VUELTA ATRÁS (BACKTRACKING)

INTRODUCCIÓN

- Hay algoritmos que permiten resolver problemas de optimización cuya solución se determina a través del recorrido de un grafo.

- En esos casos, el recorrido del grafo se realiza con un criterio o un fin concreto: encontrar el circuito hamiltoniano de coste mínimo o el camino mínimo entre el nodo origen y el nodo destino.

- Ejemplo: Embarcaderos.

  Dado un grafo, hay que determinar el camino mínimo

  entre el nodo 1 y el nodo n.

INTRODUCCIÓN

- Pero, en otras ocasiones, los nodos del grafo se utilizan para representar las posibles soluciones de un problema.

- La búsqueda de la solución se obtiene a través de un recorrido exhaustivo por todos los nodos del grafo. Esta es la idea central de los algoritmos de Vuelta Atrás o Backtracking.

- Lógicamente, esta estrategia podría ser muy ineficiente si la búsqueda se realiza a ciegas (sin ningún criterio) y el grafo es grande. Para aminorar esta dificultad, debemos asegurarnos de que ese recorrido se hace adecuadamente (por ejemplo, evitando la posibilidad de visitar un nodo dos veces, pues esa solución ya habría sido examinada).

INTRODUCCIÓN

- En este contexto se incluye la metodología de Vuelta Atrás o Backtracking.
- En este caso, el grafo a recorrer corresponde a un ÁRBOL, es decir, es un grafo conexo y sin ciclos.
- El recorrido se entiende en el sentido de que pretendemos visitar cada nodo y hacerlo de una manera ordenada y sistemática.
- No queremos realizar más visitas de las estrictamente necesarias y no queremos proseguir el recorrido más allá de ningún nodo para el que sabemos que sus descendientes no pueden contener la solución que buscamos.

INTRODUCCIÓN

- El algoritmo de Vuelta Atrás o Backtracking es una variante de otro más general: el algoritmo de recorrido en profundidad de un grafo.

- En este tipo de recorrido (recorrido en profundidad) se da prioridad a los nodos más profundos en el árbol, de tal forma que éstos se expanden primero que otros nodos menos profundos.

- El árbol en cuestión es sólo VIRTUAL y se utiliza para REPRESENTAR el espacio de soluciones del problema. Esto quiere decir, que no es necesario crear ninguna estructura de representación de árboles.

VUELTA ATRÁS: ÁMBITO DE APLICACIÓN

Este método general de resolución de problemas se aplica en las siguientes circunstancias:

- Vuelta Atrás o Backtracking es una técnica general de resolución de problemas, aplicable a problemas de optimización.
- También nos proporciona el conjunto de todas las soluciones que satisfacen ciertas restricciones o una solución (la primera) que satisfaga ciertas restricciones.
- Se trata de un método sistemático que realiza una búsqueda exhaustiva de las soluciones, pero esta BÚSQUEDA NO es CAÓTICA sino ORGANIZADA, por lo que la mayoría de las veces evita tener que examinar todas las posibilidades, reduciendo así la complejidad.

VUELTA ATRÁS: ÁMBITO DE APLICACIÓN

- Para aplicar Backtracking, la solución debe poder expresarse como una secuencia de decisiones (x_1, x_2, …, x_n) donde cada x_i representa una decisión y se elige de entre un conjunto S_i de valores posibles. Los S_i pueden ser iguales, es decir, en todas las decisiones tendremos las mismas alternativas, o no.

- Se pretende obtener o bien una solución factible o todas las factibles, o de entre todas las factibles, la óptima. Existe por tanto una FUNCIÓN OBJETIVO y unas RESTRICCIONES.

- Debe ser posible determinar si una secuencia de decisiones (completa o parcial) es factible, analizando si verifica las restricciones del problema.

VUELTA ATRÁS: RESTRICCIONES

Una secuencia debe satisfacer una serie de restricciones que podemos agrupar en dos categorías:

- Restricciones explícitas: Son las reglas que definen el dominio S_i de la decisión x_i.
- Restricciones implícitas: Son las reglas que definen las relaciones que debe haber entre las distintas x_i de una misma secuencia de decisiones, también denominada tupla.

MOCHILA 0/1: RESTRICCIONES

Mochila con una capacidad C, n objetos, P[1..n] pesos de los n objetos y B[1..n] beneficios de los n objetos.

Secuencia de decisiones:

$$< x_1, x_2, \ldots, x_n >$$

Restricciones explícitas:

$$(\forall i)(x_i \in \{0,1\}: 1 \leq i \leq n)$$

Restricciones implícitas:

$$\sum_{i=1}^{n} x_i P_i \leq C$$

VUELTA ATRÁS: RESTRICCIONES

En base a la clasificación de las restricciones se definen los siguientes conceptos.-

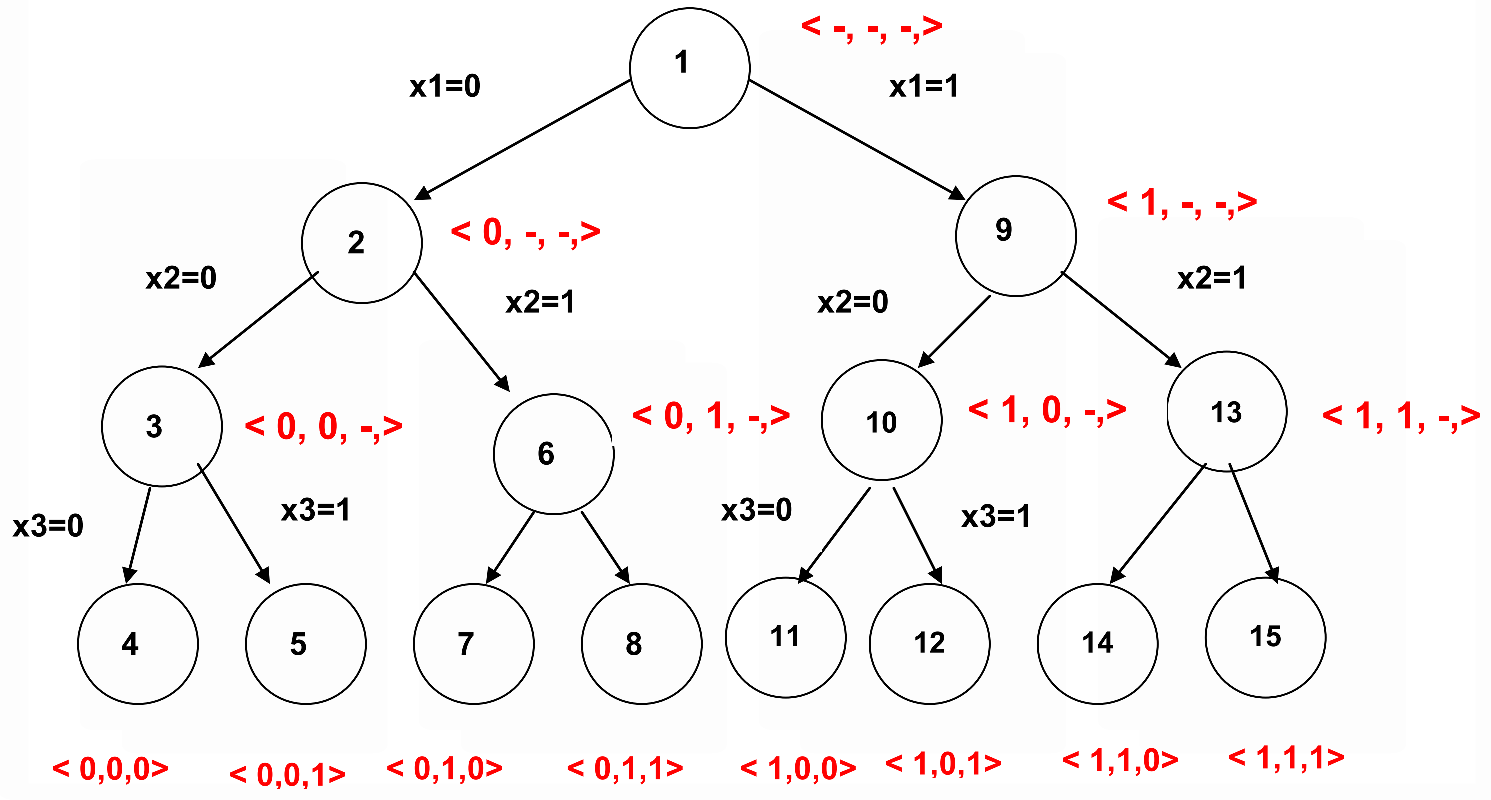
- ESPACIO DE ESTADOS POSIBLES: conjunto de todas las secuencias, parciales y completas, que cumplen restricciones explícitas.
- ESPACIO DE SOLUCIONES POSIBLES: conjunto de todas las secuencias completas que cumplen restricciones explícitas.
- ESPACIO DE SOLUCIONES FACTIBLES: conjunto de todas las secuencias completas que cumplen restricciones explícitas e implícitas.

VUELTA ATRÁS: ÁRBOL DE BÚSQUEDA

La aplicación del método se facilita organizando la búsqueda según el recorrido de un árbol que representa una instancia particular del problema y que se llama ÁRBOL DE BÚSQUEDA.

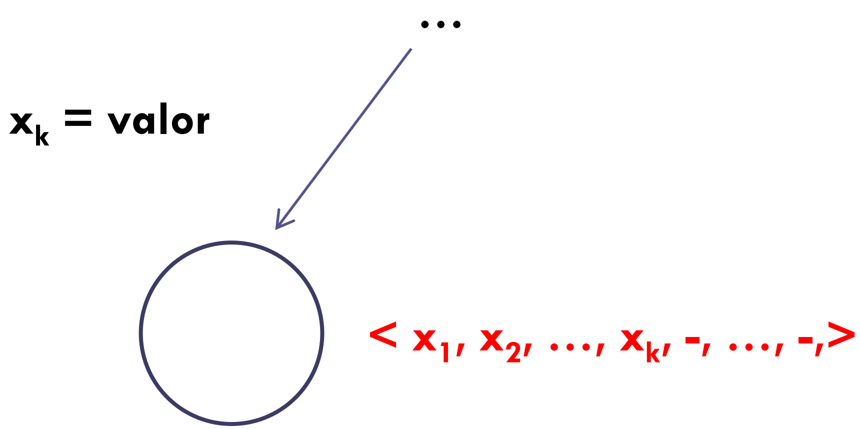
Veamos un ejemplo: Problema de la mochila 0/1

MOCHILA 0/1: ÁRBOL DE BÚSQUEDA



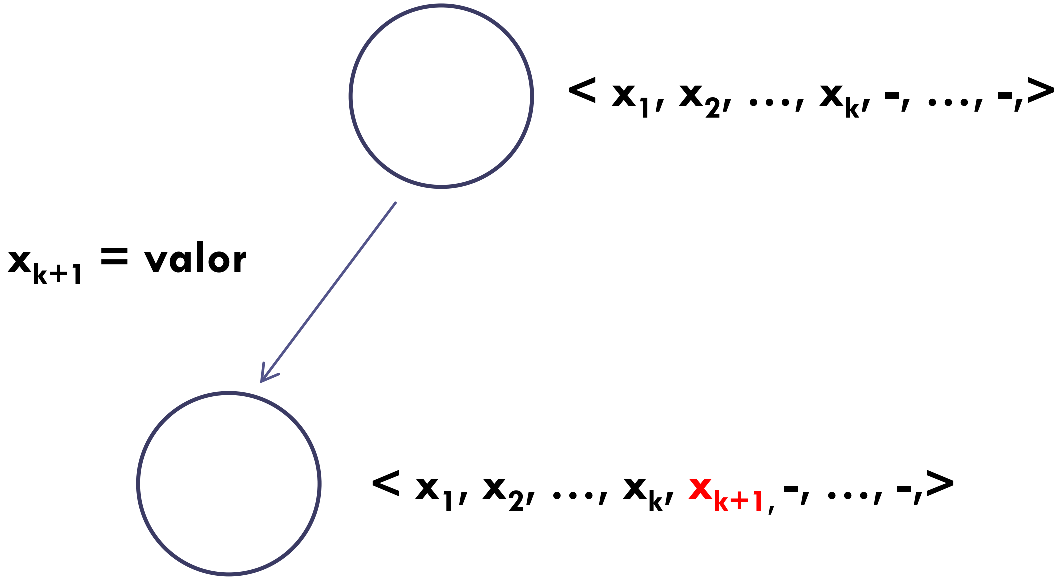
VUELTA ATRÁS: FUNCIONAMIENTO DEL MÉTODO

En un determinado momento, el algoritmo se encontrará en un cierto nivel k, con una solución parcial $(x_1, \ldots, x_k)$.



SITUACIÓN 1: La tupla parcial $(x_1, \ldots, x_k)$ es factible

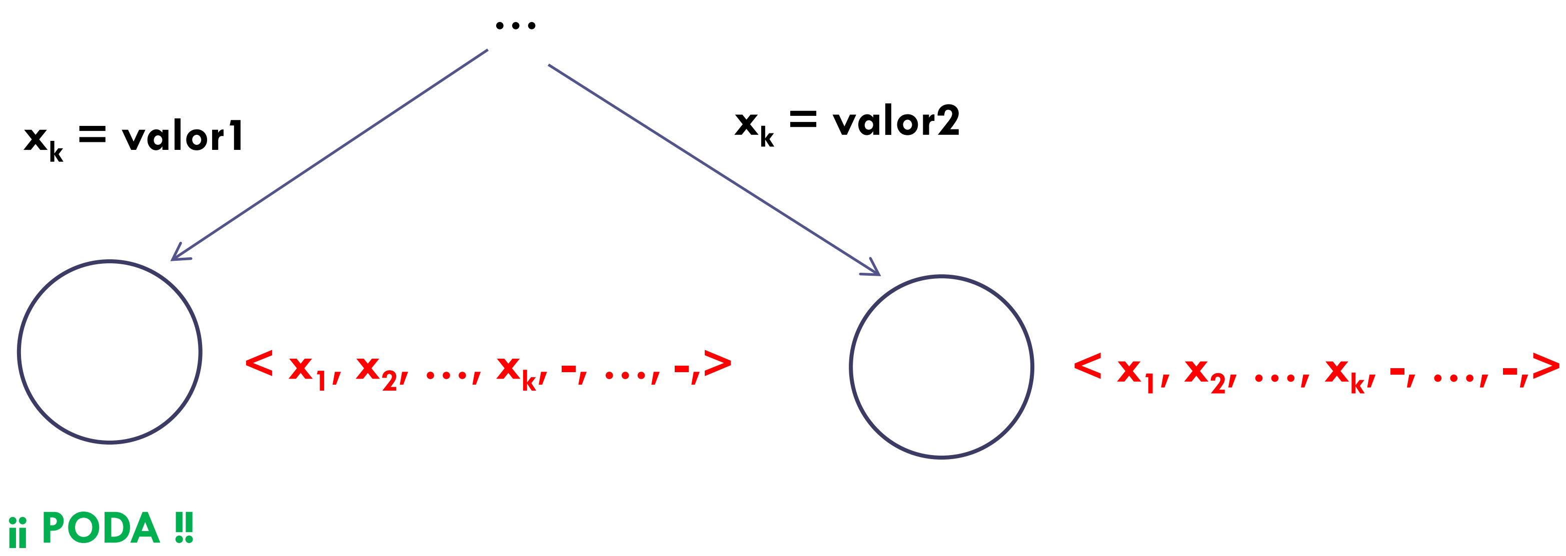
entonces se puede añadir un nuevo elemento a la tupla y asignando un valor a x_{k+1}. → AMPLIAR LA TUPLA ("BAJAR UN NIVEL EN EL ÁRBOL")



SITUACIÓN 2: La tupla parcial (x_1, …, x_k) NO es FACTIBLE

entonces se realiza una PODA.

Esto quiere decir que a x_k se le asigna otro valor de entre las posibles alternativas → "IR AL HERMANO"

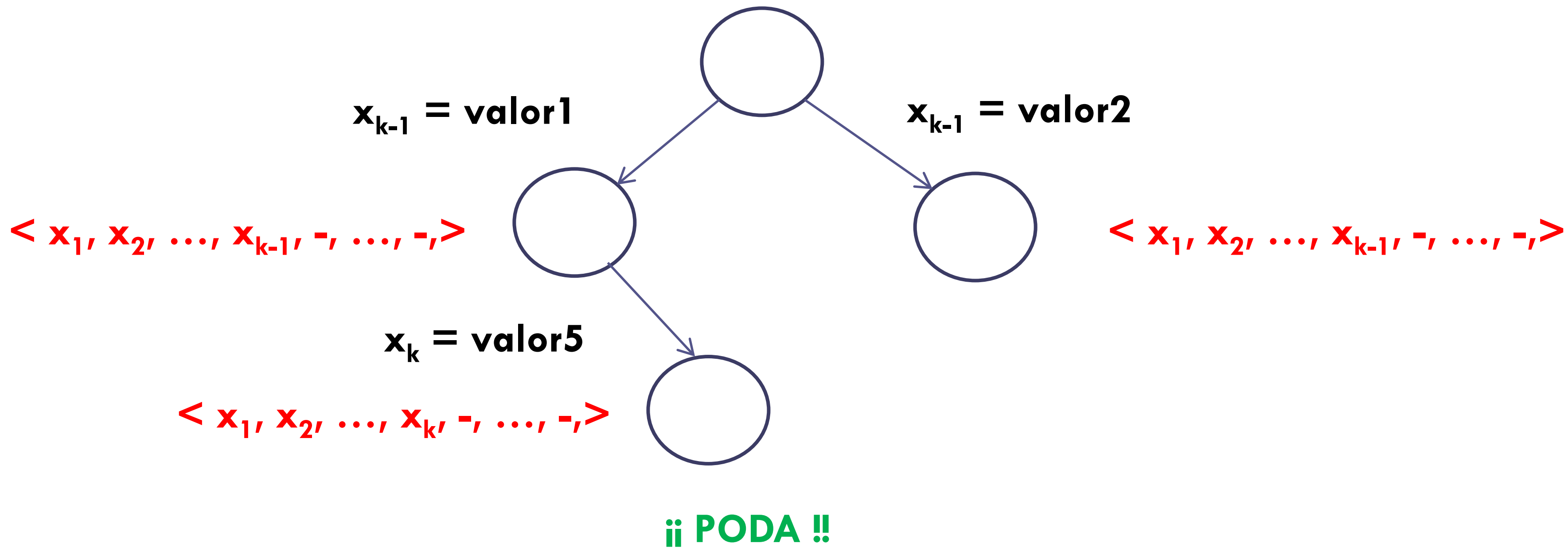


SITUACIÓN 3: La tupla parcial $(x_1, \ldots, x_k)$ NO es FACTIBLE

pero no quedan más alternativas para x_k, entonces al realizar la PODA,

hay que analizar la decisión x_{k-1} con alternativas pendientes →

→ DAR MARCHA ATRÁS ("SUBIR UN NIVEL")



SITUACIÓN 3

Si en x_{k-1} no hubiera más alternativas pendientes, se volvería a x_{k-2} y así sucesivamente.

Cuando se consiga replantear una decisión anterior se vuelve paso a paso hacia adelante.

VUELTA ATRÁS: ESQUEMAS

Existen tres esquemas para obtener:

- TODAS LAS SOLUCIONES FACTIBLES
- UNA SOLUCIÓN FACTIBLE
- SOLUCIÓN ÓPTIMA

VUELTA ATRÁS: ESQUEMA "TODAS LAS FACTIBLES"


```
Procedimiento Backtracking_TODAS (D:datos_problema; k:entero; e/s x:tupla)
/*  k es un parámetro de nivel que representa:
                1.- el grado de recursión
                2.- el nivel en el árbol
                3.- la decisión en curso=posición en la tupla (secuencia de decisiones)
        llamada inicial al procedimiento: Backtracking_TODAS (D,1,x)   */
preparar_recorrido_nivel_k;
mientras ∃_hermano_nivel_k hacer
     siguiente_hermano_nivel_k;
     opción
           solución(D,x,k) ∧ correcto(D,x,k): tratar(x);
           ¬solución(D,x,k) ∧ correcto(D,x,k): Backtracking_TODAS(D, k+1, x);
           en otro caso: nada
     fopción
fmientras
fprocedimiento
```

RESTRICCIONES EXPLÍCITAS EN EL ESQUEMA


```
Procedimiento Backtracking_TODAS (D:datos_problema; k:entero; e/s x:tupla)
/* operación que inicializa la k-ésima componente de x */
preparar_recorrido_nivel_k;
/* cierto si quedan alternativas pendientes para la decisión en curso xk , falso si no */
mientras ∃_hermano_nivel_k hacer
    /* devuelve nuevo posible valor (el siguiente) para decisión en curso xk */
    siguiente_hermano_nivel_k;
    opción
        solución(D,x,k) ∧ correcto(D,x,k): tratar(x);
        ¬solución(D,x,k) ∧ correcto(D,x,k): Backtracking_TODAS(D,k+1,x);
    fopción
fmientras
fprocedimiento
```

EJEMPLO: MOCHILA 0/1

Restricciones explícitas: $(\forall i)(\ x_i \in \{0,1\}:\ 1 \le i \le n\ )$

Procedimiento Backtracking_TODAS (P[1..n]:vector de enteros; n, C, k:entero; e/s x:tupla)

preparar_recorrido_nivel_k; → x[k] = - 1;

mientras ∃_hermano_nivel_k hacer → mientras x[k] < 1 hacer

siguiente_hermano_nivel_k; → x[k] = x[k] +1;

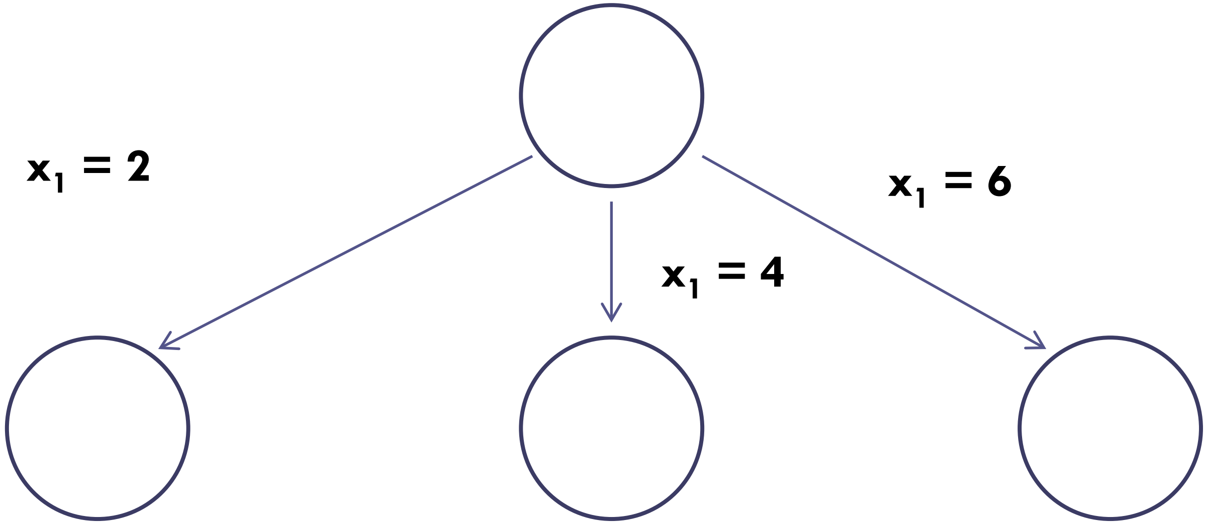
…

fmientras

fprocedimiento

EJEMPLO

Restricciones explícitas: $(\forall i)(x_i \in \{2,4,6\}: 1 \leq i \leq n)$



EJEMPLO

Restricciones explícitas: $(\forall i)(x_i \in \{2,4,6\}: 1 \leq i \leq n)$

Procedimiento Backtracking_TODAS (…)

…

preparar_recorrido_nivel_k; → x[k] = ¿?

mientras ∃_hermano_nivel_k hacer → mientras x[k] < ¿? hacer

    siguiente_hermano_nivel_k; → x[k] = x[k] + ¿?

    …

fmientras

fprocedimiento

EJEMPLO

Restricciones explícitas: $(\forall i)(x_i \in \{2,4,6\}: 1 \leq i \leq n)$

```
Procedimiento Backtracking_TODAS (…)
…
preparar_recorrido_nivel_k;  → x[ k ] = 0
mientras ∃_hermano_nivel_k hacer   →  mientras x[ k ] < 6 hacer
        siguiente_hermano_nivel_k; →  x[ k ] = x[ k ] + 2

     …
fmientras

fprocedimiento
```

FUNCIÓN SOLUCIÓN EN EL ESQUEMA


```
Procedimiento Backtracking_TODAS (D:datos_problema; k:entero; e/s x:tupla)
preparar_recorrido_nivel_k;
mientras ∃_hermano_nivel_k hacer
    siguiente_hermano_nivel_k;
    opción
            solución(D,x,k) ∧ correcto(D,x,k): tratar(x);
            ¬solución(D,x,k) ∧ correcto(D,x,k): Backtracking_TODAS(D,k+1,x);
    fopción
fmientras
fprocedimiento
```

Nota.- la función solución devuelve cierto si x es una solución completa (nodo hoja), falso en caso contrario

EJEMPLO: MOCHILA 0/1

SECUENCIA DE DECISIONES: $< x_1, x_2, \ldots, x_n >$

Procedimiento Backtracking_TODAS (P[1..n]:vector de enteros, n, C:entero; k:entero; e/s x:tupla)

...

opción

$k=n \land$ correcto(D,x,k): tratar(x);

$k<n \land$ correcto(D,x,k): Backtracking_TODAS(P, n, C, k+1, x);

fopción

...

fprocedimiento

RESTRICCIONES IMPLÍCITAS EN EL ESQUEMA


```
Procedimiento Backtracking_TODAS (D:datos_problema; k:entero; e/s x:tupla)
preparar_recorrido_nivel_k;
mientras ∃_hermano_nivel_k hacer
    siguiente_hermano_nivel_k;
    opción
            solución(D,x,k) ∧ correcto(D,x,k): tratar(x);
            ¬solución(D,x,k) ∧ correcto(D,x,k): Backtracking_TODAS(D, k+1, x);
    fopción
fmientras
fprocedimiento
```

Nota.- la función correcto devuelve cierto si la solución (parcial/completa) no incumple las restricciones implícitas, falso en caso contrario.

VUELTA ATRÁS: ESQUEMA "UNA FACTIBLE"


```
Procedimiento Backtracking_UNA (D:datos_problema; k:entero; e/s x:tupla; e/s flag:booleano)

/* llamada inicial al procedimiento: Backtracking_UNA (D, 1, x, verdadero) */

preparar_recorrido_nivel_k;
mientras ∃_hermano_nivel_k ∧ flag hacer
    siguiente_hermano_nivel_k;
    opción
            solución(D,x,k) ∧ correcto(D,x,k): tratar(x); flag=falso;
            ¬solución(D,x,k) ∧ correcto(D,x,k): Backtracking_UNA(D, k+1, x, flag);
    fopción
fmientras
fprocedimiento
```

VUELTA ATRÁS: ESQUEMA "ÓPTIMA"


```
Procedimiento Backtracking_OPTIMA (D:datos_problema; k:entero; e/s x, x_mejor:tupla; e/s v_mejor:valor)
preparar_recorrido_nivel_k;
mientras ∃_hermano_nivel_k hacer
      siguiente_hermano_nivel_k;
      opción
            solución(D,x,k) ∧ correcto(D,x,k): si valor(D,x,k) > v_mejor entonces
                                                      x_mejor=x;
                                                      v_mejor=valor(D,x,k);
                                          fsi
            ¬solución(D,x,k) ∧ correcto(D,x,k): Backtracking_OPTIMA (D, k+1, x, x_mejor, v_mejor);
      fopción
fmientras
fprocedimiento
```

Notas.- la función valor calcula el valor asociado a la secuencia de decisiones (función objetivo)

Para problema maximizar "valor(D,x,k) >v_mejor", para problema minimizar "valor(D,x,k) < v_mejor"

MOCHILA 0/1: TODAS LAS FACTIBLES


```
Procedimiento Mochila01_TODAS (P[1..n]:vector de enteros, n:entero; C:entero, k:entero, e/s x: tupla)

/* Llamada inicial: Mochila01_TODAS (P, n, C, 1, x) */

x[ k ] = - 1;
mientras x[ k ] < 1 hacer
      x[ k ] = x[ k ] + 1;
      opción
                 k = n ∧ PESO(P, x, k) ≤ C: imprimir(x);
                 k < n ∧ PESO(P, x, k) ≤ C: Mochila01_TODAS (P, n, C, k+1, x);
      fopción
fmientras
fprocedimiento
```

MOCHILA 0/1: FUNCIÓN AUXILIAR


```
Función PESO (P[1..n]:vector de enteros, x:tupla, k:entero) retorna (p:entero)
var
      i:entero;
      peso_acumulado:entero;
fvar
 i = 0;
peso_acumulado = 0;
mientras (i < k) hacer
      i = i + 1;
      peso_acumulado = peso_acumulado + P[i] * x[i];
fmientras
retorna peso_acumulado
ffunción
```

N-REINAS

El problema consiste en encontrar una forma de colocar N reinas en un tablero de ajedrez sin que se den jaque (dos reinas se dan jaque si comparten fila, columna o diagonal).

Una primera idea sería construir sistemáticamente todas las formas posibles de colocar N reinas en un tablero y verificar si alguna de ellas es una solución factible. Esta estrategia es impracticable ya que en el caso de 8 reinas, dicha estrategia generaría el siguiente número de posibilidades a ensayar: 4.426.165.368.

La primera mejora que podemos hacer es no poner nunca más de una reina en una fila. Esto reduce la representación del tablero a un vector de 8 elementos, cada uno de los cuales da la posición de la reina dentro de la fila correspondiente. Podemos replantear el problema del siguiente modo: "colocar una reina en cada fila del tablero de ajedrez de forma que no se den jaque".

En este caso el número de posibilidades es $8^8 = 16.777.216$.

N-REINAS

- La solución será una N-tupla

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

donde x_i representa la columna en la que se coloca la i-ésima reina.

- Buscaremos UNA solución factible.

- Restricciones explícitas:

$$(\forall i)\ (x_i \in \{1, 2, \ldots, N\} : 1 \leq i \leq N)$$

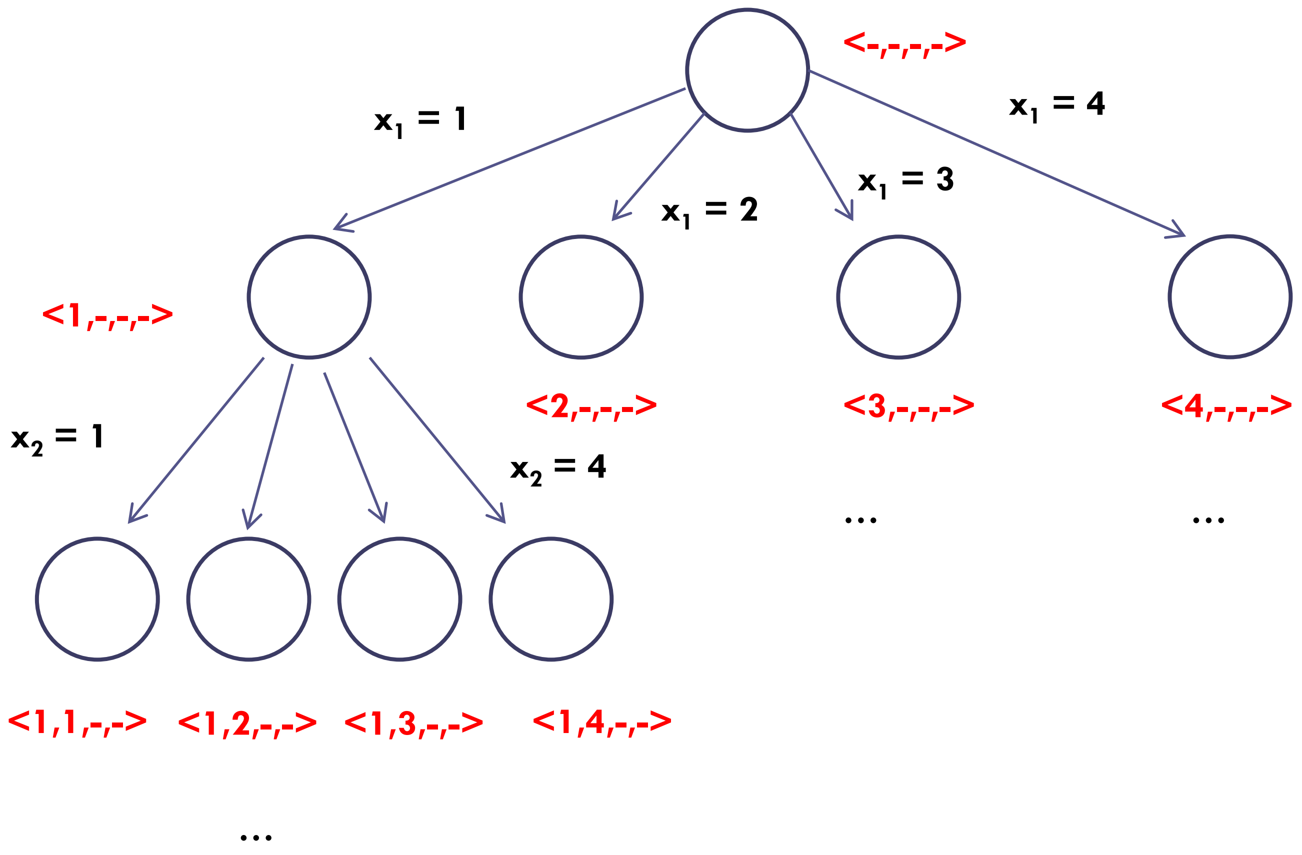
N-REINAS

Reina 1 ➔ $(1, x_1)$ Reina 2 ➔ $(2, x_2)$ …

$$(x_1, x_2, x_3, x_4) = (2, 4, 1, 3)$$

	1	2	3	4
1		X		
2				X
3	X			
4			X	

4-REINAS: ÁRBOL DE BÚSQUEDA



N-REINAS: RESTRICCIONES IMPLÍCITAS

Es posible determinar si una secuencia completa es factible y es posible determinar si una secuencia parcial puede conducir o no a una solución factible a través de las siguientes restricciones implícitas:

- Dos reinas NO pueden estar en la misma columna.
- Dos reinas NO pueden estar en la misma diagonal.

N-REINAS: RESTRICCIÓN IMPLÍCITA 1

Dos reinas i y j están en una misma columna si $x_i = x_j$

Por lo que una solución $(x_1, x_2, \ldots, x_N)$ es factible si cumple lo siguiente

$$(\forall i)\ ((\forall j)\ (\ i \neq j \Rightarrow x_i \neq x_j : 1 \leq j \leq N) : 1 \leq i \leq N)$$

	1	2	3	4	5	6
1						
2		X				
3						
4						
5		X				
6						

N-REINAS: RESTRICCIÓN IMPLÍCITA 2

Dos reinas están en una misma diagonal:

- del tipo "/" si tienen el mismo valor de "fila + columna", esto es, $i + x_i = j + x_j$ y
- del tipo "\" si tienen el mismo valor de "fila - columna", esto es, $i - x_i = j - x_j$

N-REINAS: RESTRICCIÓN IMPLÍCITA 2

Reina1 (1,6) y Reina5 (5,2) están en la misma diagonal del tipo "/", por tanto tienen el mismo valor de fila+columna, veámoslo

$$1 + 6 = 2 + 5 = 7$$

	1	2	3	4	5	6
1						X
2						
3						
4						
5		X				
6						

N-REINAS: RESTRICCIÓN IMPLÍCITA 2

Reina3 (3,2) y Reina5 (5,4) están en la misma diagonal del tipo "\", por tanto tienen el mismo valor de fila-columna, veámoslo

$$3 - 2 = 5 - 4 = 1$$

	1	2	3	4	5	6
1						
2						
3		X				
4						
5				X		
6						

N-REINAS: RESTRICCIÓN IMPLÍCITA 2

Por tanto, dos reinas están en la misma diagonal si ocurre

$$(i + x_i = j + x_j) \vee (i - x_i = j - x_j),$$

o lo que es lo mismo

$$(x_i - x_j = j - i) \vee (x_i - x_j = i - j),$$

lo que equivale a

$$| x_i - x_j | = | i - j |$$

N-REINAS: FUNCIÓN CORRECTO

Utilizaremos una función que llamaremos BUEN_SITIO(x,k) que devuelva:

- Cierto si la k-ésima reina se puede colocar en la posición x[k], es decir, si está en distinta columna y diagonal que las k-1 reinas anteriores.

- Falso en caso contrario.

N-REINAS: UNA FACTIBLE


```
Procedimiento N-reinas (N:entero, k:entero, e/s x:tupla, e/s flag:booleano)

/* Llamada inicial: N-reinas( N, 1, x, verdadero) */

x[ k ] = 0;
mientras (x[k] < N ∧ flag ) hacer
    x[ k ] = x[k] + 1;
    opción
        k = N ∧ BUEN_SITIO(x,k) : imprimir(x); flag=falso;
        k < N ∧ BUEN_SITIO(x,k) : N-reinas(N, k+1, x, flag);
    fopción
fmientras
fprocedimiento
```

N-REINAS: FUNCIÓN CORRECTO


```
Funcion BUEN_SITIO (x:tupla, k:entero) retorna (b:booleano)
var
    i:entero;
    ok:booleano;
fvar
i = 0;
ok = verdadero;
mientras (i < k-1 ∧ ok) hacer
    i = i + 1;
    si ( x[ i ] = x[ k ] ) ∨ ( |x[ i ] – x[ k ] | = | i – k |) entonces
        ok=falso;
    fsi
fmientras
retorna ok
ffunción
```

LABERINTO

Dado un laberinto representado por una matriz cuadrada L de n x n elementos:

Cada L[i][j], con $1 \leq i, j \leq n$, tiene dos posibles valores:

- ´•´ representa una posición abierta y
- ´×´ representa una posición bloqueada.

Se trata de generar todos los caminos que comenzando en L[1][1] terminen en la posición L[n][n].

Para ello se considerarán únicamente dos tipos de movimientos: movimiento hacia la casilla inferior (↓), que representaremos por el valor 1, y movimiento hacia la casilla derecha (→), que representaremos por el valor 2.

No se puede atravesar una posición bloqueada ni salir del laberinto.

LABERINTO

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO

SOLUCIÓN.-

- La solución se puede expresar como una secuencia de decisiones que son los distintos movimientos que nos permiten ir desde la casilla L[1][1] a la L[n][n].

- Una secuencia completa será del tipo $<x_1,x_2, \ldots, x_{2(n-1)}>$ ya que serán necesarios 2(n-1) movimientos para conseguir el objetivo, x_i será 1 o 2 dependiendo del movimiento a realizar.

- TODAS las soluciones factibles.

LABERINTO

RESTRICCIONES EXPLÍCITAS:

$(\forall i)(\ x_i \in \{1,2\}:\ 1 \le i \le 2(n-1)\ )$

Es posible determinar si una solución completa es factible y es posible determinar si una solución parcial puede conducir o no a una solución factible a través de las siguientes RESTRICCIONES IMPLÍCITAS:

- No salirse del tablero
- No atravesar una posición bloqueada

LABERINTO

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10})=(2, 2, 1, 1, 1, 1, 1, 2, 2, 2)$$

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10})=(2, 2, 2, 2, 2, 1, 1, 1, 1, 1)$$

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO

$$(X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}) = (2, 1, 2, 1, 1, 1, 1, 2, 2, 2)$$

	1	2	3	4	5	6
1	•	•	•	•	•	•
2	X	•	•	X	X	•
3	X	X	•	X	X	•
4	•	X	•	X	X	•
5	•	X	•	X	X	•
6	•	X	•	•	•	•

LABERINTO


```
procedimiento LABERINTO (L[1..n][1..n]:matriz de enteros, k:entero, e/s x:tupla)

/* Llamada inicial: LABERINTO (L, 1, x) */

x[ k ] = 0;
mientras x[ k ] < 2 hacer
    x[ k ] = x[ k ] + 1;
    opción
        k = 2(n-1) ∧ MOVIMIENTO_CORRECTO(L, k, x): imprimir(x);
        k < 2(n-1) ∧ MOVIMIENTO_CORRECTO(L, k, x): LABERINTO(L, k+1, x);
    fopción
fmientras

fprocedimiento
```

LABERINTO


```
funcion MOVIMIENTO_CORRECTO (L[1..n][1..n]:matriz de enteros, k:entero, x:tupla) retorna (b:booleano)
  var
    i, fila, columna: entero
  fvar
  i = 1, fila = 1, columna = 1;
  mientras i ≤ k hacer
     si x[ i ] = 1 entonces     fila = fila + 1
        si no columna = columna + 1
     fsi
     i = i + 1
  fmientras
  si (fila ≤ n) ∧ (columna ≤ n) ∧ (L [fila][columna]= ´•´) entonces retorna cierto
      si no retorna falso
  fsi
ffunción
```

COLOREADO DE UN GRAFO

Dado un grafo no orientado G = (V, E) con V = { 1, 2, …, n }, determinar todas las formas posibles en las que pueden pintarse los nodos del grafo, de modo que no haya dos nodos adyacentes del mismo color y no se usen más de M colores.

Se representará el grafo mediante una matriz de adyacencias L con valores tales que:

- L[i][j] = 1 indicará que los nodos i y j son adyacentes y
- L[i][j] = 0 indicará que los nodos i y j no son adyacentes

COLOREADO DE UN GRAFO

SOLUCIÓN.-

- La solución se puede expresar como una secuencia de decisiones,

  $<x_1, x_2, \ldots, x_n>$

  siendo n el número de nodos de G, y siendo x_i el color con el que se va a pintar el nodo i.

- TODAS las soluciones factibles.

COLOREADO DE UN GRAFO

RESTRICCIONES EXPLÍCITAS:

$$(\forall i)(\ x_i \in \{\ 1,\ 2,\ \ldots,\ M\ \} : 1 \leq i \leq n\ )$$

Es posible determinar si una solución completa es factible y es posible determinar si una solución parcial puede conducir o no a una solución factible a través de la siguiente **RESTRICCIÓN IMPLÍCITA**:

No debe haber nodos adyacentes con el mismo color, es decir,

$$(\forall i)\ ((\forall j)\ (\ L[\ i\ ][\ j\ ]=1 \Rightarrow x_i \neq x_j : 1 \leq j \leq n) : 1 \leq i \leq n)$$

COLOREADO DE UN GRAFO


```
procedimiento COLOREAR (L[1..n][1..n]:matriz de enteros, M:entero, k:entero, e/s x:tupla)

/* Llamada inicial: COLOREAR (L, M, 1, x) */

x[ k ] = 0;
mientras x[ k ] < M hacer
    x[ k ] = x[ k ] + 1;
    opción
            k = n ∧ COLOR_CORRECTO(L, k, x): imprimir(x);
            k < n ∧ COLOR_CORRECTO(L, k, x): COLOREAR (L, M, k+1, x);
    fopción
fmientras
fprocedimiento
```

COLOREADO DE UN GRAFO


```
funcion COLOR_CORRECTO (L[1..n][1..n]:matriz de enteros, k:entero, x:tupla)
                                                        retorna (b:booleano)
var
    i:entero;
    ok:booleano;
fvar
i = 0;
ok = cierto;
mientras ( i < k-1  ∧ ok ) hacer
    i = i + 1;
    si ( x[ i ] = x[ k ] ∧  L[ i ][ k ] = 1)  entonces ok = falso  fsi
fmientras
retorna ok
ffunción
```

VIAJANTE DE COMERCIO

Sea $G = (V, A)$ un grafo orientado con $V = \{1, 2, \ldots, n\}$.

Llamaremos L[i][j] al peso del arco (i, j) (entero no negativo) siendo

L[i][j] = ∞ si no existe dicho arco.

Se trata de encontrar todos los circuitos que comiencen y terminen en el mismo vértice y pasen exactamente una vez por cada uno de los vértices restantes (circuito hamiltoniano)

VIAJANTE DE COMERCIO

SOLUCIÓN.-

- La solución se puede expresar como una secuencia de decisiones,

$$<x_1, x_2, \ldots, x_n>$$

siendo n el número de nodos de G, siendo x_i el nodo visitado en i-ésimo lugar. Al tratarse de recorridos cíclicos se puede establecer el primer nodo como nodo de partida sin perder soluciones. Siguiendo este convenio, fijaremos $x_1=1$ y se define la solución desde x_2 hasta x_n.

- TODAS las soluciones factibles.

VIAJANTE DE COMERCIO

RESTRICCIONES EXPLÍCITAS:

$$(\forall i)(\ x_i \in \{2, \ldots, n\}: 2 \leq i \leq n\ )$$

Es posible determinar si una solución completa es factible y es posible determinar si una solución parcial puede conducir o no a una solución factible a través de las siguientes **RESTRICCIONES IMPLÍCITAS**:

- No debe haber nodos repetidos.
- Todo nodo debe ser adyacente a su antecesor en la solución, y en el caso del último, éste además debe ser adyacente al primero.

VIAJANTE DE COMERCIO


```
procedimiento VIAJANTE (L[1..n][1..n]:matriz de enteros, k:entero, e/s x:tupla)

/* Se inicializa adecuadamente x₁ y k:   x[1]=1 y k=2
                                   Llamada inicial:  VIAJANTE ( L, 2, x) */

x[ k ] = 1;
mientras x[k] < n hacer
      x[ k ] = x[ k ] + 1;
      opción
            k = n ∧ L[x[n-1]][x[n]] ≠ ∞ ∧ NO_REPETIDO(k,x) ∧ L[x[n]][x[1]] ≠ ∞: imprimir(x);
            k < n ∧ L[x[k-1]][x[k]] ≠ ∞ ∧ NO_REPETIDO(k,x): VIAJANTE (L, k+1, x);
      fopción
fmientras
fprocedimiento
```

VIAJANTE DE COMERCIO


```
funcion NO_REPETIDO ( k:entero, x:tupla ) retorna (b:booleano)
var
      i:entero;
      no_repetido:booleano;
fvar

i = 0;
no_repetido = verdadero;
mientras ( i < k-1 ∧ no_repe) hacer
      i = i + 1;
      si (x[ i ] = x[ k ]) entonces no_repetido = falso fsi
fmientras
retorna no_repetido;
ffunción
```

ESQUEMA ITERATIVO “TODAS LAS FACTIBLES”


```
procedimiento Backtracking_TODAS ( D : datos_problema )
var   x:tupla;  k:entero; fvar
k = 1;
preparar_recorrido_nivel_k;
mientras k > 0 hacer
      si ∃_hermano_nivel_k entonces
         siguiente_hermano_nivel_k;
           opción
                solución(D,k,x) ∧ correcto(D,k,x): tratar(x);
                ¬solución(D,k,x) ∧ correcto(D,k,x): k = k + 1; preparar_recorrido_nivel_k;
                en otro caso:nada
            fopción
       sino k = k - 1;
       fsi
fmientras
fprocedimiento
```

ESQUEMA ITERATIVO "UNA FACTIBLE"


```
procedimiento Backtracking_UNA (D:datos_problema)
var
        x:tupla; k:entero; flag:booleano;
fvar
k = 1;
flag = cierto;
preparar_recorrido_nivel_k;
mientras k > 0 ∧ flag hacer
        si ∃_hermano_nivel_k entonces
             siguiente_hermano_nivel_k;
                 opción
                          solución(D,k,x) ∧ correcto(D,k,x): tratar(x); flag = falso;
                         ¬solución(D,k,x) ∧ correcto(D,k,x): k = k + 1; preparar_recorrido_nivel_k;
                          en otro caso: nada
                  fopción
        sino k = k - 1;
        fsi
fmientras
fprocedimiento
```

ESQUEMA ITERATIVO "LA ÓPTIMA"


```
procedimiento Backtracking_OPTIMA (D:datos_problema)
var x, x_mejor : tupla; k : entero; v_mejor : valor; fvar
k = 1;
v_mejor = 0;
preparar_recorrido_nivel_k;
mientras k > 0 hacer
      si ∃_hermano_nivel_k entonces
            siguiente_hermano_nivel_k;
            opción
                  solución(D,k,x) ∧ correcto(D,k,x): si valor(x,k) > v_mejor entonces     /*  problema maximizar ">", minimizar "<" */
                                                            x_mejor = x;
                                                            v_mejor = valor(x,k)
                                                      fsi
                  ¬solución(D,k,x) ∧ correcto(D,k,x): k = k + 1; preparar_recorrido_nivel_k;
                  en otro caso: nada
            fopción
      si no k = k - 1;
      fsi
fmientras
fprocedimiento
```