

# Vuelta atrás

Algoritmos y estructuras de datos  
Curso 2017/2018

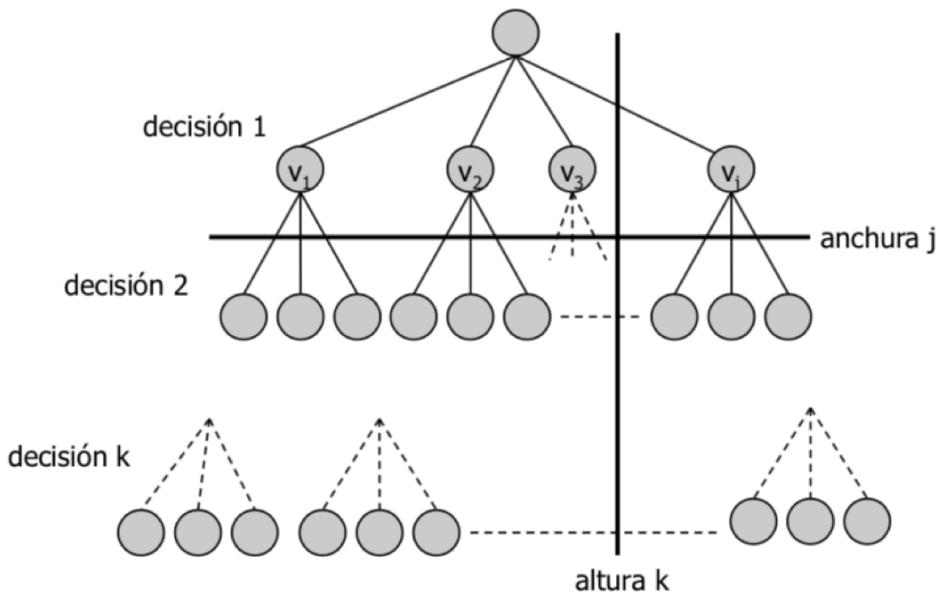
Fernando Ortega, PhD.

# Algoritmo de vuelta atrás

- Es un esquema que de forma sistemática y organizada, genera y recorre un espacio que contiene **todas los posibles estados** (secuencias de decisiones).
  - Si existe solución, seguro que la encuentra.
- Este espacio se denomina el **espacio de búsqueda** del problema, y se representa como un árbol sobre el que el algoritmo hace un recorrido en profundidad partiendo de la raíz.
- Se conoce de antemano el **orden** en que se van a generar y recorrer sus nodos, y se continua recorriendo el árbol mientras se cumplan las restricciones.
- Normalmente el árbol de búsqueda **crece de forma exponencial** y es habitual plantear criterios de poda para evitar explorar caminos que no conducen al éxito.
- Tiene tres posibles esquemas: encontrar una solución factible, encontrar todas las soluciones factibles, encontrar la mejor solución factible.

# Espacio de búsqueda

- Es **espacio de búsqueda** se define por:
  - la **altura del espacio**: hay  $k$  estados por los que pasar para formar una solución.
  - la **anchura del espacio**: cada estado tiene asociado un dominio formado por  $j$  decisión/candidatos distintos.



# Esquema básico

- La **solución** se puede expresar como una tupla  $(x_1, x_2, \dots, x_n)$ , satisfaciendo unas restricciones  $P(x_1, x_2, \dots, x_n)$  y tal vez optimizando una cierta función objetivo.
- En cada momento, el algoritmo se encontrará en un cierto nivel  $k$ , con una **solución parcial**  $(x_1, \dots, x_k)$ .
  - Si se puede añadir un nuevo elemento a la solución  $x_{k+1}$ , se genera y se avanza al nivel  $k+1$ .
  - Si no, se prueban otros valores de  $x_k$ .
  - Si no existe ningún valor posible por probar, entonces se retrocede al nivel anterior  $k-1$ .
- Se sigue hasta que la solución parcial sea una **solución completa** del problema, o hasta que **no queden más posibilidades**.
- El resultado es equivalente a hacer un **recorrido en profundidad** en el árbol de soluciones.
  - Este árbol es implícito, no se almacena en ningún lugar.

# Esquema general

```
FUNCION vuelta_atras (estado, éxito, ...)  
    iniciar conjunto de candidatos  
    REPETIR  
        seleccionar un candidato  
        SI aceptable(candidato) ENTONCES  
            SI es_solución(candidato) ENTONCES  
                éxito = true  
            SINO  
                vuelta_atras (proximo_estado, éxito, ...)  
                SI NO éxito ENTONCES  
                    desanotar candidato  
    HASTA éxito OR no más candidatos
```

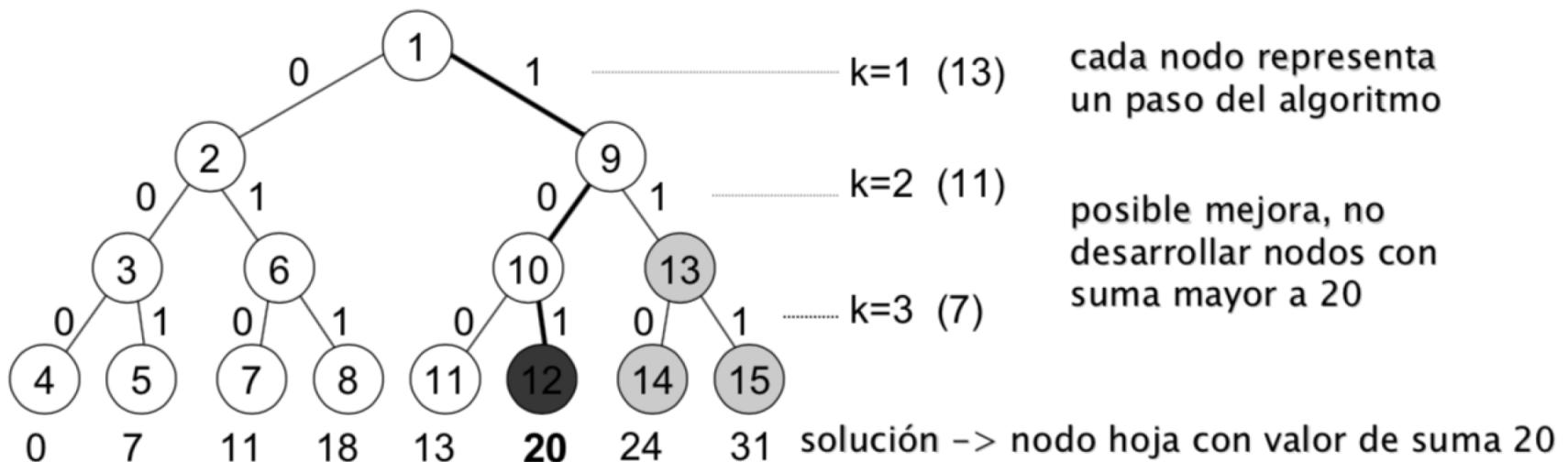
# Ejemplo: suma de enteros (I)

- Dado un conjunto de números enteros  $\{13, 11, 7\}$ , encontrar si existe algún subconjunto cuya suma sea exactamente 20.

¿Cómo lo hacemos?

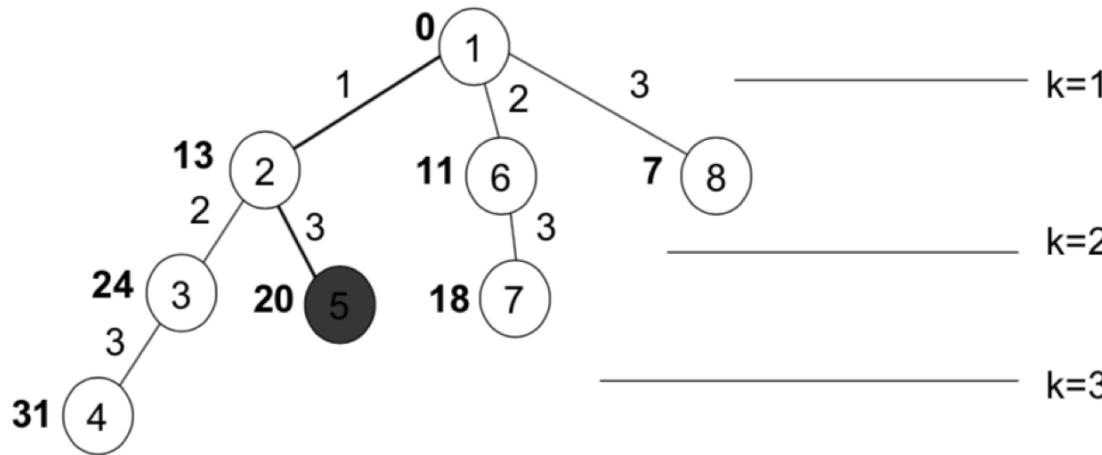
# Ejemplo: suma de enteros (II)

- **Árbol binario**: en cada nivel  $k$  decidir si el elemento  $i$  está o no en la solución.
- Representación de la solución:  $(x_1, x_2, x_3)$ , donde  $x_i = (0, 1)$ .



# Ejemplo: suma de enteros (III)

- **Árbol combinatorio:** En cada nivel  $k$  decidir qué elemento se añade (1, 2 o 3).
- Representación de la solución  $(s_1, \dots, s_m)$ , donde  $m \leq n$  y  $s_i \in \{1, 2, 3\}$ .



- Cada nodo es una  **posible solución**. Será válida si la suma es 20.
- Necesitamos funciones para generar los nodos, para descartar nodos y para saber si un nodo es solución.

# Resolución de problemas (I)

## 1. Identificar el problema.

- ¿Es la vuelta atrás el método más adecuado?

## 2. Identificar la información que define el estado del problema.

- Determinar el conjunto de propiedades que definen cada uno de los posibles estados del problema.
- Elegir cómo se representa la información del estado (estructura de datos).

## 3. Identificar el estado inicial del proceso de búsqueda.

- Determinar el valor inicial del conjunto de propiedades que definen al estado del problema.
- En aquellos casos en que el estado inicial sea un estado solución no habrá búsqueda exhaustiva.

# Resolución de problemas (II)

## 4. Identificar la solución del problema

- Determinar el valor del conjunto de propiedades que definen al estado del problema para que dicho estado sea un estado solución.
- Una vez se haya alcanzado un estado solución en la búsqueda, el proceso de búsqueda deberá terminar.

## 5. Identificar el conjunto de candidatos posibles

- Determinar las candidatos por las que se puede optar en cada momento.

## 6. Definir el criterio de selección del candidato

- La elección de un determinado candidato debe acercarnos más a la solución del problema.
- El criterio de selección ideal es aquel que nos permita alcanzar la solución de forma más rápida (no siempre es posible contar con esta información).

# Resolución de problemas (III)

## 7. Identificar cuando no se pueden elegir más candidatos:

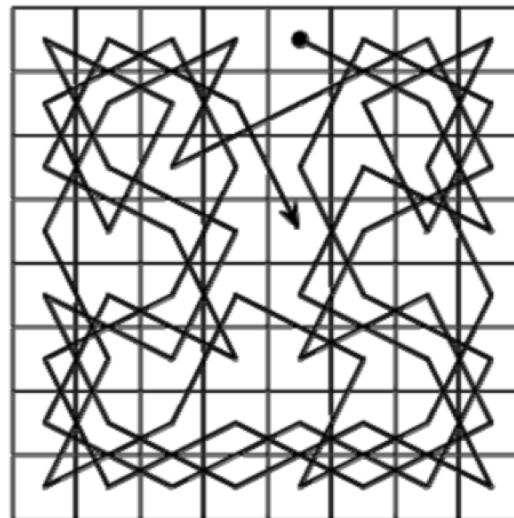
- Se ha de definir cuándo no se pueden elegir más opciones/candidatos, a pesar de que no se ha encontrado la solución del problema.
- Esto es necesario, ya que el proceso podría caer en un bucle infinito y no acabar nunca, y porque si en un estado ya se han realizado todos los movimientos posibles y no se ha encontrado solución, se debe realizar un retroceso para poder continuar con el proceso de búsqueda.
- Por lo tanto, es necesario especificar los valores que deben alcanzar determinadas propiedades para dar por concluido y sin solución, la exploración de un estado en concreto o el proceso de búsqueda global.
- Durante el proceso de búsqueda será necesario realizar un mantenimiento de dichas propiedades.

# Resolución de problemas (IV)

8. Definir cómo se realiza la acción que permite avanzar en el espacio de búsqueda (*anotar*)
  - Si una decisión/candidato es aceptable se avanza en el proceso de búsqueda, y por tanto en el espacio de estados.
  - Este avance se realiza mediante una llamada recursiva.
9. Definir cómo se realiza la acción que permite retroceder en el espacio de búsqueda (*desanotar*)
  - Si una decisión/candidato no es aceptable se retrocede en el proceso de búsqueda, y por tanto en el espacio de estados.
  - Los cambios realizados por la llamada deben deshacerse para volver al estado de recursión anterior y tomar otro camino de búsqueda (vuelta atrás).
10. Identificar cuándo una decisión/candidato es aceptable
  - Una decisión/candidato es aceptable si cumple todas las restricciones que deben cumplirse en el problema dado.

# El problema del caballero (I)

- El Problema del viaje del caballero es un antiguo problema matemático en el que se pide que, teniendo una cuadricula de  $N \times N$  casillas y un caballo de ajedrez colocado en una posición inicial cualquiera  $(x,y)$ , el caballo pase por todas las casillas y una sola vez por cada una de ellas.



# El problema del caballero (II)

- Identificar el problema:
  - El número de casillas de un tablero es  $N \times N = N^2$ , como se parte de una casilla, el número de casillas a alcanzar es de  $N^2 - 1$ , y por lo tanto se deberán realizar un total de  $N^2 - 1$  movimientos para alcanzar las de  $N^2 - 1$  casillas.
- Identificar la información que define el estado del problema:
  - Lo que define el estado del problema es el tablero completo de ajedrez.
  - Podemos utilizar como estructura de datos una matriz de  $N \times N$ .

```
int **tablero = (int **) malloc(N * sizeof(int *));
for(int i = 0; i < N; i++) {
    tablero[i] = (int *)malloc(N * sizeof(int));
    for (int j = 0; j < N; j++) tablero[i][j] = 0;
}
```

Valores enteros de forma que si una casilla tiene un valor distinto de 0 es que ha sido visitada en el  $i$ -ésimo movimiento.

```
tablero[x][y] = 0; // la casilla <x,y> no ha sido visitada
tablero[x][y] = i; // la casilla <x,y> ha sido visitada en el movimiento i
```

# El problema del caballero (III)

- Identificar el estado inicial del proceso de búsqueda:
  - Posición inicial no está impuesta por el problema. Será una variable que dará valor el usuario en el momento en el que se comience el proceso de búsqueda. Importante: llamada inicial del proceso.

```
if (posicionValida(x0, y0) {  
    tablero[x0][y0] = 1;  
    exito = false;  
    probar(tablero, 2, x0, y0, &exito);  
    ...  
}
```

- Identificar la solución del problema:

- Cuando se hayan realizado un total de  $N^2-1$  movimientos, es porque se han visitado las casillas por lo tanto, se ha alcanzado la solución.

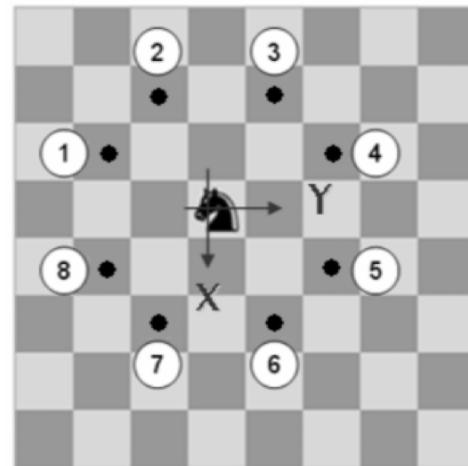
```
if (numMovimientos < N*N) {  
    // Tablero no lleno, solución no alcanzada  
} else {  
    // Solución alcanzada  
}
```

# El problema del caballero (IV)

- Identificar el conjunto de movimientos posibles:
  - Tiene que tener en cuenta la restricción de los movimientos del caballo en el ajedrez.
  - Especificar los movimientos posibles.

Por suma de diferencias de coordenadas

```
int dx[8] = {-1, -2, -2, -1, 1, 2, 2, 1};  
int dy[8] = {-2, -1, 1, 2, 2, 1, -1, -2};
```



# El problema del caballero (V)

- **Definir el criterio de selección del movimiento:**
  - El conjunto de movimientos posibles se almacenará en una estructura de datos de forma que se irá recorriendo posición a posición.
  - El criterio de selección viene impuesto por la posición que ocupe cada uno de los movimientos en la estructura de datos.
  - En este caso el orden se fija al inicializar las variables `dx` y `dy`.
- **Identificar cuándo un movimiento es aceptable:**
  - Debe estar permitido por las reglas de movimiento del caballo.

```
u = x + dx[k];
v = y + dy[k];
```
  - Debe alcanzar una posición valida del tablero, es decir, la posición debe estar dentro de los límites del tablero.
$$(0 \leq u) \ \&\& \ (u < N) \ \&\& \ (0 \leq v) \ \&\& \ (v < N)$$
  - Debe alcanzar una posición no visitada.

```
tablero[u][v] == 0
```

# El problema del caballero (VI)

- Definir cómo se realiza la acción que permite avanzar en el espacio de búsqueda (*anotar*):
  - Si el movimiento del caballero es válido, se anota el número de movimiento en la posición alcanzada.
  - Se incrementa el número de movimiento.

```
tablero[u][v] = numMovimientos;
```
  - Si el número de movimiento se incrementa en la llamada recursiva, el valor pertenece al nuevo estado y no al estado actual. Además facilita la operación de desanotar.
- Si no se ha alcanzado la solución con esta anotación se lanza otra llamada recursiva con el número de movimientos actualizado.

```
if (numMovimientos < N*N) {  
    probar(tablero, numMovimientos + 1, u, v, éxito);  
    ...  
}
```

# El problema del caballero (VII)

- Definir cómo se realiza la acción que permite retroceder en el espacio de búsqueda (*desanotar*)
  - Si no se pueden realizar más movimientos en ese estado y no se ha alcanzado solución, el valor de la posición se ha de poner a 0.
  - El movimiento que se ha procesado no es valido porque no lleva a una solución.

```
if (numMovimientos < N*N) {  
    probar(tblero, numMovimientos + 1, u, v, éxito);  
    if (!éxito) tablero[u][v] = 0;  
} else {  
    éxito = true;  
}
```

- Identificar cuando no se pueden realizar más movimientos:
  - Se han realizado los  $N^2 - 1$  movimientos que recorren el tablero.
  - Se han realizado todos los movimientos posibles de la pieza del caballo dada una determinada posición.

# El problema del caballero (VIII)

```
void probar (int **tablero, int numMovimientos, int x, int y, bool *exito) {  
    int k = -1; // posibles movimientos  
    do {  
        k++;  
        int u = x + dx[k];  
        int v = y + dy[k];  
        if ((0 <= u) && (u < N) && (0 <= v) && (v < N)) {  
            if (tablero[u][v] == 0) {  
                tablero[u][v] = numMovimientos;  
                if (numMovimientos < N*N) {  
                    probar(tablero, numMovimientos+1, u, v, exito);  
                    if (!*exito) tablero[u][v] = 0;  
                } else {  
                    *exito = true;  
                }  
            }  
        }  
    } while (!*exito && k < 7);  
}
```

# El problema del caballero (IX)

