



Universidad
Zaragoza

Inteligencia Artificial (30223)

Lección 1. Resolución de
Problemas y Búsqueda



Universidad
Zaragoza

Curso 2022-2023

José Ángel Bañares

19/09/2021. Dpto. Informática e Ingeniería de Sistemas.

Índice

- Agente de **Resolución de Problemas**
 - Agente basado en objetivos
- **Formulación** del problema
 - Problemas ejemplo
- Algoritmos de Búsqueda básicos
 - Ciegos / **Sin información**

Tareas

- Búsqueda no informada (Estudiar)

Chapter 3. “**Solving problems by searching**” Artificial Intelligence a modern approach” Third Edition. Stuart Russell and Peter Norvig. Pearson 2010.

<https://www.pearsonhighered.com/assets/samplechapter/0/1/3/6/0136042597.pdf>

- Chapter 2. “**State Space Search**”. A First Course in Artificial Intelligence, 1e por Deepak Khemani. Capítulo accesible en Google Books.

- Lecture 2. **Uninformed Search**. UC Berkeley (Slides/Videos)

<http://ai.berkeley.edu/home.html>

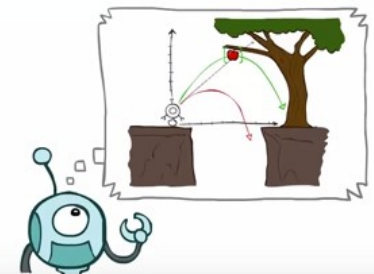
- Transparencias: moodle.unizar.es

- Grabación

- [Lección1. 14/09/2020.](#)

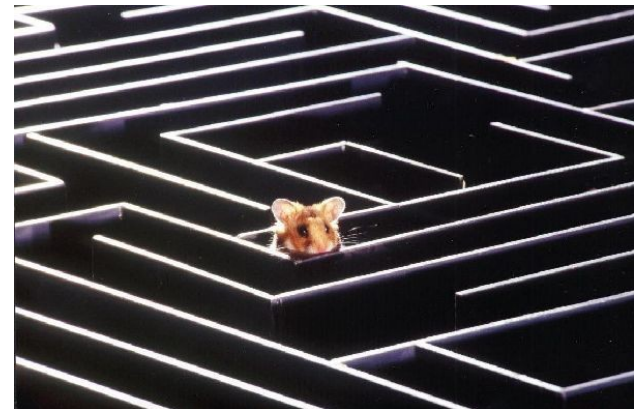
Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
 - Depth-First Search
 - Breadth-First Search
 - Uniform-Cost Search



Resolución de problemas

- La resolución de problemas es una capacidad que consideramos inteligente
- P. e. Encontrar el camino en un laberinto. Resolver un crucigrama. Jugar a un juego.
- ... El objetivo es que un programa también sea capaz de resolverlos



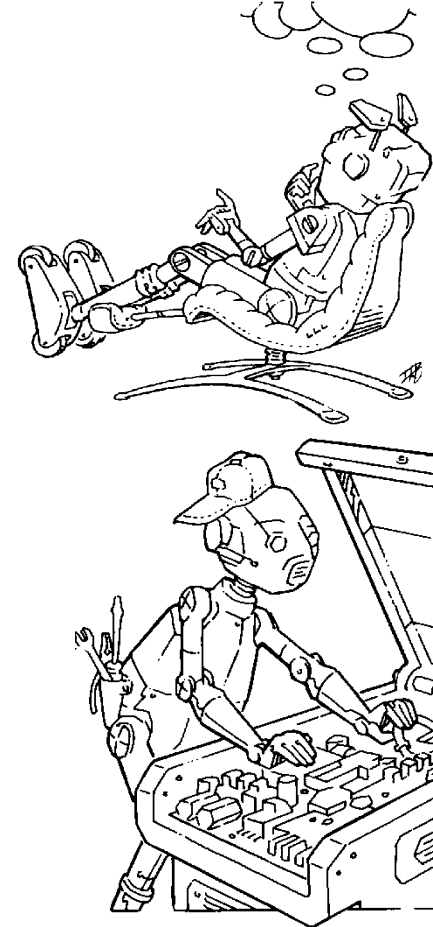
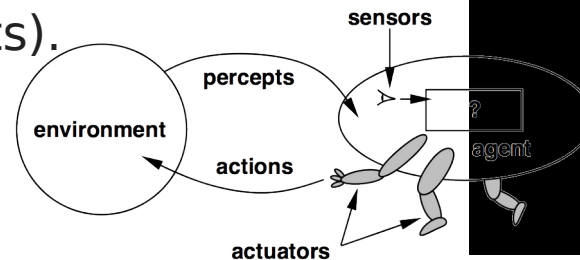
Agentes Racionales

- Este curso es sobre el diseño de agentes racionales
 - Entidad que **percibe** y **actúa** para conseguir unos **objetivos**.
 - De forma abstracta un agente es una función de las percepciones recogidas a las acciones

$$f : P^* \rightarrow A$$

- Percepción y actuación depende del entorno en el que esté situado (softbots, robots).

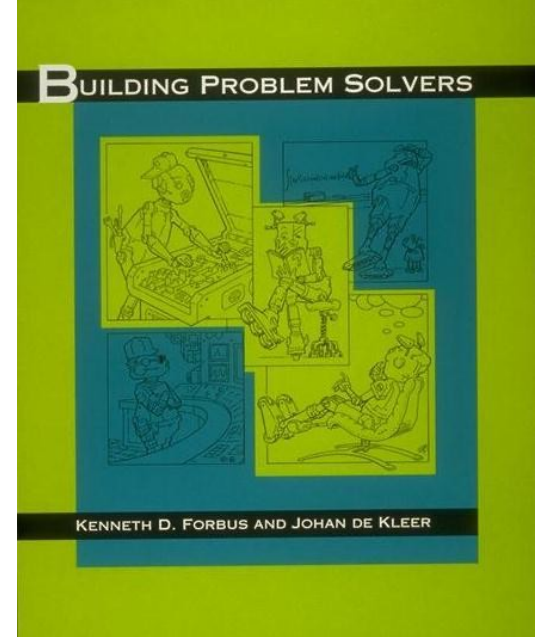
Agente: Persona o cosa que percibe su **entorno** a través de **sensores** y actúa en él a través de **actuadores**



Classical Problem Solving


*En los primeros días de la IA, se tenía la esperanza de un pequeño conjunto de grandes principios podrían ser la base para comprender la naturaleza de la inteligencia, de la misma forma que las leyes de Newton son la base para entender las interacciones de la fuerza, la materia y el movimiento. Uno de estos principios propuestos fue la **búsqueda**. Se asume que la inteligencia humana es una clase de cómputo. La inteligencia parece requerir la habilidad de intentar, comprobar si ha ido bien, e intentar algo diferente si no es el caso.*

(Building Problem Solvers , Kenneth D. Forbus and Johan de Kleer 1993)



1960 Stanford Research Institute (Shakey robot – A* y Strips)

Classical Problem Solving



The screenshot shows the Maxima interface with a command prompt and the result of an integration. The command is `(%i3) integrate (1 / (1 + x^4), x);`. The result, labeled `(%o3)`, is a complex expression involving logarithms and arctangents. A left arrow is visible on the left side of the interface.

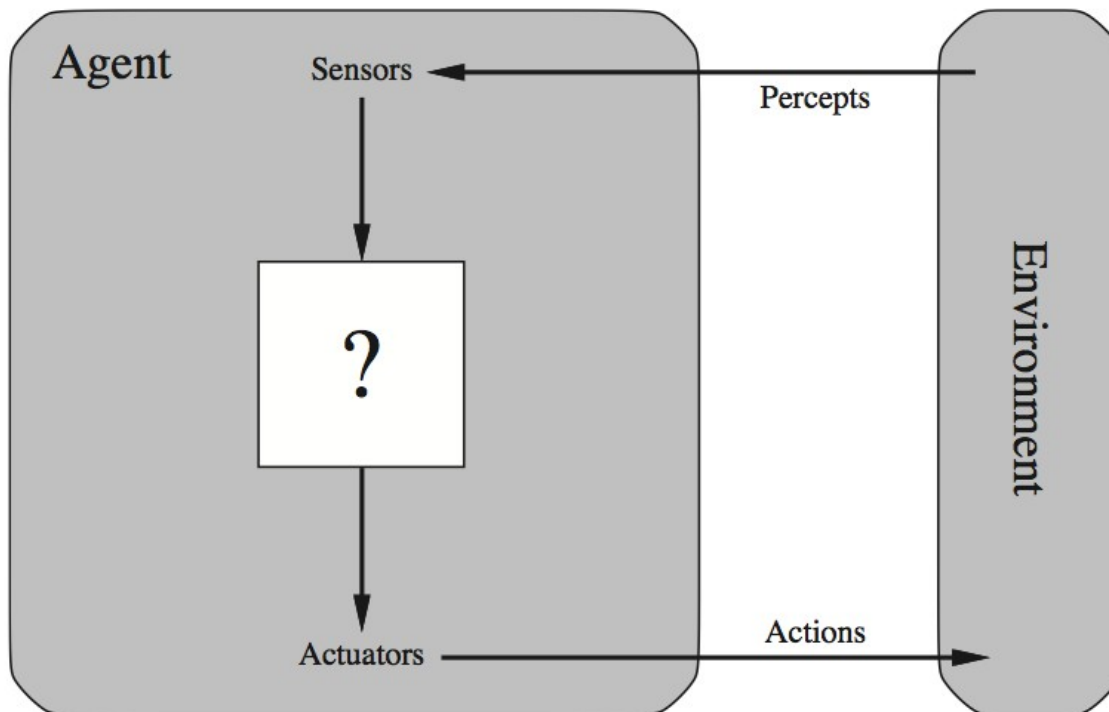
```
(%i3) integrate ( 1 / (1 + x^4), x);
```

$$\begin{aligned}
 & \frac{\log(x^2 + \sqrt{2}x + 1)}{4\sqrt{2}} - \frac{\log(x^2 - \sqrt{2}x + 1)}{4\sqrt{2}} \\
 & + \frac{\operatorname{atan}\left(\frac{2x + \sqrt{2}}{\sqrt{2}}\right)}{2\sqrt{2}} - \frac{\operatorname{atan}\left(\frac{2x - \sqrt{2}}{\sqrt{2}}\right)}{2\sqrt{2}}
 \end{aligned}$$

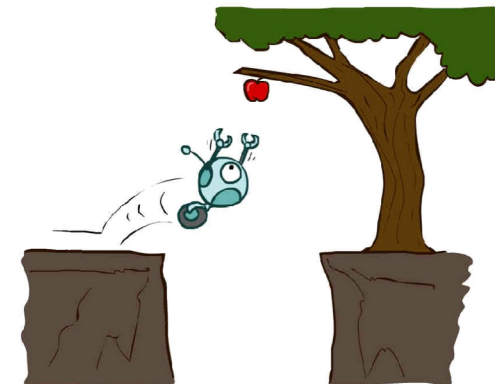
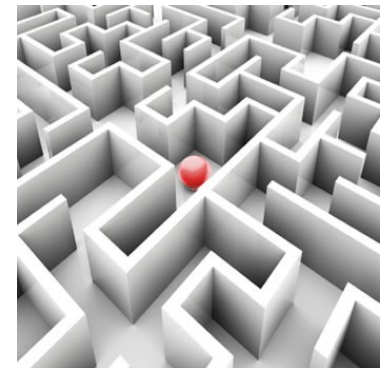
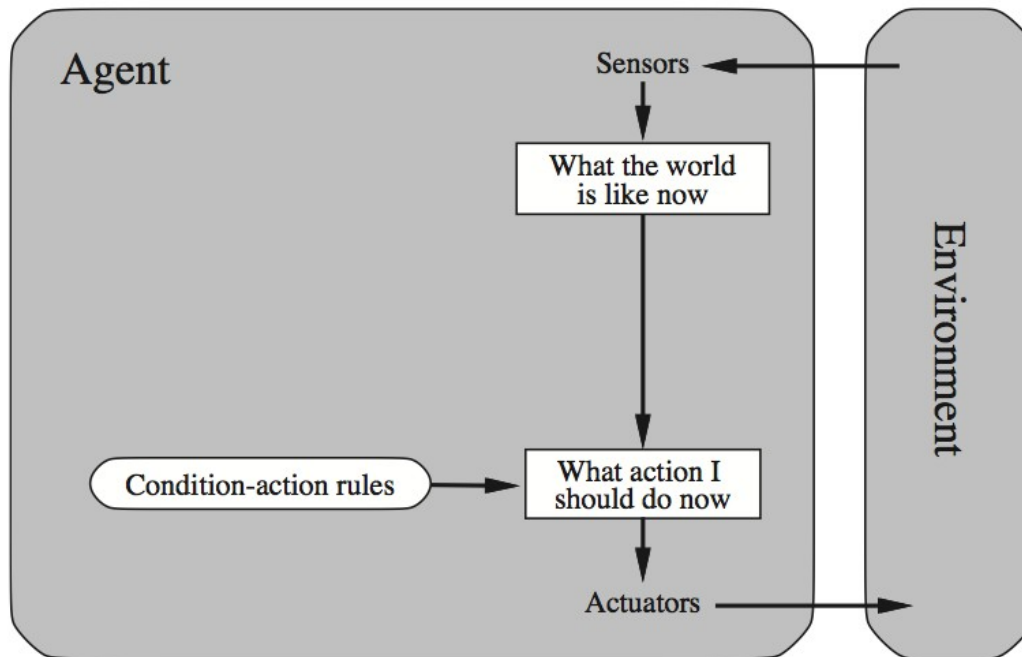
Maxima es un descendiente de **Macsyma**, el legendario sistema de álgebra computacional desarrollado a finales de 1960 en el Instituto Tecnológico de Massachusetts (MIT). Este es el único sistema basado en ese programa que está todavía disponible públicamente y con una comunidad activa de usuarios, gracias a la naturaleza del software abierto. Macsyma fue revolucionario en sus días y muchos sistemas posteriores, tales como Maple y Mathematica, se inspiraron en él.

Modelo de agente

$$f : P^* \rightarrow A$$



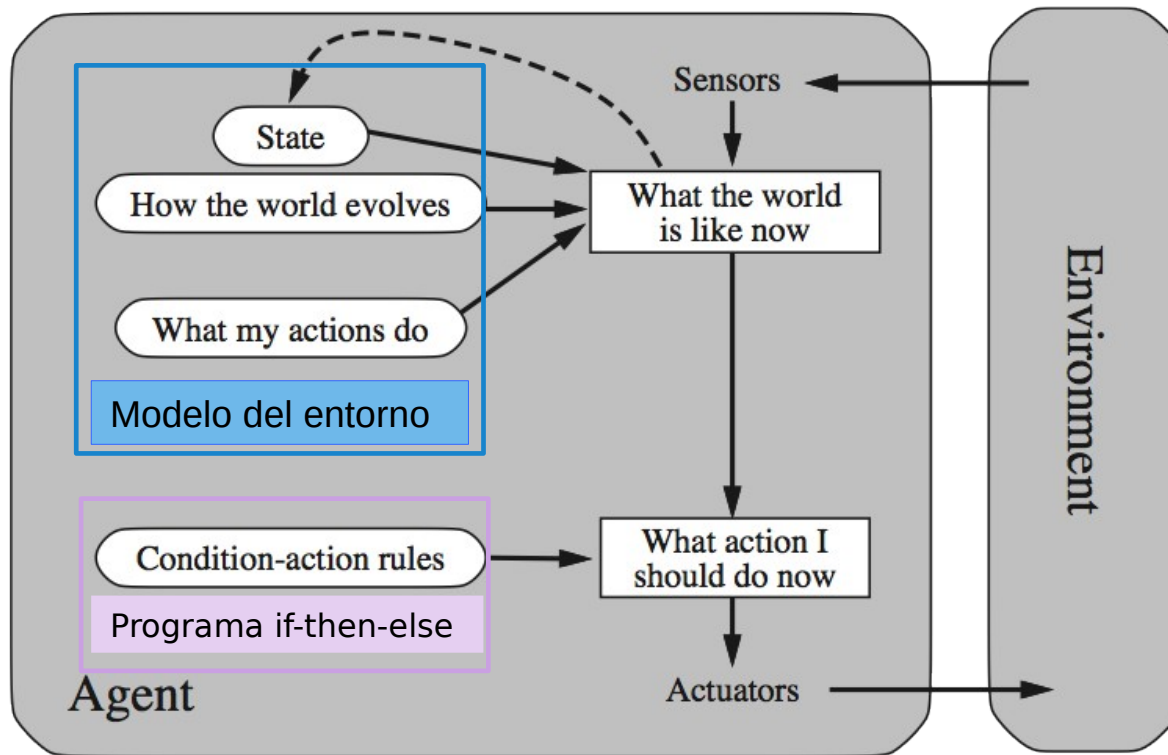
Agente Reflejo



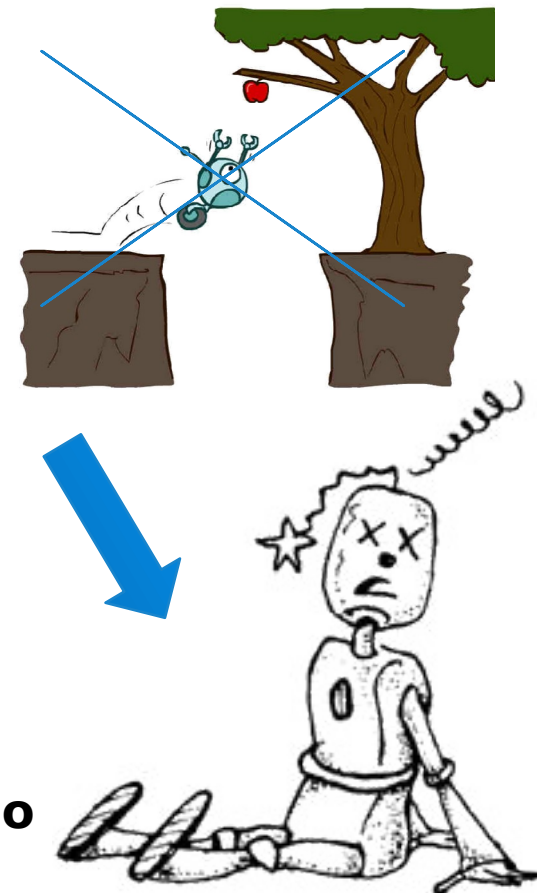
Ignora la historia

No considera las consecuencias de sus acciones

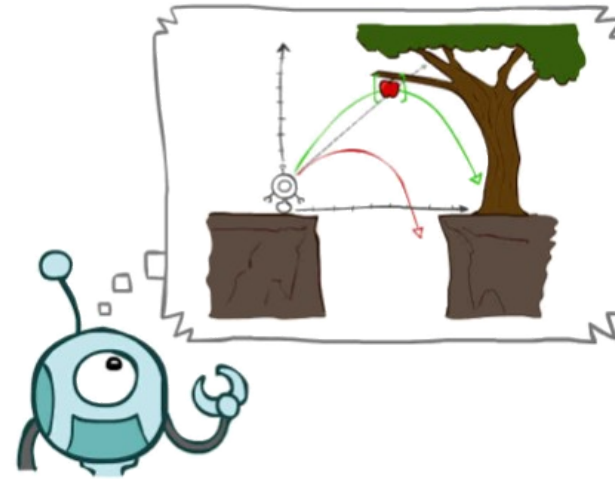
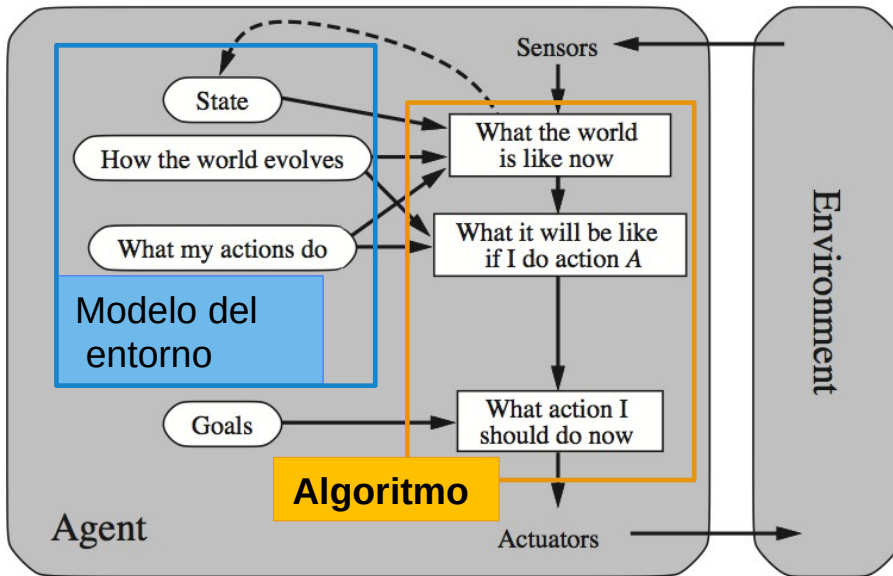
Agente reflejo basado en modelo



Modelo para representar el estado interno
(dependiente de la historia)



Agente basado en objetivo y modelo



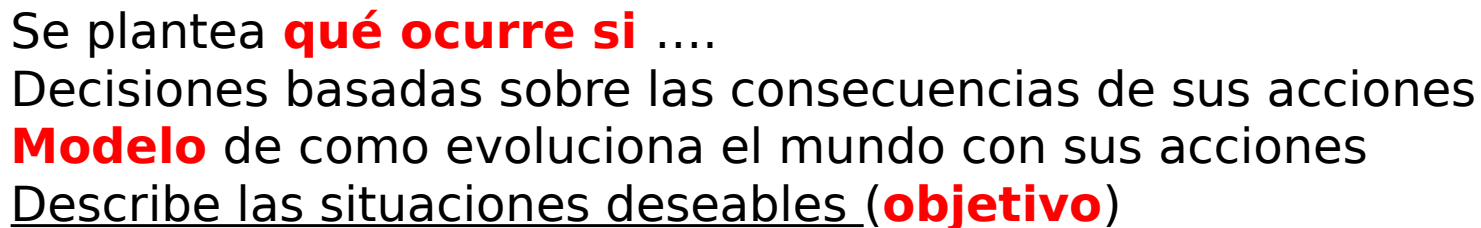
Se plantea **qué ocurre si**

Decisiones basadas sobre las consecuencias de sus acciones

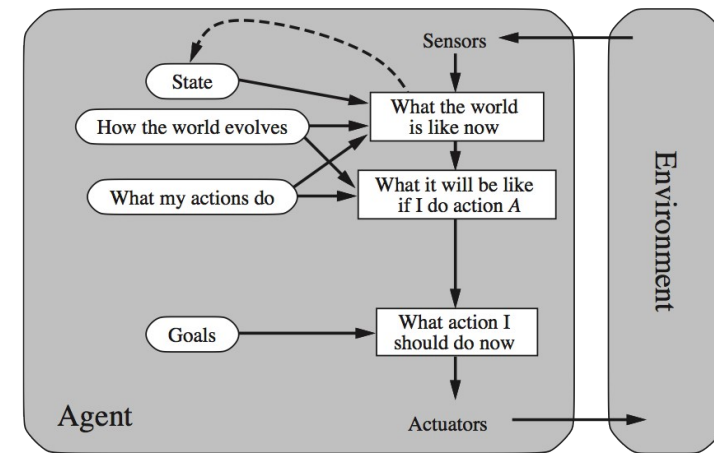
Modelo de como evoluciona el mundo con sus acciones

Describe las situaciones deseables (**objetivo**)

The diagram illustrates the architecture of an AI Agent interacting with its Environment. The Environment is represented by a large grey rounded rectangle on the right. Inside the Environment, there is a vertical stack of components: 'Sensors' at the top, followed by 'What the world is like now', 'What it will be like if I do action A', 'What action I should do now', and 'Actuators' at the bottom. A dashed arrow points from 'Sensors' to 'State' in the 'Problema Agent' box. The 'Problema Agent' box (black) contains 'State', 'How the world evolves', 'What my actions do', 'Modelo del entorno', and 'Goals'. The 'Algoritmo' box (yellow) contains 'What the world is like now', 'What it will be like if I do action A', and 'What action I should do now'. The 'Plan' box (green) is at the bottom right. Arrows show the flow of information: from 'Sensors' to 'What the world is like now', from 'What the world is like now' to 'What it will be like if I do action A', from 'What it will be like if I do action A' to 'What action I should do now', from 'What action I should do now' to 'Actuators', and from 'Actuators' to the 'Plan' box. The 'Plan' box then points to the 'Environment'. The 'Problema Agent' box also has arrows pointing to 'What the world is like now' and 'What it will be like if I do action A'. The 'Modelo del entorno' box is a blue rectangle within the 'Problema Agent' box. The 'Goals' box is a white oval within the 'Problema Agent' box. The 'State' box is a white oval within the 'Problema Agent' box. The 'How the world evolves' box is a white oval within the 'Problema Agent' box. The 'What my actions do' box is a white oval within the 'Problema Agent' box.



Agente de resolución de problemas



- Los agentes más sencillos son **agentes reflejos** que asocian una acción a cada estado.
 - Estos agentes no son viables cuando hay demasiados estados y/o el proceso de aprendizaje llevaría mucho tiempo.
- Los **agentes de resolución de problemas** son **agentes basados en objetivos**
 - **Proceso** que **elige y organiza acciones** anticipando cual será su resultado
 - Las acciones van dirigidas a conseguir un objetivo (preestablecido) en el futuro.

¿Cómo automatizar la resolución de problemas?

- Tres aspectos a considerar
 - **Modelos conceptuales/Modelos matemáticos** para **formalizar** la clase de problemas (Modelo de estados, modelos probabilísticos, ...)
 - **Lenguajes de representación** atómica, factorizada, estructurada, ...
 - **Algoritmos**, para resolver los problemas representados por los modelos

¿Por qué un modelo Conceptual?

- Modelo conceptual: La forma de describir *los elementos del problema*
- Bueno para:
 - Explicar los conceptos básicos
 - **Clarificar que asumimos: Restricciones impuestas al modelo**
 - Analizar requisitos
 - Probar propiedades
- No considera:
 - Algoritmos eficientes y aspectos computacionales



Modelo Conceptual: Agente de resolución de problemas

- Pasos generales para resolver problemas
 - Formulación del **objetivo**
 - ¿Cuales son los **estados** del mundo que queremos **alcanzar**?
 - Formulación del **problema**
 - ¿Qué **estado** y **acciones** vamos a considerar para alcanzar el objetivo?
 - **Búsqueda**
 - Determinar la posible **secuencia de acciones** que nos llevaran al objetivo.
 - Ejecutar
 - Realizar las acciones de la secuencia

Problema

Estados
Acciones
Estado Inicial
Objetivo



Formalización Resolución

Problemas (**espacio de estados**)

■ Modelo de Estados Restringido: Sistemas de transición de estados/Sistemas de eventos discretos (Σ)

- Espacio de estados finito y discreto S
- Estado inicial $s_0 \in S$
- Un conjunto de estados objetivos $G \subseteq S$
- Acciones aplicables $A(s) \subseteq A$ en cada estado $s \in S$
- Una función transición $f(s, a)$ para $s \in S$ y $a \in A(s)$
- Y una función de coste $c(a, s) > 0$

■ **Una solución** es una secuencia aplicable de acciones $a_i, i = 0, \dots, n$ que lleva desde el estado inicial s_0 al estado objetivo $s \in SG$; es decir, $s_{n+1} \in SG$ y para $i = 0, \dots, n$

$$s_{i+1} = f(a_i, s_i) \text{ y } a_i \in A(s_i)$$

■ **La solución óptima** minimiza el coste

$$\sum_{i=0}^n c(a_i, s_i)$$



Formalización Resolución

Problemas (**espacio de estados**)

■ Modelo de Estados Restringido

■ Restringido porque Asumimos

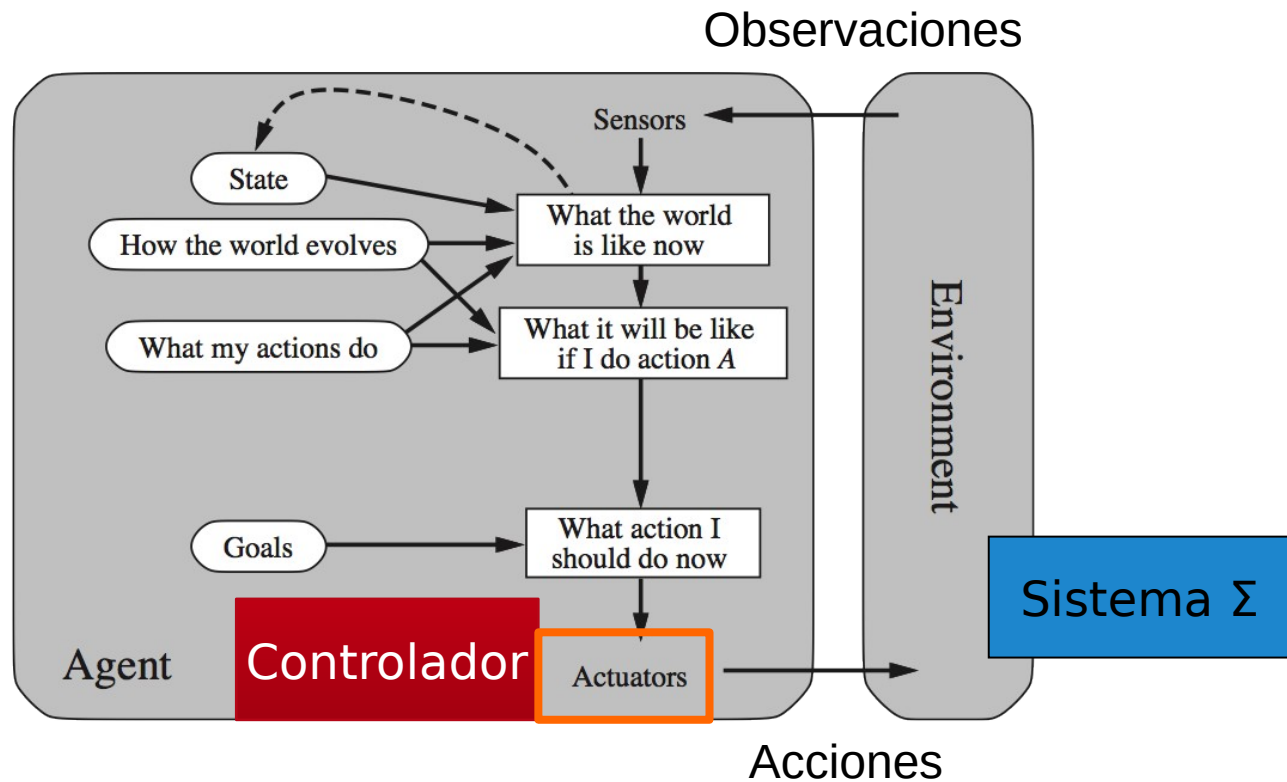
- Estados **finito**
- El Σ es completamente **observable**. Conocimiento completo del estado
- El Σ es **determinista**. Para cada estado, si aplicamos una acción sólo hay un posible estado sucesor.
- El Σ es **estático**. No hay eventos que cambien el estado del sistema
- Objetivos **restringidos**: Los objetivos son un estado o un conjunto de estados objetivos.
- La solución es una secuencia ordenada de acciones
- Las acciones no tienen duración. No se representa el tiempo explícitamente

■ Surgen otros modelos interesantes si relajamos las restricciones.

Un sistema de transición de estados define un Grafo

- Un sistema de transición de estados Σ se puede representar por un grafo dirigido etiquetado $G = (N_G, E_G)$ donde:
 - Los nodos corresponden a los estado in S , es decir $N_G = S$;
y
 - Hay un arco de $s \in N_G$ a $s' \in N_G$, es decir $s \rightarrow s' \in E_G$, con la etiqueta $a_i \in (A)$ si y solo sí hay una función de transición $s' = f(a_i, s)$ y $a_i \in A(s_i)$

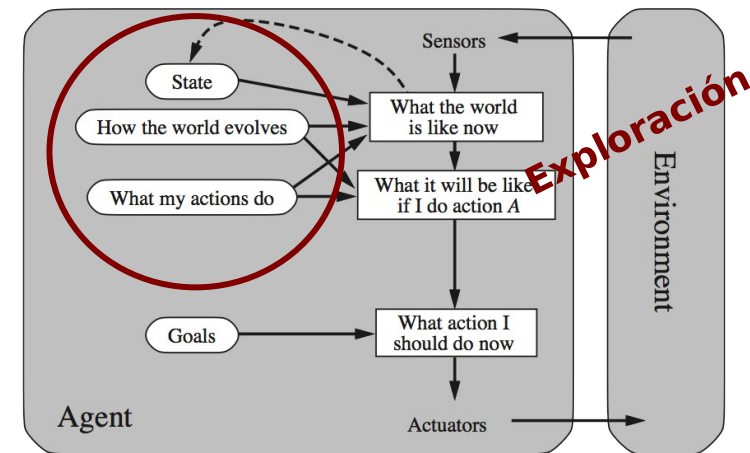
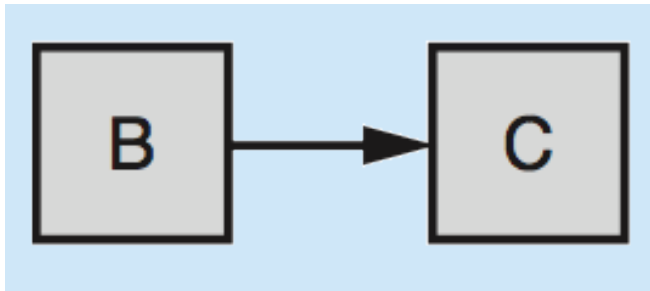
Plan y ejecución del plan



Agente de resolución de problemas

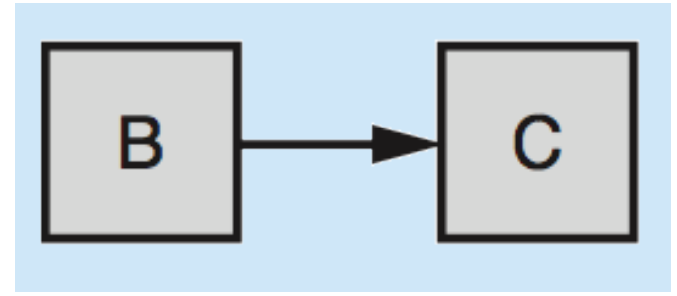
Representación de resolución de problemas

- Los agentes de resolución de problemas son los **agentes dirigidos por objetivos** que utilizan **representaciones atómicas (Cada estado un símbolo)**.
 - Se identifican los estados sin ninguna estructura, p.e **estado A, B, C**



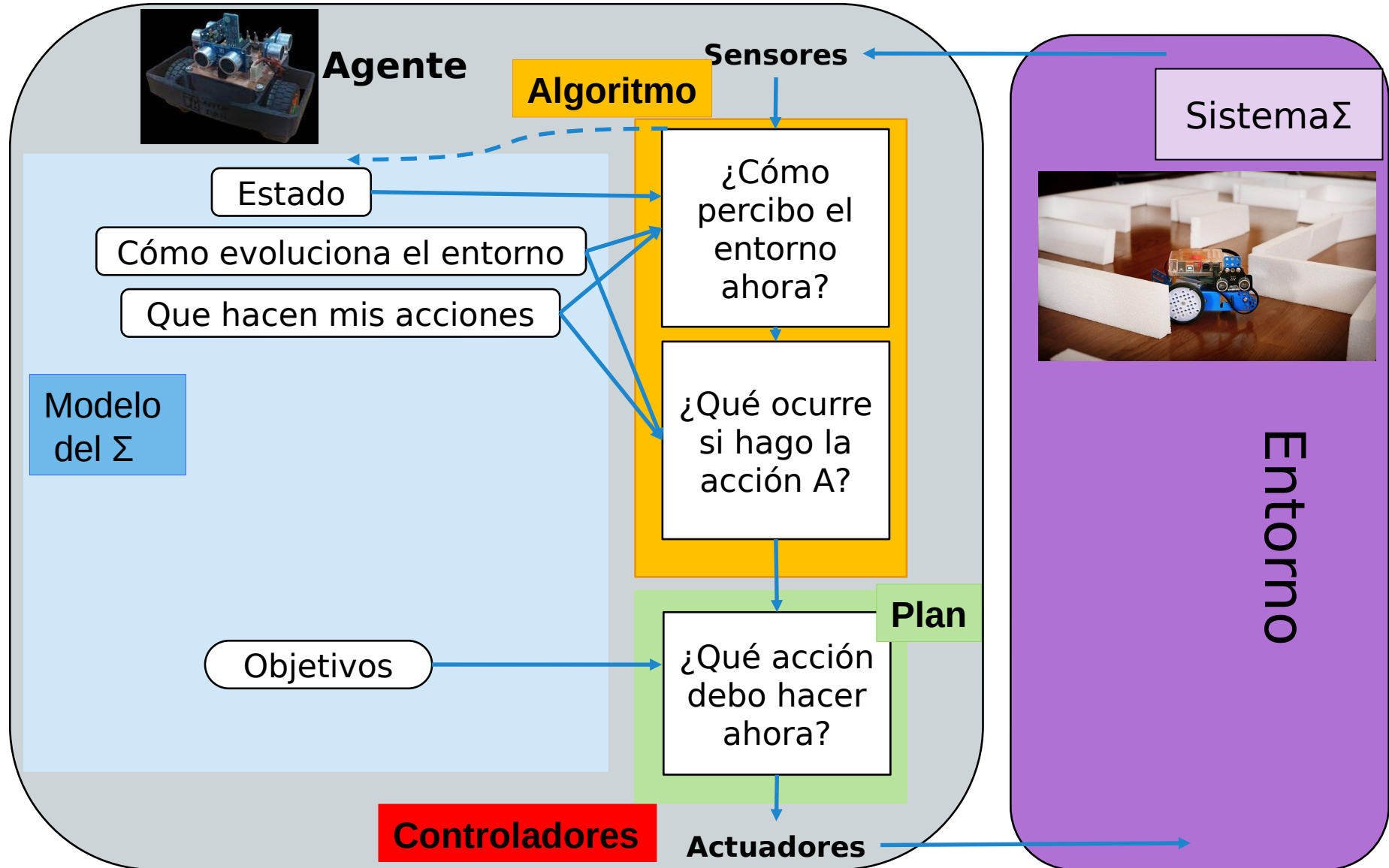
- El modelo informa al algoritmo **si dos estados son iguales o no** y las **acciones aplicables a un estado**, y los **estados sucesores**, y **si un estado es objetivo**.
- Agentes dirigidos por objetivos** que utilizan **representaciones más sofisticadas** (factorizadas, estructuradas), se denominan **agentes de planificación**.

Representación de resolución de problemas

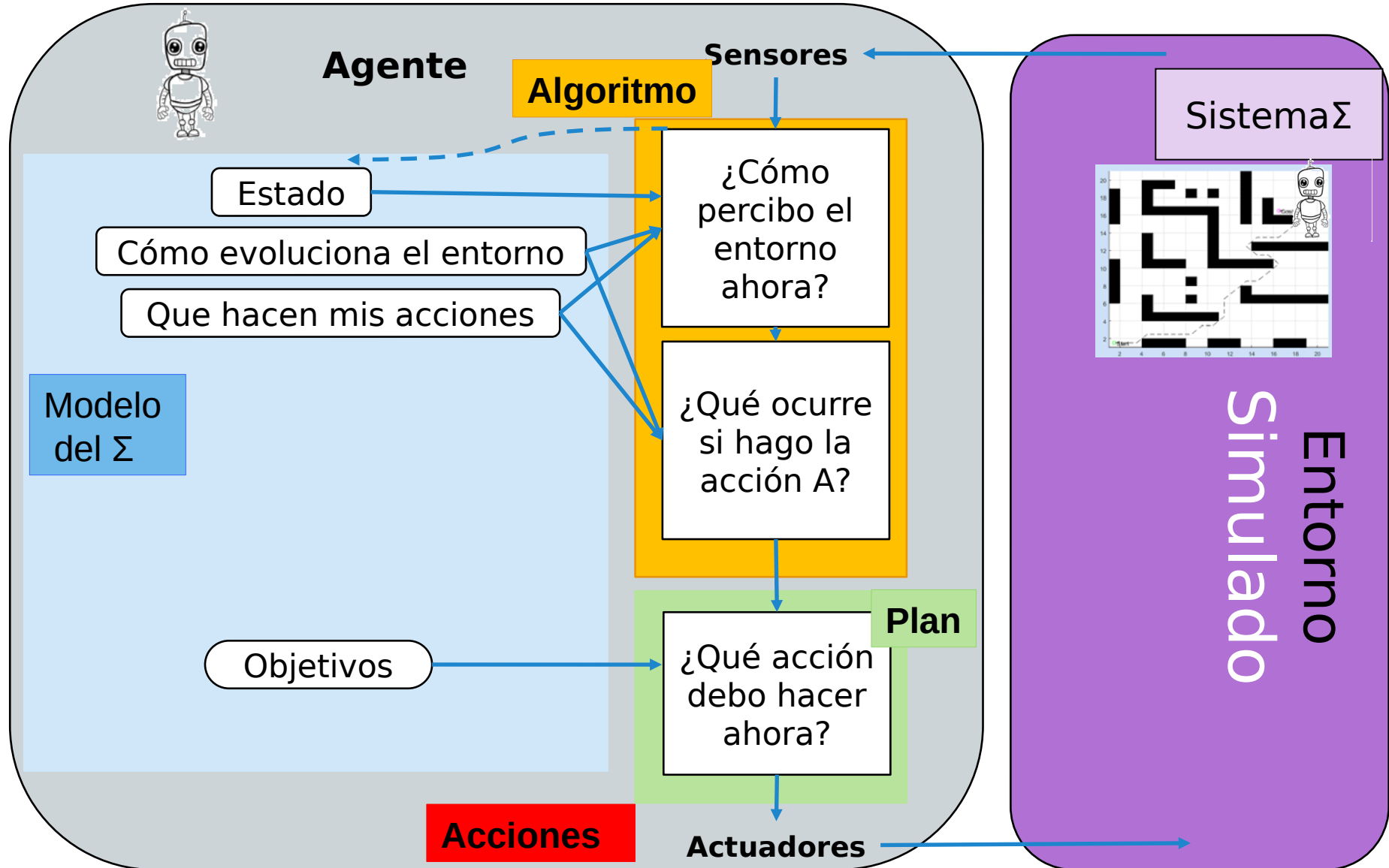


- Los agentes de resolución de problemas son los **agentes dirigidos por objetivos** que utilizan **representaciones atómicas (Cada estado un símbolo).**
- A lo largo del curso, y en muchos ejemplos de agente de resolución de problemas de IA ¡**Simulamos el entorno!**

Ag. Resolución Pb. Dirigido objetivos



Ag. Resolución Pb. Dirigido objetivos



Ag. Resolución Pb. Dirigido objetivos



Agente

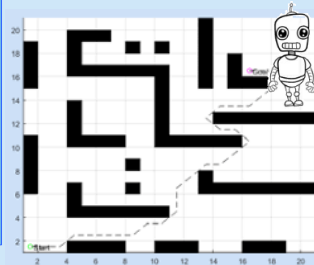
Algoritmo

Estado

Cómo evoluciona el entorno

Que hacen mis acciones

Modelo
del Σ
(Simulación
del entorno)



Objetivos

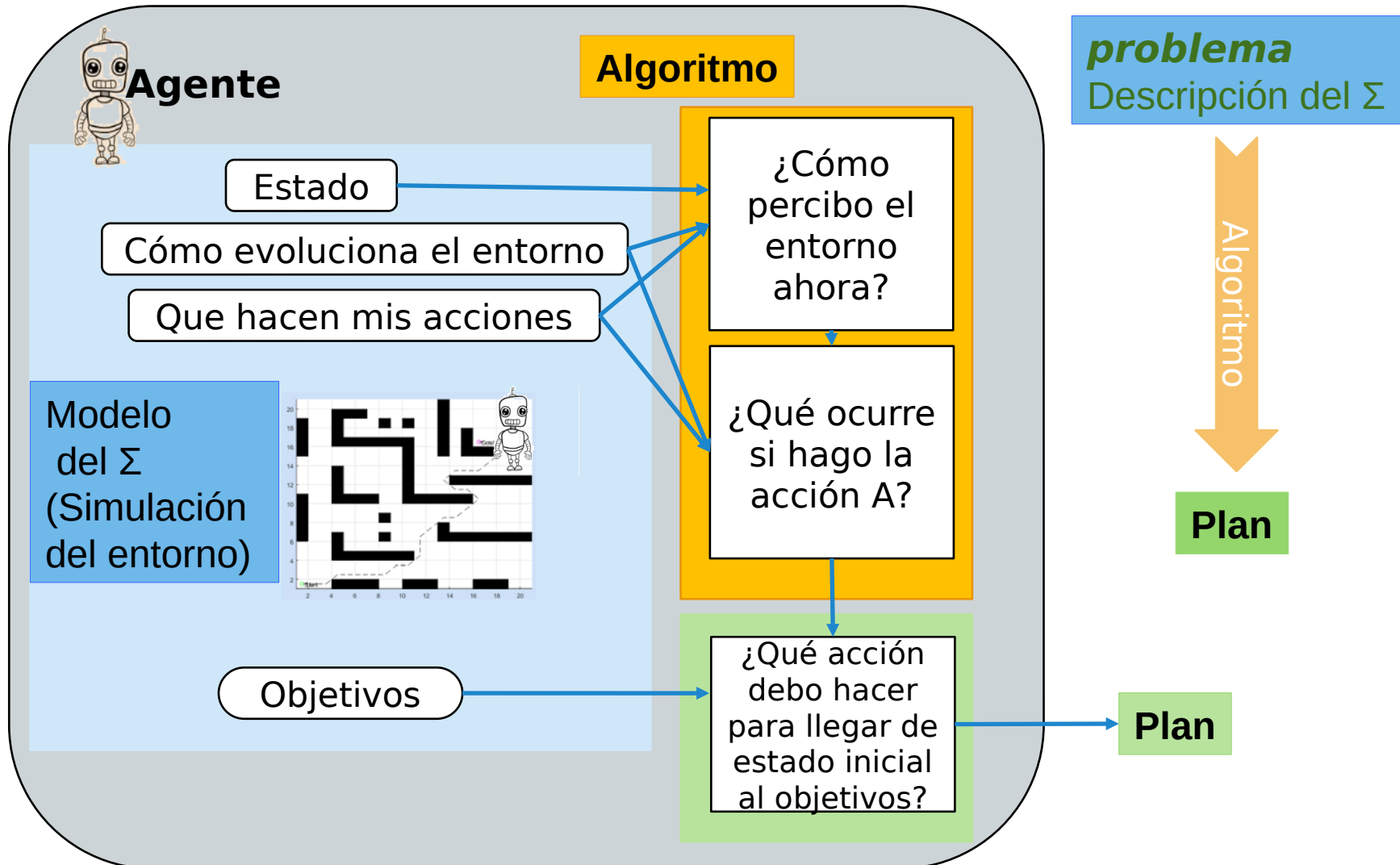
¿Cómo
percibo el
entorno
ahora?

¿Qué ocurre
si hago la
acción A?

¿Qué acción
debo hacer
para llegar de
estado inicial
al objetivos?

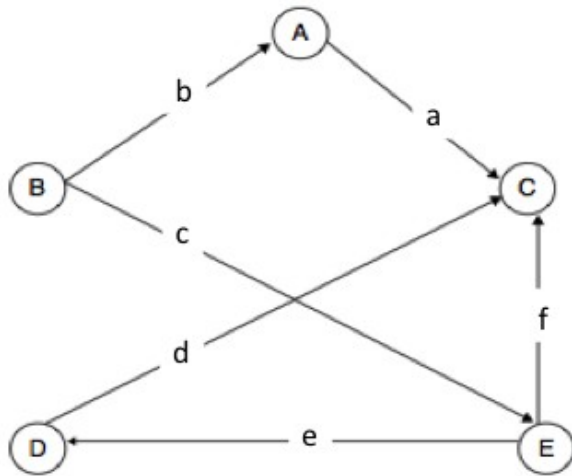
Plan

Ag. Resolución Pb. Dirigido objetivos



function **BÚSQUEDA** (*problema*) returns **solución** o fallo

Representación Espacio de estados (¿IA?)



| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 0 | a | 0 | 0 |
| B | b | 0 | 0 | 0 | c |
| C | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | d | 0 | 0 |
| E | 0 | 0 | f | e | 0 |

Matriz de adyacencia A

Caminos

$$R = A + A^2 + A^3 + A^4$$

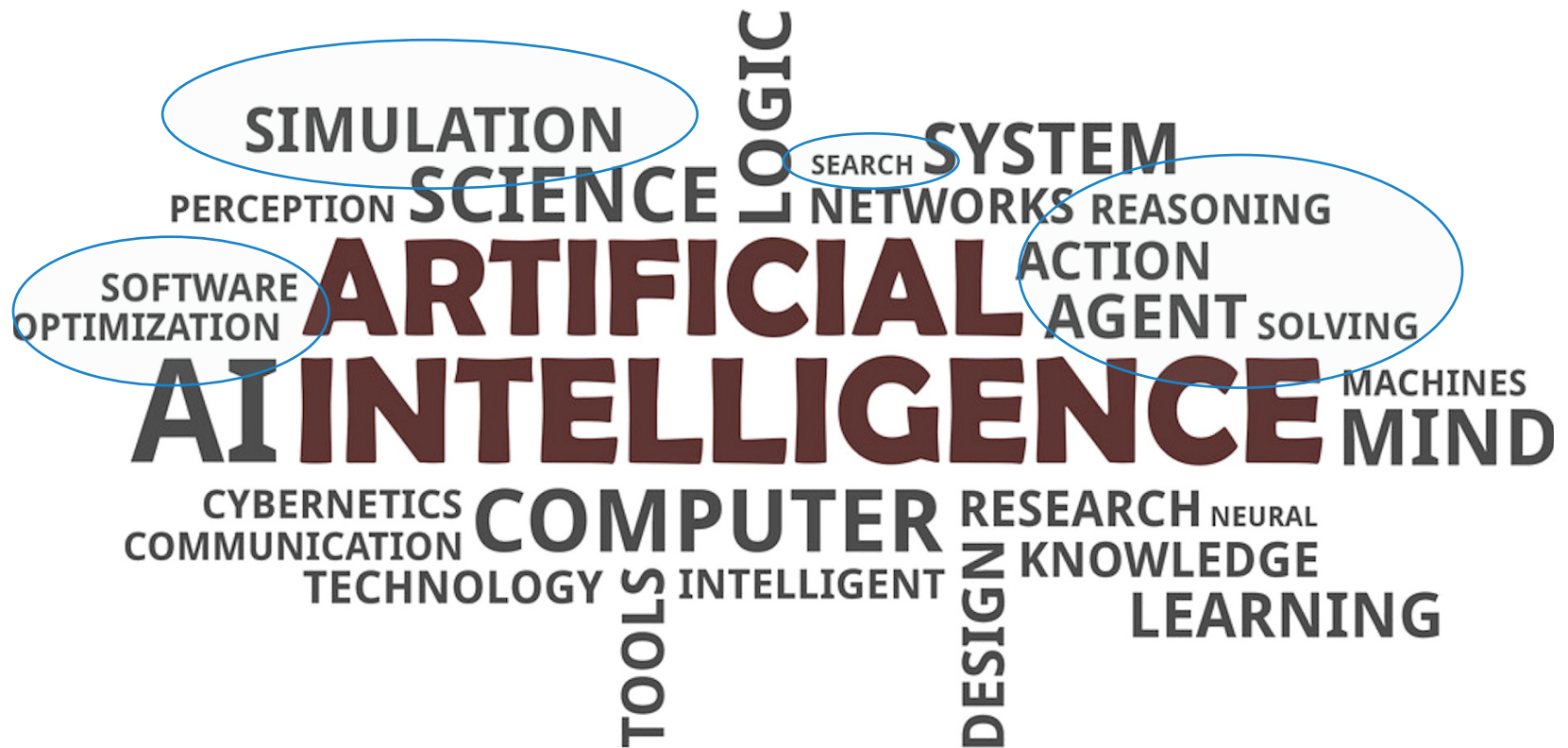
| | A | B | C | D | E |
|---|----------|---|------------------|-----------|---|
| A | 0 | 0 | a | 0 | 0 |
| B | <u>b</u> | 0 | <u>ba+cf+ced</u> | <u>ce</u> | c |
| C | 0 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | d | 0 | 0 |
| E | 0 | 0 | <u>f+ed</u> | e | 0 |

IA

- Modelado de Sistemas con espacios de estados de gran escala
- **No se puede representar todo el espacio de estados en memoria**
- **La definición del problema (Modelo del Sistema), va generando el espacio de estados durante la exploración.**

Modelo
del Σ
(Simulación del entorno)

Artificial Intelligence



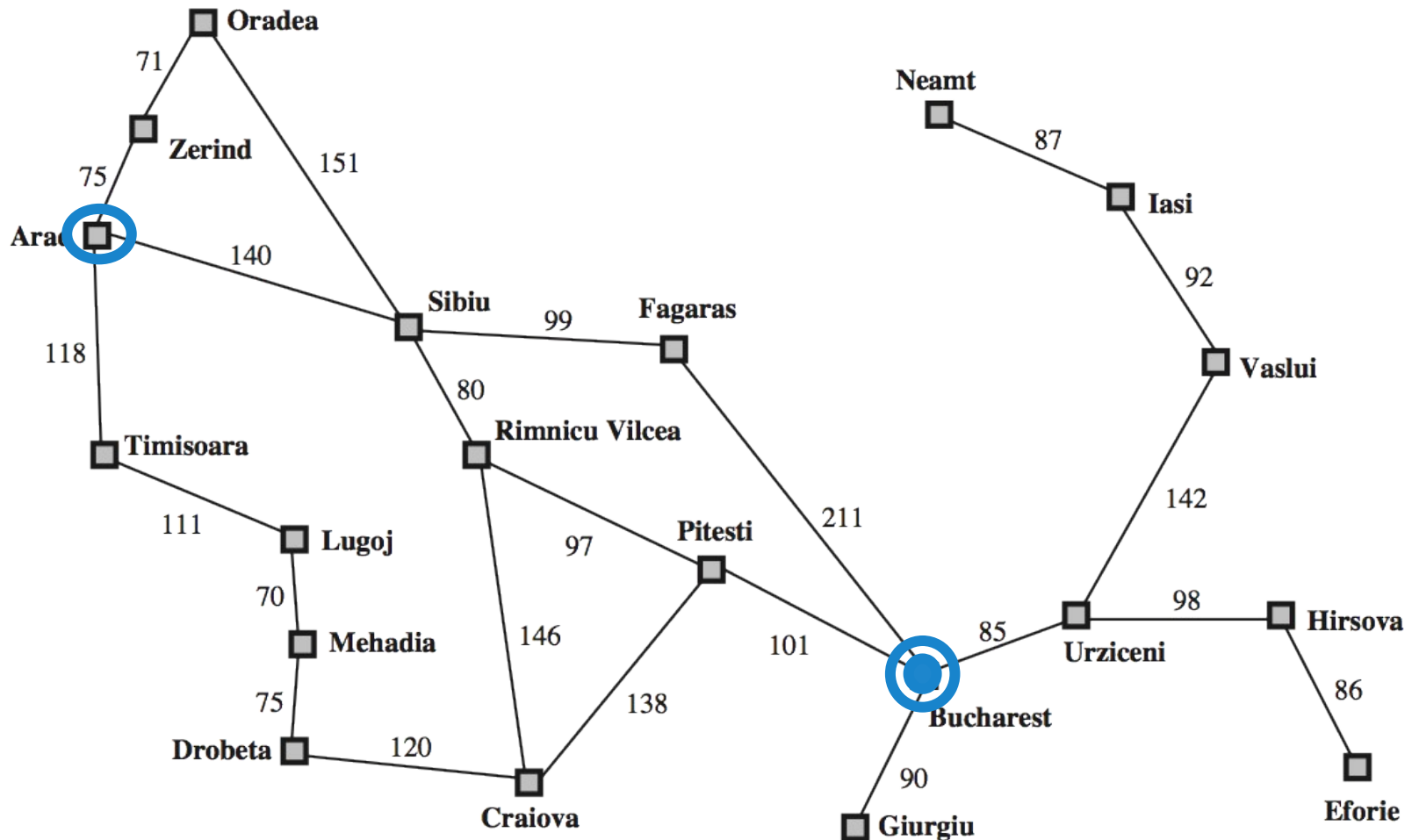


Definición del Problema

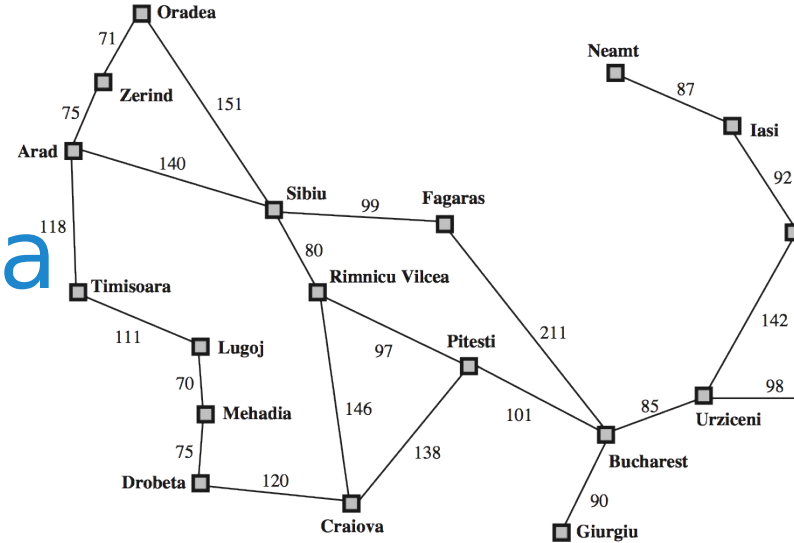
Ejemplo Problema : Rumanía

- De vacaciones en Rumanía; estamos en Arad
 - Nuestro vuelo sale mañana de Bucharest
- Definimos Objetivo
 - Estar en Bucharest
- Definimos problema
 - Estados: varias ciudades
 - Acciones: conducir entre ciudades
- Solución
 - Secuencia de ciudades; p.e. Arad, Sibiu, Fagaras, Bucharest, ...

Ejemplo: Rumanía

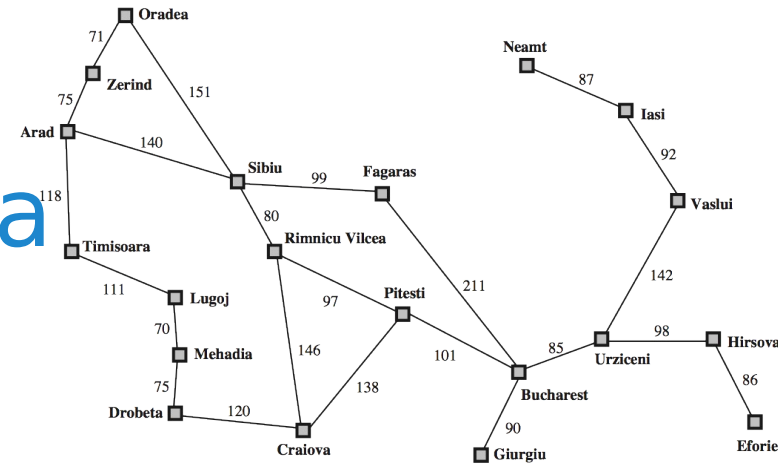


Formulación Problema Rumanía



- **Estado inicial** $\text{En}(\text{Arad})$
- **Acciones aplicables**, p.e. $\text{En}(\text{Arad})$ las acciones aplicables son
 $\{\text{Ir}(\text{Sibio}), \text{Ir}(\text{Timisoara}), \text{Ir}(\text{Zerind})\}$
- **El modelo de transición**, p.e.
 $\text{Result}(\text{En}(\text{Arad}), \text{Ir}(\text{Zerind})) = \text{En}(\text{Zerind})$
- **Test Objetivo**, puede ser
 - Explícito, p.e. $\{\text{En}(\text{Bucharest})\}$
 - Implícito, p.e. $\text{JaqueMate}(x)$, $\text{NoSucio}(x)$
- **Coste del camino** (aditivo), p.e. **suma de distancias**, número de acciones ejecutadas, ...
 - $c(x, a, y)$ es el coste por paso, asumimos que es ≥ 0

Formulación Problema Rumanía



■ Espacio de estados del Problema

- **Definido por:** Estado Inicial + Acciones + Modelo de transiciones
- El **espacio de estado** forma **un grafo dirigido** en el que los nodos son estados y los enlaces son acciones.
 - El mapa de Rumanía puede ser interpretado como un **grafo de estados** si vemos cada carretera como un enlace en cada sentido.
 - Un **camino** es una secuencia de estados conectados por una secuencia de acciones.
 - La **solución** es la secuencia de acciones que lleva del estado inicial al final.

Selección del espacio de estados

- El mundo real es complejo

El espacio de estado debe abstraer el mundo real.

- (**Abstraer**) estado = conjunto de estados reales.

- (**Abstraer**) acción = combinación de acciones.

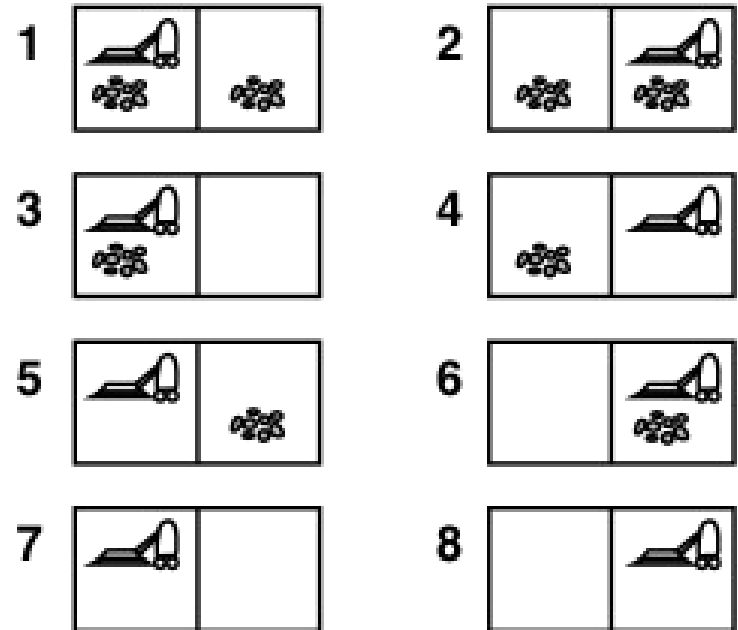
- p. e. Arad → Zerind representa un conjunto complejo de rutas posibles, rotondas, estaciones de servicio, etc.
- La abstracción es válida si el camino entre los dos estados queda reflejado en el mundo real.

- (**Abstraer**) solución = conjunto real de caminos en el mundo real.

- Cada acción de abstracción “simplifica” el mundo real

Ejemplo: Mundo del aspirador

- Acciones aspirar y mover de derecha a izquierda.
 - Sensor localización y suciedad.



Ejemplo: Mundo del aspirador

■ Estados

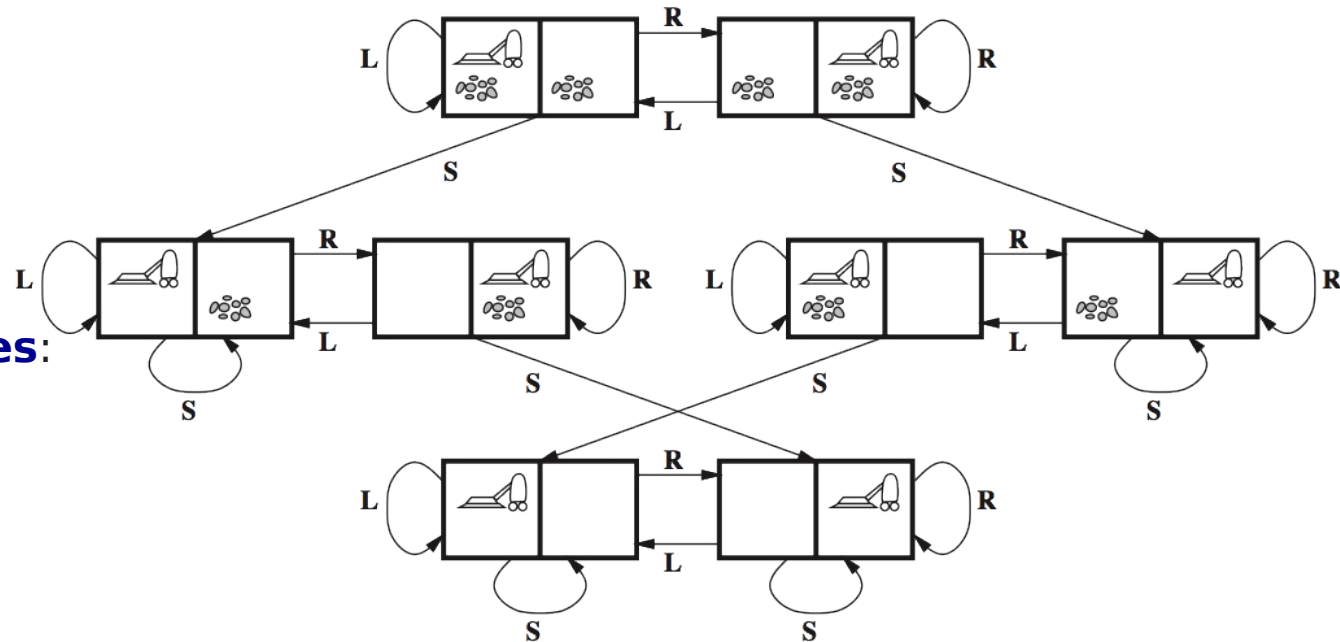
■ Estado Inicial:

■ Acciones:

■ Modelo Transiciones:

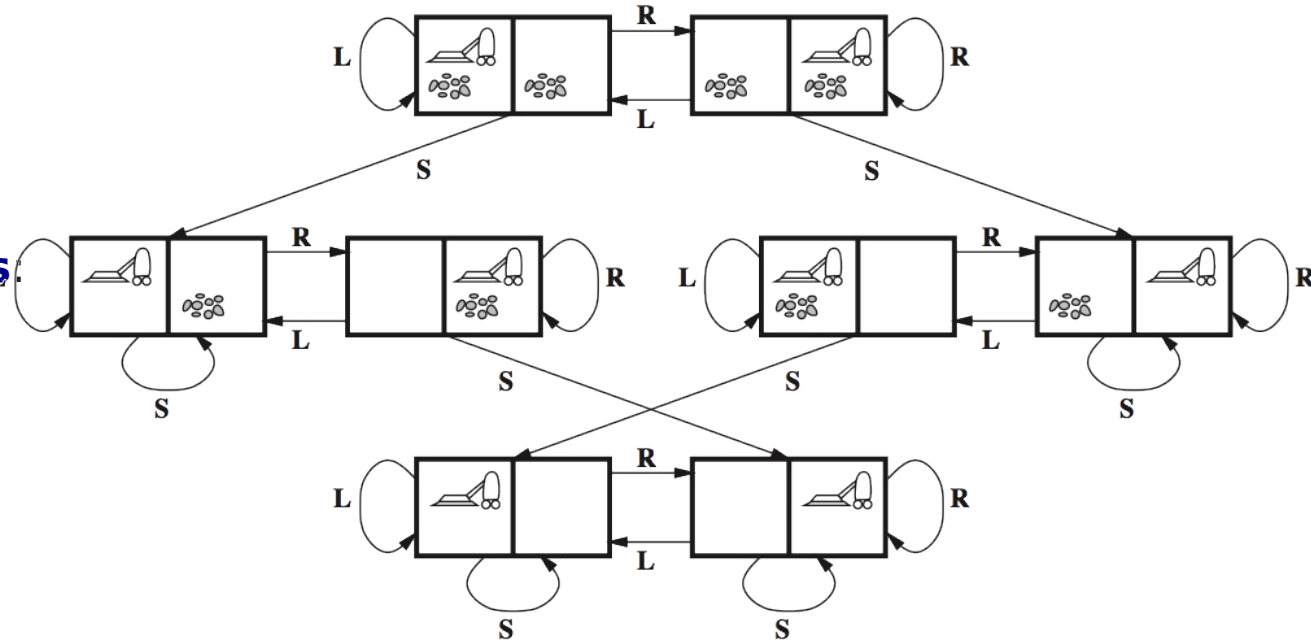
■ Test Objetivo:

■ Coste camino:



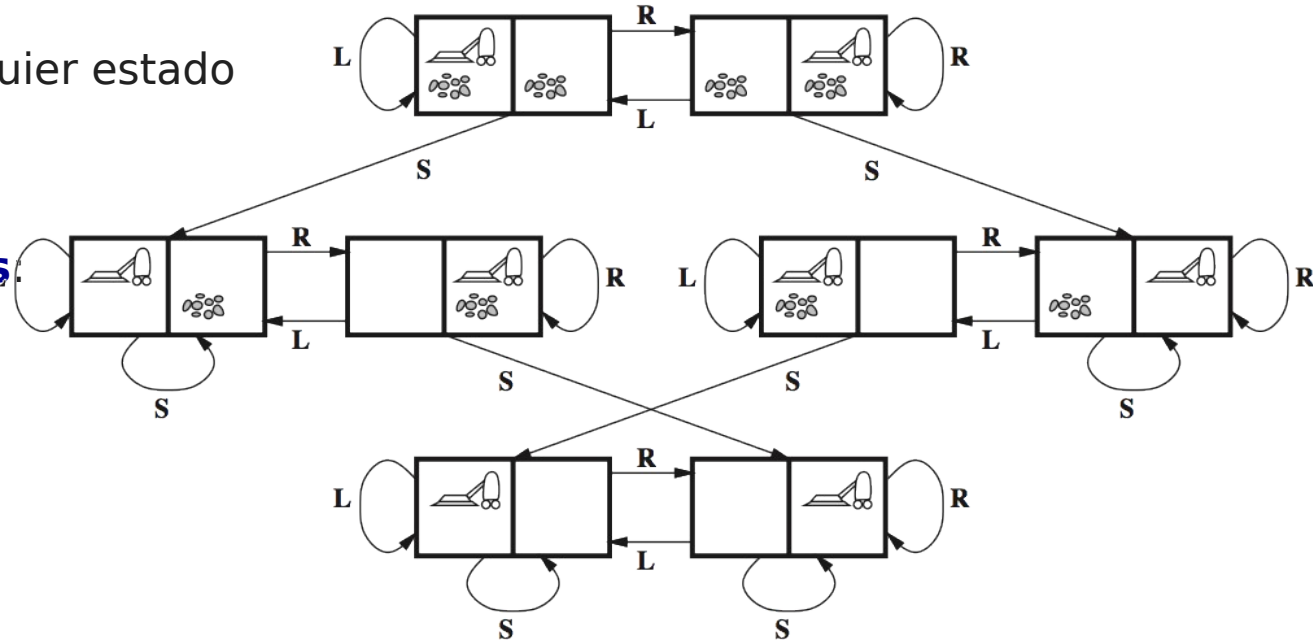
Ejemplo: Mundo del aspirador

- **Estados:** **localización** robot **L** o **R** y cada localización puede estar **Sucio/Limpio (0,1)** y. Son $2 \times 2^2 = 8$ estados. Con n localizaciones $n \times 2^n$ estados.
- **Estado Inicial:**
- **Acciones:**
- **Modelo Transiciones:**
- **Test Objetivo:**
- **Coste camino:**



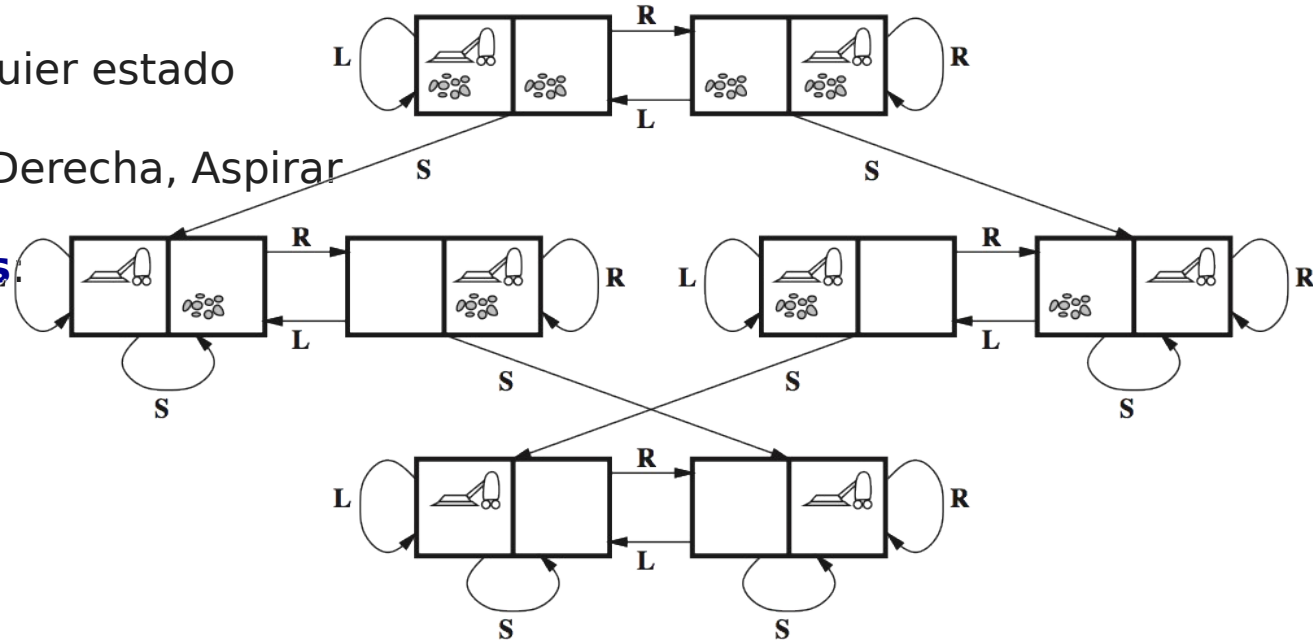
Ejemplo: Mundo del aspirador

- **Estados:** : **localización** robot **L** o **R** y cada localización puede estar **Sucio/Limpio (0,1)** y. Son $2 \times 2^2 = 8$ estados. Con n localizaciones $n \times 2^n$ estados.
- **Estado Inicial:** Cualquier estado
- **Acciones:**
- **Modelo Transiciones:**
- **Test Objetivo:**
- **Coste camino:**



Ejemplo: Mundo del aspirador

- **Estados**: : **localización** robot **L** o **R** y cada localización puede estar **Sucio/Limpio (0,1)** y. Son $2 \times 2^2 = 8$ estados. Con n localizaciones $n \times 2^n$ estados.
- **Estado Inicial**: Cualquier estado
- **Acciones**: Izquierda, Derecha, Aspirar
- **Modelo Transiciones**:
- **Test Objetivo**:
- **Coste camino**:



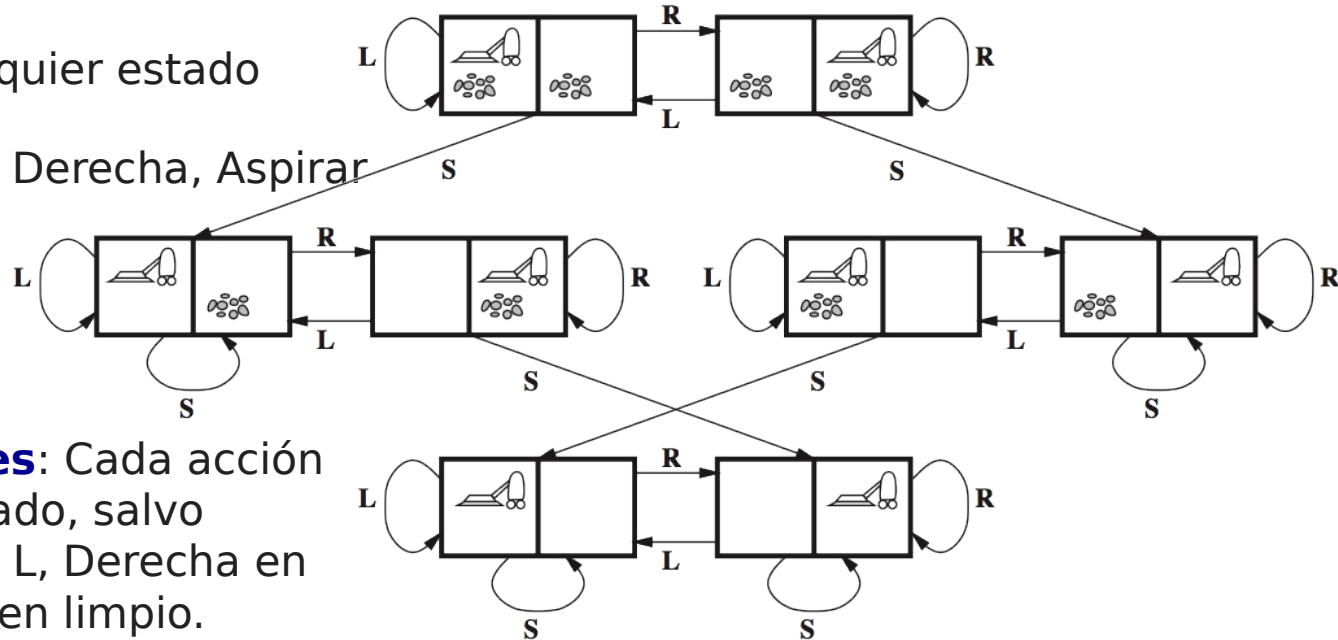
Ejemplo: Mundo del aspirador

- **Estados**: : **localización** robot **L** o **R** y cada localización puede estar **Sucio/Limpio (0,1)** y. Son $2 \times 2^2 = 8$ estados. Con n localizaciones $n \times 2^n$ estados.

- **Estado Inicial**: Cualquier estado

- **Acciones**: Izquierda, Derecha, Aspirar

Left, Right, Suck



- **Modelo Transiciones**: Cada acción tiene el efecto esperado, salvo izquierda en posición L, Derecha en posición R, y Aspirar en limpio.

- **Test Objetivo**:

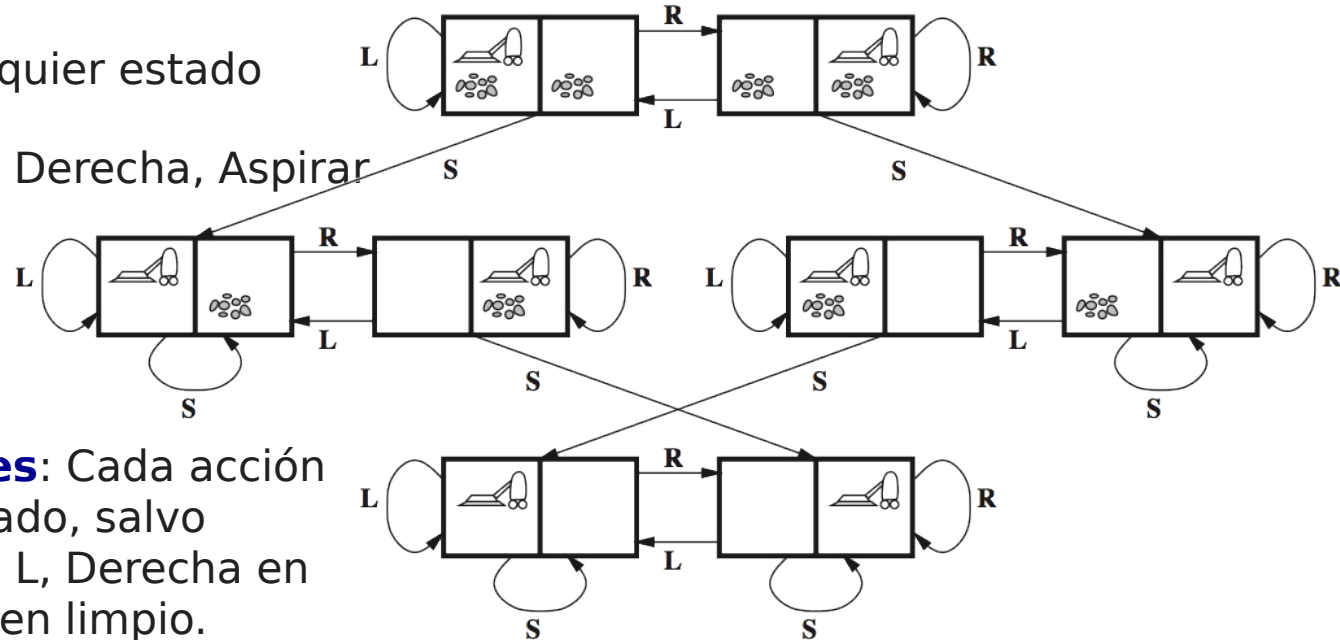
- **Coste camino**:

Ejemplo: Mundo del aspirador

- **Estados**: : **localización** robot **L** o **R** y cada localización puede estar **Sucio/Limpio (0,1)** y. Son $2 \times 2^2 = 8$ estados. Con n localizaciones $n \times 2^n$ estados.

- **Estado Inicial**: Cualquier estado

- **Acciones**: Izquierda, Derecha, Aspirar



- **Modelo Transiciones**: Cada acción tiene el efecto esperado, salvo izquierda en posición L, Derecha en posición R, y Aspirar en limpio.

- **Test Objetivo**: Todas las localizaciones limpias

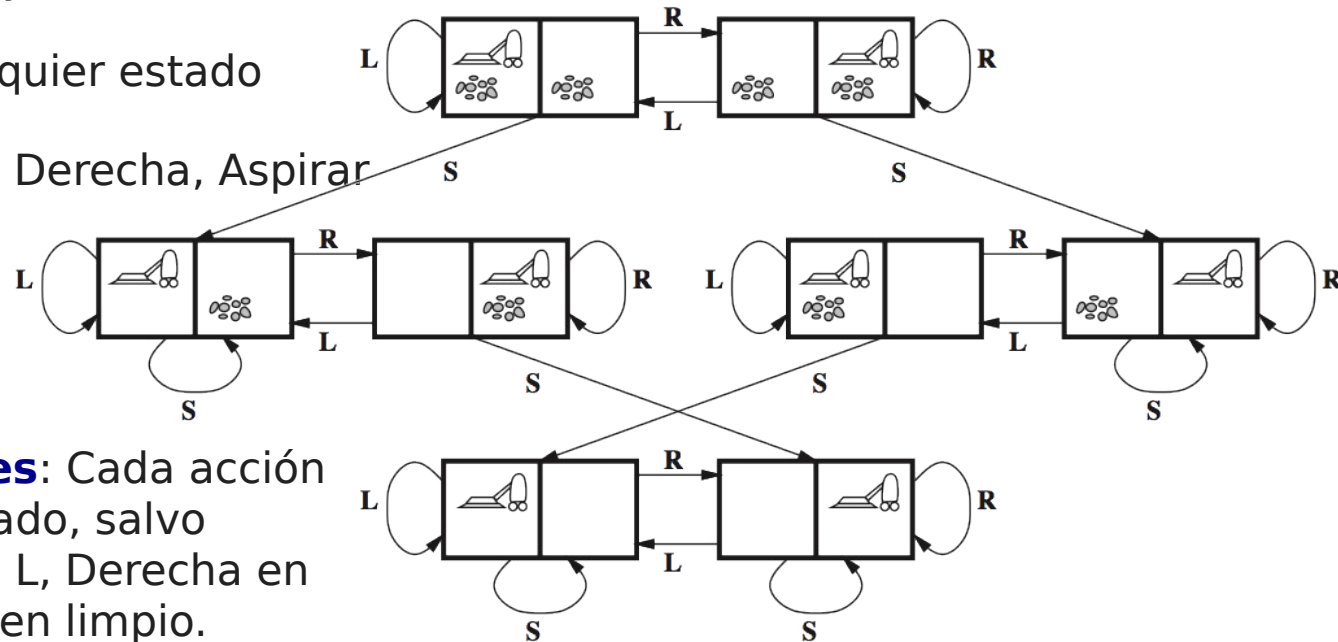
- **Coste camino**:

Ejemplo: Mundo del aspirador

- **Estados**: : **localización** robot **L** o **R** y cada localización puede estar **Sucio/Limpio (0,1)** y. Son $2 \times 2^2 = 8$ estados. Con n localizaciones $n \times 2^n$ estados.

- **Estado Inicial**: Cualquier estado

- **Acciones**: Izquierda, Derecha, Aspirar



- **Modelo Transiciones**: Cada acción tiene el efecto esperado, salvo izquierda en posición L, Derecha en posición R, y Aspirar en limpio.

- **Test Objetivo**: Todas las localizaciones limpias

- **Coste camino**: Cada paso coste 1. Coste camino= longitud del camino.

Ejemplo: 8-puzzle

- **Estados:**
- **Estado Inicial:**
- **Acciones:**
- **Modelo Transiciones:**
- **Test Objetivo:**
- **Coste camino:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Ejemplo: 8-puzzle

- **Estados:**

Localización de cada casilla

- **Estado Inicial:**

- **Acciones:**

- **Modelo Transiciones:**

- **Test Objetivo:**

- **Coste camino:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Ejemplo: 8-puzzle

- **Estados:**

Localización de cada casilla

- **Estado Inicial:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Cualquier estado puede ser inicial¹

- **Acciones:**

- **Modelo Transiciones:**

- **Test Objetivo:**

- **Coste camino:**

¹Cualquier estado objetivo puede ser alcanzado desde la mitad de estados iniciales. El espacio de estados está partido en 2. ESPACIO DE ESTADOS tiene $9!/2 = 181.440$ estados

Ejemplo: 8-puzzle

- **Estados:**

Localización de cada casilla

- **Estado Inicial:**

Cualquier estado puede ser inicial

- **Acciones:** Mover el hueco Arriba, Abajo, Izquierda, Derecha

- **Modelo Transiciones:**

- **Test Objetivo:**

- **Coste camino:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Ejemplo: 8-puzzle

- **Estados:**

Localización de cada casilla

- **Estado Inicial:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Cualquier estado puede ser inicial

- **Acciones:** Mover el hueco **Arriba, Abajo, Izquierda, Derecha**

- **Modelo Transiciones:** Dado el estado y la acción resulta el estado resultante. P.e. Si aplicamos izquierda al estado inicial tenemos el estado resultante de intercambiar el 5 y el hueco.

- **Test Objetivo:**

- **Coste camino:**

Ejemplo: 8-puzzle

- **Estados:**

Localización de cada casilla

- **Estado Inicial:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Cualquier estado puede ser inicial

- **Acciones:** Mover el hueco **Arriba, Abajo, Izquierda, Derecha**

- **Modelo Transiciones:** Dado el estado y la acción resulta el estado resultante. P.e. Si aplicamos izquierda al estado inicial tenemos el estado resultante de intercambiar el 5 y el hueco.

- **Test Objetivo:** Comprueba si es el estado final

- **Coste camino:**

Ejemplo: 8-puzzle

- **Estados:**

Localización de cada casilla

- **Estado Inicial:**

| | | |
|---|---|---|
| 7 | 2 | 4 |
| 5 | | 6 |
| 8 | 3 | 1 |

Estado inicial

| | | |
|---|---|---|
| | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Estado objetivo

Cualquier estado puede ser inicial

- **Acciones:** Mover el hueco **Arriba, Abajo, Izquierda, Derecha**

- **Modelo Transiciones:** Dado el estado y la acción resulta el estado resultante. P.e. Si aplicamos izquierda al estado inicial tenemos el estado resultante de intercambiar el 5 y el hueco.

- **Test Objetivo:** Comprueba si es el estado final

- **Coste camino:** Cada paso coste 1. Coste = longitud del camino.

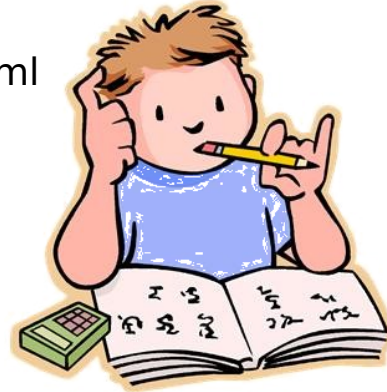
Averiguar el porqué de este reto.

Pista para resolver:

<http://www.cut-the-knot.com/pythagoras/fifteen.shtml>

15-puzzle

- Presentado en 1878 por Sam Lloyd, quien se autodenominó “El mayor experto americano en puzles”



A \$1,000.00 Cash Prize Puzzle

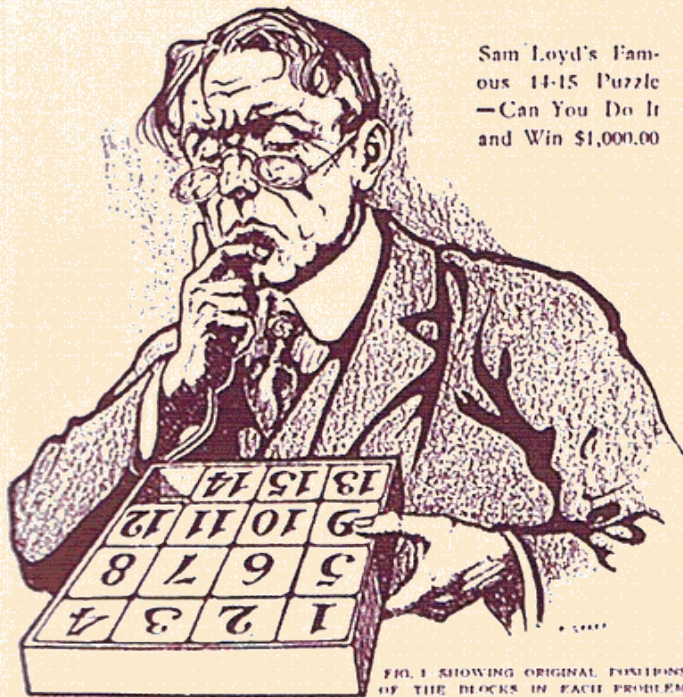
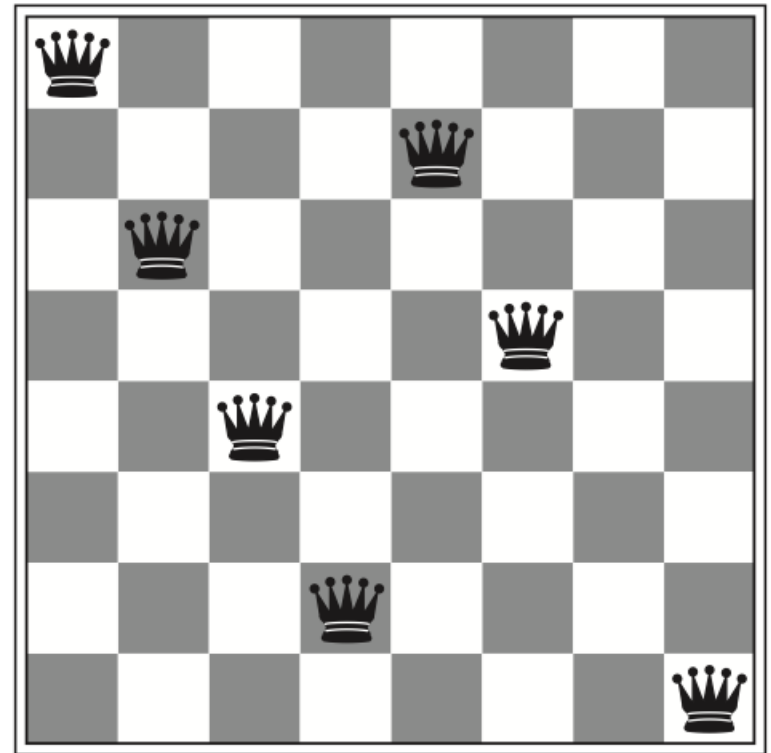


FIG. 1. SHOWING ORIGINAL POSITIONS OF THE BLOCKS IN EACH PROBLEM

Ejemplo: 8-reinas

Problema: Colocar las 8 reinas en un tablero 8x8 sin que se coman.

- **Estados**
- **Estado Inicial:**
- **Acciones:**
- **Modelo Transiciones:**
- **Test Objetivo:**
- **Coste camino:**



Ejemplo: 8-reinas

Problema: Colocar las 8 reinas en un tablero 8x8 sin que se coman.

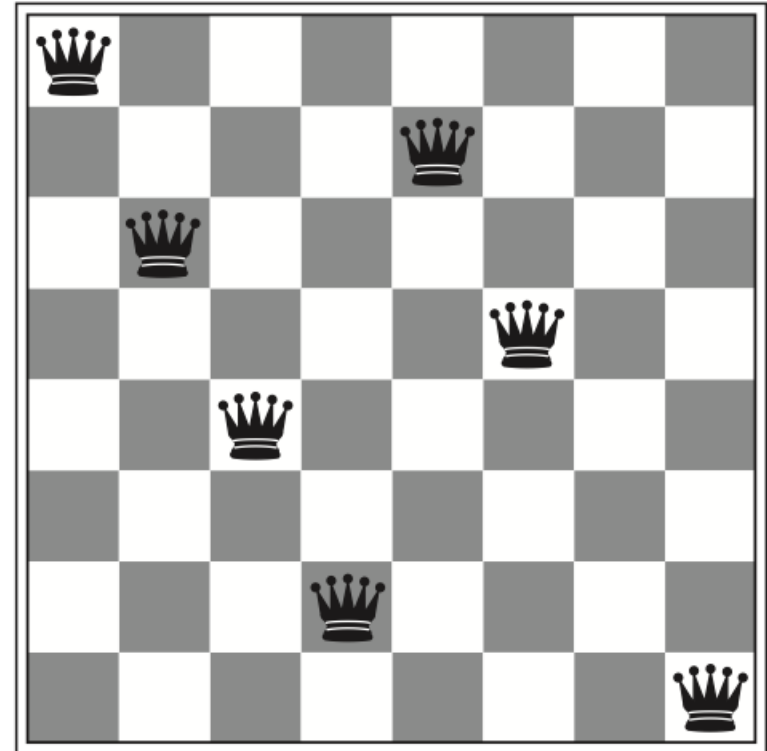
■ Dos posibles formulaciones

■ Estado completo

- Parto de una configuración cualquiera con 8 reinas y las movemos para cambiar de estado.

■ Incremental

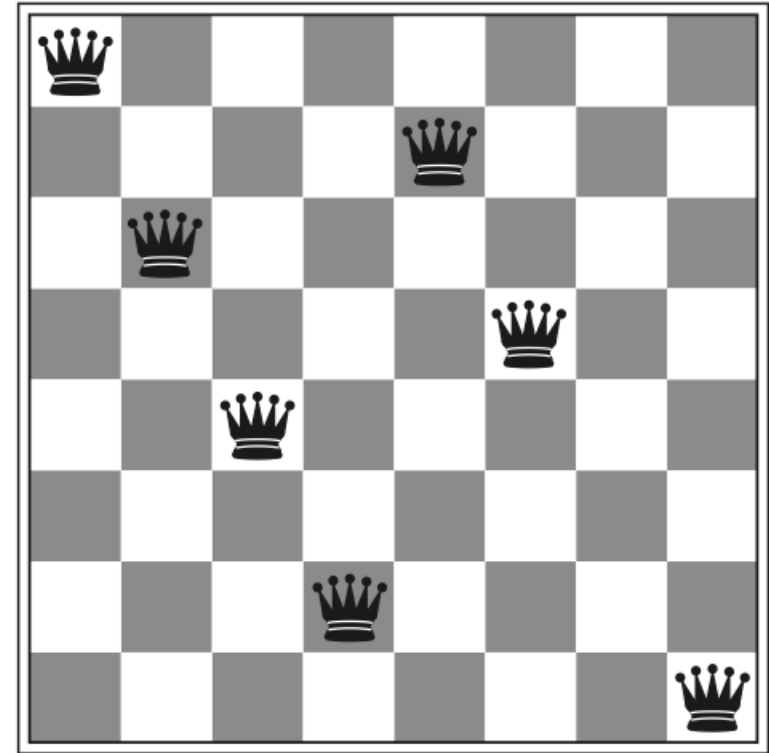
- Parto de un estado con el tablero vacío y coloco una reina cada vez



Ejemplo: 8-reinas

Primera versión incremental. Parto del tablero vacío y coloco una reina cada vez.

- **Estados:** Cualquier combinación de 8 reinas en el tablero
- **Estado Inicial:** Tablero vacío
- **Acciones:** Añadir una reina en un cuadrado vacío
- **Modelo Transiciones:** Devuelve el tablero con la reina añadida en el cuadro especificado.
- **Test Objetivo:** 8 reinas en el tablero sin atacarse
- **Coste camino:** No hay coste. No nos interesa el camino. Sólo nos interesa el estado final.

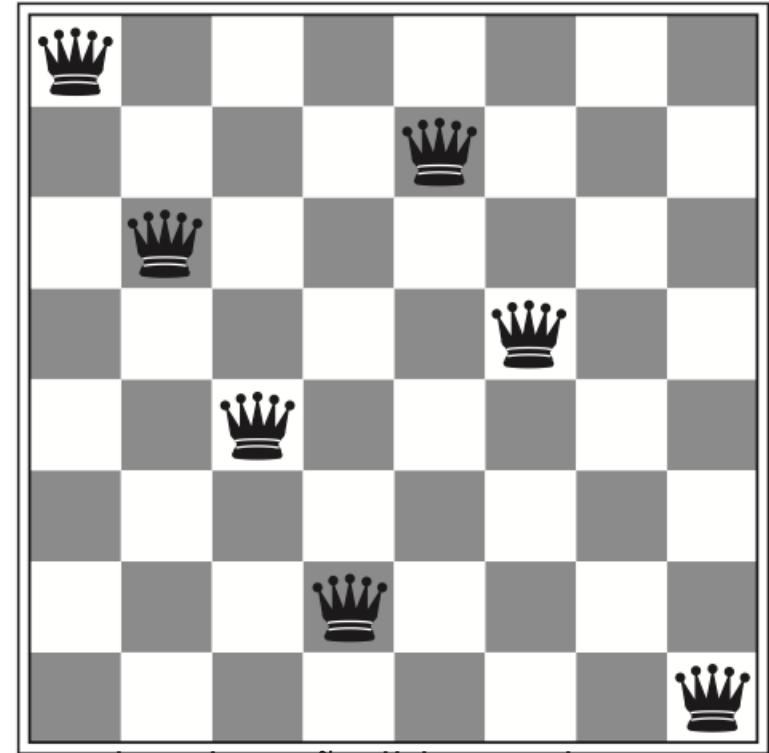


Espacio Estados
 $64 \times 63 \times \dots \times 57 = 3 \times 10^{14}$ estados

Ejemplo: 8-reinas

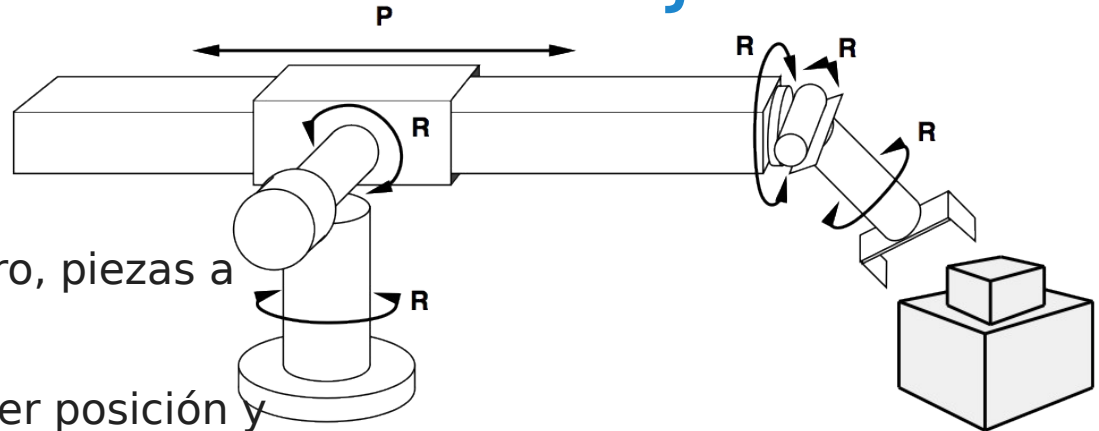
Segunda versión incremental. Parto del tablero vacío y coloco una reina cada vez en una columna de izquierda a derecha

- **Estados:** Cualquier combinación de 8 reinas en el tablero
- **Estado Inicial:** Tablero vacío
- **Acciones:** Añadir una reina en cualquier cuadrado de la columna libre más a la izquierda en la que no es atacada.
- **Modelo Transiciones:** Devuelve el tablero con la reina añadida en el cuadro especificado
- **Test Objetivo:** 8 reinas en el tablero sin atacarse
- **Coste camino:** No hay coste. Sólo nos interesa el estado final.



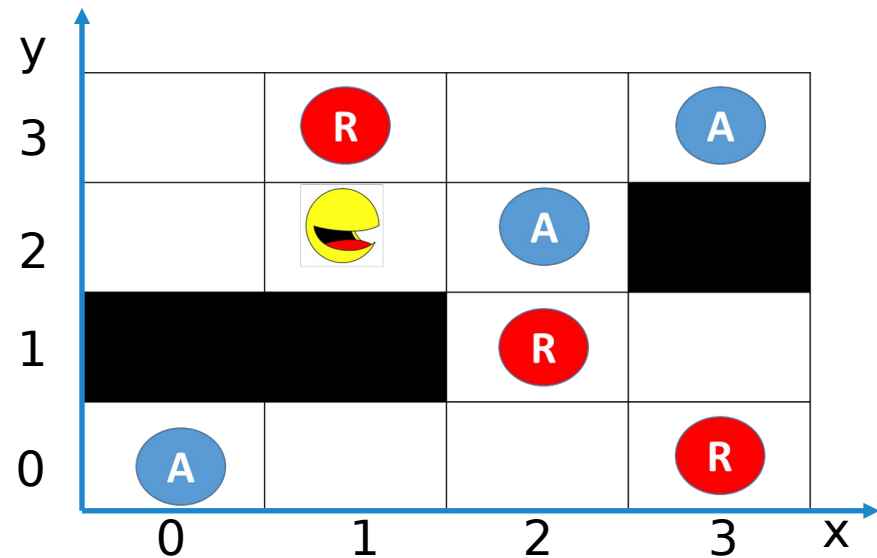
Espacio Estados
2057 estados

Ejemplo : Robot ensamblaje



- **Estados:** Ángulos de giro, piezas a ensamblar
- **Estado Inicial:** Cualquier posición y configuración de objetos
- **Acciones:** Movimiento continuo de la articulaciones del robot.
- **Modelo Transiciones:** Estado resultante del movimiento
- **Test Objetivo:** Ensamblado
- **Coste camino:** Tiempo/consumo energetico

Comecocos



El comecocos representado en el tablero puede comer dos clases de piezas de fruta representadas por los colores rojo (R) y azul (A). El objetivo del comecocos es saborear los dos tipos de frutas: Una vez que ha comido una fruta de cada tipo el juego finaliza (aunque el comecocos podría comer más de una pieza de cada fruta antes de que el juego acabe). Los cuatro movimientos posibles son arriba, abajo, izquierda y derecha. Debe moverse en todos los turnos. No es posible cruzar obstáculos (casillas negras). Se considera que la fruta que ocupa una casilla es comida una vez que el comecocos alcanza la casilla.

Define una **representación del problema**, especificando una representación del estado, estado inicial, estados finales, así como operador(es) y su coste

Estado

Define una **representación del problema**, especificando una representación del estado, estado inicial, estados finales, así como operadores(es) y su coste.

El estado viene dado por una tupla $(x, y, ComR, ComA)$ donde $x \in [1,4]$, $y \in [1,4]$, $ComR \in \{T, F\}$ y $ComA \in \{T, F\}$

*El **estado inicial** viene dado por $(2, 2, F, F)$*

*Se define el **test objetivo** como $(ComR = T \ \& \ ComF = T)$ que cumplen todas las tuplas (x, y, T, T) .*

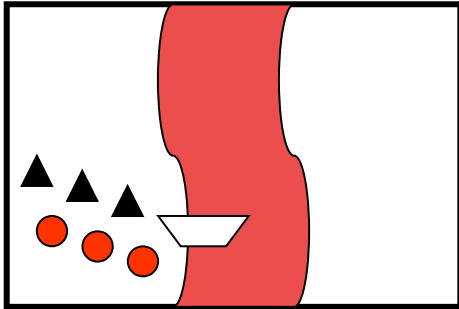
Comecocos

El **modelo de transiciones** debe representar las **precondiciones** de cada acción y sus **efectos**

| Acciones | Precondiciones | Resultado |
|-------------------------------|--|---|
| Abajo (x,y,ComR,ComA) | $y > 0$, $\text{tablero}[x,y-1] \neq \text{negro}$ | si $(\text{tablero}[x,y-1]=A)$ $\text{comA}=T$, si $(\text{tablero}[x,y-1]=R)$ $\text{comR}=T$, $y=y-1$ |
| Arriba (x,y,ComR,ComA) | $y < 3$, $\text{tablero}[x,y+1] \neq \text{negro}$ | si $(\text{tablero}[x,y+1]=A)$ $\text{comA}=T$, si $(\text{tablero}[x,y+1]=R)$ $\text{comR}=T$, $y=y+1$ |
| Izq (x,y,ComR,ComA) | $x > 0$, $\text{tablero}[x-1,y] \neq \text{negro}$ | si $(\text{tablero}[x-1,y]=A)$ $\text{comA} = T$, si $(\text{tablero}[x-1,y]=R)$ $\text{comR} = T$, $x=x-1$ |
| Dcha (x,y,ComR,ComA) | $x < 3$, $\text{tablero}[x+1,y] \neq \text{negro}$ | si $(\text{tablero}[x+1,y]=A)$ $\text{comA} = T$, si $(\text{tablero}[x+1,y]=R)$ $\text{comR} = T$, $x=x+1$ |

Coste camino = Suma de número de pasos.

Problema de Juguete: Misioneros y Caníbales



En una orilla del río hay tres misioneros (triángulos negros) y tres caníbales (círculos rojos). Hay un bote disponible

que puede llevar a dos personas a la vez como máximo de un lado al otro del río. Si los caníbales superan a los misioneros en número en cualquier de los lados del río, se comen a los misioneros ¿Cómo puedes llevar a los misioneros y los caníbales de forma segura de una orilla a la otra?

Plantear el problema y dibujar espacio de estados para la próxima clase



Algoritmos de exploración

**Búsqueda en
el espacio de estados**

Algoritmos básicos de búsqueda

- Búsqueda como solución a problemas
 - Idea básica de la **Búsqueda en el espacio de estados**:
 - **Simular** la exploración del **espacio de estados**
 - Generando los sucesores de los estados ya explorados (**Expandir estados**).
 - La búsqueda genera un **árbol de búsqueda**
 - **Nodo raíz** = estado inicial.
 - **Nodos** (estados) y hojas (nodos sin hijos) generados por la función sucesor.
 - En general la búsqueda genera un **árbol de búsqueda**. Se puede llegar a un estado por **múltiples caminos**.
 - Podemos volver al mismo estado después de visitarlo
 - Podemos encontrar diferentes caminos al mismo

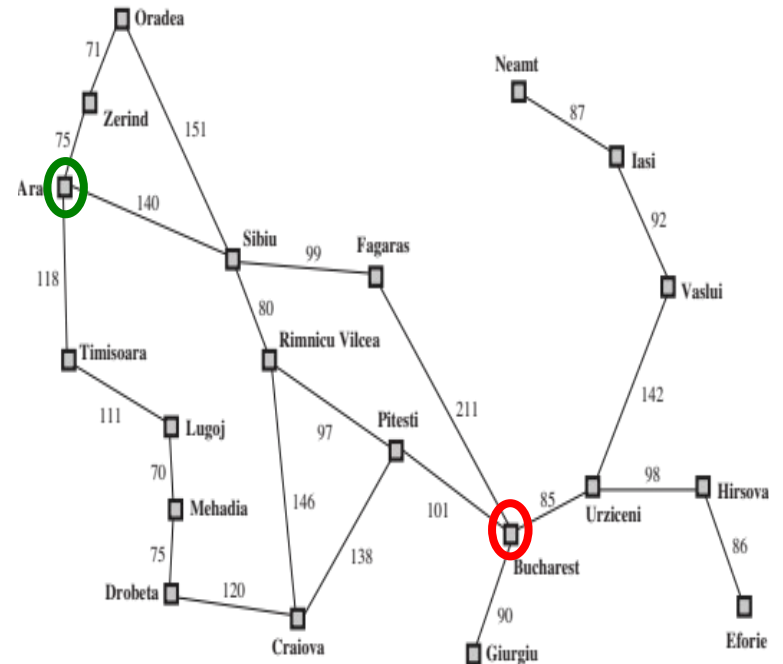
Algoritmos básicos de búsqueda

■ Nomenclatura

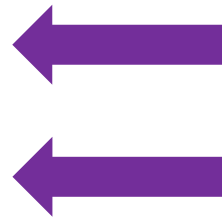
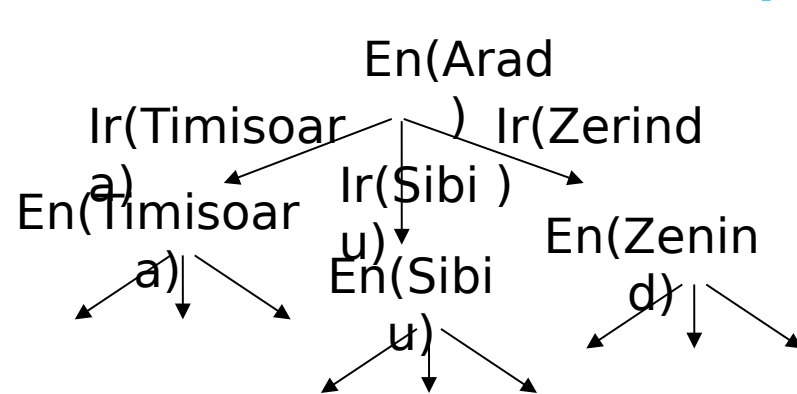
- La búsqueda genera un **árbol de búsqueda**
- **Nodo** representa estado (en un árbol de búsqueda, y apunta al padre y a los sucesores)
- **Expandir** estado es generar nuevos estados aplicando acciones permitidas sobre un estado.
- El nodo expandido es el **nodo padre** de los nodos resultantes (**nodos hijos**).
- Al conjunto de nodos sin hijos (no expandidos) se le denomina **frontera** o **lista abierta**.

Grafo del Espacio Estados

- Grafo **del espacio de estados**: Una representación matemática del problema de búsqueda
 - Los **nodos** son configuraciones del mundo (abstracciones de estados posibles)
 - Los **arcos** representan transiciones de estado como resultado de las acciones
 - El **test de objetivo** es un conjunto de uno o más nodos
- En un grafo de estados, cada estado aparece sólo una vez.
- Normalmente **no podremos representar este grafo completamente en memoria** (Suele ser muy grande)



Árbol de búsqueda

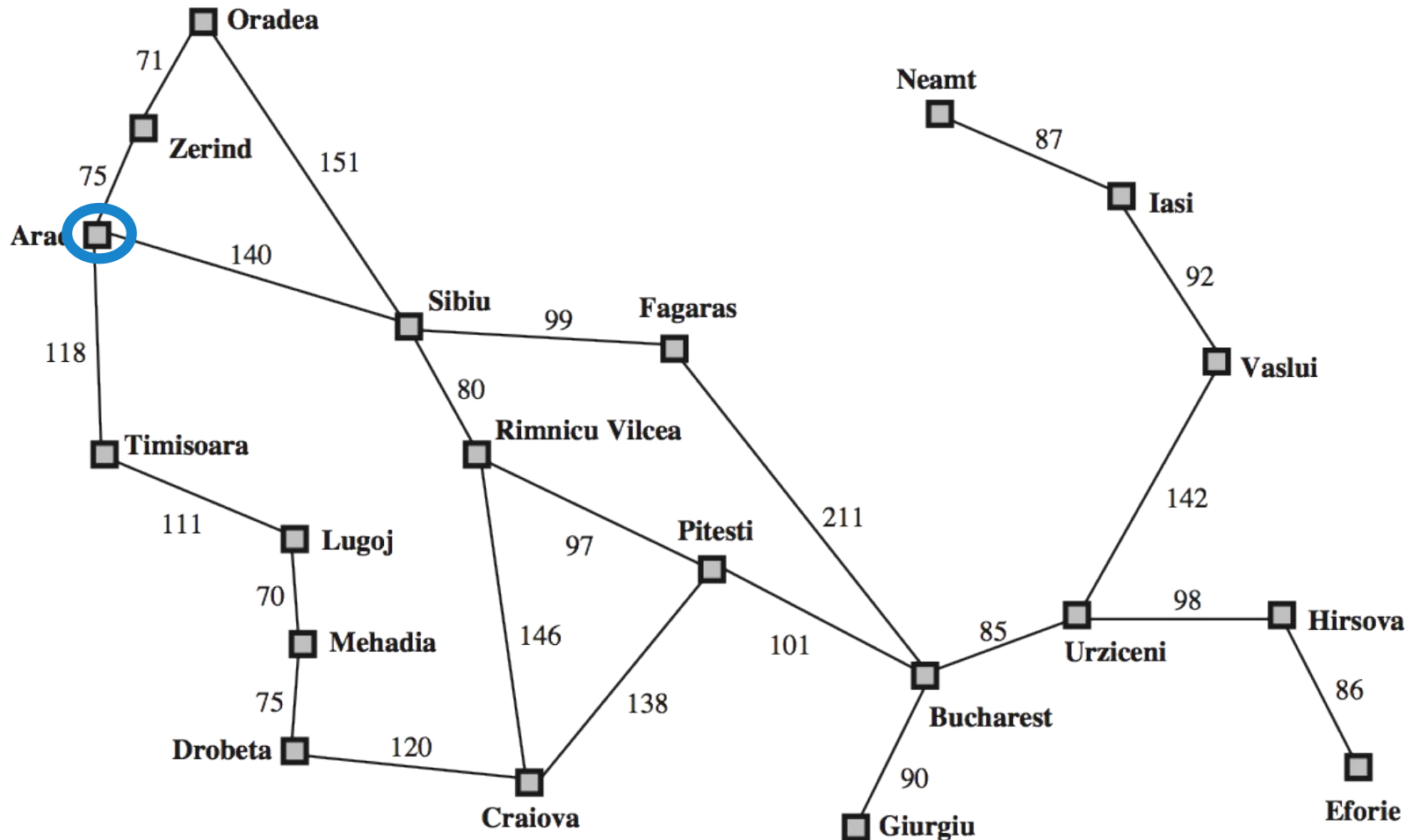


Nodo en curso/
Principio
Posibles estados
futuros

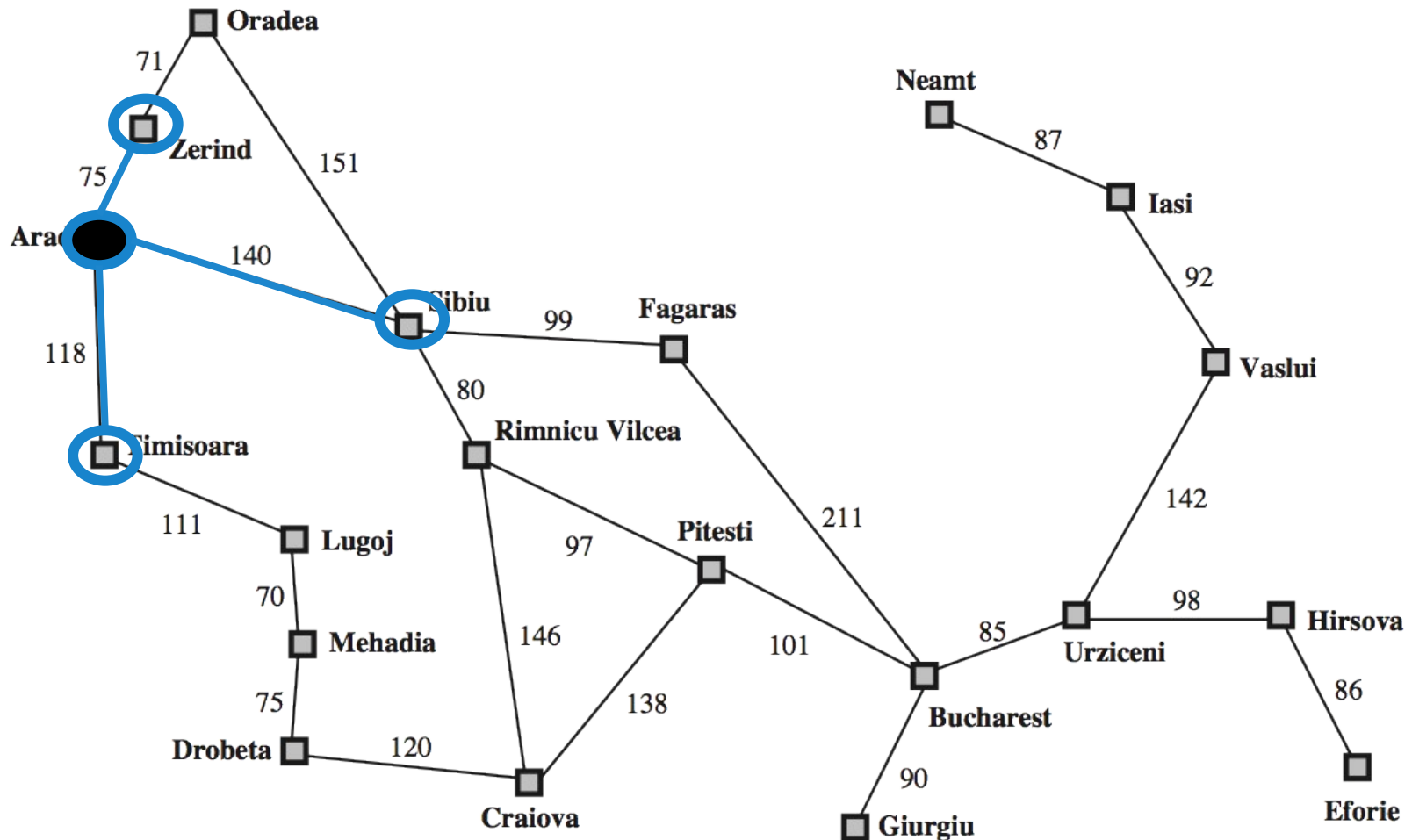


- Un árbol de búsqueda/exploración:
 - Un árbol con “las posibilidades” y sus resultados
 - El nodo inicial es la raíz
 - Los hijos corresponden a los resultados de las acciones
 - Los nodos son estados, pero se corresponden con los PLANES para lograr esos estados.
 - Para la mayoría de problemas no construiremos realmente el árbol completo.

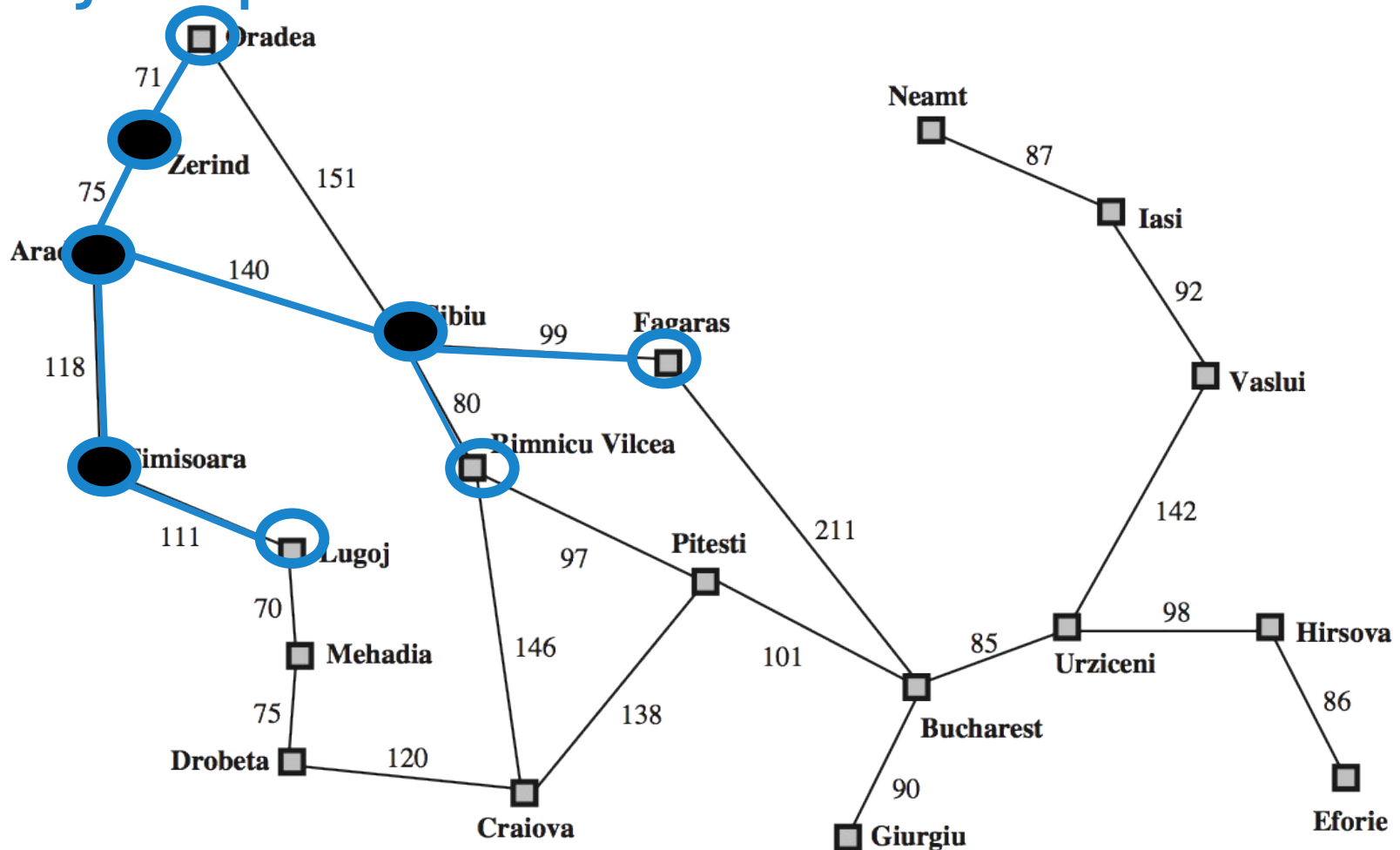
Ejemplo: Rumanía



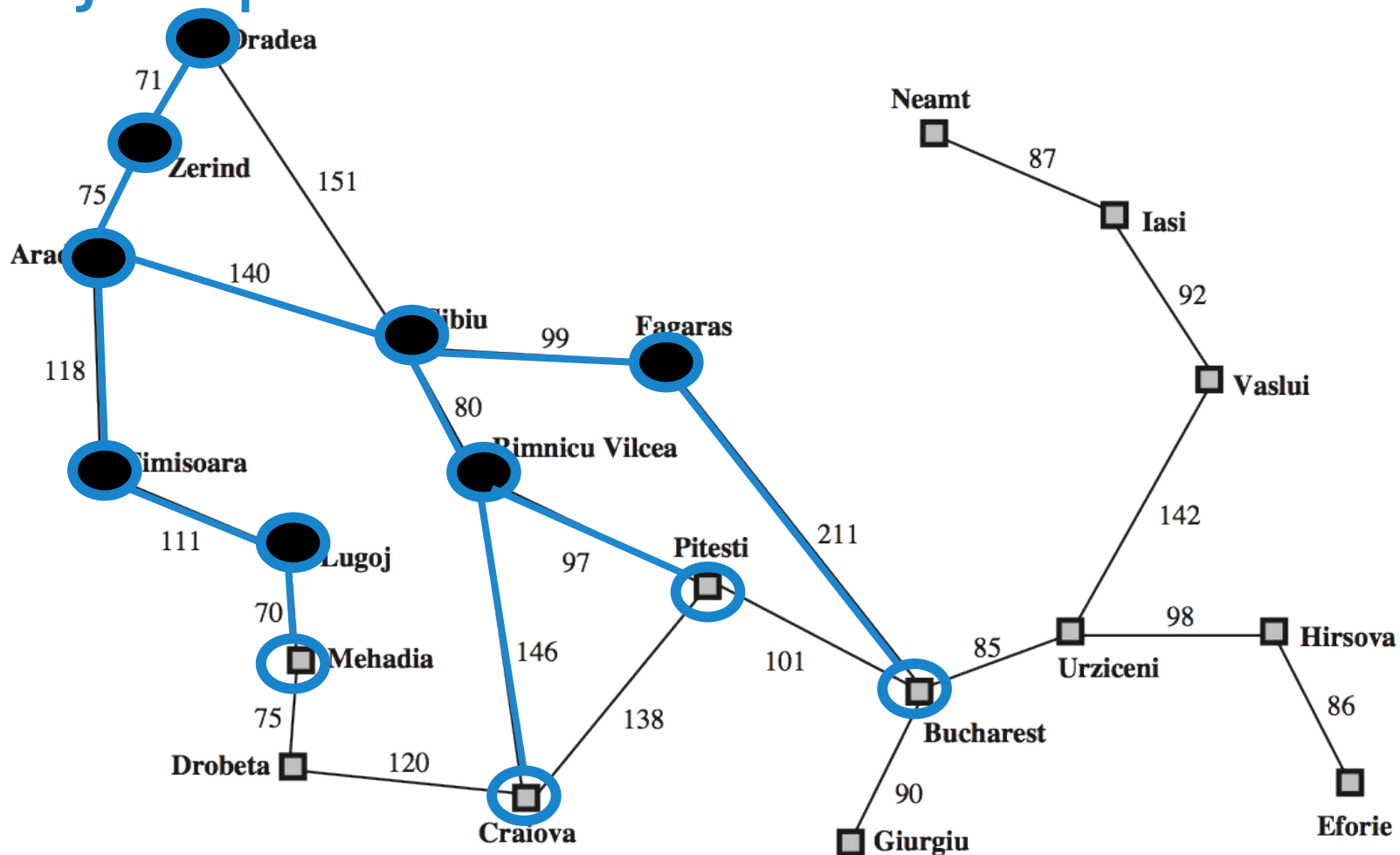
Ejemplo: Rumanía



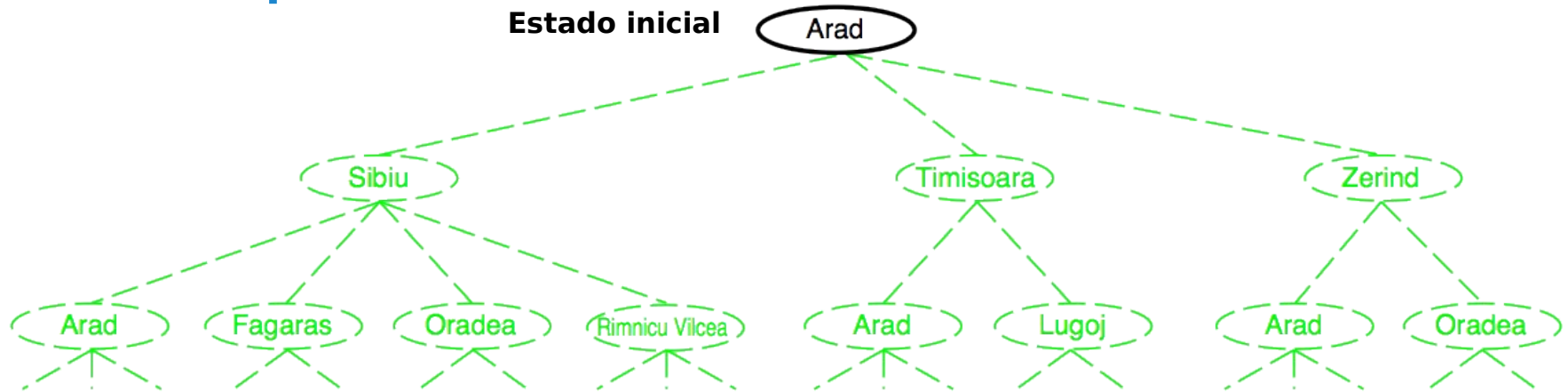
Ejemplo: Rumanía



Ejemplo: Rumanía



Descripción informal de la búsqueda en árbol



function BÚSQUEDA-ÁRBOL (*problema*, *estrategia*) **returns** solución o fallo

Inicializa la **frontera** utilizando el estado inicial del problema

loop do

if la **frontera** está vacía **then return fallo**

elige un **nodo** hoja de la frontera de acuerdo a una **estrategia**

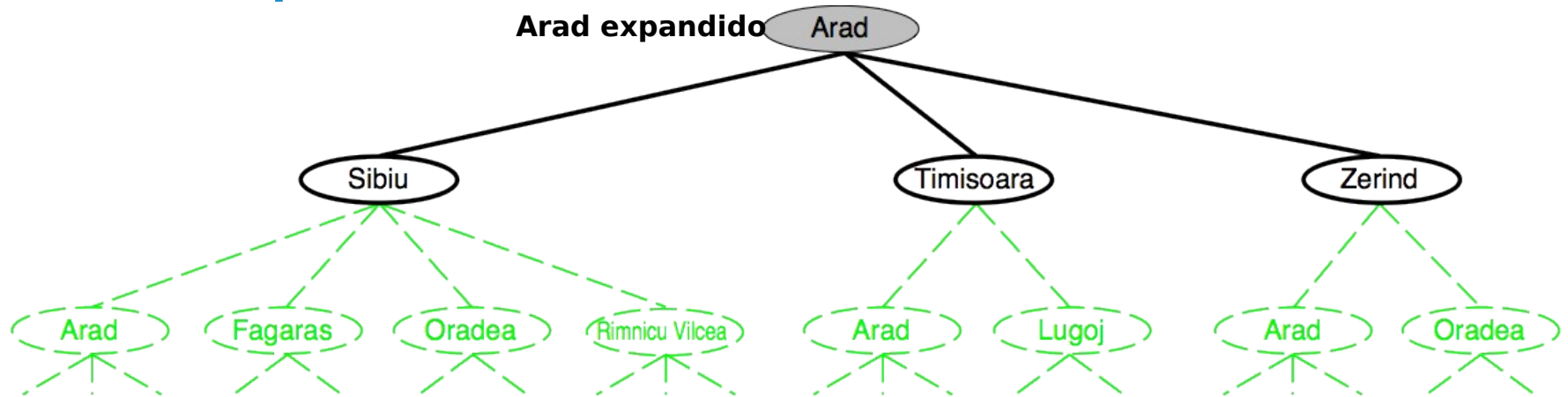
if el **nodo** contiene un nodo objetivo **then return solución**

expande el nodo elegido, añadiendo los nodos resultantes a la

frontera



Descripción informal de la búsqueda en **árbol**



function BÚSQUEDA-ÁRBOL (*problema, estrategia*) **returns** solución o fallo

Inicializa la **frontera** utilizando el estado inicial del problema

loop do

if la **frontera** está vacía **then return fallo**

elige un **nodo** hoja de la frontera de acuerdo a una **estrategia**

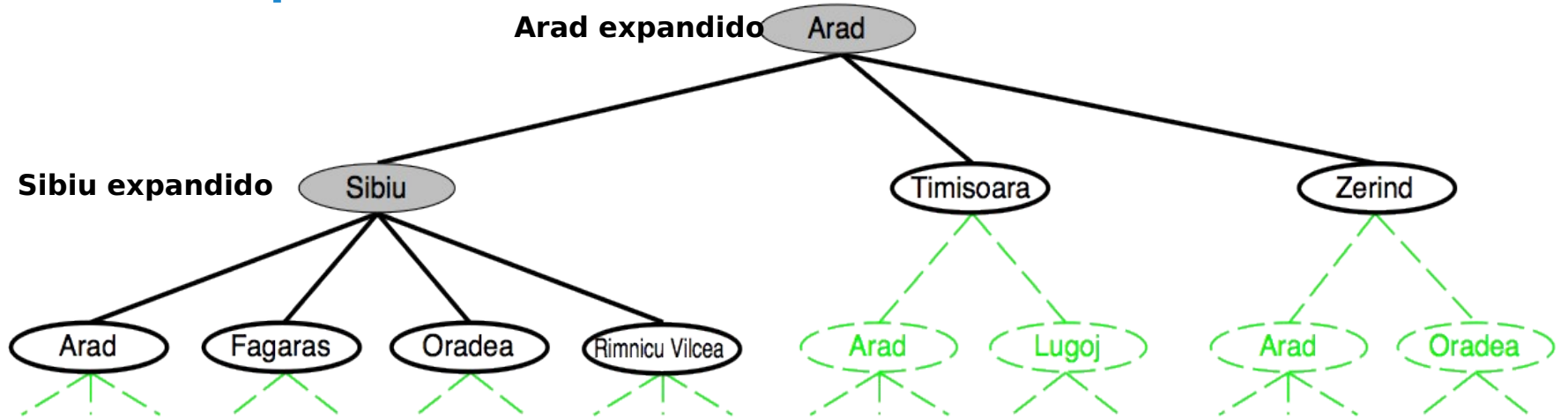
if el **nodo** contiene un nodo objetivo **then return solución**

expande el nodo elegido, añadiendo los nodos resultantes a la

frontera



Descripción informal de la búsqueda en **árbol**



function BÚSQUEDA-ÁRBOL (*problema, estrategia*) **returns** solución o fallo

Inicializa la *frontera* utilizando el estado inicial del problema

loop do

if la *frontera* está vacía **then return fallo**

elige un *nodo* hoja de la frontera de acuerdo a una *estrategia*

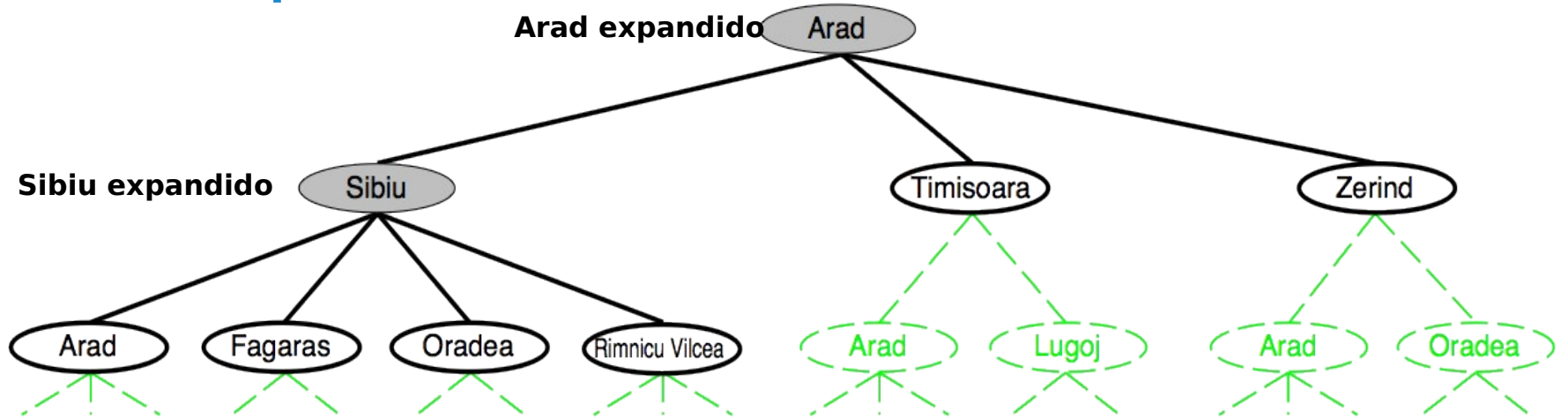
if el *nodo* contiene un nodo objetivo **then return solución**

expande el nodo elegido, añadiendo los nodos resultantes a la

frontera



Descripción informal de la búsqueda en **árbol**



function BÚSQUEDA-ÁRBOL (*problema, estrategia*) **returns** solución o fallo

¡Determina el proceso de búsqueda! Inicializa la **frontera** utilizando el estado inicial del problema

loop do

➔ **if** la **frontera** está vacía **then return** **fallo**

elige un **nodo** hoja de la frontera de acuerdo a una **estrategia**

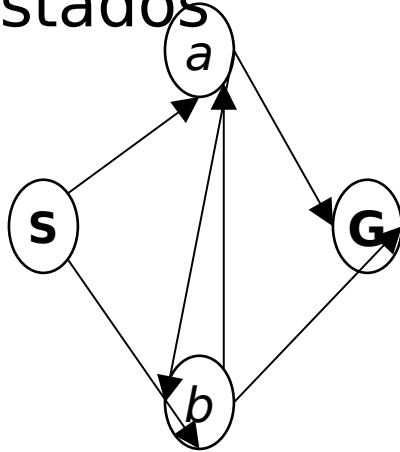
if el **nodo** contiene un nodo objetivo **then return** **solución**

expande el nodo elegido, añadiendo los nodos resultantes a la

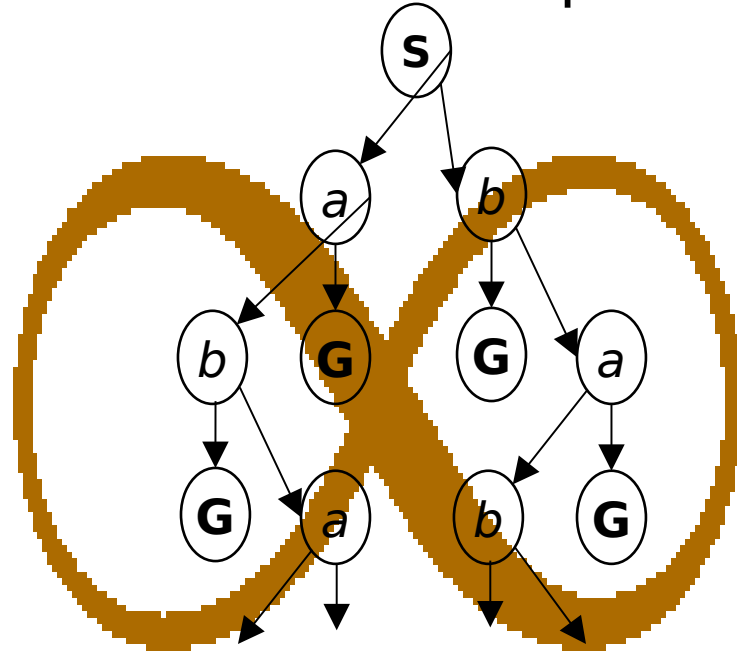
frontera

Búsqueda en grafo / Búsqueda en árbol

Dado el siguiente
grafo con 4
estados



¿Cómo de grande es el
árbol de búsqueda?



Importante: ¡En un árbol de búsqueda se repiten
un montón de veces la misma estructura!

Descripción informal de la búsqueda en grafo (evitar lazos)

function BÚSQUEDA-GRAFO(*problema*, *estrategia*) **returns** solución o fallo

Inicializa la *frontera* utilizando el estado inicial del problema

Inicializa el conjunto de nodos *explorados* a vacío

loop do

if la *frontera* está vacía **then return** fallo

elige un *nodo* hoja de la frontera de acuerdo a una *estrategia*

if el *nodo* contiene un nodo objetivo **then return** solución

añade el *nodo* al conjunto de nodos *explorados*²

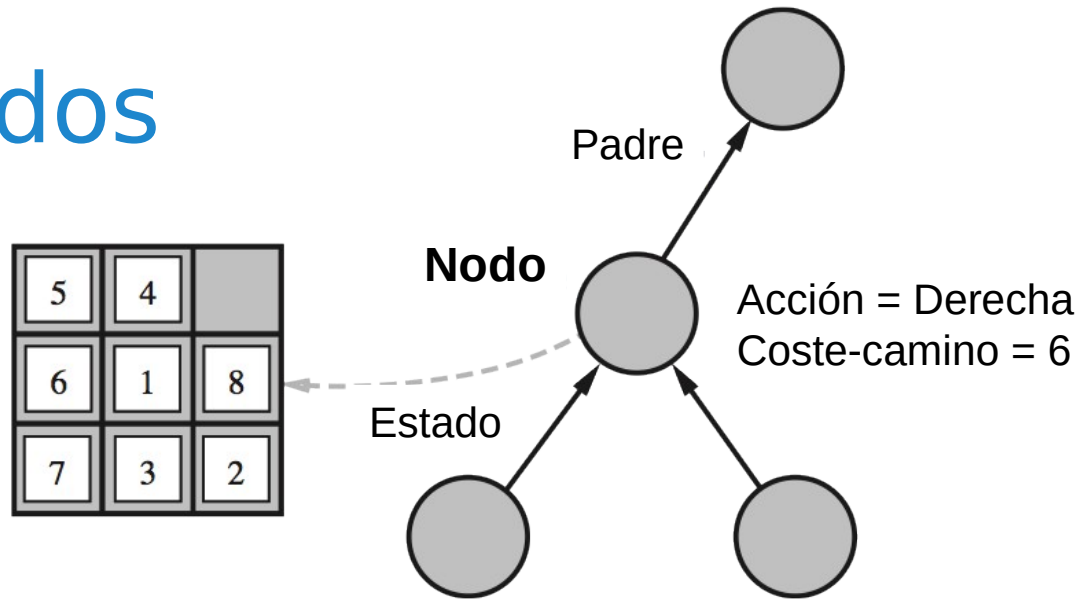
expande el nodo elegido, añadiendo los nodos resultantes a la

frontera sólo si no están en la *frontera* o en el conjunto

explorados

² Algunos autores denominan **lista abierta** o **abiertos** a la *frontera*, y **lista cerrada** a los nodos *explorados*

Estados y nodos



- Un **estado** es una (representación de una) configuración física
- Un **nodo** una estructura de datos que forma el árbol de búsqueda
 - Incluye **padre**, **hijo**, ... y puede incluir **profundidad**, función de **coste camino $g(x)$** , **acción**
- ¡Los estados no tienen padres, ni hijos, ni profundidad ni función de coste!

Función nodo-hijo

function NODO-HIJO(*problema*, *padre*, *acción*) **returns** a **node**
return a *node* with
ESTADO= *problema*.RESULT(**parent**.ESTADO, *acción*),
PADRE= *padre*,
ACCIÓN = *acción*,
COSTE-CAMINO= *padre*.COSTE-CAMINO +
problema.COSTE-
PASO(*padre*.ESTADO, *acción*)

Estrategias de búsqueda

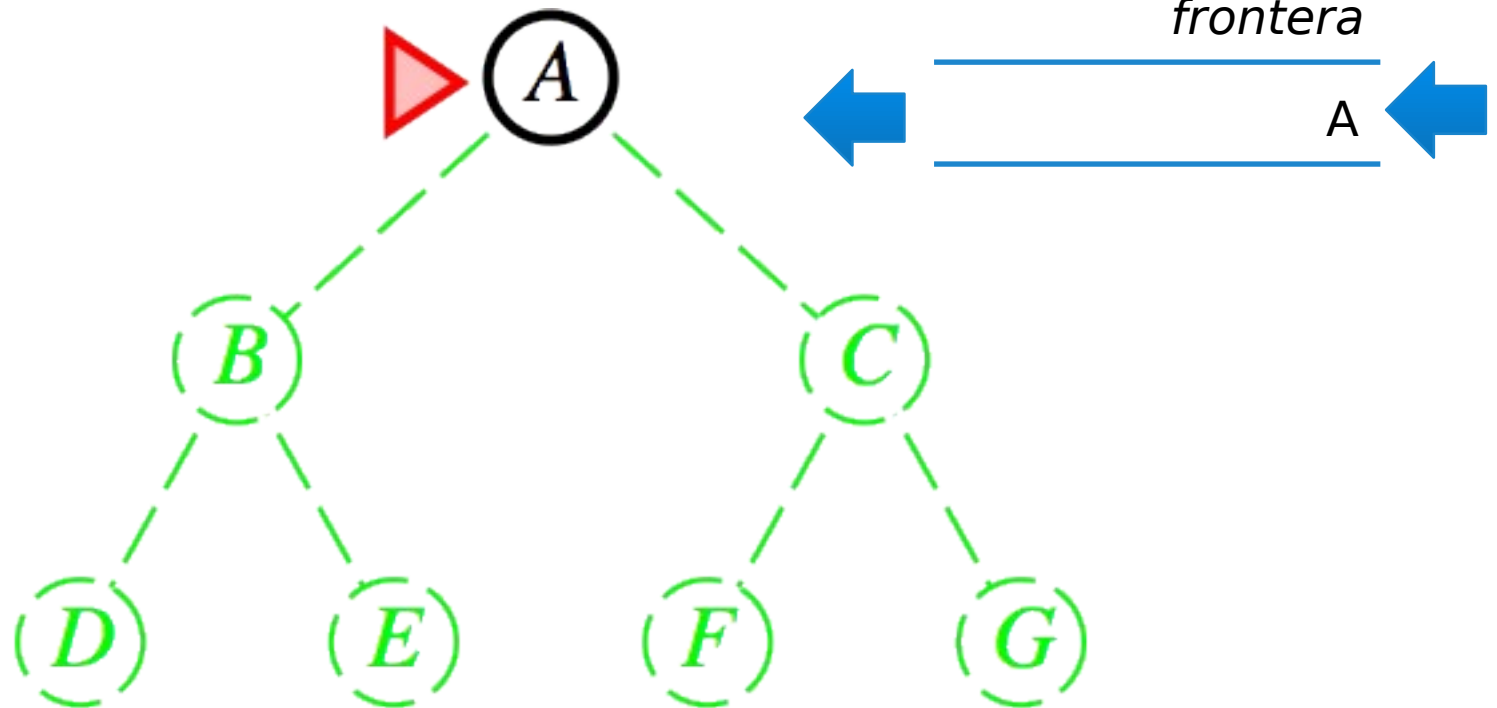
- Una estrategia viene definida por la elección del **orden de expansión de nodos**.
- Las **estrategias se evalúan** de acuerdo a estas 4 dimensiones:
 - Complejidad**: ¿Encuentra una solución si existe?
 - Optimalidad**: ¿Encuentra siempre la solución de menor coste?
 - Complejidad temporal**: ¿Número de **nodos generados/expandidos**?
 - Complejidad espacial**: ¿Número de **nodos almacenados** en memoria en la búsqueda?
- La **complejidad temporal y espacial** se miden **en términos de la dificultad** del problema mediante:
 - b*** -factor de ramificación máximo -> Número máximo de sucesores de un nodo
 - d*** - profundidad de la solución de menor coste.
 - m*** - máxima profundidad de cualquier camino en espacio de estados (puede ser ∞)

Estrategias no informadas (búsqueda ciega)

- Búsqueda ciega = utiliza sólo **información disponible en la definición del problema**
 - *Cuando las estrategias pueden determinar si un nodo que no es objetivo es mejor que otro ☑ búsqueda informada*
- Categorías definidas por el algoritmo de expansión:
 - Búsqueda primero en anchura / Breadth-first search (BFS)
 - Búsqueda coste uniforme / Uniform-cost search (UC)
 - Búsqueda primero en profundidad / Depth-first search (DFS)
 - Búsqueda profundidad limitada / Depth-limited search (DLS)
 - Búsqueda profundidad iterativa / Iterative deepening depth first search (IDS)
 - Búsqueda Bidireccional / Bidirectional Search

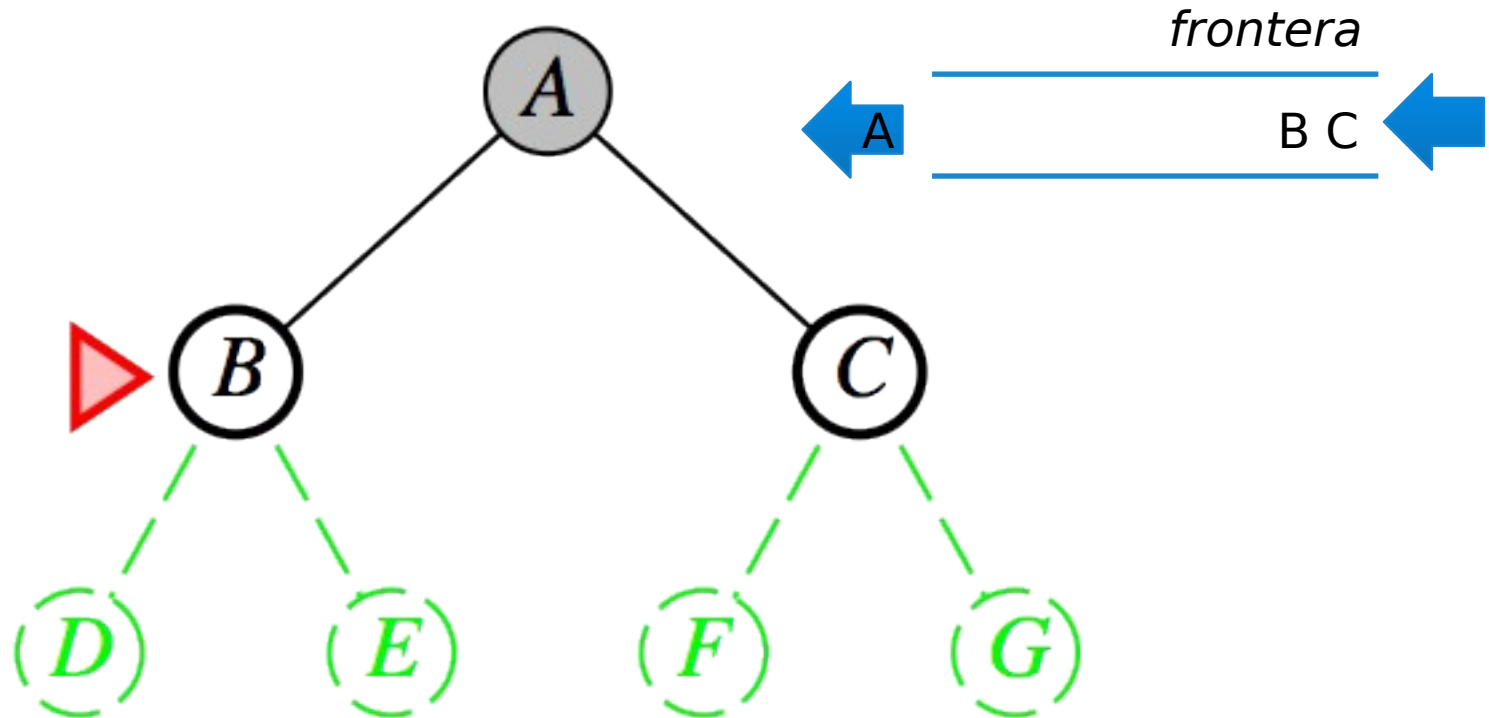
Primero en anchura, un ejemplo

- Expande el nodo no expandido menos profundo
- Implementación: *la frontera es una cola FIFO*



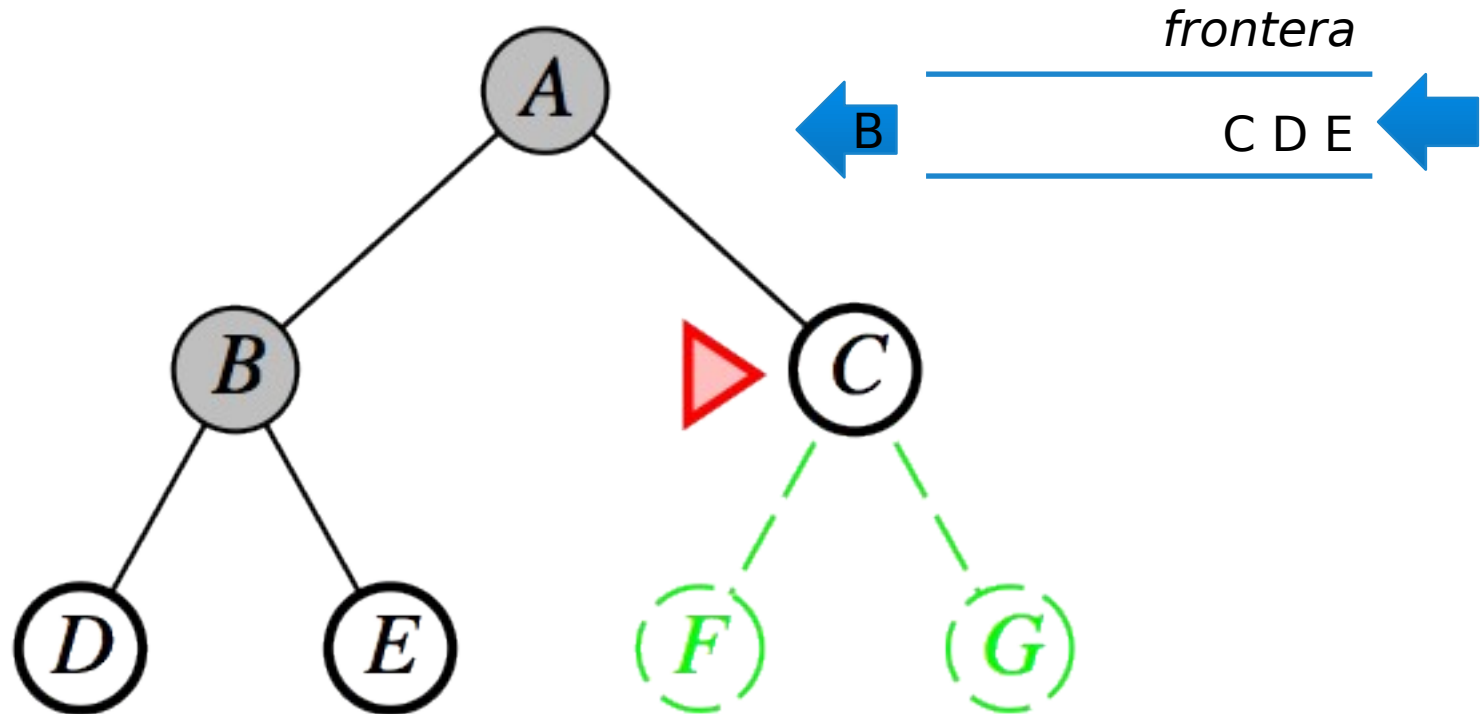
Primero en anchura, un ejemplo

- Expande el nodo no expandido menos profundo
- Implementación: *la frontera es una cola FIFO*



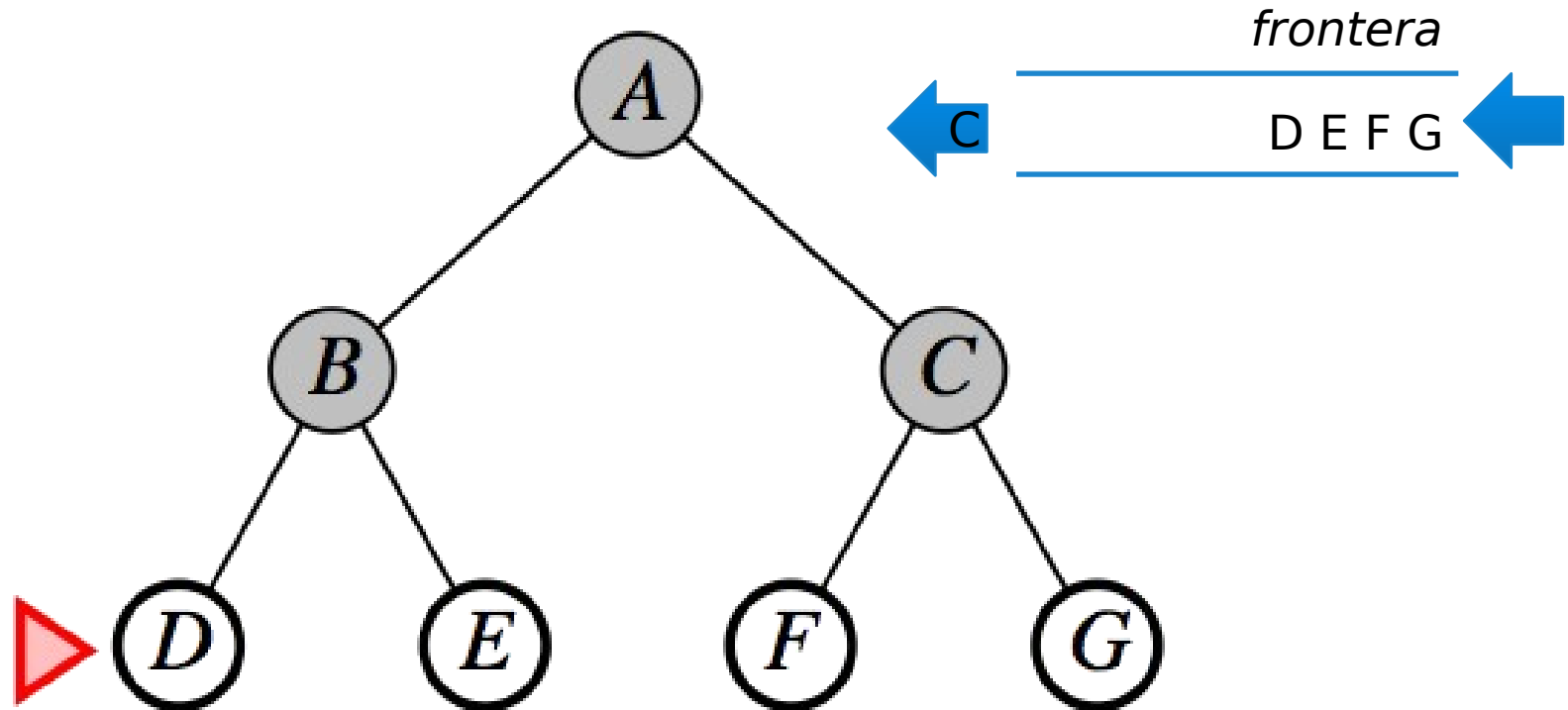
Primero en anchura, un ejemplo

- Expande el nodo no expandido menos profundo
- Implementación: *la frontera es una cola FIFO*



Primero en anchura, un ejemplo

- Expande el nodo no expandido menos profundo
- Implementación: *la frontera es una cola FIFO*



Primero en Achura. Búsqueda en grafo.

function BÚSQUEDA-PRIMERO-ANCHURA(*problema*) **returns** solución o fallo

nodo ← un nodo con ESTADO= *problema*.ESTADO-INICIAL,
COSTE-CAMINO= 0



if *problema*.TEST-OBJETIVO(*nodo*.ESTADO) **then return**
SOLUCION(*nodo*)

frontera ← una **FIFO** con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return** fallo

nodo ← **POP**(*frontera*) /*Elige el nodo menos profundo de la
frontera */

add *nodo*.ESTADO a *explorado*

for each *acción* **in** *problema*.ACCIONES(*nodo*.ESTADO) **do**

hijo ← NODO-HIJO(*problema*, *nodo*, *accion*)

if *hijo*.ESTADO no está en *explorado* o *frontera* **then**

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO)

Primero en Achura. Búsqueda en grafo.

function BÚSQUEDA-PRIMERO-ANCHURA(*problema*) **returns** solución o fallo

nodo ← un nodo con ESTADO= *problema*.ESTADO-INICIAL,
COSTE-CAMINO= 0

if *problema*.TEST-OBJETIVO(*nodo*.ESTADO) **then return**
SOLUCION(*nodo*)

frontera ← una FIFO con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return** fallo

nodo ← POP(*frontera*) /*Elige el nodo menos profundo de la
frontera */

add *nodo*.ESTADO a *explorado*

for each acción in *problema*.ACCIONES(*nodo*.ESTADO) **do**

hijo ← NODO-HIJO(*problema*, *nodo*, acción)

if *hijo*.ESTADO no está en *explorado* o *frontera* **then**

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO)

Evaluación: Primero en anchura

■ Completitud:

¿Encuentra siempre una solución? SI

- Si el nodo objetivo menos profundo está en una profundidad finita **d**
- Condición: SI b es finito
 - (núm. de nodos encontrados es finito)

Búsqueda en anchura- evaluación

■ Optimalidad:

¿Encuentra siempre la solución optima? En general
SI

- **Si el coste del camino es una función no decreciente de la profundidad.**
- El objetivo menos profundo no tiene porque ser el optimo si las acciones tienen diferente coste.

Evaluación: Primero en anchura

■ Complejidad temporal:

- Asumiendo un espacio de estados en el que cada estado tienen b sucesores.
- Raíz tiene b sucesores, cada nodo en el siguiente nivel tiene de nuevo b sucesores, ... Asumiendo que la solución tiene una profundidad d , en el peor caso generamos todos los nodos del último nivel

$$b + b^2 + b^3 + \dots + b^d = O(b^d)$$



Suponiendo que el algoritmo aplica el **test del objetivo** cuando se **genera** el nodo, y no cuando va a ser expandido

Evaluación: Primero en anchura

■ Complejidad temporal:

- Asumiendo un espacio de estados en el que cada estado tienen b sucesores.
- Raíz tiene b sucesores, cada nodo en el siguiente nivel tiene de nuevo b sucesores, ... Asumiendo que la solución tiene una profundidad d , en el peor caso generamos todos los nodos del último nivel

$$b + b^2 + b^3 + \dots + b^d = O(b^d)$$

- Peor caso; expandir todos menos el último nodo de profundidad d

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

Suponiendo
que el
algoritmo

**aplica el test
del objetivo
cuando se
expande el
nodo**

Evaluación: Primero en anchura

■ Complejidad Espacial: $O(b^d)$

- En la búsqueda, cada nodo generado permanece en memoria
 - En búsqueda en **grafo** habrá $O(b^{d-1})$ nodos en el conjunto explorado (cerrados) y $O(b^d)$ en la frontera (abiertos). Por lo tanto la complejidad espacial viene dada por el tamaño de nodos en la frontera.
 - Si en lugar de un grafo, tuviéramos un **árbol** (no guardamos el conjunto explorado), la complejidad espacial sería la misma (no ahorraríamos mucho espacio). En un espacio con muchos caminos redundantes perderíamos mucho tiempo.

Evaluación: Primero en anchura (BFS, breadth first search)

■ Completitud:

- SI (si b es finito)

■ Complejidad Temporal

- Total de nodos generados:

$$b + b^2 + b^3 + \dots + b^d + (b^d) = O(b^d)$$

■ Complejidad Espacial

- Nodos en la frontera
nodos en el conjunto explorado)

$$O(b^d) \quad (y$$

■ Optimalidad:

- En general Si. Si el coste del camino es una función no decreciente de la profundidad. p.e todas las acciones el mismo coste.



Búsqueda en anchura- evaluación

- Los **requisitos de memoria** son el problema
- La complejidad exponencial de los problemas de búsqueda no puede ser resuelta por la búsqueda ciega salvo para ejemplos pequeños.



| Profundidad | Nodos | Tiempo | Memoria |
|--|-----------|------------------|----------------|
| 2 | 110 | 1.1 milisegundos | 107 Kilobytes |
| 4 | 11.110 | 111 milisegundos | 10.6 megabytes |
| 6 | 10^6 | 11 segundos | 1 gigabyte |
| 8 | 10^8 | 19 minutos | 103 gigabytes |
| 10 | 10^{10} | 31 horas | 10 terabytes |
| 12 | 10^{12} | 129 días | 1 perabyte |
| 14 | 10^{14} | 35 años | 99 petabytes |
| 16 | 10^{16} | 3500 años | 1 exabyte |
| Complejidad temporal y espacial para una búsqueda en anchura de factor de ramificación b=10 , 1 millón de nodos generados/segundo, 1000 bytes por nodo. | | | |

Búsqueda coste uniforme

- Extensión de búsqueda en anchura:
 - Modificamos el algoritmo para que sea **óptimo para cualquier función de coste** en cada paso
 - Expandimos nodo con menor coste
- Implementación: *frontera* = cola ordenada por **coste de caminos**.
- Búsqueda-CU y Búsqueda-Anchura son iguales cuando todos los costes de las operaciones son iguales.

Coste Uniforme. Búsqueda en grafo.

function BÚSQUEDA-COSTE-UNIFORME(*problema*) **returns** solución o fallo

nodo ← un nodo con ESTADO = *problema*.ESTADO-INICIAL, COSTE-CAMINO = 0

frontera ← cola ordenada por COSTE-CAMINO, con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return** fallo

nodo ← POP(*frontera*) /*Elige el nodo de menor coste de la frontera */

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO) **then return** SOLUCIÓN(*hijo*)

 add *nodo*.ESTADO a *explorado*

for each acción in *problema*.ACCIONES(*nodo*.ESTADO) **do**

hijo ← NODO-HIJO(*problema*, *nodo*, acción)

if *hijo*.ESTADO no está en *explorado* o *frontera* **then**

frontera ← INSERT(*hijo*, *frontera*)

Coste Uniforme. Búsqueda en grafo.

function BÚSQUEDA-COSTE-UNIFORME(*problema*) **returns** solución o fallo

nodo ← un nodo con ESTADO= *problema*.ESTADO-INICIAL, COSTE-CAMINO=0

frontera ← cola ordenada por COSTE-CAMINO, con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return** fallo

nodo ← POP(*frontera*) /*Elige el nodo de menor coste de la frontera */

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO) **then return** SOLUCIÓN(*hijo*)

 add *nodo*.ESTADO a *explorado*

for each acción in *problema*.ACCIONES(*nodo*.ESTADO) **do**

hijo ← **Test cuando es expandido.**

if *hijo*. **Los nodos de menor coste ya habrán sido expandido**

frontera ← INSERT(*hijo*, *frontera*)

Primero en Achura. Búsqueda en grafo.

function BÚSQUEDA-PRIMERO-ANCHURA(*problema*) **returns** solución o fallo

nodo ← un nodo con ESTADO= *problema*.ESTADO-INICIAL,
COSTE-CAMINO= 0

if *problema*.TEST-OBJETIVO(*nodo*.ESTADO) **then return**
SOLUCION(*nodo*)

frontera ← una FIFO con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return** fallo

nodo ← POP(*frontera*) /*Elige el nodo menos profundo de la
frontera */

add *nodo*.ESTADO a *explorado*

for each acción in *problema*.ACCIONES(*nodo*.ESTADO) **do**

hijo ← NODO-HIJO(*problema*, *nodo*, acción)

if *hijo*.ESTADO no está en *explorado* o *frontera* **then**

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO)

Coste Uniforme. Búsqueda en grafo.

function BÚSQUEDA-COSTE-UNIFORME(*problema*) **returns** solución o fallo

nodo ← un nodo con ESTADO= *problema*.ESTADO-INICIAL, COSTE-CAMINO=0

frontera ← cola ordenada por COSTE-CAMINO, con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return** fallo

nodo ← POP(*frontera*) /*Elige el nodo de menor coste de la frontera */

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO) **then return**

SOLUCIÓN(*hijo*)

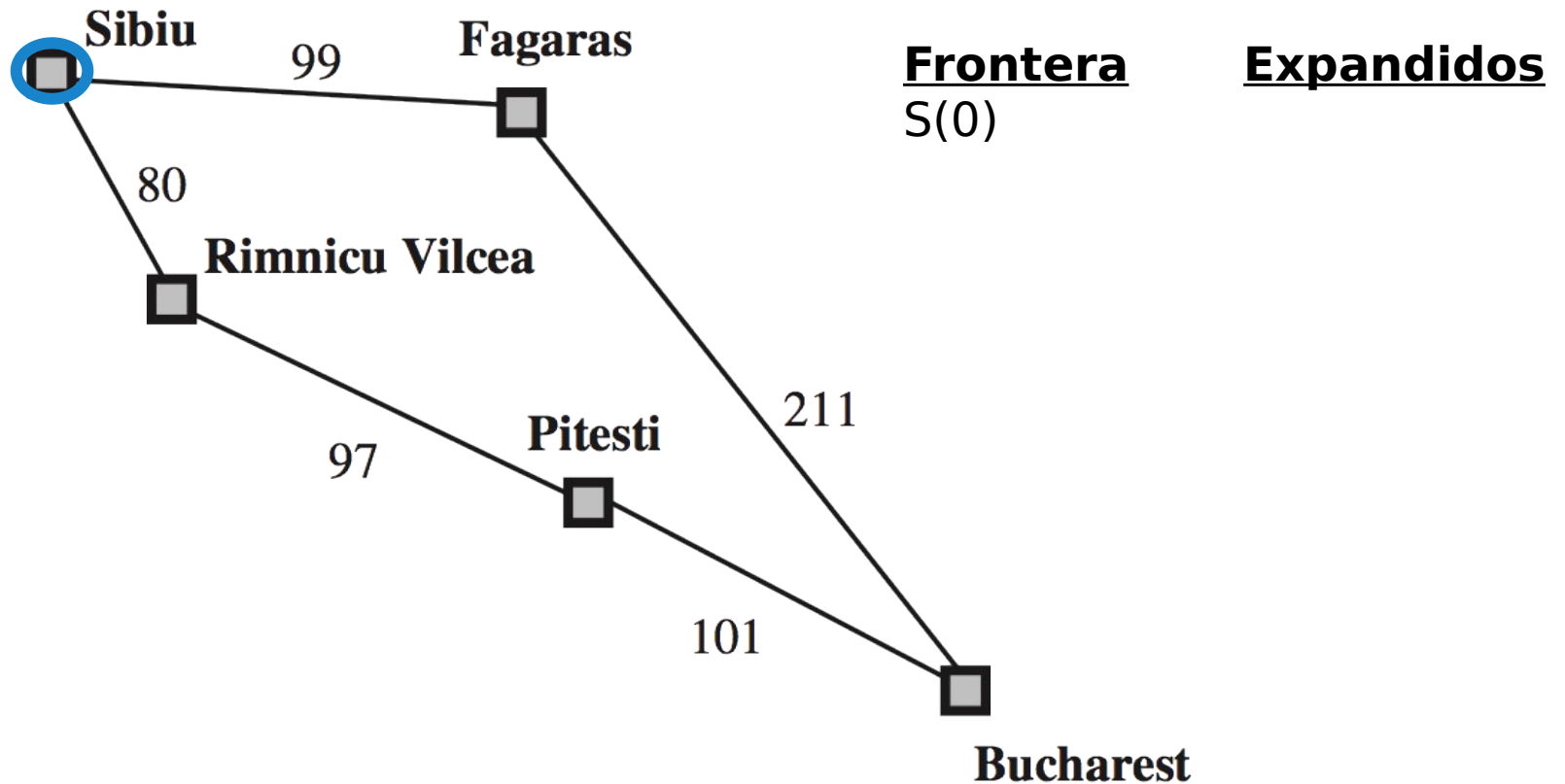
 add *nodo*.ESTADO a *explorado*

for each acción in *problema*.ACCIONES(*nodo*.ESTADO) **do**

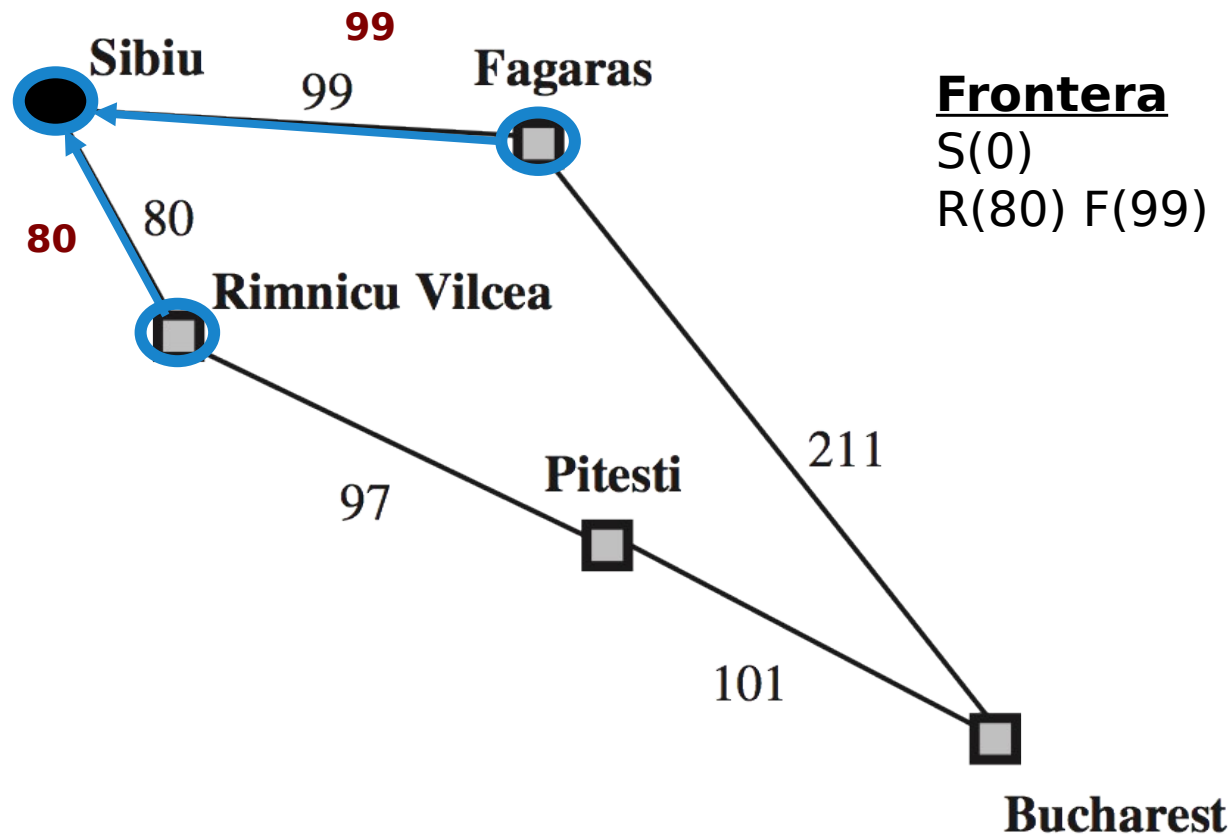
hijo ← **Actualizamos coste de los nodos con el del camino d**
 if *hijo*. menor coste

frontera ← INSERT(*hijo*, *frontera*)

Búsqueda coste uniforme



Búsqueda coste uniforme



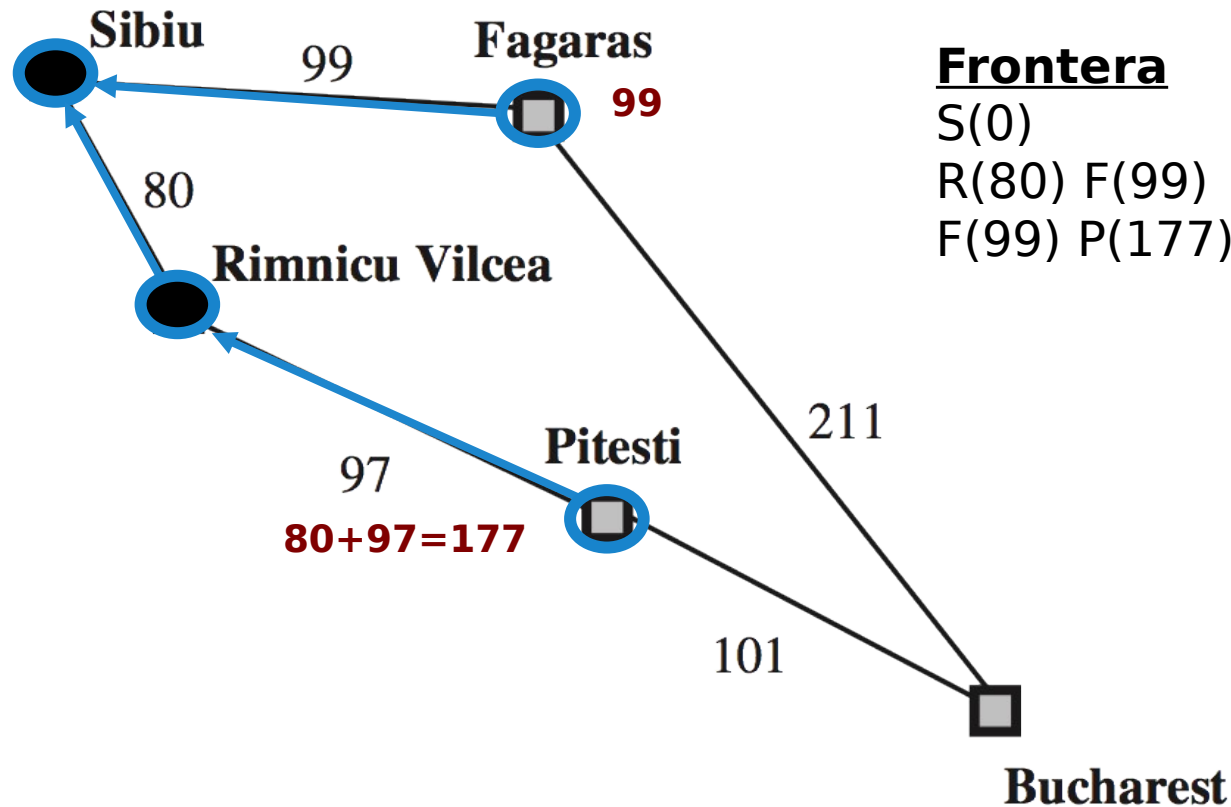
Frontera

S(0)
R(80) F(99)

Expandidos

S(0)

Búsqueda coste uniforme



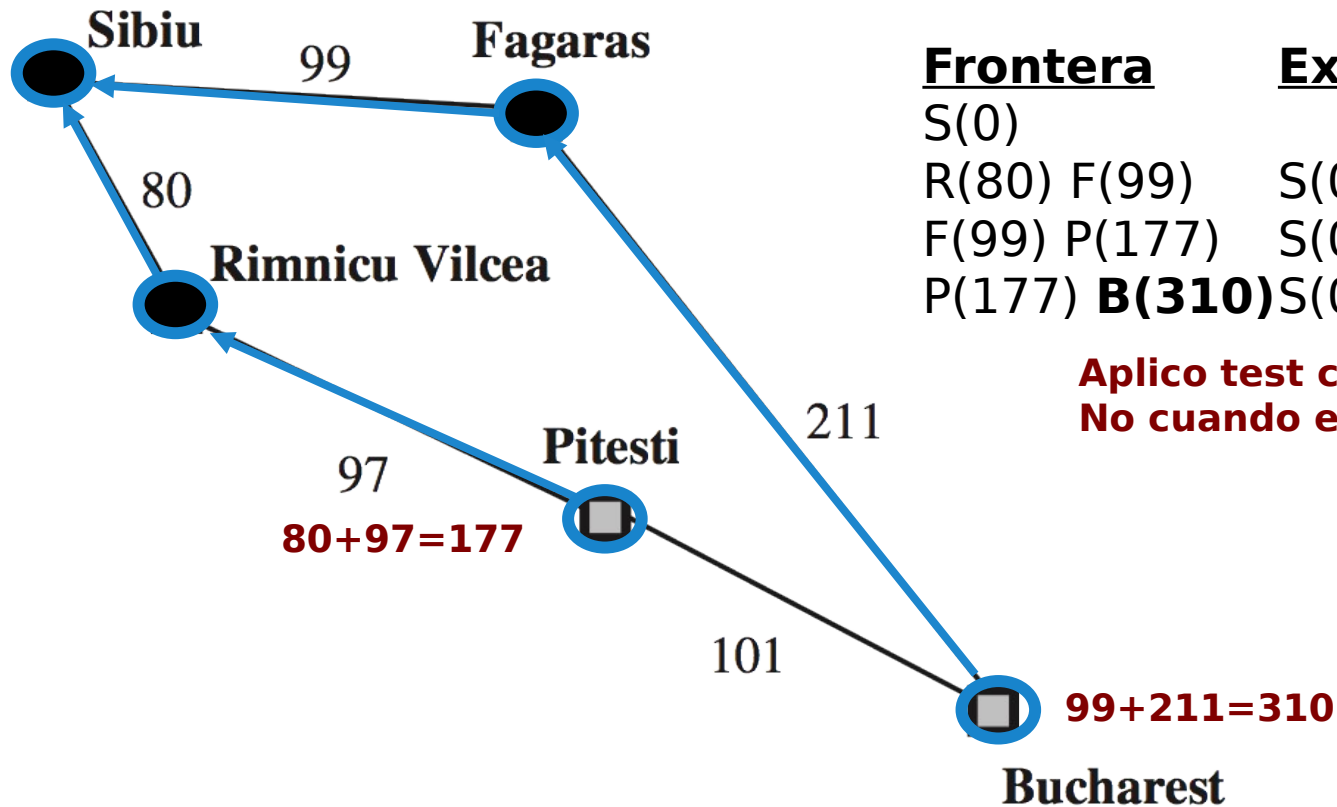
Frontera

S(0)
R(80) F(99)
F(99) P(177)

Expandidos

S(0)
S(0) R(80)

Búsqueda coste uniforme



Frontera

S(0)

R(80) F(99)

F(99) P(177)

P(177) **B(310)**

Expandidos

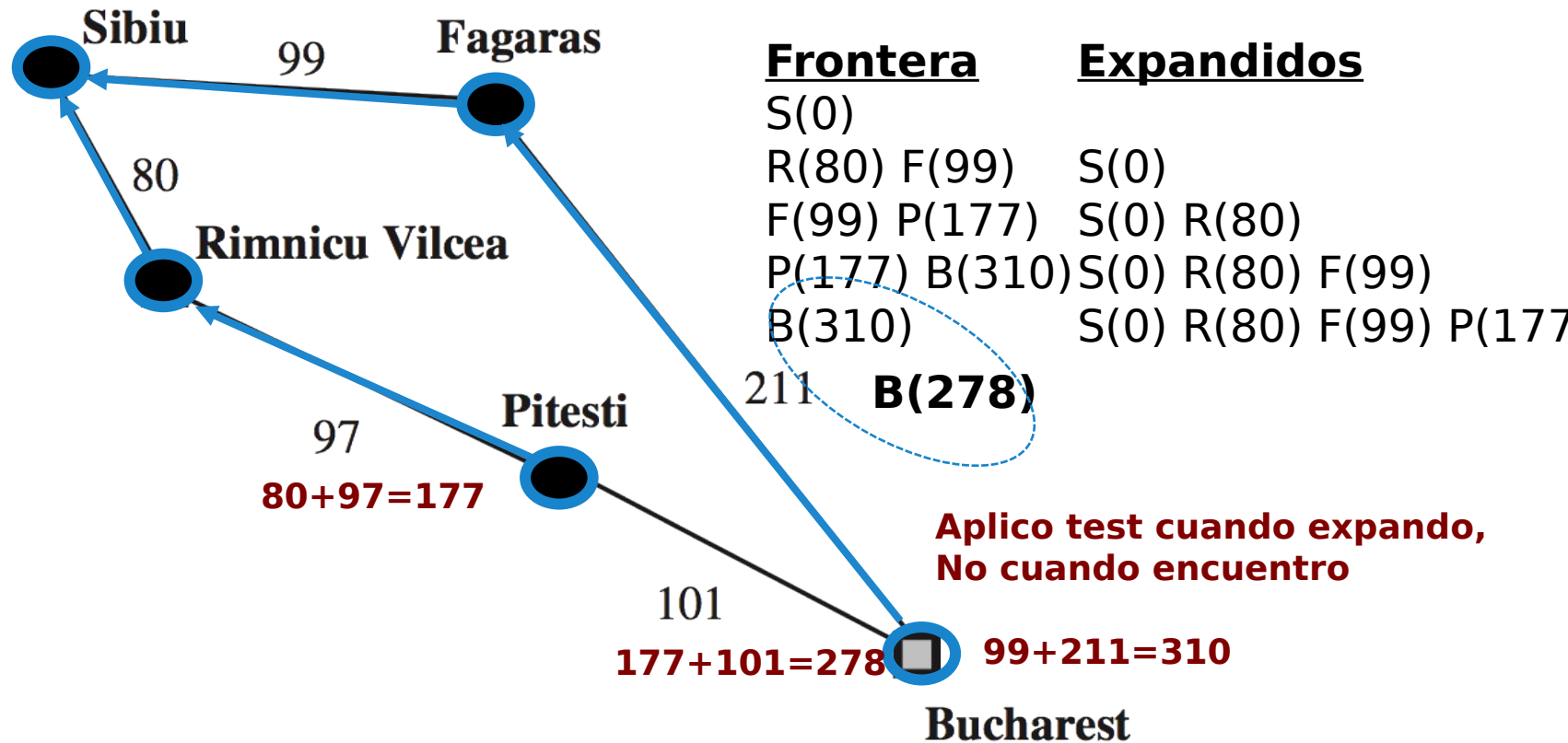
S(0)

S(0) R(80)

S(0) R(80) F(99)

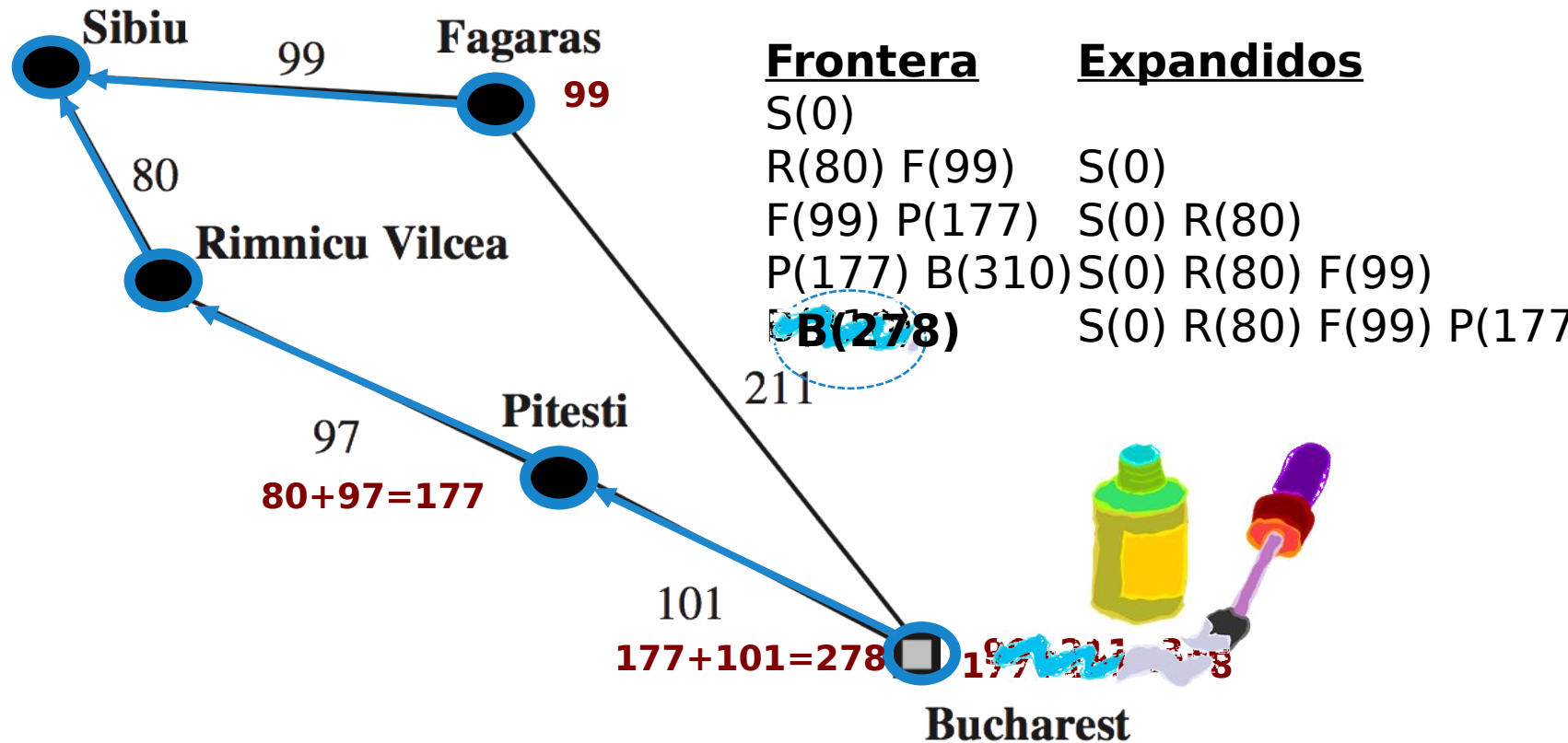
**Aplico test cuando expando,
No cuando encuentro**

Búsqueda coste uniforme



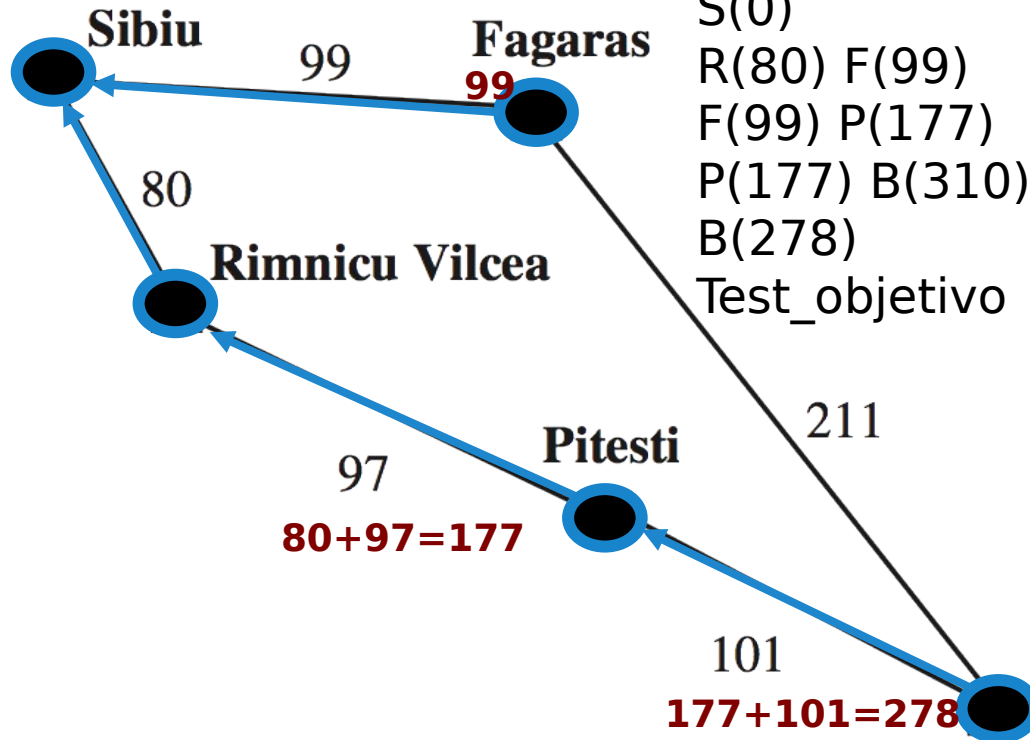
Búsqueda coste uniforme

B(278)

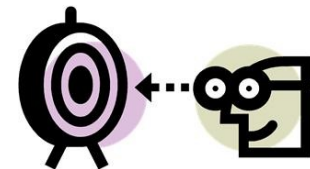


Aplico test cuando expando,
No cuando encuentro

Búsqueda coste uniforme



| <u>Frontera</u> | <u>Expandidos</u> |
|-----------------|-------------------------|
| S(0) | |
| R(80) F(99) | S(0) |
| F(99) P(177) | S(0) R(80) |
| P(177) B(310) | S(0) R(80) F(99) |
| B(278) | S(0) R(80) F(99) P(177) |
| Test_objetivo | S(0) R(80) F(99) P(177) |
| | B(278) |



Bucharest

Aplico test cuando expando, No cuando encuentro

Evaluación: Coste uniforme

Es óptima

- ***Cuando elegimos un nodo para expandir, el camino óptimo a ese nodo ya ha sido encontrado***
 - Si el nodo a expandir ***n*** no tuviera el menor coste, tendría que haber otro nodo en la frontera ***n'*** **en el camino óptimo del inicio a *n***, pero entonces tendría menor coste y debería haber sido seleccionado antes.
- Como los pasos son no negativos, los caminos nunca tienen menor coste al añadir un nodo.
- La búsqueda uniforme expande nodos en orden de sus caminos óptimos

Evaluación: Coste uniforme

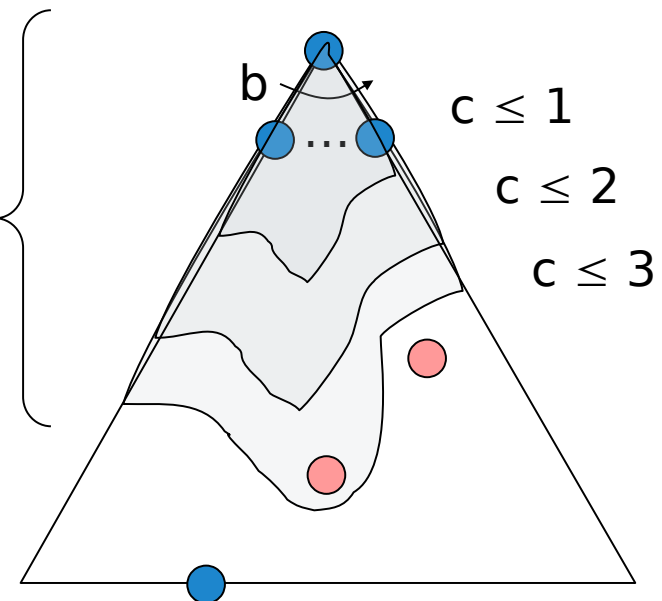
Completitud:

- La búsqueda de coste uniforme no se preocupa del número de pasos. Si las acciones tienen coste cero puede quedar atascado en un bucle infinito. Por ejemplo, una secuencia infinita de operaciones **NoOp**.
- La completitud queda garantizada si cada paso tiene al menos un coste mayor o igual a un pequeño valor constante positivo ϵ

Complejidad

- No es fácilmente caracterizarla en términos de a y b .
- En su lugar,
 - sea C^* sea el coste de la solución óptima y asumamos que cada acción cuesta al menos ϵ
 $\Rightarrow C^*/\epsilon$ es el número de pasos necesarios en el peor caso (+ 1 test al expandir).

C^*/ε “niveles”



Evaluación: Coste uniforme

■ Completitud:

- Si, si coste-paso $> \varepsilon$ (constante positiva pequeña)

■ Complejidad temporal:

- Asumimos C^* coste solución optima.
- Asumimos que cada acción cuesta al menos ε
- Peor caso es:

$$O(b^{1+(C^*/\varepsilon)})$$

■ Complejidad espacial

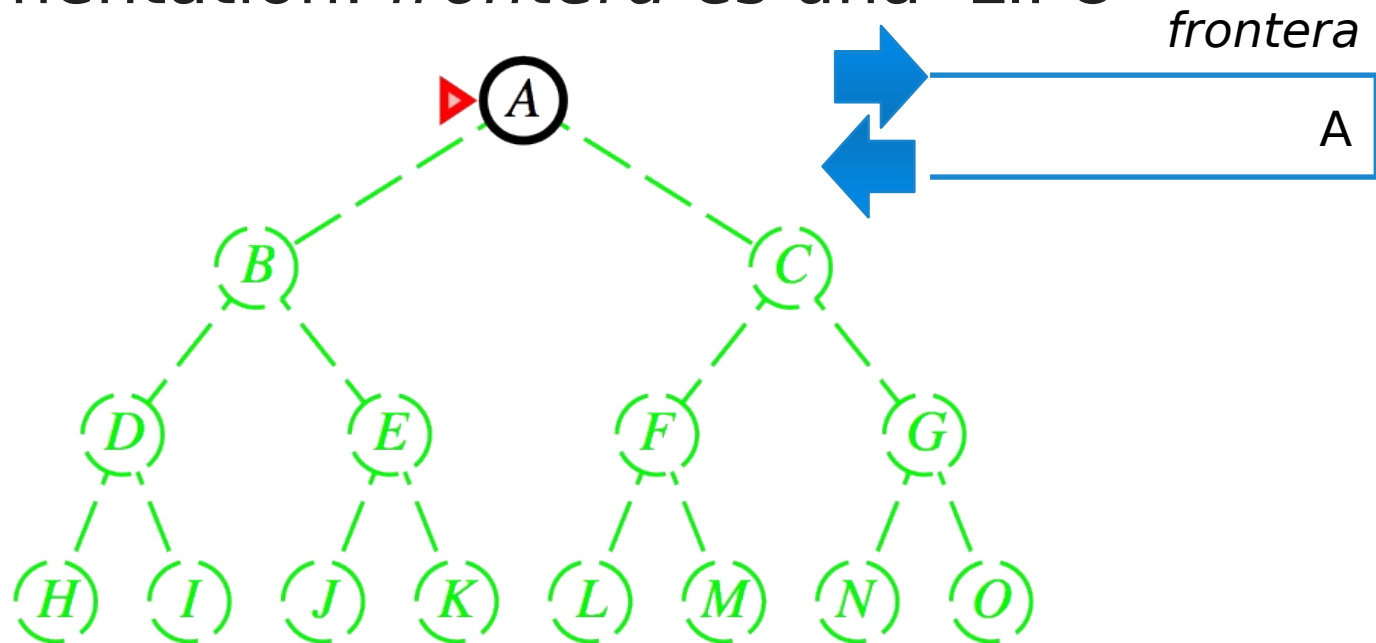
- Idem complejidad temporal

■ Optimalidad:

- Nodos expandidos en orden de coste creciente.
- SI, si completa.

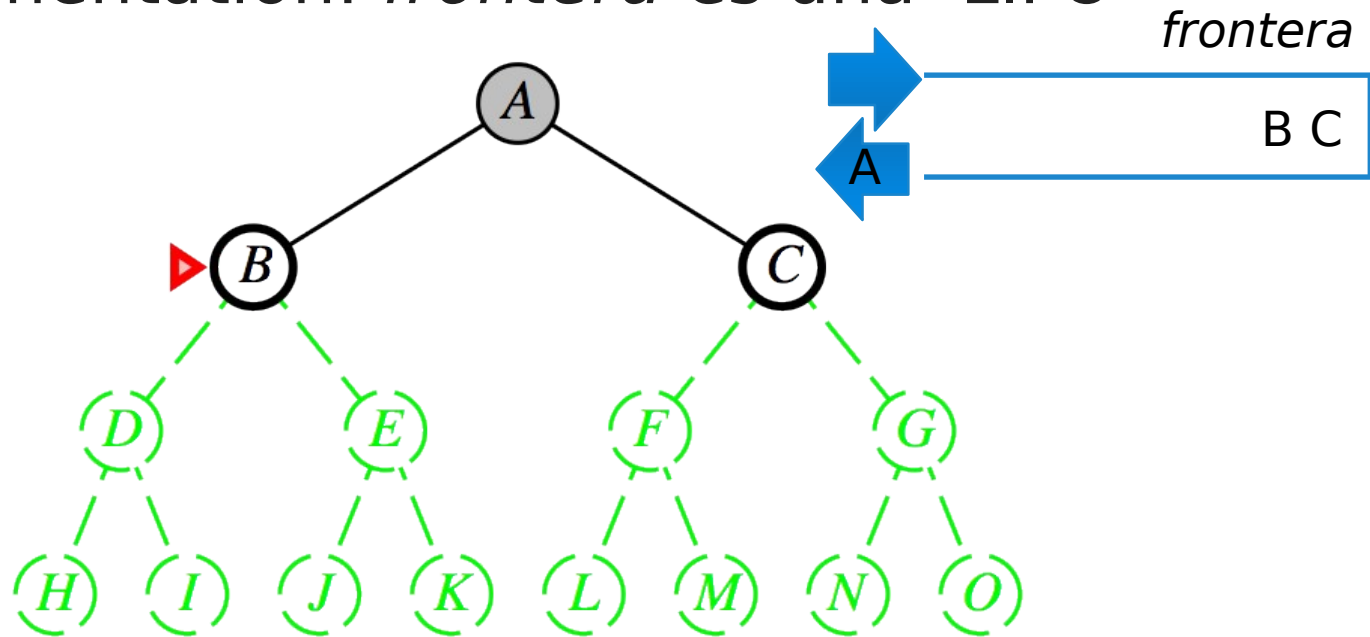
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



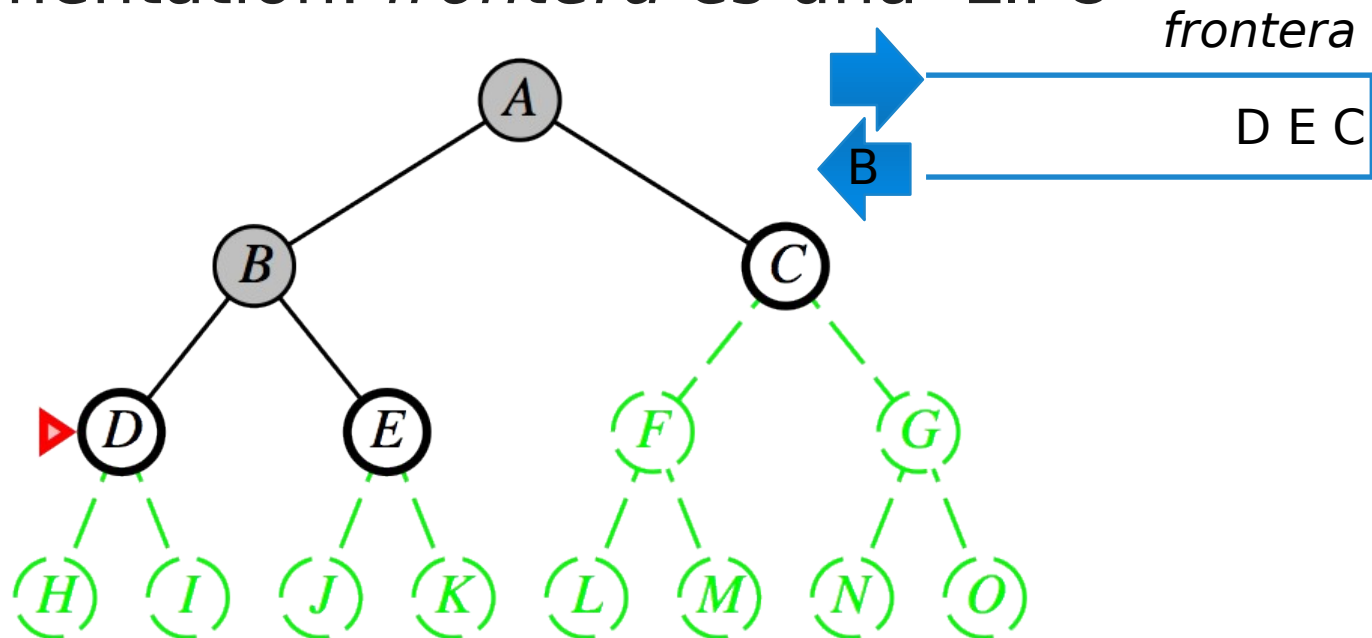
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



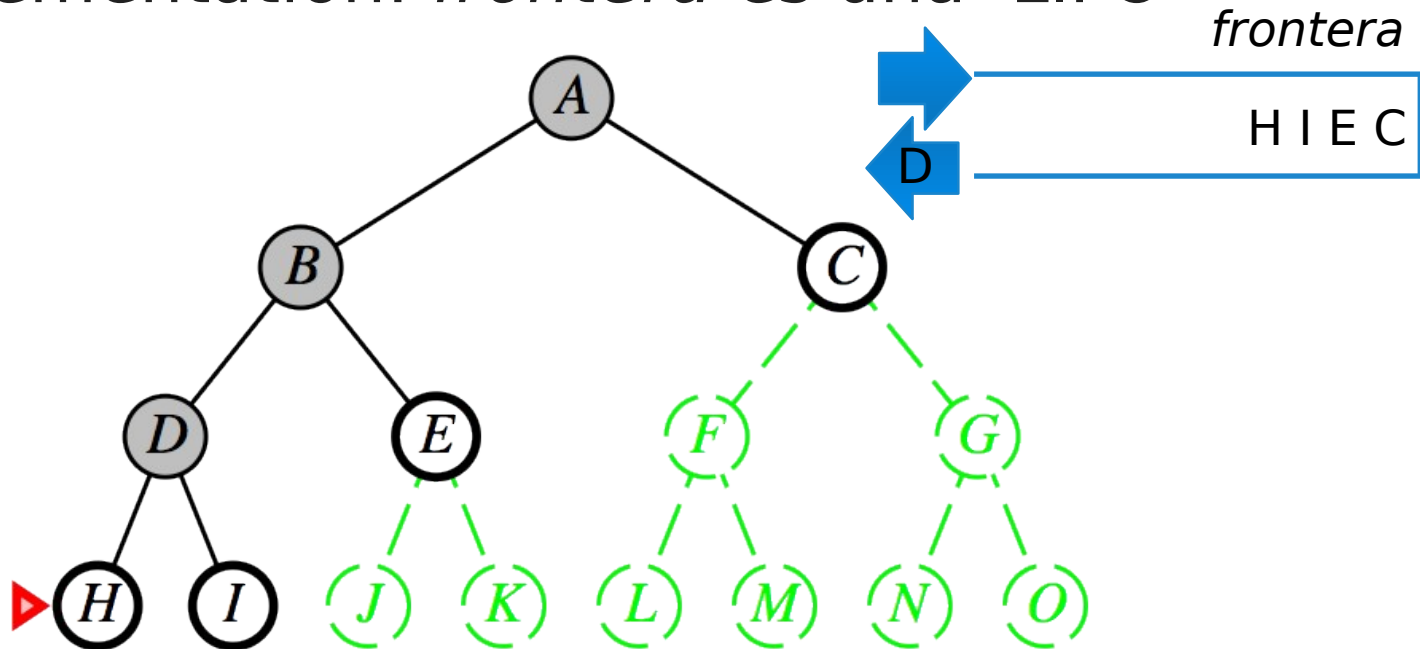
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



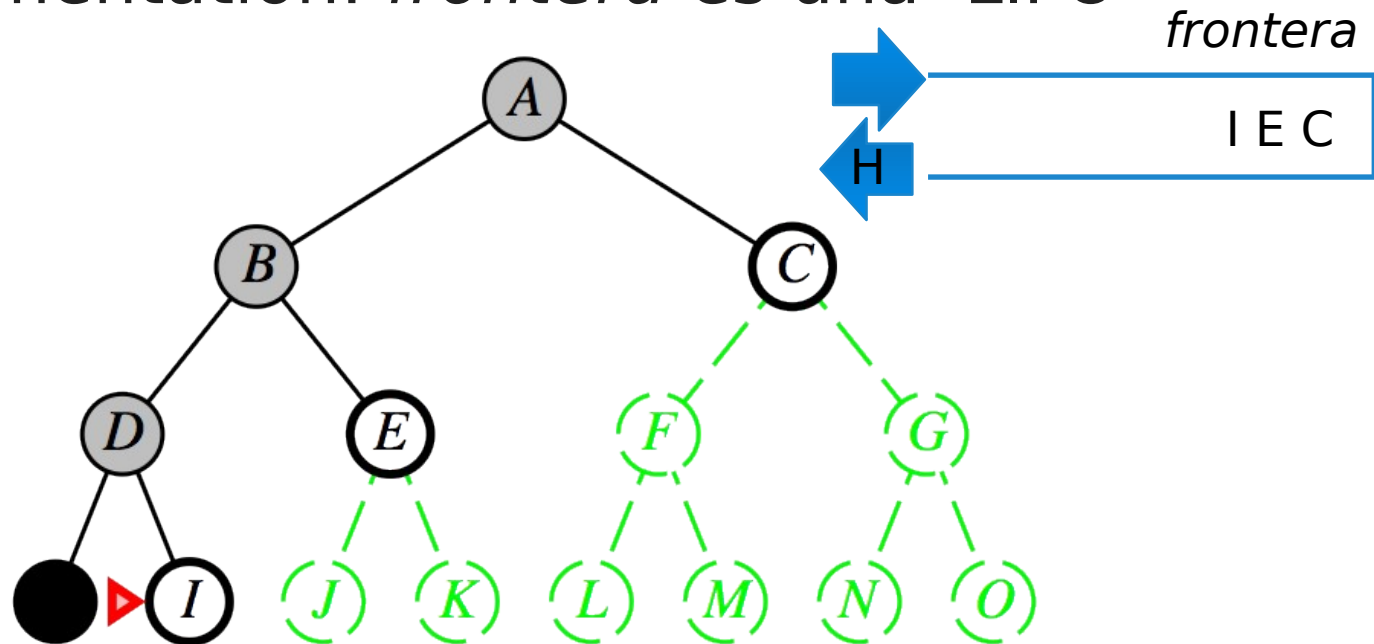
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



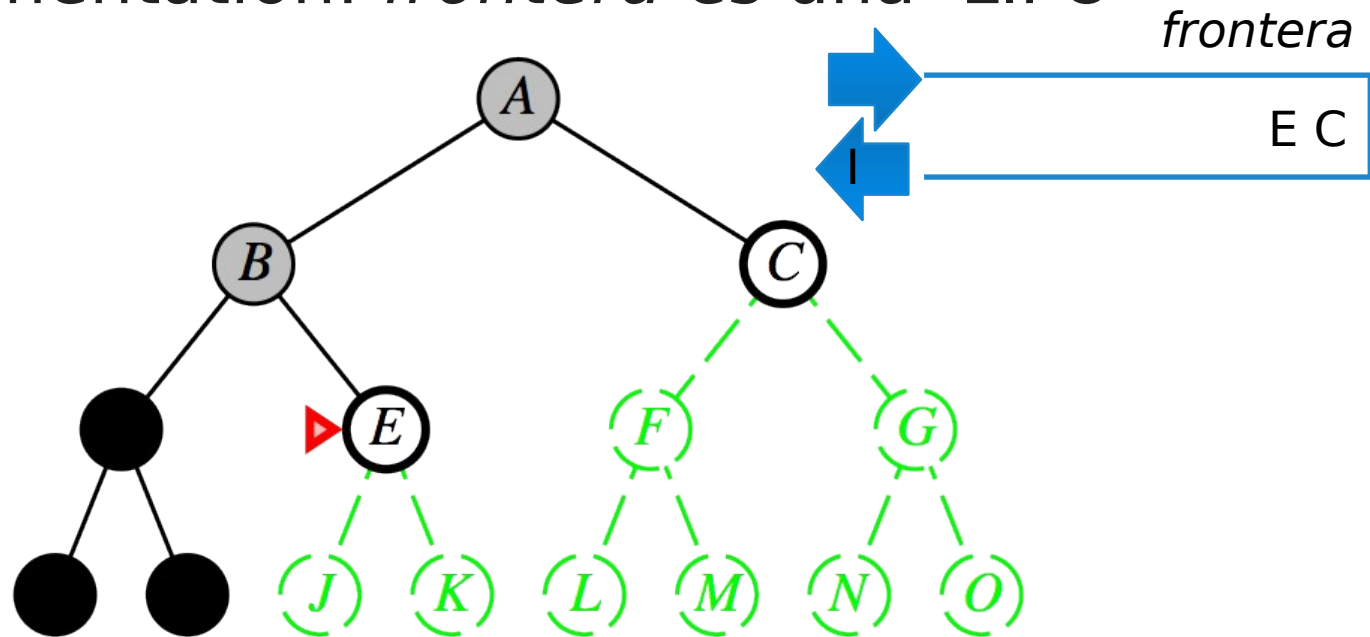
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



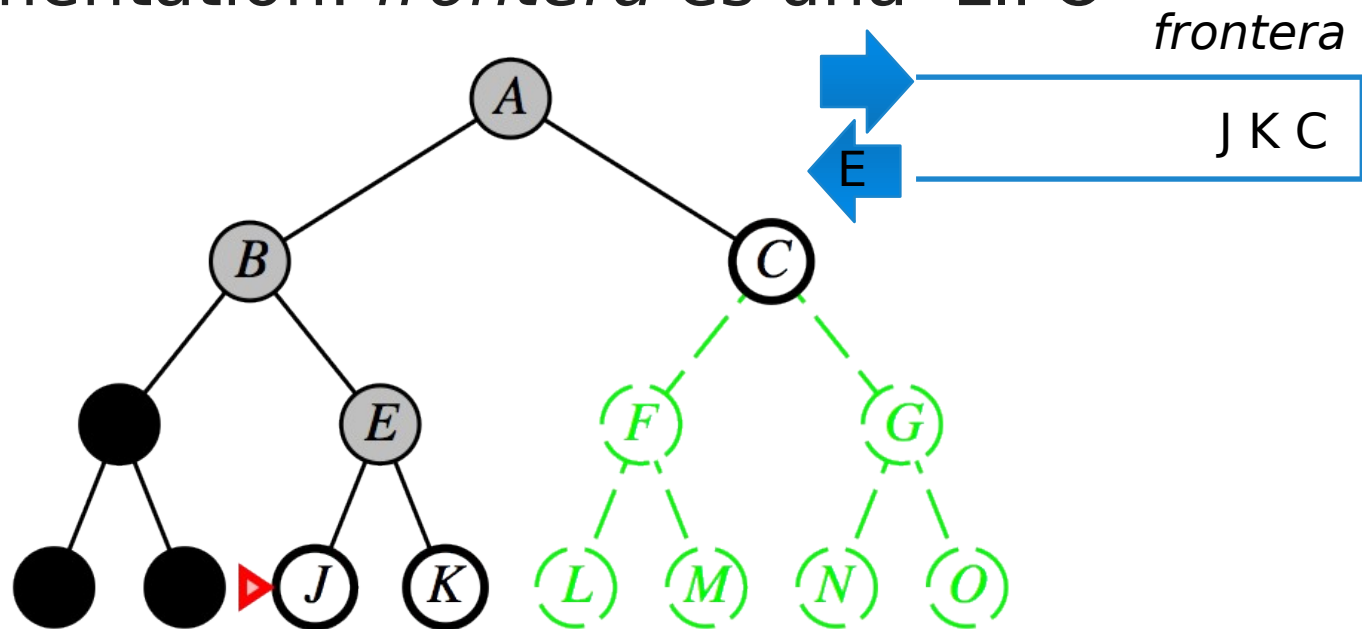
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



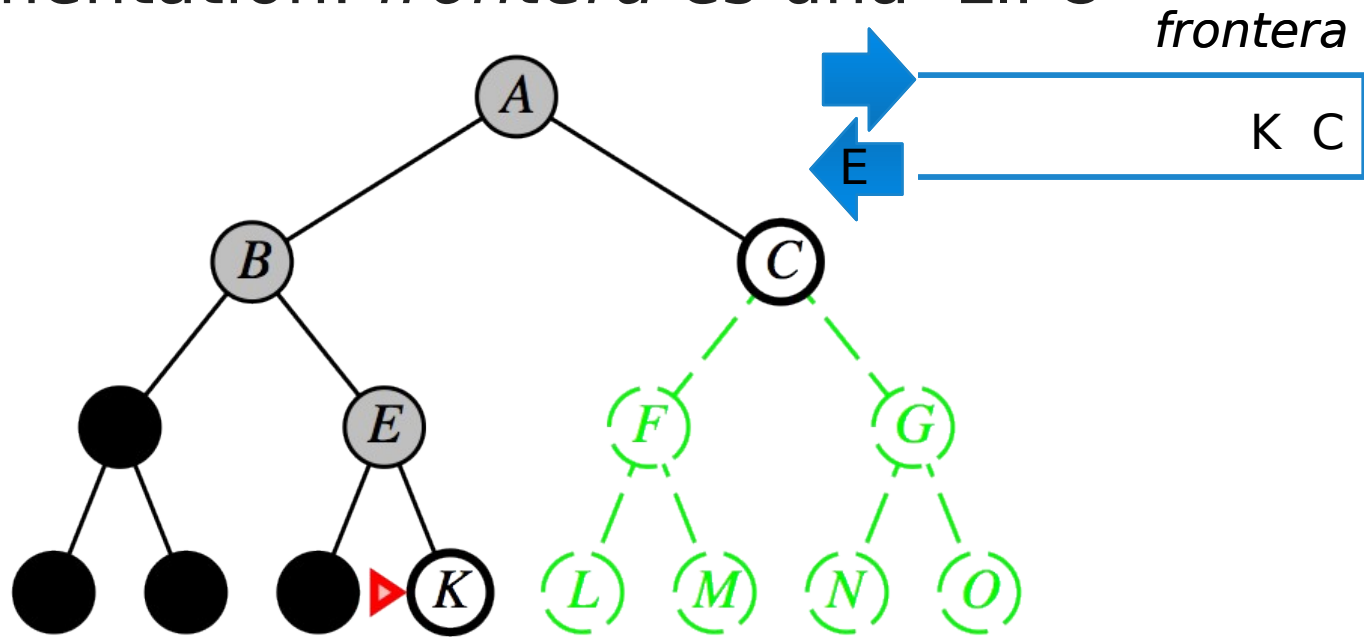
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



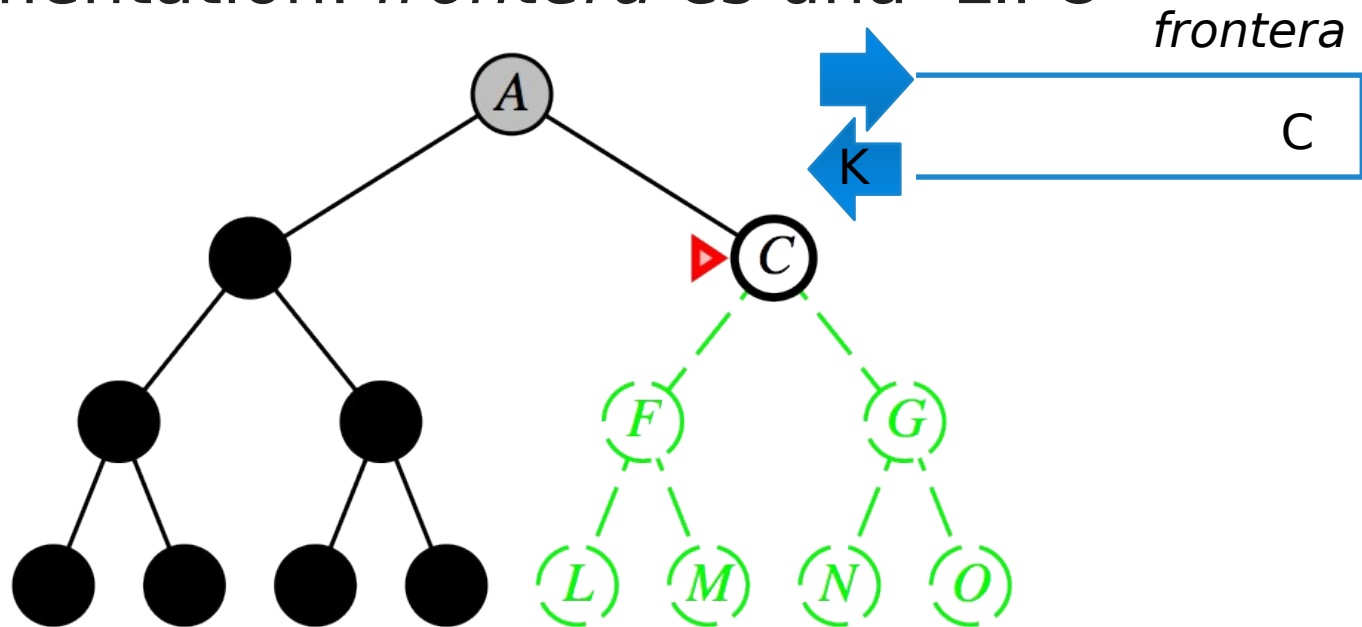
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



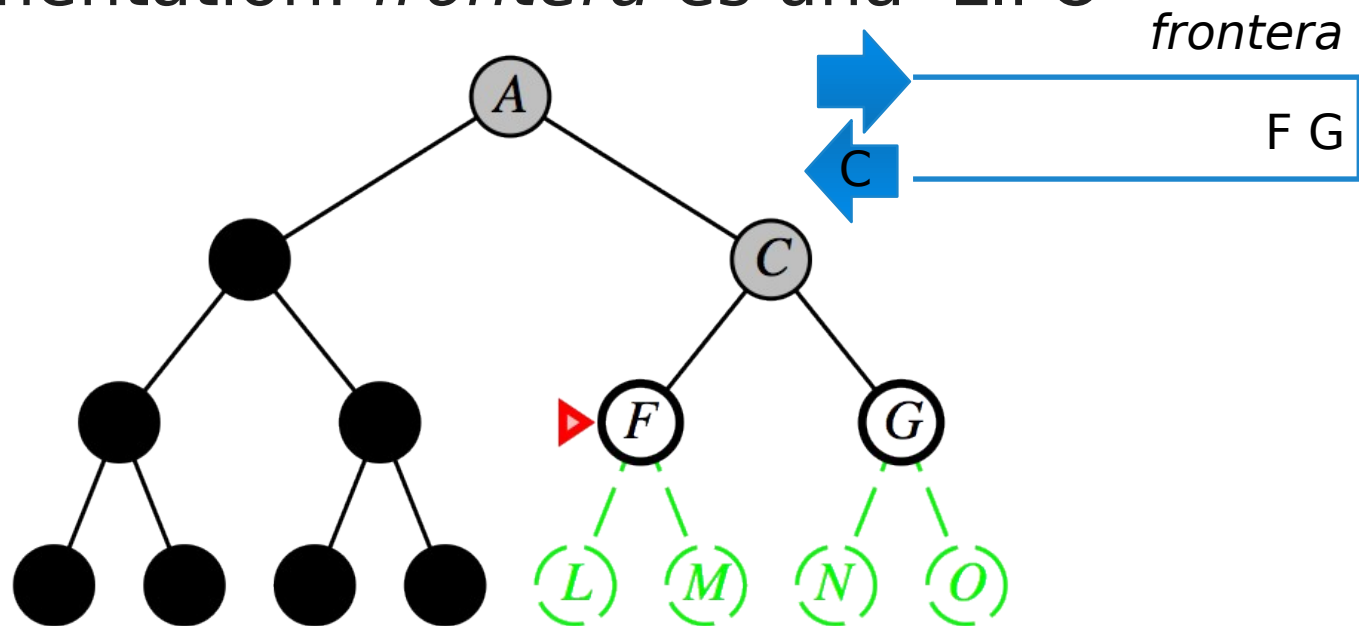
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



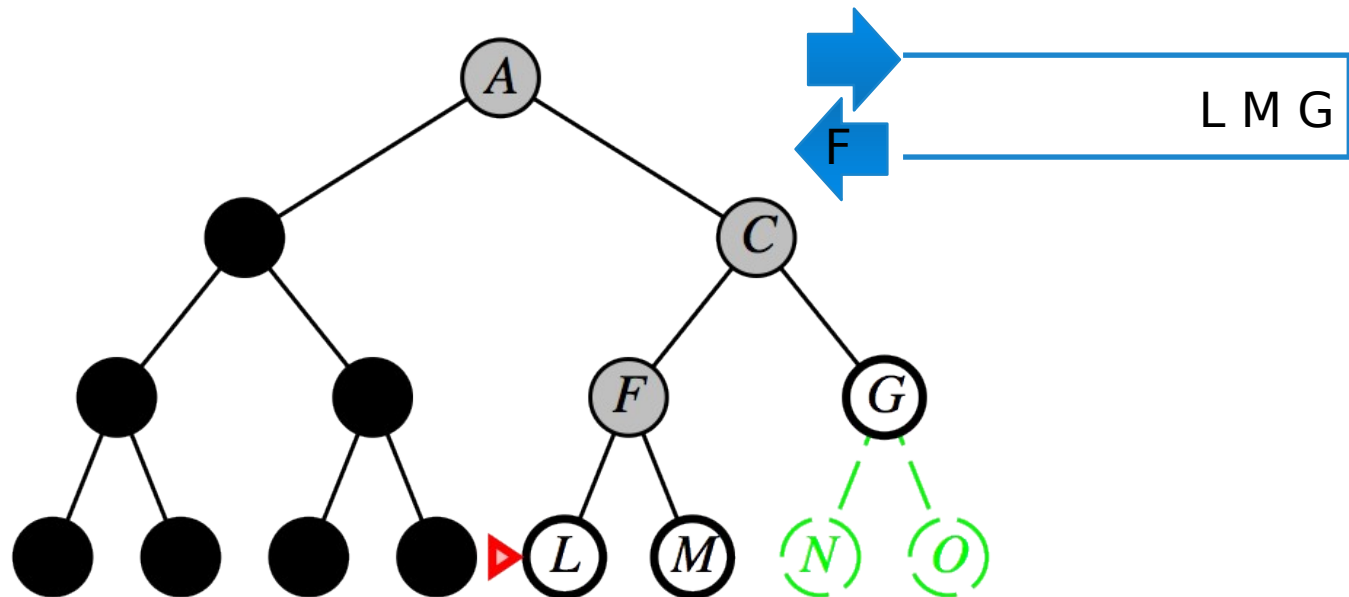
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



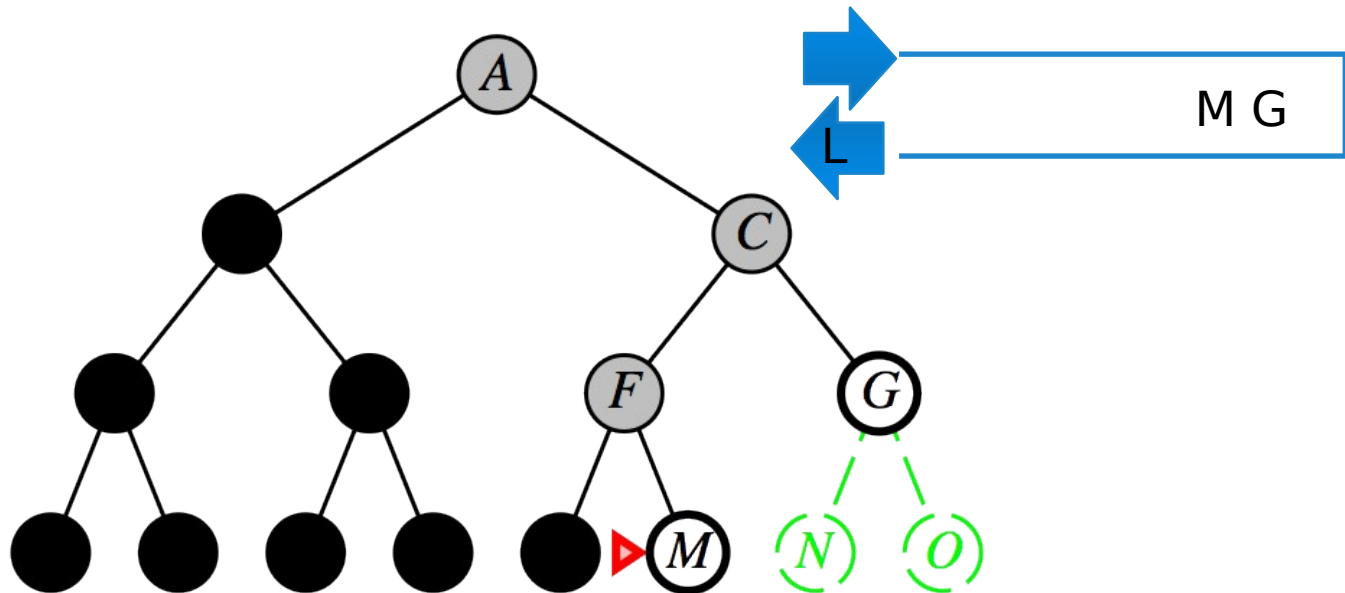
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



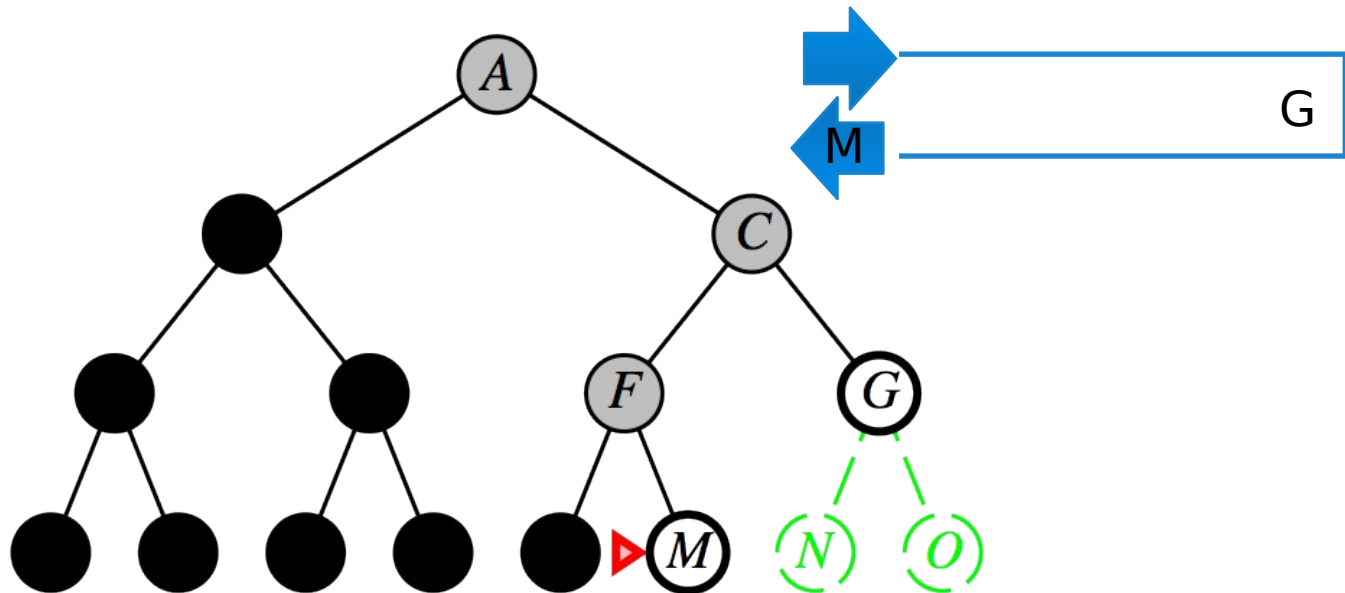
Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



Primero en profundidad, Ejemplo

- Expande nodo más profundo primero
- Implementation: *frontera* es una LIFO



Primero Profundidad. Búsqueda grafo.

function BÚSQUEDA-PRIMERO-PROFUNDIDAD(*problema*) **returns**
solución **o** **fallo**

nodo ← un nodo con ESTADO = *problema*.ESTADO-INICIAL,
COSTE-CAMINO = 0

if *problema*.TEST-OBJETIVO(*nodo*.ESTADO) **then return**
SOLUCION(*nodo*)

frontera ← una **LIFO** con *nodo* como único elemento

explorado ← un conjunto vacío

loop do

if VACIO?(*frontera*) **then return fallo**

nodo ← POP(*frontera*) /*Elige el nodo menos profundo de la
frontera */

add *nodo*.ESTADO a *explorado*

for each *acción* **in** *problema*.ACCIONES(*nodo*.ESTADO) **do**

hijo ← NODO-HIJO(*problema*, *nodo*, *accion*)

if *hijo*.ESTADO no está en *explorado* o *frontera* **then**

if *problema*.TEST-OBJETIVO(*hijo*.ESTADO)

Implementación **Recursiva** con límite. (**recursive** depth limited search **dls**)

```
function BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema, límite)  
    returns solución o fallo  
  
/corte  
    return BPL-RECURSIVA (MAKE-NODE(problema.ESTADO-INICIAL),  
                           problema, límite)  
  
function BPL-RECURSIVA (nodo, problema, límite) returns solución, o  
fallo/corte  
    if problema.TEST-OBJETIVO(nodo.ESTADO) then return  
SOLUCION(nodo)  
    else if límite = 0 then return corte  
    else  
        alcanzado_límite? ← false  
        for each acción in problema.ACCIONES(nodo.ESTADO) do  
            hijo ← NODO-HIJO(problema, nodo, acción)  
            resultado ← BPL-RECURSIVA (hijo, problema, límite – 1)  
            if resultado = corte then alcanzado_límite? ← true  
            else if resultado ≠ fallo then return resultado  
    if alcanzado_límite? then return corte else return fallo
```

Evaluación búsqueda profundidad (completitud)

- Las **propiedades** de la búsqueda en profundidad depende de si la búsqueda es en grafo o en árbol
 - **La búsqueda en grafo**
 - evita estados repetidos y caminos redundantes
 - Es completa en estados finitos, porque en el peor caso expandirá cada nodo
 - **La búsqueda en árbol**
 - Es **no completa** porque puede dar lugar a **bucles infinitos**
 - Puede ser modificada, sin coste de memoria, para que compruebe no hay estados repetidos en el camino de la raíz al nodo en cursos.
 - Esto no evita caminos redundantes
- En espacios infinitos ambas versiones pueden fallar
- Ambas **no son optimas.**

Complejidad temporal (DFS)

- Búsqueda en grafo, acotada por el tamaño del espacio de estados que puede ser infinito.
- Búsqueda en árbol, primero en profundidad **iterativo**
 - puede generar todos los nodos del árbol: $O(b^m)$
 - donde **m, es la máxima profundidad** de cualquier nodo.
- $O(b^m)$ Puede ser mucho mayor que el tamaño del espacio de estados.
- Fíjate que m puede ser mucho mayor que d. Y puede ser infinita en un árbol no acotado.
- **NO parece** haber ventaja en la búsqueda en profundidad.

¿Porqué contemplar la búsqueda en profundidad?

- La búsqueda en profundidad puede ser interesante por la **complejidad espacial**
 - Para una búsqueda **en grafo no hay ventaja**
 - Para una búsqueda **en árbol necesita almacenar sólo un único camino de la raíz al nodo hoja**, junto con los nodos no expandidos de cada nodo en el camino.

$$O(bm)$$

- Con el mismo ejemplo¹ que utilizamos para la búsqueda en profundidad requiere 156 Kb frente a 10 exabytes para $d=16$

¹ Complejidad temporal y espacial para una búsqueda en anchura de factor de ramificación $b=10$, 1 millón de nodos generados/segundo, 1000 bytes por nodo y asumiendo que los nodos al mismo nivel que el nodo objetivo no tienen sucesores,



¿Porqué contemplar la búsqueda en profundidad?

- El menor coste espacial de la búsqueda en profundidad hace que sea el caballo de batalla de otras técnicas en IA (Satisfacción de restricciones (CSP), programación lógica ...).
- Variantes de la búsqueda en profundidad (**búsqueda con backtracking**), utiliza todavía menos memoria
 - Se genera sólo un sucesor en lugar de todos
 - Cada nodo parcialmente expandido, recuerdo cual generar a continuación $O(m)$
- Además se puede guardar la acción en cada paso, en lugar de cada estado, si el estado es costoso de representar.



Evaluación búsqueda profundidad

■ Completitud

- *Si: En espacios de estados finitos.*
- NO en general: Falla en espacios infinitamente profundos, espacios con bucles.

Evaluación búsqueda profundidad

■ Complejidad temporal $O(b^m)$

Terrible si **m** es mucho mayor que **d**

b -factor de ramificación máximo -> Número máximo de sucesores de un nodo

d - profundidad de la solución de menor coste.

m - máxima profundidad del espacio de estados (puede ser ∞)

Evaluación búsqueda profundidad

- Complejidad espacial $O(bm + 1)$
 - Un sucesor en lugar de todos los sucesores
 - Es lineal

Evaluación búsqueda profundidad

- Completa;
 - NO si el espacio de estados no es finito.
- Complejidad temporal $O(b^m)$
- Complejidad espacial $O(bm + 1)$
- Optimalidad: No

Búsqueda limitada en profundidad

- Búsqueda en profundidad con límite profundidad
 - Los nodos en la profundidad l no tienen sucesores.
 - Se puede utilizar el conocimiento del problema para establecer un límite.
- Resuelve el problema de los caminos infinitos.
- Si $l < d$ es incompleto.
- Si $l > d$ no es óptimo.
- Complejidad Temporal
- Complejidad Espacial

$$O(b^l)$$

$$O(bl)$$



Búsqueda profundidad iterativa

- Intenta **combinar el comportamiento espacial de la búsqueda en profundidad** con la **optimalidad de la búsqueda en anchura**
- El algoritmo consiste en realizar búsquedas en profundidad sucesivas con un nivel de profundidad máximo acotado y creciente en cada iteración
- Así se consigue el comportamiento de la búsqueda en anchura pero sin su coste espacial, ya que la exploración es en profundidad, y además los nodos se regeneran a cada iteración
- Además esto permite evitar los casos en que la búsqueda en profundidad no acaba (existen ramas infinitas)
- En la primera iteración la profundidad máxima será 1 y este valor irá aumentando en sucesivas iteraciones hasta llegar a la solución
- Para garantizar que el algoritmo acaba si no hay solución, se puede definir una cota máxima de profundidad en la exploración

```

function BÚSQUEDA-PROFUNDIZACION-ITERATIVA (problema) returns
solucion,
o fallo
    for profundidad = 0 to  $\infty$  do
        resultado  $\leftarrow$  BÚSQUEDA-PROFUNDIDAD-LIMITADA (problema,
profundidad)
        if resultado  $\neq$  corte then return resultado

```

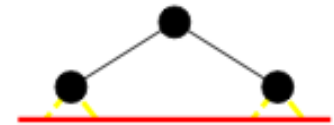
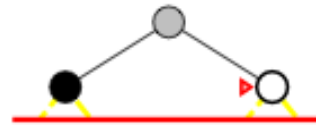
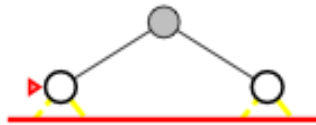
ID-search, example

■ Limit=0



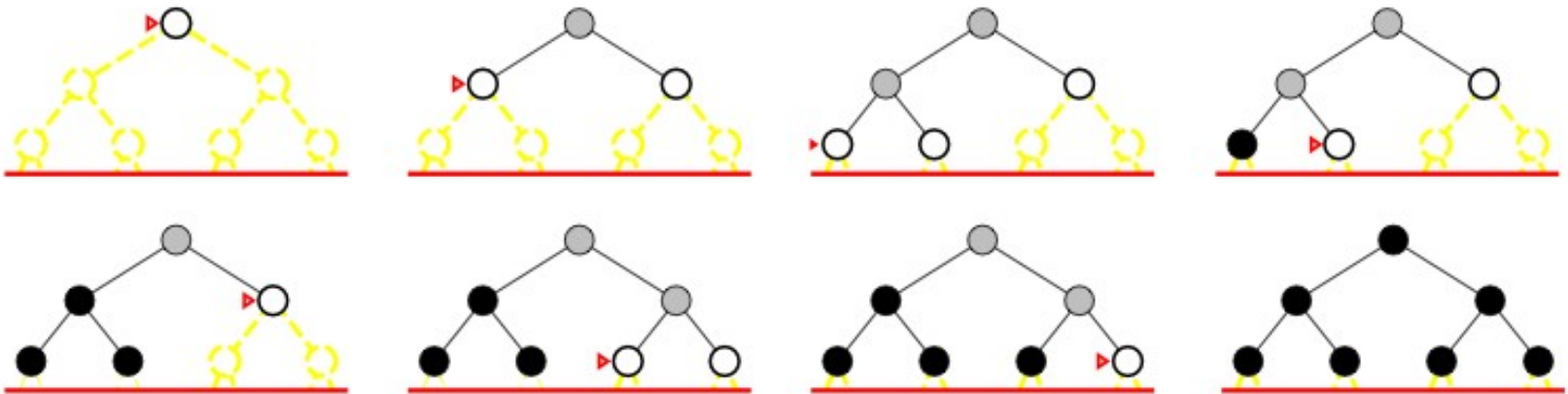
ID-search, example

■ Limit=1



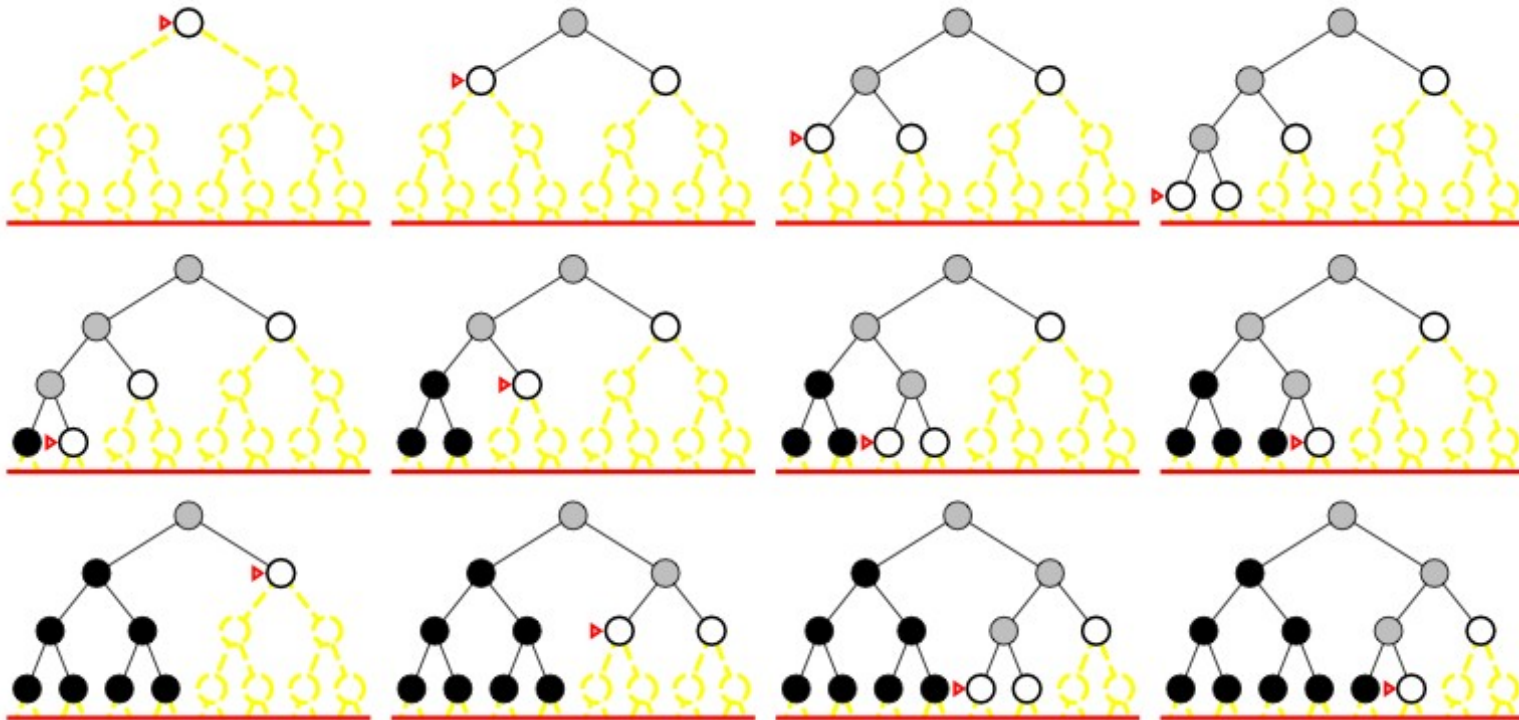
ID-search, example

■ Limit=2



ID-search, example

■ Limit=3



Evaluación búsqueda profundidad iterativa

■ Completitud:

- SI (sin caminos infinitos)

■ Complejidad temporal

- Generación de nodos:

- Nivel d: una vez

$$O(b^d)$$

- Nivel d-1: 2

- Nivel d-2: 3

$$N(IDS) = (d)b + (d-1)b^2 + \dots + (1)b^d$$

- ...

- Nivel 2: d-1

$$N(BFS) = b + b^2 + \dots + b^d + (b^{d+1} - b)$$

- Nivel 1: d

Num. Comparaciones para $b=10$ y $d=5$

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

ID search, evaluation

■ Completitud:

- SI (sin caminos infinitos)

■ Complejidad Temporal $O(b^d)$

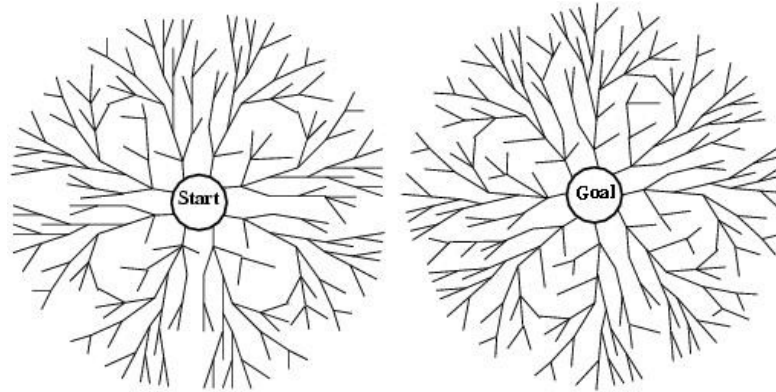
■ Complejidad Espacial $O(bd)$

- Igual que la búsqueda en profundidad

■ Optimalidad:

- Si igual que la búsqueda en anchura.
- Puede ser extendida con la misma idea que la búsqueda uniforme.

Búsqueda bidireccional



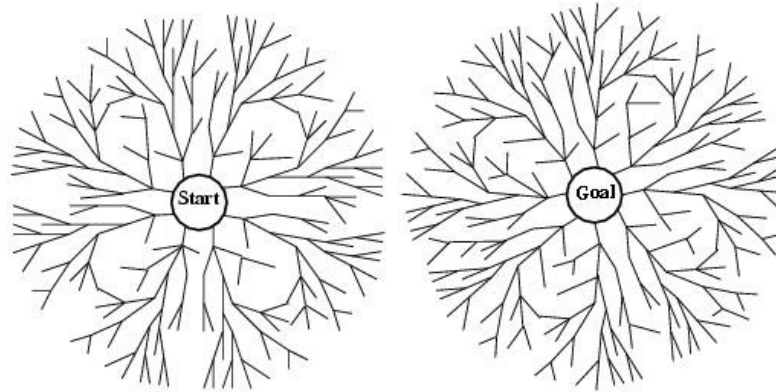
- Búsquedas simultaneas desde el objetivo y el inicio.

- Motivación:

$$b^{d/2} + b^{d/2} \neq b^d$$

- Comprueba si el nodo pertenece a la otra frontera antes de expandir.
- La complejidad espacial es el punto débil
- Completo y optimo si ambas búsquedas son en anchura.

¿Cómo buscar hacia atrás?



- El predecesor de cada nodo debe ser computable fácilmente.
 - Cuando las acciones son reversibles

Resumen de algoritmos

| Criterio | Breadth-First | Uniform-cost | Depth-First | Depth-limited | Iterative deepening | Bidirectional search |
|-------------|---------------|--------------|-------------|-----------------------|---------------------|----------------------|
| Compleitud | YES* | YES* | NO | YES, if $l \geq d$ | YES | YES* |
| Temporal | b^{d+1} | $b^{C*/e}$ | b^m | b^l | b^d | $b^{d/2}$ |
| Espacial | b^{d+1} | $b^{C*/e}$ | bm | bl | bd | $b^{d/2}$ |
| Optimalidad | YES* | YES* | NO | NO | YES | YES |



Universidad
Zaragoza



Inteligencia Artificial

(30223) Grado en Ingeniería Informática

lección 3. Resolución de problemas y
búsqueda. Secciones 3.1 a 3.4 (AIMA)