Búsqueda Heurística

Javier Béjar

Inteligencia Artificial - 2021/2022 1Q

CS - GEI

Búsqueda Heurística

- Supone la existencia de una función de evaluación que debe medir la distancia estimada al (a un) objetivo (h(n))
- Esta función de evaluación se utiliza para guiar el proceso haciendo que en cada momento se seleccione el estado o las operaciones más prometedores
- No siempre se garantiza encontrar una solución (de existir ésta)
- No siempre se garantiza encontrar la solución más próxima (la que se encuentra a una distancia, número de operaciones menor)
- Existen múltiples algoritmos:
    - Branch and Bound, Best First Search
    - A, A*
    - IDA*
    - Búsqueda local (Hill climbing, Simulated annealing, Alg. Genéticos)

Ramificación y acotación

- Generaliza BFS, DFS
- Se guarda para cada estado el coste (hasta el momento) de llegar desde el estado inicial a dicho estado. Guarda el coste mínimo global hasta el momento
- Deja de explorar una rama cuando su coste es mayor que el mínimo actual
- Si el coste de los nodos es uniforme equivale a una búsqueda por niveles

Algorithm: Greedy Best First

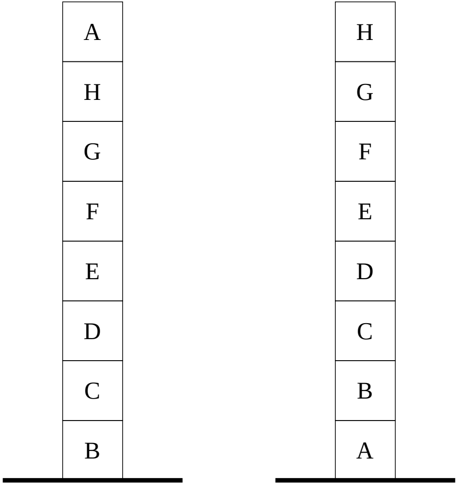
```
Est_abiertos.insertar(Estado inicial)
Actual← Est_abiertos.primero()
while no es_final?(Actual) y no Est_abiertos.vacía?() do
    Est_abiertos.borrar_primero()
    Est_cerrados.insertar(Actual)
    hijos ← generar_sucesores (Actual)
    hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)
    Est_abiertos.insertar(Hijos)
    Actual ← Est_abiertos.primero()
```

- La estructura de abiertos es una cola con prioridad
- La prioridad la marca la funcion de estimación (coste del camino que falta hasta la solucion)
- En cada iteración se escoge el nodo mas cercano a la solucion (el primero de la cola), esto provoca que no se garantice la solucion optima

Operaciones:
- situar un bloque libre en la mesa
- situar un bloque libre sobre otro bloque libre

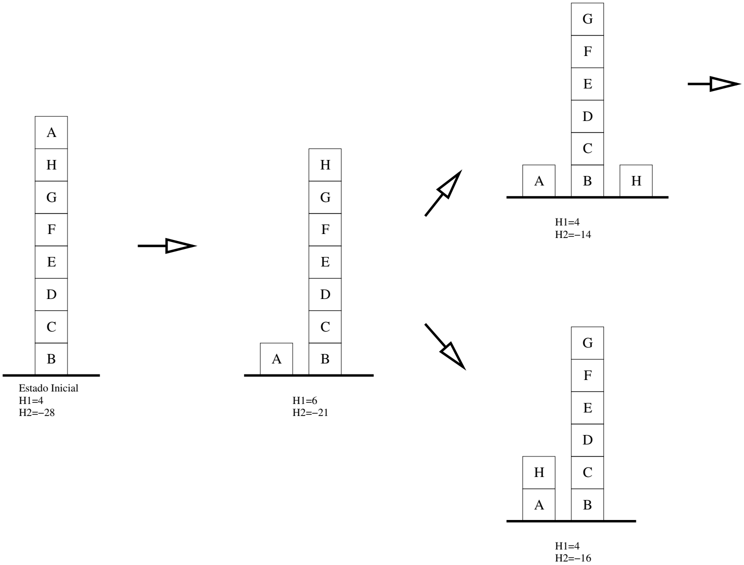
Heurístico 1:
- sumar 1 por cada bloque que esté colocado sobre el bloque que debe
- restar 1 si el bloque no está colocado sobre el que debe

Heurístico 2:
- si la estructura de apoyo es correcta sumar 1 por cada bloque de dicha estructura
- si la estructura de apoyo no es correcta restar 1 por cada bloque de dicha estructura



Estado Inicial
H1=4
H2=−28

Estado Final
H1=8
H2= 28 (= 7+6+5+4+3+2+1)


A
H
G
F
E
D
C
B
Estado Inicial
H1=4
H2=−28
H
G
F
E
D
C
A
B
H1=6
H2=−21
G
F
E
D
C
A
B
H
H1=4
H2=−14
G
F
E
D
H
C
A
B
H1=4
H2=−16

2	8	3
1	6	4
7		5

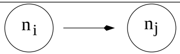
Estado Inicial

1	2	3
8		4
7	6	5

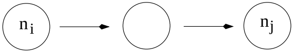
Estado Final

Posibles heurísticos (estimadores del coste a la solución)

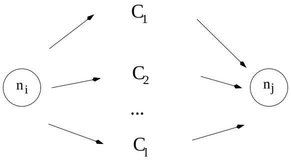
- $h(n) = w(n) =$ #desclasificados
- $h(n) = p(n) =$ suma de distancias a la posición final
- $h(n) = p(n) + 3 \cdot s(n)$
  donde $s(n)$ se obtiene recorriendo las posiciones no centrales y si una ficha no va seguida por su sucesora sumar 2, si hay ficha en el centro sumar 1



Coste de un arco $c(n_i, n_j) > 0$



Coste de un camino $C = \sum_{x=i}^{j-1} c(n_x, n_{x+1})$



Coste del camino mínimo $K(n_i, n_j) = min_{k=1}^{l} C_k$

Si n_j es un nodo terminal $h^*(n_i) = K(n_i, n_j)$

Si n_i es un nodo inicial $g^*(n_j) = K(n_i, n_j)$

Si existen varios nodos terminales $T = \{t_1, \ldots, t_l\}$ $h^*(n_i) = min_{k=1}^{l} K(n_i, t_k)$

Si existen varios nodos iniciales $S = \{s_1, \ldots, s_l\}$ $g^*(n_j) = min_{k=1}^{l} K(s_k, n_j)$

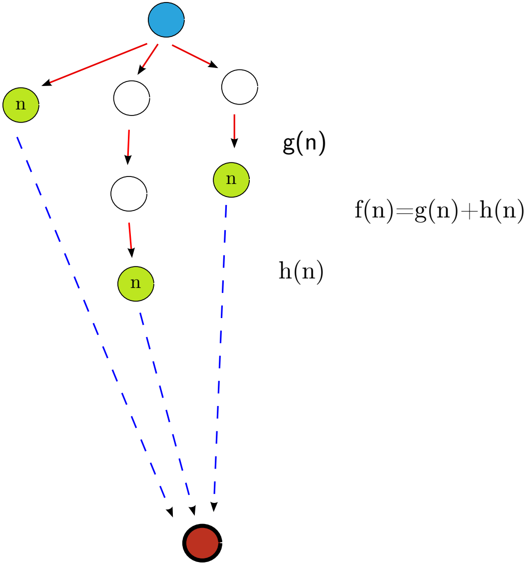
La función de evaluación tiene dos componentes:

1. coste para ir desde el (un) inicio al nodo actual
2. coste (estimado) para ir desde el nodo actual a una solución

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

- f es un valor estimado del coste total del camino que pasa por n
- h (heurístico) es un valor estimado de lo que falta para llegar desde n al (a un) objetivo
- g es un coste real (lo gastado por el camino más corto conocido hasta n)

La preferencia es siempre del nodo con menor f, en caso de empate, la preferencia es del nodo con menor h.


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

Con esta función podemos variar el comportamiento del algoritmo

- Si $\forall n \;\; h(n) = 0$, todo estará controlado por g (estaremos en presencia de un algoritmo de Branch & Bound)
- Si $\forall n \;\; h(n) = 0$ y además el coste de todos los arcos es 1 estaremos realizando una búsqueda en anchura. Si dicho coste fuera 0, la búsqueda sería aleatoria
- Al ser h una estimación del verdadero coste h^*, cuanto más se aproxime h a h^* mayor será la tendencia a explorar en profundidad. Si $h = h^*$ entonces A* converge directamente hacia el objetivo

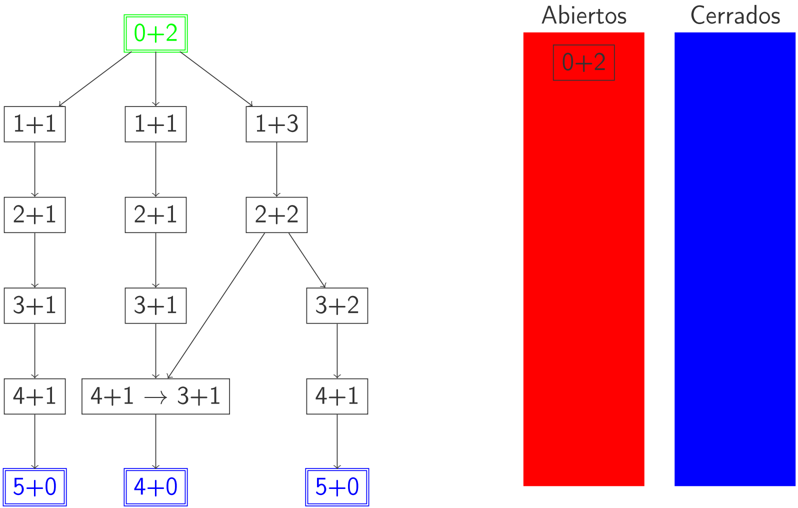
Se puede demostrar que si $h(n)$ es un minorante del coste real $h^*(n)$, es decir si $\forall n \;\; h(n) \leq h^*(n)$ A* encontrará (de haberlo) un camino óptimo.

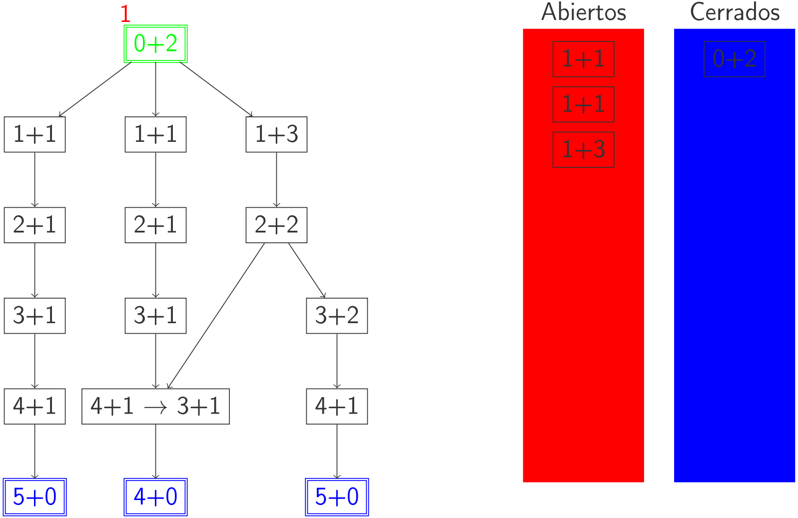
Algorithm: A*

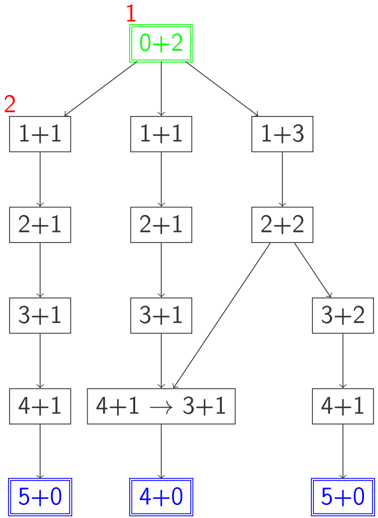
Est_abiertos.insertar(Estado inicial)
Actual ← Est_abiertos.primero()
while no *es_final?(Actual)* **y no** *Est_abiertos.vacía?()* **do**
    Est_abiertos.borrar_primero()
    Est_cerrados.insertar(Actual)
    hijos ← `generar_sucesores` (Actual)
    hijos ← `tratar_repetidos` (Hijos, Est_cerrados, Est_abiertos)
    Est_abiertos.insertar(Hijos)
    Actual ← Est_abiertos.primero()

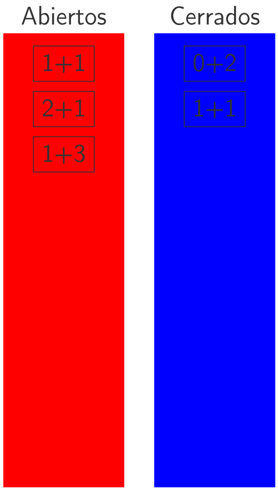
- La estructura de abiertos es una cola con prioridad
- La prioridad la marca la función de estimación ($f(n) = g(n) + h(n)$)
- En cada iteración se escoge el mejor camino (el primero de la cola)
- ¡Es el mismo algoritmo que el Best First!

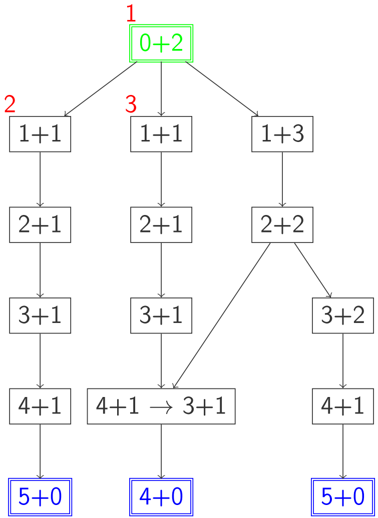
- Si es un repetido que está en la estructura de abiertos
    - Si su coste es menor substituimos el coste por el nuevo, esto podrá variar su posición en la estructura de abiertos
    - Si su coste es igual o mayor nos olvidamos del nodo
- Si es un repetido que esta en la estructura de cerrados
    - Si su coste es menor reabrimos el nodo insertándolo en la estructura de abiertos con el nuevo coste ¡Atención! No hacemos nada con sus sucesores, ya se reabrirán si hace falta
    - Si su coste es mayor o igual nos olvidamos del nodo

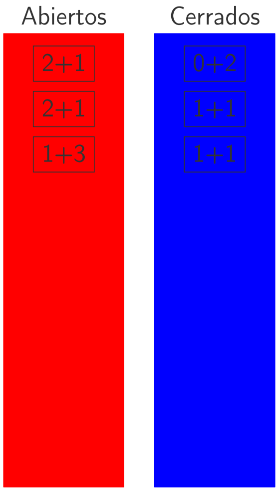

0+2
1+1
1+1
1+3
2+1
2+1
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Abiertos
0+2
Cerrados

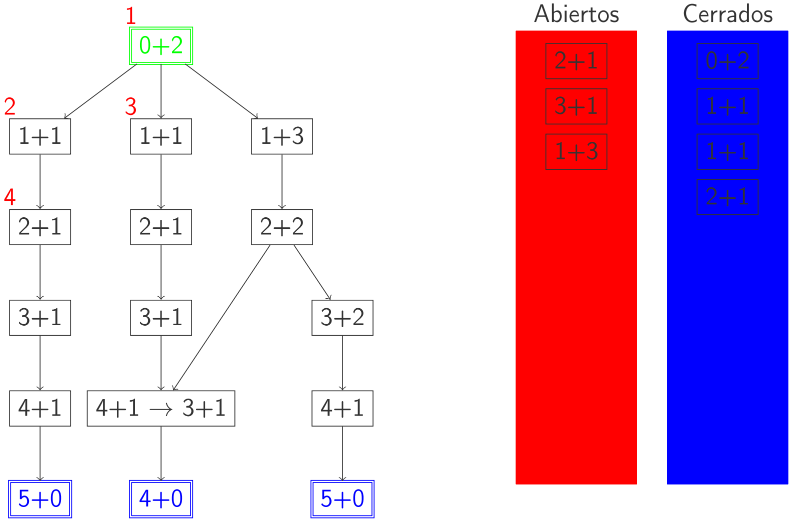

1
0+2
1+1
1+1
1+3
2+1
2+1
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Abiertos
1+1
1+1
1+3
Cerrados
0+2


1
0+2
2
1+1
1+1
1+3
2+1
2+1
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

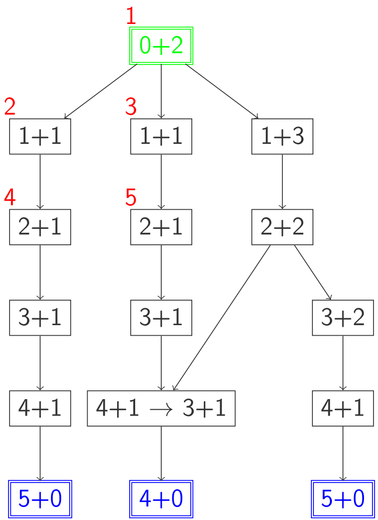

Abiertos
1+1
2+1
1+3
Cerrados
0+2
1+1


1
0+2
2
3
1+1
1+1
1+3
2+1
2+1
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

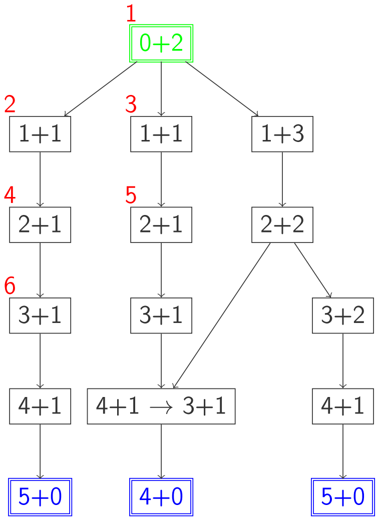

Abiertos
2+1
2+1
1+3
Cerrados
0+2
1+1
1+1


1
0+2
2
3
1+1
1+1
1+3
4
2+1
2+1
2+2
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0
Abiertos
2+1
3+1
1+3
Cerrados
0+2
1+1
1+1
2+1

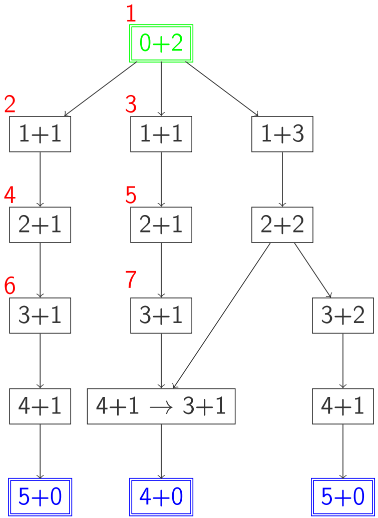
A* Ejemplo

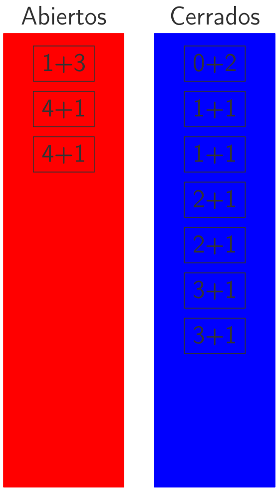


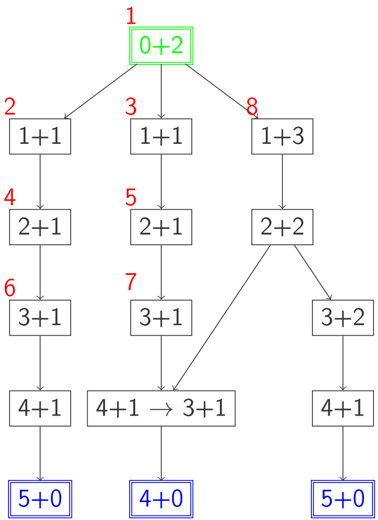
Abiertos	Cerrados
3+1	0+2
3+1	1+1
1+3	1+1
	2+1
	2+1

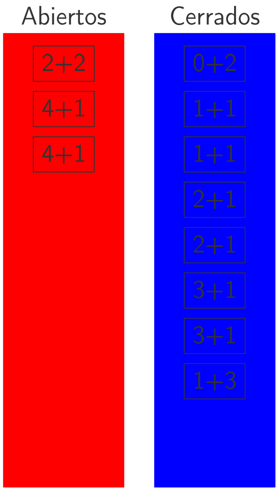


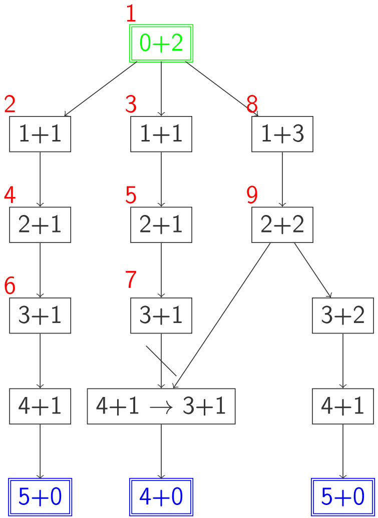
Abiertos	Cerrados
3+1	0+2
1+3	1+1
4+1	1+1
	2+1
	2+1
	3+1


1
0+2
2
1+1
3
1+1
1+3
4
2+1
5
2+1
2+2
6
3+1
7
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

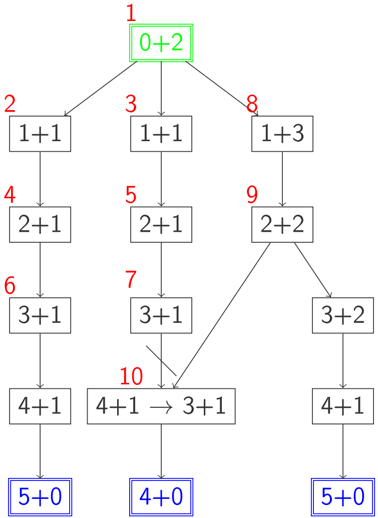

Abiertos
1+3
4+1
4+1
Cerrados
0+2
1+1
1+1
2+1
2+1
3+1
3+1


1
0+2
2
3
8
1+1
1+1
1+3
4
5
2+1
2+1
2+2
6
7
3+1
3+1
3+2
4+1
4+1 → 3+1
4+1
5+0
4+0
5+0

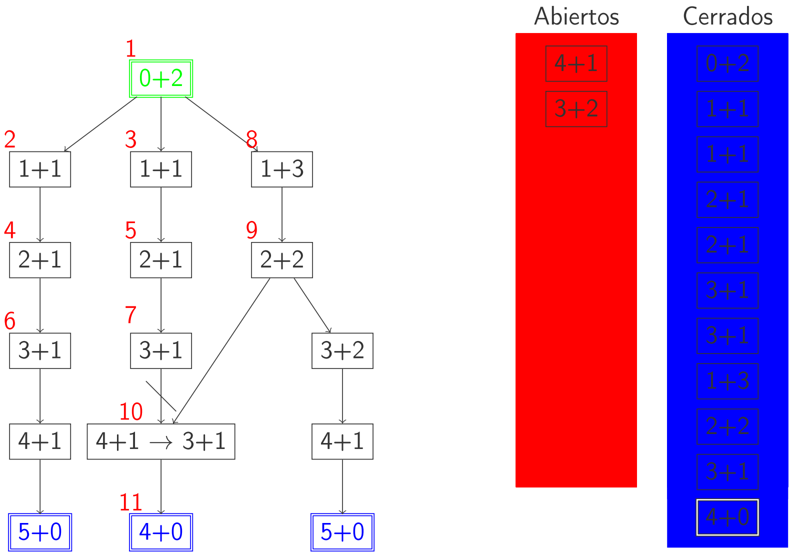

Abiertos
2+2
4+1
4+1
Cerrados
0+2
1+1
1+1
2+1
2+1
3+1
3+1
1+3



Abiertos	Cerrados
3+1	0+2
4+1	1+1
3+2	1+1
	2+1
	2+1
	3+1
	3+1
	1+3
	2+2



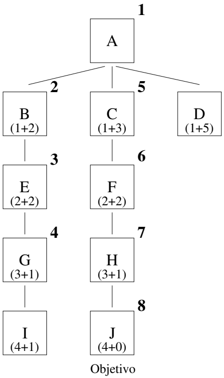
Abiertos	Cerrados
4+0	0+2
4+1	1+1
3+2	1+1
	2+1
	2+1
	3+1
	3+1
	1+3
	2+2
	3+1

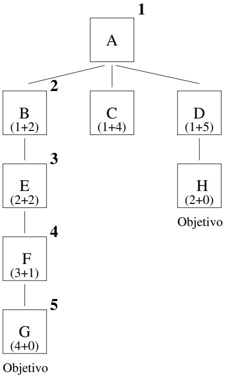

1
0+2
2
1+1
3
1+1
8
1+3
4
2+1
5
2+1
9
2+2
6
3+1
7
3+1
3+2
10
4+1 → 3+1
4+1
4+1
11
5+0
4+0
5+0
Abiertos
4+1
3+2
Cerrados
0+2
1+1
1+1
2+1
2+1
3+1
3+1
1+3
2+2
3+1
4+0

- El algoritmo A* encontrará la solución óptima dependiendo del heurístico
- Si el heurístico es admisible la **optimalidad** está asegurada
- Un heurístico es admisible si se cumple la siguiente propiedad

$$\forall n \;\; 0 \leq h(n) \leq h^*(n)$$

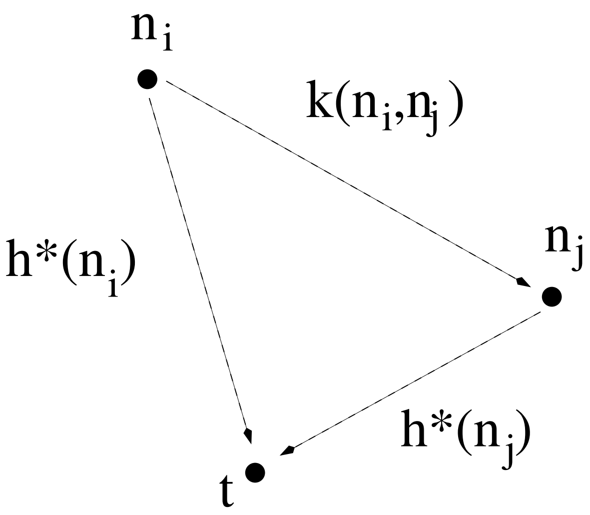
- Por lo tanto, $h(n)$ ha de ser un estimador optimista, nunca ha de sobreestimar $h^*(n)$





h subestima h^*
$h(B) = 2$, que no es real, hemos perdido tiempo pero acabaremos explorando otro camino (C)

h sobreestima h^*
si partiendo de D existiera un camino directo más corto nunca lo alcanzaríamos



- $h^*(n_i)$ = coste mínimo desde n_i a t
- $h^*(n_j)$ = coste mínimo desde n_j a t
- $h^*(n)$ cumple desigualdad triangular

$$h^*(n_i) \leq k(n_i, n_j) + h^*(n_j)$$

- La condición de consistencia exige que $h(n)$ se comporte como $h^*(n)$

$$h(n_i) - h(n_j) \leq k(n_i, n_j)$$

- $h(n)$ consistente $\Longrightarrow$ $h(n)$ estimador uniforme de $h(n)$

- $h(n)$ consistente $\Longrightarrow g(n) = k(s, n)$ (camino óptimo a n)
- Si $g(n) = k(s, n)$, dado que $h(n)$ siempre es constante $\Longrightarrow f(n)$ es mínima
- En este caso no es necesario tratar los nodos duplicados cerrados ya que los nodos expandidos ya no se podrán reexpandir (hemos llegado a ellos por el camino mínimo)

Dado un problema, existen tantos A^* para resolverlo como estimadores podamos definir.

Más informado

Para h_1 y h_2 admisibles, si se cumple

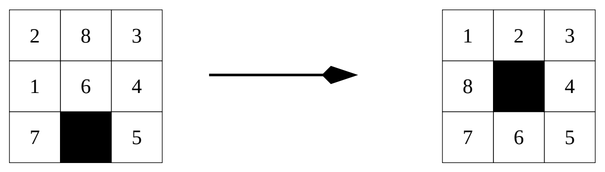
$$\forall n \neq final \;\; 0 \leq h_2(n) < h_1(n) \leq h^*(n)$$

Entonces A_1^* es más informado que A_2^*

- Si A_1^* es más informado que A_2^* entonces si el nodo n es expandido por $A_1^* \Longrightarrow n$ es expandido por A_2^* (pero no al revés)
- Eso quiere decir que A_1^* expande menor número de nodos que A_2^*

- ¿Siempre elegiremos algoritmos más informados?
- Compromiso entre:
  - Tiempo de cálculo de h
    - $h_1(n)$ requerirá más tiempo de cálculo que $h_2(n)$
  - Número de reexpansiones
    - A_1^* puede que reexpanda más nodos que A_2^*
    - Pero si A_1^* es consistente seguro que no lo hará
- Perdida de admisibilidad
  - Puede interesar trabajar con no admisibles para ganar rapidez
  - Algoritmos A_ϵ^* (ϵ-admisibilidad)

Ocho puzzle



h_0

$h_0(n) = 0$ Equivalente a anchura prioritaria, h_0 admisible, muchas generaciones y expansiones

h_1

$h_1(n) = \#piezas\ mal\ colocadas$ h_1 admisible, h_1 mas informado que h_0

h_2

$h_2(n) = \sum_{i \in [1,8]} d_i \quad d_i \equiv$ distancia entre posición de la pieza i y su posición final

h_2 admisible, h_2 no más informado que h_1

$h_2([1,2,3,4,5,6,7,\emptyset,8]) = h_1([1,2,3,4,5,6,7,\emptyset,8]) = 1$

h_3

$h_3(n) = \sum_{i \in [1,8]} d_i + 3 \cdot S(n) \ \ ; con\ S(n) = \sum_{i \in [1,8]} s_i$

$$s_i = \begin{cases} \text{0 si pieza i no en el centro y sucesora correcta} \\ \text{1 si pieza i en el centro} \\ \text{2 si pieza i no en el centro y sucesora incorrecta} \end{cases}$$

h_3 no es admisible, y por lo tanto no es más informado que h_2, pero es más rápido

- El algoritmo A* resuelve problemas en los que es necesario encontrar la mejor solución
- Su coste en espacio y tiempo en el caso medio es mejor que los algoritmos de búsqueda ciega si el heurístico es adecuado
- Existen problemas en los que la dimensión del espacio de búsqueda no permite su solución con A*
- Ademas los nodos a almacenar por A* crecen exponencialmente si:

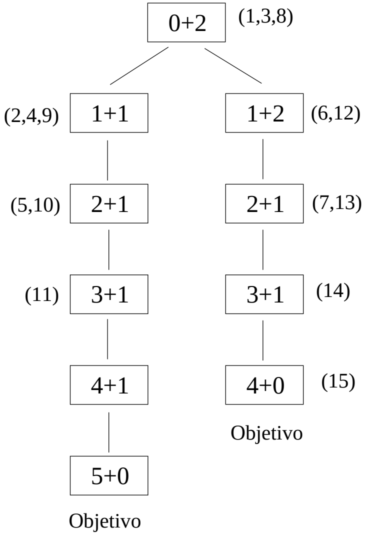
$$|h(n) - h^*(n)| \geq \log(h^*(n))$$

- Existen algoritmos que permiten obtener el óptimo limitando la memoria usada:
  - IDA*
  - Best First Recursivo
  - Memory Bound A* (MA*)

- Similar a ID (es decir iteración de búsqueda en profundidad con un límite en la búsqueda)
- En ID el límite lo daba una cota máxima en la profundidad
- En IDA* el límite lo da una cota máxima sobre el valor de la función f

¡Ojo! La búsqueda es una DFS normal y corriente, el heurístico f sólo se utiliza para podar

Empezamos con corte = f(inicial)



```
Algorithm: IDA* (limite entero)
prof ← f(Estado inicial)
Actual ← Estado inicial
while no es_final?(Actual) y prof<limite do
    Est_abiertos.incializa()
    Est_abiertos.insertar(Estado inicial)
    Actual ← Est_abiertos.primero()
    while no es_final?(Actual) y no Est_abiertos.vacía?() do
        Est_abiertos.borrar_primero()
        Est_cerrados.insertar(Actual)
        Hijos ← generar_sucesores (Actual, prof)
        Hijos ← tratar_repetidos (Hijos, Est_cerrados,
        Est_abiertos)
        Est_abiertos.insertar(Hijos)
        Actual ← Est_abiertos.primero()
    prof ← prof+1
```

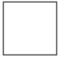
- La función `generar_sucesores` solo genera aquellos con una f menor o igual a la del limite de la iteración
- La estructura de abiertos es ahora una pila (búsqueda en profundidad)
- Hemos de tener en cuenta que si tratamos los nodos repetidos el ahorro en espacio es nulo

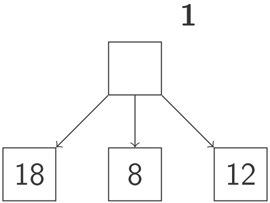
- Las reexpansiones de IDA* pueden suponer un elevado coste temporal
- Existen algoritmos que por lo general reexpanden menos nodos
- Su funcionamiento se basa en eliminar los nodos menos prometedores y guardar información que permita reexpandirlos
- Ejemplos:
  - Best first recursivo
  - Memory Bound A* (MA*)

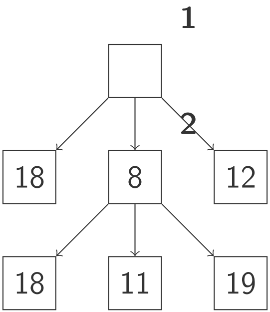
- Es una implementación del Best First recursiva con coste lineal en espacio $O(rp)$
- Olvida una rama cuando su coste supera la mejor alternativa
- El coste de la rama olvidada se almacena en el padre como su nuevo coste
- La rama es reexpandida si su coste vuelve a ser el mejor (regeneramos toda la rama olvidada)

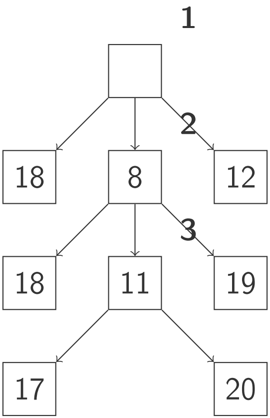
Procedure: BFS-recursivo (nodo,c_alternativo,ref nuevo_coste,ref solucion)

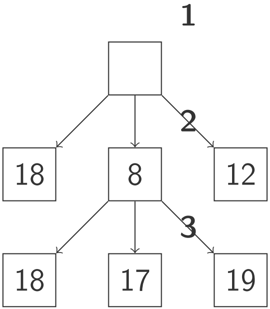
```
if es_solucion?(nodo) then
    solucion.añadir(nodo)
else
    sucesores ← generar_sucesores (nodo)
    if sucesores.vacio?() then
        nuevo_coste ← +∞; solucion.vacio()
    else
        fin ← falso
        while no fin do
            mejor ← sucesores.mejor_nodo()
            if mejor.coste() > c_alternativo then
                fin ← cierto; solucion.vacio(); nuevo_coste ← mejor.coste()
            else
                segundo ← sucesores.segundo_mejor_nodo()
                BFS-recursivo(mejor,min(c_alternativo,segundo.coste()),nuevo_coste, solucion)
                if solucion.vacio?() then
                    mejor.coste(nuevo_coste)
                else
                    solucion.añadir(mejor); fin ← cierto
```

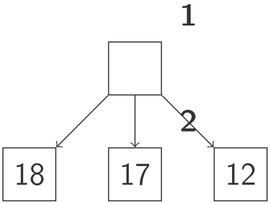


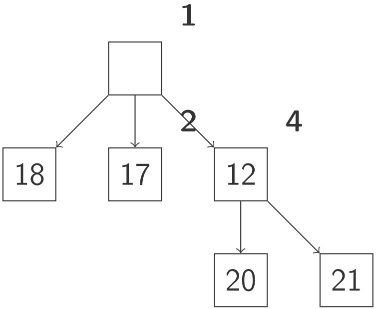

1
18
8
12

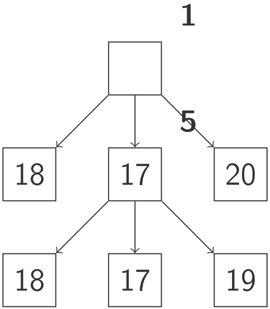

1
2
18
8
12
18
11
19

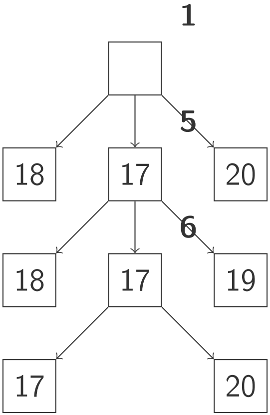

1
2
18
8
12
3
18
11
19
17
20

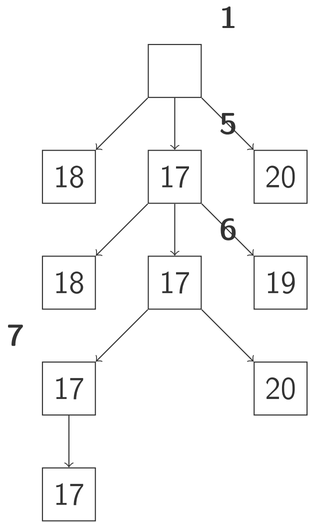

1
2
3
18
8
12
18
17
19

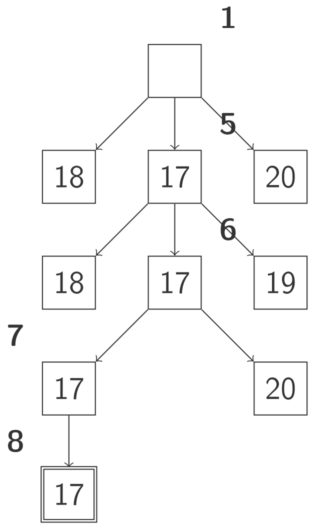

1
2
18
17
12


1
2
4
18
17
12
20
21


1
5
18
17
20
18
17
19


1
5
6
18
17
20
18
17
19
17
20


1
5
6
7
18
17
20
18
17
19
17
20
17


1
5
18
17
20
6
18
17
19
7
17
20
8
17

- Por lo general reexpande menos nodos que IDA*
- El límite de memoria (lineal en espacio) provoca muchas reexpansiones en ciertos problemas
- Al no poder controlar los repetidos su coste en tiempo puede elevarse si hay ciclos
- Solución: Relajar la restricción de memoria

- Impone un límite de memoria (número de nodos que se pueden almacenar, mayor que $O(rp)$)
- Exploramos usando A* y almacenamos nodos mientras quepan en la memoria
- Cuando no quepan eliminamos los peores guardando el mejor coste de los descendientes olvidados
- Reexpandimos si los nodos olvidados son mejores
- El algoritmo es completo si el camino solución cabe en memoria