Metodología y Tecnología de la Programación

Curso 2007-2008

Esquemas algorítmicos. Backtracking

Yolanda García Ruiz D228 ygarciar@fdi.ucm.es
Jesús Correas D228 jcorreas@fdi.ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(elaborado a partir de [GV00], [PS03] y notas de S. Estévez y R. González del Campo)

Bibliografía

- **Importante**: Estas transparencias son un material de apoyo a las clases presenciales y no sustituyen a la bibliografía básica ni a las propias clases presenciales para el estudio de la asignatura
- Bibliografía básica:
  - [GV00]: capítulo 6
  - [PS03][1]: capítulo 7
- Bibliografía complementaria:
  - [NN98]: capítulo 5
  - [GC01][2]: capítulo 6
  - [BB97]: capítulo 9 (apartado 9.6)
- Ejercicios resueltos:
  - [MOV04]: capítulo 14

(1) [PS03] J. I. Peláez Sánchez et al. *Análisis y Diseño de algoritmos: un enfoque teórico y práctico*, Universidad de Málaga, 2003

(2) [GC01] D. Giménez Cánovas *Apuntes y problemas de algorítmica*, Universidad de Murcia, 2001. Disponible en

`http://servinf.dif.um.es/˜domingo/apuntes/Algoritmica/apuntes.pdf`

Esquemas algorítmicos. *Backtracking*

1. $\boxed{\text{Características generales}}$
2. Esquema general de un algoritmo de *backtracking*
3. Estudio de complejidad
4. Problema de la mochila 0-1
5. Asignación de tareas
6. El caballo de ajedrez
7. Coloreado de mapas
8. Las *n* reinas
9. Trenes
10. Franqueo de Postales
11. Ciclos Hamiltonianos

Características generales

- Las técnicas vistas hasta ahora intentan construir la solución basándose en ciertas propiedades de ésta
- Sin embargo, ciertos problemas no pueden solucionarse con ninguna de las técnicas anteriores.
  - La única manera de resolver estos problemas es a través de un estudio exhaustivo de un conjunto de posibles soluciones.
- La técnica de *backtracking* permite realizar este estudio exhaustivo
- Cada solución es el resultado de una secuencia de decisiones
  - Pero a diferencia del método voraz, las decisiones pueden deshacerse ya sea porque no lleven a una solución o porque se quieran explorar todas las soluciones (para obtener la solución óptima)
- Existe una función objetivo que debe ser satisfecha u optimizada por cada selección
- Las etapas por las que pasa el algoritmo se pueden representar mediante un árbol de expansión (o *árbol del espacio de estados*).
- El árbol de expansión no se construye realmente, sino que está implícito en la ejecución del algoritmo
- Cada nivel del árbol representa una etapa de la secuencia de decisiones

Características generales. Ejemplo de *backtracking*

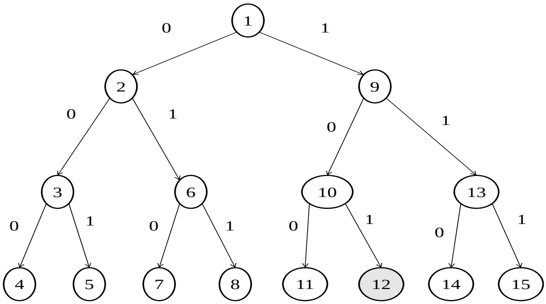
- Se debe diseñar un algoritmo que permita obtener un subconjunto de números dentro del conjunto $\{13, 11, 7\}$ cuya suma sea 20
- ¿Cómo representamos la solución?
  Tupla o vector de 3 elementos $[x_1, x_2, x_3]$ con $x_i \in \{0, 1\}$
  Restricciones explícitas: indican qué valores pueden tomar los componentes de la solución
  - $x_i = 0$ indica que el elemento i **no** está en la solución
  - $x_i = 1$ indica que el elemento i **sí** está en la solución
- La solución parcial debe cumplir que $\sum_{i=1}^{3} x_i \cdot dato_i \leq 20$
  Restricciones implícitas: indican qué tuplas pueden dar lugar a soluciones válidas
- La solución debe cumplir el siguiente *objetivo*: $\sum_{i=1}^{3} x_i \cdot dato_i = 20$
- Para este problema existe una única solución

$$[1, 0, 1]$$

Características generales. Ejemplo de *backtracking* (Cont.)

Existen dos formas de proceder:

1. Generar todas las combinaciones posibles y escoger aquellas que sean solución

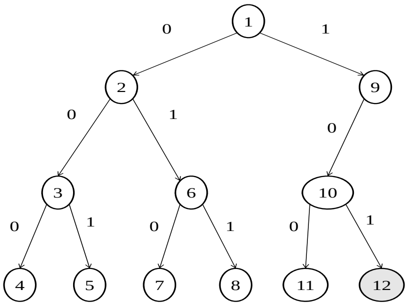


- Cada camino de la raíz a las hojas define una posible solución
- El árbol de expansión tiene 2^3 hojas (8 posibles soluciones)
- Se generan tuplas que no son soluciones → ineficiente

Características generales. Ejemplo de *backtracking* (Cont.)

Existen dos formas de proceder:

2. Utilizar la técnica de backtracking: a medida que se construye la tupla, se comprueba si ésta puede llegar a ser una solución al problema. En caso negativo, se ignora y se vuelve al estado anterior.

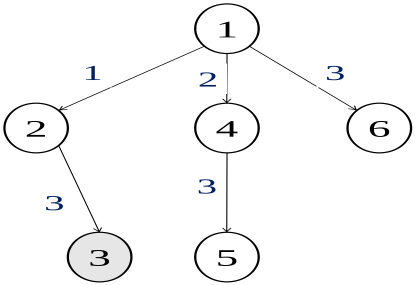


Ejemplo de *backtracking*. Otra representación de la solución

- Otra posible representación de la solución:
  Tupla o vector de a lo sumo 3 elementos **ordenados** con valores entre 1 y 3 (los índices de los elementos del conjunto anterior)
  - 1 indica que el elemento *1* (con valor 13) está en la solución
  - 2 indica que el elemento *2* (con valor 11) está en la solución
  - 3 indica que el elemento *3* (con valor 7) está en la solución
- Para este problema existe una única solución

$$[1, 3]$$

Ejemplo de *backtracking*. Otra representación de la solución



Esquemas algorítmicos. *Backtracking*

Esquema general de un algoritmo de *backtracking*

- La técnica de backtracking es un recorrido en profundidad (preorden) del árbol de expansión
- En cada momento el algoritmo se encontrará en un cierto nivel K
- En el nivel K se tiene una solución parcial $(x_1, \ldots x_k)$
- Se comprueba si se puede añadir un nuevo elemento x_{k+1} a la solución
  - En caso afirmativo, $(x_1, \ldots x_{k+1})$ es prometedor $\longrightarrow$ se genera la solución parcial $(x_1, \ldots x_{k+1})$ y se avanza al nivel $k+1$
  - En otro caso $\longrightarrow$ se prueban otros valores de x_k
- Si ya no existen más valores para x_k, se retrocede (se <u>vuelve atrás</u>–*backtrack*) al nivel anterior $k-1$
- El algoritmo continúa hasta que la solución parcial sea una solución completa del algoritmo, o hasta que no queden más posibilidades

Esquema general de *backtracking* recursivo

```
fun backtrackingRec(solucion[1..n], etapa)
   // valores es un vector [1..opciones]
   IniciarValores(valores, etapa)
   repetir
      nuevovalor ← SeleccionarNuevoValor(valores)
      si alcanzable(nuevovalor) entonces
         AnotarNuevoValor(solucion,nuevovalor)
         si SolucionIncompleta(solucion) entonces
            backtrackingRec(solucion, siguienteEtapa)
         si no si EsSolucion(solucion)  entonces
            escribir(solucion)
         fin si
         Desanotar(solucion)
      fin si
   hasta UltimoValor(valores)
fin fun
```

Funciones que aparecen en el esquema recursivo

- `IniciarValores(valores)`
  Genera todas las opciones del nivel donde se encuentra
- `SeleccionarNuevoValor(valores)`
  Considera un nuevo valor de los posibles
- `Alcanzable(nuevovalor)`
  Comprueba si la opción *nuevovalor* puede forma parte de la solucion
- `AnotarNuevoValor(solucion,nuevovalor)`
  Anota en *solucion* el valor *nuevovalor*
- `EsSolucion(solucion)`
  Indica si *solucion* es una solución para el problema
- `Desanotar(solucion)`
  Elimina la última anotación en el vector *solucion*
- `UltimoValor(valores)`
  Indica si ya no quedan más nodos por expandir

Esquema General de *backtracking* sin recursión

proc backtracking($solucion[1..n]$)
    $nivel \leftarrow 1$
    $fin \leftarrow false$
    repetir
        $solucion[nivel] \leftarrow generar(nivel, solucion)$
        si $esSolucion(solucion)$ **entonces**
            $fin \leftarrow \mathrm{true}$
        si no si $alcanzable(nivel, solucion)$ **entonces**
            $nivel \leftarrow nivel + 1$
        si no
            mientras not($hayMasHermanos(nivel, solucion)$) **hacer**
                $retroceder(nivel, solucion)$
            fin mientras
        fin si
    hasta fin = true
fin proc

Funciones que aparecen en el esquema no recursivo

- `generar(nivel, sol)`

  Dado un nivel, genera el siguiente hermano (o el primero). Devuelve el valor a añadir a la solución parcial actual. Ejemplo: $\textit{generar}(1, [0, -, -]) = 1$
- `esSolucion(sol)`

  Comprueba si la solución calculada hasta el momento es una solución válida para el problema. Ejemplo: $\textit{esSolucion}([0, 0, 1]) = \textit{false}$
- `alcanzable(nivel, sol)`

  Comprueba si a partir de la solución parcial actual es posible llegar a una solución válida. Ejemplos:

  $$\textit{alcanzable}(2, [0, 0, -]) = \textit{true} \qquad \textit{alcanzable}(2, [1, 1, -]) = \textit{false}$$
- `hayMasHermanos(nivel,sol))`

  Devuelve el valor true si el nodo actual tiene hermanos que aún no han sido generados. Ejemplo: $\textit{hayMasHermanos}(2, [1, 1, -]) = \textit{false}$
- `retroceder(nivel,sol))`

  Retrocede un nivel en el arbol de soluciones. Disminuye en 1 el valor del nivel y actualiza la solución actual.

Observaciones

- La representación de las soluciones determina la forma del árbol de expansión
  - Cantidad de descendientes de un nodo
  - Profundidad del árbol
  - Cantidad de nodos del árbol
- La representación de las soluciones determina, como consecuencia, la eficiencia del algoritmo ya que el tiempo de ejecución depende del número de nodos generados
- El árbol tendrá tantos niveles como valores tenga la secuencia solución
- En cada nodo se debe poder determinar:
  - Si es solución o posible solución del problema
  - Si tiene hermanos sin generar
  - Si a partir de este nodo se puede llegar a una solución

Esquemas algorítmicos. *Backtracking*

Estudio de complejidad

- En general se obtienen órdenes de complejidad exponencial y factorial
- El orden de complejidad depende del **número de nodos generados** y del tiempo requerido para cada nodo (que podemos considerar constante)
- Si la solución es de la forma $(x_1, \ldots, x_n)$, donde x_i admite m_i valores
- En el caso peor, se generarán todas las posibles combinaciones para cada x_i

Nivel 1	m_1 nodos
Nivel 2	$m_1 * m_2$ nodos
...	
Nivel n	$m_1 * m_2 * \ldots * m_n$ nodos

- Para el ejemplo planteado anteriormente, $m_i = 2$

  $$T(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2$$

  Tiempo exponencial
- Cada caso depende de cómo se realice la poda del árbol, y de la instancia del problema

Estudio de complejidad (cont.)

- Para el problema de calcular todas las permutaciones de $(1, 2, \ldots, n)$
- Representamos la solución como una tupla $\{x_1, x_2, \ldots x_n\}$
- *Restricciones explícitas*: $x_i \in \{i, \ldots, n\}$
- En el nivel 1, tenemos n posibilidades, en el nivel 2 $n - 1$

Nivel 1	n nodos
Nivel 2	$n * (n-1)$ nodos
...	
Nivel n	$n * (n-1) * \ldots * 1$ nodos

- Tiempo factorial

$$T(n) = n + n * (n-1) + \ldots + n! \in \mathcal{O}(n!)$$

Esquemas algorítmicos. *Backtracking*

Problema de la mochila 0-1

- Es el mismo problema visto en programación dinámica
- Se dispone de n objetos y una mochila de capacidad $C > 0$,
  - El peso del objeto i es $w_i > 0$
  - La inclusión del objeto i en la mochila produce un beneficio $b_i > 0$
- El objetivo consiste en llenar la mochila maximizando el valor de los objetos transportados sin sobrepasar la capacidad de la mochila
- Los objetos no son fraccionables

Problema de la mochila 0-1 (cont.)

- La solución se puede representar como una tupla $\{x_1, x_2, \ldots x_n\}$
- *Restricciones explícitas*: $x_i \in (0, 1)$
  Si $x_i = 0$ el objeto i no se introduce en la mochila
  Si $x_i = 1$ el objeto i se introduce en la mochila
- *Restricciones implícitas*: $\sum_{i=1}^{n} x_i \cdot w_i \leq C$
- El **objetivo** es maximizar la función $\sum_{i=1}^{n} x_i \cdot b_i$

Problema de la mochila 0-1 (cont.)

- Almacenaremos una **solución parcial** que se irá actualizando al encontrar una nueva solución con mayor beneficio
- Solo los nodos terminales del árbol de expansión pueden ser solución al problema
- La función `alcanzable` comprueba que los pesos acumulados hasta el momento no excedan la capacidad de la mochila. Esta función permite la poda de nodos

Problema de la mochila 0-1 (cont.)

```
// elem[1..n] es un vector de estructuras con dos campos: beneficio y peso
proc mochila(elem[1..n],solAct[1..n],sol[1..n],benActIni,ben,pesoActIni,etapa)
  desde obj ← 0 hasta 1 hacer
    solAct[etapa] ← obj
    benAct ← benActIni + obj*elem[etapa].beneficio
    pesoAct ← pesoActIni + obj*elem[etapa].peso
    si (pesoAct ≤ C) entonces
      si  etapa = n entonces
        si benAct > ben  entonces
          sol ← solAct // Se asigna el vector completo
          ben ← benAct
        fin si
      si no
        mochila(elem,solAct,sol,benAct,ben,pesoAct,etapa+1)
      fin si
    fin si
  fin desde
fin proc
```

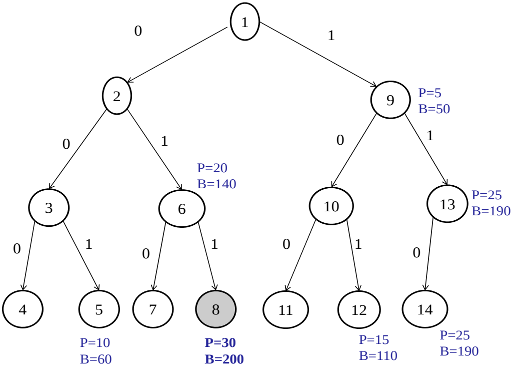
Problema de la mochila [0,1] (cont.)

- El procedimiento llamador puede ser el siguiente:

  proc llamador_mochila(elem[1..n],sol[1..n])
    crear solAct[1..n]
    mochila(elem,solAct,sol,0,$-\infty$,0,1)
  fin proc

- Ejemplo: Con una mochila de capacidad C=30 y los siguientes objetos, el árbol resultante es:

objeto	A	C	B
peso	5	20	10
valor	50	140	60



Esquemas algorítmicos. *Backtracking*

Asignación de tareas

(Basado en [GV00], p. 229.)

- Disponemos de n empleados y n tareas a realizar
- Mediante una tabla M de tamaño $n \times n$, representamos con $M[i, j]$ el coste de realizar la tarea j por el empleado i, para $i, j = 1, \ldots, n$
- El problema consiste en asignar a cada operario i una tarea j de forma que se minimice el coste total.

- La solución se puede representar como una tupla $Sol = \langle x_1, x_2, \ldots x_n \rangle$
- *Restricciones explícitas*: $x_i \in \{1, \ldots n\}$
  x_i es la tarea asignada al i-ésimo empleado
- *Restricciones implícitas*: $x_i \neq x_j, \forall i \neq j$
- El objetivo es minimizar la función $\sum_{i=1}^{n} M[i, x_i]$
- Por cada solución que encuentre el algoritmo, se anotará su coste y se comparará con el coste de la mejor solución encontrada hasta el momento

Asignación de tareas (Cont.)

```
proc Tareas(M[1..n,1..n],XAct[1..n],mejorX[1...n],costeAcIni,coste,etapa)
   XAct[etapa] ← 0
   repetir
      XAct[etapa] ← XAct[etapa] + 1
      si TareaNoAsignada(XAct,etapa) entonces
         costeAc ← costeAcIni + M[etapa,XAct[etapa]]
         si (costeAc ≤ coste) entonces
            si  etapa < n entonces
               tareas(M,XAct,mejorX,costeAc,coste,etapa+1)
            si no
               mejorX ← XAct
               coste ← costeAc
            fin si
         fin si
      fin si
   hasta XAct[etapa]=n
fin proc
```

Asignación de tareas (Cont.)

- El código de TareaNoAsignada es el siguiente:

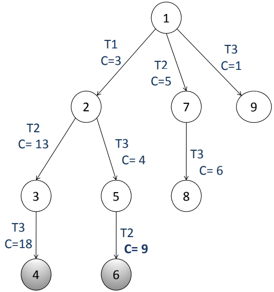
  fun TareaNoAsignada(Asignadas, fin)
      desde i $\leftarrow$ 1 **hasta** fin-1 **hacer**
          si Asignadas[i] = Asignadas[fin] **entonces**
              devolver falso
          fin si
      fin desde
      devolver cierto
  fin fun

Asignación de tareas (Cont.)

- El procedimiento llamador es:

proc llamador_tareas(M[1..n,1..n],X[1..n],C)
  crear XAct[1..n]
  C $\leftarrow \infty$
  Tareas(M,XAct,X,0,C,1)
fin proc

- Este algoritmo realiza podas en el árbol de expansión eliminando aquellos nodos que no van a llevar a la solución óptima



$$\begin{bmatrix} 3 & 5 & 1 \\ 10 & 10 & 1 \\ 8 & 5 & 5 \end{bmatrix}$$

Asignación de tareas (Cont.)

$$\begin{bmatrix} 3 & 5 & 1 \\ 10 & 10 & 1 \\ 8 & 5 & 5 \end{bmatrix}$$

- La secuencia de llamadas para la matriz de Tareas
  tareas([0,0,0], [0,0,0], 0, ∞, 1)
  - tareas([1,0,0], [0,0,0], 3, ∞, 2)
    - tareas([1,2,0], [0,0,0], 13, ∞, 3)
    - tareas([1,3,0], [1,2,3], 4, 18, 3)
  - tareas([2,0,0], [1,3,2], 5, 9, 2)
    - tareas([2,3,0], [1,3,2], 6, 9, 3)
  - tareas([3,0,0], [1,3,2], 1, 9, 2)

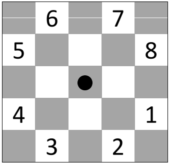
Esquemas algorítmicos. *Backtracking*

El caballo de ajedrez

- Consideremos un caballo de ajedrez colocado en una posición, X, Y de un tablero de dimensiones $n \times n$. Se trata de encontrar una sucesión de movimientos válidos de un caballo de ajedrez de forma que éste pueda visitar todas y cada una de las casillas del tablero sin repetir ninguna
- El algoritmo voraz no encontraba solución para un tablero con $n = 5$ y partiendo de la posición inicial $(x, y) = (5, 3)$

El caballo de ajedrez (Cont.)

- La solución la representamos como una matriz de enteros de dos dimensiones $Sol[1..n, 1..n]$
  - Inicialmente $Sol[i,j] = 0 \rightarrow$ la casilla (i,j) no ha sido visitada
  - $Sol[i,j] = k$ significa que el caballo ha pasado por la casilla (i,j) en la etapa k
- Los movimientos que puede hacer el caballo se representan mediante vectores constantes



dx = (	2	1	-1	-2	-2	-1	1	2	)
dy = (	1	2	2	1	-1	-2	-2	-1	)

El caballo de ajedrez (Cont.)

- Esquema de alto nivel del algoritmo:

proc saltoCaballo
  repetir
    seleccionar un movimiento m válido del caballo (solo hay 8)
    si (m está en el tablero) $\wedge$ (la casilla no está repetida) **entonces**
      anotar movimiento
      si $numeroSaltos < n \times n$ **entonces**
        seguir moviendo
        si no alcanzado solución **entonces**
          deshacer anotación anterior
        fin si
      fin si
    fin si
  hasta (visitado las $n \times n$ casillas) or (agotados los 8 movimientos)
fin proc

El caballo de ajedrez (Cont.)

- Para definir el algoritmo detallado, tenemos las siguientes consideraciones
  - Los vectores *dx* y *dy* son constantes globales.
  - La solución se representa con la matriz *sol*
  - Las variables *posXini* y *posYini* representan la posición de partida del caballo en el tablero
  - La variable *exito* devuelve true si después de un movimiento se puede encontrar una solución
  - El programa llamador podría ser algo como

fun llamador_caballo(sol[1..n,1..n])
  inicializar(sol, dx, dy)
  posXini ← 1
  posYini ← 1
  $sol[1, 1] \leftarrow 1$
  exito ← falso
  saltoCaballo(sol, 2, posXini, posYini, exito)
  devolver exito
fin fun

El caballo de ajedrez (Cont.)

proc $saltoCaballo(sol[1..n, 1..n], etapa, posX, posY, exito)$
  k $\leftarrow$ 0 // *indica el número de movimientos del caballo(8)*
  repetir
    k $\leftarrow k + 1$
    $Nx \leftarrow posX + dx[k]$
    $Ny \leftarrow posY + dy[k]$
    si $((1 \leq Nx \leq n) \wedge (1 \leq Ny \leq n))$ **entonces**
      si $sol[Nx, Ny] = 0$ **entonces**
        $sol[Nx, Ny] \leftarrow etapa$
        si $etapa < n \times n$ **entonces**
          $saltoCaballo(sol, etapa + 1, Nx, Ny, exito)$
          si ¬exito **entonces** $sol[Nx, Ny] \leftarrow 0$
        si no
          exito $\leftarrow$ cierto
        fin si
      fin si
    fin si
  hasta (exito $\vee$ k =8)
fin proc

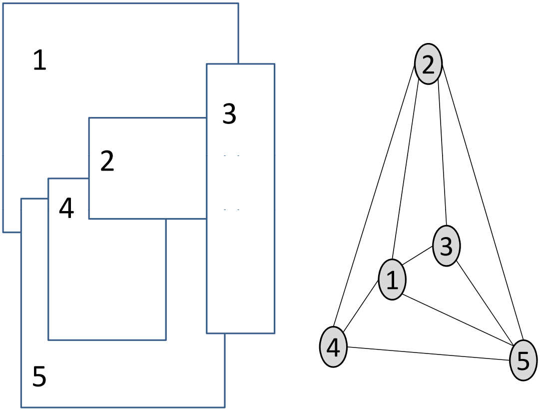
Esquemas algorítmicos. *Backtracking*

Coloreado de mapas

[GV00], p. 246.

- Dado un grafo conexo no dirigido y un numero $m > 0$, llamamos colorear el grafo a asignar un numero i ($1 \leq i \leq m$) a cada vértice, de forma que dos vertices adyacentes nunca tengan asignados numeros iguales
- Supongamos que el grafo tiene n vértices
- Deseamos implementar un algoritmo que coloree un grafo dado
- Este problema proviene de un problema clásico de coloreado de mapas en el plano. Dado un mapa, ¿Cuál es el mínimo número de colores necesario para colorear sus regiones de forma que no haya dos regiones adyacentes de igual color?
  - Cada región se corresponde con un nodo
  - Si dos regiones son adyacentes, sus nodos se conectan con un arco
  - Así, siempre se obtiene un grafo donde sus arcos nunca se cruzan (grafo planar)

Coloreado de mapas (Cont.)



- Se ha demostrado que 4 colores siempre son suficientes para colorear un mapa.

Coloreado de mapas (Cont.)

- La solución se puede representar como una tupla $\langle x_1, x_2, \ldots x_n \rangle$ donde x_i es el color del vértice i
- Si se dispone de m colores, en la etapa k el algoritmo asigna un color $c \in \{1, \ldots m\}$ al vértice k.
- *Restricciones explícitas*: $x_i \in \{1, \ldots, m\}$
- *Restricciones implícitas*: Vértices adyacentes no pueden ser del mismo color
- El objetivo es buscar una forma de colorear el grafo utilizando solo m colores

Coloreado de mapas (Cont.)

```
// grafo es una matriz de tipo booleano
// solucion es un vector de enteros (contiene numeros de color, 1..m)
fun colorearMapa(grafo[1..n, 1..n], solucion[1..n], m, etapa)
   solucion[etapa] ← 0 ; exito ← falso
   repetir
      solucion[etapa] ← solucion[etapa] + 1
      si aceptable(grafo, solucion, etapa) entonces
         si  etapa < n entonces
            exito ← colorearMapa(grafo, solucion, m, etapa + 1)
         si no
            exito ← cierto
         fin si
      fin si
   hasta exito ∨ solucion[etapa]= m
   devolver exito
fin fun
```

Coloreado de mapas (Cont.)

- El programa llamador podría ser el siguiente

```
fun llamador_mapa(grafo[1..n,1..n],m,solucion[1..n])
   desde i ← 1 hasta n hacer
      solucion[i] ← 0
   fin desde
   devolver colorearMapa(grafo,solucion,m,1)
fin fun
```

Coloreado de mapas (Cont.)

- La función `aceptable` comprueba que dos vértices adyacentes no tengan el mismo color

```
fun aceptable(grafo[1..n,1..n],sol[1..n],etapa)
   desde j ← 1 hasta etapa-1 hacer
      si grafo[etapa,j] ∧ sol[etapa]=sol[j] entonces
         devolver false
      fin si
   fin desde
   devolver true
fin fun
```

Coloreado de mapas (Cont.)

- Supongamos ahora que lo que deseamos es obtener todas las formas distintas de colorear un grafo

```
proc colorearMapaTodasSoluciones(grafo[1..n,1..n],solucion[1..n],m,etapa)

  solucion[etapa] ← 0
  repetir
    solucion[etapa] ← solucion[etapa]+1
    si aceptable(grafo,solucion,etapa) entonces
      si  etapa < n entonces
        colorearMapaTodasSoluciones(grafo,solucion,m,etapa+1)
      si no
        comunicar(solucion)
      fin si
    fin si
  hasta solucion[etapa]= m // hasta el ultimo color
fin proc
```

Coloreado de mapas (Cont.)

- Supongamos ahora que lo que deseamos es colorear un grafo con el mínimo número de colores

```
proc colorearMapaOptimizado(grafo[1..n,1..n],solucion[1..n],m,etapa)
  solucion[etapa] ← 0
  repetir
    solucion[etapa] ← solucion[etapa]+1
    si aceptable(grafo,solucion,etapa) entonces
      si  etapa < n entonces
        colorearMapaOptimizado(grafo,solucion,m,etapa+1)
      si no
        numcol ← numColoresEn(solucion,m)
        si minimo > numcol entonces mejorSol ← solucion ; minimo ← numcol
      fin si
    fin si
  hasta solucion[etapa]= m // hasta el ultimo color
fin proc
```

- Las variables mejorSol y minimo son globales. ¿Cómo sería el programa llamador en este caso? ¿Cómo sería sin variables globales?

Coloreado de mapas (Cont.)

- La función `numColoresEn` calcula el número de colores diferentes utilizados en una solución

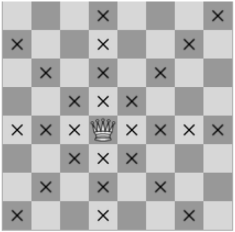
```
// m es el numero de colores
fun numColoresEn(sol,m)
   contador ← 0
   desde j ← 1 hasta m hacer
      i ← 1
      continuar ← true
      mientras (i ≤ n) and (continuar) hacer
         si solucion[i]= j  entonces
            contador ← contador+1
            continuar ← false
         fin si
         i ← i+1
      fin mientras
   fin desde
   devolver contador
fin fun
```

Esquemas algorítmicos. *Backtracking*

Las *n* reinas

[GV00], p. 212.

- Es otro problema clásico de computación
- Consiste en encontrar la forma de disponer n reinas en un tablero de ajedrez $n \times n$ de forma que no se amenacen entre ellas
- Los movimientos que puede realizar una reina de ajedrez son:



Las n reinas (cont.)

- En este caso, por los movimientos que se pueden realizar, no es posible que una solución del problema tenga dos reinas en la misma fila o columna
- Por ello, si debemos colocar n reinas en un tablero $n \times n$, cada fila y cada columna tendrán una sola reina
- En lugar de representar la solución mediante una matriz $n \times n$, podemos utilizar un vector de tamaño n, X[1..n], en el que X[i] representa la columna en la que está situada la reina i-ésima (que estará en la fila i-ésima)
- Las restricciones de este problema son las siguientes:
  - *restricciones explícitas*: los elementos del vector solución solamente pueden tener valores entre 1 y n (columnas). La restricción sobre las filas está determinada por la propia estructura de la solución (al ser un vector X[1..n])
  - *restricciones implícitas*: no puede haber dos reinas en la misma columna ni en la misma diagonal.

Las n reinas (cont.)

```
fun reinas(k,sol[1..n])
  exito ← falso
  sol[k] ← 0
  repetir
    sol[k] ← sol[k] + 1
    si valido(sol,k) entonces
      si k ≠ n entonces
        exito ← reinas(sol,k+1)
      si no
        exito ← cierto
      fin si
    fin si
  hasta (sol[k] = n) ∨ exito
  devolver exito
fin fun
```

```
fun valido(sol[1..n],k)
  desde i ← 1 hasta k-1 hacer
    si sol[i]=sol[k] ∨ |sol[i]-sol[k]| = |i-k|
    entonces
      devolver falso
    fin si
  fin desde
  devolver cierto
fin fun
```

- Llamada inicial: reinas(1,sol)

Esquemas algorítmicos. *Backtracking*

Trenes

(basado en [GV00], p. 242).

- Sea una región con *n* ciudades. Se pretende ir de una ciudad a otra de la misma región
- Para ello disponemos de los **horarios** de todos los trenes que comunican las ciudades de esa región
- Se conoce también la **duración de cada viaje**
- Se supone que, a lo sumo, hay un tren para cada par de ciudades
- El objetivo es encontrar un camino para viajar entre dos ciudades dadas, de forma que se minimice el tiempo empleado.

Trenes (Cont.)

- Supongamos que las ciudades están numeradas de 1 a n.
- Los horarios y duraciones de viaje de los trenes se representan en una matriz de horarios $H[1..n, 1..n]$ en la que cada elemento tiene dos atributos, `duracion` y `hora`.
- Los elementos de esta matriz tienen ∞ en ambos atributos si las ciudades i y j no están comunicadas.
- Representamos la solución del problema mediante una tupla $Sol = \langle x_1, x_2, \ldots x_k \rangle$
- La tupla solución solo tiene valores hasta el nivel k, siendo k el número de ciudades por las que tenemos que pasar $(k \leq n)$
- Tiene que cumplirse
  - x_1 es la ciudad de origen
  - x_k es la ciudad de destino
  - x_i está comunicada con la ciudad x_{i+1} para $1 \leq i \leq (k-1)$

Trenes (Cont.)

```
// H es una matriz de estructuras con dos atributos: duracion y hora (de salida).
proc trenes(H[1..n,1..n],solAct[1..n],sol[1..n],dest,etapa,hora,durAcIni,durMinimo)
  j ← 0 // j representa la ciudad
  repetir
    j ← j + 1 ; solAct[etapa] ← j
    si aceptable(H,solAct,hora,etapa) entonces
      duracionEspera ← H[solAct[etapa-1], j].hora - hora
      duracionAc ← durAcIni + H[solAct[etapa-1], j].duracion + duracionEspera
      si j = dest entonces
        si duracionAc ≤ durMinimo entonces
          sol ← solAct
          durMinimo ← duracionAc
        fin si
      si no si etapa < n entonces
        horallegada ← H[solAct[etapa-1], j].hora+ H[solAct[etapa-1], j].duracion
        trenes(H,solAct,sol,dest,etapa+1, horallegada, duracionAc,durMinimo)
      fin si
    fin si
  hasta j = n
fin proc
```

Trenes (Cont.)

- Los argumentos `sol` y `durMinimo` contienen la solución óptima: el recorrido óptimo y el tiempo total del trayecto. Se podrían definir como variables globales
- El argumento `durAcIni` en la etapa k almacena el tiempo empleado en recorrer el camino entre las ciudades origen y `sol[k-1]`: la suma de los tiempos de espera entre los diferentes trenes y las duraciones de los trayectos hasta $k - 1$.
- Se supone en este problema que todos los recorridos tienen lugar en el mismo día. Si no fuera así, debería tenerse en cuenta que en algunos casos es necesario esperar al día siguiente para continuar el recorrido

Trenes (Cont.)

- El programa llamador podría ser el siguiente

```
proc llamador_trenes(H[1..n,1..n],origen,dest,sol[1..n],duracion)
  crear solAct[1..n]
  duracion ←∞
  hora ← "00:00"
  solAct[1] ←origen
  trenes(H,solAct,sol,dest,2,hora, 0,duracion)
fin proc
```

- Si el problema no tiene solución, al final de la ejecución la variable `duracion` contendrá ∞.

Trenes (Cont.)

- La función `aceptable` comprueba
  - que no se pase dos veces por la misma ciudad
  - que toda ciudad debe estar comunicada con la anterior
  - que la hora de llegada a la ciudad de la etapa anterior sea menor que la hora de salida del tren a la ciudad j

fun aceptable(H,sol,hora,etapa)
  desde i $\leftarrow$ 1 **hasta** etapa-1 **hacer**
    si sol[i]= sol[etapa] **entonces**
      devolver falso
    fin si
  fin desde
  devolver (H[sol[etapa-1], sol[etapa]].hora $< \infty$)
        $\wedge$ (hora $<$H[sol[etapa-1], sol[etapa]].hora)
fin fun

Esquemas algorítmicos. *Backtracking*

Franqueo de postales

- Consideremos una postal en la que se pueden poner un máximo de M sellos
- Se dispone de un número limitado de unidades de cada tipo de sellos
- Para poder enviar la postal, se necesita franquearla con una tarifa mínima
- Además, se supone que el orden de los sellos en la postal debe ser tal que su valor sea decreciente
- Determinar la forma de franquear la postal de forma que el coste sea mínimo

Franqueo de postales (Cont.)

- La información de los tipos de sellos se proporciona mediante un vector S[1..N] con dos atributos:
  - `valor`: valor de cada tipo de sello
  - `cantidad`: número de unidades de cada tipo
- El vector S debe contener los tipos de sello ordenados en orden decreciente de valor
- Representamos la solución del problema mediante una tupla $Sol = \langle x_1, x_2, \ldots x_k \rangle$, $k \leq m$, donde x_i indica el tipo de sello en la posición i de la postal
- Debe cumplirse la restricción explícita $x_i \in \{1..N\}$
- y la restricción implícita siguiente: el valor del sello x_i debe ser mayor o igual que el valor del sello x_{i+1} para $1 < i < M - 1$ y $x_i \neq 0$

Franqueo de postales (Cont.)

```
// S es un vector de estructuras con dos atributos: cantidad y valor.
proc franqueo(S[1..N],tarifa,etapa,tipoSello,solAct[1..M],costeAcIni,sol[1..M],coste)
  si etapa ≤ M entonces
    desde j ← tipoSello hasta N hacer
      si (S[j].cantidad>0) entonces
        solAct[etapa] ← j
        costeAc ← costeAcIni + S[j].valor
        S[j].cantidad ← S[j].cantidad - 1
        si (costeAc ≥ tarifa) entonces
          si (costeAc < coste) entonces  coste ← costeAc ; sol← solAct
        si no
          franqueo(S,tarifa,etapa+1,j,solAct,costeAc,sol,coste)
        fin si
        S[j].cantidad ← S[j].cantidad + 1 // Deshacer etapa
      fin si
    fin desde
  fin si
fin proc
```

Franqueo de postales (Cont.)

- El programa llamador podría ser

  proc llamador_franqueo(S,tarifa,sol[1..M],coste)
      crear solAct[1..M]
      desde i $\leftarrow$ 1 **hasta** n **hacer**
          solAct[i] $\leftarrow$ 0 // inicialmente no hay sellos
      fin desde
      coste $\leftarrow \infty$
      franqueo(S,tarifa,1,1,solAct,0,sol,coste)
  fin proc

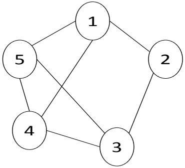
- Si el problema no tiene solución, al final de la ejecución la variable `coste` contendrá ∞.

Esquemas algorítmicos. *Backtracking*

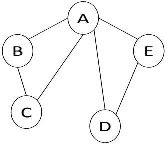
1. Características generales
2. Esquema general de un algoritmo de *backtracking*
3. Estudio de complejidad
4. Problema de la mochila 0-1
5. Asignación de tareas
6. El caballo de ajedrez
7. Coloreado de mapas
8. Las *n* reinas
9. Trenes
10. Franqueo de Postales
11. Ciclos Hamiltonianos

Ciclos Hamiltonianos

- Sea g un grafo no dirigido con n vértices
- Llamamos ciclo hamiltoniano a un camino que visita una sola vez todos los vértices y vuelve al vértice inicial



1 – 2 – 3 – 4 – 5 – 1
1 – 2 – 3 – 5 – 4 – 1
1 – 5 – 4 – 3 – 2 – 1
1 – 4 – 5 – 3 – 2 – 1



Ningún hamiltoniano

- El objetivo es encontrar todos los ciclos hamiltonianos de un grafo
- Existen algoritmos voraces muy eficientes para este tipo de problemas, pero pueden no proporcionar la solución óptima

Ciclos Hamiltonianos (Cont.)

- Utilizaremos la representación del grafo mediante una matriz de adyacencia $G[1..n, 1..n]$ binaria
- Representamos la solución del problema mediante una tupla $sol = \langle x_1, x_2, \ldots x_n \rangle$ donde x_i indica el vértice visitado en el $i-$ésimo lugar en el ciclo
- Para evitar repetir varias veces el mismo ciclo, partimos siempre del vértice 1 (por ejemplo), es decir, $x_1 = 1$
- Han de cumplirse las siguientes restricciones
  - $x_i \neq x_j$ para $i, j \in \{1, \ldots, n\}, i \neq j$
  - Los vértices x_i y x_{i+1} han de estar conectados para $i \in \{1, \ldots, n-1\}$, es decir, $G[x_i, x_{i+1}] = cierto$
  - El vértice x_n ha de estar conectado con el vértice x_1 y con el x_{n-1}

Ciclos Hamiltonianos (Cont.)

```
proc hamiltoniano(G[1..n,1..n],etapa,sol[1..n])
  repetir
    siguienteVertice(G,sol,etapa) // asigna a sol[etapa] el siguiente vertice
    si sol[etapa] ≠ 0 entonces
      si etapa = n entonces
        comunicar(sol)
      si no
        hamiltoniano(G,etapa+1,sol)
      fin si
    fin si
  hasta sol[etapa] = 0
fin proc
```

Ciclos Hamiltonianos (Cont.)

- `siguienteVertice` busca un vértice v válido para la etapa k partiendo de una solución parcial de $k-1$ vértices
  - v está conectado con el vértice de la etapa anterior: $G[sol[k-1], v] = cierto$
  - v no aparece en la solución parcial de k-1 vértices: $sol[i] \neq v$ para $1 \leq i \leq k-1$
  - Si estamos en la última etapa $(k = n)$, debemos exigir que v esté conectado con el primer vértice.
- `siguienteVertice` puede ser llamado varias veces para una misma etapa, seleccionando en cada llamada el siguiente vértice que cumple las restricciones
- Si no encontramos ningún vértice (más) que verifique las restricciones, entonces `siguienteVertice` asigna el valor 0 a sol[k]

Ciclos Hamiltonianos (Cont.)

```
proc siguienteVertice(G[1..n,1..n],sol[1..n], etapa)
  repetir
    sol[etapa] ← (sol[etapa]+1) mod (n+1) //asigna siguiente valor, entre 1 y n
    si sol[etapa] ≠ 0 entonces
      encontrado ← falso
      si G[sol[etapa-1],sol[etapa]] entonces
        j ← 1, encontrado ← cierto
        mientras j ≤ etapa-1 ∧ encontrado hacer
          si sol[j] = sol[etapa] entonces  encontrado ← falso
          si no j ← j+1
        fin mientras
        si encontrado ∧ etapa=n ∧ ¬G[sol[n],1] entonces
          encontrado ← falso
        fin si
      fin si
    fin si
  hasta sol[etapa] = 0 ∨ encontrado
fin proc
```

Ciclos Hamiltonianos (Cont.)

- El programa llamador sería

proc llamador_hamiltoniano(G[1..n,1..n])
    crear sol[1..n]
    desde i ← 2 **hasta** n **hacer**
        sol[i] ← 0
    fin desde
    sol[1] ← 1
    hamiltoniano(G,2,sol)
fin proc