Búsqueda con retroceso (*backtracking*)

Simona Bernardi

Universidad de Zaragoza

Presentación reducida y adaptada de la original de J. Campos

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
  $\Rightarrow$ Técnica **voraz**
- Se cumple el principio de optimalidad de Bellman
  $\Rightarrow$ 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 $(x_1, \ldots, x_n)$ 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 safisfacen 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 $(x_1, \ldots, x_i)$ puede conducir a una tupla completa satisfactoria

Planteamiento del problema II

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

Diferencia entre fuerza bruta y búsqueda con retroceso

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

- 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 $C_1 \times \cdots \times C_n$)
  - **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

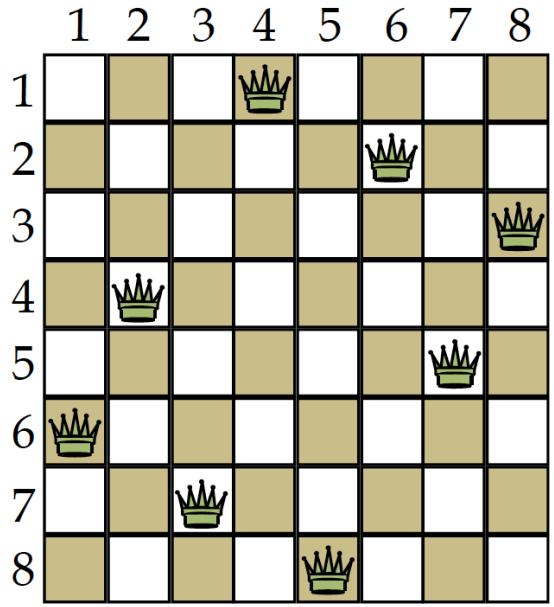
Representación de una solución

Toda solución del problema puede representarse con una 8-tupla $(x_1, \ldots, x_8)$ 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

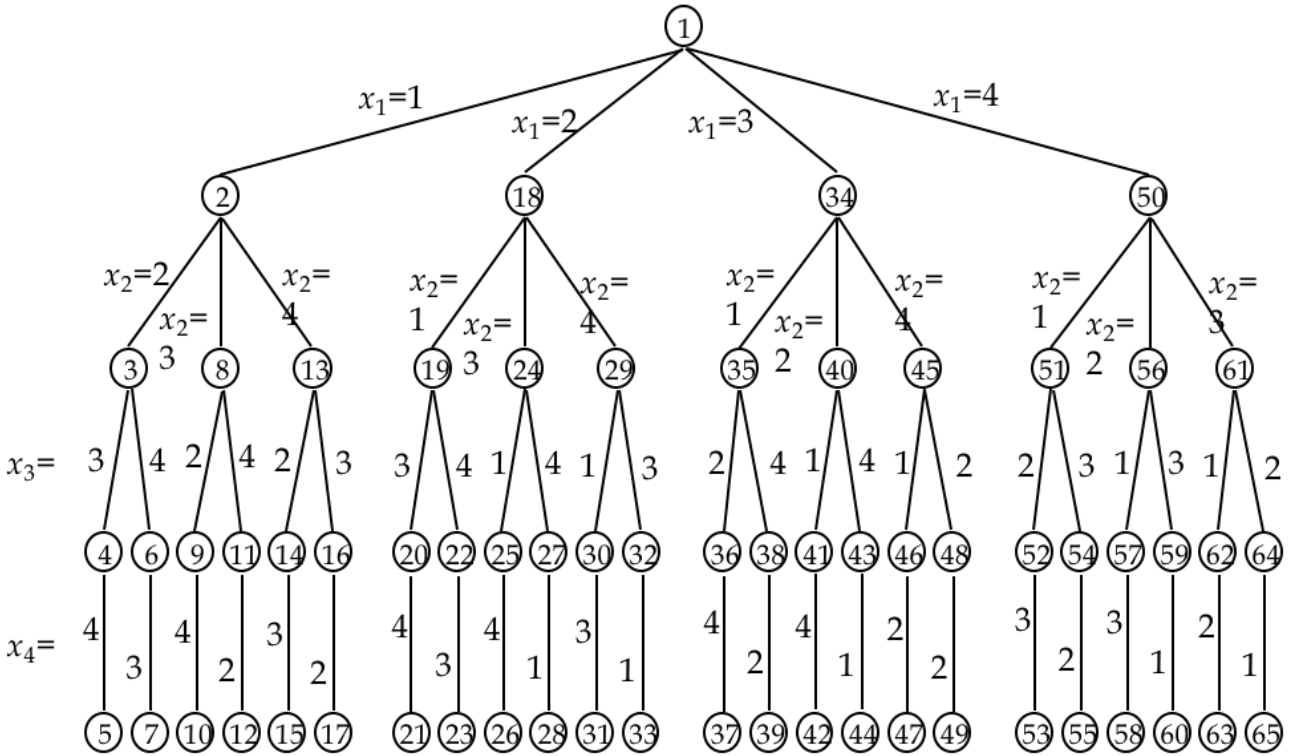
Una solución: $(4, 6, 8, 2, 7, 1, 3, 5)$



Representación del espacio de soluciones

- Para facilitar la búsqueda, se adopta una **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

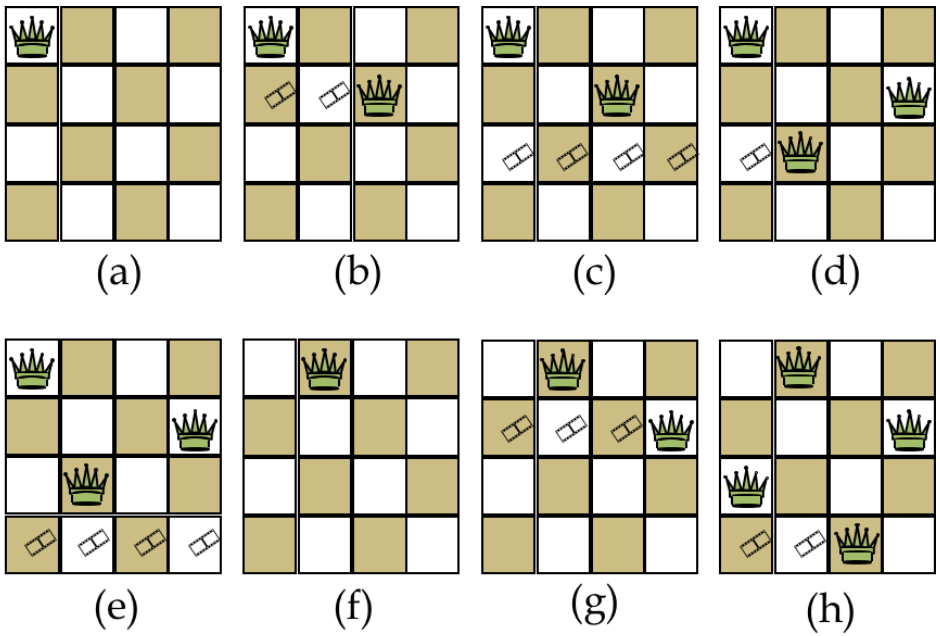
- Sean
  - $(x_1, \ldots, x_i)$ un camino de la raíz hasta un nodo del árbol del espacio de estados
  - $G(x_1, \ldots, x_i)$ el conjunto de los valores posibles de x_{i+1} tales que $(x_1, \ldots, x_{i+1})$ es un camino hasta un nodo del árbol
- Suponemos que existe algún predicado acotador A tal que
  - $A(x_1, \ldots, x_{i+1}) = \mathit{falso} \Rightarrow$ el camino $(x_1, \ldots, x_{i+1})$ no puede extenderse para alcanzar una solución
- Por tanto, los candidatos x_{i+1} son los valores de $G(x_1, \ldots, x_i)$ tales que $A(x_1, \ldots, x_{i+1}) = \mathit{verdadero}$
- Supongamos finalmente, que existe un predicado R que determina si un camino $(x_1, \ldots, x_{i+1})$ es una solución.

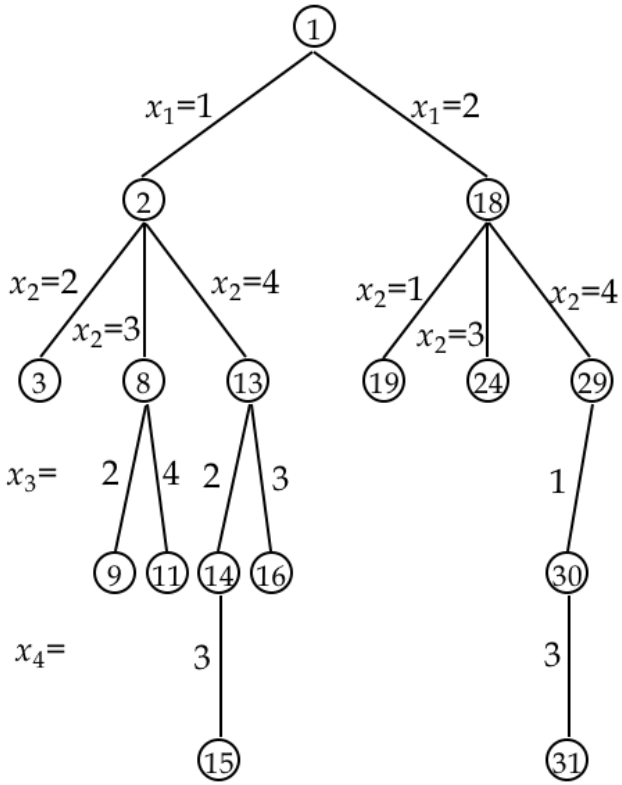
Esquema algorítmico


```
algoritmo buscarSol(ent k:entero; entsal sol:vector[1..n] de elmto)
{Pre: sol[1..k-1] es prometedora}
variable nodo:elmto
principio
  para todo nodo en G(sol,1,k-1) hacer {G(sol,1,n) = \emptyset}
    sol[k]:=nodo;
    si A(sol,1,k) entonces
      si R(sol,1,k) entonces
        guardar(sol,1,k) {se guardan todas las soluciones}
      fsi;
      buscarSol(k+1,sol)
    fsi
  fpara
fin
```

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

Problema de las cuatro reinas





16 nodos frente a los 65 del árbol completo

Observaciones

- El árbol no se construye explícitamente sino implicítamente mediante las llamadas recursivas del algoritmo de búsqueda
- El algoritmo no hace llamada recursivas cuando
  - $k = n + 1$ ($\texttt{G(sol,1,n)} = \emptyset$) 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 (realizando la recursión sólo si no se ha encontrado un nodo 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 $|C_1| < |C_2| < \cdots < |C_n|$, 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:

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

Algoritmo de estimación


```
funcion estimación devuelve entero
variables k,m,r,card: entero; nodo: elmto; sol: vector[1..n] de elmto
principio
  k:=1; m:=1; r:=1;
  repetir
    card:=0;
    para todo nodo en G(sol,1,k-1) hacer
      sol[k]:= nodo;
      si A(sol,1,k) entonces card:=card+1 fsi
    fpara
    si card <> 0 entonces
      r=r*card;
      m:=m+r;
      sol[k] := elegirAleatorio(G(sol,1,k-1));
      k:=k+1;
    fsi
  hastaQue R(sol,1,k) or (card=0);
  devuelve m
fin
```

Problema general de las n reinas

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

- Cada solución se representa por una n-pla $(x_1, \ldots, x_n)$, 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 $(1, 2, \ldots, n)$
- 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 $\nearrow$ si tienen el mismo valor de `fila+columna`
  - en la misma diagonal $\nwarrow$ 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


```
funcion buenSitio(k: entero; x: vector[1..n] de entero) devuelve bool
{Devuelve verdadero si y sólo si se puede colocar la k-ésima reina en x[k],
 habiendo sido colocadas ya las k-1 reinas anteriores}
variables i: entero; esBueno: bool
principio
  i:=1; esBueno:=verdadero;
  mientrasQue (i<k) and esBueno hacer
    si x[i]=x[k] or abs(x[i]-x[k]) = abs(i-k) entonces
       esBueno = falso
    sino
       i:=i+1
    fsi
  fmientrasQue
  devuelve esBueno
fin
```

- ¿Coste de la función? $O(k-1)$

Algoritmo de búsqueda

Versión recursiva

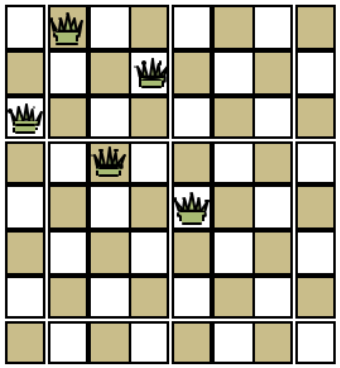

```
algoritmo colocarReinas(ent k: entero; entsal sol: vector[1..n] de entero)
{sol[1..k-1] están bien colocadas}
variables i: entero
principio
  para i:=1 hasta n hacer
    sol[k]:=i;
    si buenSitio(k,sol) entonces
      si k=n entonces escribir(sol)
        sino colocarReinas(k+1,sol)
      fsi
    fsi
  fpara
fin
```

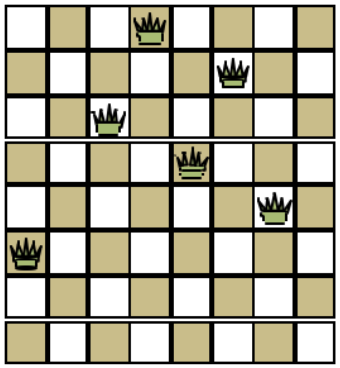
- La llamada inicial: `colocarReinas(1,solución);`

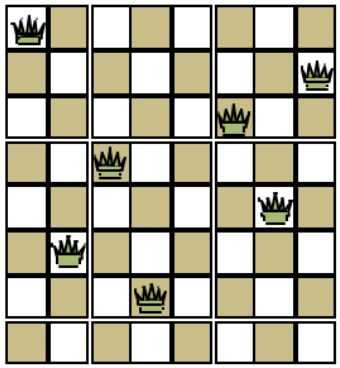
Estimación del coste

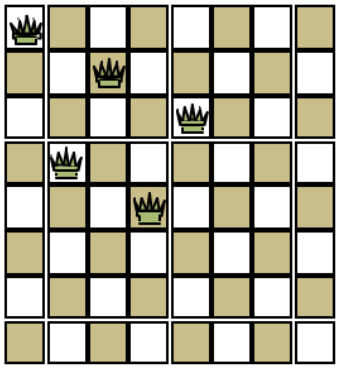
Comparación con el enfoque fuerza bruta

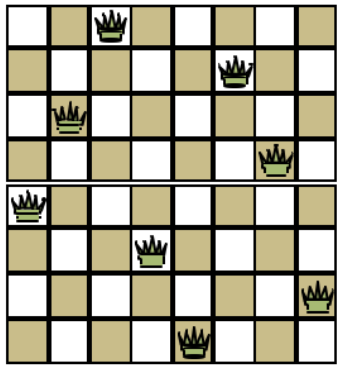
Caso 8 reinas, 5 evaluaciones de la función `estimación`


(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 $m = 1 + m_1 + m_1 m_2 + m_1 m_2 m_3 + \ldots$

Estimación del coste II

Comparación con el enfoque fuerza bruta

- $\overline{m} = \frac{\sum_{i=1}^{5} m_i}{5} = 1645$ (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 $x_1 = 2, 3, \ldots, \lceil n/2 \rceil$

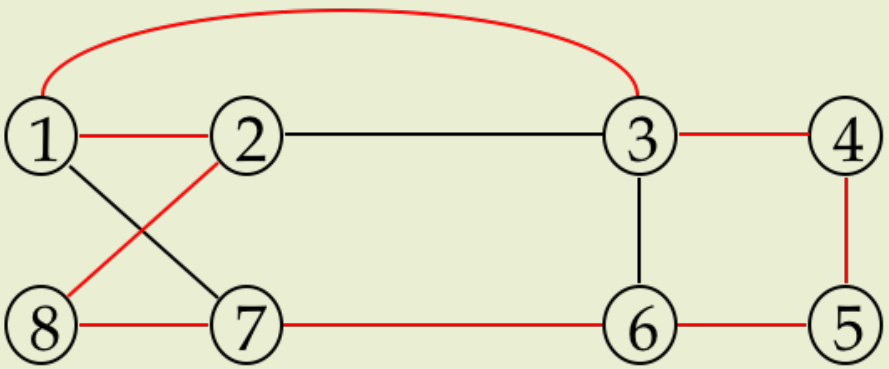
Ciclos hamiltonianos

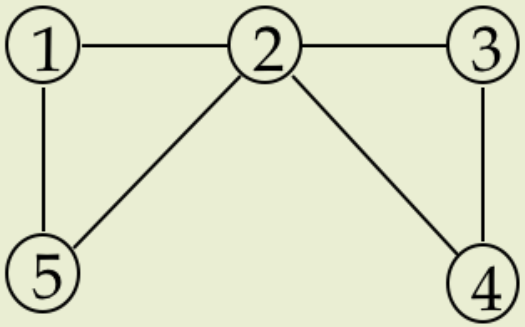
- Encontrar todos los ciclos hamiltonianos de un grafo

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 $(x_1, \ldots, x_n)$, x_i representa el i-ésimo vértice visitado
- Si ya $x_1, \ldots, x_{k-1}$ 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 $x_1, \ldots, x_{k-1}$ 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

El grafo

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

La solución

```
tipo sol = vector[1..n] de 1..n
```

Algoritmo (restricciones)


```
algoritmo siguienteValor(entsal x:sol; ent k:entero; ent g: grafo)
variables encontrado:bool; j:entero
principio
  repetir
    x[k]:=(x[k]+1) mod (n+1); {vértice prometedor}
    si x[k]<>0 entonces {no se ha asignado ningún vértice todavía}
      encontrado:=falso;
      si g[x[k-1],x[k]] entonces {el vértice está conectado con el previo}
        j:=1; encontrado:=verdadero; {tiene que ser distinto de los previos}
        mientrasQue (j<=k-1) and encontrado hacer
          si x[j]=x[k] entonces encontrado:=falso sino j:=j+1 fsi
        fmientrasQue;
        si encontrado entonces
        {caso último vértice: tiene que estar conectado con el primero}
          si (k=n) and not g[x[n],1] entonces encontrado:=falso fsi
        fsi
      fsi
    fsi
  hastaQue (x[k]=0) or encontrado
fin
```

Algoritmo de búsqueda


```
algoritmo hamiltoniano(ent k:entero; entsal x:sol; ent g: grafo)
{En x se tiene la parte de la solución ya calculada, es decir hasta x[k-1]}
principio
  repetir
    siguienteValor(x,k,g);
    si x[k]<>0 entonces
      si k=n entonces escribir(x)
      sino hamiltoniano(k+1,x,g) fsi
    fsi
  hastaQue x[k]=0
fin
```

Llamada inicial

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

Atravesar un laberinto

- Hay una entrada y una salida en el laberinto y hay que intentar atravesarlo
- El mapa del laberinto: una matriz $N \times M$ de casillas marcadas como *libre* u *ocupadas* por una pared
  - Entrada: casilla $(2, 2)$
  - Salida: casilla $(N-1, M-1)$
- Es posible pasar de una casilla a otra moviéndose sólamente en vertical u horizontal

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


```
{Estructura de datos}
tipos
  casilla = (libre,pared,camino,imposible)
  laberinto = vector[1..N,1..M] de casilla

{Funciones auxiliares}
funcion fueraLaberinto(fila,columna: entero) devuelve bool
principio
  devuelve (fila < 2) or (fila > N-1) or (columna < 2) or (columna > M-1)
fin
funcion esLibre(lab: laberinto; fila,columna: entero) devuelve bool
principio
  devuelve lab[fila][columna] = libre
fin
funcion esSalida(fila,columna: entero) devuelve bool
principio
  devuelve (fila = N-1) and (columna = M-1)
fin
```

Solución de búsqueda con retroceso


```
algoritmo buscarCamino(entsal lab:laberinto; ent fila, columna: entero;
                       sal encontrado:booleano)
principio
  si fueraLaberinto(fila,columna) or not esLibre(lab,fila,columna) entonces
    encontrado:=falso
  sino
    lab[fila,columna]:=camino;
    si esSalida(fila,columna) entonces {se ha encontrado una solución}
        encontrado:=verdadero
    sino
      buscarCamino(fila,columna+1,lab,encontrado);
      si not encontrado entonces buscarCamino(fila-1,columna,lab,encontrado) fsi;
      si not encontrado entonces buscarCamino(fila,columna-1,lab,encontrado) fsi;
      si not encontrado entonces buscarCamino(fila+1,columna,lab,encontrado) fsi;
      si not encontrado entonces lab[fila,columna]:=imposible fsi
    fsi
  fsi
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

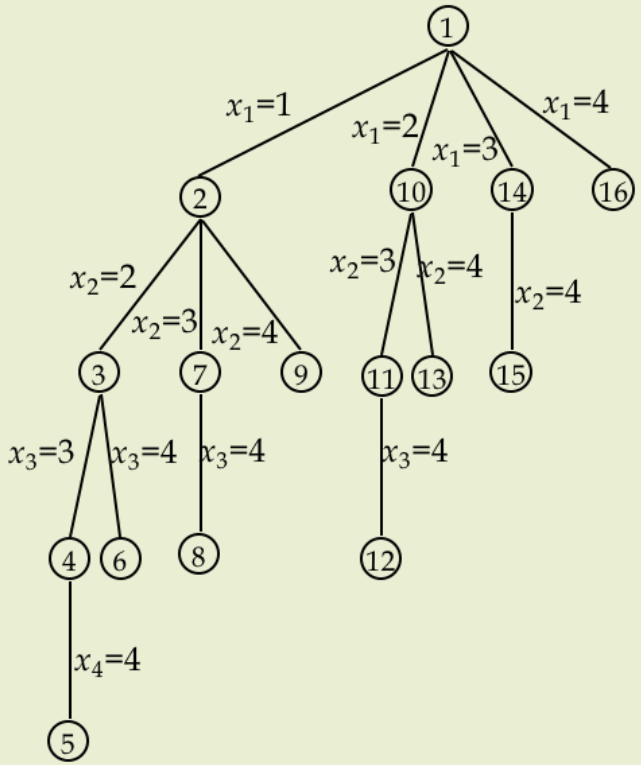
- Objetos fraccionables ($0 \leq x_i \leq 1, 1 \leq i \leq n$): **solución voraz**
- Mochila 0-1 con pesos, beneficios y capacidad de la mochila números naturales (**solución de programación dinámica**)
- En el caso general, ninguna de las dos soluciones funciona

Espacio de soluciones

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

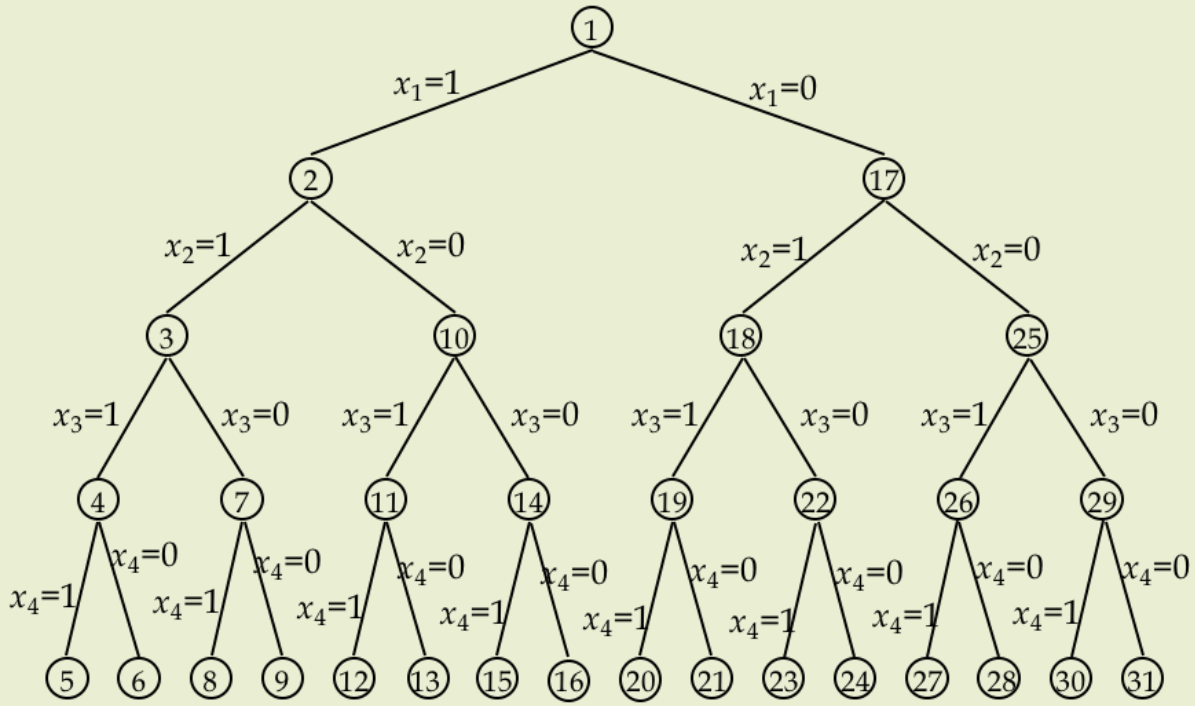
Tuplas de tamaño variable:

- 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:

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

Algoritmo para el cálculo de la cota


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

```
{Pre: forall i = 1..n: peso[i]>0 and
      forall i=1..n-1: benef[i]/peso[i] >= benef[i+1]/peso[i+1]}
funcion cota(benef,peso:vectReal; cap,ben:real; obj:entero) devuelve real
{cap=capacidad aún libre de la mochila; ben=beneficio actual;
 obj=índice del primer objeto a considerar}
principio
 si obj>N or cap=0.0 entonces devuelve ben
 sino
   si peso[obj]>cap entonces devuelve ben+cap/peso[obj]*benef[obj]
   sino
      devuelve cota(benef,peso,cap-peso[obj],ben+benef[obj],obj+1)
   fsi
 fsi
fin
```

Algoritmo de búsqueda con retroceso

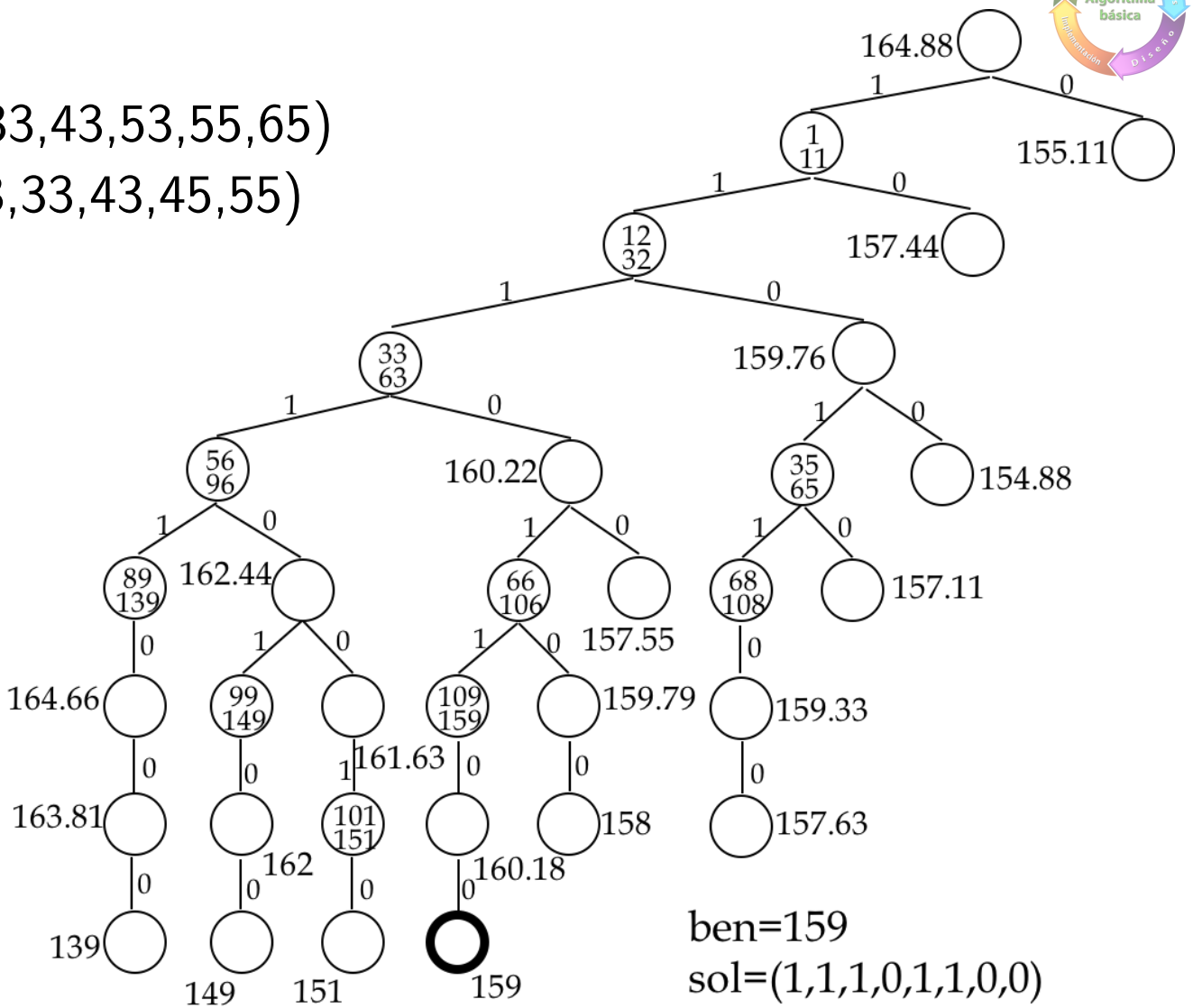

```
tipo solución=vector[1..N] de 0..1
{Variables globales: benef,peso:vectReal; cap:real}

algoritmo búsqueda(ent solAct:solución; ent benAct,pesAct:real;
                   ent obj:entero; entsal sol:solución, entsal ben:real)
variable decisión: 0..1
principio
  para decisión:=1 descendiendo hasta 0 hacer
    solAct[obj]:=decisión;
    benAct:=benAct+decisión*benef[obj];
    pesAct:=pesAct+decisión*peso[obj];
    si pesAct<=cap and ben<cota(benef,peso,cap-pesAct,benAct,obj+1) entonces
      si obj=N entonces
        si benAct>ben entonces sol:=solAct; ben:=benAct fsi
      sino búqueda(solAct,benAct,pesAct,obj+1,sol,ben)
      fsi
   fsi
  fpara
fin
```

Algoritmo solución


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

Ejemplo

- benef=(11,21,31,33,43,53,55,65)
- peso =(1,11,21,23,33,43,45,55)
- 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 $0 < x_i < 1$

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

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

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

- 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