Programación II

Tema 9. Vuelta atrás

9.1 Introducción a la Vuelta Atrás

Los algoritmos de "Vuelta Atrás" (*Backtracking*) realizan una *exploración sistemática de las posibles soluciones* del problema

Vuelta Atrás suele utilizarse para resolver problemas complejos

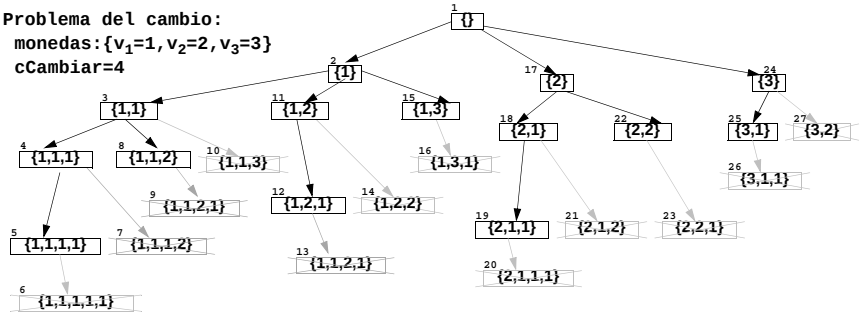
- **en los que la única forma de encontrar la solución es analizando todas las posibilidades**
  - **ritmo de crecimiento exponencial**
- **constituye una forma sistemática de recorrer todo el espacio de soluciones**

En general, podremos utilizar Vuelta Atrás para problemas:

- **con solución representada por una n-tupla $\{x_1, x_2, \ldots, x_n\}$**
- **en los que cada una de las componentes x_i de ese vector es elegida en cada etapa de entre un conjunto finito de valores $(y_1, y_2, \ldots, y_v)$**

El algoritmo va obteniendo soluciones parciales:

- **normalmente el espacio de soluciones parciales es un árbol**
  - ***árbol de búsqueda* de las soluciones**
- **el algoritmo realiza un recorrido en profundidad del árbol**
- **cuando desde un nodo no se puede seguir buscando la solución se produce una *vuelta atrás***



Algunas definiciones:

- ***Nodo vivo*: nodo del que todavía no se han generado todos sus hijos**
- ***Nodo prometedor*: nodo que no es solución pero desde el que todavía podría ser posible llegar a la solución**

Dependiendo de si buscamos una solución cualquiera o la óptima

- **el algoritmo se detiene una vez encontrada la primera solución**
- **o continúa buscando el resto de soluciones**

Estos algoritmos no crean ni gestionan el árbol explícitamente

- **se crea implícitamente con las llamadas *recursivas* al algoritmo**

9.2 Eficiencia de los algoritmos VA

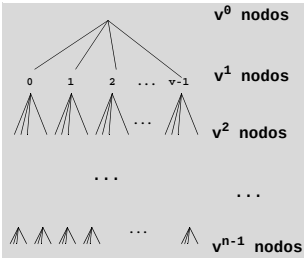
La eficiencia de este tipo de algoritmos depende del número de nodos que sea necesario explorar para cada caso

- **es imposible de calcular a priori de forma exacta**

Pero es posible calcular una *cota superior*

- **sea n la longitud máxima de la solución y `0..v-1` el rango de valores para cada decisión tenemos que:**

$$nodos = \sum_{i=0}^{n-1} v^i \cong v^n$$



- **luego su eficiencia temporal será `O(v^n)`**
  - **resultado muy pesimista en la mayoría de los casos**

Tienen unos requisitos de memoria `O(n)`

- **máxima "profundidad" de las llamadas recursivas**

9.3 Problema de la mochila con Vuelta Atrás

Vamos a resolver otra versión del problema de la mochila:

- **se dispone un número ilimitado de objetos de cada tipo**
- **los objetos no se pueden fraccionar ("mochila entera" o "mochila {0,1}")**
- **Problema NP-completo**

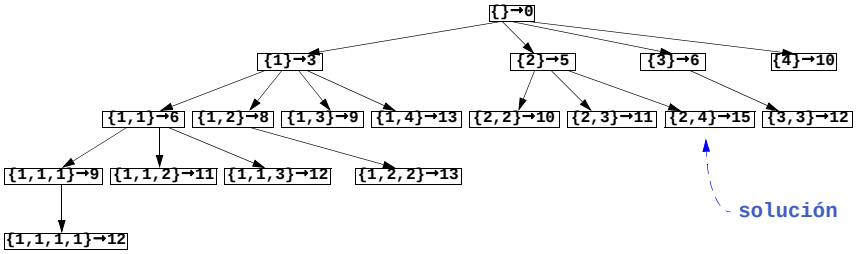
El problema verifica las condiciones necesarias para que pueda ser abordable utilizando Vuelta Atrás:

- **la solución se puede representar con una N-tupla**
  - **conjunto de objetos metidos en la mochila**
- **en cada etapa de formación de la solución de elige un nuevo objeto de entre todos los objetos existentes (un conjunto finito)**

Árbol de búsqueda del problema de la mochila

Veamos el árbol de búsqueda para un caso particular

- **tipos de objetos: $\{p_0=2, v_0=3\}$, $\{p_1=3, v_1=5\}$, $\{p_2=4, v_2=6\}$ y $\{p_3=5, v_3=10\}$, $N=4$**
- **la mochila soporta un peso máximo $P_m=8$**



- **La solución es la pareja $\{2,4\}$**

Pseudocódigo (sólo da valor máximo)

```
// retorna el máximo valor que podemos guardar en
// una mochila de peso máximo pm
método mochila(entero pm) retorna entero
  v:=0   // valor almacenado
  // probamos si cabe cada clase de objeto
  desde n:=0 hasta N-1 hacer
    si p_n<=pm entonces
      // el objeto cabe en la mochila
      v:=max(v, v_n+mochila(pm-p_n))
    fsi
  fhacer
  retorna v
fmétodo
```

Se puede optimizar este algoritmo

- **para no repetir en una llamada recursiva casos ya explorados con anterioridad**

Pseudocódigo optimizado

```
// retorna el máximo valor que podemos guardar en
// una mochila de peso máximo pm utilizando sólo
// objetos de los tipos n0..N-1
método mochila(entero n0, entero pm) retorna entero
  v:=0   // valor almacenado
  // probamos si cabe cada clase de objeto
  desde n:=n0 hasta N-1 hacer
    si p_n<=pm entonces
      // el objeto cabe en la mochila
      v:=max(v, v_n+mochila(n,pm-p_n))
    fsi
  fhacer
  retorna v
fmétodo
```

La primera llamada a este método será:

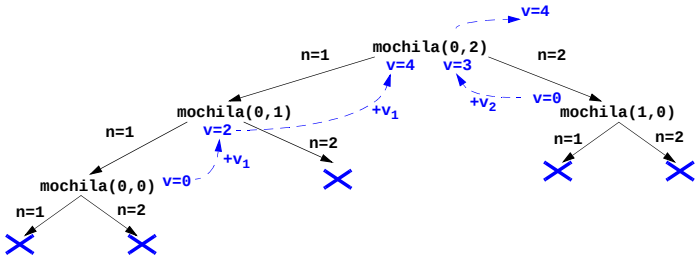
```
mochila(0,P)  // donde P es el peso máximo
```


Árbol de búsqueda

La llamadas recursivas incluidas en el pseudocódigo anterior generan el árbol de búsqueda formado por las soluciones parciales

Ejemplo de árbol de búsqueda para un caso sencillo:

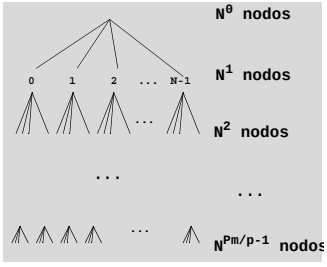
- **tipos de objetos: $\{p_0=1, v_0=2\}$ y $\{p_1=2, v_1=3\}$, $N=2$**
- **la mochila soporta un peso máximo $P_m=2$**



Eficiencia (resultado pesimista)

Para un problema con N tipos de objetos, en que el peso máximo es P_m y el peso del objeto más ligero es p

- **el número máximo de objetos que caben en la mochila es P_m/p**



Su eficiencia es proporcional al máximo número de nodos que puede ser necesario explorar

- **Eficiencia: $O(N^{Pm/p})$**

Pseudocódigo (retorna objetos incluidos)

```
método mochila(entero n0, entero pm)
                retorna listaIncluidos
  listaIncluidos:=∅   // lista vacía
  // probamos si cabe cada clase de objeto
  desde n:=n0 hasta N-1 hacer
    si p_n<=pm entonces
      lista := mochila(n, pm-p_n)
      si v_n+lista.v > listaIncluidos.v entonces
        // 'lista' más el nuevo objeto tiene mejor
        // valor que 'listaIncluidos'
        listaIncluidos := lista
        añade objeto n a listaIncluidos
      fsi
    fsi
  fhacer
  retorna listaIncluidos
fmétodo
```


Implementación (retorna objetos incluidos)

```
// cada uno de los tipos de objetos disponibles para
// incluir en la mochila
public static class ObjetoMochila {
  int v;
  int p;
  String nombre;
  public ObjetoMochila(int v, int p, String nombre){
    this.v=v; this.p=p; this.nombre=nombre;
  }
}

// solución del algoritmo: lista de objetos
// incluidos en la mochila y suma de su valor
public static class SoluciónMochila {
  int v = 0;
  LinkedList<ObjetoMochila> lista =
            new LinkedList<ObjetoMochila>();
}
```



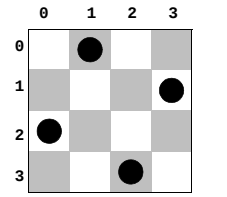
```
// método que retorna el contenido óptimo de la mochila
// (conjunto de objetos con el que se consigue el mayor valor)
// Recibe como argumentos el array con los objetos y
// el peso máximo soportado por la mochila
// Llama a mochilaRec
public static SoluciónMochila
                mochila(ObjetoMochila[] objs, int pm){
  return mochilaRec(0,objs,pm);
}


// retorna el conjunto de objetos con mayor valor que podemos
// guardar en una mochila de peso máximo pm utilizando sólo
// objetos de los tipos n..obj.length-1
private static SoluciónMochila
    mochilaRec(int n0, ObjetoMochila[] objs, int pm){
  ...  // código en la transparencia siguiente
```

Implementación (retorna objetos incluidos) (cont.)

```
private static SoluciónMochila
    mochilaRec(int n0, ObjetoMochila[] objs, int pm) {
  SoluciónMochila sm = new SoluciónMochila();
  // probamos si cabe cada clase de objeto
  for(int n=n0; n<objs.length; n++) {
    if (objs[n].p<=pm) {
      SoluciónMochila sm1=
               mochilaRec(n, objs, pm-objs[n].p);
      if (objs[n].v+sm1.v > sm.v) {
        // sm1+objs[n] tiene mejor valor que sm
        // elegimos sm1+objs[n] como la sol. actual
        sm=sm1;
        sm.v = objs[n].v+sm.v;
        sm.lista.add(objs[n]);
      }
    }
  } // fin for
  return sm;
}
```


9.4 Las N Reinas

Colocar N reinas en un tablero de ajedrez de N×N de forma que no se puedan "comer" entre sí



Una de las dos posibles soluciones al problema de las 4 reinas

En una solución nunca podrá haber dos reinas en la misma fila

- **este hecho permite expresar la solución como una N-tupla $\{c_0, c_1, \ldots, c_{N-1}\}$**
- **siendo c_f la columna donde se sitúa la reina de la f-ésima fila**
  - **la solución del dibujo se representa por la 4-tupla: {1,3,0,2}**

El problema verifica las condiciones necesarias para que pueda ser abordable utilizando Vuelta Atrás:

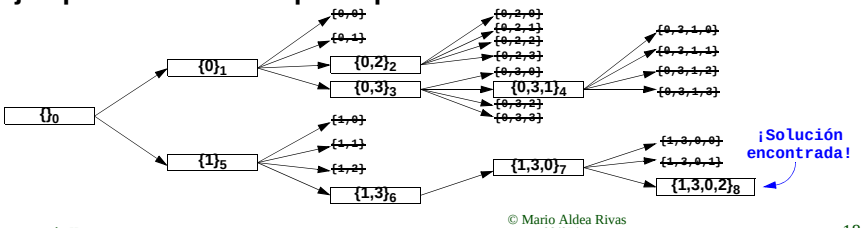
- **la solución se puede representar con una N-tupla**
- **los componentes c_i se eligen uno a uno de entre un conjunto finito de valores (las columnas de la tabla)**

Árbol de búsqueda de las soluciones

Deberemos recorrer las posibles soluciones parciales

- **cada posible solución parcial es una n-tupla (n≤N)**
- **una solución parcial es *n-prometedora* si define una colocación de n reinas en las n primeras filas sin que se "coman"**
- **la expansión de una rama se corta al encontrar una solución NO n-prometedora**
- **una solución al problema es cualquier solución N-prometedora**

Ejemplo: árbol de búsqueda para un tablero de 4×4:



Pseudocódigo N Reinas

```
// método que encuentra una solución para el
// problema de las N reinas
// Retorna un array con la columna en que se sitúa
// la reina de cada fila
método nReinas(entero N) retorna entero[0..N-1]
  solución:=nuevo entero[0..N-1]
  si nReinasRec(0,solución) entonces
    retorna solución
  else
    retorna nulo
  fsi
fmétodo

// Coloca recursivamente las reinas de las filas f..N-1
// Retorna verdadero si encuentra la solución
método nReinasRec(entero f, entero[0..N-1] sol)
                                    retorna Booleano
  ...    // código en la transparencia siguiente
```



```
método nReinasRec(entero f, entero[0..N-1] sol)
                                    retorna Booleano
  // para cada columna
  desde c:=0 hasta N-1 hacer
    sol[f]:=c // reina de la fila f en la columna c
    si nPrometedora(sol[0..f]) entonces
      si f==N-1 entonces // ¿es f la última fila?
        // la solución incluye a todas las reinas
        retorna verdadero // solución encontrada
      fsi
      // llamada recursiva para la siguiente fila
      solEncontrada:=nReinasRec(f+1, sol)
      si solEncontrada entonces
        retorna verdadero
      fsi
    fsi
  fhacer
  retorna falso
fmétodo
```


Necesitamos un método para determinar si una solución parcial (`sParcial`) es n-prometedora (n≤N):

```
método nPrometedora(entero[0..n] sParcial) retorna Booleano
  {sParcial[0..n-1] es (n-1)prometedora}
  desde i:=0 hasta n-1 hacer
    // misma columna o misma diagonal?
    si sParcial[i] == sParcial[n] o
       |sParcial[i]-sParcial[n]| == |i-n| entonces
      retorna falso // no n-prometedora
    fsi
  fhacer
  // sParcial[0..n] es n-prometedora
  retorna true
fmétodo
```

Dos reinas están en la misma diagonal cuando:

$|columna_a - columna_b| == |fila_a - fila_b|$

Eficiencia

Eficiencia temporal:

- **longitud de la solución: `N` (número de reinas)**
- **alternativas en cada etapa: `N` (número de columnas)**
- ***eficiencia*: `O(N^N)`**

Eficiencia espacial:

- **un array de tamaño `N`: `O(N)`**
- **variables auxiliares: `O(1)`**
- **como máximo `N` llamadas recursivas anidadas: `O(N)`**
- ***eficiencia*: `O(N)`**

9.5 Laberinto

```
{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
{-1,00,-1,-1,-1,-1,-2,-1,00,00,-1},
{-1,00,-1,00,-1,-1,00,-1,-1,00,-1},
{-1,00,-1,00,-1,-1,00,00,00,00,-1},
{-1,00,00,00,00,00,00,-1,00,-1,-1},
{-1,-1,00,-1,00,-1,-1,-1,00,-1,-1},
{-1,-1,00,-1,00,-1,-1,-1,00,-1,-1},
{-1,00,00,-1,00,00,00,00,00,00,-1},
{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}
```

Encontrar la salida de un laberinto

- **representado mediante una matriz filas×columnas**
- **Cada posición del laberinto puede tener los valores: `-1:muro, -2:salida, 0:camino`**

Es un problema abordable con Vuelta Atrás:

- **su solución está constituida por una secuencia de decisiones:**
  - **cada uno de los pasos en el camino hacia la salida**
- **en cada decisión elegimos entre un número finito de opciones:**
  - **movimientos a izquierda, arriba, abajo o derecha**

Algoritmo (encuentra salida)

```
{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
{-1, 1,-1,-1,-1,-1,-2,-1,00,00,-1},
{-1, 2,-1,12,-1,-1,27,-1,-1,00,-1},
{-1, 3,-1,11,-1,-1,26,25,24,00,-1},
{-1, 4, 5,10,13,00,00,-1,23,-1,-1},
{-1,-1, 6,-1,14,-1,-1,-1,22,-1,-1},
{-1,-1, 7,-1,15,-1,-1,-1,21,-1,-1},
{-1, 9, 8,-1,16,17,18,19,20,00,-1},
{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}
Solución encontrada
```

En cada paso el algoritmo:

- **escribe el número de paso en la posición actual**
- **realiza llamadas recursivas para cada uno de los movimientos válidos (posiciones con valor 0)**

Cuando para una posición no existe ningún movimiento válido se produce una "vuelta atrás"

- **se retrocede hasta la posición anterior que tenga todavía algún movimiento sin realizar**

```
// encuentra la salida de un laberinto partiendo
// desde la posición xIni, yIni
método resuelveLaberinto(entero xIni, entero yIni,
       entero[][] laberinto) retorna Booleano
  ...    // código en la transparencia siguiente
```

Pseudocódigo (encuentra salida)

```
entero paso = 0  // contador de pasos

método resuelveLaberinto(entero xIni, entero yIni,
        entero[][] laberinto) retorna Booleano
  paso++        laberinto[yIni][xIni]:=paso

  para mov en (izquierda,arriba,abajo,derecha) hacer
    (xNew, yNew):=movimiento(mov,xIni,YIni)

    si laberinto[yNew][xNew] es la salida entonces
      retorna verdadero // salida alcanzada
    sino si laberinto[yNew][xNew] es camino entonces
      // llamada recursiva desde la nueva posición
      si resuelveLaberinto(xNew, yNew,
                           laberinto) entonces
        retorna verdadero // salida alcanzada
      fsi
    fsi
  fhacer
  retorna falso // no hay solución
fmétodo
```


Eficiencia

La eficiencia temporal depende del número de posiciones del laberinto que sean "camino" (`C`):

- **máxima longitud de la solución: `C`**
- **alternativas en cada etapa: 3 (como máximo, menos en la 1ª que podrían ser 4)**
- **eficiencia: $O(3^C)$**
- **`C` está acotado por el tamaño del laberinto (`filas`×`columnas`)**

Eficiencia espacial:

- **laberinto (tabla `filas`×`columnas`): `O(filas×columnas)`**
- **variables auxiliares: `O(1)`**
- **como máximo `C` llamadas recursivas anidadas: `O(C)`**
- **eficiencia: `O(filas×columnas)`**

Algoritmo (camino más corto)

El algoritmo anterior paraba al encontrar la primera solución

- **supongamos que no nos basta con eso, sino que queremos *encontrar el camino más corto* entre la entrada y la salida**

Algoritmo modificado para encontrar el camino más corto:

- **va almacenando la solución parcial en curso (camino recorrido)**
  - **una solución contiene una copia del laberinto (con los pasos ya dados hasta llegar a la posición actual), la posición actual y el número de pasos dados**
- **va apuntando la mejor solución encontrada hasta el momento (`mejorSol`)**
- **a cada llamada recursiva se le pasa *una copia* de la solución en curso**
  - **de esta forma cuando se produce una "vuelta atrás" se vuelve a la situación original**

Pseudocódigo (camino más corto)

```
// encuentra el camino más corto en un laberinto
// comenzando desde la posición xIni, yIni
método caminoMasCortoLaberinto(entero xIni,
   entero yIni, entero[][] labe) retorna entero[][]
  SoluciónLaberinto situaciónInicial:=
     (pasos:=0, x:=Ini,y:=yIni,laberinto:=labe)
  mejorSol.pasos:=valor muy grande
  caminoMasCortoRec(situaciónInicial)
  retorna mejorSol.laberinto;
fmétodo



método caminoMasCortoRec(SoluciónLaberinto lab)
  ...    // código en la transparencia siguiente
```



```
método caminoMasCortoRec(SoluciónLaberinto lab)
  lab.paso++      lab.laberinto[lab.y][lab.x]:=lab.paso
  para mov en (izquierda,arriba,abajo,derecha) hacer
    (xNew, yNew):=movimiento(mov,lab.x,lab.y)
    si lab[yNew][xNew] es la salida entonces
      si lab.pasos<mejorSol.pasos entonces
        // el camino encontrado pasa es el mejor
        mejorSol=copia(lab);
      sino si lab.laberinto[yNew][xNew] es camino entonces
        // Copia el estado actual del laberinto y
        // realiza la llamada recursiva
        SoluciónLaberinto copiaLab = copia(lab)
        copiaLab.(x,y):=(xNew,yNew)
        caminoMasCortoLaberintoRec(copiaLab)
      fsi
    fsi
  fhacer
fmétodo
```


Eficiencia

La eficiencia temporal depende del número de posiciones del laberinto que sean "camino" (`C`) y del máximo número de caminos distintos (`nCaminos`)

- **máxima longitud de la solución: `C` (`C` $\leq$ `filas`$\times$`columnas`)**
- **alternativas en cada etapa: 3 (menos en la 1ª que podrían ser 4)**
- **tiempo para resolver cada camino: $O(3^C)$**
- **eficiencia: $O(nCaminos \cdot 3^C)$ (resultado muy pesimista)**

Eficiencia espacial:

- **una copia del laberinto por cada llamada recursiva $O(filas \times columnas)$**
- **como máximo `C` llamadas recursivas anidadas: $O(C)$**
- **eficiencia: $O(C \cdot filas \times columnas)$**

Posibles mejoras

Sería posible aplicar algunas optimizaciones una vez que sepamos donde está la salida (encontremos la primera solución):

- **elegir primero el movimiento que más nos acerque a ella**
- **cortar el desarrollo de una rama cuando el número de pasos sea mayor que el de la mejor solución encontrada**
- **o mejor todavía: cortar el desarrollo de una rama cuando el número de pasos más el mínimo número de pasos que necesitaríamos hasta la salida sea mayor que el de la mejor solución**

También estaría bien poder realizar una "vuelta atrás temporal" cuando un camino se aleja demasiado de la salida

- **de forma que resolveríamos las posiciones prometedoras por orden de cercanía a la salida**
- **veremos como *esto se puede hacer con "Ramificación y Poda"***

9.6 Bibliografía

[1] Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.

[2] Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.

[3] Sartaj Sahni, *Data Structures, Algoriths, and Applications in Java*. Mc Graw Hill, 2000
El capítulo sobre Vuelta Atrás está en la web:
http://www.cise.ufl.edu/~sahni/dsaaj/