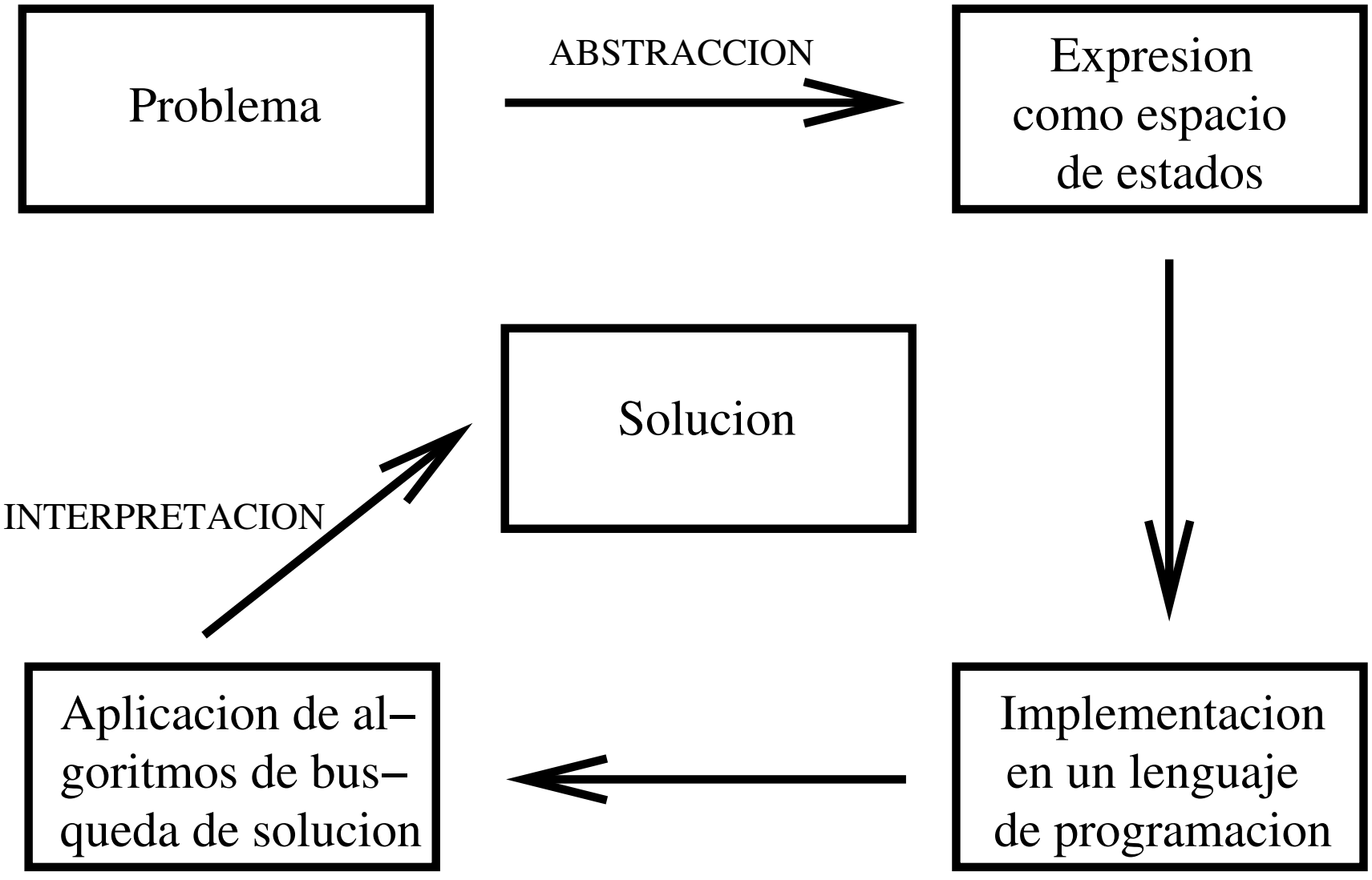
Tema 3: Búsqueda

José Luis Ruiz Reina
José Antonio Alonso
Franciso J. Martín Mateos
Miguel A. Gutiérrez Naranjo

Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Inteligencia Artificial, 2016-17

Método de solución de problemas



Definición de un problema como espacio de estados

- Paso previo a la búsqueda de soluciones de un problema:
  - Especificación del problema
- Elementos del problema:
  - ¿Cuál la situación *inicial* desde la que se parte?
  - ¿Cuál es el *objetivo final*?
  - ¿Cómo describir las diferentes situaciones o *estados* por los que podríamos pasar?
  - ¿Qué pasos elementales o *acciones* para cambiar de estado y cómo actúan?
- Especificar un problema como *espacio de estados* consiste en describir de manera clara cada de uno de estos componentes
  - Ventaja: procedimientos generales de búsqueda de soluciones
  - *Independientes* del problema

Planteamiento del problema del 8-puzle

- Un tablero cuadrado (3x3) en el que hay situados 8 bloques cuadrados numerados (con lo cual se deja un hueco del tamaño de un bloque). Un bloque adyacente al hueco puede deslizarse hacia él. El juego consiste en transformar una posición inicial en la posición final mediante el deslizamiento de los bloques. En particular, consideramos el estado inicial y final siguientes:

2	8	3
1	6	4
7		5

Estado inicial

1	2	3
8		4
7	6	5

Estado final

Representación de estados

- Estado: descripción de una posible situación en el problema
  - Abstracción de propiedades
- Importancia de una buena representación de los estados
  - Sólo considerar información relevante para el problema
  - La representación escogida influye en el número de estados y éste en los procedimientos de búsqueda de soluciones
- Ejemplo: 8-puzle: Elementos de la representación:
  - relevante: localización de cada bloque y del hueco;
  - irrelevante: tipo de material de los bloques, colores de los bloques,...

Representación de estados

- Ejemplo del 8-puzle: Representaciones del estado

2	8	3
1	6	4
7		5

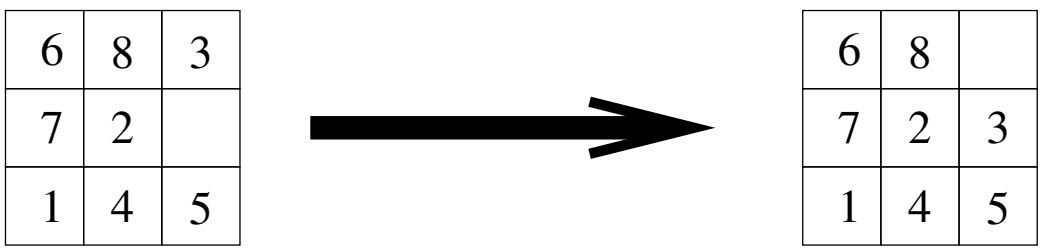
  - Descripción de la posición exacta de cada uno de los bloques
- Representación *vs.* implementación
  - Tuplas: `(2,8,3,1,6,4,7,"H",5)`, `(2,8,3,4,5,"H",7,1,6)`
  - Listas anidadas: `[[2,8,3],[1,6,4],[7,"H",5]]`
  - Diccionarios: `{"primeraizquierda":2, "primeracentro":8, ...}`
- Número de estados: 9! = 362.880.

Acciones

- Acciones:
  - Representan un conjunto finito de operadores básicos que transforman unos estados en otros
- Elementos que describen una acción
  - Aplicabilidad: precondición y postcondición
  - Estado resultante de la aplicación de una acción (aplicable) a un estado
- Criterio para elegir acciones.
  - Depende de la representación de los estados
  - Preferencia por representaciones con menor número de acciones
- Ejemplo: Acciones del 8-puzle:
  - Según los movimientos de los bloques: 32 = (8 bl. $\times$ 4 mov.)
  - Según los movimientos del hueco: 4.

Acciones

- Acciones en el 8 puzle
  - Mover el hueco hacia arriba
  - Mover el hueco hacia abajo
  - Mover el hueco hacia la derecha
  - Mover el hueco hacia la izquierda
- Descripción de la acción “Mover el hueco hacia arriba”
  - Aplicabilidad: es aplicable a estados que no tengan el hueco en la primera fila
  - Resultado de aplicarlo: intercambiar las posiciones del hueco y del bloque que está encima de éste



- Los tres restantes, análogamente

Estado inicial

- Estado inicial
  - Un estado que describe la situación de partida
- Estado inicial en el ejemplo del 8-puzle

2	8	3
1	6	4
7		5

Estados finales

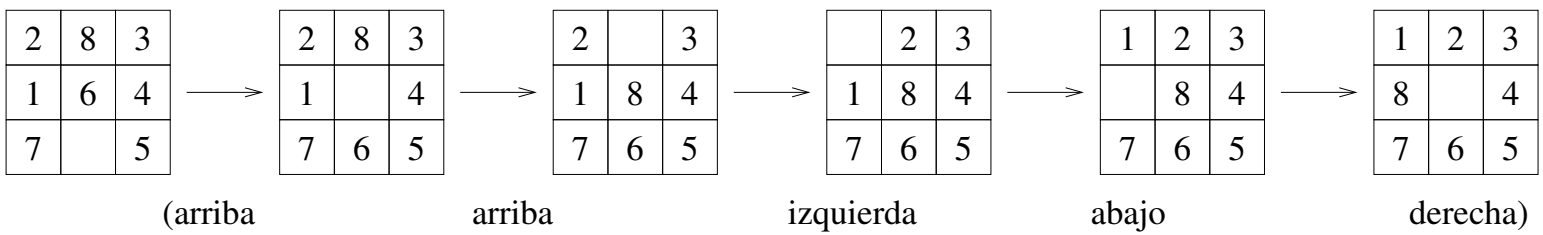
- Descripción del objetivo
  - Usualmente, un conjunto de estados, que llamaremos *finales*
  - A veces, aunque no necesariamente, un único estado final
- Ejemplo del 8-puzle (un único estado final)

1	2	3
8		4
7	6	5

- Formas de describir los estados finales:
  - Enumerativa.
  - Declarativa.

Soluciones de un problema

- Definición de solución de un problema.
  - Secuencia de acciones a realizar para conseguir el objetivo
  - Secuencia de acciones cuya aplicación desde el estado inicial obtiene un estado final
- Ejemplo: Una solución del 8-puzle:

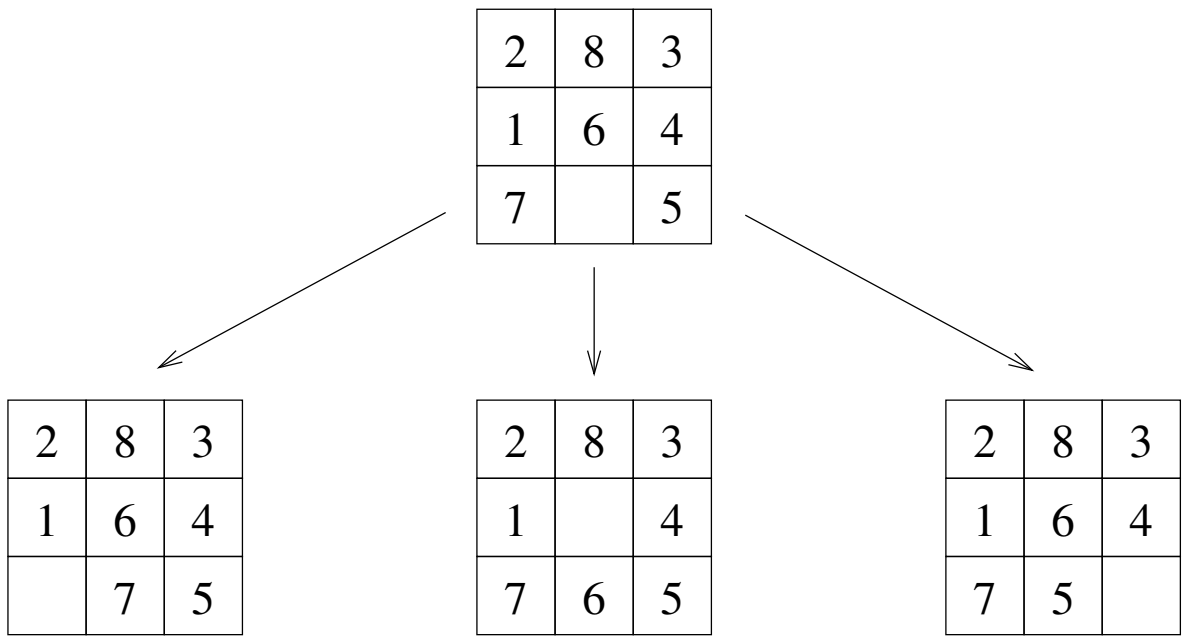


Soluciones de un problema

- Tipos de problemas:
  - Buscar una solución.
  - Determinar si existe solución y encontrar un estado final.
  - Buscar cualquier solución lo más rápidamente posible.
  - Buscar todas las soluciones.
  - Buscar la solución más corta.
  - Buscar la solución menos costosa.

Espacio de estados como un grafo

- Un espacio de estados se puede ver como un grafo dirigido
  - Los vértices de dicho grafo son los estados
  - Sucesores de un estado: aquellos obtenidos a partir del estado aplicando una acción aplicable
- Ejemplo en el 8-puzle



Elementos para la implementación

- Elección de una representación (estructura de datos):
  - para los estados
  - para las acciones
- La implementación de un problema como espacio de estados consta de:
  - Una variable **`*ESTADO-INICIAL*`**
    - Almacena la representación del estado inicial
  - Una función **`ES-ESTADO-FINAL(ESTADO)`**
    - Comprueba si un estado es final o no
  - Una función **`ACCIONES(ESTADO)`**.
    - Devuelve las acciones aplicables a un estado dado
  - Una función **`APLICA(ACCIÓN,ESTADO)`**
    - Obtiene el estado resultante de aplicar una acción a un estado (se supone que la acción es aplicable al estado)

Planteamiento del problema del granjero

- Enunciado:
  - Un granjero está con un lobo, una cabra y una col en una orilla de un río.
  - Desea pasarlos a la otra orilla.
  - Dispone de una barca en la que sólo puede llevar una cosa cada vez.
  - El lobo se come a la cabra si no está el granjero.
  - La cabra se come la col si no está el granjero.
- Información de los estados: orilla en la que está cada elemento
  - La orilla de la barca es redundante (está en la orilla donde esté el granjero)

Formulación del problema del granjero

- Representación de estados: (x y z u) en $\{i,d\}^4$.
  - Número de estados: 16.
- Estado inicial: (i i i i).
- Estado final (único): (d d d d).
- Acciones:
  - Pasa el granjero solo.
  - Pasa el granjero con el lobo.
  - Pasa el granjero con la cabra.
  - Pasa el granjero con la col.

Formulación del problema del granjero

- Aplicabilidad de las acciones
  - Precondición (en los tres últimos): los dos elementos que pasan han de estar en la misma orilla
  - Poscondición: en el estado resultante no deben estar el lobo y la cabra, o la cabra y la col, en la misma orilla sin que el granjero esté en esa misma orilla
- Estado resultante de aplicar la acción
  - Cambiar la orilla de los elementos que pasan por la orilla opuesta

Planteamiento del problema de las jarras

- Enunciado:
  - Se tienen dos jarras, de 4 y 3 litros respectivamente.
  - Ninguna de ellas tiene marcas de medición.
  - Se tiene una bomba que permite llenar las jarras de agua.
  - Averiguar cómo se puede lograr tener exactamente 2 litros de agua en la jarra de 4 litros de capacidad.
- Representación de estados: (x y) con x en {0,1,2,3,4} e y en {0,1,2,3}.
- Número de estados: 20.

Formulación del problema de las jarras

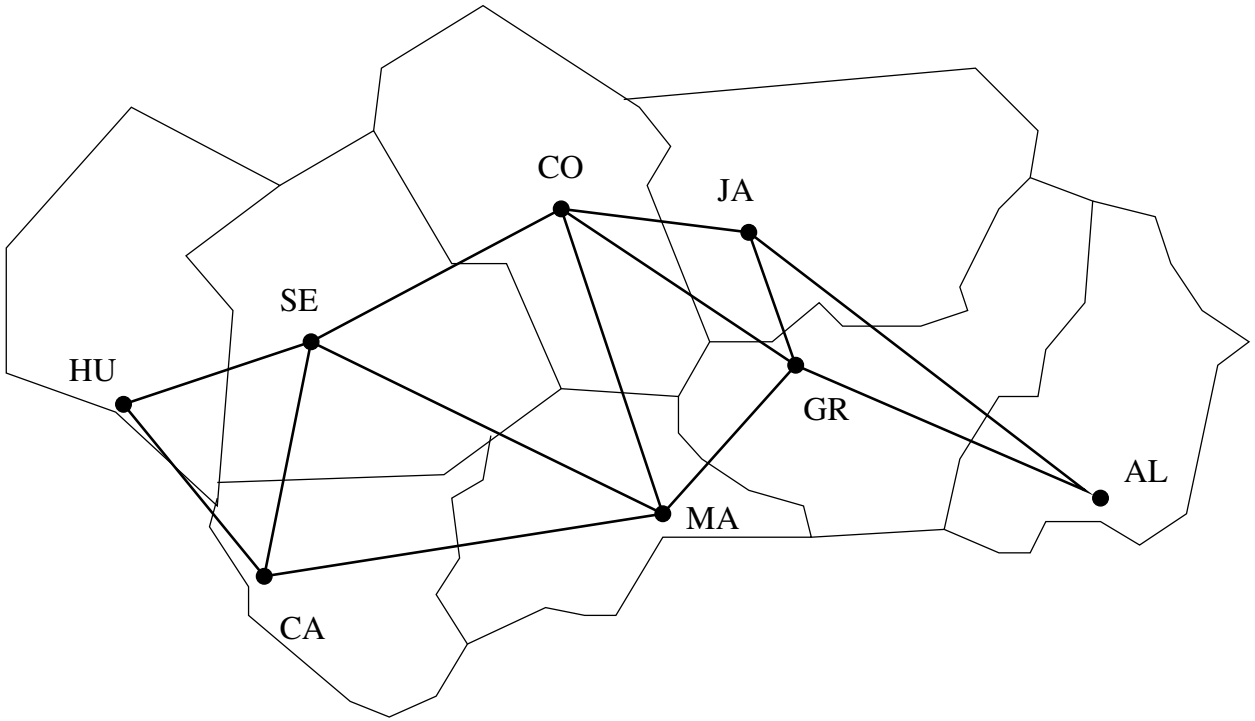
- Estado inicial: (0 0).
- Estados finales: todos los estados de la forma (2 _).
- Acciones:
  - Llenar la jarra de 4 litros con la bomba.
  - Llenar la jarra de 3 litros con la bomba.
  - Vaciar la jarra de 4 litros en el suelo.
  - Vaciar la jarra de 3 litros en el suelo.
  - Llenar la jarra de 4 litros con la jarra de 3 litros.
  - Llenar la jarra de 3 litros con la jarra de 4 litros.
  - Vaciar la jarra de 3 litros en la jarra de 4 litros.
  - Vaciar la jarra de 4 litros en la jarra de 3 litros.

Planteamiento del problema de las jarras

- Aplicación de acciones a un estado (x y)
- Acción “Llenar jarra de 3”
  - Aplicabilidad: y$<$3 (precondición)
  - Estado resultante: (x 3)
- Acción “Llenar jarra de 4 con jarra de 3”
  - Aplicabilidad: x$<$4, y$>$0, x+y$>$4 (precondición)
  - Estado resultante: (4 x+y-4)
- Acción “Vaciar jarra de 3 en jarra de 4”
  - Aplicabilidad: y$>$0, x+y$\leq$4 (precondición)
  - Estado resultante: (x+y 0)
- Análogamente las demás acciones

Planteamiento del problema del viaje

- Enunciado:
  - Nos encontramos en una capital andaluza (p.e. Sevilla).
  - Deseamos ir a otra capital andaluza (p.e. Almería).
  - Los autobuses sólo van de cada capital a sus vecinas.



Formulación del problema del viaje

- 8 posibles estados:
  - Almería, Cádiz, Córdoba, Granada, Huelva, Jaen, Málaga, Sevilla
- Estado inicial: Sevilla.
- Estado final: Almeria.
- Acciones:
  - Ir a Almería, Ir a Cádiz, Ir a Córdoba, Ir a Granada, Ir a Huelva, Ir a Jaén, Ir a Málaga, Ir a Sevilla.
- Ejemplo: aplicación de “Ir a Málaga” a un estado x
  - Aplicabilidad: x debe ser provincia vecina de Málaga
  - Estado resultante: Málaga

Problemas de la vida real

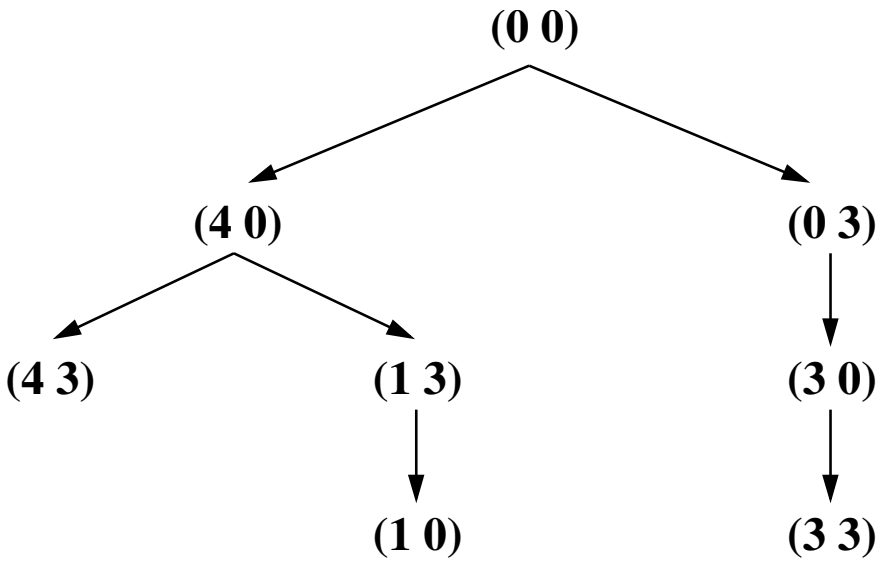
- Problemas de la vida real que se pueden plantear y resolver como espacios de estados:
  - Búsqueda de rutas en redes informáticas
  - Rutas aéreas para viajar
  - Problema del viajante
  - Diseño de microchips
  - Ensamblaje de componentes
  - Desplazamiento de robots

Búsqueda de soluciones en espacios de estados

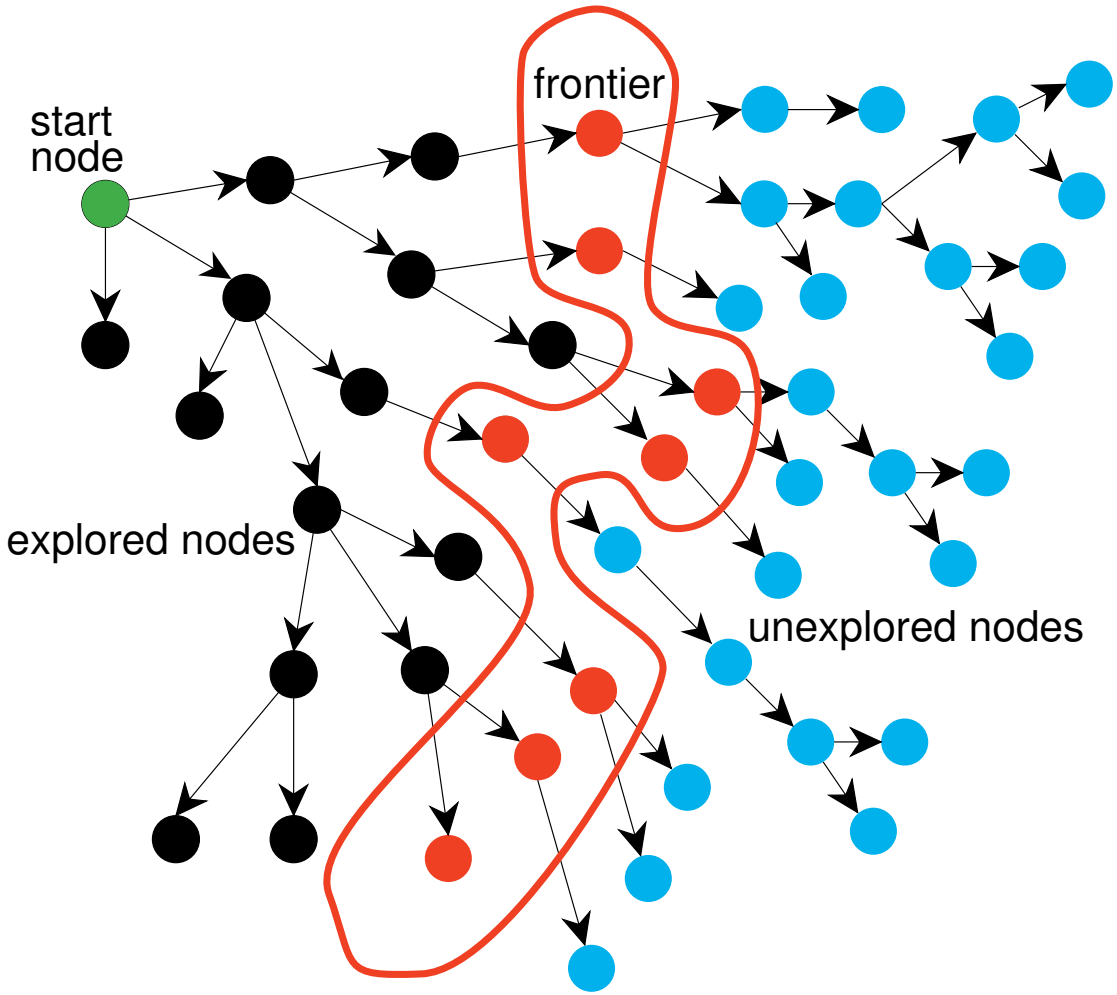
- Objetivo: encontrar una secuencia de acciones que, partiendo del estado inicial, obtenga un estado final
- Idea básica: exploración del *grafo* del espacio de estados
  - En cada momento se analiza un estado actual (en un principio, el inicial)
  - Si el estado actual es final, acabar (recopilando la sucesión de acciones)
  - En caso contrario, obtener los sucesores del estado actual (*expandir*)
  - *Elegir* un nuevo estado actual, *dejando* los restantes para analizarlos posteriormente (si fuera necesario)
  - Repetir el proceso mientras haya estados por analizar
- La elección del estado actual en cada momento determina una *estrategia* de búsqueda

Árboles de búsqueda

- El proceso anterior puede verse como la construcción incremental de un árbol de búsqueda
- Ejemplo en el problema de las jarras:



Árboles de búsqueda, intuición gráfica



Árboles de búsqueda

- Nodo de un árbol de búsqueda, componentes:
  - Estado
  - Nodo padre (puntero)
  - Acción (la que se aplica al padre para obtenerlo)
  - Profundidad (número de acciones desde el nodo raiz)
- Nodo raíz del árbol de búsqueda: el correspondiente al estado inicial
- Nodos hoja del árbol de búsqueda:
  - Nodos cuya expansión no ha producido sucesores nuevos
  - Nodos pendientes de considerar (y expandir en su caso)
- Nótese que es fácil calcular el camino (desde el nodo raíz) hasta el nodo
- Espacio de estados *vs.* árbol de búsqueda:
  - Nodos del árbol de búsqueda: estado + cómo se llega hasta él
  - El árbol se construye incrementalmente y refleja un proceso de búsqueda sobre el grafo del espacio de estados

Recordar: implementación de un problema de espacio de estados

- Elección de una representación (estructura de datos):
  - para los estados
  - para los acciones
- La implementación de un problema como espacio de estados consta de:
  - Una variable **`*ESTADO-INICIAL*`**
    - Almacena la representación del estado inicial
  - Una función **`ES-ESTADO-FINAL(ESTADO)`**
    - Comprueba si un estado es final o no
  - Una función **`ACCIONES(ESTADO)`**.
    - Devuelve las acciones aplicables a un estado dado
  - Una función **`APLICA(ACCIÓN,ESTADO)`**
    - Obtiene el estado resultante de aplicar una acción a un estado (se supone que la acción es aplicable al estado)

Implementación de la búsqueda (funciones auxiliares)

- Nodo de búsqueda: estado + padre + acción + profundidad
  - Funciones de acceso: **`ESTADO(NODO)`**, **`ANTECESOR(NODO)`**, **`ACCIÓN(NODO)`**, **`PROFUNDIDAD(NODO)`**
- Sucesores de un nodo:
  - **`SUCESOR(NODO,ACCIÓN)`** calcula el estado resultado de aplicar **`ACCIÓN`** al estado asociado al **`NODO`** y devuelve un **`NUEVO NODO`**
  - **`SUCESORES(NODO)`** toma **`SUCESORES`** igual a vacío, y para cada **`ACCIÓN`** aplicable sobre el **`NODO`**, añade a **`SUCESORES`** el resultado de **`SUCESOR(NODO,ACCIÓN)`**

Implementación de un procedimiento general de búsqueda

```
FUNCION BUSQUEDA-GENERAL()
1. Hacer ABIERTOS la "cola" formada por el nodo inicial (es decir,
    el nodo cuyo estado es *ESTADO-INICIAL*);
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario,
             2.4.2.1 Hacer NUEVOS-SUCESORES la lista de nodos
                      de SUCESORES(ACTUAL) cuyo estado no está
                      ni en ABIERTOS ni en CERRADOS
             2.4.2.2 Hacer ABIERTOS igual a
                      GESTIONA-COLA(ABIERTOS,NUEVOS SUCESORES)
3. Devolver FALLO.
```

Procedimiento general de búsqueda: comentarios

- La implementación anterior es *independiente del problema*
- **`ABIERTOS`** puede verse como una "cola" en la que esperan los nodos para ser analizados (*frontera* de la búsqueda)
- **`CERRADOS`** contiene los nodos ya analizados:
  - Permite no iniciar la búsqueda en estados analizados
  - En particular, nos permite evitar ciclos en el proceso de búsqueda
  - En determinados problemas es prescindible
- **`GESTIONA-COLA(ABIERTOS,NUEVOS SUCESORES)`**:
  - Añade **`NUEVOS-SUCESORES`** a **`ABIERTOS`**, reordenando según algún criterio concreto
  - Distintas concreciones de esta función dan lugar a distintos algoritmos de búsqueda (*estrategias* de búsqueda)
  - Búsqueda no informada o ciega *vs.* búsqueda informada

Propiedades a estudiar de los algoritmos de búsqueda

- Completitud: si existe solución, ¿la encuentra?
- Solución óptima o mínima: ¿obtiene la solución de menor número de pasos o de menor coste?
- Complejidad en tiempo: ¿cuánto se tarda en encontrar una solución?
- Complejidad en espacio: ¿cuánta memoria necesitamos?

Observaciones sobre la complejidad

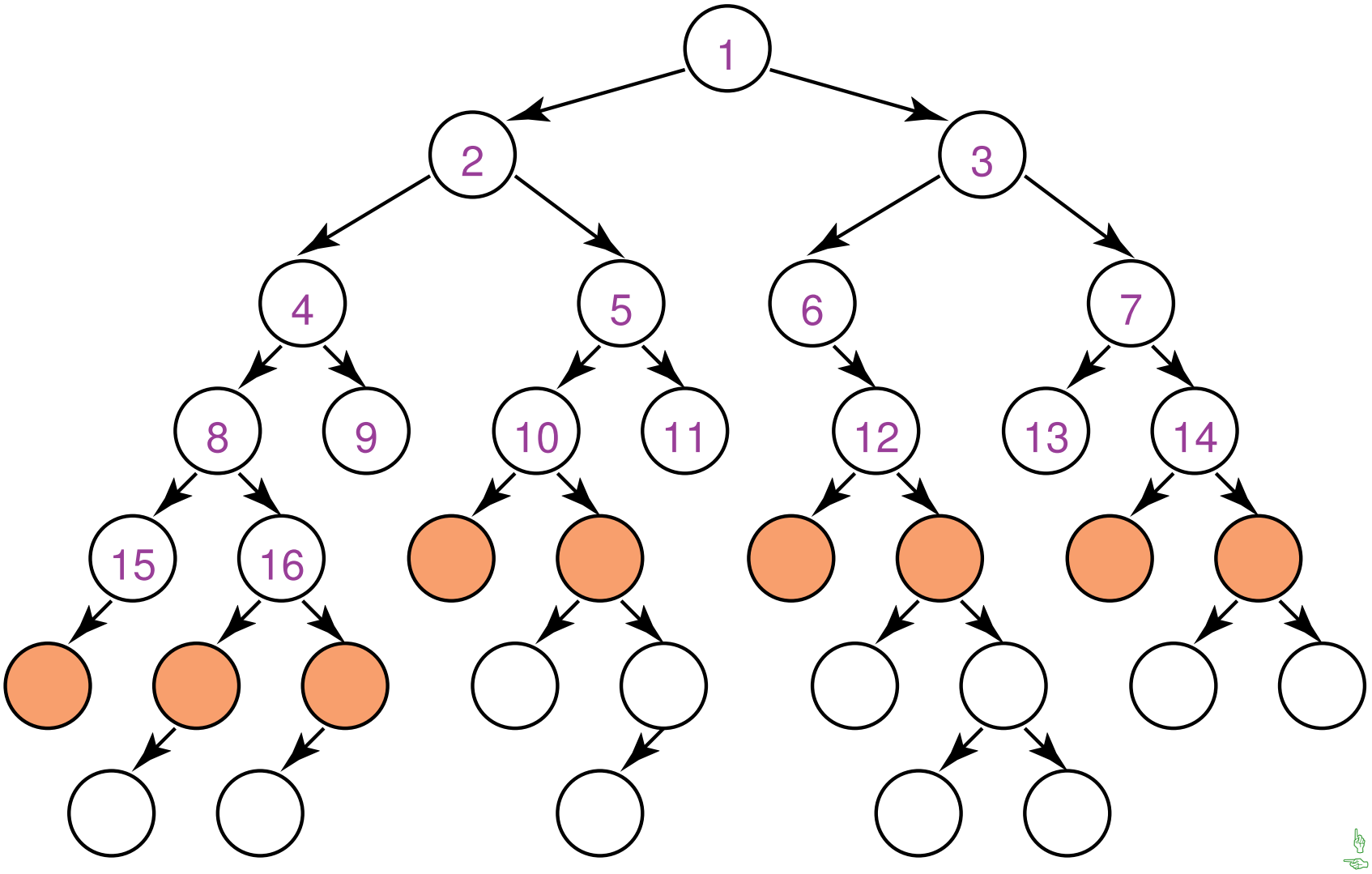
- Siempre en notación *O*, considerando el peor caso
- En función del tamaño del problema de entrada: ***r*** (máximo número de sucesores de un nodo, llamado *factor de ramificación*), ***p*** (mínima profundidad de una solución) y ***m*** (profundidad máxima de un camino)
- Complejidad en tiempo: número de nodos generados
- Complejidad en espacio: tamaño máximo de `ABIERTOS` (y `CERRADOS`) durante el proceso de búsqueda

Implementación de la búsqueda en anchura

- En la búsqueda en anchura, **ABIERTOS** se gestiona como una cola (FIFO):

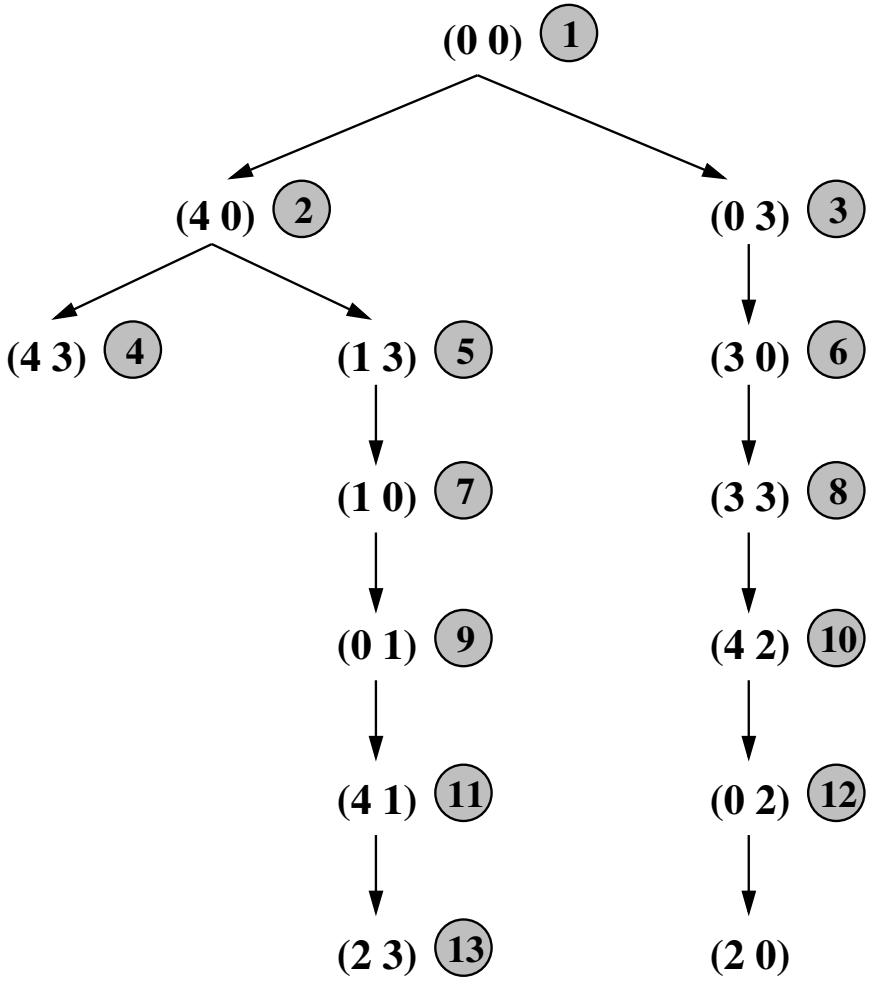
```
FUNCION BUSQUEDA-EN-ANCHURA()
1. Hacer ABIERTOS la cola formada por el nodo inicial (el nodo
    cuyo estado es *ESTADO-INICIAL*)
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario,
             2.4.2.1 Hacer NUEVOS-SUCESORES la lista de nodos
                      de SUCESORES(ACTUAL) cuyo estado no está
                      ni en ABIERTOS ni en CERRADOS
             2.4.2.2 Hacer ABIERTOS el resultado de incluir
                      NUEVOS-SUCESORES al final de ABIERTOS
3. Devolver FALLO.
```

Búsqueda en anchura



Búsqueda en anchura

- Árbol de búsqueda en anchura en el problema de las jarras:



Búsqueda en anchura

- Búsqueda en anchura para el problema de las jarras:

Nodo	Actual	Sucesores	Abiertos
			((0 0))
1	(0 0)	((4 0) (0 3))	((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((0 3) (4 3) (1 3))
3	(0 3)	((3 0))	((4 3) (1 3) (3 0))
4	(4 3)	()	((1 3) (3 0))
5	(1 3)	((1 0))	((3 0) (1 0))
6	(3 0)	((3 3))	((1 0) (3 3))
7	(1 0)	((0 1))	((3 3) (0 1))
8	(3 3)	((4 2))	((0 1) (4 2))
9	(0 1)	((4 1))	((4 2) (4 1))
10	(4 2)	((0 2))	((4 1) (0 2))
11	(4 1)	((2 3))	((0 2) (2 3))
12	(0 2)	((2 0))	((2 3) (2 0))
13	(2 3)		

Propiedades de la búsqueda en anchura

- Complejidad en tiempo $O(r^{p+1})$, donde:
  - r: factor de ramificación.
  - p: profundidad de la solución más corta.

$$r + r^2 + \cdots + (r^{p+1} - r) = O(r^{p+1})$$

- Complejidad en espacio $O(r^{p+1})$.
- Es completa.
- Obtiene una solución con el mínimo número de acciones.

Limitaciones de la búsqueda en anchura

- Tiempo y espacio exponenciales, la hace muchas veces inasumible en la práctica
- Suponiendo ramificación **10**, $\mathbf{10^6}$ nodos generados por seg. y 1000 bytes por nodo, aproximadamente:

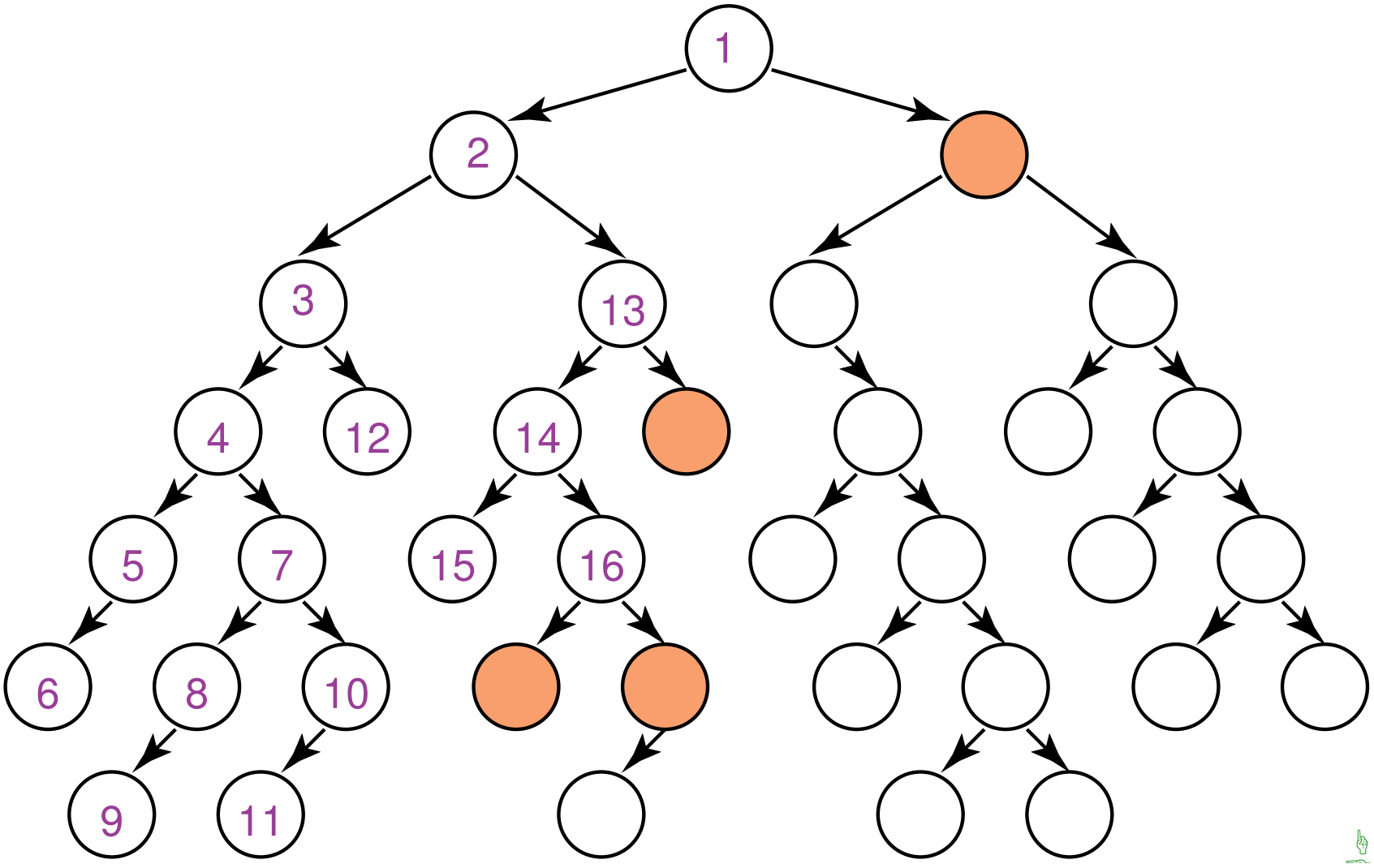
Profundidad	Nodos$\sim$	Tiempo	Espacio
2	**1100**	1.1 ms.	1 Mb
4	**111100**	110 ms.	106 Mb
6	$\mathbf{10^7}$	10 s.	10 Gb
8	$\mathbf{10^9}$	17 min.	1 Tb
10	$\mathbf{10^{11}}$	28 horas	100 Tb
12	$\mathbf{10^{13}}$	115 días	10 Pb
14	$\mathbf{10^{15}}$	32 años	1 Eb
16	$\mathbf{10^{17}}$	3170 años	100 Eb

Implementación de la búsqueda en profundidad

- Búsqueda en profundidad, **ABIERTOS** se gestiona como una pila (LIFO):

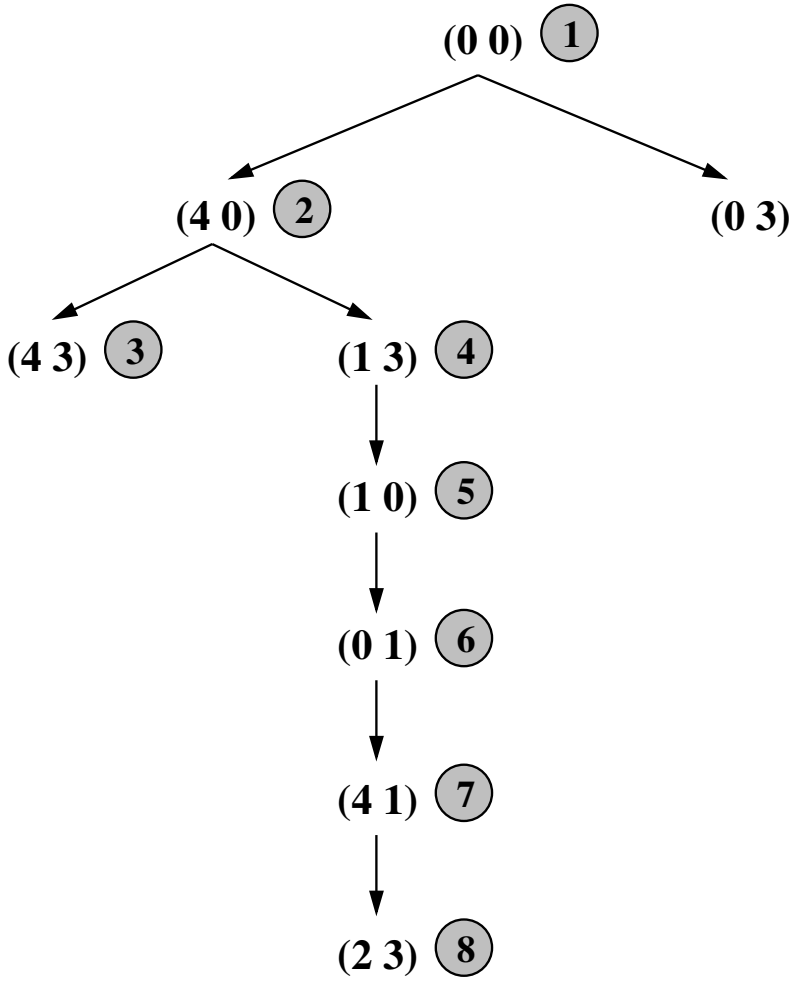
```
FUNCION BUSQUEDA-EN-PROFUNDIDAD()
1. Hacer ABIERTOS la pila formada por el nodo inicial (el nodo
    cuyo estado es *ESTADO-INICIAL*)
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario,
             2.4.2.1 Hacer NUEVOS-SUCESORES la lista de nodos
                     de SUCESORES(ACTUAL) cuyo estado no está
                     ni en ABIERTOS ni en CERRADOS
             2.4.2.2 Hacer ABIERTOS el resultado de incluir
                     NUEVOS-SUCESORES al principio de ABIERTOS
3. Devolver FALLO.
```

Búsqueda en profundidad



Búsqueda en profundidad

- Arbol de búsqueda en profundidad para el problema de las jarras:



Búsqueda en profundidad

- Tabla de búsqueda en profundidad para el problema de las jarras:

Nodo	Actual	Sucesores	Abiertos
			((0 0))
1	(0 0)	((4 0) (0 3))	((4 0) (0 3))
2	(4 0)	((4 3) (1 3))	((4 3) (1 3) (0 3))
3	(4 3)	()	((1 3) (0 3))
4	(1 3)	((1 0))	((1 0) (0 3))
5	(1 0)	((0 1))	((0 1) (0 3))
6	(0 1)	((4 1))	((4 1) (0 3))
7	(4 1)	((2 3))	((2 3) (0 3))
8	(2 3)		

Propiedades de la búsqueda en profundidad

- No es completa (puede no terminar)
  - Sí lo es en espacios finitos
- Cuando termina, no necesariamente obtiene una solución mínima.
- Complejidad en tiempo $O(r^m)$, donde:
  - r: factor de ramificación.
  - m: máxima profundidad de un camino.
- Complejidad en espacio:
  - en la implementación dada anteriormente (que usa la lista de `CERRADOS`), $O(r^m)$

Observaciones sobre la complejidad en espacio

- En la implementación que se ha dado:
  - El máximo tamaño de la lista de **ABIERTOS** es $O(r \cdot m)$
  - La complejidad exponencial la introduce la lista de **CERRADOS**
- Detección de ciclos en la búsqueda:
  - En algunos espacios de estados no hay posibilidad de caminos cíclicos (y por tanto no se necesita la lista de **CERRADOS** para detectar ciclos)
  - En general, para detectar ciclos sólo es necesario almacenar los nodos de la rama que se está explorando en cada momento
  - Es decir, es sencillo implementar una búsqueda en profundidad que, aún detectando ciclos, tiene complejidad espacial $O(r \cdot m)$
- Por tanto, en general consideraremos que la búsqueda en profundidad tiene complejidad espacial $O(r \cdot m)$

Búsqueda en profundidad acotada

- Podemos paliar en cierto modo la incompletitud de la búsqueda en profundidad
- Idea: no explorar caminos más allá de una determinada longitud.
- Problema: tampoco es completa, si la cota es menor de la longitud de la solución más corta
- Aunque en muchos casos se puede conocer de antemano una cota adecuada.

Implementación de la búsqueda en profundidad acotada

```
FUNCION BUSQUEDA-EN-PROFUNDIDAD-ACOTADA(COTA)
1. Hacer ABIERTOS la pila formada por el nodo inicial (el nodo
         cuyo estado es *ESTADO-INICIAL*);
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario, si PROFUNDIDAD(ACTUAL)
             es menor que la cota,
             2.4.2.1 Hacer NUEVOS-SUCESORES la lista de nodos
                      de SUCESORES(ACTUAL) cuyo estado no está
                      ni en ABIERTOS ni en CERRADOS
             2.4.2.2 Hacer ABIERTOS el resultado de incluir
                      NUEVOS-SUCESORES al principio de ABIERTOS
3. Devolver FALLO.
```

Propiedades de la búsqueda en profundidad acotada

- Complejidad en tiempo $O(r^c)$, donde:
  - r: factor de ramificación
  - c: cota de la profundidad
- Complejidad en espacio: $O(rc)$
- En general, no es completa
  - Si la cota es demasiado pequeña
  - Incluso si la cota se toma mayor que la longitud de una solución, la implementación anterior podría no encontrar solución, debido a `CERRADOS`
  - La versión que no usa `CERRADOS` sí es completa cuando la cota es mayor que la profundidad de una solución
- No necesariamente obtiene una solución mínima

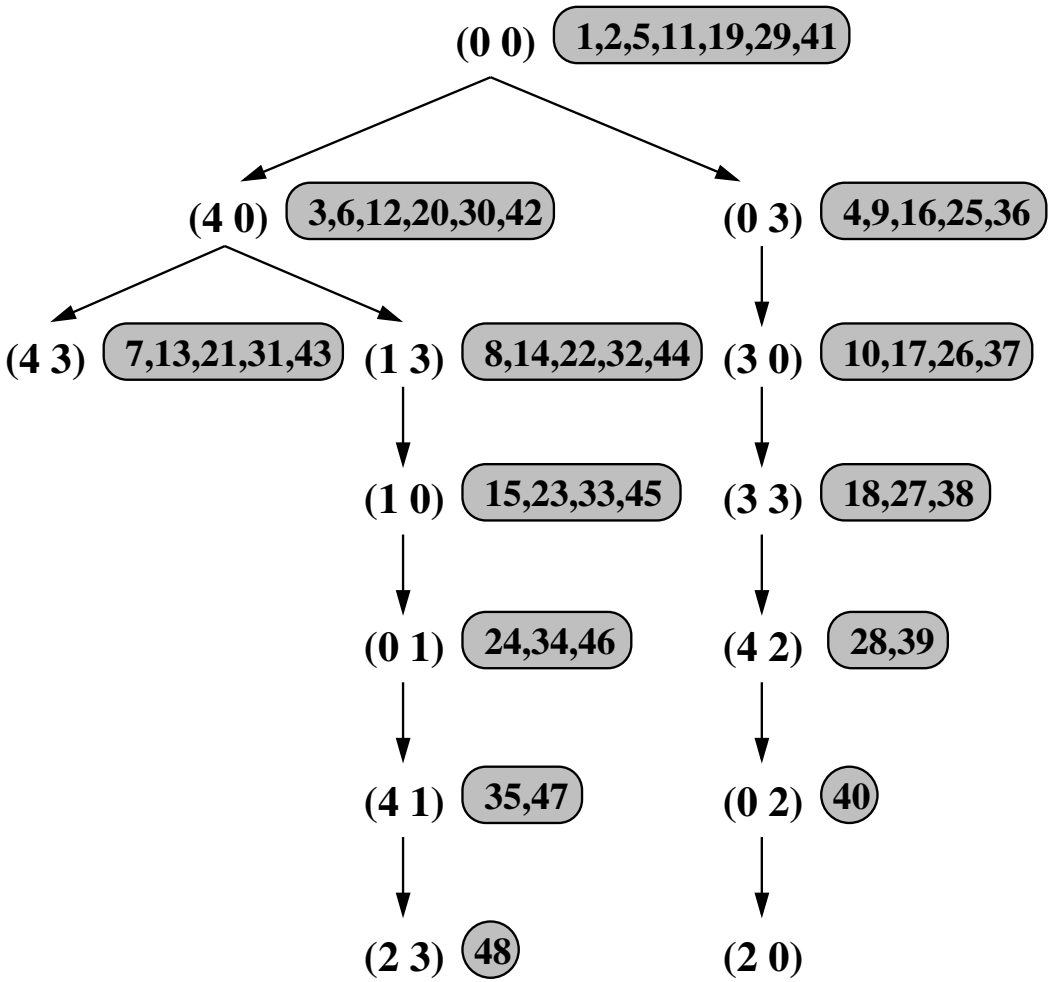
Búsqueda en profundidad iterativa

- Cuando no se conoce la cota, una opción para evitar la incompletitud es realizar búsquedas acotadas, incrementando la cota gradualmente
- Implementación de la búsqueda en profundidad iterativa

```
FUNCION BUSQUEDA-EN-PROFUNDIDAD-ITERATIVA(COTA-INICIAL)
1. Hacer N=COTA-INICIAL
2. Si BUSQUEDA-EN-PROFUNDIDAD-ACOTADA(N) no devuelve fallo,
      2.1 devolver su resultado y terminar.
      2.2 en caso contrario, hacer N igual a N+1 y volver a 2
```

Búsqueda en profundidad iterativa

- Arbol de búsqueda en profundidad iterativa (problema de las jarras):



Propiedades de la búsqueda en profundidad iterativa

- Complejidad en tiempo $O(r^p)$, donde:
  - r: factor de ramificación.
  - p: profundidad de solución.
- Complejidad en espacio, $O(rp)$.
- Es completa
- Obtiene una solución con el mínimo número de acciones (en la versión que no usa `CERRADOS`)

Propiedades de la búsqueda en profundidad iterativa

- La búsqueda iterativa es el método preferido si el espacio de búsqueda es grande y no se conoce la profundidad de la solución
- La redundancia en la expansión queda compensada con la completitud
  - Además, la redundancia no es significativa
  - Ejemplo, $\boldsymbol{r = 10}$ y $\boldsymbol{p = 5}$:
  - Búsqueda acotada, nodos generados:
    $\mathbf{10 + 100 + 1000 + 10000 + 100000 = 111110}$
  - Búsqueda iterativa, nodos generados:
    $\mathbf{10 + 110 + 1110 + 11110 + 111110 = 123450}$
  - Tan sólo un 10 % más
  - Razón: la mayoría de los nodos están en el último nivel del árbol

Comparación de procedimientos

	Anchura	Profundidad	Profundidad acotada	Profundidad iterativa
Tiempo	$O(r^{p+1})$	$O(r^m)$	$O(r^c)$	$O(r^p)$
Espacio	$O(r^{p+1})$	$O(rm)$	$O(rc)$	$O(rp)$
Completa	Sí	No	Sí, si $c >= p$	Sí
Mínima	Sí	No	No	Sí

- r: factor de ramificación.
- p: profundidad de la solución.
- m: máxima profundidad de la búsqueda.
- c: cota de la profundidad.

Espacio de estados con coste

- En algunos problemas, existe un coste (positivo) asociado a la aplicación de los acciones.
  - **`coste-de-aplicar-acción(estado,acción)`**.
  - Asumimos que **`acción`** es aplicable a **`estado`**.
- Coste asociado a un camino:
  - Suma de los costes de la aplicación de cada uno de sus acciones.
  - Una *solución óptima* es una solución con coste mínimo.
- En algunos problemas, no existe un coste explícito
  - En ese caso, **`coste-de-aplicar-acción(estado,acción)=1`**, y las soluciones óptimas son las minimales.
- Coste en el problema del viaje: distancia euclídea entre **`estado`** y **`aplica(acción,estado)`**

Idea de la búsqueda de coste uniforme

- Analizar primero los nodos con menor coste.
- Es decir, ordenar la cola de abiertos por coste, de menor a mayor
- De esta manera, cuando se llega por primera vez a un estado, se llega con el menor coste posible.
  - Y en particular, la primera vez que se llega a un estado final tenemos una solución óptima
- Se trata de una *búsqueda ciega* o *no informada*:
  - No usa conocimiento para guiar la búsqueda hacia el objetivo
  - Caso particular: búsqueda en anchura.

Implementación de la búsqueda de coste uniforme

- Nodo de búsqueda: añadir el campo “coste del camino”
  - Función de acceso: **COSTE-CAMINO(nodo)**
- Sucesores de un nodo con coste:

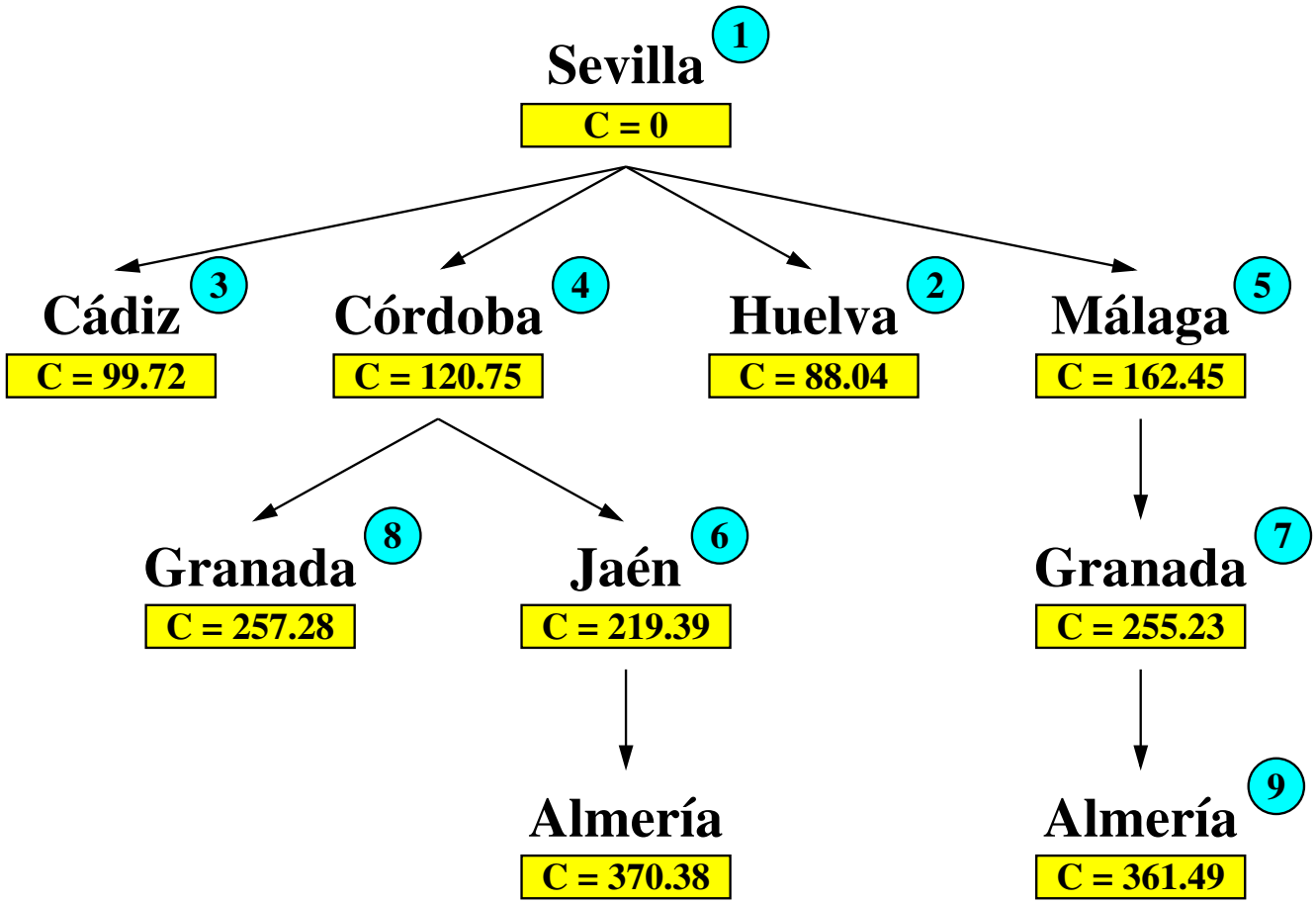
```
FUNCION SUCESOR(NODO,ACCIÓN)
1. Hacer ESTADO-SUCESOR igual a APLICA(ACCIÓN,ESTADO(NODO))
2. Devolver un nodo cuyo estado es ESTADO-SUCESOR, cuyo padre es NODO,
   cuya acción es ACCIÓN, cuya profundidad es PROFUNDIDAD(NODO)+1 y
   cuyo  coste es  COSTE-CAMINO(NODO) más
   COSTE-DE-APLICAR-ACCIÓN(ESTADO(NODO),ACCIÓN)

FUNCION SUCESORES(NODO)
1. Hacer SUCESORES igual a vacío
1. Para cada ACCIÓN en ACCIONES(ESTADO(NODO)),
          incluir SUCESOR(NODO,ACCIÓN) en SUCESORES
3. Devolver SUCESORES
```

Implementación de la búsqueda de coste uniforme

```
FUNCION BUSQUEDA-COSTE-UNIFORME()
1. Hacer ABIERTOS la cola formada por el nodo inicial (el nodo con
        estado *ESTADO-INICIAL*, camino vacío y coste 0);
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primero de ABIERTOS
       Hacer ABIERTOS el resto de ABIERTOS
   2.2 Poner el nodo ACTUAL en CERRADOS.
   2.3 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.3.1 devolver el nodo ACTUAL y terminar.
       2.3.2 en caso contrario,
             2.3.2.1 Hacer NUEVOS-SUCESORES la lista de nodos de
                     SUCESORES(ACTUAL) que o bien tienen un estado
                     que no aparece en los nodos de ABIERTOS ni de
                     CERRADOS, o bien su coste es menor que cualquier
                     otro nodo con el mismo estado que apareciera
                     en ABIERTOS o en CERRADOS
             2.3.2.2 Hacer ABIERTOS el resultado de incluir
                     NUEVOS-SUCESORES en ABIERTOS y ordenar
                     todo en orden creciente de los costes
                     de los caminos de los nodos
3. Devolver FALLO.
```

Árbol de búsqueda de coste uniforme para el problema del viaje



Propiedades de la búsqueda de coste uniforme

- Es completa (siempre que todos los costes individuales sean mayores que un cierto $\epsilon > \mathbf{0}$)
- Complejidad en tiempo y espacio: $\boldsymbol{O(r^{1+\lfloor C/\epsilon \rfloor})}$, donde $\boldsymbol{C}$ es el coste de una solución óptima.
  - Exponencial (es una generalización de la búsqueda en anchura)
- Siempre encuentra solución óptima.
- Salvo en espacios de estados pequeños, en la práctica esta búsqueda no es posible, debido a la cantidad de tiempo y espacio que necesita

Búsqueda informada

- Búsqueda ciega o no informada (anchura, profundidad,...): no cuenta con ningún conocimiento sobre cómo llegar al objetivo.
- Búsqueda informada: aplicar *conocimiento* al proceso de búsqueda para hacerlo más eficiente.
- El conocimiento vendrá dado por una función que *estima* la "bondad" de los estados:
  - Dar preferencia a los estados mejores.
  - Ordenando la cola de **`ABIERTOS`**, comparando su bondad estimada.
  - Objetivo: reducir el árbol de búsqueda, ganando *eficiencia* en la práctica.

Concepto de heurística

- Heurística:
  - Del griego *heuriskein*, descubrir: *¡Eureka!*
  - Según el DRAE: "técnica de la indagación y del descubrimiento".
  - Otro significado: método para resolver problemas que no garantiza la solución, pero que en general funciona bien.
  - En nuestro caso, una heurística será una función numérica sobre los estados.
- Función heurística, `heurística(estado)`:
  - Estima la "distancia" al objetivo.
  - Siempre mayor o igual que 0.
  - Valor en los estados finales: 0.
  - Admitimos valor ∞.
- Todo el conocimiento específico que se va a usar sobre el problema está codificado en la función heurística.

Recordar: problema del viaje por Andalucía

- Ir de Sevilla a Almería, mediante una secuencia de desplazamientos a capitales de provincia fronterizas
- 8 posibles estados:
  - Almería, Cádiz, Córdoba, Granada, Huelva, Jaen, Málaga, Sevilla
- Estado inicial: Sevilla.
- Estado final: Almeria.
- Acciones:
  - Ir a Almería, Ir a Cádiz, Ir a Córdoba, Ir a Granada, Ir a Huelva, Ir a Jaén, Ir a Málaga, Ir a Sevilla.

Una heurística para el problema del viaje

- Coordenadas:

```
Almeria : (409.5, 93  )
Granada : (309  ,127.5)
Malaga  : (232.5, 75  )
Cadiz   : ( 63  , 57  )
Huelva  : (  3  ,139.5)
Sevilla : ( 90  ,153  )
Cordoba : (198  ,207  )
Jaen    : (295.5,192  )
```

- Función heurística (distancia en línea recta):

```
heurística(estado) =
      distancia(coordenadas(estado),
                coordenadas(almeria))
```

Recordar: problema del 8-puzle

- Un tablero cuadrado (3x3) en el que hay situados 8 bloques cuadrados numerados (con lo cual se deja un hueco del tamaño de un bloque). Un bloque adyacente al hueco puede deslizarse hacia él. El juego consiste en transformar una posición inicial en la posición final mediante el deslizamiento de los bloques. En particular, consideramos el estado inicial y final siguientes:

2	8	3
1	6	4
7		5

Estado inicial

1	2	3
8		4
7	6	5

Estado final

- Estados: cada una de las posibles configuraciones del tablero
- Acciones: arriba, abajo, izquierda, derecha (movimientos del hueco)

Primera heurística en el problema del 8–puzle

- Problema del 8-puzle (primera heurística):
  - **`heurística(estado)`**: número de piezas descolocadas respecto de su posición en el estado final
- Ejemplo:

2	8	3
1	6	4
7		5

H = 4

1	2	3
8		4
7	6	5

H = 0

Segunda heurística en el problema del 8–puzle

- Problema del 8-puzle (segunda heurística):
  - **`heurística(estado)`**: suma de las distancias Manhattan de cada pieza a donde debería estar en el estado final
- Ejemplo:

2	8	3
1	6	4
7		5

H = 5

1	2	3
8		4
7	6	5

H = 0

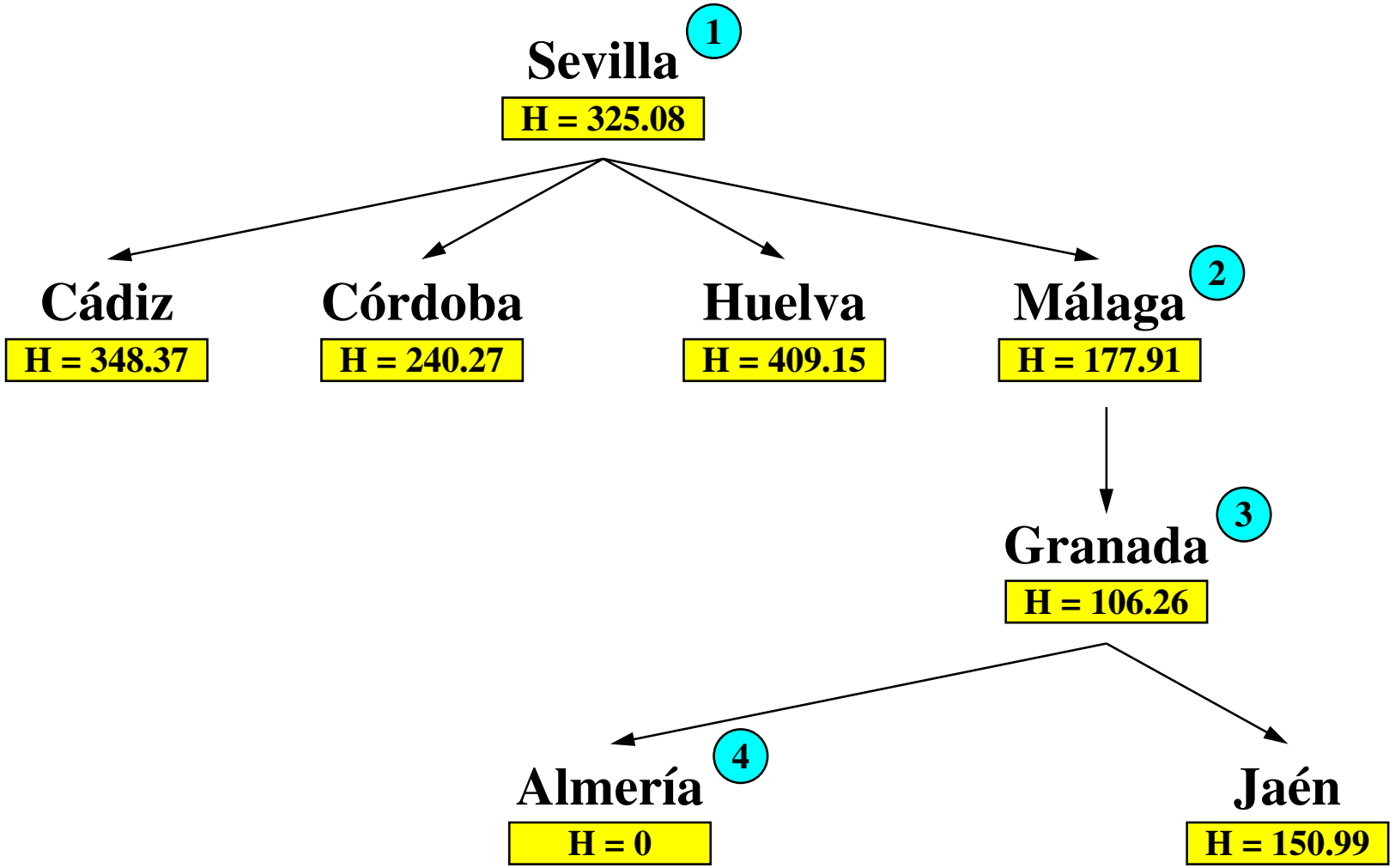
Idea de la búsqueda por primero el mejor

- Búsqueda por primero el mejor:
  - Analizar preferentemente los nodos con heurística más baja.
  - Ordenar la cola de abiertos por heurística, de menor a mayor
- También llamada búsqueda *voraz* o *codiciosa* (del inglés *"greedy"*)
  - Porque siempre elige expandir *lo que estima* que está más "cerca" del objetivo
- Su rendimiento dependerá de la bondad de la heurística usada.

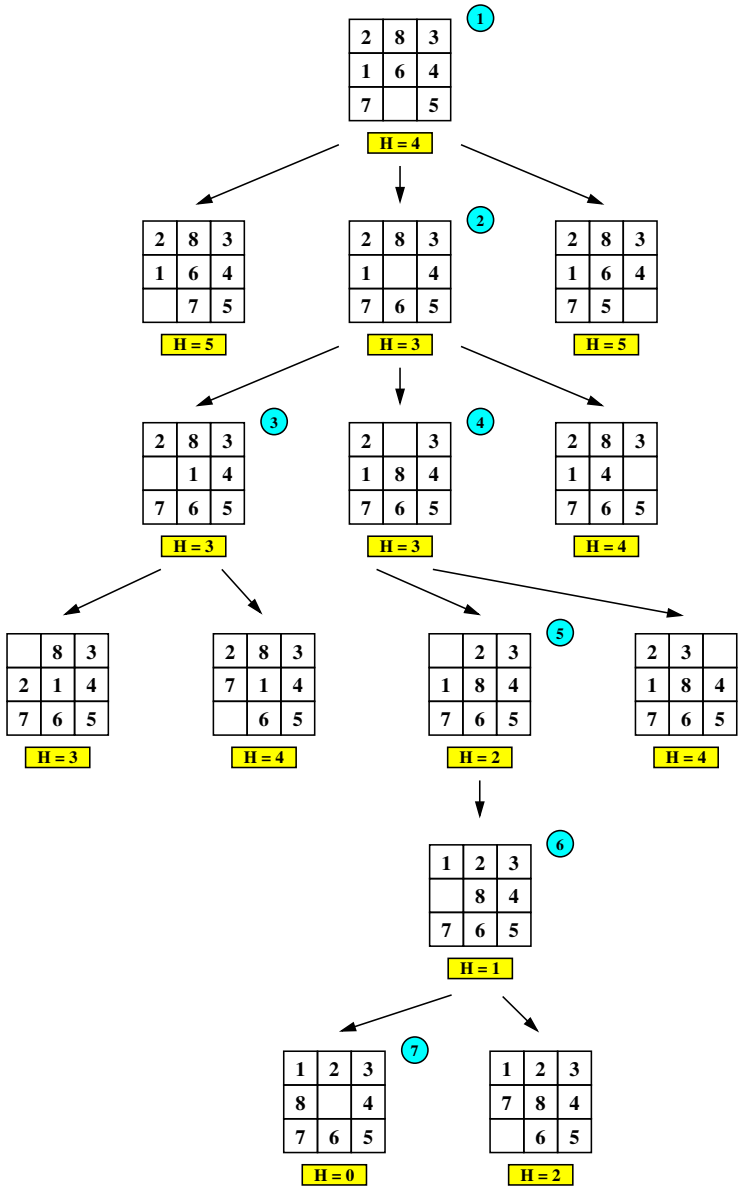
Implementación de la búsqueda por primero el mejor

```
FUNCION BUSQUEDA-POR-PRIMERO-EL-MEJOR()
1. Hacer ABIERTOS la cola formada por el nodo inicial (el nodo
    cuyo estado es *ESTADO-INICIAL*);
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
   2.2 Hacer ABIERTOS el resto de ABIERTOS
   2.3 Poner el nodo ACTUAL en CERRADOS.
   2.4 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.4.1 devolver el nodo ACTUAL y terminar.
       2.4.2 en caso contrario,
             2.4.2.1 Hacer NUEVOS-SUCESORES la lista de nodos
                     de SUCESORES(ACTUAL) cuyo estado no está
                     ni en ABIERTOS ni en CERRADOS
             2.4.2.2 Hacer ABIERTOS el resultado de incluir
                     NUEVOS-SUCESORES en ABIERTOS y ordenar
                     en orden creciente de las heurísticas
                     de los estados de los nodos
3. Devolver FALLO.
```

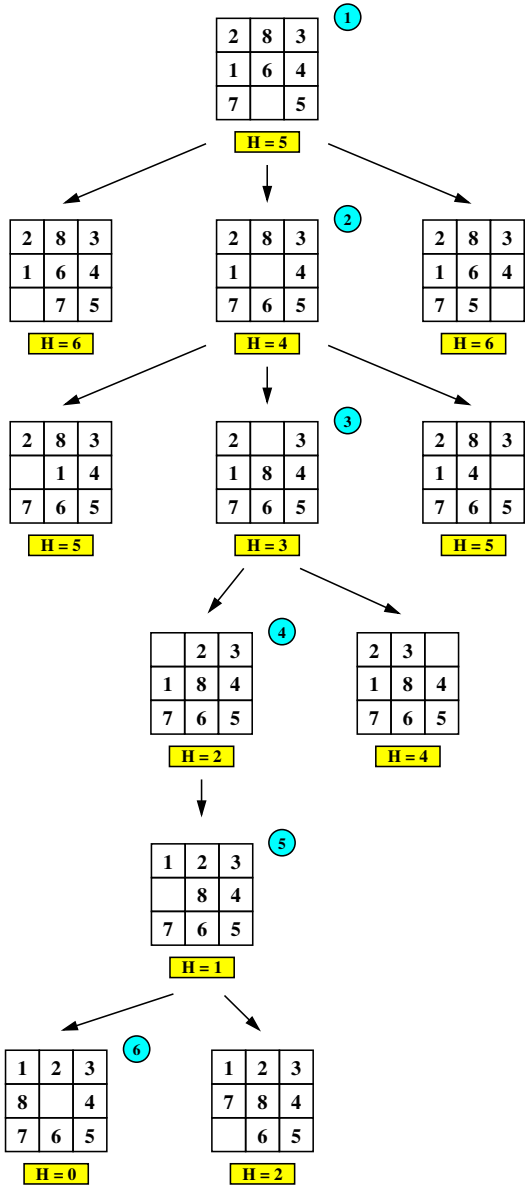
Primero el mejor para el problema del viaje



8–puzle por primero el mejor: heurística 1



8–puzle por primero-el-mejor: heurística 2

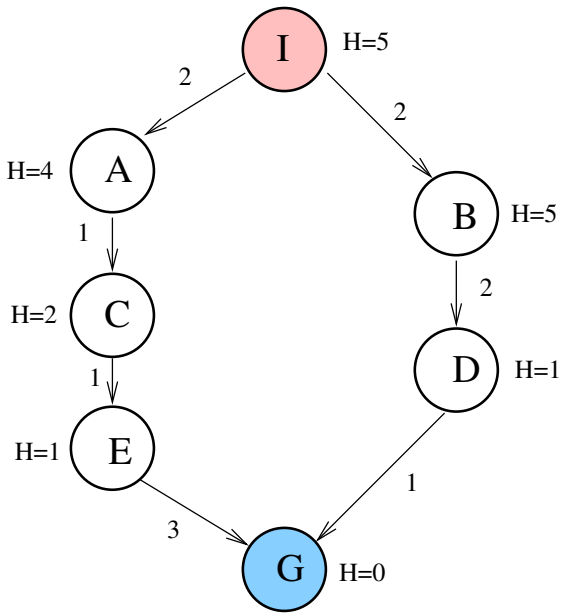


Propiedades de la búsqueda por primero el mejor

- Complejidad en tiempo $O(r^m)$, donde:
  - r: factor de ramificación.
  - m: profundidad máxima del árbol de búsqueda.
- Complejidad en espacio: $O(r^m)$.
- En la práctica, el tiempo y espacio necesario depende del problema concreto y de la calidad de la heurística usada
- No es completa, en general.
  - Por ejemplo, una mala heurística podría hacer que se tomara un camino infinito.
- No es minimal (no garantiza soluciones con el menor número de acciones).
  - La heurística podría guiar hacia una solución no minimal

Uso de heurísticas para buscar soluciones óptimas

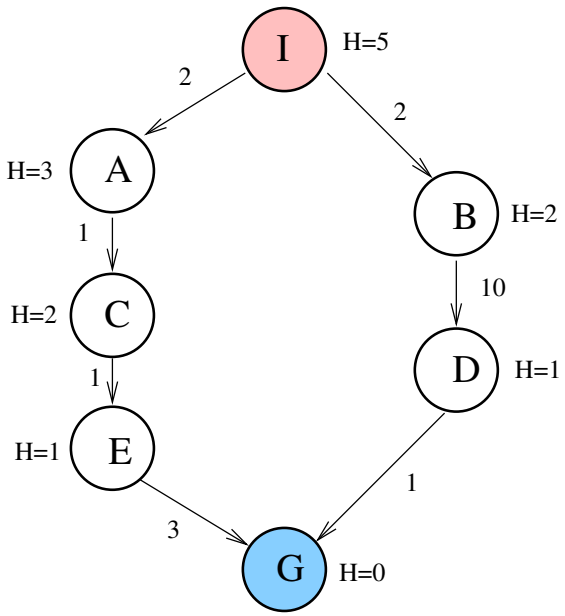
- ¿Podemos usar búsqueda por primero el mejor para acelerar la búsqueda y aún estar seguro de obtener soluciones óptimas? En general, NO.
- Ejemplo 1:



  - Solución encontrada por primero el mejor: I-A-C-E-G (subóptima)
  - Causa: la heurística *sobrestima* el coste real

Uso de heurísticas para buscar soluciones óptimas

- Ejemplo 2:



  - Solución encontrada por primero el mejor: I-B-D-G (subóptima)
  - Causa: no se han tenido en cuenta *los costes* de los caminos ya recorridos

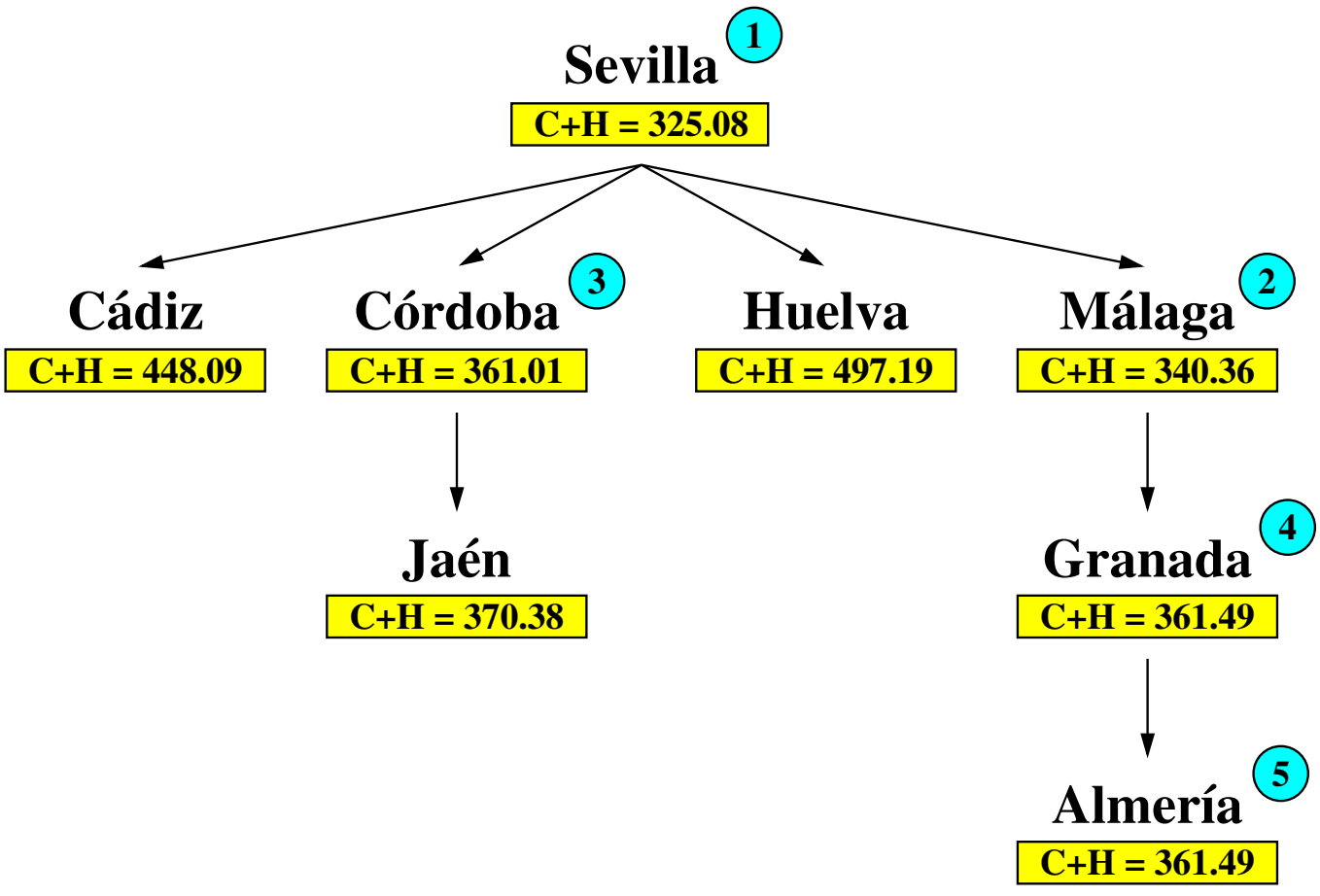
Idea de la búsqueda A*

- Objetivo de la búsqueda A*:
  - conseguir buenas soluciones (óptimas).
  - ganar en eficiencia (reduciendo el árbol de búsqueda).
- Idea: asignar a cada nodo n un valor $f(n) = g(n) + h(n)$
  - $g(n)$: coste del camino hasta n
  - $h(n)$: heurística del nodo, *estimación* del coste *de un camino óptimo* desde n hasta un estado final
  - $f(n)$: estimación del coste total de una solución óptima *que pasa por* n
- Seleccionar siempre el nodo con menor valor de f
  - ordenando la cola de `ABIERTOS` en orden creciente respecto a f

Implementación de la búsqueda A*

```
FUNCION BUSQUEDA-A-ESTRELLA()
1. Hacer ABIERTOS la cola formada por el nodo inicial (el nodo cuyo
    estado es *ESTADO-INICIAL*, cuyo camino es vacío, cuyo coste
    es 0, y cuyo coste-más-heurística es HEURISTICA(*ESTADO-INICIAL*))
   Hacer CERRADOS vacío
2. Mientras que ABIERTOS no esté vacía,
   2.1 Hacer ACTUAL el primer nodo de ABIERTOS
       Hacer ABIERTOS el resto de ABIERTOS
   2.2 Poner el nodo ACTUAL en CERRADOS.
   2.3 Si ES-ESTADO-FINAL(ESTADO(ACTUAL)),
       2.3.1 devolver el nodo ACTUAL y terminar.
       2.3.2 en caso contrario,
          2.3.2.1 Hacer NUEVOS-SUCESORES la lista de nodos de
                  SUCESORES(ACTUAL) que o bien tienen un estado
                  que no aparece en los nodos de ABIERTOS ni
                  de CERRADOS, o bien su coste es menor que
                  cualquier otro nodo con el mismo estado que
                  apareciera en ABIERTOS o en CERRADOS
          2.3.2.2 Hacer ABIERTOS el resultado de incluir
                  NUEVOS-SUCESORES en ABIERTOS y ordenar
                  todo en orden creciente de
                  del coste+heurística de los nodos
3. Devolver FALLO.
```

Árbol de búsqueda A* para el viaje



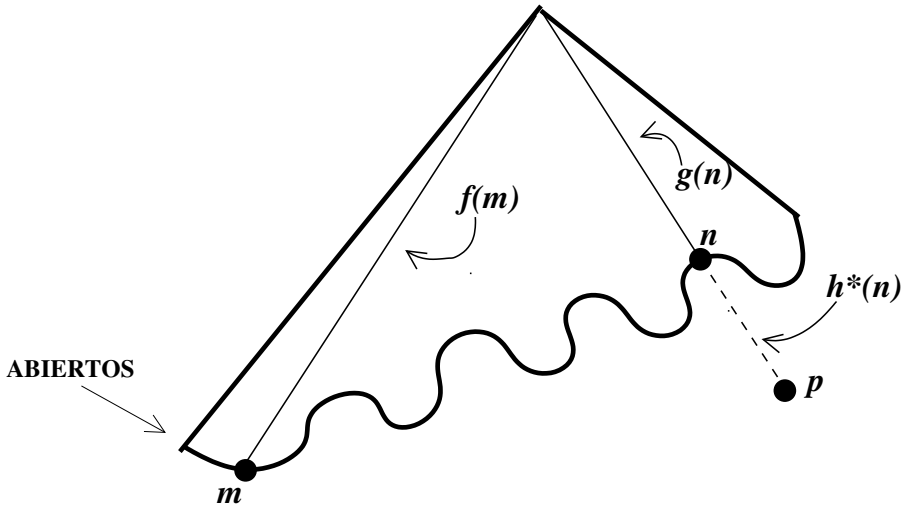
Propiedades de A*

- Sea $h^*(n)$ coste de un camino óptimo desde el estado de n hasta un estado final
  - $f^*(n) = g(n) + h^*(n)$ coste total de una solución óptima *que pasa por* n
- En la práctica, no conocemos h^*
  - Usamos una función heurística h que estima h^*
- Posibilidades:
  - Para todo nodo n, $h(n) = 0$: búsqueda de coste uniforme, no hay reducción del árbol de búsqueda
  - Para todo nodo n, $h(n) = h^*(n)$: estimación perfecta, no hay búsqueda
  - Para todo nodo n, $0 \leq h(n) \leq h^*(n)$: heurística *admisible*, solución óptima garantizada
  - Para algún nodo n, $h(n) > h^*(n)$: no se puede garantizar que la solución encontrada sea óptima

Propiedades de A*

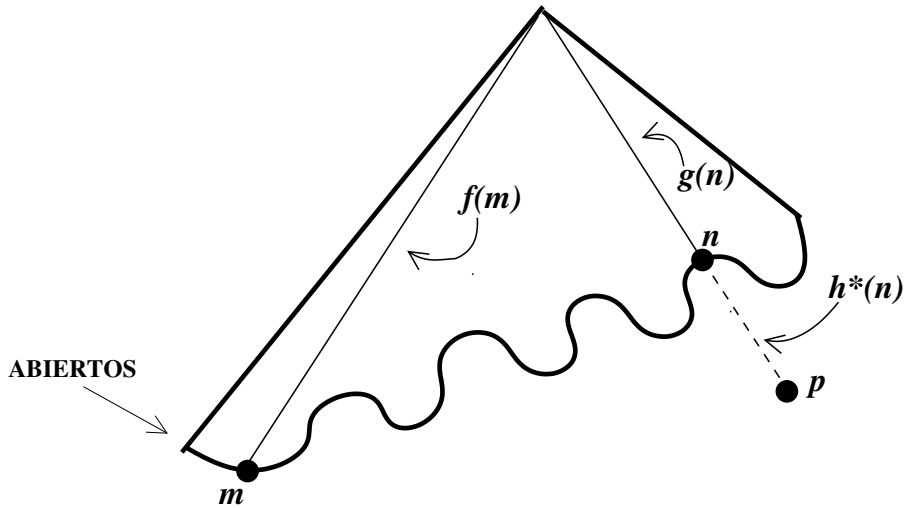
- Usando una heurística admisible:
  - Es completa.
  - Encuentra siempre solución óptima.
- Complejidad en tiempo y en espacio: en el peor de los casos, como la búsqueda de coste uniforme
- En la práctica, el coste en tiempo y espacio depende del problema particular y de la calidad de la heurística usada

Demostración de que A* es óptimo



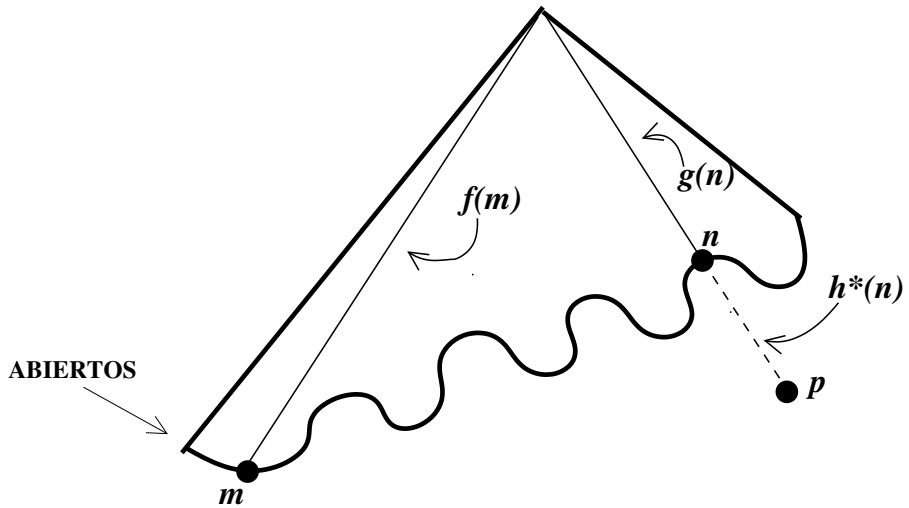
Sea ***m*** un nodo correspondiente a una solución *no óptima*. Vamos a probar que si la heurística ***h*** es admisible, entonces el nodo ***m*** nunca es el primero en la lista de **`ABIERTOS`** y por tanto el algoritmo ***A**** nunca devuelve ***m*** como solución.

Demostración de que A* es óptimo



- Para ello, sea ***p*** un nodo correspondiente a una solución *óptima* y consideremos el camino que une el nodo inicial con ***p***.
- Puesto que **`ABIERTOS`** siempre contiene a los nodos no explorados, cualquier conjunto **`ABIERTOS`** que contenga a ***m*** también contendrá algún nodo del camino que une al estado inicial con ***p***. Llamaremos ***n*** a ese nodo.

Demostración de que A* es óptimo



Puesto que ***m*** y ***n*** pertenecen al mismo conjunto **ABIERTOS**, para ver que ***m*** nunca es el primero, basta ver que $f(n) < f(m)$. Vamos a probarlo.

Demostración de que A* es óptimo

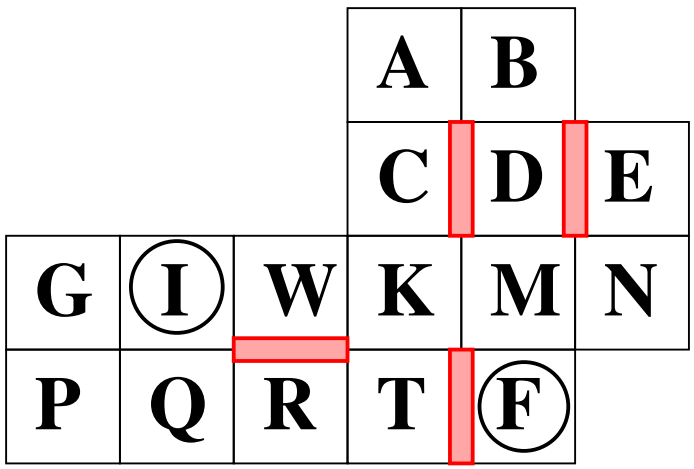
$$
\begin{aligned}
f(n) &= g(n) + h(n) && \text{[Por definición de } f\text{]} \\
&\leq g(n) + h^*(n) && \text{[Por ser } h \text{ admisible]} \\
&= g(p) && \text{[Por definición de } h^*\text{]} \\
&< g(m) && \text{[Por ser } m \text{ final no óptimo]} \\
&= g(m) + h(m) && [m \text{ es final, } h(m) = 0] \\
&= f(m) && \text{[Por definición de } f\text{]}
\end{aligned}
$$

Invención de heurísticas

- Recordar: estimación del coste mínimo restante hasta un estado final
- Comparación de heurísticas:
  - Si para todo nodo n, $0 \leq h_1(n) \leq h_2(n) \leq h^*(n)$, entonces h_2 está *más informada* que h_1 y mejora la eficiencia
- Técnicas para encontrar heurísticas
  - Relajación del problema
  - Combinación de heurísticas admisibles
  - Uso de información estadística
- Evaluación eficiente de la función heurística

Problema del laberinto

- En el siguiente laberinto, se puede pasar desde una casilla a otra de las posibles adyacentes (arriba, abajo, izquierda, derecha), salvo si existe una barrera entre ellas

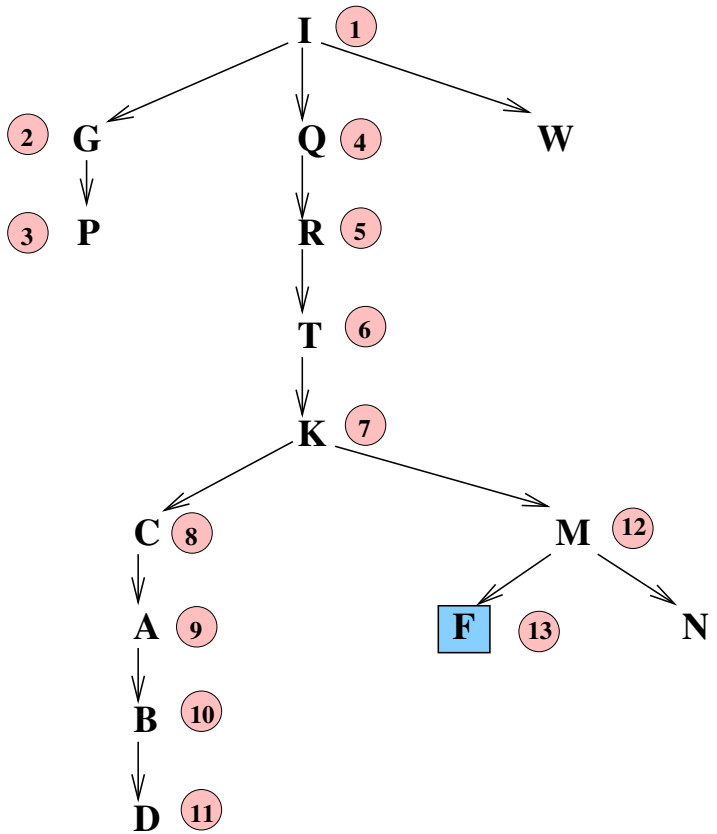


- Objetivo: ir de I a F
- Como ejercicio final, resolveremos el problema aplicando las búsquedas en profundidad, primero el mejor y A*

Problema del laberinto como problema de espacio de estados

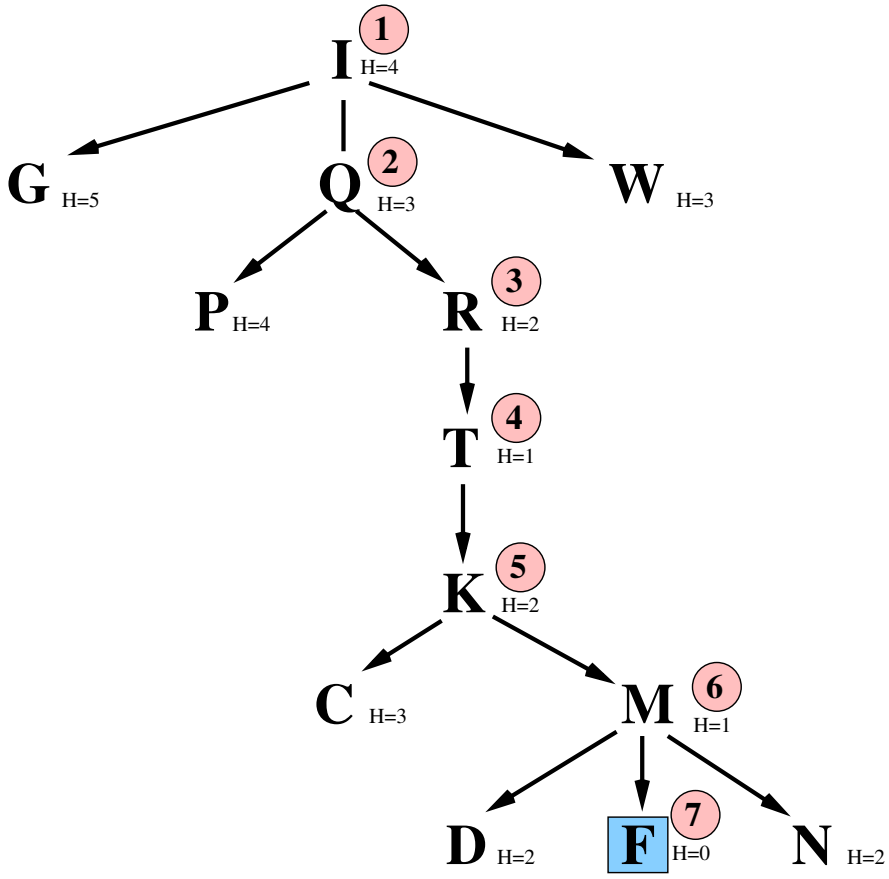
- Estados: A, B, C, D, E, G, I, W, K, M, N, P, Q, R, T y F
- Estado inicial: I
- Estado final: F
- Acciones: arriba, abajo, izquierda, derecha
- Aplicabilidad y resultado de la aplicación
  - "Arriba" es aplicable a un estado si existe una casilla arriba y no está separada por barrera; el resultado de aplicarlo es la casilla de arriba y su coste es 1
  - El resto de acciones, análogamente
  - Consideraremos que los sucesores están ordenados alfabéticamente
- Heurística (admisible): "distancia Manhattan del estado a la casilla F"
- Durante la búsqueda, y con igual valoración, elegir por orden alfabético

Laberinto: búsqueda en profundidad



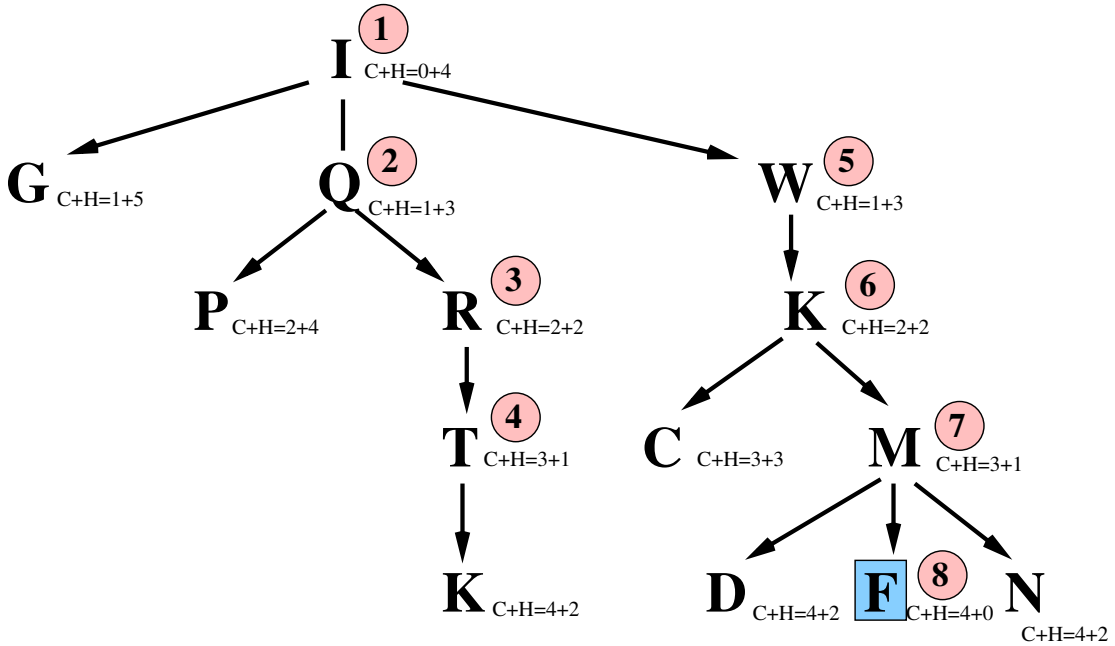
- Solución encontrada: I-Q-R-T-K-M-F
  - No es óptima
  - Nodos analizados: 13

Laberinto: búsqueda primero el mejor



- Solución encontrada: I-Q-R-T-K-M-F
  - No es óptima
  - Nodos analizados: 7
  - La heurística ha servido para reducir el espacio de búsqueda

Laberinto: búsqueda A*



- Solución encontrada: I-W-K-M-F
  - Óptima (heurística admisible)
  - Nodos analizados: 8
  - Aunque en este caso se analizan más nodos (se ha explorado parcialmente un camino equivocado), la solución óptima está asegurada
  - Nótese que se generan dos nodos distintos con el mismo estado K (pero distinto camino)

Bibliografía

- Russell, S. y Norvig, P. *Inteligencia artificial: Un enfoque moderno (segunda edición)* (Prentice Hall, 2004).
  - Cap. 3: “Solución de problemas mediante búsqueda”
- Russell, S. y Norvig, P. *Artificial Intelligence (A Modern Approach)* (Prentice–Hall, 2010). Third Edition
  - Cap. 3: “Solving problems by searching”.

Bibliografía complementaria

- Luger, G.F. *Artificial Intelligence (Structures and Strategies for Complex Problem Solving (4 edition)* (Addison–Wesley, 2002)
  - Cap. 3: “Structure and strategies for state space search”
  - Cap. 4: “Heuristic search”
- Nilsson, N.J. *Inteligencia artificial (Una nueva síntesis)* (McGraw–Hill, 2001)]
  - Cap. 8: “Búsqueda”
- Poole, D.; Mackworth, A. y Goebel, R. *Computational Intelligence (A Logical Approach)* (Oxford University Press, 1998).
  - Cap. 4: “Searching”