Algorítmica

Tema 2: Búsqueda con retroceso

Grado en Ingeniería Informática – ETSINF

Resultados de aprendizaje

- Conocer los conceptos básicos de la estrategia algorítmica "búsqueda con retroceso" ("vuelta atrás", o *backtracking*).
- Estudiar problemas que se resuelven con búsqueda por retroceso:
  - El problema de las *n* reinas
  - La suma del subconjunto
  - Existencia ciclo hamiltoniano

El material de estos apuntes...

...ha sido extraído del libro de apuntes de algorítmica de:

- Andrés Marzal
- María José Castro
- Pablo Aibar

Introducción

Búsqueda con retroceso: Para qué se usa

- Permite resolver problemas en los que se plantea la búsqueda de una solución factible (no necesariamente la óptima) de entre un conjunto de soluciones $\Rightarrow$ No es habitual usar búqueda con retroceso para resolver problemas de optimización.
- No obstante, también puede emplearse en problemas de optimización a partir de variantes que permiten obtener todas las soluciones factibles (aunque normalmente de forma poco eficiente).

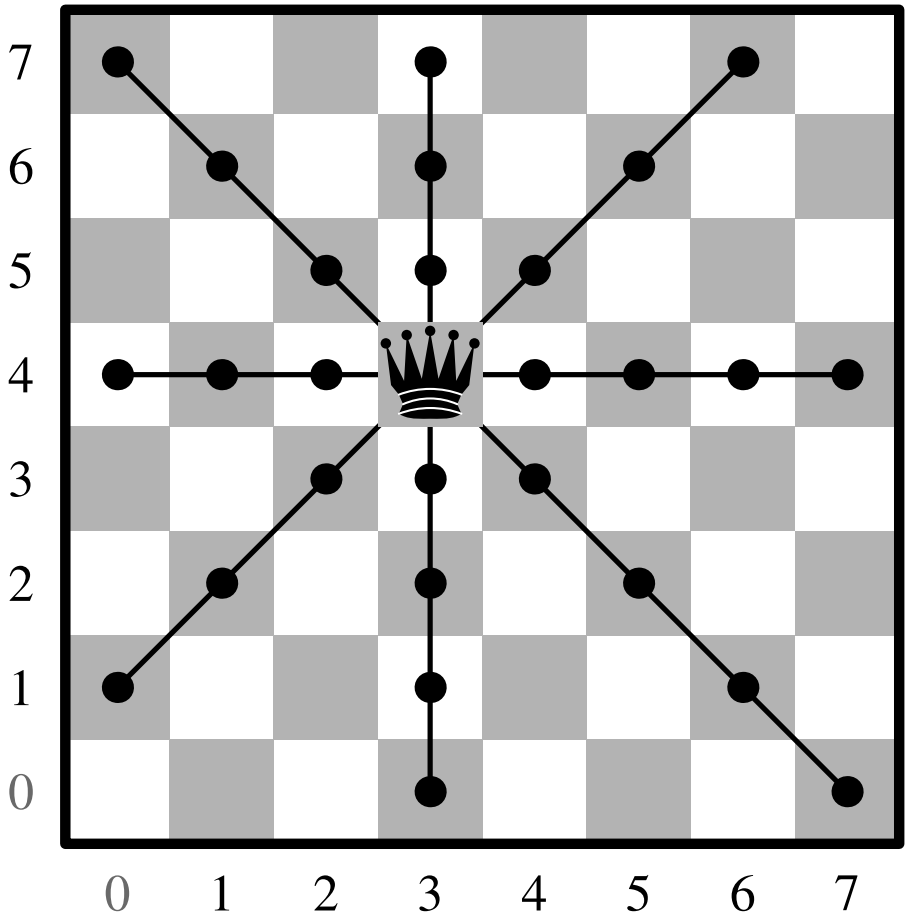
Búsqueda con retroceso: En qué consiste

- **Fuerza bruta** vs. **búsqueda con retroceso**:
  - Método de la **fuerza bruta**: exploración sistemática de todas las soluciones hasta dar con una factible.
  - **Búsqueda con retroceso**: también realiza una exploración sobre el conjunto de soluciones pero, a diferencia de la fuerza bruta, puede dejar de explorar grupos de soluciones para las que ”se sabe” que no incluyen ninguna factible.
- Técnica típicamente **recursiva** (admite transformación iterativa).
- Las soluciones se expresan como tuplas (se construyen **incrementalmente**).
- (Enseguida veremos un ejemplo.)
- No garantiza algoritmos con coste polinómico.
- Un primer paso para entender otras técnicas de búsqueda (en particular, “ramificación y poda”).

El problema de las *n* reinas

Descripción del problema

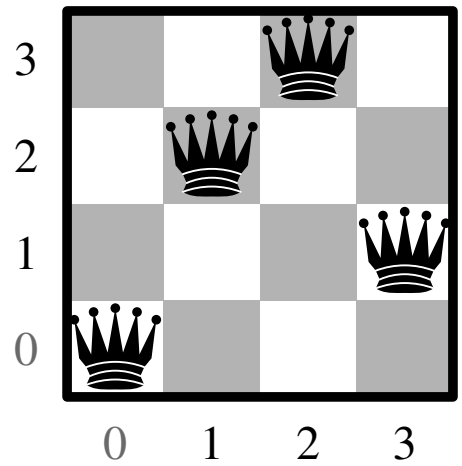
Deseamos disponer n reinas en un tablero de ajedrez de $n \times n$ escaques de modo que ninguna de ellas amenace a otra.



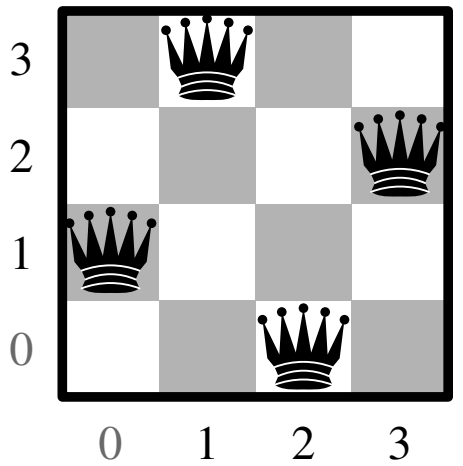
Descripción del problema

Solución: disposición de n reinas en un tablero de $n \times n$, se amenacen o no.
Solución factible: solución en la que las reinas no se amenazan.

Dos configuraciones (soluciones) de 4 reinas en un tablero de 4×4:



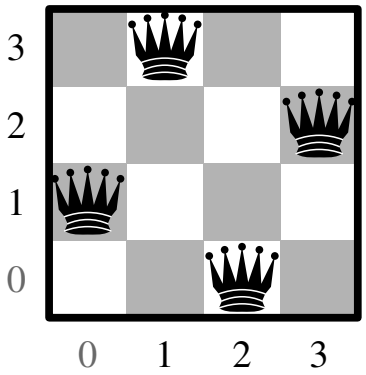
Solución (no factible)



Solución (factible)

Modelado del problema

- Cada pieza se pone en una casilla: un par (i, j) (columna, fila).
- Una configuración de n reinas (una solución) es una **secuencia (una tupla) de n pares**: $((i_0, j_0), (i_1, j_1), \ldots, (i_{n-1}, j_{n-1}))$. Cada uno de los elementos de la configuración es un par de enteros que indican la columna y la fila de una reina (cada i_k y j_k está comprendido entre 0 y $n - 1$):

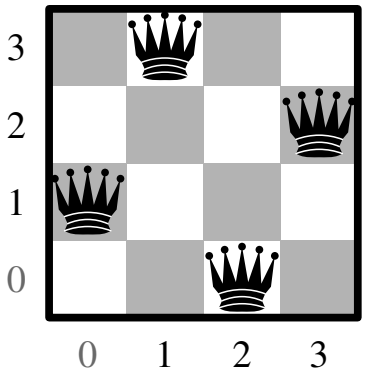


$$((0, 1), (1, 3), (2, 0), (3, 2))$$

Redundante: en cada columna sólo puede haber una reina (o se amenazarían).

Modelado del problema

Basta con una secuencia de filas $(j_0, j_1, \ldots, j_{n-1})$ para representar la configuración (la posición i-ésima de la secuencia, j_i, indica que la reina i-ésima se coloca en la columna i, fila j_i). El mismo ejemplo:



$$(1, 3, 0, 2)$$
$$((0, 1), (1, 3), (2, 0), (3, 2))$$

Por tanto, el **conjunto X' de soluciones** (factibles o no) del problema se puede definir formalmente como:

$$X' = [0..n-1]^n$$

Modelado del problema

Conjunto X' de soluciones (factibles o no) del problema:

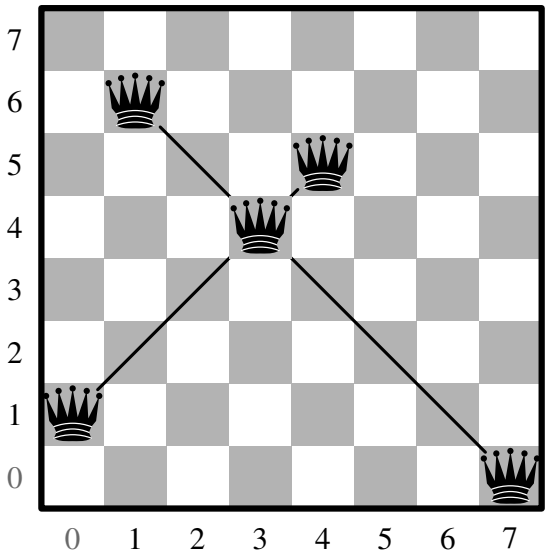
$$X' = [0..n-1]^n$$

Ahora debemos definir el conjunto X de configuraciones que cumplen todos los requisitos: las **soluciones factibles**. Para que una solución de X' sea factible no se deben amenazar ningún par de reinas:

- en la tupla no se pueden dar elementos j_k repetidos (compartirían fila) y ...
- no pueden amenazarse en las diagonales. ¿Cómo sé que no se amenazan en las diagonales?

Modelado del problema

Amenazas en las diagonales: si me dan una configuración $(j_0, j_1, \ldots, j_{n-1})$ en la que todo par de valores j_k y j_l $(0 \leq k < l < n)$ son distintos,



$$
\begin{aligned}
l - k &= |j_l - j_k| \\
3 - 0 &= |4 - 1| \\
3 - 1 &= |4 - 6| \\
4 - 3 &= |5 - 4| \\
7 - 3 &= |0 - 4|
\end{aligned}
$$

Conjunto de soluciones factibles:

$$X = \{(j_0, j_1, \ldots, j_{n-1}) \in [0..n-1]^n | j_k \neq j_l \wedge |j_l - j_k| \neq l - k, 0 \leq k < l < n\}$$

Una posible solución: fuerza bruta

- Genero toda posible permutación de $[0..n-1]$ . . .
- . . . y pruebo, una a una, si viola alguna restricción (las de las diagonales).
- Paro cuando encuentro una que satisface todas las restricciones.

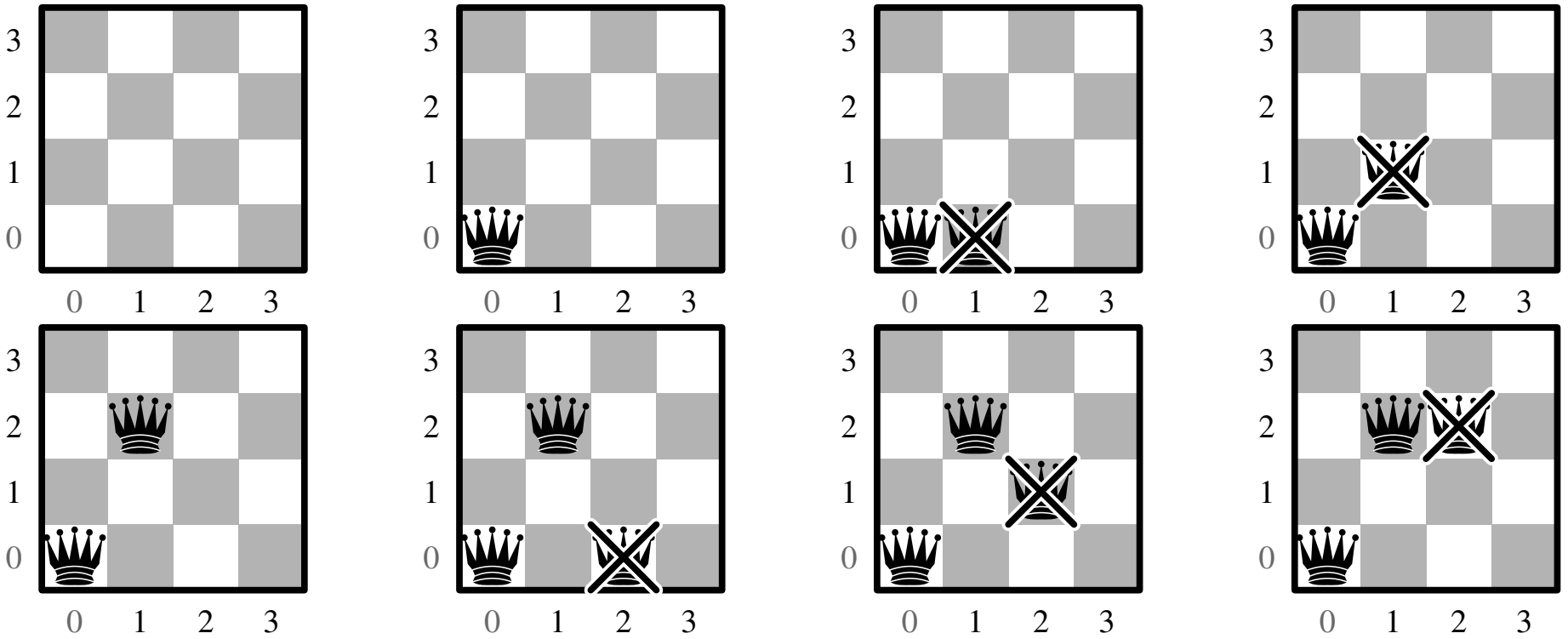
Tiempo **exponencial**: $O(n!)$.

Búqueda con retroceso

- Se construye la tupla elemento a elemento.
- Se descartan las configuraciones no válidas.
- Cuando no podamos avanzar $\Rightarrow$ retrocedemos (en orden inverso) hasta poder avanzar de nuevo (de forma diferente).
- Acabamos al encontrar una solución factible.

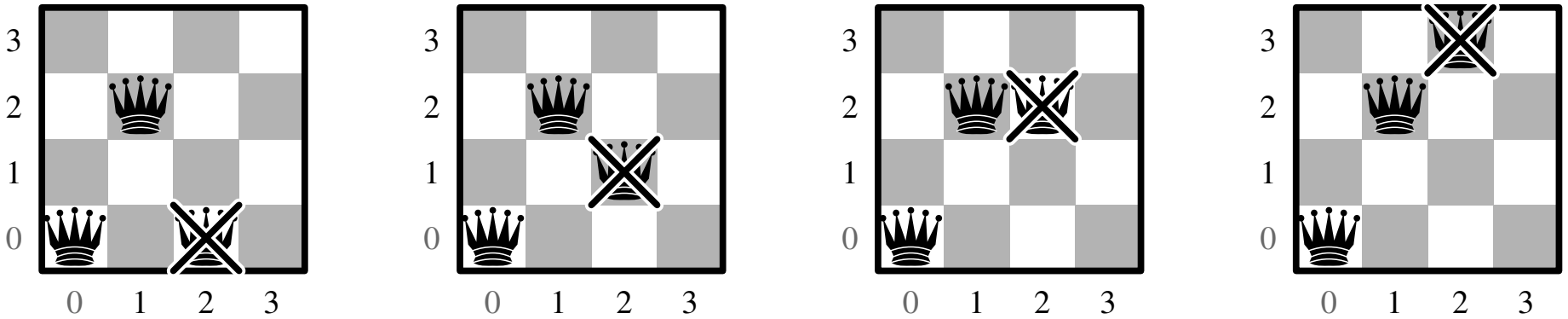
Reina a reina

Pongamos reina a reina… y pongamos reinas sólo donde podemos hacerlo:



¿Ya está?

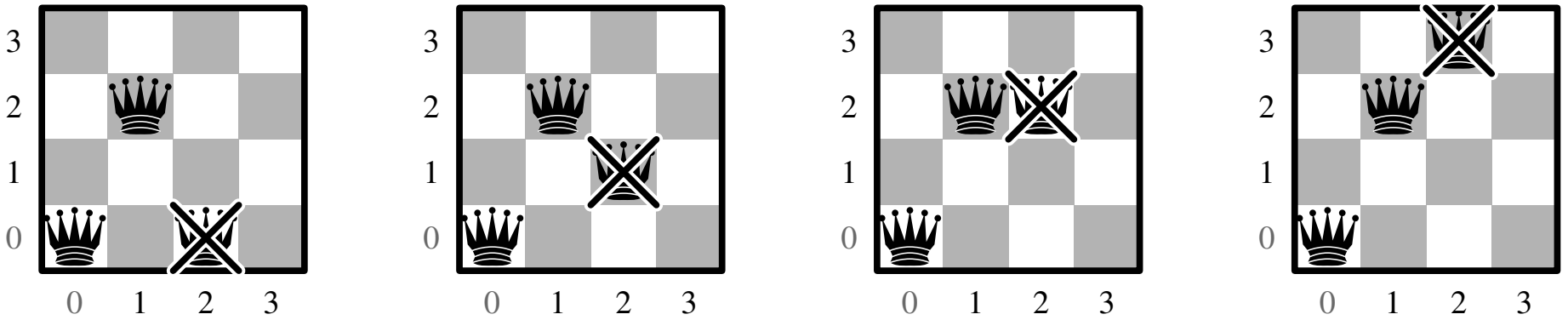
¿Basta con ir ramificando y descartando los problemas?



¿Y ahora qué? ¡La tercera reina no se puede poner en ningún sitio!

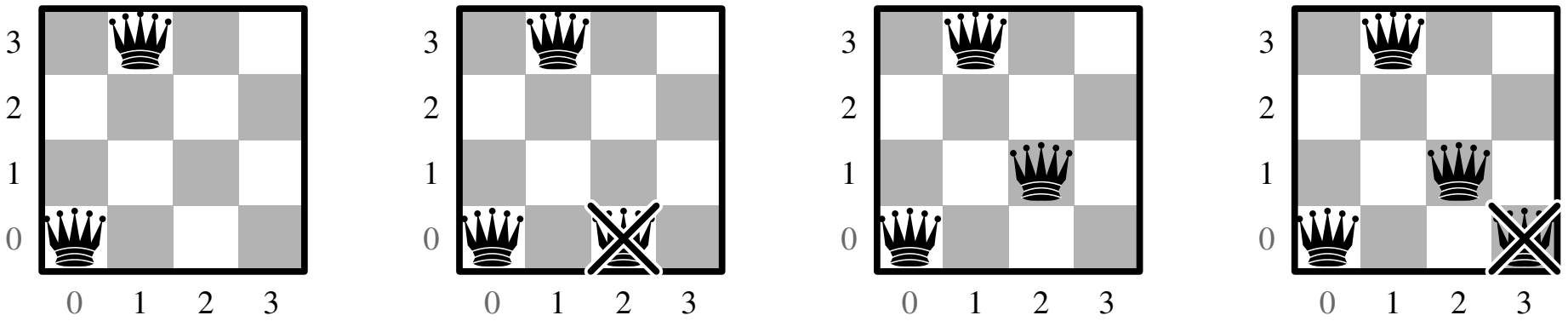
¡Retroceso!

¿Basta con ir ramificando y descartando los problemas?

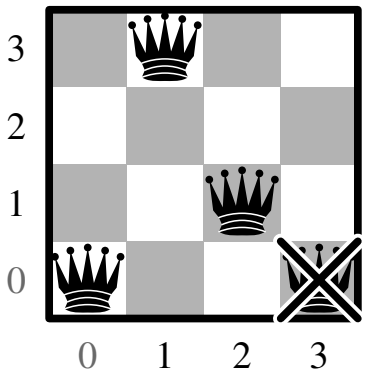


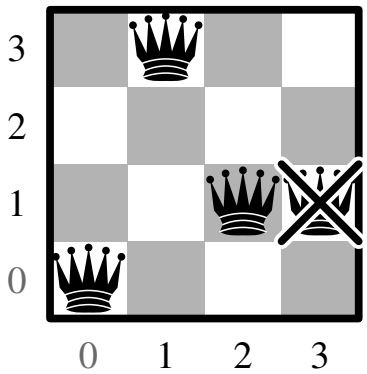
¿Y ahora qué? ¡La tercera reina no se puede poner en ningún sitio! ⇒ Hemos de revisar qué ocurrió con la segunda reina, no puede ir a la fila 2 porque lleva a un callejón sin salida ⇒ Probemos en la fila 3

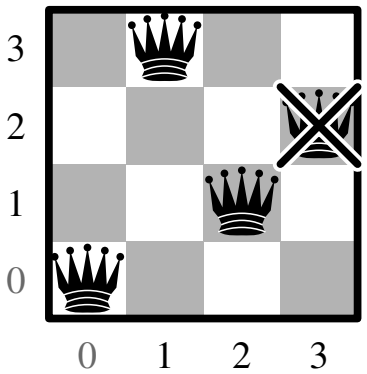
¡Retroceso!

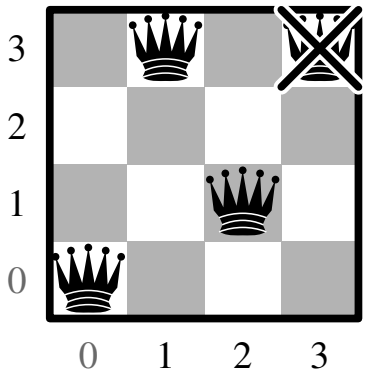


Sigamos

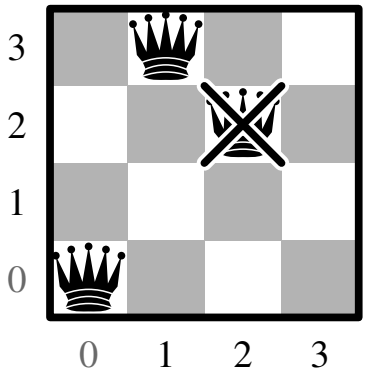


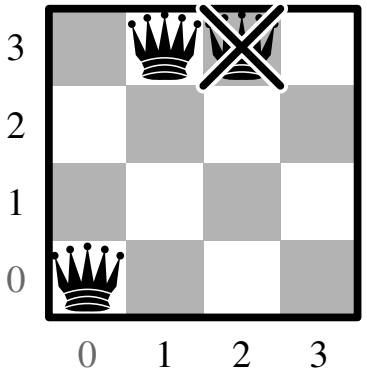






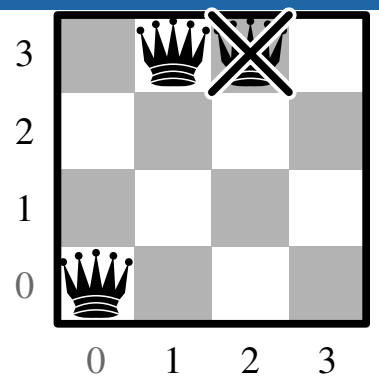
Otro callejón sin salida. Replanteemos qué pasa con la tercera reina.



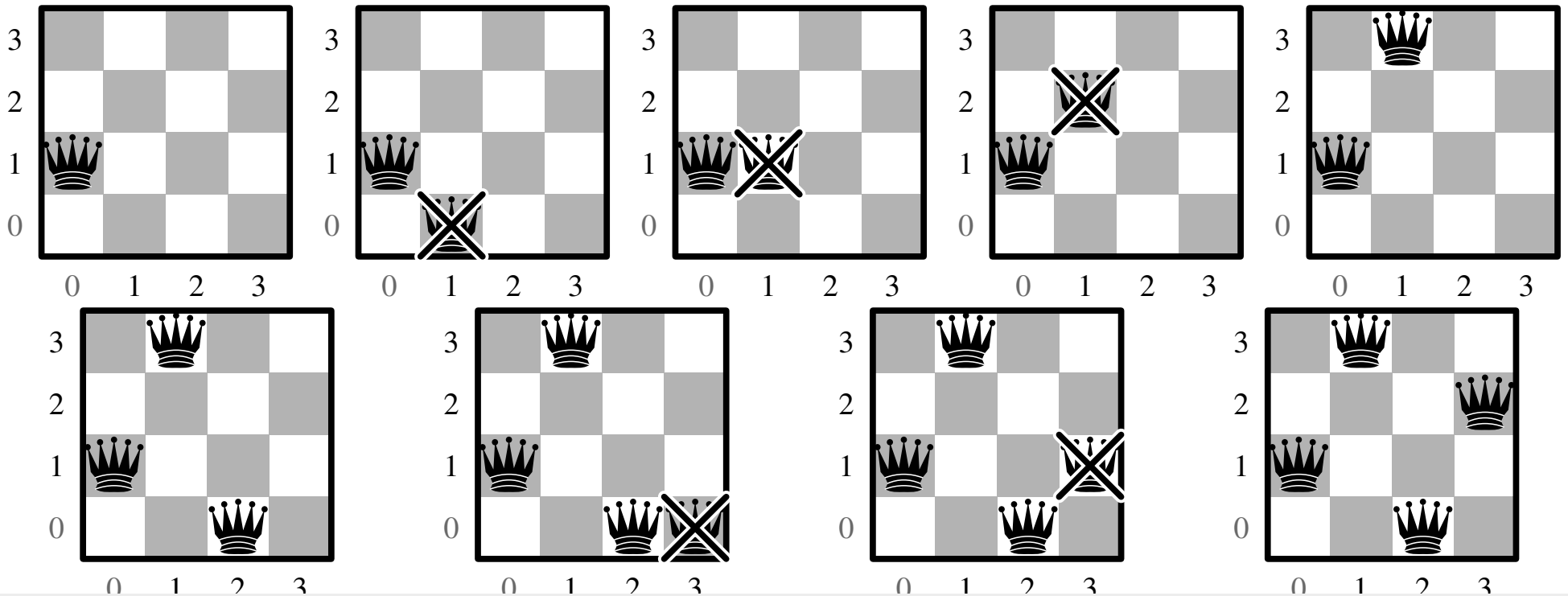


¡Otro problema! ¡¡¡Hemos de retroceder aún más!!!

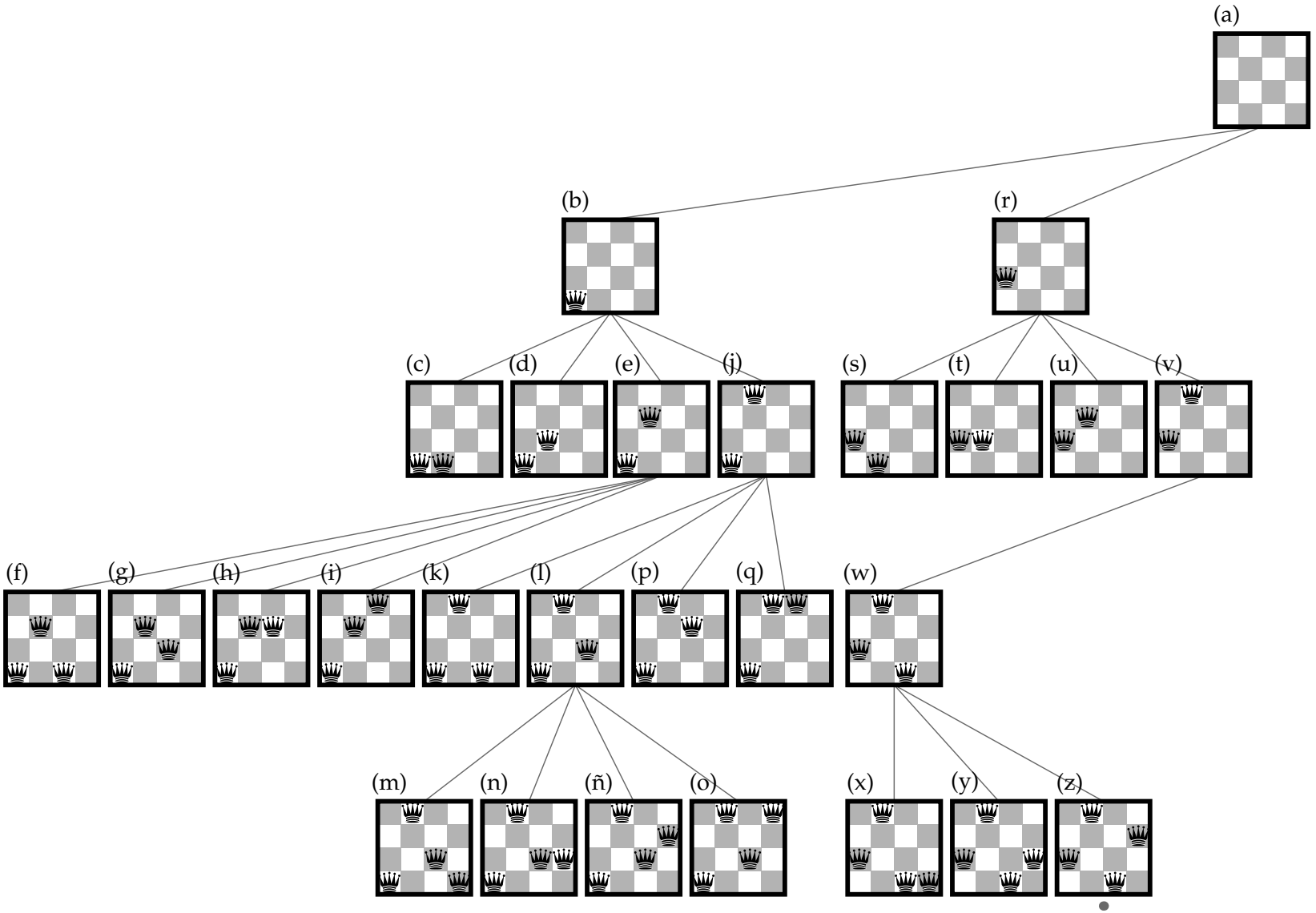
Sigamos



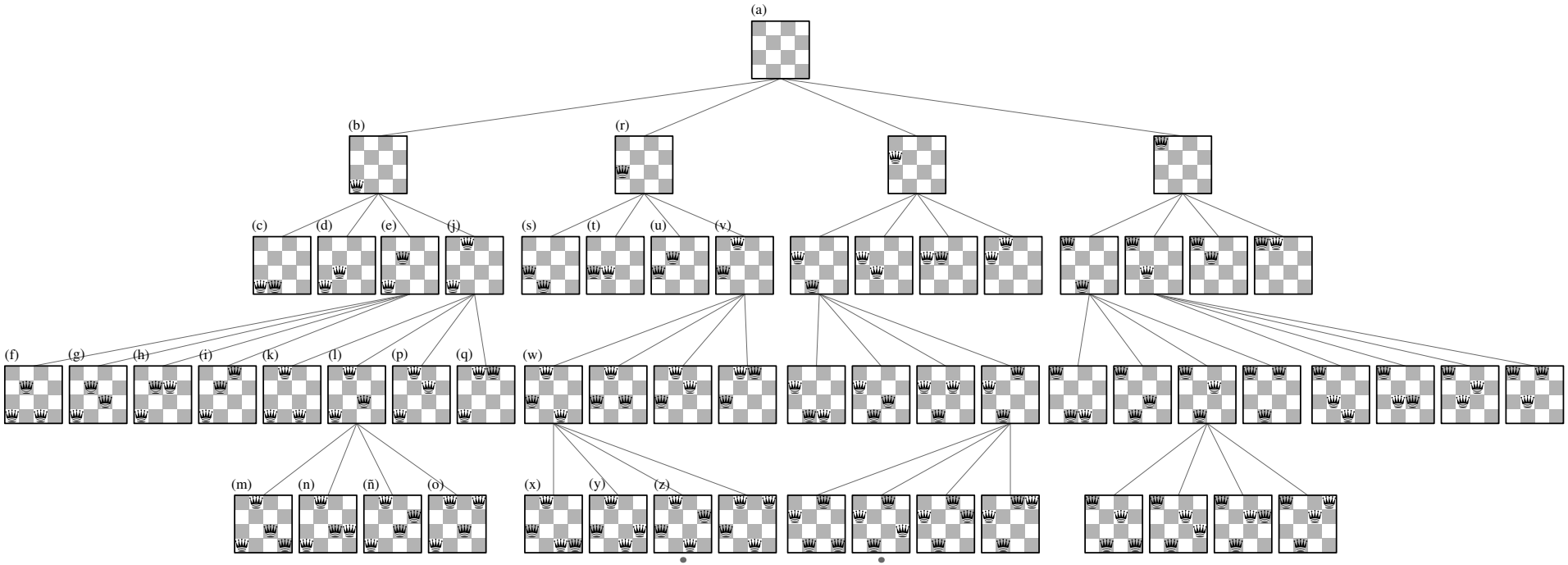
Deberíamos plantearnos de nuevo qué pasa con la segunda reina. Pero la segunda ha agotado sus opciones. Hemos de ir a ver qué pasa con la primera.



Lo realmente explorado



Una visión del problema: exploración de un árbol con raíz

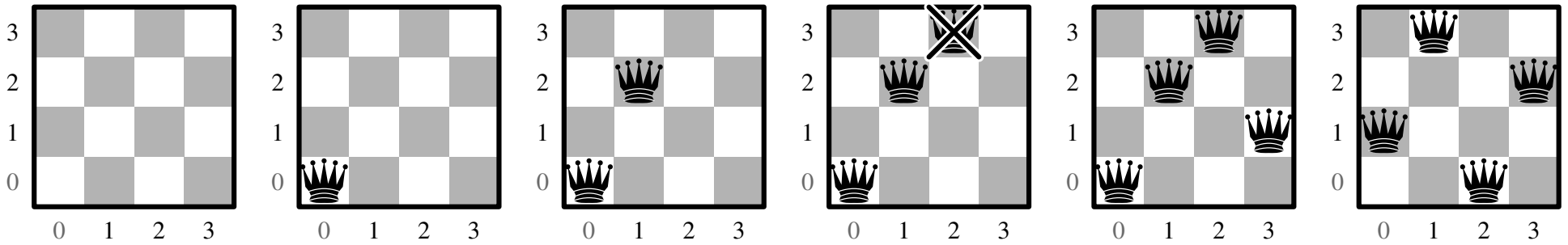


Estados

Un **estado** es un *conjunto de posibilidades*, un **grupo de soluciones** (un subconjunto de X'), y se representa mediante una **tupla**.

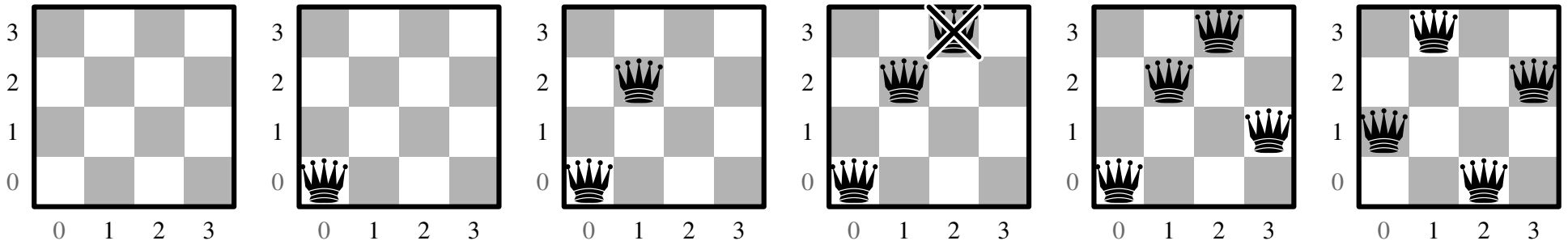
- En el ejemplo, cada posible configuración (con todas o parte de las reinas) es un estado. Es decir, un estado es el conjunto de soluciones que nos quedan cuando ya hemos puesto m reinas (m puede ser 0, menor que n o igual a n):

$$(s_0, s_1, \ldots, s_{m-1}, ?) = \{(x_0, x_1, \ldots, x_{n-1} \in X' | x_i = s_i, 1 \leq i < m\}$$



(?) (0,?) (0,2,?) (0,2,3,?) (0,2,3,1) (1,3,0,2)

Estados



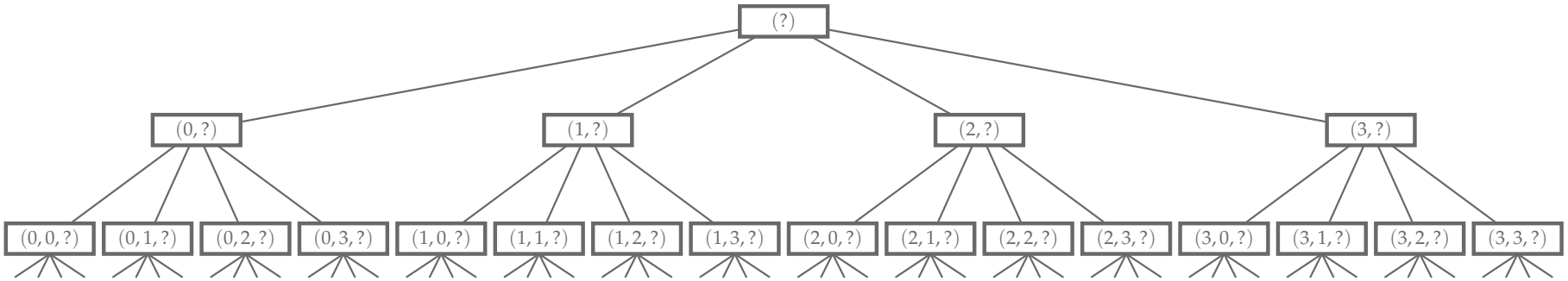
(?) (0,?) (0,2,?) (0,2,3,?) (0,2,3,1) (1,3,0,2)

- El interrogante final representa que aún podrían añadirse nuevas reinas: que la tupla contiene un conjunto de soluciones. Por ejemplo, el estado $(0, 2, 3, ?)$ representa el conjunto formado por 4 soluciones: $\{(0, 2, 3, 0), (0, 2, 3, 1), (0, 2, 3, 2), (0, 2, 3, 3)\}$.
- **Estado terminal o completo**: conjunto con un sólo elemento (conjunto unitario), que es una solución. *No tiene interrogantes.*
- **Estado prometedor**: conjunto de soluciones que podría contener, al menos, una solución factible.
- **Estado no prometedor**: conjunto de soluciones que seguro que no contiene ninguna solución factible.

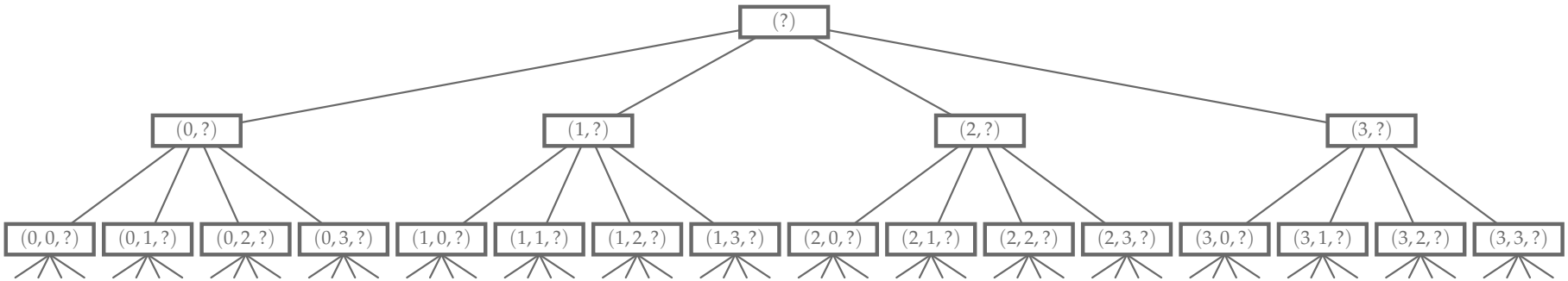
¿Cómo se generan los estados? ⇒ Árbol de estados

Se sigue un proceso de "ramificación" que, aplicado sobre el **estado inicial**, $(?)$ (el conjunto de soluciones X'), induce el denominado **árbol de estados**: cada nodo del árbol es un estado; el estado inicial es la **raíz**.

- Los hijos de un nodo son aquellos estados que se obtienen al **ramificarlo**.
- La **ramificación** de un estado consiste en generar tantos nodos como posibles valores pueda tomar el interrogante del estado (se produce una **partición** del conjunto de soluciones que representa).
- Los **nodos internos** del árbol son estados $(s_0, s_1, \ldots, s_{m-1}, ?)$, con $m < n$.
- Las **hojas** son estados completos, de la forma $(s_0, s_1, \ldots, s_{n-1})$.



BcR como exploración "informada" del árbol



- Idea: recorrer el árbol de estados desde la raíz por primero en profundidad.
- Pero no ramificar todos los estados, sólo aquellos que sean prometedores $\Rightarrow$ **Poda por factibilidad**:
  - Si un estado (no terminal) **no es prometedor**, no tiene sentido aplicarle el proceso de ramificación para dividirlo en nuevos estados.
  - Nunca llegamos a considerar los estados accesibles por ramas que le tienen a él por raíz: las "**podamos**".
  - Ejemplo en las 4 reinas: $(0, 0, ?, ?)$ no puede conducir a una solución factible y se poda.
- Parar al llegar a la primera hoja que contenga una solución factible.

Algoritmo

- Diseñaremos una clase para resolver el problema (*NQueensSolver2*) para un valor de n. El constructor recibirá el valor de n y creará el (único) atributo de la clase:

```
class NQueensSolver1:
    def __init__(self, n):
        self.n = n
```

- Los estados $(s_0, s_1, \ldots, s_{m-1}, ?)$ se representarán con una lista con m elementos. Por ejemplo: $(0, 2, ?) \Rightarrow [0, 2]$ (los estados terminales se identificarán por la longitud de la lista ($m = n$)).
- Para resolver el problema partiremos de la lista vacía (la raíz $(?)$):

```
def solve(self):
    return self.backtracking([])
```

a la que añadiremos un elemento cada vez que fijemos una reina.

- *backtracking* será el método (recursivo) que resuelva el problema: recibirá un estado y devolverá una solución factible asociada a él (si la hay).

Algoritmo

- Tendremos un método que nos indique si una configuración es completa (se han dispuesto todas las reinas):

```
def is_complete(self, s):
    return len(s) == self.n
```

- Dispondremos de otro método que nos diga si la configuración es prometedora (no hay reinas que se amenacen mutuamente (*is_promising*), para ello comprobamos si la última reina no amenaza a las anteriores):

```
def is_promising(self, s, row):
  return all(row != s[i] and len(s)-i != abs(row-s[i]) for i in range(len(s)))
```

¿Qué es *all*?: devuelve *True* si todos los elementos que se le pasan son *True*. El código sería equivalente a este otro:

```
def is_promising(self, s, row):
  for i in range(len(s)):
    if row == s[i] or len(s)-i == abs(row-s[i]): return False
  return True
```

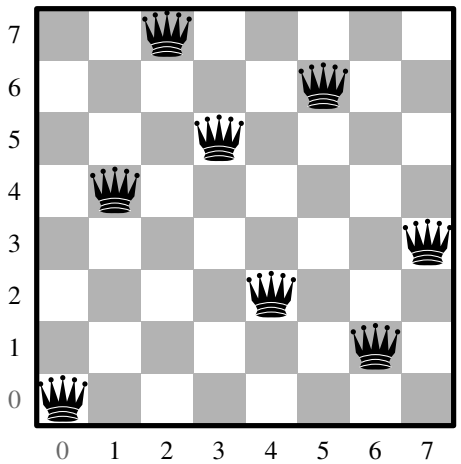
Algoritmo

```
class NQueensSolver1:
    def __init__(self, n):
        self.n = n
    def is_complete(self, s):
        return len(s) == self.n
   def is_promising(self, s, row):
        return all(row != s[i] and len(s)-i != abs(row-s[i]) for i in range(len(s)))

    def backtracking(self, s):
        if self.is_complete(s): return s
        for row in range(self.n):
            if self.is_promising(s, row):
                found = self.backtracking(s+[row])
                if found != None: return found
        return None

    def solve(self):
        return self.backtracking([])
```

Algoritmo: Soluciones

```
from nqueens import NQueensSolver1

for n in range(1, 9):
    print("Solución con %d reinas: %s." % (n, NQueensSolver1(n).solve()))
```

```
Solución con 1 reinas: [0].
Solución con 2 reinas: None.
Solución con 3 reinas: None.
Solución con 4 reinas: [1, 3, 0, 2].
Solución con 5 reinas: [0, 2, 4, 1, 3].
Solución con 6 reinas: [1, 3, 5, 0, 2, 4].
Solución con 7 reinas: [0, 2, 4, 6, 1, 3, 5].
Solución con 8 reinas: [0, 4, 7, 5, 2, 6, 1, 3].
```



Algoritmo: Soluciones con prefijo

La implementación permite encontrar, si las hay, **soluciones factibles con un prefijo predeterminado**. Basta con llamar directamente a *backtracking* suministrando dicho prefijo:

```
from nqueens import NQueensSolver1

print("Solución con 8 reinas: %s." % NQueensSolver1(8).backtracking([2, 4]))
```

```
Solución con 8 reinas: [2, 4, 1, 7, 0, 6, 3, 5].
```

Eficiencia

El algoritmo también es **exponencial**.
Aún así, es **más rápido** que generar toda permutación: aquí abortamos rápidamente los prefijos que no llevan a nada.
¿Qué afecta, en la práctica, a la **eficiencia**?:

- La **representación** de los estados.
- La forma en que **añadimos/quitamos** cosas a los estados (ramificación).
- El **orden** en que ramificamos.
- La comprobación de si un estado es **prometedor** (o completo).

Eficiencia

En nuestro caso:

- Comprobar si m reinas forman un estado prometedor (*is_promising*): $O(m)$.
- Añadir una reina a un estado con m: $O(m)$.

Hemos de añadir n reinas para llegar a un estado completo, y hay que preguntar a cada estado intermedio si es o no prometedor: $\Omega(n^2)$.
Y esto es en el **mejor de los casos**: que el algoritmo vaya de la raíz a la solución directamente.
¿Y el **coste espacial**?

Coste espacial

Cada estado generado requiere espacio $O(n)$ (se copia la memoria de un estado previo y se añade la posición de una nueva reina).
Como la pila de llamadas a función puede tener profundidad $O(n)$, el **coste espacial** será $O(n^2)$.
Todas las listas almacenadas comparten prefijos (cada estado es el anterior más el componente de la ramificación): una **única lista** sería suficiente.
Mejor: una **pila** (con acceso directo a todos sus elementos). Al ramificar apilamos un elemento, al retroceder desapilamos el último y al comprobar si un estado es prometedor recorremos sus elementos como si fuera un vector.

Un refinamiento del algoritmo: reducción coste espacial

```
from lifo import Stack
class NQueensSolver(NQueensSolver1):
    def __init__(self, n):
        self.n = n

    def backtracking(self, s):
        Q = Stack([si for si in s])

        def _backtracking():
            if len(Q) == self.n: return True
            for row in range(self.n):
                if self.is_promising(Q, row):
                    Q.push(row)
                    if _backtracking(): return True
                    Q.pop()
            return False

        if _backtracking(): return list(Q)
        return None
```

Reducimos el coste espacial: en la pila se almacenan todos los estados, $O(n)$.
Temporal: $\Omega(n^2)$ (exponencial en el peor caso).

Coste: ¿Cuántos estados visitamos?

n	estados
1	1
2	6
3	18
4	26
5	15
6	171
7	42
8	876
9	333
10	975
11	517
12	3 066
13	1 365
14	26 495
15	20 280
16	160 712
17	91 222
18	743 229
19	48 184
20	3 992 510
21	179 592
22	38 217 905
23	584 591
24	9 878 316

n	estados
1	1
2	6
5	15
3	18
4	26
7	42
6	171
9	333
11	517
8	876
10	975
13	1 365
12	3 066
15	20 280
14	26 495
19	48 184
17	91 222
16	160 712
21	179 592
23	584 591
18	743 229
20	3 992 510
24	9 878 316
22	38 217 905

Coste: Estados visitados y árbol de estados completo

n	estados	árbol de estados (completo)
1	1	1
2	6	6
3	18	39
4	26	340
5	15	3 905
6	171	55 986
7	42	960 799
8	876	19 173 960
9	333	435 848 049
10	975	11 111 111 110
11	517	313 842 837 671
12	3 066	9 726 655 034 460
13	1 365	328 114 698 808 273
14	26 495	11 966 776 581 370 170
15	20 280	469 172 025 408 063 615
16	160 712	19 676 527 011 956 855 056
17	91 222	878 942 778 254 232 811 937

Versión iterativa

Que se lo mire sólo quien quiera saber más.

Esquema algorítmico

Esquema algorítmico

Definiciones: soluciones (X'), soluciones factibles ($X \subseteq X'$), estado, estado completo, **estado factible**, estado prometedor y no prometedor, función de ramificación, árbol de estados, búsqueda por primero en profundidad, poda por factibilidad …

- Esquema 1: encontrar una solución factible.
- Esquema 2: encontrarlas todas. Interés del esquema 2:
  - Enumerar todas las soluciones.
  - Optimización: encontrar sólo la que hace máximo/mínimo cierto valor.

Esquema que encuentra una solución factible

```
class BacktrackingScheme:
    def is_complete(self , s): #Devuelve cierto sii s es un estado completo

    def is_feasible(self , s): #Devuelve cierto sii la única solución de s es factible

    def is_promising(self , s): #Devuelve falso sii es imposible que s contenga una solución
      factible

    def branch(self , s): #Devuelve una secuencia de estados (que puede ser vacía) obtenidos por
      ramificación de s

    def backtracking(self , s): #Explora el subárbol que tiene por raíz a s y devuelve la primera
       solución factible que encuentra (si la hay). Si no hay ninguna, devuelve None.
        if self.is_complete(s):
            if self.is_feasible(s):
                return s
            else:
            for s0 in self.branch(s):
                if self.is_promising(s0):
                    found = self.backtracking(s0)
                    if found != None: return found
            return None

    def solve (self , s):
        return self.backtracking([])
```

Esquema que enumera todas las soluciones factibles

```
class AllSoluctionsBacktrackingScheme(BacktrackingScheme):
    def backtracking(self, s, solutions): #Explora el subárbol que tiene por raíz a s y genera
      todas las soluciones factibles que encuentra (si las hay)
        if self.is_complete(s):
            if self.is_feasible(s):
              solutions.append(s)
        else:
            for s0 in self.branch(s):
                if self.is_promising(s0):
                    self.backtracking(s0):

    def solve (self , s):
        solutions = []
        self.backtracking([], solutions)
```

A partir del último esquema es fácil proponer la búsqueda de la solución factible que hace óptimo el valor de una función objetivo, es decir, usar la estrategia como técnica para la resolución de ciertos problemas de optimización.

Corrección

Condición 1. Para todo estado s con $|s| > 1$ y para todo conjunto de estados $s' \in \mathit{branch}(s)$, se cumple $s' \subset s$.
Condición 2. Para todo estado s con $|s| > 1$ se cumple $\bigcup_{s' \in \mathit{branch}(s)} s' = s$.
Teorema. *Si la cardinalidad del estado inicial, X, es finita, entonces el esquema algorítmico de búsqueda con retroceso termina.*
Demo: Cada estado tiene menos elementos que su padre y $|X|$ es finito. Y la poda no pierde soluciones.

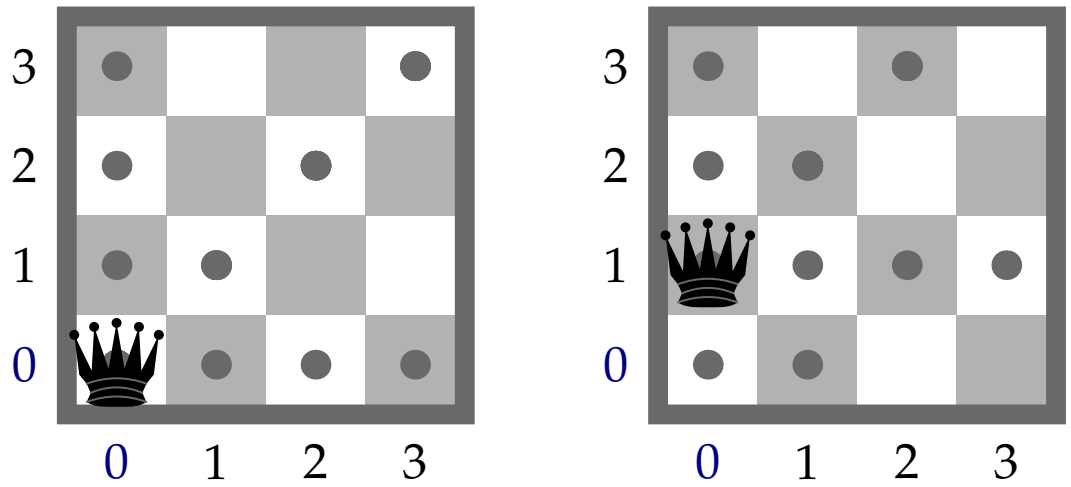
Coste temporal

Depende del tamaño del árbol de estados (suele ser exponencial). **Interesa**:

- Controlar el coste de las funciones del esquema.
- Intentar hacer los cálculos de manera incremental.
- Que se cumpla la

Condición 3 (opcional). Para todo estado s con $|s| > 1$ y para todo par de estados s' y s'' en $branch(s)$, se cumple que $s' \cap s'' = \emptyset$.

- El orden de la ramificación, de forma que se exploren primero los hijos que conduzcan a menos estados prometedores (factor de ramaje bajo en los niveles superiores del árbol):



Coste espacial

La ocupación espacial se debe a:

- La **representación de los estados** $\Rightarrow$ intentar construirlos de manera **incremental**: una sóla variable ($O(n)$) para todas las tuplas (**pila o lista**) sobre la que se añade (al ramificar) y se elimina (al retroceder).
- La **pila de llamadas** (su número dependerá de la tupla de mayor longitud que pueda generarse). Hay que tener cuidado con las variables locales de *backtracking*.

La suma del subconjunto

El problema de la suma del subconjunto

Disponemos de N objetos con pesos $w_1, w_2, \ldots, w_N$ y de una mochila con capacidad para soportar una carga W. Deseamos cargar la mochila con una selección arbitraria de objetos cuyo peso sea *exactamente* W.
Ejemplo:

- $N = 4$
- $w = [4, 5, 3, 6]$
- $W = 8$

¿Por dónde empezar?

Diseñando los estados (las tuplas) y la ramificación:

- ¿Soluciones?
- ¿Estados?
- ¿Ramificación?
- Un estado:
  - ¿Cuándo es completo?
  - ¿Cuándo es no completo?
  - ¿Cuándo es factible?
  - ¿Cuándo es prometedor?

Estados

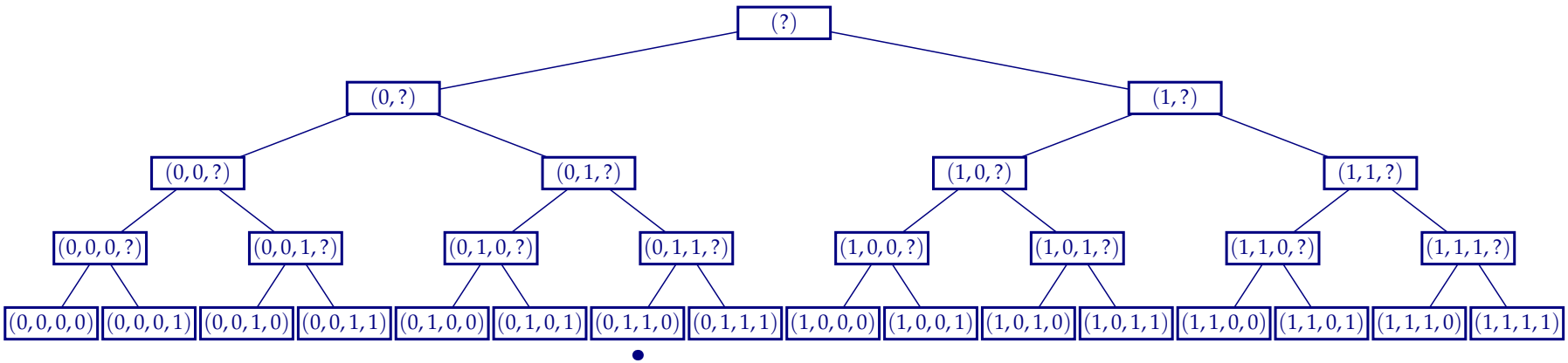
- Un **estado completo** es una tupla $(x_1, x_2, \ldots, x_N) \in \{0, 1\}^N$:
  - si x_i vale 1, el objeto de peso w_i se carga en la mochila;
  - si vale 0, no.

  Hay, pues, 2^N estados completos.
  En nuestro ejemplo ($N = 4$, $w = [4, 5, 3, 6]$ y $W = 8$), $(1, 0, 1, 0)$ representaría cargar los objetos 1 y 3, con un peso asociado de 7.
- Un **estado completo es factible** si
$$\sum_{1 \leq i \leq N} x_i w_i = W$$
- Un **estado no completo** es una tupla con $m < N$ componentes y la representaremos así
$$(x_1, x_2, \ldots, x_m, ?)$$

Ramificación

La **ramificación** de un estado $(x_1, x_2, \ldots, x_m, ?)$ produce los estados: $(x_1, x_2, \ldots, x_m, 0, ?)$ y $(x_1, x_2, \ldots, x_m, 1, ?)$.
Cuando $m = N - 1$, la ramificación da lugar a dos estados completos: $(x_1, x_2, ..., x_m, 0)$ y $(x_1, x_2, ..., x_m, 1)$.

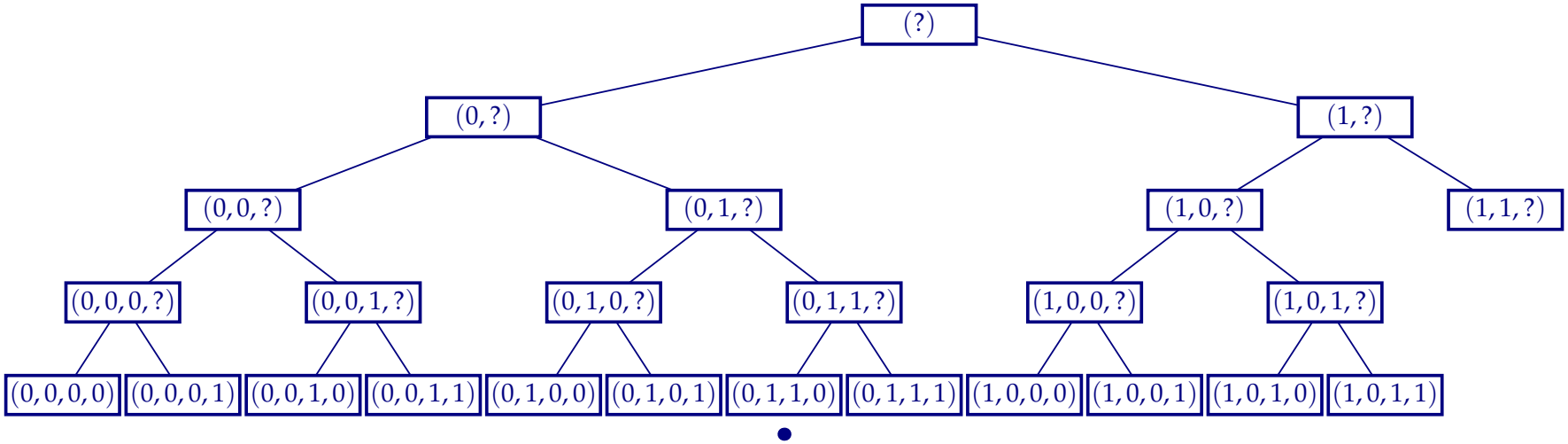
Árbol de estados **completo**



$w_1 = 4$, $w_2 = 5$, $w_3 = 3$, $w_4 = 6$ y $W = 8$

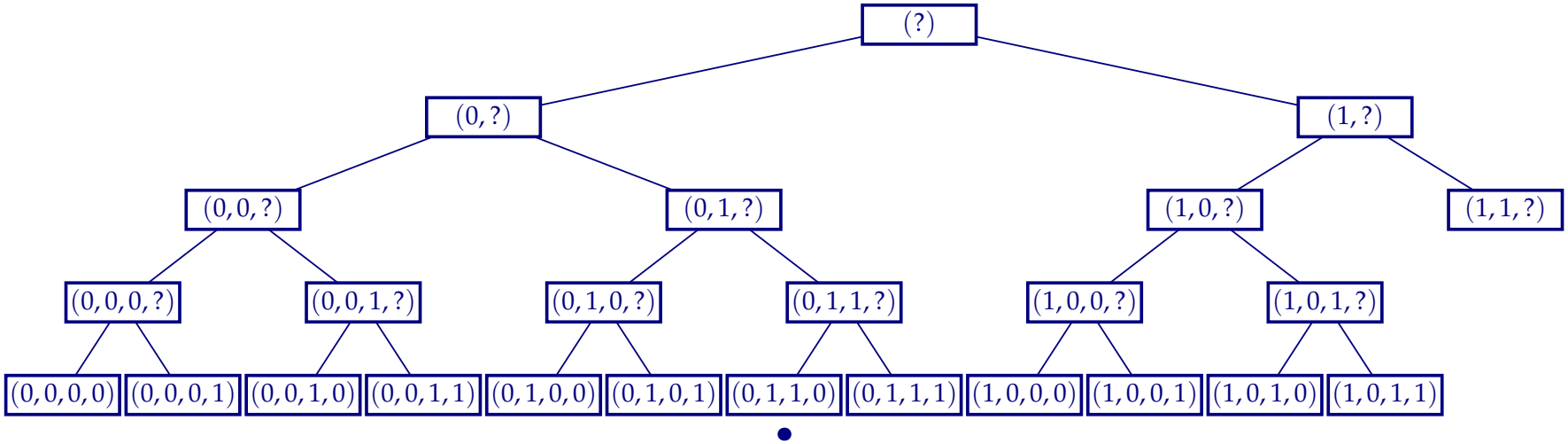
¿Cuándo es **prometedor** un estado? $\Rightarrow$ Un estado es prometedor si su suma de pesos es menor o igual que W.

Árbol de estados sin ramificación de estados no prometedores

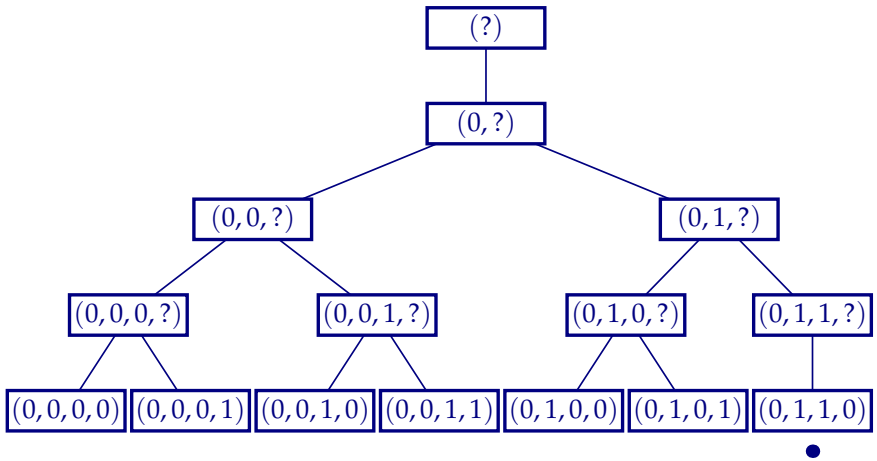


Árbol de estados

Arbol de estados **sin ramificación de estados no prometedores**



Arbol de estados **visitados** en el proceso



Primer algoritmo

- Para representar la tupla, usaremos un vector de N posiciones, x, inicialmente vacío (a 0), como variable global.
- En la búsqueda con retroceso pasaremos como parámetro la posición del vector, i, sobre la que tenemos que actuar (el interrogante).
- El algoritmo recursivo contendrá directamente el cuerpo de las funciones auxiliares del esquema.

Primer algoritmo

```
from offsetarray import OffsetArray

def subset_sum(w, W):
  x = OffsetArray([0] * len(w))
  def backtracking(i):
    if i == len(w)+1:  #is_complete
        if sum([w[j]*x[j] for j in range(1, len(w)+1)]) == W: return x #is_feasible
    else:
      for x[i] in 0, 1:  #branch
        if sum([w[j]*x[j] for j in range(1, i+1)]) <= W:  #is_promising
          found = backtracking(i+1)
          if found != None: return found
    return None
  return backtracking(1)
```

Para este problema, un estado completo y prometedor **podría no corresponder a una solución factible**.

Primer algoritmo

```
from subset_sum1 import subset_sum
from offsetarray import OffsetArray

W, w = 8, OffsetArray([4,5,3,6])
print("Selección para pesos %s y capacidad %d: %s" % (w, W, subset_sum(w, W)))


Selección para pesos [4, 5, 3, 6] y capacidad 8: [0, 1, 1, 0]
```

Refinamientos

1. Considerar los **objetos ordenados de menor a mayor peso**: simplifica saber si un estado es prometedor. Si el objeto *actual* no cabe, no cabrá ninguno de los que le siguen, por lo que el estado *no es prometedor y se poda*.
2. Mantener el **peso que aún queda por cargar en una variable** W que pasaremos como parámetro: evitamos sumatorios.
3. Si el **peso consumido coincide con** W, ya hemos acabado: el estado completo que se obtiene rellenando el resto con ceros es una solución factible (no es necesario llegar a un estado completo).
4. **Considerar primero la inclusión** de un objeto en la mochila que su no inclusión: la rama del 1 tiene, en principio, menos hojas.
5. Conocer la **suma de los pesos de los objetos aún no considerados**: si es menor que el peso disponible, el estado es no prometedor y se poda (para ello hay que preprocesar los datos).

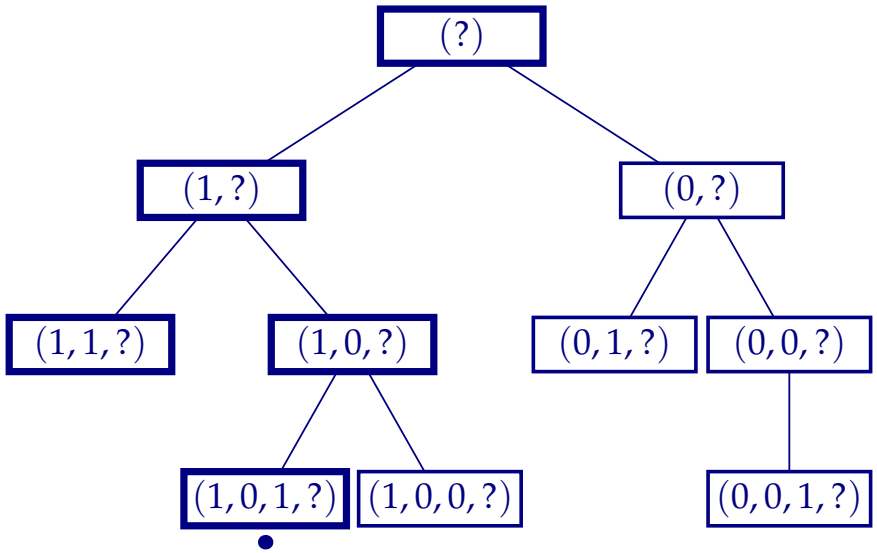
Algoritmo

```
from offsetarray import OffsetArray

def subset_sum(w, W):
  w = OffsetArray(sorted(w))
  sum_w = OffsetArray([0]* (len(w)+1))
  sum_w[len(w)] = w[len(w)]
  for i in range(len(w)-1, 0, -1): sum_w[i] = sum_w[i+1] + w[i]
  x = OffsetArray([0] * len(w))

  def backtracking(i, W):
    if W == 0:
      for j in range(i+1, len(w)+1): x[j] = 0
      return x
    elif i <= len(w):
      for x[i] in 1, 0:
        if W-x[i]*w[i] >= 0 and sum_w[i+1] >= W-x[i]*w[i]:
          found = backtracking(i+1, W-x[i]*w[i])
          if found != None: return found
        else:
          return None
    return None

  return backtracking(1, W)
```

Algoritmo

```
from subset_sum import subset_sum
from offsetarray import OffsetArray

W, w = 8, OffsetArray([4,5,3,6])
print("Selección para pesos %s y capacidad %d: %s" % (sorted(w), W, subset_sum(w, W)))


Selección [3, 4, 5, 6] y capacidad 8: [1, 0, 1, 0]
```

Árbol de estados sin ramificación de no prometedores (en trazo **grueso**, los visitados).



Traza

Haced una traza del siguiente problema con el algoritmo original y el refinado:

- $N = 5$
- $w = [2, 3, 4, 7, 1]$
- $W = 15$

Dibujad los correspondientes árboles de **estados visitados** por los algoritmos.

Traza

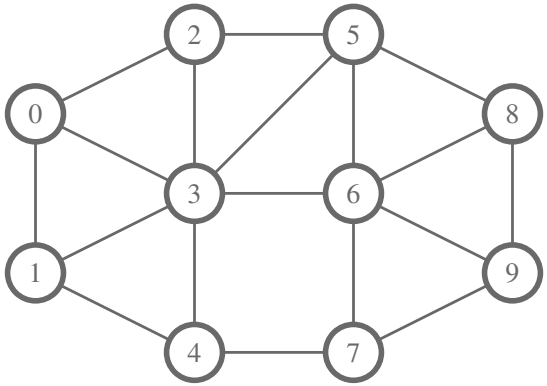
Número de llamadas a *backtracking* (estados visitados) con 100 instancias aleatorias del problema (con solución) para N variando entre 1 y 15. La última columna muestra el porcentaje de estados visitados por el algoritmo refinado con respecto al original.

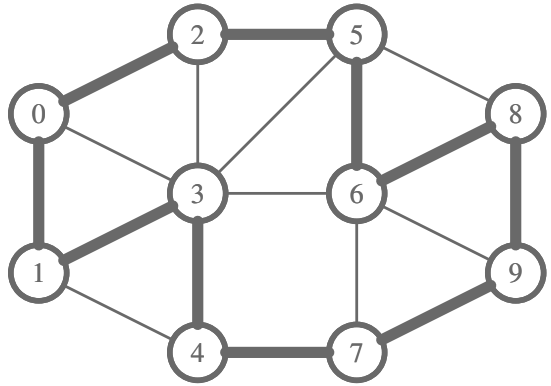
N	Original	Refinado	%
1	300	200	66.7
2	577	302	52.3
3	1051	429	40.8
4	1897	570	30.0
5	3395	839	24.7
6	5428	1168	21.5
7	11660	1424	12.2
8	18899	1931	10.2
9	30934	2072	6.7
10	54353	2456	4.5
11	99718	2979	3.0
12	216248	3511	1.6
13	359453	3493	1.0
14	597507	3966	0.7
15	1298615	3587	0.3

Existencia ciclo Hamiltoniano

Existencia de un ciclo Hamiltoniano

Dado un grafo $G = (V, E)$ deseamos encontrar un camino que parta de un vértice, termine en el mismo vértice y visite todos los vértices una sola vez (excepto, claro está, el de partida, que también es de llegada). Un camino como el descrito es un **ciclo hamiltoniano**:





¿Cuántos ciclos hamiltonianos podría haber? $O(|V|!)$ Un algoritmo basado en la **fuerza bruta** podría generar todas las permutaciones y comprobar, para cada una, si todo par de vértices consecutivos es una arista del grafo.

Existencia de un ciclo Hamiltoniano

- ¿Estados? $\Rightarrow$ Tupla de vértices (vector)
- ¿Ramificación? $\Rightarrow$ Sucesores del último vértice de la tupla
- Un estado:
  - ¿Cuándo es completo? $\Rightarrow$ $|V|$ vértices.
  - ¿Cuándo es factible? $\Rightarrow$ Si hay una arista entre el último y el primer vértice.
  - ¿Cuándo es prometedor? $\Rightarrow$ Ningún vértice repetido (al añadir el último comprobar que no estaba).
- Punto de partida: un vértice cualquiera.

Algoritmo inicial

```
from random import sample
def hamiltonian_cycle(G):
  def backtracking(path):
    if len(path) == len(G.V):
      if (path[-1], path[0]) in G.E: return path+[path[0]]
    else:
      for v in G.succs(path[-1]):
        if v not in path:
          found = backtracking(path+[v])
          if found: return found
    return None
 [random vertex] = sample(G.V, 1) # Selecciona un vértice al azar.
 return backtracking([random vertex])


from hamiltonian_cycle1 import hamiltonian_cycle
from graph import Graph

G = Graph(E=[(0,1), (0,2), (0,3), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5),
                        (3,6), (4,7), (5,6), (5,8), (6,7), (6,8), (6,9), (7,9), (8,9)])
print("Ciclo Hamiltoniano:", hamiltonian_cycle(G))


Ciclo hamiltoniano: [0, 1, 3, 4, 7, 9, 8, 6, 5, 2, 0]
```

Mejoras (reducción coste espacial y temporal)

1. Usar **una única lista para representar el estado actual**, reduciendo el coste de generar un nuevo estado (crear un vector estático y utilizar un parámetro *m* para indicar la posición del último vértice en el vector).
2. Usar una representación de los vértices que forman parte del camino que **no** requiera **recorrer una lista para determinar si un vértice pertenece o no al camino**. $\Rightarrow$ Conjunto *visited* adicional.

Algoritmo mejorado

```
from random import sample
def hamiltonian_cycle(G):
  [random_vertex] = sample(G.V , 1)
  path = [random_vertex] + [None] * (len(G.V)-1) + [random_vertex]
  def backtracking(m, visited):
    if m == len(G.V)-1:
      if (path[m], path[0]) in G.E: return path
    else:
      for v in G.succs(path[m]):
        if v not in visited:
          path[m+1] = v
          visited.add(v)
          found = backtracking(m+1, visited)
          if found: return found
          visited.remove(v)
    return None
  return backtracking(0, set([random_vertex]))
```

Es fácil analizar el coste en el mejor de los casos. ¿Qué coste tiene *en el peor*?

Bibliografía

Bibliografía

- *Algorithms*, de Jeff Erikson. Capítulo 2. Disponible en estos enlaces:
  - Todo el libro
  - Capítulo 2
- Capítulo *"Búsqueda con retroceso"* del libro *Curso de algoritmia* de Andrés Marzal, María José Castro y Pablo Aibar.