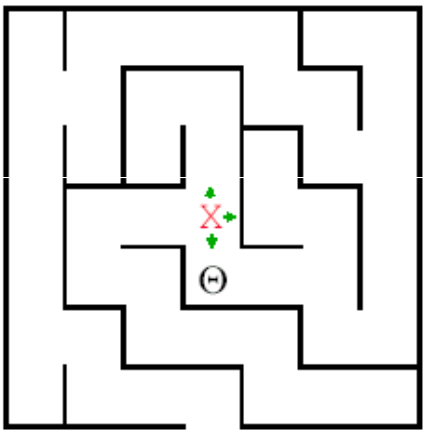
… momento de hacer backtracking

Backtracking
Laberintos

Ejemplo



Volvemos al punto de partida…



… pero ahora cogemos rumbo al Sur…

Backtracking
Laberintos

Ejemplo