Natalia Esteban Sánchez
J. Ángel Velázquez Iturbide

Revisión Bibliográfica de Problemas Resolubles por la Técnica de Vuelta Atrás

Número 2012-04

Serie de Informes Técnicos DLSI1-URJC
ISSN 1988-8074
Departamento de Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos

Índice

1 Problemas de Decisión 3
1.1 Problema de las N-Reinas 3
1.1.1 Tabla resumen sobre el problema de las n-reinas 13
1.2 Problemas sobre grafos 14
1.2.1 Tabla resumen sobre problemas de grafos 31
1.3 Problemas de caracteres 33
1.3.1 Tabla resumen sobre problemas de caracteres 39
1.4 Problema del salto de caballo 40
1.4.1 Tabla resumen sobre el problema del salto de caballo 44
1.5 Problema del laberinto 45
1.6 Problemas de juegos 46
1.6.5 Tabla resumen sobre problemas de juegos 56
1.7 Problemas de subconjuntos 58
1.7.1 Tabla resumen sobre problemas de subconjuntos 65
1.8 Otros problemas de decisión 66
2 Problemas de optimización 72
2.1 Problema de la Mochila 0/1 72
2.1.1 Tabla resumen sobre el problema de la mochila 0/1 83
2.2 Problemas de asignación 84
2.3 Otros problemas de optimización 86
3 Tabla Resumen 95
4 Conclusiones 108
Referencias 108

Índice de ilustraciones

Ilustración 1, tablero ajedrez 4-reinas 4
Ilustración 2, árbol de búsqueda generado y tablero solución (4-reinas) 4
Ilustración 3, tablero ajedrez 8-reinas 6
Ilustración 4, árbol de búsqueda potencial (4-reinas) 6
Ilustración 5, secuencia de tableros generados (4-reinas) 7
Ilustración 6, árbol de búsqueda generado (4-reinas) 7
Ilustración 7, tablero de ajedrez (numeración de diagonales) 8
Ilustración 8, tablero de ajedrez 8-reinas 9
Ilustración 9, tablero de ajedrez (4-reinas) 10
Ilustración 10, árbol de búsqueda potencial (4-reinas) 10
Ilustración 11, árbol de búsqueda generado (4-reinas) 11
Ilustración 12, secuencia de tableros generados (4-reinas) 11
Ilustración 13, árbol de búsqueda potencial (3-coloring) 14
Ilustración 14, figura compuesta (grafo y árbol generado 3-coloring) 15
Ilustración 15, árbol de búsqueda potencial (3-coloring) 16
Ilustración 16, figura compuesta (grafo y árbol potencial, 3-coloring) 16
Ilustración 17, grafo 4 nodos 17
Ilustración 18, árbol de búsqueda generado abreviado (3-coloring) 18
Ilustración 19, grafo planar 5 nodos 18
Ilustración 20, dos grafos 21
Ilustración 21, árbol de búsqueda potencial abreviado (ciclos hamiltonianos) 22
Ilustración 22, dos grafos 24
Ilustración 23, grafo de 4 nodos 25
Ilustración 24, árbol de búsqueda potencial (ciclos hamiltonianos) 25
Ilustración 25, figura compuesta (grafo y árbol de búsqueda generado) 28
Ilustración 26, dos grafos isomorfos 29
Ilustración 27, árbol potencial abreviado (variaciones) 35
Ilustración 28, tablero movimientos caballo de ajedrez 40
Ilustración 29, tablero parcial con movimientos caballo válidos 42
Ilustración 30, movimientos válidos laberinto 45
Ilustración 31, tablero solución cuadrado mágico 47
Ilustración 32, tablero solución cuadrado latino 49
Ilustración 33, numeración de celdas del cuadrado latino 49
Ilustración 34, ejemplo y solución de sudoku 51
Ilustración 35, ejemplo de partida del dominó 53
Ilustración 36, árbol de búsqueda potencial abreviado (dominó) 55
Ilustración 37, árbol de búsqueda potencial (subconjuntos) 58
Ilustración 38, árbol de búsqueda potencial (subconjuntos) 59
Ilustración 39, árbol de búsqueda generado (subconjuntos) 59
Ilustración 40, árbol de búsqueda potencial (subconjuntos) 61
Ilustración 41, árbol de búsqueda potencial (subconjuntos) 61
Ilustración 42, árbol de búsqueda generado (subconjuntos) 62
Ilustración 43, árbol de búsqueda potencial, (factorías) 66

Ilustración 44, árbol de búsqueda potencial (sellos)..... 68
Ilustración 45, árbol de búsqueda potencial (sellos)..... 68
Ilustración 46, árbol de búsqueda potencial (reparto) 71
Ilustración 47, árbol de búsqueda generado (mochila)..... 73
Ilustración 48, árbol de búsqueda generado (mochila)..... 74
Ilustración 49, árbol de búsqueda generado (mochila)..... 74
Ilustración 50, árbol de búsqueda potencial (canciones)..... 78
Ilustración 51, árbol de búsqueda generado (mochila)..... 81
Ilustración 52, árbol de búsqueda potencial (mochila)..... 82
Ilustración 53, árbol de búsqueda potencial (comensales) 89
Ilustración 54, árbol de búsqueda potencial (envases). 93

Revisión Bibliográfica de Problemas Resolubles por la Técnica de Vuelta Atrás

Natalia Esteban Sánchez, J. Ángel Velázquez Iturbide

Departamento de Lenguajes y Sistemas Informáticos I, Universidad Rey Juan Carlos,
C/ Tulipán s/n, 28933, Móstoles, Madrid
{natalia.esteban,angel.velazquez}@urjc.es

Abstract. El siguiente trabajo recopila y recoge desde varios libros de texto los tipos de problemas que se pueden solucionar mediante algoritmos de vuelta atrás, llegando a una revisión bibliográfica que organiza y agrupa los diferentes tipos de problemas.

Keywords: backtracking, vuelta atrás, búsqueda en espacio de estados

1 Introducción

El presente trabajo recoge y recopila numerosos problemas resolubles mediante la técnica de vuelta atrás o backtracking.

Esta técnica se utiliza para resolver problemas en los que la solución es compuesta y donde cada solución es el resultado de una secuencia de decisiones. Los problemas a resolver por vuelta atrás son los siguientes:

- Problemas de decisión: Buscan la solución o soluciones que satisfacen ciertas restricciones.
- Problemas de optimización: Buscan la solución óptima en base a una función objetivo.

Para realizar la recopilación se han seleccionado 14 libros de texto prestigiosos sobre algoritmos [1-14]. No se define aquí la técnica de vuelta atrás porque puede encontrase en casi todos los libros citados.

Para llevar a cabo la búsqueda, en cada libro, en primer lugar se ha buscado si en el índice había algún capítulo sobre vuelta atrás o algoritmos de búsqueda en espacios de estados. En caso afirmativo, se ha realizado la búsqueda en el mismo; si no, se ha buscado en el glosario de términos el nombre de problemas conocidos como por ejemplo, problema de las n-reinas, del salto de caballo, del laberinto, etc.

El objetivo del trabajo es múltiple:

- Disponer de una recopilación de problemas utilizados en los libros de texto para ilustrar la técnica de diseño de algoritmos. Este objetivo es de interés para profesores o investigadores de algoritmos.

- Explorar la posibilidad de generalizar la forma de organizar los problemas resolubles por esta técnica de diseño. Este objetivo es de interés para la línea de investigación que tenemos abierta sobre el diseño de ayudantes interactivos para el aprendizaje de algoritmos de vuelta atrás.

Los problemas se han organizado en los siguientes grupos:

1. Problemas de decisión
    1.1. Problema de las 8-reinas
    1.2. Problemas de grafos
    1.3. Problemas de cadenas de caracteres
    1.4. Problema del salto de caballo
    1.5. Problema del laberinto
    1.6. Problemas de juegos
        1.6.1. Problema del cuadrado mágico
        1.6.2. Problema del latino
        1.6.3. Problema del dominó
        1.6.4. Problema del sudoku
    1.7. Problemas de subconjuntos
    1.8. Otros problemas de decisión
        1.8.1. Problema de las factorías
        1.8.2. Problema de franquear postales
        1.8.3. Problema del reparto
2. Problemas de optimización
    2.1. Problema de la mochila 0/1
    2.2. Problemas de asignación
        2.2.1. Asignación de tareas
    2.3. Otros problemas de optimización
        2.3.1. Problema de recolección
        2.3.2. Problema de los comensales
        2.3.3. Problema de cableado de longitud mínima
        2.3.4. Problema de franquear postales
        2.3.5. Problema de minimizar el número de envases

Para mantener la originalidad del problema que venía en cada libro, se han recogido los problemas de forma literal, es decir, se ha mantenido el idioma (ingles o español), y el nombre de los problemas (unos con nombre significativo y otros con solamente un número).

Para cada problema se incluirá el enunciado general del mismo, en algunos casos puede ser extraído de algunos de los libros revisados, la nomenclatura y enunciado encontrado en cada uno de los libros, así como aquellas figuras relevantes y el código o seudocódigo utilizado por los autores en cada problema.

Así, con el objeto de proporcionar el máximo de información resumida sobre cada problema, se han confeccionado unas tablas de resumen para el lector. Su formato es, en general, el siguiente:

Libro	Capítulo/apartado	Visualización	Implementación

Al finalizar la recopilación de cada tipo de problema, se presentará una tabla de resumen para el lector que incluirá todos los libros donde se puede consultar el problema en cuestión y la información que se puede encontrar en cada uno de los libros.

1 Problemas de Decisión

En este apartado se verá un gran número de problemas distintos donde la solución o soluciones satisfacen una serie de restricciones.

1.1 Problema de las N-Reinas

En primer lugar se da una descripción del problema.

Problema **1**. Se dispone de un tablero de ajedrez de tamaño $n \times n$, y se trata de colocar en él n reinas de manera que no se amenacen según las normas del ajedrez, es decir, que no se encuentren en la misma fila, ni en la misma columna ni en la misma diagonal.

A continuación se muestra cómo se formula el problema en los distintos libros revisados. En cada ejemplo se mostrará una tabla resumen, que indica qué información se puede encontrar en el libro correspondiente, el enunciado del problema, las imágenes incluidas en cada problema y la implementación del mismo.

Libro	Capítulo/apartado	Visualización	Implementación
M. H. Alsuwaiyel, *Algorithm Design Techniques and Analysis*	Capítulo 13	Fig. p.358 *Tablero* Fig. p.360 *Árbol de búsqueda*	Pseudocódigo p. 359 *Una solución Iterativo*

13.3 The 8-Queens Problem

The classical 8-QUEENS can be stated as follows. How can we arrange 8 queens on an 8 x 8 chessboard so that no two queens can attack each other?

Two queens can attack each other if they are in the same row, column or diagonal. The n-queens problem is defined similarly, where in this case we have n queens and an $n \times n$ chessboard for an arbitrary value of $n \geq 1$.

To simplify the discussion, we will study the 4-queens problem, and the generalization to any arbitrary n is straightforward.

Este libro incluye las siguientes figuras:

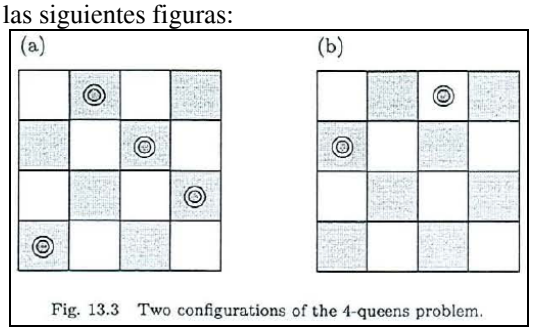


Fig. 13.3 Two configurations of the 4-queens problem.

Ilustración 1, tablero ajedrez 4-reinas

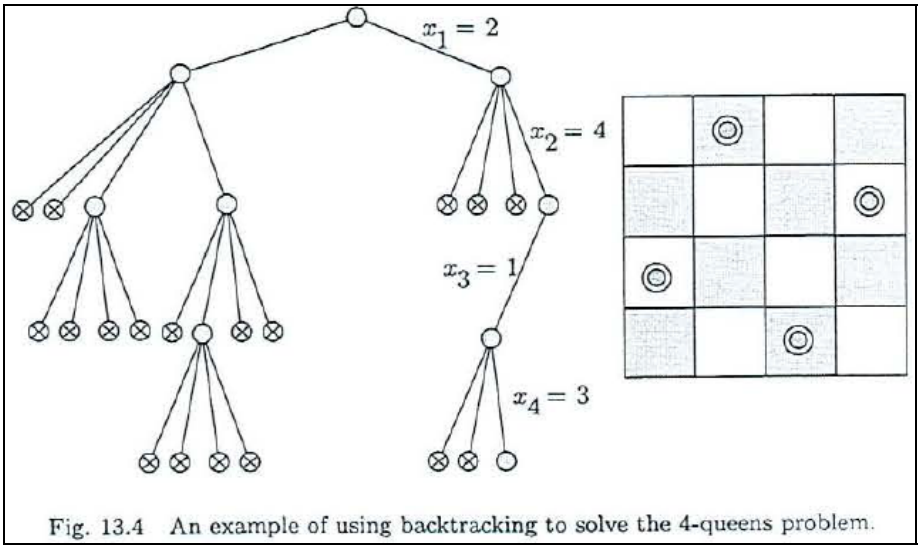


Fig. 13.4 An example of using backtracking to solve the 4-queens problem.

Ilustración 2, árbol de búsqueda generado y tablero solución (4-reinas)

El pseudocódigo que resuelve el problema es el siguiente:

Algorithm 13.3 4-QUEENS
Input: none.
Output: A vector $x[1..4]$ corresponding to the solution of the 4-queens problem.

```
1.  for k ← 1 to 4
2.      x[k] ← 0      {no queens are placed on the chessboard}
3.  end for
4.  flag ← false
5.  k ← 1
6.  while k ≥ 1
7.      while x[k] ≤ 3
8.          x[k] ← x[k] + 1
9.          if x is a legal placement then set flag ← true
            and exit from the two while loops.
10.         else if x is partial then k ← k + 1      {advance}
11.     end while
12.     x[k] ← 0
13.     k ← k − 1      {backtrack}
14. end while
15. if flag then output x
16. else output "no solution"
```

Libro	Capítulo/apartado	Visualización	Implementación
G. Brassard y P. Bratley, *Fundamentals of Algorithmics*	Capítulo 9 Apartado 6	-	Pseudocódigo p. 311 *Todas las soluciones Recursivo*

9.6.2 The eight queens problem

For our second example of backtracking, consider the classic problem of placing eight queens on a chessboard in such a way that none of them threatens any of the others. Recall that a queen threatens the squares in the same row, in the same column, or on the same diagonals.

Este problema incluye el siguiente pseudocódigo:

In the following procedure $sol[1..8]$ is a global array. To print all the solutions to the eight queens problem, call $queens(0, \emptyset, \emptyset, \emptyset)$.

```
procedure queens(k, col, diag45, diag135)
    {sol[1..k] is k-promising,
     col = {sol[i] | 1 ≤ i ≤ k},
     diag45 = {sol[i]−i + 1 | 1 ≤ i ≤ k}, and
     diag135 = {sol[i]+i − 1 | 1 ≤ i ≤ k}}
    if k = 8 then {an 8-promising vector is a solution}
                  write sol
    else {explore (k + 1)-promising extensions of sol}
         for j ← 1 to 8 do
             if j ∉ col and j − k ∉ diag45 and j + k ∉ diag135
             then sol[k + 1] ← j
                  {sol[1..k + 1] is (k + 1)-promising}
                  queens(k + 1, col ∪ {j},
                         diag45 ∪ {j − k}, diag135 ∪ {j + k})
```

Libro	Capítulo/ apartado	Visualización	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.325 *Tablero* Fig. p.326 *Árbol de búsqueda* Fig. p.331 *Tablero y árbol de búsqueda*	Implementación p. 338 *Todas las soluciones Iterativo*

Example 7.1 (8-queens) A classic combinatorial problem is to place eight queens on an 8×8 chessboard so that no two "attack", that is so that no two of them are on the same row, column or diagonal. Let us number the rows and columns of the chessboard 1 through 8 (figure 7 .1).

Example 7.3 (n-queens) The *n*-queens problem is a generalization of the 8-queens problem of Example 7.1. *n* queens are to be placed on a *n*×*n* chessboard so that no two attack, i.e., no two queens are on the same row, column or diagonal.

Example 7.5 (4-queens) Let us see how backtracking works on the 4-queens problem of Example 7.3. As a bounding function we will use the obvious criteria that if (x_1, x_2, …, x_i) is the path to the current E-node then all children nodes with parent-child

labelings x_{i+1} are such that $(x_1,\ldots, x_{i+1})$ represents a chessboard configuration in which no two queens are attacking.

Las figuras incluidas en el libro son:

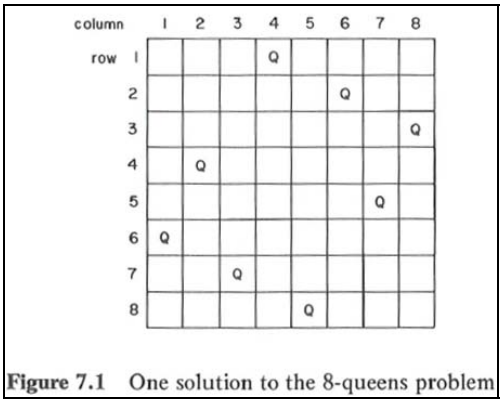


Figure 7.1 One solution to the 8-queens problem

Ilustración 3, tablero ajedrez 8-reinas

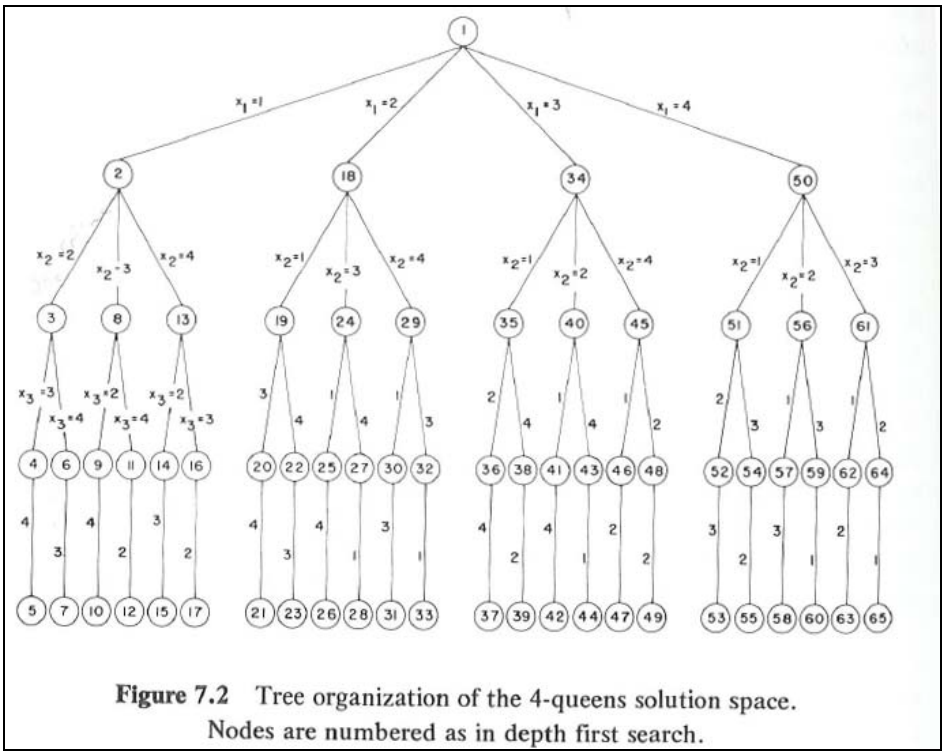


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Ilustración 4, árbol de búsqueda potencial (4-reinas)

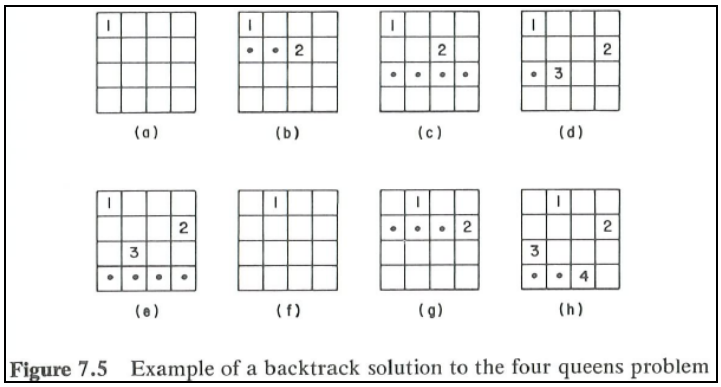


Figure 7.5 Example of a backtrack solution to the four queens problem

Ilustración 5, secuencia de tableros generados (4-reinas)

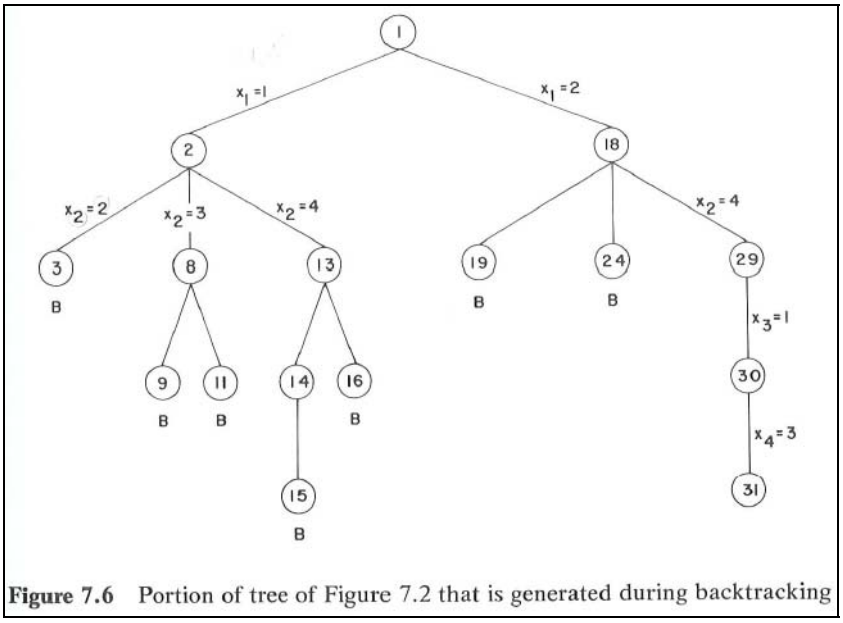


Figure 7.6 Portion of tree of Figure 7.2 that is generated during backtracking

Ilustración 6, árbol de búsqueda generado (4-reinas)

```
procedure NQUEENS(n)
  //using backtracking this procedure prints a
  //n queens on an n × n chessboard so tha
  integer k, n, X(1:n)
  X(1) ← 0; k ← 1   //k is the current row;
  while k > 0 do   //for all rows do//
    X(k) ← X(k) + 1   //move to the next colu
    while X(k) ≤ n and not PLACE(k) do   //c
      X(k) ← X(k) + 1
    repeat
    if X(k) ≤ n   //a position is found//
      then if k = n   //is a solution complete?
           then print(X)   //yes, print the arra
           else k ← k + 1; X(k) ← 0   //go to
           endif
      else k ← k - 1   //backtrack//
    endif
  repeat
end NQUEENS
```

Algorithm 7.5 All solutions to the n-queens problem

```
procedure PLACE(k)
  //returns true if a queen can be placed in
  //X(k)th column. Otherwise it returns fals
  //X is a global array whose first k values h
  //ABS(r) returns the absolute value of r//
  global X(1: k); integer i, k
  for i ← 1 to k do
    if X(i) = X(k)   //two in the same colun
      or ABS(X(i) - X(k)) = ABS(i - k)
        then return(false)
    endif
  repeat
  return(true)
end PLACE
```

Algorithm 7.4 Can a new queen be placed?

Libro	Capítulo/ apartado	Visualización	Implementación
N, Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.466 *Tablero*	Implementación p. 466 *Todas las soluciones Recursivo*

14.8 Diseñar un algoritmo que encuentre todas las formas de colocar n reinas sobre un tablero de ajedrez de tamaño $n \times n$ de tal forma que no haya dos reinas dándose jaque, es decir, que no haya dos reinas en una misma fila, columna o diagonal.

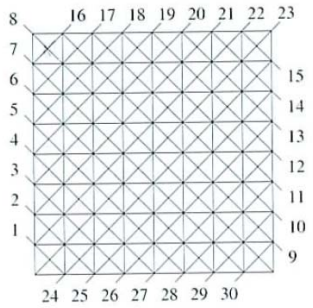


Figura 14.6: Numeración de diagonales de un tablero de tamaño 8 × 8.

Ilustración 7, tablero de ajedrez (numeración de diagonales)

```
proc reinas-va(sol[1..n] de 1..n, e k : 1..n, C[1..n], D[1..4n − 2] de bool)
    para columna = 1 hasta n hacer
        sol[k] := columna
        si ¬C[sol[k]] ∧ ¬D[sol[k] − k + n] ∧ ¬D[k + sol[k] + 2n − 2] entonces
            { marcar }
            C[sol[k]] := cierto
            D[sol[k] − k + n] := cierto ; D[k + sol[k] + 2n − 2] := cierto
            si k = n entonces imprimir(sol)
            si no reinas-va(sol, k + 1, C, D)
            fsi
            { desmarcar }
            C[sol[k]] := falso
            D[sol[k] − k + n] := falso ; D[k + sol[k] + 2n − 2] := falso
        fsi
    fpara
fproc
```

Libro	Capítulo/apartado	Visualización	Implementación
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.179 *Árbol de búsqueda* Fig. p.181 *Tablero* Fig. p.182 *Árbol de búsqueda* Fig. p.183 *Tablero* Fig. p.186 *Tablero*	Implementación p. 186-87 *Todas las soluciones* *Recursivo*

5.1 THE BACKTRACKING TECHNIQUE

The classic example of backtracking is the *n*-Queens Problem. The goal in this problem is to position *n* queens on an *n*x*n* chessboard so that no two queens threaten each other. That is, no two queens may be in the same row, column, or diagonal.

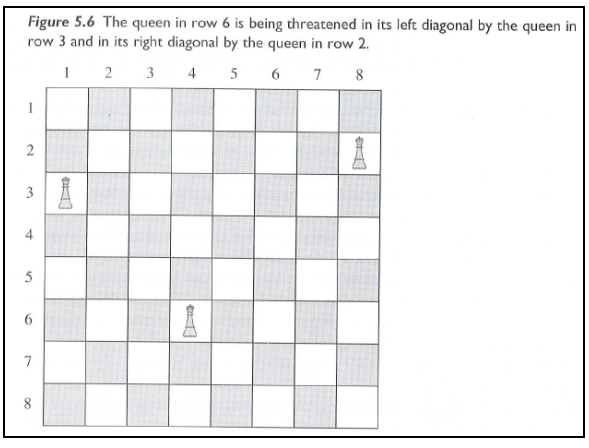


Figure 5.6 The queen in row 6 is being threatened in its left diagonal by the queen in row 3 and in its right diagonal by the queen in row 2.

Ilustración 8, tablero de ajedrez 8-reinas

Example 5.1
5.2 THE *n*-QUEENS PROBLEM

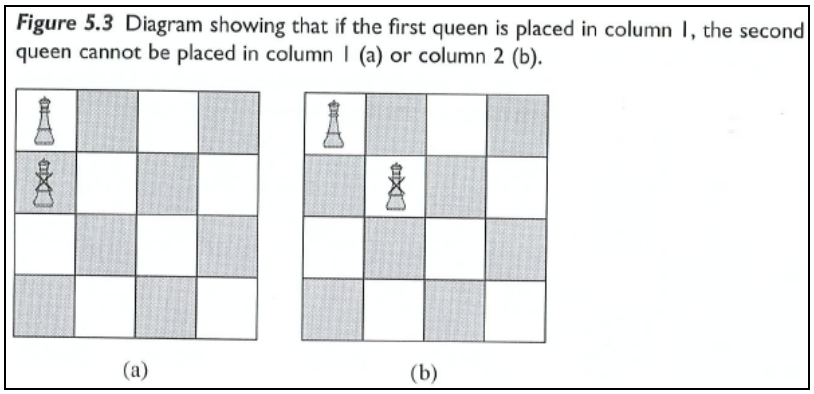


Figure 5.3 Diagram showing that if the first queen is placed in column 1, the second queen cannot be placed in column 1 (a) or column 2 (b).

Ilustración 9, tablero de ajedrez (4-reinas)

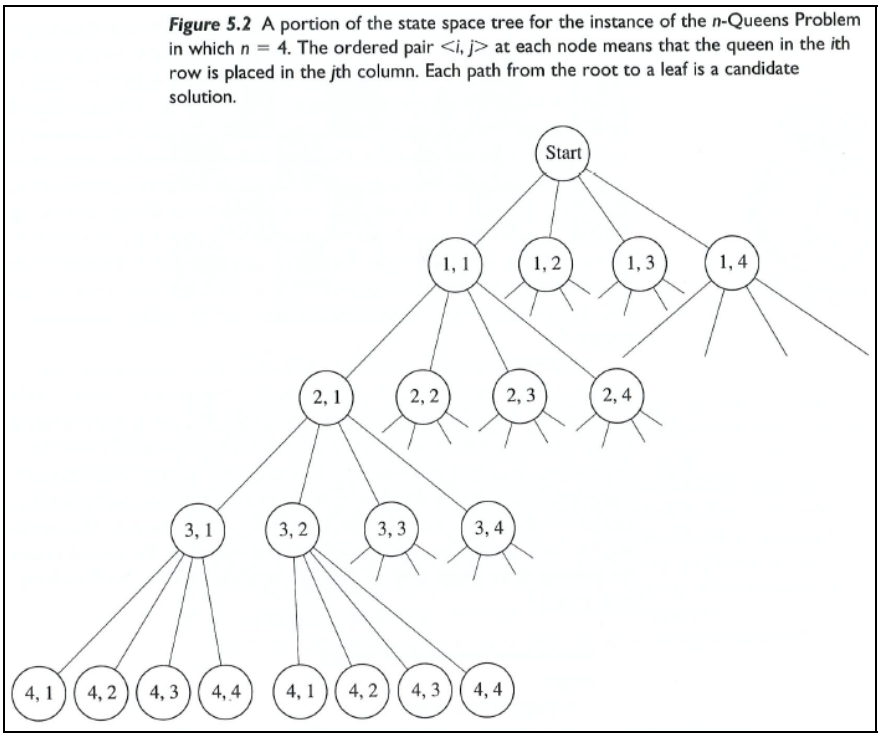


Figure 5.2 A portion of the state space tree for the instance of the *n*-Queens Problem in which $n = 4$. The ordered pair $<i, j>$ at each node means that the queen in the ith row is placed in the jth column. Each path from the root to a leaf is a candidate solution.

Ilustración 10, árbol de búsqueda potencial (4-reinas)

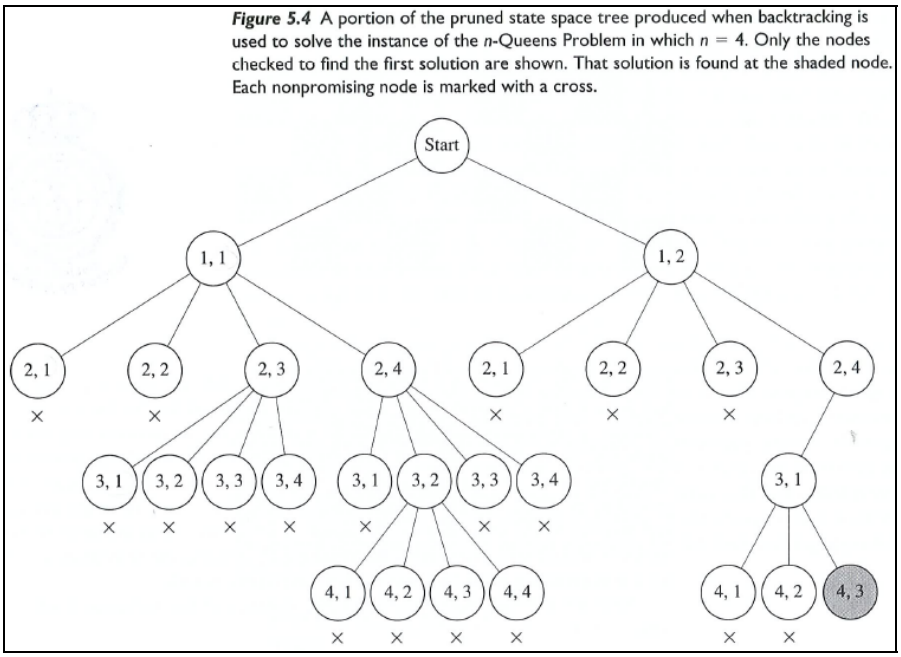


Ilustración 11, árbol de búsqueda generado (4-reinas)

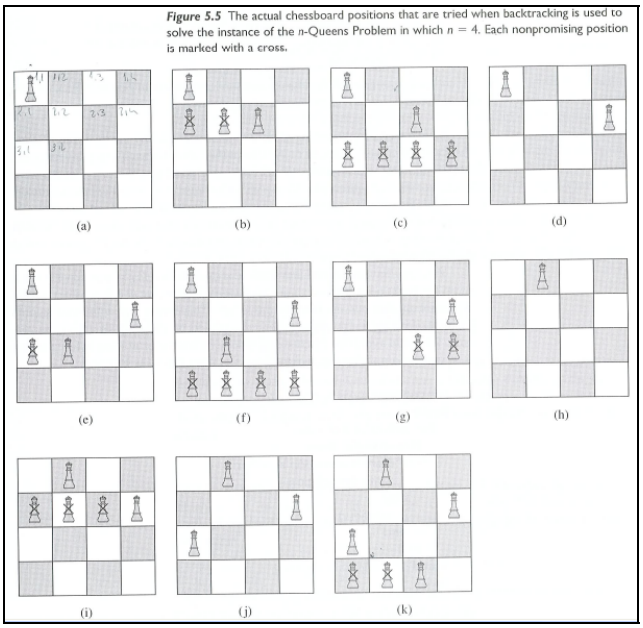


Ilustración 12, secuencia de tableros generados (4-reinas)

```
procedure queens(i: index);
var
  j: index;
begin
  if promising (i) then
    if i=n then
      write(col[1] through col[n])
    else
      for j:=1 to n do
        col[i+1]:= j;
        queens(i+1)
      end
    end
  end;
```

```
function promising(i:index): boolean;
var
  k: index;
begin
  k:= 1;
  promising:= true;
  while k < i and promising do
    if col[i] = col[k] or abs(col[i] - col[k])
      promising:= false
    end;
    k:= k+1
  end
end;
```

1.1.1 Tabla resumen sobre el problema de las n-reinas

A continuación se muestra una tabla resumen de todos los libros donde se puede consultar el problema de las N Reinas, con el objeto de que el lector pueda saber, a simple vista, en qué libros se puede encontrar el problema indicado y qué incluye cada libro.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
M. H. Alsuwaiyel, Algorithms *Design Techniques and Analysis*	Capítulo 13	Fig. p.358 *Tablero* Fig. p.360 *Árbol de búsqueda*	The 8-Queens Problem	*Pseudocódigo p. 359* *Una solución* *Iterativo*
G. Brassard y P. Bratley, *Fundamentals of Algorithmics*	Capítulo 9 Apartado 6	-	The eight queens problem	Pseudocódigo p. 345 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.325 *Tablero* Fig. p.326 *Árbol de búsqueda* Fig. p.331 *Tablero y árbol de búsqueda*	8-queens/n-queens/4-queens	Implementación p. 338 *Todas las soluciones* *Iterativo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.466 *Tablero*	14.8	Implementación p. 466 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.179 *Árbol de búsqueda* Fig. p.181 *Tablero* Fig. p.182 *Árbol de búsqueda* Fig. p.183 *Tablero* Fig. p.186 *Tablero*	THE *n*-QUEENS PROBLEM	Implementación p. 186-87 *Todas las soluciones* *Recursivo*

1.2 Problemas sobre grafos

En este apartado se verán un conjunto de problemas típicos sobre grafos.

1.2.1 Coloreado de Grafos

Este problema también se conoce como coloreado de mapas. A continuación se especifica la descripción general del problema.

Problema 2. Dado un grafo conexo y un número m>0, llamamos colorear el grafo a asignar un número i ($1 \leq i \leq m$) a cada vértice, de forma que dos vértices adyacentes nunca tengan asignados números iguales.

Se pasa a especificar los libros donde se puede consultar el problema indicado.

Libro	Capítulo/apartado	Visualización	Implementación
M. H. Alsuwaiyel, *Algorithm Design Techniques and Analysis*	Capítulo 13	Fig. p.354 *Árbol de búsqueda* Fig. p.355 *Grafo y árbol de búsqueda*	Pseudocódigo p. 356-7 *Recursivo e iterativo*

13.2 The 3-Coloring Problem

Consider the problem 3-COLORING: Given an undirected graph G = (V, E), it is required to color each vertex in V with one of three colors, say 1, 2, and 3, such that no two adjacent vertices have the same color. We call such a coloring legal; otherwise, if two adjacent vertices have the same color, it is illegal. A coloring can be represented by an n-tuple (c_1, c_2, . . . , c_n) such that $c_i \in \{1,2,3\}$, $1 \leq i \leq n$. For example, (1,2,2,3,1) denotes a coloring of a graph with five vertices.

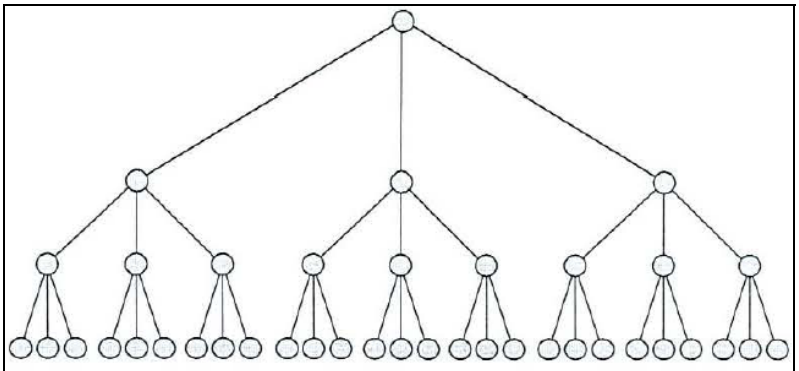


Fig. 13.1 The search tree for all possible 3-colorings for a graph with 3 vertices.

Ilustración 13, árbol de búsqueda potencial (3-coloring)

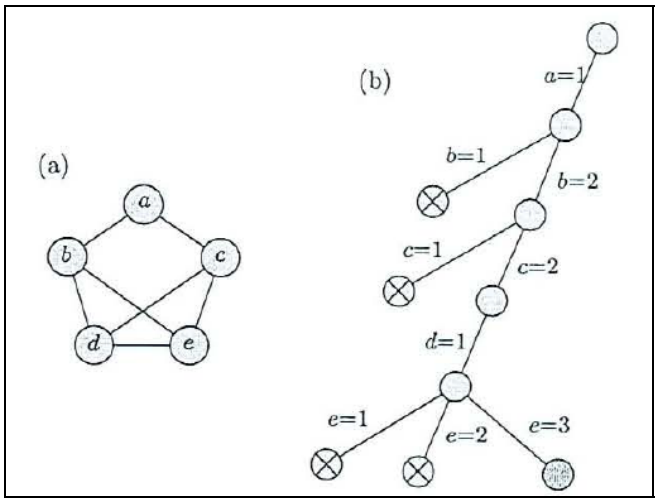


Ilustración 14, figura compuesta (grafo y árbol generado 3-coloring)

Algorithm 13.1 3-COLORREC
Input: An undirected graph $G = (V, E)$.
Output: A 3-coloring $c[1..n]$ of the vertices of G, where each $c[j]$ is 1,2 or 3.

```
1. for k ← 1 to n
2.     c[k] ← 0
3. end for
4. flag ← false
5. graphcolor(1)
6. if flag then output c
7. else output "no solution"

Procedure graphcolor(k)
1. for color = 1 to 3
2.     c[k] ← color
3.     if c is a legal coloring then set flag ← true and exit
4.     else if c is partial then graphcolor(k + 1)
5. end for
```

Algorithm 13.2 3-COLORITER
Input: An undirected graph $G = (V, E)$.
Output: A 3-coloring $c[1..n]$ of the vertices of G, where each $c[j]$ is 1,2 or 3.

```
1. for k ← 1 to n
2.     c[k] ← 0
3. end for
4. flag ← false
5. k ← 1
6. while k ≥ 1
7.     while c[k] ≤ 2
8.         c[k] ← c[k] + 1
9.         if c is a legal coloring then set flag ← true
           and exit from the two while loops.
10.        else if c is partial then k ← k + 1    {advance}
11.    end while
12.    c[k] ← 0
13.    k ← k − 1    {backtrack}
14. end while
15. if flag then output c
16. else output "no solution"
```

Libro	Capítulo/ apartado	Visualización	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.346 *Árbol de búsqueda* Fig. p.347 *Grafo y árbol de búsqueda*	Implementación p. 345 *Todas las soluciones Recursivo*

7.4 GRAPH COLORING

Let G be a graph and *m* be a given positive integer. We want to discover if the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only *m* colors are used.

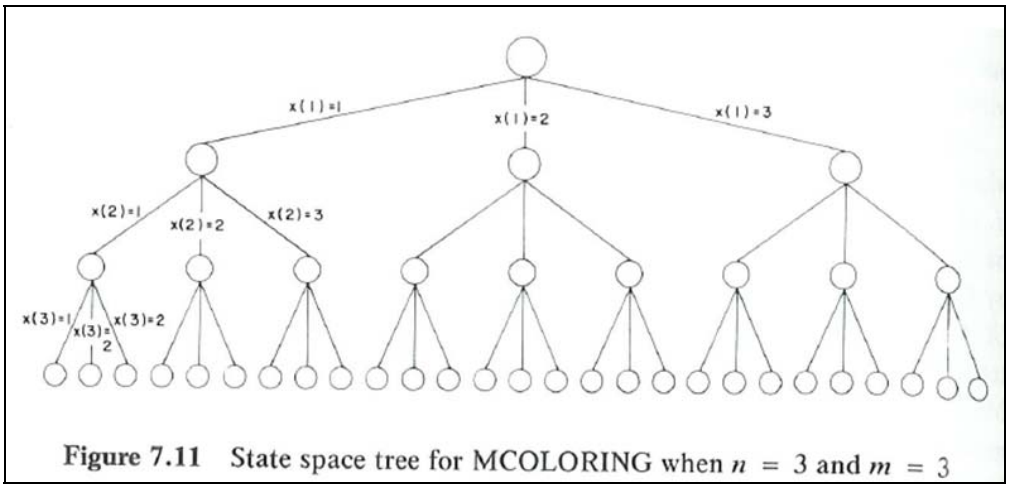


Figure 7.11 State space tree for MCOLORING when $n = 3$ and $m = 3$

Ilustración 15, árbol de búsqueda potencial (3-coloring)

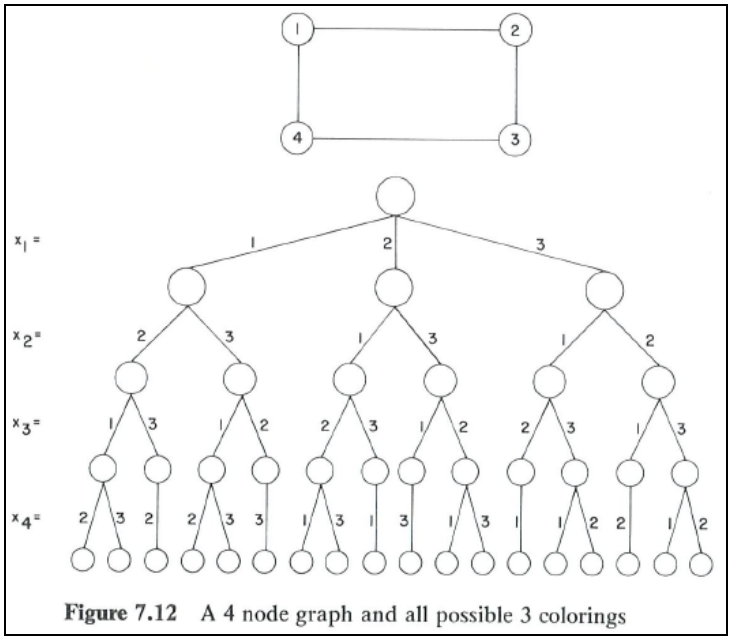


Figure 7.12 A 4 node graph and all possible 3 colorings

Ilustración 16, figura compuesta (grafo y árbol potencial, 3-coloring)

```
procedure MCOLORING(k)
  //This program was formed using the recursive backtracking schema.//
  //The graph is represented by its boolean adjacency matrix GRAPH(1:
  //n, 1:n). All assignments of 1, 2, ..., m to the vertices of the graph//
  //such that adjacent vertices are assigned distinct integers are printed.//
  //k is the index of the next vertex to color//
  global integer m, n, X(1:n) boolean GRAPH(1:n, 1:n)
  integer k
  loop  //generate all legal assignments for X(k)//
    call NEXTVALUE(k)  //assign to X(k) a legal color//
    if X(k) = 0 then exit endif  //no new color possible//
    if k = n
      then print(X)  //at most m colors are assigned to n vertices//
      else call MCOLORING(k + 1)
    endif
  repeat
end MCOLORING
```

Algorithm 7.7 Finding all m-colorings of a graph

Libro	Capítulo/ apartado	Visualización	Implementación
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.200 *Grafo* Fig. p.201 *Grafo* Fig. p.203 *Árbol de búsqueda*	Implementación p. 202 *Todas las soluciones Recursivo*

5.5 GRAPH COLORING

The *m*-Coloring Problem concerns finding all ways to color an undirected graph using at most *m* different colors, so that no two adjacent vertices are the same color. We usually call the m-Coloring Problem a unique problem for each value of *m*.

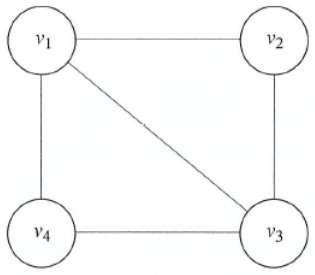


Figure 5.10 Graph for which there is no solution to the 2-Coloring Problem. A solution to the 3-Coloring Problem for this graph is shown in Example 5.5.

Ilustración 17, grafo 4 nodos

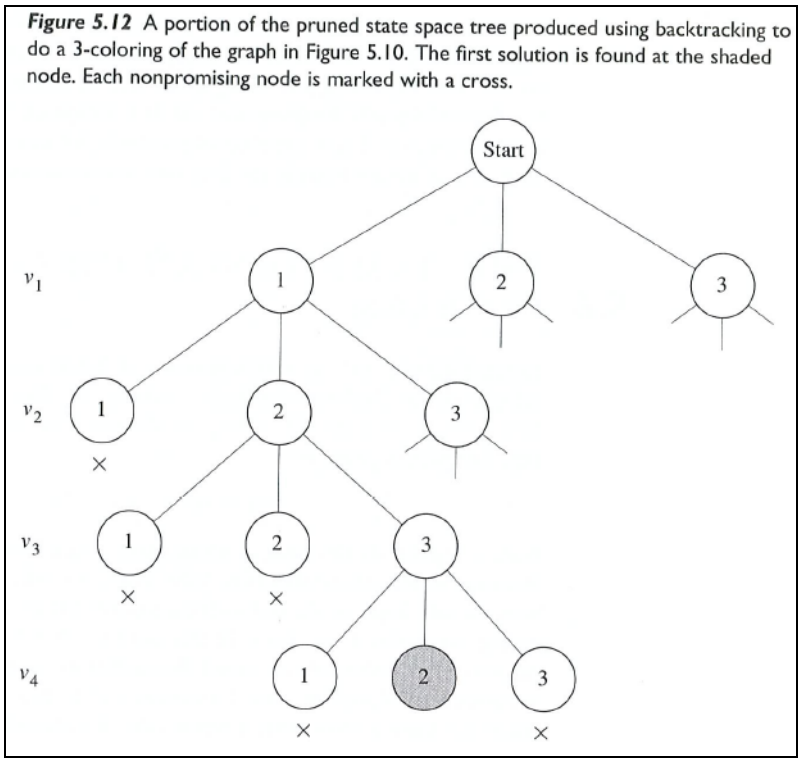


Ilustración 18, árbol de búsqueda generado abreviado (3-coloring)

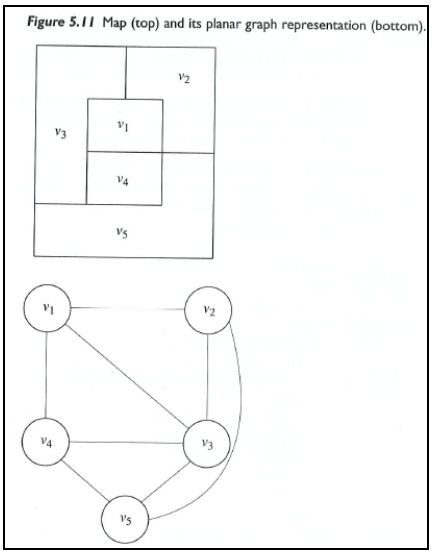


Ilustración 19, grafo planar 5 nodos

```
procedure m_coloring(i: index);
var
   color: integer
begin
   if promising(i) then
      if i=n then
         write(vcolor[1] through vcolor[n])
      else
         for color:= 1 to m do
            vcolor[i+1]:= color;
            m_coloring(i+1)
         end
      end
   end
end;
```

```
function promising(i: index): boolean;
var
   j: index;
begin
   promising:= true;
   j:=1;
   while j<i and promising do
      if W[i,j] and vcolor[i]=vcolor[j] then
         promising:= false
      end;
      j:= j+1
   end
end;
```

A continuación se muestra una tabla resumen de todos los libros donde se puede consultar el problema sobre coloreado de grafos, con el fin de que el lector pueda consultar de forma rápida en qué libros se puede encontrar el problema indicado y qué puede encontrar en cada libro.

Libro	Capítulo/ apartado	Visualización	Nomenclatura	Implementación
M. H. Alsuwaiyel, *Algorithms Design Techniques and Analysis*	Capítulo 13	Fig. p.354 *Árbol de búsqueda* Fig. p.355 *Grafo y árbol de búsqueda*	The 3-Coloring Problem	Pseudocódigo p. 356-7 *Recursivo e iterativo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.346 *Árbol de búsqueda* Fig. p.347 *Grafo y árbol de búsqueda*	GRAPH COLORING	Implementación p. 345 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.200 *Grafo* Fig. p.201 *Grafo* Fig. p.203 *Árbol de búsqueda*	GRAPH COLORING	Implementación p. 202 *Todas las soluciones* *Recursivo*

1.2.2 Ciclos Hamiltonianos

Este problema también se conoce como el problema del viajante de comercio o vendedor ambulante.

Problema 3: Dado un grafo conexo, se llama Ciclo Hamiltoniano a aquel ciclo que visita exactamente una vez cada vértice del grafo y vuelve al punto de partida. El problema consiste en detectar la presencia de ciclos Hamiltonianos en un grafo dado.

A continuación se detallan los libros donde se puede consultar el problema.

Libro	Capítulo/ apartado	Visualización	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.348 *Grafos*	Implementación p. 350 *Todas las soluciones Recursivo*

7.5 HAMILTONIAN CYCLES

Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round trip path along n edges of G which visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$ then the edges (v_1, v_{n+1}) are in E, $1 \leq i \leq n$ and the v_i are distinct except for v_1, and v_{n+1} which are equal.

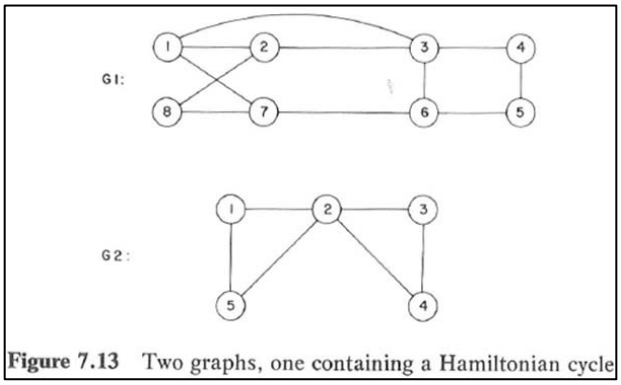


Figure 7.13 Two graphs, one containing a Hamiltonian cycle

Ilustración 20, dos grafos

```
procedure HAMILTONIAN(k)
  //This procedure uses the recursive formulation of backtracking//
  //to find all the Hamiltonian cycles of a graph. The graph//
  //is stored as a boolean adjacency matrix in GRAPH(1:n, 1:n). All//
  //cycles begin at vertex 1.//
  global integer X(1:n)
  local integer k, n
  loop   //generate values for X(k)//
     call NEXTVALUE(k)   //assign a legal next vertex to X(k)//
     if X(k) = 0 then return endif
     if k = n
        then print (X, '1')   //a cycle is printed//
        else call HAMILTONIAN(k + 1)
     endif
  repeat
end HAMILTONIAN
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.476 *Árbol de búsqueda*	Implementación p. 477 *Todas las soluciones Recursivo*

14.14 Un vendedor ambulante de alfombras persas tiene que recorrer *n* ciudades volviendo tras ello al punto de partida. El buen señor se ha informado sobre las posibles conexiones directas por ferrocarril entre las ciudades y sobre tarifas correspondientes. El vendedor desea conocer:

a) todos los circuitos en tren que recorren cada ciudad exactamente una vez y regresen a la ciudad de partida.

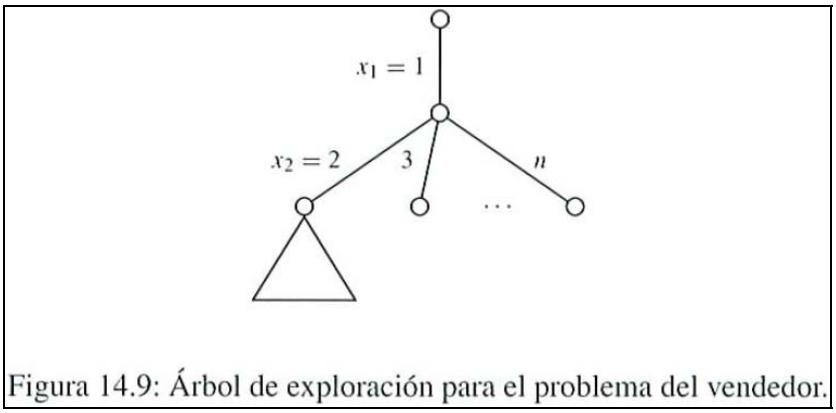


Figura 14.9: Árbol de exploración para el problema del vendedor.

Ilustración 21, árbol de búsqueda potencial abreviado (ciclos hamiltonianos)

```
proc ciclo-hamiltoniano-va(e G : grafo-val[n], sol[1..n] de 1..n, e k : 1..n, usado[1..n] de bool)
    para vértice = 2 hasta n hacer
        si ¬usado[vértice] ∧ gv-está-arista?(sol[k − 1], vértice, G) entonces
            sol[k] := vértice
            usado[vértice] := cierto     { marcar }
            si k = n entonces
                { falta comprobar que se cierra el ciclo }
                si gv-está-arista?(sol[n], 1, G) entonces imprimir(sol) fsi
            si no ciclo-hamiltoniano-va(G, sol, k + 1, usado)
            fsi
            usado[vértice] := falso     { desmarcar }
        fsi
    fpara
fproc
        proc ciclo-hamiltoniano(e G : grafo-val[n])
        var sol[1..n] de 1..n, usado[1..n] de bool
            sol[1] := 1
            usado[1] := cierto ;  usado[2..n] := [falso]
            ciclo-hamiltoniano-va(G, sol, 2, usado)
        ffun
```

Libro	Capítulo/ apartado	Visualización	Implementación
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.205 *Grafos*	Implementación p. 206 *Todas las soluciones Recursivo*

5.6 THE HAMILTONIAN CIRCUITS PROBLEM

…This problem is called the Hamiltonian Circuits Problem, named after Sir William Hamilton, who suggested it. The problem can be stated for either a directed graph (the way we stated the Traveling Salesperson Problem) or an undirected graph. Because it is usually stated for an undirected graph, this is the way we will state it here. As applied to Nancy's dilemma, this means that there is a unique twoway road connecting two cities when they are connected at all.

Specifically, given a connected, undirected graph, a *Hamiltonian Circuit* (also called a tour) is a path that starts at a given vertex, visits each vertex in the graph exactly once, and ends at the starting vertex. The graph in Figure 5.13(a) contains the Hamiltonian Circuit [v_1,v_2,v_8,v_7,v_6,v_5,v_4,v_3,v_1] but the one in Figure 5.13(b) does not contain a Hamiltonian Circuit. The Hamiltonian Circuits Problem determines the Hamiltonian Circuits in a connected, undirected graph.

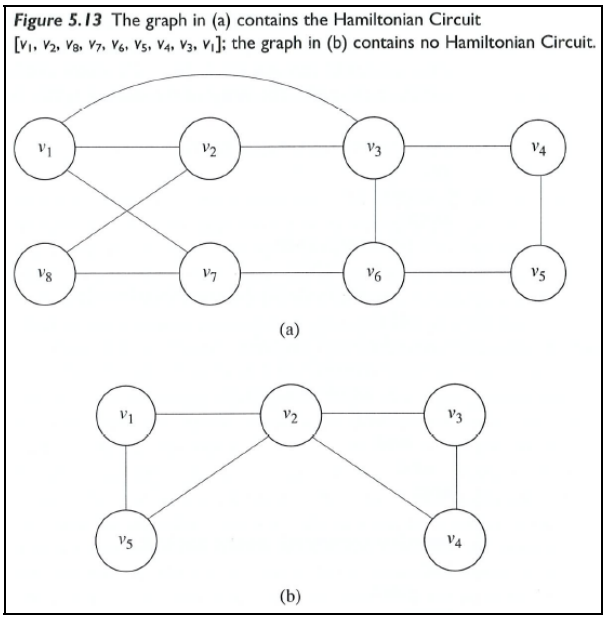


Ilustración 22, dos grafos

```
procedure hamiltonian(i: index);
var
  j: index
begin
  if promising(i) then
    if i=n-1 then
      write(vindex[0] through vindex[n-1])
    else
      for j:= 2 to n do                          {Try all vertices as}
        vindex[i+1]:= j;                         {next one.}
        hamiltonian(i+1);
      end
    end
  end
end;
function promising(i: index): boolean;
var
  j: index;
begin
  if i=n-1 and not W([vindex[n-1], vindex[0]) then   {First vertex must}
    promising:= false                                {be adjacent to last.}
  else if i > 0 and not W([vindex[i-1], vindex[i]) then {ith vertex must be}
    promising:= false                                {adjacent to (i - 1)st.}
  else
    promising:= true;
    j:=1;
    while j<i and promising do                       {Check if vertex is}
      if vindex[i] = vindex[j] then                  {already selected.}
        promising:= false;
      j:= j+1
    end
  end
end;
```

Libro	Capítulo/ apartado	Visualización	Implementación
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Fig. p.839 *Grafos* Fig. p.840 *Árbol de búsqueda*	Implementación p. 862 *Todas las soluciones Recursivo*

Example 21.3 [Traveling Salesperson] In this problem we are given an *n* vertex network (either directed or undirected) and are to find a cycle of minimum cost that includes all *n* vertices. Any cycle that includes all *n* vertices of a network is called a tour. In the traveling-salesperson problem, we are to find a least-cost our.

A four-vertex undirected network appears in Figure 21.4. Some of the tours in this network are 1,2,4,3,1; 1,3,2,4,1; and 1,4,3,2,1. The tours 2,4,3,1,2; 4,3,1,2,4; and 3,1,2,4,3 are the same as the tour 1,2,4,3,1, whereas the tour 1,3,4,2,1 is the reverse of the tour 1,2,4,3,1. The cost of the tour 1,2,4,3,1 is 66; that of 1,3,2,4,1 is 25; and that of 1,4,3,2,1 is 59. 1,3,2,4,1 is the least-cost tour in the network.

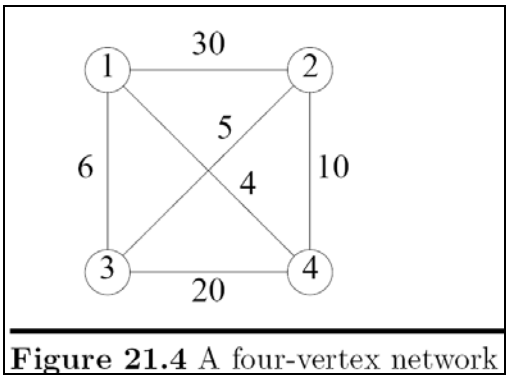


Figure 21.4 A four-vertex network

Ilustración 23, grafo de 4 nodos

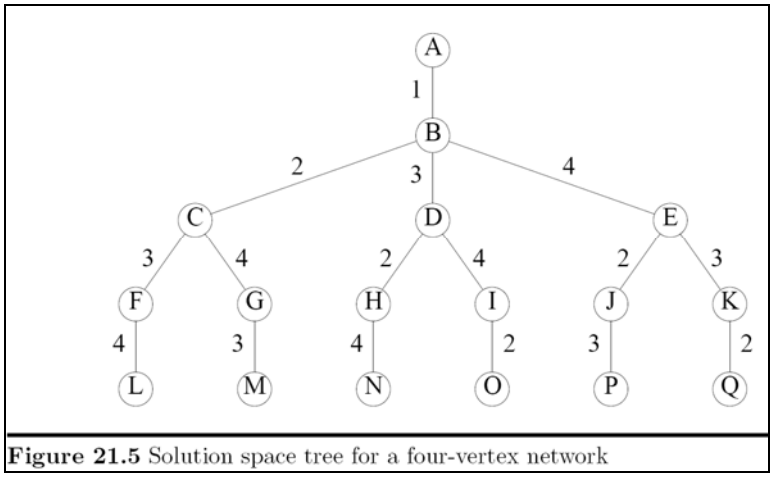


Figure 21.5 Solution space tree for a four-vertex network

Ilustración 24, árbol de búsqueda potencial (ciclos hamiltonianos)

```
private void rTSP(int currentLevel)
{// search from a node at currentLevel
   if (currentLevel == n)
   {// at parent of a leaf
      // complete tour by adding last two edges
      if (a[partialTour[n - 1]][partialTour[n]] != null &&
          a[partialTour[n]][1] != null &&
          (costOfBestTourSoFar == null ||
          ((Operable) ((Operable) costOfPartialTour.
                       add(a[partialTour[n - 1]][partialTour[n]])).
                       add(a[partialTour[n]][1])).
                       compareTo(costOfBestTourSoFar) < 0))
      {// better tour found
         for (int j = 1; j <= n; j++)
            bestTourSoFar[j] = partialTour[j];
         costOfBestTourSoFar = ((Operable) costOfPartialTour.
                          add(a[partialTour[n - 1]][partialTour[n]])).
                          add(a[partialTour[n]][1]);
      }
   }
   else
   {// try out subtrees
      for (int j = currentLevel; j <= n; j++)
         // is move to subtree labeled partialTour[j] possible?
         if (a[partialTour[currentLevel - 1]][partialTour[j]] != null &&
             (costOfBestTourSoFar == null ||
              ((Operable) costOfPartialTour.
                add(a[partialTour[currentLevel - 1]][partialTour[j]])).
                compareTo(costOfBestTourSoFar) < 0))
         {// search this subtree
            MyMath.swap(partialTour, currentLevel, j);
            costOfPartialTour.increment(a[partialTour[currentLevel - 1]]
                                        [partialTour[currentLevel]]);
            rTSP(currentLevel + 1);
            costOfPartialTour.decrement(a[partialTour[currentLevel - 1]]
                                        [partialTour[currentLevel]]);
            MyMath.swap(partialTour, currentLevel, j);
         }
   }
}
```

Program 21.13 Recursive backtracking for traveling salesperson

A continuación se muestra una tabla resumen de todos los libros donde se puede consultar distintos problemas sobre ciclos hamiltonianos.

Libro	Capítulo/ apartado	Visualización	Nomenclatura	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.348 *Grafos*	HAMILTONIAN CYCLES	Implementación p. 350 *Todas las soluciones Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.476 *Árbol de búsqueda*	14.14	Implementación p. 477 *Todas las soluciones Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.205 *Grafos*	THE HAMILTONIAN CIRCUITS PROBLEM	Implementación p. 206 *Todas las soluciones Recursivo*
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Fig. p.839 *Grafos* Fig. p.840 *Árbol de búsqueda*	Example 21.3 [Traveling Salesperson]	Implementación p. 862 *Todas las soluciones Recursivo*

1.2.3 Otros problemas de grafos

1.2.3.1 Caminos de un grafo

Libro	Capítulo/ apartado	Visualización	Implementación
S. Skiena, 1998, *The Algorithm Design Manual*	Capítulo 7	Figura p. 236 Árbol de búsqueda	Implementación p. 237 Todas las soluciones Recursivo

Problema 4: [**7.1.3 Constructing All Paths in a Graph]** Enumerating all the simple *s* to *t* paths through a given graph is a more complicated problem than listing permutations or subsets. There is no explicit formula that counts the number of solutions as a function of the number of edges or vertices, because the number of paths depends upon the structure of the graph.

The starting point of any path from *s* to *t* is always *s*. Thus, *s* is the only candidate for the first position and S1 = {*s*}. The possible candidates for the second position are the vertices *v* such that (*s*,*v*) is an edge of the graph, for the path wanders from vertex to vertex using edges to define the legal step.

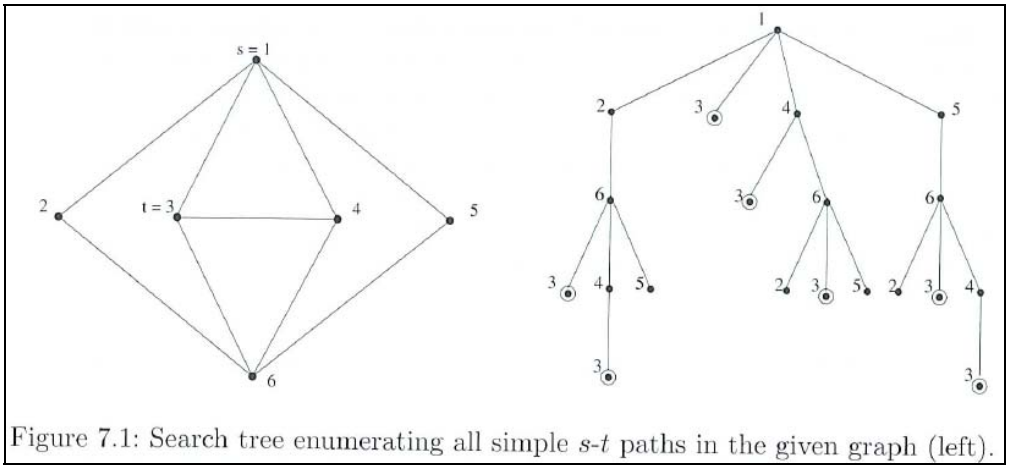


Figure 7.1: Search tree enumerating all simple *s*-*t* paths in the given graph (left).

Ilustración 25, figura compuesta (grafo y árbol de búsqueda generado)

.

```
bool finished = FALSE;            /* found all solutions yet?
backtrack(int a[], int k, data input)
{
     int c[MAXCANDIDATES];     /* candidates for next positic
     int ncandidates;          /* next position candidate cou
     int i;                    /* counter */

     if (is_a_solution(a,k,input))
            process_solution(a,k,input);
     else {
            k = k+1;
            construct_candidates(a,k,input,c,&ncandidates);
            for (i=0; i<ncandidates; i++) {
                   a[k] = c[i];
                   make_move(a,k,input);
                   backtrack(a,k,input);
                   unmake_move(a,k,input);
                   if (finished) return;   /* terminate ear
            }
     }
}
```

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
     int i;                    /* counters */
     bool in_sol[NMAX];        /* what's already in the solution? */
     edgenode *p;              /* temporary pointer */
     int last;                 /* last vertex on current path */

     for (i=1; i<NMAX; i++) in_sol[i] = FALSE;
     for (i=1; i<k; i++) in_sol[ a[i] ] = TRUE;

     if (k==1) {               /* always start from vertex 1 */
            c[0] = 1;
            *ncandidates = 1;
     }
     else    {
            *ncandidates = 0;
            last = a[k-1];
            p = g.edges[last];
            while (p != NULL) {
                   if (!in_sol[ p->y ]) {
                          c[*ncandidates] = p->y;
                          *ncandidates = *ncandidates + 1;
                   }
                   p = p->next;
            }
     }
}
```

1.2.3.2 Grafos isomorfos

Libro	Capítulo/ apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.457 *Árbol de búsqueda*	Implementación p. 457-8 *Todas las soluciones Recursivo*

Problema 5: [14.3] Dos grafos dirigidos G y G' son isomorfos si existe una función biyectiva *f* ente los vértices de ambos grafos de forma que (*x*,*y*) es una arista de G y si y solo sí (f(*x*),f(*y*)) es una arista de G'. Por ejemplo, los grafos de la Figura 14.2 son isomorfos. Diseñar un algoritmo que verifique si dos grafos dados son isomorfos.

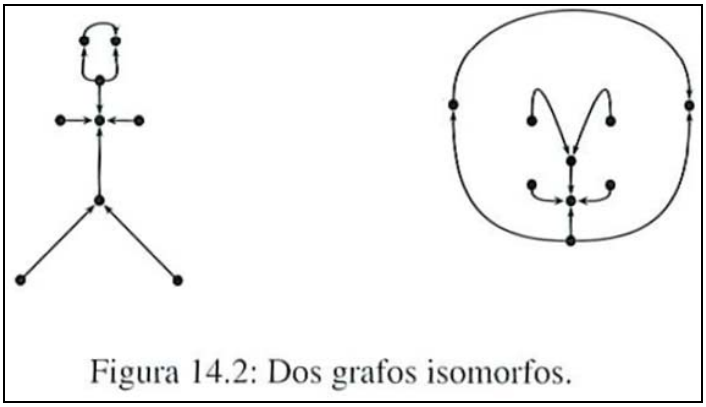


Figura 14.2: Dos grafos isomorfos.

Ilustración 26, dos grafos isomorfos

```
fun obtener-grados(G : grafo[n]) dev ⟨ E[1..n], S[1..n] de 0..n ⟩
    E[1..n] := [0] ;  S[1..n] := [0]
    para i = 1 hasta n hacer
        para j = 1 hasta n hacer
            si G[i, j] entonces E[j] := E[j] + 1 ;  S[i] := S[i] + 1 fsi
        fpara
    fpara
ffun

proc isomorfos-va(e G, G' : grafo[n], e E[1..n], S[1..n], E'[1..n], S'[1..n] de 0..n,
                  sol[1..n] de 1..n, e k : 1..n, usado[1..n] de bool, éxito : bool)
    vértice := 1
    mientras vértice ≤ n ∧ ¬éxito hacer
        si ¬usado[vértice] entonces
            sol[k] := vértice
            usado[vértice] := cierto    { marcar }
            si (E[k] = E'[sol[k]] ∧ S[k] = S'[sol[k]]) ∧ hay-aristas?(G, G', sol, k) entonces
                si k = n entonces éxito := cierto
                si no isomorfos-va(G, G', E, S, E', S', sol, k + 1, usado, éxito)
                fsi
            fsi
            usado[vértice] := falso    { desmarcar }
        fsi
        vértice := vértice + 1
    fmientras
fproc

fun hay-aristas?(G, G' : grafo[n], sol[1..n] de nat, k : 1..n) dev respuesta : bool
    i := 1
    respuesta := cierto
    mientras respuesta ∧ i < k hacer
        respuesta := (G[i, k] = G'[sol[i], sol[k]]) ∧ (G[k, i] = G'[sol[k], sol[i]])
        i := i + 1
    fmientras
ffun

fun isomorfos?(G, G' : grafo[n]) dev ⟨ éxito : bool, sol[1..n] de 1..n ⟩
var usado[1..n] de bool,  E[1..n], S[1..n], E'[1..n], S'[1..n] de 0..n
    usado[1..n] := [falso]
    ⟨ E, S ⟩ := obtener-grados(G)
    ⟨ E', S' ⟩ := obtener-grados(G')
    éxito := falso
    isomorfos-va(G, G', E, S, E', S', sol, 1, usado, éxito)
ffun
```

1.2.1 Tabla resumen sobre problemas de grafos

En la siguiente tabla se muestra un resumen de los libros en los que se puede consultar los distintos problemas sobre grafos mostrados anteriormente.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
M. H. Alsuwaiyel, *Algorithm Design Techniques and Analysis*	Capítulo 13	Fig. p.354 *Árbol de búsqueda* Fig. p.355 *Grafo y árbol de búsqueda*	The3-Coloring Problem	Pseudocódigo p. 356-7 *Recursivo e iterativo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.346 *Árbol de búsqueda* Fig. p.347 *Grafo y árbol de búsqueda*	GRAPH COLORING	Implementación p. 345 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.200 *Grafo* Fig. p.201 *Grafo* Fig. p.203 *Árbol de búsqueda*	GRAPH COLORING	Implementación p. 202 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.348 *Grafos*	HAMILTONIAN CYCLES	Implementación p. 350 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.476 *Árbol de búsqueda*	14.14	Implementación p. 477 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.205 *Grafos*	THE HAMILTONIAN CIRCUITS PROBLEM	Implementación p. 206 *Todas las soluciones* *Recursivo*
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Fig. p.839 *Grafos* Fig. p.840 *Árbol*	Example 21.3 [Traveling Salesperson]	Implementación p. 862 *Todas las soluciones*

		de búsqueda		*Recursivo*
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	Fig. p.236 *Árbol de búsqueda*	Constructing All Paths in a Graph	Implementación p. 237 *Todas las soluciones Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.457 *Árbol de búsqueda*	Grafos isomorfos	Implementación p. 457-8 *Todas las soluciones Recursivo*

1.3 Problemas de caracteres

En este apartado se verán un conjunto de problemas típicos sobre cadenas de caracteres.

Problema 6: Dado un conjunto de caracteres generar aquellas palabras que cumplan con las restricciones que impone el problema concreto.

A continuación se listan todos los problemas encontrados en los libros sobre cadenas de caracteres. Como siempre, en primer lugar se indica, en forma de tabla, el libro donde se puede consultar el problema correspondiente, qué se puede consultar en dicho libro, el enunciado del problema y aquellas imágenes y código que se puede consultar.

Libro	Capítulo/apartado	Visualización	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	-	Pseudocódigo p. 77-79 *Todas las soluciones* *Recursivo*

4.1 Generar palabras con restricciones

Dado el conjunto de caracteres alfabéticos, se quieren generar todas las palabras de cuatro letras que cumplan las siguientes condiciones: a) La primera letra debe ser vocal. b) Sólo pueden aparecer dos vocales seguidas si son diferentes. c) No puede haber ni tres vocales ni tres consonantes seguidas. d) Existe un conjunto *C* de parejas de consonantes que no pueden aparecer seguidas.

```
fun compleciones (e:ensayo) dev lista de palabra
 lista-compleciones ← lista-vacía
   para cada letra en {a,b,c,..,z} hacer
   hijo ← añadir-letra (e,letra);
   lista-compleciones ← añadir(lista-compleciones,hijo)
   fpara
   dev lista-compleciones
ffun
fun añadir-letra (e:ensayo,l:letra) dev ensayo
 nuevo-ensayo ← e;
 nuevo-ensayo.palabra ← añadir(nuevo-ensayo.palabra,l);
 nuevo-ensayo.indice ← nuevo-ensayo.indice +1
   dev nuevo-ensayo
ffun
```

```
fun condiciones de poda (e:ensayo) dev bool
  dev condicion1(e) ∧ condicion2(e) ∧
    condicion3(e)∧ condicion4(e)
ffun

fun caracteres dev lista de ensayo
  dev vuelta-atrás (ensayo-vacio)
ffun

fun vocal (l:letra) dev bool
  dev pertenece(l,{a,e,i,o,u})
ffun

fun condicion1 (e:ensayo) dev bool
  dev vocal (e.palabra[1])
ffun

fun condicion2 (e:ensayo) dev bool
   dev e.indice < 2 ∨
   consonante(ensayo.palabra[e.indice]) ∨
   e.palabra[e.indice] ≠ e.palabra[e.indice-1]
ffun

fun condicion3 (e:ensayo) dev bool
 i ← e.indice;
  dev i < 3 ∨
    vocal(e.palabra[i-1]) ≠ vocal(e.palabra[i]) ∨
    vocal(e.palabra[i-2]) ≠ vocal(e.palabra[i])
ffun

fun condicion4 (e:ensayo) dev bool
 i ← e.indice;
  dev ¬pertenece((e.palabra[i-1],e.palabra[i]),C)
ffun


fun consonante (l:letra) dev bool
  dev ¬ vocal (l)
ffun
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A., Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.454 *Árbol de búsqueda* -	14.1 Implementación p. 455 *Todas las soluciones* *Recursivo* 14.6 Implementación p. 463-4 *Todas las soluciones* *Recursivo*

14.1 Dadas *m* letras, todas ellas diferentes, y $m \leq n$, diseñar un algoritmo que calcule todas las palabras con m letras diferentes escogidas entre las dadas.

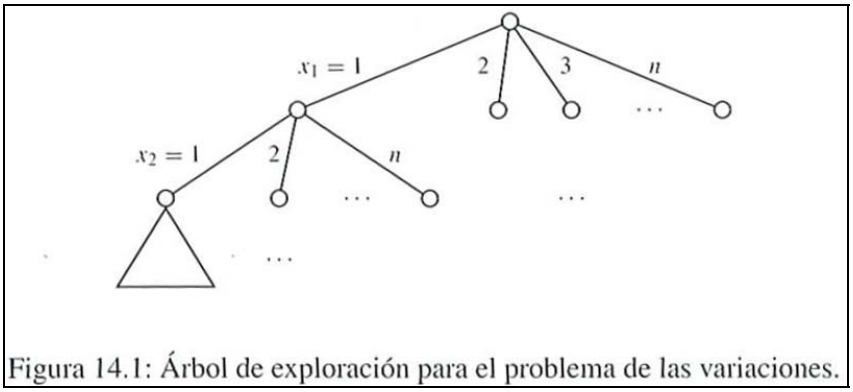


Figura 14.1: Árbol de exploración para el problema de las variaciones.

Ilustración 27, árbol potencial abreviado (variaciones)

```
proc variaciones-va1(e n : nat+, sol[1..m] de 1..n, e k : 1..m)
    para j = 1 hasta n hacer
        sol[k] := j
        si no-repetida?(sol, k) entonces
            si k = m entonces imprimir(sol)     { es una solución }
            si no variaciones-va1(n, sol, k + 1)
            fsi
        fsi
    fpara
fproc

fun no-repetida?(sol[1..m] de nat, k : 1..n) dev respuesta : bool
    i := 1
    mientras sol[i] ≠ sol[k] hacer
        i := i + 1
    fmientras
    respuesta := (i = k)
ffun
```

```
proc variaciones-va2(e n : nat+, sol[1..m] de 1..n, e k : 1..m, usada[1..n] de bool)
    para j = 1 hasta n hacer
        si ¬usada[j] entonces
            sol[k] := j
            usada[j] := cierto     { marcar }
            si k = m entonces imprimir(sol)
            si no variaciones-va2(n, sol, k + 1, usada)
            fsi
            usada[j] := falso      { desmarcar }
        fsi
    fpara
fproc

proc variaciones(e n : nat+)
var sol[1..m] de 1..n, usada[1..n] de bool
    usada[1..n] := [falso]
    variaciones-va2(n, sol, 1, usada)
fproc
```

14.6 Se dispone de n cubos identificados por un número del 1 al n. Cada cubo tiene impresa en cada una de sus caras una letra distinta. Se indica además una palabra de n letras. Se trata de colocar los n cubos uno a continuación de otro, de forma que con esa disposición se pueda formar la palabra dada. Como en diferentes cubos puede haber letras repetidas, la solución puede no ser única o no existir.

```
fun cubos(P[1..n] de car, C[1..n] de cubo) dev (éxito : bool, sol[1..n] de par)
var usado[1..n]
    éxito := falso
    usado[1..n] := [falso]
    cubos-va(P, C, sol, 1, usado, éxito)
ffun
```

```
proc cubos-va(e P[1..n] de car, e C[1..n] de cubo, sol[1..n] de par, e k : 1..n,
              usado[1..n] de bool, éxito : bool)
    cubo := 1     { recorre los cubos }
    mientras cubo ≤ n ∧ ¬éxito hacer
        si ¬usado[cubo] entonces
            usado[cubo] := cierto     { marcar }
            cara := 1     { recorre las caras del cubo }
            mientras cara ≤ 6 ∧ ¬éxito hacer
                si P[k] = C[cubo][cara] entonces
                    sol[k].cubo := cubo ; sol[k].cara := cara
                    si k = n entonces éxito := cierto
                    si no cubos-va(P, C, sol, k + 1, usado, éxito)
                    fsi
                fsi
                cara := cara + 1
            fmientras
            usado[cubo] := falso     { desmarcar }
        fsi
        cubo := cubo + 1
    fmientras
fproc
```

Libro	Capítulo/ apartado	Visualización	Implementación
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	-	Implementación p. 235-6 *Todas las soluciones Recursivo*

7.1.2 Constructing All Permutations

Counting permutations of {1, .. . , *n*} is a necessary prerequisite to generating them. There arc *n* distinct choices for the value of the first element of a permutation.

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidate
{
        int i;                          /* counter */
        bool in_perm[NMAX];             /* who is in the permutation? *

        for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
        for (i=0; i<k; i++) in_perm[ a[i] ] = TRUE;

        *ncandidates = 0;
        for (i=1; i<=n; i++)
                if (in_perm[i] == FALSE) {
                        c[ *ncandidates] = i;
                        *ncandidates = *ncandidates + 1;
                }
}

process_solution(int a[], int k)
{
        int i;                          /* counter */

        for (i=1; i<=k; i++) printf(" %d",a[i]);

        printf("\n");
}
```

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);
}

generate_permutations(int n)
{
        int a[NMAX];                    /* solution vector

        backtrack(a,0,n);
}
```

1.3.1 Tabla resumen sobre problemas de caracteres

En la siguiente tabla se adjunta un resumen de los libros que contienen problemas sobre cadenas de caracteres.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	-	Generar palabras con restricciones	Pseudocódigo p. 77-79 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.454 *Árbol de búsqueda* -	14.1 14.6	Implementación p. 455 *Todas las soluciones* *Recursivo* 14.6 Implementación p. 463-4 *Todas las soluciones* *Recursivo*
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	-	Constructing All Permutations	Implementación p. 235-6 *Todas las soluciones* *Recursivo*

1.4 Problema del salto de caballo

Se pasa a detallar el enunciado general de dicho problema.

Problema 7: Partiendo de una cuadrícula de $n \times n$ casillas y un caballo de ajedrez colocado en una posición cualquiera (x,y), el problema consiste en encontrar una secuencia de movimientos, siguiendo las reglas del ajedrez, de modo que el caballo recorra todas las casillas del tablero visitando cada casilla una sola vez.

Se pasa a describir los libros donde se puede consultar el problema conocido como el salto de caballo.

Libro	Capítulo/ apartado	Visualización	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Fig. p.86 *Tablero*	Pseudocódigo p. 87-90 *Primera solución Recursivo*

4.3 Recorrido del Caballo de Ajedrez
Sobre un tablero de ajedrez de tamaño N (con N>5) se coloca un caballo.

Determinar una secuencia de movimientos del caballo que pase por todas las casillas del tablero sin pasar dos veces por la misma casilla. La posición inicial podrá ser cualquiera.

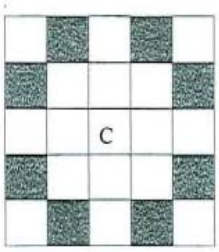


Figura 4.2: Las casillas grises son las alcanzables por un movimiento de caballo desde la casilla **C**

Ilustración 28, tablero movimientos caballo de ajedrez

```
fun  compleciones(ensayo)
 lista ← lista-vacía;
 (i, j)  ← ensayo.ultima-posición;
 N ← ensayo.numero-movs;
 ultimo-tablero ← ensayo.tablero;
 /* Generamos todas las posibles ramas */
  para  i ← -2  hasta  2  hacer
  para  j ← -2  hasta  2  hacer
   si   ( abs(i) + abs(j)  =  3)  ∧
          ( ensayo.tablero[i, j] = 0)
   entonces    hacer
     nuevo-tablero  ← ultimo-tablero;
     nuevo-tablero[i,j]  ← N+1;
     nueva-posición  ← (i,j);
     nuevo-ensayo  ← < nuevo-tablero,nueva-posición,N+1 >
     lista ← añadir(lista,nuevo-ensayo);
   fsi
 fpara
 fpara
dev  lista
ffun
fun  saltocaballo (ensayo)
 si  válido (ensayo )
 entonces    escribe  ensayo
 si no    hacer
  lista ← compleciones(ensayo)
   mientras  no vacía(lista)   hacer
     w ← PrimerElemento(lista)
     saltocaballo(w)
     lista ← Resto(lista)
   fmientras
 fsi
ffun
fun  válido (ensayo)
 dev (ensayo.numero-movs = N^2 )
ffun
```

```
fun  vuelta-atrás (ensayo)
  si  válido (ensayo)
  entonces   dev  ensayo
  si no para hijo ∈ compleciones(ensayo)
      hacer si  condiciones de poda(hijo)
        entonces  vuelta-atrás(hijo)
  fsi
ffun
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	*Tablero*	Implementación p. 468-70 *Primera solución* *Vuelta a la casilla de partida* *Recursivo*

14.10 Dado un tablero de ajedrez de *nxn* posiciones, y un caballo colocado en una posición arbitraria (*x*,*y*), se pide diseñar un algoritmo que determine (si es que existe) una secuencia de n^2-1 movimientos de caballo de tal forma que se visiten todos los cuadrados del tablero una sola vez. Modificar el algoritmo añadiendo el requisito adicional de que, con un último movimiento, el caballo debe volver a la casilla de partida.

```
fun nueva-casilla(act : posición, mov : 1..8) dev sig : posición
   ⟨ i, j ⟩ := ⟨ act.fila, act.columna ⟩
   casos
        mov = 1 →  ⟨ i', j' ⟩ := ⟨ i − 2, j + 1 ⟩
      □ mov = 2 →  ⟨ i', j' ⟩ := ⟨ i − 1, j + 2 ⟩
      □ mov = 3 →  ⟨ i', j' ⟩ := ⟨ i + 1, j + 2 ⟩
      □ mov = 4 →  ⟨ i', j' ⟩ := ⟨ i + 2, j + 1 ⟩
      □ mov = 5 →  ⟨ i', j' ⟩ := ⟨ i + 2, j − 1 ⟩
      □ mov = 6 →  ⟨ i', j' ⟩ := ⟨ i + 1, j − 2 ⟩
      □ mov = 7 →  ⟨ i', j' ⟩ := ⟨ i − 1, j − 2 ⟩
      □ mov = 8 →  ⟨ i', j' ⟩ := ⟨ i − 2, j − 1 ⟩
   fcasos
   ⟨ sig.fila, sig.columna ⟩ := ⟨ i', j' ⟩
ffun
```

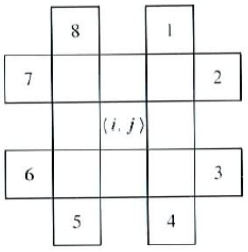


Ilustración 29, tablero parcial con movimientos caballo válidos

```
fun cierra?(pos1, pos2 : posición) dev respuesta : bool
    respuesta := falso
    movimiento := 1
    mientras movimiento ≤ 8 ∧ ¬respuesta hacer
        respuesta := (pos2 = nueva-casilla(pos1, movimiento))
        movimiento := movimiento + 1
    fmientras
ffun
```

```
proc caballo-va(sol[1..n²] de posición, e k : 1..n², usada[1..n, 1..n] de bool, éxito : bool)
    movimiento := 1    { genera movimientos }
    mientras movimiento ≤ 8 ∧ ¬éxito hacer
        sol[k] := nueva-casilla(sol[k − 1], movimiento)
        ⟨i, j⟩ := ⟨sol[k].fila, sol[k].columna⟩
        si 1 ≤ i ∧ i ≤ n ∧ 1 ≤ j ∧ j ≤ n ∧ ¬usada[i, j] entonces
            usada[i, j] := cierto    { marcar }
            si k = n² entonces éxito := cierto
            si no caballo-va(sol, k + 1, usada, éxito)
            fsi
            usada[i, j] := falso    { desmarcar }
        fsi
        movimiento := movimiento + 1    { siguiente movimiento }
    fmientras
fproc
```

```
fun caballo(n : nat+, pos : posición) dev ⟨éxito : bool, sol[1..n²] de posición⟩
var usada[1..n, 1..n] de bool
    sol[1] := pos    { desde donde empieza el caballo }
    usada[1..n, 1..n] := [falso]
    usada[pos.fila, pos.columna] := cierto
    éxito := falso
    caballo-va(sol, 2, usada, éxito)
fl tipos
    posición = reg
                   fila : 1..n
                   columna : 1..n
               freg
  ftipos
```

La modificación en la cual se comprueba que el recorrido se cierra volviendo a la casilla inicial es:

```
si k = n² entonces
    éxito := cierra?(sol[n²], sol[1])
si no caballo-va(sol, k + 1, usada, éxito)
fsi
```

1.4.1 Tabla resumen sobre el problema del salto de caballo

En la siguiente tabla se adjunta un resumen de los libros donde se puede consultar el problema del salto de caballo.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Fig. p.86 *Tablero*	Recorrido del Caballo de Ajedrez	Pseudocódigo p. 87-90 *Primera solución Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	*Tablero*	14.10	Implementación p. 468-70 *Primera solución Vuelta a la casilla de partida Recursivo*

1.5 Problema del laberinto

Problema 8: Dada una matriz con un laberinto, el problema consiste en diseñar un algoritmo que encuentre un camino, si existe, para ir de la casilla de entrada (1,1) a la casilla de salida (*n*,*n*).

A continuación se muestra el libro donde se puede consultar dicho problema.

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	*Tablero*	Implementación p. 472-3 *Recursivo*

14.12. A partir de una matriz booleana L[1..*n*,1..*n*] se puede representar un laberinto de la siguiente forma: a partir de una casilla dada, los movimientos posibles son desplazarse a cada una de las cuatro casillas adyacentes (vertical y horizontalmente). Si L[*i*,*j*]= falso no se puede pasar por la casilla. Suponiendo que L[1,1]=L[*n*,*n*]=cierto escribir un algoritmo que encuentre, si existe, un camino de la casilla {1,1} a la casilla {*n*,*n*}.

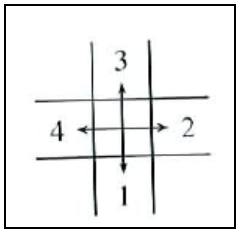
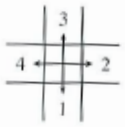


Ilustración 30, movimientos válidos laberinto

```
tipos
  posición= reg
              fila : 1..n
              columna : 1..n
            freg
ftipos
fun casilla-válida?(L[1..n, 1..n], usada[1..n, 1..n] de bool, pos : posición) dev respuesta : bool
    (i, j) := (pos.fila, pos.columna)
    respuesta := 1 ≤ i ∧ i ≤ n ∧ 1 ≤ j ∧ j ≤ n ∧ L[i, j] ∧ ¬usada[i, j]
ffun
```

```
fun siguiente(m : 1..4, act : posición) dev sig : posición
    ⟨i, j⟩ := ⟨act.fila, act.columna⟩
    casos
          m = 1 →  ⟨i', j'⟩ := ⟨i + 1, j⟩
       [] m = 2 →  ⟨i', j'⟩ := ⟨i, j + 1⟩
       [] m = 3 →  ⟨i', j'⟩ := ⟨i − 1, j⟩
       [] m = 4 →  ⟨i', j'⟩ := ⟨i, j − 1⟩
    fcasos
    ⟨sig.fila, sig.columna⟩ := ⟨i', j'⟩
ffun
```




```
fun laberinto(L[1..n, 1..n] de bool) dev (éxito : bool, sol[1..n²] de posición, núm-movimientos : 1..n²)
var usada[1..n, 1..n] de bool
    ⟨sol[1].fila, sol[1].columna⟩ := ⟨1, 1⟩
    usada[1..n, 1..n] := [falso] ;  usada[1, 1] := cierto
    éxito := falso
    laberinto-va(L, sol, 2, usada, éxito, núm-movimientos)
ffun
```

```
proc laberinto-va(e L[1..n, 1..n] de bool, sol[1..n²] de posición, e k : 1..n², usada[1..n, 1..n] de bool,
                  éxito : bool, núm-movimientos : 1..n²)
    m := 1    { genera movimientos }
    mientras m ≤ 4 ∧ ¬éxito hacer
        pos := siguiente(m, sol[k − 1])
        si casilla-válida?(L, usada, pos) entonces
            sol[k] := pos
            usada[pos.fila, pos.columna] := cierto    { marcar }
            si pos.fila = n ∧ pos.columna = n entonces
                éxito := cierto
                núm-movimientos := k
            si no
                laberinto-va(L, sol, k + 1, usada, éxito, núm-movimientos)
            fsi
        fsi
        m := m + 1    { siguiente movimiento }
    fmientras
fproc
```

1.6 Problemas de juegos

En este apartado se listarán los libros donde se pueden consultar distintos problemas sobre juegos resueltos con la técnica de vuelta atrás.

1.6.1 Problema del Cuadrado Mágico

Problema 9: El problema consiste en colocar los números de 1 a n^2, siendo n el número de filas y de columnas del cuadrado mágico, de modo que la suma de los números por filas, columnas y diagonales sea la misma.

Este problema se puede consultar en el siguiente libro:

Libro	Capítulo/apartado	Visualización	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Fig. p.92 *Cuadrado mágico*	Pseudocódigo p. 92-3 *Todas las soluciones Recursivo*

4.4 Cuadrado mágico de suma 15

Sobre una cuadrícula de 3×3 casillas se quieren colocar los dígitos del 1 al 9 sin repetir ninguno y de manera que sumen 15 en todas direcciones, incluidas las diagonales. Diseñar un algoritmo que busque todas las soluciones posibles.

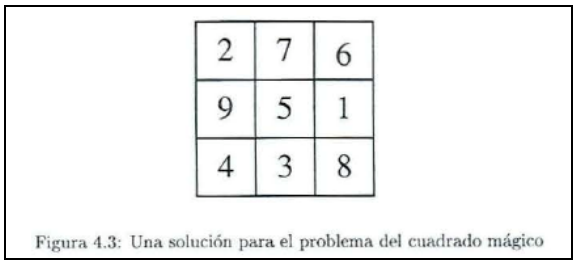


Figura 4.3: Una solución para el problema del cuadrado mágico

Ilustración 31, tablero solución cuadrado mágico

```
fun suma-fila(matriz: vector [1..3,1..3] de natural ; fila: natural )
    s ← 0;
     para i ← 1 hasta 3 hacer
       s ← s + matriz[fila,i];
     fpara
     dev s
fun num-f(matriz: vector [1..3,1..3] de natural ; fila: natural )
fun num-c(matriz: vector [1..3,1..3] de natural ; columna: natural )

fun generar-ensayo(cuadrado: tipoCuadrado, i,j: natural ;n: natural )
   var nodo: tipoCuadrado;
  nodo ← cuadrado;
  nodo.cuadrícula[i,j] ← n;
  nodo.num-ocupadas ← nodo.num-ocupadas +1;
  nodo.candidatos ← nodo.candidatos - {n};
    dev nodo
ffun
```

```
fun compleciones(cuadrado)
 lista ← lista'vacía;
  si condiciones de poda(cuadrado) entonces
    para n ← 1 hasta 9 hacer
     para i ← 1 hasta 3 hacer
       para j ← 1 hasta 3 hacer
         si n en cuadrado.disponibles ∧
            cuadrado[i,j] = VACIO ∧
             parcialmente-valido(cuadrado.cuadrícula)
          entonces
            w ← generar-ensayo(cuadrado,i);
            lista ← insertar(lista,w);
         fsi
       fpara
     fpara
    fpara
  fsi
ffun
```

1.6.2 Problema del cuadrado latino

Problema 10: En este caso, se dispone de n colores y un tablero de $n \times n$, el problema es una variación del problema anterior y consiste en pintar el cuadro de modo que los colores de las filas y columnas no se repitan.

Este problema se puede consultar en el siguiente libro:

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.459, *Tablero* Fig. p.459, *Tablero*	Implementación p. 460-1 *Todas las soluciones Recursivo*

14.4 Un cuadrado latino de tamaño n es un tablero de $n \times n$ posiciones, cada una de las cuales está pintada de un color escogido entre n diferentes. Ha de cumplir las restricciones de que no puede haber colores repetidos en ninguna de sus filas ni en ninguna de sus columnas. La Figura 14.3 muestra un cuadrado latino de tamaño 4. Dados n colores diferentes:

(a) Escribir un algoritmo que imprima todos los cuadrados latinos posibles de tamaño n.

(b) Considerando que una solución es equivalente a todas las que se generen a partir de ella simplemente mediante la permutación de los colores, modificar el algoritmo del apartado anterior para que solo se obtenga un representante de cada clase de equivalencia.

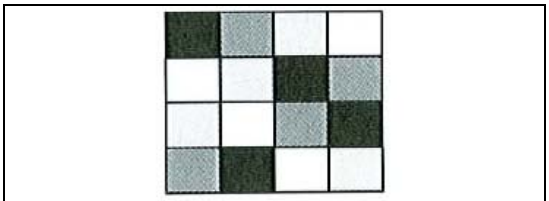


Figura 14.3: Cuadrado latino de tamaño 4.

Ilustración 32, tablero solución cuadrado latino

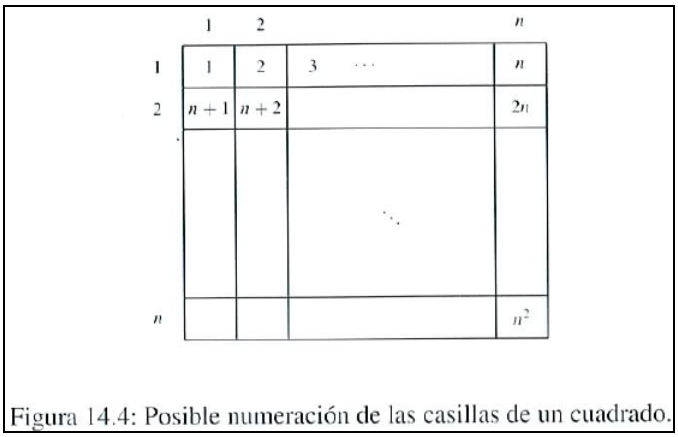


Figura 14.4: Posible numeración de las casillas de un cuadrado.

Ilustración 33, numeración de celdas del cuadrado latino

Apartado (a)

$F[i, color]$ $\iff$ en la fila i hemos usado el color $color$,
$C[j, color]$ $\iff$ en la columna j hemos usado el color $color$.

```
proc latino2(e n : nat+)
var sol[1..n, 1..n] de 1..n, F[1..n, 1..n], C[1..n, 1..n] de bool
    F[1..n, 1..n] := [falso] ; C[1..n, 1..n] := [falso]
    latino-va2(sol, 1, 1, F, C)
fproc
proc latino1(e n : nat+)
var sol[1..n²] de 1..n, F[1..n, 1..n], C[1..n, 1..n] de bool
    F[1..n, 1..n] := [falso] ; C[1..n, 1..n] := [falso]
    latino-va1(sol, 1, F, C)
fproc
```

```
proc latino-va1(sol[1..n^2] de 1..n, e k : 1..n, F[1..n, 1..n], C[1..n, 1..n] de bool)
    fila := fila(k) ;  columna := columna(k)
    para color = 1 hasta n hacer
        si ¬F[fila, color] ∧ ¬C[columna, color] entonces
            sol[k] := color
            F[fila, color] := cierto ;  C[columna, color] := cierto     { marcar }
            si k = n^2 entonces imprimir(sol)
            si no latino-va1(sol, k + 1, F, C)
            fsi
            F[fila, color] := falso ;  C[columna, color] := falso     { desmarcar }
        fsi
    fpara
fproc
proc latino-va2(sol[1..n, 1..n] de 1..n, e i, j : 1..n, F[1..n, 1..n], C[1..n, 1..n] de bool)
    para color = 1 hasta n hacer
        si ¬F[i, color] ∧ ¬C[j, color] entonces
            sol[i, j] := color
            F[i, color] := cierto ;  C[j, color] := cierto     { marcar }
            casos
                  i = n ∧ j = n  →  imprimir(sol)
                □ i < n ∧ j = n  →  latino-va2(sol, i + 1, 1, F, C)
                □ i ≤ n ∧ j < n  →  latino-va2(sol, i, j + 1, F, C)
            fcasos
            F[i, color] := falso ;  C[j, color] := falso     { desmarcar }
        fsi
    fpara
fproc
```

Apartado (b)

```
proc latino-sin-equivalencias(e n : nat+)
var sol[1..n, 1..n] de 1..n, F[1..n, 1..n], C[1..n, 1..n] de bool
    F[1..n, 1..n] := [falso] ;  C[1..n, 1..n] := [falso]
    para j = 1 hasta n hacer
        sol[1, j] := j
        F[1, j] := cierto ;  C[j, j] := cierto
    fpara
    latino-va2(sol, 2, 1, F, C)
fproc
```

1.6.3 Problema del sudoku

Problema 11: Este problema es una variación de los problemas anteriores y consiste en rellenar una cuadricula de n^2xn^2 celdas divida en subcuadrículas de nxn con las cifras del 1 al n^2 partiendo de algunos números ya dispuestos en alguna de las celdas.

Libro	Capítulo/apartado	Visualización	Implementación
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	Fig. p.239 *Tablero*	Implementación p. 239-41 *Recursivo*

1. 7.3 Sudoku

What is Sudoku? In its most. common form, it consists of a 9 x 9 grild filled with blanks and the digits 1 to 9. The puzzle is completed when every row, column, and sector (3 x 3 subproblems corresponding to the nine sectors of a tic-tac-toe puzzle) contain the digits I through 9 with no deletions or repetition. Figure 7.2 presents a challenging Sudoku puzzle and its solution.

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

Figure 7.2: Challenging Sudoku puzzle (l) with solution (r)

Ilustración 34, ejemplo y solución de sudoku

```
#define DIMENSION 9
#define NCELLS  DIMENSION*DIMENSIO
typedef struct {
    int x, y;
} point;
```

```
typedef struct {
    int m[DIMENSION+1][DIMENSION
    int freecount;
    point move[NCELLS+1];
} boardtype;
```

```
make_move(int a[], int k, boardtype *board)
{
  fill_square(board->move[k].x,board->move[k].y,a[k],boar
}
process_solution(int a[], int k, boardtype *board)
{
  print_board(board);
  finished = TRUE;
}
unmake_move(int a[], int k, boardtype *board)
{
  free_square(board->move[k].x,board->move[k].y,board);
}

 construct_candidates(int a[], int k, boardtype *board, int c[],
                      int *ncandidates)
 {
    int x,y;
    int i;
    bool possible[DIMENSION+1];
    next_square(&x,&y,board);
    board->move[k].x = x;
    board->move[k].y = y;
    *ncandidates = 0;
   if ((x<0) && (y<0)) return;
   possible_values(x,y,board,possible);
   for (i=0; i<=DIMENSION; i++)
          if (possible[i] == TRUE) {
                  c[*ncandidates] = i;
                  *ncandidates = *ncandidates + 1;
          }
 }
 is_a_solution(int a[], int k, boardtype *board)
 {
   if (board->freecount == 0)
          return (TRUE);
   else
          return(FALSE);
 }
```

1.6.4 Problema del dominó

Problema 12: El problema consiste en generar todas las cadenas válidas con las 28 fichas del dominó siguiendo las reglas del juego.

El listado de libros donde se ha encontrado el problema es:

Libro	Capítulo/apartado	Visualización	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resuelto*	Capítulo 4	Fig. p.81 *Dominó*	Pseudocódigo p. 82-4 *Todas las soluciones Recursivo*

4.2 Cadena de fichas del Dominó
Las 28 fichas de dominó son de la forma (i,j) con $i,j = 0\ldots6$. Una ficha de dominó puede colocarse a continuación de la anterior si coinciden los valores de los extremes que se tocan. Por ejemplo, a continuación de la ficha (1,2) puede colocarse la (2,4). Diséñese un algoritmo que produzca todas las cadenas permisibles que contengan todas las fichas del dominó.

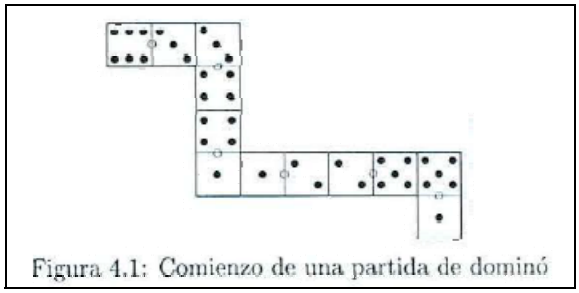


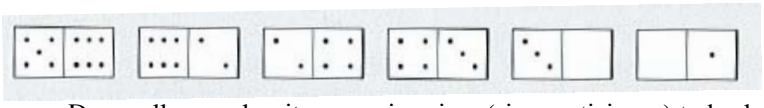
Figura 4.1: Comienzo de una partida de dominó

Ilustración 35, ejemplo de partida del dominó

```
juego =  tupla
   caja:  arreglo [0..6, 0..6] de  bool
   cadena:  arreglo [1..28] de ficha
   ultima:  entero
ftupla

fun domino (juego)
  si válido (juego)
  entonces   dev juego.cadena
  si no
  lista-c ← compleciones(juego)
    mientras ¬ vacia(lista-c)  hacer
    hijo ← primero(lista-c)
    lista-c ← resto(lista-c)
      si condiciones de poda(hijo)
      entonces vuelta-atrás(hijo)
      fsi
    fmientras
  fsi
ffun

fun compleciones (juego)  dev lista-c: lista
 lista-c ← lista-vacía
 ultima-ficha ← juego.cadena[última]
 j ← ultima-ficha.j
   para i ← 0  hasta 6  hacer
     si caja[j,i] = cierto  entonces
     juego-nuevo ← juego
     ficha ← crear-ficha(j,i)
     juego-nuevo.caja[i,j] ←  falso
     juego-nuevo.caja[j,i] ←  falso
     juego-nuevo.cadena[j+1] ← ficha
     juego-nuevo.ultima ← juego-nuevo.ultima + 1

     lista-c ← añadir(juego,lista-c)
     fsi
   fpara
   dev lista-c
ffun
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.467 *Árbol de búsqueda*	Implementación p. 468 *Todas las soluciones* *Recursivo*

14.9. El juego usual del dominó tiene 28 fichas diferentes. Cada ficha es rectangular y tiene grabado en cada extremo un número de puntos entre 0 y 6. Siguiendo las reglas del juego, las fichas se colocan formando una cadena de tal manera que cada par de fichas consecutivas tienen iguales los números correspondientes a los dos extremos que se tocan. El siguiente diagrama muestra parte de un ejemplo de cadena de dominós.



Desarrollar un algoritmo que imprima (sin repeticiones) todas las cadenas circulares correctas de 28 fichas.

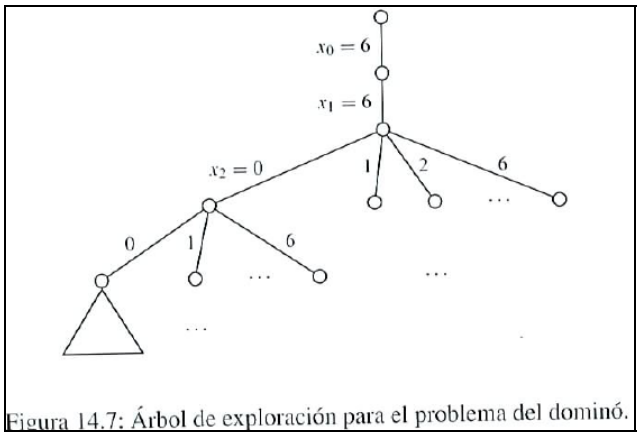


Figura 14.7: Árbol de exploración para el problema del dominó.

Ilustración 36, árbol de búsqueda potencial abreviado (dominó)

```
proc dominó()
var sol[0..28] de 0..6, usada[0..6, 0..6] de bool
    sol[0] := 6 ;  sol[1] := 6
    usada[0..6, 0..6] := [falso] ;  usada[6, 6] := cierto
    dominó-va(sol, 2, usada)
fproc

proc dominó-va(sol[0..28] de 0..6, e k : 2..28, usada[0..6, 0..6] de bool)
   para j = 0 hasta 6 hacer
      si ¬usada[sol[k − 1], j] entonces
         sol[k] := j
         { marcar }
         usada[sol[k − 1], j] := cierto ;  usada[j, sol[k − 1]] := cierto
         si k = 28 entonces
            si sol[28] = sol[0] entonces imprimir(sol) fsi
         si no dominó-va(sol, k + 1, usada)
         fsi
         { desmarcar }
         usada[sol[k − 1], j] := falso ;  usada[j, sol[k − 1]] := falso
      fsi
   fpara
fproc
```

En la siguiente tabla se adjunta un resumen de los libros donde se puede consultar el problema del dominó.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Fig. p.81 *Dominó*	Cadena de fichas del Domino	Pseudocódigo p. 82-4 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.467 *Árbol de búsqueda*	14.9 (problema del dominó)	Implementación p. 468 Todas las soluciones Recursivo

1.6.5 Tabla resumen sobre problemas de juegos

A continuación se muestra una tabla que incluye un resumen de los libros donde el lector puede consultar los distintos problemas sobre juegos mostrados anteriormente.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Fig. p.92 *Cuadrado mágico*	Cuadrado Mágico de suma 15	Pseudocódigo p. 92-3 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.459 *Tablero*	14.4 (cuadrado latino*)*	Implementación p. 460-1 *Todas las soluciones* *Recursivo*
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	Fig. p.239 *Tablero*	Sudoku	Implementación p. 239-41 *Recursivo*
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Fig. p.81 *Dominó*	Cadena de fichas del Dominó	Pseudocódigo p. 82-4 *Todas las soluciones* *Recursivo*

N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.467 *Árbol de búsqueda*	14.9 (problema del dominó)	Implementación p. 468 *Todas las soluciones Recursivo*

1.7 Problemas de subconjuntos

Problema 13: En este caso se pueden dar dos tipos de problemas, por un lado, se puede querer obtener todos los subconjuntos de un conjunto dado de elementos o se puede querer encontrar aquellos subconjuntos que sumen una cantidad dada.

Libro	Capítulo/ apartado	Visualización	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.327-8,344 *Árbol de búsqueda*	Implementación p. 342 *Todas las soluciones Recursivo*

7.3 SUM OF SUBSETS

Suppose we are given *n* distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sum is *M*. This is called the sum of subsets problem.

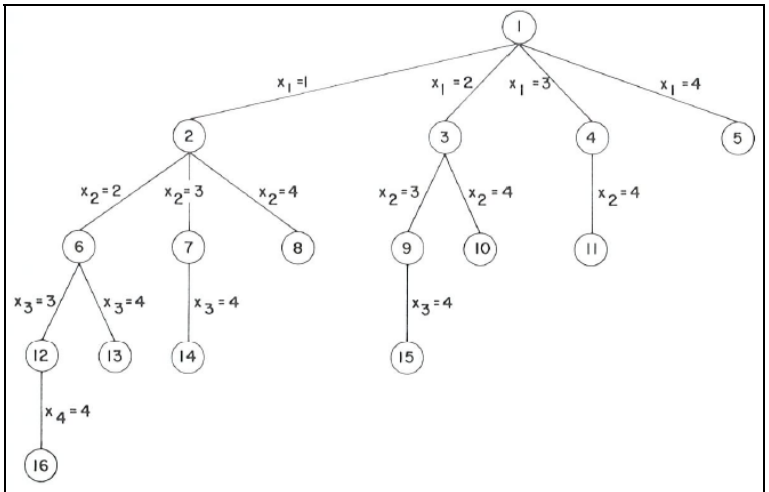


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth first search.

Ilustración 37, árbol de búsqueda potencial (subconjuntos)

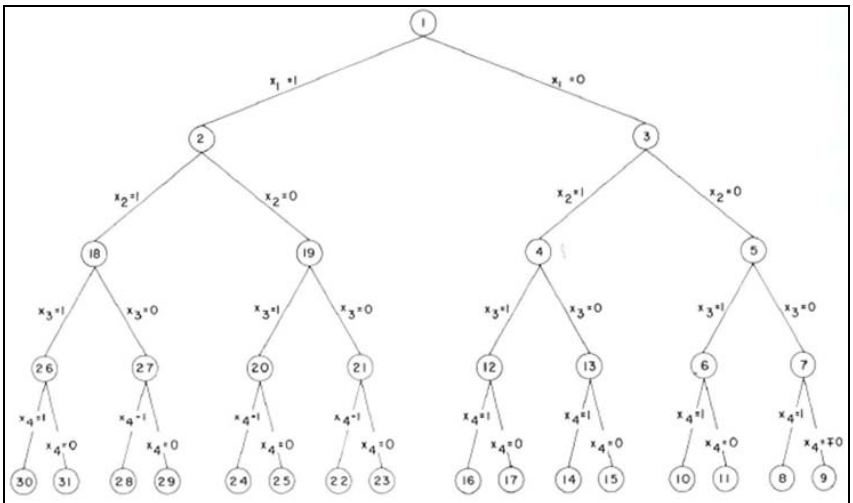


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

Ilustración 38, árbol de búsqueda potencial (subconjuntos)

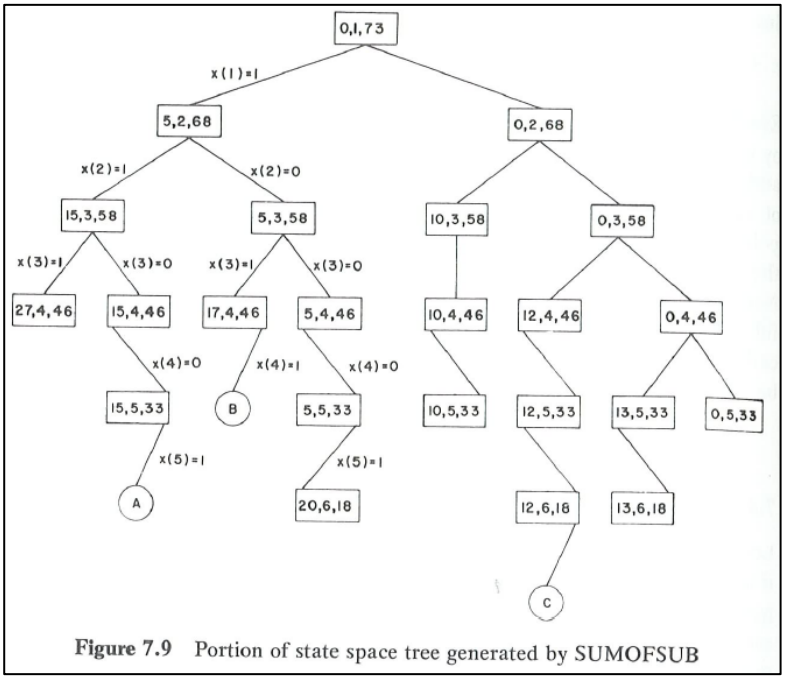


Figure 7.9 Portion of state space tree generated by SUMOFSUB

Ilustración 39, árbol de búsqueda generado (subconjuntos)

```
procedure SUMOFSUB(s, k, r)
 //find all subsets of W(1:n) that sum to M. The values of//
 //X(j), 1 ≤ j < k have already been determined. s = Σ_{j=1}^{k-1} W(j)X(j)//
 //and r = Σ_{j=k}^{n} W(j) The W(j)s are in nondecreasing order.//
 //It is assumed that W(1) ≤ M and Σ_{i=1}^{n} W(i) ≥ M.//
1  global integer M, n; global real W(1:n); global boolean X(1:n)
2  real r, s; integer k, j
   //generate left child. Note that s + W(k) ≤ M because B_{k-1} = true//
3  X(k) ← 1
4  if s + W(k) = M   //subset found//
5       then print (X(j), j ← 1 to k)
        //there is no recursive call here as W(j) > 0, 1 ≤ j ≤ n//
6       else
7         if s + W(k) + W(k + 1) ≤ M then   //B_k = true//
8              call SUMOFSUB(s + W(k), k + 1, r - W(k))
9         endif
10 endif
   //generate right child and evaluate B_k//
11 if s + r - W(k) ≥ M and s + W(k + 1) ≤ M   //B_k = true//
12      then X(k) ← 0
13           call SUMOFSUB(s, k + 1, r - W(k))
14 endif
15  end SUMOFSUB
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

Libro	Capítulo/apartado	Visualización	Implementación
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figuras p. 195,197	Implementación p. 198 *Todas las soluciones Recursivo*

5.4 THE SUM OF SUBSETS PROBLEM

Specifically, in the Sum-of-Subsets Problem, there are *n* positive integers (weights) *w*, and a positive integer W. The goal is to find all subsets of the integers that sum to W. As mentioned earlier, we usually state our problems so as to find all solutions. For the purposes of the thief's application, however, only one solution need be found.

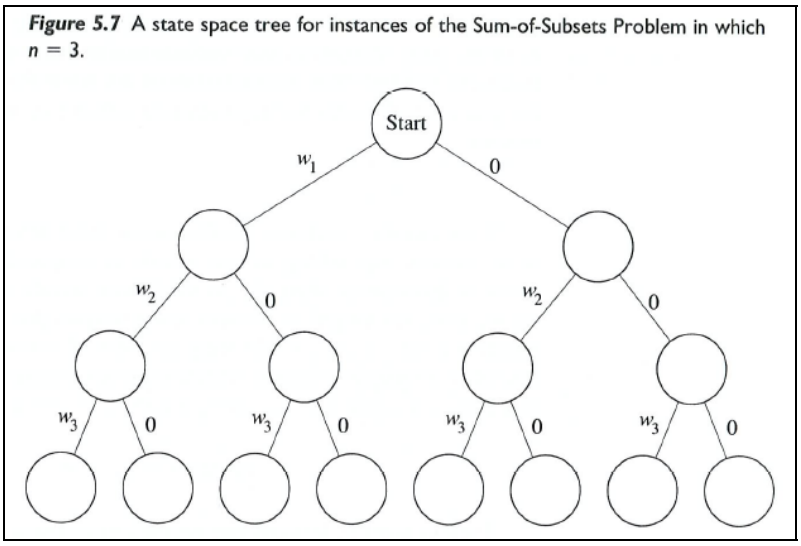


Ilustración 40, árbol de búsqueda potencial (subconjuntos)

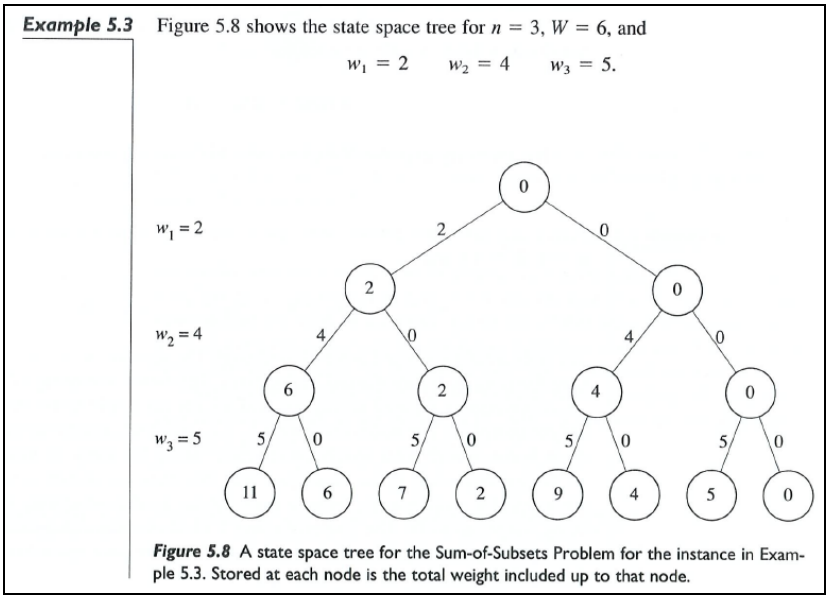


Ilustración 41, árbol de búsqueda potencial (subconjuntos)

Figure 5.9 The pruned state space tree produced using backtracking in Example 5.4. Stored at each node is the total weight included up to that node. The only solution is found at the shaded node. Each nonpromising node is marked with a cross.

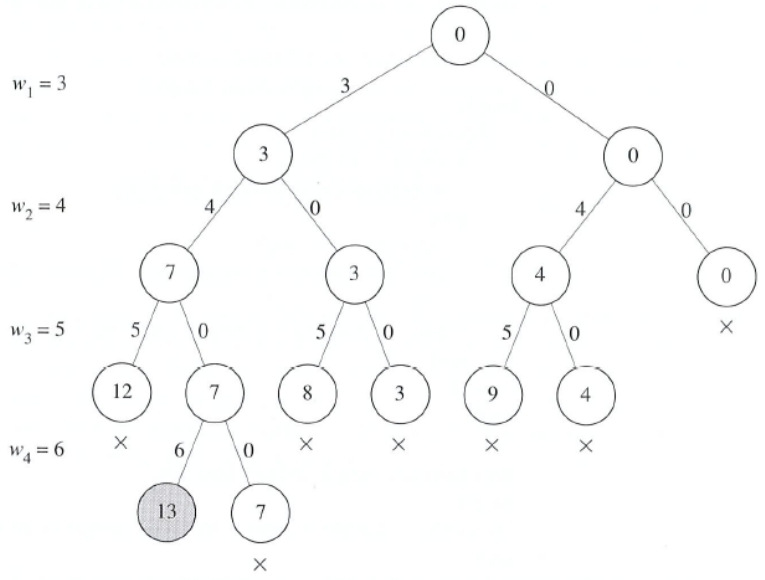


Ilustración 42, árbol de búsqueda generado (subconjuntos)

```
Inputs: positive integer n, array w containing n positive integers sorted in non-
decreasing order, and positive integer W.
Outputs: all combinations of the integers that sum to W.
procedure sum_of_subsets (i: index;
                          weight, total: integer);
begin
  if promising(i) then
    if weight = W then
      write(include[1] through include[i])
    else
      include[i+1]:= 'yes';                          {Include w[i+1].}
      sum_of_subsets(i+1, weight+w[i+1], total-w[i+1]);
      include[i+1]:= 'no';                           {Do not include}
      sum_of_subsets(i+1, weight, total-w[i+1])      {w[i+1].}
    end
  end
end;
function promising(i: index): boolean;
begin
 promising:= (weight + total ≥ W) and (weight = W or weight + w[i+1] ≤ W)
end;
```

Libro	Capítulo/ apartado	Visualización	Implementación
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	-	Implementación p. 234 *Todas las soluciones Recursivo*

7.1.1 Constructing All Subsets

A critical issue when designing state spaces to represent combinatorial objects is how many objects need representing. How many subsets arc there of a *n*-element set, say the integers {l,...,*n*}? There are exactly two subsets for *n*=1, namely {} and {1}.

```
generate_subsets(int n)
{
    int a[NMAX];

    backtrack(a,0,n);
}
process_solution(int a[], int k)
{
    int i;  /* counter */

    printf("{");
    for (i=1; i<=k; i++)
            if (a[i] == TRUE) printf(" %d",i);

    printf(" }\n");
}
construct_candidates(int a[], int k, int n, int c[], int *ncandida
{
    c[0] = TRUE;
    c[1] = FALSE;
    *ncandidates = 2;
}
```

```
is_a_solution(int a[], int k, int n)
{
        return (k == n);
}
```

Libro	Capítulo/ apartado	Visualización	Implementación
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	-	Implementación p. 845 *Todas las soluciones Recursivo*

21.2.1 Container Loading

In Section 18.3.1 we considered the problem of loading a ship with the maximum number of containers. Now we will consider a variant of this problem in which we have two ships and n containers. The capacity of the first ship is c_1, and that of the second c_2. w_i is the weight of container i, and we wish to determine whether there is a way to load all n containers. In case there is, then such a loading is to be determined.

$$\sum_{i=1}^{n} w_i \leq c_1 + c_2$$

```
public class RecursiveBTLoading1
{
  // class data members
  static int numberOfContainers;
  static int [] weight;
  static int capacity;
  static int weightOfCurrentLoading;
  static int maxWeightSoFar;
  public static int maxLoading(int [] theWeight, int theCapacity)
  {
     // set class data members
     numberOfContainers = theWeight.length - 1;
     weight = theWeight;
     capacity = theCapacity;
     weightOfCurrentLoading = 0;
     maxWeightSoFar = 0;
     // compute weight of max loading
     rLoad(1);
     return maxWeightSoFar;
  }
  /** actual method to find max loading */
  private static void rLoad(int currentLevel)
  {// search from a node at currentLevel
     if (currentLevel > numberOfContainers)
     {// at a leaf
        if (weightOfCurrentLoading > maxWeightSoFar)
           maxWeightSoFar = weightOfCurrentLoading;
        return;
     }
     // not at a leaf, check subtrees
     if (weightOfCurrentLoading + weight[currentLevel] <= capacity)
     {// try left subtree; i.e., x[currentLevel] = 1
        weightOfCurrentLoading += weight[currentLevel];
        rLoad(currentLevel + 1);
        weightOfCurrentLoading -= weight[currentLevel];
     }
     rLoad(currentLevel + 1);  // try right subtree
  }
}
```

Program 21.1 First backtracking code for the loading problem

1.7.1 Tabla resumen sobre problemas de subconjuntos

En la siguiente tabla se puede ver un resumen de los libros donde se puede consultar distintos problemas sobre subconjuntos.

Libro	Capítulo/apartado	Visualización	Nomenclatura	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Fig. p.327-8,344 *Árbol de búsqueda*	SUM OF SUBSETS	Implementación p. 342 *Todas las soluciones Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figuras p. 195,197	THE SUM-OF-SUBSETS PROBLEM	Implementación p. 198 *Todas las soluciones Recursivo*
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	-	Constructing All Subsets	Implementación p. 234 *Todas las soluciones Recursivo*
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	-	Container Loading	Implementación p. 845 *Todas las soluciones Recursivo*

1.8 Otros problemas de decisión

A continuación se muestran otros problemas de decisión

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 462 *Árbol de búsqueda*	Implementación p. 462 *Recursivo*

14.5 La República de Fanfanisflán acaba de abrir *n* factorías de tecnología punta, que requieren una instalación informática y personal que la dirija. El Gobierno Supremo les ha asignado *n* máquinas y otros tantos técnicos recién licenciados, que deberán ser distribuidos entre las fábricas. Como no todas las máquinas son apropiadas para las necesidades de todas las fábricas, se dispone de una función booleana apropiada?(m, *f)* que indica si la máquina *m* es apropiada para la fábrica *f.* Asimismo, no todos los técnicos conocen todas las máquinas, por lo que se dispone de la correspondiente función conoce?(*t,m*) que indica si el técnico *i* conoce la máquina *m.* Finalmente, no todos los técnicos están dispuestos a trabajar en todas las fábricas, ya que algunas de ellas están relacionadas con la industria bélica, por lo que se dispone de la función aceptaría?(*t,f*) que indica si el técnico *t* estaría dispuesto a trabajar para la fábrica *f .* Se pide encontrar, si la hubiera, alguna forma factible de asignar a cada fábrica una máquina apropiada a sus necesidades, y un técnico que conozca dicha máquina y acepte trabajar para la fábrica en cuestión.

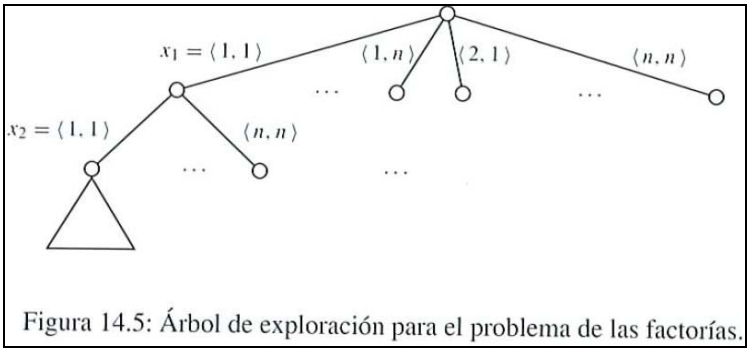


Figura 14.5: Árbol de exploración para el problema de las factorías.

Ilustración 43, árbol de búsqueda potencial, (factorías)

```
fun aceptable?(f, m, t : 1..n, usada-máq[1..n], usado-téc[1..n] de bool) dev respuesta : bool
    respuesta := ¬usada-máq[m] ∧ ¬usado-téc[t]
                 ∧ apropiada?(m, f) ∧ conoce?(t, m) ∧ aceptaría?(t, f)
ffun
fun fábricas(n : nat+) dev (éxito : bool, sol[1..n] de pareja)
var usada-máq[1..n], usado-téc[1..n] de bool
    usada-máq[1..n] := [falso] ; usado-téc[1..n] := [falso]
    éxito := falso
    fábricas-va1(sol, 1, usada-máq, usado-téc, éxito)
ffun
```

```
tipos
    pareja  =  reg
                  máquina : 1..n
                  técnico : 1..n
               freg
ftipos

proc fábricas-va1(sol[1..n] de pareja, e k : 1..n, usada-máq[1..n], usado-téc[1..n] de bool, éxito : bool)
    m := 1     { recorre las máquinas }
    mientras m ≤ n ∧ ¬éxito hacer
        t := 1     { recorre los técnicos }
        mientras t ≤ n ∧ ¬éxito hacer
            si aceptable?(k, m, t, usada-máq, usado-téc) entonces
                sol[k].máquina := m ;  sol[k].técnico := t
                usada-máq[m] := cierto ;  usado-téc[t] := cierto     { marcar }
                si k = n entonces éxito := cierto
                si no fábricas-va1(sol, k + 1, usada-máq, usado-téc, éxito)
                fsi
                usada-máq[m] := falso ;  usado-téc[t] := falso     { desmarcar }
            fsi
            t := t + 1
        fmientras
        m := m + 1
    fmientras
fproc
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 494-5 *Árbol de búsqueda*	Implementación p. 494-5 *Recursivo*

14.21 Juanito el Explorador está pasando unas vacaciones en Tombuctú. Juanito disfruta mandando postales de los exóticos lugares que visita, para que sus amigos rabien de envidia cuando las reciban. A tal efecto se ha pasado por la oficina e correos más cercana y se ha hecho con un lote de sellos de *n* valores diferentes, disponiendo de tres sellos de cada valor. En correos también le han informado de las diferentes tarifas que de franqueo de tarjetas postales, y le han explicado que en la esquina superior derecha de cada postal aparece un bloque remarcado destinado a su franqueo. Dicho bloque está dividido en cinco casillas, destinadas a un sello, de forma que un franqueo solo es admisible si se alcanza la tarifa correspondiente y se cubren exactamente las cinco casillas.

(a) Escribir un algoritmo para generar todas las formas admisibles de franquear una postal de tarifa T si entendemos que el orden en el que aparecen los sellos en las casillas es significativo.

(b) Escribir ahora un algoritmo para generar todas las formas admisibles de franquear una postal de tarifa T si el orden de los sellos no es significativo.

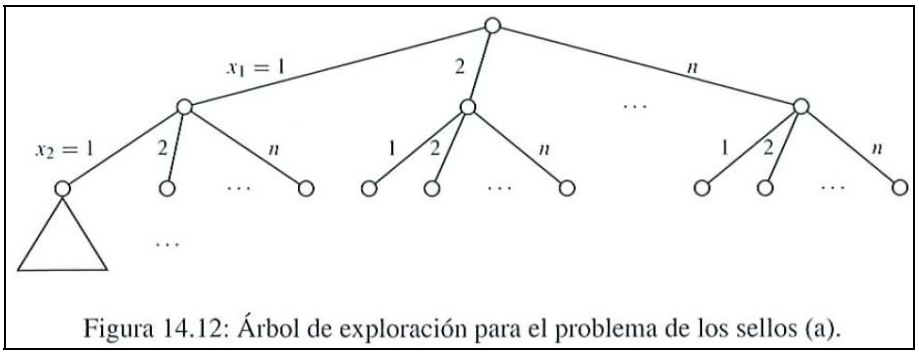


Figura 14.12: Árbol de exploración para el problema de los sellos (a).

Ilustración 44, árbol de búsqueda potencial (sellos)

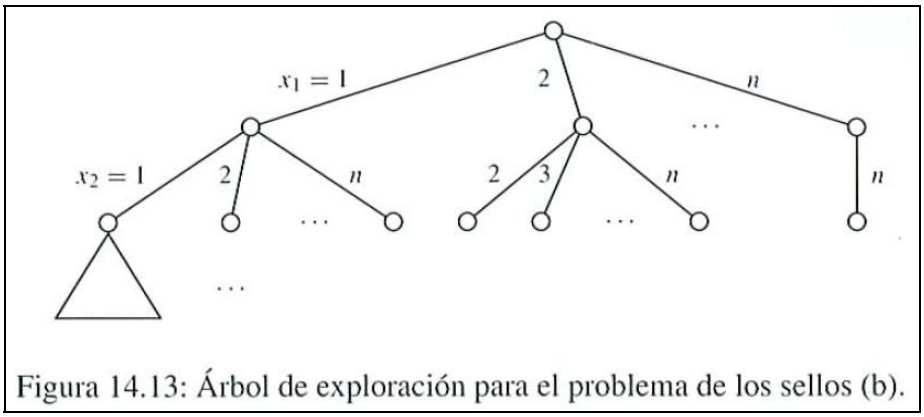


Figura 14.13: Árbol de exploración para el problema de los sellos (b).

Ilustración 45, árbol de búsqueda potencial (sellos)

Apartado (a)

```
proc sellos-va1(e V[1..n] de real+, e T : real+, sol[1..5] de 1..n, e k : 1..5, coste : real,
                usados[1..n] de 0..3)
    para s = 1 hasta n hacer
        si usados[s] < 3 entonces
            sol[k] := s
            usados[s] := usados[s] + 1 ;  coste := coste + V[s]     { marcar }
            si k = 5 entonces
                si coste ≥ T entonces imprimir(sol) fsi
            si no sellos-va1(V, T, sol, k + 1, coste, usados)
            fsi
            usados[s] := usados[s] − 1 ;  coste := coste − V[s]     { desmarcar }
        fsi
    fpara
fproc
proc sellos1(e V[1..n] de real+, e T : real+)
var sol[1..5] de 1..n, usados[1..n] de 0..3
    suma-todos := 0
    para i = 1 hasta n hacer suma-todos := suma-todos + 3 * V[i] fpara
    si 3 * n ≥ 5 ∧ suma-todos ≥ T entonces
        coste := 0 ;  usados[1..n] := [0]
        sellos-va1(V, T, sol, 1, coste, usados)
    fsi
fproc
```

Apartado (b)

```
fun factible?(T : real, k : 1..5, último : 1..n, usados : 0..3, coste, suma-libres : real)
              dev respuesta : bool
    { hay sellos para cubrir los huecos libres }
    respuesta := (3 * (n − último) + 3 − usados) ≥ (5 − k)
    { y se puede cubrir la cantidad pendiente }
    respuesta := respuesta ∧ suma-libres ≥ (T − coste)
ffun
proc sellos2(e V[1..n] de real+, e T : real+)
var sol[1..5] de nat
    suma-todos := 0
    para i = 1 hasta n hacer suma-todos := suma-todos + 3 * V[i] fpara
    si 3 * n ≥ 5 ∧ suma-todos ≥ T entonces
        último := 1 ;  usados := 0 ;  coste := 0
        sellos-va2(V, T, sol, 1, último, usados, coste, suma-todos)
    fsi
fproc
```

```
proc sellos-va2(e V[1..n] de real+, e T : real+, sol[1..5] de 1..n, e k : 1..5, último : 1..n
                usados : 0..3, coste, suma-libres : real)
    { probamos un sello de tipo último }
    si usados < 3 entonces
        sol[k] := último
        usados := usados + 1 ;  coste := coste + V[último]     { marcar }
        suma-libres := suma-libres − V[último]
        si k = 5 entonces
            si coste ≥ T entonces imprimir(sol) fsi
        si no
            si factible?(T, k, último, usados, coste, suma-libres) entonces
                sellos-va2(V, T, sol, k + 1, último, usados, coste, suma-libres)
            fsi
        fsi
        usados := usados − 1 ;  coste := coste − V[último]     { desmarcar }
        suma-libres := suma-libres + V[último]
    fsi
    { descartamos los sellos no utilizados de tipo último }
    suma-libres-act := suma-libres − (3 − usados) ∗ V[último]
    { probamos con el resto de tipos de sellos }
    usados-act := 1
    para s = último + 1 hasta n hacer
        sol[k] := s
        coste := coste + V[s]     { marcar }
        suma-libres-act := suma-libres-act − V[s]     { quitamos un sello de tipo s }
        si k = 5 entonces
            si coste ≥ T entonces imprimir(sol) fsi
        si no
            si factible?(T, k, s, usados-act, coste, suma-libres-act) entonces
                sellos-va2(V, T, sol, k + 1, s, usados-act, coste, suma-libres-act)
            fsi
        fsi
        suma-libres-act := suma-libres-act + V[s]
        coste := coste − V[s]     { desmarcar }
        { quitamos los dos sellos restantes de tipo s }
        suma-libres-act := suma-libres-act − 2 ∗ V[s]
    fpara
fproc
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.471 *Árbol de búsqueda*	Implementación p. 470-2 *Recursivo*

14.11 El Maqui y el Popeye acaban de desvalijar la reserva de oro. Los lingotes están empaquetados en n cajas de diferentes pesos (reales) y, como no tienen tiempo de desempaquetarlos para dividir el botín, deciden basarse en los pesos de las cajas para intentar distribuir el botín a medias. Al cabo de un buen rato todavía no han conseguido repartirse el botín, por lo cual acuden al Teclas para saber si el botín se puede dividir en dos partes iguales sin desempaquetar ninguna de las cajas.

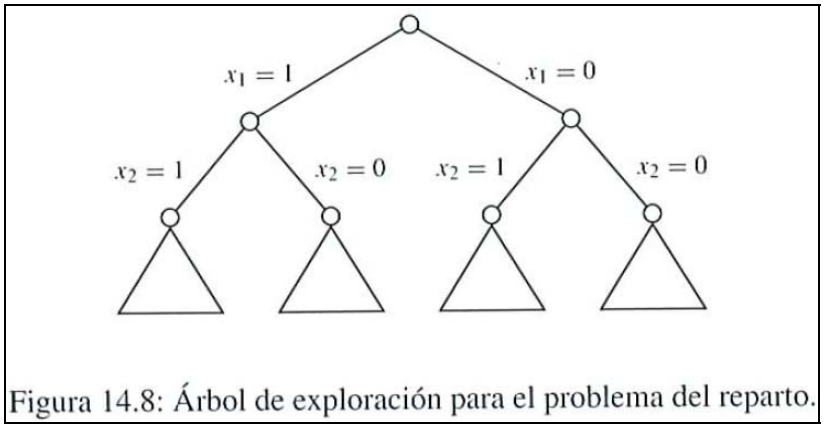


Figura 14.8: Árbol de exploración para el problema del reparto.

Ilustración 46, árbol de búsqueda potencial (reparto)

```
fun repartir-botín-va(P[1..n] de real+) dev ⟨éxito : bool, sol[1..n] de 0..1⟩
    peso := 0
    para i = 1 hasta n hacer peso := peso + P[i] fpara     { peso = Σ_{i=1}^{n} P[i] }
    suma := 0 ; resto := peso
    éxito := falso
    conseguir-cantidad2(P, peso/2, sol, 1, suma, resto, éxito)
ffun

proc conseguir-cantidad1(e P[1..n] de real+, e C : real, sol[1..n] de 0..1, e k : 1..n, suma : real,
                         éxito : bool)
   { hijo izquierdo — añadir peso }
   si suma + P[k] ≤ C entonces
      sol[k] := 1
      suma := suma + P[k]     { marcar }
```

```
        si suma = C entonces
            éxito := cierto ;  sol[k + 1..n] := [0]
        si no
            si k < n ∧ suma < C entonces
                conseguir-cantidad1(P, C, sol, k + 1, suma, éxito)
            fsi
        fsi
        suma := suma − P[k]     { desmarcar }
    fsi
    { hijo derecho — no añadir peso, no puede generar solución }
    si ¬éxito ∧ k < n entonces
        sol[k] := 0
        conseguir-cantidad1(P, C, sol, k + 1, suma, éxito)
    fsi
fproc
```

2 Problemas de optimización

En este apartado se verá un gran número de problemas distintos que buscan una solución óptima en base a una función objetivo.

2.1 Problema de la Mochila 0/1

Problema 16: Se dispone de n objetos no fraccionables de pesos p_i y beneficios b_i. El peso máximo que puede llevar la mochila es C. En general, el problema consiste en llenar la mochila con objetos, de forma que se maximice el beneficio.

Libro	Capítulo/apartado	Visualización	Implementación
G. Brassard y P. Bratley, *Fundamentals of Algorithmics*	Capítulo 9 Apartado 6	Fig. p.307 *Árbol de búsqueda*	Pseudocódigo p. 308 *Todas las soluciones Recursivo*

9.6.1 The knapsack problem (3)
For a first example illustrating the general principle, we return to the knapsack problem described in Section 8.4. Recall that we are given a certain number of objects and a knapsack. This time, however, instead of supposing that we have n objects available, we shall suppose that we have n types of object, and that an adequate number of objects of each type are available. This does not alter the problem in any important way. For $i = 1, 2, \ldots, n$ an object of type i has a positive weight w_i and a positive value v_i. The knapsack can carry a weight not exceeding W. Our aim is to fill the knapsack in a way that maximizes the value of the included objects, while respecting the capacity constraint. We may take an object or to leave it behind, but we may not take a fraction of an object.

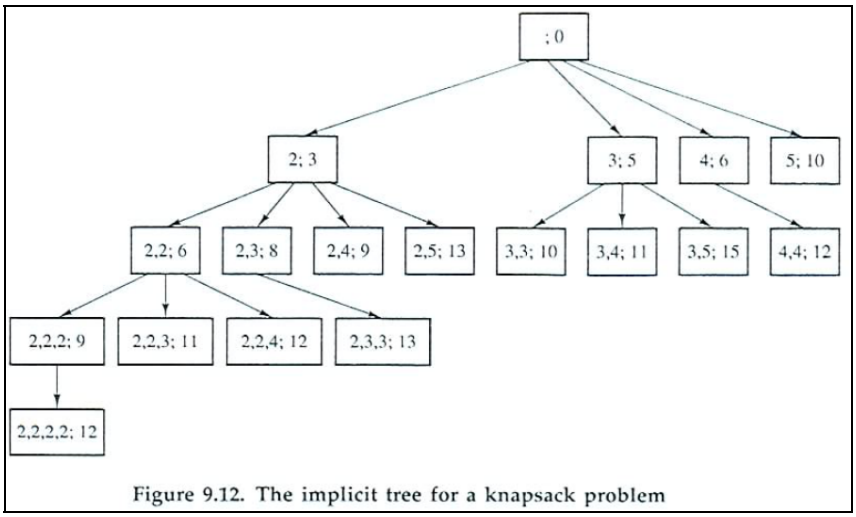


Figure 9.12. The implicit tree for a knapsack problem

Ilustración 47, árbol de búsqueda generado (mochila)

```
function backpack(i, r)
    {Calculates the value of the best load that can
     be constructed using items of types i to n
     and whose total weight does not exceed r}
    b ← 0
    {Try each allowed kind of item in turn}
    for k ← i to n do
        if w[k] ≤ r then
            b ← max(b, v[k]+backpack(k, r − w[k]))
    return b
```

1.

Libro	Capítulo/apartado	Visualización	Implementación
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figuras p. 354, 358	Implementación p. 352, 355 *Iterativo*

7.6 KNAPSACK PROBLEM

Given n positive weights w_i, n positive profits p_i, and a positive number M which is the knapsack capacity, this problem calls for choosing a subset of the weights such that $\sum_{1 \le i \le n} w_i x_i \le M$ and $\sum_{1 \le i \le n} p_i x_i$ is maximized (7.2).

The x's constitute a zero-one valued vector.

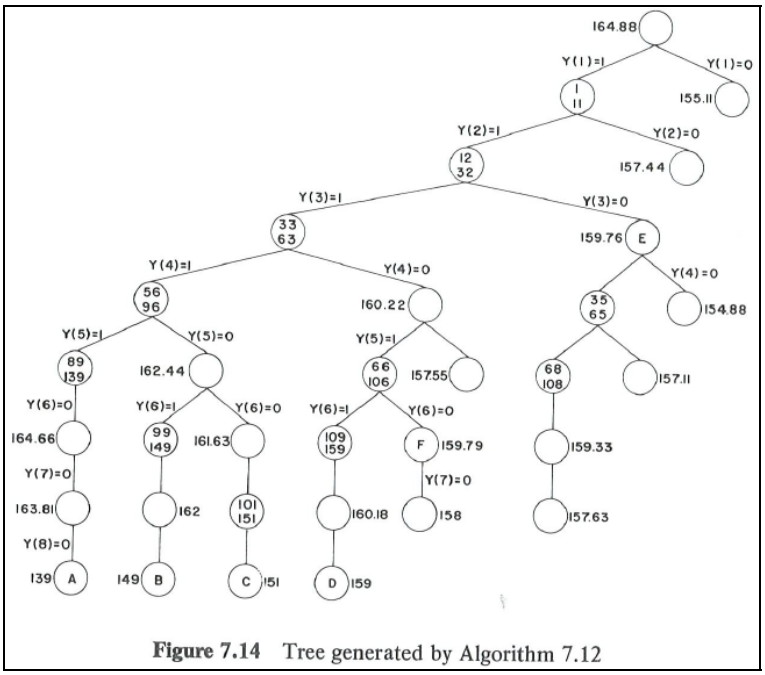


Figure 7.14 Tree generated by Algorithm 7.12

Ilustración 48, árbol de búsqueda generado (mochila)

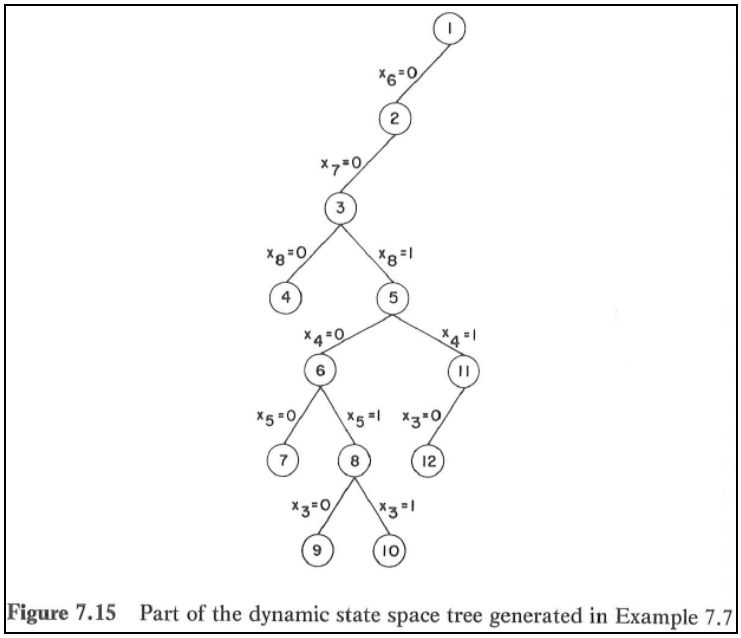


Figure 7.15 Part of the dynamic state space tree generated in Example 7.7

Ilustración 49, árbol de búsqueda generado (mochila)

```
procedure BKNAP2(M, n, W, P, fw, fp, X)
//same as BKNAP1//
integer n, k, Y(1:n), i, j, X(1:n)
real W(1:n), P(1:n), M, fw, fp, pp, ww, cw, cp
cw ← cp ← k ← 0; fp ← -1
loop
  while BOUND1(cp, cw, k, M, pp, ww, j) ≤ fp do
      while k ≠ 0 and Y(k) ≠ 1 do
          k ← k - 1
      repeat
      if k = 0 then return endif
      Y(k) ← 0; cw ← cw - W(k); cp ← cp - P(k)
  repeat
  cp ← pp; cw ← ww; k ← j    //equivalent to loop of lines 4-6 in//
                                                    //BKNAP1//
  if k > n then fp ← cp; fw ← cw; k ← n; X ← Y
           else Y(k) ← 0
  endif
repeat
end BKNAP2
```

Algorithm 7.14 Modified knapsack algorithm

```
procedure BOUND1(p, w, k, M, pp, ww, i)
  //pp and ww are the profit and weight corresponding to the last left//
  //child move. i is the index of the first object that does not fit.//
  //It is n + 1 if no objects remain.//
  global n, P(1:n), W(1:n), Y(1:n)
  integer k, i; real p, w, pp, ww, M, b
  pp ← p; ww ← w
  for i ← k + 1 to n do
    if ww + W(i) ≤ M then ww ← ww + W(i); pp ← pp + P(i); Y(i) ← 1
                     else return (pp + (M - ww)*P(i)/W(i))
    endif
  repeat
  return(pp)
end BOUND1
```

Algorithm 7.13 Generating a bound

```
procedure BKNAP1(M, n, W, P, fw, fp, X)
 //M, the size of the knapsack//
 //n, the number of weights and profits//
 //W(1:n), the weights//
 //P(1:n), the corresponding profits; P(i)/W(i) ≥ P(i + 1)/W(i + 1)//
 //fw, the final weight of the knapsack//
 //fp, the final maximum profit//
 //X(1:n), either zero or one. X(k) = 0 if W(k) is not in the knapsack//
 //else X(k) = 1//
1  integer n, k, Y(1:n), i, X(1:n); real M, W(1:n), P(1:n), fw, fp, cw, cp;
2  cw ← cp ← 0; k ← 1; fp ← −1 //cw = current weight, cp = cur-//
                                                    //rent profit//
3  loop
4    while k ≤ n and cw + W(k) ≤ M do  //place k into knapsack//
5      cw ← cw + W(k); cp ← cp + P(k); Y(k) ← 1; k ← k + 1
     //place W(k) in the knapsack//
6    repeat
7    if k > n then fp ← cp; fw ← cw; k ← n; X ← Y //update so-//
                                                    //lution//
8      else Y(k) ← 0  //M is exceeded so object k does not fit//
9    endif
10   while BOUND(cp, cw, k, M) ≤ fp do  //after fp is set above,//
                                          //BOUND = fp//
11     while k ≠ 0 and Y(k) ≠ 1 do
12       k ← k − 1  //find the last weight included in the knapsack//
13     repeat
14     if k = 0 then return endif  //the algorithm ends here//
15     Y(k) ← 0; cw ← cw − W(k); cp ← cp − P(k) //remove the kth//
                                                    //item//
16   repeat
17   k ← k + 1
18 repeat
19 end BKNAP1
```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	-	Implementación p. 490-1 *Recursivo*
		Fig. p.492 *Árbol de búsqueda*	Implementación p. 492-3 *Recursivo*

14.19 Cuando Alí-Babá por fin consigue entrar en la Cueva de los Cuarenta Ladrones encuentra allí gran cantidad de objetos muy valiosos. A pesar de su pobreza, Alí-Babá conoce muy bien el peso y valor (números reales) de cada uno de los objetos en la cueva. Debido a los peligros que tiene afrontar en su camino de vuelta, solo puede llevar consigo aquellas riquezas que quepan en su pequeña mochila, que soporta un peso máximo conocido. Determinar qué objetos debe elegir Alí-Babá para maximizar el valor total de lo que pueda llevarse en su mochila.

```
proc mochila-va(e P[1..n], V[1..n] de real+, e M : real, sol[1..n] de 0..1, e k : 1..n,
                peso, beneficio : real, sol-mejor[1..n] de 0..1, beneficio-mejor : real)
    { hijo izquierdo — coger objeto, no hacemos estimación }
    sol[k] := 1
    peso := peso + P[k] ;  beneficio := beneficio + V[k]     { marcar }
    si peso ≤ M entonces
        si k = n entonces
            sol-mejor := sol ;  beneficio-mejor := beneficio
        si no
            mochila-va(P, V, M, sol, k + 1, peso, beneficio, sol-mejor, beneficio-mejor)
        fsi
    fsi
    peso := peso − P[k] ;  beneficio := beneficio − V[k]     { desmarcar }
    { hijo derecho — no coger objeto, no se marca pero sí se hace estimación }
    sol[k] := 0
    beneficio-estimado := cálculo-estimación(P, V, M, k, peso, beneficio)
    si beneficio-estimado > beneficio-mejor entonces
        si k = n entonces
            sol-mejor := sol ;  beneficio-mejor := beneficio
        si no
            mochila-va(P, V, M, sol, k + 1, peso, beneficio, sol-mejor, beneficio-mejor)
        fsi
    fsi
fproc
fun cálculo-estimación(P[1..n], V[1..n] de real+, M : real+, k : 1..n, peso, beneficio : real)
                       dev estimación : real
    hueco := M − peso ;  estimación := beneficio
    j := k + 1
    mientras j ≤ n ∧ P[j] ≤ hueco hacer
        { podemos coger el objeto j entero }
        hueco := hueco − P[j] ;  estimación := estimación + V[j]
        j := j + 1
    fmientras
    si j ≤ n entonces { quedan objetos por probar }
        { fraccionamos el objeto j (solución voraz) }
        estimación := estimación + (hueco/P[j]) * V[j]
    fsi
ffun
fun mochila-principal(P[1..n], V[1..n] de real+, M : real+)
                      dev ⟨sol-mejor[1..n] de 0..1, beneficio-mejor : real⟩
var sol[1..n] de 0..1
    peso := 0 ;  beneficio := 0
    beneficio-mejor := −1     { peor que cualquier solución }
    mochila-va(P, V, M, sol, 1, peso, beneficio, sol-mejor, beneficio-mejor)
ffun
```

El siguiente problema es una variación del problema de la mochila, en este caso en lugar de tener que repartir los objetos en una sola mochila, se deben repartir en dos.

14.20 Pepe Casanova es un ligón de los de antaño, que intenta encandilar a las chicas con canciones románticas. A tal efecto, y de cara al veraneo en una playa del sur, decide conseguir una cinta para el radiocasete de su coche con las mejores canciones de amor. Pepe es muy peculiar en sus gustos, y además anda algo escaso de dinero, por lo que en lugar de comprar una de tantas recopilaciones que circulan por el mercado discográfico, quiere grabársela él mismo. Rebuscando entre sus viejos vinilos, ha confeccionado una lista de sus n canciones favoritas, junto con la duración individual de cada una. Lamentablemente, su cinta (de dos caras) de T minutos no tiene capacidad suficiente para contener todas las canciones, así que Pepe ha otorgado una puntuación a cada canción (cuanto más le gusta mayor es la puntuación). Ayudar a Pepe a conseguir la mejor cinta, teniendo en cuenta que las canciones escogidas han de caber enteras y no es admisible que una canción se corte a la mitad de una cara de la cinta.

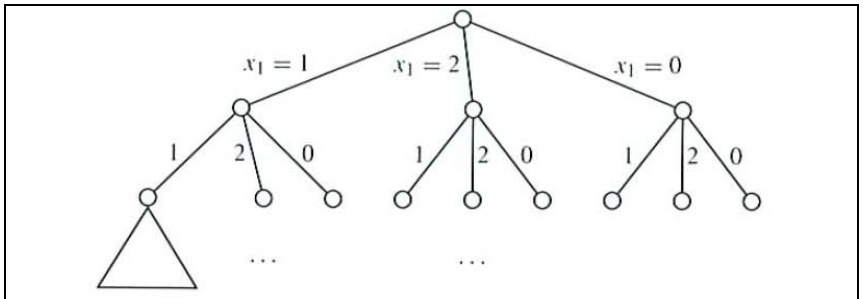


Figura 14.11: Árbol de exploración para el problema de las canciones.

Ilustración 50, árbol de búsqueda potencial (canciones)

```
fun canciones(D[1..n], P[1..n] de real+, T : real+)
                dev ⟨sol-mejor[1..n] de 0..2, beneficio-mejor : real⟩
var sol[1..n] de 0..2, ocupada[1..2] de real
    ocupada[1..2] := [0] ;  beneficio := 0
    beneficio-mejor := −1      { peor que cualquier solución }
    canciones-va(D, P, T, sol, 1, ocupada, beneficio, sol-mejor, beneficio-mejor)
ffun
```

```
proc canciones-va(e D[1..n], P[1..n] de real+, e T : real+, sol[1..n] de 0..2, e k : 1..n,
                  ocupada[1..2] de real, beneficio : real, sol-mejor[1..n] de 0..2, beneficio-mejor : re
    { primer hijo — grabar la canción k en la cara 1 }
    sol[k] := 1
    ocupada[1] := ocupada[1] + D[k] ;  beneficio := beneficio + P[k]     { marcar }
    si ocupada[1] ≤ T/2 entonces       { la estimación coincide con la del nodo padre }
        si k = n entonces
            sol-mejor := sol ;  beneficio-mejor := beneficio
        si no
            canciones-va(D, P, T, sol, k + 1, ocupada, beneficio, sol-mejor, beneficio-mejor)
        fsi
    fsi
    ocupada[1] := ocupada[1] − D[k] ;  beneficio := beneficio − P[k]     { desmarcar }
    si ocupada[1] ≠ ocupada[2] entonces
        { segundo hijo — grabar la canción k en la cara 2 }
        sol[k] := 2
        ocupada[2] := ocupada[2] + D[k] ;  beneficio := beneficio + P[k]     { marcar }
        si ocupada[2] ≤ T/2 entonces
            beneficio-estimado := cálculo-estimación(D, P, T, k, beneficio, ocupada)
            si beneficio-estimado > beneficio-mejor entonces
                si k = n entonces
                    sol-mejor := sol ;  beneficio-mejor := beneficio
                si no
                    canciones-va(D, P, T, sol, k + 1, ocupada, beneficio, sol-mejor, beneficio-mejor)
                fsi
            fsi
        fsi
        ocupada[2] := ocupada[2] − D[k] ;  beneficio := beneficio − P[k]     { desma
    fsi
    { tercer hijo — no grabar la canción k (siempre es factible) }
    sol[k] := 0
    beneficio-estimado := cálculo-estimación(D, P, T, k, beneficio, ocupada)
    si beneficio-estimado > beneficio-mejor entonces
        si k = n entonces
            sol-mejor := sol ;  beneficio-mejor := beneficio
        si no
            canciones-va(D, P, T, sol, k + 1, ocupada, beneficio, sol-mejor, beneficio-mej
        fsi
    fsi
fproc
```

```
fun cálculo-estimación(D[1..n], P[1..n] de real+, T : real+, k : 1..n, beneficio : real,
                       ocupada[1..2] de real) dev estimación : real
    hueco := T − (ocupada[1] + ocupada[2])
    estimación := beneficio
    j := k + 1
    mientras j ≤ n ∧ D[j] ≤ hueco hacer
        { podemos grabar la canción j entera }
        hueco := hueco − D[j]
        estimación := estimación + P[j]
        j := j + 1
    fmientras
    si j ≤ n entonces { quedan canciones por probar }
        { cortamos la canción j }
        estimación := estimación + (hueco/D[j]) ∗ P[j]
    fsi
ffun
```

Libro	Capítulo/ apartado	Visualización	Implementación
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.210, *Árbol de búsqueda*	Implementación p. 214-5 *Recursivo*

Algorithm 5. 7 A Backtracking Algorithm for the 0-1 Knapsack Problem
Problem: Let *n* items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer W be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed W.
Inputs: positive integers *n* and W, arrays *w* and *p*, each containing *n* positive integers and sorted in nonincreasing order according to the value of *p*[*i*]/*w*[*i*].
Outputs: an *n*-element array besrset, where the value of besrser[*i*] is "yes" if the *i*th item is included in the optimal set and is "no" otherwise; and an integer maxprofit that is the maximum profit.

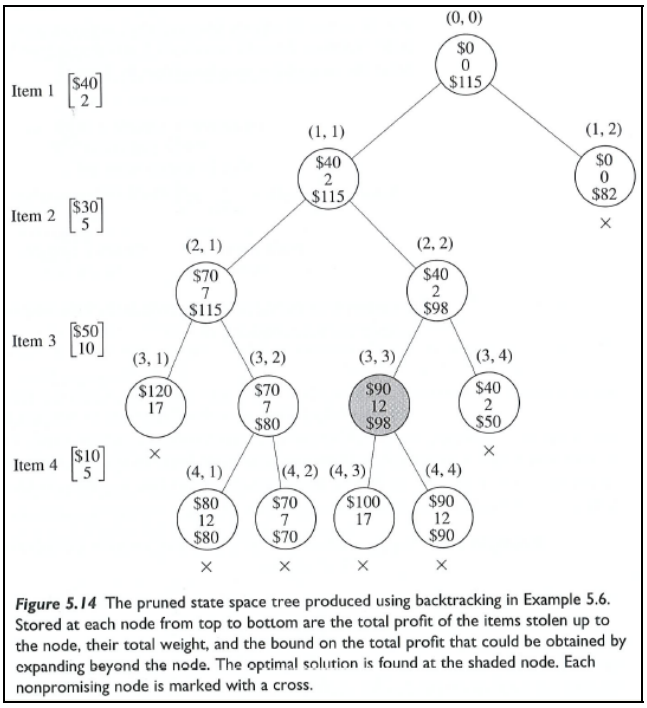


Figure 5.14 The pruned state space tree produced using backtracking in Example 5.6. Stored at each node from top to bottom are the total profit of the items stolen up to the node, their total weight, and the bound on the total profit that could be obtained by expanding beyond the node. The optimal solution is found at the shaded node. Each nonpromising node is marked with a cross.

Ilustración 51, árbol de búsqueda generado (mochila)

```
procedure knapsack (i: index;
          profit, weight: integer);
begin
  if weight ≤ W and profit > maxprofit then
    maxprofit:= profit;
    numbest:=i;
    bestset:= include
  end;
  if promising(i) then
    include[i+1]:= 'yes';
    knapsack(i+1, profit+p[i+1], weight+w[i+1]);
    include[i+1]:= 'no';
    knapsack(i+1, profit, weight)
  end
end;
```

```
function promising(i: index): boolean;
var
  j,k: index;
  totweight: integer;
  bound: real;
begin
  if weight ≥ W then
    promising:= false
  else
    j:=i+1;
    bound:= profit;
    totweight:= weight;
    while j ≤ n and totweight + w[j] ≤ W do
      totweight:= totweight + w[j];
      bound:= bound + p[j];
      j:= j+1
    end;
    k:= j;
    if k ≤ n then
      bound:= bound + (W − totweight)*p[k]/w[k]
    end;
    promising:= bound > maxprofit
  end
end;
```

Libro	Capítulo/ apartado	Visualización	Implementación
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Fig. p.837 *Árbol de búsqueda*	Implementación p. 855 *Recursivo*

Example 21.2 [0/1 Knapsack] Consider the knapsack instance n=3, w=[20,15,15], p=[40,25,25], and c=30. We search the tree of Figure 21.2, beginning at the root.

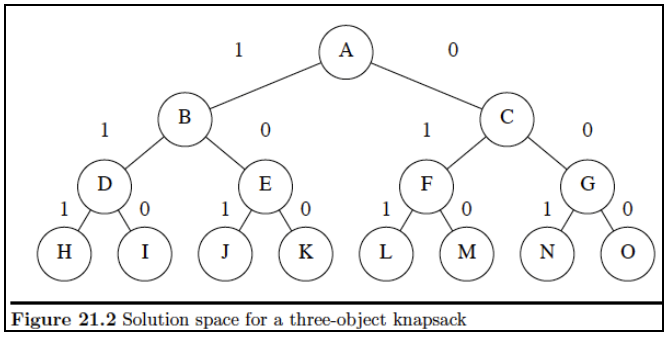


Figure 21.2 Solution space for a three-object knapsack

Ilustración 52, árbol de búsqueda potencial (mochila)

```
public static double knapsack(double [] theProfit, double [] theWeight,
                              double theCapacity)
{
   // set class data members
   capacity = theCapacity;
   numberOfObjects = theProfit.length - 1;
   weightOfCurrentPacking = 0.0;
   profitFromCurrentPacking = 0.0;
   maxProfitSoFar = 0.0;

   // define an Element array for profit densities
   Element [] q  = new Element [numberOfObjects];

   // set up densities in q[0:n-1]
   for (int i = 1; i <= numberOfObjects; i++)
      q[i - 1] = new Element(i, theProfit[i] / theWeight[i]);

   // sort into increasing density order
   MergeSort.mergeSort(q);

   // initialize class members
   profit = new double [numberOfObjects + 1];
   weight = new double [numberOfObjects + 1];
   for (int i = 1; i <= numberOfObjects; i++)
   {// profits and weights in decreasing density order
      profit[i] = theProfit[q[numberOfObjects - i].id];
      weight[i] = theWeight[q[numberOfObjects - i].id];
   }

   rKnap(1);  // compute max profit
   return maxProfitSoFar;
}
```

Program 21.7 The method `RecursiveBTKnapsack.knapsack`

2.1.1 Tabla resumen sobre el problema de la mochila 0/1

En la siguiente tabla se puede ver un resumen de los libros consultados que contienen el problema de la mochila 0/1.

Libro	Capítulo/ apartado	Visualización	Nomenclatura	Implementación
G. Brassard y P. Bratley, *Fundamentals of Algorithmics*	Capítulo 9 Apartado 6	Fig. p.307 *Árbol de búsqueda*	The knapsack problem (3)	Pseudocódigo p. 308 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figuras p. 354,358	KNAPSACK PROBLEM	Implementación p. 352, 355 *Iterativo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	- Fig. p.492 *Árbol de búsqueda*	14.19 14.20	Implementación p. 490-1 *Recursivo* Implementación p. 492-3. *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Fig. p.210 *Árbol de búsqueda*	The 0-1 Knapsack Problem	Implementación p. 214-5 *Recursivo*
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Fig. p.837 *Árbol de búsqueda*	0/1 Knapsack	Implementación p. 855 *Recursivo*

2.2 Problemas de asignación

Problema 17: El problema consiste en repartir tareas con el objetivo de maximizar el beneficio o minimizar el tiempo total empleado.

Libro	Capítulo/ apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	-	Implementación p. 480-3 *Recursivo*

14.15 El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus n funcionarios, para lo que se ha sacado de la manga n trabajos. A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros y unos lo hacen peor que otros. La información al respecto se recoge en dos tablas T[1..*n*,1..*n*] y E[1..*n*,1..*n*], donde T[*i*,*j*] representa el tiempo que el funcionario *i* tarda en realizar el trabajo *j*. Para justificar su puesto, Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a funcionarios en cada uno de los dos sentidos diferentes (e independientes) siguientes:
(a) de modo que la suma total de tiempos sea mínima, y
(b) de modo que la suma total de eficacias sea máxima.

Apartado (a)

```
proc funcionarios-mín-va(e T[1..n, 1..n] de real+, e est[1..n] de real, sol[1..n] de 1..n, e k : 1..n,
                         tiempo : real, asignado[1..n] de bool,
                         sol-mejor[1..n] de 1..n, tiempo-mejor : real∞)
    para t = 1 hasta n hacer
        si ¬asignado[t] entonces
            sol[k] := t
            asignado[t] := cierto ;  tiempo := tiempo + T[k, sol[k]]    { marcar }
            tiempo-estimado := tiempo + est[k]
            si tiempo-estimado < tiempo-mejor entonces      { se puede mejorar }
                si k = n entonces
                    sol-mejor := sol ;  tiempo-mejor := tiempo
                si no
                    funcionarios-mín-va(T, est, sol, k + 1, tiempo, asignado, sol-mejor, tiempo-mejor)
                fsi
            fsi
            asignado[t] := falso ;  tiempo := tiempo − T[k, sol[k]]    { desmarcar }
        fsi
    fpara
fproc
```

```
fun cálculo-estimaciones-mín(T[1..n, 1..n] de real+) dev est[1..n] de real
var rápido[1..n] de real+
    { cálculo de los mínimos por cada fila de T }
    para i = 1 hasta n hacer
        rápido[i] := T[i, 1]
        para j = 2 hasta n hacer
            rápido[i] := mín(rápido[i], T[i, j])
        fpara
    fpara
    { cálculo de las estimaciones }
    est[n] := 0
    para i = n − 1 hasta 1 paso −1 hacer
        est[i] := est[i + 1] + rápido[i + 1]
    fpara
ffun
fun funcionarios-mín(T[1..n, 1..n] de real+) dev ⟨sol-mejor[1..n] de 1..n, tiempo-mejor : real∞⟩
var est[1..n] de real, sol[1..n] de 1..n, asignado[1..n] de bool
    est[1..n] := cálculo-estimaciones-mín(T)
    tiempo := 0
    asignado[1..n] := [falso]
    tiempo-mejor := +∞
    funcionarios-mín-va(T, est, sol, 1, tiempo, asignado, sol-mejor, tiempo-mejor)
ffun
```

Apartado (b)

```
proc funcionarios-máx-va(e E[1..n, 1..n] de real+, e est[1..n] de real, sol[1..n] de 1..n, e k : 1..n,
                         eficacia : real, asignado[1..n] de bool,
                         sol-mejor[1..n] de 1..n, eficacia-mejor : real∞)
    para t = 1 hasta n hacer
        si ¬asignado[t] entonces
            sol[k] := t
            asignado[t] := cierto ; eficacia := eficacia + E[k, sol[k]]    { marcar }
            eficacia-estimada := eficacia + est[k]
            si eficacia-estimada > eficacia-mejor entonces      { se puede mejorar }
                si k = n entonces
                    sol-mejor := sol ; eficacia-mejor := eficacia
                si no
                    funcionarios-máx-va(E, est, sol, k + 1, eficacia, asignado, sol-mejor, eficacia-mejor)
                fsi
            fsi
            asignado[t] := falso ; eficacia := eficacia − E[k, sol[k]]    { desmarcar }
        fsi
    fpara
fproc
```

```
fun cálculo-estimaciones-máx(E[1..n, 1..n] de real+) dev est[1..n] de real
var mejor[1..n] de real+
    { cálculo de los máximos por cada fila de E }
    para i = 1 hasta n hacer
        mejor[i] := E[i, 1]
        para j = 2 hasta n hacer
            mejor[i] := máx(mejor[i], E[i, j])
        fpara
    fpara
    { cálculo de las estimaciones }
    est[n] := 0
    para i = n − 1 hasta 1 paso −1 hacer
        est[i] := est[i + 1] + mejor[i + 1]
    fpara
ffun
fun funcionarios-máx(E[1..n, 1..n] de real+) dev (sol-mejor[1..n] de 1..n, eficacia-mejor
var est[1..n] de real, sol[1..n] de 1..n, asignado[1..n] de bool
    est[1..n] := cálculo-estimaciones-máx(E)
    eficacia := 0
    asignado[1..n] := [falso]
    eficacia-mejor := −∞
    funcionarios-máx-va(E, est, sol, 1, eficacia, asignado, sol-mejor, eficacia-mejor)
ffun
```

2.3 Otros problemas de optimización

En este apartado se presentan otros problemas de optimización resueltos con la técnica de vuelta atrás.

Libro	Capítulo/ apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	-	Implementación p. 484-5 *Recursivo*
		-	Implementación p. 487-7 *Recursivo*
		Fig. p.488 *Árbol de búsqueda para el problema de los comensales*	Implementación p. 488-9 *Recursivo*

14.16 El tío Facundo posee n huertas, cada una con un tipo diferente de árboles frutales. Las frutas ya han madurado y es hora de recolectarlas. El tío Facundo conoce, para cada una de las huertas, el beneficio b_i que obtendría por la venta de lo recolectado. El tiempo que tarda en recoger los frutos de cada finca es asimismo variable y viene dado por t_i. También sabe los días d_i que tardan en producirse los frutos de cada huerta. Ayudar al tío Facundo a decidir qué debe recolectar para maximizar el beneficio total obtenido.

```
proc huertas-va(e T[1..n], D[1..n], B[1..n] de real+, sol[1..n] de 0..1, e k : 1..n, tiempo,
                beneficio : real, sol-mejor[1..n] de 0..1, beneficio-mejor : real∞)
    { hijo izquierdo — recolectar }
    sol[k] := 1
    si tiempo + T[k] ≤ D[k] entonces      { es factible }
        tiempo := tiempo + T[k] ; beneficio := beneficio + B[k]    { marcar }
        beneficio-est := cálculo-estimación(T, D, B, k, tiempo, beneficio)
        si beneficio-est > beneficio-mejor entonces
            si k = n entonces
                sol-mejor := sol ; beneficio-mejor := beneficio
            si no
                huertas-va(T, D, B, sol, k + 1, tiempo, beneficio, sol-mejor, beneficio-mejo
            fsi
        fsi
        tiempo := tiempo − T[k] ; beneficio := beneficio − B[k]    { desmarcar }
    fsi
    { hijo derecho — no recolectar }
    sol[k] := 0
    { no hace falta test de factibilidad }
    beneficio-est := cálculo-estimación(T, D, B, k, tiempo, beneficio)
    si beneficio-est > beneficio-mejor entonces      { se puede mejorar }
        si k = n entonces
            sol-mejor := sol ; beneficio-mejor := beneficio
        si no
            huertas-va(T, D, B, sol, k + 1, tiempo, beneficio, sol-mejor, beneficio-mejor)
        fsi
    fsi
fproc
fun cálculo-estimación(T[1..n], D[1..n], B[1..n] de real+, k : 1..n, tiempo, beneficio : rea
                       dev beneficio-est : real
    beneficio-est := beneficio
    para i = k + 1 hasta n hacer
        si tiempo + T[i] ≤ D[i] entonces beneficio-est := beneficio-est + B[i] fsi
    fpara
ffun
fun huertas-principal(T[1..n], D[1..n], B[1..n] de real+)
                      dev ⟨sol-mejor[1..n] de 0..1, beneficio-mejor : real⟩
var sol[1..n] de 0..1
    tiempo := 0 ; beneficio := 0
    beneficio-mejor := −1    { peor que cualquier solución }
    huertas-va(T, D, B, sol, 1, tiempo, beneficio, sol-mejor, beneficio-mejor)
ffun
```

14.17 Tenemos un conjunto de *n* componentes electrónicas para colocar *n* posiciones sobre una placa. Se nos dan dos matrices de dimensiones *n*×*n* y D, donde N[*i*,*j*] indica el número de conexiones necesarias entre la componente *i* y la componente *j*, y D[*p*,*q*] indica la distancia sobre la placa entre la posición *p* y la posición *q* (ambas matrices son simétricas y con diagonales nulas). Un cableado (x_1,…,x_n) de la placa consiste en

la colocación de cada componente *i* en una posición distinta x_i. La longitud total de este cableado viene dada por la fórmula:

$$\sum_{i<j} N[i, j]D[x_i, x_j].$$

Escribir un algoritmo para encontrar el cableado de longitud mínima.

```
proc cableado-va(e N[1..n, 1..n] de nat, e D[1..n, 1..n] de real, e est[1..n] de real, sol[1..n] de 1..n,
                 e k : 1..n, coste : real, usada[1..n] de bool, sol-mejor[1..n] de 1..n,
                 coste-mejor : real∞)
    para p = 1 hasta n hacer
        si ¬usada[p] entonces
            sol[k] := p
            { marcar }
            usada[p] := cierto
            coste-comp := coste-componente(N, D, sol, k)
            coste := coste + coste-comp
            coste-estimado := coste + est[k]
            si coste-estimado < coste-mejor entonces { se puede mejorar sol-mejor }
                si k = n entonces
                    sol-mejor := sol ; coste-mejor := coste
                si no
                    cableado-va(N, D, est, sol, k + 1, coste, usada, sol-mejor, coste-mejor)
                fsi
            fsi
            { desmarcar }
            coste := coste − coste-comp
            usada[p] := falso
        fsi
    fpara
fproc
fun coste-componente(N[1..n, 1..n] de nat, D[1..n, 1..n] de real, sol[1..n] de 1..n, k : 1..n)
                     dev coste-comp : real
    coste-comp := 0
    para i = 1 hasta k − 1 hacer
        coste-comp := coste-comp + N[i, k] * D[sol[i], sol[k]]
    fpara
ffun
fun pre-cálculo-estimaciones(N[1..n, 1..n] de nat, D[1..n, 1..n] de real) dev est[1..n] de real
    { cálculo del mínimo de D }
    mínD := +∞
    para i = 1 hasta n hacer
        para j = i + 1 hasta n hacer
            mínD := mín(mínD, D[i, j])
        fpara
    fpara
```

```
    { cálculo de las estimaciones }
    est[n] := 0
    para k = n − 1 hasta 1 paso − 1 hacer
        est[k] := est[k + 1]
        para i = 1 hasta k hacer
            est[k] := est[k] + N[i, k + 1]
        fpara
    fpara
    para k = 1 hasta n hacer est[k] := mínD * est[k] fpara
ffun
{ est[k] = mínD Σ_{j=k+1}^{n} Σ_{i=1}^{j−1} N[i, j] }
fun cableado(N[1..n, 1..n] de nat, D[1..n, 1..n] de real)
                dev ⟨sol-mejor[1..n] de 1..n, coste-mejor : real_∞⟩
var est[1..n] de real, sol[1..n] de 1..n, usada[1..n] de bool
    est[1..n] := pre-cálculo-estimaciones(N, D)
    coste := 0
    usada[1..n] := [falso]
    coste-mejor := +∞
    cableado-va(N, D, est, sol, 1, coste, usada, sol-mejor, coste-mejor)
ffun
```

4.18 Un grupo de amigos, formado por *n* parejas (*n* mujeres y *n* hombres), se reúnen para cenar. El protocolo del buen comensal exige que, a la hora de sentarse en la mesa (redonda), hombres y mujeres deben alternarse y que nadie debe sentarse al lado de su pareja habitual. Para que la velada sea lo más agradable posible, hay que maximizar el grado de bienestar total, obtenido sumando los grados de afinidad mutuos entre los comensales sentados en posiciones adyacentes. Al efecto, se dispone de sendas matrices que nos indican la afinidad entre hombres y mujeres y entre mujeres y hombres. Una vez establecida cada pareja de vecinos, la afinidad mutua se calcula multiplicando la de cada miembro por la del contrario. Diseñar un algoritmo que encuentre una solución óptima al problema.

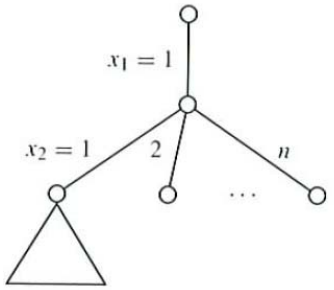


Figura 14.10: Árbol de exploración para el problema de los comensales.

Ilustración 53, árbol de búsqueda potencial (comensales)

```
fun cena(P[0..1, 1..n, 1..n] de real) dev ⟨sol-mejor[1..2n] de 1..n, bienestar-mejor : real∞⟩
var sol[1..2n] de 1..n, sentado[0..1, 1..n] de bool
    { cálculo de los máximos }
    máxH := −∞ ; máxM := −∞
    para i = 1 hasta n hacer
        para j − 1 hasta n hacer
            si i ≠ j entonces
                máxH := máx(máxH, P[1, i, j]) ; máxM := máx(máxM, P[0, i, j])
            fsi
        fpara
    fpara
    sol[1] := 1
    bienestar := 0
    sentado[0..1, 1..n] := [falso] ; sentado[1, 1] := cierto
    bienestar-mejor := −∞
    cena-va(P, máxH, máxM, sol, 2, bienestar, sentado, sol-mejor, bienestar-mejor)
ffun

proc cena-va(e P[0..1, 1..n, 1..n] de real, e máxH, máxM : real, sol[1..2n] de 1..n, e k : 1..2n,
             bienestar : real, sentado[0..1, 1..n] de bool,
             sol-mejor[1..2n] de 1..n, bienestar-mejor : real∞)
    sexo := k mod 2 ; anterior := sol[k − 1]
    para i = 1 hasta n hacer
        si ¬sentado[sexo, i] ∧ i ≠ anterior entonces
            sol[k] := i
            { marcar }
            sentado[sexo, i] := cierto
            bienestar := bienestar + P[sexo, i, anterior] ∗ P[1 − sexo, anterior, i]
            si k = 2 ∗ n entonces
                si i ≠ 1 ∧ bienestar + P[1, 1, i] ∗ P[0, i, 1] > bienestar-mejor entonces
                    sol-mejor := sol ; bienestar-mejor := bienestar + P[1, 1, i] ∗ P[0, i, 1]
                fsi
            si no
                si bienestar-estimado > bienestar-mejor entonces      { se puede mejorar }
                    cena-va(P, máxH, máxM, sol, k + 1, bienestar, sentado, sol-mejor, bienestar-mejor)
                fsi
            fsi
            { desmarcar }
            sentado[sexo, i] := falso
            bienestar := bienestar − P[sexo, i, anterior] ∗ P[1 − sexo, anterior, i]
        fsi
    fpara
fproc
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 494-5 *Árbol de búsqueda*	Implementación p. 494-5 *Recursivo*

Los apartados (a) y (b) se han incluido en el apartado de problemas de decisión.

14.21 Juanito el Explorador está pasando unas vacaciones Tombuctú. Juanito disfruta mandando postales de los exóticos lugares que visita, para que sus amigos rabien de envidia cuando las reciban. A tal efecto se ha pasado por la oficina e correos más cercana y se ha hecho con un lote de sellos de n valores diferentes, disponiendo de tres sellos de cada valor. En correos también le han informado de las diferentes tarifas que de franqueo de tarjetas postales, y le han explicado que en la esquina superior derecha de cada postal aparece un bloque remarcado destinado a su franqueo. Dicho bloque está dividido en cinco casillas, destinadas a un sello, de forma que un franqueo solo es admisible si se alcanza la tarifa correspondiente y se cubren exactamente las cinco casillas.
(c) Escribir un algoritmo para obtener una solución con el mínimo coste posible

Apartado (c)

```
{ V[1] ≤ V[2] ≤ ... ≤ V[n] }
fun coste-estimado(V[1..n] de real+, T : real+, k : 1..5, último : 1..n, usados : 0..3, coste : real)
                   dev estimación : real
    estimación := coste
    j := k + 1 ;  i := último
    mientras j ≤ 5 ∧ i ≤ n hacer
        si usados < 3 entonces
            estimación := estimación + V[i]
            usados := usados + 1 ;   j := j + 1
        si no
            i := i + 1 ;  usados := 0
        fsi
    fmientras
    estimación := máx(estimación, T)
ffun
```

```
{ V[1] ≤ V[2] ≤ ... ≤ V[n] }
proc sellos-mín-va(e V[1..n] de real+, e T : real+, sol[1..5] de 1..n, e k : 1..5, último : 1..n,
                   usados : 0..3, coste, suma-libres : real, sol-mejor[1..5] de nat,
                   coste-mejor : real∞, encontrada : bool)
    { probamos un sello de tipo último }
    si usados < 3 entonces
        sol[k] := último
        usados := usados + 1 ;  coste := coste + V[último]     { marcar }
        suma-libres := suma-libres − V[último]
        si k = 5 entonces
            si coste ≥ T entonces      { es solución }
                sol-mejor := sol ;  coste-mejor := coste
                encontrada := (coste = T)     { terminar }
            fsi
        si no
            si factible?(T, k, último, usados, coste, suma-libres) entonces
                sellos-mín-va(V, T, sol, k + 1, último, usados, coste, suma-libres,
                              sol-mejor, coste-mejor, encontrada)
            fsi
        fsi
        usados := usados − 1 ;  coste := coste − V[último]     { desmarcar }
        suma-libres := suma-libres + V[último]
    fsi
    { descartamos los sellos no utilizados de tipo último }
    suma-libres-act := suma-libres − (3 − usados) * V[último]
    { probamos con el resto de tipos de sellos }
    usados-act := 1
    s := último + 1
    mientras ¬encontrada ∧ s ≤ n hacer
        sol[k] := s
        coste := coste + V[s]     { marcar }
        suma-libres-act := suma-libres-act − V[s]     { quitamos un sello de tipo s }
        si coste-estimado(V, T, k, s, usados-act, coste) < coste-mejor entonces
            si k = 5 entonces
                si coste ≥ T entonces      { es solución }
                    sol-mejor := sol ;  coste-mejor := coste
                    encontrada := (coste = T)     { terminar }
                fsi
            si no
                si factible?(T, k, s, usados-act, coste, suma-libres-act) entonces
                    sellos-mín-va(V, T, sol, k + 1, s, usados-act, coste, suma-libres-act,
                                  sol-mejor, coste-mejor, encontrada)
                fsi
            fsi
        fsi
        coste := coste − V[s]     { desmarcar }
        { quitamos los dos sellos restantes de tipo s }
        suma-libres-act := suma-libres-act − 2 * V[s]
        s := s + 1
    fmientras
fproc
```

```
{ V[1] ≤ V[2] ≤ ... ≤ V[n] }
fun sellos-mín(V[1..n] de real+, T : real+) dev ⟨sol-mejor[1..5] de 1..n, coste-mejor : real∞⟩
var sol[1..5] de 1..n
    suma-todos := 0
    para i = 1 hasta n hacer suma-todos := suma-todos + 3 * V[i] fpara
    si 3 * n ≥ 5 ∧ suma-todos ≥ T entonces
        último := 1 ; usados := 0 ; coste := 0
        coste-mejor := +∞ ; encontrada := falso
        sellos-mín-va(V, T, sol, 1, último, usados, coste, suma-todos, sol-mejor, coste-mejor, encontrad
    fsi
fproc
```

Libro	Capítulo/apartado	Visualización	Implementación
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 499 *Árbol de búsqueda*	Implementación p. 500-1 *Recursivo*

14.22 Se tiene una colección de n objetos "moldeables", cada uno con un volumen v_i, para i entre 1 y n, que hay que empaquetar utilizando envases de capacidad E. Diseñar un algoritmo que calcule el empaquetamiento óptimo, es decir, que minimice la cantidad de envases utilizados, teniendo en cuenta que los objetos no se pueden fraccionar.

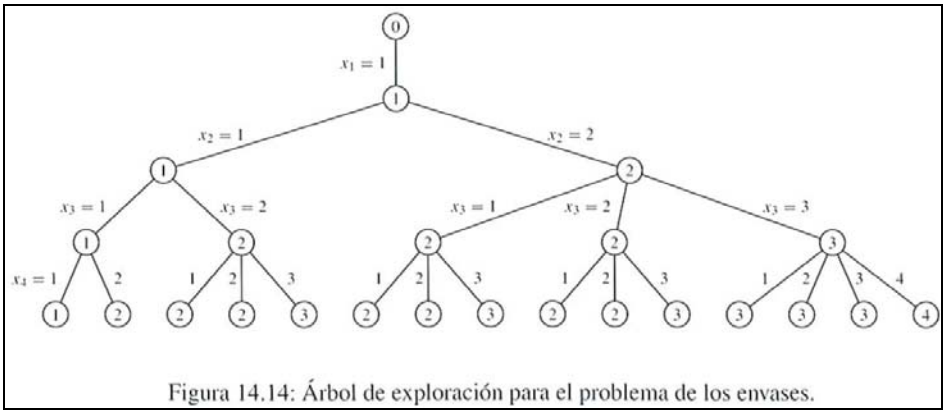


Figura 14.14: Árbol de exploración para el problema de los envases.

Ilustración 54, árbol de búsqueda potencial (envases).

```
{ (∀i : 1 ≤ i ≤ n : V[i] ≤ E) ∧ óptimo = ⌈(∑_{i=1}^{n} v_i)/E⌉ }
proc empaquetar-va(e E : real+, e V[1..n] de real+, sol[1..n] de 1..n, e k : 1..n, envases : 1..n,
                   capacidad[1..n] de real, e óptimo : 1..n,
                   sol-mejor[1..n] de 1..n, envases-mejor : 1..n + 1, encontrada : bool)
    { probamos con cada envase ya utilizado }
    i := 1
    mientras i ≤ envases ∧ ¬encontrada hacer
        si capacidad[i] ≥ V[k] entonces
            sol[k] := i
            capacidad[i] := capacidad[i] − V[k]     { marcar }
            si k = n entonces
                sol-mejor := sol ; envases-mejor := envases
                encontrada := (envases-mejor = óptimo)     { terminar }
            si no
                empaquetar-va(E, V, sol, k + 1, envases, capacidad, óptimo,
                              sol-mejor, envases-mejor, encontrada)
            fsi
            capacidad[i] := capacidad[i] + V[k]     { desmarcar }
        fsi
        i := i + 1
    fmientras
    si ¬encontrada entonces
        { probamos con un envase nuevo }
        sol[k] := envases + 1
        envases := envases + 1 ; capacidad[envases] := E − V[k]     { marcar }
        si envases < envases-mejor entonces
            si k = n entonces
                sol-mejor := sol ; envases-mejor := envases
                encontrada := (envases-mejor = óptimo)     { terminar }
            si no
                empaquetar-va(E, V, sol, k + 1, envases, capacidad, óptimo,
                              sol-mejor, envases-mejor, encontrada)
            fsi
        fsi
        capacidad[envases] := E ; envases := envases − 1     { desmarcar }
    fsi
fproc
fun empaquetar(E : real+, V[1..n] de real+) dev ⟨sol-mejor[1..n] de 1..n, envases-mejor : 1..n + 1⟩
var sol[1..n] de 1..n, capacidad[1..n] de real
    sol[1] := 1
    envases := 1
    capacidad[1] := E − V[1] ; capacidad[2..n] := [E]
    total := 0
    para i = 1 hasta n hacer total := total + V[i] fpara
    óptimo := ⌈total/E⌉
    encontrada := falso
    envases-mejor := n + 1     { peor que cualquier solución }
    empaquetar-va(E, V, sol, 2, envases, capacidad, óptimo, sol-mejor, envases-mejor, encontrada)
ffun
```

3 Tabla Resumen

A continuación se presenta una tabla donde se puede consultar un resumen de los problemas descritos en los apartados anteriores.

		Problemas de decisión		
Libro	**Capítulo/apartado**	**Visualización**	**Nomenclatura**	**Implementación**
M. H. Alsuwaiyel, *Algorithm Design Techniques and Analysis*	Capítulo 13	Figura p. 358 *Tablero* Figura p. 360 *Árbol de búsqueda*	The 8-Queens Problem	*Pseudocódigo p. 359* *Una solución* *Iterativo*
G. Brassard y P. Bratley, *Fundamentals of Algorithmics*	Capítulo 9 Apartado 6	-	The eight queens problem	Pseudocódigo p. 345 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figura p. 325 *Tablero* Figura p. 326 *Árbol de búsqueda* Figura p. 331 *Tablero y árbol de búsqueda*	8-queens/n-queens/4-queens	Implementación p. 338 *Todas las soluciones* *Iterativo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 466 *Tablero*	14.8	Implementación p. 466 *Todas las soluciones* *Recursivo*

R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figura p. 179 *Árbol de búsqueda* Figura p. 181 *Tablero* Figura p. 182 *Árbol de búsqueda* Figura p. 183 *Tablero* Figura p. 186 *Tablero*	THE *n*-QUEENS PROBLEM	Implementación p. 186-87 *Todas las soluciones* *Recursivo*
M. H. Alsuwaiyel, *Algorithm Design Techniques and Analysis*	Capítulo 13	Figura p. 354 *Árbol de búsqueda* Figura p. 355 Grafo y árbol de búsqueda	The3-Coloring Problem	Pseudocódigo p. 356-7 *Recursivo e iterativo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figura p. 346 *Árbol de búsqueda* Figura p. 347 *Grafo y árbol de búsqueda*	GRAPH COLORING	Implementación p. 345 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figura p. 200 *Grafo* Figura p. 201 *Grafo* Figura p. 203 *Árbol de búsqueda*	GRAPH COLORING	Implementación p. 202 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figura p. 348 *Grafos*	HAMILTONIAN CYCLES	Implementación p. 350 *Todas las soluciones* *Recursivo*

N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 476 *Árbol de búsqueda*	14.4 (problema del vendedor)	Implementación p. 477 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figura p. 205 *Grafos*	THE HAMILTONIAN CIRCUITS PROBLEM	Implementación p. 206 *Todas las soluciones* *Recursivo*
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Figura p. 839 *Grafos* Figura p. 840 *Árbol de búsqueda*	Example 21.3 [Traveling Salesperson]	Implementación p. 862 *Todas las soluciones* *Recursivo*
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	Fig. p.236 *Árbol de búsqueda*	Constructing All Paths in a Graph	Implementación p. 237 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Fig. p.457 *Árbol de búsqueda*	Grafos isomorfos	Implementación p. 457-8 *Todas las soluciones* *Recursivo*
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resuelto*	Capítulo 4	-	Generar palabras con restricciones	Pseudocódigo p. 77-79 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 454 *Árbol de búsqueda* -	14.1 (problema de las variaciones) 14.6	Implementación p. 455 *Todas las soluciones* *Recursivo* 14.6 Implementación p. 463-4 *Todas las soluciones* *Recursivo*

S. Skiena, The Algorithm Design Manual	Capítulo 7	-	Constructing All Permutations	Implementación p. 235-6 Todas las soluciones Recursivo
J. Gonzalo Arroyo y M. Rodríguez Artacho, Esquemas algorítmicos enfoque metodológico y problemas resueltos	Capítulo 4	Figura p. 86 Tablero	Recorrido del Caballo de Ajedrez	Pseudocódigo p. 87-90 Primera solución Recursivo
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	-	14.10	Implementación p. 468-70 *Primera solución* *Vuelta a la casilla de partida* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	-	14.12	Implementación p. 472-3 Recursivo
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas resueltos*	Capítulo 4	Figura p. 92 *Cuadrado mágico*	Cuadrado Mágico de suma 15	Pseudocódigo p. 92-3 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 459 *Tablero*	14.4 (cuadrado latino)	Implementación p. 460-1 *Todas las soluciones* *Recursivo*
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	Figura p. 239 *Tablero*	Sudoku	Implementación p. 239-41 *Recursivo*
J. Gonzalo Arroyo y M. Rodríguez Artacho, *Esquemas algorítmicos enfoque metodológico y problemas*	Capítulo 4	Figura p. 81 *Dominó*	Cadena de fichas del Dominó	Pseudocódigo p. 82-4 *Todas las soluciones* *Recursivo*

resueltos, UNED				
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 467 *Árbol de búsqueda*	14.9 (problema del dominó)	Implementación p. 468 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figura p. 327-8,344 *Árbol de búsqueda*	SUM OF SUBSETS	Implementación p. 342 *Todas las soluciones* *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figuras p. 195,197	THE SUM-OF -SUBSETS PROBLEM	Implementación p. 198 Todas las soluciones Recursivo
S. Skiena, *The Algorithm Design Manual*	Capítulo 7	-	Constructing All Subsets	Implementación p. 234 Todas las soluciones Recursivo
S. Sahni, 2000 *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	-	Container Loading	Implementación p. 845 *Todas las soluciones* *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 462 Árbol de búsqueda	14.5 (problema de las factorias)	Implementación p. 462 *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 494-5 Árbol de búsqueda	14.21 (problema de los sellos)	Implementación p. 494-5 *Recursivo*

N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 471 *Árbol de búsqueda*	14.11 (problema del reparto)	Implementación p. 470-2 *Recursivo*
Problemas de optimización				
Libro	**Capítulo/apartado**	**Visualización**	**Nomenclatura**	**Implementación**
G. Brassard y P. Bratley, *Fundamentals of Algorithmics*	Capítulo 9 Apartado 6	Figura p. 307 *Árbol de búsqueda*	The knapsack problem (3)	Pseudocódigo p. 308 *Todas las soluciones* *Recursivo*
E. Horowitz y S. Sahni, *Fundamentals of Computer Algorithms*	Capítulo 7	Figuras p. 354,358	KNAPSACK PROBLEM	Implementación p. 352, 355 *Iterativo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	- Figura p. 492 *Árbol de búsqueda*	14.19 14.20 (problema de las canciones)	Implementación p. 490-1 *Recursivo* Implementación p. 492-3. *Recursivo*
R. Neapolitan y K. Naimipour, *Foundations of Algorithms*	Capítulo 5 Apartado 1	Figura p. 210 *Árbol de búsqueda*	The 0-1 Knapsack Problem	Implementación p. 214-5 *Recursivo*
S. Sahni, *Data Structures, Algorithms, and Applications in Java*	Capítulo 21	Fig. p.837 *Árbol de búsqueda*	0/1 Knapsack	Implementación p. 855 *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	-	14.15 *(asignación de trabajo)*	Implementación p. 480-3 *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	- - Fig. p.488 *Árbol de búsqueda para el problema de los*	14.16 *(recolección)* 14.17 *(conexiones de placas)* 14.18	Implementación p. 484-5 *Recursivo* Implementación p. 487-7 *Recursivo* Implementación p. 488-9

		comensales		*Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figuras p. 494-5 Árbol de búsqueda	14.21 (problema de los sellos)	Implementación p. 494-5 *Recursivo*
N. Martí Oliet, Y. Ortega y J. A. Verdejo, *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*	Capítulo 14	Figura p. 499 *Árbol de búsqueda*	14.22 (problema de los envases)	Implementación p. 500-1 *Recursivo*

A continuación se presenta otra tabla donde se encuentran todos problemas presentados en los apartados anteriores, con un primer análisis de las características de las figuras.

Nomenclatura	**Libro**	**Representación**	**Características**
The 8-Queens Problem	Alsuwaiyel, Capítulo 13, Fig. p.358	Dependiente del dominio	• Tablero de ajedrez (soluciones completa y parcial)
The 8-Queens Problem	Alsuwaiyel, Capítulo 13, Fig. p.360	Figura compuesta	• Árbol generado y representación dependiente del dominio (solución) • Nodos circulares (válidos: vacíos; inválidos: con cruz) • Etiquetas de decisiones en rama de solución válida
8-queens	Horowitz y Sahni, Capítulo 7, Fig. p.326	Árbol de búsqueda	• Árbol de búsqueda potencial • Nodos circulares, con número de orden • Etiquetas de decisiones en todas las ramas
n-queens	Horowitz y Sahni, Capítulo 7, Fig. p.325	Dependiente del dominio	• Tablero de ajedrez (solución completa)
4-queens	Horowitz y Sahni, Capítulo 7, Fig. p.331	Secuencia de estados	• Secuencia de tableros generados (dependiente del dominio) • Cada tablero representa varios nodos hermanos del árbol de búsqueda -- (diferenciando estados válidos e inválidos)
4-queens	Horowitz y Sahni, Capítulo 7, Fig. p.331	Árbol de búsqueda	• Árbol de búsqueda generado

			• Nodos circulares, con número de orden • Etiquetas de decisiones en algunas ramas (tipo ojo de pez) • Etiqueta adicional (B) en nodos inválidos
14.8	Martí Oliet *et al.*, Capítulo 14, Fig. p.466	Dependiente del dominio	• Tablero de ajedrez (numeración de diagonales)
THE *n*-QUEENS PROBLEM	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.179	Árbol de búsqueda	• Árbol de búsqueda potencial • Árbol abreviado, con subárboles suprimidos (tipo ojo de pez) • Nodos circulares, con coordenadas de reinas
THE *n*-QUEENS PROBLEM	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.181	Dependiente del dominio	Tablero de ajedrez (dos soluciones parciales inválidas)
THE *n*-QUEENS PROBLEM	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.182	Árbol de búsqueda	• Árbol de búsqueda generado • Nodos circulares, con coordenadas de reinas • Etiqueta adicional (x) en nodos inválidos • Nodo sombreado con solución válida
THE *n*-QUEENS PROBLEM	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.183	Secuencia de estados	• Secuencia de tableros generados (dependiente del dominio) • Cada tablero representa varios nodos hermanos del árbol de búsqueda (sin diferenciar estados válidos e inválidos)
THE *n*-QUEENS PROBLEM	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.186	Dependiente del dominio	• Tablero de ajedrez (solución parcial)
The3-Coloring Problem	Alsuwaiyel, Capítulo 13, Fig. p.354	Árbol de búsqueda	• Árbol de búsqueda potencial, genérico para tamaño 3 • Etiquetas • Nodos circulares, vacíos
The3-Coloring Problem	Alsuwaiyel, Capítulo 13, Fig. p.355	Figura compuesta	• Representación dependiente del dominio (grafo de 5 nodos) y árbol generado • Nodos circulares (válidos: vacíos; inválidos: con cruz; solución final: sombreado) • Etiquetas de decisiones en rama de solución válida

GRAPH COLORING	Horowitz y Sahni, Capítulo 7, Fig. p.346	Árbol de búsqueda	• Árbol de búsqueda potencial, genérico para tamaños 3 y 3 • Nodos circulares, vacíos • Etiquetas de decisiones en algunas ramas (tipo ojo de pez)
GRAPH COLORING	Horowitz y Sahni, Capítulo 7, Fig. p.347	Figura compuesta	• Representación dependiente del dominio (grafo de 4 nodos) y árbol potencial • Nodos circulares, vacíos • Etiquetas de decisiones por niveles (identificador de variable) y en ramas (valores)
GRAPH COLORING	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.200	Dependiente del dominio	• Grafo de 4 nodos
GRAPH COLORING	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.201	Dependiente del dominio	• Dos Grafos planar
GRAPH COLORING	Neapolitan y Naimipour, Capítulo 5 Apartado 1, Fig. p.203	Árbol de búsqueda	• Árbol de búsqueda generado • Árbol abreviado, con subárboles suprimidos (tipo ojo de pez) • Nodos circulares, con número de nodo • Nodo de solución final, sombreado
HAMILTONIAN CYCLES	Horowitz y Sahni, Capítulo 7, Fig. p.348	Dependiente del dominio	• Dos grafos, uno con solución
14.14 (problema del vendedor)	Martí Oliet *et al.*, Capítulo 14, Fig. p.476	Árbol de búsqueda	• Árbol de búsqueda potencial • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos, y arcos suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (algunas con identificador, tipo ojo de pez)
THE HAMILTONIAN CIRCUITS PROBLEM	Neapolitan y Naimipour, Capítulo 5, Figura p. 205	Dependiente del dominio	• Dos grafos, uno con solución (los mismos que en Horowitz y Sahni, 2 filas arriba)

Example 21.3 [Traveling Salesperson]	Sahni, Capítulo 21, Fig. p.839	Dependiente del dominio	• Grafo de 4 nodos
Example 21.3 [Traveling Salesperson]	Sahni, Capítulo 21, Fig. p.840	Árbol de búsqueda	• Árbol de búsqueda potencial • Nodos circulares (numerados con letras) • Etiquetas de decisiones en ramas (nodo de destino)
7.1.3 Constructing All Paths in a Graph	Skiena, Capítulo 7, Fig. p.236	Figura compuesta	• Representación dependiente del dominio (grafo de 6 nodos) y árbol generado • Nodos puntuales (parciales: puntos; completos: puntos rodeados), con etiqueta de número de nodo
14.3 (grafos isomorfos)	Martí Oliet *et al.*, Capítulo 14, Fig. p.476	Dependiente del dominio	• Dos grafos isomorfos
14.1 (problema de las variaciones)	Martí Oliet *et al.*, Capítulo 14, Fig. p.454	Árbol de búsqueda	• Árbol de búsqueda potencial • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos, y arcos suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (algunas con identificador, tipo ojo de pez)
Recorrido del Caballo de Ajedrez	Gonzalo Arroyo y Rodríguez Artacho, Capítulo 4, Fig. p.86	Dependiente del dominio	• Tablero de 5×5 (muestra y numera movimientos válidos)
14.10	Martí Oliet *et al.*, Capítulo 14, p.469	Dependiente del dominio	• Tablero parcial (muestra y numera movimientos válidos)
14.12	Martí Oliet *et al.*, Capítulo 14, p.473	Dependiente del dominio	• Laberinto parcial (muestra y numera movimientos válidos)
Cuadrado Mágico de suma 15	Gonzalo Arroyo y Rodríguez Artacho, Capítulo 4, Fig. p.92	Dependiente del dominio	• Solución del cuadrado mágico
14.4 (cuadrado latino)	Martí Oliet *et al.*, Capítulo 14, Fig. p.459	Dependiente del dominio	• Solución del cuadrado latino

14.4 (cuadrado latino)	Martí Oliet *et al.*, Capítulo 14, Fig. p.459	Dependiente del dominio	• Numeración de celdas del cuadrado latino
Sudoku	Skiena, Capítulo 7, Fig. p.239	Dependiente del dominio	• Ejemplo y solución de sudoku
Cadena de fichas del Dominó	Gonzalo Arroyo y Rodríguez Artacho, Capítulo 4, Fig. p.81	Dependiente del dominio	• Ejemplo de partida (parcial) de dominó
14.9 (problema del dominó)	Martí Oliet *et al.*, Capítulo 14, Fig. p.467	Árbol de búsqueda	• Árbol de búsqueda potencial • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos, y arcos suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (algunas con identificador, tipo ojo de pez)
SUM OF SUBSETS	Horowitz y Sahni, Capítulo 7, Fig. p.327	Árbol de búsqueda	• Árbol de búsqueda potencial para un problema concreto • Nodos circulares, numerados por orden de generación en anchura • Etiquetas de decisiones en ramas (variable y valor)
SUM OF SUBSETS	Horowitz y Sahni, Capítulo 7, Fig. p.328	Árbol de búsqueda	• Árbol de búsqueda potencial para un problema concreto • Nodos circulares, numerados por orden de generación en búqueda-D • Etiquetas de decisiones en ramas (variable y valor)
SUM OF SUBSETS	Horowitz y Sahni, Capítulo 7, Fig. p.344	Árbol de búsqueda	• Árbol de búsqueda generado para un problema concreto • Nodos rectangulares, con valores de algunas variables • Etiquetas de decisiones en ramas (variable y valor) en árbol izquierdo
THE SUM-OF - SUBSETS PROBLEM	Neapolitan y Naimipour, Capítulo 5, Apartado 1, Figura p. 195	Árbol de búsqueda	• Árbol de búsqueda potencial • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (variable y valor)
THE SUM-OF - SUBSETS PROBLEM	Neapolitan y Naimipour, Capítulo 5, Apartado 1, Figura p. 195,197	Árbol de búsqueda	• Árbol de búsqueda potencial • Nodos circulares, con valor • Etiquetas de decisiones en niveles (variable y valor) y en ramas (valor)
THE SUM-OF -	Neapolitan y Naimipour,	Árbol de búsqueda	• Árbol de búsqueda generado

SUBSETS PROBLEM	Capítulo 5, Apartado 1, Figura p. 197		• Nodos circulares, con valor • Nodo de solución final, sombreado; nodos inválidos, con x inferior • Etiquetas de decisiones en niveles (variable y valor) y en ramas (valor)
14.5 (problema de las factorias)	Martí Oliet *et al.*, Capítulo 14, Figura p. 494	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico) • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos, y arcos suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (algunas con identificador, tipo ojo de pez)
14.21 (problema de los sellos)	Martí Oliet *et al.*, Capítulo 14, Figura p. 494	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico) • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos, y arcos suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (algunas con identificador, tipo ojo de pez)
14.21 (problema de los sellos)	Martí Oliet *et al.*, Capítulo 14, Figura p. 495	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico) • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos, y arcos suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (algunas con identificador, tipo ojo de pez)
14.11 (problema del reparto)	Martí Oliet *et al.*, Capítulo 14, Figura p. 471	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (variable y valor)
The knapsack problem (3)	Brassard y Bratley, Capítulo 9 Apartado 6, Fig. p.307	Árbol de búsqueda	• Árbol de búsqueda generado (problema concreto) • Nodos rectangulares, con valores de algunas variables • Sin etiquetas
KNAPSACK PROBLEM	Horowitz y Sahni, Capítulo 7, Figura p. 354	Árbol de búsqueda	• Árbol de búsqueda generado (problema concreto) • Nodos circulares, con valores de algunas variables (a veces dentro, a veces

			fuera) • Etiquetas de decisiones en algunas ramas (variables y valores)
KNAPSACK PROBLEM	Horowitz y Sahni, Capítulo 7, Figura p. 358	Árbol de búsqueda	• Porción de árbol de búsqueda generado (problema concreto) • Nodos circulares, con valores de algunas variables (a veces dentro, a veces fuera) • Etiquetas de decisiones en algunas ramas (variables y valores)
14.20 (problema de las canciones)	Martí Oliet *et al.*, Capítulo 14, Fig. p.492	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico) • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas (en el primer nivel, con identificador)
0-1 Knapsack Problem	Neapolitan y Naimipour, Capítulo 5, Apartado 1, Fig. p.210	Árbol de búsqueda	• Árbol de búsqueda generado (problema concreto) • Nodos circulares, con valores de algunas variables dentro • Nodo de solución óptima, sombreado; otros nodos, con x inferior • Etiquetas de decisiones en niveles (variable y valor) y en ramas (valor)
0/1 Knapsack	Sahni, Capítulo 21, Fig. p.837	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico para tamaño 3) • Nodos circulares, con etiqueta interna alfabética por niveles • Etiquetas de decisiones en ramas
14.18 (problema de los comensales)	Martí Oliet *et al.*, Capítulo 14,	Árbol de búsqueda	• Árbol de búsqueda potencial (genérico) • Árbol abreviado, con subárboles comprimidos en triángulo o suprimidos y ramas suprimidas con puntos suspensivos (quizá ojo de pez) • Nodos circulares, vacíos • Etiquetas de decisiones en ramas, a veces con identificador (tipo ojo de pez)
14.22 (problema de los envases)	Martí Oliet *et al.*, Capítulo 14, Figura p. 499	Árbol de búsqueda	• Árbol de búsqueda potencial (problema concreto) • Nodos circulares, con valores • Etiquetas de decisiones en ramas (en los dos primeros niveles, con identificador)

4 Conclusiones

Se ha presentado el resultado de un trabajo de revisión bibliográfica, con las carencias o inexactitudes que puede tener. Se han incluido las figuras con que se ilustran los problemas en los libros de texto así como el código que los resuelven. Los problemas se han presentado agrupados en cuatro clases: de juegos y estrategia (JE), grafos (G), otros problemas de decisión (OD) y de optimización (PO).

El estudio es susceptible de ser ampliado con nuevos problemas o libros de texto. Así, los resultados serán utilizados en la línea de trabajo sobre ayudantes interactivos en la que actualmente estamos trabajando.

Agradecimientos. Este trabajo se ha financiado con el proyecto TIN2011-29542-C02-01 del Ministerio de Innovación y Ciencia.

Referencias

1. M. H. Alsuwaiyel: Algorithms Design Techniques and Analysis. World Scientific (1999)
2. J. Gonzalo Arroyo y M. Rodríguez Artacho: Esquemas algorítmicos. Enfoque metodológico y problemas resueltos. UNED (2000)
3. S Baase y A Van Gelder: Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley Longman (2000)
4. G. Brassard y P. Bratley: Fundamentos de algoritmia. Prentice-Hall (1996)
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein: Introduction to Algorithms. The MIT Press, 2ª ed (2001)
6. M. T. Goodrich y R. Tamassia: Data Structures and Algorithms in Java. John Wiley & Sons, 2ª ed. (2001)
7. E. Horowitz y S. Sahni: Fundamentals of Computer Algorithms. Pitman (1978)
8. R. Neapolitan y K. Naimipour: Foundations of Algorithms. Jones and Bartlett (1997)
9. N. Martí Oliet, Y. Ortega y J. A. Verdejo: Estructuras de datos y métodos algorítmicos: ejercicios resueltos. Pearson (2004)
10. I. Parberry: Problems on Algorithms. Prentice Hall (1995)
11. S. S. Tseng R. C. T. Lee, R. C. Chang e y Y. T. Tsai: Introducción al diseño y análisis de algoritmos. McGraw-Hill (2005)
12. S. Sahni: Data Structures, Algorithms, and Applications in Java. Silicon Press (2005)
13. R. Sedgewick: Algorithms in Java. Addison-Wesley (2002)
14. S. Skiena: The Algorithm Design Manual. Springer-Verlag (1998)