

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)

Branch and Bound

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

Greedy Best First

Algoritmo: Greedy Best First

Est_abiertos.insertar(Estado inicial)

Actual ← Est_abiertos.primer()

mientras no es_final?(Actual) **y** no Est_abiertos.vací() **hacer**

 Est_abiertos.borrar_primer()

 Est_cerrados.insertar(Actual)

 hijos ← generar_sucesores (Actual)

 hijos ← tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)

 Est_abiertos.insertar(Hijos)

 Actual ← Est_abiertos.primer()

fin

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

Importancia del estimador

Operaciones:

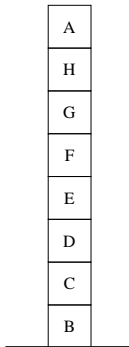
- situar un bloque libre en la mesa
- situar un bloque libre sobre otro bloque libre

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:

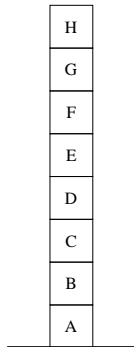
- si la estructura de apoyo es correcta sumar 1 por cada bloque de dicha estructura
- si la estructura de apoyo no es correcta restar 1 por cada bloque de dicha estructura



Estado Inicial

H1=4

H2=-28

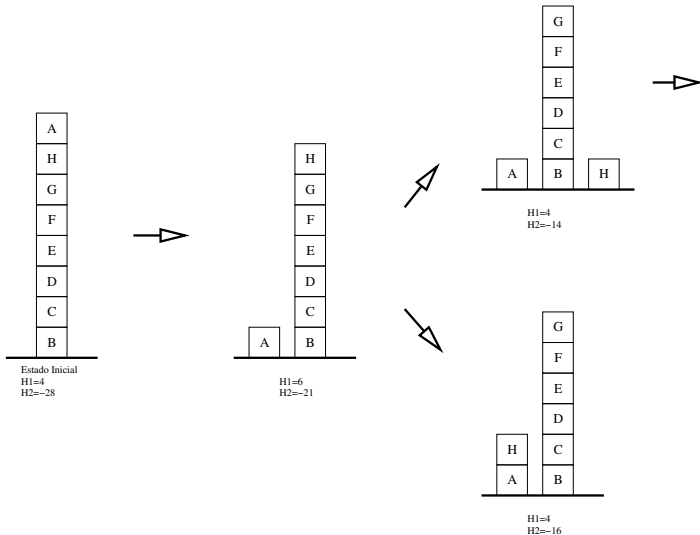


Estado Final

H1=8

H2= 28 (= 7+6+5+4+3+2+1)

Importancia del estimador



Heurísticos

2	8	3
1	6	4
7		5

Estado Inicial

1	2	3
8		4
7	6	5



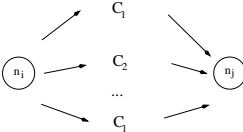
Estado Final

Posibles heurísticos (estimadores del coste a la solución)

- $h(n) = w(n) = \# \text{desclasificados}$
- $h(n) = p(n) = \text{suma de distancias a la posición final}$
- $h(n) = p(n) + 3 \cdot s(n)$

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

Costes

	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 $T = \{t_1, \dots, t_l\}$

$$h^*(n_i) = \min_{k=1}^l K(n_i, t_k)$$

Si existen varios nodos iniciales $S = \{s_1, \dots, s_l\}$

$$g^*(n_j) = \min_{k=1}^l K(s_k, n_j)$$

Búsqueda A*

La función de evaluación tiene dos componentes:

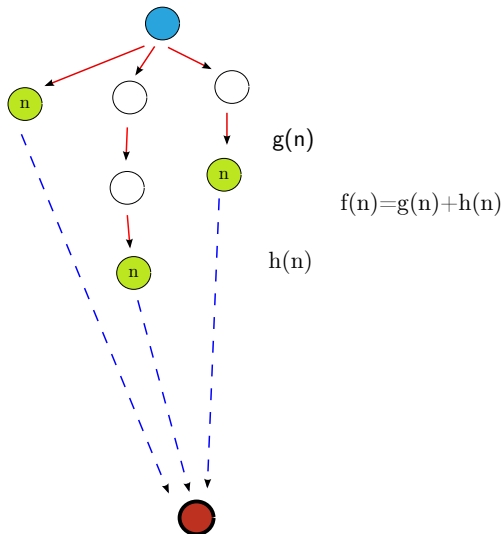
- 1 coste para ir desde el (un) inicio al nodo actual
- 2 coste (estimado) para ir desde el nodo actual a una solución

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

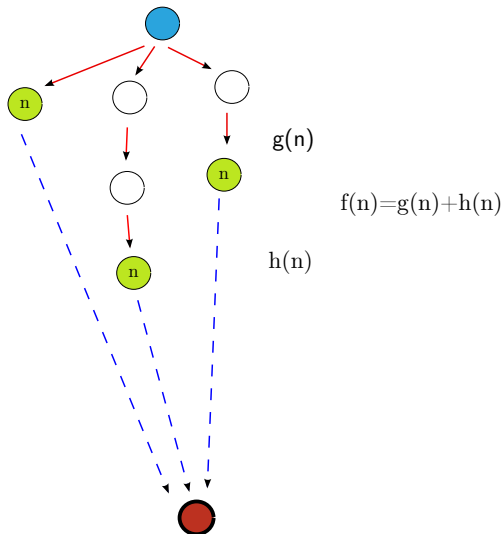
- f es un valor **estimado** del coste total del camino que pasa por n
- h (heurístico) es un valor **estimado** de lo que falta para llegar desde n al (a un) objetivo
- g es un coste **real** (lo gastado por el camino más corto **conocido** hasta n)

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

Búsqueda A*



Búsqueda A*



Búsqueda A*

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.

El algoritmo A*

Algoritmo: A*

Est_abiertos.insertar(Estado inicial)

Actual \leftarrow Est_abiertos.primer()

mientras no es_final?(Actual) y no Est_abiertos.vací() **hacer**

 Est_abiertos.borrar_primer()

 Est_cerrados.insertar(Actual)

 hijos \leftarrow generar_sucesores (Actual)

 hijos \leftarrow tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)

 Est_abiertos.insertar(Hijos)

 Actual \leftarrow Est_abiertos.primer()

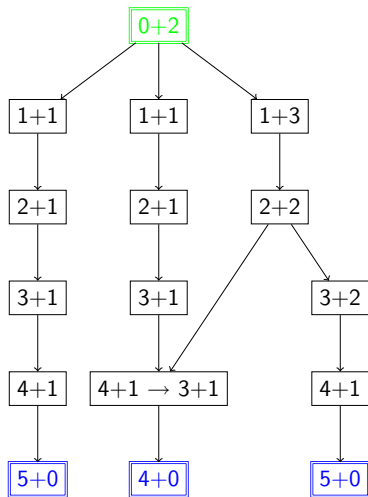
fin

- 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!

Tratamiento de repetidos

- Si es un repetido que está en la estructura de abiertos
 - Si su coste es menor sustituimos 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

A* Ejemplo

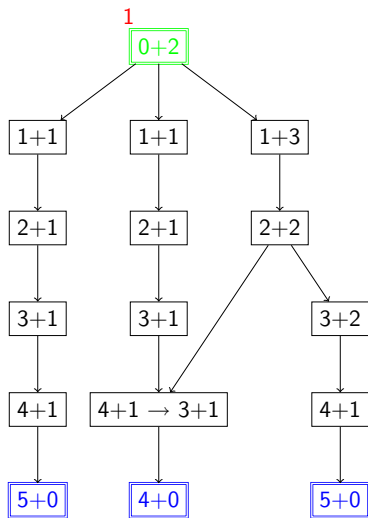


Abiertos

0+2

Cerrados

A* Ejemplo



Abiertos

1+1

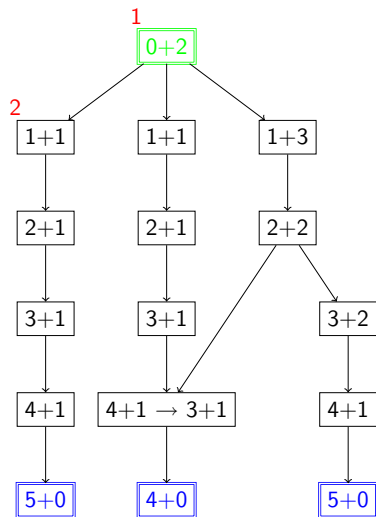
1+1

1+3

Cerrados

0+2

A* Ejemplo



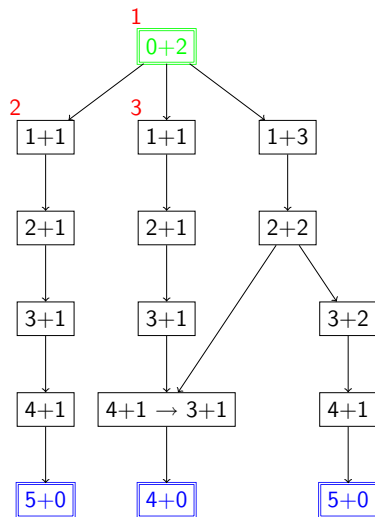
Abiertos

 $1+1$ $2+1$ $1+3$

Cerrados

 $0+2$ $1+1$

A* Ejemplo



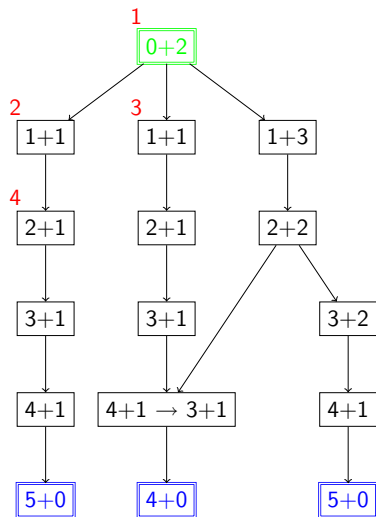
Abiertos

 $2+1$ $2+1$ $1+3$

Cerrados

 $0+2$ $1+1$ $1+1$

A* Ejemplo



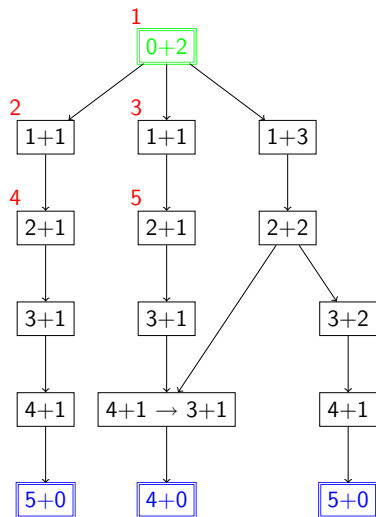
Abiertos

 $2+1$ $3+1$ $1+3$

Cerrados

 $0+2$ $1+1$ $1+1$ $2+1$

A* Ejemplo



Abiertos

3+1

3+1

1+3

Cerrados

0+2

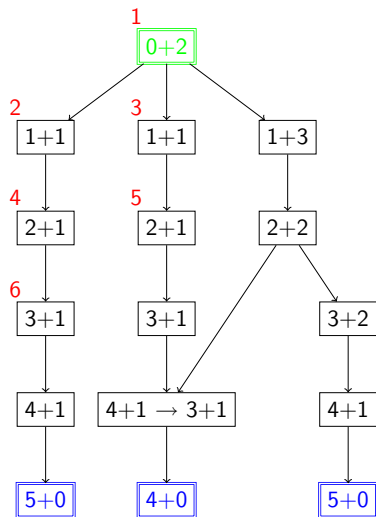
1+1

1+1

2+1

2+1

A* Ejemplo



Abiertos

3+1

1+3

4+1

Cerrados

0+2

1+1

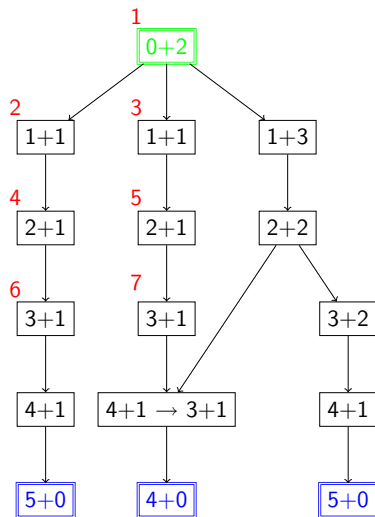
1+1

2+1

2+1

3+1

A* Ejemplo



Abiertos

1+3

4+1

4+1

Cerrados

0+2

1+1

1+1

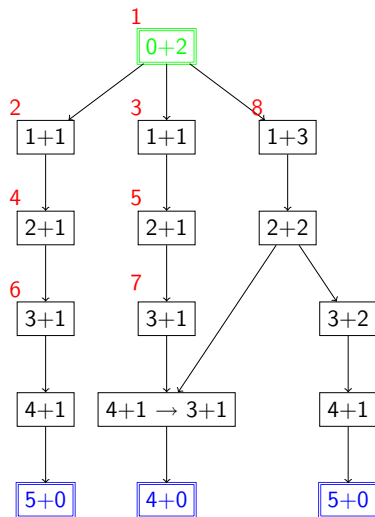
2+1

2+1

3+1

3+1

A* Ejemplo



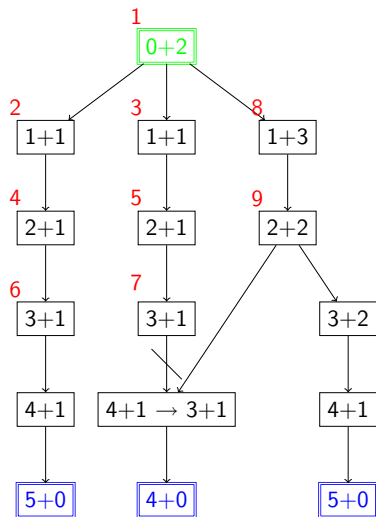
Abiertos

 $2+2$ $4+1$ $4+1$

Cerrados

 $0+2$ $1+1$ $1+1$ $2+1$ $2+1$ $3+1$ $3+1$ $1+3$

A* Ejemplo



Abiertos

3+1

4+1

3+2

Cerrados

0+2

1+1

1+1

2+1

2+1

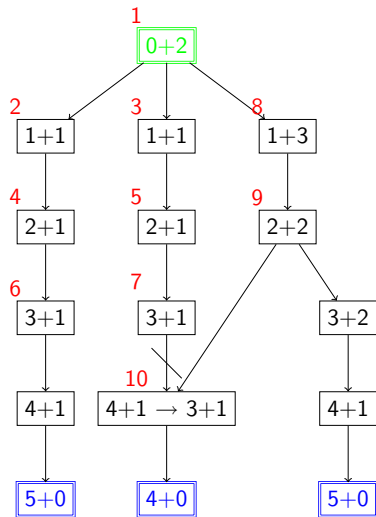
3+1

3+1

1+3

2+2

A* Ejemplo



Abiertos

4+0

4+1

3+2

Cerrados

0+2

1+1

1+1

2+1

2+1

3+1

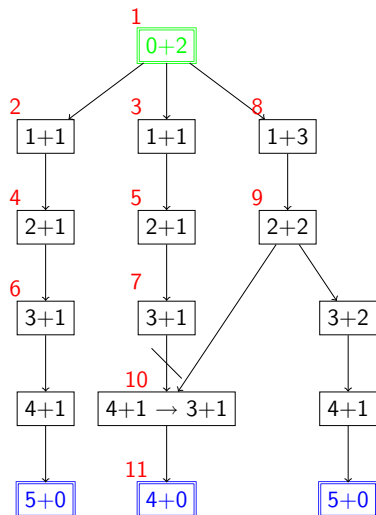
3+1

1+3

2+2

3+1

A* Ejemplo



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

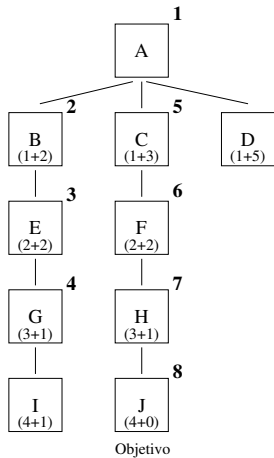
4+0

Admisibilidad

- 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

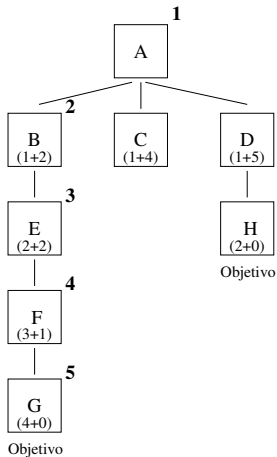
$$\forall n \quad 0 \leq h(n) \leq h^*(n)$$

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



h subestima h^*

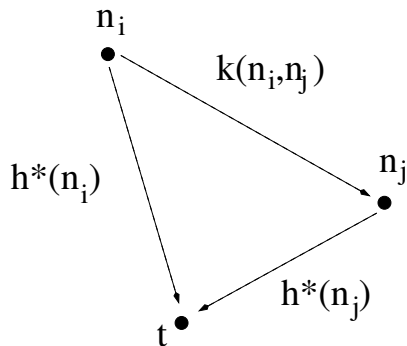
$h(B) = 2$, que no es real, hemos perdido tiempo pero acabaremos explorando otro camino (C)



h sobreestima h^*

si partiendo de D existiera un camino directo más corto nunca lo alcanzaríamos

Condición de Consistencia



- $h^*(n_i)$ = 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 $\implies h(n)$ estimador uniforme de $h(n)$

Utilidad de la consistencia

- $h(n)$ consistente $\implies g(n) = k(s, n)$ (camino óptimo a n)
- Si $g(n) = k(s, n)$, dado que $h(n)$ siempre es constante $\implies f(n)$ 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)

Algoritmos más informados

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

Más informado

Para h_1 y h_2 admisibles, si se cumple

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

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 $A_1^* \implies n$ es expandido por A_2^* (pero no al revés)
- Eso quiere decir que A_1^* expande menor número de nodos que A_2^*

Algoritmos más informados

- ¿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
- Pérdida de admisibilidad
 - Puede interesar trabajar con no admisibles para ganar rapidez
 - Algoritmos A_ϵ^* (ϵ -admisibilidad)

Algoritmos más Informados - 8 puzzle

Ocho puzzle

2	8	3
1	6	4
7		5



1	2	3
8		4
7	6	5

h_0

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

h_1

$h_1(n) = \# \text{piezas mal colocadas}$ h_1 admisible, A_1^* mas informado que A_0^*

Algoritmos más Informados - 8 puzzle

h_2

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

h_2 admisible, Estadísticamente se comprueba que A_2^* es más informado que A_1^*

h_3

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

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

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

Óptimo con limitación de memoria

- 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^*
- Además los nodos a almacenar por A^* crecen exponencialmente si:

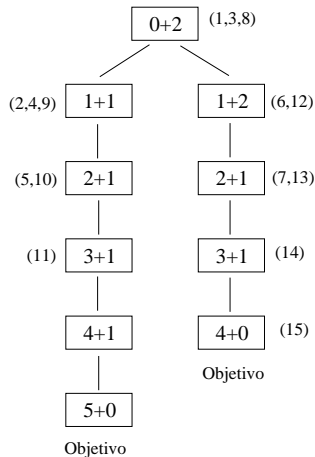
$$|h(n) - h^*(n)| \geq \log(h^*(n))$$

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

Búsqueda IDA*

- 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 $\text{corte} = f(\text{inicial})$



Algoritmo IDA*

Algoritmo: IDA* (límite entero)

$prof \leftarrow f(\text{Estado inicial})$

Actual \leftarrow Estado inicial

mientras no *es_final?*(Actual) **y** $prof < \text{límite}$ **hacer**

 Est_abiertos.inicializa()

 Est_abiertos.insertar(Estado inicial)

 Actual \leftarrow Est_abiertos.primer()

mientras no *es_final?*(Actual) **y no** *Est_abiertos.vací?*() **hacer**

 Est_abiertos.borrar_primer()

 Est_cerrados.insertar(Actual)

 Hijos \leftarrow generar_sucesores (Actual, prof)

 Hijos \leftarrow tratar_repetidos (Hijos, Est_cerrados, Est_abiertos)

 Est_abiertos.insertar(Hijos)

 Actual \leftarrow Est_abiertos.primer()

 prof \leftarrow prof+1

- La función *generar_sucesores* solo genera aquellos con una *f* menor o igual a la del límite 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

Otros algoritmos con limitación de memoria

- 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*)

Best First Recursivo

- 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)

Best First Recursivo - Algoritmo

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

si *es_solucion?(nodo)* **entonces**

| solucion.añadir(nodo)

sino

| sucesores \leftarrow generar_sucesores (nodo)

si *sucesores.vacio?()* **entonces**

| nuevo_coste $\leftarrow +\infty$; solucion.vacio()

sino

| fin \leftarrow falso

mientras no fin hacer

| mejor \leftarrow sucesores.mejor_nodo()

si *mejor.coste() > c_alternativo* **entonces**

| fin \leftarrow cierto; solucion.vacio(); nuevo_coste \leftarrow mejor.coste()

sino

| segundo \leftarrow sucesores.segundo_mejor_nodo()

| BFS-recursivo(mejor,min(c_alternativo,segundo.coste()),nuevo_coste, solucion)

si *solucion.vacio?()* **entonces**

| mejor.coste(nuevo_coste)

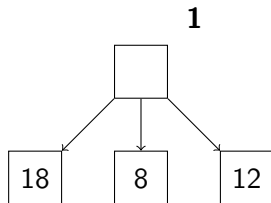
sino

| solucion.añadir(mejor); fin \leftarrow cierto

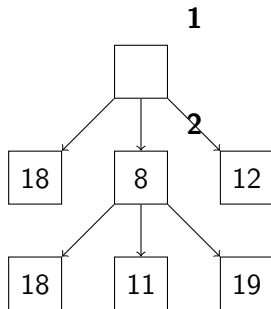
Best First Recursivo - Ejemplo



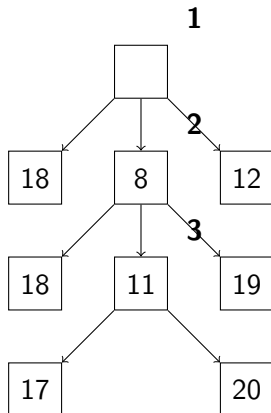
Best First Recursivo - Ejemplo



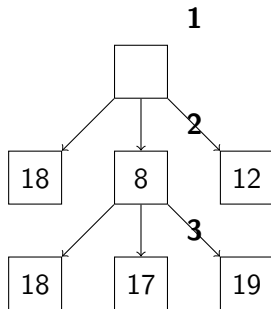
Best First Recursivo - Ejemplo



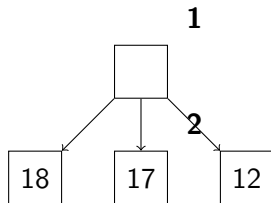
Best First Recursivo - Ejemplo



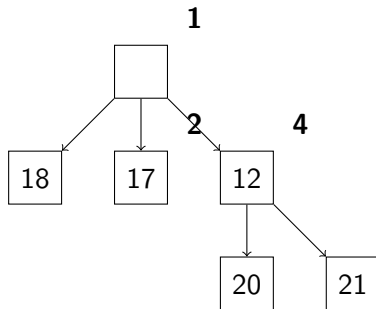
Best First Recursivo - Ejemplo



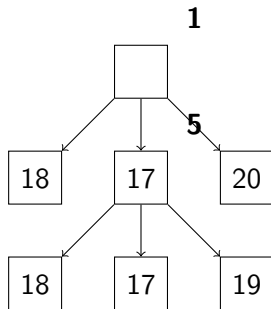
Best First Recursivo - Ejemplo



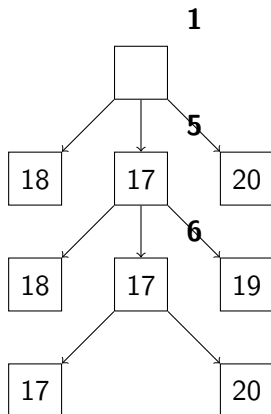
Best First Recursivo - Ejemplo



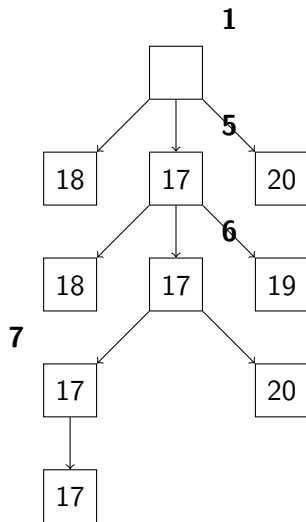
Best First Recursivo - Ejemplo



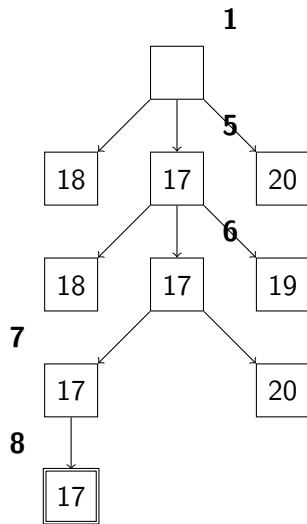
Best First Recursivo - Ejemplo



Best First Recursivo - Ejemplo



Best First Recursivo - Ejemplo



Best First Recursivo

- 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

A* con memoria limitada (MA*)

- 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