

Estructuras de Datos y Algoritmos

Facultad de Informática

Universidad Politécnica de Valencia

Curso 2007/2008

Tema 7:

Búsqueda con retroceso

TEMA 7. Búsqueda con retroceso

Búsqueda con retroceso: Técnica para explorar un conjunto de soluciones hasta encontrar una factible (satisface un conjunto de restricciones).

Contenidos

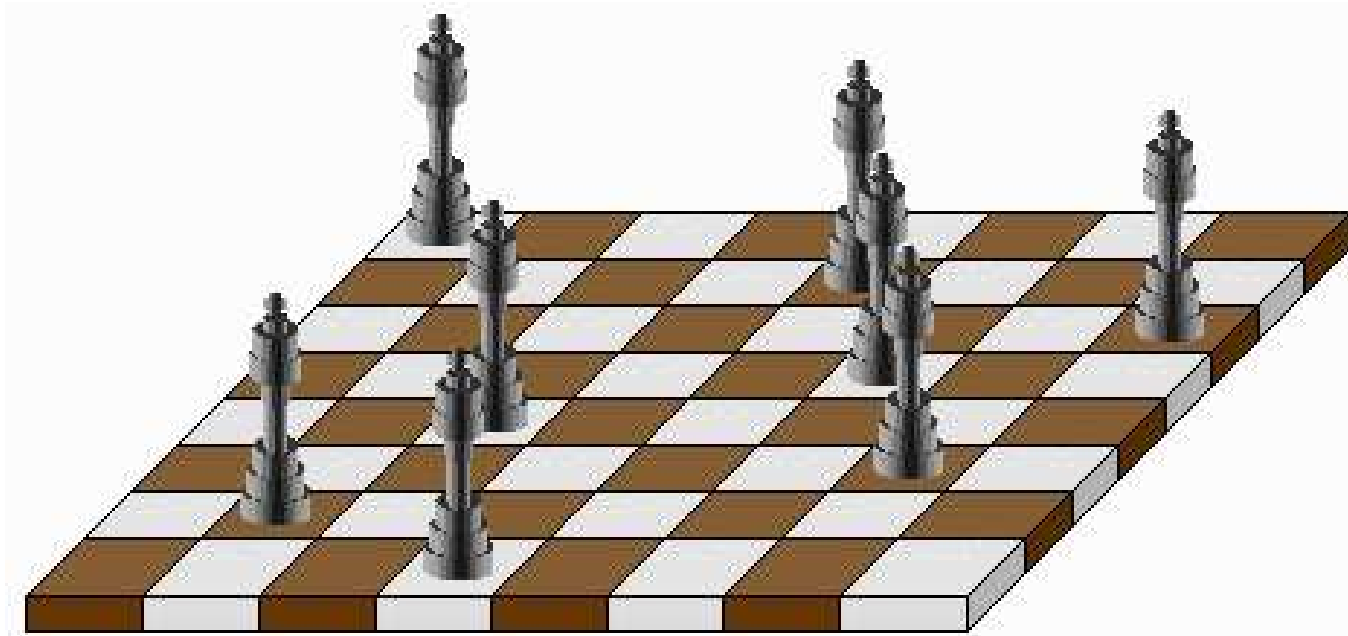
- 1 Introducción: El problema de las n -reinas.
- 2 Esquema general de la búsqueda con retroceso.
- 3 La suma de subconjuntos.
- 4 Sudoku.
- 5 Cubrimiento exacto de un conjunto (Exact Cover)
- 7 Reducción del Sudoku al cubrimiento exacto.
- 6 Reducción de las n -reinas al cubrimiento exacto.
- 8 Ciclo Hamiltoniano: el salto del caballo (Knight's tour)
- 9 Ciclo Euleriano: secuencia de fichas de dominó.

Bibliografía

- *Fundamentos de Algoritmia*, de Brassard y Bratley (apartado 9.6) [BB97]
- *Computer algorithms*, de Horowitz, Sahni y Rajasekaran (capítulos 7 y 8) [HSR98]
- *Problems on Algorithms*, de Parberry (apartado 10.5, problemas) [Par95]

1. Problema de las 8 reinas

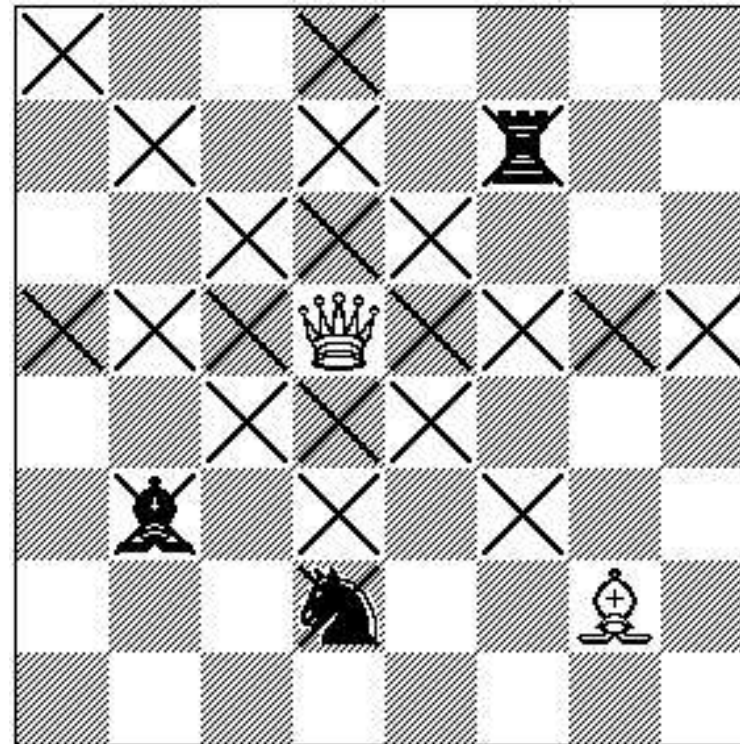
Colocar 8 reinas en un tablero de ajedrez de tal modo que ninguna de ellas amenace a las demás.



1. Problema de las 8 reinas

Colocar 8 reinas en un tablero de ajedrez de tal modo que ninguna de ellas amenace a las demás.

Una reina se puede mover a cualquier distancia en horizontal, vertical o diagonal



1. Problema de las 8 reinas fuerza bruta

- Explorar todas las formas posibles de colocar 8 piezas en el tablero.

Combinaciones: $\binom{64}{8} = 4\,426\,165\,368$

- Poner 1 reina por columna. Ejemplo (2, 0, 5, 1, 7, 5, 2, 6) indica una reina en columna 0 y fila 2, otra reina en columna 1 y fila 0, etc.

Combinaciones: $8^8 = 16\,777\,216$.

- **Restricciones implícitas:** Son las reglas que restringen cada x_i a tomar únicamente los valores de un conjunto determinado S_i . Ejemplo: por la forma de representar el tablero, no habrá más de una reina en una misma columna.
- **Restricciones explícitas:** Describen la forma en que los distintos valores x_i debe relacionarse entre sí. Por ejemplo, que dos reinas no estén en la misma fila ni diagonal.

1. Problema de las 8 reinas

¿Es posible un algoritmo voraz?

Introducimos la primera reina

●	×	×	×	×	×	×	×
×	×						
×		×					
×			×				
×				×			
×					×		
×						×	
×							×

1. Problema de las 8 reinas

¿Es posible un algoritmo voraz?

Introducimos la segunda reina

●	×	×	×	×	×	×	×
×	×	×					
×	●	×	×	×	×	×	×
×	×	×	×				
×	×		×	×			
×	×			×	×		
×	×				×	×	
×	×					×	×

1. Problema de las 8 reinas

¿Es posible un algoritmo voraz?

Introducimos la tercera reina

●	×	×	×	×	×	×	×
×	×	×			×		
×	●	×	×	×	×	×	×
×	×	×	×				
×	×	●	×	×	×	×	×
×	×	×	×	×	×		
×	×	×		×	×	×	
×	×	×			×	×	×

1. Problema de las 8 reinas

¿Es posible un algoritmo voraz?

Introducimos la cuarta reina

●	×	×	×	×	×	×	×
×	×	×	●	×	×	×	×
×	●	×	×	×	×	×	×
×	×	×	×		×		
×	×	●	×	×	×	×	×
×	×	×	×	×	×	×	
×	×	×		×	×	×	×
×	×	×			×	×	×

1. Problema de las 8 reinas

¿Es posible un algoritmo voraz?

No encontramos solución

●	×	×	×	×	×	×	×
×	×	×	●	×	×	×	×
×	●	×	×	×	×	×	×
×	×	×	×		×		
×	×	●	×	×	×	×	×
×	×	×	×	×	×	×	
×	×	×		×	×	×	×
×	×	×			×	×	×

1. Problema de las 8 reinas

¿Es posible un algoritmo voraz?

- Con la estrategia voraz nunca se vuelve a cuestionar una decisión una vez ha sido tomada.
- En un caso general, esto resulta demasiado restrictivo: acabamos de ver que **no** permite resolver cualquier problema.

2. Búsqueda con retroceso Backtracking

- Las soluciones del problema se pueden representar como una n -tupla (x_0, \dots, x_{n-1}) .
- El objetivo consiste en encontrar soluciones factibles.
- La idea consiste en construir el vector solución elemento a elemento usando una función factible modificada para estimar si una solución **parcial** o **incompleta** tiene alguna posibilidad de éxito.

2. Búsqueda con retroceso Backtracking

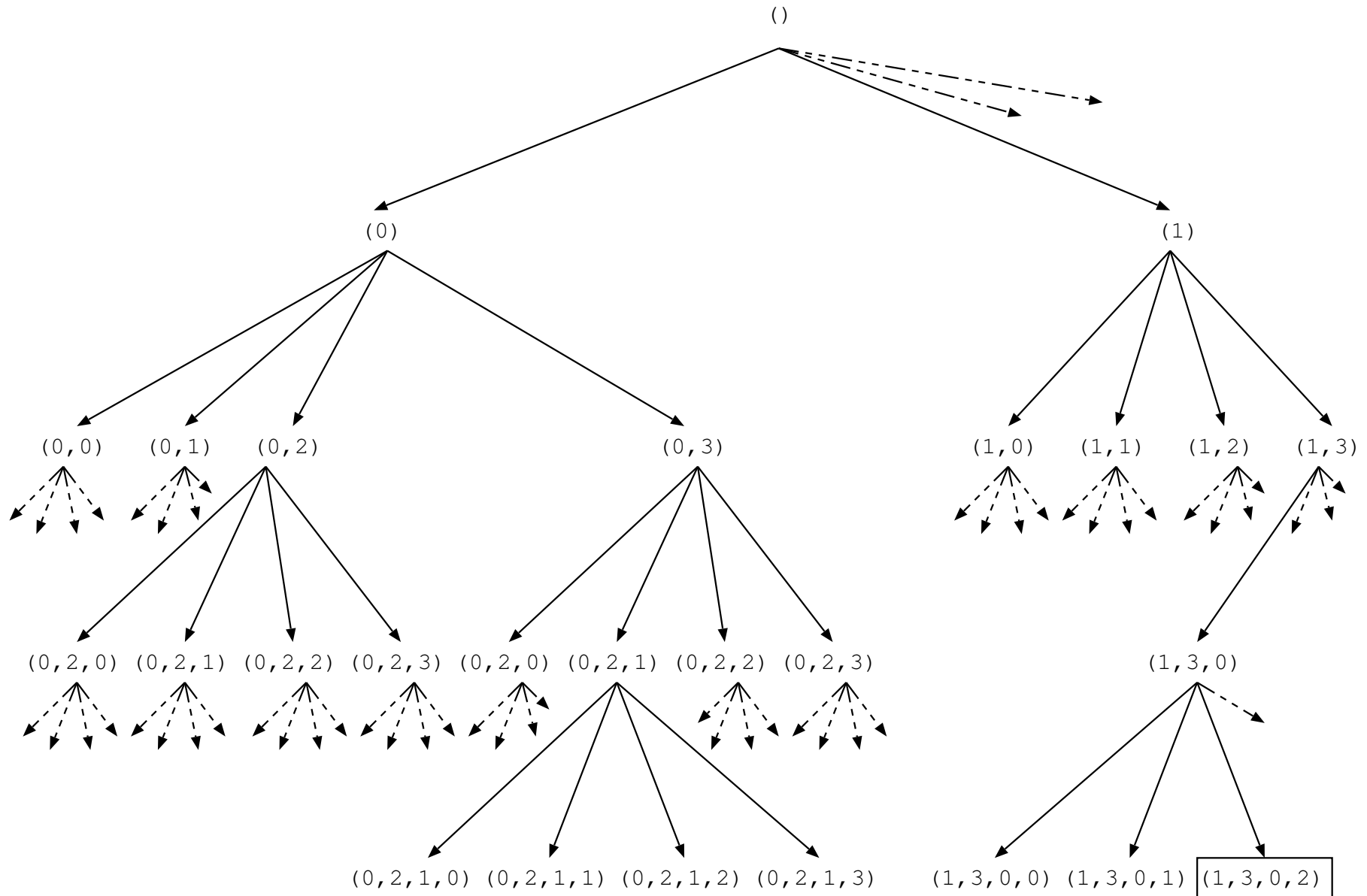
- Cada una de las tuplas $(x_0, \dots, x_i, ?)$ donde $i \leq n$ se denomina un **estado** y denota un conjunto de soluciones.
- Un estado puede ser:
 - **estado terminal** o **solución**: describe un conjunto con un sólo elemento;
 - **estado no terminal** o **solución parcial**: representa implícitamente un conjunto de varias soluciones;

En el ejemplo de las n -reinas, los estados terminales son aquellos con n valores.

- Se dice que un estado no terminal es **factible** o **prometedor** cuando no se puede descartar que contenga alguna solución factible.
- El conjunto de estados se organiza formando un **árbol de estados**.

2. Búsqueda con retroceso

Arbol de estados



2. Búsqueda con retroceso

Arbol de estados

- Un estado puede dividirse en otros estados fijando otro elemento de la tupla, generando un proceso de **ramificación** que parte de un **estado inicial** y que induce el árbol de estados.
- El **estado inicial** es la raíz del árbol de estados y contiene implícitamente todo el espacio de soluciones.
- Si el árbol de estados tiene un número elevado de nodos, será imposible construirlo explícitamente antes de aplicar una de las técnicas de recorrido. Se puede utilizar un “árbol implícito”, en el sentido de que se van construyendo sus parte relevantes a medida que avanza el proceso de recorrido.
- Búsqueda con retroceso = **recorrido primero en profundidad**.

2. Búsqueda con retroceso

Esquema general

```

1 backtracking(s) {           // s es un estado o conjunto
2   if (es_terminal(s)) {    // de soluciones, normalmente
3     if (es_factible(s))    // es un vector v[1..k]
4       escribir_solucion(s); // y es_terminal(s) equivale a k==n
5   } else // si no es terminal entonces se ramifica
6     for s2 in ramificar(s) { // s2 es de la forma v[1..k+1]
7       if (es_prometedor(s2)) // poda por factibilidad
8         backtracking(s2);   // llamada recursiva
9     }
10 }

```

La condición:

```
if (es_prometedor(s2)) //poda por factibilidad
```

evita descender por ramas del árbol de estados generadas a partir de un estado no prometedor, por lo que se reduce significativamente el espacio de búsqueda a explorar sin que por ello se pierda ninguna solución. Esto se denomina **poda por factibilidad**.

2. Búsqueda con retroceso

Ejemplo: n reinas

```

41 void reinas::calcula_reinas(int k) {// k es num reinas q llevamos
42     if (k==n) {
43         escribe_solucion();
44     } else
45         for(int i=0; i<n; i++) {
46             if (fila[i] && DIAG45(k,i) && DIAG135(k,i)) {
47                 columna[k]=i; // apuntamos para escribir la solucion
48                 fila[i] = DIAG45(k,i)= DIAG135(k,i)= false; // marcamos
49                 calcula_reinas(k+1); // llamada recursiva
50                 fila[i] = DIAG45(k,i)= DIAG135(k,i)= true; // restauramos
51             }
52         }
53 }

```

Este código sigue casi el esquema propuesto, como se puede observar si el estado procesado es terminal no se mira si es factible, esto es así porque para las n -reinas se cumple:

prometedor + terminal \Rightarrow factible

2. Búsqueda con retroceso

Ejemplo: n reinas

```
1 using namespace std;
2 #include<iostream>
3 #include<stdlib.h>
4 class reinas { //Problema de las N reinas. Version vuelta atras
5     int n, soluciones; // tablero de nxn, contador de soluciones
6     int *columna; // configuracion del tablero, 1 reina por columna
7     bool *fila, *diag45, *diag135; // para comprobar si se amenazan
8     bool& DIAG45(int col, int fil) { return diag45[col+fil]; }
9     bool& DIAG135(int col, int fil) { return diag135[(n-1)-col+fil]; }
10 public:
11     reinas(int n);
12     ~reinas();
13     void escribe_solucion();
14     void calcula_reinas(int k); // recursivo
15 };
16 reinas::reinas(int n) {
17     this->n = n; soluciones = 0;
18     columna = new int[n]; diag45 = new bool[2*n];
19     fila = new bool[n]; diag135 = new bool[2*n];
20     for(int i=0; i<n; i++) { // inicializar vectores
```

2. Búsqueda con retroceso

Ejemplo: n reinas

```

21     columna[i]=-1; fila[i] = true; // true indica "libre"
22     diag45[i*2] = diag45[i*2+1] = true;
23     diag135[i*2] = diag135[i*2+1] = true;
24 } }
25 reinas::~reinas() { // liberar memoria
26     delete[] columna; delete[] diag45;
27     delete[] fila;     delete[] diag135;
28 }
29 void reinas::escribe_solucion() {
30     soluciones++; cout << "Solucion_" << soluciones << endl;
31     for(int i=0;i<n;i++) {
32         cout << "_"; for(int j=0;j<n;j++) cout<<"---_";
33         cout << "\n|";
34         for(int j=0;j<n;j++)
35             cout << ((columna[j] == i) ? "_Q" : "_") << "_|";
36         cout << endl;
37     }
38     cout<<"_"; for(int j=0;j<n;j++) cout<<"---_";
39     cout << endl << endl;
40 }

```

2. Búsqueda con retroceso

Ejemplo: n reinas

```
41 void reinas::calcula_reinas(int k) {// k es num reinas q llevamos
42     if (k==n) {
43         escribe_solucion();
44     } else
45         for(int i=0; i<n; i++) {
46             if (fila[i] && DIAG45(k,i) && DIAG135(k,i)) {
47                 columna[k]=i; // apuntamos para escribir la solucion
48                 fila[i] = DIAG45(k,i)= DIAG135(k,i)= false; // marcamos
49                 calcula_reinas(k+1); // llamada recursiva
50                 fila[i] = DIAG45(k,i)= DIAG135(k,i)= true; // restauramos
51             }
52         }
53 }
54 int main(int argc, char **argv) {
55     if (argc != 2) { cerr<<"Uso: _" << argv[0] << "_numero_de_reinas" << endl;
56         exit(1); }
57     int n=atoi(argv[1]); cout<<"Problema_de_las_" << n << "-Reinas" << endl;
58     reinas(n).calcula_reinas(0);
59     return 0;
60 }
```

2. Búsqueda con retroceso

Corrección

Para que el esquema de búsqueda con retroceso conduzca a un procedimiento correcto, la función de ramificación debe satisfacer ciertas condiciones:

Condición 1 La ramificación de un estado no terminal produce subconjuntos propios suyos: Para todo estado s con $|s| > 1$ y para todo $s_2 \in \text{ramifica}(s)$, se cumple $s_2 \subset s$

Condición 2 Toda solución factible es parte de algún subconjunto: Para todo estado s con $|s| > 1$, $\cup_{s_2 \in \text{ramifica}(s)} s_2 = s$.

Condición 3 (opcional) No solapamiento entre estados hermanos: Para todo estado s con $|s| > 1$, si tanto s_2 como s_3 pertenecen a $\text{ramifica}(s)$ y $s_2 \neq s_3$ entonces $s_2 \cap s_3 = \emptyset$.

Con las dos primeras condiciones, se puede probar el siguiente teorema:

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

2. Búsqueda con retroceso

Coste espacial

El coste espacial dedende de:

- el espacio necesario para la tupla con que se representa el estado,
- el espacio de la pila de las llamadas recursivas.

Conviene tener una sola variable para la tupla, de modo que cada llamada recursiva añada información a la misma y la suprima al efectuar el retroceso. De esta manera, aunque el coste temporal pueda ser mucho mayor, el coste espacial puede mantenerse en $O(n)$.

Observa que con esta forma de proceder el bucle:

```
for s2 in ramificar(s) { //s2 es de la forma v[1..k+1]
```

no es más que poner el valor adecuado en la posición $v[k+1]$ en cada iteración, de modo que en un instante dado, hay un único estado representado por la única variable para la tupla, el vector v .

2. Búsqueda con retroceso

Coste temporal

- El coste de los algoritmos de búsqueda con retroceso depende del tamaño explorado en el árbol de estados y del coste de explorar cada nodo (el coste de ramificar y de verificar si un estado es prometedor o factible).
- El tamaño del árbol de estados puede crecer exponencialmente con la talla del problema. Por tanto, en el peor caso, puede que tengamos que recorrer todo el árbol.
- Los estados podados (poda por factibilidad) permiten evitar explorar subárboles enteros cuyo tamaño también es enorme.
- Es importante intentar reducir el coste de cada paso: muchas veces se puede comprobar si es factible o prometedor de forma incremental y en coste constante.

3. Problema de la suma de subconjuntos

Disponemos de N objetos con pesos $(w_0, w_1, \dots, w_{N-1})$ y de una mochila con capacidad para soportar una carga W . Deseamos cargar la mochila con una selección de objetos cuyo peso sea **exactamente** W .

Por ejemplo, si $N = 7$, los objetos pesan $(5, 10, 7, 3, 1, 1, 2)$ y $W = 12$, las distintas soluciones al problema son:

Solucion: $w[0] = 5\text{Kg}$. $w[2] = 7\text{Kg}$.

Solucion: $w[0] = 5\text{Kg}$. $w[3] = 3\text{Kg}$. $w[4] = 1\text{Kg}$. $w[5] = 1\text{Kg}$. $w[6] = 2\text{Kg}$.

Solucion: $w[1] = 10\text{Kg}$. $w[4] = 1\text{Kg}$. $w[5] = 1\text{Kg}$.

Solucion: $w[1] = 10\text{Kg}$. $w[6] = 2\text{Kg}$.

Solucion: $w[2] = 7\text{Kg}$. $w[3] = 3\text{Kg}$. $w[4] = 1\text{Kg}$. $w[5] = 1\text{Kg}$.

Solucion: $w[2] = 7\text{Kg}$. $w[3] = 3\text{Kg}$. $w[6] = 2\text{Kg}$.

Una posible forma de representar una solución es mediante un vector $(x_0, x_1, \dots, x_{N-1})$ de $\{0, 1\}^N$. Si x_i vale 1, el objeto de peso w_i se carga en la mochila, si $x_i = 0$ no. Hay, pues, 2^N soluciones completas.

Una solución completa es factible si $\sum_{0 \leq i \leq N-1} x_i w_i = W$

3. Problema de la suma de subconjuntos

¿Cómo poder determinar si un estado $(x_0, x_1, \dots, x_i, ?)$ con $i \leq N - 1$ es prometedor?

Una forma sencilla es ver si la suma de sus pesos es menor o igual que W ya que, en caso contrario, seguro que nos hemos pasado de capacidad (aunque ya no metamos nada más en la mochila):

$$\text{prometedor}((x_0, \dots, x_i, ?)) \equiv \sum_{j=0}^i x_j w_j \leq W$$

Podemos refinar un poco más añadiendo una condición extra, y es que con todos los objetos que quedan por meter tendremos que llegar a la capacidad de la mochila:

$$\text{prometedor}((x_0, \dots, x_i, ?)) \equiv \left(\sum_{j=0}^i x_j w_j \leq W \right) \wedge \left(\sum_{j=0}^i x_j w_j + \sum_{j=i+1}^{N-1} w_j \geq W \right)$$

Ventaja: prometedor y terminal implica factible :)

3. Problema de la suma de subconjuntos

```
1 using namespace std;
2 #include<iostream>
3 #include<iomanip>
4 class sumas {
5     int N, W, suma, *x, *w, *maxqueda;
6 public:
7     sumas(int N, int *w, int W);
8     ~sumas();
9     void procesar_solucion();
10    void backtracking(int k); // recursivo
11 };
12 sumas::sumas(int num, int *pesos, int maxpes) {
13     N = num; W = maxpes; w = new int[N];
14     for(int i=0; i<N; i++) w[i] = pesos[i];
15     x = new int[N]; maxqueda = new int[N]; suma = 0;
16     maxqueda[N-1] = 0;
17     for(int i=N-2; i >= 0; i--)
18         maxqueda[i] = w[i+1]+maxqueda[i+1];
19 }
20 sumas::~~sumas() { delete[] x; delete[] w; delete[] maxqueda; }
```

3. Problema de la suma de subconjuntos

```
21 void procesar_solucion() { cout << "Solucion:";
22   for(int i=0;i<N;i++)
23     if (x[i] == 1)
24       cout<<"_w["<<i<<"]="<<setw(2)<<w[i]<< "Kg.";cout<<endl;
25 }
26 void sumas::backtracking(int i) {
27   if (i == N) {
28     procesar_solucion();
29   } else { // consideramos objeto i-esimo:
30     suma += w[i]; // suponemos que incluimos el objeto
31     if ((suma <= W) && (suma+maxqueda[i] >= W))
32       { x[i] = 1; backtracking(i+1); }
33     suma -= w[i]; // restaturamos el estado
34     // caso en que no incluimos el objeto
35     if (suma+maxqueda[i] >= W) // seguro que suma <= W
36       { x[i] = 0; backtracking(i+1); }
37   }
38 }
39 int main(int argc, char **argv) {
40   int N=7, w[]={5,10,7,3,1,1,2}, W=12;
41   sumas(N,w,W).backtracking(0);
42 }
```

4. Sudoku

- Aunque fue publicado en una revista de puzzles americana en 1979, ganó popularidad en Japón mucho más tarde (2005?). “Sudoku” es la abreviación japonesa de la frase “suuji wa dokushin ni kagiru” que significa “los dígitos deben quedar únicos”.
- Es un puzzle de dígitos numéricos del 1 al 9. Se trata de una matriz de 9×9 formada por 3×3 submatrices (llamadas “regiones”). Algunas casillas tienen valores ya dados. El objetivo consiste en rellenar todas las posiciones con estas restricciones:

Cada fila, columna y region debe tener una sola instancia de cada dígito.

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

4. Sudoku

- Existen otras dos restricciones para definir un Sudoku:
 - Debe tener una única solución.
 - Debe poder resolverse utilizando la “lógica”.
- Existen algoritmos que resuelven Sudokus simulando determinados razonamientos lógicos.
- Nosotros vamos a ignorar estas dos restricciones y cualquier tipo de simulación de razonamiento. Si existen varias soluciones, podríamos encontrarlas todas.

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

4. Sudoku

```
1 #include <iostream>
2 using namespace std;
3 class sudoku {
4     int matrix[9][9];
5     bool use_row[9][9];    // digit row
6     bool use_col[9][9];   // digit col
7     bool use_box[9][3][3]; // digit box_row box_col
8     void write_sudoku(ostream &f);
9     int count_solution;
10 public:
11     bool read_sudoku(istream &);
12     void solve_sudoku(int pos);
13 };
14 int main() {
15     sudoku s;
16     if (!s.read_sudoku(cin))
17         cout << "El sudoku propuesto no se puede resolver\n";
18     else
19         s.solve_sudoku(0);
20     return 0;
21 }
```

4. Sudoku

```
22 bool sudoku::read_sudoku(istream &f) {
23     for (int r=0;r<9;r++) //----- CLEAR
24         for (int c=0;c<9;c++) {
25             use_row[r][c] = use_col[r][c] = use_box[r][c/3][c%3] = false;
26         }
27     count_solution = 0;
28     for (int i=0;i<9;i++) { //----- READ
29         char line[10]; f >> line;
30         for (int j=0;j<9;j++) matrix[i][j] = line[j]-'1';
31     }
32     for (int r=0;r<9;r++) //----- CHECK
33         for (int c=0;c<9;c++) {
34             int d = matrix[r][c];
35             if (d != -1) {
36                 if (use_row[d][r] || use_col[d][c] || use_box[d][r/3][c/3])
37                     return false;
38                 use_row[d][r] = use_col[d][c] = use_box[d][r/3][c/3] = true;
39             }
40         }
41     return true;
42 }
```

4. Sudoku

```
54 void sudoku::solve_sudoku(int pos) {
55     if (pos == 81) { write_sudoku(cout); return; }
56     int r=pos/9, c=pos%9;
57     if (matrix[r][c] != -1) { solve_sudoku(pos+1); return; }
58     for (int digit = 0; digit<9; digit++) // ramificar
59         if (!use_row[digit][r] &&
60             !use_col[digit][c] &&
61             !use_box[digit][r/3][c/3]) {
62         matrix[r][c] = digit;
63         use_row[digit][r] = true; // reservar
64         use_col[digit][c] = true;
65         use_box[digit][r/3][c/3] = true;
66         solve_sudoku(pos+1); // llamada recursiva
67         use_row[digit][r] = false; // restaurar
68         use_col[digit][c] = false;
69         use_box[digit][r/3][c/3] = false;
70     }
71     matrix[r][c] = -1; // restaurar
72 }
```

EJERCICIO

Coloreado de grafos

Dado un grafo y N colores, se pide:

- asociar un color a cada vértice
- todas las aristas del grafo deben conectar vértices de color diferente

Ejemplo:

```
> colorear 3 grafo.txt <- num colores y fichero grafo
> cat grafo.txt
5      <- num vertices
0 1    <- arista = par de vertices
0 2
1 2
2 3
3 4
2 4
```

EJERCICIO

Reducción del sudoku al coloreado de grafos

¿Cómo convertirías el problema de resolver un sudoku al de colorear un grafo?

¿Qué serían los colores? ¿Y los vértices del grafo? ¿Y las aristas?

5. Problema del cubrimiento exacto (“exact cover”)

Dado un universo de elementos U y una colección finita de subconjuntos $S_i \subseteq U$, un cubrimiento exacto es una colección de algunos subconjuntos S_i tal que cada elemento de U aparece en exactamente uno de ellos.

Por ejemplo, Noé tiene que subir al arca una pareja de cada especie, tenemos 7 especies (para simplificar), que sería el “universo” del problema: perros, gatos, caballos, ratones, tigres, leones y elefantes.

Le proponen los siguientes conjuntos de parejas:

conjunto A caballos tigres leones

conjunto B perros, ratones, elefantes.

conjunto C gatos, caballos, leones.

conjunto D perros, ratones.

conjunto E gatos, elefantes.

conjunto F ratones, tigres, elefantes

¿Hay alguna combinación que lé de una pareja de cada?

5. Problema del cubrimiento exacto (“exact cover”)

Dada una matriz de ceros y unos, encontrar un subconjunto de filas de forma que la matriz con esas filas tenga un solo uno en cada columna.

Ejemplo (una solución sería elegir las filas de color rojo) :

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

El problema de encontrar filas de la matriz es equivalente al de seleccionar conjuntos que cubran un conjunto:

- cada columna equivale a un elemento del universo,
- cada fila corresponde a un subconjunto.

5. Problema del cubrimiento exacto

Algoritmo X

```
1 algoritmoX() { // de Donald Knuth
2   si (numero de elementos == 0) {
3     procesar_solucion();
4   } si no {
5     si (numero de conjuntos > 0) {
6       elem = el elemento que aparezca en menos conjuntos;
7       para cjt conjunto donde aparezca elem { // ramificar
8         incluir cjt en la solucion;
9         para elem2 perteneciente a cjt {
10          quitar elem2;
11          quitar conjuntos que contengan elem2;
12        }
13        algoritmoX(); // llamada sobre problema reducido
14        meter todos elementos y conjuntos que hemos eliminado;
15        quitar cjt de la solucion;
16      }
17    }
18  }
19 }
```

5. Problema del cubrimiento exacto

Algoritmo X

- este algoritmo se describe normalmente en términos de una matriz de 0s y 1s,
- debe acceder a las posiciones a uno de la matriz,
- es importante poder eliminar filas y columnas, pero también poder deshacer este cambio volviéndolas a insertar,
- en la práctica las matrices sobre las que se aplica este algoritmo son *dispersas*.

Por estos motivos, una implementación eficiente propuesta por Donald Knuth representa la matriz de forma dispersa mediante listas doblemente enlazadas y además con una técnica (dancing links) que permite deshacer los cambios de forma muy eficiente.

No obstante, vamos a estudiar un código bastante **ineficiente** pero más fácil de seguir, usando una matriz no dispersa y dos vectores para indicar qué conjuntos y qué elementos siguen formando parte de la matriz:

5. Problema del cubrimiento exacto

```

29 class exact_cover { // es ineficiente...
30     int numCjt, numElem; // talla matriz del problema original
31     bool *mat; // la matriz, talla numCjt x numElem
32     bool &matriz(int i, int j) { return mat[i*numElem+j]; }
33     int *vCjt; // vectores para indicar que elementos y que
34     int *vElem; // conjuntos siguen formando parte del problema
35     // -1 indica libre, 0,1,...indica eliminado en nivel k de recursion
36     bool *vSolucion; // vector de talla numCjt, marca cjt seleccionados
37     int quedanCjt; // deben coincidir con el num valores a true en vCjt
38     int quedanElem; // y en vElem respectivamente
39     int numSolutions; // contador de soluciones
40
41     void procesarSolucion(); // imprime una solucion
42     int elegirElemento(); // heuristico, elige el elemento que
43         // pertenece a menos cjt
44     void mostrar(); // para hacer una traza
45 public:
46     exact_cover(int ncjt, int nelem);
47     ~exact_cover();
48     void poner_elemento(int cjt, int elem);
49     void backtracking(int nivel); // recursivo
50 };

```

5. Problema del cubrimiento exacto

```
135 int main() {
136     // leemos una matriz de talla F x C con el formato:
137     // F C
138     // FxC valores 0 o 1
139     int F,C; // cada fila es un cjt, C el numero de elementos totales
140     cin >> F >> C;
141     exact_cover EXACT_COVER(F,C);
142     for (int f=0; f<F; ++f)
143         for (int c=0; c<C; ++c) {
144             int x; cin >> x;
145             if (x == 1) EXACT_COVER.poner_elemento(f,c);
146         }
147     EXACT_COVER.backtracking(0);
148 }
```

5. Problema del cubrimiento exacto

```
52 exact_cover::exact_cover(int ncjt, int nelem) {
53     numSolutions=0;
54     numCjt      = quedanCjt  = ncjt;
55     numElem     = quedanElem = nelem;
56     mat         = new bool[numCjt*numElem];
57     vSolucion= new bool[numCjt];
58     vCjt       = new int[numCjt];
59     vElem      = new int[numElem];
60     for (int i=0;i<numElem;++i) vElem[i] = -1;
61     for (int i=0;i<numCjt;++i) { vCjt[i] = -1; vSolucion[i] = false; }
62     for (int i=0;i<numElem*numCjt;++i) mat[i] = false;
63 }
64 exact_cover::~~exact_cover() {
65     delete[] mat;          delete[] vCjt;
66     delete[] vSolucion; delete[] vElem;
67 }
```

5. Problema del cubrimiento exacto

```
96 // buscar elem que aparezca en menos conjuntos
97 int exact_cover::elegirElemento() {
98     int minveces=numCjt+1, elegido=-1;
99     for (int elem=0; elem<numElem; ++elem) // para cada elemento
100         if (vElem[elem]<0) { // si todavia forma parte del problema
101             // cuento en cuantos elementos aparece:
102             int veces = 0;
103             for (int cjt=0; cjt<numCjt; ++cjt)
104                 if (vCjt[cjt]<0 && matriz(cjt,elem)) veces++;
105             if (veces < minveces) {
106                 minveces = veces;
107                 elegido = elem;
108             }
109         }
110     return elegido;
111 }
```

5. Problema del cubrimiento exacto

```

112 void exact_cover::backtracking(int nivel) { // nivel de recursion
113     if (quedanElem == 0) { procesarSolucion(); return; }
114     int elemElegido = elegirElemento();
115     for (int cjtRamificar = 0; cjtRamificar < numCjt; ++cjtRamificar)
116         if (vCjt[cjtRamificar]<0 && matriz(cjtRamificar,elemElegido)) {
117             vSolucion[cjtRamificar] = true; // incluirlo en la solucion
118             for (int elem=0; elem<numElem; ++elem) // quitar elementos de
119                 if (vElem[elem]<0 && matriz(cjtRamificar,elem)) {
120                     vElem[elem] = nivel; quedanElem--; // cjtRamificar
121                     for (int cjt=0; cjt<numCjt; ++cjt) // y los cjt q los
122                         if (vCjt[cjt]<0 && matriz(cjt,elem)) { // contengan
123                             vCjt[cjt] = nivel; quedanCjt--;
124                         }
125                 }
126             backtracking(nivel+1); // llamada recursiva
127             for (int elem=0; elem<numElem; ++elem) // deshacer cambios:
128                 if (vElem[elem]==nivel) { vElem[elem]=-1; quedanElem++; }
129             for (int cjt=0; cjt<numCjt; ++cjt)
130                 if (vCjt[cjt]==nivel) { vCjt[cjt]=-1; quedanCjt++; }
131             vSolucion[cjtRamificar] = false;
132         }
133 }

```

5. Reducción de Sudoku a cubrimiento exacto

1. Generamos una matriz que representa el conjunto de restricciones del juego:
 - Cada fila representa un posible movimiento y se representa $DdRrCc$ siendo d, r, c valores de 1 a 9. Por ejemplo, $D2R5C7$ indica que el dígito 2 se ha puesto en la posición 5, 7 (fila 5 columna 7). Por tanto hay $9^3 = 729$ filas en la matriz.
 - Cada columna representa una restricción (ahora lo veremos).
2. Utilizamos los datos de partida (los dígitos ya conocidos) para eliminar de la matriz las filas asociadas a los datos de partida. Por ejemplo:
 - Si el dígito 2 aparece en la fila 5 y columna 7, entonces eliminamos la fila $D2R5C7$ de la matriz.
 - Debemos eliminar también las columnas que estén a 1 en esa fila.

Atención: Si las filas y columnas eliminadas no respetan el cubrimiento exacto (más de un 1 en una columna) el juego no tiene solución.
3. Resolvemos el problema del cubrimiento exacto sobre la matriz reducida. Las filas seleccionadas nos indican los dígitos a rellenar en los huecos.

5. Reducción de Sudoku a cubrimiento exacto

Columnas de la matriz

- Cada casilla del sudoku tiene un único dígito:

Ejemplo: D^*R5C7 indica que la celda fila 5 y columna 7 tiene un sólo dígito. Las filas donde el $*$ se reemplaza por 1, 2, ..., 9 tienen un 1 en esa columna.

- Un dígito no se repite en una misma fila.

Ejemplo: $D3R2C^*$ indica que el dígito 3 debe aparecer una sola vez en la fila 2. Las filas $D3R2C1, D3R2C2, \dots, D3R2C9$ tienen un 1 en esa columna.

- Un dígito no se repite en una misma columna.

Ejemplo: $D2R^*C5$ indica que el dígito 2 debe aparecer una sola vez en la columna 5. Las filas $D2R^*C5$ con $*$ igual a 1, 2, ..., 9 tienen un 1 en esa columna.

- Un dígito no se repite en una misma región.

Ejemplo: $D7boxRxCy$ donde el 7 se refiere al dígito y los valores x e y pueden tomar 3 valores cada uno (a para 1,2,3, luego b para 4,5,6 y c para 7,8,9) codificando así las 9 regiones.

5. Reducción de Sudoku a cubrimiento exacto

- Cada una de las 4 restricciones corresponde a $9 \times 9 = 81$ columnas. Por tanto, se trata de una matriz 729×324 .
- Se trata de una matriz muy *dispersa*:
 - Cada una de las 729 filas tiene 4 unos y 725 ceros.
Ejemplo: la fila D2R5C7 tiene posiciones no nulas en las columnas: D*R5C7, D2R*C7, D2R5C* y D2boxRbCc
 - Cada columna tiene exactamente 9 unos y 720 ceros.
- El orden de las filas o de las columnas es irrelevante.

6. Reducción de las n -reinas al cubrimiento exacto

Generamos una matriz de ceros y unos. Para simplificar, asociamos un nombre a las filas y columnas:

- Cada fila corresponde a una posición en el tablero.

Ejemplo: La fila R3C2 corresponde a la casilla (3, 2) del tablero.

- Cada columna representa una restricción del problema:

- No podemos situar más de una reina en una misma fila:

Ejemplo: La fila R3C2 tiene un uno en la columna de nombre R3.

- No podemos situar más de una reina en una misma columna:

Ejemplo: La fila R3C2 tiene un uno en la columna de nombre C3.

- No podemos situar más de una reina en una misma diagonal:

Ejemplo: La fila R3C2 tiene un uno en las columnas de nombres X5 e Y1. En general, la fila (i, j) ocupa las diagonales $i + j$ e $i - j$.

- Se trata de una matrix con n^2 filas y $n + n + 2(n - 1) = 4n - 2$ columnas. Cada fila tiene sólo 4 unos y el resto a cero. Se trata de una matriz *dispersa*.
- El orden de las filas y columnas es irrelevante.

6. Reducción de las n -reinas al cubrimiento exacto

- Si buscamos un cubrimiento exacto de esta matriz, el algoritmo intentará buscar un subconjunto de filas de la matriz (de casillas del tablero) tal que:
 - Sitúe exactamente una reina en cada fila del tablero.
 - Sitúe exactamente una reina en cada columna del tablero.
 - Sitúe exactamente una reina en cada diagonal del tablero.
- **Atención:** ¿una reina en cada diagonal del tablero? Si hay n reinas y $4n - 2$ diagonales en general es *imposible*, algo falla. La última condición debe cambiarse por esta otra:
 - Sitúe *como mucho* una reina en cada diagonal del tablero.
- **Solución:** Utilizar una versión ampliada del problema del cubrimiento exacto.

6. Problema del cubrimiento exacto con columnas primarias y secundarias

Dada una matriz de ceros y unos y una clasificación de sus columnas en **primarias** y **secundarias**, se pide encontrar un subconjunto de filas de forma que la matriz formada por esas filas tenga:

- Un sólo uno en las columnas primarias.
- Como mucho un uno en las columnas secundarias.

Ejemplo (izquierda columnas primarias, derecha columnas secundarias, una solución sería elegir las filas de color rojo) :

$$\left(\begin{array}{cccc|ccc} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{array} \right)$$

6. Problema del cubrimiento exacto con columnas primarias y secundarias

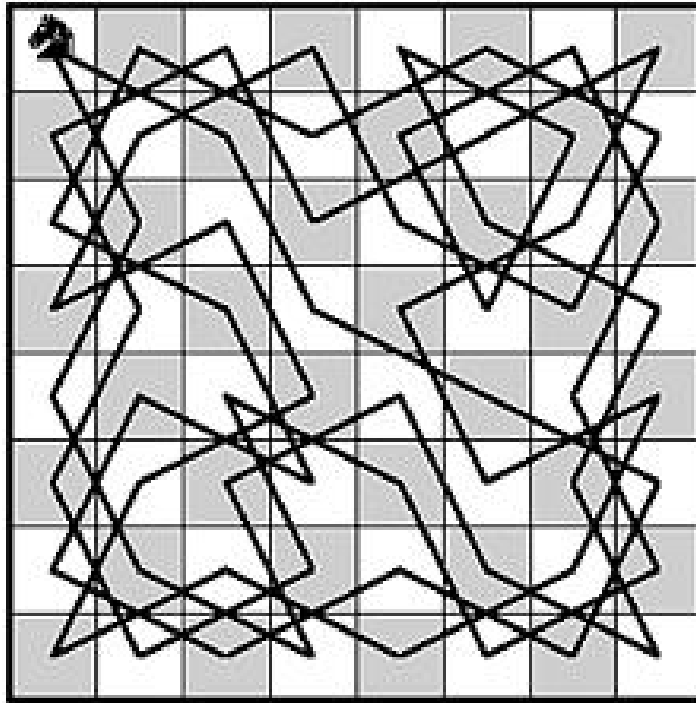
- Es inmediato reducir el problema del cubrimiento exacto con matrices primarias y secundarias al cubrimiento exacto normal (sólo matrices primarias):

Por cada columna secundaria incluimos una nueva fila auxiliar que tenga un uno en la columna secundaria en cuestión y el resto de columnas a cero.

- No obstante, resulta más eficiente implementar una versión adaptada del “algoritmo X” para tratar con columnas primarias y secundarias. Este algoritmo no daría todas las soluciones, sino sólo las que necesite tomar las columnas secundarias imprescindibles. Por ejemplo, si nos dan una matriz sin columnas primarias devolvería el conjunto vacío de filas.

7. El salto del caballo (Knight's tour)

¿Es posible mover un caballo en un tablero de ajedrez de forma que pase una sola vez por cada casilla?



7. El salto del caballo (Knight's tour)

Existen diversas variantes. Por ejemplo, se puede exigir además que el caballo se sitúe en una posición desde la cual pueda volver a su posición inicial.

Este problema está íntimamente relacionado con un problema más general, el *ciclo Hamiltoniano* en un grafo: Encontrar un ciclo en un grafo no dirigido que visite cada vértice una sola vez.

El problema del ciclo Hamiltoniano es un problema de decisión, no obstante puede reducirse a otro problema de optimización muy conocido: “El viajante de comercio” :

Dado un conjunto de ciudades y los costes de viajar de una ciudad a otra, ¿cuál es la ruta menos costosa para visitar todas las ciudades una sola vez y terminar en la ciudad de origen?

7. El salto del caballo (Knight's tour)

Si bien el problema general es NP-completo, existen soluciones muy eficientes para encontrar alguna solución al Knight's tour utilizando la técnica "divide y vencerás".

No obstante, vamos a estudiar una solución basada en búsqueda con retroceso.

El interés de este problema es mostrar que un *heurístico* en el orden de ramificación de los distintos estados hijo puede acelerar enormemente la búsqueda. El heurístico que vamos a probar fue descubierto por H. C. von Warnsdorf en 1823 y dice:

Ir a la posición desde la que existan menos movimientos válidos posibles.

7. El salto del caballo (Knight's tour)

Otra técnica que es útil conocer es que conviene relajar las condiciones de un problema para ver si un estado es prometedor. En el caso de un ciclo con movimientos del caballo, dada una solución parcial podemos ver si se puede completar el ciclo de esta forma:

*La solución parcial es prometedora si es posible volver desde la última posición del caballo hasta la posición inicial mediante un **camino** de movimientos válidos por casillas libres.*

Fíjate que al hablar de camino no exigimos que se pase por **todos** los vértices libres, con lo que relajamos el problema original. Buscar un camino es muy eficiente usando, por ejemplo, un recorrido primero en profundidad del grafo.

7. El salto del caballo (Knight's tour)

```
1 // ciclo Hamiltoniano con un caballo mediante búsqueda
2 // con retroceso y el heurístico de Warnsdorf:
3 using namespace std;
4 #include <iostream>
5 #include <iomanip>
6
7 struct posicion { // representa una posicion en el tablero
8     int x,y;
9     posicion() {};
10    posicion(int a, int b) { x=a; y=b; }
11 };
12 class ktour { // ciclo Hamiltoniano con un Caballo
13     // los 8 movimientos de un caballo:
14     static const int nmovimientos = 8;
15     static const int dx[nmovimientos];
16     static const int dy[nmovimientos];
17     static const int ntablero = 8;
18     static const int longciclo = ntablero*ntablero;
19     posicion solucion[longciclo];
20     // para marcar casillas libres
21     bool libre[ntablero][ntablero];
```

7. El salto del caballo (Knight's tour)

```
22  int longitud,soluciones,explorados;
23  bool valida(posicion);
24  posicion mover(posicion,int);
25  bool es_terminal();
26  bool es_factible(posicion p);
27  void imprimir_solucion();
28  // ordenamos para heuristico:
29  void insertionSort(posicion p[], int r[], int n);
30  // utilizamos algoritmo dfs para ver si es prometedor:
31  int visitadfs[ntablero][ntablero],stampdfs;
32  bool prometedor_rekurs(posicion d);
33  bool prometedor(posicion d);
34  public:
35  ktour();
36  void buscar();
37  };
38  // los desplazamientos relativos para el movimiento de un Caballo:
39  const int ktour::dx[] = {-2, -2, -1, 1, 2, 2, 1, -1};
40  const int ktour::dy[] = {-1, 1, 2, 2, 1, -1, -2, -2};
41  ktour::ktour() {
42  for (int i=0;i<ntablero;i++)
43  for (int j=0;j<ntablero;j++) {
```

7. El salto del caballo (Knight's tour)

```
44     libre[i][j] = true;
45     visitadfs[i][j] = 0; // para algoritmo dfs
46 }
47 stampdfs = 0; // para algoritmo dfs
48 soluciones = 0;
49 explorados = 0;
50 // empezamos en casilla (0,0)
51 solucion[0].x = solucion[0].y = 0;
52 libre[0][0] = false;
53 longitud = 1;
54 }
55 inline bool ktour::valida(posicion p) {
56     return 0<=p.x && p.x<ntablero &&
57           0<=p.y && p.y<ntablero && libre[p.x][p.y];
58 }
59 inline posicion ktour::mover(posicion p,int i) {
60     return posicion(p.x+dx[i],p.y+dy[i]);
61 }
62 inline bool ktour::es_terminal() {
63     return longitud == longciclo;
64 }
```

7. El salto del caballo (Knight's tour)

```
65 inline bool ktour::es_factible(posicion p) { // para ciclo Hamiltoniano
66     // las casillas (1,2) y (2,1) son las únicas desde las que podemos
67     // volver a la posición inicial (0,0)
68     return ((p.x == 1 && p.y == 2) || (p.x == 2 && p.y == 1));
69 }
70 bool ktour::prometedor_rekurs(posicion d) { // busca camino a (1,2)
71     if (es_factible(d)) return true;      // o a (2,1)
72     for (int i=0;i<nmovimientos;i++) {
73         posicion a = mover(d,i);
74         if (valida(a) && visitadfs[a.x][a.y] != stampdfs) {
75             visitadfs[a.x][a.y] = stampdfs;
76             if (prometedor_rekurs(a)) return true;
77         }
78     }
79     return false;
80 }
81 bool ktour::prometedor(posicion d) {
82     stampdfs++;
83     return prometedor_rekurs(d);
84 }
```

7. El salto del caballo (Knight's tour)

```
114 void ktour::buscar() {
115     explorados++;
116     if (es_terminal()) {
117         if (es_factible(solucion[longciclo-1]))
118             imprimir_solucion();
119     } else { // ramificar
120         int nhijos = 0;
121         posicion p[nmovimientos];
122         int r[nmovimientos];
123         for (int i=0;i<nmovimientos;i++) {
124             posicion a = mover(solucion[longitud-1], i);
125             if (valida(a)) {
126                 int grado = 0;
127                 for (int j=0;j<nmovimientos;j++)
128                     if (valida(mover(a, j)))
129                         grado++;
130                 p[nhijos] = a;
131                 r[nhijos] = grado;
132                 nhijos++;
133             }
134         }
```

7. El salto del caballo (Knight's tour)

```
135 // ordenamos los movimientos de menor a mayor grado:
136 insertionSort(p,r,nhijos);
137 // llamada recursiva:
138 for (int i=0; i<nhijos; i++) {
139     // marcar
140     solucion[longitud] = p[i]; longitud++;
141     libre[p[i].x][p[i].y] = false;
142     // llamada recursiva
143     if (prometedor(p[i]))
144         buscar();
145     // desmarcar
146     libre[p[i].x][p[i].y] = true;
147     longitud--;
148 }
149 }
150 }
151 int main() {
152     ktour x;
153     x.buscar();
154     return 0;
155 }
```