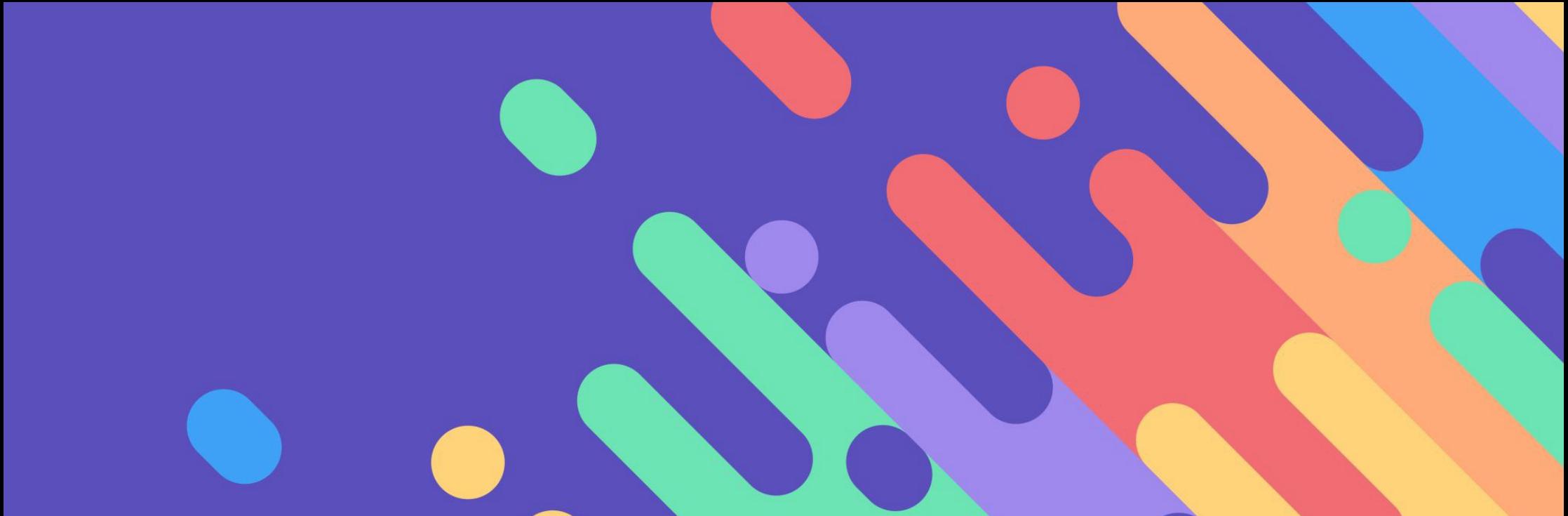


AGENTES RESOLUCIÓN DE PROBLEMAS



Inteligencia Artificial
CEIA - FIUBA
Dr. Ing. Facundo Adrián
Lucianna



LO QUE VIMOS LA CLASE ANTERIOR...

INTELIGENCIA ARTIFICIAL

- Según Stuart Russell y Peter Norvig:
 - A veces se define en función de:
 - La fidelidad del desempeño humano (u otro animal)
 - Hacer “lo correcto” (racionalidad)
 - También se considera una propiedad:
 - De los procesos de pensamiento y razonamiento internos.
 - Del comportamiento, es decir una característica externa.

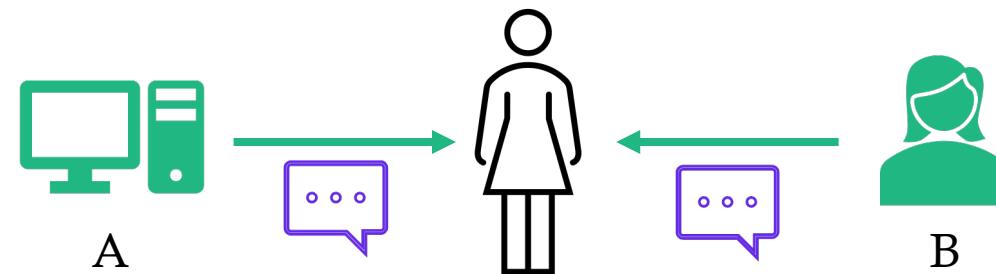
INTELIGENCIA ARTIFICIAL

Actuando humanamente

El test de Turing

En el test, un interrogador humano interactúa con dos participantes, A y B, intercambiando mensajes escritos.

Si el interrogador no es capaz de determinar qué participante, A o B, es una computadora y cuál es un ser humano, se considera que la computadora ha superado la prueba.



INTELIGENCIA ARTIFICIAL

Pensando racionalmente

Un agente es algo que actúa. Un agente se espera que opere autónomamente, perciba el ambiente, persista sobre un tiempo prolongado, se adapte y cree y cumpla objetivos.

Un agente racional es aquel que llega al mejor escenario, o si hay incertidumbre, al mejor escenario esperado.

IA se ha enfocado en el estudio y construcción de agentes que hacen lo correcto. Que cuenta por hacer lo correcto es el objetivo que le proveemos al agente. Esto se llama el **modelo estándar**.

Este modelo no solo ha sido predominante en IA, sino además en *teoría del control*, en *estadística*, y *economía*.

INTELIGENCIA ARTIFICIAL

Máquinas beneficiosas

El **modelo estándar** asume que se va a un objetivo específico a resolver... algo que, en la vida real, es mucho más difícil especificar el objetivo completamente y correctamente.

INTELIGENCIA ARTIFICIAL

Máquinas beneficiosas

El **modelo estándar** asume que se va a un objetivo específico a resolver... algo que, en la vida real, es mucho más difícil especificar el objetivo completamente y correctamente.

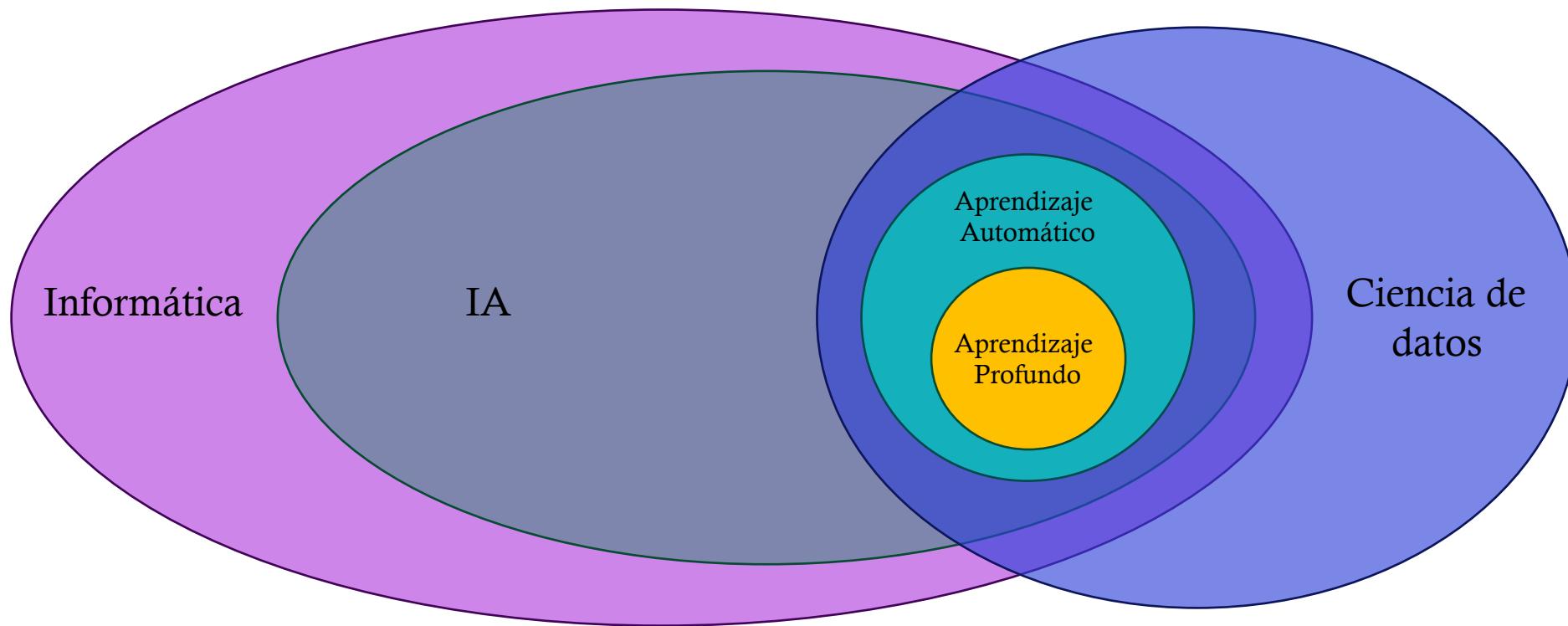
El balance de lograr un acuerdo entre nuestras preferencias y el objetivo que tiene la maquina se llama problema de alineaciones de valores.

En la vida real, esto no es posible. Por lo que el modelo estándar **no** es apropiado. Se necesita una nueva formulación.

Cuando una máquina sabe que no conoce el objetivo completo, tiene un incentivo para actuar con cautela, pedir permiso, aprender más sobre nuestras preferencias a través de la observación y ceder al control humano.

INTELIGENCIA ARTIFICIAL

Campos conectados



PYTHON

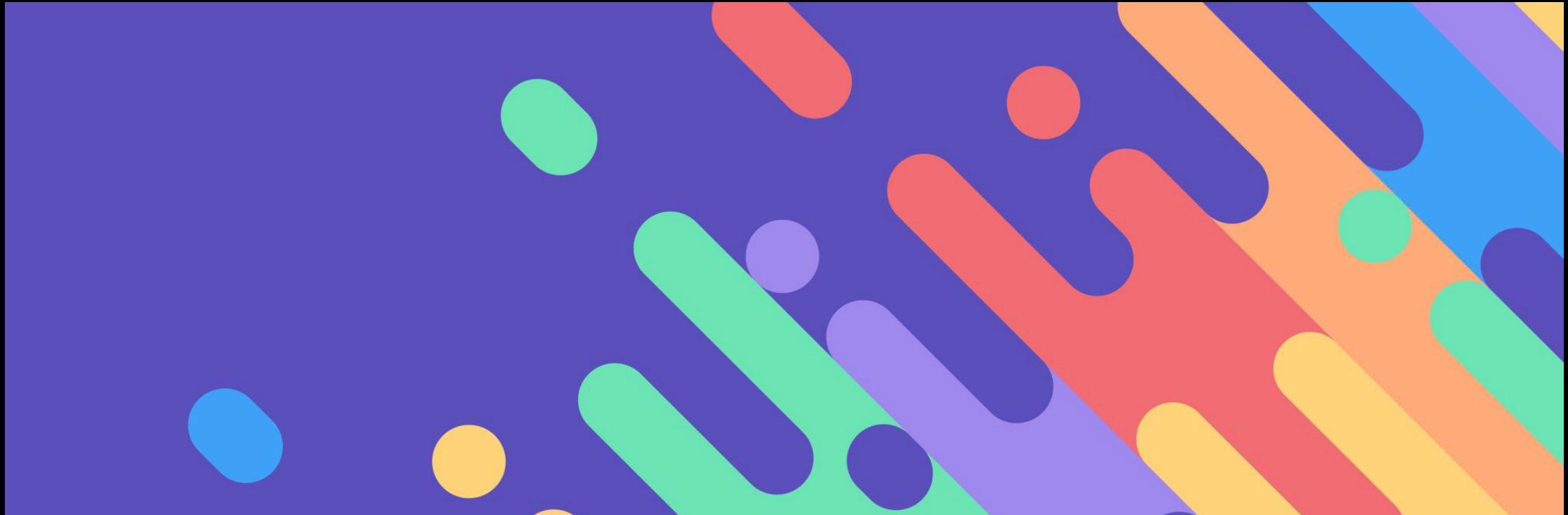
- Python es un lenguaje de alto nivel de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código.
- Python es un lenguaje de programación **multiparadigma**. Permite varios estilos: **programación orientada a objetos**, **programación imperativa** y **programación funcional**.
- OBS: En el fondo, Python es un lenguaje orientado a objetos, todo, absolutamente todo es un objeto.
- Python usa tipado dinámico y conteo de referencias para la gestión de memoria.
- Python reemplazó en gran medida a LISP en IA, principalmente por ser multiparadigma.

PYTHON

Python es famoso por ser lento comparado con lenguajes como C++, por qué se usa en Machine Learning o IA?

- La respuesta es que no se usa librerías hechas Python. Ninguna de las bibliotecas que se utilizan está realmente escrita en Python.
- Casi siempre están escritos en Fortran o C++ y simplemente interactúan con Python a través de algún wrapper.
- La velocidad de Python es irrelevante si solo se interactúa con las librerías escritas en un C++ altamente optimizado.

Fuente: <https://qr.ae/pKrGdr>

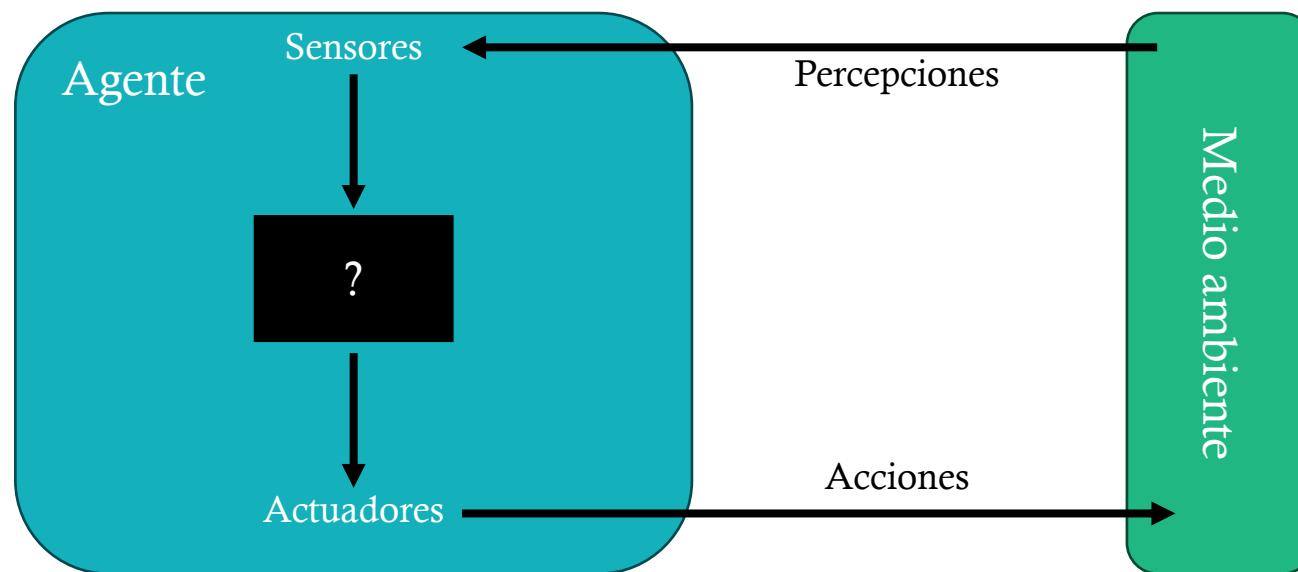


AGENTE RACIONAL

AGENTES RACIONALES

Agente

Un agente es cualquier cosa capaz de percibir su **medioambiente** con la ayuda de **sensores** y actuar en ese medio utilizando **actuadores**.



AGENTES RACIONALES

Agente

El término **percepción** se utiliza para indicar que el agente puede recibir entradas en cualquier instante.

La **secuencia de percepciones** de un agente refleja el historial completo de lo que el agente ha recibido.

Una elección de acción de un agente en un momento dado puede depender en su conocimiento incorporado y en la secuencia completa de percepciones hasta ese instante, pero no en cualquier cosa que no haya percibido.

AGENTES RACIONALES

Agente

En términos matemáticos, el comportamiento del agente viene dado por la **función del agente** que mapea una percepción dada en una acción.

En principio, con tiempo infinito, podemos construir una tabla que tabule cada acción dada una secuencia de percepción.

La tabla es una caracterización externa del agente. Internamente, la **función del agente** para un agente artificial será implementada por un **programa del agente**.

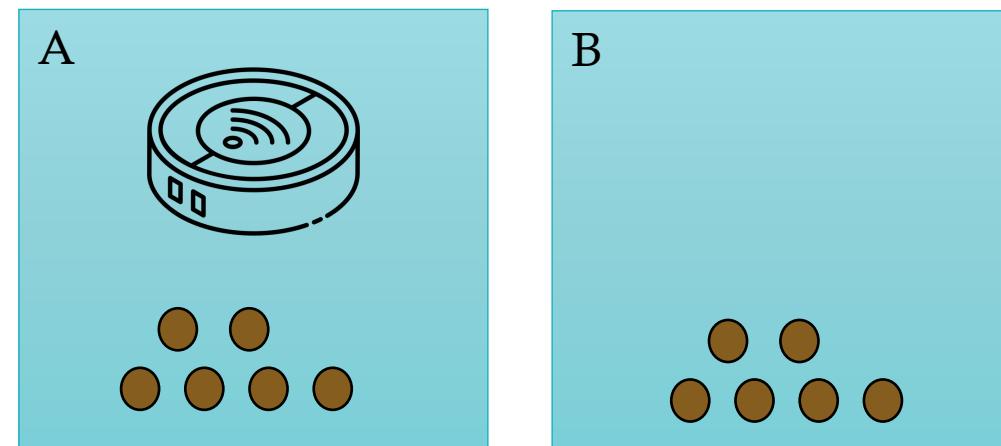
Es importante mantener estas dos ideas distintas.

- La **función del agente** es una descripción matemática abstracta.
- El **programa del agente** es una implementación concreta que se ejecuta dentro de algún sistema físico.

AGENTES RACIONALES

Agente - Ejemplo

Tenemos una **agente aspiradora** en un mundo que consiste en dos cuadros que pueden estar sucios o limpios (A y B). La aspiradora percibe si el cuadro en que está se encuentra limpio o sucio. Sus acciones son: **Mover a la izquierda**, **Mover a la derecha**, **limpiar el cuadrado**, **no hacer nada**.



AGENTES RACIONALES

Agente - Ejemplo

La tabla (de forma parcial) de secuencia de percepción y acción:

Secuencia de percepción	Acción
[A, Limpio]	Derecha
[A, Sucio]	Limpiar
[B, Limpio]	Izquierda
[B, Sucio]	Limpiar
[A, Limpio], [A, Limpio]	Derecha
[A, Limpio], [A, Sucio]	Limpiar
...	
[A, Limpio], [A, Limpio], [A, Limpio]	Derecha
[A, Limpio], [A, Limpio], [A, Sucio]	Limpiar
...	

AGENTES RACIONALES

Agente - Ejemplo

Un programa del agente para el ambiente de dos cajas, este programa implementa la tabla:

```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```

codetoplay.com

AGENTES RACIONALES

Medida de rendimiento



Las medidas de rendimiento incluyen los criterios que determinan el éxito en el comportamiento del agente. Estos deben ser objetivos, y en general, determinados por el diseñador.

El ejemplo de la aspiradora se puede proponer utilizar como medida de rendimiento **la cantidad de suciedad limpiada en un período de 8 horas**.

Pero ojo, un agente racional puede maximizar esta medida de rendimiento limpiando la suciedad, tirando la basura al suelo, limpiándola de nuevo, y así sucesivamente.

La selección de la medida de rendimiento no es siempre fácil.

AGENTES RACIONALES

Medida de rendimiento



Como regla general, es mejor diseñar medidas de utilidad de acuerdo con lo que se quiere para el entorno, más que de acuerdo con cómo se cree que el agente debe comportarse.

Por ejemplo, recompensar al agente un punto por cada cuadricula limpia en cada periodo de tiempo.

Pero esto también puede llevarnos a un problema, dado que la noción de limpieza de suelo limpio está basada en un en nivel de limpieza promedio a lo largo del tiempo. Y esto se puede alcanzar:

- Haciendo una limpieza mediocre pero continua
- Limpiando en profundidad, con largos descansos.

La forma más adecuada es una cuestión filosófica.

AGENTES RACIONALES

Racionalidad

La racionalidad en un momento dado depende de cuatro factores:

- La medida de rendimiento que define el criterio de éxito.
- El conocimiento previo del agente sobre el entorno..
- Las acciones que el agente puede llevar a cabo.
- La secuencia de percepciones del agente hasta este momento.

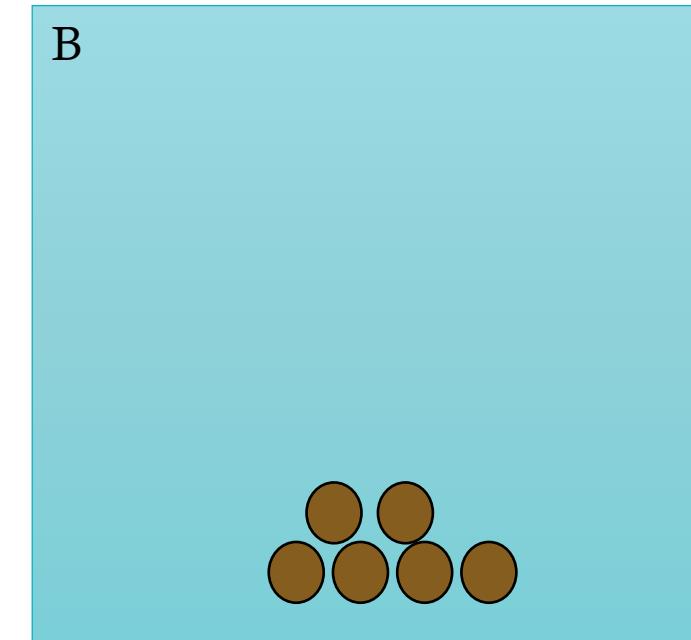
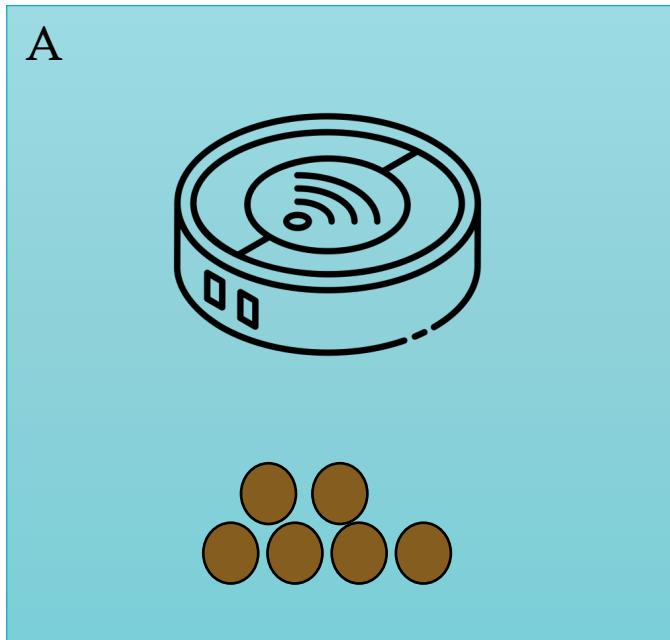
Definición de racionalidad

En cada posible secuencia de percepciones, un agente racional deberá seleccionar aquella acción que supuestamente maximice su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en el conocimiento que el agente mantiene almacenado.

AGENTES RACIONALES

Racionalidad

¿La función de la aspiradora es racional?

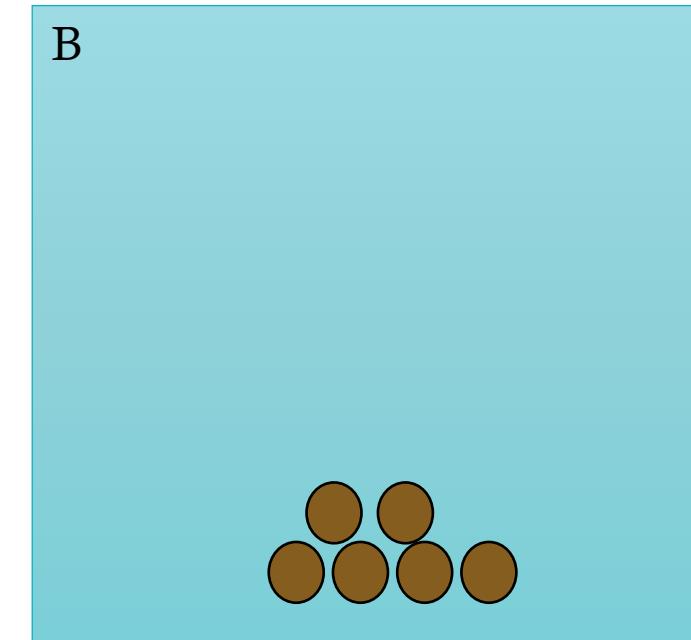
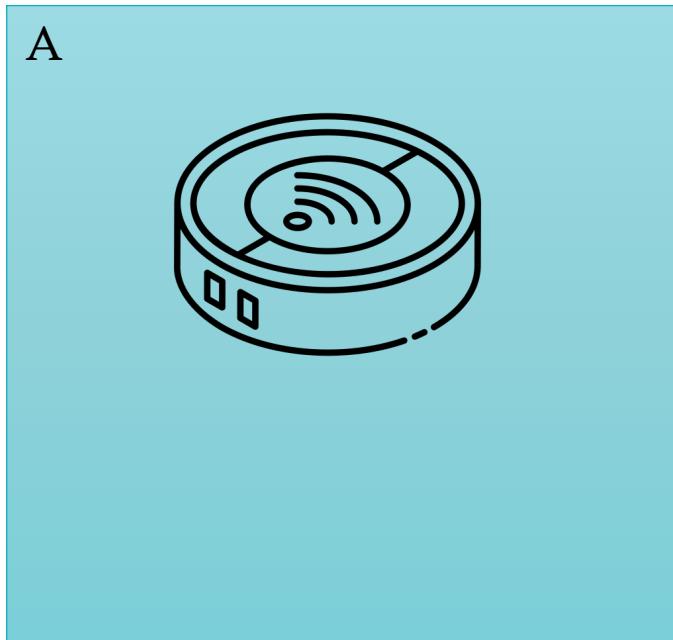


```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```

AGENTES RACIONALES

Racionalidad

¿La función de la aspiradora es racional?



```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```

AGENTES RACIONALES

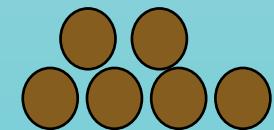
Racionalidad

¿La función de la aspiradora es racional?

A

B

```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```



AGENTES RACIONALES

Racionalidad

¿La función de la aspiradora es racional?

A

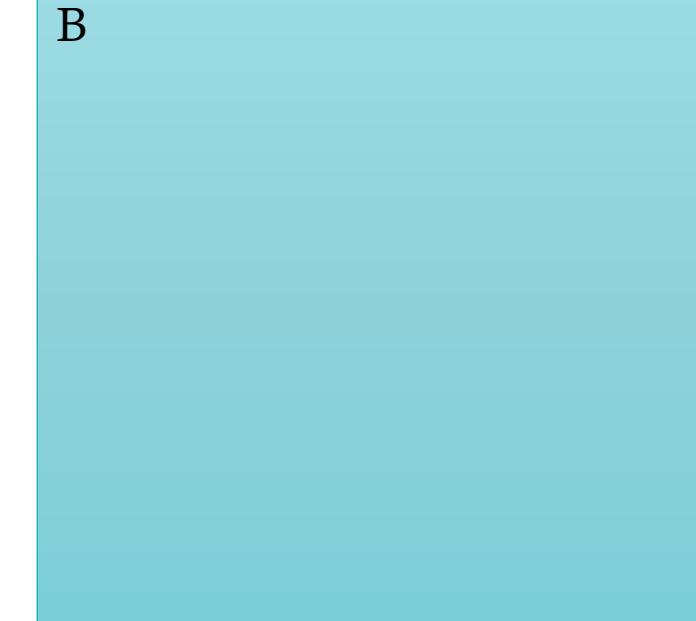
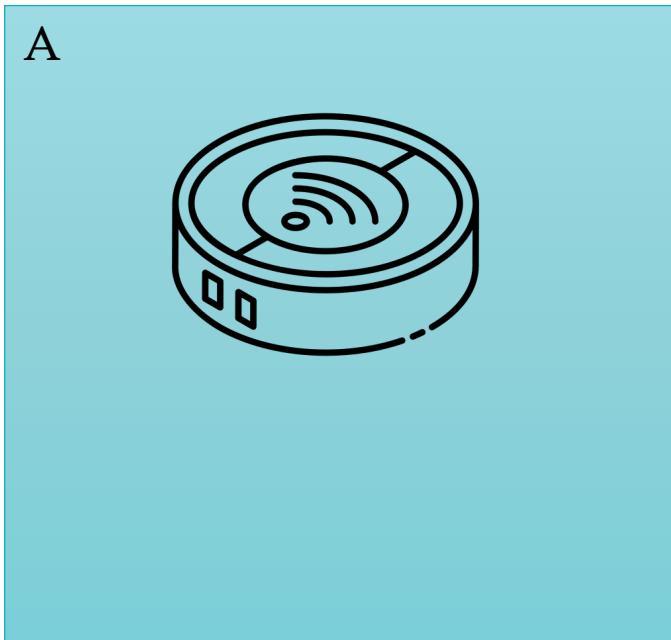
B

```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```

AGENTES RACIONALES

Racionalidad

¿La función de la aspiradora es racional?



```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```

AGENTES RACIONALES

Racionalidad

¿La función de la aspiradora es racional?

A

B

```
def REFLEX_VACUUM_AGENT(location, status) -> Action:  
    if status == "Dirty":  
        return Suck  
    elif location == A:  
        return Right  
    else: # location == B  
        return Left
```

AGENTES RACIONALES

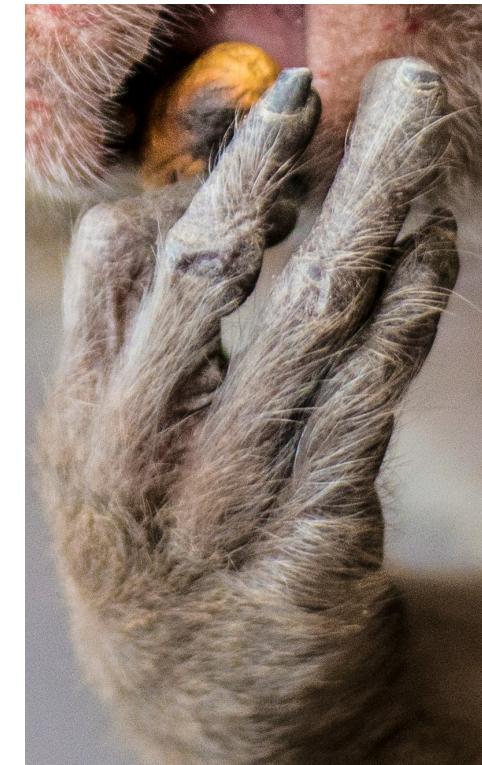
Racionalidad

¿La función de la aspiradora es racional? Depende

Asumamos lo siguiente:

- Damos un punto por rendimiento por cada espacio limpio en un tiempo de vida de 10000 pasos
- Se conoce la geografía del ambiente a “a priori”, pero no la distribución de suciedad. El espacio una vez que se limpia, se mantiene. Y el movimiento de izquierda y derecha siempre asegura el movimiento correcto.
- Solo tenemos tres acciones disponibles: Derecha, Izquierda y Limpiar.
- El agente siempre percibe correctamente su ubicación y si el piso está sucio.

Bajo estas circunstancias el agente es **racional**.



AGENTES RACIONALES

Racionalidad

¿La función de la aspiradora es racional? Depende

Pero cambiando un poco las condiciones para que el mismo sea **irracional**:

Agregamos una medida de penalización de un punto por cada movimiento. Esta función del agente lo deja oscilando.

- Una solución mejor es para a la aspiradora cuando termina de aspirar, algo que implica memoria.
- Si el piso se puede volver a ensuciar, se puede realizar rutinas de chequeo.
- Si la geografía no se encuentra, el agente necesita explorar mediante algún algoritmo de búsqueda.

AGENTES RACIONALES

Especificación del entorno de trabajo

En el ejemplo de racionalidad de una agente aspiradora, hubo que especificar las medidas de rendimiento, el entorno, y los actuadores y sensores del agente.

Todo ello forma lo que se llama el entorno de trabajo, cuya denominación es **PEAS**:

- **Performance**
- **Environment**
- **Actuators**
- **Sensors**

AGENTES RACIONALES

Especificación del entorno de trabajo

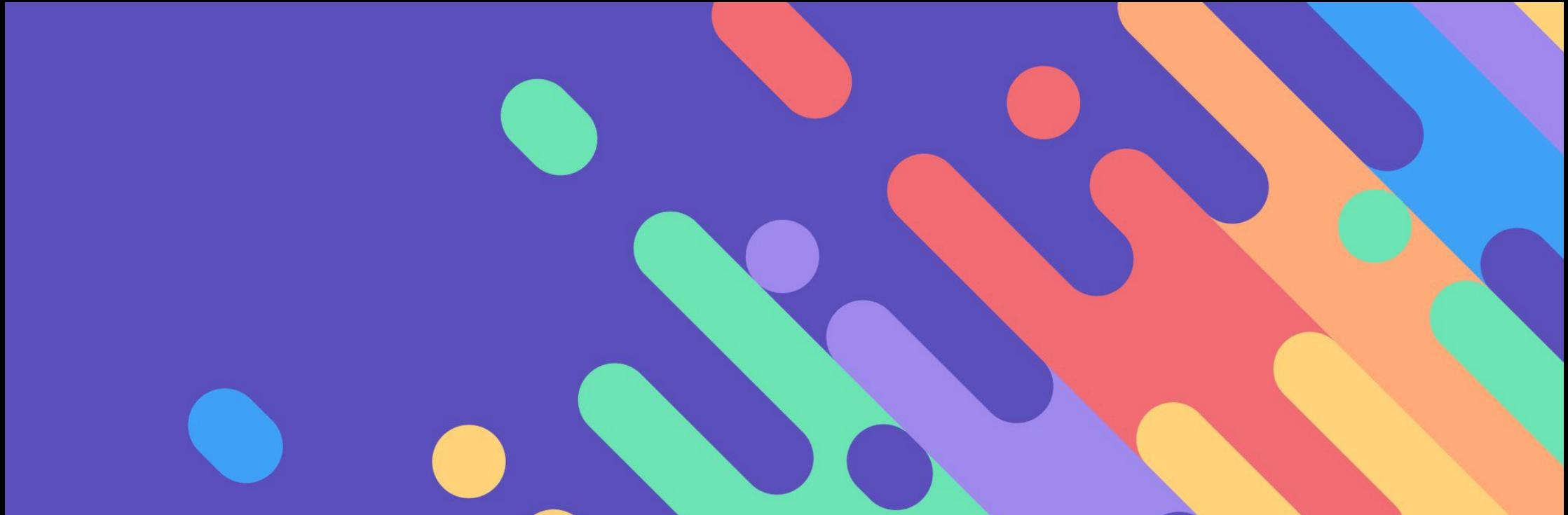
Agente	Performance	Environment	Actuators	Sensors
Sistema de diagnóstico medico	Paciente sano, reducir costos	Paciente, personal del hospital	Display de preguntas, test, diagnosis y tratamientos	Pantalla táctil/entrada por voz
Sistema de análisis de imágenes de satélite	Categorización correcta de objetos, terreno	Satélite en órbita, enlace, clima	Visualización de categorización de escenas	Cámara digital de alta resolución
Robot levanta piezas	Porcentaje de piezas en contenedores correctos	Cinta transportadora con piezas Contenedores	Brazo y mano articulados	Sensores de cámara, táctiles y de ángulo articular

AGENTES RACIONALES

Propiedades del entorno de trabajo

Veamos algunas dimensiones que podemos categorizar a los entornos:

- Totalmente observable vs. parcialmente observable
- Deterministas vs. Estocástico
- Episódico vs. Secuencial
- Estático vs. Dinámico
- Discreto vs. Continuo
- Agente individual vs. Multiagente (competitivo o cooperativo)



PROGRAMA DE LOS AGENTES

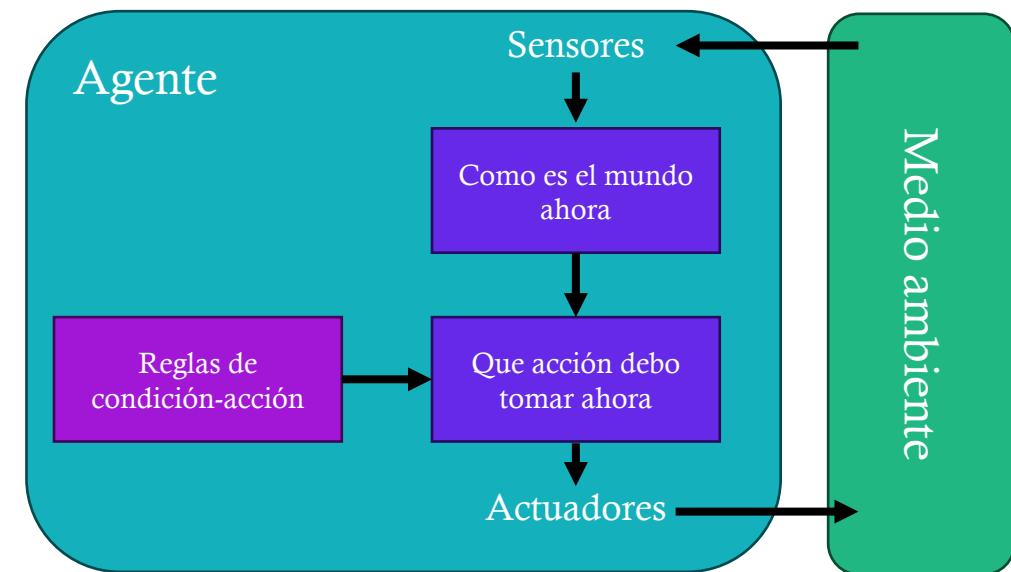
PROGRAMA DE LOS AGENTES

Agentes reactivos simples

```
def SimpleReflexAgentProgram(rules, interpret_input):

    def program(percept):
        state = interpret_input(percept)
        rule = rule_match(state, rules)
        action = rule.action
        return action

    return program
```



codeturing.com

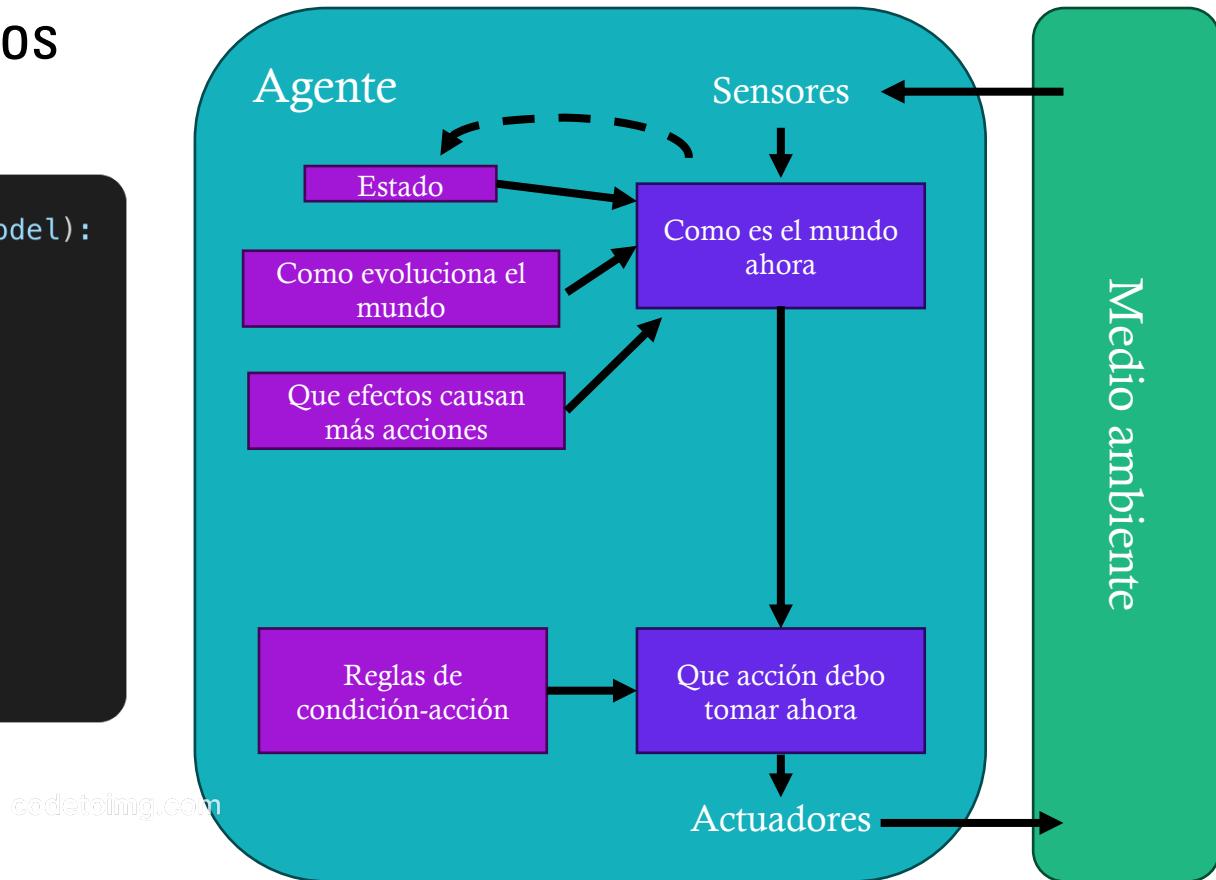
PROGRAMA DE LOS AGENTES

Agentes reactivos basados en modelos

```
def ModelBasedReflexAgentProgram(rules, update_state, model):

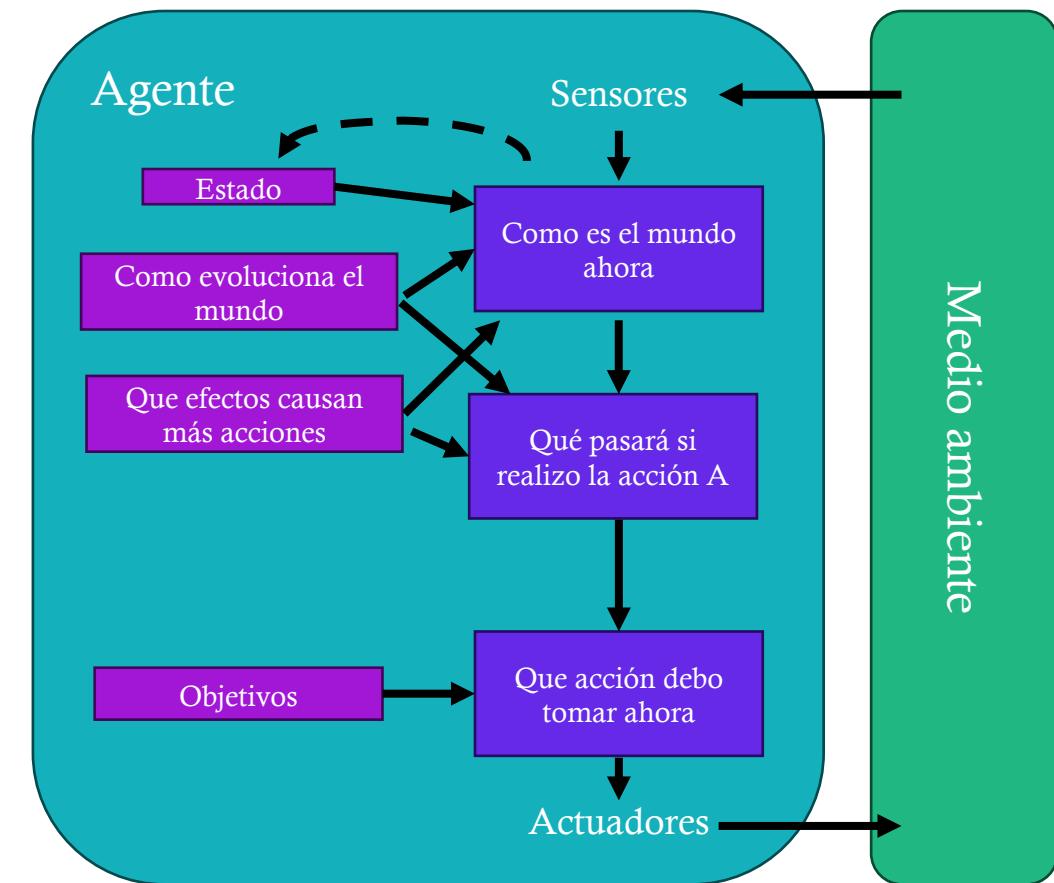
    def program(percept):
        program.state = update_state(program.state,
                                      program.action,
                                      percept, model)
        rule = rule_match(program.state, rules)
        action = rule.action
        return action

    program.state = program.action = None
    return program
```



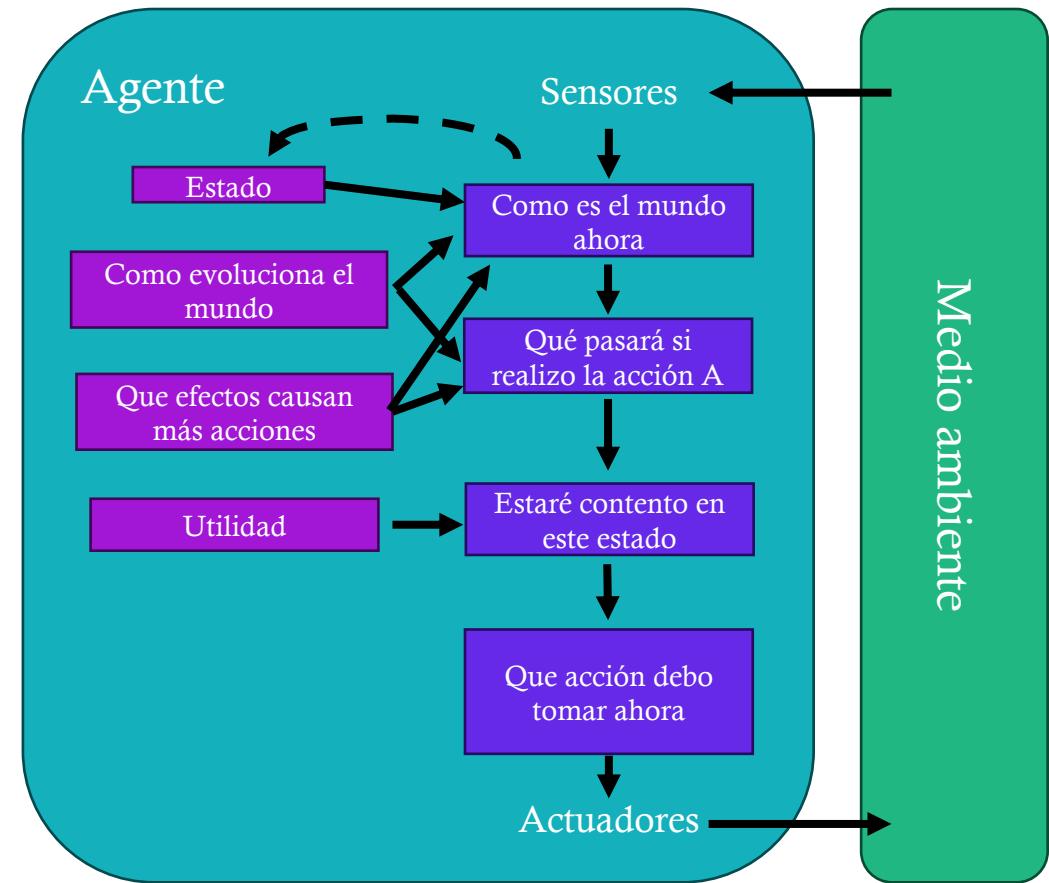
PROGRAMA DE LOS AGENTES

Agentes basados en objetivos



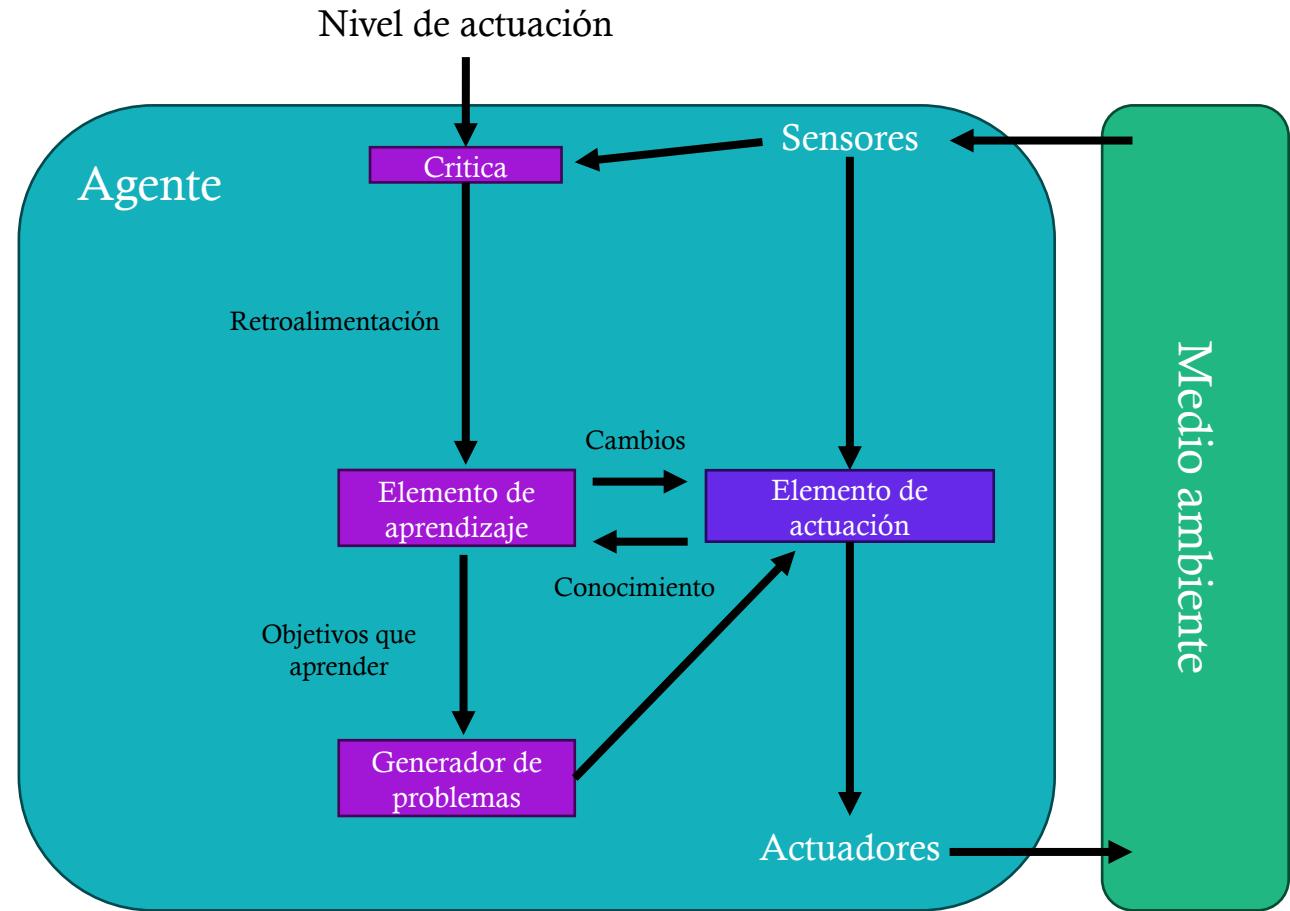
PROGRAMA DE LOS AGENTES

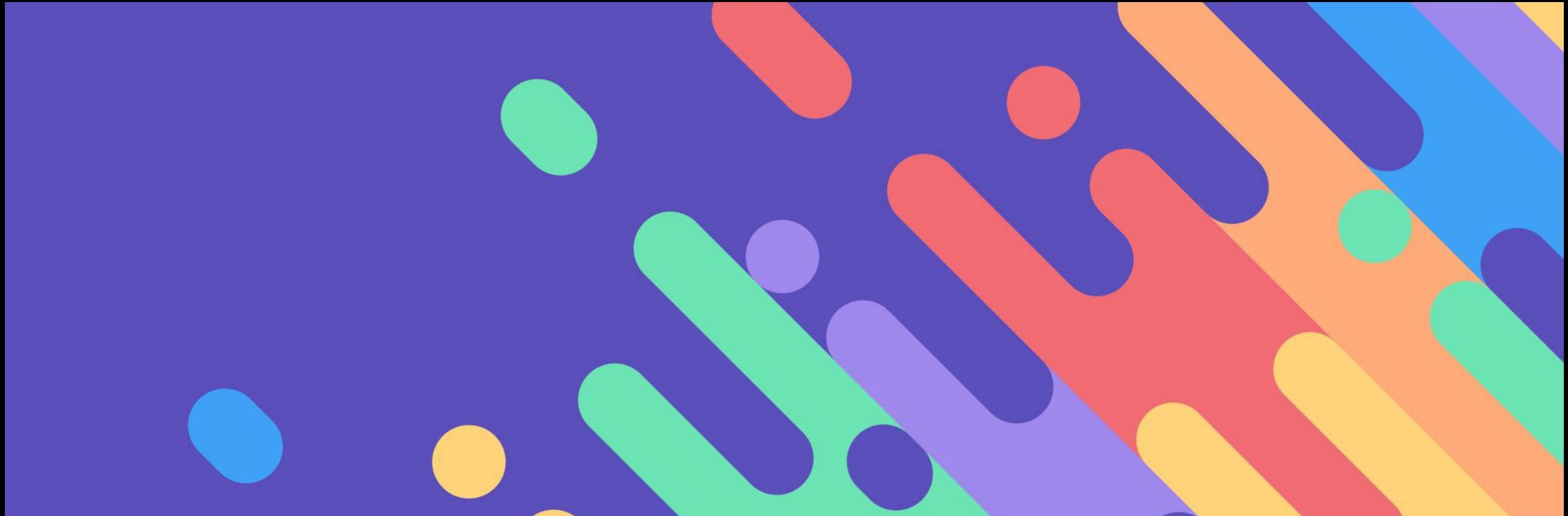
Agentes basados en utilidad



PROGRAMA DE LOS AGENTES

Agentes que aprenden





RESOLUCIÓN DE PROBLEMAS MEDIANTE BÚSQUEDA

RESOLUCIÓN DE PROBLEMAS MEDIANTE BÚSQUEDA

Agentes de resolución de problemas

Cuando la acción correcta a tomar no es inmediatamente obvia, un agente puede necesitar planificar con anticipación: considerar una secuencia de acciones que formen un camino hacia un estado objetivo. A dicho agente se le llama **agente de resolución de problemas** y el proceso computacional que lleva a cabo se llama **búsqueda**.

Para estos métodos de búsquedas, se considera sólo los entornos más simples: *episódico, de agente único, totalmente observable, determinista, estático, discreto y conocido*.

Dado estas condiciones, el agente puede llevar un proceso de 4 fases:

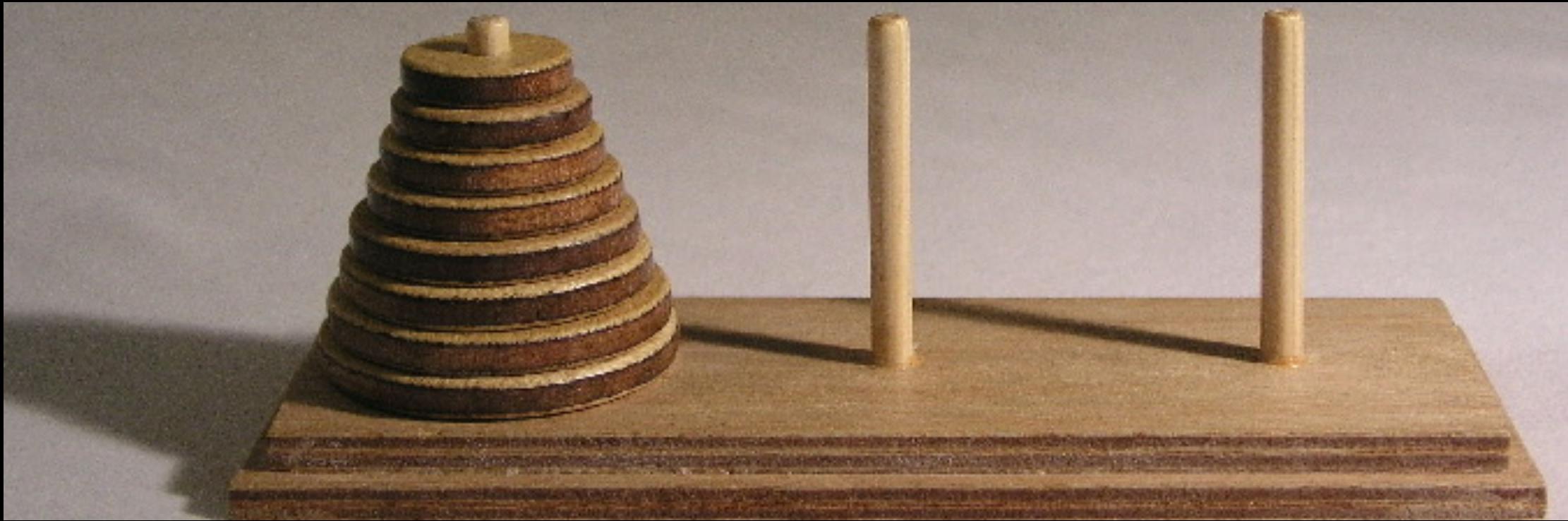
- **Formulación de objetivo:** El agente adopta el objetivo basado en la situación actual y la medida de rendimiento del agente.
- **Formulación del problema:** El agente diseña una descripción de los estados y acciones necesarias para alcanzar el objetivo: un modelo abstracto de la parte relevante del entorno.
- **Búsqueda:** Antes de realizar cualquier acción en el mundo real, el agente simula secuencias de acciones en su modelo, buscando hasta encontrar una secuencia de acciones que alcance el objetivo. Esta secuencia se llama solución.
- **Ejecución:** El agente ahora puede ejecutar las acciones de la solución, de a un paso por vez.

RESOLUCIÓN DE PROBLEMAS MEDIANTE BÚSQUEDA

Problemas de búsquedas y soluciones

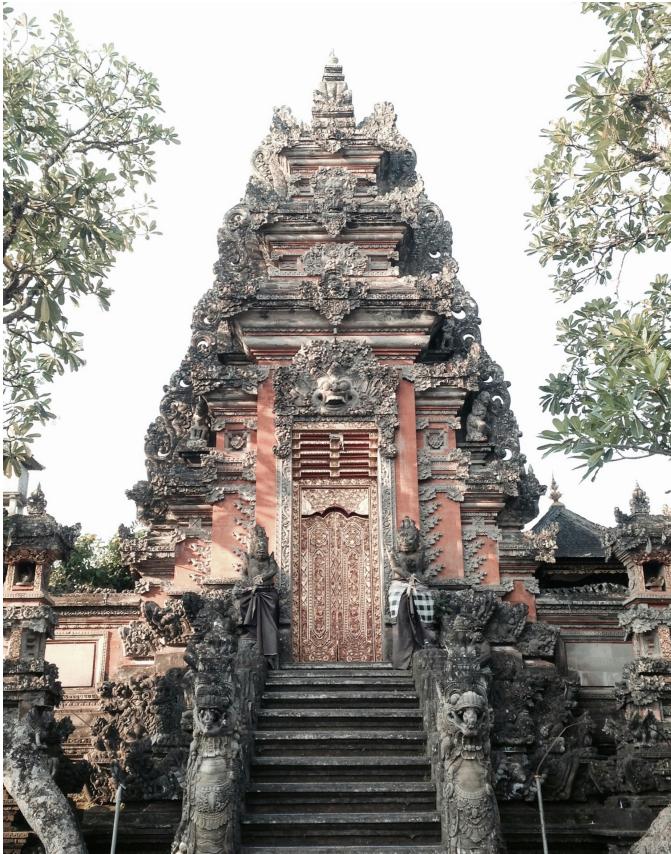
Un problema de búsqueda puede ser definido formalmente como:

- Un conjunto de estados posibles en los que puede estar el entorno, llamado **espacio de estados**.
- El **estado inicial** en que el agente comienza.
- Un set de uno o más **estados objetivos**.
- Las **acciones** disponibles al agente. Dado un estado s , $\text{ACTIONS}(s)$ retorna un numero finito de acciones que puede ejecutarse en s . Decimos que cada una de estas **acciones** es *aplicable* en s .
- Un **modelo de transición**, que describe lo que hace cada acción. $\text{RESULT}(s, a)$ devuelve el estado que resulta de realizar la acción a en el estado s .
- Una **función de costo de acción** ($\text{ACTION-COST}(s, a, s')$) que nos devuelva un número que denote el costo de aplicar una acción a a un estado s para llegar al estado s' .
- Una secuencia de acción forma un **camino**, y una **solución** es el camino del estado inicial a un estado objetivo.
- Si asumimos el costo es aditivo y positivo, el costo total es la suma del costo de cada acción. **Una solución óptima** es aquella que el costo es mínimo.

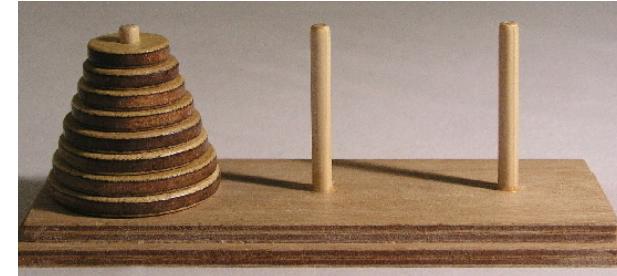


TORRE DE HANOI

TORRE DE HANOI



Pongámonos místicos...

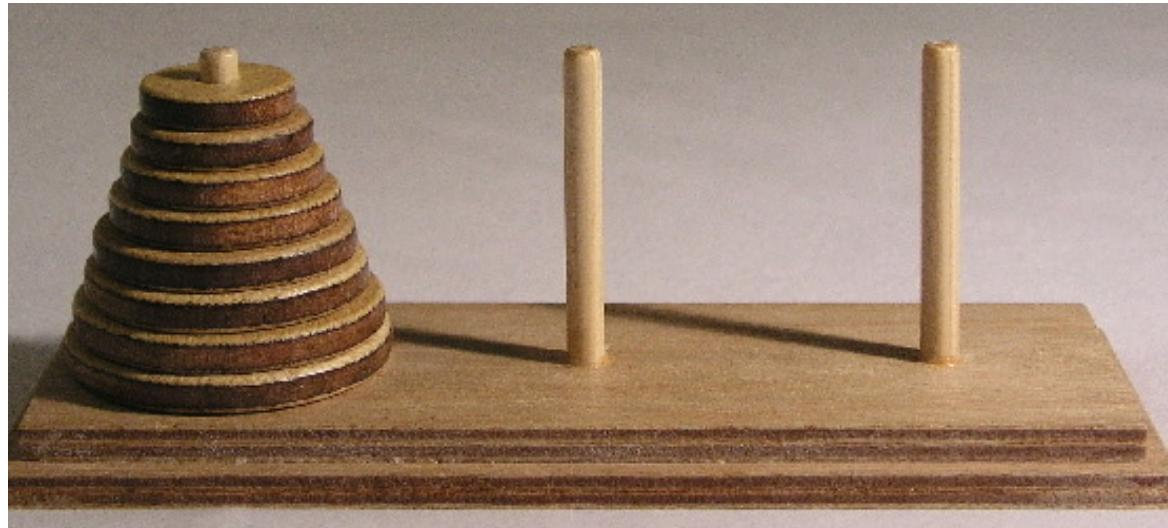


Cuenta la leyenda que unos brahmanes en un templo de Benarés han estado realizando el movimiento de la "Torre Sagrada de Brahma" sin parar desde hace siglos, la torre está formada por sesenta y cuatro discos de oro, y los movimientos obedecen a las siguientes místicas reglas:

1. Sólo se puede mover un disco a la vez.
2. Cada movimiento consiste en recoger el disco superior de una de las pilas y colocarlo encima de otra pila o sobre una varilla vacía.
3. Ningún disco podrá colocarse encima de un disco que sea más pequeño que él.

Una vez que finalicen la torre, va a llegar el fin del mundo.

TORRE DE HANOI



La Torre de Hanói es un rompecabezas inventado en 1883 por el matemático francés Édouard Lucas.

El rompecabezas comienza con los discos apilados en una varilla en orden de tamaño decreciente, el más pequeño en la parte superior, aproximándose así a una forma cónica.

El objetivo del rompecabezas es mover toda la pila a una de las otras barras, con las reglas de la leyenda:

1. Sólo se puede mover un disco a la vez.
2. Cada movimiento consiste en coger el disco superior de una de las pilas y colocarlo encima de otra pila o sobre una varilla vacía.
3. Ningún disco podrá colocarse encima de un disco que sea más pequeño que él.

TORRE DE HANOI

Resolviendo este problema usando IA

Este problema es un típico problema para aplicar métodos de búsquedas. Podemos crear un agente que pueda resolver este problema.

Limitemos a 5 discos, *salvo que quieran usar 64 discos como los brahmanes.*

El agente puede percibir cuantos discos y en qué orden hay en cada varilla. Además, puede tomar cualquier disco que se encuentre en la parte superior y moverlo a cualquier otra varilla que este permitido moverlo.

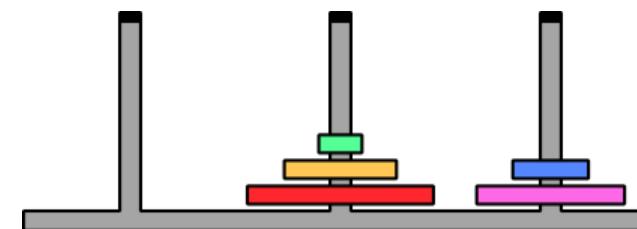
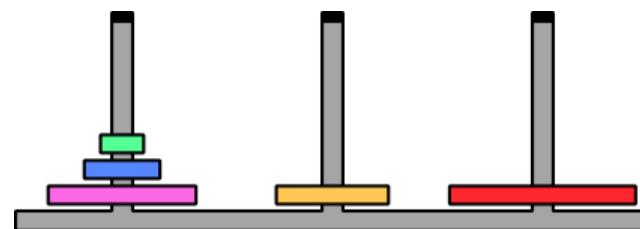
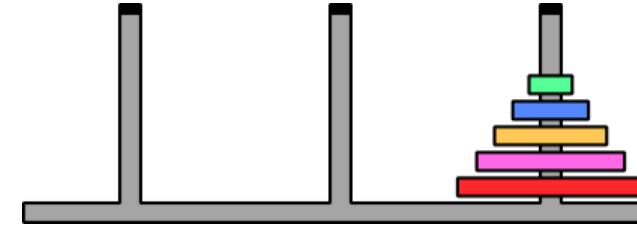
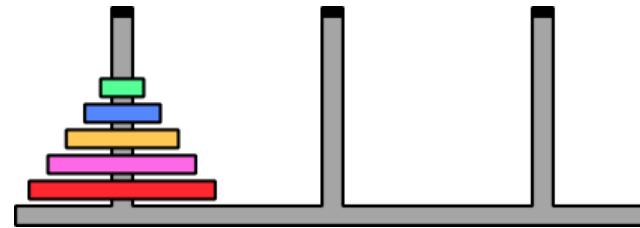
¿Cuáles son los **PEAS** de este problema? ¿Cuál es la característica del entorno de trabajo? Esto queda como tarea.

TORRE DE HANOI

Resolviendo este problema usando IA

Definamos el problema con las características que vimos:

Espacio de estados: Para 5 discos, tenemos $3^5 = 243$ posibles estados.

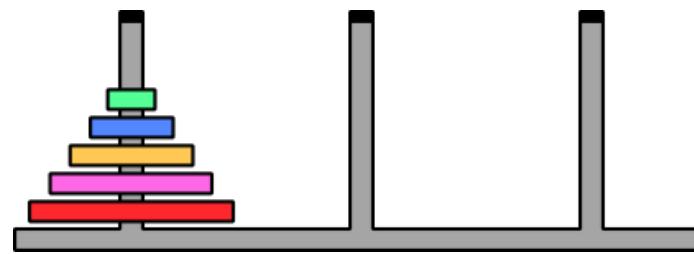


TORRE DE HANOI

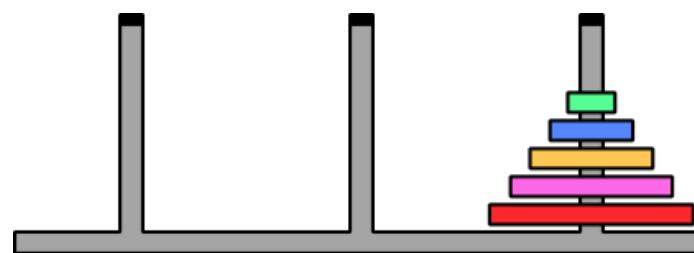
Resolviendo este problema usando IA

Definamos el problema con las características que vimos:

Estado inicial:



Estado objetivo: Para simplificar, vamos a tener un solo estado objetivo de los dos posibles.

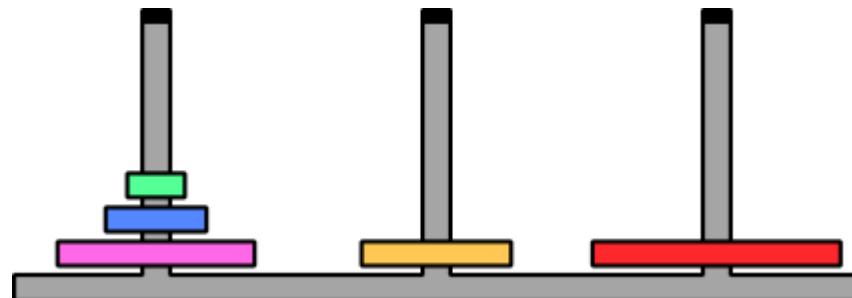


TORRE DE HANOI

Resolviendo este problema usando IA

Definamos el problema con las características que vimos:

Acciones ACTIONS (s), por ejemplo, para el siguiente estado, tenemos las siguientes acciones:



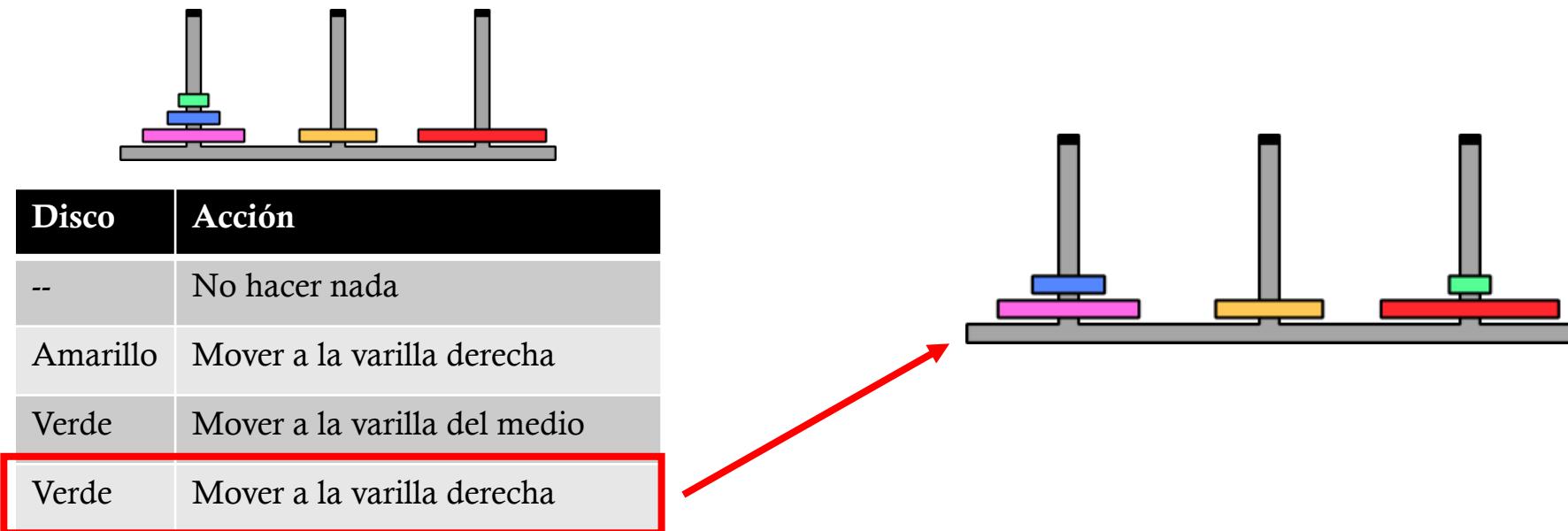
Disco	Acción
--	No hacer nada
Amarillo	Mover a la varilla derecha
Verde	Mover a la varilla del medio
Verde	Mover a la varilla derecha

TORRE DE HANOI

Resolviendo este problema usando IA

Definamos el problema con las características que vimos:

Modelo de transición: $\text{RESULT}(s, a)$, por ejemplo:



TORRE DE HANOI

Resolviendo este problema usando IA

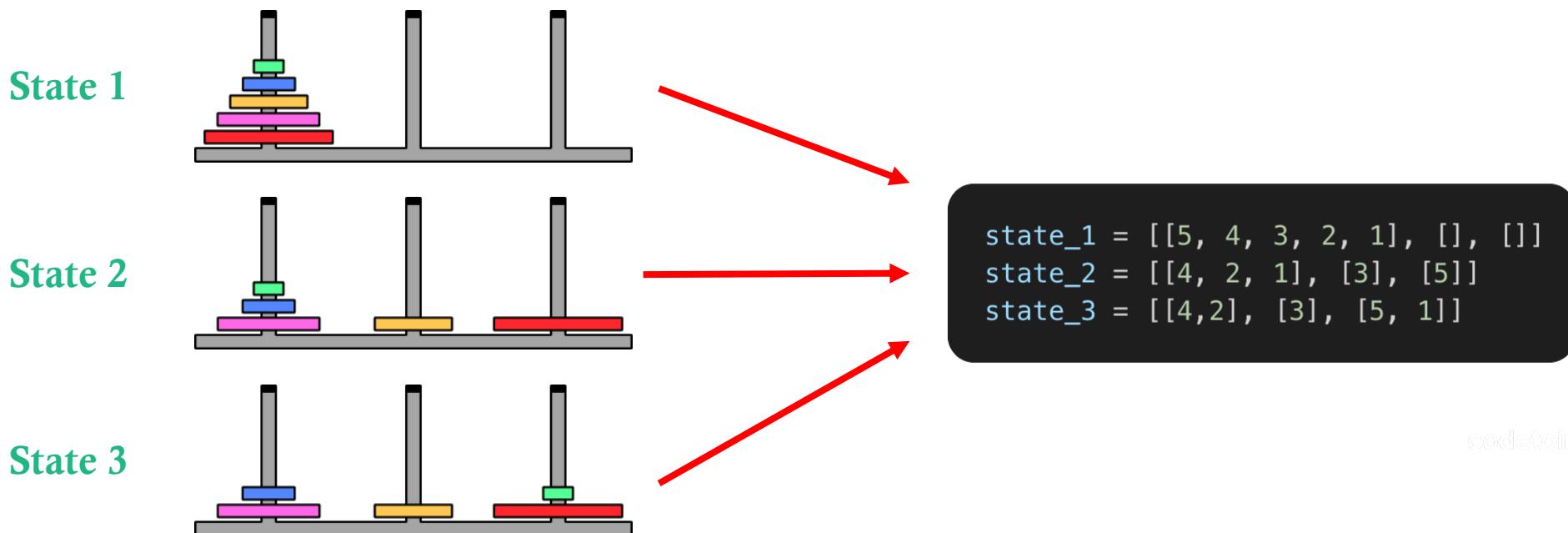
Definamos el problema con las características que vimos:

Función de costo de acción (**ACTION-COST** (s , a , s')). Mover un disco de una varilla a otra, siempre que sea un movimiento permitido, cuesta lo mismo, que podemos definir como 1.

TORRE DE HANOI

Resolviendo este problema usando IA

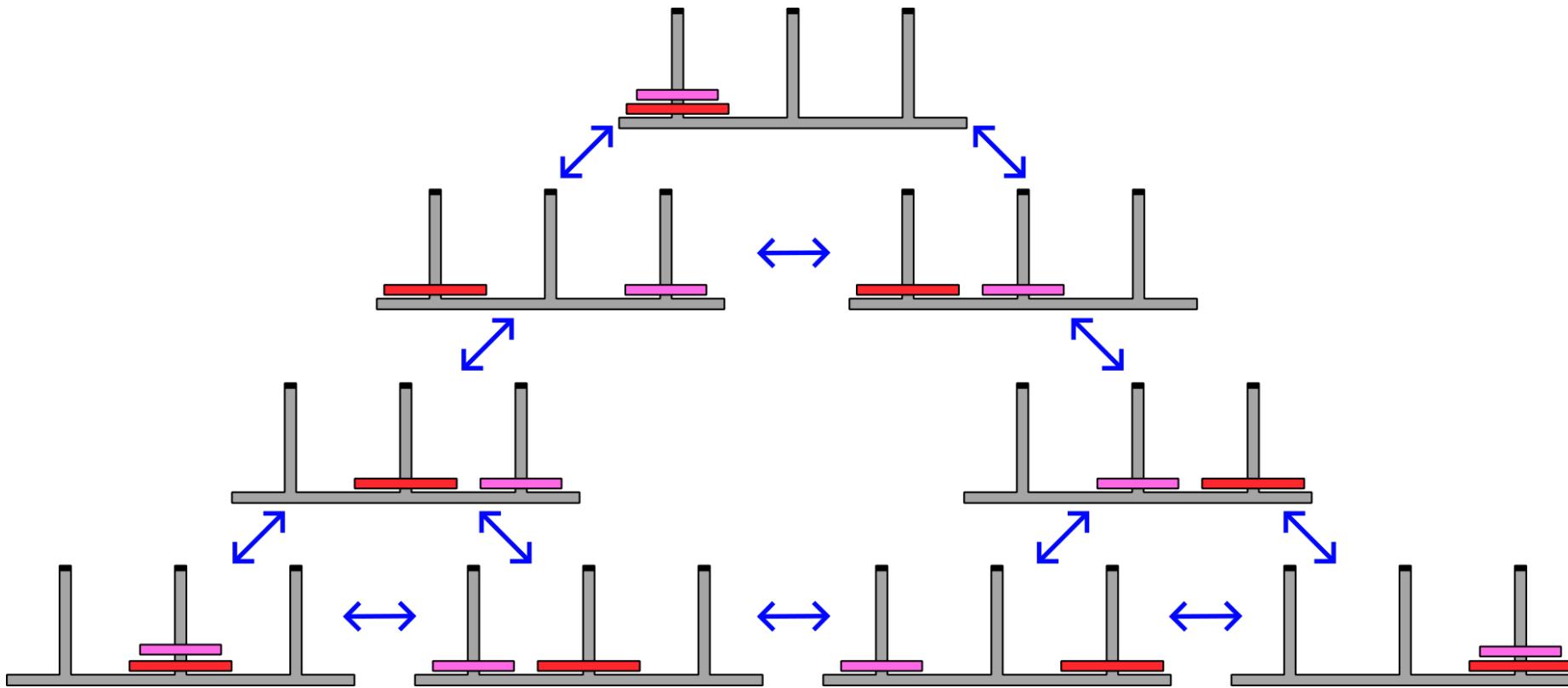
Debemos encontrar alguna forma de representación de estados y acciones que no sea solo con las imágenes y tablas. Para ello usemos a Python. Podemos usar tres listas, uno por varilla, y un número del 1 al 5 para cada disco, podemos aprovechar a los números para determinar si el disco puede ser colocado en una varilla o no:



codetomg.com

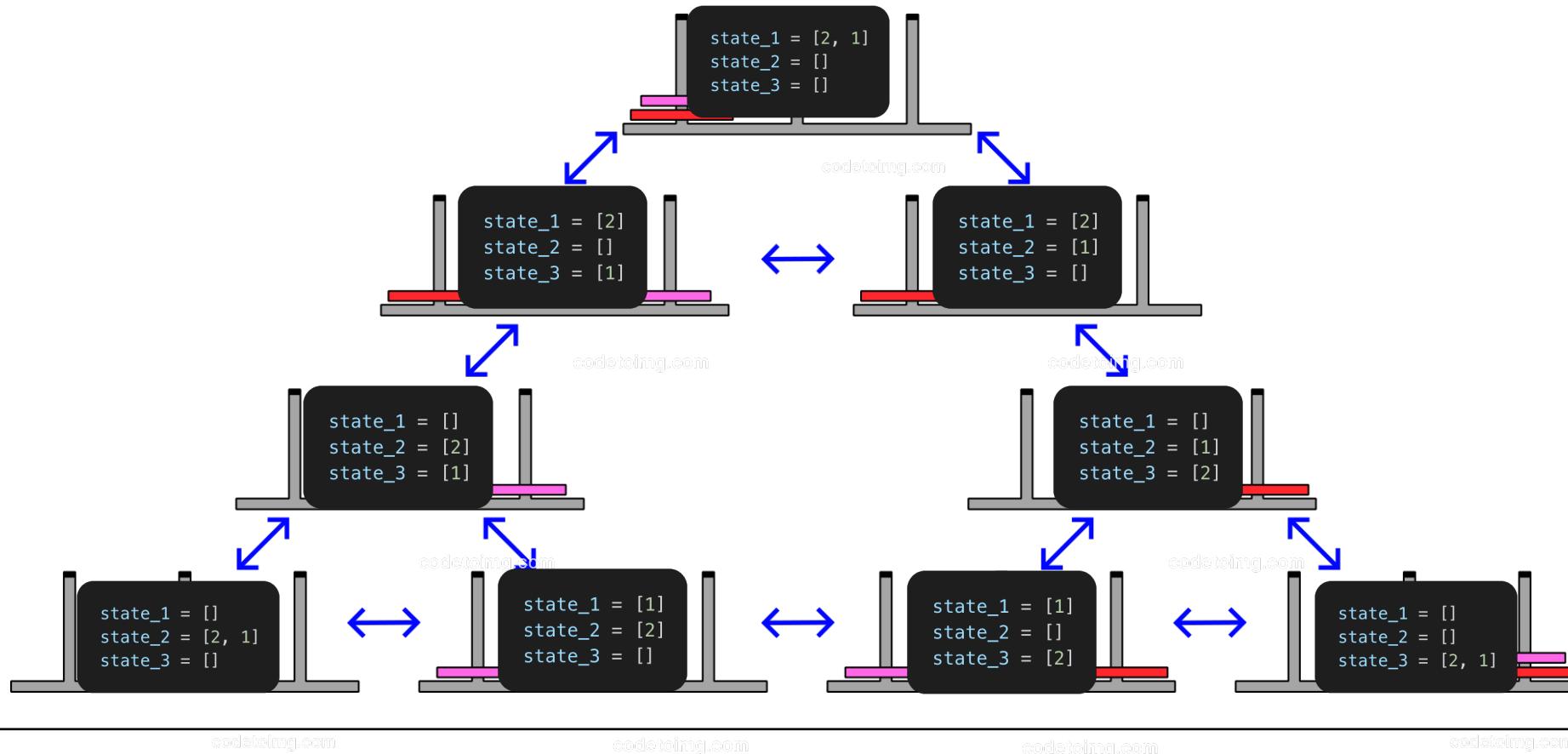
TORRE DE HANOI

Grafo de estados



TORRE DE HANOI

Grafo de estados

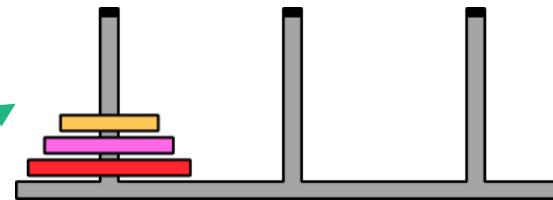


TORRE DE HANOI

Implementación en Python

```
class StatesHanoi:  
    # Atributos  
    self.rods = [rod1, rod2, rod3]  
    self.number_of_disks  
    self.number_of_pegs = 3  
    # Importante para el alg. de busqueda  
    self.accumulated_cost  
  
    # Metodos  
    get_last_disk_rod(number_rod)  
    check_valid_disk_in_rod(number_rod, disk)  
    put_disk_in_rod(number_rod, disk)  
    accumulate_cost(cost)  
  
    # Ejemplo  
    initial_state = StatesHanoi([3, 2, 1], [], [], max_disks=3)
```

Con esta clase podemos representar un nodo del grafo de la torre de Hanoi.

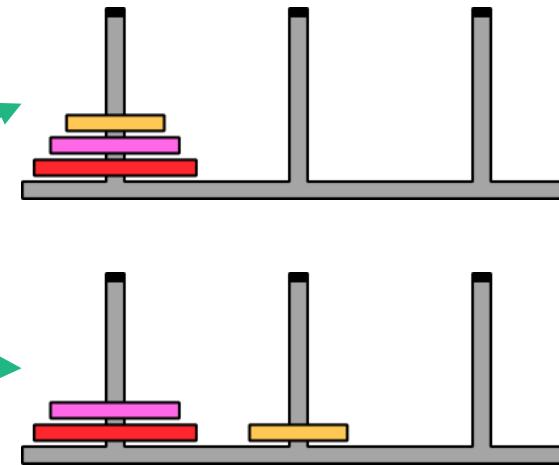


TORRE DE HANOI

Implementación en Python

```
class ActionHanoi:  
    # Atributos  
    self.disk  
    self.rod_input  
    self.rod_out  
    self.cost = 1.0  
  
    # Metodos  
    execute(StatesHanoi)  
  
    # Ejemplo  
    initial_state = StatesHanoi([3, 2, 1], [], [], max_disks=3)  
    action = ActionHanoi(disk=1, rod_input=1, rod_out=2)  
    new_state = action.execute()
```

Con esta clase podemos realizar una acción de mover un disco

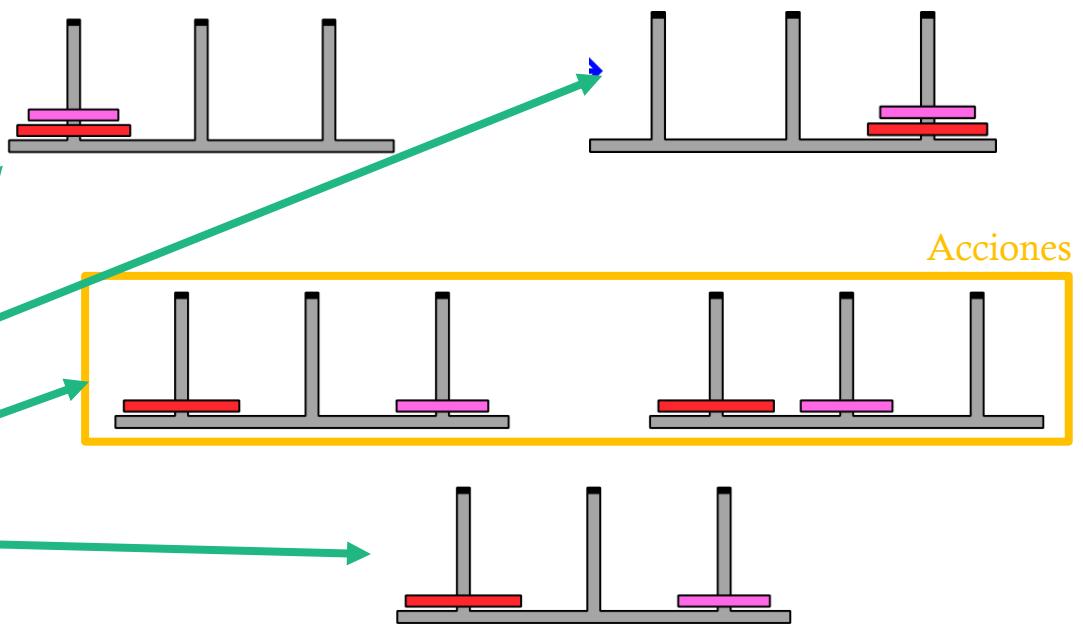


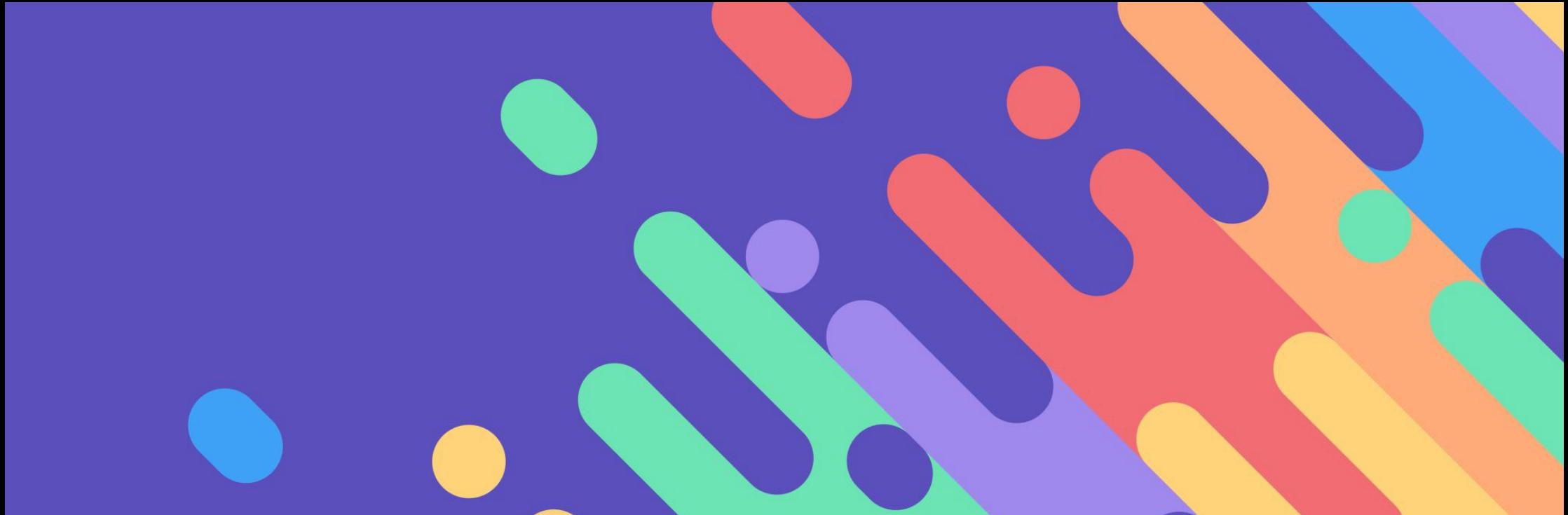
TORRE DE HANOI

Implementación en Python

```
class ProblemHanoi:  
    # Atributos  
    self.initial = StatesHanoi  
    self.goal = StatesHanoi  
  
    # Metodos  
    actions(StatesHanoi)  
    result(StatesHanoi, ActionHanoi)  
    goal_test(StatesHanoi)  
    path_cost(StatesHanoi, ActionHanoi)  
  
    # Ejemplo  
    initial_state = StatesHanoi([2, 1], [], [], max_disks=2)  
    goal_state = StatesHanoi([], [], [2, 1], max_disks=2)  
    problem = ProblemHanoi(initial_state, goal_state)  
  
    lista_extension = problem.actions(initial_state)  
    new_state = result(initial_state, lista_extension[0])
```

Con esta clase podemos describir el problema que queremos resolver.





ALGORITMOS DE BÚSQUEDA

ALGORITMOS DE BÚSQUEDA

Árbol de búsqueda

Un algoritmo de búsqueda toma un **problema de búsqueda** como entrada y retorna una **solución**, o una indicación de falla.

Vamos a considerar únicamente, a modo de cortar un tema inmenso, a solo aquello que superponen un **árbol de búsqueda** sobre el **grafo de espacios de estados**. La idea es buscar un camino que llegue al estado objetivo.

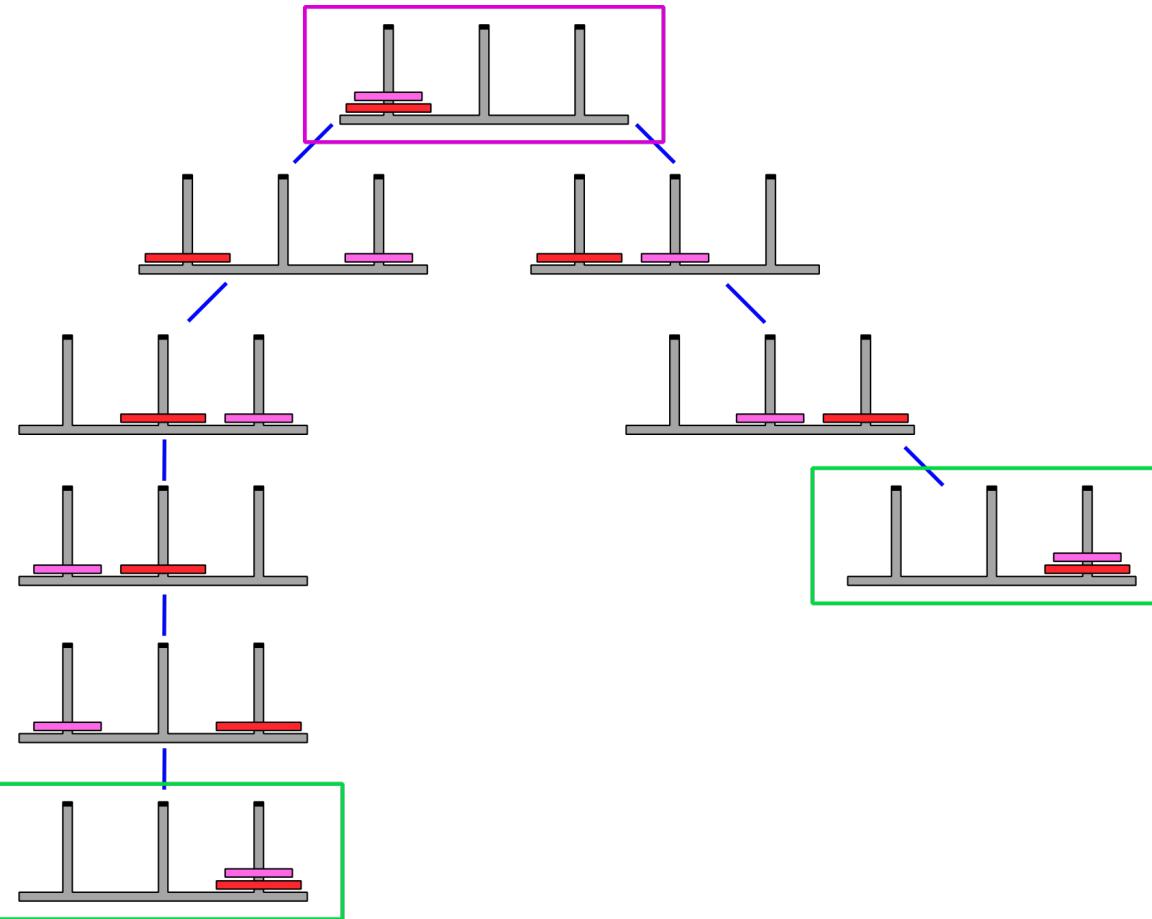
Cada **nodo** del árbol corresponde a un **estado** y las **aristas** corresponde a una **acción**.

Importante, el árbol **NO** es el grafo de estados. El grafo describe todo el set de estados, y las acciones que llevan de un lado a otro.

El árbol describe el camino entre estos estados, para alcanzar el objetivo.

ALGORITMOS DE BÚSQUEDA

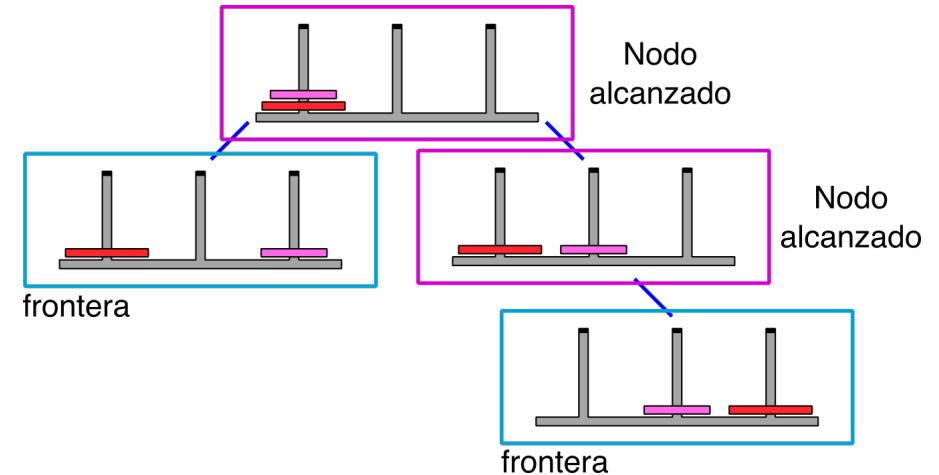
Árbol de búsqueda



ALGORITMOS DE BÚSQUEDA

Árbol de búsqueda

La frontera separa dos regiones del grafo, aquella que ya fue explorada por el algoritmo y aquella que no.



ALGORITMOS DE BÚSQUEDA

Árbol de búsqueda – Estructura de datos

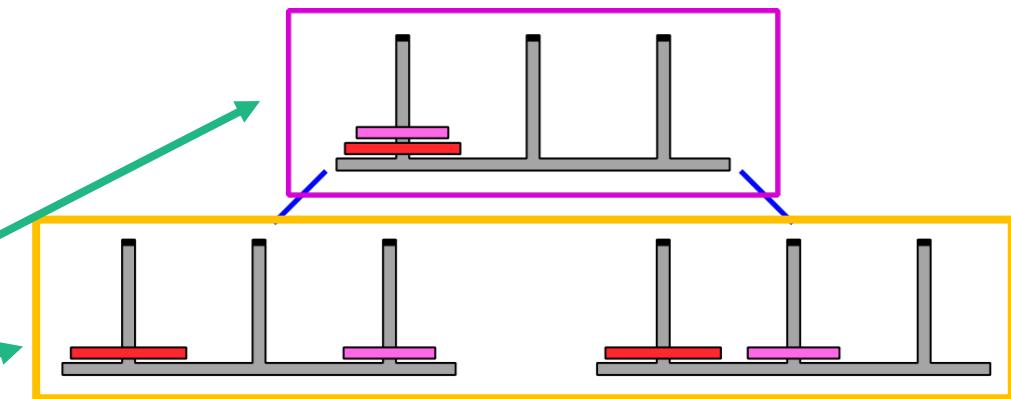
Para poder aplicar los algoritmos, debemos definir la estructura de datos para hacer seguimiento del árbol. Los **nodos** del árbol son representados con los siguientes componentes:

- **STATE**: El estado, del espacio de estados, que corresponde el nodo.
- **NODE PARENT**: El nodo en el árbol de búsqueda que ha generado al nodo.
- **ACTION**: La acción que se aplicará al padre para generar el nodo.
- **PATH-COST**: El costo $g(n)$ de un camino desde el nodo inicial al nodo.

ALGORITMOS DE BÚSQUEDA

Árbol de búsqueda – Implementación de Python para la torre de Hanoi

```
class NodeHanoi:  
    # Atributos  
    self.state = StatesHanoi  
    self.parent = NodeHanoi | None  
    self.action = ActionHanoi  
    self.path_cost  
  
    # Metodos  
    child_node(ProblemHanoi, ActionHanoi)  
    expand(ProblemHanoi)  
  
    # Ejemplo  
    initial_state = StatesHanoi([2, 1], [], [], max_disks=2)  
    root = NodeHanoi(initial_state, parent=None, action=None)  
    problem = ProblemHanoi(initial_state, goal_state)  
  
    lista_nodos_frontera = root.expand(problem)
```



ALGORITMOS DE BÚSQUEDA

Árbol de búsqueda – Estructura de datos

Necesitamos una estructura para la frontera. Seleccionamos una cola, porque las operaciones en la frontera son:

- **IS-EMPTY(FRONTIER)**: Retorna True si no hay nodos en la frontera.
- **POP(FRONTIER)**: Quita el primer nodo en la cola.
- **TOP(FRONTIER)**: Devuelve, pero no quita al primer nodo en la cola
- **ADD(FRONTIER)**: Inserta el nodo en su correspondiente lugar de la cola

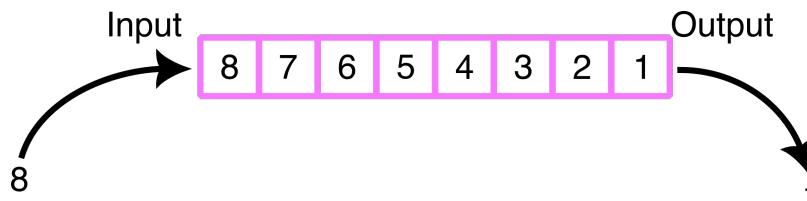
Tres tipos de colas se usan en los algoritmos, los cuales nos pueden dar diferentes tipos de resultados:

- Una **cola prioritaria** que primer quita nodos con el mínimo costo de acuerdo con una función de evaluación f .
- Una **cola FIFO** (primero entra, primero sale) que toma los nodos en el mismo modo que se agregan.
- Una **cola LIFO** (último en salir, sale primero... o stack) quita el nodo más reciente.

ALGORITMOS DE BÚSQUEDA

Estructura de colas:

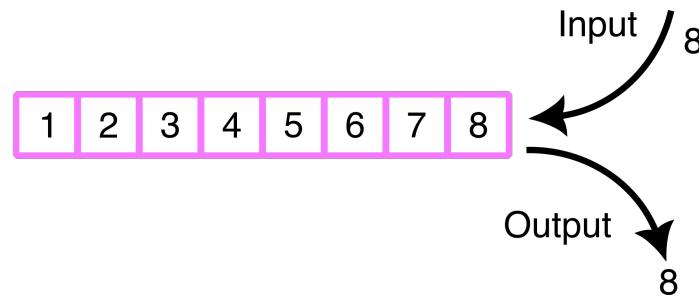
Cola FIFO:



```
fifo = []
fifo.insert(0, "1")
output = fifo.pop()
```

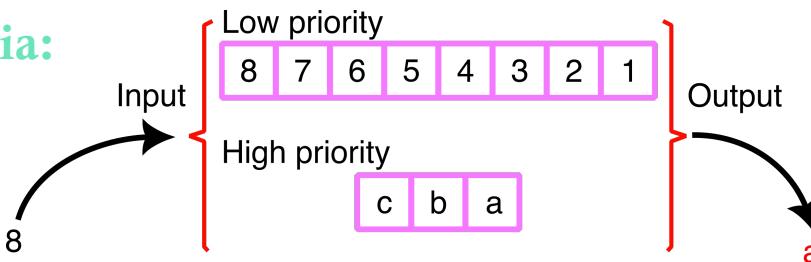
```
from collections import deque
fifo = deque()
fifo.append("1")
output = frontier.popleft()
```

Cola LIFO:



```
lifo = []
lifo.append("1")
output = lifo.pop()
```

Cola prioritaria:



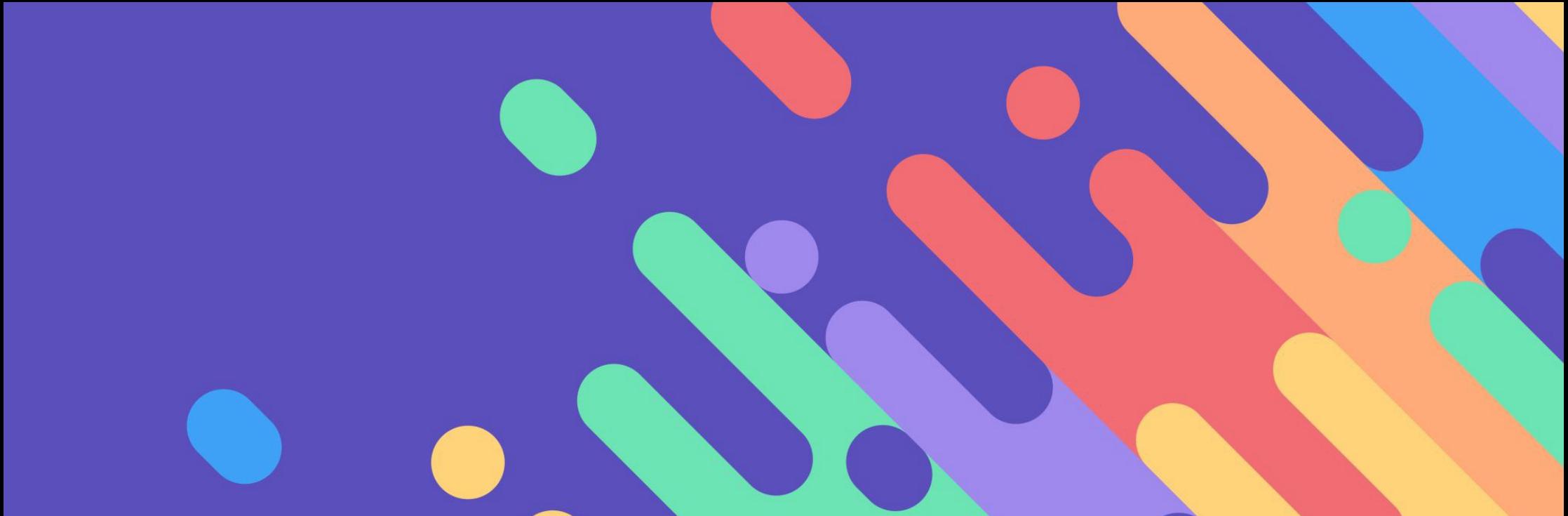
```
from aima import PriorityQueue
pq = PriorityQueue(order='min', func)
pq.append("1")
output = pq.pop()
```

ALGORITMOS DE BÚSQUEDA

Midiendo el rendimiento

Para poder evaluar a los algoritmos de búsquedas, debemos usar un criterio para elegir:

- **Completitud**: ¿El algoritmo garantiza encontrar una solución cuando hay una, y correctamente informar cuando no lo haya?
- **Optimización**: ¿encuentra la estrategia la solución óptima, es decir el camino más corto?
- **Complejidad de tiempo**: Cuanto tiempo le lleva encontrar la solución.
- **Complejidad de espacio**: Cuanta memoria es necesaria para la búsqueda



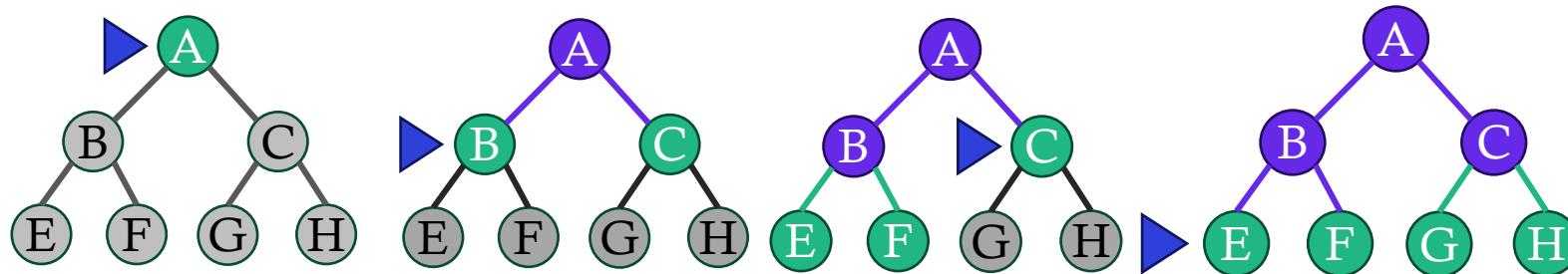
ALGORITMOS DE BÚSQUEDA NO INFORMADA

ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda primero en anchura

Es una estrategia sencilla en la que se expande primero el nodo raíz, a continuación, se expanden todos los sucesores del nodo raíz, después sus sucesores, etc.

Complejidad: $O(b^{d+1})$



```
# Pseudo-código
def breadth_first_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = FIFO(node)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s is not reached:
                reached.append(s)
                frontera.append(child)
    return "Failure"
```

ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda de costo uniforme o algoritmo de Dijkstra

En vez de expandir el nodo más superficial, la búsqueda de costo uniforme expande el nodo con el camino de costo más pequeño. **Si todos los costos son iguales, es idéntico a la búsqueda primero en anchura.**

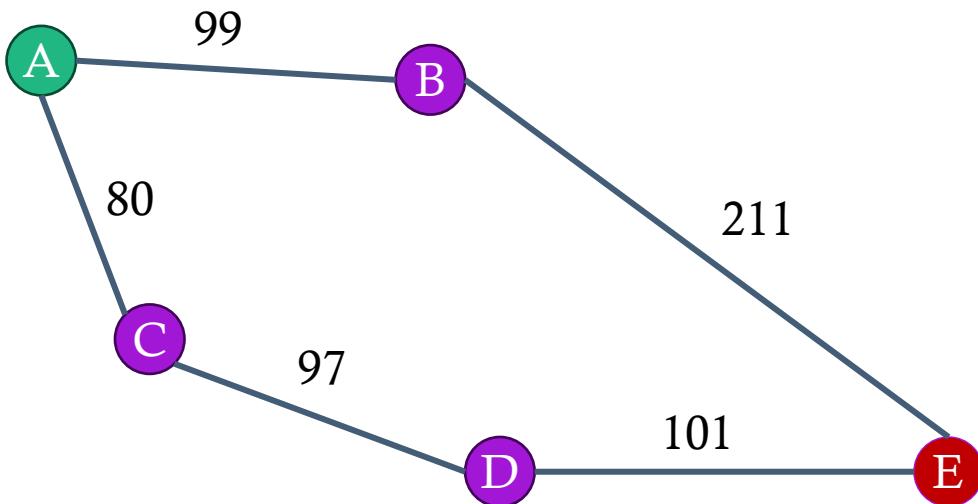
Para que funcione todos los caminos deben tener un costo positivo y mayor que cero, sino puede entrar en bucles infinitos.

Este método de búsqueda expande los caminos más cortos y luego los más grandes.

```
# Pseudo-código
def uniform_cost_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = PriorityQueue(node, f=PathCost)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s is not reached:
                reached.append(s)
                frontera.append(child)
    return "Failure"
```

ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda de costo uniforme o algoritmo de Dijkstra



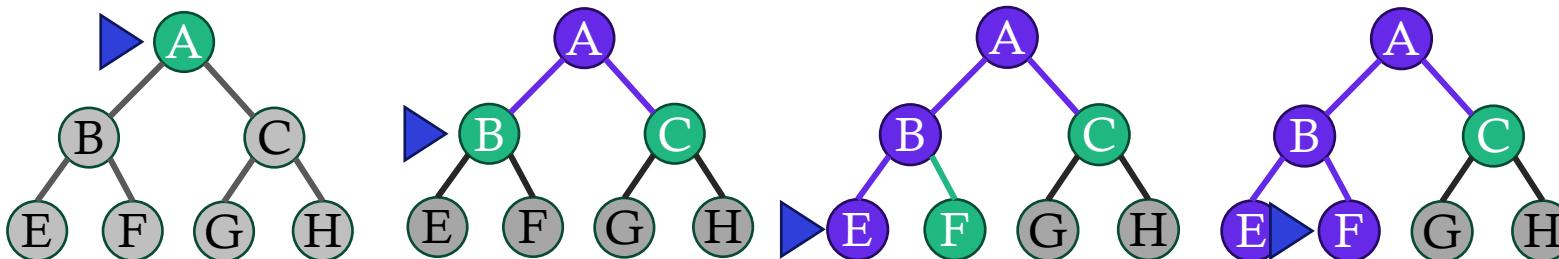
```
# Pseudo-código
def uniform_cost_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = PriorityQueue(node, f=PathCost)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s is not reached:
                reached.append(s)
                frontera.append(child)
    return "Failure"
```

ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda primero en profundidad

Siempre expande el nodo más profundo en la frontera actual del árbol de búsqueda. Cuando esos nodos se expanden, son quitados de la frontera, así entonces la búsqueda “**retrocede**” al siguiente nodo más superficial.

No encuentra la solución más eficiente, pero consume muy poca memoria $O(bm)$, y el tiempo es proporcional a la cantidad de estados.



```
# Pseudo-código
def best_first_search(problem):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = STACK(node)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s is not reached:
                reached.append(s)
                frontera.append(child)
    return "Failure"
```

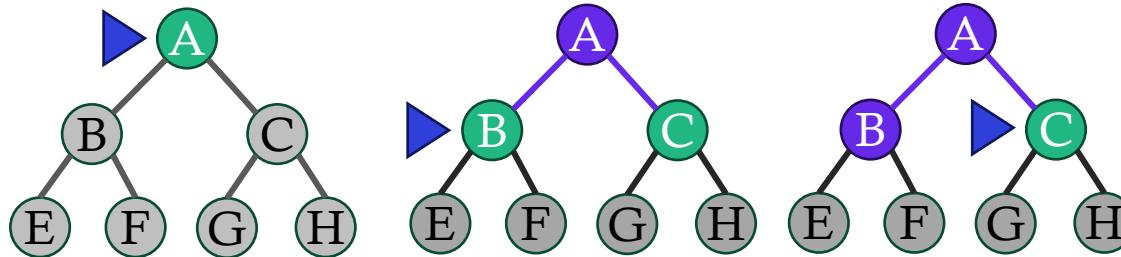
ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda de profundidad limitada

Para los casos de árboles muy grandes o infinitos, se puede limitar la búsqueda en profundidad hasta un cierto nivel. Por ejemplo, si elegimos una profundidad de 2, se llegaría hasta los nodos B y C.

Esto restringe que tan profundo avanza, pero la dificultad está en que se debe elegir la profundidad.

Complejidad en tiempo $O(b^l)$ y en memoria $O(bl)$



```
# Pseudo-código
def depth_limited_search(problem, limit):
    reached = []

    def recursive_dls(node, problem, limit):
        nonlocal explored
        if problem.is_goal(node.state):
            return node
        elif limit == 0:
            return 'cutoff'
        else:
            cutoff_occurred = False
            for child in expand(problem, node):
                s = child.state
                if s not in reached:
                    reached.append(s)
                    result = recursive_dls(child, problem, limit - 1)
                    if result == 'cutoff':
                        cutoff_occurred = True
                    elif result is not None:
                        return result
            if cutoff_occurred:
                return 'cutoff'
            return None

    return recursive_dls(Node(problem.initial), problem, limit)
```

ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda de profundidad limitada con profundidad iterativa

Esta búsqueda es una estrategia que se realiza aumentando la profundidad iterativamente hasta alcanzar la solución. Se parte de un límite de 0, 1, ...

La profundidad iterativa es el método de búsqueda no informada preferido cuando hay un espacio grande de búsqueda y no se conoce la profundidad de la solución.

```
# Pseudo-código
def iterative_deepening_search(problem):
    for depth in range(0, Inf):
        result = depth_limited_search(problem, depth)
        if result != 'cutoff':
            return result
```

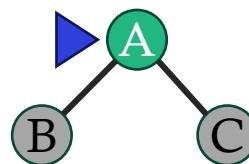
ALGORITMOS DE BÚSQUEDA NO INFORMADA

Búsqueda de profundidad limitada con profundidad iterativa

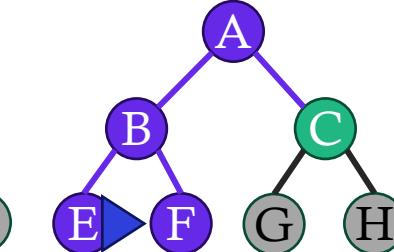
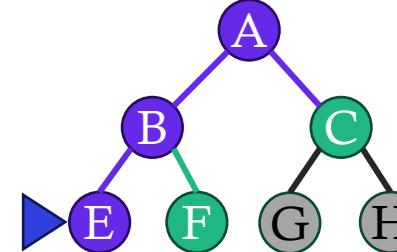
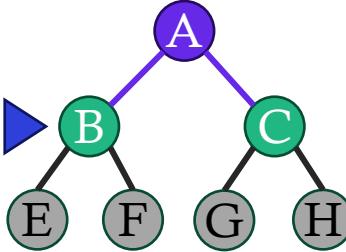
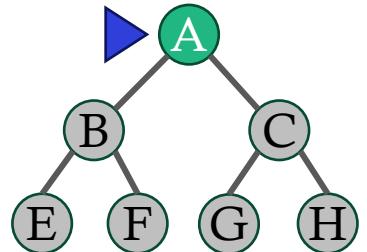
Límite = 0



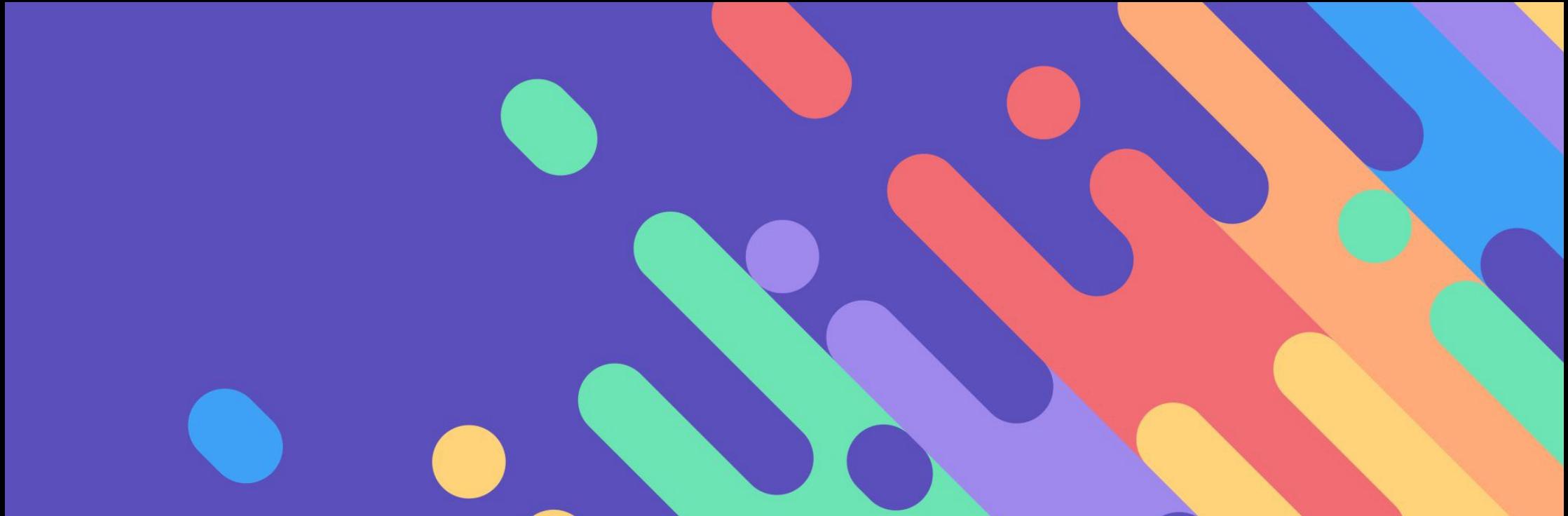
Límite = 1



Límite = 2



...



ALGORITMOS DE BÚSQUEDA INFORMADA

ALGORITMOS DE BÚSQUEDA INFORMADA

Función heurística

Si tuviéramos alguna forma de saber que tan lejos estamos del objetivo, podríamos hacer una búsqueda más eficiente de la solución más que probar diferentes caminos. **El problema es que, para saber la distancia, debemos resolver el problema primero.**

Entonces, una forma que podemos resolver esto es usar una estimación que llamamos **función heurística**:

$$h(n) = \text{costo estimado del camino más barato del estado del nodo } n \text{ al estado objetivo}$$



ALGORITMOS DE BÚSQUEDA INFORMADA

Función heurística

Para la torre de Hanoi podemos usar la siguiente función heurística:

$h(n)$ es un punto menos por cada disco ubicado en la posición correcta.

Por ejemplo, para un juego de 7 discos, si 5 de los 7 anillos están en la posición correcta de la varilla correcta, entonces $h(n) = -5$.

Si desean usar otro tipo de heurística, basada en conocimiento previo (cercano a sistemas expertos), recomiendo:

[Garcia & Chávez R - Heuristic function in an algorithm of First-Best search for the problem of Tower of Hanoi: optimal route for n disks.](#)

ALGORITMOS DE BÚSQUEDA INFORMADA

Búsqueda voraz (greedy) primero el mejor

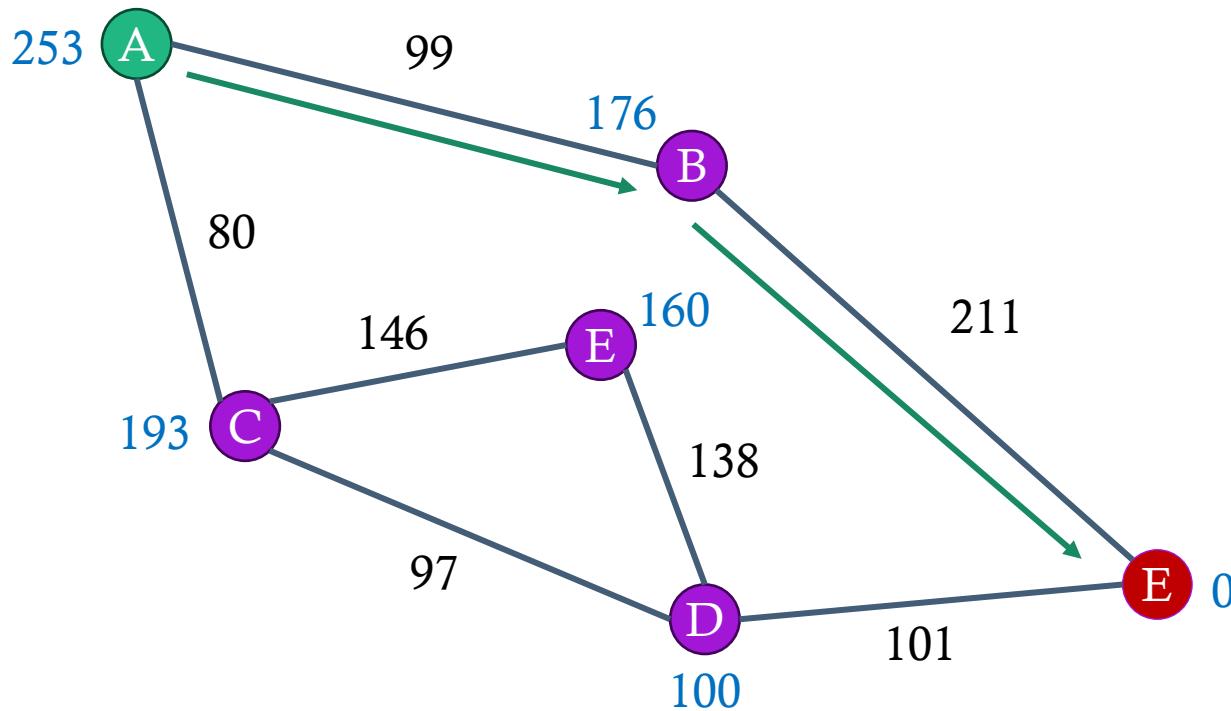
Esta búsqueda trata de expandir el nodo **con el valor más bajo de $h(n)$** (el nodo que parece más cerca del objetivo), alegando que probablemente conduzca rápidamente a una solución.

Esta estrategia no siempre asegura que se encuentre el mejor camino, pero nos permite llegar más rápido a la solución que las búsquedas no informadas.

```
# Pseudo-código
def greedy_best_first_graph_search(problem, heuristic_func):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = PriorityQueue(node, f=heuristic_func)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.append(s)
                frontera.append(child)
            elif child in frontera:
                if heuristic_func(child) < frontera[child]:
                    del frontera[child]
                    frontera.append(child)
    return 'failure'
```

ALGORITMOS DE BÚSQUEDA INFORMADA

Búsqueda voraz (greedy) primero el mejor



```
# Pseudo-código
def greedy_best_first_graph_search(problem, heuristic_func):
    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = PriorityQueue(node, f=heuristic_func)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.append(s)
                frontera.append(child)
            elif child in frontera:
                if heuristic_func(child) < frontera[child]:
                    del frontera[child]
                    frontera.append(child)
    return 'failure'
```

ALGORITMOS DE BÚSQUEDA INFORMADA

Búsqueda A*

Esta búsqueda no solo usa la función heurística, sino también utiliza el costo del camino tomado para llegar al nodo:

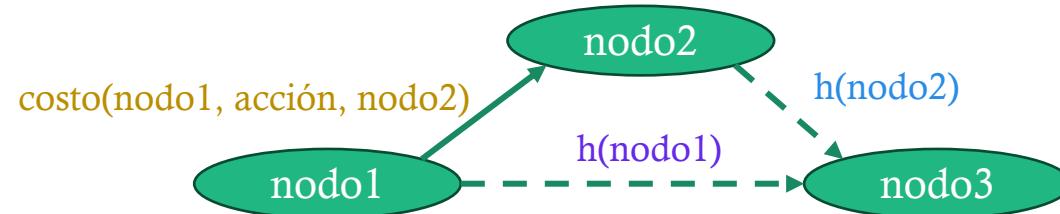
$$f(\text{nodo}) = \text{costo}(\text{nodo}) + h(\text{nodo})$$

Si todos los costos son >0 , se asegura que la búsqueda es **completa**.

Encontrar la solución más eficiente depende de si la función heurística *nunca sobreestima el costo de llegar al resultado (admisible)*.

Además, si la función heurística es **consistente**:

$$h(\text{nodo1}) < \text{costo}(\text{nodo1}, \text{acción}, \text{nodo2}) + h(\text{nodo2})$$



Entonces A* cada nodo al que llegue siempre va a ser el camino más óptimo.

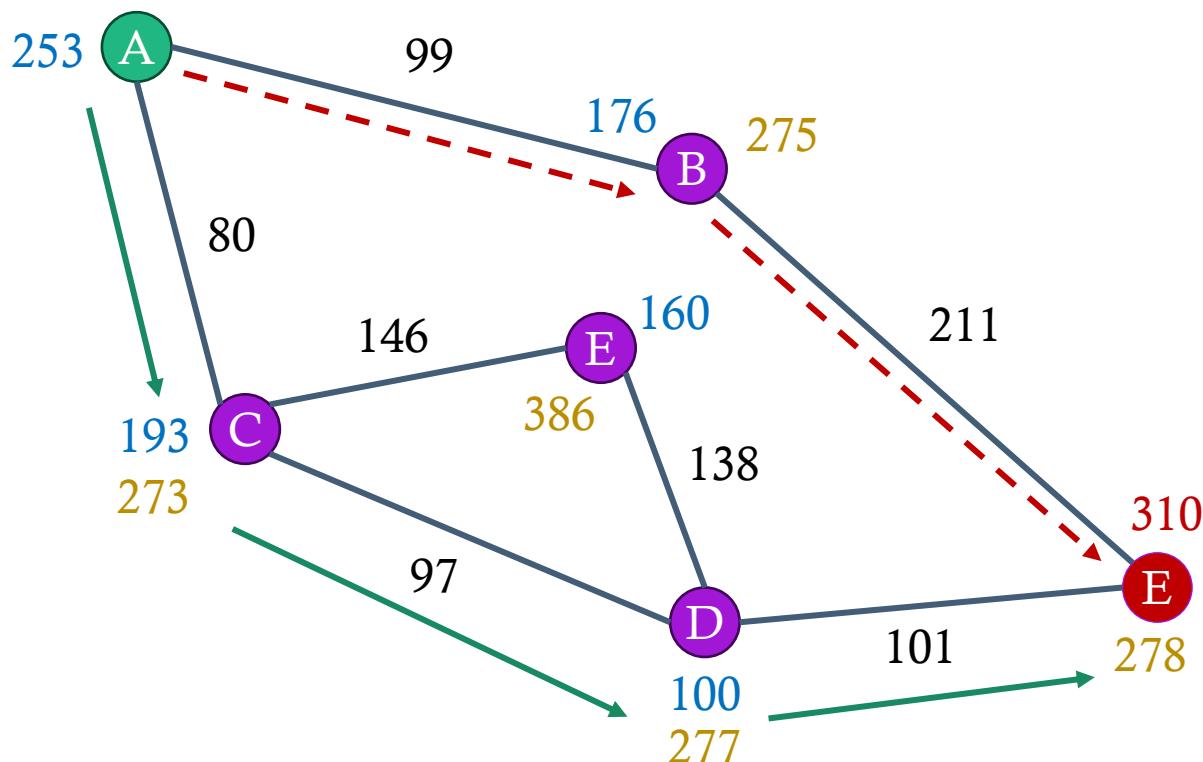
```
# Pseudo-código
def astar_search(problem, heuristic_func):

    def f(new_node):
        return new_node.path_cost + heuristic_func(new_node)

    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = PriorityQueue(node, f=f)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.append(s)
                frontera.append(child)
            elif child in frontera:
                if f(child) < frontera[child]:
                    del frontera[child]
                    frontera.append(child)
    return 'failure'
```

ALGORITMOS DE BÚSQUEDA INFORMADA

Búsqueda A*



```
# Pseudo-código
def astar_search(problem, heuristic_func):

    def f(new_node):
        return new_node.path_cost + heuristic_func(new_node)

    node = Node(problem.initial)
    if problem.is_goal(node.state):
        return node
    frontera = PriorityQueue(node, f=f)
    reached = []
    while not frontera.empty():
        node = frontera.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.append(s)
                frontera.append(child)
            elif child in frontera:
                if f(child) < frontera[child]:
                    del frontera[child]
                    frontera.append(child)
    return 'failure'
```