Inteligencia Artificial

SCALAB
Grupo de Inteligencia Artificial

Universidad Carlos III de Madrid

Curso 2019-2020

Contenido

Búsqueda

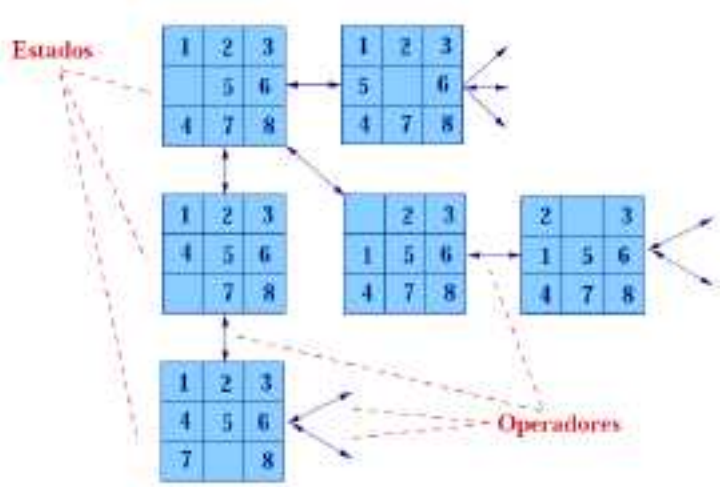
En este tema

Búsqueda

Introducción

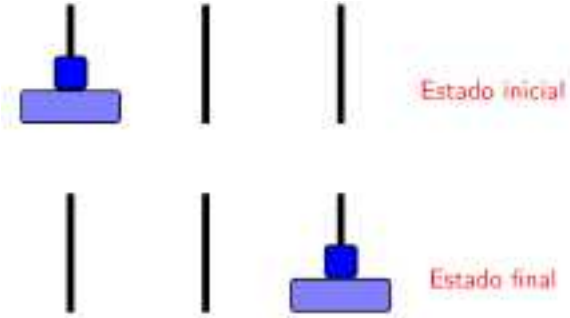
- Espacio de problemas
  - Un conjunto de estados
  - Un conjunto de operadores
  - Estado inicial
  - Metas o estado final
- Representación del problema en forma de grafo
- Resolución del problema = búsqueda en el grafo
- Normalmente, el proceso de búsqueda **genera un árbol**
- Parámetros importantes
  - **Factor de ramificación**, *b*
  - **Profundidad** del árbol de búsqueda, *d*

8-puzzle

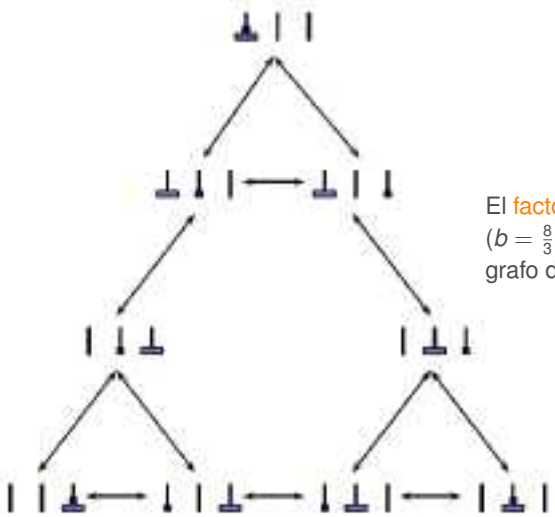


Torres de hanoi

Torres de hanoi (3,2)

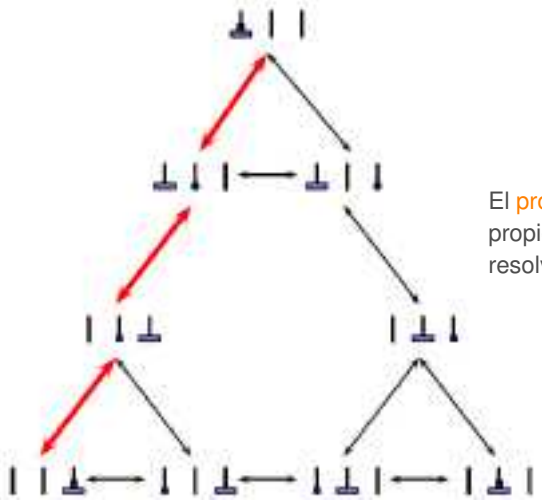


Torres de hanoi (3,2)



El factor de ramificación ($b = \frac{8}{3}$) es una propiedad del grafo de estados

Torres de hanoi (3,2)



El profundidad ($d = 3$) es una propiedad del problema a resolver

Explosión combinatoria

Dominio	Número de estados	Tiempo (10^7 nodos/s)
8-puzzle	$\left(\frac{N^2!}{2}\right)\Big\|_{N=3} = 181.440$	0.01 segundos
15-puzzle	$\left(\frac{N^2!}{2}\right)\Big\|_{N=4} = 10^{13}$	11,5 días
24-puzzle	$\left(\frac{N^2!}{2}\right)\Big\|_{N=5} = 10^{25}$	$31,7 \times 10^9$ años
Hanoi (3,2)	$(3^n)\big\|_{n=2} = 9$	9×10^{-7} segundos
Hanoi (3,4)	$(3^n)\big\|_{n=4} = 81$	$8,1 \times 10^{-6}$ segundos
Hanoi (3,8)	$(3^n)\big\|_{n=8} = 6561$	$6,5 \times 10^{-4}$ segundos
Hanoi (3,16)	$(3^n)\big\|_{n=16} = 4,3 \times 10^7$	4,3 segundos
Hanoi (3,24)	$(3^n)\big\|_{n=24} = 2,824 \times 10^{11}$	0,32 días
Cubo de Rubik $2 \times 2 \times 2$	10^6	0,1 segundos
Cubo de Rubik $3 \times 3 \times 3$	$4,32 \times 10^{19}$	31.000 años

Problema del camino más corto

Juegos. Civilization

Vehículos autónomos

Problema del viajante (TSP: *Travelling Salesman Problem*)

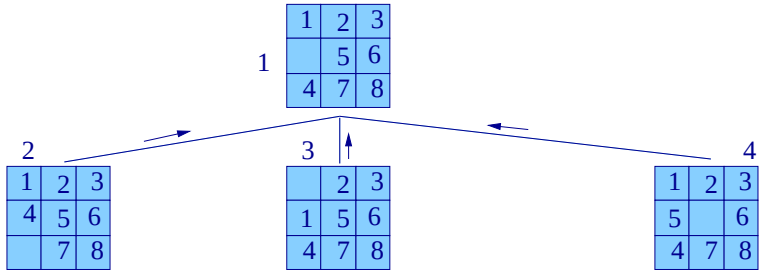
Curiosidades: http://www.wired.com/wiredscience/2013/01/traveling-salesman-problem/all/

Problemas de safisfacción de restricciones (CSP). N-reinas

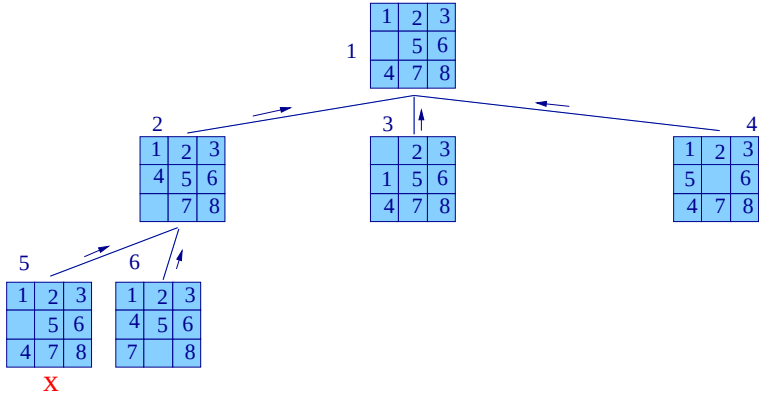
En este tema

En este tema

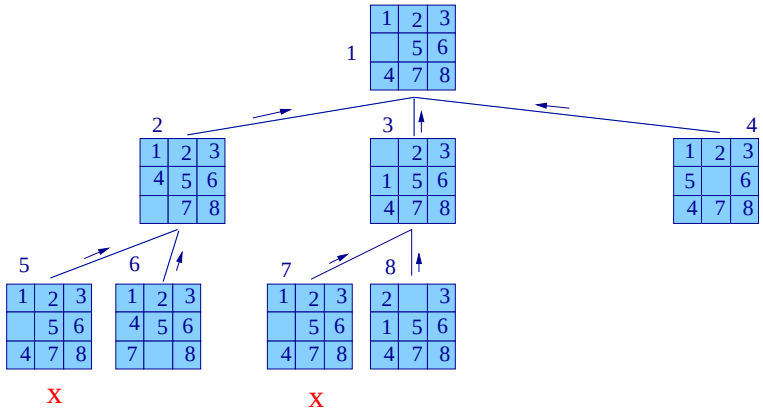
8-Puzzle – Amplitud



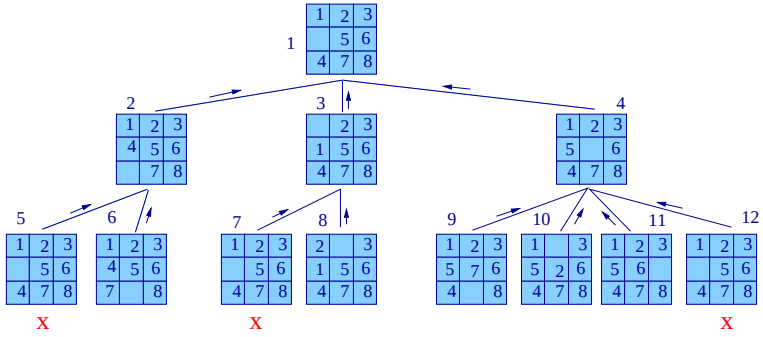
8-Puzzle – Amplitud



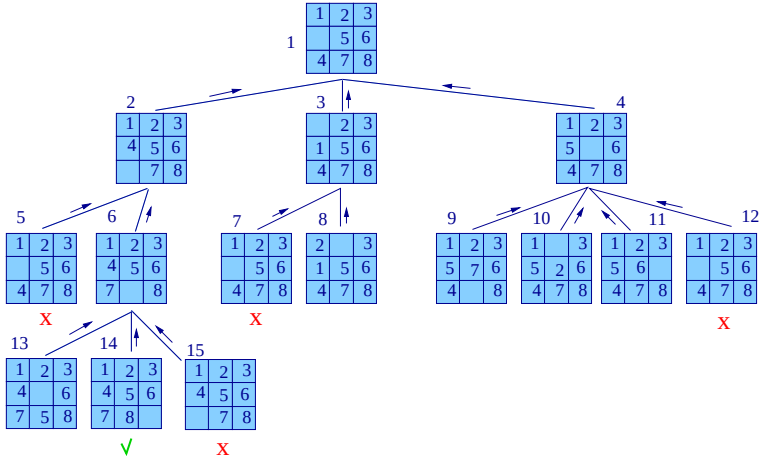
8-Puzzle – Amplitud



8-Puzzle – Amplitud



8-Puzzle – Amplitud

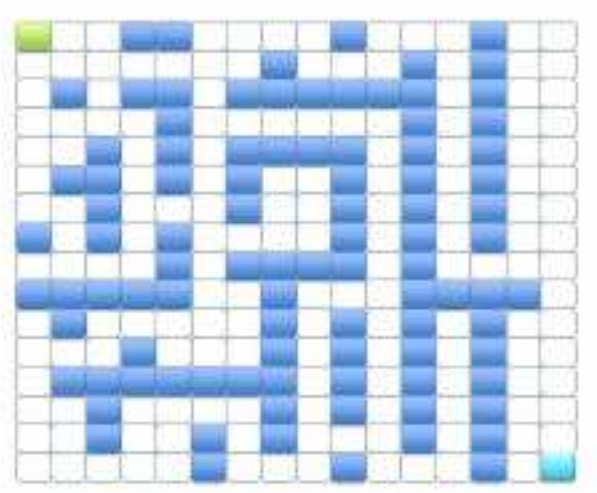


Búsqueda en amplitud

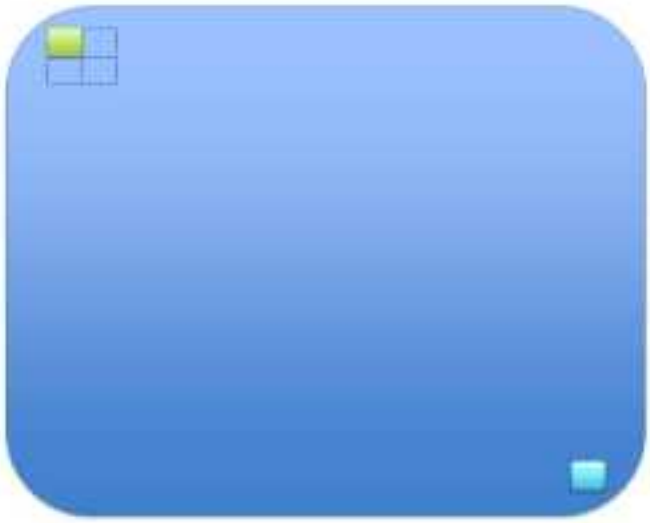
Amplitud (I: estado inicial, F: estado final)

```
Crear lista ABIERTA con el nodo inicial I (estado inicial);
EXITO = False;
while ABIERTA no esté vacía y no EXITO do
    Extraer el primer nodo N de ABIERTA;
    if N tiene sucesores then
        Generar los sucesores de N (expandir N) (eliminar repetidos?);
        Crear un puntero desde cada sucesor hasta N;
        if algún sucesor es un nodo meta then
            EXITO = True;
        else
            Añadir los sucesores de N al final de ABIERTA;
if EXITO then
    Solución = camino desde I hasta N siguiendo los punteros;
else
    Fracaso
```

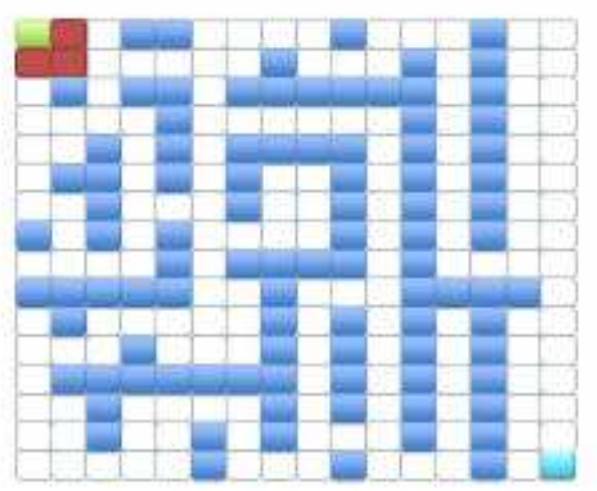
Laberinto



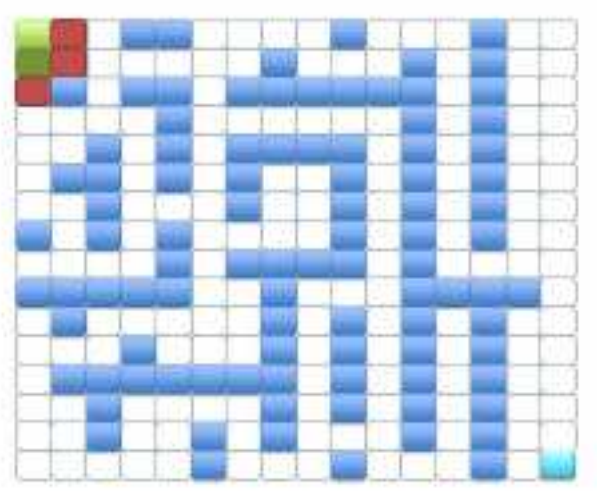
Laberinto



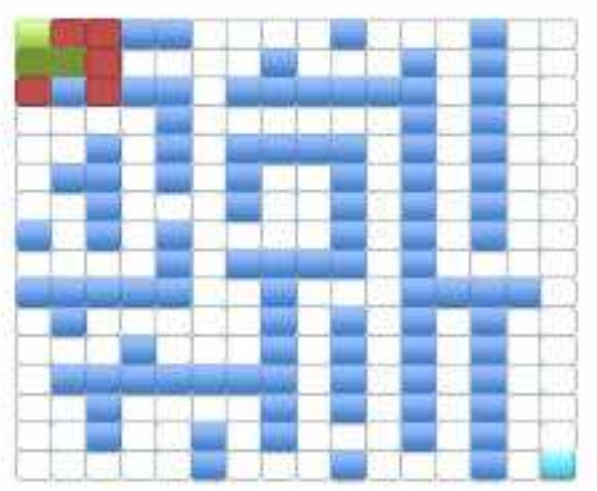
Laberinto



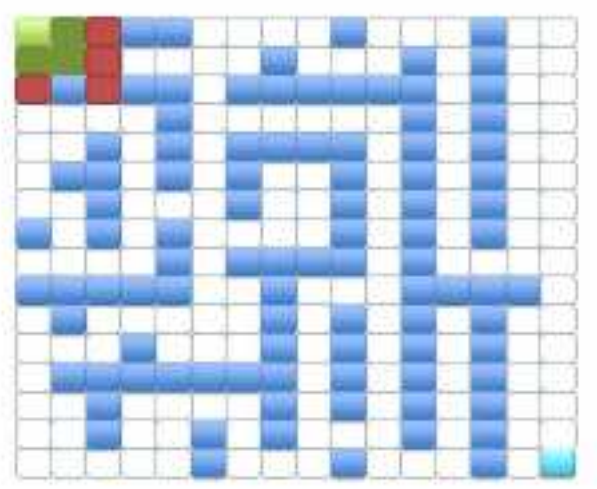
Laberinto



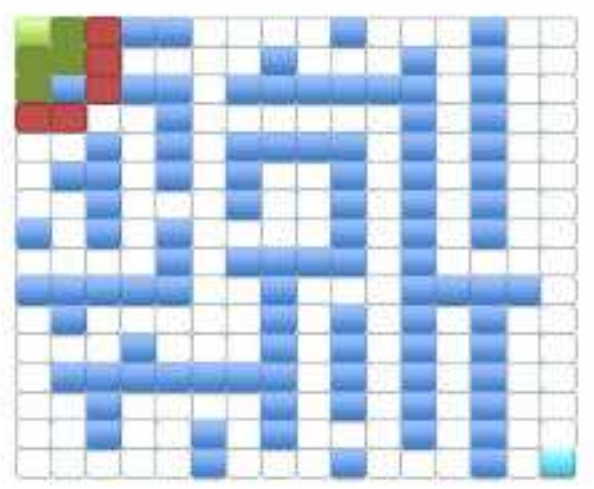
Laberinto



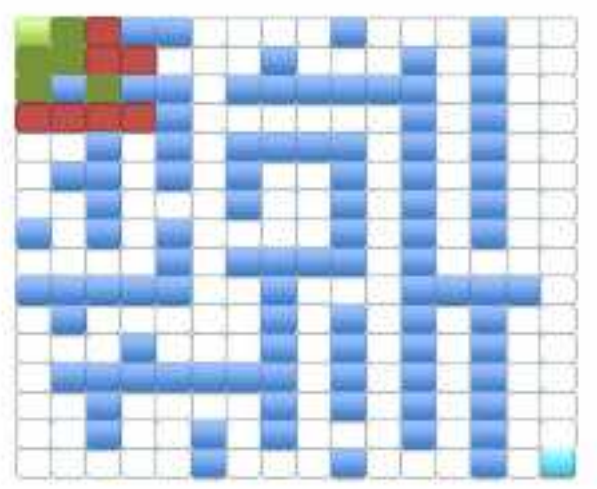
Laberinto



Laberinto



Laberinto

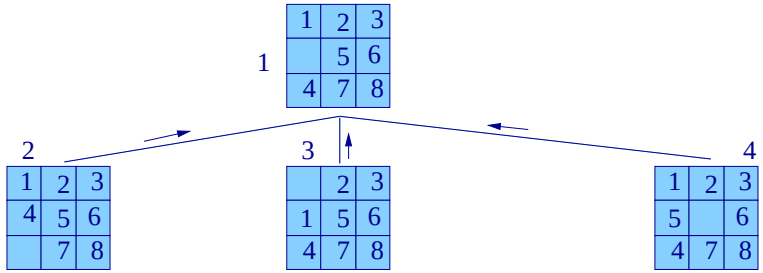


Características

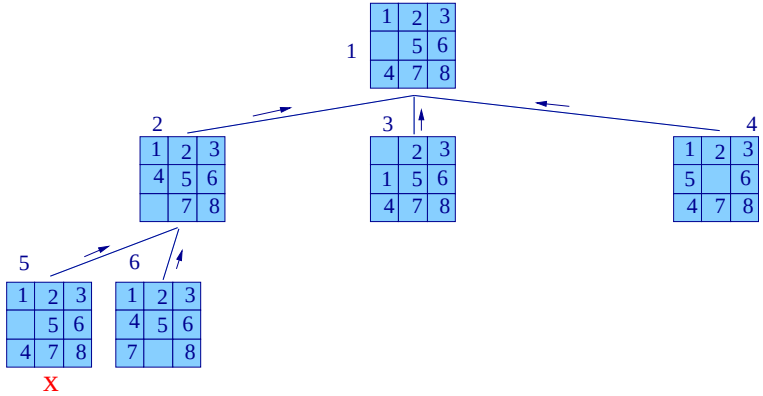
- **Completitud**: Si hay solución y el factor de ramificación es finito en cada nodo, encontrará la solución
- **Admisibilidad**: Si todos los nodos tienen el mismo coste, encontrará la solución óptima
- **Eficiencia**: buena si las metas están cerca del estado inicial
- **Problema**: consumo de memoria exponencial

En este tema

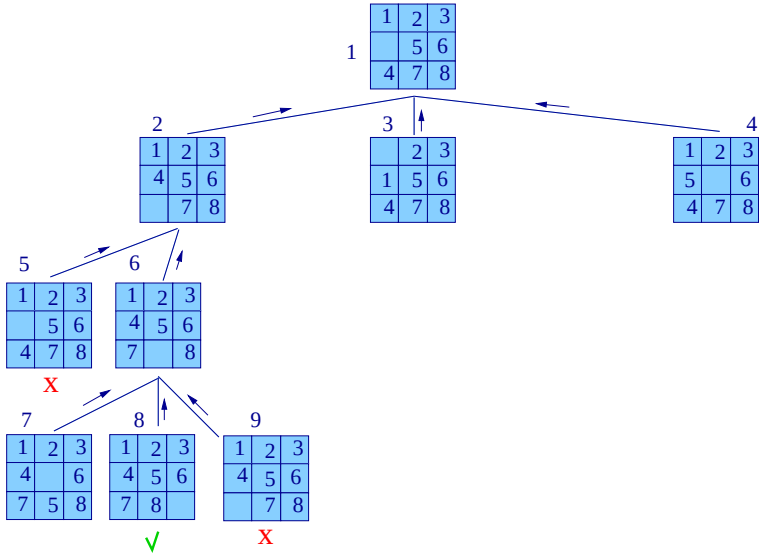
8-Puzzle – Profundidad



8-Puzzle – Profundidad



8-Puzzle – Profundidad

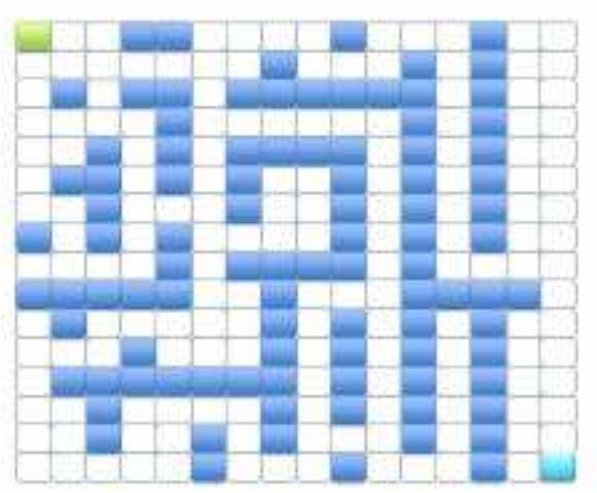


Búsqueda en profundidad

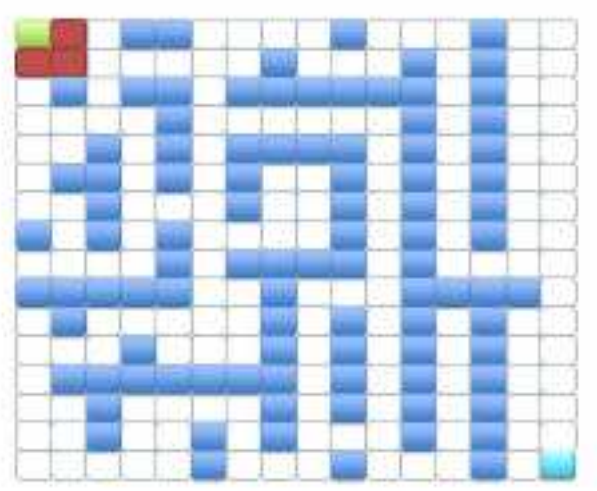
Profundidad (I: estado inicial, F: estado final, límite_profundidad)

Crear lista *ABIERTA* con el nodo inicial *I* (estado inicial), profundidad $d(N) = 0$;
EXITO = False;
while *ABIERTA no esté vacía y no EXITO* **do**
    Extraer el primer nodo *N* de *ABIERTA*;
    if $d(N) <$ *límite_profundidad y N tiene sucesores* **then**
        Generar los sucesores de *N* (expandir *N*) (evitar ciclos en el camino?);
        Crear un puntero desde cada sucesor hasta *N*;
        if *algún sucesor es un nodo meta* **then**
            EXITO = True;
        else
            Añadir los sucesores de *N* con d(suc) = d(N)+1 al principio de *ABIERTA*;
if *EXITO* **then**
    Solución = camino desde I hasta N siguiendo los punteros;
else
    Fracaso

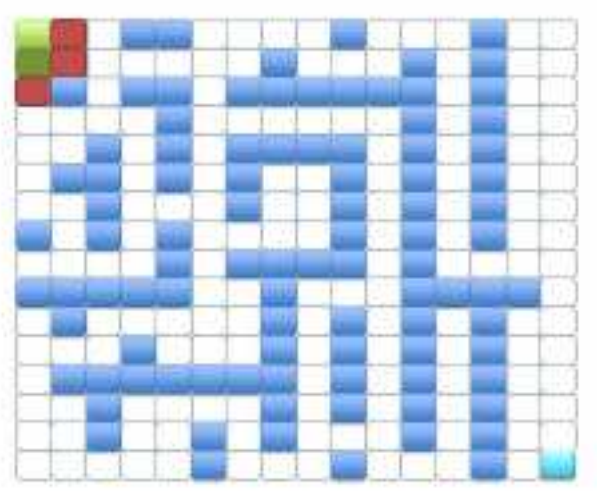
Laberinto



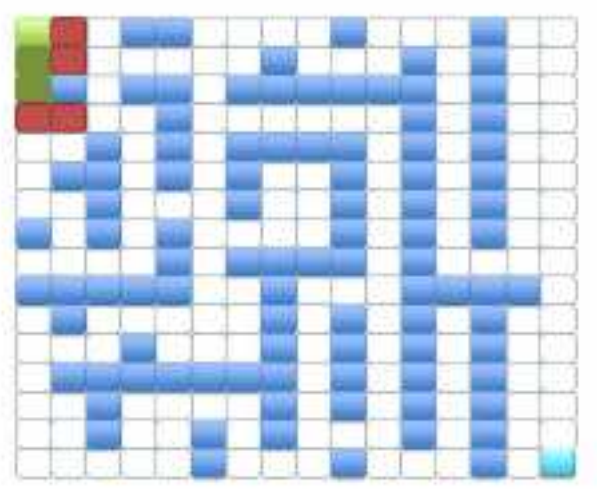
Laberinto



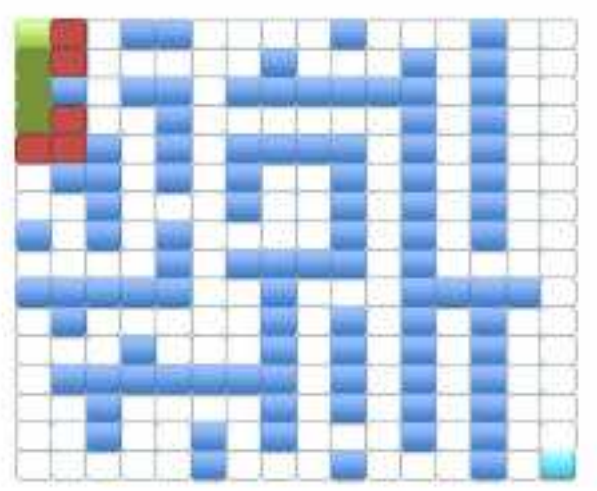
Laberinto



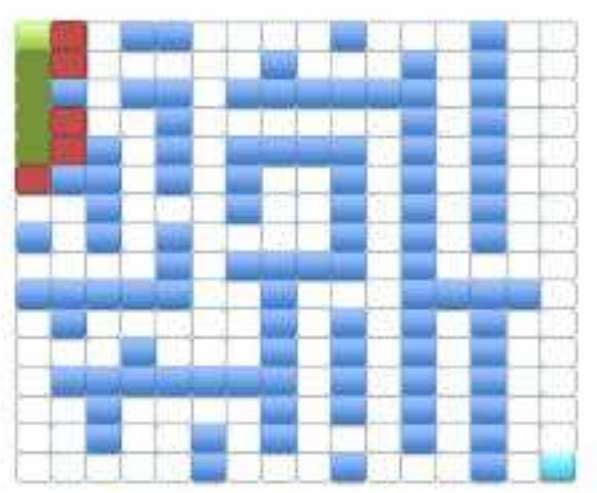
Laberinto



Laberinto



Laberinto



Características

- Requiere backtracking
- Cuándo:
  - Alcanza el límite de profundidad
  - Ha explorado todos los sucesores de un nodo sin encontrar solución
  - Genera un estado duplicado
- **Completitud**: No asegura encontrar la solución, aunque ésta exista (incompleto)
- **Admisibilidad**: No asegura encontrar la solución óptima (no admisible)
- **Eficiencia**: buena cuando las metas están lejos del estado inicial o hay problemas de memoria. Mala si hay ciclos.

Análisis de complejidad (peor caso)

Búsqueda	Complejidad temporal	Complejidad espacial
Amplitud	$O(b^d)$	$O(b^d)$
Profundidad	$O(b^d)$	$O(d)$

- Cuando la complejidad temporal y espacial son iguales la memoria se agota antes del límite de tiempo

Ejemplo laberinto

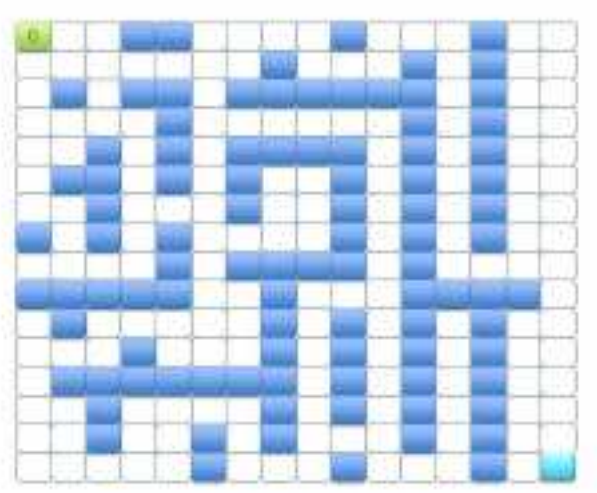
- Amplitud: `http://www.youtube.com/watch?v=tDtMj9wWtEk`
- Profundidad: `http://www.youtube.com/watch?v=AKgo5I5eYAg`

En este tema

Búsqueda de coste no uniforme

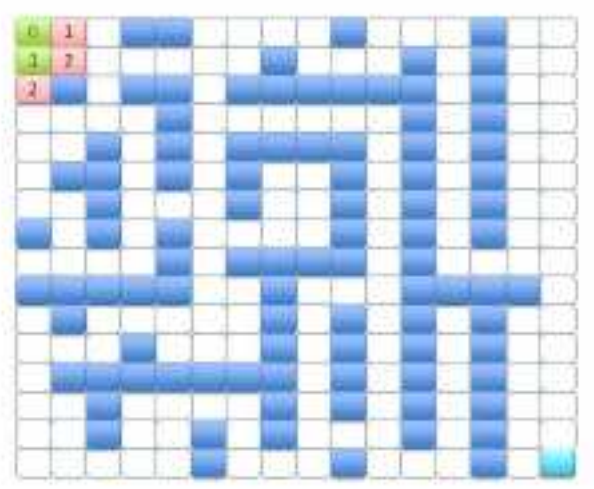
- **Dijkstra**
  - Búsqueda en amplitud
  - $g(n)$: coste de alcanzar el nodo n desde el nodo raíz (I)
  - Expande el nodo con mínima $g(n)$
- **Ramificación y acotación (*Branch and Bound* B&B)**
  - Normalmente búsqueda en profundidad
  - Cuando se encuentra una solución, su coste se utiliza como límite para podar nodos posteriores

Laberinto



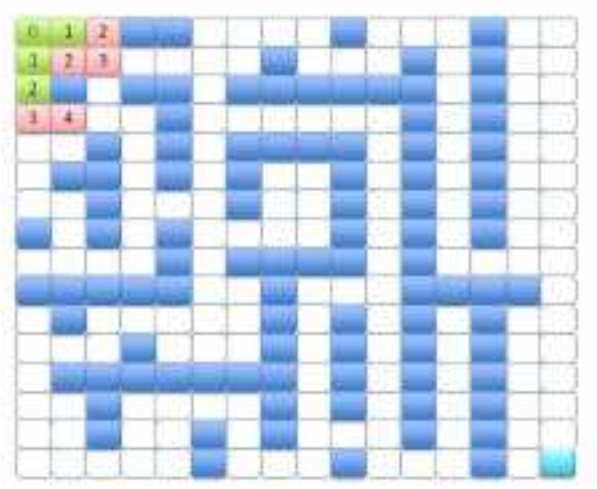
Laberinto

Laberinto



Laberinto

Laberinto



Laberinto

Resumen Búsqueda

1. Formalizar el problema: ESPACIO DE PROBLEMA
   - **Estados**
   - **Operadores**
   - Estado inicial
   - Metas o estado final
2. Estimación de la **complejidad del problema**. Calcular el número de estados diferentes del problema
3. **Algoritmos** de búsqueda:
   - Orden en que se generan los nodos
   - Lista de abierta, nodos generados, nodos expandidos
   - **Propiedades** del algoritmo: completitud, admisibilidad, complejidad temporal y espacial (factor de ramificación y profundidad)
   - Amplitud (completo, admisible, complejidad exponencial), Profundidad (no completo, no admisible, complejidad espacial lineal)

En este tema

En este tema

Heurística

- Si se tiene conocimiento perfecto $\longrightarrow$ algoritmo exacto
- Si no se tiene conocimiento $\longrightarrow$ búsqueda sin información
- En la mayor parte de los problemas que resuelven los humanos, se está en posiciones intermedias
- **Heurística:** (del griego "*heurisko*" ($\varepsilon\acute{\upsilon}\rho\iota\sigma\kappa\omega$): "**yo encuentro**") conocimiento parcial sobre un problema/dominio que permite resolver problemas eficientemente en ese problema/dominio

Heurística

- Si se tiene conocimiento perfecto $\longrightarrow$ algoritmo exacto
- Si no se tiene conocimiento $\longrightarrow$ búsqueda sin información
- En la mayor parte de los problemas que resuelven los humanos, se está en posiciones intermedias
- **Heurística:** (del griego "*heurisko*" ($\varepsilon\acute{\upsilon}\rho\iota\sigma\kappa\omega$): "**yo encuentro**") conocimiento parcial sobre un problema/dominio que permite resolver problemas eficientemente en ese problema/dominio
- Representación de las heurísticas
  - **funciones** $h(n)$: coste estimado desde n a la solución
  - metareglas
- **Las funciones heurísticas se descubren resolviendo modelos *simplificados* (relajados) del problema real**:
  Coste óptimo del problema relajado

Heurística

- Si se tiene conocimiento perfecto $\longrightarrow$ algoritmo exacto
- Si no se tiene conocimiento $\longrightarrow$ búsqueda sin información
- En la mayor parte de los problemas que resuelven los humanos, se está en posiciones intermedias
- **Heurística:** (del griego "*heurisko*" ($\varepsilon\acute{\upsilon}\rho\iota\sigma\kappa\omega$): "**yo encuentro**") conocimiento parcial sobre un problema/dominio que permite resolver problemas eficientemente en ese problema/dominio
- Representación de las heurísticas
  - **funciones** $h(n)$: coste estimado desde n a la solución
  - metareglas
- **Las funciones heurísticas se descubren resolviendo modelos *simplificados* (relajados) del problema real**:
  Coste óptimo del problema relajado

Problemas relajados. 8-puzzle

- **Restricciones**:
  - sólo se puede mover intercambiando el blanco
  - el movimiento sólo se puede hacer a casillas adyacentes horizontal o vertical
  - en cada paso, se intercambian los contenidos de dos casillas
- **Relajaciones:**
  - si quitamos las dos primeras restricciones, generamos la heurística de número de casillas mal colocadas
  - si quitamos la primera restricción, generamos la heurística de la distancia de Manhattan

Problemas relajados. 8-puzzle

- **Restricciones**:
  - sólo se puede mover intercambiando el blanco
  - el movimiento sólo se puede hacer a casillas adyacentes horizontal o vertical
  - en cada paso, se intercambian los contenidos de dos casillas
- **Relajaciones:**
  - si quitamos las dos primeras restricciones, generamos la heurística de número de casillas mal colocadas
  - si quitamos la primera restricción, generamos la heurística de la distancia de Manhattan

Relajación del 8-puzzle. Casillas mal colocadas

- Se representa con un operador:
  Move(x, y, z): mueve el dígito x de la posición y a la posición z
  IF On(x,y), Free(z), Adjacent(y,z)
  THEN On(x,z), Free(y), NOT On(x,y), NOT Free(z)
- Se puede relajar quitando precondiciones al operador
- Una alternativa: relajar *Free*(z) y *Adjacent*(y, z)
  IF On(x,y)
  THEN On(x,z), NOT On(x,y)
- Solución óptima: casillas mal colocadas

Relajación del 8-puzzle. Distania de Manhattan

- Relajamos solo *Free*(z)
  IF On(x,y), Adjacent(y,z)
  THEN On(x,z), NOT On(x,y)
- Solución óptima: distancia de Manhattan

En este tema

Búsqueda Escalada (Hill Climbing)

Escalada (I: estado inicial, F: estado final)

EXITO = False;
Crear lista *ABIERTA* con el nodo inicial *I* (estado inicial);
while *ABIERTA no esté vacía y no EXITO* **do**
  Extraer el primer nodo *N* de *ABIERTA*;
  Generar los sucesores de *N* (expandir *N*);
  if *algún sucesor es un nodo meta (F)* **then**
    EXITO = True;
  else
    Evaluar cada sucesor con la función de evaluación $f(\cdot) = h(\cdot)$;
    S ← mejor sucesor;
    if $f(S)$ *mejor* $f(N)$ **then**
      ABIERTA ← S;
if *EXITO* **then**
  Solución = camino desde I hasta N siguiendo los punteros;
else
  Fracaso

- La búsqueda en escalada es un tipo de búsqueda avara (greedy) con heurística y sin retroceso

Búsqueda Escalada (Hill Climbing)

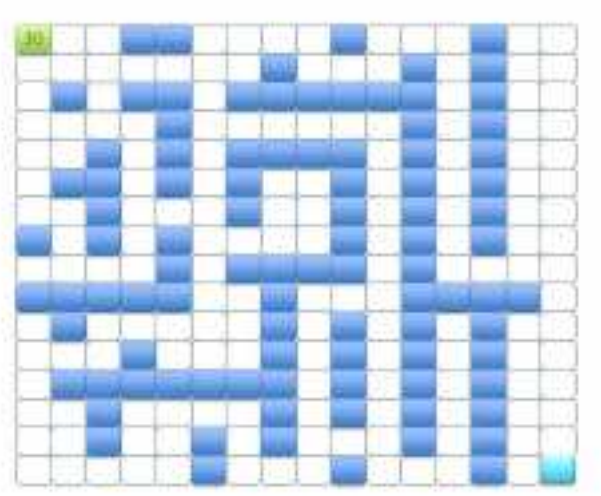
Escalada (I: estado inicial, F: estado final)

```
EXITO = False;
Crear lista ABIERTA con el nodo inicial I (estado inicial);
while ABIERTA no esté vacía y no EXITO do
    Extraer el primer nodo N de ABIERTA;
    Generar los sucesores de N (expandir N);
    if algún sucesor es un nodo meta (F) then
        EXITO = True;
    else
        Evaluar cada sucesor con la función de evaluación f(·) = h(·);
        S ← mejor sucesor;
        if f(S) mejor f(N) then
            ABIERTA ← S;
if EXITO then
    Solución = camino desde I hasta N siguiendo los punteros;
else
    Fracaso
```

- La búsqueda en escalada es un tipo de búsqueda avara (greedy) con heurística y sin retroceso

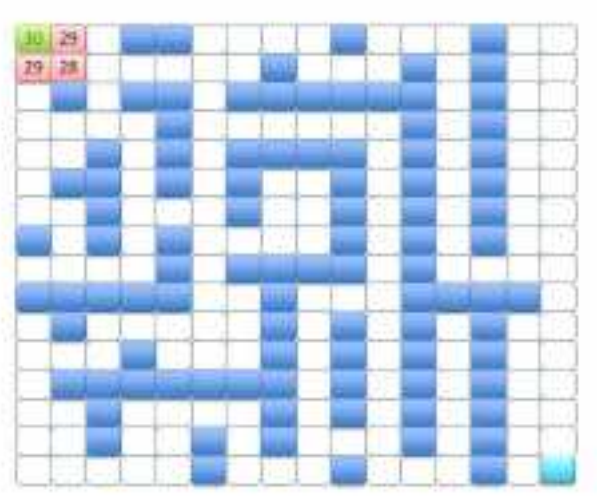
Laberinto

$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$ (distancia de Manhattan)



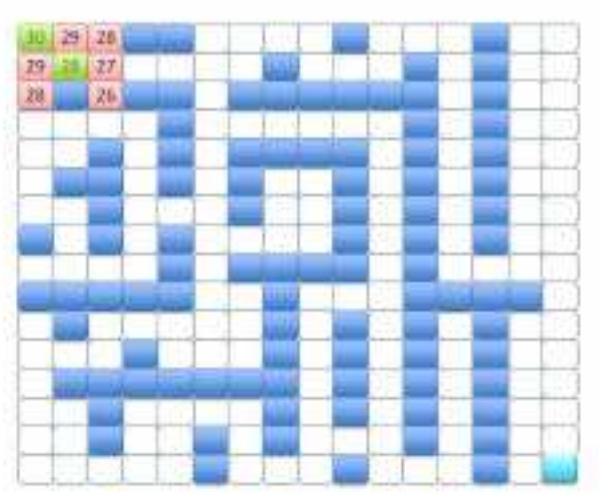
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$$



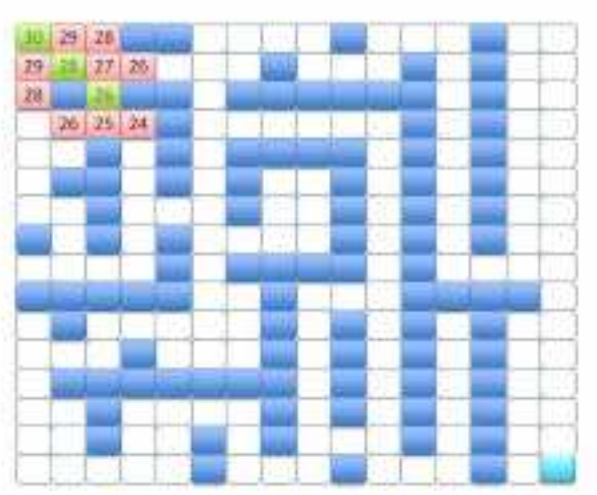
Laberinto

$h(n) = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$



Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$$

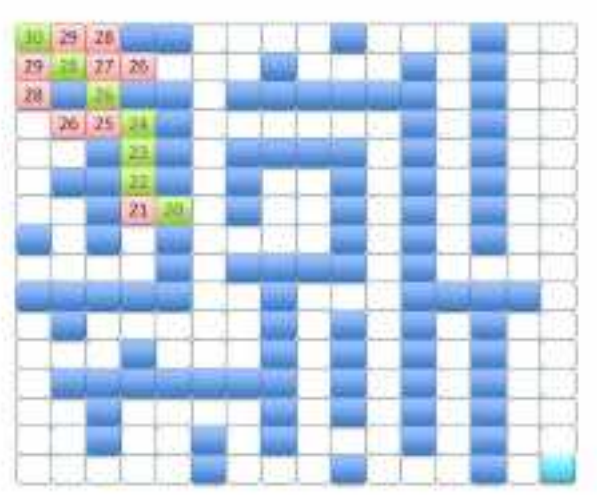


Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$$

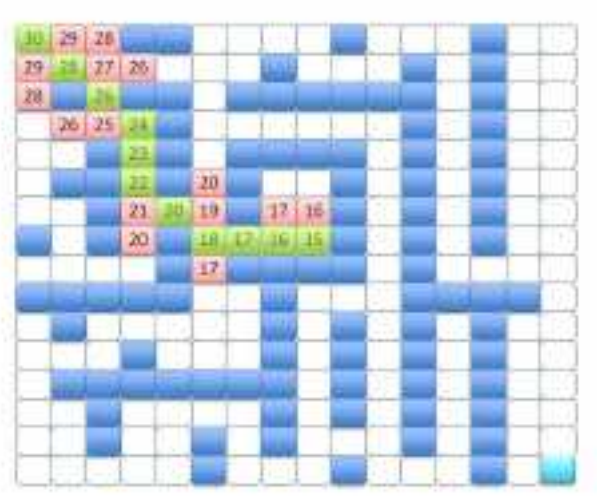
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$$



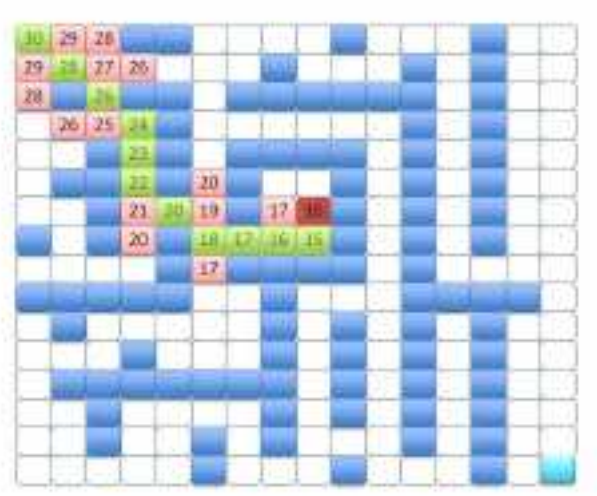
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$$



Laberinto

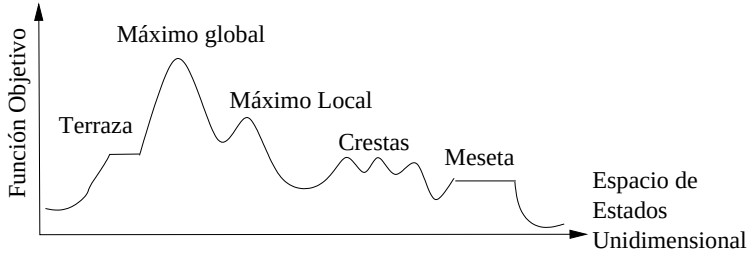
$$h(n = (x_i, y_i), m = (x_m, y_m)) = |x_i - x_m| + |y_i - y_m|$$



Características

Problemas de los métodos *avariciosos*

- **Máximos (o mínimos) locales**: pico que es más alto que cada uno de sus estados vecinos, pero más bajo que el máximo global
- **Mesetas**: zona del espacio de estados con función de evaluación plana
- **Crestas**: zona del espacio de estados con varios máximos (mínimos) locales



Características

Soluciones

- Backtraking (retroceso)
- Dar más de un paso
- Reinicio aleatorio
- Seguir aunque no mejore la f del padre

Propiedades

- **Completitud**: no tiene porqué encontrar la solución
- **Admisibilidad**: no siendo completo, aún menos será admisible
- **Eficiencia**: rápido y útil si la función es monótona (de)creciente

Características

Soluciones

- Backtraking (retroceso)
- Dar más de un paso
- Reinicio aleatorio
- Seguir aunque no mejore la f del padre

Si hay vecinos con el mismo coste, se sigue lo indicado.

Propiedades

- **Completitud**: no tiene porqué encontrar la solución
- **Admisibilidad**: no siendo completo, aún menos será admisible
- **Eficiencia**: rápido y útil si la función es monótona (de)creciente

En este tema

Búsqueda Mejor Primero

Mejor-Primero (I: estado inicial, F: estado final)

ABIERTA = {I}, *CERRADA* = ∅, *EXITO* = False;
while *ABIERTA no esté vacía y no EXITO* **do**
  Extraer el primer nodo *N* de *ABIERTA* y añadirlo a *CERRADA*;
  if *N es un estado meta (F)* **then**
    EXITO = True
  else
    Expandir *N* generando sus sucesores (que no son predecesores de *N*): conjunto *S*;
    for *cada estado s* ∈ *S* **do**
      if *s* ∉ *ABIERTA* ∪ *CERRADA* **then**
        Añadir *s* a *ABIERTA* y generar un puntero de *s* a *N*
      else
        if *s* ∈ *OPEN* **then**
          g(s) = min {previous-g(), new-g()}
        else
          if *new-g()* < *previous-g()* **then**
            mover s de *CERRADA* a *ABIERTA*
    Ordenar *ABIERTA* según *f*(·)
if *EXITO* **then**
  Solución = camino desde I hasta N siguiendo los punteros;
else
  Fracaso

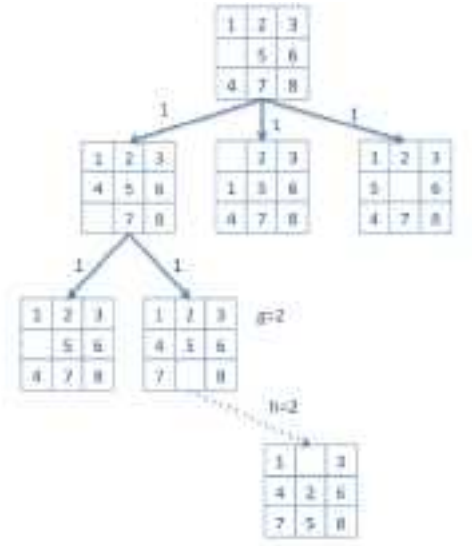
Este algoritmo encuentra siempre la solución y esta será la optima.
No desecha nodos, se mantienen ambos y los utilizados pasan a cerrado
Si encontramos repetido, lo que hacemos es calcular su coste y comparlo con el de cerrado y quedarnos con el menor.
Y ordenamos abierta en coste ascendente.

A* (Hart, Nilsson y Rafael, 1968)

- Función de ordenación de nodos: $f(n) = g(n) + h(n)$
  - $f(n)$: función de evaluación
  - $g(n)$: función de coste de ir desde el nodo inicial al nodo n
  - $h(n)$: función heurística que mide la distancia estimada desde n a algún nodo meta
- $g(n)$ se calcula como la suma de los costes de los arcos recorridos, $k(n_i, n_j)$
- Los valores reales sólo se pueden conocer al final de la búsqueda
  - $f^*(n)$: coste real para ir desde el nodo inicial a algún nodo meta a través de n
  - $g^*(n)$: coste real para ir desde el nodo inicial al nodo n
  - $h^*(n)$: coste real para ir desde el nodo n a algún nodo meta

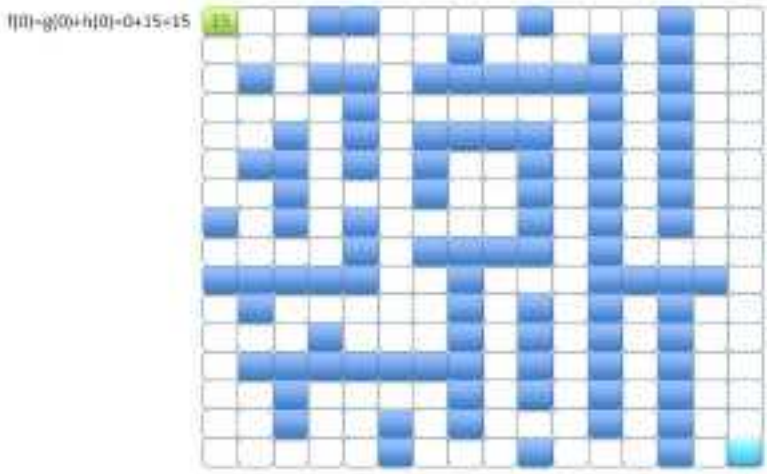
$g+h=f$

A*



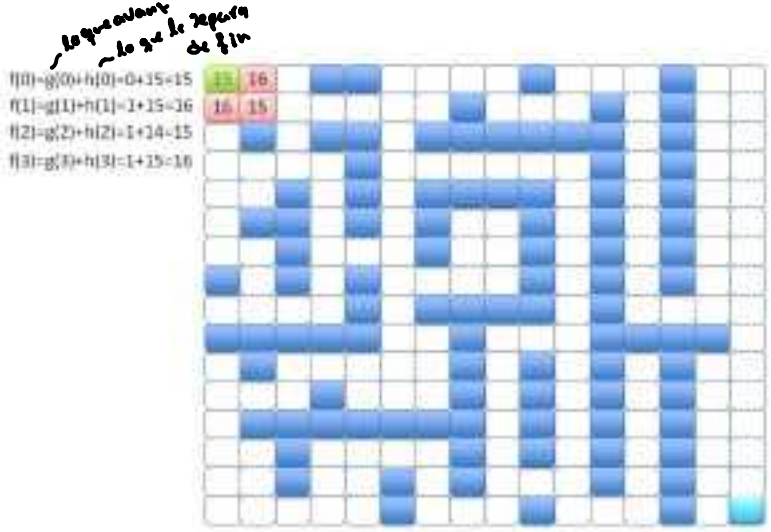
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



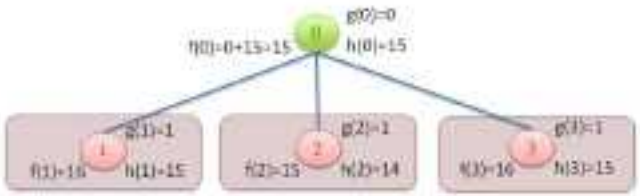
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



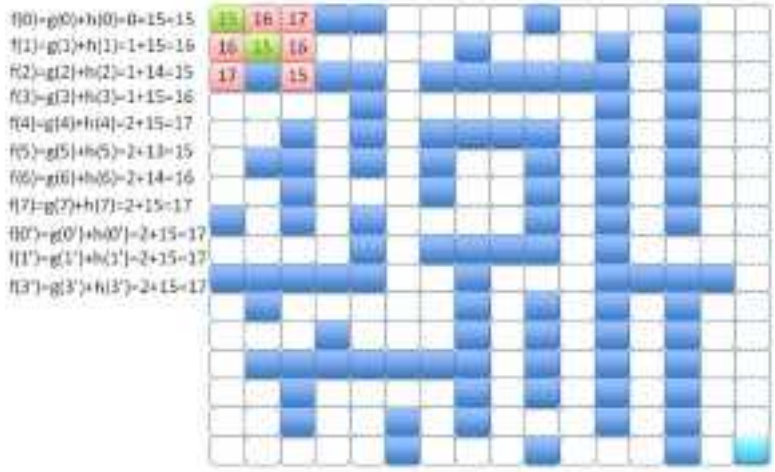
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



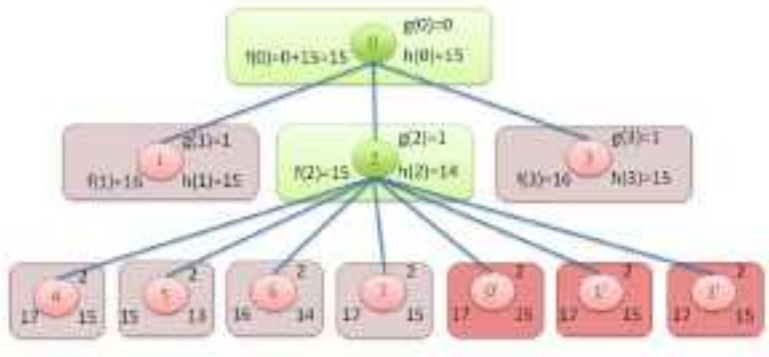
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



Laberinto

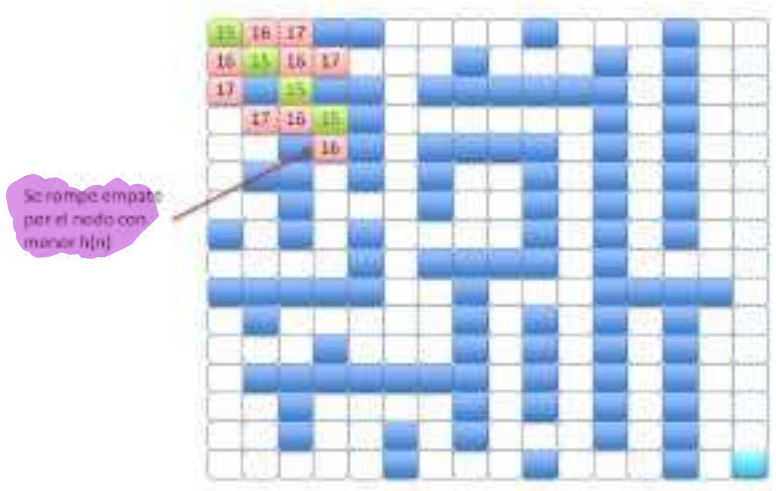
$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$

Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$

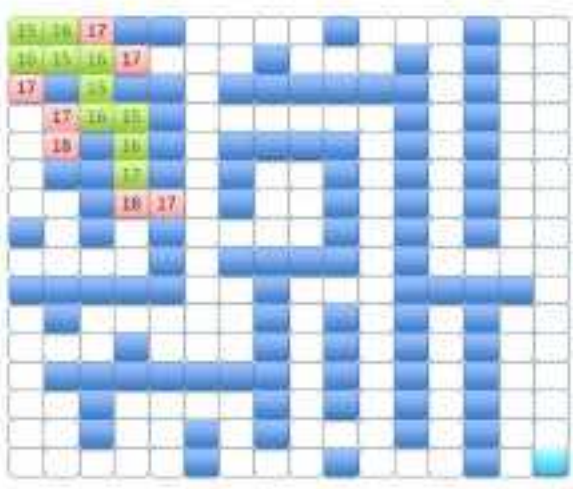
Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$



Laberinto

$$h(n = (x_i, y_i), m = (x_m, y_m)) = \max(|x_i - x_m|, |y_i - y_m|)$$

Características

- **Completitud**: si existe solución, la encuentra
- **Admisibilidad**: si hay una solución óptima, la encuentra si
  - el número de sucesores es finito para cada nodo,
  - $k(n_i, n_j) > \epsilon > 0$ en cada arco, y
  - La función $h(n)$ es admisible: $h(n) \leq h^*(n)\ \forall n$
- Si $\forall n, h_1(n) \leq h_2(n)$,
  - $h_2(n)$ es una heurística mejor informada que $h_1(n)$
  - $h_2(n)$ expande un número de nodos menor o igual que $h_1(n)$
  - Ejemplo: la distancia de Manhattan está mejor informada (menos relajada) que el número de piezas mal colocadas

Información adicional

- Extremos
  - $h(n) = 0$ para cada nodo: no se tiene información (Dijkstra)
  - $h(n) = h^*(n)$ para cada nodo: se tiene información perfecta
- No tiene sentido dedicar más coste computacional a calcular una buena $h(n)$ que a realizar la búsqueda equivalente: **equilibrio**
- Heurística **admisible**: $h(n) \leq h^*(n)\ \forall n$
- Heurística **consistente**: $h(n) \leq k(n, n') + h(n')\ \forall n, n'$ (n' sucesor de n)
- Heurística *goal-aware*: $h(n) = 0\ \forall n \in G$ (G nodos meta)
- Si $h(n)$ es consistente, no hace falta reabrir nodos
- Propiedad: toda heuríca consistente y *goal-aware* es admisible

Técnicas mejor primero

- **Búsqueda no informada**
  - Amplitud: $f(n) = profundidad(n)$
  - Dijkstra: $f(n) = g(n)$
- **Búsqueda informada o heurística**
  - A* [Hart *et. al.*, 1968]
  - Otras:
    - Búsqueda mejor primero avara (GBFS): $f(n) = h(n)$
    - IDA* [Korf, 1985]: $f(n) = g(n) + h(n)$
    - A* ponderado [Polh, 1970] (weighted A*) : $f(n) = g(n) + \omega h(n), \omega > 1$

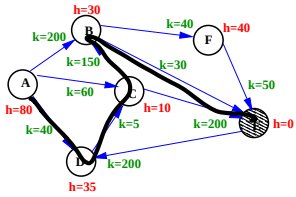
Resumen

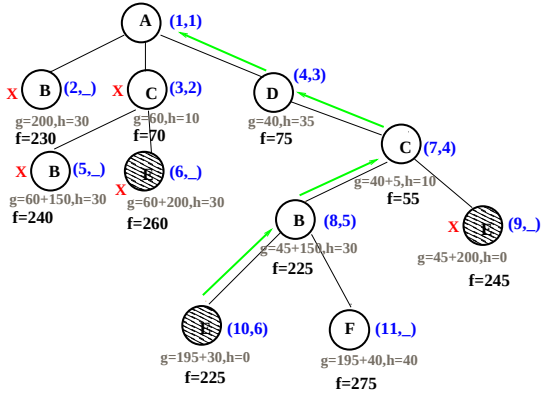
Dado un $P =< S, O, s_i, meta >$

- Función heurística: $h : S \rightarrow \Re^+$
  $h(s, meta)$: coste estimado de alcanzar la meta desde s
- Heurística admisible: $\forall s \in S, h(s) <= h^*(s)$
- Se obtiene planteando una **relajación del problema** $\rightarrow$ solución óptima
- Métricas a optimizar: longitud o coste de la solución

- Algoritmos de búsqueda heurística
  - **Escalada**: avaricioso o *greedy*
    No completo, no admisible, complejidad temporal y espacial lineal
  - A^*: mejor primero
    Completo, *admisible*, complejidad exponencial
    condiciones: factor ramificación finito, coste > 0, **heurística admisible**

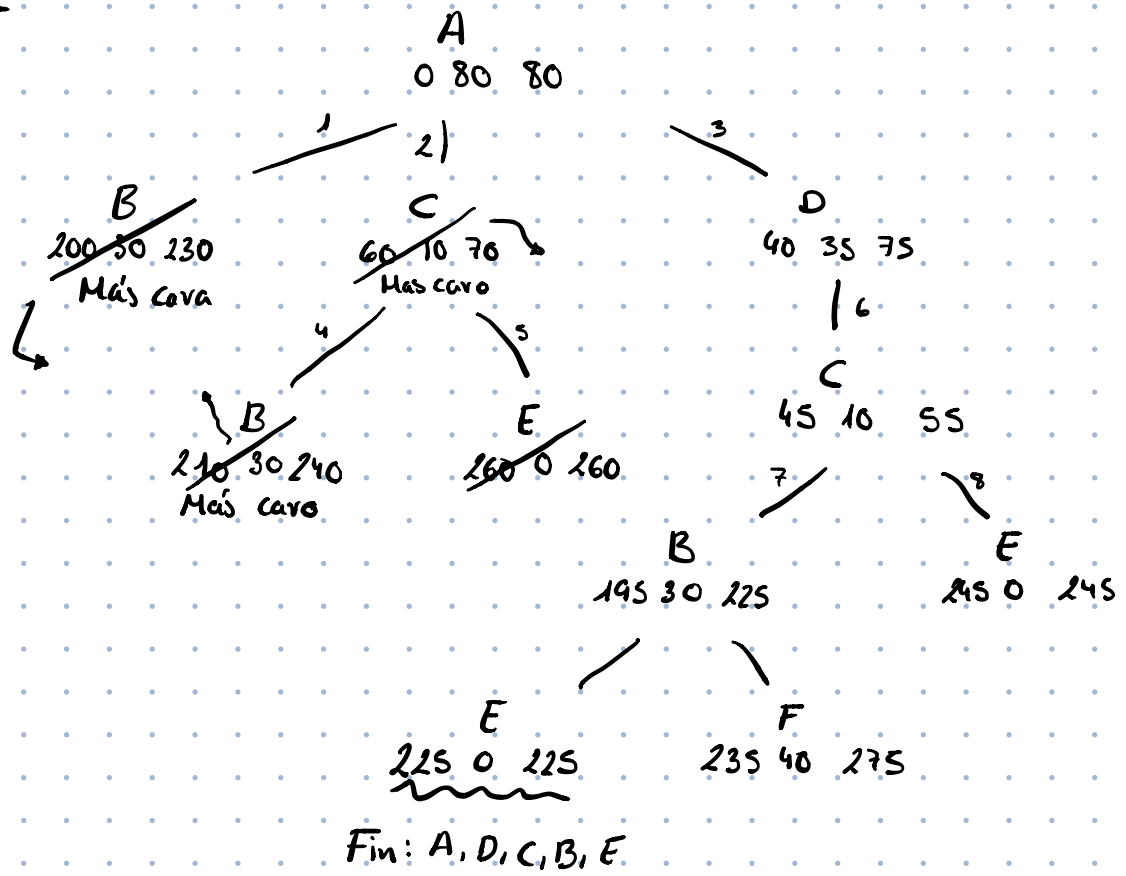
Resumen: Ejemplo A^*





- $g(s)$: coste desde estado inicial hasta s (suma de ks)
- $h(s)$: coste estimado de s a la meta
- $f(s) = g(s) + h(s)$
- Se expanden nodo de menor $f(s)$

A*


A
0 80 80
1
2
3
B
200 30 230
Más cara
C
60 10 70
Mas caro
D
40 35 75
4
5
6
B
210 30 240
Más caro
E
260 0 260
C
45 10 55
7
8
B
195 30 225
E
245 0 245
E
225 0 225
F
235 40 275

Fin: A, D, C, B, E

Resumen: Algoritmos de búsqueda

- Búsqueda no informada
  - Coste uniforme:
    - **Amplitud (BFS)**: completo, admisible, complejidad exponencial
    - **Profundidad (DFS)**: no completo, no admisible, complejidad espacial lineal
  - Coste no uniforme
    - **Dijkstra**: completo, admisible, complejidad exponencial
    - **Branch and Boud**: *no completo*, *no admisible*, complejidad espacial lineal
- Búsqueda heurística
  - **Escalada (Hill climbing)**: no completo, no admisible, complejidad lineal
  - A^*: *Completo*, *admisible*, complejidad exponencial

condiciones:
factor ramificación finito, coste > 0, **heurística admisible**

Ejemplos en el mapa de un juego

- Amplitud: `https://www.youtube.com/watch?v=z6lUnb9ktkE`
- Profundidad: `https://www.youtube.com/watch?v=dtoFAvtVE4U`
- A*: `https://www.youtube.com/watch?v=huJEgJ82360`

Resumen representación de estados

Dibujar ayuda.

1. Identif. est. inicial
2. Identif. operadores, que permiten pasar de un estado a otro.

   Cambian los estados poco a poco. Son genericos y solo las necesarias.
   Abstracto
3. Identif. estados accesibles desde el inicial
4. Identif. estados finales.
5. Identif. el coste en nº de op. ejecutadas.
6. Solución desde un est. inicial a uno final.
7. Puede haber prioridades entre reglas.

Problema 1: Examen 2016-2017

Cambia el estado de la x el pulsar ■.

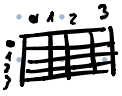
Est. ini = Aleatorias apagadas
Tablero 4x4
Est. f = Todas encendidas.

Solución si se pueden pulsar también bombillas encendida.

BH: stat(x,y,estado) — Casilla por x,y está estado
inv(on,off) — Lo invierte si on es el
inv(off,on) — Lo invierte si off es on.
AllOn(0) — Será terminado

$0 \le x < 4$ $0 \le y < 4$ estado ∈ {on, off}

stat se rellena con las encendidas y apagadas, pero son aleatorio y no se conoce a priori.



BR:

Pulsar: stat(x,y,s), inv(s,t) $\xrightarrow{x<3}$ stat(x+1,y,v), inv(v,w) → modify stat(x,y,t) modif stat(x,y,w)

PulsarBordeD1: stat(3,y,s), inv(s,t) $\xrightarrow{y\neq 0}$ stat(3,y+1,v), inv(v,w) → modify stat(3,y,t), modify stat(3,y+1,w)

PulsarBordeD2: stat(3,0,s), inv(s,t) → modify stat(3,0,t)

ojo

Fin: stat(0,0,on), stat(1,0,on)... stat(0,1,on), stat(1,1,on)... → AllOn(1) STOP

+ prioridad

Fin2: not stat(x,y,off) → AllOn(1) STOP

Si no hay ninguna off → Acaba.

No es eficiente para cortes. Habría que hacer búsqueda, A° o amplitud.