Tema 3: Esquema algorítmico de vuelta atrás

Slides adaptadas a partir del original de Enrique Martín
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Fac. Informática

Contenidos

Introducción

Introducción

- Análisis de eficiencia
  1. Análisis de eficiencia de los algoritmos ✔
- Algoritmos
  2. Divide y vencerás (DV), o *Divide-and-Conquer* ✔
  3. **Vuelta atrás (VA), o *backtracking***
- Estructuras de datos
  4. Especificación e implementación de TADs
  5. Tipos de datos lineales
  6. Tipos de datos arborescentes
  7. Diccionarios
  8. Aplicación de TADs

Vuelta atrás

Vuelta atrás (o *backtracking*)

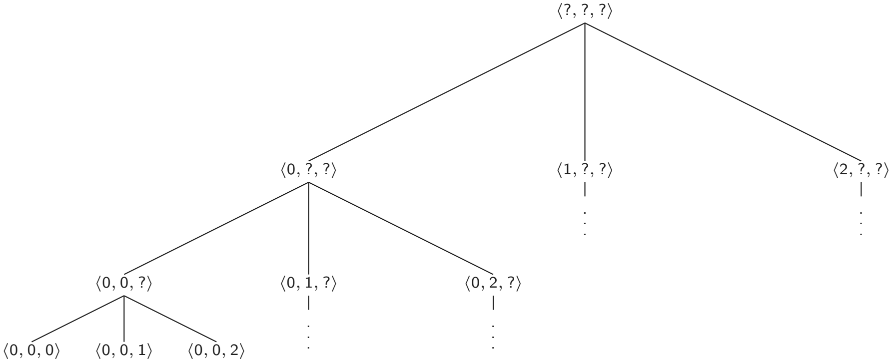
- A veces los problemas que abordamos son tan complejos que la única opción disponible es la *fuerza bruta*: construir **todas las potenciales soluciones** una a una y comprobar si son realmente soluciones
- Esto nos obliga a explorar todo el espacio de soluciones, que usualmente es inmenso (exponencial o factorial). Por lo tanto esta técnica únicamente es aplicable a instancia pequeñas
- Nos centraremos en problemas cuya solución es una tupla $\langle x_0, \ldots, x_n \rangle$, donde x_i es la elección tomada en la etapa i-ésima. Cada elección se escoge de un conjunto *finito* de posibilidades
- La vuelta atrás nos permite organizar esta búsqueda exhaustiva y construir la solución paso a paso

Vuelta atrás (o *backtracking*)

- Para recorrer todo el espacio de soluciones comenzaremos con una solución vacía $\langle ?, \ldots, ? \rangle$
- Para cada posible valor $v_0 \ldots v_k$ de la primera elección x_0 generamos una solución parcial: $\langle v_0, ?, \ldots, ? \rangle$, $\langle v_1, ?, \ldots, ? \rangle$, $\ldots$, $\langle v_k, ?, \ldots, ? \rangle$
- Para cada solución parcial completada hasta el nivel k, probamos todas las posibilidades para la siguiente elección $k+1$
- Repetimos este proceso hasta llegar a una solución completa, que debemos verificar que cumple las restricciones del problema
- Podemos ver este **espacio de búsqueda** como un **árbol**

Vuelta atrás (o *backtracking*)

Imaginemos una solución de 3 elecciones $\langle x_0, x_1, x_2 \rangle$, donde cada elección toma un valor entre 0 y 2. El espacio de búsqueda sería:



En total el espacio de búsqueda tiene tamaño $3^3 = 27$ soluciones candidatas. ¿Cómo generarlas todas de manera ordenada?

Esquema de vuelta atrás

```
proc vuelta_atras(sol : tupla, k : nat)
  for c in candidatos(k)
    sol[k] := c
    if es_solucion(sol,k) then
      procesar_solucion(sol)
    else if es_completable(sol,k) then
      vuelta_atras(sol,k+1)
```

- *sol* es una solución parcial con elecciones **correctas** hasta la posición k **no incluida**
- Para cada posible valor c para la posición k comprobamos:
  - Si ya es solución, la procesamos (guardamos, mostramos por pantalla, etc.)
  - Si no es solución pero es *completable* (es decir, se puede seguir rellenando hasta formar una solución) realizamos una llamada recursiva para seguir rellenando desde la posición $k+1$

Problema de las 4 reinas

Problema de las 4 reinas

Problema

Queremos colocar 4 reinas en un tablero de ajedrez 4×4 de tal manera que no puedan atacarse entre ellas.
(La reina ataca en vertical, horizontal y en cualquier diagonal)

- Está claro que tendremos que poner cada reina en una columna diferente, así que podemos representar la solución como $\langle x_0, x_1, x_2, x_3 \rangle$, donde $x_i \in \{0, 1, 2, 3\}$ es la fila donde está la reina de la columna i. P.ej. $\langle 2, 0, 3, ? \rangle$ sería:

	0	**1**	**2**	**3**
0		Q		
1				
2	Q			
3			Q	

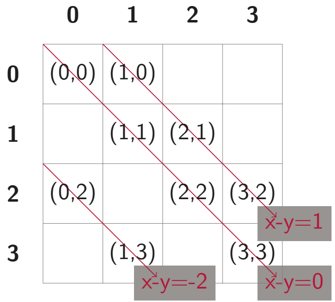
Problema de las 4 reinas

```
proc 4_reinas(sol : tupla, k : nat)
  for c in [0..4)
    sol[k] := c
    if es_solucion(sol,k) then
      procesar_solucion(sol,k)
    else if es_completable(sol,k) then
      4_reinas(sol,k+1)
```

- procesar_solucion muestra los valores por pantalla $\in \mathcal{O}(1)$
- Para implementar es_solucion y es_completable necesitamos comprobar que la última reina colocada (posición k) cumple que:
  1. No comparte fila con reinas anteriores
  2. No comparte diagonal ascendente con reinas anteriores
  3. No comparte diagonal descendente con reinas anteriores
- Para ello vamos a basarnos en una función no_ataca(sol,k) que determina si la reina en la posición k de *sol* no ataca a las reinas anteriores $\in [0..k)$

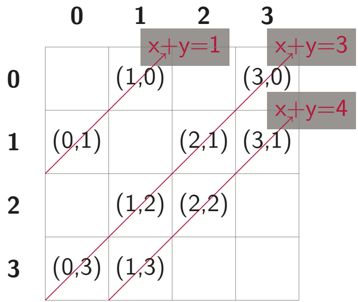
Diagonales en el tablero

- Para determinar si dos posiciones (x, y) y (x', y') están en la misma diagonal, distinguiremos entre diagonales descendentes y ascendentes (de izquierda a derecha)
- Las casillas de una misma diagonal descendente cumplen que $x - y = K$



Diagonales en el tablero

- De la misma manera, las casillas de una misma diagonal ascendente cumplen que $x + y = K$



Comprobar soluciones

```
fun diag_desc(x,y : nat) dev d : ent
  d := x − y

fun diag_asc(x,y : nat) dev d : ent
  d := x + y

fun no_ataca(sol : tupla, k : nat) dev b : bool
  b := true
  i := 0
  while b ∧ i < k // Compruebo cada reina 'i' con 'k'
    b := b ∧
      (sol[i] ≠ sol[k]) ∧
      (diag_asc(i,sol[i]) ≠ diag_asc(k,sol[k])) ∧
      (diag_desc(i,sol[i]) ≠ diag_desc(k,sol[k]))
    i := i + 1
```

Comprobar soluciones

- Usando la función no_ataca es sencillo implementar es_solucion y es_completable
- En ambos casos suponemos que la solución parcial es correcta hasta $k-1$. Únicamente tenemos que comprobar que la reina en posición k es correcta
- La diferencia entre es_solucion y es_completable es la comprobación de la longitud de la solución

```
fun es_solucion(sol : tupla, k : nat) dev b : bool
  b := (k = 3) ∧ no_ataca(sol,k)

fun es_completable(sol : tupla, k : nat) b : bool
  b : = (k < 3) ∧ no_ataca(sol,k)
```

Mejorar las 4 reinas

- El algoritmo de las 4 reinas genera *potencialmente* $4! = 24$ soluciones (permutaciones de $\langle 0, 1, 2, 3 \rangle$), es decir, hojas en el espacio de búsqueda
- El espacio de búsqueda total recorrido por este algoritmo es $1 + 4 + 4 \cdot 3 + 4 \cdot 3 \cdot 2 = 41$
- Además, la comprobación `no_ataca(s,k)` necesita recorrer $k - 1$ posiciones para comprobar que la solución actual es válida. ¿Se puede mejorar?
- Sí, utilizando **marcaje**: los **marcadores** son parámetros adicionales de **entrada/salida** con información adicional que acelera las comprobaciones
- Hay que **actualizar** los marcadores justo **antes** de realizar cada llamada recursiva, y **restaurarlos** a su valor original justo **después** de la llamada recursiva

Mejorar las 4 reinas

- En el caso de las 4 reinas, para acelerar las llamadas a no_ataca(s,k) deberíamos llevar la cuenta de las filas y diagonales ocupadas hasta la posición $k-1$
- Los marcadores serían una serie de tuplas *filas*, $diag_{asc}$, $diag_{des}$:
  - *filas*$[i]$ establece si la fila i-ésima está ocupada
  - $diag_{asc}[i]$ establece si la diagonal ascendente i-ésima está ocupada
  - $diag_{desc}[i]$ establece si la diagonal descendente i-ésima está ocupada[1]

  Con esta información, el coste de no_ataca(s,k) sería constante

```
fun no_ataca_m(sol : tupla, k : nat,
               filas, d_asc, d_desc : tupla) dev b : bool
  b := ¬filas[sol[k]] ∧
       ¬d_asc[diag_asc(k, sol[k])] ∧
       ¬d_desc[diag_desc(k, sol[k])]
```

[1] **¡Ojo!** Si la tupla d_desc indexa desde 0 habrá que modificar el código de diag_desc() para evitar generar diagonales negativas.

Mejorar las 4 reinas

De la misma manera habría que extender los métodos es_solucion y es_completable para que utilicen los marcadores:

```
fun es_solucion_m(sol : tupla, k : nat,
                  filas, d_asc, d_desc : tupla) dev b : bool
  b := (k = 3) ∧
       no_ataca_m(sol,k,filas,d_asc,d_desc)

fun es_completable_m(sol : tupla, k : nat,
                  filas, d_asc, d_desc : tupla) dev b : bool
  b := (k < 3) ∧
       no_ataca_m(sol,k,filas,d_asc,d_desc)
```

Mejorar las 4 reinas

Finalmente, habría que **marcar y desmarcar** los marcadores antes y después de realizar la llamada recursiva:

```
proc 4_reinas_m( sol : tupla, k : nat,
                 filas, d_asc, d_desc : tupla)
  for c in [0..4)
    sol[k] := c
    if es_solucion(sol,k,filas,d_asc,d_desc) then
      procesar_solucion(sol,k)
    else if es_completable(sol,k,filas,d_asc,d_desc) then
      filas[c] := true                 //Marcado
      d_asc[diag_asc(k,c)] := true     //Marcado
      d_desc[diag_desc(k,c)] := true   //Marcado
      4_reinas_m(sol,k+1,filas,d_asc,d_desc)
      filas[c] := false                 //Desmarcado
      d_asc[diag_asc(k,c)] := false     //Desmarcado
      d_desc[diag_desc(k,c)] := false   //Desmarcado
```

Parar la búsqueda de manera prematura

- Los algoritmos que hemos visto recorren **completamente** el espacio de soluciones, mostrando todas las soluciones encontradas
- En algunas ocasiones no queremos conocer *todas* las soluciones, sino únicamente *una solución*. En estos casos podemos parar la búsqueda tan pronto como encontremos la primera solución
- Para abortar la búsqueda debemos utilizar un parámetro de entrada/salida encontrada : **bool** (similar a los marcadores) que comienza con valor **false**. Cuando se encuentra la primera solución, se asigna **true**
- Extendemos el bucle de la búsqueda para iterar **únicamente mientras** $\neg$encontrada. De esta manera todas las llamadas recursivas pendientes irán terminando en cadena

Las n-reinas

- Generalizar nuestra solución de las 4-reinas a un número arbitrario *n* de reinas es sencillo, solo necesitamos un nuevo parámetro *n* y adaptar ligeramente nuestro código:

```
proc n_reinas( sol : tupla , k , n : nat ,
                    filas , d_asc , d_desc : tupla)
  for c in [0..n)   // n candidatos
    sol[k] := c
    if es_solucion(sol,k,n,filas,d_asc,d_desc) then
      procesar_solucion(sol,k,n)
    else if es_completable(sol,k,n,filas,d_asc,d_desc) then
      filas[c] := true                    //Marcado
      d_asc[diag_asc(k,c)] := true        //Marcado
      d_desc[diag_desc(k,c)] := true      //Marcado
      n_reinas(sol,k+1,n,filas,d_asc,d_desc)
      filas[c] := false                   //Desmarcado
      d_asc[diag_asc(k,c)] := false       //Desmarcado
      d_desc[diag_desc(k,c)] := false     //Desmarcado
```

Las n-reinas

La función n_reinas encuentra todas las posibles maneras de colocar n reinas en un tablero $n \times n$, pero todavía quedan algunas cosas por decidir:

- ¿Qué tamaño tiene sol?
- ¿Qué tamaño tiene filas?
- ¿Qué tamaño tiene d_asc?
- ¿Qué tamaño tiene d_desc?
- ¿Cómo cambia es_solucion y es_completable?
- ¿procesar_solucion cambia mucho?

Coste de n_reinas

Obtener la recurrencia de un algoritmo de vuelta atrás normalmente genera ecuaciones complicadas. Por ejemplo, para las n-reinas tendríamos que:

- Si suponemos que es_completable siempre devuelve **true**, haremos n llamadas recursivas en cada invocación:

$$T(n,k) = \begin{cases} c_1 n & \text{si } k = n-1 \\ nT(n,k+1) + c_2 n & \text{si } k < n-1 \end{cases}$$

- Si suponemos que es_completable revisa al menos que no se repitan las filas, en cada invocación haremos **una llamada recursiva menos**:

$$T(n,k) = \begin{cases} c_1 n & \text{si } k = n-1 \\ (n-k)T(n,k+1) + c_2 n & \text{si } k < n-1 \end{cases}$$

- Estas recurrencias son **difíciles de expandir** y además **no encajan con las plantillas de coste**.

Coste de algoritmos de vuelta atrás

- En estos casos seguiremos un enfoque alternativo:
  1. Calcular el número total de invocaciones que realizaremos al recorrer el árbol de búsqueda. Si este cálculo es complicado, podemos sobreaproximarlo.
  2. Calcular el coste que tendrá cada invocación por sí sola, sin tener en cuenta sus llamadas recursivas. Si ese cálculo es complejo, se puede sobreaproximar.
  3. Multiplicar el número de invocaciones por el coste de cada invocacioń.

Coste de las n-reinas (versión simplista)

Si consideramos que es_completable devuelve siempre **true** tendremos que:

1. El árbol de llamadas de n_reinas(n,0) tiene n niveles.

   $k = 0$) En el primer nivel hay una llamada
   $k = 1$) En el segundo nivel hay n llamadas
   $k = 2$) En el tercero hay $n.n = n^2$ llamadas
   ...
   $k = n - 1$) En el último nivel hay n^{n-1} llamadas

   $$\text{En total hay } \sum_{i=0}^{n-1} n^i \leq n \cdot n^{n-1} = n^n \text{ llamadas}$$

2. Cada llamada, si se usan marcadores, tendrá un coste en $O(n)$ porque el cuerpo del bucle tiene coste constante
3. En total, el coste de n_reinas(n,0) será $O(n^n \cdot n)$

Coste de las n-reinas

Si consideramos que es_completable al menos evita filas duplicadas, tendremos que:

1. El árbol de llamadas de n_reinas(n,0) tiene n niveles.

$k = 0$) En el primer nivel hay una llamada
$k = 1$) En el segundo nivel hay n llamadas
$k = 2$) En el tercer nivel hay $n \cdot (n-1)$ llamadas
$k = 3$) En el cuarto nivel hay $n \cdot (n-1) \cdot (n-2)$
...
$k = n-1$) En el último nivel hay $n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 = n!$ llamadas

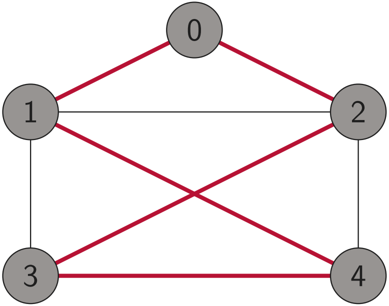
$$\text{En total hay } \sum_{i=1}^{n} \frac{n!}{i!} \leq n \cdot n! \text{ llamadas}$$

2. Cada llamada, si se usan marcadores, tendrá un coste en $O(n)$ porque el cuerpo del bucle tiene coste constante
3. En total, el coste de n_reinas(n,0) será $O(n \cdot n! \cdot n) = O(n^2 \cdot n!)$

Ciclos hamiltonianos

Ciclos hamiltonianos

- Un **ciclo hamiltoniano** es un recorrido de un grafo que recorre cada vértice **exactamente una vez** y regresa al vértice de origen



$0 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 0$

Problema

Queremos encontrar todos los ciclos hamiltonianos de un grafo que **comienzan en el nodo** 0

Ciclos hamiltonianos

- Para un grafo de n nodos podemos representar la solución como una tupla $\langle 0, x_1, x_2, \ldots, x_{n-1} \rangle$, donde $x_1, \ldots, x_{n-1} \in \{1, 2, \ldots n-1\}$
- Esta solución la podemos ir construyendo paso a paso, así que encaja perfectamente con el esquema de *vuelta atrás*
- Dada una solución parcial `sol` correcta hasta el paso $k-1$, saber si `es_solucion(sol,k)` implica:
  1. `k = n−1`
  2. Existe una arista `sol[k−1]` $\rightarrow$ `sol[k]`
  3. Existe una arista `sol[k]` $\rightarrow$ 0, es decir, cerramos el ciclo
  4. El nodo `sol[k]` no aparece ya en `sol [1.. k)`[2]
- Por otro lado, comprobar que `sol` es completable tras añadir el nodo k únicamente requiere verificar que:
  1. `k < n−1`
  2. Existe una arista `sol[k−1]` $\rightarrow$ `sol[k]`
  3. El nodo `sol[k]` no aparece ya en `sol [1.. k)`

[2] Porque el nodo 0 nunca será candidato

Ciclos hamiltonianos: consultar el grafo

- Descubrir si existe una arista $a \to b$ es comprobar los datos de entrada del problema. Depende de la representación de nuestro grafo
- Existen varias maneras de representar grafos (*lo veréis con detalle en la asignatura de 3º «Métodos algorítmicos en resolución de problemas»*), pero aquí supondremos la versión más sencilla: **una matriz de adyacencia** `adj` de tipo `matriz<`**bool**`>`
  1. Si `adj[a][b]` = **true** entonces existe una arista de $a \to b$
  2. Si `adj[a][b]` = **false** entonces no existe ninguna arista $a \to b$
- Si el grafo tiene n nodos la matriz de adyacencia ocupará $n \times n$ posiciones, pero **cada comprobación será constante**

Ciclos hamiltonianos: comprobación de restricciones

- Verificar que el nodo en sol[k] no aparecía anteriormente en sol[1..k) requiere:
  1. Recorrer las $k-1$ posiciones $\in \mathcal{O}(k)$: **¡lento!**
  2. Utilizar una tupla de n posiciones booleanas como marcador para «tachar» los nodos ya utilizados. Usaríamos n posiciones de memoria adicional, pero la comprobación será en $\mathcal{O}(1)$. Eso sí, recordad que hay que marcar **antes** y desmarcar **después** de la llamada recursiva

```
fun es_solucion(sol : tupla, k,n : nat, adj : matriz<bool>,
                usado : tupla) dev b : bool
  b := ¬usado[sol[k]] ∧ (k = n−1) ∧
       adj[sol[k−1]][sol[k]] ∧ adj[sol[k]][0]

fun es_completable(sol : tupla, k,n : nat,
    adj : matriz<bool>, usado : tupla) dev b : bool
  b := ¬usado[sol[k]] ∧ (k < n−1) ∧ adj[sol[k−1]][sol[k]]
```

```
proc ciclo_hamiltoniano(sol : tupla, k,n : nat,
        adj : matriz<bool>, usado : tupla)

   // Suponemos k > 0 y existen aristas
   // sol[0] -> sol[1] -> sol[2] -> ... -> sol[k-1]
   for c in [1..n) // No probamos el nodo 0
      sol[k] = c
      if es_solucion(sol,k,n,adj,usado) then
         procesar_solucion(sol)
      else if es_completable(sol,k,n,adj,usado) then
         usado[c] := true    // Marcado
         ciclo_hamiltoniano(sol,k+1,n,adj,usado)
         usado[c] := false   // Desmarcado
```

Como el ciclo siempre empieza en el nodo 0, la llamada inicial será con sol [0] := 0, k := 1 y usado[0] := **true**

Coste de la búsqueda de ciclos hamiltonianos

1. El árbol de llamadas de `ciclo_hamiltoniano(n,1)` tiene $n-1$ niveles. En cada nivel tenemos un candidato menos:

$k=1)$ En el primer nivel hay una llamada
$k=2)$ En el segundo nivel hay $n-1$ llamadas
$k=3)$ En el tercer nivel hay $(n-1)\cdot(n-2)$ llamadas
$k=4)$ En el cuarto nivel hay $(n-1)\cdot(n-2)\cdot(n-3)$
...
$k=n-1)$ En el último nivel hay $(n-1)\cdot(n-2)\cdots 3\cdot 2=(n-1)!$ llamadas

$$\text{En total hay } \sum_{i=1}^{n-1}\frac{(n-1)!}{i!}\leq (n-1)\cdot(n-1)! \text{ llamadas}$$

2. Cada llamada, si se usan marcadores, tendrá un coste en $O(n)$ porque el cuerpo del bucle tiene coste constante

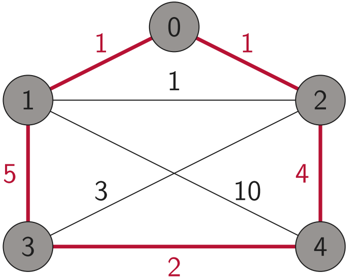
3. En total, el coste de `ciclo_hamiltoniano(n,1)` será $O((n-1)\cdot(n-1)!\cdot n)=O((n-1)\cdot n!)$

Problema del viajante

Problema

Dado un mapa con distancias entre ciudades, ¿cuál es el camino **más corto** que visita todas las ciudades **exactamente una vez** y vuelve a la ciudad origen?

- El problema de los ciclos hamiltonianos es similar al **problema del viajante** (*Travelling Salesperson Problem, o TSP*)
- Está claro que estamos buscando un ciclo hamiltoniano, pero no cualquiera, sino aquel que minimice la distancia recorrida.



$0 \to 1 \to 3 \to 4 \to 2 \to 0 \equiv 13$

$0 \to 1 \to 4 \to 3 \to 2 \to 0 \equiv 17$

Problema del viajante de comercio

- El problema TSP es un problema clásico de **optimización**: existe una *función objetivo* que queremos minimizar o maximizar (en este caso minimizar la distancia recorrida)
- Para este tipo de problemas también se puede utilizar vuelta atrás, aunque deberemos adaptarlo para conservar la mejor solución hasta el momento
- Por lo tanto el esquema es muy parecido: construiremos paso a paso todas las posibles soluciones (mientras sean completables) pero **únicamente** procesaremos una solución cuando su *coste* sea **mejor** que la mejor solución encontrada hasta el momento
- Además, utilizaremos la mejor solución encontrada hasta el momento junto con *estimaciones optimistas* para «podar» ramas que no llevan a soluciones mejores

Vuelta atrás aplicado a optimización

```
proc vuelta_atras_opt(sol:tupla, k:nat, valor:num,
                      sol_mejor:tupla, valor_mejor:num)
  for c in candidatos(k)
    sol[k] := c
    nuevo_valor := actualiza(valor, sol, k)
    if es_solucion(sol, k) then
      if mejor(nuevo_valor, valor_mejor) then
        sol_mejor := sol, valor_mejor := nuevo_valor
    else if es_completable(sol, k) ∧
        es_prometedora(sol, k, nuevo_valor, valor_mejor) then
      vuelta_atras_opt(sol, k+1, nuevo_valor,
                       sol_mejor, valor_mejor)
```

- También se pueden añadir marcadores para acelerar las comprobaciones *(omitido por simplicidad)*

Vuelta atrás aplicado a optimización

Han aparecido nuevas funciones:

- actualiza (valor , sol , k): Considera que valor es el coste de la solución parcial sol [0.. k). Devuelve el coste considerando la decisión tomada en la posición k, es decir, sol [0.. k]
- mejor(nuevo_valor,valor_mejor): decide si nuevo_valor es un valor más beneficioso que valor_mejor
- es_prometedora(sol,k,nuevo_valor,valor_mejor): comprueba si será posible completar la solución parcial sol [0.. k], que tiene valor nuevo_valor, y conseguir una solución completa con un valor más beneficioso que valor_mejor. Para esta comprobación se utilizan **aproximaciones**:
  - Si es_prometedora devuelve **false** es porque es **imposible** completar sol y conseguir una solución más beneficiosa que valor_mejor
  - Si es_prometedora devuelve **true** *existe la posibilidad potencial* de completar sol y conseguir una solución más beneficiosa que valor_mejor, pero si al final eso no ocurre no pasaría nada

Problema del viajante de comercio

```
proc tsp(sol:tupla,k,n:nat,adj:matriz<nat>,
    usado:vector<bool>,valor,valor_mejor:num,
    sol_mejor:tupla,min_arista:nat)
  for c in [1..n):
    sol[k] := c
    nuevo_valor := actualiza_tsp(valor,sol,k,n,adj)
    if es_solucion_tsp(sol,k,n,adj,usado) then
      if nuevo_valor < valor_mejor then
        valor_mejor := nuevo_valor
        sol_mejor := sol
    else if es_completable_tsp(sol,k,n,adj,usado) ∧
        es_prometedora_tsp(sol,k,n,nuevo_valor,
                           valor_mejor,min_arista) then
      usado[c] := true; //Marcado
      tsp(sol,k+1,n,adj,usado,nuevo_valor,
          valor_mejor,sol_mejor,min_arista)
      usado[c] := false //Desmarcado
```

Funciones usadas en TSP

- Ahora la matriz de adyacencia no contiene booleanos sino números naturales, ya que debe almacenar el coste de cada arista. Supondremos que adj[a][b] ≤ 0 si no existe camino $a \to b$ (*Las aristas con coste* 0 *tiene poco sentido*)
- El código de es_solucion_tsp y es_completable_tsp es igual que el de los ciclos hamiltonianos pero cambiando la comprobación de existencia de arista a $adj[sol[k-1]][sol[k]] > 0$. Por ejemplo:

```
fun es_solucion_tsp(sol:tupla,k,n:nat,adj:matriz<nat>,
                     usado:vector<bool>) dev b:bool
  b := ¬usado[sol[k]] ∧ (k = n−1) ∧
       adj[sol[k−1]][sol[k]] > 0 ∧
       adj[sol[k]][0] > 0
```

Funciones usadas en TSP

- Actualizar el valor de una solución parcial es sencillo
  1. Si la nueva solución sigue siendo parcial, sumamos la distancia de la última arista $sol[k-1] \rightarrow sol[k]$
  2. Si la nueva solución es completa, sumamos la distancia de la última arista $sol[k-1] \rightarrow sol[k]$ pero también de la arista que cierra el ciclo $sol[k] \rightarrow 0$

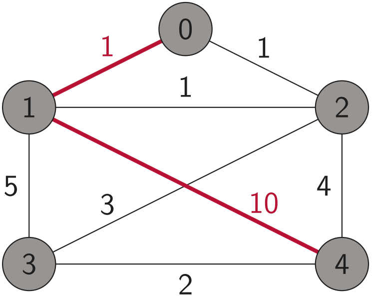
```
fun actualiza_tsp(valor:num, sol:tupla, k, n:nat,
                  adj:matriz<int>) dev v:num
  if k = n - 1 then
    v := valor + adj[sol[k-1]][sol[k]] + adj[sol[k]][0]
  else
    v := valor + adj[sol[k-1]][sol[k]]
```

Funciones usadas en TSP

- Para decidir si una solución parcial sol [0.. k] es prometedora, aproximaremos su coste considerando *el caso más optimista posible* conociendo el valor de la menor arista del grafo: min_arista > 0
- Si tenemos una solución parcial sol [0.. k] con distancia v, sabemos que le faltan $n - k$ aristas para completar el ciclo. **En el mejor de los casos** esas aristas tendrían valor min_arista, por lo tanto sumarían como mínimo una distancia adicional de (n−k)∗min_arista
- Si este coste mínimo es igual o superior al mejor coste encontrado hasta el momento no tiene sentido tratar de completar la solución parcial: **en ningún caso podrá genera una solución completa mejor que la que ya tenemos**

```
fun es_prometedora_tsp(sol:tupla,k,n:nat,valor,
                       valor_mejor:num, min_arista:nat) dev b:bool
  b := (valor + min_arista * (n-k)) < valor_mejor
```

Ejemplo de solución prometedora en TSP



- En este grafo la **arista mínima** tiene distancia 1
- Imaginad que ya hemos encontrado la solución $\langle 0, 1, 3, 4, 2 \rangle$ con distancia $0 \xrightarrow{1} 1 \xrightarrow{5} 3 \xrightarrow{2} 4 \xrightarrow{4} 2 \xrightarrow{1} 0 = 13$
- La solución parcial $\langle 0, 1, 4, ?, ? \rangle$ tiene distancia 11. Como le faltan 3 aristas, la mejor distancia total que puede llegar a tener es $11 + 3 \times 1 = 14$
- No es necesario seguir explorando, puesto que **no podrá mejorar**

Coste de la búsqueda de ciclos hamiltonianos

1. El árbol de llamadas de tsp(n,1) tiene $n-1$ niveles y es exactamente el mismo que el de ciclo_hamiltoniano(n,1). Por lo tanto tiene

$$\sum_{i=1}^{n-1} \frac{(n-1)!}{i!} \leq (n-1) \cdot (n-1)! \text{ llamadas}$$

2. Ignorando las llamadas recursivas, cada invocación a tsp tiene un coste lineal en $\mathcal{O}(n)$ porque el bucle "**for** c in [1..n)" realiza $n-1$ iteraciones y cada iteración sigue siendo constante (actualiza_tsp y es_prometedora_tsp son constantes).
3. En total, el coste de ciclo_hamiltoniano(n,1) será $O((n-1) \cdot (n-1)! \cdot n) = O((n-1) \cdot n!)$

Problema de la mochila 0-1

Problema de la mochila 0-1

Problema

Tenemos una mochila que admite un peso máximo P, y tenemos una serie de n artículos $a_0, \ldots, a_{n-1}$. Cada artículo tiene un peso p_i y un valor v_i. Queremos llenar la mochila de artículos sin sobrepasar el límite P pero maximizando el valor de lo que nos llevamos.

- Nuestras soluciones siguen siendo tuplas de n elementos $\langle x_0, \ldots, x_{n-1} \rangle$, pero ahora cada elección es booleana:
  1. sol[i] = **true**: meto el artículo a_i en la mochila
  2. sol[i] = **false**: dejo el artículo a_i fuera de la mochila
- Al igual que TSP es un problema de optimización, pero ahora queremos **maximizar** el valor total de los artículos de la mochila

Funciones del problema de la mochila 0-1

Este problema encaja perfectamente en el esquema vuelta_atras_opt, pero tenemos que definir las distintas funciones:

- actualiza (valor , sol , k): añade el valor del artículo a_k a valor si este elemento ha sido elegido.
- es_solucion (sol , k): cualquier solución completa con peso total $\leq P$ es una solución válida para el problema
- mejor(nuevo_valor, valor_mejor): Será mejor si es **mayor**
- es_completable(sol , k): cualquier solución parcial con peso $\leq P$ es completable

Soluciones prometedoras

- Quizá sea más complicado detectar cuándo una solución parcial es prometedora. Podríamos utilizar la *densidad* d_i de los artículos[3], es decir, la proporción $d_i = \frac{v_i}{p_i}$. El artículo con mayor densidad será el que más valor nos proporciona por unidad de peso
- Sea D la densidad más alta de todos los artículos y p el peso actual de la mochila para una solución parcial con coste v
- En este caso, el mayor valor que podríamos obtener sería rellenar todo el peso restante con artículos de la máxima densidad, es decir, $(P - p) * D$
- Por lo tanto una aproximación optimista sería considerar que una solución parcial con valor v se podría completar (en el mejor de los casos) hasta un valor de $v\_max = v + (P - p) * D$

[3] Se pueden encontrar otras aproximaciones

Soluciones prometedoras

- Dependiendo del mejor valor valor_mejor encontrado hasta el momento y del valor máximo v_max posible tenemos que:
  1. Si v_max $<=$ valor_mejor: no tiene sentido explorar esa solución parcial, puesto que en el mejor caso llegará a una solución tan buena como la que ya tenemos $\rightarrow$ **podar**
  2. Si v_max $>$ valor_mejor: la solución parcial tiene potencial para mejorar la mejor solución encontrada hasta el momento, así que debemos explorarla

Ejemplos de soluciones prometedora

	Art 0	*Art* 1	*Art* 2	*Art* 3	*Art* 4
Valor	21	12	6	9	10,5
Peso	10	6	6	6	6
Densidad	**2,1**	2	1	1,5	1,75

- Consideremos una mochila con peso máximo $P = 13$, y que durante la búsqueda con vuelta atrás ya hemos encontrado la solución $\langle t, f, f, f, f \rangle$ con valor total 21.
- La solución parcial $\langle f, f, t, ?, ? \rangle$ tiene peso $p = 6$ y valor $v = 6$. Si llenásemos los 7 kilos que nos quedan con el artículo más denso posible alcanzaríamos $6 + 2{,}1 \times 7 = 20{,}7 \leq 21 \rightarrow$ **No es una solución prometedora**
- La solución parcial $\langle f, t, ?, ?, ? \rangle$ tiene peso $p = 6$ y valor $v = 12$. Si llenásemos los 7 kilos que nos quedan con el artículo más denso posible alcanzaríamos $12 + 7 \times 2{,}1 = 26{,}7 > 21 \rightarrow$ **¡Solución prometedora!**

Coste del problema de la mochila 0-1

- El cálculo del coste del problema de la mochila 0-1 es más sencillo porque podemos crear una recurrencia «simple».
- En cada llamada recursiva con parámetro k probamos **dos candidatos**: incorporar el artículo k-ésimo (sol[k] = **true**) o dejarlo fuera de la mochila (sol[k] = **false**).
- Por cada candidato hacemos una llamada recursiva con $k+1$.
- Ignorando la llamada recursiva, en cada invocación se ejecutan una cantidad constante de instrucciones.
- En resumen, considerando i el número de artículos por procesar (es decir $i = n - k$) la recurrencia será:

$$T(i) = \begin{cases} c_1 & \text{si } i = 1 \\ 2T(i-1) + c_2 & \text{si } i > 1 \end{cases} \in \mathcal{O}(2^i)$$

- La llamada inicial es con $k = 0$, luego el coste es términos de n será $\mathcal{O}(2^n)$.

Conclusiones

Conclusiones

- Cuando recurrimos a *vuelta atrás* es porque no hemos encontrado un algoritmo más ingenioso. El coste suele ser muy elevado: exponencial, factorial, etc.
- En la mayoría de los casos, usamos *vuelta atrás* para resolver problemas *complicados* cuya mejor solución conocida es listar todas las combinaciones posibles. Estos problemas están en una *clase de complejidad* llamada **NP** (relacionada con NP-completo y NP-difícil).
- Sin embargo, alguno de los problemas tratados admite una solución utilizando el esquema algorítmico de *programación dinámica*, por ejemplo el problema de la mochila 0-1.[4]

[4] Más detalles en MARP de 3º

Conclusiones

- Los problemas que hemos visto tratan de proporcionar valores a unas variables x_i, cada una con un dominio de valores posibles que deben satisfacer unas determinadas restricciones
- Este tipo de problemas es muy común, y de hecho cuentan con su propio paradigma de programación llamado *programación con restricciones*.
- Existen sistemas específicos para resolver este tipo de problemas (p.ej. ILOG o Gecode) y también existen resolutores de restricciones integrados en otros paradigmas de programación (p.ej. programación lógica con restricciones, CLP)

```
:- use_module(library(clpfd)).

reinas4(Xs) :-
  % Xi es la fila de la reina en columna 'i'
  Xs = [X1,X2,X3,X4], Xs ins 0..3,
  all_different(Xs),

  % Diagonales ascendentes
  DUs = [DU1, DU2, DU3, DU4], DUs ins 0..6,
  DU1 #= 0+X1, DU2 #= 1+X2, DU3 #= 2+X3, DU4 #= 3+X4,
  all_different(DUs),

  % Diagonales descendentes
  DDs = [DD1, DD2, DD3, DD4], DDs ins -3..3,
  DD1 #= 0-X1, DD2 #= 1-X2, DD3 #= 2-X3, DD4 #= 3-X4,
  all_different(DDs),

  % Busca valores para las variables Xi
  label(Xs).
```

Las soluciones encontradas por **SWI-Prolog** serían:

```
?- reinas4(L).
L = [1, 3, 0, 2] ; L = [2, 0, 3, 1] ;
```

Bibliografía

Bibliografía

- Narciso Martí, Yolanda Ortega, Alberto Verdejo. *Estructuras de Datos y Métodos Algorítmicos: 213 Ejercicios resueltos (2ª Edición)*. Garceta, 2013. **Capítulo 14**.
  http://cisne.sim.ucm.es/record=b3290150~S6*spi
  También está disponible la versión de Pearson Prentice-Hall:
  *http://cisne.sim.ucm.es/record=b2789524~S6*spi*
- Gilles Brassard, Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996. **Capítulo 9.6**.
  Hay pocos ejemplares pero *existe versión en español*
  *http://cisne.sim.ucm.es/record=b2082127~S6*spi*