

# Búsqueda

Inteligencia Artificial e Ingeniería del Conocimiento

---

Constantino Antonio García Martínez

Universidad San Pablo Ceu

## Bibliografía

- Russell, Stuart J., and Peter Norvig. Artificial intelligence: a modern approach. Pearson, 2016.

## Motivación

---

## Motivación

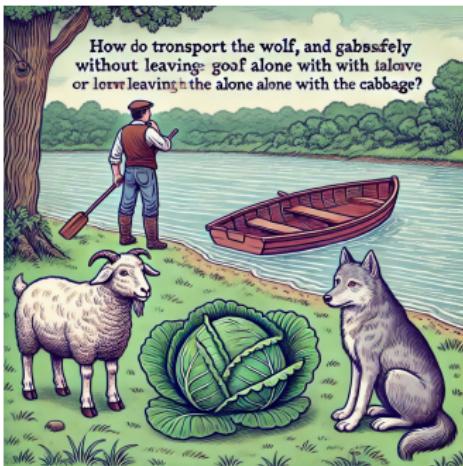
---

### Problema motivante

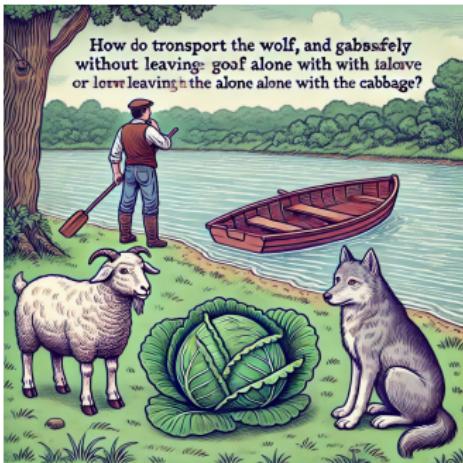
# Motivación: el problema del granjero

## Example: El problema del granjero

Un granjero quiere llevar su repollo, su cabra y su lobo al otro lado de un río. Tiene un bote que solo puede llevar a dos pasajeros. No puede dejar solos al repollo y la cabra, ni a la cabra y el lobo. ¿Cuántos cruces del río necesita hacer?

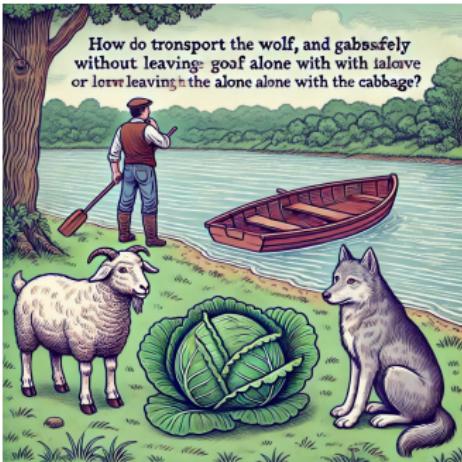


# El problema del granjero



Enfoque: construir un árbol de búsqueda ("¿qué pasaría si...?")

# El problema del granjero



Enfoque: construir un árbol de búsqueda ("¿qué pasaría si...?") En primer lugar, debemos **modelar el problema**. Por ejemplo, ¿qué acciones son posibles?:

F>

FC>

FG>

FW>

F<

FC<

FG<

FW<

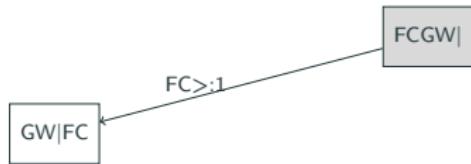
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**

FCGW|

# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



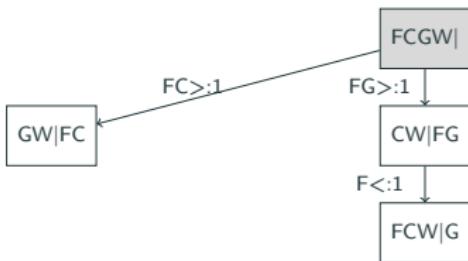
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



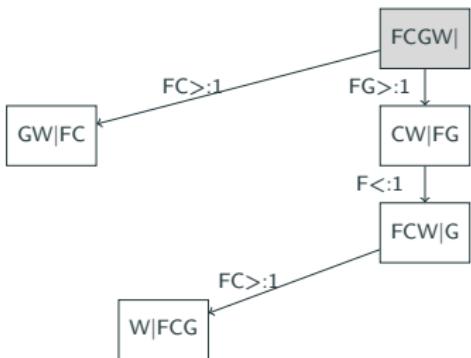
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



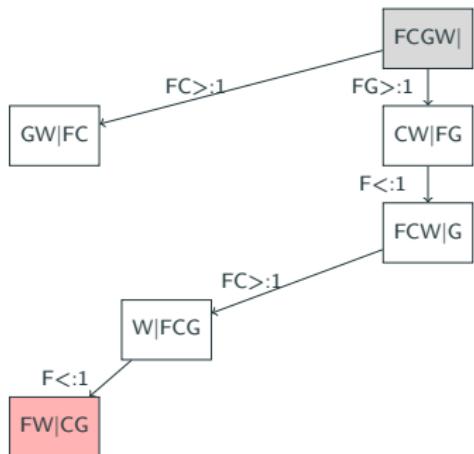
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



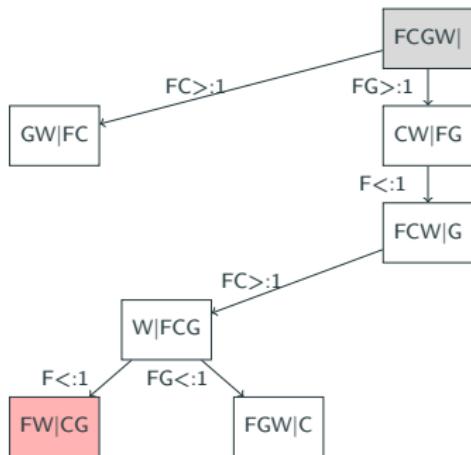
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



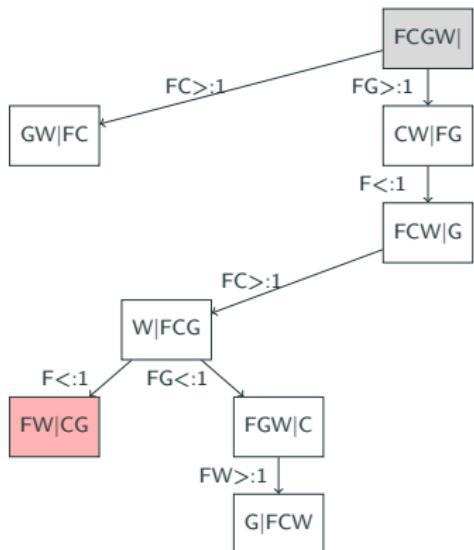
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



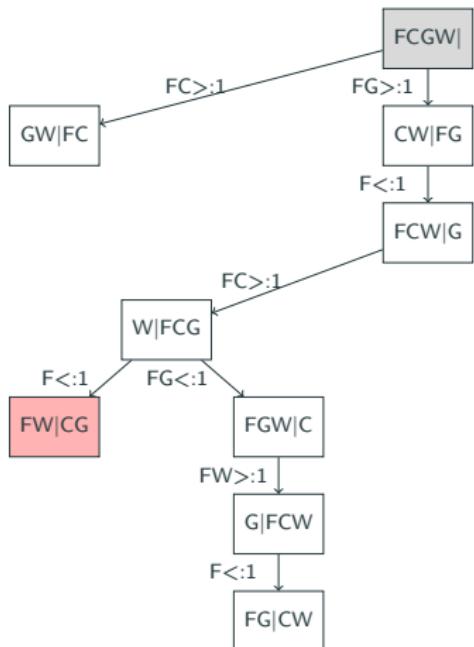
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



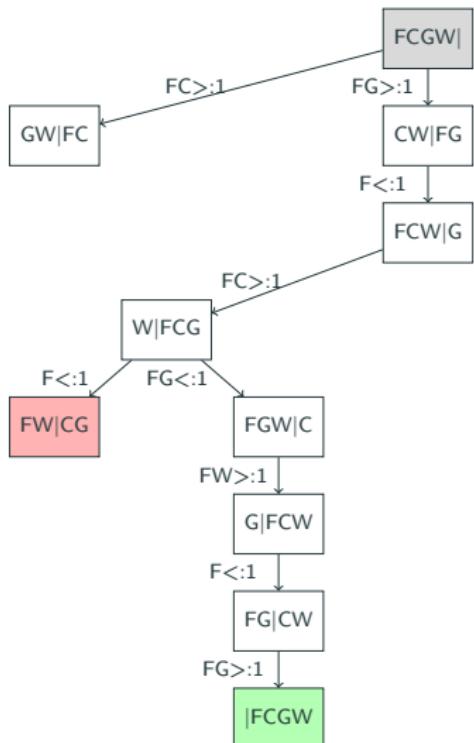
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



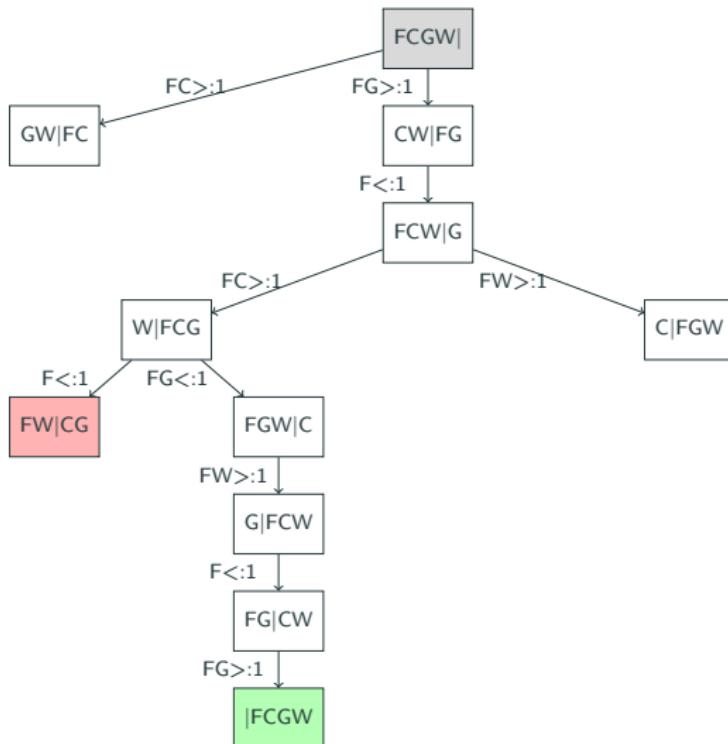
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



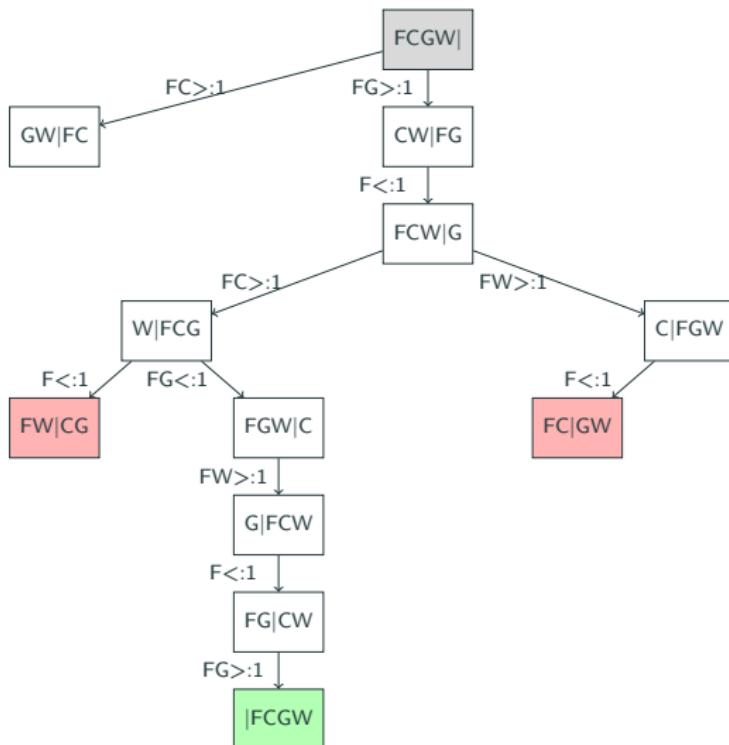
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



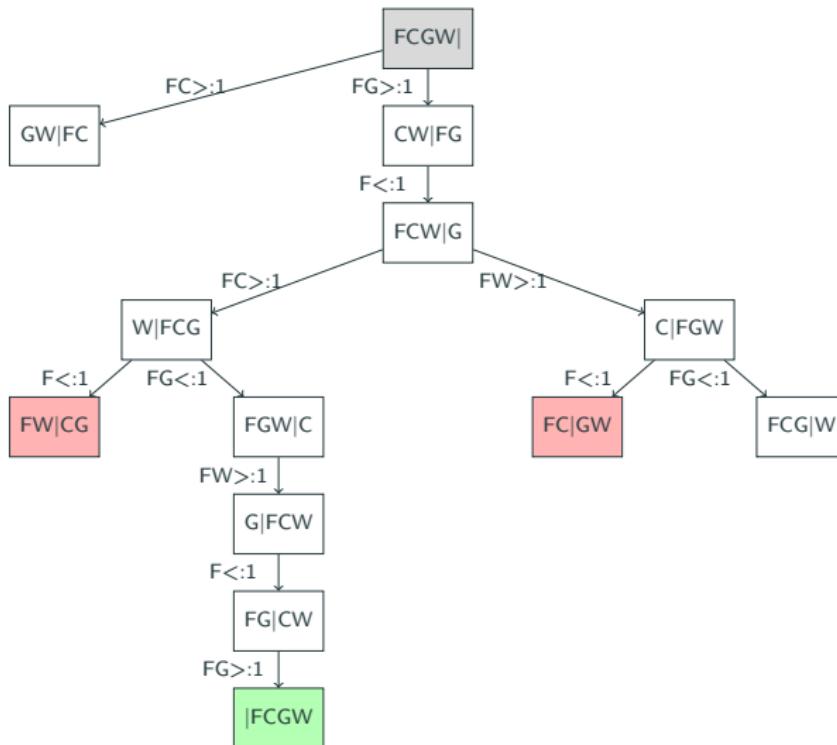
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



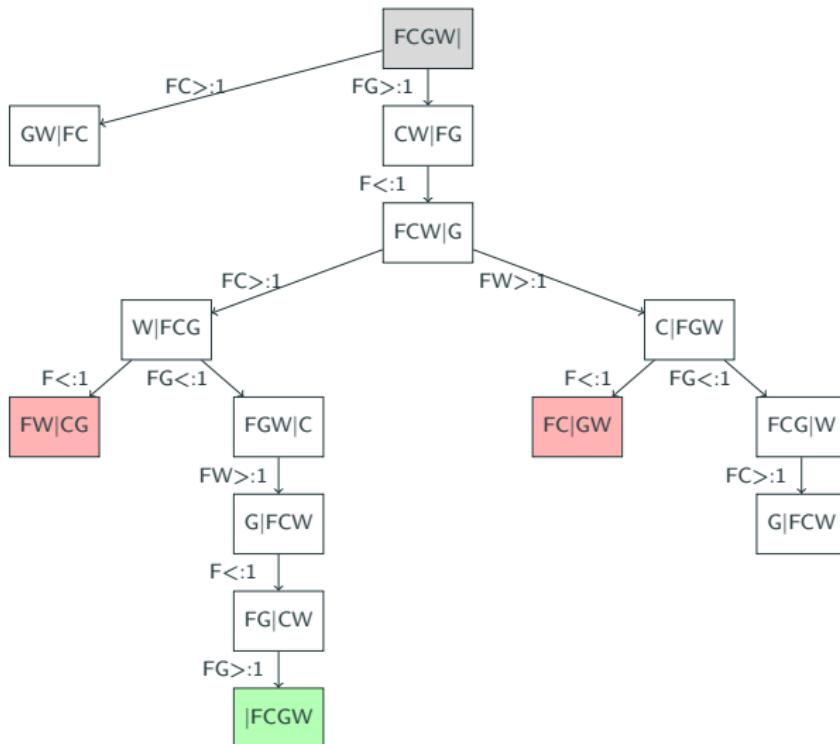
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



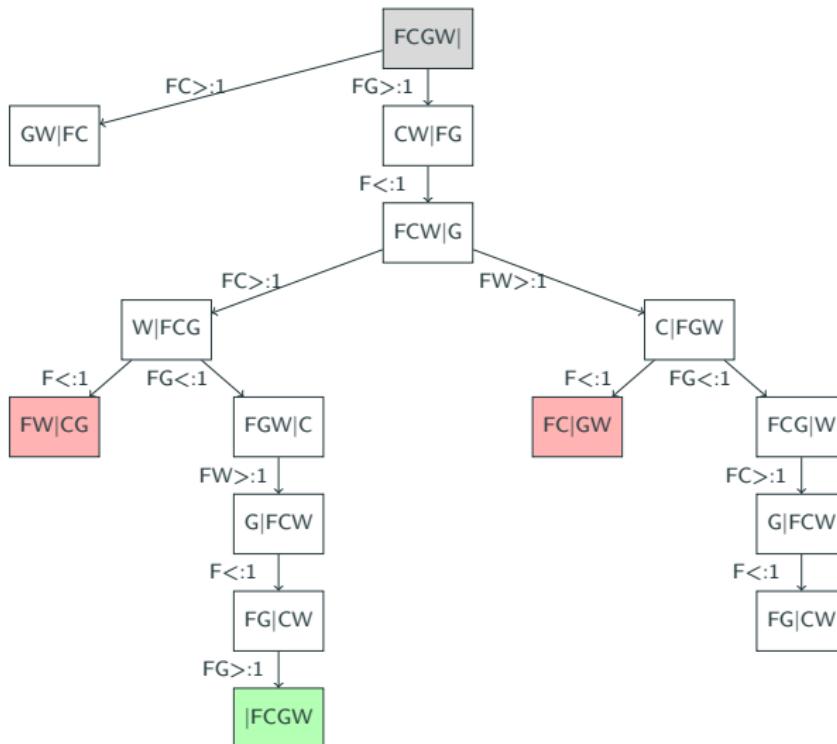
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



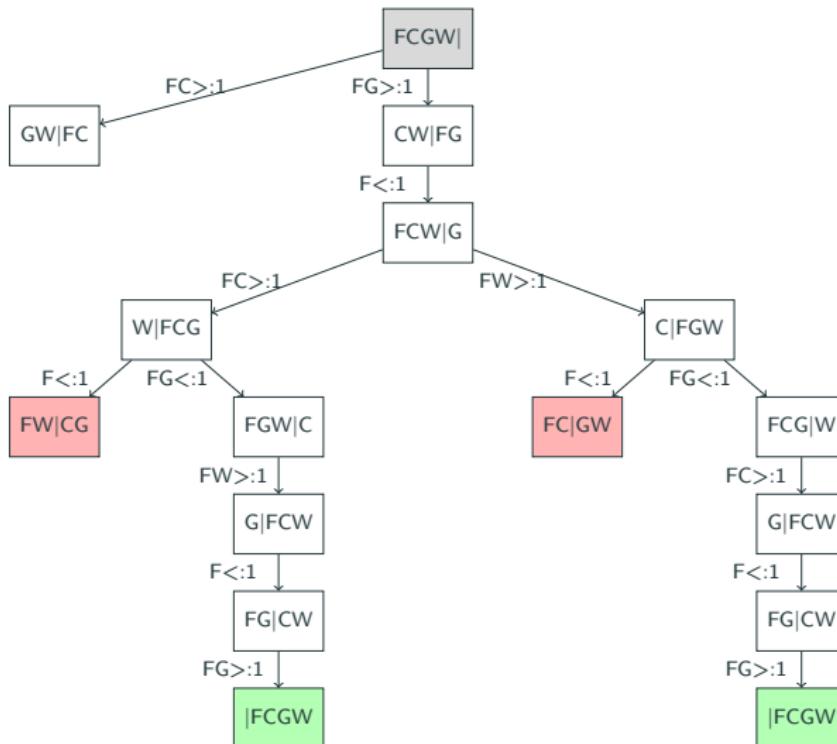
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



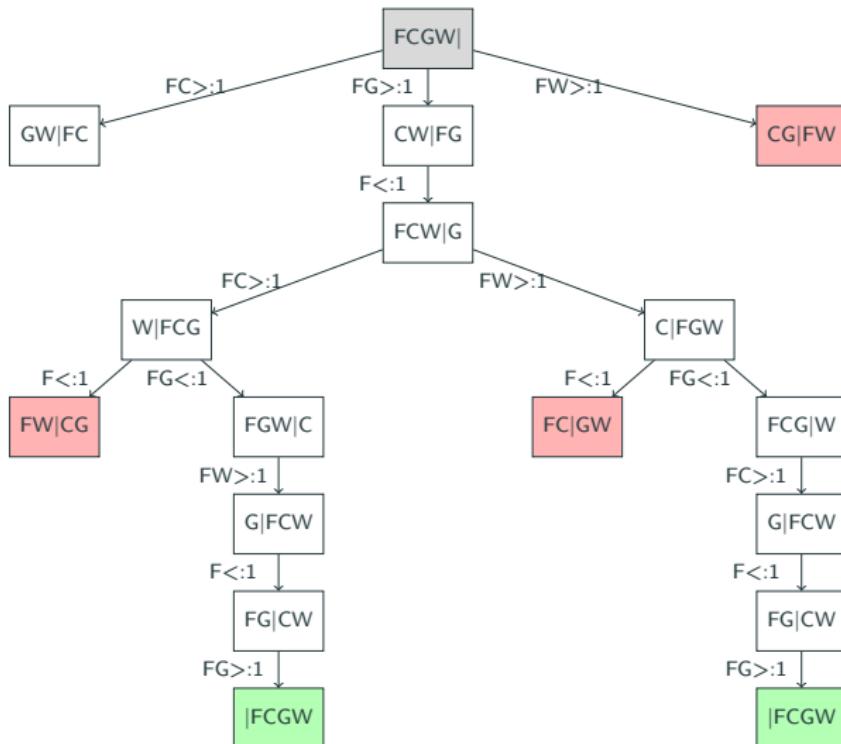
# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**



# El problema del granjero

Una vez se ha modelado el problema, solo es necesaria la **inferencia**

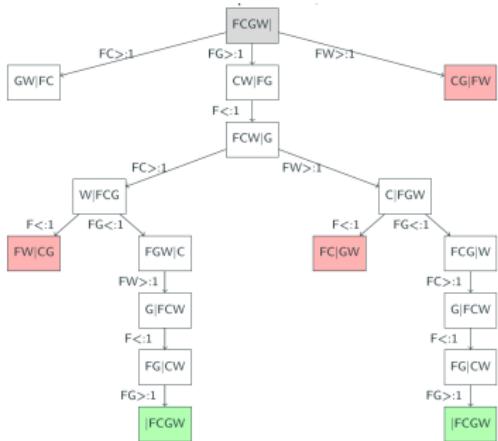


## Modelado Vs. Inferencia

Modelado Vs. Inferencia

# Problema de búsqueda

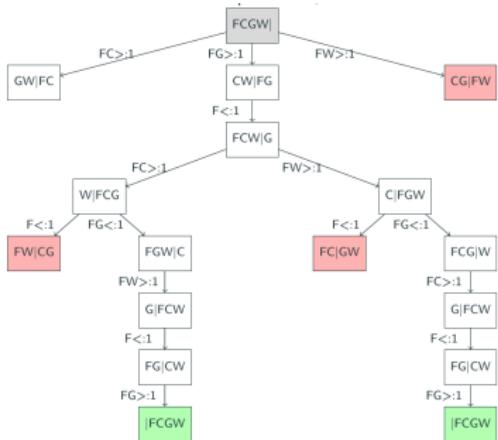
## Modelado de un problema



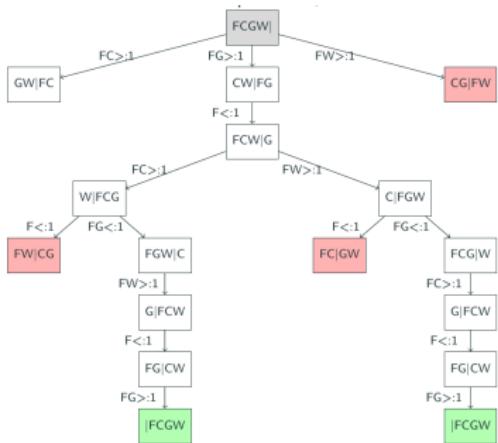
# Problema de búsqueda

## Modelado de un problema

- Estado inicial



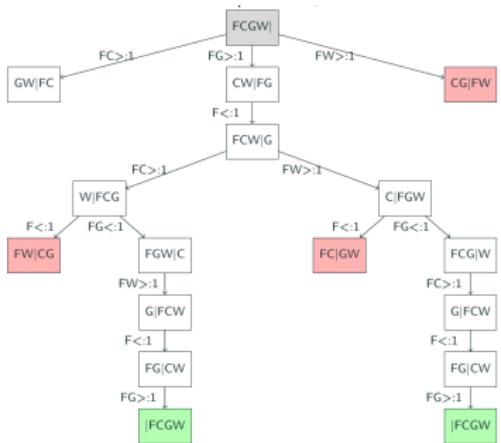
# Problema de búsqueda



## Modelado de un problema

- Estado inicial
- actions(state): acciones

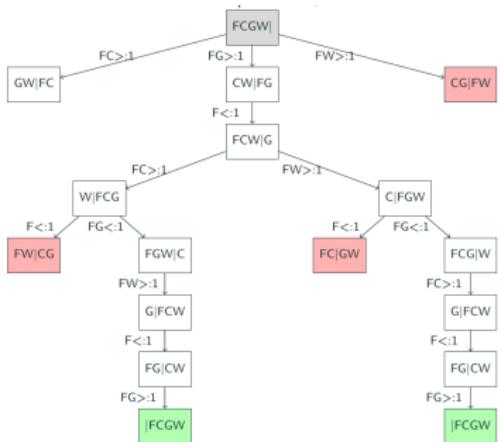
# Problema de búsqueda



## Modelado de un problema

- Estado inicial
- `actions(state)`: acciones
- `result(state, action)`: estado resultante de la acción (a veces se le llama `succ(state, action)`).

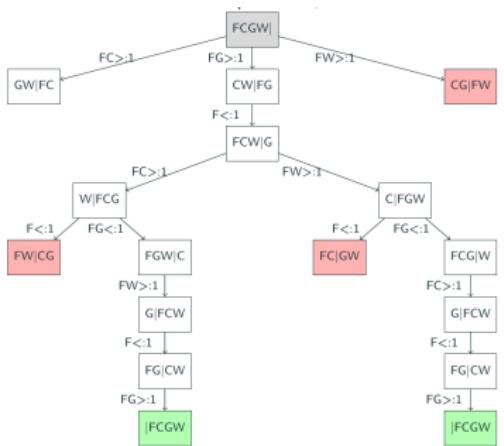
# Problema de búsqueda



## Modelado de un problema

- Estado inicial
- `actions(state)`: acciones
- `result(state, action)`: estado resultante de la acción (a veces se le llama `succ(state, action)`).
- `path_cost(c, state1, action, state2)`: coste total para llegar al `state2`.

# Problema de búsqueda



## Modelado de un problema

- Estado inicial
- actions(state): acciones
- result(state, action): estado resultante de la acción (a veces se le llama succ(state, action)).
- path\_cost(c, state1, action, state2): coste total para llegar al state2.
- goal\_test(state): ¿es un estado final?

## Motivación

---

## Aplicaciones reales

# Aplicaciones del Mundo Real de Algoritmos de Búsqueda

- **Algoritmos de búsqueda de rutas:**
  - Sistemas de navegación en automóviles y redes de video.
  - Planificación de operaciones militares y de viajes aéreos.
- **Problemas de recorrido:**
  - Ejemplo: Problema del Viajante (TSP).
  - Aplicaciones: Optimización de rutas para autobuses escolares, planificación de viajes, etc.
  - Beneficios: Ahorros significativos, reducción de tráfico y contaminación.
- **Diseño y enrutamiento de VLSI:**
  - Posicionar millones de componentes en un chip.
  - Minimizar área, retrasos de circuito y capacitancias no deseadas.
- **Navegación de robots:**
  - Espacios de búsqueda multidimensionales para controlar brazos y piernas.
  - Desafíos: errores en sensores, controladores, y entornos dinámicos.
- **Secuenciación de ensamblaje automático:**
  - Ejemplo: Ensamblaje de motores eléctricos.
  - Minimizar trabajo manual, optimizar secuencias.
  - Problemas de diseño de proteínas para tratamientos médicos.

## Primeros pasos: Naive Backtracking

---

## Ejemplo de transporte

### Code Example: Modelado del problema del transporte

- Una calle con bloques numerados del 1 al  $n$ .
- Caminar de  $s$  a  $s + 1$  toma 1 minuto.
- Tomar un tranvía mágico de  $s$  a  $2s$  toma 2 minutos.
- ¿Cómo viajar del 1 al  $n$  en el menor tiempo posible?

### Code Example: Inferencia en el problema del transporte: Naive Backtracking

### Code Exercise: Inferencia y tamaño del problema (transportation\_main.py)

1. Ejecuta el código con  $n=1000$ .
2. Incluye ...

```
import sys  
sys.setrecursionlimit(10**6)
```

... y ejecuta el código con  $n=1000$ .

## Búsqueda con Naive Backtracking

Evaluamos los algoritmos de cuatro maneras:

- **Completitud:** ¿El algoritmo garantiza encontrar una solución cuando existe una, y reportar correctamente el fallo cuando no existe?
- **Optimalidad de costo:** ¿Encuentra una solución con el menor costo de camino entre todas las soluciones?
- **Complejidad temporal:** ¿Cuánto tiempo tarda en encontrar una solución? Esto puede medirse en segundos, o de manera más abstracta por el número de estados y acciones consideradas.
- **Complejidad espacial:** ¿Cuánta memoria se necesita para realizar la búsqueda?

## Evaluación de la búsqueda con Naive Backtracking

La evaluación se hace en base al árbol de búsqueda construido. Si hay  $b$  acciones por estado<sup>1</sup>, y la profundidad máxima es  $m$ :

- ¿Completo? Sí.
- ¿Óptimo? Sí.
- Memoria:  $O(m)$  (pequeño)
- Tiempo:  $O(b^m)$  (enorme).

---

<sup>1</sup>  $b$  también es conocido como factor de ramificación

## Búsqueda en árbol vs búsqueda en grafo

**Code Exercise: El problema del granjero: modelado**

Modela el problema del granjero (farmer\_main.py)

**Code Exercise: El problema del granjero: inferencia**

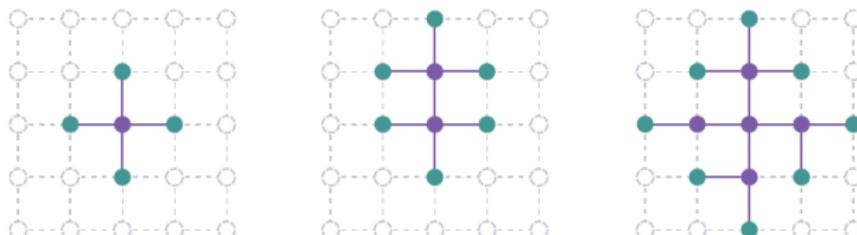
Ejecuta el problema del granjero (farmer\_main.py). Ahora cambia el algoritmo por backtracking. ¿Qué ocurre? ¿Por qué?

## Estrategias de Búsqueda No Informada

---

## Estrategia de alto nivel

Para mejores algoritmos de búsqueda, selecciona astutamente qué nodos expandir y sal cuando encuentres una solución.



- Explorado: estados para los que hemos encontrado el camino óptimo
- Frontera: estados que hemos visto, aún estamos averiguando cómo llegar allí de manera económica
- No explorado: estados que no hemos visto

Todos los algoritmos siguientes pueden verse como instancias de **Best-First Search**, en la que elegimos un nodo,  $n$ , con el valor mínimo de alguna función de evaluación,  $f(n)$ . Sin embargo, suelen hacerse implementaciones especializadas no basadas en Best-First Search para optimizar el código y estructuras de datos.

## Estrategia de alto nivel

Recordando *el problema del granjero*: los algoritmos que no pueden recordar el pasado están condenados a repetirlo.

- **Búsqueda tipo árbol:** no verifica los estados alcanzados.
- **Búsqueda en grafo:** verifica los estados alcanzados.

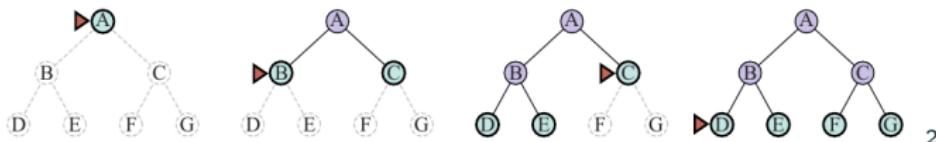
Los algoritmos de búsqueda en grafo funcionan con bucles, pero tienen un peor rendimiento de memoria.

## Búsqueda en Anchura (Breadth-First Search)

**Suposición: costes de acción constantes**

Supongamos que los costes de acción  $\text{Cost}(s, a) = c$  para algún  $c \geq 0$ .

**Idea:** explorar todos los nodos en orden de profundidad creciente.



Leyenda:  $b$  acciones por estado, la primera solución tiene  $d$  acciones.

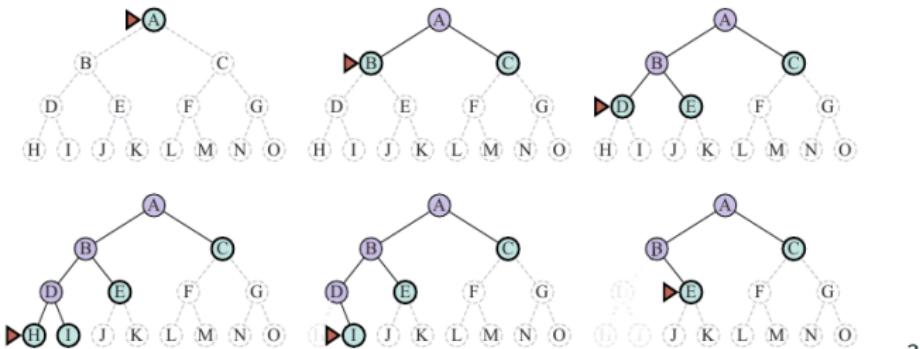
- ¿Completo? Sí.
- ¿Óptimo? Sí, aunque solo si se cumple la suposición de costes ctes.
- Espacio: ahora  $O(b^d)$  (¡mucho peor!)
- Tiempo:  $O(b^d)$  (mejor, depende de  $d$ , no de  $m$ )

---

<sup>2</sup>Imagen tomada de AIMA

## Búsqueda en Profundidad (Depth-First Search, DFS)

Idea: Búsqueda con Backtracking + parar cuando se encuentra el primer estado final



3

Si hay  $b$  acciones por estado, la profundidad máxima es  $m$ .

- ¿Completo? No.
- ¿Óptimo? No.
- Espacio:  $O(bm)$  en el peor caso.
- Tiempo:  $O(b^m)$  en el peor caso, pero podría ser mucho mejor si las soluciones son fáciles de encontrar.

<sup>3</sup>Imagen tomada de AIMA

## DFS con profundización iterativa (DFS with Iterative Deepening, DFS-ID)

### Suposición: costes de acción constantes

Supongamos que los costes de acción  $\text{Cost}(s, a) = c$  para algún  $c \geq 0$ .

Idea:

- Modificar DFS para parar a una profundidad máxima.
- Llamar a DFS para profundidades máximas  $1, 2, \dots$ . (DFS en  $d$  pregunta: ¿hay una solución con  $d$  acciones?)

Leyenda:  $b$  acciones por estado,  $d$  la profundidad de la primera solución.

- ¿Completo? Sí.
- ¿Óptimo? Sí, si se cumple la suposición de costes constantes.
- Espacio:  $O(bd)$ .
- Tiempo:  $O(b^d)$ .

# Implementación de Algoritmos de búsqueda

## Code Exercise: Implementación de algoritmos

Implementa los siguientes algoritmos sin basarse en Best-First-Search.

1. BFS (tanto en árboles como en grafos).
2. DFS (tanto en árboles como en grafos).
3. DFS-ID (tanto en árboles como en grafos).

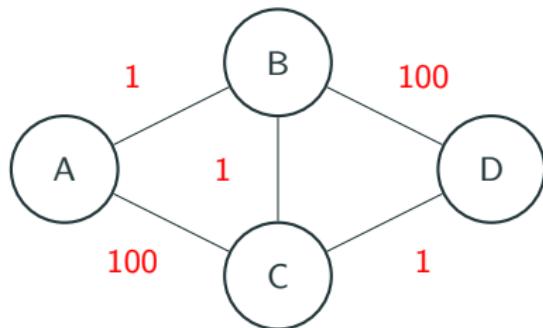
## Estrategias de Búsqueda No Informada

---

**Costes no constantes**

## Búsqueda de costo uniforme (Uniform cost search, UCS)

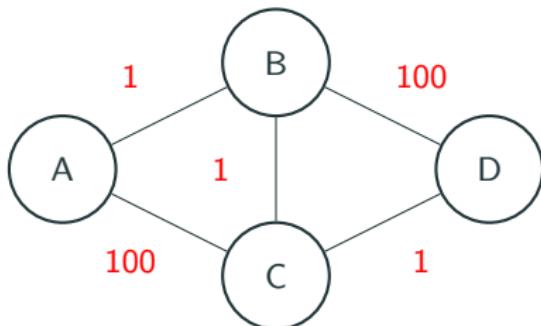
¿Qué ocurre si los costes no son constantes?



Estado inicial: A, estado final: D

## Búsqueda de costo uniforme (Uniform cost search, UCS)

¿Qué ocurre si los costes no son constantes?



Estado inicial: A, estado final: D

Idea: ordenación de estados

UCS enumera los estados en orden de costo pasado creciente.

**Suposición: no negatividad**

Todos los costes de acción son no negativos:  $\text{Cost}(s, a) \geq 0$ .

Camino de costo mínimo:

$A \rightarrow B \rightarrow C \rightarrow D$  con costo 3

# Búsqueda de costo uniforme (UCS)

Code Example: UCS (Dijkstra, 1956) basado en Best-First Search

Code Exercise: Program UCS y resuelve el problema del grafo (graph\_main.py)

Theorem (corrección de UCS)

Cuando un estado  $s$  se saca de la frontera y se mueve a explorado, su prioridad es el costo mínimo a  $s$ .

- ¿Completo? Sí.
- ¿Óptimo? Sí.
- Espacio:  $O(b^{1+\lceil C^*/\epsilon \rceil}) \approx O(b^d)$ . Leyenda<sup>4</sup>.
- Tiempo:  $O(b^{1+\lceil C^*/\epsilon \rceil}) \approx O(b^d)$ .

---

<sup>4</sup>  $C^*$  es el coste óptimo y  $\epsilon$  una cota inferior del coste de cada acción

# Resumen de Algoritmos de Búsqueda No Informada

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>1</sup>	Yes <sup>1,2</sup>	No	No	Yes <sup>1</sup>	Yes <sup>1,4</sup>
Optimal cost?	Yes <sup>3</sup>	Yes	No	No	Yes <sup>3</sup>	Yes <sup>3,4</sup>
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

**Figure 3.15** Evaluation of search algorithms.  $b$  is the branching factor;  $m$  is the maximum depth of the search tree;  $d$  is the depth of the shallowest solution, or is  $m$  when there is no solution;  $\ell$  is the depth limit. Superscript caveats are as follows: <sup>1</sup> complete if  $b$  is finite, and the state space either has a solution or is finite; <sup>2</sup> complete if all action costs are  $\geq \epsilon > 0$ ; <sup>3</sup> cost-optimal if action costs are all identical; <sup>4</sup> if both directions are breadth-first or uniform-cost.

5

- Siempre tiempo exponencial
- Evitar espacio exponencial con DFS-ID

---

<sup>5</sup>Imagen tomada de AIMA

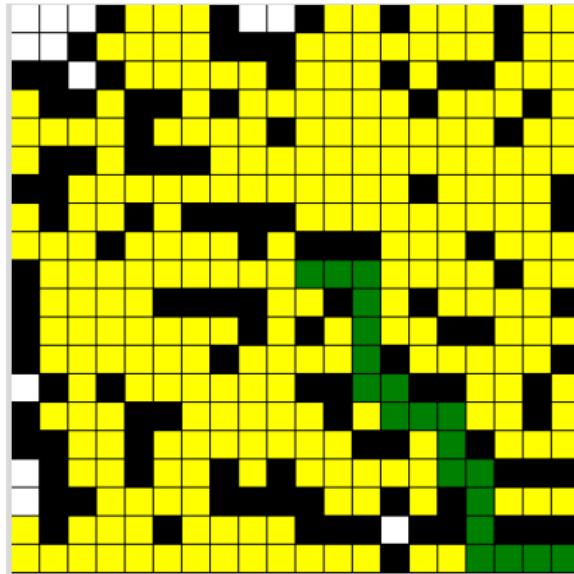
## Algoritmos de Búsqueda Informada: A\*

---

# ¿Puede mejorar la búsqueda de costo uniforme?

## Demo: UCS Maze

UCS malgasta recursos:



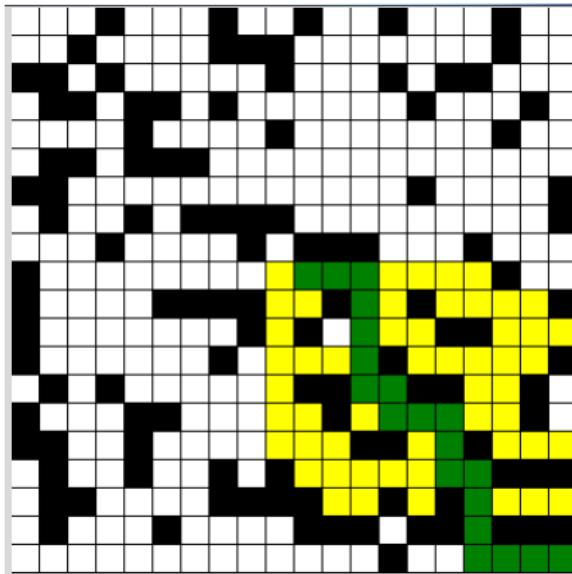
**Problema:** UCS ordena los estados por costo desde  $s_{\text{inicio}}$  hasta  $s$

**Objetivo:** tener en cuenta el costo desde  $s$  hasta  $s_{\text{final}}$

# ¿Puede mejorar la búsqueda de costo uniforme?

## Demo: UCS Maze

En realidad, nos gustaría:

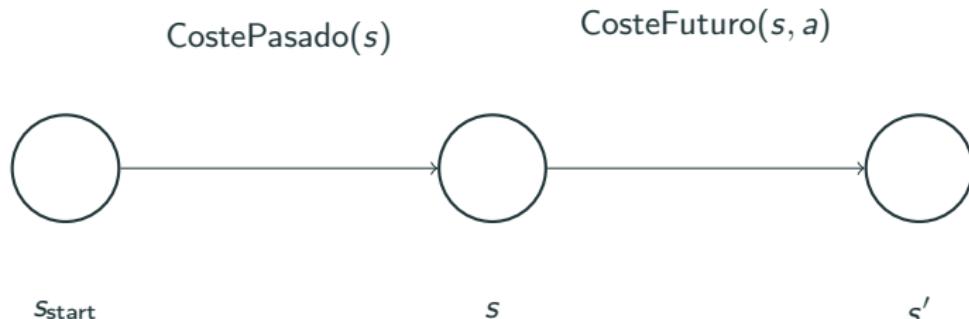


**Problema:** UCS ordena los estados por costo desde  $s_{\text{inicio}}$  hasta  $s$

**Objetivo:** tener en cuenta el costo desde  $s$  hasta  $s_{\text{final}}$

## Exploración de estados

**UCS:** explora los estados en orden de CostoPasado( $s$ )



**Ideal:** explora en orden de CostePasado( $s$ ) + CosteFuturo( $s$ )

**A\*** (Hart/Nilsson/Raphael, 1968): explora en orden de CostePasado( $s$ ) +  $h(s)$

**Definition (Función heurística)**

Una heurística  $h(s)$  es cualquier estimación de CosteFuturo( $s$ ).

## $A^*$ algorithm

Demo: UCS Maze with  $A^*$

Code Exercise: Programa  $A^*$  usando Best-First search

Code Exercise: Resuelve el Maze con USC y  $A^*$  (`maze_main.py`)

## Admisibilidad

### Definition (Admisibilidad)

Una heurística  $h(s)$  es admisible si  $h(s) \leq \text{CostoFuturo}(s)$

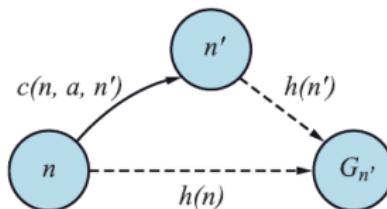
Intuición: las heurísticas admisibles son optimistas.

### Lemma (corrección)

*Si  $h$  es admisible,  $A^*$  devuelve la ruta de costo mínimo.*

# Heurísticas consistentes

Una propiedad más fuerte es la consistencia...



**Figure 3.19** Triangle inequality: If the heuristic  $h$  is **consistent**, then the single number  $h(n)$  will be less than the sum of the cost  $c(n, a, n')$  of the action from  $n$  to  $n'$  plus the heuristic estimate  $h(n')$ .

6

## Definition (Consistencia)

Una heurística  $h$  es consistente si:

$$h(s) \leq \text{Costo}(s, a) + h(\text{Succ}(s, a))$$

$$h(s_{\text{final}}) = 0.$$

---

<sup>6</sup>Imagen tomada de AIMA

# Consistencia y Admisibilidad

**Theorem (Consistencia implica admisibilidad)**

*Si una heurística  $h(s)$  es consistente, entonces  $h(s)$  es admisible.*

**Lemma (corrección)**

*Si  $h$  es consistente,  $A^*$  devuelve la ruta de costo mínimo.*

¿Por qué preocuparse por la admisibilidad?

- ¡ $h$  consistente no reexpande nodos!

## Propiedades de A\*

- ¿Completa? Sí.
- ¿Óptima? Sí, si la heurística es admisible.
- Espacio:  $O(b^d)$  en el peor caso (generalmente, mucho más bajo).
- Tiempo:  $O(b^d)$  en el peor caso (generalmente, mucho más bajo).

## Algoritmos de Búsqueda Informada: A\*

---

### Funciones heurísticas

## ¿Cómo obtener buenas heurísticas? Relajación

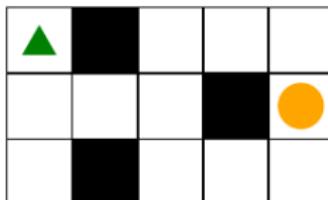
Intuición: idealmente, usar  $h(s) = \text{CostoFuturo}(s)$ , pero eso es tan difícil como resolver el problema original.

### Idea: Relajación

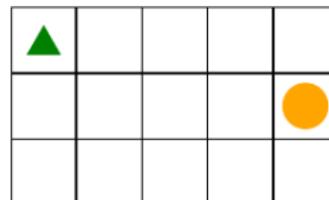
Dado que las heurística optimistas son admisibles y las restricciones complican la vida, las ¡eliminamos!

## Example: Relajación

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

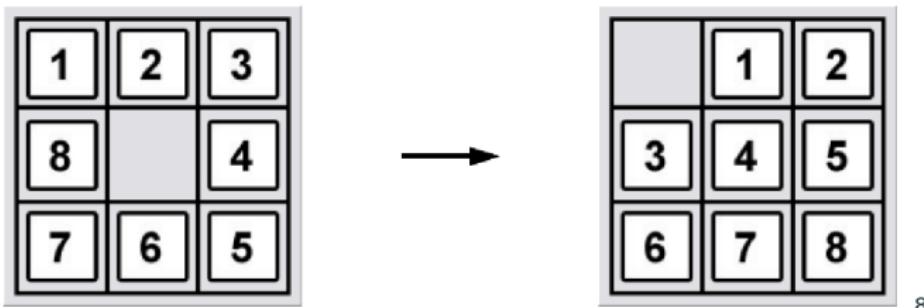
$$\text{e.g., } h((1, 1)) = 5$$

7

<sup>7</sup>Imagen tomada de Standford CS221

## Subproblemas independientes

Example: 8 puzzle



**Problema original:** las piezas no pueden superponerse (restricción)

**Problema relajado:** las piezas pueden superponerse (sin restricción)

**Solución relajada:** 8 problemas independientes, cada uno en forma cerrada.

Podemos combinarlas con max.

---

<sup>8</sup>Imagen tomada de AIMA

**Para saber más**

---

## Para saber más...

- **Beam Search:** Técnica de búsqueda heurística que limita el número de nodos expandidos en cada nivel.

*Interesante porque:* Ofrece un equilibrio entre exploración y eficiencia, y es útil en problemas de gran escala.

- **Pattern Databases:** Bases de datos precomputadas para almacenar soluciones óptimas de subproblemas.

*Interesante porque:* Reducen el tiempo de búsqueda en problemas de espacios de estado complejos.

- **Otros algoritmos:** Bidirectional Search, IDA\*, etc.