# Tema 3: Planificación

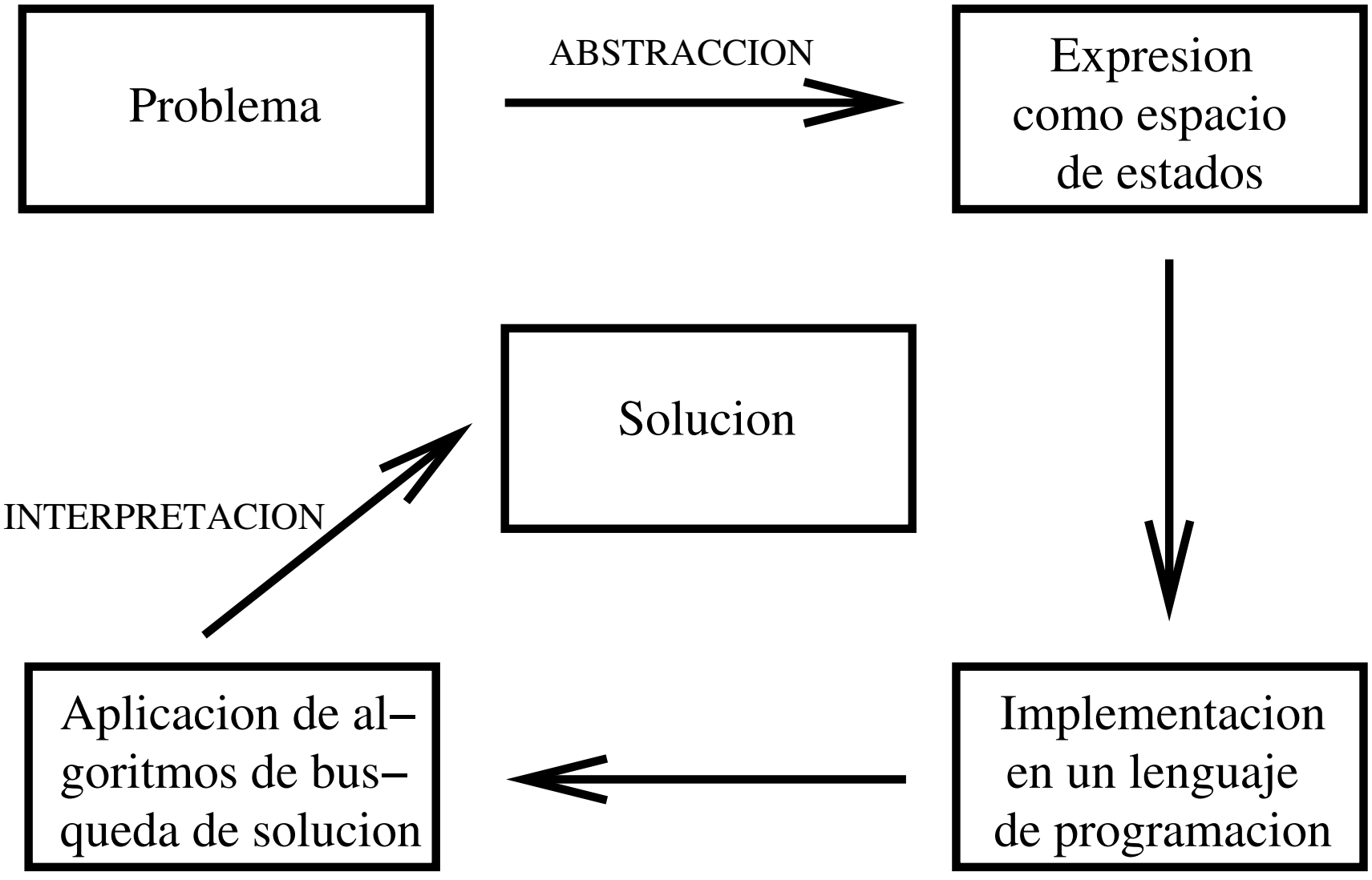
José Luis Ruiz Reina
Miguel A. Gutiérrez Naranjo
Francisco J. Martín Mateos

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

Inteligencia Artificial

# Índice

# 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

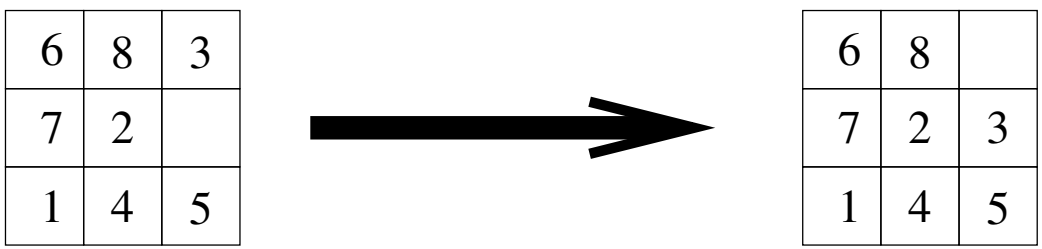
- Estado: descripción de una posible situación en el problema
  - Abstracción de propiedades
- Importancia de una buena representación de los estados
  - Sólo considerar información relevante para el problema
  - La representación escogida influye en el número de estados y éste en los procedimientos 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,...

# 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

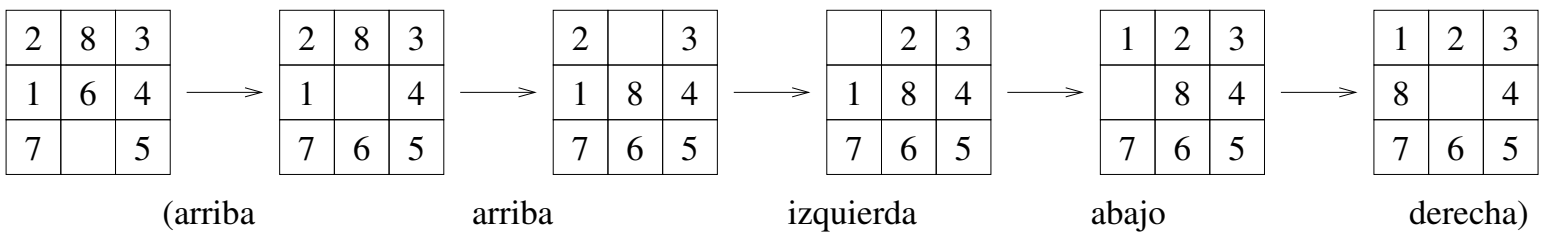
- Descripción del objetivo
  - Usualmente, un conjunto de estados, que llamaremos *finales*
  - A veces, aunque no necesariamente, un único estado final
- Ejemplo del 8-puzle (un único estado final)

| | | |
|---|---|---|
| 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:

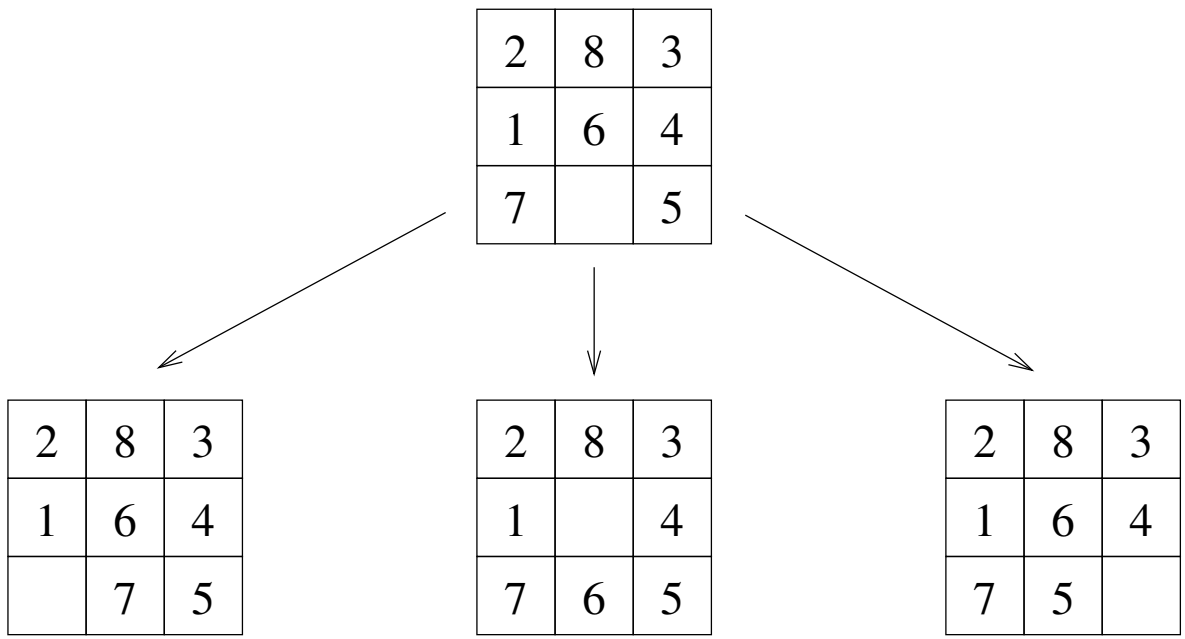


# Soluciones de un problema

- 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



# 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

# Problemas de la vida real

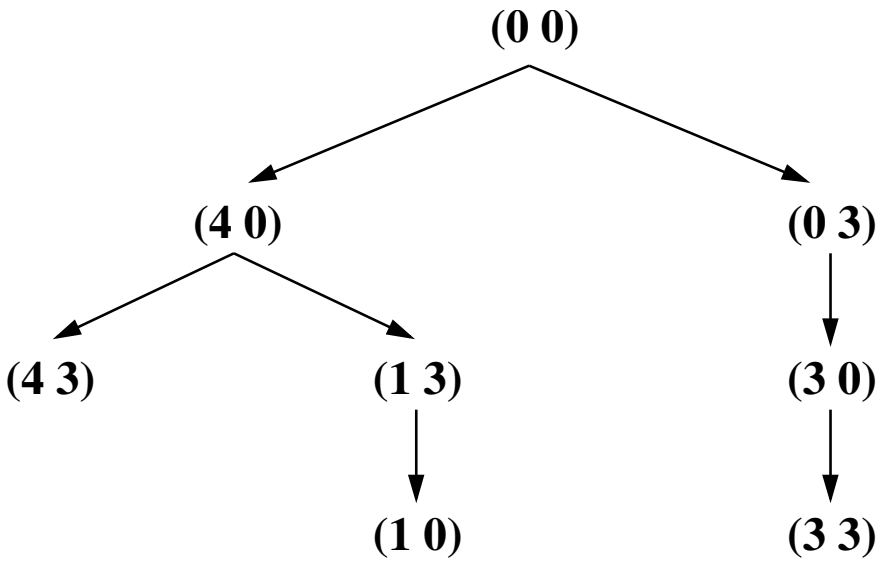
- Problemas de la vida real que se pueden plantear y resolver como espacios de estados:
  - Búsqueda de rutas en redes informáticas
  - Rutas aéreas para viajar
  - Problema del viajante
  - Diseño de microchips
  - Ensamblaje de componentes
  - Desplazamiento de robots

# Búsqueda de soluciones en espacios de estados

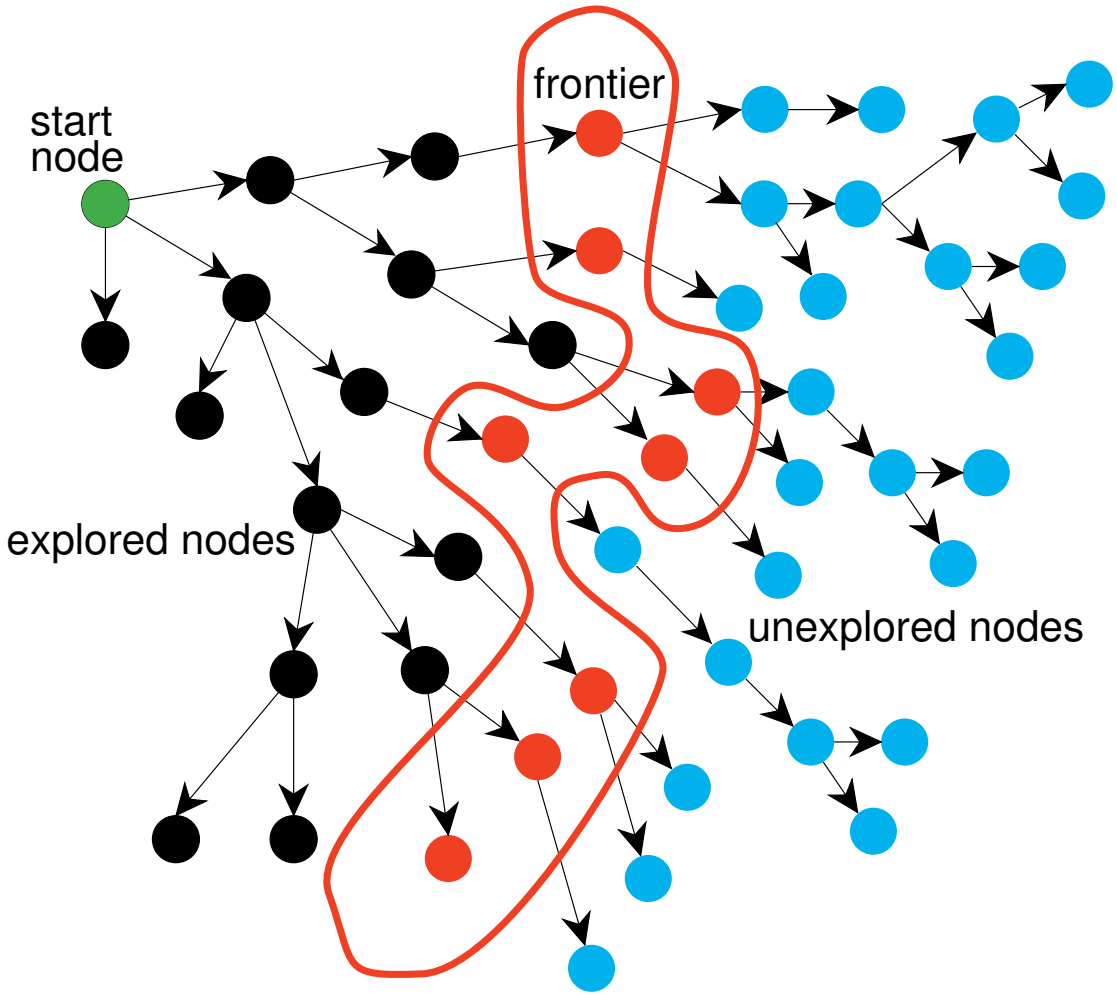
- Objetivo: encontrar una secuencia de acciones que, partiendo del estado inicial, obtenga un estado final
- Idea básica: exploración del *grafo* del espacio de estados
  - En cada momento se analiza un estado *actual* (en un principio, el *inicial*) y se mantiene una *frontera* de exploración (llamada lista de **`ABIERTOS`**)
  - Si el estado actual es *final*, devolver la sucesión de acciones que han llevado hasta él
  - En caso contrario, obtener los sucesores del estado actual (*expandir*) y añadirlos a la frontera de exploración
  - *Elegir* un nuevo estado actual de la frontera
  - Repetir el proceso mientras haya estados en la frontera de exploración
- 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



# Implementación del espacio de estados

- 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)

# Implementación de la búsqueda (funciones auxiliares)

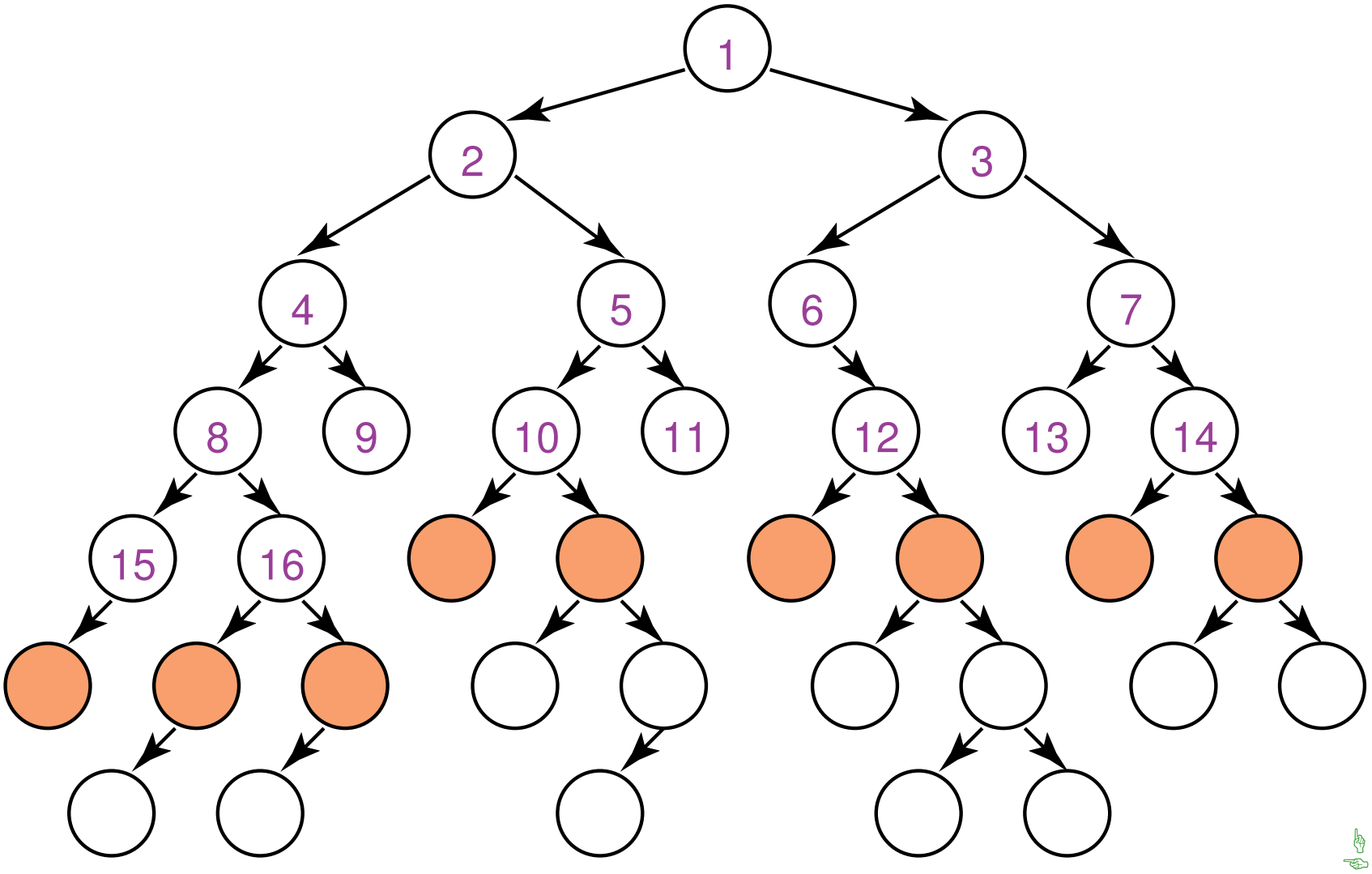
- Nodo de búsqueda: estado + padre + acción + profundidad
  - Funciones de acceso: **`ESTADO(NODO)`**, **`ANTECESOR(NODO)`**, **`ACCIÓN(NODO)`**, **`PROFUNDIDAD(NODO)`**
- Sucesores de un nodo:
  - **`SUCESOR(NODO,ACCIÓN)`** calcula **`NUEVO-ESTADO`**, resultado de aplicar **`ACCIÓN`** al estado asociado al **`NODO`**, y devuelve un nuevo nodo, cuyo estado es **`NUEVO-ESTADO`**, cuyo padre es **`NODO`**, cuya acción es **`ACCIÓN`** y cuya profundidad es **`PROFUNDIDAD(NODO)+1`**
  - **`SUCESORES(NODO)`** devuelve la lista de nodos obtenida aplicando la función anterior con cada **`ACCIÓN`** que sea aplicable a **`ESTADO(NODO)`**

# Implementación de la búsqueda en anchura

- En la búsqueda en anchura, la frontera de **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



# Complejidad algorítmica 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:

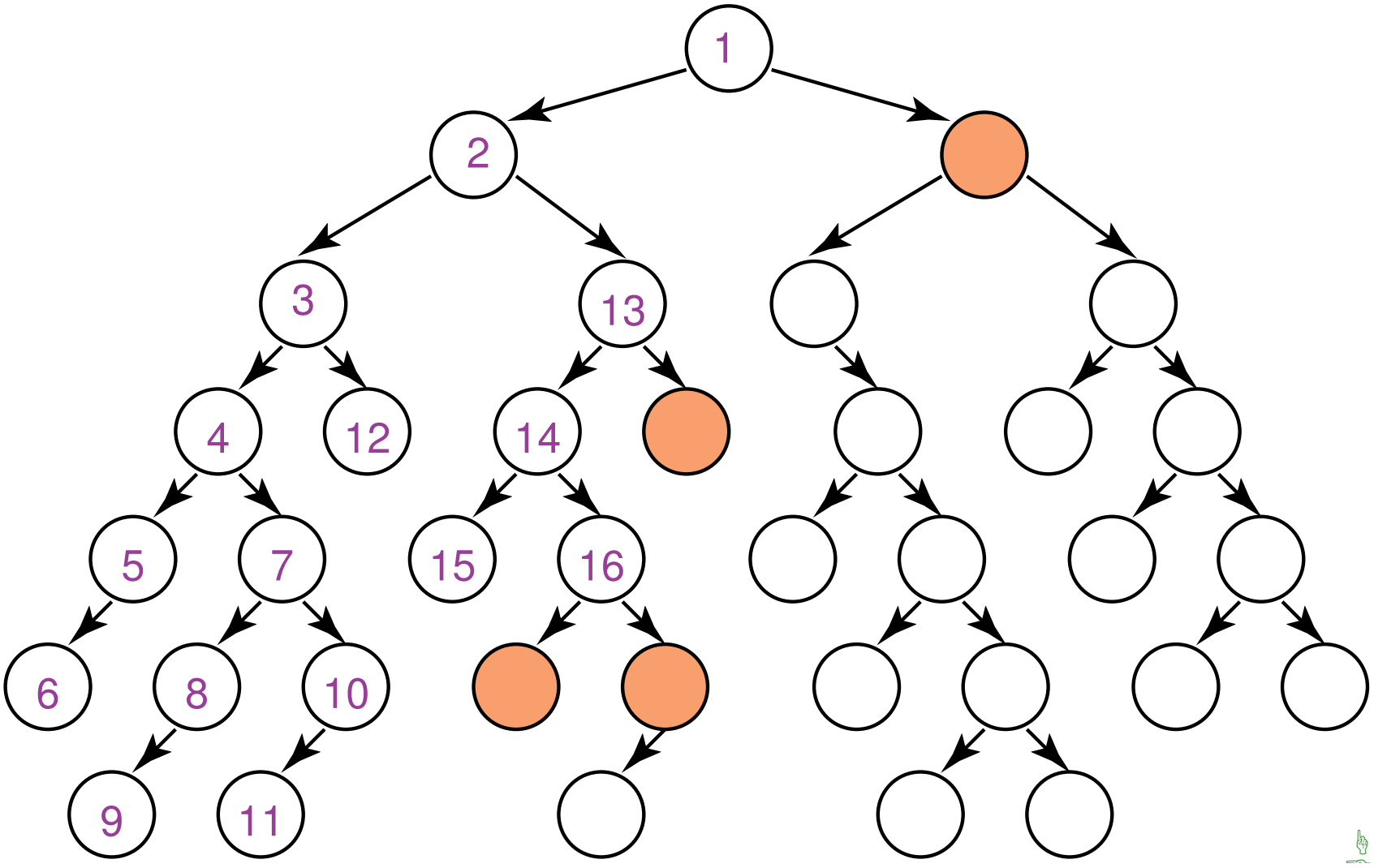
| Profundidad | Nodos$\sim$ | Tiempo | Espacio |
|---|---|---|---|
| 2 | **1100** | 1.1 ms. | 1 Mb |
| 4 | **111100** | 110 ms. | 106 Mb |
| 6 | $\mathbf{10^7}$ | 10 s. | 10 Gb |
| 8 | $\mathbf{10^9}$ | 17 min. | 1 Tb |
| 10 | $\mathbf{10^{11}}$ | 28 horas | 100 Tb |
| 12 | $\mathbf{10^{13}}$ | 115 días | 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 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.
```

# 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
```

# Procedimientos de búsqueda: propiedades

| | 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.

# 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.
  - Método *heurístico*
- El conocimiento vendrá dado por una función que *estima* la "bondad" de los estados:

# Función heurística

- En este contexto, la técnica heurística se refleja en una *función* que cuantifica cómo de "cerca" estamos de un estado final
- Función heurística, **`heurística(estado)`**:
  - Estima la "distancia" al objetivo.
  - Siempre mayor o igual que 0.
  - Valor en los estados finales: 0.
  - Admitimos valor $\infty$.
- Todo el conocimiento específico que se va a usar sobre el problema está codificado en la función heurística.

# 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**

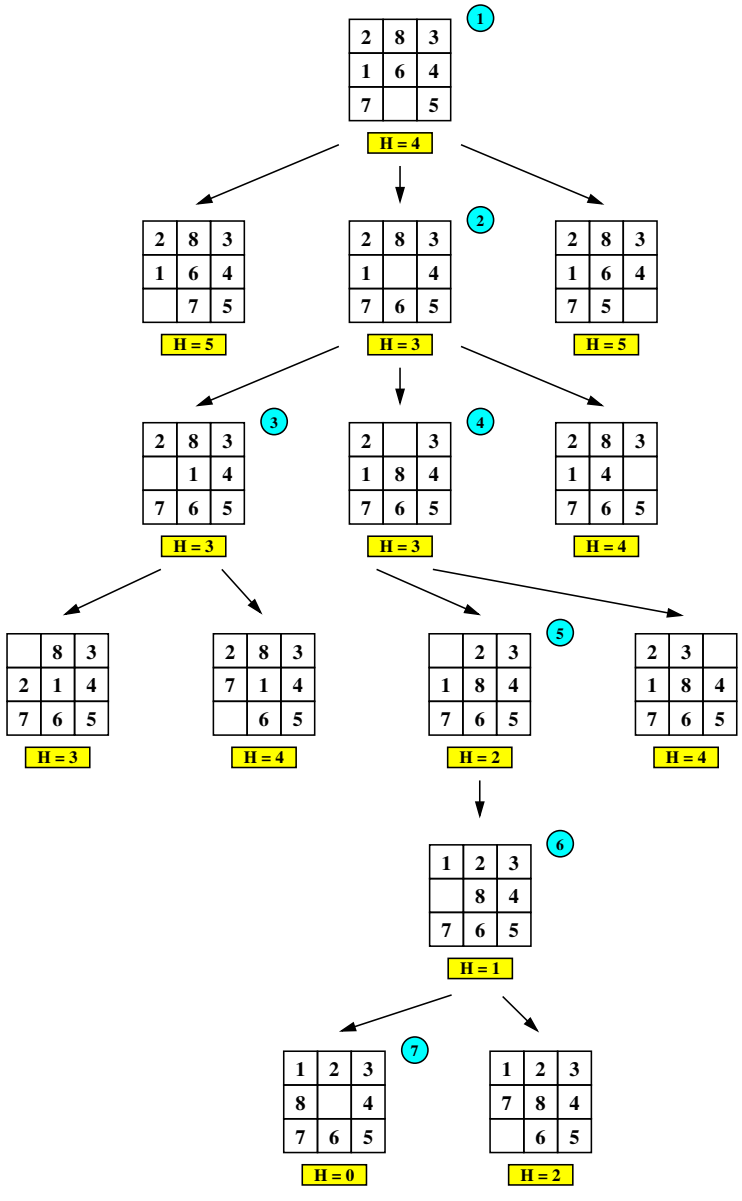
# Idea de la búsqueda por primero el mejor

- 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.

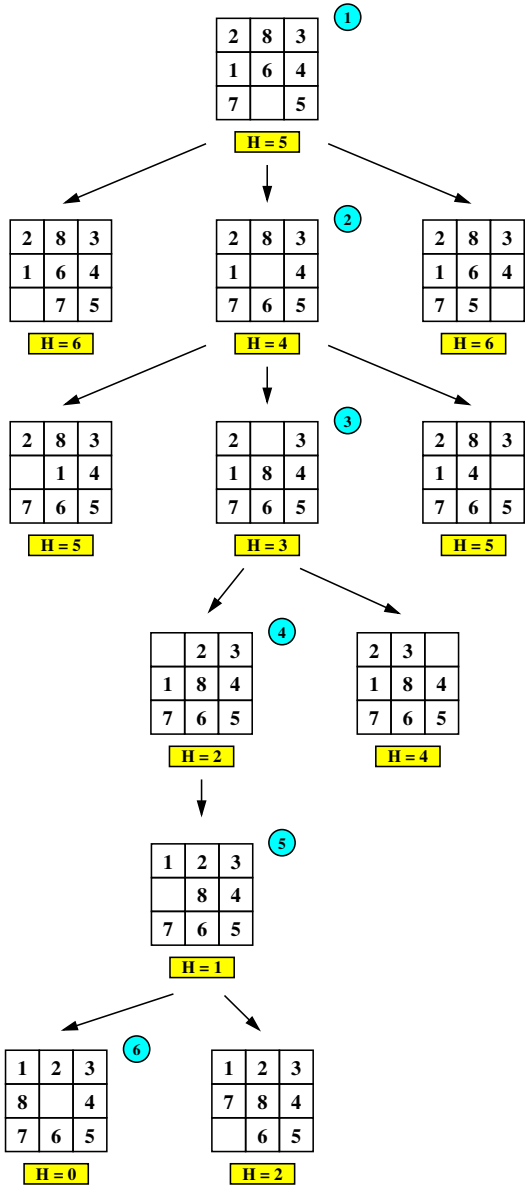
# Implementación de la búsqueda por primero el mejor

```
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.
```

# 8–puzle por primero el mejor: heurística 1



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



# Invención de heurísticas

- Recordar: estimación de la “cercanía” a un estado final
- Técnicas para encontrar heurísticas
  - Relajación del problema
  - Combinación de heurísticas
  - Uso de información estadística
- Evaluación eficiente de la función heurística

# Planificación en IA

- Planificar: encontrar una secuencia de acciones que alcanzan un determinado objetivo si se ejecutan desde un determinado estado inicial.
- Plan: secuencia de acciones que consiguen el objetivo
- Aplicaciones del mundo real:
  - Robótica
  - Fabricación mediante ensamblado de componentes
  - Misiones espaciales
- *Planning* vs *Scheduling*
  - *Planning* se refiere a la secuenciación de acciones para conseguir un objetivo y *Scheduling* pone más énfasis en la utilización eficiente de los recursos disponibles, por parte de las acciones

# El problema de la planificación en IA

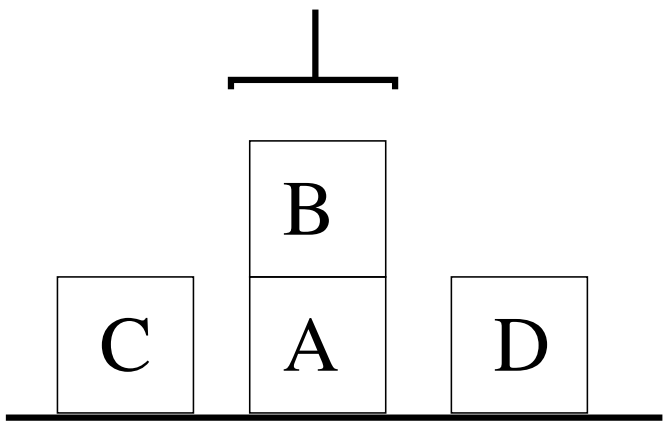
- Posibles cuestiones a abordar:
  - **Representación del mundo y de las acciones que lo transforman**
  - **Algoritmos de búsqueda de planes**
  - Minimizar los recursos consumidos por el plan
  - Tiempo en el que se realiza cada acción
  - Monitorizar la ejecución del plan, revisándolo en caso de errores o contingencias
- Por simplificar, en este tema supondremos:
  - Un número finito de estados
  - Completamente observables
  - Acciones deterministas, totalmente definidas por su especificación
  - Tiempo implícito: los acciones actúan *sin duración*
  - La planificación se realiza *a priori*
  - Hipótesis del mundo cerrado

# Planificación y búsqueda en espacio de estados

- Problema abordable con búsqueda en espacio de estados, pero en problemas de escala real hay que abordar además:
  - Descripción de los estados en el mundo real extremadamente compleja
  - Gran cantidad de posibles acciones, muchas de ellas irrelevantes para la consecución del objetivo final
  - Las acciones sólo cambian una pequeña porción del mundo (*el problema del marco*)
  - Necesitamos heurísticas *independientes* del dominio
  - La necesidad de una acción puede detectarse sin necesidad de que se haya decidido las acciones previas (*compromiso mínimo*)
  - A veces es aconsejable *descomponer* en subproblemas más simples
- Idea: usar la lógica para representar estados, acciones y objetivos, y algoritmos que operan sobre esta representación

# Ejemplo: el mundo de los bloques

- Ejemplo clásico en planificación.
- Elementos que intervienen:
  - Una superficie plana.
  - Una serie de bloques cúbicos.
  - Un brazo robotizado, que puede coger un bloque cada vez.
  - Un bloque puede estar sobre la mesa o apilado sobre otro bloque.

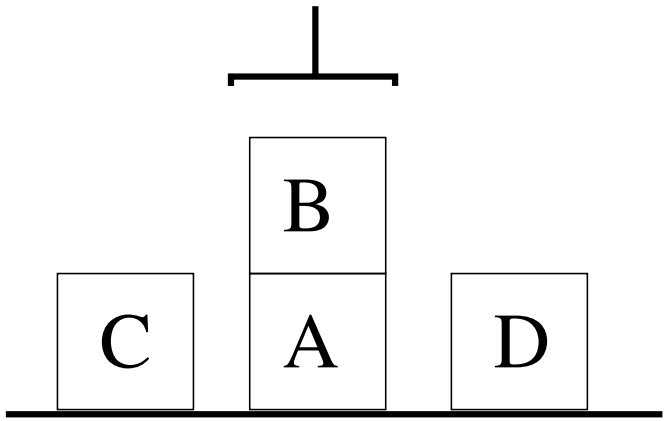


# Formalismo lógico: el lenguaje PDDL

- Un lenguaje para representar problemas de planificación:
  - Constantes: objetos del mundo (en mayúsculas)
  - Variables para representar objetos genéricos (en minúsculas)
  - Símbolos de predicados (para expresar propiedades de los objetos)
  - Símbolos para representar las acciones
- Terminología:
  - Atomos: fórmulas de la forma $P(o_1, \ldots, o_n)$, donde $P$ es un símbolo de predicado y cada $o_i$ es una constante o una variable (no hay símbolos de función)
  - Literales: átomos o negación de átomos (usaremos el símblo – para la negación)
  - Atomos y literales cerrados: sin variables
- Estados: conjunción de átomos cerrados
  - Hipótesis del mundo cerrado: los átomos que no se mencionan se suponen falsos

# Representación de estados en el mundo de los bloques

- Descripción de un estado:




**DESPEJADO(B), DESPEJADO(C), DESPEJADO(D), BRAZOLIBRE(), SOBRE(B,A), SOBRELAMESA(C), SOBRELAMESA(D), SOBRELAMESA(A)**

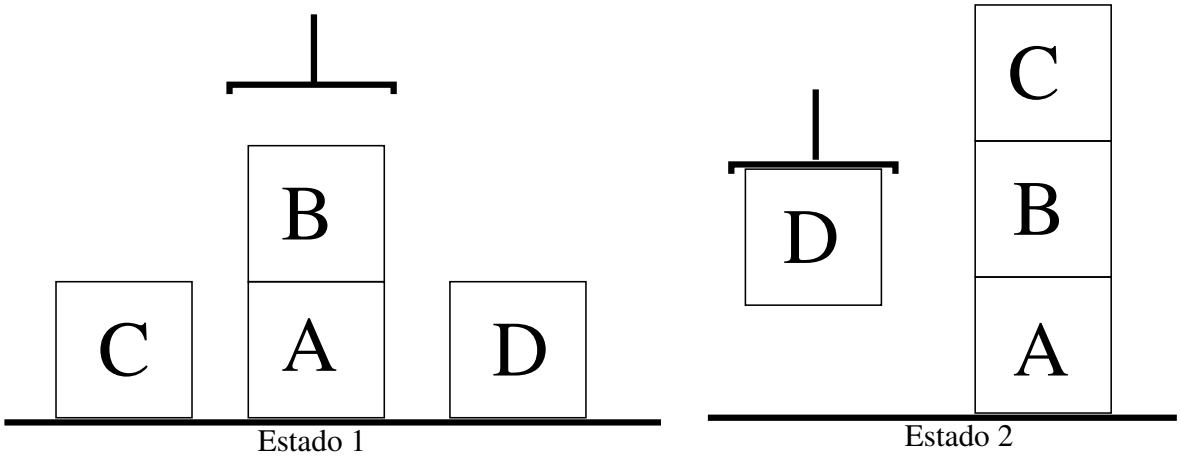
- Predicados lógicos usados en esta representación:

**DESPEJADO(x), el bloque x está despejado.**
**BRAZOLIBRE(), el brazo no agarra ningún bloque.**
**SOBRELAMESA(x), el bloque x está sobre la mesa.**
**SOBRE(x,y), el bloque x está sobre el y.**
**AGARRADO(x), el bloque X está sujeto por el brazo.**

# Representación de objetivos

- Objetivos: descripción de los estados finales.
- Los objetivos se representan como conjunción de literales (con posibilidad de usar variables)
  - Las variables en los objetivos se interpretan como existencialmente cuantificadas
- Satisfacer un objetivo:
  - Un estado satisface un objetivo si es posible sustituir las variables del objetivo por constantes de manera que el estado contiene todos los literales positivos del *objetivo instanciado* y ninguno de los negativos
  - Un estado es estado final si satisface el objetivo requerido
- Importante: verificar si un estado satisface un objetivo es un cálculo meramente simbólico

# Ejemplo de objetivos en el mundo de los bloques




- Ejemplos de objetivos:
  - **`SOBRE(B,A), SOBRELAMESA(A), -SOBRE(C,B)`** es satisfecho por el estado 1 pero no por el estado 2
  - **`SOBRE(x,A), DESPEJADO(x), BRAZOLIBRE()`** es satisfecho por el estado 1 pero no por el estado 2
  - **`SOBRE(x,A), SOBRE(y,x)`** no es satisfecho por el estado 1 pero sí por el estado 2
  - El objetivo **`SOBRE(x,A), -SOBRE(C,x)`** es satisfecho por el estado 1 pero no por el estado 2

# Descripción de esquemas de acciones

- Para intentar solucionar el *problema del marco*, sólo se especifica lo que *cambia* por la acción.
- Una acción se describe mediante:
  - Su nombre y *todas* las variables involucradas: $O(x_1, \ldots, x_n)$
  - Precondición: lista de literales que deben cumplirse para poder aplicar la acción.
  - Efectos: lista de literales que indica los cambios que se producirán cuando se aplique.

# Descripción de esquemas de acciones

- En la lista de efectos distinguimos:
  - Efectos positivos (o lista de *adición*): átomos que pasarán a ser ciertos
  - Efectos negativos (o lista de *borrado*): átomos que dejarán de ser ciertos
- El uso de variables hace que una acción usualmente represente, en realidad, un *esquema de acción*:
  - Por cada manera de sustituir las variables por constantes, tenemos una *acción* concreta (una *instancia* de la acción)

# Ejemplo: acciones en el mundo de los bloques

- Colocar un bloque sobre otro:

```
APILAR(x,y)
  Prec.: DESPEJADO(y), AGARRADO(x)
  Efec.: -DESPEJADO(y), -AGARRADO(x),
          BRAZOLIBRE(), SOBRE(x,y), DESPEJADO(x)
```

- Quitar un bloque que estaba sobre otro:

```
DESAPILAR(x,y)
  Prec.: SOBRE(x,y), DESPEJADO(x), BRAZOLIBRE()
  Efec.: -SOBRE(x,y), -DESPEJADO(x), -BRAZOLIBRE(),
         AGARRADO(x), DESPEJADO(y),
```

- Agarrar un bloque con el robot:

```
AGARRAR(x)
  Prec.: DESPEJADO(x), SOBRELAMESA(x), BRAZOLIBRE()
  Efec.: -DESPEJADO(x), -SOBRELAMESA(x), -BRAZOLIBRE(),
         AGARRADO(x)
```

- Bajar un bloque hasta la superficie:

```
BAJAR(x)
  Prec.: AGARRADO(x)
  Efec.: -AGARRADO(x),
         SOBRELAMESA(x), BRAZOLIBRE(), DESPEJADO(x)
```

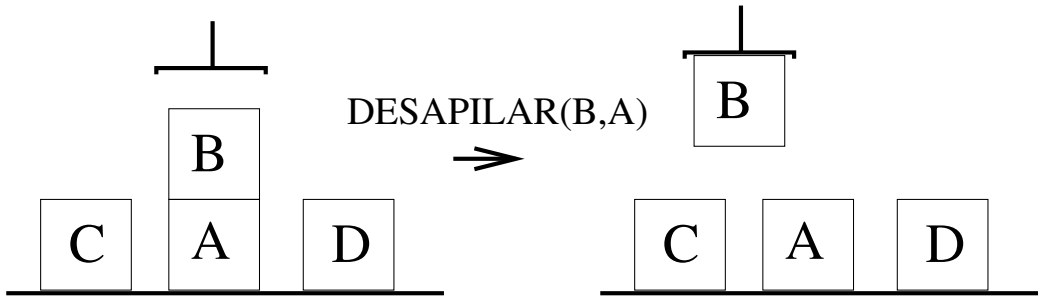
# Aplicabilidad de una acción

- Una acción es *aplicable* a un estado si éste satisface su precondición
  - Si aparecen variables en la precondición, la aplicabilidad se define *respecto de la sustitución* $\theta$ usada para satisfacer la precondición
  - Por abreviar, a veces la sustitución usada aparecerá implícita al hablar de la acción
  - Por ejemplo, hablaremos de **`DESAPILAR(A,B)`** para referirnos a **`DESAPILAR(x,y)`** con la sustitución **`[x/A,y/B]`**
  - Un mismo esquema de acción puede dar lugar a distintas acciones que son aplicables al mismo estado

# Resultado de aplicar una acción

- El resultado de aplicar una acción aplicable (respecto de una sustitución $\theta$) a un estado $\boldsymbol{E}$ es el estado resultante de:
  - Eliminar de $\boldsymbol{E}$ los átomos, instanciados por $\theta$, correspondiente a la lista de efectos negativos (si estuvieran)
  - Añadir a $\boldsymbol{E}$ los átomos, instanciados por $\theta$, correspondientes a la lista de efectos positivos (si no estuvieran)

# Ejemplo de aplicación de acción (I)



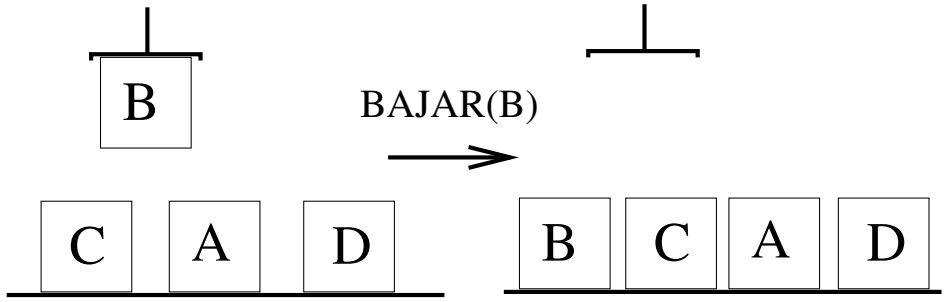

```
* Estado antes de aplicar DESAPILAR(B,A):
E = {DESPEJADO(B), DESPEJADO(C), DESPEJADO(D), BRAZOLIBRE(),
     SOBRE(B,A), SOBRELAMESA(C), SOBRELAMESA(D), SOBRELAMESA(A)}

* Precondiciones de DESAPILAR(B,A):
Prec = {SOBRE(B,A), DESPEJADO(B), BRAZOLIBRE()}
------ Condiciones satisfechas en el estado -------
------            (acción aplicable)          -------

* Efectos de DESAPILAR(B,A):
Efec- = {SOBRE(B,A), DESPEJADO(B), BRAZOLIBRE()}
Efec+ = {AGARRADO(B), DESPEJADO(A)}

* Estado después de aplicar DESAPILAR(B,A):
E' = (E - Efec-) U Efec+ =
{DESPEJADO(C), DESPEJADO(D), SOBRELAMESA(C),
 SOBRELAMESA(D), SOBRELAMESA(A), AGARRADO(B), DESPEJADO(A)}
```

# Ejemplo de aplicación de acción (II)




*** Estado antes de aplicar BAJAR(B):**
**E = {DESPEJADO(C), DESPEJADO(A), DESPEJADO(D), SOBRELAMESA(C), SOBRELAMESA(A), SOBRELAMESA(D), AGARRADO(B)}**

*** Precondiciones de BAJAR(B):**
**Prec = {AGARRADO(B)}**
**------ Condiciones satisfechas en el estado -------**
**------ (acción aplicable) -------**

*** Efectos de BAJAR(B):**
**Efec- = {AGARRADO(B)}**
**Efec+ = {SOBRELAMESA(B), BRAZOLIBRE(), DESPEJADO(B)}**

*** Estado después de aplicar BAJAR(B):**
**E′ = (E - Efec-) U Efec+ =**
**{DESPEJADO(C), DESPEJADO(A), DESPEJADO(D), SOBRELAMESA(C), SOBRELAMESA(A), SOBRELAMESA(D),SOBRELAMESA(B), BRAZOLIBRE(), DESPEJADO(B)}**

# Planes y soluciones

- Plan: secuencia de acciones
  - La primera aplicable al estado inicial y cada una de ellas aplicable al resultado de la anterior
- Solución: plan que a partir del estado inicial obtiene un estado que satisface el objetivo

# Ejemplo: cambio de rueda pinchada

- Lenguaje:
  - Objetos: **RUEDA-REPUESTO**, **RUEDA-PINCHADA**, **EJE**, **MALETERO**, **SUELO**
  - Predicado: **EN(-,-)**
- Estado inicial:

  **EN(RUEDA-PINCHADA,EJE),EN(RUEDA-REPUESTO,MALETERO)**
- Estado final:

  **EN(RUEDA-REPUESTO,EJE)**

# Acciones en el cambio de rueda pinchada

- Sacar la rueda de repuesto del maletero:

```
QUITAR(RUEDA-REPUESTO,MALETERO)
 Prec.: EN(RUEDA-REPUESTO,MALETERO)
 Efec.: EN(RUEDA-REPUESTO,SUELO), -EN(RUEDA-REPUESTO,MALETERO)
```

- Quitar la rueda pinchada del eje:

```
QUITAR(RUEDA-PINCHADA,EJE)
 Prec.: EN(RUEDA-PINCHADA,EJE)
 Efec.: -EN(RUEDA-PINCHADA,EJE), EN(RUEDA-PINCHADA,SUELO)
```

- Colocar la rueda de repuesto en el eje:

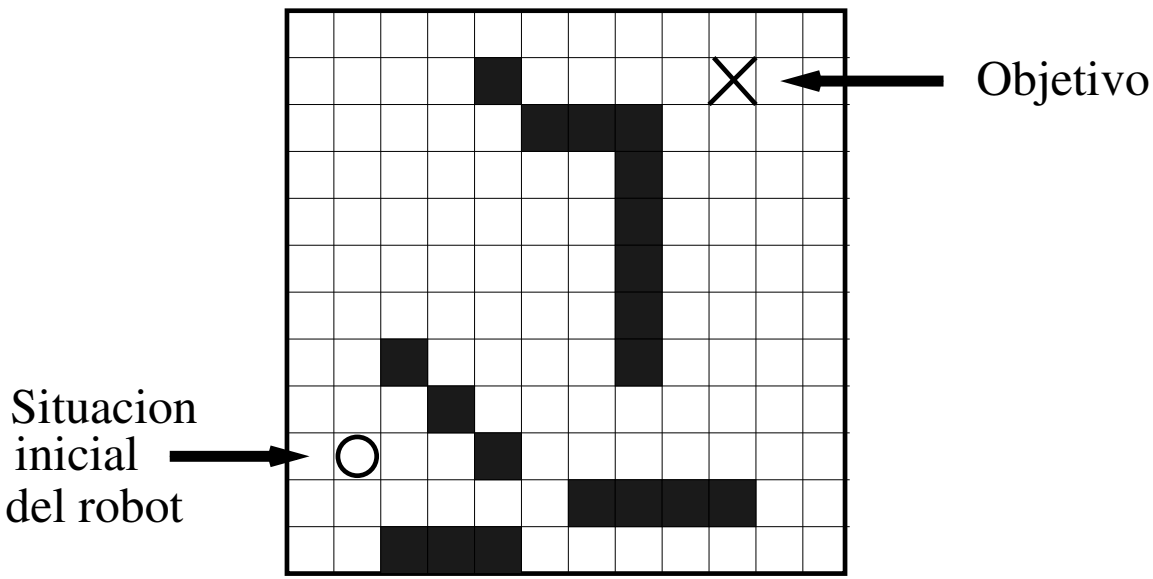
```
PONER(RUEDA-REPUESTO,EJE)
 Prec.: -EN(RUEDA-PINCHADA,EJE), EN(RUEDA-REPUESTO,SUELO)
 Efec.: -EN(RUEDA-REPUESTO,SUELO), EN(RUEDA-REPUESTO,EJE)
```

- Dejar el coche solo hasta la mañana siguiente:

```
DEJARSOLO()
 Prec.: {}
 Efec.: -EN(RUEDA-REPUESTO,SUELO), -EN(RUEDA-REPUESTO,EJE),
        -EN(RUEDA-REPUESTO,MALETERO) -EN(RUEDA-PINCHADA,EJE),
        -EN(RUEDA-PINCHADA,SUELO)
```

# Ejemplo: movimiento de robot por una rejilla

- Un robot ha de desplazarse por una rejilla, desde una posición inicial a una final
  - 8 movimientos posibles: N, S,E,O,NO,NE,SO,SE
  - En algunas de las rejillas existen obstáculos no franqueables



# Representación del problema del movimiento de robot

- Lenguaje:
  - Constantes: números que indican coordenas horizontales y verticales
  - Predicados: `ROBOT-EN(-,-)` y `LIBRE(-,-)`
- Estado inicial (casillas sin obstáculos y posición del robot):

```
LIBRE(1,1),...,LIBRE(6,2),LIBRE(11,2),...,LIBRE(12,12),
ROBOT-EN(2,3).
```

- Objetivo: `ROBOT-EN(10,11)`
- Acciones (sólo una, las siete restantes son análogos):

```
MOVER-SE(x,y)
  Prec.: ROBOT-EN(x,y), LIBRE(x+1,y-1)
  Efec.: -ROBOT-EN(x,y), -LIBRE(x+1,y-1),
          ROBOT-EN(x+1,y-1), LIBRE(x,y)
```

  - En este caso, necesitamos símbolos de función ($+$ y $-$), y que el test de satisfacibilidad maneje las nociones de numero siguiente y anterior (extendemos la semántica)

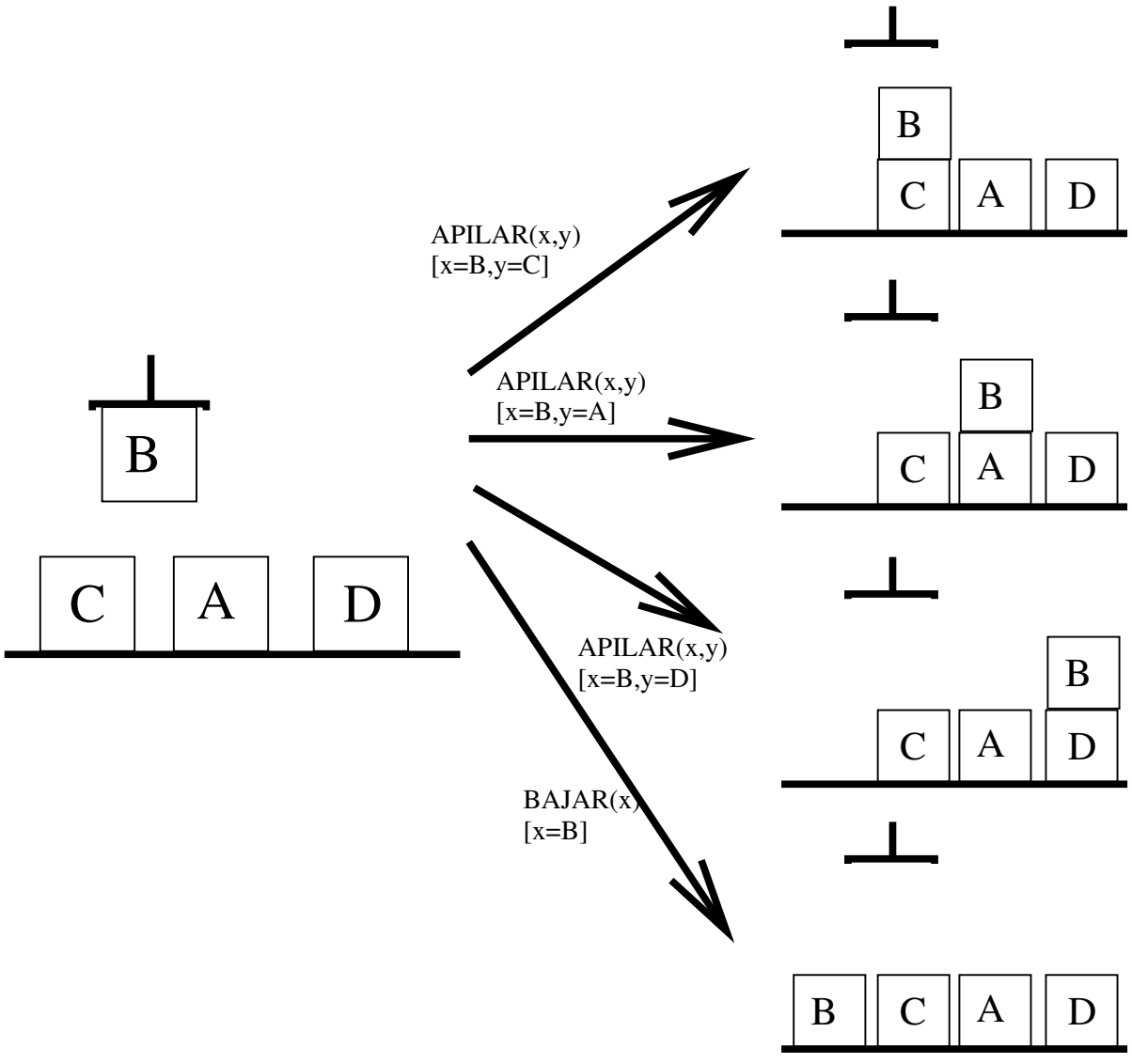
# Extensiones al formalismo PDDL

- Existen sistemas de planificación que usan un lenguaje de representación más expresivo
- Por ejemplo:
  - Variables con tipos (no es una extensión que aumente la capacidad expresiva)
  - Uso de símbolos de función
  - Manejo del símbolo de igualdad
  - Evaluación de funciones al aplicar acciones
  - Disyunciones en las precondiciones
- En general existe un compromiso entre expresividad del lenguaje y simplicidad de los algoritmos que manejan la representación

# Búsqueda en el espacio de estados

- Un problema de planificación se puede plantear como un problema de espacio de estados:
  - Estados descritos mediante listas de átomos cerrados.
  - Acciones como listas de precondiciones y efectos.
  - Función **`es-estado-final`** descrita por un objetivo.
- La búsqueda de planes podría hacerse usando los algoritmos de búsqueda ya vistos en los temas anteriores: anchura, profundidad, primero el mejor, A*,...
- El uso del formalismo lógico permite el uso de heurísticas *independientes del dominio*
  - Por ejemplo, el número de literales en el objetivo que quedan por satisfacer en un estado

# Cálculo de sucesores



# Cálculo de sucesores

- Ejemplo:

```
E = {DESPEJADO(C), DESPEJADO(A), DESPEJADO(D), SOBRELAMESA(C),
     SOBRELAMESA(A), SOBRELAMESA(D), AGARRADO(B)}
Cuatro posibles acciones aplicables:
 1), 2) y 3): O=APILAR(x,y) con THETA=[x=B,y=C],
              THETA=[x=B,y=A] y THETA=[x=B,y=D], resp.
 4): O=DEJAR(x) con sustitución THETA=[x=B].

Sucesores:

E1= { DESPEJADO(A), DESPEJADO(D), SOBRELAMESA(C), SOBRELAMESA(A),
      SOBRELAMESA(D), BRAZOLIBRE(), DESPEJADO(B), SOBRE(B,C)}

E2= { DESPEJADO(C), DESPEJADO(D), SOBRELAMESA(C), SOBRELAMESA(A),
      SOBRELAMESA(D), BRAZOLIBRE(), DESPEJADO(B), SOBRE(B,A)}

E3= { DESPEJADO(A), DESPEJADO(C), SOBRELAMESA(C), SOBRELAMESA(A),
      SOBRELAMESA(D), BRAZOLIBRE(), DESPEJADO(B), SOBRE(B,D)}

E4= { DESPEJADO(A), DESPEJADO(C), DESPEJADO(D), SOBRELAMESA(C),
      SOBRELAMESA(A), SOBRELAMESA(D), BRAZOLIBRE(), DESPEJADO(B),
      SOBRELAMESA(B)}
```

# Búsqueda hacia adelante en profundidad con heurística

```
FUNCION BUSQUEDA-EN-PROFUNDIDAD-H(ESTADO-INICIAL,OBJETIVO,ACCIONES)
 Devolver BEP-H-REC({},{},ESTADO-INICIAL,OBJETIVO,ACCIONES)

FUNCION BEP-H-REC(PLAN,VISITADOS,ACTUAL,OBJ,ACCIONES)
1. Si ACTUAL satisface OBJ, devolver PLAN
2. Hacer APLICABLES igual a la lista de acciones que sean instancias
   de una acción de ACCIONES, que sean aplicables a ACTUAL y cuya
   aplicación no resulte en un estado de VISITADOS
3. Hacer ORD-APLICABLES igual a ORDENA-POR-HEURISTICA(APLICABLES)
4. Para cada ACCION en ORD-APLICABLES
   4.1 Hacer E' el resultado de aplicar ACCION a ACTUAL
   4.2 Hacer RES igual a
         BEP-H-REC(PLAN·ACCION,VISITADOS U {E'},E',OBJ,ACCIONES)
   4.3 Si RES no es FALLO, devolver RES y terminar
5. Devolver FALLO
```

- En el algoritmo anterior, queda por especificar la función `ORDENA-POR-HEURISTICA`
  - En general, se define a traves de una heurística `H` sobre los estados, que estima el número de pasos hasta el objetivo

# Búsqueda hacia adelante en profundidad (propiedades)

- Es un algoritmo tipo *Backtracking* en el que se ordenan por heurística los sucesores del estado actual
- Propiedades:
  - Correcto, completo y siempre termina. No se garantiza la solución más corta
  - Complejidad en tiempo: exponencial
  - La complejidad en espacio es lineal respecto de la máxima profundidad del árbol de búsqueda (esto lo hace preferible a primero el mejor)
- En la práctica, su eficiencia depende de la bondad de la heurística
  - Comentaremos más sobre heurísticas posteriormente

# Búsqueda hacia atrás

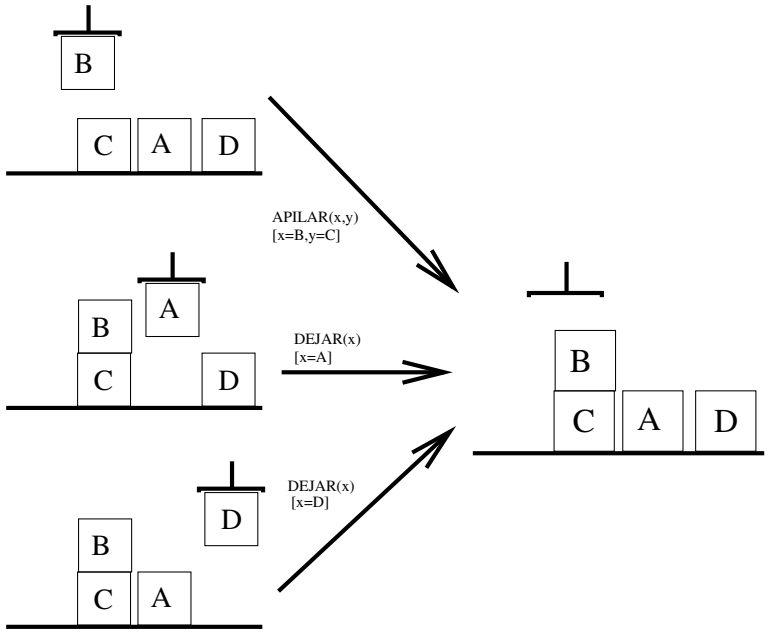
- Si la heurística no es buena, la búsqueda hacia adelante tiene el problema de la excesiva ramificación
  - Muchas acciones aplicables, pero la mayoría de ellas irrelevante para el objetivo
- Alternativa a la búsqueda hacia adelante: buscar hacia atrás, dirigidos por el objetivo
  - Se empieza en el objetivo, se aplican las acciones a la inversa y se trata de llegar al estado inicial
  - Nodos del árbol de búsqueda: objetivos
- La clave: sólo aplicar hacia atrás las acciones relevantes para el objetivo

# Acción relevante para un objetivo

- Sea ***G*** un objetivo *sin variables* (más adelante trataremos el caso con variables). Decimos que una acción ***A*** es *relevante* para ***G*** si:
  - Al menos, uno de los efectos (positivo o negativo) de la acción está en ***G*** con el mismo "signo"
  - Ninguno de los efectos (positivo o negativo) de la acción aparece en ***G*** con distinto “signo”
- Intuitivamente: una acción es relevante si podría ser la última de un plan que llevara al objetivo

# Acciones relevantes: ejemplo

- Objetivo: **{DESPEJADO(A), DESPEJADO(B), DESPEJADO(D), SOBRE(B,C)}**
  - Acciones relevantes: **APILAR(B,C)**, **DEJAR(A)** y **DEJAR(D)**




  - Ejemplo de acción no relevante para el mismo objetivo: **APILAR(A,D)** (aunque tiene a **DESPEJADO(A)** como efecto positivo sin embargo tiene como efecto negativo **−DESPEJADO(D)**)

# Cálculo de predecesores

- Si ***A*** es una acción relevante para un objetivo ***G*** (sin variables), entonces el *objetivo predecesor* de ***G*** respecto a ***A*** es $(\boldsymbol{G} - \boldsymbol{efectos}(\boldsymbol{A})) \cup \boldsymbol{precond}(\boldsymbol{A})$
- Ejemplo:

```
G = {DESPEJADO(A), DESPEJADO(B), DESPEJADO(D), SOBRE(B,C)}

Predecesor respecto de la acción relevante APILAR(B,C)

G' = (G - {-DESPEJADO(C), -AGARRADO(B), BRAZOLIBRE(),
           SOBRE(B,C), DESPEJADO(B)})
       U {DESPEJADO(C), AGARRADO(B)}
   = {DESPEJADO(A), DESPEJADO(D), DESPEJADO(C), AGARRADO(B)}
```

- Nótese que el formalismo lógico nos permite calcular predecesores de manera sencilla

# Búsqueda hacia atrás con heurística (sin variables)

```
FUNCION BUSQUEDA-HACIA-ATRÁS-H(ESTADO-INICIAL,OBJ,ACCIONES)
  Devolver BHA-H-REC({},{},ESTADO-INICIAL,OBJ,ACCIONES)

FUNCION BHA-H-REC(PLAN,VISITADOS,ESTADO-INICIAL,G-ACTUAL,ACCIONES)
1. Si ESTADO-INICIAL satisface G-ACTUAL, devolver PLAN
2. Hacer RELEVANTES igual a la lista de acciones que sean instancias
    de una acción de ACCIONES, que sean relevantes para G-ACTUAL
    y tal que el predecesor de G-ACTUAL respecto de la acción
    no sea un objetivo que contiene a alguno de VISITADOS
3. Hacer RELEVANTES-ORDENADOS
     igual a ORDENA-POR-HEURISTICA(RELEVANTES)
4. Para cada ACCION en RELEVANTES-ORDENADOS
   4.1 Hacer G' el objetivo predecesor de G-ACTUAL respecto a ACCION
   4.2 Hacer RES igual a
        BHA-H-REC(ACCION·PLAN,VISITADOS U {G'},
                  ESTADO-INICIAL,G',ACCIONES)
   4.4 Si RES no es FALLO, devolver RES y terminar
5. Devolver FALLO
```

- En el algoritmo anterior, queda por especificar la función **ORDENA-POR-HEURISTICA**
  - Se va a definir a traves de una heurística **H** sobre los objetivos, que estima el número de pasos desde el estado inicial al objetivo

# Búsqueda hacia atrás con heurística (propiedades)

- Es un algoritmo de búsqueda en profundidad en un espacio de estados (el de los objetivos)
  - Incorpora heurística para ordenar objetivos
- Propiedades
  - Correcto, completo y siempre termina. No se garantiza la solución más corta
  - Complejidad teórica: como en la búsqueda hacia adelante
- En la práctica, y nuevamente debido a la elevada ramificación, su eficiencia depende de la heurística usada

# Heurísticas para planificación basada en espacios de estados

- Una componente fundamental para la eficiencia práctica de los anteriores algoritmos es la heurística usada para ordenar los estados o los objetivos
- Esta heurística debe estimar distancia entre estados y objetivos (número de acciones necesarias)
  - En búsqueda hacia adelante: para cada estado, distancia hasta el objetivo
  - En búsqueda hacia atrás: para cada objetivo, distancia hasta el estado inicial
- Si esta estimación está por debajo del mínimo número de acciones real, diremos que la heurística es *admisible*
- Buscamos heurísticas independientes del dominio
  - Basadas en la representación lógica de estados, objetivos y acciones

# Heurísticas basadas en relajar del problema

- Heurísticas obtenidas relajando algunas restricciones del problema y calculando el número de acciones necesarias en ese problema relajado
- Algunas ideas para relajar el problema:
  - Ignorar las precondiciones y/o efectos negativos de lss acciones
  - Suponer que cada literal de un objetivo se alcanza de manera independiente: como si el número de acciones necesarias para alcanzar un objetivo fuera la *suma* del número de pasos necesario para alcanzar cada literal del objetivo
  - Suponer que el número de acciones necesarias para alcanzar un objetivo es el número de pasos *máximo* necesario para alcanzar *uno* de sus literales
  - Ignorar todas las precondiciones de las acciones
  - Ignorar determinados predicados

# La heurística $\Delta_0$

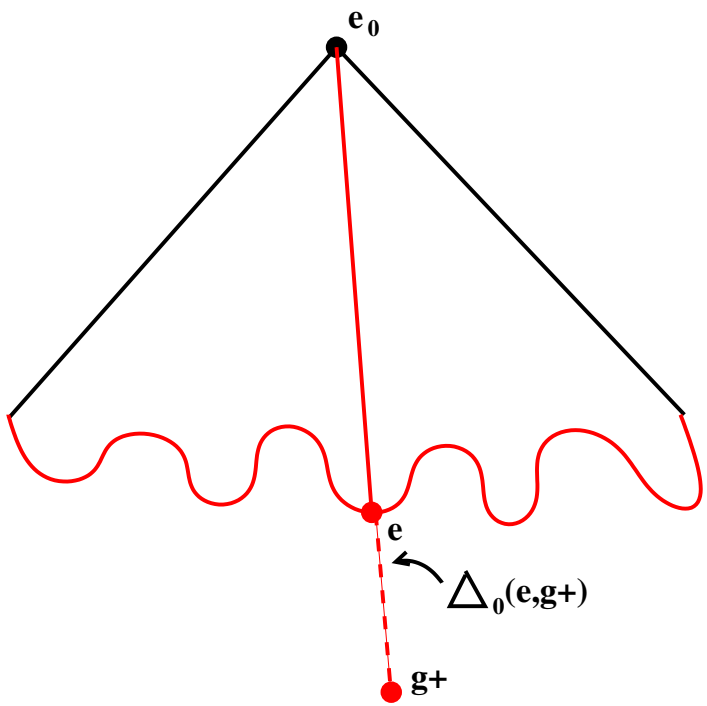
- Vamos a definir a continuación una heurística $\Delta_0$ basada en las dos primeras ideas anteriores
- Dado un estado $e$, un átomo $p$ y un objetivo $g$ que sólo tiene literales positivos sin variables, definimos recursivamente $\Delta_0(e, p)$ y $\Delta_0(e, g)$ de la siguiente manera:
  - Si $p$ aparece en $e$, $\Delta_0(e, p) = 0$
  - Si $p$ no aparece en $e$ ni en los efectos positivos de ninguna acción, $\Delta_0(e, p) = +\infty$
  - En otro caso, $\Delta_0(e, p) = min_A\{1 + \sum_{q \in precond^+(A)} \Delta_0(e, q) | p \in efectos^+(A)\}$
  - $\Delta_0(e, g) = \sum_{p \in g} \Delta_0(e, p)$
- Notación: $precond^+(A)$ y $efectos^+(A)$ denotan respectivamente los literales positivos de la precondición y de los efectos de una acción $A$

# La heurística $\Delta_0$ (propiedades)

- Intuitivamente:
  - $\Delta_0(e, p)$ cuenta el menor número de pasos necesarios para que se verifique $p$ a partir de $e$, suponiendo que las acciones no tienen precondiciones negativas ni efectos negativos
  - $\Delta_0(e, g)$ es la suma de las anteriores estimaciones para cada átomo $p \in g$
- Nótese que en general, la estimación que realiza $\Delta_0$ no es admisible
  - Aunque en cualquier caso puede funcionar bien en la práctica
  - Y además no la usaremos con A*, sino con búsqueda en profundidad
- Dado un $e$, los valores $\Delta_0(e, p)$ para cada átomo $p$ se pueden calcular con un algoritmo similar al algoritmo de caminos mínimos de Dijkstra

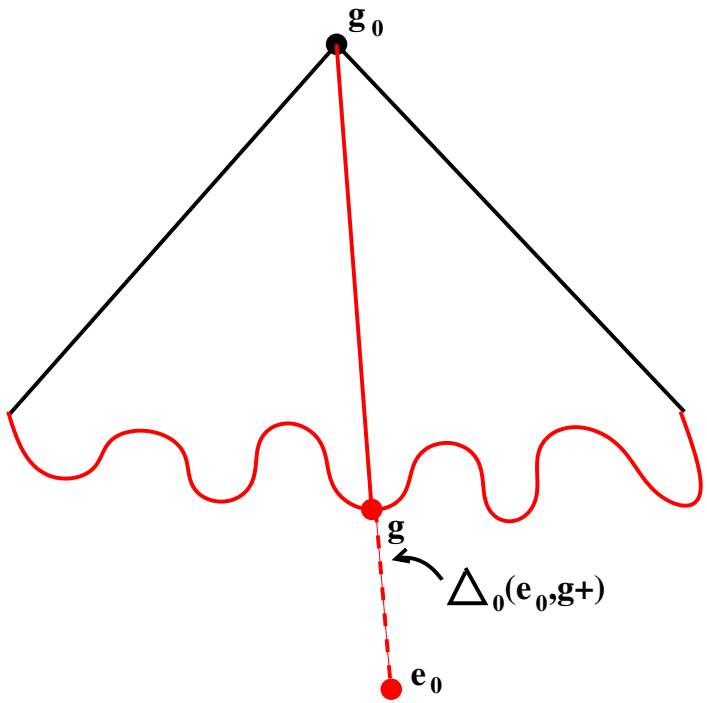
# Usando la heurística $\Delta_0$ en las búsquedas (I)

- En la búsqueda hacia adelante:
  - A cada acción ***A*** aplicable al estado actual se le asigna el valor heurístico $\Delta_0(e, g+)$, donde $g+$ es el conjunto de literales positivos del objetivo y ***e*** el estado que resulta al aplicar la acción ***A***



# Usando la heurística $\Delta_0$ en las búsquedas (II)

- En la búsqueda hacia atrás:
  - A cada acción ***A*** relevante respecto del objetivo actual se le asigna el valor $\Delta_0(e_0, g+)$, donde $e_0$ es el estado inicial, ***g*** es el predecesor correspondiente a la acción ***A*** y $g+$ es el conjunto de literales positivos de ***g***



# Usando la heurística $\Delta_0$ en las búsquedas (III)

- En el caso de que haya variables en los objetivos:
  - Si $p$ es un átomo con variables, $\Delta_0(e, p)$ se define como el mínimo de entre todos los $\Delta_0(e, \sigma(p))$, siendo $\sigma(p)$ instancia sin variables de $p$
  - Si $g$ es un objetivo con variables, $\Delta_0(e, g)$ es la suma de $\Delta_0(e, p)$ para cada literal positivo $p$ de $g$

# Otras aproximaciones a la planificación

- Planificación de orden parcial (POP)
- SATPLAN, uso de técnicas basadas en lógica proposicional
- GRAPHPLAN
- Planificación como PSR
- Planificación con recursos limitados
- Planificación con tiempo explícito
- Planificación mediante descomposición jerárquica
- Planificación en entornos con incertidumbre
- Ejecución del plan: vigilancia y replanificación
- Planificación continua
- Planificación multiagente

# Bibliografía


- Russell, S. y Norvig, P. *Artificial Intelligence (A Modern Approach)* (Prentice–Hall, 2010). Third Edition
  - Cap. 10: “Planning”.
- Russell, S. y Norvig, P. *Inteligencia Artificial (un enfoque moderno)* (Pearson Educación, 2004). Segunda edición)
  - Cap. 11: “Planificación”.
- Ghallab M., Nau D. y Traverso P. *Automated Planning: theory and practice* (Elsevier, 2004)
  - Sec. 2.3 “Classical representation”.
  - Cap. 4 “State space planning”.
  - Cap. 9 “Heuristics in planning”.