Algoritmia
Grado en Ingeniería Informática del Software
Escuela de Ingeniería Informática – Universidad de Oviedo

Backtracking

Juan Ramón Pérez Pérez

jrpp@uniovi.es

Colocar 8 reinas sin que se coman

Problema de las *n* reinas

Se pretende colocar *n* reinas en un tablero de ajedrez sin que ninguna reina pueda comer a otra.

Una reina puede moverse cualquier número de casillas en horizontal, vertical o en diagonal.

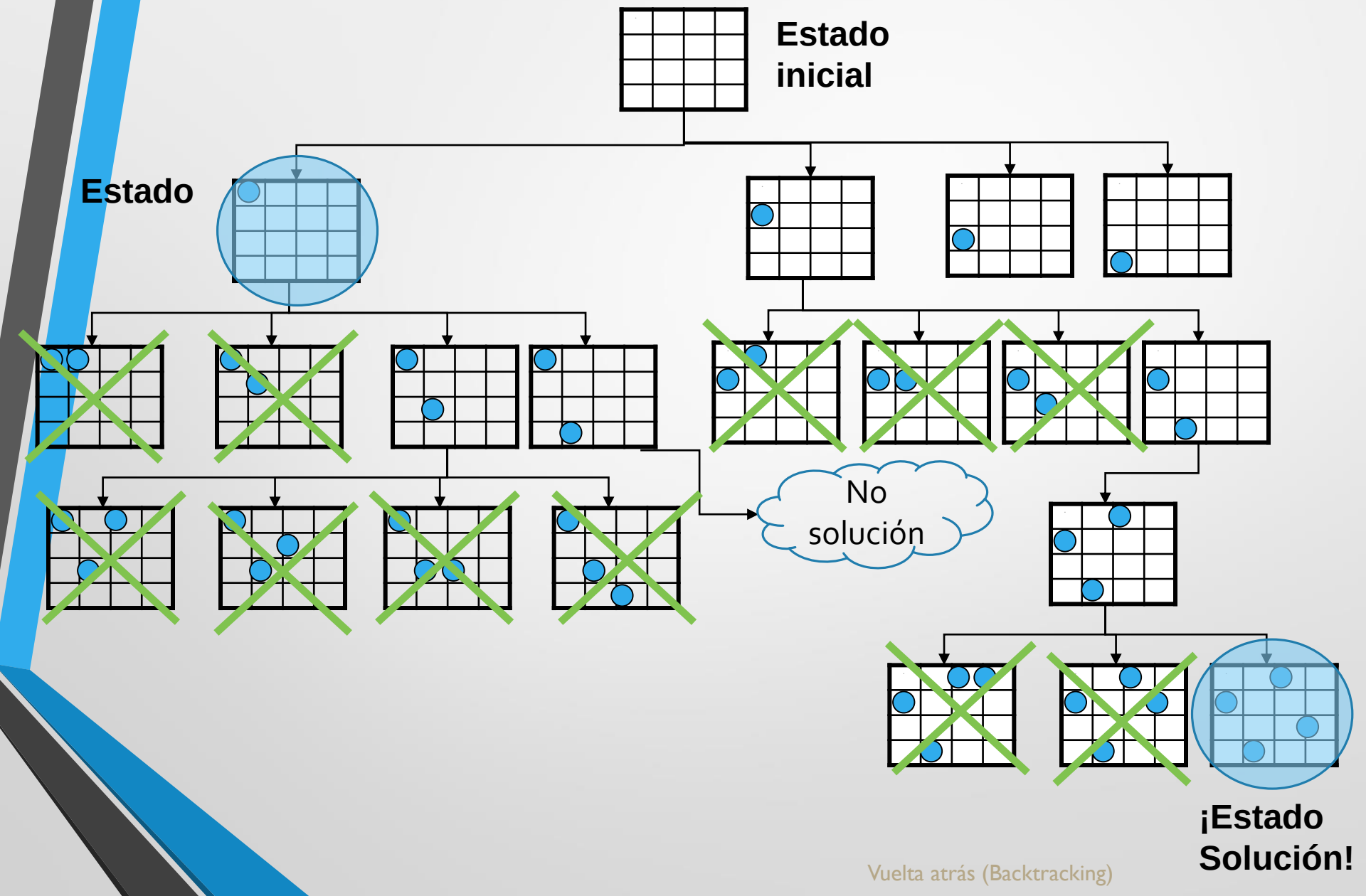
Problemas representados en forma de grafos

Muchos problemas que se pueden concebir en términos de **grafos** abstractos.

Un **nodo** del grafo representa el **estado** del problema. Las aristas representan cambios válidos.

Para solucionar el problema debemos buscar un **nodo solución** y un **camino** en el grafo asociado.

Grafo 4 reinas

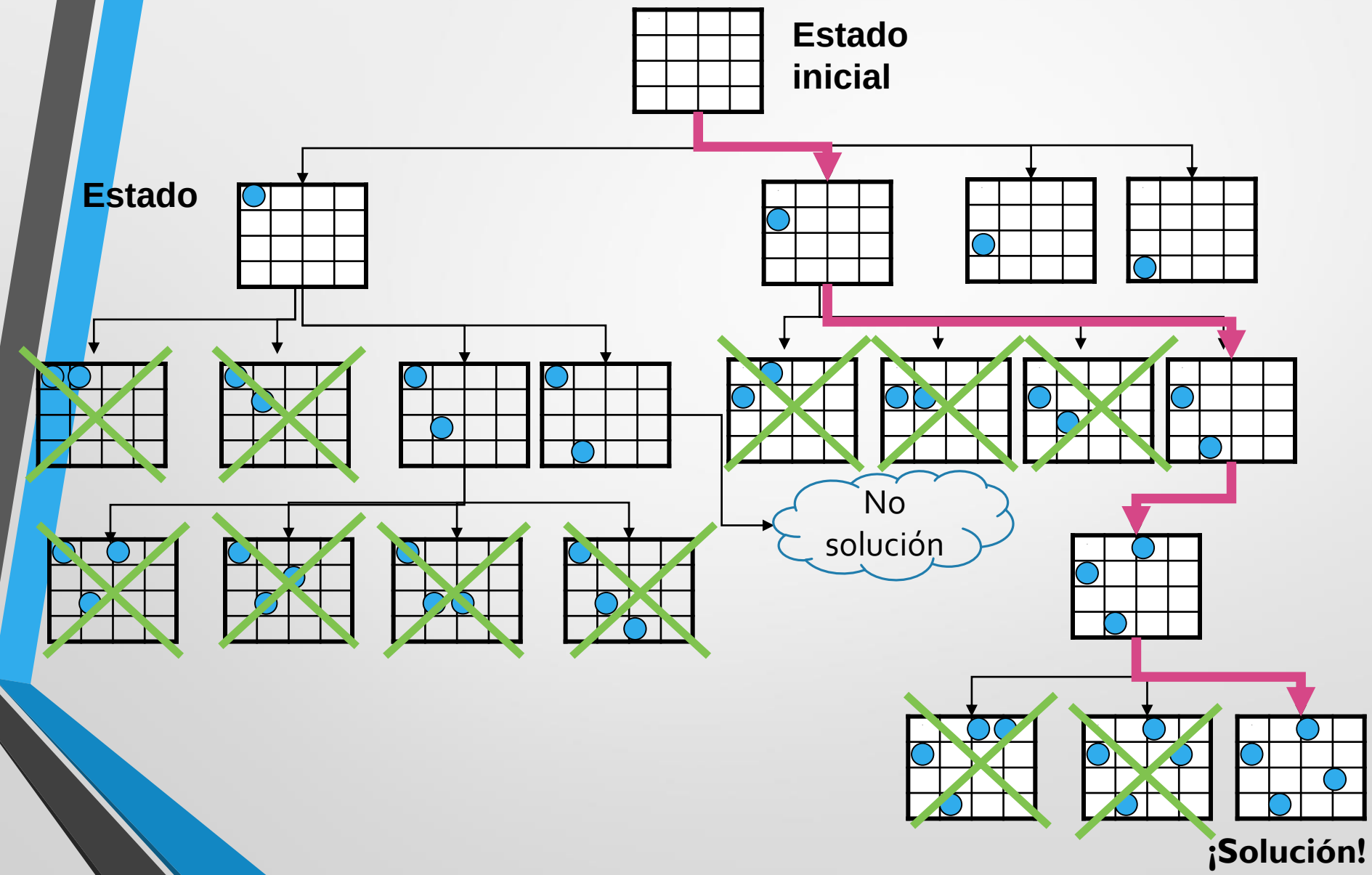


Recorrido en profundidad de un grafo

```
public recorridoProfundidad(Grafo g)
{
    Nodo n;
    for (; g.hayNodos(); n= g.siguienteNodo())
      if (!n.getVisitado())
            recorridoProfundidad(n);
}

private recorridoProfundidad(Nodo n)
{
    Nodo nodoAdyacente;
    n.setVisitado(true);
    for (; n.quedanAdyacentes(); nodoAdyacente= n.getSiguienteAdyacente())
      if (!nodoAdyacente.getVisitado())
            recorridoProfundidad(nodoAdyacente);
}
```

Recorrido en profundidad n reinas



Grafo implícito

Si el grafo contiene un número elevado o infinito de nodos es imposible construirlo explícitamente

No creamos el grafo → ***grafo implícito***.

Disponemos de descripción de sus nodos y aristas → construir el grafo a medida que progresa el recorrido.

Se **ahorra tiempo de computación**, si el recorrido tiene éxito antes de haber construido todo el grafo.

Economía de espacio, ya que se pueden descartar nodos una vez examinados.

Construcción de la solución del problema

La **solución** de un problema de backtracking se puede expresar como un vector $(x_1, x_2, \ldots, x_n)$.

Este vector se construye progresivamente. En cada momento, el algoritmo se encontrará en un cierto nivel **k**, con un **estado** $(x_1, \ldots, x_k)$ que constituye una solución parcial.

Si se puede añadir un nuevo elemento a la solución $\boldsymbol{x_{k+1}}$, se genera y se avanza al nivel ***k+1***.

Recorrido en profundidad del árbol

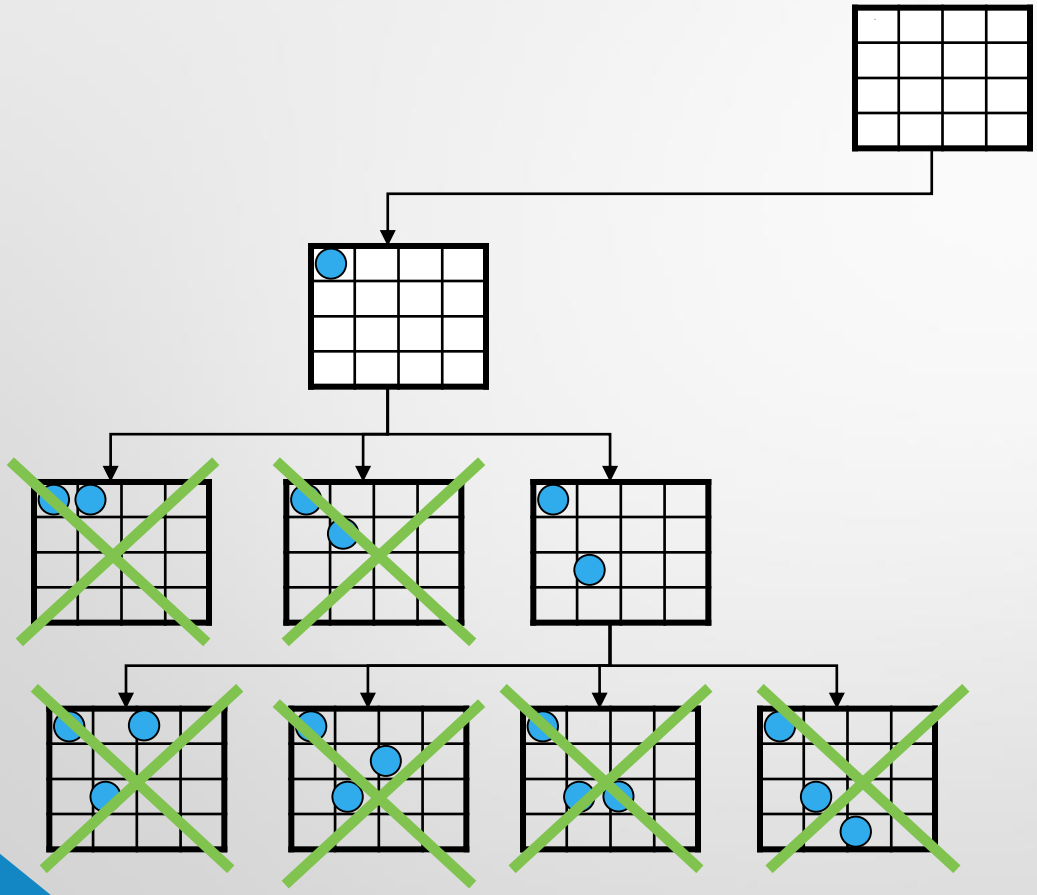
El grafo implícito que se recorre en backtracking se suele reducir a un **árbol**, se denomina árbol de estados.

Con la técnica del backtracking se realiza un **recorrido en profundidad** del árbol implícito.

Objetivo del recorrido: encontrar los estados que sean solución al problema.

Para ello realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones.

Recorrido en profundidad del árbol implícito



Búsqueda de una o más soluciones

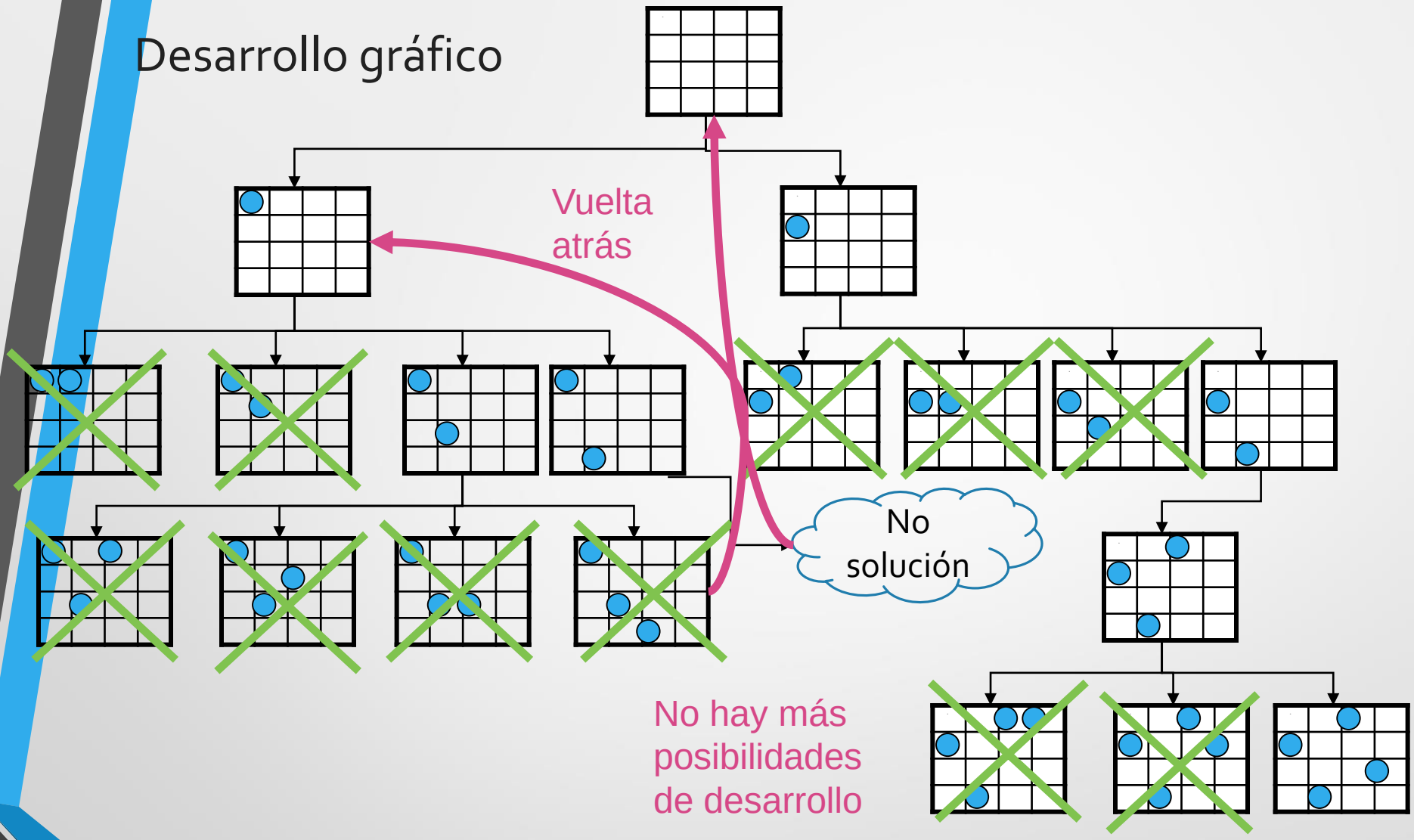
El recorrido tiene **éxito** si se puede definir por completo una solución.

- En este caso, el algoritmo puede detenerse (si lo único que se necesita es una solución al problema), **primera solución**.
- O seguir buscando soluciones alternativas (si deseamos examinarlas todas), **todas las soluciones**.
  - Esto se aplica cuando buscamos una **solución óptima**

Qué es la vuelta atrás

- El recorrido **no tiene éxito** si en alguna etapa la solución parcial construida hasta el momento no se puede completar.
- En tal caso, el recorrido **vuelve atrás**,
  - Eliminando sobre la marcha los elementos que se han añadido en cada fase
  - Cuando vuelve a un nodo que tiene uno o más hijos sin explorar, prosigue el recorrido por estos.

Desarrollo gráfico



¡Solución!

Esquema Backtracking "primera solución" (POO)

```
public static void backtracking(Estado e)
{
     if (e.esSolucion()) {
          System.out.println(e);
          haySolucion= true;
     }
     else {
          Estado estadoHijo= null;
          while(e.hasNextHijos() && !haySolucion) {
               // siguiente estado hijo válido
               estadoHijo= e.nextHijo();

               if (estadoHijo!=null) // puede que no queden hijos válidos
                    backtracking(estadoHijo);
          }
     }
}
```

Clase Estado (POO)

```
public interface Estado
{
/** Devuelve true si este estado es una solución al problema */
public abstract boolean esSolucion();

/** Devuelve el siguiente hijo válido del estado
 * @return devuelve una referencia al siguiente hijo válido, null si no hay
*/
public abstract Estado nextHijo();

/** Devuelve true si quedan hijos *posibles* (puede que no quede ninguno
válido)
 * @return true- si quedan hijos posibles */
public abstract boolean hasNextHijos();
}
```

n reinas. Solución generación de hijos

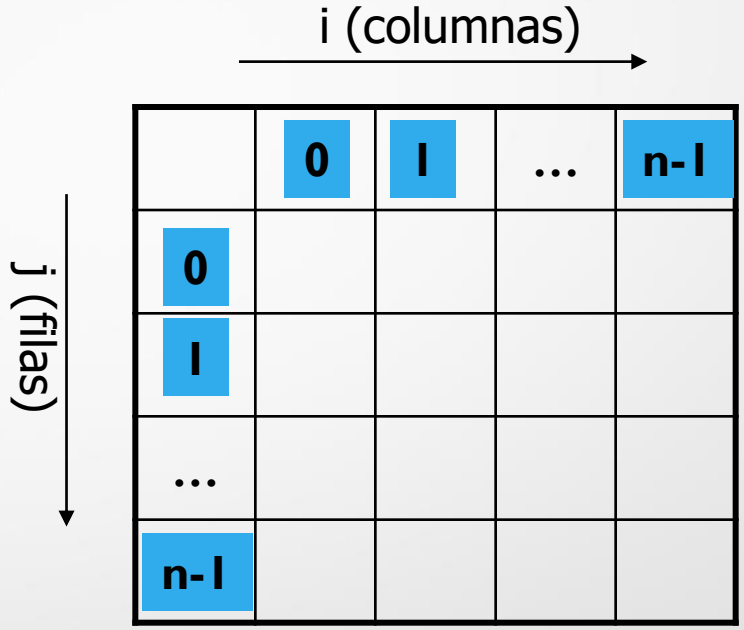
- **Solución**: Cuando tengamos n reinas colocadas
- **Generación de estados hijo**: Tantos como posibilidades tengamos, n, uno por cada fila.
- Comprobar que son **estados válidos**

n reinas. Estado

Para representar el tablero, se van a utilizar una estructura unidimensional

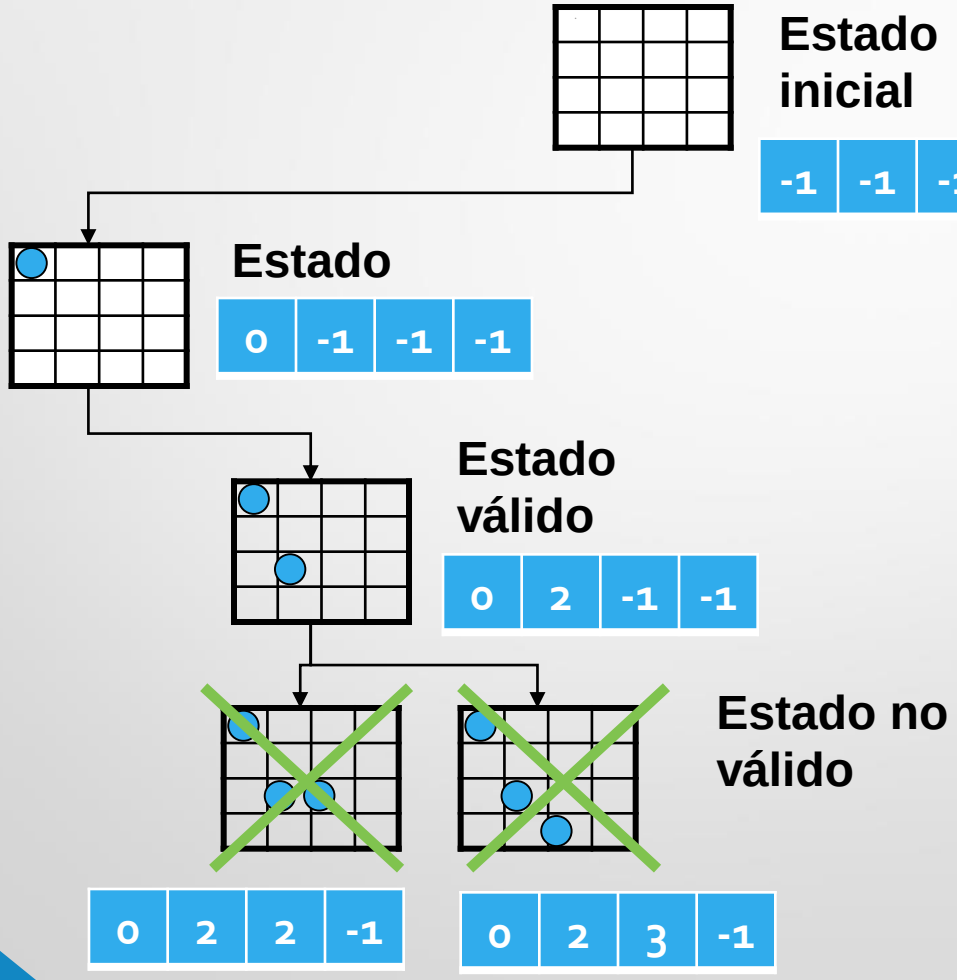
x: array[1..n] of 1..n → int[]
x= new int[n];

si x[i]=j → hay una reina en la columna i y en la fila j.



0	1	...	n-1

Estado de las n reinas



n reinas. Estructuras datos auxiliares para comprobar si posición válida

array [1..n] of boolean → boolean[] a= new boolean[n]

Filas

si a[j]= false → Hay una reina colocada en la fila j.

si a[j]= true → NO hay una reina en la fila j

array [2..2*n] of boolean → boolean[] b= new boolean[(n-1)*2+1]

Las **diagonales**  45° cumplen (i+j)=cte.

b[i+j]=false → Hay reina en la diagonal i+j

b[i+j]=true → NO hay reina en la diagonal i+j

array[-n+1..n-1] of boolean → boolean[] c= new boolean[(n-1)*2+1]

Las **diagonales**  135° cumplen (i-j)=cte.

c[i-j]=false → Hay reina en la diagonal i-j

c[i-j]=true → NO hay reina en la diagonal i-j

n reinas: esquema "primera solución" (versión sin clases)

```
static void backtracking (int j)  {// j: nivel en el árbol de estados
    if (j==n)  {  // solución: ya están colocadas las n reinas
        haySolucion=true;
        imprimir_solucíon("SOLUCION ENCONTRADA");
    }
    else
        for (int i=0;i<n;i++)  // Prueba todas las posibilidades
            if (a[i] && b[i+j] && c[i-j+n-1]   // Estado válido
                    && !haySolucion)  {  // Para la búsqueda
                sol[j] =i;    // Asigna valor para crear estado hijo
                a[i]=false;   b[i+j]=false;         c[i-j+n-1]=false;

                backtracking (j+1); // Recursivo: Profundiza en árbol

                a[i]=true;         b[i+j]=true;     c[i-j+n-1]=true;
            }
}
```

Esquema Backtracking "todas las soluciones"

```
public static void backtracking(Estado e)
{
     if (e.esSolucion()) {
          System.out.println(e);
          cont++;
     }
     else {   // eliminar si hay soluciones después de una solución
          Estado estadoHijo= null;
          while(e.hasNextHijos()) {
              // siguiente estado hijo válido
              estadoHijo= e.nextHijo();

              if (estadoHijo!=null)// puede que no queden hijos válidos
               backtracking(estadoHijo);
          }
     }
}
```

Búsqueda de todas las soluciones

- En la búsqueda de todas las soluciones se desarrollan todos los estados del árbol
- Según el tipo de problema se pueden dar dos casos que habrá que tratar de forma diferente:
  - Por debajo de un estado solución no hay más soluciones.
    - Esto es lo más habitual. Ej.: n reinas
  - Por debajo de un estado solución sí hay más soluciones.
    - Ej.: Buscar caminos no simples en de un grafo, con una longitud <= K.

n reinas: esquema "todas las soluciones" al problema

```
static void backtracking (int j) {  // j: nivel en el árbol de estados
    if (j==n) {   // solución: ya están colocadas las n reinas
        cont++;
        imprimir_solucíon("SOLUCION ENCONTRADA");
    }
    else
        for (int i=0;i<n;i++)   // Prueba todas las posibilidades
            if (a[i] && b[i+j] && c[i-j+n-1]) {// Estado válido
                sol[j] =i;  // Asigna valor para crear estado hijo
                a[i]=false;   b[i+j]=false;         c[i-j+n-1]=false;

                backtracking (j+1); // Recursivo: Profundiza en árbol

                a[i]=true;    b[i+j]=true;      c[i-j+n-1]=true;
            }
}
```

Búsqueda de la solución óptima

- En backtracking, buscar la **solución óptima** implica explorar todos los estados para encontrar todas las soluciones
- En vez de guardar todas las soluciones:
  - Cada vez que encontramos una nueva la comparamos con la mejor de las anteriores
  - Sólo la almacenamos si es mejor

Mejora de backtracking: Poda de nodos

La **poda** en backtracking consiste en establecer una condición que impida seguir desarrollando nodos aunque sean válidos

- Esta condición se aplica en el mismo punto que la condición de estado válido

Se aplica cuando tenemos que buscar una solución óptima

- La condición que establecemos consiste en comprobar que podríamos obtener una solución mejor que la que ya tenemos.
- Por ejemplo, en el problema del cambio descartaríamos todos los nodos que impliquen más monedas que la solución obtenida hasta el momento.

Análisis de la complejidad de backtracking

El tiempo de ejecución depende del **número de nodos** generados y del tiempo requerido para **cada nodo**.

Por lo general, el tiempo en cada nodo es constante.

Suponiendo que una solución sea de la forma: $(x_1, x_2, \ldots, x_n)$, en el peor caso se generarán todas las posibles **combinaciones para cada x_i**.

Análisis de la complejidad de backtracking (II)

Si el número de posibles valores para cada $\mathbf{x}_i$ es $\mathbf{m}_i$ (grado del árbol) entonces se generan:

m_1 nodos en el nivel 1

$m_1 \cdot m_2$ nodos en el nivel 2

$m_1 \cdot m_2 \cdot \ldots \cdot m_n$ nodos en el nivel n

Para un problema $m_i = 2$ (grado 2). El número de nodos generados es:

$t(n) = 2 + 2^2 + 2^3 + \ldots + 2^n = 2^{n+1} - 2 \in O(2^n)$

Para un problema en el que el grado del árbol se reducen en uno en cada nivel.

$t(n) = n + n \cdot (n-1) + n \cdot (n-1) \cdot (n-2) + \ldots + n! \in O(n!)$

En general, órdenes de complejidad factoriales o exponenciales.

Ejemplos de vuelta atrás

Formar palabras a partir de un conjunto de letras

- Disponemos de un conjunto de ***n***=*3* letras:{*a, b, c*}
- Queremos formar palabras con una longitud ***m***=*2* sin que se repitan las letras. Siendo *m*<=*n*.
- Llamamos palabra a cualquier conjunto de letras, que no necesariamente tiene significado.

Variaciones sin repetición

- Denominamos ***variaciones ordinarias*** o sin repetición de **n** elementos tomados de **m** en **m** (siendo **m<=n**) a cada uno de los distintos grupos de m elementos escogidos de entre los n, de manera que:
  - En cada grupo, los m elementos sean distintos.
  - Dos grupos son distintos, si difieren en algún elemento o en el orden de colocación.

Variaciones de n elementos de un vector v

```
public static void backtracking (int nivel)
{
    if (nivel==m)  //hay ya una palabra completa
    {
        mostrarSolucion();
        cont++;
    }
    else
        for (int i=0;i<n;i++)
            if (!marca[i])
            {
                sol[nivel] =vLetras[i];
                marca[i]=true;
                backtracking (nivel+1);
                marca[i]=false;
            }
}
```

Número de variaciones y variables de implementación

- El número de variaciones ordinarias lo representamos $V_{n,m}$ y se calcula:
  - $V_{n,m}$=n.(n-1).(n-2)…(n-m+1)
- Tenemos tres letras: {a, b, c}, y queremos formar palabras de dos letras:
  - n= 3, m= 2
  - $V_{3,2}$= 3·2= 6
- Análisis de la complejidad
  - Grado del árbol de estados: n=3 pero en cada nivel hay un hijo menos
  - La complejidad máxima suponiendo todos los estados válidos es O(n!)= O(3!)

Problema de la asignación de tareas a agentes

- Hay que asignar *j* tareas a *i* agentes, de forma que cada agente realiza sólo una tarea.
- Se dispone de una matriz de costes de ejecución de una tarea *j* por un agente *i*.
- Objetivo: Minimizar la suma de los costes de ejecutar las *n* tareas.

j (tareas) →

i (agentes) ↓

	1	**2**	**…**	**n**
a	11	12	...	40
b	14	15	...	22
…				
n	17	14	...	28

Código asignación agentes tareas

```
static void backtracking (int nivel) {
     if (nivel==n) { //hay ya una asignación completa
          if (coste<costeMejor) {
               for (int k=0;k<n;k++)
                    asigMejor[k]=asig[k];
               costeMejor=coste;
          }
     }
     else {
          for (int i=0;i<n;i++)
               if (!marca[i] && coste<costeMejor ) {
                    asig[nivel]=i;
                    coste=coste+w[nivel][i];
                    marca[i]=true;
                    backtracking (nivel+1);
                    coste=coste-w[nivel][i];
                    marca[i]=false;
               }
     }
}
```

Problema del salto del caballo. Descripción

Consiste en recorrer un tablero de ajedrez completo, utilizando los movimientos como los que realiza el caballo.

1		7			
8				6	
	2				
	9				5
		3			
		10		4	

	5		4	
6				3
		*		
7				2
	8		1	

Las casillas a las que puede acceder un caballo desde una dada se muestran en la tabla adjunta.

Problema del salto del caballo. Representación del estado

- t array que representa el tablero:
- **int [][] t= new int [n][n]**
  - t[x][y]=o, la casilla (x y) no ha sido visitada.
  - t[x][y]=i, la casilla (x, y) ha sido visitada en el orden i-ésimo.
- i, número de movimiento del caballo.
- x, y: coordenadas del último movimiento del caballo
- Casilla inicial [1][1]

	1	2	3	4	5	6	7	n
1	1	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0

Problema del salto del caballo.
Reglas de variación de los estados

Los desplazamientos posibles del caballo los expresamos mediante dos vectores:

Desplazamiento horizontal, *h*

Desplazamiento vertical, *v*

	5º		4º	
6º				3º
		*		
7º				2º
	8º		1º	

Se ha establecido, de forma arbitraria, el orden para escoger los movimientos del caballo.

h	1	2	2	1	-1	-2	-2	-1
v	2	1	-1	-2	-2	-1	1	2

Aplicación del esquema "1ª solución" a este problema

- Llamada recursiva, pasamos como parámetros el número de movimiento y las coordenadas actuales.
  - backtracking(int i, int x, int y) ← Declaración de parámetros
  - backtracking(i+1,u,v) ← Llamada
- Condición de solución
  - i == n*n+1 → Completamos el tablero
- Bucle para seleccionar un estado
  - Índice k que varía entre 0 y NUM_MOVIMIENTO del caballo
  - Posición destino: u= x+h[k]; v= y+v[k];
- Condición de estado válido
  - Comprobar que esté dentro de los límites del tablero.
  - Comprobar que la casilla no haya sido visitada.
- Anotar nuevo estado → tablero[u][v]= i;
- Borrar anotación → tablero[u][v]= 0;

Problema del salto del caballo. Código

```
static void backtracking (int salto,int x,int y) {
    if (salto==n*n+1)  {  // ya  acabó de recorrer tablero
        seEncontro=true;
        mostrarSolucion();
    }
    else
        for (int k=0;k<=7;k++)  {
            int u=x+h[k];   // nueva posición
            int v=y+v[k];

            if (!seEncontro &&
                u>=0 && u<=n-1 && v>=0 && v<=n-1 &&  // dentro tablero
                tab[u][v]==0)  {  // casilla no utilizada
                 tab[u][v]=salto;
                 backtracking (salto+1,u,v);
                 tab[u][v]=0;
            }
        }
}
```

Análisis de la complejidad

- x, y: coordenadas en el tablero de la posición del caballo en cada momento.
- u, v: coordenadas destino del caballo.

- Análisis de la complejidad:
  - Grado del árbol de estados: 8
  - Altura del árbol de estados: n^2
  - La complejidad máxima suponiendo todos los estados válidos es $O(8^{n^2})$

Problema propuesto

Subconjuntos de una suma dada

Dado un vector de enteros positivos y un valor, realizar un programa que devuelva los subconjuntos del vector que sumen ese valor dado.

Si, ahora, buscamos el subconjunto con menor número de elementos. ¿Qué habría que cambiar en el código?¿Se podría optimizar la búsqueda de la solución?

Ejercicio propuesto 2

Camino Simple Menor

Se pide calcular el camino simple de **menor** coste entre un nodo origen y un nodo destino (origen <> destino) en un grafo dirigido de pesos positivos.

Camino Simple Mayor

Se pide calcular el camino simple de **mayor** coste entre un nodo origen y un nodo destino (origen <> destino) en un grafo dirigido de pesos positivos.

Resuelto en el seminario 4