

# Inteligencia Artificial

Víctor Mijangos de la Cruz

## III. Solución de problemas mediante búsqueda



# Agentes para resolver problemas

# Problemas de búsqueda

Algunos problemas que surgen en el “mundo real” son del tipo:

**Problema para encontrar rutas:** Son problemas donde se busca encontrar las rutas óptimas para moverse de un lugar a otro. Un **problema de Turing** es un problema de ruta donde se busca pasar por puntos específicos durante el trayecto. El **problema del viajante de comercio** (PVC) busca pasar por todos los lugares de la manera más eficiente.

**Distribución VLSI:** Busca acomodar componentes en un chip, minimizando el área que se ocupa.

**Ensamblaje automático:** Problemas que quieren encontrar la forma más óptima de ordenar los objetos para ensamblar. Un ejemplo es el diseño de proteínas.

Estos problemas requieren de una **planificación**, y pueden verse como problemas de búsqueda.

# Agente para resolución de problemas

## Planeación

Una planeación considera una secuencia de acciones que permitan llegar a un objetivo dado.

## Agente para resolución de problemas

Un agente para resolución de problemas es un agente basado en objetivo que realiza una planeación para cumplir con los objetivos dados.

Los agentes para resolución de problemas se basan en la **búsqueda**. Se tienen dos tipos:

**Búsqueda informada:** El agente puede estimar que tan lejos se encuentra del objetivo.

**Búsqueda desinformada:** El agente no es capaz de estimar su distancia del objetivo.

# Solución de problemas

Dada información acerca del mundo, un agente se basará en los siguientes pasos para solucionar un problema:

- Formulación del objetivo: ¿qué meta se está buscando? Determinar **estados objetivo**.
- Formulación del problema: Descripción de **estados** y las **acciones** para alcanzar un objetivo (abstracción del mundo).
- Búsqueda: Simula secuencia de acciones hasta encontrar una que llegué al objetivo. A estas secuencias se les llama **soluciones**.
- Ejecución: Se ejecutan las acciones de la solución en el mundo.

# Sistemas de ciclos

Según si el agente tiene información o no del mundo al ejecutar la solución, se habla de dos tipos de sistemas:

**Sistemas de ciclo abierto:** (o Open-loop) Son aquellos en que, una vez que el agente ha encontrado una solución, la ejecuta ignorando las percepciones mientras lo hace.

**Sistemas de ciclo cerrado:** (o Closed-loop) Son aquellos en que el agente ejecuta la solución, pero mientras lo hace monitorea las percepciones que recibe.

# Búsqueda

## Problema de búsqueda

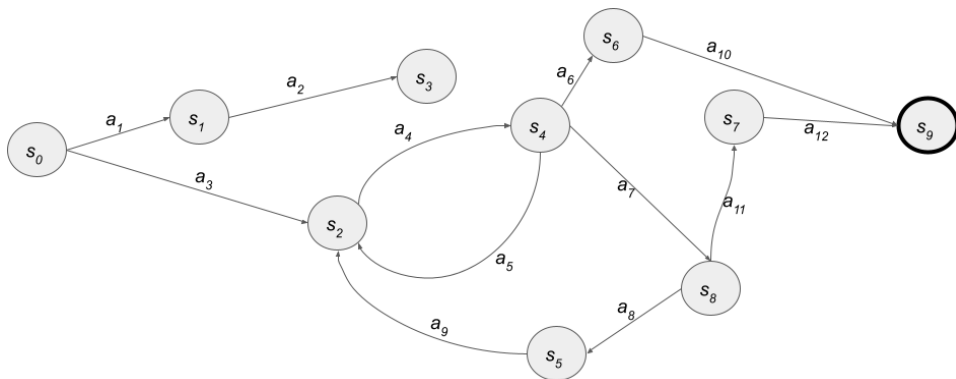
Un problema de búsqueda se puede definir por medio de la 6-tupla  $SP = (S, s_0, F, A, T, c)$ , donde:

- $S = s_0, s_1, \dots, s_T$  es un conjunto de estados en que el mundo puede encontrarse.
- $s_0 \in S$  es el estado inicial del agente.
- $F \subseteq S$  es un conjunto de estados objetivo.
- $A = a_1, \dots, a_N$  es un conjunto de acciones.
- $T: S \times A \rightarrow S$  es un modelo de transición.
- $c: S \times A \times S \rightarrow \mathbb{R}$  es una función de costo.

# Representación de problema de búsqueda

La representación del problema de búsqueda se hace mediante una gráfica:

$G = (V, E)$ , tal que  $V \cong S$  y  $E \cong A$ . La función  $c$  puede verse como una función de peso sobre las aristas  $E$ .





# Soluciones

## Camino

Un camino es una secuencia de acciones  $a_{i_0}, a_{i_2}, \dots a_{i_m} \subseteq A$ . Determina un camino en la gráfica.

## Solución

Una solución es un camino  $a_{i_0}, a_{i_2}, \dots a_{i_m} \subseteq A$  que cumple que  $T(s_{i_m}, a_{i_m}) \in F$ ; es decir que termina en un objetivo.

## Solución óptima

Una solución óptima es una solución que minimiza la función de costo sobre el camino; es decir, es una solución a la función objetivo:

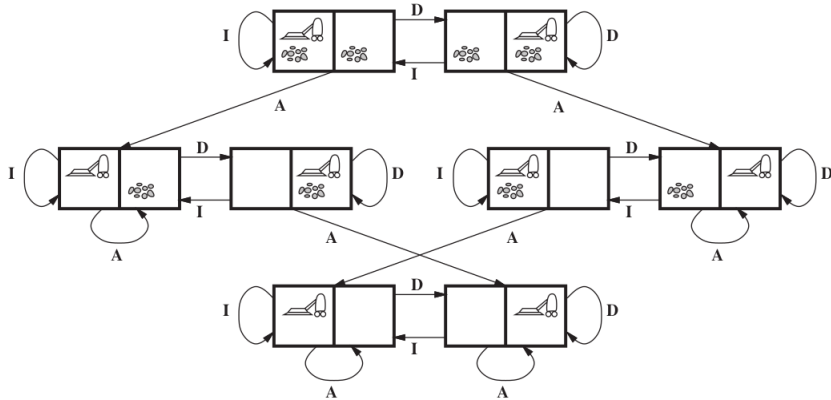
$$\arg \min_{a_{i_0}, a_{i_2}, \dots a_{i_m}} \sum_{t=0}^m c(s_{i_t}, a_{i_t}, s_{i_{t+1}}) \quad (1)$$

# Mundo de la aspiradora

El mundo de la aspiradora puede modelarse como un problema de búsqueda de la siguiente forma:

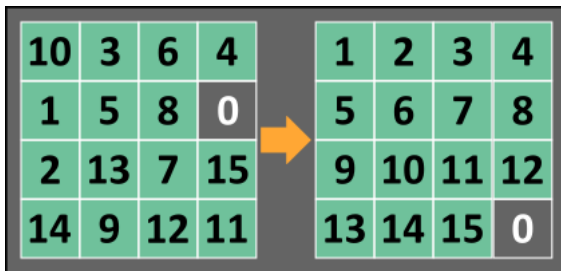
- Estados: Depende del número de cuadrados,  $n$ ; en general,  $|S| = n \cdot 2^n$ .
- Estado inicial: Depende de donde inicie el agente (puede ser aleatorio).
- Estados finales: Los estados en donde todos los cuadrados estén limpios.
- Acciones: Responden a los movimientos que puede hacer el agente.
- Modelo de transición: Para cada estado, el resultado de la acción. Ejemplo:  
 $T((A, 1), \text{limpiar}) = (A, 0)$ .
- Función de costo: Pude determinarse como 1 para toda acción.

# Mundo de la aspiradora



# Rompecabezas deslizante

Un rompecabezas deslizante consiste de una cuadrícula de  $n \times n$  números que deben moverse para obtener una forma ordenada de estos números en la cuadrícula.



¿Cómo se puede modelar este rompecabezas como un problema de búsqueda?

# Problema de encontrar rutas

Encontrar rutas de un punto A a un punto B puede modelarse de la siguiente forma:

- Los **estados** son los lugares por los que se debe pasar.
- El **estado inicial** es el punto A donde nos encontramos.
- Los **estados finales** son los puntos a los que queremos llegar, en este caso el punto B.
- Las **acciones** son los desplazamientos que hacemos entre los lugares/estados.
- Un **modelo de transición** indica como nos movemos entre los diferentes lugares según cómo nos desplazamos.
- La **función de costo** puede consistir en las distancias o el tiempo de desplazamiento. Pero también puede incorporar información como los costos monetarios, etc.

# Algoritmos de búsqueda

# Algoritmos de búsqueda

## Algoritmo de búsqueda

Un algoritmo de búsqueda es un algoritmo que toma como entrada un problema de búsqueda y que devuelve una solución a este problema.

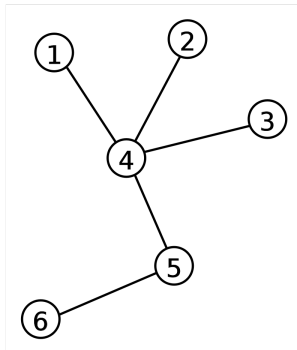
Muchos algoritmos de búsqueda se basan en buscar soluciones en una estructura de árbol, llamada **árbol de búsqueda**.



# Árboles

## Gráfica conectada

Una gráfica  $G = (V, E)$  está conectada si para todo  $u, v \in V$  existe un camino que inicia en  $u$  y termina en  $v$ .



## Ciclo

Dado una gráfica  $G = (V, E)$ , un ciclo es un camino  $v_0, v_1, \dots, v_n \subseteq V$  tal que  $v_0 = v_n$ .

## Árbol

Un árbol es una gráfica  $T = (V, E)$  conectada que no contiene ciclos.



# Árboles ordenados

## Árbol ordenado

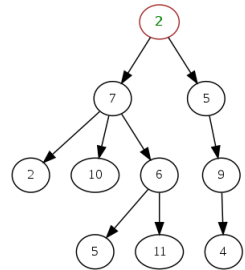
Un árbol ordenado  $T = (V, E, >)$  es un árbol en donde existe un orden,  $>$ , entre los nodos. Si  $u > v$ , decimos que  $u$  es **nodo padre** de  $v$  y que  $v$  es **nodo hijo** de  $u$ .

## Raíz

El nodo raíz de un árbol ordenado es el nodo  $v \in V$ , tal que no existe  $u \in V$  que cumpla  $u > v$ .

## Hojas

Un nodo  $v \in V$  de un árbol ordenado es una hoja si no existe  $u \in V$  tal que  $v > u$ .



# Árboles de búsqueda

## Árbol de búsqueda

Dado un problema de búsqueda  $SP = (S, s_0, F, T, c)$ , un árbol de búsqueda es un árbol ordenado  $T = (V, E, >)$  tal que  $S \cong V$ ,  $s_0 \cong v_0$  ( $v_0$  raíz),  $F \cong \{v \in V : v \text{ es hoja}\}$  y el orden  $>$  está determinado por  $T$ .

Un árbol de búsqueda describe los caminos entre un estado inicial (la raíz) y el objetivo (las hojas).

## Expansión de un nodo

Dado un nodo (estado) del árbol de búsqueda, se dice que se expande cuando se determinan sus nodos hijos a partir de la función de transición del problema de búsqueda.

# Búsqueda en árboles

## Búsqueda en árbol

Una búsqueda en el árbol de búsqueda consiste en determinar el nodo hijo más adecuado que expanda a un nodo padre.

Los nodos del árbol de búsqueda pueden representarse a partir de los siguientes elementos:

**Estado:** Estado  $s_i$  asociado al nodo.

**Padre:** Nodo padre del actual nodo.

**Acción:** La acción aplicada al padre para llegar al nodo actual.

**Costo:** Costo acumulado del camino desde la raíz al nodo actual.

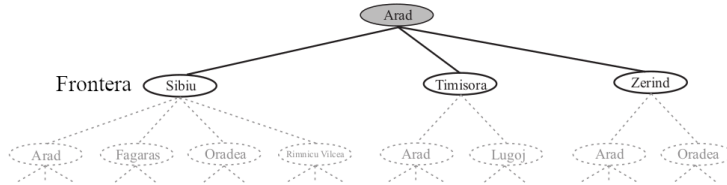
**Profundidad:** Número de aristas entre la raíz y el nodo.

# Búsqueda en árboles

## Frontera

La frontera de un árbol de búsqueda, en un estado dado, son los nodos que no se han expandido.

La frontera genera una partición entre nodos expandido, y nodos que todavía no se han alcanzado.



Si un nodo en la frontera es un objetivo, no se expandirá este nodo, resultando en una hoja.

# Estructura de nodos

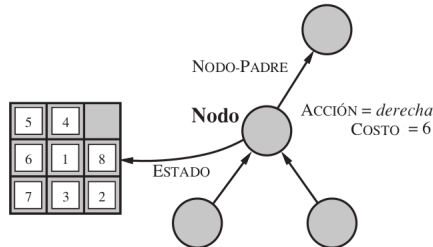
Los nodos de un árbol de búsqueda  $v \in V$  son estructuras abstractas. En la aplicación, tienen una estructura de datos particular que se compone de los siguientes **elementos**:

**STATE**: Estado (del problema de búsqueda) que corresponde al nodo.

**PARENT**: El nodo del que deriva (por medio de expansión).

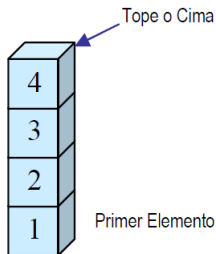
**ACTION**: La acción que derivo el nodo desde su padre.

**COST**: Costo del camino a partir del nodo inicial.



# Estructura de la frontera

La frontera es una **pila** que guarda información de los nodos obtenidos. Puede ser de los siguientes tipos:



**Pila de prioridad:** El tope de la pila es el elemento con menor valor de acuerdo a una función de costo.

**Pila FIFO:** (First-in-First Out) El tope de la cola es el primer elemento añadido.

**Pila LIFO:** (Last-in-First Out) El tope de la cola es el último elemento añadido.

# Estructura de la frontera

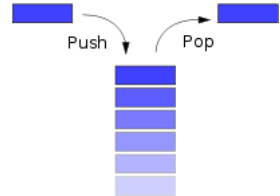
La frontera, en tanto pila, debe contar con las siguientes **operaciones**:

`isEmpty()`: Indica si la pila está o no vacía (booleano).

`Pop()`: Elimina y devuelve el elemento en el tope de la pila.

`Push(node)`: Añade un nuevo elemento a la pila.

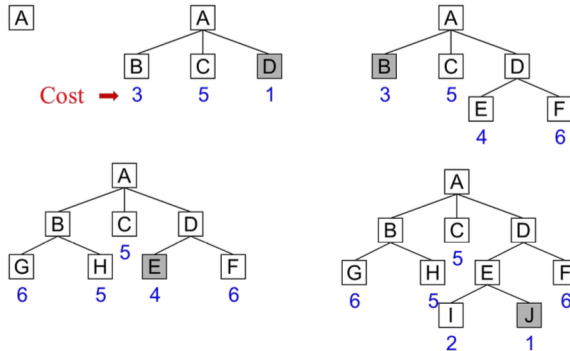
`Top()`: Devuelve el elemento en el tope de la pila.



# Algoritmo Best-First Search

El algoritmo de Best Search busca encontrar el camino más adecuado en un árbol de búsqueda, en base a una función de costo, que lleve a la solución de un problema.

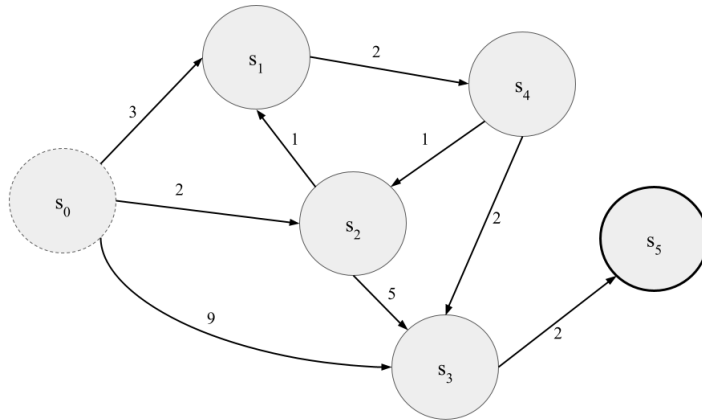
Se basa en una pila de **prioridad**, en base a la función de **costo**.





# Ejemplo: Best-First Search

En la siguiente gráfica el **estado inicial** es  $s_0$ , **final**  $s_1$  y el **costo** el peso de las aristas.

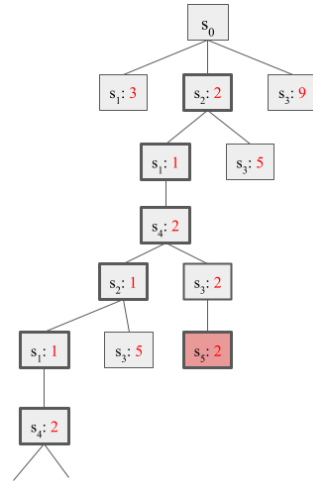


# Ejemplo: Best-First Search

La búsqueda de mejor primero o Best-First expande en primer lugar los nodos del árbol de búsqueda que tienen un menor valor.

En algunos casos, se expanden nodos que ya se han expandido, lo que puede ser problemático para converger rápidamente.

Una forma de mejorar estos elementos es observar que no hayan sido alcanzados estos elementos.



# Expansión de nodos

---

## Algorithm Función de expansión de nodos

---

```
1: procedure EXPAND(problem, node)
2:    $s \leftarrow \text{node.STATE}$ 
3:   for action in problem.ACTIONS do
4:      $s' \leftarrow \text{problem.TRANSITION}(s, \text{action})$ 
5:      $\text{cost} \leftarrow \text{node.COST} + \text{problem.c}(s, \text{action}, s')$ 
6:     yield NODE(STATE= $s$ , PARENT=node, ACTION=action, COST=cost)
7:   end for
8: end procedure
```

---

# Algoritmo Best-First Search

Dividimos el algoritmo en dos partes, la primera, la inicialización de los elementos y la segunda (WHILE), la parte central del método.

---

## Algorithm Algoritmo Best-First Search

---

```
1: procedure BESTFIRSTSEARCH(problem)
2:   node  $\leftarrow$  NODE(STATE=problem.INITIAL)
3:   frontier  $\leftarrow$  PRIORITYQUEUE.push(node)
4:   reached  $\leftarrow$  TABLE(problem.INITIAL : node)
5:   procedure WHILE
6:     :
7:   end procedure
8:   return failure
9: end procedure
```

---

# Algoritmo Best-First Search

---

## Algorithm Algoritmo Best-First Search

---

```
1: while frontier.isEmpty() = False do
2:   node ← frontier.pop()
3:   if node is problem.GOAL then
4:     return node
5:   end if
6:   for child in EXPAND(problem, node) do
7:     s ← child.STATE
8:     if s not in reached or child.COST < reached[s].COST then
9:       reachde[s] ← child and frontier.push(child)
10:    end if
11:  end for
12: end while
```

---

# Caminos redundantes

Uno de los problemas que debe evitarse en el diseño de algoritmos de búsquedas es el caer en loops.

## Estado repetido

Un estado repetido se da cuando la expansión de un nodo alcanza a un nodo que ya había sido alcanzado previamente. Esto genera un ciclo.

## Camino redundante

Un camino redundante es un ciclo en donde se dan uno o más caminos repetidos.

Los caminos redundantes pueden generar ciclos infinitos y son problemáticos para los algoritmos de búsqueda.

# Solución a caminos redundantes

Algunas aproximaciones para evitar los caminos redundantes son:

- Recordar todos los estados que se han alcanzado previamente. El problema es que esto puede requerir de mucha memoria.
- Dejar de la do la preocupación por el pasado, en algunos problemas es raro encontrar estados repetidos.
- Buscar ciclos en las gráficas generadas por los métodos.

## Búsqueda gráfica

Un algoritmo de búsqueda que considera los caminos redundantes se dice que es una búsqueda gráfica (*graph search*). Si no lo hace, se dice que es una búsqueda tipo árbol (*tree-like search*).

# Costo computacional

Algunos conceptos que deben tomarse a consideración par aun buen diseño de un algoritmo en IA son:

**Completo:** Un algoritmo de búsqueda se dice que es completo si podemos garantizar que puede encontrar la solución (cuando esta existe).

**Costo óptimo:** Refiere a encontrar la solución con el menor costo (profundidad) de entre todas las posibles soluciones.

**Complejidad en tiempo:** El tiempo que tarda el algoritmo en encontrar la solución.

**Complejidad en espacio:** La memoria necesaria para llevar a cabo la búsqueda.



# Consideraciones

Dependiendo de la estructura de datos, podemos medir la complejidad de diferentes formas:

**Estructura explícita:** Conocemos el comportamiento y estructura del problema. La complejidad en espacio se puede medir, entonces, como  $|V| + |E|$ .

**Estructura implícita:** Sólo conocemos el estado inicial, las acciones y el modelo de transición, pero no la estructura completa del problema. Varias alternativas:

- Medida de profundidad, las acciones necesarias para encontrar una solución óptima.
- El máximo número de acciones que se pueden dar en un camino.
- El factor de ramificación, el número máximo de sucesores de cualquier nodo.

# Búsqueda desinformada

# Búsqueda desinformada

## Algoritmo de búsqueda desinformada

Un algoritmo de búsqueda se dice que es algoritmo de búsqueda desinformada si, estando en un estado, no cuenta con información de la cercanía de este estado con un estado final o meta.

Ante la falta de información, un algoritmo desinformado:

- Encuentra soluciones de forma más lenta.
- Tiene mayor costo y consume mayor tiempo.
- Su implementación es más larga

# Estrategias de prueba de metas

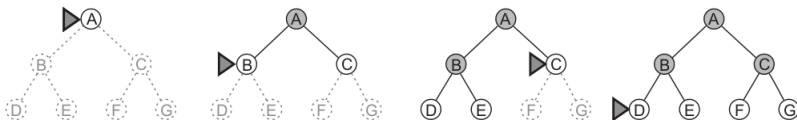
Tenemos dos estrategias para revisar si un nodo es un nodo meta o no:

**Early Goal Test:** Se revisa si un nodo es una solución tan pronto como es generado (por una expansión).

**Late Goal Test:** Se espera a revisar si el nodo es una meta cuando el nodo se ha eliminado (pop) de la pila.

# Algoritmo Breadth-First

El algoritmo Breadth-First expande los nodos de modo jerárquico, comenzando por el nodo raíz, expandiendo después sus hijos y así sucesivamente.



Este algoritmo utiliza una **pila FIFO**, y es útil cuando las acciones tienen siempre el mismo costo.

# Algoritmo Breadth-First Search

---

## Algorithm Algoritmo Breadth-First Search

---

```
1: procedure BREADTH-FIRST-SEARCH(problem)
2:   node  $\leftarrow$  NODE(STATE=problem.INITIAL)
3:   if node is problem.GOAL then
4:     return node
5:   end if
6:   frontier  $\leftarrow$  FIFOQUEUE.push(node)
7:   reached  $\leftarrow$  TABLE(problem.INITIAL : node)
8:   while do
9:      $\vdots$ 
10:  end while
11:  return Failure
12: end procedure
```

---

# Algoritmo Breadth-First Search

---

## Algorithm Algoritmo Breadth-First Search

---

```
1: while frontier.isEmpty() = False do  
2:   node ← frontier.pop()  
3:   for child in EXPAND(problem, node) do  
4:     s ← child.STATE  
5:     if s is problem.GOAL then  
6:       return child  
7:     end if  
8:     if s not in reached then  
9:       reachde[s] ← child and frontier.push(child)  
10:    end if  
11:  end for  
12: end while
```

---

# Algoritmo Breadth-First Search

El algoritmo de Breadth-First Search siempre encuentra una solución con el mínimo número de acciones. *En un nodo de profundidad  $d$ , se han revisado los  $d - 1$  nodos anteriores, si alguno de ellos es solución, ya ha sido encontrado.*

Su complejidad es  $O(b^d)$ , donde  $b$  es el número máximo de hijos de cada nodo, y  $d$  es la profundidad de la solución.



# Algoritmo de Dijkstra



## Algoritmo de Dijkstra

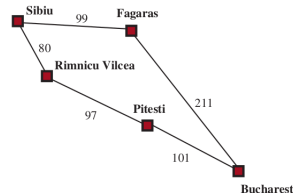
El algoritmo de Dijkstra o búsqueda de costo uniforme consiste en encontrar el camino más corto (de menor costo) entre un nodo (estado inicial) y cada arista.

El algoritmo de Dijkstra puede programarse a través del algoritmo de Best-First Search, utilizando como función de costo (en la pila) el **costo del camino** del nodo.

# Algoritmo de Dijkstra

Algunas de las propiedades del algoritmo de Dijkstra son:

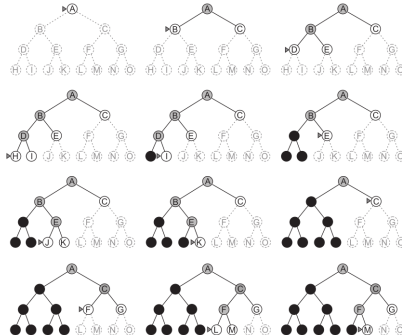
- El algoritmo de Dijkstra es completo y de costo óptimo.
- Su complejidad es  $O(b^{1+\lceil C^+/\epsilon \rceil})$ , donde  $b$  es el factor de ramificación,  $C^*$  es el valor óptimo del costo y  $\epsilon > 0$  es la cota inferior del costo de las acciones.



# Algoritmo Depth-First Search

## Algoritmo Depth-First Search

El algoritmo Depth-First Search (o primero en profundidad) se basa en el principio de expandir en primer lugar el nodo de mayor profundidad en la frontera.



# Depth-First Search

Algunas propiedades del algoritmo de Depth-First Search son:

- Su implementación es tipo árbol, **no** guarda una tabla de los nodos alcanzados.
- Puede pensarse como Best-First Search con función de profundidad negativa.
- Complejidad espacial es  $O(bm)$ ,  $b$  factor de ramificación y  $m$  máxima profundidad del árbol.
- No es de costo óptimo, devuelve la primera solución que encuentra.
- Sólo es completo cuando el espacio de estados del problema son árboles.

# Depth-Limited Search

El método de Depth-First Search puede atorarse en ciclos infinitos al no guardar los nodos alcanzados, y sólo es útil cuando un método de búsqueda tipo árbol es feasible.

Una alternativa es el algoritmo de Depth-Limited Search:

## Depth-Limited Search

El algoritmo de Depth-Limited Search (o de profundidad limitada) es una versión del Depth-First Search en el que se impone un límite de profundidad  $l$ . Los nodos con profundidad mayor a  $l$  no son expandidos.

Su complejidad temporal es  $O(b^l)$ , y la complejidad espacial  $O(bl)$ ,  $b$  factor de ramificación.

# Depth-Limited Search

---

## Algorithm Algoritmo Depth-Limited Search

---

```
1: procedure DEPTH-LIMITED-SEARCH(problem, l)
2:   frontier  $\leftarrow$  LIFOQUEUE.push(NODE(STATE=problem.INITIAL))
3:   result  $\leftarrow$  failure
4:   while frontier.isEmpty() = False do node  $\leftarrow$  frontier.pop()
5:     if node is problem.GOAL then return node
6:     end if
7:     if DEPTH(node) > l then result  $\leftarrow$  cutoff
8:     end if
9:     if not IS-CYCLE(node) then
10:       for child in EXPAND(problem, node) do frontier.push(child)
11:       end for
12:     end if
13:   end while
14:   return result
```

# Depth-Limited Search

Un problema del algoritmo de Depth-Limited Search es que puede no encontrar una solución cuando ésta se encuentra más allá de la profundidad  $l$  definida.

Para solucionar esto, se puede proponer el **diámetro** de la gráfica del espacio de estados como profundidad  $l$ .

## Diámetro

El diámetro de una gráfica es la distancia máxima entre cualesquiera dos nodos. Esto es, el diámetro de  $G = (V, E)$  es:

$$\text{diam}(G) = \max\{d(u, v) : u, v \in V\}$$

Determinar el parámetro  $l$  requiere un conocimiento del problema, y es problemático cuando el espacio de estados es infinito.

# Iterative Deepening Search

Para solucionar el problema de no alcanzar una solución que se da con el algoritmo de Depth-Limited Search, se propone otro método que toma como base este algoritmo.

## Iterative Deepening Search

El algoritmo de Iterative Deepening Search (o de profundidad iterativa) se basa en el algoritmo de Depth-Limited Search, pero itera sobre la profundidad para encontrar una solución al problema.

Este método prueba con todos los valores de la profundidad  $l$  hasta encontrar uno adecuado.

Su complejidad espacial es de  $O(bd)$ ,  $b$  factor de ramificación y  $d$  profundidad de la solución.  
Su complejidad temporal es  $O(b^d)$ .



# Iterative Deepening Search

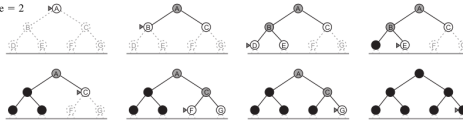
Límite = 0



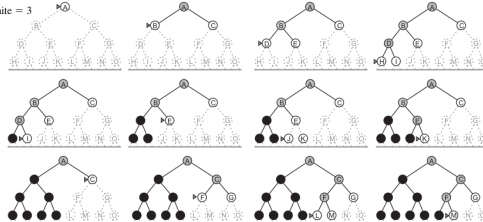
Límite = 1



Límite = 2



Límite = 3



# Iterative Deepening Search

---

## Algorithm Algoritmo Iterative Deepening Search

---

```
1: procedure ITERATIVE-DEEPENING-SEARCH(problem)
2:   for  $l = 0$  to  $\infty$  do
3:     result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem,  $l$ )
4:     if result  $\neq$  cutoff then
5:       return result
6:     end if
7:   end for
8: end procedure
```

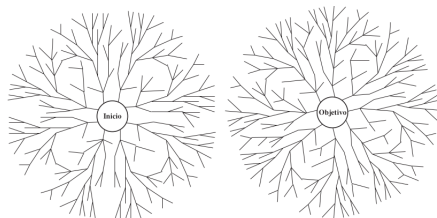
---

# Bidirectional Search

Los algoritmos de búsqueda que hemos revisado avanzan desde el estado inicial, buscando llegar a un estado final. Pero un sistema inverso es también posible.

## Búsqueda bidireccional

Una búsqueda bidireccional busca de manera simultánea en forma de avance (desde estado inicial al final) como en forma de retroceso (desde el estado final hasta el inicial). El objetivo es que las dos búsquedas coincidan.



# Bidirectional Search

En la búsqueda bidireccional se definen dos problemas:

**Problema de avance:** El problema típico que define la búsqueda del estado inicial al final.

**Problema de retroceso:** Problema inverso que va del estado final al inicial. Si  $s$  y  $s'$  es un padre y un hijo en el árbol de avance, entonces  $s'$  será padre y  $s$  hijo en el problema de retroceso.

Las características de la búsqueda bidireccional son:

- Es completo y de costo óptimo.
- Su complejidad (tanto temporal como espacial) es  $O(b^{d/2})$ ,  $b$  factor de ramificación,  $d$  profundidad de la solución.

# Bidirectional Search

```
1: procedure PROCEED(dir, problem, frontier, reached, reached2, solution)
2:   node ← frontier.POP()
3:   for child in EXPAND(problem, node) do
4:     s ← child.STATE
5:     if s not in reached or PATH-COST(child) < PATH-COST(reached[s]) then
6:       reached[s] ← child and frontier.push(child)
7:       if s in reached2 then
8:         solution2 ← JOIN-NODES(dir, child, reached2[s])
9:       end if
10:      if PAST-COTS(solution2) < PAST-COTS(solution) then
11:        solution ← solution2
12:      end if
13:    end if
14:  end for
15:  return solution
16: end procedure
```

# Bidirectional Search

---

## Algorithm Algoritmo de Bidirectional Search

---

```
1: procedure BIBF(problemF,  $f_F$ , problemB,  $f_B$ )
2:   nodeF  $\leftarrow$  NODE(problemF.INITIAL); nodeB  $\leftarrow$  NODE(problemB.INITIAL)
3:   frontierF  $\leftarrow$  PRIORITYQUEUE.push(nodeF,  $f_F$ ); frontierB  $\leftarrow$  PRIORITYQUEUE.push(nodeB,  $f_B$ )
4:   reachedF  $\leftarrow$  nodeF.STATE; nodeF; reachedB  $\leftarrow$  nodeB.STATE; nodeB
5:   solution  $\leftarrow$  failure
6:   while TERMINATE(solution, frontierF, frontierB) do
7:     if  $f_F$ (frontierF.TOP) <  $f_B$ (frontierB.TOP) then
8:       solution  $\leftarrow$  PROCEED(F, problemF, frontierF, reachedF, reachedB, solution)
9:     else solution  $\leftarrow$  PROCEED(B, problemB, frontierB, reachedB, reachedF, solution)
10:    end if
11:  end while
12:  return solution
13: end procedure
```

---

# Comparación algoritmos de búsqueda desinformada

Algoritmo	¿Es completo?	¿Costo óptimo?	Tiempo	Espacio
Breadth-First Search	Sí	Sí	$O(b^d)$	$O(b^d)$
Dijkstra	Sí	Sí	$O(b^{1+\lfloor C^+/\epsilon \rfloor})$	$O(b^{1+\lfloor C^+/\epsilon \rfloor})$
Depth-First Search	No	No	$O(b^m)$	$O(bm)$
Depth-Limited Search	No	No	$O(b^l)$	$O(bl)$
Iterative Deepening	Sí	Sí	$O(b^d)$	$O(bd)$
Bidirectional	Sí	Sí	$O(b^{d/2})$	$O(b^{d/2})$

# Búsqueda informada (heurística)



# Búsqueda informada

La búsqueda informada o heurística se basa en la integración de una función heurística, la cual se puede entender como:

## Función heurística

Una función  $h : V \rightarrow \mathbb{R}$  sobre los nodos de un árbol de búsqueda, es una función heurística si estima el costo del camino más corto entre un nodo  $n \in V$  y un nodo meta.

## Estrategia de búsqueda informada

Una estrategia de búsqueda informada utiliza información del dominio para conocer la locación de las metas. En particular, se basa en una función heurística.

# Greedy Best-First Search

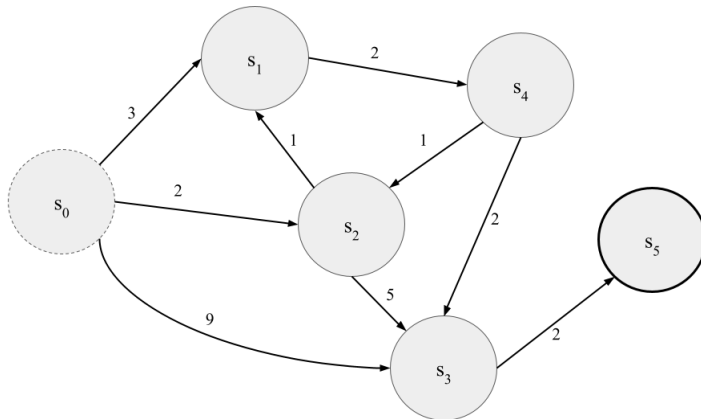
## Greedy Best-First Search

El algoritmo de Greedy Best-First Search expande en primer lugar el nodo  $n$  con el valor  $h(n)$  más pequeño. En este sentido:  $f(n) = h(n)$ , donde  $h$  es una función heurística.

El algoritmo de Greedy Best-First Search es **completo** si el espacio de estados es finito.

Su complejidad espacial y temporal es  $O(|V|)$ , donde  $V$  es el número de nodos.

# Ejemplo: Greedy-Best First Search



Podemos tomar una heurística del camino más corto de nodo actual al nodo meta:

<b>n</b>	<b><math>h(n)</math></b>
$s_0$	3
$s_1$	3
$s_2$	2
$s_3$	1
$s_4$	2
$s_5$	0

# Búsqueda A\*

## Búsqueda A\*

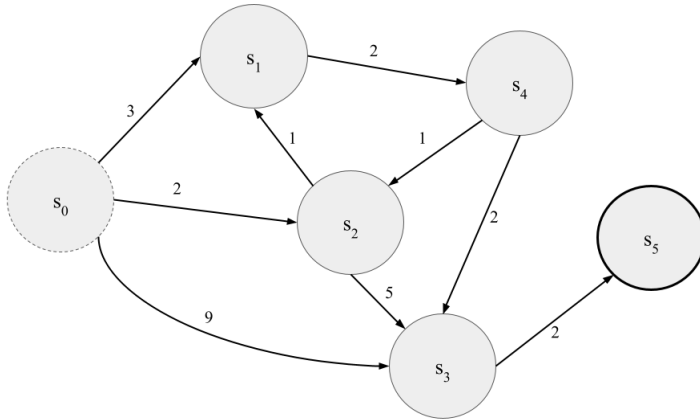
El algoritmo de búsqueda A\* (A-estrella) se basa en Best-First: expande el nodo,  $n$ , que minimiza la función  $f(n)$  definida como:

$$f(n) = g(n) + h(n)$$

donde  $g(n)$  es el costo del camino desde el estado inicial al nodo  $n$ , y  $h(n)$  es una función heurística que estima el costo del nodo  $n$  hacia un nodo final.

El algoritmo de A\* es **completo**. Pero el que sea de costo óptimo depende de las propiedades de la heurística.

# Ejemplo: A\*



Podemos tomar una heurística del camino más corto de nodo actual al nodo meta:

n	h(n)
s <sub>0</sub>	3
s <sub>1</sub>	3
s <sub>2</sub>	2
s <sub>3</sub>	1
s <sub>4</sub>	2
s <sub>5</sub>	0

# Heurística admisible

Un criterio que es importante para el algoritmo de  $A^*$  es el de heurística admisible:

## Heurística admisible

Una heurística admisible es una que nunca sobre-estima el costo de alcanzar una meta. Esto es,  $h$  es admisible si para todo  $n \in V$ ,

$$h(n) \leq h^*(n)$$

donde  $h^*$  es el costo óptimo.

**Ejemplo:** En el problema de encontrar la dirección de un punto a otro, la heurística de **la línea recta** entre el punto actual y la meta es una **heurística admisible**.

# Heurística admisible

## Proposición

Si  $h$  es una heurística admisible para  $A^*$ , entonces  $A^*$  es de costo óptimo.

Sea  $C^*$  es el costo del camino óptimo y  $C$  el costo actual. Supóngase que  $C^* < C$ , entonces existe  $n$  dentro del camino óptimo que no ha sido expandido. Tomemos el primer  $n$  que cumpla esto. Sean  $g^*(n)$  es el costo del camino óptimo desde el inicial hasta  $n$  y  $h^*(n)$  de  $n$  hacia la meta. Ya que hemos tomado el primer  $n$  del camino óptimo que no se ha expandido, entonces  $g^*(n) = g(n)$  y  $C = f(n) = g^*(n) + h(n)$  Pero como  $h$  es admisible, entonces

$$f(n) \leq g^*(n) + h^*(n) = C^*$$

lo que contradice la suposición inicial. Por tanto,  $C = C^*$  (costo óptimo).

# Consistencia

## Heurística consistente

Una heurística  $h$  es consistente si para todo nodo  $n$  y todo sucesor  $n'$  generado por la acción  $a$  se tiene que:

$$h(n) \leq c(n, a, n') + h(n')$$

Con la heurística consistente, aseguramos que cada vez que se alcance un nodo será por un camino óptimo.

La línea recta es también un ejemplo de heurística consistente.



# Consistencia

## Proposición

Si  $h$  es una heurística consistente, entonces  $h$  es una heurística admisible.

Inductivamente, sea  $n_{goal}$  nodo final, y  $n - 1$  el nodo anterior, entonces

$$h(n) \leq c(n - 1, a, n_{goal}) + h^*(n_{goal})$$

, pero  $h(n_{goal}) = h^*(n_{goal})$  y por tanto

$$c(n - 1, a, n_{goal}) + h^*(n_{goal}) = h^*(n - 1)$$

por lo que  $h(n - 1) \leq h^*(n - 1)$ .

Ahora si tomamos cualquier  $n < n_{goal}$ , tenemos  $h(n - 1) \leq c(n - 1, a, n) + h(n)$  y por hipótesis de inducción

$$c(n - 1, a, n) + h(n) \leq c(n - 1, a, n) + h^*(n)$$

De aquí que:  $h(n - 1) \leq c(n - 1, a, n) + h^*(n) = h^*(n - 1)$ .

# Heurística monótona y nodos ciertamente expandidos

## Heurística monótona

Una heurística  $h$  se dice que es monótona si  $h(n) \leq h(n')$  cuando  $n'$  es un nodo generado (hijo) de  $n$ .

Es claro que una heurística es monotónica si y sólo si es **consistente**.

## Nodos ciertamente expandidos

Un nodo es ciertamente expandido (*surely expande*) si puede ser alcanzado desde el estado inicial por medio de un camino tal que para todo nodo  $n$  en el camino se cumple  $f(n) < C^*$ .

DEL algoritmo de  $A^*$  **no expande** nodos tales que  $f(n) > C^*$ .

# Dominio

## Dominio

Dadas dos heurísticas  $h_1$  y  $h_2$  se dice que  $h_1$  **domina** a  $h_2$  si para todo nodo  $n$  se tiene que  $h_1 \geq h_2$ .

## Proposición

Si  $h_1$  domina a  $h_2$  entonces  $h_1$  es más eficiente ( $h_2$  expande al menos tanto nodos como  $h_1$ ).

Ya que todo nodo  $n$  con  $f(n) < C^*$  es candidato a un camino óptimo, para estos nodos, de  $f(n) = g(n) + h_1(n)$  se sigue que  $h_1(n) < C^* - g(n)$ . Si  $h_1$  es consistente, todo nodo que cumpla esta propiedad se expandirá, pero como  $h_1$  domina a  $h_2$  se tiene que  $h_2(n) \leq h_1 < C^* - g(n)$ ; es decir,  $h_2$  expande al menos el mismo número de nodos que  $h_1$ .

# Propiedades de $A^*$

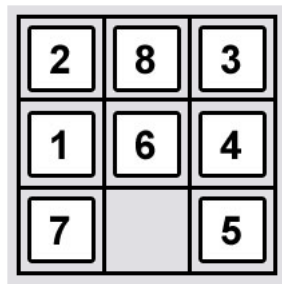
El algoritmo de  $A^*$  cumple:

- Es **óptimamente eficiente**: cualquier otro algoritmo que use la heurística  $h$  debe expandir todos los nodos que son **ciertamente expandidos** en  $A^*$ .
- Es **completo**: encuentra una solución si esta existe.
- Es **de costo óptimo**, cuando la heurística es admisible.

## Ejemplo: 8-puzzle

El juego 8-puzzle consiste en una cuadrícula de  $3 \times 3$  con números desde 1 a 8, y un espacio vacío.

El objetivo es ordenar los números en la cuadrícula a partir de moverlos hacia el espacio el blanco (mover el espacio en blanco).



2	8	3
1	6	4
7		5

# Ejemplo: 8-puzzle

Podemos proponer dos heurísticas para el juego de 8-puzzle:

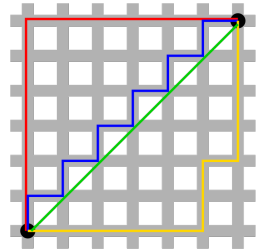
- Número de piezas desordenadas:

$$h_1(n) = \sum_{i=1}^9 \delta_{state_i \neq goal_i}$$

- Distancia Manhattan:

$$h_2(n) = \sum_{i=1}^9 |state_i - goal_i|$$

Ambas heurísticas son admisibles.



# Factor de ramificación eficaz

Una forma de caracterizar la **calidad de una heurística** es por el factor de ramificación eficaz:

## Factor de ramificación eficaz

Sean  $N$  el número de nodos en el árbol de búsqueda generado por  $A^*$  y  $d$  la profundidad de la solución, el factor de ramificación eficaz es el número  $b^*$  que es solución a:

$$N + 1 = \sum_{k=0}^d (b^*)^k$$

Por ejemplo si  $N = 52$  y  $d = 5$ , tenemos un factor de ramificación eficaz:

$$b^* = 1.92$$

# Factor de ramificación eficaz en 8-puzzle

Para los ejemplos de 6-puzzle, se han calculado los siguientes datos (Russel y Norvig, 2021):

<b>d</b>	<b>Nodos generados</b>		<b>Ramificación eficaz</b>	
	$A^*(h_1)$	$A^*(h_2)$	$A^*(h_1)$	$A^*(h_2)$
6	24	19	1.42	1.34
8	48	31	1.40	1.30
10	116	48	1.48	1.27
12	276	84	1.45	1.25
14	678	174	1.47	1.31





La heurística de la distancia Manhattan **domina** a la del número de piezas desordenadas.

## A\* pesado

### Solución satisfactoria

Una solución satisfactoria es aquella que, sin ser óptima, es lo 'suficientemente buena'-

Una forma de encontrar soluciones sub-óptimas y satisfactorias es a partir de pesar las heurísticas:

### A\* pesado

El algoritmo de  $A^*$  es una búsqueda  $A^*$  con la función de costo:

$$f(n) = g(n) + \alpha h(n)$$

para algún  $\alpha \geq 0$ .

$A^*$  encuentra una solución con costo entre  $C^*$  y  $\alpha C^*$ .

# A\* pesado

Variar el valor del peso  $\alpha$  en el método de A\* pesado permite obtener otros métodos de búsqueda:

- Si  $\alpha = 1$ , se obtiene el algoritmo de A\*:

$$f(n) = g(n) + h(n)$$

- Si  $\alpha = 0$ , se obtiene el método de costo uniforme:

$$f(n) = g(n)$$

- Si asumimos  $\alpha = \infty$ , tenemos una búsqueda voraz (greedy):

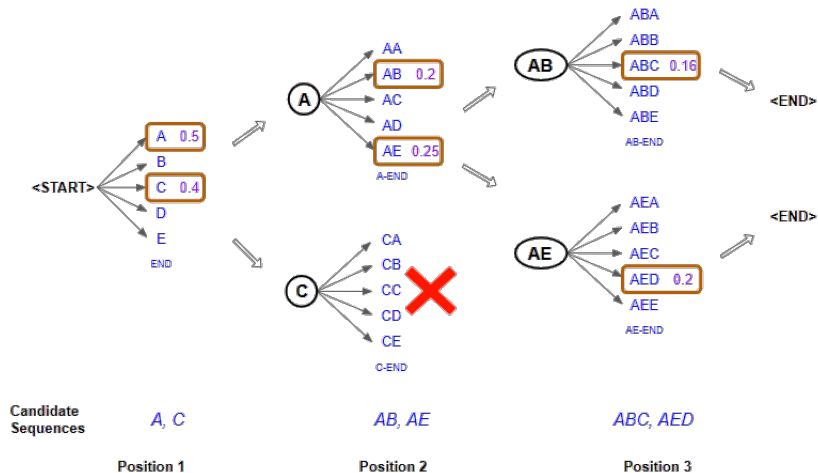
$$f(n) = h(n)$$

# Beam Search

Un método común para búsqueda con **memoria acotada** es el de **Beam Search** que puede resumirse como:

- 1 Seleccionar los  $k$  nodo menor costo dado el **inicial**, y guardar sólo esos nodos.
- 2 En cada nueva iteración se expande sólo los  $k$  nodos obtenidos en la iteración anterior. Los otros nodos se olvidan.
- 3 Se conservan sólo los  $k$  nodos con menor costo en cada iteración.
- 4 Se repite hasta alcanzar un estado meta.

# Beam Search



# Búsqueda bidireccional con heurística

En la búsqueda heurística bidireccional, debemos considerar pares de nodos para asegurar que encontramos un camino óptimo.

Defínanse los problemas de avance y retroceso en base a las funciones de costo, como:

- **Avance:**  $f_F(n) = g_F(n) + h_F(n)$
- **Retroceso:**  $f_B(n) = g_B(n) + h_B(n)$

Si  $m$  es nodo en el camino de avance y  $n$  en el de retroceso, una **cota inferior** está dada como:

$$lb(m, n) = \max\{g_F(m) + g_B(n), f_F(m), f_B(n)\}$$

# Búsqueda heurística bidireccional

## Teorema

Para todo par de nodos  $m, n$  con  $lb(m, n) < C^*$ , el camino que los atraviesa es una solución óptima potencial.

Esto es, se deben expandir los nodos que cumplan la condición  $lb(m, n) < C^*$ . Para tratar de encontrar estos nodos se propone la función:

$$f_2(n) = \max\{2g(n), g(n) + h(n)\}$$

Esta función definirá nuestro método de **búsqueda heurística bidireccional**.

# Determinación de funciones heurísticas

Las funciones heurísticas juegan un papel central en la **eficiencia** del método de búsqueda.

Una heurística es mejor que otra si esta **domina** a la segunda.

Se proponen diferentes métodos para poder obtener heurísticas adecuadas (admisibles, consistentes):

- A partir de **relajar** las restricciones de un problema.
- A partir de pre-calcular costos para conformar **base de datos de patrones**.
- A partir de definir **puntos de referencia**.
- A partir de la **experiencia**.



# Problemas relajados

## Problema relajado

Un problema relajado consiste en un planteamiento del problema original pero que considera menores restricciones.

Dado el grafo  $G = (V, E)$  del espacio de estados original, el **espacio de estados del problema relajado** es un **supergrafo** de  $G$  (i.e.  $G \leq G_{relaxed}$ ), puesto que  $E \subseteq E_{relaxed}$ .

Cualquier solución óptima del problema original, es una **solución** del problema relajado.

El **costo de una solución óptima** del problema relajado es una **heurística admisible** del problema original.

# Determinación de problemas relajados

Dada la definición de un problema planteada en **lenguaje formal** el problema relajado se determina a partir de **eliminar condiciones**:

$$X_0 \wedge X_1 \wedge \dots \wedge X_n \implies Y \mapsto X_{i_0} \wedge \dots \wedge X_{i_n} \implies Y$$

tal que  $X_{i_0}, \dots, X_{i_n} \subset X_0, \dots, X_n$ .

Por ejemplo en el  $n$ -puzzle, tenemos una condición de la forma:

*Si  $X$  y  $Y$  son adyacentes y  $Y$  es vacío,  $X$  puede moverse a  $Y$*

El cual se puede relajar como:

- Si  $X$  y  $Y$  son adyacentes,  $X$  puede moverse a  $Y$  (heurística: distancia Manhattan).
- $X$  puede moverse a  $Y$  (heurística: número de casillas mal colocadas).

# Base de datos de patrones

Una estrategia para eficientar el cálculo de heurísticas es considerar **subproblemas** del problema original y **guardar** las soluciones y costos en una base de datos.

## Heurística basada en bases de datos

Una heurística basada en base de datos consiste en considerar la solución de subproblemas (almacenadas en una base de datos) para calcular una heurística que tome en cuenta estos costos.

Por ejemplo, en el 8-puzzle, se pueden eliminar elementos para simplificar el espacio de estados:

*	2	4
*		*
*	3	1

Start State

	1	2
3	4	*
*	*	*

Goal State

# Puntos de referencia

Los **puntos de referencia** son nodos en el espacio de estados de los cuales se puede considerar un costo desde el estado inicial.

## Heurística diferencial

Dado un conjunto de puntos de referencia  $\mathcal{L}$  la heurística diferencial se define como:

$$h_{DH}(n) = \max_{L \in \mathcal{L}} |C^*(n, L) - C^*(goal, L)|$$

donde  $C^*(n, m)$  es el costo óptimo desde  $n$  hasta  $m$ .

La heurística diferencial es admisible y eficiente.

Idealmente, se prefiere una selección de puntos de referencia que estén **dispersos** sobre el espacio de estados.

# Heurísticas de la experiencia

Una forma de estimar una heurística es a partir de explorar los posibles resultados de las acciones y cómo afectan los estados. Es decir por **experiencia**.

Podemos representar los estados por **rasgos**. Sea  $x_i(n)$  un rasgo en el nodo  $n$  de la búsqueda. Una heurística se puede definir como:

$$h(n) = \sum_i w_i x_i(n)$$

Donde los valores  $w_i$  son **pesos** que determinan la relevancia de los rasgos.

Estos pesos se **aprenden** a partir de la experiencia. Esto determina lo que se conoce como **aprendizaje de máquina**.

# Resumen: Problemas de búsqueda

Consideramos los siguientes conceptos en **problemas de búsqueda**:

- Búsqueda desinformada: No cuenta con información de su distancia al objetivo.
- Búsqueda heurística: Cuenta con información de su distancia al objetivo.
- Árboles de búsqueda: Representación gráfica de la búsqueda de una solución del problema.
  - Estado inicial: nodo raíz del árbol, configuración inicial del problema.
  - Solución: Es el camino desde el estado inicial hacia la meta.
  - Función de costo: Determina el costo de una acción para avanzar a un nuevo estado.

Los problemas de búsqueda funcionan cuando el entorno de trabajo es: **totalmente observable, determinista, episódico y discreto**.

# Resumen: Algoritmos de búsqueda

Algunos de los **algoritmos de búsqueda** que hemos revisado son:

- **Algoritmos de búsqueda desinformada:**
  - Breadth-First Search
  - Algoritmo de Dijkstra
  - Depth-First Search
  - Depth-Limited Search, Iterative Deepening Search
  - Bidirectional Search
- **Algoritmos de búsqueda informada:**
  - Greedy Best-First Search
  - $A^*$
  - $A^*$  pesado
  - Beam Search
  - Bidirectional Heuristic Search

# Resumen: Heurísticas

Hemos considerado las **heurísticas**, que informan acerca de la distancia entre el estado actual y la meta.

Tenemos algunas formas para determinar heurísticas:

- Por problemas relajados: quitando restricciones al problema original.
- Por bases de datos: pre-computando soluciones para subproblemas en una base de datos de patrones.
- Por puntos de referencia: seleccionando puntos de referencia y determinando sus costos desde el estado actual.
- Por experiencia: por medio de métodos de aprendizaje de máquina.



# Material recomendado

**Py-Search:** Biblioteca para problemas de búsqueda en pytorch:

<https://pypi.org/project/py-search/>

Russle, S. y Norvig, P. (2021). 'Ch. 3. Solving Problems by Searching'. *Artificial Intelligence: A Modern Approach*. Pearson, pp. 81-128.

Edelkamp, S. y Schrödl, S. (2011). *Heuristic Search: Theory and Applications*. Elsevier.

Joshi, P. (2017). 'Ch. 7. Heuristic Search Techniques'. *Artificial Intelligence with Python*. Packt, pp. 172-196.