

# Inteligencia Artificial

## Búsqueda informada y exploración

Primavera 2007

profesor: Luigi Ceccaroni



# Introducción

- La búsqueda informada utiliza el conocimiento específico del problema.
- Puede encontrar soluciones de una manera más eficiente.
- Una función heurística,  $h(n)$ , mide el coste estimado más barato desde el nodo  $n$  a un nodo objetivo.
- $h(n)$  se utiliza para guiar el proceso haciendo que en cada momento se seleccione el estado o las operaciones más prometedores.

# Importancia del estimador



Estado inicial

H1 = 4

H2 = -28



Estado final

H1 = 8

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

Operaciones:

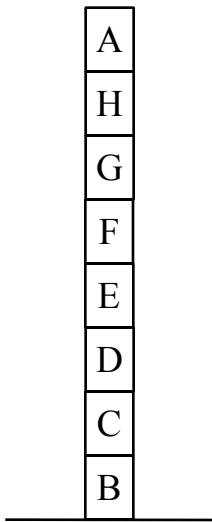
- situar un bloque libre en la mesa
- situar un bloque libre sobre otro bloque libre

Estimador H1:

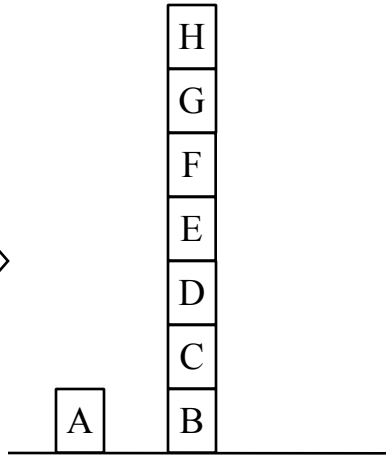
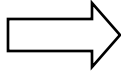
- sumar 1 por cada bloque que esté colocado sobre el bloque que debe
- restar 1 si el bloque no está colocado sobre el que debe

Estimador H2:

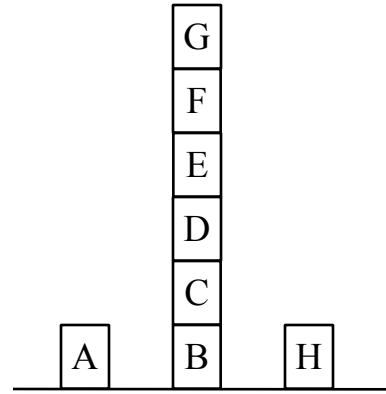
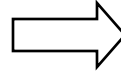
- si la estructura de apoyo es correcta  
sumar 1 por cada bloque de dicha estructura
- si la estructura de apoyo no es correcta  
restar 1 por cada bloque de dicha estructura



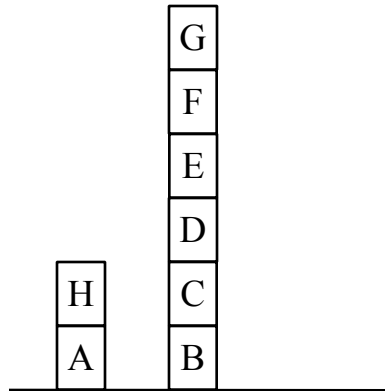
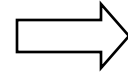
Estado inicial  
H1 = 4  
H2 = -28



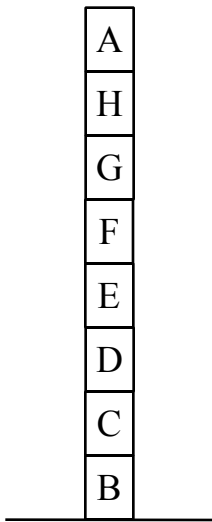
H1 = ?  
H2 = ?



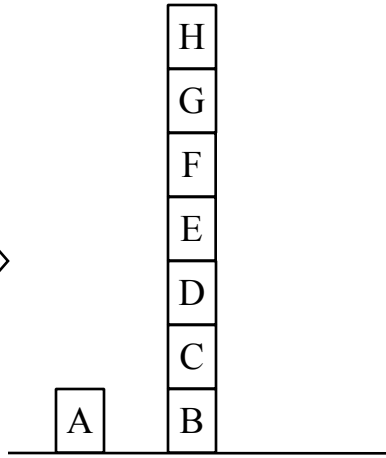
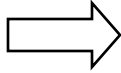
H1 = ?  
H2 = ?



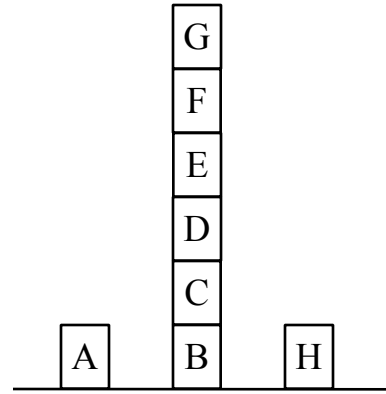
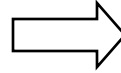
H1 = ?  
H2 = ?



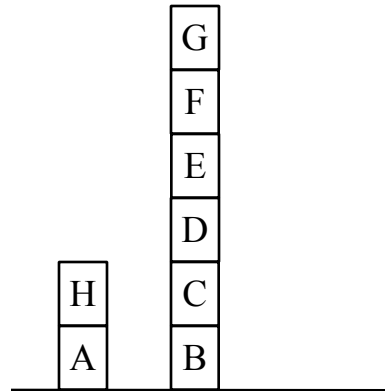
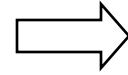
Estado inicial  
 $H1 = 4$   
 $H2 = -28$



$H1 = 6$   
 $H2 = -21$



$H1 = 4$   
 $H2 = -15$



$H1 = 4$   
 $H2 = -16$



# Estrategias de búsqueda informada (heurísticas)

- 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).
- BB (Branch & Bound), Búsqueda primero el mejor
- A, A\*
- A\*PI
- Búsqueda local:
  - ascensión de colinas
  - temple simulado
  - algoritmos genéticos
  - búsqueda en línea

# Búsqueda con ramificación y acotación (Branch & Bound)

- Generaliza BPA y BPP.
- Se guarda para cada estado el coste de llegar desde el estado inicial a dicho estado:  $g(n)$
- 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.



# Búsqueda voraz primero el mejor

- La **búsqueda voraz primero el mejor** expande el nodo más cercano al objetivo.
  - Probablemente conduce rápidamente a una solución.
- Evalúa los nodos utilizando solamente la función heurística, que, en general, se minimiza, porque se refiere a un coste:
  - $f(n) = h(n)$
- La minimización de  $h(n)$  es susceptible de ventajas falsas.

# Búsqueda voraz primero el mejor

**Algoritmo** Greedy Best First

Est\_abiertos.insertar(Estado inicial)

Actual= Est\_abiertos.primerero()

**Mientras no** es\_final?(Actual) **y no** Est\_abiertos.vacia?() **hacer**

Est\_abiertos.borrar\_primerero()

Est\_cerrados.insertar(Actual)

hijos= generar\_sucesores(Actual)

hijos= tratar\_repetidos (Hijos, Est\_cerrados, Est\_abiertos)

Est\_abiertos.insrtar (Hijos)

Actual= Est\_abiertos.primerero()


**fMientras**

**fAlgoritmo**

- La estructura de abiertos es una cola con prioridad.
- La prioridad la marca la función heurística (coste estimado 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.

# Funciones heurísticas

estado inicial

2	8	3
1	6	4
7		5


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

1	2	3
8		4
7	6	5

estado final

# Costes

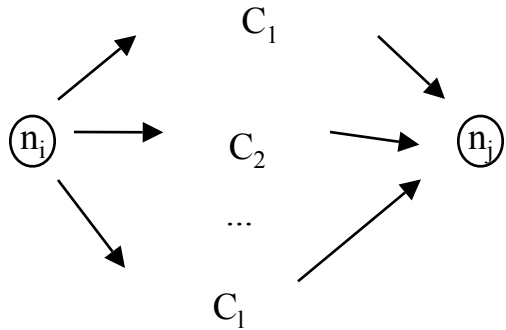


Coste de un arco

$$C(n_i, n_j)$$



Coste de un camino



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  $S = \{s_1, \dots, s_l\}$

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

# Algoritmos A y A\*

La función de evaluación tiene dos componentes:

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

**A**

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

- $f$  es un valor estimado del coste total.
- $h$  (función heurística) es un valor estimado de lo que falta por llegar al (a un) objetivo.
- $g$  es un coste real: lo gastado por el camino más corto conocido hasta el momento.
- Preferencia siempre al nodo con menor  $f$
- En caso de empate, preferencia al nodo con una menor  $h$
- $h$  es una estimación del verdadero coste de alcanzar el objetivo ( $h^*$ ).
- Cuanto más se aproxime  $h$  al verdadero coste mejor.

Si  $h(n)$  nunca sobrestima el coste real, es decir  $\forall n h(n) \leq h^*(n)$  se puede demostrar que el algoritmo encontrará (de haberlo) un camino óptimo.

**A\***

# Algoritmo A\*

1. Put the start node  $S$  on the *nodes* list, called OPEN
2. If OPEN is empty, exit with failure
3. Remove from OPEN and place on CLOSED a node  $n$  for which  $f(n)$  is minimum
4. If  $n$  is a goal node, exit (trace back pointers from  $n$  to  $S$ )
5. Expand  $n$ , generating all its successors and attach to them pointers back to  $n$ . For each successor  $n'$  of  $n$ 
  1. If  $n'$  is not already on OPEN or CLOSED estimate  $h(n')$ ,  $g(n')=g(n)+c(n,n')$ ,  $f(n')=g(n')+h(n')$ , and place it on OPEN.
  2. If  $n'$  is already on OPEN or CLOSED, then check if  $g(n')$  is lower for the new version of  $n'$ . If so, then:
    - (i) Redirect pointers backward from  $n'$  along path yielding lower  $g(n')$ .
    - (ii) Put  $n'$  on OPEN.If  $g(n')$  is not lower for the new version, do nothing.
3. Goto 2

# Algoritmo A\*

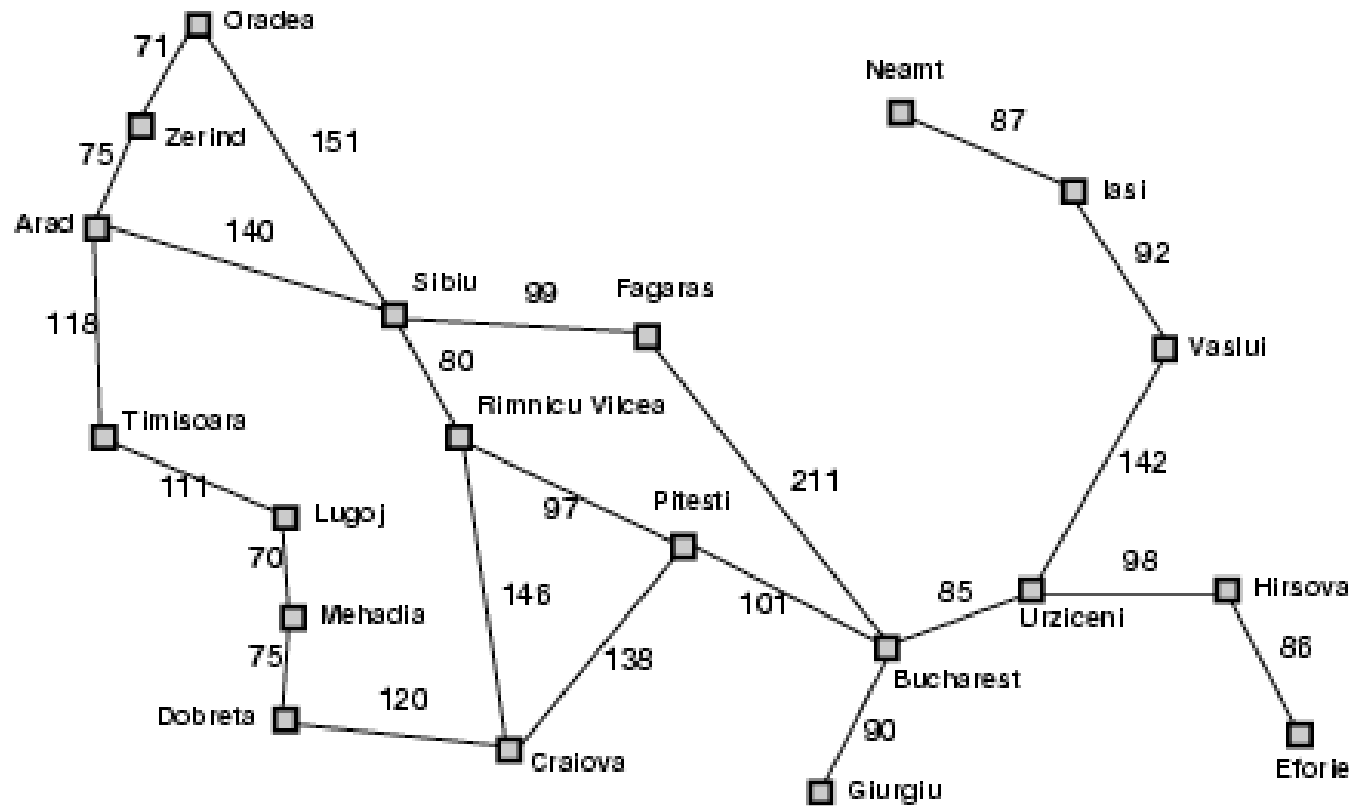
- La estructura de abiertos es una cola con prioridad.
- La prioridad la marca la función de estimación  $f(n)=g(n)+h(n)$ .
- En cada iteración se escoge el mejor camino estimado (el primero de la cola).
- A\* es una instancia de la clase de algoritmos de búsqueda primero el mejor.
- A\* es **completo** cuando el factor de ramificación es finito y cada operador tiene un coste positivo fijo.

# Tratamiento de repetidos

- Si es un repetido que está en la estructura de abiertos:
  - Si su nuevo coste ( $g$ ) es menor sustituimos el coste por el nuevo; esto podrá variar su posición en la estructura de abiertos
  - Si su nuevo coste ( $g$ ) es igual o mayor nos olvidamos del nodo
- Si es un repetido que está en la estructura de cerrados
  - Si su nuevo coste ( $g$ ) 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 nuevo coste ( $g$ ) es mayor o igual nos olvidamos del nodo



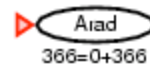
# Ejemplo: encontrar una ruta en Rumanía



Straight-line distance to Bucharest

<b>Arad</b>	366
<b>Bucharest</b>	0
<b>Craiova</b>	160
<b>Dobreta</b>	242
<b>Eforie</b>	161
<b>Fagaras</b>	176
<b>Giurgiu</b>	77
<b>Hirsova</b>	151
<b>Iasi</b>	226
<b>Lugoj</b>	244
<b>Mehadia</b>	241
<b>Neamt</b>	234
<b>Oradea</b>	380
<b>Pitesti</b>	10
<b>Rimnicu Vilcea</b>	193
<b>Sibiu</b>	253
<b>Timisoara</b>	329
<b>Urziceni</b>	80
<b>Vaslui</b>	199
<b>Zerind</b>	374

# Ejemplo de búsqueda A\*



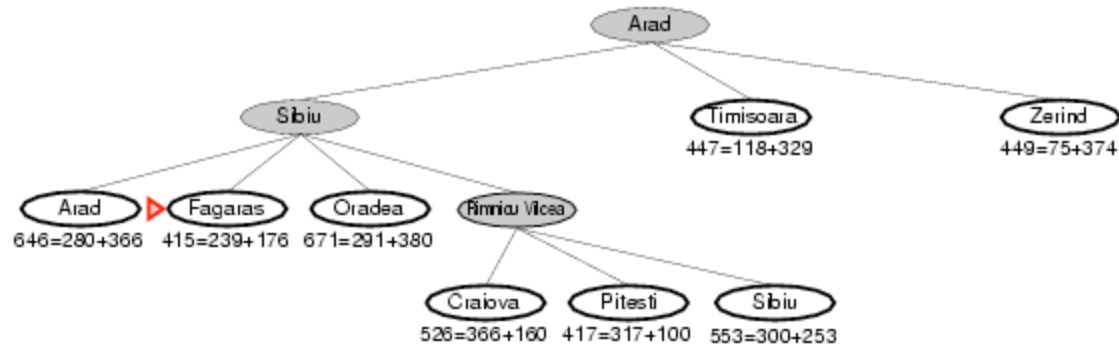
# Ejemplo de búsqueda A\*



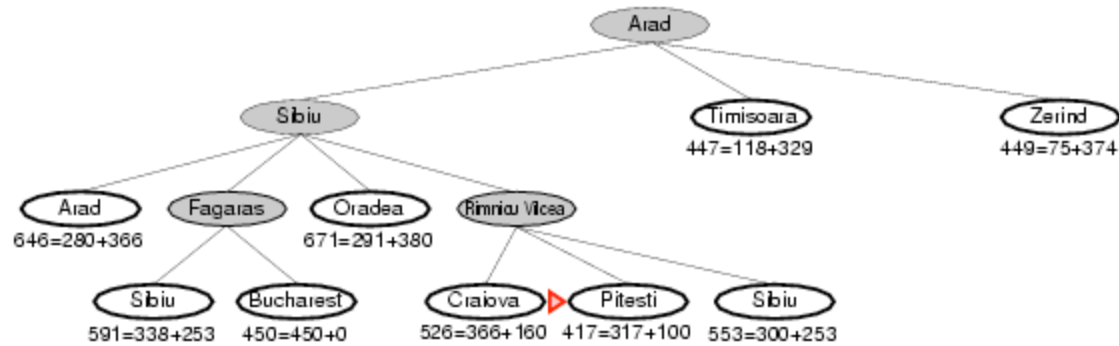
# Ejemplo de búsqueda A\*



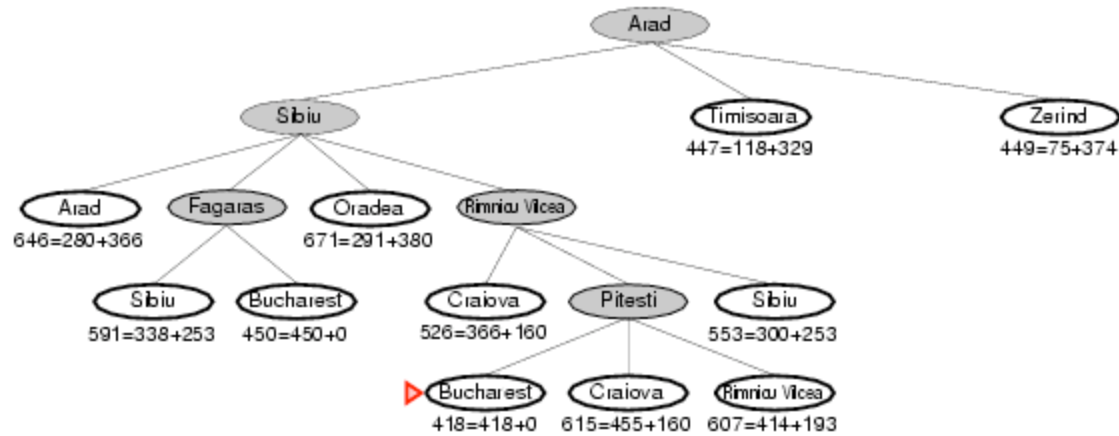
# Ejemplo de búsqueda A\*



# Ejemplo de búsqueda A\*



# Ejemplo de búsqueda A\*



# A\*: admisibilidad

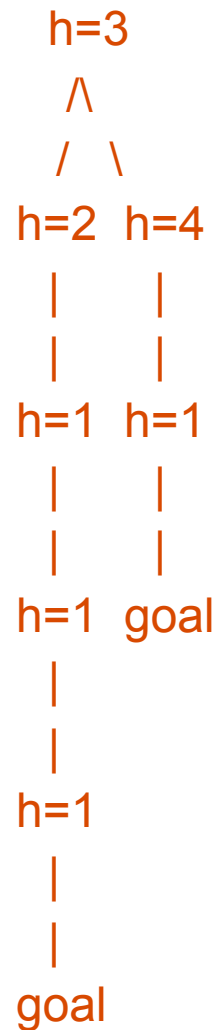
- El algoritmo A, dependiendo de la heurística, encontrará o no una solución óptima.
- Si la heurística es admisible (y se utiliza el algoritmo de búsqueda en árboles), la **optimización** está asegurada.
- Una heurística es admisible si se cumple la siguiente propiedad:

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

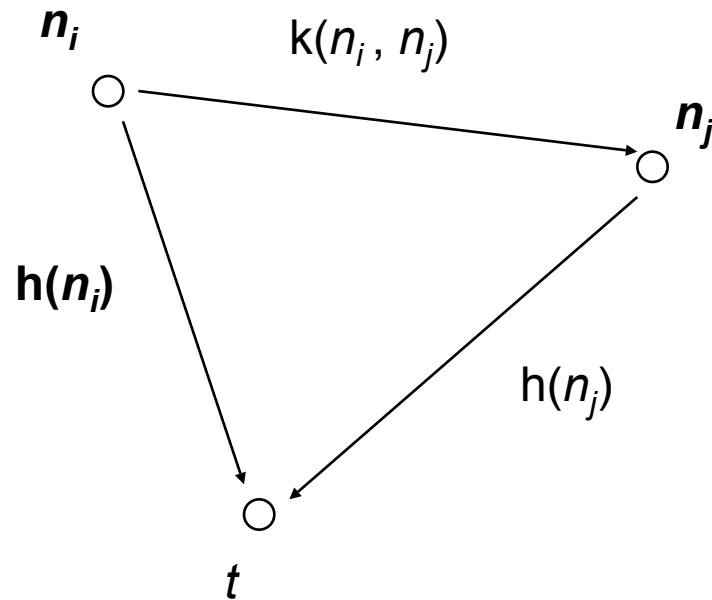
- Por lo tanto,  $h(n)$  ha de ser un estimador optimista, nunca ha de sobreestimar  $h^*(n)$ .
- Usar una heurística admisible garantiza que un nodo en el camino óptimo no pueda parecer tan malo como para no considerarlo nunca.



# Ejemplo: no admisibilidad



# Condición de consistencia (o monotonía)

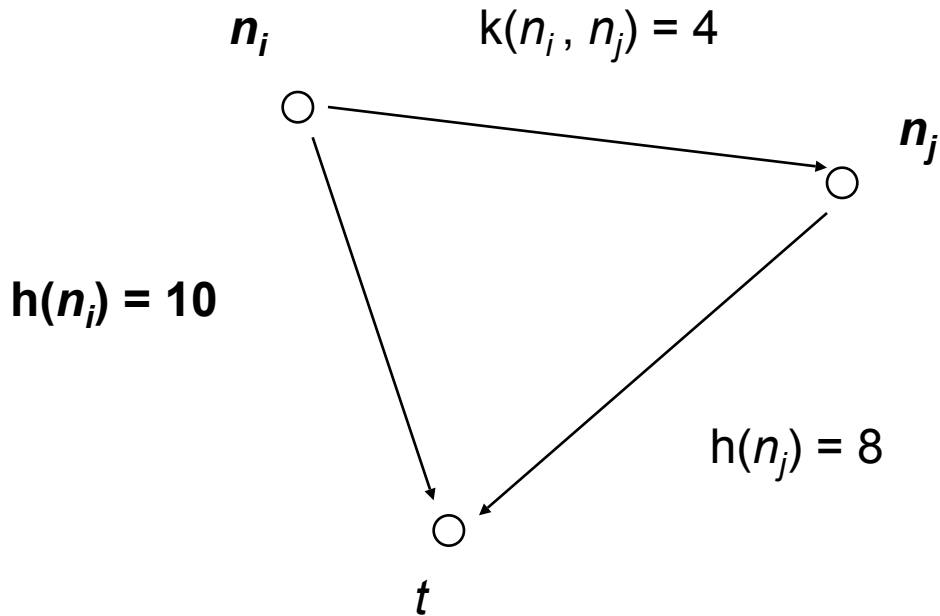


- Esta condición asegura la optimización también cuándo se utiliza el algoritmo de búsqueda en grafos.
- El nodo  $n'$  es sucesor de  $n$
- $h(n)$  = coste estimado desde  $n$  a  $t$
- $h(n)$  cumple la desigualdad triangular:

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

- $h(n_i) - h(n_j) \leq k(n_i, n_j)$
- $h(n)$  consistente  $\Rightarrow$  estimador uniforme del coste

# Condición de Consistencia



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

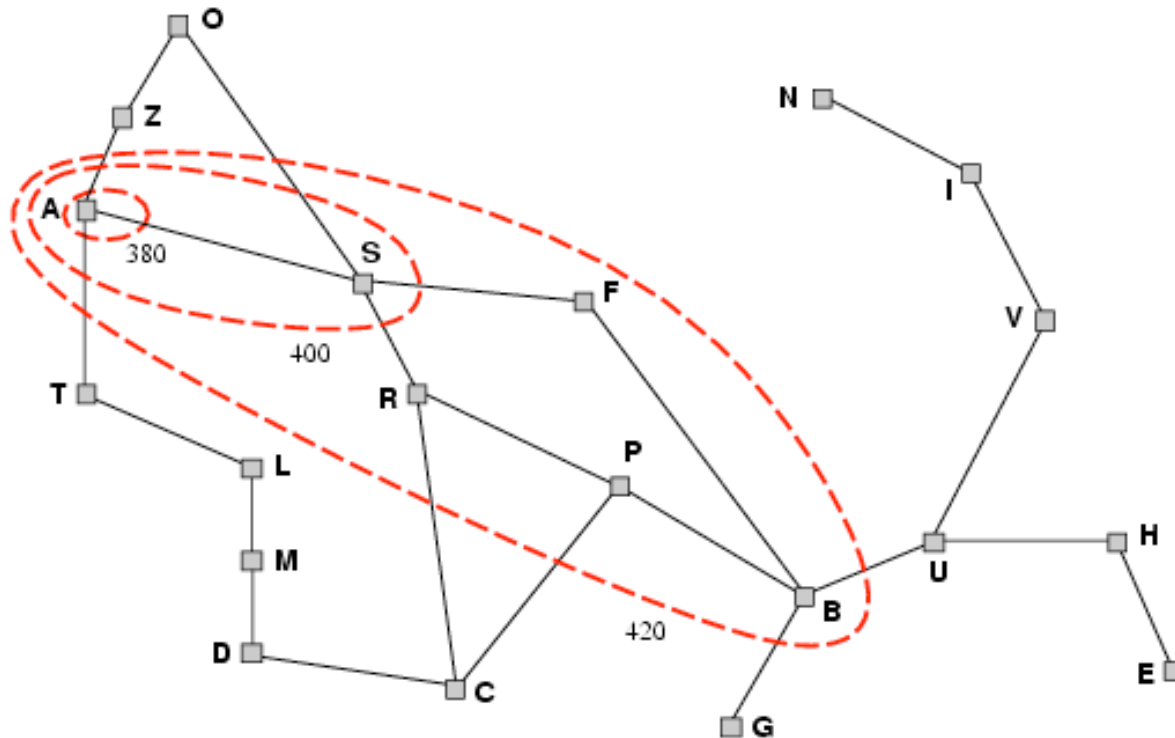
Si  $h(n_j) = 4$ , la estimación de  $h(n)$  no es uniforme y la heurística no es consistente.

# Condición de Consistencia

- Se puede demostrar que toda heurística consistente es también admisible.
- $A^*$  utilizando la búsqueda en grafos es óptimo si la  $h(n)$  es consistente.
- La consistencia es una exigencia más estricta que la admisibilidad.
- Sin embargo, es difícil crear heurísticas que sean admisibles, pero no consistentes.
- Si  $h(n)$  es consistente, entonces los valores de  $f(n)$ , a lo largo de cualquier camino, no disminuyen.

# Optimality of A\*

- A\* expands nodes in order of increasing  $f$  value: si  $h(n)$  es consistente, entonces los valores de  $f(n)$ , a lo largo de cualquier camino, no disminuyen.
- Gradually adds " $f$ -contours" of nodes.
- Contour  $i$  has all nodes with  $f=f_i$ , where  $f_i < f_{i+1}$ .



# Algoritmos más o menos informados

- Dado un problema, existen tantos  $A^*$  para resolverlo como heurísticas podemos definir.

$A_1^*, A_2^*$  admisibles

$\forall n \neq \text{final}: 0 \leq h_2(n) < h_1(n) \leq h^*(n)$

$\Rightarrow$

$A_1^*$  más informado que  $A_2^*$

# Algoritmos más informados

$A_1^*$  más informado que  $A_2^*$



$n$  expandido por  $A_1^* \Rightarrow n$  expandido por  $A_2^*$



$A_1^*$  expande menor número de nodos que  $A_2^*$



En principio interesan algoritmos más informados

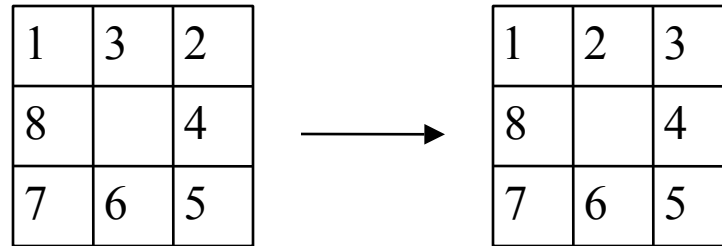
# Algoritmos más informados

- Compromiso entre:
  - Tiempo de cálculo  
 $h_1(n)$  requiere más tiempo de cálculo que  $h_2(n)$  !
  - Número de reexpansiones  
 $A_1^*$  puede que re-expanda más nodos que  $A_2^*$  !
    - Si  $A_1^*$  es consistente seguro que no
    - Si se trabaja con árboles (y no grafos) seguro que no
- Pérdida de admisibilidad
  - Puede interesar trabajar con heurísticas no admisibles para ganar rapidez.



# Rompecabezas de 8 piezas

(coste operaciones = 1)



$h_0(n)=0$

anchura prioritaria,  $A_0^*$  admisible,

muchas generaciones y expansiones

$h_1(n)=\#$  piezas mal colocadas

$A_1^*$  admisible,

$A_1^*$  mas informado que  $A_0^*$

$$h_2(n) = \sum_{i \in [1,8]} d_i$$

$d_i \equiv$  distancia entre posición de la pieza  $i$  y su posición final (suponiendo camino libre)

*distancia*  $\equiv$  número mínimo de movimientos para llevar la pieza de una posición a otra

$A_2^*$  admisible.

Estadísticamente se comprueba que  $A_2^*$  es mejor que  $A_1^*$ , pero no se puede decir que sea más informado.

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

$$S(n) = \sum_{i \in [1,8]} s_i$$

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

1		3
8	2	4
7	6	5

$$h_3(n) = 1 + 3(2 + 1) = 10$$

$$h^*(n) = 1$$

}  $A_3^*$  no admisible

$A_3^*$  no se puede comparar con  $A_1^*$  o  $A_2^*$  pero va más rápido (aunque la  $h$  a calcular requiera más tiempo).

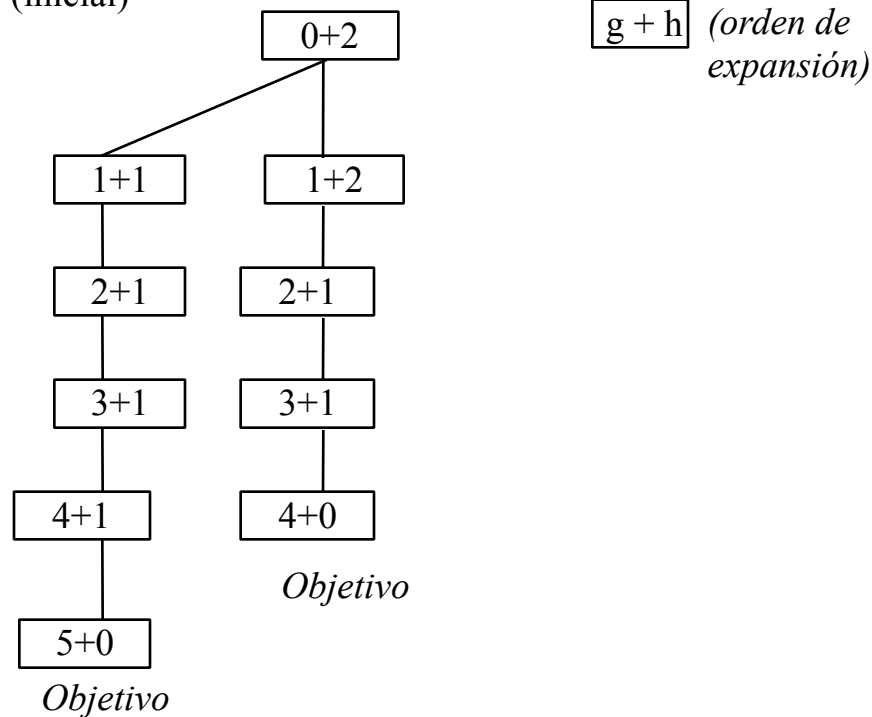
# Óptimo con limitación de memoria

- 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  $A^*$ .
- Existen algoritmos que permiten obtener el óptimo limitando la memoria usada:
  - $A^*PI$
  - Primero el mejor recursivo
  - $A^*$  con limitación de memoria ( $MA^*$ )

# A\*PI

- A\* en profundidad iterativa es similar a PI (es decir iteración de búsqueda en profundidad con un límite en la búsqueda).
  - En PI el límite lo daba una cota máxima en la profundidad.
  - En A\*PI el límite lo da una cota máxima sobre el valor de la función  $f$ .
- ¡Ojo! La búsqueda es una BPP normal y corriente, el heurístico  $f$  sólo se utiliza para podar.

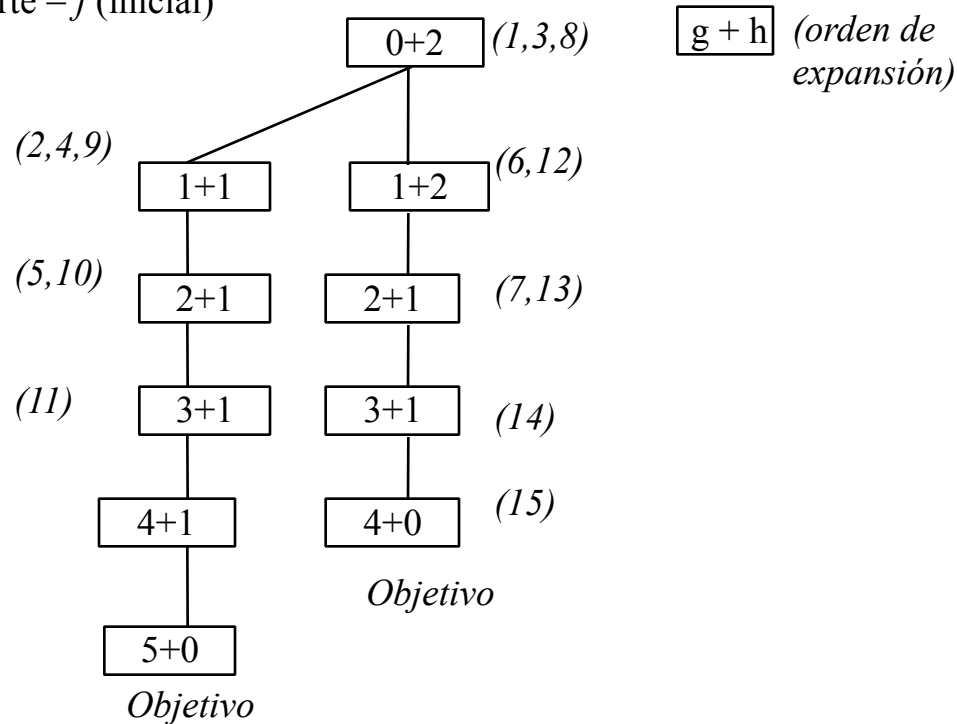
Empezamos con corte =  $f$  (inicial)



# A\*PI

- A\* en profundidad iterativa es similar a PI (es decir iteración de búsqueda en profundidad con un límite en la búsqueda).
  - En PI el límite lo daba una cota máxima en la profundidad.
  - En A\*PI el límite lo da una cota máxima sobre el valor de la función  $f$ .
- ¡Ojo! La búsqueda es una BPP normal y corriente, el heurístico  $f$  sólo se utiliza para podar.

Empezamos con corte =  $f$ (inicial)



# Algoritmo A\*PI

*Algoritmo* IDA\*

prof=f(Estado\_inicial)

**Mientras no** es\_final?(Actual) **hacer**

Est\_abiertos.insertar(Estado\_inicial)

Actual= Est\_abiertos.primerero()

**Mientras no** es\_final?(Actual) **y no** Est\_abiertos.vacía?() **hacer**

Est\_abiertos.borrar\_primerero()

Est\_cerrados.insertar(Actual)

Hijos= generar\_sucesores(Actual, prof)

Hijos= tratar\_repetidos(Hijos, Est\_cerrados, Est\_abiertos)

Est\_abiertos.insertar(Hijos)

Actual= Est\_abiertos.primerero()

**fMientras**

prof=prof+1

Est\_abiertos.inicializa()

**fMientras**

**fAlgoritmo**

- La función *generar\_sucesores* sólo **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).
- Tener en cuenta que si se tratan los nodos repetidos el ahorro en espacio es nulo.
- **Sólo se guarda en memoria el camino (la rama del árbol) actual.**

# Otros algoritmos con limitación de memoria

- Las reexpansiones de A\*PI pueden suponer un elevado coste temporal.
- Hay algoritmos que, en general, re-expanden menos nodos.
- Su funcionamiento se basa en eliminar los nodos menos prometedores y guardar información que permita re-expandirlos (si fuera necesario).
- Ejemplos:
  - Primero el mejor recursivo
  - A\* con limitación de memoria (MA\*)



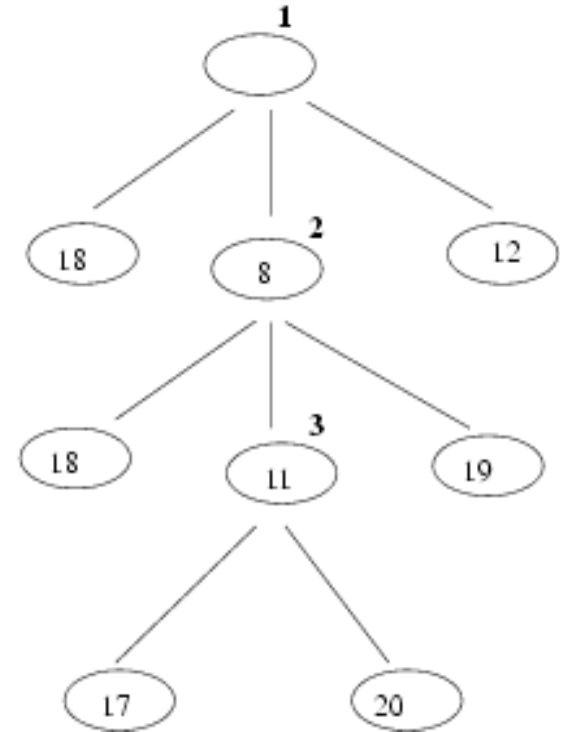
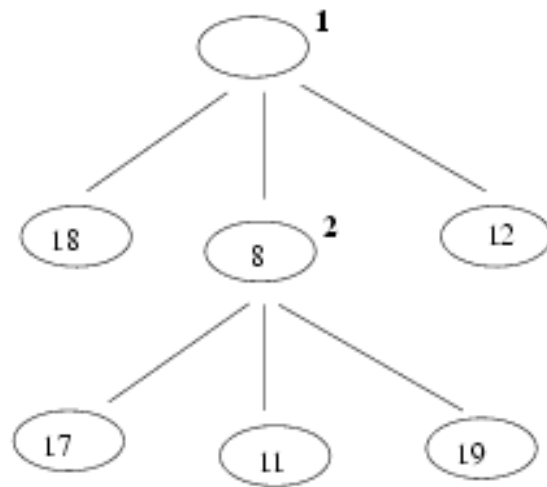
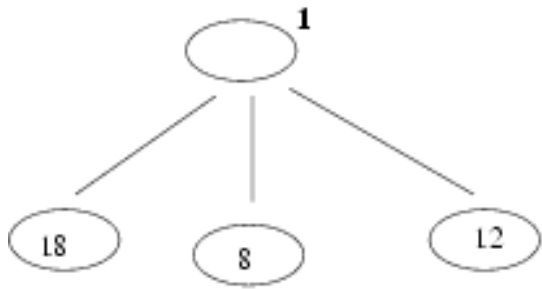
# Primero el mejor recursivo

- Es una implementación recursiva de *Primero el mejor* 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 re-expandida si su coste vuelve a ser el mejor.
  - Se regenera toda la rama olvidada.
- Por lo general re-expande menos nodos que A\*PI.
- Al no poder controlar los repetidos su coste en tiempo puede elevarse si hay ciclos.

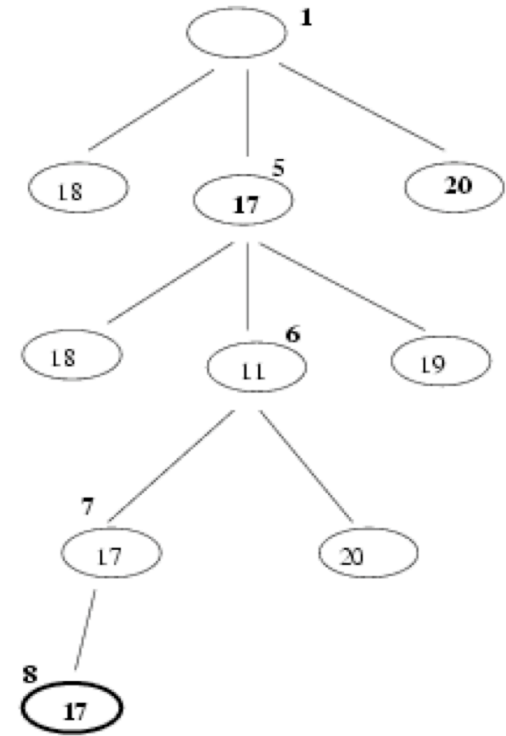
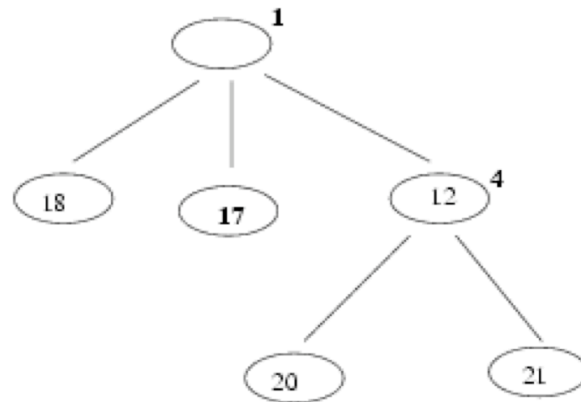
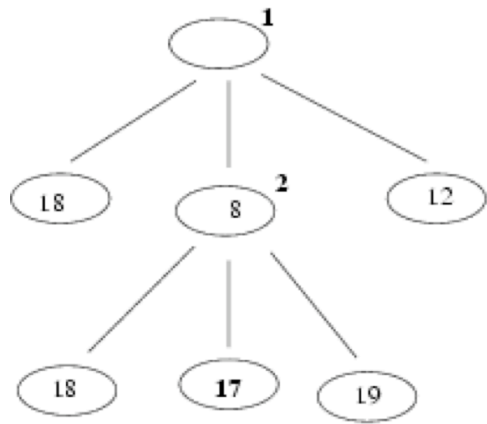
# Primero el mejor recursivo: algoritmo

```
Accion BPM-recursivo (nodo, c_alternativo, ref nuevo_coste, ref solucion)
  si es_solucion?(nodo) entonces solucion.añadir(nodo)
  si no sucesores= genera_sucesores(nodo)
    si sucesores.vacio?() entonces
      nuevo_coste=+∞; solucion.vacio()
    si no fin=falso
      mientras no fin hacer
        mejor= sucesores.mejor_nodo()
        si mejor.coste() > c_alternativo entonces
          fin=cierto; solucion.vacio(); nuevo_coste=mejor.coste()
        si no segundo=sucesores.segundo_mejor_nodo()
          BPM-recursivo(mejor,min(c_alternativo, segundo.coste()), nuevo_coste, solucion)
          si solucion.vacio?() entonces
            mejor.coste(nuevo_coste)
          si no solucion.añadir(mejor); fin=cierto
        fsi
      fsi
    fmientras
  fsi
faccion
```

# Primero el mejor recursivo - Ejemplo



# Primero el mejor recursivo - Ejemplo



# A\* con memoria limitada (MA\*)

- Impone un límite de memoria (número de nodos que se pueden almacenar).
- Se explora usando A\* y se almacenan nodos mientras quepan en la memoria.
- Cuando no quepan se eliminan los peores guardando el mejor coste de los descendientes olvidados.
- Se re-expande si los nodos olvidados son mejores.
- El algoritmo es completo si el camino solución cabe en memoria.