# Apuntes de Algorítmica

Andrés Marzal
María José Castro
Pablo Aibar

Borrador
6 de febrero de 2007

# Capítulo 6
# BÚSQUEDA CON RETROCESO

Recordemos brevemente algo de la terminología propia de los problemas de optimización. En ellos proponíamos la búsqueda de un elemento de cierto conjunto que ofreciera el valor extremo de una función y, a la vez, satisficiera una serie de restricciones. Al conjunto de elementos se le denominada conjunto de soluciones, y a las soluciones que satisficieran las restricciones, soluciones factibles. En este capítulo no nos encargaremos principalmente de problemas de optimización: buscaremos una solución factible que satisfaga un conjunto de restricciones, pero sin exigir que una función proporcione un valor máximo o mínimo sobre el resultado de la búsqueda. Los algoritmos de **búsqueda con retroceso** proponen la exploración sistemática del conjunto de soluciones hasta encontrar una factible *cualquiera*. La estrategia de la búsqueda con retroceso permite abordar ciertos problemas mediante un mecanismo de «casi fuerza bruta», esto es, de exploración sistemática de (potencialmente) todas las soluciones (factibles o no) hasta dar con una que sea factible. Se diferencia de la fuerza bruta en que puede decidir no explorar grandes conjuntos de soluciones cuando determina que entre ellas no hay ninguna factible.

Empezaremos ilustrando la técnica con un ejemplo paradigmático, el conocido como «problema de las $n$ reinas», y presentaremos después el esquema algorítmico. A continuación estudiaremos la resolución de diferentes problemas. El capítulo finaliza con la exposición de «los enlaces bailarines», una técnica de implementación de la búsqueda con retroceso aplicable en la resolución de problemas reducibles al denominado «problema de la cobertura exacta». Veremos cómo su aplicación al «problema de las $n$ reinas» resulta en un algoritmo mucho más eficiente que el que presentamos en el primer apartado.

## 6.1. El problema de las $n$ reinas

En el juego del ajedrez, una reina amenaza a cualquier pieza dispuesta en su misma fila, columna o diagonal. La figura 6.1 muestra las casillas en las que una pieza se vería amenazada por una reina.

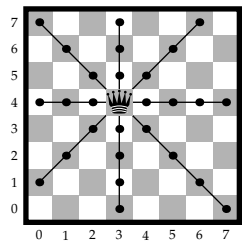



*Figura 6.1: Tablero de* $8 \times 8$ *en el que se muestran, marcadas con un punto, las casillas amenazadas por una reina. Dichas casillas ocupan la misma fila, columna o diagonal que la reina.*

Deseamos disponer $n$ reinas en un tablero de ajedrez de $n \times n$ escaques de modo que ninguna amenace a otra. La figura 6.2 muestra dos configuraciones de 4 reinas en un tablero de $4 \times 4$. Cada configuración recibirá el nombre de «solución»: la primera no satisface la condición impuesta y la segunda sí. Distinguiremos entre soluciones (cualesquiera) y soluciones factibles: denominamos solución a cualquier disposición de $n$ reinas, contenga o no piezas que se amenacen, y solución factible a aquella en la que ningún par de reinas se amenazan mutuamente.

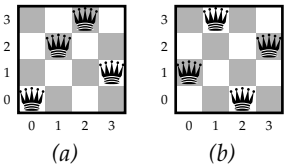



*Figura 6.2: Dos configuraciones de 4 reinas en un tablero de* $4 \times 4$*. La configuración (a) es una solución, pero no es factible, pues las reinas de la segunda y tercera columnas se amenazan mutuamente. La configuración (b) sí es una solución factible.*

### 6.1.1. Modelado

Podemos describir una solución (factible o no) con una secuencia de pares de enteros $((i_0, j_0), (i_1, j_1), \ldots, (i_{n-1}, j_{n-1}))$. Cada par indica la posición (columna y fila) de una reina. Con la numeración de columnas y filas que se muestra en las figuras, cada $i_k$ y $j_k$ está comprendido entre 0 y $n-1$. Por ejemplo, la configuración de la figura 6.2 (b) se describe con $((0,1),(1,3),(2,0),(3,2))$.

La notación se puede simplificar si tenemos en cuenta que no puede haber dos reinas en una misma columna y que, por ello, ha de haber una reina en cada una de las $n$ columnas. Podemos acordar que la reina dispuesta en la columna $i$ ocupe la posición $i$ de la tupla. De este modo es posible convertir la tupla de pares en una tupla de enteros: en cada elemento se indica la fila en la que ponemos la reina de la correspondiente columna. Una solución (factible o no) puede describirse entonces con una $n$-tupla de enteros $(j_0, j_1, \ldots, j_{n-1})$. Con esta notación más compacta, la configuración de la figura 6.2 (b) se denota así: $(1,3,0,2)$. Denotaremos con $X'$ al conjunto de soluciones (factibles o no), que puede definirse formalmente así: $X' = [0..n-1]^n$.

Para que un elemento de $X'$ represente una solución factible, no puede haber elementos repetidos entre los $n$ valores: la tupla debe formar una *permutación* de los números $\{0, 1, \ldots, n-1\}$. Pero no todas la permutaciones definen soluciones factibles: el hecho de que una reina amenace a cualquier pieza en una diagonal hace que permutaciones como $(0,2,3,1)$ no sean admisibles (véase la figura 6.2 (a)). ¿Cómo detectar que dos reinas ocupan una misma diagonal? Muy sencillo: si la diferencia entre sus columnas coincide (en

valor absoluto) con la diferencia entre sus filas. El conjunto de las soluciones factibles, que denotaremos con $X$, puede definirse así:

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

### 6.1.2. Búsqueda con retroceso

Podríamos intentar solucionar el problema generando las $n!$ permutaciones de los $n$ enteros y considerando si alguna de ellas describe una solución factible. El número de permutaciones crece exponencialmente con $n$ (recordemos que Stirling aproxima $n!$ con $\sqrt{2\pi n}(n/e)^n$), por lo que este algoritmo resulta altamente ineficiente.

La técnica de la exploración con retroceso procede construyendo una tupla elemento a elemento: primero considera un posible valor para la primera reina; después, uno para la segunda, y así sucesivamente. Cuando se detecta que una configuración no puede conducir a una solución factible, se retrocede (se eliminan reinas en orden inverso a aquél con el que se fueron disponiendo) en la serie de configuraciones incompletas exploradas. El retroceso se detiene al encontrar una configuración que permita avanzar de forma diferente. Ilustramos cada acción del proceso para $n = 4$ en la figura 6.3.

Es posible ver el problema como la exploración de un árbol con raíz de posibles configuraciones del tablero con cualquier número de reinas entre 0 y $n - 1$. La figura 6.4 muestra un árbol con algunas configuraciones del tablero para el problema de las 4 reinas. Nótese que no es un árbol completo, pues algunas configuraciones con menos de cuatro reinas no se ramifican: aquellas en las que se ha detectado una amenaza entre algunas de las reinas dispuestas y que, por tanto, no tiene sentido extender.

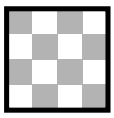
El procedimiento que hemos seguido antes describe un recorrido por primero en profundidad sobre el árbol. En cada instante de la ejecución del algoritmo se está visitando un nodo del árbol que corresponde a un conjunto de soluciones. Si el conjunto contiene una única solución, ésta puede ser factible o no. El árbol no se explora en su totalidad porque el procedimiento finaliza tan pronto encontramos una solución factible. En una exploración por primero en profundidad del árbol de la figura 6.4, por ejemplo, la búsqueda se detiene cuando se alcanza la solución marcada con (z) aunque, en principio, podríamos seguir explorando otros dos hijos de la raíz (y descendientes de estos hijos).

### 6.1.3. Estados y estados terminales

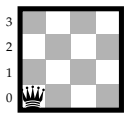
Podemos denotar cada una de las disposiciones incompletas de reinas con una secuencia como ésta: $(s_0, s_1, \ldots, s_{m-1}, ?)$, donde $m < n$. El interrogante denota que la tupla no es una solución completa y que estamos representando todo un conjunto de soluciones: las que tienen como prefijo a la propia tupla sin el interrogante.

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

Cuando hay $n$ componentes con valor, el interrogante sobra, pues no hay más reinas que disponer. Cada tupla es un **estado** y denota un conjunto de soluciones: en el problema de las 4 reinas, el estado $(0, 2, 3, ?)$, por ejemplo, denota el conjunto formado por cuatro

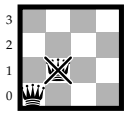


*(a) El procedimiento parte del tablero vacío.*



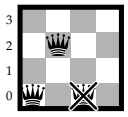
*(b) Considera un valor arbitrario para el primer elemento de una tupla, la fila en la que ubicaremos la reina de la primera columna. Empieza por la fila 0.*

*(c) Considera el valor 0 para la segunda reina, pero no puede formar una solución factible (comparte fila con la primera), así que lo descarta.*



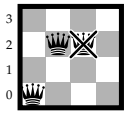
*(d) Pasa al valor 1, pero tampoco es posible formar una solución factible (la segunda reina comparte diagonal con la primera), así que lo descarta.*

*(e) Pasa a considerar el valor 2, que permite colocar la reina sin que sea amenazada.*



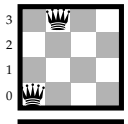
*(f) Ahora que ha determinado los dos primeros valores de la tupla, considera valores para el tercer elemento. Y empieza por el valor 0, que no es aceptable.*

*(g) Como tampoco lo es el valor 1,*



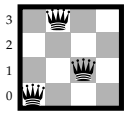
*(h) ni el valor 2,*

*(i) ni el valor 3. No hay posición alguna válida para la tercera reina, a menos que reconsideremos la ubicación de la segunda.*



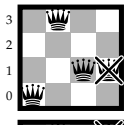
*(j) Se reconsidera la posición de la segunda reina, que pasa a ubicarse en la fila 3.*

*(k) Se vuelve a estudiar qué posiciones son válidas para la tercera reina. No es válida la fila 0,*

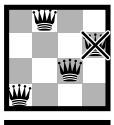


*(l) pero sí la fila 1.*

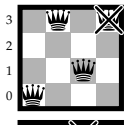
*(m) Pasamos a considerar la colocación de la última reina. No es válida la fila 0,*



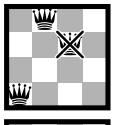
*(n) ni la fila 1,*



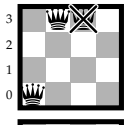
*(ñ) ni la fila 2,*



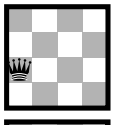
*(o) ni tampoco la fila 3. ¿Qué hacer entonces?*



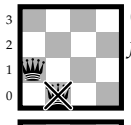
*(p) Reconsiderar la ubicación de la tercera reina. No podemos situarla en la fila 2,*



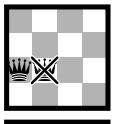
*(q) ni en la 3.*



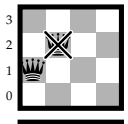
*(r) No es posible considerar nuevas posiciones para la segunda reina a menos que reubiquemos la primera. La ponemos en la fila 1.*



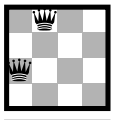
*(s) La segunda reina no puede ponerse en la fila 0,*



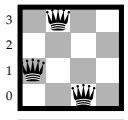
*(t) ni en la fila 1,*



*(u) ni en la fila 2,*

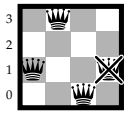


*(v) pero sí en la fila 3.*

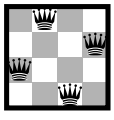


*(w) Pasamos a considerar posibles ubicaciones de la tercera reina. La fila 0 es una buena ubicación.*

*(x) Y pasamos a considerar dónde colocar la cuarta reina. Ubicarla en la fila 0 expone a la cuarta reina al ataque de la anterior reina.*



*(y) La fila 1 está amenazada,*



*(z) pero la fila 2 es una ubicación segura. Ya está. Hemos colocado las cuatro reinas.*

*Figura 6.3: Pasos seguidos en el algoritmo con retroceso para ubicar las 4 reinas.*

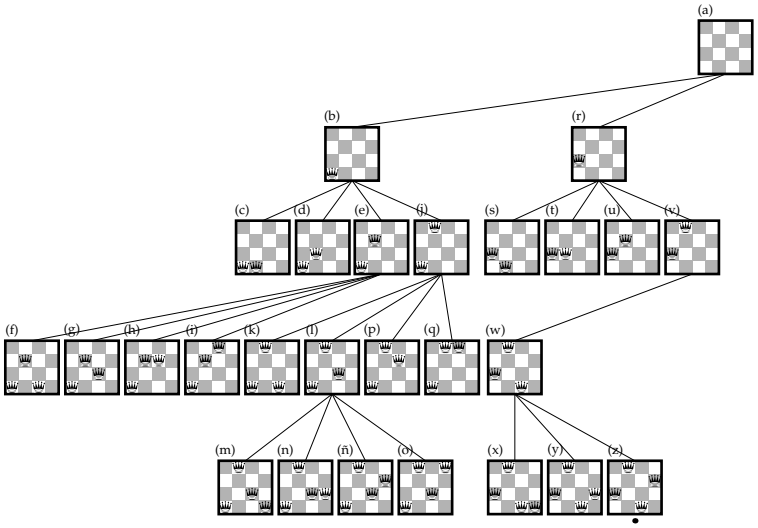



*Figura 6.4: Organización de las configuraciones de la figura 6.3 en forma de árbol con raíz. La relación padre-hijos muestra qué configuraciones se generan a partir de cuál. La relación de orden entre hermanos es consecuente con el propio orden exploración seguido. Nótese que las configuraciones conflictivas con menos de cuatro reinas no tienen hijos.*

soluciones: $\{(0,2,3,0),(0,2,3,1),(0,2,3,2),(0,2,3,3)\}$. Un estado con las $n$ reinas ya dispuestas, es un **estado terminal** o **completo** y describe un conjunto con un sólo elemento, es decir, un **conjunto unitario** cuyo elemento es una solución.

### 6.1.4. El árbol de estados

Un estado puede dividirse en otros estados dando un valor al interrogante de la tupla y extendiendo ésta, si procede, con otro interrogante. Como el interrogante puede tomar varios valores, se genera un proceso de «ramificación» que, aplicado sobre la raíz o **estado inicial**, $(?)$, induce el denominado **árbol de estados**, un árbol con raíz cuyos nodos son estados. El estado inicial es uno de sus nodos: la raíz. Los nodos internos del árbol son estados de la forma $(s_0, s_1, \ldots, s_{m-1}, ?)$, con $m < n$ y las hojas son estados completos, es decir, de la forma $(s_0, s_1, \ldots, s_{n-1})$.

En el problema de las 4 reinas, la raíz o estado inicial es la tupla $(?)$: representa al conjunto de todas las configuraciones posibles de 4 reinas. Cada uno de sus hijos considera una de las 4 posibles ubicaciones de la primera reina, y son $(0,?)$, $(1,?)$, $(2,?)$ y $(3,?)$. Los cuatro conjuntos son una partición del conjunto del padre en cuatro subconjuntos disjuntos. Los nodos hijos de cada uno de ellos añaden una segunda reina y dividen el conjunto en cuatro nuevos subconjuntos. La figura 6.5 muestra los tres primeros niveles del árbol de estados para el problema de las 4 reinas.

### 6.1.5. Estados prometedores

Un estado es **prometedor** si *puede* contener una solución factible y, por tanto, merece ser explorado. En nuestro caso, consideramos que un estado es prometedor si ninguna de las reinas ya dispuestas amenaza a otra. Nótese que si un estado es no prometedor sabemos que no contiene solución factible alguna, pero que si es prometedor *no hay garantía de que contenga una solución factible*.

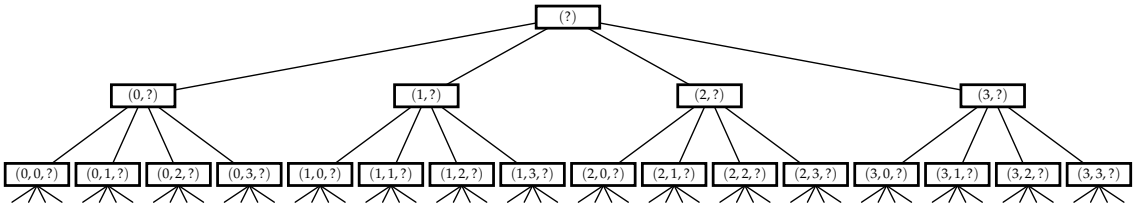



*Figura 6.5: Primeros tres niveles del árbol de estados para el problema de las 4 reinas.*

¿Cómo decidir, en el problema de las $n$ reinas, si el estado $(s_0, s_1, \ldots, s_{m-1}, ?)$ es o no es prometedor? Dos reinas se amenazan si comparten columna, fila o diagonal. Por la representación adoptada, dos reinas no pueden compartir columna, y no comparten fila si $s_i \neq s_j$, para $0 \leq i < j < m$. Finalmente, dos reinas no comparten diagonal si las diferencias entre sus respectivas filas y columnas (en valor absoluto) tienen valores distintos (véase la figura 6.6), es decir, si $|s_j - s_i| \neq j - i$, para $0 \leq i < j < m$.

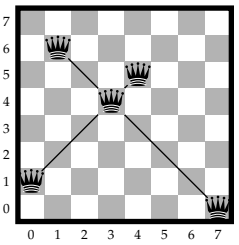



*Figura 6.6: Cuatro reinas amenazadas por la reina central. El valor absoluto de la diferencia entre el número de filas y el número de columnas de cada reina con respecto a la reina del centro coincide en todas las reinas que comparten diagonal. La reina central está en la fila 4 y columna 3. La reina de la fila 6 está en la columna 1. Puede comprobarse que $|6 - 4| = |1 - 3| = 2$.*

### 6.1.6. Poda por factibilidad

Si un estado (no terminal) no es prometedor, no tiene sentido aplicarle el proceso de ramificación para dividirlo en nuevos estados: si no contiene solución factible alguna, ¿para qué explorar los estados que surgen de ramificar éste? Así pues, nunca llegamos a considerar los estados accesibles por ramas que le tienen a él por raíz: las «podamos». Esta poda del árbol reduce significativamente el espacio de búsqueda pendiente de explorar y recibe el nombre de **poda por factibilidad**.

En el problema de las 4 reinas, por ejemplo, el estado $(0, 0, ?)$ no puede conducir a una solución factible y, por tanto, podemos no ramificarlo y tener la garantía de no estar pasando por alto alguna solución factible. No considerar los descendientes del estado no prometedor $(0, 0, ?)$ nos ahorra visitar 20 estados (de los cuales 16 son completos).

### 6.1.7. Un algoritmo recursivo

Diseñaremos una clase para la resolución del problema. Una vez instanciada, un método permitirá resolver instancias con diferentes valores de $n$. Nos conviene disponer de métodos que determinen si una configuración de reinas es completa (ya se han dispuesto todas las reinas) y si es prometedora (no contiene reinas que se amenacen mutuamente):

nqueens.py

```
1  class NQueensSolver1:
2      def is_complete(self, s):
3          return len(s) == self.n
4
5      def is_promising(self, s):
6          return all(s[j] != s[i] and j-i != abs(s[j]-s[i]) for i in xrange(len(s)) \
7                                                           for j in xrange(i+1, len(s)))
```

En *is_complete* hacemos referencia a un atributo *self*.*n*. Es el número de reinas y se proporcionará al constructor:

nqueens.py *(cont.)*

```
10     def __init__(self, n):
11         self.n = n
```

He aquí, expresado en Python, un método recursivo que resuelve el problema usando las funciones que hemos definido:

nqueens.py *(cont.)*

```
13     def backtracking(self, s):
14         if self.is_complete(s): return s
15         for row in xrange(self.n):
16             s' = s + [row]
17             if self.is_promising(s'):
18                 found = self.backtracking(s')
19                 if found != None: return found
20         return None
```

Hemos representado cada estado de la forma $(s_0, s_1, \ldots, s_{m-1}, ?)$ con una lista de longitud $m$. Cada llamada a *backtracking* recibe una lista con $m$ reinas ya dispuestas. Si la lista representa un estado terminal, es decir, si $m$ es igual a $n$, entonces se devuelve la lista como solución. En caso contrario, se procede a ramificar el estado considerando todas las posiciones posibles para una nueva reina. Esta ramificación supone generar nuevos estados que se forman copiando la lista original y añadiéndole un nuevo elemento. Sólo se efectúa una llamada recursiva a *backtracking* para los estados prometedores. Tras cada llamada a *backtracking* se comprueba si ésta tuvo éxito y encontró una solución factible. Si es así, se devuelve dicha solución inmediatamente. La última línea de *backtracking* sólo se ejecuta cuando no es posible formar una solución factible a partir de la configuración suministrada.

La lista vacía representa al estado inicial, $(?)$, así que resolver el problema es iniciar la búsqueda con ella:

nqueens.py *(cont.)*

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

Probemos el algoritmo calculando una solución factible para cada uno de los problemas de las $n$ reinas con $n$ entre 1 y 8:

```
from nqueens import NQueensSolver1

for n in xrange(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].
```

Nótese que no hay solución factible alguna cuando $n$ es 2 o 3. Mostramos en la figura 6.7 la solución encontrada por el algoritmo para el problema de las 8 reinas.

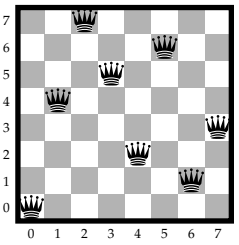



*Figura 6.7: Solución del problema de las 8 reinas encontrada por el algoritmo de búsqueda con retroceso.*

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

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

## 6.1.8. Un algoritmo que genera todas las soluciones factibles

Es sencillo modificar el algoritmo presentado para generar todas las soluciones factibles:

nqueens.py *(cont.)*

```
class AllNQueensSolver(NQueensSolver1):
    def backtracking(self, s):
        if self.is_complete(s): yield s
        for row in xrange(self.n):
            s' = s + [row]
            if self.is_promising(s'):
                for s'' in self.backtracking(s'):
                    yield s''
```

He aquí todas las soluciones del problema con 4, 5 y 6 reinas:

```
1  from nqueens import AllNQueensSolver
2
3  for n in xrange(4, 7):
4      for q in AllNQueensSolver(n).solve(): print q,
5      print '\n'
```

```
[1, 3, 0, 2] [2, 0, 3, 1]

[0, 2, 4, 1, 3] [0, 3, 1, 4, 2] [1, 3, 0, 2, 4] [1, 4, 2, 0, 3] [2, 0, 3, 1, 4
] [2, 4, 1, 3, 0] [3, 0, 2, 4, 1] [3, 1, 4, 2, 0] [4, 1, 3, 0, 2] [4, 2, 0, 3,
 1]

[1, 3, 5, 0, 2, 4] [2, 5, 1, 4, 0, 3] [3, 0, 4, 1, 5, 2] [4, 2, 0, 5, 3, 1]
```

### 6.1.9. Sobre la eficiencia al decidir si un estado es prometedor

Determinar, como hemos hecho, si una estado con $m$ reinas es prometedor comprobando las dos condiciones para todo par de reinas, requiere tiempo $O(m^2)$. En el mejor de los casos, es decir, si el algoritmo genera directamente la solución factible $(s_0, s_1, \ldots, s_{n-1})$ descendiendo directamente por la rama apropiada del árbol, efectuará $n-1$ comprobaciones como ésta y el coste del algoritmo será, pues, $\Omega(n^3)$.

No es necesario efectuar el cálculo completo cuando extendemos un estado de la forma $(s_0, s_1, \ldots, s_{m-1}, ?)$ y generamos un estado, $(s_0, s_1, \ldots, s_{m-1}, s_m, ?)$: basta con comprobar si la última reina añadida a la configuración amenaza a cualquiera de las anteriores, pues ya sabemos que las reinas dispuestas con anterioridad no se amenazan entre sí. La comprobación se puede realizar en tiempo $O(m)$ y el coste para el mejor de los casos del algoritmo completo se reduce entonces a $\Omega(n^2)$. Además, sólo tras la comprobación extenderemos efectivamente el estado. En esta implementación se recoge la idea expuesta y sustituimos la llamada a *is_complete* por el correspondiente cálculo, que es sencillo:

nqueens.py *(cont.)*

```
34 class NQueensSolver2(NQueensSolver1):
35     def is_promising(self, s, row):
36         return all(row != s[i] and len(s)-i != abs(row-s[i]) for i in xrange(len(s)))
37
38     def backtracking(self, s):
39         if len(s) == self.n: return s
40         for row in xrange(self.n):
41             if self.is_promising(s, row):
42                 found = self.backtracking(s+[row])
43                 if found != None: return found
44         return None
```

### 6.1.10. Otro refinamiento del algoritmo

El coste espacial del último algoritmo es mejorable. Cada estado generado requiere espacio $O(n)$, pues se obtiene copiando la memoria de un estado previo y añadiendo la

posición de una nueva reina. La pila de llamadas a función, que puede tener profundidad $O(n)$, ocupará espacio $O(n^2)$.

Pero, en realidad, todas las listas almacenadas en la pila comparten prefijos, pues cada estado se forma añadiendo un componente al estado desde el que se ramifica. Una única lista puede mantener toda la información necesaria. Es más, el perfil de uso de esta lista corresponde a una pila que ofrezca acceso directo a todos sus elementos: cuando ramificamos, apilamos un nuevo elemento, y cuando retrocedemos, desapilamos el útimo elemento; y al comprobar si un estado es o no prometedor, recorreremos sus elementos como si tratara de un vector:

nqueens.py *(cont.)*

```
46 from lifo import Stack
47
48 class NQueensSolver(NQueensSolver2):
49     def __init__(self, n):
50         self.n = n
51
52     def backtracking(self, s):
53         Q = Stack([si for si in s])
54
55         def _backtracking():
56             if len(Q) == self.n: return True
57             for row in xrange(self.n):
58                 if self.is_promising(Q, row):
59                     Q.push(row)
60                     if _backtracking(): return True
61                     Q.pop()
62             return False
63
64         if _backtracking(): return list(Q)
65         return None
```

En cada instante, la pila $Q$ representa a todos los estados en un camino desde la raíz hasta un estado del árbol de estados. Como sólo hay una pila para representarlos a todos, el consumo espacial del algoritmo se reduce a $O(n)$.

EJERCICIOS

**6-1** Dado un tablero con $n \times n$ escaques y $m$ caballos de ajedrez, encuentra una disposición de las piezas de modo que dos caballos no se amenacen mutuamente.

**6-2** Dispón, si es posible, $n$ reinas en un tablero de $n \times n$ escaques de modo que no se amenacen y ninguna de ellas esté en las diagonales principales.

## 6.1.11. Coste

Ya hemos acotado el coste espacial del algoritmo: es $O(n)$. El coste temporal es $\Omega(n^2)$. Resulta difícil acotar superiormente con precisión el coste temporal del algoritmo. Parece que podemos explorar hasta $O(n!)$ hojas. La tabla 6.1 (a) muestra el número de estados (completos e incompletos) explorados para diferentes valores de $n$. Curiosamente,

el número de estados explorados no crece monótonamente con $n$: el algoritmo se comporta mejor con valores impares de $n$ que con valores pares. La tabla 6.1 (b) muestra los problemas ordenados por número de estados explorados.

*Tabla 6.1: (a) Número de estados visitados al resolver el problema de las n reinas. (b) Los mismos datos, pero ordenados según el número de estados visitados.*

| $n$ | **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 |

*(a)*

| $n$ | **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 |

*(b)*

## 6.1.12. Una solución iterativa al problema de la $n$ reinas

La técnica de búsqueda con retroceso se expresa de modo natural en términos recursivos. No obstante, es posible diseñar algoritmos iterativos que efectúen una búsqueda equivalente. Este algoritmo, por ejemplo, resuelve iterativamente el problema de las $n$ reinas:

`nqueens.py` *(cont.)*

```
67 from lifo import Stack
68
69 class IterativeNQueensSolver1(NQueensSolver):
70     def __init__(self, n):
71         self.n = n
72
73     def backtracking(self, s):
74         Q = Stack()
75         for i in xrange(len(s)): Q.push(s[:i])
76         while len(Q) > 0:
```

```
77            s = Q.pop()
78            if len(s) == self.n: return s
79            for row in reversed(xrange(self.n)):
80                if self.is_promising(s, row): Q.push(s+[row])
81        return None
```

En la figura 6.8 se ilustra el estado de la pila $Q$ en cada iteración del bucle **while**, justo antes de ejecutar la línea 74, cuando tratamos de resolver el problema de las 4 reinas.

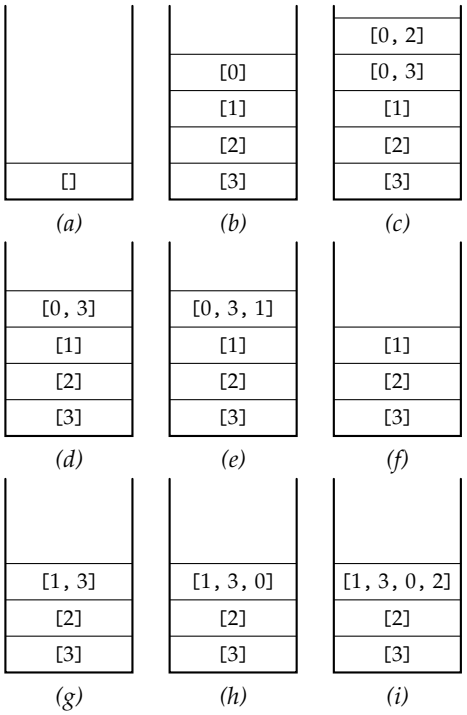



*Figura 6.8: Estado de la pila Q al inicio de cada iteración del bucle* **while**. *(a) Inicialmente sólo se ha apilado el estado raíz, representado con una lista vacía. (b) Al extraer el estado raíz se consideran los 4 valores que puede ocupar la primera reina. Como todos los estados resultantes son prometedores, se apilan. A continuación se ramificará el estado* `[0]` *y los estados* `[1]`, `[2]` *y* `[3]` *no serán explorados hasta que se agote la exploración de estados que se pueden obtener extendiendo* `[0]`. *(c) Tras extraer el estado* `[0]` *se generan, por ramificación, los estados* `[0, 3]`, `[0, 2]`, `[0, 1]` *y* `[0, 0]`. *Sólo los dos primeros son prometedores, por lo que sólo ellos ingresan en la pila. (d) El estado* `[0, 2]`, *que ocupaba la cima de la pila, se ha extraído, pero ninguno de sus descendientes es factible, por lo que no se añade nada a la pila. (e) El estado* `[0, 3]` *ha generado un único descendiente prometedor: el estado* `[0, 3, 1]`. *(f) La ramificación de dicho estado no proporciona nuevos estados prometedores. (g) El estado* `[1]` *se ha ramificado y ha generado un único estado prometedor:* `[1, 3]`. *(h) Dicho estado, tras ramificarse, sólo ha generado un estado prometedor:* `[1, 3, 0]`. *(i) Nuevamente, el estado de la cima se ha ramificado y ha producido un único estado prometedor:* `[1, 3, 0, 2]`. *Cuando se extraiga ese estado, que es completo, el bucle finalizará y se proporcionará el estado como solución.*

Este algoritmo iterativo presenta un par de inconvenientes serios:

- cada estado se forma copiando el estado padre y añadiendo un elemento,
- la pila no sólo contiene los estados de una rama raíz-estado: contiene, además, los hermanos pendientes de consideración para cada estado en dicha rama.

## Un refinamiento: de cada hijo se almacena el último elemento y los valores pendientes

El primer problema se puede superar si no se copia memoria y un único vector representa implícitamente al padre y al hijo. El segundo requiere que enriquezcamos la información del estado con una representación compacta de los hermanos pendientes de análisis.

En la siguiente implementación, cada elemento de la pila $Q$ es un par posición-conjunto: la posición de la última reina dispuesta en el tablero y el conjunto de posiciones para dicha reina que aún no han sido consideradas. Este conjunto se representa con una cola,

que permite extraer las posiciones en un orden determinado. La figura 6.9 muestra el estado de la pila en un instante de la ejecución del algoritmo.

nqueens.py *(cont.)*

```
83 from fifo import FIFO
84
85 class IterativeNQueensSolver2(IterativeNQueensSolver1):
86     def backtracking(self, s):
87         Q = Stack([(si, FIFO()) for si in s])
88         Q.push( (None, FIFO(xrange(self.n))) )
89         while len(Q) > 0:
90             q, b = Q.pop()
91             while len(b) > 0:
92                 q = b.pop()
93                 if self.is_promising([q' for (q', b') in Q], q):
94                     Q.push( (q, b) )
95                     if len(Q) == self.n: return [ q' for (q', b') in Q ]
96                     Q.push( (None, FIFO(xrange(self.n))) )
97                     break
98         return None
```

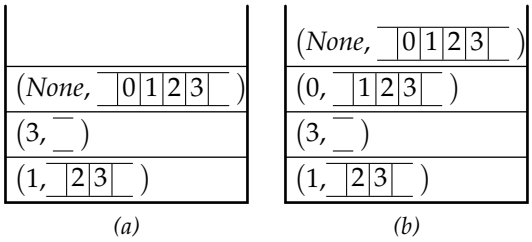



*Figura 6.9: (a) Pila Q en IterativeNQueensSolver2 al explorar el estado* $(1,3,?)$*. Cada celda es un par* $(q,b)$*, donde q es un entero o* None *y b es una cola. Los valores conocidos del estado se encuentran en la primera componente de las diferentes celdas de Q (de abajo hacia arriba). Los valores no considerados aún en la correspondiente posición se hallan en la cola. (b) Estado* $(1,3,0,?)$*, generado a partir del estado anterior.*

## Un refinamiento adicional: los valores pendientes están descritos implícitamente

La representación explícita del conjunto de hermanos no explorados de cada estado de la pila supone que el coste espacial sea $O(n^2)$. Si la última reina de un estado está en la fila $j$, las filas pendientes de consideración para esa reina son $[j+1..n-1]$. Y si $j$ es *None*, las filas pendientes de consideración son $[0..n-1]$. Así pues, no es necesario representar explícitamente la pila de hermanos pendientes de estudio. La figura 6.10 muestra la representación utilizada en este nuevo algoritmo.

nqueens.py *(cont.)*

```
100 class IterativeNQueensSolver(IterativeNQueensSolver2):
101     def backtracking(self, s):
102         Q = Stack(s+[None])
103         while len(Q) > 0:
104             q = Q.pop()
105             if q == None: i = 0
106             else: i = q + 1
```

```
107             while i < self.n:
108                 q, i = i, i+1
109                 if self.is_promising(Q, q):
110                     Q.push(q)
111                     if len(Q) == self.n: return list(Q)
112                     Q.push(None)
113                     break
114         return None
```

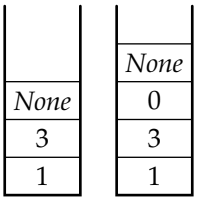



*Figura 6.10: Estado de la pila Q en el algoritmo recursivo IterativeNQueensSolver al explorar el estado* $(1, 3, ?)$*. Cada celda alberga un entero o el valor None. Los valores conocidos del estado se encuentran en las celdas de la pila Q (de abajo hacia arriba). Si la celda i-ésima de la pila contiene el entero j, el conjunto de filas no consideradas para la reina i-ésima es* $[j + 1..3]$*; y si la celda contiene el valor None, dicho conjunto es* $[0..3]$*. (b) Estado* $(1, 3, 0, ?)$ *en la pila Q.*

EJERCICIOS

**6-3** ¿Es correcta esta otra implementación?

nqueens.py *(cont.)*

```
116     def backtracking(self, s):
117         s = s + [-1]
118         while len(s) > 0:
119             if len(s) == n and s[-1] != -1: return s
120             for s[-1] in xrange(s[-1]+1, n):
121                 is_promising = all(s[-1] != s[i] and len()-1-i != abs(s[-1]-s[i])
122                                                                 for i in xrange(len(s)-1))
123                 if is_promising:
124                     s.append(-1)
125                     break
126             if not is_promising: s.pop()
127         return None
```

## 6.2. Esquema algorítmico

El ejemplo anterior ilustra los aspectos fundamentales de la técnica de búsqueda con retroceso. En esta sección presentamos algunos esquemas algorítmicos e introducimos nomenclatura que usaremos en la resolución de otros problemas por el método de búsqueda con retroceso.

Sea $X'$ un conjunto cuyos elementos reciben el nombre de **soluciones** y sea $X \subseteq X'$ un conjunto de elementos definido mediante la imposición de restricciones a los elementos de $X'$. Los elementos de $X$ reciben el nombre de **soluciones factibles**. Un conjunto cualquiera de soluciones recibe el nombre de **estado**. Un conjunto con única solución es

un **estado completo** o **terminal**. Si la única solución de un estado completo es factible, decimos, por extensión, que es un **estado factible**.

La estrategia de búsqueda con retroceso se propone encontrar una solución factible (o enumerar todas las soluciones factibles) siguiendo un proceso ordenado de exploración del conjunto de estados. Una función *branch* : $\mathcal{P}(X) \to (\mathcal{P}(X) - \{\emptyset\})^*$ proporciona una secuencia de estados a partir de un estado cualquiera y se denomina **función de ramificación**. Los estados que se obtienen por ramificación de otro son subconjuntos suyos. El conjunto de estados que se pueden obtener por ramificación reiterada de $X'$ es el **espacio de estados**. La función de ramificación induce una jerarquía en el espacio de estados que puede representarse mediante el denominado **árbol de estados** (el estado sobre el que se aplica la función es el padre y los que forman la secuencia resultante son sus hijos). El método de búsqueda con retroceso propone la exploración del árbol *por primero en profundidad* hasta encontrar un estado completo (que será una hoja) cuya única solución es factible. A tal efecto, se gestiona una pila con todos los estados que se encuentran entre la raíz y el que se está considerando en un instante dado, junto a información que permita saber qué estado apilar cuando se elimina la cima. Esta información puede ser la propia pila de llamadas a función cuando el algoritmo es recursivo o una estructura de datos explícita si se opta por una implementación iterativa.

Un estado es **no prometedor** si sabemos que no contiene solución factible alguna, y es **prometedor** en caso contrario (es decir, si *puede* contener alguna solución factible). La búsqueda con retroceso efectúa *poda por factibilidad*, es decir, no ramifica los estados no prometedores.

Muchos de los problemas abordables por búsqueda con retroceso presentan soluciones estructuradas como secuencias de valores de un conjunto $D$, es decir, $X' \in D^*$ (o, en ocasiones, $X' \in D^n$ para algún valor fijo de $n$). En tales casos es corriente que los estados correspondan a conjuntos de soluciones que comparten un mismo prefijo, $(s_0, s_1, \ldots, s_{m-1})$. El estado se puede representar entonces con dicho prefijo seguido de un interrogante, $(s_0, s_1, \ldots, s_{m-1}, ?)$, indicando así que no es una solución, sino un conjunto de soluciones que se diferencian por el sufijo. Al ramificar $(s_0, s_1, \ldots, s_{m-1}, ?)$ se añade al prefijo un nuevo elemento de $D$ seguido del interrogante y/o, si el prefijo describe una solución completa, se elimina el interrogante.

### 6.2.1. Esquema recursivo

Ya podemos presentar un esquema algorítmico para la búsqueda con retroceso de una solución factible cualquiera:

backtrackingscheme.py

```
1 class BacktrackingScheme:
2     def is_complete(self, s):
3         """Devuelve cierto sii s es un estado completo y falso en caso contrario."""
4 
5     def is_feasible(self, s):
6         """Devuelve cierto sii la única solución de s es factible y falso en caso contrario."""
7 
```

```
8     def is_promising(self, s):
9         """Devuelve falso sii es imposible que s contenga una solución factible."""
10
11    def branch(self, s):
12        """Devuelve una secuencia de estados (que puede ser vacía) obtenidos por
13            ramificación de s."""
14
15    def backtracking(self, s):
16        """Explora el subárbol que tiene por raíz a s y devuelve la primera solución factible
17            que encuentra (si la hay). Si s no contiene al menos una solución factible,
18            devuelve None"""
19        if self.is_complete(s):
20            if self.is_feasible(s):
21                return s
22        else:
23            for s' in self.branch(s):
24                if self.is_promising(s'):
25                    found = self.backtracking(s')
26                    if found != None: return found
27        return None
28
29    def solve(self, s):
30        return self.backtracking([])
```

Si se está interesado en enumerar todas las soluciones factibles, se puede recurrir a este otro método:

backtrackingscheme.py *(cont.)*

```
33 class AllSoluctionsBacktrackingScheme(BacktrackingScheme):
34     def backtracking(self, s):
35         """Explora el subárbol que tiene por raíz a s y genera todas las soluciones factibles
36             que encuentra (si las hay)."""
37         if self.is_complete(s):
38             if self.is_feasible(s):
39                 yield s
40         else:
41             for s' in self.branch(s):
42                 if self.is_promising(s'):
43                     for s'' in self.backtracking(s'):
44                         yield s'
```

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.

EJERCICIOS

**6-4** ¿Qué configuración del problema de las $n$ reinas hace que la suma de distancias de cada reina a la siguiente (diferencia del número de fila en la que se encuentra cada una) sea mínima?

### 6.2.2. Esquema iterativo

Es posible diseñar un algoritmo iterativo a partir del algoritmo recursivo. Una pila permite llevar cuenta de la rama del árbol en la que nos encontramos en cada instante:

backtrackingscheme.py *(cont.)*

```
46 from lifo import Stack
47
48 class IterativeBacktrackingScheme(BacktrackingScheme):
49     def backtracking(self, s):
50         Q = Stack([s])
51         while len(Q) > 0:
52             s = Q.pop()
53             if self.is_complete(s):
54                 if is_feasible(s): return s
55             else:
56                 for s' in self.branch(s):
57                     if self.is_promising(s'): Q.push(s')
58         return None
59
60 class IterativeAllSolutionsBacktrackingScheme(BacktrackingScheme):
61     def backtracking(self, s):
62         Q = Stack([s])
63         while len(Q) > 0:
64             s = Q.pop()
65             if self.is_complete(s):
66                 if is_feasible(s): yield s
67             else:
68                 for s' in self.branch(s):
69                     if self.is_promising(s'): Q.push(s')
```

### 6.2.3. Corrección

Para que el esquema algorítmico recursivo conduzca a un procedimiento correcto la función de ramificación debe satisfacer ciertas condiciones.

**Condición 6.1** *Para todo estado $s$ con $|s| > 1$ y para todo conjunto de estados $s' \in branch(s)$, se cumple $s' \subset s$.*

**Condición 6.2** *Para todo estado $s$ con $|s| > 1$ se cumple $\bigcup_{s' \in branch(s)} s' = s$.*

La Condición 6.1 nos dice que la ramificación de un estado incompleto produce subconjuntos propios y la Condición 6.2 implica que si una solución factible pertenece a un conjunto, debe pertenecer a alguno de los conjuntos resultantes de la ramificación. Con estas condiciones podemos demostrar el siguiente teorema:

**Teorema 6.1** *Si la cardinalidad del estado inicial, $X'$, es finita, entonces el esquema algorítmico de búsqueda con retroceso termina.*

*Demostración.* Como el procedimiento de ramificación genera subconjuntos propios (Condición 6.1), los estados en un camino de la raíz a las hojas son conjuntos de cardinalidad decreciente y como $X'$, el estado raíz, es finito, la altura del árbol también lo es. El algoritmo no puede, pues, entrar en recursión sin fin.

Si existe alguna solución factible y aún no se ha visitado ningún estado completo que la represente, la Condición 6.2 garantiza que dicha solución se encuentra en alguno de los estados no explorados, pues al menos uno de los estados resultantes de la ramificación contiene cualquiera de las soluciones factibles contenidas en el estado raíz, $X'$.

El algoritmo es un recorrido por primero en profundidad de un árbol finito, así que, si hay alguna solución factible, acaba por explorar alguna de las hojas cuyo estado es un conjunto unitario formado por una solución factible. Si no hay solución factible, en algún momento se habrán visitado todos los estados del árbol y el algoritmo finalizará. □

Hemos demostrado la terminación (y también la corrección) del algoritmo cuando $X'$ es de cardinalidad finita. Si no lo fuera, es posible que una estrategia de búsqueda con retroceso encontrara una solución factible, pero habría de demostrarse.

EJERCICIOS

**6-5** Demuestra que el algoritmo desarrollado para resolver el problema de las $n$ reinas satisface las Condiciones 6.1 y 6.2.

**6-6** Si $X'$ es de talla infinita y toda solución factible es de talla finita, ¿encuentra siempre la solución un algoritmo de búsqueda en el árbol de estados que efectúe un recorrido por primero en anchura?

## 6.2.4. Coste

El coste temporal de los algoritmos de búsqueda con retroceso depende del tamaño del árbol de estados, que puede ser exponencial, y del orden de ramificación, y el coste espacial depende de la profundidad máxima del árbol y de la representación de los estados.

### Tiempo

El esquema algorítmico propone una exploración por primero en profundidad de un árbol de estados. Como su tamaño suele ser exponencial en función de la talla del problema, es frecuente que el coste temporal de los algoritmos con retroceso resulte prohibitivo.

En cualquier caso tiene interés prestar atención al modo en que se efectúa la determinación de si una solución incompleta es o no prometedora y de si un estado terminal denota una solución factible o no: frecuentemente se puede efectuar el cálculo incrementalmente, es decir, aprovechando los cálculos ya realizados para estados con un prefijo común al que se estudia en ese instante.

Hay una condición no exigible a la función de ramificación, pero sí deseable por su impacto en la eficiencia computacional del algoritmo:

**Condición 6.3 (opcional)** *Para todo estado $s$ con $|s| > 1$ y para todo par de estados $s'$ y $s''$ en $branch(s)$, se cumple que $s' \cap s'' = \emptyset$.*

Esta condición, unida a las anteriores, garantiza que la ramificación particiona el conjunto representado por un estado. Ello contribuye a reducir efectivamente el espacio de búsqueda.

EJERCICIOS

**6-7** ¿Satisface también la Condición 6.3 el algoritmo de búsqueda con retroceso para el problema de las $n$ reinas?

Un factor que puede ser de extrema importancia es el orden en el que se ramifican los estados. Un principio general es que *interesa ramificar de modo que el número de hijos prometedores sea el menor posible*. Consideremos el problema de las $n$ reinas, donde hemos procedido a ubicar reinas por columnas, de izquierda a derecha. Al ubicar la primera reina en la primera columna hemos empezado considerando la primera fila, pero se puede reducir el número de hijos que vamos a explorar a continuación si en lugar de considerar la primera fila hubiésemos considerado la segunda, como muestra la figura 6.11. Es más, resultaría más eficiente aún considerar que la primera reina no se disponga en la primera columna, sino en una columna (y fila) central, pues el número de casillas amenazadas es mayor si se ubica ahí y, por tanto, se reduce el número de posibles ubicaciones para las siguientes reinas.

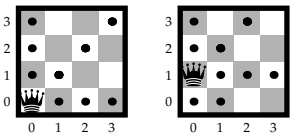



*Figura 6.11: En el problema de las 4 reinas, considerar la primera fila en primer lugar es peor que considerar una fila central. A mano izquierda podemos ver que empezar disponiendo la primera reina en la primera fila deja dos opciones para la segunda reina. Si hubiésemos empezado por la segunda fila, como se muestra a mano derecha, sólo hubiésemos dejado una posible ubicación para la segunda reina.*

Debe tenerse en cuenta que la eficiencia de los algoritmos de búsqueda con retroceso depende del orden en el que se generan los estados por ramificación. Experimentalmente se ha observado que conviene, por regla general, que los niveles superiores del árbol presenten un factor de ramaje bajo.

Las preguntas clave acerca de la complejidad computacional, no obstante lo dicho, guardan relación con el propio proceso de búsqueda. ¿Sería mejor una exploración diferente del árbol de estados? ¿Por primero en anchura? ¿Por prioridad basada en algún cálculo sobre los estados? ¿Con la garantía de acertar? Estas estrategias de exploración alternativas dan pie a algoritmos de búsqueda de diferente naturaleza, como los algoritmos «voraces» o los algoritmos de «ramificación y poda», que estudiaremos más adelante.

## Espacio

Hay dos «fuentes» de ocupación espacial en los algoritmos recursivos de búsqueda con retroceso: el espacio necesario para la tupla con que se representa cada estado y el espacio que requiere la pila de las llamadas recursivas. Sobre la primera fuente de ocupación espacial cabe decir que conviene mantener en memoria una sola variable para la tupla. De este modo es posible diseñar estrategias para que con cada llamada recursiva se añada eficientemente información a la misma y se suprima, también eficientemente, al efectuar

un retroceso. Una pila que ofrezca acceso a todos sus elementos o una simple lista pueden ser implementaciones adecuadas para una tupla en la que sólo se requiera añadir por el final y eliminar elementos del mismo extremo. Sobre la segunda fuente de consumo de memoria podemos indicar que, si el algoritmo es recursivo, necesitaremos un número de registros de activación que es proporcional a la tupla de mayor longitud explorada. El consumo de cada registro dependerá, naturalmente, de si contiene variables locales de dimensión variable.

## 6.3. 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* $W$.

Podemos describir un estado completo mediante 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, $2^N$ soluciones completas, así que un algoritmo de fuerza bruta podría enumerarlas todas y seleccionar una factible en tiempo $O(2^N)$. Un estado completo es factible si $\sum_{1 \le i \le N} x_i w_i = W$. Un estado no terminal es un vector con $m < N$ componentes y lo representaremos con $(x_1, x_2, \ldots, x_m, ?)$.

Observa la ventaja de representar con 1 o 0 la inclusión o descarte de cada objeto: podemos multiplicar el peso de cada objeto por su respectivo 1 o 0 para saber cuánto cargamos en la mochila. Así, la expresión $\sum_{1 \le i \le N} x_i w_i$ representa la carga que supone la solución representada con el vector $(x_1, x_2, \ldots, x_N)$.

Cuando $m < N - 1$, la ramificación de un estado $(x_1, x_2, \ldots, x_m, ?)$ produce dos nuevos estados: $(x_1, x_2, \ldots, x_m, 0, ?)$ y $(x_1, x_2, \ldots, x_m, 1, ?)$. Y cuando $m = N - 1$, produce dos estados completos $(x_1, x_2, \ldots, x_{N-1}, 0)$ y $(x_1, x_2, \ldots, x_{N-1}, 1)$.

La figura 6.12 muestra el árbol de estados completo para un problema concreto: la carga de una mochila de capacidad 7 con objetos de pesos 4, 5, 3 y 6. La única solución factible, $(1, 0, 1, 0)$, aparece marcada con un punto.

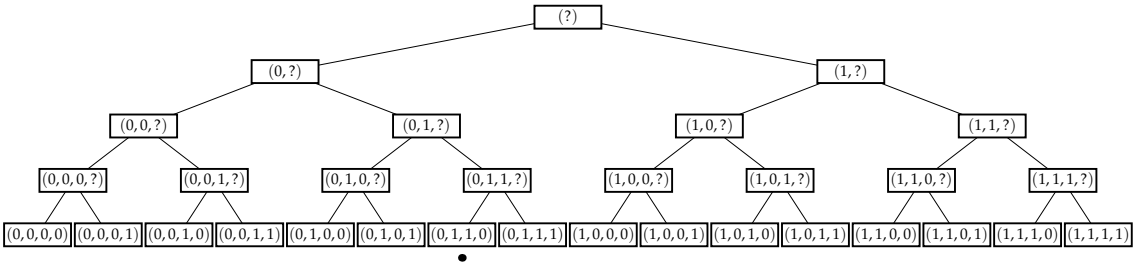



*Figura 6.12: Árbol de estados completo para una instancia del problema de la suma del subconjunto con* $w_1 = 4$, $w_2 = 5$, $w_3 = 3$, $w_4 = 6$ *y* $W = 8$.

Un estado es prometedor si su suma de pesos es menor o igual que $W$. Al ramificar un estado incompleto prometedor $(x_1, x_2, \ldots, x_m, ?)$ obtenemos uno o dos nuevos estados

prometedores, uno para el caso en que no se incluye el objeto de índice $m$ y *posiblemente* otro para el caso en que sí. El árbol de estados 6.13 muestra el efecto de reducción del espacio de búsqueda derivado de no ramificar estados no prometedores.

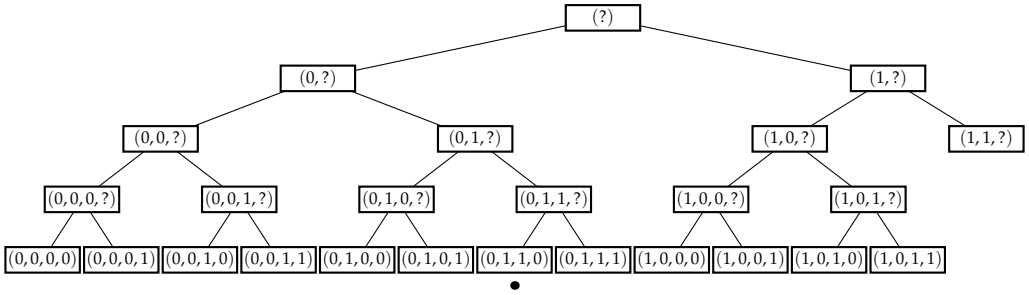



*Figura 6.13: Árbol de estados sin ramificación de estados no prometedores para una instancia del problema de la suma del subconjunto con $w_1 = 4$, $w_2 = 5$, $w_3 = 3$, $w_4 = 6$ y $W = 8$.*

Pero el método de búsqueda con retroceso ni siquiera visita todos los estados de ese árbol; se limita a visitar los que se muestran en el árbol de la figura 6.14: el proceso termina tan pronto se encuentra una solución factible.

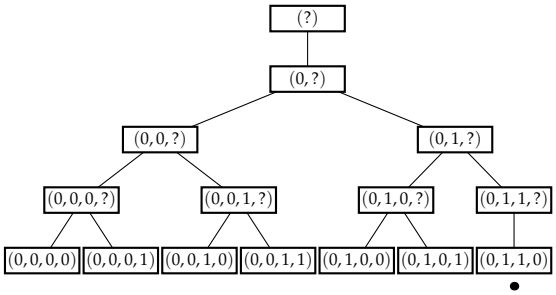



*Figura 6.14: Árbol de estados visitados en la búsqueda con retroceso para una instancia del problema de la suma del subconjunto con $w_1 = 4$, $w_2 = 5$, $w_3 = 3$, $w_4 = 6$ y $W = 8$.*

Disponemos ya de suficientes elementos como para diseñar un primer algoritmo de búsqueda con retroceso:

subset_sum1.py

```
1  from offsetarray import OffsetArray
2
3  def subset_sum(w, W):
4      x = OffsetArray([0] * len(w))
5      def backtracking(i):
6          if i == len(w)+1:
7              if sum([w[j]*x[j] for j in xrange(1, len(w)+1)]) == W: return x
8          else:
9              for x[i] in 0, 1:
10                 if sum([w[j]*x[j] for j in xrange(1, i+1)]) <= W:
11                     found = backtracking(i+1)
12                     if found != None: return found
13         return None
14     return backtracking(1)
```

```
1 from subset_sum1 import subset_sum
2 from offsetarray import OffsetArray
3
4 W, w = 8, OffsetArray([4,5,3,6])
5 print 'Selección con pesos %s y capacidad %d: %s' % (w, W, subset_sum(w, W))
```

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

Nótese que en este problema un estado completo y prometedor puede no corresponder a una solución factible: para que la solución que representa sea factible, la suma de pesos de objetos considerados debe ser igual a $W$, y no sólo menor o igual. Ello obliga a comprobar que la solución representada por un estado terminal es factible.

### 6.3.1. Corrección

Nótese que $X'$ es de cardinalidad finita (está acotada superiormente por $2^N$) y que el procedimiento de ramificación observa las Condiciones 6.1 y 6.2, así que el algoritmo termina y tiene garantizada la corrección.

### 6.3.2. Coste y algunos refinamientos

El algoritmo presenta un coste temporal acotado superiormente por $O(2^N)$, pues el número de estados potencialmente prometedores es de ese orden. Esta cota temporal no puede reducirse, pero sí es posible refinar el algoritmo y mejorar diversos aspectos que afectan a su velocidad de ejecución en la práctica:

- Conviene considerar los objetos ordenados de menor a mayor peso porque ello permite acelerar la calificación de un estado $(x_1, x_2, \ldots, x_m, ?)$ como no prometedor: si estudiamos el valor de $x_{m+1}$, que es el objeto de menor peso de cuantos quedan y cuya inclusión se va estudiar a continuación, sabemos que si no cabe, no cabrá ningún otro de los que le siguen en el orden considerado, pues son de peso mayor o igual.

- En lugar de recalcular para cada estado el peso de los objetos ya cargados, podemos enriquecer los estados con la capacidad de carga disponible en la mochila.

- No es necesario esperar a tener una solución completa para finalizar la búsqueda: si una solución parcial tiene peso $W$, hay una solución completa trivial: la que no añade ningún otro objeto.

- Por otra parte, conviene considerar antes la inclusión de un objeto en la mochila que su no inclusión. De ese modo es más probable que exploremos menos nodos del árbol de estados porque las ramas de más a la izquierda (las que primero recorre el algoritmo en su exploración por primero en profundidad) presentarán, probablemente, menor número de estados y menor altura (ya que la mochila presentará una capacidad menor conforme se desciende por ramas de la izquierda).

- Un estado puede considerarse no prometedor si la suma de los pesos de los objetos aún no considerados es menor que el peso que nos queda por cargar. Podemos precalcular, para cada $i$ entre 1 y $N$ y con un coste total $O(N)$, el peso total de los objetos $w_{i+1}, w_{i+2}, \ldots, w_N$.

subset_sum.py

```
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 xrange(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 xrange(i, 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)
```

```
from subset_sum import subset_sum
from offsetarray import OffsetArray

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

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

La figura 6.15 ilustra el aspecto del árbol de estados que subyace a la nueva búsqueda. Si se compara este árbol con el de la figura 6.13 se puede apreciar la enorme reducción del espacio de búsqueda. La mejora es espectacular si comparamos los estados visitados por uno y otro algoritmo (compárese la figura 6.14 con los estados representados en trazo grueso en la figura 6.15): el primero visita 15 y el segundo sólo 5.

A efectos de ilustrar lo efectivo de la reducción del número de estados visitados hemos realizado un pequeño experimento: hemos generado 100 instancias sintéticas del problema (todas ellas con solución) para cada valor de $N$ entre 1 y 15. La tabla 6.2 muestra el número de llamadas a *backtracking* (que es el número de estados visitados) con el

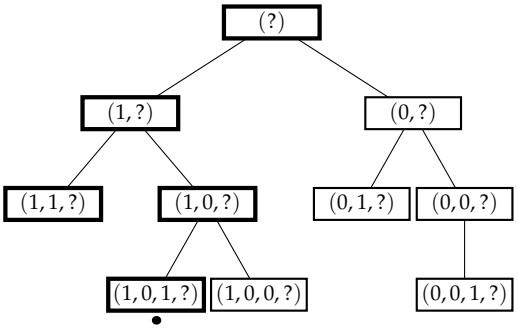



*Figura 6.15: Árbol de estados (sin ramificación de estados no prometedores) para una instancia del problema de la suma de subconjuntos con pesos (ordenados crecientemente) $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 6$ y $W = 8$. En este árbol, algunas soluciones parciales se consideran completas en tanto que conducen a una solución factible trivial: basta para formarla con no incluir más objetos. En trazo grueso, estados efectivamente explorados.*

algoritmo original y con el refinado, así como el porcentaje que supone la segunda cantidad sobre la primera. Se puede apreciar que la efectividad de la reducción del número de llamadas es tanto mayor cuanto mayor es la talla del problema.

*Tabla 6.2: 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 | 1 051 | 429 | 40.8 |
| 4 | 1 897 | 570 | 30.0 |
| 5 | 3 395 | 839 | 24.7 |
| 6 | 5 428 | 1 168 | 21.5 |
| 7 | 11 660 | 1 424 | 12.2 |
| 8 | 18 899 | 1 931 | 10.2 |
| 9 | 30 934 | 2 072 | 6.7 |
| 10 | 54 353 | 2 456 | 4.5 |
| 11 | 99 718 | 2 979 | 3.0 |
| 12 | 216 248 | 3 511 | 1.6 |
| 13 | 359 453 | 3 493 | 1.0 |
| 14 | 597 507 | 3 966 | 0.7 |
| 15 | 1 298 615 | 3 587 | 0.3 |

Aunque supone una reducción efectiva del tiempo de ejecución, sólo podemos decir de su coste temporal que es $O(2^N)$.

EJERCICIOS

**6-8** Analiza el coste espacial del algoritmo.

**6-9** Repite el experimento cuyo resultado se muestra en la tabla 6.2 para instancias del problema que no tienen solución.

**6-10** Haz una traza para una mochila con capacidad para 8 kilogramos y 4 objetos de pesos 2, 3, 5 y 6 kilogramos, respectivamente. Dibuja el árbol de estados visitados.

**6-11** Diseña un algoritmo de búsqueda con retroceso que cuente el número de soluciones válidas para una colección de objetos. Prueba el algoritmo con el ejemplo del ejercicio anterior.

**6-12** Disponemos de $N$ objetos con pesos $w_1, w_2, \ldots, w_N$ que deseamos colocar en los dos platos de una balanza. ¿Es posible repartirlos de modo que ambos platos pesen exactamente lo mismo?

**6-13** Tenemos una mochila con capacidad para cargar $W$ unidades de peso y $N$ objetos que podemos cargar en ella. Cada objeto tiene un peso $w_i \in \mathbb{Z}^{>0}$ y un valor $v_i \in \mathbb{R}$, para $1 \leq i \leq N$.

El beneficio que aporta cargar el objeto $i$ es su valor $v_i$. ¿Qué objetos seleccionamos para cargar la mochila sin exceder su capacidad y de modo que el beneficio sea máximo? Diseña un algoritmo de búsqueda con retroceso que resuelva este problema (conocido como problema de la mochila discreta).

**6-14** Dado un (multi)conjunto con $N$ enteros, $A = \{x_1, x_2, \ldots, x_N\}$, deseamos saber si hay alguna forma de particionarlo en subconjuntos cuya suma de elementos sea exactamente $M$. Por ejemplo, sea $A = \{1, 2, 2, 3, 4\}$ y sea $M = 4$; el conjunto $\{\{1, 3\}, \{2, 2\}, \{4\}\}$ es una partición de $A$ tal que todos los subconjuntos que la forman suman 4.

Queremos diseñar un algoritmo de búsqueda con vuelta atrás. Debes describir las soluciones factibles mediante $N$-tuplas, indicando claramente las restricciones que deben satisfacerse. Describe, además, la representación de los estados y el proceso de ramificación.

## 6.4. Recubrimiento con poliominós

Si se juntan lado con lado cuatro baldosas cuadradas de todas las formas posibles, sin tener en cuenta semejanzas por simetría, se obtienen las cinco piezas que se muestran en la figura 6.16 (a). Cada una de ella recibe un nombre en función de su semejanza con una letra del alfabeto: `I`, `O`, `L`, `T` y `Z`. Estas piezas son los denominados tetrominós. Si consideramos las diferentes figuras que podemos obtener por rotación y reflexión de los cinco tetrominós obtenemos las 20 piezas de la figura 6.16 (b).

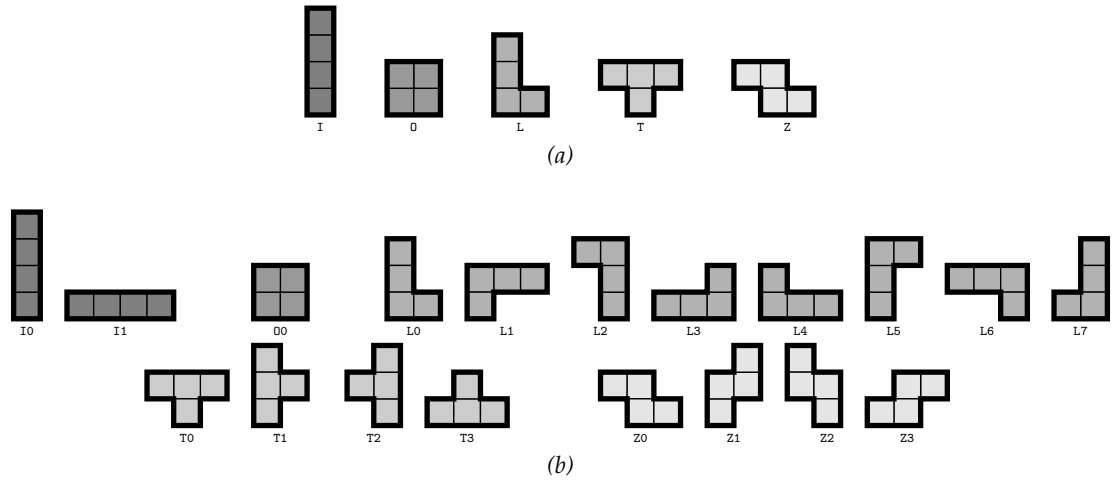


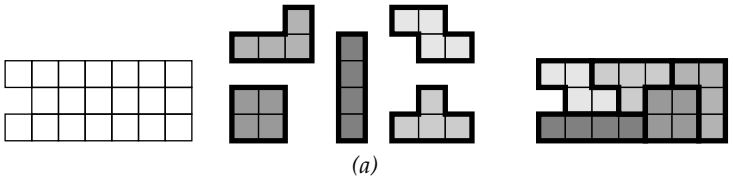

*Figura 6.16: (a) Los cinco tetrominós. (b) Figuras equivalentes bajo simetría de cada uno de los 5 tetrominós.*

Los tetrominós son un caso particular de poliominós: los dominós son poliominós formados por dos baldosas; los triominós, por tres; los tetrominós, por cuatro; los pentominós, por cinco; los hexominós, por seis, y así sucesivamente.

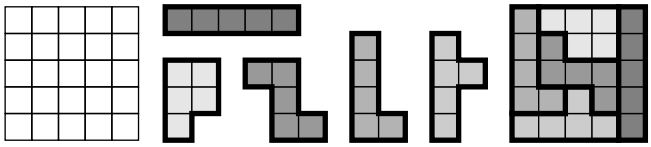
El «recubrimiento con poliominós» es un conocido pasatiempo: dado un tablero y un conjunto de poliominós, encuéntrese alguna forma de azulejar el tablero con los poliominós de modo que cada pieza del conjunto aparezca en cualquiera de sus formas

simétricamente equivalente y sin usar ninguna de las piezas más de una vez. La figura 6.17 muestra dos instancias del problema del recubrimiento con poliominós, una con tetrominós y otra con pentominós.

Podemos abordar el problema mediante la técnica de búsqueda con retroceso. Centraremos la exposición en los tetrominós, aunque es perfectamente posible gestionar otros conjunto de piezas.



*(a)*



*(b)*

*Figura 6.17: (a) Un problema de recubrimiento con tetrominós: a la izquierda se muestra un tablero y los 5 tetrominós. A la derecha, el tablero cubierto con las piezas. (b) Ídem para un problema con pentominós.*

Los tetrominós son muy conocidos por ser las piezas de un famoso videojuego: el Tetris. Los poliominós fueron inventados por Solomon Golomb en 1954. Los juegos con poliominós son uno de los temas favoritos de la matemáticas recreativas. Un problema clásico con poliominós es la determinación del número de $n$-minós. Hay 1 dominó, 2 triominós, 5 tetrominós, 12 pentominós, 35 hexominós, 107 heptominós, 363 octominós, 1 248 nonominós, 4 460 decominós... No se conoce fórmula alguna que permita saber cuántos $n$-minós hay.

## 6.4.1. Representación de los poliominós y del tablero

Cada rotación y simetría especular de un poliominó puede codificarse con una secuencia de pares que indican las coordenadas, en relación a algún punto de referencia, de cada uno de los cuadrados. El tetrominó `OO`, por ejemplo, puede codificarse con la lista `[(0,0),(1,0),(0,1),(1,1)]`.

Todas las rotaciones y simetrías especulares de una misma pieza puede formar parte de una lista asociada al identificador de la pieza. Una tupla con un identificador de pieza y su lista de rotaciones y simetrías vale para representar un conjunto de poliominós. El conjunto de los cinco tetrominós con todas sus presentaciones (rotaciones y simetrías especulares) queda así:

tetrominos.py

```
1 tetrominos = (('I', [[(0,0),(1,0),(2,0),(3,0)], [(0,0),(0,1),(0,2),(0,3)]]),
2               ('O', [[(0,0),(1,0),(0,1),(1,1)]]),
3               ('L', [[(0,0),(1,0),(2,0),(2,1)], [(0,0),(0,-1),(0,-2),(1,-2)],
4                      [(0,0),(-1,0),(-2,0),(-2,-1)], [(0,0),(0,1),(0,2),(-1,2)],
5                      [(0,0),(0,-1),(0,-2),(-1,-2)], [(0,0),(-1,0),(-2,0),(-2,1)],
6                      [(0,0),(0,1),(0,2),(1,2)], [(0,0),(1,0),(2,0),(2,-1)]]),
```

```
7             ('T', [[(0,0),(0,1),(0,2),(1,1)], [(0,0),(1,0),(2,0),(1,1)],
8                    [(0,0),(-1,0),(-2,0),(-1,-1)], [(0,0),(0,1),(0,2),(-1,1)]]),
9             ('Z', [[(0,0),(0,1),(1,1),(1,2)], [(0,0),(1,0),(1,-1),(2,-1)],
10                   [(0,0),(1,0),(1,1),(2,1)], [(0,0),(0,1),(-1,1),(-1,2)]]))
```

Por otra parte, el tablero puede codificarse como una lista de filas, cada una de las cuales es una cadena que codifica qué columnas pueden ser ocupadas con un cuadrado y cuáles no. El tablero de la figura 6.17 (a) se puede proporcionar como una cadena de asteriscos y puntos:

```
1 "+++++++\n*++++++\n+++++++"
```

Los asteriscos indican casillas que no pueden ser ocupadas y los puntos representan casillas que se pueden ocupar. La primera letra de cada identificador de pieza puede usarse para señalar las casillas ocupadas por la piezas en el tablero. Dado que las cadenas son inmutables, codificaremos los tableros como listas de listas de caracteres. Esta disposición de fichas corresponde a la que se muestra en la figura 6.18.

```
1 [['T','T','T','+','L','L','L'],
2  ['*','T','+','+','L','O','O'],
3  ['+','+','+','+','+','O','O']]
```

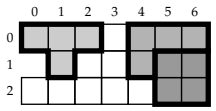



*Figura 6.18: Una configuración de tablero.*

El constructor del resolutor de problemas de recubrimiento recibe como datos un tablero y una relación de poliominós:

polyominos_tiling.py

```
1 class PolyominosTiling:
2     def __init__(self, board, polyominos):
3         self.board = [list(row) for row in board.split('\n')]
4         self.pieces = polyominos
```

## 6.4.2. Estados

Un estado es una configuración del tablero con cero o más piezas dispuestas en él. Podemos describir una configuración mediante una tupla en la que cada componente corresponde a una de las piezas. Si hemos de disponer un total de $N$ piezas, la tupla tendrá también $N$ elementos. Si la tupla codifica la pieza (con su orientación) y las coordenadas en las que se dispone (fila y columna) la casilla de referencia de cada pieza (la que tiene coordenadas relativas $(0,0)$), una configuración como la de la figura 6.18 podría describirse con la tupla como $((\texttt{OO},1,5),(\texttt{L1},0,4),(\texttt{T1},0,0),?)$.

En cada instante sólo es necesario mantener un estado en memoria: el que se visita en ese momento. Nos conviene, eso sí, disponer de funciones que nos ayuden a:

- determinar si una pieza cabe o no en unas coordenadas dadas,

- disponer un pieza en unas coordenadas dadas,
- eliminar una pieza de unas coordenadas dadas.

La representación con tuplas, por sí sola, resulta incómoda: cada vez que deseemos saber si una nueva pieza cabe o no en ciertas coordenadas, tendremos que efectuar un cálculo complicado y/o recomponer el tablero para saber qué casillas del mismo están libres. Mantendremos un tablero que recoja la ocupación en cada instante y que simplifique la detección de conflictos al tratar de ubicar fichas en el tablero. El atributo *self.board* puede jugar ese papel. Estas funciones se encargan de efectuar estas tres acciones:

polyominos_tiling.py *(cont.)*

```
7     def fits(self, piece, x, y):
8         return all(0 <= j+y < len(self.board) and 0 <= i+x < len(self.board[j+y]) and\
9                    self.board[j+y][i+x] == '+' for (i,j) in piece)
10
11    def put(self, symbol, piece, x, y):
12        for (i,j) in piece: self.board[j+y][i+x] = symbol
13
14    def remove(self, piece, x, y):
15        for (i,j) in piece: self.board[j+y][i+x] = '+'
```

### 6.4.3. Ramificación

Al ramificar un estado pasamos a considerar toda posible ubicación de la siguiente pieza en el orden dado. La configuración de la figura 6.18, por ejemplo, da lugar a estos nuevos estados:

```
1  [['T','T','T','+','L','L','L'],
2   ['*','T','Z','Z','L','O','O'],
3   ['+','+','+','Z','Z','O','O']]
4
5  [['T','T','T','+','L','L','L'],
6   ['*','T','Z','Z','L','O','O'],
7   ['+','Z','Z','+','+','O','O']]
8
9  [['T','T','T','Z','L','L','L'],
10  ['*','T','Z','Z','L','O','O'],
11  ['+','+','Z','+','+','O','O']]
```

Hemos de considerar, pues, que la siguiente figura puede aparecer en cualquiera de las posiciones y en cualquiera de sus formas simétricas.

Es evidente que la función de ramificación observa las dos condiciones exigibles para garantizar la corrección del método de búsqueda con retroceso.

### 6.4.4. Algoritmo recursivo

Resulta fácil, a tenor del análisis efectuado, instanciar el esquema de búsqueda con retroceso a nuestro problema:

polyominos_tiling.py *(cont.)*

```
17    def backtracking(self, i, cells):
18        if i == len(self.pieces):
19            if cells == 0: return self.board
20        else:
21            name, polyominos = self.pieces[i]
22            for polyomino in polyominos:
23                for y in xrange(len(self.board)):
24                    for x in xrange(len(self.board[y])):
25                        if self.fits(polyomino, x, y):
26                            self.put(name, polyomino, x,y)
27                            found = self.backtracking(i+1, cells-len(polyomino))
28                            if found != None: return found
29                            self.remove(polyomino, x,y)
30        return None
31
32    def solve(self):
33        return self.backtracking(0, sum(cell == '+' for row in self.board for cell in row))
```

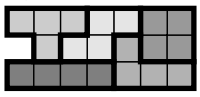
```
1  from polyominos_tiling import PolyominosTiling
2  from tetrominos import tetrominos
3
4  solution = PolyominosTiling('+++++++\n*++++++\n+++++++', tetrominos).solve()
5  if solution != None: print '\n'.join(''.join(line) for line in solution)
```

```
IIIIOOL
*TZZOOL
TTTZZLL
```

La corrección del algoritmo está garantizada al cumplir la función de ramificación con las condiciones exigibles y ser finito el espacio de búsqueda.

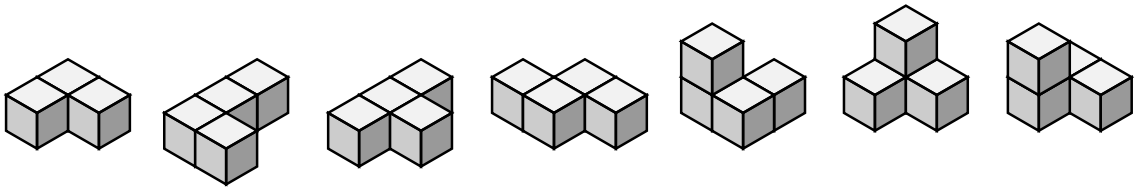
. . . . . EJERCICIOS . . . . .

**6-15** El problema del recubrimiento con poliominós puede admitir más de una solución factible. El problema de la figura 6.17, por ejemplo, admite esta otra solución:

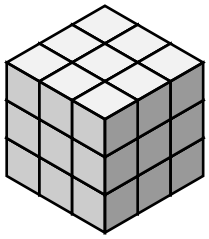


Diseña un algoritmo de búsqueda con retroceso que cuente el número de soluciones factibles.

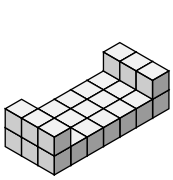
**6-16** *Soma* es un rompecabezas tridimensional que se juega con estas 7 piezas:

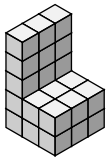


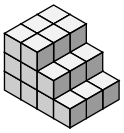
El objetivo del juego es formar figuras tridimensionales, como este cubo de $3 \times 3 \times 3$:

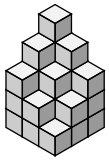


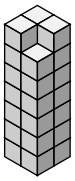
He aquí otras figuras que puedes formar con las piezas de *Soma*:

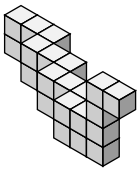

La cama

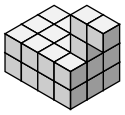

La silla

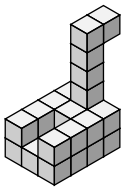

La escalera

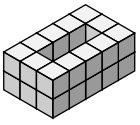

El cristal

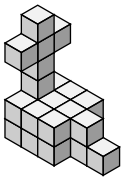

La torre

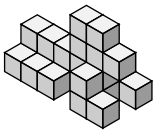

La serpiente

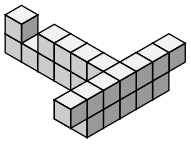

El pato

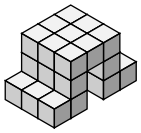

El cisne

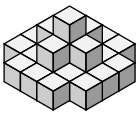

La bañera

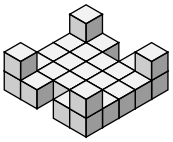

La tumba

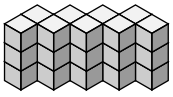

El perro


El avión


El tunel


La pirámide


El castillo


El zig-zag

Diseña un algoritmo de búsqueda con retroceso que, dada una figura, utilice las 7 piezas de *Soma* para construirla y nos indique cómo hemos de disponer cada una de ellas.

## 6.5. El 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. La figura 6.19 muestra un grafo y un ciclo hamiltoniano sobre él.


El denominado «problema del viajante de comercio» está relacionado con este problema. En él se propone la búsqueda del ciclo hamiltoniano de peso mínimo en un grafo ponderado.

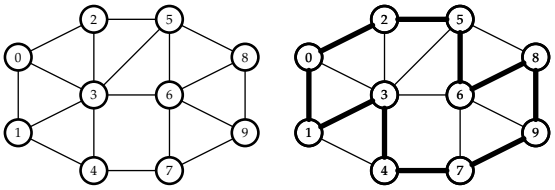



*Figura 6.19: (a) Grafo sobre el que se plantea la búsqueda de un ciclo hamiltoniano. (b) Un ciclo hamiltoniano en el grafo.*

Un ciclo hamiltoniano queda descrito con una permutación de los vértices (el último elemento es igual que el primero, por lo que podemos no proporcionarlo explícitamente). Hay, por tanto, $|V|!$ ciclos hamiltonianos posibles. Un algoritmo basado en la fuerza bruta puede generar todas las permutaciones y comprobar, para cada una, si todo par de vértices consecutivos es una arista del grafo.

### 6.5.1. Algoritmo de búsqueda con retroceso

Es fácil diseñar un algoritmo que siga la estrategia de búsqueda con retroceso. Nótese que no tenemos por qué fijar un vértice inicial: el ciclo hamiltoniano, si existe, pasa por todos los vértices y cualquiera puede considerarse el primero del camino. El algoritmo mantiene un estado en cada instante al que va añadiendo un nuevo vértice con cada llamada recursiva (y que elimina convenientemente cuando efectúa el retroceso) tras comprobar que no había sido visitado. El estado será terminal si la tupla contiene todos los vértices (una sola vez). Y será, además, factible si existe una arista que una el primer vértice con el último:

hamiltonian_cycle1.py

```
1  from random import sample
2
3  def hamiltonian_cycle(G):
4      def backtracking(path):
5          if len(path) == len(G.V):
6              if (path[-1], path[0]) in G.E: return path+[path[0]]
7          else:
8              for v in G.succs(path[-1]):
9                  if v not in path:
10                     found = backtracking(path+[v])
11                     if found: return found
12         return None
13     [random_vertex] = sample(G.V, 1) # Selecciona un vértice al azar.
14     return backtracking([random_vertex])
```

```
1  from hamiltonian_cycle1 import hamiltonian_cycle
2  from graph import Graph
3
4  G = Graph(E=[(0,1), (0,2), (0,3), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5),
5               (3,6), (4,7), (5,6), (5,8), (6,7), (6,8), (6,9), (7,9), (8,9)])
6  print 'Ciclo hamiltoniano:', hamiltonian_cycle(G)
```

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

En este algoritmo, generar un nuevo estado a partir del camino parcial que se suministra como argumento es una operación $O(|V|)$. Podemos mejorar el programa en al menos dos sentidos:

- usando una única lista para representar el estado actual, reduciendo el coste de generar un nuevo estado;
- y usando 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.

hamiltonian_cycle.py

```
1  from random import sample
2  def hamiltonian_cycle(G):
3      [random_vertex] = sample(G.V, 1)
4      path = [random_vertex] + [None] * (len(G.V)-1) + [random_vertex]
5      def backtracking(m, visited):
6          if m == len(G.V)-1:
7              if (path[m], path[0]) in G.E: return path
8          else:
9              for v in G.succs(path[m]):
10                 if v not in visited:
11                     path[m+1] = v
12                     visited.add(v)
13                     found = backtracking(m+1, visited)
14                     if found: return found
15                     visited.remove(v)
16         return None
17     return backtracking(0, set([random_vertex]))
```

```
1  from hamiltonian_cycle import hamiltonian_cycle
2  from graph import Graph
3
4  G = Graph(E=[(0,1), (0,2), (0,3), (1,3), (1,4), (2,3), (2,5), (3,4), (3,5),
5               (3,6), (4,7), (5,6), (5,8), (6,7), (6,8), (6,9), (7,9), (8,9)])
6  print 'Ciclo hamiltoniano:', hamiltonian_cycle(G)
```

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

## 6.5.2. Sobre el coste

Generar un nuevo estado a partir de otro es ahora una operación ejecutable en tiempo $O(1)$ para el peor de los casos. Hemos usado conjuntos implementados con tablas de dispersión, lo que hace que debamos matizar la última afirmación y hablar de coste esperado, pero resulta sencillo sustituir el conjunto por un vector de valores booleanos

indexado por los vértices y así poder hablar con propiedad de coste en el peor de los casos. En cualquier caso, el algoritmo presenta un coste temporal $\Omega(|V|)$ y $O(|V|!)$. Su peor caso presenta el coste propio del algoritmo de fuerza bruta.

EJERCICIOS

**6-17** ¿Es correcto el algoritmo desarrollado?

**6-18** Ciertos grafos no contienen ningún ciclo hamiltoniano. En ellos, el algoritmo ha de explorar completamente el árbol de estados para concluir que no existe el ciclo. El número de estados puede ser exponencial con el número de vértices, así que el tiempo de ejecución puede resultar no permisible. Averigua si hay comprobaciones computacionalmente eficientes que detecten, en algunos grafos, la inexistencia de ciclos hamiltonianos.

## 6.6. Cálculo de la envolvente convexa de puntos en el plano: el algoritmo de barrido de Graham

Ya planteamos el problema de la envolvente convexa en la sección 5.11 y propusimos una solución basada en la estrategia «divide y vencerás»: QuickHull. En esta sección estudiaremos el algoritmo conocido como «barrido de Graham», que es un algoritmo de búsqueda con retroceso. Antes hemos de definir algunos conceptos.

El **ángulo polar** de un punto $(x_j, y_j)$ respecto de un punto $(x_i, y_i)$ es el ángulo entre una línea horizontal y la línea que une a ambos puntos (véase la figura 6.20).

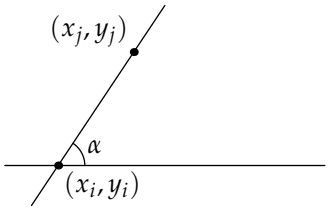



*Figura 6.20: Ángulo polar del punto $(x_j, y_j)$ respecto del punto $(x_i, y_i)$.*

Indicaremos los conjuntos de puntos que definen el polígono de la envolvente convexa siguiendo un orden antihorario. Si ordenamos los puntos de $S$ por valor creciente del ángulo polar con respecto a un punto del polígono convexo, al que denominamos **punto de referencia**, el polígono será una subsecuencia de la secuencia ordenada de puntos. El punto de referencia puede ser, por ejemplo, el de menor valor de $y_i$. Si dos puntos presentan el mismo ángulo polar, nos quedaremos únicamente con el que más dista del punto de referencia: en tal caso ambos son colineales con el de referencia y el más próximo puede suprimirse sin que ello afecte a la solución. Resulta sencillo comprobar que los puntos de menor y mayor ángulo polar forman parte de la envolvente convexa. La figura 6.21 ilustra este orden para un conjunto determinado de puntos.

Una solución completa es un conjunto de puntos de $S$ que empieza con tres vértices: el punto de mayor ángulo polar, el punto de referencia y el punto de menor ángulo polar. Si $n$ es el número de puntos de $S$ y éstos se numeran por ángulo polar creciente, un estado puede representarse, pues, con una tupla de la forma $(n-1, 0, 1, ?)$. Marcamos el final de

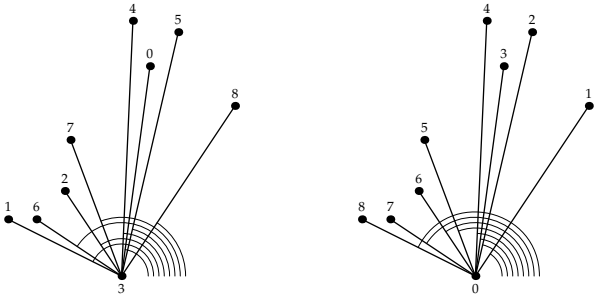



*Figura 6.21: (a) Ángulos polares de cada uno de los puntos de S. (b) Reordenamiento de los puntos por valor creciente del ángulo polar.*

la tupla con un único interrogante porque el tamaño de la tupla es indeterminado. Un estado terminal es de la forma $(n-1, 0, 1, \ldots, n-1)$.

El algoritmo debe considerar, uno a uno, los puntos de $S$ en el orden indicado para decidir si forman parte del polígono convexo. Hay una propiedad interesante que podemos aprovechar: si un punto forma parte de un giro a la derecha, no puede formar parte de la envolvente convexa. Así pues, si el punto considerado en un instante supone dar un giro a la izquierda con respecto al último punto, se añade al polígono; si, por contra, supone un giro a la derecha, no se añade y se efectúa un retroceso. Se puede determinar si un giro es a izquierdas o a derechas calculando el ángulo que forman los tres puntos implicados, como se muestra en la figura 6.22.

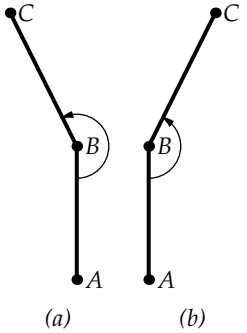



*Figura 6.22: En el recorrido de los puntos A, B y C (por ese orden), un giro a mano izquierda (a) se detecta porque el ángulo formado en $\widehat{ABC}$ tiene más de 180 grados y un giro a mano derecha (b), porque dicho ángulo tiene menos de 180 grados.*

La figura 6.23 ilustra el proceso que ahora describimos paso a paso:

a) Los puntos 8, 0 y 1 siempre forman parte de la envolvente convexa. Partimos, pues, del estado $(8, 0, 1, ?)$.

b) La esquina $\widehat{012}$ supone dar un giro a la izquierda, así que el punto 2 se incorpora a la envolvente. Se considera que el estado $(8, 0, 1, 2, ?)$ es prometedor.

c) También la esquina $\widehat{123}$ supone dar un giro a la izquierda, así que añadimos el punto 3 al estado actual, que pasa a ser $(8, 0, 1, 2, 3, ?)$.

d) Se estudia el estado $(8, 0, 1, 2, 3, 4, ?)$. La esquina $\widehat{234}$ da un giro a la derecha, así que se determina que el estado es no prometedor, se elimina el punto 3 del conjunto de puntos que pueden ser vértices del polígono y se efectúa un retroceso.

e) Se pasa a considerar el estado $(8,0,1,2,4,?)$ y se estudia la esquina $\widehat{124}$, que da un giro a la izquierda.

f) Añadimos el punto 5. El estado actual es $(8,0,1,2,4,5,?)$.

g) Y añadimos también el punto 6 para generar el estado $(8,0,1,2,4,5,6,?)$.

h) Pasamos al estado $(8,0,1,2,4,5,6,7,?)$. El punto 7 fuerza un giro a la derecha, por lo que se descarta el punto 6 y se considera no prometedor al estado actual.

i) Efectuamos un retroceso y volvemos al estado $(8,0,1,2,4,5,?)$ que prologamos añadiendo el punto 7 para obtener el estado prometedor $(8,0,1,2,4,5,7,?)$

j) Pero dicho estado no puede prolongarse para incluir al punto 8, pues la esquina $\widehat{578}$ da un giro a la derecha. El punto 7 se descarta y se efectúa un retroceso.

k) El estado $(8,0,1,2,4,5,8)$, que es terminal, no resulta factible: su última esquina da un giro a la derecha y descarta al punto 5.

l) El estado terminal $(8,0,1,2,4,8)$ es una solución factible.

Los índices utilizados son los propios del vector de puntos ordenados por ángulo polar creciente. Sobre el vector original, la envolvente convexa comprende los puntos de índices $(3,8,5,4,1)$.

### 6.6.1. Algunos aspectos de la implementación

Estudiemos en primer lugar cómo implementar algunos aspectos del algoritmo: selección del punto de referencia, ordenación por ángulo polar creciente y determinación de la dirección de un giro.

Sea $S$ una lista de puntos, cada uno de los cuales es una tupla de reales (un par). El punto de referencia es el que presenta menor valor de su ordenada. El ángulo de cada punto con respecto al punto de referencia puede calcularse con la función *atan2*, que determina el ángulo de un punto a partir de sus coordenadas. Con este cálculo podemos tener los puntos ordenados por ángulo polar creciente:

```
1 p = min(xrange(len(S)), key=lambda i: S[i][1])
2 sorted_indices = [p] + sorted((q for q in xrange(len(S)) if q != p), \
3                               key=lambda q: atan2(S[p][1]-S[q][1], S[p][0]-S[q][0]))
```

La forma de decidir si un giro es a izquierdas o a derechas es sencilla (se presentó en la sección 5.11.1 y las funciones de cálculo correspondientes están disponibles en el módulo `geometry.py`).

En realidad no es necesario calcular el ángulo polar de cada punto: basta con calcular la tangente del vector diferencia con el punto de referencia. Si lo hacemos, los puntos situados a la derecha del punto de referencia presentan tangente positiva y los que hay a su izquierda, tangente negativa. Se puede evitar este problema, que afecta a la ordenación, rotando 90 grados cada punto. Basta, para ello, con considerar el punto $(-y,x)$ en lugar de $(x,y)$. Este procedimiento evita recurrir a funciones de la librería matemática, que generalmente son costosas.

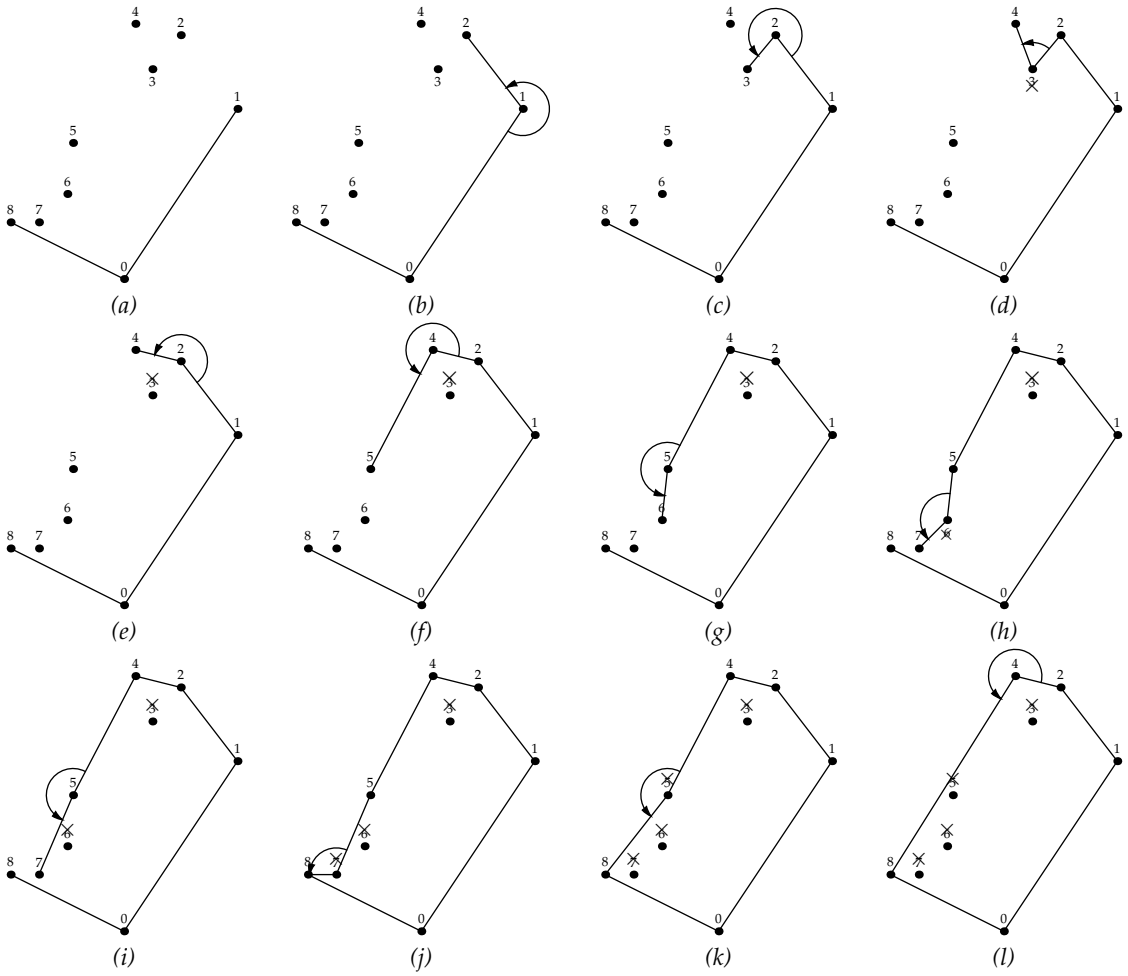



*Figura 6.23: Barrido de Graham, paso a paso. Los puntos de S están ordenados por valor creciente del ángulo polar.*

## 6.6.2. Un algoritmo recursivo

Hemos de tener en cuenta que un punto que produce un giro a la derecha no puede ser uno de los vértices que definen la envolvente convexa y, por tanto, puede descartarse. Usamos un vector de valores booleanos para indicar qué puntos siguen siendo candidatos a vértices del polígono convexo:

convex_hull1.py

```
1  from math import atan2
2  from geometry import left
3
4  def graham_scan(S):
5      discarded = [False] * (len(S))
6      hull = [0, 1] + [None] * (len(S)-2)
7      def backtracking(i):
8          if hull[i-1] == len(S)-1:
```

```
            return hull[:i]
        else:
            for p in xrange(hull[i-1]+1, len(S)):
                if not discarded[p]:
                    hull[i] = p
                    if left(S[hull[i-2]], S[hull[i-1]], S[hull[i]]):
                        found = backtracking(i+1)
                        if found != None: return found
                    else:
                        discarded[hull[i-1]] = True
                        return None
        return None

    p = min(xrange(len(S)), key=lambda i: S[i][1])
    sorted_indices = [p] + sorted((q for q in xrange(len(S)) if q != p), \
                                  key=lambda q: atan2(S[p][1]-S[q][1], S[p][0]-S[q][0]))
    S = [S[q] for q in sorted_indices]
    return [sorted_indices[i] for i in backtracking(2) ]
```

Pongamos a prueba nuestra implementación con los puntos de la figura 6.21 (a):

```
from convex_hull1 import graham_scan

S = [(2.5,2.7),(0,0),(1,.5),(2,-1),(2.2,3.5),(3,3.3),(.5,0),(1.1,1.4),(4,2)]
print 'Envolvente convexa:', graham_scan(S)
```

```
Envolvente convexa: [3, 8, 5, 4, 1]
```

### 6.6.3. Una versión iterativa

Podemos dar una versión iterativa del algoritmo de Graham (que es la versión más popular del algoritmo):

convex_hull.py

```
from lifo import Stack
from math import atan2
from geometry import left

def graham_scan(S):
    p = min(xrange(len(S)), key=lambda i: S[i][1])
    sorted_indices = [p] + sorted((q for q in xrange(len(S)) if q != p), \
                                  key=lambda q: atan2(S[p][1]-S[q][1], S[p][0]-S[q][0]))
    S = [S[q] for q in sorted_indices]
    Q = Stack()
    Q.push(0); Q.push(1); Q.push(2)
    for pi in xrange(3, len(S)):
        pj, pk = Q[-1], Q[-2]
        while not left(S[pk], S[pj], S[pi]):
            Q.pop()
```

```
16            pj, pk = Q[-1], Q[-2]
17        Q.push(pi)
18    convex_hull = [ sorted_indices[Q.pop()] for i in xrange(len(Q)) ]
19    convex_hull.reverse()
20    return convex_hull
```

```
1  from convex_hull import graham_scan
2
3  S = [(2.5,2.7),(0,0),(1,.5),(2,-1),(2.2,3.5),(3,3.3),(.5,0),(1.1,1.4),(4,2)]
4  print 'Envolvente convexa:', graham_scan(S)
```

```
Envolvente convexa: [3, 8, 5, 4, 1]
```

La solución proporcionada por el algoritmo iterativo es una lista con los vértices del polígono en sentido *horario*, no antihorario.

El algoritmo difiere del que obtendríamos aplicando directamente el esquema iterativo porque el hecho de que podamos descartar puntos que efectúan giros a la derecha permite una manipulación de la pila más eficiente.

Ronald L. Graham ideó el algoritmo del barrido de Graham en 1972 y lo publicó en el primer volumen de la revista Information Processing Letters, en un artículo titulado «An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set».

### 6.6.4. Corrección

Demostremos que el algoritmo de barrido de Graham funciona correctamente. Consideraremos su versión iterativa.

Sean $p_0$, $p_1$, $\ldots p_m$ los puntos seleccionados por el algoritmo y que suponemos que definen la envolvente convexa de $S$. Si un punto $q$ ha sido descartado en la fase de ordenación por tener el mismo ángulo polar que otro $p_i$ con respecto al punto de referencia, entonces $q$ está en el segmento $\overline{p_0 p_i}$, ya que $p_i$ está más lejos de $p_0$ que $q$. Así pues, $q$ forma parte de la envolvente convexa. Sin pérdida de generalidad, supongamos que $S$ no contiene puntos de igual ángulo polar y colineales con respecto al punto de referencia.

Sea $C_i$ la envolvente convexa de $p_0, \ldots, p_i$. Demostremos por inducción sobre el índice del bucle $i$, para $i$ entre 2 y $n$, que, tras cada iteración, los puntos apilados en $Q$ son los vértices de $C_i$ en orden horario. Antes de entrar en el bucle, la pila contiene los tres vértices de $C_2$ en orden horario. Como la envolvente convexa de 3 puntos no colineales son los mismo tres puntos, ya tenemos demostrada la base de inducción. Supongamos ahora que $i$ es mayor que 2, es decir, que ejecutamos el bucle. El punto $p_i$ siempre acaba en la cima de la pila tras la ejecución del bucle. Como el ángulo polar de $p_i$ es mayor que el ángulo polar de $p_{i-1}$, el polígono $(p_0, p_{i-1}, p_i)$ forma un triángulo que no está contenido en $C_{i-1}$, como se puede ver en la figura 6.24 (a). Así pues, el punto pertenece a $C_i$.

En cada iteración se consideran los dos puntos de la cima de la pila, $p_j$ y $p_k$, y el nuevo punto, $p_i$. El giro $\widehat{p_k p_j p_i}$ puede darse a la derecha o a la izquierda.

- Si es a la derecha, el triángulo $\triangle p_k p_j p_i$ está incluido en el triángulo $\triangle p_k p_0 p_i$, así que $p_j$ está contenido en la envolvente convexa de $\{p_0, p_k, p_i\}$ y no es un vértice de $C_i$, como ilustra la figura 6.24 (b). Puede descartarse sin problemas.
- Si es a la izquierda, el triángulo $\triangle p_k p_j p_i$ no está contenido en el triángulo $\triangle p_k p_0 p_i$. Así pues, $p_j$ es un vértice de $C_i$. Puede comprobarse que los vértices de $C_{i-1}$ entre $p_0$ y $p_j$ en sentido antihorario siguen siendo vértices de $C_i$.

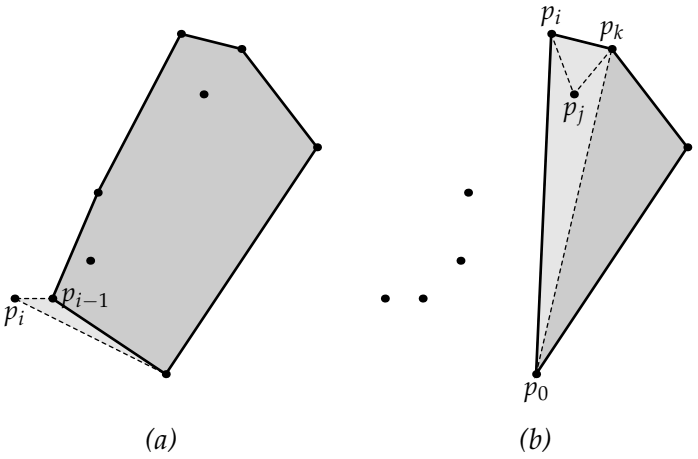



*Figura 6.24: (a) El último punto considerado siempre forma parte de la envolvente convexa. (b) Cuando se produce un giro a la derecha, el punto $p_j$ puede descartarse.*

## 6.6.5. Análisis

La ordenación de los puntos por valor polar es una operación $O(n \lg n)$. El bucle principal se ejecuta $n-2$ veces, es decir, $O(n)$ veces. El bucle interior se ejecuta a los sumo una vez por punto, ya que el resultado neto de sus acciones es que cada punto ingresa una vez en la pila y sólo puede desapilarse definitivamente una vez. El bucle interno supone la ejecución, pues, de $O(n)$ pasos. El coste temporal del algoritmo es, por tanto, $O(n \lg n)$.

Ya estudiamos el algoritmo QuickHull, que requería tiempo $O(n^2)$, y ahora hemos presentado otro algoritmo que lo soluciona en tiempo $O(n \lg n)$, el algoritmo del barrido de Graham. Existen otros algoritmos que resuelven el problema, entre los que podemos citar el «algoritmo incremental», que requiere tiempo $O(n^2)$; MergeHull, otro método «divide y vencerás», que resuelve el problema en tiempo $O(n \lg n)$ y la «marcha de Jarvis» o «envoltura del regalo», que se ejecuta en tiempo $O(nh)$, siendo $h$ el número de vértices del polígono que defina la envolvente convexa (el valor de $h$ es $O(n)$, así que en casos patológicos, el algoritmo es $O(n^2)$). Cabe indicar, finalmente, que hemos estudiado sólo el problema en dos dimensiones, pero que también tiene interés práctico con 3 o más dimensiones.

EJERCICIOS

**6-19** Analiza el coste espacial del algoritmo.

## 6.7. Cálculo de funciones de dispersión perfectas

El almacenamiento de pares clave-valor en un tabla de dispersión pasa por el cálculo de una función de dispersión que asigna un número entero a cada clave. Se produce una colisión cuando la función proporciona el mismo entero para dos claves distintas. Cuando se consulta la tabla, la velocidad de acceso depende del número de colisiones. En una tabla sin colisiones, el tiempo de acceso es proporcional al tiempo de cálculo de la función de dispersión. Si no se conoce a priori el conjunto de claves que ingresarán en la tabla, no es posible definir una función de dispersión que garantice que no se produce colisión alguna. Pero en ciertos problemas se conoce de antemano el conjunto completo de claves, por lo que es factible diseñar una función de dispersión sin colisiones, es decir, una **función de dispersión perfecta**.

En el diseño de intérpretes y compiladores, por ejemplo, se desea saber si una cadena formada por letras es el identificador de una variable o una palabra clave. El mismo problema surge, por ejemplo, en los editores de texto sensibles a la sintaxis, que deben dar un formato especial a las palabras clave de un lenguaje. Una solución clásica a este problema pasa por disponer las palabras en un vector ordenado y efectuar una búsqueda binaria cuando se encuentra una cadena formada por caracteres. Así es posible saber si se trata de una palabra clave en tiempo $O(\lg n)$. Pero puede resultar mucho más eficiente almacenarlas en una tabla de dispersión si ésta no presenta colisiones. En tal caso, el coste de la consulta es $O(1)$. Generar la función de dispersión perfecta puede ser un proceso costoso, pero se realiza en una fase previa a la de explotación del intérprete o compilador, así que resulta permisible.

Estudiaremos una técnica de búsqueda de una función de dispersión perfecta basada en la búsqueda por retroceso que es una versión de la ideada por Richard J. Cichelli. El método trata de encontrar una función de dispersión de la forma

$$h(w) = (|w| + g(w_0) + g(w_{|w|-1})),$$

donde $w$ es una cadena, $|w|$ es su talla y $w_0$ y $w_{|w|-1}$ son su primer y último carácter, respectivamente. Nótese que la función $h$ puede calcularse en tiempo $O(1)$ si $g$ es computable en tiempo constante.

El método original se publicó en el artículo «*Minimal Perfect Hash Functions Made Simple*», de Richard J. Cichelli, en el número 23 de *Communications of the ACM*, en 1980. El primer método que presentamos está inspirado en el algoritmo de Cichelli, que estudiamos más adelante.

Hemos de definir los valores de $g(c)$ para todo carácter que pueda aparecer como primer o último carácter. Y hemos de hacerlo de modo que $h(w)$ proporcione un valor distinto para toda cadena de cierto conjunto. El objetivo, naturalmente, es que no haya dos palabras de la lista que proporcionen un mismo valor de la función $h$.

Sean $\{c_0, c_1, \ldots, c_{k-1}\}$ el conjunto de caracteres que aparecen en primera o última posición de las palabras de la lista. Buscamos una tupla $(v_0, v_1, \ldots, v_{k-1})$ que representa

una definición de la función $g$ consistente en que $g(c_i) = v_i$, para $0 \leq i < k$. Podemos buscar la tupla por un procedimiento de búsqueda con retroceso.

Los estados de la forma $(v_0, v_1, \ldots, v_{j-1}, ?)$, con $j \leq k$, serán prometedores si la función $h$ aplicada a aquellas palabras para las que se puede evaluar ya la función de dispersión no presenta colisiones. Nótese que un estado completo y prometedor representa una solución factible.

### 6.7.1. Un algoritmo recursivo

Hemos de tener en cuenta que no hay límite al número de hijos de un estado: nada se dice acerca de imponer límite alguno a los valores almacenados en $g$. Es posible, pues, que exista un número infinito de hijos para cada estado no terminal, lo que puede conducir a la no terminación del algoritmo. Cichelli propuso fijar un límite superior al valor que puede devolver $g$. Este algoritmo resuelve el problema:

perfect_hash.py

```
def is_promising(g, words):
    used = set()
    for w in words:
        if w[0] in g and w[-1] in g:
            h = len(w) + g[w[0]] + g[w[-1]]
            if h in used: return False
            used.add(h)
    return True

def perfect_hash1(words, max_value):
    chars = list(set(w[0] for w in words) | set(w[-1] for w in words))
    g = {}
    def backtracking(i):
        if i == len(chars): return g
        for g[chars[i]] in xrange(max_value):
            if is_promising(g, words):
                found = backtracking(i+1)
                if found: return found
        del g[chars[i]]
        return None

    g = backtracking(0)
    def h(w): return len(w) + g[w[0]] + g[w[-1]]
    return h
```

```
from perfect_hash import perfect_hash1

words = ('break case continue default do else ' + \
        'for goto if return struct switch while').split()
h = perfect_hash1(words, 20)
for word in words: print '%s:%d' % (word, h(word)),
```

```
break:7 case:5 continue:9 default:13 do:8 else:6 for:12 goto:10 if:2 return:15
 struct:17 switch:11 while:14
```

## 6.7.2. Algunos refinamientos

El algoritmo presentado puede mejorarse en varios aspectos:

- Conviene explorar los caracteres en orden de mayor a menor frecuencia: de ese modo, la lista de palabras para las que se conoce el valor de $h$ crece más rápidamente en niveles poco profundos del árbol.
- La detección de si un estado es o no es prometedor puede realizarse sin necesidad de recalcular completamente el valor de $h$ sobre todas las palabras. Podemos, en una fase de preproceso, determinar para qué palabras se conoce el valor de $h$ cuando se conocen las $j$ primeras componentes de la solución.

perfect_hash.py

```
1  def perfect_hash(words, max_value):
2      c = {}
3      for w in words:
4          c[w[0]] = c.get(w[0], 0) + 1
5          c[w[-1]] = c.get(w[-1], 0) + 1
6      chars = sorted(c, key=lambda x: -c[x]) # Caracteres ordenados por frecuencia decr.
7      g = {}
8      available = dict((char, []) for char in chars) # w de las que se conoce g(w_0) y g(w_|w|).
9      pending = set(words)
10     known = set()
11     for char in chars:
12         known.add(char)
13         for word in pending:
14             if word[0] in known and word[-1] in known: available[char].append(word)
15         for word in available[char]: pending.remove(word)
16     used = set() # Valores de h ya utilizados.
17     def backtracking(i):
18         if i == len(chars): return g
19         for g[chars[i]] in xrange(max_value):
20             aux = set(len(word)+g[word[0]]+g[word[-1]] for word in available[chars[i]])
21             if len(aux) == len(available[chars[i]]) and len(aux & used) == 0:
22                 used.update(aux)
23                 found = backtracking(i+1)
24                 if found: return found
25                 used.difference_update(aux)
26         del g[chars[i]]
27         return None
28
29     g = backtracking(0)
30     def h(w): return len(w) + g[w[0]] + g[w[-1]]
31     return h
```

```
from perfect_hash import perfect_hash

words = ('break case continue default do else ' + \
        'for goto if return struct switch while').split()
h = perfect_hash(words,20)
for word in words: print '%s:%d' % (word, h(word)),
```

```
break:12 case:5 continue:9 default:7 do:2 else:4 for:3 goto:8 if:10 return:13
struct:6 switch:11 while:14
```

### 6.7.3. El algoritmo de Cichelli

Cichelli propuso un algoritmo de búsqueda con retroceso diferente del que hemos presentado. Una tupla $(v_0, v_1, \ldots, v_{n-1})$, con tantos elementos como palabras hay que almacenar, contiene en $v_i$ el valor de $h$ aplicado a la palabra de índice $i$. Ese valor se obtiene asignando un valor a $g$ para dos caracteres. Puede que cuando llegue la hora de considerar valores para una palabra, sus caracteres inicial y/o final ya hayan sido considerados al considerar palabras anteriores, lo que complica ligeramente la gestión de los valores almacenados en $g$. Por otra parte, y con objeto de acelerar la búsqueda, las palabras se consideran en orden de mayor a menor (suma de) frecuencias de aparición de sus letras inicial y última:

perfect_hash.py

```
def perfect_hash_cichelli(words, max_value):
    # c almacena la frecuencia de aparición de las letras primera y última en las palabras.
    c = {}
    for word in words:
        c[word[0]] = c.get(word[0], 0) + 1
        c[word[-1]] = c.get(word[-1], 0) + 1
    # Se ordenan las palabras por frecuencia de aparición de sus letras.
    words = sorted(words, key=lambda word: c[word[0]]+c[word[-1]], reverse=True)
    used = set()
    g = {}
    def backtracking(i):
        if i == len(words): return g
        first, last = words[i][0], words[i][-1]

        if first in g: first_del, first_from, first_to = False, g[first], g[first]+1
        else: first_del, first_from, first_to = True, 0, max_value

        if last in g: last_del, last_from, last_to = False, g[last], g[last]+1
        else: last_del, last_from, last_to = True, 0, max_value

        for g[first] in xrange(first_from, first_to):
            for g[last] in xrange(last_from, last_to):
                value = len(words[i]) + g[first] + g[last]
                if value not in used:
```

```
                used.add(value)
                found = backtracking(i+1)
                if found: return found
                used.remove(value)
        if first_del: del g[first]
        if last_del: del g[last]
        return None
    g = backtracking(0)
    def h(w): return len(w) + g[w[0]] + g[w[-1]]
    return h
```

```
from perfect_hash import perfect_hash_cichelli

words = ('break case continue default do else ' + \
        'for goto if return struct switch while').split()
h = perfect_hash_cichelli(words,20)
for word in words: print '%s:%d' % (word, h(word)),
```

```
break:14 case:5 continue:9 default:7 do:2 else:4 for:3 goto:10 if:11 return:12
 struct:8 switch:13 while:6
```

EJERCICIOS

**6-20** Compara el número de estados visitados por el algoritmo de *perfect_hash* (versión refinada) y por *perfect_hash_cichelli*.

## 6.8. El problema de la cobertura exacta y los «enlaces bailarines»

Hay un enfoque ligeramente diferente y que permite acometer con mayor eficiencia la búsqueda con retroceso en ciertos tipos de problema. Para ilustrarlo abordaremos la resolución del denominado «problema de la cobertura exacta», que se enuncia así: dada una matriz de ceros y unos con dimensión $n \times m$, ¿hay un conjunto de filas que contenga exactamente un uno en cada columna? De existir, dicho conjunto de filas es una cobertura exacta de la matriz.

Consideremos la instancia del problema definida por esta matriz con $n = 6$ filas y $m = 6$ columnas:

$$M = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{6.1}$$

El conjunto formado por las filas de índices 0 y 5, por ejemplo, constituye una solución factible: la fila 0 tiene unos en las columnas 0, 1 y 5 y la fila 5 tiene unos en las columnas 2, 3, 4.

### 6.8.1. Una aproximación convencional

Podemos abordar el problema siguiendo la aproximación de problemas anteriores: modelamos una solución con una tupla de $n$ valores booleanos, cada uno de los cuales indica si una de las filas pertenece o no a la cobertura exacta. Los estados se modelan con tuplas de talla menor o igual que el número de filas en la matriz y determinamos que un estado es prometedor si no hay una columna con dos o más unos en las filas seleccionadas. Un estado es completo cuando su talla es igual a $n$ y es, además, prometedor. He aquí una implementación directa de esta idea:

exactcover.py

```
from backtrackingscheme import BacktrackingScheme

class ExactCover(BacktrackingScheme):
    def __init__(self, matrix):
        self.M = matrix
        self.n = len(self.M)
        self.m = len(self.M[0])

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

    def is_feasible(self, s):
        for j in xrange(self.m):
            one = False
            for i in xrange(self.n):
                if s[i] and self.M[i][j]:
                    one = True
                    break
            if not one: return False
        return True

    def is_promising(self, s):
        ones = [False] * self.m
        for i in xrange(len(s)):
            if s[i]:
                for j in xrange(self.m):
                    if self.M[i][j]:
                        if ones[j]: return False
                        else: ones[j] = self.M[i][j]
        return True

    def branch(self, s):
        yield s + [True]
```

```
34          yield s + [False]
```

```
1  from exactcover import ExactCover
2
3  print ExactCover([[1,1,0,0,0,1],
4                    [0,0,0,1,1,0],
5                    [0,0,1,1,0,1],
6                    [1,0,1,1,0,0],
7                    [0,1,0,0,1,1],
8                    [0,0,1,1,1,0]]).backtracking([])
```

```
[True, False, False, False, False, True]
```

EJERCICIOS

**6-21** Determina el coste temporal en el mejor caso y el coste espacial del algoritmo propuesto.

Cabe esperar una mejora en la eficiencia del algoritmo si seguimos dos principios de diseño:

- Representar los estados con una única variable global sobre la que hacemos/deshacemos cambios al avanzar la exploración.
- Ordenar el proceso de ramificación de modo que se genere el menor número posible de hijos en cada instante.

Vamos a presentar una aproximación muy diferente que explota ambas líneas de mejora: la que se conoce por algoritmo DLX (que viene de contraer en inglés la expresión «dancing links», es decir, punteros o enlaces bailarines).

La idea de los enlaces bailarines apareció por vez primera al resolver el problema de las $n$ reinas en un trabajo de Hirosi Hitotumatu y Kohei Noshita. Fue extendida a un marco más general de búsqueda con retroceso y presentada por Donald E. Knuth en su trabajo «Dancing Links», del que este apartado es un breve resumen.

## 6.8.2. El algoritmo DLX

El algoritmo aborda el problema de un modo diferente. Un estado es un conjunto de filas seleccionadas y una matriz. Inicialmente, el conjunto de filas está vacío y la matriz es la matriz completa de unos y cero, $M$. A continuación se aplica el siguiente procedimiento recursivo:

1. Escoger la columna con menos unos. Sea $j$ su índice.
2. Para toda fila $i$ con un uno en la columna $j$, hacer:

   2.1 «Cubrir» toda columna en la que la fila $i$ tiene un uno, es decir, eliminar esa columna y toda fila en la que ésta contiene un uno.

2.2 Resolver recursivamente el problema sobre la matriz resultante.

2.3 Si se encontró una solución, añadir a la solución la fila $i$ y devolver el conjunto resultante.

2.4 Si no se encontró una solución, deshacer los cambios efectuados a la matriz.

Si aplicamos el método a la matriz (6.1), empezaremos por seleccionar la columna con menos unos, que es la primera (la de índice 0). Las filas 0 y 3 tienen sendos unos en dicha columna, por lo que hemos de considerar si forman parte de la cobertura o no. Empezamos, por ejemplo, por la fila 0, que pasamos a eliminar de la matriz. Al seleccionar la fila 0 tendremos que cubrir las columnas 0, 1 y 5 (que es donde contiene unos). Al cubrir la columna 0 eliminamos las filas 0 y 3, quedando así la matriz (por el momento):

$$\begin{array}{c} \\ 1 \\ 2 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array} \tag{6.2}$$

Como hemos de cubrir, además, las columnas 1 (que elimina la fila 4) y 5 (que elimina las filas 2 y 4), obtenemos una nueva matriz reducida:

$$\begin{array}{c} \\ 1 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccc} 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \end{array}$$

Ahora tratamos de resolvemos el problema de la cobertura máxima para la matriz reducida. Escogemos la columna con menos unos, es decir, la columna de índice 2. Al hacerlo pasamos a seleccionar y eliminar la fila de índice 5, y cubrimos las columnas 2, 3 y 4, con lo que se suprime la fila 1. Ya está. La matriz queda vacía y el algoritmo finaliza devolviendo el conjunto $\{5\}$ como solución para la matriz reducida y el conjunto $\{0,5\}$ como solución para la matriz completa.

El cuello de botella del método parece radicar en la gestión de las diferentes matrices. Crear una nueva matriz cada vez que se reduce una eliminando columnas y filas es muy gravoso, tanto en tiempo como en espacio. Interesa mantener una única matriz sobre la que anotar de algún modo qué filas y columnas «sobreviven» en cada paso y que haga posible deshacer eficientemente los borrados de filas y columnas efectuadas en cada paso. Podemos usar una única matriz y mantener una relación de filas y columnas supervivientes, pero el tiempo necesario para encontrar unos en la matriz seguirá resultando elevado. El algoritmo DLX se basa en la gestión de una estructura de datos que permite borrar/restaurar eficientemente filas y columnas de la matriz de acuerdo con el proceso de búsqueda, así como efectuar recorridos de filas y columnas con tiempo proporcional al número de unos que contienen en un instante dado. Ilustramos esquemáticamente esta estructura de datos para la matriz de ejemplo en la figura 6.25. La estructura está formada por:

- Una lista circular doblemente enlazada para cada columna, cada una con un nodo especial: una «cabecera» en la que se anota un identificador de columna y un contador con el número de sus elementos. Cada uno de sus nodos representa un uno en la matriz y contiene punteros a su antecesor y predecesor y a su cabecera.
- Una lista circular doblemente enlazada para cada fila. Cada uno de sus nodos representa un uno en la matriz. Los nodos se comparten con la lista circular de la correspondiente columna. A los punteros referidos antes, se añaden en cada nodo punteros al antecesor y al predecesor en la lista de la correspondiente fila.
- Una lista circular doblemente enlazada de cabeceras de lista de columna con un nodo especial: su propia cabecera, que apunta a las cabezas de la primera y última columnas. Este nodo especial se denotará con el símbolo *self*.

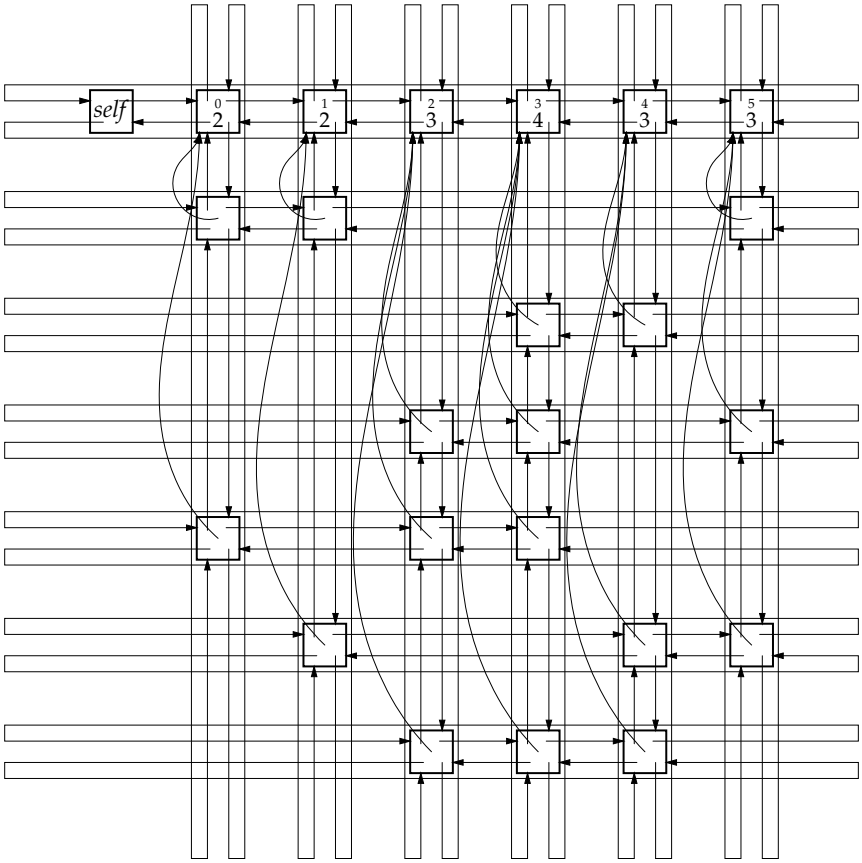



*Figura 6.25: Estructura de datos para representar una matriz de unos y ceros.*

La estructura permite hacer eficientemente ciertas operaciones necesarias:

- Conocer la (cabecera de) columna que menos unos tiene (en tiempo proporcional al número de columnas).
- Dada una columna, acceder a todas las filas que tienen un uno en ella (en tiempo proporcional al número de unos de la columna).

- Dada una fila, acceder a las (cabeceras de) columnas que tienen un uno en ella (en tiempo proporcional al número de unos de la fila).

Pero lo realmente interesante es la eficiencia con la que permite eliminar filas y columnas, así como reintegrarlas cuando convenga. El principio en el que se basa la eficiencia de estas operaciones se ilustra en la figura 6.26 y da nombre al algoritmo. En ella se muestran tres nodos consecutivos de una lista circular doblemente enlazada, con el nodo central apuntado por un puntero $p$. En cada nodo, el puntero $R$ (por «right») apunta al siguiente nodo y el puntero $L$ (por «left») apunta al anterior.

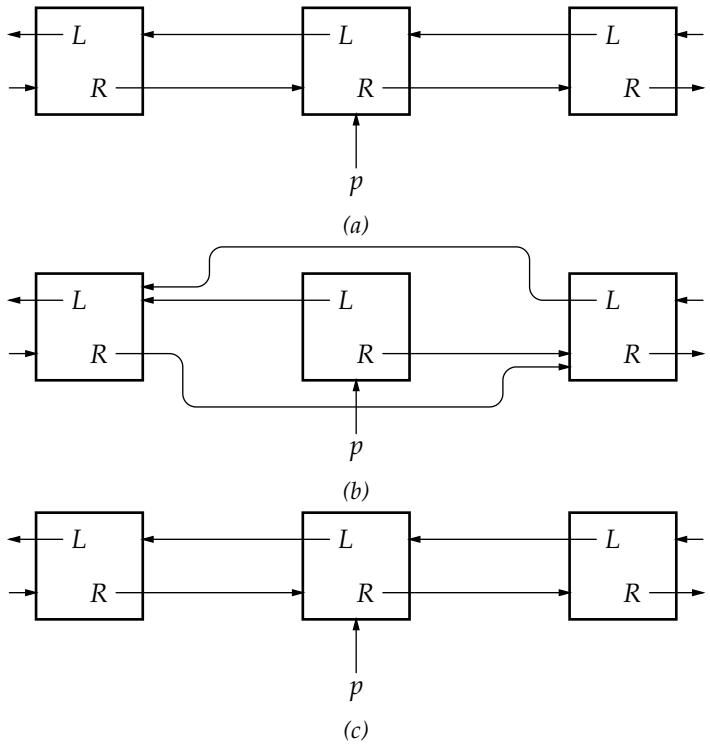



*Figura 6.26: (a) Tres nodos en una lista doblemente enlazada. El nodo central es apuntado por p. (b) Resultado de ejecutar* $p.L.R,\ p.R.L = p.R,\ p.L$ *sobre la lista de arriba. (c) Resultado de ejecutar a continuación* $p.L.R = p.R.L = p$*, que es idéntico a la situación de partida.*

Si deseamos borrar el nodo apuntado por $p$, basta con ejecutar la sentencia

```
1 p.L.R, p.R.L = p.R, p.L
```

con el resultado que se muestra en 6.26 (b). Y si $p$ sigue apuntando al nodo eliminado, podemos reintegrarlo eficientemente a la lista con la sentencia

```
1 p.L.R = p.R.L = p
```

el resultado que se muestra en 6.26 (c). Esto ocurre porque $p$, aun cuando ha sido eliminado de la lista, «recuerda» qué nodos eran su sucesor y predecesor en la lista. Los dos pares de sentencias pueden aplicarse a cualquier nodo de una lista circular doblemente enlazada que tenga dos o más elementos.

Cuando hemos de eliminar una columna, basta con aplicar la primera sentencia a su cabecera en tiempo $O(1)$. Y la segunda sentencia permite restaurarla con la misma complejidad temporal. Cuando en el algoritmo se cubre una columna, hemos de eliminarla

a ella y a toda fila en la que dicha columna tiene un uno. El borrado de las filas requiere cierta explicación. Cada uno de la matriz es una celda que está integrada en dos listas circulares: la de su fila y la de su columna. Cuando borremos una fila como consecuencia de la cobertura de una columna, eliminaremos de las demás listas de columna correspondientes todos los unos de dicha fila. En la figura 6.27 se muestra el efecto de cubrir la columna 0 en la matriz de ejemplo. Se puede observar que la cabecera de la columna 0 ha sido suprimida de la lista de cabeceras de columna y que las filas 0 y 3 se han eliminado: sus unos en columnas distintas de la eliminada se han borrado de la columna correspondiente. La operación realizada es fácilmente reversible: basta con recorrer la lista de unos en cada fila eliminada en orden inverso (lo que es posible porque sus punteros izquierda/derecha no se han alterado) para restituirlos a su lista de columna y, finalmente, restituir la columna borrada. Es preciso recorrer la lista en orden inverso para asegurar que se encuentra en el mismo estado en que quedó una vez ejecutada la acción que deshacemos en cada paso.

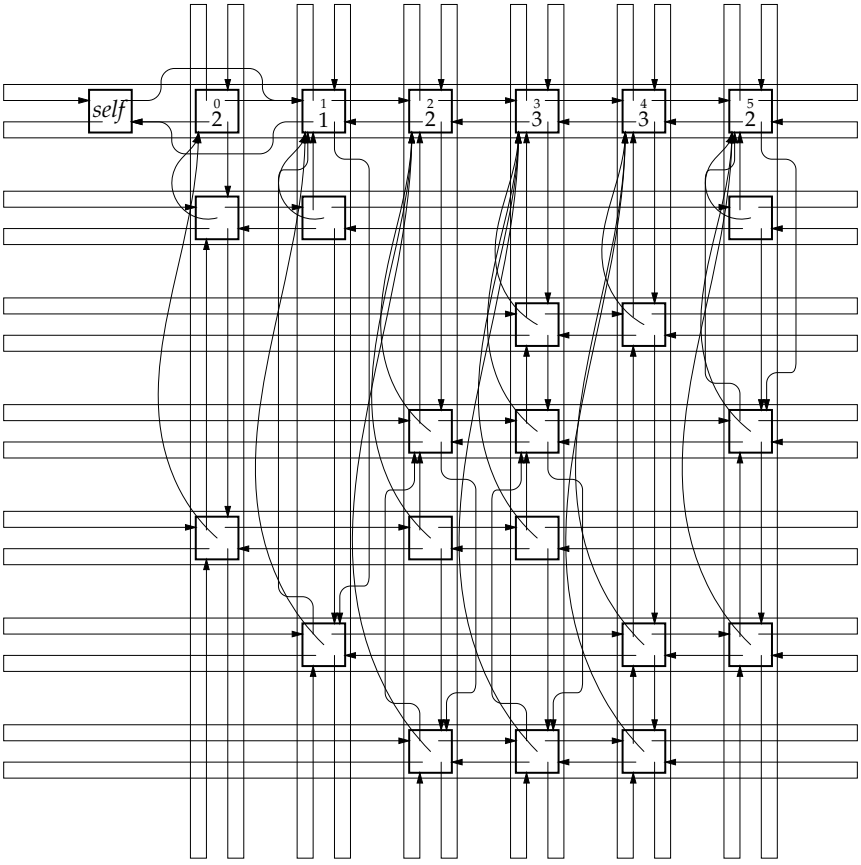



*Figura 6.27: Estado de la estructura de datos al cubrir la columna 0. Su cabecera no forma parte de la lista circular de cabeceras. Las filas en las que la columna 0 contiene unos también se ven afectadas: los unos que no están en esa columna han sido suprimidos de su respectiva lista circular de fila.*

Presentemos con mayor detalle los componentes de la estructura. Los nodos que representan unos contienen cinco punteros:

- *U* y *D* (por «Up» y «Down»), que le unen a la lista de su columna,

- *L* y *R* (por «Left» y «Right»), que le unen a la lista de su fila,
- y *H* (por «Header»), que le une a la cabeza de su columna.

Los representaremos con un clase que ofrece métodos para borrar/restituir un nodo de su lista de columna:

dlx.py

```
1  class One:
2      def __init__(self, U=None, D=None, H=None):
3          self.L, self.R = self, self
4          self.U, self.D = U, D
5          self.H = H
6
7      def remove(self):
8          self.D.U, self.U.D = self.U, self.D
9          self.H.S -= 1
10
11     def restore(self):
12         self.D.U = self.U.D = self
13         self.H.S += 1
```

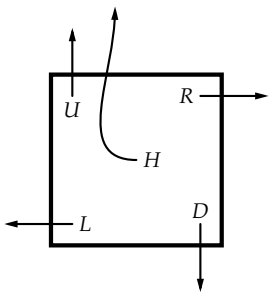



*Figura 6.28: Cada nodo de la estructura de datos presenta 5 punteros: dos le unen a la lista circular de su columna (U y D), otros dos a la de su fila (L y R) y uno (H) le une directamente a la cabeza de su lista de columna.*

Los nodos que conforman la lista de cabezas de columna contienen 4 punteros (*U*, *D*, *L* y *R*) y un campo con el número de nodos convencionales que forman la lista (*S* por «size»). Este campo permite buscar cómodamente la columna con menor número de unos, que es la que se escoge en cada recursión del algoritmo. Estos nodos son instancias de la siguiente clase, que proporciona métodos para cubrir/recuperar una columna:

dlx.py *(cont.)*

```
16 class Column:
17     def __init__(self, L=None, R=None, S=0):
18         self.L, self.R = L, R
19         self.U, self.D = self, self
20         self.S = S
21
22     def cover(self):
23         self.R.L, self.L.R = self.L, self.R
24         ptr = self.D
25         while ptr != self:
```

```
26            cell = ptr.R
27            while cell != ptr:
28                cell.remove()
29                cell = cell.R
30            ptr = ptr.D
31
32        def uncover(self):
33            ptr = self.U
34            while ptr != self:
35                cell = ptr.L
36                while cell != ptr:
37                    cell.restore()
38                    cell = cell.L
39                ptr = ptr.U
40            self.R.L = self.L.R = self
```

Finalmente, el nodo *self*, es la instancia de una clase que representa a la matriz. Contiene, entre otros campos, dos punteros ($L$ y $R$) con los que podemos acceder a la lista de cabeceras de columna. Su constructor recibe una matriz de unos y ceros y construye la estructura de listas circulares doblemente enlazadas.

dlx.py *(cont.)*

```
42  class DLX:
43      def __init__(self, ones):
44          self.n, self.m = len(ones), len(ones[0])
45          self.R = self.L = None
46          self.row_index = {}
47
48          # Crea lista circular de columnas.
49          ptr = self
50          for i in xrange(self.m):
51              ptr.R = Column()
52              ptr.R.L = ptr
53              ptr = ptr.R
54          ptr.R = self
55          self.L = ptr
56
57          # Crea listas circulares con los unos.
58          for row in xrange(self.n):
59              current_col = self
60              first = None
61              for col in xrange(self.m):
62                  current_col = current_col.R
63                  if ones[row][col]:
64                      current_col.S += 1
65                      one = One(H=current_col, U=current_col.U, D=current_col)
66                      one.ROW = row
67                      self.row_index[one] = row
68                      if first == None:
69                          first = one
```

```
70                  else:
71                      one.L = first.L
72                      one.R = first
73                      one.L.R = one
74                      first.L = one
75                  current_col.U = one
76                  one.U.D = one
```

La rutina de selección de la siguiente columna a cubrir recorre la lista de cabeceras y devuelve la que menos unos tiene:

dlx.py *(cont.)*

```
78      def choose_column(self):
79          col = self.R
80          size, bestcol = self.n + 1, None
81          while col != self:
82              if col.S < size:
83                  size, bestcol = col.S, col
84              if size == 1: break
85              col = col.R
86          return bestcol
```

Y llegamos, por fin, a la implementación de la búsqueda con retroceso:

dlx.py *(cont.)*

```
88      def backtracking(self, s):
89          if self.R == self: return s
90
91          col = self.choose_column()
92          col.cover()
93
94          first_cell_in_row = col.D
95          while first_cell_in_row != col:
96              s.add(self.row_index[first_cell_in_row])
97              cell = first_cell_in_row.R
98              while cell != first_cell_in_row:
99                  cell.H.cover()
100                 cell = cell.R
101
102             found = self.backtracking(s)
103             if found != None: return found
104
105             s.remove(self.row_index[first_cell_in_row])
106             cell = first_cell_in_row.L
107             while cell != first_cell_in_row:
108                 cell.H.uncover()
109                 cell = cell.L
110
111             first_cell_in_row = first_cell_in_row.D
112         col.uncover()
```

```
from dlx import DLX

print DLX([[1,1,0,0,0,1],
           [0,0,0,1,1,0],
           [0,0,1,1,0,1],
           [1,0,1,1,0,0],
           [0,1,0,0,1,1],
           [0,0,1,1,1,0]]).backtracking(set([]))
```

```
set([0, 5])
```

### 6.8.3. Aplicación del algoritmo DLX a otros problemas

Resulta interesante comprobar que muchos de los problemas abordables por búsqueda con retroceso pueden reducirse al problema de la cobertura exacta, es decir, dada una instancia de un problema nuevo, podemos definir una instancia del de la cobertura exacta cuya solución es solución del original.

Tomemos por caso el problema del recubrimiento con poliominós¡recubrimiento—con poliominós. Consideremos la instancia que se describe (y aparece resuelta) en la figura 6.17. Podemos definir una columna para cada una de la piezas que hemos de poner en el tablero. Hay 5: la primera para la pieza O, la segunda para la `I`, la tercera para la `L`, la cuarta para la `T` y la quinta para la `Z`. Añadimos una columna más por cada casilla del tablero, de las que hay 20 (pongamos que las columnas se asocian a las casillas siguiendo un orden de izquierda a derecha y de arriba a abajo en el tablero).

A continuación construimos una fila de la matriz para cada pieza y cada posición posible de la pieza en el tablero. La `O`, puede ponerse de 10 formas diferentes; la `I`, de 11; la `L`, de 40; la `T`, de 28; y la `Z`, de 28. La figura 6.29 muestra un resumen de la matriz.

| | O | I | L | T | Z | (0,0) | (1,0) | (2,0) | (3,0) | (4,0) | (5,0) | (6,0) | (1,1) | (2,1) | (3,1) | (4,1) | (5,1) | (6,1) | (0,2) | (1,2) | (2,2) | (3,2) | (4,2) | (5,2) | (6,2) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 6 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 7 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 9 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 10 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 11 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 12 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 20 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 116 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |

*Figura 6.29: Resumen de la matriz que modela una instancia del problema del recubrimiento con tetrominós como problema de cobertura exacta.*

Una cobertura exacta de esta matriz es una solución a la instancia del problema de los tetrominós: cada pieza se pone en el tablero una sola vez, cada casilla debe estar ocupada por a lo sumo una pieza, todas las piezas se han de poner en el tablero y toda casilla del tablero debe ocuparse con alguna pieza.

### 6.8.4. Una modificación del algoritmo DLX que permite abarcar una familia de problemas mayor

Podemos tratar de modelar el problema de las $n$ reinas como un problema de cobertura exacta. Como cada reina puede ocupar una columna del tablero y sólo puede haber una reina en cada columna, asociaremos a cada columna del tablero una columna de la matriz. Lo mismo haremos con las filas del tablero. Pero, ¿qué hacer con las $4n - 2$ diagonales? Es cierto que no podemos poner dos reinas en una diagonal, por lo que parece razonable poner una columna de la matriz para cada diagonal. Pero si lo hacemos y tratamos de resolver el problema de la cobertura exacta estaremos exigiendo que haya una reina en cada diagonal, lo que es innecesario (es más, es imposible).

El problema de la cobertura exacta no es apropiado, pero si lo es una modificación suya. En nuestra nueva versión del problema consideraremos que hay dos tipos de columna: primarias y secundarias. La solución del problema será una selección de filas con un único uno por columna primaria y con *a lo sumo* un uno por columna secundaria.

Basta con modificar la comprobación de completitud de un estado y el método de selección de la siguiente columna que cubrir: en ambos casos hemos de considerar únicamente columnas primarias. Para determinar que hemos encontrado una solución completa basta con comprobar que todas las columnas «vivas» son secundarias.

`dlx.py` *(cont.)*

```
115 class DLX2(DLX):
116     def __init__(self, ones, primary_columns):
117         DLX.__init__(self, ones)
118         i = 0
119         ptr = self.R
120         while ptr != self:
121             ptr.primary = i in primary_columns
122             ptr = ptr.R
123             i += 1
124 
125     def choose_column(self):
126         col = self.R
127         size, bestcol = self.n + 1, None
128         while col != self:
129             if col.primary and col.S < size:
130                 size, bestcol = col.S, col
131             if size == 1: break
132             col = col.R
133         return bestcol
134 
```

```
135     def all_primaries_are_covered(self):
136         ptr = self.R
137         while ptr != self:
138             if ptr.primary: return False
139             ptr = ptr.R
140         return True
141 
142     def backtracking(self, s):
143         if self.all_primaries_are_covered(): return s
144 
145         col = self.choose_column()
146         col.cover()
147 
148         first_cell_in_row = col.D
149         while first_cell_in_row != col:
150             s.add(self.row_index[first_cell_in_row])
151             cell = first_cell_in_row.R
152             while cell != first_cell_in_row:
153                 cell.H.cover()
154                 cell = cell.R
155 
156             found = self.backtracking(s)
157             if found != None: return found
158 
159             s.remove(self.row_index[first_cell_in_row])
160 
161             cell = first_cell_in_row.L
162             while cell != first_cell_in_row:
163                 cell.H.uncover()
164                 cell = cell.L
165 
166             first_cell_in_row = first_cell_in_row.D
167         col.uncover()
```

Apliquemos el nuevo algoritmo al problema de las $n$ reinas:

```
1  from dlx import DLX2
2  
3  def nqueens(n):
4      M = []
5      for i in xrange(n):
6          for j in xrange(n):
7              M.append( [0] * (6*n-2) )
8              M[-1][i] = 1
9              M[-1][n + j] = 1
10             M[-1][2*n+ n-1 - i+j] = 1
11             M[-1][4*n-1 + i+j] = 1
12     s = DLX2(M, range(2*n)).backtracking(set([]))
13     if s:
14         sol = [None] * n
```

```
15          for r in s:
16              col, row = M[r][:n].index(1), M[r][n:2*n].index(1)
17              sol[col] = row
18          return sol
19      return None
20
21  for n in xrange(1, 9):
22      print "Solución con %d reinas: %s." % (n, nqueens(n))
```

```
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].
```

¿Ha valido la pena tanta complicación para resolver de otro modo lo que ya habíamos resuelto? La verdad es que sí. La eficiencia del nuevo método es mucho mayor. Dado que en cada paso selecciona eficientemente la mejor forma de ramificar (las filas que permiten cubrir la columna con menos unos, es decir, el menor factor de ramaje posible), se visitan muchos menos estados a la hora de dar con una solución. La tabla 6.3 permite comparar el número de estados visitados con la búsqueda con retroceso convencional y con el algoritmo DLX2. La diferencia a favor de DLX2 es, sencillamente, brutal.

### EJERCICIOS

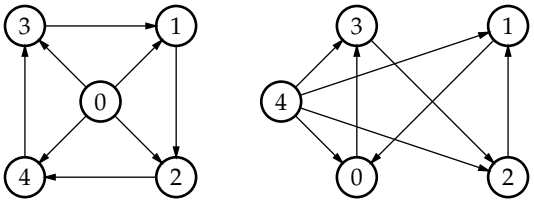
**6-22** Diseña un algoritmo de búsqueda con retroceso que encuentre la salida de un laberinto generado con el algoritmo propuesto en el ejercicio 4-23 (página 4-23).

**6-23** Diseña un algoritmo que, dado un tablero con casillas blancas y negras y una lista de palabras, genere un crucigrama que ocupe todas las casillas blancas.

**6-24** Diseña un algoritmo que encuentre un recorrido de un tablero de ajedrez de $n \times n$ con un caballo de modo que se visiten todas las casillas, pero que ninguna se visite más de una vez.

**6-25** Diseña un programa que dados dos grafos dirigidos determine si son isomorfos o no. Dos grafos son isomorfos si son el mismo cuando se renombran sus vértices.

He aquí dos grafos isomorfos:




Si renombramos los vértices de modo que el vértice 0 pase a ser el 4, el 1 pase a ser el 2, el 2 pase a ser el 1, el 3 siga siendo el 3 y el 4 pase a ser el 0, ambos grafos son exactamente el mismo.

**6-26** Diseña un algoritmo de búsqueda con retroceso que, dados un grafo y un valor $k$, asigne un color a cada vértice con uno de los $k$ colores de modo que no haya dos vértices adyacentes con un mismo color.

*Tabla 6.3: Número de estados visitados al resolver el problema de las n reinas por el método convencional de búsqueda con retroceso y con el algoritmo DLX2.*

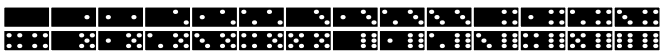
| | **estados** | |
|---|---|---|
| $n$ | método clásico | algoritmo DLX2 |
| 1 | 1 | 2 |
| 2 | 6 | 3 |
| 3 | 18 | 6 |
| 4 | 26 | 9 |
| 5 | 15 | 6 |
| 6 | 171 | 32 |
| 7 | 42 | 8 |
| 8 | 876 | 76 |
| 9 | 333 | 24 |
| 10 | 975 | 31 |
| 11 | 517 | 70 |
| 12 | 3 066 | 96 |
| 13 | 1 365 | 142 |
| 14 | 26 495 | 79 |
| 15 | 20 280 | 33 |
| 16 | 160 712 | 45 |
| 17 | 91 222 | 51 |
| 18 | 743 229 | 52 |
| 19 | 48 184 | 85 |
| 20 | 3 992 510 | 127 |
| 21 | 179 592 | 37 |
| 22 | 38 217 905 | 54 |
| 23 | 584 591 | 37 |
| 24 | 9 878 316 | 61 |

**6-27** Las $n$ parejas ($n$ hombres y $n$ mujeres) que representan a otros tantos países asisten a una importante cena diplomática y deben sentarse en una mesa redonda. El organizador debe observar un estricto protocolo: se deben alternar hombres y mujeres, pero de modo tal que no haya dos personas de una misma pareja en posiciones contiguas ni se sienten juntas a personas de dos embajadas con malas relaciones. Una tabla de $n \times n$ elementos nos indica si dos países mantienen malas relaciones.

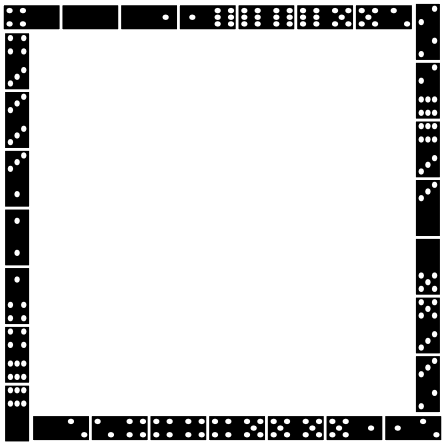
Diseña un algoritmo de búsqueda con retroceso que proporcione, si la hay, una asignación de asientos no conflictiva.

**6-28** Una agencia matrimonial dispone de un fichero informatizado con todos sus clientes, $n$ hombres y $n$ mujeres. Cada cliente ha puntuado, en una escala de 0 a 10, a cada una de las personas del otro sexo. Queremos cerrar la agencia y casar a todos los hombres con todas las mujeres. Un matrimonio es inestable si uno de sus miembros preferiría estar casado/a con el miembro de otra pareja antes que con su cónyuge y si, al mismo tiempo, esa otra persona preferiría estar casada/o con el primero antes que con su respectivo cónyuge. Diseña un algoritmo que nos diga si es posible formar $n$ matrimonios estables y, si lo es, nos diga cómo formar los $n$ matrimonios.

**6-29** Hay 28 piezas de dominó:

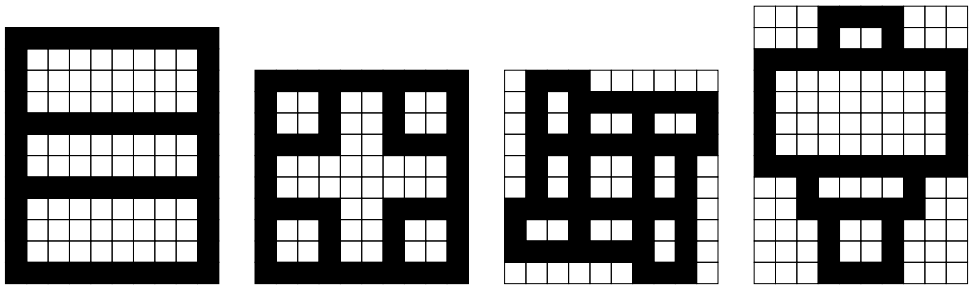


Podemos diseñar figuras con las piezas de dominó respetando la regla de que dos piezas puede unirse sólo si hacen coincidir números iguales. Se puede conseguir, por ejemplo, realizar un marco de $15 \times 15$ con las 28 piezas:



Diseña un algoritmo de búsqueda con retroceso que reciba una descripción de la región que deseamos «rellenar» con piezas de dominó, nos indique si es posible efectuar el relleno y, en tal caso, cómo disponer las piezas.

Aquí tienes algunas figuras realizables con las 28 piezas:



**6-30** La «alfamética» se encarga de los problemas aritméticos con palabras cuyas letras son dígitos. Dos letras distintas no pueden corresponder al mismo dígito y el dígito asociado a la letra que inicia una palabra no puede ser 0. El primer problema alfamético conocido consiste en averiguar la correspondencia entre dígitos y letras para que sea cierta esta operación:

```
  SEND
+ MORE
------
 MONEY
```

Su (única) solución es ésta:

```
  9567
+ 1085
------
 10652
```

He aquí un par de problemas alfaméticos:

- SEVEN + SEVEN + SIX = TWENTY
- TERRIBLE + NUMBER = THIRTEEN

Diseña un algoritmo de búsqueda con retroceso que resuelva problemas alfaméticos. Cada problema se describe con una lista de $n$ palabras: las $n-1$ primeras son sumandos y la última es el resultado.

**6-31** Un barquero debe llevar un zorro, un pato y un saco de maíz de una orilla a otra. En el bote sólo caben el barquero y un animal o el saco de maíz. El problema estriba en que nunca pueden quedar solos, en una misma orilla y sin vigilancia, el zorro y el pato o el pato y el maíz, pues el primero siempre se come al segundo. Diseña un algoritmo que resuelva el problema.

**6-32** Diseña un algoritmo que inserte signos de adición o diferencia en una cadena de dígitos para formar una expresión cuyo valor sea una cantidad dada $N$. Por ejemplo, si la cadena es 123456789 y hemos de obtener el valor 99, podemos insertar los símbolos de suma y resta así: $12+3-4-5+6+78+9=99$.

**6-33** Nos sirven $n$ cartas con números impresos entre 1 y 999 y $n-1$ cartas con las operaciones de suma y producto (`+` y `*`). Podemos interpretar una secuencia de cartas como instrucciones sobre una máquina de pila. La secuencia «2, 10, 2, +, 1, *, 3, +» se interpreta como: «apila el valor 2; apila el valor 10; apila el valor 2; sustituye los dos elementos más altos en la pila por su suma; apila el valor 1; sustituye los dos elementos más altos en la pila por su producto; ...».

Diseña un algoritmo que, dado un conjunto de $n$ cartas con números y $n-1$ operaciones encuentre, si la hay, una forma de obtener exactamente un valor dado al que denotaremos con $m$.

**6-34** Las expresiones booleanas pueden representarse con árboles de sintaxis abstracta cuyos vértices interiores son operaciones (y-lógica, o-lógica o negación) y cuyas hojas son identificadores de variable o constantes lógicas. Por ejemplo, la expresión «*a* **or** *b* **and not** (*False* **or** *c*)» puede representarse con un árbol así:

```
['or', ['a'], ['and', ['b'], ['not', ['or', ['False'], ['c']]]]]
```

Dada una expresión booleana expresada con un árbol de sintaxis abstracta, ¿hay alguna configuración de sus variables que la evalúe a «cierto»? Diseña un programa de búsqueda con retroceso que determine si existe tal configuración y, en caso afirmativo, la muestre.

**6-35** Un Sudoku es un tablero de $9 \times 9$ casillas dividido en 9 bloques de $3 \times 3$ casillas. Algunas de estas casillas contienen números ente 1 y 9. El objetivo es poner un número entre 1 y 9 en cada casilla vacía de modo que cada fila y columna del tablero y cada bloque de $3 \times 3$ casillas contenga los 9 números. He aquí un Sudoku:

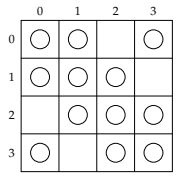
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | 3 | 1 | 6 | | 5 | 9 |
| | | 6 | | | | 8 | | 7 |
| | | | | | | 2 | | |
| | 5 | | | 3 | | | 9 | |
| 7 | 9 | | 6 | | 2 | | 1 | 8 |
| | 1 | | | 8 | | | 4 | |
| | | 8 | | | | | | |
| 3 | | 9 | | | | 6 | | |
| 5 | 6 | | 8 | 4 | 7 | | | |

Diseña un algoritmo de búsqueda con retroceso para resolver Sudokus.

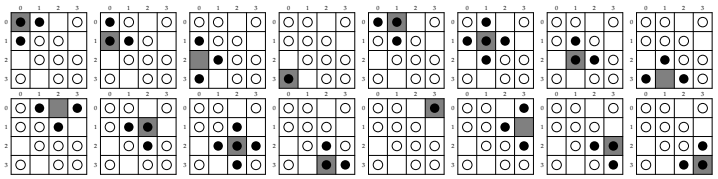
**6-36** Modela la resolución de Sudokus como un problema de cobertura exacta y aplica el algoritmo DLX (o DLX2).

**6-37** Dado un tablero de $n \times n$ casillas, definimos el vecindario de la casilla de coordenadas $(i, j)$ como ella misma y las (hasta) cuatro casillas adyacentes (en los bordes, el vecindario puede tener sólo 3 o 2 casillas adyacentes). En cada casilla podemos poner un ficha. Decimos que un vecindario es impar si hay un número impar de fichas en las casillas que lo forman. Una disposición de fichas en el tablero es *vecino-impar* si todos los vecindarios del tablero son impares.

Disponemos de $m$ fichas y deseamos saber si es posible encontrar una disposición *vecino-impar* que use, como mucho, $m$ fichas. He aquí un ejemplo de disposición *vecino-impar* para $n=4$ y $m=13$ (el valor de $m$ es 13 aunque la solución sólo usa 12 fichas):



Te mostramos a continuación los 16 vecindarios del tablero de $4 \times 4$ y la disposición de fichas *vecino-impar* del ejemplo (con sus respectivas casillas centrales en gris) para que compruebes que todos contienen un número impar de fichas (marcadas en negro):



Se pide un algoritmo de búsqueda con retroceso que, dados $n$ y $m$, encuentre, si la hay, una configuración *vecino-impar*. Discute los aspectos relativos a la eficiencia de los diferentes elementos que conforman tu propuesta.

(Ten cuidado con la determinación de factibilidad en un estado. Si pones una ficha en el tablero, no necesariamente conoces si su vecindad es par o impar en el momento de hacerlo: aún no sabes cuántas fichas habrá finalmente en todas las casillas del vecindario.)