UNIVERSIDAD COMPLUTENSE MADRID

Tema 3: Esquema algorítmico de vuelta atrás

Miguel Gómez-Zamalloa y Enrique Martín
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Facultad de Informática

Temario

- 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 ✔

Contenidos

Introducción

Ejemplo 1: Todas las secuencias posibles

- Nos piden imprimir todas las posibles secuencias de n caracteres usando las primeras m letras del alfabeto.
- Podemos escribir el siguiente algoritmo recursivo:

```
void palabrasRec(vector<char>& soluc, int k, int n, int m){
    for (char c = 'a'; c < 'a' + m; c++){
        soluc[k] = c;
        if (k == n - 1) cout << soluc << endl;
        else palabrasRec(soluc, k + 1, n, m);
    }
}

void palabras(){
    int n, m;
    cin >> n >> m;
    vector<char> soluc(n);
    palabrasRec(soluc,0,n,m);
}
```

Ejemplo 2: Todas las secuencias posibles sin repeticiones

- Nos piden ahora imprimir todas las posibles secuencias de n caracteres usando las primeras m letras del alfabeto pero sin repetir letras.
- Una primera estrategia sería usar el algoritmo anterior y simplemente chequear antes de imprimir que la secuencia no tiene repeticiones:

```
void palabrasRec(vector<char>& soluc, int k, int n, int m){
    for (char c = 'a'; c < 'a' + m; c++){
        soluc[k] = c;
        if (k == n - 1){
            if (todasDistintas(soluc)) cout << soluc << endl;
        } else palabrasRec(soluc, k + 1, n, m);
    }
}
```

- No parece una estrategia muy inteligente... Se están generando muchas soluciones potenciales inútiles.
- Ver árbol de exploración. Tiene $VR_m^n = m^n$ nodos terminales y solo $V_m^n = m!/(m-n)!$ son soluciones. Además, en cada uno llamamos a todasDistintas, que en el mejor de los casos sería lineal (usando un *set*).

Ejemplo 3: Todos los subconjuntos posibles

- ¿Y si nos piden las secuencias sin repeticiones y sin importar el orden? Es decir, por ej., *ab* sería la misma que *ba*.
- Podemos seguir con la misma estrategia de generar todas y luego descartar las que son permutaciones, por ejemplo obligando a que estén ordenadas.
- En este caso la problemática es análoga pero mucho más evidente.
- Ver árbol de exploración. Tiene $VR_m^n = m^n$ nodos terminales y en este caso solo $C_m^n = \binom{m}{n} = m!/(n! * (m-n)!)$ son soluciones.

Ejemplo 2: Una estrategia más inteligente

- Sería mucho más inteligente (y además sencillo de implementar) descartar las soluciones incorrectas cuanto antes.
- En este caso, tan pronto como pongamos una letra repetida (por ej. [a,a,...]) cortamos la exploración.
- Fácil: Al probar con una letra, comprobamos que esa letra no haya aparecido antes.

```
void palabrasRec(vector<char>& soluc, int k, int n, int m){
    for (char c = 'a'; c < 'a'+ m; c++){
        soluc[k] = c;
        if (!count(soluc.begin(),soluc.begin()+k,c)){
            if (k == n - 1) cout << soluc << endl;
            else
                palabrasRec(soluc, k + 1, n, m);
        }
    }
}
```

Ejemplo 2: Una estrategia aún más inteligente

- Podemos evitar la búsqueda lineal (que lleva a cabo el count) llevando una tabla de apariciones (vector de booleanos de tamaño m):

```
void palabrasRec(vector<char>& soluc, int k, int n, int m,
                 vector<bool>& usadas){
  for (char c = 'a'; c < 'a'+ m; c++){
    soluc[k] = c;
    if (!usadas[c-'a']) {
      if (k == n - 1) {
        cout << soluc << endl;
      } else {
        usadas[c-'a'] = true;
        palabrasRec(soluc, k + 1, n, m, usadas);
        usadas[c-'a'] = false; // Ojo, es necesario esto!
      }
    }
  }
}
```

- Habría que inicializarlo así: vector<**bool**> usadas(m,**false**);

Ejemplo 3: Una estrategia más inteligente

- En el caso del Ej. 3 (subconjuntos) podemos descartar subsoluciones no ordenadas así:

```
void palabrasRec(vector<char>& soluc, int k, int n, int m,
                 vector<bool>& usadas){
  for (char c = 'a'; c < 'a'+ m; c++){
    soluc[k] = c;
    if ((k == 0 || c > soluc[k-1]) && !usadas[c-'a']) {
      if (k == n - 1) {
        cout << soluc << endl;
      } else {
        usadas[c-'a'] = true;
        palabrasRec(soluc, k + 1, n, m, usadas);
        usadas[c-'a'] = false; // Ojo, es necesario esto!
      }
    }
  }
}
```

- Observa que ya no sería necesario el vector usadas.

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_k es la elección tomada en la etapa k-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, pudiendo descartar cuanto antes soluciones no válidas.

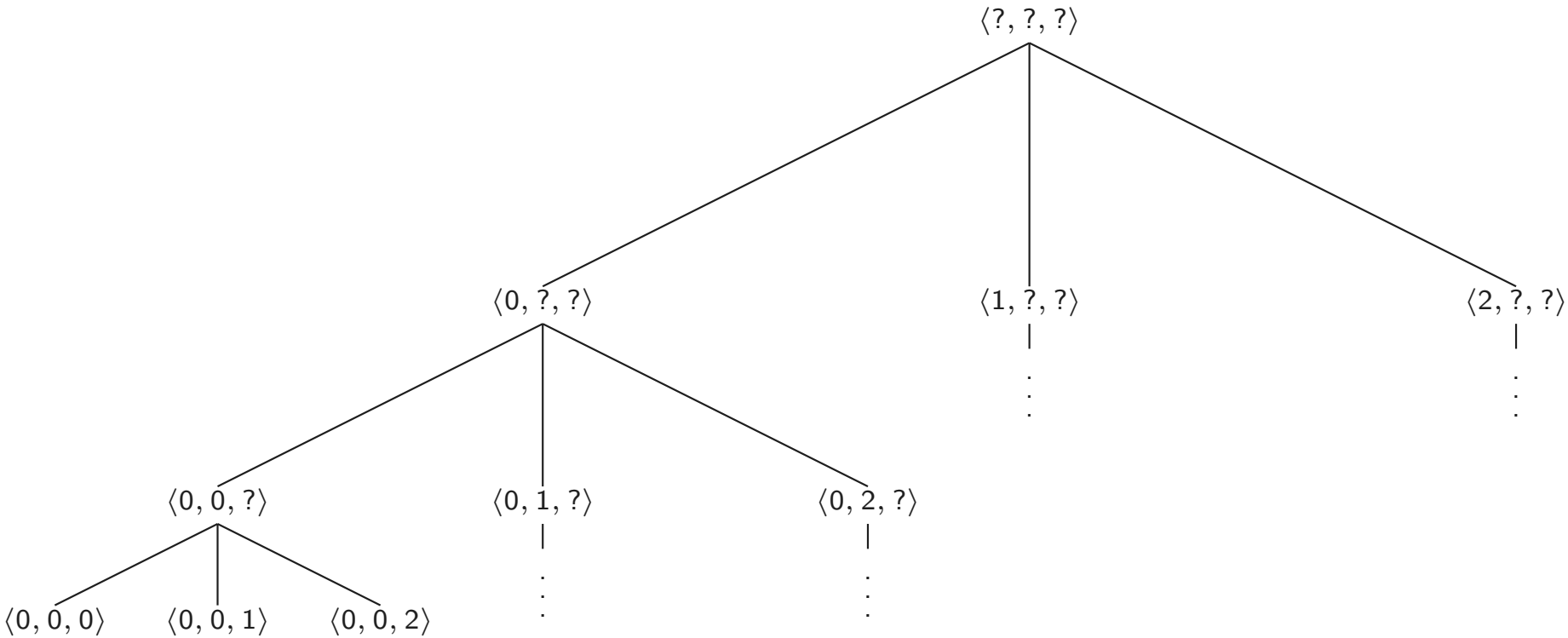
Precisamente lo que hemos ejemplificado en el problema anterior!

Vuelta atrás (o *backtracking*)

- Para recorrer todo el espacio de soluciones comenzaremos con una solución vacía $\langle ?, \ldots, ? \rangle$.
- Para cada posible valor $v_0 \ldots v_m$ de la primera elección x_0 generamos una solución parcial: $\langle v_0, ?, \ldots, ? \rangle$, $\langle v_1, ?, \ldots, ? \rangle$, $\ldots$, $\langle v_m, ?, \ldots, ? \rangle$
- 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

```
void backtracking(vector<T>& sol, int n, int k, ...) {
  for (T c : candidatos(k)) {
    sol[k] = c;
    if (esValida(sol, k, ...)) {
      if (esSolucion(sol, k, n))
        tratarSolucion(sol);
      else if (k < n-1)
        backtracking(sol, n, k + 1, ...)
    }
  }
}
```

- *sol* es una solución parcial con elecciones **correctas** hasta la posición *k* **no incluida**. Inicialmente k=0.
- Para cada posible valor *c* para la posición *k* comprobamos si puede formar una solución potencial correcta. En tal caso:
  - Si ya es una solución completa la procesamos (guardamos, mostramos por pantalla, etc.).
  - Si no realizamos una llamada recursiva para seguir rellenando desde la posición $k + 1$.

Esquema de vuelta atrás con marcadores

- Habitualmente, para poder implementar esValida de manera eficiente (por ej. evitando recorrer la solución) se usan *marcadores*.

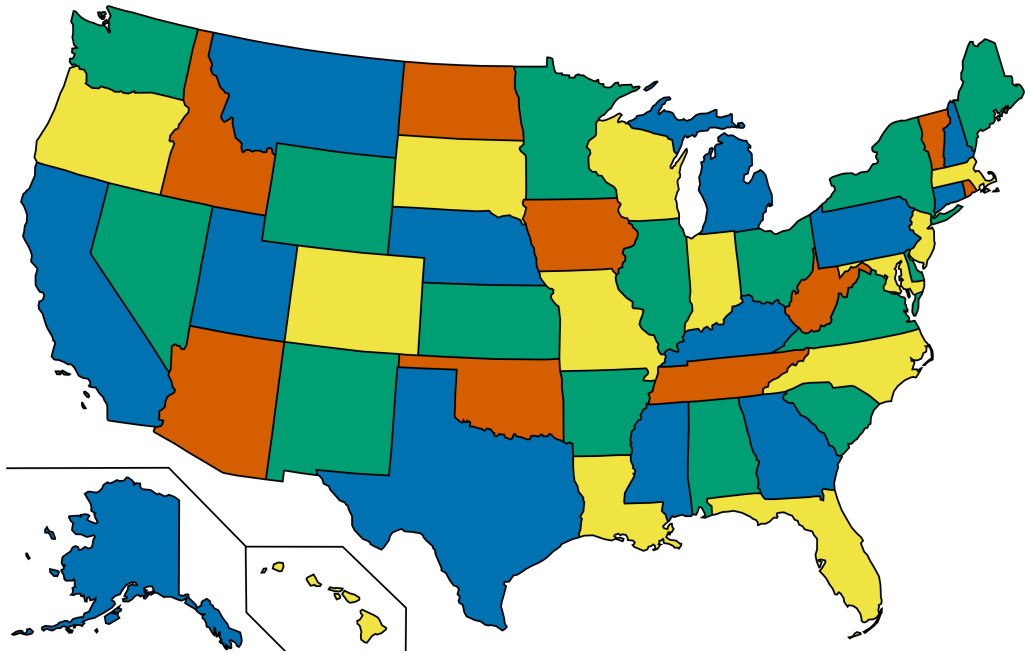
```
void backtracking(vector<T>& sol, int n, int k, Marcador& m) {
  for (T c : candidatos(k)) {
    sol[k] = c;
    if (esValida(sol, k, ...)) {
      if (esSolucion(sol, k, n))
        tratarSolucion(sol);
      else if (k < n-1) {
        marcar(m,...);
        backtracking(sol, n, k + 1, ...);
        desmarcar(m,...);
      }
    }
  }
}
```

- En el ejemplo de las palabras, el vector usadas sería un marcador. Permite evitar la búsqueda para chequear repeticiones.
- Al ser una estructura por referencia, al bajar se marca y al subir se desmarca.

Coloreado de un mapa

Coloreado de un mapa

Dado un mapa con n países y un número $m > 0$ se pide encontrar las formas de colorear el mapa utilizando un máximo de m colores, de manera que ningún par de países fronterizos tenga el mismo color.



- El mapa se puede representar como un grafo: los nodos son los países y la presencia de una arista entre dos nodos indica que tienen frontera.

Coloreado de un mapa

- El problema encaja perfectamente con el esquema de backtracking:

```
void coloreado(vector<int>& sol, int n, int k, int m,
               const Mapa& mapa) {
  for (int color = 0; color < m; ++color) {
    sol[k] = color;
    if (esValida(sol,k,mapa)) {
      if (k == n - 1) cout << soluc;
      else coloreado(sol, n, k+1, m, mapa);
    }
  }
}
```

- En este caso (como es habitual) no es necesario el **if** (k < n−1).
- El grafo se puede representar mediante una matriz de adyacencia:

```
using Mapa = vector<vector<bool>>;
```

de manera que mapa[i][j] == **true** indica que los países *i* y *j* tienen frontera.

Coloreado de un mapa

- La función principal quedaría así:

```
int mainColoreado(){
  cin >> n >> m;
  Mapa mapa(n, vector<bool>(n, false));
  //Ej. para ilustrar que los nodos 0 y 1 son adyacentes
  mapa[0][1] = true;
  ...
  vector<int> soluc(n);
  coloreado(soluc, n, 0, m, mapa);
}
```

Coloreado de un mapa

- Solo quedaría implementar la función esValida. Debe comprobar que los nodos ya coloreados adyacentes a k tienen un color distinto al asignado.

```
bool esValida(const Mapa& mapa, const vector<int>& sol, int k) {
  bool valida = true;
  int i = 0;
  while (valida && i < k){
    if (mapa[i][k] && soluc[i] == soluc[k]) valida = false;
    else ++i;
  }
  return valida;
}
```

- Se pueden intentar cosas más sofisticadas para que esValida sea más eficiente. Por ej. se podrían usar listas de adyacencia y solo iterar por los adyacentes.

Problema de las n reinas

Problema de las n reinas

Queremos colocar n reinas en un tablero de ajedrez $n \times n$ 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 fila distinta, así que podemos representar la solución como $\langle x_0, x_1, \ldots, x_{n-1} \rangle$, donde $x_i \in 0..n-1$ es la columna donde está la reina de la fila i. Por ej., para $n = 4$, $\langle 1, 3, 0, ? \rangle$ representa:

	0	1	2	3
0		Q		
1				Q
2	Q			
3				

Problema de las n reinas

- De nuevo, podemos plantar directamente nuestro esquema de backtracking:

```
void nreinas(vector<int>& sol, int n, int k) {
  for (int i = 0; i < n; ++i) {
    sol[k] = i;
    if (esValida(sol,k)) {
      if (k == n - 1) cout << soluc;
      else nreinas(sol, n, k+1);
    }
  }
}
```

- esValida debe comprobar que la reina k y ninguna i anterior ($i < k$):
  1. no estén en la misma columna (sol[i] != sol[k]), y
  2. no compartan diagonal (abs(sol[k] - sol[i]) != (k - i)).
- Obviamente nunca puede estar en la misma fila por la manera en que construimos la solución.

Problema de las n reinas

```
bool esValida(const vector<int>& sol, int k){
  bool valida = true;
  int i = 0;
  while (valida && i < k) {
    if (sol[k] == sol[i] || (k - i == abs(sol[k] - sol[i])))
      valida = false;
    else ++i;
  }
  return valida;
}
```

- Podemos usar un marcador (vector<**bool**>) para marcar las columnas ya ocupadas.
- Para evitar el bucle necesataríamos otro(s) para marcar también las diagonales ocupadas. Se deja como ejercicio.

Complejidad de algoritmos de backtracking

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

- Si suponemos que esValida siempre devuelve **true** y es constante, 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 esValida revisa al menos que no se repitan las columnas, 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 bien con las plantillas de coste**.

Complejidad de algoritmos de backtracking

- A la vista del último caso de la plantilla por sustracción se intuye, eso sí, una complejidad exponencial.
- Sigamos 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 invocación.

Complejidad de algoritmos de backtracking

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

1. El árbol de llamadas de nreinas(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

$$\text{En total hay } \sum_{i=0}^{n-1} n^i \leq 2 \cdot n^{n-1} \text{ llamadas}$$

2. Cada llamada, si se usan marcadores, tendrá un coste en $O(n)$ porque el cuerpo del bucle tiene coste constante.
3. Por tanto, aproximamos el coste de nreinas(n,0) a $O(n^n)$.

Complejidad de algoritmos de backtracking

Si esValida al menos evita columnas duplicadas, tendremos que:

1. El árbol de llamadas de nreinas(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

$$\text{En total hay } \sum_{i=1}^{n} \frac{n!}{i!} \leq 2 \cdot n! \text{ llamadas}$$

2. Cada llamada, si se usan marcadores, tendrá un coste en $O(n)$ porque el cuerpo del bucle tiene coste constante
3. Por tanto, aproximamos el coste de nreinas(n,0) a $O(n \cdot n!)$.

Backtracking para encontrar la primera solución

- 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 **bool** fin inicializado a **false**. Cuando se encuentra la primera solución se asigna **true**.
- Extendemos el bucle de la búsqueda para iterar **únicamente mientras** ! fin . De esta manera todas las llamadas recursivas pendientes irán terminando en cadena.

Backtracking para encontrar la primera solución

- Por ejemplo, el algoritmo de las *nreinas* para encontrar solo una solución quedaría así:

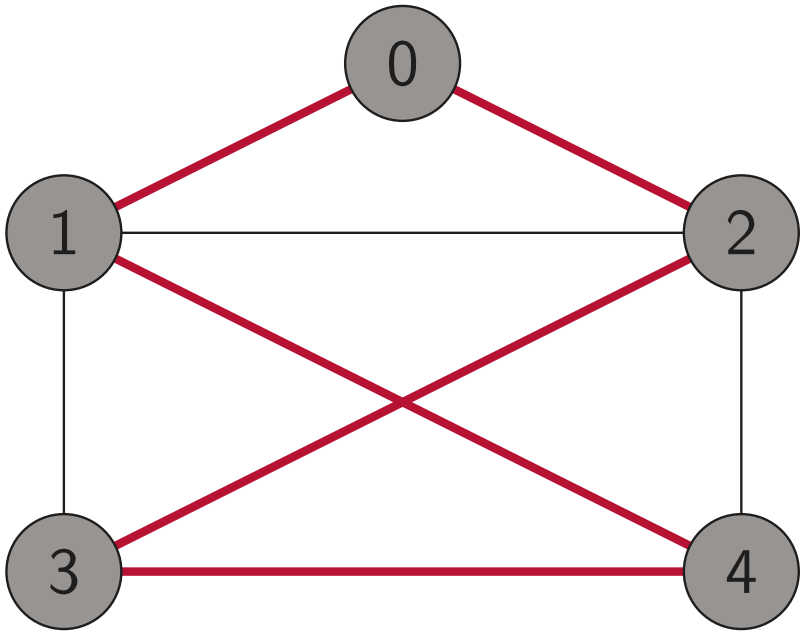
```
void nreinas(vector<int>& sol, int n, int k, bool& fin) {
  for (int i = 0; i < n && !fin; ++i) {
    sol[k] = i;
    if (esValida(sol,k)) {
      if (k == n - 1) {
          cout << soluc;
          fin = true;
      } else
          nreinas(sol, n, k+1);
    }
  }
}

int mainNReinas(){
    cin >> n;
    vector<int> soluc(n);
    bool fin = false;
    nreinas(soluc, n, 0, fin);
}
```

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.



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

Problema

Queremos encontrar un ciclo hamiltoniano de un grafo que **comience 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 $x_1, \ldots, x_{n-1} \in 1 \ldots n-1$.
- Esta solución la podemos ir construyendo paso a paso, así que encaja perfectamente con el esquema de *backtracking*.
- Dada una solución parcial sol correcta hasta el paso $k-1$, saber si es una solución requiere que:
  1. k = n−1, y
  2. existe una arista sol [k] → 0, es decir, cerramos el ciclo.
- Por otro lado, comprobar que sol es válida tras añadir el nodo k requiere verificar que:
  1. Existe una arista sol [k−1] → sol [k], y
  2. el nodo sol [k] no aparece ya en sol [1.. k). Para ello usaremos un marcador para marcar los nodos ya visitados.

Representación del 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 vector<vector<**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

- De nuevo, plantamos directamente nuestro esquema:

```
void cicloHamil(vector<int>& sol, int k, int n, Grafo& grafo,
                vector<bool>& visitados, bool& fin){
  for (int i = 1; i < n && !fin; ++i){ // Nodo 0 ya usado!
    sol[k] = i;
    if (!visitados[i] && grafo[sol[k-1]][i]) { // esValida
      if (k == n - 1 && grafo[n-1][0]) { // esSolucion
        cout << soluc << endl;
        fin = true;
      } else if (k < n-1) {
        visitados[i] = true;
        cicloHamil(sol, k + 1, n, grafo, visitados, fin);
        visitados[i] = false;
      }
    }
  }
}
```

- Importante! En este caso sí que es necesario el **if** (k < n−1). Si no podría no terminar y acceder fuera del rango de los vectores.
- La llamada inicial será con sol [0] = 0, k = 1 y visitados [0] = **true**.

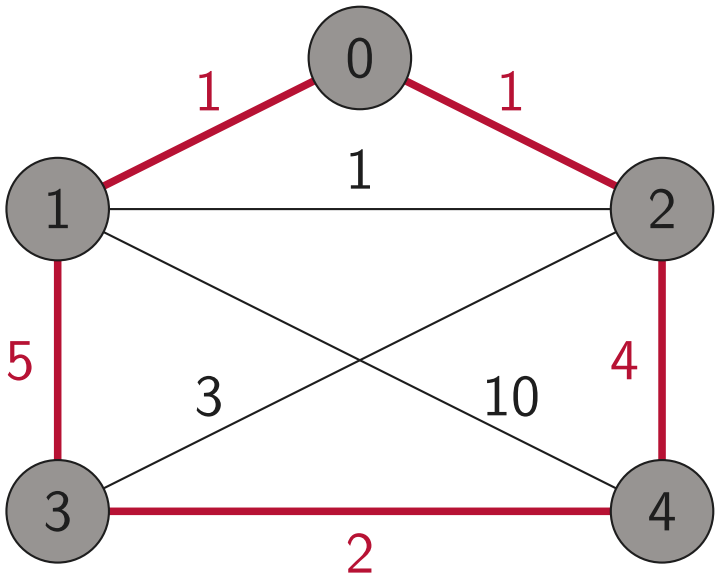
Problema del viajante (*Travelling Salesperson Problem*, o *TSP*)

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?

- Muy parecido al problema de los ciclos hamiltonianos.
- 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

- 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 válidas) pero **únicamente** procesaremos una solución cuando su *valor* sea **mejor** que el de la mejor solución encontrada hasta el momento.
- Ahora la matriz de adyacencia (mapa) no contiene booleanos sino enteros, para almacenar las distancias entre nodos. Supondremos que mapa[a][b] $= 0$ si no existe camino $a \rightarrow b$.

Problema del viajante

```
1  void tsp(vector<int>& sol, int n, int k, Mapa const& mapa,
2           vector<bool>& visitadas, int distancia, int& mejorDist) {
3    for (int ciudad = 1; ciudad < n; ++ciudad) {
4      sol[k] = ciudad;
5      if (!visitadas[ciudad] && mapa[sol[k-1]][ciudad]) { // esValida
6        distancia += mapa[sol[k-1]][ciudad];
7        if (k == n-1 && mapa[ciudad][0]) { // esSolucion
8          if (distancia + mapa[ciudad][0] < mejorDist) // esMejor
9            mejorDist = distancia + mapa[ciudad][0];
10       }
11       else if (k < n-1){ // Necesario, igual que en cicloHamil
12         visitadas[ciudad] = true;
13         tsp(sol, n, k+1, mapa, visitadas, distancia, mejorDist);
14         visitadas[ciudad] = false;
15       }
16       distancia -= mapa[sol[k-1]][ciudad]; //Ojo, necesario tb!
17     }
18   }
19 }
```

- Lo nuevo para hacer optimización son las variables distancia y mejorDist, y las líneas 6, 8, 9 y 16.

Problema del viajante

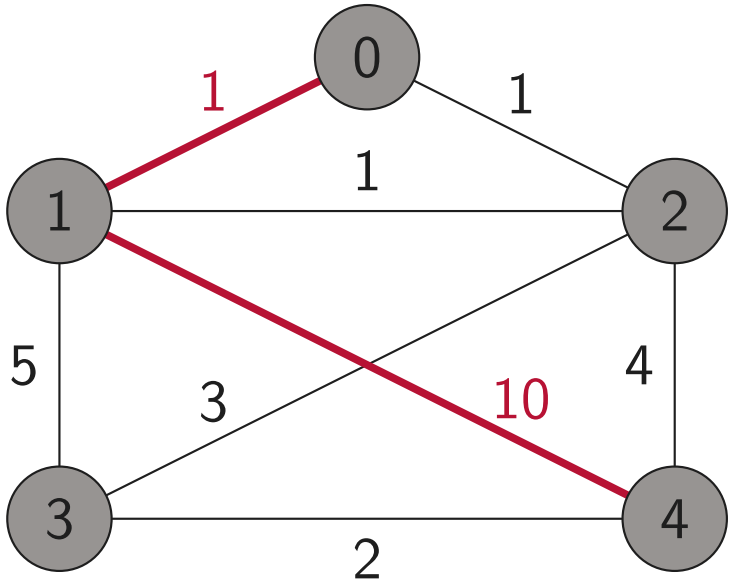
- La función principal tendría esta pinta:

```
void mainTsp(){
    int n;
    cin >> n;
    Mapa mapa(n,vector<int>(n,0));
    // Lectura del mapa
    vector<bool> visitadas(n,false);
    vector<int> soluc(n);
    soluc[0] = 0;
    visitadas[0] = true;
    int mejorDist = INT_MAX; // Para asegurar que se actualiza
    tsp(soluc, n, 1, mapa, visitadas, 0, mejorDist);
}
```

Poda por estimación para el TSP

- Podemos utilizar la mejor solución encontrada hasta el momento junto con *estimaciones optimistas* para «podar» ramas que no llevan a soluciones mejores.
- Para este problema podemos hacer lo siguiente:
  - Sea minArista > 0 el valor de la menor arista del mapa.
  - 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 minArista, por lo tanto sumarían como mínimo una distancia adicional de (n−k)∗minArista
  - Si esta distancia adicional sumada a v es igual o superior a la distancia de la mejor solución encontrada hasta el momento no tiene sentido seguir explorando: **en ningún caso podrá genera una solución completa mejor que la que ya tenemos**.

Ejemplo de poda por estimación para el TSP



- En este grafo la **arista mínima** tiene distancia 1.
- Imaginemos 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**.

Implementando la poda por estimación

- Simplemente tendríamos que extender el cuerpo del **else if** así:

```
...
else if (k < n-1){
   int estimacionOptimista = (n-k)*aristaMin;
   if (distancia + estimacionOptimista < mejorDist){
      visitadas[ciudad] = true;
      tsp(sol, n, k+1, mapa, visitadas, distancia, mejorDist, aristaMin);
      visitadas[ciudad] = false;
   }
}
...
```

Esquema de backtracking para optimización

```
void backtrackingOpt(vector<T>& sol, int n, int k, ...,
                     int valor, int mejorValor, vector<T>& mejorSol) {
  for (auto c : candidatos(k)) {
    sol[k] = c;
    valor += ...; // Se incrementa el valor de la solución correspondientemente
    if (esValida(sol, k, ...)) {
      if (esSolucion(sol, k, n))
        if (esMejor(valor, mejorValor) { // Suele ser simplemente un > o un <
          mejorValor = valor;
          mejorSol = sol;
      } else if (k < n-1) {
        marcar(...);
        if (esMejor(valor + estimacionOptimista(...), mejorValor)) {
          backtrackingOpt(sol, n, k + 1, ...)
        }
        desmarcar(...);
      }
    }
    valor -= incrementoValor(...); // Se decrementa lo mismo que se incrementó
  }
}
```

- Para calcular la estimación optimista es habitual que haya que hacer precálculos antes de empezar el backtracking y pasar más parámetros.
- El vector solución no siempre es necesario pero en cualquier caso es bueno tenerlo por claridad y para depurar.

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.

- Podríamos definir la tupla solución con los productos que vamos eligiendo. Habría que evitar permutaciones y se complicaría el asunto. Se deja como ejercicio.
- Tomaremos mejor tuplas de n elementos $\langle x_0, \ldots, x_{n-1} \rangle$, pero con cada elección de tipo booleano:
  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.

Problema de la mochila 0-1

Podemos aplicar el esquema de backtracking para optimización solo que al haber dos alternativas es mejor desdoblar el bucle **for** en dos casos explícitos. El resto será fácil de completar:

- Para incrementar el valor simplemente sumamos el valor del artículo a_k si este elemento ha sido elegido.
- Cualquier tupla será solución cuando $\mathtt{k} == \mathtt{n}-1$.
- esMejor(valor , mejorValor): Será mejor si es **mayor**.
- esValida (sol , k): Simplemente hay que asegurar que el peso ya incrementado no excede a P.
- Solo faltaría implementar una poda por estimación.

Problema de la mochila 0-1

- Desdoblando los hijos izquierdo y derecho el algoritmo quedaría así:

```
void mochila(vector<bool>& sol, int n, int k, vector<int> const& p,
             vector<int> const& v, int& peso, int P, int valor, int& mejorValor) {
  // Hijo izquierdo: Tomo el objeto k-ésimo
  sol[k] = true;
  peso += p[k];
  valor += v[k];
  if (peso <= P){
    if (k == n-1)
      if (valor > mejorValor) mejorValor = valor;
    else
      mochila(sol, n, k+1, p, v, peso, P, valor, mejorValor);
  }
  peso -= p[k];
  valor -= v[k];

  // Hijo derecho: No tomo el objeto k-ésimo
  sol[k] = false; // Al no tomarlo no hace falta actualizar peso y valor
  if (k == n-1) // Tampoco es necesario chequear peso <= P
    if (valor > mejorValor) mejorValor = valor;
  else
    mochila(sol, n, k+1, p, v, peso, P, valor, mejorValor);
}
```

- Solo faltaría plantear una poda por estimación.

Poda por estimación para problema de la mochila

- Podríamos utilizar la *densidad* d_i de los artículos[1], 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 *peso* el peso actual de la mochila para una solución parcial con valor *valor*.
- En este caso, el mayor valor que podríamos obtener sería rellenar todo el peso restante con artículos de la máxima densidad, es decir, $(P - peso) * D$.
- Por lo tanto una aproximación optimista sería considerar que una solución parcial con valor *valor* se podría completar (en el mejor de los casos) hasta un valor de $valor + (P - peso) * D$.
- Si con esa estimación optimista no mejoramos el mejor valor encontrado hasta el momento no hay que seguir explorando esta rama.

[1] Se pueden encontrar otras aproximaciones

Ejemplo de poda por estimación

	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 sería 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!**

Implementando la poda por estimación

- Simplemente tendríamos que extender los cuerpos de los dos **else** así:

```
...
else {
  float estimacionOptimista = (P-peso)*D;
  if (valor + estimacionOptimista > mejorValor)
    mochila(sol, n, k+1, p, v, peso, P, valor, mejorValor, D);
}
...
```

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 ($\text{sol}[k] = \textbf{true}$) o dejarlo fuera de la mochila ($\text{sol}[k] = \textbf{false}$).
- 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 suelen estar 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.[2]

[2] 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, Gecode, Z3) y también existen resolutores de restricciones integrados en otros paradigmas de programación (p.ej. programación lógica con restricciones, CLP).

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`