

Búsqueda con retroceso (*backtracking*)



Simona Bernardi

Universidad de Zaragoza

Presentación reducida y adaptada de la original de J. Campos

Este documento está sujeto a una licencia de uso Creative Commons. No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.



Resumen



- Método general
- Aplicación
 - Problema de las reinas
 - Ciclos hamiltonianos
 - Atravesar un laberinto
 - El problema de la mochila 0-1



Problema de búsqueda

Búsqueda de la mejor solución o todas las soluciones que **satisfacen a ciertas condiciones**

- Cada solución es el resultado de una secuencia de decisiones
- Existe una función objetivo que debe ser satisfecha por cada selección u optimizada

Tipos de problemas

- Se conoce un criterio óptimo de selección en cada decisión
⇒ Técnica **voraz**
- Se cumple el principio de optimalidad de Bellman
⇒ Técnica de la **programación dinámica**
- Otros en lo que no hay más remedio que **buscar**



Planteamiento del problema

- Buscar todas las soluciones que satisfagan a un predicado P
- Solución: una n -tupla (x_1, \dots, x_n) donde cada $x_i \in C_i$
- Si $|C_i| = t_i$ entonces el tamaño del espacio de soluciones posibles es $t = \prod_{i=1}^n t_i$

Métodos de búsqueda

- **Fuerza bruta**: examinar todas las t tuplas y seleccionar las que satisfacen P
- **Retroceso** (*backtracking*, en inglés): formar cada tupla de manera progresiva, elemento a elemento, comprobando para cada elemento x_i añadido a la tupla que (x_1, \dots, x_i) puede conducir a una tupla completa satisfactoria



Planteamiento del problema II

- Deben existir unos **predicados acotadores** $P_i(x_1, \dots, x_i)$
- Los P_i dicen si (x_1, \dots, x_i) puede conducir a una solución

Diferencia entre fuerza bruta y búsqueda con retroceso

Si se comprueba que $P_i(x_1, \dots, x_i) = \text{false}$, se evita formar las $t_{i+1} \times \dots \times t_n$ tuplas que comienzan por (x_1, \dots, x_i)

- Tipos de restricciones que tienen que cumplir las n -tuplas solución
 - **Explícitas:** describen el conjunto C_i de valores que puede tomar x_i (espacio de soluciones posibles $C_1 \times \dots \times C_n$)
 - **Implícita:** describen las relaciones que deben cumplirse entre los x_i (qué soluciones posibles satisfacen el predicado objetivo P)



Problema de las ocho reinas

- Colocar ocho reinas en un tablero de ajedrez sin que se den jaque
- Dos reinas se **dan jaque** si comparten fila, columna o diagonal

Fuerza bruta: $\binom{64}{8} = 4.426.165.368$

- **Replanteamiento del problema:** “colocar una reina en cada fila del tablero de forma que no se den jaque”
- Para ver si dos reinas se dan jaque basta con ver si comparten columna o diagonal

Representación de una solución

Toda solución del problema puede representarse con una 8-tupla (x_1, \dots, x_8) en la que x_i es la columna en la que se coloca la reina que está en la fila i del tablero



Restricciones

- **Explícitas:** $C_i = \{1, 2, 3, 4, 5, 6, 7, 8\}, 1 \leq i \leq 8$
 - El espacio de soluciones consta de 8^8 (16.777.216) 8-tuplas
- **Implícitas:** No puede haber dos reinas en la misma columna o en la misma diagonal
 - Se deduce que todas las soluciones son permutaciones de la 8-tupla (1, 2, 3, 4, 5, 6, 7, 8)
 - El espacio de soluciones se reduce de 8^8 a $8!$ (40.320) 8-tuplas

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

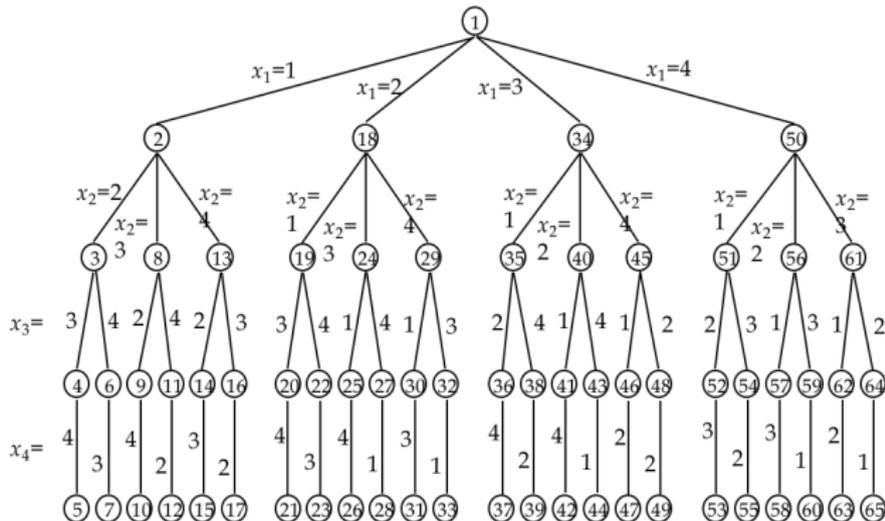
Una solución: (4, 6, 8, 2, 7, 1, 3, 5)



Representación del espacio de soluciones

- **Organización en árbol** del espacio de soluciones
- Ejemplo para el problema de las cuatro reinas (tablero 4×4):

- **Árbol de permutaciones**
- El espacio de soluciones está definido por todos los caminos desde la raíz a cada hoja (hay 4! hojas)





Esquema algorítmico

Notación

(x_1, \dots, x_i)	un camino de la raíz hasta un nodo del árbol
$G(x_1, \dots, x_i)$	el conjunto de los valores posibles de x_{i+1} tales que (x_1, \dots, x_{i+1}) es un camino hasta un nodo del árbol
$A(x_1, \dots, x_{i+1})$	predicado acotador, si es falso \Rightarrow el camino (x_1, \dots, x_{i+1}) no puede extenderse para alcanzar una solución
$R(x_1, \dots, x_{i+1})$	predicado que determina si un camino (x_1, \dots, x_{i+1}) es una solución

- Los candidatos x_{i+1} son los valores de $G(x_1, \dots, x_i)$ tales que $A(x_1, \dots, x_{i+1}) = \text{verdadero}$



Esquema algorítmico

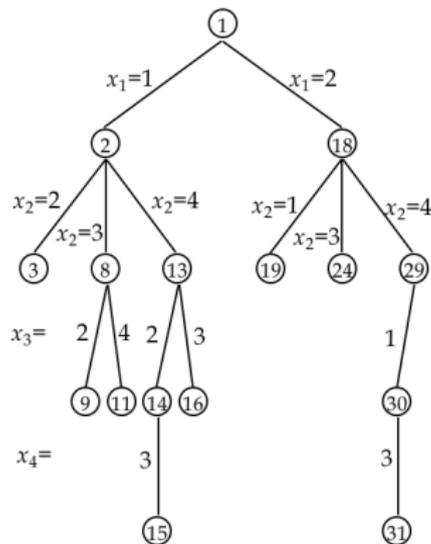
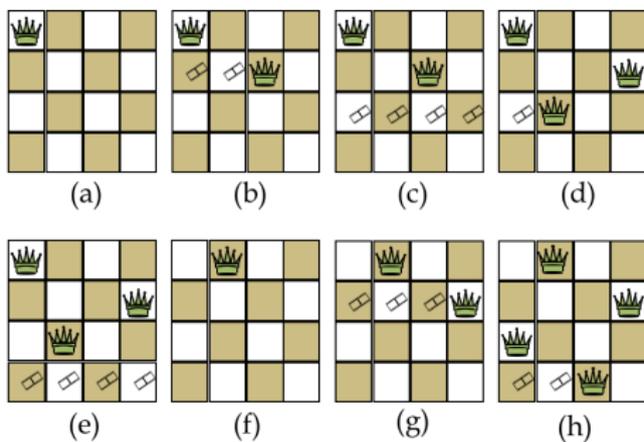
```

1  {Pre:  $1 \leq k \leq n$  y la tupla  $sol[1..k-1]$  es prometedora
2  Post: Guarda todas las soluciones del problema}
3  algoritmo buscarSol(ent k:entero; entsal sol:vector[1..n] de elmt)
4  variable nodo:elmt
5  principio
6  para todo nodo en  $G(sol,1,k-1)$  hacer  $\{G(sol,1,n) = \emptyset\}$ 
7      sol[k] :=nodo;
8      si  $A(sol,1,k)$  entonces
9          si  $R(sol,1,k)$  entonces
10             guardar(sol,1,k)
11         fsi;
12         buscarSol(k+1,sol)
13     fsi
14 fpara
15 fin

```

- La llamada inicial: `buscarSol(1,solución);`

Problema de las cuatro reinas



16 nodos frente a los 31 del árbol completo



Observaciones

- El árbol se construye implícitamente mediante las llamadas recursivas
 - Línea 12 del algoritmo de búsqueda
- El algoritmo no hace llamadas recursivas cuando
 - $k = n + 1 \Rightarrow G(\text{sol}, 1, n) = \emptyset$, o
 - ningún nodo generado por G satisface el predicado A
- **Backtracking**
 - Búsqueda de primero en profundidad y con predicados acotadores
- El algoritmo halla **todas las soluciones** y además éstas pueden ser de longitud variable



Variantes a la solución propuesta

- Limitar el número de soluciones a una sola añadiendo un parámetro booleano de salida que indique si se ha encontrado la solución
- Forzar a que sólo los nodos hoja puedan significar una solución

```
si R(sol,1,k) entonces
    guardar(sol,1,k)
sino
    buscarSol(k+1,sol)
fsi
```

- Resolver problemas de optimización
 - Hay que guardar la mejor solución encontrada hasta el momento
 - Se mejora la eficiencia de la búsqueda si los predicados acotadores permiten eliminar los nodos de los que se sabe que no pueden llevar a una solución mejor que la ahora disponible (**métodos de ramificación y poda**).



Eficiencia del algoritmo de búsqueda

Factores

- 1 El tiempo necesario para generar $sol[k]$
 - 2 El número de elementos solución que satisfacen G
 - 3 El tiempo de ejecución de los predicados acotadores A
 - 4 El número de elementos $sol[k]$ que satisfacen los predicados A
- **Buenos predicados** A reducen mucho el número de nodos generados
 - A menudo buenos predicados precisan mucho tiempo de evaluación
 - Si lo reducen a un solo nodo generado (solución **voraz**): $O(n)$ nodos a generar en total
 - Caso peor: $O(p(n) \times 2^n)$ o $O(p(n)n!)$, con $p(n)$ polinomio
 - **Reestructuración** de las selecciones de forma que $|C_1| < |C_2| < \dots < |C_n|$, y así cabe esperar que se explorarán menos caminos



Estimación a priori del número de nodos generados

- **Idea:** generar un camino aleatorio X en el árbol del espacio de estados
- Sea $x_i \in X$ el nodo en el nivel i y sea m_i el número de hijos de x_i que satisfacen el predicado acotador A
- El siguiente nodo de X se obtiene aleatoriamente de entre esos m_i
- La generación termina en un nodo solución o en un nodo por el que no se puede seguir (ninguno de sus hijos satisfacen el predicado acotador)
- Si los predicados acotadores son **estáticos** y si los nodos de un mismo nivel tiene todos igual grado
 - El número estimado de nodos que se generará con el algoritmo de búsqueda con retroceso es:

$$m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \dots$$



Algoritmo de estimación

```
1  funcion estimación devuelve entero
2  variables k,m,r,card: entero; nodo: elmtto; sol: vector[1..n] de elmtto
3  principio
4      k:=1; m:=1; r:=1;
5      repetir
6          card:=0;
7          para todo nodo en G(sol,1,k-1) hacer
8              sol[k] := nodo;
9              si A(sol,1,k) entonces card:=card+1 fsi
10         fpara
11         si card ≠ 0 entonces
12             r=r*card;
13             m:=m+r;
14             sol[k] := elegirAleatorio(G(sol,1,k-1));
15             k:=k+1;
16         fsi
17     hastaQue R(sol,1,k) or (card=0);
18     devuelve m
19 fin
```



Problema general de las n reinas

Colocar las n reinas en un tablero de dimensiones $n \times n$,
sin que se den jaque

- Cada solución se representa por una n -pla (x_1, \dots, x_n) , en la que x_i es la columna de la i -ésima fila en la que se coloca la i -ésima reina
- El espacio de soluciones se reduce a $n!$ elementos teniendo en cuenta que todas ellas han de ser permutaciones de $(1, 2, \dots, n)$
- Además no deben compartir diagonal



Definición del predicado acotador

- La función `buenSitio` devuelve el valor *verdadero* si la k -ésima reina se puede colocar en $x[k]$, es decir, si está en distinta columna y diagonal que las $k - 1$ reinas anteriores
- Dos reinas están:
 - en la misma diagonal ↗ si tienen el mismo valor de `fila+columna`
 - en la misma diagonal ↘ si tienen el mismo valor de `fila-columna`

Condición de jaque en diagonal

$$(f_1 + c_1 = f_2 + c_2) \vee (f_1 - c_1 = f_2 - c_2) \Leftrightarrow$$

$$(c_1 - c_2 = f_1 - f_2) \vee (c_1 - c_2 = f_2 - f_1) \Leftrightarrow$$

$$|c_1 - c_2| = |f_1 - f_2|$$



Algoritmo del predicado acotador

```

1  {Pre: Las k-1 reinas anteriores ya han sido colocadas
2   Post: buenSitio(k,x) = verdadero ⇔ se puede colocar la kª reina en x[k]}
3  funcion buenSitio(k: entero; x: vector[1..n] de entero) devuelve bool
4  variables i: entero; esBueno: bool
5  principio
6   i:=1; esBueno:=verdadero;
7   mientrasQue (i<k) and esBueno hacer
8     si x[i]=x[k] or abs(x[i]-x[k]) = abs(i-k) entonces
9       esBueno = falso
10    sino
11      i:=i+1
12    fsi
13  fmientrasQue
14  devuelve esBueno
15  fin

```

- ¿Coste de la función? $O(k - 1)$



Algoritmo de búsqueda

Versión recursiva

```

1  {Pre: sol[1..k-1] están bien colocadas
2   Post: escribe todas las soluciones del problema}
3  algoritmo colocarReinas(ent k: entero; entsal sol: vector[1..n] de entero)
4  variables i: entero
5  principio
6     para i:=1 hasta n hacer
7         sol[k]:=i;
8         si buenSitio(k,sol) entonces
9             si k=n entonces escribir(sol)
10            sino colocarReinas(k+1,sol)
11        fsi
12    fsi
13  fpara
14  fin

```

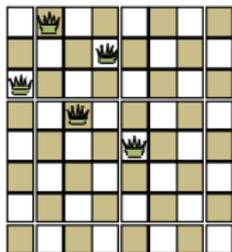
- La llamada inicial: `colocarReinas(1,solución);`



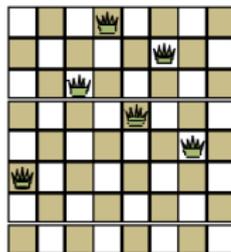
Estimación del coste

Comparación con el enfoque fuerza bruta

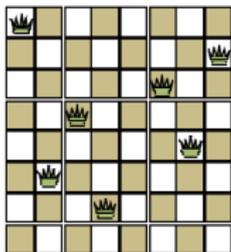
Caso 8 reinas, 5 evaluaciones de la función estimación



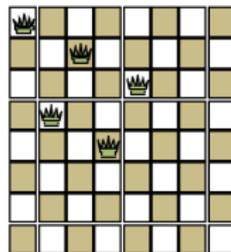
$$(8,5,4,3,2)=1649$$



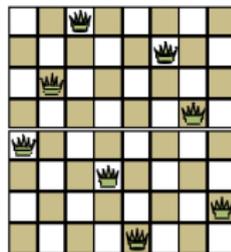
$$(8,5,3,1,2,1)=769$$



$$(8,6,4,2,1,1,1)=1401$$



$$(8,6,4,3,2)=1977$$



$$(8,5,3,2,2,1,1,1)=2329$$

- Con cada elección, se guarda el número de columnas en que es posible colocar la reina y a partir de él se calcula el valor que devuelve la función
- Recordar $m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \dots$



Estimación del coste II

Comparación con el enfoque fuerza bruta

- $\bar{m} = \frac{\sum_{i=1}^5 m_i}{5} = 1645$ (media evaluaciones)
- Tamaño espacio de estados: $1 + \sum_{j=0}^7 \left[\prod_{i=0}^j (8 - i) \right] = 69.281$
- Porcentaje de nodos explorados por el algoritmo (si la estimación es acertada): 2,34 %
 - Estimación optimista: el número de nodos explorados es 2057 (porcentaje de nodos explorados 2,97 %)
- Número de soluciones para 8 reinas: 92
- Mejora adicional
 - Observar que algunas soluciones son simplemente rotaciones o reflexiones de otras
 - Para encontrar soluciones no equivalentes, el algoritmo sólo debe probar con $x_1 = 2, 3, \dots, \lceil n/2 \rceil$



Ciclos hamiltonianos

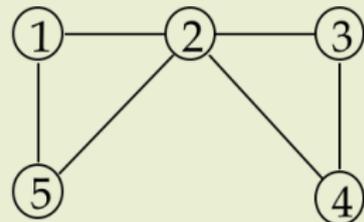
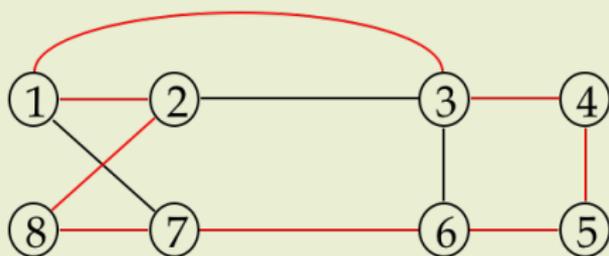
- Encontrar todos los ciclos hamiltonianos de un grafo

Definición

- Sea $G = (V, A)$ un grafo conexo con n vértices
- Un **ciclo hamiltoniano** es un camino que visita una vez cada vértice y vuelve al vértice inicial
- Es decir, $(v_1 v_2 \dots v_{n+1})$ tal que:
 - $v_i \in V, i = 1, \dots, n + 1$
 - $(v_i, v_{i+1}) \in A, i = 1, \dots, n$
 - $v_1 = v_{n+1}$
 - $v_i \neq v_j, \forall i, j = 1, \dots, n : i \neq j$



Ejemplos



- Grafo a la izda
 - Hamiltoniano: **1-2-8-7-6-5-4-3-1**
- Grafo a la dcha
 - No contiene ningún hamiltoniano



Observaciones

- No se conoce algoritmo eficiente para resolver el problema
- Hay una relación entre este problema y el problema del viajante de comercio para el caso de un grafo con todas las distancias entre vértices iguales
 - Hay una heurística voraz muy eficiente pero subóptima para el problema del viajante de comercio
 - Hay una solución de programación dinámica



Análisis del problema y restricciones

- En el vector solución (x_1, \dots, x_n) , x_i representa el i -ésimo vértice visitado
- Si ya x_1, \dots, x_{k-1} ya tienen valores asignados, calculamos los posibles valores para x_k
 - $k = 1$: fijamos como x_1 un cualquier vértice, por ejemplo $x_1 = 1$
 - $1 < k < n$: x_k puede ser cualquier vértice distinto de x_1, \dots, x_{k-1} y conectado por un arco con x_{k-1}
 - $k = n$: x_n sólo puede ser el vértice que queda sin visitar y debe estar conectado por sendos arcos con x_1 y x_{n-1}



Estructura de datos

- Sólo se necesita saber si un arco existe o no

```
tipo grafo = vector[1..n,1..n] de bool
tipo sol = vector[1..n] de 1..n
```



Algoritmo (restricciones)

```

1  algoritmo siguienteValor(entsal x:sol; ent k:entero; ent g: grafo)
2  variables encontrado:bool; j:entero
3  principio
4  repetir
5      x[k]:= (x[k]+1) mod (n+1); {vértice prometedor}
6      si x[k] ≠ 0 entonces {no se ha asignado ningún vértice todavía}
7          encontrado:=falso;
8          si g[x[k-1],x[k]] entonces {el vértice está conectado con el previo}
9              j:=1; encontrado:=verdadero; {tiene que ser distinto de los previos}
10             mientrasQue (j ≤ k-1) and encontrado hacer
11                 si x[j]=x[k] entonces encontrado:=falso sino j:=j+1 fsi
12             fmientrasQue;
13             si encontrado entonces
14                 {caso último vértice: tiene que estar conectado con el primero}
15                 si (k=n) and not g[x[n],1] entonces encontrado:=falso fsi
16             fsi
17         fsi
18     fsi
19 hastaQue (x[k]=0) or encontrado
20 fin

```



Algoritmo de búsqueda

```

1  {Pre: Solución parcial hasta x[k-1]}
2  algoritmo hamiltoniano(ent k:entero; entsal x:sol; ent g: grafo)
3  principio
4      repetir
5          siguienteValor(x,k,g);
6          si x[k] ≠ 0 entonces
7              si k=n entonces escribir(x)
8              sino hamiltoniano(k+1,x,g) fsi
9          fsi
10     hastaQue x[k]=0
11     fin

```

Llamada inicial

```

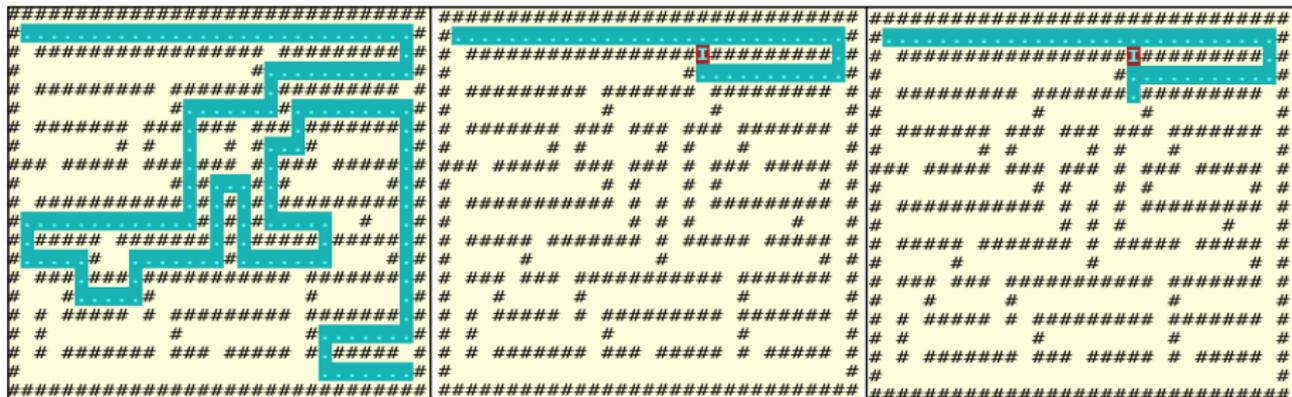
1  {g contiene la matriz de adyacencia del grafo y el vector solución está
2  inicializado a x=[1,0,...,0]}
3  hamiltoniano(2,x,g);

```




Diseño de un algoritmo de *backtracking*

- Se marcará en la misma matriz un camino solución (si existe)
- Si se llega a una casilla desde la que es imposible encontrar una solución, hay que volver atrás y buscar otro camino
- Hay que marcar las casillas por donde ya se ha pasado para evitar meterse varias veces en el mismo callejón sin salida, dar vueltas alrededor de columnas . . .



(a) Camino encontrado

(b) Caso retroceso

(c) Búsqueda otro camino



Estructura de datos y funciones auxiliares

```
1  {Estructura de datos}
2  tipos
3    casilla = (libre,pared,camino,imposible)
4    laberinto = vector[1..N,1..M] de casilla
5
6  {Funciones auxiliares}
7  funcion fueraLaberinto(fila,columna: entero) devuelve bool
8  principio
9    devuelve (fila < 2) or (fila > N-1) or (columna < 2) or (columna > M-1)
10 fin
11 funcion esLibre(lab: laberinto; fila,columna: entero) devuelve bool
12 principio
13   devuelve lab[fila][columna] = libre
14 fin
15 funcion esSalida(fila,columna: entero) devuelve bool
16 principio
17   devuelve (fila = N-1) and (columna = M-1)
18 fin
```



Solución de búsqueda con retroceso

```

1  algoritmo buscarCamino(entsal lab:laberinto; ent fila, columna: entero;
2      sal encontrado:booleano)
3  principio
4      si fueraLaberinto(fila,columna) or not esLibre(lab,fila,columna) entonces
5          encontrado:=falso
6      sino
7          lab[fila,columna]:=camino;
8          si esSalida(fila,columna) entonces {se ha encontrado una solución}
9              encontrado:=verdadero
10         sino
11             buscarCamino(fila,columna+1,lab,encontrado);
12             si not encontrado entonces buscarCamino(fila-1,columna,lab,encontrado)
13             si not encontrado entonces buscarCamino(fila,columna-1,lab,encontrado)
14             si not encontrado entonces buscarCamino(fila+1,columna,lab,encontrado)
15             si not encontrado entonces lab[fila,columna]:=imposible fsi
16         fsi
17     fsi
18 fin

```



Problema de la mochila 0-1

- Hay n objetos y una mochila
- El objeto i tiene peso p_i y su inclusión en la mochila produce un beneficio b_i
- **Objetivo:** llenar la mochila de capacidad C , de manera que se maximice el beneficio

$$\max \sum_{i=1}^n b_i x_i$$

$$t.q. \sum_{i=1}^n p_i x_i \leq C$$

$$x_i \in \{0, 1\}, b_i, p_i > 0, 1 \leq i \leq n$$

Soluciones a problemas parecidos

- Objetos fraccionables ($0 \leq x_i \leq 1, 1 \leq i \leq n$): **solución voraz**
- Mochila 0-1 con pesos, beneficios y capacidad de la mochila números naturales (**solución de programación dinámica**)
- En el caso general, ninguna de las dos soluciones funciona

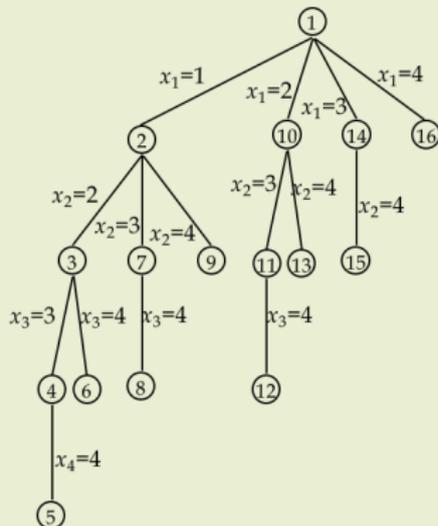


Espacio de soluciones

- 2^n modos de asignar los valores 0 ó 1 a las x_i ;
- Dos formas de representar la solución

Tuplas de tamaño variable:

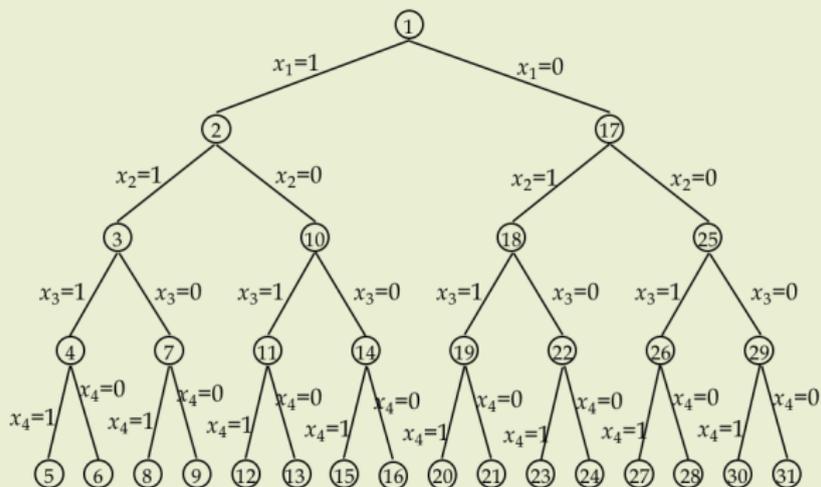
- $x_i =$ objeto introducido en el i -ésimo lugar





Espacio de soluciones

Tuplas de tamaño fijo



- $x_i = 1$ si el objeto i -ésimo se introduce
- $x_i = 0$ si el objeto i -ésimo no se introduce

Elegimos ésta última representación



Elección de funciones acotadoras

Intentar **podar** ramas que no puedan producir soluciones mejores que la disponible actualmente

- Se llama *poda basada en el coste de la mejor solución en curso*
- Calcular una cota superior del valor de la mejor solución posible al expandir un nodo y sus descendientes
- Si la cota es menor o igual que el valor de la mejor solución disponible hasta el momento, no expandir ese nodo
- ¿Cómo calcular esa cota superior?
 - Suponer que en el nodo actual ya se han determinado $x_i, 1 \leq i \leq k$
 - Relajar el requisito de integridad:

$$x_i \in \{0, 1\}, k + 1 \leq i \leq n \implies 0 \leq x_i \leq 1, k + 1 \leq i \leq n$$
 - Aplicar el algoritmo voraz



Algoritmo para el cálculo de la cota

```

1  const N=... {número de objetos}
2  tipo vectReal=vector[1..N] de real

```

```

1  {Pre:  $\forall i = 1, \dots, n : peso[i] > 0 \wedge \forall i = 1, \dots, n-1 : \frac{benef[i]}{peso[i]} \geq \frac{benef[i+1]}{peso[i+1]}$ }
2  funcion cota(benef,peso:vectReal; cap,ben:real; obj:entero) devuelve real
3  {cap=capacidad aún libre de la mochila; ben=beneficio actual;
4   obj=índice del primer objeto a considerar}
5  principio
6   si obj>N or cap=0.0 entonces devuelve ben
7   sino
8     si peso[obj]>cap entonces devuelve ben+cap/peso[obj]*benef[obj]
9     sino
10      devuelve cota(benef,peso,cap-peso[obj],ben+benef[obj],obj+1)
11     fsi
12   fsi
13  fin

```



Algoritmo de búsqueda con retroceso

```

1  tipo solución=vector[1..N] de 0..1
2  {Variables globales: benef,peso:vectReal; cap:real}
3
4  algoritmo búsqueda(ent solAct:solución; ent benAct,pesAct:real;
5                      ent obj:entero; entsal sol:solución, entsal ben:real)
6  variable decisión: 0..1
7  principio
8    para decisión:=1 descendiendo hasta 0 hacer
9      solAct[obj]:=decisión;
10     benAct:=benAct+decisión*benef[obj];
11     pesAct:=pesAct+decisión*peso[obj];
12     si pesAct<=cap and ben<cota(benef,peso,cap-pesAct,benAct,obj+1) entonces
13       si obj=N entonces
14         si benAct>ben entonces sol:=solAct; ben:=benAct fsi
15       sino búsqueda(solAct,benAct,pesAct,obj+1,sol,ben)
16     fsi
17   fsi
18   fpara
19   fin

```

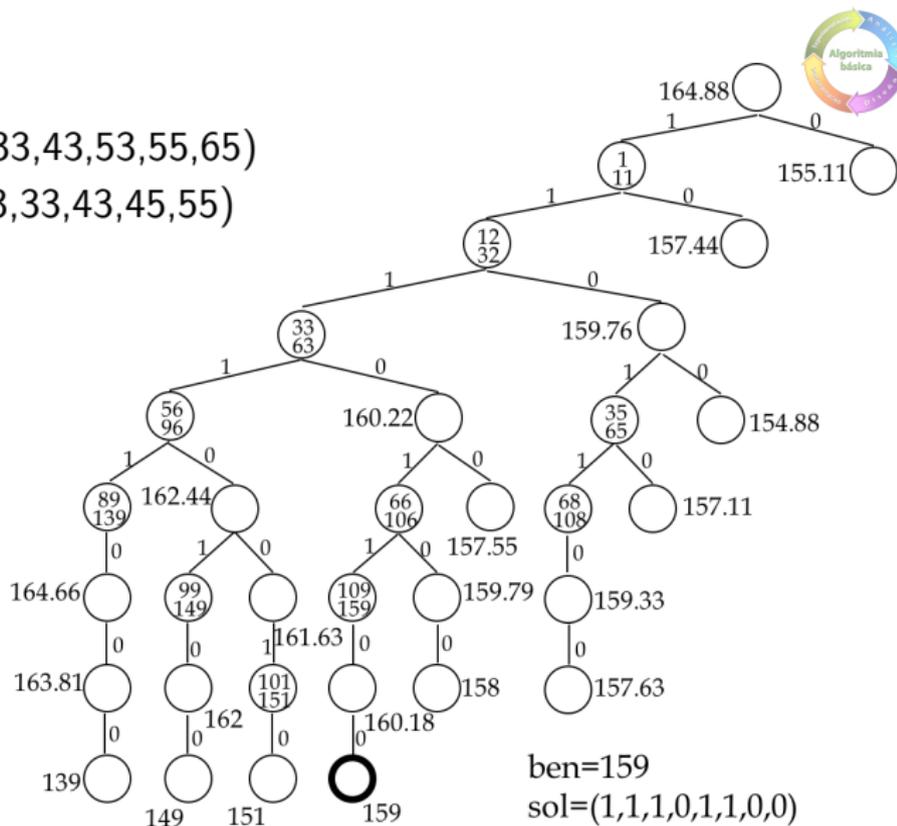


Algoritmo solución

```
1  algoritmo mochila0-1(ent benef, peso:vectReal; ent cap:real;
2      sal sol:solución; sal ben: real)
3  variables solAct:solución
4  principio
5      {inicialización vectores solAct=[0,..0] y sol[0,..,0]}
6      ben:=0.0;
7      búsqueda(solAct,0.0,0.0,1,sol,ben)
8  fin
```

Ejemplo

- $\text{benef} = (11, 21, 31, 33, 43, 53, 55, 65)$
- $\text{peso} = (1, 11, 21, 23, 33, 43, 45, 55)$
- $\text{cap} = 110$
- $N = 8$





Problema de la mochila 0-1

- Tamaño del espacio de estados: $2^9 - 1 (= 511)$ nodos
- Aplicando el algoritmo de backtracking, sólo se generan 33
- Se podía haber reducido a 26 simplemente sustituyendo la condición

```
ben < cota(...)
```

en el algoritmo de búsqueda por:

```
ben < parteEntera(cota(...))
```



Observaciones

- Hasta ahora hemos visto algoritmos de búsqueda con retroceso basados en árbol de espacio de estados **estático**
 - Es decir, la estructura del árbol es independiente de la instancia del problema
- Se pueden diseñar algoritmos basados en árboles de espacio de estados **dinámicos**



Solución del problema basada en un árbol dinámico

- Resolver el problema sin la restricción de integridad, con algoritmo voraz

- Si la solución es entera, también es óptima para el problema 0-1
- Si no es entera, existe exactamente un x_i tal que $0 < x_i < 1$

$$\begin{aligned} \max \sum_{i=1}^n b_i x_i & \quad (1) \\ \text{s.t. } \sum_{i=1}^n p_i x_i & \leq C \\ 0 \leq x_i \leq 1, b_i, p_i & > 0, 1 \leq i \leq n \end{aligned}$$

- Se parte el espacio de soluciones en dos: en uno (subárbol izquierdo) $x_i = 0$ y en otro (subárbol derecho) $x_i = 1$
- En general, en cada nodo, se usa el algoritmo voraz para resolver el problema (1) con las restricciones añadidas a las asignaciones ya realizadas a lo largo del camino desde la raíz al nodo