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

Ramificación y poda

Juan Ramón Pérez Pérez

jrpp@uniovi.es

Cómo elegir el mejor equipo posible
Para el 4x100 estilos

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

Problema de la asignación de tareas a agentes.

Ejemplo de cálculo de costes

Ejemplo de cálculo de costes:

- Si asignamos 1→a, 2→b, 3→c, 4→d. El coste total será 11+15+19+28= 73.
- Tarea 4→a,2→b,1→c,3→d. El coste será de 40+15+11+20= 86
- La asignación óptima en este caso es: 1→a,3→b,4→c,2→d. Cuyo coste es 11+13+23+14= 61.

j (tareas)

i (agentes)	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Ramificación y poda: exploración de un árbol de estados

Esta técnica trata de explorar un árbol implícito, igual que el backtracking.

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

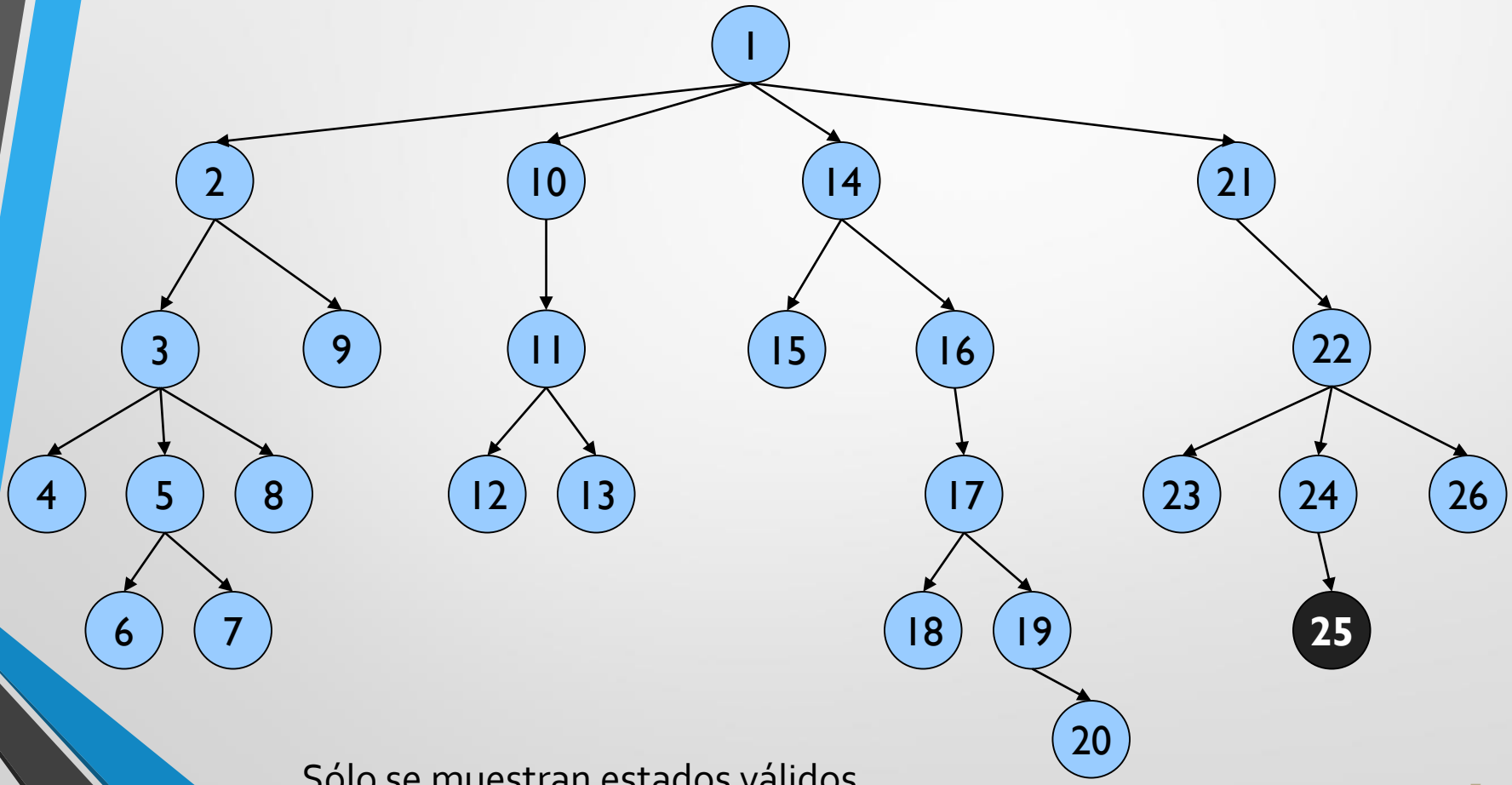
Para solucionar el problema debemos buscar el nodo solución y un camino en el árbol asociado.

Ramificación y poda: recorrido en anchura

En backtracking se desarrolla el árbol de estados en profundidad.

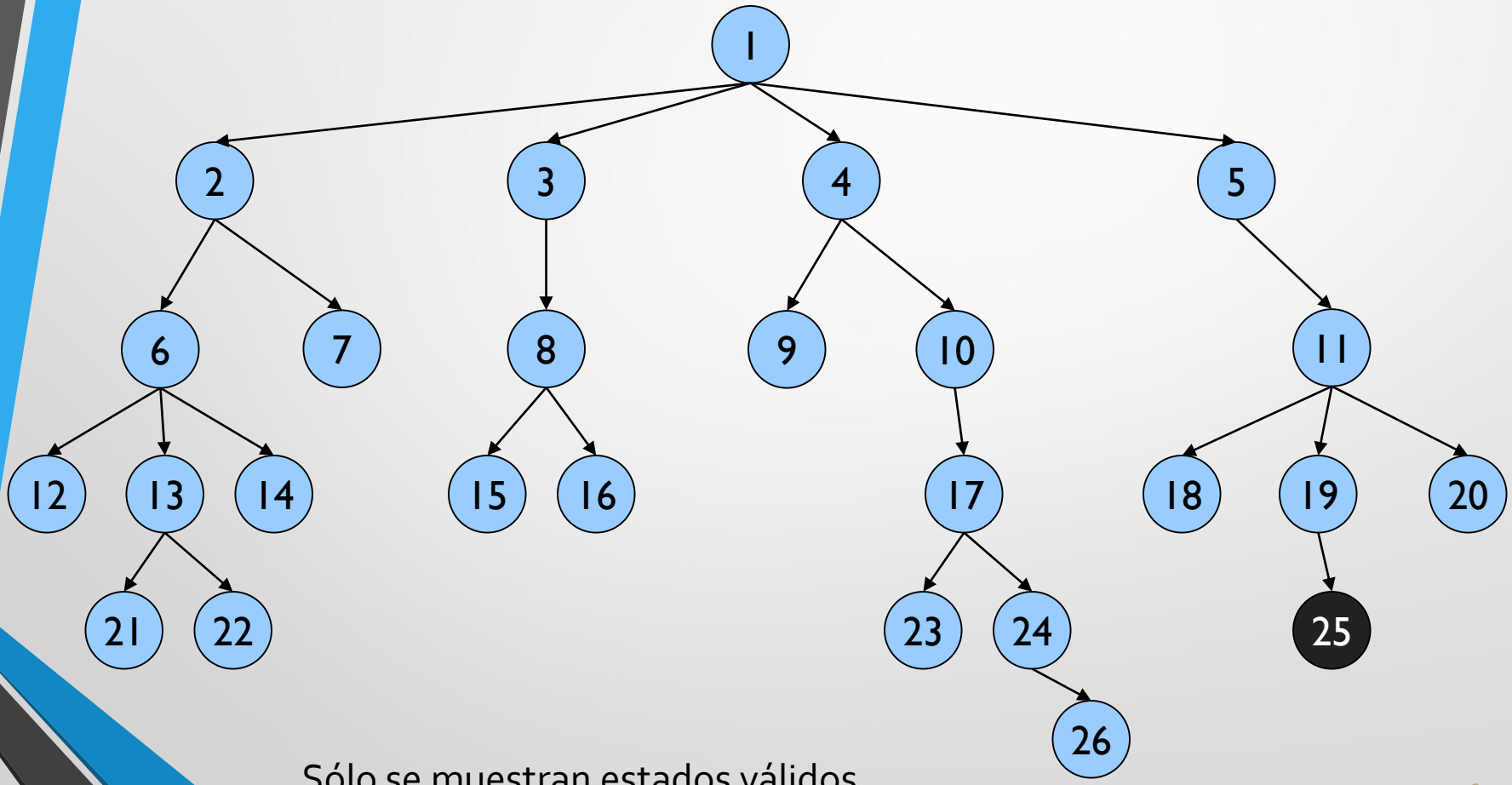
En Ramificación y poda se generan todos los nodos hijos del actual antes de pasar a un nuevo estado → se hace un recorrido en anchura.

Exploración árbol de estados en profundidad (backtracking)



Sólo se muestran estados válidos

Exploración del árbol de estados en anchura



Sólo se muestran estados válidos

Esquema recorrido en anchura primera solución

```
public void realizarAnchura(Estado e)
{
    Cola cola= new Cola();  // Cola FIFO (1o en entrar --> 1o en salir)
    boolean haySolucion= false; // Para buscar la primera solución
    Estado actual;      // Estado actual

    cola.insertar(e);   // mete estado e en la cola
    while (!cola.esVacia() && !haySolucion)
    {
        actual= cola.extraer();
        // Examinar todos los hijos del estado actual
        for (Estado estadoHijo : actual.expandir())
        {
            if (estadoHijo.esSolucion())
                haySolucion= true;
            else
                cola.insertar(estadoHijo);
        }
    }
}
```

Clase Estado

```
public abstract class Estado
{

    public abstract ArrayList<Estado> expandir();
    public abstract boolean esSolucion();
     […]
}
```

Clase Cola

```
public class Cola
{
    public void insertar(Estado nodo) {
        […]
    }

    public boolean esVacia() {
       […]
    }

    public Estado extraer() {
       […]
    }

}
```

Esquema recorrido en anchura todas las soluciones

```
public void realizarAnchuraTodasSoluciones(Estado e)
{
    Cola cola= new Cola();  // Cola FIFO (1o en entrar --> 1o en salir)
    Estado actual;      // Estado actual

    cola.insertar(e);   // mete estado e en la cola
    while (!cola.esVacia())
    {
        actual= cola.extraer();
        // Examinar todos los hijos del estado actual
        for (Estado estadoHijo : actual.expandir())
        {
            if (estadoHijo.esSolucion())
                System.out.println(estadoHijo);
            else
                cola.insertar(estadoHijo);
        }
    }
}
```

Ventajas del recorrido en anchura

Si estamos en el caso de buscar sólo la primera solución y hay estados solución cerca de la raíz.

En backtracking (profundidad) es posible que nos metamos por una rama con muchos nodos que no llegue a ninguna solución o incluso que no tenga fin (infinita), esto con el recorrido en anchura no pasa nunca.

Problema de la asignación de tareas a agentes. Resolución mediante ramifica y poda

- Exploraremos el árbol cuyos estados corresponden a las asignaciones parciales.
- En la raíz no hay ninguna asignación realizada.
- En cada nivel se determina la asignación de un agente más.

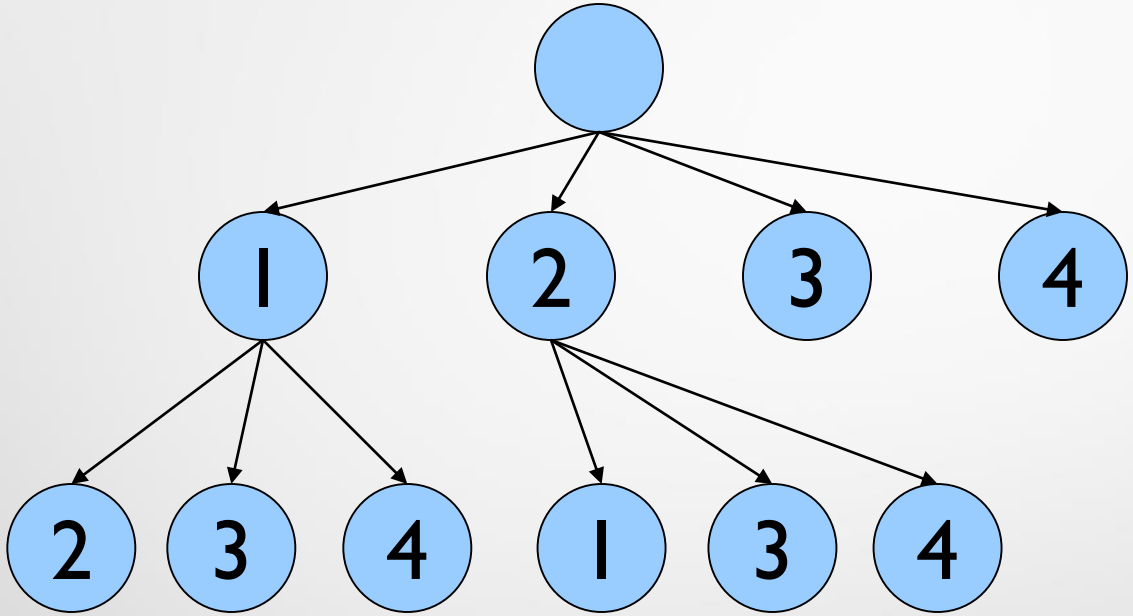
Problema de la asignación de tareas a agentes.

Árbol de estados en anchura

Asigna. a

Asigna. b

Asigna. c, d



Funciones de ramificación

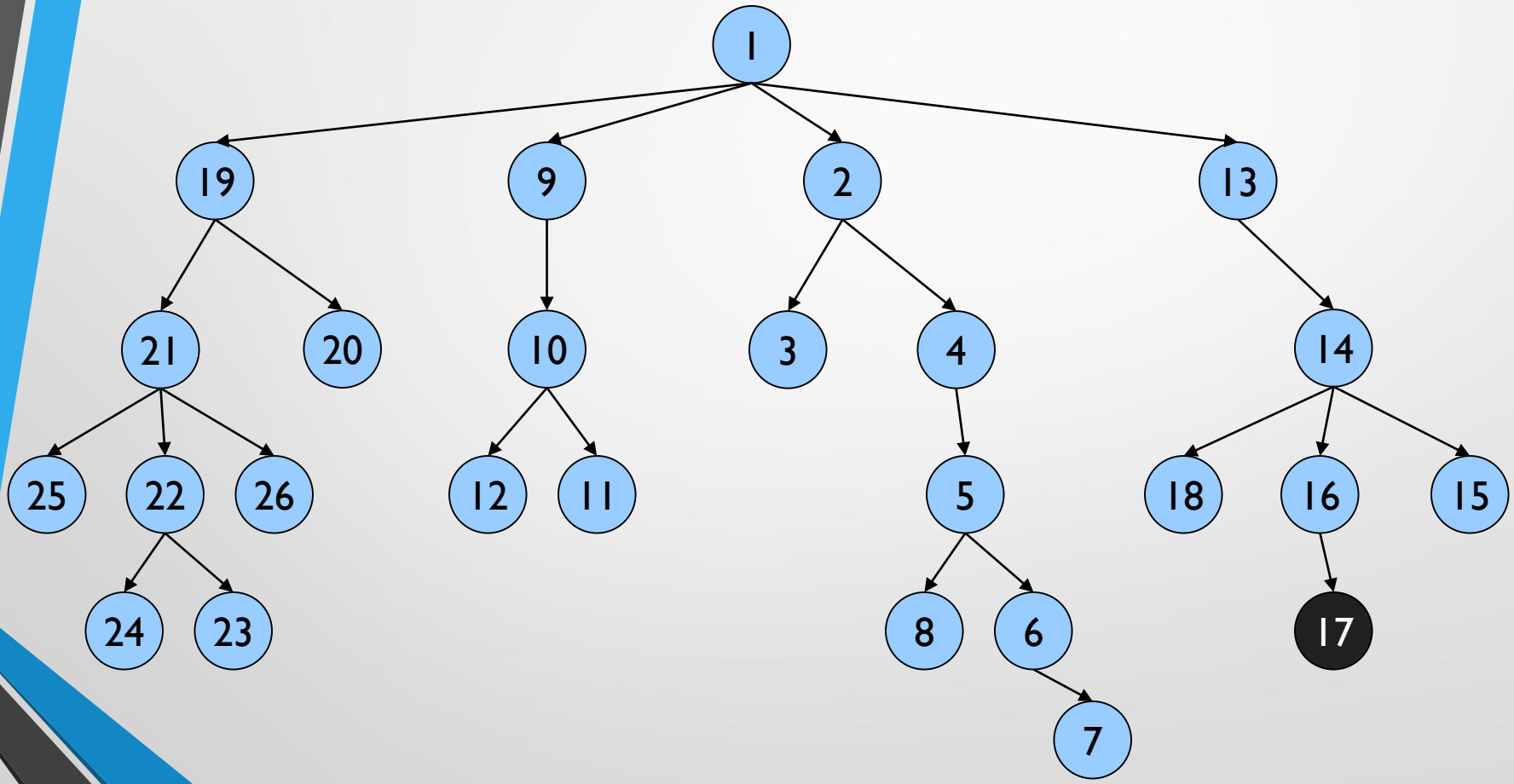
Los algoritmos de ramificación consisten en ir **desarrollando los nodos según el orden** que indique una **función heurística** de ramificación.

En cada momento se coge como **nodo a desarrollar** el **mejor**, que puede estar en cualquier nivel del árbol.

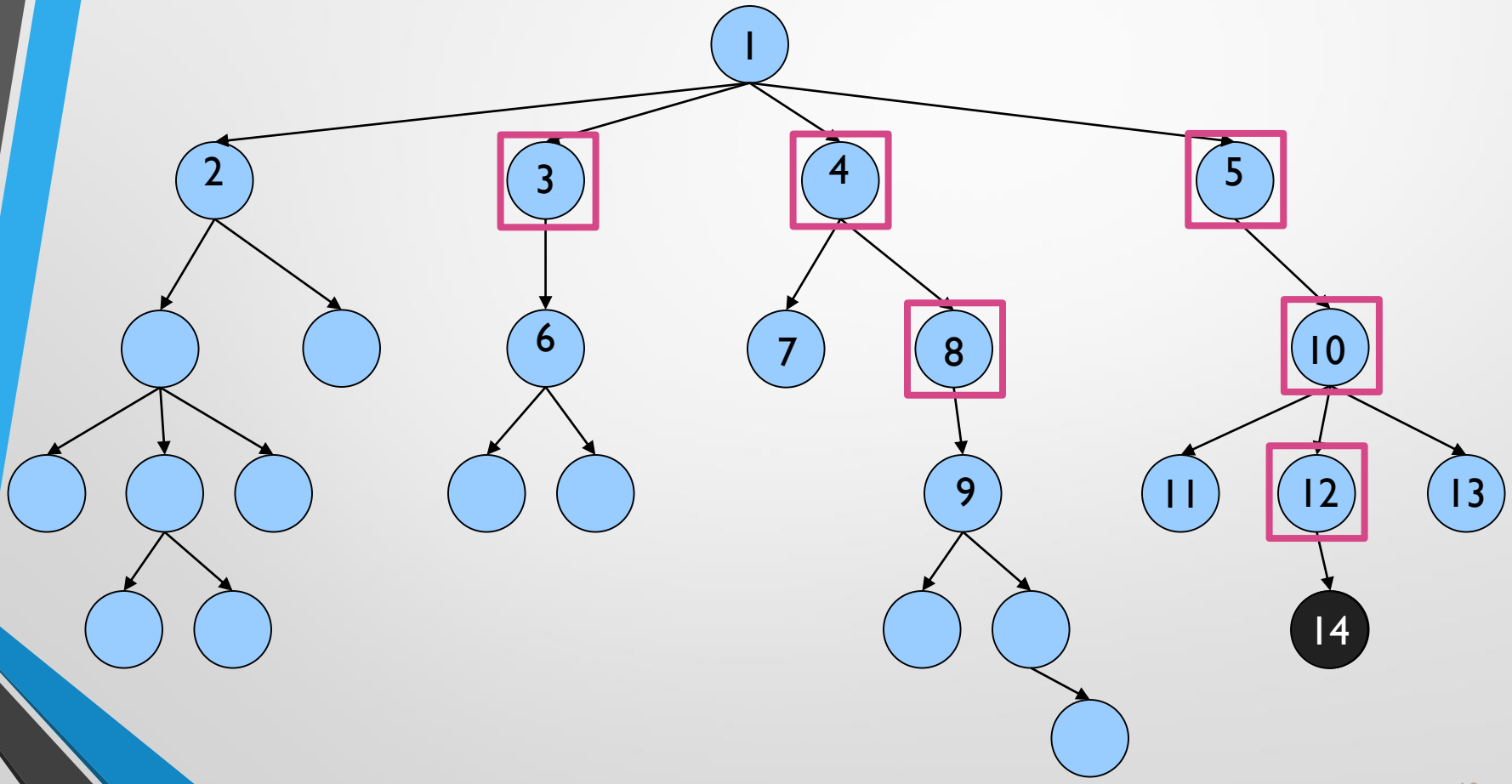
Los algoritmos con ramificación tienen sentido cuando se trata de obtener la **primera solución** y salir.

Si hay que desarrollar todo el árbol es mejor el backtracking.

Exploración árbol de estados en profundidad (heurístico)



Exploración del árbol estados funciones de ramificación



Estructura de datos ramificación

- El esquema general es igual al visto para la exploración en anchura, cambiando la cola FIFO por una cola de prioridad, para almacenar los estados "activos" por desarrollar.
- Cola de prioridad → montículo.
- Los estados quedan ordenados según el valor de su heurístico

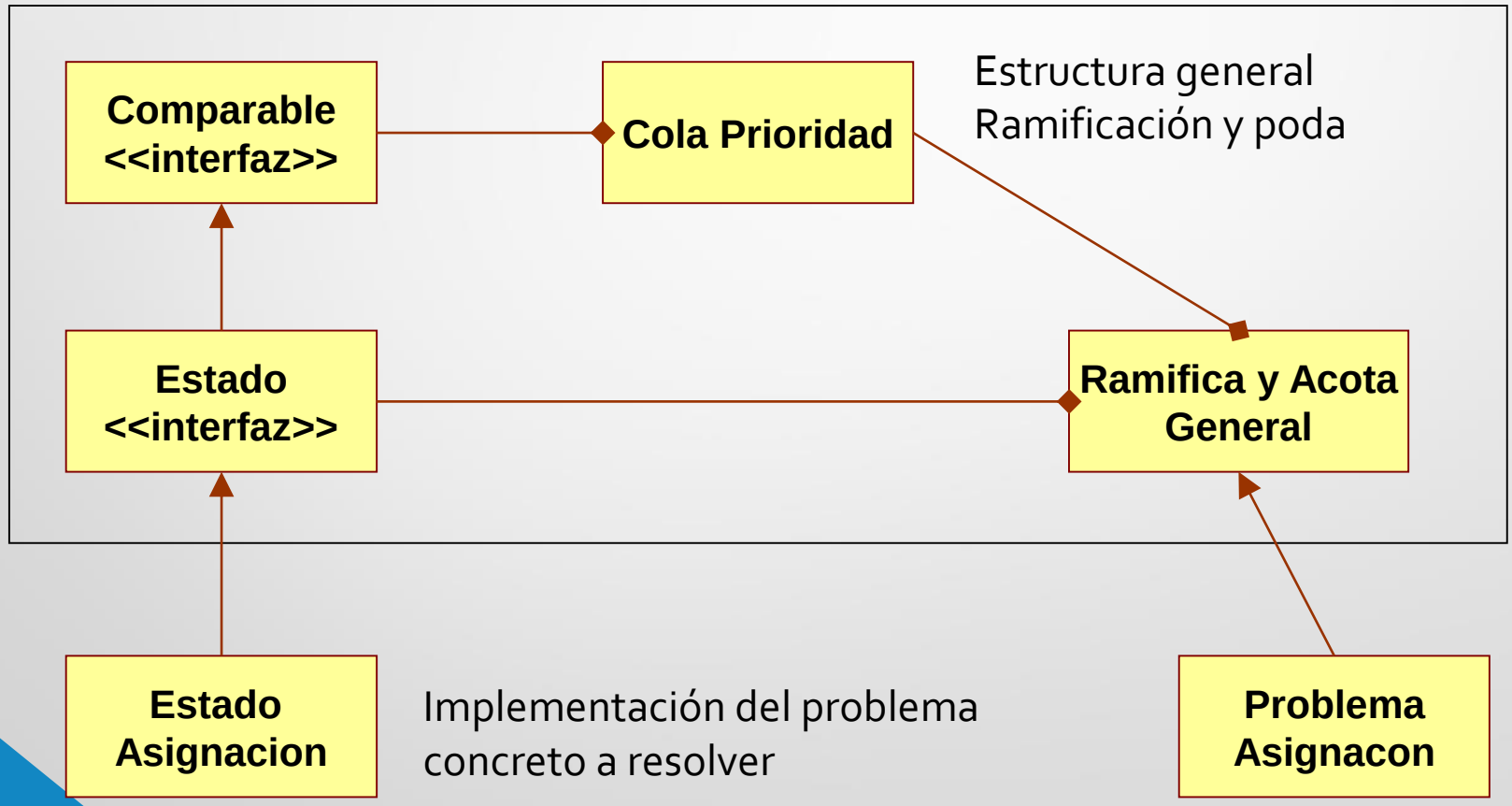
Estado ramificación

```
public abstract class Estado implements Comparable
{
    protected int valorHeuristico; //valor del heurico calculado

    […]
    // Incluimos todo el código que ya teníamos para la clase Estado

    public abstract void calcularValorHeuristico();

    public int getHeuristico() {
        return valorHeuristico;
    }

    @Override
    // Se utiliza en la cola de prioridad para comparar nodos
    public int compareTo(Nodo nodo2)
    {
        if (valorHeuristico>nodo2.valorHeuristico)
            return 1; //this tiene más prioridad
        else if (valorHeuristico == nodo2.valorHeuristico)
            return 0; //los dos tienen la misma prioridad
        else return -1; //this tiene menos prioridad
    }

}
```

Esquema recorrido en anchura con ramificación

```
public void realizarRamificacion(Estado e)
{
    ColaPrioridad cola= new MonticuloBinario();
    boolean haySolucion= false; // Para buscar la primera solución
    Estado actual;      // Estado actual

    cola.insertar(e);   // mete estado e en la cola
    while (!cola.esVacia() && !haySolucion)
    {
        actual= cola.extraer();
        // Examinar todos los hijos del estado actual
        for (Estado estadoHijo : actual.expandir())
        {
            if (estadoHijo.esSolucion())
                haySolucion= true;
            else
                cola.insertar(estadoHijo);
        }
    }
}
```

Diagrama de clases Ramificación y Poda



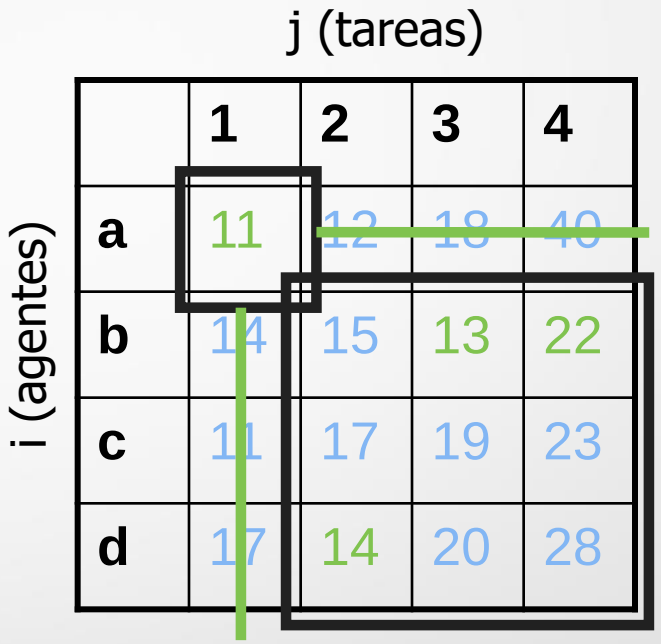
Problema de la asignación de tareas a agentes.

Heurístico de ramificación

Cálculo: La suma de lo ya asignado, más el caso mejor (mínimo por columnas) de lo que queda por asignar.

Ejemplo: al estado a→1, le corresponde un coste de 11+14+13+22=60.

Empleo del heurístico: Explorar el estado del árbol que menor valor del heurístico presente.



Poda o acotación

A cualquier algoritmo de desarrollo del árbol de estados de los ya vistos (profundidad, anchura, ramifica), se les puede añadir una función heurística de poda.

La función de poda impide el desarrollo de ciertos estados porque no aportan nada para encontrar la solución al problema:

- No conducen a ninguna solución, o
- No conducen a soluciones mejores, o
- Suponen desarrollos repetidos.

Consiste en una cota que permite determinar si podamos el estado o no.

Poda en algoritmos precedentes

La poda que habíamos visto hasta ahora es trivial:

- Estados no válidos
- Los estados hoja no tenían más hijos posibles
- En el caso de todas las soluciones, si un estado solución no tenía más soluciones por debajo

Se pretende realizar una poda más fuerte, podando estados que puedan tener por debajo incluso otros muchos estados.

Esquema Ramificación y poda mejor solución

```
public void realizarRamificacionPoda(Estado estadoInicial) {
    ColaPrioridad cola= new Monticulo();    // Cola prioridad
    Estado actual;      // Estado actual
    Estado mejorSolucion;
    int cotaPoda= nodoRaiz.valorInicialPoda();

    cola.insertar(estadoInicial);   // mete estado e en la cola
    while (!cola.esVacia() && cola.estimacionMejor() <cotaPoda) {
        actual= cola.extraer();
        // Examinar todos los hijos del estado actual
        for (Estado estadoHijo : actual.expandir())  {
            // Comprueba que no hay que podar
            if (estadoHijo.getHeuristico() < cotaPoda)
                if (estadoHijo.esSolucion()) {
                        mejorSolucion = estadoHijo;
                        cotaPoda= estadoHijo.getHeuristico();
                    }
                else
                    cola.insertar(estadoHijo);
        } // for
    } // while
}
```

Cálculo del heurístico de poda

- Este tipo de heurístico se aplica muchas veces en problemas de optimización
- La poda se realiza en nodos a partir de los cuales se generan soluciones peores que la encontrada hasta el momento.
  - Cota: mejor solución hasta el momento

Cota de poda inicial

- El anterior planteamiento tiene el inconveniente de no poder realizar podas hasta que se haya encontrado la primera solución.
- Podemos establecer una primera cota, antes de empezar a desarrollar el árbol
- Esto se realiza cuando es fácil generar una solución aleatoria.

Problema de la asignación de tareas a agentes.

Cota de poda inicial

Cálculo: Inicialmente, la cota de poda es la menor de la sumas de las dos diagonales de la matriz de costes.

Empleo: Se poda todo estado que presente un valor calculado del heurístico de ramificación, mayor o igual que la cota de poda.

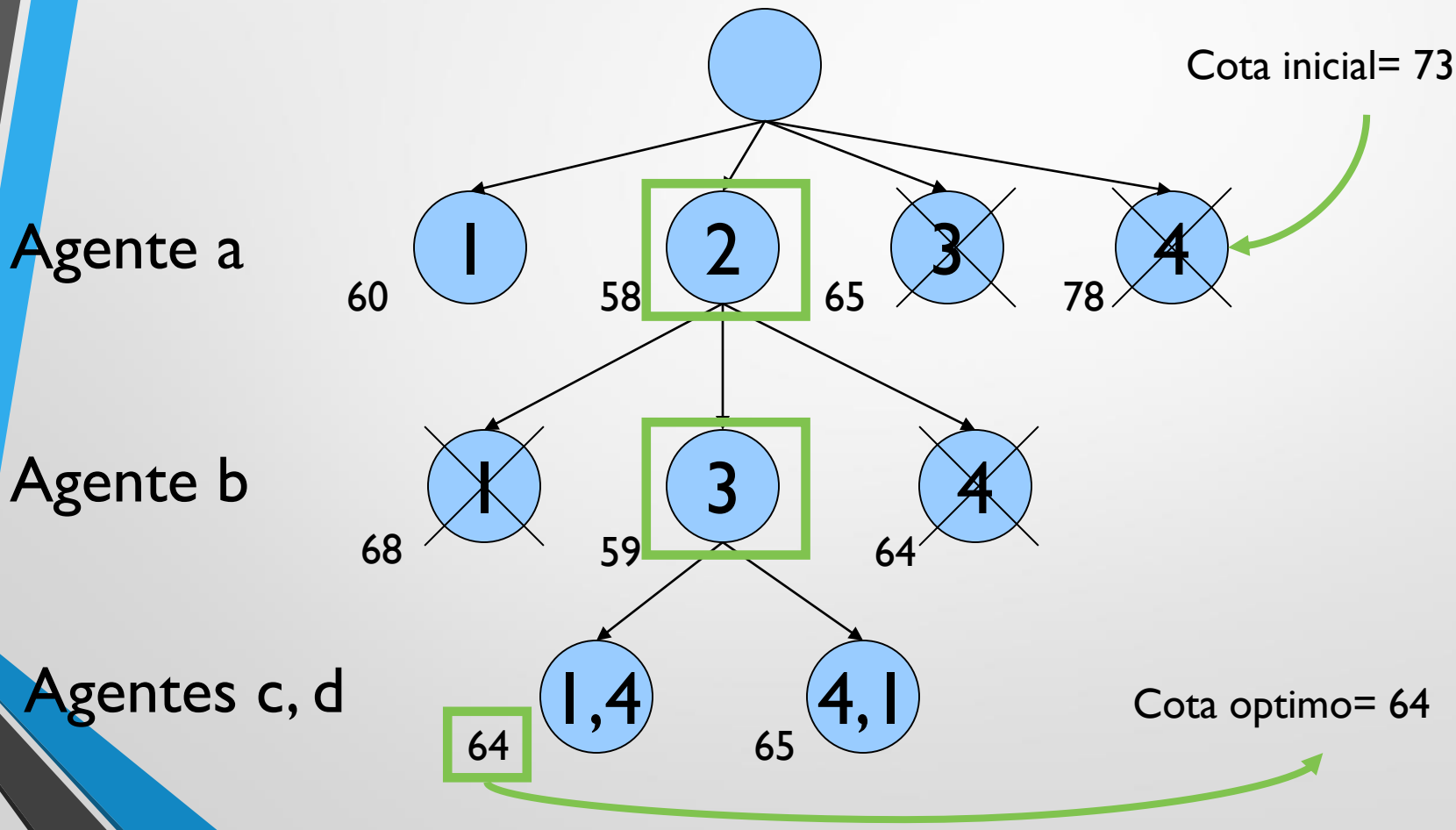
Cambio: La cota cambia cuando encontramos una mejor solución.

j (tareas)

i (agentes)	1	2	3	4
a	**11**	12	18	**40**
b	14	**15**	**13**	22
c	11	**17**	**19**	23
d	**17**	14	20	**28**

- Valor cota inicial: min(73,87)=73.

Resolución mediante ramifica y poda de la Asignación (IV)

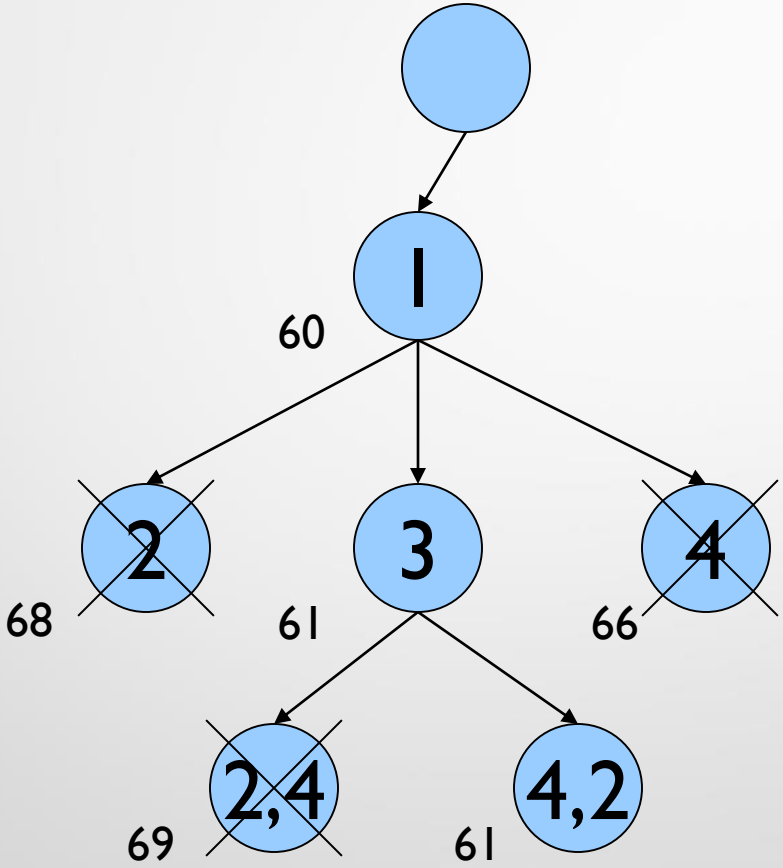


Resolución mediante ramifica y poda de la Asignación (V)

Agente a

Agente b

Agentes c, d



Cota óptimo 64

Resolución mediante ramifica y poda de la Asignación (VI)

- Solución óptima:
  - (a, 1), (b, 3), (c, 4), (d, 2)
- Sólo hemos desarrollado 4 estados finales (6 si contamos las 2 diagonales para inicializar el umbral de poda), en lugar de las 4!=24 que implica el desarrollo por backtracking.
- Este algoritmo es mucho más eficiente en la búsqueda de soluciones óptimas.

Estados repetidos

- Un caso particular de poda sería la que permite evitar el desarrollo de estados repetidos.
- Estructura para los estados que ya aparecieron, de forma que la poda comprobaría si el estado está o no está en dicha estructura.
- En la práctica, esto presenta una serie de problemas:
  - Memoria, para albergar todos los estados diferentes ya aparecidos.
  - Tiempo, para buscar si el estado actual pertenece ya a la estructura.

Ejemplos de Ramificación y poda

Problema de las n reinas (I)

- Problema ya resuelto con la técnica de backtracking.
- Generamos sólo los estados en los que las reinas no se coman.
- Heurístico de ramificación:
  - Desarrollar el estado que suponga tener más reinas colocadas.
  - En caso de empate, aquel que coloque una reina en una casilla cuya longitud de la diagonal más larga sea la menor.

Problema de las n reinas (II)

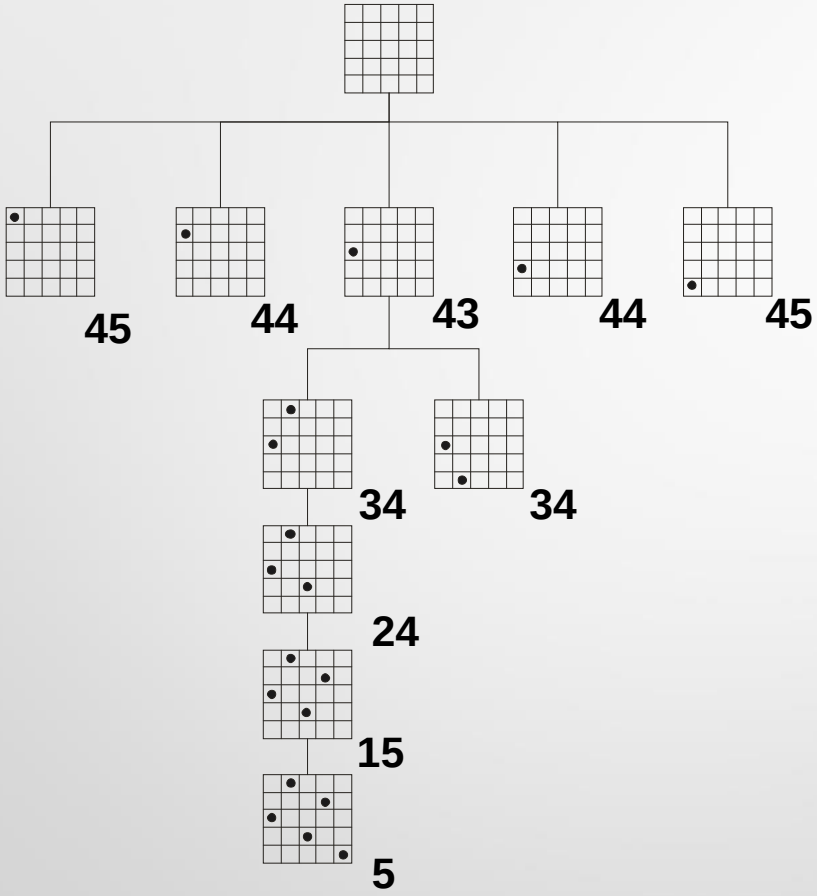
- *diagonal(i,j)*, longitud de la diagonal más larga que pasa por la casilla (i,j).
- La tabla de la derecha refleja los valores de esta función en cada casilla.

5	4	3	4	5
4	5	4	5	4
3	4	5	4	3
4	5	4	5	4
5	4	3	4	5

Formula para asignar valores numéricos a cada estado

- Buscamos una función que a menor valor indique un estado mejor para desarrollar.
  - Estado con más reinas colocadas:
    - *n – nº reinas colocadas*
  - En caso de empate, reina en una casilla *diagonal(i,j)* sea la menor.
- Para fusionar los términos:
  - (*n – nº reinas colocadas*) * 10 + *diagonal(i,j)*

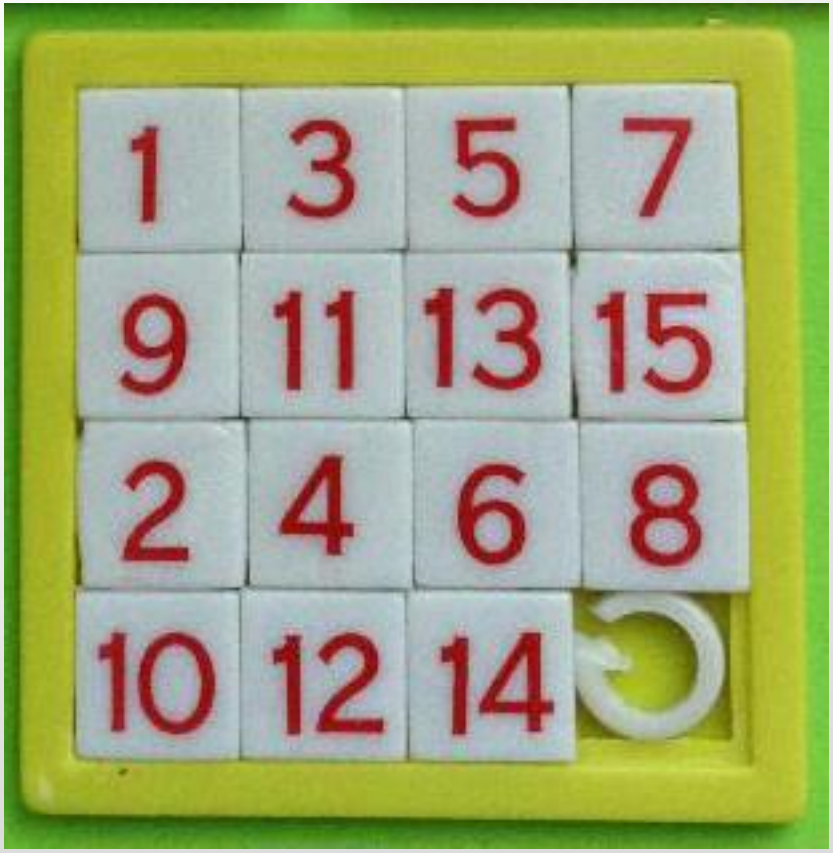
Problema de las n reinas (III)



- En cada momento los estados "vivos" son las hojas del árbol generadas por el heurístico descrito y almacenadas en la cola de prioridad.

Problema del puzle (I)

- Este problema se desarrolla sobre un tablero de 16 posiciones, donde hay colocadas 15 fichas, quedando una posición vacía.
- Las fichas no se pueden levantar del tablero, por lo que sólo es posible su movimiento por medio de deslizamientos sobre el mismo.



Problema del puzle (II)

- El objetivo del juego es, a partir de un estado inicial (fichas desordenadas), obtener el estado final ordenado, por medio de una serie de movimientos legales.
- Movimientos válidos son aquellos en que una ficha adyacente a la posición libre, se mueve hacia ésta. Se pueden ver estos movimientos permitidos como un desplazamiento de la casilla vacía hacia una de sus cuatro vecinas.

Problema del puzle (III)

- Estado alcanzable $\leftrightarrow$ hay una secuencia de movimientos legales desde el estado inicial hasta este estado.
- El espacio de estados de un estado inicial, estará formado por todos los estados alcanzables desde ese estado.
- Forma más directa de resolver este problema: buscar el estado objetivo dentro del espacio de estados, buscando un camino desde el inicial al final.

Problema del puzle (IV)

- El espacio de estados completo está compuesto por 16! estados, sería computacionalmente inaceptable el coste de crear y recorrer ese árbol.
- En realidad, solo la mitad de esos estados son alcanzables a partir de un estado inicial dado, pero aún así el número de estados sigue siendo muy grande.
- Tenemos que ser capaces de determinar si a partir de un cierto estado, se puede alcanzar un estado objetivo o no.

Problema del puzle (V)

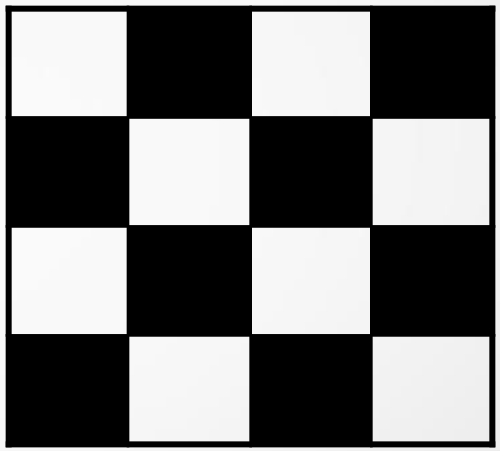
- Al problema del gran número de estados posible se añade otro.
- Generación de estados repetidos y formación de ciclos sin fin.
  - Se pueden evitar hijos repetidos directos de forma sencilla
  - Pero el tratamiento de las repeticiones por ciclos largos tiene que ser siguiendo el procedimiento citado anteriormente.

Heurísticos para el problema del puzle

- Heurísticos de ejemplo para ir desarrollando estados:
  - Número de fichas colocadas en la situación final. A mayor valor, mejor estado para desarrollar.
  - $\sum_{i=1}^{16} \text{distancia } (i)$, donde distancia(i) es el número de movimientos necesarios para colocar la ficha i en la posición i.

Problema del puzle (V)

- Numerando las casillas de 1 a 16 definimos:
  - posición(i): posición en el estado inicial de la ficha número i. posición(16) será la posición en el tablero de la casilla vacía.
  - menor(i): es el número de fichas j tales que j<i y posición(j)>posicion(i)



$$\sum_{i=1}^{16} menor(i) + x$$

- El estado objetivo será alcanzable desde un estado si y sólo si Sumatorio(menor(i))+x es par.
  - Donde x es 1 si la casilla vacía se encuentra en una de las casillas sombreadas del tablero adjunto y 0 si no es así.

Ejemplo aplicación heurístico (I)

menor(1)=0	menor(9)=0
menor(2)=0	menor(10)=0
menor(3)=1	menor(11)=3
menor(4)=1	menor(12)=6
menor(5)=0	menor(13)=0
menor(6)=0	menor(14)=4
menor(7)=1	menor(15)=11
menor(8)=0	menor(16)=10

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

$$\sum menor(i) + x = 37 \quad (x = 0)$$

- Por tanto, el estado no permite alcanzar el estado objetivo.

Ejemplo aplicación heurístico (II)

menor(1)=0	menor(9)=0
menor(2)=0	menor(10)=0
menor(3)=0	menor(11)=0
menor(4)=0	menor(12)=0
menor(5)=0	menor(13)=1
menor(6)=0	menor(14)=1
menor(7)=0	menor(15)=1
menor(8)=0	menor(16)=5

1	2	3	4
5	6	7	8
9	10		11
13	14	15	12

$$\sum menor(i) + x = 8 \quad (\mathrm{x} = 0)$$

- Por tanto, el estado permite alcanzar el estado objetivo.

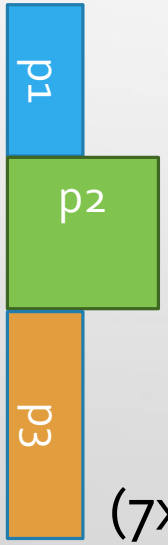
Problema propuesto: minimizar área rectángulos

- Tenemos n piezas rectangulares planas $p_1, p_2, \ldots, p_n$, cada una con un área (a_i, b_i) donde $1 \leq i \leq n$.
- Queremos colocar las piezas en un tablero
- El problema consisten en encontrar la disposición en la que las n piezas ocupen la menor área rectangular posible. Esta área se calculará como el producto del máximo alto ocupado por el máximo ancho ocupado.

Un ejemplo de piezas

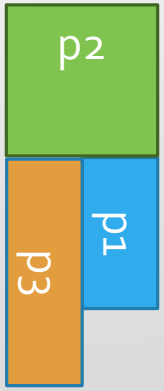
- Tres piezas: p1= (1,2), p2= (2,2), p3= (1,3)
- Disposiciones:

(4x3)= 12



(7x2)= 14

(5x2)= 10



(3x3)= 9